Uma Estratégia para
Testes de Regressão
utilizando Classes Testáveis

Vanessa Gindri Vieira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uma Estratégia para Testes de Regressão utilizando Classes Testáveis

Vanessa Gindri Vieira[1]

Outubro de 2004

Banca Examinadora:

- Profa. Dra. Eliane Martins (Orientadora)
- Profa. Dra. Cecília Mary Fischer Rubira
  IC - UNICAMP
- Prof. Dr. Marcos Lordello Chaim
  CNPTIA - EMBRAPA
- Profa. Dra. Maria Beatriz Felgar de Toledo (Suplente)
  IC - UNICAMP

[1]**Parcialmente financiado por CNPq**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Vieira, Vanessa Gindri

V673e Uma estratégia para testes de regressão utilizando classes testáveis / Vanessa Gindri Vieira -- Campinas, [S.P. :s.n.], 2004.

Orientadora: Eliane Martins

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

1. Engenharia de Software. 2. Software – Testes. 3. Software – Validação. 4. Software (Manutenção). I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 22 de outubro de 2004, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Marcos Lordello Chaim
EMBRAPA

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Profa. Dra. Eliane Martins
IC - UNICAMP

Uma Estratégia para Testes de Regressão utilizando Classes Testáveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Vanessa Gindri Vieira e aprovada pela Banca Examinadora.

Campinas, 22 de outubro de 2004.

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.

Dedico este trabalho à memória de meu pai Acioli Viçosi Vieira.

Agradecimentos

Meu primeiro agradecimento é a Deus pela força concedida todos esses anos e especialmente durante a elaboração deste trabalho.

À minha orientadora Eliane Martins pelos conhecimentos compartilhados, pelo incentivo para a conclusão desta dissertação e pela ajuda nos momentos mais difíceis, sem os quais este trabalho não se concretizaria. Obrigada pela confiança.

Ao meu marido, Vinícius Garcia, pelo apoio incondicional, pelo companherismo, pelo amor, pela dedicação, pela colaboração e incentivo para chegar a esse dia.

À minha família, que embora fisicamente distante, sempre encontrou forças para incentivar minha iniciativa. À minha mãe Neuza, que sempre apoiou-me nessa jornada e quando a saudade batia não media esforços para viajar até Campinas. Aos meus irmãos, Michele e Vinícius, pelo carinho e admiração. Obrigada por vocês existirem.

Aos familiares pelo carinho e preocupação, em especial Antônio, Maria Helena, Rômulo, Laura, Rafael, Henrique e Victor.

Aos colegas e amigos do CPqD/SAGRE que sempre compreenderam meus momentos de aflição e me apoiaram. Especialmente: Amanda, Jane, Luciano, Rodrigo, Rosamaria, Renatinho e Mônica.

Ao Luciano pelas sugestões, pela troca de experiências e pela ajuda com a ConCAT.

À Maria da Glória, uma profissional brilhante que devolveu-me a auto-estima e confiança.

Aos amigos, muitos também vindos do Rio Grande do Sul, que sempre estiveram por perto para lembrar-me da importância da convivência social e para ajudar a diminuir a saudade do sul.

Resumo

Uma classe reutilizável precisa ser testável, já que a mesma pode ser testada várias vezes: quando é modificada, quando sua superclasse é modificada, quando suas clientes e/ou servidoras são modificadas. Daí a importância de que essa classe seja fácil de testar, ou seja, testável. Com a utilização de classes testáveis ocorre um aumento da testabilidade do sistema que as contém. Em trabalho anterior foi definida uma classe testável que inclui, além da própria classe, um modelo representando o comportamento da classe, bem como mecanismos embutidos de testes, ou BIT (*Built-in Test*). A atividade de teste de regressão envolve o teste de modificações do sistema para garantir que o sistema não regrediu, ou seja, que as funcionalidades que executavam corretamente numa versão anterior não foram indesejavelmente afetadas pelas modificações. Esse trabalho teve por objetivo responder à seguinte pergunta: como utilizar informações de testes contidas em uma classe testável nos testes de regressão? Para respondê-la foi necessário definir: (i) uma forma de seleção de testes de regressão - nesse trabalho nós propomos uma técnica de seleção baseada no modelo de comportamento da classe, apesar da maioria das técnicas existentes serem baseadas no código; e (ii) uma forma de gerar testes para novas características resultantes da modificação. A técnica proposta é aplicável tanto no contexto da classe base quanto das classes derivadas. Além de não precisar do código fonte, o que a torna útil para testes de componentes nos quais o código fonte não está disponível, a técnica também pode ser totalmente automatizada.

Abstract

A reusable class has to be testable since it should be tested many times: when a class is changed, when its superclass is changed, when its client and/or server classes are changed. Therefore, it is important that this class be easy to test, that is, be testable. There is an increase on the testability of the system when testable classes are used. A previous research elaborated a testable class, which includes the class implementation, a model to represent its behaviour, as well as built-in test (BIT) mechanisms. Regression testing activity involves testing the modified program to ensure that new features do not regress to the existing features, that is, regression testing is applied to the modified software to provide confidence that the unchanged parts have not been adversely affected by the modification. This work intends to answer the following question: how to use test information contained on the testable class to do regression testing? The answer involves some definitions: (i) a strategy of regression test selection - in this work we propose a regression test selection technique based on the class behaviour model, despite the fact that the majority of existing techniques are code-based; and (ii) a way to generate tests to new features resulted from the class modifications. This technique is applied not only to base class context but also to derived classes. Besides the fact that the source code is not needed in this technique, which makes it useful to component testing in which the source code is not available, the technique can also be totally automated.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Contexto e Motivação

A atividade de teste de regressão vem crescendo dentro dos projetos de grandes sistemas comerciais ou acadêmicos. Essa importante atividade para garantia de qualidade de software passou a fazer parte das preocupações, inclusive dos clientes, para que seja garantida que as funcionalidades que já estavam em uso continuem a executar corretamente mesmo depois da mudança de versão do sistema.

Muitas técnicas de teste de regressão são propostas na literatura para explorar os quatro problemas levantados com o uso de teste de regressão: 1- Como automatizar a identificação dos componentes afetados pelas modificações de um componente; 2- Qual estratégia deve ser usada para retestar estes componentes afetados; 3- Qual o critério de cobertura para retestar estes componentes; 4- Como selecionar, reusar e modificar os casos de teste existentes e gerar novos testes.

Atualmente, existem vários trabalhos que visam tratar os problemas descritos (alguns são citados no capítulo 4). Porém a grande maioria dos trabalhos propõe técnicas baseadas em código, o que aumenta as chances de serem seguras. No entanto, técnicas baseadas em código são dependentes de linguagem e apresentam problemas de escalabilidade. Uma vez que para o uso dessas técnicas é necessário conhecer o código e entender seu funcionamento, muitas vezes a utilização dessas técnicas torna-se inviável pelo grande volume de informações necessárias.

As técnicas seletivas para teste de regressão visam fornecer economia de tempo e esforço de teste, já que buscam selecionar testes apenas para as funcionalidades modificadas ou afetadas pelas modificações.

Mais recentemente, técnicas seletivas de teste de regressão baseadas em especificação vem sendo propostas para suprir a atividade de testes em fase de manutenção de sistemas ou componentes em que o usuário não tem acesso aos testes realizados anteriormente, e

muitas vezes nem ao código do sistema. Dessa forma, é necessário saber qual caso de teste verifica um dado requisito baseado no comportamento e especificação do sistema.

Neste contexto, este trabalho propõe uma técnica seletiva de teste de regressão baseada em especificação que adapta o padrão de reteste baseado em segmentos modificados (Binder 2000a) e o combina com a Técnica Incremental Hierárquica (Harrold, McGregor & Fitzpatrick 1992) para prover reuso de testes.

Com a busca pelos segmentos modificados de cada classe do sistema sob teste são selecionados os casos de testes a serem reaplicados na classe e em suas subclasses a fim de realizar um teste de regressão mais seguro. Para gerar o conjunto inicial de testes para cada classe é considerada a técnica que permite o reuso dos testes possibilitando a classificação das classes do sistema para facilitar a seleção dos testes.

No trabalho de Toyota (2000) foi definido o conceito de classe testável para aumentar a testabilidade de um sistema e ajudar nos testes da classe quando ela é reutilizada. São descritas as características para compor essa classe e como devem ser utilizadas para testes. Além disso, foi desenvolvido um protótipo denominado ConCAT para auxiliar na obtenção do conjunto de testes.

A técnica seletiva proposta faz uso das vantagens das classes testáveis inseridas no sistema para diminuir o custo de teste quando essa classe é reusada.

1.2 Objetivos

O objetivo principal desse trabalho é propor uma estratégia seletiva para teste de regressão baseada em especificação e em classes testáveis, sendo que essa estratégia também considera a seleção de testes para subclasses.

Outro objetivo é analisar as dificuldades e as vantagens da utilização da estratégia proposta. Além de realizar a avaliação da estratégia conforme *framework* para comparação de estratégias seletivas para testes de regressão proposto em Harrold & Rothermel (1994).

1.3 Organização da Dissertação

Os demais capítulos dessa dissertação estão organizados conforme descrito a seguir:

Capítulo 2 apresenta uma visão geral dos conceitos básicos relacionados a teste de sistemas convencionais e sistemas orientados a objetos. São descritas a terminologia adotada nessa dissertação, as fases de teste e particularidades dos testes em sistemas orientados a objetos.

Capítulo 3 apresenta o conceito de Teste de Regressão, as estratégias possíveis para a sua realização, um *Framework* para comparação e avaliação das técnicas seletivas, informações sobre a manutenção do conjunto de testes. Também são descritas as características de teste de regressão para sistemas OO, algumas abordagens para teste e algumas limitações do teste de regressão.

Capítulo 4 traz alguns trabalhos relacionados apresentados na literatura que tratam de teste de regressão seletivo.

Capítulo 5 descreve o conceito de testabilidade, os principais aspectos que influenciam a testabilidade de um sistema, as características de testabilidade em sistemas OO e a utilização de projeto visando a testabilidade de software. Além disso, é apresentada a técnica incremental hierárquica que permite o reuso de testes e a automatização dos testes com o uso da ConCAT.

Capítulo 6 apresenta uma estratégia seletiva para teste de regressão, desde a construção da classe testável até os passos para a aplicação da estratégia. Para facilitar o entendimento da estratégia é utilizado um pequeno exemplo. Além disso, é mostrado como realizar o teste de regressão no contexto da subclasse, e depois uma avaliação da estratégia utilizando o *framework* descrito anteriormente.

Capítulo 7 descreve o estudo empírico realizado com uma aplicação real. Sendo realizada uma descrição das classes sob teste, a preparação necessária para os testes de regressão e o procedimento adotado. Por último é realizada uma análise dos resultados obtidos com a aplicação da estratégia no aplicativo real.

Capítulo 8 apresenta as conclusões deste trabalho, as principais contribuições e sugestões de trabalhos futuros.

Capítulo 2

Teste de Software

Teste é um processo formal que compreende, segundo (Beizer 1990): preparação das entradas de dados, com os resultados esperados; documentação dos testes; execução dos comandos e observação dos resultados encontrados.

O propósito dos testes é mostrar que um sistema de software tem falhas. Para isso, deve-se começar os testes com condições conhecidas, usando rotinas pré-definidas, e com resultados previamente conhecidos. Porém, mesmo nessa situação apenas é possível afirmar, como resultado dos testes, que o programa passou ou não nesses testes. Por isso, os testes devem ser planejados, projetados e escalonados para que demonstrem as falhas ou aparente corretude do sistema em teste.

A realização dos testes é importante para complementar outras formas de verificação e validação, pois trata-se da única que permite exercitar o comportamento operacional do programa. Testes também podem ser úteis para a obtenção de medida de qualidade do software, por exemplo: nível de confiabilidade e desempenho.

Nesse capítulo são apresentados os conceitos de teste em sistemas convencionais e sistemas orientados a objetos. Alguns pontos relevantes que diferem a atividade de teste dentre esses sistemas também são abordados. Inicialmente são definidos os termos utilizados que ainda não são consenso na língua portuguesa. A seguir, uma breve descrição das fases do processo de teste. Depois particularidades dos testes em sistemas OO.

2.1 Terminologia

A terminologia utilizada no texto segue o padrão IEEE 610.12/1990 (IEEE 1990). Já em português, os termos utilizados estão de acordo com as definições introduzidas em (Leite & Loques 1987). Um engano cometido pelo desenvolvedor durante a implementação do sistema é considerada uma falha (*fault*). Quando a instrução com a falha é ativada é gerado um erro (*error*) que pode levar o software a produzir um resultado incorreto, isto

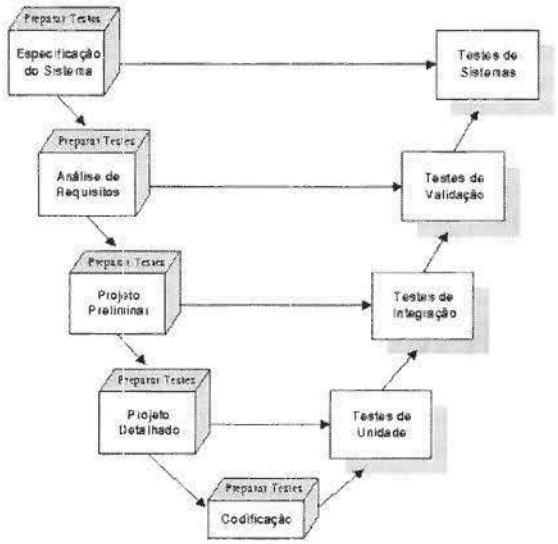

Figura 2.1: Fases de teste no processo de desenvolvimento.

é, apresentar um defeito (*failure*). Diz-se que um software tem um defeito quando o serviço que ele fornece a outros sistemas ou ao usuário viola sua especificação.

2.2 Fases de Teste

A atividade de teste deve acompanhar todas as fases de desenvolvimento de software, e também está dividida em fases, sendo que os casos de teste para cada fase de teste são preparados em uma determinada etapa do desenvolvimento. O diagrama representado na figura 2.1 apresenta a relação entre as fases de desenvolvimento e de teste.

Cada fase de teste tem um foco diferente e é aplicada numa fase distinta do desenvolvimento de software. A seguir serão descritas as estratégias consideradas para sistemas tradicionais: Testes de Unidades, Testes de Integração, Testes de Validação, Testes de Sistemas e Testes de Regressão.

Testes de Unidades

Os testes de unidades têm como foco a verificação de cada unidade ou módulo do sistema, e visam abordar os testes: de interface, de estruturas locais de dados, de condições limites, de caminhos independentes e caminhos de tratamento de erros (por exemplo: processamento de condições de exceção, descrição do erro deve fornecer informações suficientes para ajudar na localização da causa do erro).

Como um módulo não é um programa independente, para cada unidade de teste é necessário desenvolver *drivers* e/ou *stubs*. Em muitas aplicações, um driver nada mais é do que uma espécie de "programa principal"que aceita dados de casos de testes, passando cada dado para um módulo (para testá-lo) e mostrando os resultados relevantes (Pressman 1997a).

Já os *stubs* são componentes que substituem os módulos que interagem com a unidade sob teste, portanto possuem a mesma interface dos módulos que substituem. A implementação de um *stub* pode variar de um processamento simples, como a exibição de uma mensagem de rastreamento, a um processamento mais complexo como o mapeamento de um parâmetro de entrada a um valor de retorno associado.

Testes de Integração

Nos testes de integração o principal objetivo é verificar a existência de falhas na interação entre as unidades ao formarem módulos ou subsistemas.

Testes de Validação

Visam verificar a satisfação dos requisitos que foram definidos na fase de Análise do software.

Testes de Sistema

São desempenhados fora do ambiente de desenvolvimento, o mais próximo possível do ambiente (software e hardware) em que o sistema será executado. O principal objetivo é testar comportamentos que são mostrados apenas testando o sistema como um todo. Inclui testes de desempenho, de segurança, de estresse e de recuperação (Pressman 1997a).

Teste de Regressão

Na fase dos testes de integração normalmente é utilizada a abordagem de Testes de Regressão para complementar os testes realizados nos módulos. Eles também são amplamente utilizados durante a manutenção. Os testes de regressão possuem o objetivo de avaliar se as novas implementações não interferiram de forma equivocada nas funcionalidades que já existiam no sistema, isto é, verificar se as alterações não provocaram erros no sistema. Os testes de regressão serão mais explorados no capítulo 3.

2.3 Métodos de Desenvolvimento de Teste

Segundo Pressman (1997a), nas últimas décadas uma rica variedade de métodos de projeto de casos de testes tem evoluído para software. Estes métodos fornecem aos desenvolvedores uma abordagem sistemática para testes. Também fornecem um mecanismo que pode ajudar a garantir a completeza dos testes e fornecer uma alta probabilidade para a descoberta de erros no software.

Para a realização de teste de qualquer produto de software um dos seguintes modos deve ser considerado (Pressman 1997a): (1) conhecendo a função específica que um produto foi projetado para desempenhar, os testes podem ser conduzidos para demonstrar que cada função está plenamente operacional, ao mesmo tempo que a busca por erros é feita em cada função; (2) conhecendo o trabalho interno de um produto, os testes podem ser conduzidos para garantir que as operações internas desempenham de acordo com o especificado e todos os componentes internos foram adequadamente exercitados. A primeira abordagem de teste é denominada **teste caixa-preta** e a segunda, **teste caixa-branca**.

O teste caixa-preta refere-se aos testes conduzidos à interface do software. Embora eles sejam projetados para descobrir erros, os testes caixa-preeta são usados para demonstrar que as funções do software estão operacionais; que as entradas são devidamente aceitas e saídas foram corretamente produzidas; e que a integridade da informação externa é mantida.

Por outro lado, o teste caixa-branca está preocupado em examinar os detalhes procedurais. Este método usa a estrutura de controle do projeto procedural para derivar casos de testes que: garantam que todos os caminhos independentes num módulo sejam exercitados pelo menos uma vez; exercitam todas as decisões lógicas tanto verdadeiras quanto falsas; executam as fronteiras de todos os *loops* e seus limites operacionais; e exercitam estruturas de dados internas para validá-las.

2.4 Teste de Software Orientado a Objetos

Os conceitos do paradigma de orientação a objetos não serão explicitados nesse trabalho, devido ao foco do trabalho estar voltado aos testes desses sistemas. Para a obtenção de detalhes conceituais de software orientado a objetos as seguintes referências podem ser consultadas (Pressman 1997b, Buzato & Rubira 1998).

Os testes de software orientados a objetos mantém o mesmo objetivo fundamental de teste: encontrar o máximo de erros com o menor custo. O que muda é a estratégia e a tática de teste (Pressman 1997a).

Para a avaliação da qualidade dos sistemas orientados a objetos, os testes tornam-se mais importantes do que em sistemas convencionais, já que nesses a utilização de

verificação estática é mais efetiva e eficiente para remover falhas do que os testes. No entanto, em implementações orientadas a objetos, segundo Binder (2001), o código pode ser considerado mais difícil de ler porque o controle é distribuído, por pelo menos três razões: (1) o problema da fragmentação de funcionalidades que podem resultar de herança; (2) ligação dinâmica, e; (3) estratégias de controle cooperativa onde o fluxo de controle e o controle de estado são distribuídos entre todas as classes. Com esses obstáculos de compreensão, técnicas estáticas serão menos efetivas, requerendo mais testes para adquirir o mesmo nível de qualidade.

Ao mesmo tempo em que as linguagens orientadas a objetos reduzem a ocorrência de alguns tipos de erros, elas aumentam as possibilidades de outros, como pode ser verificado no decorrer desta seção.

2.4.1 Conceitos e Características

Nesta seção serão descritos os aspectos de orientação a objetos que afetam a realização dos testes, sendo que as dificuldades levantadas para os testes foram baseadas em (Binder 2000a) e (Martins 1999).

Algumas características essenciais de linguagens orientadas a objetos apresentam novos riscos de falhas em relação às linguagens convencionais:

- ligação dinâmica e estruturas complexas de herança geram muitas chances de ocorrência de falhas devido à ligações não antecipadas e má interpretação do uso correto;
- erros de programação de interface são a principal causa de falhas em linguagens procedimentais. Programas orientados a objetos tipicamente têm muitos componentes pequenos e conseqüentemente mais interfaces. Portanto, erros de interface são mais prováveis;
- objetos preservam o estado, mas o estado de controle (a seqüência acessível de eventos) é tipicamente distribuído sobre o programa inteiro. Erros de estado de controle ocorrerão possivelmente com mais freqüência.

A seguir são descritos os conceitos e características de orientação a objetos, bem como suas implicações na atividade de testes.

Encapsulamento

Encapsulamento é definido como uma técnica para minimizar as interdependências entre os módulos programados através de interfaces externas restritas. Um módulo está encapsulado se os clientes só podem fazer acesso aos seus métodos através de sua interface

externa, o que é fornecido pela linguagem de programação. Portanto, essa característica garante aos projetistas que mudanças serão feitas de modo seguro, facilitando a manutenção e evolução do sistema.

Do ponto de vista de testes, o encapsulamento reduz problemas relativos ao escopo de variáveis, pois oculta detalhes internos da implementação de atributos e métodos. Com isso, reduz o risco de que alterações em características de uma classe afetem outras, já que a troca de mensagens entre instâncias destas (objetos) é possível entre aquelas cujas interfaces sejam conhecidas; e reduz a possibilidade de ocorrência de falhas de caminho de execução, uma vez que, em geral, os métodos são pequenos, com algoritmos complexos devido à alta coesão existente dentro de uma classe.

Apesar da vantagem que o encapsulamento e ocultação de informação trazem em termos de melhoria de modularidade, essas características limitam a habilidade de controle e observação de uma classe, reduzindo seu habilidade em satisfazer aos critérios de teste pré-estabelecidosa, pois detalhes internos de implementação de atributos e métodos não são acessíveis.

Embora encapsulamento não contribua diretamente para a ocorrência de erros, ele pode apresentar um obstáculo para os testes (Binder 2000a), já que um objeto não consegue detectar mudanças de estados produzidas em outro objeto. Assim, durante a atividade de teste não é possível relacionar o estado concreto e abstrato de um objeto, dificultando a verificação do resultado do teste.

Herança

Herança é um mecanismo que permite a uma subclasse (classe filha) herdar atributos e métodos de sua superclasse, que são acrescidos às novas características da subclasse.

Existem dois usos principais de herança: herança de implementação e herança de comportamento, conforme descrito em Buzato & Rubira (1998). Na herança de implementação a subclasse pode remover ou renomear algumas características da superclasse, não sendo garantido que a subclasse tenha o mesmo comportamento que a sua superclasse. Já na herança de comportamento, a subclasse se comporta como se fosse a superclasse; à subclasse podem ser acrescentadas novas características além daquelas herdadas, e estas podem ser redefinidas.

Em relação aos testes, poder-se-ia pensar que o mecanismo de herança facilitaria o reuso de conjuntos de testes da mesma forma que favorece o reuso de código e de projeto, porém não é tão simples assim. A seguir, são citadas algumas implicações apresentadas por Martins (1999):

- o reuso de código/projeto aumenta a necessidade de testes, pois as características herdadas devem ser testadas a cada reutilização, devido a possíveis mudanças no

contexto de uso das mesmas;

- o reuso de casos de testes não é simples de ser decidido, pois se a herança não for usada de forma disciplinada, os casos de teste precisam ser novamente gerados para a subclasse, mesmo para características que não foram alteradas;

- 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 a complexidade. Isto implica aumento do esforço necessário para a preparação dos testes, especialmente para determinar que entradas de testes podem ser reaplicadas e quais devem ser modificadas;

- 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 escolha desta linguagem.

Polimorfismo e ligação dinâmica

Polimorfismo é a habilidade de vincular uma referência a mais de um objeto (Binder 2000a). No contexto de orientação a objetos, o polimorfismo significa que diferentes tipos de objetos podem responder a mesma mensagem de maneiras diferentes.

Polimorfismo está associado a uma série de conceitos, dos quais serão apresentados os mais simples (Martins 1999): redefinição, sobrecarga e parametrização. (1) Redefinição permite que um método da superclasse seja redefinido na subclasse. O método redefinido na subclasse deve ter a mesma interface que o da superclasse; métodos com o mesmo nome mas com interfaces distintas são considerados métodos novos. (2) Sobrecarga permite associar um método a um objeto em tempo de execução. (3) A parametrização, permite o uso de classes genéricas ou classes parametrizadas, as quais servem de moldes para outras classes.

A ligação dinâmica é uma forma de implementar o polimorfismo. Pode ser definida como sendo a associação, em tempo de execução, de uma mensagem ao objeto que contém o método que a aceite.

O uso de Polimorfismo e a ligação dinâmica introduzem as seguintes dificuldades para os testes:

- aumento da complexidade na geração de testes baseados na implementação (uma vez que o código a ser executado é mais dependente das condições de execução do que nas linguagens imperativas).

- ocorrência de muitos erros de execução causados pelo uso de herança de implementação, já que a classe derivada não tem o mesmo comportamento de suas ancestrais, dificultando a seleção de entradas de teste que permitam que esses erros sejam detectados;

- impossibilidade de determinar todos os usos polimórficos de um objeto ou característica.

Relacionamentos

Objetos podem se relacionar de diferentes formas (Siegel 1996): um objeto pode conter outro objeto (isto é, um objeto pode ter outros objetos como atributos); objetos podem enviar mensagens para outros objetos, com argumentos e valores de retorno, tanto uns quanto os outros podem ser objetos; objetos podem ser executados concorrentemente, em diferentes *threads*, trocando mensagens entre si de maneira síncrona ou assíncrona; objetos podem ser compatíveis ou incompatíveis uns com os outros em determinado contexto.

Um objeto pode conter outros de diferentes formas: uma classe pode ser declarada internamente a outra; uma instância de um objeto é criada dentro de outro objeto; através de herança, as características de uma classe são compartilhadas pelas derivadas.

No envio de mensagens, um objeto invoca um método de outro objeto, havendo opcionalmente o uso de argumentos e valores de retorno; objetos podem ser passados como argumento ou como valor de retorno. O envio de mensagens está relacionado com a visibilidade, pois corresponde à invocação de métodos na interface pública de um objeto. Logo, se um objeto está encapsulado em outro (não é visível externamente), não pode receber mensagens de outros objetos.

Com esses relacionamentos, mais algumas dificuldades são acrescentadas aos testes:

- o estado de um objeto, de um modo geral, não depende somente dos valores de suas variáveis internas, mas também dos valores de suas associações e dos estados dos objetos que o compõem;

- o uso de estratégias incrementais para a integração se torna mais difícil;

- a construção de *stubs* torna-se muito mais complexa;

- os relacionamentos, apesar de serem descritos estaticamente, variam a cada execução do sistema;

- a concorrência aumenta a complexidade dos relacionamentos, dificultando ainda mais os testes.

Riscos inerentes a linguagem OO

Cada linguagem de programação orientada a objetos apresenta suas particularidades, o que proporciona maior ou menor tendência a apresentar certos riscos para a atividade de teste dos sistemas.

Em Binder (2000a) é mostrado o relacionamento entre erros, falhas e defeitos, conforme uma taxonomia de falhas proposta por Purchase & Winder (1991), desenvolvida para auxiliar no teste de códigos orientados a objetos, como a linguagem C++. Falhas são classificadas de acordo com o processo (atividade na qual a falha foi introduzida), com o defeito (estado ou comportamento incorreto observado) e com o erro (enganos cometidos pelo desenvolvedor que levam à falha). Na tabela 2.1 é mostrado o escopo no qual cada erro é tipicamente revelado durante os testes, onde A = aplicação, S = subsistema, C = classe e M = método.

2.4.2 Fases de Teste

Em sistemas orientados a objetos a menor unidade testável é a classe ou o objeto (Pressman 1997a). Assim, as fases de teste possuem um foco diferente dos testes de sistemas convencionais:

- Teste de unidade: o teste de classe em software OO corresponde ao teste de unidades de software convencionais. Esse teste é dirigido aos métodos encapsulados pela classe e ao seu comportamento.

- Teste de integração: duas abordagens são consideradas nos testes de integração em software OO, já que eles não apresentam uma estrutura de controle hierárquica como nos convencionais. A primeira - testes baseados em *threads* - visa integrar o conjunto de classes necessárias para responder a uma entrada ou evento do sistema; também aplica-se testes de regressão nessa etapa. Outra abordagem são os testes baseados no uso, que começam a integração do sistema pelas classes que não se relacionam com outras classes (classes independentes), depois são integradas às classes que se relacionam diretamente com as independentes, e assim por diante até que todas as classes do sistema sejam integradas. Um outro passo do teste de integração é o teste de *cluster* (uma coleção de classes que colaboram entre si), que visa encontrar erros na colaboração dos objetos das classes do *cluster*.

- Testes de validação ou de sistema: os detalhes de conexão das classes não são mais de interesse dos testes. Nessa etapa, o enfoque está nas ações de interação dos usuários com o sistema (informações fornecidas pelo usuário). O que não difere muito dos testes convencionais. Métodos de testes caixa-preta são suficientes para a sua realização.

Processo	Erro	Escopo				Defeito
		A	S	C	M	
Definição de requisitos	Percentual: análise do problema inadequada. Defeito de comunicação.	X	X	X		Problema da aplicação não resolvido.
Projeto	Análise do problema inadequada. Erro de especificação.		X	X	X	Requisitos ou especificação não conferem.
	Abstração: conhecimento inadequado de técnicas de projeto de OO. Construção prematura de hierarquia ou níveis.		X	X		Estrutura de classe pobre, construção ruim, hierarquia de herança relaxada, inconsistente.
	Algoritmo: especificação equivocada. Implementação incorreta.			X	X	Método de saída incorreto.
	Conhecimento de reuso inadequado. Componente reusado ou hierarquia equivocada. Uso incorreto de polimorfismo.	X	X	X		Comportamento não esperado e indesejável no reuso de componentes.
Implementação	Mau uso de componentes reusáveis, extensões *ad-hoc* de heranças.			X	X	Método de saída incorreto.
	Semântica: conhecimento de programação inadequado ou mau uso dos serviços do ambiente alvo.			X	X	término anormal devido a erro de tipo com ligação atrasada.
	Sintática: conhecimento de programação inadequado, erro tipográfico.			X	X	Saída incorreta, término anormal.
Tempo de execução	Conhecimento inadequado da aplicação ou serviços alvo.			X	X	Exceção em tempo de execução, defeito na fonte de mensagens.

Tabela 2.1: Erros e defeitos.

Uma vez que os atributos e operações são encapsulados, o teste de operações fora da classe não se justifica (Pressman 1997a). Dessa forma, o encapsulamento pode tornar difícil a obtenção das informações do estado de um objeto, a menos que, dentre os métodos desenvolvidos, existam aqueles que apresentam os valores dos atributos das classes, porém ainda é difícil obter o estado de um objeto em determinado instante. A herança também traz preocupações novas para o projeto de casos de testes de sistemas OO, já que para cada novo contexto de uso requer reteste. No entanto, se a subclasse é usada em contextos completamente diferentes, os casos de testes da superclasse não serão reaproveitados pela subclasse, sendo necessário gerar um novo conjunto de testes para ela.

O método de teste caixa-branca, que é utilizado em testes de sistemas convencionais para gerar casos de testes baseado no código fonte, pode ser utilizado para o teste das operações definidas para a classe, mas não é suficiente para os testes de classe. Para garantir que muitas declarações de uma operação foram testadas podem ser utilizadas as técnicas de caminho base, teste de *looping* ou fluxo de dados. O problema com testes caixa-branca está no grande número de casos de testes, aumentando o custo dos testes.

O desenvolvimento de casos de testes que utilizam requisitos ou especificações são referenciados como testes caixa-preta. Esse método é útil para os testes de sistemas orientados a objetos. Mesmo especificações bem desenvolvidas não expressam todos os detalhes de implementação necessários para a geração dos casos de testes, sendo necessário alguns exames no código fonte, por exemplo: o reteste de características herdadas requer análise da estrutura da classe. Mesmo assim, a distância entre especificações orientadas a objetos e implementações são tipicamente menores quando comparadas com sistemas convencionais (Binder 2001).

Os testes baseados em falhas, em sistemas OO, têm por objetivo projetar testes que possuam alta probabilidade de descobrir falhas plausíveis (isto é, aspectos da implementação do sistema que pode resultar em defeitos). Para desempenhar testes baseados em falhas, é interessante iniciar pelo modelo de análise construído com base nos requisitos do cliente, no entanto, os casos de testes são projetados para exercitar o projeto ou o código, a fim de determinar se as falhas existem.

Para complementar os testes baseados em falhas, os testes baseados em cenários são utilizados para cobrir os testes de especificação incorreta e integração entre subsistemas. Quando ocorrem erros associados a incorreta especificação, o produto não faz o que o cliente quer. Já os erros associados com interação de subsistemas ocorrem quando o comportamento de um subsistema gera eventos ou fluxos de dados que levam outro subsistema a falhar. Esses testes concentram-se no que o usuário faz, não no que o produto faz. Dessa forma, são capturadas as tarefas que o usuário deve desempenhar através de casos de uso.

2.4.3 Métodos de teste de classes

Durante os testes orientados a objetos são utilizados alguns métodos para exercitar uma classe: teste aleatório e teste de partição.

Por teste aleatório (*random testing*) compreende-se a execução de diferentes seqüências de testes para exercitar diferentes históricos de vida (comportamentos) de uma instância da classe.

Os testes de partição reduzem o número de casos de testes requeridos para exercitar uma classe. Entradas e saídas são classificadas, e os casos de testes são designados para exercitar cada categoria. Em Pressman (1997a) são determinadas três partições: (1) A

partição baseada em estado classifica as operações da classe com relação a sua habilidade de mudar o estado da classe; (2) A partição baseada em atributo classifica as operações da classe em relação aos atributos que elas usam; (3) A partição baseada em categorias classifica as operações da classe baseada em funções genéricas que cada uma desempenha, por exemplo: operações de inicialização, operações modificadoras, de iteração e operações de término.

2.5 Considerações Finais

Neste capítulo foi exposta uma visão geral de sistemas, principalmente as particularidades referentes a testes em sistemas orientados a objetos.

No próximo capítulo serão exploradas as características dos testes de regressão, objetivo deste trabalho.

Capítulo 3

Teste de Regressão

Na fase de manutenção de um software, quando são realizadas melhorias, correções ou incorporadas novas funcionalidades ao sistema, o teste do software é tão importante quanto aquele realizado durante a fase de desenvolvimento do software. Dessa forma, para garantir que não foram introduzidas novas falhas no sistema é indispensável o uso de teste de regressão que compõe uma das atividades de manutenção e/ou evolução do software. Embora o esse teste seja considerado uma necessidade no processo de manutenção, é uma atividade dispendiosa, chegando a corresponder pela metade do custo de manutenção do software (Beizer 1990).

As partes do sistema que sofreram modificações devem ser tão revalidadas quanto aquelas que não sofreram, com o propósito de garantir que funcionalidades não modificadas não tenham sido afetadas (Leung & White 1991). Com isso, é possível garantir que o programa ainda satisfaz seus requisitos (Harrold & Rothermel 1994), isto é, comporta-se como previsto e que as alterações não afetaram indesejavelmente o comportamento do código não modificado (Rothermel, Harrold & Dedhia 2000).

