Testes unitários evolutivos

Nuno Alexandre Palma

Dissertação submetida para obtenção do grau de
MESTRE EM ENGENHARIA INFORMÁTICA
pela
Universidade de Évora
Departamento de informática

Orientador: Luís Arriaga da Cunha

Outubro de 2007

Testes unitários evolutivos

Nuno Alexandre Palma

Dissertação submetida para obtenção do grau de
MESTRE EM ENGENHARIA INFORMÁTICA
pela
Universidade de Évora
Departamento de informática

Orientador: Luís Arriaga da Cunha

"Esta dissertação não inclui as críticas e sugestões feitas pelo júri."

Outubro de 2007

Resumo

Testes unitários evolutivos

A fase de teste representa, cada vez mais, um papel importante no processo de desenvolvimento de uma aplicação. Esta torna-se ainda mais relevante quando se adoptam metodologias de desenvolvimento nas quais os testes unitários têm um papel crucial no sucesso dos mesmos.

As ferramentas actuais para testes unitários implicam a geração e execução dos mesmos pelo programador. Estas tarefas são lentas e repetitivas, levando a uma saturação por parte do programador e consequentemente à degradação da qualidade do software.

Estudos indicam que 50% do tempo dispensado no desenvolvimento de aplicações está relacionado com a fase de testes. Torna-se então imperativo criar mecanismos de geração e execução automática de testes unitários, que por um lado libertem o programador destas tarefas e outro lado aumente a qualidade da fase de testes.

O trabalho a realizar no âmbito desta tese de mestrado, visa implementar uma ferramenta que permita a automatização da geração e execução de testes unitários para aplicações desenvolvidas na linguagem Java de forma dinâmica e inteligente, tentando gerar testes mais fiáveis.

Abstract

Evolutionary Unit Testing

The test phase plays an important role in the software application development process, furthermore when development methodologies (e.g. Extreme Programming) are adopt, in which unit tests are paramount for their success.

Existing tools for performing unit testing demand oblige programmers to design, implement and execute the tests. These tasks are time consuming and repetitive, and therefore programmers tend to neglect them. This degrades, inevitably, the quality of the software.

Studies indicate that about 50% of the application development time is, directly or indirectly, related to testing.

Thus, it is crucial to develop mechanisms to automate the generation of unit tests, which, on one hand, free the programmers from those tasks, and, on the other hand, increase the quality of the tests. The goal of the work being developed in the context of this Master Thesis is to implement a tool enabling automatic unit tests generation, for Java-based applications, and that, unlike existing tools, test generation is performed in a dynamic and intelligent way, increasing reliability of the tests.

agradecimentos / acknowledgements

Quero dedicar este trabalho a quatro pessoas muito especiais na minha vida (a ordem nada interessa): Manuel, Natividade, Marco e Joana. Sem a vossa ajuda, paciência e persistência teria sido impossível ter chegado onde cheguei. Muito obrigado.

Esta tese de mestrado não teria sido possível sem a ajuda e o apoio de algumas pessoas. Reservo este espaço para escrever algumas palavras de agradecimento àqueles que a tornaram possível.

Quero agradecer em primeiro lugar ao professor Arriaga da Cunha por ter aceite o meu convite para orientador desta dissertação. Os seus conselhos e opiniões foram muito importantes na elaboração desta tese.

Um obrigado ao pessoal da equipa de desenvolvimento da Saphety (Costa, Mesquita, Galvão, Ruben) pela paciência e terem conseguido dar conta do recado quando eu não estava.

Obrigado Artur pelos "então, já chegamos ?" que foram cruciais quando a motivação e a força para acabar esta tese pareciam estar esgotadas.

Por último (mas os últimos são sempre os primeiros), Joana, obrigado por tudo.

Conteúdo

Lista de Tabelas

Lista de Figuras

Listas

Capítulo 1

Introdução

1.1 Motivação

Esta tese teve como origem a experiência como programador adquirida ao logo de quatro anos em projectos na área da segurança electrónica.

Os testes unitários são uma fase crucial no sucesso da fase de desenvolvimento de qualquer software, tendo como principal função validar se os vários componentes implementam de forma correcta e isolada os requisitos para os quais foram programados, fornecendo também uma medida de qualidade do mesmo.

Desenvolver software com qualidade é hoje um dos maiores desafios da indústria de software. Este está cada vez mais complexo e possui uma importância cada vez maior no dia a dia das pessoas em que erros encontrados em sistemas em produção possuem um impacto cada vez maior. Nos Estados Unidos, foram estimadas perdas no valor de 59,500,000,000 milhões de dólares no ano de 2002 devido a erros de software [oCNIoST02].

Devido à complexidade dos sistemas desenvolvidos, é praticamente impossível realizar testes unitários em todo o código. Na figura 1.1 encontra-se exemplificado um exemplo de uma janela de configuração de um programa. É uma janela simples, com nove caixas de selecção e uma caixa de escolha com 7 opções. Para testar esta janela com todas as combinações possíveis e demorando em média um minuto por cada teste, são necessários cerca de $7 * 2^9$ minutos para realizar os testes possíveis, equivalente a um dia de trabalho de uma pessoa.

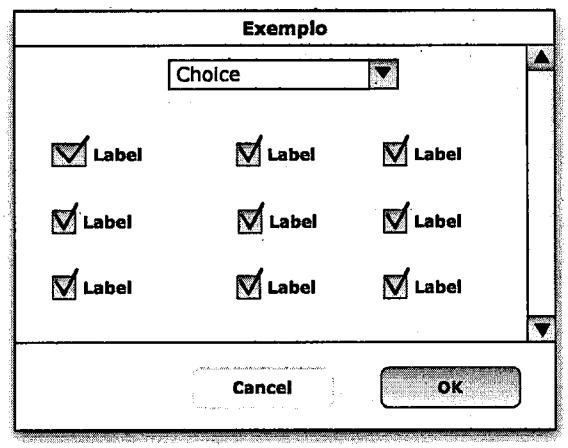

Figura 1.1: Exemplo de uma janela de configuração

Actualmente, o grande problema dos testes unitários é o facto das tecnologias adjacentes aos processos de teste (analise, implementação, execução e geração de relatórios) não conseguiram acompanhar a evolução dos sistemas de software. Em média, 50% do tempo de duração de um projecto de desenvolvimento de software é utilizado para testes [GBP03], o que torna a tarefa muito dispendiosa.

Outro problema da fase de testes é quando esta é executada. Na maioria dos projectos, os testes são realizados no fim, fazendo com que desvios das fases anteriores sejam compensados retirando tempo à fase de testes.

Outro facto interessante é que a maior parte dos programadores preferem escrever código do que escrever sobre o código. Os testes unitários são associados à fase de desenvolvimento e quando estes são realizados pelos próprios programadores, são normalmente bem comportados, servindo apenas para o programador validar que o seu raciocínio está correcto.

1.2 Objectivo

Pretende-se com este trabalho criar uma ferramenta de geração automática de testes unitários para software desenvolvido em Java, recorrendo ao uso de três tecnologias: **Programação por Contrato, Técnicas de Optimização Baseadas em Algoritmos Naturais** e **Testes por Análise de Mutantes**

Esta automatização terá como objectivo três pontos:

- Diminuir a necessidade de intervenção do programador nestas fases;
- Aumentar a qualidade dos testes unitários gerados;
- Ganhar confiança no código produzido;

1.3 Evolução dos testes de software

Confiança e disponibilidade sempre foram características desejadas nos sistemas informáticos. Desde a existência dos primeiros computadores, é notável como os componentes de hardware cresceram em confiança. No entanto, o software está cada vez mais complexo e apresenta cada vez mais problemas. Infelizmente só com a confiança do hardware não se consegue garantir a qualidade desejada nos sistemas informáticos.

A maioria dos métodos formais e métricas de teste foram definidos nas décadas de 70 e 80, era dos mainframes dominada pela IBM. Os computadores eram muito caros e desenhados para durar muito tempo. Era perfeitamente expectável o hardware durar dez anos e o software sensivelmente metade, idades que coincidiam em média com a depreciação do material. Devido ao elevado preço do software e do hardware, as decisões de implementação de sistemas informáticos eram efectuadas nos conselhos de administração, tendo como objectivos principais garantir a viabilidade e a qualidade do investimento realizado. Era comum existir um vendedor para o hardware, software, suporte, formação e consultoria, sendo as relações entre o comprador e o vendedor muito próximas. Devido ao número reduzido de empresas, a concorrência entre as empresas era reduzida. O lançamento de novas versões era algo de muito raro, acontecendo normalmente em intervalos de 5 anos.

Uma década mais tarde o paradigma mudou. O computador tornou-se mais "ubiquo", mais barato e com maior capacidade de computação. Surgiu a Lei de Moore, segundo a qual o número de circuitos num chip de silicone duplica entre 18 e 24 meses [Moo07]. Os fabricantes de software foram obrigados a adoptar novas metodologias de desenvolvimento baseadas em lançamentos mais rápidos com o objectivo de conseguir acompanhar a evolução do hardware e das necessidades dos clientes que começavam a ser cada vez mais e maiores. Ao contrário das décadas anteriores em que as novas versões eram muito semelhantes às anteriores na década de 90, novas versões eram na sua maioria muito diferentes das antecedentes, o que originava novos erros. Esta nova agenda de lançamentos baseadas na sua maioria por decisões dos departamentos de marketing fez com que o tempo gasto na fase de testes diminuísse.

No final da década de 90, a Web explodiu. Um dos grandes problemas na altura era o

mecanismo de distribuição de correeccção de erros ser muito dispendioso. As equipas de suporte eram muito numerosas e os clientes estavam descontentes com o software. Rapidamente os fabricantes de software começaram a tirar partido das funcionalidades da Web para tornar mais eficiente o processo do correcção de erros, onde os utilizadores efectuavam o *download* de correcções no site do fabricante. Deixou de haver a necessidade dos clientes esperarem pela nova versão para terem os erros corrigidos. A Web tornou-se então a ferramenta de eleição para distribuição de actualizações de software. A Netscape foi pioneira na utilização da Internet para testar as suas aplicações através do lançamento de versões *betas* em larga escala. A sua metodologia era bastante simples: assim que a equipa de desenvolvimento fornecesse uma versão *"compilável"* (que pudesse ser executada), esta era lançada na Internet para que milhões de utilizadores a usassem e testassem, reportando o erros que encontrassem. Esta atitude foi muito prejudicial para a Netscape, pois ao ser utilizada durante muito tempo, os utilizadores começaram a ter uma ideia negativa da qualidade do software produzido pela companhia.

Os consumidores começaram a escolher o software baseado nas suas funcionalidades e não na sua fiabilidade. Devido a este facto, as companhias colocaram no fundo das prioridades a fiabilidade, dando prioridade às funcionalidades e redução de custos. Durante a década de 90, a Microsoft lançou 16 versões do seu sistema operativo Windows, ultrapassando em média uma por ano [hop07]. Grande parte do software que foi comprado e nunca foi sequer instalado (*shelfware*) [Hut04]

Por volta de 2000, o paradigma voltou a mudar. Com o fim da especulação sobre as empresas *"dot.com"* (ver figura 1.2), os clientes deixaram de comprar software apenas porque "era novo". As vendas do sistema operativo Windows 2000 ©não conseguiram acompanhar o que estava previsto [Wil07]. Apenas 10% dos utilizadores que tinham sistemas anteriores realizaram as actualizações para o novo sistema operativo.

Os investimentos em tecnologia passaram a ter um controlo financeiro muito forte e passou a existir análise prévia do retorno de investimento (ROI) antes de se tomar qualquer decisão. A área de teste de software voltou a ter novamente a importância necessária, mas com um grande problema: durante mais de uma década, a evolução de técnicas e processos de teste foi quase nula e não estava preparada para os novos desafios.

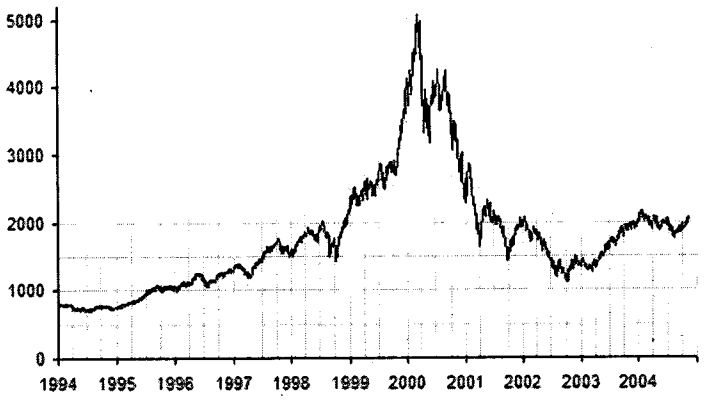

Figura 1.2: Gráfico que representa a evolução do índice NASDAQ entre 1994 e 2004

1.4 Situação actual

Nos dias de hoje, o software transformou-se numa parte indispensável de qualquer negócio quer seja no apoio ao desenvolvimento, na produção, no marketing ou no suporte de produtos e serviços. Em 2000, as vendas totais do software alcançaram aproximadamente $180 bilhões de dólares nos Estados Unidos da América, número este que revela a importância da venda de software [oCNIoST02].

Existem inúmeros casos conhecidos de perdas monetárias devido a uma fase de teste insuficiente, onde talvez o caso mais falado ultimamente tenha sido quando os investigadores da NASA em Novembro de 2006 chegaram à conclusão de que uma actualização no software em Junho causou a destruição da Mars Global Surveyor no mês de Junho do mesmo ano [Cow07].

Infelizmente os danos causados por erros de software não se limitam a perdas monetárias. Entre 1985 e 1987 pelo menos seis pessoas faleceram devido à exposição excessiva de radiação causada por um erro no software que controlava os níveis de radiação[Lev95].
Devido às grandes pressões nas equipas de desenvolvimento para que não existam desvios nos custos e nos tempos orçamentados, os testes de software são realizados para mostrar que um determinado software funciona e que cumpre os requisitos para que foi desenhado, usando dados de teste bem comportados e com a certeza que nada irá correr mal. Este comportamento não segue o principio geral de teste de software, *"descobrir que realmente*

possui erros"[Mye04].

Com cada vez mais empresas na área da produção e comercialização de software no mercado, estas tendem a reduzir os custos de desenvolvimento, reduzindo a fase de testes, levando a uma perda na qualidade no software produzido. Ao mesmo tempo, a complexidade do software aumenta a uma taxa considerável. Já não se fala em milhares de linhas de código, mas sim em milhões de linhas de código. O sistema operativo Mac OS X 10.4 possui 86 milhões de linhas de código [Blo07]. Este aumento de complexidade faz com que o número possível de erros aumente. Estes erros, além de criarem enormes prejuízos monetários, causam também falta de qualidade, o que leva a uma perda de confiança no software produzido. Em Dezembro de 2002 no jornal americano *The Economist* foi publicado um artigo em que era posta a causa a fiabilidade do Windows, indicando que *"se a Microsoft fabricasse automóveis, os processos legais em que era arguida devido a problemas de fiabilidade podiam levar a mesma à falência"* [Cfo07]. As empresas de software para se protegerem contra processos legais, comercializam o software com licenças indicando expressamente que não são responsáveis por prejuízos causados por defeitos no software.

Está provado que o custo de encontrar e corrigir um defeito aumenta consideravelmente com o tempo [Agi07]. A atitude certa é criar uma fase de testes associada a uma fase do projecto (ver figura 1.3 [Agi07])

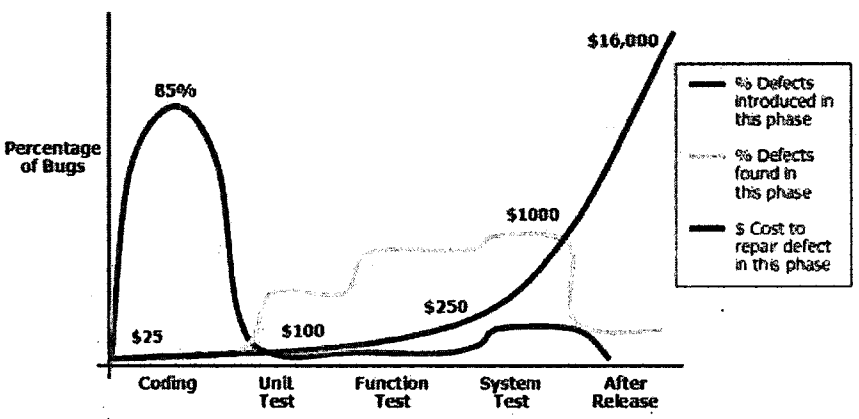

Figura 1.3: Custos de correcção de defeitos em função do tempo

Capítulo 2

Conceitos Fundamentais

"People make errors. The problem here was not the error. It was the failure of us to look at it end-to-end and find it. It's unfair to rely on any one person."

— Tom Gavin, JPL administrator da Lockheed Martin.

2.1 Testes de software

2.1.1 Introdução

O principal objectivo dos testes de software é garantir maior e confiança no software desenvolvido. Esta confiança é obtida através da execução do software com a utilização de dados de teste, dados estes que são escolhidos segundo algum padrão de forma a cobrir o maior número de casos possíveis de utilização do software [Mye04].

A necessidade cada vez maior de lançamentos rápidos e incrementais originou um conjunto de novas metodologias de desenvolvimento de software. O modelo tradicional de cascata representado na figura 2.1 está a perder notoriedade para modelos mais ágeis tais como Extreme Programming (bases representadas na figura 2.2 [Jef07]), Scrum [Cha07] ou DSDM [Con07], capazes de satisfazer as necessidades dos clientes. Estes novos paradigmas de desenvolvimento baseado em lançamentos rápidos e incrementais só são possíveis graças a testes de software que possam ser rapidamente executados sempre que necessário. Em tal cenário

os testes manuais são pouco vantajosos, visto que muitos testes são novamente executados a cada lançamento do sistema.

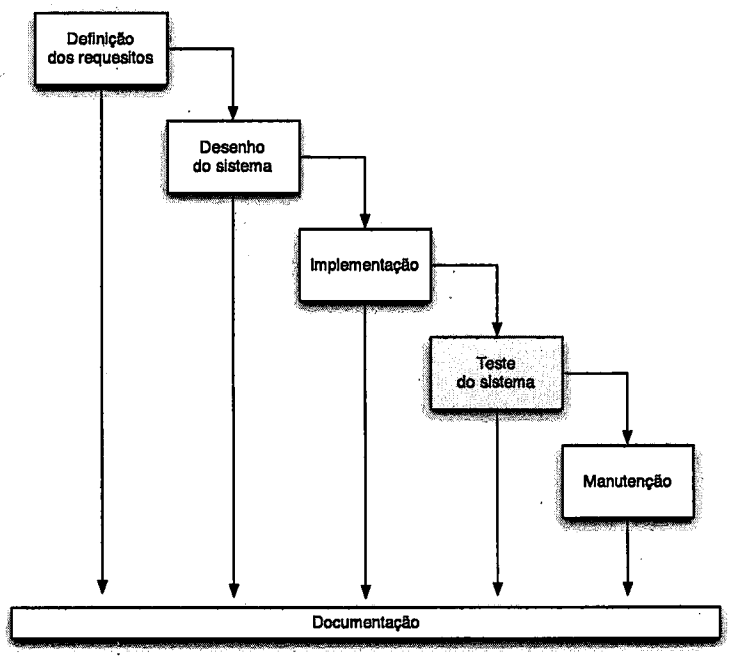

Figura 2.1: Modelo de desenvolvimento em cascata

2.1.2 Definições

Para enquadrar o leitor na temática do teste de software, vejamos o seguinte exemplo:

> Ao desenhar uma ponte, o arquitecto deve ter em consideração um conjunto de requisitos ambientais sobre os quais a ponte irá funcionar tais como a dimensão, força da corrente da água (se existir), força do vento, etc.
> Suponhamos que um camião de 20 toneladas atravessa a ponte e a ponte cai. Do ponto de vista do condutor do camião, a ponte *falhou*. Mas qual foi o defeito que levou à falha da ponte?

Existem várias repostas a esta questão:

- O condutor do camião poderá ter ignorado um sinal de proibição;

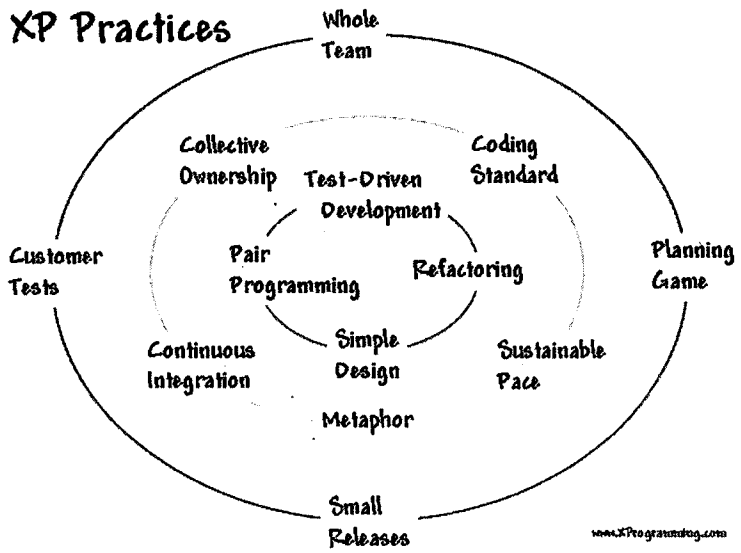

Figura 2.2: Práticas do Extreme Programming

- Um funcionário da ponte poderá ter colocado mal o sinal;
- Minutos antes deste camião ter passado, poderá ter passado um outro de 30 toneladas e quebrado a estrutura;
- O arquitecto poderá não ter desenhado correctamente a ponte;
- O construtor poderá não ter seguido a planta definida pelo arquitecto.

Neste exemplo são referidos os três conceitos essenciais em testes de software, **erro**, **defeito** e **falha**, que embora possam parecer que são sinónimos entre si, referem-se a conceitos diferentes na área de testes de software.

Erro (Error)

Um erro é uma falha humana que foi inserida no software, tendo como origem o facto do ser humano cometer equívocos. Os erros podem ser originados por programadores, analistas, equipas de teste e até mesmo por clientes.

A maioria dos erros são originados pelos programadores (seguindo a analogia do exemplo

anterior, podem não colocar bem o sinal de proibição). Os analistas também cometem erros (exemplo: não tiveram em consideração a força das águas na construção da ponte) e as equipas de teste também podem falhar (exemplo: na fase de testes não realizaram testes para verificar a robustez com ventos superiores a 50 km/hora).

De uma forma geral, os erros são classificados em[Ei90]:

Erros de computação O erro provoca uma computação incorrecta mas o caminho executado (sequências de comandos) é igual ao caminho esperado;

Erros de domínio O caminho efectivamente executado é diferente do caminho esperado, ou seja, um caminho errado é seleccionado.

Defeito (Fault)

Um defeito é uma consequência do erro no código (ou especificação / desenho) que irá funcionar de forma defeituosa. Um defeito é normalmente denominado por *"bug"*. Seguindo a analogia do exemplo anterior, pelo facto de não haver a sinalização adequada, um camião com um peso superior é permitido tentar atravessar a ponte.

Falha (Failure)

Um defeito pode ou não ter consequências finais. No caso de ter, irá provocar uma ou mais falhas, que são uma não conformidade do sistema com os requisitos. Temos uma falha quando por consequência de um erro, a informação é corrompida. Quando um estado pode levar à ocorrência de um defeito, pode-se dizer que o sistema está em estado de falha. O utilizador irá notar que o sistema teve um comportamento anormal.

Estes três conceitos são a base dos testes de software e estão interligados entre si (ver figura 2.3

Além destes três conceitos chave em testes software, o leitor devem também ter a noção de **Caso de Teste**, **Teste**, ***Test Oracle***, **Ambiente de Teste** e **Qualidade**.

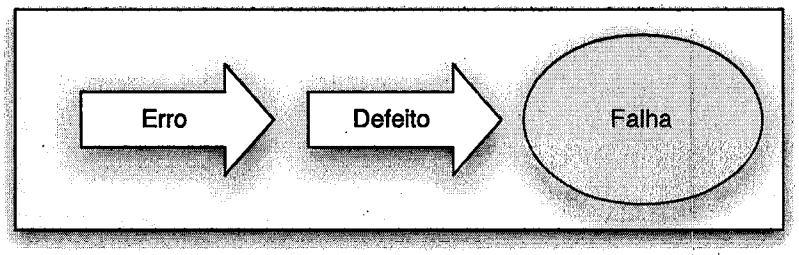

Figura 2.3: Relação entre erro, defeito e falha

Caso de teste (Test case)

O método mais usual para validar se um determinado componente de software possui erros é exercitar esse mesmo componente num determinado ambiente predefinido, por norma o mais semelhante ao ambiente real, gerando para tal um conjunto de dados de entrada.

A decisão sobre se um teste passa ou falha é baseada num resultado final esperado determinado no inicio do teste.

Segundo o IEEE [IEE90], um caso de teste pode ser definido como um terno composto por:

- Um conjunto de dados de entrada que servirá para exercitar o código a testar;
- Um resultado esperado depois de ser executado o código com os dados de entrada previamente assumidos;
- Um ambiente sobre o qual o código irá ser exercitado.

Teste

Um teste é um conjunto de casos de teste

Test Oracle

O *Test Oracle* é um documento ou software que especifica ou produz o resultado esperado de um determinado teste.

Ambiente de teste

O ambiente de teste é todo o ambiente (software e hardware) necessário para testar um determinado componente de software (sistema operativo, bases de dados, simuladores, hardware, etc ...)

Qualidade

O conceito de qualidade é algo que deve estar sempre presente no desenvolvimento de qualquer software. Segundo o IEEE [IEE90], qualidade de software pode ser definida como:

1. O grau com que um determinado sistema ou componente satisfaz os requisitos;
2. O grau com que um determinado componente ou sistema satisfaz as expectativas do utilizador.

