FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automação de Testes Baseados em Cenários com UML e Programação Orientada a Aspetos

Mário Jorge Ventura de Castro

Mestrado Integrado em Engenharia Informática e Computação

Orientador: João Carlos Pascoal de Faria

Janeiro de 2013

Automação de Testes Baseados em Cenários com UML e Programação Orientada a Aspetos

Mário Jorge Ventura de Castro

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Nuno Honório Rodrigues Flores

Arguente: Fernando Brito e Abreu

Vogal: João Carlos Pascoal de Faria

Janeiro de 2013

Resumo

UML (*Unified Modeling Language*) é uma linguagem de modelação de artefactos de sistemas de *software* que tem vindo cada vez mais a ser utilizada tanto a nível académico como industrial. No entanto, tornou-se prática frequente a utilização dos modelos UML apenas para documentação. Esta utilização passiva acarreta vários inconvenientes, desde logo, o consumo de tempo implicado neste processo não ter associado um benefício proporcional. Os modelos UML podem também não expressar com exatidão o respetivo sistema de *software* e estando este em contínua evolução, facilmente a documentação fica em desacordo com o código.

Para combater os problemas que advêm da utilização de UML apenas para documentação, têm sido desenvolvidas nos últimos anos diversas técnicas de geração de código e testes a partir dos modelos. Inspirando-se nestas técnicas, foi desenvolvido um protótipo interno experimental (FEUP-SBT) para a geração automática de testes unitários JUnit a partir de diagramas de sequência UML, na forma de uma extensão (*add-in*) para a ferramenta de modelação Enterprise Architect.

Este trabalho de dissertação, que se serviu do protótipo FEUP-SBT como base, teve como principal objetivo evoluí-lo para que, além da geração de código de teste, permitisse também ordenar a execução dos cenários de teste (especificados em diagramas de sequência UML) e receber o *feedback* visual dos resultados da sua execução diretamente no ambiente de modelação Enterprise Architect, tornando o processo de geração de código de testes transparente para o utilizador.

Foi ainda decidido implementar adicionalmente uma outra funcionalidade que permitisse analisar a cobertura dos testes executados, ou seja, verificar se todas as interações especificadas no modelo foram efetivamente exercitadas, apresentando uma vez mais os resultados através de um esquema de cores diretamente no modelo UML. Estipulou-se ainda a construção de um módulo de configuração para a ferramenta.

Beneficiando da implementação das novas funcionalidades, pretendia-se ainda detetar possíveis falhas ou limitações do protótipo, realizando um processo de refabricação (*refactoring*) ao mesmo, aumentando a sua robustez e manutenibilidade.

No desenvolvimento dos objetivos propostos, foi utilizada uma abordagem iterativa, onde semanalmente era validada a implementação dos requisitos anteriores e definidos os seguintes. Esta metodologia revelou-se bastante útil para verificar a correta evolução do desenvolvimento.

Neste documento é apresentado inicialmente uma análise do protótipo FEUP-SBT-1.0 e do estado da arte sobre o tema "automação de testes baseados em diagramas de sequência UML". Posteriormente é descrita a conceção e implementação da solução e o manual de utilização da versão 2.0 do FEUP-SBT. Por fim é apresentado o processo de validação aplicado nas funcionalidades implementadas.

Abstract

UML (Unified Modeling Language) is a language for modeling artifacts of software systems whose usage has been increasing both in academic and industrial projects. However, it has become a common practice to use UML models for documentation only. This passive usage entails several drawbacks. First, the time consumption involved in this process has not associated a proportional benefit. The UML models also may not accurately express the respective software system. Being the software in a continuous evolution, the documentation easily gets in disagreement with the code.

To overcome the problems arising from the usage of UML for documentation only, several techniques have been developed in recent years to generate code and tests from the models. Inspired by these techniques, it was developed an experimental prototype (FEUP-SBT) for the automatic generation of JUnit unit tests from UML sequence diagrams, as an add-in to Enterprise Architect modeling tool.

This dissertation work was based on the prototype FEUP-SBT and aimed to evolve it so that, beyond the test generation, it would also be possible to order the execution of test scenarios (specified in UML sequence diagrams) and receive a visual feedback of the execution results directly in the Enterprise Architect modeling environment, thus hiding generated test code from users.

It was also decided to further implement another feature so that it could be possible to analyze the coverage of tests execution, in other words, verify that all interactions specified in the model were actually exercised, showing once again the result through a color scheme directly in the UML model. It was still further decided to build a configuration module for the tool.

Benefiting from the implementation of the new functionalities, it was also intended to further detect possible failures or limitations presents on the prototype, performing a refactoring process, in order to enlarge its robustness and maintainability.

To develop the proposed objectives, it was used an iterative approach, where in each week were validated prior requirements and defined the following ones. This methodology proved to be very useful to check the correct evolution of development.

In this report it's initially presented an analysis of the prototype FEUP-SBT-1.0 and the state of the art on the theme "automated testing based on UML sequence diagrams". Later is described the conception and implementation of the solution and the usage manual of FEUP-SBT-2.0. Finally is presented the validation process applied to the features implemented.

Agradecimentos

Este trabalho de dissertação culmina um longo percurso, o Mestrado Integrado em Engenharia Informática e Computação, no qual devo vários agradecimentos:

Em primeiro lugar, ao meu orientador Prof. João Carlos Pascoal de Faria, pela total disponibilidade e pelo incansável entusiasmo dispensados durante os últimos meses na orientação deste trabalho, o meu muito obrigado!

Agradeço à Iris, principal responsável pelo ponto de viragem vital nesta caminhada, pela confiança e serenidade que todos os dias me transmite.

Aos meus pais, cujo orgulho neste trajeto sempre se manteve inabalável, agradeço a estabilidade e carinho que sempre me ofereceram.

Agradeço à minha família, pilar indispensável, pelo apoio e orgulho inesgotáveis.

Aos meus colegas, pela honra que me deram em juntos ultrapassarmos os inúmeros desafios com que nos deparamos, agradeço a amizade.

Aos professores com quem tive o privilégio de interagir ao longo do meu percurso académico, agradeço os incontáveis ensinamentos e o entusiasmo sempre demonstrado.

Finalmente, mas não em último, agradeço à FEUP por muito mais que formar engenheiros, formar homens e mulheres com sentido de responsabilidade na sociedade.

Mário Jorge Ventura de Castro

"A vida foge-nos, escapa-se-nos como água entre os dedos.
Morremos a cada respiração, a cada palavra, a cada olhar, momento a momento encurta-se a distância que nos separa do nosso fim, nascemos e já estamos condenados à morte.
A vida é breve, não passa de um instante fugaz, de um brilho efémero nas trevas da eternidade."

José Rodrigues dos Santos

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

AOP	Programação Orientada a Aspetos (*Aspect-Oriented Programming*)
API	Interface de Programação de Aplicações (*Application Programming Interface*)
DLL	Bibliotecas de Vínculo Dinâmico (*Dynamic-Link Library*)
EA	Enterprise Architect
EAP	Projeto do Enterprise Architect (Enterprise Architect *Project*)
FEUP	Faculdade de Engenharia da Universidade do Porto
IDE	Ambiente Integrado para Desenvolvimento de *Software* (*Integrated Development Environment*)
JAR	Java ARchive
MBT	Teste Baseado em Modelos (*Model Based Testing*)
MDA	Arquitetura Dirigida por Modelos (*Model-Driven Architecture*)
MDD	Desenvolvimento Dirigido a Modelos (*Model Driven Development*)
MIEIC	Mestrado Integrado em Engenharia Informática e Computação
OMG	*Object Management Group*
PIM	Modelo Independente de Plataforma (*Platform-Independent Model*)
PDIS	Preparação para a Dissertação
PSM	Modelo para Plataforma Específica (*Platform-Specific Model*)
SBT	Teste Baseado em Cenários (*Scenario Based Testing*)
SDK	Pacote de Desenvolvimento de *Software* (*Software Development Kit*)
SMC	Sequência de Chamadas a Métodos (*Sequence of Methods Calls*)
TDD	Desenvolvimento Dirigido por Testes (*Test Driven Development*)
UML	*Unified Modeling Language*
XML	*eXtensible Markup Language*

Capítulo 1

Introdução

No presente capítulo é introduzido o trabalho de dissertação, registando a motivação para o seu desenvolvimento e listando os objetivos que se pretendiam cumprir. Por fim, é apresentada a estrutura deste documento.

1.1 Motivação

UML (*Unified Modeling Language*) é uma linguagem de modelação para visualização, especificação, construção e documentação de artefactos de sistemas de *software* [Alh99]. Idealizada a partir da necessidade de unificação de diferentes métodos de análise e desenho orientado por objetos que foram sendo propostos nas décadas de 80 e 90, tornou-se padrão do consórcio OMG (*Object Management Group*) em 1997 [FS99]. Desde então, tem-se tornado numa linguagem de modelação cada vez mais utilizada a nível académico e industrial, sendo constantemente evoluída. No entanto, e atendendo às suas características, crê-se que a utilização de UML possa ser ainda mais potenciada em relação ao que acontece atualmente.

A utilização dos modelos UML apenas para documentação (de sistemas de *software* a construir ou já existentes) tornou-se prática frequente no desenvolvimento de *software*, tanto a nível educacional como no mundo profissional. Esta utilização passiva dos modelos UML acarreta vários inconvenientes. Desde logo, o consumo de tempo implicado neste processo não tem associado um benefício proporcional. Tratando-se de um processo que geralmente não pode ser totalmente automatizado, exigindo consequentemente intervenção humana, o resultado pode não ser totalmente correto. Por fim, tratando-se o desenvolvimento de *software* dum processo iterativo e estando o *software* em constante evolução, facilmente a documentação fica em desacordo com o código ou é descuidada a sua manutenção.

Para combater os problemas que advêm da utilização de UML apenas para documentação, e tornar a utilização de UML mais eficaz e eficiente, têm sido desenvolvidas nos últimos anos diversas técnicas de geração de código a partir dos modelos (MDD – *Model Driven Development*) e de geração de testes a partir dos modelos (MBT – *Model Based Testing*). Com estas técnicas, para além de se recuperar o tempo despendido no desenho dos modelos UML, é possível verificar

e melhorar a qualidade destes modelos e aumentar a probabilidade destes se manterem atualizados. Esta técnica põe em prática os conceitos de Desenvolvimento Dirigido a Modelos (MDD – *Model Driven Development*) e Teste Baseado em Modelos (MBT – *Model Based Testing*). Desenvolvimento Dirigido a Modelos (MDD) pressupõe que se for possível construir um modelo de um sistema, depois este poderá ser transformado em algo real, mais concretamente, em código [MCF03]. Já Teste Baseado em Modelos (MBT) pressupõe a geração de testes a partir de modelos, ou mais especificamente, a automação do desenho de testes de caixa preta [UL07], isto é, a avaliação do comportamento externo do *software*.

Com base na ideia anterior de geração automática de código e testes a partir dos modelos UML, foi desenvolvido um protótipo experimental, denominado FEUP-SBT-1.0, que possibilita a geração automática de testes unitários JUnit [JUn12] a partir de diagramas de sequência UML, na forma de uma extensão (*add-in*) para a aplicação Enterprise Architect (EA) [Sys12], mas que se encontra ainda num estado que não permite a sua utilização prática fora de um contexto de investigação. Este protótipo servirá de base para esta dissertação, sendo analisadas as suas limitações e necessidades em maior detalhe.

A motivação pessoal principal para a realização desta dissertação é incentivar a utilização dos modelos UML desde o início do desenvolvimento de *software*, dotando os engenheiros de *software* de uma ferramenta que lhes permita lucrar com o tempo despendido na elaboração dos modelos.

1.2 Objetivos

O objetivo principal deste trabalho de dissertação passava por evoluir a versão do protótipo experimental FEUP-SBT-1.0, na qual é possível gerar o código de teste JUnit a partir do modelo UML, para uma versão 2.0 que permite ordenar a execução dos cenários de teste (especificados em diagramas de sequência UML) e receber o *feedback* visual dos resultados da sua execução diretamente no ambiente de modelação Enterprise Architect, tornando o processo de geração de código de testes transparente para o utilizador.

Estipulou-se adicionalmente o objetivo de implementar uma outra funcionalidade que permitisse adicionalmente analisar a cobertura da execução dos cenários de testes, ou seja, verificar se todas as interações especificadas no modelo foram efetivamente exercitadas, apresentando uma vez mais os resultados através de um esquema de cores diretamente no modelo UML. Além desta nova funcionalidade, pretendia-se ainda construir um módulo de configurações, útil para a definição de opções várias do gerador e executor de testes.

Beneficiando da implementação desta nova funcionalidade, era pretendido efetuar uma profunda análise aos componentes que compõem o protótipo, detetando e corrigindo possíveis falhas, inconsistências, limitações ou necessidades e ainda estruturando de forma conveniente e robusta o código fonte dos referidos componentes.

Paralelamente a esta análise cuidada do protótipo experimental, pretendia-se realizar uma revisão bibliográfica sobre o tema "automação de testes baseados em diagramas de sequência UML",

da qual se desejava reunir um conjunto de abordagens que posteriormente pudessem ser comparadas com o protótipo experimental e assim verificar as suas mais valias e possíveis ideias de melhoramentos a aplicar no protótipo.

1.3 Estrutura do documento

Para além deste capítulo introdutório, este documento contará com mais 6 capítulos. No capítulo 2 é apresentada uma análise do protótipo existente, explicando a abordagem em que se insere, os seus componentes e as suas limitações. No capítulo 3 encontra-se a revisão bibliográfica incidida sobre o tema "automação de testes baseados em diagramas de sequência UML", com a análise comparativa entre as abordagens identificadas no processo e o protótipo existente. O capítulo 4 apresenta a conceção e implementação das novas funcionalidades do protótipo, enquanto o capítulo 5 contém todas as informações sobre a utilização da ferramenta. O leitor pode considerar mais conveniente ler primeiro o capítulo 5, onde é apresentada uma visão externa do protótipo desenvolvido, antes do capítulo 4, que apresenta uma visão interna do mesmo. No capítulo 6 são apresentadas as ações de validação da implementação através de um sistema real e dos casos de teste existentes. Por último, o capítulo 7 contém as conclusões retiradas da realização deste trabalho.

Capítulo 2

Análise do protótipo existente

Neste capítulo é apresentado o resultado da análise ao protótipo experimental FEUP-SBT-1.0, para geração automática de testes unitários a partir de diagramas de sequência UML. É descrita a abordagem para a qual o protótipo foi pensado e demonstrado o seu funcionamento. São também expostas as tecnologias utilizadas pelo protótipo. Por fim, são analisadas as limitações e necessidades principais da sua versão atual.

2.1 Abordagem

Podendo ser utilizado noutras situações, o protótipo FEUP-SBT-1.0 foi desenvolvido com o objetivo de potenciar uma utilização ativa dos modelos UML no desenvolvimento de *software*, combinando Desenvolvimento Dirigido a Modelos (MDD), Teste Baseado em Modelos (MBT) e Desenvolvimento Dirigido por Testes (TDD). Esta abordagem está descrita através do diagrama da figura 2.1 e será de seguida explicada passo a passo. De salientar que este é um processo iterativo, portanto, a sequência de passos deve ser repetida as vezes que se considerar necessário até os modelos UML e respetivo código de teste atingirem o estado pretendido pelo engenheiro de *software*.

Passo 1. Modelação

O protótipo FEUP-SBT-1.0 é composto por um *add-in* para a ferramenta de desenho de modelos UML Enterprise Architect. Esta ferramenta, entre outras funcionalidades, permite que os seus utilizadores modelem diagramas de classes UML para definição da estrutura do sistema e diagramas de sequência UML para definição das interações entre os objetos que fazem parte do sistema. A modelação destes 2 tipos diagramas é o primeiro passo da abordagem.

Passo 2. Verificação

A consistência e a completude do modelo comportamental desenvolvido no ponto 1 podem ser verificadas através de outra funcionalidade do *add-in* desenvolvido para o Enterprise Archi-

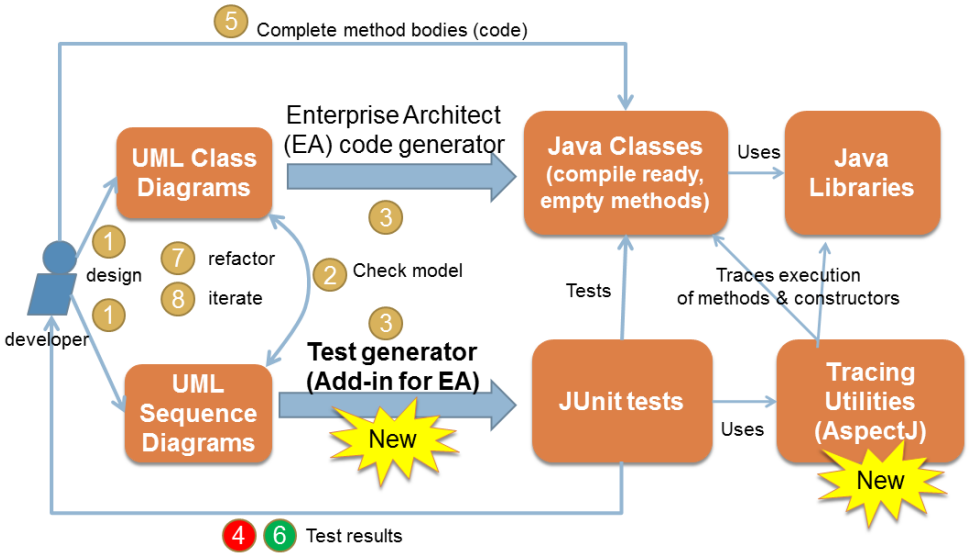


Figura 2.1: Abordagem em que se insere o protótipo FEUP-SBT-1.0.

tect, denominada *UML Checker*, e que permite ao utilizador verificar se os métodos utilizados no diagrama de sequência têm correta correspondência no diagrama de classes e ainda se o diagrama de sequência abrange todos os métodos definidos no diagrama de classes.

Passo 3. Geração de código

A ferramenta Enterprise Architect já disponibiliza de origem uma funcionalidade, denominada *"Enterprise Architect code generator"*, que permite geração de código, ou mais especificamente para esta abordagem, classes Java a partir dos diagramas de classes definidos. As classes criadas estão compiláveis e os seus métodos são criados sem conteúdo, prontos a ser implementados pelo engenheiro de *software*. De seguida, o protótipo FEUP-SBT-1.0, através dos menus do Enterprise Architect, gera os testes unitários JUnit a partir dos diagramas de sequência UML previamente modelados. É neste ponto que o protótipo intervém de forma direta. A geração de código e teste compõe o passo 2 na abordagem proposta.

O código de teste gerado não verifica apenas as interações externas, verificando também se todas as interações internas ocorrem como especificado. Para tal, foi implementada e disponibilizada a biblioteca Java para análise do traçado de execução, denominada *TracingUtilities*, que mais à frente será apresentada em detalhe.

Passo 4. Execução dos testes (com métodos por implementar)

De acordo com as normas estipuladas pela metodologia Desenvolvimento Dirigido por Testes (TDD), "os testes devem ser executados e confirmado que os mesmos falham [Bec02]" desde que estejam prontos a executar. É precisamente nestas circunstâncias em que o processo se encontra neste momento, representado pelo ponto 3 na abordagem.

Passo 5. Implementação dos métodos

A etapa seguinte, ponto 4 da abordagem proposta, passa pelo engenheiro de *software* implementar os métodos automaticamente gerados no ponto 1, para que os testes que falharam na etapa anterior possam agora ser superados com sucesso. O Desenvolvimento Dirigido por Testes (TDD) incute que os métodos devem ser implementados com o mínimo custo ou esforço para que possam passar os testes propostos. Posteriormente, estes sofrerão um processo de refabricação para otimizar o código desenvolvido.

Passo 6. Execução dos testes (com métodos já implementados)

Esta etapa, que é representada pelo ponto 5, assemelha-se à da alínea D, diferenciando-se por nesta altura os métodos das classes já estarem implementados e assim, o resultado esperado dos testes é que estes passem. Se por alguma razão algum teste falhar, terão de ser processadas as devidas modificações nos métodos implicados e novamente executados os testes.

2.2 Exemplo

Esta secção apresenta um exemplo ilustrativo dos passos que compõe a abordagem anteriormente exposta, usando como exemplo um mecanismo de folhas de cálculo (*Spreadsheet Engine*). São expostos os artefactos implementados e gerados pela execução da abordagem.

A. Modelo Estrutural

O primeiro passo da abordagem, conforme foi descrito anteriormente, é elaborar os diagramas estruturais e comportamentais para o sistema de *software* em causa. Começando pelo diagrama estrutural, a figura 2.2 apresenta o diagrama de classes UML do mecanismo de folhas de cálculo.

O mecanismo de folhas de cálculo suporta a criação de folhas de cálculo (*Spreadsheet*), que são compostas por células (*Cell*), permitindo a adição das mesmas a uma folha de cálculo (*addCell*). Uma célula contém um valor (*value*) e um indicador se se encontra em processamento de cálculo (*calculating*). A uma célula podem ser realizadas *queries* para determinar o seu valor (*getValue*) e pode ser atribuído um valor diretamente (*setValue*) ou uma fórmula (*setFormula*). É disponibilizado ainda um analisador (*Parser*) que permite converter expressões textuais numa representação em árvore definida pela classe Fórmula (*Formula*). Uma fórmula só pode conter constantes (*Const*), operadores binários (*BinOp*) e referências para células (*CellRef*). A classe *SpreadSheetCLI* representa uma interface por linha de comandos disponibilizada pelo mecanismo de folhas de cálculo. A classe (*CircularReferenceException*) representa exceções que são lançadas quando é detetado a ocorrência de cálculo de uma célula como uma função de si mesma. É possível ainda verificar que a classe do analisador (*Parser*) está definida com o estereótipo *«stub»*, indicando que esta classe contém métodos ainda não implementados.

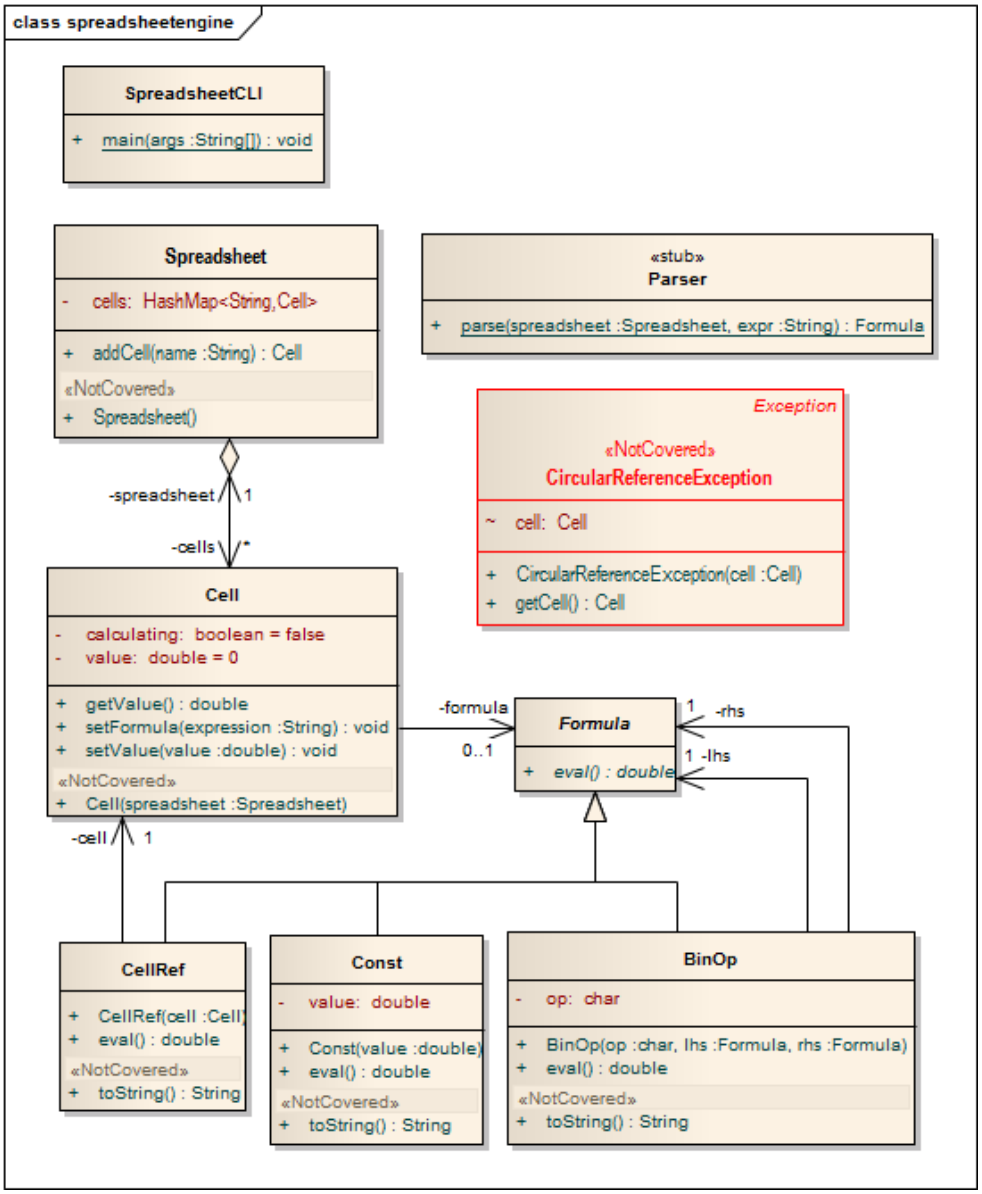


Figura 2.2: Diagrama de classes UML do mecanismo de folhas de cálculo.

B. Modelo Comportamental

Paralelamente à construção do modelo estrutural acima apresentado, é construído o modelo comportamental através de diagramas de sequência UML, representando o comportamento pretendido pelo mecanismo de folhas de cálculo, tanto para as interações externas com clientes/utilizadores como as interações internas entre os vários objetos. As figuras 2.3 e 2.4 apresentam os 2 diagramas de sequência implementados para testar o comportamento do mecanismo de folhas de cálculo.

Na figura 2.3 é apresentado o comportamento previsto da aplicação nas situações em que o utilizador define e efetua *queries* sobre células de uma dada folha de cálculo. Os cenários de teste

estão devidamente parametrizados numa nota estereotipada no fundo do diagrama, podendo assim ser executado o mesmo código de teste para diferentes situações, isto é, diferentes dados de teste.

Inicialmente começa-se por definir a folha de cálculo e adicionar duas células à mesma (*x* e *y*). De seguida, é testada a funcionalidade de atribuição de um valor e de uma fórmula a cada uma das células criadas. Conforme indicado, as fórmulas seguem a sintaxe *"y = x op cVal"*, por exemplo *"y = x + 2"*, tendo posteriormente de ser convertidas de texto para uma representação em árvore na memória. Este processo é orquestrado pelo analisador (*Parser*), que ainda não está implementado mas que já pode fazer parte do processo de teste, sendo este o principal objetivo da utilização de *stubs*.

Para tirar partido das funcionalidades do UML 2.x é utilizado um fragmento combinado, que permite, de forma rápida e concisa, definir diferentes traçados autorizados na execução do diagrama. São utilizados dois fragmentos combinados *"par"*, abreviatura de paralelo (*parallel*), atendendo a que o processamento do lado esquerdo (*lhs*) e do lado direito (*rhs*) das expressões pode ser realizado paralelamente (que na definição do UML não significa que ocorram ao mesmo tempo, antes que possam ocorrer por qualquer ordem [OMG11]), tanto na fase de atribuição de uma fórmula a uma célula, como na posterior verificação do seu valor.

Já em relação ao diagrama de sequência presente na figura 2.4, este representa como um utilizador pode interagir com mecanismo de folhas de cálculo através da interface por linha de comandos definida. É apresentando ainda como se espera que a classe encarregue de lidar com as interações do utilizador (*SpreadSheetCLI*) interaja por si só com os restantes elementos da aplicação.

No diagrama podemos verificar que em termos de interações externas, apenas são utilizadas as funções básicas `start`, `enter` e `display`. No entanto, para cada uma delas, é apresentado o comportamento interno esperado. Por exemplo, quando o utilizador lança a aplicação (`start`), internamente é esperado que seja criada uma folha de cálculo. Identicamente, quando o utilizador opta por introduzir (`enter`) a expressão *"x=1"* é esperado que internamente seja criada a célula *x* na folha de cálculo anteriormente criada, e que lhe seja atribuído o valor *1*.

C. Verificação do Modelo

Com os modelos estrutural e comportamental elaborados, pode ser agora utilizado o *plug-in UML Checker*, cuja execução alertará para possíveis inconsistências ou incompletudes nos modelos. Estes problemas são apresentados no próprio modelo, através da coloração e adição de etiquetas e notas no mesmo.

Os diagramas presentes nas figuras 2.2, 2.3 e 2.4 já contêm as modificações operadas pela execução do verificador *UML Checker*. No diagrama de classes presente na figura 2.2, a classe `CircularReferenceException` encontra-se com os contornos a vermelho e com a etiqueta *«NotCovered»* acima do seu nome. Esta informação indica que a classe não foi utilizada em qualquer cenário definido no modelo comportamental. Para além da utilização de cada classe, é ainda verificada a utilização de todos os métodos de todas as classes nos cenários definidos. É assim possível verificar que existem vários métodos que não se encontram em utilização, por

exemplo, os métodos `toString` das classes (`Const`, `BinOp` e `CellRef`). Esta funcionalidade é extremamente útil no caso do utilizador tencionar que o modelo comportamental atinja 100% de cobertura do modelo estrutural.

Nos diagramas das figuras 2.3 e 2.4 é possível verificar que existem inconsistências entre o modelo comportamental e estrutural. Estas inconsistências caracterizam-se por métodos referenciados nos diagramas de sequência UML que não têm paralelo no diagrama de classes UML, ou seja, não existe um método com o mesmo nome e o mesmo número de parâmetros. No diagrama de sequência da figura 2.3 é possível verificar algumas destas inconsistências, assinaladas através da coloração a vermelho das mensagens com problemas. Por exemplo, os construtores das classes folha de cálculo (`SpreadSheet`) e célula (`Cell`) têm um diferente número de parâmetros em relação ao definido no modelo estrutural. Outras inconsistências ocorrem tanto no diagrama da figura 2.3 como no da figura 2.4, dado que os métodos `getCellByName` e `eval` não estão definidos nas classes `Parser` e `SpreadSheetCLI`, respetivamente. Foram ocultadas dos diagramas as notas correspondentes a cada um dos erros encontrados, para não dificultar a visualização e perceção dos mesmos.

D. Geração de código e teste

Após corrigidas as inconsistências detetadas pelo verificador *UML Checker*, os modelos ficam prontos para serem transformados em código. Para a geração das classes a partir do diagrama de classes UML basta utilizar as funcionalidades nativas da ferramenta Enterprise Architect, enquanto que para a geração do código de teste, apenas é necessário executar o *plug-in TestGenerator*, integrado no protótipo FEUP-SBT, a partir dos menus do Enterprise Architect.

O código de teste gerado tem de ser integrado com a biblioteca *TracingUtilities* (traceutils.jar) e tem de ser ativada a execução de aspetos em AspectJ, antes do código de teste poder ser executado recorrendo ao JUnit3. A classe de testes gerada estende a classe `InteracTestCase` que se baseia na classe `TestCase` [API12] da framework JUnit3. Por cada diagrama de sequência é gerado um método de teste e nos casos em que são utilizados cenários parametrizados, como na figura 2.3, é adicionalmente gerado um método de teste por cada combinação de valores de parâmetros (caso de teste).

