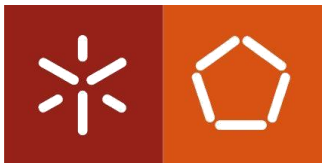


Universidade do Minho

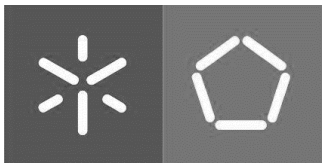
Escola de Engenharia

Departamento de Sistemas de Informação

Graciano Filipe Direito Fernandes

Geração Automática de Casos de Teste a partir de Requisitos

Guimarães, Outubro de 2014



Universidade do Minho

Escola de Engenharia

Departamento de Sistemas de Informação

Graciano Filipe Direito Fernandes

Geração Automática de Casos de Teste a partir de Requisitos

Dissertação de Mestrado

Mestrado Integrado em Engenharia e Gestão de Sistemas de Informação

Trabalho efetuado sob a orientação do

Professor Doutor Ricardo Jorge Silvério Magalhães Machado

Agradecimentos

A realização deste trabalho foi um processo duro e demorado de muito trabalho que não conseguiria realizar da mesma forma sem o contributo direto e indireto de algumas pessoas, às quais faço questão de agradecer.

Em primeiro lugar quero agradecer ao meu orientador Professor Doutor Ricardo Machado pela sua disponibilidade para me orientar neste projeto e pela sua ajuda sempre que solicitado.

Em seguida queria agradecer todas as pessoas da empresa i2S pela oportunidade de poder realizar este trabalho nessa empresa e por me terem recebido tão bem. Queria fazer um agradecimento em especial ao meu orientador na empresa Engenheiro Abílio Pinto pela sua disponibilidade e ajuda e também ao Eduardo Costa pois sem a ajuda deste tenho a certeza que o trabalho seria bem mais difícil.

Quero deixar o agradecimento mais especial de todos aos meus pais, meu irmão, minha cunhada, meus sobrinhos e minha namorada por serem os pilares da minha vida e por todo o apoio que me deram nesta etapa tão importante, porque seguramente sem eles nada disto seria possível.

Por fim queria agradecer a todos os meus amigos, àqueles que fazem parte da minha vida, desde que me lembro e aos que foi encontrando no percurso deste longo caminho, pois têm grande responsabilidade naquilo em que me tornei.

Um OBRIGADO a todos.

Resumo

O facto de vivermos num mundo cada vez mais informatizado e em que os sistemas informáticos já fazem parte do quotidiano das pessoas e das organizações levou a que os sistemas informáticos se tornassem cada vez maiores e mais complexos. Para se construir um sistema que dê resposta ao pretendido e que tenha qualidade, existe um processo de desenvolvimento que deve ser seguido. Durante o processo de desenvolvimento de *software* existem várias etapas pelas quais se tem de passar, uma dessas etapas é a de testes.

Sendo a etapa de testes uma das mais "caras" em termos de recursos e tempo no processo de desenvolvimento de *software*, a automatização de processos que compõem esta área tornou-se um dos principais desafios e interesses para as organizações.

Assim, nasceu a necessidade de se construir uma ferramenta que a partir dos requisitos especificados para um projeto de *software* conseguir-se identificar quais os casos de teste de uma forma automática, garantindo, não só, uma maior rapidez mas também uma maior qualidade no processo de identificação de casos de teste. O que consequentemente faz com que seja desenvolvido um produto de melhor qualidade. Deste modo, o tema abordado neste documento baseia-se no desenvolvimento de uma solução para um problema numa organização real.

O facto da solução abordada neste documento ser realizada para uma organização real, faz com que existam processos e abordagens utilizadas na organização com as quais se tem de trabalhar. Uma das abordagens utilizadas na organização e consequentemente utilizadas para a criação da solução descrita neste documento é o conceito de DSL (*Domain-Specific Languages*), que são linguagens criadas para um domínio especifico e as quais são utilizadas nesta solução para a especificação dos casos de teste.

Este trabalho apresenta uma contribuição para a área de testes de *software*, com a aplicação de uma solução que permita a identificação de casos de teste de uma forma automática a partir de requisitos especificados para um determinado projeto.