Para determinar a qualidade de um determinado sistema, este é classificado segundo um conjunto de **atributos de qualidade.** Para esta classificação são usadas métricas de qualidade. Uma **métrica de qualidade** é uma medida quantitativa que avalia o grau que o sistema em causa possui de um determinado atributo de qualidade.

Exemplos de atributos de qualidade:

Comportamento apropriado . Permite conhecer o grau com que o software realiza as tarefas esperadas

Fiabilidade . Indica o grau com que determinado software executa as funções previstas num determinado período de tempo;

Usabilidade . Representa o grau necessário para aprender, operar, preparar os dados de entrada e compreender os dados de saída de um determinado software;

Integridade . Representa o grau com que um determinado software consegue "resistir" a ataques intencionais ou forçados;

Portabilidade Indica o grau de capacidade de um determinado software ser transferido entre ambientes;

Sustentabilidade Compreende o esforço necessário para alterar um determinado software;

Interoperabilidade Representa o esforço necessário para interligar um software com outro Também existe métricas que medem capacidade de o software ser testado.

2.1.3 Tipos de teste

Teste Funcional

O teste funcional tem como objectivo a verificação da aceitação dos dados, do processamento, da resposta a este processamento e a implementação apropriada das regras de negócio. Este tipo de teste é baseado nas técnicas de **Black Box** (adiante explicada) isto é, verificar o sistema e seu processo interno pela sua interacção através da interface e da análise das saídas ou resultados.

Teste de Volume

O teste de volume submete grandes quantidades de dados ao sistema para determinar se limites que causam a falha do software são alcançados. Este tipo de teste também identifica a carga ou volume máximo persistente que o sistema pode suportar por um dado período.

Teste de Segurança

Os testes de segurança têm como objectivo validar questões relacionadas com a aplicação, podendo ser classificados em dois tipos:

Segurança ao nível de aplicação Valida se os requisitos de segurança são cumpridos ao nível da aplicação;

Segurança ao nível do sistema Valida se as questões relacionadas com o ambiente que rodeia a aplicação cumpre os requisitos definidos.

Teste de Acessibilidade

O teste de acessibilidade verifica se a interface do utilizador fornece o acesso apropriado às funções do sistema e a navegação adequada. Além disso, estes testes garantem que os objectos dentro da interface do utilizador funcionam de acordo com os padrões definidos pelo cliente. Estes testes só são realizados se existir uma interface para o utilizador.

Teste de Usabilidade

O teste de usabilidade verifica a facilidade que o software possui de ser claramente entendido e usado pelos utilizadores.

Teste de Stress

O teste de stress mede o comportamento do sistema em condições limites ou fora da tolerância esperada. Tipicamente envolve recursos com pouca disponibilidade ou acessos concorrentes. Recursos com pouca disponibilidade revelam defeitos que não são aparentes em condições normais. Outros defeitos devem resultar da concorrência por recursos partilhados, por exemplo largura de banda, acessos a bases de dados, espaço em disco,etc ....

Teste de Regressão

Testes de regressão são testes que são aplicados a software que foi modificado, tendo como como propósito objectivo validar que qualquer falha tenha sido reparada e que nenhuma operação que funcionava anteriormente tenha falhado após os reparos, ou seja, que as novas características adicionadas não criaram problemas com as versões anteriores ou com outros sistemas.

Teste de Carga

Os estes de carga submetem o sistema à variação de carga de trabalho para medir e avaliar os comportamentos de desempenho e a sua capacidade de continuar a funcionar apropriadamente sob cargas de trabalho diferentes. São utilizados também para avalias as características de desempenho, assim como tempos de resposta, taxas de transacções e outras características sensíveis ao tempo.

Teste de Instalação

O teste de instalação possui dois objectivos: garantir que o software pode ser instalado sob condições apropriadas; e verificar que uma vez instalado, o software funciona correctamente.

Teste de Configuração

O teste de configuração verifica a operação do sistema em diferentes configurações de software e hardware. As especificações de hardware e software podem mudar e esta mudança pode afectar o sistema.

2.1.4 Métodos de teste

Black Box Testing

Os métodos de Black Box Testing[dCZ03] concentram-se nos requisitos funcionais do software, procurando descobrir erros nas seguintes categorias:

- Procedimentos ou funções incorrectas;
- Erros de interface;
- Erros de desempenho;
- Erros em estruturas de dados ou no acesso a bases de dados externas;

Particionamento por Classes de Equivalência (Equivalence Partitioning) . Já foi referido que é impossível testar um determinado software com todos os casos de entrada possíveis. A técnica de Particionamento por Classes de Equivalência[dCZ03] tenta ultrapassar esta questão dividindo o domínio de entrada de um programa em classes de dados a partir das quais os casos de teste podem ser gerados. Este método procura definir um caso de teste que descubra classes de erros, reduzindo o número total de casos de teste que devem ser resolvidos.

Análise de valor limite (Boundary value analysis) .

O programador tende a ignorar as fronteiras dos dados de entrada, assumindo erradamente que se o sistema tem um comportamento esperado no **centro**, também o terá para o resto do dominio da aplicação[dCZ03] . Existindo menos atenção pelo programador em dados de entrada fronteiriços, a probabilidade em existir erros nestas fronteiras é maior do que no centro. Esta metodologia tenta ultrapassar este problema tendo como objectivo testar dados fronteiriços.

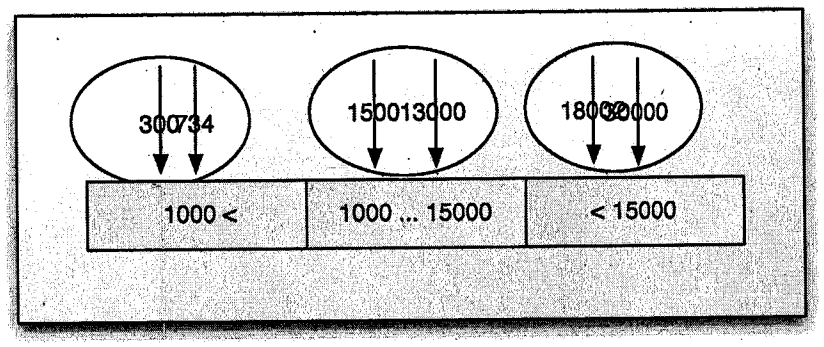

Figura 2.4: Representação gráfica da Técnica de Particionamento por Classes de Equivalência

Se uma condição de entrada definir um intervalo delimitado pelos valores a e b, os casos de teste devem ser projectados com valores a e b logo acima e logo abaixo de a e b, respectivamente. Se uma condição de entrada especificar uma série de valores, os casos de teste que ponham à prova números máximos e mínimos devem ser desenvolvidos. Valores logo acima e logo abaixo do mínimo e do máximo também devem ser testados.

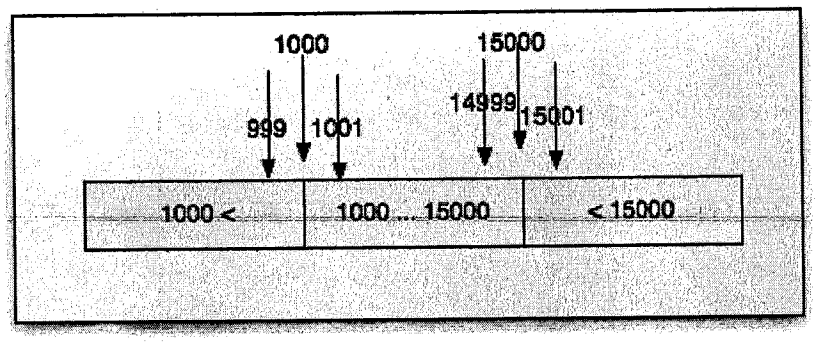

Figura 2.5: Representação gráfica da técnica Análise de valor limite

Técnicas de grafo de causa-efeito O grafo de causa-efeito é uma técnica de desenho de casos de teste que oferece uma representação concisa das condições lógicas e das acções correspondentes. A técnica segue quatro passos:

- Causas (condições de entrada) e efeitos (acções) são relacionados para cada módulo, sendo-lhe atribuído um identificador.

- É gerado um grafo de causa efeito.
- O grafo é convertido numa tabela de decisão.
- As regras da tabela de decisão são convertidas em casos de teste.

White Box Testing

Os métodos de White Box envolvem sempre o conhecimento do componente que se está a testar. Os casos de teste são criados com base nesse conhecimento, desenvolvidos na maioria das vezes pelos programadores. Estes métodos têm como objectivo garantir que as funções implementadas são executadas pelo menos uma vez e seja validado o seu comportamento.

2.1.5 Fases de teste

A execução dos vários tipos de teste é algo que não é estático, mudando consoante a metodologia de desenvolvimento adoptada para um determinado software. Neste trabalho foram escolhidas as fases de teste existentes no modelo de desenvolvimento e validação em V, actualmente o mais usado.

Modelo em V

A maior parte do software desenvolvido segundo esta metodologia adopta quatro fases:

- Testes de Componente
- Testes de Integração
- Testes de Sistema
- Testes de Aceitação

Cada uma das fases possui objectivos de teste bem definidos, havendo uma relação muito próxima entre as fases de desenvolvimento e de testes. A titulo de exemplo, a fase de testes de componentes apenas se preocupa em testar os componentes, tentando encontrar defeitos estruturais nos vários componentes que compõem o sistema.

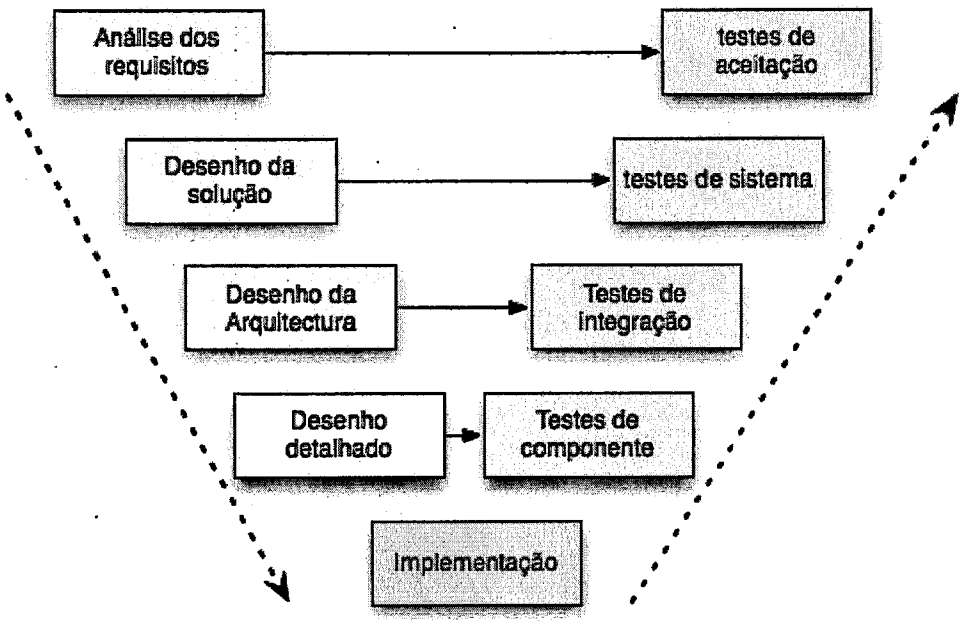

Figura 2.6: Modelo de desenvolvimento em V

Testes de unidade

O teste unitário ou de unidade é um processo que consiste na verificação da menor unidade do projecto de software. Em sistemas construídos com uso de linguagens orientadas a objectos, um teste invoca um método de uma classe, verificando se é obtido o retorno previsto.

Os testes unitários são da responsabilidade do próprio programador durante a fase de implementação, logo após de ter finalizado a programação do componente. Geralmente, um teste unitário executa um método individualmente e compara uma saída conhecida após o processamento da mesma. Os testes unitários são considerados os primeiros de uma cadeia de testes à qual um software pode ser submetido. Nesta fase não se pretende testar toda a funcionalidade de uma aplicação, mas sim os diversos componentes de forma isolada.
Novas metodologias de desenvolvimento (**Test-Drive-Development (TDD)**) valorizam os testes unitários ao máximo, sugerindo que o foco do desenvolvimento são os testes unitários e que estes deveriam ser os primeiros a serem desenhados e implementados, definindo que o software estaria terminado quando o resultado de todos os testes unitários fosse positivo.

Níveis de automatização de testes unitários Existem cinco fases necessárias para realizar correctamente a automatização de testes unitários (Desenho, Implementação, Preparação, Execução e Avaliação). Quanto maior for o número de fases automatizadas, maior é a eficiência dos testes unitários. Testes unitários podem ser classificados consoante o número de fases que são automatizadas:

Nível 1 . Todas as fases são realizadas manualmente, não havendo qualquer automatização.

Nível 2 . Embora não havendo nenhuma fase automatizada, é utilizado uma metodologia para o desenho dos testes.

Nível 3 . Neste nível os testes já são executados de forma automatizada, normalmente através de *scripts* desenvolvidos especificamente para o software em análise. No entanto, todas as restantes fases são realizadas manualmente.

Nível 4 . Neste nível de maturação, os testes unitários são desenhados, implementados, preparados e executados de forma automática mas a analise ainda é feita de forma manual.

Nível 5 . Todas as fases são realizadas de forma automatizada.

Testes de Integração

Na fase de teste de integração, o objectivo é encontrar falhas originadas pela integração interna das unidades que compõem um sistema. Normalmente os tipos de falhas encontradas são de envio e recepção de dados.

Testes de Sistema

Na fase de Teste de Sistema o objectivo é executar o sistema sob o ponto de vista do seu utilizador final, percorrendo todas as funcionalidade tentando encontrar falhas que ainda não tenham sido detectadas. Estes testes deverão ser executados num ambiente o mais semelhante possível com o real.

Testes de Aceitação

A fase de testes de Aceitação tem como objectivo determinar se um determinado sistema satisfaz ou não os seus critérios de aceitação, validando se o cliente aceita ou não o sistema em causa. São testes que normalmente são realizados por um grupo restrito de utilizadores finais do sistema, simulam operações de rotina no sistema de modo a verificar se o comportamento é o esperado. É utilizado para a validação de um software pelo comprador, pelo utilizador ou por uma terceira entidade, aplicando dados e cenários específicos ou reais. Podem incluir testes funcionais, de configuração, de recuperação de falhas, de segurança e de desempenho.

Teste de Regressão

É uma fase de teste aplicável a uma nova versão de software ou então à necessidade de se executar um novo ciclo de teste durante o processo de desenvolvimento. Consiste em aplicar, a cada nova versão do software ou a cada ciclo de desenvolvimento todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema.

Para efeito de aumento de produtividade e de viabilidade dos testes, é recomendada a utilização de ferramentas de automatização de testes, de forma que, sobre a nova versão ou ciclo de teste, todos os testes anteriores possam ser executados novamente com maior agilidade.

2.2 Testes unitários em Java

2.2.1 JUnit

JUnit[JUn07] é a *framework* de referência para testes automatizados em JAVA desenvolvida por Kent Beck e Erick Gamma A sua enorme expansão deve-se ao facto da sua enorme simplicidade e de ser código aberto.

O seu conceito é muito simples e aparentemente os programadores gostam de a utilizar porque um teste implica a construção de um método numa classe especial, tornando a sua utilização muito simples e de fácil aprendizagem para o programador.

JUnit é constituída essencialmente por três classes (ver figura 2.7):

- A classe `TestCase` é a classe típica de teste. Cada uma das restantes classes representadas depende desta. A `TestCase` é uma subclasse da classe `Test`.

- Uma `TestSuite` representa um conjunto de testes, derivando igualmente da classe `Test`

- Uma `TestRunner` permite executar uma lista de `TestCases`.

Para utilizar o JUnit, é necessário criar uma classe que estenda `junit.framework.TestCase`. A partir daí, para cada método a ser testado é necessário definir métodos, os quais devem ser públicos e sem retorno de argumentos e cujo nome comece por `test`.

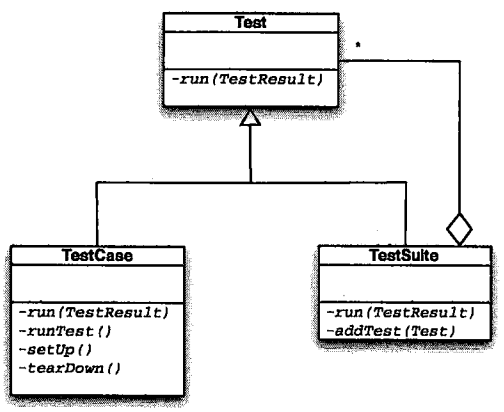

Figura 2.7: Diagrama de classes simplificado da ferramenta Junit

No anexo K encontra-se um exemplo de dois testes unitários feitos à classe J.

JUnit assume por omissão que um teste sucede a não ser que o método gere uma excepção não tratada ou seja indicado que o mesmo falhou através do uso de `fail()`.

Esta ferramenta possui dois métodos que ajudam à construção dos casos de teste: `setUp()` e `tearDown()`. O primeiro (linha 10 do exemplo K) permite configurar todo o ambiente de testes enquanto que o segundo permite voltar a colocar ao estado inicial o ambiente de testes (linha 15 do exemplo K).

2.3 Teste Baseado em Erros

Não é fácil classificar se os testes realizados são os adequados ao software em análise. Se uma determinada aplicação passa com sucesso um conjunto determinado de casos de testes, apenas podemos concluir que a aplicação cumpre os requisitos naquele determinado conjunto. Quanto maior for o número de casos de teste, maior será a probabilidade de a aplicação ter um funcionamento correcto quando passar a produção. Actualmente não existe nenhum método determinístico que determine a eficácia dos casos de teste realizado numa determinada aplicação.

A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais frequentes no processo de desenvolvimento de software para gerar os requisitos de teste. Esta técnica baseia-se nos erros que o programador ou arquitecto possa cometer durante o as suas tarefas e nas abordagens que podem ser usadas para detectar a sua ocorrência.

Inserção de erros(*Error Seeding*) [Bud01] e **Análise de Mutantes**(*Mutation Analysis*) [RAD78] são critérios típicos que se concentram em erros para testar um determinado software. Esta tese dá importância ao critério Análise de Mutantes.

2.3.1 Teste por Mutação

O critério Análise de Mutantes surgiu na década de 70 em duas universidades: Yale University e Georgia Institute of Technology, possuindo um forte relacionamento com um método clássico para detecção de erros lógicos em circuitos digitais – modelo de teste de falha única [Fri75]

Vejamos o seguinte exemplo:

> Para estimar o número de peixes que existem num determinado lago, uma maneira será identificar um conjunto limitado de peixes (por exemplo 20), marcá-los e voltar a colocá-los no lago. Depois Iremos tentar apanhar alguns. Se conseguirmos apanhar 40 peixes e desses 4 são, então podemos estimar que a população rondará os 200 peixes. Se os apanharmos todos, então teremos encontrado provavelmente toda a população.

É este o principio que está por detrás da mutação por testes: gerar alguns defeitos no

código e tentar encontrar esses mesmo defeitos. Se for possível encontrar todos os defeitos gerados, então a nossa *rede de pesca* (bateria de testes) conseguirá com grande probabilidade também apanhar todos aqueles que não foram gerados propositadamente.

Vejamos o seguinte exemplo de código feito na linguagem C indicado no anexo M
Gerando três mutantes ficamos com:

1. Mutante 1 (ver anexo N) : alterar a linha 8 para a linha

```
if(atoi(argv[2]) <= 5)
```

2. Mutante 2 (ver anexo O) : alterar a linha 6 para a linha

```
if(atoi(argv[1]) >= 3)
```

3. Mutante 3 (ver anexo M) : alterar a linha 5 para a linha

```
int c=3;
```

Os mutantes 1 e 3 não alteram sintacticamente o resultado final da aplicação quando foram executados os cinco casos de teste, por isso são **mutantes equivalentes.** Pelo contrário, o mutante dois obteve um comportamento diferente do esperado, tornando-o um **mutante eliminado.**

Efectuando uma analise estatística, permite verificar que ao criar três mutantes, apenas um foi eliminado. Isto indica que o nível de mutação da bateria de testes foi de $\frac{1}{3}$. O valor $\frac{1}{3}$ não é muito elevado, indicando que não estamos a realizar os testes necessários. Em suma, o programa possui dois erros sérios que não foram detectados. Analisando com mais detalhe o mutante 2, este fez com que o programa falhasse, detectando um erro no código [Kol99].

Esta é a ideia principal dos testes por mutação. Se um determinado conjunto de testes conseguir detectar todas as alterações feitas ao código (eliminar o mutante), então irá detectar falhas reais que não são conhecidas [Bybro].

2.3.2 hipótese do programador competente e efeito de união

DeMillo, Lipton e Sayward publicaram um artigo em 1978 descrevendo a ideia de teste de mutantes [RAD78]. A ideia apresentada, conhecida como hipótese do programador competente

(*competent programmer hypothesis*) [RAD78], assume que os programadores com experiência escrevem programas correctos ou muito próximos do correcto. Assumindo a validade desta hipótese, pode-se afirmar que erros são introduzidos nos programas através de pequenos desvios sintácticos que, embora não causem erros sintácticos, alteram a semântica do programa e, consequentemente, conduzem o programa a um comportamento incorrecto. Para revelar tais erros, a Análise de Mutantes identifica os desvios sintácticos mais comuns e, através da aplicação de pequenas transformações sobre o programa em teste, encoraja aquele que testa a construir casos de testes que mostrem que tais transformações levam a um programa incorrecto [ADH+89].

Uma outra hipótese explorada na aplicação do critério Análise de Mutantes é o efeito de união (coupling effect) [RAD78], a qual assume que erros complexos estão relacionados a erros mais simples. Assim sendo, espera-se, (e alguns estudos empíricos já confirmaram esta hipótese [Bud80]), que conjuntos de casos de teste capazes de revelar erros simples são também capazes de revelar erros complexos. Nesse sentido, aplica-se uma mutação de cada vez no programa em teste, ou seja, cada mutante contém apenas uma transformação de sintaxe. Um mutante com k transformações sintácticas é referenciado por k-mutante; neste texto são utilizados apenas 1-mutantes.

Assumindo estas duas hipóteses, ao testar um software P através da utilização de um conjunto de casos de teste T, se os testes encontrarem algum defeito, os testes terminam, caso contrário são adicionados erros ao código dando origem $P_1, P_2, \ldots P_n$ denominados **mutantes**.

Com o objectivo de definir os erros sintácticos mais comuns, os **operadores de mutação** (*mutant operators*) são aplicados ao programa P, transformando-o em mutantes de P. Define-se como *operador de mutação* um conjunto de regras que definem as alterações a aplicar a P.

Operadores de Mutação

Os operadores de mutação possuem dois objectivos [ABGJ02]:

1. Inserir mudanças sintácticas simples baseada nos erros cometidos pelos programadores (por exemplo alterar operadores aritméticos)
2. Inserir mudanças com o objectivo de forçar determinados objectivos de teste (como executar cada arco do programa)

Os operadores de mutação devem sempre que possível causar o insucesso do teste. Nesta tese, a linguagem utilizada na implementação da framework é a linguagem Java e os operadores utilizados estão indicados na secção 2.3.3.

Um ponto importante destacado em [RAD78] é que a Análise de Mutantes fornece uma medida objectiva do grau de adequação dos casos de teste analisados através da definição de um **nível de mutação** (*mutation score*), que relaciona o número de mutantes eliminados com o número de mutantes gerados. O nível de mutação é calculado da seguinte forma:

$$nm(P,T) = \frac{DM(P,T)}{M(P)}$$

sendo:

- DM(P,T): número de mutantes eliminados pelos casos de teste em T;
- M(P): número total de mutantes gerados;

O nível de mutação varia no intervalo entre 0 e 1, quanto maior for o seu valor, mais adequado é o conjunto de casos de teste para o programa a ser testado .

2.3.3 Testes por Análise de Mutantes em Java

Operadores de Mutação

A linguagem Java devido às suas características, obriga a que existam dois tipos de operadores de mutação:

1. Operadores de mutação para métodos.
2. Operadores de mutação para classes.

Na implementação dos operadores de mutação, foi seguido em grande parte o trabalho de Yu-Seung Ma e Jeff Offutt compilado em dois artigos: *Description of Method-level Mutation Operators for Java*[MO05b] e *Description of Class Mutation Mutation Operators for Java* [MO05a]

Operadores de mutação para métodos

O principal objectivo dos operadores de mutação para métodos é encontrar erros causados por descuido do programador na utilização simples dos operadores disponíveis na linguagem.

Operadores aritméticos Java possui cinco operadores aritméticos para valores inteiros (`int`, `long`, `short`, `byte`) e decimais (`double`, `float`): `+`, `-`, `*`, `/` e `%`.

Todos estes operadores são binários, no entanto os operadores `+` e `-` também podem ser utilizados de forma unária (por exemplo -1) e de uma terceira forma, através da abreviação de expressões (`op++`, `op--`, `--op`, `++op`). Na tabela 2.1 encontram-se a lista de operadores.

Operadores	Descrição
AOR_b	Altera os operadores binários aritméticos por outros (Exemplo: substituir `a+2` por `a-2`)
AOR_u	Altera os operadores unários aritméticos por outros (Exemplo: substituir `int x=-1` por `int x=+1`)
AOR_s	Altera os operadores aritméticos de abreviatura por outros (Exemplo: substituir `a++` por `a--`)
AOI_u	Insere um operador aritmético unário (Exemplo: substituir `int foo=1` por `int foo=-1`)
AOI_s	Insere um operador artimético de abreviatura (Exemplo: substituir `int foo=a` por `int foo=a++`)
AOD_u	Elimina um operador aritmético unário (Exemplo: substituir `int foo=-1` por `int foo=1`)
AOD_s	Elimina um operador aritmético de abreviatura (Exemplo: substituir `int foo=bar--` por `int foo=bar`)

Tabela 2.1: Lista de operadores de mutação aritméticos para métodos em JAVA

Operadores relacionais A Linguagem Java possui seis operadores relacionais: `>`, `>=`, `<`, `<=`, `==` e `!=`. Os operadores relacionais são binários, pelo que apenas é permitido haver permuta entre operadores. Na tabela 2.3 encontra-se o operador.