Uma estratégia de teste de regressão é executar novamente todos os casos de teste, porém o reteste total pode consumir tempo e recursos inaceitáveis. Já as técnicas de reteste seletivo tentam reduzir o tempo necessário para retestar um programa modificado, pois selecionam testes para reuso nas partes alteradas do programa. Com isso, essas técnicas levantam dois problemas: o problema de selecionar testes a partir de um conjunto existente de testes, e o problema de determinar quando testes adicionais são necessários. O primeiro problema é amplamente discutido na literatura com intuito de buscar técnicas de seleção que reduzam significativamente o custo do teste de regressão no programa modificado (Binkley 1996, Graves, Harrold, Kim, Porter & Rothermel 1998, Kung, Gao, Hsia, Toyoshima & Chen 1996, Rothermel et al. 2000).

Neste capítulo serão abordados alguns pontos relevantes de teste de regressão, como considerações iniciais, os problemas enfrentados nesta fase, os passos para execução dos

testes, formas de comparar as diversas técnicas de teste de regressão seletivo, dentre outros.

3.1 Teste de Regressão

Quando componentes novos ou modificados inseridos num software podem causar falhas ao interagir com os componentes não alterados, pela geração de efeitos colaterais ou devido a características de interação, diz-se que o sistema sob testes regrediu (Binder 2000b), por isso os testes aplicados nesse novo sistema são denominados teste de regressão.

Segundo Binder (2000a), teste de regressão normalmente é aplicado durante o desenvolvimento iterativo, depois de depurar o programa, durante a produção de uma nova instanciação de um componente reutilizável, como o primeiro passo para integração, e como suporte para a manutenção de aplicações.

- Desenvolvimento iterativo. Com o alto ritmo de alterações em desenvolvimento orientado a objetos, alguns autores chegam a reportar o uso de testes de regressão mais de um vez por dia, outros acreditam que os testes devem ser desenvolvidos para cada classe e toda vez que a classe é alterada os testes deveriam ser reaplicados.

- Desenvolvimento por reuso. Com a utilização de componentes de software é cada vez mais importante que um conjunto de testes seja disponibilizado junto com o componente. Assim, o cliente pode reutilizar esses casos de testes como testes de regressão ao gerar o seu componente resultante.

- Integração. Um primeiro passo para o teste de integração é o teste de regressão, já que a re-execução de conjuntos de testes acumulados no componente garante a construção de um conjunto de regressão incremental capaz de revelar falhas de regressão.

- Manutenção de software. Teste de regressão pode ser realizado em manutenções corretivas, adaptativas, preventivas ou evolutivas para revelar efeitos colaterais e ajustes errôneos através do conjunto de testes do componente original.

- Avaliação de compatibilidade. Alguns conjuntos de teste são projetados para executar em várias plataformas e sistemas de aplicação para estabelecer conformidade com um padrão ou calcular desempenho de tempo e espaço. A intenção dos testes é revelar diferenças de desempenho ou funcionalidade dependentes de plataforma, não necessariamente de falhas de regressão.

Depois que um conjunto inicial de testes foi executado no sistema, testes de regressão podem ser aplicados em qualquer escopo da implementação sob teste e em qualquer ponto do seu desenvolvimento.

Para a aplicação de testes de regressão numa versão beta[1] do software, deve-se seguir os seguintes passos:

1. Identificar as modificações realizadas na versão beta em relação a original;
2. Selecionar os casos de teste a serem re-executados dentre os pertencentes a seqüência original;
3. Aplicar os testes selecionados na versão beta;
4. Gerar novos testes, caso existam funcionalidades novas em beta;
5. Aplicar os novos testes;
6. Atualizar o histórico de testes.

O teste de regressão possui duas fases distintas: a **fase preliminar** e a **fase crítica**. A fase preliminar começa depois do lançamento de alguma versão do software; durante essa fase ocorrem pequenas melhorias e correções do software. Quando as correções estão completas, a fase crítica do teste de regressão inicia; nesta fase o teste de regressão é a atividade dominante, e seu tempo é limitado pelo prazo de entrega do produto (Harrold & Rothermel 1997).

3.2 Estratégias para Teste de Regressão

A mais simples estratégia de teste de regressão é a de reteste total (*retest all*), que utiliza todos os testes do conjunto original. Porém essa abordagem pode consumir tempo e recursos excessivos.

Uma alternativa, reteste seletivo (*selective retest*), escolhe do conjunto de testes original aqueles testes que são considerados indispensáveis para testar o programa modificado e, quando necessário, novos casos de testes são acrescentados.

Uma das dificuldades das técnicas seletivas está em escolher um subconjunto de testes que seja capaz de exercitar as partes do programa que foram alteradas para, com isso, ter mais chances de encontrar falhas.

[1] Versão Beta de um sistema é uma nova versão do sistema que ainda não foi aprovada, ou seja, ainda não passou por teste.

Essas técnicas têm sido cada vez mais discutidas e aprimoradas, a fim de diminuir o custo dessa seleção (?).

As técnicas seletivas de teste de regressão são classificadas em: técnicas de minimização, de cobertura, técnicas seguras e aleatória. Essa classificação está detalhada na seção 3.2.1.

Como na fase de manutenção de um software busca-se a redução de custos, é muito importante a utilização das técnicas seletivas para a realização dos testes de regressão.

Uma técnica seletiva de teste de regressão normalmente segue os seguintes passos, onde: **P** é um programa, e **P'** é uma versão modificada de **P**; **T** é conjunto de testes criado para testar **P** e T' o conjunto de testes selecionado a partir do conjunto de testes do programa original.

1. selecionar **T'** $\subseteq$ **T**, um conjunto de testes para executar em **P'**;

2. testar **P'** com **T'**, para determinar a corretude de **P'** em relação a **T'**;

3. se necessário, criar **T"**, um conjunto de novos testes funcionais ou estruturais para **P'**;

4. testar **P'** com **T"**, para determinar a corretude de **P'** em relação a **T"**;

5. criar **T"'**, um novo conjunto de testes e histórico de testes de **P'**, a partir de **T**, **T'** e **T"**.

O conjunto **T"'** inclui o conjunto de testes do programa original para manter o histórico de todos os testes executados no programa, tanto na versão original como nas demais. Desta forma, o novo conjunto **T"'** contém todos os testes possíveis para a nova alteração do programa.

Nas próximas seções serão descritas a classificação das técnicas seletivas de teste de regressão e algumas métricas para a comparação entre as diversas técnicas de reteste seletivo existentes.

3.2.1 Classificação das técnicas de reteste seletivo

As técnicas seletivas utilizadas no teste de regressão podem ser classificadas de acordo com a forma com que o subconjunto de testes é escolhido (Harrold & Rothermel 1994) e (Graves et al. 1998):

Técnica de Minimização

Na minimização assume-se que o objetivo global dos testes de regressão é restabelecer a satisfação de alguns critérios de cobertura estrutural, e visa identificar um conjunto mínimo de testes que devem ser reaplicados para satisfazer a esses critérios.

Técnica de Cobertura

As técnicas de cobertura têm como base os critérios de cobertura estrutural, mas ao contrário da anterior não há minimização; em vez disso, busca a seleção de testes que exercitem os componentes alterados ou afetados pelas alterações, que podem produzir saídas diferentes, e usa critérios de cobertura como guia de seleção para tais testes.

Técnica Segura

As técnicas seguras (*safe tecniques*) visam selecionar todos os testes que induzirão o programa modificado a produzir saídas diferentes das geradas pelo programa original, dando menos ênfase ao critério de cobertura.

Técnica Aleatória

A técnica aleatória ou *ad hoc* é usada quando a restrição de tempo torna proibitivo o uso da estratégia de reteste total, e não há ferramentas para a seleção de teste.

3.3 Framework para Avaliação e Comparação de Técnicas Seletivas de Teste de Regressão

Diante da grande variedade de técnicas seletivas para teste de regressão, em (Harrold & Rothermel 1994), foi proposta uma forma de avaliá-las e compará-las, pois os autores identificaram que apesar das técnicas seguirem filosofias distintas, como minimização, cobertura e segurança, elas possuíam categorias nas quais poderiam ser comparadas e avaliadas, sendo elas: inclusão, precisão, eficiência, generalidade e disponibilidade. A partir dessa descoberta, foi proposto e descrito um *framework*.

Em (Harrold & Rothermel 1996), os autores citam que o maior benefício do *framework* é que ele fornece uma maneira de avaliar e comparar técnicas seletivas de teste de regressão existentes. Com isso, facilita a escolha da técnica mais apropriada para uma aplicação em particular. Além disso, a avaliação e comparação das técnicas também fornece dicas das vantagens e limitações de cada técnica.

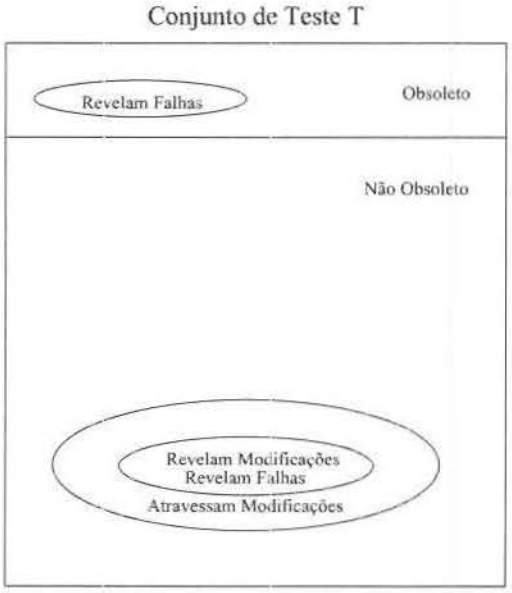

Figura 3.1: Relacionamento entre classes de teste.

Para a definição do *framework*, primeiro os autores classificaram os casos de teste em três classes: testes que revelam falhas (*fault-revealing*), testes que revelam modificações (*modification-revealing*) e aqueles que atravessam modificações (*modification-traversing*).

Os casos de testes classificados como aqueles que revelam falhas são os testes que revelam falhas existentes no programa. Entretanto, não existe um mecanismo eficiente para encontrar esses casos de teste a partir de um conjunto inicial de testes.

Os casos de teste que revelam modificações são os que revelam um comportamento diferente entre **P** e **P'**, onde **P** é o código original e **P'** o código modificado. Entretanto, para selecioná-los seria necessário executar todo o conjunto de casos de teste para saber quais conseguem revelar as modificações.

A solução é selecionar aqueles casos de teste que passam pelas modificações feitas no programa: os casos de teste que atravessam modificações, que incluem os casos de teste que revelam falhas e que revelam modificações. Assim, os casos de teste que atravessam modificações são aqueles com maiores chances de revelarem defeitos. Na figura 3.1 é mostrada a relação entre as três diferentes classes de conjuntos de casos de teste.

Além das três classes de conjuntos de teste definidos para este *framework*, tem-se a divisão dos casos de testes em obsoletos e não obsoletos. Os casos de teste obsoletos são aqueles que não serão reutilizados para as atividades de teste de regressão, enquanto que os casos de teste não obsoletos são aqueles que serão reutilizados. Ambos os casos de teste estão caracterizados na figura 3.1.

As três classes de teste descritas contribuíram para a especificação do *framework* para avaliação e comparação de técnicas seletivas para testes de regressão por diversas razões:

- O relacionamento entre as três classes fornece subsídios para uma avaliação analítica das técnicas de seleção de testes de regressão em termos de suas habilidades em selecionar ou evitar a rejeição de testes do tipo que revelam falhas. Apesar dessas técnicas terem por objetivo a seleção de casos de teste para satisfazer algum critério de teste, é razoável e importante avaliá-las em termos de suas habilidades de revelar falhas.

- As três classes também podem ser utilizadas para distinguir técnicas de teste de regressão. Muitas daquelas que são baseadas em código tentam identificar testes que executam componentes alterados no programa modificado. Muitas tentam ser mais precisas, eliminando do conjunto de testes, que executam os componentes alterados, alguns testes que claramente não causam saídas diferentes[2] nos programas original e modificado.

- Para grandes sistemas, o conjunto de testes deve ser funcional, já que o objetivo do teste não é cobertura de código de componentes. Quando o conjunto de testes é construído visando a cobertura de código, pode fazer pouco sentido selecionar todos os testes que passam através do componente, porque apenas alguns deles fornecerão essa cobertura. Mas, quando os conjuntos de testes são funcionais, parece particularmente importante não omitir testes do conjunto de testes do programa original que podem revelar falhas no programa modificado, mesmo que eles possam exercitar componentes que já foram exercitados por outros testes.

A seguir serão apresentadas cada uma das medidas que compõem o *framework* descrito em (Harrold & Rothermel 1996), para tanto será utilizada a seguinte notação:
P = programa original;
P' = programa modificado;
M = técnica de seleção de teste de regressão;
T = conjunto de testes de **P**;
T' = conjunto de testes selecionados por **M** dado **P**, **P'**, e **T**.

Inclusão

Inclusão mede a extensão com a qual a técnica de reteste seletivo escolhe os testes que induzem o programa modificado a produzir saídas diferentes quanto comparadas àquelas do programa original. Com isso, revela-se os defeitos causados pelas modificações.

[2]Teste que produz o mesmo resultado quando executado nas versões base e modificada.

A definição de Inclusão relativa a um programa particular **P**, programa modificado **P'** e conjunto de teste **T**, é como segue:

Definição 1 - Supondo que $\mathbb{T}$ contenha **n** casos de teste que sejam do tipo que revelam modificações para **P** e **P'**, e supondo que **M** selecione **m** desses casos de teste, a inclusão de **M** relativa a **P**, **P'** e **T** é:

(1) a porcentagem dada pela expressão (100(m/n)) se $n \neq 0$ ou

(2) 100% se n=0.

Definição 2 - Se para todo **P**, **P'** e **T**, a inclusão de **M** relativa a **P**, **P'** e **T** é 100%, então **M** é segura.

Tanto a inclusão quanto a segurança são medidas importantes. Se **M** é segura então **M** seleciona todos os testes não obsoletos em **T** que são aqueles que revelam falhas para **P'**. Se uma técnica de seleção de teste de regressão **M1** tem maior inclusão do que uma técnica **M2**, então **M1** tem maior habilidade em revelar falhas do que **M2**.

Precisão

Precisão mede a habilidade de uma técnica em evitar a escolha de testes que não causarão saídas diferentes no programa modificado em relação ao programa original.

A definição de Precisão relativa a um programa particular **P**, programa modificado **P'** e conjunto de teste **T**, é como segue:

Definição 3 - Supondo que **T** contenha **n** casos de teste que sejam do tipo que não revelam modificações para **P** e **P'**, e supondo que **M** omita **m** desses casos de teste. A precisão de **M** relativa a **P**, **P'** e **T** é:

(1) a porcentagem dada pela expressão (100(m/n)) se $n \neq 0$ ou

(2) 100% se n=0.

A precisão mede a extensão com a qual **M** evita selecionar testes que não causam um comportamento diferente no programa modificado. Em geral, quando compara-se técnicas de seleção de testes em termos de precisão, é possível identificar as técnicas que menos escolhem casos de teste desnecessários. Ao comparar a segurança das técnicas de seleção em termos de precisão, pode-se identificar técnicas que se aproximam mais do objetivo de selecionar exatamente os testes do tipo revelam modificações.

Eficiência

A eficiência de uma técnica de seleção de teste de regressão é medida em termos do espaço e do tempo gastos para aplicar e avaliar tal técnica. Quando o tempo é considerado, uma técnica seletiva é mais econômica que a técnica de reteste total (*retest all*) se o custo para selecionar **T'** é menor que o custo de executar os testes de **T - T'** (Leung & White 1991).

A eficiência de tempo e espaço depende do tamanho do conjunto de testes que a técnica seleciona, além do seu custo computacional.

A seguir são mostrados alguns fatores que influenciam na avaliação da eficiência de uma técnica de seleção:

- Se as fases preliminar e crítica dos testes de regressão são realizadas (conforme seção 3.1);
- Se é necessário muito esforço humano durante os testes;
- Se é necessário determinar cada componente modificado, removido ou adicionado ao programa original assim que ele aparece;
- Se a técnica reavalia ou atualiza incrementalmente o histórico de testes depois de considerar cada modificação.

Generalidade

A generalidade de uma técnica de seleção de teste de regressão é a sua habilidade em funcionar diante diversas situações. Para avaliar uma técnica quanto à generalidade, os seguintes fatores devem ser considerados:

- Se a técnica funciona para vários tipos de programas, com várias construções estruturais;
- Se a técnica manipula programas com modificações realistas;
- Se a técnica depende do ambiente de teste e de manutenção;
- Se a técnica depende da disponibilidade de uma ferramenta de análise particular;
- Se a técnica suporta seleção de teste de unidade (intraprocedural) ou de integração (interprocedural). Na prática, teste de regressão é freqüentemente realizado com testes de integração.

Generalidade poderia ser definida mais quantitativamente, porém Harrold & Rothermel (1996) preferem medir generalidade em relação às comparações qualitativas, já que julgaram-as suficientes para o *framework*.

3.4 Manutenção do Conjunto de Testes

A deterioração do conjunto de teste de regressão é inevitável e ocorre por diversas razões: modificações na interface da aplicação, suas capacidades e implementação. Dessa forma, os seguintes tipos de casos de teste da base devem ser descartados (Binder 2000b):

- Caso de teste quebrado: aquele que usa um código, componente ou interface que foi removida ou alterada, ou cujo resultado é comparado usando uma interface ou estrutura de dados que sofreu alteração. Esses casos de teste falham sem começar a execução e devem ser removidos ou retrabalhados.

- Caso de teste obsoleto: um caso de teste que é aceito por um sistema sobre teste, porém não é mais apropriado porque seus requisitos sofreram alterações. Isso ocorre quando os limites de determinada variável aumentam de forma que um teste anterior do limitante superior continua executando com sucesso. Porém, como o limitante sofreu alteração, esse teste deixa de validar o verdadeiro limitante, inclusive excluindo incorretamente valores que agora fazem parte dos limitantes para a variável.

- Caso de teste incontrolável: aquele que é sensível à entradas ou estados com valores suspeitos, não sendo possível reproduzi-los nem mesmo no ciclo de testes inicial.

- Caso de teste redundante: dois ou mais casos de teste são redundantes se suas entradas e saídas para uma interface específica são idênticas.

O seguinte procedimento deve ser aplicado para manter um conjunto de teste de regressão com um alto nível de eficiência (Binder 2000b):

1. Executar o conjunto de teste da versão base em beta. Remover ou refazer casos de teste quebrados ou obsoletos.

2. Corrigir todos os erros revelados. Se algum erro de baixa severidade não for corrigido, os mesmos deverão ser devidamente explicados na documentação de casos de teste. Estes casos que revelaram erros não corrigidos não devem ser removidos.

3. Juntar os casos de teste de escopo de componente, desenvolvidos para as novas capacidades do componente beta, com o conjunto de teste base.

4. Instrumentar a versão beta usando um analisador de cobertura para mapear os caminhos ou segmentos dos casos de teste.

5. Re-executar o conjunto de testes em beta instrumentado.

6. Analisar os testes que atravessam os mesmos caminhos de entrada-saída. Verificar se existem testes necessários para realizar algum estado, detalhe de implementação ou uma dependência de suporte. Em caso negativo, remover todos exceto um dos testes cuja entrada e saída são idênticas para um conjunto de segmentos ou um caminho entrada-saída.

7. Se algum segmento, componente ou interface crítica não é exercitada, desenvolver um novo teste para alcançá-la.

8. Re-executar o conjunto de teste na versão beta não instrumentada. Todos os testes devem passar. Se não, executar e depurar o sistema sobre teste (ou os testes) até que todos passem.

9. Verificar o conjunto de teste revisado com o novo conjunto de testes base. Quando possível associar o conjunto de testes base com a nova construção do sistema sobre teste.

3.5 Teste de Regressão para Software Orientado a Objetos

Embora o problema de seleção de testes de regressão tenha sido amplamente discutido para software procedural, apenas nos últimos anos as pesquisas estão sendo voltadas para software orientados a objetos (Rothermel & Harrold 1994). Devido a ênfase do paradigma de orientação a objetos em reuso, o custo do teste de regressão sofre um significativo aumento. Ao mesmo tempo, esse paradigma apresenta um grande potencial para obtenção de economia em teste pelo uso de técnicas seletivas.

O teste de software orientado a objetos requer uma técnica para teste de classes (Harrold et al. 1992). Essa técnica invoca algumas seqüências de métodos em diversas ordens, e depois verifica se o estado resultante dos objetos manipulados pelos métodos está correto.

Quando uma classe é modificada, as alterações decorrentes podem causar impacto em todas as aplicações que usam esta classe, e em várias classes derivadas dela; logo, idealmente, todas aplicações e classes derivadas devem ser retestadas.

Conforme Binder (2000a), o teste de regressão de classe pode ser utilizado em várias situações:

- Quando uma nova subclasse é desenvolvida, é necessário re-executar os testes da superclasse na subclasse além de gerar novos testes para esta classe especializada.

- Quando uma superclasse é alterada, re-executar os testes da superclasse nela e nas subclasses. Re-executar os testes das subclasses.

- Quando a correção de um erro é completada, se o erro havia sido revelado por um teste, re-executar o teste - ele agora deve passar. Então, faz-se necessário re-executar os testes para revelar efeitos não pretendidos da correção do erro.

- Quando uma nova construção do sistema é produzida, isto é, uma nova versão do sistema. Um conjunto de testes construído passaria se a construção for bem sucedida e não houver erros extraordinários. Um conjunto de testes tipicamente inclui testes que exercitam cada característica referente a primeira construção do sistema sob teste.

- Quando uma versão do sistema está pronta e a versão final é gerada, contendo somente arquivos executáveis, código em local específico e os códigos de teste estão desabilitados (como assertivas). O conjunto de testes do sistema inteiro deve ser re-executado para revelar os erros introduzidos inadivertidamente e os itens omitidos.

3.6 Abordagens para Teste de Regressão

Em Binder (2000a) são determinados cinco abordagens para a seleção de conjuntos de testes de regressão: total, baseada em casos de uso, por perfil, de segmentos modificados e no *firewall*. Essas abordagens (*patterns*) de teste são aplicáveis em qualquer escopo: podem ser usadas em desenvolvimento iterativo, na instanciação de componentes reusáveis, e manutenção.

Em todas as abordagens, os casos de teste do programa origem são reutilizados depois que os casos de testes obsoletos são removidos. O resultado dos testes do programa origem são utilizados como oráculo, e os testes do componente beta passam se a comparação dos valores encontrados com este oráculo resultar em igualdade.

Um componente beta pode induzir ao defeito por vários motivos, principalmente devido a uma incompatibilidade, efeito colateral, ou interação indesejável.

As abordagens de testes de regressão podem ser aplicadas quando as seguintes condições são satisfeitas: (1) o componente beta passa pelo teste de escopo do componente; (2) existe um conjunto de testes para o programa origem adequado; (3) o ambiente de teste é restaurado para a mesma configuração usada na execução do conjunto de teste da versão original.

Os testes terminam quando: todos os casos de teste que não passaram revelam erros cuja presença e severidade são julgadas aceitáveis; e todos os casos de teste restantes passaram.

3.6.1 Reteste total

É a opção padrão, mais simples e mais fácil. Possui o menor risco de perder uma falha de regressão, porém torna-se proibitivo em certas aplicações pelo tempo e custo de executar novamente todo o conjunto de testes.

Consiste em re-executar todo o conjunto de testes da versão base depois da remoção dos casos de teste obsoletos (aqueles testes que não fazem mais sentido na nova versão, testam funcionalidade e/ou seqüências de testes que não existem mais).

Diz-se que os testes passaram quando a execução dos casos de teste no sistema produz os mesmos resultados se comparados àqueles obtidos na versão base. Ou seja, o sistema ainda atende as mesmas funcionalidades. Dessa forma, o oráculo é mantido pelos resultados da execução dos testes na versão base.

Pode-se citar como conseqüências de reteste total:

- Inclusão: como todos os casos de testes da versão base são selecionados, essa abordagem é segura.
- Precisão: nenhum teste, cuja execução poderia não ocorrer, foi omitido. Por essa razão, é a menos precisa.
- Eficiência: reteste total tem o mais baixo custo de análise, mas o mais alto custo de execução quando comparado com as abordagens seletivas de regressão.
- Generalidade: pode ser aplicado em quase todas as circunstâncias. A execução dos testes é fácil e não requer qualquer análise ou desenvolvimento especial.

3.6.2 Reteste baseado em casos de uso de maior risco

Essa abordagem faz uso de heurísticas baseadas em risco para selecionar o conjunto de testes de regressão parcial, o qual será aplicado posteriormente no teste do componente beta.

Normalmente, reteste parcial é utilizado quando o tempo, pessoal ou equipamentos disponíveis não são suficientes para executar toda a regressão. Em Binder (2000a) é posto o seguinte questionamento: como um subconjunto do conjunto de testes base pode ser selecionado sem uma análise detalhada da implementação?

A seguir são definidos os critérios de risco que podem ser utilizados para a seleção de um subconjunto de testes a partir do conjunto base:

- Casos de uso suspeitos: seleciona testes para casos de uso que dependam do comportamento, dos objetos ou de outros recursos que (1) são individualmente não estáveis e não provados, (2) não trabalharam juntos antes, (3) implementam regras

de negócio complexas, (4) possuem implementação complexa, ou (5) foram sujeitos a grandes problemas durante o desenvolvimento.

- Casos de uso críticos: seleciona testes para casos de uso que são necessários para operações seguras e eficientes. Para cada caso de uso, tenta-se identificar os efeitos de defeitos no pior caso. Omitir testes para aqueles onde um defeito pode ter níveis de severidade mais baixos.

O objetivo dessa abordagem é omitir casos de uso não críticos, de baixa prioridade ou altamente estáveis, de forma a satisfazer o prazo final ou restrição de orçamento para a realização dos testes.

Sendo assim, o procedimento adotado por essa abordagem é identificar, desenvolver e executar um conjunto de testes reduzido. Os testes passam se o sistema sob teste produz resultados idênticos àqueles obtidos na versão base.

Os critérios de entrada para esse procedimento são compreendidos pelas seguintes condições: (1) o componente beta passou pelo teste no escopo de componente; (2) existe um conjunto de teste base adequado; (3) o ambiente de teste foi restaurado para a mesma configuração usada para executar os testes na versão base, caso isso não seja possível, componentes podem apresentar comportamento incontrolável e o risco da execução dos testes deve ser avaliada.

Como critério de saída, esta abordagem apresenta: todos os casos de teste que não passaram revelam erros cuja presença e severidade são julgadas aceitáveis. E todos os demais casos de teste passaram, isto é, não resultaram na descoberta de erros.

Reteste baseado em casos de uso de maior risco têm um risco moderado de perder falhas de regressão e geralmente um baixo custo de análise e instalação, sendo classificado quanto as métricas:

- Inclusão: essa abordagem não é segura porque os casos de teste não são selecionados pela análise de dependências.

- Precisão: alguns testes que podiam ser omitidos podem ser repetidos.

- Eficiência: o tempo e custo associado com a identificação do conjunto de teste de regressão reduzido são desprezíveis. Como a seleção é baseada em casos de uso (requisitos) e não nas dependências de implementação, a análise pode ser feita sem análise de código ou conhecimento técnico profundo do sistema sobre teste. O tempo considerado será o de execução dos casos de teste e análise dos casos de uso.

- Generalidade: pode ser aplicado em quase qualquer circunstância e em qualquer escopo.

3.6.3 Reteste por perfil operacional

Essa abordagem seletiva usa o perfil operacional caso haja restrição de orçamento para selecionar o conjunto de testes de regressão parcial, executando esse subconjunto na versão beta. A grande questão a ser resolvida é: como um subconjunto do conjunto de testes base pode ser selecionado de forma a fornecer maior fidelidade com o prazo final ou orçamento?

Ela requer que o conjunto de casos de teste do programa base tenha sido desenvolvido considerando a Alocação de Testes por Perfil. A alocação de testes por perfil tem por objetivo alocar todo o orçamento de teste, dividindo-o proporcionalmente para cada caso de uso conforme sua freqüência relativa. O princípio que motiva o teste sobre o perfil operacional é o seguinte: a freqüência de defeitos que aparece para o cliente/usuário é diretamente relacionado com a freqüência de uso (Binder 2000a).

O procedimento adotado é verificar se todos os casos de uso críticos e variantes de casos de uso são incluídos no conjunto de testes de regressão, mesmo se eles ocorrem com uma baixa freqüência.

O oráculo, no caso de reteste por perfil operacional, é fornecido pelos resultados da execução dos testes na versão base.

Como critério de entrada, essa abordagem requer que o conjunto de testes da versão base seja desenvolvido utilizando alocação de teste por perfil. O teste de regressão pode começar quando: (1) o componente beta passa pelo teste ao escopo de componente; (2) existe um conjunto de teste para a versão base; (3) o ambiente de teste está restaurado para a mesma configuração usada para executar o conjunto de teste original.

O critério de saída é alcançado quando todos os casos de teste que não passaram revelam erros cuja presença e severidade são julgadas aceitáveis. E quando todos os casos de teste restantes passaram, isto é, não resultaram na descoberta de erros.

Reteste baseado no perfil operacional tem um risco moderado de perder uma falha de regressão, apresentando as seguintes conseqüências em relação as métricas:

- Inclusão: essa abordagem não é segura porque os casos de teste não são selecionados pela análise de dependências.

- Precisão: alguns testes que podiam ser omitidos podem ser repetidos.

- Eficiência: o tempo e custo associado com a identificação do conjunto de teste de regressão reduzido são desprezíveis. Como a seleção é baseada em casos de uso (requisitos) e não nas dependências de implementação, a análise pode ser feita sem análise de código. Se o perfil operacional já foi estabelecido, o custo de análise de seleção pode ser baixo. O tempo considerado será o de execução dos casos de teste e análise dos perfis operacionais.

- Generalidade: pode ser aplicado em muitas circunstâncias. Como a eficácia dos testes baseados em perfil diminui num escopo menor, essa técnica é mais adequada pra testes de escopo de sistema de aplicações, para as quais um perfil operacional preciso pode ser desenvolvido.

3.6.4 Reteste baseado em segmentos modificados

Usa a análise de códigos modificados para a seleção de conjunto de testes de regressão parcial e executa esse subconjunto na versão beta. Essa abordagem procura resolver a seguinte questão: como um subconjunto do conjunto de testes base pode ser selecionado usando análise automatizada de código?

Essa abordagem pode ser aplicada no escopo de classe, cluster ou subsistema e depois na manutenção, para reuso, e durante desenvolvimento iterativo.

O critério de entrada é composto por: (1) componente beta que passou pelo teste no escopo de componente; (2) um conjunto de testes para a versão base; (3) um ambiente de teste restaurado para a mesma configuração usada para execução dos testes na versão base, se isso não for possível os componentes podem apresentar comportamento incontrolável e o risco da execução dos testes deve ser avaliado.

A seleção de testes de regressão baseados em implementação concentra-se em encontrar testes do conjunto base que revelarão falhas de regressão na versão beta. Uma falha de regressão deve ser relacionada com um segmento de código novo, alterado ou removido. Os testes do conjunto base são selecionados pela comparação de cada segmento no código base e no componente beta através da utilização de um algoritmo de percurso em grafos, já que as declarações de cada uma das versões da classe são representadas por grafos.

O procedimento de teste adotado nessa abordagem é composto por cinco passos, definidos como segue:

1. Obter um relatório de análise de cobertura que liste os casos de teste e quais segmentos do sistema sobre teste são exercitados em cada um destes casos;

2. Extrair pares com a identificação de cada segmento e a identificação do caso de teste que exercita esse segmento. Pode ocorrer mais de um registro para o mesmo segmento, pois mais de um caso de teste pode exercitar determinado segmento do sistema sobre teste, resultando no arquivo TesteBloco.txt;

3. Usar uma ferramenta de controle de alterações para relacionar as diferenças entre a versão base e a beta. Em cada registro deve existir um número que identifica o segmento e o tipo de alteração: mesmo, novo, alterado ou removido. Como resultado, deve-se obter um arquivo (AlteracaoBloco.txt) com a união de todos

os blocos que identificam os segmentos e sua classificação em relação ao tipo de alteração que ocorreu;

4. Concatenar os resultados do passo 2 (arquivo TesteBloco.txt) e 3 (arquivo AlteracaoBloco.txt), ordenar as informações por segmento e casos de teste e gerar um arquivo: DeltaOrdenado.txt;

5. A lista de testes que devem ser usados e desprezados nos testes de regressão pode ser obtida da lista do passo 4. As regras para seleção são simples: testes classificados como 'mesmo' devem ser desprezados, enquanto aqueles classificados como 'alterado' ou 'removido' devem ser incluídos. Os outros classificados como 'novos' não devem ser considerados, porque não são considerados no teste de regressão, são segmentos que não existiam na versão base, logo não existem casos de teste que os exercitem. Desta forma, todos os casos de teste que forem considerados incluídos formarão o conjunto selecionado de testes para regressão.

Como oráculo para essa abordagem são utilizados os resultados da execução dos testes na versão base. Deste modo, se o sistema sob teste fornece os mesmos resultados na execução dos testes de regressão em relação àqueles gerados na execução dos testes na versão base, o teste passou.

Os testes terminam quando todos os casos de teste que não passaram revelam falhas cuja presença e severidade são aceitáveis, com os demais testes apresentando execução com sucesso.

O modelo de alteração de código básico não considera fluxo de dados, fluxo de controle, ou outras dependências. Também não considera explicitamente os efeitos de herança e ligação dinâmica.

- Inclusão: todos os testes da versão original que podem produzir resultados diferentes são selecionados, portanto Reteste baseado no código modificado é seguro.

- Precisão: embora poucos testes que poderiam ser omitidos sejam incluídos, essa abordagem é considerada a mais precisa das estratégias de regressão parcial dentre os testes caixa-branca (Harrold & Rothermel 1998), uma vez que os testes originais, que exercitam códigos não alterados, não são selecionados para o reteste. Essa característica pode reduzir significativamente o custo dos testes em relação a extensão das alterações.

- Eficiência: a computação está relacionada com a ordem de tamanho do conjunto de testes original vezes maior número de segmentos modificados entre o original ou o componente beta. Já o custo direto da análise e configuração pode ser menor caso seja automatizado.

- Generalidade: Reteste baseado em segmentos modificados pode ser aplicado no escopo de classe e requer que o programador tenha habilidades para analisar o código e ferramentas relacionadas.

3.6.5 Reteste no *firewall*

Usa análise de dependência de código para selecionar conjunto de testes para regressão parcial e executa esse subconjunto na versão beta. Procura responder a seguinte questão: como um subconjunto do conjunto de testes original pode ser selecionado utilizando dependência de código?

Esse padrão é aplicável em escopo de classe, cluster ou subsistema, depois da manutenção, para reuso e durante o desenvolvimento iterativo.

