UNIVERSIDADE DE SÃO PAULO

Escola de Artes, Ciências e Humanidades

Fernando Henrique Inocência Borba Ferreira

Framework de Geração de Dados de Teste para Programas Orientados a Objetos

São Paulo
Dezembro 2012

Fernando Henrique Inocência Borba Ferreira

Framework de Geração de Dados de Teste para Programas Orientados a Objetos

Dissertação apresentada ao Programa de Pós-graduação em Sistemas de Informação da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo como requisito parcial para obtenção do título de Mestre em Ciências.

Orientador: Prof. Dr. Marcio Eduardo Delamaro

Versão corrigida contendo as alterações e correções sugeridas pela banca examinadora. A versão original encontra-se na Biblioteca da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo.

São Paulo
Dezembro 2012

Dissertação de mestrado sob o título “*Framework de Geração de Dados de Teste para Programas Orientados a Objetos*”, defendida por Fernando Henrique Inocêncio Borba Ferreira e aprovada em 13 de dezembro de 2012, em São Paulo, Estado de São Paulo, pela banca examinadora constituída pelos doutores:

Prof. Dr. Marcio Eduardo Delamaro
Orientador

Prof. Dr. Mario Jino
Universidade Estadual de Campinas

Prof. Dr. Marcos Lordello Chaim
Universidade de São Paulo

Dedico este trabalho aos meus pais e a todos professores que tanto me inspiram.

Agradeço a Deus, minha família, meus amigos e meus orientadores pelo auxílio, paciência e compreensão durante o decorrer do projeto.

Uma mente que se abre para uma nova ideia, jamais
retorna ao seu tamanho inicial.

(Albert Einstein)

Resumo

A geração de dados de teste é uma tarefa obrigatória do processo de teste de software. Em geral, é realizada por profissionais de teste, o que torna seu custo elevado e sua automatização necessária. Os *frameworks* existentes que auxiliam essa atividade são restritos, fornecendo apenas uma única técnica de geração de dados de teste, uma única função de aptidão para avaliação dos indivíduos e apenas um algoritmo de seleção. Este trabalho apresenta o *framework* JaBTeG (*Java Bytecode Test Generation*) de geração de dados de teste. A principal característica do *framework* é permitir o desenvolvimento de métodos de geração de dados de teste por meio da seleção da técnica de geração de dados de teste, da função de aptidão, do algoritmo de seleção e critério de teste estrutural. Utilizando o *framework* JaBTeG, técnicas de geração de dados de teste podem ser criadas e experimentadas. O *framework* está associado à ferramenta de teste JaBUTi (*Java Bytecode Understanding and Testing*) para auxiliar a geração de dados de teste. Quatro técnicas de geração de dados de teste, duas funções de aptidão e quatro algoritmos de seleção foram desenvolvidos para validação da abordagem proposta pelo *framework*. De maneira complementar, cinco programas com características diferentes foram testados com dados gerados usando os métodos providos pelo *framework* JaBTeG.

Palavras-chave: Geração automática de dados de teste; *Framework* para geração automática de dados de teste; Geração de dados de teste para software orientado a objetos.

Abstract

Test data generation is a mandatory activity of the software testing process. In general, it is carried out by testing practitioners, which makes it costly and its automation needed. Existing frameworks to support this activity are restricted, providing only one data generation technique, a single fitness function to evaluate individuals, and a unique selection algorithm. This work describes the JaBTeG (Test Java Bytecode Generation) framework for testing data generation. The main characteristic of JaBTeG is to allow the development of data generation methods by selecting the data generation technique, the fitness function, the selection algorithm and the structural testing criteria. By using JaBTeG, new methods for testing data generation can be developed and experimented. The framework was associated with JaBUTi (Java Bytecode Understanding and Testing) to support testing data creation. Four data generation techniques, two fitness functions, and four selection algorithms were developed to validate the approach proposed by the framework. In addition, five programs with different characteristics were tested with data generated using the methods supported by JaBTeG.

Keywords: Automatic test data generation; Framework for automatic test data generation; Test data generation for object-oriented software.

Sumário

Lista de Figuras	xii
Lista de Tabelas	xv
1 Introdução	1
2 Teste de software e ferramentas	4
2.1 Defeito, erro, falha e engano	4
2.2 Teste de software	5
2.3 Teste funcional	5
2.4 Teste baseado em defeitos	6
2.5 Teste estrutural	6
2.5.1 Modelo de Programa	6
2.5.2 Critérios baseados em fluxo de controle	7
2.5.3 Critérios baseados em fluxo de dados	8
2.6 Ferramentas de teste	10
2.6.1 JaBUTi	10
2.6.2 POKE-TOOL	11
2.7 Geradores de dados de teste	11
2.8 Considerações finais	12
3 Geração de dados de teste	14
3.1 Algoritmos de geração de dados de teste	14

3.1.1	Geração aleatória	14
3.1.2	Execução simbólica	16
3.1.3	Teste baseado em busca	22
3.1.3.1	Subida de Encosta	23
3.1.3.2	Têmpera Simulada	24
3.1.3.3	Algoritmos Genéticos	24
3.1.3.4	Algoritmos Evolucionários	25
3.2	Representação de Dados de Teste	27
3.2.1	Operações com indivíduos de teste	31
3.3	Desafios para geração de dados de teste	32
3.3.1	Vetores e ponteiros	33
3.3.2	Objetos	33
3.3.3	Laços de repetição	36
3.3.4	Módulos	36
3.3.5	Caminhos não executáveis	37
3.4	Considerações finais	37
4	Frameworks Geradores de Dados de Teste	38
4.1	Identificação de Trabalhos	38
4.2	Trabalhos Relevantes	40
4.2.1	Evacom	41
4.2.2	TestFul	44
4.2.3	TDSGen/OO	45
4.2.4	AutoTest/Eiffel	47
4.2.5	Têmpera Simulada/Ada	50
4.3	Discussão	52
4.4	Considerações finais	53

5	Framework JaBTeG	55
5.1	Arquitetura do <i>framework</i>	55
5.1.1	Componentes do <i>framework</i>	55
5.1.2	Estruturas extensíveis	59
5.1.3	Análise do código do programa	67
5.1.4	Fabricação de indivíduos	67
5.1.5	Geração dirigida de valores aleatórios	69
5.1.6	Geração de valores para vetores e matrizes	69
5.1.7	Formatos para exportação dos dados gerados	70
5.1.8	Critérios de teste suportados	70
5.1.9	Limitações do <i>framework</i> JaBTeG	70
5.1.10	Considerações finais	70
6	Aplicações do Framework JaBTeG	72
6.1	Composição de técnicas de geração de dados de teste	72
6.1.1	Algoritmo Aleatório	72
6.1.2	Algoritmo Evolucionário	73
6.1.3	Subida de Encosta	75
6.1.4	Têmpera Simulada	76
6.2	Adequação à interface visual	77
6.3	Instalação de <i>plug-ins</i>	78
6.4	Configuração da ferramenta de teste	80
6.5	Aplicação de Técnicas de Geração de Dados de Teste	83
6.5.1	Geração de dados de teste para tipos primitivos	83
	<i>Sorting</i>	83
	<i>Trityp</i>	85
6.5.2	Geração de dados de teste para objetos	86

6.6	Discussão dos resultados	88
6.6.1	Recursos do <i>framework</i>	88
6.6.2	Tipos primitivos	89
6.6.3	Objetos complexos	91
6.7	Considerações Finais	93
7	Conclusão	94
	Referências	98
	Apêndice A Estruturas para extensão	103
	BaseGenerationStrategy	103
	MetaheuristicBaseGenerationStrategy	106
	Apêndice B Geração aleatória	109
	Apêndice C Algoritmo evolucionário	111
	Apêndice D Funções de aptidão	112
	Similaridade	112
	Ineditismo	113
	Apêndice E Algoritmos de seleção	116
	Elitismo	116
	Torneio	116
	Roleta	117
	Média	117
	Apêndice F Subida de encosta	119
	Apêndice G Têmpera simulada	122

Apêndice H Benchmarks	126
Insertion Sort	126
Quick Sort	126
Merge Sort	127
TryTip	127

Lista de Figuras

2.1	Visão sobre a atividade de teste. Fonte: Delamaro; Chaim; Vincenzi, 2010.	5
2.2	Blocos de comando e grafo de fluxo de controle do bubble-sort. Fonte: Chaim; Delamaro; Vincenzi, 2010.	8
2.3	Estrutura de um gerador de dados de teste. Fonte: Edvardsson, 1999. . .	12
3.1	Exemplo de código para geração aleatória. Fonte: Edvardsson, 1999. . . .	15
3.2	Exemplo de código com declarações propensas a defeitos. Fonte: Godefroid; Klarlund; Sen, 2005.	18
3.3	Resultados obtidos após avaliação do software Replace. Fonte: Burnim; Sen, 2006.	21
3.4	Resultados obtidos após avaliação do software Grep. Fonte: Burnim; Sen, 2006.	21
3.5	Resultados obtidos após avaliação do software Vim. Fonte: Burnim; Sen, 2006.	22
3.6	Esboço de recombinação Crossover. Fonte: Pinheiro, 2010.	25
3.7	Esboço de mutação. Fonte: Pinheiro, 2010.	25
3.8	Estrutura do algoritmo de Tonella. Fonte: Tonella, 2004.	27
3.9	Aplicação da representação de Tonella. Fonte: Criado com base em Tonella (2004)	28
3.10	Codificação binária - perda de informação. Fonte: Tracey et al., 2002. . .	30
3.11	Codificação binária - corrupção binária. Fonte: Tracey et al., 2002. . . .	30
3.12	Representação de Tonella: mutação de valores de entrada	31
3.13	Representação de Tonella: mudança de construtor	31
3.14	Representação de Tonella: inclusão de chamada a método	31

3.15	Representação de Tonella: remoção de chamada a método	32
3.16	Representação de Tonella: <i>crossover</i>	32
3.17	Exemplo de utilização de um vetor. Fonte: Edvardsson, 1999.	33
3.18	Classe para teste de estados de objetos. Criado com base em Tonella (2004).	34
3.19	Teste de unidade A. Criado com base em Tonella (2004).	35
3.20	Teste de unidade B. Criado com base em Tonella (2004).	35
3.21	Teste de unidade C. Criado com base em Tonella (2004).	36
4.1	Comparação entre as três abordagens. Fonte: Silva; Someren, 2010.	49
5.1	Arquitetura de integração do <i>framework</i>	56
5.2	Diagrama de atividades do processo de geração de dados de teste do <i>framework</i> JaBTeG.	57
5.3	Módulos do <i>framework</i> JaBTeG.	59
5.4	Estrutura extensível provida pelo <i>framework</i> JaBTeG.	60
5.5	Estrutura do <i>design pattern Template Method</i> . Fonte: GAMMA et al., 2000.	62
5.6	Estrutura da classe <i>BaseGenerationStrategy</i>	63
5.7	Estrutura da classe <i>MetaheuristicBaseGenerationStrategy</i>	65
6.1	Algoritmo aleatório desenvolvido com o <i>framework</i> JaBTeG.	73
6.2	Algoritmo Evolucionário desenvolvido com o <i>framework</i> JaBTeG.	74
6.3	Funções de aptidão criadas com o <i>framework</i> JaBTeG.	74
6.4	Técnicas de seleção criadas criadas com o <i>framework</i> JaBTeG.	75
6.5	Algoritmo de Subida de Encosta desenvolvido com o <i>framework</i> JaBTeG.	76
6.6	Algoritmo de Têmpera Simulada desenvolvido com o <i>framework</i> JaBTeG.	77
6.7	Ferramenta de teste JaBuTi.	79
6.8	Menu para instalação de novos <i>plug-ins</i> de geração de dados de teste.	79
6.9	Janela de instalação de novos <i>plug-ins</i> de geração de dados de teste.	79
6.10	Submenu do novo <i>plug-in</i> instalado.	80

6.11	Formulário do <i>plug-in</i> com dados das técnicas de geração de dados de teste.	80
6.12	Interface visual genérica para composição de cenários de teste.	81
6.13	Modelo de entidades utilizado pela técnica.	86
6.14	Teste unitário de um indivíduo simples gerado pelo <i>framework</i> JaBTeG. . .	87
6.15	Teste unitário de um dos indivíduos gerado pelo <i>framework</i> JaBTeG. . . .	88

Lista de Tabelas

4.1	Artigos selecionados após critérios de seleção da revisão sistemática	40
4.2	Classes utilizadas no teste de Evacon. Fonte: INKUMSAH; XIE, 2008 . . .	43
4.3	Cobertura de ramos obtida pelas seis abordagens testadas. Fonte: IN- KUMSAH; XIE, 2008	44
4.4	Classes sob teste. Fonte: Silva; Someren, 2010.	48
4.5	Número de defeitos encontrados pelo algoritmo aleatório. Fonte: Silva; Someren, 2010.	49
4.6	Número de defeitos encontrados pelo algoritmo aleatório com análise estática. Fonte: Silva; Someren, 2010.	50
4.7	Número de defeitos encontrados pelo algoritmo evolucionário. Fonte: Silva; Someren, 2010.	51
4.8	Definição da vizinhança. Fonte: Tracey; Clark; Mander; McDermid, 1998 .	52
4.9	Resultado dos experimentos com têmpera simulada. Fonte: Tracey; Clark; Mander; McDermid, 1998	52
4.10	Características dos <i>frameworks</i> identificados como relevantes na literatura.	53
5.1	Domínio padrão de valores para geração aleatória de indivíduos.	69
6.1	Esforço em linhas de código para criação de componentes de geração de dados de teste.	78
6.2	Geração de dados de teste para o método <i>Insertion Sort</i>	84
6.3	Geração de dados de teste para o método <i>Quick Sort</i>	84
6.4	Geração de dados de teste para o método <i>Merge Sort</i>	85
6.5	Geração de dados de teste para o método <i>Trityp</i> - Inteiros de 0 a 100. . . .	85
6.6	Geração de dados de teste para o método <i>Trityp</i> - Inteiros de 0 a 10. . . .	86

6.7	Quantidade de indivíduos gerados e o tempo de execução necessário para criá-los.	88
-----	---	----

Capítulo 1

Introdução

Após 50 anos de grande influência da computação no nosso dia-a-dia, tornou-se indiscutível sua importância para a evolução de qualquer área, seja ela voltada à indústria, educação, medicina, finanças ou engenharia. O mundo demanda produtividade e o uso de software supre essa necessidade. Com o passar dos anos e com o aumento da necessidade de destaque das empresas diante de seus concorrentes, a procura por software complexo e confiável emergiu e, assim, abordagens que garantem a qualidade tornaram-se questões chave para a indústria (TRACEY et al., 1998; SAGARNA et al., 2007; SILVA; SOMEREN, 2010).

Uma das maneiras de aumentar a qualidade do software é por meio do seu teste. Porém, essa atividade requer um processo caro que consome muito tempo. Diante desse cenário diversas técnicas e ferramentas foram desenvolvidas para melhorar o processo de teste de software. As técnicas de teste dividem-se em: funcional, estrutural e baseada em defeitos. A técnica funcional corresponde a um teste baseado em especificações do software. A técnica estrutural de teste requer a existência de uma implementação para a identificação de estruturas de interesse que devem ser exercitadas pelos casos de teste, enquanto que a baseada em defeitos insere pequenos defeitos no programa sob teste e verifica se os casos de teste são capazes de revelá-los (TRACEY et al., 1998).

Além das técnicas de teste, também foram construídas ferramentas que auxiliam o trabalho do testador, fornecendo recursos para apoiar o teste de software. No entanto, dois problemas críticos e de difícil solução são ainda pesquisados pela comunidade acadêmica, a saber: geração automática de dados de teste e automatização de oráculos de teste. Este trabalho aborda um deles, a geração automática de dados de teste.

A geração automática de dados de teste é uma abordagem vital para avanços do estado da arte do teste de software, pois a automatização permite a redução do custo de desenvolvimento e o aumento da qualidade do software (SAGARNA et al., 2007; SILVA; SOMEREN, 2010).

Recentemente, várias técnicas (SILVA; SOMEREN, 2010), como algoritmos metaheurísticos, geração aleatória e análise estática, foram propostas para geração automática de dados de teste, mas ainda não está claro como elas se comportam na prática e qual o

relacionamento que pode existir entre elas.

A geração automática de dados de teste pode ser classificada em subdivisões, as mais comuns são: aleatória, estática e dinâmica (TRACEY et al., 1998; DELAMARO et al., 2010). A geração de dados de teste aleatória não exige a análise de representações do sistema (e.g., código-fonte) para geração de casos de teste, isto é, nenhum critério baseado no software é utilizado para auxiliar o processo de geração de dados de teste. Sua eficiência é controversa, pois alguns autores defendem seu uso (PACHECO et al., 2008), enquanto outros não acreditam que sua utilização seja efetiva (BURNIM; SEN, 2008). As abordagens estáticas são caracterizadas pela análise de representações do sistema – tais como a documentação e o código-fonte – e não exigem a execução do sistema sob teste para que sejam criados os dados de entrada para os testes do sistema. Em sua maioria, os geradores de dados de teste que utilizam a abordagem estática adotam a execução simbólica como principal abordagem (TRACEY et al., 1998).

A execução simbólica estende a execução normal do software sob teste, na qual os operadores básicos da linguagem são estendidos para aceitar símbolos como entrada e produzir fórmulas simbólicas como saídas. Muitos desafios ainda estão atrelados a esta abordagem, pois é difícil analisar recursos como: recursão, estruturas de dados dinâmicas, índices de vetores que dependem de variáveis de entrada e laços de repetição. A geração de dados de teste dinâmica envolve a execução do software sob teste e uma busca por dados de teste que atendam critérios esperados pela aplicação. Algoritmos metaheurísticos são vistos como boas soluções para geração dinâmica de dados de teste (SILVA; SOMEREN, 2010), pois são direcionados pelo uso de funções de aptidão que verificam o quanto cada caso de teste proposto é apropriado para o software sendo testado.

Apesar de existirem diferentes técnicas de geração de dados de teste, é difícil afirmar qual delas é a mais adequada a um determinado escopo. Esse problema decorre do fato de não existirem mecanismos que possibilitem a comparação entre as técnicas de geração de dados de teste. E ao estudar as técnicas de geração de dados de teste junto com várias ferramentas, notou-se a ausência de uma abordagem que forneça múltiplas técnicas de geração de dados de teste, e que também disponha de recursos para auxiliar no desenvolvimento de técnicas de geração de dados de teste.

Diante desta lacuna este trabalho descreve a construção do *framework* JaBTeG (*Java Bytecode Test Generation*), que fornece recursos para geração de dados de teste, além de um modelo reutilizável para ser adotado como padrão de composição de técnicas metaheurísticas de geração de dados de teste. O objetivo principal deste *framework* é reduzir

o esforço e o custo da construção de técnicas de geração de dados de teste.

A construção deste *framework* resultou na criação de um provedor de recursos para geração de dados de teste, além de uma estrutura extensível para composição de técnicas de geração de dados de teste. Também foi desenvolvida uma biblioteca de geração de dados de teste composta por quatro técnicas de geração de dados de teste (Aleatória, Evolucionária, Subida de Encosta e Têmpera Simulada), duas funções de aptidão (Similaridade e Ineditismo) e quatro algoritmos de seleção (Elitismo, Torneio, Roleta e Média). Este trabalho também adaptou a ferramenta de teste JaBUTi para adequá-la a instalação de *plug-ins* geradores de dados de teste, também criando uma interface visual genérica para composição de diferentes cenários de geração de dados de teste, com o objetivo de facilitar a interação do usuário testador com o *framework* JaBTeG.

O próximo capítulo descreve conceitos de teste de software e ferramentas de teste. O Capítulo 3 apresenta a geração automática de dados de teste, descrevendo as técnicas mais utilizadas e indicando exemplos de abordagens que as utilizam. O levantamento bibliográfico realizado é apresentado no Capítulo 4, enquanto o trabalho construído é apresentado no Capítulo 5. Os resultados obtidos são apresentados no Capítulo 6 e a conclusão no Capítulo 7.

Capítulo 2

Teste de software e ferramentas

Uma das maneiras de assegurar a qualidade de um software é por meio do teste de software. Porém, testar um software é um processo caro que consome muito tempo, especialmente em aplicações críticas, que envolvem softwares com requisitos de alta disponibilidade ou critérios rígidos de segurança. Para minimizar esta questão, diversas ferramentas e técnicas de teste foram desenvolvidas (SAGARNA et al., 2007; SILVA; SOMEREN, 2010). O teste manual é o método mais utilizado para averiguar o funcionamento de um software, mas é um processo lento e propenso a erros do testador. Por isso, existe uma necessidade de estratégias avançadas de teste de software, pois os sistemas estão tornando-se cada vez mais complexos e os prazos de desenvolvimento mais curtos (TRACEY et al., 1998; SAGARNA et al., 2007; SILVA; SOMEREN, 2010). Neste capítulo são discutidos os principais conceitos de teste de software. Em particular, aqueles relacionados com o teste estrutural de software.

2.1 Defeito, erro, falha e engano

Defeitos são caracterizados como passos, processos ou definições de dados incorretos, inserido no programa durante a codificação. O defeito é a consequência de um *engano* cometido por um desenvolvedor. O erro consiste de um estado inconsistente na execução de um programa originado por um defeito, como por exemplo um operador de comparação que causa um desvio de fluxo incorreto na execução do programa. Os erros são a causa das falhas. Falhas são desvios da especificação, isto é, comportamentos da aplicação que diferem do comportamento esperado, percebidos por quem o executa. A manifestação de uma falha indica a existência de um defeito no programa (DELAMARO et al., 2007; DELAMARO et al., 2010).

2.2 Teste de software

Considerado como um assunto vital no cenário de desenvolvimento de software (DELAMARO et al., 2010), o teste de software consiste da atividade de escolher dados para executar um determinado software e verificar se o resultado produzido corresponde ao resultado esperado.

Figura 2.1 – Visão sobre a atividade de teste. Fonte: Delamaro; Chaim; Vincenzi, 2010.

Com a execução de testes espera-se que ocorram situações nas quais o software não funcione como esperado e que, caso essas situações não ocorram, tenha-se uma indicação de que o software vai, sempre ou pelo menos na maioria dos casos, funcionar sem problemas. A Figura 2.1 resume o que se entende por teste de software (DELAMARO et al., 2010). O elemento principal da Figura 2.1 é o programa sob teste, representado pelo retângulo com o rótulo *P*. O retângulo à esquerda, rotulado com a letra *T* representa o conjunto de dados de teste. Um conjunto de dados de teste é o conjunto de dados que pode ser utilizado para executar *P*. De maneira complementar, um par formado por um dado de teste e seu correspondente resultado esperado é chamado de caso de teste (DELAMARO et al., 2007; DELAMARO et al., 2010).

2.3 Teste funcional

O teste funcional é uma técnica de teste que considera o programa como uma caixa preta, na qual são fornecidas entradas e avaliadas as saídas geradas. As saídas são avaliadas para verificar se estão em conformidade com os objetivos esperados. No teste funcional não são considerados os detalhes de implementação, pois o software é avaliado segundo o ponto de vista do usuário (FABBRI et al., 2007).

Inicialmente, previa-se que o teste funcional pudesse detectar todos os defeitos, submetendo o programa a todas as possíveis entradas, assim denominado como teste exaustivo. No entanto, o domínio de entrada pode ser infinito ou muito grande, de modo a tornar o tempo desta atividade de teste impraticável (FABBRI et al., 2007).

2.4 Teste baseado em defeitos

Nessa técnica de teste são adotados defeitos comuns do processo de implementação de software para derivação dos requisitos de teste. O Teste de Mutação é um critério de teste baseado em defeitos, no qual o programa sob teste é alterado diversas vezes, incluindo defeitos, como se estivessem sendo inseridos no programa original. Estas alterações no programa original geram um conjunto de programas alternativos, também conhecidos como mutantes. O trabalho do usuário testador é construir casos de teste que mostrem a existência destes defeitos e a diferença de comportamento entre o programa original e os programas mutantes (DELAMARO et al., 2007).

Atualmente, devido a grande aceitação da comunidade de teste de software, muitos trabalhos na literatura utilizam o Teste de Mutação como técnica para validação da efetividade de novos critérios de teste (DELAMARO et al., 2007).

2.5 Teste estrutural

Segundo Barbosa et al. (2007) o teste estrutural é baseado no conhecimento da estrutura do programa, sendo os aspectos de implementação fundamentais para a geração dos casos de teste. Em sua maioria, os critérios estruturais utilizam uma representação do programa intitulada grafo de fluxo de controle. A partir dele podem ser escolhidos os elementos que devem ser executados, caracterizando assim o teste estrutural. Tais elementos podem ser comandos, desvios, caminhos ou definições e usos de variáveis do programa (BARBOSA et al., 2007).

2.5.1 Modelo de Programa

Um programa pode ser considerado como uma função $P: S \rightarrow R$, onde P é o programa, S é o conjunto de todas as possíveis entradas e R corresponde ao conjunto de todas as possíveis saídas. Enquanto que x em P corresponde a uma variável que será utilizada como parâmetro de entrada de P ou como um comando de leitura que exija a entrada de valores por parte do usuário executor. A execução de P para uma entrada x é denotada como $P(x)$ (EDVARDSSON, 1999).

Grafos de fluxo de controle (*control flow graphs*) são adotados para representação de sequências de execução de programas. Existem diferentes definições para representação dos grafos de fluxo de controle, e tais definições são baseadas nas características da lin-

guagem (EDVARDSSON, 1999; DELAMARO et al., 2010).

O grafo de fluxo de controle de um programa é um grafo direcionado $G = (N, E, s, e)$, em que N representa um conjunto de nós e E um conjunto de arestas que conectam os nós. Além de nós especiais, como um nó de entrada s e um ou mais nós de saída e (EDVARDSSON, 1999).

Um nó, ou bloco básico (*basic block*), corresponde a uma sequência de instruções, na qual o fluxo de controle entra na primeira instrução e sai na última instrução, sem a existência de desvios. A utilização de arestas entre dois nós corresponde a transferências de controle. Se um nó possuir mais de uma aresta de saída, então se deve classificar o nó como condição e as arestas como ramos.

Para a construção de grafos de fluxo de controle é necessária a análise do código considerando a linguagem de programação com a qual o programa foi construído. Assim, para cada linguagem de programação, obriga-se uma nova interpretação de como cada construção da linguagem deve guiar a construção do grafo. Esta análise é chamada de modelo de fluxo de controle. Ferramentas que analisam o programa fonte e criam automaticamente seu grafo de fluxo de controle implementam um modelo de fluxo de controle (DELAMARO et al., 2010).

Considerando um grafo de fluxo de controle, um caminho de um programa é uma sequência de nós, $p = (p_1, p_2, \dots, p_q)$, onde existe uma aresta entre p_i e p_{i+1} . Se $P(x)$ percorrer o caminho p , então pode-se afirmar que x percorre p . Um caminho que inicia no nó de entrada e termina em um nó de saída é chamado de caminho completo, senão é chamado de caminho incompleto ou segmento de caminho. Um caminho é viável se existe uma entrada ($x \in S$), que o percorra, senão o caminho é inviável ou não executável (EDVARDSSON, 1999).

2.5.2 Critérios baseados em fluxo de controle

De acordo com Delamaro, Vincenzi e Chaim (2010) os critérios de teste baseados em fluxo de controle utilizam informações contidas no grafo de fluxo de controle para derivar seus requisitos de teste. Alguns desses critérios são:

- Critério todos-nós: exige que um conjunto de teste execute pelo menos uma vez cada um dos nós do GFC. Isto significa que, dado um conjunto de teste $T = \{t_1, t_2, \dots, t_n\}$ e os caminhos cobertos por ele percorridos, definidos como $\Pi = \{\pi_1, \pi_2, \dots,$

π_n }, exige-se que cada um dos nós apareça pelo menos uma vez em algum caminho de Π .

- Critério todas-arestas: similar ao critério todos-nós, exceto que o requisito de teste é a passagem por todas as arestas, em vez de todos os nós. Dado um conjunto de teste $T = \{t_1, t_2, \dots, t_n\}$ e os respectivos caminhos cobertos por ele, definidos como $\Pi = \{ \pi_1, \pi_2, \dots, \pi_n \}$, exige-se que cada uma das arestas apareça pelo menos uma vez em algum caminho de Π .

Figura 2.2 – Blocos de comando e grafo de fluxo de controle do bubble-sort. Fonte: Chaim; Delamaro; Vincenzi, 2010.

A Figura 2.2 apresenta o programa e o GFC relativo ao *bubble-sort*. Com a entrada [3, 2, 1] garante-se que cada nó é executado ao menos uma vez; entretanto, nem todas as arestas são cobertas. A aresta (7,6) não é executada nenhuma vez com esses dados de entrada. Mas, ao executar o algoritmo com a entrada [1, 2, 3] os critérios todos-nós e todas-arestas são cobertos. Com este exemplo pode-se notar que o critério todas-arestas inclui o critério todos-nós, isto é, sempre que todas as arestas forem cobertas, todos os nós também o são (DELAMARO et al., 2010).

2.5.3 Critérios baseados em fluxo de dados

Os critérios baseados em fluxo de dados utilizam a análise de fluxo de dados como fonte de informação para derivar os requisitos de teste. Tais critérios baseiam-se nas associações entre a definição de uma variável – isto é, ponto em que uma variável recebe um valor – e seus possíveis usos. A ideia é que cada vez que uma variável receber um

valor, esse valor deve ser verificado em algum ponto do programa. A motivação para o uso de critérios baseados em fluxo de dados é a indicação de que, mesmo para programas pequenos, o teste baseado unicamente no fluxo de controle não é eficaz para revelar a presença mesmo de defeitos simples. As formas de utilização de uma variável podem ser duas (BARBOSA et al., 2007):

Definição – toda referência feita a uma variável que faz com que o valor dessa variável possa ser alterado (i.e., variável no lado esquerdo de um comando de atribuição, variável em chamadas de procedimentos como parâmetro de saída, variável em um comando de entrada).

Uso – todas as demais referências a uma variável, quando o valor armazenado na variável é utilizado mas não modificado. O uso das variáveis ainda pode ser caracterizado como: predicativo (ou p-uso), quando o valor da variável é usado para definir o fluxo de controle do programa (i.e., uso de variáveis em blocos de decisão ou em laços de repetição); ou computacional (ou c-uso): todos os demais usos que não são p-usos (por exemplo, uso de variáveis em expressões matemáticas).

Rapps e Weyuker (1982) propuseram o conceito Grafo Def-Uso, que consiste de uma extensão do grafo de fluxo de controle. Nesta extensão são adicionadas ao grafo de fluxo de controle informações a respeito do fluxo de dados do programa, descrevendo associações entre pontos do programa nos quais são atribuídos valores às variáveis e pontos nos quais esses valores são utilizados. Os requisitos de teste são criados com base em tais associações (BARBOSA et al., 2007).

Além disso, Rapps e Weyuker propuseram uma família de critérios de fluxo de dados, tendo como principais critérios:

- Todas-Definições: exige que para cada definição de variável, um uso seja exercitado (BARBOSA et al., 2007).
- Todos-Usos: requer que para cada definição de variável, todos os usos existentes sejam exercitados (BARBOSA et al., 2007).
- Todos-Potenciais-Usos: baseia-se na associação entre uma definição de variável e seus possíveis subsequentes usos para a derivação de casos de teste (BARBOSA et al., 2007).

- Todos-Du-Caminhos: requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos dessa variável seja exercitada por caminhos livres de definição e livres de laço (BARBOSA et al., 2007).

2.6 Ferramentas de teste

Para auxiliar o trabalho dos testadores existem ferramentas que fornecem recursos para o teste de software; alguns exemplos de ferramentas são: Cobertura¹, JaCoCo², EMMA³, POKE-TOOL (CHAIM, 1991), JaBUTi (DELAMARO et al., 2010) e Coverlipse⁴. Essas ferramentas fornecem apoio para execução de casos de teste e monitoramento de execuções. A seguir serão discutidas as características de duas dessas ferramentas, JaBUTi e POKE-TOOL, por estarem disponíveis para o uso público e representarem o conjunto de ferramentas que poderão utilizar os recursos do *framework* construído.