O excerto de código em 2.1 é uma amostra do código de teste criado pelo gerador de testes, com algumas linhas de código omitidas por limitações de espaço. Para o primeiro diagrama (figura 2.3) é criado o método de teste `testSpreadsheetAPI` que recebe como argumentos os parâmetros definidos no diagrama origem. Este método é depois chamado uma vez por cada caso de teste com os seus respetivos valores. As interações com o utilizador são testadas através da função `assertEquals` para verificar os valores de retorno. As interações externas são também analisadas, sendo apenas apresentada uma no código em 2.1, como exemplo, atendendo às limitações de espaço. Nestas situações é criada uma árvore de execução, onde são estruturadas as chamadas que estão previstas ocorrer, para no fim esta ser confrontada com a árvore de execução resultante da própria execução do código. Em relação ao segundo cenário, é utilizada uma classe auxiliar (`Console`) criada para lidar com a interface de utilizador, sendo utilizada novamente a

função `assertEquals` para confirmar valores de retorno.

```
// package and import declarations ommitted
public class SpreadsheetTest extends InteracTestCase {
public void testSpreadsheetAPI(double xVal, char op, double cVal,
  double yVal, String yExpr) {
Spreadsheet s = new Spreadsheet("s");
 Cell x = s.addCell("x");
 Cell y = s.addCell("y");
 x.setValue(xVal);

 ObjHandler<CellRef> r = new ObjHandler<CellRef>();
 ObjHandler<Const> c = new ObjHandler<Const>();
 ObjHandler<BinOp> f = new ObjHandler<BinOp>();
 Trace.expect(
  new Call("Cell.setFormula", y, yExpr, null,
   new CallStub("Parser",null,"parse",args(s,yExpr),f,
    new CombPar(
     new CombStrict(
      new Call("Spreadsheet.getCellByName",s,"x",x),
      new Constr("CellRef", r, x)),
     new Constr("Const", c, cVal)),
     new Constr("BinOp", f, args(op,r,c)))));
 y.setFormula(yExpr);
 Trace.finalCheck();
// internal interact. checking omitted in other cases
 assertEquals(yVal, y.getValue());
}
public void testSpreadsheetAPI_0() {
 testSpreadsheetAPI(1.0, '+', 2.0, 3.0, "x + 2");
} // 3 similar methods ommitted for other test data
public void testSpreadsheetCLI(){
 // code for checking internal interactions omitted
  Console.start(SpreadsheetCLI.class, null);
  Console.enter("x = 1");
  Console.enter("x + 1");
  assertEquals(2.0, Console.check());
  Console.enter("");
  Console.stop();
}
}
```

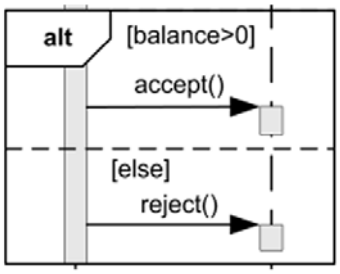
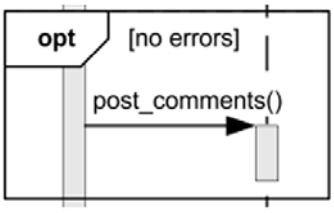
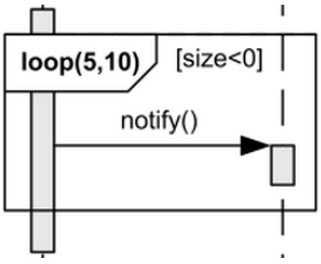
Excerto de Código 2.1: Amostra de código de teste gerado para o mecanismo de folhas de cálculo.

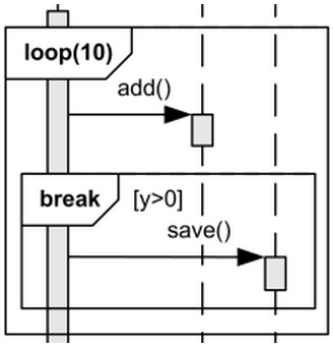
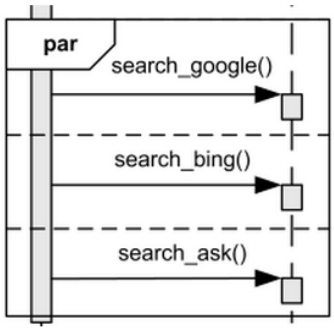
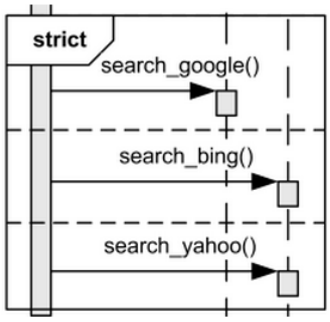
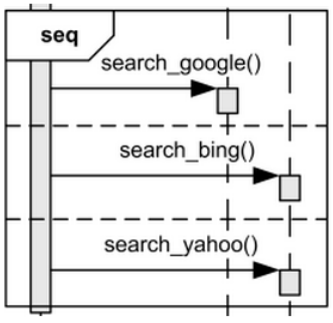
E. Ciclo iterativo de implementação e teste dos métodos

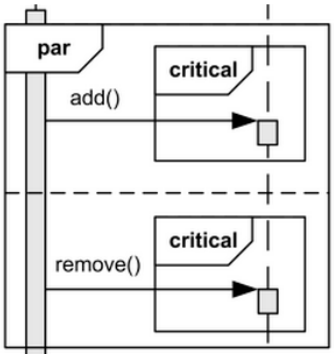
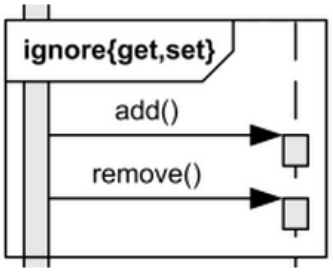
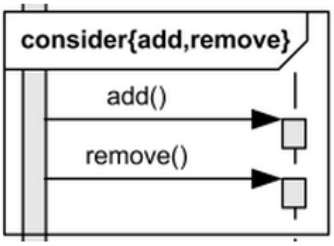
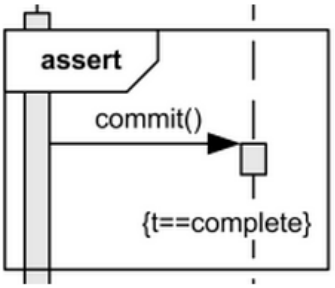
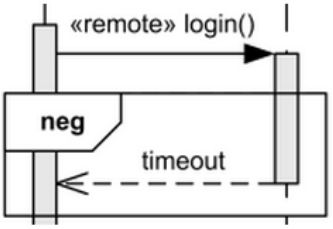
Na fase atual, os métodos das classes Java estão definidos e prontos a ser implementados para validarem os testes gerados. Entra-se agora no ciclo iterativo de implementação de métodos, execução dos testes e posterior refabricação ou correção dos métodos, conforme os resultados dos testes. A figura 2.5 demonstra o resultado da execução de um teste que ainda contém um erro que necessita de ser corrigido para todos os testes passarem.

2.3 Funcionalidades suportadas

Nesta secção são analisadas um conjunto de funcionalidades para verificação se estas são suportadas nos diagramas de sequência UML e na implementação Java. A tabela 2.1 apresenta para cada funcionalidade analisada, um exemplo e a indicação se esta é suportada ou não pelo FEUP-SBT-1.0.

Funcionalidade	**Exemplo**	**Suportado?**
Fragmento combinado: **alt** (*alternative*)		Parcialmente. É suportado o fragmento *"alt"*, mas sem definição de condições e sem utilização do operador *"else"*.
Fragmento combinado: **opt** (*option*)		Sim, incluindo suporte para condições de guarda, implementado no decorrer deste projeto.
Fragmento combinado: **loop**		Parcialmente. É suportado o fragmento *"loop"*, com indicação do número mínimo e máximo de iterações mas sem definição de condições.

Fragmento combinado: **break**		Não.
Fragmento combinado: **par** (*parallel*)		Parcialmente. Não prevê execução concorrente, apenas intercalada (*interleaved*).
Fragmento combinado: **strict** (*Strict Sequencing*)		Sim.
Fragmento combinado: **seq** (*Weak Sequencing*)		Parcialmente. Como não prevê execução concorrente, reduz-se ao fragmento combinado *"strict"*.

Fragmento combinado: **critical** (*Critical Region*)		Não.
Fragmento combinado: **ignore**		Não.
Fragmento combinado: **consider**		Não.
Fragmento combinado: **assert** (*Assertion*)		Não.
Fragmento combinado: **neg** (*Negative*)		Não.

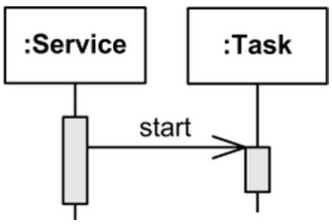
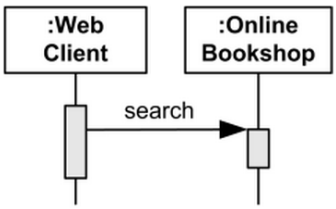
Concorrência (*threads*): mensagens assíncronas		Não.
Mensagens Síncronas		Sim.

Tabela 2.1: Funcionalidades suportadas pelo protótipo

2.4 Limitações

Através da análise do protótipo FEUP-SBT-1.0, apresentada nas secções anteriores, foi possível registar um conjunto de limitações que afetam o protótipo. Desde logo, as limitações em algumas funcionalidades dos diagramas de sequência UML acima referidas podem aumentar a probabilidade do utilizador criar um modelo estrutural não suportado.

Na versão 1.0 do protótipo, a estruturação do seu código fonte não é a desejada, devendo a sua manutenibilidade, isto é, a facilidade de manutenção do código desenvolvido, ser melhorada. No entanto, a limitação considerada mais relevante prende-se com a usabilidade do protótipo devido ao facto de os utilizadores serem forçados a sair da ferramenta Enterprise Architect e eles próprios compilarem e executarem os testes. Este pode ser um fator desmotivante para a utilização do produto, que deve ser contornado.

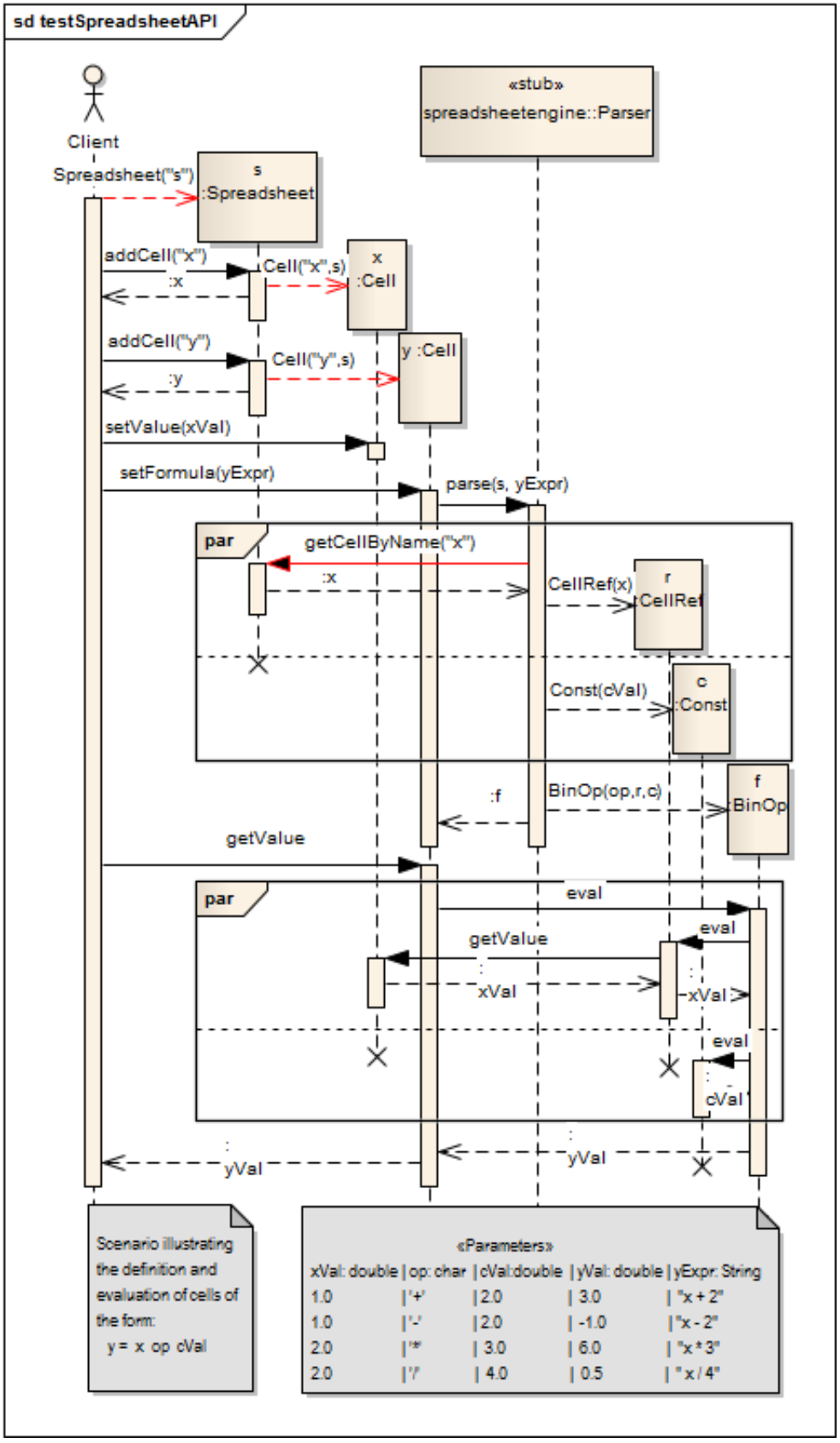


Figura 2.3: Diagrama de sequência UML para teste da API do mecanismo de folhas de cálculo.

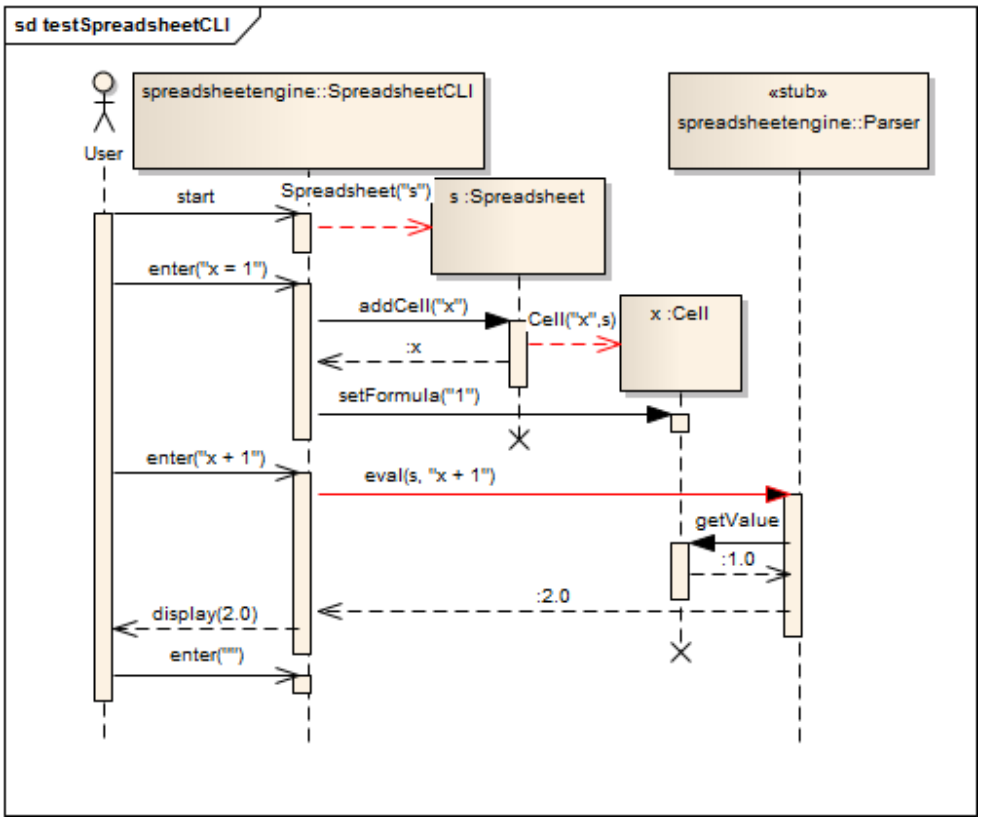


Figura 2.4: Diagrama de sequência UML para teste da Interface do mecanismo de folhas de cálculo.

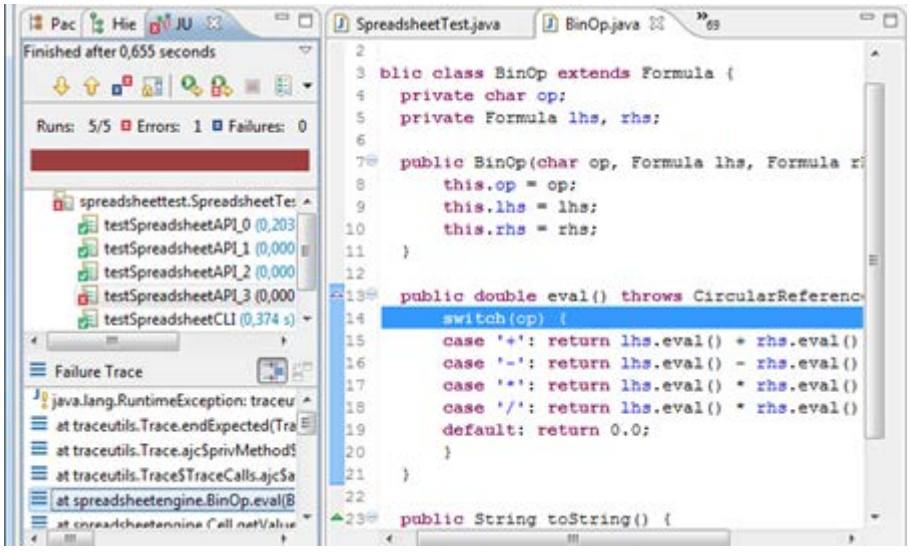


Figura 2.5: Execução e verificação dos testes gerados.

Capítulo 3

Análise do estado da arte

Neste capítulo é apresentado o estado da arte após revisão bibliográfica sobre o tema "automação de testes baseados em diagramas de sequência UML". São descritas individualmente as abordagens descobertas e, por fim, é elaborada uma análise comparativa destas abordagens com o protótipo FEUP-SBT-1.0.

3.1 Objetivos

No capítulo 2 foi apresentado o protótipo FEUP-SBT-1.0 para a geração automática de testes unitários a partir de diagramas de sequência UML, verificando as interações internas. Para encontrar possíveis ferramentas ou abordagens que realizassem uma função idêntica foi efetuada uma revisão bibliográfica sobre o tema "automação de testes baseados em diagramas de sequencial UML", com especial foco na verificação das interações internas. A elaboração desta revisão bibliográfica tinha como principal objetivo descobrir noutras abordagens ideias para possíveis melhoramentos do protótipo FEUP-SBT-1.0. Desta pesquisa resultaram 3 abordagens que se aproximam da seguida pelo FEUP-SBT-1.0 e que de seguida são apresentadas. De salientar que se encontraram muito mais abordagens de geração de testes a partir de diagramas UML, mas que consideram outros diagramas (diagramas de estados, diagramas de atividades) ou outros aspetos a montante na geração de teste (geração de caminhos de teste, geração de dados de teste).

3.2 SeDiTeC

Uma das abordagens mais citadas na área de geração de testes baseados em diagramas de sequência UML é apresentada por Falk Fraikin e Thomas Leonhardt na *"17th IEEE International Conference on Automated Software Engineering (ASE 2002) "*, com o documento *"SeDiTeC – Testing Based on Sequence Diagrams"* [FL02]. No referido documento é apresentada uma abordagem para teste automatizado a aplicações orientadas a objetos e a ferramenta SeDiTeC que aplica esses conceitos para aplicações Java. A motivação para o desenvolvimento desta ferramenta assemelha-se à motivação para o desenvolvimento do FEUP-SBT-1.0, isto é, fomentar nos

engenheiros de *software* uma utilização ativa dos modelos UML, oferecendo soluções que lhes permitam beneficiar com o tempo despendido no desenho dos modelos.

O SeDiTeC utiliza diagramas de sequência UML, que são complementados por dados de teste, como especificação de teste que posteriormente pode ser integrada no processo de desenvolvimento desde o seu início. No entanto, o SeDiTeC não suporta o desenho destes diagramas, integrando-se com a ferramenta Together Control Center [Foc12] para a realização desta tarefa. O SeDiTeC liga-se à ferramenta Together através da sua API, podendo executar diretamente dentro da ferramenta diagramas de sequência individuais ou então, caso seja mais que um diagrama, exportá-los em formato XML que posteriormente podem ser carregados para o SeDiTeC. Posteriormente, o SeDiTeC permite ao utilizador combinar diagramas de sequência, definir diferentes conjuntos de dados de teste e executar os próprios testes. A execução dos testes gera um novo diagrama de sequência, chamado diagrama de sequência observado, que é automaticamente comparado com o diagrama de sequência original, verificando a existência de possíveis erros. Em certas situações, esta verificação exige a colaboração do utilizador, caso o diagrama de sequência observado possua alguns detalhes adicionais que possam não ter sido definidos pelo utilizador no diagrama original, não podendo ser considerados à partida erros, dado que o utilizador não tem por obrigação apresentar todos os detalhes de implementação nos seus modelos comportamentais.

O SeDiTeC possibilita ainda a definição de *stubs*, permitindo envolver nos testes, métodos ainda não implementados e são também verificadas automaticamente as interações internas, analisando o traço de execução. No entanto, existem algumas limitações, desde logo por já não se encontrar disponível. Nem todos os diagramas de sequência podem ser testados, não sendo suportadas as funcionalidades do UML 2.x e existindo uma lista de requisitos que um diagrama de sequência tem de cumprir para ser considerado testável.

3.3 Abordagem utilizando Arquitetura Dirigida por Modelos (MDA)

A.Z. Javed, P.A. Strooper e G.N. Watson apresentaram uma alternativa diferente no *"2nd International Workshop on Automation of Software Test (AST 2007)"* com o documento *"Automated generation of test cases using Model-Driven Architecture"* [JSW07], que demonstra uma abordagem que utiliza os conceitos da Arquitetura Dirigida por Modelos (MDA) para geração de testes unitários a partir de diagramas de sequência UML.

A força motriz por trás da Arquitetura Dirigida por Modelos (MDA) é a ideia de que o *software* irá ser implementado em mais que uma plataforma, diferentes entre si [MSUW02]. Uma utilização simples da MDA é modelar uma aplicação numa linguagem de modelação independente de plataformas. O modelo independente de plataforma (PIM – *platform-independent model*) pode ser posteriormente convertido para um modelo para uma plataforma específica (PSM – *platform-specific model*) através da definição de um conjunto de regras de transformação, tendo em vista uma plataforma específica, por exemplo, Java [Poo01].

A figura 3.1 mostra uma visão geral da abordagem apresentada, assentando em 2 passos principais. No primeiro passo, o diagrama de sequência UML é transformado em modelo de teste

independente de linguagem (xUnit) através de transformação horizontal. Esta transformação mantém o grau de abstração, tratando-se assim de uma transformação de PIM para PIM. No segundo passo o modelo de teste (xUnit) é convertido num código de teste concreto e executável através de transformação vertical, isto é, PIM para PSM.

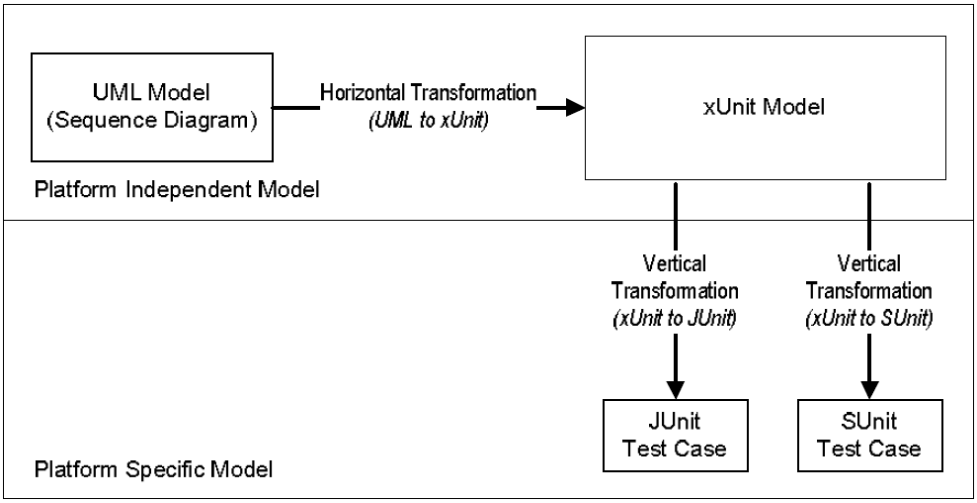


Figura 3.1: Visão geral da abordagem MDA para geração de testes. [JSW07]

Este processo tem envolvidas uma série de aplicações, todas integrantes do ambiente de desenvolvimento de *software* (IDE) Eclipse [Ecl12]. O diagrama de sequência tem de ser convertido numa sequência de chamadas a métodos (SMC) manualmente, que depois é introduzida na Eclipse Modeling Framework [Fra12a], uma extensão para Eclipse. De seguida, o SMC é convertido em xUnit utilizando outra extensão para Eclipse, o TefKat [Tef12], tratando-se de um mecanismo de transformação de modelos baseado na Eclipse Modeling Framework através da definição de regras (*rules*). Finalmente, o modelo de teste (xUnit) é convertido em código de teste (JUnit ou SUnit) através de nova extensão para o Eclipse MOFScript [MOF12], uma ferramenta para transformação de modelo para texto. Por fim, a execução dos testes unitários gerados fica ao encargo do utilizador. Todo este processo está detalhado na figura 3.2, com todos os artefactos gerados ao longo do processo.

A grande vantagem desta abordagem é, já estando disponível para JUnit e SUnit, poder ser potenciada para várias outras linguagens, uma vez que as transformações verticais, de modelo para código, são lineares, variando apenas as especificações de linguagem. Esta ferramenta está também disponível sem custos, pronta a ser utilizada. Porém, existem algumas contrariedades. Desde logo, os dados sendo especificados separadamente do diagrama de sequência, são posteriormente codificados diretamente no código de teste (JUnit ou SUnit) o que obriga o utilizador efetuar as transformações modelo para texto, sempre que pretender alterar os dados de teste. Também não são suportadas as funcionalidades da versão 2.x de UML nem a geração de *stubs*. Por último, as interações internas são capturadas numa lista com o traçado de execução da aplicação, no entanto, estas têm de ser manualmente testadas pelo utilizador.

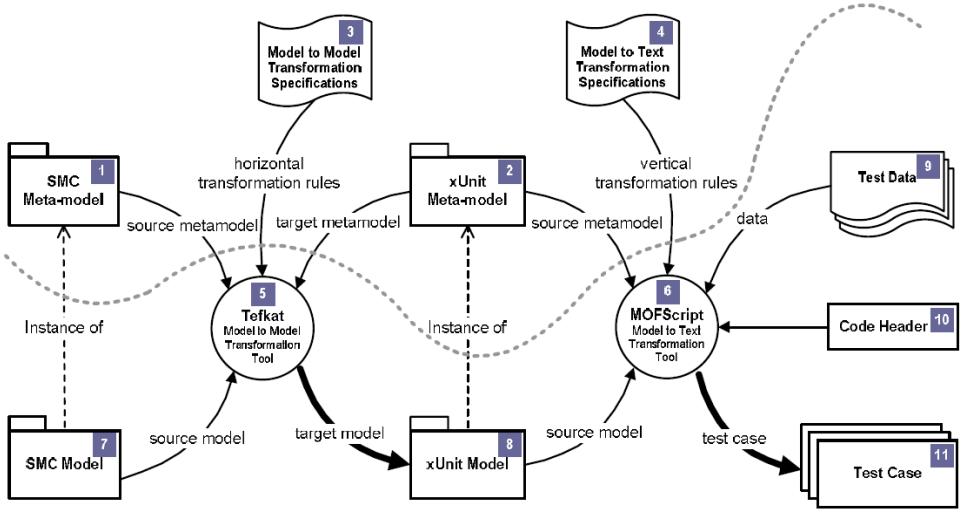


Figura 3.2: Processo detalhado da abordagem MDA para geração de testes. [JSW07]

3.4 SCENTOR

Na mesma conferência de apresentação da abordagem anterior, o *"2nd International Workshop on Automation of Software Test (AST 2007)"*, foi também apresentada por Jeremiah Wittevrongel e Frank Maurer uma ferramenta que cria *drivers* de teste funcionais a partir de diagramas de sequência, criado para aplicações de *e-business*. Esta ferramenta encontra-se descrita no documento *"SCENTOR: Scenario-Based Testing of E-Business Applications"* [WM07].

Começando pela definição de *driver* de teste, "um *driver* é uma classe que simula o programa principal do elemento a ser testado, ou seja, faz chamadas ao módulo a ser testado" [Bia06]. A ferramenta SCENTOR encontra-se disponível na Internet (embora atualmente se encontre *offline*) através de um *web browser* para a interface com o utilizador e de um *web server* para lidar com pedidos e respostas e permite a geração, compilação e execução de *drivers* de testes através da sua interface.

O processo em que se insere o SCENTOR encontra-se exemplificado pela figura 3.3. Os modelos UML têm de ser desenhados numa ferramenta externa que posteriormente possa exportar os diagramas num formato XMI. Para esse efeito, foi utilizada a ferramenta Rational Rose [Ros12]. De seguida, os ficheiros XMI são carregados no SCENTOR através da sua interface (*web site*). É também introduzido código de configuração, caso seja necessário executar algumas operações antes de executar propriamente o código de teste e, por fim, são introduzidos os resultados esperados dos métodos utilizados (em Java) para depois serem comparados. Posteriormente, a ferramenta SCENTOR gera os *drivers* de teste em Java que podem ser compilados e executados diretamente a partir da ferramenta, interagindo com o servidor Enterprise JavaBeans (EJB) [Jav12] nesse processo.

Tratando-se de uma ferramenta que utiliza uma abordagem diferente da do protótipo FEUP-SBT-1.0, vocacionada para aplicações de *e-business*, a sua comparação torna-se difícil. Desde

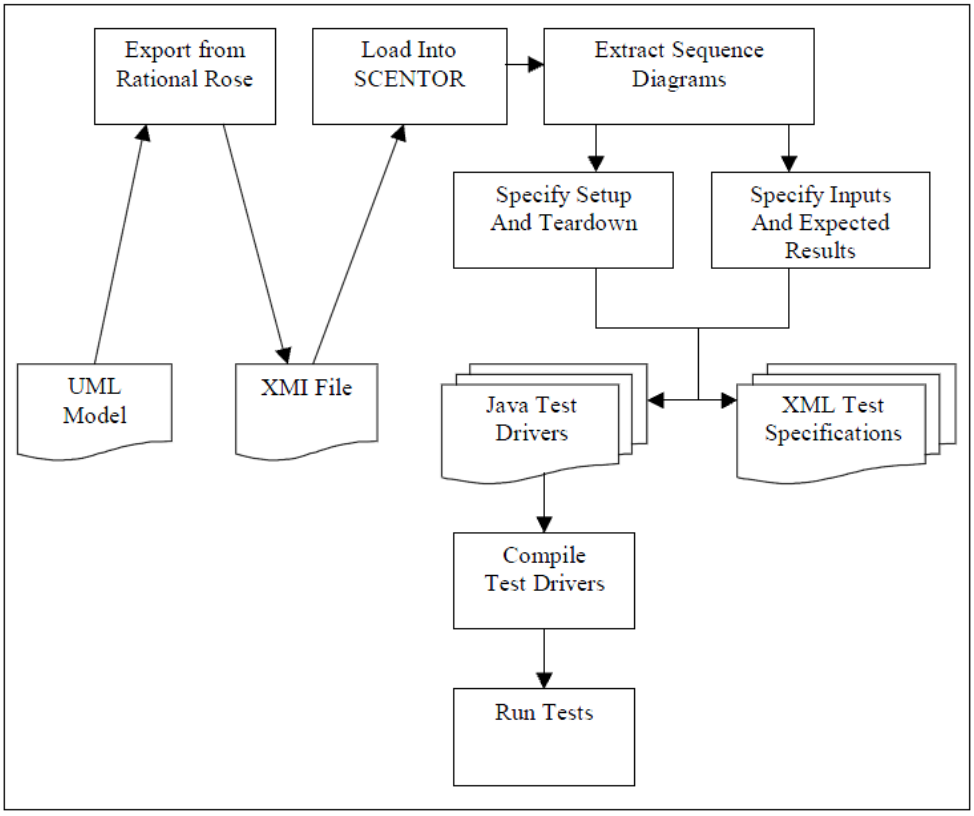


Figura 3.3: Abordagem em que se insere o SCENTOR. [WM07]

logo, a ferramenta já não se encontra disponível, impedindo o seu teste direto. Não há qualquer informação sobre a versão de UML suportada, nem sobre a verificação de consistência modelos desenhados.

