UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Davi Augusto Gadêlha Silva

EvolUniT: Geração e Evolução de Testes de Unidade em Java utilizando Algoritmos Genéticos

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: Profª. FLÁVIA DE ALMEIDA BARROS
CO-ORIENTADOR: Prof. RICARDO BASTOS C. PRUDÊNCIO

RECIFE, JUNHO/2008

Pós-Graduação em Ciência da Computação

"EvolUniT: Geração e Evolução de Testes de Unidade em Java utilizando Algoritmos Genéticos"

Por

Davi Augusto Gadelha Silva

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, JUNHO/2008

Agradecimentos

Em primeiro lugar, agradeço à minha orientadora, Flávia Barros; ao meu co-orientador Ricardo Prudêncio e ao Centro de Informática por me terem acolhido após meu ingresso na UFPE e por todo o suporte oferecido durante a jornada de esforços e aprendizado propiciados pelo curso de Mestrado desta Instituição.

Agradeço também ao projeto CIn / BTC da Motorola por me haver concedido uma bolsa de estudos e a todos os seus integrantes, pelo apoio e suporte logístico dispensados, fundamentais para a realização deste trabalho de Mestrado.

Por um apoio diferente, mas não menos importante, agradeço aos meus pais, Frederico Holanda e Carla Gadêlha, e à minha noiva, Nadilma Nunes, que me deram o suporte emocional necessário para enfrentar as dificuldades encontradas no caminho percorrido.

E agradeço também aos professores e amigos que me incentivaram, ajudaram ou simplesmente estiveram lutando ao meu lado durante os mais de dois anos que constituíram esta jornada de conhecimento.

Resumo

Este trabalho apresenta a ferramenta EvolUniT (Evolutionary Unit Testing), uma ferramenta para automatização de testes de unidade de código orientado a objetos (classes Java). A EvolUniT recebe como entrada uma classe Java a ser testada; gera uma classe de teste usando o framework JUnit; gera dados (parâmetros de construtores e métodos) inicialmente aleatórios para compor os casos de teste; e utiliza um Algoritmo Genético (AG) para evoluir os dados, de acordo com uma função de aptidão criada com base nas coberturas de código capturadas.

A evolução dos dados se dá através de sucessivas execuções da classe sendo testada, até que um número máximo de gerações do AG seja atingido ou que uma cobertura máxima pré-definida seja atingida. A ferramenta foi implementada em Java, em forma de plug-in do Eclipse. A ferramenta proporciona uma semi-automação de testes de unidade, ao invés de automação completa, pois em alguns casos, o engenheiro de software ou de testes precisará complementar manualmente as classes de teste geradas. A vantagem desta semi-automação é que o conhecimento do desenvolvedor ou testador será acrescido aos testes gerados pela ferramenta, possibilitando assim melhores resultados.

Foram realizados três estudos para avaliar a EvolUniT, e os resultados alcançados foram satisfatórios. A EvolUniT traz contribuições para duas áreas diferentes. Para a Engenharia de Software, com a semi-automação do processo de testes de unidade, reduz-se significativamente o tempo e o esforço por parte dos desenvolvedores, já que estes passam a usar seus conhecimentos para configurar a ferramenta, ao invés de escrever as classes de teste. Para a área de Computação Inteligente, a contribuição é na utilização de uma técnica de otimização evolutiva, os Algoritmos Genéticos, para resolver o problema da escolha de bons dados para testes estruturais, que nem sempre é bem resolvido por algoritmos convencionais ou técnicas aleatórias.

Abstract

This work presents EvolUniT (Evolutionary Unit Testing), a tool for automation of unit tests of object-oriented software (Java classes). EvolUniT receives as entry a Java class to be tested; creates a test class using the JUnit framework; initially generates random test data (parameters of consctructors and methods) to compose test cases; and uses a genetic algorithm (GA) to evolve the test data, according to a fitness function based on code coverage.

The evolution of data takes place through successive executions of the class been tested, until the AG maximum number of generations is reached or that a pre-set maximum coverage is reached. The tool has been implemented in Java as an Eclipse plug-in. The tool provides a semi-automation of unit testing, rather than full automation, because in some cases, the software or test engineer will have to manually complete the generated test classes. The advantage of the semi-automation is that the developer's knowledge will be added to the testes generated by the tool, thus enabling better results.

Three case studies were conducted to assess EvolUniT, and the results were satisfactory. EvolUniT brings contributions to two different áreas. For Software Engineering, with a semi-automation of the unit testing process, the time and effort of the developers is reduced significantly, since they only have to configure the tool, instead of write the test classes. In the area of Intelligent Computing, the contribution is to use a technique of evolutionary optimization, the Genetic Algorithms, to solve the problem of search for good test data for structural testing, which is not always well done by conventional algorithms or random techniques.

Sumário

Lista de Figuras

Lista de Quadros e Tabelas

Lista de Abreviações

AEs: Algoritmos Evolutivos

AGs: Algoritmos Genéticos

CA: Coverage Analyser

ES: Engenharia de Software

GAC: Genetic Algorithm Component

IA: Inteligência Artificial

OO: Orientado a Objetos

OS: Open-Source

TCG: Test Case Generator

TE: Teste Evolutivo

1 Introdução

A engenharia de software (ES) tem como um de seus objetivos prover qualidade à aplicação a ser desenvolvida. Vários processos de desenvolvimento foram criados e estabelecidos ao longo da última década, a maioria deles incluindo testes como uma das etapas mais importantes para a garantia da qualidade do software [Nagarajan et al., 2003].

Diante disso, os engenheiros de software passaram a valorizar mais a atividade de testes, que foi desmembrada em várias subetapas ao longo de um processo de desenvolvimento incremental - em contraste com a antiga prática de se ter apenas uma etapa de testes ao final do ciclo de vida de um software. Contudo, o processo de teste é caro e demorado, especialmente em aplicações críticas de segurança, tipicamente consumindo pelo menos 50% dos custos totais envolvidos no desenvolvimento de software [Beizer, 1996]. Claramente, o uso de métodos e técnicas capazes de agilizar esse processo, como sua automação (total ou parcial), poderia resultar em redução de custos, esforço e tempo, garantindo ainda a qualidade do software sendo desenvolvido.

Dentre as técnicas usadas na automação de teste, destaca-se a Execução Simbólica [McMinn, 2004]. Esta é uma técnica estática (não efetua a execução real do programa), que consiste no processo de atribuir expressões a variáveis de um programa enquanto um caminho é seguido através da estrutura do código. Uma das vantagens desta técnica é que ela dispensa a execução do código, poupando o tanto o tempo gasto com a sua instrumentação (para possibilitar o monitoramento dos caminhos percorridos), como o tempo da própria execução do SW. Contudo, a desvantagem de utilizar esta técnica cresce à medida que a complexidade do código também cresce. A presença de loops no código, bem como a utilização de estruturas dinâmicas, reduzem a eficácia desta técnica.

Como alternativa a essa abordagem, citamos as técnicas de otimização meta-heurística, que têm sido utilizadas para auxiliar em algumas etapas do desenvolvimento de software (sobretudo em geração de dados para testes, uma das principais formas de sua automação) [McMinn, 2004]. Dentre essas técnicas, destacam-se os Algoritmos Evolutivos [Mantere e Alander, 2005], opção que se mostra adequada quando o conjunto de possíveis

entradas para testar o software é grande, e a escolha das melhores entradas (no conjunto de possibilidades) pode ser guiada por uma "função de aptidão" (*fitness function*).

Nesse contexto, surge o conceito de Teste Evolutivo *(Evolutionary Testing),* que se trata de uma técnica de ES baseada em busca [Baresel et al, 2004]. Esta técnica utiliza algoritmos evolutivos para gerar automaticamente dados de entrada para testes com boa qualidade, com o objetivo de cobrir determinados critérios de avaliação (*e.g.,* cobertura, tempo de execução), utilizando algoritmos evolutivos para realizar uma busca no conjunto de possíveis entradas do programa sendo testado.

Os Algoritmos Genéticos (AGs), que são o tipo mais conhecido de algoritmos evolutivos, têm sido utilizados com sucesso para realização de testes evolutivos [Mantere e Alander, 2005]. De fato, essa técnica já foi bastante empregada para testar softwares procedimentais [Sthamer, 1996][Wegener et al., 2002][Baresel et al., 2004]. Contudo, sua utilização para teste de software orientado a objetos (OO) ainda é recente, havendo poucos trabalhos disponíveis [Tonella, 2004] [Wappler e Lammermann, 2005].

1.1 Trabalho Realizado

Este trabalho de mestrado investigou técnicas para geração automática de dados de teste, em particular, os algoritmos genéticos. Como resultado, foi implementada a EvolUnit, uma ferramenta para automação de testes de unidade de código OO (classes Java), que usa Algoritmos Genéticos para escolher e evoluir, dentre um conjunto de candidatos, os parâmetros de entrada dos métodos sendo testados. A evolução da ferramenta tem o objetivo de maximizar a cobertura de decisões (*branch coverage*) dos métodos públicos das classes sendo testadas, para que o máximo possível de suas estruturas sejam exercitadas.

O EvolUnit foi desenvolvido como *plugin* da IDE (*Integrated Development Environment*) Eclipse (uma das IDEs mais usadas para desenvolvimento em linguagem Java), para geração de classes de teste JUnit, que é o *framework* mais utilizado para testes unitários de código Java.

O trabalho foi desenvolvido em várias etapas. Inicialmente, foi realizada uma pesquisa bibliográfica a respeito de técnicas utilizadas na escolha automática de dados para realização dos testes. O foco do estudo foi voltado para os trabalhos de Teste Evolutivo, que foi a técnica escolhida para a realização da automação de testes desejada. Esse estudo revelou a possibilidade de utilização desta técnica na automação de teste, bem como em mais de uma fase em um processo de testes.

A partir daí, definimos uma ferramenta para geração automática de testes de unidade de classes Java, usando uma abordagem híbrida, que possui características de duas abordagens utilizadas em Teste Evolutivo: (1) a Convencional, utilizada para testar códigos procedimentais; e (2) Orientada a Objetos, abordagem mais complexa para testar métodos públicos de classes OO. A ferramenta se trata de um plug-in que executa na plataforma Eclipse (versão 3.2 ou superiores). Sua idéia básica é gerar classes de teste, usando o framework *JUnit*, que utiliza dados gerados inicialmente de forma aleatória para testar métodos públicos de outras classes. A seguir, esses dados são evoluídos com a utilização de um algoritmo genético (AG), através de sucessivas execuções, com o objetivo de maximizar a cobertura de decisões desses métodos. É importante salientar que a ferramenta não gera pré-condições às vezes necessárias antes das chamadas dos métodos públicos das classes. Além disso, a ferramenta evolui com o AG apenas os dados (parâmetros dos construtores e métodos) que são de tipos primitivos ou *Strings*.

Foram realizados três estudos de caso para avaliar o EvolUniT. O primeiro consistiu na realização de vários experimentos em apenas uma classe Java, para comparação da geração evolutiva de dados da ferramenta com a geração aleatória, alterando as configurações genéticas do AG para cada experimento. Com isso, a cobertura de decisões máxima atingida pelos dados com geração evolutiva foi de 100%, em contraste com o máximo de 97,22% da geração aleatória.

O segundo estudo consistiu na estimativa de esforço poupado com a semi-automação proporcionada pelo EvolUniT. Utilizando um corpus de teste de 6 classes aproveitadas de um sistema desenvolvido pelo CIn/Motorola, foi obtido um resultado de aproximadamente 44,21% de esforço poupado, quando se comparando à geração automática com a escrita manual de testes.

O terceiro estudo teve o foco na comparação do EvolUniT com outra ferramenta de geração automática de casos de teste Java, chamada Randoop. Para esse estudo foi levado em consideração o mesmo corpus de teste do estudo 2. Os resultados, com relação à cobertura de decisões, foram melhores com a utilização da EvolUniT. A Randoop, porém, deteve um melhor desempenho, gerando uma grande quantidade de testes em pouco tempo. A Randoop também gera *test oracles* automaticamente, validações dos métodos de teste que a EvolUniT ainda não gera.

A ferramenta implementada tem duas contribuições principais, para duas áreas diferentes. Para a Engenharia de Software, a contribuição é a semi-automação no processo de realização de testes de unidade, causando uma redução de tempo e esforço por parte de desenvolvedores Java, já que estes passam a apenas configurar a ferramenta e completar os testes gerados quando necessário, ao invés de escreve-los de forma completa, no mesmo ambiente em que desenvolvem sua aplicação. Para a área de Computação Inteligente, a contribuição é a utilização de uma técnica de otimização evolutiva, que é o caso dos Algoritmos Genéticos, para resolver um problema que pode não ser bem resolvido com algoritmos convencionais ou técnicas aleatórias, que é o problema da escolha de bons dados para testes.

1.2 Organização da dissertação

Além deste capítulo de introdução, esta dissertação é composta de outros cinco capítulos descritos brevemente a seguir:

Capítulo 2:

Apresenta uma introdução a Teste de Software. Utilizando uma abordagem *top-down*, que começa com Engenharia de Software, qualidade e processo de desenvolvimento de software, e vai até Teste de Software. O capítulo expõe os principais conceitos relacionados com esse assunto, assim como as principais técnicas ou abordagens disponíveis para Teste. Além disso também são vistas as fases de teste presentes atualmente em um processo que adota um modelo de desenvolvimento ou ciclo de vida incremental, com destaque para testes de unidade, bem como os principais critérios utilizados para avaliação desses testes.

Capítulo 3:

Este capítulo apresenta o conceito de Teste Evolutivo, assim como outras técnicas utilizadas para automação de testes. Algumas técnicas de otimização baseadas em busca metaheurística são explicadas, com destaque para Algoritmos Genéticos, que se trata de uma técnica evolutiva. Trabalhos anteriores de teste evolutivo também são expostos, assim como uma análise dos mesmos.

Capítulo 4:

Descreve a ferramenta desenvolvida nesse trabalho de mestrado, que se trata de um plug-in para o Eclipse que combina características das duas abordagens existentes de Teste Evolutivo. O capítulo explica o propósito da ferramenta, exibe sua arquitetura e explica os seus principais módulos.

Capítulo 5:

Apresenta os estudos de caso realizados na ferramenta de Teste Evolutivo implementada e análises dos resultados.

Capítulo 6:

Apresenta as considerações finais sobre o trabalho desenvolvido, suas principais contribuições e algumas propostas de trabalhos futuros.

2 Teste de Software

Foi dito anteriormente que a Engenharia de Software visa garantir a qualidade de qualquer aplicação sendo desenvolvida, e para isso conta com a etapa de testes, que está presente em qualquer processo de desenvolvimento de software da atualidade.

Teste de Software é o processo de executar um programa com o objetivo de encontrar defeitos [Myers, 1979]. Um teste bem sucedido é aquele que consegue montar um conjunto de casos de teste que detecte falhas no software sendo testado (teste de defeito), ou ainda que exercite a maior parte de sua estrutura, para tentar garantir que o software não possui falhas (teste de validação). Um *caso de teste* é composto por um conjunto de entradas de teste, condições de execução e resultados esperados desenvolvidos para um objetivo particular, (como exercitar um caminho particular de um programa ou verificar a conformidade com um requisito específico [IEEE/ANSI, 1990]).

Neste capítulo, veremos alguns conceitos básicos da área de Teste de Software, incluindo sua contextualização dentro de um processo de desenvolvimento de software. Serão apresentadas as fases de teste que são realizadas em paralelo com as atividades de um processo interativo, bem como as principais técnicas de teste que são utilizadas durante essas fases. Por fim, veremos os critérios de análise de testes mais utilizados, e algumas considerações finais serão expostas sobre o assunto abordado.

2.1 Processo de Desenvolvimento de Software

A ES é muitas vezes tratada como uma "tecnologia em camadas" [Pressman, 2006], e toda iniciativa da mesma deve ser apoiada por um compromisso com a qualidade. Uma dessas camadas da ES é a de qualidade, e dentro dela encontram-se os processos de desenvolvimento de software com seus métodos e ferramentas.

Segundo [Sommerville, 2003], Processo de Desenvolvimento de Software (PDS) é o conjunto de atividades e resultados associados que levam à produção de software. Ao longo da história da ES, vários modelos de processo foram concebidos, porém nenhum pode ser considerado ideal, devido às suas divergências e a aplicabilidade de cada um deles

em diferentes contextos. Apesar das diferenças, todos compartilham atividades fundamentais como: especificação de requisitos de software; projeto de software; implementação de software; validação e testes de software e evolução de software.

As atividades relacionadas a um processo de software estão diretamente vinculadas com a produção do software como produto final. A fim de especificar quais atividades devem ser executadas, e em qual ordem, diversos modelos de desenvolvimento de software foram criados. Os principais modelos criados foram:

(1) o modelo *Cascata*, também conhecido como abordagem *top-down*, que tem como características a simplicidade e a forma seqüencial como cada atividade é executada (apenas uma vez), sendo o produto de cada atividade visto como entrada para a próxima atividade;

(2) o modelo *Espiral*, no qual o projeto de SW é tratado como uma série de pequenos ciclos, cada um construindo uma versão de um software executável;

(3) o modelo *Iterativo e Incremental*, que combina aspectos do modelo em cascata, aplicados, porém, de maneira iterativa, de forma que cada iteração ou seqüência de atividades produz "incrementos" do software que já podem ser entregues.

(4) o modelo de *Prototipagem*, que, como o próprio nome diz, é baseado na construção de protótipos, e procura superar as limitações do modelo Cascata.

Após falar dos modelos mais conhecidos, é importante também citar o processo de desenvolvimento de software mais famoso, que é o Processo Unificado da Rational, também chamado de RUP (*Rational Unified Process*). O RUP é um processo de software proposto por [Booch et al., 1999] que utiliza a UML (*Unified Modeling Language*) [UML, 2008] como notação de uma série de modelos que compõem os principais resultados das atividades do processo. Esse processo é composto por quatro fases: Concepção, Elaboração, Construção e a Transição.

O RUP adota um modelo iterativo e incremental, em que cada projeto constitui um ciclo, que entrega uma liberação do produto. Além das quatro fases, o RUP possui fluxos de trabalho (*workflows*), que são sub-processos, ou disciplinas, em que as atividades técnicas são divididas. As disciplinas definidas no processo são: Modelagem de Negócios,

Requisitos, Análise e Desenho, Implementação, Testes e Implantação. Uma visão geral deste processo está ilustrada na figura 2.1 [RUP, 2008].

Um outro processo que merece destaque pelas suas características modernas, ágeis e ampla utilização em pequenas organizações, é o XP (*Extreme Programming*) [XP, 2008]. O XP é uma metodologia mais leve para equipes de desenvolvimento de software de pequenas e médias empresas com requisitos de mudanças rápidas [Beck, 1999]. As principais características do XP são: pequenos ciclos com concreto e contínuo feedback; abordagem incremental, que se desenvolve durante toda a vida do projeto; agenda flexível da implementação de funcionalidades; confiança em testes automáticos escritos por programadores e clientes para monitorar o progresso do software; confiança na comunicação oral, testes e códigos-fonte para comunicar a estrutura e objetivo do sistema; e confiança no processo de desenho (modelagem) de forma evolutiva.

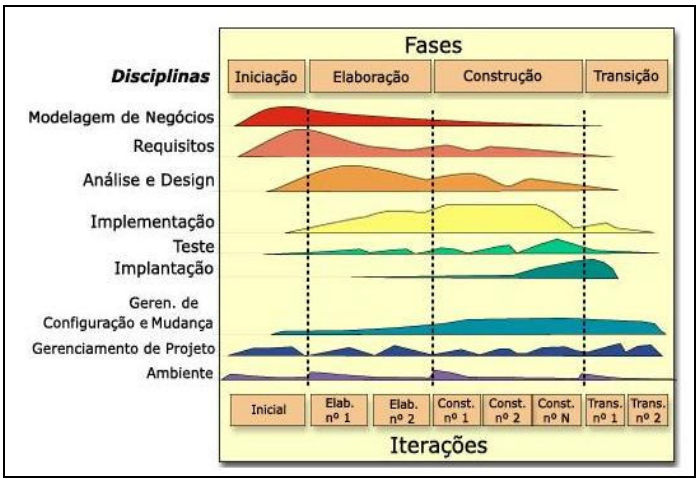

Figura 2.1 - Processo Unificado (RUP) [RUP, 2008].

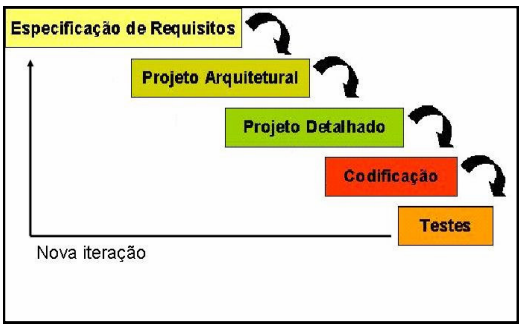

Figura 2.2 - Modelo de desenvolvimento Iterativo e Incremental.

O modelo Iterativo e Incremental, ilustrando na figura 2.2, além de servir de base para outros modelos, é um dos mais adotados ou indicados por processos de software, como é o caso do RUP. Neste processo as iterações são passos em um fluxo maior que inclui todas as atividades, e os incrementos são evoluções do produto. Esse modelo possui vantagens como: a possibilidade de avaliação de riscos e pontos críticos de um projeto mais cedo do que nos demais processos, e identificar medidas para removê-los ou minimizá-los; redução dos riscos envolvendo custos a um único incremento; definição de uma arquitetura que melhor possa orientar todo o desenvolvimento; disponibilização natural de um conjunto de regras para melhor controlar os inevitáveis pedidos de alterações futuras; entre outros. Para mais informações sobre este ou os outros modelos de processo, ver [Sumerville, 2003] [Pressman, 2006].

Na próxima seção, veremos em detalhes a etapa de testes de um processo de desenvolvimento iterativo e incremental. Focamos apenas nesse modelo de processo porque ele foi utilizado como base para o desenvolvimento do trabalho de mestrado aqui apresentado.

2.2 Fases de Teste

Nos últimos anos, a visão de teste de software evoluiu, e testes não são mais vistos como apenas como uma etapa posterior à implementação do software, sendo executado de forma integrada, não em etapas, mas de uma só vez, processo este chamado de *Big-Bang*

Testing [Jorgensen, 1995]. O fluxo padrão de atividades de um processo de desenvolvimento iterativo incremental era colocar apenas uma etapa de testes, normalmente apenas no final (ver figura 2.1).

Atualmente, a atividade de teste é vista como um processo formado por etapas que podem ser realizadas em vários pontos durante o processo de desenvolvimento convencional; ou seja, os testes são definidos e executados paralelamente ao desenvolvimento do software [Coelho et. al., 2006]. Sendo assim, cada fase de desenvolvimento ou manutenção deve possuir uma fase de testes correspondente (ver figura 2.3). É importante salientar que cada algumas destas fases de teste podem ser conduzidas de forma automática. O capítulo 3 traz detalhes de técnicas que podem ser usadas para auxiliar o processo de automação de determinadas fases.

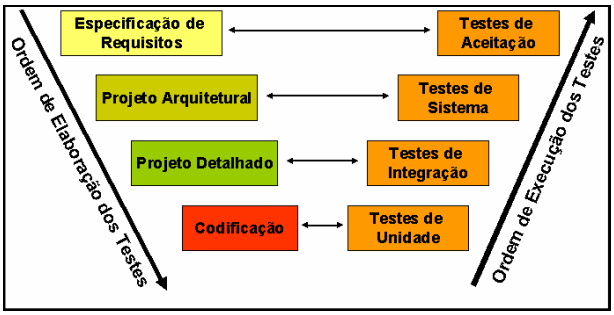

Figura 2.3 - Correspondência entre processo de desenvolvimento e de testes [Myers, 2004].

Nas próximas seções, as fases de teste presentes em um processo serão apresentadas de forma sucinta.

2.2.1 Teste de Unidade

O Teste de Unidade, também chamado de Teste Unitário, é a fase de um processo de teste na qual as menores unidades de um software em desenvolvimento (componentes, módulos) são testadas, normalmente pelos próprios desenvolvedores (diferentemente das

demais fases de teste). Deste modo, o universo alvo desse tipo de teste normalmente são funções e métodos ou mesmo pequenos fragmentos de código.

A complexidade dos testes de unidade e dos erros descobertos é limitada pelo escopo restrito estabelecido para este tipo de teste. O teste de unidade enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Esse tipo de teste pode ser conduzido em paralelo para diversos componentes [Pressman, 2006].

Pode-se observar uma tendência de pular a fase de testes unitários, partindo para o teste de entidades funcionais maiores; isto é, construir o software usando uma abordagem "*big-bang*", onde todos os componentes são combinados com antecedência [Jorgensen, 1995]. Isto normalmente resulta em caos, pois muitos erros podem ser encontrados de uma só vez e a sua correção fica difícil, pois há muito espaço para procurar e isolar a causa do erro. Uma situação como essa demonstra a importância dos testes de unidade.

2.2.2 Teste de Integração

Depois que as funcionalidades de cada unidade foram testadas, as entidades funcionais maiores são construídas juntando estas unidades. Este processo de combinação das unidades, que ocorre até que o subsistema ou sistema esteja completo, é chamado de integração do sistema.

Após a integração, as unidades precisam ser novamente testadas, a fim de verificar se elas funcionam corretamente quando combinadas em entidades maiores. *Teste de integração* é uma fase sistemática na qual, enquanto a arquitetura do software está sendo construída, ao mesmo tempo, testes são conduzidos para descobrir erros associados às interfaces entre as unidades [Pressman, 2006]. A melhor forma de conduzir esses testes é de forma incremental, de forma que o programa seja construído e testado em pequenos incrementos, fazendo com que os erros sejam mais fáceis de serem isolados e corrigidos.

Esta fase, assim como a de Testes de Unidade, é igualmente importante para que erros entre interfaces não sejam descobertos mais na frente, quando o foco já é nos testes do funcionamento do sistema como um todo.

2.2.3 Teste de Sistema

Após as unidades serem testadas de forma isolada e de forma integrada, é preciso executar o sistema do ponto de vista do usuário final, observando cada funcionalidade em busca de falhas. Nessa fase, os testes devem ser executados em condições similares àquelas em que o usuário utilizará o sistema no seu dia-a-dia, considerando ambiente, interfaces gráficas e massa de dados. Quanto mais próximas estas condições estiverem das reais, mais efetivo será o teste de sistema.

De fato, Teste de sistema consiste em uma série de diferentes testes cuja finalidade principal é exercitar por completo o sistema. Cada um destes testes tem um objetivo específico, porém a junção deles deve verificar se os elementos do sistema foram integrados de forma adequada e se executam as funcionalidades corretamente [Pressman, 2006]. São estes:

(1) teste funcional, que testa as regras de negócio e condições válidas e inválidas;

(2) teste de recuperação, que verifica a eficiência dos procedimentos de recuperação quando o software falha;

(3) teste de desempenho, que verifica o tempo de resposta e processamento para diferentes configurações;

(4) teste de estresse, que executa o sistema com demanda de recursos em quantidades grandes;

(5) teste de segurança, que verifica se os mecanismos de proteção de acesso e de dados estão funcionando; e

(6) teste de interfaces com o usuário, que testa a navegação e consistência das interfaces.

Antes de passar para a última fase de testes, que envolve a participação do cliente, é preciso que os testes de unidade, de integração e de sistema tenham sido conduzidos da melhor forma possível, para evitar excesso de não conformidades com os requisitos do cliente.

2.2.4 Teste de Aceitação

Nesta fase, os testes devem ser conduzidos pelos usuários finais do sistema. O objetivo é de demonstrar a conformidade com os requisitos definidos pelo usuário. Esta fase é bastante importante, pois seu resultado irá determinar se um sistema satisfaz ou não os critérios de aceitação, e permitir ao cliente julgar se aceita ou não o sistema.

A maioria dos desenvolvedores de software realiza 2 tipos de teste, chamados de *alfa* e *beta,* para conduzir os testes de aceitação. O teste *alfa* é conduzido pelo usuário, geralmente nas instalações do desenvolvedor. No decorrer dos testes, o desenvolvedor observa e registra erros e inconsistências. O teste *beta* é também conduzido pelo usuário, porém em suas próprias instalações. Diferente do teste *alfa*, aqui o desenvolvedor geralmente não está presente. O próprio cliente registra todos os problemas que encontra durante o teste *beta*, e os relata ao desenvolvedor em determinados intervalos de tempo.

2.2.5 Teste de Regressão

Testes de Regressão são usados em uma nova versão do software, ou quando existe a necessidade de se executar um novo ciclo de teste durante o processo de desenvolvimento. Ele consiste em executar novamente subconjuntos de testes que já foram conduzidos, a cada versão do software ou a cada ciclo, para garantir que as modificações ocorridas em cada um destes ciclos não causaram efeitos colaterais.

Geralmente, ferramentas de automação são utilizadas para aumentar a produtividade e viabilidade dos testes, para que os testes que já foram executados anteriormente possam ser novamente executados com maior agilidade.

A seguir, discutiremos abordagens dentro das quais os testes podem ser desenvolvidos e conduzidos.

2.3 Abordagens de Teste

Diferentes abordagens podem ser utilizadas para conduzir os testes em cada fase apresentada na seção anterior. Também chamadas de “Técnicas de Teste”, cada uma dessas

abordagens leva em consideração diferentes aspectos para determinar os requisitos do teste, e avaliá-lo.

Esta seção apresenta as quatro abordagens mais difundidas para teste de software: funcional ou caixa preta, estrutural ou caixa branca, caixa cinza e teste baseado em falhas. Destas, as mais conhecidas e usadas são as duas primeiras. Discutiremos brevemente cada abordagem, indicando quais critérios podem ser usados em cada uma delas. Essas abordagens podem ser aplicadas tanto a software procedimental como a software orientado a objetos, e várias outras técnicas podem ser utilizadas em conjunto com elas para a obtenção de melhores resultados, ou até mesmo para dar suporte a um processo de automação de testes.

2.3.1 Teste Funcional (Caixa Preta)

Esta abordagem (ou técnica) de teste tem esse nome porque trata o componente de software a ser testado como se fosse uma caixa-preta, ou seja, não considera seu comportamento interno. Aqui, apenas o "lado externo" do programa fica visível (ver figura 2.4). Desta forma, o testador usa basicamente a especificação do software para obter os requisitos do teste ou os dados de teste, sem ter nenhuma preocupação com a implementação [Myers, 1976].