Operadores condicionais Java possui seis operadores condicionais, cinco binários e um unário. Os operadores binários são: `&&`, `||`, `&`, `|` e `^`. O operador unário é o `!`. Na tabela 2.5 encontra-se os operadores relacionais.

Operadores	Descrição
ROR	Substitui operadores relacionais por outros (Exemplo: substituir `a > 2` por `a < 2`)

Tabela 2.3: Lista de operadores de mutação relacionais para métodos em JAVA

Operadores	Descrição
COR	Substitui operadores condicionais binários por outros (Exemplo: substituir `foo && bar` por `foo || bar`)
COI	Insere operadores condicionais unários (Exemplo: substituir `if (accept)` por `if (!accept))`
COD	Remove um operador condicional unário (Exemplo: substituir `if (!accept)` por `if (accept))`

Tabela 2.5: Lista de operadores de mutação condicionais para métodos em JAVA

Operadores de bits A Linguagem Java possui sete operadores de manipulação de *bits*. Os operadores são : >>,<<,>>>>, &, —, ^ e ~. Na tabela 2.7 encontra-se descritos os operadores de manipulação de bit's.

Operadores de afectação A Linguagem Java possui 11 tipos de afectação (exp_{left} **op** op_{right}): +=, -=, *=, /=, %=, |=, &=, ^, <<=, >>=, >>>>=. Na tabela 2.9 encontra-se descritos os operadores de manipulação de bit's.

Operadores	Descrição
SOR	Permuta operadores binários (Exemplo: substituir `foo << 2` por `foo >> 2`)
LOI	Adiciona um operador de bit unário (Exemplo: substituir `foo=bar` por `foo=~bar`)
LOI	Remove um operador de bit unário (Exemplo: substituir `foo=~bar` por `foo=bar`)

Tabela 2.7: Lista de operadores de mutação para operações da manipulação de *bit's*

Operadores	Descrição
AOI_u	Permuta operadores de afectação do mesmo tipo (Exemplo: substituir `int foo+=bar` por `int foo-=bar`)

Tabela 2.9: Lista de operadores de mutação para operações da afectação

Operadores de mutação para classes

Os operadores de mutação para classes estão definidos em quatro grupos [MO05a]:

1. Encapsulamento.
2. Herança.
3. Polimorfismo.
4. Características especificas da linguagem.

Encapsulamento

AMC **Access modifier change** . Os conceitos de encapsulamento de variaveis de instância e de métodos são, na maioria das vezes, mal utilizados pelos programadores, levando a que possam existir problemas na fase de integração. Na tabela 2.10 encontra-se um exemplo da aplicação deste operador.

Original	Mutante
`private Stack stack;`	**`public Stack stack;`**

Tabela 2.10: Exemplo da aplicação do operador de mutação *AMC*

Herança

IHD **Hiding variable deletion** *Variable shadowing* [Bra07] é uma propriedade muito interessante da linguagem JAVA que permite que variáveis de instância da **superclasse** possam ser ocultadas em subclasses, no entanto uma má utilização desta característica pode levar a erros. Este operador possui o objectivo de encontrar más utilizações, eliminando uma variável protegida[MO05a] . Na tabela 2.11 encontra-se um exemplo da aplicação deste operador.

IHI **Hiding variable insertion** É o caso contrário do operador *IHD*, inserindo uma variável numa subclasse que já exista na **superclasse**. Na tabela 2.12 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

class List {
  int size;
  ...
}
class Stack extends List{
 int size ;
  ...
}
	class List {
	  int size;
	  ...
	}
	class Stack extends List{
	 //int size ;
	  ...
	}

Tabela 2.11: Exemplo da aplicação do operador de mutação

Original	Mutante

class List {
  int size;
  ...
}
class Stack extends List{
  ...
}
	class List {
	  int size;
	  ...
	}
	class Stack extends List{
	 int size ;
	  ...
	}

Tabela 2.12: Exemplo da aplicação do operador de mutação

IOD **Overriding method deletion** A linguagem permite que **subclasses** possam redefinir métodos que já existam na **superclasse**. É necessário então validar se o método a invocar é o correcto. O operador *IOD* elimina um método de uma **subclasse** que já exista na **superclasse**. Na tabela 2.13 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

`class List {`
`.... ....`
`void push(Object a){...}`
`}`
 	
`class Stack extends List`
`{`
`.... ....`
`void push(Object a){...}`
`}`
	`class List {`
	`.... ....`
	`void push(Object a){...}`
	`}`
	
	`class Stack extends List`
	`{`
	`.... ....`
	`//void push(Object a){...}`
	`}`

Tabela 2.13: Exemplo da aplicação do operador de mutação

IOP **Overridden method calling position change** Em algumas situações, o programador pode ter a necessidade de invocar um método da **superclasse** que está a redefinir. Quando isto é necessário, pode haver um engano pelo programador e invocar o método na altura errada. Este operador altera a posição das chamadas de métodos a redefinir. Na tabela 2.14 encontra-se um exemplo da aplicação deste operador.

IOR **Overridden method rename** Este operador valida se ao redefinir métodos não afecta o comportamento de outros métodos. Na tabela 2.15 encontra-se um exemplo da aplicação deste operador.

ISI **super keyword insertion** A palavra reservada **super** é usada para aceder aos membros da **superclasse**. Quando existe *Variable shadowing* [Bra07] poderá haver uma má utilização. O operador de mutação *ISI* insere a palavra reservada **super**, garantindo que são usados os membros correctos. Na tabela 2.16 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

class List {
...
void SetEnv(){size=5}
}
 	
class Stack extends List
{
void setEnv(){
super.setEnv();
size=10;
}
}
	class List {
	...
	void SetEnv(){size=5}
	}
	
	class Stack extends List
	{
	void setEnv(){
	size=10;
	super.setEnv();
	}
	}

Tabela 2.14: Exemplo da aplicação do operador de mutação *IOP*

Original	Mutante

class List {
....
void f(){...}
void m(){... f();...}
}
 	
class Stack extends List {
....
void f(){...}
void m(){... f();...}
}
	class List {
	....
	void f'(){...}
	void m(){... f'();...}
	}
	
	class Stack extends List {
	....
	void f(){...}
	void m(){... f();...}
	}

Tabela 2.15: Exemplo da aplicação do operador de mutação *IOR*

Original	Mutante

`class List {`
`.... ....`
`}`
 	
`class Stack extends List {`
`int size(){`
`return size;`
`}`
`}`
	`class List {`
	`.... ....`
	`}`
	
	`class Stack extends List {`
	`int size(){`
	`return super.size;`
	`}`
	`}`

Tabela 2.16: Exemplo da aplicação do operador de mutação *ISI*

ISD **super keyword deletion** Operador contrário ao *ISI* [MO05a]. Na tabela 2.17.

Original	Mutante

`class List {`
`.... ....`
`}`
 	
`class Stack extends List {`
`int size(){`
`return super.size;`
`}`
`}`
	`class List {`
	`.... ....`
	`}`
	
	`class Stack extends List {`
	`int size(){`
	`return size;`
	`}`
	`}`

Tabela 2.17: Exemplo da aplicação do operador de mutação *ISD*

IPC **Explicit call of a parent's constructor deletion** O operador *IPC* remove do construtor de uma **subclasse** a invocação do construtor da **superclasse** de forma a encontrar possíveis erros no construtor da **superclasse** [MO05a]. Na tabela 2.18 encontra-se um exemplo de aplicação.

Polimorfismo Polimorfismo em JAVA permite que um objecto possa ter comportamentos diferentes consoante o tipo do objecto, é necessário então testar todos possíveis tipos de modo a validar a robustez do programa.

Original	Mutante

`class List {`
`.... ....`
`}`
 	
`class Stack extends List {`
`Stack(int size){`
`super(size);`
`}`
`}`
	`class List {`
	`.... ....`
	`}`
	
	`class Stack extends List {`
	`Stack(int size){`
	`//super(size);`
	`}`
	`}`

Tabela 2.18: Exemplo da aplicação do operador de mutação *IPC*

PNC **new method call with child class type** Permite a alteração do tipo de uma referência, fazendo com que a referência a um objecto seja diferente da declarada. Na tabela 2.19 encontra-se exemplificado uma aplicação deste operador.

Original	Mutante
`Parent p;` `p=new Parent();`	`Parent p;` `p=new Child();`

Tabela 2.19: Exemplo da aplicação do operador de mutação *PNC*

PMD **Member variable declaration with parent class type** Altera o tipo da declaração de uma variável de instância para a da **superclasse.**

Original	Mutante
`Child c;` `c=new Child();`	`Parent p;` `p=new Child();`

Tabela 2.20: Exemplo da aplicação do operador de mutação *PMD*

PPD **Parameter variable declaration with child class type** Tem um comportamento

semelhante ao operador PMD, no entanto altera o tipo da variável na altura em que é usado através da operação de *cast* . Na tabela 2.21 encontra-se um exemplo da aplicação deste operador.

Original	Mutante
`boolean equals(Child o) {...}`	`boolean equals(Parent o) {...}`

Tabela 2.21: Exemplo da aplicação do operador de mutação PPD

PCI **Type cast operator insertion** Permite a alteração do tipo de um objecto para o da **superclasse** ou **subclasse**. Este mutante tem como objectivo encontrar diferentes comportamentos quando o objecto no qual é efectuado o *cast* possui variáveis protegidas ou métodos redefinidos. Na tabela 2.22 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

`Child cRef;`
`Parent pRef=cRef;`
`pRef.toString();`
	`Child cRef;`
	`Parent pRef=cRef;`
	`((Child)pRef).toString();`

Tabela 2.22: Exemplo da aplicação do operador de mutação PCI

PCD **Type cast operator deletion** É o caso inverso do operador PCI. Na tabela 2.23 encontra-se um exemplo da aplicação.

PCC **Cast type change** Altera o tipo da variavel do *cast* para o valor da **superclasse** ou **subclasse**.A tabela 2.24 remete-se a um exemplo de aplicação

PRV **Reference assignment with other compatible type** Referências a objectos podem ser do tipos de **subclasses** da classe declarada. Este operador altera a afectação de uma variável por outra compatível com o tipo declarado. Na tabela 2.25 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

`Child cRef;`
`Parent pRef=cRef;`
`((Child)pRef).toString();`
	`Child cRef;`
	`Parent pRef=cRef;`
	`pRef.toString();`

Tabela 2.23: Exemplo da aplicação do operador de mutação PCD

Original	Mutante
`((Parent)pRef).toString();`	**`((Child)pRef).toString();`**

Tabela 2.24: Exemplo da aplicação do operador de mutação PCC

Original	Mutante

`Object obj;`
`String s="Hello";`
`Integer i=new Integer(4);`
`obj=s;`
	`Object obj;`
	`String s="Hello";`
	`Integer i=new Integer(4);`
	`obj=i;`

Tabela 2.25: Exemplo da aplicação do operador de mutação PVR

OMR **Overloading method contents change** A linguagem Java permite que dois ou mais métodos da mesma classe possuam o mesmo nome desde que os argumentos possuam tipos diferentes. É normal os programadores confundirem os métodos com o mesmo nome. Este operador altera o corpo de um método pela invocação de um outro com o mesmo nome através a utilização da *keyword* `this`. Na tabela 2.26 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

`class List {`
`...`
`  void add(Object o){...};`
 	
`  void add(Object o,int n){...};`
`...`
`}`
	`class List {`
	`...`
	`  void add(Object o){...};`
	
	`  void add(Object o,int n){`
	`  this.add(o);`
	`  };`
	`...`
	`}`

Tabela 2.26: Exemplo da aplicação do operador de mutação *OMR*

OMD **Overloading method deletion** Este operador remove métodos com o mesmo nome, um de cada vez. Se o mutante não for eliminado, poderá haver um erro num nos métodos com o mesmo nome. Na tabela 2.27 encontra-se um exemplo da aplicação deste operador.

Original	Mutante

`class List {`
`...`
`  void add(int i){...};`
 	
`  void add(float j){...};`
`...`
`}`
	`class List {`
	`...`
	`  // void add(int i){...};`
	
	`  void add(float j){...};`
	`...`
	`}`

Tabela 2.27: Exemplo da aplicação do operador de mutação *OMD*

OAC **Argument of overloading method change** Altera a ordem ou o número dos argumentos na altura da invocação de um método se existir mais do que um método com o mesmo nome compatível com os argumentos. Na tabela 2.28 encontra-se um exemplo da aplicação deste operador.

Original	Mutante
`hashtable.push("key","value");`	`hastable.push("value","key");`

Tabela 2.28: Exemplo da aplicação do operador de mutação *OAC*

2.4 Ferramentas de teste por mutação para a linguagem Java

Foram analisadas duas ferramentas de teste por mutação em Java em que os pontos de análise foram :

- Possuir código aberto.
- Implementar o maior número de operadores de mutação possível.
- Possuir uma arquitectura simples e expansível.
- Documentação.
- Integração com Junit.

Jumble

A ferramenta Jumble é uma ferramenta de testes por Análise de Mutantes em Java , desenvolvida inicialmente pela companhia Reel Two sediada na Nova Zelandia[Two07] encontrando-se actualmente com licenciamento *open source* [Jum07].

Actualmente os mutantes suportados são por esta ferramenta são:

- Operadores de bit's.
- Operadores aritméticos.

- Operadores relacionais.
- Operadores de afectação.
- Operadores condicionais.
- Operadores de mutação de constantes.
- Operadores de mutação de `switch's`

O funcionamento da ferramenta Jumble é bastante simples, são necessários dois *input's*: a classe original compilada e um caso de teste em Junit, ao contrário da maioria das ferramentas, as mutação realizadas são feitas no *bytecode* gerado pelo compilador Java e no código fonte, eliminando assim a necessidade de compilar o código alterado sempre que é aplicado um operador de mutação. A execução começa sempre por realizar os testes definidos na classe de teste. Se todos os testes passarem com sucesso, passa para uma segunda fase onde irão sendo aplicados operadores de mutação até que todos os possíveis operadores de mutação tenham sido aplicados.

Lista 2.1: Exemplo da execução da ferramenta Jumble

```
java -jar jumble.jar --classpath=. example/Mover
Mutating example.Mover
Tests: example.MoverTest
Mutation points = 10, unit test time limit 2.04s
.....M FAIL: example.Mover:27: - -> +
..M FAIL: example.Mover:30: + -> -

Score: 80%
```

Na lista 2.1 encontra-se o resultado da execução da classe descrita no anexo L, indicando que foram realizados dez mutantes no código em que dois não foram eliminados, gerando um *score* de 80%.

μJava

μJava Java (muJava) é um sistema de mutação para programas desenvolvidos em Java. Suporta operadores de mutação em métodos e classes. Os operadores de mutação em métodos,

tal como a ferramenta Jumble são aplicados directamente no bytecode, enquanto que os métodos mutação por classe obrigam, a que o código seja compilado.

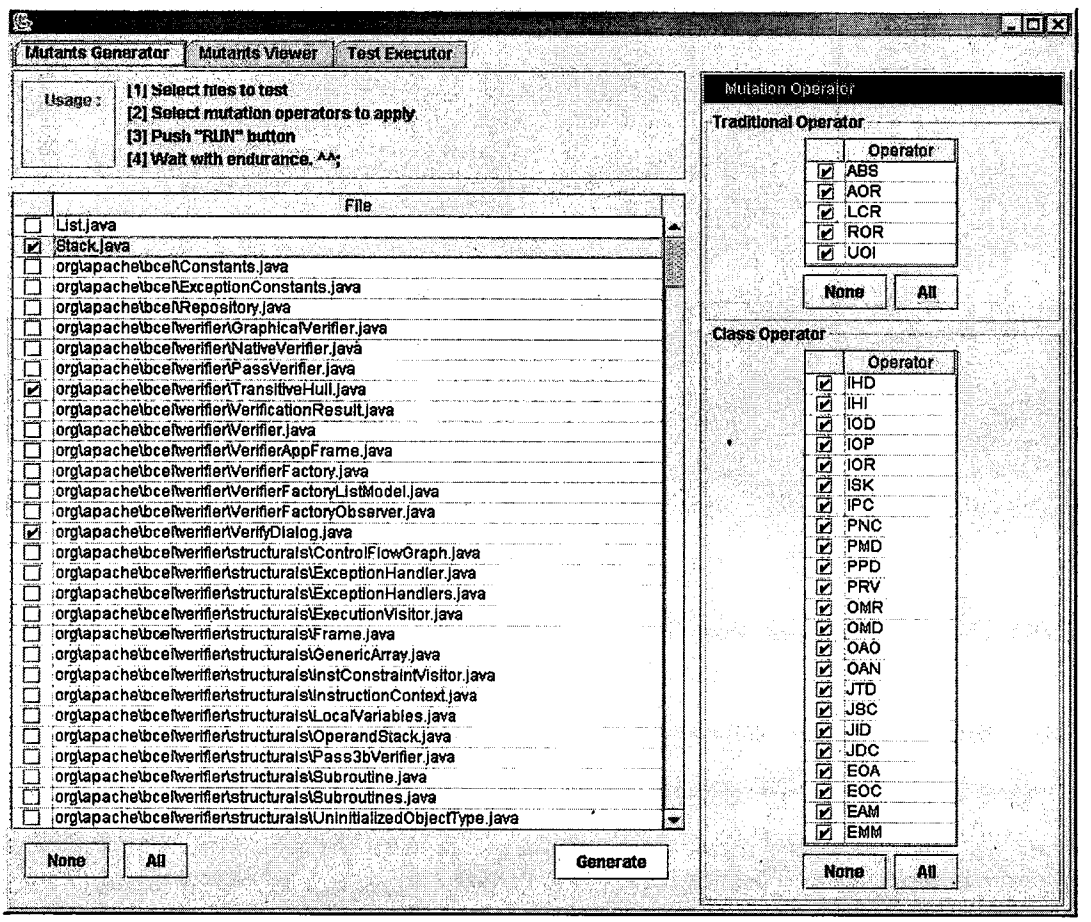

Figura 2.8: Componente gráfico para gerar mutantes em μJava

Considerações

Na tabela 2.29 encontra-se um sumário da comparação entre as duas ferramentas linguagens segundo o critério definido no ponto 2.4 do presente trabalho.

Jumble possui uma arquitectura bastante simples e intuitiva, possuindo integração com Junit. Outra grande vantagem é o facto de não haver compilação dos mutantes gerados pois os operadores são realizados ao nível to *bytecode*. No entanto, o número de operadores é reduzindo, não havendo actualmente nenhum operador de mutação de classe.

Por outro lado, a ferramenta μJava é uma ferramenta muito completa em relação aos operadores de mutação disponibilidades. Surgiu no meio académico, havendo um vasto número de artigos que validam o seu funcionamento. No entanto é mais complexa de utilizar, e não foi possível encontrar o código fonte da mesma, condição essencial para a utilização. Também não tem suporte para a ferramenta Junit, estando os testes no código da classe a testar.

Nome	Código aberto	Operadores de mutação	Arquitectura	Documentação	Integração com Junit
Jumble	Sim	Métodos	Simples	Pouca	Sim
μJava	[1]	Ambos	Média	Pouca	Não

Tabela 2.29: Comparação das duas ferramentas analisadas

Com base nesta análise, foi decidido utilizar como base para a geração/execução dos testes por mutação a ferramenta Jumble.

2.5 Computação natural

O ser humano sempre teve grande curiosidade em tentar perceber e imitar comportamentos e características dos animais e plantas que o rodeiam, com o objectivo de criar ferramentas que o ajudem a ultrapassar obstáculos, existindo actualmente inúmeros artefactos desenvolvidos com inspiração na natureza (ver figura 2.9). Além de artefactos, a natureza permitiu também o desenvolvimento de teorias para descrever o seu comportamento. As leis de Newton são um bom exemplo [EB99].

A ciência da computação não demorou muito tempo a utilizar esses comportamentos e processos. A computação natural é a ciência que tenta implementar sistemas computacionais baseados em comportamentos verificados na natureza. Existem três grandes disciplinas na computação natural[D99]:

- Estudos sobre a natureza através da computação. Envolve a utilização de mecanismos computacionais para analisar comportamentos naturais e processos biológicos.
- Computação com mecanismos naturais. É um paradigma de computação recente onde mecanismos naturais (por exemplo cadeias de DNA) são utilizados como estruturas de dados para o desenvolvimento de *computadores naturais.*

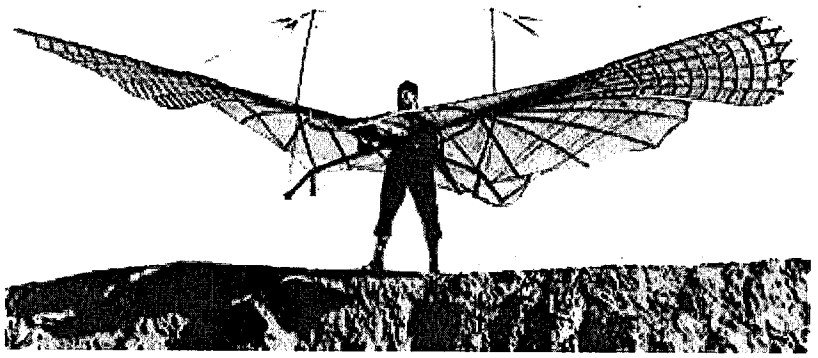

Figura 2.9: Prototipo de um avião baseado na imitação do comportamento de aves

- Computação inspirada na natureza (descrito na secção 2.5.1).

2.5.1 Computação inspirada na natureza

Com a descoberta de vários princípios e teorias da natureza, verificou-se que era possível usar estes princípios e teorias na ciência da computação para resolver problemas. Existem três grandes linhas de investigação na área da computação inspirada na natureza[D99] :

- Redes Neuronais.
- Algoritmo de Optimização por Bando de Partículas (discutido na secção 2.5.1).
- Sistemas Imunológicos.
- Algoritmos Genéticos.

Um dos primeiros trabalhos pioneiros na Computação inspirada na natureza foi o de McCulloch & Pitts [WM43] ao introduzir o conceito de neurónio na matemática que deu origem à disciplina de Redes Neuronais.

Mais tarde, por volta da década de 60, surgiu uma segunda linha de investigação: denominado por *algoritmos genéticos*. Esta disciplina baseou-se no conceito de evolução (biologia): o processo através no qual ocorrem mudanças ou transformações em seres vivos ao longo do tempo que dão origem a espécies novas.

Em meados da década de 90, Kennedy e Eberhart publicaram o artigo *Particle Swarm Optimization* [JK95], dando origem ao Algoritmo de Optimização por Bando de Partículas .

Optimização por Bando de Partículas

São inúmeras as espécies que tiram partido do facto de viverem em grupos sociais (*bandos*). Este modo de viver aumenta a probabilidade de acasalamento, facilita a recolha de alimentos, reduz a probabilidade de ataque de predadores e permite a divisão de trabalho. Durante anos, os cientistas estudaram formigas, abelhas e vespas devido à sua tremenda eficiência social [EB01].

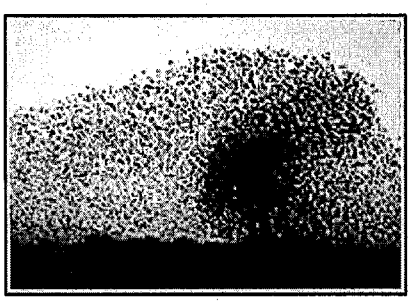

Figura 2.10: Estorninhos voando em grupo para se protegerem dos ataques dos predadores

A inteligência de bando pode ser definida como uma tentativa de projectar algoritmos ou dispositivos distribuídos para encontrar soluções a problemas, soluções estas que são inspiradas no comportamento em bando de insectos sociais e outras sociedades animais [EB99].

O termo *bando* é utilizado nessas técnicas para se referir de forma genérica a um conjunto estruturado de agentes capazes de interagir entre si. O exemplo clássico de um *bando* é um enxame de abelhas, embora a metáfora do *bando* possa estendida a outros sistemas com uma arquitectura similar, um bando de aves é um *bando* onde os agentes são as aves, um engarrafamento pode ser considerado um *bando* onde os agentes são carros, uma multidão é um bando de pessoas, um sistema imunológico é um bando de células e moléculas, e uma economia pode ser encarada como um bando de agentes económicos [EB01].

A inteligência colectiva pode ser definida como uma propriedade de sistemas compostos por agentes em bando com pouca inteligência e capacidade individual limitada, capazes de apresentar comportamentos inteligentes [TW98] quando operam em conjunto.

A inteligência colectiva é obtida pelas seguintes propriedades[EB01]:

- **Proximidade**: os agentes devem ser capazes de interagir entre si;
- **Qualidade**: os agentes devem ser capazes de avaliar seus comportamentos;

- **Diversidade:** permite ao sistema reagir a situações inesperadas;
- **Estabilidade:** nem todas as variações ambientais devem afetar o comportamento de um agente;
- **Adaptabilidade:** capacidade de se adequar a variações ambientais.

Estes comportamentos sociais inspiraram o desenvolvimento de diversas ferramentas computacionais para a solução de problemas e estratégias de coordenação e controle de robôs.

O Algoritmo de Optimização por Bando de Partículas (*Particle Swarm Optimization*) foi apresentado pela primeira vez por Kennedy e Eberhart, sendo baseado no comportamento de pássaros e peixes [JK95] quando actuam em bando para procurar alimento e fugir dos predadores.

