UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Plug-in Eclipse para geração automática de requisitos de teste

Ana Catarina da Eira Freire Pereira

Mestrado em Engenharia Informática
Engenharia de Software

Dissertação orientada por:
Francisco Cipriano da Cunha Martins
João Pedro Neto

2020

Resumo

Produzir *software* com qualidade que satisfaça os requisitos funcionais é o objetivo principal da Engenharia de *Software*. Para fazer face a este desafio concorrem diversos fatores, entre os quais, o teste das aplicações. É, portanto, primordial dominar as técnicas adequadas de teste de *software*.

O PESTT *Educational Software Testing Tool* (PESTT) é uma ferramenta que auxilia as actividades de desenho e de análise de cobertura de testes unitários baseados em grafos de controlo de fluxo (CFG). Esta ferramenta foi pensada para ser usada como apoio ao ensino dos conceitos e técnicas introdutórias de teste de *software*. O PESTT foi inicialmente desenvolvido em 2012 para integrar o Eclipse IDE 3.x, no âmbito da tese de mestrado do Rui Manuel da Silva Gameiro, orientado pelo Prof. Doutor Francisco Cipriano da Cunha Martins.

Desde 2012, novas versões do Eclipse foram lançadas, estando atualmente na versão 4.11.2. Após o lançamento da 4ª versão do Eclipse, o funcionamento do PESTT foi comprometido. O principal objetivo desta tese é o de adaptar o PESTT para ficar em conformidade com as mudanças efetuadas do Eclipse 3.x para o Eclipse 4.x. Para tal, alterou-se significativamente o desenho do pacote da *User Interface* para tirar partido do novo MVC disponibilizado pelo Eclipse 4.

O trabalho realizado no âmbito desta tese resultou naquilo que era desde cedo o nosso objetivo: voltar a disponibilizar o PESTT para as versões atuais do Eclipse, embora para um subconjunto das funcionalidades. As funcionalidades disponibilizadas nesta versão do *plug-in* cobrem a maioria dos aspetos que tornaram o PESTT uma ferramenta com valor suficiente para que este projeto tenha sido criado e desenvolvido.

Neste documento são descritos os objetivos do projeto, a motivação e importância da sua concretização, o trabalho anteriormente feito e relacionado, as metodologias utilizadas, os resultados alcançados e possível trabalho futuro.

Palavras-chave: *Plug-in*, PESTT, teste de *software*, engenharia de *software*, grafo, Eclipse.

Abstract

The main goal of Software Engineering is to produce quality software that meets the functional requirements. Several factors contribute to achieve this challenge, including application testing. It is, therefore, paramount to appropriate master software testing techniques.

The PESTT Educational Software Testing Tool (PESTT) is a tool that assists in the design and coverage analysis of unit tests based on control flow graphs (CFG). This tool is intended to assist in teaching the concepts and introductory techniques of software testing. PESTT was initially developed in 2012 to integrate the Eclipse IDE 3.x, within the context of the master's thesis of Rui Manuel da Silva Gameiro, supervised by Prof. Doctor Francisco Cipriano da Cunha Martins.

Since 2012, new versions of Eclipse have been released and are currently in the 2019-09 version (4.11.2). After the release of Eclipse 4th version, it was found that PESTT execution was compromised. The main purpose of this thesis is to adapt PESTT to conform to changes made from Eclipse 3.x to Eclipse 4.x. For that, the design of the user interface packet has significantly changed to take advantage of the new MVC provided by Eclipse 4.

The work carried out under this thesis fulfilled our main goal: to make PESTT available again for the current versions Eclipse, although some functionalities were left outside. The features provided in this version of the plug-in cover most aspects that have made PESTT a valuable tool for the creation and execution of this project.

This document describes the objectives of the project, the motivation and importance of its implementation, the related and previously done work, the used methodologies, the achieved results, and possible future work.

Keywords: Plug-in, PESTT, software test, software engineering, graph, Eclipse.

Índice

Lista de quadros e figuras

Capítulo 1
Introdução

1.1 Contexto

Na sociedade atual o *software* é um elemento chave em diversos dispositivos preponderantes no nosso dia a dia [1]. É tentador que, ao pensar em s*oftware,* imaginemos uma sequência de instruções escritas numa dada linguagem de programação, que em conjunto formam um programa [2]. Este programa é, normalmente, designado por *código fonte*.

O *Software* é uma composição de vários programas de computador, procedimentos, documentação e dados necessários para operar um sistema [3,4]. O *software* é o responsável por definir o comportamento de, por exemplo, sistemas embebidos que controlam aviões e sistemas de controlo de tráfego aéreo, *routers*, bem como carros, telemóveis e simples comandos de abertura de portões [1].

Dada a ubiquidade do *software*, é fácil perceber que, se este contiver defeitos, estes podem ser desastrosos, tanto em termos monetários, como em termos de vidas humanas. Existem várias referências a episódios de falhas de *software* ao longo da história que sustentam esta afirmação, algumas delas com repercussões bastante graves. A título de exemplo, em 2016 as falhas no *software* tiveram um custo para a economia mundial de 1.1 triliões de dólares. Estas falhas foram encontradas em 363 empresas e afetaram 4.4 biliões de clientes. A maioria destes incidentes poderia ter sido evitada se as passagens a produção tivessem sido feitas com um acompanhamento apropriado em termos de qualidade [27]. Um episódio bastante grave envolveu um acelerador utilizado no tratamento de tumores [6], que emitia radiação de alta energia sobre células cancerígenas, sem causar danos aos tecidos circundantes. Devido a uma falha de programação durante um processo de melhoramento do sistema, o sistema passou a emitir 100 vezes a radiação requerida. Como consequência, morreram pelo menos cinco pessoas e várias dezenas sofreram os efeitos de ficarem expostas a uma elevada radiação [6]. Mais recentemente, mais precisamente em 2012, uma falha de *software* que despoletou uma perda de 440 milhões de dólares em apenas 30 minutos na

Knight, uma das maiores empresas de compra e venda de ações nos Estados Unidos. Em 2017 um *bug* no sistema de alocação de pilotos a voos da *American Airlines* fez com que fossem atribuídas folgas a demasiados pilotos durante a época de natal, levando a que a empresa tivesse de compensar os pilotos em causa com 1,5 folgas por cada folga, de modo a evitar o cancelamento de 15000 voos.

Apesar de existirem vários fatores que contribuem para a construção de *software* fiável, testar é o primeiro método utilizado pela indústria para avaliar a qualidade de *software* em desenvolvimento [1]. Testar tem quatro objetivos primários [5]:

1. Identificar a magnitude e as fontes de risco inerentes ao desenvolvimento;
2. Reduzir os riscos anteriormente identificados;
3. Ganhar confiança no *software*;
4. Fazer a gestão da atividade de testes como parte integrante do desenvolvimento.

Os testes feitos ao *software* devem ser eficazes na deteção de defeitos, mas também devem ser o mais eficientes possível (rápido desempenho e baixo custo) [7, 8]. Apesar de extremamente benéfico, testar *software* é caro e trabalhoso, representando mais de 50% dos custos do seu desenvolvimento [1]. Para que se possam reduzir os custos e minimizar erros humanos durante a atividade de teste, esta deve ser a mais automatizada possível. Para esse efeito existem várias ferramentas de auxílio à automatização de testes, seguindo abordagens diversas [22, 23].

A ferramenta *PESTT*, disponibilizada sob a forma de um *plug-in* para Eclipse, auxilia as atividades de desenho e de análise de cobertura de testes **unitários** baseados em **grafos de fluxo de controlo** (GFC) — estas noções são descritas detalhadamente no capítulo 2. O desenvolvimento desta ferramenta foi motivado pela importância, cada vez maior, da realização de testes de *software*, pelos benefícios que a automatização dos processos de desenho e análise têm em termos de tempo despendido e ocorrência de erros, pelo impacto que a ferramenta pode ter quando usada para a aprendizagem de técnicas e conceitos de teste de *software*, e pela influência positiva na qualidade do *software* produzido, que pode advir do uso da ferramenta. O *PESTT* foi desenvolvido para a versão 3.x do Eclipse. Contudo, o

Eclipse foi alvo de uma transformação extensa, em particular, no funcionamento do seu GUI e o *PESTT* deixou de ser compatível com estas alterações e, assim, de estar disponível para auxílio ao desenho e análise de cobertura de testes.

1.2 Motivação e Objetivos

Atualmente não é possível utilizar o PESTT dada a incompatibilidade detetada com a atual versão do Eclipse IDE. Assim, o principal objetivo deste projeto é o de tornar o PESTT novamente utilizável em ambiente Eclipse; para isso, dividimos este macro objetivo nos seguintes objetivos:

1. tornar funcional esta ferramenta educativa, interessante e de elevada utilidade;
2. melhorar a ferramenta, usufruindo dos benefícios inerentes à migração da API 3.x para a API 4.x do Eclipse IDE;
3. determinar, de forma precisa e automática, o grau de cobertura dos requisitos de teste.

1.3 Estrutura do documento

O tópico principal desta dissertação é o PESTT e o trabalho de engenharia levado a cabo para refazer as partes que não estão de acordo com o atual modelo arquitetural do Eclipse 4.x. Primeiramente, neste capítulo introdutório, explicamos a elevada presença e o impacto que atualmente o *software* tem no dia-a-dia da maioria das pessoas. Sendo o *software* de elevada criticidade, é fácil entender a importância extrema em testar as aplicações produzidas. No Capítulo 2 revemos os conceitos base relacionados com testes de *software*, bem como a ferramenta Eclipse e o seu processo de desenvolvimento criação de *plug-ins*. No capítulo 3 é feita uma análise retrospetiva das ferramentas semelhantes ou que dão a sua contribuição no desenvolvimento do PESTT. No capítulo 4 descrevemos o trabalho de análise ao estado anterior do PESTT, e o trabalho realizado para adaptar esta ferramenta ao Eclipse 4, apresentando também um sumário do plano inicial de trabalho versus o realizado. Concluímos este documento com uma análise crítica e uma proposta de trabalho futuro.

1.4 Resultados obtidos

O desenvolvimento deste trabalho culminou com aquilo que era o principal objetivo deste projeto: disponibilizar para o Eclipse IDE 4.x um conjunto de funcionalidades, relativamente a testes sobre grafos de fluxo de controlo (GFC). Estas funcionalidades caracterizam-se pela simplicidade tanto de compreensão como de utilização, pela utilidade e pela facilidade na aprendizagem e compreensão da ferramenta.

As funcionalidades acima referidas são:

- desenho do GFC de um método a partir da seleção de um pedaço de código correspondente à mesma;
- realce de linhas de código a partir da seleção dos nós ou arestas correspondentes no GFC;
- obtenção de requisitos de testes sobre o GFC consoante o critério de cobertura escolhido interativamente pelo utilizador;
- obtenção de estatísticas sobre caminhos de testes introduzidos interativamente pelo utilizador.

Estas funcionalidades não cobrem a totalidade das funcionalidades que inicialmente previmos disponibilizar. Apesar disso, a ferramenta entregue, num todo, satisfaz um conjunto de requisitos e aspetos que lhe atribuem valor e utilidade suficiente que justifiquem a sua disponibilização. A figura 1.1. esquematiza as funcionalidades inicialmente previstas a disponibilizar nesta versão do PESTT e as que efetivamente serão disponibilizadas, ficando claro quais destas serão alvo de um possível trabalho futuro:

Funcionalidade	Previsto	Implementado
Desenho do GFC	✓	✓
Adicionar informação extra ao GFC	✓	✗
Apresentação da vista de seleção de critérios	✓	✓
Apresentação da vista de conjunto de requisitos de teste	✓	✓
Apresentação da vista de requisitos de teste manuais	✓	✓
Apresentação da vista de estatísticas	✓	✓
Seleção de um nó do GFC e realce do correspondente pedaço de código	✓	✓
Seleção de um pedaço de código e realce do correspondente nó do GFC	✓	✗
Visualização de caminhos de teste no código e no GFC	✓	✗

Figura 1.1.: Esquema de funcionalidades implementadas na nova versão do PESTT.

Capítulo 2
Conceitos base

2.1 Testes de *software*

O número de potenciais entradas diferentes para um programa é tão grande que é impraticável testar todas elas — basta considerar-se o caso em que uma das entrada é uma *string* (há potencialmente infinitas *strings* distintas). Não sendo exequível testar um programa sobre todas as entradas possíveis, recorrem-se a critérios para escolher que entradas são relevantes para testar o *software*. A execução do código sobre uma dada entrada testa uma parte do código que se diz ser "coberta" por esse teste — é esta a noção de "cobertura" de um teste [1].

2.1.1 Tipos de teste

Uma das formas mais eficazes de escrever testes é fazê-lo paralelamente a cada fase do ciclo de desenvolvimento de *software*, podendo os testes derivar de especificações, requisitos, artefactos de desenho ou do código fonte. Existem vários níveis de testes (o chamado modelo em V)[1], sendo que cada nível acompanha uma atividade específica de desenvolvimento:

- Testes de aceitação – são desenhados para determinar se o produto final vai ao encontro das necessidades e expectativas do cliente. Estes testes são feitos por utilizadores ou outros indivíduos que tenham um forte conhecimento do domínio;
- Testes de sistema – são desenhados para determinar se o sistema está de acordo com a especificação. Nesta fase, assume-se que as peças funcionam individualmente, sendo a grande questão a de testar se o sistema funciona como um todo. Normalmente, não são feitos por programadores, mas sim por uma equipa de testes separada;
- Testes de Integração – avaliam se as interfaces entre módulos num dado subsistema têm assunções coerentes e comunicam corretamente entre si. Nesta fase, assume-se que o funcionamento dos módulos está correto. Este tipo de testes é, normalmente, da responsabilidade da equipa de desenvolvimento;
- Testes de Módulos – são desenhados para avaliar os vários módulos

individualmente, incluindo como é que as várias unidades interagem umas com as outras. Normalmente, estes testes são efetuados pelos próprios programadores;

- Testes Unitários – são os testes de mais baixo nível e avaliam as unidades produzidas durante a fase de implementação do *software*. À semelhança dos testes de módulos, estes testes costumam ser da responsabilidade do programador.

2.1.2 Casos de Teste e Critérios e Cobertura

A todo o *software* que é produzido deve estar associado um conjunto de casos de teste. Um caso de teste é uma entrada sobre a qual um programa é executado, e à qual está associado um valor esperado [8]. O problema é que o número de potenciais entradas para a maioria dos programas tende a ser muito grande, não podendo ser explicitamente enumeradas. Um projeto de teste de *software* envolve, entre outras coisas, a escrita de um conjunto de testes que se pretende que seja o mais completo possível. Contudo, avaliar o grau de cobertura destes testes é uma tarefa que não se pode automatizar na totalidade. Sendo impossível testar programas para todas as entradas, utilizam-se *critérios de cobertura*. Um critério de cobertura é um procedimento efetivo para gerar requisitos de teste de forma sistemática [1]. Utilizar critérios de cobertura auxilia na decisão de quais as entradas a usar durante os testes, tornando mais provável que os defeitos no *software* em estudo sejam encontrados. Para além disso, o uso de critérios de cobertura é também útil para que o engenheiro de testes saiba quando deve parar de testar, assegurando-se de que o *software* testado é confiável e de alta qualidade.

Os critérios de cobertura podem ser classificados em quatro categorias [1]:

- Grafos de fluxo de controlo;
- Expressões lógicas;
- Partição do espaço de entrada;
- Sintaxe.

No caso do PESTT, os requisitos de teste são gerados com base em grafos de fluxo de controlo, que são abordados em maior detalhe na secção seguinte.

2.1.3 Grafo de Fluxo de Controlo e Cobertura de Grafo

Grafos de fluxo de controlo são grafos dirigidos cujos nós correspondem a grupos de instruções no código sem ramos e cujos arcos dirigidos têm uma correspondência um-para-um com cada possível ramo de computação do programa fonte. Desta forma, a cada arco fica associada, pelo menos implicitamente, a condição (*e.g.*, restrições sobre valores de variáveis do código) que permitem que o fluxo de execução transite do nó origem para o nó destino do arco.