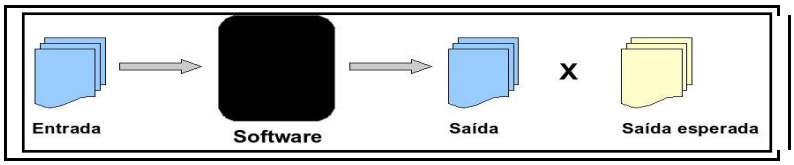

Figura 2.4 - Abordagem Funcional ou Caixa Preta (adaptada de [Coelho, 2005]).

O teste caixa-preta também é conhecido como teste funcional, pois se baseia na identificação de requisitos funcionais, ou seja, funções matemáticas são especificadas usando apenas suas entradas e saídas, sem o conhecimento da estrutura do código [Mayrhauser, 1990]. Sendo assim, uma especificação de alta qualidade que cubra os requisitos do cliente é fundamental para a aplicação do teste funcional.

Uma das grandes vantagens do teste caixa preta é que ele pode ser aplicado em todas as fases de teste (de unidade, integração, sistema e aceitação), quase sem nenhuma modificação na forma de aplicação dos testes, apesar de ser mais comum a sua utilização nas duas últimas fases do processo de testes. Além disso, já que os critérios de teste funcionais são baseados nas especificações, eles também se tornam independentes de linguagem ou plataforma. Essa característica faz com que eles possam ser usados para testar software procedimental, orientado a objetos, e/ou orientado a aspectos, além de componentes de software [Binder, 1999] [Offutt e Irvine, 1995].

As categorias de erros que o teste caixa-preta procura encontrar são: (1) funções incorretas ou omitidas, (2) erros de interface, (3) erros de estrutura de dados ou de acesso a bases de dados externas, (4) erros de comportamento ou desempenho, e (5) erros de inicialização e término [Pressman, 2006].

A aplicação da abordagem Caixa Preta implica na utilização de um ou mais métodos que foram criados para atender os critérios dessa abordagem. Alguns dos métodos utilizados por esta abordagem são: Particionamento de Equivalência (*Equivalence Partition*) e Análise de Valores de Fronteira (*Boundary Value Analyses*) [Pressman, 2006]. O método de Particionamento de Equivalência divide o domínio de entrada de um programa em classes, para que casos de teste sejam derivados destas classes. Dessa maneira, um caso de teste definido por esse método é eficaz se descobre classes de erros, reduzindo assim o número total de casos de teste desenvolvidos. O método de Análise de Valores de Fronteira parte do princípio de que muitos erros ocorrem nas fronteiras do domínio de entrada, e não no "centro". Esse método é um complemento do método de particionamento de equivalência, e procura selecionar casos de teste que exercitam nessas fronteiras.

Apesar de suas vantagens, a aplicação apenas da abordagem Caixa Preta não é suficiente, pois não é possível garantir que determinadas partes essenciais da implementação do software foram exercitadas. Além deste problema, as especificações do sistema, que são a única fonte de informação para os testes funcionais, podem estar incompletas ou escritas de forma ambígua, tornando os testes também insatisfatórios. A

próxima seção trata de uma outra importante abordagem de teste, que complementa a abordagem Caixa Preta.

2.3.2 Teste Estrutural (Caixa Branca)

A abordagem de teste Caixa-Branca, ou Estrutural, avalia o comportamento interno de um componente de software (ver figura 2.5). Ela atua diretamente no código-fonte de um módulo do software para avaliar aspectos como: condições, fluxo de dados, ciclos, caminhos lógicos, entre outros. Esta abordagem (ou técnica) é vista como um complemento da abordagem funcional.

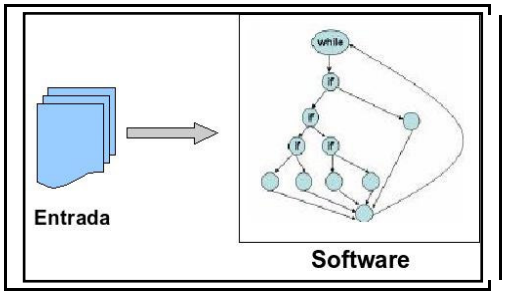

Figura 2.5 - Abordagem Estrutural ou Caixa Branca (adaptada de [Coelho, 2005]).

A abordagem caixa-branca é recomendada para as fases de teste de unidade e teste de integração, já que ambos os tipos são realizados diretamente no código fonte sob responsabilidade de desenvolvedores de software.

Usando critérios de teste caixa-branca, o engenheiro de software pode elaborar casos de teste que cubram diversos aspectos de um componente de software, realizando tarefas como: (1) garantir que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez, (2) exercitar todas as decisões lógicas com os valores “verdadeiro” e “falso”, (3) executar todos os ciclos nos seus limites e dentro de seus intervalos operacionais, e (4) exercitar as estruturas de dados internas para garantir sua validade [Pressman, 2006].

As informações obtidas pela aplicação desses testes são também bastante relevantes para outras atividades de engenharia de software, como: manutenção, *debugging*, estimação da confiabilidade do software, e melhoria de processo [Harrold, 2000] [Pressman, 2006].

Um exemplo prático e muito conhecido da aplicação abordagem caixa-branca é a utilização do *framework JUnit* [JUnit], para o desenvolvimento de classes de teste (*test cases*) para realizar testes de unidade de classes Java. Esta aplicação é um dos focos deste trabalho.

2.3.3 Teste Híbrido (Caixa-Cinza)

A abordagem ou técnica de teste Caixa Cinza combina tanto informações estruturais quanto funcionais para realizar os testes [McMinn, 2004]. Funciona como um teste Caixa Preta, porém, o testador possui um conhecimento (limitado) sobre detalhes da implementação, ou sobre o algoritmo do software.

Este tipo de teste é bastante usado em aplicações que utilizam servidores como base de dados, além de sistemas que têm bases de dados como repositório de informação.

2.3.4 Teste Baseado em Falhas

A abordagem baseada em falhas (*fault-based testing*) usa informações de falhas que são encontradas freqüentemente em desenvolvimento de software (falhas comuns a qualquer sistema) e também de tipos de falhas específicas que o testador pode querer descobrir [Cavalcanti e Gaudel, 2007].

Existem duas técnicas que são os mais utilizados por esta abordagem, que são *Error Seeding* e *Mutation Testing*. *Error Seeding* é um técnica para estimar o número de falhas em um programa, através de inserções propositais de falhas [Knight e Ammann, 1985] [Offut e Hayes, 1996]. Nessa técnica o teste é avaliado baseado no número de falhas artificiais encontradas. *Mutation Testing* é uma técnica que parte do pressuposto de que um programa vai ser bem testado se todas as suas falhas simples forem detectadas e removidas [Offut et. al., 1996]. Essas falhas, assim como na técnica *Error Seeding*, também são inseridas no programa propositadamente, porém, em forma de operadores mutantes

(*mutation operators*). Um exemplo de um operador mutante seria mudar um operador *AND* por um operador *OR* em uma expressão condicional do programa. Cada mudança ou mutação criada por um operador mutante é codificada em uma versão mutante do programa original. Um mutante é "morto" por um caso de teste que o faz produzir saídas incorretas, ou seja, um caso de teste que detecte o mutante é considerado eficaz em achar falhas no programa, e outros casos de teste não precisarão detectar novamente o mesmo mutante.

A seção seguinte fala dos principais critérios que podem ser utilizados nas abordagens de teste apresentadas.

2.4 Análise de Teste Estrutural

Cada abordagem (ou técnica) de teste possui um ou mais critérios para verificar se os casos de teste utilizados estão realmente cobrindo os requisitos do teste em questão. Esta seção aponta alguns dos principais critérios utilizados pela abordagem Caixa Branca, que merece destaque neste trabalho.

2.4.1 Critérios de Análise

Como já visto, na abordagem Estrutural, os aspectos de implementação do software (código fonte) são fundamentais para a criação de casos de teste. O próprio termo "estrutural" é relacionado ao conhecimento da estrutura interna do software. Os primeiros critérios estruturais foram exclusivamente baseados em estruturas de fluxo de controle (*crontrol-flow structures*), dentre os quais, os mais conhecidos são *All-Nodes* (cobertura de linhas), *All-Edges* (cobertura de decisões) e *All-Paths* (cobertura de caminhos) [Myers, 1979].

Por volta da década de 1970, surgiram os critérios baseados em fluxo de dados (data-flow criteria), que requerem iterações em definições (declarações) de variáveis e utilizações das mesmas no código para serem exercitados [Herman, 1976] [Rapps e Weyuker, 1985]. Ainda existem os critérios baseados em complexidade (complexity based criteria), que usam informações sobre a complexidade do software para derivar o conjunto de requisitos dos testes [McCabe, 1976].

Atualmente os critérios mais utilizados por ferramentas de cobertura [Clover] [EMAM] [DjUnit] são os baseados em fluxo de controle (*crontrol-flow*), e por serem tão amplamente utilizados eles são normalmente citados como critérios de análise de cobertura de código.

Como o foco deste trabalho é na utilização da abordagem estrutural e na fase de testes de unidade, os principais critérios para esta combinação serão detalhados na próxima seção.

2.4.2 Análise de Cobertura de Código

Análise de cobertura de código é um critério muito usado em teste estrutural, e consiste em um processo que engloba três atividades principais: (1) encontrar áreas de um programa não exercitadas por um conjunto de casos de teste; (2) criar casos de teste adicionais para aumentar a cobertura; e (3) prover uma medida quantitativa de cobertura de código, que consiste em uma medida indireta de qualidade [Cornett, 1996]. Análise de cobertura não garante a qualidade do software sendo desenvolvido, e sim a qualidade dos conjuntos de teste que estão testando o software. Esta técnica é bastante útil na avaliação dos resultados de testes de unidade, que analisam diretamente o comportamento do código.

Diversas ferramentas, comerciais ou *open-source*, automatizam o processo de análise de cobertura de código, utilizando instrumentação de código. Instrumentação consiste em inserir fragmentos adicionais de código para computar resultados de cobertura [Kessis et al., 2005]. Estes fragmentos de código não interferem na realização da funcionalidade a que se destina o software, mas apenas monitoram a sua execução para determinado propósito. Algumas dessas ferramentas precisam ter acesso ao código fonte [Clover], a fim de instrumentá-lo e compilá-lo com as informações necessárias para capturar a cobertura após a execução do código. Outras ferramentas instrumentam o código binário diretamente com o mesmo propósito [Jcoverage], não utilizando o código fonte e nem precisando recompilá-lo.

Existem diversas métricas que podem ser utilizadas no processo de análise de cobertura. Cada métrica tem seus pontos fortes e fracos. As próximas seções falam sobre as

quatro métricas básicas e mais utilizadas (cobertura de linhas, de decisão, de condições e de caminhos).

Cobertura de Linhas (*Statement Coverage*)

Esta métrica reporta quando cada linha é executada (também chamada de cobertura de segmentos, de nós ou cobertura de blocos básicos) [Cornett, 1996]. A grande vantagem desta métrica é que ela pode ser aplicada diretamente a código objeto, isto é, ela não requer processamento de código fonte. Entretanto, alguns consideram esta métrica como tendo a mais fraca granularidade de cobertura, devido à sua insensibilidade a qualquer linha condicional ou de múltiplas condições [Agustin, 2003] [Kaner et al, 1999][Myers, 1979]. Por exemplo, considere o seguinte fragmento de código Java (Exemplo 2.1):

```
linha 1: int b = 0;
linha 2: if (condição)
linha 3: b = b + x;
linha 4: y = a / b;
```

Exemplo 2.1 - fragmento de código Java

Se não houver um caso de teste que torne a condição verdadeira, este código falha na linha quatro. Contudo, se a condição for verdadeira, a execução deste código obterá 100% de cobertura, sem falhar. Este é o principal problema desta métrica de cobertura, já que expressões condicionais IF são bastante comuns.

Cobertura de Decisões (*Decision / Branch Coverage*)

Esta medida, também conhecida como *branch coverage* ou cobertura de ramos / seções, reporta quando expressões booleanas testadas em estruturas de controle (como *if* e

while) retornam verdadeiro ou falso [Cornett, 1996]. Para tanto, toda a expressão booleana é avaliada, independente do resultado de suas sub-expressões, separadas por operadores lógicos.

A principal vantagem desta métrica é a simplicidade, e a ausência dos problemas presentes em cobertura de linhas. A desvantagem desta métrica é que ela ignora ramos dentro de expressões booleanas em determinadas situações de utilização de operadores lógicos. Por exemplo, considere o seguinte fragmento de código:

```
linha 1: if (a > b &&  (b  > c || isTrue(a-b))
linha 2:   a = b + c;
linha 3: else
linha 4:   a = b – c;
```

Exemplo 2.2 - fragmento de código Java

Neste exemplo, a expressão booleana retorna verdadeiro se a > b e b > c, e retorna falso se a <= b. Com isto, esta métrica consideraria este bloco de código completamente exercitado, obtendo 100% de cobertura, sem chamar a função *isTrue*.

Cobertura de Condições (*Condition Coverage*)

Esta é uma métrica similar à cobertura de decisões, porém, tem maior sensibilidade ao fluxo de controle, ou seja, capacidade de observá-lo melhor. Ao invés de tratar expressões booleanas de múltipla decisão separadas por operadores lógicos *and* ou *or* como uma única expressão booleana, cada combinação de sub-expressões é considerada como um teste isolado [Agustin, 2003].

Ao considerar o fragmento de código da seção 2.4.2, utilizando este tipo de métrica há 2^3 combinações de teste, pois existem 3 sub-expressões, e cada uma pode assumir 2 valores: verdadeiro e falso.

Cobertura de Caminhos (Path Coverage)

Um caminho (*path*) é uma seqüência única de *branches* do início de uma função ou método até o seu final. A métrica de cobertura de caminhos reporta quando cada um dos possíveis caminhos de cada função ou método foram percorridos [Cornett, 1996]. Esta métrica também é chamada de cobertura de predicados.

A cobertura de caminhos possui a vantagem de exigir um teste completo, porém, como toda métrica, também possui desvantagens. Uma delas se deve ao fato de que o número de caminhos é exponencial em relação ao número de *branches*. Por exemplo, um método que possui 10 condições tem 1024 caminhos para serem testados. A outra desvantagem é que alguns caminhos são impossíveis de serem exercitados quando há relação entre dados. Por exemplo, considere o seguinte fragmento de código:

```
linha 1: boolean maiorQue = maiorQue(a,b);
linha 2: if (maiorQue)
linha 3:    c = a;
linha 4: x = c + a;
linha 5: if (maiorQue)
linha 4:    d = a;
```

Exemplo 2.3 - fragmento de código Java

Utilizando cobertura de caminhos, este código contém 4 caminhos possíveis, enquanto na verdade apenas 2 são possíveis: maiorQue = true e maiorQue = false.

2.5 Técnicas de Automação de Testes Estruturais

A realização de testes, com o objetivo de alcançar alguns dos critérios mencionados anteriormente, pode ser facilitada pelo uso de determinadas técnicas. A utilização dessas técnicas, além de possibilitar o aumento da qualidade dos testes em questão, implica na

automação dos mesmos. Algumas das técnicas mais utilizadas são brevemente abordadas nesta seção.

2.5.1 Execução Simbólica

A geração de dados estruturais de forma estática é baseada na análise da estrutura interna de um programa, sem a necessidade que ele seja executado. Execução Simbólica [King, 1976], [McMinn, 2004], em contraste com a execução real de um programa, trata-se do processo de atribuir expressões a variáveis de um programa enquanto um caminho é seguido através da estrutura do código. Esta técnica é amplamente utilizada para montar restrições em termos de variáveis de entrada que contém as condições necessárias para passar por determinados caminhos.

Apesar de Execução Simbólica ter sido bastante utilizada, e ter a vantagem de não precisar realmente executar o programa para tentar descobrir os dados necessários para diversos caminhos da estrutura do código, esta técnica também possui desvantagens em relação às técnicas dinâmicas. Se o objetivo de um teste é a execução de uma linha específica, todos os caminhos que levam a execução para a linha em questão são explorados. Com a presença de loops no código isso se torna um problema, na medida em que o número de caminhos cresce bastante. Outros problemas desta técnica envolvem o uso de *arrays* e ponteiros, e a imprecisão do domínio de entradas encontrado, quando se comparado com a utilização de uma técnica de busca metaheurística.

Apesar de técnicas estáticas e dinâmicas serem mais utilizadas de forma isolada para geração automática de dados, elas não são excludentes, e uma pode complementar a outra, como já foi realizado no trabalho de Gupta et al. (1998).

2.5.2 Geração Manual e Automática de Dados de Teste

A alternativa à execução simbólica é a aplicação (de forma manual ou automática) de dados de entrada para executar o código sendo testado. O objetivo aqui é gerar dados de entrada para cobrir certos critérios estruturais ou funcionais de um programa.

A forma manual é a mais utilizada, porém tem a grande desvantagem de ser custosa (longo tempo de preparação dos dados, longo tempo de execução do software). A melhor alternativa é então gerar esses dados automaticamente.

A geração automática de dados pode ser aleatória ou por meio de determinadas técnicas criadas para procurar um bom conjunto de dados de teste. Entende-se um bom conjunto de dados como sendo aquele que cobre os critérios estabelecidos para o teste em questão, como atingir uma alta cobertura de código, como mostrado na seção 1.4.2. Algumas das técnicas usadas para essa busca são: variável alternativa (*alternating variable*) [Ferguson e Korel, 1996], relaxamento iterativo (*iterative relaxation*) [Gupta et. al., 1998], têmpera simulada (*simulated annealing*) [Clark et. al., 1998], algoritmos genéticos (Teste Evolutivo) [Michael e McGraw, 1998], geração baseada em regras (rule-based) [Deason et. al., 1991], entre outras. Algumas dessas técnicas, que usam heurísticas em suas buscas, são vistas com mais detalhes no capítulo 3, com mais atenção para Algoritmos Evolutivos.

2.6 Considerações Finais

Neste capítulo, introduzimos conceitos básicos da área de Teste de Software. Foi visto que a atividade de testes tem uma importância fundamental para que o software seja entregue com qualidade, e por este motivo esta atividade é dividida em fases que são executadas em paralelo às fases de um processo iterativo de desenvolvimento de software. Cada fase tem sua importância, e sua realização é fundamental para que a próxima fase seja também realizada de forma adequada.

Teste de Unidade é a primeira destas fases, e é executada pelo desenvolvedor do software logo após a codificação. É fundamental que uma boca cobertura seja atingida durante essa fase de Testes, para que certos problemas não sejam encontrados apenas nas demais fases, dificultando a sua localização e correção.

Além de ser divididos em fases, foi visto que os testes também podem ser executados usando diferentes abordagens: funcional ou caixa preta, estrutural ou caixa branca, caixa cinza e baseado em falhas. Em se tratando de testes estruturais, que é o foco deste trabalho, dentre os diversos critérios existentes para avaliar a qualidade dos testes, ou

seja, para verificar se os mesmos estão de acordo com seus requisitos, demos destaque às medidas de cobertura de código. Esse destaque se deve ao fato de que nosso trabalho adotou uma dessas medidas como critério de qualidade: a cobertura de decisões (*branch coverage*). Tal medida nos parece mais adequada por equilibrar bem duas características: eficiência e facilidade de implementação.

No próximo capítulo veremos como técnicas metaheurísticas podem ser e já foram utilizadas para auxiliar a realização automática das abordagens de teste descritas, além de ser introduzido o conceito de Teste Evolutivo, que é o foco deste trabalho.

3 Testes Evolutivos

Como mencionado anteriormente, teste é uma etapa fundamental para a qualidade de software. Contudo é um processo caro por consumir boa parte dos custos e esforços envolvidos no desenvolvimento do software. Assim, a utilização de técnicas e estratégias para diminuir os custos associados a cada fase de teste, bem como o tempo gasto, é de grande importância e relevância para o universo da ES.

A Engenharia de Software como um todo contém uma diversidade de problemas que possuem uma grande quantidade de soluções possíveis, de forma que encontrar a solução ideal é, em muitas situações, teoricamente impossível ou intratável na prática [Harman e Jones, 2001]. Essa característica torna as técnicas de otimização e busca fortes candidatas para encontrar soluções para os problemas do mundo da ES.

Teste de Software pode ser visto como um dos problemas da ES que necessitam de soluções para que suas técnicas sejam bem aplicadas e suas fases executadas da melhor forma possível. Existem algumas formas de agilizar o processo de Teste de Software, sem pecar pela falta de qualidade. Uma destas formas é a automação ou semi-automação dos testes, que transfere os esforços dos desenvolvedores e testadores da parte mecânica da construção de casos de teste para apenas a aplicação de seus *expertizes* na configuração de ferramentas, que fariam o trabalho para eles.

Uma contribuição relevante para esse processo de automação é a geração automática de dados de teste usando Algoritmos Evolutivos, também chamada de Teste Evolutivo [Baresel et al, 2004]. A técnica consiste em gerar dados de entrada para cobrir certos critérios estruturais ou funcionais de um programa, utilizando Algoritmos Evolutivos para realizar uma busca no espaço de possíveis entradas do sistema.

A Seção 3.1 descreve brevemente técnicas de busca meta-heurística que podem ser utilizadas para automação de teste de software, incluindo Subida na Encosta (Hill Climbing) e Têmpera Simulada (Simulated Annealing), além de Algoritmos Evolutivos. Na Seção 3.2, discutimos especificamente o tema de Teste Evolutivo, que une algoritmos evolutivos e Teste de Software para ajudar a otimizar a qualidade de determinadas fases do ciclo de testes. Alguns trabalhos relacionados à utilização destas técnicas para geração de

dados de testes também serão apresentados. Finalmente, na Seção 3.3, tecemos algumas considerações finais.

3.1 Técnicas de Otimização Baseadas em Busca Meta-heurística

Técnicas de busca meta-heurística são arcabouços de alto nível que usam heurísticas para achar soluções de problemas sem precisar realizar uma enumeração completa e exaustiva de um espaço de busca [McMinn, 2005]. A vantagem destas técnicas é de achar boas soluções para problemas classificados como NP-completo (NP-complete) ou NP-difícil (NP-hard), ou ainda problemas que possuam soluções algorítmicas não praticáveis, com um custo computacional considerável. Por serem arcabouços gerais, as técnicas metaheurísticas tratam-se de estratégias que podem ser adaptadas para diversos problemas específicos. Ao serem bem adaptadas para o Teste de Software, essas técnicas podem se tornar formas dinâmicas para a geração de dados para testes.

As técnicas de busca meta-heurística possuem dois aspectos básicos: um *espaço de busca*, que contém as possíveis soluções (ou *estados*) do problema; e uma *função objetivo* (ou função de aptidão), que avalia as soluções achadas, associando-as a "notas". Pode ser considerado ainda um terceiro aspecto, que são os *operadores de busca*. Uma técnica meta-heurística inicia sua busca com uma ou mais soluções do espaço de busca. Os operadores de busca têm a finalidade de iterativamente gerar novas soluções a partir das soluções atuais, até que alguma condição de parada seja atingida, possibilitando assim o funcionamento da busca. Nesse processo, a melhor solução encontrada na busca (i.e. solução com o valor mais alto de função objetivo) é retornada ao usuário. As próximas seções apresentam três técnicas de busca meta-heurística bastante difundidas na literatura.

3.1.1 Subida na Encosta (Hill Climbing)

O algoritmo de busca de Subida na Encosta (ou Hill Climbing) é simplesmente um laço repetitivo que busca de forma contínua soluções de valor crescente para a função objetivo, até encontrar um "pico" na função onde nenhum vizinho tenha valor mais alto [Russel e Novig, 2003]. A Figura 3.1 mostra uma visão de alto nível do algoritmo de

Subida na Encosta. Inicialmente, uma solução é escolhida de forma aleatória no espaço e é definida como ponto atual da busca. A cada passo do algoritmo, os operadores de busca são aplicados para gerar novas soluções a partir da solução atual. Esse conjunto de novas soluções é definido como a *vizinhança* da solução atual e a definição dos operadores de busca é dependente do problema em questão. Em cada laço do algoritmo, se uma solução da vizinhança tiver função objetivo maior que a solução atual, então a solução é definida como novo ponto atual da busca e um novo laço é iniciado. Caso contrário, se nenhuma solução da vizinhança for melhor que o ponto de busca atual então o algoritmo termina a sua execução, retornando a solução atual como melhor solução do problema.

O algoritmo de Subida na Encosta é simples de ser implementado e em geral converge para uma solução rapidamente. Contudo, o algoritmo apresenta a desvantagem de ser suscetível a cair em *ótimos locais*, i.e. soluções que são as melhores, considerando uma região localizada do espaço de busca, mas não são as melhores soluções globalmente. De fato, quando o algoritmo atinge uma solução ótima local, podemos considerar que ele converge prematuramente sem explorar boas regiões do espaço de busca. Existe uma variação do algoritmo que possibilita sair desta situação, através de uma série de execuções com reinício aleatório, onde cada execução parte de uma solução inicial diferente. Nessa variação, a solução final retornada é definida como a melhor solução local encontrada nas diferentes execuções do algoritmo. Isso não garante que a melhor solução global seja encontrada, mas aumenta essa probabilidade.

Selecionar uma solução inicial $s \in S$
Repetir
  Selecionar $s' \in N(s)$ onde funcaoObj(s') > funcaoObj(s)
  s ← s'
Até funcaoObj(s) ≥ funcaoObj(s'), $\forall s' \in N(s)$

Figura 3.1 - Descrição de alto nível do algoritmo de Subida na Encosta para um problema com espaço de busca S; conjunto de vizinhos N; e funcaoObj, sendo a função objetivo a ser maximizada [McMinn, 2005].

3.1.2 Têmpera Simulada (Simulated Annealing)

O algoritmo de Subida na Encosta é incompleto, pois nunca faz movimentos "encosta abaixo", ou seja, em direção a estados com valores mais baixos visando explorar novas regiões do espaço de busca com possíveis ótimos globais. Com esta característica, é fácil o algoritmo ficar preso em um ótimo ou máximo local. A variação do algoritmo com reinício aleatório pode ser mais eficaz, mas é extremamente ineficiente. O algoritmo de Têmpera Simulada tenta combinar a subida na encosta com um percurso aleatório que resulte de algum modo em eficiência e completeza [Russel e Novig, 2003].

A Figura 3.2 mostra uma visão de alto nível do algoritmo de Têmpera Simulada. Assim como na Subida de Encosta, em cada passo da Têmpera Simulada é gerada uma vizinhança a partir do ponto atual de busca. Enquanto que o algoritmo de Subida na Encosta somente aceita como novos pontos de busca as soluções da vizinhança com maior valor de função objetivo, o algoritmo de Têmpera Simulada aceita uma solução da vizinhança dependendo de dois fatores:

(1) a diferença entre a função objetivo da solução da vizinhança sendo considerada e o valor da função objetivo da solução atual: se a solução vizinha for melhor que a solução atual então a solução vizinha é aceita como ponto de busca (assim como a Subida de Encosta); caso contrário, se a solução vizinha for pior que a atual, o algoritmo pode aceitá-la com uma probabilidade que depende da diferença do valor da função objetivo observada. Nesse caso, soluções piores são aceitas com maior probabilidade se o valor da função objetivo da solução vizinha considerada não cair drasticamente;

(2) o tempo de execução do algoritmo: de forma que nas primeiras iterações do algoritmo, soluções piores são aceitas com maior probabilidade, visando uma maior exploração do espaço de busca. Nas iterações finais, quando se espera que a busca se encontre em uma boa região do espaço de busca, então soluções piores são aceitas com baixa probabilidade.

Assim, a idéia do algoritmo de Têmpera Simulada é possibilitar eventualmente a busca por pontos piores do espaço de busca, visando encontrar caminhos alternativos para as melhores regiões. Quando uma boa região é encontrada, o algoritmo diminui a

probabilidade de aceitação de soluções piores, visando refinar a busca na região e garantir a convergência do algoritmo.

O nome deste algoritmo é originário da analogia com o processo de temperar ou endurecer metais e vidros. Neste processo, o metal ou vidro é aquecido a altas temperaturas até seu ponto de derretimento e depois esfriado até se tornar sólido novamente, ou seja, o material passa por uma sucessão de estados com maior e menor grau de endurecimento. Isto ocorre várias vezes, até que o material seja misturado e fique em determinado estado desejado. A probabilidade de aceitação *p* de uma solução inferior muda ao decorrer da busca, e é calculada com a seguinte fórmula:

$$p = e^{\frac{\delta}{t}}$$

onde δ é a diferença do valor da função objetivo entre a solução atual e a solução vizinha sob análise, e *t* é um parâmetro de controle chamado de temperatura e dependo da iteração em que o algoritmo se encontra. Inicialmente a temperatura é alta, para que os movimentos sejam maiores no espaço de busca, e depois vai esfriando de acordo com um "plano de esfriamento". Se o esfriamento ocorre muito rápido, as chances de o algoritmo ficar preso em um máximo local são maiores, já que não estará explorando suficientemente o espaço de busca. Se o esfriamento for muito lento, o algoritmo explora mais regiões do espaço de busca, entretanto demora mais a convergir.

O Algoritmo da Figura 3.2 segue os seguintes passos: (1) inicialmente uma solução inicial (s) é selecionada de forma aleatória no espaço de busca, e a temperatura inicial (t) é estabelecida; (2) depois o algoritmo entra em um *loop* que seleciona aleatoriamente outra solução (s') pertencente ao conjunto de vizinhos (N), até que uma condição de parada seja atingida; (3) um novo laço é iniciado para cada solução considerada no espaço de busca; (4) se a diferença (Δe) entre os valores das funções objetivos da solução candidata e da solução atual for menor que zero ele adota a nova solução, caso contrário, verifica se vale a pena pega a solução com pior avaliação através da fórmula de probabilidade (p) explicada anteriormente; (5) o valor da temperatura (t) é decrementado a cada iteração do *loop* mais externo, para que, de acordo com o plano de esfriamento, haja cada vez menos probabilidade de se aceitar soluções com piores valores da função objetivo.

```
Selecionar uma solução inicial s ∈ S
Selecionar uma temperatura inicial t > 0
Repetir
    ti ← 0
        Repetir
        Selecionar s'∈ N(s) aleatoriamente
        Δe ← funcaoObj(s') – funcaoObj(s)
        Se Δe < 0
            s ← s'
        Senão
            Gerar número aleatório n tal que 0 ≤ n < 1
            Se n < e^(-δ/t)  Entao s ← s'
        Fim Se
        ti ← ti + 1
    Até ti = numSols
    Decrementar valor de t de acordo com o plano de esfriamento
Até a condição de parada ser atingida
```

Figura 3.2 - Descrição de alto nível do algoritmo de Têmpera Simulada para um problema com espaço de busca S; conjunto de vizinhos N; número de soluções numSols, consideradas a cada nível de temperatura t; e funcaoObj, sendo a função objetivo a ser maximizada [McMinn, 2005].