É um algoritmo de pesquisa baseado em populações onde os indivíduos são designados por **partículas** são agrupados em *bandos* [Bri02]. Cada partícula é representada por uma matriz (ver figura 2.11) constituída por coordenadas que representam a posição da partícula. É essa posição que representa uma possível solução para um problema de optimização. As partículas percorrem livremente o espaço, ajustando a sua posição de pesquisa baseada na experiência adquirida e a transmitida pelos seus vizinhos com o objectivo de se deslocar para uma posição que estará mais próxima da solução óptima. Em muito semelhante aos algoritmos genéticos [BR98], o conjunto de indivíduos tende a preservar aquelas posições que determinam uma maior aptidão e a descartar as posições de menor aptidão.

$$p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_k \end{bmatrix}$$

Figura 2.11: Representação de uma partícula

Algoritmo de Optimização por Bando de Partículas é uma uma abstração do processo natural onde, a procura pela posição mais apta é a busca de uma solução óptima para um problema onde, o conjunto de possíveis posições dos indivíduos é o espaço de busca do problema e cada posição ocupada por um indivíduo representa uma possível solução para o problema.

Neste algoritmo, cada indivíduo é designado por **partícula** e está sujeito a sofrer deslocações num espaço multidimensional que representa o seu espaço de confiança.
Para esta deslocação, é associado um vector de velocidade às partículas representado na figura 2.12

$$v_p = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ p_k \end{bmatrix}$$

Figura 2.12: Representação do vector de velocidade de uma partícula

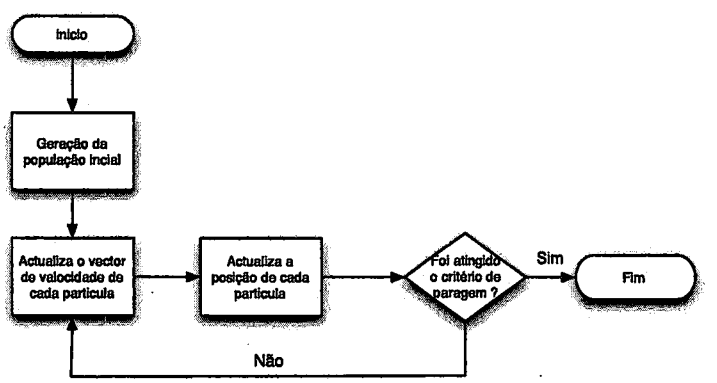

Figura 2.13: Fluxograma do Algoritmo de Optimização por Bando de Partículas

O algoritmo pode ser descrito como:

1. Gera-se uma população inicial onde a posição de cada indivíduo representa uma solução possível para o problema. A escolha inicial é feita de forma aleatória num espaço de pesquisa para o problema em que a posição é avaliada por uma função de aptidão (*fitness*).

2. Gera-se uma matriz de deslocação para cada indivíduo de forma aleatória e que todos os indivíduos se desloquem das suas posições iniciais.

3. Para cada uma das partículas, define-se a melhor posição até ao momento como sendo a inicial.

4. Repete-se o seguinte enquanto não se verificar o critério de paragem:

   (a) Actualiza-se o valor da função de aptidão de cada partícula p_i usando para tal a função de aptidão f e a posição actual da partícula.

   (b) Actualiza-se o valor da melhor posição até ao momento da partícula.

   (c) Actualiza-se a velocidade de cada partícula.

   (d) Actualiza-se a posição de cada partícula.

No anexo P [Hu07] encontra-se um pseudo-algoritmo onde, na lista P.2 estão definidas duas funções: (1) e (2) responsáveis respectivamente por encontrar a próxima posição da partícula e por deslocar a partícula para a nova posição.

O cálculo da nova posição pode ser interpretado de forma geométrica como a soma de dois vectores: um vector cognitivo que representa a experiência individual onde está a solução $(c1 * rand() * (pbest[] - present[])$ e um segundo vector social que representa a experiência devolvida pelo bando da posição da solução $(c2 * rand() * (gbest[] - present[]))$.

Analisemos o seguinte exemplo:

$$present = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, pbest = \begin{bmatrix} 2 \\ 5 \end{bmatrix}, gbest = \begin{bmatrix} 7 \\ 1,5 \end{bmatrix}, v = \begin{bmatrix} 0,5 \\ 1 \end{bmatrix}$$

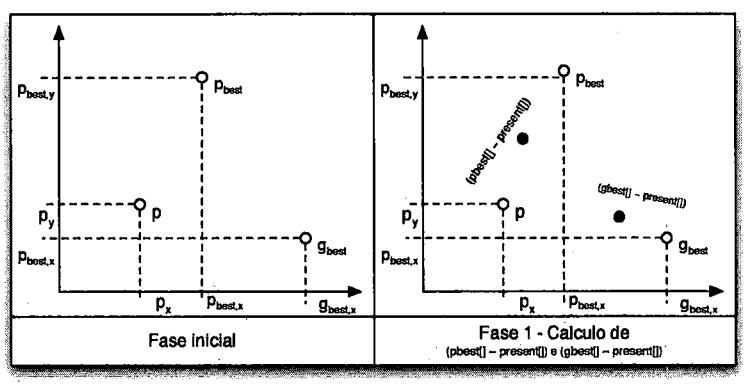

Figura 2.14: Representação gráfica do funcionamento do Algoritmo de Optimização por Bando de Partículas (Fase 0 e 1)

$$pbest - present = \begin{bmatrix} 1 \\ 4 \end{bmatrix},$$

$$gbest - present = \begin{bmatrix} 6 \\ 0,5 \end{bmatrix} \text{ (Fase 1)}$$

Depois de calculados os vectores $pbest - present$ e $gbest - present$ (ver figura 2.14), é aplicado a cada um dos vectores um coeficiente de mutação ($c1 * rand()$ e $c2 * rand()$ respectivamente). Ficando então com:

$$c1 * rand() * (pbest - present) = \begin{bmatrix} 0,8 \\ 2,7 \end{bmatrix},$$

$$c2 * rand() * (gbest - present) = \begin{bmatrix} 5 \\ 0,5 \end{bmatrix} \text{ (Fase 2)}$$

A representação gráfica encontra-se na figura 2.15.

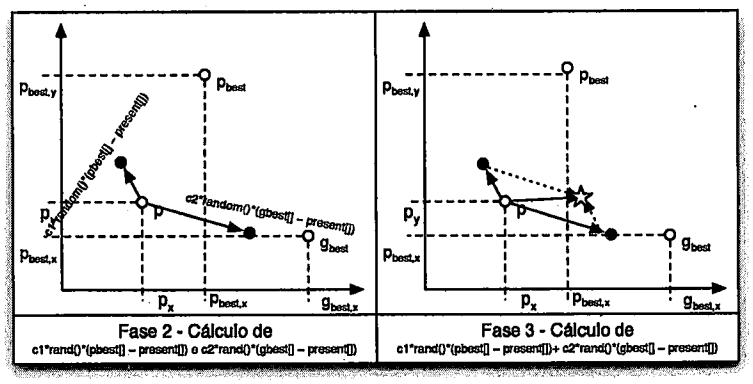

Figura 2.15: Representação gráfica do funcionamento do Algoritmo de Optimização por Bando de Partículas (Fase 2 e 3)

Após ter sido calculado o novo vector de velocidade v:

$$c1 * rand() * (pbest - present) + c2 * rand() * (gbest - present) = \begin{bmatrix} 5,8 \\ 3,2 \end{bmatrix} \text{ (Fase 3)}$$

fica-se com a relação:

$$v + c1 * rand() * (pbest - present) + c2 * rand() * (gbest - present) = \begin{bmatrix} 6,3 \\ 4,2 \end{bmatrix} \text{ (Fase 4).}$$

Este novo vector é aplicado à posição p, fazendo com que a partícula se desloque. (ver figura 2.16 $v + p = \begin{bmatrix} 7,3 \\ 5,2 \end{bmatrix}$ (Fase 5)

Aplicando sucessivamente este processo, as partículas vão convergindo para a solução óptima, simulando o comportamento de um bando (ver figura 2.17).

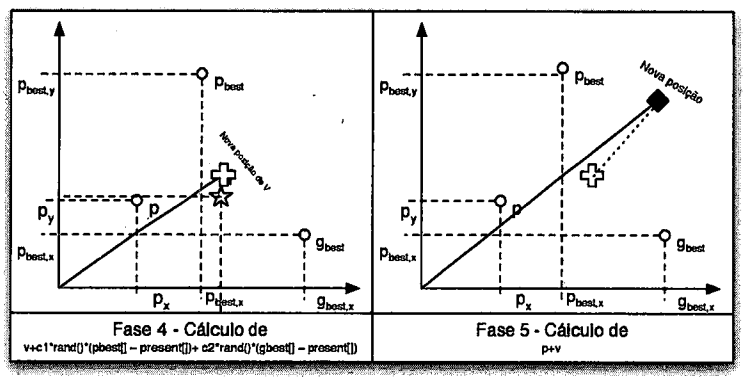

Figura 2.16: Representação gráfica do funcionamento do Algoritmo de Optimização por Bando de Partículas (Fase 4 e 5)

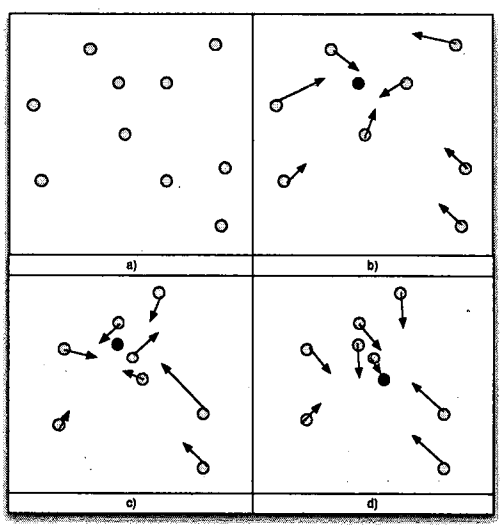

Figura 2.17: Representação gráfica do funcionamento do Algoritmo de Optimização por Bando de Partículas

Este algoritmo fornece um mecanismo equilibrado entre diversificação e a intensificação. A diversificação é fornecida pela utilização da velocidade anteriormente calculada, a intensificação é conseguida por $c1 * rand() * (pbest - present) + c2 * rand() * (gbest - present)$.

Em relação às constantes $c1$ e $c2$, estas são usadas para controlar em que é que a partícula acredita mais: na posição que ela já conhece como sendo a melhor ou na posição que o bando encontrou como melhor. Se $c1 = c2$, a partícula não tem preferencia. Caso $c1 > c2$, o algoritmo dá mais peso à posição cognitiva, fazendo com que as partículas se desloquem a velocidade maior para a solução óptima encontrada pela partícula. Para $c2 > c1$, as partículas terão mais tendência a acreditar que a posição encontrada pelo bando é melhor do que aquela que a partícula já presenciou, levando a que a um deslocamento mais rápido para a solução óptima encontrada pelo bando.

Uma grande vantagem deste algoritmo é a facilidade de implementação, sendo necessários configurar poucos parâmetros, em comparação com outros algoritmos naturais, tais como os algoritmos genéticos[MSS06].

Ambientes Discretos Algoritmo de Optimização por Bando de Partículas foi inicialmente desenhado para operar em ambientes contínuos, no entanto este também pode ser muito eficiente na resolução de problemas em ambientes discretos, fazendo algumas modificações ao algoritmo. Para ambientes discretos a posição das partículas poderá ser arredondada para o valor inteiro mais próximo do valor calculado pelo algoritmo.

Implementações em Java Foram analisadas duas implementações deAlgoritmo de Optimização por Bando de Partículas nos seguintes critérios:

- Possuir licença de código aberto.
- Ser fácil de usar.
- Permitir definir o domínio de pesquisa.

JSwarm-PSO JSwarm-PSO[Cin07] é uma implementação de Algoritmo de Optimização por Bando de Partículas para Java desenvolvido por Pablo Cingolani. É uma implementação

muito simples de usar e que pelo facto de ser distribuída com licença GPL. A sua maior vantagem é a simplicidade de programação, no entanto apenas foi desenhada para domínios reais.

CILib

Mais do que uma implementação de Algoritmo de Optimização por Bando de Partículas , CILib[Pee07] é uma ferramenta para desenvolver programas que utilizem algoritmos naturais, tais como: Algoritmo de Optimização por Bando de Partículas , Redes Neuronais, Fuzzy Logic e Sistemas Imunológicos. Tal como o JSwarm-PSO, esta ferramenta é distribuída com licença GPL.É mais difícil de perceber e de utilizar, no entanto não impõe nenhuma restrição ao domínio das partículas (ao contrário da JSwarm-PSO), permitindo que este seja definido pelo programador.

Com base nesta analise, foi decidido que iria ser usado a ferramenta CILib.

2.6 Programação por Contrato (Design By Contract)

No dia a dia todos nós possuímos contratos: contratos de trabalho, contratos de arrendamento, contratos de manutenção, etc. Todos são realizados por duas entidades onde uma das partes realiza uma tarefa em prol da outra. Existindo boa fé em ambas as partes, ambas esperam beneficio na relação, sujeitando-se a restrições. Os benefícios de uma são as obrigações da outra. O objectivo de um contrato não é mais do que especificar os benefícios e as obrigações de cada uma das partes.

Esta ideia de relação de entidades pode ser transposta para o contexto de software. Para que execução de uma determinada tarefa dependa de uma outra, é necessário especificar com detalhe a relação entre o cliente (aquele que invoca) e o contratado (a tarefa que é invocada). É este o conceito principal da Programação por Contrato (*Design by Contract*, DBC). É uma metodologia de desenvolvimento que visa a construção de sistemas mais confiáveis, baseando-se na ideia da existência de um contrato entre dois módulos de software: um módulo denominado **cliente** que obrigatoriamente cumpre um conjunto de requisitos antes de invocar o segundo módulo que designado por **fornecedor** que se compromete a devolver algo que respeita um conjunto de requisitos definidos ao inicio.

As princípais vantagens da utilização desta metodologia são[Mey07]:

- Uma melhor documentação pois o comportamento dos métodos são descritos numa linguagem formal.
- Os processos de teste e depuração tornam-se mais simples pois quando uma das asserções é quebrada, sabe-se exactamente qual foi e qual foi a parte que as quebrou.
- Possibilita ter uma visão geral de como é que o software foi desenhado e construído;
- Permite que exista uma melhor compreensão do fluxo do programa;
- Diminui o número de erros possíveis no código.
- O controlo de excepções torna-se mais seguro.

Embora a origem desta metodologia remonte à linguagem Eiffel [Mey07], nos dias de hoje a maioria das linguagens implementam este conceito, de forma embebida ou através de API's externas.

Em DBD, o mecanismo para expressar as condições nas quais o contrato é estabelecido possuem o nome de **asserções**, existindo três tipos: **pré-condições**, **pós-condições** e **invariantes.** As pré-condições definem os direitos do fornecedor e as pós condições definem os deveres desse mesmo fornecedor. As asserções invariantes são propriedades do contrato que são sempre válidas em qualquer altura do mesmo.

2.6.1 Pré e Pós condições

As pré-condições são usadas para especificar quais as condições que são validas antes do método ser invocado, enquanto que as pós-condições qual o compromisso do método depois de ter sido invocado.

As asserções não devem descrever casos especiais mas sim situações esperadas resultantes do uso de um determinado método. Na lista 2.2 encontra-se um exemplo de pré-condições e pós condições para o construtor da classe Bag descrita no anexo L.1.

Lista 2.2: Exemplo de pré-condições e pós-condições

```
...
pre-condição: input.length>=0;
pos-condição: n=a.length;
```

```
...
```

Qualquer violação das asserções deve ser considerada como um erro no software. Duas situações podem acontecer[Mad03]:

1. A violação de uma pré-condição indica que existe **um erro no método cliente**;
2. A violação de uma pós-condição indica que existe **um erro no método fornecedor.**

2.6.2 Invariantes

Embora as pré-condições e pós-condições forneçam mecanismos que permitam estabelecer os direitos e obrigações de cada método de forma individual, é necessário também existir algum mecanismo para representar condições globais, nos quais todos os métodos a devem preservar. Para tal é usado o conceito de invariante: propriedades que têm de ser validas em todos os estados *visíveis* dos objectos de uma determinada classe. Um exemplo de uma asserção invariante pode ser encontrado no exemplo no anexo I: a variável n não pode ter um valor superior ao tamanho do vector a (ver lista 2.3).

Lista 2.3: Exemplo uma asserção invariante

```
...
invariante: a.length >= n;
...
```

Para uma correcta utilização das invariantes, é necessário:

- A invariante deve ser satisfeita logo após a criação da variável de instância. Isto quer dizer que os construtores devem estabelecer essa invariante;
- Qualquer método não estático deve garantir que a invariante é garantida depois da sua execução (se esta o for antes da sua invocação), mesmo que terminem de forma anormal (lançamento de uma excepção)

2.6.3 Hierarquia

Uma das consequências directas a utilização de DBC em linguagens orientadas por objectos é um melhor controle da herança e hierarquia, conceitos fundamentais nestas linguagens.

A maioria das linguagens permite que exista redefinição de métodos, permitindo que estes possuam comportamentos diferentes consoante a subclasse onde foram redefinidos. Pode acontecer também que só em tempo de execução se perceba que um determinado método foi redefinido.

Estes são importantes em DBC pois é necessário garantir que ao haver a redefinição da especificação de um determinado método, esta não seja incompatível com a especificação formal realizada no método da superclasse.

Existem duas situações em que as condições da superclasse podem ser quebradas:

1. Seja definido uma pré-condição mais restritiva:

```
...
pré-condição da superclasse: a.length >= n;
...
pré-condição da classe: a.length == n;
...
```

2. Seja definido uma pós-condição menos restritiva:

```
...
pós-condição da superclasse: a.length == n;

...
pós-condição da classe : a.length >= n;
...
```

Estas situações não podem surgir pois poderão gerar inconsistências, no entanto o contrário é válido: uma redefinição pode tornar mais fraca uma pré-condição ou fortalecer uma pós-condição.

Em relação às asserções invariantes, estas são sempre passadas aos seus descendentes

2.6.4 Controlo de Excepções

Esta metodologia impõe que a execução de um método seja apenas realizado quando as pré-condições desse método sejam verificadas caso contrário não é garantido que o método realize qualquer operação. Quando isto acontece, o método a invocar é obrigado a ter dois comportamentos: terminar e devolvendo alguma coisa (caso a sua definição o obrigue) ou gerar uma excepção. Terminando com sucesso, este é obrigado a satisfazer as pré-condições.

Caso o método gere excepções, a metodologia permite que sejam definidas um conjunto de pós-condições excepcionais.

Lista 2.4: Exemplo de uma asserção exceptional da lista 2.7

```
...
excepção: IllegalArgumentException;
...
```

2.7 Implementações de DBC para a linguagem Java

Existem alguns factores importantes devem ser considerados na escolha de uma linguagem de descrição de comportamento e especificação de propriedades:

- A linguagem deverá ser o mais parecida com a linguagem Java pois o utilizador final da ferramenta resultante desta dissertação é o programador. Quanto maior for a semelhança entre as linguagens, mais fácil será para o programador aprender a usar a ferramenta. É também importante o programador ter controlo total sobre o novel de abstracção utilizado no processo de verificação.

- Deve ser uma linguagem de anotação para permitir que as asserções possam estar junto do código, diminuindo assim possíveis problemas resultantes da falta de sincronização entre o código fonte e o ficheiro da linguagem.

- Deve possuir um conjunto de ferramentas que permita a geração/execução de testes unitários baseados na descrição do comportamento realizada pelo programador.

- Deve suportar o conceito de hierarquia. Permitir redefinições de contratos é essencial em linguagens orientadas a objectos como é o caso do Java.

- Possuir código aberto.

2.7.1 C4J

Contracts for Java (C4J) é uma *framework* de DBC para Java cuja característica principal é a simplicidade de utilização.

As asserções (invariantes, pré-condições e pós-condições) são implementadas numa classe à parte (que representam o contracto) que é associado à classe em questão através de uma simples anotação `@ContractReference(contractClassName = "NomeDoContrato"` colocada na definição da classe.

As características desta *framework* são[C4J07]:

- Asserções invariantes;
- Asserções de pré-condições;
- Asserções de pós-condições;
- Permite que seja feito depuração dos contratos;
- Possui mecanismos de acesso a membros privados;
- Possui mecanismos para aceder ao valor de retorno dos métodos;
- Permite que os contratos sejam refinados por implementações de subclasses;
- Permite herança de contratos;
- Permite que sejam definidos contratos em interfaces que são automaticamente herdados em todas as classes que implementem esse método;
- Permite que sejam criados contratos para métodos abstractos que são automaticamente herdados nas classes quem os implementam;
- Possui mecanismos relatórios avançados;

Lista 2.5: Exemplo da definição de pré-condições e pós-condições

```
@ContractReference(contractClassName = "DummyContract")
  public class Dummy
  {
    public double divide(double x, double y)
    {
      return x / y;
    }
  }
```

Lista 2.6: Exemplo da pré-condições e pós-condições para o construtor da classe

```
public class DummyContract
 {
   public void classInvariant()
   {
     // Nothing to do here
   }

   public void pre_divide(double x, double y)
   {
     assert y != 0;
   }
 }
```

2.7.2 Java Modeling Language

JML (Java Modeling Language) é uma linguagem de especificação formal do comportamento da interface para Java. JML é utilizada como uma poderosa ferramenta de Design by Contract (DBC) em Java, possuindo as notações essenciais usadas em Design By Contract desenvolvida por Gary T. Leaven, professor de Computer Science na Iowa State University.

JML é uma linguagem de anotação sob a forma de comentário, usando para tal o caracter especial '@': `//@invariant fBufferSize>=0;`. Pelo facto de serem comentários, não alteram em nada o código compilado.

Asserções

As asserções são as peças chave para a definição do comportamento de um determinado método. As condições são expressões boleanas do tipo `x >0` ou `x == y` construídas usando a linguagem Java, tornando muito simples a sua aprendizagem.

A definição de uma pré-condição faz-se através do uso da clausula **`requires`** seguido de um predicado, no qual os parâmetros do método a ser especificado podem ser usados. Podem ser usadas múltiplos predicados **`requires`**. Na lista 2.7 encontra-se um exemplo de um asserções em JML. A clausula **`ensures`** especifica a propriedade que deve ser satisfeita no final da execução de um determinado método, definido assim uma pós-condição. As regras

de utilização são semelhantes à clausula `requires`.

Lista 2.7: Exemplo de pré e pós-condições em JML

```
public class Pessoa {
private String nome;
private int peso;
/*@ requires peso + kgs >= 0;
@ ensures kgs>=0 && peso==\verb!\!old(peso) + kgs;
@ signals (IllegalArgumentException e) kgs < 0;
@*/
public void adicionaKgs(int kgs) throws IllegalArgumentException{
if(kgs < 0)
throw new IllegalArgumentException();
this.peso += kgs;
}
}
```

Em relação às invariantes, a linguagem JML implementa as asserções invariantes em todos os *estados visíveis* do objecto[Rie07].

- No fim de cada construtor.
- No inicio do componente que remove o objecto da memória.
- No inicio ou no fim da invocação de todos os métodos definidos na classe.

Lista 2.8: Exemplo de invariantes em JML

```
public class Buffer
{
//@invariant fBufferSize>=0;
private int fBufferSize;
//@constraint data.length>=\old(data.length)
private Object[] data;
public /*helper */ voidskipMe(){...}
public void checkMe(){...}
}
```

A definição de invariantes é feito através das clausulas `invariant` e `constraint`. As asserções definidas pela clausula `invariant` serão validadas no inicio, enquanto que as asserções definidas pela clausula `constraint` serão validadas no fim da execução.

Por fim, as asserções de excepção são definidas através da clausula `signals`: `signals (IllegalArgumentException e) kgs < 0;`. Através desta a clausula define-se o comportamento esperado do método quando uma determinada excepção é gerada. No exemplo descrito na lista 2.7, a asserção `signals (IllegalArgumentException e) kgs < 0;` define o comportamento do método deve retornar a variável `kg` menor que `0`. Uma das vantagens desta linguagem é o vasto número de funcionalidades. Uma delas é dar a possibilidade de ser definido cenários caso o método tenha um comportamento normal e um outro caso ocorra alguma excepção. Isto é feito através do uso das clausulas `normal_behavior` e `exceptional_behavior`. Na lista 2.9 encontra-se um exemplo de utilização desta funcionalidade.

Lista 2.9: Exemplo de utilização de `normal_behavior` e `exceptional_behavior`

```
/*@ public normal_behavior
 @ requires !theStack.isEmpty();
 @ assignable size, theStack;
 @ ensures theStack.equals(\old(theStack.trailer()));
 @ also
 @ public exceptional_behavior
 @ requires theStack.isEmpty();
 @ assignable \nothing;
 @ signals (BoundedStackException);
@*/
public void pop( ) throws BoundedStackException;
```

Herança e hierarquia

Tal como na linguagem Java, em JML, a especificação pode ter uma visibilidade, que varia de `public` a `private`, passando por `protected` e `default`. As especificações devem seguir uma regra básica para a utilização de visibilidade: uma especificação não pode ter maior visibilidade que a sua implementação, ou seja, não é possível ter uma especificação publica para um método publico, e nem uma especificação protected para um método privado.

Esta ferramenta permite também que determinadas invariantes não sejam validadas na invocação de métodos através do uso da expressão `/*helper */`

Ferramentas

Uma das grandes vantagens da JML é o conjunto importante de ferramentas que são fornecidas: JML checker, JmlRac (JML Runtime Assertion Checker) JmlUnit e JmlDoc

JML checker É o componente principal desta *framework* pois é responsável por validar a sintaxe e a semântica das anotações.

JmlRac (JML Runtime Assertion Checker) Possui a responsabilidade de validar as anotações em tempo de execução, detectando se algumas das asserções não são validadas de forma transparente para o utilizador. Na lista 2.10 encontra-se encontra-se representado um exemplo de relatório gerado por esta ferramenta.