Um *firewall* é um conjunto de componentes cujos casos de testes serão incluídos num teste de regressão. É considerado um limite imaginário que engloba o conjunto de módulos que devem ser retestados, em função das alterações que as classes ou objetos sofreram.

Um conjunto *firewall* é identificado pela análise das alterações que cada componente do sistema sob teste e das suas dependências com outros componentes. Cada par de componentes, **A** e **B**, é analisado, pois um dos dois ou ambos podem ter sofrido mudanças. Uma mudança pode ser de contrato ou de implementação:

- Uma mudança de contrato altera a interface externa de um componente e/ou o contrato visível externamente de uma classe.

- Uma mudança de implementação são as demais: todas as outras alterações na implementação de uma classe não são visíveis para os clientes dadas as regras de encapsulamento de uma linguagem particular usada pelo sistema sob teste. Uma alteração de implementação não afeta a interface externa ou o contrato de uma classe visível externamente.

A dependência de relacionamento entre cada par de componentes é usada para selecionar casos de teste do conjunto de testes original, segundo (Binder 2000a). A expressão **B** *usa* **A** refere-se a um dos quatro relacionamentos entre códigos:

1. **B** é um cliente de **A**, ou seja, um objeto do tipo **A** é declarado em **B**;

2. **B** tem um ponteiro para **A**, possivelmente usado para acessar membros de uma coleção;

3. **B** usa um objeto da classe **A** como um argumento num parâmetro de um método;

4. **B** é uma classe genérica e usa **A** como um tipo de parâmetro.

O critério de entrada e saída desse padrão é idêntico ao dos outros padrões apresentados.

O oráculo também é obtido a partir dos resultados obtidos quando da execução do conjunto de testes na versão base. Logo, se o sistema sobre teste produz os mesmos resultados na execução dos testes de regressão em relação àqueles apresentados quando foi executado na versão base, os testes passam.

São conseqüências do padrão de reteste no *firewall*:

- Inclusão: todos os testes da versão original que podem produzir resultados diferentes são selecionados, portanto, Reteste no *firewall* é seguro.
- Precisão: poucos testes que poderiam ser omitidos são incluídos.
- Eficiência: no pior caso, a computação é da ordem de tamanho do conjunto de testes original vezes o maior número de componentes entre o original e o componente beta. Já o custo direto da análise e configuração pode ser menor, se automatizado, porém as ferramentas também tem um custo indireto. Reteste no *firewall* não garante qualquer tempo ou redução de custo.
- Generalidade: Reteste no *firewall* é aplicável no escopo de classe.

Na tabela 3.1 são comparados o custo relativo e o risco das abordagens de teste de regressão descritas nessa seção.

As abordagens Reteste baseada em casos de uso de maior risco e Reteste por perfil são estratégias caixa-preta, onde a seleção do conjunto de testes é feita pela análise de relações independente de implementação, requisitos e capacidades, produzindo um conjunto de testes não seguro.

Já as abordagens Reteste de segmentos modificados e Reteste no *firewall* são estratégias caixa-branca para regressão parcial que requerem análise de dependência de implementação e produzem conjuntos de testes seguros.

Algumas das atividades apresentadas na tabela merecem um esclarecimento: *Remoção de casos de testes obsoletos* envolve a manutenção do conjunto de testes; e *Análise de dependências* é relativa as abordagens que consideram a dependência entre componentes e o custo gerado por esta atividade.

3.7 Limitações dos Testes de Regressão

Teste de regressão também possui suas limitações, pois não é um método autosuficiente para testar um sistema.

Atividade de Teste de Regressão	Custo e Risco das Abordagens de Teste de Regressão				
	Reteste total	Reteste baseado em caso de uso de alto risco	Reteste baseado em perfil operacional	Reteste com firewall	Reteste baseado em código modificado
Escopo	Sistema	Sistema	Sistema	Cluster	Classe
Remoção de casos de testes obsoletos	baixíssimo custo	baixíssimo custo	baixíssimo custo	baixíssimo custo	baixíssimo custo
Análise de dependências	–	–	–	baixíssimo à médio custo	baixíssimo à médio custo
Remoção de casos de teste obsoletos	–	–	–	baixíssimo à mais alto custo	baixíssimo à mais alto custo
Remoção de casos de teste redundantes	–	–	–	baixíssimo à mais alto custo	baixíssimo à mais alto custo
Configuração da execução dos testes	baixíssimo custo	baixo custo	baixo custo	baixo custo	baixo custo
Custo do projeto de teste	mais baixo	baixo	baixo	mais alto	alto
Custo de execução de teste	mais alto	orçamento limitado	orçamento limitado	tamanho do firewall	tamanho de beta
Avaliação de nenhum teste passado	baixo custo	menor custo	menor custo	menor custo	menor custo
Risco de omissão	mais baixo	moderado	moderado	baixo	baixo
Seguro?	sim	não	não	sim	sim

Tabela 3.1: Comparação de custos e riscos das abordagens de teste de regressão.

Teste de regressão não diminui a necessidade de desenvolvimento e execução de testes de funcionalidades novas ou alteradas no escopo de componente ou sistema (Binder 2000a).

Quando um conjunto de testes pode ser reusado como um conjunto de testes de regressão, esse conjunto não será mais eficaz como conjunto de testes primário.

Um conjunto de testes de regressão não tem a mesma meta de cobertura do que os testes primários, dessa forma, teste de regressão não deve ser usado como único meio para testar um software.

Além disso, o uso de um conjunto de testes primário inadequado como conjunto de testes de regressão não vai melhorar a eficiência do conjunto de testes. Se os testes primários não conseguem uma cobertura adequada, então re-executar os testes como conjunto de testes de regressão não deve conseguir cobertura mais alta.

Capítulo 4

Trabalhos Relacionados

Neste capítulo são apresentados alguns trabalhos apresentados na literatura que abordam propostas de técnicas seletivas para teste de regressão, e que também apresentam preocupações com a melhoria da testabilidade de software orientados a objetos no que se refere ao reteste desses sistemas.

No final, será apresentada uma tabela com a comparação dos trabalhos descritos.

4.1 Chen, Probert & Sims

No trabalho de Chen, Probert & Sims (2002) é apresentado um método para seleção de teste de regressão baseado na especificação e em análise de risco. Para representar os requisitos baseados nas características e comportamentos desejados pelo cliente os autores utilizam o Diagrama de Atividades (da UML - *Unified Modeling Language*). O Diagrama de Atividades é bastante usado para projeto de sistemas, e mostra-se bom para descrever comportamento de sistemas e fluxos de trabalhos.

É apresentado um processo para identificar os casos de teste afetados para a realização do teste de regressão que foi classificado em dois tipos: (1) testes dirigidos (*Targeted tests*) que garantem que requisitos importantes exigidas pelo cliente ainda são adequadamente atendidas pela nova versão do sistema; e (2) testes de segurança (*Safety tests*) direcionados a risco, visam garantir que áreas com potencial problemas sejam devidamente testadas.

No teste baseado em especificação é necessário conhecer quais casos de teste verificam um dado requisito. Os casos de teste podem ser designados com base nos requisitos representados no diagrama de atividades, dado que cada caso de teste corresponde um caminho específico no grafo representado pelo diagrama de atividades.

Uma matriz de rastreabilidade (que relaciona casos de teste e requisitos) deve ser montada de forma que para cada nó ou arco do diagrama de atividades sejam listados os casos de teste que o executam.

Dependendo do tipo de alteração no sistema: de especificação ou de código, verifica-se se o comportamento do sistema sofreu alteração e conseqüentemente se o diagrama de atividades precisa ser atualizado para servir de entrada para a seleção do conjunto de teste de regressão. Somente quando alterações de especificação são identificadas, o diagrama de atividade sofre atualização para refletir o novo comportamento do sistema.

A técnica de seleção de teste de regressão apresentada é baseada no diagrama de atividades que é semelhante a um grafo de fluxo de controle (CFG), porém os nós descrevem atividades ao invés de instruções como no CFG. Dessa forma, os autores aplicam o algoritmo baseado em grafo de fluxo de controle proposto por (Rothermel et al. 2000) e depois que todos os arcos afetados são identificados, eles são usados para selecionar os casos de teste (testes dirigidos).

Já para selecionar os casos de teste para os Testes de Segurança é realizada uma análise de risco. O modelo de análise de risco é utilizado para medir quantitativamente a qualidade do conjunto de teste, complementando os testes dirigidos; servem para garantir que o sistema não seja entregue com falhas residuais que levem a defeitos graves em fase de operação.

Para essa técnica, que é orientada a cliente e também baseada em risco, o conjunto final de teste de regressão será composto pela união dos Testes Dirigidos e os Testes de Segurança.

4.2 Rothermel, Harrold & Dedhia

O trabalho proposto por Rothermel et al. (2000) apresenta uma técnica de seleção de teste de regressão para software orientado a objetos, mais especificamente para software desenvolvido em linguagem C++. A técnica propõe a construção de um grafo (fluxo de controle) para representar um software, e faz uso dele para selecionar os testes, do conjunto de teste original, que executam código que foi alterado na nova versão do software.

Trata-se de uma técnica estritamente baseada em código que seleciona testes usando as informações reunidas com a análise do código, e não requer nenhum conhecimento do método usado para especificar ou testar inicialmente o software.

O grafo de fluxo de controle (CFG) para um método P contém um nó para cada instrução simples ou condicional em P e os arcos entre nós representam o fluxo de controle entre as instruções.

Para a representação de um fluxo de controle que representa as interações entre métodos que possuem um ponto de entrada único, foi proposto o grafo de fluxo de controle interprocedural (ICFG). Um ICFG contém um grafo de fluxo de controle para cada método em P, com cada local de chamada de um novo método representado com um par de arcos denominados *call* e *return*. Cada nó de chamada é conectado ao nó de entrada

do método chamado pelo arco *call*, e cada nó de saída é conectado ao nó de retorno do método que chama por um arco *return*.

Uma classe (método ou programa) deve ser instrumentada para que o caminho percorrido no grafo ao executar um caso de teste seja guardado, formando uma tabela que relaciona a execução de um teste com os arcos que são atravessados por ele.

O modelo de teste de regressão apresentado no trabalho considera duas fases distintas de teste de regressão: fase preliminar (começa durante as primeiras fases de desenvolvimento do software), e fase crítica (quando realmente começa a fase de teste de regressão, uma versão do software está disponível no ambiente para testes).

Para a seleção do conjunto de teste de regressão foi proposto um algoritmo similiar ao apresentado no capítulo 6, ilustrado na figura 6.9.

A seleção de testes de regressão para classes modificadas ou derivadas em C++ é feita através de um grafo de fluxo de controle de classe (CCFG), que é uma coleção de grafos de fluxos de controle individuais para os métodos na classe, no qual os locais de chamadas são expandidos e conectados em chamadas de métodos da mesma forma que em ICFG. Além disso, é adicionado uma estrutura (abstração de um programa driver) ao ICFG para formar o CCFG. Essa estrutura é composta inicialmente por nove vértices que depois são conectados aos grafos de fluxo de controle de cada método da classe.

A mesma representação, CCFG, é utilizada para classes derivadas. Essencialmente, cada CCFGs herda o CFGs das classes bases, incluindo novos CFGs para representar métodos novos ou redefinidos.

Após a criação do grafo que representa as classes, o algoritmo é aplicado para encontrar o conjunto de teste de regressão para elas.

Essa técnica é aplicável para classes modificadas e derivadas, e para programas aplicativos que usam as classes modificadas, pois seleciona casos de teste do conjunto de teste existente que executa código modificado em C++.

4.3 Harrold et. al

Nesse trabalho foi proposta uma técnica de seleção de teste de regressão segura que, baseada no uso de uma representação adequada, manipula as características da linguagem Java (Harrold & et. al 2001), tais como polimorfismo, ligação dinâmica e manipulação de exceção.

Foi adaptada da técnica baseada em algoritmo de caminho em grafo discutido em (Harrold & Rothermel 1997) e (Rothermel et al. 2000), onde é utilizada uma representação baseada no fluxo de controle da versão original e modificada do software para seleção de casos de teste a serem reaplicados.

O grafo proposto nessa nova técnica representa eficientemente características da linguagem Java; o algoritmo de caminho no grafo seguramente seleciona todos os casos de teste no conjunto de teste original que podem revelar falhas no programa modificado.

Outra novidade dessa técnica é que o modelo representa (sobre certas condições) os efeitos de partes não analisadas do software, como bibliotecas (*libraries*), possibilitando o uso para uma seleção segura do conjunto de teste de regressão de aplicações sem necessitar de análise completa de bibliotecas que esses aplicativos utilizam, por considerar-se que as bibliotecas são externas, embora utilizadas pelo programa sobre teste, elas não fazem parte das modificações.

Para representar de forma adequada um programa em Java, foi proposta uma extensão ao grafo de fluxo de controle para um procedimento simples - CFG (descrito na seção 4.2). A nova representação gráfica denominada *Java Interclass Graph* - JIG permite representar programas escritos em outras linguagens orientadas a objetos que não apenas Java. O JIG manipula cinco tipos de características de Java: (1) informações de tipo de variáveis e objetos; (2) métodos internos e externos (aqueles pertencentes a métodos de classes não analisadas pela estratégia); (3) interações interprocedurais como chamadas para métodos internos ou externos a partir de métodos internos; (4) interações interprocedurais como chamadas para métodos internos a partir de métodos externos; e (5) manipulação de exceção.

O algoritmo para percorrer o JIG e identificar os arcos afetados é similar ao apresentado no trabalho (Rothermel et al. 2000) que caminha num CFG de um procedimento, que foi apresentado na seção 4.2.

Nesse trabalho também é descrito o sistema RETEST (*Regression-Test-Selection System*) que implementa a técnica proposta para seleção de teste de regressão para Java. O RETEST é formado por três componentes principais: *Profiler*, que colhe informações dinâmicas; um componente de análise estática denominado *DejaVOO* para identificar as partes afetadas pelas modificações ao comparar duas versões de um programa, e um componente de seleção de teste *SelectTests*.

Com o uso do sistema RETEST os autores demonstraram uma redução significativa no conjunto de teste de regressão para os estudos realizados.

4.4 Granja & Jino

No trabalho de Granja & Jino (1999) são discutidas duas técnicas de teste de regressão: uma técnica seletiva que identifica as modificações estruturais e seleciona o conjunto de casos de teste que as exercitam (baseada na família de critérios de potenciais-usos); e a segunda técnica engloba procedimentos para atividades de teste e de teste de regressão.

Uma família de critérios de teste estrutural Potencial-uso é baseada na análise de fluxo

de dados; o que implica na análise entre um requisito de associação ou caminhos entre uma definição e os usos (referências explícitas) de uma variável a ser exercitada.

Para a seleção de teste de regressão a técnica baseada em critérios potencial-uso identifica as modificações no programa: alterações de fluxo de controle, alterações de fluxo de dados e alterações que não afetam dados ou fluxo de controle.

A seleção dos elementos requeridos para o conjunto de teste de regressão é realizado em duas fases: na primeira, é determinado o escopo de modificações pelo critério de potencial-uso; e na segunda, os elementos requeridos do programa modificado são classificados e é estabelecida a equivalência ou equivalência-potencial em relação ao programa original, criando três grupos de elementos requeridos: preservados, retestáveis e removidos.

Para a seleção de T' em relação ao conjunto original T, a técnica classifica cada caso de teste original em um dos três conjuntos conforme um dado critério c: reusável, retestável ou obsoleto. Cada um dos conjuntos são baseados na classificação prévia dos elementos requeridos. O conjunto inicial de teste de regressão T' será composto apenas pelo conjunto classificado como retestável. Já que apenas o conjunto de casos de teste retestável exercitam pelo menos um elemento requerido retestável no programa modificado, logo ao ser aplicado no programa modificado tem chances de produzir resultados diferentes do programa original.

Após a execução de T' para o programa modificado, seus resultados são analisados de acordo com a especificação do programa modificado.

Quando necessário, novos casos de teste T'' são projetados e executados. E depois os resultados também devem ser analisados.

Como última atividade do processo seletivo deve ser realizada a manutenção do conjunto de teste de regressão. Isso deve ser feito de forma que o conjunto final seja composto pelos casos de teste reusáveis (T - T') mais o novo conjunto de casos de testes Novos e Retestáveis (T' + T''). Os casos de teste obsoletos e redundantes devem ser removidos do conjunto de teste.

A técnica proposta tem por objetivo selecionar um conjunto de casos de teste reusável, para investigar as possíveis funcionalidades modificadas de um programa e fornecer uma boa cobertura estrutural dos elementos requeridos.

Cada caso de teste do programa original passa a ser representado por uma tupla-3 $< TC, RE(c), F >$ composta por: um caso de teste TC, um conjunto de elementos requeridos $RE(c)$ executados pelo caso de teste dado um critério c, e F é a funcionalidade testada por TC. $RE(c)$ contém todos os elementos estruturais requeridos exercitados por TC para examinar a funcionalidade F.

Assim, para cada tupla-3, a funcionalidade F de um programa original é preservada se cada elemento requerido em $RE(c)$ tem um equivalente no programa modificado que pertence aos elementos requeridos preservados. Da mesma forma, uma funcionalidade F

é possivelmente modificada se houver, pelo menos, um elemento requerido em *RE(c)* que possui seus equivalentes-potenciais no programa modificado classificado como retestável. Uma funcionalidade F é removida, se todos os elementos requeridos em *RE(c)* são classificados como removido no programa modificado.

A classificação das funcionalidades dos programas modificados é a tarefa mais importante nessa técnica. Além disso, faz-se necessário estabelecer uma relação entre cada funcionalidade e seus casos de teste para manter os testes de regressão.

Os autores destacaram que muitos fatores podem influenciar no resultado dessa técnica, pois o tipo de modificação e a localização da modificação no programa, além do modelo do grafo de fluxo de controle (expandido ou contraído) podem influenciar no tamanho dos elementos requeridos retestáveis e no conjunto de casos de teste reusável.

4.5 Wang, King & Wickburg

Em Wang, King & Wickburg (1999) foram identificados tipos de manutenção de software e classificados no escopo de manutenção de software baseados em componentes. Um método prático baseado em mecanismos de teste embutido foi proposto para o desenvolvimento de software fácil de manter (*maintanable*), a partir da implementação de componentes BIT.

Teste embutido (*Built-in Test*), segundo os autores, é um novo tipo de teste de software orientada a manutenção, o qual é explicitamente descrito no código fonte do software como funções membros para realçar a característica de manutenção do software. O teste de software convencional está focado na geração de teste para sistemas existentes, enquanto o método BIT preocupa-se em construir testabilidade para o sistema, por isso o teste e manutenção do software devem ser contidos no próprio software.

O software voltado para manutenção pode ser executado de dois modos: normal e de manutenção/teste. Na execução normal, o software tem o mesmo comportamento do componente original, o que ocorre quando apenas funções membros normais são chamadas. Na execução em modo de manutenção e teste, os mecanismos BIT são ativados através de chamadas de funções membros BIT incorporadas ao componente.

Um protótipo de objeto com teste embutido é desenvolvido como mostrado na figura 4.1, onde as declarações de teste são incluídas na interface do objeto e os casos de teste na sua implementação, tornando-se padrões da estrutura do objeto.

A principal característica dessa técnica é que os casos de teste podem ser herdados e reutilizados do mesmo modo que qualquer outro código em sistemas orientados a objetos convencionais. Assim, os mecanismos BIT podem ser reutilizados toda vez que houver necessidade de reteste e manutenção do componente, além da possibilidade de reuso e herança desses mecanismos pelas classes derivadas. Isto é possível porque os mecanismos

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

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

Figura 4.1: Um objeto com mecanismos BIT.

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 (Wang, King, Court, Ross & Staples 1997). Existe apenas a necessidade de incorporar novos testes embutidos à classe derivada para testar as alterações realizadas.

Os benefícios de mecanismos de BITs para manutenção auto-contidos, tanto manutenção de software corretiva, adaptativa, perfeccionista, preventiva e reengenharia, é que esses processos podem ser significativamente simplificados (Wang et al. 1999).

4.6 Kung et. al

No trabalho proposto em (Kung et al. 1996), foi definido um método para identificar os tipos de alterações de classes em programas orientados a objetos a partir de um grafo de relação de objetos. O grafo retrata as relações de herança, agregação e associação existentes no programa orientado a objetos a ser testado.

O modelo de teste de regressão foi desenvolvido para capturar e representar relações complexas e interdependências entre várias partes de um programa em C++. Ele consiste em três tipos de diagramas:

1. diagrama de relação de objetos (ORD - *Object Relation Diagram*), que facilita o entendimento de relações de herança, agregação e associação entre classes e suas dependências. É usado para identificar as classes afetadas pela alteração de uma ou mais classes e gerar uma ordem de teste com custo eficaz para testar as classes afetadas.

2. diagrama de ramificações de blocos (BBD - *Block Branch Diagram*), que é usado para facilitar o entendimento da interface e estrutura de controle de uma função membro de uma classe, bem como suas relações com outros ítens de dados (como dados globais, dados de classe) e funções membros de classes.

3. diagrama de estados de objetos (OSD - *Object State Diagram*), é projetado para capturar o comportamento dinâmico do objeto de uma classe, incluindo estados do objeto e suas transições.

No trabalho é descrito como as alterações possíveis em programas orientados a objetos (alterações de dados, de métodos, de classe e bibliotecas de classes) podem ser identificados e representados.

A técnica proposta, denominada técnica ORD, instrumenta o código para reportar as classes exercitadas pelos casos de teste.

Após uma alteração ser realizada num programa OO, os testes de regressão envolvem reteste de componentes em diferentes níveis: membros de classes, classe/objeto, subsistema e sistemas. Porém é necessário identificar aqueles componentes afetados pela modificação para realizar o reteste, de forma a diminuir o esforço e tempo do teste de regressão. Os autores definiram um "*firewall*" para uma classe C como sendo um conjunto de classes que diretamente são dependentes de C ou que podem ser afetadas pelas alterações em C (em virtude de herança, agregação ou associação). Quando a classe C é modificada, a técnica ORD seleciona todos os testes que foram determinados através da instrumentação para exercitar uma ou mais classes do "*firewall*" para C.

Depois da identificação do "*firewall*" da classe alterada, o testador precisa retestar cada classe a nível de unidade e de integração. Porém, para teste de regressão o número de *stubs* e *drivers* necessários para realizar testes nesses níveis exige um grande trabalho e o custo é alto. Nesse trabalho foi proposto um algoritmo que usa uma estratégia de teste denominada Ordem de Teste (*test order*) que serve como um mapa detalhado no reteste da classe e no teste de reintegração para a redução do custo de teste.

O algoritmo seleciona todos os testes associados às classes do "*firewall*" independente dos testes efetivamente executar o código alterado, ou código que tem acesso às alterações de dados do objeto.

O algoritmo proposto para gerar a ordem dos testes pode ser aplicado em grafos acíclicos ou cíclicos.

4.7 Jang et. al

No trabalho de Jang, Chae, Kwon & Bae (1998) é proposta uma estratégia para análise de impacto das alterações na hierarquia de classes a fim de diminuir o esforço e o tempo para

retestar programas modificados pela seleção somente das partes que podem ser afetadas pelas modificações. Para a identificação das partes a serem retestadas é utilizada a análise de impacto das alterações.

Esta estratégia é baseada em *firewall* de classes, e objetiva reduzir significativamente o esforço de reteste por considerar funções membros como um teste de unidade.

Primeiramente são definidos casos base para tratar dos impactos das modificações associadas com várias características de orientação a objetos, tais como: herança, ligação dinâmica e encapsulamento, e são relacionados a um *firewall* correspondente para cada caso base. Depois é realizada a classificação dos tipos de alteração que podem ocorrer em cada nível de um dado, uma função membro, uma classe e uma relação de herança e é construído um *firewall* para cada tipo utilizando estes casos base.

As relações de dependência dos membros das classes são ilustradas num grafo de dependência de membros (*Member dependancy graph* - MDG) que é uma extensão do grafo de classes proposto em (Harrold et al. 1992). Porém, neste trabalho são cobertas mais características de orientação a objetos e investiga os vários níveis de alteração, conseguindo identificar mais precisamente as partes afetadas pelas alterações. No MDG são adicionados nós para representar uma classe e funções membro virtuais/não virtuais e arcos para representar a relação de herança, dependência de controle, definição de dados e dependência de uso.

Para cada tipo de alteração das funções membro, dados, classes e relação de herança numa hierarquia de classes é definido um *firewall* correspondente a cada tipo usado nos casos base previamente desenvolvidos. Para a construção de cada um dos *firewall* é proposto um algoritmo. Um firewall pode ser de unidade ou de integração e define se os testes a serem reaplicados devem ser de unidade ou integração para a característica alterada.

4.8 Martimiano

O trabalho de Martimiano (1999) apresenta estudos empíricos e comparativos utilizando as técnicas de teste de regressão propostas em (Wong, Maldonado & Delamaro 1997, Wong, Horgan, London & Bellcore 1997): técnica baseada em modificação e técnica baseada em mutação seletiva.

A técnica baseada em modificação tem por objetivo selecionar aqueles casos de teste que revelam um comportamento diferente entre o programa modificado e o original. Essa técnica sugere que seja encontrado o conjunto de testes **T"** que é um subconjunto de **T** baseado em modificações. A partir de **T"** são aplicados os mecanismos de minimização (seleciona subconjunto preservando a cobertura em relação a um dado critério a partir de **T**) e mecanismos de prioridades (seleciona subconjunto realizando uma análise de cada caso de teste segundo um critério selecionado). Com a aplicação desses mecanismos é

buscada a diminuição do conjunto T" utilizando duas abordagens distintas, no final serão dois subconjuntos de casos de teste de regressão: o mínimo e o priorizado.

Dessa forma, a utilização da técnica baseada em modificações, juntamente com a minimização e priorização, permite a escolha de um conjunto eficaz para a realização dos testes de regressão com um menor custo. Para avaliar essa técnica foram realizados experimentos utilizando o programa *SPACE* e a ferramenta de teste *ATAC*.

A Técnica de Teste de Regressão baseada em mutação seletiva utiliza mutação seletiva por meio de um conjunto de operadores de mutação (que introduzem pequenos erros sintáticos), visando encontrar um subconjunto de casos de teste a ser aplicado durante os testes de regressão. O objetivo da técnica é examinar como um critério de teste pode ser utilizado no teste de regressão para ajudar a determinar quais casos de teste devem ser selecionados ou que possuem maior prioridade. A solução apresentada por essa técnica depende do programa original, do conjunto de teste de regressão e do conjunto de operadores de mutação, dessa forma, a seleção do conjunto de teste pode ser realizada por um processo *off-line* antes que qualquer modificação seja realizada.

A principal estratégia utilizada por esta técnica é examinar como um critério **C** pode ser utilizado no teste de regressão a fim de auxiliar na determinação de quais casos de teste devem ser selecionados ou têm maior prioridade para revalidar as funcionalidades herdadas da versão anterior do sistema, e quais devem ser omitidos ou têm baixa prioridade. Estudos empíricos publicados na literatura demonstram que a mutação seletiva é bastante eficiente em revelar defeitos e pode contribuir para a redução dos custos aplicados nas atividades de teste. Para avaliar a técnica foram realizados experimentos utilizando um conjunto de programas utilitários do UNIX, como: Cal, Checkeq, Col, Comm, etc., e a ferramenta *PROTEUM*, utilizando doze dos seus operadores de mutação.

Com a aplicação das técnicas, Martimiano (1999) concluiu que técnicas que selecionam casos de teste que exercitam modificações são capazes de prover uma maior redução no conjunto de casos de teste original, como é o caso da técnica baseada em Modificação. A técnica baseada em Mutação Seletiva obteve bons resultados de redução de tamanho do conjunto de casos de teste sem afetar a eficácia desse conjunto em revelar defeitos.

Outro ponto importante que foi levantado é que os mecanismos de priorização são eficazes em revelar defeitos reduzindo bastante o número de casos de teste a serem reexecutados. E estes podem ser combinados com qualquer técnica de teste de regressão, assim como os mecanismos de minimização.

4.9 Comparação dos Trabalhos

Esta seção apresenta uma comparação dos trabalhos descritos nas seções anteriores incluindo o trabalho apresentado nessa dissertação. A tabela 4.1 apresenta um resumo das

características desses trabalhos com relação: a linguagem coberta, em que a estratégia de teste de regressão é baseada (especificação, código, outro), se é possível a automatização da estratégia de teste de regressão e se é previsto reuso de testes da superclasse.

Trabalhos	Linguagem	Estratégia baseada em:	Automatização da estratégia	Reutilização de testes da superclasse
Chen et. al	Qualquer OO	Especificação	–	–
Rothermel, Harrold & Dedhia	C++	Código	Permite	Os casos de teste da superclasse são herdados pela subclasse, mas não utilizados para os testes da nova versão da superclasse.
Harrold et. al	Java	Código	Sistema RETEST	Reutiliza testes para a superclasse.
Granja & Jino	Não OO	Código	POKE-TOOL RePoke-Tool	Não trata linguagens orientadas a objetos.
Wang, King & Wickburg	Java, C++	Código	–	Como o mecanismo BIT é incluído na classe, ele também é herdado.
Kung et. al	C++	Código	Test order generation	–
Jang et. al	Qualquer OO	Código	Permite	Não.
Martimiano	C	Código	ATAC PROTEUM	–
Vieira	C++	Especificação	Permite	Os testes da superclasse são reutilizados.

Tabela 4.1: Comparação dos trabalhos.

Capítulo 5

Testabilidade e Autoteste

O custo com a realização de testes numa organização chega a 40% do esforço de desenvolvimento, segundo (Pressman 1997a). Por isso, a grande necessidade de produzir sistemas, componentes ou, até mesmo, classes que sejam cada vez mais testáveis a fim de diminuir o custo com os testes.

Neste capítulo serão abordados os principais aspectos de Testabilidade de Software Orientado a Objetos e abordada a necessidade crescente de Automação de Testes. Inclusive serão mostradas características de classes autotestáveis e de uma ferramenta desenvolvida para auxiliar nos testes de sistemas em linguagem C++ - ConCAT.

5.1 Testabilidade

A testabilidade de um software pode ser definida através da medida de quão fácil é satisfazer aos critérios de teste pré-estabelecidos, sendo que o grau de testabilidade de um componente ou sistema tem influência direta na quantidade de esforços gastos na fase de testes.

Uma definição mais formal de testabilidade de software foi definida pelo padrão IEEE 610-12/90 como sendo:

- em que medida um sistema facilita o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos;
- em que medida a definição de requisitos é representado de modo a permitir o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram satisfeitos.

Para que um software tenha uma boa testabilidade deve-se levar em consideração a avaliação das seguintes características (Pressman 1997a): operabilidade (*operability*),

observabilidade (*observability*), controlabilidade (*controllability*), ser decomponível (*decomposability*), simplicidade (*simplicity*), estabilidade (*stability*) e ser compreensível (*understandability*).

- Operabilidade: facilidade de testar eficientemente um sistema que apresenta poucos erros de análise e relatados no processo de teste;
- Observabilidade: facilidade em se determinar o quanto as entradas fornecidas afetam as saídas obtidas;
- Controlabilidade: facilidade em produzir a saída desejada a partir da entrada fornecida;
- Ser decomponível: facilidade em controlar o escopo de teste, de forma a isolar rapidamente os problemas e executar um melhor reteste;
- Simplicidade: quanto maior a simplicidade funcional, estrutural e de codificação, mais rapidamente um sistema pode ser testado;
- Estabilidade: para avaliar a estabilidade do sistema é necessário verificar se as alterações no software são: infreqüentes, controladas e não invalidam os casos de teste existentes, além da facilidade do software em recuperar-se de defeitos;
- Ser compreensível: quanto mais informações (de projeto, dependências, documentação) houverem sobre o software melhor serão os testes.

Um sistema testável requer menos esforço para a realização de testes e também melhora a eficiência dos testes (potencial para encontrar falhas).

O conceito de testabilidade tem um importante papel no projeto e manufatura de circuitos integrados (chips). O teste de circuitos é feito para detectar falhas no processo de manufatura. Enquanto que o teste de software procura falhas no projeto de desenvolvimento do software.

São três as principais abordagens para o projeto para testabilidade em circuitos integrados (Binder 1994):

- Soluções ad hoc: são soluções específicas para um componente;
- Abordagem estruturada: envolve o estabelecimento de regras de projeto que facilitem os testes;

- Teste embutido ou BIT (*Built-in Test*): envolve a utilização de circuitos padrões para testes, de modo que seja possível colocar o circuito ou placa em modo teste, transmitir as strings de bits de entrada e capturar strings de bits de saída. As capacidades básicas de BIT foram estendidas para fornecer a geração automática de seqüências de testes dando origem a abordagem BIST (*Built-in Self Test*).

Testabilidade apresenta algumas particularidades quando o sistema a ser avaliado é um sistema orientado a objetos. Na seção 5.1.2 serão mostrados como os conceitos de teste aplicados em hardware podem ser aplicados em sistemas OO.

5.1.1 Pontos que afetam a testabilidade

Segundo Binder (1994), uma boa testabilidade é resultado de uma boa prática de engenharia e um processo de desenvolvimento bem definido. Na construção de software visando testabilidade, alguns aspectos básicos devem ser observados, pois afetam diretamente a melhoria da testabilidade:

Representação

Refere-se as informações fornecidas para um software. A representação de sistemas é essencial para a realização dos testes, seja ela em linguagem natural ou especificação formal. Sem uma representação definida, é praticamente impossível testar um sistema, já que não se pode determinar o comportamento desejado para validar os testes realizados. Se o software possui requisitos bem definidos e uma documentação atualizada do projeto, isto contribui para a testabilidade do software. Dentre os tipos de informações que podem ser apresentadas estão: informações destinadas a usuários (como manuais de uso, manuais de referência) e informações destinadas aos desenvolvedores do sistema (como especificações de análise e projeto, documentação de testes e manutenção). Além de documentos que facilitam o entendimento do software: código fonte e elementos de suporte, como manuais de instalação e relatórios de qualidade.

Implementação

A estrutura do sistema é um aspecto importante para determinar a testabilidade de um software. Por isso, a forma como o sistema foi implementado pode dificultar os testes e diminuir a testabilidade.

Classes altamente acopladas são exemplos típicos de uma estrutura que dificulta o controle da classe sob teste, e conseqüentemente diminui a testabilidade. O uso de encapsulamento possibilita que a propagação de mudanças seja reduzida entre classes e métodos, pois permite que um objeto esconda seu estado dos demais. Já a utilização de herança

provoca o aumento de testes, devido à necessidade de testes de métodos herdados de uma classe principal quando esta sofre alteração. Quando as duas características (encapsulamento e herança) são utilizadas juntas, a ocorrência de encapsulamento ocasiona uma baixa observabilidade, pois externamente não se tem acesso ao estado interno do objeto, dificultando assim os testes e diminuindo a testabilidade. Já módulos com funcionalidades bem específicas facilitam a realização dos testes, pois melhoram o controle do escopo dos testes, podendo-se isolar e detectar mais rapidamente as falhas.