2.6.1 JaBUTi

A JaBUTi (*Java Bytecode Understanding and Testing*) é uma ferramenta de apoio à aplicação de critérios estruturais baseados no fluxo de controle e no fluxo de dados de programas, construída para o entendimento e o teste de programas Java. A JaBUTi é composta por diversos módulos de análise de software, dentre eles: módulo de análise de cobertura, módulo de *slicing* e módulo de cálculo de métricas de software orientadas a objetos. O módulo de cobertura é utilizado para avaliar a qualidade de um dado conjunto de teste. O módulo de fatiamento de programas (*slicing*) é apropriado para identificar regiões sujeitas a defeitos no código, sendo bastante útil em processos de depuração. O módulo de cálculo de métricas é utilizado para identificar a complexidade e o tamanho de cada classe sob teste (VINCENZI et al., 2003; VINCENZI et al., 2007; DELAMARO et al., 2010).

A JaBUTi foi criada para analisar *bytecodes* Java, de forma que nenhum código fonte é necessário para que ela execute suas funções. Um arquivo bytecode é uma representação binária que contém informações sobre uma classe, tais como: seu nome, o nome de sua superclasse, informações sobre os métodos, variáveis e constantes utilizadas, além das instruções de cada um de seus métodos. Instruções de bytecode são parecidas com instruções

¹<http://cobertura.sourceforge.net/>

²<http://www.eclEmma.org/jacoco/>

³<http://emma.sourceforge.net/>

⁴<http://coverlipse.sourceforge.net/>

em linguagem *assembly*, mas armazenam informações de alto nível sobre o programa. Trabalhando diretamente com o bytecode Java, tanto o desenvolvedor de um componente quanto seus clientes podem utilizar a mesma representação e os mesmos critérios para testar componentes Java (VINCENZI et al., 2007; DELAMARO et al., 2010).

2.6.2 POKE-TOOL

POKE-TOOL é uma ferramenta de teste de software, disponível em ambiente UNIX, que apóia o uso dos critérios todos-nós, todas-arestas e os critérios básicos da família potenciais-usos (MALDONADO et al., 1989) no teste de unidade de programas escritos na linguagem C. A ferramenta POKE-TOOL possui módulos funcionais cuja utilização ocorre por meio de interface gráfica ou linha de comando (*shell scripts*). Por meio da interface, o usuário pode indicar qual programa deve ser testado e qual critério de teste deve ser aplicado. Em seguida, a ferramenta executa os testes necessários, coletando informações de cobertura dos critérios de teste estruturais apoiados (BARBOSA et al., 2007) (CHAIM, 1991).

O uso de linhas de comando (*shell scripts*) é recomendado a testadores mais experientes, pois exige conhecimentos de programação, conhecimentos sobre conceitos de teste e domínio sobre o conjunto de programas que compõem a ferramenta POKE-TOOL. A grande vantagem da utilização de linhas de comando é a possibilidade de executar estudos experimentais nos quais uma mesma sequência de passos deve ser executada várias vezes até que os resultados obtidos sejam significativos do ponto de vista estatístico.

Segundo Barbosa et al (2007), a POKE-TOOL foi projetada como uma ferramenta interativa cuja operação é orientada a uma sessão de teste. O termo “sessão de teste” é adotado para designar as atividades envolvendo o teste, sendo elas: análise estática da unidade, preparação para o teste, submissão de casos de teste, avaliação de casos de teste e administração dos resultados de teste.

2.7 Geradores de dados de teste

Como afirma Korel (1990), geradores de dados de teste são ferramentas que auxiliam o desenvolvedor na geração de dados para testar um programa. Seu objetivo é encontrar conjuntos de valores de entrada que exercitem os requisitos de teste especificados. Um gerador de dados de teste consiste geralmente de três componentes: um analisador do programa, um seletor de caminho e um gerador de dados. O componente analisador do

programa fornece todas as informações que podem ser extraídas do programa, tais como grafos de dependência de dados e grafos de fluxo de controle. O seletor de caminho procura identificar, por meio do grafo de fluxo de controle, os possíveis caminhos para os quais o componente gerador de dados deverá criar valores de entrada. A Figura 2.3 apresenta os três componentes de um gerador de dados de teste (EDVARDSSON, 1999).

Figura 2.3 – Estrutura de um gerador de dados de teste. Fonte: Edvardsson, 1999.

Os geradores de dados de teste podem utilizar três métodos para análise dos programas, sendo eles:

Método Estático: não exige a execução do programa e o analisa pelas representações do sistema (e.g., documento de requisitos, diagramas de projeto e código-fonte);

Método Dinâmico: executa o programa uma primeira vez com dados aleatórios de entrada e monitora o fluxo de execução do programa verificando se o caminho desejado foi percorrido ou não. Caso não tenha sido percorrido, então retorna-se ao ponto de desvio e altera-se as entradas para identificar os dados que levam à execução do caminho desejado;

Método Híbrido: combina os métodos estático e dinâmico, de modo que os benefícios das suas técnicas sejam combinados (DELAMARO et al., 2010).

2.8 Considerações finais

Neste capítulo foram apresentadas as motivações para o teste de software, os conceitos de teste estrutural de software e os critérios de teste mais utilizados, além da apresentação

das ferramentas de teste JaBUTi e Poke-Tool e dos conceitos introdutórios sobre os geradores de dados de teste. No próximo capítulo o funcionamento dos geradores de dados de teste é detalhado por meio da apresentação das técnicas mais populares de geração de dados de teste, dos modos de representação dos dados de entrada e da discussão sobre as dificuldades encontradas para geração de dados de teste.

Capítulo 3

Geração de dados de teste

Projetar casos de teste manualmente é entediante, caro e propenso a erros; por isso, sua automatização é indicada. A automatização do processo de teste pode permitir tanto a redução do custo de desenvolvimento quanto o aumento da qualidade do software. Neste capítulo são discutidas técnicas de geração de dados de teste, modelos de representação de dados de teste e desafios da geração de dados de teste.

3.1 Algoritmos de geração de dados de teste

Uma quantidade grande de métodos – como geração aleatória, execução simbólica e testes baseados em busca – é utilizada para apoiar o processo de geração de dados de teste (SAGARNA et al., 2007; MIRAZ et al., 2009; SILVA; SOMEREN, 2010). Neste seção, são descritas as principais técnicas de geração de dados de teste, bem como as dificuldades associadas a elas.

3.1.1 Geração aleatória

O método de geração aleatória é o mais simples de todos, pois sua utilização não exige a análise de representações do sistema (e.g., código-fonte). Em sistemas complexos ou programas que possuam um conjunto de critérios de adequação complexos, este método pode ser uma má escolha, pois a probabilidade de selecionar uma entrada adequada dentro de um conjunto gerado de forma aleatória é baixa. Outro problema da execução aleatória é que, ao longo de sua execução, conjuntos de valores que exercitam o mesmo comportamento são gerados. Este cenário não é adequado, pois torna boa parte dos resultados redundantes (EDVARDSSON, 1999; SEN et al., 2005; BURNIM; SEN, 2008; DELAMARO et al., 2010).

Conforme o exemplo de Edvardsson (1999), se avaliado o código da Figura 3.1, pode-se perceber que a probabilidade de execução do comando “*write(1)*” é $1/n$, onde n é o maior número inteiro possível de ser gerado aleatoriamente, já que para executar este comando a e b devem ser iguais.


```
void foo(int a, int b)
{
    if (a == b) then
        write(1);
    else
        write(2);
}
```

Figura 3.1 – Exemplo de código para geração aleatória. Fonte: Edvardsson, 1999.

Segundo Pacheco, Lahiri e Ball (2008), a eficiência do teste aleatório é uma questão não resolvida dentro da comunidade de teste, pois alguns estudos sugerem que o teste aleatório não é tão efetivo quanto as demais técnicas de geração de dados de teste. Em contraponto, outros artigos afirmam que o teste aleatório, devido a sua velocidade e escalabilidade, é uma técnica capaz de superar as demais.

Uma ferramenta relevante de teste aleatório de software é o Randoop (PACHECO; ERNST, 2007; PACHECO et al., 2008). Randoop (*Random Tester for Object-Oriented Programs*) utiliza *Feedback-Directed Random Testing*, técnica de geração aleatória de dados de teste que gera um conjunto de casos de teste para descoberta de defeitos em programas orientados a objetos. Seu algoritmo cria sequências de chamadas a métodos utilizando métodos e construtores públicos das classes, executa as sequências de métodos e, com base no resultado de suas execuções, identifica as entradas reveladoras de defeitos.

De acordo com Pacheco, Lahiri e Ball 2008, engenheiros do time de teste da Microsoft utilizaram Randoop para os testes de um componente pertencente ao *.Net Framework*. Tal componente é utilizado em diversas aplicações escritas na Microsoft e é bastante extenso (possui cerca de 100 mil linhas de código, escritas em C# e C++) e, por esta razão, teve aproximadamente 40 profissionais de teste dedicados exclusivamente para o teste de seu funcionamento durante um período de cinco anos. O time de teste havia testado o componente utilizando muitas técnicas e ferramentas, desde o teste manual e testes de *stress* até ferramentas que utilizam lógica *fuzzy*. Um engenheiro de teste, trabalhando dedicadamente com este componente, utilizando as ferramentas existentes, era capaz de encontrar 20 erros por ano. Depois de 15 horas de esforço humano e 150 horas acumuladas de processamento computacional sobre este componente, a ferramenta Randoop foi capaz de encontrar mais erros do que um engenheiro de teste ao longo de um ano, levando-se em consideração que um engenheiro de teste trabalhando com as ferramentas e metodologias existentes encontra em média 20 novos erros por ano.

3.1.2 Execução simbólica

A execução simbólica é uma técnica empregada para geração automática de dados de entrada visando, por exemplo, a cobertura dos ramos (fluxos) do código. Esta técnica de execução é uma extensão natural da execução normal na qual os operadores básicos da linguagem são estendidos para aceitar entradas simbólicas e produzir uma expressão simbólica de saída. Expressões simbólicas de saída são representações das variáveis de saída em termos das variáveis de entrada, enquanto que as entradas simbólicas são representações simbólicas das variáveis de entrada. Esta técnica foi originalmente proposta por James C. King, em 1976 (KING, 1976; VERGILIO et al., 2007; TILLMANN; HALLEAUX, 2008; ZHANG et al., 2010).

A execução simbólica foi proposta originalmente como uma técnica estática de análise de programas, isto é, uma técnica que considerava apenas o código fonte do programa sob teste e que não exigia sua execução. Este cenário é o ideal desde que todas as decisões do caminho possam ser executadas considerando-se apenas o código-fonte. A análise estática tornou-se limitada quando os programas começaram a utilizar instruções que não podiam ser resolvidas facilmente (e.g., acesso a memória através de ponteiros arbitrários ou cálculos aritméticos de ponto flutuante) ou quando partes do comportamento do programa eram desconhecidas (e.g., quando o programa se comunica com o ambiente do qual nenhum código-fonte está disponível e cujo comportamento não foi especificado). Para resolver tais problemas foi necessária a adoção de uma nova abordagem que utilizasse informações do ambiente no qual o programa está incorporado, permitindo que outras características, além do código-fonte, pudessem ser avaliadas para cobertura de todas as possíveis condições de uma aplicação (TILLMANN; HALLEAUX, 2008).

A execução dinâmica exige a execução do programa sob teste para coleta de informações dinâmicas que são observadas durante sua execução concreta. Assim, a execução simbólica dinâmica faz a análise das informações dinâmicas coletadas, para resolução de questões que eram difíceis ou impossíveis de serem respondidas pela execução simbólica estática (TILLMANN; HALLEAUX, 2008).

Diante do desafio de criar novas ferramentas para geração automática de dados de teste, Tillmann e Halleaux (2008) construíram, nos laboratórios do Microsoft Research, uma ferramenta de geração automática de teste para plataforma Microsoft .Net, intitulada Pex. A ferramenta Pex produz conjuntos de entrada com alta cobertura do código de programas .Net por meio do monitoramento do fluxo de suas execuções.

Para obter resultados favoráveis – isto é, resultados que indiquem a existência de defeitos – o programa sob teste é executado de maneira simbólica dinâmica, mas este conceito de execução não é novo, e Pex procura estender este conceito agregando novas técnicas. Uma das novas técnicas adotadas por Tillmann e Halleaux é a utilização de um solucionador de restrições chamado Z3 (BALL et al., 2010; VEANES et al., 2009), que constrói representações simbólicas fiéis a restrições que caracterizam caminhos de execução de programas .Net. Além desse solucionador de restrições, Pex utiliza um conjunto de estratégias de busca para navegar por entre os ramos da aplicação em uma pequena quantidade de tempo, ao contrário da execução simbólica, que por padrão utiliza busca em profundidade. Outro ponto de destaque de seu funcionamento é que Pex consegue trabalhar sobre conjuntos encarados como inseguros – pontos inseguros são todos aqueles pontos que fazem acessos a memória através de vetores ou ponteiros.

Iniciando de um método que contenha parâmetros, a ferramenta Pex inicia um modelo de verificação orientado a caminho que combina repetidas execuções do programa e resolução de restrições simbólicas do sistema para obtenção de dados de entrada que guiem o programa ao longo de diferentes caminhos de execução (TILLMANN; HALLEAUX, 2008).

Como experimento, a ferramenta Pex foi executada sobre um componente pertencente ao núcleo da plataforma Microsoft .Net. Este componente foi testado durante anos por diversos profissionais de teste e é utilizado como base de outras bibliotecas. Como resultado, Pex foi eficaz o suficiente para detectar defeitos, incluindo problemas sérios, de grande impacto.

Uma abordagem complementar à execução simbólica é a CONCOLIC (GODEFROID et al., 2005), que combina a execução concreta (real) com a execução simbólica de um programa para geração de dados de entrada para testes, isto é, o programa sob teste é executado de forma concreta e ao mesmo tempo executa computação simbólica. Dessa forma, durante a execução concreta de um programa, ao longo de seu caminho de execução, é gerado um conjunto de restrições simbólicas que devem ser resolvidas para que sejam determinados os dados de entrada. Se tais restrições puderem ser resolvidas então serão gerados dados de entradas que guiarão o programa ao longo do seu caminho de execução. Se não puderem ser resolvidas então propõe-se a simples substituição por valores aleatórios (SEN et al., 2005; BURNIM; SEN, 2008).

Larson e Austin (2003) foram os primeiros a propor a combinação de execução concreta (real) e execução simbólica, mas Godefroid, Klarlung e Sen (2005) foram os primeiros a

propor a geração de entradas de teste utilizando este tipo de execução.

Godefroid et al (2005) desenvolveram uma ferramenta intitulada *Directed Automated Random Testing* (DART, em português Teste Automático Aleatório Dirigido) que permite a automatização de testes de qualquer programa compilável sem a necessidade de escrever um roteiro de testes ou escrita de mais código (e.g., testes de unidade). Durante o teste, a ferramenta DART procura detectar: defeitos do programa, violações de memória e laços infinitos de programas escritos na linguagem C.

Para detecção dos defeitos, a ferramenta DART utiliza a técnica CONCOLIC, executa o programa sob teste de forma concreta (iniciando sua execução com valores aleatórios) e simbólica (calculando restrições simbólicas sobre os predicados encontrados durante seu caminho de execução) (GODEFROID et al., 2005).

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
    if (x != y)
        if (f(x) == x + 10)
            abort();          /* error
    return 0;
}
```

Figura 3.2 – Exemplo de código com declarações propensas a defeitos. Fonte: Godefroid; Klarlund; Sen, 2005.

Para Godefroid, Klarlund e Sen (2005), a função *h*, presente na Figura 3.2, é defeituosa porque pode conduzir para uma declaração *abort*, que acarretará um erro, para a combinação de alguns parâmetros de entrada *x* e *y*. Executando a função *h* com valores aleatórios para *x* e *y* é muito improvável detectar o erro. Esse problema é típico para entradas aleatórias, pois é difícil gerar valores de entrada que guiem o programa por todos os possíveis caminhos de execução. De acordo com os autores, DART é capaz de reunir dinamicamente conhecimento sobre a execução do programa. O programa sob teste será executado a primeira vez com uma entrada aleatória, e a cada execução irá calcular um novo vetor de entrada para a próxima execução. Este novo vetor de entrada irá conter valores que são a solução de restrições simbólicas recolhidas a partir de predicados descobertos durante o caminho de execução do programa sob teste. A geração de novos vetores de entrada é importante, pois força a execução do programa a seguir através de um novo caminho, além de acarretar na composição de dados de teste eficazes o suficiente para varrer todos os caminhos executáveis.

A ferramenta DART combina três fases para detecção de erros: (GODEFROID et al.,

2005)

Extração automática da interface do programa: depois de fornecido um programa para teste, DART identifica a interface externa pela qual o programa pode obter entradas. Essa identificação é feita por um analisador estático de código-fonte. A interface externa é definida por variáveis externas, funções externas e argumentos definidos pelo desenvolvedor para a função principal que inicia a execução do programa.

Geração automática de um roteiro de teste: uma vez que a interface externa do programa tenha sido identificada, é gerado um roteiro de teste aleatório simulando o ambiente mais genérico de execução para o programa e suas interfaces. Este roteiro de teste é o resultado da execução do programa sob teste com entradas aleatórias.

Análise dinâmica de sua execução: esta fase identifica como o programa se comporta com entradas aleatórias e com novas entradas geradas pela execução simbólica.

A utilização da técnica CONCOLIC possui bom desempenho, pois pode-se utilizar os valores da execução concreta para processar estruturas de dados complexas, bem como simplificar as restrições intratáveis. Porém, apesar das técnicas simbólica e CONCOLIC se mostrarem muito eficazes em programas pequenos, estas técnicas têm falhado ao processar programas grandes em que apenas uma pequena fração do grande número de possíveis caminhos de execução do programa são cobertos (BURNIM; SEN, 2008).

Diante desse cenário de baixa eficácia na execução de programas grandes, foi adotado o uso de estratégias de busca, guiadas pelo grafo de fluxo de controle dos programas, para maximizar o funcionamento da técnica CONCOLIC. Os autores demonstram experimentalmente que esta proposição maximiza a quantidade de ramos descobertos e promove a cobertura mais rápida do programa em comparação à estratégia de busca em profundidade, que é a estratégia de busca utilizada como padrão (BURNIM; SEN, 2008).

As quatro estratégias de busca propostas por Burnin e Sen (2008), são:

- *Control-Flow Directed Search*: o objetivo desta estratégia de busca é utilizar a estrutura estática do programa sob teste para orientar a busca dinâmica do seu caminho. Para isso, constrói-se o grafo de fluxo de controle de cada função a fim de se orientar a busca por caminhos que já possuem suas ramificações cobertas.
- *Uniform Random Search*: esta estratégia de busca foi inspirada na geração aleatória

de dados de entrada e propõe que o programa seja executado ao longo de caminhos aleatórios.

- *Bounded Depth-First Search*: o funcionamento desta estratégia de busca procura forçar todas as instruções condicionais que surgem durante o caminho de execução do programa, já que para cada condição dois ramos de execução diferentes podem ser obtidos. Para um número de condições $2d$ maior que zero, pode-se restringir a estratégia de busca a forçar o primeiro d número de ramos viáveis ao longo de qualquer caminho, já que a estratégia de busca irá encontrar $2d$ possibilidades de caminhos de execução, desde que todos os caminhos sejam executáveis.

- *Random Branch Search*: esta estratégia escolhe um dos ramos ao longo do caminho de forma aleatória e depois força a execução para que não seja conduzida por este ramo. A estratégia repete-se por diversas vezes, sempre com reinícios aleatórios, cobrindo novos ramos.

Para realização dos experimentos, os autores compararam o funcionamento da técnica CONCOLIC, atrelada à execução de suas quatro estratégias de busca, com um algoritmo de execução aleatória. Como *benchmarks*, foram escolhidos três programas de código aberto (open-source), sendo eles: *Replace*, processador de texto escrito em 600 linhas de código e integrante do *Siemens Benchmark Suite*; *Grep*, buscador de texto por expressões regulares, escrito em 15.000 linhas de código; Vim, editor de texto escrito em 150.000 linhas de código (BURNIM; SEN, 2008; VIM, 2011). Como critério de avaliação os autores limitaram o número de iterações das técnicas e compararam a quantidade de ramos cobertos usando-se cada uma das técnicas ao término de sua execução.

Como pode ser visto na Figura 3.3, ao executarem os experimentos sobre o programa *Replace*, todos os algoritmos que utilizaram a técnica CONCOLIC foram eficazes o suficiente a ponto de cobrir mais de 80% de todos os ramos da aplicação, sendo que os melhores resultados obtiveram cobertura de 90% de todos os ramos.

Ao serem feitos os experimentos no programa *Grep*, pode-se notar que as estratégias de busca *Random Branch Search* e *Control-Flow Directed Search* superaram os demais algoritmos e obtiveram resultados semelhantes entre si, enquanto que a estratégia de busca *Bounded Depth-First Search* teve eficácia baixa e apresentou resultados piores que o algoritmo aleatório. Esses resultados são apresentados na Figura 3.4.

A execução do experimento com o programa Vim mostrou que as estratégias de busca mais eficientes alcançaram cobertura de cerca de um terço dos ramos estimados como

Figura 3.3 – Resultados obtidos após avaliação do software Replace. Fonte: Burnim; Sen, 2006.

Figura 3.4 – Resultados obtidos após avaliação do software Grep. Fonte: Burnim; Sen, 2006.

acessíveis. As estratégias de busca Random Branch Search e Control-Flow Directed Search atingiram mais de duas vezes a cobertura dos outros métodos e demonstraram ser mais eficazes. A Figura 3.5 apresenta os resultados obtidos.

Assim, Burnin et al (2008), por meio dos resultados de seus experimentos sugerem que estratégias de busca sofisticadas, aquelas que se guiam por informações estáticas (e.g., grafo de fluxo de controle), permitem à técnica CONCOLIC obter maior cobertura de ramos em programas de maior porte.

Figura 3.5 – Resultados obtidos após avaliação do software Vim. Fonte: Burnim; Sen, 2006.

3.1.3 Teste baseado em busca

Em problemas complexos que exigem a escolha de uma solução em um conjunto demasiadamente grande de possíveis soluções, são exigidas abordagens automatizadas que possam tratar de forma eficiente os aspectos relacionados ao problema. O processo de geração automática de dados de teste se enquadra nesse cenário complexo, pois a seleção de dados de testes não pode ser facilmente descrita por meio de regras textuais ou passos registrados em documentos, além de ser caracterizada pela busca de uma solução apropriada em um espaço muito grande de possíveis soluções. Diante de problemas como este, a modelagem matemática de parâmetros e critérios de satisfação em relação a determinadas características se mostra a mais adequada (HARMAN, 2007; FREITAS et al., 2009).

Na engenharia de software baseada em busca (em inglês, *Search-based Software Engineering*), os problemas de engenharia de software são tratados como problemas de otimização de alta complexidade. Diante de problemas com essa dificuldade, o objetivo principal é otimizar uma função ou um grupo de funções de satisfação nas quais as variáveis que definem as funções de aptidão devem satisfazer um conjunto de equações criadas de acordo com cada instância do problema. As funções de aptidão (e as funções de restrição) devem ser lineares e apresentar continuidade; porém, muitos problemas de otimização presentes na engenharia de software não se enquadram nessas características. Nestes casos, a resolução pode ser feita por algoritmos metaheurísticos, tais como: Têmpera Simulada, Subida de Encosta, Algoritmos Genéticos e GRASP (*Greedy Randomized Adaptive Search Procedure*) (FREITAS et al., 2009).

Uma das primeiras utilizações de técnicas de otimização na resolução de problemas de engenharia de software foi documentada por Miller e Spooner (1976), que propunham a geração de dados de teste por meio de maximização numérica. O termo “Search-based Software Engineering” (SBSE) foi empregado em 2001, por Harman e Jones (2001), quando as pesquisas em torno do tema voltaram e tornaram-se intensas. A SBSE complementa as técnicas existentes e permite que problemas que não eram completamente resolvidos ou não tratados possam ser estudados e solucionados (FREITAS et al., 2009).

Algoritmos metaheurísticos representam um conjunto de algoritmos heurísticos que se baseiam em ideias de diversas fontes para solução de problemas de otimização. A função de aptidão (em inglês, *fitness*) pode ser pensada como uma medida de desempenho, lucratividade, utilidade e excelência que se queira maximizar (ARAKI, 2009).

A função de aptidão é associada ao grau de resistência e adaptabilidade ao meio onde o indivíduo vive. Com isso, indivíduos com maior aptidão terão maior chance de sobreviver e serão responsáveis pela próxima geração.

Algumas metaheurísticas amplamente difundidas são: Têmpera Simulada, Subida de Encosta, Algoritmos Genéticos e GRASP.

Nem sempre a solução retornada por um algoritmo metaheurístico é a melhor solução para um problema, porém sua utilização é oportuna em problemas com mais de uma função de aptidão ou em problemas em que não se conheça algum algoritmo exato que encerre a execução em tempo prático (FREITAS et al., 2009; PINHEIRO, 2010).

A utilização de funções de aptidão nesses algoritmos é muito comum, pois é o recurso indicador de quanto uma solução candidata é apropriada para o domínio de entrada. Essa informação funciona como guia para uma trajetória eficiente (SRIVASTAVA; KIM, 2009). Por isso, Harman (2007) ainda afirma que “o ser humano formaliza suas hipóteses em funções aptidão”.

Os principais algoritmos metaheurísticos citados na literatura estão relacionados nas seções seguintes.

3.1.3.1 Subida de Encosta

Em inglês *Hill-Climbing*, é uma técnica de otimização pertencente à família dos algoritmos de busca local. Devido ao seu modo de funcionamento, faz-se uma analogia da subida progressiva em uma encosta de uma paisagem. O algoritmo inicia com uma solução aleatória e a cada iteração executa pequenas alterações na solução, melhorando-a

pouco a pouco. Quando o algoritmo verifica que não existem melhorias a serem feitas, ele termina e apresenta uma solução ótima local. O algoritmo pode utilizar duas estratégias de busca: subida íngreme - toda vizinhança é analisada e assim eleger-se a melhor solução local; ou subida aleatória - a vizinhança é explorada aleatoriamente e substitui a solução corrente pela primeira que oferecer o melhor resultado (MCMINN, 2004).

As principais vantagens da utilização do algoritmo Subida de Encosta são: baixa utilização de memória e possibilidade de encontrar soluções razoáveis em conjuntos grandes ou infinitos. A desvantagem do algoritmo é que por ser um algoritmo de busca local, o algoritmo para no máximo local, isto é, a função de avaliação leva a um valor máximo para o caminho local que foi percorrido. Este problema pode ser resolvido utilizando técnicas de busca aleatória (MCMINN, 2004; PINHEIRO, 2010).

3.1.3.2 Têmpera Simulada

O algoritmo Têmpera Simulada (*Simulated Annealing*, em inglês), é um método probabilístico proposto por Kirkpatrick, Gelett e Vecchi, em 1983. O funcionamento do algoritmo é similar ao do algoritmo Subida de Encosta, porém fornece maneiras de escapar de máximos locais sem a utilização de busca aleatória. Para escapar dos máximos locais o algoritmo Têmpera Simulada utiliza *backtracking*, retrocedendo ao ponto anterior e tomando um novo caminho. Esses retrocessos são chamados de passos indiretos. A analogia feita a esta técnica, que deu origem ao nome Têmpera Simulada, está relacionada ao processo metalúrgico de endurecimento de vidros e metais, em que a fase de aquecimento representa a busca pela solução e a fase de resfriamento ao processo de reinicialização (retrocessos) (BERTSIMAS; TSITSIKLIS, 1993; BARROS; TEDESCO, 2008; PINHEIRO, 2010).

3.1.3.3 Algoritmos Genéticos

Os Algoritmos Genéticos fazem analogia à genética e à seleção natural. Com base nisso, seu objetivo é evoluir uma população por meio de competição, recombinação e mutação de seus indivíduos, de forma que a aptidão da população seja melhorada a cada iteração (PINHEIRO, 2010; SKINNER, 2010).

A execução mais comum de Algoritmos Genéticos segue as seguintes etapas:

a) Seleção: o tipo mais comum de seleção é a Seleção Roleta, na qual para cada indivíduo é atribuída uma probabilidade de sorteio, sendo que tal probabilidade é proporcional a sua aptidão (proximidade de solução para o problema). Então, depois de

atribuídas as probabilidades, dois indivíduos são escolhidos aleatoriamente (com base nessas probabilidades) e então produzem-se descendentes (PINHEIRO, 2010; SKINNER, 2010).

b) Recombinação: após a seleção arbitrária de dois indivíduos, devemos produzir descendentes com eles. A solução mais utilizada é chamada de cruzamento (em inglês, *crossover*), em que cada indivíduo descendente fica com uma parte do indivíduo pai. A Figura 3.6 apresenta um esboço de recombinação para o cruzamento. Às vezes, baseando-se em um conjunto de probabilidades, a recombinação não é executada e os indivíduos pais são copiados diretamente para a nova população (PINHEIRO, 2010; SKINNER, 2010).

Figura 3.6 – Esboço de recombinação Crossover. Fonte: Pinheiro, 2010.

Figura 3.7 – Esboço de mutação. Fonte: Pinheiro, 2010.

c) Mutação: depois de feita a seleção e a recombinação, é gerada uma nova população de indivíduos. Desta nova população, alguns indivíduos são originários de cruzamento e outros são simples cópias de seus indivíduos pais, para assegurar que não existem indivíduos iguais deve-se percorrer os novos indivíduos e alterar uma pequena parte para um novo valor. A taxa de mutação geralmente encontra-se entre 0,1% e 0,2%. A Figura 3.7 apresenta um esboço da mutação de um indivíduo (PINHEIRO, 2010; SKINNER, 2010).

3.1.3.4 Algoritmos Evolucionários

Tonella (2004) em sua proposta de geração de dados de teste apresenta seus casos de teste descritos por cromossomos aliados a algoritmos evolucionários, que incluem informações sobre quais objetos criar, quais métodos executar e quais valores devem ser utilizados como parâmetros de entrada.

O procedimento seguido para a construção dos casos de teste inclui alguns passos, aplicados a cada método sob teste. Tais passos são:

1. Um objeto da classe sob teste é criado utilizando um dos seus construtores disponíveis.
2. Uma sequência de zero ou mais métodos intermediários é chamada, a fim de construir um estado apropriado para o objeto.
3. O método sob teste é executado.

Prevê-se também que construtores, métodos intermediários e métodos sob teste possam exigir a passagem de objetos como parâmetros. Neste caso, prevê-se a repetição dos passos 1 e 2 recursivamente, até que todos os objetos necessários estejam disponíveis.

Assim, um caso de teste de uma classe consiste de uma sequência de criações de objetos, chamadas de métodos (para adequar os objetos aos seus devidos estados) e uma chamada final ao método sob teste.

A estrutura dos cromossomos pode ser bastante simples quando o teste evolucionário é aplicado a software procedimental, pois consiste basicamente da sequência de valores de entrada a serem fornecidos durante a execução de um programa. No caso do teste de software orientado a objetos uma simples sequência de valores de entrada não é suficiente. Assim, para o teste de software orientado a objetos, o caso de teste é uma sequência de construtores e chamadas a métodos, incluindo os valores de seus parâmetros.

A Figura 3.8 apresenta a visão macro do algoritmo evolucionário proposto por Tonella. O primeiro passo para execução do algoritmo é a identificação de todos os objetivos (e.g., ramos, nós) que devem ser cobertos pela geração de dados de teste. O segundo passo gera uma população inicial de forma aleatória. A execução do algoritmo gera novos casos de teste até que todos os objetivos sejam cobertos, ou até que o tempo máximo de execução do algoritmo seja atingido. A cada iteração um objetivo é selecionado dentro do conjunto de objetivos que ainda não foram cobertos. Em seguida, os casos de teste contidos na população são executados, a fim de cobrir o objetivo selecionado. Se o objetivo sob avaliação não for coberto por nenhum dos indivíduos da população, então a medida de aptidão de cada indivíduo é calculada. Depois do cálculo da medida de aptidão, uma nova população é criada por meio da extração dos melhores itens da população anterior. Essa extração é feita com base no valor da aptidão dos indivíduos. De acordo com a abordagem de Tonella, as medidas de aptidão resultam em valores dentro do intervalo de 0 a 1.