Palavras-chave: Testes de Software; Casos de Teste; Casos de Uso; Automatização de Testes; Especificação de Requisitos; Linguagens Especificas de Domínio; Desenvolvimento Orientado a Modelos.

Abstract

The fact that we live in a world increasingly computerized and the computer systems that are already part of everyday life of people and organizations that led to the computer systems become increasingly larger and more complex. To build a system that is responsive and has the desired quality, there is a developmental process that must be followed. During the process of software development there are various stages through which it must pass one of these steps is to test.

As the stage of testing one of the most "expensive" in terms of resources and time in the software development process, the automation of processes that make up this area has become a major challenge for organizations and interests.

So, the need to build a tool that was born from the specified requirements for a software project get to identify which test cases in an automated manner, ensuring not only greater speed but also a higher quality process identification of test cases. What therefore causes a better quality product is developed. This way, the issue addressed in this document is based on developing a solution to a problem in a real organization.

The fact that the solution discussed in this document be performed to a real organization, means that there are processes and approaches used in the organization with whom they must work. One of the approaches used in the organization and consequently used to create the solution described in this paper is the concept of DSL (Domain-Specific Languages) are languages created for a specific domain and which are used in this solution for the specification of cases test.

This work presents a contribution to the field of software testing, with the application of a solution that enables the identification of cases in an automatic way from test requirements specified for a particular project.

Key words: Software Test; Test Cases; Use Cases; Test Automation; Requirements Specification; Domain-Specific Languages; Model-Driven Development.

Índice

Índice de Figuras

Índice de Tabelas

Lista de Acrónimos

DSL – Domain Specific Language

DSR – Design Science Research

EMF – Eclipse Modeling Framework

MDD – Model-Driven Development

TI – Tecnologias de Informação

UML – Unified Modeling Language

XML – eXtensible Markup Language

Capítulo 1

Introdução

Neste capítulo é feita uma introdução relativa ao tema proposto a ser realizada para esta dissertação. No capítulo de introdução começa-se por fazer um enquadramento sobre o tema e qual motivação para o desenvolvimento deste trabalho, em seguida é apresentada os principais objetivos e resultados esperados. Por fim, é descrito qual a organização que este documento de dissertação apresenta.

1.1 Motivação e enquadramento

Historicamente, o desenvolvimento de *software* tem sido caracterizado por ser centrado no produto. No entanto, mais recentemente, investigadores e programadores têm mudado o foco dos seus esforços para a dimensão do processo. Atingir a maturidade do processo de desenvolvimento de *software* é uma meta estabelecida por diversas empresas após terem obtido resultados menos positivos devido à aplicação de novas metodologias e tecnologias de software.

Como se sabe, o processo de desenvolvimento de *software* em larga escala é extremamente complexo, pois é necessário considerar o grande número de recursos humanos, software e hardware envolvidos. Gerir estes recursos, de forma eficiente, torna-se uma tarefa quase impossível de ser realizada se o processo de desenvolvimento não estiver automatizado, ou pelo menos, parcialmente automatizado. A automatização de processos de desenvolvimento de *software* tem o potencial de melhorar significativamente a qualidade e produtividade do *software*.

É neste contexto que esta dissertação surge, na medida que o seu foco é pretender automatizar determinados passos do processo de desenvolvimento de *software*, tornando as soluções mais robustas, com maior qualidade e reduzindo o tempo de resposta para com os clientes.

Um dos pontos onde a automatização é desejada, e atualmente ainda não foi encontrada uma solução satisfatória, é na realização dos testes de *software*. Há alguns anos os testes eram feitos apenas de forma manual, mas com o passar dos tempos foram-se criando ferramentas automáticas de teste. Mas o facto de aparecerem essas ferramentas, não fez com que a

percentagem de erros encontrados aumentasse, antes pelo contrário. Essas baixas percentagens devem-se ao facto de, muita das vezes, os testes apenas serem realizados no final do processo de *software* não cobrindo assim o ciclo de vida total do desenvolvimento de software. Refira-se, também, que a execução dos testes de *software*, apesar de ser uma atividade essencial no desenvolvimento de software, é algo que consome em grande escala vários recursos de uma empresa. Para agravar esse problema, os *Tester's* apoiam-se em ferramentas rígidas, lentas e de grande complexidade, o que obriga a um moroso período de adaptação. Por outro lado, são ferramentas que não acompanharam as tendências do setor, obrigando a que se despenda demasiado tempo na elaboração de scripts de testes.