Um bloco básico de código é uma sequência máxima de instruções tal que, se uma das instruções do bloco for executada, todas as instruções do bloco serão igualmente executadas. Assim, um bloco básico tem um só ponto de entrada em execução e um só ponto de saída de execução [1]. Simples sequências de instruções no código, i.e., blocos básicos, correspondem assim a grafos em que cada nó tem no máximo um arco de entrada e no máximo um arco de saída. A figura 2.1. ilustra 2 blocos básicos. No bloco básico à esquerda estão ilustrados os casos de nós com um arco de entrada, nós com um arco de saída, nós sem arcos de entrada e nós sem arcos de saída. O bloco básico à direita ilustra o caso de um nó sem arcos de entrada e sem arcos de saída.

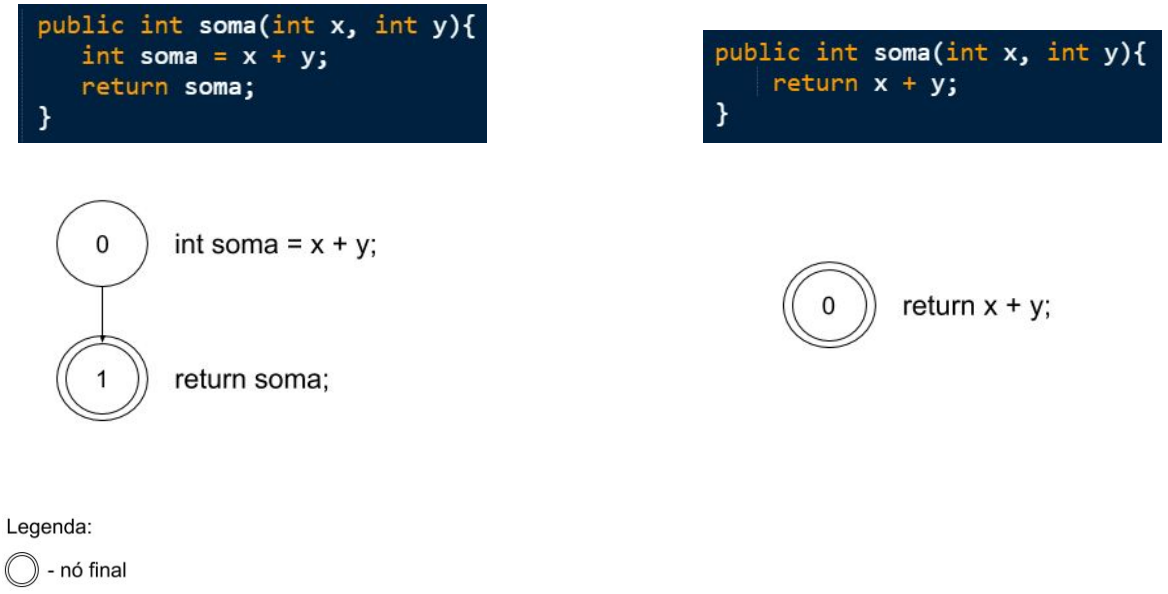

Figura 2.1.: Blocos básicos de código.

Estruturas mais complexas de código refletem-se em grafos aninhados, *e.g.*, um *if* traduz-se num grafo com dois ramos sendo que o código a executar no ramo *then* será ele próprio um sub-grafo, bem como o código a executar no ramo *else*. Num ciclo, o corpo será um grafo aninhado dentro de um grafo maior representado o próprio ciclo.

Os testes de *software* baseados em GFC têm como abordagem testar o comportamento do *software* ao longo dos vários caminhos de execução possíveis, traduzidos pelos caminhos no seu GFC. Diversos critérios de teste são possíveis, refletindo-se estes em diversas estratégias de seleção dos caminhos de execução a testar no grafo.

Seguindo a abordagem de testes baseados em GFC, um conjunto de teste é composto por casos de teste que têm por objetivo exercitar determinados caminhos no grafo. Assim, um conjunto de teste "cobre" uma certa proporção de todos os caminhos possíveis no grafo.

2.1.4 Requisitos de Teste e Critérios de Cobertura

Um requisito de teste é um elemento específico de um artefato de *software* que um caso de teste deve satisfazer ou cobrir [1]. Se o objetivo da etapa de teste for cobrir todas as decisões que podem ser tomadas no programa (cobrir todos os ramos de execução), então cada ponto de decisão leva a dois requisitos de teste: um para o caso em que a condição da decisão é avaliada a *true* e outro em que é avaliada a *false*.

Um critério de cobertura é um procedimento efetivo para gerar requisitos de teste de forma sistemática (ver secção 2.1.2.). Além disso, pode ser utilizado como uma forma de avaliar um conjunto de teste na medida em que os caminhos correspondentes aos casos de teste contidos naquele "cobrem" o grafo que representa o código fonte [1].

Definição 1: Um caminho simples é aquele que não contém ciclos.
Definição 2: Um caminho primo é um caminho simples (exceto quando ele próprio é um ciclo) que não é um sub-caminho de nenhum outro caminho simples.
Definição 3: Um caminho de ida e volta é um caminho primo de comprimento maior que 0, que começa e termina no mesmo nó.
Os critérios de cobertura utilizados no âmbito do PESTT são critérios de controlo de fluxo, e

são os seguintes:

- **Cobertura de nós** (do inglês *Node coverage*), onde o conjunto de requisitos de teste contém todos os nós do grafo (caminhos de comprimento 0).
- **Cobertura de arestas** (do inglês *Edge coverage*), onde o conjunto de requisitos de teste contém todos os caminhos do grafo que têm comprimento 0 ou 1.
- **Cobertura de par de arestas** (do inglês *Edge-Pair coverage*), onde o conjunto de requisitos de teste contém todos os caminhos do grafo que têm comprimento 0, 1 ou 2.
- **Cobertura de caminhos primos** (do inglês *Prime Path coverage*), onde o conjunto de requisitos de teste contém todos os caminhos primos do grafo.
- **Cobertura de ida e volta simples** (do inglês *Simple Round Trip coverage*), onde o conjunto de requisitos de teste contém pelo menos um caminho de ida e volta (r*ound trip path*) para cada nó alcançável no grafo, no qual começa e acaba um *Round Trip Path*.
- **Cobertura de ida e volta completa** (do inglês *Complete Round Trip coverage*), onde o conjunto de requisitos de teste contém todos os caminhos de ida e volta (r*ound trip path*) para cada nó alcançável no grafo.
- **Cobertura de todos os caminhos** (do inglês *Complete Path coverage*), onde o conjunto de requisitos de teste contém todos os caminhos do grafo.

De notar que existem outros critérios baseados em grafos de fluxo de controlo, para além dos mencionados acima. Por exemplo, os critérios baseados em fluxo de dados também são testes sobre grafos. Estes são testes estruturais com foco nos pontos em que as variáveis do programa tomam valores e nos pontos em que essas variáveis são usadas ou alteradas [28]. Ou seja, os critérios baseados em fluxo de dados usam o grafo de fluxo de controlo para mitigar potenciais variações anómalas dos dados do programa.

2.2 Modelo-Vista-Controlador

O Modelo-Vista-Controlador (MVC), do inglês *Model-View-Controller*, é um padrão arquitetural de *software* orientado a objetos. É comummente usado no desenvolvimento de aplicações com interface gráfica com o utilizador (*Graphical User Interface* — GUI) e divide a aplicação em três partes que comunicam entre si.

O Modelo é a componente que contém as estruturas de dados dinâmicas da aplicação, independentes da interface com o utilizador, e é responsável pela gestão dos dados, lógica e regras da aplicação.

A Vista é a forma de apresentação da informação ao utilizador, nomeadamente, através de uma interface gráfica. Esta apresentação de informação pode tomar as mais variadas formas, tais como gráficos, tabelas, textos, imagens, ou quaisquer outras formas inteligíveis pelo utilizador..

O Controlador é o responsável por receber todos os eventos (ou pedidos) do utilizador. Esta componente contém várias ações que são responsáveis por cada página, controlando qual o Modelo que deve ser utilizado em cada situação e qual a Vista que deverá ser mostrada ao utilizador.

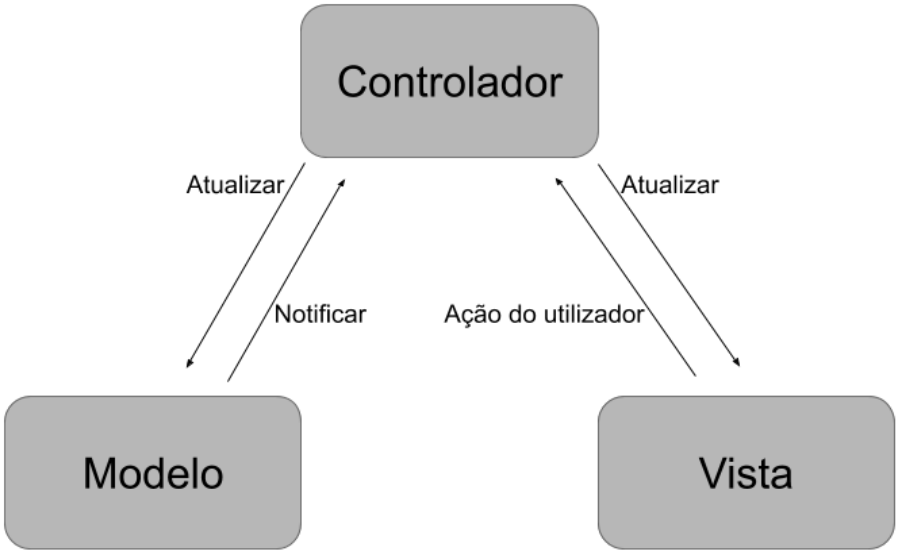

Figura 2.2.: Ilustração do padrão arquitetural MVC (adaptada de [29]).

Em suma:

1) o utilizador vê a Vista e interage com ela introduzindo os seus dados*;*

2) os dados do utilizador são capturado pelo Controlador que os transforma em atualizações da Vista ou do Modelo*;*

3) quando o Modelo recebe um pedido de atualização do Controlador*,* executa os mecanismos necessários às atualizações dos dados e notifica o Controlador, que por sua vez atualiza a Vista*.*

Esta decomposição tem por objetivo tanto a separação de responsabilidades entre as componentes, como a separação da representação interna da informação da sua apresentação externa para o utilizador. O desacoplamento das componentes de código obtido pela estruturação da aplicação segundo este padrão permite obter uma elevada reusabilidade de código das componentes, e facilita o seu desenvolvimento em paralelo.

2.3 Eclipse IDE

O Eclipse é um IDE para desenvolvimento Java. Porém, suporta várias outras linguagens a partir de *plug-ins* (programa usado para adicionar funções a outros programas maiores) como C/C++, PHP, ColdFusion, Python, Scala e plataforma Android. O Eclipse IDE é desenvolvido em Java, segue uma filosofia *open source* para desenvolvimento de *software*, e contém um conjunto padrão de *plug-ins*, incluindo as indispensáveis Ferramentas de Desenvolvimento Java (JDT, do inglês *Java Development Tools*).

No final dos anos 90, o projeto Eclipse foi iniciado na IBM, que desenvolveu a primeira versão do produto e o doou como *software* livre à comunidade [17]. O gasto inicial da IBM no produto foi de mais de 40 milhões de dólares. Hoje, o Eclipse é o IDE Java mais utilizado no mundo. Possui como forte característica o uso da SWT como biblioteca gráfica, a forte orientação ao desenvolvimento baseado em *plug-ins* e o amplo suporte ao programador com centenas de *plug-ins* que procuram atender as diferentes necessidades de diferentes programadores [17].

Sendo as aplicações Eclipse feitas à base de componentes de *software* individuais é preciso configurar e gerir o seu comportamento. O ficheiro mais importante para esta configuração é chamado *MANIFEST.MF*. Neste ficheiro estão definidos meta-dados referentes aos *plug-ins*, como por exemplo, identificadores únicos, APIs e dependências.

À semelhança de uma comunidade, a maioria dos *plug-ins* não atua isoladamente. Na maior parte dos casos, os *plug-ins* usam serviços fornecidos por outros *plug-ins,* ou expõem os seus serviços para serem consumidos por outros *plug-ins*. Alguns grupos de *plug-ins* estão diretamente relacionados, como é o caso de *plug-ins* que fornecem ferramentas para o desenvolvimento Java *(Java Development Tools)*, enquanto que outros são *plug-ins* isolados sem qualquer consciência da presença de outros *plug-ins* – como é o caso do *Standard Widget Toolkit (SWT)*.

Embora a maior utilização do Eclipse seja feita com o foco num ambiente de desenvolvimento integrado (IDE) Java, o Eclipse inclui também o *Plug-in Development Environment* (PDE), que serve para desenvolver ferramentas que se integram perfeitamente com ambiente do Eclipse, estendendo o mesmo.

2.3.1 Arquitetura Eclipse IDE 4.x

As arquiteturas devem ser continuamente examinadas para avaliar se ainda são apropriadas. É capaz de incorporar novas tecnologias? Incentiva o crescimento da comunidade? É fácil atrair novos colaboradores? No final de 2007, os responsáveis pelo desenvolvimento do projeto Eclipse decidiram que a resposta para essas perguntas era negativa e começaram o projeto de uma nova visão para o Eclipse. Ao mesmo tempo, percebeu-se que havia milhares de aplicações Eclipse que dependiam da API existente. No final de 2008 foi criado o projeto e4: um projeto de incubadora para a exploração da comunidade de tecnologias para a Plataforma Eclipse. Este projeto foi usado para o desenvolvimento do mecanismo subjacente da plataforma Eclipse 4 e foi concluído com o seu lançamento [34]. O projeto e4 tinha, então, três objetivos específicos: simplificar o modelo de programação do Eclipse; atrair novos contribuidores; e permitir que a plataforma aproveitasse as novas tecnologias baseadas na Web, através de uma arquitetura *open source* [31].

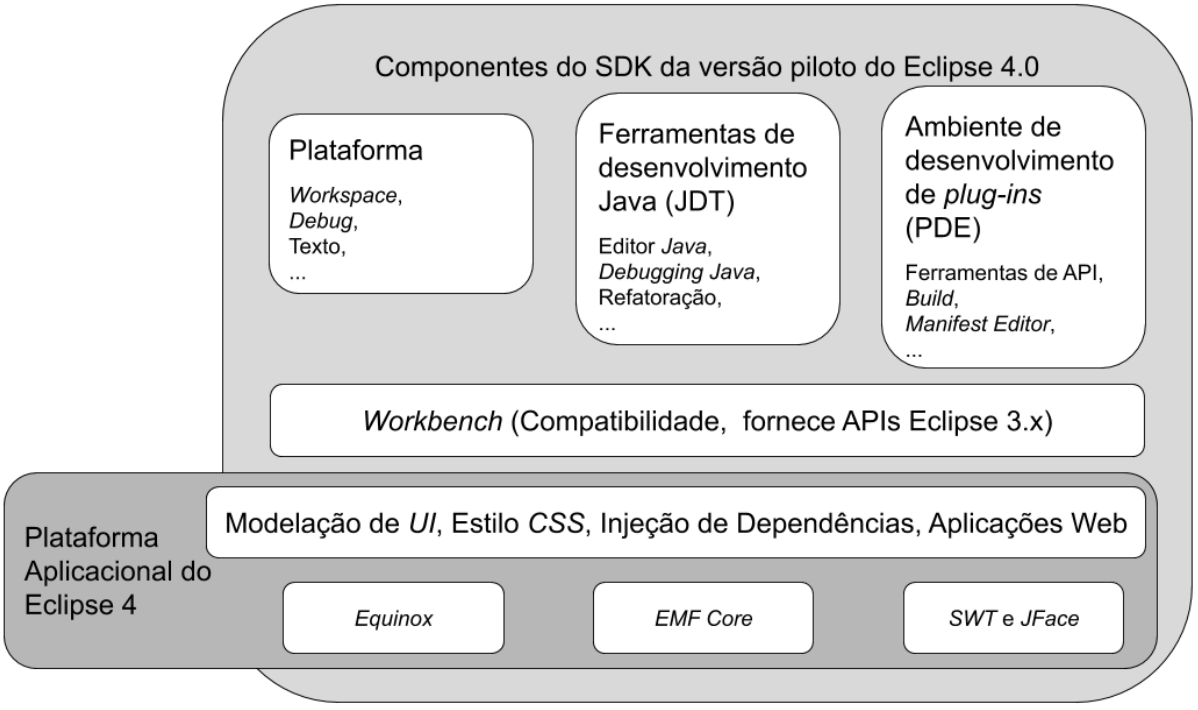

Figura 2.3.: Ilustração dos componentes do SDK da versão piloto do Eclipse 4.0 (adaptado de [31]).

