UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

AUTOMATIZAÇÃO DE TESTES DE MUTAÇÃO EM JAVA

Sheilla Cristina Fernandes Simões

PROJETO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Sistemas de Informação

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

AUTOMATIZAÇÃO DE TESTES DE MUTAÇÃO EM JAVA

Sheilla Cristina Fernandes Simões

PROJETO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Sistemas de Informação

Trabalho orientado pelo Prof. Doutor Francisco Cipriano da Cunha Martins

2014

Agradecimentos

Agradeço em primeiro lugar a DEUS, por me ter ajudado a ultrapassar todas as dificuldades, e permitir a conclusão deste trabalho.

Agradeço de forma muito especial ao meu orientador Professor Doutor Francisco Cipriano da Cunha Martins, por ter acreditado em mim, por todo o apoio e dedicação prestados nos momentos que mais precisei, pois sem isto não conseguiria concluir esta etapa da minha vida. Muito Obrigada Professor!

Quero em seguida agradecer, as pessoas mais importantes da minha vida, os meus queridos pais Joaquim Becker e Mariana Simões, as minhas irmãs Rosália Simões, Joelma Simões e Iracelma Fernandes, ao meu namorado Marco Prado, ao meu Avô e a resto da minha família. Muito Obrigada por todo o incentivo, disponibilidade, compreensão, suporte, apoio nos momentos mais difíceis, por me ajudarem a acreditar que era possível terminar o meu trabalho, pelo carinho e dedicação, que me presenteiam ao longo da minha vida.

Agradeço também de forma muito especial aos meus colegas, companheiros e amigos, Neida Moreno, Badjinca Baticam, por todos os momentos que passamos juntos, por terem sido um suporte nos momentos difíceis, pelo carinho e a partilha deste bem precioso que é o conhecimento.

Quero agradecer ainda aos meus amigos pelo apoio e incentivo que me deram e a todos os docentes da Faculdade de Ciências da Universidade de Lisboa que fizeram parte do meu percurso académico.

Aos meus pais e a todos aqueles que contribuíram para a minha formação.

Resumo

A importância da execução de testes durante o processo de desenvolvimento de *software* tem vindo a crescer, uma vez que estes são cruciais para garantir a qualidade do *software* desenvolvido, representando o principal método para a redução de ocorrência de erros. Isto tem um carater fundamental, pois no nosso dia-a-dia o *software* é ubíquo, e indispensável à tecnologia que dependemos. É, pois, fundamental garantir que o *software* possua um nível de qualidade elevado, pois uma falha poderá causar graves consequências a nível financeiro ou até mesmo ao nível humano. Torna-se assim imprescindível que os engenheiros de *software* dominem as técnicas de testes de *software*.

O presente trabalho tem como objetivo construir uma ferramenta para apoiar o ensino de testes de mutação. Esta tem sido muito explorada recentemente e tem uma eficácia comprovada na atividade de testes. Os testes de mutação são uma técnica baseada em faltas e são utilizados para testar a robustez de um conjunto de casos de teste, com base no número de falhas identificadas. Cada uma das faltas produzidas por esta técnica representa um requisito de teste, que neste modelo de testes é designado por mutante. No entanto, o número de mutantes gerados é de tal ordem extenso que não pode ser sistematicamente tratado de forma manual. Daí a importância da utilização de uma ferramenta para este tipo de testes.

A concretização deste trabalho resultou no desenvolvimento de uma ferramenta integrada com o ambiente de desenvolvimento Eclipse, de grande utilização tanto a nível profissional como a nível académico. Este documento apresenta aspetos relacionados com a ferramenta desenvolvida, nomeadamente, os conceitos teóricos essenciais à sua implementação; a sua integração na plataforma Eclipse e as dependências com outros elementos desta plataforma; a descrição da sua estrutura interna: a apresentação de detalhes da implementação de processos fundamentais para a concretização das funcionalidades presentes na ferramenta; e a descrição dos elementos da interface gráfica que permitem acompanhar as várias fases que compõem o processo de testes de mutação, como por exemplo, apresentação dos mutantes gerados, a pontuação dos testes de mutação, entre outras informações.

Palavras-chave: Testes de Mutação, Mutantes, Engenheiro de *Software*, Eclipse, Ensino.

Abstract

The importance of running tests during the development process of software has been growing, as these are crucial to ensuring the quality of software developed, representing the primary method for reducing errors. This has a fundamental character, because in our day- to-day software is ubiquitous and essential to the technology we depend. It is therefore essential to ensure that the software has a high level of quality because a failure may cause serious consequences in financial terms or even the human level. It thus becomes essential that software engineers master the techniques of software testing.

This paper aims to build a tool to support the teaching of mutation testing. This has been explored very recently and has a proven efficacy in testing activity. Changing tests are based on technical faults and are used to test the robustness of a set of test cases based on the number of identified faults. Each of faults produced by this technique is a requirement of testing in this test model is designated mutant. However, the number of mutants generated is so extensive that the order cannot be systematically handled manually. Hence the importance of using a tool for this type of testing.

The completion of this work resulted in the development of an integrated tool with the Eclipse development environment, great use both professionally and academically. This paper presents aspects related to the developed tool, in particular, the essential theoretical concepts to their implementation; its integration into the Eclipse platform and dependencies with other elements of this platform; a description of their internal structure: the submission of details of the implementation of key processes for the realization of the features present in the tool; and the description of graphical interface elements for monitoring the various stages that make up the process of mutation tests, such as presentation of mutants generated, the scores of the mutation tests, among other information.

Keywords: Mutation Tests, Mutants, Software Engineer, Eclipse, Education.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As atividades que exercemos no nosso dia-a-dia têm cada vez mais o auxílio de dispositivos que integram *software*. Para nós é fundamental que o *software* funcione corretamente, pois uma falha na sua execução pode originar danos muito graves ao nível financeiro, material, ou até chegar a pôr em risco vidas humanas. Por isso, é muito importante garantir a qualidade do *software*.

A Engenharia de *Software* integra várias atividades de garantia de qualidade no processo de construção de um produto, como por exemplo, a realização de testes de verificação e validação do produto, com a finalidade de minimizar os riscos de possíveis problemas causados pelo funcionamento incorreto do produto. Desenvolver um *software* de alta qualidade e a baixo custo é um dos objetivos da Engenharia de *Software*. A definição de qualidade, na perspetiva de Pressman [1], é a "Concordância com os requisitos funcionais e não funcionais, com normas de desenvolvimento explicitamente documentados e com as características implícitas em todo o *software* desenvolvido profissionalmente."

Verificação, Validação e Teste (VV&T) são atividades que fazem parte do processo de Garantia de Qualidade de *Software*, com o objetivo de minimizar a ocorrência de erros e riscos associados a utilização do produto desenvolvido. A validação pretende garantir que o produto final construído corresponde aos requisitos do cliente. A verificação tem como objetivo garantir que um sistema ou um componente implementa corretamente uma função específica. O processo de teste avalia um sistema ou um componente de um sistema por meios manuais ou automáticos para verificar se este está em conformidade com os requisitos especificados.

Os testes de *software* representam um elemento fundamental para a verificação e eliminação de erros. Segundo Myers [2], "teste é o processo de executar um programa com a intenção de encontrar erros." O principal objetivo dos testes é revelar a presença de erros no *software* para que estes possam ser corrigidos atempadamente e deste modo aumentar a confiabilidade do produto. No entanto, a concretização da atividade de testes

pode ser bastante dispendiosa. Por exemplo, em alguns projetos pode utilizar-se entre 30% a 40% do esforço total e até 50% do orçamento total. O princípio de *Pareto*[1] afirma que há um equilíbrio entre causas e efeitos, esforços e resultados e entre ações e objetivos alcançados. Aplicando este princípio aos testes de *software*, podemos dizer que 20% dos componentes de um *software* concentram 80% das falhas produzidas. Este princípio sugere que os programadores devem priorizar as componentes críticas de um sistema na fase de teste.

Um caso de teste compreende um conjunto de valores de entrada, as condições de execução e os resultados esperados para o teste. O processo de elaboração de casos de teste pode produzir imensos casos de teste, tornando impossível a escrita de testes que possam cobrir todas as possibilidades existentes. Por este motivo, afirma-se que os testes permitem identificar a existência de falhas num determinado produto, mas não a sua ausência. Podemos comprovar esta afirmação através do seguinte exemplo:

```
int sucessor(int i){ … }
```

Supondo que o parâmetro *i* da função é um inteiro de 32 bits, são necessários 2^{32} testes diferentes para cobrir todas as possibilidades. Este exemplo mostra que cobrir todos os casos de teste há geralmente que empregar muito esforço.

Sendo a atividade de testes um elemento crítico no custo e na duração do projeto é imprescindível diminuir estes valores de forma a aumentar a sua eficácia. Para dar resposta a estes desafios, têm sido essenciais os estudos teóricos e o desenvolvimento de ferramentas [3, 4] para o aperfeiçoamento de técnicas e métodos de teste, dando um grande contributo à Engenharia de *Software*.

As técnicas de testes são utilizadas na atividade de teste para geração, seleção e avaliação de casos de teste e também avaliação da qualidade da própria atividade de teste, uma vez que o sucesso desta atividade em revelar falhas presentes no *software* está diretamente relacionado com a qualidade do conjunto de casos de teste que é utilizado. As técnicas de teste podem ser classificadas em teste baseados em expressões lógicas, em grafos e em faltas. Os testes baseados em expressões lógicas partem do princípio que o programador tem conhecimento da estrutura interna do sistema incluindo o código fonte, e os testes são projetados em função da estrutura de um componente do sistema. Isto permite que uma avaliação mais precisa do mesmo. Os testes baseados em grafos têm como base a informação sobre os requisitos funcionais do sistema. Estes são concebidos para avaliar o sistema na perspetiva do utilizador, e a sua aplicação permite a identificação de problemas relacionados como erros de interface,

[1] http://en.wikipedia.org/wiki/Pareto_principle

erros de desempenho, etc. Os teste baseados em faltas são uma técnica que utiliza a informação sobre o tipo de erros mais frequentes gerados durante o processo de desenvolvimento de *software* para definição de casos de teste, criados para que possam detetar estes erros. Os tipos de erros que podem ser gerados durante o desenvolvimento de um sistema são definidos com base nas características da linguagem de programação utilizada.

Com base em estudos teóricos e empíricos [3, 4], tem sido comprovado a eficácia para a avaliação de um conjunto de casos de testes, uma da técnica de testes baseada em faltas: os testes de mutação. Esta técnica consiste em efetuar pequenas alterações a um programa, que representam os tipos de faltas mais comuns que podem ser produzidas por um programador. Cada falta dá origem a uma nova versão do programa chamada de mutante. Com o objetivo de avaliar a adequação de um conjunto de casos de teste para testar um programa, estes são executados sobre o programa e sobre os seus mutantes. Os resultados obtidos são analisados para verificar se os casos de testes são capazes de descobrir as diferenças comportamentais existentes entre o programa e os seus mutantes. A eficácia de um conjunto de casos de testes é determinada com base no número de faltas que estes conseguem detetar, ou seja, no número de mutantes "mortos". As faltas causadas representam alterações sintáticas, que embora não causem erros sintáticos, modificam a semântica do programa conduzindo-o a um estado incorreto. O conjunto de faltas que pode ser introduzido em um determinado programa resulta da aplicação dos operadores de mutação. A Figura 1.1 apresenta o exemplo de uma mutação obtida da aplicação do operador de mutação que substitui um operador aritmético. Este consiste na alteração de um operador aritmético por outro; por exemplo, na expressão aritmética "x -= 3", uma das mutações que resultaria da aplicação deste operador seria a alteração de "-=" por "+=".

Código original	Mutante
x -= 3	x += 3

Figura 1.1 Exemplo de uma mutação

Contudo, está é uma técnica que tem um elevado custo computacional porque o número de mutantes gerados pode ser muito grande. Por este motivo os testes de mutação só podem ser concretizados eficazmente com o auxílio de uma ferramenta automatizada. Além disso, a automatização garante vários benefícios à atividade de testes, por exemplo, a redução de geração de erros devido à diminuição da intervenção humana. Isto permite aumentar a qualidade e a produtividade desta atividade que consequentemente será refletida no aumento da confiabilidade do sistema. Mothra [5, 6]

é o exemplo de uma das primeiras ferramentas proposta no contexto de testes de mutação.

Outro aspeto que deve ser tido em consideração é a linguagem utilizada para desenvolver o *software*, isto porque a aplicação de determinada técnica e método de teste deve ser feita tendo em conta as características da linguagem de programação utilizada. Por exemplo, os tipos de erros produzidos em sistemas desenvolvidos em linguagens Orientadas a Objetos (OO) são diferentes dos erros que podem ser introduzidos em sistemas concebidos numa linguagem procedimental.

A linguagem OO utilizada no contexto deste trabalho é a linguagem Java. No seguimento do trabalho, serão apresentados os motivos que estão na origem do desenvolvimento de um *plug-in* para aplicação da técnica de teste de mutação em programas Java.

1.1 Motivação

A confiabilidade de um *software* é o fator que determina a probabilidade deste operar sem que haja a ocorrência de falhas, durante um período de tempo num ambiente.

A atividade de teste quando planeada de forma sistemática e rigorosa, permite estimar a confiabilidade e a qualidade do produto construído.

Sabendo que durante definição/escrita de um programa OO são introduzidos erros, é importante que sejam executados testes para a identificação e correção dos erros. Idealmente, o programa deveria ser testado com todos os valores possíveis do domínio de entrada, no entanto devido à cardinalidade deste domínio, muitas vezes torna-se impraticável testar todas as condições possíveis, ou seja, aplicar testes exaustivos. Assim, sabendo que a atividade de testes não garante que um determinado programa está isento de erros, foram propostas técnicas de teste, como por exemplo, os testes de mutação, para a seleção de um subconjunto de valores com maior probabilidade na deteção de erros caso existam.

Torna-se proveitoso para a Engenharia de *Software*, combinar a utilização do paradigma OO, pelas vantagens que este paradigma oferece, com a aplicação da técnica de testes de mutação, a fim de obter uma medida objetiva do nível de confiança e de qualidade alcançados pelos testes realizados em programas OO. Contudo, está é uma técnica que tem um elevado custo computacional porque o número de mutantes gerados pode ser muito grande. Por este motivo os testes de mutação só podem ser concretizados eficazmente com o auxílio de uma ferramenta automatizada.

Um dos objetivos da Engenharia de Software é automatizar tanto quanto possível a atividade de testes, para garantir vários benefícios, como por exemplo, a redução de geração de erros devido à diminuição da intervenção humana e a redução do tempo para a realização dos testes. Isto permite aumentar a qualidade e a produtividade desta atividade que consequentemente será refletida no aumento da confiabilidade do sistema.

Englobar várias funcionalidades em uma ferramenta utilizada para a processo de desenvolvimento de *software* pode tornar mais simples algumas tarefas, como por exemplo, programar, escrever casos de teste, permite que sejam realizadas várias atividades deste processo utilizando uma única ferramenta. O Eclipse é um ambiente integrado de desenvolvimento (IDE) que representa a concretização deste tipo de ferramenta. Desenvolvido na linguagem Java, este *software* livre de código aberto, permite que as suas funcionalidades possam ser estendidas através da instalação de *plug-ins*.

O processo de teste de um sistema engloba várias atividades que são desempenhadas por profissionais, por exemplo, planeamento e controle, análise de resultados, revisão de documentação, escrita de relatórios, etc. Podemos dizer que a produtividade da atividade de teste depende da competência e das condições que são reunidas para que as várias tarefas do processo sejam executadas. Deste modo, o conhecimento e a experiência das pessoas integradas no processo são fundamentais para garantir o sucesso deste processo. Para isso, é fundamental que haja a disponibilização de materiais que permitam auxiliar na formação e capacitação destas pessoas. De acordo com Myers [2] a pesquisa de técnica de testes nos últimos anos trouxe grandes contributos para a Engenharia de *Software*, no entanto o autor salienta que a componente ensino não pode ser deixada de parte, ou seja, que estes conhecimentos obtidos nas pesquisas devem ser canalizados para a formação de profissionais. Assim, a realização da atividade de testes no processo de desenvolvimento de *software* poderá ser concretizada com mais robustez.

O custo associado a disponibilização, a configuração e a preparação de material necessário para o ensino prático das técnicas de teste representa um obstáculo para o ensino destas técnicas. Assim, a disponibilização de um ambiente integrado para a concretização das técnicas de teste em conjunto com manuais e documentação, poderá auxiliar no aprendizado de tais técnicas. A consolidação de conhecimento pode ser um grande contributo para a atividade de testes, pois permitirá que seja feita uma melhor avaliação na escolha da técnica a ser utilizada, tendo em conta os custos associados, o paradigma utilizado entre outros aspetos.

O *plug-in* PESTTMuTest (*Plug-in* Educational Software Testing Tool – Mutation Test) foi projetado para que possa ser utilizado a um nível académico, para auxiliar no

aprendizado dos testes de mutação, através da observação experimental da execução de testes de mutação num contexto real, bem como inferir a partir de dados experimentais obtidos.

1.2 Objetivos

Este trabalho propõe-se integrar a técnica de testes de mutação em linguagem Java na plataforma de desenvolvimento Eclipse, através do desenvolvimento de um *plug-in* designado PESTTMuTest. O objetivo da criação deste *plug-in* é facilitar o ensino desta técnica e permitir a concretização automatizada de testes de mutação em programas Java.

Pretende-se que a ferramenta seja extensível quanto aos operadores de mutação, ou seja, que possam ser incluídos novos operadores de mutação, uma vez que, o propósito deste trabalho não é implementar todos os operadores de mutação existentes para a linguagem Java, mas montar a infraestrutura global de testes de mutação.

Para que seja possível a execução de casos de testes, disponibilizados pelos utilizadores sobre os mutantes gerados pelo PESTTMuTest é necessário ligar o *plug-in* com o JUnit. O JUnit é uma plataforma de código aberto, utilizada para a escrita e execução de teste unitários para a linguagem de programação Java.

Em síntese, este trabalho tem como objetivo contribui para a atividade de testes de programas em Java, através da aplicação de testes de mutação que permitirão escrever casos de testes com maior garantia de qualidade e consequentemente avaliar a qualidade do *software* produzido.

1.3 Estrutura do documento

A presente dissertação foi estruturada em cinco capítulos para além desta introdução. No Capítulo 2 será apresentado o trabalho relacionado e também uma breve descrição das ferramentas de testes de mutação existentes para linguagem Java. O Capítulo 3 descreve de forma detalhada a técnica de testes de mutação, como é que esta é aplicada no paradigma OO e a descrição e exemplificação dos operadores de mutação que podem ser implementados para simular faltas em programas Java. No Capítulo 4 é apresentada a análise do problema e as atividades realizadas, e as decisões tomadas durante a implementação da ferramenta. Os testes utilizados para a avaliação do *plug-in* que elaborámos e os resultados obtidos são apresentados no Capítulo 5. Finalmente, o Capítulo 6 versa sobre a análise final do relatório, que inclui conclusões sobre o trabalho realizado e a identificação de trabalho futuro que poderá dar continuidade ao que versamos nesta tese.

Capítulo 2

Trabalho Relacionado

Nos anos 70 surgiram os primeiros trabalhos sobre testes de mutação, introduzidos pelos autores Budd e Sayward [7], Hamlet [4], DeMillo, Lipton e Sayward [3].

O artigo de DeMillo [3], publicado em 1978 foi um trabalho seminal na área e atraiu o interesse de vários investigadores para a temática dos testes [8]. Neste trabalho foram definidas duas hipóteses fundamentais para os testes de mutação: a hipótese do programador competente e o efeito de acoplamento.

A hipótese do programador competente assume que um programador "competente" tende a escrever programas muito próximos de um conjunto de programas corretos. Assim, ainda que estes programas possam conter erros, admite-se que estes erros são simples e que podem ser facilmente corrigidos através de pequenas alterações no código. É com base nesta hipótese que a técnica de testes de mutação, para simular faltas que podem ser introduzidas por programadores "competentes" num programa, aplica mudanças sintáticas simples no programa que não geram erros de sintaxe, mas que alterar a semântica do programa. Como suporte a esta hipótese, existem trabalhos relacionados com o conceito de vizinhança, tendo como exemplos, os trabalhos elaborados por Budd [7] e Geist [9].

O efeito de acoplamento afirma que se um de caso de teste é capaz de revelar faltas simples, então este poderá também detetar uma falta mais complexa, existindo assim um acoplamento entre as faltas simples e as faltas complexas. Deste modo, é possível avaliar a eficácia de um caso de teste em revelar faltas em um determinado programa utilizando a técnica de testes de mutação, pois se um caso de teste for capaz de revelar uma falta simples, então existe uma grande probabilidade deste revelar faltas mais complexas no mesmo programa. Uma falta simples é gerada por uma única alteração de código, enquanto uma falta complexa é criada por mais do que uma alteração sintática. Alguns estudos empíricos suportam esta hipótese, nomeadamente os trabalhos apresentados por Offutt em 1992 [10] e por Wah em 1995 [11].

Ao longo dos anos, os trabalhos relacionados com testes de mutação tiveram incidência nos seguintes pontos [12]: definição de operadores de mutação, experimentação, desenvolvimento de ferramentas e redução de custos associados a análise de mutação.

Os operadores de mutação são uma das partes fundamentais dos testes de mutação, e do qual depende a sua eficácia. Em 1977 foi proposto o primeiro desenho de operadores de mutação para a linguagem de programação Fortran IV. Em 1987, com base em trabalhos anteriores, os autores Offutt e King [13, 14] propuseram 22 operadores de mutação para a linguagem de programação Fortran 77, sendo este o primeiro conjunto de operadores de mutação a ser formalizado. Os trabalhos seguintes de definição de operadores de mutação para outras linguagens de programação foram influenciados por este conjunto de operadores. Por exemplo em 1988 Bowser [15] definiu operadores para a linguagem de programação Ada, em 1989 [16] foi proposto um conjunto amplo de operadores de mutação para a linguagem C.

O conjunto de operadores de mutação tradicional, não é suficiente para a realização de testes a linguagens OO, dada a diferença de estrutura apresentada pelas linguagens OO e pelas suas características. Por exemplo, herança e polimorfismo fundamentais no paradigma OO não são cobertas pelos operadores tradicionais. Os primeiros 20 operadores de mutação para a linguagem Java foram apresentados por Kim *et al.* [17]. Utilizando estudos empíricos para determinar a eficácia dos operadores de mutação, e com base em estudos anteriores, como por exemplo os estudos realizados por Kim, Ma *et al.* [18, 19] propôs 24 operadores de mutação para a linguagem Java, classificando-os em seis grupos: controlo de acesso, herança, polimorfismo, sobrecarga, características específicas da linguagem Java e erros comuns de programação imperativa.

Estudos empíricos comprovaram a eficácia dos testes de mutação em detetar faltas em comparação com outras técnicas, como por exemplo, fluxo de dados [20, 21]. Também tem sido importante na busca de métodos que permitam reduzir os custos associados à aplicação de testes de mutação, por exemplo, ajudar a entender a utilização de um determinado operador de mutação em determinado contexto, isto pode permitir reduzir o número de mutantes que devem ser gerados.

Para que os testes de mutação se tornem praticáveis é fundamental que sejam aplicadas com o auxílio de ferramentas. As primeiras ferramentas de testes de mutação desenvolvidas para fins académicos foram Mothra, Proteum e Tums. O sistema Mothra foi construído para a linguagem Fortran em meados dos anos 1980 na Georgia Tech [6, 22]. Continha a implementação de 22 operadores de mutação. DeMillo, Offutt, King, Krauser e Spafford foram os investigadores envolvidos no projeto Mothra, o auge deste sistema sucedeu na década de 1990 [1]. As ferramentas Proteum e Tums foram

construídas para a linguagem C [23]. Finalmente, nos anos mais recentes têm sido desenvolvidas ferramentas automatizadas para a aplicação de testes de mutação de natureza académica, por exemplo o MuJava [19], outras de código aberto como é o caso do Jester [24], e ainda outras para fins comerciais, como por exemplo o PlexTest, desenvolvido pela empresa ITRegister[2], para realizar testes de mutação em sistemas escritos na linguagem C++, ou a ferramenta PIT[3] para aplicações Java.

O elevado custo computacional necessário para executar cada caso de teste sobre um número elevado de mutantes gerados e também a quantidade de esforço que é preciso na realização dos testes de mutação são fatores que contribuem para a redução da utilização desta técnica. No entanto, vários estudos têm sido feitos com o intuito de promover a resolução dos problemas apresentados pela aplicação dos testes de mutação. Offutt e Untch [25] definiram três estratégias: "*do fewer*" – encontrar formas de reduzir o número de mutantes gerados sem perda significativa de informação; "*do smarter*" – procurar distribuir o custo computacional por várias máquinas (processadores); e "*do faster*" – encontrar métodos de executar os testes sobre os mutantes o mais rápido possível.

A redução do número de mutantes gerados, sem que haja perda significativa da eficácia dos testes de mutação, é uma das formas apresentadas para otimizar esta técnica, permitindo a redução do custo computacional. As quatro técnicas disponíveis para a redução do número de mutantes gerados são as seguintes: agrupamento de mutantes, amostragem de mutantes, mutação seletiva e mutação de ordem superior (HOM). Agrupamento de mutante baseia-se em algoritmos de agrupamento, onde é escolhido um subconjunto de mutantes. *K-means* e agrupamento em massa são exemplo de algoritmos de agrupamento. A técnica de amostragem de mutação consiste em escolher aleatoriamente um subconjunto a partir de um conjunto de mutantes. A redução do número dos mutantes gerados pode também ser feita através da seleção de um pequeno conjunto de operadores de mutação que permite gerar um subconjunto de todos os mutantes possíveis sem a perda significativa da eficácia dos testes. Por fim, a técnica da ordem superior de mutação consiste em aplicar mais do que um operador de mutação para gerar um mutante, isto dá origem a um mutante de ordem superior.