Lista 2.10: Exemplo da execução da ferramenta de Jmlrac

```
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
by method Bag. Bag regarding specifications at
File "Bag.java", line 20, character 29 when
' input' is null
at Bag.checkPre$$init$$Bag(Bag.java:45)
at Bag.<init>(Bag.java:22)
at Bag.main(Bag.java:33)
```

JmlUnit A ferramenta JmlUnit permite a geração de testes unitários de interfaces para serem executados na *framework* de testes unitários para Java Junit, eliminando a necessidade do programador se preocupar com esta tarefa.

As classes de teste geradas validam se o método em questão satisfaz as condições definidas nas anotações, baseando-se nos seguintes princípios:

1. Quando é violada uma pré-condição, não se pode concluir que o método possui algum erro.

2. Quando são verificadas todas as pré-condições e alguma das invariantes ou pós condições falha, o método não teve o comportamento esperado, logo podemos concluir que o código possui um erro, assumindo claro que as asserções estão correctas.

3. Se um determinado método em que são válidas todas as pré-condições e invariantes ao invocar um segundo método faz violar as pós-condições deste, é porque existe um erro no código.

JmlDoc Esta ferramenta possui um comportamento semelhante à javadoc da linguagem Java, permitindo gerar documentação em formato HTML baseada nas anotações existentes no código. Na figura 2.18 encontra-se encontra-se representado um exemplo da documentação gerada por esta ferramenta.

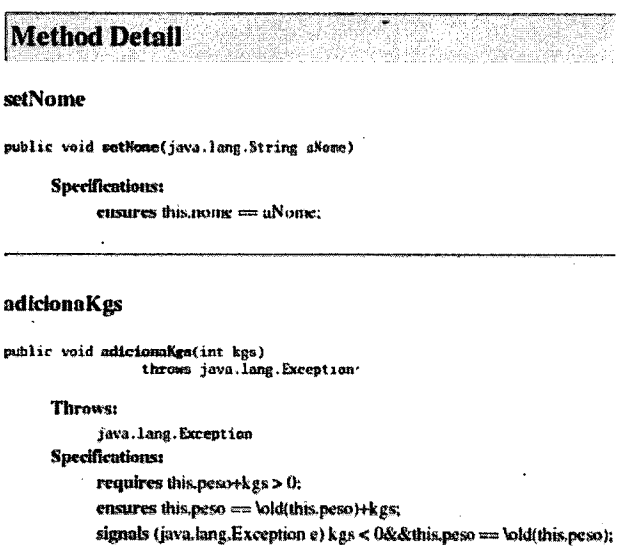

Figura 2.18: Exemplo da documentação gerada pela ferramenta jmlrac

2.7.3 Contract4J

Contract4J é uma ferramenta que implementa Design By Contract através da utilização de asserções, usando para tal anotações em Java, implementado os três tipos de asserções: pré-condições, pós-condições e invariantes.

Uma das diferenças das restantes implementações analisadas tem a ver com o facto de não haver necessidade de compilar o código pois as asserções são interpretadas em tempo de execução.

Invariantes

As invariantes são expressas utilizando a anotação `@Invar`, definindo dois elementos:

`value` Expressão booleana que representa a asserção a realizar

`message` Mensagem que irá ser devolvida ao utilizador caso a invariante deixe de ser válida.

Na lista 2.11 pode-se encontrar um exemplo de uma asserção invariante em Contract4J

Lista 2.11: exemplo de uma asserção invariante em Contract4J

```
@Invar("fBackupBuffer!=null")
public classB uffer {
private Object[] fBackupBuffer=newObject[]{};
@Invar(value="fBufferSize>=0", message="Illegalbuffer ?size.")
private in tfBufferSize=0;
}
```

Pré e Pós condições

Para ser definido uma pré-condição é usado a anotação `@Pre` cujo argumento é uma cadeia de caracteres que representa em Java código a ser avaliado e que representa essa pré-condição que tem de devolver um booleano (ver listagem 2.12).

Lista 2.12: exemplo de uma pré-condição em Contract4J

```
@Pre("item!=null && $this.isFull() == false")
public void add(Objectitem) {...}
```

No caso das pós-condições é usado a anotação `@Post` tamém com um argumento que representa em Java o código a ser validado. Na lista L.2 encontra-se um exemplo de uma pós condição

Lista 2.13: exemplo de uma pós-condição em Contract4J

```
@Post("$result!=null&&$old(fBufferSize) 1==fBufferSize")
public Object removeLast() {...}
```

Hierarquia

Esta ferramenta possui suporte limitado para o conceito de Hierarquia visto o mecanismo usado para definir asserções ser através do uso de anotações disponíveis na linguagem Java, versão 1,5 e não existe o conceito de Hierarquia em anotações nesta versão de Java

2.7.4 Considerações

Na tabela 2.7.4 encontra-se um sumário da comparação entre as três linguagens segundo o critério definido em 2.7.

Nome	Tempo de aprendizagem	Asserções junto do código	Geração de testes unitários	Código aberto
C4J	Médio	Não	Não	Sim
JML	Curto	Sim	Não	Sim
Contract4J	Médio	Sim	Não	Sim

Tabela 2.30: Comparação das três linguagens analisadas

Das três linguagens e respectivas implementações analisadas a JML foi aquela que mais funcionalidades, maior documentação e maior ferramentas disponíveis, sendo a única que cumpria os requisitos necessários para implementar a ferramenta Zoonomia, objectivo desta tese. JML distingue-se das restantes pela riqueza da linguagem e por existir mais do que um mecanismo para implementar uma determinada asserção.

2.8 Ferramentas de detecção de erros no software

Começa a existir um número considerável de ferramentas de automatização de testes, grande parte orientada para o teste estático, tentando encontrar erros no código baseado em heurísticas.

2.8.1 Jlint

O Jlint [Kni07a] é uma ferramenta orientada para a verificação estática de programas Java, tentando encontrar possíveis erros, problemas de inconsistência e sincronização através da análise de fluxo de execução.

É uma ferramenta *open source*, desenvolvida na linguagem *C* por Konstantin Knizhnik na Moscow State University é composta por dois módulos:

AntiC . Analisador sintáctico do programa. Os erros sintácticos capazes de serem detectados pelo AntiC são [Kni07b]:

Erros em tokens Erros causados por utilização de *tokens* errados, por exemplo o uso de `'l'` em vez de `'1'`

Prioridade de operadores Alguns operadores não possuem uma precedência clara, podendo levar o programador a cometer erros . `x || y && z` é uma expressão não muito clara pois o operador **AND** (`&&`) tem prioridade sobre o operador **OR** (||).

Conjunto de instruções . É prática corrente em Java agrupar um conjunto de instruções com o uso de { e }. Programadores menos experientes poderão escrever algo do género:

```
while (x != 0)
    x >>= 1;
    n += 1;
return x;
```

Quando na verdade pretendiam que a variável `n` fosse incrementada de um valor.

JLint Analisador semântico do programa. Extrai informação das classes Java através da análise de invocações de métodos em que é possível detectar a passagem de possíveis parâmetros nulos. Além disso, a partir das dependências entre as classes, Jlint constrói um grafo que é usado para detectar situações que podem causar *deadlock* em sistemas concorrentes. Esta ferramenta interpreta os *bytecodes* gerados pelo compilador de Java, extraindo informações que o levam a detectar erros pré-estabelecidos. As principais validações são[Kni07b]:

Sincronização de código . A Linguagem Java não possui mecanismos que validem possíveis erros de *deadlock* e *race conditions*.

Problemas de Herança O Jlint consegue detectar erros relacionados com herança de classes tal como *variable shadowing*.

Questões relacionadas com o fluxo de dados Jlint possui mecanismos que permitem analisar o fluxo de um determinado dado e conseguir por exemplo determinar o valor máximo e o valor mínimo de uma variável do tipo `Integer`. O Jlint também consegue detectar condições inválidas (ou válidas) como a seguinte:

```
public void foo(int x) {
    if (x > 0) {
        ...
        if (x == 0) //sempre falso
        {
        }
    }
}
```

JLint é uma ferramenta limitada pois carece de uma linguagem de anotação de código (ou qualquer outro mecanismo) que modele o comportamento do sistema ou que especifique as propriedades a serem verificadas. O número de erros possíveis de encontrar dependem sempre das propriedades do JLint. Além do mais, a maioria dos ambientes de programação para a linguagem Java (Eclipse, Netbeans, IntelliJ) já realizam validações sintácticas e semânticas em muito maior número do que esta ferramenta.

2.8.2 Perfect Developer

O Perfect Developer [Tec07] é uma ferramenta comercial, desenvolvida pela empresa Eschertech cuja principal característica consiste no facto de que o código final da aplicação deve ser uma implementação da sua especificação realizada na ferramenta.

Perfect Developer possui um verificador automático de teoremas que consegue comprovar a conformidade entre a especificação e os requisitos dessa linguagem. Depois desta validação é possível gerar o código pronto para compilação. Actualmente, a ferramenta permite gerar código para Java, ADA e C++, permitindo ainda que o código gerado possa ser alterado, e que sejam feitos pequenos ajustes pelo programador.

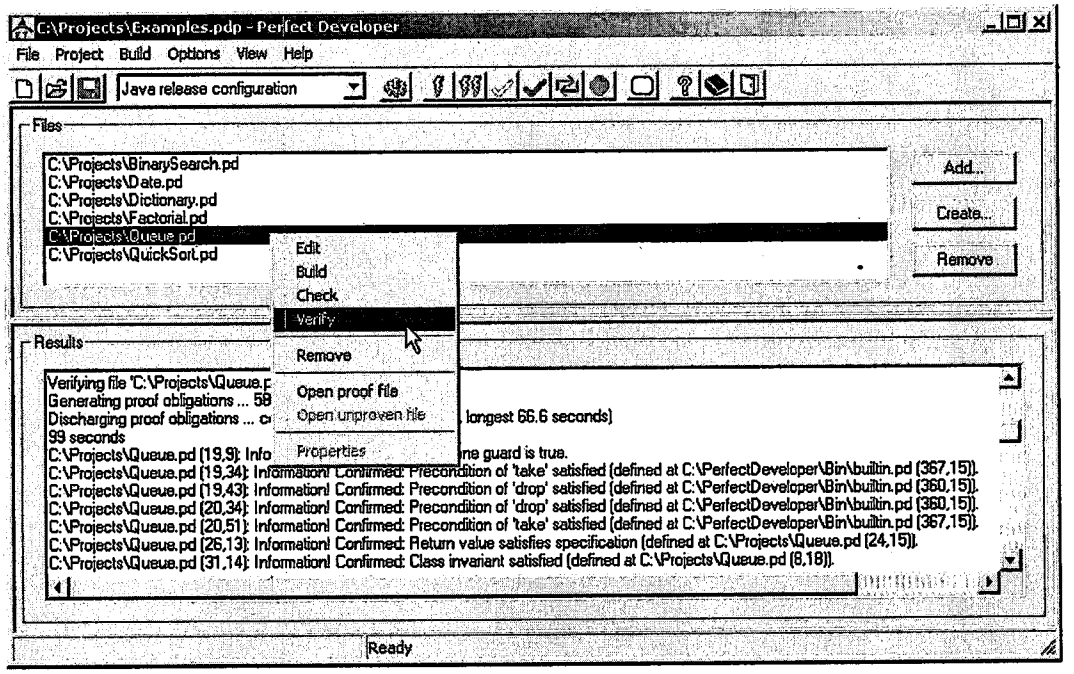

Figura 2.19: Ambiente gráfico do Perfect Developer

Uma outra funcionalidade muito interessante desta ferramenta é a possibilidade de importar modelos UML, pois permite gerar as estruturas necessárias para a implementação das especificações.

A principal desvantagem desta ferramenta é não possuir suporte para desenvolvimento de sistemas concorrentes, aspecto essencial no desenvolvimento de sistemas modernos.

Outra desvantagem é o facto de a ferramenta possuir uma linguagem própria, a *Perfect Language*, obrigando a que os programadores tenham de aprender uma linguagem nova.

2.8.3 Esc/Java2

Esc/Java2 (*extended static checking*) [Kin07] é uma ferramenta para verificação de programas desenvolvidos em Java, mantida pelo grupo KindSoftware.

É uma ferramenta de análise estática sobre o código, tentando procurar erros que não sejam detectados pelo compilador (referência a variáveis nulas, erros de indexação de arrays, erros de coerção de tipos, etc ...). Esta ferramenta usa um validador automático de teoremas, chamado Simplify3 [Kin07], para verificar as propriedades especificadas. Ao contrário da ferramenta Perfect Developer, esta utiliza a linguagem aberta (JML) para realizar a validação

do comportamento do código.

Além de verificar as inconsistências entre design e código, também fornece avisos sobre potenciais erros existentes na implementação, que podem gerar falhas em tempo de execução.

Na figura I.2 pode-se encontra o output gerado pela ferramenta ao ser feita a análise da classe que se encontra no anexo I.

2.8.4 Java PathFinder

Java PathFinder (JPF) [Pat07] é uma máquina virtual Java que tenta executar o programa de todas as formas possíveis (ao contrário das maioria das máquinas virtuais o faz de uma única forma), verificando violações de propriedades como *deadlocks* ou excepções não tratadas ao longo de todos os caminhos de execução.

Esta ferramenta conta também com várias técnicas de abstração para diminuir o problema do enorme espaço de estados possíveis[VHBP00].

JPF recebe como entrada os *bytecodes* executáveis do programa em causa, devolvendo um relatório com informação sobre possíveis erros existentes no mesmo.

Para tornar viável a verificação de modelos, directamente a partir do programa, são utilizadas heurísticas e abstrações de estados no processo de verificação.

A ferramenta também é capaz de verificar se as propriedades especificadas pelo utilizador são satisfeitas pelo programa. Como a técnica de verificação de modelos é feita directamente no código a partir de abstracções automáticas do mesmo, não existe um modelo de comportamentos do sistema descrito pelo programador. Este deve apenas implementar as propriedades (asserções) em classes e métodos. Estes métodos devem ser chamados em determinados pontos do código do programa onde se pretende que as propriedades sejam verdadeiras.

Tal como Esc/Java2, a ferramenta JPF não lê bytecodes gerados por um compilador Java diferente o compilador para o qual foi implementado. Assim sendo, nenhum programa implementado com novos recursos pode ser verificado.

Capítulo 3

Implementação

Este capitulo tem como objectivo fazer uma descrição detalhada da arquitectura na implementação da Zoonomia[1], nome dado à plataforma de automatização de testes unitários para a linguagem Java, objectivo desta tese.

3.1 Descrição da solução

A base da ferramenta Zoonomia é aplicação do Algoritmo de Optimização por Bando de Partículas para tentar encontrar o caso de teste com maior *mutation score*. Segundo Mattias Bybro[Bybro], se existir algum erro no código, este com grande probabilidade será encontrado por casos de teste com *mutation score* elevados.

A ferramenta implementada é divida em duas partes: uma parte de configuração de todo o ambiente necessário (Fase 1) e uma segunda de execução do Algoritmo de Optimização por Bando de Partículas .

Na figura 3.1 encontra-se a arquitectura da ferramenta apresentada, bem como as interacções possíveis entre cada um dos módulos que constituem a ferramenta.

[1] *Zoonomia or the laws of organic life* foi um livro publicado por Erasmus Darwin (avô de Charles Darwin), que para muitos foi considerado como das primeiras referências à Teoria da Evolução mais tarde apresentada pelo seu neto Charles

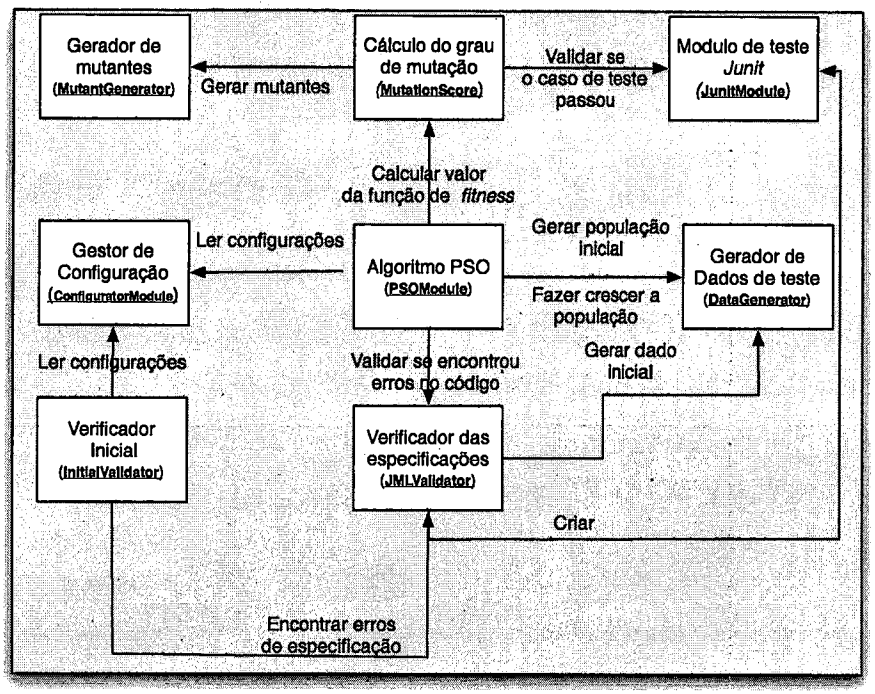

Figura 3.1: Arquitectura da Zoonomia

3.1.1 Gestor de Configuração

(*ConfigurationModule*)

O gestor de configuração é responsável por processar, interpretar e validar os parâmetros de configuração fornecidos pelo utilizador, permitindo que estes sejam acedidos de forma transparente pelos restantes módulos do sistema.

Os parâmetros necessários para o funcionamento da ferramenta e que devem ser fornecidos pelo utilizador são:

- As bibliotecas necessárias para poder executar o software a ser testado.
- O método a testar.
- Qual o critério de paragem do algoritmo de optimização por partículas.
- Se é para ser feita a verificação de redundância dos dados de teste gerados.
- O número de dados gerados inicialmente.

- Qual a população inicial.

Na lista A.1 encontra-se um exemplo da configuração possível. Estes parâmetros podem ser materializados numa bases de dados relacional ou em ficheiros. Na figura A.5 encontra-se o diagrama de sequência para o módulo de Gestão de Configuração.

3.1.2 Verificador Inicial

(*InitialValidator*)

O módulo Verificador Inicial é responsável por fazer validações iniciais do programa a testar, compilando o código fonte para tentar encontrar erros de sintaxe. A compilação do código é feito através do uso do compilador Java embebido na aplicação biblioteca `tools.jar`[Mic07]. As classes compiladas serão também necessárias mais tarde. Na figura B.1 encontra-se o diagrama de sequência deste módulo, bem como o seu fluxograma.

3.1.3 Verificador das especificações

(*JMLValidator*)

Através deste módulo são validadas sintacticamente as especificações que foram definidas no método a testar através da utilização do compilador de JML: `jmlc`. O diagrama de sequência deste módulo encontra-se descrito na figura C.1. Este módulo também é responsável por configurar o caso de teste genérico que irá ser usado mais tarde pelo módulo (*JunitModule*).

3.1.4 Módulo de teste Junit

(*JunitModule*)

Este módulo é responsável por verificar se um determinado teste passa com sucesso ou não. Um caso de teste passa com sucesso se as pós-condições das asserções em JML são validadas depois da invocação do método. Esta validação é feita através do uso da ferramenta JmlRac. O diagrama de sequência deste módulo encontra-se descrito na figura G.1.

3.1.5 Cálculo do grau de Mutação

(*MutationScore*)

Este componente é responsável por calcular o $mutation_score$ de um determinado caso de teste. Foi desenvolvido a partir da ferramenta Jumble. O seu funcionamento encontra-se descrito na figura D.2 e figura D.3

3.1.6 Gerador de dados

(*DataGenerator*)

Uma das actividades mais importantes desta ferramenta é a construção do domínio de pesquisa necessário para o Algoritmo de Optimização por Bando de Partículas , é nesse domínio que o algoritmo irá operar, tentando encontrar o caso de teste que possua o maior valor na função de *fitness*.

Este espaço é constituído por tuplos (id, $best\_mutation\_score$, arg_1 ,arg_2, arg_3 ..., arg_n) em que:

- id Identifica univocamente a partícula.
- $best\_fitness$ Representa o melhor valor obtido pela partícula na função de *fitness*.
- arg_1 ,arg_2, arg_3 ..., arg_n Representa os *genes* que constituem a partícula,isto é, o conjunto dos argumentos que irão ser usados para testar o método.

Id	mutation_score	side1	side2	side3
1	0.43	1	433	12
2	0.82	-2	-23	34
3	0.34	0	0	5
4	0.1	193	999	-1

Tabela 3.1: Dados gerados de forma aleatória

Na tabela 3.1.6 encontra-se uma lista de dados de teste gerados de forma aleatória para o método `public static String getType(int side1 , int side2, int side3)` (ver anexo Q.1).

Sendo um dos mecanismos mais simples para a geração de dados, este apenas é aceitável para tipos de dados básicos (`Integer`, `Long`, `Double` ...).

Tomemos como exemplo a criação de um objecto da classe `java.net.Socket`: um dos construtores possíveis é `Socket(String host, int port)`, émuito difícil gerar de forma

aleatória a cadeira de caracteres `host` que represente um *URL* válido instanciar correctamente a classe `Socket`.

Outro problema relacionado com a geração aleatória de dados de teste têm a ver com a redundância dos dados gerados. Estudos [TXN04] indicam haver entre 50% e 90% de redundância de dados quando estes são gerados de forma aleatória. Dados redundantes apenas aumentam o tempo de teste sem trazer nenhum aumento de qualidade.

No caso do exemplo anterior, os dados de entrada `new Socket( "foo://bar.com", 80)` e `new Socket( "bar://foo.com", 80)` são dados redundantes pois geram o mesmo resultado: uma excepção é gerada pois os dois endereços não estão de acordo com as regras definidas para a construção de *URL*'s. [2]

Ao adicionar mecanismos de especificação (como é o caso do JML), estamos ainda a levantar um terceiro problema: os dados gerados terão obrigatoriamente de garantir que as pré-condições e invariantes não são quebradas.

Uma das possíveis abordagens para resolver estes problemas é a construção iterativa dos dados de teste[CRM07] implementada na Zoonomia. Nesta abordagem, os dados de teste são construídos de forma incremental baseados nas especificações do método, verificando sempre se os dados a ser gerados são equivalentes a dados já construídos, e que novos dados são baseados em dados já existentes (*object pooling*).

O algoritmo proposto por Yoonsik Cheon e Carlos E. Rubio-Medrano [CRM07] encontra-se descrito nas listas 3.1, 3.2, 3.3 e 3.4.

Lista 3.1: Pseudo algoritmo para geração de dados de teste

```
1 Object generate(Method m) {
2   do {
3     c = declaring_class(m) ;
4     r = generate_class(c);
5     foreach (p_i of m's parameter)
6       a_i = generate_arguments(T_i);
7   } while (is_redundant(r,a_1,...,a_n);
8   return r, a_1,...,a_n;
9 }
```

[2] por exemplo `http://www.faqs.org/rfcs/rfc1738.html`

O algoritmo implementado inicia o processo determinando qual o tipo da classe a testar (linha 3 da lista 3.1). Depois tenta encontrar um construtor válido para essa classe (linha 4 da lista 3.1). Posteriormente vai tentar instanciar cada um dos argumentos com valores gerados de forma dinâmica (linhas 5 e 6 da lista 3.1).

Lista 3.2: Pseudo algoritmo para geração de dados de teste

```
1
2  Object generate_arguments(Class T) {
3    if (pool_size() >0) {
4      r = pick_from_pool();
5    } else {
6      do {
7        Constructor c = pick_constructor(T);
8        foreach (p_i of c's parameter)
9          a_i = generate_random_argument(T_i);
10         r = new T(a_1 , a_2 , ..., a_n );
11     } while (invocation fails && assertions fails);
12   }
13   r = mutate(r);
14   add_to_pool(r);
15   return r;
16 }
```

A segunda fase deste algoritmo está representado na lista 3.2 e é responsável por instanciar um conjunto argumentos válidos que irão ser usados para construir um determinado objecto. Este método baseia a instanciação de novos objectos a partir de objectos já existentes, aplicando nestes mecanismos de mutação, aumentando assim a probabilidade de que os novos objectos cumpram as asserções definidas. Este processo é composto por duas fases [CRM07]:

- Criação de um objecto da classe em questão. Caso ainda nenhum elemento no domínio de pesquisa, é gerado um de forma aleatória (linhas 6-11 da lista 3.2).
- Mutação do objecto. O novo estado é gerado através do uso de vários operadores de mutação dos dados sendo posteriormente validado se as asserções são validadas (linha 13 da lista 3.2).

Lista 3.3: Pseudo algoritmo de mutação

```
1  Object mutate(Object o) {
2    for (several times) {
3      do {
4        m = pick mutator(o);
5        foreach (p_i of m's parameter)
6          a_i = generate(T_i);
7        invoke "o.m(a_1 , a_2 , ..., a_n)";
8      } while (m(a_1 , a_2 , ..., a_n) fails);
9    }
10   return o;
11 }
```

Uma das fases do algoritmo proposto é a mutação de um determinado objecto no processo de geração de novos dados. Foram implementadas duas abordagens: caso o objecto em questão implemente o interface `com.npalma.zoonomia.Mutable`, é invocado o método `mutate()`, é da responsabilidade do programador definir a mutação dos objectos. Se a classe do objecto não implementar o interface acima referido, são aplicados operadores de mutação ao objecto. Estes operadores servem para alterar o estado do objecto instanciado.