O Eclipse 4.0 foi lançado pela primeira vez em julho de 2010 para obter *feedback* dos primeiros utilizadores, e consistia numa combinação de pacotes SDK que faziam parte da versão 3.6 e em novos pacotes que se formaram no novo projeto. Tal como no Eclipse 3.x,

havia uma camada de compatibilidade para que os pacotes existentes pudessem funcionar com a nova versão, mas com a ressalva de compatibilidade apenas no caso de estarem a ser usadas APIs públicas. Tal garantia não se estendeu a pacotes ou código interno.

Na versão 4 do Eclipse IDE foram introduzidas um conjunto de alterações arquiteturais. Algumas das alterações mais relevantes são as seguintes [30]:

- Alteração da representação interna da interface do utilizador:

A interface do utilizador da plataforma Eclipse passou a ser representada internamente usando a *Eclipse Modeling Framework* (EMF) [32], uma *framework* de modelação e geração de código para construir aplicações com base num modelo de dados estruturados. A partir de uma especificação de modelo descrita num XMI (*XML Metadata Interchange*), o EMF fornece ferramentas e suporte em tempo de execução para produzir um conjunto de classes Java para o modelo, juntamente com um conjunto de classes adaptadoras que permitem a visualização e edição baseada em comandos do modelo e um editor básico. As aplicações podem reconfigurar ou estender esse modelo para obter apresentações muito diferentes de sua aplicação sem necessidade de adição de código. Normalizar a estrutura do *workbench* como um modelo bem definido tem o benefício adicional de tornar o código do próprio ambiente de trabalho muito mais simples e menos propenso a erros. O mais importante é que isso permite *layouts* UI muito diferentes do *workbench*, como partes que vivem fora de perspectivas, visualizações e editores em diálogos e outros *designs* não permitidos anteriormente pelo *workbench* mais antigo com um modelo rígido e manual. A adoção do EMF permite suportar a construção de ferramentas mais avançadas para *designers* de aplicações, como ferramentas de desenho visual [30, 31].

- Alteração da estrutura interna dos modelos:

O modelo do *workbench* é separado em vários níveis de blocos de construção. Esses são conjuntos de funcionalidades de modelos relacionados que podem ser usados para aumentar os recursos básicos do modelo *Rich Client Platform* (RPC). Esta facilita a integração de componentes de *software* independentes, onde a maioria do processamento de dados acontece do lado do cliente [33] para aprimorar a interface do utilizador. Isso permite que os programadores de RCP escolham os aprimoramentos específicos de que realmente precisam,

em vez de terem de escolher entre um modelo que seja simplista em comparação com um que possa suportar a interface de utilizador do IDE [30, 31].

- Alteração da camada de apresentação:

O modelo de ambiente de trabalho agora é traduzido em *widgets* (janelas, botões, etc) concretizados por meio de uma API genérica do mecanismo de apresentação. A plataforma inclui um mecanismo de apresentação padrão que renderiza o modelo por meio de *widgets* SWT tradicionais, mas as aplicações podem empregar mecanismos de apresentação alternativos para renderizar o seu modelo aplicacional usando um *kit* de ferramentas de *widget* diferente [30, 31].

- Fragmentos de modelo (do inglês *Model fragments*)*:*

A extensibilidade no modelo do *workbench* é obtida através de um conceito de fragmentos de modelo. Os autores de *plug-ins* podem fornecer fragmentos de modelo que o ambiente de trabalho funde no modelo "base" da aplicação. Isso é feito por *plug-ins* que contribuem com arquivos XMI usando um *container model fragment* do meta-modelo do *workbench*. Um *container* é uma unidade padrão de *software* que encapsula código e todas as suas dependências para que a aplicação seja executada de maneira rápida e confiável em vários ambientes computacionais. Quando uma aplicação da plataforma Eclipse 4.x é inicializada, são incluídos no modelo base fragmentos de modelo enviados por *plug-ins*, permitindo, por exemplo, que uma visualização seja incluída numa perspectiva definida pela aplicação [30, 31].

- Pontos de extensão de UI (do inglês *UI extension points*)*:*

Como a interface do utilizador foi alterada para ser baseada num modelo, faz sentido converter os pontos de extensão de UI em fragmentos de modelo. Com essa abordagem, uma vista pode ser descrita como um fragmento de modelo contendo um *PartDescriptor*, em vez de uma extensão para o ponto de extensão *org. eclipse.ui.views*.

Isso reduz o número de diferentes paradigmas que o programador precisa de ter em mente: todas as informações que definem os pontos de extensão da interface do utilizador vêm do modelo e todas as extensões podem ser colocadas e referenciadas num modelo [30, 31].

- Contextos hierárquicos:

O pacote *org.eclipse.e4.core.contexts* introduz a noção de contextos hierárquicos por meio da API *IEclipseContext*. Os contextos fornecem um mecanismo para isolar o código da aplicação da *framework* Eclipse, fornecendo uma abstração através da qual o código da aplicação pode obter objetos e serviços da *framework*. Os contextos também fornecem uma maneira de o código da aplicação disponibilizar dados e serviços para outro código da aplicação de uma maneira fracamente acoplada. Os contextos atualmente suportam [30, 31]:

➢ Hierarquias de contexto. Os contextos podem ser agregados num contexto pai para substituir os serviços fornecidos pelo pai. Isso permite que o código da aplicação escrito num contexto seja facilmente transferido para funcionar em diferentes contextos.

➢ Pesquisa de serviço OSGi. Os contextos podem ser usados para obter referências aos serviços OSGi. O contexto é o responsável por rastrear mudanças dinâmicas de serviço e da limpeza de serviços não utilizados quando os contextos são descartados. Um serviço OSGi é um conjunto de especificações que define um sistema dinâmico de componentes para a Plataforma Java. Estas especificações têm como objectivo reduzir a complexidade do *software*, fornecendo uma arquitetura modular e orientada a serviços para grandes sistemas distribuídos, pequenas aplicações, bem como aplicações embebidas. A plataforma OSGi tem vindo a tornar-se o padrão para o desenvolvimento de aplicações modulares em Java.

➢ Injeção de dependências. Os valores de um contexto podem ser injetados num objeto da aplicação. Isso resulta no fornecimento de campos e métodos injetáveis no objeto da aplicação com valores definidos no contexto. A injeção remove completamente a dependência direta do código da aplicação na estrutura. O conceito de injeção de dependências vai ser revisitado em maior detalhe no capítulo 4.

➢ Armazenamento de funções. Os contextos podem armazenar funções que são avaliadas para obter valores de contexto. Por outras palavras, é possível armazenar uma *ContextFunction* num *IEclipseContext* através do uso da chave que mapeia o objeto que se vai obter. *IEclipseContext* é um mapa chave-valor hierárquico. As chaves são, normalmente, *strings* e os valores são qualquer objeto *Java*. Cada contexto tem um *parent*, de modo a que os contextos estejam ligados para que formem uma estrutura. Quando uma chave não devolveu resultados num determinado contexto, é feita uma nova tentativa, desta vez no seu *parent*, sendo este processo repetido até que o valor seja encontrado, ou até que se chegue à *root* da

estrutura. Contextos incluem um tipo particular de valores chamado *ContextFunctions*. Quando um valor devolvido por uma chave é uma *ContextFunction*, o *IEclipseContext* executa um método chamado *compute(context, key)* e devolve o resultado dessa computação [50].

➢ Notificação de alteração tradicional e registo de código de atualização de estilo de vinculação de dados com um contexto. Cada vez que o código de atualização registado é executado, o contexto faz uma procura de quais os valores de contexto que foram referenciados. Alterações subsequentes em qualquer um desses valores resultarão na repetição do código de atualização. Atualizações e eventos são agrupados e alinhados para evitar notificações desnecessárias.

Na adaptação do PESTT levada a cabo neste projeto, muitas das alterações do Eclipse versão 3 para a versão 4 acima mencionadas tiveram um papel de destaque. O maior exemplo recai sobre os contextos hierárquicos, onde se fez um extenso uso da injeção de dependências e hierarquias de contexto. Também o conceito de *extension points,* e a necessidade da sua conversão para *model fragments,* foi um aspecto de grande peso, uma vez que a adoção de *model fragments* em detrimento dos *extension points*, introduzida no Eclipse 4, foi a principal causa do não funcionamento do PESTT nesta nova versão do Eclipse.

Capítulo 3

Trabalho Relacionado

As ferramentas para a geração automática de testes podem classificar-se em duas categorias [1, 9]: (i) geradores*,* isto é, ferramentas que geram automaticamente valores que satisfazem um dado critério, e (ii) reconhecedores, que decidem se um dado conjunto de testes satisfaz ou não um critério. O PESTT, dependendo do modo de utilização, pode ser utilizado como gerador ou reconhecedor. Nas secções seguintes são apresentadas ferramentas do estado da arte para cada uma destas categorias, bem como a sua comparação em termos de funcionalidade com o PESTT.

3.1 Geradores

3.1.1 Jwalk

O Jwalk [11] é uma ferramenta que foi desenvolvida para realizar testes unitários sistemáticos no contexto de metodologias ágeis, e assenta no paradigma de testes L*azy*. *Lazy Systematic Unit Testing* é um modo de efetuar testes unitários com base na [11]:

- capacidade para inferir a especificação da unidade de *software* em causa, através de uma análise dinâmica;
- capacidade para explorar e testar o espaço de estados da unidade de forma exaustiva para profundidades limitadas.

Isto é conseguido, em parte, usando a capacidade de reflexão do Java e através da interação com o utilizador. Desta forma, o Jwalk tem dois modos de execução: sem intervenção humana, o Jwalk executa uma exploração exaustiva e limitada dos métodos da classe, que pode ser direcionada para explorar o espaço de construções algébricas ou o espaço de estados da classe testada; com intervenção humana, o Jwalk executa testes totalmente automatizados baseados em estados, a partir de uma especificação do utilizador adquirida de forma incremental. Enquanto que o Jwalk "aprende", de forma interactiva com o utilizador,

qual a especificação do *software* e executa os testes necessários para verificar o seu cumprimento, o PESTT gera caminhos de execução de código do *software* que, em conjunto, cobrem parte do seu GFC de acordo com o critério selecionado.

3.1.2 Randoop

Testes de regressão é uma técnica que consiste na aplicação de um conjunto de testes a versões mais recentes de um *software*, para garantir que não surgiram novos defeitos em componentes já analisados [51].

O Randoop [12] é uma ferramenta que gera testes unitários para código Java através da geração de testes aleatórios. Esta técnica gera sequências aleatórias de invocações de métodos e construtores para as classes a testar. O Randoop executa as sequências por si criadas, utilizando os resultados para criar asserções que capturem o comportamento do programa. A partir destas sequências e asserções, são criados os testes.

Esta ferramenta pode ser utilizada com dois propósitos:

- encontrar defeitos no programa;
- criar testes de regressão.

Enquanto o Randoop gera casos de testes de forma automática que servem como base a testes unitários, o PESTT gera requisitos de testes com base em critérios de cobertura.

3.1.3 Korat

O Korat [9] é uma plataforma para gerar e executar de forma automatizada testes unitários para programas Java. Dada uma especificação formal de um método, esta ferramenta baseia-se nas pré-condições para automaticamente gerar todos os casos de testes até um certo tamanho, definido pelo utilizador. De seguida, o método é executado para esses casos de teste, sendo o resultado comparado com as pós-condições para verificar a sua correção.

Enquanto o Korat gera e executa casos de testes de forma automática, o PESTT gera requisitos de testes. Ambos se baseiam em critérios escolhidos pelo utilizador.

3.1.4 Evosuite

O EvoSuite [10] é uma ferramenta que gera automaticamente testes unitários para Junit. Pode ser integrada com o Eclipse, IntelliJ [15] e Maven [16] e atua ao nível do *bytecode* Java. É uma ferramenta totalmente automatizada que não necessita de testes manualmente escritos ou testes unitários parametrizados. Por exemplo, quando utilizado no Eclipse, o utilizador apenas precisa de selecionar a classe e os testes são gerados com um clique. Esta ferramenta utiliza um algoritmo genético para evoluir testes candidatos, usando operadores inspirados na evolução natural, para produzir soluções melhores iterativamente.

À semelhança do EvoSuite, o PESTT é uma ferramenta que pode ser utilizada em conjunto com o Eclipse IDE e em projetos Maven, mas no caso do PESTT é necessário que o utilizador escreva os testes unitários para cobrir os requisitos de teste gerados automaticamente pelo PESTT.

3.1.5 JUnit

O JUnit [9] é uma ferramenta para a execução de testes unitários para Java, que tem sido importante no âmbito de *Test-Driven Development* — técnica de desenvolvimento de *software* que se baseia num ciclo curto de repetições: o programador escreve um caso de teste que define uma melhoria ou funcionalidade, sendo depois produzido código que torna o teste válido. Esta ferramenta de testes, *open source*, é da família das ferramentas coletivamente conhecidas como xUnit, e serve-se de anotações para identificar os métodos que especificam testes.

Com o JUnit é possível usufruir de uma interface gráfica (GUI) que facilita e agiliza a escrita e execução de testes de código fonte, isto porque é possível observar-se o progresso da execução dos testes através de uma barra que fica verde ou vermelha consoante o resultado dos testes efetuados. É apresentada uma lista dos testes que falharam, comparando os valores esperados com os obtidos. O JUnit é uma ferramenta de fácil utilização, o que possibilita a correção dos defeitos à medida que são detetados.

Tal como o JUnit, o foco do PESTT é em testes unitários em Java, usufruindo de uma interface gráfica (GUI). No caso do PESTT, esta interface é essencial para facilitar e agilizar a escrita de testes baseados em Grafos de Fluxo de Controlo e identificando os caminhos de testes e a cobertura de testes executados. Por outro lado, enquanto o JUnit é uma ferramenta com foco na execução e interpretação de resultados de testes unitários, esta versão do PESTT tem como foco a geração automática de requisitos de testes através da interacção com o utilizador, sendo a sua escrita, execução e análise da responsabilidade do utilizador.

3.2 Reconhecedores

3.2.1 Jtest

O Jtest [9] é uma solução comercial que pertence ao grupo de plataformas xUnit, podendo também ser usado como gerador. O Jtest foca-se na validação de aplicações Java com vista ao melhoramento da produtividade de equipas e aumento da qualidade do *software*. Em particular, o Jtest permite a execução de testes unitários, análise estática de código e deteção de falhas em tempo de execução. Em termos das caraterísticas comuns às duas ferramentas, nomeadamente análise estática de código, o PESTT disponibiliza visualizações do grafo de fluxo de controlo, bem como critérios de cobertura não disponíveis no Jtest.

3.2.2 Clover

O Clover [9] é um produto comercial, também disponibilizado de forma gratuita para projetos *open source* ou instituições sem fins lucrativos, que fornece métricas com vista a equilibrar o esforço inerente à escrita de código aplicacional e código de teste. O Clover pode ser executado nos ambientes de desenvolvimento Eclipse ou IntelliJ, ou em ambientes de integração contínua, enquanto o PESTT apenas pode ser executado no Eclipse IDE.

Como funcionalidades de anotação, o Clover inclui anotações visuais baseadas em cores para cobertura de teste, testes de cobertura linha a linha a partir dos ficheiros de código fonte, e informação sobre a execução com sucesso de testes recentes. Por sua vez, o PESTT também inclui anotações visuais baseadas em cores e informação de cobertura linha a linha, mas neste caso extraída a partir da estática do código fonte. O PESTT consegue ainda fazer uma representação gráfica do código, uma funcionalidade que não existe no Clover.

3.2.3 Google CodePro Analytix

Google CodePro Analytix [9] é uma ferramenta *open source* para a análise de cobertura de código. A ferramenta fornece um conjunto rico de funcionalidades, incluindo análise de código, diferentes tipos de métricas sobre o código, possibilidade de geração de testes JUnit, edição de testes JUnit, critérios de cobertura de código, análise de dependências, análise de semelhança de código, entre outras.