```

geracaoCasoTeste(classeSobTeste : Classe)
01  objetivosParaCobrir ← objetivos(classeSobTeste)
02  populacaoAtual ← gerarPopulacaoAleatoria(tamanho)
03  enquanto objetivosParaCobrir ≠ 0 e tempoExecucao < tempoMaximoExecucao
04      t ← seleccioneObjetivo(objetivosParaCobrir), tentativas ← 0
05      enquanto não coberto(t) e tentativas < maximaQtdeTentativas
06          execute casosDeTeste com populacaoAtual
07          atualize objetivosParaCobrir
08          se coberto(t) pare
09          calcule aptidao(t) de casosDeTeste com populacaoAtual
10          extraia novaPopulacao de populacaoAtual de acordo com aptidao(t)
11          mute novaPopulacao
12          populacaoAtual ← novaPopulacao
13          tentativas ← tentativas + 1
14      fim enquanto
14  fim enquanto

```

Figura 3.8 – Estrutura do algoritmo de Tonella. Fonte: Tonella, 2004.

As medidas de aptidão mais próximas a 1 correspondem aos indivíduos que chegam mais próximos de cobrir o objetivo, enquanto que as medidas de aptidão mais próximas a 0 correspondem aos indivíduos mais distantes de cobrir o objetivo. Ao criar uma nova população, reunindo apenas os indivíduos com as melhores medidas de aptidão (aquelas mais próximas a 1), aumenta-se a probabilidade de cobrir o objetivo, pois os indivíduos utilizados possuem características próximas às desejadas para cobri-lo. Depois de gerada a nova população, esta passa por um processo de mutação, no qual pequenas alterações são feitas nos indivíduos com a intenção de evoluí-los para que consigam cobrir o objetivo. Depois de mutada a nova população reinicia-se o fluxo de testes dos indivíduos e o algoritmo continua processando até que o tempo limite de execução seja atingido ou até que todos os objetivos sejam cobertos.

3.2 Representação de Dados de Teste

Para geração de dados de teste para programas orientados a objetos é necessário representar objetos, métodos e seus valores em uma codificação possível de ser executada por seus algoritmos. Essa seção apresenta dois modelos de representação de dados de teste encontrados na literatura.

Tonella (2004) propõe uma representação de dados de teste para o teste evolucionário

de software orientado a objetos. Sua representação especifica uma estrutura cromossômica que agrupa sequências de comandos, criação de objetos, mudanças de estados e chamada de métodos. Essa estrutura cromossômica constitui uma entrada de dados para um algoritmo de teste, a qual consideramos como indivíduo de teste. Na representação de Tonella um cromossomo (indivíduo) é dividido em duas partes, separadas pelo caractere “@” (arroba). A primeira parte contém uma sequência de ações (i.e., construtores e métodos), separadas pelo caractere “:” (dois pontos). Cada ação pode conter um novo objeto, atribuído a uma variável do cromossomo, indicada como “\$id”.

A segunda parte contém os valores de entrada dos métodos para serem usados nas suas chamadas. Valores de entrada de métodos ou construtores podem ser de tipos primitivos (i.e., int, double, boolean), separados pelo caractere “,” (vírgula).

Figura 3.9 – Aplicação da representação de Tonella. Fonte: Criado com base em Tonella (2004)

A Figura 3.9 apresenta o modo como a representação de Tonella é aplicada. Do lado esquerdo pode-se observar um bloco de código e do lado direito sua representação utilizando a representação de Tonella. Pode-se notar que os valores inteiros utilizados como parâmetros para os métodos são posicionados do lado direito do símbolo de “@” (arroba), enquanto que as chamadas a métodos são posicionadas do lado esquerdo. Vale ressaltar a sintaxe utilizada para representar a construção de instâncias de objetos e as chamadas a métodos. No caso, a instrução “A a = new A();” foi escrita na representação de Tonella (2004) com a sintaxe “\$a=A()”, assim como a sintaxe da chamada de método “b.f(2);” foi representada com a sintaxe “\$b.f(int)”. Nota-se que a representação de Tonella mantém a apresentação do indivíduo de forma intuitiva, o que facilita a sua leitura e compreensão. Todo o conjunto de instruções foi adequado a uma nova representação que ordena todos os comandos em uma única linha.

A geração de valores para os parâmetros de tipo primitivo é aleatória, mas segue algumas regras, como:

Valores inteiros e de ponto flutuante – valores inteiros e de ponto flutuante são selecionados no intervalo de 0 a 100.

Booleanos – valores booleanos true (verdadeiro) e false (falso) são escolhidos aleatoriamente, assumindo probabilidade igual (0,5).

Strings e caracteres – valores escolhidos uniformemente dentre os caracteres alfanuméricos (i.e., [a-z A-Z 0-9]).

Segundo Silva e Someren (2010) um dos principais fatores que levam à escolha da representação de Tonella é o risco de utilizar uma estrutura na qual deve-se tomar cuidado com a compatibilidade dos parâmetros quando testados os métodos, a fim de que não haja corrompimento dos cromossomos. A estrutura de Tonella resolve este problema por meio de uma representação bem estruturada de composição dos cromossomos. Silva e Someren (2010) ainda afirmam que outra vantagem de sua utilização é a possibilidade de desacoplar o cromossomo do sistema, tornando fácil a aplicação e construção de diferentes operadores de mutação e *crossover*.

A representação de Tonella não é a única utilizada. Segundo Tracey *et al.* (2002), os algoritmos genéticos tradicionalmente utilizam codificação binária para suas soluções. Isso decorre do desenvolvimento histórico dos algoritmos genéticos, no qual se faz uma analogia entre os bits e os cromossomos na evolução natural. No entanto, para geração de dados de teste, três problemas podem ser encontrados: perda de informação, corrupção binária e disparidade espacial da solução original. A disparidade espacial é causada por soluções muito próximas no espaço de solução, mas que são muito distantes no espaço de solução codificado. Por exemplo, a representação binária do número 31 é “1 1 1 1 1”, enquanto que o número 32 (o número posterior a 31) a representação é “1 0 0 0 0”. Isto é, dois números tão próximos na solução original, mas que são muito diferentes em suas codificações. Operações de mutação e *crossover* têm dificuldade de executar movimentos entre estas duas soluções.

Para resolver esta questão, os autores propõem o uso da codificação de Gray (TRACEY *et al.*, 2002, p. 9) como forma de solução para o problema da disparidade espacial. A codificação de Gray auxilia com os tipos numéricos, mas não ajuda com tipos não-numéricos, além de não resolver o problema da perda de informação e da corrupção binária.

A perda de informação ocorre durante o *crossover*, quando a representação binária de um valor é acidentalmente partida em duas partes gerando um terceiro valor, diferente dos valores dos indivíduos que originaram a evolução. Conforme a Figura 3.10, podemos

identificar dois indivíduos P_1 e P_2 , cujos atributos X possuem valores diferentes (35 e 12, respectivamente), mas que após uma divisão errada do indivíduo para execução do crossover, geram um novo indivíduo com um valor para X diferente dos demais (TRACEY et al., 2002).

Outro problema enfrentado é a corrupção binária que ocorre durante a mutação de um cromossomo, devido à cópia indevida de alguns bits de uma representação para outra, modificando o valor de um número que não deveria ser alterado. Na Figura 3.11 pode-se observar que a mutação copia indevidamente os primeiros bits da representação de um atributo que não deveria ser mutado, acarretando na mutação de mais de um atributo ao mesmo tempo (TRACEY et al., 2002).

Figura 3.10 – Codificação binária - perda de informação. Fonte: Tracey et al., 2002.

Figura 3.11 – Codificação binária - corrupção binária. Fonte: Tracey et al., 2002.

3.2.1 Operações com indivíduos de teste

Em sua abordagem, Tonella (2004), propõe um conjunto de operadores de mutação e *crossover*, com o objetivo de evoluir os indivíduos da população para obter uma população mais adaptada. Os operadores de Tonella funcionam da seguinte maneira:

Mutação de valores de entrada: o valor de um parâmetro primitivo é substituído por outro valor, de mesmo tipo, aleatoriamente gerado.

$$\begin{aligned} \$a=A() : \$b=B() : \$b.f(int) : \$a.m(int, \$b) @ \underline{2}, 5 \\ \$a=A() : \$b=B() : \$b.f(int) : \$a.m(int, \$b) @ \underline{6}, 5 \end{aligned}$$

Figura 3.12 – Representação de Tonella: mutação de valores de entrada

A Figura 3.12 mostra o parâmetro passado para o método “f(int)” da classe “B” alterado de 2 para 6.

Mudança de construtor: um dos construtores do indivíduo é aleatoriamente trocado. Esse operador pode acarretar a inclusão de novos parâmetros ou objetos na estrutura do indivíduo, dependendo dos parâmetros esperados pelo novo construtor.

$$\begin{aligned} \$a=A() : \$b=B() : \$b.f(int) : \$a.m(int, \$b) @ 2, 5 \\ \$a=A() : \underline{\$c=C() : \$b=B(\$c, int)} : \$b.f(int) : \$a.m(int, \$b) @ \underline{3}, 2, 5 \end{aligned}$$

Figura 3.13 – Representação de Tonella: mudança de construtor

No exemplo da Figura 3.13, o construtor da classe “B” é substituído por outro construtor, que recebe dois parâmetros, onde um deles é outro objeto, do tipo “C”, e o segundo parâmetro é de um tipo primitivo.

Inclusão de chamada a método: novas chamadas a métodos são incluídas aleatoriamente na estrutura do indivíduo. Essa operação pode exigir a inclusão de novos parâmetros (i.e., objetos, variáveis de tipos primitivos).

$$\begin{aligned} \$a=A() : \$b=B() : \$b.f(int) : \$a.m(int, \$b) @ 2, 5 \\ \$a=A() : \$b=B() : \underline{\$b.g(int)} : \$b.f(int) : \$a.m(int, \$b) @ \underline{3}, 2, 5 \end{aligned}$$

Figura 3.14 – Representação de Tonella: inclusão de chamada a método

Como se pode observar, no exemplo da Figura 3.14, foi incluída uma chamada ao método “g(int)”, que acarretou a inclusão de um parâmetro de tipo primitivo com valor 3.

Remoção de chamada a método: na estrutura do indivíduo, após a aplicação deste operador, um dos métodos intermediários é removido, junto com seus respectivos parâmetros.

$$\begin{aligned} \$a=A() : \$b=B() : \$b.f(int) : \$a.m(int, \$b) @ 2, 5 \\ \$a=A() : \$b=\overline{B()} : \$a.m(int, \$b) @ 5 \end{aligned}$$

Figura 3.15 – Representação de Tonella: remoção de chamada a método

O exemplo da Figura 3.15 mostra a aplicação deste operador removendo a chamada do método intermediário “f(int)” da classe “B”.

Crossover: Depois de selecionar dois indivíduos aleatoriamente, trocam-se partes dos cromossomos entre si, com o objetivo de gerar dois novos cromossomos na população.

$$\begin{aligned} \$a=A() : \$b=B() : \$b.f(int) : \$a.m(int, \$b) @ \underline{2}, 5 \\ \$a=A() : \$b=B() : \$b.n(int) : \underline{\$b.g(int,int)} : \$a.m(int, \$b) @ 8, \underline{4}, 7, 6 \\ \$a=A() : \$b=B() : \underline{\$b.g(int,int)} : \$a.m(int, \$b) @ \underline{4}, 7, 5 \\ \$a=A() : \$b=B() : \$b.n(int) : \underline{\$b.f(int)} : \$a.m(int, \$b) @ 8, \underline{2}, 6 \end{aligned}$$

Figura 3.16 – Representação de Tonella: crossover

Nota-se na Figura 3.16 que no exemplo apresentado o método intermediário “f(int)” da classe “B” do primeiro cromossomo, é substituído pelo método intermediário “g(int, int)” da classe “B” do segundo cromossomo, junto com seus respectivos parâmetros. Da mesma forma que o segundo indivíduo troca o método “g(int, int)” da classe “B” pelo método intermediário “f(int)” da classe “B” do primeiro cromossomo, junto com seus respectivos parâmetros.

3.3 Desafios para geração de dados de teste

O processo de construção de ferramentas geradoras de dados de teste não é fácil. Alguns recursos computacionais são, porém, mais caros de ser tratados do que outros durante o processo de geração automática de dados. Esta seção tem por objetivo destacar alguns dos problemas enfrentados.

3.3.1 Vetores e ponteiros

Vetores e ponteiros são construídos de forma similar e acarretam o mesmo tipo de problema. Durante os testes é difícil a representação de vetores e ponteiros, pois estes elementos são de difícil substituição e representação quando seus valores ou variáveis não são conhecidos. O exemplo da Figura 3.17 apresenta uma condição que utiliza o elemento de um vetor sendo indexado por uma variável (EDVARDSSON, 1999).

```
input(i,j);  
a(j) = 2;  
a(i) = 0;  
a(j) = a(j) + 1;  
if (a(j) = 3) ...
```

Figura 3.17 – Exemplo de utilização de um vetor. Fonte: Edvardsson, 1999.

Levando-se em consideração o exemplo da Figura 3.17, nota-se que o bloco de código exemplificado apresenta o manuseio de um vetor de inteiros. Na primeira linha pode-se notar a entrada de duas variáveis, *i* e *j*. Após a entrada dessas variáveis são utilizados seus valores para o manuseio do vetor, no qual o espaço de memória indexado pelo valor de *j* recebe o inteiro 2, e o espaço de memória indexado pelo valor de *i* recebe o inteiro 0. Logo em seguida uma operação soma o valor do espaço de memória indexado por *j* com 1, e atribui o resultado ao próprio espaço de memória indexado por *j*. Assim, espera-se que no final da execução o valor contido no espaço de memória indexado por *j* seja igual a 3, mas este comportamento não ocorre quando as entradas das variáveis *i* e *j* atribuem o mesmo valor para ambas. Mas este problema de indexação não é o único a ser considerado, deve-se estar pronto para controlar estruturas que possam ficar demasiadamente grandes e que acarretem estouros de memória (EDVARDSSON, 1999).

3.3.2 Objetos

A geração automática de objetos no teste de programas orientados a objetos é difícil, pois os objetos são alocados dinamicamente. Dessa forma, a utilização de conceitos como herança, polimorfismo e classes abstratas torna impossível determinar em tempo de compilação qual código será chamado. Isso demonstra que qualquer solução para resolver este problema deve ser dinâmica, isto é, dependerá de uma execução prévia do programa sob teste (EDVARDSSON, 1999).

Ao contrário do software procedimental, o software orientado a objetos nem sempre se comporta da mesma maneira ao executarmos o mesmo método com os mesmos valores de entrada. Essa afirmação exemplifica uma das dificuldades de testar software orientado a objetos.

Uma das maiores peculiaridades do software orientado a objetos é o controle de estados. O estado de um objeto caracteriza sua composição em um dado momento de tempo, e pode modificar o comportamento de seus métodos. O controle de estados é tão importante que dentro da UML temos diversos diagramas (i.e., Diagrama Temporal, Diagrama de Estados) que descrevem a troca de estados dos objetos. Para ilustrar tal dificuldade, tomemos como exemplo a classe ilustrada na Figura 3.18.

```
public class SampleRex {  
    private int _sault;  
  
    public SampleRex() {  
        this._sault = 0;  
    }  
  
    public void ChangeRex() {  
        this._sault = 11;  
    }  
  
    public void AlterRex() {  
        this._sault = -42;  
    }  
  
    public int MultipleRex(int value) {  
        if (this._sault > 0)  
            return value * 10;  
        else if (this._sault < 0)  
            return value * -100;  
        else  
            return value;  
    }  
}
```

Figura 3.18 – Classe para teste de estados de objetos. Criado com base em Tonella (2004).

Para criarmos um caso de teste para o método “MultipleRex(int)” poderíamos tomar como exemplo o código da Figura 3.19. Esse código diz respeito a um teste de unidade chamado “TestMethodA”; neste método unitário é executado o método “MultipleRex” sendo passado o valor inteiro 5. Para assegurar que o teste de unidade foi executado com sucesso são utilizados métodos pertencentes à classe “Assert” para averiguação do valor de retorno do método sob teste.

```
[TestMethod]
public void TestMethodA() {
    SampleRex miniRex = new SampleRex();

    int returnValue = miniRex.MultipleRex(5);

    Assert.AreEqual(5, returnValue);
}
```

Figura 3.19 – Teste de unidade A. Criado com base em Tonella (2004).

Após observar o comportamento do método “MultipleRex(int)”, vê-se que existe dependência a um atributo interno. Tal dependência é tão forte que o valor deste atributo influencia quais blocos de código podem ou não ser executados. Um conjunto de condições aninhadas, que validam o atributo “_sault” direciona a execução do método para um bloco de código distinto dos demais.

Se incluída uma chamada intermediária de método entre a execução do método sob teste e a instância do objeto, será alterado o comportamento do método sob teste, mesmo passando os mesmos valores de entrada. A Figura 3.20 exemplifica esta situação.

```
[TestMethod]
public void TestMethodB()
{
    SampleRex miniRex = new SampleRex();

    miniRex.AlterRex();

    int returnValue = miniRex.MultipleRex(5);

    Assert.AreEqual(5, returnValue);
}
```

Figura 3.20 – Teste de unidade B. Criado com base em Tonella (2004).

A execução do método “AlterRex()” modifica o valor do atributo privado e altera os blocos de código executados pelo método “MultipleRex(int)”, acarretando um valor de retorno diferente do esperado.

Se modificado o teste, e trocado o método intermediário, promove-se a execução de um terceiro comportamento distinto. A alteração pode ser vista na Figura 3.21:

Assim, pode-se notar que o teste de software orientado a objetos exige atenção durante a construção dos estados dos objetos, para que os casos de teste sejam abrangentes o suficiente, a fim de executar todos os possíveis blocos do método sob teste.

```
[TestMethod]
public void TestMethodC()
{
    SampleRex miniRex = new SampleRex();

    miniRex.ChangeRex();

    int returnValue = miniRex.MultipleRex(5);