A otimização do processo de execução de um mutante é uma forma que permite a redução do custo computacional dos testes de mutação.

Assim, as técnicas que permitem otimizar o processo de execução de um mutante que têm como base a forma como se determina quando um mutante é morto podem ser

[2] http://www.itregister.com.au/

[3] http://pitest.org/

classificadas como: mutação forte, mutação fraca e mutação firme. A mutação forte considera que um mutante é morto quando produz um resultado diferente do programa original. Assumindo que um programa é constituído por um conjunto de componentes, na técnica de mutação fraca, para validar se um mutante foi morto, os mutantes são apenas verificados imediatamente após o ponto de execução do componente mutado. A mutação firme surgiu com o objetivo de ultrapassar as desvantagens apresentadas pelas duas técnicas referidas anteriormente adicionando possibilidades intermedias para a verificação de um mutante morto, assim a "comparação de estado" da mutação firme fica entre o estado intermédio apos a execução (mutação fraca) e o resultado final (mutação forte).

Outra das técnicas que pode ser utlizada para a otimização do processo da execução dos mutantes é através da otimização do tempo de execução. Podemos referir como exemplo a técnica baseada em intérprete, a técnica baseada na tradução de *bytecode* e técnica de esquema de mutante.

O teste de mutação paralela, a distribuição do custo computacional por vários processadores, a execução de mutantes em simultâneo sobre máquinas SIMD[4], a distribuição do custo de execução dos testes de mutação através da utilização de máquinas MIMD[5], são exemplos de suporte de plataformas avançadas para os testes de mutação que permitem a distribuição do custo computacional por várias máquinas.

O problema da determinação de um mutante equivalente está relacionado com elevada quantidade de esforço Humano envolvido na utilização dos testes de mutação. Dado um programa *p*, e sendo *m* um mutante gerado a partir de *p*, diz-se que *m* é um mutante equivalente se *m* é sintaticamente diferente de *p*, mas têm o mesmo comportamento. Sabe-se que para verificar se um programa e os seus mutantes são equivalentes é indecidível, o que implica a intervenção Humana para a identificação dos mutantes equivalentes. No entanto, têm sido realizados trabalhos com objetivo de definir técnicas que auxiliem na deteção de mutantes equivalentes.

Uma vez que o objetivo deste trabalho é o desenvolvimento de uma ferramenta de testes de mutação para dar suporte a sistemas Java, torna-se relevante fazer uma breve descrição de algumas ferramentas automatizadas existentes para a realização de testes de mutação em programas Java. Assim, podemos referir as seguintes ferramentas: MuJava, MuClipse, JesTer e Judy.

MuJava [19] é uma ferramenta que dispõe dos operadores de mutação OO e os operadores de mutação tradicionais. Nesta ferramenta foram implementadas duas

[4] http://en.wikipedia.org/wiki/SIMD

[5] http://en.wikipedia.org/wiki/MIMD

tecnologias para a otimização do tempo de execução: MSG [26] e tradução de *bytecode*. A geração de um mutante implica a definição de um novo ficheiro *.class*. MuJava, no entanto, apresenta algumas limitações, como por exemplo, não suporta o JUnit e só permite aplicações desenvolvidas com a versão do Java 1.5. Contudo, recentemente foi desenvolvido o *plug-in* MuClipse [27], que surge como uma forma de ligar o MuJava a plataforma Eclipse IDE. Este *plug-in* foi criado com o objetivo de simplificar a instalação e configuração da ferramenta, permitir a integração com o JUnit. Porém, algumas limitações são apresentadas pelo MuClipse, como por exemplo, no processo de teste não é possível selecionar em simultâneo mais do que uma classe para a geração dos mutantes. A Figura 2.1 e a Figura 2.3 apresentam a GUI do *plug-in* MuClipse, e a Figura 2.2 e a Figura 2.4 mostra a GUI da ferramenta MuJava.

A ferramenta Jester [24] apesar de suportar o JUnit e um conjunto de operadores de mutação apresenta problemas de performance e de confiabilidade. Isto porque, a execução dos testes de mutação demora muito tempo e ainda é necessário esforço manual para a interpretação dos resultados, o que poder originar erros.

Com o objetivo de reduzir os custos computacionais associados ao tempo de geração de mutantes, a ferramenta Jumble [28] aplica as alterações a nível de *bytecode*. Esta ferramenta está integrada com o JUnit e foi desenvolvida para ser integrada em um ambiente industrial, existindo também uma versão em *plug-in* desenvolvido para a plataforma Eclipse. Contudo, o número de operadores de mutação suportados pela Jumble é muito limitado, por exemplo, não suporta os operadores de mutação OO, permite aplicar apenas uma mutação das várias mutações que podem ser geradas por um operador de mutação tradicional.

Judy [12] é outra ferramenta desenvolvida com base no mecanismo de programação orientada a aspetos. O objetivo era conseguir aumentar o desempenho do processo de testes de mutação, implementar tanto quanto possível os operadores de mutação OO e os operadores de mutação tradicionais e integrar com uma ferramenta de ambiente de desenvolvimento profissional. Isto foi alcançado através da implementação de FAMTA Light, cuja concretização foi utilizada a linguagem Java e extensões AspectJ. Esta ferramenta foi desenvolvida de modo a que seja possível a integração de novos operadores de mutação que possam vir a ser implementados.

Concluindo, os diversos trabalhos propostos sobre a técnica de testes de mutação de carater prático e teórico tem sido fundamentais para a resolução de alguns dos problemas apresentados por esta técnica e também têm servido como incentivo para a realização de novos trabalhos que permitam a otimização desta técnica, a construção de ferramentas cada vez mais completas para a concretização de testes de mutação tanto a nível académico como profissional.

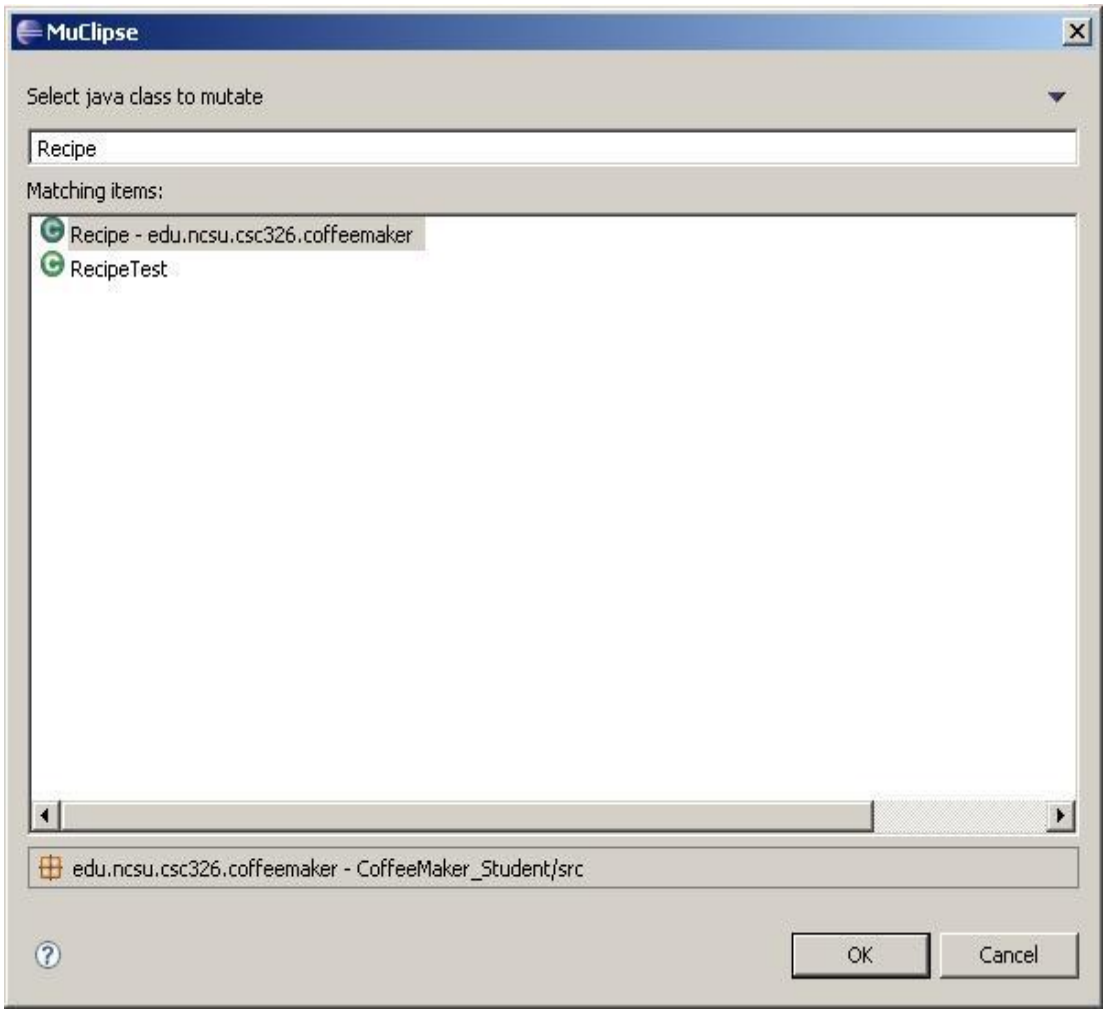

Figura 2.1 Interface gráfica para selecionar uma classe em MuClipse [41]

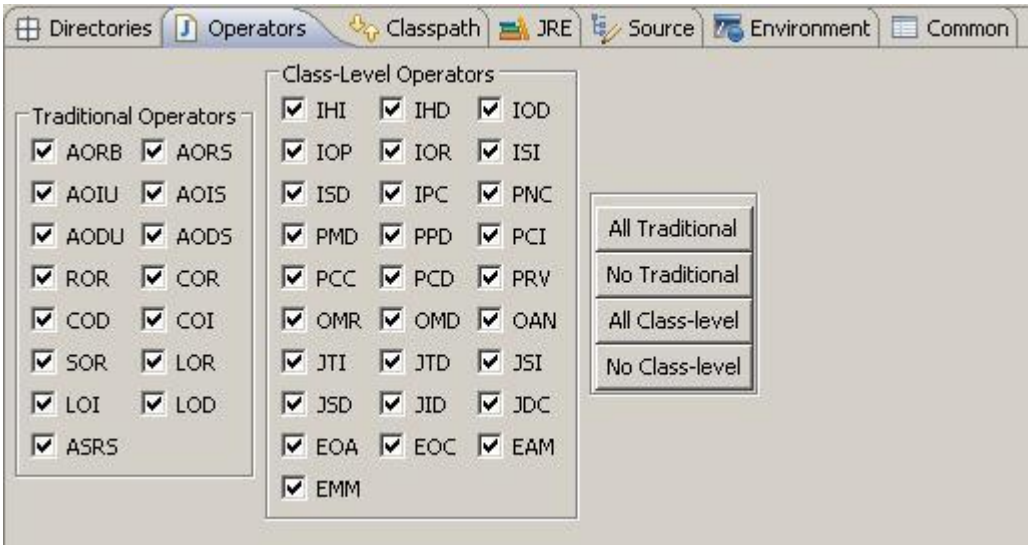

Figura 2.3 Interface gráfica para selecionar operadores de mutação em MuClipse [41]

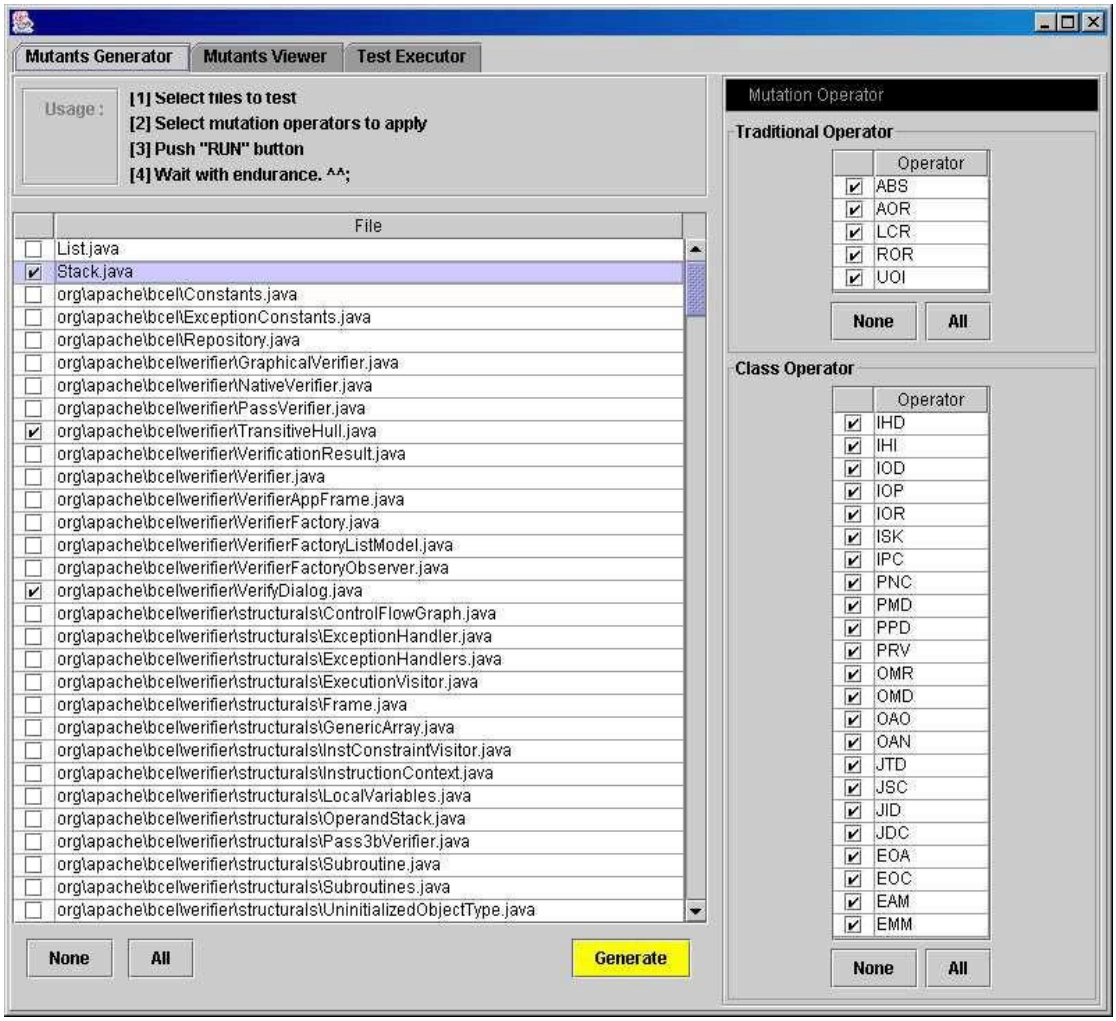

Figura 2.2 Interface gráfica para geração de mutantes em MuJava [19]

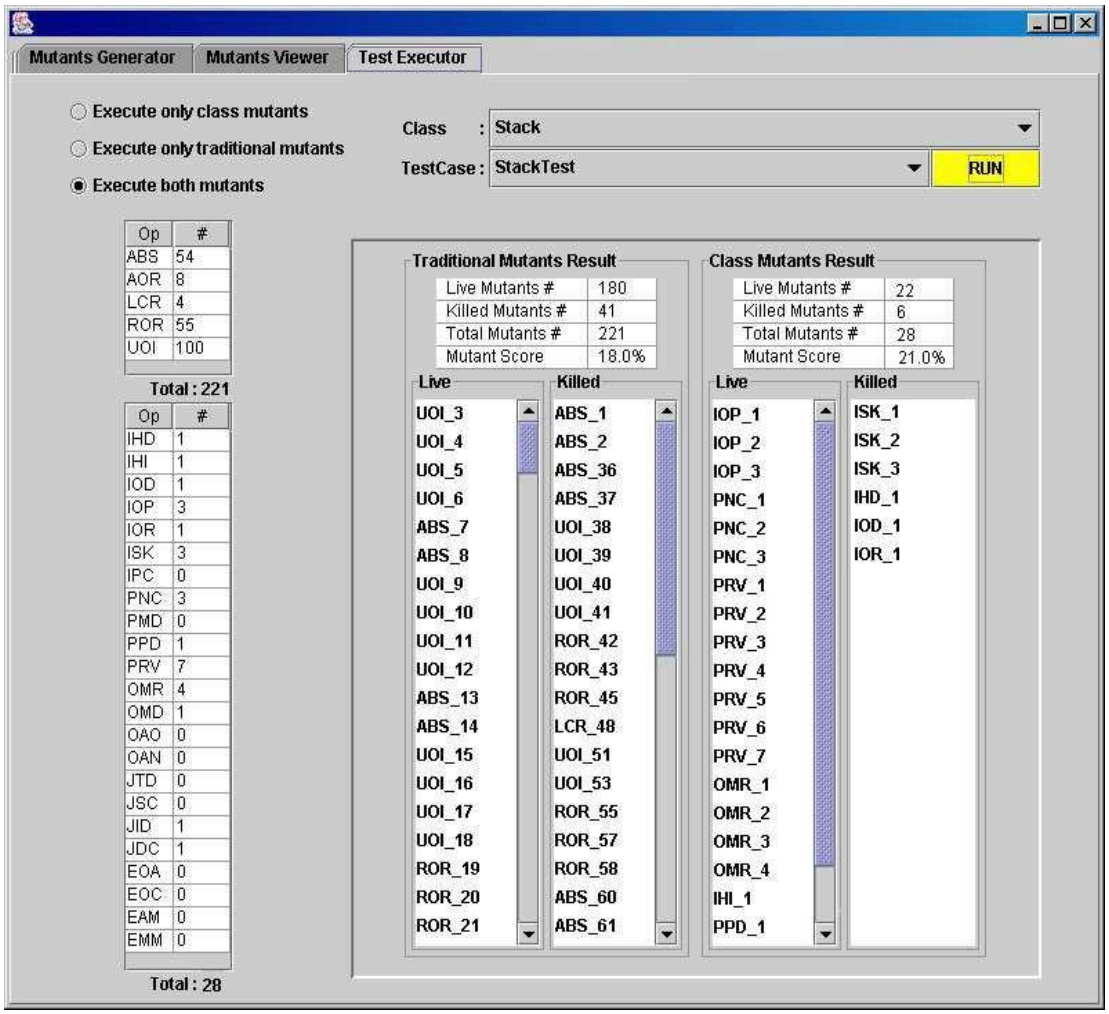

Figura 2.4 Interface gráfica para análise dos mutantes em MuJava [19]

Capítulo 3

Testes de Mutação

Este capítulo apresenta os conceitos utilizados no processo de testes; a descrição do processo de testes de mutação, como pode ser aplicada em programas OO e os tipos de faltas que podem ser introduzidos. Apresenta ainda uma descrição dos operadores de mutação OO e o tipo de faltas que cobre.

3.1 Conceitos

A compreensão e diferenciação de alguns conceitos utilizados no processo de teste são fundamentais para executar corretamente esta atividade. Com base na norma IEEE 610.12, apresentamos a definição e exemplificação dos termos falha, erro e falta no contexto da Engenharia de *Software*. Falta é um passo, processo ou a definição de dados incorretos, por exemplo, o uso de uma instrução ou comando de forma incorreta. Erro é a consequência de uma falta no estado do sistema a diferença entre o valor obtido e o valor esperado ou seja, um estado intermediário incorreto ou resultado inesperado na execução do programa. Falha é a produção de uma saída incorreta em relação à especificação. Uma falha pode ser causada por diversos erros e alguns erros podem não originar uma falha, sendo esta, causada apenas quando o erro é propagado até ao dispositivo de saída, podendo assim ser visualizada pelo utilizador.

No processo de desenvolvimento de *software*, os testes podem ser utilizados para avaliar os requisitos, a especificação, os elementos de desenho, ou o código fonte. De acordo com os autores Ammann e Offutt [8], pode-se distinguir os seguintes níveis de teste: teste de aceitação, teste de sistema, teste de integração, teste de módulo e teste de unitários. O teste de aceitação permite verificar se o produto está de acordo com a especificação dos requisitos. Teste de sistema é realizado após a integração de todas as partes de um sistema com o objetivo de validar se o seu funcionamento está de acordo com o que foi especificado. Após a integração das unidades de um sistema são realizados os testes de integração, sobreposição ou conflito de funcionalidades e tratamento de erros incorreto. São exemplos de falhas que podem surgir com a

integração de unidades de um sistema. O teste unitário permite avaliar a menor unidade de *software*, o módulo; o objetivo nesta fase consiste na identificação de erros de lógica e implementação. Segundo a norma IEEE 620.12 uma unidade é uma componente de *software* que não pode ser subdividida. O teste de módulo permite avaliar o *software* em relação ao projeto detalhado.

Um dos objetivos da atividade de testes é a identificação do maior número possível de erros, torna-se importante avaliar a qualidade e adequação dos testes escritos. Segundo Pressman [1], um bom caso de teste é aquele que tem grande probabilidade de revelar um erro ainda não encontrado. Este autor afirma ainda que a utilização de bons casos de testes permite que os programadores consigam, com menores recursos (tempo, recurso computacionais, esforço Humano), encontrar sistematicamente diferentes classes de erros.

A técnica de teste de mutação pode ser aplicada para testar tanto a especificação de um programa (Mutação de Especificação) [29] como o seu código fonte (Mutação de Programa) [30].

A mutação de especificação pertence a técnica de testes baseados em grafos. Inicialmente este tipo de mutação foi proposta como uma técnica de testes baseados em expressões lógicas, e era utilizada nos testes ao nível da implementação e do desenho de um sistema. A mutação de especificação consiste em implantar faltas na especificação do programa [31].

A mutação de programa pertence à categoria de testes baseados em expressões lógicas. Este tipo de mutação consiste em gerar faltas sobre o código fonte de um programa. As mutações podem ser aplicadas tanto em testes de unidade [3] como em testes de integração [32]. Os mutantes gerados nos testes de unidade representam as faltas que podem ser causadas dentro de uma unidade de um *software*. As mutações geradas nos testes de integração, também conhecidas como mutação de interface, representam as faltas de integração causadas pela ligação ou interação entre as unidades do *software*.

Considerando a mutação de programa, um mutante é gerado se existir uma instrução válida, designada de "*ground string*", sobre o qual se pode aplicar um operador de mutação. Uma instrução é válida se pertencer a gramática especificada pela linguagem, e inválida caso contrário. Um operador de mutação representa uma regra que específica variações sintáticas de instruções que são geradas a partir de uma gramática. Assim, o resultado da aplicação de um operador de mutação sobre uma *ground string* é chamado de mutante.

Quantos operadores de mutação devem ser aplicados para a geração de um mutante? Esta é uma questão muito importante, que podemos responder com base na hipótese de efeito de acoplamento [3], que afirma que as faltas geradas devem ser simples, assim, deve ser aplicado apenas um operador de mutação de cada vez para que sejam gerados mutantes de primeira ordem (FOM).

No teste de mutação é utilizado apenas um subconjunto do conjunto das possíveis faltas que podem existir em um programa, cuja versão se aproxime da versão mais correta do programa, na expetativa de que estas faltas serão suficientes para simular todas as faltas que podem ser produzidas no processo de desenvolvimento de um programa. Esta ideia tem como base teórica duas hipóteses: hipótese do programador competente e o efeito de acoplamento.

Quando é gerado um mutante, pretende-se que no conjunto de casos de testes exista pelo menos um capaz de identificar a falta produzida. Caso isto aconteça, podemos dizer que o mutante foi "morto". Assim, diz-se que um teste "matou" um mutante, se dos dados dados de saída obtidos da execução do teste sobre o mutante, forem diferentes dos dados de saída resultante da execução do teste sobre o programa. No entanto, alguns mutantes gerados podem ser equivalentes, ou seja, o resultado da execução do caso de teste sobre um mutante é igual ao resultado obtido da execução do caso de teste sobre o programa. Para estes casos não existe casos de testes que possam "matar" o mutante. A deteção automática de mutantes equivalentes é impossível, pois estes são indecidíveis e representam requisitos de testes inexequíveis. Por isso, é necessário a aplicação de esforço Humano para a sua identificação.

Caso se pretenda verificar com mais rigor se um caso de teste matou um mutante, isto poder ser feito através do modelo falta/falha RIP, que afirma que uma falha é observável quando se verificam as três condições:

1. Atingir: O elemento que sofreu mutação no programa deve ser alcançado;
2. Infetar: Depois de executar do programa que contém a mutação, o seu estado deve ser incorreto;
3. Propagar: O estado incorreto do programa deve propagar e gerar uma saída incorreta.

A Figura 3.1 representa um processo de teste de um programa utilizando a técnica de teste de mutação. Assumindo que os testes de mutação são aplicados a um programa com o suporte de uma ferramenta, e considerando um conjunto de casos de testes, o processo de teste de mutação pode ser descrito da seguinte forma:

1. São selecionados os operadores de mutação para introduzirem pequenas modificações no programa dando origem a novas versões, ou seja, um

conjunto de mutantes. Na fase de geração de mutantes pode ser aplicada alguma das técnicas descritas no Capítulo 2 para a redução do número dos mutantes gerados;

2. A etapa seguinte consiste em executar os casos de teste no programa, se o resultado obtido for diferente do valor especificado, então foi detetada uma falha no programa, senão é executado os casos de testes sobre os mutantes. Concluída a execução dos casos de testes, é comparado os resultados obtidos para identificar os mutantes que foram "mortos".

3. Terminada a fase de execução de teste, é identificado o conjunto dos mutantes "vivos" e verificados se dentro deste conjunto existem mutantes equivalentes, e com esta informação calcular a pontuação da adequação do conjunto de casos de teste.