O *Google CodePro Analytix* é uma das ferramentas de eleição para programadores que usam o Eclipse IDE. As funcionalidades em comum com o PESTT são a cobertura de código e as métricas. No entanto, o PESTT fornece um maior conjunto de critérios de análise (que vão para além dos critérios de cobertura baseados em nós e arestas) disponibilizados no *Google CodePro Analytix*, enquanto o *Google CodePro Analytix* tem uma maior variedade de funcionalidades, como por exemplo, análise de dependências, análise de código semelhante e manutenção de Javadoc.

3.2.4 EclEmma

O EclEmma [9, 13] é uma ferramenta *open source* de cobertura de código para Java que usa informação gerada pela execução de testes JUnit. O EclEmma oferece funcionalidades como o realce de código fonte, a importação de dados e a exportação de relatórios. É um *plug-in* gratuito, disponível para o Eclipse, inspirado na biblioteca EMMA, desenvolvida por Vlad Roubtsov. A partir da versão 2.0, o EclEmma passou a ter como base o JaCoCo (biblioteca de cobertura de código Java).

Em comparação, o PESTT fornece um conjunto de funcionalidades também oferecidas pelo EclEmma, nomeadamente, análise detalhada de cobertura e métricas sobre o código fonte. O PESTT é mais completo em termos dos critérios de cobertura que fornece, bem como da componente de visualização do grafo de fluxo de controlo. Por outro lado, o EclEmma permite gerar relatórios sobre sessões de cobertura, isto é, informação sobre a cobertura de código de um programa em particular, classes consideradas, e detalhes da análise de cobertura.

Capítulo 4

Transformação da Ferramenta

4.1 Estado anterior do PESTT

Para facilitar o processo de desenvolvimento da interface gráfica do PESTT, nomeadamente a atualização dinâmica dos seus elementos gráfico (*e.g.*, nós dos grafos, pedaços de código correspondentes), a versão anterior recorreu à implementação do padrão MVC utilizando o padrão *Observer* [9].

4.1.1 Padrão *Observer*

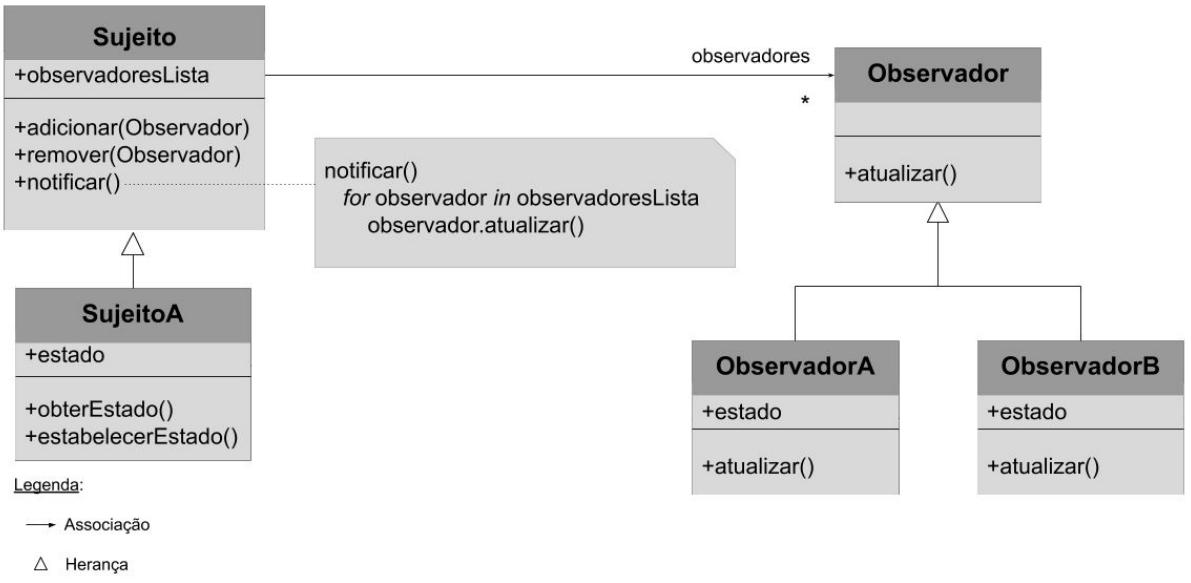

Figura 4.1.: Diagrama de classes do padrão *Observer*.

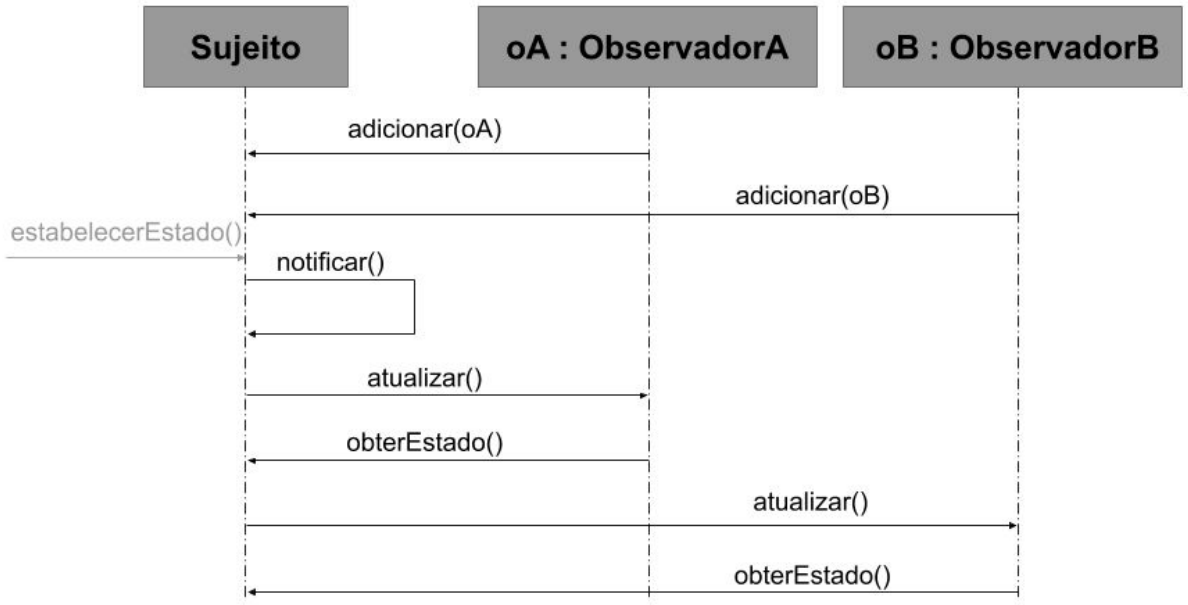

Figura 4.2.: Diagrama de sequência do padrão *Observer*.

O *Observer* é um padrão de s*oftware* orientado a objetos que materializa uma parte central do padrão *Model-View-Controller*. O padrão *Observer* surgiu como resposta a um problema comum do particionamento de um sistema num conjunto de classes cooperantes: a necessidade de manter a consistência entre objetos relacionados, sem tornar as classes fortemente acopladas, pois isso reduz a sua reutilização [35]. Os objetos-chave neste padrão são o *Sujeito* e o *Observador*. Um *Sujeito* mantém um estado, conhece os seus *Observadores* e é responsável por fornecer uma interface para anexar e desanexar objetos *Observador*. Um *Sujeito* pode ter um qualquer número de *Observadores* interessados em receber notificações sempre que o estado do *Sujeito* é alterado. Um *Observador* define uma interface de atualização para objetos que será utilizada sempre que ocorre uma alteração no estado do *Sujeito*.

A principal vantagem no uso deste padrão é o desacoplamento entre objetos que interagem entre si. Para além disso, permite o envio de dados para outros objetos de forma eficaz, sem qualquer alteração nas classes *Sujeito* ou *Observador* quando se adicionam ou removem observadores.

Por outro lado, a classe *Observable* da biblioteca padrão do *Java* força o uso de herança (ao invés da programação de uma interface). Se não for implementado de forma correcta, o *Observer* pode adicionar complexidade e levar a problemas de desempenho inadvertidos, e a ordem das notificações do *Observador* não é confiável, o que pode resultar em condições de corrida ou inconsistência.

O modelo de comunicação entre *Sujeitos* e *Observadores* pode ser de natureza ***push*** ou ***pull***. Na comunicação ***push*** os *Observadores* são informados, pelos *Sujeitos*, que ocorreu uma alteração. A grande vantagem desta natureza é o baixo acoplamento entre O*bservadores* e o *Sujeitos*. Por outro lado, como os *Sujeitos* necessitam de enviar dados para os *Observadores*, pode tornar-se confuso no caso de haver grandes quantidades de informação de diferentes tipos. No modelo de comunicação ***pull*** mantém-se a responsabilidade dos *Sujeitos* em notificar os *Observadores* aquando da alteração de dados do seu interesse. Porém, neste caso, o *Sujeito* partilha todo o objeto que sofreu alterações, sendo da responsabilidade do *Observador* extrair os detalhes que lhe dizem respeito. Este mecanismo é mais flexível, uma vez que cada *Observador* pode decidir por si mesmo o que extrair, sem colocar no *Sujeito* a responsabilidade de enviar apenas a informação de interesse para o *Observador*. Por outro lado, com este mecanismo, é necessário que o *Observador* tenha conhecimento da interface do *Sujeito*, para que possa extrair a informação corretamente do objeto que foi partilhado na totalidade [26].

4.2 Comunicação entre componentes no PESTT — Requisitos

Para além da adaptação do PESTT à nova versão gráfica do *Eclipse*, temos por objetivo, decorrente dos princípios gerais da Engenharia de *Software*, aumentar o nível de modularidade desta aplicação, reforçando a separação de responsabilidades (do inglês *Separation of Concerns (SoC)* – princípio de desenho de *software* que separa um programa em secções distintas em função das suas funcionalidades) para que cada secção seja responsável por um conjunto de informação.

Para cumprir o principal objetivo desta tese, foi necessário perceber que a alteração da versão 3 para a versão 4 do *Eclipse* foi a forma de representação da interface do utilizador. No *Eclipse* 4, a extensibilidade do *workbench* – ambiente de desenvolvimento – foi implementada a partir do conceito de *model fragments*. Um *model fragment* é um pequeno modelo aplicacional que define os elementos que necessitam de ser adicionados ao modelo aplicacional base. Neste caso, os autores de *plug-ins* fornecem os seus modelos aplicacionais que posteriormente o *workbench* vai fundir no modelo aplicacional base [44].

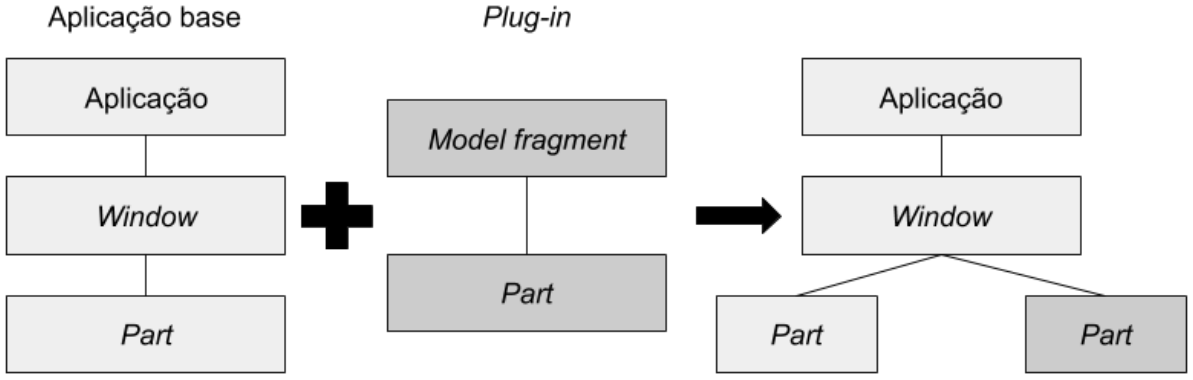

Figura 4.3.: Fusão de um *model fragment* de um *plug-in* no modelo aplicacional base.

Uma vez alterada a interface do utilizador para que esta passe a ser baseada em modelos, faz sentido converter os pontos de extensão (*ui extension points*) em fragmentos de modelo (*model fragments*). Deste modo, uma *view* pode ser descrita como um fragmento de modelo que contém um *part descriptor*, ao invés de uma extensão ao ponto de extensão *org.eclipse.ui.views*. Um *part descriptor* é um modelo para a criação de uma *part*. Por

omissão, o meio utilizado para persistir os fragmentos de modelo é um ficheiro XMI [43,44]. Para refazer a camada de *UI,* de modo a cumprir o segundo objetivo mencionado no início desta secção, optámos por refazer a comunicação entre componentes com base em injeção de dependências, substituindo assim o padrão *Observer* para a comunicação entre modelos e vistas.

Esta necessidade de refazer a camada de *UI* do PESTT resultou na criação, a partir do zero, de todas as *views*, *commands* e *handlers*, bem como dos seus modelos e meios de comunicação entre estes e as *views*. Todo o sistema baseado no padrão *Observer* foi removido. No total, entre a criação de código necessário à implementação dos *model fragments* e a criação/alteração de código de modo a suportar a comunicação entre modelos e *views* com base em injeção de dependências, foram criadas/alteradas um total de 27 classes *java*. É de notar que estas alterações resultaram numa redução do código destas classes, sendo também esse um ponto favorável a apontar no que toca à decisão de substituir a comunicação entre modelos e *views* à luz da injeção de dependências. Para além disso, este novo modelo permite uma interação mais simples entre objetos e a eliminação de bugs que resultavam do registo dos objetos como *Observadores*. O conceito de injeção de dependências será descrito na secção seguinte, bem como os conceitos de *part*, *view*, *command*, *handler* e *window*, entre outros. Também na secção seguinte serão apresentados exemplos dos modelos acima mencionados.

4.2.1 Injeção de Dependências

A injeção de dependências (do inglês *Dependency Injection*) é um padrão de desenho de *software* utilizado quando é necessário manter baixo o nível de acoplamento entre diferentes módulos de um sistema. Nesta solução as dependências entre módulos são definidas pela configuração de uma infraestrutura de *software* chamada *container*. Um *container* é um objeto que contém outros objetos que podem ser incluídos ou removidos em tempo de execução. É da responsabilidade do *container* injetar em cada componente as suas dependências declaradas. Uma dependência é um objeto que pode ser usado (no fundo, é um serviço), e uma injeção é a passagem da dependência para um objeto dependente (o cliente do serviço) que a pode usar.

A injeção de dependências é uma instância de uma família alargada de técnicas de inversão do fluxo de controlo de um programa, comparativamente ao fluxo tradicional, designada por *Inversão de Controlo (IoC)*. *IoC* é o nome dado ao padrão de desenvolvimento de *software*, onde a sequência (controlo) de chamadas dos métodos é invertida em relação à programação tradicional, não sendo determinada diretamente pelo programador da aplicação, mas sim pelo *container*. Um fluxo normal de execução acontece quando um determinado programa cria chamadas a outros métodos e assim sucessivamente, deixando a criação dos componentes, o início da execução e o fim da execução sob controlo do programador. A inversão de controlo ocorre quando, ao invés de se criar explicitamente um código, ou acompanhar todo o ciclo de vida de uma execução, o programador delega alguma dessas funcionalidades no *container* [18].

Os objetos injetados podem ser **atributos** de uma classe, parâmetros de um **construtor** ou parâmetros de um **método** e, no caso mais simples, a injeção é acionada pela anotação ***@Inject*** e distinguida pelo tipo de objeto declarado para injetar. Os construtores devem incluir parâmetros essenciais para estabelecer a invariante do objeto. Parâmetros desnecessários "limitam" a capacidade de reutilização de um objeto. Um exemplo típico de injeção de dependências no construtor é a injeção do *composite*[1] de uma *View*. Como as visualizações são inicializadas num determinado contexto do modelo aplicacional, a especificação do *composite* neste caso é clara e nenhuma anotação adicional é necessária. Após a execução do construtor da classe, os atributos da classe são injetados. Uma aplicação típica é a injeção de serviços que estarão disponíveis globalmente na classe. Um exemplo disso é o Serviço de Seleção para definir a *View* que está atualmente selecionada. Depois do construtor e dos atributos, ao inicializar uma classe, todos os métodos anotados (com *@Inject*) são executados sequencialmente. Se um dos parâmetros injetados de um método for alterado posteriormente, o método será chamado novamente com os novos parâmetros. Um bom exemplo de uma injeção em métodos é a seleção atual. Sem mais informações, a pesquisa para o objeto correto será de acordo com o tipo correspondente do parâmetro ou campo. Para

[1] *Composite* é um padrão que descreve um grupo de objetos que é tratado da mesma maneira que uma única instância do mesmo tipo de objeto. A intenção de um *composite* é "compor" objetos em estruturas de árvore para representar hierarquias de partes inteiras [49].

injetar um objeto específico identificado por uma *string*, utiliza-se a anotação ***@Named***. Os serviços disponibilizados pelos Eclipse 4 incluem automaticamente objetos identificados por constantes:

- *ACTIVE_CONTEXTS:* conjunto de contextos actualmente ativos;
- *ACTIVE_PART: Part* ativa num determinado contexto. O conceito de *part* será abordado na secção 4.2.2.;
- *ACTIVE_SELECTION:* seleção atual;
- *ACTIVE_SHELL: Shell* ativa num determinado momento. A *shell* é a *window* que se vê no ecrã.