Lista 3.4: Pseudo algoritmo de validação de casos de teste redundantes

```
1  boolean is_redundant(Class r, Object a_1 , Object a_2 , ..., Object a_j) {
2    foreach (p_i of population) {
3      foreach (a_j) {
4        if (a_j instanceof primitive_type){
5          if ((a_j==get_arg(p_i,j)))
6            return true;
7        } else if (a_i redefine java.lang.Object.equals() ) {
8          if (a_j.equals(get_arg(p_i,j))))
9            return true;
10       else {
11         if (r(a_i)=r(get_args(p_i)))
12           return true;
13       }
```

```
14    }
15    return false;
16  }
```

Depois de encontrados os argumentos do o construtor, é gerado uma instância da classe baseada nos argumentos encontrados. Conforme foi referido na secção 3.1.6, a ferramenta tenta optimizar o espaço de pesquisa eliminando dados de teste redundantes.

Dois objectos `data1` e `data2` são redundantes se [CRM07]:

- Sendo `data1` e `data2` de tipos primitivos (`int`,`long`,...) e o resultado operador de comparação == devolver `true`.
- Sendo `data1` e `data2` do tipo `array`, tendo o mesmo tamanho (`data1.length == datas2.length`) e cada um dos elementos respectivos dos vector forem equivalentes (`for(i=0;i<=data1.lengh;i++) data1[i] == data2[i]`).
- Sendo `data1` e `data2` objectos, duas situações podem acontecer:
  1. Os dois objectos implementam o método `equals()` (`data1 .equals(data2)` devolve o valor `true`).
  2. Não implementado o método `equals()`, os dados são equivalentes se o resultado do teste for o mesmo.)

Para validar se um determinado dado é redundante, a comparação é feita com todos os elementos já existentes no domínio através do algoritmo descrito na lista 3.4

Lista 3.5: Pseudo algoritmo para geração de novas elementos no domínio

```
1  Object add_new_elements(Method m) {
2    Population new_population=new Population();
3    c = declaring_class(m);
4    r = generate_class(c);
5
6    foreach (data_i of population) {
```

```
7       int mutants_to_generate=get_ratio(data_i);
8
9     do {
10      foreach (arg_j of data_i arguments)
11        Object arg_new=mutant(arg_j);
12
13    } while ((is_redundant(r,arg_new1...arg_newj) on population) or (is_redundant(r,arg
          _new1...arg_newj) on new_population) )
14  }
15  population=population ⋃ new_population;
16 }
```

É da responsabilidade deste módulo a geração do domínio de pesquisa. Sendo um domínio discreto e fechado constituído por partículas cuja estrutura foi definida em 3.1.6, faz com que em várias iterações, todos os elementos tenham sido analisados, levando à sua exaustão.

Para ultrapassar este problema existe um processo em paralelo ao Algoritmo de Optimização por Bando de Partículas que analisa o domínio e valida qual o estado de exaustão deste, adicionando novos elementos caso exista necessidade. Na lista 3.5 encontra-se o algoritmo que realiza esta operação.

O processo de geração de novos elementos domínio percorre todos os elementos já existentes e consoante o nível de mutação do mesmo é-lhe atribuído um peso que irá determinar o número de novos elementos que irão ser baseados neste ($\texttt{data}_i$). O número de novos dados que irão ser adicionados ao domínio resulta da relação entre o valor de *fitness* obtido pela partícula e uma taxa de crescimento definida pelo utilizador. A solução implementada usa a função $Math.floor((mutation\_score * growing\_rate)$ em que *growing_rate* é a taxa de crescimento indicada pelo utilizador no inicio da execução da ferramenta através da opção `zoonomia.growing.rate`. Assim garante-se que haverá um maior número de novas partículas do dominio junto daqueles que possuem maior valor de *fitness*.

Como exemplo, se uma determinada partícula possuir um valor de *fitness* de 0.34 e a taxa de crescimento definida pelo utilizador for de dez, serão gerados seis novos elementos baseados em pequenas mutações do original.

3.1.7 Algoritmo PSO

(*PSOModule*)

Este módulo é responsável por pesquisar os casos de teste com maior *mutation_score* através do uso do Algoritmo de Optimização por Bando de Partículas

Durante esta pesquisa, podem acontecer duas situações:

- Nas partículas geradas e analisadas, não é quebrada nenhuma asserção, logo não foi possível encontrar nenhum erro no método em causa.
- É encontrado uma inconsistência entre as pós-condições ,invariantes e o resultado do método. Neste caso, considera-se que o método não cumpre as especificações para o qual foi desenhado, sendo esta situação reportada ao utilizador indicando qual foi o caso de teste, bem como a asserção que foi quebrada. É essencial que os dados de teste validem sempre as pré-condições e as invariantes são sempre validadas antes da execução do método. Esta situação é sempre verificada , pois é feita uma validação na altura da geração do domínio de dados. Poderá ainda acontecer um erro na asserção que foi quebrada. Cabe ao programador validar se realmente é erro de programação ou de especificação.

Para esta pesquisa são necessárias as seguintes estruturas:

- Domínio de pesquisa.
- População inicial.
- Um critério de paragem.
- Uma função de fitness.
- Um operador de velocidade.

Domínio de pesquisa O domínio de pesquisa é gerado pelo módulo (*DataGenerator*)

População inicial Como domínio de dados está definindo no inicio, a população inicial representa um subconjunto do dominio de dados. Na ferramenta apresentada, a população inicial sobre a qual o algoritmo irá operar encontra-se materializada numa estrutura de dicionário em que cada entrada representa um elemento do bando e este é constituido por um

vector que representa a velocidade e por um valor real que representa o melhor valor que a partícula já possuiu (*pbest*). Para definir a população inicial, basta escolher aleatoriamente partículas do dominio de dados. O vector de velocidade é gerado de forma aleatória.

Critério de paragem O critério de paragem fica a cargo do utilizador. Foram implementadas dois critérios: baseado no numero máximo de iterações sobre o domínio de dados ou quando se atinge um valor esperado na função de fitness. O critério é definido pelo utilizador através da definição da propriedade `zoonomia.pso.stop.criteria`.

Função de fitness A função de fitness classifica cada uma das partículas em função do seu grau de mutação (*mutation score*) obtido a partir do módulo (***MutationScore***). Quanto maior for o número de mutantes eliminados, maior será a probabilidade de este caso de teste encontrar erros não intencionais[Bybro]. O valor máximo que um determinada partícula pode ter é 1.

Função de velocidade A função de velocidade serve para deslocar as partículas do bando pelo dominio de dados . Como o dominio de dados é discreto, as partículas não se podem deslocar livremente, sendo necessário criar mecanismos para garantir que o bando apenas se desloca para posições válidas.

O Algoritmo de Optimização por Bando de Partículas foi originalmente pensado para optimizações de funções em que o dominio de pesquisa era constituido por partículas onde as operações aritméticas usuais podiam ser aplicadas às partículas para calcular novas posições dos elementos do bando. Em linguagens orientadas por objectos, não é possível aplicar operações aritméticas em objectos pois este conceito não existe. É necessário então utilizar um outro mecanismo de movimentação das partículas.

O mecanismo utilizado no funcionamento da Zoonomia baseia-se no conceito de **distância** entre objectos.

Uma distância ($D(a,b)$) é uma função que respeita as seguintes condições[CLOM06]:

- $D(a,b) \geq 0$.
- $D(a,b) = 0 \Leftrightarrow a = b$.
- $D(a,b) = D(b,a)$.

- $\forall r : D(a,b) \leq D(a,r) + D(r,b)$

Quando a e b são tipos primitivos ou simples (`int`, `double`, `long`, `Integer`, ...) é simples transpor o conceito de distância para objectos, bastando definir a função de distância como $D(a,b) = |a,b|$. Se os tipos forem do tipo `String`, será fácil transpor o conceito pois poderá ser usado o algoritmo Levenshtein[Lev07] que mede a distância entre duas `String`'s.

Quando a e b são instâncias de objectos, $D(a,b)$ pode ser calculada através da definição de uma relação entre as distâncias dos vários membros que compõem a classe. Existem várias relações estudadas para calcular a distância entre dois objectos[CLOM06]. Na implementação desta ferramenta foi utilizada a distância Euclidiana:

$$dist(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

Figura 3.2: Distância Euclidiana

Na figura E.2 encontra-se o fluxograma do algoritmo que mede a distância entre dois objectos. O algoritmo implementado tenta, através o uso da recursividade encontrar as distâncias entre as várias variáveis que compõem o os objectos. Este algoritmo assume que os objectos em causa possuem sempre o mesmo tipo.

Suponhamos que temos das partículas p e $pBest$, compostas respectivamente pelos argumentos $\begin{bmatrix} "foo" & "bar" & 33 \end{bmatrix}$ e $\begin{bmatrix} "foa" & "bar0" & 55 \end{bmatrix}$.

O vector de distância será então $\begin{bmatrix} 1 & 1 & 22 \end{bmatrix}$ representando para cada uma das variáveis dos dois objectos a distância associada a cada uma delas.

No algoritmo PSO tradicional, o processo de movimentação é composto por duas fases: uma em que é calculado o valor da velocidade e um segundo em que é aplicado esse vector à partícula.

Na solução implementada na Zoonomia, o processo de calculo do vector de velocidade é realizado em várias fases através da utilização do vector de distância:

1. Cálculo do vector de distância $v_1 = dist(pbest, present)$
2. Cálculo do vector de distância $v_2 = dist(gbest, present)$
3. Cálculo do vector $v_1' = c1 * rand() * v_1$

4. Cálculo do vector $v_2' = c2 * rand() * v_2$

5. Cálculo do novo vector de velocidade $v = v_1' + v_2'$

No final da fase 5 foi encontrado o "pulo"que a partícula de de dar. Como não é possível aplicar directamente o algoritmo ($present[] = present[] + v[]$) pelas razões já indicadas, é necessário seguir uma outra abordagem:

1. Para cada uma das partículas existentes no domínio de pesquisa, calcular a respectiva distância.

2. De entre as partículas do domínio, escolher aquelas que possuem a distância mais próxima daquela que foi calculada na fase anterior.

3. De entre a lista das partículas calculadas no passo anterior, escolher a que possuir a menor distância *gBest*. Caso exista mais do que uma posição, é escolhida uma aleatoriamente

O passo três serve para garantir que as partículas do bando são deslocadas para posições mais próximas daquela onde foi alcançado o maior valor de $mutation_score$.

A base do Algoritmo de Optimização por Bando de Partículas utilizado nesta ferramenta encontra-se representado no anexo E.1.

3.1.8 Fase 1 (Inicialização)

Nesta fase são preparadas todas as estruturas necessárias para a execução dos testes. Nesta fase são realizadas também as validações código e nas asserções. É também gerado também o espaço inicial de pesquisa necessário para o algoritmo de optimização por bando de partículas. Na figura 3.3 encontra-se representada a fase 1 do processo de teste.

O processo é iniciado pelo programador que através da criação de um ficheiro de configuração ou de uma base de dados relacional, define os parâmetros de entrada necessários para a ferramenta funcionar. Na figura A.1 encontra-se um exemplo de um ficheiro de configuração. Nesse exemplo são definidas 4 bibliotecas, o método a testar será o método `add()` que possui a assinatura de ter dois argumentos inteiros e o critério de paragem é quando o algoritmo de optimização tiver realizado 10 iterações.

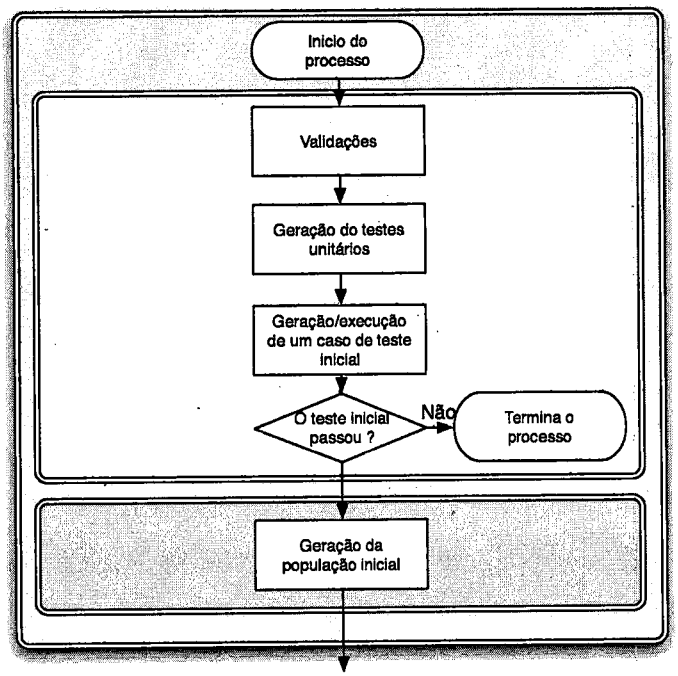

Figura 3.3: Diagrama geral da fase 1

Depois de verificar se todos os parâmetros de entrada, o módulo (***InitialValidator***) é invocado e o código é validado através do uso do compilador `javac`. Erros encontrados nesta fase obrigam a que a ferramenta termine. Posteriormente é feita uma validação das asserções existentes no código através do uso do módulo (***JMLValidator***).

Com as validações possíveis no código e nas asserções, é gerado e executado um caso de teste de forma automática pelo módulo (***JMLValidator***). Com este teste pretende-se confirmar que o método em causa consegue validar as pré-condições, pós-condições e invariantes pois não faz sentido avançar com a análise do método sem existir a certeza que o código não contém erros.

De seguida é gerado o domínio de dados através do módulo (***DataGenerator***).

3.1.9 Fase 2 (Execução)

A Fase 2 é preenchida pela execução módulo (***PSOModule***). Na figura E.1 encontra-se o fluxograma de funcionamento do módulo.

3.2 Casos de estudo

Nesta secção irão ser apresentados dois casos de estudo baseados no mesmo problema e os respectivos resultados.

3.2.1 Classificação do Triângulo

No anexo Q encontra-se um programa que classifica um triângulo baseado na dimensão dos lados. Os triângulos pode ser classificados em três tipos: **equilátero** se possuir todos os lados iguais, **isósceles** se possuir pelo menos dois lados com a mesma dimensão e **escaleno** se possuir os três lados com dimensões diferentes.

Para o primeiro teste, os parâmetros usados estão definidos na figura 3.4 foram:

```
zoonomia.classpath=../bin
zoonomia.class= Triangle
zoonomia.method= getType(int,int,int)
zoonomia.pso.stop.criteria=com.npalma.zoonomia.MaxIterations
com.npalma.zoonomia.MaxIterations.args.1=5
zoonomia.redundancy.checked=true
zoonomia.population.size=5
zoonomia.initial.domain.size=70
zoonomia.growing.rate=10
```

Figura 3.4: Configurações do caso de estudo

Na tabela 3.2.1 encontra-se o domínio de pesquisa calculado na primeira fase, sendo composto por partículas constituídas por valores que validam as pré-condições existentes definidas no método `getType` (`side1 > 0 && side2 > 0 && side3 > 0side1 + side2 >= side3, side2 + side3 >= side1` e `side1 + side3 >= side2`). Foram necessárias 2832 iterações para conseguir atingir uma população de 70 partículas válidas, o que dá uma taxa de sucesso de cerca de 2.5%. A fase de geração do domínio demorou 192 segundos num computador com processador PowerPC G4 1 Mhz.

O bando inicial foi calculado ao acaso, sendo escolhido ao caso 5 partículas. Na simulação que deu origem a estes dados foram escolhidas as partículas $p_{14} = \begin{bmatrix} 113 & 60 & 124 \end{bmatrix}$, $p_{33} = \begin{bmatrix} 107 & 107 & 61 \end{bmatrix}$, $p_{44} = \begin{bmatrix} 137 & 140 & 26 \end{bmatrix}$, $p_{62} = \begin{bmatrix} 72 & 70 & 113 \end{bmatrix}$, $p_{65} = \begin{bmatrix} 68 & 70 & 69 \end{bmatrix}$.

01-0-(86,45,126)	02-0-(96,84,59)	03-0-(158,93,149)	04-0-(107,89,125)	05-0-(52,23,43)	06-0-(98,103,38)
07-0-(128,64,149)	08-0-(82,55,136)	09-0-(115,56,86)	10-0-(103,67,107)	11-0-(111,104,9)	12-0-(91,70,67)
13-0-(90,80,131)	14-0-(113,60,124)	15-0-(29,73,92)	16-0-(25,37,49)	17-0-(155,26,129)	18-0-(40,55,25)
19-0-(94,30,74)	20-0-(114,69,56)	21-0-(208,228,162)	22-0-(39,87,118)	23-0-(85,52,113)	24-0-(22,59,6)
25-0-(82,23,100)	26-0-(108,209,117)	27-0-(113,45,149)	28-0-(34,107,76)	29-0-(124,152,172)	30-0-(68,17,62)
31-0-(101,124,82)	32-0-(175,122,55)	33-0-(107,107,61)	34-0-(89,73,126)	35-0-(78,129,88)	36-0-(89,104,42)
37-0-(136,222,96)	38-0-(96,128,72)	39-0-(110,126,66)	40-0-(125,109,134)	41-0-(54,54,51)	42-0-(95,127,47)
43-0-(126,88,74)	44-0-(137,140,26)	45-0-(102,125,63)	46-0-(100,63,104)	47-0-(131,204,73)	48-0-(116,57,143)
49-0-(89,31,79)	50-0-(74,56,25,45)	51-0-(46,113,95)	52-0-(135,23,114)	53-0-(148,155,43)	54-0-(138,83,156)
55-0-(123,86,58)	56-0-(131,130,131)	57-0-(106,91,26)	58-0-(154,110,73)	59-0-(61,111,67)	60-0-(66,126,117)
61-0-(23,10,13)	62-0-(72,70,113)	63-0-(123,92,121)	64-0-(138,120,31)	65-0-(68,70,69)	66-0-(32,130,106)
67-0-(63,68,69)	68-0-(90,54,71)	69-0-(44,40,53)	70-0-(63,67,60)		

Tabela 3.2: Domínio de pesquisa inicial encontrado para o caso de estudo 1

Todos os elementos do bando foram configurados com o melhor valor de *fitness* a 0 (`pBest=0`) e os respectivos vectores de deslocação foram configurados com valores aleatórios. Não foi activado o sistema que adiciona novos dados ao dominio de pesquisa.

Após a execução das 10 iterações, o bando encontrava-se nas posições: $p_{23} = \begin{bmatrix} 85 & 52 & 113 \end{bmatrix}$, $p_{57} = \begin{bmatrix} 106 & 91 & 26 \end{bmatrix}$, $p_{42} = \begin{bmatrix} 95 & 127 & 47 \end{bmatrix}$, $p_{28} = \begin{bmatrix} 28 & 107 & 76 \end{bmatrix}$, $p_{04} = \begin{bmatrix} 107 & 89 & 125 \end{bmatrix}$.

Nas 10 iterações não foi encontrado nenhum erro no código (nenhuma partícula quebrou as especificações do método), tendo sido obtido em cada uma das partículas do bando um grau de mutação de 45%. O valor não é muito alto, devendo-se essencialmente ao facto do número de operadores não ser muito elevado.

3.2.2 Classificação do Triângulo com erro no código

No anexo Q encontra-se o programa Q.2 que é baseado no programa utilizado no teste anterior mas com algumas alterações, forçando a existência de um erro quando o produto dos valores `p.side1`,`p.side2` e `p.side3` for superior a 500000.

Foi também alterado o método `getType()` que passou a receber um objecto do tipo `Particle` em vez de três inteiros. O objectivo desta alteração é testar a capacidade de gerar dados de teste para objectos e validar que a função de velocidade usada no algoritmo PSO é valida. Para simplificar a descrição do caso de teste, a expressão (x, y, z) representa o objecto `new Particle(x,y,z)`.

Para o primeiro teste, os parâmetros usados foram (ver figura 3.5):

```
zoonomia.classpath=../bin
zoonomia.class= TriangleWithError
zoonomia.method= getType(int,int,int)
zoonomia.pso.stop.criteria=com.npalma.zoonomia.MaxIterations
com.npalma.zoonomia.MaxIterations.args.1=50
zoonomia.redundancy.checked=true
zoonomia.population.size=5
zoonomia.initial.domain.size=10
zoonomia.growing.rate=10
```

Figura 3.5: Configurações do caso de estudo

A população inicial foi reduzida para testar também a função de geração de novas partículas ao dominio de teste.

01-0-(74,46,57)	02-0-(25,67,60)	03-0-(40, 71, 96)	04-0-(30,96,121)	05-0-(26,34,45)	06-0-(89,51,51)
07-0-(37,84,95)	08-0-(50, 88, 76)	09-0-(67,95,48)	10-0-(40, 53, 90)		)

Tabela 3.3: Domínio de pesquisa inicial para o algoritmo de pesquisa para o caso de estudo 2

Na tabela 3.2.2 encontra-se o domínio de pesquisa calculado na primeira fase, sendo composto por partículas constituídas por valores que validam as pré-condições existentes definidas no método `getType ( p.side1 > 0 && p.side2 > 0 && p.side3 > 0 p.side1 + p.side2 >= p.side3, p.side2 + p.side3 >= p.side1` e `p.side1 + p.side3 >= p.side2`).

Foram necessárias 558 iterações para conseguir atingir uma população de 10 partículas válidas, o que dá uma taxa de sucesso de cerca de 1.7%. A fase de geração do dominio demorou 192 segundos num computador com processor PowerPC G4 1 Mhz

O bando inicial é calculado ao acaso, sendo escolhido ao caso 5 partículas. Na simulação que deu origem a estes dados, foram escolhidas as partículas $p_{01} = \begin{bmatrix} 74 & 46 & 57 \end{bmatrix}$, $p_{08} = \begin{bmatrix} 50 & 88 & 76 \end{bmatrix}$, $p_{06} = \begin{bmatrix} 89 & 51 & 51 \end{bmatrix}$, $p_{04} = \begin{bmatrix} 30 & 96 & 121 \end{bmatrix}$, $p_{10} = \begin{bmatrix} 40 & 53 & 90 \end{bmatrix}$.

Todos os elementos do bando são instanciados com o melhor valor de *fitness* a 0 (`pBest=0`) e os respectivos vectores de deslocação foram configurados com valores aleatórios.

No fim da iteração número 37, na fase de geração de novas partículas foi gerada a partícula *new Particle(88, 101,60)* que cumpre as pré-condições, mas faz com que as pós condições fossem quebradas. Foi encontrado um erro.

Capítulo 4

Conclusão e trabalho futuro

O principal objectivo desta dissertação foi validar a viabilidade da utilização de tecnologias de áreas tão diversas como Algoritmo de Optimização por Bando de Partículas , Testes por Mutação e Programação por Contrato para a construção de uma ferramenta de geração e execução de testes de forma automática, uma abordagem diferente do habitual.

A interacção do programador com a ferramenta reduz-se a alguns parâmetros, pois o uso de linguagens formais para descrever o comportamento do código é simples de utilizar e é retirado o peso do programador de criar os casos de teste para os métodos. O uso da linguagem JML veio a tornar-se muito importante devido ao grande número de operadores e ao facto de possuir um número considerável de extensões que ajuda a transformar as especificações em testes unitários.

Um desafio também alcançado foi a utilização de técnicas de teste por mutação que permitem classificar com sucesso que fornecem uma medida do grau de adequação para o código a testar.

Por fim temos a utilização de Algoritmo de Optimização por Bando de Partículas para pesquisa e optimização dos testes que tornou-se essencial para o sucesso da ferramenta para pesquisar de entre os testes gerados pela utilização das linguagens formais, quais aqueles que conseguiam ter maior grau de adequação. O algoritmo foca-se em realizar casos de teste com maior grau de adequação, aumentando a probabilidade de encontrar erros e em paralelo aumenta a qualidade da bateria de testes (dominio de pesquisa).

Um dos grandes desafios desta tese foi criar um mecanismo de geração de dados para

teste. De nada servem as ferramentas de apoio ao desenvolvimento de software se estas não suportarem a maioria das características da linguagem para que foram desenhadas.

Permitir que fossem gerados dados de teste não primitivos é sem duvida uma mais valia para a Zoonomia, que não suportando as características todas, é uma prova de conceito de que é possível gerar testes automáticos para grande parte dos programas desenvolvidos em Java. O uso da ferramenta JmlUnit foi essencial neste componente.

Tentar encontrar erros de software utilizando técnicas inteligentes é uma área ainda embrionária, ao contrário das ferramentas de analise de código que já estão numa fase mais madura. Começa a surgir alguma investigação na área de utilização dos algoritmos naturais para testar software em tempo de compilação. Espero que este trabalho contribua um pouco mais para a investigação de técnicas de teste baseadas em algoritmos naturais.

Durante a fase de implementação e testes da ferramentas, foram surgindo questões que devem ser analisadas no futuro:

- Suportar o conceito de mutantes equivalentes. Mutantes equivalentes são mutações inseridas no código que não vão alterar o comportamento do programa. O uso de técnicas de análise de mutantes equivalentes irá diminuir o número de mutantes gerados, aumentando assim o desempenho.

- Melhorar o processo de operadores de mutação na fase de geração de dados. Existem alguns melhoramentos que podem ser realizados na fase de mutação de dados, nomeadamente a geração de subclasses.

- Acrescentar os operadores de mutação de classe. A ferramenta desenvolvida apenas suporta operadores de mutação para métodos. Suportar operadores de mutação para classes é uma característica que irá permitir encontrar erros derivados da especificidade da linguagem Java.

- Dar peso de ocorrência maior a determinados operadores de mutação.

- Aumentar o número de operadores específicos para a linguagem Java, tais como invocar o método `reset()` a `InputStream` e adicionar objectos a listas.

- Suportar construtores não básicos.
- Suportar geração de dados utilizando diferentes construtores. Caso exista mais do que um construtor para a classe em questão, permitir que o objecto seja gerado utilizando mais do que um construtor
- Suporte para IDE's. Nos dias de hoje é muito importante que ferramentas de apoio ao desenvolvimento e execução de testes possam estar inseridas em ambientes de desenvolvimento integrados para uma melhor utilização.