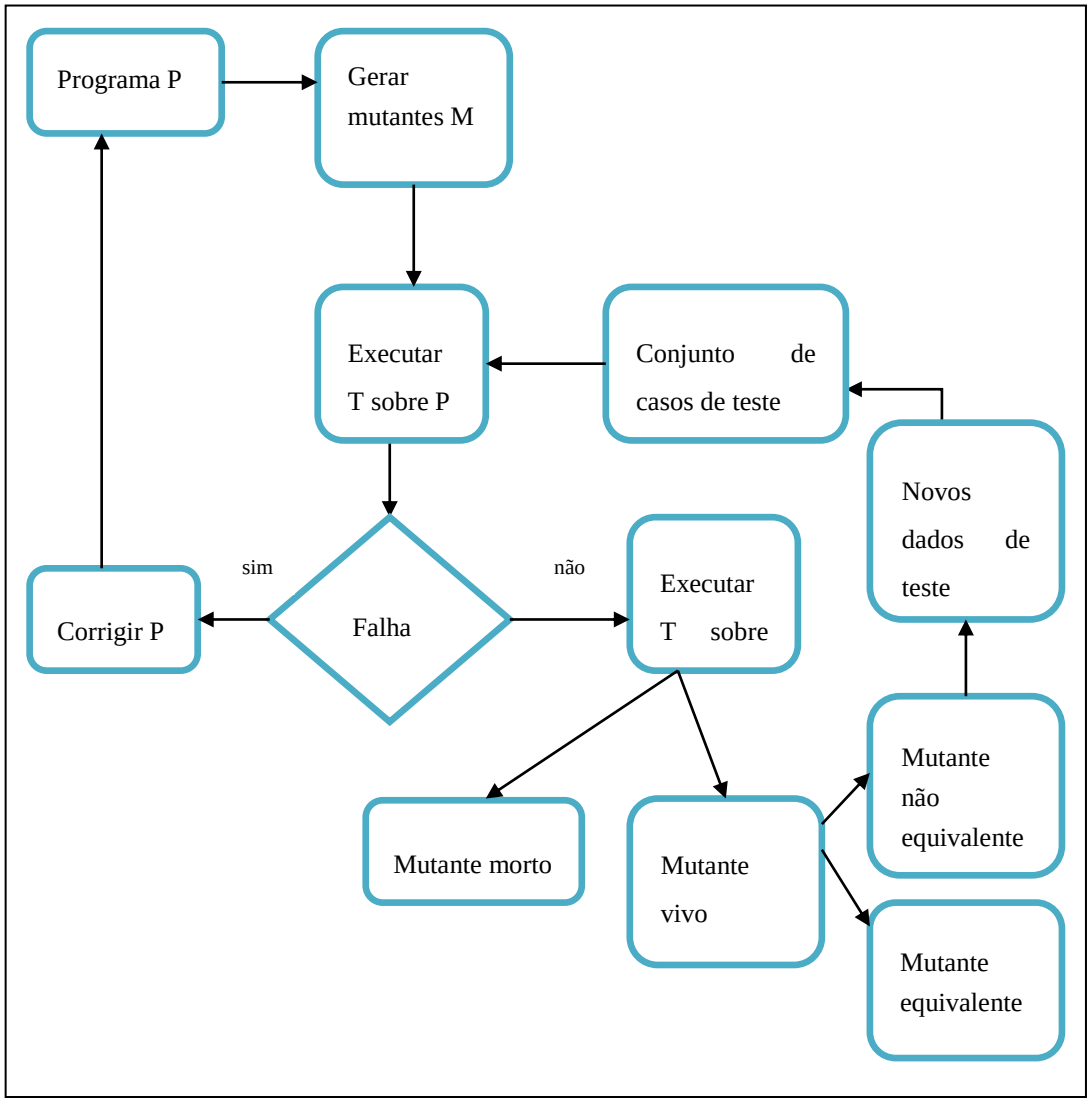

Figura 3.1 Processo de teste de mutação

A análise de mutantes é a etapa dos testes de mutação que requer maior intervenção Humana. O nível de confiança da adequação dos casos de testes é obtido na análise de mutantes, a partir da pontuação da mutação, que relaciona o número de mutantes mortos com o número de mutantes não equivalentes gerados. Dado um programa P e um conjunto de casos de teste T, o pontuação da mutação MS(P,T) é calculado da seguinte forma:

$$MS(P,T) = DM(P,T) / (M(P) - EM(P))$$

Sendo que:

- **DM(P,T)**: quantidade de mutantes mortos pelos casos de teste em T.
- **M(P)**: quantidade total de mutantes gerados.
- **EM(P)**: número de mutantes equivalentes a P.

O valor obtido na pontuação da mutação varia entre 0 e 1, se o resultado obtido estiver mais próximo do valor 1, significa que o conjunto de casos de testes T tem um grau elevado de eficiência para testar o programa P, pois este foi capaz de conseguir identificar um número considerável de mutantes gerados.

3.2 Testes de mutação para programas OO

O procedimento da atividade de teste é determinado pelo paradigma utilizado no desenvolvimento de um determinado programa, um exemplo são os requisitos definidos pelas técnicas e critérios que são influenciados pelo paradigma utilizado. Assim, apesar de inicialmente as técnicas de testes serem propostas para o paradigma procedimental [13, 16], foi necessário haver adaptação para que pudessem ser aplicadas ao paradigma OO. Este apresenta uma nova perspetiva de como pode ser decomposto um determinado problema e novas formas de modelar e implementar soluções para a resolução do problema. Assim, a realização de teste em *software* OO apresenta novos desafios a Engenharia de *Software*, relacionadas com as características do paradigma OO, como por exemplo encapsulamento, herança, polimorfismo.

Deste modo, a classificação dos níveis de testes definida para os testes tradicionais, sofreu uma alteração para satisfazer as necessidades dos testes OO. Os autores Harrold e Rothermel [33], Gallagher e Offutt [34] propuseram quatro categorias de testes OO: intra-método, inter-método, intra-classe e inter-classe. Considerando que a menor unidade de um programa OO é a uma classe, podemos dizer que os testes intra-método, inter-método, intra-classe são variações de testes de unidade, enquanto o teste inter-classe é uma variação de teste de integração. Num teste intra-método, cada método é testado individualmente com o objetivo de identificar falhas na sua implementação.

Vários métodos da mesma classe são testados em conjunto nos testes de inter-método. Os testes de intra-classe são construídos para testar uma classe. No teste de inter-classe, duas ou mais classes são testadas em conjunto para procurar falhas na forma como elas são integradas.

Nesta secção será apresentada uma breve descrição dos outros operadores de mutação tradicionais que também podem ser aplicados em linguagens OO, os tipos de faltas dos programas OO e os operadores de mutação que permitem causar faltas a nível de inter-class e inter-método em programas Java.

3.2.1 Faltas específicas em programas OO

Para explicar o tipo de problemas que podem ocorrer em programas OO é necessário entender os novos conceitos introduzidos pelo paradigma OO como herança, encapsulamento e polimorfismo. Pois, embora estes conceitos proporcionem a redução entre a distância da descrição de um problema no mundo real para um modelo abstrato construído para o mundo computacional, permitindo assim a definição da solução de forma mais rápida, também introduzem novos erros, por exemplo, problemas relacionados com o controle de acesso. Alguns trabalhos desenvolvidos permitiram a identificação dos vários tipos de falhas específicas das linguagens OO [35], que podem ser produzidas pela má aplicação destes conceitos.

No paradigma OO, o objeto é um elemento que representa no domínio da solução uma entidade (abstrata ou concreta) do domínio de um problema sob análise. Um objeto contém um estado e um comportamento, sendo que o seu estado é definido pelas suas propriedades, e o seu comportamento é representado pelo conjunto de operações que o objeto fornece. Outro conceito importante que deve ser mencionado é o de classe, uma classe no paradigma OO é uma abstração que permite descrever um conjunto de objetos com as mesmas propriedades, o mesmo comportamento e as ligações com outros objetos.

Um mecanismo de abstração que permite controlar o acesso dos atributos e métodos de um objeto é o encapsulamento. Este mecanismo do paradigma OO promove a ocultação de informação e permite de forma eficiente proteger os dados manipulados, garantir maior flexibilidade. No entanto, se este conceito não for utilizado corretamente pode originar a definição incorreta dos níveis de acessos das variáveis e dos métodos.

A herança é uma característica única do paradigma OO, permitido que características comuns a várias classes sejam agrupadas a uma única classe designada como superclasse. A herança promove a reutilização de código, através da agregação de elementos comuns a várias classes (subclasses) em uma única classe (superclasse). Deste modo, é possível diminuir o esforço de implementação, tornar mais fácil a

manutenção do código e minimizar a possibilidade de erros. No entanto, segundo Binder [36], este mecanismo enfraquece o encapsulamento e se utilizado de forma incorreta pode aumentar a complexidade do programa, diminuindo a sua compreensão e aumentando a possibilidade de ocorrência de erros. Existem outros conceitos associados ao conceito de herança na linguagem Java: redefinição de método, ocultação de atributos e construtor da classe. A redefinição de método permite que uma subclasse possa definir uma implementação mais específica de um método herdado da superclasse. A ocultação de atributos consiste na definição de uma variável na subclasse com o mesmo tipo e o mesmo nome de uma variável definida na superclasse, "escondendo" a variável definida na superclasse. No conceito de herança a subclasse herda da superclasse os seus atributos e métodos. No entanto, existe um método especial denominado de construtor que não é herdado pela subclasse. No contexto deste trabalho não será considerada a herança múltipla, que é a capacidade de herdar características de duas ou mais superclasses, uma vez que a linguagem Java não permite este mecanismo.

A herança entre classes estabelece uma relação entre os objetos designada de polimorfismo. Esta característica permite, que um objeto possa assumir várias formas, isto é possível através da manipulação de instâncias de classes que são subclasses de uma mesma classe de forma unificada. Assim, a concretização do polimorfismo é a ligação dinâmica, isto é, durante o tempo de execução (e não em tempo de compilação) é determinada o método que deve ser executado de acordo com o tipo de objeto. A linguagem Java suporta a característica de método e atributo polimórfico. No caso de um atributo polimórfico a instância de um objeto declarado pode ser igual ao tipo declarado ou pode ser igual ao tipo de qualquer uma das subclasses do tipo declarado. No caso de um método polimórfico é semelhante ao atributo polimórfico, mas aplicado ao tipo do argumento do método. No entanto, a utilização incorreta do polimorfismo pode dar origem a geração de falhas.

A sobrecarga de métodos acorre, quando em uma determinada classe são definidos dois métodos com o mesmo nome mas com a assinatura (número de argumentos) diferente. Um dos problemas que pode ocorrer com a utilização da sobrecarga de métodos é invocar o método errado.

O uso incorreto de algumas palavras reservadas como "*super*", "*this*", "*static*", a inicialização incorreta de variáveis, a implementação incorreta da sobrecarga de métodos são alguns problemas associados a incompreensão/aplicação incorreta de conceitos apresentados anteriormente em programas Java, representam fontes de possíveis falhas que podem ser utilizadas no desenho de operadores de mutação.

3.2.2 Operadores de mutação tradicionais

Para além dos operadores de mutação mencionados na secção 3.2.2 , existem outros operadores designados de operadores de mutação tradicionais. Estes permitem a modificação de operadores unários e binários, como por exemplo, operadores lógicos, operadores aritméticos. Inicialmente estes operadores de mutação foram definidos para a linguagem Fortran 77 [13], no entanto estes foram adaptados para serem aplicados em programas Java [8]. A Tabela 3.1 apresenta os operadores de mutação tradicionais adaptados para a linguagem Java.

Abreviação	Descrição
ABS	Inserção do valor absoluto
AOR	Substituição do operador aritmético
ROR	Substituição do operador relacional
COR	Substituição do operador condicional
SOR	Substituição do operador *shift*
LOR	Substituição do operador lógico
ASR	Substituição do operador de atribuição
UOI	Inserção do operador unário
UOD	Eliminar o operador unário
SVR	Substituição de variável escalar
BSR	Substituição por instrução “bomba”

Tabela 3.1 Operadores de mutação tradicionais

3.2.3 Operadores de mutação para a linguagem Java

Para os testes de mutação, a identificação do tipo de faltas causadas pelos programadores durante o processo de implementação de um sistema é muito importante, isto porque, é com base nesta informação que são desenhados os operadores de mutação. Assim, a definição dos operadores de mutação é feita com o intuito de satisfazer dois objetivos. O primeiro consiste na reprodução de erros cometidos durante o processo de desenvolvimento de um sistema, de modo a que os testes criados sejam

capazes de identificar estes erros. O segundo objetivo consiste em garantir que os testes são desenhados para testar com eficácia um determinado sistema.

Algumas faltas são comuns dentro das linguagens de programação de um determinado paradigma. No entanto, cada linguagem de programação tem características próprias, o que implica a existência de faltas específicas a cada linguagem de programação. Nesta secção serão apresentados os operadores de mutação para a linguagem Java.

Ma, Kwon e Offutt [18] definiram seis grupos de classes de operadores de mutação, com base nas características da linguagem Java:

1. Controle de acesso;
2. Herança;
3. Polimorfismo;
4. Sobrecarga;
5. Características específicas de Java;
6. Erros comuns de programação.

Nos primeiros quatro grupos estão representadas as faltas associadas as características comuns a todas as linguagens OO. No quinto grupo estão concretizadas as faltas específicas para a linguagem Java, como por exemplo, a definição de atributos da classe utilizando o modificador "*this*". O último grupo inclui operadores de mutação baseados nos erros comuns da programação imperativa.

Em seguida será apresentada uma breve descrição de cada um dos operadores de mutação dos grupos enumerados anteriormente. Em cada uma das descrições, será apresentado um exemplo com o objetivo de ilustrar a aplicação dos operadores de mutação. O símbolo Δ assinala as mutações aplicadas nos exemplos apresentados.

Controle de acesso

- **AMC – Alteração do modificador de acesso:** consiste em alterar o nível de acesso dos atributos e dos métodos de uma classe.

Código original	Mutante AMC

public class A{
  private String s;
  …
}
	public class A{
	  Δ **public String** s;
	  …
	}

Herança

- **IHD – Apagar o encobrimento de um atributo:** este operador apaga a declaração de um atributo definido para esconder o atributo da superclasse. Eliminando a encobrimento da definição de um atributo, implica que todas

Código original

```
public class A{
    String s;
}

public class B extends A{
    String s;
    void m(){
      s = "Hello!";
    }
}
```

Mutante IHD

```
public class A{
   String s;
}

public class B extends A {
Δ  // String s;
  void m(){
     s = "Hello!";
    }
}
```

as operações serão feitas sobre o atributo da superclasse.

- **IHI – Inserir atributo que encobre outro:** este operador insere na subclasse um atributo com o mesmo nome e tipo do atributo existente na superclasse, "escondendo" o atributo da superclasse. Isto implica que as operações serão feitas sobre o novo atributo definido na subclasse.

Código original

```
public class A{
    String s;
}

public class B extends A{

    void m(){
      s = "Hello!";
    }
}
```

Mutante IHI

```
public class A{
   String s;
}

public class B extends A {
Δ String s;
   void m(){
     s = "Hello!";
    }
}
```

- **IOD – Apagar a redefinição de método:** este operador apaga a redefinição de um método declarado na subclasse. Deste modo, será invocado o método da superclasse.

Código original

```
public class B extends A{
…
    int count(int a){…}
}
```

Mutante IOD

```
public class extends A{
…
 Δ // int count(int a){…}
}
```

- **IOP – Alterar a posição da chamada do método redefinido:** na redefinição de um método é por vezes necessário invocar o método da

superclasse. Este operador altera a posição de chamada do método redefinido para a primeira ou a última instrução do método da subclasse.

Código original

```
public class A{
    …
    void count(){
      i = 10;
    }
}

public class A{
    …
    void count(){
      super.count();
      i = 5;
    }
}
```

Mutante IOP

```
public class A{
   …
   void count(){
        i = 10;
    }
}

public class A{
   …
   void count(){
        Δ i = 5;
        Δ super.count();
    }
}
```

- **IOR – Alterar o nome do método:** Este operador altera o nome do método da superclasse que foi redefinido.

Código original

```
public class A{
    …
    void m(){… f(); …}
    void f(){…}

}

public class B extends A{
    …
    void f(){…}
}
```

Mutante IOR

```
public class A{
   …
Δ void m(){… f'(); …}
Δ void f'(){…}

}

public B extends A{
   …
   void f(){…}
}
```

- **ISK – Apagar a palavra reservada *super*:** este operador apaga a palavra *super* que permite referenciar os atributos ou métodos da superclasse redefinidos. Isto permite verificar se os atributos e os métodos redefinidos são utilizados de forma adequada.

Código original

```
public class B extends A{
    int num;
    int count(){
    …
    return i*super.num;
    }
}
```

Mutante ISK

```
public class extends A{
    int num;
    int count(){
    …
  Δ  return i*num;
    }
}
```

- **IPC – Eliminar a invocação explícita do construtor da superclasse:** Este operador de mutação elimina a chamada explícita do construtor da superclasse, para que seja invocado o construtor padrão da superclasse.

Código original

```
public class B extends A{
    B(){}
    B(int a){
      super(a);
    …
    }
}
```

Mutante IPC

```
public class B extends A{
    …
    B(int a){
     Δ //super(a);
     …
    }
}
```

Polimorfismo

O polimorfismo permite que o tipo real da referência de um determinado objeto possa ser diferente do tipo declarado, ou seja, a instância de um objeto pode ser de uma subclasse do tipo utilizado na declaração do objeto. Os operadores testam o correto funcionamento de um determinado programa com todas as possíveis ligações de um tipo polimórfico declarado.

- **PNC – Chamada a *new* com o tipo de uma subclasse:** este operador altera o tipo de instância referido por um objeto. Deste modo, o tipo da instância do objeto é diferente do tipo declarado. No exemplo apresentado a

Código original

```
A a;
a = new A();
```

Mutante PNC

```
A a;
Δ a = new B();
```

classe B é uma subclasse de A.

- **PMD – Declaração de variável com o tipo da superclasse:** é alterado o tipo da declaração de um objeto para o tipo da superclasse. No exemplo apresentado a classe B é uma subclasse de A.

Código original

```
B b;
b = new B();
```

Mutante PMD

```
Δ A b;
b = new B();
```

- **PPD – Declaração do parâmetro com o tipo da subclasse:** o operador PPD é idêntico ao operador PMD, exceto que a alteração é feita sobre o parâmetro de um método. No exemplo apresentado a classe B é uma subclasse de A.

Código original	Mutante PPD
boolean passed(B b)	Δ **boolean** passed(A a)

- **PRV – Atribuição de uma referência de um tipo compatível:** este

Código original	Mutante PRV
Object obj; Integer i = new Integer(2); String s = “Hello”; obj = s;	Object obj; Integer i = new Integer(2); String s = “Hello”; Δ obj = i;

operador altera o operando de atribuição de um objeto para o valor da referência de uma subclasse.

Sobrecarga

A sobrecarga de método permite a definição de métodos com o mesmo nome, mas com assinatura diferente. Assim, é importante verificar se o método chamado é o pretendido.

- **OMR – Alteração do conteúdo do método de sobrecarga:** este ooperador substitui o corpo do método pela invocação de outro método com o mesmo nome, utilizando a palavra-chave *this*. Este operador permite verificar se os métodos de sobrecarga são chamados de forma adequada.

Código original	Mutante OMR

class A{
…
void add(**int** i){…}
void add(**int** i, **int** j){…}
 	
}
	class A{
	…
	void add(**int** i){…}
	void add(**int** i, **int** j){
	Δ this.add(i);
	}

- **OMD – Apagar o método de sobrecarga:** é apagada a declaração de um método de sobrecarga, um de cada vez. Este operador permite garantir que todos os métodos de sobrecarga são chamados pelo menos uma vez, evitando assim a definição de métodos de sobrecarga que não estão a ser utilizados.

Código original	Mutante OMD

class B **extends** A{
…
void add(**int** i){…}
void add(**float** f){…}
}
	class B **extends** A{
	…
	Δ //**void** add(**int** i){…}
	void add(**float** f){…}
	}

- **OAO – Alteração da ordem dos argumentos:** é modificado a ordem que os argumentos são passados num método. Isto só é possível se houver um método de sobrecarga que aceita a nova lista de argumentos.

Código original	Mutante OAO
s.make(0.5, 3);	Δ s.make(3, 0.5);

- **OAN – Alteração do número de argumentos:** é alterado o número de argumentos invocado em um determinado método. Isto só é possível se houver um método de sobrecarga que aceita a nova lista de argumentos.

Código original	Mutante OAN
s.make(0.5, 3);	Δ s.make(3); Δ s.make(0.5);

Características específicas de Java

- **JTD – Eliminar a palavra-chave *this:*** este operador, apagada a ocorrência da palavra *this*. Os atributos da instância de um objeto podem ser referidas no corpo de um método pelo seu nome, no entanto, pode acontecer que este atributo esteja "escondido" porque o método tem um parâmetro com o mesmo tipo e nome que o atributo. Assim é necessário utilizar a palavra *this* para fazer referência o atributo da instância do objeto.

Código original	Mutante JTD

class B{
private int length;
void setLength(**int** length){
this.length = length;
}
}
	class B{
	private int length;
	void setLength(**int** length){
	Δ length = length;
	}
	}

- **JSC – Alteração do modificador *static*:** operador remove o modificador *static* para alterar variáveis de classe para variáveis de instância, ou adiciona o modificador *static* para modificar variáveis de instância para variáveis de classe. Este operador permite avaliar o comportamento do sistema aplicando variáveis de classe e de instâncias.

Código original	Mutante JSC
private int length;	Δ **public static int** length;

- **JID – Eliminação da inicialização do atributo:** os atributos de uma classe podem ser inicializados na sua declaração ou no construtor da classe. Este operador de mutação elimina a inicialização feita na declaração do atributo

de uma classe. Deste modo, os atributos serão inicializados com o valor pré-definido. Isto permite verificar se as inicializações dos atributos de uma classe estão corretas.

Código original	Mutante JID

class B{
  private **int** length = 50;
…
}
	class B{
	  Δ **private int** length;
	…
	}

- **JDC – Utilizar o construtor padrão suportado pelo Java:** é eliminado o construtor padrão implementado em uma determinada classe. Deste modo, será utilizado o construtor padrão suportado pela linguagem Java. Este operador de mutação permite verificar se a implementação do construtor padrão está correta.

Código original	Mutante JDC

class B{
  B(){…}
…
}
	class B{
	Δ // B(){…}
	…
	 }

Erros comuns de programação

Os operadores de mutação deste grupo visam modelar as faltas típicas cometidas pelos programadores no desenvolvimento de programas OO. Estas faltas estão relacionadas com a manipulação de tipo por referências e os com métodos utilizados para obter e alterar os atributos dos objetos.

- **EOA – Substituição da referência pela atribuição do conteúdo:** este operador de mutação substitui a atribuição de uma referência a uma variável, pela cópia do objeto, utilizando o método *clone*() da linguagem Java.

Código original	Mutante EAO

List list1, list2;
list1 = **new** List();
list2 = list1;
	List list1, list2;
	list1 = **new** List();
	Δ list2 = list1.clone();

- **EOC – Substituição da comparação de referências pela comparação de conteúdo:** alguns programadores podem trocar a comparação das referências de objetos pela comparação dos seus conteúdos. Este operador de mutação substitui a comparação entre referências pela comparação do conteúdo dos objetos através do método *equals*() da linguagem Java.

Código original	Mutante EOC
Number n1 = **new** Number(1); Number n2 = **new** Number(5); **boolean** f = (n1 == n2);	Number n1 = **new** Number(1); Number n2 = **new** Number(5); Δ **boolean** f = (n1.equals(n2));

- **EAM – Alteração do método de acesso:** uma boa prática de programação OO é a definição de acesso privado dos atributos de uma classe, ou seja, estes são acedidos diretamente apenas por instâncias da classe a qual pertencem. Deste modo, são definidos métodos (públicos), que permitem o acesso e a modificação dos atributos de uma determinada classe. Estes métodos são conhecidos como "*get*" (aceder) e "*set*" (modificar), e por convenção o nome do método é formado pelo "*get*" ou "*set*" sucedido do nome da variável à que estão relacionados. No entanto, uma determinada classe, com muitas variáveis, cujos nomes são muito semelhantes, o programador pode causar alguns erros na definição dos métodos de acesso e modificação. Este operador de mutação altera o nome do método que permite aceder um determinado atributo privado de uma classe.

Código original	Mutante EAM
number.getY();	Δ number.getX();

- **EMM – Alteração do método de modificação:** o operador de mutação EMM altera o nome do método que permite modificar um determinado atributo privado de uma classe.

Código original	Mutante EMM
number.setY(1);	Δ number.setX(1);

Neste capítulo foram apresentados os conceitos relacionados com os testes de mutação, foi também ilustrado o processo de aplicação dos testes de mutação. Sendo o desenho dos operadores de mutação, um ponto importante nos testes de mutação e sabendo que a sua definição depende das características da linguagem de programação utilizada, foram também referidos os tipos de faltas que podem ocorrer em programas OO devido as suas caraterísticas, atribuindo ênfase a linguagem Java. Por fim, foram apresentados os operadores de mutação definidos para a linguagem Java.

Capítulo 4

Trabalho Realizado

Este trabalho foi desenvolvido no contexto do Projeto de Engenharia Informática, para o Mestrado em Engenharia Informática na Faculdade de Ciências da Universidade de Lisboa, no grupo de investigação do Departamento de Informática, LaSIGE – Laboratório de Sistemas Informáticos de Grande Escala.

O presente capítulo descreve a concretização da ferramenta proposta para auxiliar o ensino da técnica de testes de mutação. Assim, será apresentada a análise do problema que consiste na definição dos requisitos funcionais e dos requisitos não funcionais, e o desenho da solução. A última secção deste capítulo descreve a implementação da solução, mostrando a estrutura interna do *plug-in*, as dependências com outros elementos externos, detalhes de decisões de desenho e outros aspetos considerados relevantes.

4.1 Análise do problema e desenho da solução

4.1.1 Requisitos funcionais

Os requisitos funcionais representam a especificação de funções ou tarefas que o sistema deve fornecer.