    Assert.AreEqual(5, returnValue);
}
```

Figura 3.21 – Teste de unidade C. Criado com base em Tonella (2004).

3.3.3 Laços de repetição

No caso da execução simbólica não se pode processar um laço de repetição a menos que o número de iterações seja precisamente conhecido. Se o número de iterações de um laço de repetição depender de algum valor de entrada, é necessário deduzir o valor de cada variável referenciada dentro do laço de repetição. Outra opção é a manipulação de um laço de repetição executando-o de forma simbólica k vezes, sendo que as k vezes devem ser especificadas pelo usuário ou escolhidas pelo sistema gerador de dados de teste. Essa alternativa é adotada pela maioria dos geradores de dados de teste devido a sua simplicidade de implementação (RAMAMOORTHY et al., 1976; EDVARDSSON, 1999).

3.3.4 Módulos

Geralmente, um programa de linguagem procedimental é dividido em funções ou procedimentos. Existem dois meios de executar simbolicamente uma função (não recursiva) que possui outra chamada de função (RAMAMOORTHY et al., 1976; EDVARDSSON, 1999):

- Primeira solução: executar substituições da função chamada dentro da função chamadora – esta é a solução mais adotada. Essa solução visa a simulação do modo como o programa é executado; entretanto, o número de possíveis caminhos de execução pode crescer rapidamente.

- Segunda solução: pré-processar o programa, acompanhar sua execução, e identificar quais caminhos são percorridos e o caminho no qual cada módulo é executado para encontrar suas restrições de caminho. Esta solução é mais adequada para programas grandes.

Outra grande dificuldade ocorre quando são encontrados códigos-fonte de funções e

procedimentos que não podem ser acessados (e.g., bibliotecas pré-compiladas) e que, desta forma, não podem ser avaliados (EDVARDSSON, 1999).

3.3.5 Caminhos não executáveis

Dependendo da técnica de geração de dados de teste, para percorrer um caminho exige-se a solução de um sistema de equações. Se esse sistema de equações não possui solução, então pode-se concluir que o caminho dado é inviável (não executável). Nessas condições devemos considerar a utilização de um número máximo de tentativas antes de considerar um caminho como não executável (EDVARDSSON, 1999).

3.4 Considerações finais

Neste capítulo foram detalhadas três abordagens de geração de dados de teste com o objetivo de obter a cobertura com relação a critérios de teste estrutural, a saber: geração aleatória, execução simbólica e algoritmos metaheurísticos. Também foram apresentadas as motivações para o uso de algoritmos de otimização na resolução de problemas complexos, como a geração automática de dados de teste. E por fim, foram descritos modelos de representação de dados de teste encontrados na literatura e dificuldades enfrentadas durante o processo de geração de dados de teste. O Capítulo 4 apresenta o levantamento bibliográfico e resume abordagens para a construção de *frameworks* para geração de dados de teste.

Capítulo 4

Frameworks Geradores de Dados de Teste

Este capítulo apresenta um levantamento bibliográfico sobre *frameworks* geradores de dados de teste. Com o objetivo de obter uma visão ampla da pesquisa sobre os *frameworks* geradores de dados de teste foi conduzida uma revisão sistemática. A divisão deste capítulo se faz em duas seções: na Seção 4.1 apresenta-se a revisão sistemática conduzida, e na Seção 4.2 são apresentados os trabalhos relevantes identificados.

4.1 Identificação de Trabalhos

O objetivo da revisão sistemática foi identificar trabalhos que desenvolveram *frameworks* como proposta para automatizar a geração de dados de teste para software orientado a objetos, além de identificar ferramentas desenvolvidas, experimentos realizados e analisar as características dos trabalhos existentes. Para atender aos objetivos propostos, foram elaboradas duas questões de pesquisa principais (Q1 e Q2), cada uma com critérios próprios de inclusão e exclusão de trabalhos:

Q1 Existem *frameworks* de geração de dados de teste genéricos para programas orientados a objetos?

Q2 Quais as técnicas utilizadas para geração de dados de teste de programas orientados a objetos?

Para seleção de conteúdo foram utilizadas bibliotecas digitais, sendo elas: ACM (*Association for Computing Machinery*), IEEE (*Institute of Eletronical and Eletronics Engineers*) e *Microsoft Research*. Também foram utilizadas bases eletrônicas indexadas, tais como: *Google Scholar* e *Citeseer*, além de anais de eventos, tal como o principal simpósio da área, intitulado *Symposium on Search Based Software Engineering* (SSBSE). De acordo com o interesse da pesquisa, foram utilizadas diferentes cadeias de busca para cada uma das bibliotecas, sendo:

ACM (<http://portal.acm.org/>) “test data generation” AND “*Framework*” AND “object-oriented software”

IEEE (<http://ieeexplore.ieee.org/Xplore/guesthome.jsp>) “*Framework* data test generation object-oriented”

CiteSeerX (<http://citeseerx.ist.psu.edu/>) title:(framework test data)

As pesquisas foram realizadas no período de 1 de Julho de 2011 a 20 de Julho de 2011.

Os critérios de inclusão de artigos utilizados foram: **C1** - Trabalhos que abordem a utilização ou construção de *frameworks* para geração de dados de teste; **C2** - Trabalhos que apresentem abordagens para o tratamento de estruturas complexas para geração automática de dados de teste de software orientado a objetos.

Os critérios de exclusão de artigos utilizados foram: **E1** - Trabalhos que utilizem *frameworks* como forma de solução para problemas diferentes da geração automática de dados de teste; **E2** - Trabalhos que não abordem a geração de dados de teste para software orientado a objetos.

Foram identificados 62 trabalhos, cuja leitura e aplicação dos critérios de inclusão e exclusão, acarretaram a inclusão de 16 artigos (listados na Tabela 4.1) e a exclusão dos demais artigos. De forma complementar, foram aplicados critérios para avaliação dos trabalhos selecionados, a fim de destacar os trabalhos que possuem técnicas mais apuradas dentre os demais. Esta avaliação foi baseada em critérios escolhidos a partir de características consideradas como necessárias em *frameworks* de geração de dados de teste. Tais critérios são:

Flexibilidade do framework. Este critério avalia o quão extensível o *framework* gerador de dados de teste é, a ponto de permitir a composição de técnicas de geração de teste de forma simples, integrada e consistente.

Quantidade de técnicas. Para que o *framework* seja versátil, é preciso prover um conjunto de técnicas distintas de geração de dados de teste, não bastando apenas fornecer uma simples técnica de geração de dados de teste, o que limita a sua utilização a uma única abordagem.

Quantidade de funções de aptidão. Assim como é importante que o *framework* seja versátil, também é importante que sejam parametrizáveis seus critérios de satisfação (funções objetivo), com o objetivo de que o usuário final do *framework* possa escolher

diferentes critérios de satisfação a fim de obter diferentes resultados, para poder selecionar aquele que for melhor adequado a sua necessidade.

Após a aplicação destes critérios, foram selecionados cinco trabalhos como os mais relevantes da área de pesquisa, os quais serão detalhados logo a seguir.

Tabela 4.1 – Artigos selecionados após critérios de seleção da revisão sistemática

Abordagem	Características
(ARAKI; VERGILIO, 2010)	<i>Framework</i> de geração de dados de teste que satisfaz critérios baseados em fluxo de controle e em fluxo de dados considerando o <i>bytecode</i> Java do software sob teste e que utiliza funções multi-objetivo para construção de seus casos de teste.
(INKUMSAH; XIE, 2008)	Abordagem que executa a geração de dados de teste por meio da mescla de teste evolucionário e execução simbólica.
(BARESI; MIRAZ, 2010)	Abordagem que faz uso de algoritmo evolutivo e subida de encosta para geração automática de dados de teste.
(MIRAZ et al., 2009)	Abordagem que faz uso de algoritmo evolutivo e subida de encosta para geração automática de dados de teste.
(WANG et al., 2010)	Abordagem que propõe a separação do código e dos dados, a fim de diminuir a redundância dos mesmos.
(SILVA; SOMEREN, 2010)	<i>Framework</i> de geração de dados de teste que utiliza algoritmos genéticos e análise estática.
(FIX, 2009)	Ferramenta de teste que auxiliada de trabalho manual faz a criação de testes unitários a partir da edição de arquivos XSLT.
(LIASKOS; ROPER, 2007)	Proposta de experimento utilizando algoritmos genéticos e imunológicos para geração automática de dados de teste, em que os nós alvos que são difíceis de serem cobertos utilize-se um sistema imunológico artificial com um caminho orientado, que possa guiar a busca pelas mais promissoras área do espaço de pesquisa.
(TRACEY et al., 1998)	Abordagem que faz uso do algoritmo metaheurístico <i>têmpera simulada</i> para geração de dados de teste.
(NADEEM; LYU, 2006)	Teste manual que faz a geração de casos de teste para teste de herança utilizando o formato de especificação VDM++.
(DENG et al., 2007)	Utiliza a assinatura dos métodos para apoiar a construção de objetos <i>mock</i> (objetos que simulam o comportamento de objetos reais de forma controlada) de sistemas orientados a objetos.
(SHIROLE et al., 2011)	Abordagem de teste baseada em um modelo para geração de casos de teste que combinam informações de diagramas UML de estado com algoritmos genéticos. Nos experimentos foram gerados casos de teste transformando máquinas finitas de estado estendidas em grafos de fluxo de controle.
(SAGARNA et al., 2007)	A abordagem gerar dados de teste através de uma família de algoritmos que a cada passo induzem uma probabilidade de distribuição dos indivíduos selecionados.
(BOYAPATI et al., 2002)	A ferramenta Korat lê a especificação descrita no cabeçalho dos métodos (via <i>Java Modeling Language</i>) e identifica o espaço de estado dos valores de entrada, e com base nesses valores faz a geração de dados de teste.
(GOTLIEB et al., 2000)	Proposta de utilização de restrições de lógica de programação para geração automática de dados de teste, por meio de inteligência artificial e PROLOG ¹ .
(DENG et al., 2006)	Abordagem que utiliza execução simbólica e resolução de restrições para descoberta de defeitos.

4.2 Trabalhos Relevantes

Nesta seção serão apresentados os trabalhos mais relevantes relacionados a *frameworks* geradores de dados de teste. Dentre os trabalhos selecionados os mais relevantes são: Evacom (INKUMSAH; XIE, 2008), TestFul (MIRAZ et al., 2009; BARESI; MIRAZ, 2010), TDS-

Gen/OO (ARAKI; VERGILIO, 2010), AutoTest/Eiffel (SILVA; SOMEREN, 2010) e Têmpera Simulada/Ada (TRACEY et al., 1998).

Outros trabalhos também importantes foram analisados, mas não foram resumidos neste trabalho, porém seus principais pontos são resumidos a seguir e também apresentados na Tabela 4.1. Os trabalhos (WANG et al., 2010), (FIX, 2009), (NADEEM; LYU, 2006) e (BOYAPATI et al., 2002) não foram incluídos, pois exigem algum tipo de interferência manual para geração dos dados de teste. Os trabalhos (LIASKOS; ROPER, 2007) e (GOTLIEB et al., 2000) são propostas para as quais não foram identificados os trabalhos concretos. O trabalho (SHIROLE et al., 2011) combina a utilização de diagramas UML com a geração de dados de teste, enquanto que o trabalho (SAGARNA et al., 2007) faz uso de cálculos probabilísticos para criação de indivíduos. As abordagens (DENG et al., 2007) e (DENG et al., 2006) possuem uma linha de trabalho muito próxima com os trabalhos (INKUMSAH; XIE, 2008) e (SILVA; SOMEREN, 2010), mas não foram incluídos por abordarem apenas a execução simbólica e não combinarem outra técnica, como algum tipo de algoritmo metaheurístico.

4.2.1 Evacom

Inkumsah e Xie (2008) propõem um *framework* intitulado Evacom que integra algoritmo evolucionário (conforme apresentado na Seção 3.3) com execução simbólica. O objetivo dessa mistura é fortalecer a geração de dados de teste, obtendo melhores resultados por meio do trabalho em conjunto dessas duas técnicas, gerando casos de teste que possuam maior cobertura de ramos do que os casos de teste gerados por cada técnica individualmente.

O algoritmo evolucionário não utiliza a estrutura do programa ou algum conhecimento semântico para geração de dados de teste. Esta característica não é vantajosa, segundo os pesquisadores, pois o teste evolucionário não consegue fornecer valores tão precisos quanto o desejado para os parâmetros primitivos (i.e., int, char, double, boolean), ao contrário da execução simbólica, que é a técnica ideal para geração de valores para parâmetros primitivos, mesmo quando estes estão atrelados a predicados difíceis de serem cobertos. Por outro lado, a execução simbólica não provê recursos para construção de sequências de chamada de métodos, funcionalidade necessária para construção de diferentes estados do objeto, e também não fornece recursos para construção de parâmetros não primitivos (i.e., instâncias de objetos).

Inkumsah e Xie (2008) combinam as duas técnicas de modo que o algoritmo evolu-

cionário guia a geração de dados de teste com a utilização da execução simbólica. Esta abordagem faz com que a execução simbólica fique dedicada à geração de valores para os parâmetros primitivos, enquanto que o algoritmo evolucionário fica encarregado de construir sequências de métodos para construção de estados do objeto sob teste, além de construir valores para os parâmetros não primitivos. A ferramenta Evacom é dividido em quatro componentes, sendo eles:

Teste Evolucionário. O algoritmo evolucionário adotado segue a representação e a estrutura propostas por Tonella (2004). De forma complementar, a fim de melhorar os resultados obtidos, Evacom substitui a geração aleatória, que formula a população inicial do algoritmo, por uma população gerada por uma pré-execução simbólica do método sob teste. Quando um requisito de teste não pode ser coberto pela população existente, a aptidão de cada indivíduo é calculada. A aptidão de um indivíduo é baseada na relação entre as bordas de controle e os ramos percorridos durante a execução do indivíduo de teste sobre as bordas de controle e os ramos que levam para o requisito a ser coberto.

Execução Simbólica. Para execução simbólica, a ferramenta Evacom utiliza uma ferramenta de execução simbólica chamada jCUTE (SEN et al., 2005), cujo objetivo é definir valores para os parâmetros primitivos dos métodos.

Transformação de Parâmetros Simbólicos. Esta transformação permite ao jCUTE fazer execução simbólica e concreta dos parâmetros primitivos. Para cada chamada de método que exigir ao menos um parâmetro primitivo, tal parâmetro é substituído por um parâmetro simbólico equivalente, utilizado para guiar a execução simbólica.

Construção de Cromossomos. O componente de construção de cromossomos constrói o cromossomo de acordo com a representação de Tonella (2004). A construção do cromossomo envolve dois passos: primeiro, extração das sequências de métodos intermediários; segundo, transformação das sequências em cromossomos.

Para realização das avaliações, Inkumsah e Xie reuniram 13 classes utilizadas nos testes de Rostra (*framework* para detecção de testes de unidade redundantes) (XIE et al., 2004), de um gerador de dados de entrada para coleções Java (VISSER et al., 2006) e no algoritmo de Tonella (2004) com o objetivo de comparar os resultados obtidos por tais abordagens com os resultados obtidos por Evacom.

Tabela 4.2 – Classes utilizadas no teste de Evacon. Fonte: INKUMSAH; XIE, 2008

Classe	métodos públicos	Ramos	LOC
BankAccount	6	6	60
BinarySearchTree	16	67	260
BinomialHeap	10	94	215
BitSet	25	130	638
DisjSet	6	44	140
FibonacciHeap	9	92	207
HashMap	10	89	374
LinkedList	29	105	738
ShoppingCart	6	13	117
Stack	5	16	160
StringTokenizer	5	47	222
TreeMap	47	252	1626
TreeSet	13	20	301

O tamanho das classes varia entre 60 linhas de código e 1626 linhas de código. O número de métodos públicos varia entre 6 e 47 e o número de ramos nas classes varia de 6 a 252. A Tabela 4.2 apresenta as características de tais classes.

Para a comparação entre as técnicas, foram selecionadas quatro ferramentas geradoras de teste, que representam as técnicas de geração de dados de teste mais utilizadas. As quatro abordagens selecionadas são:

- *eToc*: ferramenta de teste evolucionário para programas orientados a objetos utilizando algoritmos genéticos.
- *jCute*: ferramenta de geração de dados de teste por meio de execução simbólica.
- *JUnit Factory*: ferramenta experimental de geração de dados de teste, como representante das ferramentas industriais de geração de dados de teste.
- *Randoop*: representante das técnicas de geração aleatória de dados de teste.

Para a realização dos testes foram aplicados dois tipos de integração utilizando Evacon: a primeira faz ligação entre a ferramenta *eToc* para o *jCute* por meio do componente de transformação de parâmetro (nomeado como Evacon-A); e a segunda faz integração da ferramenta *jCute* para a ferramenta *eToc* por meio do componente de construção de cromossomos (nomeado como Evacon-B).

A Tabela 4.3 apresenta a cobertura obtida com os experimentos. Para comparar as seis abordagens, foi medida a cobertura de ramos atingida por cada uma das técnicas em

Tabela 4.3 – Cobertura de ramos obtida pelas seis abordagens testadas. Fonte: INKUM-SAH; XIE, 2008

Classe	Tempo (segundos)	Evacon-A (%)	Evacon-B (%)	eToc (%)	jCUTE (%)	JUnit Fact (%)	Randoop (%)	Todos (%)
BankAccount	28	100.0	100.0	100.0	100.0	100.0	100.0	100.0
BinarySearchTree	238	94.0	95.5	92.5	88.6	88.1	98.5	100.0
BinomialHeap	228	94.7	92.6	90.4	89.9	88.3	95.7	98.9
BitSet	559	96.2	90.8	89.2	46.1	92.3	96.9	100.0
Disjset	346	95.5	93.8	90.9	55.8	86.4	90.9	100.0
FibonacciHeap	229	97.8	96.7	95.7	84.3	73.9	81.5	100.0
HashMap	374	95.5	91.0	80.9	58.8	80.9	60.7	100.0
LinkedList	687	81.0	77.1	79.0	64.0	67.6	1.0	100.0
ShoppingCart	145	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Stack	28	100.0	93.8	93.8	95.8	93.8	100.0	100.0
StringTokenizer	291	100.0	97.9	89.4	80.4	100.0	91.5	100.0
TreeMap	276	59.9	57.9	48.8	53.4	23.4	15.5	74.6
TreeSet	135	95.0	85.0	65.0	86.1	25.0	80.0	95.0
Média	274	93.05	90.16	85.82	77.17	78.44	77.86	97.58

um mesmo período de tempo (conforme demonstrado na coluna “Tempo(segundos)” da Tabela 4.3).

Os resultados obtidos mostram que Evacon (tanto Evacon-A, quanto Evacon-B) apresentaram melhores resultados em relação às demais abordagens. Isto sugere que para algumas classes, a utilização de uma única técnica pode não ser suficiente para atingir o melhor caso e a combinação de múltiplas técnicas pode ser benéfica.

4.2.2 TestFul

TestFul (MIRAZ et al., 2009; BARESI; MIRAZ, 2010) é uma proposta de geração de dados de teste que adota a ideia de que testes devem colocar os objetos em seus devidos estados e fornecer os valores para os parâmetros de entrada das funções sob teste. Assim, TestFul divide seu trabalho em duas partes: a primeira cria o estado desejado dos objetos e a segunda executa o comportamento real.

O algoritmo de TestFul identifica estados dos objetos e tenta executar pequenas modificações em seus valores a fim de melhorá-los, utilizando Algoritmo Evolucionário e Subida de Encosta. Para tanto, TestFul faz uma pré-análise da classe sob teste, com o objetivo de identificar todas as classes que devem ser envolvidas no teste. Isso é feito considerando todos os parâmetros de todos os construtores e métodos públicos contidos na classe sob teste.

Após a pré-análise, TestFul dá início a um algoritmo evolucionário que segue a estru-

tura e a representação de Tonella (2004). O algoritmo evolucionário gera aleatoriamente uma população inicial, executando o método sob teste utilizando esta população e monitorando sua execução, a fim de identificar se o ramo alvo foi coberto pelo caso de teste. Caso o ramo alvo não seja executado, são feitas pequenas alterações nos valores da população, utilizando Subida de Encosta, com a intenção de melhorar os valores dos parâmetros de entradas, para que o ramo alvo seja coberto. Caso o ramo alvo ainda não seja coberto, então é calculada sua aptidão (diferença entre o ramo alvo e o ramo executado), cujo valor é utilizado como parâmetro para criação de uma nova população para novos testes. Quando o ramo alvo for coberto, então o teste gerado tem sua cobertura estrutural de ramos melhorada, criando a próxima população com base no caso de teste que resultou na cobertura do último ramo alvo.

Baresi, Miaz e Lanzi (2009) compararam TestFul com o teste aleatório. As duas abordagens são aplicadas sobre VIM, editor de texto distribuído na maioria dos sistemas UNIX (VIM, 2011). Os autores afirmam que os resultados obtidos são encorajadores e afirmam que TestFul foi capaz de cobrir 100% dos ramos do método sob teste em duas horas, enquanto que a abordagem aleatória foi capaz de cobrir 35% dos métodos sob teste em 6 horas.

4.2.3 TDSGen/OO

Araki e Vergilio (2010) afirmam que muitas abordagens de geração de dados de teste utilizam critérios baseados na especificação (i.e., diagramas de estados, diagramas de classes, casos de uso), enquanto que outros critérios são baseados no programa (i.e., código fonte, grafo de fluxo de controle, fluxo de dados). Araki e Vergilio (2010) propõem um *framework* chamado TDSGen/OO (*Test Data Set Generator for Object-Oriented Software*) que tem como objetivo gerar dados de teste por meio de critérios estruturais de teste que considerem informações do código objeto Java (bytecode). O *framework* implementa um algoritmo genético e outros mecanismos, a fim de ampliar o funcionamento da ferramenta de teste TDSGen (FERREIRA; VERGILIO, 2005). TDSGen é uma ferramenta de teste procedimental que gera dados de teste para critérios estruturais e critérios baseados em defeito, para programas em C.

TDSGen/OO gera dados de teste a partir dos critérios estruturais fornecidos pela ferramenta de teste JaBUTi. A JaBUTi fornece critérios estruturais baseados em fluxo de controle e em fluxo de dados que consideram o bytecode. O *framework* TDSGen/OO é dividido em cinco módulos, sendo eles:

- Início: módulo responsável por receber a configuração inicial (arquivo de configuração preenchido pelo testador) e por gerenciar os demais módulos. A configuração inicial é dividida em duas seções, sendo: ferramenta de teste, seção que inclui parâmetros para a JaBUTi (i.e., nome do arquivo fonte que contém a classe sob teste, critério de teste escolhido); a estratégia de evolução, seção que parametriza o processo de evolução, informando a taxa de *crossover*, taxa de mutação, tamanho da população, número máximo de gerações, método de seleção, elitismo e ineditismo.

- Geração da População: módulo que utiliza as informações contidas na configuração inicial para geração da população inicial do algoritmo evolucionário. Este módulo também é responsável por codificar e decodificar cada indivíduo, garantindo que cada cromossomo seja bem formado, de acordo com a representação de Tonella, que foi a representação escolhida para abstração dos indivíduos.

- Ferramenta de Teste: a ferramenta de teste escolhida para execução dos testes e acompanhamento de sua execução foi a JaBUTi.

- Avaliação: este módulo avalia a cobertura de cada indivíduo da população usando o módulo de avaliação da JaBUTi – para essa avaliação cada indivíduo da população é convertido em um arquivo JUnit (JUNIT, 2012).

- Evolução: este módulo possui a habilidade de evoluir e aplicar os operadores genéticos sobre os indivíduos da população. Os operadores foram implementados conforme o trabalho de Tonella, dividindo-se em: mutação e *crossover*. O processo de evolução é encerrado quando o número máximo de gerações é atingido ou quando o elemento alvo é coberto. A aptidão de um indivíduo (*fitness*) é calculada pela da Equação (4.1):

$$fitness = \frac{número_de_elementos_cobertos}{número_de_elementos_requeridos}$$

Além de evoluir os indivíduos e calcular sua aptidão, este módulo também é o responsável por refinar a população, aplicando técnicas de elitismo e ineditismo. Elitismo é seleção dos melhores indivíduos através de sua aptidão, isto é, a escolha dos indivíduos com melhor taxa de aptidão. O ineditismo é a estratégia que objetiva a redução de indivíduos semelhantes na população.

Para os testes do *framework* foram escolhidos quatro programas bastante simples, para que fosse possível fornecer uma ideia inicial sobre o funcionamento da técnica. Os testes compararam 3 técnicas distintas: geração aleatória, geração baseada em algoritmo genético sem as técnicas de elitismo e ineditismo, e geração baseada na estratégia de

elitismo e ineditismo. Como resultado final, observou-se que a técnica baseada na estratégia de elitismo e ineditismo foi superior às demais técnicas em todos os testes entre os diferentes programas. Uma possível explicação para este fato é que possivelmente a geração baseada em elitismo e ineditismo reduz a população, o que diminui a quantidade de gerações, contribuindo para a melhora do desempenho e diminuição dos custos de execução do algoritmo genético.

4.2.4 **AutoTest/Eiffel**

Silva e Someren (2010) apresentam neste trabalho uma abordagem baseada na ferramenta de teste Autotest que combina a utilização de algoritmos genéticos com análise estática, para o teste automatizado de classes, na linguagem Eiffel, utilizando o número de defeitos encontrados como parâmetro otimizador. O principal objetivo deste trabalho, segundo os autores, é provar que o algoritmo evolucionário pode encontrar mais defeitos do que a geração aleatória em menos tempo. Silva e Someren afirmam que é plausível assumir que cada classe possui um conjunto ótimo de dados de entrada que encontrarão todos os defeitos. A partir dessa afirmação, acreditam que um algoritmo evolucionário pode então ser utilizado para envolver uma estratégia de teste que seja adaptável a cada classe. Este projeto melhora o estado da arte de duas formas: a primeira, ao utilizar o número de defeitos ao invés de medidas de cobertura do código como o principal parâmetro de otimização para o algoritmo evolucionário; e a segunda, ao combinar a análise estática com o algoritmo evolucionário.

Neste trabalho, Silva e Someren, utilizam a representação de Tonella (2004), de forma que a estratégia do teste é dada por um cromossomo que especifica quais métodos devem ser testados, e em qual ordem, além de descreverem quais seus parâmetros.

A análise estática utilizada procura, dentro da classe sob teste, a ocorrência de valores primitivos (i.e., inteiros, booleanos, caracteres), para que sejam aproveitados esses valores para direcionar a geração da população inicial usada pelo algoritmo evolucionário. Assim, com a utilização de tais valores objetiva-se construir uma população inicial com maior probabilidade de ser relevante, pois dentro de seus parâmetros primitivos, utiliza valores que possuem ocorrência dentro da classe e que podem ser mais relevantes.

Para a execução dos experimentos, a ferramenta Autotest foi utilizada como ferramenta de teste. Autotest fornece recursos para geração aleatória de dados de teste, selecionando um método e criando os objetos necessários para testá-lo. Para a execução da abordagem proposta foi necessária a extensão da ferramenta incluindo um módulo de

teste evolucionário.

Tabela 4.4 – *Classes sob teste. Fonte: Silva; Someren, 2010.*

Biblioteca	Nome da classe	Número de métodos
Lex	automaton	38
	dfa	42
	error_list	121
	fixed_automaton	99
	fixed_dfa	112
	fixed_integer_set	58
	high_builder	410
	text_filler	59
	lex_builder	365
	lexical	99
	linked_automaton	116
	metalex	420
	ndfa	56
	pdfa	311
	scanning	426
	state_of_dfa	101
Time	absolute	81
	time	408
	interval	365
	duration	70
	date_time_parser	379
	date_time_validity_checker	87

Vinte e duas classes foram aleatoriamente selecionadas dentro de duas bibliotecas muito utilizadas, *time* e *lex*, fornecidas pelo EiffelStudio 6.3 (EIFFELSTUDIO, 2010). A biblioteca *lex* fornece um mecanismo de análise léxica baseada em expressões regulares, enquanto que a biblioteca *time* fornece representação e cálculos computacionais para tipos de dados relacionados a data e tempo. A Tabela 4.4 apresenta as classes selecionadas e a quantidade de métodos existente em cada uma delas.

A comparação entre os algoritmos é feita entre o módulo de algoritmo evolutivo e o módulo de teste aleatório, ambos presentes na Autotest. O principal critério para a comparação é o número de defeitos encontrados dentro de um mesmo período de tempo.

Para determinar o número de defeitos encontrados por Autotest executando geração aleatória, foram executados três testes para 15, 30 e 60 minutos. As médias de defeitos encontrados para cada classe são apresentadas na Tabela 4.5 .

A utilização de análise estática para direcionar um conjunto inicial de valores primitivos é uma técnica que pode ser implementada na ferramenta Autotest. Por ter sido utilizada no algoritmo evolucionário, a estratégia de teste aleatório utilizada por Autotest foi estendida para usar a mesma análise estática para seleção de valores primitivos. Esta

Tabela 4.5 – Número de defeitos encontrados pelo algoritmo aleatório. Fonte: Silva; Someren, 2010.

ClassName	15min	30min	60min
automaton	9	9	9
dfa	17	17	17
error_list	12	13	14
fixed_automaton	10	10	11
fixed_dfa	24	28	29
fixed_integer	1	1	1
high_builder	19	24	26
absolute	1	1	1
time	5	6	6
interval	1	1	1
duration	0	0	0
date_time_parser	0	0	0
date_time_validity_checker	1	1	1
text_filler	12	13	14
lex_builder	16	18	22
lexical	24	26	26
linked_automaton	8	10	10
metalex	22	29	33
ndfa	8	9	9
pdfa	11	12	13
scanning	29	31	34
state_of_dfa	11	13	13
Sum	241	272	290

alteração acarretou em mais uma série de testes, avaliando os resultados obtidos com execuções de 15, 30 e 60 minutos. Os resultados obtidos são apresentados na Tabela 4.6.

Figura 4.1 – Comparação entre as três abordagens. Fonte: Silva; Someren, 2010.

Tabela 4.6 – Número de defeitos encontrados pelo algoritmo aleatório com análise estática. Fonte: Silva; Someren, 2010.

ClassName	15min	30min	60min
automaton	6	6	6
dfa	16	16	16
error_list	11	12	15
fixed_automaton	14	22	31
fixed_dfa	30	35	36
fixed_integer_set	2	2	2
high_builder	19	16	28
absolute	2	2	2
time	5	6	8
interval	3	3	3
duration	0	0	0
date_time_parser	1	1	3
date_time_validity_checker	3	3	3
text_filler	14	15	15
lex_builder	15	17	27
lexical	27	30	30
linked_automaton	8	8	8
metalex	27	30	33
ndfa	9	9	9
pdfa	11	15	19
scanning	24	30	40
state_of_dfa	17	20	25
Sum	264	298	359

O algoritmo evolucionário também é executado por 15, 30 e 60 minutos. Esses tempos incluem tanto a execução do algoritmo evolucionário como a análise estática. O número de defeitos descobertos é apresentado na Tabela 4.7. A comparação entre o algoritmo aleatório, o algoritmo aleatório com análise estática e o algoritmo evolucionário é ilustrada na Figura 4.1, e mostra que o algoritmo evolucionário possui a melhor performance.

Segundo os pesquisadores, não era esperado que o algoritmo evolucionário fosse mais eficaz do que o algoritmo aleatório no teste com duração de 15 minutos pois, para que a evolução resultasse em uma estratégia boa de teste, esperava-se que ela levasse um longo tempo. Os autores afirmam que a possível causa para esse resultado seja a evolução da população, que possivelmente tenha sido o fator chave para fortalecer a relevância da população.

4.2.5 Têmpera Simulada/Ada

O fundamento do trabalho de Tracey et al. (1998) é o uso de uma heurística de otimização global para construção de um *framework* genérico de geração de dados de teste. Segundo os autores, o domínio de entrada da maioria dos programas é bastante

Tabela 4.7 – Número de defeitos encontrados pelo algoritmo evolucionário. Fonte: Silva; Someren, 2010.

ClassName	15min	30min	60min
automaton	9	9	9
dfa	16	17	17
error_list	13	13	15
fixed_automaton	17	23	31
fixed_dfa	32	39	41
fixed_integer_set	2	2	2
high_builder	18	23	26
absolute	3	2	4
time	6	5	10
interval	3	5	5
duration	0	0	0
date_time_parser	0	5	5
date_time_validity_checker	3	3	3
text_filler	16	16	16
lex_builder	18	24	34
lexical	27	28	32
linked_automaton	7	8	11
metalex	25	31	43
ndfa	9	9	11
pdfa	12	14	23
scanning	27	34	41
state_of_dfa	16	24	29
Sum	279	334	408

amplo e técnicas de otimização global são desenhadas para encontrar boas aproximações para soluções ótimas em espaços de busca grandes e complexos. Para seus estudos os autores escolheram têmpera simulada como técnica de otimização para avaliar a eficácia de uma técnica baseada em otimização para o problema de geração de dados de teste. Têmpera simulada é uma técnica de busca global por vizinhança e, por isso, solucionando o problema de soluções ótimas locais (TRACEY et al., 1998).

Para aplicar a técnica de têmpera simulada é necessário construir um conjunto inicial de soluções candidatas e uma vizinhança para desenvolver uma função de custo que possa medir a qualidade dos candidatos a solução. A vizinhança deve representar o conjunto de soluções que de alguma forma chegue perto de uma solução candidata e pode ser definida conforme o conteúdo da Tabela 4.8.

Em seu experimento preliminar foi utilizada uma série de programas escritos na linguagem de programação Ada 95 que variam de tamanho entre 20 e 200 linhas de código. Cada uma das tentativas de geração de dados de teste foi repetida 50 vezes, e os resultados apresentados referem-se à média dos resultados obtidos após todos os testes. A Tabela 4.9 apresenta os resultados obtidos nos experimentos realizados pelos autores.

Tabela 4.8 – *Definição da vizinhança. Fonte: Tracey; Clark; Mander; McDermid, 1998*

Tipo de Dados	Vizinhança
Integer	Alguma proporção de um intervalo permitido
Float	Alguma proporção de um intervalo permitido
Boolean	verdadeiro ou falso
Enumerações	Qualquer valor da enumeração

Tabela 4.9 – *Resultado dos experimentos com têmpera simulada. Fonte: Tracey; Clark; Mander; McDermid, 1998*

Nome do Programa	Tamanho do conjunto de dados gerado	Cobertura	Tempo
Middle	7	100%	1,3s
Find	6	100%	2s
Remainder	5	100%	2,4s
Tomorrow	16	100%	4,8s
Triangle	14	100%	7,2s
Bigint Divide	29	99%	36,2s

Pode-se ver na Tabela 4.9 que foi possível cobrir 100% dos ramos em quase todos os programas, exceto em um. Para este caso, 48 das 50 tentativas atingiram 100% de cobertura, nas demais tentativas um dos ramos não pôde ser coberto. Tracey, Clark, Mander e McDermid discutem que a abordagem de geração de dados de teste baseada em otimização pode ser eficiente e efetiva. Entretanto, os softwares testados eram relativamente pequenos, e não se pode comprovar o comportamento do algoritmo com programas maiores.

4.3 Discussão

Após a revisão sistemática pôde-se ter conhecimento dos trabalhos que abordam a construção de *frameworks* geradores de dados de teste. Dos trabalhos mais relevantes notou-se que a representação de Tonella (2004) é uma técnica bastante empregada e tida como solução para a representação de dados de teste para software orientado a objetos. Os trabalhos apresentados fornecem mecanismos para a geração de dados de teste para programas OO e a maioria utiliza a representação definida por Tonella.

Não foram encontradas abordagens flexíveis que tornassem os *frameworks* extensíveis a ponto de permitir a composição de técnicas de geração de dados de teste. Entendendo-se como flexibilidade a capacidade do *framework* permitir sua extensão para a composição de técnicas de forma simples, integrada e consistente.

Tabela 4.10 – Características dos *frameworks* identificados como relevantes na literatura.

	Flexibilidade	Técnicas	Funções de Aptidão	Tonella
Evacom	Não	1	1	Sim
Testful	Não	1	1	Sim
TDSGen/OO	Não	1	2	Sim
AutoTest/Eiffel	Não	1	1	Sim
Têmpera Simulada/Ada	Não	1	1	Não

Cada uma das abordagens estudadas fornece uma única técnica de geração de dados de teste. Desta forma os *frameworks* tornam-se pouco versáteis com utilização limitada a uma única possibilidade de geração de dados de teste.

Os *frameworks* estudados não fornecem funções de aptidão parametrizáveis que pudessem ser escolhidas a fim de diversificar o processo de geração de dados de teste. Abrindo ressalvas para o trabalho de Araki e Vergilio (2010), que permite a utilização de funções multi-objetivo, a parametrização de valores para o processo de geração de dados de teste, mas que não permite a composição de técnicas de geração de dados de teste.

A Tabela 4.10 apresenta as características dos *frameworks* relevantes na literatura.

4.4 Considerações finais

Neste capítulo, foi apresentado um levantamento bibliográfico realizado a partir de uma revisão sistemática que consistiu da seleção de conteúdo por meio das bibliotecas digitais ACM, IEEE e Microsoft Research, além do uso de bibliotecas indexadas como Google Scholar e CiteSeerx, e da consulta de anais de eventos, como o SSBSE. Foram aplicados critérios de inclusão e exclusão sobre os trabalhos encontrados, além de critérios de seleção que avaliassem o funcionamento dos algoritmos.

A partir desse levantamento, pode-se observar que não existem *frameworks* que sejam flexíveis, isto é, permitam a utilização de diferentes técnicas de geração de dados de teste com diferentes funções de aptidão. As ferramentas identificadas baseiam-se em uma única técnica de geração, com um único tipo de função de aptidão. A maioria das técnicas identificadas como relevantes adotam a representação de Tonella para estruturar seus dados de teste; isso comprova a importância da representação. Assim pode-se notar a ausência de um *framework* gerador de dados de teste que auxilie na composição de técnicas, fornecendo um conjunto de recursos extensíveis e reutilizáveis.

No capítulo a seguir, será apresentado o *framework* JaBTeG, que provê diferentes

técnicas de geração de dados de teste, além de recursos extensíveis e reutilizáveis que contribuem com a composição de técnicas de geração de dados de teste.

Capítulo 5

Framework JaBTeG

Os *frameworks* disponíveis na literatura apresentam diferentes abordagens de geração de dados de teste. Dentre essas abordagens notam-se trabalhos que utilizam distintas técnicas de geração de dados de teste entre si, como: geração aleatória, execução simbólica, algoritmos de otimização, além de trabalhos que mesclam duas técnicas de teste e formam uma única técnica. Mas uma lacuna é vista entre todos esses trabalhos; cada *framework* estudado funciona com uma única técnica de geração, isto é, nenhuma das abordagens utiliza mais de uma técnica de geração de dados de teste, além de não fornecerem uma estrutura extensível (i.e., interfaces, classes abstratas) que possa ser utilizada como base para a composição de abordagens de geração de dados de teste.

Este trabalho apresenta o *framework* JaBTeG (*Java Bytecode Test Generation*) que fornece uma estrutura extensível para composição de técnicas de geração de dados de teste, além de recursos reutilizáveis para geração de dados de teste, com o objetivo de reduzir o esforço e o custo da composição de técnicas de geração de dados de teste e de ferramentas de teste de software orientado a objetos. Também é apresentado um estudo de caso descrevendo a incorporação do *framework* JaBTeG a uma ferramenta de teste.

5.1 Arquitetura do *framework*

Essa seção descreve a arquitetura do *framework* JaBTeG, detalhando seus componentes, sua estrutura extensível e seu funcionamento.

5.1.1 Componentes do *framework*

O *framework* desenvolvido funciona de maneira integrada com a ferramenta de teste JaBUTi. Para essa integração são previstos três componentes principais na solução, sendo eles:

- Bloco *Framework*: neste bloco existe toda a estrutura e funcionamento do *framework* JaBTeG, sem dependências diretas aos demais blocos de forma que seja fácil a sua

customização e extensão por projetos futuros.

- Bloco *Plug-In*: este bloco possui duas funcionalidades. A primeira funcionalidade é fornecer uma camada de apresentação (i.e., interface visual) para que os usuários testadores possam interagir com o *framework* de geração de dados de teste de maneira mais prática. A segunda funcionalidade é traduzir as interações do usuário com a interface visual para o *framework*.
- Bloco Ferramenta de Teste: este bloco representa a ferramenta de teste com todas as suas funcionalidades. Este bloco é o responsável por suportar o *plug-in* desenvolvido; adaptações foram necessárias para adequação da ferramenta de teste com o *plug-in*.

Figura 5.1 – Arquitetura de integração do framework.

A Figura 5.1 apresenta a integração entre os principais componentes da solução e seu ponto de comunicação com o usuário testador. Na Figura 5.1, o usuário testador envia sua requisição para o *framework* por meio da interface visual do *plug-in*, esta requisição contém a classe e o método de teste que devem ser avaliados e o cenário de geração de dados de teste, especificando a técnica de geração de dados de teste, a função de aptidão e o critério de seleção, além de outros parâmetros complementares.

O *plug-in* é apresentado ao usuário testador por meio da ferramenta de teste, que por sua vez fornece recursos para auxiliar na geração de dados de teste ao permitir a compilação e identificação da cobertura dos indivíduos de teste. Cada indivíduo de teste corresponde a um método de um teste de unidade. Este método é compilado e executado pela ferramenta de teste que identifica quais os requisitos de teste cobertos pelo indivíduo de teste.

Ao fim da geração de dados de teste o *framework* apresenta ao testador os indivíduos de teste que foram capazes de cobrir diferentes requisitos de teste.

Figura 5.2 – Diagrama de atividades do processo de geração de dados de teste do framework JaBTeG.

A Figura 5.2 apresenta o diagrama de atividades do processo de geração de dados de teste. Neste diagrama pode-se observar a ferramenta de teste carregando recursos de integração do *plug-in*, tais como interface gráfica e bibliotecas de comunicação com a ferramenta de teste. O *plug-in* apresenta a interface gráfica de integração com o *framework* e recebe a solicitação de geração de dados de teste feita pelo testador, em seguida transforma a solicitação em parâmetros para o *framework* de geração de dados de teste e o executa. O *framework* identifica qual a técnica de geração de dados de teste escolhida, executa o algoritmo, orquestrando chamadas à ferramenta de teste na espera de dados sobre a cobertura dos indivíduos, e retorna para o *plug-in* os dados de teste gerados. O

plug-in, ao receber os dados de retorno do *framework*, apresenta-os ao usuário testador e aguarda novas interações.

O *framework* JaBTeG divide seu funcionamento em seis módulos:

Leitor de *Bytecode*. Módulo de leitura de informações do *bytecode* Java. Neste módulo são identificadas características da classe sob teste e de suas dependências, tais como: visibilidade, construtores públicos, métodos públicos e tipos de dados de seus parâmetros (i.e., complexos ou primitivos). Estas informações poderiam ser obtidas pela ferramenta JaBUTi, mas com o intuito de reduzir as dependências para trabalhos futuros que queiram utilizar outra ferramenta de teste, este módulo foi criado.

Fábrica de Indivíduos. Este módulo do *framework* é responsável por fornecer uma estrutura, baseada na proposta de Tonella (2004), para representação dos resultados gerados pelas técnicas de geração de dados de teste. Neste módulo são encapsuladas funções de geração de valores aleatórios para tipos de dados primitivos e complexos, dado um tipo de dados (i.e., objeto ou tipo primitivo). O domínio dos valores gerados aleatoriamente segue os conceitos discutidos na Seção 5.1.5 apresentada a seguir. A estrutura de Tonella foi escolhida como base para representação dos resultados, pois graças ao seu modelo estrutural permite fácil adaptação e composição das diferentes técnicas de geração de dados de teste.

Biblioteca de Algoritmos. Este módulo fornece implementação das distintas técnicas de geração automática de dados de teste. Os demais módulos funcionam em torno deste, sendo executados conforme o funcionamento das técnicas.

Gerador de Testes de Unidade. Módulo de construção e execução de testes de unidade. Para que a aptidão de um indivíduo seja calculada é preciso que sua cobertura seja identificada. Para isso, é criado um arquivo no formato JUnit no qual cada indivíduo de teste corresponde a um método de teste de unidade no formato JUnit. Após criado o arquivo, ele é então compilado e executado pela ferramenta de teste, retornando ao *framework* a respectiva cobertura de cada indivíduo de teste, para assim calcular a aptidão dos indivíduos. Este módulo é responsável por criar os arquivos no formato JUnit com base nos indivíduos da população e enviá-los para a ferramenta de teste.

Biblioteca de Aptidões. Este módulo é um dos mais importantes do *framework*, pois sua existência permite a utilização de diferentes funções de aptidão nos critérios

suportados pelo *framework*. Este módulo fornece uma estrutura extensível, a fim de permitir a extensão das funções de aptidão, para que técnicas de geração de dados de teste possam ser construídas a partir da mudança do cálculo da aptidão, assim como nas abordagens multi-objetivo.

Biblioteca de Seleções. Este módulo permite o uso de diferentes critérios de seleção de indivíduos para construção de novas populações. Após o estudo da literatura sobre geração de dados de teste notou-se o uso de diferentes critérios para seleção dos melhores indivíduos nas técnicas metaheurísticas; desta forma foi criada uma estrutura extensível que permite a inclusão de diferentes técnicas de seleção de indivíduos.

A Figura 5.3 apresenta as dependências entre os módulos por meio de um diagrama de pacotes. A ordem das chamadas aos módulos do *framework* será determinada pela lógica contida nos algoritmos de geração de dados de teste. O módulo Biblioteca de Algoritmos é o módulo central do *framework* e tem a responsabilidade de orquestrar a geração de dados de teste e o funcionamento dos demais módulos.

Figura 5.3 – Módulos do framework JaBTeG.

5.1.2 Estruturas extensíveis

Após a análise de diferentes técnicas de geração de dados de teste foi identificado um conjunto de ações que se repetem nas técnicas analisadas. Este conjunto de ações comuns nas técnicas analisadas foi abstraído e utilizado como base para criação de estruturas padrões, que pudessem ser estendidas para composição de técnicas de geração de

dados de teste. Para construção dessas estruturas padrões foram seguidas as experiências relacionadas à detecção de padrões documentadas por Gamma et al. (1995).

A Figura 5.4 detalha o conjunto de classes e interfaces construído no *framework* JaB-TeG que provê recursos para criação de diferentes técnicas de geração de dados de teste. A seguir são detalhados o funcionamento e a motivação para existência de cada classe e interface do conjunto construído.

Figura 5.4 – Estrutura extensível provida pelo framework JaBTeG.

Interface *IFitness*

A interface *IFitness* permite a criação de novas funções de aptidão. Com essa interface é possível construir diferentes funções de aptidão por meio de um conjunto de membros (o termo “membro” é utilizado para representar os métodos existentes nas interfaces que, de forma contrária aos métodos das classes, não possuem uma implementação e funcionam apenas como um modelo de assinatura a ser seguido) que permitem o cálculo da aptidão. O membro “*computeFitness(GenerationRequirement, GenerationCoverage)*” define que é preciso utilizar como parâmetro um requisito de teste e a cobertura de um indivíduo,

para assim calcular sua aptidão. O Apêndice D apresenta a implementação das funções de aptidão construídas para validação do *framework* e que são melhor detalhadas no próximo capítulo.

Interface *ISelection*