3.1.3 Algoritmos Evolutivos (*Evolutionary Algorithms*)

Algoritmos Evolutivos (AEs) são técnicas de busca meta-heurística baseadas na teoria da sobrevivência do mais apto, de Charles Darwin. Estas técnicas simulam a evolução como uma estratégia de busca para gerar soluções candidatas, usando operadores inspirados na genética e na seleção natural [McMinn, 2005]. Algoritmos Evolutivos podem ser eficazes para achar máximos locais de problemas complexos e não-contínuos que são muito difíceis de serem resolvidos [Rela, 2004].

Dentre os métodos que compõem o grupo de Algoritmos Evolutivos, podemos citar: Algoritmos Genéticos, Estratégias de Evolução, Algoritmos Culturais e Programação Genética [Mantere e Alander, 2005].

Algoritmos Genéticos são a forma mais conhecida de AEs, e foram inicialmente estudados e apresentados por Holland (1975). AGs serão explicados com mais detalhes na

próxima subseção. Programação Genética é uma técnica de aprendizagem de máquina utilizada para otimizar uma população de programas de acordo com uma função objetivo, sendo esta baseada em algum problema que os programas candidatos precisem resolver. Estratégias de Evolução (EE) trabalham com vetores de números reais como representação das soluções, e utiliza mutação e seleção como operadores. As taxas de mutação são normalmente auto-adaptativas. Algoritmos Culturais trata-se de um *framework* computacional que visa expressar diferentes modelos de evolução cultural, de forma que as razões para esta evolução possam ser isoladas e aproveitadas para resolver problemas computacionais de larga escala [Reynolds, 1994].

3.1.4 Algoritmos Genéticos

Algoritmos Genéticos (AGs) são técnicas de busca e otimização inspiradas em evolução, sendo o mais conhecido algoritmo evolutivo [Whitley, 1993]. Estes algoritmos são baseados no princípio da seleção natural e sobrevivência do mais apto. A Figura 3.3 mostra um algoritmo genético típico.

Os algoritmos genéticos trabalham com populações, onde cada indivíduo da população corresponde a uma solução no espaço de busca é representado por um *cromossomo*. O cromossomo é comumente representado por uma cadeia de bits, que são parte da solução do problema de otimização em questão. A representação de uma solução na forma de um cromossomo é dependente do problema.

```
Seja S(t) a população de cromossomos na geração t.

t ← 0
inicializar S(t)
avaliar S(t)
enquanto o critério de parada não for satisfeito faça
        t ← t+ 1
        selecionar S(t) a partir de S(t- 1)
        aplicar crossover sobre S(t)
        aplicar mutação sobre S(t)
        avaliar S(t)
fim enquanto
```

Figura 3.3 - Um Algoritmo Genético Típico [Lacerda e Carvalho, 1999]

O primeiro passo de um algoritmo genético, como é mostrado na Figura 3.3, é gerar uma população inicial (S) com N cromossomos, i.e. com N pontos do espaço de busca. Esta geração normalmente é aleatória, a não ser que exista conhecimento prévio do espaço de busca. Em seguida, cada cromossomo dessa população inicial é avaliado de acordo com uma função objetivo, e novas populações são evoluídas de forma iterativa, a cadê geração (t) através dos seguintes passos:

(1) Seleção: consiste em selecionar os melhores indivíduos da população atual S(t-1), ou seja, os que possuem mais alto valor de função objetivo. Na nomenclatura de AGs, a função objetivo é normalmente chamada de *função de aptidão* (ou *fitness*). Os indivíduos selecionados são armazenados em uma população intermediária.

(2) Cruzamento (ou *crossover*): uma nova população S(t) é gerada a partir do cruzamento dos indivíduos mais aptos selecionados na etapa anterior. Na operação de crossover, um par de indivíduos da população intermediária é selecionado por vez e seus cromossomos são combinados. Essa combinação é feita, em geral, escolhendo um ponto de separação aleatoriamente ao longo do tamanho de cada cromossomo. A Figura 3.4 ilustra a utilização deste operador. Na figura pode-se observar que cada cromossomo pai (representados por cadeias de bits) é cortado em um determinado ponto. Em seguida, a primeira parte do cromossomo Pai 1 é concatenada à segunda parte do cromossomo Pai 2, e a primeira parte do cromossomo Pai 2 é concatenada à segunda parte do cromossomo Pai 1, gerando dois novos cromossomos filhos [Duda, 2000]. O crossover é aplicado com uma dada probabilidade para cada par de cromossomos, sendo esta probabilidade chamada de *taxa de crossover*. Quando não ocorre crossover, os filhos são iguais aos pais e certas características são preservadas com isso. O operador de crossover é aplicado sucessivamente para diferentes pares de indivíduos selecionados até que uma nova população completa de indivíduos seja gerada.

(3) Mutação: onde cada bit em um cromossomo tem uma pequena chance de ser mudado de 1 para 0, ou vice-versa [Duda, 2000]. Essa operação visa aumentar a diversidade das soluções geradas de uma população para outra. O operador de mutação é também aplicado com uma dada probabilidade, a *taxa de mutação*. Para evitar uma

variação muito abrupta de uma população para outra, é recomendada a utilização de pequenas taxas de mutação, normalmente entre 0,1% e 5% [Lacerda e Carvalho, 1999]. A Figura 3.5 ilustra a utilização deste operador, onde alguns bits dos cromossomos filhos são modificados conforma a taxa de mutação.

(4) Re-avaliação: Os cromossomos da nova população S(t) são re-avaliados de acordo com a função de aptidão, para que uma nova população seja gerada a partir dos operadores citados anteriormente, até que uma condição de parada seja satisfeita.

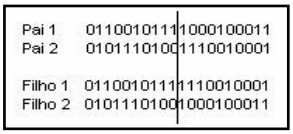

Figura 3.4 - Crossover

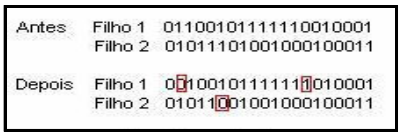

Figura 3.5 - Mutação

A idéia básica dos AGs é que novos pontos de busca sejam definidos, através da combinação de soluções bem sucedidas das populações anteriores, como ocorre na natureza. A taxa de mutação visa explorar novas regiões do espaço de busca e evitar convergência prematura. Outros operadores podem ser usados ainda, como o Elitismo [Dejong, 1975], que consiste em manter o melhor cromossomo de uma geração para outra durante a utilização do algoritmo, com a intenção de preservar a melhor solução encontrada até então.

3.2 Teste Evolutivo

Como explicado anteriormente, Teste Evolutivo (TE) é uma técnica de teste de software que usa Algoritmos Evolutivos para procurar por dados de teste que cumpram determinados objetivos de teste. A utilização da técnica de testes evolutivos por si só implica em uma automação ou semi-automação da fase de teste. Para testes de software utilizando AEs, o objetivo do teste precisa ser representado através de uma função de aptidão (*fitness*), para que os algoritmos possam ser utilizados para evoluir os dados de teste inicialmente gerados [Sthamer et al., 2002].

A conversão de problemas de teste em tarefas de otimização normalmente resulta em espaços de busca complexos, descontínuos e não-lineares. Métodos de busca pela vizinhança como Subida na Encosta não são apropriados nestes casos. Por outro lado, Algoritmos Evolutivos são empregados para resolver esse tipo de problema por serem robustos e eficientes, como verificado em trabalhos anteriores [Sthamer, 1996] [Tracey et al., 1998]. O sucesso da aplicação de AEs a testes deve-se à habilidade de produzir soluções eficazes para espaços de busca complexos e com muitas dimensões. A complexidade e o número de dimensões estão relacionados com o número de parâmetros de entradas da unidade, módulo ou sistema sendo testado.

A técnica de Teste Evolutivo pode ser dividida em 2 categorias diferentes: Teste Evolutivo Convencional (TEC) e Teste Evolutivo Orientado a Objetos (TEOO). O escopo do TE convencional é achar dados que sirvam como entrada para a unidade de software procedimental sendo testada. A Figura 3.6, que foi inspirada no trabalho de [Sthamer et al., 2002], mostra o funcionamento de TEC, que possui os seguintes passos: (1) o conjunto inicial de dados é normalmente gerado de forma aleatória, porém, também é possível gerar este conjunto aproveitando algum conhecimento prévio que o testador possua do sistema sendo testado; (2) cada indivíduo da população, representado por um conjunto de dados de teste, é executado (ciclo interno da Figura 3.6); (3) para cada conjunto de dados de teste a execução é monitorada e o valor de aptidão é determinado para o indivíduo correspondente; (4) os dados de teste com maiores valores de aptidão são selecionados com maior probabilidade que os demais e são sujeitos aos operadores de cruzamento (*crossover*) e mutação, para que uma nova população de dados de teste seja gerada (ciclo externo da Figura 3.6. Após a geração da nova população o processo se repete até que determinado objetivo do teste seja atingido ou alguma outra condição de parada seja satisfeita, como o número máximo de gerações. A figura 3.6, inspirada no trabalho de [Sthamer et al., 2002], mostra o funcionamento de TE convencional.

Quando se utiliza Teste Evolutivo OO, a busca visa produzir programas de teste completos (casos de teste) uma vez que dados de entrada apenas não são suficientes para executar o teste. Um caso de teste precisa também descrever como criar os objetos participando do teste e como colocá-los em estado apropriado para que o objetivo do teste seja atingido [Wappler e Lammermann, 2005]. Isso implica que a representação do

cromossomo de um indivíduo não será apenas uma seqüência de dados numéricos e caracteres, mas também uma seqüência de chamadas a construtores e métodos. A figura 3.7, inspirada no trabalho de [Wappler e Lammermann, 2005], mostra o funcionamento de TE OO. A única diferença desta abordagem de TE para a abordagem convencional é o ciclo interno, que possui os seguintes passos: (1) os indivíduos gerados inicialmente de forma aleatória em alguma representação, são transformados em programas de teste completos; (2) cada programa de teste é executado e monitorado, e estes passam por avaliações para calculo de suas aptidões; (3) os indivíduos sofrem uma decodificação de programas de testes para a representação anterior, que os AEs entendam e possam aplicar seus operadores baseados na evolução.

A abordagem orientada a objetos possui a mesma essência da convencional, porém é mais complexa na medida que um processo de conversão precisa ser estabelecido entre a execução dos testes e a execução do algoritmo evolutivo.

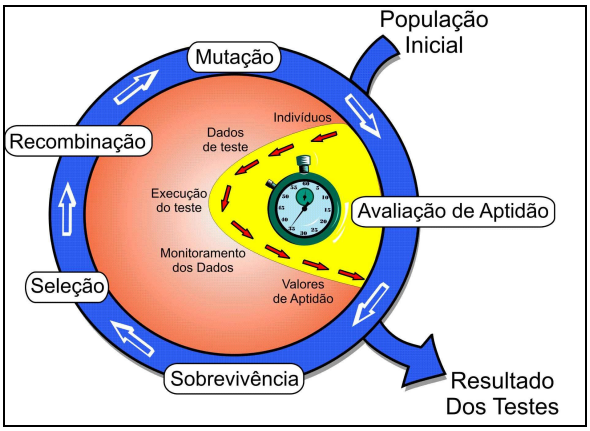

Figura 3.6 - Visão Geral de Teste Evolutivo Convencional.

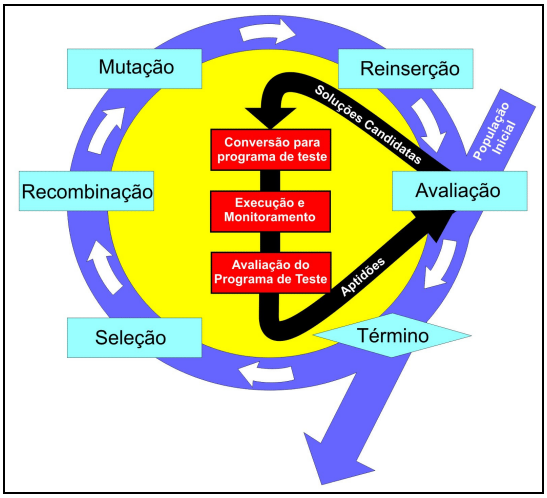

Figura 3.7 - Visão Geral de Teste Evolutivo OO.

As próximas seções apresentam detalhes de TE conforme o tipo de teste abordado. A seção 3.2.1 aborda os testes evolutivos estruturais, e a seção 3.2.2 fala sobre testes evolutivos não-estruturais. Por fim, a seção 3.2.3 apresenta uma análise comparativa entre os trabalhos relacionados.

3.2.1 Teste Evolutivo Estrutural

Esta seção apresenta trabalhos anteriores que utilizaram Algoritmos Evolutivos para gerar dados automaticamente para testes caixa-branca, ou seja, testes que possuem acesso ao código fonte e podem utilizar critérios de cobertura para definir a função objetivo. Os trabalhos estão separados em duas categorias: utilização de AEs em software procedimental, ou seja, aqueles que foram escritos em linguagens estruturadas como C e Ada e são baseados em funções; e a utilização de AEs em software Orientado a Objetos, como os escritos em Java, C++ e baseados em classes. Esta separação dentro de testes estruturais deve-se às diferenças entre TE convencional e TE OO, explicadas anteriormente.

(a) Software Procedimental

Sthamer (1996) realizou um trabalho de investigação de teste randômico e de algoritmos genéticos como meio prático de gerar conjuntos de dados para testes automaticamente. Ele criou programas em ADA para exemplificar a utilização de AGs, e verificou que esses algoritmos mostraram bons resultados na busca de entradas para conjuntos de teste. Os experimentos com AGs atingiram 100% de cobertura de decisões, com um número menor de casos de teste, superando em muito a técnica de geração randômica para o mesmo propósito. O trabalho também investigou aspectos relacionados aos AGs como: qual a melhor forma de representação, qual a estratégia de evolução mais eficiente, qual estratégia de cruzamento mais eficiente, qual tamanho ideal da população e a taxa ideal de mutação.

Harman et al. (2002) combinaram técnicas de teste evolutivo e técnicas de transformação de código, visando aumentar a cobertura. As técnicas de transformação de código modificavam programas que possuem *flags* (variáveis booleanas) representando expressões condicionais, transformando-os em programas livres de *flags*. Isso foi pensado devido ao fato de que a utilização de *flags*, em substituição às expressões condicionais tradicionais, interferem na aplicação de funções objetivo baseadas nos predicados dessas expressões. Utilizando a técnica de transformação do código antes da geração dos dados com AGs, uma melhor cobertura do código foi atingida, além de uma diminuição no número de gerações necessárias para atingir a cobertura máxima. Um trabalho semelhante foi realizado por Baresel et al. (2004), utilizando uma abordagem de transformação de códigos com flags atribuídos dentro de loops. A técnica de transformação combinada aos algoritmos evolucionários obteve bons resultados como no trabalho anterior, porém o código transformado não preservava a equivalência funcional do código original.

McMinn (2005) propõe uma abordagem estendida de geração de dados para testes estruturais, que permite a geração de seqüências de entrada para se colocar um objeto de teste em um estado necessário para atingir determinado objetivo do teste. Este trabalho é baseado na abordagem encadeada (*chaining approach*), cuja idéia é achar uma seqüência de linhas de código (*statements*), que envolvem variáveis internas e que precisam ser

executadas para atingir o objetivo do teste. Ele concluiu que a utilização dessa abordagem estendida funcionou melhor do que a convencional para geração de dados para testes.

(b) Software OO

Tonella (2004) utiliza um AG para produzir automaticamente dados para testes unitários de classes em um cenário genérico. Os dados de teste, representados por cromossomos no AG, incluem informações de quais objetos instanciar, quais métodos invocar e quais valores usar como entradas para os métodos. O método proposto neste trabalho foi implementado em uma ferramenta chamada *eToc* e foi aplicado com sucesso em testes de unidade de classes da biblioteca padrão de Java. Tonella concluiu em seu trabalho que a utilização de AGs para testes unitários de classes é extremamente poderosa, tendo sido obtida uma ótima cobertura dos *branches* dos métodos públicos das classes testadas, além de um tempo computacional razoável.

Wappler e Lammermann (2005) utilizaram uma nova abordagem para aplicar algoritmos evolutivos na geração automática de casos de teste para testes caixa branca de software orientado a objetos. Os autores argumentam que, em contraste com os testes evolutivos convencionais, o escopo de teste evolucionário OO não se resume apenas a achar dados de teste que sirvam de entrada para a unidade sendo testada. Essa busca tem também o objetivo de produzir programas de teste completos. Um caso de teste também precisa descrever como criar os objetos que participam do teste e como colocá-los no estado apropriado para que o objetivo do teste seja atingido. A função objetivo deve ser construída levando em consideração estes novos aspectos. A abordagem utiliza algoritmos evolutivos, providos por *toolboxes* populares e independentes do domínio da aplicação. Para utilizar estes algoritmos, uma codificação foi definida para representar os casos de teste como estruturas contendo valores de tipos básicos de dados. Um protótipo foi implementado para validar a abordagem, e foi comparado com teste aleatório (*random test*) através de experimentos. Nesses testes, algoritmos evolutivos puderam ser aplicados com sucesso a testes caixa branca de softwares OO, além de terem superado o teste aleatório em todos os aspectos.

Cheon et al. (2005) afirmam em seu trabalho que teste de software deveria ser menos custoso, consumir menos tempo e ser mais automatizado. Para tentar atingir esse objetivo eles propuseram uma abordagem para automatizar completamente testes de unidade para programas Java, e para tanto foi preciso automatizar três componentes de teste: a seleção de dados de teste, a execução do teste e a comparação dos resultados da execução com os resultados esperados (*test oracle*). A essência da abordagem deste trabalho é combinar JML (Java Modeling Language), que é uma linguagem formal de especificação comportamental de interfaces, com Algoritmos Genéticos. Para completar, o JUnit é usado como plataforma de execução de testes. No trabalho, JML é usada como ferramenta para descrever *test oracles* e como base para geração de dados de teste. Cada classe a ser testada precisa conter afirmações (*assertions*) em JML, como pré-condições, pós-condições e constantes que descrevam o comportamento da classe. JML possui um componente chamado *runtime assertion checker* que é usado como detector de violações dos *assertions* em tempo de execução, além de interpretá-los como sucesso ou fracasso do teste. Para isso, uma classe de teste *JUnit*, chamada pelo sistema de *test oracle*, é gerada automaticamente.

A automação da geração de dados em Cheon et al. (2005) é realizada usando AGs. O conjunto inicial de dados de teste é gerado aleatoriamente baseado nas assinaturas dos métodos das classes. O valor de aptidão de cada dado indivíduo é calculado baseado em cobertura de caixa-preta (cobertura de condições em pós-condições) ou cobertura de caixa-branca (cobertura de decisões ou *branches*). O conjunto de dados é filtrado usando as pré-condições para eliminar os testes chamados de insignificantes. Se o conjunto de dados satisfaz os critérios de cobertura, o conjunto ideal foi achado, do contrário, a população de dados de teste é enriquecida pela aplicação dos operadores genéticos de cruzamento e mutação, resultado na geração de uma nova população. O processo evolutivo é repetido até que o conjunto apropriado de dados seja achado.

3.2.2 Teste Evolutivo não Estrutural

Esta seção apresenta trabalhos que procuraram gerar dados usando AEs para realizar testes não estruturais, ou seja, que não dependem do código fonte e são executados

diretamente em uma versão do software completo. Teste caixa-preta, caixa-cinza e testes para achar o pior e o melhor tempo de execução de um sistema estão incluídos nesta categoria.

Testes Funcionais (Caixa-preta)

Buehler e Weneger (2003) apresentaram a aplicação de testes evolutivos nos testes funcionais (caixa preta) de um sistema de estacionamento automático de carros. O trabalho já se diferencia dos outros por ter o foco em testes funcionais diferente da maioria dos outros trabalhos. A função de aptidão (*fitness*) utilizada pelo AG foi criada de forma a associar bons valores de aptidão para dados que levassem o sistema a fazer o veículo entrar na área de colisão, ou terminar em uma posição inadequada de estacionamento. Em contrapartida, valores ruins de aptidão são associados a conjuntos de dados que levam o sistema a encontrar boas posições de estacionamento. Os resultados do desenvolvimento de testes funcionais evolutivos foram promissores. Vários cenários de estacionamento foram identificados automaticamente com experimentos iniciais, sendo possível identificar e corrigir erros no sistema com esta abordagem.

Outro trabalho, realizado por Buehler e Weneger (2005), mostra um ambiente de teste evolucionário funcional de funções de auxílio a freios de automóveis. Um assistente de freios (brake assistant) é uma função de auxílio de veículos cujo objetivo é ajudar motoristas em situações críticas para diminuir a velocidade ou parar seus veículos de forma rápida e efetiva. Para a construção da função objetivo, foram utilizadas duas quantidades: tempo-para-colisão (*time-to-collision*), que considera a distância entre veículos e a diferença da velocidade entre eles; e o momento do freio (*brake momentum*), que é a soma do momento de freio requisitado pelo motorista e o momento de freio adicionado pelo sistema assistente do freio. O ambiente de teste montado foi capaz de identificar automaticamente situações em que o comportamento do sistema se desviou dos requisitos funcionais definidos. Concluiu-se que a implementação utilizada para a simulação continha erros que seriam difíceis de achar utilizando técnicas convencionais de teste.

Baresel et al. (2003) apresentam uma abordagem de teste evolutivo seqüencial tanto para testes caixa branca (estruturais), quanto para testes caixa preta (funcionais). Para testes

evolutivos funcionais, o autor afirma que os sistemas não devem ser testados por apenas alguns passos de simulação, e sim por seqüências de entrada que sejam longas e possuam as qualidades necessárias para estimular o sistema adequadamente. Além disso, as seqüências de saída devem ser avaliadas considerando diferentes condições. Um ambiente implementado em Matlab foi usado para testar modelos de fluxo de estados. Os resultados dos experimentos realizados mostraram que é possível achar seqüências de testes, sem a necessidade de interação com o usuário, para problemas que anteriormente não podiam ser resolvidos automaticamente.

Testes Temporais

Conhecer o pior caso de tempo de execução é um requisito essencial para o funcionamento seguro e correto de um sistema de tempo real. Em seu trabalho, Gross (2001) propôs a aplicação de uma análise evolutiva de casos de pior tempo, substituindo técnicas de análise estática. A idéia é utilizar este tipo de análise para gerar dados que maximizem o tempo de execução do sistema considerado. A abordagem foi comparada com testes randômicos e com o desempenho de um testador humano, que conhecia o sistema sendo testado e fez uma análise manual para tentar identificar a execução de maior tempo. Teste randômico é a técnica de validação de propriedades de tempo mais importante utilizada na indústria, porém, os experimentos realizados em Gross (2001) demonstraram que teste evolutivo foi mais eficaz para achar tempos de execução longos. Os testes randômicos conseguiram produzir apenas 85% dos tempos de execução encontrados pelo teste evolutivo. O autor afirma também que, para certos programas críticos, um testador humano pode obter um melhor desempenho em achar os casos de teste necessários, e direcionar a execução por *branches* mais longos. O autor destaca que a estratégia ideal seria a combinação de teste evolutivo com o suporte do conhecimento humano do objeto sendo testado.

ERTT (Evolutionary Real-Time Testing) é uma abordagem para testar as restrições de tempo de sistemas de tempo real [Tlili et al., 2006]. Um algoritmo evolutivo é utilizado para maximizar ou minimizar o tempo de execução de um sistema, encontrando o pior caso ou o melhor caso. Tlili et al. (2006) argumentam em seu trabalho que os resultados de

ERTT nem sempre são confiáveis. Diferentes execuções do mesmo teste resultam em tempos de execução diferentes. Além disso, o número de gerações necessárias para achar os maiores tempos de execução é normalmente alto, até para objetos de teste simples. Diante disso, os autores propõem dois métodos para achar os maiores e menores tempos de execução com menos números de gerações e de uma forma mais robusta. O primeiro método é baseado na exploração de novas regiões do espaço de busca com a ajuda de um algoritmo evolutivo para guiar os dados estruturais de teste. O segundo método é baseado na restrição do domínio das variáveis de entrada e das variáveis globais na população inicial, para que o processo de otimização leve a regiões do espaço de busca onde o domínio das variáveis é realmente usado. Como conseqüência da utilização deste método, mais código é executado, e assim a confiança nos resultados fornecidos pelo ERTT é reforçada. Os autores afirmam que os resultados dos experimentos conduzidos confirmaram as expectativas de que os métodos propostos são capazes de encontrar tempos de execução, de forma mais robusta e eficiente, em comparação com a configuração clássica de ERTT.

3.2.3 Análise Comparativa

Esta seção tem o objetivo de fazer uma análise dos trabalhos resumidos acima, que utilizaram Algoritmos Evolutivos para automatizar tanto testes estruturais quanto funcionais, e suas diversas abordagens. A análise tem o foco na observação das técnicas e critérios utilizados, bem como dos resultados de cada trabalho. As tabelas 3.7 e 3.8 mostram um resumo dos aspectos relevantes para a análise, e tratam separadamente de Teste Evolutivo Convencional e Orientado a Objetos, respectivamente. A análise visa identificar quais dos trabalhos anteriores possuem aspectos em comum e podem ser posteriormente comparados com o trabalho aqui desenvolvido.

Quadro 3.1: Características dos trabalhos relacionados de TEC citados na seção 3.6

Autores da ferramenta de TE	**Função Objetivo**	**Técnica de Teste**	**Critérios para Avaliação**	**Resultados**
Sthamer (1996)	Uma função (Recíproca, Gaussiana ou Distância *Hamming*) para cada predicado (condição) que se quer atingir em uma subrotina	Caixa-Branca	Cobertura de condições (*branch*) e número de testes gerados	100% de cobertura e menos testes gerados em comparação com teste aleatório
Gross (2001)	A função mede o tempo de execução do objeto de teste para uma situação particular de dados de entrada.	Caixa-Branca	O tempo de execução em micro-segundos do código sendo testado. O AE tenta maximizar este tempo.	O TE superou Teste Aleatório em todos os casos (15 módulos) de busca do pior tempo de execução. Em 4 dos 15 módulos, o teste manual superou TE, devido ao conhecimento prévio do testador a respeito dos códigos.
Harman (2002)	Uma função para cada predicado (condição) que se quer atingir em um programa (*Triangle Classification Program* e *Calendar Program*).	Caixa-Branca	A Cobertura de condições (b*ranch*) e numero de avaliações de aptidão (fitness).	Com a abordagem de transformação utilizada (remoção de flags) uma cobertura maior é conseguida (100%) e com menos avaliações de *fitness* que a versão com flags.
Baresel (2003)	Para os testes estruturais a função analisa todos os caminhos de execução de uma seqüência de casos de teste (dados de entrada) e usa o caminho mais próximo ao alvo como aptidão (*fitness*) da seqüência.	Caixa-Branca e Caixa-Preta	Cobertura de linhas (*statement*), cobertura de decisões (*branch*) e cobertura de condições para os testes estruturais.	De 3 funções testadas, as 3 coberturas analisadas variaram de 91% a 99%.
Buehler e Weneger (2003)	O valor da menor distância entre o carro e a área de colisão.	Caixa-Preta	Coordenadas do carro muito próximas à zona de colisão.	Em aproximadamente 900 cenários simulados, em mais de 25 houve colisão. Uma falha foi detectada no ambiente de simulação.

Buehler e Weneger (2005)	A função trata-se de uma fórmula que combina o tempo-para-colisão (time-to-collision) e o momento do freio (brake momentum).	Caixa-Preta	O tempo que o freio inicia, a força inicial do freio, o tempo que o assistente de freio começa a agir, a força que o assistente adiciona ao freio.	Situações foram identificadas em que o comportamento do sistema se desviou dos requisitos funcionais definidos
Buehler e Weneger (2005)	A função trata-se de uma fórmula que combina o tempo-para-colisão (time-to-collision) e o momento do freio (brake momentum).	Caixa-Preta	O tempo que o freio inicia, a força inicial do freio, o tempo que o assistente de freio começa a agir, a força que o assistente adiciona ao freio.	Situações foram identificadas em que o comportamento do sistema se desviou dos requisitos funcionais definidos
McMinn (2005)	A função da abordagem híbrida de McMinn utiliza a distancia para o nó alvo combinado com a distancia entre cada predicado de uma condição (*branch*).	Caixa-Branca	Taxa de sucesso, cobertura (b*ranch*), número de avaliações, tempo médio e tempo máximo de das buscas.	A abordagem estendida atingiu 100% de cobertura para 7 dos 9 objetos de teste, superando as abordagens evolutivas padrão e a seqüencial.
Tlili et al. (2006)	A função trata-se apenas do tempo de execução de um conjunto de dados (indivíduos), apesar de que o também computa dados estruturais (cobertura).	Caixa-Branca e Caixa-Preta	Tempo de execução e Cobertura de decisões (*branch*), já que mais código executado acarreta em maior tempo de execução.	Utilizando o método proposto no trabalho (restrição do domínio das variáveis de entrada da população inicial) maiores e menores de execução foram achados, superando a forma clássica de Teste Evolutivo de Tempo Real.

.

Quadro 3.2: Características dos trabalhos relacionados de TEOO citados na seção 3.6.