3.5 Análise comparativa

Analisadas individualmente cada uma das abordagens descobertas que se relacionam com o protótipo FEUP-SBT-1.0, é agora apresentada uma tabela com os pontos considerados mais relevantes de comparação de cada uma das delas com o FEUP-SBT.1.0. São utilizados como identificadores um sinal positivo verde e um sinal negativo vermelho para serem melhor percetíveis os pontos fortes e fracos, respetivamente, de cada uma das abordagens.

	SeDiTeC	**Abordagem MDA**	**SCENTOR**	**FEUP-SBT-1.0**
Integração	Together	Eclipse	Rational Rose	Enterprise Architect

Abordagem da Automação de Testes	Executa diretamente diagramas de sequência.	Transforma em modelo de teste e depois em código de teste.	Gera *drivers* de teste.	Gera JUnit.
Feedback	Comparação do diagrama inicial com o resultado da execução.	Testes JUnit executados pelo utilizador.	Resultados apresentados pela ferramenta.	Testes JUnit executados pelo utilizador. ⊖
Geração de stubs	Sim. ⊕	Não. ⊖	Não. ⊖	Sim. ⊕
Versão UML suportada	UML 1.x ⊖	UML 1.x ⊖	-	UML 2.x ⊕
Disponibilidade	Não disponível. ⊖	Disponível sem custos. ⊕	WEB: *offline*. ⊖	Protótipo interno.
Dados de teste	Separados do diagrama. ⊕	Especificados separadamente do diagrama ⊕, codificados hardcoded. ⊖	Separados do diagrama. ⊕	Cenários parametrizados e dados de teste separados.
Linguagens Alvo	Java.	Java, SmallTalk (potencial para várias outras). ⊕	Java.	Java.
Captura das interações internas	Captura interações internas. ⊕	Captura em listagens. ⊕	Não. ⊖	Captura recorrendo a AOP. ⊕
Verificação das interações internas	Verifica automaticamente. ⊕	Verificação manual pelo utilizador. ⊖	- ⊖	Verifica automaticamente atendendo à conformidade definida. ⊕

Interface de Utilizador (UI)	Desenho de diagramas no Together. UI da aplicação não disponível.	Definição da SMC no Eclipse Modeling Framework.	Desenho de diagramas no Rational Rose. Website para execução dos testes.	Desenho dos diagramas e geração dos testes através dos menus do EA.
Verificação da Consistência do Modelo	Não. ⛔	Não. ⛔	Não. ⛔	Verifica consistência e completude dos modelos. ✅

Tabela 3.1: Tabela comparativa das abordagem com o FEUP-SBT-1.0.

Desta análise comparativa que resume o trabalho de investigação efetuado para verificação do estado da arte sobre o tema "geração automática de testes unitários a partir de diagramas de sequência UML"foi possível verificar que as abordagens que mais se relacionam com o protótipo FEUP-SBT-1.0 são o SeDiTeC e a abordagem baseada na Arquitetura Dirigida por Modelos. Por se tratarem de abordagens bastante distintas, têm pontos fortes e fracos distintos, não sendo evidente que uma sobressaia em relação outra.

Em relação à comparação direta com o protótipo FEUP-SBT-1.0, a linguagem alvo dos testes é um ponto a favor para a abordagem MDA, que pode ser potenciada para várias linguagens enquanto o protótipo foi pensado para Java. Ainda assim, trata-se de uma boa ideia para trabalho futuro no aperfeiçoamento do protótipo. Em termos de usabilidade, crê-se que o método utilizado pelo SeDiTeC, que compara o diagrama original com o fruto da execução, não forçando o utilizador a compilar e executar os testes é uma mais valia e é a prioridade em termos de melhoramentos para o protótipo.

Capítulo 4

Conceção e implementação

No presente capítulo são apresentados os detalhes de conceção e implementação da ferramenta FEUP-SBT-2.0. É exposta a reformulação da abordagem e modo de funcionamento geral. É ainda especificada a estrutura dos componentes da ferramenta, o *add-in* para EA e a biblioteca para análise do traçado e cobertura. Por fim, é apresentada a concretização das principais funcionalidades do FEUP-SBT-2.0.

4.1 Abordagem e modo de funcionamento geral

Da análise do protótipo existente e da revisão bibliográfica apresentados nos capítulos anteriores foi possível comprovar algumas limitações e necessidades que à partida eram apontadas à versão 1.0 do protótipo FEUP-SBT. As componentes do protótipo foram assim alvo de um processo de refabricação (*refactoring*), principalmente o gerador de testes mas também a biblioteca *TracingUtilities*, aperfeiçoando as questões de manutenibilidade do código e desempenho da execução do protótipo.

No entanto, o principal foco foi sempre contornar os problemas de usabilidade do protótipo presentes no FEUP-SBT-1.0, onde o utilizador tinha total responsabilidade em executar os testes gerados e analisar os seus resultados. Assim foi desenvolvida a nova funcionalidade que permite a execução dos testes e apresentação dos respetivos resultados diretamente na ferramenta Enterprise Architect.

Para um melhor entendimento das alterações que esta nova funcionalidade opera na abordagem em que se insere o FEUP-SBT, encontra-se na figura 4.1 ilustrado o esquema representativo da abordagem do FEUP-SBT-2.0. Além de registadas as refabricações de ambos os componentes através da etiqueta "*v2*", é visível que a geração e execução de código de teste e análise de resultados compõem uma única etapa (3/5) nesta nova abordagem. Aos diagramas de sequência UML chegam os resultados da execução do código de teste e, através destes, o utilizador tem acesso aos resultados.

Para demonstração da modo de funcionamento geral, foi elaborado um diagrama de comunicação que visa apresentar a comunicação estabelecida entre todos os artefactos envolvidos no

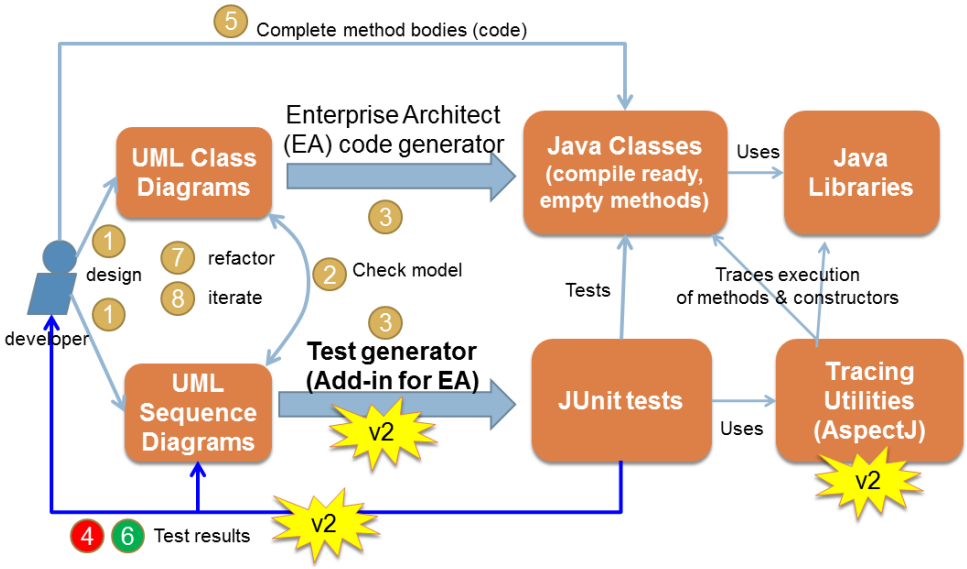


Figura 4.1: Abordagem do FEUP-SBT-2.0.

processo. Este diagrama encontra-se ilustrado na figura 4.2, sendo de seguida apresentada uma breve descrição de todas as etapas nele inseridas. Assinalados a cor garrida estão os dois componentes que compõem o FEUP-SBT-2.0, o *add-in* "*TestGenerator*"para o Enterprise Architect e a biblioteca Java para análise do traçado e cobertura "*TracingUtilities*".

O processo inicia-se através da construção, por parte do utilizador, do modelo UML, mais especificamente os diagramas de sequência, bem como a aplicação a testar (*etapa 1*). A *etapa 2* é também da responsabilidade do utilizador e passa por mandar executar o *plug-in* FEUP-SBT-2.0, que depois interpreta o modelo UML desenhado (*etapa 3*). De seguida, o *plug-in* gera o código de teste JUnit e instala a biblioteca auxiliar *TracingUtilities* (*etapa 4*) e posteriormente executa o *Test Runner* do JUnit (*etapa 5*). Este *Test Runner* executa o código de teste JUnit (*etapa 6*), que por sua vez invoca a aplicação a testar (*etapa 7*), utilizando a biblioteca *TracingUtilities* (*etapa 8*) para fazer o traçado da aplicação a testar (*etapa 9*). Na fase final, o *Test Runner* JUnit, gera os resultados (*etapa 10*), que são lidos pelo *plug-in* (*etapa 11*), utilizando-os para colorir o modelo UML (*etapa 12*). Após esta etapa, o modelo apresenta as edições respetivas para nele serem analisados os resultados da execução pelo utilizador.

4.2 Estrutura dos componentes da ferramenta

O FEUP-SBT-2.0 é composto por dois componentes, que sendo independentes entre si, foram ambos construídos para funcionar de forma colaborante. Ambos os componentes já existiam na versão 1.0 do FEUP-SBT, no entanto sofreram modificações no desenvolvimento das novas funcionalidades. O FEUP-SBT é assim composto pelo gerador e executor de testes (*TestGenerator*) e pela biblioteca de análise do traçado de execução (*TracingUtilities*). Nesta secção, ambos serão apresentados, abordando as suas arquiteturas e os detalhes mais relevantes. Um dos objetivos

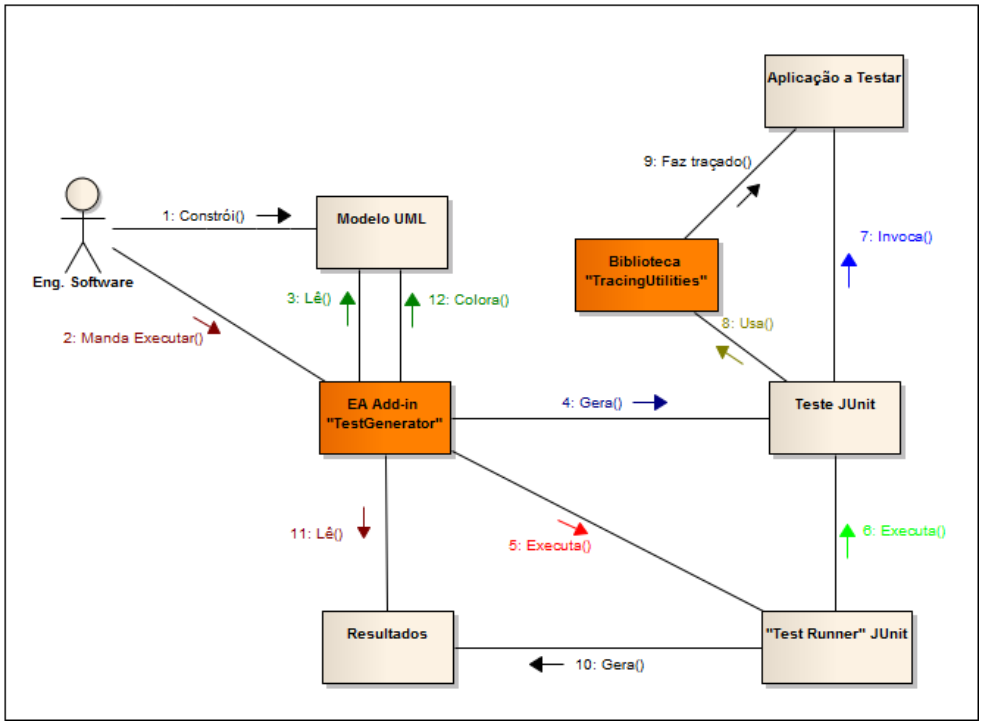


Figura 4.2: Diagrama de comunicação representativo do funcionamento geral do protótipo FEUP-SBT-2.0.

desta dissertação passava por realizar um processo de refabricação (*refactoring*) aos componentes do protótipo FEUP-SBT-1.0, sobretudo ao nível da manutenibilidade e robustez. Encontrando-se a biblioteca *TracingUtilities* com uma arquitetura bem estruturada, o principal foco das alterações residiu no *add-in TestGenerator*.

4.2.1 Gerador e executor de testes (*TestGenerator*)

O gerador e executor de testes (*TestGenerator*) é um *add-in* para a ferramenta Enterprise Architect que permitia a geração automática de testes unitários JUnit3 a partir de diagramas de sequência UML na versão 1.0 do protótipo, adicionando-se agora as funcionalidades de execução dos testes e análise de cobertura com apresentação dos resultados diretamente a partir do modelo. Este *add-in* foi implementado em C# recorrendo ao pacote de desenvolvimento de *software* (SDK) da ferramenta EA, fazendo uso da API que esta disponibiliza. Este pacote permite personalizar e estender as funcionalidades nativas do Enterprise Architect.

O Enterprise Architect disponibiliza um modelo de objetos no qual são codificados os modelos UML. Este modelo de objetos é acessível via API o que permite aceder, manipular, modificar e criar os modelos UML da ferramenta. Foi desta forma possível percorrer os diagramas de sequência, criar os respetivos testes e editar o modelo conforme o resultado da execução dos testes. Um resumo do modelo de objetos do Enterprise Architect está defindo na figura 4.3 [Spa10].

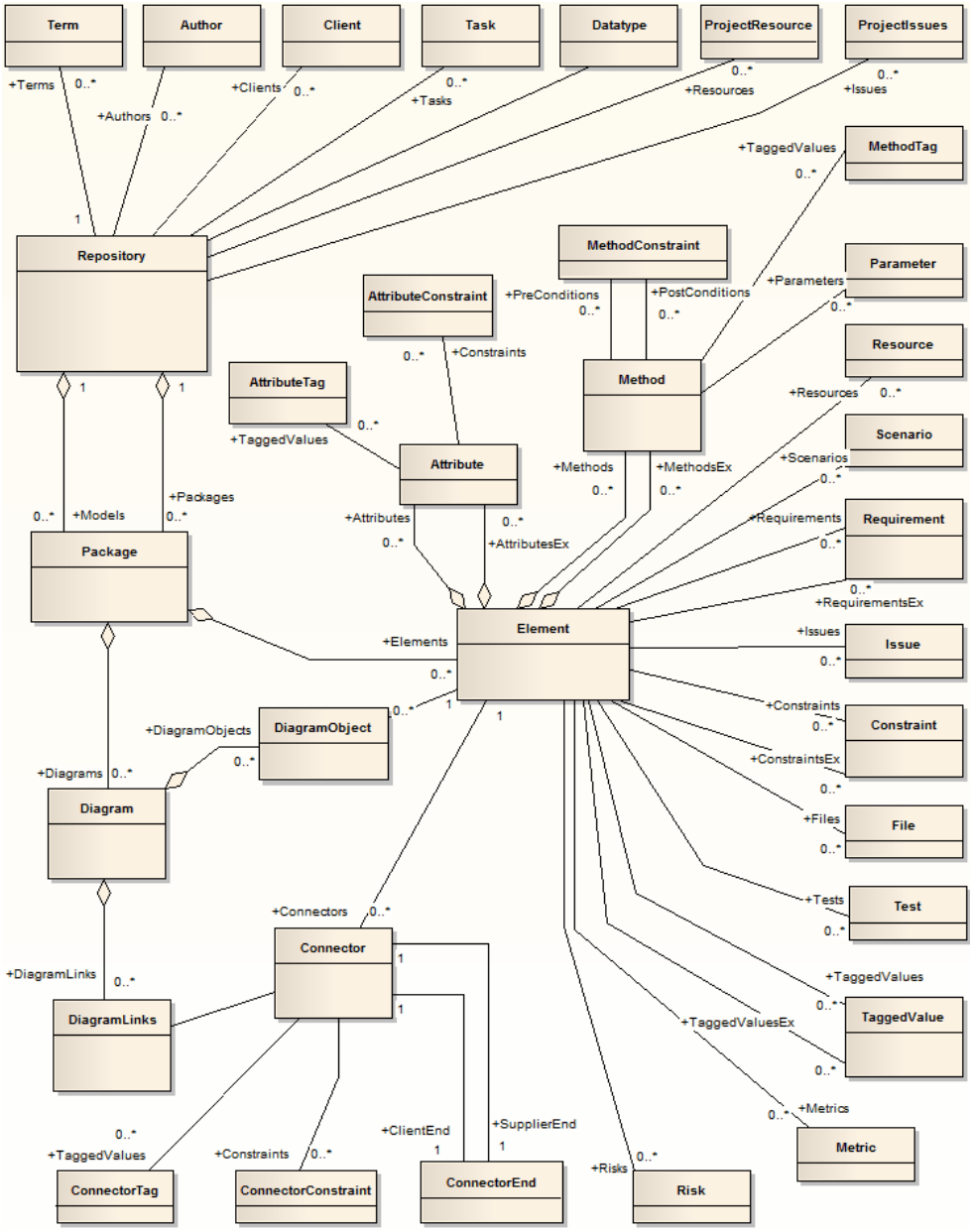


Figura 4.3: Modelo de Objetos do Enterprise Architect.

Das classes apresentadas, destacam-se o `Repository` (repositório) que é o principal contentor de todas as estruturas, como `Models` (modelos), `Packages` (pacotes) e `Elements` (elementos), também estes bastante utilizados para a recolha da informação necessária do modelo. Uma outra classe extremamente importante na implementação foi o `Connector` (conector) que representa mensagens no modelo e que contém métodos de edição utilizados para a coloração de mensagens.

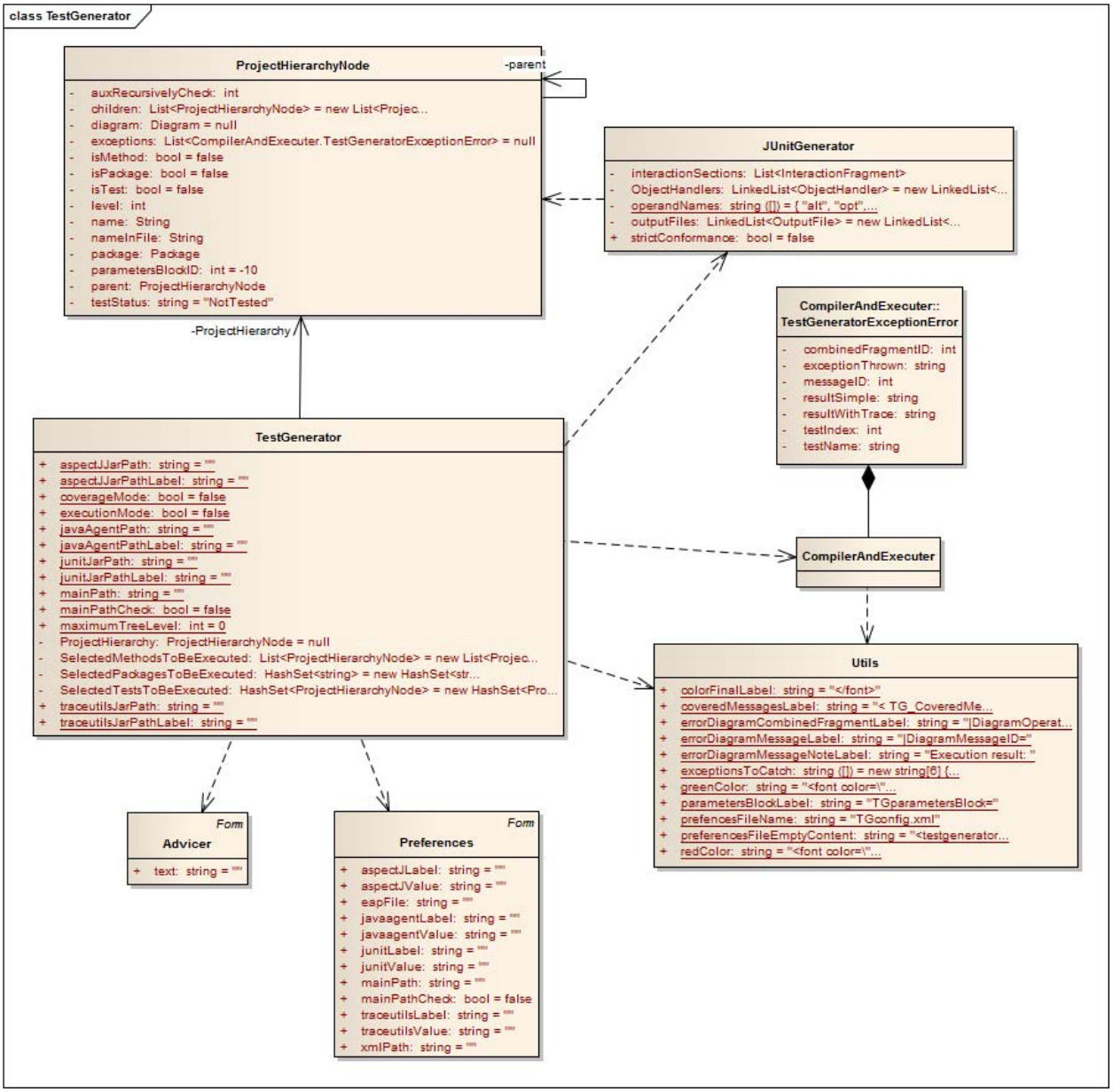


Figura 4.4: Diagrama de classes do *add-in* (*TestGenerator*), excluindo operações.

Arquitetura

O diagrama de classes presente em 4.4 ilustra a arquitetura do *add-in*, sendo de seguida efetuada uma breve descrição de cada classe.

TestGenerator é a principal classe do *add-in*, contendo os métodos que estabelecem a ligação entre o Enterprise Architect e o *add-in* desenvolvido e a construção de toda a sua estrutura. Contém uma série de atributos vitais para o funcionamento do *add-in*, como a estrutura *ProjectHierarchy* que contém toda a informação do modelo a testar. São a partir desta classes invocados todas as funções do *add-in*, como o carregamento da informação, geração, compilação e execução do

código de teste, entre outras.

ProjectHierarchyNode é a classe utilizada para representar cada nó da estrutura de pacotes do modelo, isto é, cada um dos pacotes que compõem um modelo desenvolvido no Enterprise Architect, desde o pacote principal *Model*, até ao pacote no nível mais profundo. Será mais adiante no documento abordada em mais detalhe os motivos da utilização desta classe.

JUnitGenerator, tal como o nome indicia, contém todos os métodos e estruturas necessários para a geração do código de teste a partir dos diagramas de sequência UML. Contém inúmeras chamadas à API do Enterprise Architect para recolher toda a informação necessária para a correta geração do código de teste.

CompilerAndExecuter conta com a implementação necessária para a compilação e execução do código de teste e interpretação dos resultados das referidas operações. Conta com um método executor de comandos do sistema, e uma série de métodos de complexidade mais elevada que varrem o resultado da execução, armazenando a informação sobre os possíveis erros, numa estrutura denominada **TestGeneratorExceptionError**. Esta estrutura servirá posteriormente de fonte de informação para edição do modelo.

Advicer e **Preferences** representam classes para a criação de janelas *Windows Forms* nas quais é apresentada informação ao utilizador. A primeira serve para avisos simples, por exemplo, de indicação de erros na compilação. A segunda apresenta uma janela contendo todas as informações de configuração do *add-in*.

Utils é uma classe de métodos utilitários, desde a criação de janelas para seleção de ficheiros a métodos de pesquisa de palavras-chave num texto, utilizados na interpretação de resultados de execução.

4.2.2 Biblioteca para análise do traçado de execução (*TracingUtilities*)

Conforme já foi referido anteriormente, *TracingUtilities* é uma biblioteca em Java, desenvolvida no projeto do protótipo FEUP-SBT-1.0 e tem por missão possibilitar que os testes gerados possam verificar as interações internas dos seus objetos, para além das interações externas com o cliente. Nesta secção é apresentada a arquitetura da biblioteca e de que forma a Programação Orientada a Aspetos intervém no processo de análise do traçado realizado pela aplicação.

4.2.2.1 Arquitetura

Na figura 4.7 está representada a arquitetura da biblioteca *"Tracing Utilities"* através do seu diagrama de classes UML, incluindo as respetivas classes e ligações. Para permitir uma melhor compreensão do funcionamento da biblioteca é apresentada uma descrição de cada uma das classes que a compõe.

CallNode é uma classe abstrata que representa uma chamada ou um fragmento combinado presente no modelo, sendo assim superclasse de *Call* e *CombOperator*. É utilizada para representar um conjunto de interações, com chamadas e/ou fragmentos combinados, possibilitando a

criação de uma estrutura em árvore, contendo as chamadas encaixadas realizadas a partir da *CallNode* origem.

Call é uma subclasse de *CallNode* e representa uma mensagem do modelo que será codificada numa chamada no código de teste. Contém a classe e objeto alvo, bem como o método, os parâmetros e valor de retorno. A estes atributos foram ainda adicionados os identificadores do elemento respetivo no Enterprise Architect para serem implementadas as novas funcionalidades. *Call* é ainda superclasse de *CallStub*.

CallStub é uma subclasse de *Call* e tem por objetivo lidar com as chamadas que envolvam métodos ainda não implementados, denominados métodos *stub*.

ActualCall é uma classe auxiliar que permite definir uma chamada executada, intercetada através de aspetos. A modelação das chamadas intercetadas através desta classe permitem uma comparação com as chamadas esperadas mais simples.

CombOperator é uma classe abstrata representando os vários fragmentos combinados suportados pela ferramenta, dos quais é superclasse. Um fragmento combinado é usado para agrupar conjuntos de mensagens e para mostrar o fluxo condicional num diagrama de sequência. A exemplo da classe *Call*, também foram adicionados os identificadores de elemento no Enterprise Architect para implementação das novas funcionalidades.

CombOpt, **CombAlt** e **CombLoop** são subclasses de *CombOperator* e representam cada uma um fragmento combinado para definição de condições de ocorrência de mensagens. *CombOpt* representa o fragmento *opt*, contendo uma sequência de mensagens que dependendo de uma certa condição poderão ou não ocorrer. Já *CombAlt* representa o fragmento *alt*, que se assemelha ao operador *opt*, criando no entanto um conjunto de alternativas, cada uma contendo um conjunto de mensagens e uma condição que a valide, sendo apenas uma alternativa executada. A classe *CombLoop* representa um ciclo, onde um conjunto de mensagens serão executadas um determinado número de vezes. É possível a representação de vários tipos de ciclos através desta classe.

CombStrict, **CombPerm** e **CombInter** são também, a exemplo das anteriores, subclasses de *CombOperator*, para definição da ordem de ocorrência das mensagens. *CombStrict* representa o fragmento combinado *strict* enquanto que a sua subclasse **CombSeq** representa o fragmento combinado *seq*. Estes dois fragmentos modelam um conjunto de mensagens que ocorrerá por uma determinada ordem. Atendendo a que não se prevê execução concorrente, o fragmento *seq* reduz-se ao fragmento *strict*. *CombPerm* representa um conjunto de mensagens que poderá ocorrer por uma qualquer ordem. Por fim, *CombInter* e a sua subclasse **CombPar** (representando o fragmento *par*), encerram um conjunto de mensagens cuja execução poderá ocorrer de forma entrelaçada ou paralelamente, respetivamente.

ConformanceChecker é uma das principais classes desta biblioteca, onde estão implementados os diversos métodos utilizados na análise de conformidade da execução obtida com a esperada. É ainda superclasse de *Trace*.

Trace é a classe contendo a definição dos vários aspetos para intercetar as mensagens do sistema, para posterior análise de conformidade pela sua superclasse ConformanceChecker. É nesta

classe que foi definida a lista de identificadores de mensagens executadas, utilizada no funcionalidade de análise de cobertura.

InteracTestCase é uma classe que estende a classe abstrata *TestCase* [1], usada para definir um conjunto de casos de teste. Tem definidos os métodos invocados pelos testes unitários gerados, como as várias versões do método `assertEquals`.

Console tem por objectivo simular a interação com o utilizador. Possui como membros principais duas listas, uma para os valores de entrada e outra para os respetivos valor de saída. Contém igualmente métodos para introdução e leitura de valores e uma classe encaixada para tratamento de exceções. Nesta classe também foram adicionados identificadores de mensagens para implementação das novas funcionalidades de execução dos testes.

ConformanceException é uma classe abstrata, sendo superclasse de **ArgumentException**, **MissingCallException**, **ReturnValueException** e **UnexpectedCallException**, definindo assim um conjunto de exceções criadas para representar situações de erro previstas. Tal como os nomes sugerem, *ArgumentException* representa um erro nos argumentos duma função e *MissingCallException* uma mensagem que era prevista e que não ocorreu. Em relação a *ReturnValueException*, acontece quando existe um erro no valor de retorno e *UnexpectedCallException* quando uma chamada que não era prevista foi detetada.

ObjectHandler é uma classe genérica criada para representar objetos que necessitem de ser utilizados antes da sua definição. Esta situação acontece quando é necessário definir uma árvore de execução esperada, contendo construtores de objetos.

Unknown representa uma variável que pode tomar qualquer valor, sendo representada no modelo pelo caracter "–".

Randomize é uma classe auxiliar contendo os métodos implementados para gerar valores aleatórios, necessários à implementação de várias funcionalidades.

4.2.2.2 Utilização de Programação Orientada a Aspetos (AOP)

Programação Orientada a Aspetos (AOP) é um paradigma de programação que permite aumentar a modularidade do código, separando o código que implementa funções específicas e que afeta partes diferentes do sistema, as chamadas preocupações ortogonais (*crosscutting concern*) [Asp12]. Para por em prática os conceitos da Programação Orientada a Aspetos, foi utilizada a extensão orientada a aspetos da linguagem java, AspectJ. O conceito principal do AspectJ é o aspeto. Cada aspeto define uma função específica que pode afetar várias partes do sistema [Asp12]. São ainda utilizados *join points* que permitem intercetar pontos no fluxo de execução, como por exemplo, a chamada ou execução de métodos. Um *pointcut* é definido pela composição de vários *join points* e um *advice* representa um código adicional que deve ser executado aquando (antes ou depois) da interceção de um *join point*.

O AspectJ é utilizado pela biblioteca *TracingUtilities* com o objetivo de intercetar as execuções de métodos, podendo assim construir o traçado da execução do programa e compará-lo com o que seria previsto. Foram implementados 2 aspetos na biblioteca *TracingUtilities*, um

para o controlo da execução e um auxiliar para simulação de interação com o utilizador. O primeiro aspeto (`TraceCalls`) faz uso dos *wildcards* "*" para intercetar a execução de qualquer método com qualquer nome e retorno e um qualquer número de parâmetros, definido através do *join point* `execution(* *(..))`. Esta aspeto interceta ainda a execução de métodos não implementados (definidos como *«stub»* no modelo), fazendo-os retornar o valor esperado especificado. O segundo aspeto é utilizado para intercetar as interações do utilizador, através dos métodos `java.util.Scanner.nextLine(..)` e `java.io.PrintStream.println(..)`, para os métodos `enter` e `display`, respetivamente, da classe `Console`.

4.3 Concretização das principais funcionalidades

Seguidamente são apresentados os aspetos técnicos mais importantes da implementação das principais funcionalidades do FEUP-SBT-2.0. Entre elas estão a construção do módulo de configurações, representação em memória da estrutura do modelo, a execução e o respetivo *feedback* dos testes no Enterprise Architect e, finalmente a informação da cobertura dos testes.