Com o objetivo de atender as necessidades dos utilizadores que pretendem aplicar a técnica de testes de mutação, foram disponibilizadas as seguintes funcionalidades no *plug-in* PESTTMuTest:

- Selecionar os operadores de mutação dentro de um conjunto de operadores de mutação disponíveis;
- Obter a lista das *ground string* que podem ser aplicadas as mutações, de acordo com os operadores de mutação selecionados;

- Obter os operadores de mutação que podem ser aplicados a uma determinada g*round string* (este lista de operadores de mutação está condicionada aos operadores de mutação selecionados pelo utilizador);
- Obter os mutantes resultantes da aplicação de um determinado operador de mutação, de acordo com a g*round string* selecionada;
- Executar o conjunto de testes sobre todos os mutantes gerados, com base nos operadores de mutação selecionados;
- Executar conjunto de testes sobre um mutante selecionado de forma aleatória num conjunto de mutantes gerados. Isto é feito para cada um dos operadores de mutação selecionados;
- Calcular a pontuação dos testes de mutação.

4.1.2 Requisitos não funcionais

Nem todas as características de um sistema podem ser definidas em termos de funcionalidades. Os requisitos não funcionais são as características mínimas que um sistema desenvolvido com qualidade deve satisfazer, e definem as condições e restrições de um sistema.

Durante o desenvolvimento de um *software*, deve-se identificar estes requisitos para que sejam considerados ao longo do processo de desenvolvimento, pois a sua aplicação permite demonstrar a qualidade do *software*.

Foram considerados alguns requisitos não funcionais, em particular no que se refere a extensibilidade, portabilidade, usabilidade e desempenho, no processo de desenvolvimento da aplicação PESTTMuTest.

Uma das principais características que permite determinar, se uma ferramenta é fácil de usar, mnemonizável e resolve eficazmente as tarefas para as quais foi implementada, é a usabilidade. Para permitir que o utilizador sinta-se familiarizado com o novo ambiente de trabalho, e facilite a compreensão da organização dos elementos de interface gráfica do *plug-in* PESTTMuTest, seguiu-se o padrão de organização dos elementos de interface gráfica utilizado pela plataforma Eclipse.

Outros atributos de qualidade beneficiados pelo PESTTMuTest, por ser um *plug-in* da plataforma Eclipse são: a extensibilidade, a portabilidade. A possibilidade de serem adicionadas novas funções ao PESTTMuTest é garantida pelo facto da plataforma Eclipse permitir que sejam adicionas funcionalidades/recursos a um determinado *plug-in*, através do mecanismo de definição de pontos de extensão. No caso do PESTTMuTest, pretende-se que seja possível a integração de novos operadores de

mutação. Uma vez que o Eclipse pode ser utilizado em diferentes sistemas operativos, e sendo o PESTTMuTest um *plug-in* do Eclipse, está garantida a sua portabilidade.

Os recursos de feedback e de prevenção de erros também foram implementados na aplicação, podendo-se numerar os seguintes:

(1) Não é possível iniciar o processo de testes de mutação caso não exista um projeto Java criado/aberto. Assim, o utilizador é informado que para iniciar o processo de testes tem que criar/abrir um projeto Java (ver Figura 4.1).

(2) Se o utilizador pretender dar início ao processo de teste de mutação sem que previamente tenha selecionado algum dos operadores de mutação disponíveis é apresentado uma mensagem informativa (ver Figura 4.2).

(3) Se após o início do processo de testes de mutação o utilizador efetuar alterações num projeto Java, e em seguida selecionar alguma operação disponível no PESTTMuTest (que não seja iniciar o processo de testes de mutação) o utilizador é informado que os dados apresentados podem não corresponder ao que consta atualmente no projeto (ver Figura 4.3).

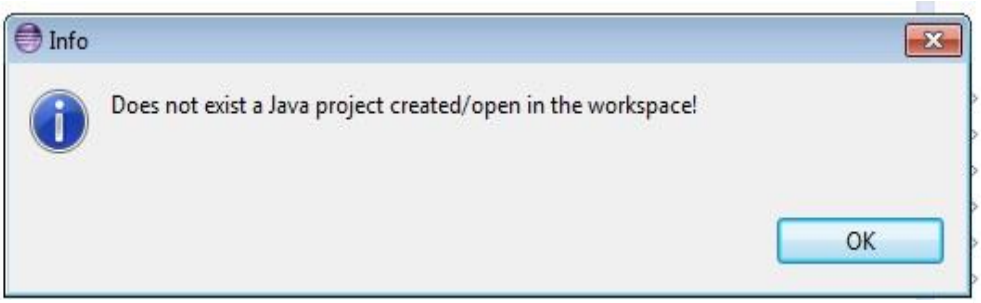

Figura 4.1 Mensagem de informação sobre a inexistência de um projeto Java.

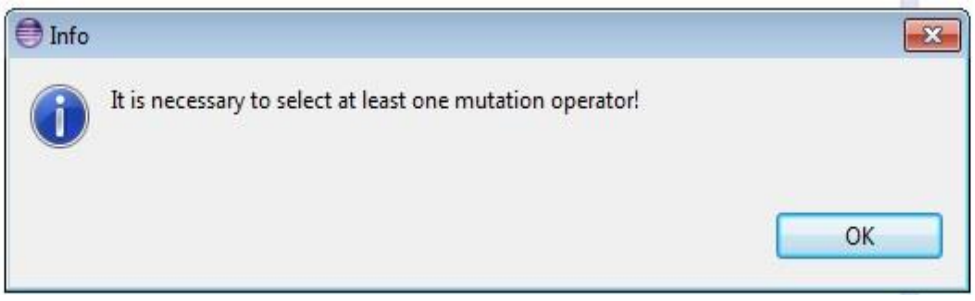

Figura 4.2 Mensagem de informação sobre ausência de operadores de mutação selecionado.

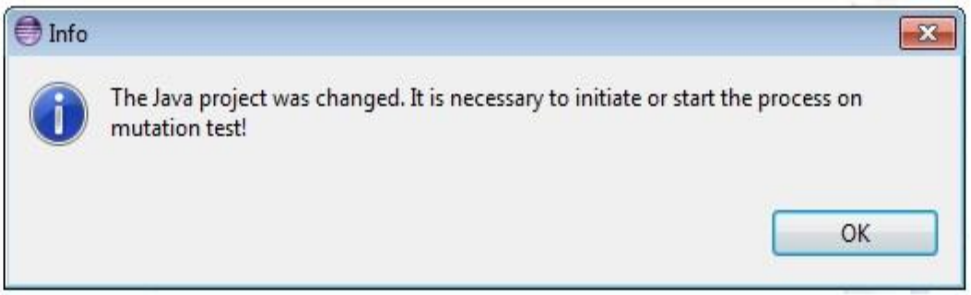

Figura 4.3 Mensagem de informação sobre Alteração do projeto Java.

4.1.3 Modelo de casos de uso

O diagrama de casos de uso permite apresentar uma visão externa do sistema.

Sendo o principal objetivo do PESTTMuTest a concretização de testes de mutação em Java, foram identificados os casos de uso relacionados com cada processo, de modo a determinar as funcionalidades a implementar. Deste modo, foi possível capturar os requisitos funcionais descritos na secção anterior.

O diagrama de caso de uso representa a interação dos atores com as funcionalidades do sistema. A Figura 4.4 ilustra um diagrama que apresenta as funcionalidades do PESTTMuTest e faz uma narração que descreve eventos de atores (utilizador, compilador Java e JUnit) que interagem com a ferramenta.

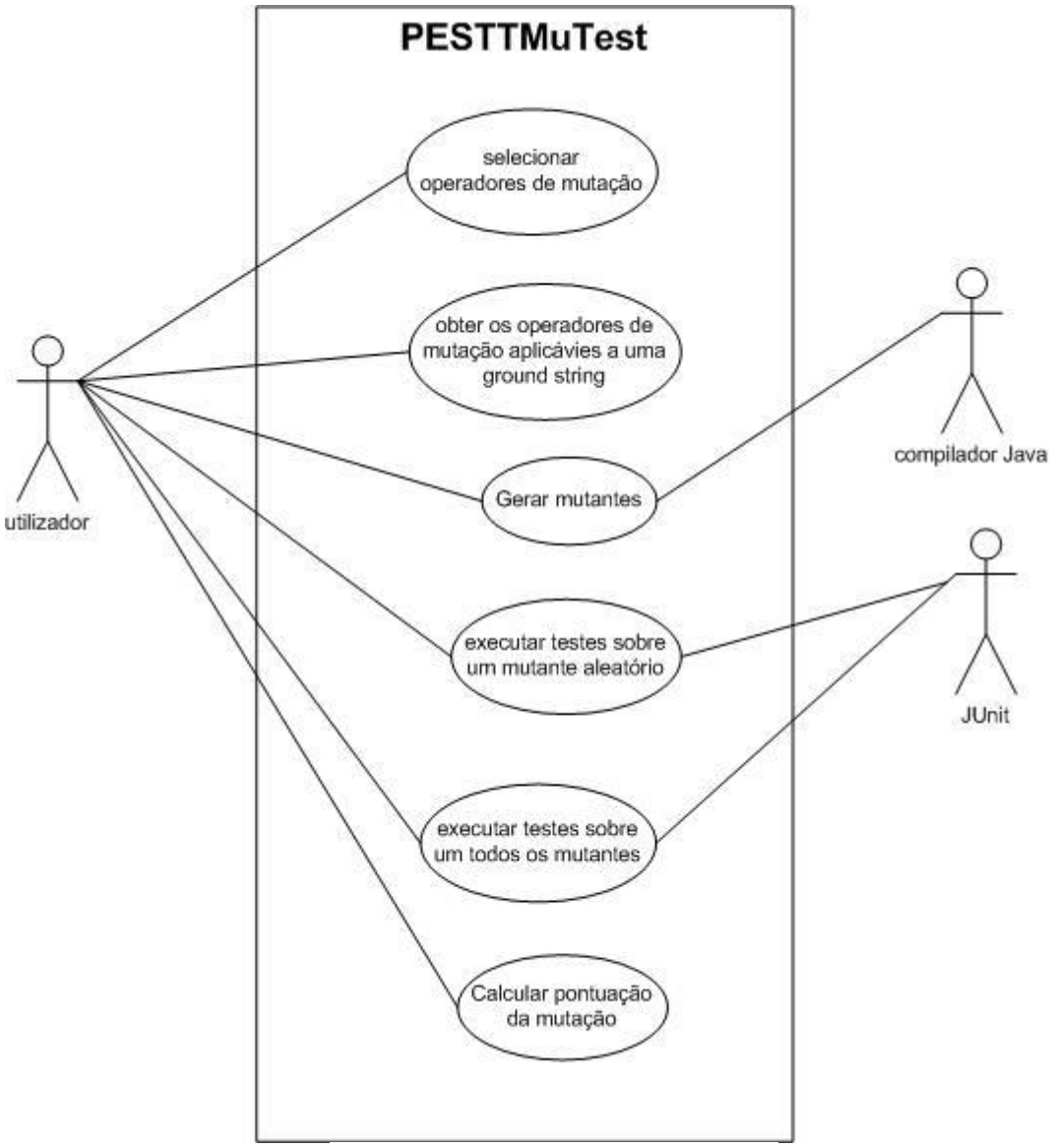

Figura 4.4 Casos de uso

4.1.4 Modelo de domínio

O modelo de domínio é a representação visual das classes conceptuais ou objetos do mundo real relacionados com o domínio do problema.

A elaboração do modelo de domínio permitiu apurar os conceitos associados ao domínio do problema e também as relações existentes entre estes conceitos. Deste modo, foi possível:

- Identificar as classes conceptuais;
- Identificar as relações/associações entre as classes conceptuais.

O modelo de domínio detalhado para o PESTTMuTest é ilustrado na Figura 4.5.

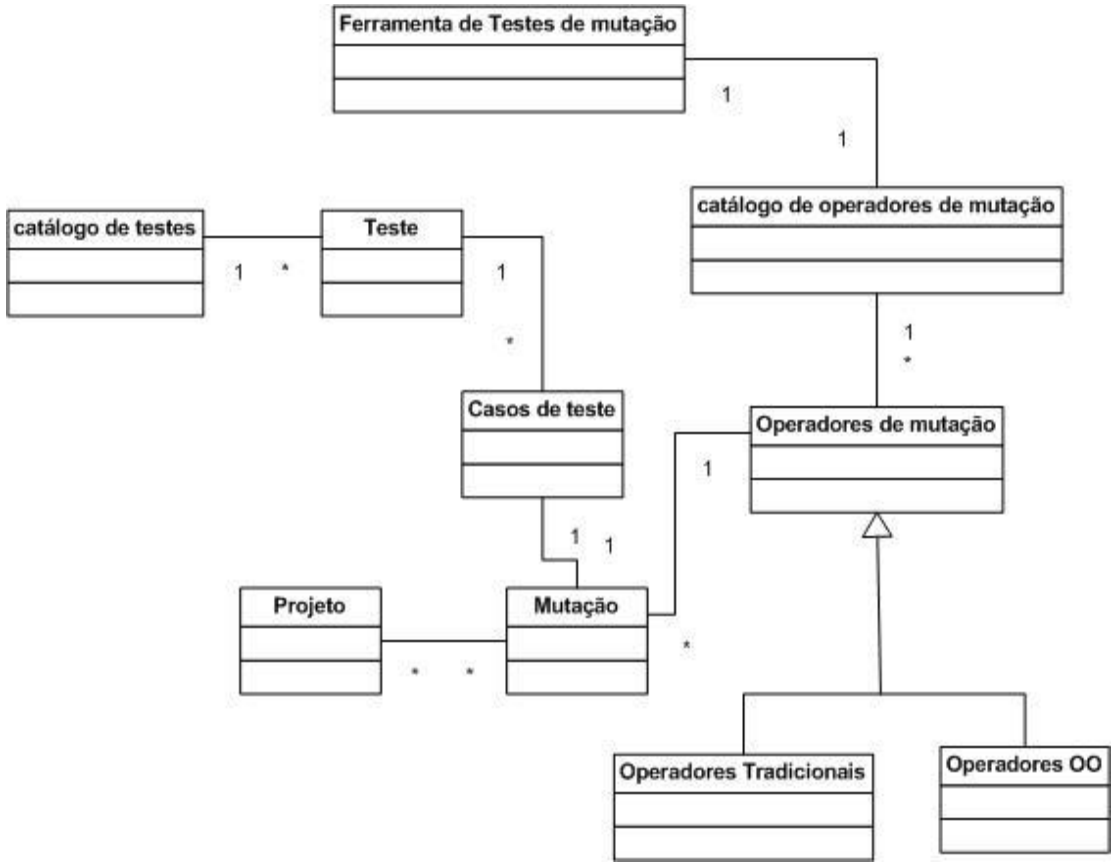

Figura 4.5 Modelo de domínio

4.2 Implementação da solução

Esta secção descreve a implementação do *plug-in* PESTTMuTest. Inicialmente descrito o ambiente de programação utilizado na elaboração da ferramenta. Em seguida, será apresentada uma descrição sobre a arquitetura da plataforma Eclipse e também as dependências do PESTTMuTest com outros *plug-ins*, de modo a permitir uma melhor compreensão da forma como o PESTTMuTest é integrado no Eclipse. Finalmente, será detalhada a organização interna do *plug-in* PESTTMuTest.

4.2.1 Ambiente de programação

O *plug-in* PESTTMuTest foi desenvolvido na plataforma Eclipse versão Kepler. Os *plug-ins* para esta plataforma são escritos na linguagem Java, assim para a implementação do *plug-in* PESTTMuTest foi utilizada a versão 7 do Java. A plataforma Eclipse disponibiliza um ambiente de desenvolvimento de *plug-ins* (PDE). O PDE é um *plug-in* que permite que os utilizadores possam construir ferramentas que se integrem perfeitamente no ambiente Eclipse e deste modo contribuir para o aumento de

funcionalidades disponíveis nesta plataforma. Assim, para os utilizadores que pertentem desenvolver *plug-ins* para o Eclipse, o PDE oferece ferramentas para criar, desenvolver, testar, depurar, compilar e integrar um *plug-in* no Eclipse. Para a concretização do PESTTMuTest foi utilizado o PDE da plataforma Eclipse, a Figura 4.6 mostra o ambiente de desenvolvimento.

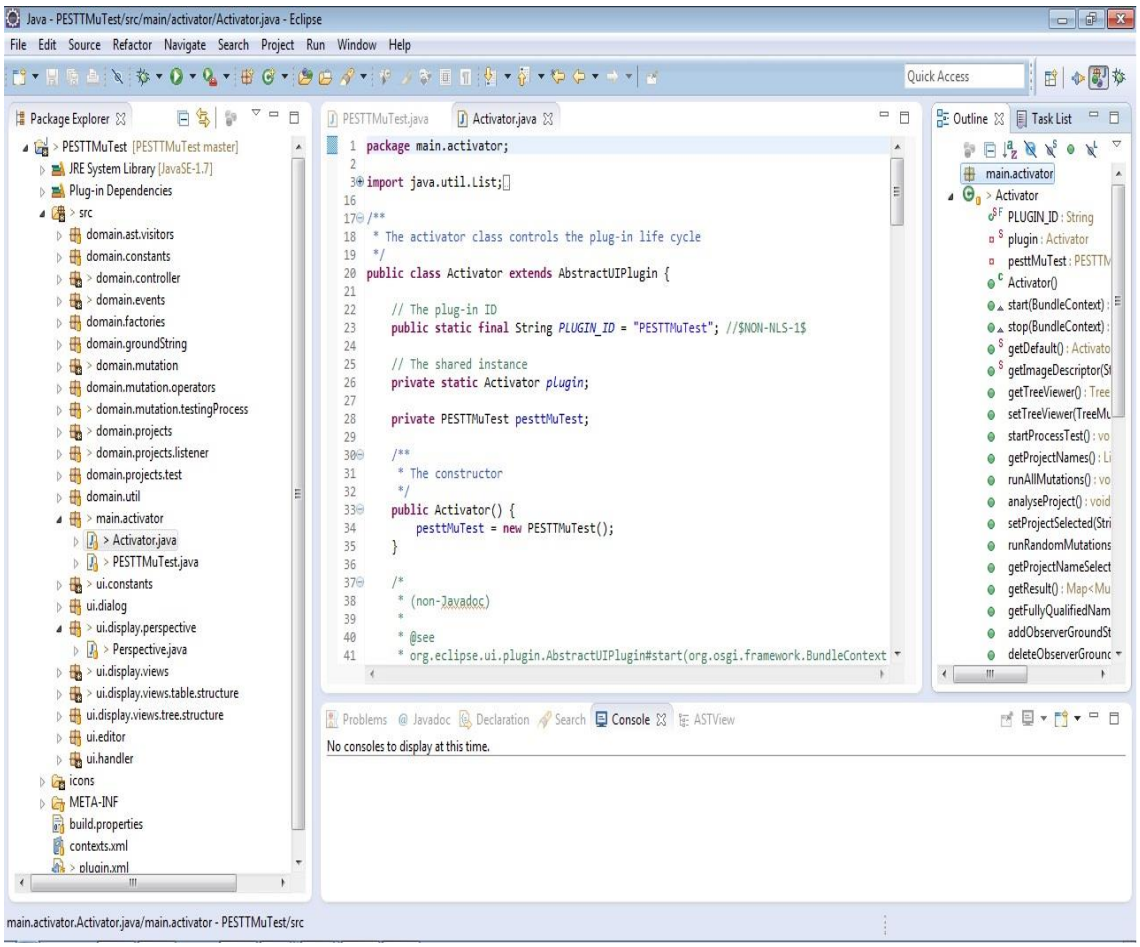

Figura 4.6 Ambiente de desenvolvimento em Eclipse versão Kepler

4.2.2 Arquitetura do Eclipse

O Eclipse é uma plataforma universal, para a integração de ferramentas de desenvolvimento. O Eclipse foi construído com base em uma implementação da especificação *OSGi* [37], designada como Equinox [38]. A especificação *OSGi* define um modelo de componentes (módulos) e de serviços para gerir o ciclo de vida de *software*. A arquitetura definida por esta especificação tem como objetivo reduzir a complexidade de um *software.* Em *OSGi* um componente de um *software* é designado de pacote. A definição de pacotes permite a reutilização de componentes de outras aplicações, o controle de forma eficaz da API e das dependências de um *plug-in.* Isto é possível porque, um pacote define de forma explícita as suas dependências em relação a outros componentes e serviços e também a sua API.

Equinox é também um módulo de tempo de execução que permite o desenvolvimento de uma aplicação como um conjunto de pacotes (ou *plug-ins*, como são designados no Eclipse), utilizando serviços e infraestrutura comuns.

A arquitetura da plataforma Eclipse é extensível e baseada em *plug-ins*. Um *plug-in* representa a menor unidade de modularização, pacotes estruturados de código e/ou dados que permitem contribuir na ampliação de funcionalidades de um sistema. Para a declaração de um *plug-in* são definidos dois ficheiros: manifesto *OSGi* (`MANIFEST.MF`) e manifesto *plug-in* (`plug-in.xml`). O ficheiro `MANIFEST.MF` contém informação sobre o *plug-in*, tais como, nome, ID, o número de versão, a referência a dependências a outros *plug-in*, a versão da linguagem Java utilizada e outras informações sobre o *plug-in* de carater opcional. O ficheiro `plug-in.xml` descreve as extensões e os pontos de extensão. A declaração de pontos de extensão consiste na definição de um contrato que permite que outros *plug-ins* possam estender as funcionalidades de um determinado *plug-in*. As extensões são contribuições dadas por um *plug-in* com base no contrato definido por um ponto de extensão existente.

A base da arquitetura da plataforma Eclipse é constituída pelos seguintes componentes:

- A plataforma de tempo de execução;
- O *workspace* é o ambiente que permite a manipulação dos recursos das ferramentas;
- O *workbench* representa os elementos de interface do utilizador da plataforma Eclipse;
- O sistema de controlo de versões (VCM);
- O sistema de ajuda;

A Figura 4.7 permite ilustrar a arquitetura do Eclipse SDK.

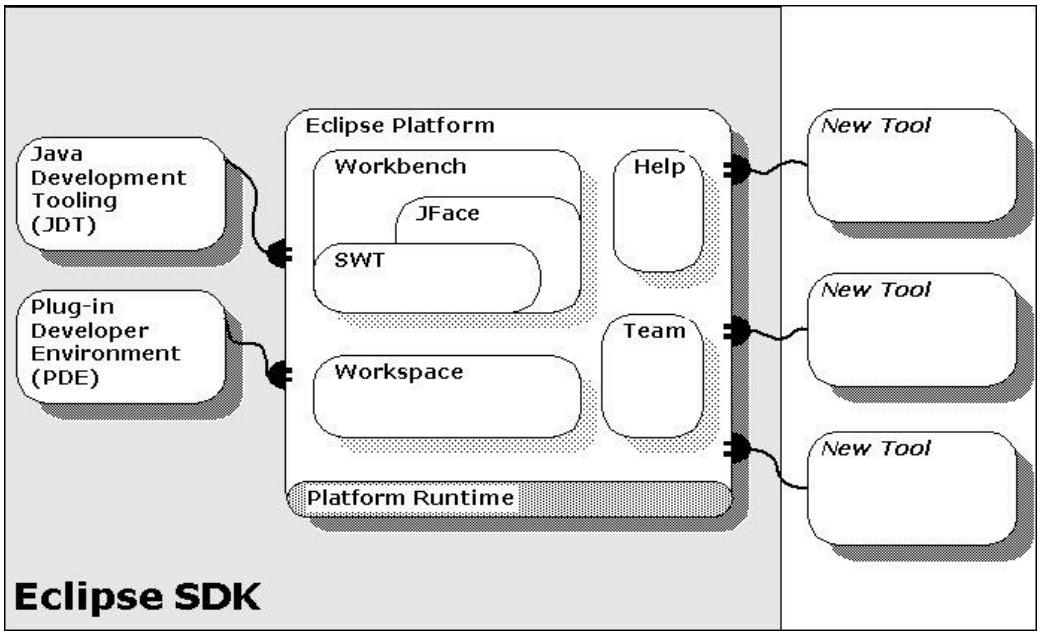

Figura 4.7 Esboço da Arquitetura do Eclipse [42]

A plataforma de tempo de execução foi implementado com base no modelo de serviços de *OSGi*, representa o núcleo do Eclipse, cuja responsabilidade é de inicializar a plataforma e dinamicamente carregar os *plug-ins* que podem ser requisitados. Com exceção deste componente, os restantes estão integrados na plataforma Eclipse como *plug-ins*. O *workspace* define um modelo de recursos que permite que os *plug-ins* possam interagir com os recursos, incluindo projetos, pastas e ficheiros. Este componente é um ponto central, pois contém os ficheiros de dados dos utilizadores. O *workbench* é uma plataforma que permite que os utilizadores possam navegar e manipular os recursos do *workspace*. Este contém pontos de extensão que permitem que um determinado *plug-in* possa estender os componentes de interface de utilizador do Eclipse, por exemplo, a definição de ações para a barra de tarefas, menus, editores, etc. O *workbench* é composto por dois elementos: o JFace e o *Standart Widget Toolkit* (SWT). O SWT é um conjunto de ferramentas de baixo nível, projetado para tornar eficaz a portabilidade de elementos de interface do utilizador nos sistemas operativos. O JFace foi construído sobre o SWT, é um componente com maior nível de abstração, o seu objetivo é proporcionar um conjunto de componentes reutilizáveis que permitam tornar mais fácil a definição de uma interface gráfica do utilizador em Java. O *plug-in* VCM é uma implementação de um sistema de controlo de versão disponível na plataforma Eclipse. Por fim, o sistema de ajuda disponibiliza documentação *online*, de modo, proporcionar assistência aos utilizadores de determinada aplicação.

A plataforma Eclipse inclui duas ferramentas importantes para a sua estrutura. A ferramenta de desenvolvimento Java (JDT) implementa um ambiente com vários recursos para o desenvolvimento de aplicações em Java. E o ambiente de

desenvolvimento de *plug-in* (PDE) que proporciona ferramentas especializadas que permitem tornar mais simples o desenvolvimento de *plug-ins*, não apenas para o Eclipse, mas também para outras aplicações.