Capacidade de teste embutido

A capacidade de teste embutido (do inglês, *Built-In Test - BIT*) fornece separação explícita entre os testes e a funcionalidade da aplicação, através da inserção de mecanismos de testes no componente. Os mecanismos de testes são: métodos *set/reset*, métodos relatores e assertivas (Binder 1994). Os métodos *set/reset* permitem colocar o sistema num estado pré-definido, independente do seu estado corrente. Os métodos relatores observam e armazenam o estado interno de um programa durante a execução dos testes. Uma assertiva é um teste sobre todo ou parte do estado de um programa em execução, dado pelo valor de suas variáveis. As assertivas são utilizadas para testar uma condição referente ao estado de um objeto e são compostas por condições e por uma rotina de exceção associada que trata das condições não satisfeitas, o que indica que uma das computações realizadas previamente e das quais o valor da variável depende, podem estar incorretas. Assim, estes mecanismos podem contribuir para aumentar a testabilidade por proporcionar a capacidade de observação embutida no componente em teste.

Seqüência de teste

Uma seqüência de teste é composta por uma coleção de casos de teste para o componente e a estratégia utilizada para executá-los. A existência de uma seqüência de teste é fundamental, juntamente com a existência de um oráculo e uma boa documentação dos testes, para facilitar a aplicação dos testes.

Ferramentas de teste

Com a ausência de ferramentas automatizadas para a realização dos testes, menos testes serão realizados, e portanto maior será o custo dos testes para que um dado objetivo de confiabilidade seja alcançado, causando a diminuição da testabilidade.

Processo de desenvolvimento de software

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

A partir dessas características, pode-se concluir que um sistema testável deve apresentar uma boa documentação, inclusive planos e casos de teste, e também formas para melhorar e facilitar o controle as entradas e saídas de cada componente, por exemplo: o mecanismo BIT.

5.1.2 Testabilidade em Sistemas Orientados a Objetos

Em sistemas orientados a objetos, o conceito de projeto de teste visando testabilidade foi introduzido para permitir a construção de classes testáveis. Uma classe testável é aquela que contém, além da sua implementação, uma especificação de teste e métodos padrões para os testes, que permitem melhorar a controlabilidade e a observabilidade da classe e apóiam a geração e análise automática de testes.

Os componentes extras necessários para tornar uma classe testável[1] são (Toyota & Martins 1999): especificação de testes, usada na geração dos testes; mecanismos embutidos de teste ou BIT, usados no controle e observação do estado interno e na análise dos resultados; e um *driver* (programa que executa os casos de teste sobre a classe e retorna os resultados), que gera os testes a partir da especificação embutida na classe, executa e avalia os testes, utilizando os procedimentos de testes embutidos.

O uso de autoteste é muito adequado tanto para os testes aplicados durante o desenvolvimento quanto na manutenção, porém o seu uso requer um certo esforço na confecção dos componentes extras necessários.

Esses esforços podem ser minimizados com a automatização da metodologia para tornar as classes testáveis. Esse é o objetivo da ConCAT, descrita na seção 5.3.

5.1.3 Projeto Visando a Testabilidade de Software

Diante dos fatores que contribuem para a testabilidade de software foi proposta a utilização da noção de projeto visando testabilidade ou DFT (*Design For Testability*) e do

[1]No trabalho citado a classe foi denominada *autotestável*, porém neste trabalho ela é referenciada como *testável* por entendermos que a classe possui mecanismos para auxiliar nos testes, mas isto não basta para a execução dos testes na classe, sendo necessário um ambiente que utilize esta facilidade.

conceito de autoteste (mecanismos para a geração de padrões de teste e análise de resultados), oriundos do teste de hardware, para o desenvolvimento de sistemas Orientados a Objetos (Binder 1994). E para facilitar o acesso a componentes internos de um objeto, foi introduzida a capacidade de teste embutido ou BIT, além da automatização da geração, execução e análise dos testes, o que é denominado BIST (*Built-in Self Test*). Assim, tem-se a aplicação do conceito de autoteste às classes.

Um software orientado a objetos apresenta alguns obstáculos específicos, além daqueles existentes em software estruturado, na tentativa de alcançar uma alta testabilidade. A partir de uma analogia entre circuitos integrados e objetos alguns autores propuseram a aplicação das técnicas de DFT utilizadas no projeto de circuitos às classes (Hoffman 1989) e (Binder 1994). A seguir será apresentado como seria a aplicação das abordagens de DFT (citadas na seção 5.1) na construção de classes.

- Abordagem *ad hoc*

  Para testar uma classe são construídos *driver* e *stubs* específicos, além da inserção de assertivas e/ou outras formas de instrumentação para melhorar as capacidades de controle e observação. Uma desvantagem é que essa abordagem exige longas seqüências de configuração para colocar a classe em determinado estado, além de ferramentas de depuração para verificar o estado do objeto. Com isso, pode ocorrer um aumento do custo dos testes ou redução da confiabilidade dos mesmos.

- Abordagem Estruturada ou Padrão

  São adicionadas à classe funções padrões de BIT para prover capacidade de observação e controle, assim como funções para colocar a classe em determinado estado e para relatar o estado interno da classe. Porém, ainda é necessário a codificação de um *driver* para cada classe e a geração manual de casos de teste, isto é, existe limitação quanto à automatização da geração, execução e análise dos testes.

- Abordagem BIST

  Na abordagem BIST existe uma especificação de teste embutida à classe sob teste (CUT - do inglês *Class Under Test*) e um *driver* genérico capaz de gerar a seqüência de testes a partir desta especificação, executar esta seqüência sobre a classe sob teste e avaliar os resultados dos testes.

Na figura 5.1 são apresentados os componentes necessários para a inserção dos mecanismos BIST na classe para que ela torne-se testável:

- Classe testável - composta pelos seguintes elementos: classe sob teste (CUT), especificação de teste, e mecanismos BIT (métodos relatores e assertivas).

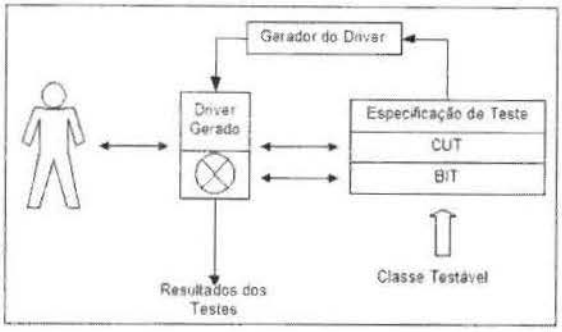

Figura 5.1: Abordagem de DFT BIST para classes.

- Gerador do *driver* - responsável pela geração do *driver* específico que ativará e controlará a CUT durante a execução dos testes.

- *Driver* gerado (específico) - corresponde à seqüência executável, gerada de acordo com o critério de teste implementado pelo gerador de *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.

5.2 Técnica Incremental Hierárquica

Em (Harrold et al. 1992), foi proposta a Técnica Incremental Hierárquica ou HIT (do inglês *Hierarchical Incremental Testing*) para permitir que o reuso também se estendesse aos testes. Seu principal benefício 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 dita hierárquica porque baseia-se na hierarquia entre classes introduzida pelo mecanismo de herança, e é incremental porque visa reutilizar os testes de uma classe em um nível hierárquico para testar classes em níveis subseqüentes.

O reuso dos testes depende da classificação das características (métodos e atributos) da classe sob teste (Siegel 1996). São definidos três tipos de métodos e cada tipo exige uma quantidade de teste diferente:

1. Método novo - definido apenas na subclasse (inclusive aqueles com nome igual ao de métodos na superclasse mas com parâmetros diferentes) e necessita de testes completos;

2. Método recursivo - definidos na superclasse e herdado sem modificação, necessita apenas de testes quando interage com métodos novos ou redefinidos;

3. Método redefinido - definido na superclasse e modificado (sobrecarregado ou reescrito) na subclasse, pode reutilizar testes funcionais da superclasse.

A HIT impõe algumas restrições de projeto e implementação para a sua utilização, tais como:

1. a herança existente deve ser de comportamento;
2. não é aplicável quando existe herança múltipla;
3. os programas devem ser implementados em linguagem C++, ou semelhantes que possuam níveis de visibilidade dos atributos;
4. o nível de visibilidade de um atributo na subclasse deve ser pelo menos tão restritivo quanto apresentado na superclasse.

Na técnica de teste HIT, as classes base são inicialmente testadas utilizando casos de testes baseados na especificação e na implementação, com isso, o histórico das informações dos testes são armazenadas. A classe base é primeiramente testada para cada função membro separadamente e depois a interação entre elas é testada.

Uma subclasse herda além dos atributos da superclasse, o histórico de testes da superclasse. Esse histórico de testes herdado é atualizado incrementalmente para refletir as diferenças em relação à superclasse e os resultados são armazenados no histórico de testes da subclasse, que serve como base para a execução dos casos de teste indicando quais casos de teste devem ser executados na subclasse. Somente novos atributos ou aqueles que foram herdados, atributos afetados e suas interações são testadas. Adicionalmente, casos de teste do conjunto de teste da superclasse são reusados, quando possível, para testar a subclasse. Com isso, obtém-se um ganho de tempo de análise da classe para determinar quais características deveriam ser testadas e ganho para executar os casos de testes.

Maiores informações sobre a técnica incremental hierárquica podem ser obtidas em (Harrold et al. 1992).

5.3 Automatização dos testes

A aplicação efetiva de um critério de teste exige a utilização de uma ferramenta de teste automatizada. Desta forma, foi desenvolvido um protótipo de uma ferramenta para auxiliar o testador na construção e uso das classes testáveis.

A ConCAT (ferramenta de Construção de Classes Testáveis) é um protótipo que tem por objetivo dar apoio à construção e utilização de classes testáveis.

A ferramenta visa auxiliar o produtor de uma classe a introduzir os mecanismos de autoteste (métodos relatores - para obtenção do estado interno - e assertivas - para checagem do estado interno em tempo de execução) e associar a especificação de teste à classe (Toyota & Martins 1999).

A metodologia utilizada pela ferramenta para permitir a construção de classes testáveis é composta por (Toyota 2000):

1. construção do modelo de teste para a classe sob teste;
2. construção de uma representação para o modelo de teste construído e associação desta à classe sob teste;
3. instrumentação da classe;
4. criação de um repositório de teste;
5. geração da seqüência de teste;
6. geração do oráculo;
7. compilação da classe em modo teste e execução da seqüência de teste.

5.3.1 Modelo de Teste

Para gerar o modelo de teste é construído um modelo de fluxo de transação para a classe sob teste, que é um modelo funcional utilizado para teste de sistemas (Beizer 1990) e adaptado para o teste de classes (Siegel 1996). Esse modelo descreve as seqüências válidas de invocação de métodos de um objeto.

Durante a geração dos casos de teste deve ser analisada a possibilidade de reuso. Para tanto, é utilizada a técnica incremental hierárquica (descrita na seção 5.2) com o objetivo de permitir a reutilização de testes de uma superclasse nos testes de suas subclasses.

Essa abordagem foi adaptada para os testes com base no modelo de fluxo de transação da seguinte forma: cada transação (seqüência de operações da classe) é tratada como um todo, ou seja, são consideradas as classificações de cada método que compõe a transação para classificá-la. Assim, se todos os métodos forem recursivos (definidos na superclasse e não sobrecarregados ou reescritos na subclasse) não é necessário retestar a transação. No entanto, se algum deles for redefinido (sobrecarregados ou reescritos na subclasse) e os demais recursivos pode-se aplicar o caso de teste gerado para esse objeto na superclasse. Caso algum deles seja novo, então um novo caso de teste deve ser gerado.

5.3.2 Representação do modelo de teste

Após a construção do grafo do modelo de fluxo de transação, essas informações precisam ser transformadas para que sejam utilizadas como entrada para o protótipo ConCAT construir o *driver* específico (correspondente à seqüência executável dos testes).

Em Toyota (2000) foi proposto um formato de especificação para o modelo de teste utilizado. Nesse formato, mostrado na figura 5.2, são fornecidas as informações de cada arco e nó do grafo, além de informações da classe, seus atributos e métodos, para que seja possível derivar mensagens aceitas pelos objetos da classe sob testes.

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

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

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

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

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

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

Figura 5.2: Formato para o modelo de teste.

As informações sobre a classe auxiliarão na geração automática de casos e seqüências

de testes, indicando as bibliotecas necessárias para compilar a classe e como manipulá-la. As informações sobre os parâmetros e os atributos serão importantes para a geração dos dados de teste. A classificação dos métodos permitirá verificar a possibilidade de reutilização dos casos de teste.

Após a elaboração da representação do modelo de teste, ele deve ser associado à classe sob teste, permitindo que toda a vez que a classe seja reutilizada sua especificação base esteja disponível.

5.3.3 Instrumentação da Classe sob Teste

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

Estes mecanismos devem ter visibilidade sobre os membros privados da classe a fim de facilitar o teste e a análise dos resultados dos casos de teste aplicados. Além disso, de acordo com Binder (1994) estes mecanismos devem ter uma interface padrão para facilitar a geração da seqüência de teste. A visibilidade dos membros privados é necessária para facilitar a análise dos resultados dos casos de testes aplicados.

Assertivas

Uma assertiva é uma expressão Booleana que define condições necessárias para a execução correta da classe (Binder 2000a). Elas podem expressar restrições específicas da implementação ou responsabilidades da classe independentes da sua implementação. Os principais usos de assertivas são:

- especificar as condições que devem ser verdadeiras na entrada para a execução de um método (pré-condição);
- especificar as condições que devem ser verdadeiras ao final de um método (pós-condição);
- especificar condições que devem ser verdadeiras para todas as operações de um objeto e também para suas superclasse e subclasses (invariante).

As assertivas são compostas por três partes (Binder 2000a): um predicado, uma ação e um mecanismo para habilitar/desabilitar as assertivas durante a execução. Os predicados descrevem as condições que devem ser avaliadas. Uma ação pode ser tão simples quanto a escrita de uma mensagem ou tão complicada quanto inicializar um mecanismo de recuperação. Geralmente, as assertivas são habilitadas ou desabilitadas em tempo de

transação através de um parâmetro. Este recurso, auxilia a configuração de uma classe em modo teste (assertivas habilitadas) ou em modo normal (assertivas desabilitadas).

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

As assertivas também podem ser derivadas da OCL (do inglês, *Object Constraint Language*) (Fowler & Scott 1997). A OCL é uma linguagem definida para expressar condições e outras expressões podendo ser utilizada juntamente com os diagramas da UML (do inglês, *Unified Model Language*), sendo que as assertivas utilizadas nos testes são inseridas no modelo de análise do sistema.

Já neste trabalho, as assertivas foram definidas através de algumas macros propostas em Toyota & Martins (1999).

Métodos Relatores

Os métodos relatores possuem a função de armazenar os valores dos atributos dos objetos durante a execução dos testes. Logo, eles mostram o estado da classe durante a execução dos casos de teste.

Para tanto, cada classe deve implementar um método relator, pois cada uma possui seus próprios atributos, os quais devem ser armazenados em arquivo para posterior análise. Dessa forma, é importante que os métodos relatores sejam executados antes e depois da execução de cada método durante os testes, só assim será possível analisar o estado antes e depois da execução de cada método.

5.3.4 Repositório de teste

Um repositório de teste faz-se necessário para armazenar todos os objetos relacionados aos testes da classe, como: o código da classe, sua especificação base de teste, as seqüências e os casos de testes gerados, os resultados obtidos e *stubs*.

Um repositório não precisa ser necessariamente uma base de dados relacional ou orientada a objetos (Siegel 1996), podendo ser também um sistema de arquivos estruturado ou outro tipo de armazenamento organizado para facilitar o armazenamento e recuperação das informações envolvidas na atividade de testes.

5.3.5 Seqüência de teste

A seqüência de teste determina a execução dos casos de testes disponibilizando as estruturas de dados necessárias. Representa um *driver* para a classe, onde cada caso de teste

deve testar uma transação do modelo de teste. A ConCAT gera a seqüência de teste num arquivo com sufixo "st0" e extensão ".cc" e trata a transação como um todo, para aplicar a técnica HIT ao modelo de fluxo de transação, levando em consideração a classificação de cada método que compõe a seqüência (exceto os métodos construtores e destrutores).

Dessa forma, se a seqüência é composta apenas por métodos recursivos não será necessário testar novamente esse objeto de teste. Se pelo menos um dos métodos for redefinido e os demais recursivos, o caso de teste gerado para esse objeto na classe base poderá ser reutilizado. Caso algum dos métodos seja novo, um novo caso de teste deverá ser gerado (Toyota 2000).

5.3.6 Oráculo

Os resultados esperados para um teste são essenciais para o projeto de teste, pois a avaliação de um caso de teste, para decidir se ele passa ou não, é realizada pela comparação do resultado atual com o resultado esperado de uma fonte confiável. O oráculo é uma fonte segura de resultados esperados para os casos de teste (Binder 2000a).

Um oráculo perfeito teria um comportamento equivalente a implementação sob teste e completamente confiável, isto é, uma versão livre de defeitos. Aceitaria todas as entradas especificadas e sempre produziria um resultado correto. Porém, encontrar um oráculo perfeito é tão difícil quanto resolver o problema do projeto original, além de impor restrições adicionais de corretude.

Embora um oráculo seja imprescindível para que os testes possam ser validades, nem sempre é possível obter-se um oráculo, pois às vezes não existe uma referência confiável para determinar as saídas corretas para um determinado caso de teste (Martins 1999).

No contexto da ConCAT o oráculo considerado é a saída esperada para cada método, ou uma exceção, caso o teste viole alguma das assertivas.

5.4 Considerações Finais

Neste capítulo foram apresentados os conceitos de Testabilidade de software juntamente com suas particularidades diante de sistemas orientados a objetos. Depois foi explorada a proposta de projeto visando a testabilidade. Culminando na utilização do conceito de autoteste no desenvolvimento de classes, para auxiliar na herança dos casos de teste e aumentar a testabilidade do software. No capítulo seguinte será definida a metodologia usada para utilizar classes testáveis para auxiliar nos testes de regressão.

Capítulo 6

Estratégia para seleção de Testes de Regressão

Baseado nos estudos apresentados nos capítulos anteriores, é possível verificar a importância de um alto grau de testabilidade de um software, para o baixo custo de sua manutenção. Por isso, a necessidade de definir uma boa estratégia para a realização de testes de regressão tanto na fase de desenvolvimento quanto durante a manutenção do sistema.

A estratégia utiliza informações de teste embutida na classe, dentre elas o modelo de comportamento da classe, permitindo a seleção dos testes independente da forma como elas foram implementadas. Para ilustrar a estratégia será utilizada a classe *Elevator* extraída do artigo Rothermel et al. (2000), cujo código é mostrado na figura 6.1.

Neste capítulo será mostrada a estratégia para teste de regressão de uma classe testável. O protótipo ConCAT (descrito na seção 5.3) é utilizado para auxiliar na geração dos casos de testes para novas funcionalidades da versão beta da classe sob teste.

6.1 Construção de Classe Testável

Baseada na metodologia proposta para utilização da ConCAT, mostrada na seção 5.3, no decorrer do capítulo serão descritos os passos para chegar a uma classe testável e como realizar testes de regressão para essa classe.

6.1.1 Modelo de Testes da classe

Para a realização dos testes em sistemas orientados a objetos é necessário definir primeiramente o modelo de testes da classe, pois o enfoque aqui são os testes de unidade e, em Orientação a Objetos, a unidade considerada é a classe (Pressman 1997a).

```
1. #include <iostream.h>
2. #include <stdlib.h>
3. #define UP 1
4. #define DOWN 2
5. typedef int Direction;
6.
7. class Elevator {
8. public:
9.   Elevator (int l_top_floor) {
10.    current_floor = -1;
11.    current_direction = UP;
12.    top_floor = l_top_floor;
13.    bottom_floor = 1;
14.  }
15.
16.  virtual ~Elevator() { }
17.
18.  void up() {
19.    current_direction = UP;
20.  }
21.
22.  void down() {
23.    current_direction = DOWN;
24.  }
25.
26.  Direction direction() {
27.    return current_direction;
28.  }
29.
30.  virtual void go (int floor) {
31.    int valid = valid_floor(floor);
32.    if (! valid_floor(floor)) {
33.      cout << "Invalid floor request\n";
34.      return;
35.    }
36.    if (floor > current_floor) {
37.      up();
38.      cout << "Elevator is going up";
39.    }
40.    else if (floor < current_floor) {
41.      down();
42.      cout << "Elevator is going down";
43.    }
44.    else
45.      return;
46.    if (current_direction == UP) {
47.      while ((current_floor != floor)
         && (current_floor <= top_floor))
48.        add(current_floor, 1);
49.    }
50.    else {
51.      while ((current_floor != floor)
         && (current_floor <= bottom_floor))
52.        add(current_floor, -1);
53.    }
54.  };
55.
56. private:
57.   add(int &a, const int &b) {
58.     a = a+b;
59.   }
60.
61.   int valid_floor(int floor) {
62.     if ((floor > top_floor) || (floor < bottom_floor))
63.       return 0;
64.     return 1;
65.   };
66.
67. protected:
68.   int current_floor;
69.   Direction current_direction;
70.   int top_floor;
71.   int bottom_floor;
72. };
73.
74. void main (int argc, char **argv) {
75.   Elevator *e_ptr;
76.
77.   e_ptr = new Elevator(10);
78.   e_ptr -> go(2);
79. }
```

Figura 6.1: Código da classe *Elevator*.

O modelo de teste utilizado nesse trabalho foi o Modelo de Fluxo de Transação (MFT - *Transaction Flow Model*). Trata-se de um modelo funcional, que foi definido por Beizer (1990) para o teste de sistemas concorrentes e adaptado em Siegel (1996) para o teste de objetos. O modelo ilustra as diferentes maneiras de criar um objeto, diferentes tarefas ou processos que um objeto pode fazer e as diferentes formas de destruir o objeto (Siegel 1996).

Uma transação representa um ciclo de vida de um objeto, ou seja, o fluxo entre a criação do objeto até sua destruição, indicando uma seqüência de operações que podem ser realizadas pelo sistema, pessoas ou dispositivos que estão fora do sistema e que usam os serviços prestados pelo objeto. O modelo é representado por um grafo, onde os nós designam as etapas (ou tarefas) de processamento de uma transação, ou seja, refletem os atributos e métodos públicos da classe, encapsulando todas as referências para outros objetos e os arcos representam as seqüências de execuções válidas entre os nós.

Uma característica interessante do modelo de fluxo de transação é que ele fornece recursos que auxiliam na especificação de sistemas concorrentes, indicando quais tarefas podem ser executadas concorrentemente e os pontos onde elas devem ser sincronizadas. Porém essa funcionalidade do MFT não será tratada nesse trabalho, já que a ferramenta utilizada para a aplicação da estratégia (ConCAT, descrita em 5.3) permite apenas geração

de casos de testes para sistemas que não apresentam concorrência.

O modelo de fluxo de transação deve ser construído a partir do refinamento dos casos de uso da classe através dos passos a seguir (Siegel 1996):

1. Eliminar qualquer referência externa, como referências a bases de dados, arquivos, métodos de outras classes;

2. Simplificar o modelo combinando métodos em um único processo (nó) quando possível;

3. Verificar cenários de uso que não aparecem no projeto de classes, pois isto pode indicar que alguns requisitos não foram satisfeitos;

4. Revisar os métodos públicos que não aparecem no modelo construído, pois isto pode indicar que algum requisito não foi satisfeito ou que estes métodos não deveriam pertencer à interface pública da classe.

O MFT é um grafo em que os nós representam as tarefas de processamento de uma transação e os arcos direcionados designam as seqüências entre os nós. Nesse trabalho, uma transação determinará uma seqüência de execução de métodos pertencentes à interface pública da classe sob teste, indicando um caminho do nó inicial ao nó final.

Para exemplificar, a figura 6.2 mostra o modelo de fluxo de transação que representa a classe *Elevator* apresentada na figura 6.1.

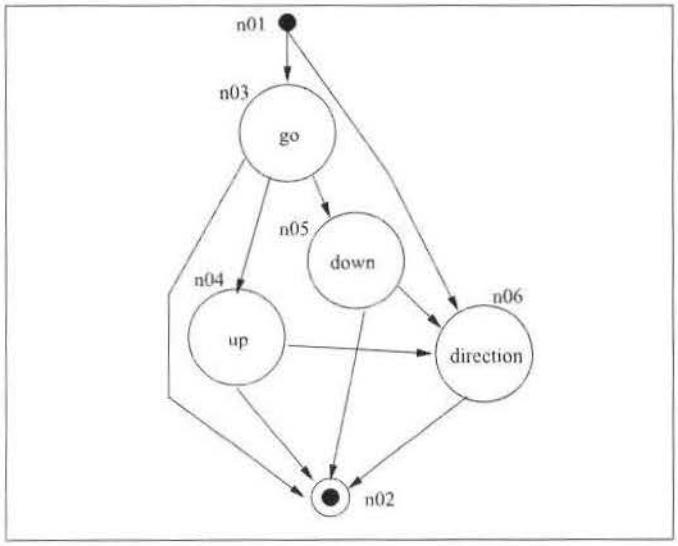

Figura 6.2: Modelo de Fluxo de Transação para a classe *Elevator*.

A geração de casos de teste a partir do MFT é realizada através do percurso do grafo de fluxo de transação. Em Beizer (1995) são propostos alguns métodos para a derivação de casos de testes a partir do Modelo de Fluxo de Transação.

A ConCAT implementa cobertura de caminhos livres de laços. Um caminho inicia em um nó inicial e termina em um nó final, representando uma transação. Os laços (ou ciclos) devem ser eliminados pois o algoritmo implementado, que realiza uma busca em profundidade no grafo que representa o MFT, não termina nesses casos.

Através do MFT pode-se encontrar as seguintes falhas: falta de transações (funcionalidades), falta de arcos/nós e problemas com os tipos de nós (sincronização) (Beizer 1990).

6.1.2 Especificação de base para os testes

É importante que a especificação usada nos testes descreva todas as seqüências de tarefas válidas e também os parâmetros necessários para cada operação e os seus domínios, possibilitando que todos os caminhos sejam exercitados no sistema, o que também é o objetivo do modelo de fluxo de transação.

Uma vez criado o modelo de fluxo de transação para cada classe, este deve ser descrito de acordo com o formato indicado na figura 5.2 que foi apresentado no capítulo anterior (seção 5.3.2). As informações relacionadas a cada nó auxiliarão no percurso do grafo para a derivação das transações. Além disso, também são fornecidas informações sobre os métodos, atributos e parâmetros para auxiliarem na geração dos casos de teste.

A descrição do modelo é armazenada em um arquivo no mesmo diretório da classe sob teste. Esse arquivo tem a extensão ".mt"e é utilizado pela ConCAT para a geração dos casos de teste. Esta descrição deve ser associada à classe sob teste, de forma que toda vez que a classe seja reutilizada sua especificação esteja disponível (Toyota 2000).

Para o exemplo utilizado, classe *Elevator*, uma especificação de base para os testes para o modelo de fluxo de transação mostrado na figura 6.2 está ilustrado na figura 6.3.

A partir dessa especificação de testes, a ConCAT consegue gerar uma seqüência de testes para exercitar todas as transações possíveis no grafo que representa o modelo de fluxo de transação e que foi representado no formato proposto na figura 5.2.

6.1.3 Instrumentação da classe

A instrumentação da classe consiste na inclusão dos mecanismos de teste embutido (BIT) que são relatores e pré e pós-condições que devem ser satisfeitas em cada método da interface da classe sob testes. Como esses mecanismos são colocados diretamente no código fonte da classe, eles são incorporados à classe e passam a fazer parte do seu código, caracterizando um processo intrusivo de teste. No entanto essa intrusão é necessária pois é através desses mecanismos que obtém-se visibilidade sobre os membros privados

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

parametros_template([]).

atributo('current_floor',int,_,_).
atributo('current_direction',string,['Direction'],_).
atributo('top_floor',int,_,_).
atributo('bottom_floor',int,_,_).

metodo(m01,'Elevator',_,construtor,1).
metodo(m02,'~Elevator',_,destrutor,0).
metodo(m03,'go','void',novo,1).
metodo(m04,'up','void',novo,0).
metodo(m05,'down','int',novo,0).
metodo(m06,'direction','Direction',novo,0).

parametro('l_top_floor',m01,objeto,'int',_).
parametro('floor',m03,objeto,'int',_).
```

```
no(n01,sim,2,[m01]).
no(n02,nao,0,[m02]).
no(n03,nao,3,[m03]).
no(n04,nao,2,[m04]).
no(n05,nao,2,[m05]).
no(n06,nao,1,[m06]).

arco(n01,n03).
arco(n01,n06).
arco(n03,n04).
arco(n03,n05).
arco(n04,n06).
arco(n05,n06).
arco(n06,n02).
arco(n03,n02).
arco(n04,n02).
arco(n05,n02).
```

Figura 6.3: Descrição do modelo de teste para classe *Elevator*.

das classes, permitindo que seja obtido o estado interno do objeto, útil para análise dos resultados.

Os relatores, representados pelo método *Relator*, coletam os valores dos atributos da classe durante a execução dos testes e os grava em arquivo para posterior análise. Como o relator guarda o estado do objeto, é importante que o método seja executado antes e depois de cada caso de teste.

As assertivas, compostas por invariantes, pré e pós-condições, descrevem as condições necessárias para a correta execução dos objetos da classe. Essas condições devem ser mantidas entre as classes e seus clientes durante seus relacionamentos, isto é, as pré e pós-condições implementam o contrato que deve ser satisfeito entre as classes e seus clientes. Enquanto que as invariantes especificam condições que devem ser válidas para todas as operações de um objeto. Para esse trabalho, as assertivas foram definidas através de algumas macros propostas em (Toyota 2000). Quando uma assertiva é violada, gerará uma mensagem de notificação no arquivo de saída e será dada continuidade à seqüência de testes.

6.1.4 Geração de casos de teste

Após a criação do modelo de teste e da instrumentação da classe com os mecanismos de teste embutido, ainda é necessário um gerador de *driver* capaz de ler a especificação de teste e gerar um *driver* específico que contenha a seqüência de casos de teste e seja responsável por ativar e controlar a classe durante a execução dos casos de teste.

Parte desse passo é feito automaticamente pela ConCAT. Um exemplo dos casos de teste gerados é mostrado na figura 6.4(A). Este deve ser completado com valores de parâmetros e dados globais (Toyota 2000). Os *stubs*, que substituem as classes que prestam serviço à classe sob testes, também devem ser criados pelo usuário. O caso de teste instanciado é mostrado na figura 6.4(B).

Uma classe sob teste é definida como subclasse da classe *Autoteste* (que possui a definição dos métodos BIT). Os métodos "Relator" e "TestaInvariante" devem ser redefinidos na classe para expressar os valores relevantes para análise na classe. No exemplo de caso de teste, o método "TestaInvariante" é chamado antes e depois de cada método a ser testado. Já o método "Relator" é chamado ao final da execução de cada caso de teste para passar todos os atributos da classe para o arquivo que conterá os resultados dos testes.

No arquivo definido pela variável *arq* na figura 6.4(B) serão armazenadas as informações da execução dos testes (o caso de teste executado e informações analisadas pelo Relator). Os métodos são chamados dentro de um bloco "try-catch" para que seja possível capturar alguma exceção sinalizada pela violação de pós-condições ou invariantes de classe. Caso uma exceção seja levantada são registrados: uma mensagem indicando o caso de teste sob execução, a condição violada e o estado interno da classe, através do método Relator (conforme apresentado no bloco "catch").

A ConCAT permite o reuso de casos de testes da superclasse quando são gerados testes da subclasse como será visto na seção 6.3.

Para a geração de testes, o modelo não deve conter ciclos. De acordo com Beizer (1990)(cap. 4), ciclos são pouco freqüentes nesse tipo de modelo. Caso ocorram, deve-se optar por um dos caminhos; um dos arcos que provocam ciclos devem ser removidos da especificação de testes.

6.1.5 Oráculo

Embora Binder (2000a) considere que os resultados dos testes da versão base sustentem o Oráculo para todos os padrões de teste de regressão seletivos, nessa estratégia, optou-se por considerar as pós-condições como oráculo, já que como a classe pode ser reutilizada, seus usuários talvez não tenham acesso aos resultados dos testes aplicados na fase de desenvolvimento do sistema.

6.2 Testes de Regressão - uma Estratégia

A estratégia de teste baseada nos passos para a realização de testes de regressão seletiva, é descrita a seguir:

```
template <class Tipo>
void Caso_teste2_0(Tipo* cst) {
  char* met= new char[30];
  ofstream arq("Result.txt",ios::app);
  if (! arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Arquivo criado! \n";
  try {
    cst->Testa_Invariante();
    met = "go(Parametro deve ser um objeto do tipoint)";
    cst->go(Parametro deve ser um objeto do tipoint);
    cst->Testa_Invariante();
    met = "up()";
    cst->up();
    cst->Testa_Invariante();
    arq << "Caso_teste2_0 OK!\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
    delete cst;
  }

  catch(char* c) {
    arq << "Caso_teste2_0 \n";
    arq.flush();
    cout << "...";
    arq << c << "\n";
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
  }

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

(A)

```
template <class Tipo>
void Caso_teste2_0(Tipo* cst) {
  char* met= new char[30];
  ofstream arq("Result.txt",ios::app);
  int option = 3;

  if (! arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Arquivo criado! \n";
  try {
    cst->Testa_Invariante();
    met = "go(int option)";
    cst->go(option);
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();

    cst->Testa_Invariante();
    met = "up()";
    cst->up();
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();

    cst->Testa_Invariante();
    arq << "Caso_teste2_0 OK!\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
    delete cst;
  }

  catch(char* c) {
    arq << "Caso_teste2_0 \n";
    arq.flush();
    cout << "...";
    arq << c << "\n";
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
  }

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

(B)

Figura 6.4: (A) Exemplo de caso de teste gerado pela ConCAT para a classe *Elevator*; (B) Mesmo Caso de teste instanciado.

1. Construir o modelo de fluxo de transação para a versão base;
2. Determinar o conjunto de testes **T** da versão base **P** utilizando a ConCAT;
3. Instrumentar a classe da versão base para executar o conjunto de testes **T**. A instrumentação consiste na inclusão de pós-condições, invariantes e métodos relatores;
4. Selecionar o conjunto de testes **T'** para a versão beta, sendo que **T'** $\subseteq$ **T** é gerada utilizando a estratégia de teste de regressão proposta em 6.2.1;
5. Atualizar instrumentação da classe para a versão beta;
6. Testar a versão beta **P'** utilizando o conjunto de testes **T'**;
7. Gerar o conjunto de testes **T"** para testar as novas funcionalidades da versão beta, novamente utilizando a ConCAT e conseqüentemente a técnica HIT.
8. Testar a versão beta usando o novo conjunto de testes **T"**;
9. Criar **T"'**, um novo conjunto de testes e histórico de testes de **P'**, onde **T"'** é igual a união de **T'**, **T"** e **T** (exceto obsoletos).

O passo inicial desse processo prevê a construção do MFT (detalhado na seção 6.1.1) para a versão base, que servirá como entrada para a ConCAT após conversão para a representação adequada, conforme mostrado na seção 6.1.2. Esse passo só será necessário se a classe não for testável.

O segundo passo do processo consiste em gerar a seqüência de testes e *driver* utilizando a ConCAT conforme mostrado na seção 6.1.4. O que só será necessário caso a classe não tenha sido testada antes, caso contrário, os testes já estarão no histórico de testes.

A informação de cobertura de testes para a classe *Elevator* é apresentada na figura 6.5. Nessa figura se observa que um objeto de teste relaciona uma transação com o caso de teste que a cobre. Cada caso de teste é identificado por: "Caso_testeX_0", onde X é um número seqüencial. Cada linha apresenta uma transação possível, onde cada um dos 'm' representam a identificação de um dos métodos da classe (conforme definido na descrição do modelo de teste - figura 6.3). Além disso, também é apresentado o nome do arquivo onde está a implementação de cada caso de teste: "Elevator_ct0.cc".

Na figura 6.6, é apresentado um *driver* específico para execução da seqüência de teste, onde existe a inclusão do arquivo contendo os casos de teste (*Elevator_ct0.cc*) e também do arquivo do código da classe (*Elevator.cpp*). No *driver* também são instanciados objetos da classe, um para cada caso de teste definido.

No terceiro passo do processo incluem-se assertivas no código da classe, antes da execução do conjunto de testes, conforme descrito na seção 6.1.3, que só será necessário

```
objeto_tst(1,['m01','m03','m02'],'Caso_teste1_0','Elevator_ct0.cc').
objeto_tst(2,['m01','m03','m04','m02'],'Caso_teste2_0','Elevator_ct0.cc').
objeto_tst(3,['m01','m03','m04','m06','m02'],'Caso_teste3_0','Elevator_ct0.cc').
objeto_tst(4,['m01','m03','m05','m02'],'Caso_teste4_0','Elevator_ct0.cc').
objeto_tst(5,['m01','m03','m05','m06','m02'],'Caso_teste5_0','Elevator_ct0.cc').
objeto_tst(6,['m01','m06','m02'],'Caso_teste6_0','Elevator_ct0.cc').
```

Figura 6.5: Cobertura dos testes gerados pela ConCAT para a classe *Elevator*.

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

#include "Elevator_ct0.cc"

#define MAX_ANDAR 4

#define TESTE

void main() {
  Elevator* obj1 = new Elevator(MAX_ANDAR);
  Caso_teste1_0(obj1);
  Elevator* obj2 = new Elevator(MAX_ANDAR);
  Caso_teste2_0(obj2);
  Elevator* obj3 = new Elevator(MAX_ANDAR);
  Caso_teste3_0(obj3);
  Elevator* obj4 = new Elevator(MAX_ANDAR);
  Caso_teste4_0(obj4);
  Elevator* obj5 = new Elevator(MAX_ANDAR);
  Caso_teste5_0(obj5);
  Elevator* obj6 = new Elevator(MAX_ANDAR);
  Caso_teste6_0(obj6);
}
```

Figura 6.6: Driver de teste para a classe *Elevator*.

se a classe ainda não foi testada. A figura 6.7 representa a classe Elevator após tornar-se uma classe testável.

```
#include <iostream.h>
#include <stdlib.h>
#define UP 1
#define DOWN 2
typedef int Direction;

class Elevator: public Autoteste {
 public:
#ifdef TESTE
void Testa_Invariante() {
  Invariante(this != NULL);
}
      resultados << "CURRENT_FLOOR: " << current_floor << "\n";

void Relator(char* arquivo) {
  ofstream resultados(arquivo, ios::app);
  if(!resultados){
    cout << "Erro abrindo arquivo relator! \n";
    exit(0);
   }
   else {
    resultados << "CURRENT_FLOOR: " << current_floor << "\n";
    resultados << "CURRENT_DIRECTION: " << current_direction << "\n";
    resultados << "TOP_FLOOR: " << TOP_FLOOR << "\n";
    resultados << "BOTTOM_FLOOR: " << bottom_floor << "\n";
    resultados.flush();
    resultados << "\n";
    resultados.close();
   }
 }
#endif

  Elevator (int l_top_floor) {
   current_floor = -1;
   current_direction = UP;
   top_floor = l_top_floor;
   bottom_floor = 1;
  }

  virtual ~Elevator() { }

  void up() {
   current_direction = UP;
   Pos_condicao(current_floor >= 1);
  }

void down() {
 current_direction = DOWN;
 Pos_condicao(current_floor >= 1);
}

Direction direction() {
Pos_condicao(current_direction == UP || current_direction == DOWN);
 return current_direction;
}

virtual void go (int floor) {
 int valid = valid_floor(floor);
 if (!valid_floor(floor)) {
  cout << "Invalid floor request\n";
 Pos_condicao(floor > top_floor || floor < bottom_floor);
  return;
 }
 if (floor > current_floor) {
  up();
  cout << "Elevator is going up";
 }
 else if (floor < current_floor) {
  down();
  cout << "Elevator is going down";
 }
 else {
 Pos_condicao(floor == current_floor);
  return;
 }
 if (current_direction == UP) {
  while ((current_floor != floor)
   && (current_floor <= top_floor))
    add(current_floor, 1);
 }
 else {
  while ((current_floor != floor)
   && (current_floor <= bottom_floor))
    add(current_floor, -1);
 }
Pos_condicao(current_floor == floor);
};

private:
```

Figura 6.7: Classe *Elevator* após a instrumentação.