Ferramenta de Teste	**Função Objetivo**	**Técnica de Teste**	**Critérios para Avaliação**	**Resultados**
Tonella (2004)	Função que da a nota dos indivíduos de acordo com o número de nós de controle dependentes para se chegar ao alvo.	Caixa-Branca	Cobertura de condições (*branch coverage*), número de test cases gerados, número de execuções (gerações), o tempo total em segundos e falhas ou erros incluídos propositalmente nas classes sendo testadas.	100% de cobertura em 2 das 7 classes testadas da biblioteca padrão de Java, devido a nós alvos muito difíceis de serem atingidos. A execução dos casos de teste e a inserção manual de *assertions* identificaram um erro na classe *LinkedList*. Tempos de execução variados de acordo com o tamanho e complexidade da classe.
Wappler e Lammermann (2005)	A função avalia primeiramente se um indivíduo pode ser decodificado em um programa, associando notas de acordo com a quantidade de erros. Em segundo lugar, para os indivíduos decodificados com sucesso, é aplicada uma combinação de nível de aproximação do nó alvo e distância condicional.	Caixa-Branca	O alcance da linha alvo, e as medidas de erro utilizadas para evoluir os indivíduos "inconvertíveis" , ou seja, os que não puderam ser decodificados para programas após a evolução do Algoritmo Evolutivo.	Um objeto de teste (classe Java) foi utilizado com o objetivo de se alcançar uma linha (alvo) de um único método. O alcance desta linha depende que várias condições sejam antes verdadeiras. O teste aleatório não conseguiu atingir o alvo. A abordagem evolutiva conseguiu em 14 gerações.
Cheon et al. (2005)	Ainda não existe função objetivo, pois a parte Evolutiva da ferramenta ainda não tinha sido realizada na época em que o artigo foi escrito.	Caixa-Branca	Por enquanto a ferramenta, que não se encontrava pronta na época do artigo, observava cobertura de condições das pós-condições, especificadas através de JML em cada método da classe sendo testada.	Ainda não havia resultados na época em que o artigo foi escrito.

Através das informações descritas nas tabelas anteriores, pode-se observar que os dados mais utilizados para compor a função objetivo, ou de aptidão (*fitness*) no caso de um Algoritmo Evolutivo, são: a distância do último nó (condição) executado para o nó alvo, a distancia entre os predicados de uma condição (nó alvo) e o tempo de execução, sendo tudo isso para um determinado conjunto de dados de entrada ou caso de teste. A técnica de teste varia entre caixa-preta e caixa-branca, sendo teste estrutural mais comum para lidar com Algoritmos Evolutivos, por causa da maior quantidade de métricas disponíveis para montar a função objetivo. Cobertura de condições e tempo de execução são os objetivos mais comuns de teste, por se tratarem de medidas indiretas da qualidade de um software, e os resultados são também expostos, em sua maioria, em função destas duas medidas.

3.3 Considerações Finais

Existem diferentes formas de diminuir os custos e o acelerar a elaboração de um software, e a automação de determinadas fases de teste com a geração automática de dados ou programas de teste é uma técnica amplamente utilizada nas últimas décadas. Neste capítulo, vimos técnicas metaheurísticas que são dinâmicas e podem ser adaptadas para diferentes objetivos de teste. Os algoritmos de Subida na Encosta (*Hill Climbing*) e Têmpera Simulada (*Simulated Annealing*) foram brevemente discutidos e Algoritmos Evolutivos foram apresentados como o *framework* mais eficiente para realizar esta forma de automação de testes, com destaque para Algoritmos Genéticos.

A principal técnica exposta no capítulo foi Teste Evolutivo, que visa unir algum Algoritmo Evolutivo e a geração de dados para testes. A utilização desta técnica por si só implica em uma automação ou semi-automação do processo de testes, e os resultados obtidos podem ser muito satisfatórios, como comprovado em diversos trabalhos anteriores. Foi visto que TE estrutural pode se dividir em 2 principais categorias: TE Convencional e TE Orientado a Objetos. A primeira visa apenas geração automática de dados de entrada, enquanto a segunda tem a necessidade de gerar programas completos de teste, e ambas têm o objetivo exercitar todos os caminhos de uma classe ou medir o tempo máximo de uma aplicação. Vários trabalhos relacionados foram apresentados, categorizados por técnica de teste (estrutural, funcional e temporal), e abordagem de TE, em se tratando de teste

estrutural. Diante da análise destes trabalhos, verificou-se que as funções objetivo mais observadas durante os testes são cobertura de código (mais especificamente cobertura condicional) e o tempo de execução de determinado.

No próximo capítulo, será apresentada uma ferramenta chamada EvolUniT, que foi implementada para confirmar a eficácia de Teste Evolutivo para geração de testes de unidade, em uma abordagem híbrida que utiliza características da abordagem Convencional e da abordagem Orientada a Objetos.

4 EvolUniT: Aplicando Algoritmos Genéticos na Geração de Testes de Unidade Java

Testes extensivos só podem ser realizados através de uma automação do processo de teste [Staknis, 1990]. Como visto anteriormente, os benefícios alcançados com a automação incluem a diminuição dos custos dos testes e, como conseqüência, de todo o processo de desenvolvimento do software.

Ferramentas de análise estática analisam o software que está sendo testado sem executar o seu código, tanto de forma manual quanto automática. Ferramentas de Execução Simbólica são o exemplo mais amplamente conhecido de análise estática para geração de dados de teste. Diversos trabalhos já foram realizados utilizando esta técnica para automação na geração de dados [Ramamoorthy, 1976] [Howden, 1977], porém é uma técnica limitada para programas contendo referências a *arrays*, variáveis de tipo ponteiro e outras estruturas dinâmicas. Além disso, Execução Simbólica requer muito esforço computacional, diante da necessidade de derivar cada expressão associada a um caminho, já que os valores de cada variável são mantidos como expressões algébricas em termos de nomes simbólicos [King, 1976].

Diante das vantagens do Teste Evolutivo como uma técnica poderosa para geração de dados, ou de casos de teste completos no que diz respeito à POO, uma ferramenta foi implementada em Java com três principais objetivos: construir uma nova forma de Teste Evolutivo, que visa utilizar características de Teste Evolutivo Convencional e Orientado a Objetos; montar uma ferramenta que proporciona uma semi-automação e que de fato diminua os esforços de engenheiros de software na fase de testes de unidade; e comparar esta nova forma de Teste Evolutivo com Teste Aleatório, como já foi feito realizado em alguns trabalhos anteriores [Sthamer, 1996] [Gross, 2001] [Wappler e Lammermann, 2005], e também com outras abordagens de teste. A utilização da abordagem hibrida de Teste Evolutivo tem o objetivo de validar a utilização da abordagem convencional, que se trata de uma abordagem de mais fácil implementação, em um ambiente orientado a objetos. Com essa combinação esperam-se bons resultados de cobertura com os testes gerados, e a ausência da complexidade presente na abordagem OO de TE.

Esse capítulo será apresentado como se segue. Na Seção 4.1 é dada uma visão geral do funcionamento da ferramenta implementada, chamada de EvolUniT (Evolutionary Unit Testing), incluindo sua arquitetura. Em seguida, na Seção 4.2, os módulos principais da ferramenta serão descritos de forma detalhada. Na Seção 4.3, a utilização do *plug-in* no ambiente Eclipse é demonstrado, através de um exemplo. Finalmente, o capítulo é concluído na Seção 4.4.

4.1 Visão Geral do EvolUniT

Após o estudo e análise da teoria de Teste Evolutivo e da aplicação desta técnica e suas abordagens em vários trabalhos anteriores, este trabalho propõe uma nova forma de realização de Teste Evolutivo e a implementação de uma ferramenta para testar sua eficácia. Esta ferramenta foi implementada em Java, em forma de *plug-in* do eclipse, já que um dos objetivos da mesma é de poupar esforços aos engenheiros de software. Uma vez presente no próprio ambiente de desenvolvimento de um engenheiro de software, além da vantagem da semi-automação dos testes, existe a comodidade de não precisar mudar de ferramenta para geração e evolução dos casos de teste. A ferramenta proporciona uma semi-automação, ao invés de automação completa, porque o engenheiro poderá ter que complementar os casos de teste gerados, com possíveis pré-condições de testes, para que o objetivo do teste seja atingido. A vantagem desta semi-automação é que o conhecimento do desenvolvedor, das estruturas dos códigos sendo testados, será acrescido ao produto final da ferramenta, melhorando assim a eficácia dos casos de teste.

O *plug-in* criado chama-se EvolUniT (*Evolutionary Unit Testing*) e, como o próprio nome diz, tem foco apenas em testes de unidade, que normalmente são executados pelos próprios desenvolvedores do software. A Figura 4.1 mostra a arquitetura de alto nível da ferramenta, mostrando os módulos responsáveis pelo fluxo de entradas e saídas. Tendo em vista os 3 objetivos citados anteriormente, a ferramenta possui o seguinte fluxo padrão: (1) recebe como entrada uma classe Java a ser testada (Java CUT – *Class Under Test*); (2) gera para esta classe 2 outras classes, sendo uma de teste (utilizando o *framework JUnit*) e outra para chamar o caso de teste quantas vezes forem necessárias; (3) gera dados (parâmetros de construtores e métodos) inicialmente aleatórios para compor os casos de teste; (4) gera uma versão modificada da classe Java sendo testada e a executa para cada conjunto de dados,

capturando os valores de cobertura de cada execução; (5) utiliza um algoritmo genético para evoluir apenas os dados gerados, de acordo com uma função objetivo criada com base nas coberturas capturadas pelas execuções monitoradas; (6) gera novos conjuntos de dados evoluídos, e reinicia o ciclo até que o número máximo de gerações do AG seja atingido ou que uma cobertura máxima pré-definida seja atingida.

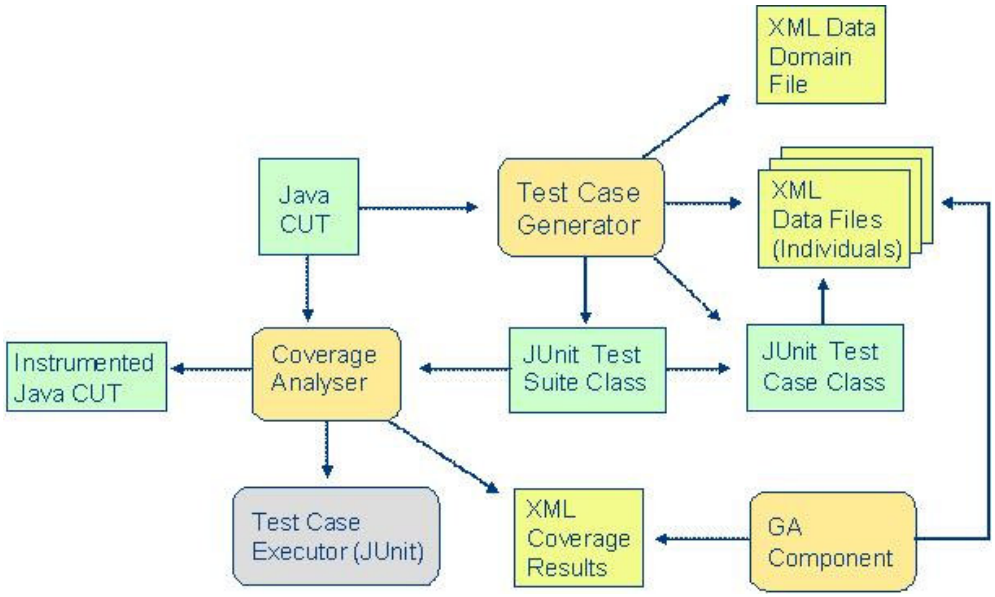

Figura 4.1: Arquitetura de alto nível do EvolUniT.

As próximas subseções explicam de forma detalhada o papel de cada módulo na ferramenta e o que cada um recebe como entrada, bem como em que formato gera suas saídas.

4.2 Principais Módulos do EvolUniT

Para cumprir com os seus objetivos, o EvolUniT é composto por 3 principais módulos, destacados em cor laranja na Figura 4.1., chamados de*: Test Case Generator*, *Coverage Analyser* e *GA Component*. O *Test Case Generator* é responsável por gerar todos os casos de teste de cada geração. O *Coverage Analyser* tem o objetivo de executar a classe sendo testada e monitorar esta execução, para gerar um relatório de cobertura. O terceiro e

último módulo da ferramenta, porém o mais importante, é responsável por evoluir os casos de teste (indivíduos) de cada geração, porém apenas no que diz respeito aos dados (parâmetros de entrada de métodos e construtores).

Cada um destes módulos tem um objetivo específico e será analisado de forma isolada, para que se entenda como de forma integrada eles conseguem cumprir com os objetivos da ferramenta.

4.2.1 Test Case Generator

Para que a automação ou semi-automação dos testes de unidade, citada anteriormente como um dos objetos principais da ferramenta, realmente ocorra, é necessário que exista geração automática de classes de teste bem como de dados de entrada iniciais para testar a classe em questão (identificada na Figura 4.1 como "Java CUT"). Esta geração é realizada pelo primeiro módulo do fluxo padrão de utilização do EvolUniT, chamado de *Test Case Generator* (TCG).

Como o EvolUniT precisava da geração automática de código para cumprir um de seus objetivos, algumas ferramentas de geração automática de casos de teste JUnit foram pesquisadas, dentre as principais [JUB] e [JUnitDoclet]. A ferramenta escolhida foi o *plugin* JUB (JUnit test case Builder) por possuir um gerador de código um pouco mais extensivo do que os providos por outras ferramentas, já que este leva em consideração a presença de múltiplos construtores e sobrecarga de métodos. Apesar dessas vantagens, o JUB serviu apenas de ponto de partida para construção do TCG, pois o mesmo apenas gera um *skelleton* de uma classe de teste, sendo necessário completá-lo em grande parte, de acordo com os métodos da classe sendo testada. Diante desta limitação, e aproveitando que a licença do JUB é LGPL (*Lesser General Public License*), que permite a alteração e redistribuição do código da ferramenta, aproximadamente 50% do código do JUB foi alterado e incrementado para realizar as atividades necessárias ao EvolUniT.

O TCG gera 4 artefatos, conforme ilustra a Figura 4.1: (1) uma classe Java que corresponde ao caso de teste (*JUnit Test Case Class*), responsável por testar várias vezes a CUT; (2) outra classe Java que corresponde a uma *test suite,* responsável por executar a classe de teste tantas vezes quanto foram estabelecidas pelas configurações do AG e gerar eventos de captura de informações; (3) uma lista de arquivos XML com o nome da classe

de teste, que corresponde a cada indivíduo da população do AG a ser evoluído e possui um número variado de casos de teste (parâmetros de entrada para construtores e métodos da CUT); e (4) um arquivo XML de domínio, que possibilita ao desenvolvedor ou testador determinar intervalos para os parâmetros gerados nos casos de teste, aproveitando assim o seu conhecimento prévio do sistema em prol dos testes. É importante salientar que a classe de teste gerada pelo TCG não possui *Test Oracles*, ou seja, as expressões que verificam se as saídas do sistema estão de acordo com as saídas esperadas. A inserção das expressões Java responsáveis por tais tarefas, as *assertions*, precisam ser feitas pelo desenvolvedor por enquanto. Para um exemplo de geração automática dos artefatos pelo TCG, considere a classe Java simples *ExtendedHelloWorld* ilustrada na Figura 4.2. Esta classe possui apenas um método que exibe uma *String* passada por parâmetro na tela, caso outro parâmetro booleano *show* seja passado como *true*.

```
public class ExtendedHelloWorld {

    public void extendedHello(String text, boolean show){

        if (show){
            if(text != null){
                System.out.println(text);
            }
        }
        else{
            System.out.println("texto não pode ser exibido!");
        }
    }
}
```

Figura 4.2: Classe Java ExtendedHelloWorld.

O primeiro artefato a ser gerado pelo EvolUnit é o arquivo de domínio, pois a partir dele o desenvolvedor irá determinar as palavras que estarão presentes nos casos de teste para o parâmetro *text*. O parâmetro show não precisa de domínios especificados, pois a variável boolean já possui um domínio definido de *true* ou *false*. As figuras 4.3 e 4.4 mostram respectivamente o arquivo de domínio gerado pelo EvolUnit e o mesmo preenchido pelo usuário. Nas 2 figuras nota-se a presença de 4 tags XML: a primeira, denominada *domains*, possui a propriedade *className* que indica o nome da classe sendo testada; a segunda, chamada *method*, representa cada método público na classe sendo testada, que no caso é apenas o método *extendedHello*; a tag *params* contém os elementos

que são parâmetros do método correspondente; a tag *param* representa cada parâmetro propriamente dito, contendo as propriedades *name* e *type*, que são respectivamente o nome e o tipo do parâmetro; e a tag *vocabulary*, contendo um ou mais elementos *word*, representa o vocabulário de um parâmetro do tipo String do método, ou seja, o conjunto de palavras do domínio que podem ser escolhidas como parâmetro. Quando o método sendo testado possui um parâmetro do tipo String, o sistema gera o arquivo de domínios com apenas uma palavra *default*, conforme mostra a Figura 4.3. O usuário pode aumentar o domínio do vocabulário acrescentando novas palavras, como mostra a Figura 4.4.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<domains className="ExtendedHelloWorld">
    <method name="extendedHello">
        <params>
            <param name="text" type="String">
                <vocabulary>
                    <word>test</word>
                </vocabulary>
            </param>
        </params>
    </method>
</domains>
```

Figura 4.3: Arquivo de domínios gerado pelo EvolUniT.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<domains className="ExtendedHelloWorld">
    <method name="extendedHello">
        <params>
            <param name="text" type="String">
                <vocabulary>
                    <word>Hello World!!</word>
                    <word>null</word>
                    <word> </word>
                </vocabulary>
            </param>
        </params>
    </method>
</domains>
```

Figura 4.4: Arquivo de domínios gerado pelo EvolUniT e preenchido.

Depois de estabelecer o domínio na geração automática dos parâmetros que vão compor os casos de teste iniciais, os outros 3 artefatos podem ser gerados simultaneamente pelo TCG. Nos Apêndices A e B encontram-se respectivamente os códigos da classe caso de teste (ExtendedHelloWorldTest.java) e da classe de suíte de testes (ExtendedHelloWorldTestSuite.java). A Figura 4.6, que se encontra na seção mais à frente que fala do *GA Component*, mostra um dos arquivos XML gerados, que representa um indivíduo ou cromossomo no AG e sua coleção de casos de teste. Quanto ao último, a seqüência ao final do nome do arquivo identifica o número da geração e o id do indivíduo gerado. Como no momento da geração o Algoritmo Genético ainda não foi iniciado, a geração correspondente é a primeira, identificada pelo número 0. O número de casos de teste deste exemplo (3 casos de teste) é o fruto de uma escolha aleatória realizada dentro de um intervalo previamente estabelecido no arquivo de propriedades do EvolUniT.

A próxima sub-seção mostra como funciona o segundo módulo do EvolUniT, responsável pela atividade de monitoração e captura dos resultados dos testes.

4.2.2 Coverage Analyser

A avaliação da cobertura de código é o ato de identificar as partes de um programa que foram executadas em uma ou mais execuções deste programa. Desenvolvedores e testadores utilizam cobertura de código para tentar assegurar que todas as linhas de um programa foram executadas pelo menos uma vez no processo de teste [Tikir e Hollingsworth, 2002]. Tendo em vista a automação de testes de unidade Java, gerando e executando automaticamente casos de teste utilizando *JUnit*, e a evolução destes casos de teste usando Algoritmos Genéticos, é imprescindível que haja uma forma de medir o grau de eficiência dos casos de teste gerados. O módulo *Coverage Analyser* (CA) é responsável por fornecer esta métrica para posterior utilização, que nada mais é do que a porcentagem de decisões (*branches*) dos métodos de uma classe executadas por cada caso de teste, conforme descrita na Seção 2.4.2. Esta métrica de análise de cobertura foi escolhida dentre as outras por fornecer uma combinação de simplicidade e eficiência no que diz respeito à avaliação da cobertura de um programa.

O *Coverage Analyser*, conforme dito anteriormente, foi criado com o objetivo de gerar relatórios de cobertura necessários para a avaliação dos casos de teste gerados. Como

os objetivos principais do EvolUniT são a automação e evolução dos casos de teste, não foi necessária a construção de uma ferramenta de análise de cobertura, tendo em vista que existem várias no universo *open-source*. Sendo assim, várias ferramentas *open-source* foram pesquisadas para serem o ponto de partida do módulo *Coverage Analyser* [EMMA] [Cobertura] [DjUnit]. Ferramentas pagas como [Clover], que é uma das mais utilizadas para relatórios de cobertura, não foram consideradas já que precisariam ser modificadas para fazerem parte do EvolUniT. Para isso, era preciso uma ferramenta que possuísse uma licença como a GPL (*GNU General Public License*), para que fosse possível modificá-la e redistribuí-la.

A ferramenta escolhida no nosso trabalho foi o plugin para Eclipse chamado *DJUnit*, que possui licença GPL e continha os requisitos necessários para compor o *Coverage Analyser*. O *DJUnit* é um executor de testes baseado no *JUnit* com um carregador de classes customizado. O carregador de classes (*ClassLoader*) do DJUnit modifica a classe antes de ser carregada para a máquina virtual Java, ou seja, instrumenta o código binário já compilado utilizando um outro plug-in chamado *JCoverage* [JCoverage], e roda testes usando a classe modificada. Desta forma, relatórios de cobertura podem ser gerados com considerável facilidade.

O EvolUniT utiliza o *Coverage Analyser* da seguinte forma. O desenvolvedor determina no arquivo de propriedades do EvolUniT o número máximo de execuções (gerações) que o algoritmo vai rodar e uma condição de parada (cobertura máxima) caso deseje, permitindo assim que o AG pare antes de completar todas as gerações. A classe de teste gerada pelo *Test Case Generator* é executada e o CA modifica a classe sendo testada e assegura que esta é a que será executada. Um arquivo XML de registros de cobertura para cada caso de teste, chamado *evolunitresults.xml*, é gerado no diretório da aplicação. Para cada caso de teste terminado, um evento é gerado pelo CA para que um registro seja adicionado ao relatório de cobertura. Caso a condição de parada seja atingida, um outro evento é gerado e o CA não mais registra dados no arquivo. Além da cobertura de condições, o relatório também guarda o tamanho de cada conjunto de casos de teste ou indivíduo, para ser posteriormente utilizado na função objetivo. A Figura 4.5 mostra um exemplo de relatório gerado pelo *Coverage Analyser*, após os testes da classe *ExtededHelloWorld* apresentada como exemplo. O arquivo possui os seguintes elementos: a

tag *results*, que possui o nome da classe de teste; a tag *generation*, que indica qual a população (geração) sendo executada e o tamanho (nº de indivíduos) da mesma; e a tag *individual*, que possui o nome do arquivo XML que representa o indivíduo, a cobertura de decisões atingida pelo conjunto de dados desse indivíduo e o tamanho dele.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <results name="ExtendedHelloWorldTest">
  - <generation population="0" size="2">
      <individual branchCoverageRate="0.91" id="0" individualSize="2" name="ExtendedHelloWorldTest-data00" />
      <individual branchCoverageRate="1.0" id="1" individualSize="2" name="ExtendedHelloWorldTest-data01" />
    </generation>
  </results>
```

Figura 4.5: Arquivo evolunitresults.xml gerado pela Coverage Analyser após os testes da classe ExtendedHelloWorld.java.

A seção a seguir explica detalhadamente como funciona o terceiro e último módulo da ferramenta, que guarda todas as operações do Algoritmo Genético executado pelo EvolUniT, e é responsável pela evolução dos casos de teste.

4.2.3 GA Component

Os 2 módulos descritos acima são responsáveis pela automação do processo de testes de unidade e sua avaliação. Para que a ferramenta implementada fosse de Teste Evolutivo, era preciso que um algoritmo evolutivo fosse utilizado para evoluir os dados ou casos de teste completos gerados. O módulo do EvolUniT, chamado de *GA Component*, foi criado com o propósito de cumprir este requisito essencial de evolução dos dados, utilizando um Algoritmo Genético para tanto.

Conforme foi explicado na Seção 3.4, os Algoritmos Genéticos possuem determinados passos padrões que caracterizam sua utilização, tais como: a geração de uma população inicial de cromossomos; a avaliação de cada cromossomo através de uma função de aptidão definida previamente; a seleção dos cromossomos mais aptos para compor a próxima geração; e as modificações dos cromossomos selecionados, com determinada taxa de probabilidade, através dos operadores de cruzamento (crossover) e mutação [Lacerda e

Carvalho, 1999]. As próximas seções descrevem a forma com que cada um destes passos e operadores foi implementado no EvolUniT.

Representação dos Cromossomos

Conforme explicado na Seção 4.2.1, a primeira geração da população de cromossomos do AG, representados por arquivos XML, é gerada automaticamente pelo TCG de forma aleatória, porém com um domínio especificado. Na seção 3.4, quando falamos de Algoritmos Genéticos, foi visto que a representação mais simples de um cromossomo em um AG é uma estrutura de dados, como um vetor ou cadeia de bit, que é a estrutura mais tradicional. Como neste trabalho estamos aplicando AGs a um problema específico e muito mais complexo, a representação dos indivíduos da população também tem sua complexidade aumentada.

Assim como nos trabalhos já realizados de Teste Evolutivo OO [Tonella, 2004] [Cheon et al., 2005] [Wappler e Lammermann, 2005] [Wappler e Wegener, 2006], em que os cromossomos representavam programas de teste completos, diferentemente de Teste Evolutivo Convencional, o EvolUniT também possui uma representação característica dos cromossomos no mundo dos testes. Porém, diferentemente destes trabalhos citados, e apesar de o EvolUniT lidar com software orientado a objetos, a evolução é aplicada apenas aos dados gerados, tornando a representação do cromossomo um híbrido entre a convencional e a orientada a objetos. Essa combinação visa aproveitar a simplicidade da abordagem de TE Convencional, onde os indivíduos são representados apenas por conjuntos de dados de teste, no ambiente orientado a objetos.

A figura 4.6 mostra a representação de um cromossomo do EvolUniT, gerado na população inicial. O mesmo consiste em um arquivo XML contendo: uma ou mais *tags* "*method"*, que corresponde a um método de teste, de forma que a quantidade de tags varia de acordo com os métodos públicos presentes na classe sendo testada; um conjunto de *tags* "test-case", com tamanho variando dentro de um intervalo especificado no arquivo de propriedades; e um conjunto de *tags* "*param"*, representando os parâmetros de cada método público testado com seus tipos e valores.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://jtestcase.sourceforge.net/dtd/jtestcase2.xsd">
    <class name="ExtendedHelloWorld" size="3">
        <method name="extendedHello">
            <test-case name="testExtendedHello10">
                <params>
                    <param name="text" type="String">Hello World!!</param>
                    <param name="show" type="boolean">true</param>
                </params>
            </test-case>
            <test-case name="testExtendedHello11">
                <params>
                    <param name="text" type="String">null</param>
                    <param name="show" type="boolean">false</param>
                </params>
            </test-case>
            <test-case name="testExtendedHello12">
                <params>
                    <param name="text" type="String">null</param>
                    <param name="show" type="boolean">true</param>
                </params>
            </test-case>
        </method>
    </class>
</tests>
```

Figura 4.6: Arquivo ExtendedHelloWorldTest-data00.xml gerado, que representa um cromossomo na população.

A próxima seção trata de que forma tais indivíduos gerados são selecionados para compor as gerações subseqüentes.

Função Objetivo e Seleção

Após a geração inicial da população de cromossomos pelo TCG e após os resultados capturados pelo CA de acordo com a execução dos casos de teste (conjuntos de dados de teste) de cada indivíduo, o *GA Component* entra de fato em ação. O seu primeiro passo é utilizar uma função de aptidão para avaliar cada indivíduo. Esta avaliação ocorre apenas para os dados gerados, já que a classe de teste é a mesma para todos os indivíduos. A função de aptidão (*fitness*) de cada indivíduo usada na ferramenta é definida na equação abaixo:

$$fitness_{individual} = \sum_{i=1}^{size} (\text{cov}\, erageRate_{testcase} \times 100) - (\alpha \times size_{individual}) \qquad (4.1)$$

A equação acima é composta por dois critérios: (1) a nota de cada indivíduo (*fitness*), representada por um valor real, é dada pela soma acumulada dos valores de

cobertura de decisões (*coverageRate*) monitoradas pelo CA, multiplicados por 100 (escala de 0 a 100) de todos os casos de teste de um indivíduo; (2) subtraída do tamanho (número de casos de teste) de cada indivíduo (arquivo XML) ponderada por uma constante α. Isto significa que a aptidão do indivíduo cresce à medida que sua cobertura também cresce, e decresce quanto maior for o seu tamanho. Desta forma, a medida tenta garantir que o resultado final contenha os indivíduos que cobriram mais linhas de código com o custo mais baixo, isto é, com menos casos de teste. A constante alfa possui um valor muito pequeno (0,1), para que o tamanho do indivíduo tenha também um peso muito pequeno na avaliação, para servir apenas como critério de desempate na seleção de indivíduos com coberturas iguais. A cobertura fica então sendo o critério dominante na escolha dos indivíduos para a próxima geração.

Em vários dos trabalhos anteriores que utilizam TE, a função objetivo é calculada para cada predicado ou condição que se quer atingir usando a distância *Hamming* [Hamming Distance] entre os predicados [Sthamer, 1996] [Harman 2002], ou a distância que falta para o nó alvo [Baresel, 2003], ou ainda combinação destas duas medidas [McMinn, 2005]. Diferentemente destes trabalhos, a função objetivo proposta por nós para o EvolUniT é calculada de forma global. Isto é, o objetivo não é tratado de forma isolada, para cada predicado ou nó alvo, e sim de forma que toda a unidade sendo testada obtenha o seu valor máximo de cobertura.

A abordagem diferente de lidar com os valores de cobertura em um software de Teste Evolutivo, e a utilização desta pelo EvolUniT implica em vantagens e desvantagens. A principal desvantagem é que o AG não guia tão bem a busca para uma área do espaço de busca que possua o domínio dos predicados, quanto às abordagens que tratam de cada predicado de forma isolada. Essas abordagens, porém, funcionam bem quando os componentes do predicado são de tipos numéricos, e possuem sérios problemas quando trata de valores booleanos [Harman et al., 2002] [Bottacci, 2002] [Baresel et al., 2004] ou não são tão simples quanto se trata de tipos complexos, como objetos. A vantagem da abordagem aqui apresentada é de que ela não depende dos predicados, e também de que o domínio, perdido com a falta de exploração dos predicados, é compensado com a inserção de forma manual dos intervalos dos parâmetros por parte do próprio desenvolvedor. Essa abordagem aproveita o conhecimento prévio que o desenvolvedor deverá possuir das

unidades sendo testadas. Com a desvantagem de retirar um pouco da automação do processo, deixando-o semi-automático, mas acrescentando o conhecimento do engenheiro responsável pelos testes, a função de aptidão aqui apresentada tem o que se precisa para conseguir bons resultados na ferramenta.