A figura seguinte exemplifica um caso real de utilização destas anotações no PESTT. Este caso ilustra a injeção do parâmetro de um método (anotado com *@Inject*) com o objeto de seleção atual, que é identificado pela constante *ACTIVE_SELECTION* (anotado com *@Named*) para a obtenção do critério de cobertura selecionado pelo utilizador. Este exemplo utiliza também a anotação ***@Optional*** para prevenir que o mecanismo de injeção de dependências do Eclipse 4 lance um erro quando não há uma seleção está no contexto que ocorre, por exemplo, quando a aplicação é iniciada.

```
    @Inject
    @Optional
    public void selectedCoverage(@Named(IServiceConstants.ACTIVE_SELECTION)
    CoverageCriteriumSelection selection) {
        if (selection != null) {
            GraphCoverageCriteriaId criterium = selection.getCriterium();

            GraphAccessLayer gal = GraphAccessLayer.getInstance();

            gal.cleanUpInformation();

            Set<AbstractPath<Integer>> testRequirementSet =
                    gal.getTestRequirements(criterium);

            if(testRequirementSet != null) {
                this.testRequirementViewer.setViewerInput(testRequirementSet);
            }
```

Figura 4.4.: Utilização das anotações *@Inject*, *@Named* e *@Optional* no PESTT.

Resumindo e concluindo, a ordem de injeção desempenha um papel crucial. Quando uma classe é instanciada, os parâmetros do construtor são injetados primeiro. Imediatamente a seguir, os atributos relevantes são injetados. Como consequência, os atributos a injetar não podem ser acedidos através do construtor. Os parâmetros dos métodos são injetados quando

esses métodos são chamados. Todos os métodos marcados com @Inject também são chamados para inicializar o objeto após o construtor e os atributos. Se o objeto injetado mudar de contexto, ele será reinjetado. Os métodos são, portanto, chamados novamente quando os valores injetados mudam.

Para além das anotações acima referidas, existem outras anotações disponíveis no Eclipse 4, cuja utilização é semelhante, mas com comportamento distinto. Algumas dessas anotações são:

- @PostConstruct: um método anotado com @PostConstruct é chamado após a classe ser inicializada e após todos os atributos terem sido injetados;
- @PreDestroy: um método anotado com @PreDestroy é chamado antes do objeto ser destruído, por exemplo, quando a *View* correspondente é fechada;
- @Focus: um método anotado com @Focus é chamado quando o elemento de UI correspondente recebe o foco. Esta anotação é utilizada em contexto de elementos visuais, por exemplo, em *Parts*;
- @Persist: um método anotado com @Persist é chamado quando é necessário persistir alguma informação numa *Part.* Por exemplo, se a *Part* em questão representar um editor de texto, o seu conteúdo é gravado num ficheiro.

4.2.2 Elementos do modelo da interface de utilizador

No Eclipse 4, as aplicações têm como base o modelo aplicacional do Eclipse, que é uma descrição abstrata da estrutura de uma aplicação, e contém os elementos visuais e não-visuais da aplicação. Os elementos visuais são, por exemplo, *windows*, *parts*, *perspectives* e *menu contributions*. Os elementos não-visuais são, por exemplo, componentes como *handlers* e *commands* [39]. Os referidos elementos serão descritos abaixo:

Window

As aplicações Eclipse consistem em uma ou mais *windows*. Tipicamente, uma aplicação tem apenas uma *window*, mas este facto não representa uma obrigatoriedade [39].

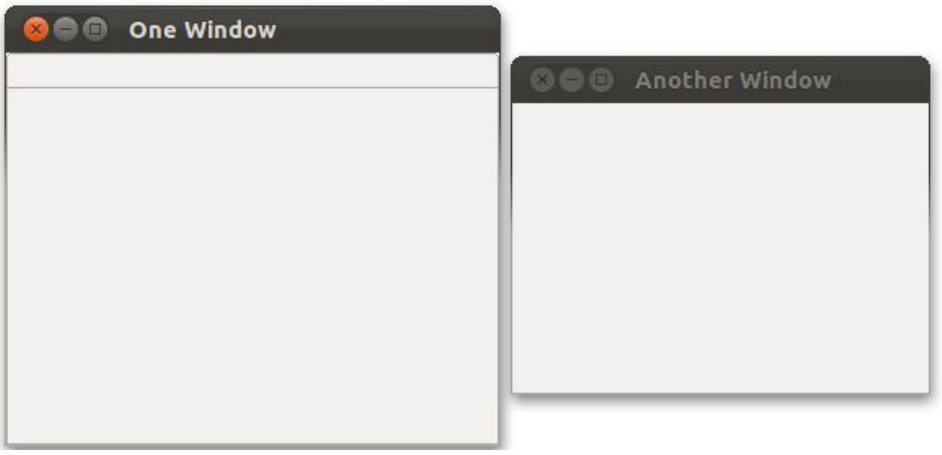

Figura 4.5.: Exemplo de *windows* de uma aplicação Eclipse (adaptado de [39]).

Parts

Parts são componentes de interface de utilizador que permitem navegação e alteração de dados, e podem ser classificadas como ***views*** e ***editors***. Uma ***view*** é, tipicamente, utilizada para trabalhar numa estrutura hierárquica. Isto significa que, se os dados sofrerem alterações pela *view*, essas alterações são diretamente aplicadas na estrutura de dados hierárquica. Por exemplo, a *view* do *Package Explorer* permite a pesquisa de ficheiros de projetos Eclipse. Se algum destes ficheiros for renomeado a partir da *view* do *Package Explorer,* essa alteração também estará refletida no sistema de ficheiros. Os ***Editors*** são tipicamente usados para modificar o conteúdo de ficheiros. Para que essas modificações persistam, o utilizador tem de explicitamente gravar as alterações. Por exemplo, o editor *Java* é utilizado para alterar ficheiros fonte, mas para que essas alterações não sejam descartadas é necessário que seja selecionado o botão *Save* [39].

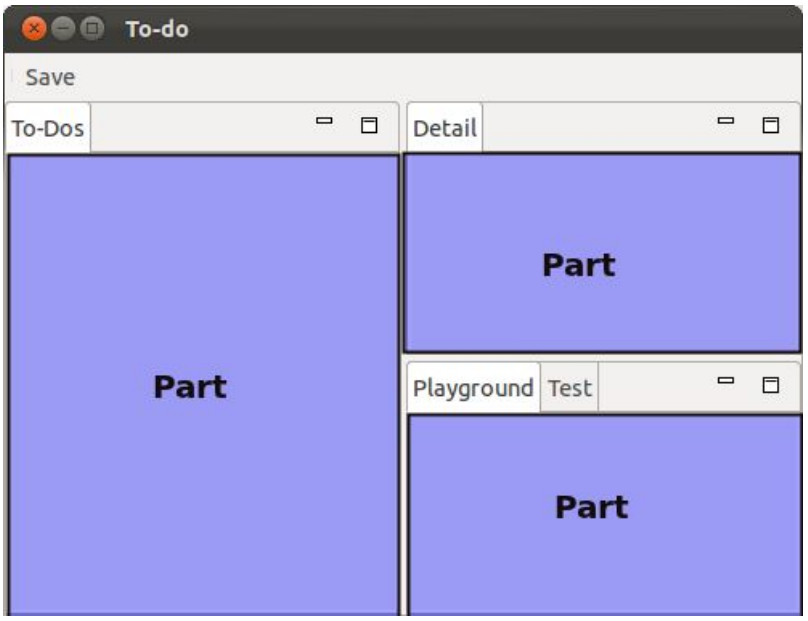

Figura 4.6.: Exemplo de *parts* de uma aplicação Eclipse (adaptado de [39]).

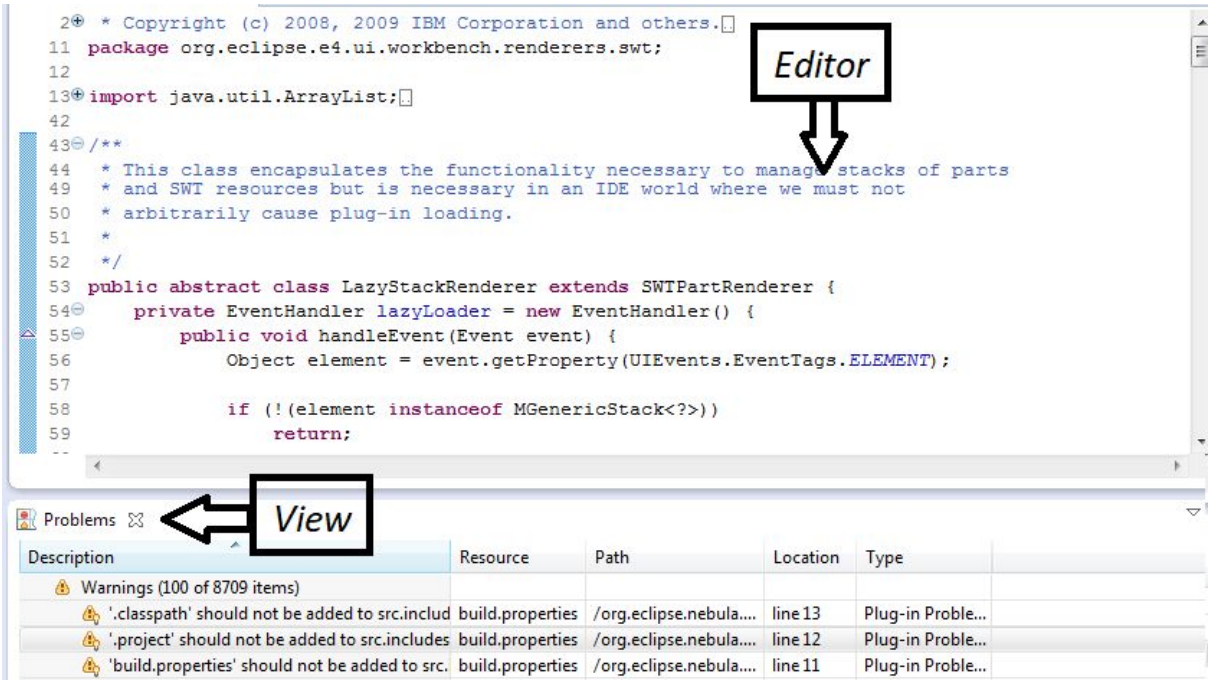

Figura 4.7.: *Editor* para a linguagem *Java* (em cima) e de uma*View* para problemas detetados (em baixo).

Perspectives

Uma *perspective* define o conjunto inicial de *parts* numa *window*. Dentro de uma *window*, cada *perspective* partilha o mesmo conjunto de *editors*. Cada *perspective* fornece um conjunto de funcionalidades para um tipo específico de tarefas com um tipo específico de recursos. Por exemplo, a *Java Perspective* combina *parts* que são utilizadas durante a edição de ficheiros *Java*, enquanto que a *Debug Perspective* contém as *parts* utilizadas no âmbito da depuração de programas *Java*. As *perspectives* controlam o que aparece em certos *menus* e *toolbars*. São as *perspectives* que definem o conjunto de ações visíveis, as quais podem ser alteradas para personalizar a *perspective* [38].

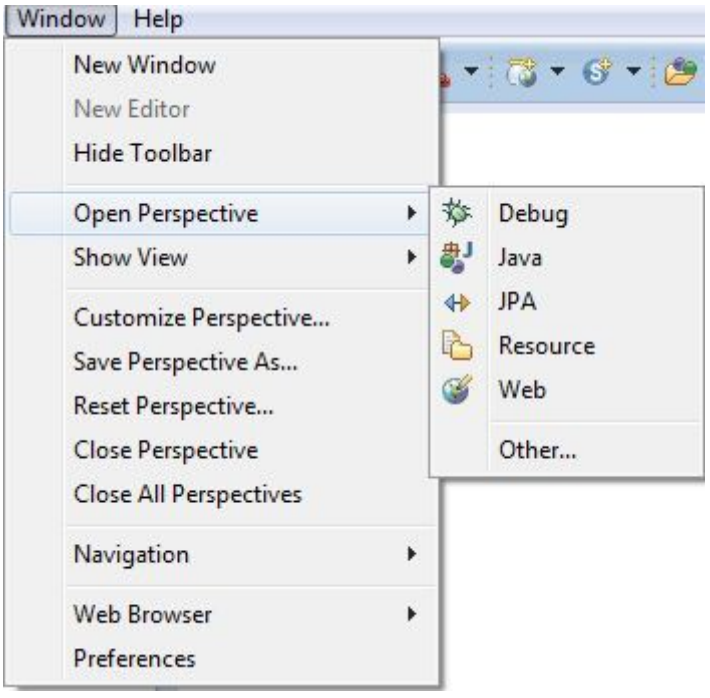

Figura 4.8.: Exemplo de *perspectives* numa aplicação Eclipse (adaptado de [42]).

Menu Contributions

Menu contributions são alterações (adicionar e esconder elementos) feitas aos menus existentes, nomeadamente quanto às suas entradas [37].

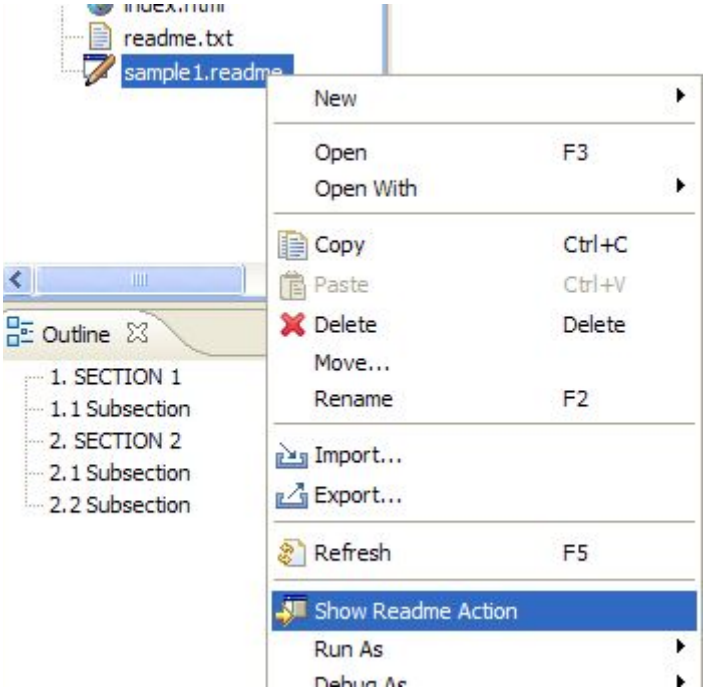

Figura 4.9.: Exemplo de uma *menu contribution* numa aplicação Eclipse (adaptado de [45]).

Handlers

Handlers são componentes acionados por botões da aplicação, e definem o comportamento específico a ser desencadeado por estes. Na versão 4 do Eclipse, é permitida a separação entre os *handlers* e a *framework* em si, o que promove a reutilização e testabilidade [47, 48].

Os *Handlers* podem ser criados em três níveis distintos: na própria aplicação, ao nível da *window* ou para uma *part* específica. Se uma *part* específica está em foco, os seus *handlers* serão ativados. Se só existe uma implementação para um *handler*, este pode ser criado ao nível da aplicação. Se existem várias implementações para um *handler*, consoante a *window*, a criação do *handler* deve ser feita no respetivo elemento [47].