Na seção 6.2.1 é descrito como alcançar o conjunto de teste de regressão (quarto passo do processo).

Caso ocorra alguma alteração na especificação da classe entre a versão base e beta, as assertivas devem ser atualizadas para satisfazer essas mudanças, conforme passo 5.

Depois de encontrado o conjunto de testes de regressão, a versão beta da classe sob teste deve ser testada utilizando esse conjunto de teste, conforme indicado no passo 6 da

estratégia.

Após a realização de uma análise da classe na versão beta, se essa possuir novas funcionalidades, deve ser determinado quais métodos foram alterados e quais são novos em relação à classe base. A partir dessa análise, a ConCAT irá gerar os testes usando a abordagem HIT, como nos testes de subclasse, satisfazendo o passo 7. Esse novo conjunto de testes encontrado deve ser aplicado para testar a classe beta, conforme passo 8.

Por último, no passo 9, o conjunto de testes assumido como base para os testes da próxima versão deve ser atualizado considerando o conjunto de testes de regressão para a versão beta acrescida do conjunto de testes que abrangem as novas funcionalidades.

A figura 6.8 representa um esquema da estratégia proposta para teste de regressão. São mostradas a superclasse e a subclasse na primeira versão P e na segunda versão P'. Ao lado das classes de P, são mostrados os conjuntos de testes obtidos através da ConCAT: T e T_{sub}. A seta ligando a superclasse nas duas versões representa a utilização da técnica para obtenção do conjunto seletivo de teste de regressão, T'. O mesmo é observado com a seta ligando as duas subclasses, onde o conjunto T'_{sub} é obtido. Além disso, o conjunto de testes para as novas funcionalidades das classes em P' devem ser obtidos, e novamente a HIT é considerada para a reutilização dos testes, sendo gerados os conjuntos de teste T'' e T''_{sub} para a superclasse e subclasse, respectivamente.

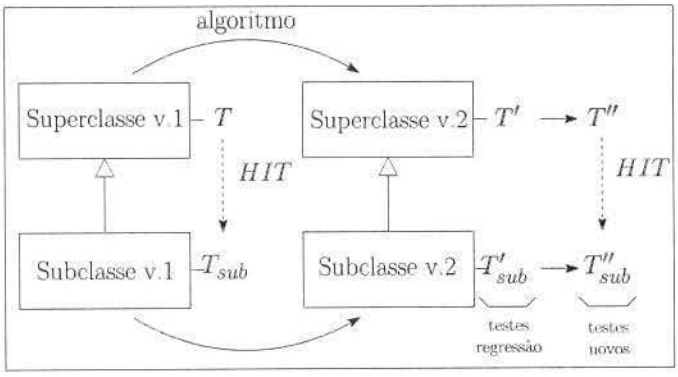

Figura 6.8: Esquema para representar a estratégia de teste de regressão.

6.2.1 Seleção de testes de regressão

Depois da obtenção do conjunto de testes para a versão base (segundo passo citado na seção anterior), a questão é como encontrar um subconjunto desses testes a ser aplicado na versão beta.

Na seção 3.6 foram apresentados padrões para redução de um conjunto de testes para teste de regressão. Dentre os padrões propostos, iremos nos basear em Reteste baseado em segmentos modificados por ser baseado em percurso em um grafo (no caso, o grafo de fluxo de controle). Dessa forma, como temos o modelo de fluxo de transação para as classes representado também por um grafo, julgamos essa técnica a mais adequada.

O reteste baseado em segmentos modificados usa o algoritmo proposto por Rothermel & Harrold (1994), onde é realizada a representação do código na forma de grafo de fluxo de controle. No grafo de fluxo de controle os nós representam instruções do programa e as arestas representam o fluxo de controle entre instruções do programa. Em um grafo de chamadas, os nós representam procedimentos, e as arestas, a hierarquia de chamadas. O algoritmo visa comparar os grafos representando os programas **P** e **P'** (nova versão de **P**), de forma a identificar as diferenças entre os dois grafos com relação a uma entidade considerada. No caso do algoritmo apresentado em (Binder 2000b), essa entidade é uma aresta (ou segmento) do grafo de fluxo de controle. São selecionados testes que exercitam os segmentos modificados.

No caso deste trabalho, o grafo representa o MFT, onde cada nó corresponde a um método da interface pública da classe, e o segmento representa o fluxo de execução dos métodos.

Para selecionar o conjunto de testes de regressão, deve-se utilizar o procedimento definido para Reteste baseado em Segmento Modificado, conforme mostrado na seção 3.6, para a obtenção da relação entre os arcos e os casos de teste que o exercitam. Esse procedimento faz-se necessário já que o algoritmo mostrado na figura 6.9 realiza comparação entre arcos do grafo (MFT) e a ConCAT gera os objetos de teste com transações (conforme mostrado na figura 6.5), representada por uma seqüência de nós.

A seguir são mostrados os passos propostos pelo procedimento para Reteste de Segmento Modificado exemplificados para as classes *Elevator* e sua nova versão *Elevator'*, cujo código está apresentado na figura 6.10:

1. Gerar os dados de análise de cobertura de casos de teste indicando quais arcos são atravessados em cada caso de teste (mostrado na figura 6.11).

2. Relacionar cada segmento (arco) com os casos de teste que o exercitam (conforme figura 6.12). A partir dos objetos de teste que foram gerados pela ConCAT mostrados na figura 6.5, e da identificação dos arcos mostrados na figura 6.3 aplica-se o procedimento implementado em "converter" (disponível no apêndice E) para obtenção dessa relação.

3. Identificar as alterações de uma versão para outra junto com a classificação dessas modificações: novo, mesmo, alterado ou removido, mostrado na figura 6.13. A

```
SELECAOTESTES(P, P', T)
1  T' ← ∅;
2  E ← ∅;
3   Obtenha G e G', MFT para P e P', com nós de entrada e e e'
4  COMPARA(e, e');
5   para  cada arco (n1,n2) pertencente a E
6  T' ← T' ⋃ CasosTeste((n1, n2));
7  retorna T';
8

COMPARA(N, N')
 1   Marque N com 'N'-visitado'
 2  se  arcos de saída para N' não equivalentes a arcos de saída de N
 3     entao
 4              para  cada sucessor C de N em G
 5             se  arco de saída C' de G' não equivalente a arco de saída C de G
 6                entao
 7                        E ← E ⋃ (N, C);
 8                        se  C não é marcado com 'C'-visitado'
 9                           entao
10                                   COMPARA(C, C');
11  senao
12           para  cada sucessor C de N em G
13          se  C não é marcado 'C'-visitado'
14             entao
15                     se  C e C' não são equivalentes
16                        entao
17                                E ← E ⋃ (N, C);
18
19                        senao
20                                COMPARA(C, C');
```

Figura 6.9: Pseudo-código do algoritmo de seleção de testes de regressão.

```
1. #include <iostream.h>
2. #include <stdlib.h>
3. #define UP 1
4. #define DOWN 2
5. typedef int Direction;
6.
7. class Elevator {
8. public:
9.   Elevator (int l_top_floor) {
10.    current_floor = -1;
11.    current_direction = UP;
12.    top_floor = l_top_floor;
13.    bottom_floor = 1;
14.  }
15.
16.  virtual ~Elevator() { }
17.
18.  void up() {
19.    current_direction = UP;
20.  }
21.
22.  void down() {
23.    current_direction = DOWN;
24.  }
25.
26.  Direction direction() {
27.    return current_direction;
28.  }
29.
29.1.  int which_floor() {
29.2.    return current_floor;
29.3.  }
30.  virtual void go (int floor) {
31.    int valid = valid_floor(floor);
32.    if (!valid_floor(floor)) {
33.      cout <<"Invalid floor request\n";
34.      return;
35.    }
36.    if (floor > current_floor) {
37.      up();
38.      cout << "Elevator is going up";
39.    }
40'.    else if (floor <= current_floor) {
41'.      down();
42'.      cout <<"Elevator is going down";
43'.    }
44'.    else{
44.1      cout <<"Elevator is on the same floor";
45'.      return; }
46.    if (current_direction == UP) {
47.      while ((current_floor != floor)
         && (current_floor <= top_floor))
48.        add(current_floor, 1);
49.    }
50.    else {
51.      while ((current_floor != floor)
         && (current_floor <= bottom_floor))
52.        add(current_floor, -1);
53.    }
54.  };
55.
56. private:
57.   add(int &a, const int &b) {
58.     a = a+b;
59.   }
60.
61.   int valid_floor(int floor) {
62.     if ((floor > top_floor) || (floor < bottom_floor))
63.       return 0;
64.     return 1;
65.   };
66.
67. protected:
68.   int current_floor;
69.   Direction current_direction;
70.   int top_floor;
71.   int bottom_floor;
72. };
73.
74. void main (int argc, char **argv) {
75.   Elevator *e_ptr;
76.
77.   e_ptr = new Elevator(10);
78.   e_ptr -> go(2);
79. }
```

Figura 6.10: Código da segunda versão da classe *Elevator'*.

Identificação de cada arco:
a01 = arco (n01, n03);
a02 = arco (n03, n02);
a03 = arco (n03, n04);
a04 = arco (n04, n06);
a05 = arco (n03, n05);
a06 = arco (n01, n06);
a07 = arco (n05, n06);
a08 = arco (n04, n02);
a09 = arco (n05, n02);
a10 = arco (n06, n02);

Análise de cobertura por arco:
caso_teste1_0: a01, a02
caso_teste2_0: a01, a03, a04, a10
caso_teste3_0: a01, a03, a08
caso_teste4_0: a01, a05, a07, a10
caso_teste5_0: a01, a05, a09
caso_teste6_0: a06, a10

Figura 6.11: Análise de cobertura de arcos para a classe *Elevator*.

Relação entre cada arco e caso de teste que o exercita:

a01	Caso_teste1_0	a05	Caso_teste4_0
a01	Caso_teste2_0	a05	Caso_teste5_0
a01	Caso_teste3_0	a06	Caso_teste6_0
a01	Caso_teste4_0	a07	Caso_teste4_0
a01	Caso_teste5_0	a08	Caso_teste3_0
a02	Caso_teste1_0	a09	Caso_teste5_0
a03	Caso_teste2_0	a10	Caso_teste2_0
a03	Caso_teste3_0	a10	Caso_teste4_0
a04	Caso_teste2_0	a10	Caso_teste6_0

Figura 6.12: Relação de cada segmento com a identificação do caso de teste que o exercita.

obtenção dos métodos novos, inalterados ou modificados pode ser feita usando-se uma ferramenta de controle de versões.

Relatório de controle de alteração:

Base	Beta	Classificação
a01=(n01,n03)	(n01,n13)	alterado
a02=(n03,n02)	(n13,n02)	alterado
a03=(n03,n04)	(n13,n04)	alterado
a04=(n04,n06)	(n04,n06)	mesmo
a05=(n03,n05)	(n13,n05)	alterado
a06=(n01,n06)	(n01,n06)	mesmo
a07=(n05,n06)	(n05,n06)	mesmo
a08=(n04,n02)	(n04,n02)	mesmo
a09=(n05,n02)	(n05,n02)	mesmo
a10=(n06,n02)	(n06,n02)	mesmo
	a11=(n01,n17)	novo
	a12=(n17,n02)	novo

Classificação de cada arco de beta em relação a versão base:

a01	alterado
a02	alterado
a03	alterado
a04	mesmo
a05	alterado
a06	mesmo
a07	mesmo
a08	mesmo
a09	mesmo
a10	mesmo
a11	novo
a12	novo

Figura 6.13: Comparação e classificação de cada segmento na versão base e beta.

4. Ordenar dos dados apresentados nas figuras 6.12 e 6.13 para obtenção de uma lista ordenada de caso de teste e classificação do método na versão beta, conforme mostrado na figura 6.14.

Com as informações obtidas no quarto passo (mostrada na figura 6.14) é possível definir os testes que devem ser aplicados para testar a versão alterada: se um método de uma classe é considerado *alterado*, então os casos de teste que o exercitam farão parte do conjunto de testes de regressão; para as demais classificações dos métodos (conforme passo 3), os casos de teste não são selecionados para o conjunto de teste de regressão.

Como se trata de testes caixa-preta, a regra definida em (Binder 2000b) referente aos segmentos removidos não se aplica, dado que quando um método é removido da interface pública da classe, não há como executar os testes desse método, já que ele não existe mais. Por tanto, o algoritmo, mostrado na figura 6.9, foi adaptado para o universo intra-classe,

Ordenação pelo identificador do caso de teste:

a01 alterado a01 t1 a01 t2 a01 t3 a01 t4 a01 t5	a06 mesmo a06 t6
a02 alterado a02 t1	a07 mesmo a07 t4
a03 alterado a03 t2 a03 t3	a08 mesmo a08 t3
a04 mesmo a04 t2	a09 mesmo a09 t5
a05 alterado a05 t4 a05 t5	a10 mesmo a10 t2 a10 t4 a10 t6
	a11 novo a12 novo

Figura 6.14: Lista ordenada de casos de teste e classificação das alterações de cada segmento.

sendo que só serão selecionadas seqüências de teste que exercitam métodos (representados em nós no grafo) classificados como alterados.

Para encontrar o conjunto de teste de regressão para testar a nova versão da classe *Elevator'* deve-se aplicar o algoritmo da seguinte forma: inicialmente o MFT para cada versão da classe deve ser obtido, conforme mostrado na figura 6.15, sendo G o MFT de *Elevator*, e G' o MFT de *Elevator'*. A seguir, a função *Compara* é chamada com os parâmetros $e = N = n01$ e $e' = N' = n01$, sendo e o primeiro nó do grafo G e e' o primeiro nó de G'. *Compara* marca $n01$ de G com o texto "n01-visitado". Ao comparar $n01$ de G com $n01$ de G', verifica-se que seus arcos de saída $(n01, n03)$ e $(n01, n13)$, respectivamente, não são equivalentes e depois de incluir o arco de G no conjunto E que é o conjunto de arcos não equivalentes, de forma que $E = (n01, n03)$, todos os nós C sucessores de e em G são comparados com os sucessores C' de e' em G'. Ao comparar $n03$ de G com $n13$ de G', após marcar o nó $n03$ com o texto "n13-visitado", novamente é descoberto que os arcos de saída dos dois nós comparados não são equivalentes: arcos de saída de $n03 = (n03, n02), (n03, n04), (n03, n05)$; arcos de saída de $n13 = (n13, n02), (n13, n04), (n13, n05)$; assim novamente os arcos de saída de G são incluídos no conjunto E, logo $E = (n01, n03), (n03, n02), (n03, n04), (n03, n05)$. Observando a figura 6.11, conclui-se que os arcos a01, a02, a03 e a05 fazem parte do conjunto E.

Depois que todos os nós de G estão marcados com o texto "N'-visitado", a comparação termina, e a execução volta para *SelecaoTestes*, onde todos os casos de teste que exercitam os arcos contidos no conjunto $E = \{a01, a02, a03, a05\}$, que são os arcos classificados como alterados (Binder 2000b), serão incluídos no conjunto de testes de regressão **T'**. Dessa

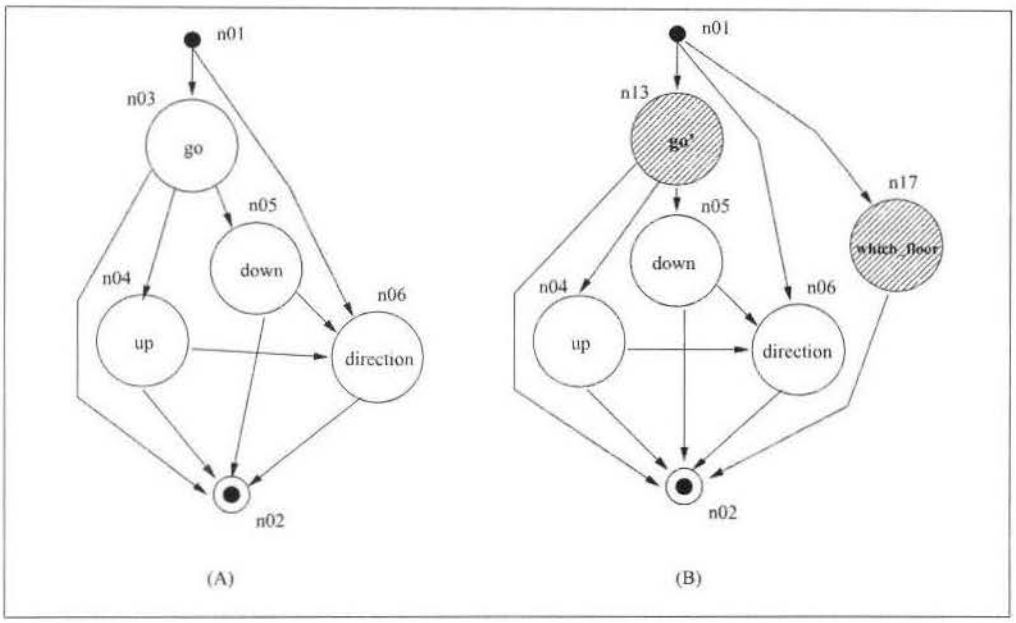

Figura 6.15: (A) MFT para a classe *Elevator*; (B) MFT para a nova classe *Elevator'*.

forma **T'** conterá todos os casos de testes gerados pela ConCAT que envolvem testes do nó $n03$ correspondente ao método "go" da classe *Elevator*.

Conforme mostrado em 6.11 nota-se que apenas o sexto caso de teste não exercita esse nó. Logo o conjunto de testes de regressão **T'** será composto por: $Caso\_teste1\_0$, $Caso\_teste2\_0$, $Caso\_teste3\_0$, $Caso\_teste4\_0$, $Caso\_teste5\_0$ (conforme figura 6.16).

Resultado obtido pela aplicação do algoritmo nas versões da classe Elevator:

T' = {Caso_teste1_0, Caso_teste2_0, Caso_teste3_0, Caso_teste4_0, Caso_teste5_0}

Caso_teste1_0 = {Elevator, go, ~Elevator}
Caso_teste2_0 = {Elevator, go, up, ~Elevator}
Caso_teste3_0 = {Elevator, go, up, direction, ~Elevator}
Caso_teste4_0 = {Elevator, go, down, ~Elevator}
Caso_teste5_0 = {Elevator, go, down, direction, ~Elevator}
Caso_teste6_0 = {Elevator, direction, ~Elevator}

Figura 6.16: Conjunto de teste de regressão para a classe *Elevator*.

O exemplo apresentado referente à classe *Elevator* trata do pior caso para a estratégia de seleção de testes de regressão. Como todas as transações, exceto uma, passam pelo nó alterado, referente ao método "go", o ganho com relação a estratégia retesta tudo não é

alto, pois só elimina um caso de teste do conjunto inicial.

Caso a versão beta da classe - *Elevator'* - apresentasse alteração no método "down" ao invés do método "go", o nó que teria sofrido alteração seria **n05**. Ao executar o algoritmo, o conjunto dos arcos alterados seria: $E = \{a05, a07, a09\}$, sendo que o conjunto de teste de regressão **T'** seria composto por todos os casos de teste que passam por esses arcos (aqueles que envolvem o nó **n05**), **T'** seria composto por: *Caso_teste*4_0 e *Caso_teste*5_0. Logo, o ganho com a seleção do conjunto de teste de regressão já representaria uma economia de dois terços nos testes a serem executados.

6.2.2 Manutenção do conjunto de testes de regressão

Para que o conjunto de teste de regressão para uma terceira versão da classe *Elevator* possa ser encontrado, faz-se necessário manter o conjunto de testes da classe sempre atualizado. Dessa forma, sempre que os testes de uma nova versão são planejados, o conjunto de testes deve passar por manutenção.

Para que o conjunto de teste de regressão mantenha-se atualizado, é necessário remover aqueles testes que se enquadram na classificação proposta na seção 3.4. No exemplo proposto, como nenhum método foi removido de uma versão para outra, ou mesmo teve sua funcionalidade alterada, nenhum dos testes seria removido do conjunto inicial. Porém, os testes referentes aos novos métodos devem ser acrescentados ao conjunto inicial para formar o novo conjunto de teste de regressão.

6.3 Teste de Regressão no contexto da subclasse

Testes de regressão devem ser aplicados sempre que (Binder 2000b): uma nova subclasse é criada, uma nova versão da superclasse é desenvolvida, e uma nova versão da subclasse é criada.

A seguir serão mostradas as técnicas adotadas para seleção dos testes de regressão em cada caso.

6.3.1 Criação de uma nova subclasse

Neste caso os testes da superclasse devem ser reaplicados na subclasse e novos testes criados para a subclasse. Para tanto a ConCAT permite o uso da abordagem HIT para determinar quais casos de teste podem ser reutilizados da superclasse, e quais precisam ser gerados para englobar as novas características da subclasse. Para que a técnica HIT seja aplicada, existem algumas restrições que devem ser observadas, conforme descrito no capítulo 5, seção 5.2. Com isso, ter-se-á o primeiro conjunto de testes para a subclasse.

A classe utilizada como exemplo nesse capítulo, *Elevator*, possui uma subclasse denominada *AlarmElevator*, cujo código é apresentado na figura 6.17. A subclasse *AlarmElevator* apresenta dois métodos novos em relação a superclasse: "check_alarm" e "check_weight". Além disso, o método "go" é redefinido. Com isso, o modelo de fluxo de transação também precisa ser atualizado: a figura 6.18 representa a interface pública da subclasse, considerando os métodos herdados da superclasse. Na figura, os nós destacados correspondem aos métodos novos da subclasse e ao método redefinido.

```
100.class AlarmElevator : public Elevator {
101. public:
102.   AlarmElevator(int top_floor) : Elevator(top_floor) {
103.    elevAlarm = new Alarm();
104.    fire_alarm_on = 0;
105.   }
106.   void go(int floor) {
107.    if (!(fire_alarm_on))
108.      Elevator::go(floor);
109.   }
110.   void check_weight(int weight) {
111.    if(weight > MAXWEIGHT) {
112.      elevAlarm->alarm_type = WEIGHT;
113.      check_alarm(elevAlarm);
114.    }
115.   }
116.   void check_alarm(Alarm *x) {
117.    if(x->alarm_type == FIRE) {
118.      x->set_alarm();
119.      fire_alarm_on = 1;
120.    }
121.    else if((x->alarm_type == WEIGHT)
          && !(fire_alarm_on)) {
122.      x->set_alarm();
123.      sleep(10);
124.      x->reset_alarm();
125.    }
126.    else {
127.      x->reset_alarm();
128.      fire_alarm_on = 0;
129.    }
130.   }
131. protect:
132.   Alarm *elevAlarm;
133.   int fire_alarm_on;
134. };

200. class Alarm {
201.  public:
202.   Alarm()
203.     { alarm_on = 0;}
204.   void set_alarm()
205.     { alarm_on = 1;}
206.   void reset_alarm()
207.     { alarm_on = 0;}
208.   int is_alarm_on()
209.     { return alarm_on;}
210.   int alarm_type;
211.  protected:
212.    int alarm_on;
213. };
```

Figura 6.17: Código da subclasse *AlarmElevator* da classe *Elevator*.

Para o teste da subclasse, a especificação base do modelo de testes (representada na figura 6.19) é passada para a ConCAT, indicando que *AlarmElevator* é subclasse de *Elevator* conforme indicado na primeira linha da especificação base (terceiro parâmetro é o nome da superclasse). A partir dessas informações, o protótipo gerará a seqüência de testes que deve ser aplicada para os testes dessa subclasse. Essa seqüência de teste contém apenas os testes para os métodos novos ou que sofreram alterações na subclasse em relação a superclasse, o que é indicado pelas palavras 'novo' e 'redefinido' apresentadas na figura 6.19.

No teste da subclasse são reutilizados os casos de teste da superclasse, T_E, para aqueles métodos que foram herdados sem modificações da interface pública da superclasse. Além disso, cria-se o novo conjunto de testes, T_A, para as novas funcionalidades da classe derivada. Assim, *AlarmElevator* deve ser testada com o conjunto de testes: $T_{sub} = \{T_E \bigcup T_A\}$. Onde T_E foi apresentado na figura 6.5 e $T_A = \{Caso\_teste1\_1 = \{m01, m07, m08, m02\}, Caso\_teste2\_1 = \{m01, m08, m02\}\}$, onde 'm' é a identificação de cada nó mostrados na figura 6.19. Dessa forma T_{sub} será composto por oito casos de teste: seis herdados e dois novos.

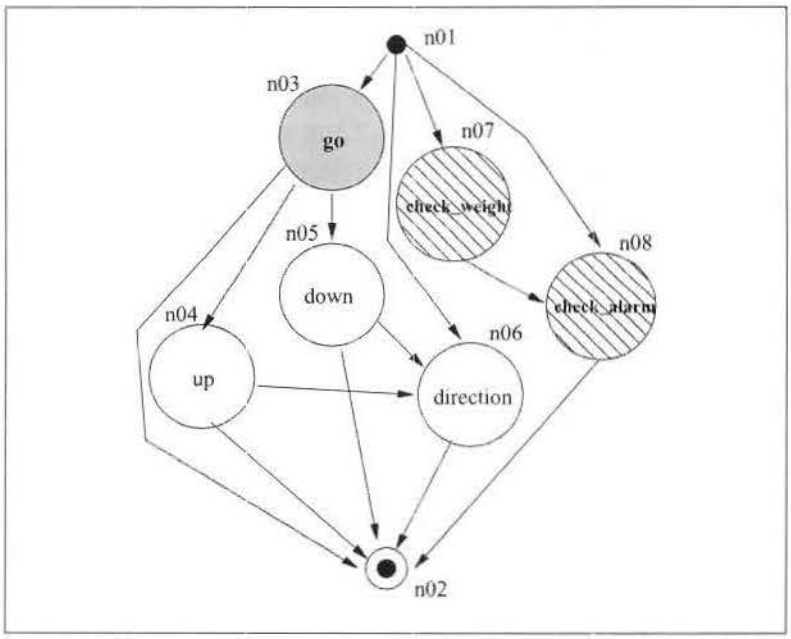

Figura 6.18: Modelo de fluxo de transação para a classe *AlarmElevator*.

```
classe('AlarmElevator',n,'Elevator',[]).

parametros_template([]).

atributo('elevAlarm',string,['Alarm *'],_).
atributo('fire_alarm_on',int,_,_).

metodo(m01,'AlarmElevator',_,construtor,1).
metodo(m02,'~AlarmElevator',_,destrutor,0).
metodo(m03,'go','void',redefinido,1).
metodo(m04,'up','void',recursivo,0).
metodo(m05,'down','int',recursivo,0).
metodo(m06,'direction','Direction',recursivo,0).
metodo(m07,'check_weight','void',novo,1).
metodo(m08,'check_alarm','void',novo,1).

parametro('top_floor',m01,objeto,'int',_).
parametro('floor',m03,objeto,'int',_).
parametro('weight',m07,objeto,'int',_).
parametro('x',m08,string,['Alarm *'],_).

no(n01,sim,4,[m01]).
no(n02,nao,0,[m02]).
no(n03,nao,3,[m03]).
no(n04,nao,2,[m04]).
no(n05,nao,2,[m05]).
no(n06,nao,1,[m06]).
no(n07,nao,1,[m07]).
no(n08,nao,1,[m08]).

arco(n01,n03).
arco(n01,n06).
arco(n01,n07).
arco(n01,n08).
arco(n03,n04).
arco(n03,n05).
arco(n07,n08).
arco(n04,n06).
arco(n05,n06).
arco(n03,n02).
arco(n04,n02).
arco(n05,n02).
arco(n08,n02).
arco(n06,n02).
```

Figura 6.19: Representação do modelo de teste para a classe *AlarmElevator*.

6.3.2 Alteração da superclasse

Neste caso os testes da superclasse devem ser reaplicados tanto na superclasse quanto nas suas subclasses. E os testes da subclasse também devem ser reaplicados.

Para encontrar o conjunto de teste de regressão para a subclasse, T'_{sub}, o algoritmo é aplicado nas duas versões do MFT da subclasse. Dessa forma, serão selecionados apenas testes que passam por segmentos modificados. Como apenas a superclasse mudou, todas as alterações selecionadas na subclasse já terão testes gerados para a superclasse, e eles devem ser reutilizados.

Caso exista alguma interação entre os métodos adicionados à superclasse, que são herdados, e os métodos da subclasse, o conjunto de testes para cobrir essas transações deverá ser criado, T''_{sub}, com o auxílio da ConCAT.

Para o exemplo, ao considerar que a nova versão da superclasse *Elevator'* possui o MFT apresentado na figura 6.15(B), onde é mostrado o novo método "which_floor" e a alteração no método "go". O MFT para a subclasse *AlarmElevator* possuiria a representação mostrada na figura 6.20.

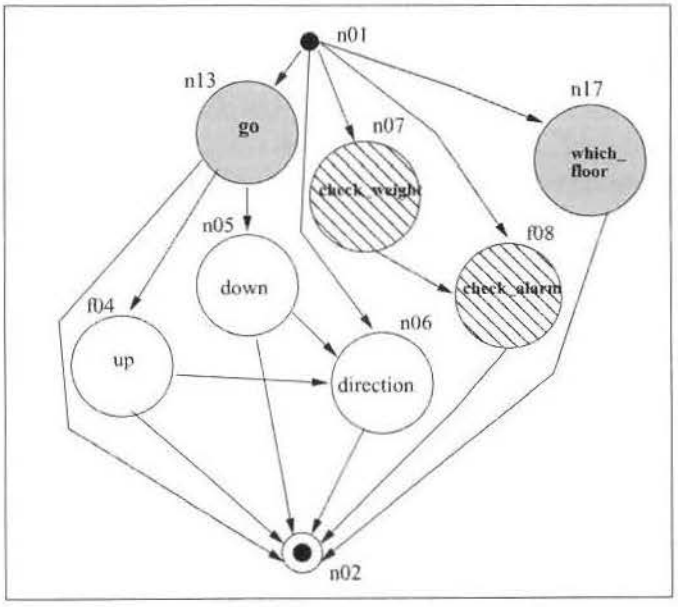

Figura 6.20: Modelo de fluxo de transação para a subclasse *AlarmElevator* da classe *Elevator'*.