4.3.1 Construção do módulo de configurações

O módulo de configurações tem como meio de persistência de dados um ficheiro XML, denominado `TGconfig .xml`, que é alojado no diretório onde se encontram as bibliotecas de vínculo dinâmico (dll) de instalação da ferramenta. Neste ficheiro são armazenadas todas as informações para construção da janela de configurações, bem como toda as opções nela definidas pelo utilizador. Um exemplo do conteúdo deste ficheiro pode ser observado no excerto de código 4.1. Até à linha 17 do excerto, são armazenados a etiqueta de apresentação na janela de configurações e o caminho guardado para cada uma das bibliotecas necessárias para o correto funcionamento do FEUP-SBT-2.0. A partir da linha 18 são guardadas informações de configuração por cada projeto do Enterprise Architect (EAP): localização do código fonte do respetivo modelo estrutural e ainda a indicação se o utilizador pretende utilizar sempre este diretório sem ser questionado.

```
1  <testgeneratorconfigurations>
     <junit>
3      <label>Path of JUnit jar file</label>
       <value>C:\junit4.10\junit-4.10.jar</value>
5    </junit>
     <aspectj>
7      <label>Path of AspectJrt jar file</label>
       <value>C:\aspectj1.6\lib\aspectjrt-1.6.12.jar</value>
9    </aspectj>
     <traceutils>
11     <label>Location of traceutils jar file</label>
       <value>C:\Users\mariofalcao\Desktop\traceutils.jar</value>
13   </traceutils>
     <javaagent>
```

```
15      <label>Location of javaagent jar file</label>
        <value>C:\aspectj1.6\lib\aspectjweaver.jar</value>
17    </javaagent>
    <files>
19      <file name="C:\workspace\Observer.eap">
          <path>C:\workspace\ob</path>
21        <check>true</check>
        </file>
23      <file name="C:\workspace\spreadsheetengine.eap">
          <path>C:\workspace\spread</path>
25        <check>true</check>
        </file>
27    </files>
</testgeneratorconfigurations>
```

Excerto de Código 4.1: Exemplo do conteúdo do ficheiro `TGconfig.xml`.

Toda esta informação é recolhida no início da execução da ferramenta FEUP-SBT-2.0 e também quando é lançado o módulo de configurações. Quando neste se registam alterações, procede-se à respetiva alteração do ficheiro. Este módulo apresenta ao utilizador uma janela (*Windows Form*), que contém todas as informações do ficheiro XML relevantes para o projeto a testar.

Não sendo um ficheiro oculto, a utilização direta do mesmo pelo utilizador é desencorajada, dado que a possível injeção dum erro de sintaxe XML no ficheiro poderia impossibilitar o funcionamento correto da ferramenta e todas as alterações pretendidas podem ser facilmente operadas diretamente na janela de configurações.

4.3.2 Representação em memória da estrutura do modelo

De modo a conferir uma maior robustez à criação dos modelos comportamentais foi decidido forçar o utilizador a utilizar uma estrutura de pacotes seguindo uma determinada ordem que mais à frente será explicada. Com esta decisão pretendeu-se forçar o utilizador a definir convenientemente no modelo comportamental os respetivos pacotes, classes e métodos de teste correspondentes no código de teste JUnit e assim evitar que inadvertidamente sejam introduzidos erros no modelo. Acredita-se que o tempo investido na construção de um modelo robusto é recompensado pelo tempo que se poderia perder ao gerar e executar código de teste a partir de um modelo com problemas, o que muitas vezes se podia tornar uma tarefa árdua e demorada de resolver.

A definição de uma estrutura bem elaborada permitiu ainda desenvolver a funcionalidade na qual é possível ao utilizador definir o pacote, classe ou método de teste que pretende executar, ao invés de ser executado sempre o modelo completo, podendo assim poupar tempo e focar a execução apenas no excerto do modelo que em cada situação mais lhe convier.

Apesar da existência da API do Enterprise Architect que possibilita a recolha de informação sobre o modelo e a sua estrutura, a forma de estruturação desta informação bem como o seu acesso tornaria o processo de edição do modelo mais complexo e, consequentemente, mais sujeito a falhas. Por esse motivo, para facilitar a implementação das novas funcionalidades e minimizar o

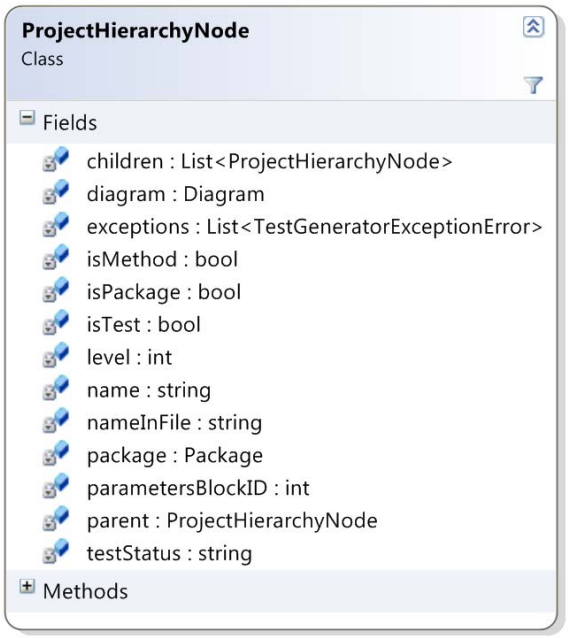


Figura 4.5: Estrutura *ProjectHierarchyNode* com os seus atributos.

número de chamadas à EA API, foi construída uma estrutura na qual é carregada a informação necessária da estrutura de pastas do modelo e referências para os respetivos diagramas de sequência.

Para carregamento da estrutura do modelo, foi assim criada de raiz a classe *ProjectHierarchyNode* que representa um nó da estrutura de pacotes referida, ou seja, cada um dos pacotes da estrutura. Os atributos desta classe estão inteiramente apresentados em 4.5, onde se destacam *parent* e *children* a que correspondem respetivamente, o nó pai e a lista de nós filho de cada nó. Através destes atributos, é possível ligar cada um dos nós numa estrutura em árvore, permitindo navegar em qualquer direção na estrutura. Existem também *flags* que indicam o tipo de pacote (Pacote, Classe ou Método de teste). Alguns atributos apenas são aplicáveis para alguns tipos de pacote, sendo exemplo disso, a lista de erros resultantes da execução que só é empregada nos nós do tipo Classe. É ainda guardada uma referência para a estrutura `Package` (Pacote) respetiva no Enterprise Architect e, para os nós do último nível, o `Diagram` (Diagrama de Sequência).

Imediatamente antes da execução do FEUP-SBT-2.0, é construída a árvore de nós *ProjectHierarchyNode* do modelo a testar, que analisa desde o pacote *Model* que representa o nó raiz, até todos os seus descendentes estarem incluídos. Após o preenchimento da árvore, é guardado o nó raiz da mesma, a partir do qual se pode percorrer a árvore por completo. Caso a árvore não contenha um mínimo de 3 níveis de nós, será considerada inválida, o utilizador será informado e a execução interrompida.

Atendendo a que a dimensão da árvore de nós é ilimitada, pois o utilizador pode definir um sem número de níveis de pacotes, as funções de navegação e procura na árvore foram implementadas com recurso à recursividade, sendo assim invocadas para o nó em questão e posteriormente para cada um dos filhos, até ser atingido o último nível da árvore.

Seleção do Excerto de Modelo a Testar

O utilizador define o excerto de modelo que pretende testar, ao invocar a execução do FEUP-SBT-2.0 numa determinada pasta, sendo todos os cenários que pertencem à descendência do pacote onde ocorreu a invocação, selecionados para ser executados. Este processo foi desenvolvido recorrendo à função `GetTreeSelectedPackage` do EA SDK [Spa10], que devolve o pacote selecionado pelo utilizador. De seguida, é localizado o pacote na árvore de nós *ProjectHierarchyNode*, e armazenada uma lista de todos os cenários de teste nele contidos.

4.3.3 Execução dos testes a partir do Enterprise Architect

O primeiro problema a solucionar no desenvolvimento da funcionalidade de execução automática dos testes, foi elaborar um mapeamento robusto entre os elementos dos diagramas de sequência UML (mensagens e fragmentos combinados) e o código gerado pelo *add-in*. Como todos os elementos que compõe um modelo UML do Enterprise Architect têm um identificador único, este tornou-se o elo de ligação entre modelo e código de teste. Os identificadores não se encontram explícitos diretamente através do Enterprise Architect, sendo alcançados através das funções da sua API.

Sendo reconhecidos pelo *add-in* três tipos de interações (com a API, com o utilizador e internas), a codificação pronta para execução de cada um dos tipos foi alterada de forma diferente, embora seguindo o mesmo método de inclusão do identificador do respetivo elemento no modelo UML. Nas interações com a API, que são modeladas através do método JUnit `assertEquals`, este foi reformulado para receber adicionalmente os identificadores das mensagens origem e de retorno, como é visível na tabela 4.1, com as alterações no código da versão 2.0 em relação à 1.0 aparecendo a negrito.

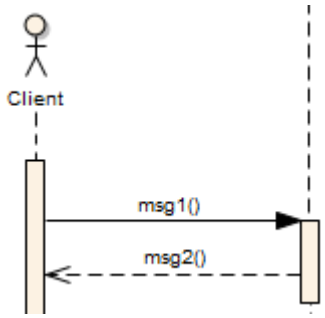
Modelo UML	
Codificação FEUP-SBT-1.0	`assertEquals(msg1(), msg2());`
Codificação FEUP-SBT-2.0	`assertEquals("`**`|DiagramMessageID=20|`**`",` **`"|RetID=30|"`**`, msg1(), msg2());`

Tabela 4.1: Codificação das interações com aplicação cliente no FEUP-SBT-1.0 e FEUP-SBT-2.0.

Relativamente às interações internas, codificadas através de estruturas definidas na biblioteca *TracingUtilities*, estas foram reformuladas para receber juntamente com todos os anteriores argumentos, o identificador do elemento respetivo no modelo. Por exemplo, conforme apresentado na tabela 4.2, para uma chamada básica, o primeiro e o penúltimo argumentos passaram a representar os identificadores de elementos. Por último, e a exemplo do anterior, a codificação das interações com o utilizador passou a incluir o identificador da mensagem respetivo, tal como presente na tabela 4.3. Uma vez mais, as diferenças na codificação entre versões são colocadas a negrito.

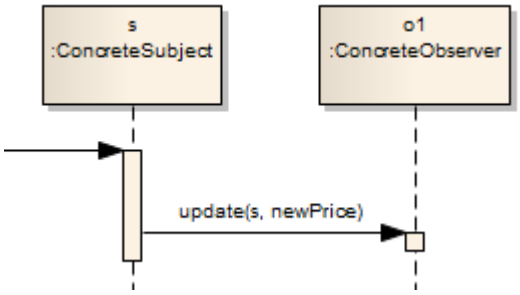
Modelo UML	
Codificação FEUP-SBT-1.0	`new Call("ConcreteObserver", o1, "update", new Object[] s, newPrice, null)`
Codificação FEUP-SBT-2.0	`new Call(7, "ConcreteObserver", o1, "update", new Object[] s, newPrice, -10, null)`

Tabela 4.2: Codificação das interações internas no FEUP-SBT-1.0 e FEUP-SBT-2.0.

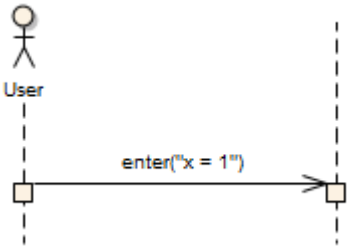
Modelo UML	
Codificação FEUP-SBT-1.0	`Console.enter("x = 1");`
Codificação FEUP-SBT-2.0	`Console.enter(150, "x = 1");`

Tabela 4.3: Codificação das interações com o utilizador no FEUP-SBT-1.0 e FEUP-SBT-2.0.

As alterações nas codificações apresentadas anteriormente, exigiram alterações nas estruturas respetivas na biblioteca *TracingUtilities* que as acompanhassem. Estas estruturas passaram agora a armazenar os identificadores de elementos correspondentes no modelo UML.

Mapeado código de teste gerado e o modelo UML, o passo seguinte passou por, aquando da deteção de um erro, incluir na sua mensagem o identificador do elemento do modelo que lhe correspondia. Encontrando-se já recolhidos todos os identificadores, este processo foi praticamente trivial.

Compilação e Execução do Código de Teste

Para efetuar a compilação e execução do código de teste gerado, foi desenvolvido um método executor de comandos `ExecuteCommand(object command, out string result)` que executa um comando passado por argumento (`command`), devolvendo verdadeiro ou falso, caso o comando tenha sido executado ou não, respetivamente, e ainda uma *string* contendo o resultado da execução (`result`).

A função `ExecuteCommand` utiliza a classe `Process`, do pacote `System.Diagnostics` que pertence à .NET Framework [Fra12b] para iniciar um processo do sistema local, executando através deste o comando pretendido e recolhendo o seu resultado. Atendendo à possibilidade do resultado ser apresentado tanto no `StandardOutput` em caso de execução sem erros, como no `StandardError` em caso de erros, haveria a possibilidade de ocorrência de um impasse (*deadlock*), que foi resolvida recorrendo à utilização de *handlers* `AutoResetEvent`, impondo uma execução não concorrente.

Os comandos utilizados para compilar e executar o código de teste são respetivamente o `javac` e o `java`, sendo efetuados de forma absolutamente transparente para o utilizador. Exemplos de instruções para compilação e execução do código de teste estão presentes em 4.2 e 4.3. É notória a indicação das bibliotecas imprescindíveis para o correto funcionamento do *add-in*: *TracingUtilities*, JUnit e AspectJ.

```
javac -cp "mainPath;<path>\junit-<version>.jar;<path>\aspectjrt-1.6.12.jar;<
    path>\traceutils.jar" "mainPath\package\*.java"
```

Excerto de Código 4.4: Exemplo de comando de compilação utilizado pelo FEUP-SBT-2.0

```
javac -cp "mainPath;<path>\junit-<version>.jar;<path>\aspectjrt-1.6.12.jar;<
    path>\traceutils.jar" -javaagent:"<path>\aspectjweaver.jar" org.junit.
    runner.JUnitCore "packageName.testName"
```

Excerto de Código 4.4: Exemplo de comando de execução utilizado pelo FEUP-SBT-2.0

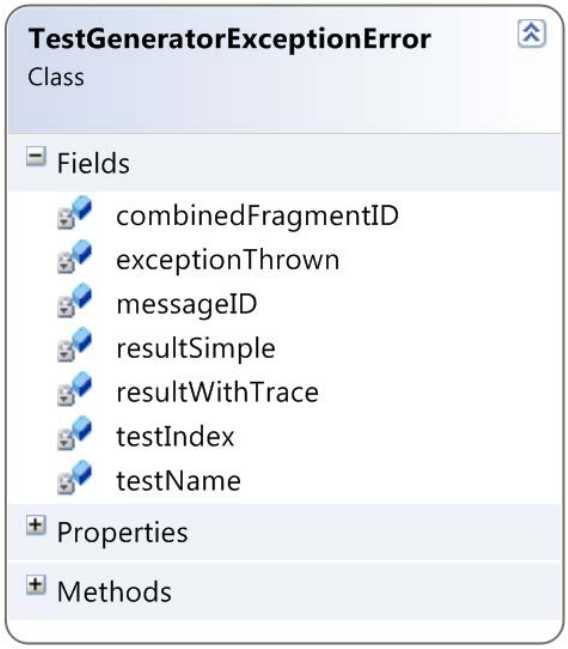


Figura 4.6: Estrutura *TestGeneratorExceptionError* com os seus atributos.

Interpretação do resultado de execução

Após execução do código de teste, toda a informação necessária para a edição do modelo UML no Enterprise Architect encontra-se sob forma textual no resultado da execução. Foi necessário construir um interpretador que percorre o resultado da execução para recolher toda esta informação. Atendendo à quantidade e às várias formas que a informação pode assumir, o interpretador construído foi uma das partes mais delicadas de desenvolver.

A informação recolhida pelo interpretador é agrupada numa estrutura construída exatamente para esse fim. O nome da estrutura é *TestGeneratorExceptionError* e os seus atributos encontram-se registados na figura 4.6. Nesta estrutura são definidos o tipo de erro apanhado (`exception Thrown`), os identificadores de elementos no Enterprise Architect onde ocorreu o erro (`message ID` e `combined Fragment ID`), o resultado da execução com e sem traçado (`resultWithTrace` e `resultSimple`) e, finalmente, o índice e nome do teste (`testIndex` e `testName`).

Uma lista de todos os erros ocorridos, organizados através da estrutura analisada anteriormente é recolhida e anexada ao nó da classe a que correspondem na árvore de nós *ProjectHierarchyNode*.

4.3.4 *Feedback* de sucesso/insucesso dos testes no Enterprise Architect

Com os possíveis erros detetados na execução do código de teste registados na árvore de nós *ProjectHierarchyNode*, o *add-in* encontra-se pronto para analisar a estrutura e editar o modelo em função dessa análise. Estas edições são realizadas percorrendo a árvore e editando cada pedaço desta de acordo com o seu resultado de execução.

Estereótipos

Uma das formas de edição do modelo UML é a atribuição de estereótipos aos pacotes do modelo, facilitando ao utilizador perceber que parte do modelo foi testada e qual o resultado dessa execução. É utilizado o atributo `StereotypeEx` de cada pacote (`Package`), que aloca, separado por vírgulas, todos os estereótipos do pacote em causa. Os estereótipos definidos pelo *add-in* não comprometem os anteriores definidos pelo utilizador, adicionando simplesmente a estes a etiqueta em função do resultado da execução e removendo-a apenas na altura de limpar o modelo para uma nova execução.

São utilizadas quatro etiquetas por convenção que serão associadas a um dado pacote: Falhou (*Failed*), Passou (*Passed*), Não Testado (*NotTested*) e Incompleto (*Incomplete*). A primeira etapa neste processo de edição é percorrer todos os nós do tipo Classe, ou seja, todas as classes e editar o estereótipos dos seus nós filhos (métodos de teste) como *Passed* ou *Failed* se esse método foi executado e passou ou falhou, respetivamente. Caso não tenha sido executado é colocada a etiqueta *NotTested*. Editados todos os nós do mais baixo nível da árvore, é iniciado um processo iterativo, que em cada iteração sobe um degrau na árvore e edita os estereótipos desse nível, em função dos filhos. Na primeira iteração é definido o estereótipo para os nós do tipo Classe, sendo atribuído *Passed* caso todos os cenários dessa classe tenham sido executados com resultado *Passed*, *Failed* caso algum dos cenários tenha falhado, *NotTested* se nenhum dos cenários foi executado e *Incomplete* se nem todos foram executados. De seguida, passa-se para o último nível dos nós do tipo Pacote e é repetido o mesmo processo, até atingir o nível mais alto da árvore, não entrando neste processo de edição os pacotes especiais *Model* e *Dynamic View*.

Edição de elementos dos diagramas de sequência

Quando é detetado que um teste contém erros resultantes da sua execução, o primeiro passo que é realizado pelo *add-in* é verificar quais dos seus cenários deram origem a esse erro, possível através da comparação do atributo *TestName* da estrutura de erro do *add-in* com o nome dos seus cenários. Uma vez concluída esta verificação, são analisados os identificadores (`MessageID` e `CombinedFragmentID`) definidos na estrutura. Os elementos a que correspondem estes elementos são coloridos a vermelho, através do seu atributo `Color`. É também adicionado às notas (`Notes`) desse elemento, o resultado simplificado, isto é, sem traçagem do erro. Uma vez mais, esta edição não compromete anteriores notas inseridas pelo utilizador. No final de cada alteração, refresca-se o elemento em questão, colocando em execução as alterações efetuadas.

Edição dos parâmetros de teste

Em cada cenário de teste que se conclua que foi executado, verificando se este consta na lista de cenário de teste a executar definida quando o utilizador invoca o *add-in*, é verificado se o cenário contém parâmetros de teste, verificando se o atributo `parametersBlockID` de cada nó tipo Método da árvore está atribuído.

Os cenários onde se verifique a existência de parâmetros de teste, são confrontados com o resultado da sua execução. Caso este não tenha produzido erros, todos os casos de teste são pintados de verde. Se existirem erros, é analisado o atributo `TestIndex` de cada estrutura representativa de um erro e pintado de vermelho o caso de teste a que corresponde.

4.3.5 Informação de cobertura dos testes

O algoritmo idealizado para a implementação desta funcionalidade adicional, acompanha a implementação do modo de execução básica, isto é, é colocada a informação necessária no resultado da execução. Posteriormente, este resultado é interpretado pelo *add-in*, que de seguida atua em função dessa análise.

Em relação à interpretação e edição gráfica do modelo, a adaptação dos métodos implementados anteriormente foi realizada sem problemas de grande relevo. Assim sendo, o verdadeiro desafio desta funcionalidade residiu em operar uma série de alterações tanto na parte de geração de código de teste, como na biblioteca *TracingUtilities*, por forma a conseguir detetar as mensagens efetivamente executadas e colocá-las no resultado da execução.

Em relação à geração de código, a implementação utilizada no modo de execução básico não permitia que as mensagens de API, ditas externas, fossem intercetadas, uma vez que a estrutura *Trace* da biblioteca *TracingUtilities* não tinha incluída na sua lista de chamadas esperadas, este conjunto de mensagens. Este problema foi solucionado, alterando a geração de código por forma a que todas as chamadas, tanto de interações internas, como acontecia anteriormente, como de interações externas, fossem codificadas através da estrutura *Trace*, ficando assim a biblioteca *TracingUtilities* capacitada para detetar a sua efetiva execução.

No que toca à biblioteca *TracingUtilities*, foi criado um contentor do tipo `HashSet` de valores inteiros, denominado *coveredMessages*, com a missão de armazenar todos os identificadores de mensagens que de facto foram executadas. Quando é confirmada a ocorrência de uma mensagem prevista pela estrutura *Trace*, o identificador dessa mensagem é automaticamente adicionado à lista *coveredMessages*.

Finalmente, no método `tearDown`, é impressa a lista *coveredMessages*, dentro de um determinado formato para ser legível pelo *add-in*. É importante ainda realçar, que este método de geração só é utilizado caso seja invocada este modo de operação, sendo mantido o modo de geração anterior para os restantes modos de operação.

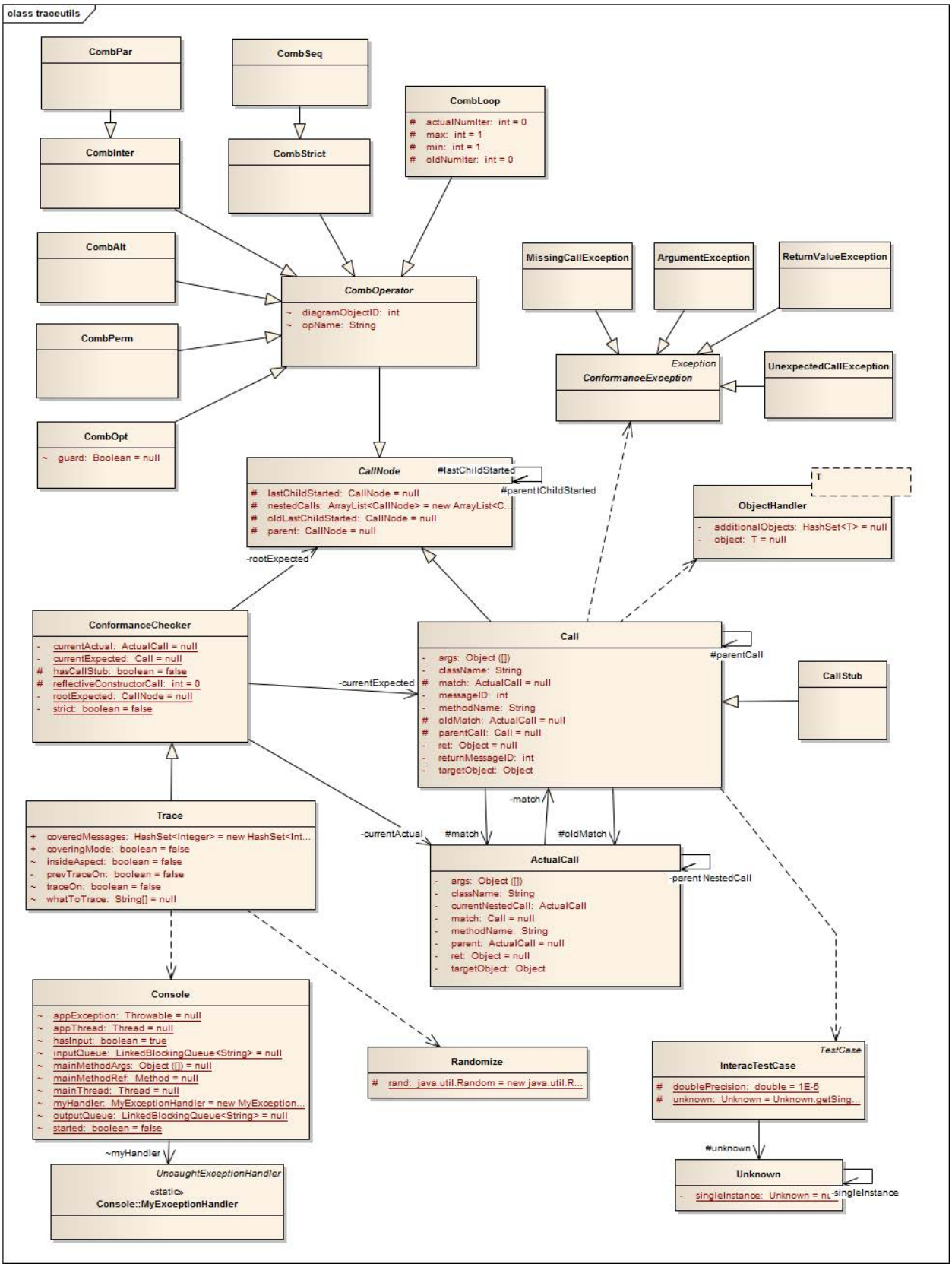


Figura 4.7: Diagrama de Classes da biblioteca *TracingUtilities*, omitindo operações.

Capítulo 5

Utilização da ferramenta

Apresenta-se neste capítulo uma descrição detalhada da utilização do protótipo FEUP-SBT-2.0. São indicadas todas as etapas necessárias para se proceder à instalação de todos os componentes necessários e respetivas configurações. Seguidamente, é descrito o processo de criação de um modelo considerado testável pela ferramenta e são analisados e comparados os seus diferentes modos de funcionamento. Por último, demonstra-se os meios utilizados para apresentar os resultados de execução ao utilizador.

5.1 Instalação

Para se proceder à instalação do *add-in* FEUP-SBT-2.0, que se encontra disponível para descarregar no endereço `https://feupload.fe.up.pt/get/l16y7nv72cBqeO0`, é necessário ter previamente instalado no sistema a ferramenta de modelação Enterprise Architect [Sys12], na versão 7.5 ou numa versão mais recente. Esta ferramenta de modelação apenas se encontra disponível para instalação direta em sistemas Microsoft® [Mic12] Windows 8, Windows 7, Windows Vista, Windows 2008, Windows 2003, Windows XP ou Windows 2000. É ainda necessário ter instalado no sistema a plataforma .NET [Fra12b] na qual vem incluída a Ferramenta de Registo de Assemblagem (Regasm.exe) [Net12] utilizada no processo de instalação.

Dependendo das funcionalidades do FEUP-SBT-2.0 que se pretende executar, existe um conjunto de programas que devem estar instalados no sistema para a sua correta execução. Por exemplo, se o utilizador apenas desejar gerar o código de testes JUnit a partir do modelo comportamental, não é necessário nenhum *software* adicional. Por outro lado, se o objetivo for executar o código de testes e verificar o seu resultado no diagrama, já é necessária a instalação de um conjunto de programas adicionais que permitem a compilação e execução do código de teste gerado pelo FEUP-SBT-2.0. Este conjunto de programas está assinalado na lista que se segue:

- Pacote de Desenvolvimento *Java JDK*
- *TracingUtilities*
- *JUnit*

- *AspectJ*

O Pacote de Desenvolvimento *Java JDK* é um ambiente desenvolvimento para construção de aplicações Java e está disponível em [Ora13]. Este ambiente de desenvolvimento exige uma configuração prévia para que todos os seus comandos possam ser utilizados pelo sistema, configuração que mais à frente será detalhadamente descrita. Em relação à biblioteca *TracingUtilities*, esta é disponibilizada juntamente com o *add-in* e a sua função é possibilitar a verificação da execução das interações internas do código de teste gerado conforme o previsto pelo utilizador. A *framework JUnit*, que possibilita a criação e execução de testes unitários, está disponível em [JUn12]. Deve ser utilizada uma versão igual ou posterior à "4.10" e confirmada a presença da biblioteca *junit-<version>.jar* na pasta de instalação. Por último, a extensão *AspectJ* é utilizada para intercetar as chamadas efetuadas durante a execução do código de teste, de modo a verificar que todas as chamadas especificadas foram de facto efetuadas. A extensão está disponível em [Asp12], onde deve ser descarregada e descomprimida a versão "1.6.12". Deve ser confirmada a presença dos ficheiros *aspectjrt-1.6.12.jar* e *aspectjweaver.jar*.

São de seguida apresentados uma lista resumida dos passos necessários para se proceder à instalação do *add-in* para Enterprise Architect. No anexo A encontra-se de forma mais detalhada, todos os passos para instalação e configuração do *add-in*.

1. Colocar as bibliotecas dll num diretório do sistema.
2. Registar das bibliotecas no sistema através da Ferramenta de Registo de Assemblagem (Regasm.exe).
3. Adicionar a chave respetiva no Editor de Registos (regedit).
4. Confirmar que o Enterprise Architect deteta a presença do *add-in*.

5.2 Configuração

Na presente secção é apresentado o módulo de configurações do FEUP-SBT-2.0 que é composto por um conjunto de informações necessárias para o funcionamento pleno do *add-in* e informações adicionais para facilitar a utilização do mesmo. De seguida é demonstrado como podem ser definidas todas as opções no menu de configuração.

O menu de configuração pode ser invocado através das opções de menu do *add-in*. Tal como apresentado em 5.1, basta clicar com o botão direito do rato numa qualquer pasta do Explorador de Projeto (*Project Browser*) clicando em "*Add-In*", seguido de "*Test Generator*" e por último "*Preferences*". Este menu pode ser automaticamente aberto caso seja detetado que alguma informação necessária não está ainda definida.

Apresentado numa única janela, o menu de configuração encontra-se dividido em 2 camadas de configuração: Configurações Gerais (*General Preferences*) e Configurações do Projeto Atual (*Current Project Preferences*). Tal como o título sugere, as Configurações Gerais são responsáveis

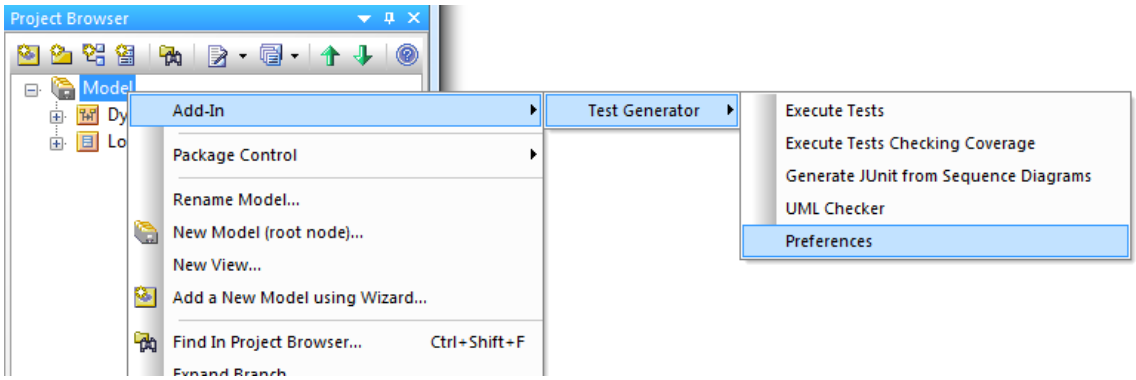


Figura 5.1: Meio de acesso ao Menu de Configuração.

pelas configurações do *add-in* que são comuns a todos os projetos enquanto que as Configurações do Projeto Atual são relativas às configurações do projeto de Enterprise Architect em execução. O aspeto do Menu de Configuração pode ser visualizado em 5.2.

Configurações gerais

Nas Configurações Gerais do Menu de Configuração é requerido ao utilizador indicar a localização no sistema de uma série de ficheiros JAR relativos ao *software* adicional necessário registado anteriormente. É assim indispensável definir o caminho para o ficheiro JAR do JUnit (*junit-<version>.jar*), para o ficheiro jar do AspectJ (*aspectjrt-1.6.12.jar*), para o ficheiro *weaver* do AspectJ (*aspectjweaver.jar*) e ainda a biblioteca adicional *Tracing Utilities* (*traceutils.jar*), disponibilizada juntamente com o *add-in*. Todos estes ficheiros são essenciais para a execução direta do código de testes.

Configurações do projeto atual

As configurações relativas ao projeto em execução têm por objetivo aumentar a usabilidade da utilização do *add-in*. Nestas configurações é possível definir a localização do código fonte do modelo estrutural, necessário para a compilação e execução do código de testes. Existe ainda a opção de utilizar sempre a localização definida sem voltar a questionar o utilizador. Assim sendo, caso esta opção esteja desativada, cada vez que é invocada a geração ou execução dos testes, é aberta uma janela para seleção manual da localização do código fonte cuja primeira localização é a definida no menu.