4.2.3 Integração e Dependências do *plug-in* PESTTMuTest

Visto que, a arquitetura do Eclipse é composta em trono de *plug-ins*, a integração da aplicação PESTTMuTest nesta plataforma será feita sobre a forma de um *plug-in.*

No processo de construção do PESTTMuTest, foi necessário a identificação de *plug-ins* do qual o este dependerá. A Figura 4.8 apresenta a lista das dependências do *plug-in* PESTTMuTest. Esta informação representa um excerto do ficheiro `MANIFEST.MF`.

```
1. org.eclipse.core.resources;bundle-version="3.8.100",
2. org.eclipse.core.runtime;bundle-version="3.9.0",
3. org.eclipse.jdt.core;bundle-version="3.9.0",
4. org.eclipse.jdt.launching;bundle-version="3.7.0",
5. org.eclipse.jface.text;bundle-version="3.8.101",
6. org.eclipse.ui;bundle-version="3.105.0",
7. org.eclipse.ui.console;bundle-version="3.5.200",
8. org.junit;bundle-version="4.11.0"
```

Figura 4.8 Lista das dependências do PESTTMuTest

As dependências mais importantes estão relacionadas com os seguintes identificadores dos *plug-ins* da ferramenta de desenvolvimento Java (JDT):

- `org.eclipse.jdt.core` – define elementos essenciais e a API do Java. O compilador Java foi utilizado para gerar a árvore de sintaxe abstrata (AST) e validar as alterações feitas no projeto Java, para garantir a produção de mutantes válidos. A AST é uma representação abstrata da estrutura semântica do código fonte de um ficheiro Java. Esta representação permite que a análise e a manipulação da estrutura semântica de um ficheiro. Para este trabalho, este elemento é fundamental, pois será a partir da AST que serão determinadas as instruções para a aplicação de mutações e aplicadas modificações no ficheiro para a concretização de um mutante. Na seção seguinte será abordado o método utilizado para percorrer a AST de um determinado ficheiro e a forma como foram aplicadas as modificações no ficheiro.
- `org.eclipse.jdt.launching` – para o PESTTMuTest este *plug-in* contribui na obtenção das classes de teste definidas pelos utilizadores, para

posteriormente executá-las sobre os mutantes gerados. Na próxima seção, será apresentada de forma detalhada como foi utilizado este *plug-in*.

- `org.junit` – O JUnit foi utilizado no PESTTMuTest para executar os testes definidos pelo utilizador sobre os mutantes. Foi utilizada a versão 4.0 do JUnit.

De uma forma geral as restantes dependências estão associadas a outros plug-ins da plataforma Eclipse, que permitem, por exemplo, a manipulação de recursos do *workspace*, a definição de aplicativos de interface para a interação com interface do utilizador (definição de ações da barra de ferramentas, de menus, etc.).

A Figura 4.9 ilustra de forma simplificada um gráfico com as dependências do *plug-in* PESTTMuTest.

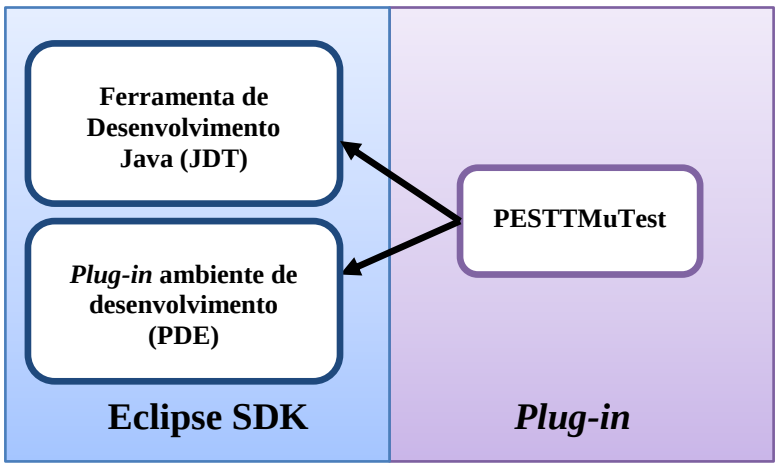

Figura 4.9 Visão simplificada das dependências do PESTTMuTest

4.2.4 Estrutura interna do *plug-in* PESTTMuTest

As suas funcionalidades do PESTTMuTest foram projetadas com a finalidade de tornar mais eficaz o ensino da aplicação dos testes de mutação em Java. Para a implementação do *plug-in* foram utilizados padrões de desenho.

A definição de padrões de desenho, utilizados na Engenharia de Software tem permitido a reutilização de soluções para problemas comuns, que podem surgir durante o processo de desenvolvimento de uma aplicação, através da padronização de boas soluções de problemas recorrentes. Assim, um padrão descreve um problema que ocorre várias vezes em um determinado contexto, e a parte central da solução para o problema, para que seja possível aplicar a solução várias vezes sem nunca implementá-la da mesma forma.

Os padrões de desenho utilizados na concretização do PESTTMuTest foram os seguintes: Modelo-Vista-Controlador (MVC) [39], Fábrica, Estratégia, Visitante, Observador e Singularidade [40].

Modelo-Vista-Controlador (MVC) divide a estrutura da aplicação em três componentes: modelo, vista e o controlador. Esta divisão tem como objetivo minimizar as dependências entre a lógica de programação e a interface gráfica, para facilitar as possíveis alterações em uma das componentes. O modelo é responsável por representar a informação/dados da aplicação e a definição de regras para a manipulação dos dados. A vista representa os componentes gráficos de uma aplicação. Por fim, o controlador que tem como função controlar A lógica de comunicação entre o modelo e a vista. Este é também responsável pelo tratamento da interação do utilizador com a aplicação.

O padrão Fábrica tem como objetivo a definição de uma interface, cuja responsabilidade é a criação do conjunto de objetos com uma implementação particular sem especificar de forma concreta a classe a que correspondem. A fábrica encapsula a responsabilidade e o processo de criação dos objetos, isolando do cliente a implementação das classes, permitindo apenas a manipulação das instâncias através das suas interfaces. Assim, é possível controlar a família de objetos que podem ser criados por uma aplicação. Este padrão foi utilizado para a criação dos objetos dos operadores de mutação.

A Estratégia é um padrão que define uma família de algoritmos, e encapsula cada um em uma classe, fazendo com que estes sejam intercambiáveis. Assim, é definido cada algoritmo/política/estratégia como uma classe separada, com uma interface comum. O padrão Estratégia foi aplicado na concretização do PESTTMuTest para permitir que cada classe que representa um operador de mutação seja responsável por definir o algoritmo utilizado para a geração dos mutantes e também pela definição do algoritmo que permite desfazer as mutações por este gerado.

O padrão Visitante permite representar uma operação a ser realizada sobre os elementos de uma estrutura de objetos, sem que seja necessário modificar a classe dos elementos sobre os quais se opera. Assim, com base neste padrão, o *plug-in* pode percorrer a AST de um ficheiro Java, usando um visitante. Foi ainda utilizado este padrão para a identificação dos ficheiros Java do projeto existente no *workspace* que representam as classes de testes.

O Observador é um padrão que define a dependência de um-para-muitos entre objetos. Assim, quando um objeto altera o seu estado, todos os seus dependentes devem ser notificados e atualizados automaticamente. O padrão Observador é útil quando diferentes tipos de objetos (observadores) estão interessados na mudança de estado ou

nos eventos de outro objeto. Com base neste padrão será atualizado automaticamente as vistas da interface gráfica.

O padrão Singularidade assegura que uma classe tem uma única instância e providencia um acesso global à instância. Algumas classes definidas no *plug-in* PESTTMuTest necessitam que seja criada apenas uma instância para a utilização das suas funções durante a execução do *plug-in*. Uma vez que uma fábrica é normalmente uma "Coisa única", este padrão foi utilizado na definição da classe que cria os objetos dos operadores de mutação e da classe utilizada para a concretização dos elementos que compõem a árvore com informação sobre os operadores de mutação.

A definição da estrutura e a comunicação dos vários componentes (vistas, controladores, representação da informação) do *plug-in* PESTTMuTest, baseou-se nos padrões MVC e Observador. Assim, este *plug-in* foi dividido em três componentes:

- *main*;
- *domain;*
- *ui*.

A componente *main* contém o elemento ativador do *plug-in.* Este é responsável pela ativação do PESTTMuTest, ou seja, o carregamento dos elementos necessários para o correto funcionamento do *plug-in* na plataforma Eclipse, de modo a permitir que o PESTTMuTest se ligue com o ambiente externo. O componente *domain* é o núcleo do *plug-in*, onde são definidas as estruturas e a relação dos conceitos relacionados com os testes de mutação, como por exemplo, o operador de mutação, a mutação. Os elementos que compõem a parte da interface gráfica (perspetiva, vista e editores) para a interação do utilizador com o PESTTMuTest estão definidos no pacote *ui.* A Figura 4.10 apresenta uma visão geral da estrutura definida para o *plug-in*.

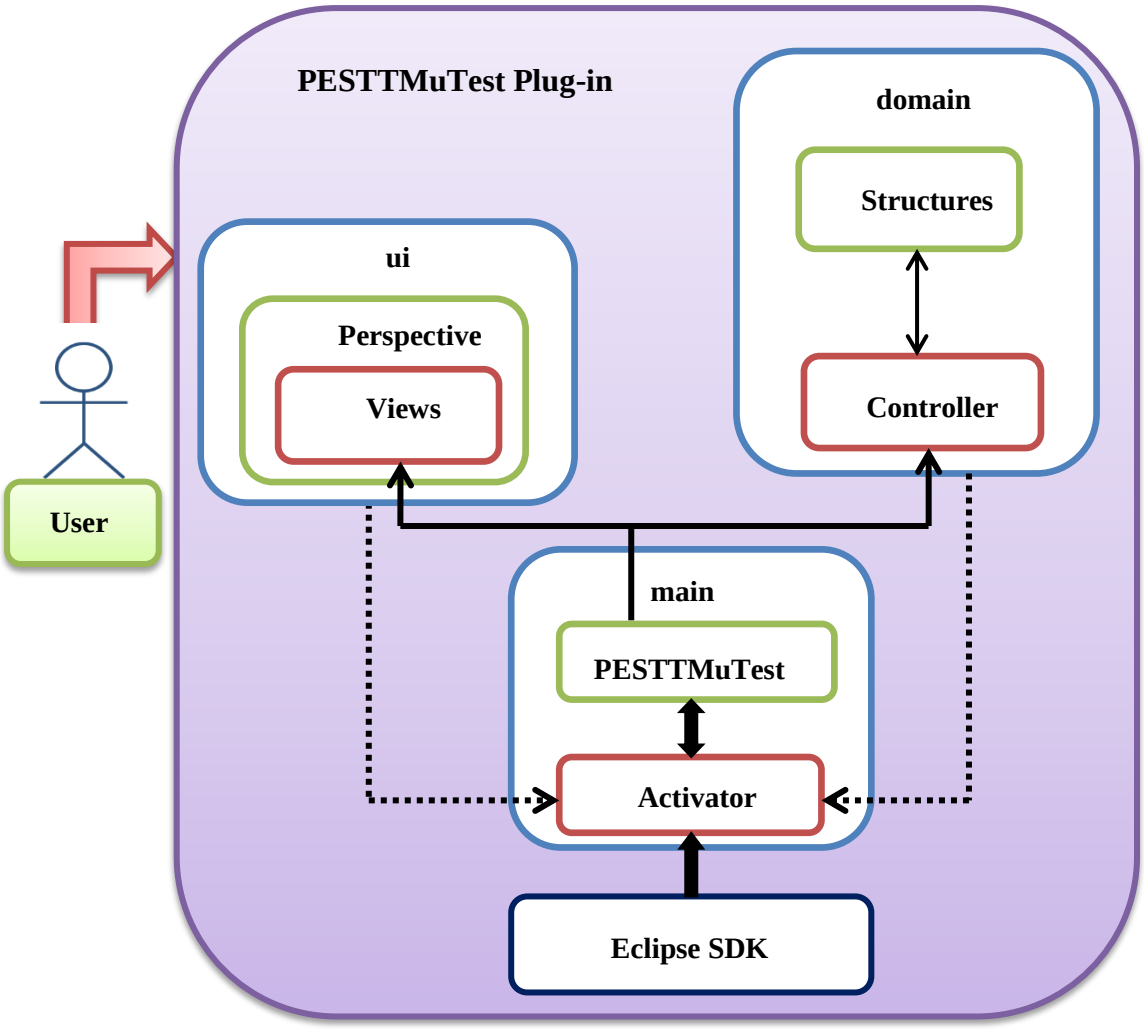

Figura 4.10 Visão geral da arquitetura do *plug-in* PESTTMuTest

As subsecções seguintes estão reservadas para a explicação dos componentes e dos processos de maior relevância do PESTTMuTest.

4.2.4.1 Implementação dos operadores de mutação escolhidos

Existem vários operadores de mutação definidos para a geração de mutantes, que podem ser implementados e utilizados no processo de testes de mutação. Embora os operadores tenham o mesmo objetivo, geração de mutantes, o modo como a é concretizado, depende de cada operador. Assim, com base nos padrões de desenho Fábrica e Estratégia que implementado o conceito de operador de mutação no *plug-in* PESTTMuTest. O padrão Estratégia foi utilizado para a definição de uma interface designada `IMutationOperators` (ver Anexo A.1), que contém a assinatura dos métodos considerados essenciais e cuja implementação dependerá da classe que representa a concretização do operador de mutação.

A definição do método que permite desfazer as alterações em um determinado ficheiro Java, criadas para a geração do mutante é fundamental, para garantir que a aplicação gera mutantes de FOM.

Como referido na secção 4.2.4 , com base no padrão Fábrica foi definida uma classe abstrata `AbstractFactoryMutationOperators`, que contém um único método que cria instâncias dos operadores de mutação. Este método será implementado pelas classes que a estendem, e que representam os sete grupos dos operadores de mutação (Operadores tradicionais, Controle de acesso, Herança, Polimorfismo, Sobrecarga, Características específicas de Java, Erros comuns de programação) descritos na secção 3.2 . Estas classes são responsáveis pela criação dos objetos que representam os operadores de mutação que compõem cada um dos grupos. A criação destes objetos depende dos eventos gerados pelas ações do utilizador. Assim, após o utilizador selecionar os elementos representados na vista `Mutation Operators` (ver secção 4.2.4.5), e iniciado o processo de testes de mutação através da ativação do comando `Start Process Mutation Test` (ver secção 4.2.4.5), cabe a classe `MutationOperatorsFactory` responsável pela criação dos objetos dos grupos dos operadores de mutação, determinar qual o objeto do respetivo grupo que deve ser criado, e fornecer a informação necessária para este instanciar o objeto do operador de mutação que deve ser criado. Isto é possível, através da informação obtidas nos elementos que compõem a árvore definida na vista `Mutation Operators (ver` secção 4.2.4.5). O diagrama de sequência descrito na Figura 4.11 permite ilustrar o processo de criação de um objeto que representa um operador de mutação.

O Figura 4.12 descreve o diagrama de classes UML onde são apresentadas as classes que fazem parte da concretização do conceito de operador de mutação no PESTTMuTest.

A definição da estrutura para a concretização dos operadores de mutação neste *plug-in*, torna possível a integração futura de novos operadores de mutação, sem que seja comprometido o bom funcionamento do *plug-in.*

A Tabela 4.1 apresenta os operadores de mutação implementados no *plug-in* PESTTMuTest.

Categoria	Abreviação	Descrição
Tradicional	AOR	Substituição do operador aritmético
Controle de acesso	AMC	Alteração do modificador de acesso

Categoria	Abreviação	Descrição
Características específicas de Java	JTD	Eliminar a palavra *this*
	JSC	Alteração do modificador *static*
	JID	Elimina a inicialização do atributo

Tabela 4.1 Operadores de mutação implementados

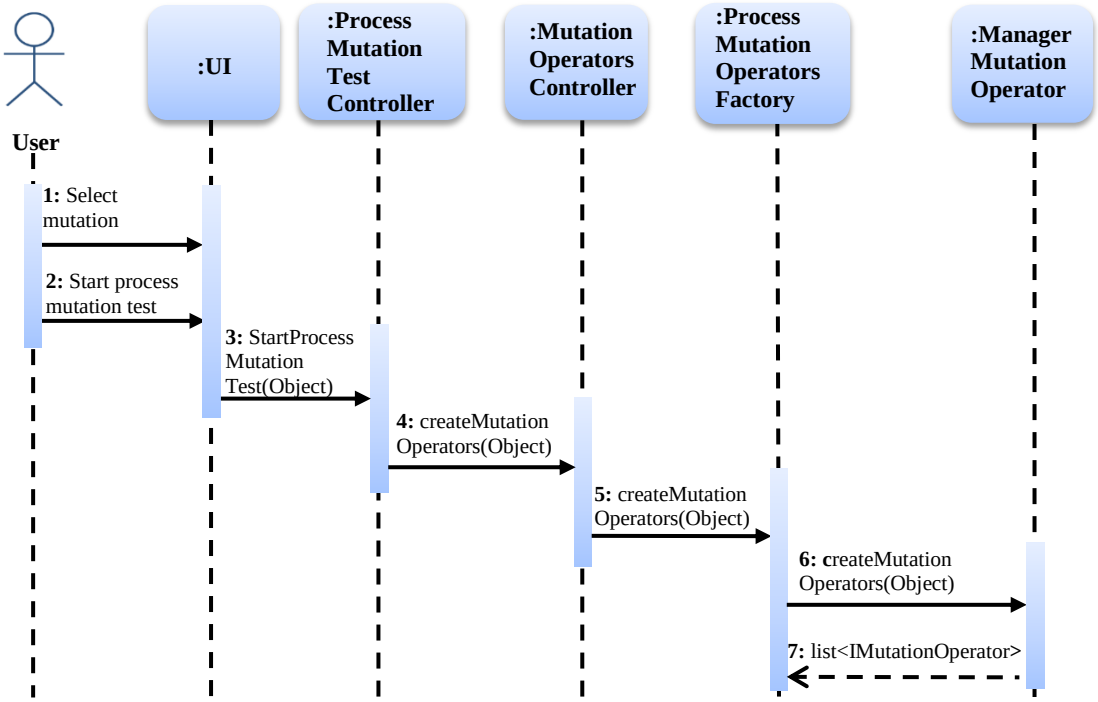

Figura 4.11 Diagrama de sequência para a criação de operadores de mutação

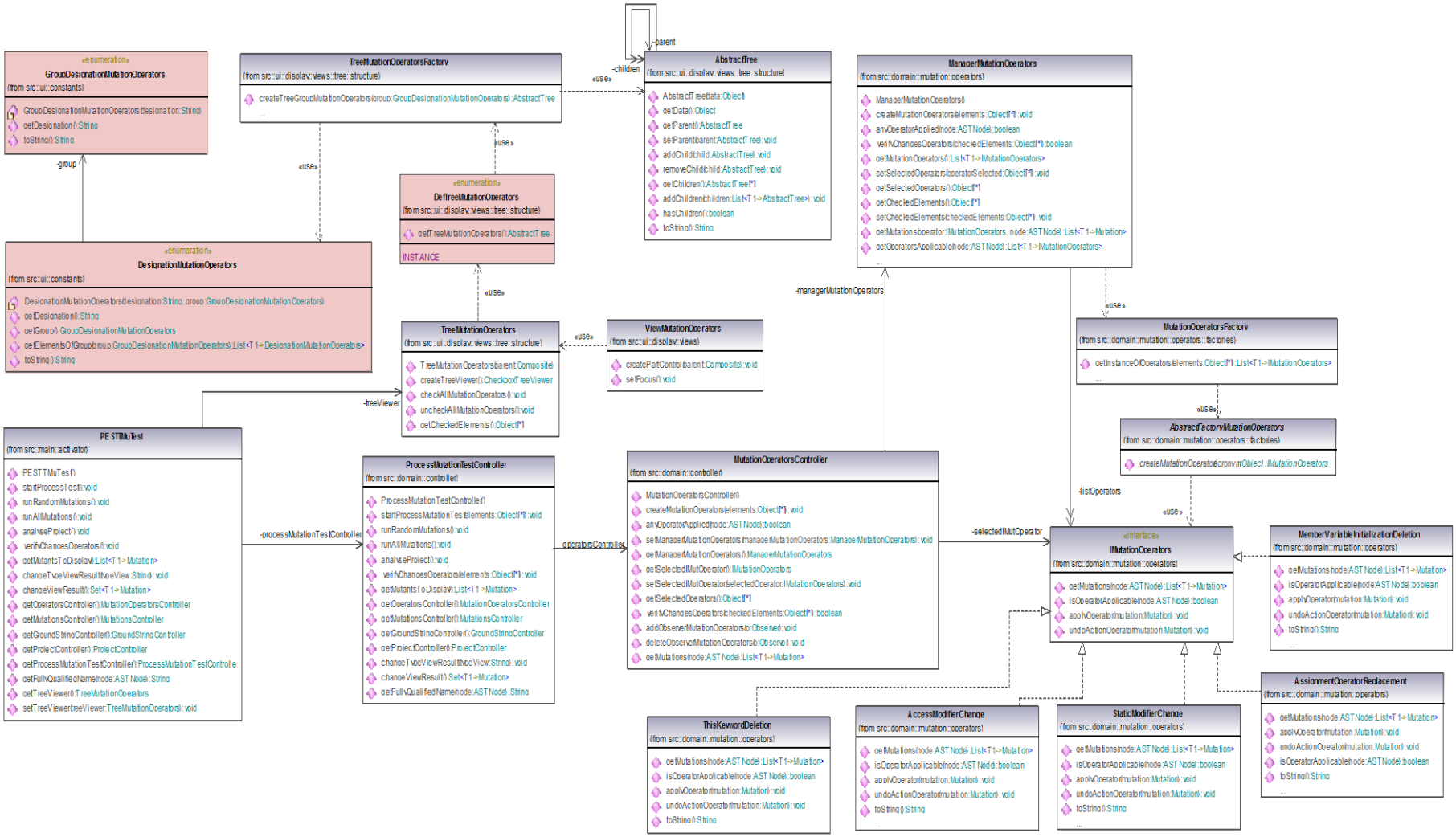

Figura 4.12 Diagrama de Classes UML para a criação de operadores de mutação

4.2.4.2 **Processo de análise da aplicação Java**

Para a aplicação de testes de mutação num programa Java é necessário que seja feita uma análise sobre os ficheiros que compõem a aplicação, para a identificação das expressões onde poderão ser aplicadas as mutações. Um ficheiro Java pode ser analisado através do seu código fonte (ficheiro .java) ou do seu *bytecode* (ficheiro .class). O código fonte de um é uma linguagem de mais alto nível utilizada pelos programadores para a construção de sistemas Java, enquanto o *bytecode* é uma linguagem de nível intermédio, gerada pelo compilador Java para ser convertida em linguagem máquina pela máquina virtual Java (JVM), onde são carregados e executados as aplicações Java.

Sendo o objetivo deste *plug-in* comunicar de forma didática com os alunos é fundamental a apresentação de informação associada aos testes de mutação e garantir que a informação apresentada esteja o mais correta possível, por exemplo, se não for feita uma validação dos mutantes gerados podem ser apresentados mutantes inválidos, ou seja, mutantes que causam erros de compilação. Isto impossibilitaria a execução dos testes sobre estes mutantes. Utilizando a manipulação (análise e modificação) do código fonte irá permitir do ponto de vista pedagógico mostrar aos utilizadores as expressões de um ficheiro Java onde podem ser aplicadas mutações, apresentar as possíveis mutações que podem ser aplicadas sobre uma determinada expressão e garantir a geração e apresentação de mutantes válidos. Isto é possível através da verificação de erros de compilação (ver secção 4.2.4.3), o que não seria possível com a manipulação de *bytecode*.

A alteração do código fonte de um ficheiro Java ou do *bytecode* são métodos que existem para que sejam gerados os mutantes, assim com base nas vantagens (apresentadas no parágrafo anterior), que a manipulação de código fonte traz para a concretização do *plug-in* PESTTMuTest, a geração de um mutante será feita através da alteração do código fonte.

Com o objetivo de salvaguardar os projetos dos utilizadores, optou-se pela criação de cópia do projeto sobre o qual será aplicada as operações do processo de testes mutação, como por exemplo, a geração de mutantes, a execução dos testes sobre os mutantes.

Para criada uma cópia de um projeto Java disponível no *workspace* é necessário aceder a este recurso. Para isso, foi utilizada a biblioteca `org.eclipse.core.resources`, disponível na plataforma Eclipse que fornece o suporte para a gestão do *workspace* e dos seus recursos. Os recursos no *workspace* estão organizados com base num estrutura em árvore, e no topo da hierarquia existe apenas um recurso designado por raiz do *workspace*. No nível seguinte encontram-se os

projetos, e nas camadas mais inferiores, encontram-se as pastas e os ficheiros. Assim, para obter os projetos do *workspace* basta pedir ao elemento que está no topo da hierarquia, a raiz do *workspace*. Contudo, podem existir projetos no *workspace* que não são do tipo Java ou o são, mas encontram-se fechados, e assim não é possível ter acesso aos seus conteúdos. Então, foi necessário definir um filtro antes de ser feita a cópia de cada projeto, verificando se o projeto está aberto, e se este é do tipo Java.

Os recursos (raiz do *workspace*, projeto, paste e ficheiro) que o *workspace* pode conter, implementam a interface `org.eclipse.core.resources.IResource`. Esta interface tem um método que permite que um recurso possa fazer uma cópia de si mesmo. Para criar a cópia de um projeto é necessário definir o seu caminho absoluto. Este caminho contém a informação sobre a localização do projeto e o seu nome. Para a definição do caminho absoluto da cópia do projeto foi obtido o caminho absoluto do projeto a ser copiado e adicionado o texto "`CopyOfProject`" ao nome de todas as cópias. Deste modo, será possível distinguir as cópias dos projetos originais.