Aplicando o algoritmo sobre os dois MFTs das subclasses, o conjunto de teste de regressão seria composto pelo conjunto de teste de regressão da superclasse, já que as alterações que existem entre os grafos são referentes aos métodos herdados da *Elevator'*, então $T'_{sub} = \{Caso\_teste1\_0, Caso\_teste2\_0, Caso\_teste3\_0, Caso\_teste4\_0, Caso\_teste5\_0\}$.

Para encontrar o conjunto de teste T''_{sub} (testes para características novas) para uma subclasse é necessário realizar uma análise de custo-benefício entre: re-instanciar os testes ou reduzir o modelo de fluxo de transação. Essa análise faz-se necessária devido à restrição da abordagem HIT quanto a herança múltipla, logo uma subclasse só pode ser considerada derivada de uma única classe. Dessa forma, os testes existentes para a mesma subclasse na versão anterior não seriam reaproveitados, tendo que ser completamente recriados e instanciados. Para tentar diminuir o impacto desse retrabalho, existe a possibilidade de analisar o MFT da subclasse para eliminar nós não modificados que não interagem com métodos modificados na superclasse. Assim, o modelo representaria apenas os novos métodos e interações, e o conjunto de teste gerado pela ConCAT não repetiria os testes da versão anterior da subclasse.

Ao considerar a redução do MFT de *AlarmElevator*, o modelo apresentado na figura 6.20 seria reduzido pela remoção do nó **n06** (método "direction") que não sofreu alteração e também não interage com as alterações herdadas da superclasse: **n13** e **n17**. Com isso os arcos $(n01, n06)$, $(n04, n06)$, $(n05, n06)$ e $(n06, n02)$ também seriam removidos.

Como o novo método da superclasse *Elevator'*, "which_floor", não interage com métodos da subclasse, o conjunto de testes para novas funcionalidades seria apenas o conjunto herdado da superclasse. Logo, $T''_{sub} = T''_E = \{Caso\_teste9\_0 = \{m01, m09, m02\}\}$.

O conjunto **T"'** para a subclasse seria formado por $T_{sub} \bigcup T'_{sub} \bigcup T''_{sub}$, que compreende aos oito casos de teste de T_{sub} mais o caso de teste para o método novo da superclasse contido em T''_{sub}, formando um conjunto com nove casos de teste. Como o conjunto de teste de regressão da subclasse corresponde ao conjunto da superclasse, os casos já estão contidos em T_{sub}, por isso não são novamente considerados.

6.3.3 Alteração da subclasse

Quando uma nova versão da subclasse é gerada, os testes da superclasse devem ser reaplicados, assim como os testes da subclasse.

Para encontrar o conjunto de teste de regressão T'_{sub} para a subclasse na versão beta, deve-se aplicar o algoritmo (descrito na figura 6.9) utilizando os modelos de fluxo de transação da subclasse e de sua versão alterada. No MFT devem ser consideradas as modificações (alteração, remoção ou inclusão) na especificação da subclasse.

Se a subclasse *AlarmElevator'* na versão beta contivesse alteração no método "up" e o método "check_alarm" tivesse sido substituído por "check_alarm1", o MFT da subclasse possuiria a representação mostrada na figura 6.21 que é uma extensão do modelo da superclasse mostrado na figura 6.15(B), que possui o novo método "which_floor" e alteração no método "go" que também são herdados pela subclasse.

Aplicando o algoritmo sobre os dois MFTs das subclasses, obter-se-ia o seguinte con-

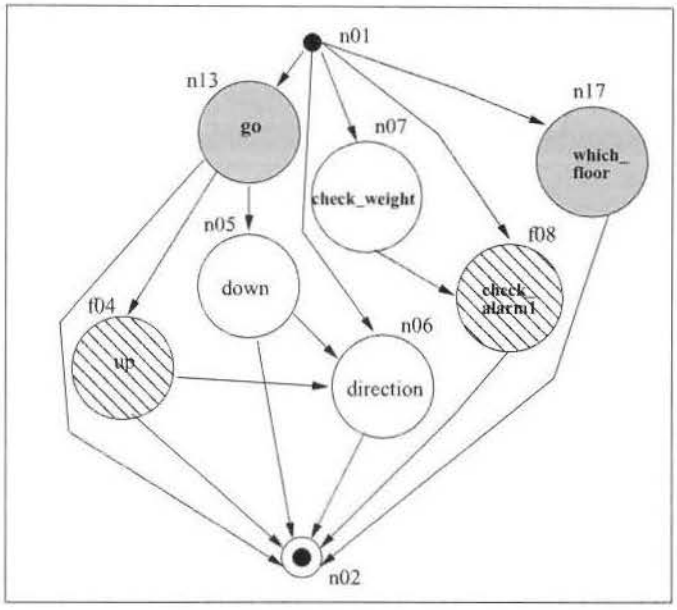

Figura 6.21: Modelo de fluxo de transação para a classe *AlarmElevator'* subclasse de *Elevator'*.

junto de arcos que sofreram alteração: $(n13, f04)$, $(f04, n06)$, $(f04, n02)$. E conseqüentemente o conjunto de teste de regressão seria formado pelas transações que exercitam o nó **f04**, mostrado na figura 6.21. No entanto, como o conjunto de teste de regressão da superclasse é herdado pela subclasse juntamente com as suas alterações, as transações que cobrem esse nó já estão nesse conjunto. Logo, T'_{sub} continuaria a ser idêntico a $T' = \{Caso\_teste1\_0, Caso\_teste2\_0, Caso\_teste3\_0, Caso\_teste4\_0, Caso\_teste5\_0\}$.

Para obtenção do conjunto de testes para as novas funcionalidades da subclasse, T''_{sub}, novamente é necessária a análise do custo da redução do modelo de fluxo de transação ou de re-instanciar os testes.

Uma vez que a alteração da subclasse possui um método novo "check_alarm1", teríamos que o conjunto $T''_{sub} = \{Caso\_teste3\_1 = \{n01, f08, n02\}, Caso\_teste4\_1 = \{n01, n07, f08, n02\}\}$.

O conjunto **T"'** para a subclasse *AlarmElevator'* seria formado por $T_{sub} \bigcup T'_{sub} \bigcup T''_{sub}$, que compreende os 9 casos de teste descritos no final da seção 6.3.2 exceto Caso_teste1_1 e Caso_teste1_2 considerados obsoletos já que o método "check_alarm" foi removido, acrescido dos dois casos novos dessa subclasse, totalizando nove casos de teste. Como T'_{sub} já está incluso no conjunto T_{sub} ele não é gerado novamente.

6.4 Análise da Estratégia

Baseado no framework para avaliação e comparação de técnicas seletivas de teste de regressão, descrita em maiores detalhes na seção 3.3, foi realizada uma análise qualitativa da estratégia proposta:

- Quanto à Inclusão, o algoritmo seleciona todos os arcos que diferem entre as duas versões da classe e todos os casos de teste que exercitam esses arcos, logo todos os testes que atravessam métodos modificados são considerados. Portanto, todos os testes da versão base que podem produzir resultados diferentes em beta são selecionados pela estratégia proposta para teste de regressão, porém ela não pode ser considerada segura já que não é baseada em código.

- Em relação à Precisão, como os casos de teste exercitam seqüências de métodos e não métodos isolados, pode-se dizer que a estratégia não seleciona seqüências que não passam por alterações, logo possui um alto grau de precisão. O algoritmo só seleciona arcos da versão base que diferem entre as duas versões em análise, conseqüentemente apenas casos de teste que exercitam essas diferenças são incluídos no conjunto de teste de regressão.

- A Eficiência da estratégia deve considerar o espaço e tempo gastos para aplicá-la e avaliá-la. Fazendo uma avaliação qualitativa da estratégia proposta, deve ser considerado o custo de armazenar a seqüência de testes base, mais informações de análise requeridas pela estratégia de seleção: MFT e informações sobre a relação dos arcos e casos de teste. No entanto, o MFT já existe não necessitando de espaço extra para armazenamento do grafo, como o que ocorre em (Rothermel et al. 2000). O único espaço extra que deveria ser considerado seria para armazenar associação entre casos de teste e segmentos, porém essa informação é extraída do arquivo gerado pela ConCAT com os objetos de teste, logo também não ocupará espaço extra. Com relação ao tempo pode ser dito que a estratégia é um pouco mais eficiente do que a proposta em (Rothermel et al. 2000) que gera o grafo de fluxo de controle para aplicar a técnica, pois na estratégia proposta o grafo (MFT) já existe: uma classe testável possui seu MFT, logo a construção do modelo de fluxo de transação não faz parte da estratégia. No mais, considerando que o desenvolvedor/equipe de manutenção precisa atualizar os *built-in test* (contratos) e atualizar o código, o testador pode utilizar a fase preliminar de teste de regressão para coletar as informações de análise e executar o algoritmo. Como o algoritmo não é baseado no código, não é preciso esperar a fase crítica para a obtenção de **T'**, pois os modelos de fluxo de transação (MFT para a versão base e para a versão beta) já estariam prontos ainda que a

implementação não estivesse concluída, e assim os segmentos modificados já seriam coletados pelo algoritmo.

- Quanto à Generalidade, a estratégia é aplicável no escopo de classes e suas derivadas necessitando de análise das classes para classificação das diferenças entre versões, além de conhecimento da ferramenta utilizada. Como se trata de uma estratégia independente do código fonte, pois o modelo de fluxo de transação utilizado para a comparação entre as versões é baseado na especificação e não nas particularidades da implementação, a técnica independe do código fonte.

Muito do que Binder (2000b) coloca como análise para a técnica Reteste baseado em Segmento Modificado aplica-se a estratégia proposta nesse trabalho, pois essa é baseada na técnica citada. Porém a grande diferença entre elas está no que é considerado um segmento: nesse trabalho o segmento é uma interação entre métodos da interface pública da classe; já na técnica (descrita em 3.6.4) o segmento é uma instrução do programa.

6.5 Considerações finais

Neste capítulo foi mostrada uma estratégia para testes de regressão seletiva que adapta a técnica de Reteste baseado em segmentos modificados e combina-a com a Técnica Incremental Hierárquica (quando trata-se de subclasses) para uma classe testável. No próximo capítulo será apresentado exemplo da utilização da estratégia proposta para uma biblioteca de classes em C++.

A estratégia para seleção do conjunto de teste de regressão proposta não cobre particularidades da orientação a objetos presentes na linguagem C++, como: polimorfismo, ligação dinâmica, etc. Dado que se trata de testes de unidade, não julgamos necessário tratar desses aspectos, os quais devem ser tratados em testes de integração. No entanto, um aspecto que não foi coberto, mas que merece ser considerado no futuro, é o tratamento de classes compostas.

Segundo Rothermel et al. (2000), para que um conjunto de testes de regressão seja confiável para testar um software modificado, ele precisa incluir testes caixa preta e caixa branca. No entanto, em alguns casos, como acontece com componentes de terceiros, como os softwares CoTS (*commercial off-the-shelf*), o código fonte não está disponível. Assim sendo a estratégia proposta é útil pois pode ser usada inclusive por usuários de componentes.

Como essa estratégia não utiliza o código, ela pode ser aplicada a partir da fase preliminar de teste de regressão, conforme descrito em 3.1. Dessa forma, existe uma economia no tempo gasto para a preparação dos testes de regressão, já que basta existir a especificação de requisitos concluída para que os testes possam ser preparados.

Capítulo 7

Estudos Empíricos realizados

Este capítulo apresenta o uso da estratégia seletiva para teste de regressão para uma aplicação real. O aplicativo utilizado é uma biblioteca de classes em C++ para suporte ao protocolo UDP.

Este estudo teve como objetivo principal verificar a aplicabilidade da proposta para classes de um sistema que evoluem por várias versões.

O capítulo está organizado da seguinte forma: na primeira seção é apresentado uma descrição do aplicativo sob teste; a seção 7.2 descreve a preparação necessária para os testes; a seção 7.3 apresenta os procedimentos utilizados para a realização dos testes de regressão; a seção 7.4 traz os resultados enumerados e analisados para o estudo realizado; por fim algumas considerações.

7.1 Descrição do aplicativo sob teste

Para a realização de uma avaliação empírica da estratégia proposta no capítulo anterior para seleção de testes de regressão, obteve-se uma biblioteca de classes em C++ disponível entre as bibliotecas na linguagem C++ da GNU que oferece abstração para vários serviços de sistema de um modo portável denominada CommonC++[1]. Dentre as classes disponíveis nessa biblioteca serão consideradas as classes Socket e sua classe derivada UDPSocket. A primeira é uma classe abstrata, e seus métodos só serão testados no contexto da subclasse dado que esta herda a interface pública da superclasse.

Após uma análise das diversas versões disponíveis da biblioteca CommonC++ foram escolhidas cinco delas (versões 1.0.0, 1.2.0, 1.3.1, 1.4.3, 1.5.0 e 1.6.0) para a realização de testes de regressão para exemplificar a estratégia proposta no capítulo anterior. Essa escolha foi baseada no número de alterações (modificações na implementação ou novos

[1]Os fontes das versões dessa biblioteca foram extraídos de ftp://ftp.gnu.org/pub/gnu/commonc++/ em janeiro/2002

métodos) para as duas classes alvo: Socket e UDPSocket. Para isso, foi utilizado o aplicativo **diff** do Unix para levantar as diferenças entre cada versão das classes alvo e suas versões sucessoras.

As classes implementam o protocolo de transmissão de datagramas sem conexão e não confiável UDP (*User Datagram Protocol*) utilizando uma implementação de *socket*. UDP faz parte da camada de transporte, considerando o esquema de organização de protocolos em camadas. Essa camada fica acima da camada de Internet Protocol e abaixo da camada do aplicativo. O protocolo UDP fornece o mecanismo principal utilizado pelos programas aplicativos para enviar datagramas a outros programas iguais (Comer 1998), pois fornece portas de protocolo para estabelecer a distinção entre os diversos programas executados em uma única máquina identificando a origem e o destino do datagrama enviado.

7.2 Preparação para os testes

Para a execução dos testes as classes são instrumentadas para que passem a ser derivadas da classe *Autoteste*, definida em (Toyota 2000), que padroniza a utilização dos métodos BIT: um relator e outro para teste de invariante da classe. Além disso, nessa classe estão definidas macros para o teste de predicados.

Nas classes alvo do teste os métodos herdados da *Autoteste* devem ser redefinidos para refletir as informações da classe. Para que os mecanismos BIT não interfiram na execução da classe quando esta não estiver em teste, eles devem ser declarados entre as diretivas do pré-compilador: "#ifdef TESTE" e "#endif".

A partir da especificação de um protocolo de comunicação UDP, e dos métodos propostos nas classes *Socket* e *UDPSocket* foram projetados modelos de teste (representados por Modelo de Fluxo de Transação) para as classes nas diversas versões analisadas. Além da instrumentação necessária para tornar as classes testáveis, conforme mostrado na seção 6.1.3, também foi incluído em cada alteração um comando para assinalar a cada vez que uma delas é executada. Um exemplo de utilização desse comando pode ser observado a partir das classes da segunda versão do aplicativo, conforme trecho do código em destaque na figura 7.1.

Como o resultado da execução dos casos de teste e mecanismos de teste embutido são armazenados num arquivo de resultados denominado *Result.txt*, as informações de cobertura de alterações também são gravadas nesse arquivo destacadas com uma seqüência de sinais: ">>>" sucedida por uma mensagem que identifica o método alterado. A figura 7.2 apresenta o caso de teste *Caso_teste18_0* que exercita o método mostrado como exemplo na figura 7.1 e dentro de um retângulo com a linha pontilhada o resultado da execução deste caso de teste.

A exemplo do que ocorreu no trabalho proposto por Ukuma (2002) para desen-

```
bool Socket::isPending(sockpend_t pending, unsigned long timeout)
{
  int status;
  #ifdef TESTE
    ofstream arq(ARQUIVO_RESULTADOS,ios::app);
    if (!arq) {
      cout << "Erro abrindo arquivo! \n";
      exit(0);
    }
    else {
      arq << ">>> isPending = alteracao na implementacao. \n";
      arq.flush();
    }
  #endif
  #ifdef HAVE_POLL
    struct pollfd pfd;

    pfd.fd = so;
    pfd.revents = 0;
    switch(pending)
    {
      case SOCKET_PENDING_INPUT:
            pfd.events = POLLIN;
            break;
      case SOCKET_PENDING_OUTPUT:
            pfd.events = POLLOUT;
            break;
      case SOCKET_PENDING_ERROR:
            pfd.events = POLLERR | POLLHUP;
            break;
    }

    if(timeout == TIMEOUT_INF)
      status = poll(&pfd, 1, -1);
    else
      status = poll(&pfd, 1, timeout);

    if(status < 1)
      return false;
    if(pfd.revents & pfd.events)
      return true;
    return false;
  #else
    struct timeval tv;
    fd_set grp;
    struct timeval *tvp = &tv;

    if(timeout == TIMEOUT_INF)
      tvp = NULL;
    else
    {
      tv.tv_usec = (timeout % 1000) * 1000;
      tv.tv_sec = timeout / 1000;
    }

    FD_ZERO(&grp);
    FD_SET(so, &grp);
    switch(pending)
    {
      case SOCKET_PENDING_INPUT:
            status = select(so + 1, &grp, NULL, NULL, tvp);
            break;
      case SOCKET_PENDING_OUTPUT:
            status = select(so + 1, NULL, &grp, NULL, tvp);
            break;
      case SOCKET_PENDING_ERROR:
            status = select(so + 1, NULL, NULL, &grp, tvp);
            break;
    }
    if(status < 1)
      return false;
    if(FD_ISSET(so, &grp))
      return true;
    return false;
  #endif
}
```

Figura 7.1: Código do método *isPending* com comando para destaque de alterações entre versões.

```
template <class Tipo>
void Caso_teste18_0(Tipo* cst) {
  char* met= new char[100];
  sockpend_t pendente = SOCKET_PENDING_INPUT;
  timeout_t tempo= 5000;
  bool resultado;

  ofstream arq(ARQUIVO_RESULTADOS,ios::app);
  if (! arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Gravando informacoes caso de teste 18_0! \n";
  try {
    cst->Testa_Invariante();
    strcpy(met,"isPending(sockpend_t pendente,");
    strcat(met,itoa(tempo));
    strcat(met,")");
    resultado = cst->isPending(pendente,tempo);
    arq << "Metodo chamado: " << met << "  resultado= " << resultado << "\n";
    arq.flush();

    cst->Testa_Invariante();
    arq << "Caso_teste18_0 OK!\n";
    arq.flush(); arq << "\n"; arq.close();
    cst->Relator(ARQUIVO_RESULTADOS);
    delete cst;
  }
  catch(char* c) {
    arq << "Caso_teste18_0 \n";
    arq.flush(); cout << "..."; arq << c << "\n";
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator(ARQUIVO_RESULTADOS);
    arq << "\n";
    arq.close();
  }

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

```
>>> isPending = alteracao na implementacao.
Metodo chamado: isPending(sockpend_t pendente,5000)  resultado= 0
Caso_teste18_0 OK!

THROWN: 0
BROADCAST: 0
ROUTE: 1
KEEPALIVE: 0
STATE: 2
ERRID: 0
ERRSTR: (null)
```

Figura 7.2: Resultado parcial da execução do caso de teste Caso_teste18_0.

volvimento de componentes autotestáveis, aqui também foi necessário realizar algumas adaptações no driver específico gerado pela ConCAT:

1. quando é declarado na especificação que uma classe é derivada de outra, o conjunto de teste da classe base que deve ser reutilizado na classe derivada deveria ser indicado no driver específico da classe derivada, porém a ConCAT não faz isso, logo o testador precisa incluir o conjunto de teste e a chamada para cada caso de teste a ser executado;

2. a numeração dos casos de teste deveria ser diferenciada quando a classe é derivada, senão a inclusão da referência aos casos de teste herdados duplica a identificação do caso de teste e torna os testes imprevisíveis. O testador precisa alterar a numeração dos casos de teste da classe derivada, uma sugestão é alterar o número no final da identificação do caso de teste de zero para um, por exemplo: um caso de teste da classe derivada identificado como "Caso_teste4_0" seria alterado para "Caso_teste4_1". Além disso para diferenciar os casos para cada versão, foi acrescentado mais um número no identificador do caso de teste, logo o caso de teste citado para a segunda versão seria identificado como "Caso_teste4_1_2".

Dentre o conjunto de classes utilizadas para a realização desse estudo empírico, existe um método que viola uma das premissas da técnica HIT quanto a visibilidade dos métodos:

ele passa a ser mais restrito na subclasse que o herda. Para prosseguir com a utilização da ConCAT para a obtenção da seqüência de testes e driver específico esse método foi desconsiderado.

Para que a classe *UDPSocket* fosse testada, foi necessário indicar a porta de entrada e saída para comunicação de dados na mesma máquina, provocando a troca de mensagens por um único processo a fim de simular a troca de informações entre processos em rede.

7.3 Procedimento

O Modelo de Fluxo de Transação obtido para as classes *Socket* e *UDPSocket* são apresentados no apêndice A, nas figuras A.1 e A.2, respectivamente.

Para as duas classes da versão base da biblioteca CommonC++, a especificação base de teste obtida a partir do modelo de teste das classes é apresentada no apêndice B.

A partir da representação do modelo de testes, a ConCAT obtém as seqüências de teste para que todas as possíveis transações sejam percorridas. Cada transação é representada por um objeto de teste que identifica quais nós do grafo (MFT) serão exercitados, e os relaciona com o caso de teste que contempla a transação. Para a classe *Socket* o conjunto de testes gerado para a primeira versão possui vinte objetos de teste conforme mostrado na figura C.1, e o conjunto para a subclasse *UDPSocket* com trinta e oito casos de teste pode ser verificado na figura C.2.

Como a classe *Socket* é abstrata, os casos de teste gerados para essa classe, só foram executados na subclasse *UDPSocket*, já que não é possível instanciar um objeto de classe abstrata. O driver da classe *Socket* foi alterado para instanciar objetos da subclasse como pode ser observado no apêndice D.1. No driver específico gerado pela ConCAT para testar a subclasse foram incluídas chamadas para os casos de teste que esta herda da *Socket*, conforme seção D.2. No apêndice D também são mostrados alguns exemplos de casos de teste gerados para a classe UDPSocket na seção D.3.

Antes da execução do driver para realizar os testes em *Socket* e *UDPSocket*, é necessário torná-las testáveis. Para tanto, no cabeçalho da classe *Socket* é alterada a declaração da classe, tornando *Socket* subclasse de *Autoteste*. Como UDPSocket é subclasse de *Socket*, ela herda a interface pública da *Autoteste* também. Além disso, foram redefinidos os métodos Relator e TestaInvariante na classe *Socket*, conforme figura 7.3.

Além disso, as invariantes e assertivas devem ser incluídas no código fonte das classes para permitir a observação do estado de seus objetos e facilitar a comparação entre suas versões. Na classe *UDPSocket* foram inseridas pós-condições manualmente nos métodos: construtor e "getPeer". Alguns exemplos da inserção das pós-condições na classe *UDPSocket* podem ser visualizados na figura 7.4.

Após a execução dos testes iniciais para a primeira versão da classe *UDPSocket* e coleta

```
void Socket::Testa_Invariante()
{
  Invariante(this->state == SOCKET_BOUND);
}

void Socket::Relator(char* arquivo)
{
  ofstream resultados(arquivo,ios::app);
  if (!resultados) {
    cout << "Erro abrindo arquivo relator! \n";
    exit(0);
  }
  else {
    resultados << "THROWN: " << flags.thrown << "\n";
    resultados << "BROADCAST: " << flags.broadcast << "\n";
    resultados << "ROUTE: " << flags.route << "\n";
    resultados << "KEEPALIVE: " << flags.keepalive << "\n";
    resultados << "STATE: " << state << "\n";
    resultados.flush();
    resultados << "\n";
    resultados.close();
  }
}
```

Figura 7.3: Implementação para os Métodos Relator e TestaInvariante em *Socket*.

```
UDPSocket::UDPSocket(InetAddress &ia, short port) :
Socket(AF_INET, SOCK_DGRAM, 0)
{
  peer.sin_family = AF_INET;
  peer.sin_addr = getaddress(ia);
  peer.sin_port = htons(port);
  if(bind(so, (struct sockaddr *)&peer, sizeof(peer))) {
    endSocket();
    Error(SOCKET_BINDING_FAILED);
    return;
  }
  state = SOCKET_BOUND;

  Pos_condicao(state == SOCKET_BOUND);
}
```

```
InetHostAddress UDPSocket::getPeer(short *port)
{
  char buf;
  socklen_t len = sizeof(peer);
  int rtn = ::recvfrom(so, &buf, 1, MSG_PEEK, (struct sockaddr *)&peer, &len);

  if(rtn < 1) {
    if(port)
      *port = 0;

    memset(&peer, 0, sizeof(peer));
  }
  else {
    if(port)
      *port = ntohs(peer.sin_port);
  }

  Pos_condicao(rtn >= 0);