Algoritmos metaheurísticos que exigem a execução de rotinas de mutação e *crossover* precisam de uma rotina de seleção dos melhores indivíduos para a evolução da população. Essas rotinas de seleção de indivíduos de teste podem conter diferentes comportamentos entre as técnicas de geração de dados de teste. Desta forma, é preciso prover suporte para a criação de diferentes técnicas de seleção de indivíduos, de modo que uma técnica de geração de dados de teste possa gerar resultados diferentes a partir do uso de diferentes critérios de seleção de indivíduos. A interface *ISelection* provê recursos para criação de diferentes algoritmos de seleção de indivíduos por meio de um único membro que, dado um conjunto de indivíduos, execute uma lógica para selecionar os melhores indivíduos para evolução da população.

Interfaces *IStrategySelection* e *IStrategyFitness*

Essas interfaces devem ser implementadas por técnicas de geração de dados de teste que exijam o uso de uma função de aptidão ou técnica de seleção de indivíduos. Quando uma técnica de geração de dados de teste exigir o uso de uma função de aptidão é preciso que essa técnica de geração de dados de teste implemente a interface *IStrategyFitness*, que fornece um membro para associação de uma função de aptidão (classe que implemente a interface *IFitness*) com a técnica de geração de dados de teste implementada. Dessa mesma forma, quando for criada uma técnica que exija a associação de uma técnica de seleção, é preciso que esta técnica de geração de dados de teste implemente a interface *IStrategySelection*, que fornece um membro para associação de uma função de seleção (classe que implemente a interface *ISelection*) com a técnica de geração de dados de teste.

Interface *ITestDataGenerationTool*

A interface *ITestDataGenerationTool* é necessária para comunicação com a ferramenta de teste. A interface *ITestDataGenerationTool* contém membros que apoiam:

Compilação de indivíduos: dado uma população de indivíduos a ferramenta de teste deve executar uma rotina de compilação destes indivíduos.

Identificação de requisitos de teste: dado um critério de teste a ferramenta de teste deve retornar um conjunto de requisitos de teste a serem cobertos.

Identificação dos requisitos cobertos por cada indivíduo: dado um critério de teste a ferramenta de teste deve, após ter compilado os indivíduos, identificar qual a cobertura de cada um dos indivíduos de teste.

Classes *BaseGenerationStrategy* e *MetaheuristicBaseGenerationStrategy*

As classes *BaseGenerationStrategy* e *MetaheuristicBaseGenerationStrategy* são o ponto central do *framework* JaBTeG. Essas classes fazem toda a orquestração do processo de geração de dados de teste, além de fornecerem uma estrutura reutilizável para composição de técnicas de geração de dados de teste. O Apêndice A apresenta a implementação dessas duas classes.

Para torná-las de fácil extensão, a estrutura das classes *BaseGenerationStrategy* e *MetaheuristicBaseGenerationStrategy* foi construída com base em um *design pattern* chamado *Template Method* (GAMMA et al., 1995). O *design pattern Template Method* tem como objetivo definir o esqueleto de um algoritmo em uma operação, delegando o comportamento de alguns passos para as suas subclasses; desta forma as subclasses têm liberdade de modificar o comportamento desses passos sem afetar a estrutura principal do algoritmo. Este *design pattern* deve ser utilizado quando for necessário implementar as partes invariantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar, assim pode-se definir um método-template que invoca as operações variantes em pontos específicos, permitindo extensões somente nestes pontos (GAMMA et al., 1995).

Figura 5.5 – Estrutura do *design pattern Template Method*. Fonte: GAMMA et al., 2000.

A Figura 5.5 apresenta um diagrama de classes com a estrutura do *design pattern Template Method*. Pode-se notar que existe uma classe abstrata chamada “*ClasseAbs-*

trata” que expõe um método chamado “*metodoTemplate()*”, que internamente executa chamadas a dois métodos abstratos, “*operacaoPrimitiva01()*” e “*operacaoPrimitiva02()*”. Tais métodos primitivos têm sua implementação construída nas subclasses de “*ClasseAbstrata*”, no caso a classe “*ClasseConcreta*”. Desta maneira, observa-se que a classe “*ClasseAbstrata*” garante uma sequência lógica de execução ao fornecer um método template, além de fornecer liberdade na construção do comportamento de seus métodos abstratos implementados em suas subclasses.

O *design pattern Template Method* foi utilizado para a construção da estrutura base das classes *BaseGenerationStrategy* e *MetaheuristicBaseGenerationStrategy* justamente por conta da necessidade de fornecer uma sequência lógica de execução invariante às técnicas de geração de dados de teste criadas, e que também permitisse a extensão de pontos específicos de acordo com cada técnica.

BaseGenerationStrategy é uma classe abstrata que deve ser utilizada como base para criação de técnicas de geração de dados de teste que não contenham: rotinas de mutação de indivíduos, *crossover* de indivíduos e características metaheurísticas. Sua estrutura pode ser vista na Figura 5.6.

Figura 5.6 – Estrutura da classe *BaseGenerationStrategy*.

A classe *BaseGenerationStrategy* expõe um método chamado “*generateData()*”, que encapsula toda a orquestração da geração de dados de teste; pode-se visualizar na nota do diagrama de classes da Figura 5.6 um resumo de seu comportamento. Além do método “*generateData()*”, outros métodos compõem a classe *BaseGenerationStrategy* e detalham o funcionamento de “*generateData()*”, são eles:

- *initializeGeneration():void* : método abstrato, utilizado para inicializar os atributos existentes na técnica de geração de dados de teste. Técnicas de geração de dados de teste que executam alguma pré-análise dos recursos a serem testados podem utilizar este método como o local para o encapsulamento de sua lógica.
- *generateInitialPopulation():Individual[]* : método abstrato, possui a responsabilidade de gerar a população inicial de indivíduos da geração de dados de teste.
- *executeIndividuals():void* : método responsável por capturar a população atual de indivíduos, enviar para a ferramenta de teste e aguardar a compilação dos indivíduos.
- *identifyRequirements():GenerationRequirement[]* : identifica os requisitos de teste que devem ser cobertos pela técnica de geração de dados de teste.
- *updateIndividuals():void* : atualiza a população de indivíduos de teste com suas respectivas coberturas, identificadas pelo método “*executeIndividuals()*”.
- *verifyIfRequirementHasBeenCovered(GenerationRequirement):boolean* : verifica se o requisito de teste passado como parâmetro foi coberto por algum dos indivíduos da população atual.
- *finalizeGeneration():void* : método abstrato, utilizado para finalizar os atributos existentes na técnica de geração de dados de teste.
- *initialize(String, int, String, String, EnumCriterion):void* : método utilizado para inicializar a técnica de geração de dados de teste; os parâmetros seguem a seguinte ordem: caminho do arquivo *bytecode* Java; tamanho inicial da população; nome da classe sob teste; assinatura do método sob teste; e o critério de teste que será utilizado.

A classe abstrata *MetaheuristicBaseGenerationStrategy* é a estrutura para criação de técnicas de geração de dados de teste que contém características metaheurísticas. Essa classe herda da classe *BaseGenerationStrategy*, reaproveitando todos os comportamentos

construídos para: execução de indivíduos, identificação de requisitos de teste, atualização de indivíduos com suas respectivas coberturas e identificação de requisitos cobertos. Além disso, a classe *MetaheuristicBaseGenerationStrategy* implementa as interfaces *IStrategySelection* e *IStrategyFitness*, suportando a parametrização de uma estratégia de seleção de indivíduos e o uso de funções de aptidão.

Figura 5.7 – Estrutura da classe *MetaheuristicBaseGenerationStrategy*.

A Figura 5.7 apresenta a estrutura da classe *MetaheuristicBaseGenerationStrategy*. A classe *MetaheuristicBaseGenerationStrategy* sobrescreve o método “*generateData()*” da classe *BaseGenerationStrategy*, modificando seu comportamento para uma lógica própria dos algoritmos metaheurísticos. Além dos métodos existentes na classe *BaseGenerationStrategy*, a classe *MetaheuristicBaseGenerationStrategy* possui um conjunto de métodos próprio para fornecer recursos para o uso das funções de aptidão, execução de mutação e *crossover*, além de uma lógica interna própria para o refinamento da população de indivíduos ao longo de sua execução. A listagem de métodos a seguir complementa os membros expostos pela classe *MetaheuristicBaseGenerationStrategy* e complementa o entendimento da Figura 5.7:

- *computeFitness(Individual[], GenerationRequirement):double* : dada uma população

de indivíduos e um requisito de teste, o método “*computeFitness(Individual[], GenerationRequirement)*” irá calcular a aptidão de cada um dos indivíduos da população em relação ao requisito de teste.

- *executeMutation(Individual[]):Individual[]* : este método aciona as rotinas de mutação e *crossover* existentes no *framework JaBTeG*. As rotinas de mutação e *crossover* são as propostas por Tonella (2004) e já discutidas anteriormente.

Após o estudo de diferentes técnicas metaheurísticas (BERTSIMAS; TSITSIKLIS, 1993; TONELLA, 2004; BARROS; TEDESCO, 2008; PINHEIRO, 2010) notou-se que utilizam a geração aleatória de indivíduos como passo comum de sua execução. Desta forma, a classe *MetaheuristicBaseGenerationStrategy* sobrescreve o método “*generateInitialPopulation()*”, implementando uma rotina de geração aleatória de indivíduos.

Interface *IGUIGenerationStrategy*

A principal motivação para construção da interface *IGUIGenerationStrategy* esta associada com a construção da interface visual da ferramenta de teste. A interface *IGUIGenerationStrategy* fornece recursos para que seja possível consultar quais critérios de teste, funções de aptidão, técnicas de seleção e parâmetros personalizados são suportados pela técnica de geração de dados de teste. Esta interface funciona como guia para composição das informações apresentadas pela interface visual da ferramenta de teste que se comunica com o *framework*. Um exemplo de construção de interface visual é apresentado no Capítulo 6.

A interface *IGUIGenerationStrategy* possui quatro membros que indicam os recursos suportados pela técnica de geração de dados de teste. O método “*getSupportedFitnesses()*” fica responsável por informar quais as funções de aptidão suportadas pela técnica de geração de dados, retornando um conjunto dinâmico de classes que implementem a interface *IFitness*; o método “*getSupportedSelections()*” retorna um conjunto de classes que implementam a interface *ISelection*, indicando quais as técnicas de seleção suportadas pela técnica de geração de dados de teste; o membro “*getSupportedCriteriaes()*” retorna um conjunto de enumerações que indicam quais os critérios de teste suportados pela técnica de teste; e, por fim, o método “*getCustomParameters()*” indica quais parâmetros personalizados da técnica de geração de dados de teste devem ser suportados pela interface visual. O método “*getCustomParameters()*” retorna um conjunto de instâncias da classe *GUIGenerationCustomParameter*; essa classe fornece informações relevantes para criação dinâmica dos parâmetros personalizados na interface visual dos *plug-ins*, tais informações

descrevem: o valor padrão do parâmetro personalizado; o tipo primitivo do parâmetro; a descrição do parâmetro; o valor atual do parâmetro; e o método *setter* que deve ser invocado via *reflection* para atribuir o valor informado pelo testador via interface visual na técnica de geração de dados de teste.

5.1.3 Análise do código do programa

Para gerar dados de teste para um método é preciso fazer a leitura das assinaturas dos métodos e das dependências da classe sob teste. Os demais métodos são importantes, pois eles serão utilizados para gerar diferentes estados do objeto sob teste, e os tipos dependentes são importantes para que seja possível saber de quais tipos primitivos ou objetos deve-se criar instâncias durante o processo de geração de dados de teste. Esse processo de identificação de métodos e tipos de dados dependentes é recursivo, pois sempre que identificada a dependência a um tipo de dados e este tipo de dados é um objeto, então, é preciso identificar seus métodos e suas dependências a outros tipos de dados. Desta forma, ganha-se desempenho durante a geração aleatória de dados, pois já se tem todos os tipos de dados dependentes mapeados e salvos em memória, não exigindo a releitura do *bytecode*, agregando desempenho à geração de dados de teste.

5.1.4 Fabricação de indivíduos

A fabricação de indivíduos é dividida em dois passos, sendo eles: criação aleatória de um indivíduo de teste e geração aleatória de valores.

Segundo Tonella (2004), um indivíduo de teste corresponde a uma sequência de um construtor e invocações de métodos, seguidos dos valores de seus parâmetros. A criação aleatória de um indivíduo parte da criação de uma instância da classe sob teste, por meio da seleção aleatória de um de seus construtores. Dado um conjunto de construtores identificados durante a leitura do *bytecode*, um construtor é selecionado aleatoriamente e atribuído ao objeto. Após este passo é preciso selecionar métodos intermediários que estimulem um estado para o indivíduo de teste. A escolha de métodos intermediários é feita de forma parecida com a seleção de construtores. Dado um conjunto de métodos identificados durante a leitura do *bytecode*, é escolhida uma quantidade aleatória N de métodos intermediários que o indivíduo deve conter. Então são selecionados N métodos aleatoriamente e adicionados ao indivíduo sob avaliação. E como parte final da construção da estrutura, é adicionado ao indivíduo o método sob teste.

Sempre que um construtor ou método é adicionado à estrutura do indivíduo é feita a leitura dos parâmetros esperados por estes elementos. Tal leitura procura identificar o tipo de dados e a multiplicidade do parâmetro. Caso o tipo de dados do parâmetro seja um objeto, então o mesmo processo de seleção de construtor e de métodos intermediários é executado para o tipo de dados do parâmetro lido. Este passo é importante para que o parâmetro que depende de um objeto também tenha uma instância construída a partir de um construtor válido e seu estado construído por meio da execução de métodos intermediários.

O segundo passo da geração aleatória de indivíduos corresponde à geração aleatória de valores. Depois de construído o indivíduo de teste, sua estrutura (construtor e chamadas a métodos) já é conhecida, além de suas dependências a outros objetos. Desta forma, a geração aleatória tem apenas de percorrer a estrutura do indivíduo (e de suas respectivas dependências) identificando parâmetros primitivos, e gerando valores aleatórios para eles. A geração aleatória de valores primitivos é mais simples se comparada à geração aleatória de objetos (discutida no passo anterior), pois os tipos primitivos pertencem a um grupo limitado de tipos, sendo esses tipos: boolean, char, short, byte, int, long, float e double.

A geração aleatória de instâncias de objetos é mais complexa, pois existem desafios como: geração de valores para tipos de dados dependentes, dependência a outros objetos, criação de instâncias de classes abstratas e recursividade durante a geração de tipos aleatórios. A geração de valores para tipos de dados dependentes corresponde à geração aleatória de instância de objetos para dependências da classe sob teste. Essa geração é trabalhosa e pode acarretar na seleção recursiva de tipos de dados, que ocorre quando um tipo de dados depende de uma instância de objeto do mesmo tipo da sua, fazendo com que a técnica gere recursivamente inúmeras instâncias de objeto do mesmo tipo, caindo em um laço infinito.

Este problema, assim como a dependência entre as classes, pode acarretar um alto grau de profundidade, isto é, acaba-se criando uma árvore de dependência de objetos muito grande, tornando o indivíduo de teste muito complexo. Para solucionar essa questão foi utilizado um limiar de profundidade, que quando ultrapassado limita a criação de novas instâncias de objetos. Esse controle de limitações é feito durante a seleção de métodos intermediários dos objetos, no qual são filtrados apenas os métodos que dependem apenas de tipos primitivos. Desta maneira, limita-se a geração de objetos, já que novos métodos intermediários que dependem apenas de tipos primitivos não irão exigir a criação de novas instâncias de objetos.

5.1.5 Geração dirigida de valores aleatórios

O *framework* JaBTeG permite a passagem de parâmetros que direcionem a geração aleatória de valores dos tipos primitivos. Tais parâmetros modificam o domínio de geração dos indivíduos e direcionam o domínio de valores a um intervalo específico de valores. Inicialmente, sem a passagem de parâmetros, a geração aleatória de valores segue o mesmo domínio proposto por Tonella (2004), conforme Tabela 5.1.

Tabela 5.1 – Domínio padrão de valores para geração aleatória de indivíduos.

Tipo de Dados	Domínio
Inteiros e números reais	De 0 a 100
Booleanos	Valores booleanos <i>verdadeiro</i> e <i>falso</i>
Strings	Caracteres alfanuméricos (<i>[a-zA-Z0-9]</i>)

O uso da geração aleatória dirigida é indicado em cenários nos quais se tem conhecimento dos possíveis domínios de entrada esperados pelo método sob teste. Desse modo, a geração de valores torna-se mais precisa, pois define um domínio de possíveis valores, tornando a geração aleatória de valores menos abrangente e mais precisa.

Um exemplo relevante para geração dirigida de valores aleatórios é a necessidade de testar um método que receba como parâmetro um caminho para um arquivo. A geração aleatória provê apenas valores aleatórios, cujo conteúdo não faz parte do esperado. Neste caso, a geração aleatória não fornece valores correspondentes ao domínio esperado; desta maneira é indicada a utilização da geração aleatória dirigida, fornecendo valores que façam parte do domínio esperado. No caso do exemplo, um conjunto de caminhos válidos para arquivos.

5.1.6 Geração de valores para vetores e matrizes

A geração de valores para vetores e matrizes é dividida em dois tipos: vetores para tipos primitivos e vetores para objetos. A geração de dados para vetores segue a mesma lógica para tipos primitivos ou objetos. A geração dos valores aleatórios para vetores inicia-se pela identificação do tipo de dados base do vetor. Após essa identificação, um valor inteiro aleatório é gerado para definir o tamanho do vetor a ser gerado. Com isso, criam-se valores para preencher o vetor seguindo as mesmas regras propostas para geração de parâmetros primitivos ou objetos.

5.1.7 Formatos para exportação dos dados gerados

Para exportação dos dados gerados pode-se escolher entre dois formatos de saída, pela representação de indivíduos de teste proposta por Tonella ou por meio da geração de classes de teste no formato JUnit. O resultado destas saídas é apresentado no Capítulo 6.

5.1.8 Critérios de teste suportados

Os critérios de teste suportados são baseados nos critérios de teste suportados pela ferramenta JaBUTi. A ferramenta de teste JaBUTi provê suporte a oito critérios de teste, sendo eles: todos-nós independentes de exceção, todos-ramos independentes de exceção, todos-usos independentes de exceção, todos-potenciais-usos independentes de exceção, todos-nós dependentes de exceção, todos-ramos dependentes de exceção, todos-usos dependentes de exceção e todos-potenciais-usos dependentes de exceção.

5.1.9 Limitações do *framework* JaBTeG

Atualmente o *framework* JaBTeG possui duas limitações em seu funcionamento:

- Parâmetros personalizados da interface visual estão limitados a tipos primitivos: a criação de parâmetros personalizados, detalhados na interface *IGUIGenerationStrategy*, estão limitados a tipos primitivos, não é possível a criação de parâmetros personalizados que correspondam a tipos complexos (i.e., objetos). Almeja-se em trabalhos futuros o estudo de modelos de serialização de estruturas complexas para o suporte a parâmetros de tipos complexos.
- Geração de valores apenas para classes concretas: a geração de valores é limitada apenas para classes concretas; não é possível a geração de dados de teste para interfaces e classes abstratas. Pretende-se em trabalhos futuros adicionar esta funcionalidade à fábrica de indivíduos de teste do *framework*.

5.1.10 Considerações finais

Neste capítulo foi apresentada a arquitetura do *framework* JaBTeG. O *framework* JaBTeG é um arcabouço que facilita a composição de técnicas de geração de dados de teste, funções de aptidão e algoritmos de seleção, além de fornecer recursos complementares para geração de dados de teste, como: fabricação de indivíduos de teste, geração dirigida de

valores aleatórios e geração de valores para tipos primitivos, objetos e vetores. A extensão do *framework* JaBTeG é possível devido a existência de uma estrutura extensível criada a partir da análise de técnicas de geração de dados de teste existentes na literatura.

No próximo capítulo são apresentados os experimentos realizados para validação do *framework* JaBTeG.

Capítulo 6

Aplicações do Framework JaBTeG

Depois de criada a arquitetura do *framework*, foi criada uma biblioteca de geração de dados de teste com base nos recursos providos pelo *framework* JaBTeG para sua validação. A ferramenta de teste JaBuTi foi escolhida para auxiliar a geração de dados de teste com o *framework* JaBTeG. As seções seguintes apresentam: o processo de composição de técnicas de geração de dados de teste, funções de aptidão e algoritmos de seleção; o passo-a-passo para instalação de novos *plug-ins* de geração de dados de teste na ferramenta JaBuTi; e os experimentos realizados para geração de dados de teste.

6.1 Composição de técnicas de geração de dados de teste

Nesta seção será apresentada a implementação de técnicas de geração de dados de teste utilizando o *framework* JaBTeG. As técnicas de geração de dados apresentadas são: aleatória, evolucionária, subida de encosta e têmpera simulada. Algumas dessas técnicas exigem a construção de funções de aptidão, critérios de seleção ou uso de parâmetros personalizados. O modo como estes recursos foram implementados também é registrado ao longo desta seção.

6.1.1 Algoritmo Aleatório

A implementação da técnica de geração de dados de teste aleatória com o *framework* JaBTeG foi trivial. Para sua construção foi preciso criar uma classe que herdasse da classe *BaseGenerationStrategy* e implementasse os métodos abstratos requeridos pela estrutura. No total foi preciso implementar a estrutura de três métodos abstratos da classe *BaseGenerationStrategy*:

- *initializeGeneration():void* : método utilizado quando for preciso executar uma rotina de pré-análise. A técnica de geração de dados de teste aleatórios não exige uma rotina deste tipo, então não foi preciso a adição de código neste método.

- *generateInitialPopulation():Individual[]*: este método é responsável por criar a população inicial de indivíduos para geração de dados de teste. A implementação dessa geração utilizou recursos nativos do *framework* JaBTeG, executando a leitura do *bytecode* Java e em seguida gerando os indivíduos aleatórios a partir das estruturas colhidas na leitura do *bytecode*.
- *finalizeGeneration():void*: método utilizado quando for preciso executar uma rotina de pós-análise. A técnica de geração de dados de teste aleatórios não precisa de uma rotina deste tipo, então não foi preciso a adição de código neste método.

A Figura 6.1 apresenta a estrutura de classes implementada, detalhando as interfaces implementadas e a herança com *BaseGenerationStrategy*. A implementação do Algoritmo Aleatório está descrita no Apêndice B.

Figura 6.1 – Algoritmo aleatório desenvolvido com o *framework* JaBTeG.

6.1.2 Algoritmo Evolucionário

A estrutura do *framework* JaBTeG que fornece um conjunto de recursos mais apropriados para implementação de uma técnica de geração de dados de teste baseada em algoritmos evolucionários é a da classe *MetaheuristicBaseGenerationStrategy*. Essa classe, apresentada no diagrama da Figura 6.2, fornece recursos para mutação de indivíduos, *crossover*, evolução de populações, uso de critérios de seleção e o uso de funções de aptidão. A implementação do Algoritmo Evolucionário está descrita no Apêndice C.

A implementação dessa técnica de geração de dados de teste exigiu a composição de funções de aptidão e técnicas de seleção. Para o desenvolvimento das funções de aptidão foi preciso construir classes que implementassem a interface *IFitness*. Foram criadas duas funções de aptidão (conforme Figura 6.3), sendo:

Figura 6.2 – Algoritmo Evolucionário desenvolvido com o framework JaBTeG.

- Função de Similaridade: a aptidão de um indivíduo é calculada a partir da divisão da quantidade de requisitos requeridos cobertos pela quantidade de requisitos requeridos, dado um caminho de execução (ARAKI, 2009).
- Função de Ineditismo: função de aptidão que ao calcular a aptidão de um indivíduo (baseada em sua similaridade) atribui um bônus de ineditismo, se esse indivíduo cobrir um elemento requerido não coberto por outros indivíduos da população (ARAKI, 2009).

As implementações destas funções de aptidão estão descritas no Apêndice D.

Figura 6.3 – Funções de aptidão criadas com o framework JaBTeG.

Para construção dos algoritmos de seleção foi preciso que as classes implementassem a interface *ISelection*. A Figura 6.4 apresenta as técnicas de seleção desenvolvidas, sendo:

- Elitismo: critério de seleção que seleciona os indivíduos com base em uma lista ordenada pelo valor de sua aptidão, assim os indivíduos com as melhores aptidões ficam garantidos na nova população (ARAKI, 2009).

- Torneio: seleciona indivíduos por meio de competições diretas entre si, usando como arma o valor de sua aptidão. Os indivíduos são selecionados de maneira aleatória, sem que exista qualquer tipo de favorecimento (LINDEN, 2012). Uma comparação entre os indivíduos $x1$, $x2$ e $x3$, com respectivas aptidões 0.9, 0.2 e 0.7, resulta na vitória do indivíduo $x1$, pois ele possui uma aptidão melhor.
- Roleta: a seleção por roleta posiciona os indivíduos em uma roleta, atribuindo a cada um uma “fatia” proporcional a sua aptidão. Depois deste posicionamento, gera-se um valor aleatório que funciona como uma agulha. O indivíduo que tiver sua fatia indicada pela agulha da roleta é então selecionado (ARAKI, 2009; MACHADO, 2012).
- Média: seleciona os indivíduos que possuem aptidão acima da média da população.

As implementações destes algoritmos de seleção estão descritas no Apêndice E.

Figura 6.4 – Técnicas de seleção criadas criadas com o framework JaBTeG.

6.1.3 Subida de Encosta

Para implementação do algoritmo de Subida de Encosta foi necessário a criação uma classe que herdasse de *MetaheuristicBaseGenerationStrategy*, devido a sua origem metaheurística. A implementação desta técnica de geração de dados de teste, ao contrário das técnicas de geração aleatória e evolucionária, exigiu que o método “*generateData()*” fosse sobrescrito, para que o seu funcionamento fosse customizado.

O funcionamento do algoritmo de Subida de Encosta é parecido com o funcionamento do algoritmo evolucionário, exceto pela execução de uma regra adicional acionada quando um requisito de teste não é coberto pela população. Esta regra adicional corresponde ao modo de funcionamento do algoritmo de Subida de Encosta, que executa pequenas

alterações nos indivíduos da população, melhorando-a pouco a pouco. A Figura 6.5 mostra a estrutura de sua implementação e o funcionamento do método “*generateData()*”.

Figura 6.5 – Algoritmo de Subida de Encosta desenvolvido com o framework JaBTeG.

O algoritmo de Subida de Encosta também exige o uso de uma função de aptidão e uma técnica de seleção. Para tanto foram reaproveitadas as funções e técnicas criadas para o algoritmo evolucionário. Como as funções de aptidão e as técnicas de seleção utilizam uma interface padrão, e a técnica de geração de dados de teste herda de uma estrutura base, existe compatibilidade entre todos os recursos criados a partir da estrutura extensível do *framework*. Essa compatibilidade entre os recursos permite que diferentes componentes comuniquem-se, reduzindo o custo de implementação e aumentando o reuso de código. A implementação do algoritmo de Subida de Encosta está descrita no Apêndice F.

6.1.4 Têmpera Simulada

Assim como no algoritmo de Subida de Encosta, o algoritmo Têmpera Simulada também utiliza a estrutura provida por *MetaheuristicBaseGenerationStrategy* e sobreescreve do método “*generateData()*”, customizando-o conforme a lógica discutida na Seção 3.1.3.

Assim como na técnica de Subida de Encosta, Têmpera Simulada também faz reuso das funções de aptidão e das técnicas de seleção desenvolvidas para o algoritmo evolu-

cionário.

O funcionamento customizado para o algoritmo Têmpera Simulada segue a mesma estrutura do algoritmo de Subida de Encosta, mas difere ao conseguir escapar dos máximos locais.

Este algoritmo também difere dos demais ao exigir a parametrização de uma variável de “energia”, que define um limite de execuções para o algoritmo, funcionando como ponto de parada para sua execução. A Figura 6.6 apresenta a estrutura criada para o algoritmo de Têmpera Simulada e o Apêndice G detalha sua implementação.

Figura 6.6 – Algoritmo de Têmpera Simulada desenvolvido com o framework JaBTeG.

6.2 Adequação à interface visual

Após o desenvolvimento dessas quatro técnicas de geração de dados de teste foi preciso que cada uma implementasse a interface *IGUIGenerationStrategy*. A implementação dessa interface foi necessária para que as técnicas de geração de dados de teste pudessem fornecer dados de seus componentes suportados para a interface visual do *plug-in*. Este recurso é necessário, pois auxilia na composição da interface visual ao restringir os dados que podem ser parametrizados pelo usuário testador. Por padrão, todas as técnicas metaheurísticas suportam todas as funções de aptidão, algoritmos de seleção e critérios de teste; a técnica de geração aleatória de dados de teste não suporta o uso de funções de aptidão e algoritmos de seleção, mas suporta todos os critérios de teste.

A criação de novos componentes para a geração de dados de teste utilizando o *framework* JaBTeG mostrou-se trivial, não exigindo muito esforço em sua customização. A Tabela 6.1 detalha a quantidade de linhas necessárias para implementação de cada um dos componentes de geração de dados de teste detalhados nessa seção.

Tabela 6.1 – *Esforço em linhas de código para criação de componentes de geração de dados de teste.*

Recurso	Tipo	Linhas de Código
Algoritmo Aleatório	Técnica de geração de dados de teste	42
Algoritmo Evolucionário	Técnica de geração de dados de teste	30
Têmpera Simulada	Técnica de geração de dados de teste	148
Subida de Encosta	Técnica de geração de dados de teste	107
Função de Similaridade	Função de aptidão	57
Função de Ineditismo	Função de aptidão	101
Seleção por Elitismo	Técnica de Seleção	16
Seleção por Torneio	Técnica de Seleção	28
Seleção por Roleta	Técnica de Seleção	30
Seleção por Média	Técnica de Seleção	29

6.3 Instalação de *plug-ins*

O processo de instalação de *plug-ins* na ferramenta JaBuTi é de fácil execução e não exige o uso de arquivos de configuração ou qualquer outro artefato. Para iniciar a instalação é preciso abrir a ferramenta de teste JaBuTi, conforme a Figura 6.7. No menu “JaBTeG” acessar o submenu com o rótulo “*Add new test data generation...*” (em português, “Adicionar nova geração de dados de teste...”), conforme a Figura 6.8, que apresenta ao usuário a janela de instalação de novos *plug-ins*.

Depois de aberta, a janela de instalação de novos *plug-ins* apresenta dois campos, “*Plug-in name*” e “*Plug-in path*”. “*Plug-in name*” é um campo descritivo cujo valor é utilizado para compor o menu “JaBTeG”, associando o ícone do menu ao *plug-in* instalado. O campo “*Plug-in path*” tem a função de indicar o caminho para um arquivo *bytecode* Java que contenha técnicas de geração de dados de teste customizadas que façam uso do *framework* JaBTeG. A Figura 6.9 apresenta os campos “*Plug-in name*” e “*Plug-in path*” preenchidos. Para concluir a instalação é preciso pressionar o botão “*Install*”.

Após a instalação do *plug-in* de geração de dados de teste, o menu “JaBTeG” é preenchido com mais um submenu, um submenu que carrega o *framework* que foi instalado, conforme a Figura 6.10. A Figura 6.11 apresenta o formulário do *plug-in* de geração de

Figura 6.7 – Ferramenta de teste JaBuTi.

Figura 6.8 – Menu para instalação de novos plug-ins de geração de dados de teste.

Figura 6.9 – Janela de instalação de novos plug-ins de geração de dados de teste.

dados de teste aberto com os dados das técnicas de geração de dados de teste contidos no *plug-in* instalado.

Figura 6.10 – Submenu do novo *plug-in* instalado.

Figura 6.11 – Formulário do *plug-in* com dados das técnicas de geração de dados de teste.

6.4 Configuração da ferramenta de teste

Para integração completa entre o framework JaBTeG e a ferramenta de teste JaBUTi poucas adaptações tiveram de ser feitas. As duas adaptações mais relevantes foram: a capacidade de instalar novos *plug-ins* de geração de dados de teste e a criação de uma interface visual genérica para renderização destes *plug-ins*.

Para facilitar a composição de diferentes cenários de teste foi necessária a criação de uma interface visual que permitisse a seleção dos componentes de geração de dados de teste, além da parametrização de valores complementares às lógicas de geração de dados de teste. A Figura 6.12 apresenta a interface visual genérica de renderização de *plug-ins*. A interface visual genérica identifica, nos *plug-ins* instalados, classes que implementem a interface *IGUIGenerationStrategy*, pois esta interface fornece dados que auxiliam na composição dos controles da interface visual. O Apêndice G apresenta a implementação da interface *IGUIGenerationStrategy* construída para a técnica de geração de dados de teste Têmpera Simulada.

Figura 6.12 – Interface visual genérica para composição de cenários de teste.

Na interface visual apresentada na Figura 6.12, pode-se notar um controle à esquerda que lista a classe sob teste e seus respectivos métodos; é por este controle visual que o método a ser testado é selecionado. No centro da interface visual existem dois controles; o controle mais ao alto lista as técnicas de geração de dados de teste existentes no *plug-in* e o controle abaixo lista as técnicas de seleção suportadas. Ao lado direito da interface visual notam-se dois controles; o controle no alto exibe as funções de aptidão suportadas pela técnica de teste, enquanto que o controle abaixo apresenta os critérios de teste suportados pela técnica de geração de dados de teste. Cada técnica de geração de dados de teste, por implementar a interface *IGUIGenerationStrategy*, possui um conjunto próprio de funções de aptidão, técnicas de seleção e critérios de teste suportados. Ao selecionar diferentes técnicas de geração de dados de teste, os controles visuais modificam

seu conteúdo, permitindo apenas a seleção dos respectivos componentes suportados pela técnica de geração de dados de teste.

Na parte de baixo da interface visual podem ser percebidas diversas abas que contêm controles que parametrizam diferentes variáveis que guiam a execução da geração de dados de teste ou apresentam o resultado de sua execução. Estas abas são:

- *Result*: apresenta dados do teste, tais como: nome da classe testada, assinatura do método testado, nome da técnica de geração de dados de teste utilizada, função de aptidão utilizada, quantidade de requisitos de teste identificados, quantidade de requisitos de teste cobertos, tempo de duração da execução, horário inicial e final da geração, número de varreduras na população e o número de vezes que os arquivos JUnit tiveram de ser executados.
- *Trace*: rastro da execução da geração de dados de teste; lista todos os passos executados para geração de dados de teste.
- *Input Domain*: controles para configuração do domínio de entrada global da geração aleatória dirigida.
- *Input Domain - Selected Method*: controles customizados que permitem a configuração de um domínio de entrada diferente para cada um dos parâmetros do método sob teste selecionado. O domínio de entrada de cada parâmetro do método sob teste sobrescreve o uso do domínio de entrada global, restringindo a geração de valores aleatórios de cada parâmetro apenas ao seu respectivo domínio de entrada.
- *Environment Variables*: controles para configuração de variáveis de ambiente, como: caminho para o *framework* de teste de unidade JUnit e caminho para o arquivo *bytecode* da classe sob teste.
- *Resulting Individuals*: após a geração de dados de teste os indivíduos de teste gerados são apresentados nesta aba no formato JUnit.
- *Parameters*: aba própria para renderização de parâmetros personalizados e parâmetros fixos (e.g., tamanho inicial da população de indivíduos).

Essa interface visual foi construída em um módulo da ferramenta JaBUTi, e pode ser reutilizada por outras ferramentas de teste que sejam compatíveis com as tecnologias utilizadas para sua construção.

6.5 Aplicação de Técnicas de Geração de Dados de Teste

Para demonstrar o funcionamento do *framework* JaBTeG, com as diferentes técnicas de geração de dados de teste construídas, são apresentados alguns dos experimentos realizados, dividindo-os entre geração de dados de teste para tipos primitivos e objetos. O objetivo destas validações é apresentar como os *benchmarks* podem ser utilizados para validar técnicas de geração de dados de teste.

Para realização dos experimentos foi utilizado um computador com sistema operacional Microsoft Windows 7/64 bits, processador Intel i7 2.50GHz, 4Gb RAM e Java versão 1.6.

6.5.1 Geração de dados de teste para tipos primitivos

Os experimentos de geração de dados de teste para tipos primitivos foram focados em *benchmarks* utilizados na literatura, sendo eles: *sorting*, conjunto de algoritmos de ordenação (DO et al., 2005); e *trityp*, composição de triângulos (ARAKI, 2009). Os experimentos utilizaram diferentes variantes de estratégias de geração de dados de teste. Os parâmetros que variaram foram os seguintes: técnica de geração de dados de teste, função de aptidão, algoritmo de seleção e critério de teste. Para realização dos experimentos foram utilizadas 17 variantes de estratégias de geração de dados de teste, combinadas com diferentes métodos de teste e apresentadas nas tabelas de resultados dos experimentos. Todas as estratégias de geração de dados de teste não levam em consideração fluxos de dados e de controle relativos ao código de exceção. Os comentários e a avaliação crítica dos experimentos são apresentados na Seção 6.6.

Sorting

Sorting trata-se de um *benchmark* localizado no repositório SIR (*Software-artifact Infrastructure Repository for Experimentation*)(DO et al., 2005). Este *benchmark* reúne diferentes algoritmos de ordenação, tais como: *Insertion Sort*, *Quick Sort* e *Merge Sort*. Todos os experimentos utilizaram um domínio de entrada com valores inteiros de 0 a 100.

O primeiro algoritmo a ser testado foi o algoritmo *Insertion Sort*, seus resultados são apresentados na Tabela 6.2. Pode-se notar que os resultados foram satisfatórios e em diferentes estratégias pode-se obter 100% da cobertura (com os critérios todos-nós

e todos-ramos) do algoritmo ou valores próximos (com os critérios todos-usos e todos-potenciais-usos). Seu código pode ser visto no Apêndice H.

Quick Sort foi o segundo algoritmo testado, utilizando diferentes variantes de geração de dados de teste, conforme a Tabela 6.3. Seu código pode ser encontrado no Apêndice H.

Os resultados obtidos com os testes do algoritmo *Merge Sort* são apresentados na Tabela 6.4, seu código é apresentado no Apêndice H.

Tabela 6.2 – Geração de dados de teste para o método *Insertion Sort*.

Técnica	Critério de teste	Aptidão	Seleção	Cobertura	Tempo de Execução
Aleatória	todos-nós	-	-	100%	4,48 seg
Aleatória	todos-ramos	-	-	100%	4,53 seg
Aleatória	todos-usos	-	-	97,22%	4,59 seg
Aleatória	todos-potenciais-usos	-	-	97,77%	4,651 seg
Evolucionário	todos-nós	Similaridade	Elitismo	100%	4,52 seg
Evolucionário	todos-nós	Ineditismo	Elitismo	100%	4,68 seg
Evolucionário	todos-nós	Similaridade	Torneio	91,66%	12,36 seg
Evolucionário	todos-nós	Similaridade	Roleta	100%	4,73 seg
Evolucionário	todos-nós	Ineditismo	Torneio	100%	4,43 seg
Evolucionário	todos-nós	Ineditismo	Roleta	95,83%	5,0 seg
Evolucionário	todos-usos	Similaridade	Elitismo	97,22%	8,7 seg
Evolucionário	todos-potenciais-usos	Similaridade	Elitismo	97,77%	12,2 seg
Subida de Encosta	todos-nós	Similaridade	Elitismo	100%	5,3 seg
Subida de Encosta	todos-ramos	Similaridade	Elitismo	100%	5,15 seg
Subida de Encosta	todos-usos	Similaridade	Elitismo	97,22%	1 min 45 seg
Têmpera Simulada	todos-nós	Similaridade	Elitismo	100%	4,44 seg
Têmpera Simulada	todos-usos	Similaridade	Elitismo	97,22%	27 min 9 seg

Tabela 6.3 – Geração de dados de teste para o método *Quick Sort*

Técnica	Critério de teste	Aptidão	Seleção	Cobertura	Tempo de Execução
Aleatória	todos-nós	-	-	33,33%	3,38 seg
Aleatória	todos-ramos	-	-	16,66%	3,26 seg
Aleatória	todos-usos	-	-	27,027%	4,85 seg
Aleatória	todos-potenciais-usos	-	-	16,66%	4,82 seg
Evolucionário	todos-nós	Similaridade	Elitismo	25,92%	25,93 seg
Evolucionário	todos-nós	Ineditismo	Elitismo	33,33%	24,79 seg
Evolucionário	todos-nós	Similaridade	Torneio	33,33%	24,40 seg
Evolucionário	todos-nós	Similaridade	Roleta	33,33%	25,37 seg
Evolucionário	todos-nós	Ineditismo	Torneio	33,33%	23,56 seg
Evolucionário	todos-nós	Ineditismo	Roleta	33,33%	22,72 seg
Evolucionário	todos-usos	Similaridade	Elitismo	27,027%	1 min 35 seg
Evolucionário	todos-potenciais-usos	Similaridade	Elitismo	16,66%	3 min 56 seg
Subida de Encosta	todos-nós	Similaridade	Elitismo	33,33%	6 min 55 seg
Subida de Encosta	todos-ramos	Similaridade	Elitismo	13,88%	7 min 25 seg
Subida de Encosta	todos-usos	Similaridade	Elitismo	27,02%	28 min 14 seg
Têmpera Simulada	todos-nós	Similaridade	Elitismo	33,33%	181 min 15 seg
Têmpera Simulada	todos-usos	Similaridade	Elitismo	54,16%	133 min 50 seg

Tabela 6.4 – Geração de dados de teste para o método Merge Sort.

Técnica	Critério de teste	Aptidão	Seleção	Cobertura	Tempo de Execução
Aleatória	todos-nós	-	-	100%	4,10 seg
Aleatória	todos-ramos	-	-	66,66%	4,81 seg
Aleatória	todos-usos	-	-	100%	5,28 seg
Aleatória	todos-potenciais-usos	-	-	94,11%	3,71 seg
Evolucionário	todos-nós	Similaridade	Elitismo	100%	5,60 seg
Evolucionário	todos-nós	Ineditismo	Elitismo	100%	5,67 seg
Evolucionário	todos-nós	Similaridade	Torneio	100%	4,24 seg
Evolucionário	todos-nós	Similaridade	Roleta	100%	4,62 seg
Evolucionário	todos-nós	Ineditismo	Torneio	100%	5,69 seg
Evolucionário	todos-nós	Ineditismo	Roleta	100%	4,81 seg
Evolucionário	todos-usos	Similaridade	Elitismo	100%	4,438 seg
Evolucionário	todos-potenciais-usos	Similaridade	Elitismo	94,11%	10,67 seg
Subida de Encosta	todos-nós	Similaridade	Elitismo	100%	5,109 seg
Subida de Encosta	todos-ramos	Similaridade	Elitismo	66,66%	2 min
Subida de Encosta	todos-usos	Similaridade	Elitismo	100%	4,11 seg
Têmpera Simulada	todos-nós	Similaridade	Elitismo	100%	3,87 seg
Têmpera Simulada	todos-usos	Similaridade	Elitismo	100%	4,68 seg

Trityp

O programa *Trityp* verifica se as suas três entradas compostas por números inteiros formam um triângulo (ARAKI, 2009). Este é um exemplo simples, mas comumente utilizado na literatura. Foram executadas duas séries de experimentos com este programa. A primeira série de experimentos utilizou um domínio de entrada de valores inteiros entre 0 e 100, os resultados obtidos são apresentados na Tabela 6.5. A segunda série de experimentos utilizou um domínio de entrada de valores inteiros entre 0 e 10. Os resultados dessa segunda série de experimentos são apresentados na Tabela 6.6. O código fonte do programa *Trityp* pode ser encontrado no Apêndice H.

Tabela 6.5 – Geração de dados de teste para o método Trityp - Inteiros de 0 a 100.

Técnica	Critério de teste	Aptidão	Seleção	Cobertura	Tempo de Execução
Aleatória	todos-nós	-	-	50%	4,5 seg
Aleatória	todos-ramos	-	-	69,44%	4,4 seg
Aleatória	todos-usos	-	-	33,33%	3,7 seg
Aleatória	todos-potenciais-usos	-	-	75%	4,12 seg
Evolucionário	todos-nós	Similaridade	Elitismo	53,33%	24,765 seg
Evolucionário	todos-nós	Ineditismo	Elitismo	60%	17,57 seg
Evolucionário	todos-nós	Similaridade	Torneio	56,66%	19,47 seg
Evolucionário	todos-nós	Similaridade	Roleta	66,66%	18,79 seg
Evolucionário	todos-nós	Ineditismo	Torneio	60%	17,64 seg
Evolucionário	todos-nós	Ineditismo	Roleta	63,33%	15,46 seg
Evolucionário	todos-usos	Similaridade	Elitismo	48,61%	39,25 seg
Evolucionário	todos-potenciais-usos	Similaridade	Elitismo	75%	49,74 seg
Subida de Encosta	todos-nós	Similaridade	Elitismo	66,66%	3 min 50 seg
Subida de Encosta	todos-ramos	Similaridade	Elitismo	66,66%	3,58 seg
Subida de Encosta	todos-usos	Similaridade	Elitismo	54,16%	12 min 6 seg
Têmpera Simulada	todos-nós	Similaridade	Elitismo	90%	35 min 53 seg
Têmpera Simulada	todos-usos	Similaridade	Elitismo	70,83%	59 min 46 seg

Tabela 6.6 – Geração de dados de teste para o método *Trityp* - Inteiros de 0 a 10.

Técnica	Critério de teste	Aptidão	Seleção	Cobertura	Tempo de Execução
Aleatória	todos-nós	-	-	90%	3,791 seg
Aleatória	todos-ramos	-	-	75%	3,75 seg
Aleatória	todos-usos	-	-	75%	4,469 seg
Aleatória	todos-potenciais-usos	-	-	75%	3,484 seg
Evolucionário	todos-nós	Similaridade	Elitismo	100%	4,594 seg
Evolucionário	todos-nós	Ineditismo	Elitismo	90%	15,109 seg
Evolucionário	todos-nós	Similaridade	Torneio	100%	3,563 seg
Evolucionário	todos-nós	Similaridade	Roleta	100%	7,531 seg
Evolucionário	todos-nós	Ineditismo	Torneio	100%	3,828 seg
Evolucionário	todos-nós	Ineditismo	Roleta	100%	3,625 seg
Evolucionário	todos-usos	Similaridade	Elitismo	83,33%	26,172 seg
Evolucionário	todos-potenciais-usos	Similaridade	Elitismo	85,93%	35,516 seg
Subida de Encosta	todos-nós	Similaridade	Elitismo	100%	4,235 seg
Subida de Encosta	todos-ramos	Similaridade	Elitismo	83,33%	3 min 28 seg
Subida de Encosta	todos-usos	Similaridade	Elitismo	66,66%	5 min 32 seg
Têmpera Simulada	todos-nós	Similaridade	Elitismo	100%	47,859 seg
Têmpera Simulada	todos-usos	Similaridade	Elitismo	75%	49 min 46 seg

6.5.2 Geração de dados de teste para objetos

Para exemplificar a geração de dados de teste para objetos foi construída uma estrutura de classes, ilustrada na Figura 6.13, que fornece construtores e métodos que exigem a passagem de objetos como parâmetros.

Figura 6.13 – Modelo de entidades utilizado pela técnica.

Esta estrutura possui quatro classes (Carro, Motor, Radiador e Cambio), estas classes dependem umas das outras e são utilizadas como parâmetros dos métodos, o que obriga a criação dinâmica de instâncias de objetos para atender a passagem de parâmetros.

A geração de indivíduos mostra-se eficaz, gerando valores para os tipos primitivos, tipos complexos (objetos) e vetores. A seguir, um indivíduo gerado pela fábrica de indivíduos do *framework* JaBTeG:

\$carro=Carro(String,int):\$carro.acelerar(float)@"AAA-5555",2008,31.197704F

Pode-se observar que este indivíduo é simples, não exige a criação dinâmica de nenhum objeto ou utilização de vetores, possui apenas a criação da classe sob teste, a chamada do método sob teste (no caso o método “acelerar(float)” da classe “Carro”) e a passagem de parâmetros primitivos. De forma complementar, a Figura 6.14 apresenta o mesmo indivíduo de teste representado no formato JUnit.