Depois de calculada a nota de cada indivíduo de uma população, a seleção ocorre através de um procedimento chamado Roda da Roleta [Lacerda e Carvalho, 1999] cujo algoritmo está descrito na Figura 4.7 e que funciona da seguinte forma: é gerado um número aleatório *rand* (tirado de uma distribuição uniforme) dentro do intervalo [0, total], onde total é a soma das aptidões da população corrente. Então o cromossomo selecionado é o primeiro que possui o valor de aptidão acumulada maior que *rand*, e uma cópia sua é colocado em uma população intermediária. Os mesmos passos são repetidos até preencher a população intermediária com N cromossomos, onde N é o número de indivíduos da população inicial. O algoritmo de Roda da Roleta nem sempre seleciona os indivíduos com aptidões mais altas, por possuir certo grau de aleatoriedade, porém garante a diversidade da população.

```
total ← Σ_{i=1}^{N} f_i   /* a soma das aptidões de todos os cromossomos da
população */
rand ← randômico(0, total)
total parcial ← 0
i ← 0
repetir
        i ← i + 1
        total parcial ← total parcial + f_i
até total parcial ≥ rand
retornar o cromossomo s_i
```

Figura 4.7: Algoritmo Roda da Roleta.

A próxima seção demonstra como são efetuados os operadores genéticos nos indivíduos gerados pelo EvolUniT.

Cruzamento e Mutação

Os operadores de cruzamento (*crossover*) e mutação são os principais mecanismos de busca dos AGs para explorar regiões desconhecidas do espaço de busca [Lacerda e

Carvalho, 1999]. Como já explicado na Seção 3.4, o operador de cruzamento é aplicado a um par de cromossomos previamente selecionado da população anterior e presentes em uma população intermediária, gerando 2 cromossomos filhos. Cada cromossomo pai é "partido" em uma posição aleatória, de acordo com o tamanho dele, e cada filho recebe uma parte de um dos pais e uma parte do outro pai.

Para melhor entender o operador crossover, observe as figuras 4.8, 4.9, que contêm dois cromossomos pai de tamanhos 2 e 3, respectivamente. Inicialmente, um ponto de corte é escolhido para cada cromossomo pai. Na figura 4.8, como exemplo, o ponto de corte foi escolhido na posição 1, enquanto na figura 4.9 o ponto de corte corresponde à posição 2. Segmentos de cada cromossomo pai são então concatenados para gerar dois cromossomos filho, ilustrados nas figuras 4.10 e 4.11.

```
<method name="extendedHello">
    <test-case name="testExtendedHello00">
        <params>
            <param name="text" type="String">Hello World!</param>
            <param name="show" type="boolean">false</param>
        </params>
    </test-case>
    <test-case name="testExtendedHello01">
        <params>
            <param name="text" type="String"> </param>
            <param name="show" type="boolean">false</param>
        </params>
    </test-case>
</method>
```

Figura 4.8: Cromossomo pai 1.

```
<method name="extendedHello">
    <test-case name="testExtendedHello10">
        <params>
            <param name="text" type="String">Hello World!!</param>
            <param name="show" type="boolean">true</param>
        </params>
    </test-case>
    <test-case name="testExtendedHello11">
        <params>
            <param name="text" type="String">null</param>
            <param name="show" type="boolean">false</param>
        </params>
    </test-case>
    <test-case name="testExtendedHello12">
        <params>
            <param name="text" type="String">null</param>
            <param name="show" type="boolean">true</param>
        </params>
    </test-case>
</method>
```

Figura 4.9: Cromossomo pai 2.

Diferentemente do caso tradicional, em que cada cromossomo tem o mesmo tamanho de genes, no EvolUniT cada cromossomo pode ter um tamanho diferente, dentro de um intervalo especificado. Isso torna o processo de cruzamento um pouco diferente, no que diz respeito ao processo de partição e no tamanho dos filhos gerados. Quando os pais têm o mesmo tamanho de genes, eles terão partições iguais, mas quando suas quantidades de casos de teste forem diferentes, haverá uma partição para cada cromossomo pai. No caso ilustrado a cima, o filho 1 que deteve a primeira metade do pai 1, destacada pelo retângulo de cima na figura 4.10, e a segunda metade do pai 2, destacada pelo retângulo de baixo, ficou com tamanho 3. Já o filho 2, que ficou com a primeira metade do pai 2 e a segunda metade do pai 1, ficou com tamanho 2, como mostra a figura 4.11. O cruzamento no EvolUniT, assim como em situações padrões, ocorre com uma determinada taxa de probabilidade para cada geração. Esta taxa de probabilidade pode ser especificada pelo usuário no arquivo de propriedades.

```
<method name="extendedHello">
    <test-case name="testExtendedHello00">
        <params>
            <param name="text" type="String">Hello World!</param>
            <param name="show" type="boolean">false</param>
        </params>
    </test-case>
    <test-case name="testExtendedHello11">
        <params>
            <param name="text" type="String">null</param>
            <param name="show" type="boolean">false</param>
        </params>
    </test-case>
    <test-case name="testExtendedHello12">
        <params>
            <param name="text" type="String">null</param>
            <param name="show" type="boolean">true</param>
        </params>
    </test-case>
</method>
```

Figura 4.10: Cromossomo filho 1.

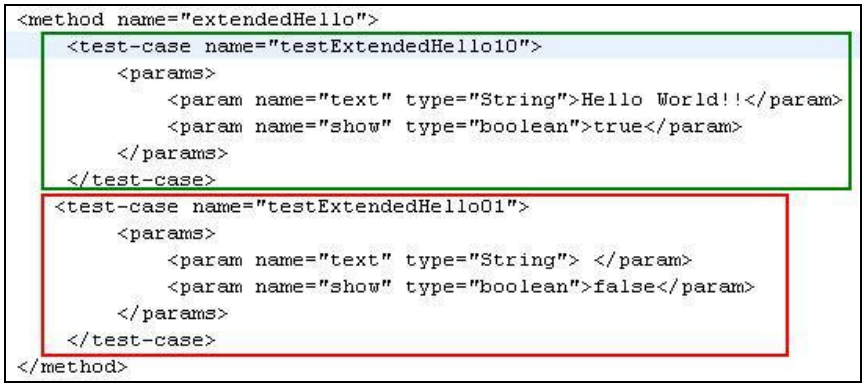

Figura 4.11: Cromossomo filho 2.

Em se tratando de mutação, o EvolUniT também possui uma forma característica e um pouco diferente de efetuá-la nos indivíduos. Na representação dos indivíduos em forma de cadeias de *tags* xml (casos de teste), um gene corresponde a uma *tag test-case*, e esta pode possuir um ou mais parâmetros. No operador de mutação implementado na ferramenta, é realizado inicialmente o teste de probabilidade de mutação para determinado indivíduo, ou seja, cada indivíduo pode sofrer ou não mutação dependendo do teste realizado com a taxa de probabilidade de mutação especificada no arquivo de propriedades. Se o indivíduo passou no teste, uma posição (caso de teste) é sorteada para este indivíduo e a mutação é realizada. Para o caso de teste que for sorteado, cada parâmetro passa por outro teste de probabilidade de 50% para definir se este será ou não preenchido com outro valor de parâmetro. O novo valor é escolhido de forma aleatória, dentro das opções especificadas no arquivo xml de domínios.

No caso de parâmetros booleanos, a substituição é simplesmente uma troca pelo valor oposto. A probabilidade de mutação, assim como a de cruzamento, é especificada no arquivo de propriedades, e esta tende a ser menor do que a taxa de cruzamento. Isso tem como objetivo fazer com que os valores permaneçam no domínio anteriormente especificado. A figura 4.12 ilustra o funcionamento do operador de mutação no EvolUniT, para um parâmetro booleano.

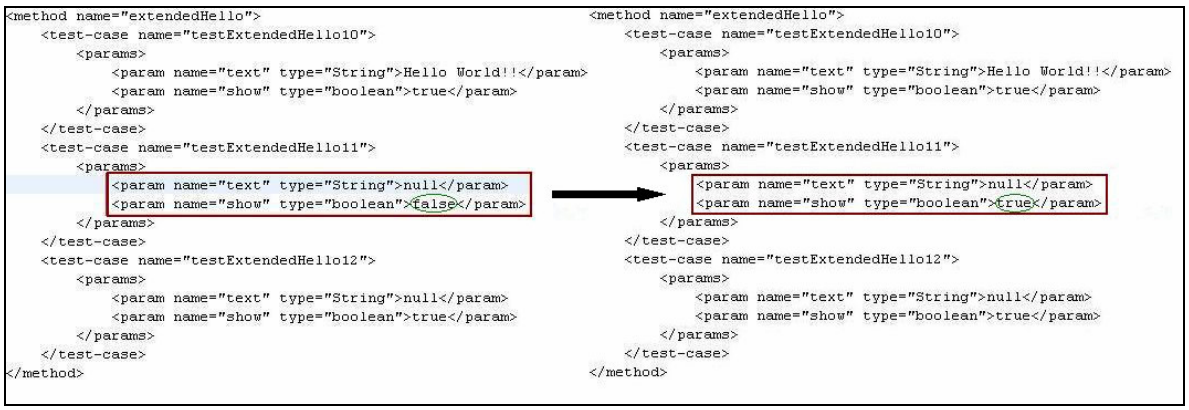

Figura 4.12: Mutação em um indivíduo.

A forma com que estes operadores genéticos foram implementados no EvolUniT é mais um diferencial deste trabalho com relação aos demais de Teste Evolutivo. A seção a seguir faz um breve apresentação do EvolUniT, mostrando o fluxo padrão de utilização deste plugin no Eclipse.

4.3 Usando o EvolUniT

As seções anteriores explicaram cada módulo do EvolUniT, de forma isolada, e mostraram exemplos dos artefatos gerados pelo mesmo, tendo como entrada a classe ExtendedHelloWorld. Agora o foco é na utilização do plugin, por parte do engenheiro de software responsável. Para o exemplo desta utilização, considere a simples classe *QuadraticEquation.java*, ilustrada na figura 4.13, que possui um método apenas chamado *quadratic*. Este método calcula o valor de delta (d) de uma equação ($b^2 - 4 \times a \times c$), e retorna se a mesma é quadrática ($a.x^2 + b.x + c$) ou não, e se suas raízes são: reais e iguais, reais e diferentes ou complexas caso seja quadrática.

```
public class QuadraticEquation {

    static final int NOT_A_QUADRATIC = 0;
    static final int REAL_AND_UNEQUAL_ROOTS = 1;
    static final int REAL_AND_EQUAL_ROOTS = 2;
    static final int COMPLEX_ROOTS = 3;

    public int quadratic(int a, int b, int c){

        double d;
        int result;

        if (a == 0){
            result = NOT_A_QUADRATIC;
        }
        else{
            d = Math.pow(b, 2) - 4*a*c;

            if (d > 0){
                result = REAL_AND_UNEQUAL_ROOTS;
            }
            else{
                if (d == 0){
                    result = REAL_AND_EQUAL_ROOTS;
                }
                else{
                    result = COMPLEX_ROOTS;
                }
            }
        }
        return result;
    }
}
```

Figura 4.13: Código da classe QuadraticEquation.

O primeiro passo para gerar os casos de teste e os evoluir com o EvolUniT é especificas as variáveis do arquivo de propriedades, chamado *EvolUnit.properties*, que precisa ser colocado no diretório do projeto em que se encontra a classe. A figura 4.14 mostra o arquivo de propriedades preparado com o exemplo desta seção. No arquivo estão especificadas as seguintes propriedades:

(1) *generations*, que corresponde ao número máximo de gerações que o AG irá rodar;

(2) *populationSize*, que é o número de indivíduos que serão gerados inicialmente e evoluídos a cada geração;

(3) *chromosomeSizeRange*, correspondente ao intervalo para o tamanho de cada indivíduo em número de casos de teste;

(4) *crossoverProb*, sendo a taxa de probabilidade de ocorrer cruzamento para cada par de cromossomos da população;

(5) *mutationProb*, sendo a taxa de probabilidade de ocorrer mutação em cada indivíduo da população;

(6) *stopCoverageCondition*, que corresponde à condição de parada do AG com relação à máxima cobertura atingida;

(7) *useDataDomains*, uma *flag* que indica se o usuário irá utilizar o arquivo de domínios ou não para a geração inicial (recomenda-se que seja sempre *true*);

(8) *individualsPath*, que indica o caminho do diretório *individuals*. Esse diretório deve ser criado pelo usuário no local desejado e é onde ficarão situados os arquivos XML correspondentes aos indivíduos das populações.

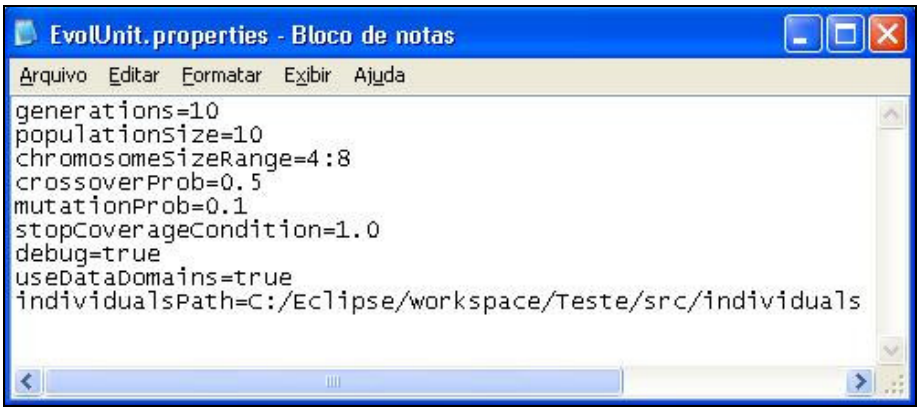

Figura 4.14: Arquivo EvolUnit.properties.

Após a criação e configuração do arquivo de propriedades, o próximo passo é gerar o arquivo de domínios da classe sendo testada. A figura 4.15 mostra como, no Eclipse, o desenvolvedor deve gerar este arquivo.

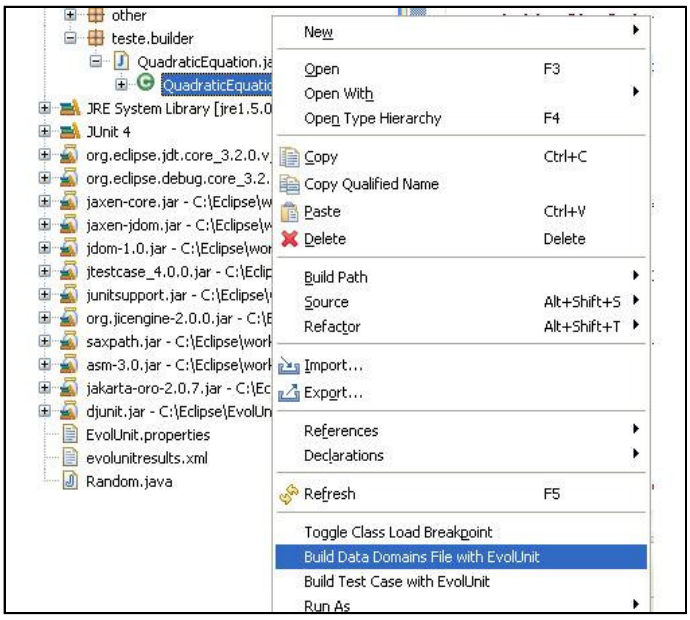

Figura 4.15: Gerando arquivo de domínios com o EvolUniT.

Depois de configurar este arquivo com os domínios necessários, o passo seguinte é gerar os outros 3 artefatos citados na seção 4.2.1: a classe de teste, a classe de suíte de testes e os arquivos XML correspondentes aos indivíduos, sendo 6 como configurado no arquivo de propriedades. A figura 4.16 ilustra esta geração e a figura 4.17 mostra onde estão situados todos os artefatos gerados, destacando em vermelho as classes de teste e de suíte geradas, em azul os indivíduos e em verde o arquivo de domínios.

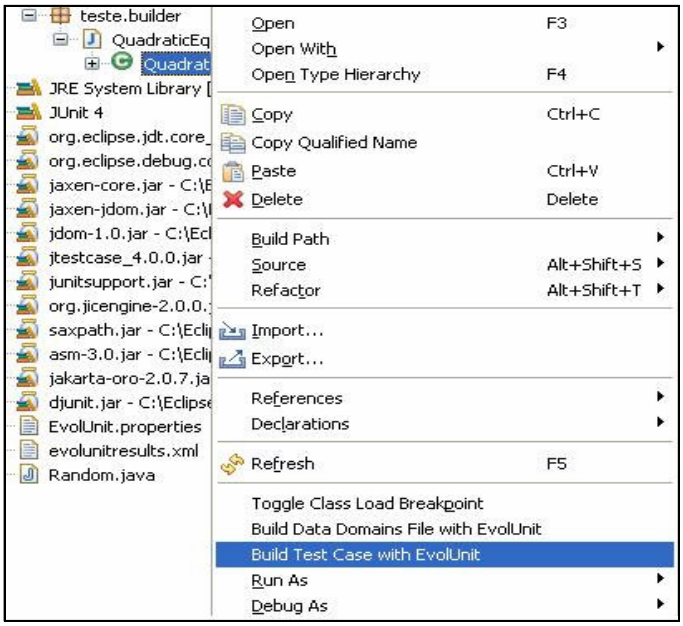

Figura 4.16: Gerando casos de teste com o EvolUniT.

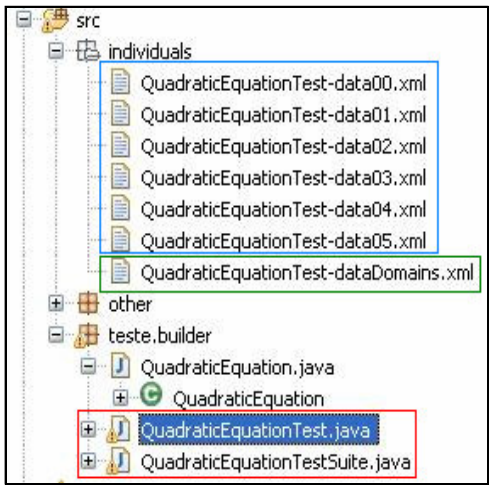

Figura 4.17: Artefatos Gerados pelo EvolUniT.

O último passo na utilização do EvolUniT é, finalmente, executar a classe de suíte, da forma que é mostrada na figura 4.18 e analisar os resultados expostos em um relatório gerado pelo plugin, como mostra a figura 4.19.

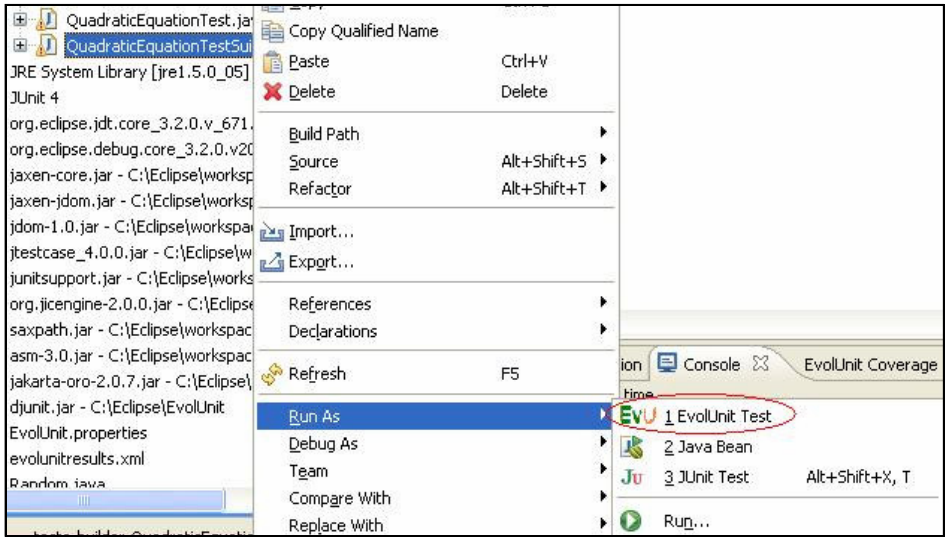

Figura 4.18: Executando o EvolUniT.

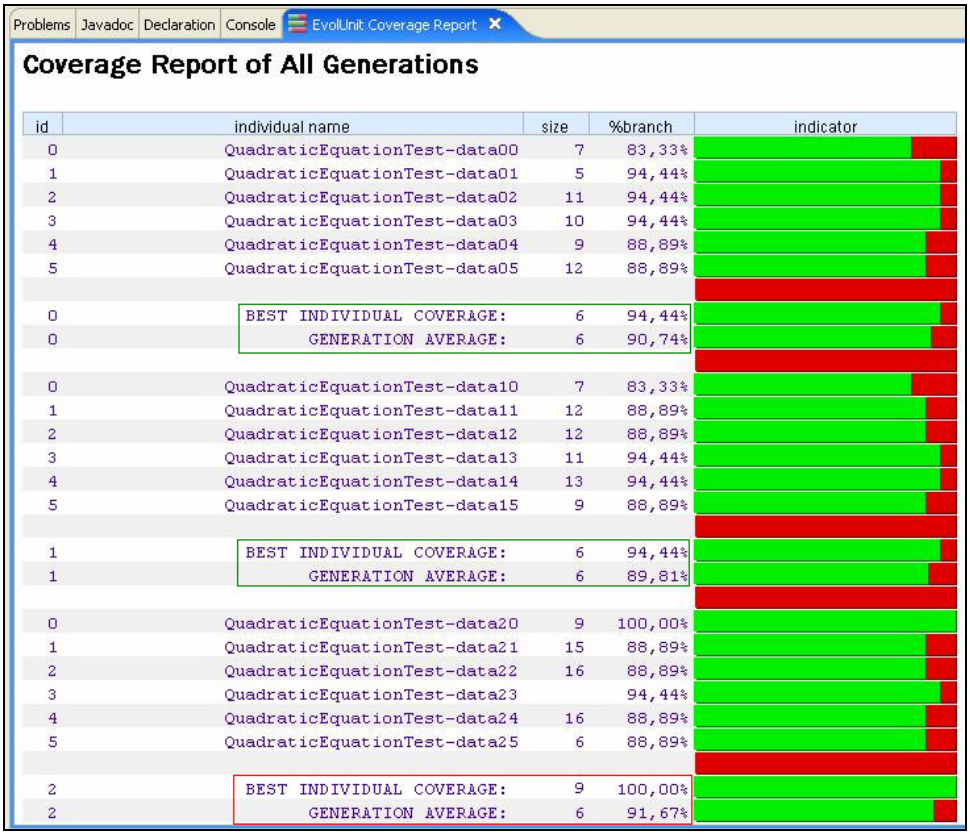

Figura 4.19: Visão do Relatório do EvolUniT no Eclipse.

O relatório possui as informações necessárias para a avaliação do AG, pois expõe a cobertura condicional atingida por cada indivíduo bem como seu tamanho, e aponta a cada geração o indivíduo com maior cobertura e a média das coberturas dos indivíduos da geração. As barras à direita mostram graficamente o percentual de cada indivíduo. Quando preenchida totalmente de verde, a barra indica que o indivíduo atingiu 100% de cobertura O relatório mostrou para esta execução que o algoritmo rodou 2 gerações além da inicial, conseguindo evoluir os dados para atingirem 100% de cobertura na 3ª geração.

4.4 Considerações Finais

Foi dito anteriormente que a aplicação da técnica de Teste Evolutivo implica na automação ou semi-automação do processo de teste que a está utilizando. Tendo em vista este preceito, foi visto que o EvolUniT foi criado para proporcionar uma semi-automação da fase de testes de unidade, utilizando para tanto: um gerador de código chamado dentro

do EvolUniT de *Test Case Generator*, para gerar os casos de teste e dados (parâmetros) dos mesmos; um Algoritmo Genético, onde seus operadores genéticos são implementados em um módulo chamado *GA Component*, criado para evoluir apenas os dados gerados pelo TCG; e um analisador de cobertura de código, chamado *Coverage Analyser*, que ajuda a montar a função objetivo do AG guardando as informações de cobertura adquiridas após cada execução dos casos de teste de cada indivíduo.

Apesar de o TCG gerar o código necessário para executar os métodos de uma classe com diferentes dados de entrada, nenhum *Test Oracle* é gerado para comparar as saídas reais com as esperadas. Por enquanto o desenvolvedor fica encarregado de inserir manualmente as asserções (*assertions*) de Java, após selecionar os melhores casos de teste expostos pelo relatório do EvolUniT. A automação deste processo ficou marcada como um trabalho futuro para ser desenvolvido no plug-in.

A função objetivo utilizada no *GA Component* é um diferencial da ferramenta. Enquanto que os trabalhos anteriores focam suas funções objetivo nos predicados de cada expressão do código e na distância ao nó alvo (no caso de cobertura de código), em tempo de execução (no caso de medir desempenho de códigos), ou ainda em eminências de falhas, o EvolUniT foca em cobertura de código global de todos os métodos não privados de uma classe, alimentando sua função de aptidão com os próprios valores de cobertura encontrados. Essa abordagem diferente tem a vantagem de fazer com que o conhecimento do engenheiro responsável pelo teste seja um componente adicional para guiar a busca a um domínio específico. A geração de testes no EvolUniT, contudo, é um processo semi-automático, uma vez que é necessário especificar o domínio do problema e complementar os casos de teste caso necessário. A busca deste domínio de forma automática, utilizando alguma análise estática do código fonte, é também um dos itens pertencentes à lista de trabalhos futuros para este plugin.

Também foi visto neste capítulo que os operadores de cruzamento e mutação do AG também possuem uma forma característica de serem aplicados no EvolUniT. No primeiro, as partições são horizontais, realizadas entre os elementos *test-case*, que representam os genes de um cromossomo. No segundo, em vez de haver uma troca há uma mudança aleatória de um ou mais parâmetros desse elemento *test-case*, onde os valores numéricos variam de acordo com uma taxa definida e as *Strings* de acordo com o vocabulário

especificado. Com esses tipos de operações definidas, espera-se que os indivíduos atinjam boas coberturas através do compartilhamento de dados entre eles, sem fugir ao domínio estabelecido.

No próximo capítulo, serão apresentados os experimentos que foram realizados no EvolUniT para validar sua eficácia, e também um estudo de caso utilizando classes de uma ferramenta industrial, para avaliar sua aplicabilidade a esse contexto.

5 Estudos de Caso

No capítulo 4, foi apresentado o plugin EvolUniT, com o intuito de automatizar o processo de testes de unidade de classes Java. A nova abordagem exposta neste trabalho envolve a geração de programas de teste, bem como os dados a serem usados como parâmetros do objeto de teste e a evolução destes através de um AG. A forma como a cobertura de decisões é observada para alimentar a função objetivo também é diferente no EvolUniT, conforme discutido.

Tendo em vista que o EvolUniT utiliza uma versão híbrida de TE, utilizando características de TE Convencional e de TE Orientado a Objetos, e que um dos objetivos da sua construção é de melhorar o desempenho e diminuir os custos de testes de unidade, 3 tipos de estudos foram realizados para avaliar a ferramenta. O primeiro estudo foi a realização de baterias de experimentos comparando o Teste Evolutivo realizado pelo EvolUniT com o Teste Aleatório. Essa comparação foi bastante utilizada em trabalhos anteriores de TE [Sthamer, 1996] [Tracey et al., 1998] [Wegener et al., 2002] [Wappler e Lammermann, 2005] e procura validar a capacidade do AG para evoluir os dados de teste gerados. Foi avaliada nesse estudo a cobertura de código (*branch coverage*) atingida pelo AG, levando em consideração as configurações dos operadores genéticos.

O segundo estudo de caso foi realizado em uma ferramenta industrial, desenvolvida por um grupo de pesquisa do projeto CIn / Motorola, chamada *TarGet*. Esse outro estudo tem como objetivo validar o plugin para utilização como ferramenta de automação de testes, por parte de profissionais da área de ES que usam o Eclipse como ferramenta de desenvolvimento. Foram avaliados nesse estudo a cobertura de código atingida, o número de gerações necessárias para atingir a cobertura máxima e o esforço poupado para cada classe de teste.

O terceiro estudo de caso comparou a EvolUniT com a ferramenta Randoop, que utiliza geração aleatória de casos de teste. O objetivo desse estudo de caso foi comparar a ferramenta desenvolvida com um sistema consolidado na literatura de geração de casos de teste.

Com os estudo realizados, visamos ter uma avaliação multidisciplinar, já que o primeiro tende para a área de IA e tem o foco na validação do Algoritmo Genético, e o

segundo e terceiro tendem para a área de ES. Após a realização de cada estudo os resultados foram analisados de forma isolada, e algumas conclusões sobre esses estudos são tiradas no final do capítulo.

5.1 Estudo de Caso 1 – Teste Evolutivo x Teste Aleatório

O primeiro estudo sobre o EvolUniT tem como principal objetivo validar a abordagem de Teste Evolutivo utilizada na ferramenta, analisando principalmente o Algoritmo Genético projetado para a evolução dos casos de teste (mais especificamente dos dados de entrada). Para a análise, será observado o valor da função objetivo alcançado para diferentes configurações do AG. Conforme dito anteriormente, a abordagem que prevalece neste estudo é a quantitativa, pois visa analisar os aspectos citados acima através da observação das medidas de cobertura de condições atingidas. Como apenas um objeto de teste é utilizado para este estudo, e como a classe de teste gerada para este não precisa de nenhum complemento (com exceção dos *test oracles*) a medida de esforço manual, ou seja, quantidade de linhas de código inseridas manualmente, não foi levada em consideração.

5.1.1 Objeto de Teste

Nos experimentos realizados, apenas um objeto de testes foi considerado neste primeiro estudo. Esse objeto se trata de uma classe Java chamada *TriangleClassification*, cujo código completo encontra-se no Apêndice C. Esta classe utiliza operadores relacionais de igualdade e desigualdade para cumprir seu objetivo, que se trata de determinar se um polígono de 3 lados dado como entrada é ou não um triângulo, e se for, classificá-lo como eqüilátero, isósceles ou escaleno. O código do método *getTriangleType* encontra-se na figura 5.1, assim como o gráfico que representa o controle de fluxo do mesmo. A classificação é realizada, efetivando a análise de 3 parâmetros de tipo inteiro de dados como entrada, que representam as medidas dos 3 lados do triângulo. Este programa de classificação foi tirado de [McMinn, 2005] e adaptado para linguagem Java. Uma versão mais completa deste também foi utilizada por [Sthamer, 1996].