Apêndice A

Linguagem Java

A linguagem Java foi a escolhida para a implementação deste sistema bem como a linguagem destino para a ferramenta proposta. É portanto importante para o leitor possuir os conceitos básicos desta linguagem.

Java surgiu como parte de um projecto de pesquisa que visava a criação de um software avançado que atendesse a uma extensa variedade de hardware e sistemas embebidos. O objectivo inicial era desenvolver um ambiente operacional pequeno, confiável, portável, distribuído e que operasse em tempo real. Inicialmente a linguagem escolhida foi C++, porém com o passar do tempo, as dificuldades encontradas com C++ aumentaram até o ponto em que os problemas poderiam ser melhor endereçados se fosse criada uma linguagem completamente nova. As decisões de arquitectura e desenho foram inspiradas em linguagens como Eiffel, SmalllTalk e C. A sintaxe tem suas raízes claramente definidas em C++. Deste modo, a linguagem Java foi projectada para atender a vários requisitos desejáveis numa linguagem de programação, como por exemplo, confiança (devido à sua gestão de memória) o que resulta num ganho de eficiência; simplicidade, (por eliminar alguns conceitos de C e C++ que dificultavam nesse sentido) e reutilização de código (por ser uma linguagem orientada por objectos).

Actualmente este linguagem é utilizada numa grande diversidade de dispositivos que vão desde aplicações Web, passando por dispositivos móveis e cartões smart cards. É mantida pela empresa Sun Microsystens, encontrando-se actualmente na versão 6, versão esta que pele primeira vez foi lançada sob a licença GPL.

A.1 Noção de Classe e Objecto

De uma forma simplista, uma classe pode ser visto como uma estrutura do tipo registo em C que para além dos campos pode também incluir procedimentos que podem ou não manipular esses campos. Na terminologia de classes, os campos são designados de atributos e esses procedimentos de métodos da classe. Quando se declara uma variável como sendo do tipo da classe X, está-se a criar um objecto (instância ou concretização de uma classe especifica). Para criar um objecto é utilizada a palavra *new* e chamado o construtor dessa classe. O construtor é um método da classe que pode ser utilizado, por exemplo, atribuir um valor inicial aos atributos). Esse objecto, sendo uma instância ou concretização da classe, retém os atributos e os métodos desta. Os atributos e os métodos podem ser acedidos externamente (quando na declaração dos mesmos é utilizada a palavra reservada public) ou simplesmente internamente. Para aceder a atributos ou métodos é utilizado o nome do objecto seguido de '.' e do nome do atributo ou do método.

Uma das principais vantagens da linguagem Java, é o conceito de *garbage collector*. O garbage collector é um sistema incluído na plataforma que executa o bytecode e vai libertando posições de memória relativas a objectos que não são mais utilizados durante a execução de uma determinada aplicação. Como a libertação dinâmica de memória é realizada pelo *garbage collector*, a linguagem não necessita de uma instrução de libertação de memória.

A.2 Noção de referência

Ao contrário da linguagem C, na linguagem Java não existe o conceito de apontadores de memória, todas as variáveis (à excepção das variáveis do tipo primitivo `long`, `int`, `short`, `byte`, e `boolean`) identificam a referência para um determinado objecto. Quando se declara uma variável como sendo do tipo da classe especificada e se lhe atribui um objecto, estamos a dizer semanticamente que esta variável é uma referência a um objecto armazenado algures na memória.

A.3 Noção de package

Em Java não existem os ficheiros separados para a definição das bibliotecas e a implementação das mesmas. É utilizado o conceito de pacote de software (**package**). Os packages que são referenciadas com a palavra `import`. O termo **package** refere-se a um pacote de software pré-desenvolvido, vulgarmente designado por biblioteca, e que se encontra à disposição do programador.

```
package com.npalma.mei.zoonomia;

import java.io.FileInputStream;
import org.apache.commons.configuration.*;
import org.apache.commons.io.IOUtils;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;
```

Figura A.1: Exemplo de importação de pacotes de bibliotecas

A.4 Excepções

A linguagem Java inclui um esquema para geração e tratamento de excepções através da utilização do conceito `try{...} catch(){..}`. Este tratamento permite canalizar as excepções de forma a que quando são geradas o programa possa provocar uma determinada reação. Ao utilizar métodos que geram excepções, a linguagem obriga a que essas mesmas excepções sejam canalizadas por estruturas try-catch ou então temos de propagar as mesmas para quem invoca o método que as utiliza

```
com.npalma.mei.zoonomia.lang.ZoonomiaException: Invalid file type
```

```
at com.npalma.mei.zoonomia.lang.parseJavaFile(Main.java:281)
at org.quartz.core.JobRunShell.run(JobRunShell.java:202)
```

A.5 Documentação

Um ponto importante no desenvolvimento de software é a documentação do código. Parte desta documentação esclarece determinadas decisões ou descreve o que é feito por cada método. A sintaxe para documentação é semelhante à linguagem C, para documentar linhas são normalmente utilizadas as duas barras para a direita (//), para comentar um conjunto de linhas é usado no incio a expressão `/*` e no fim `*/`.

A tecnologia Java integra também um gerador de documentação denominado `javadoc` que se baseia em determinados comentários no programa para produzir ficheiros html. Esses comentários são identificados por `/**`. Para além desta identificação existem também directivas que, por exemplo, relacionam certos comentários a argumentos dos métodos (directiva `@see`).

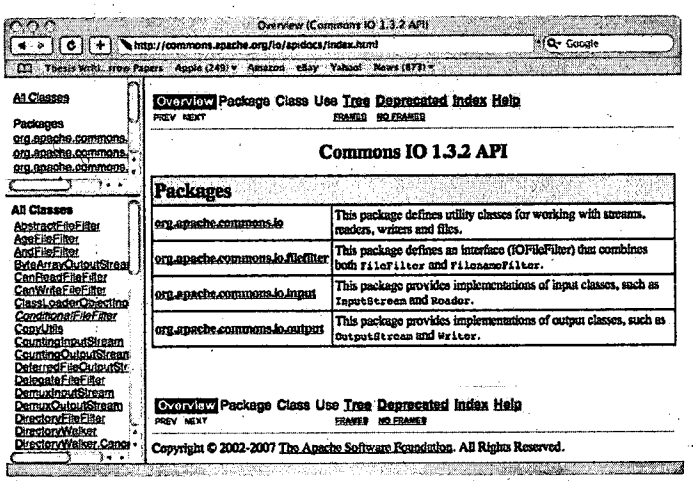

Figura A.2: Output gerado pelo sistema de documentação da linguagem Java

A.6 Principais vantagens

Simplicidade . É uma linguagem simples e intuitiva, fácil de compreender.

Orientada por objecto . Linguagens orientadas por objecto têm o foco no objecto e nas

suas interfaces. Esta orientação permite que haja uma maior e melhor reutilização do código.

Possui uma arquitectura neutra . A Linguagem de programação Java é uma linguagem orientada por objectos independente da plataforma, seguindo a máxima *"write once, run everywhere"*. Esta independência é obtida pelo facto da linguagem ser interpretada, ou seja, o compilador gera um código independente de máquina chamado *byte-code*. No momento da execução, este byte-code é interpretado por uma máquina virtual instalada na máquina. Portar Java para uma arquitectura hardware/s específica, basta instalar a máquina virtual (interpretador) adequada a essa arquitectura. Esta compatibilidade entre plataformas foi um factor muito importante para o sucesso da linguagem. Nos dias de hoje, existem compiladores Java para a maior parte dos sistemas operativos disponíveis no mercado (Windows, Mac OS, UNIX, etc.) que convertem código fonte Java em arquivos de classes de byte code. Os arquivos de classes correspondem aos arquivos binários executáveis gerados por compiladores para outras linguagens. Ao contrário dos arquivos binários nativos, o byte code Java não é específico para uma arquitectura particular de um microprocessador. A sua arquitectura "nativa"é a JVM, que actualmente existe somente em software.

Robusta e Segura . O projecto do Java visou a construção de uma linguagem para escrita de programas confiáveis. O Java enfatiza muito a detecção de possíveis problemas em tempo de compilação e em tempo de execução, realizando testes dinâmicos, eliminando situações que podem gerar erros. Como os programas Java ficam dentro do ambiente run-time Java, normalmente eles não interagem directamente com a CPU nativa ou com o sistema operativo. O ambiente run-time cuida da gestão de memória, de modo que os programadores não precisam alocar memória nem apagar da memória os objectos que deixaram de ser usados. Não há necessidade de aritmética de ponteiros, outra grande fonte de bugs em C++. A linguagem Java tem um modelo limpo e eficiente para manipulação de erros e encoraja a reutilização de código porque é orientado a objectos desde a sua concepção. Java substitui também por interfaces a complexa herança múltipla do C++. Porém, a tão propalada portabilidade do Java não está livre de defeitos. Os arquivos do Java são compilados e são convertidos de arquivos texto para

um formato que contém blocos independentes de bytes codes (código intermediário).

```
void whileInt() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

Figura A.3: Exemplo de código Java

```
Method void whileInt()
   0  iconst_0
   1  istore_1
   2  goto 8
   5  iinc 1 1
   8  iload_1
   9  bipush 100
  11  if_icmplt 5
  14  return
```

Figura A.4: Byte-code associado ao exemplo A.3

Em tempo de execução estes bytes codes são carregados, são verificados através do Byte Code Verifier (uma espécie de segurança), passando para o interpretador onde são executados. Caso este código seja acionado diversas vezes, existe um passo chamado JIT Code Generator, que elimina o utilização por demasia do tráfego da rede.

Anexos relacionados com a implementação

chapterDiagramas UML para o Componente de Gestão da Configuração

Lista A.1: Exemplo de configuração

```
1  #classpath
2  zoonomia.classpath=../lib/commons-lang-2.2.jar
3  zoonomia.classpath=../lib/commons-logging-1.1.jar
4  zoonomia.classpath=../lib/commons-pool-1.3.jar
5  zoonomia.classpath=../bin
6
7  #Define a classe e o método a testar
8  zoonomia.class=com.npalma.zoonomia.example.Test
9  zoonomia.method=add(int,int)
10
11 #criterio de paragem
12 zoonomia.pso.stop.criteria=com.npalma.zoonomia.MaxIterations
13 com.npalma.zoonomia.MaxIterations.args.1=10
14
15 #confirma se é feita a validação da redundância na geração dos dados de teste
16 zoonomia.redundancy.checked=true
17
18 #dados necessários para a geracao dos dados de entrada
19 zoonomia.population.size=30
20 zoonomia.initial.domain.size=300
21
22 #Taxa de crescimento da populacao
23 zoonomia.growing.rate=10
```

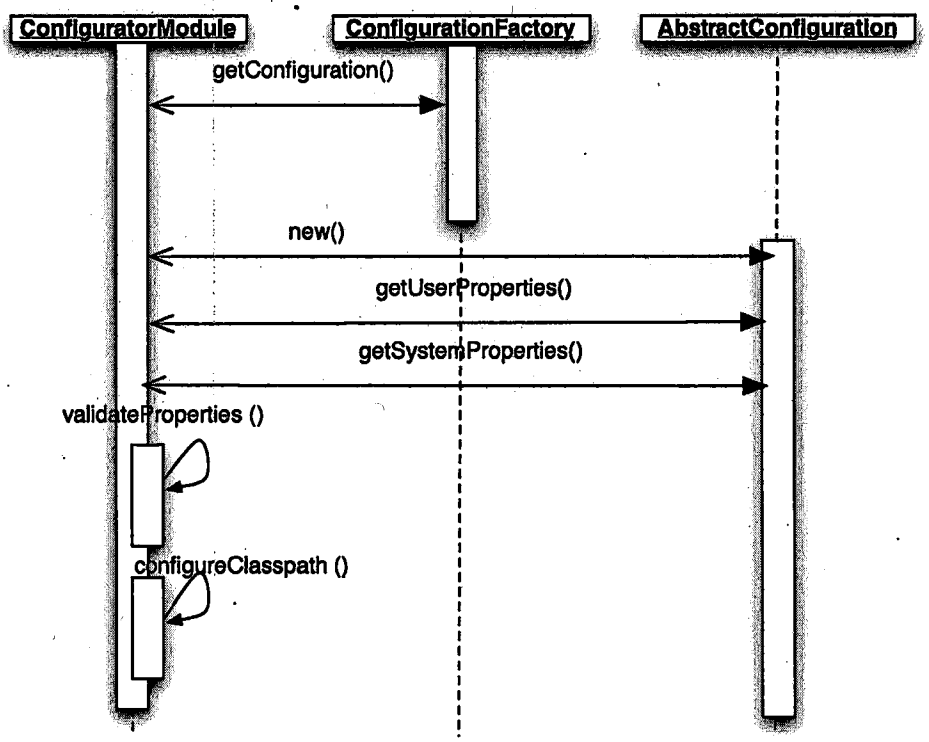

Figura A.5: Diagrama de sequência do módulo de configuração

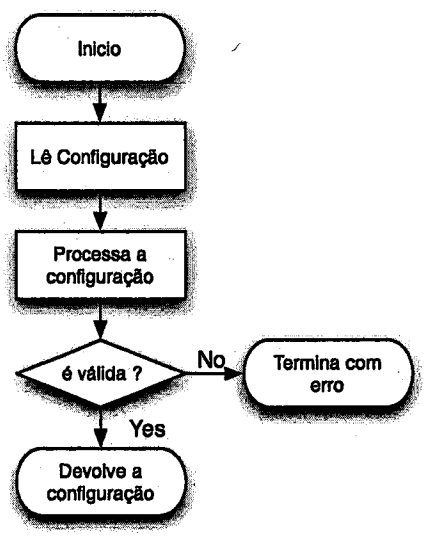

Figura A.6: Fluxograma do módulo de configuração

Apêndice B

Diagramas UML para o Componente de Verificação Inicial

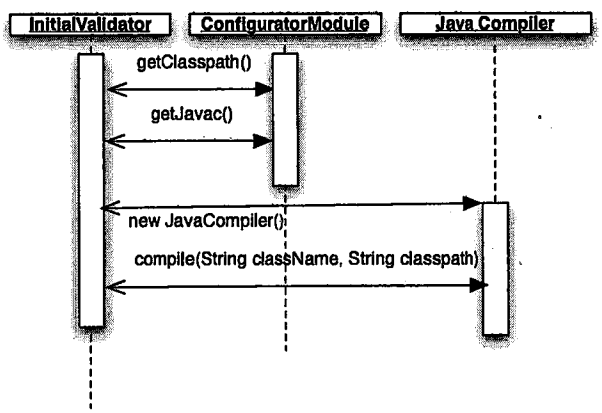

Figura B.1: Diagrama de sequência do módulo de validação inicial

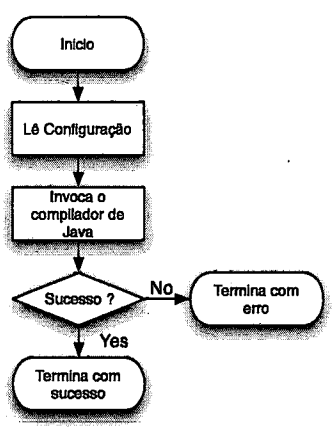

Figura B.2: Fluxograma do módulo de validação inicial

Apêndice C

Diagramas UML para o componente de Validação das Especificações

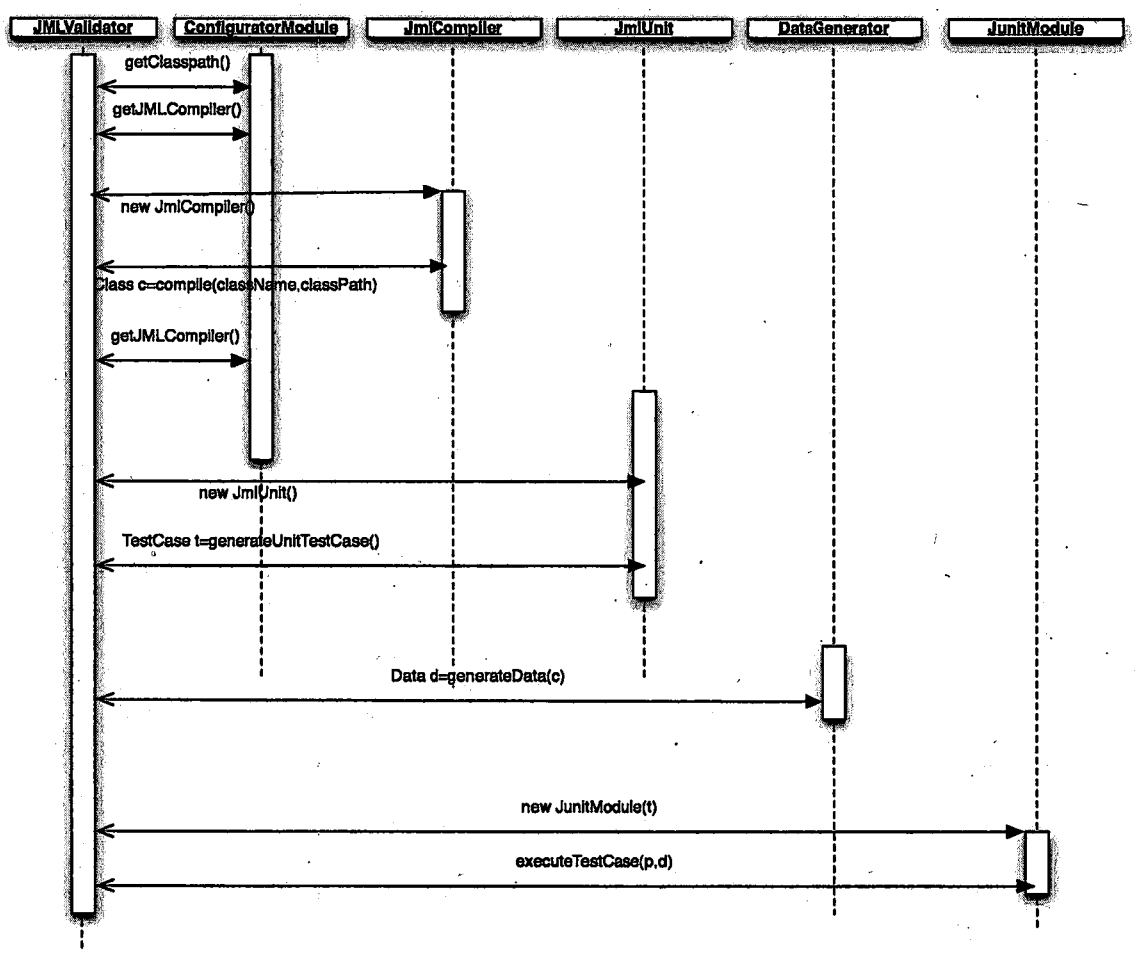

Figura C.1: Diagrama de sequência do módulo de validação das especificações

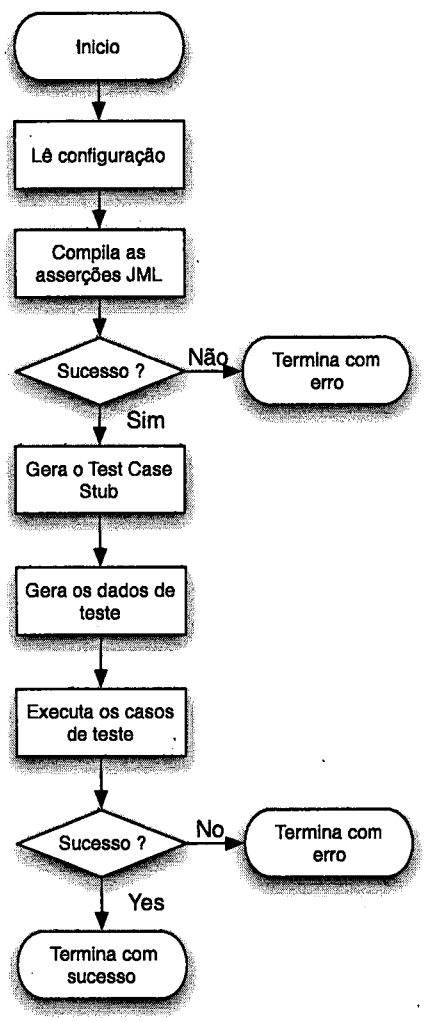

Figura C.2: Fluxograma do módulo de validação das especificações

Apêndice D

Diagramas UML para o componente MutationScore

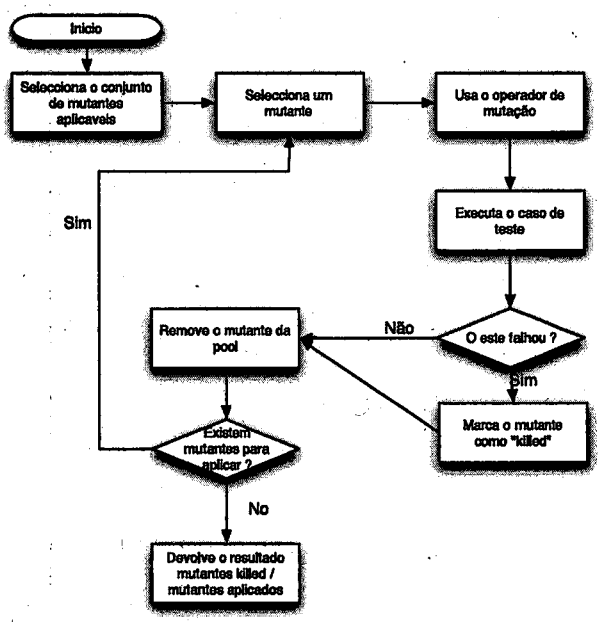

Figura D.1: Fluxograma do processo do cálculo do *mutation score*

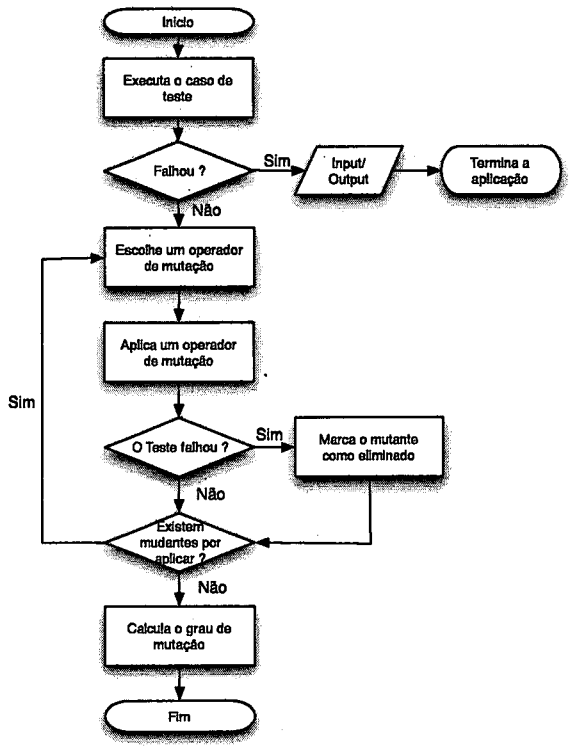

Figura D.2: Fluxograma do processo do cálculo do *mutation score*

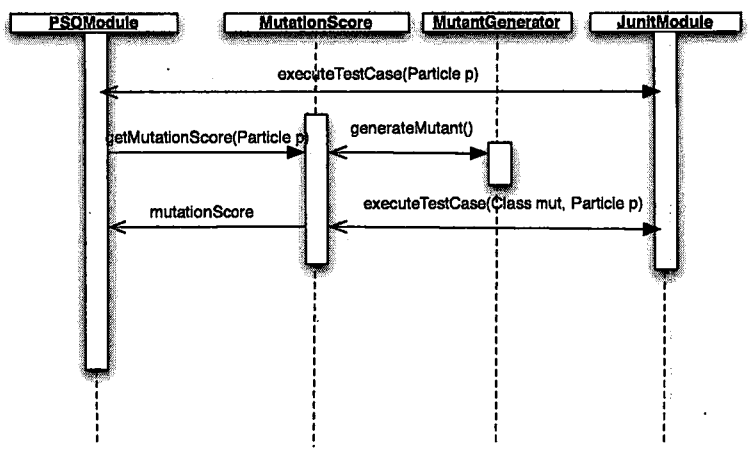

Figura D.3: Diagrama de sequência para o processo de calculo do *mutation score* de um determinado caso de teste

Apêndice E

Diagramas UML para o componente PSOModule

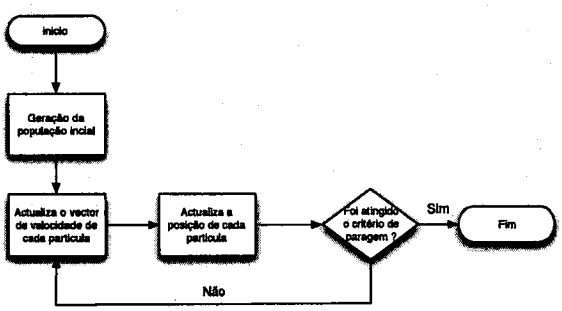

Figura E.1: Fluxograma do módulo PSOModule

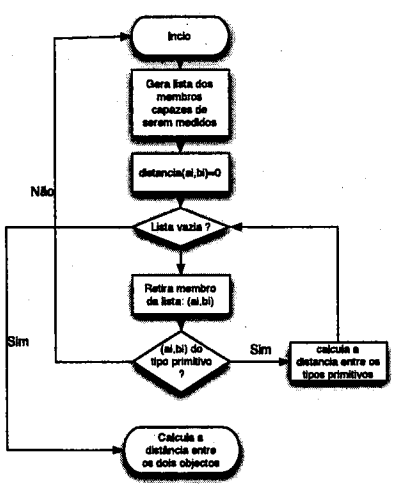

Figura E.2: Fluxograma do algoritmo de cálculo da distância entre dois objectos

Apêndice F

Diagramas UML o componente DataGeneration

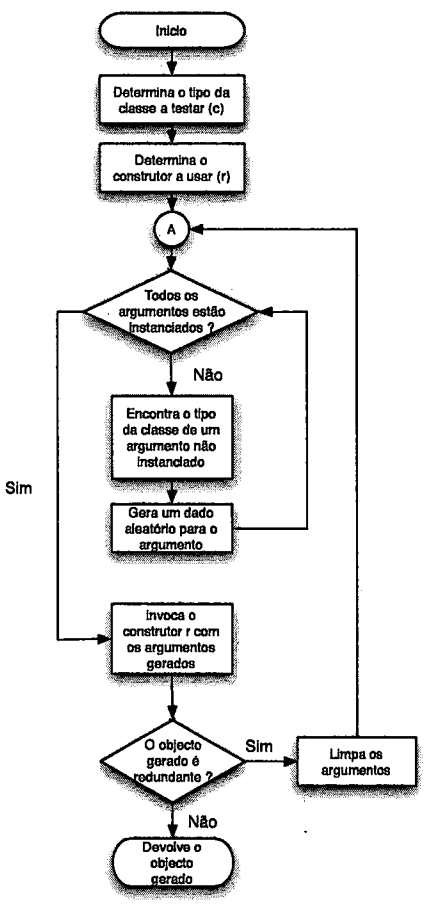

Figura F.1: Fluxograma do algoritmo descrito na lista 3.2

Apêndice G

Diagramas UML para o componente JUnitModule

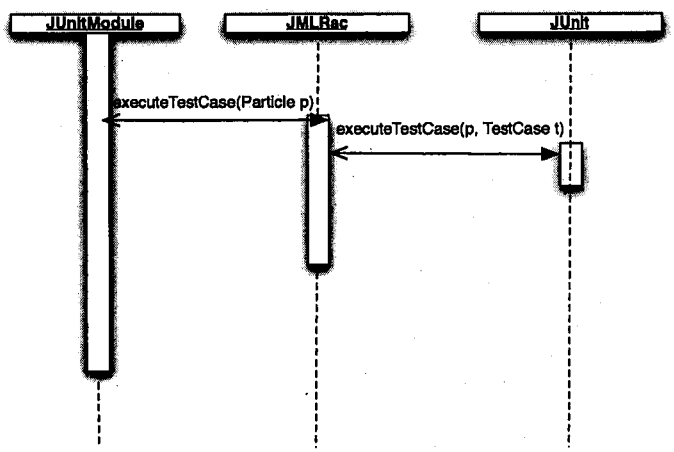

Figura G.1: Diagrama de sequência para o processo de verificação do sucesso de um caso de teste

Apêndice H

Linguagem de especificação usada na ferramenta Perfect Developer

Lista H.1: Exemplo de um caso da linguagem de especificação usada na ferramenta Perfect Developer