```
@Test
public void testMethod(){
    Carro carro= new Carro("AAA-5555",2008);
    carro.acelerar(31.197704F);
}
```

Figura 6.14 – Teste unitário de um indivíduo simples gerado pelo *framework* JaBTeG.

No exemplo a seguir, pode-se observar um indivíduo mais complexo do que o anterior, exigindo a construção dinâmica de objetos e utilização de vetores:

**\$carro=Carro():\$carro.toString():\$motor=Motor(String):\$motor.setCalibragem(int[]):
\$radiador=Radiador():\$radiador.inicializar():
\$motor.adicionarRadiador(\$radiador):\$carro.incluirMotor(\$motor):
\$carro.setCambioAutomatico(boolean):\$carro.hashCode():
\$carro.acelerar(float)@"AAA-5555",[2011,2010,2008,2006,2006],false,151.01656F**

No início do indivíduo se vê o construtor utilizado para criação do método sob teste, seguido pela sequência de métodos intermediários e a chamada do método sob teste, e, do lado direito do símbolo de arroba, os valores utilizados nos parâmetros primitivos.

Na Figura 6.15 pode-se observar o método de teste JUnit criado a partir do indivíduo gerado. Nota-se que a sequência de chamadas aos métodos intermediários continua sendo igual, a única mudança é a forma de representação do indivíduo de teste.

A técnica apresentou custo aceitável para geração dos indivíduos de teste, isto é, gerou uma quantidade significativa de indivíduos dentro de um período de tempo aceitável. A Tabela 6.7 apresenta a quantidade de indivíduos gerados e o tempo necessário para criá-los.