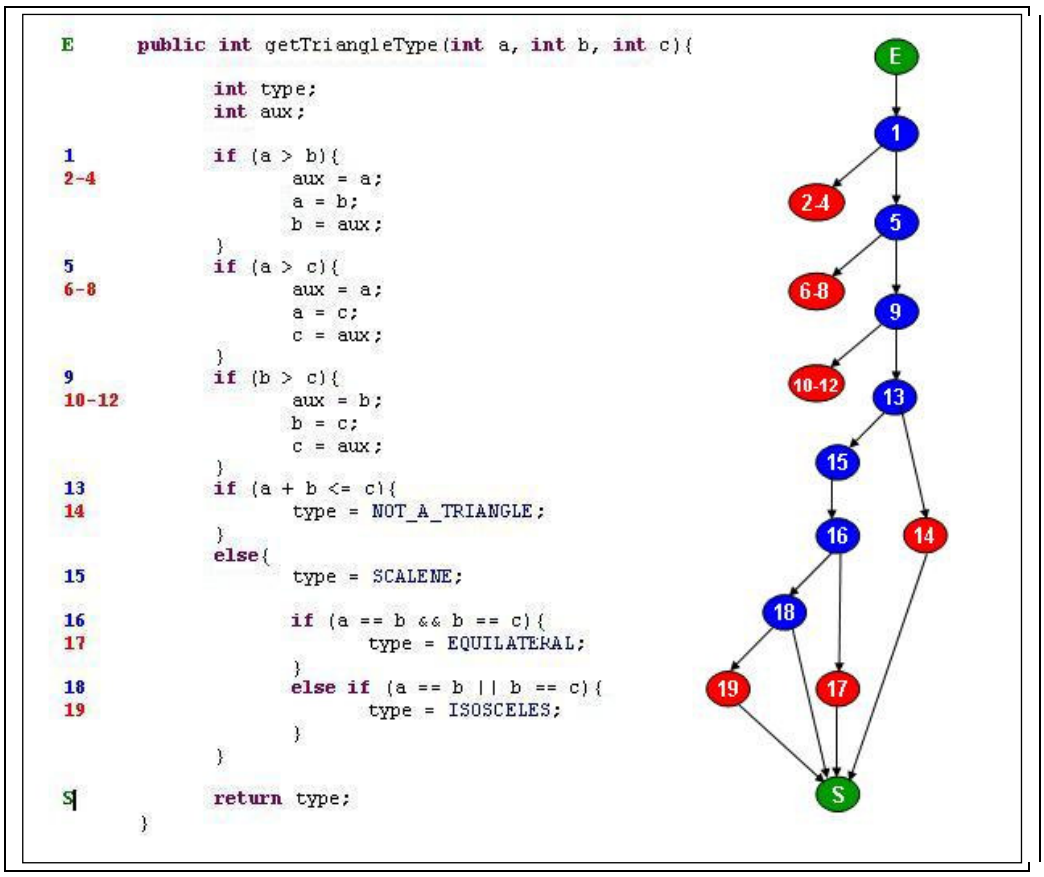

Figura 5.1 - Método *getTriangleType* e o seu gráfico de controle de fluxo.

5.1.2 Metodologia de Experimentos

Experimentos foram realizados com o EvolUniT para gerar testes para o objeto descrito na seção anterior, variando quatro parâmetros: o número máximo de gerações, o número de indivíduos por geração, a taxa de crossover e a taxa de mutação. Considerando os dois primeiros parâmetros, foram adotadas as três configurações seguintes: (1) 5 gerações e 5 indivíduos por geração, no total de 25 indivíduos gerados; (2) 10 gerações e 5 indivíduos por geração, no total de 50 indivíduos gerados; e (3) 10 gerações e 10 indivíduos por geração, no total de 100 indivíduos gerados.

Para cada uma das três configurações acima, variamos ainda os valores para a taxa de crossover e a taxa de mutação. Foram definidas quatro variações diferentes, combinando os valores de 70% e 50% para a taxa de crossover, e 30% e 10% para a taxa de mutação. Assim, considerando os quatro parâmetros avaliados, definimos um total de 12 configurações diferentes de experimentos para o AG. Ressaltamos ainda que, como os

resultados do AG são dependentes da população inicial, para cada configuração de experimentos executamos o AG por 10 vezes diferentes com população inicial aleatória.

Os resultados do AG foram comparados com o Teste Aleatório. Para realizar uma comparação justa, o Teste Aleatório foi aplicado para gerar o mesmo número total de indivíduos gerados nas execuções dos AGs, ou seja, 25, 50 e 100 indivíduos no total. Na prática, o Teste Aleatório foi implementado a partir do *GA Component*, entretanto substituindo os operadores de crossover e mutação por uma reinicialização aleatória dos indivíduos. Ou seja, ao invés de ocorrer cruzamentos e mutações entre gerações, os indivíduos foram reinicializados de forma aleatória, como ocorre na população inicial. Assim como nos experimentos com o AG, o Teste Aleatório foi executado 10 vezes para cada uma das duas três configurações de experimentos.

Para cada experimento realizado, foram registradas duas medidas de avaliação: (1) o valor médio de cobertura dos indivíduos por geração; e (2) o valor da melhor cobertura por geração. A análise dos resultados foi então realizada a partir da média obtida por esses dois critérios nas 10 execuções aleatórias.

Destacamos aqui que apesar de que o tamanho dos indivíduos foi um critério utilizado na função objetivo, para este estudo de caso, foram avaliados somente os valores de cobertura de *branches* atingidos, isto é, a porcentagem de nós de controle presentes no gráfico da figura 5.1. Isso se deve ao fato de que os tamanhos dos indivíduos foram definidos como valor fixo para os dois algoritmos avaliados. Nos experimentos, o tamanho dos indivíduos teve valor fixo 4 (casos de teste). Este valor se deve ao fato de que 4 combinações de valores de entrada são necessárias para rodar 100% do fluxo de controle do método *getTriangleType,* caso o melhor conjunto de casos de teste seja encontrado.

Outro valor importante é o intervalo dos valores de entrada do método. Estes são especificados no arquivo de domínios gerado pelo EvolUniT. Como os 3 parâmetros do método testado são do tipo inteiro, eles ficaram com intervalo entre 1 e 10.

Destacamos finalmente que o único critério de parada das execuções foi alcançar o número máximo de gerações. A condição de parada de cobertura máxima atingida foi retirada, por questões de igualdade para com a bateria de testes aleatórios. Sendo assim,

mesmo que algum indivíduo tenha atingido 100% de cobertura, os experimentos continuam a executar.

5.1.3 Resultados dos Testes Aleatórios

As tabelas 5.1, 5.2 e 5.3 mostram os resultados obtidos com o Teste Aleatório para suas três configurações de experimentos. Ao se fazer uma análise preliminar desta primeira bateria de testes, nota-se que mesmo dados aleatórios gerados, boas coberturas podem ser alcançadas (coberturas acima de 90%). Isto se deve ao fato de que o programa sendo testado é simples. De fato, o seu fluxo de controle possui vários nós que são fáceis de serem atingidos, considerando um pequeno conjunto de valores inteiros. A exceção é o nó 17, mostrado na figura 5.1. Para que este nó seja executado, é preciso que os 3 valores aleatórios gerados para um caso de teste sejam iguais, e isto é muito difícil de acontecer. A utilização de um AG na execução dos testes, no entanto, pode contornar esse problema.

Tabela 5.1 - Resultados dos testes aleatórios com 5 gerações e 5 indivíduos.

Teste	**Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração**									
	1ª		**2ª**		**3ª**		**4ª**		**5ª**	
	Melhor	**Média**	**Melhor**	**Média**	**Melhor**	**Média**	**Melhor**	**Média**	**Melhor**	**Média**
Média	95,27	92,33	95,55	92,50	96,11	93,11	94,44	92,17	95,55	92,89

Tabela 5.2 - Resultados dos testes aleatórios com 10 gerações e 5 indivíduos.

Teste	**Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração**									
Média	**1ª**		**2ª**		**3ª**		**4ª**		**5ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,37	92,59	95,06	92,34	94,75	92,22	95,06	92,47	95,68	93,23
	6ª		**7ª**		**8ª**		**9ª**		**10ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,68	92,65	94,44	92,65	95,06	92,78	94,75	91,91	94,13	91,98

Tabela 5.3 - Resultados dos testes aleatórios com 10 gerações e 10 indivíduos.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
Média	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,55	92,14	95,33	92,81	96,11	92,03	95,55	92,53	96,11	92,36
	6ª		7ª		8ª		9ª		10ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	96,11	92,22	96,11	92,61	95,55	92,47	95,83	92,86	95,28	92,15

5.1.4 Resultados dos Testes com AGs

As tabelas 5.4 a 5.15 mostram os resultados médios para as 10 execuções aleatórias de cada configuração de das taxas de *crossover* e mutação, do número de gerações e de indivíduos por geração. Para todas as configurações avaliadas para os AGs, observa-se que, em geral, os resultados são melhores quando comparados aos dos testes aleatórios. Mesmo para execuções com poucas gerações e poucos indivíduos por geração (i.e., 5 gerações e 5 indivíduos por geração), observamos, nas 4 configurações de operadores de busca, execuções de experimentos em que se chegou a 100% de cobertura, situação que não ocorreu nos testes aleatórios. Obviamente, com mais indivíduos e mais gerações, a probabilidade de que pelo menos um caso de teste consiga 100% de cobertura foi muito maior.

Os melhores resultados com AGs foram obtidos com a configuração de taxa de cruzamento de 70% e de mutação de 30%, e com a configuração de taxa de cruzamento de 50% e de mutação de 30%. Nessas configurações, em 50% dos experimentos (i.e., 5 execuções dentre as 10 repetições) pelo menos um caso de teste atingiu 100% de cobertura. Além disso, essas duas configurações também atingiram as melhores médias para cada geração.

Tabela 5.4 - Testes evolutivos com 5 gerações, 5 indivíduos, taxa de cruzamento a 70% e taxa de mutação a 10%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
Média	95,27	92,14	96,66	95,06	97,22	96,33	97,22	96,83	97,22	97,11

Tabela 5.5 - Testes evolutivos com 5 gerações, 5 indivíduos, taxa de cruzamento a 70% e taxa de mutação a 30%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
Média	95,27	92,53	96,39	95,44	97,78	96,50	98,61	97,50	98,61	97,89

Tabela 5.6 - Testes evolutivos com: 5 gerações, 5 indivíduos, taxa de cruzamento a 50% e taxa de mutação a 10%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
Média	94,81	92,43	95,83	94,95	96,29	95,92	96,99	96,43	96,99	96,90

Tabela 5.7 - Testes evolutivos com: 5 gerações, 5 indivíduos, taxa de cruzamento a 50% e taxa de mutação a 30%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
Média	95,27	93,00	96,94	94,94	98,05	96,72	98,61	97,33	98,50	98,39

Tabela 5.8 - Resultados dos testes evolutivos com: 10 gerações , 5 indivíduos, taxa de cruzamento 70% e taxa de mutação 10%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
Média	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,27	92,14	95,66	95,27	97,22	96,66	97,22	97,22	98,89	98,05
	6ª		7ª		8ª		9ª		10ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	97,22	97,11	97,22	97,22	98,89	98,33	98,05	97,44	98,89	98,88

Tabela 5.9 - Resultados dos testes evolutivos com: 10 gerações , 5 indivíduos, taxa de cruzamento 70% e taxa de mutação 30%.

Teste	**Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração**									
Média	**1ª**		**2ª**		**3ª**		**4ª**		**5ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,27	92,72	97,50	95,39	98,05	97,44	98,89	97,94	98,89	98,05
	6ª		**7ª**		**8ª**		**9ª**		**10ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	99,44	98,72	99,44	98,39	99,72	98,61	99,44	98,22	99,44	98,94

Tabela 5.10 - Resultados dos testes evolutivos com: 10 gerações , 5 indivíduos, taxa de cruzamento 50% e taxa de mutação 10%.

Teste	**Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração**									
Média	**1ª**		**2ª**		**3ª**		**4ª**		**5ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	94,81	92,11	95,66	94,95	97,22	95,55	97,22	96,11	97,22	96,66
	6ª		**7ª**		**8ª**		**9ª**		**10ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	97,22	97,11	97,22	97,22	98,05	97,44	98,89	98,05	98,89	98,33

Tabela 5.11 - Testes evolutivos com 10 gerações, 5 indivíduos, taxa de cruzamento 50% e taxa de mutação 30%.

Teste	**Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração**									
Média	**1ª**		**2ª**		**3ª**		**4ª**		**5ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,27	93,00	96,94	95,17	97,78	96,39	98,05	97,50	98,61	98,00
	6ª		**7ª**		**8ª**		**9ª**		**10ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	98,33	98,00	98,89	97,67	99,17	98,78	99,44	98,50	99,72	99,22

Tabela 5.12 - Testes evolutivos com 10 gerações, 10 indivíduos, taxa de cruzamento 70% e taxa de mutação 10%.

Teste	**Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração**									
Média	**1ª**		**2ª**		**3ª**		**4ª**		**5ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	94,81	92,11	95,66	94,44	97,22	95,55	97,22	97,22	97,22	96,11
	6ª		**7ª**		**8ª**		**9ª**		**10ª**	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	97,50	96,66	98,22	97,44	98,89	98,33	99,72	98,61	99,44	98,22

Tabela 5.13 - Testes evolutivos com 10 gerações, 10 indivíduos, taxa de cruzamento 70% e taxa de mutação 30%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
Média	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	96,11	92,25	97,78	95,11	99,72	97,31	99,44	97,75	99,44	98,39
	6ª		7ª		8ª		9ª		10ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	98,89	97,08	99,17	98,17	99,44	97,94	99,72	97,68	99,72	99,17

Tabela 5.14 - Resultados dos testes evolutivos com: 10 gerações , 10 indivíduos, taxa de cruzamento 50% e taxa de mutação 10%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
Média	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,27	92,72	95,66	94,44	97,22	96,11	97,22	97,22	97,78	96,32
	6ª		7ª		8ª		9ª		10ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	98,33	98,00	98,61	98,00	99,17	98,61	99,44	98,05	98,89	98,33

Tabela 5.15 - Testes evolutivos com 10 gerações, 10 indivíduos, taxa de cruzamento 50% e taxa de mutação 30%.

Teste	Cobertura de Decisões (%) – Média e Melhor Indivíduo por Geração									
Média	1ª		2ª		3ª		4ª		5ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	95,55	92,47	97,78	95,22	99,44	97,11	100,00	97,44	99,17	97,19
	6ª		7ª		8ª		9ª		10ª	
	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média	Melhor	Média
	99,17	97,33	99,44	97,47	99,44	98,36	99,44	97,64	100,00	98,64

5.1.5 Análise Comparativa dos Resultados

Após a execução dos experimentos, podemos tecer alguns comentários, comparando a técnica de Testes Aleatórios e os AGs. Para a configuração de 5 gerações e 5 indivíduos, pode-se observar que as médias dos melhores indivíduos com as quatro configurações de teste evolutivo no início começaram entre 95 e 96%, mas depois subiram até ficar entre 97 e 99%, ao contrário das médias do teste aleatório, que se mantiveram oscilando por perto

do valor inicial, como mostra o gráfico da figura 5.2. O mesmo padrão de resultados ocorreu se considerarmos as médias de cobertura alcançadas, como mostra o gráfico da figura 5.3.

Nesses experimentos, observamos alguns indivíduos que atingiram 100% de cobertura durante os testes evolutivos, aumentando assim as médias dos resultados. Isso se deve à aplicação do operador de mutação do EvolUniT, cuja característica de mudança em um ou mais parâmetros do conjunto de entradas de um indivíduo fez com que alguns indivíduos possuíssem ao menos um caso de teste que entrasse no nó 17 do gráfico de controle da figura 5.1. Observamos ainda que houve uma uniformidade entre os indivíduos após a aplicação dos operadores genéticos de cruzamento, fazendo com que as boas características de alguns, ou seja, determinados conjuntos de dados de entrada, fossem compartilhadas com outros.

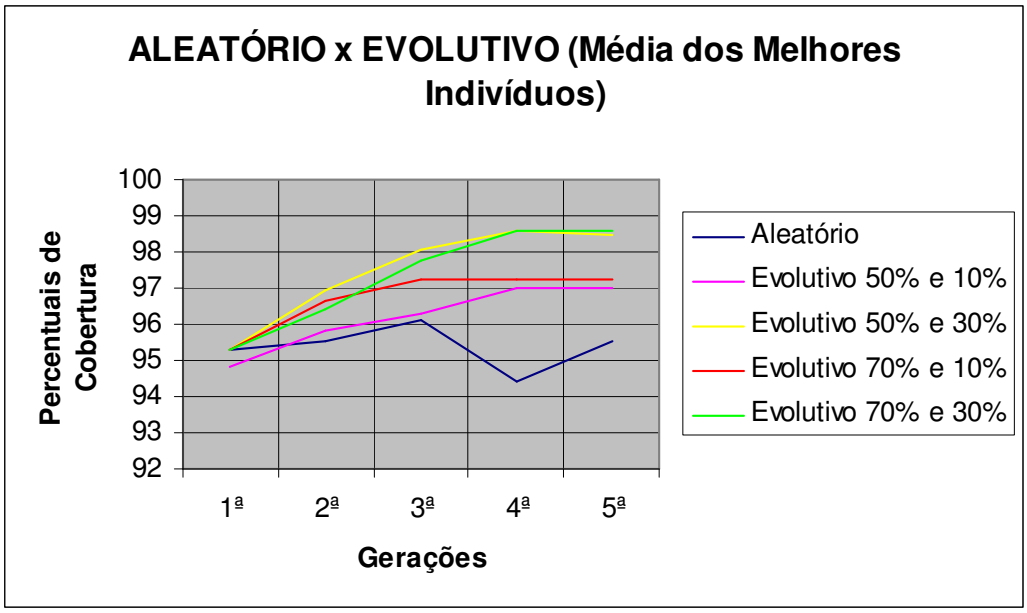

Figura 5.2 - Média dos melhores indivíduos obtidos entre as gerações, com Nº de gerações = 5 e Nº de indivíduos = 5.

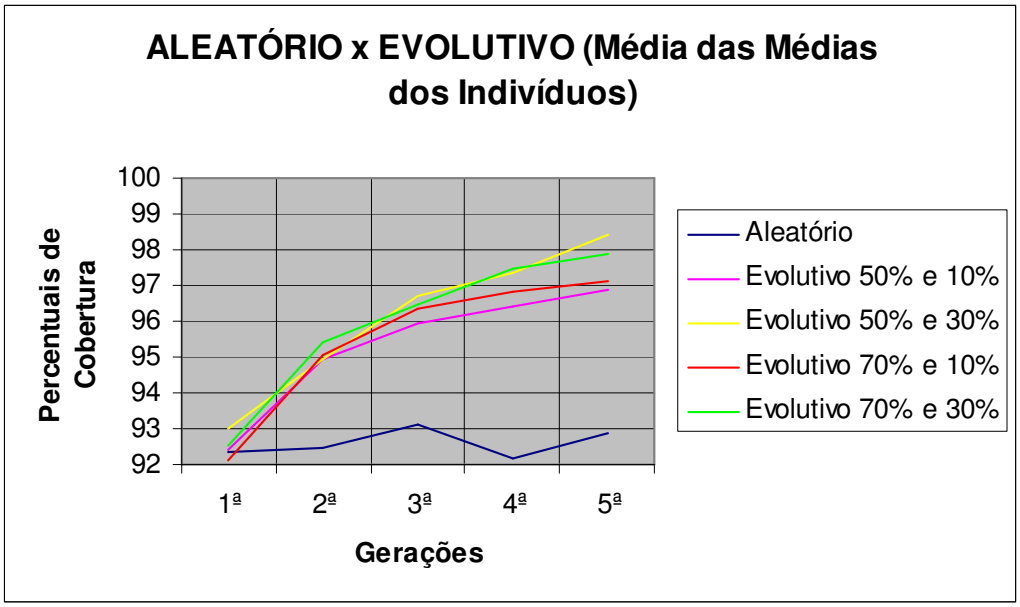

Figura 5.3 - Média das médias dos indivíduos obtidos entre as gerações, com Nº de gerações = 5 e Nº de indivíduos = 5.

Os gráficos dos resultados com 10 gerações e 5 indivíduos são mostrados nas figuras 5.4 e 5.5. Além da característica presente na configuração anterior, de oscilação dos valores dos testes aleatórios e ascensão dos valores dos testes evolutivos, nota-se que os valores máximos subiram um pouco. Isso se deve ao aumento do número de gerações, que possibilitou mais melhorias entre os indivíduos com as aplicações dos operadores genéticos. Com isso, mais indivíduos atingiram 100% de cobertura com a ajuda da mutação, e os casos de teste dos indivíduos que chegaram a esta cobertura foram melhor compartilhados entre os demais indivíduos.

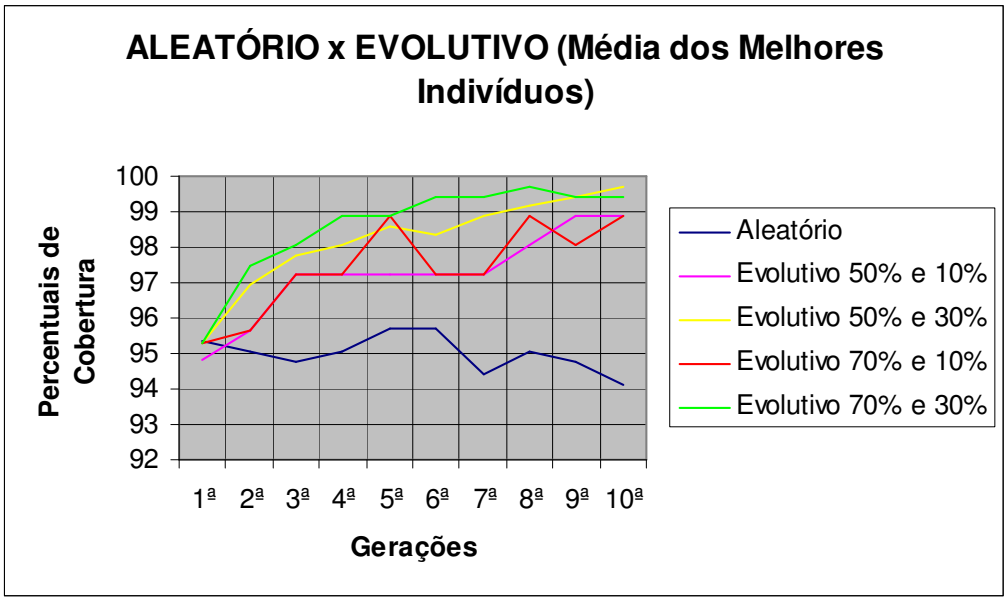

Figura 5.4 - Média dos melhores indivíduos obtidos entre as gerações, com Nº de gerações = 10 e Nº de indivíduos = 5.

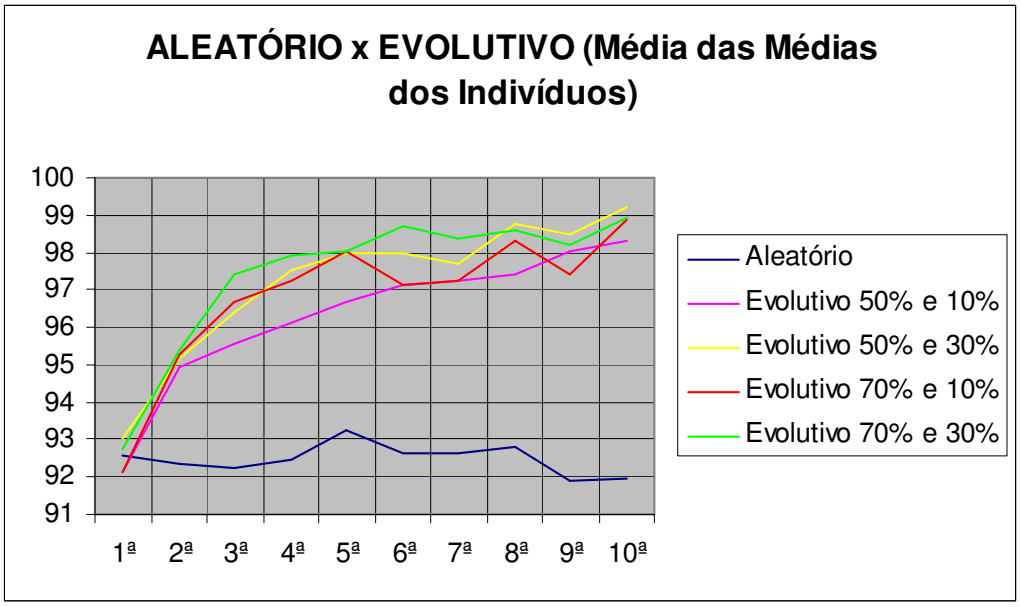

Figura 5.5 - Média das médias dos indivíduos obtidos entre as gerações, com Nº de gerações = 10 e Nº de indivíduos = 5.

Finalmente, os gráficos dos resultados com as últimas configurações, que utilizam 10 gerações e 10 indivíduos, são mostrados nas figuras 5.6 e 5.7. Nesses gráficos, nota-se que a ascensão dos valores evolutivos permanece, assim como a oscilação dos aleatórios.

Além disso, observamos que quase todas as execuções (90%) contiveram ao menos um indivíduo que atingiu 100% de cobertura.

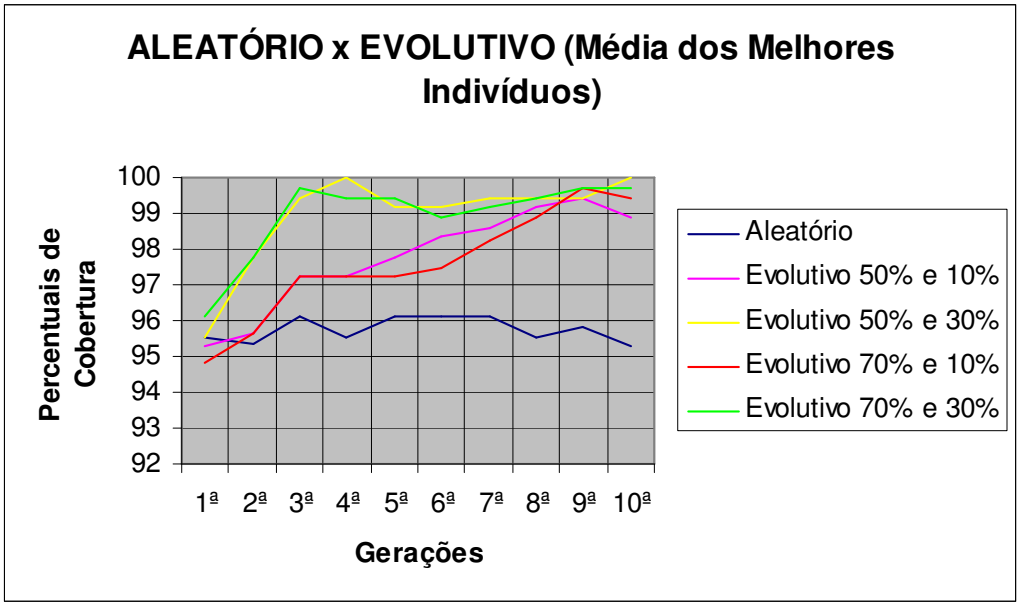

Figura 5.6 - Média dos melhores indivíduos obtidos entre as gerações, com Nº de gerações = 10 e Nº de indivíduos = 10.

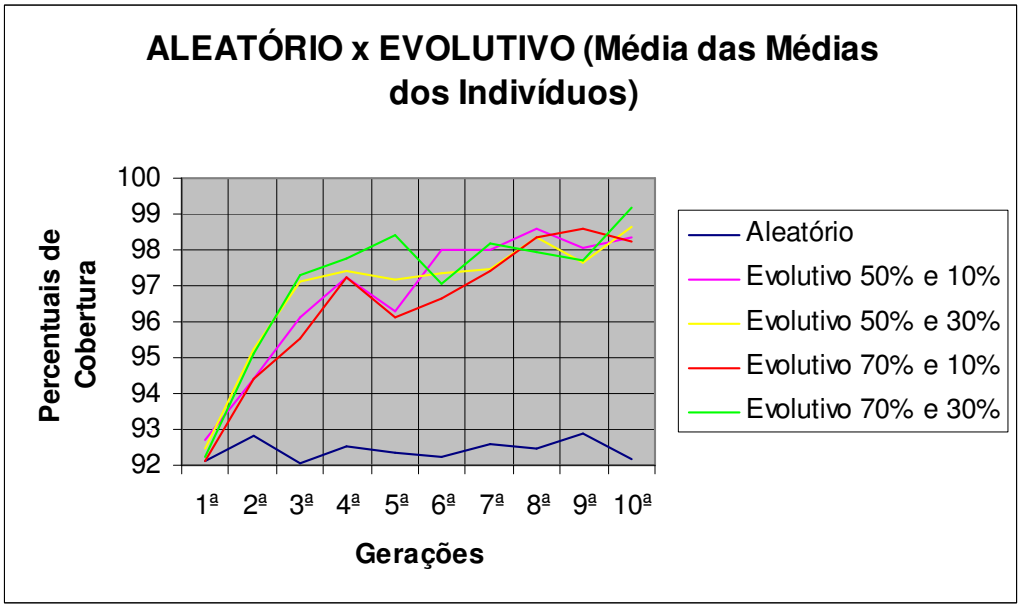

Figura 5.7 - Média das médias dos indivíduos obtidos entre as gerações, com Nº de gerações = 10 e Nº de indivíduos = 10.

Os resultados expostos nesta seção, em forma de tabela e em forma de gráfico, comprovam a superioridade do Algoritmo Genético presente no EvolUniT, quando

comparado à geração aleatória dos dados. Apesar do objeto de teste (programa *TriangleClassification*) não ser muito extenso ou complexo, e das gerações aleatórios conseguirem que determinados indivíduos atinjam até 97,22% de sua estrutura, é possível observar essa superioridade do AG quando se observa o valor máximo atingido por vários dos seus indivíduos (100%), e como a média dos indivíduos, no decorrer das gerações, forma uma curva ascendente.

Esse primeiro estudo avaliou o EvolUniT pela ótica da IA. O estudo realizado na próxima seção faz uma análise do EvolUniT considerando outras variáveis além da cobertura de condições, procurando avaliá-lo pela ótica da ES.

5.2 Estudo de Caso 2 – Teste Manual x Teste Automático

O segundo estudo de caso com o EvolUniT teve como principal objetivo validar a geração automática de código proporcionada pela ferramenta, no sentido de medir o esforço poupado por um desenvolvedor de software. Para esta comparação, serão adotados 4 critérios: (1) a cobertura de decisões, como no estudo anterior, porque esta tem que ser pelo menos tão boa quanto a conseguida de forma manual; (2) o número de casos de teste (indivíduos no caso do EvolUnit) necessários para se atingir a cobertura máxima considerada; (3) o número de gerações necessárias para se atingir a cobertura máxima, pois esta medida é diretamente proporcional ao tempo gasto para se atingir essa cobertura; e (4) a diferença do número de linhas de código entre a abordagem manual e a automática, sendo essa a mais importante para medir o esforço poupado.