Para além de possibilitar a definição de um comportamento específico a ser desencadeado por um botão, os *handlers* também possibilitam a re-definição de comportamentos a serem desencadeados por botões já existentes na aplicação. Tomando como exemplo o comportamento do botão *close* de uma *window*, por omissão, o objeto *IWindowCloseHandler* é o responsável pelo comportamento desencadeado quando um elemento do modelo *MWindow* é fechado, questionando o utilizador se pretende guardar o conteúdo do editor antes de fechar. A figura seguinte mostra como se pode alterar o comportamento por omissão desencadeado pelo botão *close* de uma *window*, através da implementação de um *handler* que vai ser sobreposto ao original em tempo de execução [55].

```
@Execute
public void execute(final Shell shell, EModelService service, MWindow window) {
    IWindowCloseHandler handler = new IWindowCloseHandler() {
        @Override
        public boolean close(MWindow window) {
            return MessageDialog.openConfirm(shell,
            "Close",
            "You will loose data. Really close?");
        }
    };
window.getContext().set(IWindowCloseHandler.class, handler);
}
```

Figura 4.10.: Exemplo de implementação de um *handler* (adaptado de [55]).

Utilizando a anotação *@Execute*, o *handler* informa a *framework* Eclipse 4 de qual o método a executar. Esta anotação tem um comportamento semelhante à anotação *@Inject* (introduzida na secção 4.2.1.) na medida em que os parâmetros necessários ao método anotado serão injetados pela *framework* [48].

A forma mais simples de integrar um *handler* com os botões da aplicação (os *items*) é através de um *Direct ToolItem*. Apesar de ser a forma mais simples e rápida de integrar *handlers*, para promover a flexibilidade e reutilização, recomenda-se a utilização de um outro componente: *commands*. Este componente vai ser apresentado com mais detalhe já de seguida [48].

Commands

A responsabilidade de regular a ligação entre os botões que irão desencadear uma ação visível – os *items* – e a implementação desta ação – os *handlers* – é delegada ao próprio Eclipse. Assim sendo, é o próprio Eclipse que define o modo como esta interação é feita, surgindo assim, o conceito de ***command***. Um *command* é a representação lógica de uma operação [47].

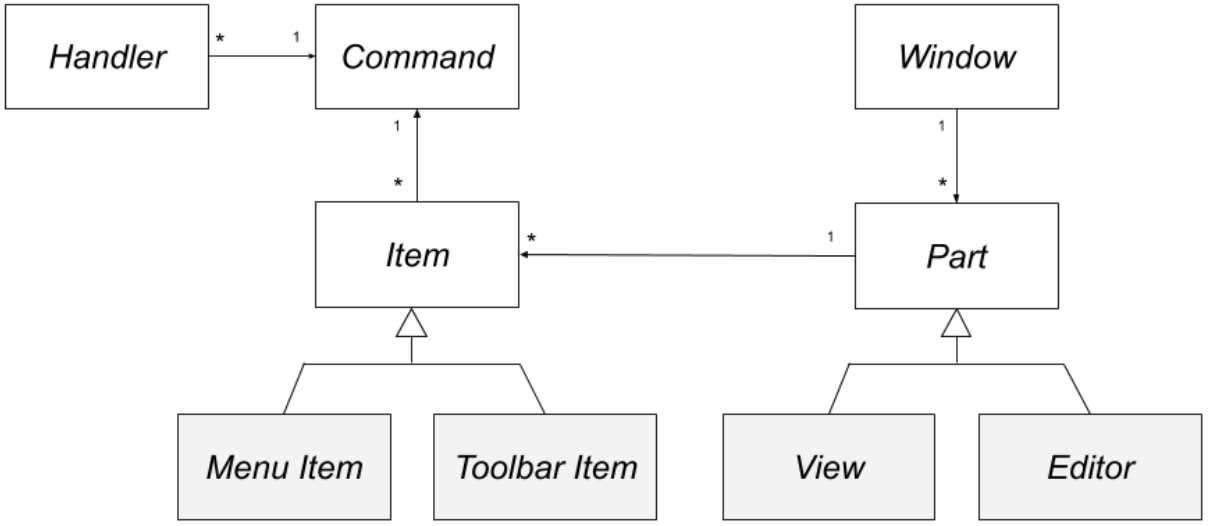

Figura 4.11.: Interação entre *handlers*, *commands*, *items*, *parts* e *windows* (adaptado de [48]).

Tal como ilustrado na figura 4.11., é possível ligar vários *items* ao mesmo *command*, para, por exemplo, mostrar simultaneamente o resultado de uma ação num *menu* e numa *toolbar*. Também é possível ligar vários *handlers* ao mesmo *command*, possibilitando implementações alternativas de uma mesma ação. A plataforma Eclipse 4 vai, automaticamente, ativar os *handlers* no contexto atual da aplicação. Por exemplo, se uma certa *part* está ativa (em foco), os seus *handlers* serão ativados [47, 48].

As figuras 4.12., 4.13. e 4.14. mostram, na prática, como se processa a criação de um *command* (primeira imagem), de um *handler* (segunda imagem), que serão depois utilizados para adicionar um *item* (*menu contribution*) de modo a executar uma determinada operação (terceira imagem).

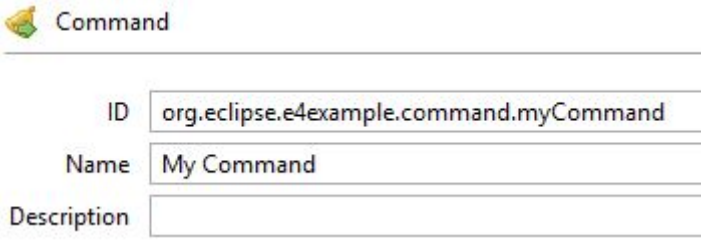

Figura 4.12.: Exemplo da criação de um *command* no Eclipse 4 (adaptado de [48]).

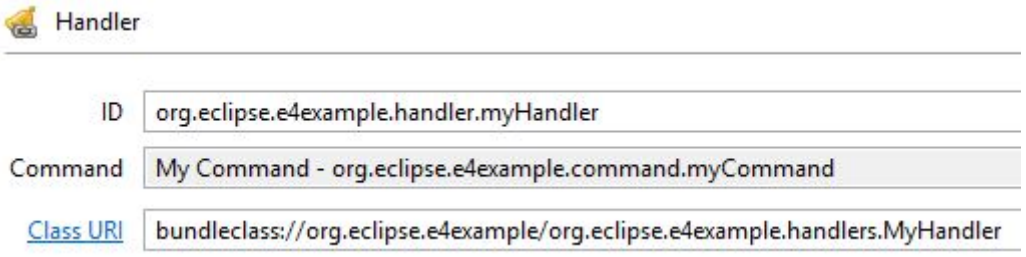

Figura 4.13.: Exemplo da criação de um *handler* no Eclipse 4 (adaptado de [48]).

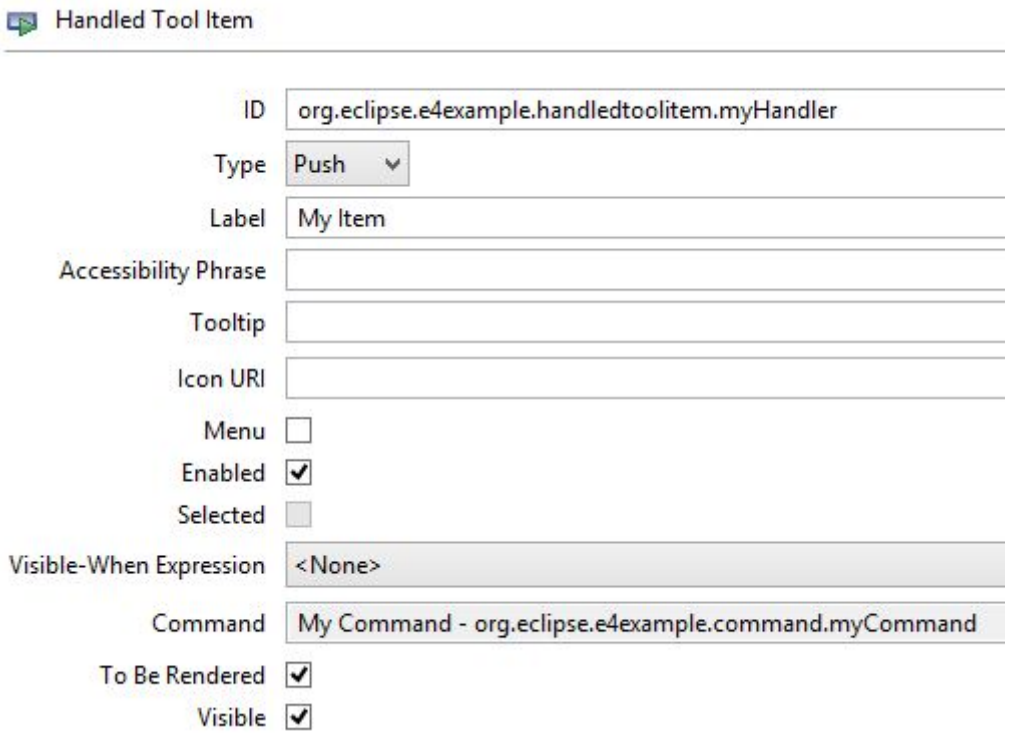

Figura 4.14.: Exemplo da criação de um *item* no Eclipse 4 (adaptado de [48]).

Como já foi brevemente referido na secção 4.2, a base do modelo aplicacional é tipicamente descrita num ficheiro estático do tipo XMI que, para as aplicações *RCP*[2] é, por omissão, chamado *Application.e4xmi*. Este ficheiro é lido no carregamento da aplicação e é usado para construir o modelo aplicacional inicial. Qualquer alteração feita é persistida e reaplicada no carregamento da aplicação [39, 41].

No âmbito do PESTT, a camada de interface de utilizador foi feita a partir do zero para estar em conformidade com a introdução de *model fragments*, o que resultou na alteração/criação de 30 classes *java*. Foi, então, criado o seguinte ficheiro *fragment.e4xmi*:

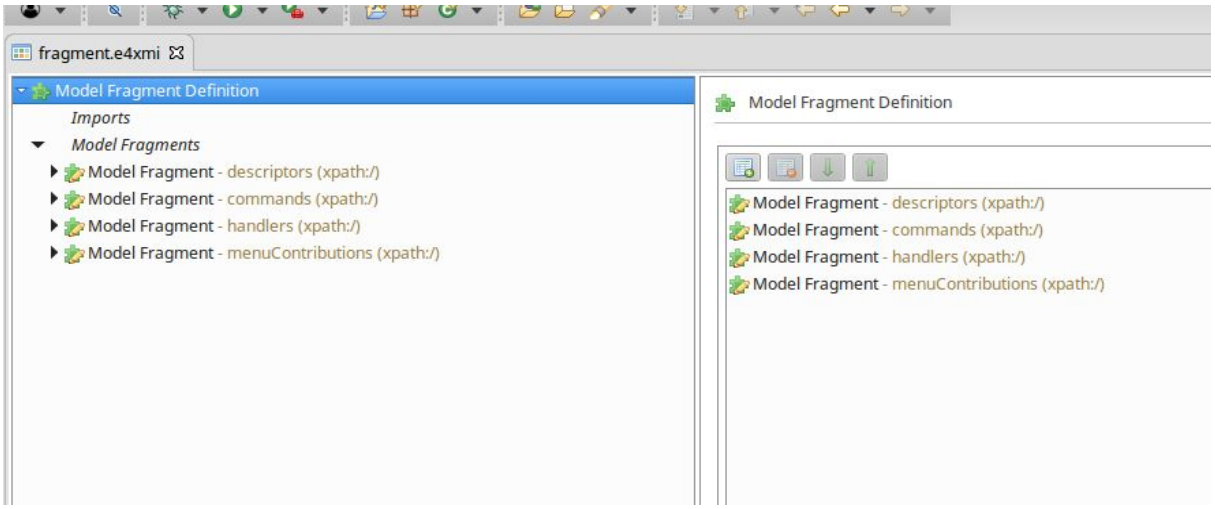

Figura 4.15.: Visão lógica do ficheiro *.e4xmi* criado no âmbito do PESTT.

Como se pode ver, o ficheiro *fragment.e4xmi* do PESTT é composto pelos seguintes *model fragments*: *part descriptors*[3], *commands*, *handlers* e *menu contributions*. Cada um destes *model fragments* tem o seguinte conteúdo:

[2] *RCP* do inglês *Rich Client Platform* é o nome dado a um conjunto de *plug-ins* necessários ao desenvolvimento de qualquer aplicação Eclipse cliente [40].
[3] Um *descriptor* armazena todas as informações que descrevem como uma instância de um determinado objeto pode ser representada numa fonte de dados [53].

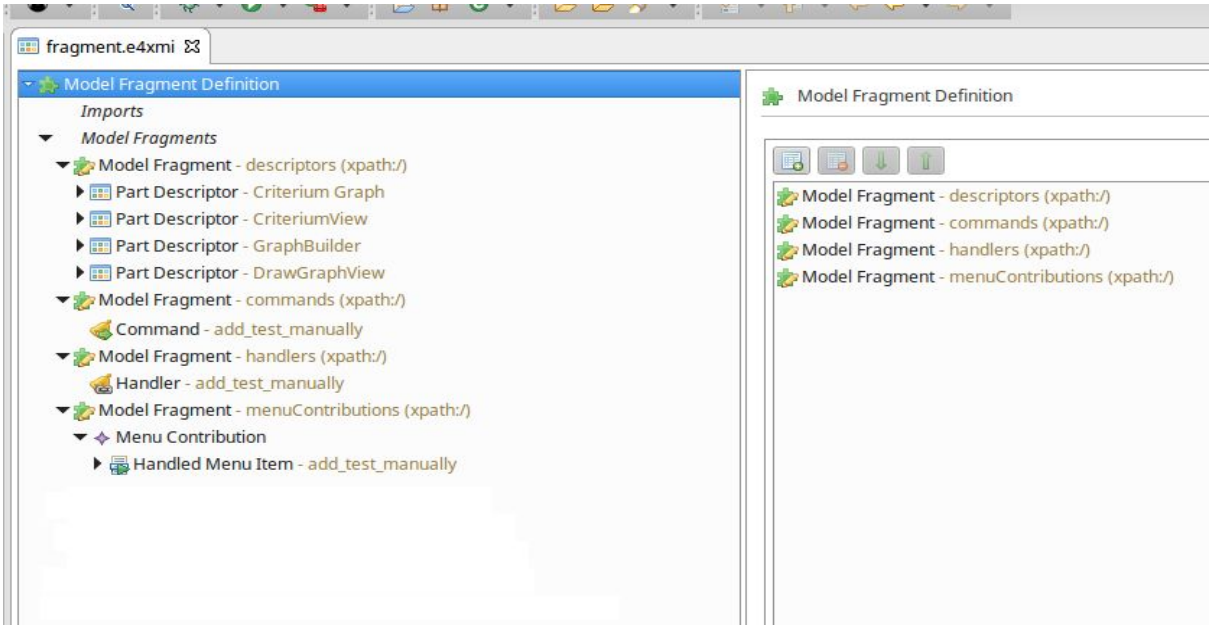

Figura 4.16.:*Models fragments* criados no âmbito do PESTT.