Em suma, a produção de testes é, ainda, um processo muito manual, e consequentemente pouco produtivo numa lógica de produto, em especial para os testes de regressão, isto é, testes que são feitos para garantir que o que estava bem, continua após uma mudança. O facto do processo de testes ser ainda muito manual, relacionado com a maior complexidade que cada vez mais os software têm faz com que esse processo seja cada vez mais difícil e moroso, o que aumenta o risco de um software ser feito com menor qualidade.

1.2 Objetivos

O objetivo da dissertação descrita neste documento consiste na criação de um conjunto de mecanismos de forma a automatizar o processo de testes no desenvolvimento de novos produtos por parte da i2S, que é uma empresa especializada no desenvolvimento de *software* para a área dos seguros. De forma mais específica, a realização deste trabalho consiste em automatizar parte da fase de testes no processo de desenvolvimento de *software*, gerando de forma automática cenários de teste com base em requisitos especificados na forma de casos de uso. Com a realização deste projeto, estima-se uma redução de cerca de 70% em termos de tempo no que diz respeito à execução de testes de uma forma manual. Para a realização deste projeto foi feito um estudo sobre a geração de casos de teste a partir de requisitos e de toda a atividade de teste de *software*, bem como um estudo sobre linguagens específicas de domínio e abordagens de desenvolvimento orientadas a modelos.

Assim sendo o principal objetivo deste trabalho é a identificação de forma automática de casos de teste a partir de requisitos no processo de desenvolvimento de *software*.

Para a concretização deste objetivo fez-se um estudo sobre um conjunto de fatores e problemas que tiveram de ser ultrapassados. Alguns desses fatores são:

- Estudo e análise da atividade de teste de *software*;
- Estudo e análise no que diz respeito à geração de casos de teste e a sua automatização a partir de requisitos;
- Estudo sobre linguagens específicas de domínio, bem como ferramentas que permitem a adoção desta abordagem;
- Compreender o que são linguagem especificas de domínio, como se desenvolvem este tipo de linguagens e analisar a linguagem já adotada pela empresa;
- Estudo e de análise da abordagem de desenvolvimento orientada a modelos (MDD – *Model-Driven Development*).

Existem ainda outros pontos importantíssimos na realização deste trabalho, alvos de um processo de estudo e exploração para que objetivo pretendido seja alcançado, principalmente, no que diz respeito a ferramentas e tecnologias usadas para no desenvolvimento deste projeto.

1.3 Abordagem Metodológica

Sendo a realização e escrita de uma tese de mestrado um processo complexo, e de forma a conseguir cumprir com os objetivos propostos, no âmbito da área dos Sistemas da Informação, foi estudada e analisada a obra de Berndtsson, Hansson et al. (2008) que tem como título "Thesis Project – A Guide Student in Computer Science and Information Systems".

Tendo em conta o cariz deste trabalho e o facto de este integrar a temática de projeto de investigação, a abordagem escolhida para a realização do projeto é *o Design Science Research*. Segundo Hevner, March, Park, & Ram (2004), a abordagem DSR consiste em dar orientação num projeto de Sistemas de Informação onde o objetivo seja a construção e aplicação de um ou mais artefactos que resolva um conjunto de problemas, identificando-se assim esta abordagem com o problema proposto para resolução neste projeto.

De acordo com Peffers et al. (2008), a DSR é uma metodologia de investigação na qual são respondidas questões relevantes e de nível científico, sendo essa resposta dada através da construção de artefactos inovadores. O mesmo autor defende ainda que para adquirir o

conhecimento que permite compreender e resolver um problema deste tipo, é necessário criar e aplicar artefactos, que podem ser novos, ou versões melhoradas de partes já existentes.

Através da abordagem de investigação *Design Science Research* é pretendido que se estude e perceba um determinado problema, num determinado domínio, que com a criação e aplicação de um ou mais artefactos e avaliação das suas capacidades se consiga uma obter a solução esperada.