As avaliações com os critérios (1), (2) e (4) podem ser feitas de forma quantitativa. O critério de número de gerações será analisado de forma qualitativa, já que na abordagem manual não há necessidade de gerações. Esta será utilizada apenas para substituir o tempo gasto para chegar ao resultado almejado, já que este não é medido nos nossos estudos.

Para medir de forma coerente o esforço poupado pela ferramenta EvolUniT, foi selecionado um sistema desenvolvido por um grupo da Motorola do Brasil, utilizado para automação de seu processo de testes. O nome da ferramenta é TaRGeT, apresentada na próxima seção.

5.2.1 Ferramenta TaRGeT

A TaRGeT (*Test and Requirements Generation Tool*) se trata de uma ferramenta baseada em MBT (*Model Based Testing*) para geração automática de casos de teste a partir de casos de uso escritos em linguagem natural [Nogueira et., al, 2007]. MBT é uma abordagem que se baseia no modelo de uma aplicação sendo testada para realizar tarefas comuns a testes, como geração de casos de teste e validação de resultados de testes [Jorgensen, 1995]. A ferramenta utiliza uma estratégia que consiste na obtenção de um modelo comportamental LTS (*Labeled Transitions Systems*) formal a partir de casos de uso, e em um algoritmo de extração para a geração dos casos de teste. Um modelo LTS fornece uma descrição global e monolítica do conjunto de todos os possíveis comportamentos do sistema, onde um caminho nesse modelo pode ser tomado como uma seqüência de teste [Cartaxo et al., 2007].

Essa ferramenta foi desenvolvida pelo grupo de pesquisa do *Brazil Test Center* [Torres et., al, 2007], que consiste em uma parceria entre o Centro de Informática (CIn) da UFPE, o Departamento de Sistemas e Computação (DSC) da UFCG, e do *Brazil Test Center* (BTC) da Motorola Industrial Ltda, com o objetivo de melhorar a qualidade de software, além de reduzir os custos de processo de testes.

Conforme dito anteriormente, essa ferramenta foi escolhida para realização do estudo de caso com foco no esforço poupado. Os motivos que levaram a essa escolha vêem do fato de que a TaRGeT é uma aplicação industrial, de fato usada pelo BTC da Motorola. Além disso, o acesso a determinadas classes de seu código fonte foi possível. As informações sobre as classes selecionadas da TarGeT para a realização deste segundo estudo estão expostas na próxima seção.

5.2.2 Objetos de Teste

Para a comparação entre os casos de teste escritos manualmente, e dos gerados de forma automática pelo EvolUniT, foram selecionadas 6 classes da TarGeT para serem os objetos de teste. Essas classes foram escolhidas por serem básicas na ferramenta TarGeT, por trabalharem com tipos primitivos e objetos String, e por possuírem loops e fluxos condicionais. O quadro 5.1 mostra as características relevantes sobre os 6 objetos de teste

para o estudo de caso, que são: o nome da classe, qual o propósito na aplicação, o número de linhas de código (LOC) e o número de métodos públicos presentes. O código das classes não pôde ser exposto nesta dissertação por motivo de política de segurança da Motorola Industrial Ltda.

Quadro 5.1 Características das classes de teste.

Classe	Descrição	LOC	Mét. Pub.
Feature	Representa uma característica de um requisito	181	9
Flow	Representa um fluxo de um caso de uso	179	9
FlowStep	Representa um passo em um fluxo de caso de uso	115	7
PhoneDocument	Representa um documento de caso de uso completo	166	10
StepId	Representa um Id de um FlowStep	126	6
UseCase	Representa um artefato de requisito de caso de uso	162	9

A próxima seção mostra os resultados atingidos pelos casos de teste escritos manualmente e os gerados automaticamente.

5.2.3 Geração dos Casos de Teste

Como dito anteriormente, os casos de teste do EvolUniT foram avaliados com ênfase apenas na capacidade do seu AG de atingir melhores coberturas que a forma aleatória. Agora o foco é avaliar se os casos de teste do EvolUniT podem substituir os escritos de forma manual por um desenvolvedor de software que participou da construção do sistema e detém conhecimento sobre o software. Para tanto, será observada tanto a cobertura de decisões atingida, quanto o número de linhas de código que o desenvolvedor teve que escrever, e o número de casos de teste e gerações do AG necessárias para se atingir a máxima cobertura conseguida. Vale lembrar que a geração de código do EvolUniT tem suas limitações quando se trata de tipos não primitivos, fazendo com que em determinados casos o usuário tenha que completar os casos de teste de forma manual.

A tabela 5.16 mostra os resultados obtidos a partir da escrita manual de classes de teste para as 6 classes do TarGeT. Essa tabela mostra respectivamente: o nome da classe de teste implementada; a cobertura de decisões máxima atingida pela classe de teste; o número de linhas de código que foram geradas pelo *wizard* do *JUnit* no Eclipse, que consiste apenas nos corpos vazios dos métodos de teste, dos métodos *setUp* e *tearDown* e os Java

Annotations acima de cada método; e as linhas de código que foram inseridas manualmente para que a cobertura máxima fosse atingida.

Tabela 5.16 - Resultados dos testes manuais nas classes do TarGeT.

Classe de Teste	Cobertura(%)	LOC Total	LOC Ger.	LOC Ins. Man.
FeatureTest	100	122	64	58
FlowTest	100	140	68	72
FlowStepTest	100	97	53	44
PhoneDocumentTest	100	127	68	59
StepIdTest	100	80	49	31
UseCaseTest	100	127	63	64

Os resultados das gerações e execuções automáticas das 6 classes escolhidas para serem testadas encontram-se na tabela 5.17, que mostra respectivamente: o nome da classe de teste gerada pelo EvolUniT; a máxima cobertura de decisões atingida, o número de gerações necessárias para atingir a cobertura máxima; o número de indivíduos que atingiram a cobertura máxima e o número de indivíduos total por geração; o número de linhas de código geradas automaticamente para a classe de teste; e o número de linhas de código que foram necessárias serem inseridas de forma manual, para completas a classe de teste.

Tabela 5.17 - Resultados dos testes automáticos nas classes do TarGeT.

Classe de Teste	Cobertura(%)	Nº Ger.	Nº Ind.	LOC Total	LOC Ger. Aut.	LOC Ins. Man.
FeatureTest	100	1	3 / 5	204	198	6
FlowTest	100	1	5 / 5	213	205	8
FlowStepTest	100	1	5 / 5	149	147	2
PhoneDocumentTest	100	1	5 / 5	360	352	8
StepIdTest	100	3	2 / 5	197	197	0
UseCaseTest	100	1	5 / 5	228	218	10

Para todos os testes o tamanho dos indivíduos foi configurado para 4, o número de indivíduos para 5, a taxa de cruzamento para 70% e a taxa de mutação para 30%. As linhas

que foram inseridas de forma manual correspondem a comandos necessários para o preenchimento dos parâmetros dos construtores das classes, quando estes possuem tipos não primitivos e / ou coleções. Trata-se de uma seqüência de instanciações de outras classes e execuções de métodos necessários para deixar objetos no estado desejado, antes de chamar o método de teste. Essa seqüência é necessária em alguns casos desde que o EvolUniT ainda não gera automaticamente dados de tipo Objeto, nem os evolui, com exceção de *Strings*.

Após as elaborações e execuções dos testes, os resultados foram analisados para comparar o esforço das duas abordagens de teste, levando em consideração as linhas de código inseridas de forma manual. Para as duas abordagens não foram levados em consideração os *test oracles*, já que tanto na abordagem automática do EvolUniT quanto na manual, com a ajuda do *wizard* do JUnit, as asserções precisam ser inseridas manualmente para a avaliação dos métodos conforme os requisitos. Também não foi levado em consideração os esforços de configuração anteriores à geração dos testes. Levando isso em consideração, a tabela 5.18 foi elaborada com base nos resultados expostos nas seções anteriores. A tabela mostra o esforço, em termos de porcentagem de linhas de código inseridas manualmente em relação ao total, tanto das gerações automáticas quanto das escritas manuais dos casos de teste.

Tabela 5.18 - Resultados dos esforços manuais das duas abordagens de teste, em termos de LOC.

Classe de Teste	Esforço Teste Manual - LOC (%)	Esforço Teste Automático -LOC (%)
FeatureTest	47,54	2,94
FlowTest	51,42	3,75
FlowStepTest	45,36	1,34
PhoneDocumentTest	46,45	2,22
StepIdTest	38,75	0
UseCaseTest	50,39	4,38

Pode-se observar que a diferença de esforço entre as duas abordagens é grande, mesmo com a utilização do wizard do JUnit que gera algumas vezes até metade das linhas de código. Para medir a diferença destes esforços em apenas um percentual, podemos fazer uma média aritmética entre os valores do teste manual e do teste automático, para as 6

classes consideradas. Isso nos dá uma média de esforço de 46,65 % de inserção manual de linhas de código na abordagem manual e uma média de 2,44 % de inserção manual na abordagem automática. Sendo assim, para este estudo de caso, foi calculada uma média de 44,21 pontos percentuais de esforço poupado quando se utiliza a ferramenta EvolUniT para a geração e execução dos testes, tendo em vista que as mesmas coberturas foram alcançadas pelas duas abordagens. O pequeno esforço manual por parte da abordagem semi-automática exige também uma análise do código sendo testado (conhecimento do testador) para que os complementos resultem em 100% de cobertura.

Com relação ao número de gerações e o número de indivíduos necessários para os resultados automáticos, não houve nenhum caso em que o esforço gasto com a re-configuração dos parâmetros do AG compensou o esforço ganho com as linhas de código. Com exceção do teste da classe StepId, que precisou de 3 gerações para que algum indivíduo (no caso 2) chegasse a 100% de cobertura, todos os outros testes só precisaram de apenas 1 geração, e todos os testes utilizaram a configuração inicial.

5.3 Estudo de Caso 3 – EvolUniT x Randoop

Este terceiro estudo de caso tem como objetivo realizar uma comparação entre a ferramenta aqui implementada e uma outra ferramenta chamada *Randoop,* que também gera testes de unidade, utilizando uma abordagem aleatória e exaustiva. Essa ferramenta foi escolhida por ser uma das ferramentas acadêmicas mais recentes criadas com o propósito de automatizar testes de unidade. Como objetos de teste serão aproveitadas as mesmas classes do *TaRGeT* utilizadas no estudo da seção 5.2.

Para esta comparação, serão adotados os seguintes critérios: a cobertura de decisões máxima atingida com os casos de teste gerados; o número de casos de teste (indivíduos no caso do EvolUnit) necessários para se atingir a cobertura máxima considerada; e o número de linhas de código inseridas de forma manual. O número de gerações necessárias, no caso do EvolUniT, não será necessária nesse estudo, porque será substituída pelo critério de tempo gasto para geração dos testes. Esse critério foi adicionado porque a ferramenta *Randoop* tem como uma de suas entradas o tempo limite para geração dos seus casos de teste.

A próxima seção descreve a ferramenta Randoop e suas características relevantes a este estudo.

5.3.1 Ferramenta Randoop

Randoop é uma ferramenta desenvolvida por [Pacheco e Ernst, 2007] com o propósito de gerar testes de unidade significativos de forma automática para classes OO, tanto implementadas na linguagem Java, como na plataforma .Net. A Randoop gera suas classes de teste utilizando uma técnica baseada em aleatoriedade e direcionamento por feedback (feedback-*directed random test generation*) [Pacheco et. al., 2006].

A Randoop tem como objetivo melhorar a eficiência de teste aleatório, e é implementada da seguinte forma: a Randoop recebe como entrada um conjunto de classes a serem testadas, um limite de tempo (em segundos) e um conjunto de "verificadores de contrato", que podem complementar os verificadores *default* já gerados pela ferramenta; e pode gerar suítes de testes como saída, uma contendo testes que violam contratos (*contract-violating tests*) e outra contento testes de regressão (*regression tests*). Os primeiros são testes que procuram exibir cenários que o código sendo testado leva à violação de algum contrato de API, como exemplo, o comportamento errôneo de algum método quando da mudança de uma versão de Java para outra. Os segundos testes, diferentemente dos primeiros, não procuram violar contratos e sim apenas capturar aspectos da implementação. Apesar dessa diferença, ambos os testes têm o propósito de achar erros no código sendo testado.

Apesar de não utilizar a cobertura como guia, a Randoop possui um importante aspecto em comum com a EvolUniT, que é a utilização de um feedback para guiar a geração aleatória, com o objetivo de evitar gerações redundantes ou entradas inválidas. Enquanto o feedback da EvolUniT é a própria cobertura de cada indivíduo, representado por um conjunto de testes executados, o feedback da Randoop é saber se a execução das seqüências de métodos previamente geradas podem ser estendidas ou não. Para tanto um vetor de *flags* é associado a uma seqüência. Cada valor da seqüência tem um valor booleano associado, que indica se pode ser usado como entrada para uma nova chamada a um método. Essas *flags* são utilizadas para "podar" o espaço de busca, de modo que o

gerador associa a *flag* de um valor para *false* se esse valor é considerado redundante ou ilegal para o propósito de criar uma nova seqüência.

Diferentemente do mecanismo de *feedback*, utilizado nas duas ferramentas, existe um aspecto importante que não é compartilhado por elas: a geração de pré-requisitos de teste. Nesse aspecto, a Randoop é vantajosa, uma vez que sua própria base é a geração de seqüências de chamadas a métodos, incluindo as seqüências que são necessárias para colocar determinados objetos, importantes para a seqüência principal, em estados desejados. Isso faz com que a Randoop seja de fato completamente automática, sem necessidade de inserção manual de código, a menos que não se consiga a cobertura desejada.

Outro aspecto importante é o domínio de geração inicial dos dados, que na Randoop também não é necessário informar como entrada. Isso evita interação do usuário com a ferramenta, mas pode ocasionar em situações em que os testes não conseguem cobrir 100% da estrutura das classes sendo testadas. Apesar do Randoop não precisar de muitas entradas, ela não é uma ferramenta integrada com o Eclipse. Sendo assim, ela precisa ser executada por linha de comando, e seus parâmetros passados também da mesma forma. Precisa ser especificado no comando o caminho e nome da(s) classes(s) a ser(em) testada(s), o tipo dos testes de saída (testes de violação ou regressão) e o tempo limite em que a ferramenta irá atuar.

Apesar de ser uma ferramenta acadêmica, a Randoop é robusta, e poderia ser utilizada como ferramenta de automação de testes de unidade em qualquer organização, seja ela de pequeno ou grande porte. Porém, apesar dela utilizar um mecanismo de feedback, ainda é uma ferramenta essencialmente de busca exaustiva, já que procura apenas melhorar a geração aleatória, mas não consegue evitar um número muito grande de testes gerados. Além disso, ela possui limitações quanto às classes de entrada. Todos esses aspectos serão vistos na próxima seção, que mostra a realização dos experimentos e comparações entre as 2 ferramentas.

5.3.2 Geração e Execução dos Testes

Nestes experimentos, foram usadas como entradas para as ferramentas as mesmas classes do TaRGeT do estudo anterior. Os experimentos na EvolUniT foram re-executados para o monitoramento do tempo de execução e evolução dos testes, em segundos. Como o limite de tempo é um dos parâmetros de entrada da *Randoop*, esse valor foi especificado de acordo com o tempo levado para execução de cada experimento na EvolUniT, para observação do quanto é coberto por cada ferramenta com os mesmos intervalos de tempo.

Para a medição do tempo usando a EvolUniT, foi levado em consideração apenas a segunda etapa de utilização da ferramenta, que consiste nas sucessivas execuções e evoluções dos dados gerados inicialmente, já que a primeira etapa exige a interação do usuário. O mesmo foi levado em consideração para as entradas da ferramenta Randoop. Além disso, a Randoop não possui a medição da cobertura integrada à ferramenta, como ocorre na EvolUniT, e o tempo usado para medir essa cobertura após a geração dos testes não foi considerado.

Como ocorreu no estudo anterior, para todos os testes o tamanho dos indivíduos foi configurado para 4, o número de indivíduos para 5, o número máximo de gerações para 5, a taxa de cruzamento para 70% e a taxa de mutação para 30%. Para efeitos de igualdade neste estudo, não foi inserido código manualmente nos testes gerados pelo EvolUniT. A tabela 5.19 mostra os resultados dos experimentos realizados. A primeira coluna mostra o nome da classe sendo testada; a segunda mostra o percentual das coberturas de decisões atingidas pelos testes gerados por cada ferramenta; a terceira mostra o número de classes de teste gerada e o número de métodos de teste de cada classe, no caso da Randoop, e o número de indivíduos (arquivos XML de dados de entrada) e o tamanho de cada indivíduo (número de conjuntos de parâmetros de entrada), no caso da EvolUniT; e a quarta e última coluna mostra o tempo que durou a execução das duas ferramentas para cada experimento. O tempos são os mesmos para as 2 ferramentas, pois uma das entradas da Randoop é o tempo em segundos em que a ferramenta ficará gerando testes para determinada classe, e cada entrada dessa foi baseada no tempo levado pela EvolUniT para executar e e evoluir seus testes.

No caso do EvolUniT, os testes paravam de rodar quando 100% de cobertura era atingido, em qualquer geração. O Randoop parava a execução de acordo com o tempo informado como entrada na linha de comando.

Tabela 5.15 – Resultados das execuções nas duas ferramentas

Classe de entrada	Cobertura máxima aproximada (%)		Número / Tamanho dos casos de teste gerados		Tempo decorrido (s)
	Randoop	EvolUniT	Randoop	EvolUniT	
Feature	9,3	66,3	1 / 13	5 / 4	30,98
Flow	4,2	69,2	1 / 11	5 / 4	25,16
FlowStep	33,3	100	1 / 14	5 / 4	9,39
PhoneDocument	79,3	79,3	21 / 500	5 / 4	31,37
StepId	84,0	100	7 / 500	5 / 4	7,95
UseCase	16,7	83,3	1 / 16	5 / 4	29,73

Ao se observar a tabela 5.15, é possível destacar 2 pontos importantes: os resultados de cobertura da EvolUniT não foram todos 100%, uma vez que não foram realizadas inserções manuais complementares dos testes; e há grandes diferenças entre os resultados de cobertura da Randoop. A primeira observação mostra como a ferramenta depende da interação com o usuário para alguns casos, por isso sendo chamada de ferramenta de semi-automação. A segunda observação deve-se a uma limitação da ferramenta Randoop. Para os casos em que a cobertura foi muito baixa com a Randoop (os testes gerados para as classes Feature, Flow e UseCase) o comando de geração de testes de regressão não conseguiu gerar seqüências, que é a base da ferramenta, fazendo com que nenhuma classe de teste fosse gerada.

Após habilitar uma opção da ferramenta chamada "*null-allowed*", que faz com que a Randoop possa usar valores *null* como entradas para métodos (apenas se não há valores não nulos disponíveis), a ferramenta conseguiu gerar seqüências e conseqüentemente classes de teste. Entretanto, várias das seqüências geradas incluíam a atribuição do próprio objeto de chamada ao método de teste ao valor nulo, ocasionando assim uma *NullPointerException* na própria chamada ao método. Esse comportamento da ferramenta explica a baixa cobertura atingida com a utilização da mesma para gerar testes para algumas das classes da TaRGeT.

Outros dados a serem considerados são os números de classes de testes geradas e seus tamanhos (número de métodos de teste contidos). É notório que a ferramenta Randoop consegue gerar muitos testes em um pequeno intervalo de tempo. As gerações para as classes *PhoneDocument* e *StepId* ilustram esse comportamento. Nos casos em que ela não gerou tantos testes, seu mecanismo de feedback indicou que não era mais necessário, fazendo com que o tempo total utilizado para a geração fosse também maior do que o necessário. Isso ocorreu nos casos das classes *Feature*, *Flow*, *FlowStep e UseCase*.

Também é importante frisar que a Randoop gera *test oracles* para seus testes de regressão e também para os testes de violação. Com isso, seus métodos de teste já possuem as asserções necessárias, poupando mais trabalho ao usuário. Esse é um fator que comprova a robustez da ferramenta, quando se comparada à EvolUniT.

Apesar de a Randoop ser uma ferramenta robusta e com um poder de geração de testes razoável, incluindo a geração de *test oracles*, surpreendentemente ela não foi muito eficiente com relação ao corpus de teste selecionado para este experimento. De fato, a Randoop conseguiu excelentes resultados em classes da biblioteca padrão de java [Pacheco e Ernst, 2007], como é o caso de *java.util.Collections* e *java.util.TreeSet*, que possuem grandes quantidades expressões condicionais e laços em suas linhas de código. A EvolUniT, embora ainda com várias limitações, conseguiu resultados razoáveis, mesmo sem a interação do usuário com a ferramenta após a geração das classes de teste. Sendo assim, para o estudo de caso aqui realizado, pode-se afirmar que o mecanismo de geração de código e evolução dos dados presente na EvolUniT foi superior ao mecanismo de geração de código e feedback presente na Randoop, no que diz respeito à maximização dos percentuais de cobertura, conforme os resultados da tabela 5.15. Por outro lado, a Randoop se mostrou uma ferramenta com melhor desempenho do que a EvolUniT, em se tratando do tempo que leva para gerar uma grande quantidade de testes, incluindo a utilização de seu mecanismo de feedback. Isso também pode ser observado na tabela 5.15, com a quantidade de classes e de métodos de teste por classe gerados pela Randoop no mesmo tempo gasto pela EvolUniT.

Na próxima seção são expostas algumas considerações sobre os três estudos de caso aqui apresentados.

5.4 Considerações Finais

Neste capítulo, apresentamos três estudos realizados para avaliação da ferramenta EvolUniT e sua abordagem de Teste Evolutivo. O primeiro estudo foi experimental, e teve o foco na validação do Algoritmo Genético implementado para evoluir os dados de entrada, gerados inicialmente de forma aleatória pela EvolUniT. Nesses experimentos foram realizadas duas baterias de teste, sendo a primeira com dados gerados de forma aleatória para cada geração, e a segunda com evolução dos dados utilizando o AG da ferramenta. Foi visto que, apesar de os testes aleatórios conseguirem atingir um razoável nível de cobertura, os valores adquiridos por esse tipo de teste permaneciam oscilando através das gerações, chegando a um valor máximo de 97,22% de cobertura. Esse valor, que é relativamente alto, se deve à baixa complexidade e tamanho do objeto de teste utilizado nos experimentos.

Diferentemente dos aleatórios, os testes evolutivos fizeram os valores de cobertura começarem a crescer a partir da segunda geração e chegar a valores tão bons ou melhores que os atingidos nos testes aleatórios, dependendo das configurações estabelecidas. Das configurações testadas, em se tratando de operadores genéticos, as melhores foram as combinações de taxa de crossover igual a 70% e taxa de mutação igual a 30%, e de taxa de crossover de 50% e taxa de mutação de 30%. Em relação ao número de gerações e de indivíduos, os melhores resultados foram conseguidos com 10 gerações e 10 indivíduos. Isso se deve ao foto de que com mais indivíduos, a probabilidade de um ou mais atingirem 100% de cobertura com a aplicação de mutação ficou maior, e com mais gerações o compartilhamento dos bons conjuntos de dados (aqueles que percorreram toda a estrutura do objeto de teste) foi maior através da aplicação de cruzamento. Apesar de a diferença do melhor valor atingido pelos testes aleatórios (97,22%) e do melhor valor atingido pelos testes evolutivos (100%) ser pequena, nota-se a superioridade do AG quando da observação das médias dos valores de cobertura atingidos pelos indivíduos.

O segundo estudo realizado utilizou classes de uma ferramenta industrial chamada TaRGeT, desenvolvida pela equipe de pesquisa do CIn / BTC da Motorola. Esse estudo teve o foco na avaliação do ganho adquirido com a utilização do EvolUniT, em termos de esforço poupado com a semi-automação da ferramenta. Para a avaliação desse esforço

poupado, foram consideradas a escrita manual e a geração automática dos testes de unidade de 6 classes da TarGeT. Diante desse cenário, foi visto que cerca de 44,21 pontos percentuais de esforço (em linhas de código) foi poupado quando da utilização da EvolUniT para gerar testes de unidade.

Foi visto também no segundo estudo de caso que, apesar da facilidade provida pela EvolUniT, em alguns casos é preciso completar os casos de teste com linhas de código complementares inseridas de forma manual. Isso se deve ao fato de que a EvolUniT não possui ainda um mecanismo para gerar os pré-requisitos (seqüências de instanciações e chamadas a métodos) necessários para deixar objetos em um estado apropriado para testar outros métodos que os utilize como parâmetros. Por enquanto a EvolUniT preenche apenas tipos primitivos e *Strings* com os dados gerados pelo *Test Case Generator* e evoluídos pelo *GA Component,* e atribui objetos a valores nulos. Apesar de o esforço manual ser mínimo na abordagem automática (cerca de 2,44 % para o cenário considerado), um dos trabalhos futuros é evoluir a geração de código para que não seja necessária nenhuma inserção manual de código. Isso inclui também a geração automática de *test oracles* para a avaliação da lógica dos métodos criados.

O terceiro estudo foi outro estudo de caso, eu que se realizou uma comparação entre a EvolUniT e outra ferramenta de geração de testes de unidade, a Randoop. Para este estudo também foram consideradas como corpus de teste as 6 classes da TaRGeT utilizadas no segundo estudo. Na comparação realizada, foi visto que a Randoop não conseguiu produzir bons casos de teste para todos os casos, baixando assim a cobertura atingida para esses casos. Para este estudo, a EvolUniT não contou com a inserção manual de código que ocorreu no segundo estudo, e mesmo assim conseguiu resultados razoáveis de cobertura. Apesar de nesse estudo a EvolUniT ter atingido melhores coberturas das classes sendo testadas, pôde-se observar que a Randoop possui robustez e performance melhores. Além de também gerar *test oracles* nos seus métodos de teste, ela consegue gerar uma grande quantidade de testes em pouco tempo, incluindo o seu mecanismo de feedback.

O próximo capítulo apresenta as conclusões e trabalhos futuros a respeito do trabalho de mestrado aqui desenvolvido.

6 Conclusões

Nesse trabalho, foi investigado o uso de Algoritmos Evolutivos para automatizar o processo de testes de software, com suas diversas abordagens e fases. Com o passar do tempo e o aumento dos sistemas gerados a partir dessa integração, esses testes automatizados passaram a ser chamados de "Testes Evolutivos".

Nesta dissertação, foram apresentadas várias formas de aplicação de Teste Evolutivo, tendo em vista a abordagem de teste sendo executada, a fase de teste em questão e seus respectivos critérios de avaliação. Além disso, foram apresentadas outras técnicas e frameworks que, assim como Algoritmos Genéticos, também já foram utilizadas como ferramentas para auxiliar no processo de testes de software. Em se tratando de Teste Evolutivo, através dos trabalhos desempenhados nessa linha de estudo, foi visto que o mesmo pode ser dividido em duas categorias: Teste Evolutivo Convencional (TEC) e Orientado a Objetos (TEOO). Tendo em vista essa divisão, implementamos uma ferramenta que possui características tanto de TEC quanto de TEOO, ferramenta esta chama de EvolUniT (*Evolutionary Unit Testing*).

A EvolUniT foi criada com o objetivo de automatizar e melhorar o processo de testes de unidade de qualquer sistema escrito em java e tendo sido desenvolvido através da plataforma Eclipse. Três estudos foram realizados em cima da ferramenta, sendo um estudo experimental e os outros dois estudos de caso. O primeiro se tratou de uma comparação da utilização do Algoritmo Genético implementado na EvolUniT com a aplicação de testes aleatórios (*Random Testing*); o segundo de uma estimativa de esforço poupado com a utilização da EvolUniT, em contraste com a escrita manual de testes de unidade; e o terceiro se tratou da comparação entra a EvolUniT e a Randoop, uma ferramenta de geração automática aleatória de testes de unidade. Os resultados dos três estudos foram promissores.

A seguir apresentamos um resumo das principais contribuições deste trabalho (seção 6.1), e alguns trabalhos futuros (seção 6.2). As considerações finais serão expostas na seção 6.3.

6.1 Resumo das Contribuições

A seguir apresentamos um resumo das principais contribuições deste trabalho de mestrado.

- Resumo das principais abordagens, fases e critérios de Teste de Software: No conteúdo desta dissertação, apresentamos os conceitos de: Engenharia de Software, modelo de desenvolvimento de software e processo de software. Esses conceitos serviram como ponto de partida para chegarmos em um dos temas em questão nesta dissertação, que são os Testes de Software. No capítulo 2, que foi o reservado para exploração desse tema, foram apresentadas: as abordagens de teste mais utilizadas (caixa preta, caixa branca, caixa cinza e baseado em falhas), as fases de teste em um processo interativo incremental (unidade, integração, sistema e aceitação) e os critérios de avaliação ou análise de testes mais utilizados (tempo decorrido, cobertura de código, falhas encontradas...). Foi visto no mesmo capítulo que nenhuma abordagem de teste é melhor que a outra, mas sim elas se complementam. Isso também vale para as fases e critérios. Apesar dessa idéia em mente, foi dado um enfoque na abordagem estrutural ou caixa branca, na fase de testes de unidade e nos critérios mais relacionados com essa combinação.

- A apresentação de Teste Evolutivo e outras técnicas e frameworks utilizados para automação de testes de software: No decorrer da dissertação, mais especificamente no capítulo 3, foi introduzido o conceito de Teste Evolutivo, que tem o objetivo de melhorar a qualidade e de diminuir custos no processo de testes, através da aplicação de um alto nível de automação desse processo [Baresel et. al., 2002]. Foi visto que, dentre os Algoritmos Evolutivos, os mais utilizados são os Algoritmos Genéticos, que tiveram uma seção dedicada a eles na dissertação. Vários trabalhos anteriores de Teste Evolutivos também foram mostrados no mesmo capítulo. Esses trabalhos foram mostrados de forma separada, pela abordagem de teste de software em questão. Posteriormente, suas principais características foram analizadas, como: a função objetivo utilizada pelo AE, os critérios utilizados para avaliação dos testes e os resultados obtidos. Além de Teste Evolutivo, também foram mencionadas

outras técnicas de otimização baseadas em busca metaheurística como Subida ne Encosta (*Hill-Climbing*) e Têmpera Simulada (*Simulated Annealing*), para uma posterior explicação do porque da escolha de AGs para este trabalho.