```
1
2  final class Queue of X ^=
3  abstract
4    var b: seq of X, // the queue data
5         maxLen: nat > 0; // maximum items in the queue
6
7    invariant #b <= maxLen;
8
9  interface
10   function empty: bool // test if the queue is empty
11     ^= #b = 0;
12
13   schema !add(x: X) // add an element to the end of the queue
14     pre ~full
15     post b!= b.append(x);
16
17   function full: bool // test if the queue is full
18     ^= #b = maxLen;
19
20   schema !remove(x!: out X) // remove the head element
21     pre ~empty
22     post x! = b.head,
```

```
23          b! = b.tail;
24
25    build{!maxLen: nat, dummy: X} // build an empty queue
26      pre maxLen ~= 0
27      post b! = seq of X{};
28
29    ghost operator =(arg); // we do not evaluate equality at run-time
30
31    // Verify that after adding an element, a queue is not empty
32    property (x: X)
33      pre ~full
34      assert ~(self after it!add(x)).empty;
35
36  // Verify that if we add an element to an empty queue,
37  // the next element we remove will be the one we added
38    ghost schema !addToEmptyThenRemove(e: X, r!: out X)
39      pre empty
40      post !add(e) then !remove(r!)
41      assert r' = e;
42
43  end;
```

Apêndice I

Exemplo de classe Java

Lista I.1: Exemplo de classe Java

```
1
2
3
4  class Bag {
5     int[] a;
6      int n;
7
8     Bag(int[] input) {
9        n = input.length;
10       a = new int[n];
11       System.arraycopy(input, 0, a, 0, n);
12     }
13
14     int extractMin() {
15      int m = Integer.MAX_VALUE;
16      int mindex = 0;
17       for (int i = 1; i <= n; i++) {
18        if (a[i] < m) {
19           mindex = i;
20           m = a[i];
21        }
22      }
23      n--;
24       a[mindex] = a[n];
25      return m;
26    }
27  }
```

Lista I.2: Resultado da analise à classe Bag que se encontra em M

```
1  Bag.java:6: Warning: Possible null dereference (Null)
2      n = input.length;
3              ^
4  Bag.java:15: Warning: Possible null dereference (Null)
5        if (a[i] < m) {
6            ^
7  Execution trace information:
8      Completed 0 loop iterations in "Bag.java", line 14, col 4.
9  Bag.java:15: Warning: Array index possibly too large (IndexTooBig)
10       if (a[i] < m) {
11           ^
12 Execution trace information:
13     Completed 0 loop iterations in "Bag.java", line 14, col 4.
14
15 Bag.java:21: Warning: Possible null dereference (Null)
16     a[mindex] = a[n];
17                 ^
18 Execution trace information:
19     Completed 0 loop iterations in "Bag.java", line 14, col 4.
```

Apêndice J

Exemplo de classe em Java

Lista J.1: Exemplo de uma classe Java

```
1  public class Door {
2    private boolean closed = true;
3    private boolean locked = true;
4
5    public boolean isOpen() {
6      return !closed;
7    }
8
9    public void open() {
10     if (!locked)
11       closed = false;
12   }
13
14   public void close() {
15     closed = true;
16   }
17
18   public void unlock() {
19     locked = false;
20   }
21
22   public void lock() {
23     if (closed)
24       locked = true;
25   }
26 }
```

Apêndice K

Caso de teste em Junit

Lista K.1: Exemplo de um caso de teste em Junit

```
1  import junit.framework.TestCase;
2
3  public class DoorTest extends TestCase {
4    private Door door = new Door();
5
6    public static void main(String[] args) {
7      junit.textui.TestRunner.run(DoorTest.class);
8    }
9
10   public void setUp() {
11     door.unlock();
12     door.open();
13   }
14
15   public void tearDown() {
16     door.close();
17     door.lock();
18   }
19
20   public void testAbrirFechar() {
21     assertTrue("A porta devia estar inicialmente aberta", door.isOpen());
22     assertEquals("A porta devia estar inicialmente aberta", true, door
23         .isOpen());
24     door.close();
25     assertFalse("Ao fechar uma porta aberta esta continuou aberta", door
26         .isOpen());
27     door.close();
28     assertFalse("Ao fechar duas vezes a porta esta ficou aberta", door
29         .isOpen());
30     door.open();
```

```
31      assertTrue("Ao␣abrir␣uma␣porta␣apenas␣fechada␣esta␣continuou␣fechada",
32          door.isOpen());
33      door.open();
34      assertTrue("Ao␣abrir␣duas␣vezes␣a␣porta␣esta␣ficou␣fechada", door
35          .isOpen());
36    }
37
38    public void testTrancarDestrancar() {
39      assertTrue("A␣porta␣devia␣estar␣inicialmente␣aberta", door.isOpen());
40      door.close();
41      door.lock();
42      assertFalse("Ao␣fechar␣e␣trancar␣uma␣porta␣aberta,␣esta␣ficou␣aberta",
43          door.isOpen());
44      door.open();
45      assertFalse("Foi␣possivel␣abrir␣uma␣porta␣que␣foi␣trancada", door
46          .isOpen());
47      door.unlock();
48      assertFalse(
49          "O␣destrancar␣de␣uma␣porta␣fez␣com␣que␣esta␣ficasse␣aberta",
50          door.isOpen());
51      door.open();
52      assertTrue("Ao␣destrancar␣e␣abrir␣uma␣porta,␣esta␣permaneceu␣fechada",
53          door.isOpen());
54    }
55
56  }
```

Apêndice L

Exemplo de código Java

Lista L.1: Classe Java

```
1  package example;
2
3  public class Mover
4  {
5    protected final int SLOWER = 5;
6
7    protected int x,y;
8
9    public void move(String direction, int speed)
10   {
11     if (direction.equals("up")) {
12       y -= speed;
13     }
14     else if (direction.equals("down")) {
15       y += speed;
16     }
17     else if (direction.equals("left")) {
18       x -= speed / SLOWER;
19     }
20     else if (direction.equals("right")) {
21       x += speed / SLOWER;
22     }
```

```
23      else {
24        throw new RuntimeException("illegal direction: "+direction);
25      }
26    }
27
28    public String toString()
29    {
30      return Integer.toString(x) + "," + Integer.toString(y);
31    }
32
33    public String prettyString()
34    {
35      return "(" + Integer.toString(x) + "," + Integer.toString(y) + ")";
36    }
37  }
```

Lista L.2: Caso de teste para a classe definida na lista L.1

```
1   package example;
2
3   import junit.framework.TestCase;
4
5   public class MoverTest extends TestCase
6   {
7     private Mover mover;
8
9     public void setUp()
10    {
11      mover = new Mover();
12    }
13
14    public void testUp()
15    {
16      mover.move("up", 2);
17      assertEquals("0,-2", mover.toString());
18    }
```

```
19
20    public void testDown()
21    {
22      mover.move("down", 2);
23      assertEquals("0,2", mover.toString());
24    }
25
26    public void testLeft()
27    {
28      mover.move("left", 2);
29      assertEquals("0,0", mover.toString());
30    }
31
32    public void testRight()
33    {
34      mover.move("right", 2);
35      assertEquals("0,0", mover.toString());
36    }
37  }
```

Apêndice M

Programa feito na linguagem C

```
1  main(argc, argv)
2  int argc;
3  char *argv[];
4  {
5    int c=0;
6      if(atoi(argv[1]) < 3){
7        printf("Got less then 3\n");
8        if(atoi(argv[2]) > 5)
9          c = 2;
10     }
11     else
12       printf("Got more then 3\n");
13   exit(0);
```

Figura M.1: Exemplo de código feito na linguagem C

```
1  [npalma@guppy mutation_testing]$ sh run_tests.sh original
2  input 2 4
3  Got less then 3
4  input 4 4
5  Got more then 3
6  input 4 6
7  Got more then 3
8  input 2 6
9  Got less then 3
10 input 4
11 Got more then 3
```

Figura M.2: Resultado de executar 5 casos de teste no programa

Apêndice N

Mutação 1 aplicada ao Anexo M

```
1  main(argc, argv)
2  int argc;
3  char *argv;
4  {
5    int c=0;
6      if(atoi(argv1) < 3){
7        printf("Got less then 3\n");
8        if(atoi(argv2) <= 5)
9          c = 2;
10     }
11     else
12       printf("Got more then 3\n");
13   exit(0);
```

Figura N.1: Mutante

```
1  [npalma@guppy mutation_testing]$ sh run_tests.sh mutant1
2  input 2 4
3  Got less then 3
4  input 4 4
5  Got more then 3
6  input 4 6
7  Got more then 3
8  input 2 6
9  Got less then 3
10 input 4
11 Got more then 3
```

Figura N.2: Resultado de executar 5 casos de teste no programa alterado

Apêndice O

Mutação 2 aplicada ao Anexo M

```
1  main(argc, argv)
2  int argc;
3  char *argv;
4  {
5    int c=0;
6      if(atoi(argv1) >= 3){
7        printf("Got less then 3\n");
8        if(atoi(argv2) > 5)
9          c = 2;
10     }
11     else
12       printf("Got more then 3\n");
13   exit(0);
```

Figura O.1: Mutante

```
1  [npalma@ninetails mutation_testing]$ sh run_tests.sh mutant2
2  input 2 4
3  Got more then 3
4  input 4 4
5  Got less then 3
6  input 4 6
7  Got less then 3
8  input 2 6
9  Got more then 3
10 input 4
11 Got less then 3
12 run_tests.sh: line 10:  4646 Segmentation fault      ./$1 4
```

Figura O.2: Resultado de executar 5 casos de teste no programa alterado

Apêndice P

Algoritmo de Optimização por Bando de Partículas

Lista P.1: Algoritmo de Optimização por Bando de Partículas

```
1  For each particle
2      Initialize particle
3  End
4  Do
5      For each particle
6          Calculate fitness value
7          If the fitness value is better than the best fitness value (pBest) in history
8              set current value as the new pBest
9      End
10
11     Choose the particle with the best fitness value of all the particles as the gBest
12     For each particle
13         Calculate particle velocity according equation  (1)
14         Update particle position according equation  (2)
15     End
16 While maximum iterations or minimum error criteria is not attained
```

Lista P.2: Fórmula para encontrar a velocidade e posição

```
1  (1)  v[] = v[] + c1 * rand() * (pbest[] − present[]) + c2 * rand() * (gbest[] − present[])
2  (2)  present[] = present[] + v[]
3
4  v[] representa a velocidade da partícula, present[] é a partícula actual. rand () é um valor aleatório
       entre 0 e 1. c1, c2 são os factores de aprendizagem, sendo normalmente c1 = c2 = 2.
```

Apêndice Q

Exemplo de classe java

Lista Q.1: Exemplo de classe Java

```
1
2  public class Triangle {
3
4  private static final String ILLEGAL_ARGUMENTS = "ILLEGAL_ARGUMENTS";
5    private static final String ILLEGAL = "ILLEGAL";
6
7
8    private static final String SCALENE = "SCALENE";
9    private static final String EQUILATERAL = "EQUILATERAL";
10   private static final String ISOSCELES = "ISOSCELES";
11
12
13   //@ requires side1 > 0 && side2 > 0 && side3 > 0;
14   //@ requires side1 + side2 >= side3;
15   //@ requires side2 + side3 >= side1;
16   //@ requires side1 + side3 >= side2;
17
18   //@ ensures \result.equals("SCALENE") || \result.equals("EQUILATERAL") || \result.equals("ISOSCELES");
19   public static String getType(int side1 , int side2, int side3)
20   {
21     return getType (new int[] { side1, side2, side3 });
22   }
23
24
25   public static String getType(int[] sides)
26   {
27     if (sides.length != 3)
28       return ILLEGAL_ARGUMENTS;
```

```
29      if (sides[0] < 0 || sides[1] < 0 || sides[2] < 0)
30        return ILLEGAL_ARGUMENTS;
31      int triang = 0;
32      if (sides[0] == sides[1])
33        triang = triang + 1;
34      if (sides[1] == sides[2])
35        triang = triang + 2;
36      if (sides[1] == sides[2])
37        triang = triang + 3;
38      if (triang == 0)
39      {
40        if (sides[0] + sides[1] < sides[2] ||
41            sides[1] + sides[2] < sides[0] ||
42            sides[0] + sides[2] < sides[1])
43          return ILLEGAL;
44        else
45          return SCALENE;
46    }
47
48      if (triang > 3)
49        return EQUILATERAL;
50      else if (triang == 1 && sides[0] + sides[1] > sides[2])
51        return ISOSCELES;
52      else if (triang == 2 && sides[0] + sides[2] > sides[1])
53        return ISOSCELES;
54      else if (triang == 3 && sides[1] + sides[2] > sides[0])
55        return ISOSCELES;
56      return ILLEGAL;
57    }
58    public static void main(String[] args) {
59    System.out.println(getType(2,2,2));
60    }
61  }
```

Lista Q.2: Exemplo de classe Java

```
1
2   public class TriangleWithError {
3
4   private static final String ILLEGAL_ARGUMENTS = "ILLEGAL_ARGUMENTS";
5     private static final String ILLEGAL = "ILLEGAL";
6
7
8     private static final String SCALENE = "SCALENE";
9     private static final String EQUILATERAL = "EQUILATERAL";
```

```
10  private static final String ISOSCELES = "ISOSCELES";
11
12
13  //@ requires p.side1 > 0 && p.side2 > 0 && p.side3 > 0;
14  //@ requires p.side1 + p.side2 >= p.side3;
15  //@ requires p.side2 + p.side3 >= p.side1;
16  //@ requires p.side1 + p.side3 >= p.side2;
17
18  //@ ensures \result.equals("SCALENE") || \result.equals("EQUILATERAL") || \result.equals("ISOSCELES");
19  public static String getType(Particle p)
20  {
21    return getType (p);
22  }
23
24
25  public static String getType(Particle p)
26  int sides[]={p.side1,p.side2,p.side3};
27  {
28    if (sides.length != 3)
29      return ILLEGAL_ARGUMENTS;
30    if (sides[0] < 0 || sides[1] < 0 || sides[2] < 0)
31      return ILLEGAL_ARGUMENTS;
32  if (((sides[0] * sides[1] * sides[2]))>=500 && (sides[0]<10 || sides[1] >= 10)) {
33    if (((sides[0] * sides[1] * sides[2]))>=5000) {
34      if (((sides[0] * sides[1] * sides[2]))>=50000) {
35        if (((sides[0] * sides[1] * sides[2]))> 500000) {
36          if (sides[1] >sides[2])
37            if (sides[1] >sides[2]) {
38              for (int i=0;i<=10;i++) {
39              }
40              System.out.println(sides[0]+","+sides[1]+","+sides[2]+",_VAI_FALHAR");
41              return "ERROR";
42            }
43        }
44      }
45    }
46  }
47
48    int triang = 0;
49    if (sides[0] == sides[1])
50      triang = triang + 1;
51    if (sides[1] == sides[2])
52      triang = triang + 2;
53    if (sides[1] == sides[2])
```

```
54        triang = triang + 3;
55      if (triang == 0)
56      {
57        if (sides[0] + sides[1] < sides[2] ||
58            sides[1] + sides[2] < sides[0] ||
59            sides[0] + sides[2] < sides[1])
60          return ILLEGAL;
61        else
62          return SCALENE;
63      }
64
65      if (triang > 3)
66        return EQUILATERAL;
67      else if (triang == 1 && sides[0] + sides[1] > sides[2])
68        return ISOSCELES;
69      else if (triang == 2 && sides[0] + sides[2] > sides[1])
70        return ISOSCELES;
71      else if (triang == 3 && sides[1] + sides[2] > sides[0])
72        return ISOSCELES;
73      return ILLEGAL;
74    }
75    public static void main(String[] args) {
76    Particle p=new Particle(2, 2, 2);
77    System.out.println(getType(p));
78    }
79  }
```

Lista Q.3: Exemplo de classe Java

```
1
2   public class Particle {
3     public int side1;
4     public int side2;
5     public int side3;
6
7
8     public Particle (int side1, int side2, int side3){
9       this.arg0=side1;
10      this.arg1=side2;
11      this.arg2=side3;
12
13    }
14
15    public String toString(){
16      return side1 +","+side2+","+side3;
```

```
17
18    }
19
20  }
```

Bibliografia

[ABGJ02] Roger T. Alexander, James M. Bieman, Sudipto Ghosh, and Bixia Ji. Mutation of java objects. *IEEE International Symposium Software Reliability Engineering*, 2002.

[ADH+89] H. Agrawal, R. Demillo, R. Hathaway, Wm. Hsu, Wynne Hsu, E. Krauser, R. J. Martin e A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[Agi07] Agitar. http://www.agitar.com/solutions/why_unit_testing.html, 2007.

[Blo07] Ryan Block. http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/, 2007.

[BR98] W. Banzhaf and C. Reeves. *Foundations of Genetic Algorithms.* Prentice Hall, 1998.

[Bra07] Ryan Brase. Avoid these java inheritance gotchas. http://articles.techrepublic.com.com/5100-22-5031837.html, 2007.

[Bri02] Riann Brits. Niching strategies for particle swarm optimization. Master's thesis, University of Pretoria, 2002.

[Bud80] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven, CT, 1980.

[Bud01] T. A. Budd. *Mutation Analysis: Ideas, Example, Problems and Prospects.* North-Holand Publishing Company, 2001.

[Bybro] Mattias Bybro. A mutation testing tool for java programs. Master's thesis, Royal Institute of Technology, Mattias Bybro.

[C4J07] C4J. http://c4j.sourceforge.net/, 2007.

[Cfo07] Cfo. http://www.cfo.com/article.cfm/3003844, 2007.

[Cha07] Control Chaos. Control chaos. http://www.controlchaos.com/, 2007.

[Cin07] Pablo Cingolani. Jswarm-pso. http://jswarm-pso.sourceforge.net/, 2007.

[CLOM06] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. Technical report, ETH Zurich, 2006.

[Con07] DSDM Consortium. Dsdm consortium. http://www.dsdm.org/, 2007.

[Cow07] Keith Cowingr. Nasa decides that a software error doomed the mars global surveyor spacecraft. http://www.spaceref.com/news/viewnews.html?id=1185, 2007.

[CRM07] Yoonsik Cheon and Carlos E. Rubio-Medrano. Random test data generation for java classes annotated with jml specifications. Technical report, IThe University of Texas, 2007.

[D99] Ballard D. *An Introduction to Natural Computation.* MIT Press, 1999.

[dCZ03] L. N. de Castro, Von Zuben, and F. J. (2004). *Software Testing Fundamentals: Methods and Metrics.* Idea Group Publishing, 2003.

[EB99] Guy Theraulaz Eric Bonabeau, Marco Dorigo. *Swarm Intelligence, From Natural to Artificial Systems.* Oxford USA Trade, 1999.

[EB01] Christopher Meyer Eric Bonabeau. Swarm intelligence. *Havard Business Review*, R0105G, 2001.

[Ei90] Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology.* IEEE, 1990.

[Fri75] A. D. Friedman. *Logical Design of Digital Systems.* Computer Science Press, 1975.

[GBP03] Diego Rodrigo Grein, Luiz Augusto Bergmann, and Otvio Rodolfo Piske. Testes de softwares. Technical report, Universidade do Contestado, 2003.

[hop07] Computer hope. http://www.computerhope.com/history/windows.htm, 2007.

[Hu07] Xiaohui Hu. http://www.swarmintelligence.org/, 2007.

[Hut04] Marnie L. Hutcheson. *Recent Developments in Biologically Inspired Computing.* Idea Group Publishing, 2004.

[IEE90] IEEE. Standard glossary of software engineering terminology. *IEEE Computer Society Press*, 1990.

[Jef07] Ron Jeffries. http://www.xprogramming.com/xpmag/whatisxp.htm, 2007.

[JK95] Russell Eberhart James Kennedy. Particle swarm optimization. *Proceedgins of the IEEE International Conference of Neural Networks*, IV:1992, 1995.

[Jum07] Jumble. http://jumble.sourceforge.net/, 2007.

[JUn07] JUnit. http://www.junit.org, 2007.

[Kin07] Joe Kiniry. Esc/java2. http://secure.ucd.ie/products/opensource/ESCJava2/, 2007.

[Kni07a] Konstantin Knizhnik. Find bugs in java programs. http://jlint.sourceforge.net, 2007.

[Kni07b] Konstantin Knizhnik. Find bugs in java programs. http://www.ispras.ru/ knizhnik/jlint/ReadMe.htm, 2007.

[Kol99] Adam Kolawa. Mutation of java objects. *STAREAST '99*, 1999.

[Lev95] Nancy Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[Lev07] Levenshtein. http://www.merriampark.com/ld.htm, 2007.

[Mad03] Per Madsen. Testing by contract - combining unit testing and design by contract. Technical report, Institute of Computer Science, Aalborg University, 2003.

[Mey07] Bertrand Meyer. Building bug-free o-o software: An introduction to design by contract(tm). http://archive.eiffel.com/doc/manuals/technology/contract/, 2007.

[Mic07] Sun Microsystems. http://java.sun.com/, 2007.

[MO05a] Yu-Seung Ma and Jeff Offutt. Description of class mutation mutation operators for java. mujava, 2005.

[MO05b] Yu-Seung Ma and Jeff Offutt. Description of method-level mutation operators for java. mujava, 2005.

[Moo07] Gordon E. Moore. http://www.intel.com/technology/mooreslaw/index.htm, 2007.

[MSS06] Michael Meissner, Michael Schmuker, , and Gisbert Schneider1. Optimized particle swarm optimization (opso) and its application to artificial neural network training. Technical report, Johann Wolfgang Goethe-Universität, 2006.

[Mye04] Glenford J. Myers. *The art of software testing.* John Wiley & Sons, Inc., 2004.

[oCNIoST02] Dept. of Commerces National Institute of Standards and Technology. Nist planning report 02-3, the economic impacts of inadequate infrastructure for software testing. Technical report, Dept. of Commerces National Institute of Standards and Technology, 2002.

[Pat07] Java PathFinder. http://javapathfinder.sourceforge.net/, 2007.

[Pee07] Edwin Peer. http://cilib.sourceforge.net/, 2007.

[RAD78] F. G. Sayward R. A. DeMillo, R. J. Lipton. Hints on test data selection: Help for the practicing programmer. *IEEE Computer, 11(4):3443*, 1978.

[Rie07] Johannes Rieken. Design by contract for java - revised. Master's thesis, Olden-Burg Falkutat II, 2007.

[Tec07] Escher Technologies. Perfect developer. http://www.eschertech.com, 2007.

[TW98] Bernard Pagurek Tony White. Towards multi-swarm problem solving in networks. *Systems and Computer Engineering, Carleton University*, 1998.

[Two07] Reel Two. www.reeltwo.com, 2007.

[TXN04] D. Marinov T. Xie and D. Notkin¡. Rostra: A framework for detecting redundant object-oriented unit tests. *EEE Conf. on Automated Sofware Engineering (ASE'04).*, 2004.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker,. Technical report, NASA, 2000.

[Wil07] Joe Wilcox. http://news.zdnet.co.uk/software/0,1000000121,2083814,00.htm, 2007.

[WM43] Walter Pitts Warren McCulloch. *A Logical Calculus of Ideas Immanent in Nervous Activity.* Bulletin of Mathematical Biophysics, 1943.