```

@Test
public void testMethod(){
    Carro carro= new Carro();
    carro.toString();
    Motor motor= new Motor("AAA-5555");
    motor.setCalibragem(new int[]{2011,2010,2008,2006,2006});
    Radiador radiador= new Radiador();
    radiador.inicializar();
    motor.adicionarRadiador(radiador);
    carro.incluirMotor(motor);
    carro.setCambioAutomatico(false);
    carro.hashCode();
    carro.acelerar(151.01656F);
}

```

Figura 6.15 – Teste unitário de um dos indivíduos gerado pelo framework JaBTeG.

Tabela 6.7 – Quantidade de indivíduos gerados e o tempo de execução necessário para criá-los.

Indivíduos	Tempo (segundos)
100	0,109
1.000	0,235
10.000	0,766
100.000	5,86

6.6 Discussão dos resultados

Este experimento apresenta a criação de uma biblioteca de geração de dados de teste por meio dos recursos providos pelo *framework* JaBTeG, além de apresentar os experimentos realizados sobre *benchmarks* utilizados na literatura. Dividiremos a discussão dos resultados em três aspectos do *framework*: recursos do *framework*, tipos primitivos e objetos complexos.

6.6.1 Recursos do *framework*

Pode-se notar, por meio da Tabela 6.1, que a quantidade de linhas de código desenvolvidas para construção dos componentes é baixa, devido a boa parte do funcionamento das técnicas estar encapsulado no *framework*. O uso do *framework* JaBTeG facilita a criação de novos componentes de geração de dados de teste, pois encapsula todo o custo da criação de recursos para execução da geração de dados de teste, e exige apenas que as *regras de negócio* dos componentes sejam definidas. Desta maneira, o desenvolvedor de novos componentes de geração de dados de teste tem apenas que trabalhar com o comportamento de seus componentes, sem implementar recursos que deem suporte operacional à geração de dados de teste.

A instalação de novos *plug-ins* na ferramenta JaBUTi é fácil de ser realizada, sem que

seja preciso preencher arquivos de configuração ou qualquer outro artefato. Basta fornecer um arquivo *bytecode* Java com os componentes criados a partir da estrutura reutilizável do *framework* JaBTeG.

Por meio da interface visual fornecida pela ferramenta JaBUTi fica fácil combinar diferentes componentes de geração de dados de teste. O uso dessa interface visual favorece a comparação entre diferentes cenários de teste, que fazem uso de diferentes técnicas de geração de dados de teste, associadas com diferentes funções de aptidão, combinadas com critérios de seleção em meio a diversos critérios de teste. Este tipo de composição, de fácil parametrização, auxilia na descoberta de diferentes resultados, pois cenários de teste que antes eram difíceis de serem simulados agora podem ser facilmente compostos.

6.6.2 Tipos primitivos

A geração de dados de teste para o método *Insertion Sort* mostrou-se bastante satisfatória. Os indivíduos de teste criados foram capazes de cobrir todos os requisitos de teste indicados pelos critérios todos-nós e todos-ramos, enquanto que os critérios todos-usos e todos-potenciais-usos obtiveram resultados muito bons, chegando próximo à cobertura de todos seus requisitos de teste. Nota-se também que a técnica de geração aleatória de indivíduos apresentou resultados muito próximos aos resultados obtidos com as técnicas metaheurísticas e em tempos muito próximos, devido ao fato de ser um algoritmo trivial de ser coberto. Pode-se notar que os critérios todos-usos e todos-potenciais-usos obtiveram a cobertura máxima de 97% em diferentes variantes de geração de dados de teste mesmo após vários minutos de execução e diversas tentativas de cobertura, devido à existência de requisitos de teste não executáveis.

De forma contrária, a geração de dados de teste para o método *Quick Sort* não apresentou resultados tão satisfatórios quanto o método *Insertion Sort*. A cobertura mais alta foi de 54,16% obtida com o critério de teste todos-usos e a cobertura mais baixa foi de 13,88% obtida com o critério de teste todos-ramos.

Observando o código do método *Quick Sort* (Apêndice H), nota-se que o seu funcionamento exige a definição de um elemento do vetor de valores, denominado pivô. Este elemento rearranja o vetor, de forma que todos elementos anteriores sejam menores que ele, e todos os elementos posteriores sejam maiores do que ele. O valor do pivô é definido pelo método *median3* e é baseado em dois parâmetros de entrada do método *Quick Sort*, chamados *left* e *right*. Ao observar o método *median3*, nota-se que o valor do elemento pivô é gerado a partir da média entre os valores de *left* e *right*, e logo em seguida, a posição

do vetor indexada pelo valor do elemento pivô é acessada.

Durante a geração aleatória de dados de teste, diferentes valores aleatórios foram gerados para os parâmetros *left* e *right*. Em sua maioria, os valores gerados acarretaram em estouros de posição de memória, devido à média dos valores *left* e *right* ser maior do que as posições existentes no vetor. Neste teste, o uso de geração aleatória na definição dos valores dos parâmetros e no tamanho do vetor não é indicado, pois os valores dos parâmetros *left* e *right* estão diretamente relacionados ao tamanho do vetor. Assim, pode-se perceber que os resultados baixos obtidos com os testes do método *Quick Sort* estão relacionados a erros que ocorreram durante a execução dos indivíduos de teste.

O método *Merge Sort* obteve resultados tão satisfatórios quanto o método *Insertion Sort*. Diferentes variantes de geração de dados de teste foram capazes de cobrir todos os requisitos de teste com os critérios de teste todos-nós e todos-usos, enquanto que os critérios todos-ramos e todos-potenciais-usos obtiveram coberturas de 66,66% e 94,11%, respectivamente. Em meio às execuções do método *Merge Sort* (conforme Tabela 6.4) percebe-se que execuções que utilizaram os critérios de teste todos-ramos e todos-potenciais-usos mesmo com diferentes funções de aptidão, critérios de seleção e técnicas de geração, obtiveram os mesmos resultados. Desta maneira percebe-se que, assim como no método *Insertion Sort*, requisitos de teste não executáveis possam existir ao utilizarmos os critérios de teste todos-ramos e todos-potenciais-usos.

A primeira série de experimentos com o método *Trytip* (conforme Tabela 6.5) geraram resultados insatisfatórios, com nenhuma das variantes de geração de dados de teste cobrindo todos os requisitos de teste. Diante desses resultados, observou-se que (conforme o Apêndice H) internamente o método *Trytip* executa comparações entre os parâmetros, averiguando se são iguais. Este tipo de comparação não é favorável quando utilizamos parâmetros gerados aleatoriamente, pois a probabilidade de dois parâmetros conterem os mesmos valores quando gerados aleatoriamente é bastante baixa se utilizado um domínio grande de valores de entrada. Com base nessas avaliações, executou-se uma segunda série de experimentos com o método *Trytip* limitando o escopo de seu domínio de valores inteiros através do uso da geração aleatória dirigida. Assim, limitaram-se os valores a um domínio de inteiros entre 0 (zero) e 10 (dez). Com esta configuração a probabilidade de dois valores gerados aleatoriamente serem iguais é maior. Com isso, obtiveram-se resultados mais satisfatórios (conforme Tabela 6.6), no qual algumas das variantes de geração de dados de teste conseguiram cobrir todos os requisitos de teste.

Após avaliar todos estes métodos percebe-se que os critérios de teste todos-usos e

todos-potenciais-usos possuem os requisitos mais difíceis de serem cobertos, e que possivelmente as funções de aptidão ou os algoritmos de seleção existentes não são eficazes o suficiente para auxiliar na cobertura de todos os requisitos de teste desses critérios. Desta forma, acredita-se que seja necessária a criação de novas funções de aptidão e algoritmos de seleção próprios para os requisitos de teste identificados pelos critérios de teste todos-usos e todos-potenciais-usos.

Com estes experimentos também foi possível averiguar que o *framework* JaBTeG apresentou custo reduzido na execução da maioria das variantes de geração de dados de teste, no qual foi possível em poucos segundos se obter uma população de indivíduos que estimulasse a execução do método sob teste, poupando esforços do usuário testador em gerar dados de teste manualmente.

É importante ressaltar que não foi o objetivo fazer uma avaliação exaustiva das técnicas de geração de dados de teste, mas demonstrar como o *framework* facilita a sua composição. Os algoritmos não foram testados exaustivamente, foram construídos com base em textos da literatura e utilizados como prova de conceito de uso do *framework* JaBTeG.

6.6.3 Objetos complexos

Com base nos experimentos realizados, pode-se notar que o *framework* JaBTeG foi eficaz na criação dinâmica de objetos complexos durante a geração de dados de teste.

Dada a classe “*Carro*” e o método “*acelerar(float)*”, respectivamente classe e método sob teste, diferentes indivíduos de teste foram gerados, a fim de compor uma população de indivíduos de teste que estimula-se a cobertura dos requisitos de teste existentes no método sob teste.

Esses indivíduos de teste são compostos por: uma chamada do construtor da classe sob teste, uma chamada do método sob teste e uma série de chamadas a métodos intermediários. Percebe-se que, quando um método ou construtor exige o uso de um parâmetro que seja um tipo complexo, o *framework* JaBTeG é capaz de construir dinamicamente uma instância do tipo de dados requerido, além de compor seu estado com chamadas a métodos intermediários relativos ao seu tipo de dados e o uso da geração aleatória de valores para atribuição de valor aos seus parâmetros.

Tomando como exemplo o indivíduo de teste da Figura 6.15, pode-se perceber que dentre seus métodos intermediários existe uma chamada ao método “*incluirMotor(Motor)*”.

Este método intermediário exige o uso de um parâmetro do tipo “Motor”. Devido ao fato deste tipo de dados não ser um tipo primitivo, é necessário que seja criada dinamicamente uma instância relativa a este tipo de dados, para que o indivíduo de teste possa ser executado.

Nota-se também que, ao invocar os métodos intermediários da instância de “Motor”, foram invocados os métodos “adicionarRadiador(Radiador)” e “setCalibragem(int[])”, dois métodos cujos parâmetros não são tipos primitivos. Mesmo a instância de “Motor” sendo criada a partir de um método intermediário, os valores dos parâmetros desses métodos foram criados corretamente. Este exemplo de indivíduo de teste mostra como o *framework* JaBTeG é eficaz na criação dinâmica de indivíduos de teste, estimulando diferentes estados dos objetos criados, por meio do uso de métodos intermediários e da composição dinâmica de seus parâmetros.

A maioria das ferramentas de geração de dados de teste apenas geram dados para tipos primitivos, suportando apenas aplicações escritas em linguagens procedimentais e parte das aplicações que utilizam linguagens orientadas a objetos. Ao comparar o *framework* JaBTeG com outras abordagens, também geradoras de dados de teste para software orientado a objeto, notam-se características em comum, além de recursos inéditos entre os trabalhos identificados na literatura.

Em comparação com a ferramenta Randoop (PACHECO; ERNST, 2007; PACHECO et al., 2008), o *framework* JaBTeG também cria sequências de chamadas a métodos para estimular diferentes estados nos objetos, mas ao invés de utilizar *Feedback-Directed Random Testing* utiliza geração aleatória dirigida de valores.

Assim como a ferramenta TestFul (MIRAZ et al., 2009; BARESI; MIRAZ, 2010), o *framework* JaBTeG cria seus indivíduos de teste colocando-os em seus devidos estados e fornecendo valores para os parâmetros de entrada de seus métodos. Além de também realizar uma pré-análise da classe sob teste, identificando todos os tipos envolvidos no teste, a fim de identificar todas as dependências envolvidas no teste e tornar-se mais eficiente.

Silva e Someren (2010) combinam em sua abordagem algoritmos genéticos com análise estática. A análise estática busca a ocorrência de valores primitivos, a fim de guiar a geração aleatória de valores. O *framework* JaBTeG também faz uso da geração aleatória dirigida para guiar a criação de seus valores aleatórios, com uma diferença: seu domínio de valores é parametrizável, não sendo definido por meio de valores primitivos encontrados dentro da classe sob teste.

O *framework* desenvolvido compartilha características com os trabalhos citados anteriormente e provê recursos que os demais trabalhos da literatura não fornecem. A geração dinâmica de instâncias de objetos não é um recurso compartilhado por todas as técnicas de geração de dados de teste para programas escritos em linguagens orientadas a objetos. O *framework* JaBTeG não somente cria instâncias dinâmicas de objetos como também estimula a construção de diferentes estados para os objetos, por meio da chamada de métodos intermediários. Essa técnica de geração de dados de teste também difere das demais ao gerar dados para vetores, recurso que não foi encontrado nas demais técnicas. Mais um recurso que sobressai das demais técnicas é a possibilidade de parametrizar valores para geração aleatória dirigida, que ao contrário de trabalhos como o de Silva e van Someren (2010), não é obrigatório e permite a interferência do testador. Outra diferença com demais trabalhos é a possibilidade de exportar os indivíduos gerados em dois formatos distintos, sendo eles: a representação proposta por Tonella (2004) e classes de teste no formato JUnit.

A fábrica de indivíduos do *framework* JaBTeG apresentou custo reduzido na geração de indivíduos de teste, gerando mil indivíduos em 0,235 segundo (235 milésimos de segundo) e 100.000 (cem mil) indivíduos em 5,85 segundos.

6.7 Considerações Finais

Este capítulo apresentou a biblioteca de geração de dados de teste construída para validação do *framework* JaBTeG. Também foram apresentadas as customizações feitas na ferramenta JaBUTi para adição de uma interface visual que permitisse a fácil composição de diferentes cenários de geração de dados de teste utilizando os componentes criados por meio do *framework* desenvolvido. Além disso, também foram apresentados os experimentos criados para geração de dados de teste de tipos primitivos e objetos, com base em *benchmarks* utilizados na literatura. No próximo capítulo serão apresentadas as conclusões e os trabalhos propostos.

Capítulo 7

Conclusão

Este trabalho teve como objetivo a criação de um *framework* extensível para facilitar o desenvolvimento de técnicas de geração de dados de teste, para que o desenvolvedor de técnicas de geração de dados de teste concentre seus esforços no desenvolvimento de componentes de geração de dados de teste, sem preocupar-se com os recursos necessários para execução dessas técnicas.

Síntese

O *framework* JaBTeG é um arcabouço que provê recursos para geração de dados de teste, além de fornecer uma estrutura extensível para composição de novas estratégia de geração de dados de teste. Seu funcionamento analisa *bytecodes* Java, de modo que nenhum código fonte é necessário para geração de dados de teste. A estrutura do *framework* JaBTeG é composta por seis módulos, a saber: leitor de *bytecode*, interpreta o conteúdo de arquivos *bytecode*; fábrica de indivíduos, encapsulada funções de geração de valores para tipos de dados primitivos e complexos; biblioteca de algoritmos de geração, fornece recursos para implementação de técnicas de geração automática de dados de teste; gerador de testes de unidade, provê recursos para construção de arquivos no formato JUnit, necessários para execução da geração de dados de teste; biblioteca de funções de aptidão, provê recursos para criação de funções de aptidão; e biblioteca de algoritmos de seleção, provê recursos para criação de algoritmos de seleção.

Para validação do *framework* foi construída uma biblioteca de algoritmos de geração de dados de teste. Essa biblioteca de algoritmos é composta por quatro técnicas: aleatória, evolucionária, subida de encosta e têmpera simulada. Além das técnicas de geração de dados de teste, essa biblioteca contém duas funções de aptidão (similaridade e ineditismo) e quatro estratégias de seleção (elitismo, torneio, roleta e média).

O *framework* JaBTeG foi associado à ferramenta de teste JaBUTi, exigindo alterações em sua estrutura para possibilitar a instalação de *plug-ins* geradores de dados de teste construídos por meio da estrutura reutilizável do *framework* JaBTeG, além da composição de uma interface visual genérica que favorece a rápida composição de diferentes cenários de teste por meio da seleção dos componentes de geração de dados de teste (e.g., técnica

de geração, função de aptidão, estratégia de seleção e critérios de teste).

Contribuições

Ao término deste trabalho pode-se identificar um conjunto de contribuições relevantes, sendo elas:

Construção de um *framework* que provê recursos para geração de dados de teste: o *framework* JaBTeG fornece recursos que auxiliam na geração de dados de teste, tais como: geração aleatória de indivíduos, geração de valores para vetores e objetos complexos, geração aleatória dirigida para o domínio da aplicação e exportação dos resultados em dois formatos distintos (JUnit e representação de Tonella (2004)); além disso, fornece uma estrutura extensível para composição de técnicas de geração de dados de teste que requer pouco esforço de implementação: um conjunto de estruturas reutilizáveis que auxiliam na composição de técnicas de geração de dados de teste, funções de aptidão e algoritmos de seleção estão disponíveis;

Construção uma biblioteca de técnicas de geração de dados de teste: para validação do *framework* foi construída uma biblioteca de geração de dados de teste composta por quatro técnicas de geração de dados de teste (Aleatória, Evolucionária, Subida de Encosta e Têmpera Simulada), duas funções de aptidão (Similaridade e Ineditismo) e quatro algoritmos de seleção (Elitismo, Torneio, Roleta e Média);

Adaptação da ferramenta JaBUTi para instalação de diferentes *plug-ins* de geração de dados de teste: para adequação da ferramenta de teste com o *framework* foi necessária a construção de um módulo que suporta a instalação de *plug-ins* de geração de dados de teste; este módulo foi construído para facilitar a experimentação de diferentes técnicas de geração de dados de teste dentro da ferramenta JaBUTi;

Desenvolvimento de uma interface visual genérica que permite a composição de diferentes cenários para geração de dados de teste: para facilitar a interação do usuário testador com o *framework* JaBTeG, foi construída uma interface visual genérica que permite a composição de diferentes cenários de teste a partir dos componentes existentes em um *plug-in* de geração de dados de teste;

Durante o desenvolvimento deste trabalho parte do *framework* foi tema de uma publicação em um evento da área. O artigo “Geração Aleatória de Dados para Programas Orientados a Objetos” (FERREIRA et al., 2012) foi publicado nos anais da sexta edição do

SAST (*Brazilian Workshop on Systematic and Automated Software Testing*). Essa publicação documenta a fábrica de indivíduos aleatórios existente no *framework* JaBTeG, destacando: a estrutura dos indivíduos; a geração de valores para parâmetros de tipos primitivos, vetores e objetos; e citando a importância da geração aleatória nas técnicas metaheurísticas de geração de dados de teste.

Limitações

O *framework* JaBTeG possui duas limitações em seu funcionamento:

- Criação de parâmetros personalizados limitados a tipos primitivos, pois não é suportada a criação de parâmetros personalizados que correspondam a tipos complexos (i.e., objetos).
- Geração automática de valores limitada ao uso de classes concretas, atualmente não é possível a geração de dados de teste para interfaces e classes abstratas.

Trabalhos futuros

Como trabalhos futuros são propostos os temas:

- Utilização de técnicas de geração de dados de teste para detecção de requisitos de teste não executáveis: o *framework* JaBTeG pode ser utilizado para desenvolver técnicas que possibilitem a identificação de requisitos de teste não executáveis.
- Composição de técnicas de geração de dados de teste que utilizem funções de aptidão multi-objetivo: com base no estudos apresentados por Araki e Vergilio (2010), seria interessante incluir recursos no *framework* para a criação de técnicas que utilizem funções de aptidão multi-objetivo.
- Implementação de novas técnicas de geração de dados de teste: técnicas de geração de dados de teste recentemente propostas podem ser implementadas e experimentadas a partir dos recursos fornecidos pelo *framework* JaBTeG (e.g., geração de dados de teste por diversidade (BUENO et al., 2011)).
- Construção de funções de aptidão mais adequadas para os critérios de teste todos-potenciais-usos e todos-usos: funções de aptidão que aproveitem as características dos critérios todos-potenciais-usos e todos-usos podem ser desenvolvidas utilizando o *framework*.

- Suporte à geração de dados de teste para interfaces e classes abstratas: adequação do *framework* JaBTeG para criação de valores aleatórios a partir de interfaces e classes abstratas.
- Suporte à criação de parâmetros personalizados que utilizem tipos complexos: adequação do *framework* JaBTeG para a criação de parâmetros personalizados baseados em tipos complexos, não limitando seu uso a tipos primitivos.
- Criação de uma estrutura extensível para o suporte à execução simbólica: criação de uma estrutura reutilizável que forneça recursos para geração de dados de teste por meio de execução simbólica.
- Criação de notações para auxiliar na geração de valores aleatórios: para auxiliar na geração aleatória de valores, é proposto o uso de notações nos métodos para indicar o domínio de valores a ser utilizado durante a geração aleatória.

Referências

- ARAKI, L. Y. Um algoritmo evolutivo de geração de dados de teste para satisfazer critérios baseados em código objeto java. *Universidade Federal do Paraná*, 2009.
- ARAKI, L. Y.; VERGILIO, S. R. Um framework de geração de dados de teste para critérios estruturais baseados em código objeto java. In: XI WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS. Gramado, 2010. v. 1.
- BALL, T.; BOUNIMOVA, E.; LEVIN, V.; MOURA, L. de. Efficient evaluation of pointer predicates with z3 smt solver in slam2. In: MICROSOFT RESEARCH. Redmond, WA 98052: One Microsoft Way, 2010. Disponível em: <http://www.research.microsoft.com>.
- BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. In: _____. 1. ed. [S.l.]: Elsevier, 2007. (Campus, Rio de Janeiro - RJ), cap. Teste Estrutural, p. 47–76.
- BARESI, L.; MIRAZ, M. Testful: automatic unit-test generation for java classes. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 281–284. ISBN 978-1-60558-719-6. Disponível em: <http://doi.acm.org/10.1145/1810295.1810353>.
- BARROS, F.; TEDESCO, P. Introdução aos agentes inteligentes - algoritmos de melhorias iterativas (otimização). *Universidade Federal de Pernambuco, Centro de Informática*, 2008.
- BERTSIMAS, D.; TSITSIKLIS, J. Simulated annealing. *Statistical Science*, v. 8, n. 1, p. 10–15, 1993.
- BOYAPATI, C.; KHURSHID, S.; MARINOV, D. Korat: Automated testing based on java predicates. *ACM SIGSOFT - International Symposium on Software Testing and Analysis*, v. 27, p. 123–133, Julho 2002.
- BUENO, P.; JINO, M.; WONG, W. E. Diversity oriented test data generation using metaheuristic search techniques. *Information Sciences - Elsevier*, 2011.
- BURNIM, J.; SEN, K. Heuristics for scalable dynamic test generation. *IEEE/ACM International Conference on Automated Software Engineering*, California, Berkeley, n. UCB/EECS-2008-123, p. 443–446, Setembro 2008.
- CHAIM, M. L. *Poke-tool : uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados*. Dissertação (Mestrado) — Faculdade de Engenharia Elétrica e de Computação - FEEC, Universidade Estadual de Campinas, Campinas, 1991.

- DELAMARO, M.; BARBOSA, E. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Introdução ao teste de software. In: _____. 1. ed. [S.l.]: Elsevier, 2007. (Campus, Rio de Janeiro - RJ), cap. Teste Funcional, p. 09–25.
- DELAMARO, M.; CHAIM, M. L.; VINCENZI, A. M. R. *Técnicas e Ferramentas de Teste de Software*. Rio de Janeiro: Editora PUC-Rio, 2010. 51-110 p. (Atualizações em informática 2010 (JAI 2010), 1).
- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Introdução ao teste de software. In: _____. 1. ed. [S.l.]: Elsevier, 2007. (Campus, Rio de Janeiro - RJ), cap. Conceitos Básicos, p. 1–7.
- DENG, X.; ROBBY; HATCLIFF, J. Kiasan: A verification and test-case generation framework for java based on symbolic execution. *Institute of Electrical and Electronics Engineers - Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, p. 137, Novembro 2006.
- DENG, X.; ROBBY; HATCLIFF, J. Kiasan/kunit: Automatic test case generation and analysis feedback for open object-oriented systems. *Institute of Electrical and Electronics Engineers - Testing: Academic and Industrial Conference - Practice And Research Techniques*, p. 3 – 12, Outubro 2007.
- DO, H.; ELBAUM, S. G.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, v. 10, n. 4, p. 405–435, 2005.
- EDVARDSSON, J. A survey on automatic test data generation. *Second Conference on Computer Science and Engineering in Linköping*, Suécia, n. 2, p. 21–28, Outubro 1999.
- EIFFELSTUDIO, I. D. E. 2010. Disponível em: http://dev.eiffel.com/Main_Page.
- FABBRI, S. P. F.; VINCENZI, A. M. R.; MALDONADO, J. C. Introdução ao teste de software. In: _____. 1. ed. [S.l.]: Elsevier, 2007. (Campus, Rio de Janeiro - RJ), cap. Teste Funcional, p. 09–25.
- FERREIRA, F. H. I. B.; DELAMARO, M.; CHAIM, M. L.; NUNES, F. L. S.; VINCENZI, A. M. R. Geração aleatória de dados para programas orientados a objetos. *SAST 2012 (VI Workshop Brasileiro de Teste de Software Sistemático e Automatizado)*, Natal - RN - Brasil, v. 13, n. 6, p. 73–82, Setembro 2012.
- FERREIRA, L. P.; VERGILIO, S. R. Tdsgen: An environment based on hybrid genetic algorithm for generation of test data. *International Conference of Software Engineering and Knowledge Engineering*, Taipei, Taiwan, v. 17, p. 312–317, Julho 2005.
- FIX, G. The design of an automated unit test code generation system. *Institute of Electrical and Electronics Engineers - Sixth International Conference on Information Technology: New Generations*, p. 743 – 747, Junho 2009.
- FREITAS, F. G.; MAIA, C. L. B.; COUTINHO, D. P.; CAMPOS, G. A. L.; SOUZA, J. T. Aplicação de metaheurísticas em problemas da engenharia de software: Revisão de literatura. II Congresso Tecnológico Infobrasil, 2009.

GAMMA, E.; HELM, R.; JOHNSON, R. E.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. ISBN 978-0-201-63361-0.

GODEFROID, P.; KLARLUND, N.; SEN, K. *DART: Directed automated random testing*. ACM New York, NY, USA: ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation 2005.

GOTLIEB, A.; BOTELLA, B.; RUEHER, M. A CLP framework for computing structural test data. In: LLOYD, J. W.; DAHL, V.; FURBACH, U.; KERBER, M.; LAU, K.-K.; PALAMIDESSI, C.; PEREIRA, L. M.; SAGIV, Y.; STUCKEY, P. J. (Ed.). *Computational Logic*. Springer, 2000. v. 1861, p. 399–413. ISBN 3-540-67797-6. Disponível em: <http://dblp.uni-trier.de/db/conf/cl/cl2000.html#GotliebBR00>.

HARMAN, M. *Automated Test Data Generation using Search Based Software Engineering*. Minneapolis, MN: Second International Workshop on Automation of Software Test, 2007. 2 p.

HARMAN, M.; PH, U.; JONES, B. F. Search-based software engineering. *Information and Software Technology*, v. 43, p. 833–839, 2001.

INKUMSAH, K.; XIE, T. *Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution*. North Carolina State Univ., Raleigh, NC, Outubro 2008. 297 - 306 p.

JUNIT. *JUnit*. 2012. Disponível em: <http://www.junit.org/>.

KING, J. C. *Symbolic Executions and Program Testing*. New York, NY, USA: 23rd IEEE/ACM International Conference on Automated Software Engineering, 1976. 297 - 306 p.

KOREL, B. *Automated Software Test Data Generation*. Piscataway, NJ, USA: IEEE Transactions on Software Engineering, 1990. 870 - 879 p.

LARSON, E.; AUSTIN, T. High coverage detection of input-related security faults. In: *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003. (SSYM'03), p. 9–9. Disponível em: <http://dl.acm.org/citation.cfm?id=1251353.1251362>.

LIASKOS, K.; ROPER, M. Automatic test-data generation: An immunological approach. *Institute of Electrical and Electronics Engineers - Testing: Academic and Industrial Conference - Practice And Research Techniques*, p. 77 – 81, Setembro 2007.

LINDEN, R. *Algoritmos Genéticos - Uma Importante Ferramenta da Inteligência Computacional*. 3. ed. [S.l.]: Editora Ciência Moderna, 2012. ISBN 9788574522654.

MACHADO, A. F. V. *Fundamentos dos Algoritmos Genéticos*. 2012. Disponível em: http://www.riopomba.ifsudestemg.edu.br/dcc/dcc/materiais/591973973_slide6_AG.ppt.

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais uso. São Paulo, SP, p. 92–95, Setembro 1989.

- MCMINN, P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, v. 14, p. 105–156, 2004.
- MILLER, W.; SPOONER, D. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2, n. 3, p. 223 – 226, Setembro 1976.
- MIRAZ, M.; LANZI, P. L.; BARESI, L. Testful: Using a hybrid evolutionary algorithm for testing stateful systems. *Genetic and Evolutionary Computation Conference*, Genetic and Evolutionary Computation Conference, p. 1947–1948, Julho 2009.
- NADEEM, A.; LYU, M. R. A framework for inheritance testing from VDM++ specifications. *12th Pacific Rim International Symposium on Dependable Computing*, 2006.
- PACHECO, C.; ERNST, M. *Randoop: feedback-directed random testing for Java*. Montreal, Canada: Object-Oriented Programming, Systems, Languages & Applications, 2007. 815 - 816 p.
- PACHECO, C.; LAHIRI, S.; BALL, T. *Finding Errors in .Net with Feedback-Directed Random Testing*. Seattle, WA, USA: International Symposium on Software Testing and Analysis, 2008. 87 - 96 p.
- PINHEIRO, A. C. *Geração Automática de Dados de Teste: Visão Geral*. São Carlos, SP, Brasil: Universidade de São Paulo, 2010. Apresentado na disciplina "SSC5877 Verificação, Validação e Teste de Software". Disponível em: <http://moodle.stoa.usp.br/mod/resource/view.php?id=13681>.
- RAMAMOORTHY, C. V.; HO, S. F.; CHEN, W. T. *On the automated generation of program test data*. Piscataway, NJ, USA: IEEE Transactions on Software Engineering, 1976. 293 - 300 p.
- SAGARNA, R.; ARCURI, A.; YAO, X. Estimation of distribution algorithms for testing object oriented software. In: *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. [S.l.: s.n.], 2007. p. 438 –444.
- SEN, K.; MARINOV, D.; AGHA, G. *CUTE: A concolic Unit Testing Engine for C*. New York, NY, USA, Setembro 2005. v. 30, n. 5, 263 - 272 p.
- SHIROLE, M.; SUTHAR, A.; KUMAR, R. Generation of improved test cases from UML state diagram using genetic algorithm. *India Software Engineering Conference 2011*, p. 125–134, 2011.
- SILVA, L. S.; SOMEREN, M. van. *Evolutionary Testing of Object-Oriented Software*. New York, NY, USA, Março 2010. 1126-1130 p.
- SKINNER, M. *Genetic Algorithms Overview*. [S.l.], 2010. Disponível em: <http://geneticalgorithms.aidepot.com/Tutorial/Overview.html>.
- SRIVASTAVA, P. R.; KIM, T. Application of genetic algorithm in software testing. *International Journal of Software Engineering and Its Applications*, v. 3, n. 4, Outubro 2009.

- TILLMANN, N.; HALLEAUX, J. *Pex-white box test generation for .Net*. Prato, Italia: Tests and Proofs, Second International Conference, 2008. (Lecture Notes in Computer Science, v. 4966).
- TONELLA, P. *Evolutionary Testing of Classes*. Italia; Povo; Trento: ITC-irst Centro per la Ricerca Scientifica e Tecnologica, 2004.
- TRACEY, N.; CLARK, J.; MANDER, K.; MCDERMID, J. An automated framework for structural test-data generation. In: *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*. [S.l.: s.n.], 1998. p. 285 – 288.
- TRACEY, N.; CLARK, J.; MCDERMID, J.; MANDER, K. A search based automated test-data generation framework for safety-critical systems. In: *Systems engineering for business process change*. New York, NY, USA: [s.n.], 2002. p. 174 – 213.
- VEANES, M.; BJØRNER, N.; MOURA, L. D. *Solving Extended Regular Constraints Symbolically*. [S.l.], Dezembro 2009. Disponível em: <http://research.microsoft.com/apps/pubs/default.aspx?id=117753>.
- VERGILIO, S. R.; MALDONADO, J.; JINO, M. Introdução ao teste de software. In: _____. 1. ed. [S.l.]: Elsevier, 2007. (Campus, Rio de Janeiro - RJ), cap. Geração de Dados de Teste, p. 269–292.
- VIM. *Vim*. 2011. Disponível em: <http://www.vim.org/>.
- VINCENZI, A. M. R.; DELAMARO, M.; MALDONADO, C. *Java Bytecode - Understanding and Testing - User's Guide*. [S.l.]: Universidade de São Paulo - Instituto de Ciências Matemáticas e de Computação, 2003.
- VINCENZI, A. M. R.; DOMINGUES, A. L. dos S.; DELAMARO, M.; MALDONADO, J. Introdução ao teste de software. In: _____. 1. ed. [S.l.]: Elsevier, 2007. (Campus, Rio de Janeiro - RJ), cap. Teste Orientado a Objetos e de Componentes, p. 119–174.
- VISSER, W.; PaSaREANU, C. S.; PELÁNEK, R. Test input generation for java containers using state matching. In: *Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2006. (ISSTA '06), p. 37–48. ISBN 1-59593-263-1. Disponível em: <http://doi.acm.org/10.1145/1146238.1146243>.
- WANG, B.; ZHU, C.; SHENG, J. MDA-based automated generation method of test cases and supporting framework. *2010 2nd International Conference on Computer Engineering and Technology*, v. 1, p. 106 – 109, Junho 2010.
- XIE, T.; MARINOV, D.; NOTKIN, D. Rostra: A framework for detecting redundant object-oriented unit tests. *IEEE International Conference on Automated Software Engineering*, p. 196–205, Setembro 2004.
- ZHANG, L.; XIE, T.; TILLMANN, N.; HALLEAUX, J. Test generation via dynamic symbolic execution for mutation testing. In: _____. Seattle, WA, USA: In Proceedings of the 26th IEEE International Conference on Software Maintenance, 2010. p. 1 – 10.

APÊNDICE A

Estruturas para extensão

BaseGenerationStrategy

```

public abstract class BaseGenerationStrategy {
    protected EnumCriterion _criterion;
    protected int ExecutionCount;
    protected int InteractionCount;
    protected int PopulationSize;
    protected String JarPath;
    protected String ClassUnderTest;
    protected String MethodUnderTest;
    protected Individual[] CurrentPopulation;
    protected GenerationRequirement[] Targets;
    protected GenerationRequirement CurrentTarget;
    protected Individual currentSolutionToRequirement;
    protected ITestDataGenerationTool _testGenerationTool;
    public void initialize(String jarPath, int populationSize,
        String classUnderTest, String methodUnderTest)
    {
        this.JarPath = jarPath;
        this.ClassUnderTest = classUnderTest;
        this.MethodUnderTest = methodUnderTest;
        this.PopulationSize = populationSize;
    }
    public void setCriterion(EnumCriterion criterion)
    {
        this._criterion = criterion;
    }
    public int getCountOfInteractions()
    {
        return this.InteractionCount;
    }
    public int getCountOfExecutions()
    {
        return this.ExecutionCount;
    }
    public GenerationRequirement[] getRequirements()
    {
        return this.Targets;
    }
    public void setInitialPopulationSize(int populationSize)
    {
        this.PopulationSize = populationSize;
    }
    public void setTestDataGenerationTool(ITestDataGenerationTool testGenerationTool)
    {
        this._testGenerationTool = testGenerationTool;
    }
    // Passo 00 - Iniciar a geração automática de dados de teste
    public Individual[] generateData() throws TestStrategyException
    {

```

```

// Zera a contagem de interações
this.InteractionCount = 0;
GenerationTrace.write("Starting test data generation.");
// Processo de inicialização
GenerationTrace.write("Initializing generation.");
initializeGeneration();
try
{
    // Geração da população inicial de indivíduos
    GenerationTrace.write("Generating initial population.");
    this.CurrentPopulation = generateInitialPopulation();
    // Executa os indivíduos
    GenerationTrace.write("Executing individuals.");
    executeIndividuals();
    // Identificação dos ramos/nós que devem ser cobertos
    GenerationTrace.write("Identifying targets.");
    this.Targets = identifyRequirements();
    // Atualizar os indivíduos
    GenerationTrace.write("Updating individuals.");
    updateIndividuals();
    GenerationTrace.write("Reading targets. Count: ", this.Targets.length);
    for (int i = 0; i < this.Targets.length; i++)
    {
        // Itera o contador de interações
        this.InteractionCount++;
        this.CurrentTarget = this.Targets[i];
        GenerationTrace.write("New target selected. Target ",
                               this.CurrentTarget.getIdRequirement());
        GenerationTrace.write("Verify if requirement ",
                               this.CurrentTarget.getIdRequirement(),
                               " has been covered.");
        if (verifyIfRequirementHasBeenCovered(this.CurrentTarget))
        {
            GenerationTrace.write("Target ",
                                   this.CurrentTarget.getIdRequirement(),
                                   " covered.");
            this.CurrentTarget.isCovered(true);
        }
        else
        {
            GenerationTrace.write("Target uncovered.");
            this.CurrentTarget.isCovered(false);
        }
    }
}
catch (TestDataGenerationToolException e)
{
    throw new TestStrategyException(e);
}
catch (Exception e)
{
    throw new TestStrategyException(e);
}
// Processo de finalização
GenerationTrace.write("Finalize generation.");
finalizeGeneration();
// Retorna a população atual
GenerationTrace.write("Solution contains ",
                      this.CurrentPopulation.length,
                      " individuals.");
return this.CurrentPopulation;
}
// Inicializar o processo de geração de dados de teste
protected abstract void initializeGeneration();
// Gerar população
protected abstract Individual[] generateInitialPopulation()
    throws TestDataGenerationToolException;
// Identificar os requisitos que devem ser cobertos
protected GenerationRequirement[] identifyRequirements()
    throws TestDataGenerationToolException {

```

```

        GenerationTrace.write("Identifying targets.");
        GenerationRequirement[] returnValue =
            this._testGenerationTool.getRequirements(this._criterion);
        return returnValue;
    }
    // Compila os indivíduos e os executa
    protected void executeIndividuals() throws TestDataGenerationToolException {
        executeIndividuals(this.CurrentPopulation);
    }
    // Compila os indivíduos e os executa
    protected void executeIndividuals(Individual[] population)
        throws TestDataGenerationToolException {
        GenerationTrace.write("Executing individuals.");
        // Criar arquivo JUnit
        TestDataFacade.createUnitTestFile(junitFileJavaPath, population);
        // Compila os indivíduos
        this._testGenerationTool.compileIndividuals();
        this.ExecutionCount++;
        GenerationTrace.
write("Individuals compiled with sucess. Execution No. ",
        this.ExecutionCount);
    }
    // Atualizar os indivíduos com os resultados
    // obtidos após a execução dos mesmos
    protected void updateIndividuals() throws TestDataGenerationToolException {
        this.CurrentPopulation = updateIndividuals(this.CurrentPopulation);
    }
    protected Individual[] updateIndividuals(Individual[] population)
throws TestDataGenerationToolException {
        GenerationTrace.write("Updating individuals. Count ",
            population.length, " individuals.");
        if (this._testGenerationTool != null)
        {
            GenerationCoverage[] coverages =
this._testGenerationTool.getCoverages(this._criterion);
            int index = 0;
            for (Individual individual : population) {
                individual.setCoverage(coverages[index]);
                index++;
            }
        }
        return population;
    }
    protected boolean
verifyIfRequirementHasBeenCovered(GenerationRequirement requirement) {
        GenerationTrace.write("Verify if requirement ",
            requirement.getIdRequirement(), " has been covered.");
        currentSolutionToRequirement = null;
        boolean returnValue = false;
        // Varre a população atual identificando se o requisito foi coberto
        for (Individual individual : this.CurrentPopulation) {
            // Captura a cobertura de um indivíduo e verifica se o requisito foi coberto
            if (verifyIndividualRequirementCovered(requirement,
individual.getCoverage()))
            {
                GenerationTrace.write("Target ",
                    requirement.getIdRequirement(), " covered.");
                returnValue = true;
                currentSolutionToRequirement = individual;
                break;
            }
        }
        return returnValue;
    }
    // Verifica se um indivíduo cobre um requisito
    protected boolean verifyIndividualRequirementCovered(GenerationRequirement requirement,
        GenerationCoverage coverage)
    {
        boolean covered = false;

```

```

        if (coverage != null)
        {
            if (coverage.getExecutionPath()
                .contains("-" + requirement.getIdRequirement() + "-"))
                covered = true;
        }
        return covered;
    }
    // Finalizar a geração de dados de teste
    protected abstract void finalizeGeneration();
    // Nome da estratégia de geração de dados de teste
    public abstract String getGenerationStrategyName();
}

```

MetaheuristicBaseGenerationStrategy

```

public abstract class MetaheuristicBaseGenerationStrategy
    extends BaseGenerationStrategy
    implements IStrategyPrimitiveTips,
               IStrategyFitness, IStrategySelection {
    protected InputPrimitiveDomain _inputDomain;
    protected IFitness _fitness;
    protected ISelection _selection;
    protected int _maxNumberOfAttempts = 2;
    // Iniciar a geração automática de dados de teste
    // Iniciar a geração automática de dados de teste
    @Override
    public Individual[] generateData() throws TestStrategyException
    {
        // Zera o contador de iterações
        GenerationTrace.write("Starting test data generation.");
        this.InteractionCount = 0;
        List<Individual> solution = new ArrayList<Individual>();
        // Processo de inicialização
        initializeGeneration();
        try
        {
            // Geração da população inicial de indivíduos
            this.CurrentPopulation = generateInitialPopulation();
            // Executa os indivíduos
            executeIndividuals();
            // Identificação dos ramos/nós que devem ser cobertos
            this.Targets = identifyRequirements();
            // Atualizar os indivíduos
            updateIndividuals();
            for (int i = 0; i < this.Targets.length; i++)
            {
                int attemptNumber = 0;
                this.CurrentTarget = this.Targets[i];
                if (this.CurrentTarget.isCovered())
                    continue;
                do
                {
                    // Itera o contador de iterações
                    this.InteractionCount++;
                    attemptNumber++;
                    if (verifyIfRequirementHasBeenCovered(this.CurrentTarget))
                    {
                        this.CurrentTarget.isCovered(true);
                        // Adiciona o indivíduo que cobriu o requisito como solução
                        solution.add(this.currentSolutionToRequirement);
                    }
                    else
                    {
                        this.CurrentTarget.isCovered(false);
                        // Calcula a fitness dos indivíduos
                        computeFitness(this.CurrentPopulation, CurrentTarget);
                    }
                } while (attemptNumber < this._maxNumberOfAttempts);
            }
        }
        catch (TestStrategyException e)
        {
            // ...
        }
    }
}

```