No âmbito deste trabalho o problema consiste em gerar casos de teste de forma automática partir de requisitos, no domínio da organização i2S - Insurance Software Solutions, através da criação e integração de ferramentas que automatizem testes, sendo posteriormente a sua avaliação feita através de demonstrações.

A figura 1.1 representada a baixo mostra as seis atividades pelas quais é composta a metodologia *Design Science Rersearch* (Peffers *et al.* 2008).

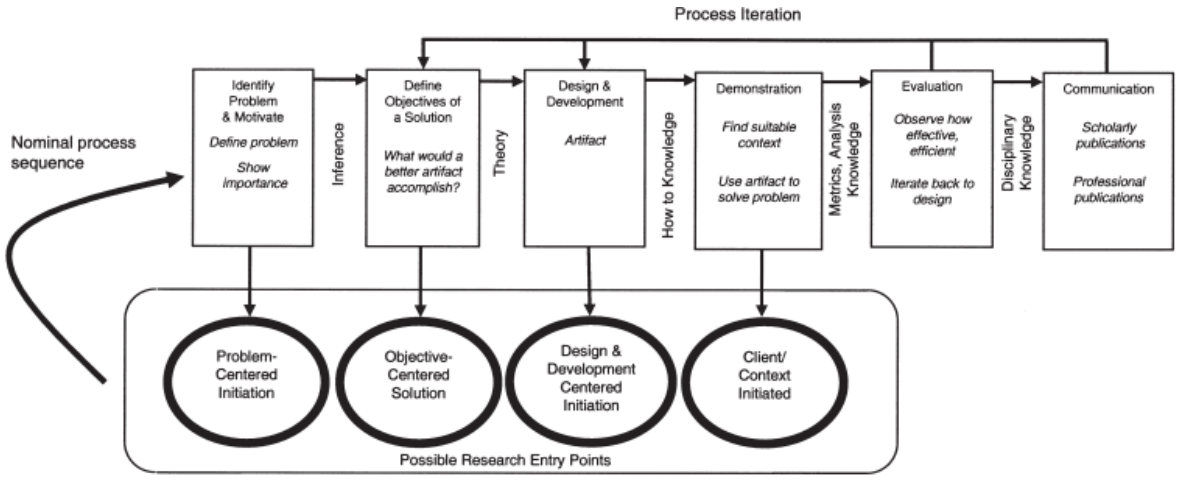


Figura 1.1 - Modelo de processos da metodologia (Peffers et al. 2008)

A primeira atividade da metodologia consiste na identificação do problema e da motivação, definindo-se a partir desta atividade a questão de pesquisa e a justificação de valor da solução pretendida para a comunidade.

Na segunda atividade define-se os objetivos da solução a partir da definição do problema, tendo em conta o conhecimento do que é possível e viável realizar. Esta definição de objetivos deve ser feita racionalmente tendo em conta o trabalho realizado na fase anterior, do conhecimento

do estado da arte e das soluções atuais existentes que possam dar resposta ao nosso problema ou a problemas semelhantes.

A fase de desenho e desenvolvimento consiste na terceira fase da metodologia e tem como objetivo a realização da atividade de conceção e implementação do artefacto produzido.

Após ser feita a implementação, procede-se à sua demonstração que é a quarta atividade da metodologia. Nesta atividade o objetivo é demonstrar que o artefacto desenvolvido tem utilidade para dar resposta às instâncias do problema identificado anteriormente. Essa demonstração pode ser realizado envolvendo a sua utilização em uma experimentação, simulação, prova de conceito ou caso de estudo.

A quinta atividade, designada por avaliação, consiste em observar e medir a qualidade do artefacto na resposta que dá ao problema a resolver. Para a realização desta atividade é necessário o conhecimento de métricas e técnicas relevantes para a análise, caso os resultados não sejam satisfatórios, poderá ter de se retroceder para a terceira atividade. Fundamentalmente nesta fase faz-se uma comparação entre os resultados daquilo que é produzido e os resultados que estavam definidos anteriormente. Esta comparação pode ser feita de várias formas dependendo do artefacto e do problema a comparar, podendo ser feitas comparações de funcionalidades com objetivos, comparações feitas entre medidas quantitativas, avaliação de métricas em experiencias controladas, entre outros tipos de comparações que se podem fazer, dependendo do artefacto, dos objetivos e do ambiente.