5.3 Criação de um modelo testável

Com o objetivo de aumentar a robustez dos modelos UML, é exigido ao utilizador inserir os diagramas de sequência numa estrutura de pacotes estipulada. Esta estrutura necessária para a

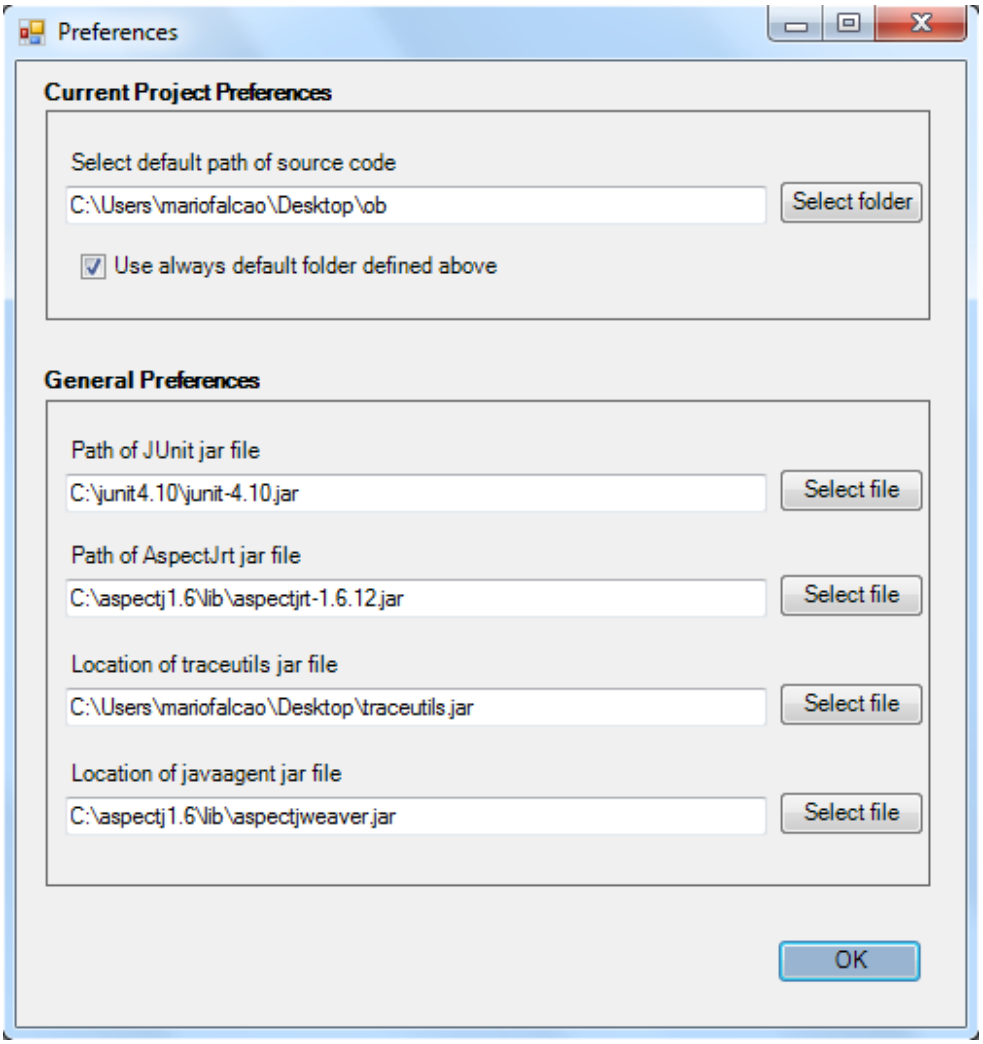


Figura 5.2: Menu de Configuração.

criação de um modelo testável é de fácil compreensão e exige um mínimo de 3 camadas de pacotes, dentro do pacote principal *Dynamic View* inserido automaticamente pelo Enterprise Architect aquando da criação de um modelo comportamental. Estas 3 pastas simbolizam respetivamente o pacote (*package*), a classe e o método de teste, sendo que para representar o pacote de testes, pode ser utilizado um número qualquer de níveis de pastas, permitindo assim definir um pacote mais complexo.

Para uma mais fácil perceção desta estrutura de pastas, é apresentado em 5.3 um modelo exemplificativo. No caso ilustrado, é visível a presença de 3 camadas para representar o pacote de testes do modelo, começando em "Pacote nível 1", seguido de "Pacote nível 2" e por último "Pacote nível N". Ao gerar código de testes JUnit a partir deste modelo seria criado no diretório raiz definido pelo utilizador uma pasta com o nome do primeiro nível do pacote, contendo a sua semelhante do segundo nível que por sua vez ia conter a do terceiro nível. Cada uma das classes de teste pertencentes a este pacote iriam ter expresso no seu código fonte a referência a que

pertenciam ao pacote (*package*) "Pacote nível 1"."Pacote nível 2"."Pacote nível N". Analisando de seguida as classes de teste, estas são sempre representadas pelas pastas do penúltimo nível da estrutura. Verifica-se assim que dentro do pacote complexo estão presentes duas classes de teste: "Classe de teste 1" e "Classe de teste 2" e que a primeira contém 2 métodos de teste ao passo que a segunda 3 métodos. Os métodos de teste são representados pelo último nível da estrutura de pastas e devem conter no seu interior o diagrama de sequência que posteriormente será convertido em código de teste.

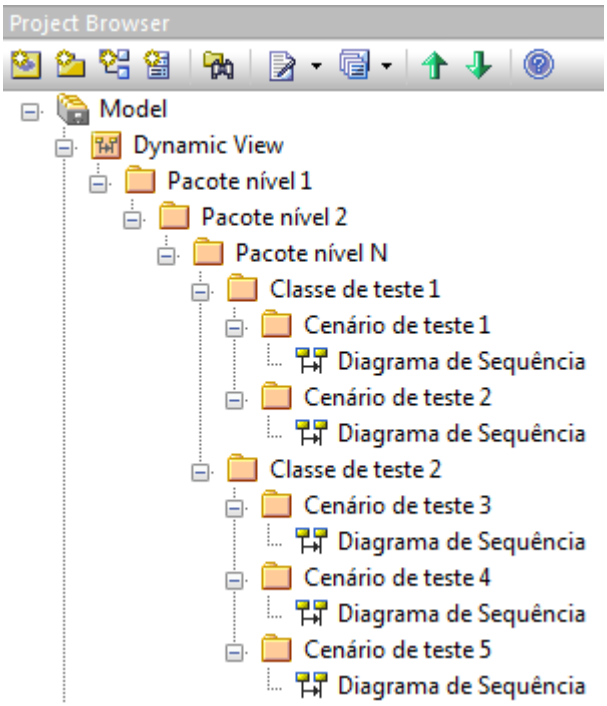


Figura 5.3: Estrutura de pastas de um modelo exemplificativo correto.

Caso a estrutura de pastas do modelo comportamental não siga as regras previamente definidas, como por exemplo acontece em 5.4 onde apenas estão presentes 2 níveis de pastas, é apresentada a mensagem de erro ilustrada em 5.5 e interrompida a geração ou execução do modelo.

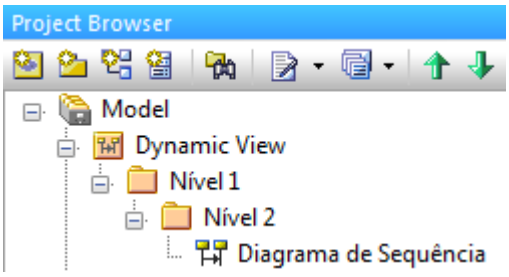


Figura 5.4: Estrutura de pastas de um modelo exemplificativo não correto.

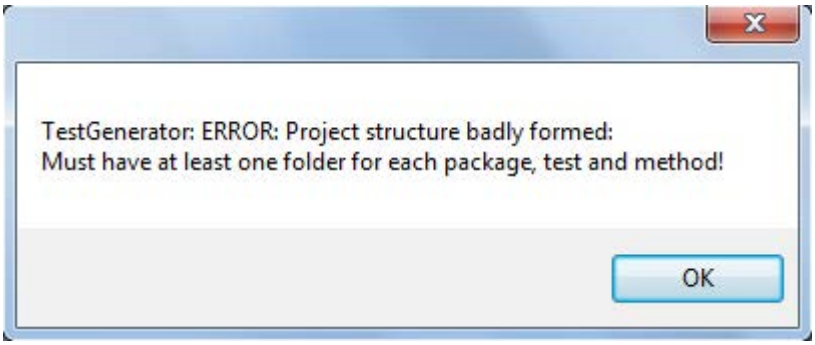


Figura 5.5: Mensagem de erro alertando má formação do modelo.

5.4 Invocação do gerador e executor de testes

Nesta secção é descrito o processo de invocação do gerador e executor de testes do FEUP-SBT-2.0. Inicialmente é apresentado como é possível ao utilizador escolher um fragmento do modelo para testar e posteriormente são apresentadas os quatro modos de operação, analisando as diferenças entre si.

5.4.1 Seleção de excerto do modelo a testar

Após a criação de um modelo bem definido e consequentemente testável, este encontra-se pronto para ser operado pelo FEUP-SBT-2.0. Mas tal como afirmado anteriormente, é permitido ao utilizador escolher que excerto do modelo deve ser considerado. Esta seleção é realizada em função da pasta selecionada no ato de invocação do *add-in*. Assim sendo, se o FEUP-SBT-2.0 for invocado a partir das pastas base *Model* ou *Dynamic View*, todo o modelo será executado. No entanto, se for selecionada uma pasta de um determinado pacote, apenas as classes e respetivos métodos de teste desse pacote serão executadas, sendo ignoradas todas as restantes. O mesmo acontece quando selecionada uma pasta representando uma classe de teste, executando nesse caso apenas essa classe com todos os seus métodos de teste. Pode ainda ser selecionado um método apenas a testar, sendo os restantes métodos da mesma classe de teste ignorados. Na figura 5.18 encontram-se alguns exemplos de possíveis seleções efetuadas pelo utilizador e o que essas seleções representariam em termos de classes e métodos de teste analisados. Encontram-se assinalados na imagem a azul, a seleção do utilizador, enquanto que a verde escuro, as classes analisadas e finalmente a verde claro, os cenários analisados.

5.4.2 Seleção do modo de operação

Existem quatro funcionalidades que podem ser invocadas pelo utilizador e acedidas através das opções de menu, como mostra a figura 5.6, onde a vermelho aparecem as mesmas assinaladas. São estas: Executar os Testes (*Execute Tests*), Executar os Testes Analisando Cobertura (*Execute Tests Checking Coverage*), Gerar JUnit a partir dos Diagramas de Sequência (*Generate JUnit from*

Sequence Diagrams) e, por último, o Analisador de UML (*UML Checker*). A quinta opção é o Menu de Configuração que já foi abordado previamente na Secção 5.2.

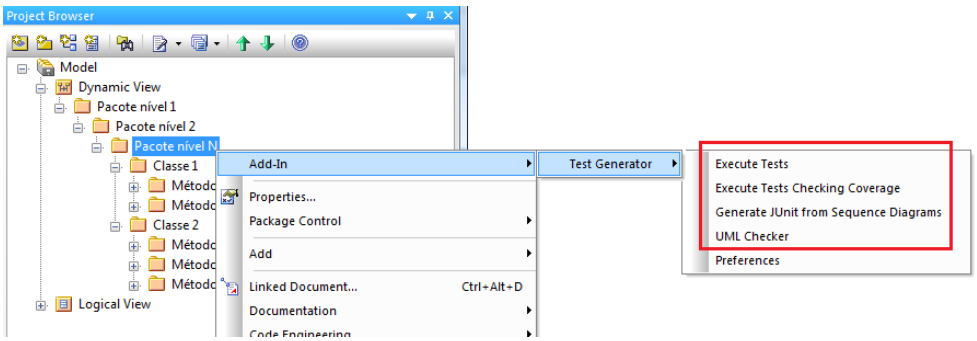


Figura 5.6: Menu para escolha da funcionalidade pretendida.

Das quatro funcionalidades, é possível agrupar as três primeiras num grupo de geração e execução de testes, sendo a quarta, *UML Checker*, mais de controlo. Nesta última funcionalidade não existiu intervenção no âmbito desta dissertação, pelo que não será analisada tão detalhadamente quanto as restantes. No entanto, o seu funcionamento é explicado em [FPY12]. O *UML Checker* permite ao utilizador detetar possíveis inconsistências e incompletudes entre o modelo estrutural e comportamental. Através desta funcionalidade, o utilizador é alertado para inconsistências como: métodos desconhecidos, número de parâmetros incorreto, tipos de parâmetros ou retorno errados, violações de acessibilidade, entre outras. O utilizador pode ainda conferir se todos os métodos e construtores definidos no modelo estrutural são exercitados no modelo comportamental.

Em relação às três primeiras funcionalidades para geração e execução de testes, a primeira ação que ocorre quando uma delas é selecionada é verificar se existe necessidade do utilizador escolher a pasta de referência. Esta pasta refere-se ao local onde são guardados os testes JUnit gerados e, em caso de execução dos testes, é também a localização do código fonte a testar. Esta pasta pode ser selecionada através da abertura de uma Janela para Seleção de Pastas (*Windows Form* [Net13]). Esta janela aparecerá se não houver nenhum diretório pré-definido nas configurações ou caso o utilizador não tenha selecionado a opção que permite utilizar sempre o diretório pré-definido também nas configurações. De seguida são apresentadas separadamente cada uma destas três funcionalidades do FEUP-SBT-2.0, não necessariamente pela ordem em que ocorrem no menu de seleção.

Gerar JUnit a partir dos diagramas de sequência

Este é o modo de operação básico do *add-in* já presente na sua versão 1.0, no qual é gerado um código de teste em JUnit a partir dos diagramas de sequência UML modelados no Enterprise Architect. Ao selecionar esta funcionalidade, será criado, caso não exista ainda, a estrutura de pastas que representa a estrutura definida no modelo UML, tal como explicado na Secção 5.3. Esta estrutura, no seu caso mais básico, pode ser apenas uma pasta com o mesmo nome do único pacote de primeiro nível definido no modelo comportamental UML e será criada dentro da pasta

de referência. Dentro das pastas a que pertencem, serão criadas as classes Java onde estarão codificados os cenários de teste modelados. Para uma melhor compreensão deste processo, se se invocasse este modo de operação no modelo definido em 5.3, seria criado na pasta referência um diretório com o nome "Pacote nível 1", no seu interior o "Pacote nível 2" e dentro deste o diretório com o nome "Pacote nível N", representando assim o primeiro nível de pacotes do modelo. No último diretório ("Pacote nível N"), seriam criados 2 ficheiros Java: "Classe de teste 1.java" e "Classe de teste 2.java", de acordo com o segundo nível de pacotes. Na "Classe de teste 1.java" estariam implementados 2 métodos representando os "Cenário de teste 1" e "Cenário de teste 2" da "Classe de teste 1", ao passo que o ficheiro "Classe de teste 2.java" contaria com 3 métodos representando os cenários de teste definidos dentro de "Classe de teste 2", ou seja, "Cenário de teste 3", "Cenário de teste 4" e "Cenário de teste 5". As especificidades do código de teste gerado encontram-se apresentadas no anexo B. Após a construção da estrutura de pastas e código de testes, esta funcionalidade encerra o seu funcionamento, ficando ao encargo do utilizador executar os testes e analisar o seu resultado.

Executar testes

O modo de operação "Executar Testes" tem por objetivo automatizar não apenas a geração como acontece no FEUP-SBT-1.0, mas também a compilação, execução e apresentação de resultados. Esta operação pode ser dividida em 3 etapas: "Geração de Testes", "Compilação/Execução de Testes" e "Edição do modelo UML de acordo com os resultados". Na primeira etapa, "Geração de Testes", tal como o nome indica, são realizadas as ações indicadas na funcionalidade anterior, ou seja, geração do código de teste JUnit e alocação do mesmo na estrutura de pastas respetiva. De seguida, é efetuada a "Compilação/Execução de Testes", de forma automática e transparente para o utilizador, sendo guardado o resultado da mesma execução, que contém as informações necessárias para a edição do modelo. Por último, é interpretado o resultado da execução e, com base nessa análise, é editado o modelo de várias formas, com o objetivo de disponibilizar ao utilizador um leque variado de informação, que integrado possa ser o mais completo para analisar os resultados da execução.

Executar os testes analisando cobertura

No seguimento do desenvolvimento da funcionalidade anterior, concluiu-se que para além de constatar e exibir os erros encontrados na execução do código de teste JUnit gerado, seria útil ao utilizador poder verificar graficamente no modelo UML as chamadas que foram, de facto, executadas, permitindo analisar diretamente no modelo a cobertura da execução. Trata-se de um critério de análise extremamente útil tanto para determinar quando é que o teste pode ser finalizado, isto é, quando os cenários de teste cobrem todas as mensagens do modelo, mas também para medir e classificar a eficácia do processo de teste [ZHM97].

Desta forma, foi adicionada informação ao resultado da execução do código de teste JUnit gerado, que posteriormente é interpretada permitindo colorir as mensagens no modelo UML que

foram executadas, para além de todas as informações já disponibilizadas no modo básico de execução de testes, descrito anteriormente.

5.5 Interpretação de resultados de execução

Nesta secção são apresentados os diversos meios utilizados para fornecer ao utilizador informação gráfica e textual sobre o resultado da execução operada. Para facilitar e acelerar a compreensão destes aspetos, é utilizado um exemplo denominado "*ATM*", contendo apenas um simples diagrama de sequência ilustrado em 5.7, no qual é simplificado um processo de levantamento monetário num sistema bancário, construído com o propósito de abranger praticamente todas as opções de apresentação de resultados de execução disponibilizadas pelo FEUP-SBT-2.0. Neste exemplo, existem duas classes `Account` e `Movement` representando, respetivamente, uma conta e um registo de movimento bancários. Faz ainda parte do modelo um conjunto de parâmetros editáveis pelo utilizador como o saldo da conta, quantidade de dinheiro a levantar, indicação se se deve registar o movimento ou não, entre outras, por forma a possibilitar a criação dos mais variados casos de teste para o mesmo cenário de teste. A resposta ao processo de levantamento é simplificada pelo retorno de palavras-chave "*OK*" ou "*FAIL*".

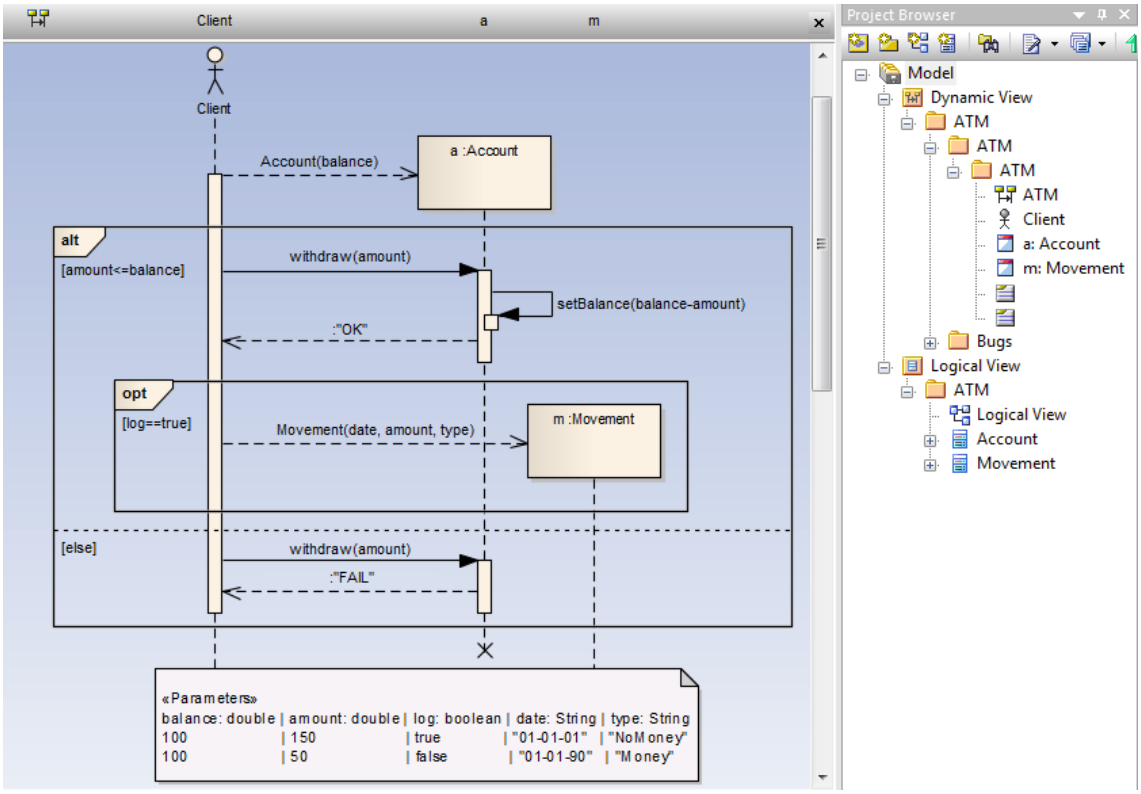


Figura 5.7: Exemplo ATM - Diagrama de sequência exemplificativo de um levantamento monetário num sistema bancário.

Mensagem de erro de compilação

O primeiro resultado que o utilizador pode receber do FEUP-SBT-2.0 acontece caso a compilação do código de teste gerado tenha produzido erros. É nesse caso apresentada uma mensagem contendo a descrição dos erros detetados pelo compilador, bem como o pacote em que se inserem, para mais facilmente ser detetada a localização do erro. No exemplo "*ATM*" foi propositadamente injetado um erro e em 5.8 é apresentada a mensagem de erro respetiva. Após confirmar esta mensagem, clicando em "*OK*", o FEUP-SBT-2.0 termina o seu funcionamento, aguardando que o utilizador corrija os erros detetados e o volte a executar.

Figura 5.8: Mensagem apresentando erros de compilação.

Mensagem com resultado de execução

Caso não ocorram erros na compilação, o FEUP-SBT-2.0 irá executar os testes anteriormente gerados. No fim deste processo, será apresentado ao utilizador uma mensagem com o resultado completo da execução, quer este contenha erros ou não. Ilustrado em 5.9 encontra-se a mensagem com o resultado da execução do exemplo "*ATM*" no seu estado original. Para demonstrar o comportamento numa situação com erros de execução, foi injetado um erro na palavra de retorno "*FAIL*", substituindo-a por "*ErroInjetado*". A mensagem apresentada após execução do modelo com este erro, encontra-se presente em 5.10.

Ao confirmar a visualização da mensagem com o resultado da execução, o utilizador depara-se com o modelo já contendo todas as alterações operadas pelo FEUP-SBT-2.0. De seguida serão descritas cada uma dessas edições do modelo tanto para o exemplo anterior e como para outros casos não cobertos pelo mesmo.

Estereótipos

Começando pelo Explorador de Projetos (*Project Browser*), em cada um dos pacotes definidos no já abordado sistema de 3 níveis de pacotes, é introduzida uma etiqueta condizente com o resultado da execução. Esta etiqueta é introduzida como estereótipo (*stereotype*) dos pacotes do diagrama de sequência, sem prejuízo para anteriores estereótipos que os utilizadores possam ter definido, atendendo a que um pacote pode estar associado a vários estereótipos. Visualmente, o

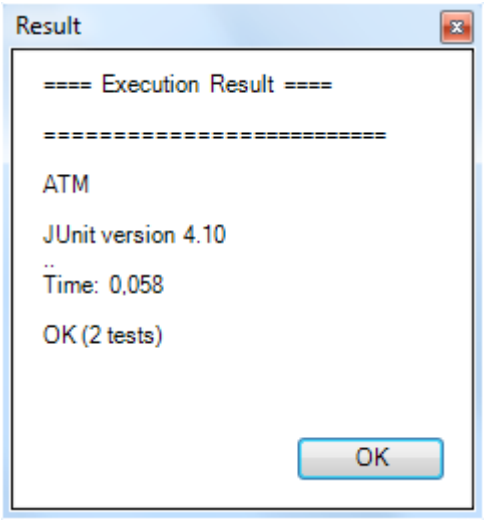


Figura 5.9: Mensagem após execução sem erros.

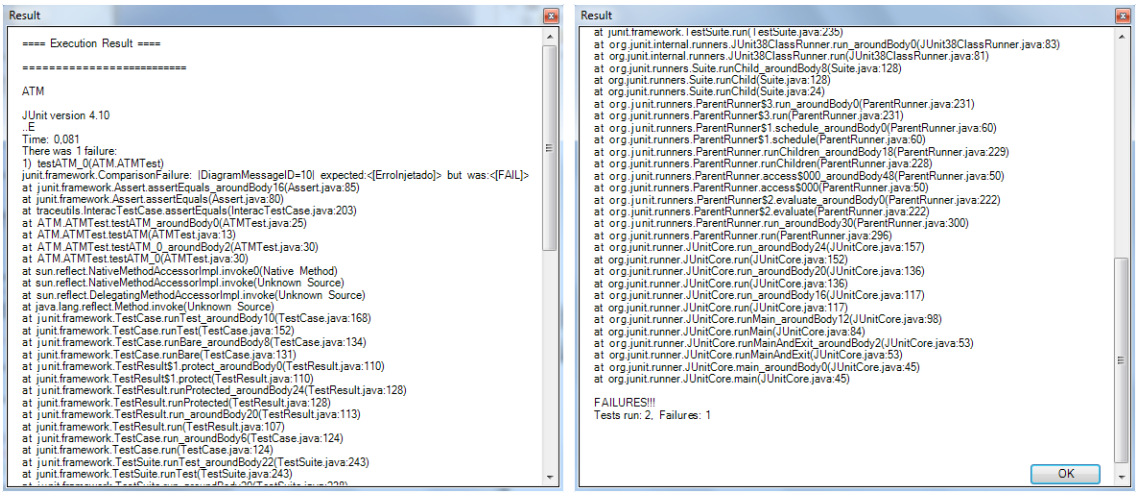


Figura 5.10: Mensagem após execução com erros. Na imagem à direita encontra-se a mesma mensagem da imagem à esquerda, percorrida até baixo.

estereótipo é apresentado entre aspas angulares («...»), imediatamente antes do nome do pacote respetivo.

Existem quatro etiquetas utilizadas pelo *add-in* para classificar cada um dos pacotes: "*Passed*", "*Failed*", "*NotTested*"e "*Incomplete*". Cada um dos pacotes do nível mais baixo, representando cenários de teste, são classificados como "*Failed*"ou "*Passed*"caso tenham sido executado com ou sem erros, respetivamente, ou como "*NotTested*"caso não tenham sido selecionados para serem executados. Já os pacotes do segundo nível, que representam classes de teste, são classificados como "*Failed*"se pelo menos um dos seus cenários de teste tiver falhado, "*Passed*"se nenhum cenário tiver falhado, "*NotTested*"se nenhum dos seus cenários tiver sido executado e, finalmente, "*Incomplete*"caso nem todos os seus cenários tenham sido selecionados. Este processo de defini-

ção dos estereótipos para o nível das classes de teste em função dos seus cenários é replicado para o nível dos pacotes de teste, em função das suas classes. Em 5.11 é visível o aspeto do Explorador de Projetos do modelo do exemplo "*ATM*" após a sua execução. Na figura à esquerda, encontra-se o resultado no exemplo sem erros e à direita com a injeção de erros. Encontram-se assinaladas a verde a etiqueta "*Passed*", a amarelo as etiquetas *"Not Tested"* e *"Incomplete"* e a vermelho a etiqueta *"Failed"*.

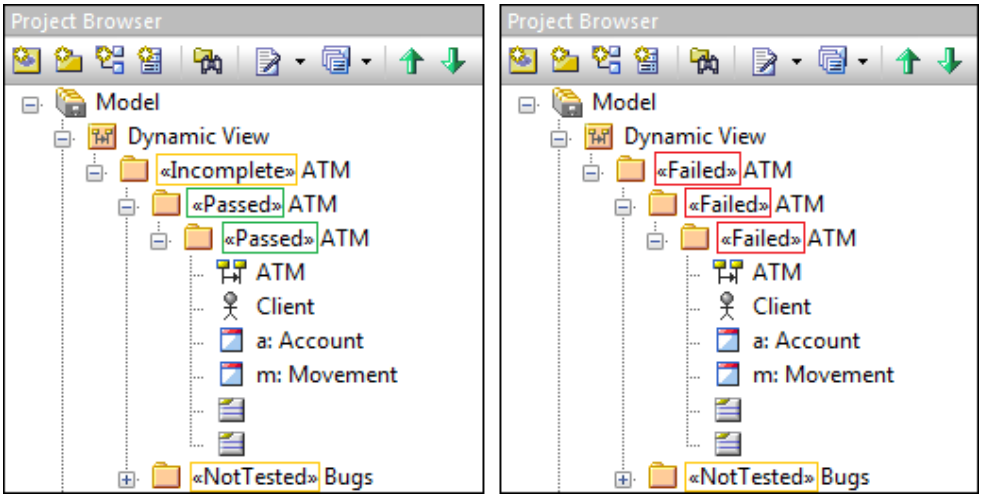


Figura 5.11: Explorador de Projetos com os estereótipos dos pacotes editados pelo FEUP-SBT-2.0.

Coloração de mensagens ou fragmentos combinados

Outra edição gráfica operada no modelo é a coloração a vermelho das mensagens onde ocorreram erros de execução. No exemplo "*ATM*" a mensagem de retorno com erro injetado será pintada de vermelho, conforme é visível em 5.12.

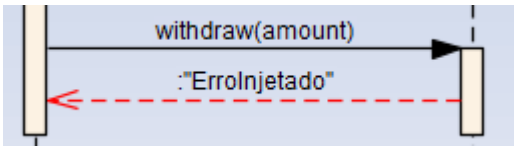


Figura 5.12: Mensagem pintada de vermelho assinalando erro de execução.

Todos os erros resultantes da execução do código de teste gerado dão origem à coloração da mensagem específica onde ocorreu o erro, seja esta a própria chamada ou a chamada de retorno, exceto em dois casos mais específicos. O primeiro ocorre quando o erro encontrado é do tipo "*UnexpectedCallException*" que acontece quando em modo estrito ("*strict*") é detetada uma chamada que não está prevista no diagrama de sequência. Tratando-se de uma chamada que não existe no modelo, o que impossibilita a sua coloração, a mensagem editada é o pai que lhe deu origem, ou seja, que iniciou a sequência de interações internas. É apresentado em 5.13, como

seria pintado o modelo se ocorresse qualquer uma outra chamada para além da indicada como "`msgFilho()`", ficando o erro assinalado em "`msgPai()`", que lhe deu origem. O segundo caso especial pode ocorrer caso o modelo conte com um fragmento combinado do tipo "alternativa" ("*alt*") e se não forem executadas nenhuma das alternativas definidas, é o próprio fragmento combinado que é editado. Um exemplo desta última situação é apresentado em 5.14. No exemplo, como tanto "`msg1(arg1,arg2)`" como "`msg2(arg3,arg3)`" não foram executadas, é o fragmento combinado "*alt*" em que estas se inserem que é editado.

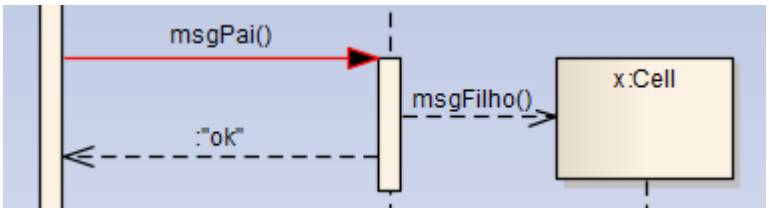


Figura 5.13: Coloração do modelo aquando de "*UnexpectedCallException*".

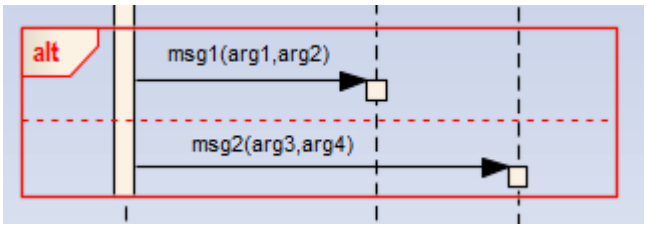


Figura 5.14: Coloração do modelo caso todas as alternativas de um fragmento combinado "*Alt*" não forem executadas.