- Uma ferramenta de Teste Evolutivo utilizando uma abordagem híbrida: Nesta dissertação, implementamos uma ferramenta, mais precisamente um plugin para a plataforma Eclipse, que procurou reunir características de TEC e TEOO. Essa ferramenta, chamada de EvolUniT, foi apresentada no capítulo 4 e tem o objetivo de efetuar a geração automática de casos de testes escritos em java, utilizando o framework de testes *JUnit*. Além disso ela executa diversas vezes os casos de teste gerados, com o objetivo de evoluir os dados que fazem parte desse casos de teste, através de um Algoritmo Genético. Com essa evolução, a ferramenta procura atingir altos índices de cobertura de decisões (branch coverage), que foi o critério de avaliação adotado após a análise das opções disponíveis. Apesar de já haver vários trabalhos de TE, poucos foram realizados para softwares orientados a objetos, e desses, não foi visto nenhum que evoluísse apenas os dados gerados, ao invés de todas as sequências de comandos. A abordagem aqui proposta, utiliza a simplicidade da abordagem convencional em um ambiente OO, aproveitando o conhecimento do desenvolvedor / testador quando necessário. Além disso, a função objetivo utilizada no AG da EvolUniT também partiu de uma nova proposta. Isso se deve ao fato de que ela utiliza como parte de sua fórmula o resultado total de cobertura atingida por cada indivíduo (conjunto de casos de teste), ao invés de adotar uma busca isolada para cada expressão condicional (branch) do código, como ocorre na maioria dos trabalhos de TE [Bottacci, 2002] [Hermadi e Ahmed, 2003] [Tonella, 2004].

- Avaliação da abordagem híbrida de Teste Evolutivo: No capítulo 5 desta dissertação, foi montado um estudo experimental para a comparação dos resultados obtidos quando os dados de entrada eram gerados pelo AG implementado na EvolUniT, com os resultados obtidos quando os dados de entrada eram gerados de forma eleatória. O objetivo deste experimento foi verificar se o AG nesse contexto superava os testes aleatórios, e em quanto os superava. O experimento foi

conduzido em apenas um objeto de teste (uma classe Java com apenas um método) e os resultados alcançados foram satisfatórios.

- Avaliação do esforço poupado: No capítulo 5, também foi realizado um estudo de caso foi realizado em cima de 6 objetos de teste (classes Java) de uma ferramenta comercial, chamada TaRGeT. O objetivo desse estudo de caso era verificar o quanto de esforço seria poupado, para um desenvolvedor de software, quando da utilização da EvolUniT para gerar automaticamente seus testes de unidade. Esse esforço foi estimado levando-se em consideração apenas a quantidade de linhas de código necessárias para a escrita manual de um caso de teste, e necessárias para complementar um caso de teste gerado de forma automática pela EvolUniT. Os resultados foram animadores.
- Comparação entre ferramentas: Ainda no capítulo 5 desta dissertação, um outro estudo foi realizado, sendo este a comparação entre a EvolUniT e uma outra ferramenta de automação de testes de unidade, chamada Randoop. O objetivo do estudo era monitorar as coberturas atingidas e os tempos de execução de cada ferramenta, em cima do mesmo corpus de teste do estudo de caso anterior, que são as 6 classes da TaRGeT. Para esse ambiente motado, a EvolUniT atingiu melhores coberturas, enquanto a Randoop mostrou mais rapidez na geração de casos de teste.

6.2 Limitações e Trabalhos Futuros

Apesar dos bons resultados adquiridos com o trabalho realizado, ainda há vários pontos a melhorar na ferramenta evolutiva implementada. Do ponto de vista da utilização de uma técnica de otimização evolutiva, que é o caso dos AGs, para resolver um problema de ES, que no caso consiste em maximizar a cobertura de testes gerados de forma automática, o objetivo aqui foi atingido. Apesar dessa "comunhão" entre testes e AGs já ter sido realizada anteriormente em diversos trabalhos, a abordagem aqui utilizada foi diferente e igualmente eficiente, considerando os estudos realizados no capítulo 5. Porém, pensando mais pelo lado da ES, do ponto de vista da ferramenta evolutiva ser de fato utilizada como produto por qualquer instituição, seja comercial ou acadêmica, algumas melhorias

necessárias foram identificadas para serem adicionadas à EvolUniT. Essas melhorias são explicadas a seguir.

- Evolução de dados de qualquer tipo: a EvolUniT evolui apenas dados de tipos primitivos e *Strings*, podendo assim garantir bons resultados a apenas certos tipos de entradas (classes cujos métodos tratem mais com tipos primitivos). Com a evolução de objetos de qualquer tipo e coleções a ferramenta conseguiria bons resultados para qualquer tipo de entrada.

- Geração automática de pré-condições: atualmente a ferramenta não coloca objetos dependentes em estado apropriado antes das chamadas dos métodos, em um método de teste. Isso requer também uma análise mais profunda dos métodos das classes sendo testadas, para além de saber qual objeto instanciar, saber também quais métodos chamar desse objeto dependente para que determinados pontos no código sejam atingidos, ou seja, deixar o objeto no estado desejado para que as pré-condições do teste sejam efetuadas. A maior parte do esforço manual ainda demandado, mesmo com a utilização do EvolUniT, vem dessa pendência.

- Aperfeiçoamento da interface com o usuário: a interface do sistema exige que o usuário insira valores de entrada diretamente em arquivos de propriedades e arquivos XML, como é o caso das configurações do algoritmo genético e os valores de domínios para a geração dos dados da primeira geração, respectivamente. A interface pode ser mais amigável, com a utilização de *wizards* para facilitar as configurações necessárias antes da utilização propriamente dita do plug-in.

- Geração automática do domínio dos dados: O domínio dos dados gerados inicialmente de forma aleatória na primeira geração é definido pelo usuário, através de um arquivo XML. Se esse domínio fosse calculado de forma automática, com a utilização de alguma técnica de análise estática, como execução simbólica [King, 1976], a automação da ferramenta seria mais completa.

- Geração Automática de *Test Oracles*: Apesar do EvolUniT se voltar apenas para o objetivo de maximizar a cobertura através de classes de teste geradas, os métodos dessas classes só são considerados de fato métodos de teste quando da inserção de T*est Oracles*. No caso do framework JUnit, os *Test Oracles* são representados por

comandos que são chamados de asserções (*assertions*), e servem para verificar se o funcionamento dos métodos está correto, conforme os requisitos. A inserção desses comandos, também de forma automática, é um outro desafio que também tornaria a EvolUniT uma ferramenta mais completa, em termos de automação.

6.3 Considerações Finais

A automação de testes usando técnicas de busca e otimização, como Algoritmos Evolutivos, tem sido bastante estudada ao longo das últimas duas décadas, porém, ainda há vários desafios a serem transpostos nessa área. Nessa dissertação, resumimos a aplicação da técnica de Teste Evolutivo e procuramos dar a nossa contribuição, através de uma ferramenta que aplica essa técnica. Esperamos que essa dissertação possa servir como fonte de consulta e estudo para pessoas interessadas em automação de testes usando técnicas de IA, e que a ferramenta implementada neste trabalho possa ser aproveitada por profissionais e estudantes da área.

Apêndice A

Código da classe ExtendedHelloWorldTest gerada pelo EvolUniT

Esta classe é usada para testar a classe EstendedHelloWorld apresentada como exemplo. Ela usa o framework JUniT e herda os métodos da classe TestCase presentes no mesmo. Esses métodos são sobrescritos pela classe de teste gerada. O método *setUp* contém a inicialização do acesso aos arquivos XML que representam os indivíduos, além da inicialização da classe sendo testada. O método *tearDown* é gerado sem corpo mas fica disponível para que o usuário o utilize caso queira efetuar "limpezas" de variáveis. Além desses são gerados métodos de teste para cada método público da classe sendo testada. Os nomes desses métodos possuem o prefixo *test*, para serem reconhecidos como métodos de teste pelo JuniT. Neste caso o único método de teste é o *testExtendedHello*, que tem o propósito de testar o método público *extendedHello* várias vezes, utilizando diferentes conjuntos de parâmetros presentes nos arquivos XML (indivíduos).

```
package teste.builder;

import junit.framework.*;
import java.io.FileNotFoundException;
import org.jtestcase.JTestCase;
import org.jtestcase.TestCaseInstance;
import org.jtestcase.JTestCaseException;
import java.util.Vector;
import java.util.HashMap;

/**
 * JUnit test suite for ExtendedHelloWorldTest
*/
public class ExtendedHelloWorldTest extends TestCase {
      //declare reusable objects to be used across multiple tests

      private JTestCase jtestcase = null;
      private String dataFile;
      private String folder;
      protected static int generation = 0;
      protected static int execution = 0;
      protected static int testCasesSize = 0;

      private ExtendedHelloWorld extendedHelloWorld;

      public ExtendedHelloWorldTest(String name) {
            super(name);
      }
      protected void setUp() {
            //define reusable objects to be used across multiple tests

            //change constructor if necessary and prepare pre-conditions
            String folder =
            "C:/Eclipse/runtime-EclipseApplication/Teste/src/individuals";
            dataFile =
```

```
            folder + "/ExtendedHelloWorldTest-data" + generation +
            execution + ".xml";
      try {
            jtestcase = new JTestCase(dataFile, "ExtendedHelloWorld");

                  extendedHelloWorld = new ExtendedHelloWorld();

            } catch (FileNotFoundException e) {
            e.printStackTrace();
            } catch (JTestCaseException e) {
            e.printStackTrace();
            } catch (Exception e){
            e.printStackTrace();
      }
}
protected void tearDown() {
//clean up after testing (if necessary)

}
public void testExtendedHello() {
      if (jtestcase == null)
            fail("Couldn´t read xml definition file");

      final String METHOD = "extendedHello";
      Vector testCases = null;

      try {
            testCases =
                  jtestcase.getTestCasesInstancesInMethod(METHOD);

      } catch (JTestCaseException e) {
            e.printStackTrace();
            fail("error parsing xml file for calculate method +
                  METHOD");
      }

      testCasesSize = testCases.size();

      for (int i = 0; i < testCases.size(); i++) {
            // Retrieve name of test case
            TestCaseInstance testCase =
                        (TestCaseInstance) testCases.elementAt(i);

            try{
                  HashMap params = testCase.getTestCaseParams();

                  String text = ((String)params.get("text"));
                  boolean show =
                       ((Boolean)params.get("show")).booleanValue();

                  extendedHelloWorld.extendedHello(text, show);

            } catch (Exception ex){
                  ex.printStackTrace();
                  fail("An error ocurred: + ex.getMessage()");
            }
      }
```

```
            }
        }
```

Apêndice B

Código da classe ExtendedHelloWorldTestSuite gerada pelo EvolUniT

Esta classe é usada como suporte para testar várias vezes a classe de exemplo *ExtendedHelloWorld*. Ela é responsável por executar várias vezes os métodos de teste da classe de teste gerada, simulando assim as várias gerações de um Algoritmo Genético. O método *getTestMethods* é responsável por capturar todos os métodos de teste da classe de teste gerada. O método *suite* é responsável por adicionar os métodos de teste a uma bateria de testes uma ou mais vezes, dependendo do número de execuções (gerações) consideradas. O método *setUp* nesse caso inicializa a classe de teste. O método *testAll* é responsável por executar todos os testes da suite e gerar eventos necessários para a captura das coberturas de forma isolada, para cada arquivo XML de dados de teste.

```
package teste.builder;

import java.lang.reflect.Method;
import java.util.Enumeration;
import java.util.Iterator;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestFailure;
import junit.framework.TestResult;
import junit.framework.TestSuite;
import jp.co.dgic.eclipse.jdt.internal.coverage.ui.CoverageTestRunListener;
import jp.co.dgic.eclipse.jdt.internal.coverage.ui.TestIterationEvent;

/**
 * JUnit test case for ExtendedHelloWorldTest
 */

public class ExtendedHelloWorldTestSuite extends TestCase {

      private static ExtendedHelloWorldTest extendedHelloWorldTest;

      public void testAll() {

            Test allTests = ExtendedHelloWorldTestSuite.suite();
            TestResult results = new TestResult();
            int generations = 2;
            int populationSize = 2;

            for(int i=0; i<generations;i++){
              CoverageTestRunListener.getInstance().suiteIterationStarted(
              new TestIterationEvent(
                  i,TestIterationEvent.SUITE_ITERATION_STARTED));

                  for(int j=0; j<populationSize;j++){
                        allTests.run(results);
                        CoverageTestRunListener.getInstance().
```

```
                        testIterationEnded(
                              new TestIterationEvent(
                              i,
                              j,
                              TestIterationEvent.TEST_ITERATION_ENDED,
                              "ExtendedHelloWorldTest"));

                        ExtendedHelloWorldTest.execution++;
                  }

                  ExtendedHelloWorldTest.execution = 0;
                  CoverageTestRunListener.getInstance().suiteIterationEnded(
                        new TestIterationEvent(
                              i,
                              TestIterationEvent.SUITE_ITERATION_ENDED,
                              "ExtendedHelloWorldTest",
                              ExtendedHelloWorldTest.testCasesSize));

                              ExtendedHelloWorldTest.generation++;
            }

            Enumeration<TestFailure> fails = results.errors();

            while (fails.hasMoreElements()){
                  TestFailure fail = fails.nextElement();
                  System.out.println(fail.trace());
            }
      }
      //suite() returns a collection of tests  that can be added to
      //another test suite.
      public static Test suite() {

            TestSuite suite = new TestSuite("ExtendedHelloWorld tests");

            Method[] methods = getTestMethods();
            int qtdExecutions = 2;
            int qtdTestMethods = methods.length;
            String methodName;
            Method method;

            for (int i=0; i<qtdExecutions; i++){
                  for (int j=0; j<qtdTestMethods; j++){
                        if (methods[j]!=null){
                              method = (Method)methods[j];
                              methodName = method.getName();
                              suite.addTest(
                                  new ExtendedHelloWorldTest(methodName));
                        }
                  }
            }

            return suite;
      }
      protected void setUp() {
      //define reusable objects to be used across multiple tests

            if (extendedHelloWorldTest == null)
```

```
                extendedHelloWorldTest = new
                ExtendedHelloWorldTest("extendedHelloWorldTest");
        }
        private static Method[] getTestMethods(){

                Method[] methods =
                        extendedHelloWorldTest.getClass().getMethods();
                Method[] testMethods = new Method[methods.length];
                Method method;

                for (int i=0; i<methods.length; i++){

                        method = methods[i];
                        if (method.getName().startsWith("test")){
                                testMethods[i] = method;
                        }
                }

                return testMethods;
        }
}
```

Apêndice C

Código da classe TriangleClassification

Esta classe é usada como objeto de teste no primeiro estudo de caso realizado neste trabalho. A classe possui apenas um método (*getTriangleType*) que utiliza operadores relacionais de igualdade e desigualdade para determinar se um polígono de 3 lados dado como entrada, através de 3 parâmetros inteiros, é ou não um triângulo. Se for, o método também o classifica como eqüilátero, isósceles ou escaleno.

```
public class TriangleClassification {

	private final int NOT_A_TRIANGLE = 0;
	private final int SCALENE = 1;
	private final int EQUILATERAL = 2;
	private final int ISOSCELES = 3;

	public int getTriangleType(int a, int b, int c){

		int type;
		int aux;

		if (a > b){

			aux = a;
			a = b;
			b = aux;
		}
		if (a > c){

			aux = a;
			a = c;
			c = aux;
		}
		if (b > c){

			aux = b;
			b = c;
			c = aux;
		}

		if (a + b <= c){
			type = NOT_A_TRIANGLE;
		}
		else{
			type = SCALENE;

			if (a == b && b == c){

				type = EQUILATERAL;
			}
			else if (a == b || b == c){
```

```
                        type = ISOSCELES;
                }
            }

            return type;
        }
    }
```

Referências

[Agustin, 2003] Agustin, J. M., Improving software quality through extreme coverage with JBlanket, M.S. Thesis CSDL-02-06, *Department of Information and Computer Sciences*, University of Hawaii, Honolulu, 2003.

[Baresel et al., 2002] Baresel, A., Sthamer, H., Wegener, J.. Suitability of Evolutionary Algorithms for Evolutionary Testing. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, Oxford, England. August 26-29, 2002.

[Baresel et al., 2003] Baresel, A., Pohlheim, H., Sadeghipour, S., Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), pp. 2428-2441, Chicago(IL), USA, July 2003.

[Baresel et al., 2004] Baresel, A., Binkley, D., Harman, M., Korel, B., Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. ISSTA, pp. 108-118, ACM, 2004.

[Beck, 1999] BECK, K., Extreme Programming Explained, Addison Wesley, 1999.

[Beizer, 1996] Beizer, B., Software System Testing and Quality Assurance, Thomson Computer Press, 1996.

[Binder, 1999] Binder, R. V., Testing Object-Oriented Systems: Models, Patterns, and Tools. Volume1, Addison Wesley Longman, Inc. 1999.

[Booch et al., 1999] Booch, G., Jacobson, I,, Rumbaugh, J., The Unified Software Development Process. Reading – MA: Addison-Wesley, 1999.

[Bottacci, 2002] Bottaci, L., Instrumenting programs with flag variables for test data search by genetic algorithm, *In Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO 2002), pages 1337 - 1342, New York, USA, 2002.

[Buehler e Weneger, 2003] Buehler, O., Wegener, J., Evolutionary functional testing of an automated parking system. International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis. Orlando, Florida, USA, 2003.

[Buehler e Weneger, 2005] Buehler O., Wegener J., Evolutionary Functional Testing of a Vehicle Brake Assistant System, Proceedings of the 6th Metaheuristics International Conference (MIC2005), Vienna, Austria, August 2005.

[Cartaxo et al., 2007] Cartaxo, E. G., Neto, F. G. O., Machado, P. D. L.. Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems. To

appear in: IEEE International Conference on Systems, Man and Cybernetics, 2007, Montreal. IEEE International Conference on Systems, Man and Cybernetics, 2007.

[Cavalcanti e Gaudel, 2007] Cavalcanti, A., Gaudel, M.C, Testing for Refinement in CSP, In *Formal Methods and Software Engineering: 9th Internation Conference on Formal Engineering Methods*, ICFEM 2007, Florida, USA, Novembrer 15-15, 2007.

[Cheon et al., 2005] Cheon, Y., Kim, M., Perumendla, A., A complete automation of unit testing for java programs. In H. R. Arabnia and H. Reza, editors, Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Volume I, Las Vegas, Nevada, June 27-29, 2005, pages 290--295. CSREA Press, 2005.

[Clark et al., 1998] Clark, J., Tracey, N., Mander, K.. Automated program flaw finding using simulated annealing. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, volume 23, pages 73-81, March 1998.

[Clover] Clover: A Code Coverage Tool for Java, http://www.atlassian.com/software/clover/.

[Cobertura] Cobertura, a free Java tool that calculates the percentage of code accessed by tests, http://cobertura.sourceforge.net/.

[Coelho, 2005] Coelho, R., Mini-Curso: Teste de Software, Apresentadora: Roberta Coelho (PUC-Rio), SBES 2005, 26/9/2005.

[Coelho et. al., 2006] Coelho, R., Kulesza, U., Staa, A. V., and Lucena, C., Unit testing in multi-agent systems using mock agents and aspects. In *SELMAS '06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pages 83–90, New York, NY, USA, 2006. ACM Press.

[Cornett, 1996] Cornett, S., Code Coverage Analysis, http://www.bullseye.com/coverage.html, 1996.

[Deason et al., 1991] Deason, W. H., Brown, D. B., Chang, K.H., Cross J. H., A rule-based software test data generator, IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 1, pp. 108-117, March 1991.

[Dejong, 1975] Dejong, K., The analysis and behaviour of a class of genetic adaptive systems. PhD thesis, University of Michigan, 1975.

[DjUnit] DjUnit plugin for Eclipse, http://works.dgic.co.jp/djunit/.

[Duda, 2000] Duda, R. O., Hart, P. E., Stork, D. G., Pattern Classification (2nd Edition), Wiley-Interscience, 2000, pags 433 a 439.

[EMMA] EMMA: a free Java code coverage tool, http://emma.sourceforge.net/.

[Ferguson e Korel, 1996] Ferguson, R., Korel, B.. The chaining approach for software test data generation. *IEEE Transactions on Software Engineering*, 5(1):63-86, January 1996.

[Gross, 2001] Gross, H.G., An Evaluation of Dynamic Optimisation-based Worst-case Execution Time Analysis, Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century, Kathmandu, Nepal, May 2003.

[Gupta et al., 1998] Gupta, N., Mathur, A. P., Soffa M. L., Automated test data generation using an iterative relaxation method. In *Proceedings of the ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pages 231{244, November 1998.

[Hamming Distance] Hamming Distance Tutorial, http://people.revoledu.com/kardi/tutorial/Similarity/HammingDistance.html.

[Harman e Jones, 2001] Harman, M., Jones, B. F., Search-based software engineering, Information & Software Technology, Vol. 43, No. 14, pp. 833-839 (2001

[Harman et al., 2002] Harman, M., Hu, L., Hierons, R., Baresel, A., Sthamer, H., Improving evolutionary testing by flag removal. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pages 1359 - 1366, New York, USA, 2002.

[Harrold, 2000] Harrold, M. J., Testing: A Roadmap, I*n 22th International Conference on Software Engineering* – Future of SE Track, 61-72, June 2000.

[Herman, 1976] Herman, P. M., A Data Flow Analysis Approach to Program Testing, Australian Computer Journal, 92-96, November 1976.

[Hermadi e Ahmed, 2003] Hermadi, I., Ahmed, M.. Genetic algorithm based test data generator. *In Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 85–91, Canberra, 8-12 December 2003. IEEE Press.

[Holland, 1975] Holland, J. H., Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.

[Howden, 1975] Howden, W. E., Methodology for the generation of program test data, *IEEE Transactions on Computer*, Vol. c-24, No. 5, pp. 554-559, May 1975.

[IEEE/ANSI, 1990] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12, 1990.

[JCoverage] JCoverage: A suite of tools and technologies designed to improve coverage and testing productivity, http://www.jcoverage.com/.

[Jorgensen, 1995] Jorgensen, P. C., Software Testing a Craftsman's Approach, CRC Press, 1995.

[JUB] JUnit test case Builder, a JUnit test case generator framework, http://jub.sourceforge.net/.

[JUnitDoclet] JUnitDoclet, generation of skeletons of test cases based on your application source code, http://www.junitdoclet.org/.

[JUnit] JUnit.org. <http://www.junit.org/index.htm>.

[Kaner et al., 1999] Kaner, C., Falk, J., Nguyen, H. Q., Testing Computer Software*, John Wiley & Sons*, Inc., New York, second edition, 1999.

[Kessis et al., 2005] Kessis, M., Ledru, Y., Vandome, G., Experiences in Coverage Testing of a Java Middleware, In *Fifth Int. Workshop on Software Engineering and Middleware (SEM 2005)*, pages 39–45, Lisbon, September 2005, ACM Press.

[King, 1976] Symbolic Execution and Program Testing, *Communications of the ACM,* 19(7)? 385-394, 1976.

[Knight e Ammann, 1985] Knight, J., Ammann, P., An experimental evaluation of simple methods for seeding program errors, In *Proceedings of the Eighth International Conference on Software Engineering,* pages 337-342, London UK, August 1985. IEEE Computer Society.

[Lacerda e Carvalho, 1999] Lacerda, E.G.M e Carvalho, A.C.P.L., Introdução aos algoritmos genéticos, In: Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais. Editado por Galvão, C.O., Valença, M.J.S. Ed. Universidade/UFRGS: Associação Brasileira de Recursos Hídricos. p. 99-150. (Coleção ABRH de Recursos Hídricos; 7.), Porto Alegre, 1999.

[Mantere e Alander, 2005] Mantere, T., Alander, J.T.: Evolutionary software engineering, a review. *Applied Soft Computing.* Vol 5 (2005) 315-331, 2005.

[Mayrhauser, 1990] Mayrhauser, A. V., Software Engineering: methods and management, *Academic Press*, 1990.

[McCabe, 1976] McCabe, T., A Complexity Measure, IEEE Transactions on Software Engineering, 306-320, December 1976.

[McMinn, 2004] McMinn, P., Search-based software test data generation: A survey, *Journal of Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[McMinn, 2005] McMinn, P., Evolutionary Search for Test Data in the Presence of State Behavior, PhD Thesis, University of Sheffield, January 2005.

[Michael e McGraw, 1998] Michael, C., McGraw, G.. Automated software test data generation for complex programs. In *13th IEEE International Conferance on Automated Software Engineering*, pages 136-146, October 1998.

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

[Myers, 2004] MYERS, G. J., The Art of Software Testing, Wiley, 2nd Ed. New Jersey: John Wiley & Sons, 2004. 227 p.

[Nagarajan et al., 2003] Nagarajan, S. V., Garcia, O., Croll, P. R., Seventh, Extreme testing practice in extreme programming (XP), *IASTED International Conference on Software Engineering and Applications*, Marina del Ray, CA; USA; 3-5 Nov. 2003. pp. 453-458. 2003.

[Nogueira et. al., 2007] Nogueira, S. C., Cartaxo, E. G., Torres, D. G., Aranha, E. H. S., Marques, R.. Model Based Test Generation: An Industrial Experience. In: *1st Brazilian Workshop on Systematic and Automated Software Testing*, 2007, João Pessoa. SBBD-SBES, 2007.

[Offutt e Irvine, 1995] Affutt, A. J., Irvine, A., Testing Object-Oriented Software Using The Category Partition Method, In *17th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, Prentice-Hall, pp 293-304, August 1995.

[Offut e Hayes, 1996] Offut, J., Hayes, J., A Semantic Model of Program Faults, In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 195–200, San Diego, January 1996.

[Offut et. al., 1996] Offut, A. J., Pan, J., Tewary, K., Zhang, T., An Experimental Evaluation of Data Flow and Mutation Testing, *Software Practice and Experience*, vol. 26, no. 2, pp. 165-176, Feb, 1996.

[Pacheco et. al., 2006] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Redmond, WA, 2006.

[Pacheco e Ernst, 2007] Pacheco, C., Ernst, M. D., Randoop: feedback-directed random testing for Java, *Conference on Object Oriented Programming Systems Languages and Applications*, pp. 815-816, Montreal, 2007.

[Pressman, 2006] Pressman, R. S., Engenharia de Software, McGraw-Hill, Sexta Edição, São Paulo, 2006.

[Ramamoorthy, 1976] Ramamoorthy, C. V., Ho S. F., Chen W. T.: On the automated generation of program test data, *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 293-300, December 1976.

[Rapps e Weyuker, 1985] Rapps, S., Weyuker, E.J., Selecting Software Test Data Using Data Flow Information, *IEEE Transactions on Software Engineering*, 367-375, April 1985.

[Rela, 2004] Rela, L., Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, Department of Technology, 2004.

[Reynolds, 1994] Reynolds, R.G., An Introduction to Cultural Algorithms. In *Proceedings of the Third Annual Conference on Evolutionary Programming* (pp. 131–139). River Edge, NJ: World Scientific, 1994.

[RUP] Rational Unified Process: Visão Geral, http://www.wthreex.com/rup/process/ovu_proc.htm.

[Russel e Novig, 2003] Russel, S., Norvig, P, Artificial intelligence: a modern approach. 2o. ed. New Jersey: Prentice Hall, 2003. ISBN 0-13-790395-2.

[Sommerville, 2003] Sommerville, I., Engenharia de Software, São Paulo: Prentice Hall: 2003.

[Staknis, 1990] Staknis, M. E., Software quality assurance through prototyping and automated testing, *Inf. Software Technol*., Vol. 32, pp. 26-33, 1990.

[Sthamer, 1996] Sthamer, H., The automatic generation of software test data using genetic algorithms, Ph.D. thesis, Department of Electronics and Information Technology, University of Glamorgan, 1996.

[Sthamer et al., 2002] Sthamer, H., Wegener , J., Baresel, A., Using evolutionary testing to improve efficiency and quality in software testing. In *Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis and Review (AsiaSTAR)*, July 2002. 22-24th July.

[Tikir e Hollingsworth, 2002] Tikir, M., M., Hollingsworth, J., K., Efficient instrumentation for code coverage testing, *ACM, Sigsoft Software Engineering Notes*, vol.27, no.4, July 2002, pp.86-96, USA.

[Tlili et al., 2006] Tlili, M., Wappler, S., Sthamer, H., Wegener, J., Improving Evolutionary Real-Time Testing, Proceedings of the 8th annual conference on Genetic and evolutionary computation(GECCO 2006), pp. 1917-1924, July 2006.

[Tonella, 2004] Tonella, P., Evolutionary testing of classes, International Symposium on Software Testing and Analysis (ISSTA), pages 119–128, 2004.

[Torres et., al, 2007] Torres, D. G., Nogueira, S. C., Cartaxo, E. G., Aranha, E. H. S., Borba, P., Barros, F., Machado, P. D. L., Sampaio, A.. Brazil Test Center Research Group.

In: *st Brazilian Workshop on Systematic and Automated Software Testing*, 2007, João Pessoa. SBBD-SBES, 2007.

[Tracey et al., 1998] Tracey, N., Clark, J., Mander, K., McDermid, J., An Automated Framework for Structural Test-Data Generation, Proceedings of the 13th IEEE Conference on Automated Software Engineering, Hawaii, USA, 1998.

[UML] Introduction to OMG's Unified Modeling Language (UML), http://www.omg.org/gettingstarted/what_is_uml.htm.

[Wappler e Lammermann, 2005] Wappler, S., Lammermann, F., Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005), pp. 1053-1060, Washington D.C., USA, June 2005.

[Wappler e Wegener, 2006] Wappler, S., Wegener, J., Evolutionary Unit Testing of Object-oriented Software Using a Hybrid Evolutionary Algorithm, In proceedings of the IEEE World Congress on Computational Intelligence (WCII-2006), pages 3193-3200, Vancouver, Canada, July 2006.

[Wegener et al., 2002] Wegener, J., Buhr, K., Pohlheim, H., Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing, *Proceedings of the Genetic and Evolutionary Computation Conference*, p.1233-1240, July 09-13, 2002.

[Whitley, 1993] Whitley, D., A Genetic Algorithm Tutorial, *Technical Report CS-93-103*, Computer Science Department, Colorado State University, March 10, 1993.

[XP] Extreme Programing: A Gentle Introduction, http://www.extremeprogramming.org/.