Por fim, a ultima atividade da metodologia corresponde à comunicação. Significa isso, que se deve comunicar o problema, qual a sua importância, qual o artefacto resultante, qual a sua utilidade e a sua eficácia para os investigadores e para outros públicos relevantes, como neste caso, para profissionais da área.

Dificuldades e riscos

No decorrer do projeto encontraram-se vários riscos e dificuldades que poderiam inviabilizar ou ameaçar o sucesso deste trabalho. Dentro de várias adversidades que surgiram no desenrolar do trabalho, identificam-se aquelas que foram mais críticas e preocupantes:

- Escolha e utilização de abordagens e tecnologias para o desenvolvimento das ferramentas de automatização que não sejam capazes de resolver o problema proposto;

- Incompreensão ou má interpretação do problema, que leva a que o trabalho realizado não vá de encontro aos resultados esperados;
- Custo envolvidos, no caso de se fazerem alterações a quando de se estar em uma avançada fase do processo de desenvolvimento;
- O tempo limitado para a realização do projeto.

No processo de desenvolvimento das ferramentas de automatização, para se definirem quais as abordagens e tecnologias utilizadas foi necessário fazer uma análise detalhada para garantir que a escolha feita foi a mais acertada e a melhor para resolver o problema. Para dar resposta a este problema e minimizar o risco de uma má escolha, fez-se uma pesquisa de informação e opiniões de trabalhos já realizados com o intuito de perceber o que poderia ser usado ou não neste trabalho.

Para diminuir o risco de ser feita uma má interpretação sobre o que realmente é o problema, fez-se fazer reuniões periódicas com o responsável do projeto e protótipos para teste com o intuito de se ter mostrar o caminho que o projeto levava e se esse caminho ia de encontro aos objetivos propostos.

A forma de prevenção para os dois últimos riscos e dificuldades descritos, é ultrapassada através de um bom planeamento de toda a atividade de desenvolvimento. O planeamento deve ser o primeiro passo do ciclo de desenvolvimento conseguindo-se assim determinar qual o caminho que se deve seguir tendo em conta o tempo disponível. A quando da realização de um projeto de caráter científico, muitas das vezes, mesmo sendo feito um planeamento, existem problemas que obrigam a uma mudança de abordagem, para que essas mudanças não remetam o projeto para o insucesso, devem ser feitos ciclos de desenvolvimento pequenos, que depois de analisados e testados, permitam seguir o projeto com a confiança em aquilo que está feito. No caso de serem encontrados problemas nesses pequenos ciclos de desenvolvimento e seja necessário mudar-se a abordagem que se está a ter, essa mudança não implicará grandes prejuízos.

Estratégia de pesquisa

De forma a melhor compreender o problema e contextualizar o âmbito onde este está inserido, fez-se um trabalho de investigação e revisão bibliográfica. Com o intuito de se obter informação, conceitos e ideias de tudo o que foi realizado até ao momento e que tenha relação com o projeto pretendido.

Para a revisão bibliográfica ser feita com qualidade, foi necessário definir palavras-chave que permitissem obter informações precisas e válidas. As palavras-chave utilizadas foram: "Teste Software"; "Casos de Teste"; "Testes Automáticos"; "Automação de Casos Teste"; "Casos de Uso"; "Desenvolvimento Orientado a Modelos" e "Linguagens Especificas de Domínio". De salientar que estes conceitos foram pesquisados em português e também em inglês.

Depois de definidas as palavras-chave, as pesquisas são feitas recorrendo a várias fontes e motores de busca, fundamentalmente:

- Google académico;
- RepositóriUM;
- Science Direct;
- Scopus;
- IEEE Xplore;
- RCAAP.

Os critérios para selecionar os documentos pesquisados através das palavras-chave baseiam-se, na leitura do *abstract* de cada documento, sendo avaliada a compatibilidade com o tema em pesquisa e com o impacto no mundo académico e científico, verificando-se a credibilidade e relevância do documento sobre o assunto em questão. Após esta primeira avalização é feita uma leitura completa do documento em questão para retirar a informação relevante para o projeto a desenvolver.