Edição das notas de mensagens ou fragmentos combinados

Para além da edição da cor das mensagens com erro, explicado nos parágrafos anteriores, é ainda adicionado às Notas (*Notes*) de cada uma dessas mensagens, o resumo do erro respetivo, sem prejuízo de notas introduzidas diretamente pelo utilizador. O resumo do erro é efetuado retirando todo o traço de execução que é incluído no resultado da execução, mantendo-se apenas o cabeçalho que sucintamente indica o erro ocorrido. No exemplo "ATM" com erro, a mensagem de retorno pintada a vermelho em 5.12 teria incluída nas suas notas o erro respetivo, tal como ilustrado em 5.15. A edição das notas acompanha a coloração das notas no que toca também aos casos especiais de exceções "*UnexpectedCallException*" e fragmentos combinados "*Alt*.

Coloração dos parâmetros de teste

Esta opção só acontece caso o diagrama de sequência contenha um bloco de Parâmetros ("*Parameters*"), tratando-se assim de um cenário de teste parametrizado, onde é possível verificar o comportamento do mesmo cenário para diferentes combinações de valores de parâmetros (casos

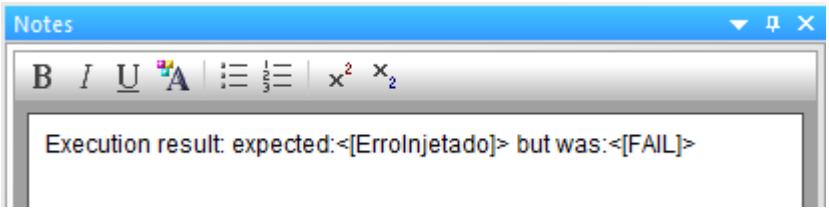


Figura 5.15: Edição das notas da mensagem com erro do exemplo "*ATM*".

de teste). Como é possível verificar na base da figura 5.7 encontram-se definidos 2 casos de teste que serão executados separadamente. Após a execução do código de teste gerado, cada linha do bloco referido será pintada de vermelho ou verde conforme a sua execução tenha produzido ou não erros, respetivamente. No exemplo "ATM" com erro, como é visível em 5.16, o primeiro cenário encontra-se a vermelho porque falhou, dado que interagia com o trecho de código onde foi injetado o erro, ao passo que, por não interagir com essa parte, o segundo cenário foi executado sem qualquer erro e portanto encontra-se pintado de verde.

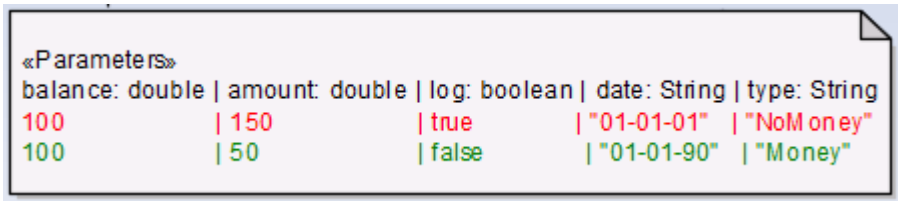


Figura 5.16: Bloco de parâmetros pintado conforme resultado da execução.

Coloração do diagrama de acordo com a análise de cobertura

Se o utilizador selecionar o modo de operação "Executar Testes Analisando Cobertura", todas as edições ao modelo descritas anteriormente nesta secção são igualmente aplicadas. Para além destas, é efetuada uma coloração a verde de todas as mensagens que o *add-in* tenha concluído que de facto foram executadas. As mensagens que não foram executadas mantém-se na cor pré-definida (preto) e as mensagens que contêm erros são pintadas de vermelho, tal como anteriormente.

Na figura 5.17, é apresentado o estado final do modelo UML do exemplo ATM com injeção de erros, após ser selecionado e executado o modo de operação "Executar Testes Analisando Cobertura", podendo assim ser verificadas as alterações descritas anteriormente.

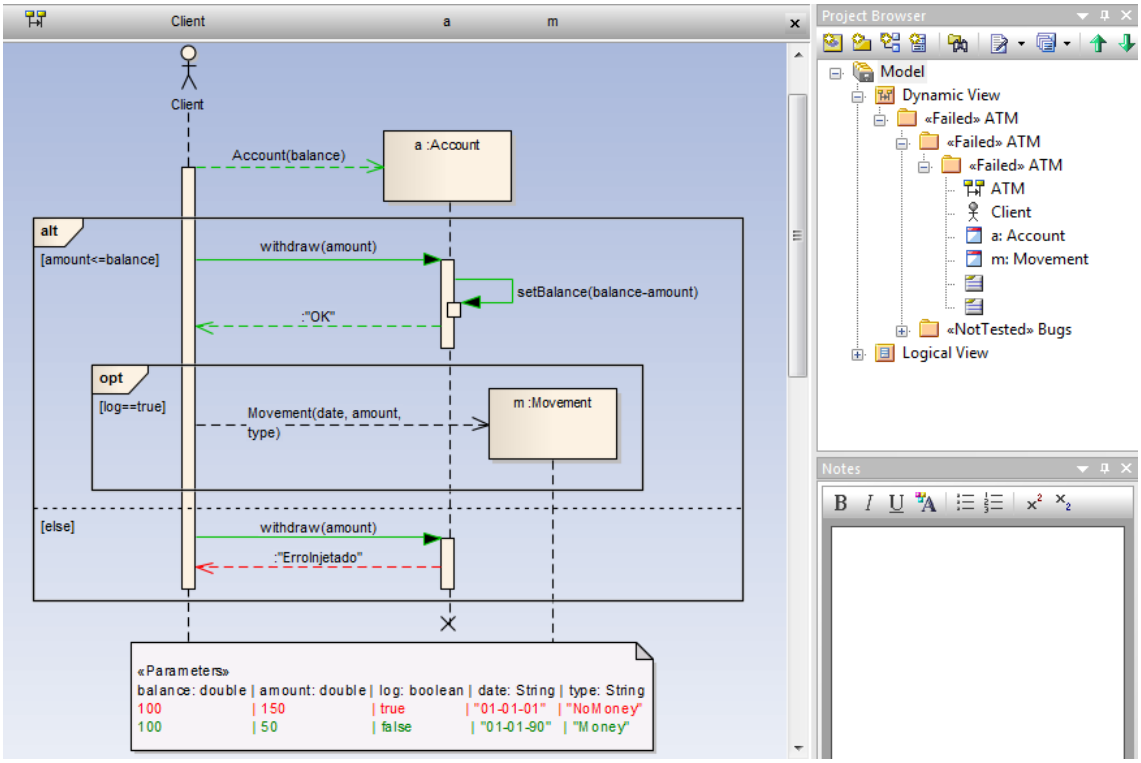


Figura 5.17: Exemplo ATM com injeção de erros após execução analisando cobertura.

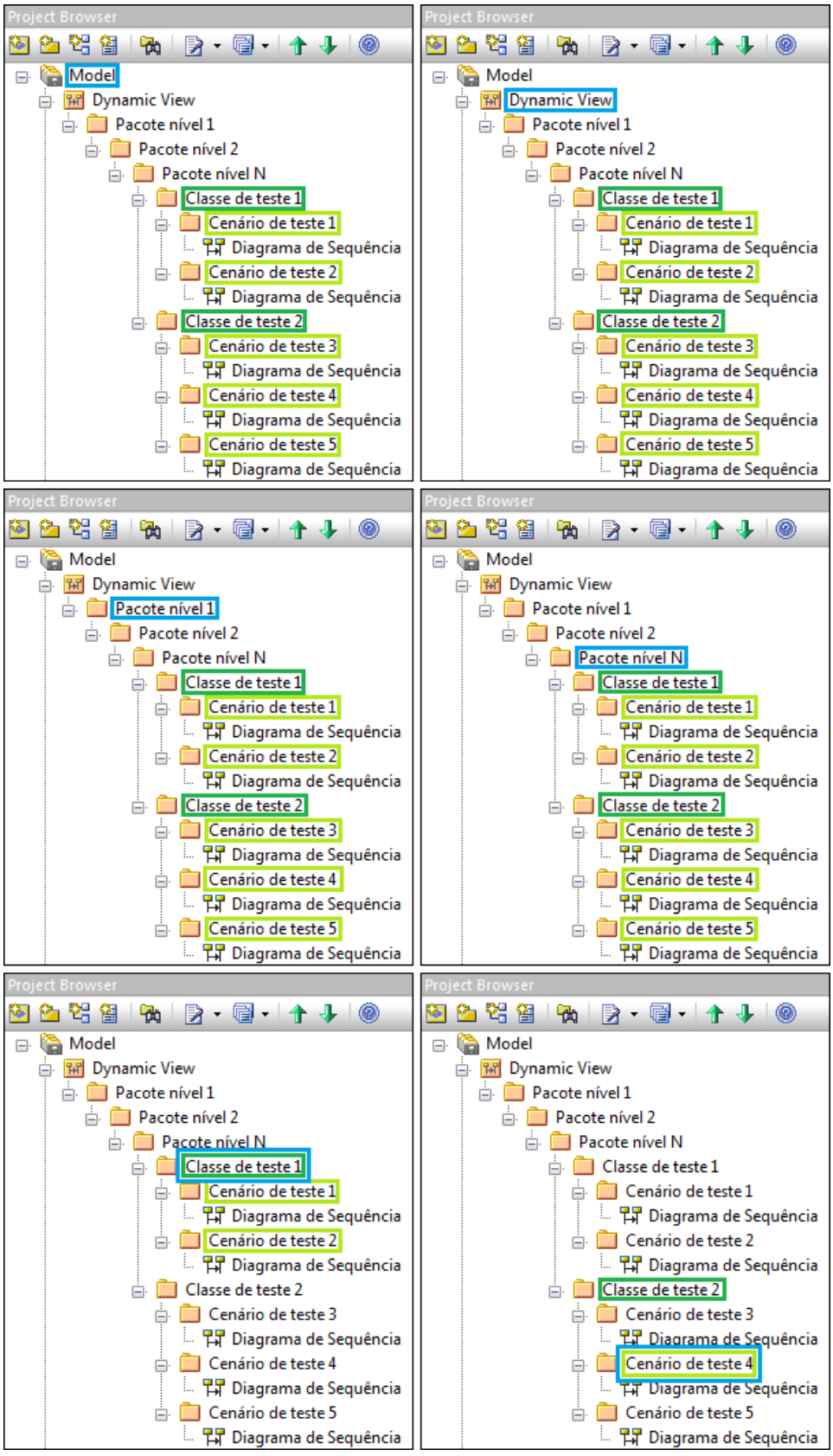


Figura 5.18: Classes e métodos de teste analisados em função da seleção do utilizador.

Capítulo 6

Validação

Neste capítulo é apresentado o processo de validação a que foi sujeita a implementação descrita anteriormente. São apresentados os casos de teste utilizados ao longo do trabalho e a análise de cobertura de funcionalidades para cada um deles. É ainda demonstrada a experimentação das novas funcionalidades num projeto já existente e finalmente a reformulação da análise comparativa do protótipo com as abordagens estudadas na análise do estado da arte, incluindo agora o FEUP-SBT-2.0 na análise.

6.1 Casos de teste e análise de cobertura de características

Durante a implementação foram utilizados três casos de teste criados com o objetivo de cobrir a maior parte das funcionalidades de modelação de testes e tipos de erros possíveis. Estes casos de teste foram extremamente úteis para guiar a correta evolução da implementação das novas funcionalidades e validar a conclusão de cada uma destas.

O mais simples dos exemplos utilizados é o denominado "ATM", que foi detalhadamente apresentado no capítulo 5. Nele está representado uma operação simplificada de levantamento bancário, contendo cenários parametrizados e fragmentos combinados condicionais, criando assim vários caminhos de execução possíveis para melhor analisar a funcionalidade de análise de cobertura.

O exemplo com o nome "Spreadsheet Engine" é um outro caso de teste, que foi também apresentado anteriormente no capítulo 2. Neste exemplo é modelado um mecanismo de folhas de cálculo, contando com vários diagramas de sequência, tornando-o no caso de teste utilizado mais completo, o que permitiu analisar uma maior diversidade de funcionalidades. Por exemplo, apenas neste caso de teste são exercitadas as funcionalidades de interação com o utilizador, conformidade estrita e exceções. São também apenas nele conferidos os tipos de erro *ReturnValueException* e *UnexpectedCallException*. Trata-se portanto de um caso de teste de importância elevada para validar aspetos mais específicos do modelo.

Além dos dois casos de teste anteriores, foi também utilizado o exemplo denominado "Observer". Este é um curto exemplo, que assim permitiu de forma versátil efetuar os primeiros testes a novas implementações, antes de partir para um teste mais profundo recorrendo ao exemplo "Spreadsheet Engine" que é bem mais complexo. Este caso de teste conta ainda com chamadas a métodos externos, ou seja, de classes não implementadas pelo utilizador e portanto, devido a uma limitação existente da biblioteca *TracingUtilities* que não deteta a ocorrência de interações internas quando estas são envolvem classes externas, ocorre um problema na análise de cobertura deste exemplo.

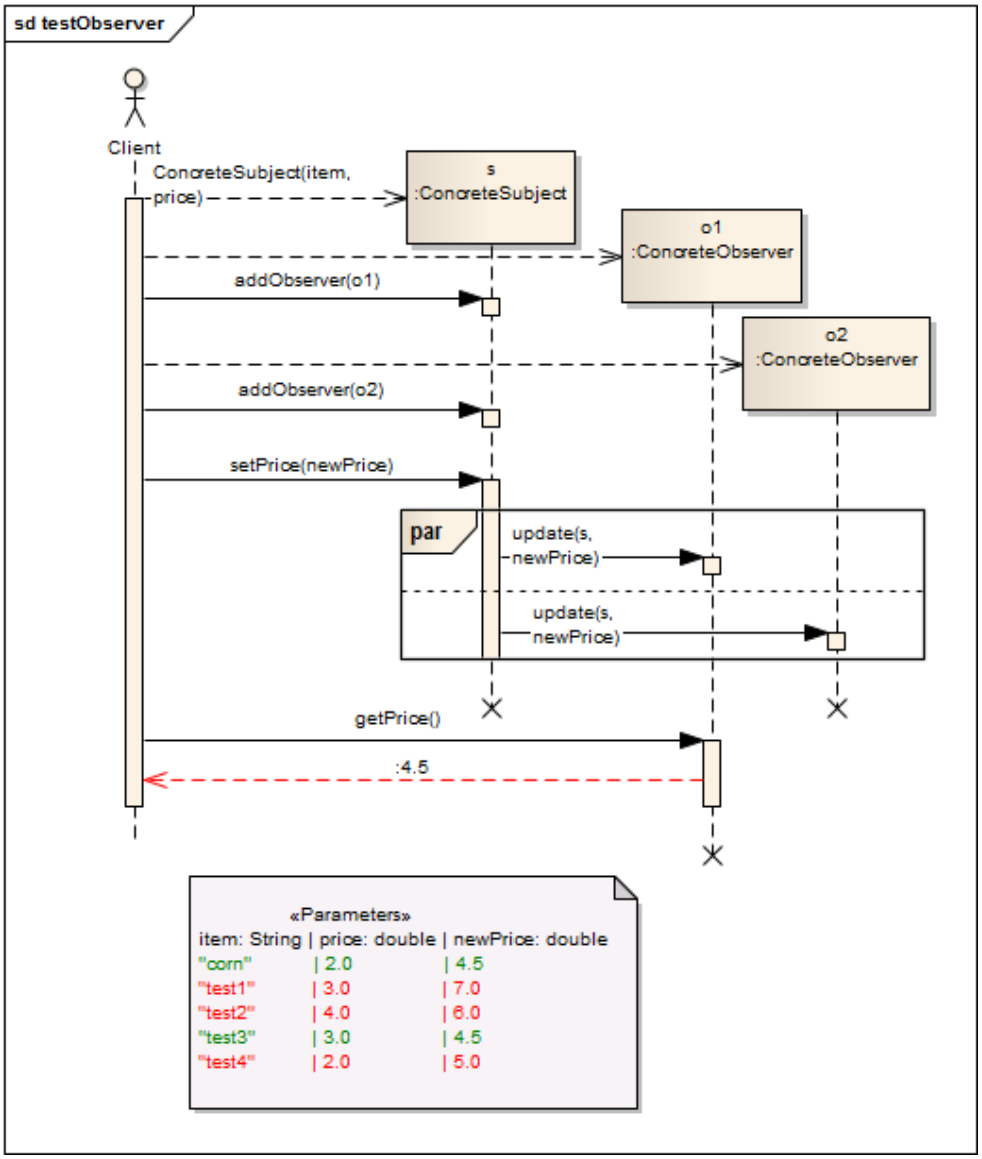


Figura 6.1: Diagrama de sequência do caso de teste "Observer".

Na figura 6.1 encontra-se modelado o diagrama de sequência do caso de teste "Observer", sendo visível o seu funcionamento. É criado um objeto concreto e dois observadores deste, atra-

vés da classe `java.util.Observable`. Posteriormente é realizada uma atualização no objeto concreto e espera-se que internamente a operação seja replicada nos observadores. Na figura é visível o resultado da execução do caso de teste, com apenas alguns dos casos de teste executados sem erro. As falhas foram introduzidas intencionalmente para verificar a ação em caso de erro.

Procurou-se que os anteriormente apresentados casos de teste, em conjunto, incluíssem nos seus modelos a grande maioria das especificidades suportadas pelo FEUP-SBT-2.0, para que desta forma pudesse ser posto à prova o funcionamento da ferramenta em cada destas, a maior parte delas já abordada ao longo do documento. Desde logo, tinha-se como intenção pôr à prova o atuação da ferramenta em cada uma das exceções definidas na biblioteca *Tracing Utilities*. É também analisado o correto funcionamento do analisador de cobertura em situações de cobertura total ou parcial. São ainda analisadas diversas *features* de modelação, como os 3 tipos de interações: internas, com aplicação cliente e de consola. É também analisado o comportamento da ferramenta para diagramas que contenham cenários parametrizados e os vários fragmentos combinados possíveis. A presença de várias classes de teste por pacote e vários métodos de teste por classe são também testadas. Finalmente, situações mais minuciosas como: conformidade estrita, isto é, tudo o que efetivamente acontece durante a execução deve estar representado no modelo, sendo que quando esta opção não é ativada, por defeito é utilizada conformidade não estrita, sendo nesse caso permitidas chamadas não especificadas no modelo; modelação de exceções; chamadas a métodos de classes externas.

De seguida é analisada a correta cobertura por parte de cada um dos casos de teste do conjunto de características descritas anteriormente. Em 6.1 está ilustrada a matriz de cobertura de características, sendo utilizados os ícones: ✔ caso a característica seja testada com sucesso no caso de teste; ✘ caso a característica seja exercitada no caso de teste, mas não é obtido o resultado pretendido; - caso a característica não se aplique ao caso de teste.

		Casos de Teste		
		Observer	*Spreadsheet Engine*	*ATM*
***Features* da modelação**	*Interações com Aplicação Cliente*	✔	✔	✔
	Interações com Utilizador	-	✔	-
	Interações Internas	✔	✔	✔
	Cenários Parametrizados	✔	✔	✔

	Fragmentos Combinados	✔	✔	✔
	Vários Cenários/Classes de Teste	-	✔	✔
	Conformidade Estrita	-	✔	-
	Exceções	-	✔	-
	Chamadas a métodos externos	✔	-	-
Tipos de Erros	*Assertion Error*	✔	✔	✔
	Mising Call Exception	✔	✔	✔
	Argument Exception	✔	✔	✔
	Return Value Exception	-	✔	-
	Unexpected Call Exception	-	✔	-
Nível de Cobertura	*Total*	✘	✔ / ✘	-
	Parcial	-	-	✔

Tabela 6.1: Matriz de cobertura de características dos casos de teste

Analisando a matriz de cobertura de características é possível concluir que todas as *features* de modelação de teste, tipos de erros e níveis de cobertura são testados com sucesso em pelo menos um caso de teste, tal como era pretendido. É ainda visível que em duas situações não é produzido o resultado desejado, ambas na funcionalidade análise de cobertura. A primeira, no caso de teste "Observer" acontece devido à limitação da biblioteca *Tracing Utilitites* que não deteta convenientemente a execução de métodos externos. Dada a existência de mensagens deste tipo no exemplo "Observer", estas mensagens são erradamente assinaladas como não executadas. Em relação à segunda falha, que acontece num dos diagramas definidos no caso de teste "Spreadsheet

Engine", esta deve-se à forma utilizada para modelar exceções cuja codificação não permite detetar em que mensagem exata foi lançada a exceção, sendo apenas detetada no final.

6.2 Experimentação num projeto existente

Para uma mais robusta validação da implementação das novas funcionalidades, recorreu-se a um projeto já existente, documentado e implementado, para nele serem executadas as novas funcionalidades de execução do código de teste e apresentação de resultados no próprio modelo UML. O objetivo era encontrar discrepâncias entre a especificação e implementação do projeto e, nesse processo, encontrar também limitações do FEUP-SBT-2.0, sobretudo em relação às especificidades de modelação do diagrama de sequência suportadas pelo protótipo.

O projeto a testar, denominado "*FileDiff*", foi desenvolvido na FEUP antes sequer da implementação do FEUP-SBT-1.0 e sem qualquer intenção de servir de suporte ao mesmo. Neste projeto foi implementado um comparador de ficheiros, que analisa a diferença entre duas versões consecutivas de ficheiros de código fonte, em número de linhas adicionadas, modificadas ou eliminadas, ignorando linhas em branco e comentários. Este utilitário tem sido usado pelo alunos do MIEIC no desenvolvimento de projetos para recolha de métricas de tamanho de código produzido/alterado. O diagrama de classes deste projeto encontra-se exposto em 6.2. Estão assim modeladas as classes: *FileDifferenceCLI*, contendo a função `main` do projeto e a implementação da interface através da linha de comandos; *SourceCodeParser* contendo métodos para interpretar e limpar o conteúdo recolhido de cada ficheiro; *DifferenceCalculator* responsável por calcular as diferenças entre os ficheiros; *Delta* que representa uma alteração; *Operation* trata-se de uma enumeração das possíveis alterações.

Encontrava-se também elaborado no modelo UML do projeto "*FileDiff*", um diagrama de sequência que se assumia representar plenamente a interação dos objetos que ocorreria na execução do "*FileDiff*". Ilustrado em 6.3 encontra-se esta primeira versão do diagrama de sequência. No entanto, esta não se encontrava num estado testável, originando imediatamente erros de compilação. Entre outras especificidades, este diagrama conta com instruções em pseudo-código, como por exemplo a última mensagem do diagrama `number of lines added, modified, deleted and unchanged`, não podendo assim ser compilado o código gerado a partir do mesmo.

A etapa que se seguiu foi adaptar a versão 1 (original) do diagrama de sequência para uma versão 2 considerada testável, isto é, compilável e executável. De seguida são apresentadas as alterações efetuadas neste processo, encontrando-se o seu resultado ilustrado em 6.4. Foi assim alterada a interação com o utilizador, passando-se a recorrer às *keywords* `start` e `display` para, respetivamente, iniciar o processo e receber informação. A utilização da opção da mensagem "*Assign To*", que não é suportada pelo FEUP-SBT-2.0, foi substituída pela criação de instâncias para cada objeto. O uso da opção "Condition" do bloco "Sequence Expression" das opções da mensagem foi também alterado, utilizando-se ao invés fragmentos combinados "*opt*". Todas as variáveis utilizadas no caso de teste, como as opções ou o nome dos ficheiros a comparar, foram

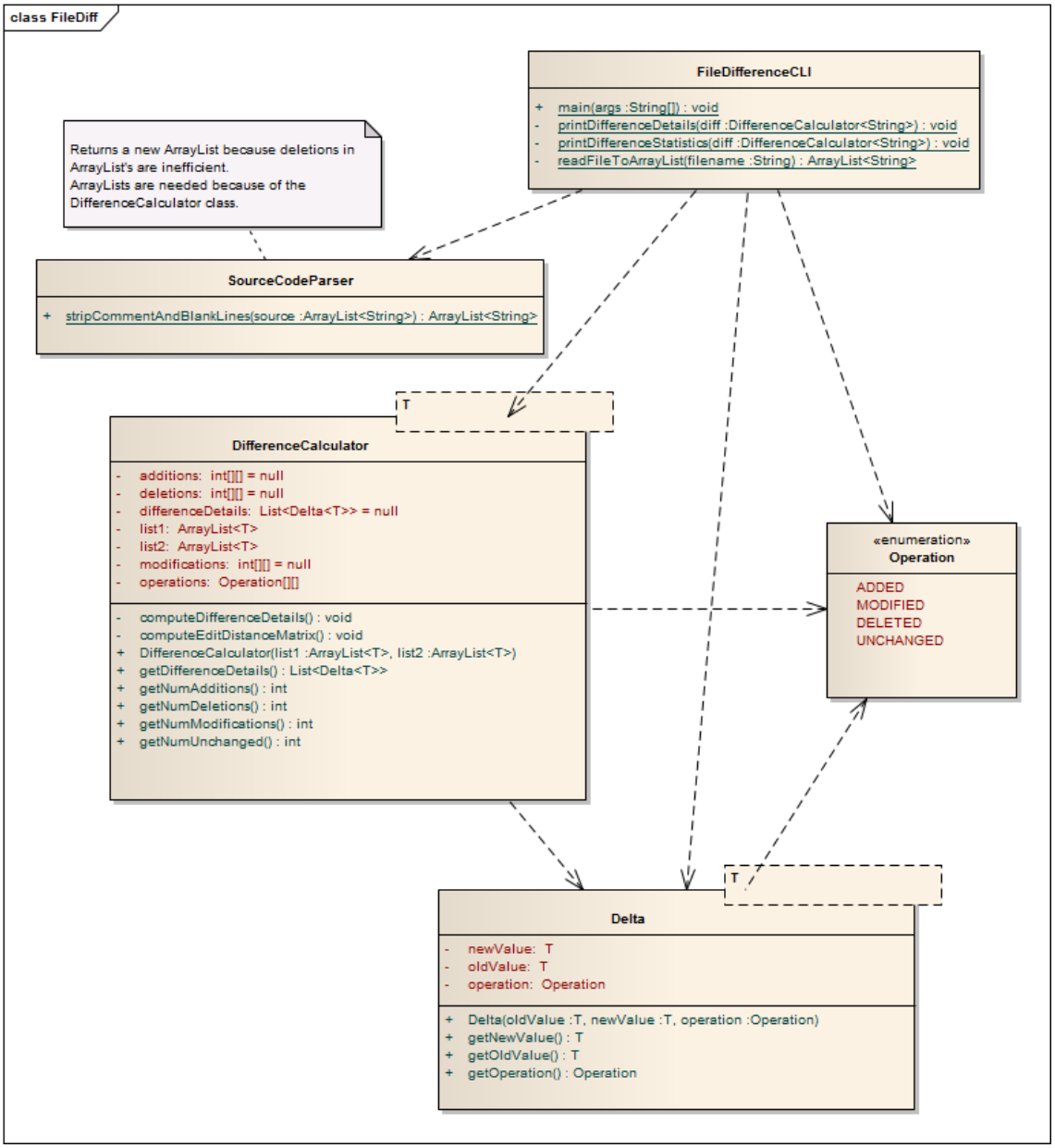


Figura 6.2: Diagrama de classes do projeto "*FileDiff*".

parametrizadas no bloco *«Parameters»*. De notar que o projeto já continha ficheiros de entrada de exemplo para teste manual, que foram registados neste bloco de parâmetros para automação desses testes.

O código gerado a partir desta versão 2 do diagrama de sequência foi compilado e executado analisando cobertura e o resultado encontra-se expresso no modelo em 6.4, apresentado através da coloração das mensagens. Foi desta forma possível detetar que ainda existiam inconsistências entre esta versão do diagrama de sequência, ou seja, a especificação de comportamento e a implementação. Desde logo é detetado um erro na parte final onde são exibidas as alterações en-

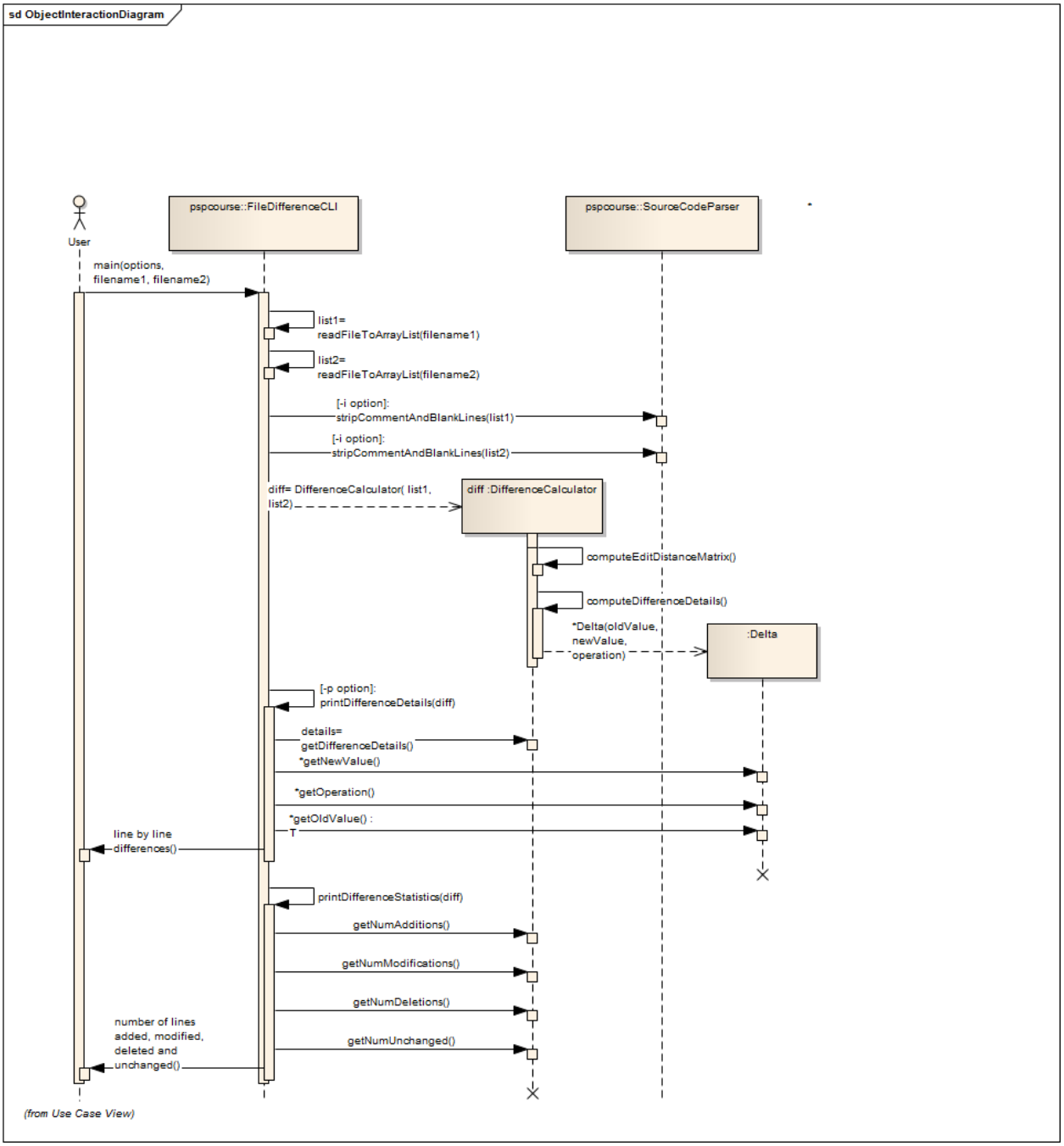


Figura 6.3: Diagrama de sequência (versão 1 - original) do projeto "*FileDiff*".

contradas. Este erro acontece porque se supõe que as diferentes alterações são todas impressas sequencialmente no final da execução, o que não acontece de facto. É também verificado que a mensagem `*getNewValue()` não é detetada contrariamente ao que era esperado. Como esta contém o *wildcard* "`*`", a não ocorrência desta mensagem não é encarada como erro. Todavia, existe aqui uma falha na modelação, dado que na implementação, `getOperation()` acontece sempre antes de `getNewValue()` mas não está assim modelado no diagrama de sequência.