Como já detalhado nas figuras 4.12., 4.13., e 4.14., a criação destes *model fragments* é feita à custa do preenchimento de campos essenciais no ficheiro *fragment.e4mxi*, como *ids* e *URI* de classes *Java*. Na figura 4.17., apresentamos um exemplo da criação de uma *part/view*, neste caso a *view* responsável pela visualização do grafo de fluxo de controlo:

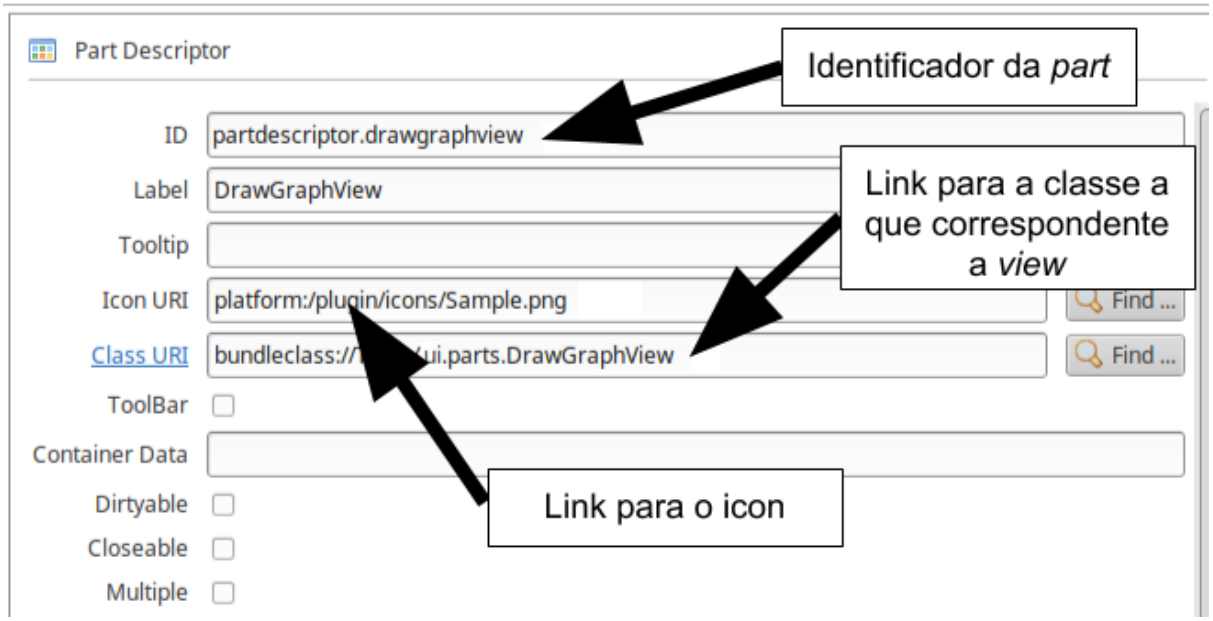

Figura 4.17.: Criação de uma *part* no âmbito do PESTT.

4.3 Arquitetura do PESTT

O PESTT está dividido em quatro pacotes *Java*. O primeiro pacote, chamado ***main***, é a parte responsável pela ativação do *plug-in* e por expôr a interface do PESTT ao Eclipse IDE. O pacote responsável por encapsular conceitos relacionados com grafos e caminhos é chamado ***adts***. Esta é a estrutura de dados base de todo o projeto. Acima dos *adts* temos o pacote *core* do *plug-in* onde são definidos os algoritmos. Este pacote é chamado ***domain***. Por último, o pacote pacote ***ui*** é responsável pela interface gráfica para as estruturas definidas no *domain.* É neste pacote que estão definidos os mecanismos de interação com o utilizador, tornando possível a inserção e a consequente representação do resultado. Este pacote encapsula as decisões relacionadas com interface de utilizador e elementos gráficos.

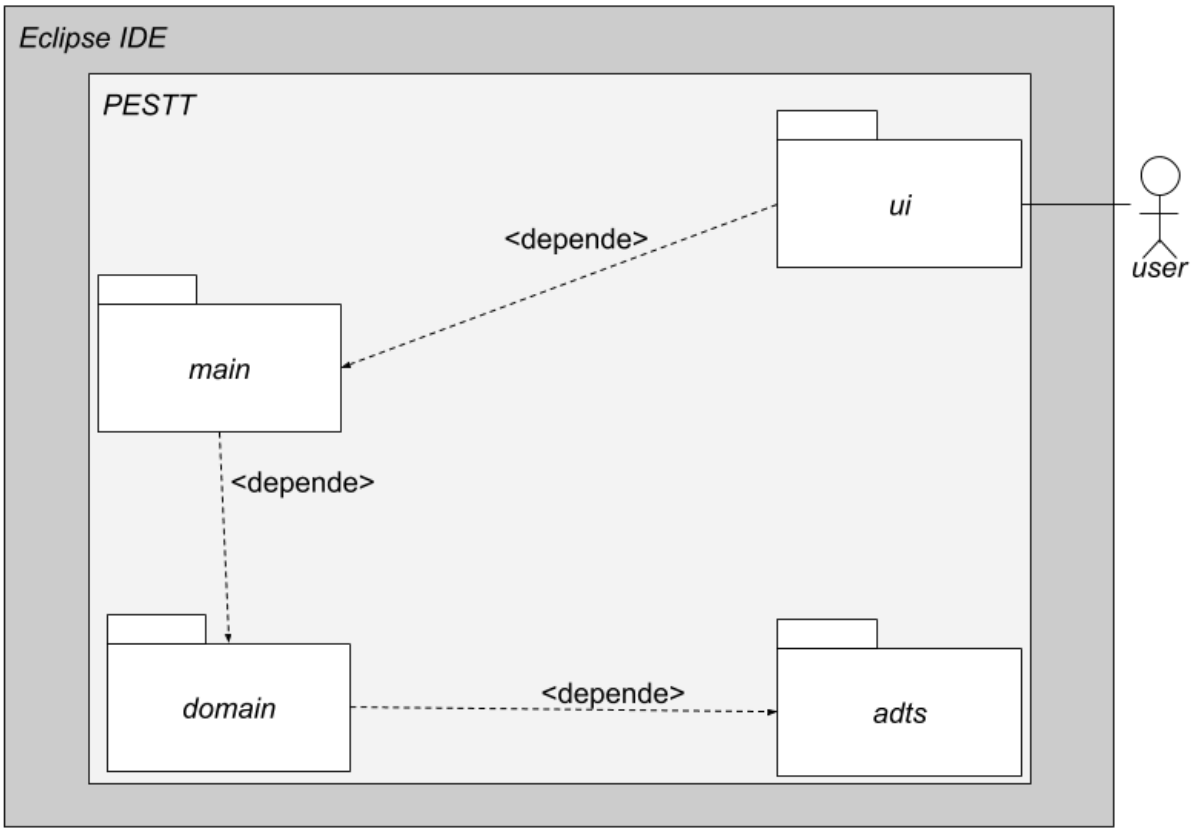

Figura 4.18.: Diagrama de pacotes do PESTT.

Capítulo 5

Validação do PESTT

A primeira coisa que foi dita relativamente a este trabalho é que produzir *software* com qualidade e que satisfaça os requisitos funcionais é o objetivo principal da Engenharia de *Software*. Sendo este um projeto de engenharia de *software* é, portanto, primordial verificar que o seu principal objetivo foi cumprido: desenvolver uma ferramenta que tenha qualidade.

Também o facto do PESTT ser uma ferramenta que visa auxiliar nos testes a outras aplicações, faz com que seja inequívoca e de extrema necessidade a execução de testes sobre o próprio PESTT. Assim sendo, neste capítulo, o tema em foco será a qualidade do PESTT e os meios utilizados para efetuar os testes à ferramenta.

5.1 Utilizar o PESTT para testar o PESTT

Sendo o PESTT uma ferramenta de verificação e validação de *software*, seria tentador considerar a hipótese de efetuar testes de *software* ao PESTT, utilizando o próprio PESTT. Contudo, esta abordagem não foi seguida, dada a dependência circular que se estaria a criar.

Consideremos os seguintes cenários:

1) O PESTT está perfeitamente correto e é executado sobre si próprio, não se detetando qualquer erro.

2) O PESTT está perfeitamente correto e é executado sobre si próprio, detetando-se pelo menos um erro – esta situação é contraditória, pelo que é impossível.

3) O PESTT tem erros e é executado sobre si próprio, não se detetando qualquer erro. Esta situação é indistinguível da primeira, pelo que, nem uma, nem a outra, acrescentam valor.

4) O PESTT tem erros, é executado sobre si próprio e são detetados erros. À partida, tendo o PESTT erros, também esta situação não acrescenta valor, pois o resultado da sua execução não é fiável.

A descrição acima tem por base o paradoxo do mentiroso [54], a partir do qual se chega a uma contradição ao tentar atribuir um valor verdade a uma afirmação. Daqui, logicamente, se conclui a necessidade de efetuar testes ao PESTT, recorrendo a mecanismos externos e independentes deste.

5.2 Testes ao PESTT

Tal como já foi abordado ao longo deste documento, mais precisamente na secção 2.2.1., existem várias opções de testes, a vários níveis, que são possíveis de realizar sobre o PESTT. As opções incluem: testes unitários, testes de módulos, testes de integração, testes de sistema e testes de aceitação. Das opções disponíveis, três delas foram descartadas à partida. Estas três opções foram: testes unitários, testes de módulos e testes de integração.

Optámos por descartar os testes unitários e de módulos, uma vez que a grande contribuição deste trabalho foi feita na camada de apresentação da ferramenta. Posto isto, a esmagadora maioria dos resultados obtidos são visuais, sendo mais fácil detetar *bugs* de um ponto de vista funcional do sistema como um todo.

Testes de integração são realizados com o objetivo de validar o encaixe entre vários módulos que funcionam de forma independente entre si. Sendo que o PESTT possui apenas um módulo, esta opção foi também descartada. Neste caso, considera-se um módulo uma unidade de software independente (composta também ela por várias classes e pacotes), que pode posteriormente ser integrada com outras unidades independentes, e não apenas uma classe *java*.

Finalmente, e uma vez que o PESTT foi concebido como uma ferramenta orientada para o utilizador final, a abordagem escolhida para efetuar testes ao PESTT recaiu sobre **testes funcionais de sistema** e **testes de aceitação**. Para efetuar estes testes, foram escolhidos quatro voluntários em ambiente profissional (empresarial) com diferentes perfis. Apesar de quatro pessoas não constituírem uma amostra estatisticamente significativa, o objetivo subjacente à sua escolha foi obter *feedback* profissional dos perfis com maior interesse para a realização de testes: analista funcional, programador, *tester* e utilizador não especialista.

A cada um dos voluntários foram explicadas as funcionalidades e comportamentos esperados do PESTT. Cada um deles efetuou um conjunto de tarefas e sequências de ações, que foram posteriormente confrontadas com os resultados esperados.

Os cenários de teste contemplados incluíram:

1) Analista funcional: Foi pedido ao utilizador que desempenhasse várias sequências de ações, verificando que todas elas levariam ao esperado. Estas sequências de ações tinham como objetivo:

- obter o Grafo de Fluxo de Controlo correspondente a um pedaço de código;
- obter os caminhos de testes consoante vários critérios escolhidos interativamente na *view* de escolha de critérios;
- obter a seleção do código correspondente a um nó do Grafo de Fluxo de Controlo, através da seleção do mesmo;
- introduzir um requisito de teste manualmente;
- obter estatísticas de cobertura.

2) *Tester*: O utilizador com perfil de *tester* fez uma análise orientada à execução de determinadas sequências de instruções que facilmente poderiam levar à obtenção de resultados diferentes dos esperados. Algumas destas execuções foram, por exemplo:

- introdução manual de caminhos inválidos no Grafo de Fluxo de Controlo;
- tentativa de desenho do Grafo de Fluxo de Controlo a partir de um pedaço de código sintaticamente incorreto;
- tentativa de obtenção de caminhos de testes segundo o critério *Round Trip* num Grafo de Fluxo de Controlo que não contém ciclos;
- tentativa de desenho do Grafo de Fluxo de Controlo a partir de uma secção exterior a um método.

3) Programador: O utilizador com perfil de programador fez a sua análise com base na escrita de código com crescente complexidade, com posterior validação dos nós, arestas, e caminhos de teste gerados segundo a escolha do critério de teste.

4) Utilizador não especialista: Neste caso, foi dada liberdade ao utilizador para, após explicação do objetivo e funcionamento da aplicação, testar o PESTT do ponto de vista da sua usabilidade, clareza das respostas e tempo decorrente até à obtenção de um resultado.

A lista abaixo exemplifica o tipo e formato de sequências de ações que foram entregues a cada perfil de utilizador. Neste caso concreto, seguem-se o conjunto de passos pedidos que o utilizador com perfil de analista funcional seguisse:

1. Na *view* do *java editor* colar o seguinte pedaço de código:

```
public class Testing {
     public int somaParA(int a, int b){
         int soma = 0;
        if (a%2 == 0 )
soma = a + b;

                    return soma;
     }
}
```

2. É esperado que na *view* do *DrawGraphView* apareça de imediato o grafo de fluxo de controlo correspondente ao pedaço de código introduzido em 1.

3. Na *view* do *CriteriaGraph* clicar num dos vários critérios de cobertura de grafos de fluxo de controlo.

4. É esperado que na *view CriteriaPaths* apareçam de imediato os caminhos gerados a partir do critério de cobertura escolhido em 3.

5. Novamente na *view* do *DrawGraphView* clicar num nó do grafo de fluxo de controlo.

6. É esperado que na *view* do *java editor* seja sublinhada a linha correspondente ao nó selecionado.

7. No menu, clicar no botão *add test manually* e introduzir o caminho [0,1,2].

8. É esperado que na *view Statistics* seja dada a informação sobre os níveis de cobertura do caminho introduzido.

5.3 Resultados obtidos

Em primeiro lugar, foi nítido que todas as sequências de ações levaram ao resultado esperado. Outro aspeto importante e realçado foi a rapidez de desenho do Grafo de Fluxo de Controlo quando se alterna entre métodos, bem como a alteração em tempo real do GFC à medida que um método é modificado. Também a facilidade de representação de grafos com alguma complexidade foi referida pelos voluntários com perfil programador e *tester* como sendo um aspeto muito positivo. Todas as tentativas de obtenção de resultados errados falharam. No caso da tentativa de desenho do GFC a partir de um método sintaticamente incorreto, ou de uma secção exterior a um método, optámos por desenhar apenas o nó inicial, como mostra a seguinte figura:

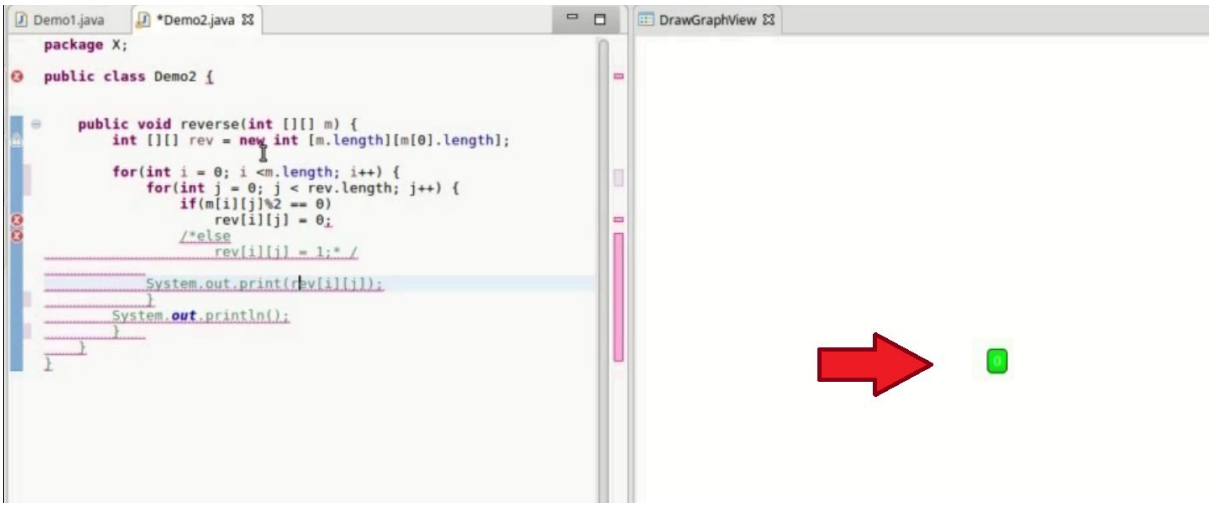

Figura 5.1.: Resposta do PESTT perante uma situação de erro.