  return InetHostAddress(peer.sin_addr);
}
```

Figura 7.4: Exemplos de pós-condições inseridas na classe *UDPSocket*.

dos resultados obtidos, assim como validação das pós condições, começa o procedimento para encontrar o conjunto de teste de regressão para testar as classes na segunda versão.

Com o algoritmo proposto para seleção de testes de regressão, os testes que atravessam métodos modificados são selecionados a partir da comparação dos modelos de fluxo de transação para as duas versões da classe *Socket* e depois da classe *UDPSocket*. Dessa forma, os testes que passam pelas modificações feitas no programa de uma versão para a outra são considerados para o conjunto de teste de regressão, portanto, os testes com maiores chances de encontrar defeitos.

Para a obtenção do conjunto de teste de regressão **T'** para testar a segunda versão da classe *UDPSocket* o algoritmo é utilizado para comparar os MFT's das duas subclasses. Além desse conjunto, para o teste da subclasse, o conjunto de teste de regressão obtido para a classe *Socket* também deve ser aplicado na subclasse, conforme seção 6.3.

Antes da execução dos testes de regressão para a classe *UDPSocket* em beta é necessário adequar as pós-condições e invariantes, caso tenha ocorrido alguma alteração de contrato.

Aplica-se o conjunto **T'** para testar a subclasse *UDPSocket*. Porém esse conjunto não exercita os métodos novos da classe, por isso, a ConCAT é novamente utilizada para gerar o conjunto de teste **T"**.

A partir desse ponto, obtém-se o conjunto **T"'**, que será o conjunto base para os testes da versão seguinte, onde: $T''' = T'' \bigcup T' \bigcup (T - \{obsoletos\})$.

Esse processo foi executado para as seis versões analisadas nesse estudo. Sempre o MFT da classe nas duas versões sob teste são considerados para a execução do algoritmo descrito em 6.9. Além do conjunto de teste de regressão seletivo, resultante da aplicação do algoritmo, faz-se necessário encontrar o conjunto que irá testar as novas funcionalidades incorporadas na versão beta.

7.4 Resultados

Após a obtenção e execução do conjunto seletivo de teste de regressão, foi realizada uma análise dos resultados coletados em cada versão testada.

A tabela 7.1 apresenta o número de LOC (*Lines Of Code*) para cada arquivo que possui o código fonte das classes envolvidas, além do tamanho binário de cada arquivo. Essa medição é realizada em duas fases: nos arquivos originais e depois da instrumentação das classes para tornarem-se testáveis.

Com os valores dos arquivos é possível observar que o aumento das classes em termos de linhas de código foi por volta a 4,5% do tamanho original, e o aumento do tamanho dos arquivos que contém as classes analisadas foi aproximadamente 5,4%.

Em relação as classes envolvidas *Socket* e *UDPSocket*, observa-se que a interface pública da classe *Socket* é composta por 16 métodos, já a subclasse *UDPSocket* possui 8 métodos

		Instrumentação			
		antes		depois	
Arquivo	Versão	nº *LOC*	tamanho (KB)	nº *LOC*	tamanho (KB)
socket.h	1.0.0	1524	43,1	1559	43,7
	1.2.0	1591	44,9	1608	45,3
	1.3.1	1815	53,8	1831	54
	1.4.3	1900	55,8	1920	56,1
	1.5.0	1911	56,3	1925	56,5
	1.6.0	2052	60,4	2066	60,6
socket.cpp	1.0.0	877	16,6	933	18,3
	1.2.0	968	18,6	1168	23,3
	1.3.1	1061	20,9	1138	22,7
	1.4.3	1251	24,5	1315	26,1
	1.5.0	1300	25,2	1354	26,6
	1.6.0	1365	26,3	1422	27,8

Tabela 7.1: Comparação dos arquivos antes e após instrumentação para cada versão.

na sua interface pública mais os métodos herdados da superclasse, totalizando 23 métodos, já que um dos métodos da superclasse será desconsiderado conforme explicado na seção 7.2.

O conjunto de testes para a primeira versão da superclasse contém vinte casos de teste. Para a subclasse, foram gerados 38 casos de teste para as características novas nessa classe e mais os testes herdados da superclasse, exceto os obsoletos. Dessa forma, o conjunto de testes para a primeira versão da subclasse contém 51 casos de teste.

Na tabela 7.2 são mostrados os resultados relacionados aos testes das classes nas versões. Para cada classe é apresentado o número de casos de teste (*tamanho*) no conjunto inicial **T**, no conjunto de testes selecionado para regressão **T'** e também no conjunto de tetes para as características novas das classes **T"**. O conjunto **T** é formado pelo conjunto **T"'** obtido na atualização do conjunto de teste para regressão ao final de cada versão.

Com a utilização do algoritmo para teste de regressão seletivo, foi possível diminuir o número de casos de teste a serem reaplicados no aplicativo para verificar aqueles métodos que não sofreram alteração de contrato.

Para algumas versões, o conjunto **T'** ficou vazio, pois a única alteração, em relação a classe na versão anterior, foi a inserção de um novo método (às vezes esse novo método é utilizado para substituir um anteriormente existente), e como a estratégia é seletiva apenas testes que exercitam alterações são escolhidos para serem reaplicados. Dessa forma, o conjunto **T"** possuirá transações para cobrir o novo método. Isso é o que ocorre nas versão 1.5.0 e 1.6.0 onde a classe *Socket* é acrescida de um novo método. No caso

Classe	Versão	tamanho T	tamanho T'	tamanho T"
Socket	1.0.0	–	–	20
	1.2.0	20	4	4
	1.3.1	20	4	0
	1.4.3	20	1	0
	1.5.0	20	0	2
	1.6.0	20	0	1
UDPSocket	1.0.0	–	–	51
	1.2.0	51	4	35
	1.3.1	53	12	0
	1.4.3	54	39	0
	1.5.0	54	0	12
	1.6.0	54	0	1

Tabela 7.2: Conjuntos de testes T, T' e T" para as classes em cada versão.

da versão 1.5.0, um dos métodos já existente na classe é substituído por outro, logo o conjunto **T"'** deve ser atualizado para a remoção dos testes que exercitavam o método removido para formar o conjunto inicial **T** para os testes da próxima versão.

Nos testes para a classe *UDPSocket* da versão 1.2.0 foi obtido o maior número de casos de teste para o conjunto de testes para características novas, **T"**, devido ao grande número de métodos que foram classificados como novos. Já nos testes da versão 1.4.3 foi obtido o maior número de casos de teste de regressão, **T'**, a serem executados, pois o método que sofreu alteração pertence à maioria das transações que exercitam a classe.

Para analisar a inclusão (conforme seção 3.6.4) foram também aplicados os testes $(T - \{obsoletos\}) - T'$, ou seja, os testes válidos mas que não foram selecionados pela técnica utilizada. O objetivo dessa etapa foi garantir que todos os testes que exercitavam as alterações de implementação da classe de uma versão para outra haviam sido selecionados pela estratégia seletiva de teste de regressão.

A tabela 7.3 apresentada a quantidade de alterações de implementação existentes em cada versão, além do número de alterações cobertas pelo conjunto de teste seletivo de regressão **T'** e pelo conjunto inicial menos o conjunto selecionado para teste de regressão **T - T'**.

Conforme os valores mostrados na tabela 7.3 verifica-se que todas as alterações de implementação das classes foram cobertas pelo conjunto seletivo de casos de teste. Além das alterações de implementação as classes também apresentam alterações de especificação que são cobertas pelo conjunto de teste **T"**, e como o objetivo da tabela era mostrar a cobertura obtida por **T'**, essas alterações não foram quantificadas nessa tabela.

Classe	Versão	Alteração de Implementação	Alterações cobertas	
			T'	T - T'
Socket	1.2.0	4	4	0
	1.3.1	2	2	0
	1.4.3	1	1	0
	1.5.0	0	0	0
	1.6.0	0	0	0
UDPSocket	1.2.0	4	4	0
	1.3.1	2	2	0
	1.4.3	2	2	0
	1.5.0	0	0	0
	1.6.0	0	0	0

Tabela 7.3: Alterações existentes e sua cobertura por T' e por T - T'.

7.5 Considerações Finais

Este capítulo apresentou um estudo de caso realizado para exercitar a estratégia seletiva de teste de regressão proposta e uma avaliação dos resultados apresentados. No próximo capítulo são apresentadas sugestões de trabalhos futuros e as conclusões deste trabalho.

Capítulo 8

Conclusões

Neste trabalho foi apresentada uma estratégia seletiva para teste de regressão baseada em especificação. A utilização do conceito de autoteste possibilitou a utilização de classes testáveis conforme proposto por (Toyota 2000), que facilitaram os testes, pois cada classe já possui seu modelo de fluxo de transação e casos de teste associados. Para a geração do conjunto inicial de testes e para a geração de testes para características novas de cada classe na versão sob teste foi utilizada a ConCAT que facilita o reuso de testes das classes, pois ela faz uso da técnica incremental hierárquica.

A atividade de teste de regressão, embora para duas classes consideradas simples por versão, gera um custo de muitos dias de trabalho, mesmo com o auxílio da estratégia proposta. Todos os passos que puderem ser automatizados contribuirão para um teste de regressão menos caro.

8.1 Contribuições

A principal contribuição deste trabalho foi apresentar uma estratégia para teste de regressão seletiva e baseada em especificação (caixa-preta), portanto independente da implementação do sistema. Além de propor uma adaptação da técnica de reteste baseado em segmento modificado para utilização a partir do modelo de fluxo de transação que representa cada classe.

Outras contribuições apresentadas por esse trabalho:

- Um estudo dos padrões para teste de regressão seletivo para verificar o mais apropriado para a realidade desse trabalho;
- Utilização da técnica incremental hierárquica para facilitar o reuso de testes para subclasses;

- Discussão para seleção de testes de regressão para subclasses, buscando o aproveitamento dos testes já identificados para a subclasse em versões anteriores, logo sem desconsiderar o histórico dos testes;
- Realização de um estudo empírico envolvendo uma biblioteca real em seis versões, onde foi possível validar a estratégia proposta;
- Ajuda para manutenção do sistema porque as classes mantêm informações de teste.

A estratégia seletiva para teste de regressão está parcialmente limitada à linguagem C++ devido à utilização da ConCAT para a geração do conjunto inicial de testes e testes para características novas. Porém, a estratégia depende de um modelo de fluxo de transação que é um modelo de teste que independe da implementação do sistema.

8.2 Trabalhos Futuros

Como sugestões de trabalhos futuros são propostas:

- Continuação dos estudos empíricos para a avaliação da estratégia proposta, principalmente com a utilização de um maior número de classes.
- Automatização da técnica de seleção de teste de regressão proposta.
- Melhorias no protótipo ConCAT para facilitar a inserção dos mecanismos de teste embutido, prover dados para os casos de teste e fornecer mecanismos para identificação dos casos de teste gerados para garantir a diferenciação de uma versão para outra.
- Generalização do protótipo ConCAT para implementar a técnica de reteste baseado em segmentos modificados, que poderia ser utilizada para auxiliar nos testes de regressão.
- Uma ferramenta para selecionar o conjunto de teste base (**T'''**) para cada versão da classe a ser testada, mantendo sempre atualizado o conjunto de testes de regressão.

Bibliografia

Beizer, B. (1990). *Software Testing Techniques*, International Thomson Computer Press.

Beizer, B. (1995). *Black-Box Testing*, John Wiley e Sons.

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

Binder, R. V. (2000a). *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison Wesley Longman.

Binder, R. V. (2000b). *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison Wesley Longman, chapter 15 - Regression Testing, pp. 755 – 797.

Binder, R. V. (2001). Testing object-oriented systems: A status report, http://www.rbsc.com/pages/ootstat.html. Artigo publicado em American Programmer em April 1994, e reimpresso em April 1995 em CrossTalk.

Binkley, D. (1996). Semantics guided regression test cost reduction, *IEEE Transactions on Software Engineering* **23**(8): 498–516.

Buzato, L. E. & Rubira, C. M. F. (1998). *Construção de Sistemas Orientados a Objetos Confiáveis*, Departamento de Ciência da Computação/IM e Núcleo de Computação Eletrônica e COPPE Sistemas - UFRJ, chapter 2 - Construção de Software Orientado a Objetos, pp. 21 – 80. 11a. Escola de Computação.

Chen, Y., Probert, R. L. & Sims, D. P. (2002). Specification-based regression test selection with risk analysis, *Conference of the Center for Advanced Studies on Collaborative research*, pp. 1–14.

Comer, D. E. (1998). *Interligação em Rede com TCP/IP: Princípios, protocolos e arquitetura*, terceira edição edn, Editora Campus. Traduzição de ARX Publicações.

Fowler, M. & Scott, K. (1997). *UML Distilled*, Addison-Wesley.

Granja, I. & Jino, M. (1999). Techniques for regression testing: Selecting test casesets taylored to possibly modified functionalities, *Third European Conference on Software Maintenance & Reengineering*, IEEE, pp. 2–11.

Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A. & Rothermel, G. (1998). An empirical study of regression test selection techniques, *The 20th International Conference on Software Engineering* pp. 188–197.

Harrold, M. J. & et. al (2001). Regression test selection for java software, *ACM SIGPLAN Notices* **36**(11): 312 – 326.

Harrold, M. J., McGregor, J. D. & Fitzpatrick, K. (1992). Incremental testing of object-oriented class structures, *The 14th International Conference on Software Engineering* pp. 68–80.

Harrold, M. J. & Rothermel, G. (1994). A framework for evaluating regression test selection techniques, *16th International Conference on Software Engineering*, IEEE, pp. 201–210.

Harrold, M. J. & Rothermel, G. (1996). Analyzing regression test selection techniques, *IEEE Transactions on Software Engineering* **22**(8): 529–551.

Harrold, M. J. & Rothermel, G. (1997). A safe, efficient regression test selection technique, *ACM Transactions on Software Engineering and Methodology* **6**(2): 173–210.

Harrold, M. J. & Rothermel, G. (1998). Empirical studies of a safe regression test selection technique, *IEEE Transactions on Software Engineering* **24**(6): 401–419.

Hoffman, D. M. (1989). Hardware testing and software ics, *Proceedings of Pacific NW Software Quality Conference* .

IEEE (1990). Glossary of software engineering terminology, IEEE Standard. Padrão 610.12/1990.

Jang, Y. K., Chae, H. S., Kwon, Y. R. & Bae, D. H. (1998). Change impact analysis for a class hierarchy, *Conference on Asia Pacific Software Engineering*, pp. 304 – 313.

Kung, D. C., Gao, J., Hsia, P., Toyoshima, Y. & Chen, C. (1996). On regression testing of object-oriented programs, *Journal Systems Software* (32): 21–40.

Leite, J. C. B. & Loques, O. L. (1987). Introdução à tolerância a falhas, *II Simpósio de Computação Tolerante a Falhas*.

Leung, H. K. N. & White, L. J. (1991). A cost model to compare regression test estrategies.

Martimiano, L. A. F. (1999). *Estudo de técnicas de teste de regressão baseado em mutação seletiva*, Master's thesis, Instituto de Ciências Matemáticas e de Computação. USP - São Carlos.

Martins, E. (1999). Verificação e validação de software, *Instituto de Computação - UNICAMP.*

Pressman, R. S. (1997a). *Software Engineering: A Practitioner's Approach*, McGraw-Hill Companies.

Pressman, R. S. (1997b). *Software Engineering: A Practitioner's Approach*, McGraw-Hill Companies, chapter Object-Oriented Concepts and Principles, pp. 551 – 580.

Purchase, J. A. & Winder, R. L. (1991). Debugging tools for object-oriented programming, *Journal of Object-Oriented Programming* **4**(3): 10–27.

Rothermel, G. & Harrold, M. J. (1994). Selecting regression tests for object-oriented software, *IEEE Proceedings of International Conference on Software Maintenance* pp. 14–25.

Rothermel, G., Harrold, M. J. & Dedhia, J. (2000). Regression test selection for c++ software, *Journal of Software Testing, Verification, and Reliability* **10**(2): 1–35.

Siegel, S. (1996). *Object-Oriented Software Testing: a Hierarchical Approach*, John Wiley e Sons.

Toyota, C. M. (2000). *Melhoria de testabilidade de classes usando o conceito de autoteste*, Master's thesis, Instituto de Computação. Universidade Estadual de Campinas.

Toyota, C. M. & Martins, E. (1999). Construção de classes autotestáveis, *VIII Simpósio de Computação Tolerante a Falhas* pp. 196–209.

Ukuma, L. H. (2002). *Uma estratégia para o desenvolvimento de componentes de software autotestáveis*, Master's thesis, Instituto de Computação. Universidade Estadual de Campinas.

Wang, Y., King, G., Court, I., Ross, M. & Staples, G. (1997). On built-in tests in object-oriented reengineering, *5th ACM Symposium on the Foundations of Software Engineering / 6th European Conference on Software Engineering / Workshop on Object-Oriented Reengineering.*

Wang, Y., King, G. & Wickburg, H. (1999). A method for built-in tests in component-based software maintenance, *Third European Conference on Software Maintenance & Reengineering*, IEEE, pp. 186–189.

Wong, E., Horgan, J. R., London, S. & Bellcore, H. A. (1997). A study of effective regression testing in practice, *Eighth Internacional Symposium on Software Reliability Engineering - ISSRE*, IEEE, pp. 264 – 273.

Wong, E., Maldonado, J. C. & Delamaro, M. E. (1997). Reducing the cost of regression testing by using selective mutation, *VIII Conferência Internacional de Tecnologia de Software: Qualidade de Software.*

Apêndice A

Modelo de Fluxo de Transação - Socket e UDPSocket

Neste apêndice são apresentados os modelos de fluxo de transação para as classes Socket (figura A.1) e UDPSocket (figura A.2) da primeira versão da biblioteca CommonC++ utilizada para os testes de regressão.

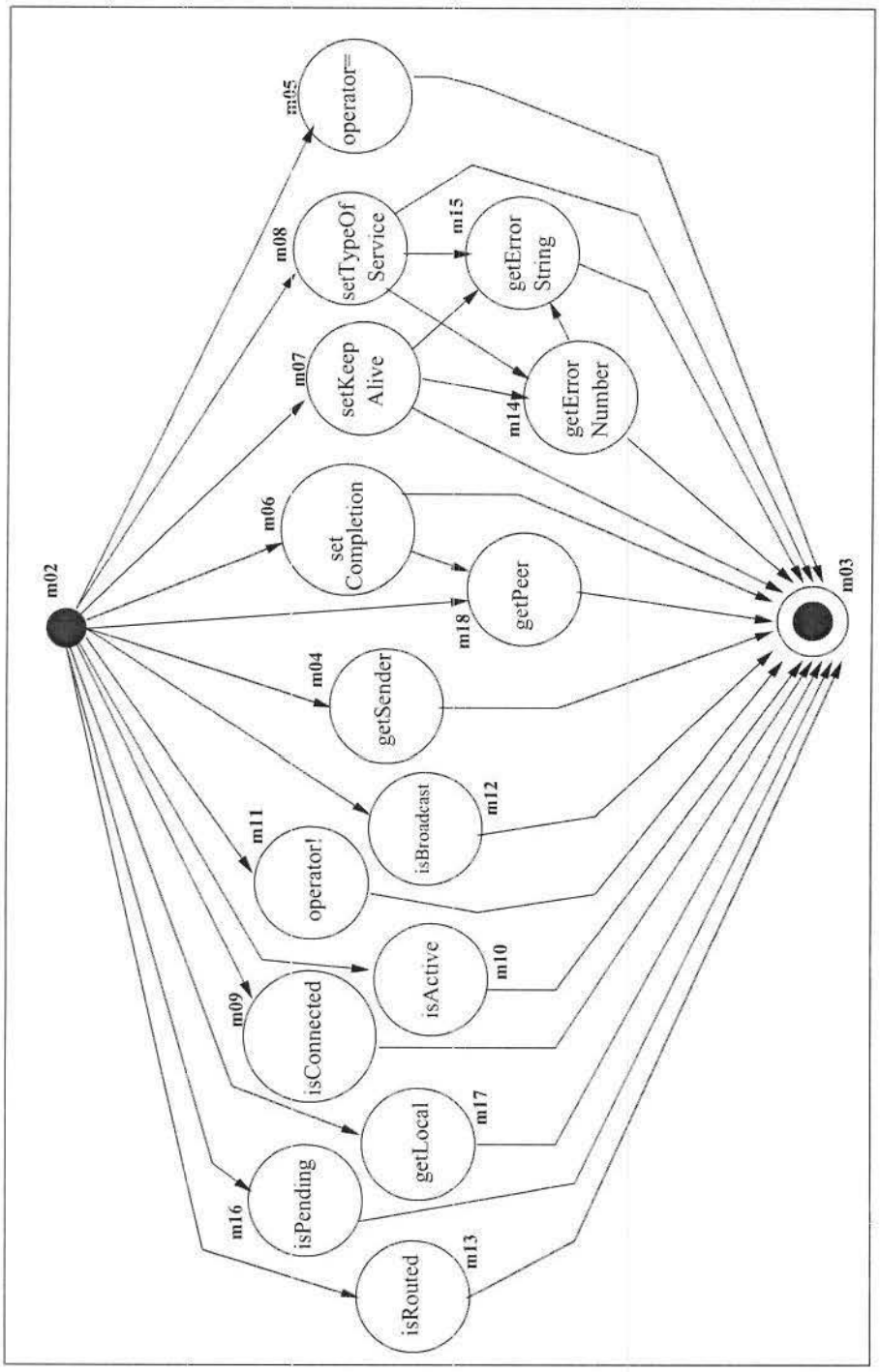

Figura A.1: Modelo de Fluxo de Transação da classe Socket - versão base.

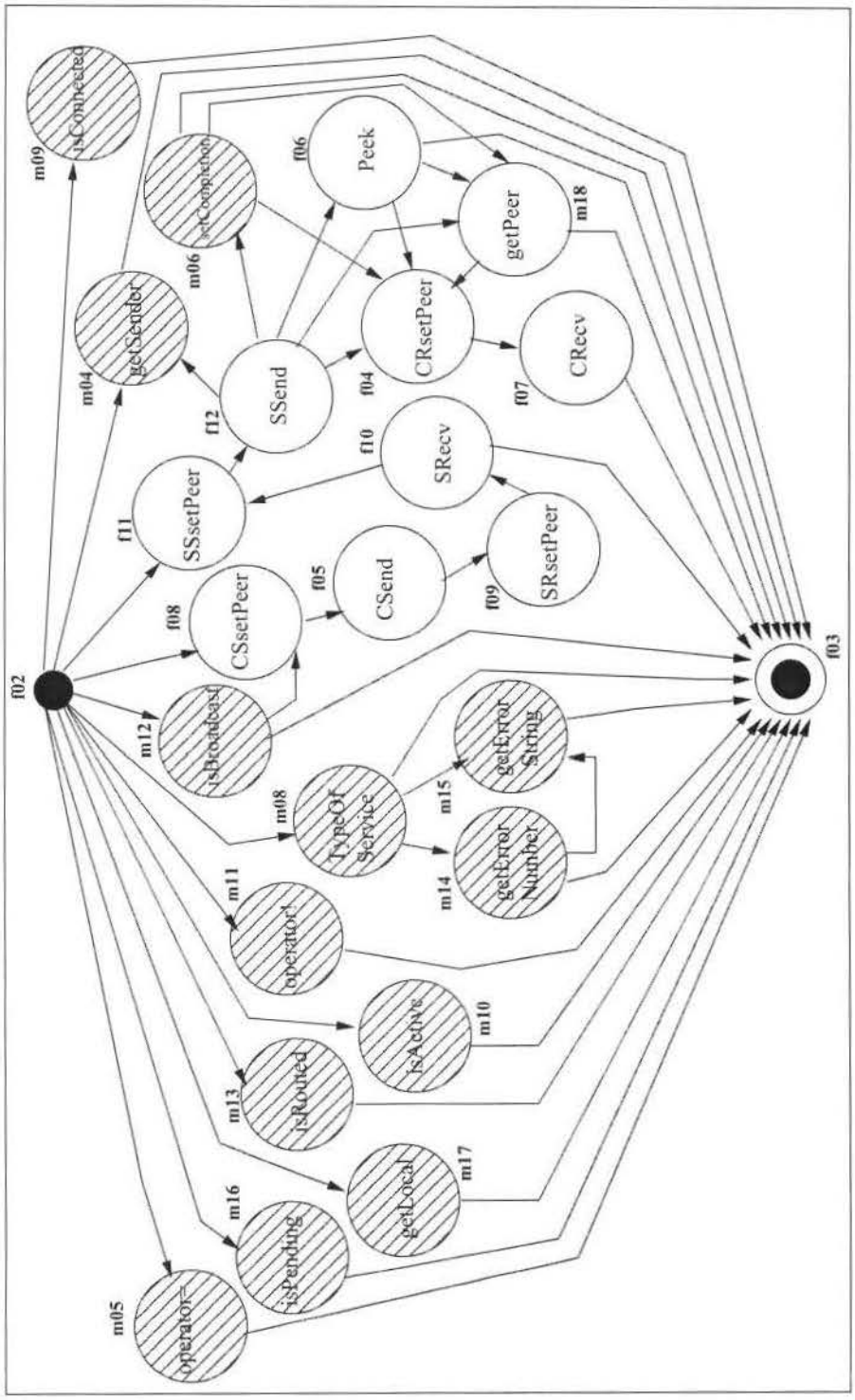

Figura A.2: Modelo de Fluxo de Transação da classe UDPSocket - versão base.

Apêndice B

Especificação de teste - classes Socket e UDPSocket

Neste apêndice são apresentadas as representações para os modelos de teste obtidas para as classes Socket e UDPSocket da primeira versão da biblioteca CommonC++, baseada no modelo de fluxo de transação para cada uma delas.

```
classe('Socket',s,_,[]).

parametros_template([]).

atributo('state',string,['sockstate_t'],_).
atributo('so',string,['int'],_).

metodo(m01,'Socket',_,construtor,1).
metodo(m02,'Socket',_,construtor,3).
metodo(m03,'~Socket',_,destrutor,0).
metodo(m04,'getSender',objeto,novo,1).
metodo(m05,'operator=',objeto,novo,1).
metodo(m06,'setCompletion','void',novo,1).
metodo(m07,'setKeepAlive','sockerror_t',novo,1).
metodo(m08,'setTypeOfService','sockerror_t',novo,1).
metodo(m09,'isConnected','bool',novo,0).
metodo(m10,'isActive','bool',novo,0).
metodo(m11,'operator!','bool',novo,0).
metodo(m12,'isBroadcast','bool',novo,0).
metodo(m13,'isRouted','bool',novo,0).
metodo(m14,'getErrorNumber','sockerror_t',novo,0).
metodo(m15,'getErrorString',string,novo,0).
metodo(m16,'isPending','bool',novo,2).
metodo(m17,'getLocal',objeto,novo,1).
metodo(m18,'getPeer',objeto,novo,1).

parametro('fd',m01,intervalo_int,9999999,1).
parametro('domain',m02,intervalo_int,9999999,1).
parametro('type',m02,intervalo_int,9999999,1).
parametro('protocol',m02,conjunto,[0,1,2],_).
parametro('port',m04,ponteiro,'short *',_).
parametro('from',m05,objeto,'Socket &',_).
parametro('completion',m06,string,['sockcomplete_t'],_).
parametro('enable',m07,string,['bool'],_).
parametro('service',m08,string,['socktos_t'],_).
parametro('pend',m16,string,['sockpend_t'],_).
parametro('timeout',m16,string,['timeout_t'],_).
parametro('port',m17,ponteiro,'short *',_).
parametro('port',m18,ponteiro,'short *',_).

no(m02,sim,3,[m02]).
no(m03,nao,0,[m03]).
no(m04,nao,1,[m04]).
no(m05,nao,1,[m05]).
no(m06,nao,2,[m06]).
no(m07,nao,3,[m07]).
no(m08,nao,3,[m08]).
no(m09,nao,2,[m09]).
no(m10,nao,1,[m10]).
no(m11,nao,1,[m11]).
no(m12,nao,1,[m12]).
no(m13,nao,1,[m13]).
no(m14,nao,2,[m14]).
no(m15,nao,1,[m15]).
no(m16,nao,1,[m16]).
no(m17,nao,1,[m17]).
no(m18,nao,1,[m18]).

arco(m02,m05).
arco(m02,m16).
arco(m02,m17).
arco(m02,m13).
arco(m02,m06).
arco(m02,m12).
arco(m02,m09).
arco(m02,m18).
arco(m02,m04).
arco(m02,m10).
arco(m02,m11).
arco(m02,m07).
arco(m02,m08).
arco(m06,m12).
arco(m06,m04).
arco(m09,m18).
arco(m07,m14).
arco(m15,m14).
arco(m07,m15).
arco(m08,m14).
arco(m08,m15).
arco(m05,m03).
arco(m16,m03).
arco(m17,m03).
arco(m13,m03).
arco(m06,m03).
arco(m12,m03).
arco(m09,m03).
arco(m18,m03).
arco(m04,m03).
arco(m10,m03).
arco(m11,m03).
arco(m07,m03).
arco(m14,m03).
arco(m15,m03).
arco(m08,m03).
```

Figura B.1: Descrição do modelo de teste para a classe Socket - versão base.

```
classe('UDPSocket',n,'Socket',[]).

parametros_template([]).

atributo('peer',string,['struct sockaddr_in'],_).

metodo(f01,'UDPSocket',_,construtor,0).
metodo(f02,'UDPSocKet',_,construtor,2).
metodo(f03,'~UDPSocket',_,destrutor,0).
metodo(f04,'CRsetPeer','void',novo,2).
metodo(f05,'CSend','int',novo,2).
metodo(f06,'Peek','int',novo,2).
metodo(f07,'CRecv','int',novo,2).
metodo(f08,'CSsetPeer','void',novo,2).
metodo(f09,'SRsetPeer','void',novo,2).
metodo(f10,'SRecv','int',novo,2).
metodo(f11,'SSsetPeer','void',novo,2).
metodo(f12,'SSend','int',novo,2).
metodo(m18,'getPeer',objeto,redefinido,1).
metodo(m04,'getSender',objeto,recursivo,1).
metodo(m05,'operator=',objeto,recursivo,1).
metodo(m06,'setCompletion','void',recursivo,1).
metodo(m08,'setTypeOfService','bool',recursivo,0).
metodo(m09,'isConnected','bool',recursivo,0).
metodo(m10,'isActive','bool',recursivo,0).
metodo(m11,'operator!','bool',recursivo,0).
metodo(m12,'isBroadcast','bool',recursivo,0).
metodo(m13,'isRouted','bool',recursivo,0).
metodo(m14,'getErrorNumber','sockerror_t',recursivo,0).
metodo(m15,'getErrorString',string,recursivo,0).
metodo(m16,'isPending','bool',recursivo,1).
metodo(m17,'getLocal',objeto,recursivo,1).

parametro('bind',f02,objeto,'InetAddress &',_).
parametro('port',f02,string,['short'],_).
parametro('host',f04,objeto,'InetHostAddress &',_).
parametro('port',f04,string,['short'],_).
parametro('buf',f05,ponteiro,'void *',_).
parametro('len',f05,string,['size_t'],_).
parametro('buf',f06,ponteiro,'void *',_).
parametro('len',f06,string,['size_t'],_).
parametro('buf',f07,ponteiro,'void *',_).
parametro('len',f07,string,['size_t'],_).
parametro('host',f08,objeto,'InetHostAddress &',_).
parametro('port',f08,string,['short'],_).
parametro('host',f09,objeto,'InetHostAddress &',_).
parametro('port',f09,string,['short'],_).
parametro('buf',f10,ponteiro,'void *',_).
parametro('len',f10,string,['size_t'],_).
parametro('host',f11,objeto,'InetHostAddress &',_).
parametro('port',f11,string,['short'],_).
parametro('buf',f12,ponteiro,'void *',_).
parametro('len',f12,string,['size_t'],_).
parametro('port',m18,ponteiro,'short *',_).
parametro('port',m04,ponteiro,'short *',_).
parametro('from',m05,objeto,'Socket &',_).
parametro('completion',m06,string,['sockcomplete_t'],_).
parametro('service',m08,string,['socktos_t'],_).
parametro('pend',m16,string,['sockpend_t'],_).
parametro('timeout',m16,string,['timeout_t'],_).
parametro('port',m17,ponteiro,'short *',_).

no(n002,sim,2,[f02]).
no(n003,nao,0,[f03]).
no(n004,nao,1,[f04]).
no(n005,nao,2,[f05]).
no(n006,nao,1,[f06]).
no(n007,nao,1,[f07]).
no(n008,nao,1,[f08]).
no(n009,nao,1,[f09]).
no(n010,nao,2,[f10]).
no(n011,nao,1,[f11]).
no(n012,nao,3,[f12]).
no(n18,nao,2,[m18]).
no(n04,nao,1,[m04]).
no(n05,nao,1,[m05]).
no(n06,nao,2,[m06]).
no(n08,nao,3,[m08]).
no(n09,nao,2,[m09]).
no(n10,nao,1,[m10]).
no(n11,nao,1,[m11]).
no(n12,nao,3,[m12]).
no(n13,nao,1,[m13]).
no(n14,nao,2,[m14]).
no(n15,nao,1,[m15]).
no(n16,nao,1,[m16]).
no(n17,nao,1,[m17]).

arco(n002,n008).
arco(n002,n011).
arco(n002,n12).
arco(n002,n08).
arco(n002,n09).
arco(n002,n04).
arco(n002,n05).
arco(n002,n16).
arco(n002,n17).
arco(n002,n13).
arco(n002,n10).
arco(n002,n11).
arco(n008,n005).
arco(n011,n012).
arco(n12,n008).
arco(n06,n18).
arco(n06,n004).
arco(n012,n04).
arco(n012,n06).
arco(n08,n14).
arco(n08,n15).
arco(n14,n15).
arco(n005,n009).
arco(n012,n004).
arco(n012,n18).
arco(n012,n006).
arco(n009,n010).
arco(n004,n007).
arco(n18,n004).
arco(n006,n004).
arco(n006,n18).
arco(n010,n011).
arco(n007,n003).
arco(n010,n003).
arco(n18,n003).
arco(n006,n003).
arco(n08,n003).
arco(n14,n003).
arco(n15,n003).
arco(n05,n003).
arco(n16,n003).
arco(n17,n003).
arco(n13,n003).
arco(n10,n003).
arco(n11,n003).
arco(n09,n003).
arco(n06,n003).
arco(n04,n003).
arco(n12,n003).
```

Figura B.2: Descrição do modelo de teste para a classe UDPSocket - versão base.

Apêndice C

Objetos de Testes para as classes

Neste apêndice são apresentadas as seqüências de testes geradas pela ConCAT para as classes Socket e UDPSocket da versão base da biblioteca CommonC++.

```
objeto_tst(1,['m02','m04','m03'],'Caso_teste1_0','Socket_ct0.cc').
objeto_tst(2,['m02','m05','m03'],'Caso_teste2_0','Socket_ct0.cc').
objeto_tst(3,['m02','m06','m03'],'Caso_teste3_0','Socket_ct0.cc').
objeto_tst(4,['m02','m06','m18','m03'],'Caso_teste4_0','Socket_ct0.cc').
objeto_tst(5,['m02','m07','m03'],'Caso_teste5_0','Socket_ct0.cc').
objeto_tst(6,['m02','m07','m14','m03'],'Caso_teste6_0','Socket_ct0.cc').
objeto_tst(7,['m02','m07','m14','m15','m03'],'Caso_teste7_0','Socket_ct0.cc').
objeto_tst(8,['m02','m07','m15','m03'],'Caso_teste8_0','Socket_ct0.cc').
objeto_tst(9,['m02','m08','m03'],'Caso_teste9_0','Socket_ct0.cc').
objeto_tst(10,['m02','m08','m14','m03'],'Caso_teste10_0','Socket_ct0.cc').
objeto_tst(11,['m02','m08','m14','m15','m03'],'Caso_teste11_0','Socket_ct0.cc').
objeto_tst(12,['m02','m08','m15','m03'],'Caso_teste12_0','Socket_ct0.cc').
objeto_tst(13,['m02','m09','m03'],'Caso_teste13_0','Socket_ct0.cc').
objeto_tst(14,['m02','m10','m03'],'Caso_teste14_0','Socket_ct0.cc').
objeto_tst(15,['m02','m11','m03'],'Caso_teste15_0','Socket_ct0.cc').
objeto_tst(16,['m02','m12','m03'],'Caso_teste16_0','Socket_ct0.cc').
objeto_tst(17,['m02','m13','m03'],'Caso_teste17_0','Socket_ct0.cc').
objeto_tst(18,['m02','m16','m03'],'Caso_teste18_0','Socket_ct0.cc').
objeto_tst(19,['m02','m17','m03'],'Caso_teste19_0','Socket_ct0.cc').
objeto_tst(20,['m02','m18','m03'],'Caso_teste20_0','Socket_ct0.cc').
```

Figura C.1: Objetos de teste para a classe Socket - versão base.

```
objeto_tst(1,['f02','f08','f05','f09','f10','f03'],'Caso_teste1_1','UDPSocket_ct0.cc').
objeto_tst(2,['f02','f08','f05','f09','f10','f11','f12','f04','f07','f03'],'Caso_teste2_1','UDPSocket_ct0.cc').
objeto_tst(3,['f02','f08','f05','f09','f10','f11','f12','f06','f03'],'Caso_teste3_1','UDPSocket_ct0.cc').
objeto_tst(4,['f02','f08','f05','f09','f10','f11','f12','f06','f04','f07','f03'],'Caso_teste4_1','UDPSocket_ct0.cc').
objeto_tst(5,['f02','f08','f05','f09','f10','f11','f12','f06','m18','f03'],'Caso_teste5_1','UDPSocket_ct0.cc').
objeto_tst(6,['f02','f08','f05','f09','f10','f11','f12','f06','m18','f04','f07','f03'],'Caso_teste6_1','UDPSocket_ct0.cc').
objeto_tst(7,['f02','f08','f05','f09','f10','f11','f12','m04','f03'],'Caso_teste7_1','UDPSocket_ct0.cc').
objeto_tst(8,['f02','f08','f05','f09','f10','f11','f12','m06','f03'],'Caso_teste8_1','UDPSocket_ct0.cc').
objeto_tst(9,['f02','f08','f05','f09','f10','f11','f12','m06','f04','f07','f03'],'Caso_teste9_1','UDPSocket_ct0.cc').
objeto_tst(10,['f02','f08','f05','f09','f10','f11','f12','m06','m18','f03'],'Caso_teste10_1','UDPSocket_ct0.cc').
objeto_tst(11,['f02','f08','f05','f09','f10','f11','f12','m06','m18','f04','f07','f03'],'Caso_teste11_1','UDPSocket_ct0.cc').
objeto_tst(12,['f02','f08','f05','f09','f10','f11','f12','m18','f03'],'Caso_teste12_1','UDPSocket_ct0.cc').
objeto_tst(13,['f02','f08','f05','f09','f10','f11','f12','m18','f04','f07','f03'],'Caso_teste13_1','UDPSocket_ct0.cc').
objeto_tst(14,['f02','f11','f12','f04','f07','f03'],'Caso_teste14_1','UDPSocket_ct0.cc').
objeto_tst(15,['f02','f11','f12','f06','f03'],'Caso_teste15_1','UDPSocket_ct0.cc').
objeto_tst(16,['f02','f11','f12','f06','f04','f07','f03'],'Caso_teste16_1','UDPSocket_ct0.cc').
objeto_tst(17,['f02','f11','f12','f06','m18','f03'],'Caso_teste17_1','UDPSocket_ct0.cc').
objeto_tst(18,['f02','f11','f12','f06','m18','f04','f07','f03'],'Caso_teste18_1','UDPSocket_ct0.cc').
objeto_tst(19,['f02','f11','f12','m04','f03'],'Caso_teste19_1','UDPSocket_ct0.cc').
objeto_tst(20,['f02','f11','f12','m06','f03'],'Caso_teste20_1','UDPSocket_ct0.cc').
objeto_tst(21,['f02','f11','f12','m06','f04','f07','f03'],'Caso_teste21_1','UDPSocket_ct0.cc').
objeto_tst(22,['f02','f11','f12','m06','m18','f03'],'Caso_teste22_1','UDPSocket_ct0.cc').
objeto_tst(23,['f02','f11','f12','m06','m18','f04','f07','f03'],'Caso_teste23_1','UDPSocket_ct0.cc').
objeto_tst(24,['f02','f11','f12','m18','f03'],'Caso_teste24_1','UDPSocket_ct0.cc').
objeto_tst(25,['f02','f11','f12','m18','f04','f07','f03'],'Caso_teste25_1','UDPSocket_ct0.cc').
objeto_tst(26,['f02','m12','f08','f05','f09','f10','f03'],'Caso_teste26_1','UDPSocket_ct0.cc').
objeto_tst(27,['f02','m12','f08','f05','f09','f10','f11','f12','f04','f07','f03'],'Caso_teste27_1','UDPSocket_ct0.cc').
objeto_tst(28,['f02','m12','f08','f05','f09','f10','f11','f12','f06','f03'],'Caso_teste28_1','UDPSocket_ct0.cc').
objeto_tst(29,['f02','m12','f08','f05','f09','f10','f11','f12','f06','f04','f07','f03'],'Caso_teste29_1','UDPSocket_ct0.cc').
objeto_tst(30,['f02','m12','f08','f05','f09','f10','f11','f12','f06','m18','f03'],'Caso_teste30_1','UDPSocket_ct0.cc').
objeto_tst(31,['f02','m12','f08','f05','f09','f10','f11','f12','f06','m18','f04','f07','f03'],'Caso_teste31_1','UDPSocket_ct0.cc').
objeto_tst(32,['f02','m12','f08','f05','f09','f10','f11','f12','m04','f03'],'Caso_teste32_1','UDPSocket_ct0.cc').
objeto_tst(33,['f02','m12','f08','f05','f09','f10','f11','f12','m06','f03'],'Caso_teste33_1','UDPSocket_ct0.cc').
objeto_tst(34,['f02','m12','f08','f05','f09','f10','f11','f12','m06','f04','f07','f03'],'Caso_teste34_1','UDPSocket_ct0.cc').
objeto_tst(35,['f02','m12','f08','f05','f09','f10','f11','f12','m06','m18','f03'],'Caso_teste35_1','UDPSocket_ct0.cc').
objeto_tst(36,['f02','m12','f08','f05','f09','f10','f11','f12','m06','m18','f04','f07','f03'],'Caso_teste36_1','UDPSocket_ct0.cc').
objeto_tst(37,['f02','m12','f08','f05','f09','f10','f11','f12','m18','f03'],'Caso_teste37_1','UDPSocket_ct0.cc').
objeto_tst(38,['f02','m12','f08','f05','f09','f10','f11','f12','m18','f04','f07','f03'],'Caso_teste38_1','UDPSocket_ct0.cc').
```

Figura C.2: Objetos de teste para a classe UDPSocket - versão base.

Apêndice D

Driver Específico - versão base

A seguir são apresentados os drivers específicos gerados pela CONCAT e devidamente complementados para a classe *Socket* e sua classe derivada *UDPSocket* da primeira versão da biblioteca CommonC++. Devido a grande extensão do código que representa os casos de teste gerados apenas alguns exemplos serão mostrados para a classe UDPSocket.

D.1 Driver para a Classe Socket

O driver específico gerado para a classe Socket sofreu alterações, conforme código abaixo, já que essa classe é Abstrata, e não é possível instanciar um objeto dela para que os testes fossem executados. Após as alterações, esse driver passará a ser chamado no driver da classe derivada UDPSocket para que os testes sejam exercitados.

```
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>

#define TESTE 1
#include "cc++/socket.h"
#define ARQUIVO_RESULTADOS "Result100.txt"
#include "Definicoes.h"
#include "casos_Socket100.h"

void main() {
   InetAddress localaddr( L_ADDR );

   UDPSocket* objs1 = new UDPSocket(localaddr, PORTA_L);
   Caso_teste1_0(objs1);
   UDPSocket* objs2 = new UDPSocket(localaddr, PORTA_L);
   Caso_teste2_0(objs2);
   UDPSocket* objs3 = new UDPSocket(localaddr, PORTA_L);
   Caso_teste3_0(objs3);
   UDPSocket* objs4 = new UDPSocket(localaddr, PORTA_L);
   Caso_teste4_0(objs4);
   UDPSocket* objs5 = new UDPSocket(localaddr, PORTA_L);
```

```
    Caso_teste5_0(objs5);
    UDPSocket* objs6 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste6_0(objs6);
    UDPSocket* objs7 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste7_0(objs7);
    UDPSocket* objs8 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste8_0(objs8);
    UDPSocket* objs9 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste9_0(objs9);
    UDPSocket* objs10 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste10_0(objs10);
    UDPSocket* objs11 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste11_0(objs11);
    UDPSocket* objs12 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste12_0(objs12);
    UDPSocket* objs13 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste13_0(objs13);
    UDPSocket* objs14 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste14_0(objs14);
    UDPSocket* objs15 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste15_0(objs15);
    UDPSocket* objs16= new UDPSocket(localaddr, PORTA_L);
    Caso_teste16_0(objs16);
    UDPSocket* objs17 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste17_0(objs17);
    UDPSocket* objs18 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste18_0(objs18);
    UDPSocket* objs19 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste19_0(objs19);
    UDPSocket* objs20 = new UDPSocket(localaddr, PORTA_L);
    Caso_teste20_0(objs20);
}
```

D.2 Driver para a Classe UDPSocket

O driver específico gerado para a classe UDPSocket considera a reutilização daqueles testes gerados para exercitar as transações na classe Socket que não tiveram alterações, conforme mostrado em D.1.