Finalmente, foram corrigidas as inconsistências do modelo na sua versão 2, criando assim a versão 3 e final do mesmo. Os métodos `display` das alterações que ocorriam no fim de forma sequencial foram colocados, um por um, logo após o retorno da alteração respetiva detetada. Enquanto que a utilização de *wildcard* "`*`" foi substituída pelo fragmento combinado "*loop*", sendo colocada a mensagem `getOperation()` antes de um bloco "*alt*", cujas opções são `getNewValue()` ou `getOldValue()`, de acordo com a implementação.

A utilização de uma abordagem por engenharia reversa, gerando o diagrama de sequência UML que representa a interação entre os objetos na execução de um programa, foi analisada como alternativa a este processo. No entanto, obter um modelo UML com o grau de abstração ao nível do apresentado em 6.5, com inclusão de fragmentos combinados, parametrização, omissão de mensagens irrelevantes, entre outras especificidades, seria inviável através duma abordagem deste tipo.

Em conclusão, através da execução do FEUP-SBT-2.0, foram encontradas disparidades entre um modelo que se assumia conforme e a sua implementação. A utilização do modo de operação "Executar os testes analisando cobertura" foi extremamente útil neste processo, detetando mensagens que não foram efetivamente executadas e consequentemente inconformidades, comprovando assim a utilidade que se previa ter. Conclui-se também que o esforço para construir de raiz um diagrama testável não seria significativamente diferente da construção de um diagrama informal, com a vantagem de se poder verificar a consistência automaticamente.

6.3 Análise comparativa em relação ao estado da arte e versão anterior

No capítulo 3, após a análise do estado da arte, foi produzida uma tabela com os pontos principais para comparação das várias abordagens com o FEUP-SBT-1.0. À mesma tabela é agora adicionada a versão 2.0 do FEUP-SBT-2.0 para uma melhor perceção dos pontos em que esta nova versão traz mais vantagens. Uma vez mais, são utilizados como identificadores um sinal positivo verde e um sinal negativo vermelho para serem melhor percetíveis os pontos fortes e fracos, respetivamente, de cada uma das abordagens.

	SeDiTeC	**Abordagem MDA**	**SCENTOR**	**FEUP-SBT-1.0**	***FEUP-SBT-2.0***
Integração	Together	Eclipse	Rational Rose	Enterprise Architect	Enterprise Architect

Abordagem da Automação de Testes	Executa diretamente diagramas de sequência.	Transforma em modelo de teste e depois em código de teste.	Gera *drivers* de teste.	Gera JUnit.	Gera JUnit e executa-o de forma transparente.
Feedback	Comparação do diagrama inicial com o resultado da execução.	Testes JUnit executados pelo utilizador.	Resultados apresentados pela ferramenta.	Testes JUnit executados pelo utilizador.	Resultados apresentados no modelo.
Geração de stubs	Sim.	Não.	Não.	Sim.	Sim.
Versão UML suportada	UML 1.x	UML 1.x	-	UML 2.x	UML 2.x
Disponibilidade	Não disponível.	Disponível sem custos.	WEB: *offline*.	Protótipo interno.	Disponível para experimentação.[1]
Dados de teste	Separados do diagrama.	Especificados separadamente do diagrama, codificados hardcoded.	Separados do diagrama.	Cenários parametrizados e dados de teste separados.	Cenários parametrizados e dados de teste separados.
Linguagens Alvo	Java.	Java, SmallTalk (potencial para várias outras).	Java.	Java.	Java.
Captura das interações internas	Captura interações internas.	Captura em listagens.	Não.	Captura recorrendo a AOP.	Captura recorrendo a AOP.

[1] https://feupload.fe.up.pt/get/l16y7nv72cBqeO0

Verificação das interações internas	Verifica automaticamente.	Verificação manual pelo utilizador.	-	Verifica automaticamente atendendo à conformidade definida.	Verifica automaticamente atendendo à conformidade definida.
Interface de Utilizador (UI)	Desenho de diagramas no Together. UI da aplicação não disponível.	Definição da SMC no Eclipse Modeling Framework.	Desenho de diagramas no Rational Rose. Website para execução dos testes.	Desenho dos diagramas e geração dos testes através dos menus do EA.	Desenho dos diagramas, geração e execução dos testes e análise dos resultados diretamente no EA.
Verificação da Consistência do Modelo	Não.	Não.	Não.	Verifica consistência e completude dos modelos.	Verifica consistência e completude dos modelos.
Análise de Cobertura	Não.	Não.	Não.	Não.	Sim.

Tabela 6.2: Tabela comparativa das abordagem com o FEUP-SBT-1.0

Desta nova análise comparativa entre as várias abordagens estudadas e as duas versões da ferramenta FEUP-SBT é possível verificar que os pontos fortes do FEUP-SBT-2.0 em relação às restantes é a abordagem utilizada e *feedback* e, consequentemente, a usabilidade da interface para o utilizador, que possibilita um processo de teste e análise dos resultados na mesma ferramenta, sem necessidade de ações extra. Para além disso, a nova funcionalidade de análise de cobertura é um aditivo bastante útil à experiência de teste, que não se encontra presente nas outras abordagens. Em relação aos pontos fracos, a limitação da linguagem alvo dos testes a Java continua a ser uma limitação do FEUP-SBT e um ponto forte da abordagem MDA, tratando-se de uma ideia para trabalho futuro.

6.4 Limitações detetadas

Através do processo de validação foram sendo detetadas algumas limitações do FEUP-SBT-2.0. Desde logo, ao nível da performance, esta não atinge os níveis desejados devido à falta de

rapidez demonstrada pela API do Enterprise Architect. Outra limitação detetada, e já abordada anteriormente, prende-se com a incapacidade da biblioteca *Tracing Utilities* intercetar chamadas a métodos de classes externas, o que pode impossibilitar a correta execução das funcionalidades do FEUP-SBT-2.0 em alguns modelos, no entanto pode ser contornado com "descompilação". Finalmente, a também já abordada falha na codificação da modelação de exceções nos diagramas de sequência, inviabiliza a deteção da mensagem exata onde foi lançada a exceção e consequentemente compromete a execução do analisador de cobertura onde estas situações se verifiquem.

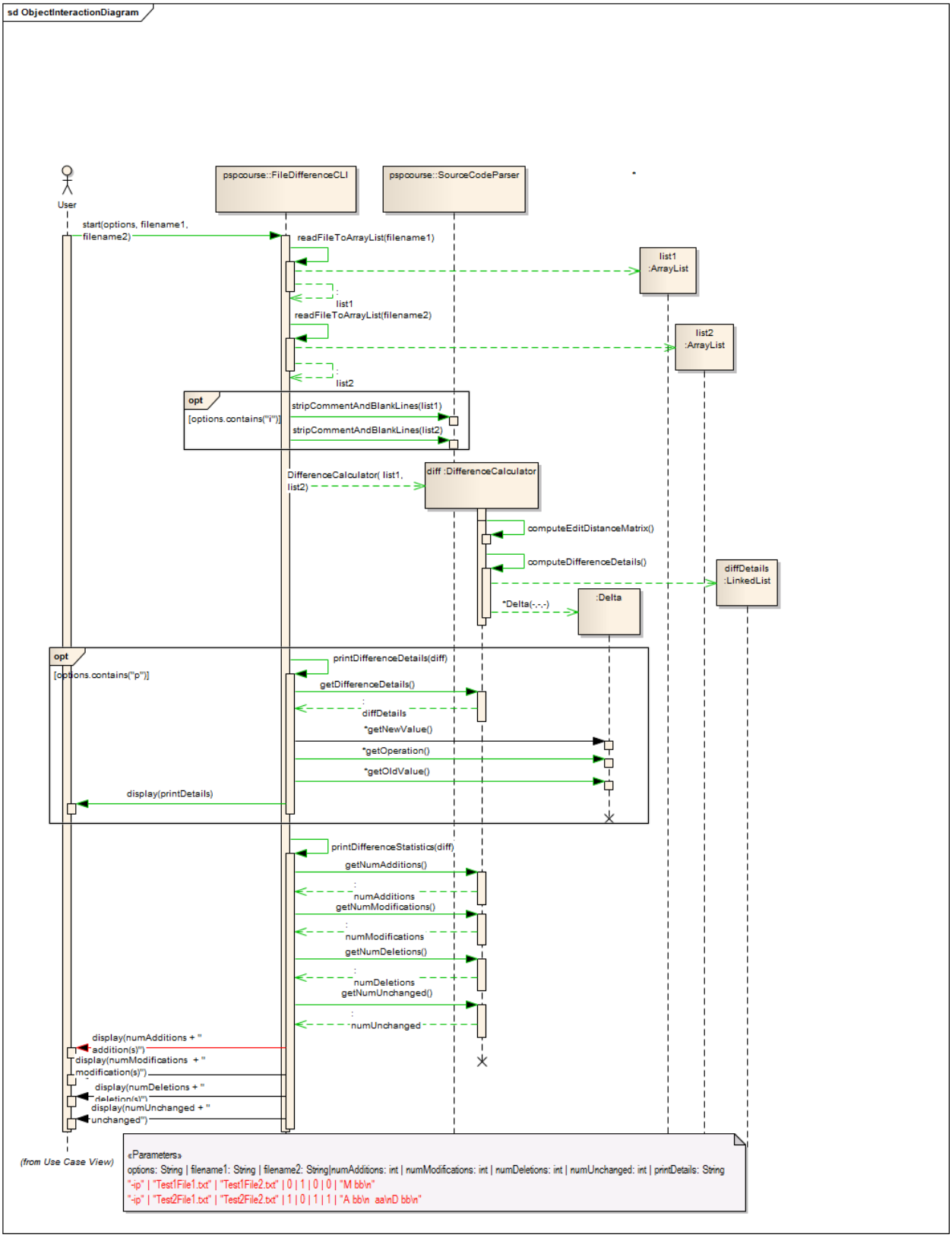


Figura 6.4: Diagrama de sequência (versão 2 - testável) do projeto "*FileDiff*", já colorido de acordo com os resultados da execução.

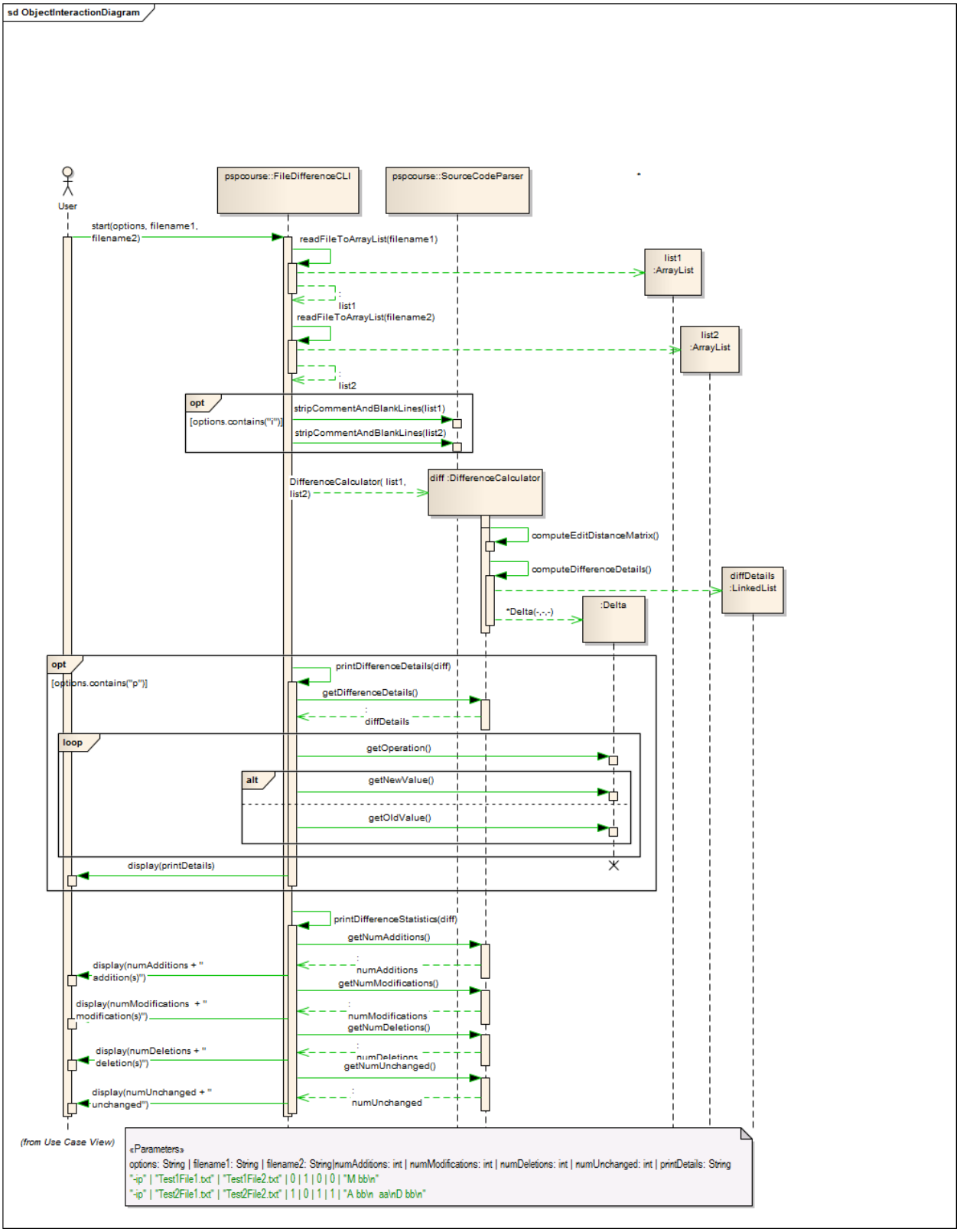


Figura 6.5: Diagrama de sequência (versão 3 - corrigida) do projeto "*FileDiff*", já colorido de acordo com os resultados da execução.

Capítulo 7

Conclusões

No trabalho de dissertação foi evoluído o protótipo experimental FEUP-SBT-1.0 para uma versão 2.0, que contém uma série de novas funcionalidades, de onde se destacam as funcionalidades de execução direta dos cenários de teste e visualização de resultados na ferramenta de modelação Enterprise Architect.

Numa primeira fase, foi efetuado um estudo aprofundado da versão inicial do protótipo, para a deteção de falhas ou limitações e uma revisão bibliográfica sobre o tema "automação de testes baseados em diagramas de sequência UML".

Seguidamente são apresentados os resultados alcançados pelo trabalho de dissertação, confrontando os mesmos com os objetivos estipulados. É ainda registado o trabalho futuro que poderá ter função de guia para evoluir a ferramenta FEUP-SBT para novas e mais poderosas versões.

7.1 Resultados alcançados

A funcionalidade principal a desenvolver, que permite executar o código de teste gerado de forma transparente e verificar os resultados da execução diretamente no ambiente de modelação, foi implementada com sucesso na sua totalidade, incluindo as diversas formas de apresentação de resultados no modelo, que incidem na coloração das mensagens e/ou fragmentos combinados, adição de mensagens de erro nas notas dos elementos em questão, edição dos estereótipos dos cenários de teste e colorindo cada cenário parametrizado conforme os respetivos resultados da sua execução e mensagens auxiliares de informação.

A funcionalidade para análise de cobertura dos testes foi também ela concluída com sucesso, sendo detetadas as mensagens efetivamente executadas e utilizando um esquema de cores para colorir o modelo em conformidade com os resultados.

Para além das funcionalidades de execução direta de testes, foi também desenvolvido um módulo de configurações, onde são editadas todas as opções necessárias para o correto funcionamento das funcionalidades anteriores, incrementando a usabilidade da utilização do protótipo FEUP-SBT-2.0.

Toda esta implementação permitiu reestruturar os componentes que compõem o FEUP-SBT-2.0, sobretudo o *add-in* para o EA, estruturando-o num sistema de classes mais robusto e que facilita a sua manutenibilidade. A análise do estado da arte foi também uma importante etapa concluída com êxito, registando algumas abordagens que se assemelham ao FEUP-SBT-2.0, permitindo registar ideias para trabalho futuro e verificar possíveis vantagens que a ferramenta pode acrescentar às já existentes.

7.2 Trabalho futuro

Existem algumas limitações do FEUP-SBT-2.0, que ao longo do documento foram registadas e que em trabalho futuro era importante serem corrigidas, para aumentar ainda mais o potencial de utilização da ferramenta. Entre estas limitações estão questões de desempenho e suporte a especificidades do modelo, como chamadas a métodos externos e modelação de exceções, bem como alguns fragmentos combinados que não são ainda suportados.

A evolução da ferramenta para outras linguagens, para além da suportada atualmente (Java), podem também ser uma importante evolução da ferramenta, até por se tratar de um ponto fraco em relação a uma das abordagens analisadas no estado da arte.

Registado como trabalho futuro encontra-se também a realização de uma experiência, utilizando a *framework* de métricas de PSP [Hum05], para avaliar a produtividade e qualidade do desenvolvimento de *software* utilizando a abordagem aqui especificada em relação a outras.

Sobretudo para pôr em prática as capacidades do analisador de cobertura, em trabalho futuro poderá ser construído um gerador automático de dados de teste, com capacidade para identificar fragmentos combinados condicionais que poderão originar vários traçados de execução, tendo como objetivo forçar a execução de todos os caminhos possíveis.

Anexo A

Tutorial de instalação e configuração

Nesta secção são apresentados em detalhe os passos necessários para conclusão da instalação, bem como as configurações necessárias para habilitar o *add-in* a executar todos os seus modos de operação.

A.1 Processo de instalação do *add-in*

De seguida são detalhadas, passo por passo, as etapas necessárias para a conclusão da instalação do *add-in* para Enterprise Architect.

Passo 1

Primariamente, é necessário ter disponível as duas bibliotecas de vínculo dinâmico (dll) referentes ao add-in desenvolvido: *TestGenerator.dll* e *Interop.EA.dll*. Estas duas bibliotecas devem ser colocadas num diretório à escolha. Para este tutorial, será usado como forma de exemplo o diretório apresentado na Instrução A.1.

```
C:\EAAddins
```

Instrução 5.1: Diretório exemplo para armazenamento das bibliotecas dll.

Passo 2

O passo seguinte passa pelo registo da biblioteca *TestGenerator.dll* no sistema. Para tal é utilizada a Ferramenta de Registo de Assemblagem (Regasm.exe) que vem incluída na plataforma .NET. Para efetuar o registo deve ser executada a Instrução A.2 na linha de comandos.

```
C:\WINDOWS\Microsoft.NET\Framework\<none version>\RegAsm.exe
"C:\EAAddins\TestGenerator.dll" /codebase
```

Instrução 5.2: Registo da biblioteca TestGenerator.dll.

A etiqueta <*version*> deve ser substituída pela versão da plataforma .NET instalada no sistema. Por fim, deve ser verificado que o registo foi efetuado com sucesso, obtendo uma resposta idêntica à presente em A.1.

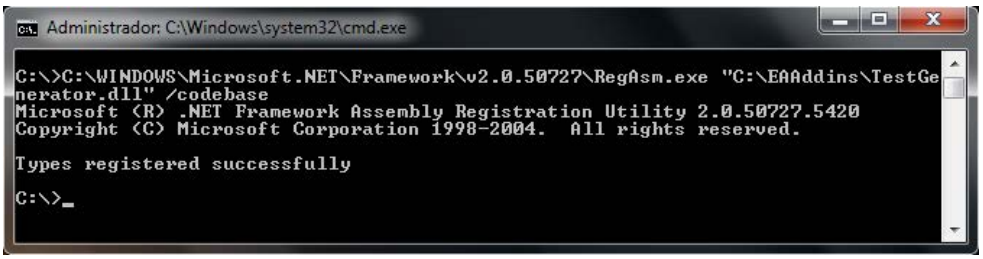


Figura A.1: Registo da biblioteca TestGenerator.dll.

Passo 3

Seguidamente, é necessário que o Enterprise Architect reconheça a presença do *add-in* FEUP-SBT-2.0. Para tal, deve ser colocada uma nova entrada no registo utilizando o Editar de Registos [Win12]. É possível aceder ao Editor de Registos clicando em "Iniciar"/"Executar" (ou através das teclas de atalho "Windows"+"R") e introduzindo `regedit`, conforme está expresso em A.2.

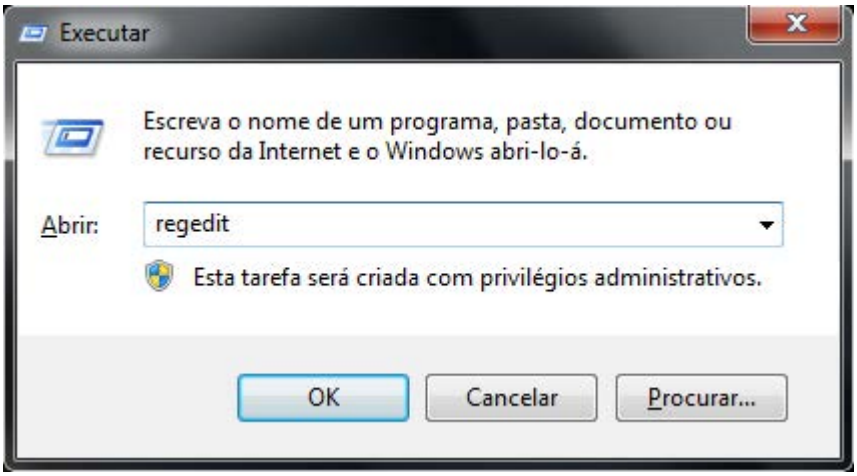


Figura A.2: Meio de acesso ao Editor de Registos.

Passo 4

Aberto o Editor de Registos, deve ser localizado o caminho apresentado na Instrução A.3. Deve ser criada dentro da localização referida, uma chave com o nome `EAAddins`, caso esta ainda não exista. A figura presente em A.3 demonstra o processo de adição de uma nova chave. Para validar as operações referidas neste passo, a estrutura de pastas no Editor de Registos deve ter uma estrutura semelhante à apresentada em A.4.

```
HKEY_CURRENT_USER\Software\Sparx Systems
```

Instrução 5.3: Localização dos EA *Add-Ins* no Editor de Registos.

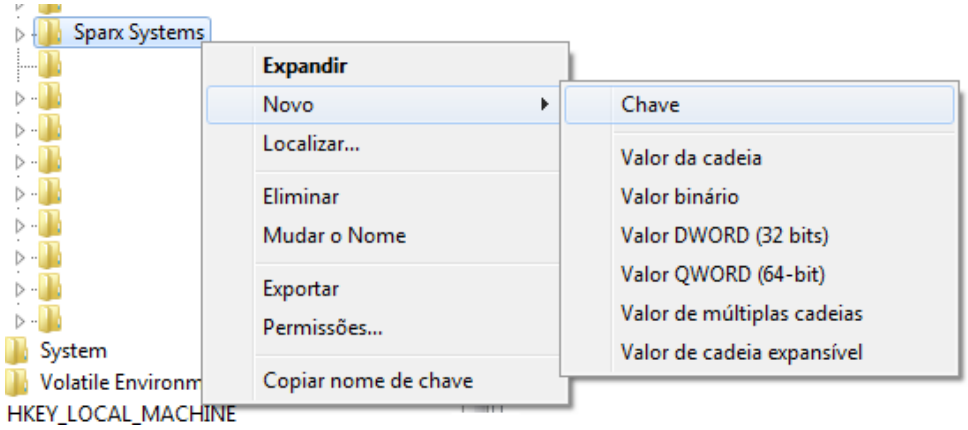


Figura A.3: Adição de uma nova chave no Editor de Registos.

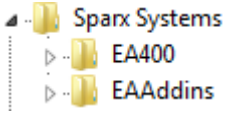


Figura A.4: Estrutura de pastas no final do Passo 4.

Passo 5

De seguida, é necessário adicionar uma chave dentro do diretório criado no passo anterior com o nome do projeto, neste caso, `TestGenerator`. Tal como anteriormente, basta seguir as instruções em A.3 para adicionar a chave e o resultado desta ação pode ser validado comparando a estrutura de pastas no Editor de Registos com a presente em A.5.

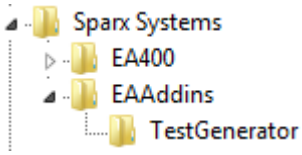


Figura A.5: Estrutura de pastas no final do Passo 5.

Passo 6

Neste passo, é necessário especificar o valor padrão assinalado como "(Predefinição)" da chave "TestGenerator" criada no passo anterior. Em A.6 está demonstrado o processo de edição do valor da chave "(Predefinição)".

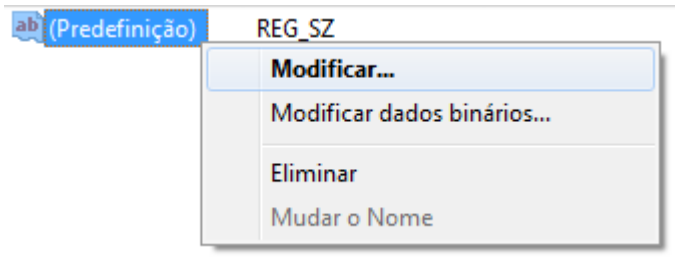


Figura A.6: Edição do valor de uma chave no Editor de Registos.

Passo 7

Com a janela de edição do valor padrão da chave TestGenerator aberta, é agora necessário editar o seu valor com a seguinte sequência <*ProjectName*>.<*ClassName*>, neste caso, "TestGenerator.TestGenerator", conforme ilustrado em A.7.

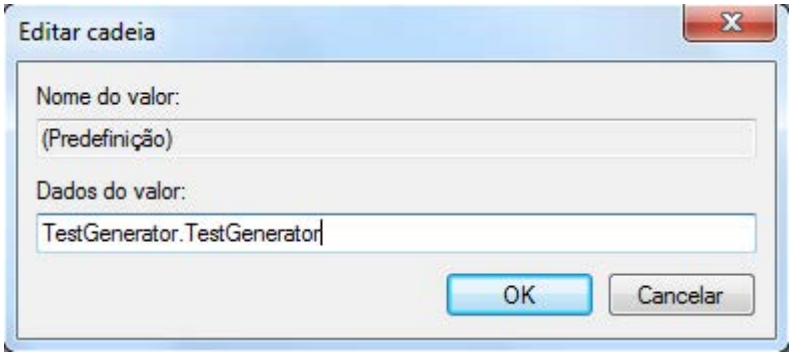


Figura A.7: Valor padrão da chave TestGenerator.

Passo 8

Se todos os passos anteriores forem concluídos com sucesso, nesta altura o *add-in* deve estar corretamente instalado no sistema, bastando apenas confirmar que o programa Enterprise Architect o detetou efetivamente e ativar o seu carregamento no início da execução do programa. Para efetuar estas operações deve ser executado o Enterprise Architect, acedendo posteriormente ao menu "Add-Ins" e de seguida, "Manage Add-Ins...", conforme apresentado em A.8.

Na janela de Gestão de *Add-Ins* deve ser confirmada a presença do *add-in* com o nome "TestGenerator" e o seu *Status* a *Enable*, isto é, de que se encontra no estado ativado. Para que o estado

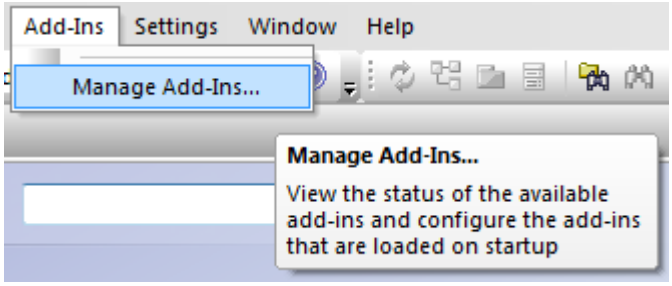


Figura A.8: Acesso ao Gestor de *Add-Ins* do Enterprise Architect.

do *add-in* possa estar ativado, é necessário que seja assinalada a opção *Load on Startup*, isto é, confirmar o carregamento do *add-in* no início da execução do Enterprise Architect. Ao confirmar estas opções, será necessário reiniciar o Enterprise Architect. Depois de reaberto, deve ser efetuada uma validação final da instalação do *add-in*, acedendo ao Gestor de *Add-Ins* conforme previamente visto em A.8 e verificar que todos os campos se encontram com aspeto idêntico ao ilustrado em A.9.

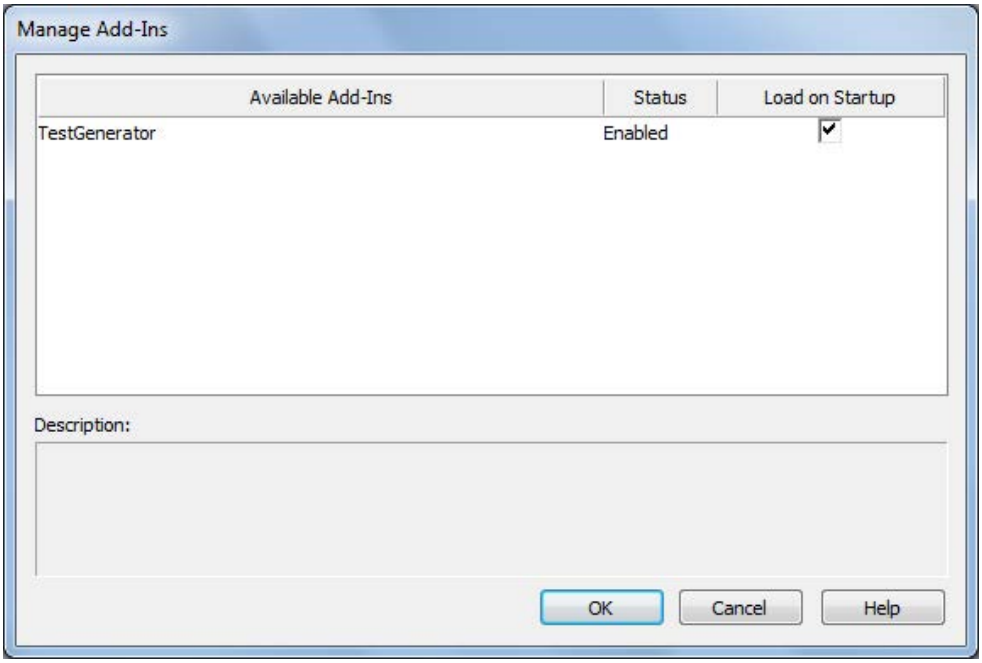


Figura A.9: Verificação da instalação com sucesso do *add-in*.

A.2 Processo de configuração

As funcionalidades que permitem a execução direta dos cenários de teste, fazem-no através de um executor que recorre ao pacote de desenvolvimento *Java JDK*. Assim, para possibilitar a utilização dos comandos disponibilizados pelo *Java JDK*, este deve ser adicionado às variáveis

ambiente dos sistema, após a sua instalação. Para tal, devem ser efetuados os seguintes passos, que podem ser ligeiramente diferentes dependendo da versão do Sistema Operativo Windows em funcionamento, não se tratando de diferenças impeditivas da conclusão com sucesso desta operação.

1. Aceder às "Propriedades do Sistema", no separador "Avançadas", clicando em "Iniciar"/"Executar" (ou através das teclas de atalho "Windows"+"R") e introduzindo `systempropertiesadvanced`.
2. Clicar em "Variáveis de ambiente" no canto inferior direito da janela.
3. Dentro da caixa "Variáveis do sistema", clicar em "Novo..."
   (a) Como "Nome da variável" utilizar `JAVA_HOME`.
   (b) Em "Valor da variável", colocar a localização da instalação do *Java JDK*. Por exemplo: `C:\Program Files\Java\jdk1.7.0_07`.
   (c) Clicar em "Ok".
4. Verificar se já existe uma chave com o nome "CLASSPATH". Se não existir, criá-la da forma explicada no ponto "*3. (a)*". Se sim, fazer duplo clique sobre ela, para abrir o seu menu de edição.
   (a) Acrescentar [1] os seguintes caminhos ao "Valor da variável":

```
.;
%JAVA_HOME%\lib;
%JAVA_HOME%\lib\tools.jar;
%JAVA_HOME%\lib\dt.jar;
%JAVA_HOME%\lib\htmlconverter.jar;
%JAVA_HOME%\jre\lib;
%JAVA_HOME%\jre\lib\rt.jar.
```

   (b) Clicar em "Ok".
5. Novamente na caixa "Variáveis do sistema", fazer duplo clique na chave "Path".
   (a) Acrescentar o seguinte texto ao "Valor da variável": `%JAVA_HOME%\bin`.
   (b) Clicar em "Ok".
6. Clicar em "Ok"na janela "Variáveis de ambiente" e "Propriedades do sistema".