```

        // Executa mutação sobre a população atual de indivíduos
        this.CurrentPopulation = executeMutation(this.CurrentPopulation);
        // Verificar porque a população atinge o tamanho ZERO
        if (this.CurrentPopulation.length == 0)
            this.CurrentPopulation = generateInitialPopulation();
        // Executar indivíduos
        executeIndividuals();
        // Atualizar os indivíduos
        updateIndividuals();
    }
    }
    while(!this.CurrentTarget.isCovered() &&
        attemptNumber < this._maxNumberOfAttempts);
    }
}
catch(TestDataGenerationStrategyException e)
{
    GenerationTrace.write("TestDataGenerationStrategyException: " +
        e.getMessage());
    throw new TestStrategyException(e);
}
catch(Exception e)
{
    throw new TestStrategyException(e);
}
// Processo de finalização
finalizeGeneration();
return solution.toArray(new Individual[solution.size()]);
}
public void setNumberMaxOfAttempts(int numberMaxOfAttempts)
{
    this._maxNumberOfAttempts = numberMaxOfAttempts;
}
@Override
public void setInputDomain(InputPrimitiveDomain inputDomain) {
    // Atribui os dados primitivos que guiarão a geração dos dados de teste
    this._inputDomain = inputDomain;
}
@Override
protected void finalizeGeneration()
{
    GenerationTrace.write("Finalizing generation.");
}
@Override
protected Individual[] generateInitialPopulation()
    throws TestDataGenerationStrategyException {
    GenerationTrace.write("Generating initial population.");
    // Lê os dados do bytecode
    BytecodeReader reader = new BytecodeReader(this.JarPath, this.ClassUnderTest);
    ReflectionClass rc = reader.getReflectionClass();
    // Gera indivíduos aleatórios
    Individual[] individuals =
        RandomValueFactory.getNewPopulation(this.PopulationSize,
                                            rc,
                                            this.MethodUnderTest,
                                            this._inputDomain);

    return individuals;
}
@Override
protected void initializeGeneration()
{
    GenerationTrace.write("Initializing generation.");
}
@Override
public void setFitness(IFitness fitness) {
    this._fitness = fitness;
}
@Override
public IFitness getFitness() {
    return this._fitness;
}
@Override

```

```

public void setSelectionMode(ISelection selection) {
    this._selection = selection;
}
@Override
public ISelection getSelectionMode() {
    return this._selection;
}
protected Individual[] executeMutation(Individual[] population)
                                     throws MutationException
{
    GenerationTrace.write("Stating mutation.");
    // Filtra a população
    if (this._selection != null)
        population = this._selection
            .selectIndividualsToMutation(population);
    if (population.length == 0)
    {
        return population;
    }
    // Cria maquina de mutacao
    MutationMachine mm = new MutationMachine(population);
    // Atribui dados de geracao aleatoria dirigida
    mm.setInputDomain(this._inputDomain);
    // Executa mutacao dos individuos
    Individual[] returnValue = mm.mutate();
    // Retorna a nova populacao
    return returnValue;
}
protected void computeFitness(Individual[] individuals,
                              GenerationRequirement requirement) throws FitnessException
{
    for (Individual individual : individuals) {
        double fitness = 0;
        fitness = computeFitness(individual, requirement);
        individual.setFitness(fitness);
    }
}
protected double computeFitness(Individual individual,
                                GenerationRequirement target) throws FitnessException
{
    double fitness = 0;
    GenerationCoverage currentCoverage = null;
    currentCoverage = individual.getCoverage();
    // Atribuir os requisitos a fitness
    this._fitness.setRequirements(this.Targets);
    // Atribuir a população atual a fitness
    this._fitness.setCurrentPopulation(this.CurrentPopulation);
    // Calcula a fitness do individuo
    fitness = this._fitness.computeFitness(target, currentCoverage);
    return fitness;
}
@Override
public abstract String getGenerationStrategyName();
}

```

APÊNDICE B

Geração aleatória

```

public class RandomGenerationStrategy extends BaseGenerationStrategy
                                     implements IStrategyPrimitiveTips,
                                     IGUIGenerationStrategy{

    private InputPrimitiveDomain _inputDomain;
    @Override
    public GUIGenerationCustomParameter[] getCustomParameters() {
        return null;
    }
    @Override
    public EnumCriterion[] getSupportedCriteriaions() {
        EnumCriterion[] criterions =
            new EnumCriterion[]{ EnumCriterion.AllNodesEi,
                                EnumCriterion.AllEdgesEi,
                                EnumCriterion.AllPotUsesEi,
                                EnumCriterion.AllUsesEi,
                                EnumCriterion.AllNodesEd,
                                EnumCriterion.AllEdgesEd,
                                EnumCriterion.AllPotUsesEd,
                                EnumCriterion.AllUsesEd };

        return criterions;
    }
    @Override
    public IFitness[] getSupportedFitnesses() {
        return null;
    }
    @Override
    public ISelection[] getSupportedSelections() {
        return null;
    }
    @Override
    public void setInputDomain(InputPrimitiveDomain inputDomain) {
        // Atribui os dados primitivos que guiarão a geração dos dados de teste
        this._inputDomain = inputDomain;
    }
    @Override
    protected void finalizeGeneration() {
    }
    @Override
    protected Individual[] generateInitialPopulation()
    {
        // Lê os dados do bytecode
        BytecodeReader reader = new BytecodeReader(this.JarPath,
                                                    this.ClassUnderTest);

        ReflectionClass rc = reader.getReflectionClass();
        // Gera indivíduos aleatórios
        Individual[] individuals = RandomValueFactory
                                    .getNewPopulation(this.PopulationSize,
                                                    rc,
                                                    this.MethodUnderTest,
                                                    this._inputDomain);

        return individuals;
    }
    @Override
    protected void initializeGeneration() {

```

```
    }  
    @Override  
    public String getGenerationStrategyName() {  
        return "Random strategy";  
    }  
}
```


APÊNDICE C

Algoritmo evolucionário

```

public class EvolutionaryGenerationStrategy
    extends MetaheuristicBaseGenerationStrategy
    implements IGUIGenerationStrategy
{
    @Override
    public GUIGenerationCustomParameter[] getCustomParameters() {
        return null;
    }
    @Override
    public EnumCriterion[] getSupportedCriteriaions() {
        EnumCriterion[] criterions =
            new EnumCriterion[]{ EnumCriterion.AllNodesEi,
                                EnumCriterion.AllEdgesEi,
                                EnumCriterion.AllPotUsesEi,
                                EnumCriterion.AllUsesEi,
                                EnumCriterion.AllNodesEd,
                                EnumCriterion.AllEdgesEd,
                                EnumCriterion.AllPotUsesEd,
                                EnumCriterion.AllUsesEd };

        return criterions;
    }
    @Override
    public IFitness[] getSupportedFitnesses() {
        return new IFitness[] { new SimilarityFitness(),
                                new IneditismFitness() };
    }
    @Override
    public ISelection[] getSupportedSelections() {
        return new ISelection[] { new ElitismSelection(),
                                new TournamentSelection(),
                                new AverageSelection(),
                                new RouletteSelection()};
    }
    @Override
    public String getGenerationStrategyName()
    {
        return "Evolutionary strategy";
    }
}

```

APÊNDICE D

Funções de aptidão

Similaridade

```

public class SimilarityFitness implements IFitness {
    @Override
    public double computeFitness(GenerationRequirement requirement,
                                GenerationCoverage coverage) throws FitnessException {
        double[] fitnesses = null;
        String[] path = null;
        // Captura os requisitos necessarios para atingir o alvo
        path = requirement.getCoveragePaths();
        fitnesses = new double[path.length];
        for (int i = 0; i < path.length; i++) {
            // Calcula a fitness de cada possivel caminho para cobrir um requisito
            fitnesses[i] = getFitness(path[i], coverage.getExecutionPath());
            // Se um dos requisitos for igual a 1, então já retorna este valor e
            // não precisa executar os próximos passos
            if (fitnesses[i] == 1)
                return 1;
        }
        if (fitnesses.length == 1)
            return fitnesses[0];
        // Identifica qual a melhor fitness para atingir um item
        double maxFitness = 0;
        for (double fitness : fitnesses) {
            if (fitness > maxFitness)
                maxFitness = fitness;
        }
        return maxFitness;
    }
    private double getFitness(String target, String realized) {
        if (realized.contains(target))
            return 1;
        // Requisitos que devem ser cobertos
        String[] requirementsTarget = getRequirements(target);
        // Requisitos que foram cobertos
        String[] requirementsRealized = getRequirements(realized);
        // Esta variável acumula a quantidade de itens que pertencem
        // ao caminho desejado e que foram cobertos
        double countOfCorrectRequirements = 0;
        // Converte o array de alvos para LIST, por que este objeto da suporte
        // ao método CONTAINS.
        List<String> realizedList = Arrays.asList(requirementsRealized);
        // Identifica quantos nós que deveriam ser cobertos foram executados.
        for (String requirement : requirementsTarget) {
            if (realizedList.contains(requirement))
                countOfCorrectRequirements++;
        }
        // Divide a quantidade de itens cobertos pela quantidade de itens necessária
    }
}

```

```

        double returnValue = countOfCorrectRequirements /
                               (requirementsTarget.length * 1.0);
        if (returnValue == 1)
            return 1;
        return returnValue;
    }
    private String[] getRequirements(String requirement) {
        // Remoção do primeiro hifen
        if (requirement.startsWith("-"))
            requirement = requirement.substring(1);
        // Remoção do ultimo hifen
        if (requirement.endsWith("-"))
            requirement = requirement.substring(0, requirement.length() - 1);
        String[] requirementsTarget = requirement.split("-");
        return requirementsTarget;
    }
    @Override
    public String getFitnessName() {
        return "Similarity Fitness";
    }
    @Override
    public void setCurrentPopulation(Individual[] individuals) {
    }
}

```

Ineditismo

```

public class IneditismFitness implements IFitness {
    private Individual[] _currentPopulation;
    @Override
    public double computeFitness(GenerationRequirement requirement,
        GenerationCoverage coverage) throws FitnessException {
        double[] fitnesses = null;
        String[] path = null;
        // Captura os requisitos necessarios para atingir o alvo
        path = requirement.getCoveragePaths();
        fitnesses = new double[path.length];
        for (int i = 0; i < path.length; i++) {
            // Calcula a fitness de cada possivel caminho para cobrir um requisito
            fitnesses[i] = getFitness(path[i], coverage.getExecutionPath());
            // Se um dos requisitos for igual a 1, então já retorna este valor e
            // não precisa executar os próximos passos
            if (fitnesses[i] == 1)
                return 1;
        }
        // Calcula o bonus do indivíduo
        double bonus = getBonus(coverage);
        double maxFitness = 0;
        if (fitnesses.length == 1)
        {
            maxFitness = fitnesses[0];
        }
        else
        {
            // Identifica qual a melhor fitness para atingir um item
            for (double fitness : fitnesses) {
                if (fitness > maxFitness)
                    maxFitness = fitness;
            }
        }
        // Verifica se a fitness não irá estourar o limite de 1.0
        if ((maxFitness + bonus) < 1.0)
            maxFitness += bonus;
        return maxFitness;
    }
    private double getBonus(GenerationCoverage coverage)
    {
        double returnValue = 0;
    }
}

```

```

int countOfOriginalRequirements = 0;
if (this._currentPopulation == null)
    return returnValue;
if (this._currentPopulation.length == 0)
    return returnValue;
// Requisitos que devem ser cobertos
String[] requirementsTarget = getRequirements(coverage.getExecutionPath());
// Navega entre os requisitos validando se ele é inédito ou não
for (String requirementCovered : requirementsTarget) {
    boolean isOriginal = true;
    // Verifica indivíduo por indivíduo da população,
    // se o requisito atual é inédito
    for (Individual individual : this._currentPopulation) {
        if (coverage == individual.getCoverage())
            continue;
        // Verifica se o indivíduo cobre o requisito atual
        if (verifyIfIndividualContainsRequirementTo(requirementCovered,
            individual.getCoverage()))
        {
            // Marca o indivíduo como "NÃO INÉDITO"
            isOriginal = false;
            break;
        }
    }
    // Verifica se o indivíduo é inédito
    if (isOriginal)
    {
        countOfOriginalRequirements++;
    }
}
if (countOfOriginalRequirements > 0)
{
    returnValue = (1 / (this._currentPopulation.length * 1.0));
    returnValue *= countOfOriginalRequirements;
}
return returnValue;
}
private boolean verifyIfIndividualContainsRequirementTo(String requirement,
    GenerationCoverage coverage)
{
    String individualCoverage = coverage.getExecutionPath();
    if (individualCoverage.contains("-" + requirement + "-"))
        return true;
    return false;
}
private double getFitness(String target, String realized)
{
    if (realized.contains(target))
        return 1;
    // Requisitos que devem ser cobertos
    String[] requirementsTarget = getRequirements(target);
    // Requisitos que foram cobertos
    String[] requirementsRealized = getRequirements(realized);
    // Esta variável acumula a quantidade de itens que pertencem
    // ao caminho desejado e que foram cobertos
    double countOfCorrectRequirements = 0;
    // Converte o array de alvos para LIST, por que este objeto da suporte
    // ao método CONTAINS.
    List<String> realizedList = Arrays.asList(requirementsRealized);
    // Identifica quantos nós que deveriam ser cobertos foram executados.
    for (String requirement : requirementsTarget) {
        if (realizedList.contains(requirement))
            countOfCorrectRequirements++;
    }
    // Divide a quantidade de itens cobertos pela quantidade de itens necessária
    double returnValue = countOfCorrectRequirements /
        (requirementsTarget.length * 1.0);
    if (returnValue == 1)
        return 1;
    return returnValue;
}

```

```
}
private String[] getRequirements(String requirement)
{
    // Remoção do primeiro hifen
    if (requirement.startsWith("-"))
        requirement = requirement.substring(1);
    // Remoção do ultimo hifen
    if (requirement.endsWith("-"))
        requirement = requirement.substring(0, requirement.length() - 1);
    String[] requirementsTarget = requirement.split("-");
    return requirementsTarget;
}
@Override
public String getFitnessName() {
    return "Ineditism Fitness";
}
@Override
public void setCurrentPopulation(Individual[] individuals) {
    this._currentPopulation = individuals;
}
}
```

APÊNDICE E

Algoritmos de seleção

Elitismo

```
public class ElitismSelection implements ISelection {
    @Override
    public Individual[] selectIndividualsToMutation(Individual[] population) {
        int quantityOfSelecteds = population.length / 2;
        Arrays.sort(population);
        Individual[] newPopulation = new Individual[quantityOfSelecteds];
        for (int i = 0; i < quantityOfSelecteds; i++) {
            newPopulation[i] = population[i];
        }
        return newPopulation;
    }
    @Override
    public String getSelectionName() {
        return "Elitism Selection";
    }
}
```

Torneio

```
public class TournamentSelection implements ISelection {
    @Override
    public Individual[] selectIndividualsToMutation(Individual[] population) {
        // Seleção por torneio, este critério de seleção faz comparações entre
        // os indivíduos, a fim de selecionar os melhores da população.
        int quantityOfSelecteds = population.length / 2;
        Individual[] newPopulation = new Individual[quantityOfSelecteds];
        for (int i = 0; i < quantityOfSelecteds; i++) {
            Random rand = new Random();
            int tournament01 = 0;
            int tournament02 = 0;
            do {
                tournament01 = rand.nextInt(population.length - 1);
                tournament02 = rand.nextInt(population.length - 1);
            } while ((tournament01 == tournament02) ||
                (tournament01 < 0) || (tournament02 < 0));
            Individual individual01 = population[tournament01];
            Individual individual02 = population[tournament02];
            if (individual02.getFitness() > individual01.getFitness())
                newPopulation[i] = individual02;
            else
                newPopulation[i] = individual01;
        }
    }
}
```

```

        return newPopulation;
    }
    @Override
    public String getSelectionName()
    {
        return "Tournament Selection";
    }
}

```

Roleta

```

public class RouletteSelection implements ISelection {
    @Override
    public Individual[] selectIndividualsToMutation(Individual[] population) {
        double sumFitness = 0;
        int quantityOfSelecteds = population.length / 2;
        Individual[] newPopulation = new Individual[quantityOfSelecteds];
        for (Individual individual : population) {
            sumFitness += individual.getFitness();
        }
        for (int i = 0; i < quantityOfSelecteds; i++) {
            Individual selected = null;
            Random rand = new Random();
            double rouletteValue = (double) (rand.nextDouble() * sumFitness);
            double rouletteSum = 0;
            for (Individual individual : population) {
                rouletteSum += individual.getFitness();
                if (rouletteSum >= rouletteValue)
                {
                    selected = individual;
                    break;
                }
            }
            newPopulation[i] = selected;
        }
        return newPopulation;
    }
    @Override
    public String getSelectionName() {
        return "Roulette Selection";
    }
}

```

Média

```

public class AverageSelection implements ISelection {
    @Override
    public Individual[] selectIndividualsToMutation(Individual[] population) {
        // Captura a fitness média
        double average = getAverage(population);
        // Retorna os indivíduos acima ou igual a média
        return getIndividualsAboveTheAverage(population, average);
    }
    // Retorna apenas os indivíduos acima da média
    private Individual[] getIndividualsAboveTheAverage(Individual[] population,
        double average) {
        List<Individual> resultIndividuals = new ArrayList<Individual>();
        for (Individual individual : population) {
            if (individual.getFitness() >= average)
                resultIndividuals.add(individual);
        }
        return resultIndividuals.toArray(new Individual[resultIndividuals.size()]);
    }
    // Identifica a média da fitness dos indivíduos da população
}

```

```
private double getAverage(Individual[] population) {
    double average = 0;
    double sum = 0;
    for (Individual individual : population) {
        sum += individual.getFitness();
    }
    if (population.length > 0)
        average = sum / (population.length * 1.0);
    return average;
}
@Override
public String getSelectionName() {
    return "Average Selection";
}
}
```


APÊNDICE F

Subida de encosta

```

public class HillClimbingGenerationStrategy
    extends MetaheuristicBaseGenerationStrategy
    implements IGUIGenerationStrategy
{
    @Override
    public GUIGenerationCustomParameter[] getCustomParameters() {
        return null;
    }
    @Override
    public EnumCriterion[] getSupportedCriteriaions() {
        EnumCriterion[] criterions =
            new EnumCriterion[]{ EnumCriterion.AllNodesEi,
                                EnumCriterion.AllEdgesEi,
                                EnumCriterion.AllPotUsesEi,
                                EnumCriterion.AllUsesEi,
                                EnumCriterion.AllNodesEd,
                                EnumCriterion.AllEdgesEd,
                                EnumCriterion.AllPotUsesEd,
                                EnumCriterion.AllUsesEd };

        return criterions;
    }
    @Override
    public IFitness[] getSupportedFitnesses() {
        return new IFitness[] { new SimilarityFitness(),
                                new IneditismFitness() };
    }
    @Override
    public ISelection[] getSupportedSelections() {
        return new ISelection[] { new ElitismSelection(),
                                new TournamentSelection(),
                                new AverageSelection(),
                                new RouletteSelection()};
    }
    // Iniciar a geração automática de dados de teste
    @Override
    public Individual[] generateData() throws TestStrategyException
    {
        GenerationTrace.write("Starting test data generation.");
        this.InteractionCount = 0;
        List<Individual> solution = new ArrayList<Individual>();
        // Processo de inicialização
        GenerationTrace.write("Initializing generation.");
        initializeGeneration();
        try {
            // Geração da população inicial de indivíduos
            GenerationTrace.write("Generating initial population.");
            this.CurrentPopulation = generateInitialPopulation();
            // Executa os indivíduos
            executeIndividuals();
            // Identificação dos ramos/nós que devem ser cobertos
            this.Targets = identifyRequirements();
            // Atualizar os indivíduos
            updateIndividuals();
        }
    }
}

```

```

    for (int i = 0; i < this.Targets.length; i++) {
        int attemptNumber = 0;
        this.CurrentTarget = this.Targets[i];
        do {
            // Itera o contador de iterações
            this.InteractionCount++;
            attemptNumber++;
            if (verifyIfRequirementHasBeenCovered(this.CurrentTarget)) {
                this.CurrentTarget.isCovered(true);
                // Adiciona o indivíduo que cobriu
                // o requisito como solução
                solution.add(this.currentSolutionToRequirement);
            }
            else {
                this.CurrentTarget.isCovered(false);
                // Calcula a fitness dos indivíduos
                computeFitness(this.CurrentPopulation, CurrentTarget);
                // Bloco de execução do Hill Climbing.
                GenerationTrace.write("Starting Hill Climbing.");
                this.CurrentPopulation =
                    executeHillClimbing(this.CurrentPopulation,
                                        CurrentTarget);
                // Verifica se o requisito foi coberto
                if (verifyIfRequirementHasBeenCovered(this.CurrentTarget))
                {
                    this.CurrentTarget.isCovered(true);
                    // Adiciona o indivíduo que cobriu o requisito como solução
                    solution.add(this.currentSolutionToRequirement);
                }
                else
                {
                    // Executa mutação sobre a população atual de indivíduos
                    this.CurrentPopulation =
                        executeMutation(this.CurrentPopulation);
                    // Executar indivíduos
                    executeIndividuals();
                    // Atualizar os indivíduos
                    updateIndividuals();
                }
            }
        }
        while(!this.CurrentTarget.isCovered()
            && attemptNumber < this._maxNumberOfAttempts);
    }
}
catch(TestDataGenerationToolException e)
{
    GenerationTrace.write("TestDataGenerationToolException: " + e.getMessage());
    throw new TestStrategyException(e);
}
catch(Exception e)
{
    throw new TestStrategyException(e);
}
// Processo de finalização
finalizeGeneration();
return solution.toArray(new Individual[solution.size()]);
}
private Individual changeIndividual(Individual individual)
{
    InputValueMutation mutationExecutor = new InputValueMutation();
    mutationExecutor.setInputDomain(this._inputDomain);
    Individual result = mutationExecutor.mutate(individual);
    return result;
}
@Override
public String getGenerationStrategyName()
{
    return "Hill climbing strategy";
}

```

```
private Individual[] executeHillClimbing(Individual[] population,
                                         GenerationRequirement currentRequirement)
    throws TestDataGenerationToolException, FitnessException
{
    if (population == null || currentRequirement == null)
        return null;
    for (int i = 0; i < population.length; i++) {
        // Itera o contador de iterações
        this.InteractionCount++;
        // Identifica o indivíduo dentro da população
        Individual individual = population[i];
        // Modifica o individuo gerando algo novo
        Individual newIndividual = changeIndividual(individual);
        // Cria novo array com um indivíduo
        Individual[] newPopulation = new Individual[] { newIndividual };
        // Executa os indivíduos
        executeIndividuals(newPopulation);
        // Atualiza os indivíduos com suas coberturas
        newPopulation = updateIndividuals(newPopulation);
        // Atualiza a variável local com os dados da execução e da cobertura
        newIndividual = newPopulation[0];
        // Calcula a fitness do indivíduo
        double fitness = computeFitness(newIndividual, currentRequirement);
        // Atribui a fitness ao indivíduo
        newIndividual.setFitness(fitness);
        // Verificar se a fitness do novo indivíduo é melhor
        if (individual.getFitness() < newIndividual.getFitness())
            population[i] = newIndividual;
    }
    return population;
}
```

APÊNDICE G

Têmpera simulada

```

public class SimulatedAnnealingGenerationStrategy
    extends MetaheuristicBaseGenerationStrategy
    implements IGUIGenerationStrategy
{
    private int _maxAttempts;
    private double _initialTemperature;
    @Override
    public GUIGenerationCustomParameter[] getCustomParameters() {
        GUIGenerationCustomParameter paramMaxAttempts =
            new GUIGenerationCustomParameter();
        paramMaxAttempts.setDefaultValue(15);
        paramMaxAttempts.setLegend("Max Attempts: ");
        paramMaxAttempts.setType(PrimitiveType.Integer);
        paramMaxAttempts.setSetterMethodSignature("setMaxAttempts(int)");
        GUIGenerationCustomParameter paramInitialTemperature =
            new GUIGenerationCustomParameter();
        paramInitialTemperature.setDefaultValue(5.0);
        paramInitialTemperature.setLegend("Initial Temperature: ");
        paramInitialTemperature.setType(PrimitiveType.Double);
        paramInitialTemperature.setSetterMethodSignature("setInitialTemperature(double)");
        GUIGenerationCustomParameter[] parameters =
            new GUIGenerationCustomParameter[]
            {
                paramMaxAttempts,
                paramInitialTemperature
            };
        return parameters;
    }
    @Override
    public EnumCriterion[] getSupportedCriteriaions() {
        EnumCriterion[] criterions =
            new EnumCriterion[] { EnumCriterion.AllNodesEi,
                                  EnumCriterion.AllEdgesEi,
                                  EnumCriterion.AllPotUsesEi,
                                  EnumCriterion.AllUsesEi,
                                  EnumCriterion.AllNodesEd,
                                  EnumCriterion.AllEdgesEd,
                                  EnumCriterion.AllPotUsesEd,
                                  EnumCriterion.AllUsesEd };
        return criterions;
    }
    @Override
    public IFitness[] getSupportedFitnesses() {
        return new IFitness[] { new SimilarityFitness(),
                                new IneditismFitness() };
    }
    @Override
    public ISelection[] getSupportedSelections() {
        return new ISelection[] { new ElitismSelection(),
                                   new TournamentSelection(),
                                   new AverageSelection(),
                                   new RouletteSelection() };
    }
    public void setMaxAttempts(int value) {

```

```

        this._maxAttempts = value;
    }
    public void setInitialTemperature(double value) {
        this._initialTemperature = value;
    }
    // Iniciar a geração automática de dados de teste
    @Override
    public Individual[] generateData() throws TestStrategyException {
        // Zera o contador de iterações
        GenerationTrace.write("Starting test data generation.");
        this.InteractionCount = 0;
        List<Individual> solution = new ArrayList<Individual>();
        // Processo de inicialização
        GenerationTrace.write("Initializing generation.");
        initializeGeneration();
        try
        {
            // Geração da população inicial de indivíduos
            GenerationTrace.write("Generating initial population.");
            this.CurrentPopulation = generateInitialPopulation();
            // Executa os indivíduos
            executeIndividuals();
            // Identificação dos ramos/nós que devem ser cobertos
            this.Targets = identifyRequirements();
            // Atualizar os indivíduos
            updateIndividuals();
            for (int i = 0; i < this.Targets.length; i++) {
                int attemptNumber = 0;
                this.CurrentTarget = this.Targets[i];
                do {
                    // Itera o contador de iterações
                    this.InteractionCount++;
                    attemptNumber++;
                    if (verifyIfRequirementHasBeenCovered(this.CurrentTarget))
                    {
                        this.CurrentTarget.isCovered(true);
                        // Adiciona o indivíduo que cobriu o requisito como solução
                        solution.add(this.currentSolutionToRequirement);
                    }
                    else
                    {
                        this.CurrentTarget.isCovered(false);
                        // Calcula a fitness dos indivíduos
                        computeFitness(this.CurrentPopulation, CurrentTarget);
                        // Bloco de execução do Hill Climbing.
                        GenerationTrace.write("Starting Simulated Annealing.");
                        this.CurrentPopulation =
                            executeSimulatedAnnealing(this.CurrentPopulation, CurrentTarget);
                        // Verifica se o requisito foi coberto
                        if (verifyIfRequirementHasBeenCovered(this.CurrentTarget))
                        {
                            this.CurrentTarget.isCovered(true);
                            // Adiciona o indivíduo que cobriu o requisito como solução
                            solution.add(this.currentSolutionToRequirement);
                        }
                        else
                        {
                            // Executa mutação sobre a população atual de indivíduos
                            this.CurrentPopulation = executeMutation(this.CurrentPopulation);
                            // Executar indivíduos
                            executeIndividuals();
                            // Atualizar os indivíduos
                            updateIndividuals();
                        }
                    }
                }
                while(!this.CurrentTarget.isCovered()
                    && attemptNumber < this._maxNumberOfAttempts);
            }
        }
        catch(TestDataGenerationToolException e) {

```

```

        GenerationTrace.write("TestDataGenerationToolException: " + e.getMessage());
        throw new TestStrategyException(e);
    }
    catch(Exception e) {
        throw new TestStrategyException(e);
    }
    // Processo de finalização
    finalizeGeneration();
    return solution.toArray(new Individual[solution.size()]);
}
private Individual changeIndividual(Individual individual) {
    InputValueMutation mutationExecutor = new InputValueMutation();
    mutationExecutor.setInputDomain(this._inputDomain);
    Individual result = mutationExecutor.mutate(individual);
    return result;
}
@Override
public String getGenerationStrategyName() {
    return "Simulated annealing strategy";
}
private double getEnergy(Individual individual) {
    double energy = 0;
    energy = individual.getFitness();
    return energy;
}
private Individual[] executeSimulatedAnnealing(Individual[] population,
        GenerationRequirement currentRequirement)
        throws TestDataGenerationToolException, FitnessException {
    int MAXATTEMPTS = this._maxAttempts;
    double INITIALTEMPERATURE = this._initialTemperature;
    if (population == null || currentRequirement == null)
        return null;
    if (population.length == 0)
        return null;
    for (int i = 0; i < population.length; i++) {
        // Inicializa com a temperatura inicial
        double currentTemperature = INITIALTEMPERATURE;
        int index = 0;
        while(currentTemperature > 0 && index < MAXATTEMPTS)
        {
            try
            {
                // Identifica o indivíduo dentro da população
                Individual individual = population[i];
                // Modifica o indivíduo gerando algo novo
                Individual newIndividual = changeIndividual(individual);
                // Cria novo array com um indivíduo
                Individual[] newPopulation = new Individual[] { newIndividual };
                // Executa os indivíduos
                executeIndividuals(newPopulation);
                // Atualiza os indivíduos com suas coberturas
                newPopulation = updateIndividuals(newPopulation);
                // Atualiza a variável local com os dados da execução e da cobertura
                newIndividual = newPopulation[0];
                // Calcula a fitness do indivíduo
                double fitness = computeFitness(newIndividual, currentRequirement);
                // Atribui a fitness ao indivíduo
                newIndividual.setFitness(fitness);
                // Se a fitness do novo indivíduo é melhor, então ele
                // substituirá o indivíduo antigo
                if (individual.getFitness() < newIndividual.getFitness())
                {
                    population[i] = newIndividual;
                    if (newIndividual.getFitness() == 1)
                        return population;
                }
                // Verifica a energia gasta com o novo indivíduo
                double energy = getEnergy(newIndividual);
                // Atualiza a temperatura
            }
            catch (Exception e) {
                // Ignora exceções durante a execução
            }
        }
    }
}

```

```
        currentTemperature -= energy;
        // Acrescenta uma interação
        index++;
    } catch (Exception ex)
    {
        String message = ex.getMessage();
    }
}
// Itera o contador de interações
this.InteractionCount++;
}
return population;
}
}
```

APÊNDICE H

Benchmarks

Insertion Sort

```
public void insertionSort( int[ ] a )
{
    int j;
    for( int p = 1; p < a.length; p++ )
    {
        int tmp = a[ p ];
        for( j = p; j > 0 && tmp<a[ j - 1 ]; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Quick Sort

```
public static int median3( int[ ] a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ]<a[ left ] )
        swap( a, left, center );
    if( a[ right ] < a[ left ] )
        swap( a, left, right );
    if( a[ right ] < a[ center ] )
        swap( a, center, right );
    // Place pivot at position right - 1
    swap( a, center, right - 1 );
    return a[ right - 1 ];
}

public static void quicksort( int[ ] a, int left, int right)
{
    if( left + CUTOFF <= right )
    {
        int pivot = median3( a, left, right );
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[ ++i ] < pivot ) { }
            while( a[ --j ] > pivot ) { }
            if( i < j )
                swap( a, i, j );
            else
                break;
        }
    }
}
```



```

        swap( a, i, right - 1 );
        quicksort( a, left, i - 1 );
        quicksort( a, i + 1, right );
    }
    else
        insertionSort( a, left, right );
}

```

Merge Sort

```

public void mergeSort( int[ ] a )
{
    int [ ] tmpArray = new int[ a.length ];
    mergeSort( a, tmpArray, 0, a.length - 1 );
}
public static void mergeSort( int[ ] a, int[ ] tmpArray,
                              int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
public static void merge( int[ ] a, int[ ] tmpArray,
                          int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ] <= a[ rightPos ] )
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];
    while( leftPos <= leftEnd )
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];
    while( rightPos <= rightEnd )
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}

```

TryTip

```

public String tryTip(int s1, int s2, int s3)
{
    if (s1 == s2 && s2 == s3)
        return "equilateral";
    if (s1 == s2 && s2 != s3)
        return "isosceles";
    if (s1 != s2 && s2 == s3)
        return "Isosceles";
    return "scalene";
}

```