A primeira vez que o utilizador inicia o processo de testes de mutação, através da ativação do respetivo comando (ver seção 4.2.4.5), são criadas as cópias dos projetos. No entanto, durante o processo de testes de mutação o utilizador tem a liberdade de realizar alterações sobre *workspace*, por exemplo, alterar um projeto, criar novos projetos, o que implica a atualização das cópias, caso contrário, a informação utilizada para a realização das operações do processo de testes de mutação não coincidirá com os novos dados do *workspace*. Assim, é solicitado ao utilizador que inicie novamente o processo de testes (ver secção 4.1.2) e desta forma serão criadas novas cópias. As alterações sobre os recursos do *workspace* são descritas pelos eventos gerados pela interface `org.eclipse.core.resources.IResourceChangeEvent`. Sendo as cópias recursos do *workspace,*é fundamental que o *plug-in* consiga diferenciar os eventos associados a modificação das cópias dos projetos, dos eventos gerados pelos recursos definidos pelo utilizador no *workspace.* Para tratar estes eventos foi definida a classe `MyResourceChangeReporter`, assim, será obtido o projeto associado ao evento gerado para verificar se no seu nome contém o texto "`CopyOfProject`", caso contenha o evento será ignorado, senão fica registado que as cópias estão desatualizadas. As cópias foram definidas para que não estejam visíveis ao utilizador, isto porque, pretende-se que o utilizador tenha a ilusão que as operações estão a ser todas realizadas sobre os seus projetos e também para evitar que sejam geradas confusões, por exemplo, o utilizador alterar algum ficheiro da cópia do projeto a julgar que está a efetuar alterações sobre o seu projeto. As cópias dos projetos são eliminadas quando termina o ciclo de vida do *plug-in* PESTTMuTest.

Uma das formas disponíveis no JDT para a manipulação de código fonte de um ficheiro Java é através da sua árvore de sintaxe abstrata (AST). A AST mapeia o código fonte Java em uma estrutura em forma de árvore, identificando de forma hierárquica todos os elementos que compõem uma determinada classe Java. Esta estrutura é criada durante o processo de compilação pelo compilador disponível no JDT, designado por Eclipse compilador para Java (JCE), este constrói uma representação abstrata do código fonte Java, que denominamos por AST. A biblioteca `org.eclipse.jdt.core.dom` disponibiliza classes que permitem manipular a AST. A AST de um ficheiro é obtida utilizando o método da classe `org.eclipse.jdt.core.dom.ASTParser`. O utilizado para criar a AST devolve a raiz da AST, que é um objeto do tipo `org.eclipse.jdt.core.dom.CompilationUnit`.

Com base no padrão Visitante foi definida o método para navegar a AST. A raiz da AST e os restantes nós que a compõem são subclasses de `org.eclipse.jdt.core.dom.ASTNode`. Baseado no padrão Visitante o método `accept` desta classe, recebe como parâmetro um objeto do tipo `org.eclipse.jdt.core.dom.ASTVisitor`, permitindo que cada nó da AST possa aceitar um visitante para navegar sobre a sua estrutura. Assim, partindo do nó da árvore que faz a primeira chamada ao método `accept`, cada nó é responsável por enviar o visitante como argumento para todos os métodos `accept` dos seus nós filhos. De forma recursiva é feita a navegação completa sobre toda a AST. Caso se pretende analisar apenas alguns nós da AST é necessário definir uma subclasse de `org.eclipse.jdt.core.dom.ASTVisitor` e redefinir os método `visit` associado aos nós que se pretende analisar.

O *plug-in* PESTTMuTest precisa de navegar sobre a AST dos ficheiros Java para obter duas informações diferentes:

1. Identificar as classes de teste existentes no projeto Java. Porque não se pode gerar mutantes sobre estas classes, uma vez que estas têm como finalidade testar os mutantes gerados e deste resultado, fazer uma avaliação da sua eficácia;
2. Analisar alguns nós específicos da AST com o objetivo de identificar as expressões (*ground string*) dos ficheiros onde podem ser aplicados os operadores de mutação selecionados pelo utilizador (ver secção 4.2.4.5) para a geração de mutantes.

Deste modo, foram criadas duas subclasses (`TestClassesVisitor` e `SourceCodeVisitor`) de `org.eclipse.jdt.core.dom.ASTVisitor` uma para identificar as classes de teste e outra para identificar as *ground string*. Apesar de

ambas as classes navegarem sobre a AST, não foi possível utilizar uma única classe para obter as duas informações. Assim, durante a análise de um projeto, para cada classe Java do projeto é verificado se esta é uma classe de teste, utilizando a classe `TestClassesVisitor` que analisa a AST do ficheiro e verificar se existe alguma notação "@Test". No caso, em que a condição é verdadeira, então identificada uma classe de teste e a informação sobre esta classe será guardada para ser utilizada no processo de execução de testes sobre os mutantes (ver secção 4.2.4.4). Se esta não for uma classe de teste, então será analisada pela classe `SourceCodeVisitor` para que sejam identificadas as *ground string*. Isto é feito através das instâncias dos operadores de mutação, criadas com base nos operadores selecionados pelo utilizador (ver secção 4.2.4.5), pois estas têm informação necessária para determinar se um nó da AST é válido para que sejam aplicadas mutações.

A Figura 4.13 descreve o diagrama de classes UML onde são apresentadas as classes que fazem parte da concretização do processo de análise de um projeto Java no PESTTMuTest.

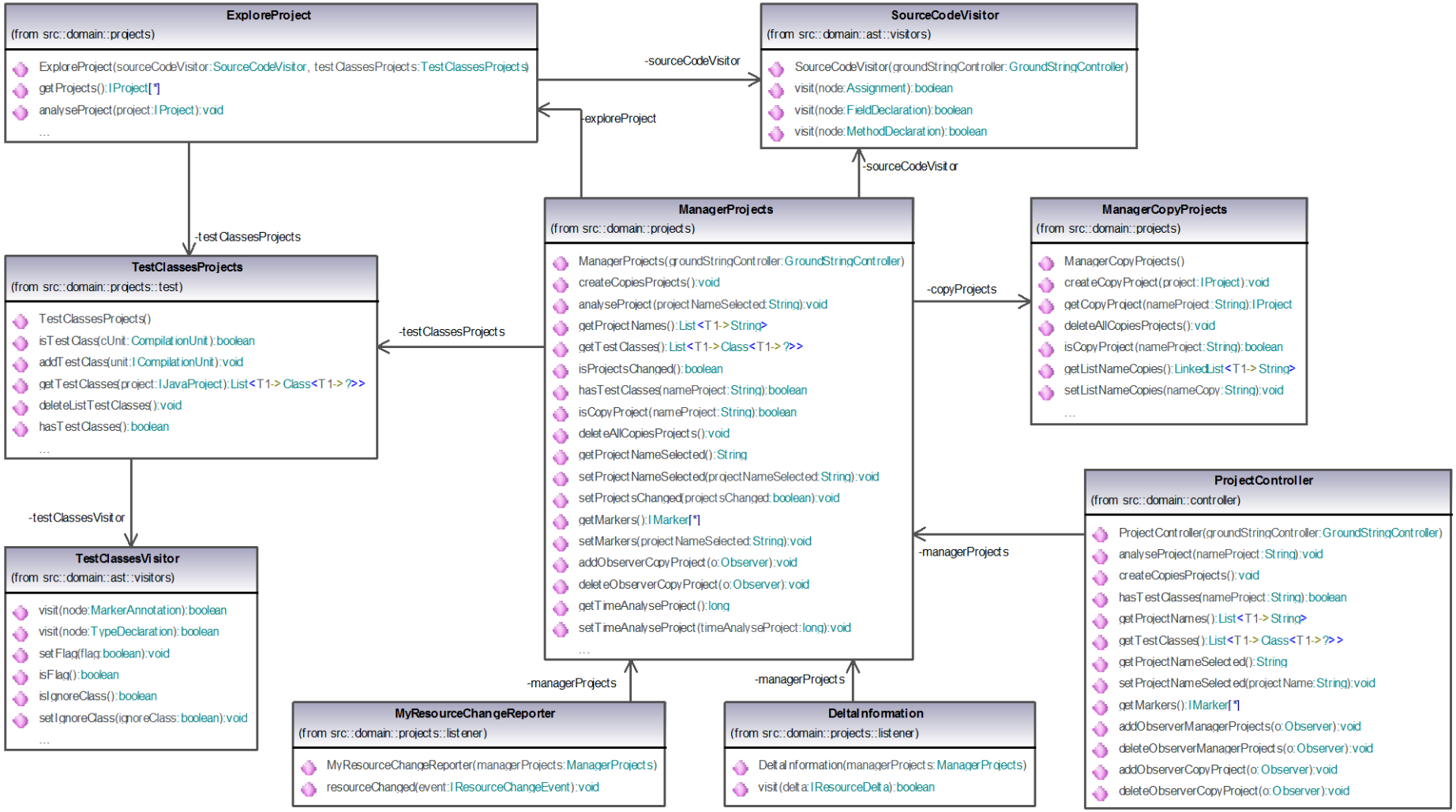

Figura 4.13 Diagrama de Classes UML para a analise de projetoJava

4.2.4.3 **Processo de geração dos mutantes**

Um mutante é a concretização de uma mutação gerada pela aplicação de um operador de mutação. Nesta secção será apresentada a forma como foi concretizado os conceitos de mutante e mutação no *plug-in* PESTTMuTest.

Quando um operador de mutação é aplicado sobre uma *ground string* são geradas mutações de acordo com o algoritmo definido pelo operador, e a partir destas mutações serão criados os mutantes. No entanto, a geração de um mutante é um processo complexo, porque é fundamental garantir as seguintes condições: gerar mutantes válidos, gerar mutantes de primeira ordem (FOM), anulando as modificações feitas sobre a aplicação para a geração de um mutante. Para a concretização do conceito de mutação foi definida a classe `Mutation` (ver Anexo A.2) que irá conter a informação necessária que permita garantir as condições descritas anteriormente neste parágrafo. Assim, para a geração de um mutante considerou-se que um objeto `Mutation` deve conter as seguintes informações: o operador de mutação responsável pela criação do objeto do tipo `Mutation`; o nó da AST que representa a *ground string*; o estado original da *ground string*; e a alteração que o operador de mutação pretende aplicar sobre a *ground string*. A informação sobre o operador de mutação é essencial, pois é este é que que executa os algoritmos alterar e desfazer as alterações aplicadas sobre o projeto Java. Como referido na secção 4.2.4.2, foi utilizada a AST para efetuar as alterações sobre o código fonte de um ficheiro Java. Portanto, cada *ground string* representa uma parte da AST de um ficheiro .java, ou seja, um objeto do tipo `org.eclipse.jdt.core.dom.ASTNode`. A informação sobre o tipo de alteração que deve ser feita é igualmente importante para que o operador de mutação saiba qual a modificação que deve ser aplicada sobre a árvore. Por fim, a informação sobre o elemento que foi modificado na AST para a geração de mutante, é essencial para que o operador de mutação possa desfazer as alterações aplicadas sobre o nó da AST.

No parágrafo anterior foi explicado o elemento `Mutation` (ver Anexo A.2) que contém a informação necessária para a concretização de um mutante, e com base nesta informação serão realizadas determinadas operações para que seja gerado um mutante válido.

Considerando que o processo de geração de um mutante tem como elemento fundamental um objeto do tipo `Mutation`, a sua concretização é realizada nas seguintes etapas: alteração da AST, aplicação das modificações na unidade de compilação, validação das modificações aplicadas e alteração da unidade de compilação para o seu estado original.

A primeira etapa é concretizada, a partir da informação contida no objeto `Mutation`, assim através da aplicação do operador de mutação sobre o nó da AST serão realizadas modificações na árvore de um determinado ficheiro Java.

A segunda etapa tem como objetivo verificar se as modificações aplicadas sobre o projeto têm como resultado um programa compilável, isto é necessário porque nem todas as mutações geram mutantes válidos. Para isso, foi utilizada a ferramenta do JDT, o compilador Java (ECJ). Apesar de a AST estar associada a uma unidade de compilação, quando são efetuadas alterações sobre ela, estas não são refletidas para a unidade de compilação, porque a AST está em memória, enquanto a unidade de compilação, que é a representação física de um ficheiro Java, está guardar em disco. O que implica que as alterações feitas sobre a AST têm que ser salvas em disco, para que possa ser válido o mutante gerado através da compilação do projeto sobre o qual foi aplicada a mutação. Assim é obtida a unidade de compilação, a partir da raiz da AST (`org.eclipse.jdt.core.dom.CompilationUnit`), que por sua vez pode ser obtida a partir de qualquer nó da AST. Tendo a unidade de compilação, esta pode ser alterada através de uma cópia sua em memória, designado por cópia de trabalho. Assim, é criada uma cópia de trabalho utilizando o método `getWorkingCopy`. Para subter as alterações ao ficheiro Java, a cópia de trabalho modifica uma zona da memória, designado por *buffer*. O conteúdo que está no *buffer* manipulado pela cópia de trabalho, é substituído pela informação da árvore da unidade compilação em forma de texto (que contém as alterações aplicadas), obtida através da raiz da AST utilizando o método `toString`. Depois, serão aplicados os seguintes métodos: `reconcilie` e `commitWorkingCopy`. O método `reconcilie` sincroniza os dados da cópia de trabalho com o *buffer*. E o método `commitWorkingCopy` substitui a os dados da unidade de compilação original, alterando assim a informação em disco.

A terceira etapa consiste em validar se as alterações aplicadas ao projeto não geram erros de compilação, para isso, é necessário obter o projeto associado a unidade de compilação alterada na segunda etapa. A partir da unidade de compilação é obtido um objeto do tipo `org.eclipse.core.resources.IProject` e são executados os métodos `build` e `findMarkers`. O método `build` tem como função compilar o projeto. Para verificar se foram gerados erros de compilação, será aplicado o método `findMarkers`, que devolve um conjunto de meta-dados associado ao projeto, e através desta informação é verificado se existe algum elemento do cujo identificador representa um `SEVERITY_ERROR`. Caso exista, então as alterações aplicadas ao projeto, geraram erros de compilação, o que significa que o mutante não é válido

A última etapa é necessária para garantir que todas as modificações feitas na unidade de compilação para gerar o mutante são desfeitas, e deste modo, assegurar que

o programa estará no seu estado original. Assim, esta etapa está dividida em duas partes: alteração da AST para o seu estado original e tornar estas alterações persistentes em disco. Como foi referida nesta secção, o operador de mutação é responsável por repor o estado original da AST. Desta forma é concretizada a primeira parte desta etapa. A segunda parte consiste em repetir os mesmos passos descritos na primeira etapa para salvar as alterações feitas na AST em disco, e depois de substituída a informação sobre a unidade de compilação será eliminada a cópia de trabalho. Esta etapa foi dividida em duas partes com a intensão de evitar que para cada um mutante gerado, seja criada uma cópia de trabalho, e minimizar o número das alterações salvas em disco.

O *plug-in* PESTTMuTest pode gerar mutantes para duas situações: apresentar o conjunto de mutantes gerados da aplicação de um operador de mutação a uma determinada *ground string* ou para a execução do conjunto de casos de teste (ver secção 4.2.4.5). É importante referir estas duas situações para explicar quando é que são aplicadas as duas partes que compõem a última etapa. Na primeira situação será alterada a AST para o seu estado original caso o mutante gerado seja válido ou inválido, e a segunda parte será realizada apenas após a validação de todas as mutações, e assim obter um conjunto com apenas as mutações válidas, para que possam ser apresentadas ao utilizador. Para a segunda situação descrita, se o mutante gerado for válido, será realizado o processo descrito na seção 4.2.4.4 e concluído o processo será aplicada a parte 1 referente a última etapa, o mesmo acontece caso o mutante seja inválido. A parte 2 só será executada depois de terem sido gerados todo os possíveis mutantes resultantes da aplicação de todos os operadores de mutação aptos para gerarem mutações sobre uma determinada *ground string*. Deste modo, serão salvas as alterações em disco, quando se prender validar um mutante ou depois de concluir a validação de todas as mutações que podem ser aplicadas a uma determinada *ground string*.

A Figura 4.14 apresenta o diagrama de classes UML onde são apresentadas as classes que compõem o processo de geração de um mutante no *plug-in* PESTTMuTest.

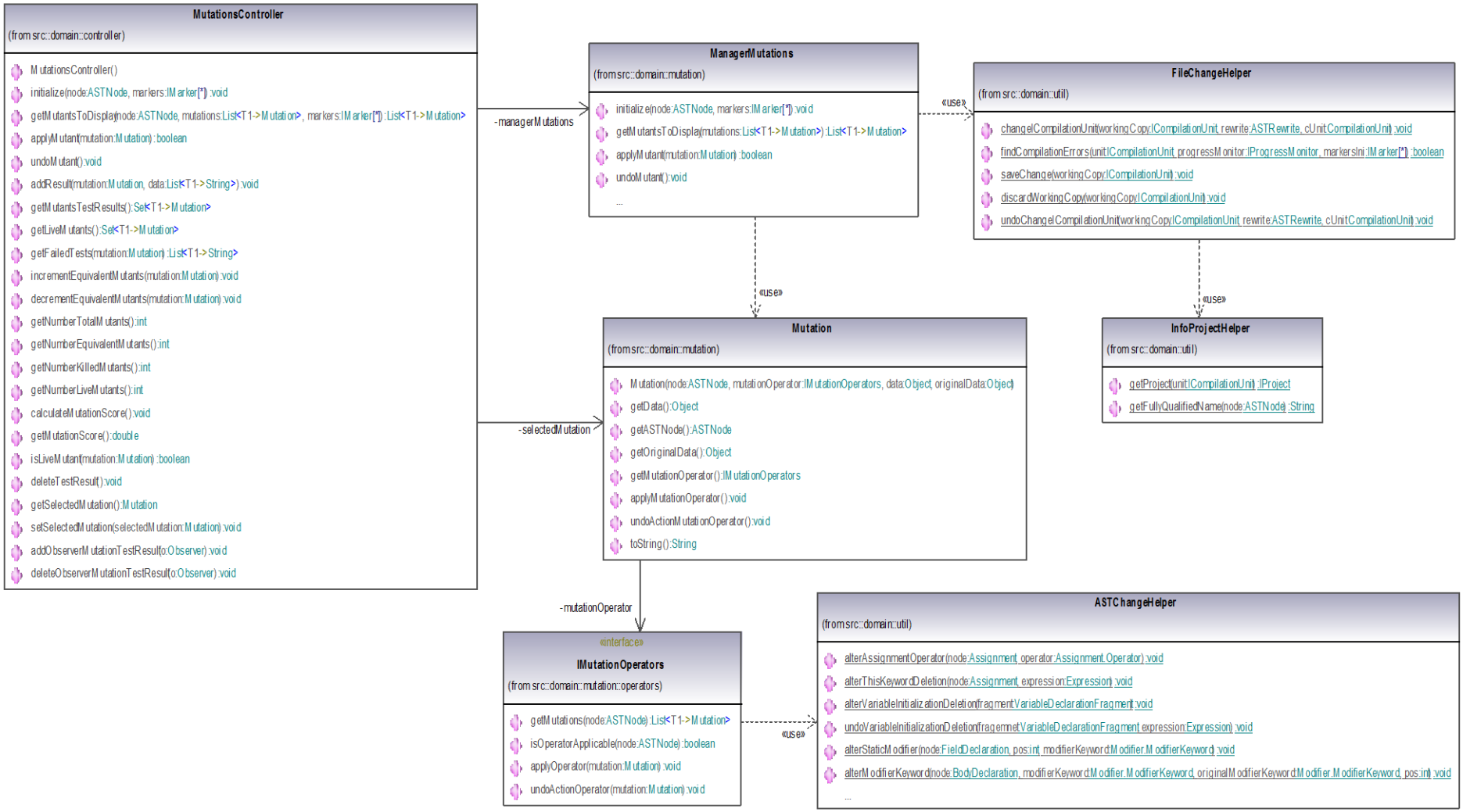

Figura 4.14 Diagrama de Classes UML para o processo de geração de um mutante

4.2.4.4 **Processo de execução de testes sobre os mutantes**

Os testes de mutação são uma técnica, cujo objetivo é avaliar a adequação (qualidade) dos casos de teste utilizados no processo de teste de um sistema. Assim, a informação necessária para avaliação dos casos de testes, resulta da execução destes sobre o programa e os mutantes gerados.

Para a concretização do processo de execução de testes sobre os mutantes no *plug-in* PESTTMuTest, foi utilizado o *plug-in* JUnit disponível na ferramenta JDT. A classe `org.junit.runner.JUnitCore`, disponível na biblioteca do JUnit, fornece um método (`run`) que permite executar os testes definidos para avaliar um determinado projeto Java. E este método recebe como parâmetro o ficheiro .class da classe de teste, e com esta informação o objeto responsável por realização do teste irá carregar a classe de teste a partir do *classloader*. O *classloader* é uma parte do ambiente de execução Java (JRE) que carrega de forma dinâmica as classes Java para a máquina virtual (JVM). No entanto, na plataforma Eclipse existe uma separação entre o mundo do *plug-in* e o mundo do *workspace* que contém os projetos do utilizador, e cada um destes mundos tem o seu *classloader*, o que significa que não seria possível ao *plug-in* PESTTMuTest carregar os ficheiros .class das classes de teste de um projeto Java existente no *workspace*, uma vez que o PESTTMuTest não contém esta informação no seu *classloader*. Assim, para contornar este problema, foi utilizado o conceito de reflexão/introspeção, que consiste em um programa aceder a sua própria estrutura. Deste modo, sempre que for necessário utilizar o JUnit para executar um conjunto de classes de teste é criado um novo *classloader* para o *plug-in,* que consiste na junção do *classloader* do próprio *plug-in* e do projeto ao qual se pretende obter as classes de teste. No entanto, para obter o ficheiro .class de uma classe de teste a partir do novo *classloader* criado é necessário a sua identificação. Esta informação é obtida durante o processo de análise do projeto (ver secção 4.2.4.2), onde é guardado o nome (*fully qualified name*), de todas as classes de teste identificadas durante a análise de um projeto. O *fully qualified name* permite a identificação dos ficheiros das classes de teste, sem que haja problemas de ambiguidade.

Uma classe de teste contém um ou mais métodos, e cada um destes métodos representa um caso de teste. Assim, um mutante é considerado "morto" se existir pelo menos um caso de testes que falhe, isto significa que no processo de execução de um determinado caso de teste sobre o mutante, o resultado esperado é diferente do resultado obtido. Para verificar se um mutante foi morto durante a execução de um caso de teste sobre o mutante foi necessário definir uma classe para tratar os eventos gerados durante o processo de execução do teste para obter a informação necessária para a realização de outras operações disponíveis no *plug-in* PESTTMuTest. Assim, foi criada uma subclasse de `org.junit.runner.notification.RunListener`

(`JUnitTestRunListener`) onde foi redefinido o método `testFailure`. A redefinição deste método irá permitir, identificar os casos de teste que "mataram" o mutante, isto porque o evento gerado devido a falha de um caso de teste é tratado por este método, e esta redefinição consiste em guardar a informação sobre o caso de teste que "matou" o mutante. Por outro lado, um mutante será considerado "vivo", se o conjunto dos casos de teste que falharam estiver vazio. Estas informações serão agregadas pelo objeto do tipo `MutationTestResult`, que representa o resultado do processo de testes de mutação, pois é através deste objeto que é obtido, por exemplo, o número dos mutantes mortos, quais os mutantes vivos, a pontuação dos testes de mutação, etc.

Para a concretização deste processo foram definidas as seguintes classes descritas na Figura 4.15 que apresenta o diagrama de classes UML.

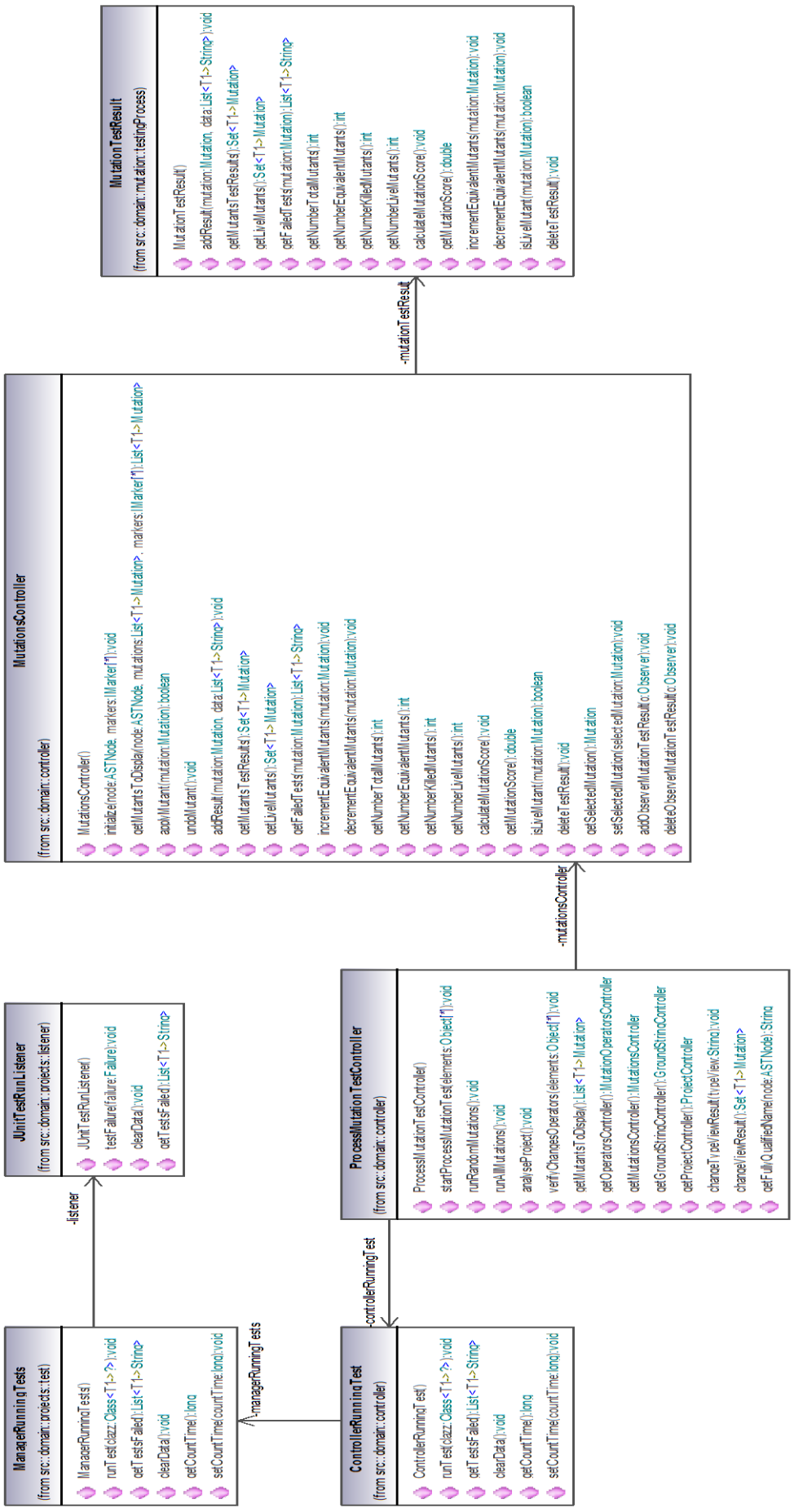

Figura 4.15 Diagrama de Classes UML para o processo de execução dos testes

4.2.4.5 **Interface com o utilizador**

Nas seções anteriores foram apresentadas as operações realizadas pelo *plug-in* em resposta a interação do utilizador com os componentes gráficos do PESTTMuTest. Neste tópico, será feita a descrição dos elementos gráficos utilizados e as suas relações com outros elementos do *plug-in*.