Perante a tentativa de obter caminhos de teste segundo o critério *Round Trip* num GFC sem ciclos, optámos por deixar em branco o espaço onde os caminhos de teste normalmente apareceriam:

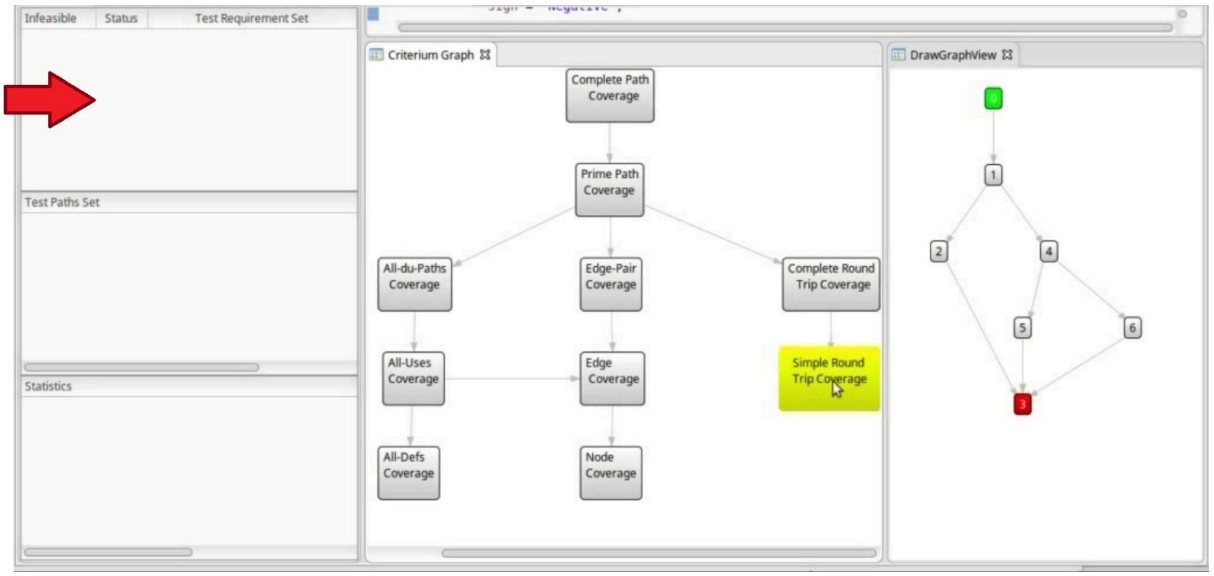

Figura 5.2.: Resposta do PESTT perante uma situação de erro.

Também as mensagens de erro apresentadas foram as esperadas e no momento esperado. Por exemplo, aquando da introdução de caminhos inválidos no grafo, neste caso o caminho [0,1,5,0], o comportamento do PESTT foi o seguinte:

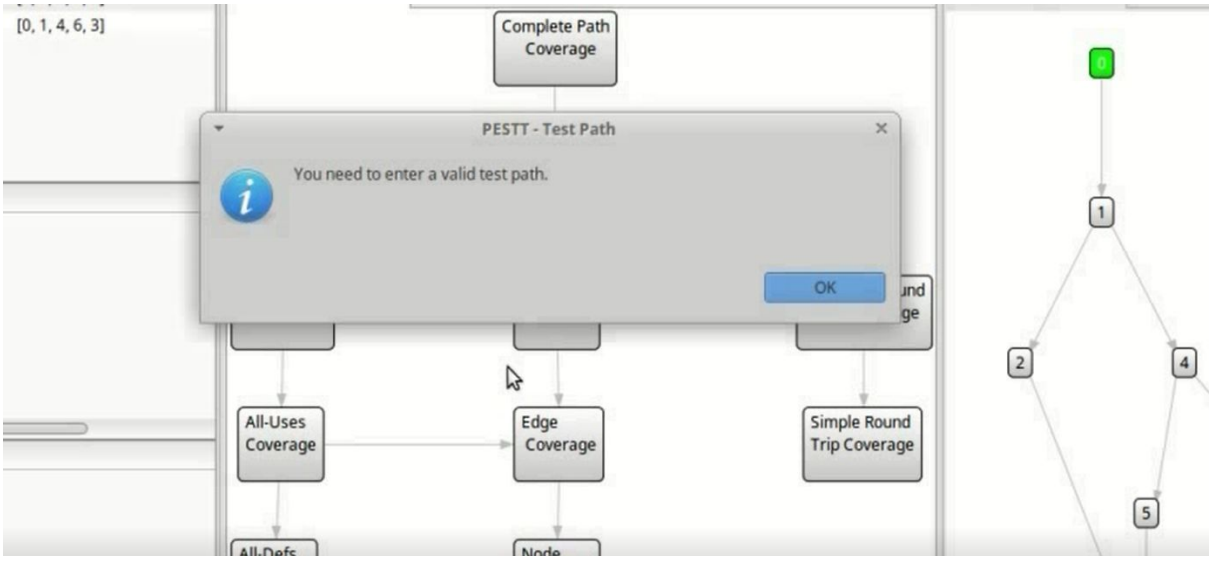

Figura 5.3.: Resposta do PESTT perante uma situação de erro.

A conclusão de todos os voluntários, tendo em conta as várias perspetivas, foi considerar o PESTT um *plug-in* intuitivo, claro e bastante útil do ponto de vista da validação dos resultados obtidos manualmente num contexto profissional, quando utilizadas técnicas sobre Grafo de Fluxo de Controlo.

Como pontos a melhorar, sugeriram as seguintes funcionalidades que fazem sentido implementar:

- realce de um caminho do Grafo de Fluxo de Controlo aquando da seleção de um pedaço de código;
- existência de um botão que permitisse desligar a atualização do Grafo de Fluxo de Controlo em tempo real, para que se possa alterar o código sem que o Grafo de Fluxo de Controlo esteja constantemente a sofrer alterações.

Apesar destas duas funcionalidades não estarem disponíveis nesta versão do PESTT, estas já faziam parte da primeira versão da ferramenta, sendo a sua implementação um objetivo futuro.

Capítulo 6
Conclusões e Trabalho Futuro

6.1 Análise Crítica

O principal objetivo desta tese foi reformular o *PESTT – Educational Software Testing Tool* para que fique em conformidade com o novo modelo do Eclipse IDE 4.x. A grande contribuição para este trabalho incidiu na reestruturação da interface gráfica do PESTT, ao nível da implementação do seu MVC: utilizar injeção de dependências em detrimento do padrão *Observer*, como padrão para a comunicação entre componentes, alterar a forma como interagir com o *Eclipse* em termos de modelo e pontos de extensão.

O PESTT já tinha sido usado num ambiente real de ensino, onde teve uma excelente aceitação, tanto por parte de professores, como de alunos. Os pontos fortes deste *plug-in* eram realçados como tendo um grande impacto na aprendizagem de conceitos (melhorando significativamente os resultados finais obtidos) e na utilidade na verificação de resultados, servindo para comparar os resultados obtidos pelos alunos com os resultados obtidos pelo PESTT. Esta foi a principal motivação deste trabalho: voltar a disponibilizar um recurso que acrescenta valor aos olhos de professores e alunos – o que foi cumprido.

Uma vez que o PESTT, na sua primeira versão, já tinha sido utilizado num ambiente real de ensino onde foi possível obter o *feedback* de professores e alunos, nesta nova versão optámos pela disponibilização num contexto profissional empresarial. Deste modo, foi-nos possível obter *feedback* de alguns dos principais *stakeholders* das equipas do mundo real do trabalho. À semelhança da primeira versão, houve uma grande aceitação da ferramenta, onde voltaram a ser destacados os seus pontos mais fortes:

- libertação do programador quanto a tarefas monótonas, demoradas e repetitivas;
- desresponsabilização do programador perante tarefas altamente sujeitas à introdução de erros;
- rapidez na obtenção de resultados;
- possibilidade de uso da ferramenta para verificar a correção da tradução de

código em grafos de fluxo de controlo, úteis não só nas tarefas de testes;

- possibilidade de uso da ferramenta para verificar a correção da obtenção de caminhos no grafo de fluxo de controlo, úteis não só nas tarefas de testes;
- clareza nas respostas apresentadas;
- interface clara e de simples utilização.

A reestruturação a que o PESTT foi sujeito, exigiu um grande esforço de investigação e trabalho de engenharia. Estas necessidades revelaram-se mais trabalhosas e demoradas do que o inicialmente previsto, não estando em conformidade com o tempo estipulado para uma tese de mestrado. Deste modo nem todas as funcionalidades previstas para esta versão do PESTT foram implementadas. Essas funcionalidades incluem:

- adição de informação extra ao grafo de fluxo de controlo;
- seleção de um pedaço de código e realce do correspondente nó do grafo de fluxo de controlo;
- visualização de caminhos de teste no código e no grafo de fluxo de controlo.

Na imagem que se segue, voltamos a apresentar um resumo da relação entre funcionalidades implementadas e funcionalidades que ficaram por implementar.

Funcionalidade	Previsto	Implementado
Desenho do GFC	✓	✓
Adicionar informação extra ao GFC	✓	✗
Apresentação da vista de seleção de critérios	✓	✓
Apresentação da vista de conjunto de requisitos de teste	✓	✓
Apresentação da vista de requisitos de teste manuais	✓	✓
Apresentação da vista de estatísticas	✓	✓
Seleção de um nó do GFC e realce do correspondente pedaço de código	✓	✓
Seleção de um pedaço de código e realce do correspondente nó do GFC	✓	✗
Visualização de caminhos de teste no código e no GFC	✓	✗

Figura 6.1.: Esquema de funcionalidades implementadas na nova versão do PESTT.

6.2 Trabalho Futuro

Os próximos passos para a reestruturação do PESTT passam claramente pela implementação das funcionalidades acima referidas como estando em falta. Mais além, seria interessante fazer três coisas com o PESTT:

1. disponibilizar a ferramenta *online*, por exemplo, em conjunto com um compilador de *java* acessível a partir de um *browser*;
2. já *online*, disponibilizar o PESTT de forma independente de um IDE em particular;
3. melhorar a análise de caminhos executada pelos testes, para ter resultados mais precisos em relação aos níveis de cobertura.

Bibliografia

[1] Paul Ammann & Jeff Offutt. *"Introduction to Software Testing"*. Cambridge University Press, 2008.

[2] Daniel Galin. *"Software Quality Assurance"*. Addison-Wesley Professional, 2004.

[3] ISO (1997) ISO 9000-3:1997(E), *Quality Management and Quality Assurance Standards – Part 3: Guidelines for the Application of ISO 9001:1994 to the Development, Supply, Installation and Maintenance of Computer Software*, 2nd edn. International Organization for Standardization 1997.

[4] ISO/IEC (2001) ISO 9000-3:2001 Software and System Engineering – Guidelines for the Application of ISO 9001:2000 to Software, Final draft, International Organization for Standardization (ISO), unpublished draft, 2001.

[5] Gerald D. Everett & Raymond McLeod Jr. "*Software Testing. Testing Across the Entire Software Development Life Cycle",* John Wiley & Sons, Inc, 2007.

[6] History's Worst Software Bugs. Therac-25 medical accelerator. https://www.wired.com/2005/11/historys-worst-software-bugs/

[7] Mark Fewster & Dorothu Graham. *"Software Test Automation. Effective use of test execution tools",* Addison-Wesley Professional, 1994.

[8] Hong Zhu & Patrick A. V. Hall & John H. R. May. *"Software Unit Test Coverage and Adequacy", ACM Computing Surveys,* 29(4):366–427, 1997.

[9] Rui M. S. Gameiro. *"PESTT: PESTT Educational Software Testing Tool"*. MSc thesis, Universidade de Lisboa 2012.

[10] Gordon Fraser & Andrea Arcuri. "*EvoSuite: Automatic Test Suite Generation for Object- Oriented Software",* Procs. 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp 416-419, 2011

[11] Anthony J. H. Simons. *"JWalk: a Tool for Lazy, Systematic Testing of Java Classes by Design Introspection and User Interaction",* Journal of Automated Software Engineering, vol 14, number 4, pp. 369--418, 2007

[12] Carlos Pacheco & Michael D. Ernst. "*Randoop: Feedback-Directed Random Testing for Java",* Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pp 815-816, 2007

[13] EclEmma 3.0.1 Java Code Coverage for Eclipse. http://www.eclemma.org/

[14] Eclispe IDE. http://www.eclipse.org

[15] IntelliJ IDEA. https://www.jetbrains.com/idea/

[16] Maven. https://maven.apache.org

[17] Eclipse IDE na Wikipédia. https://pt.wikipedia.org/wiki/Eclipse_(software)

[18] Mark Seeman, Dependency Injection in .NET, Manning Publications, 2011.

[19] Injeção de Dependências na Wikipédia. https://en.wikipedia.org/wiki/Dependency_injection

[20] Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions. http://www.vogella.com/tutorials/EclipsePlugin/article.html

[21] What is RCP and why should I care? https://www.modumind.com/what-is-rcp/

[22] JUnit. https://junit.org/junit5/

[23] TestNG. https://testng.org/doc/index.html

[24] Eclipse 4 - RCP - Dependency Injection.
https://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection

[25] Java Community Process Program - Java Specification Requests (JSR) - JSR#330.
https://jcp.org/en/jsr/detail?id=330

[26] Bert Bates, Kathy Sierra, Eric Freeman, Elisabeth Robson. "*Head First Design Patterns – A Brain-Friendly Guide*", O'Reilly Media, June 2009.

[27] Report: Software failures cost $1.1 trillion in 2016.
http://servicevirtualization.com/report-software-failures-cost-1-1-trillion-2016/

[28] Paul C. Jorgensen. "*Software Testing: A Craftsman's Approach*" Third Edition. Auerbach Publications, 2013.

[29] MVC Architecture. https://developer.chrome.com/apps/app_frameworks

[30] What's New in Eclipse 4.0
http://download.eclipse.org/e4/sdk/drops/R-4.0-201007271520/eclipse-news.html

[31] The Architecture of Open Source Applications: Eclipse.
https://www.aosabook.org/en/eclipse.html

[32] Eclipse Modeling Framework (EMF). https://www.eclipse.org/modeling/emf/

[33] Rich Client Platform (RCP).
https://searchmicroservices.techtarget.com/definition/Rich-Client-Platform-RCP

[34] e4 Project. https://www.eclipse.org/e4/

[35] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. *"Design Patterns - Elements of Reusable Object-Oriented Software",* Addison-Wesley, 1994.

[36] Dynamic menu contributions in Eclipse 4.
http://blog.vogella.com/2013/03/21/dynamic-menu-contributions-in-eclipse-e4/

[37] Menu Contributions. https://wiki.eclipse.org/Menu_Contributions

[38] Perspectives.
https://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-4.htm

[39] The Eclipse application model.
https://www.vogella.com/tutorials/EclipseRCP/article.html#the-eclipse-application-model

[40] Building a Rich Client Platform application.
https://help.eclipse.org/photon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Frcp.htm

[41] Eclipse 4 application model.
https://eclipsesource.com/blogs/2012/05/10/eclipse-4-final-sprint-part-1-the-e4-application-model

[42] Creating new dynamic web project eclipse.
http://www.thejavageek.com/2013/10/22/creating-new-dynamic-web-project-eclipse/

[43] e4 – Fundamental Overview on the Eclipse 4 Application Platform.
https://tomsondev.bestsolution.at/2011/03/16/e4-fundamental-overview-on-the-eclipse-4-application-platform/

[44] Eclipse 4 model fragments.
https://eclipsesource.com/blogs/2012/06/26/eclipse-4-e4-tutorial-part-3-extending-the-application-model/

[45] Eclipse Plug-in Developer Guide.
https://www.linuxtopia.org/online_books/eclipse_documentation/eclipse_platform_plug-in_developer_guide/topic/org.eclipse.platform.doc.isv/guide/eclipse_platform_plugin_workbench_basicext_popupMenus.htm

[46] Lars Vogel & Mike Milinkovich. *"Eclipse Rich Client Platform - Third Edition: Revised edition based on Eclipse 4.4"*, 2015.

[47] Alex Blewitt & Mike Milinkovich. *"Eclipse Plug-in Development - Begginer's Guide"*, Second Edition, 2016.

[48] From the Application Model to the Implementation of Views. https://eclipsesource.com/blogs/2012/06/12/eclipse-4-e4-tutorial-part-2/

[49] Composite Design Pattern. https://www.geeksforgeeks.org/composite-design-pattern/

[50] Eclipse4/RCP/Contexts. https://wiki.eclipse.org/Eclipse4/RCP/Contexts

[51] Regression Testing. https://www.tutorialspoint.com/software_testing_dictionary/regression_testing

[52] Back to basics: Display, Shell, Window. https://dzone.com/articles/back-basics-display-shell

[53] Common Descriptor Concepts. https://www.eclipse.org/eclipselink/documentation/2.4/concepts/descriptors001.htm

[54] Paradoxo do mentiroso. https://pt.wikipedia.org/wiki/Paradoxo_do_mentiroso

[55] Eclipse 4 RCP - Accessing and extending the Eclipse context. https://www.vogella.com/tutorials/Eclipse4ContextUsage/article.html