```
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>

#define TESTE 1
#include "cc++/socket.h"
#define ARQUIVO_RESULTADOS "Resu t100.txt"
#include "Definicoes.h"
#include "casos_Socket100.h"
#include "casos_UDPSocket100.h"

void main() {
    InetAddress localaddr( L_ADDR );

    UDPSocket* obj1 = new UDPSocket(localaddr, PORTA_L);
```

```
Caso_teste1_1(obj1);
UDPSocket* obj2 = new UDPSocket(localaddr, PORTAL);
Caso_teste2_1(obj2);
UDPSocket* obj3 = new UDPSocket(localaddr, PORTAL);
Caso_teste3_1(obj3);
UDPSocket* obj4 = new UDPSocket(localaddr, PORTAL);
Caso_teste4_1(obj4);
UDPSocket* obj5 = new UDPSocket(localaddr, PORTAL);
Caso_teste5_1(obj5);
UDPSocket* obj6 = new UDPSocket(localaddr, PORTAL);
Caso_teste6_1(obj6);
UDPSocket* obj7 = new UDPSocket(localaddr, PORTAL);
Caso_teste7_1(obj7);
UDPSocket* obj8 = new UDPSocket(localaddr, PORTAL);
Caso_teste8_1(obj8);
UDPSocket* obj9 = new UDPSocket(localaddr, PORTAL);
Caso_teste9_1(obj9);
UDPSocket* obj10 = new UDPSocket(localaddr, PORTAL);
Caso_teste10_1(obj10);
UDPSocket* obj11 = new UDPSocket(localaddr, PORTAL);
Caso_teste11_1(obj11);
UDPSocket* obj12 = new UDPSocket(localaddr, PORTAL);
Caso_teste12_1(obj12);
UDPSocket* obj13 = new UDPSocket(localaddr, PORTAL);
Caso_teste13_1(obj13);
UDPSocket* obj14 = new UDPSocket(localaddr, PORTAL);
Caso_teste14_1(obj14);
UDPSocket* obj15 = new UDPSocket(localaddr, PORTAL);
Caso_teste15_1(obj15);
UDPSocket* obj16 = new UDPSocket(localaddr, PORTAL);
Caso_teste16_1(obj16);
UDPSocket* obj17 = new UDPSocket(localaddr, PORTAL);
Caso_teste17_1(obj17);
UDPSocket* obj18 = new UDPSocket(localaddr, PORTAL);
Caso_teste18_1(obj18);
UDPSocket* obj19 = new UDPSocket(localaddr, PORTAL);
Caso_teste19_1(obj19);
UDPSocket* obj20 = new UDPSocket(localaddr, PORTAL);
Caso_teste20_1(obj20);
UDPSocket* obj21 = new UDPSocket(localaddr, PORTAL);
Caso_teste21_1(obj21);
UDPSocket* obj22 = new UDPSocket(localaddr, PORTAL);
Caso_teste22_1(obj22);
UDPSocket* obj23 = new UDPSocket(localaddr, PORTAL);
Caso_teste23_1(obj23);
UDPSocket* obj24 = new UDPSocket(localaddr, PORTAL);
Caso_teste24_1(obj24);
UDPSocket* obj25 = new UDPSocket(localaddr, PORTAL);
Caso_teste25_1(obj25);
UDPSocket* obj26 = new UDPSocket(localaddr, PORTAL);
Caso_teste26_1(obj26);
UDPSocket* obj27 = new UDPSocket(localaddr, PORTAL);
Caso_teste27_1(obj27);
UDPSocket* obj28 = new UDPSocket(localaddr, PORTAL);
Caso_teste28_1(obj28);
UDPSocket* obj29 = new UDPSocket(localaddr, PORTAL);
Caso_teste29_1(obj29);
UDPSocket* obj30 = new UDPSocket(localaddr, PORTAL);
```

```
    Caso_teste30_1(obj30);
    UDPSocket* obj31 = new UDPSocket(localaddr, PORTAL);
    Caso_teste31_1(obj31);
    UDPSocket* obj32 = new UDPSocket(localaddr, PORTAL);
    Caso_teste32_1(obj32);
    UDPSocket* obj33 = new UDPSocket(localaddr, PORTAL);
    Caso_teste33_1(obj33);
    UDPSocket* obj34 = new UDPSocket(localaddr, PORTAL);
    Caso_teste34_1(obj34);
    UDPSocket* obj35 = new UDPSocket(localaddr, PORTAL);
    Caso_teste35_1(obj35);
    UDPSocket* obj36 = new UDPSocket(localaddr, PORTAL);
    Caso_teste36_1(obj36);
    UDPSocket* obj37 = new UDPSocket(localaddr, PORTAL);
    Caso_teste37_1(obj37);
    UDPSocket* obj38 = new UDPSocket(localaddr, PORTAL);
    Caso_teste38_1(obj38);

//CASOS DE TESTE HERDADOS DA SUPERCLASSE SOCKET:
    UDPSocket* obj40 = new UDPSocket(localaddr, PORTAL);
    Caso_teste2_0(obj40);
    UDPSocket* obj41 = new UDPSocket(localaddr, PORTAL);
    Caso_teste3_0(obj41);
    UDPSocket* obj43 = new UDPSocket(localaddr, PORTAL);
    Caso_teste9_0(obj43);
    UDPSocket* obj44 = new UDPSocket(localaddr, PORTAL);
    Caso_teste10_0(obj44);
    UDPSocket* obj45 = new UDPSocket(localaddr, PORTAL);
    Caso_teste11_0(obj45);
    UDPSocket* obj46 = new UDPSocket(localaddr, PORTAL);
    Caso_teste12_0(obj46);
    UDPSocket* obj47 = new UDPSocket(localaddr, PORTAL);
    Caso_teste13_0(obj47);
    UDPSocket* obj48 = new UDPSocket(localaddr, PORTAL);
    Caso_teste14_0(obj48);
    UDPSocket* obj49 = new UDPSocket(localaddr, PORTAL);
    Caso_teste15_0(obj49);
    UDPSocket* obj50 = new UDPSocket(localaddr, PORTAL);
    Caso_teste16_0(obj50);
    UDPSocket* obj51= new UDPSocket(localaddr, PORTAL);
    Caso_teste17_0(obj51);
    UDPSocket* obj52 = new UDPSocket(localaddr, PORTAL);
    Caso_teste18_0(obj52);
    UDPSocket* obj53 = new UDPSocket(localaddr, PORTAL);
    Caso_teste19_0(obj53);
}
```

D.3 Casos de teste para a Classe UDPSocket

Exemplo da implementação de alguns dos trinta e oito casos de teste para a classe UDP-Socket que são chamados no driver específico dessa classe.

```
template <class Tipo>
void Caso_teste1_1(Tipo* cst) {
   char* met= new char[100];
```

```
char bufferRecebe[TAMMENSAGEM+1];
char *bufferEnvia=MENSAGEM;
InetHostAddress s(R_HOST);
InetHostAddress c(L_HOST);

ofstream arq(ARQUIVO_RESULTADOS, ios::app);
if (! arq)
   cout << "Erro abrindo arquivo! \n";
else
   cout << "Gravando informacoes caso de teste 1_1! \n";
try {

 cst->Testa_Invariante();
 //met="CSsetPeer(Parametro deve ser um objeto do tipo InetHostAddress &,\"short\")";
 strcpy(met,"CSsetPeer(servidor(");
 strcat(met,R_HOST);   strcat(met,"), ");
 strcat(met,itoa(PORTA_R));   strcat(met,")");
 cst->setPeer(s,(short int) PORTA_R);
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 //met= "CSend(void *bufferEnvia, size_t TAMMENSAGEM)";
 strcpy(met,"CSend('");
 strcat(met,bufferEnvia); strcat(met,"', "); strcat(met,itoa(TAMMENSAGEM));
 strcat(met,")");
 cst->Send(bufferEnvia, TAMMENSAGEM);
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 //met= "SRsetPeer(InetHostAddress s, short PORTA_L)";
 strcpy(met,"SRsetPeer(servidor(");
 strcat(met,R_HOST); strcat(met,"), "); strcat(met,itoa(PORTA_L));
 strcat(met,")");
 cst->setPeer(s, (short int) PORTA_L);
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 //met= "SRecv(void *bufferRecebe, size_t TAMMENSAGEM)";
 cst->Recv(bufferRecebe, TAMMENSAGEM);
 strcpy(met,"SRecv('");
 strcat(met,bufferRecebe); strcat(met,"', ");
 strcat(met,itoa(TAMMENSAGEM));   strcat(met,")");
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 arq << "Caso_teste1_1 OK!\n";
 arq.flush();   arq << "\n";   arq.close();
 cst->Relator(ARQUIVO_RESULTADOS);
 delete cst;
}

catch(char* c) {
   arq << "Caso_teste1_1 \n";
   arq.flush();   cout << "...";   arq << c << "\n";
```

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

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


template <class Tipo>
void Caso_teste2_1(Tipo* cst) {
    char* met= new char[100];
    char bufferRecebe[TAM_MENSAGEM+1];
    char *bufferEnvia=MENSAGEM;
    InetHostAddress s(R_HOST);
    InetHostAddress c(L_HOST);

    ofstream arq(ARQUIVO_RESULTADOS,ios::app);
    if (! arq)
        cout << "Erro abrindo arquivo! \n";
    else
        cout << "Gravando informacoes caso de teste 2_1! \n";
    try {
     cst->Testa_Invariante();
     //met="CSsetPeer(Parametro deve ser um objeto do tipoInetHostAddress &,\"short\")";
     strcpy(met,"CSsetPeer(servidor(");
     strcat(met,R_HOST); strcat(met,"), "); strcat(met,itoa(PORTA_R));
     strcat(met,")");
     cst->setPeer(s, (short int) PORTA_R);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met= "CSend(void *bufferEnvia, size_t TAM_MENSAGEM)";
     strcpy(met,"CSend('");
     strcat(met,bufferEnvia); strcat(met,"', ");
     strcat(met,itoa(TAM_MENSAGEM));
     strcat(met,")");
     cst->Send(bufferEnvia, TAM_MENSAGEM);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met= "SRsetPeer(InetHostAddress c, short PORTA_L)";
     strcpy(met,"SRsetPeer(cliente(");
     strcat(met,L_HOST); strcat(met,"), "); strcat(met,itoa(PORTA_L));
     strcat(met,")");
     cst->setPeer(c, (short int) PORTA_L);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met= "SRecv(void *bufferRecebe, size_t TAM_MENSAGEM)";
     cst->Recv(bufferRecebe, TAM_MENSAGEM);
```

```
strcpy(met,"SRecv('");
strcat(met,bufferRecebe);  strcat(met,"', ");
strcat(met,itoa(TAMMENSAGEM));  strcat(met,")");
arq << "Metodo chamado: " << met << "\n";
arq.flush();

cst->Testa_Invariante();
//met= "SSsetPeer(InetHostAddress s, short PORTA_R)";
strcpy(met,"SSsetPeer(servidor(");
strcat(met,R_HOST); strcat(met,"), "); strcat(met,itoa(PORTA_R));
strcat(met,")");
cst->setPeer(s, (short int) PORTA_R);
arq << "Metodo chamado: " << met << "\n";
arq.flush();

cst->Testa_Invariante();
//met= "SSend(void *bufferEnvia, size_t TAMMENSAGEM)";
strcpy(met,"SSend('");
strcat(met,bufferEnvia);  strcat(met,"', ");
strcat(met,itoa(TAMMENSAGEM));  strcat(met,")");
cst->Send(bufferEnvia, TAMMENSAGEM);
arq << "Metodo chamado: " << met << "\n";
arq.flush();

cst->Testa_Invariante();
//met= "CRsetPeer(InetHostAddress c, short PORTA_L)";
strcpy(met,"CRsetPeer(cliente(");
strcat(met,L_HOST); strcat(met,"), "); strcat(met,itoa(PORTA_L));
strcat(met,")");
cst->setPeer(c, PORTA_L);
arq << "Metodo chamado: " << met << "\n";
arq.flush();

cst->Testa_Invariante();
//met= "CRecv(void *bufferRecebe, size_t TAMMENSAGEM)";
cst->Recv(bufferRecebe, TAMMENSAGEM);
strcpy(met,"CRecv('");
strcat(met,bufferRecebe);  strcat(met,"', ");
strcat(met,itoa(TAMMENSAGEM));  strcat(met,")");
arq << "Metodo chamado: " << met << "\n";
arq.flush();

cst->Testa_Invariante();
arq << "Caso_teste2_1 OK!\n";
arq.flush();  arq << "\n";  arq.close();
cst->Relator(ARQUIVO_RESULTADOS);
delete cst;
}

catch(char* c) {
   arq << "Caso_teste2_1 \n";
   arq.flush();  cout << "...";  arq << c << "\n";
   arq << "Metodo chamado: " << met << "\n";
   arq.flush();
   cst->Relator(ARQUIVO_RESULTADOS);
   arq << "\n";
   arq.close();
}
```

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

template <class Tipo>
void Caso_teste8_1(Tipo* cst) {
    char* met= new char[100];
    char bufferRecebe[TAM_MENSAGEM+1];
    char *bufferEnvia=MENSAGEM;
    InetHostAddress s(R_HOST);
    InetHostAddress c(L_HOST);
    sockcomplete_t comp;
    comp=SOCKET_COMPLETION_DELAYED;

    ofstream arq(ARQUIVO_RESULTADOS,ios::app);
    if (! arq)
        cout << "Erro abrindo arquivo! \n";
    else
        cout << "Gravando informacoes caso de teste 8_1! \n";
    try {
     cst->Testa_Invariante();
     //met = "CSsetPeer(Parametro deve ser um objeto do tipo InetHostAddress &,\"short\")
      ";
     strcpy(met,"CSsetPeer(servidor(");
     strcat(met,R_HOST);    strcat(met,"), ");
     strcat(met,itoa(PORTA_R));    strcat(met,")");
     cst->setPeer(s,(short int) PORTA_R);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met= "CSend(\"Mensagem padrao ...\", 20)";
     strcpy(met,"CSend('");
     strcat(met,bufferEnvia);    strcat(met,"', ");
     strcat(met,itoa(TAM_MENSAGEM));    strcat(met,")");
     cst->Send(bufferEnvia, TAM_MENSAGEM);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met= "SRsetPeer(InetHostAddress s, short PORTA_L)";
     strcpy(met,"SRsetPeer(servidor(");
     strcat(met,R_HOST);    strcat(met,"), ");
     strcat(met,itoa(PORTA_L));    strcat(met,")");
     cst->setPeer(s, (short int) PORTA_L);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met= "SRecv(void *bufferRecebe, size_t TAM_MENSAGEM)";
     cst->Recv(bufferRecebe, TAM_MENSAGEM);
     strcpy(met,"SRecv('");
     strcat(met,bufferRecebe);    strcat(met,"', ");
     strcat(met,itoa(TAM_MENSAGEM));    strcat(met,")");
     arq << "Metodo chamado: " << met << "\n";
```

```
    arq.flush();

    cst->Testa_Invariante();
    //met= "SSsetPeer(InetHostAddress s, short PORTA_R)";
    strcpy(met,"SSsetPeer(servidor(");
    strcat(met,R_HOST);   strcat(met,"), ");
    strcat(met,itoa(PORTA_R));   strcat(met,")");
    cst->setPeer(s, (short int) PORTA_R);
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();

    cst->Testa_Invariante();
    //met= "SSend(void *bufferEnvia, size_t TAM_MENSAGEM)";
    strcpy(met,"SSend('");
    strcat(met,bufferEnvia);   strcat(met,"', ");
    strcat(met,itoa(TAM_MENSAGEM));   strcat(met,")");
    cst->Send(bufferEnvia, TAM_MENSAGEM);
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();

    cst->Testa_Invariante();
    //met = "setCompletion(\"sockcomplete_t\")";
    strcpy(met, "setCompletion(sockcomplete_t comp)");
    cst->setCompletion(comp);
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();

    cst->Testa_Invariante();
    arq << "Caso_teste8_1 OK!\n";
    arq.flush();   arq << "\n";   arq.close();
    cst->Relator(ARQUIVO_RESULTADOS);
    delete cst;
  }

  catch(char* c) {
     arq << "Caso_teste8_1 \n";
     arq.flush();   cout << "...";   arq << c << "\n";
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();
     cst->Relator(ARQUIVO_RESULTADOS);
     arq << "\n";
     arq.close();
  }

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


template <class Tipo>
void Caso_teste14_1(Tipo* cst) {
  char* met= new char[100];
  char bufferRecebe[TAM_MENSAGEM+1];
  char *bufferEnvia=MENSAGEM;
  InetHostAddress s(R_HOST);
  InetHostAddress c(L_HOST);
```

```
ofstream arq(ARQUIVO_RESULTADOS, ios::app);
if (! arq)
   cout << "Erro abrindo arquivo! \n";
else
   cout << "Gravando informacoes caso de teste 14_1! \n";
try {
 cst->Testa_Invariante();
 //met= "SSsetPeer(InetHostAddress s, short PORTA_R)";
 strcpy(met,"SSsetPeer(servidor(");
 strcat(met,R_HOST);     strcat(met,"), ");
 strcat(met,itoa(PORTA_R));     strcat(met,")");
 cst->setPeer(s, (short int) PORTA_R);
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 //met= "SSend(void *bufferEnvia, size_t TAM_MENSAGEM)";
 strcpy(met,"SSend('");
 strcat(met,bufferEnvia);     strcat(met,"', ");
 strcat(met,itoa(TAM_MENSAGEM));     strcat(met,")");
 cst->Send(bufferEnvia, TAM_MENSAGEM);
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 //met= "CRsetPeer(InetHostAddress c, short PORTA_L)";
 strcpy(met,"CRsetPeer(cliente(");
 strcat(met,L_HOST);     strcat(met,"), ");
 strcat(met,itoa(PORTA_L));     strcat(met,")");
 cst->setPeer(c, PORTA_L);
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 //met= "CRecv(void *bufferRecebe, size_t TAM_MENSAGEM)";
 cst->Recv(bufferRecebe, TAM_MENSAGEM);
 strcpy(met,"CRecv('");
 strcat(met,bufferRecebe);     strcat(met,"', ");
 strcat(met,itoa(TAM_MENSAGEM));     strcat(met,")");
 arq << "Metodo chamado: " << met << "\n";
 arq.flush();

 cst->Testa_Invariante();
 arq << "Caso_teste14_1 OK!\n";
 arq.flush();     arq << "\n";     arq.close();
 cst->Relator(ARQUIVO_RESULTADOS);
 delete cst;
}

catch(char* c) {
   arq << "Caso_teste14_1 \n";
   arq.flush();     cout << "...";     arq << c << "\n";
   arq << "Metodo chamado: " << met << "\n";
   arq.flush();
   cst->Relator(ARQUIVO_RESULTADOS);
   arq << "\n";
   arq.close();
}
```

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

template <class Tipo>
void Caso_teste26_1(Tipo* cst) {
    char* met= new char[100];
    char bufferRecebe[TAM_MENSAGEM+1];
    char *bufferEnvia=MENSAGEM;
    InetHostAddress s(R_HOST);
    InetHostAddress c(L_HOST);
    bool retorno;

    ofstream arq(ARQUIVO_RESULTADOS, ios::app);
    if (! arq)
        cout << "Erro abrindo arquivo! \n";
    else
        cout << "Gravando informacoes caso de teste 26_1! \n";
    try {

     cst->Testa_Invariante();
     strcpy(met,"isBroadcast()");
     retorno = cst->isBroadcast();
     arq << "Metodo chamado: " << met << "    retorno= " << retorno << "\n";
     arq.flush();

     cst->Testa_Invariante();
     strcpy(met,"CSsetPeer(servidor(");
     strcat(met,R_HOST);     strcat(met,"), ");
     strcat(met,itoa(PORTA_R));    strcat(met,")");
     cst->setPeer(s,(short int) PORTA_R);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met = "CSend(Parametro deve ser um ponteiro para void,\"size_t\")";
     strcpy(met,"CSend('");
     strcat(met,bufferEnvia);    strcat(met,"', ");
     strcat(met,itoa(TAM_MENSAGEM));    strcat(met,")");
     cst->Send(bufferEnvia, TAM_MENSAGEM);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met = "SRsetPeer(Parametro deve ser um objeto do tipo InetHostAddress &,\"short\")
     ";
     strcpy(met,"SRsetPeer(servidor(");
     strcat(met,R_HOST);    strcat(met,"), ");
     strcat(met,itoa(PORTA_L));    strcat(met,")");
     cst->setPeer(s, (short int) PORTA_L);
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();

     cst->Testa_Invariante();
     //met = "SRecv(Parametro deve ser um ponteiro para void,\"size_t\")";
```

```
    cst->Recv(bufferRecebe, TAM_MENSAGEM);
    strcpy(met,"SRecv('");
    strcat(met,bufferRecebe);    strcat(met,"', ");
    strcat(met,itoa(TAM_MENSAGEM));    strcat(met,")");
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();

    cst->Testa_Invariante();
    arq << "Caso_teste26_1 OK!\n";
    arq.flush();    arq << "\n";    arq.close();
    cst->Relator(ARQUIVO_RESULTADOS);
    delete cst;
  }

  catch(char* c) {
     arq << "Caso_teste26_1 \n";
     arq.flush();    cout << "...";    arq << c << "\n";
     arq << "Metodo chamado: " << met << "\n";
     arq.flush();
     cst->Relator(ARQUIVO_RESULTADOS);
     arq << "\n";
     arq.close();
  }

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

Apêndice E

Código para encontrar relação entre arcos e casos de teste

Neste apêndice é apresentado o código do procedimento para conversão das informações da descrição do modelo de teste para a classe (arquivo com extensão '.mt') e os objetos de teste (arquivo com extensão '.ot0') gerados pela ConCAT para obtenção da relação de cada arco com a transação que passa por ele. Como resultado são gerados três arquivos:

- arquivo identificado por "ident_arco.txt" onde são mostradas as identificações de cada arco e os nós que formam cada um deles;
- arquivo "ident_arco_metodo.txt" que contém a mesma identificação para cada arco mostrada no arquivo anterior acrescida do nome de cada método da classe que compõe o nó, ao invés da identificação do nó;
- arquivo "relat_arco_caso.txt" onde cada linha contém a identificação de um arco e a identificação de um caso de teste que o exercita. Pode ocorrer de um arco ser exercitado por mais de um caso de teste, neste caso existe uma linha para cada par arco/caso de teste.

```
#include <functional>
#include <fstream>
#include <iostream>
#include <math.h>
#include <sstream>
#include <stdlib.h>
#include <string>
#include <vector>

using namespace std;

#define TAM_STRING 100
```

```
#define TAM_S 1000
#define ARQ_SAIDA "relacao_arco_caso.txt"
#define ARQ_SAIDA2 "ident_arco_metodo.txt"
#define ARQ_SAIDA3 "ident_arco.txt"


typedef struct metodo{
   char id[TAM_STRING];
   char nome[TAM_STRING];
}metodo;

typedef struct no{
   char id[TAM_STRING];
   metodo* met;
}no;

typedef struct arco{
   char id[TAM_STRING];
   no* n1;
   no* n2;
}arco;

typedef struct caso{
   char id[TAM_STRING];
   int num_arcos;
   arco** arcos;
}caso;

typedef struct descritor{

   int num_metodos;
   metodo* metodos;
   int num_nos;
   no* nos;
   int num_arcos;
   arco* arcos;
   int num_casos;
   caso* casos;
}descritor;


char* itoa(int num){
   char *strNum=(char *) malloc(8*sizeof(char)), charTemp;
   int count=0,n1=num,strB=0,strE=0;

   if( num < 0 ){
      strNum[strB++]='-';
      strE++;
      n1=-1*n1;
   }

   while( n1 >= 10 ){
      strNum[strE++] = (n1%10)+'0';
      n1 = n1/10;
   }
   strNum[strE] = n1+'0';
   strNum[strE+1] = '\0';
```

```
    while( strB<strE ){
        charTemp = strNum[strB];
        strNum[strB] = strNum[strE];
        strNum[strE] = charTemp;
        strB++;
        strE--;
    }
    return(strNum);
}

void cria_estrutura_metodo(ifstream &pointArq,descritor *desc){

    int cont;
    char s[TAM_S],*ps,*ps1;

    pointArq.clear();
    pointArq.seekg (0, ios::beg);

    while( !pointArq.eof() ){
    pointArq.getline( s, TAM_S);
    ps=strstr(s,"metodo");
    if( (ps-s)==0 )
        desc->num_metodos++;
    }

    pointArq.clear();
    pointArq.seekg (0, ios::beg);
    desc->metodos=new metodo[desc->num_metodos];
    cont=0;
    while( !pointArq.eof() ){
    pointArq.getline( s, TAM_S);
    ps=strstr(s,"metodo");
    if( (ps-s)==0 ){
        ps=strstr(s,"(");
        ps++;
        ps1=strstr(s,",");
        strncpy(desc->metodos[cont].id,s+(ps-s),ps1-ps);

        ps=ps1;
        ps+=2;
        ps1=strstr(s+(ps-s),",");
        strncpy(desc->metodos[cont].nome,s+(ps-s),ps1-ps-1);
        cont++;
    }
    }
}

void cria_estrutura_no(ifstream &pointArq,descritor *desc){

    int cont,cont1;
    bool encontrou;
    char s[TAM_S],*ps,*ps1;
    char met[TAM_S];

    pointArq.clear();
    pointArq.seekg (0, ios::beg);
```

```
    while( !pointArq.eof() ){
        pointArq.getline( s, TAM_S);
        ps=strstr(s,"no");
        if( (ps-s)==0 )
            desc->num_nos++;
    }

    pointArq.clear();
    pointArq.seekg (0, ios::beg);
    desc->nos=new no[desc->num_nos];
    cont=0;
    while( !pointArq.eof() ){
    pointArq.getline( s, TAM_S);
    ps=strstr(s,"no");
    if( (ps-s)==0 ){
        ps=strstr(s,"(");
        ps++;
        ps1=strstr(s,",");
        strncpy(desc->nos[cont].id,s+(ps-s),ps1-ps);

        ps=strstr(s,"[");
        ps++;
        ps1=strstr(s+(ps-s),"]");
        strncpy(met,s+(ps-s),ps1-ps);
        met[ps1-ps]='\0';
        desc->nos[cont].met=0;
        for(cont1=0,encontrou=false;!encontrou && (cont1<desc->num_metodos);cont1++)
            if( strcmp(met,desc->metodos[cont1].id )==0 ){
                desc->nos[cont].met=&desc->metodos[cont1];
                encontrou=true;
            }
        cont++;
    }
     }
}

void cria_estrutura_arco(ifstream &pointArq,descritor *desc){

    int cont,cont1,num_zeros,temp_num_zeros;
    bool encontrou;
    char s[TAM_S],*ps,*ps1;
    char no_id[TAM_S];

    pointArq.clear();
    pointArq.seekg (0, ios::beg);

    while( !pointArq.eof() ){
        pointArq.getline( s, TAM_S);
        ps=strstr(s,"arco");
        if( (ps-s)==0 )
            desc->num_arcos++;
    }

    pointArq.clear();
    pointArq.seekg (0, ios::beg);
    desc->arcos=new arco[desc->num_arcos];
```

```
    ps=itoa(desc->num_arcos);
    num_zeros=strlen(ps)-1;
    free(ps);
    cont=0;
    while( !pointArq.eof() ){
    pointArq.getline( s, TAM_S);
    ps=strstr(s,"arco");
    if( (ps-s)==0 ){
       strcpy(desc->arcos[cont].id,"a");
       ps=itoa(cont+1);
       temp_num_zeros=strlen(ps)-1;
       for(cont1=0;cont1<(num_zeros-temp_num_zeros);cont1++)
          strcat(desc->arcos[cont].id,"0");
       strcat(desc->arcos[cont].id,ps);
       free(ps);

       ps=strstr(s,"(");
       ps++;
       ps1=strstr(s,",");
       strncpy(no_id,s+(ps-s),ps1-ps);
       no_id[ps1-ps]='\0';
       desc->arcos[cont].n1=0;
       for(cont1=0,encontrou=false;!encontrou && (cont1<desc->num_nos);cont1++)
          if( strcmp(no_id,desc->nos[cont1].id)==0 ){
             desc->arcos[cont].n1=&desc->nos[cont1];
             encontrou=true;
          }

       ps=ps1+1;
       ps1=strstr(s,")");
       no_id[0]='\0';
       strncpy(no_id,s+(ps-s),ps1-ps);
       no_id[ps1-ps]='\0';
       desc->arcos[cont].n2=0;
       for(cont1=0,encontrou=false;!encontrou && (cont1<desc->num_nos);cont1++)
          if( strcmp(no_id,desc->nos[cont1].id)==0 ){
             desc->arcos[cont].n2=&desc->nos[cont1];
             encontrou=true;
          }

       cont++;
    }
    }
}


int cria_estrutura(char *arq_espec, descritor *desc){

    ifstream pointArq(arq_espec, ifstream::in);

    if( !pointArq )
       return( -1 );

    cria_estrutura_metodo(pointArq,desc);
    cria_estrutura_no(pointArq,desc);
    cria_estrutura_arco(pointArq,desc);

    return( 1 );
```

```
}

int cria_estrutura_casos(char *arq_casos, descritor *desc){

   ifstream pointArq(arq_casos, ifstream::in);
   int cont,cont1,cont2;
   bool encontrou;
   char s[TAM_S],*ps,*ps1,*ps2;
   char no_id1[TAM_S],no_id2[TAM_S];

   if ( !pointArq )
      return( -1 );

   while( !pointArq.eof() ){
      pointArq.getline( s, TAM_S);
      ps=strstr(s,"objeto_tst(");
      if( (ps-s)==0 )
         desc->num_casos++;
   }

   pointArq.clear();
   pointArq.seekg (0, ios::beg);

   desc->casos=new caso[desc->num_casos];
   cont=0;
   while( !pointArq.eof() ){
   pointArq.getline( s, TAM_S);
   ps=strstr(s,"objeto_tst(");
   if( (ps-s)==0 ){
      ps1=strstr(s,"[");
      desc->casos[cont].num_arcos=0;
      ps2=strstr(s,"]");
      ps=strstr(s+(ps1-s),",");
      while( (ps-s)<(ps2-s) ){
         desc->casos[cont].num_arcos++;
         ps=strstr(s+(ps-s)+1,",");
      }

      ps1=ps2+3;
      ps2=strstr(s+(ps1-s),",");
      ps2--;
      strncpy(desc->casos[cont].id,s+(ps1-s),ps2-ps1);
      desc->casos[cont].id[ps2-s]='\0';

      desc->casos[cont].arcos=new arco*[desc->casos[cont].num_arcos];

      ps1=strstr(s,"[");
      ps1+=2;
      ps2=strstr(s,"]");

      for(cont2=0;cont2<desc->casos[cont].num_arcos;cont2++){
         ps=strstr(s+(ps1-s),",");
         ps--;
         no_id1[0]='\0';
         strncpy(no_id1,ps1,ps-ps1);
         no_id1[ps-ps1]='\0';
```

```
            ps1=ps+3;
            ps=strstr(s+(ps1-s),",");
            if( (ps-s)<(ps2-s) )
               ps--;
            else
               ps-=2;
            no_id2[0]='\0';
            strncpy(no_id2,ps1,ps-ps1);
            no_id2[ps-ps1]='\0';

            desc->casos[cont].arcos[cont2]=0;
            for(cont1=0,encontrou=false;!encontrou && (cont1<desc->num_arcos);cont1++)
               if( (strcmp(desc->arcos[cont1].n1->met->id,no_id1)==0) && (strcmp(desc->arcos
               [cont1].n2->met->id,no_id2)==0) ){
                  desc->casos[cont].arcos[cont2]=&desc->arcos[cont1];
                  encontrou=true;
               }
         }
         cont++;
      }
      }
      return( 1 );
}


int main(int argc, char *argv[]){

if( argc!=3 ){
   cout << "numero incorreto de parametros !!!" << endl;
   cout << "1o. caminho/nome do arquivo com o modelo de teste;" << endl;
   cout << "2o. caminho/nome do arquivo com os casos de teste." << endl;
   exit(1);
}

ofstream pointArqSaida(ARQ_SAIDA,ios::trunc);
ofstream pointArqSaida2(ARQ_SAIDA2,ios::trunc);
ofstream pointArqSaida3(ARQ_SAIDA3,ios::trunc);
descritor *desc=new descritor;
int cont,cont1,retorno;

desc->num_metodos=desc->num_nos=desc->num_arcos=0;
desc->metodos=0;
desc->nos=0;
desc->arcos=0;

retorno=cria_estrutura(argv[1],desc);
if( retorno!=1 ){
   cout << "problemas na abertura do arquivo com o modelo de teste !!!" << endl;
   return(0);
}
cout << "arquivo com o modelo de teste foi aberto !!!" << endl;

retorno=cria_estrutura_casos(argv[2],desc);
if( retorno!=1 ){
   cout << "problemas na abertura do arquivo com os casos de teste !!!" << endl;
   return(0);
}
cout << "arquivo com os casos de teste foi aberto !!!" << endl;
```

```
if( pointArqSaida && pointArqSaida2 && pointArqSaida3 ){
   for(cont=0;cont<desc->num_casos;cont++)
      for(cont1=0;cont1<desc->casos[cont].num_arcos;cont1++)
         if( desc->casos[cont].arcos[cont1]!=0 )
            pointArqSaida << desc->casos[cont].arcos[cont1]->id << " " << desc->casos[
            cont].id << endl;
   pointArqSaida.close();

   for(cont=0;cont<desc->num_arcos;cont++){
      pointArqSaida2 << desc->arcos[cont].id << " ";
      if( desc->arcos[cont].n1!=0 )
         pointArqSaida2 << desc->arcos[cont].n1->met->nome << " ";
      if( desc->arcos[cont].n2!=0 )
         pointArqSaida2 << desc->arcos[cont].n2->met->nome << " ";
      pointArqSaida2 << endl;
   }
   pointArqSaida2.close();

   for(cont=0;cont<desc->num_arcos;cont++){
      pointArqSaida3 << desc->arcos[cont].id << " ";
      if( desc->arcos[cont].n1!=0 )
         pointArqSaida3 << desc->arcos[cont].n1->met->id << " ";
      if( desc->arcos[cont].n2!=0 )
         pointArqSaida3 << desc->arcos[cont].n2->met->id << " ";
      pointArqSaida3 << endl;
   }
   pointArqSaida3.close();
}
else
   cout << "problemas na abertura de um dos arquivos de saida !!!" << endl;

return(1);
}
```