Assim, os elementos gráficos que compõem o *plug-in* são os seguintes:

- A perspetiva do *plug-in* chamada PESTTMuTest;
- As vistas
    - Mutation Operators;
    - Mutations;
    - Mutation Analysis;
    - E outras vistas disponíveis na plataforma Eclipse.
- O editor do Eclipse;
- Um conjunto de ícones que representam as ações definidas para algumas das vistas criadas.

A Figura 4.16 apresenta o aspeto visual do *plug-in*.

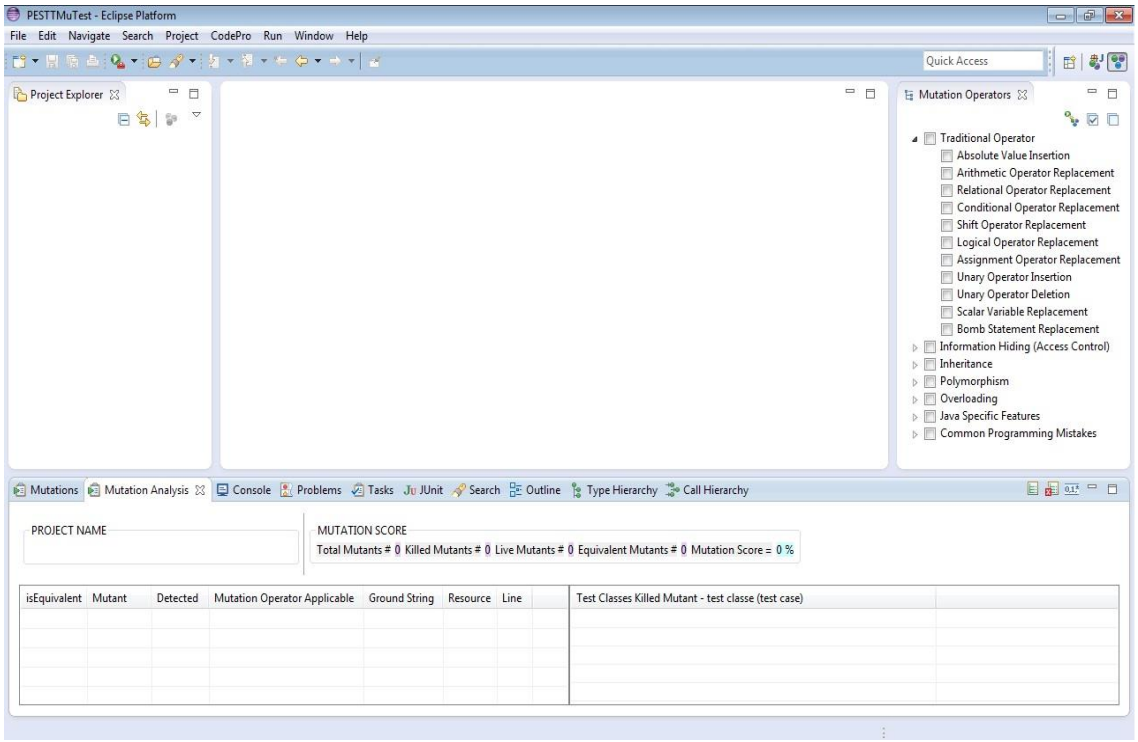

Figura 4.16 *Plug-in* PESTTMuTest

Para definir os elementos de interface do utilizador (como perspetiva, vista e ícones), criados especificamente para o *plug-in*, foram utilizadas extensões (ver secção 4.2.2). A Figura 4.17 apresenta um excerto do ficheiro `plugin.xml` com a informação sobre as extensões definidas.

```
org.eclipse.ui.perspectives
org.eclipse.ui.views
org.eclipse.ui.commands
org.eclipse.ui.menus
org.eclipse.ui.handlers
perspectiveExtensions
```

Figura 4.17 Lista das extensões do PESTTMuTest

Em seguida será apresentada uma descrição detalhada sobre a perspetiva PESTTMuTest e as vistas Mutation Operators, Mutation e Mutation Analysis, pois estes são os principais elementos UI do *plug-in*.

Perspetiva PESTTMuTest

A decisão por criar uma perspetiva, teve como fundamento, permitir a definição e a disposição dos elementos gráficos, de modo a que estes se enquadrem no contexto da realização de testes de mutação. Assim, a Figura 4.16 permite mostrar como foram definidas as posições e dimensões dos elementos UI (vistas e editor).

Para abrir a perspetiva PESTTMuTest (*Window → Open Prespective → Other… → PESTTMuTest*).

Vista Mutation Operators

Esta vista apresenta a informação sobre os operadores de mutação que podem ser escolhidos pelo utilizador, para que sejam utilizados no processo de testes de mutação.

Visto que, os operados de mutação podem ser organizados por grupos, optou-se por dispô-los usando uma estrutura em forma de árvore (ver Figura 4.12). Assim, para apresentar a informação sobre o grupo de operadores de mutação e os operadores que o compõem, foi definida uma classe, onde cada um dos seus objetos tem a informação sobre o seu ascendente e o conjunto dos seus descendentes. Convém salientar que os objetos desta classe não representam a concretização dos operadores de mutação, mas sim a informação necessária para permitir que o *plug-in* possa determinar quais os objetos dos operadores de mutação que devem ser criados (ver secção 4.2.4.1). Para criar cada subárvore que permite representar um grupo de operadores de mutação (incluindo os operadores que o compõem), foi utilizado o padrão Fábrica. No entanto,

para poder ligar todas as subárvores, foi definida uma árvore que representa a raiz, onde os seus descendentes são todas as subárvores que descrevem os grupos de operadores de mutação. Para apresentar esta informação ao utilizador, foi utilizada a interface gráfica `org.eclipse.jface.viewers.CheckboxTreeViewer`, disponível na biblioteca SWT/JFace.

Nesta vista foram definidos ícones que permitem representar as seguintes operações:

- Iniciar o processo de testes de mutação  ;
- Selecionar todos os elementos da árvore  ;
- Desfazer a seleção de todos os elementos da árvore  .

Esta vista tem um carater muito importante para o *plug-in*, uma vez que é a partir dela que o utilizador dará início ao processo de testes de mutação. Para isso, é necessário que o utilizador selecione a informação sobre os operadores de mutação presente nesta vista (ver Figura 4.16) e dê início ao processo, através da ação sobre o ícone  . Consequentemente, com base nos elementos selecionados na árvore presente, serão criados os objetos do tipo `IMutationOperators` (ver secção 4.2.4.1), que representam a concretização dos operadores de mutação. Serão também criadas as cópias dos projetos Java (ver secção 4.2.4.2) disponíveis no *workspace*, cuja informação será apresentada na vista `Mutations`, que será analisada no ponto seguinte.

Vista Mutations

A informação apresentada nesta vista, dá a possibilidade ao utilizador de realizar operações que permitam a visualização das mutações que podem ser aplicadas ao projeto Java, de acordo com os operadores de mutação selecionados na vista `Mutation Operators`, sem que seja necessário a conclusão do processo de testes de mutação (que incluiria a geração dos mutantes e a execução dos casos de teste).

Para a concretização das três tabelas que compõem a vista `Mutations` foi utilizada a classe `org.eclipse.jface.viewers.TableViewer` da biblioteca SWT/JFace. A informação em cada uma das tabelas definidas, é atualizada com base no padrão Observador, pois pretende-se que esta informação seja alterada consoante as modificações feitas sobre algumas classes do domínio, em resposta as ações realizadas pelo utilizador resultantes da sua interação com o *plug-in*.

A Figura 4.18 apresenta o resultado da interação do utilizador com alguns elementos da vista `Mutations`. As zonas assinaladas a roxo representam os elementos selecionados, e a zona assinalada a azul permite mostrar a linha que corresponde a expressão selecionada na tabela2.

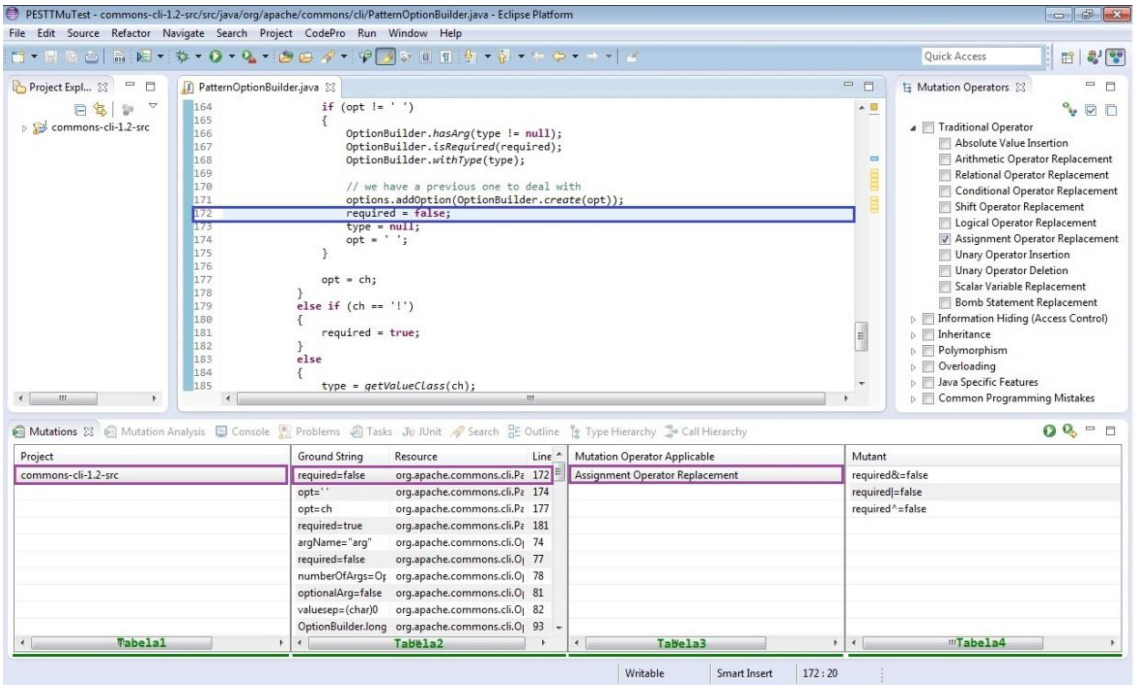

Figura 4.18 Operações do componente vista Mutations

A tabela1 (ver Figura 4.18) irá apresentar o nome dos projetos aptos para aplicação de mutações (ver secção 4.2.4.2). O nome apresentado será o nome dos projetos Java visíveis no *workspace*, no entanto, conforme referido na secção 4.2.4.2, as operações serão realizadas sobre as cópias criadas.

A tabela2 (ver Figura 4.18) está reservada para apresentar a informação sobre as *ground string* de um projeto Java selecionado na tabela1, onde podem ser aplicadas mutações, com base nos operadores de mutação selecionados pelo utilizador na vista `Mutation Operators`. Assim, quando o utilizador selecionar o nome de um projeto apresentado na tabela1, é iniciado o processo de análise do projeto, descrito na seção 4.2.4.2, que permite identificar as expressões do ficheiro onde podem ser aplicadas as mutações. Deste modo, a informação obtida será carregada para esta tabela. Uma vez que, podem em um projeto podem existir expressões idênticas, optou-se por incluir outra coluna (para além da coluna com a informação sobre a *ground string*) que contém o nome da classe a que a expressão pertence.

Para mostra a lista dos operadores de mutação que podem ser aplicados a uma determinada *ground string*, selecionada pelo utilizador na tabela2, foi definida a tabela3 (ver Figura 4.18).

A última tabela (ver Figura 4.18) apresenta o resultado dos mutantes gerados da aplicação de um operador de mutação (selecionado na terceira tabela), sobre uma expressão, presente na segunda tabela.

A Figura 4.20 permite ilustrar o diagrama de sequência UML, mostrando as operações efetuadas para apresentar as mutações de um determinado operador de mutação aplicado a *ground string* selecionada.

As outras formas de interação disponíveis na vista `Mutations` são as ações representadas pelos seguintes ícones:

- Executar os testes sobre todos os mutantes gerados . Para a concretização desta operação, é realizado o processo descrito na secção 4.2.4.3, e também o processo referente a execução dos casos de testes sobre os mutantes gerados (ver seção 4.2.4.4).
- Executar os testes sobre um mutante aleatório . Para cada operador de mutação de pode ser aplicado a uma determinada *ground string* é selecionada de forma aleatória uma mutação válida dentro do conjunto das mutações. Para a concretização desta operação serão gerados todos os mutantes escolhido, conforme descrito na seção 4.2.4.3 para que seja executo o processo descrito na seção 4.2.4.4.

Vista Mutation Analysis

Para apresentar o resultado do processo de testes de mutação, desencadeado pela ação sobre os ícones ou referenciados no tópico sobre a vista `Mutations`, foi definida a vista `Mutation Analysis`. Esta vista foi dividida em três partes (ver Figura 4.19). A parte1 irá conter o nome do projeto sobre o qual foram aplicadas mutações para a execução do conjunto de casos de teste. A parte2 está reservada para apresentar a informação sobre os mutantes que foram gerados e executados sobre os casos de teste existentes, assim para apresentar a respetiva informação foram definidas duas tabelas. A primeira tabela terá a informação sobre os mutantes. A primeira coluna (`isEquivalent`) desta tabela permite ao utilizador assinalar os mutantes equivalentes, dentro do conjunto dos mutantes vivos, isto terá como base o conhecimento do utilizador para a identificação de mutantes equivalentes. Esta informação será útil para determinar com maior exatidão a pontuação do processo de teste de mutação. É também através da primeira tabela que o utilizador poderá verificar se um mutante foi morto ou não, esta informação estará presente na terceira coluna (`Detected`), onde a imagem ✗ significa que nenhum dos casos de teste "matou" o mutante, e a imagem ✓ indica que houve pelo menos um caso de testes que conseguiu identificar o mutante como sendo uma falha. A segunda tabela terá a informação sobre

os casos de teste que foram bem-sucedidos, ou seja, que mataram o "mutante" selecionado na primeira tabela. Por fim, a parte3 terá informação em termos numéricos sobre o resultado do processo de testes de mutação. O padrão Observador, também foi utilizado nesta vista para carregar a informação sobre os elementos que compõem as três partes definidas para esta vista.

As ações representadas por estes ícones são as seguintes:

- Mostrar todos os mutantes . Esta operação, tem sentido, se anteriormente o utilizador tiver utilizado o filtro para obter todos os mutantes que não foram "mortos" por nenhum dos casos de teste, designados também como mutantes "vivos", caso contrário a informação da primeira tabela não será alterada;
- Filtrar os mutantes "vivos" . Esta operação permite ao utilizador ver todos os mutantes que não foram "mortos".
- Calcular a pontuação da mutação . Esta operação atualiza a informação da terceira parte, com base nos resultados obtidos da execução dos testes sobre os mutantes gerados, informando o utilizador sobre algumas variáveis que poderão ser úteis para análise, como por exemplo, o número de mutantes "mortos", o número total de mutantes gerados, a pontuação da dos testes de mutação, etc.

Este capítulo apresentou o desenho da solução para a concretização do *plug-in* PESTTMuTest. Também foram descritos os detalhes de implementação do *plug-in*, incluindo a definição dos processos que permitem a concretização dos testes de mutação no *plug-in*, a descrição e função de cada um dos elementos da UI.

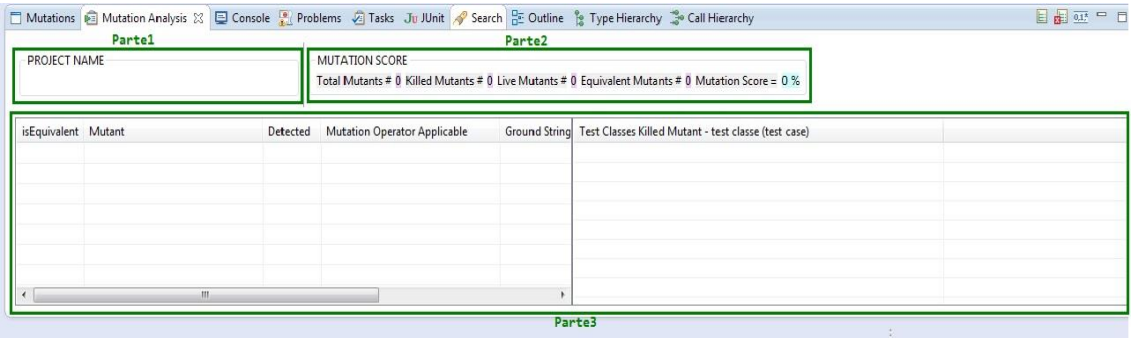

Figura 4.19 Componente UI vista Mutation Analysis

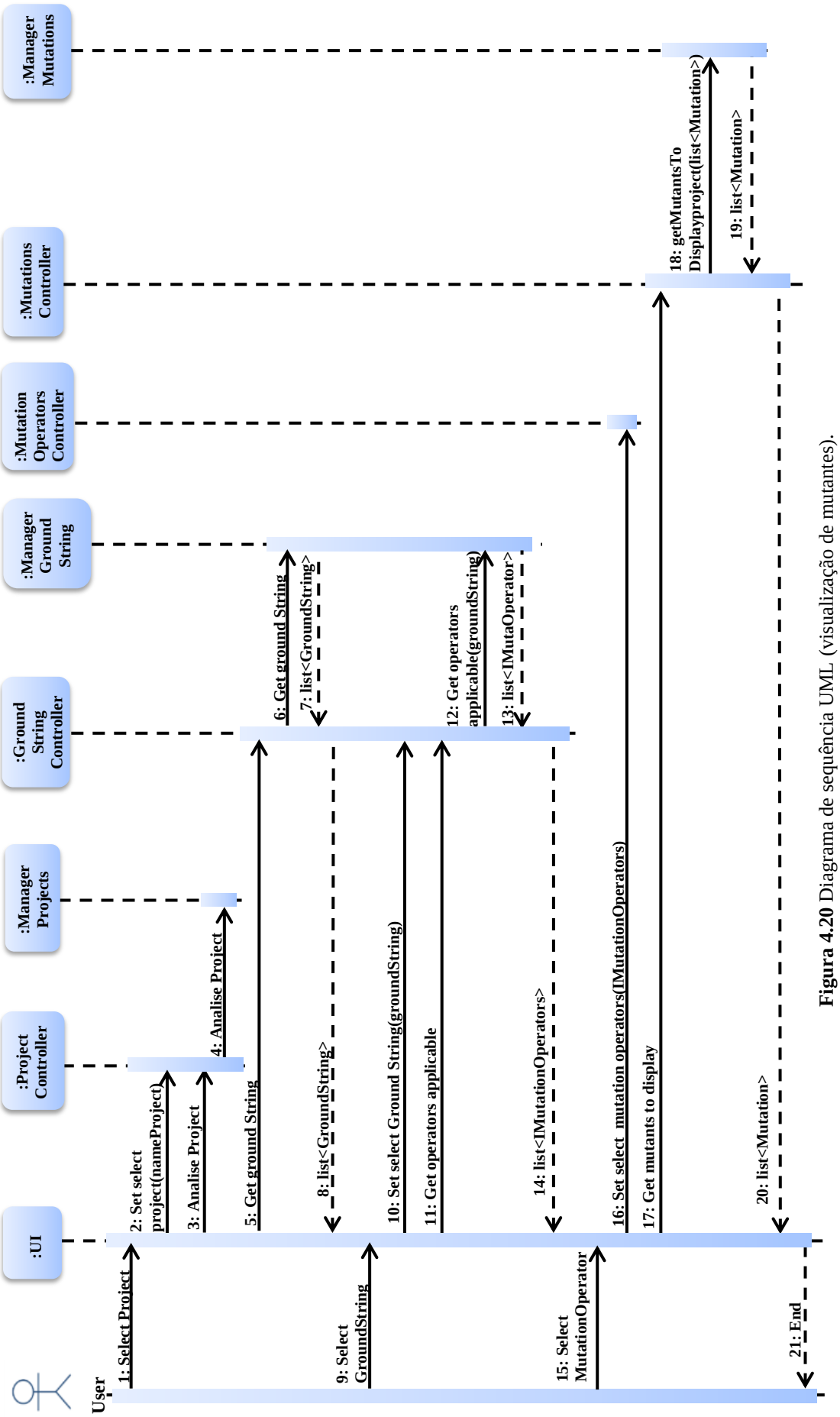

Figura 4.20 Diagrama de sequência UML (visualização de mutantes).

Capítulo 5

Avaliação

Devido ao elevado custo computacional que a técnica de testes de mutação pode ter, é necessário avaliar o desempenho da ferramenta desenvolvida para permitir aos utilizadores aplicarem esta técnica.

Assim, com base em métodos estatísticos serão analisados os dados obtidos na experimentação que será realizada para avaliar o desempenho do *plug-in* PESTTMuTest. Será também utilizada a informação sobre experiências [12] realizadas por outros autores, nas ferramentas MuJava e Judy (Capítulo 2), pois embora o objetivo deste trabalho, não seja o desenvolvimento de uma ferramenta que supere estas ferramentas em termos de desempenho da aplicação dos testes de mutação, visto que o *plug-in* PESTTMuTest foi desenvolvido para ser utilizado em um ambiente académico, esta informação permitirá situar o nível de desempenho da ferramenta desenvolvida através da comparação dos resultados obtidos em ambas as experiencias realizadas. No entanto, convém referir que, o fato de existirem diferenças, por exemplo, o ambiente de experimentação, entre a experimentação realizada para as ferramentas MuJava e Judy e a experimentação que será feita para o *plug-in* desenvolvido, terá influência sobre o resultado da comparação destas experimentações.

Neste capítulo, serão apresentados os objetivos definidos na experimentação a ser concretizada, a descrição da experimentação, os resultados obtidos e a análise feitas sobre os resultados obtidos.

5.1 Descrição da experimentação

O objetivo desta experimentação é avaliar o desempenho do *plug-in* PESTTMuTest, e deste modo analisar o seu impacto no processo de testes de mutação aplicado a uma aplicação Java.

No entanto, por este processo envolver um conjunto de processos, no qual tem fundamento avaliar o impacto de cada um deles sobre o processo global. Assim, será avaliado no *plug-in*:

1. O desempenho do processo de geração de mutantes;
2. Calcular o custo de gerar e "matar" um mutante.

A avaliação 1 do desempenho do processo de geração de mutantes é definida pelo número de mutantes gerados por segundo (NMPS). O resultado que será obtido desta avaliação terá grande influência sobre o resultado da avaliação do desempenho do PESTTMuTest, porque este processo representa o núcleo do processo de testes de mutação.

O processo de geração de mutantes inclui vários passos, no qual o seu tempo deve ser contabilizado para determinar NMPS, isto porque a geração de um mutante não envolve apenas a operação não é apenas o passo de alterar o código fonte de um determinado ficheiro Java. Assim, para calcular o NMPS serão incluídos os tempos de análise do projeto para determinar as *ground strings*, onde poderão ser aplicadas as mutações, o tempo de percorrer cada uma das *ground strings* e gerar os mutantes de cada um dos operadores de mutação disponíveis, tempo de alteração do ficheiro java e o tempo de validação das alterações efetuadas. Sendo o último tempo fundamental para garantir a geração de mutantes válidos, será analisado com mais detalhe o seu impacto sobre o processo de geração de mutantes.

Deste modo, para a avaliação 1 foram definidas duas fases de teste:

1.1 Na primeira fase, serão incluídos todos os mutantes que um operador de mutação pode gerar. Para esta fase foram definidos dois testes descritos na alinha a) e na alinha b), cuja comparação dos resultados obtidos permitirá analisar o custo que a geração de mutantes válidos representa para o desempenho do *plug-in* PESTTMuTest.

   a) Calcular o número de mutantes gerados por segundo (NMPS), incluindo o tempo de geração e de compilação. Como referido na secção 4.2.4.3, a fase de compilação do projeto é fundamental para garantir que serão gerados apenas mutantes válidos, assim este teste permitirá determinar quantos mutantes válidos o *plug-in* consegue gerar por segundo;

   b) Calcular o número de mutantes gerados por segundo (NMPS), sem que seja incluída a fase de verificação da geração de mutantes válidos. Neste teste não será considerado o tempo de compilação e dentro do conjunto de mutantes gerados poderão existir mutantes inválidos.