Por forma a constatar a correta instalação do *Java JDK*, escrever o comando `javac -version` na linha de comandos e verificar que a resposta é a mesma versão do Pacote de Desenvolvimento instalado, conforme exemplificado em A.10.

[1] Caso já exista texto no "Valor da variável", para acrescentar mais caminhos, basta inserir um ponto e vírgula seguido dos novos caminhos a inserir.

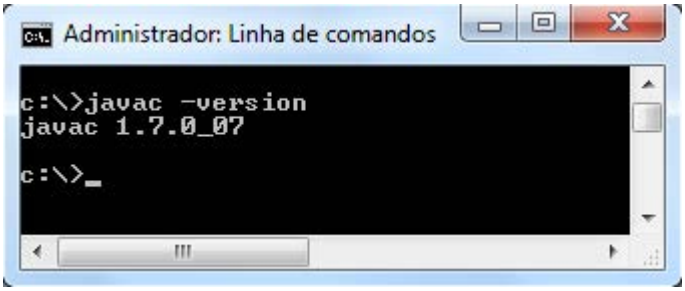


Figura A.10: Verificação da instalação com sucesso do Pacote de Desenvolvimento *Java JDK*.

Anexo B

Interpretação do código de teste gerado

Embora o objetivo principal desta dissertação seja automatizar o processo de compilação, execução e exposição de resultados, escondendo assim do utilizador o código de teste gerado, é também possível ser o próprio utilizador a criar o código de teste JUnit, tirando proveito das funcionalidades que a biblioteca *TracingUtilities* disponibiliza. Nesta secção, é assim apresentado o código de teste gerado pelo *add-in*.

Antes de mais, é conveniente esclarecer que o código de teste gerado através da opção "Executar os Testes Analisando Cobertura" tem diferenças em relação ao gerado através do modo de geração ou execução básica. De seguida, são analisados individualmente os vários aspetos do código de teste, utilizando como exemplo o código de teste gerado pelo exemplo "ATM" ilustrado anteriormente em 5.7.

B.1 Estruturação da(s) classe(s) de teste

O primeiro passo a seguir é criar uma classe de teste Java. Para tal, deve ser criado um ficheiro com extensão .java, importar todas as classes da biblioteca *TracingUtilities*, através da instrução da linha 1 em B.1 e criar uma classe que estende `"InteracTestCase"`, como é visível na linha 3.

```
1 import traceutils.*;

3 public class ATMTest extends InteracTestCase {
```

Excerto de Código B.1: Definições de importações e classes estendidas

Na classe definida, são implementados todos os métodos que representam cada um dos cenários de teste desejados. No exemplo "*ATM*", apenas existe modelado um cenário de teste pelo que o código de teste respetivo apenas contará com um método com o mesmo nome desse cenário, precedido de "test" caso o nome assim não comece. Na linha 1 de B.2 encontra-se o cabeçalho do método de teste `"testATM"`, que conta como parâmetros do método os mesmos parâmetros

definidos no bloco "*Parameters*" no modelo UML. Por cada linha desse bloco, ou seja, por cada caso de teste, é criado um novo método na classe de teste, que simplesmente invoca o seu método origem com os parâmetros indicados. Como no exemplo "*ATM*" estavam 2 casos de teste definidos, nas linhas 5 e 9 de B.2 encontram-se os 2 métodos auxiliares respetivos, com o mesmo nome do método origem mais o seu índice de ocorrência. Esta é a forma genérica para a criação de cenários parametrizados.

```
1  public void testATM(double balance, double amount, boolean log, String date,
       String type) throws Exception {
     // ...
3  }

5  public void testATM_0() throws Exception {
     testATM(100, 150, true, "01-01-01", "NoMoney");
7  }

9  public void testATM_1() throws Exception {
     testATM(100, 50, false, "01-01-90", "Money");
11 }
```

Excerto de Código B.2: Estrutura dos métodos de teste do exemplo "*ATM*"

B.2 Teste de API

Definida a estrutura da classe de teste, segue-se uma descrição mais pormenorizada da codificação de mensagens e fragmentos combinados do modelo em código de teste. As mensagens nas quais não existe interações externas, restringindo-se assim às mensagens dirigidas diretamente de e para o Ator definido no diagrama de sequência, são codificadas de uma forma simples. Se não existir retorno, a mensagem é codificada diretamente. Por exemplo, no construtor da figura B.1 é codificado diretamente pelo construtor presente no excerto de código B.3.

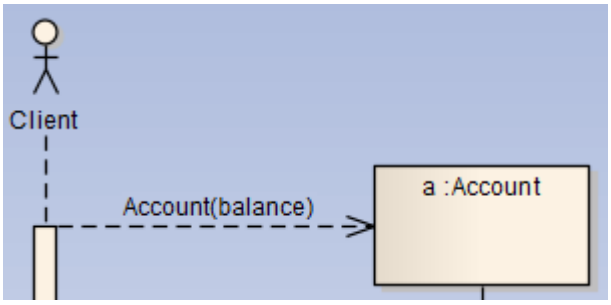


Figura B.1: Construtor de "*Account*" no exemplo "*ATM*".

```
1 Account a = new Account(balance);
```

Excerto de Código B.3: Construtor de "*Account*" no exemplo "*ATM*".

Caso uma mensagem apenas de interação externa possua mensagem de retorno, como é o caso do exemplo da figura B.2, esta é codificada com recurso ao método JUnit `"assertEquals"` demonstrado no excerto de código B.4. Este método `"assertEquals"` foi redefinido e conta com 4 parâmetros. O primeiro tem o formato `|DiagramMessageID=<id>|`, onde `<id>` deve ser substituído pelo identificador da mensagem no diagrama de sequência. De forma semelhante, o segundo parâmetro tem o formato `|RetID=<id>|`, onde `<id>` é o identificador da mensagem de retorno. O terceiro parâmetro contém o valor da mensagem de retorno que será comparado com o valor resultante da mensagem que é introduzida no quarto parâmetro.

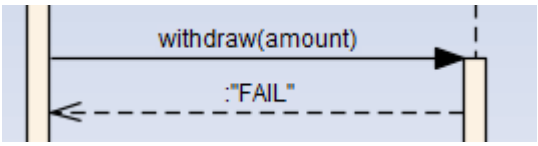


Figura B.2: Exemplo de mensagem com retorno no exemplo "*ATM*".

```
1 assertEquals("|DiagramMessageID=10|", "|RetID=35|", "FAIL", a.withdraw(amount))
    ;
```

Excerto de Código B.4: Exemplo de mensagem com retorno no exemplo "*ATM*".

B.3 Análise das interações internas

Para ser possível conferir a correta execução das mensagens internas definidas no modelo, é necessário codificar essas mesmas interações recorrendo a estruturas definidas na biblioteca *TracingUtilities* que de seguida são expostas.

B.3.1 Chamadas

A estrutura elementar para codificar uma mensagem é a estrutura "*Call*", onde é possível definir através dos seus argumentos, todos os parâmetros de uma chamada, como é visível no excerto de código B.5. Os argumentos `"messageID"` e `"returnMessageID"` representam os identificadores da mensagem origem e mensagem de retorno, respetivamente. Já os argumentos `"className"` e `"targetObject"` representam o nome da classe e o objeto alvos. Para representar o nome do méotodo, a lista de argumentos e o objeto de retorno são utilizados, respetivamente, os argumentos `"methodName"`, `"args"` e `"ret"`. Por último, podem ser definidas no último argumento `"nestedCalls"` as chamadas ou fragmentos combinados encaixados na chamada codificada.

```
Call(int messageID, String className, Object targetObject, String methodName,
    Object[] args, int returnMessageID, Object ret, CallNode ... nestedCalls)
```

Excerto de Código B.5: Estrutura "*Call*" da biblioteca *"TracingUtilities"*.

No exemplo "*ATM*", é possível constatar a existência de interações internas que têm de ser codificadas com a estrutura anterior. Um desses exemplos está na figura B.3, cuja codificação em código de teste se encontra no excerto de código B.6. A primeira mensagem `"withdraw(amount)"` tem como identificador o número `6`, a sua classe alvo é `"Account"` contida no pacote `"ATM"`, logo `"ATM.Account"`, e o objeto alvo é o `"a"` criado anteriormente. Em relação ao método, o seu nome é `"withdraw"` e conta com apenas um parâmetro, o `"amount"`. Quanto ao retorno, é a mensagem com o identificador `9` e o seu valor é `"OK"`. Como último argumento, e conforme é visível na figura B.3, está a especificação da mensagem `"setBalance(balance-amount)"` realizada de forma semelhante à descrita anteriormente. Tem como únicas diferenças, por não ter mensagem de retorno, o identificador da mesma com o valor `-10`[1] e o objeto de retorno a `"null"`. Como não contém mensagens encaixadas, não conta com o último argumento `"nestedCalls"`.

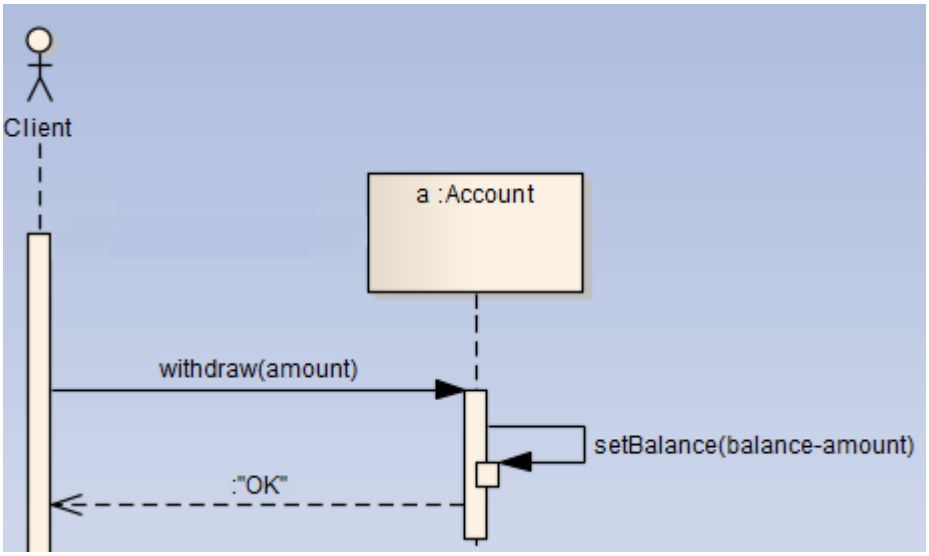


Figura B.3: Exemplo de interações internas no exemplo "*ATM*".

```
new Call(6, "ATM.Account", a, "withdraw", new Object[] {amount}, 9, "OK",
  new Call(8, "ATM.Account", a, "setBalance", new Object[] {balance-amount},
      -10, null)
)
```

Excerto de Código B.6: Exemplo de interações internas no exemplo "*ATM*".

[1] Valor utilizado em casos de não aplicabilidade

B.3.2 Fragmentos combinados

Os fragmentos combinados já discutidos anteriormente são codificados distintamente entre si. Por exemplo, os fragmentos combinados presentes no exemplo "*ATM*", "*alt*" e "*opt*", quando interagem diretamente com o utilizador, são codificados através de blocos de instruções do Java `"if / else if / else"`. Analisando isoladamente os fragmentos combinados do exemplo "*ATM*", ilustrados em B.4, a sua codificação seria semelhante à presente no excerto de código B.7, tratando-se das codificações mais simples de realizar.

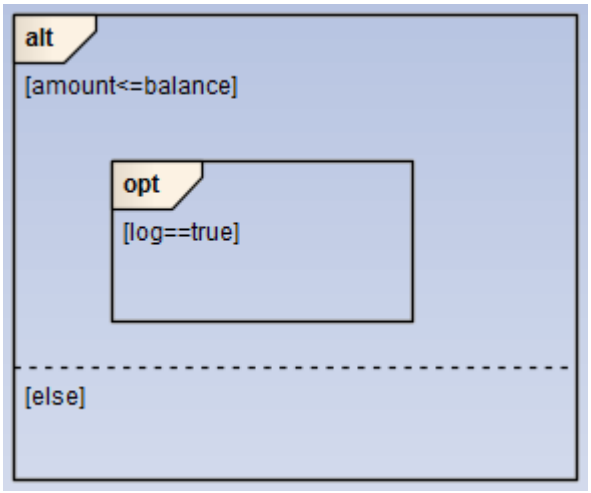


Figura B.4: Fragmentos combinados do exemplo "ATM" isolados.

```
if (amount<=balance) {
  //...
  if (log==true) {
    // ...
  }
}
else {
  // ...
}
```

Excerto de Código B.7: Codificação dos fragmentos combinados do exemplo "ATM" isolados.

Para os anteriores fragmentos codificados em interações internas e nos restantes fragmentos combinados são codificados de forma semelhante às chamadas, recorrendo a estruturas do *TracingUtilities*, idênticas à já analisada "*Call*". Por cada fragmento combinado suportado existe uma estrutura que deve ser utilizada para o codificar. Esta relação encontra-se expressa na tabela B.1, onde em cada linha é apresentado um fragmento combinado suportado pelo FEUP-SBT-2.0, o construtor da estrutura que lhe corresponde, e uma sumária descrição desse construtor.

Fragmento Combinado	**Construtor da Estrutura**	**Descrição**
Alternativa (*alt*)	• `CombAlt(int diagramObjectID, CallNode ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` as mensagens que ocorrerão para cada alternativa.
Opcional (*opt*)	• `CombOpt(int diagramObjectID, CallNode ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` o conjunto de mensagens cuja execução dependerá de uma condição.
Entrelaçado (*inter*)	• `CombInter(int diagramObjectID, Call ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` o conjunto de mensagens cuja execução pode ser entrelaçada.
Ciclo (*loop*)	• `CombLoop(int min, int max, CallNode arg)` • `CombLoop(int numIter, CallNode arg)` • `CombLoop(CallNode arg)`	Existem 3 construtores de ciclos (*loop*). É comum a todos o último argumento `arg`, onde deve ser colocado o conjunto de chamadas a operar em cada iteração do ciclo. O primeiro construtor, codifica um ciclo de `min` até `max` iterações, enquanto que no segundo, um ciclo de `numIter` iterações. No último construtor, é codificado um ciclo sem restrições do número de iterações.
Paralelo (*par*)	• `CombPar(int diagramObjectID, CallNode ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` o conjunto de mensagens cuja execução pode acontecer em paralelo.

Permutável (*perm*)	• `CombPerm(int diagramObjectID, CallNode ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` o conjunto de mensagens cuja execução pode ser por qualquer ordem.
Sequencial (*seq*)	• `CombSeq(int diagramObjectID, CallNode ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` o conjunto de mensagens que devem ocorrer de forma sequencial.
Estrito (*strict*)	• `CombStrict(int diagramObjectID, CallNode ... args)`	`diagramObjectID` representa o identificador do fragmento no Enterprise Architect e `args` o conjunto de mensagens que devem obrigatoriamente ocorrer pela ordem indicada.

Tabela B.1: Correspondência entre fragmentos combinados e respetivas estruturas do *TracingUtilities*

B.3.3 Definição das interações previstas

Apresentadas as estruturas básicas para codificação dos principais elementos de um diagrama de sequência, mensagens e fragmentos combinados, é de seguida apresentada a forma como é possível estruturar as interações internas que se espera ocorrerem no código de teste. Este processo é realizado através da classe "*Trace*" pertencente à biblioteca *TracingUtilities* e a sua forma de utilização está apresentada em B.8. Como é visível, dentro do método "`expect`" devem ser inseridas as chamadas que compõe o conjunto de interações internas previstas. De seguida deve ser codificada a mensagem principal que desencadeia as interações referidas e por fim, invocado o método "`finalCheck`". Para o exemplo apresentado em B.3, o código completo que lhe corresponde está registado no excerto de código B.9.

```
1 Trace.expect(
    // Interacoes internas esperadas
3 );
  // Chamada principal
5 Trace.finalCheck();
```

Excerto de Código B.8: Esqueleto da codificação das interações internas esperadas.

```
Trace.expect(
  new Call(6, "ATM.Account", a, "withdraw", new Object[] {amount}, 9, "OK",
    new Call(8, "ATM.Account", a, "setBalance", new Object[] {balance-amount},
        -10, null)
  )
);
assertEquals("|DiagramMessageID=9|", "|RetID=6|", "OK", a.withdraw(amount));
Trace.finalCheck();
```

Excerto de Código B.9: Codificação completa do trecho da figura B.3.

B.3.4 Manipulador de objetos

Ao definir as interações internas previstas, por vezes é necessário utilizar objetos que são criados durante essas interações e, como tal, ainda não estão definidos. Para lidar com estas situações que iriam originar erros de compilação, foi definida a classe genérica "*ObjectHandler*" que permite criar uma referência para o objeto em questão, permitindo utilizá-lo na definição das interações, sendo depois atribuído o seu real valor.

Caso se pretendesse codificar o construtor da figura B.1 como uma interação interna e, consequentemente, utilizando a classe "*Trace*", o objeto "*a*" do tipo "*Account*" teria de ser definido previamente com recurso ao manipulador de objetos, estando apresentado em B.10 o código deste processo.

```
{
  ObjectHandler<Account> a = new ObjectHandler<Account>();
  Trace.expect(new Call(5, "ATM.Account", a, "Account", new Object[] {balance
      }, -10, null));
}
Account a = new Account(balance);
Trace.finalCheck();
```

Excerto de Código B.10: Utilização do manipulador de objetos "*ObjectHandler*".

Apresentados os detalhes principais das estruturas presentes no código de teste interpretável pela biblioteca *TracingUtilities*, é apresentado no excerto de código B.11 o código de teste completo gerado pelo FEUP-SBT-2.0 para o exemplo "*ATM*" que tem vindo a ser utilizado na explicação.

```
package ATM;

import traceutils.*;

public class ATMTest extends InteracTestCase {

```

```
  public void testATM(double balance, double amount, boolean log, String date,
      String type)  throws Exception {
    Account a = new Account(balance);
    if (amount<=balance) {
      Trace.expect(
        new Call(6, "ATM.Account", a, "withdraw",  new Object[] {amount}, 9, "
            OK",
          new Call(8, "ATM.Account", a, "setBalance",  new Object[] {balance-
              amount}, -10, null)));
      assertEquals("|DiagramMessageID=9|", "|RetID=6|", "OK", a.withdraw(amount
          ));
      Trace.finalCheck();
      if (log==true) {
        Movement m = new Movement(date, amount, type);
      }
    }
    else {
      assertEquals("|DiagramMessageID=10|", "|RetID=35|", "FAIL", a.withdraw(
          amount));
    }
  }

  public void testATM_0() throws Exception {
    testATM(100, 150, true, "01-01-01", "NoMoney");
  }

  public void testATM_1() throws Exception {
    testATM(100, 50, false, "01-01-90", "Money");
  }

}
```

Excerto de Código B.11: Codificação do exemplo "*ATM*".

B.4 Interação com o utilizador

Para modelar a interação com o utilizador, é utilizada a classe "*Console*" da biblioteca *TracingUtilities*, que contém uma série de métodos que se assemelham aos utilizados em ambientes de teste de interface para o utilizador, como é exemplo o FIT [MC05]. Estão assim modelados os comandos: `start` para iniciar o método `main` da classe assinalada numa *thread* auxiliar; *enter* para o utilizador introduzir o valor especificado pelo "*standard input*"; `display` para a aplicação apresentar o valor especificado pelo "*standard output*"; `check` para verificar se o valor apresentado é o esperado; `stop` para fechar o simulador de consola e esperar que a aplicação termine a sua execução.

De seguida é apresentado o exemplo de um modelo simplificado contendo interações com o utilizador, na figura B.5. A codificação respetiva para este exemplo é apresentada no excerto de código B.12.

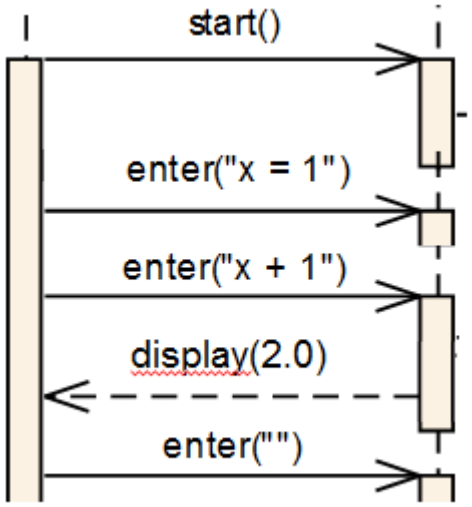


Figura B.5: Exemplo de interação com o utilizador.

```
public void testCommandLineInterface {
    Console.start(SpreadsheetApp.class, null);
    Console.enter("x = 1");
    Console.enter("x + 1");
    assertEquals(2.0, Console.check());
    Console.enter("");
    Console.stop();
}
```

Excerto de Código B.12: Codificação do exemplo de interação com o utilizador presente em B.5.

B.5 Exceções

Não existindo definida uma forma normalizada de modelar exceções em diagramas de sequência, a solução adotada para modelar este estrutura foi através das mensagens de retorno. Tomando como exemplo a figura B.6, nela está modelada uma mensagem `msg` onde deve ser lançada uma exceção `ExampleException`. No excerto de código B.13, encontra-se o código correspondente para o exemplo dado.

```
try {
  msg();
```

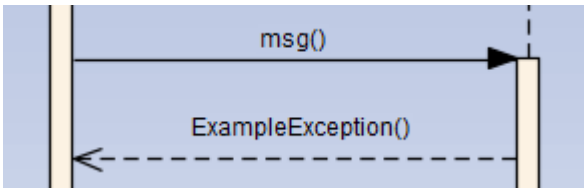


Figura B.6: Modelação de uma exceção em UML.

```
    fail("Should have thrown ExampleException");
}
catch(ExampleException exception) {
}
```

Excerto de Código B.13: Codificação de um exemplo de exceção.

B.6 Conformidade

Existem dois tipos de análise de conformidade da árvore de chamada: estrita (*strict conformance*) e alargada (*loose conformance*). Por pré-definição, todos os exemplos são tratados com análise de conformidade alargada, nos quais não são interpretados como erros possíveis chamadas adicionais às previstas. Caso o utilizador pretenda habilitar a análise de conformidade estrita, é necessário no início de cada método de teste incluir a seguinte instrução: `Trace.setStrictConformance()`, não permitindo assim que nesse exemplo sejam executadas chamadas não previstas.

B.7 Alterações na análise de cobertura

O código gerado a partir do modo de funcionamento "Executar os Testes Analisando Cobertura" tem ligeiras diferenças em relação à geração básica já apresentada. A primeira alteração é a exigência de inclusão no início de cada método de teste da seguinte instrução `Trace.coveringMode = true;`, para alertar a biblioteca auxiliar deste modo de execução. Além deste aspeto, todas as mensagens devem ser codificadas através da classe `Trace` conforme é apresentado no exemplo B.10. Esta obrigação deve-se à necessidade de informar a biblioteca auxiliar de que se está à espera (`Trace.expect`) de todas as mensagens presentes no diagrama, para posteriormente conferir se todas elas foram executadas. Por último, caso sejam utilizadas exceções, dentro do bloco `catch` respetivo, deve ser introduzida a seguinte instrução: `Trace.coveredMessages.add(<id>);` onde `<id>` deve ser substituído pelo identificador da mensagem de retorno que representa a exceção a lançar.

Interpretação do código de teste gerado

Anexo C

Metodologia

No desenvolvimento deste trabalho de dissertação foi adotada uma abordagem iterativa, na qual eram realizadas todas as semanas reuniões com o orientador de dissertação, nas quais eram definidos os requisitos a implementar na iteração seguinte e eram avaliados e validados os requisitos implementados na iteração anterior. No fim de cada iteração era suposto ter disponível uma versão executável do protótipo contendo as implementações definidas, validando assim o rumo tomado pelo processo de desenvolvimento.

De seguida é apresentada uma lista resumida das principais tarefas realizadas em cada uma das iterações definidas ao longo do trabalho. Para cada iteração é ainda indicado o intervalo de tempo a que esta correspondeu.

- *Iteração 1 (17/09/2012 a 21/09/2012)*
  - Gerar, compilar e executar código de teste, apresentando resultado numa janela independente.
- *Iteração 2 (24/09/2012 a 28/09/2012)*
  - Incluir identificadores de elemento do Enterprise Architect no código de teste gerado e na biblioteca *TracingUtilities*.
  - Mapear falhas de testes de API com a respetiva mensagem no modelo UML.
- *Iteração 3 (01/10/2012 a 05/10/2012)*
  - Mapear falhas nas interações com o utilizador.
- *Iteração 4 (08/10/2012 a 12/10/2012)*
  - Mapear falhas nas interações internas, recolhendo os vários tipos de erros.
- *Iteração 5 (15/10/2012 a 20/10/2012)*
  - Melhorar as mensagens de erro.
  - Colorir cada linha dos parâmetros de teste conforme o resultado.

  - Colorir fragmento combinado *par* caso nenhuma das mensagens seja executada.

- *Iteração 6 (22/10/2012 a 27/10/2012)*
  - Colocar informação resumida nas notas das mensagens com erro.
  - Criar estrutura para carregamento de informação do modelo (*ProjectHierarchyNode*).
- *Iteração 7 (29/10/2012 a 02/11/2012)*
  - Definir e editar os diferentes estereótipos.
  - Criar o módulo de configurações.
- *Iteração 8 (05/11/2012 a 09/11/2012)*
  - Permitir a seleção de excerto de modelo a executar.
- *Iteração 9 (12/11/2012 a 16/11/2012)*
  - Re-estruturar janela do módulo de configurações e especificidades do mesmo.
- *Iteração 10 (19/11/2012 a 23/11/2012)*
  - Corrigir vários erros encontrados após teste mais profundo.
  - Verificar possibilidade de desenvolvimento nova funcionalidade (Análise de cobertura)
- *Iteração 11 (26/11/2012 a 30/11/2012)*
  - Implementar funcionalidade de Análise de Cobertura em casos simples.
- *Iteração 12 (03/12/2012 a 07/12/2012)*
  - Adaptar a solução implementada para Análise de Cobertura para casos mais complexos.
- *Iteração 13 (10/12/2012 a 14/12/2012)*
  - Alterar a funcionalidade Análise de Cobertura isolada para incluir erros de execução.
  - Montagem de exemplo para cobertura dos vários aspetos a testar (Exemplo *ATM*).
- *Iteração 14 (17/12/2012 a 21/12/2012)*
  - Corrigir erros e validar o exemplo "ATM".
- *Iteração 15 (02/01/2013 a 21/01/2013)*
  - Validar num sistema real/existente.
  - Elaborar Relatório Final e Resumos.
- *Iteração 16 (22/01/2013 a 15/02/2013)*
  - Elaborar Artigo Científico e Website para disponibilização do FEUP-SBT-2.0.

Referências

[Alh99] Sinan Si Alhir. Understanding the unified modeling language (uml). *Methods and Tools*, 1999. URL: http://www.methodsandtools.com/archive/archive.php?id=76.

[API12] JUnit API. Testcase (junit api), 2012. Disponível em http://junit.sourceforge.net/junit3.8.1/javadoc/junit/framework/TestCase.html, acedido a última vez em 22 de Julho de 2012.

[Asp12] AspectJ. The aspectj project, 2012. Disponível em http://www.eclipse.org/aspectj/, acedido a última vez em 22 de Julho de 2012.

[Bec02] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, Novembro 2002.

[Bia06] Luciano Bathaglini Biasi. Geração automatizada de drivers e stubs de teste para junit a partir de especificações u2tp. Master's thesis, Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul, 2006.

[Ecl12] Eclipse. Eclipse - the eclipse foundation open source community website., 2012. Disponível em http://www.eclipse.org/, acedido a última vez em 22 de Julho de 2012.

[FL02] Falk Fraikin e Thomas Leonhardt. Seditec - testing based on sequence diagrams. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, 2002.

[Foc12] Micro Focus. Together - visual modeling for enterprise software applications, 2012. Disponível em http://www.microfocus.com/products/micro-focus-developer/together/index.aspx, acedido a última vez em 22 de Julho de 2012.

[FPY12] João Pascoal Faria, Ana Paiva e Zhuanli Yang. Test generation from uml sequence diagrams. In *8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*, 2012. URL: http://paginas.fe.up.pt/~jpf/research/TR-LSDBT-2012-01.pdf.

[Fra12a] Eclipse Modeling Framework. Eclipse modeling - emf, 2012. Disponível em http://www.eclipse.org/modeling/emf/, acedido a última vez em 22 de Julho de 2012.

[Fra12b] .NET Framework. .net development, 2012. Disponível em http://msdn.microsoft.com/en-us/library/ff361664.aspx, acedido a última vez em 18 de Dezembro de 2012.

[FS99] Martin Fowler e Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Agosto 1999.

[Hum05] Watts S. Humphrey. *PSP(sm): A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, Março 2005.

[Jav12] Enterprise JavaBeans. Oracle - enterprise javabeans technology, 2012. Disponível em http://www.oracle.com/technetwork/java/javaee/ejb/index.html, acedido a última vez em 22 de Julho de 2012.

[JSW07] A. Z. Javed, P. A. Strooper e G. N. Watson. Automated generation of test cases using model-driven architecture. In *2nd International Workshop on Automation of Software Test (AST '07)*, 2007.

[JUn12] JUnit. Junit.org, 2012. Disponível em http://www.junit.org/, acedido a última vez em 22 de Julho de 2012.

[MC05] Rick Mugridge e Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, Julho 2005.

[MCF03] Stephen J. Mellor, Anthony N. Clark e Takao Futagami. Model-driven development. *IEEE Software Magazine*, 20(5):14–18, Setembro/Outubro 2003. URL: http://ngis.computer.org/csdl/mags/so/2003/05/s5014.html.

[Mic12] Microsoft. Microsoft - devices and services, 2012. Disponível em http://www.microsoft.com/en-us/default.aspx, acedido a última vez em 18 de Dezembro de 2012.

[MOF12] MOFScript. Mofscript, 2012. Disponível em http://www.eclipse.org/gmt/mofscript/, acedido a última vez em 22 de Julho de 2012.

[MSUW02] Stephen J. Mellor, Kendall Scott, Axel Uhl e Dirk Weise. Model-driven architecture. In *8th International Conference on Object-Oriented Information Systems (OOIS '02)*, 2002.

[Net12] Microsoft Developer Network. Assembly registration tool (regasm.exe), 2012. Disponível em http://msdn.microsoft.com/pt-pt/library/tzat5yw6(v=vs.80).aspx, acedido a última vez em 18 de Dezembro de 2012.

[Net13] MSDN Microsoft Developer Network. Windows forms, 2013. Disponível em http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx, acedido a última vez em 7 de Janeiro de 2013.

[OMG11] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*, Agosto 2011.

[Ora13] Oracle. Java se (standard edition), 2013. Disponível em http://www.oracle.com/technetwork/java/javase/overview/index.html, acedido a última vez em 7 de Janeiro de 2013.

[Poo01] John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *15th European Conference on Object Oriented Programming (ECOOP 2001)*, 2001.

[Ros12] Rational Rose. Ibm software - rational rose, 2012. Disponível em http://www-01.ibm.com/software/awdtools/developer/rose/, acedido a última vez em 22 de Julho de 2012.

[Spa10] Sparx Systems. *Enterprise Architect Software Development Kit*, 2010.

[Sys12] Sparx Systems. Uml tools for software development and modelling - enterprise architect uml modeling tool, 2012. Disponível em http://www.sparxsystems.com.au/, acedido a última vez em 22 de Julho de 2012.

[Tef12] Tefkat. Tefkat: The emf transformation engine, 2012. Disponível em http://tefkat.sourceforge.net/, acedido a última vez em 22 de Julho de 2012.

[UL07] Mark Utting e Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, Março 2007. URL: http://books.google.pt/books?id=8hAGtY4-oOoC&lpg=PA1&hl=pt-PT&pg=PP1#v=onepage&q&f=false.

[Win12] Microsoft Windows. Using regedit.exe, 2012. Disponível em http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/tools_regeditors.mspx?mfr=true, acedido a última vez em 18 de Dezembro de 2012.

[WM07] Jeremiah Wittevrongel e Frank Maurer. Scentor: Scenario-based testing of e-business applications. In *2nd International Workshop on Automation of Software Test (AST '07)*, 2007.

[ZHM97] Hong Zhu, Patrick A. V. Hall e John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dezembro 1997.