1.2 A segunda fase consiste na realização dos testes descritos para a primeira fase, no entanto estes serão realizados para um mutante aleatório para cada

operador de mutação, considerando que para o teste apresentado na alinha a) os mutantes aleatórios gerados serão apenas mutantes válidos, enquanto o teste descrito na alinha b) no conjunto dos mutantes aleatórios gerados poderão existir mutantes inválidos.

A avaliação 2 qual é o custo de gerar e "matar" um mutante, ou seja, o tempo necessário para gerar um mutante válido e executar todos os casos de teste para determinar se existe um caso de teste capaz de identificar o mutante como sendo uma falha.

Deste modo foi definido o seguinte teste:

c) Calcular o tempo médio para gerar mutantes válidos, calcular o tempo médio referente a execução dos casos de teste sobre os mutantes gerados e dividir a soma destes tempos pelo número de mutantes. Sobre o valor obtido será retirado o tempo da execução dos testes sobre o projeto

Com o objetivo de aumentar a credibilidade da avaliação a ser realizada, serão utilizados projetos Java do mundo real de código aberto do Apache Software Foundation (ASF). Outra vantagem em utilizar estes projetos é que estes já contêm testes escritos para serem executados pelo JUnit. Assim, para a realização dos testes descritos nas alíneas a), b) e c) foram utilizados cinco projetos Java, que encontram-se descritos na secção 5.2 . Foram também utilizados os operadores de mutação descritos na Tabela 4.1.

O equipamento utilizado para a execução das experiências contém as seguintes especificações:

- Sistema Operativo: Windows® 8, 64-bit;
- Processador e frequência: Intel® Core™ i7, 1.6 GHz, 2.7GHz (frequência turbo);
- Memória RAM: 8 GB, DDR3;

5.2 Resultados das experiências

Esta secção destina-se a apresentar os resultados obtidos durante a realização dos testes descritos nas alíneas a), b) e c).

5.2.1 Primeira etapa de testes

Para apresentar os resultados experimentais da avaliação 1 descrita na secção 5.1 , foram definidas duas variáveis AM e VM. A variável AM representa o número total de

mutantes (válidos e inválidos), enquanto a variável VM corresponde somente ao número de mutantes válidos.

Durante as fases 1.1 e 1.2 associadas a avaliação 1 foram executados vinte vezes os testes a) e b) para cada uma das aplicações. Em cada iteração calculou-se o valor médio de NMPS gerados eliminando-se o melhor e o pior valor.

Os resultados obtidos durante a fase 1.1 nos testes a) e b) estão dispostos na Tabela 5.1.

Projeto	LOC	AM		VM	
		Total de Mutantes	NMPSPS	Total de Mutantes	NMPS
Apache Jakarta Commons CLI	4534	1427	40,22	1036	12,14
Apache Jakarta Commons Chain	8231	2556	68,46	1092	9,35
Apache Jakarta Commons FileUpload	4356	1937	28,75	1509	11,58
Apache Jakarta Commons Logging	5406	1469	33,31	669	7,21
Apache Jakarta Commons Validator	11948	4456	51,03	3118	13,61

Tabela 5.1 Resultados da fase 1.1 referentes aos testes a) e b)

Os resultados obtidos durante a fase 1.2, nos testes a) e b) e estão dispostos na Tabela 5.2.

Projeto	LOC	AM		VM	
		Total de Mutantes	NMPSPS	Total de Mutantes	NMPS
Apache Jakarta Commons CLI	4534	437	28,05	383	8,17
Apache Jakarta Commons Chain	8231	961	50,31	613	7,11
Apache Jakarta	4356	591	17,76	450	6,63

Commons FileUpload					
Apache Jakarta Commons Logging	5406	573	23,14	312	4,72
Apache Jakarta Commons Validator	11948	1581	38,29	1361	10,61

Tabela 5.2 Resultados da fase 1.2 referentes aos testes a) e b)

5.2.2 Segunda etapa de testes

Para apresentar os resultados experimentais da avaliação 2 descrita na secção 5.1 , foi definida uma variável TGRPM (Tempo de geração e tempo de execução por mutante). Para o cálculo do valor da variável TGRPM para cada um dos projetos foi contabilizado o tempo de geração de mutantes válidos. A este tempo foi somando o tempo de análise de cada mutante gerado, ou seja, o tempo de execução dos casos de testes sobre o mutante para verificar se existe algum caso de teste que consegue identificar o mutante como sendo uma falha.

A Tabela 5.3 apresenta os resultados obtidos nesta etapa.

Projeto	Número de casos de teste	LOC	Tempo[6] de execução do JUnit	Todos os mutantes			Mutantes aleatórios		
				Total de mutantes	TGRPM	Custo	Total de mutantes	TGRPM	Custo
Apache Jakarta Commons CLI	187	4534	0,05	1036	2,73	2,68	383	0,91	0,86
Apache Jakarta Commons Chain	116	8231	0,33	1092	0,32	-0,01	613	0,29	-0,04
Apache Jakarta Commons	71	4356	4,09	1509	2,79	-1,3	450	1,82	-2,27

[6] Este tempo representa o tempo médio em segundos da execução dos testes sobre o projeto

FileUpload									
Apache Jakarta Commons Logging	61	5406	5,82	669	6,23	0,41	312	6,02	0,2

Tabela 5.3 Resultados dos testes c)

5.3 Análise dos resultados

Os dados experimentais obtidos na primeira fase de testes foram analisados com base na estatística descritiva. Os valores para análise obtidos para NMPS estão descritos na Tabela 5.4. Os *blox plot* presentes nas Figura 5.1 e Figura 5.2 apresentam a distribuição dos resultados. Através de uma análise visual sobre estes gráficos, pode-se verificar a discrepância entre os resultados obtidos para a geração de mutantes válidos e a geração de todos os mutantes (válidos e inválidos). Os resultados apresentados nesta tabela permitem analisar o impacto que gerar apenas mutantes válidos tem sobre o desempenho da ferramenta. Tanto no caso em que são gerados mutantes aleatórios ou todos os mutantes podemos verificar que há uma perda no desempenho de 75%. Por outro lado, temos o ganho em garantir que serão apenas apresentados aos utilizadores a informação sobre os mutantes válidos gerados no processo de teste. Isto é fundamental, pois a nível académico a informação passada aos alunos deve ser coerente.

Descrição	Variável	Desvio padrão (SD)	Média (M)	Max	Mediana	Min
Mutante Aleatório	NMPS(VM)	1,94	7,45	10,61	7,11	4,72
	NMPS(AM)	11,58	31,51	50,31	28,05	17,76
Todos os mutantes	NMPS(VM)	2,25	10,78	13,61	11,58	7,21
	NMPS(AM)	14,20	44,35	68,46	40,22	28,75

Tabela 5.4 Resultados estatísticos para o NMPS

Para comprar o desempenho (em relação ao NMPS) do PESTTMuTest com as ferramentas MuJava e Judy foi utilizada a informação presenta no artigo (12). A Tabela 5.5 apresenta o NMPS calculado por cada uma das ferramentas.

Podemos ver que os resultados que a ferramenta Judy apresenta, superam os resultados das outras ferramentas. No entanto, considerando que NMPS calculado por Judy e MuJava podem incluir mutantes válidos e inválidos, e comparando com NMPS

(AM) do PESTTMuTest verifica-se que a diferença de resultados em relação à Judy é pequena, e supera os resultados do MuJava, assim como o NMPS (VM).

Contudo, não é possível concluir que em termos de desempenho o PESTTMuTest é melhor ou pior que as outras duas ferramentas pelos seguintes motivos: os operadores de mutação utilizados nas experiências eram diferentes (em termos de quantidade e tipo); a informação do artigo não é suficiente, por exemplo, não é referido o número total de mutantes gerados, ou a descrição do *hardware* utilizado.

Projeto	**NMPS**			
	Todos os mutantes		**MuJava**	**Judy**
	PESTTMuTest (AM)	**PESTTMuTest (VM)**		
Apache Jakarta Commons CLI	40,22	12,14	5,77	52,45
Apache Jakarta Commons Chain	68,46	9,35	2,06	42,47
Apache Jakarta Commons FileUpload	28,75	11,58	5,52	38,78
Apache Jakarta Commons Logging	33,31	7,21	3,04	33,44
Apache Jakarta Commons Validator	51,03	13,61	5,46	82,33

Tabela 5.5 Comparação do NMPS calculado nas ferramentas PESTTMuTest, MuJava e Judy

Da análise feita sobre os resultados obtidos na segunda etapa de testes podemos dizer que a ferramenta PESTTMuTest apresenta um custo de gerar e "matar" um mutante reduzido. Em alguns dos casos o valor da variável TGRPM é muito próximo ou inferior ao tempo do pior caso em que nenhum dos caso de testes consegue identificar o mutante como sendo uma falha, e por isso são executados todos os casos de teste. Por este motivo, alguns dos valores referentes a variável Custo, presente na Tabela 5.3 apresenta valores negativos.

Analisando em conjunto os resultados obtidos em todos os testes, podemos afirmar que esta ferramenta apresenta um bom desempenho para a realização de testes de mutação.

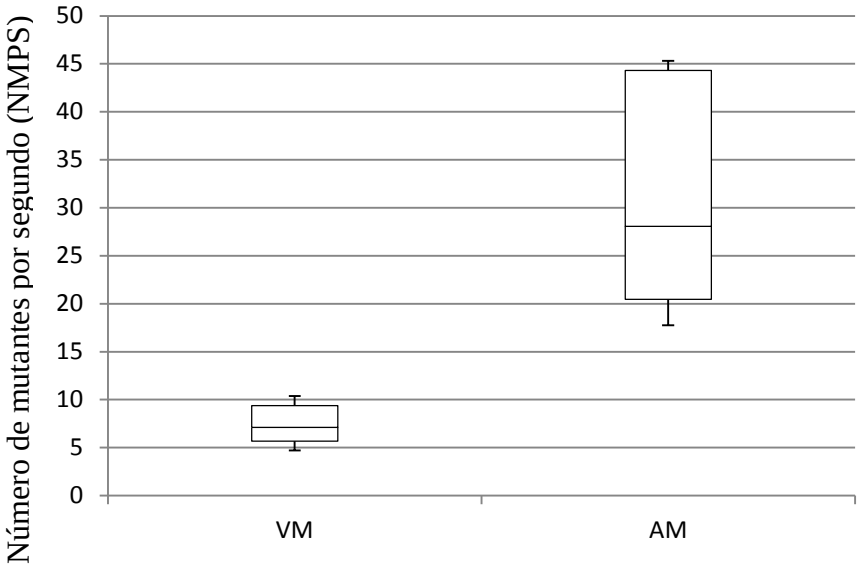

Figura 5.1 Box plot, NMPS para mutantes aleatórios

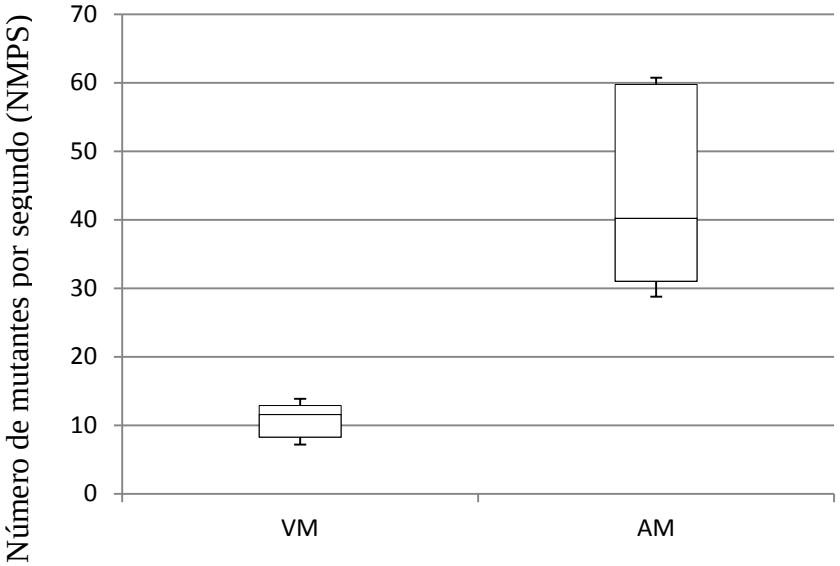

Figura 5.2 Blox plot, NMPS para todos os mutantes

Capítulo 6

Conclusão

Este projeto apresentou o PESTTMuTest, uma ferramenta para a concretização de testes de mutação em aplicações Java para ser utilizada num ambiente académico, com o objetivo de apoiar o ensino desta técnica de testes. Os testes de mutação são uma técnica utilizada para testar a robustez de um conjunto de casos de teste, que embora através de vários trabalhos tenha-se comprovado a sua eficácia, esta produz muitos requisitos de teste, que neste modelo de testes se chamam de *mutante*. O número de mutantes é de tal ordem extenso que não pode ser sistematicamente tratado de forma manual, o que implica a necessidade de uma ferramenta pra que seja possível a aplicação desta técnica.

Existem atualmente diversas ferramentas que realizam testes de mutação, mas cujo foco é o aumento de desempenho, ou seja, a geração e execução de testes de forma rápida. O nosso objetivo é diferente, pois pretendemos construir uma ferramenta capaz de ser utilizada no ensino, no entanto com um desempenho aceitável para a sua utilização. Para tal, foi desenvolvida uma ferramenta em forma de *plug-in* para ser integrada no ambiente de desenvolvimento com grande aceitação a nível académico: o Eclipse, e criado um ambiente interativo no qual o engenheiro de teste (o aluno ou professor) possa realizar várias operações, como por exemplo escolher os operadores de mutação que pretende utilizar, o projeto Java onde pretende que sejam aplicadas as mutações e outras funcionalidades que permitem ao utilizador acompanhar as várias fases do processo de teste, desde a geração dos mutantes e a sua visualização, até apresentação dos resultados da conclusão do processo de testes de mutação, como por exemplo visualizar os mutantes foram mortos ou saber quais os casos de teste que "mataram " o mutante.

Os testes de sistema realizados permitiram verificar que esta ferramenta tem um baixo custa na geração e análise de mutantes. Também foi possível verificar que apesar de haver uma redução no desempenho do PESTTMuTest devido ao requisito em garantir a geração de mutantes válidos, a comparação com as ferramentas MuJava e

Judy, embora não tenha sido possível formar uma conclusão desta comparação, foi possível verificar que os valores de PESTTMuTest são superiores aos valores obtidos para MuJava, e se considerarmos a geração de todos os mutantes (válidos e inválidos) os resultados estão próximos dos resultados obtidos para a ferramenta Judy.

A ferramenta revelou-se de grande utilidade e eficácia, acompanhando o engenheiro de testes em todas as etapas dos testes de mutação. É minha convicção que será de grande utilidade no ensino de testes de mutação.

Como trabalho futuro imediato, pretendo complementar o conjunto de operadores de mutação disponibilizados pela ferramenta, e fazer um teste "real" com alunos de uma disciplina de testes e averiguar a aceitação da ferramenta.

Abreviaturas

AM – All Mutants

API – Application Programming Interface

AST – Abstract Syntax Tree

ECJ – Eclipse Compiler for Java

FOM – First Order Mutants

GUI – Graphical User Interface

HOM – Higher Order Mutation

IDE – Integrated Development Environment

IEEE – Institute of Electrical and Electronics Engineers

JDT – Java Development Tools

JRE – Java Runtime Environment

JVM – Java Virtual Machine

LOC – lines of code

MIMD – Multiple instructions, multiple data

MSG – Mutant Schemata Generation

MVC – Model-View-Controller

NMPS – Number of Mutants Generated per Second

OO – Object Oriented

OSGi – Open System Gateway Initiative

PDE – Plug-in Development Environment

RIP – Reachability, Infection, Propagation

SDK – Software Development Kit

SIMD – Single instruction, multiple data

SWT – Standard Widget Toolkit

UI – User Interface

UML – Unified Modeling Language

VCM – Version Configuration Management

VM – Valid Mutant

VV&T – Verificação, Validação e Teste

Bibliografia

[1] **Pressman, R. S.** *Software Engineering: A Practitioner's Approach.* 7ª. New York : McGraw Hill, 2009.

[2] **Myers, G. J., et al., et al.** *The Art of Software Testinghn.* 2ª. Hoboken, New Jersey : John Wiley & Sons, Inc., 2004.

[3] *Hints on test data selection: Help for the practicing programmer.* **DeMillo, R. A., Lipton, R. J. and Sayward, F. G.** 4, April 1978, Software Engineering, IEEE Transactions on, Vol. 11, pp. 34 - 41.

[4] *Testing programs with the aid of a compiler.* **Hamlet, R. G.** 4, July 1977, Software Engineering, IEEE Transactions on, Vol. 3, pp. 279 - 290.

[5] *The Mothra Tool Set.* **Choi, B. J. et al.** Kailua-Kona, HI : s.n., 1989. Proceedings of the 22nd Hawaii International Conference on Systems and Software. Vol. 2, pp. 275 - 284.

[6] **DeMillo, R. A. et al.** *Mothra internal documentation, Version 1.0.* Software Engineering Rechearch Center, Georgia Institute of Technology. 1987. Technical report GIT-SERC-87/10.

[7] **Budd, T. and Sayward, F.** *Users guide to the Pilot mutation system.* Department of Computer Science, Yale University. 1977. Technical report 114.

[8] **Ammann, P. and Offutt, J.** *Introduction to software testing.* 1ª. s.l. : Cambridge University Press, 2008.

[9] *Estimation and enhancement of realtime software reliability through mutation analysis.* **Geist, R., Offutt, J. and Harris, F.** 5, May 1992, Computers, IEEE Transactions on, Vol. 41, pp. 550 - 558. Special issue on Fault-Tolerant Computing.

[10] *Investigations of the software testing coupling effect.* **Offutt, J.** 1, January 1992, ACM Transactions on Software Engineering Methodology, Vol. 1, pp. 5 - 20.

[11] *Fault coupling in finite bijective functions.* **How Tai Wah, K. S.** 1, March 1995, Software Testing, Verification, and Reliability, Vol. 5, pp. 3 - 47.

[12] *Judy - a mutation testing tool for java.* **Madeyski, L. and Radyk, N.** 1, July 2010, Software, IET, Vol. 4, pp. 32 - 42.

[13] *A Fortran 77 Interpreter for Mutation Analysis.* **Offutt, A. J. and King, K. N.** 7, July 1987, ACM SIGPLAN Notices, Vol. 22, pp. 177 - 188.

[14] *A Fortran Language System for Mutation- Based Software Testing.* **King, K. N. and Offutt, A. J.** 7, October 1991, Software:Practice and Experience, Vol. 21, pp. 685 - 718.

[15] **Bowser, J. H.** *Reference Manual for Ada Mutant Operators.* Georiga Institute of Technology. Atlanta, Georgia : s.n., 1988. Technique Report GITSERC-88/02.

[16] **Agrawal, H., et al., et al.** *Design of Mutant Operators for the C Programming Language.* Purdue University. West Lafayette, Indiana : s.n., 1989. Technique Report SERC-TR-41-P.

[17] **Kim, S., Clark, J. A. and McDermid, J. A.** *The Rigorous Generation of Java Mutation Operators Using HAZOP.* Department of Computer Science, University of York. Heslington, York : s.n., 1999. Technical Report, 2/8/99.

[18] *Inter-class Mutation Operators for Java.* **Ma, Y. -S., Kwon, Y. -R. and Offutt, A. J.** [ed.] IEEE Computer Society. Annapolis, Maryland : s.n., 2002. Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02). p. 352.

[19] *MuJava: An Automated Class Mutation System.* **Ma, Y.-S., Offutt, A. J. and Kwon, Y.-R.** 2, June 2005, Software Testing, Verification & Reliability, Vol. 15, pp. 97 - 133.

[20] *An experimental evaluation of data flow and mutation testing.* **Offutt, A. J., et al., et al.** 2, February 1996, Software - Practice & Experience, Vol. 26, pp. 165 - 176.

[21] *All-uses vs mutation testing: an experimental comparison of effectiveness.* **Frankl, P.G., Weiss, S.N. and HU, C.** 3, 1997, Journal of Systems and Software, Vol. 38, pp. 235 - 253.

[22] *An overview of the Mothra software testing environment.* **DeMillo, R. A. et al.** Banff, Alta : s.n., 1988. Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on. pp. 142 - 151.

[23] *Proteum – A tool for the assessment of test adequacy for C programs.* **Delamaro, M. E. and Maldonado, J. C.** New Brunswick, NJ : s.n., 1996. Conference on Performability in Computing Systems (PCS 96). pp. 79 - 95.

[24] *Jester – a JUnit test tester.* **Moore, I.** Italy : ed, 2001. Proc. Second Int. Conf. Extreme Programming and Flexible Processes in Software Engineering. pp. 84 - 87.

[25] *Mutation 2000: Uniting the Orthogonal.* **Offutt, J. and Untch, R. H.** San Jose, CA : s.n., 2000. Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries. pp. 45 - 55.

[26] *Mutation analysis using mutant schemata.* **Untch, R. H., Offutt, J. A. e Harrold, M. J.** 3, New York, NY : s.n., July de 1993, ACM SIGSOFT Software Engineering Notes, Vol. 18, pp. 139 - 148.

[27] *An empirical evaluation of the MuJava mutation operators.* **Smith, B. H. e Williams, L.** Windsor : s.n., 2007. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007. pp. 193 - 202.

[28] *Jumble Java Byte Code to Measure the Effectiveness of Unit Tests.* **Irvine, S. A., et al., et al.** Windsor : s.n., 2007. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007. pp. 169 - 175.

[29] *Program Testing by Specification Mutation.* **Budd, T. A. and Gopal, A. S.** 1, s.l. : Elsevier Ltd, 1985, Computer Languages, Vol. 10, pp. 63 - 73.

[30] **DeMillo, R. A.** *Program Mutation: An Approach to Software Testing.* Georgia Institute of Technology. Atlanta : s.n., 1983. p. 330, Technical report.

[31] **Okun, V.** *Specification Mutation for Test Generation and Analysis.* University of Maryland Baltimore County. Baltimore, Maryland : s.n., 2004. PhD Thesis.

[32] *Integration Testing Using Interface Mutation.* **Delamaro, M. E., Maldonado, J. C. e Mathur, A. P.** White Plains, New York : s.n., 1996. Proceedings of the seventh International Symposium on Software Reliability Engineering (ISSRE ’96).

[33] *Performing data flow testing on classes.* **Harrold, M. J. and Rothermel, G.** 5, December 1994, ACM SIGSOFT Software Engineering Notes, Vol. 19, pp. 154 - 163.

[34] **Gallagher, L. e Offutt, A. J.** *Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details.* Department of Information and Software Engineering, George Mason University. Fairfax, VA : s.n., 2004. GMU Technical Report ISE-TR-04-04.

[35] *A fault model for subtype inheritance and polymorphism.* **Offutt, J., et al., et al.** Hong Kong, China : s.n., 2001. Proceedings 12th International Symposium on Software Reliability Engineering. pp. 84 - 93.

[36] **Binder, R. V.** *Testing Object-oriented Systems: Models, Patterns, and Tools.* New York : Addison-Wesley, 2000.

[37] OSGi. *Open Services Gateway initiative framework.* [Online] [Cited: March 26, 2014.] http://www.osgi.org/Main/HomePage.

[38] Equinox. *Eclipse certified implementation of the OSGi R4.x framework specification.* [Online] http://www.eclipse.org/equinox/.

[39] MSDN. *The microsoft developer network.* [Online] [Citação: 26 de March de 2014.] http://msdn.microsoft.com/en-us/library/ff649643.aspx.

[40] Design Patterns & Refactoring. *Design patterns details.* [Online] [Cited: March 26, 2014.] http://sourcemaking.com/design_patterns.

[41] MuClipse. *Mutation Testing for Eclipse.* [Online] [Cited: March 25, 2014.] http://muclipse.sourceforge.net/mutants.php.

[42] *Help - Eclipse Platform.* [Online] [Cited: March 24, 2014.] http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm.

Anexos A

A.1 Interface Operador de Mutação

Este anexo apresenta o código fonte da interface definida para representar os métodos fundamentais que cada classe definida para concretizar um determinado operador de mutação de conter.

```
public interface IMutationOperators {
  public List<Mutation> getMutations(ASTNode node);
  public boolean isOperatorApplicable(ASTNode node);
  public void applyOperator(Mutation mutation);
  public void undoActionOperator(Mutation mutation);
}
```

A.2 Implementação da classe Mutação

Para a representação de uma mutação, gerada pela aplicação de um operador de mutação numa *ground string*, foi definida a classe `Mutation`. Esta classe tem um papel fundamental, pois será a partir da informação de cada uma das suas instâncias, que será concretizado um mutante.

```
public class Mutation {
  // Node where it will be applied to mutation
  private ASTNode node;
  // object that applied the mutation
  private IMutationOperators mutationOperator;
  // That mutation should be applied
  private Object data;
  // data before application operator mutation
  private Object originalData;
  private String mutant;
  public Mutation(ASTNode node, IMutationOperators
  mutationOperator, Object data, Object originalData) {
      this.node = node;
```

```
            this.mutationOperator = mutationOperator;
            this.data = data;
            this.originalData = originalData;
            mutant = "";
      }
      public Object getData() {
            return data;
      }
      public ASTNode getASTNode() {
            return node;
      }
      public Object getOriginalData() {
            return originalData;
      }
      public IMutationOperators getMutationOperator() {
            return mutationOperator;
      }
      public void applyMutationOperator() {
            mutationOperator.applyOperator(this);
            mutant = ToStringASTNode.toString(node);
      }
      public void undoActionMutationOperator() {
            mutationOperator.undoActionOperator(this);
      }
      @Override
      public String toString() {
            return mutant;
      }

   }
```