Para fazer a gestão de referências bibliográficas para este trabalho, foi utilizada a ferramenta Mendeley, que permitiu agrupar e disponibilizar as referências de uma forma mais correta e organizada.

1.4 Estrutura do Documento

O presente documento tem uma estrutura sequencial sob a forma de capítulos. Neste primeiro capítulo de introdução é apresentada uma visão sobre o enquadramento deste projeto, quais os problemas e contributos que este projeto tem e quais os objetivos que se pretendem alcançarem com o desenrolar do trabalho. É também apresentada neste capítulo a abordagem metodológica seguida para atingir os resultados esperados.

No capítulo 2 é apresentado o estado da arte, neste capítulo descreve-se aquilo que existe no âmbito do tema abordado, ficando-se assim a conhecer o que existe atualmente. A partir deste estudo consegue-se obter toda a informação atual sobre o processo de automação de testes, o que impulsiona para desenvolvimento do projeto de forma a se diferenciar do que já existe.

O capítulo 3 diz respeito à solução proposta, onde são descritas as abordagens de especificação de requisitos e a abordagem orientada a modelos necessária para a obtenção dos objetivos propostos para o trabalho. Neste capítulo é, também, explicada a forma como os requisitos são especificados na organização bem como o modelo usado para essa especificação e também a forma como é feita a derivação para a obtenção dos casos de teste pretendidos.

Capítulo 4 é o capítulo, onde é demonstrado o desenvolvimento prático do projeto. São apresentadas as tecnologias usadas e necessárias e também a forma como essas tecnologias auxiliam no processo de geração de cenários de teste de uma forma automática.

No 5º e último capítulo são feitas as conclusões sobre todo o trabalho desenvolvido. Neste último capítulo são também indicadas as limitações encontradas na realização do trabalho apresentado e são apontadas algumas propostas para trabalho futuro de forma a se poder executar de uma forma mais completa o trabalho realizado.

2 Estado da arte

2.1 Introdução

Neste capítulo de estado da arte é feita uma revisão bibliográfica sobre as principais áreas abordadas na realização deste projeto. A revisão bibliográfica feita insere-se sobre tudo em três áreas diferentes. Nessas três áreas, duas delas dizem respeito aos testes de *software* e uma às linguagens específicas de domínio (DSLs).

A primeira área de conhecimento que é focada é área do teste de *software*, onde é absorvido um conhecimento sobre técnicas e processos que são utilizadas nesse processo do ciclo de desenvolvimento de sistemas informáticos.

A outra área correspondente aos testes abordada na revisão bibliográfica diz respeito à automatização desse processo. O processo de automatização dos testes de *software* deve seguir um conjunto de abordagens e passos, podendo essas abordagens ser distintas dependendo dos testes e do ambiente para o qual esta automatização é realizada.

A outra das áreas estudadas e sobre a qual se fez um estudo aprofundado foi a de Linguagens Especificas de Domínio (DSLs). Nesta área tentou-se sobre tudo perceber qual o conceito de DSL e qual a potencialidade que este tipo de linguagens pode ter, uma vez foi construída uma DSL de testes para testes com base em requisitos.

2.2 Teste de Software

Atualmente a atividade de testes é uma das mais importantes no processo de desenvolvimento de *software*, de forma a melhorar a qualidade de um produto em desenvolvimento. A atividade de testes consiste em especificar casos de teste tendo em conta os requisitos recebidos, executar e validar os resultados obtidos.

Os testes devem começar logo a partir da fase inicial do processo de desenvolvimento, de forma a ser possível encontrar inconsistências numa fase mais prematura, facilitando assim a correção

de eventuais erros e diminuindo o custo que essa correção poderá ter, bem como encontrar melhoramentos a serem feitos (Kansomkeat & Rivepiboon, 2003).

Devido à complexidade dos sistemas, a atividade de teste de *software* tornou-se uma atividade bastante complexa. Apesar disso, geralmente, não é realizada de uma forma sistemática, devido ao facto dos sistemas desenvolvidos serem cada vez mais complexos e estarem em constante evolução. Por este motivo, a questão da automatização de casos de teste começou a assumir uma maior importância (Sharma & Mall, 2009). Pode-se gerar casos de teste automaticamente a partir de um modelo formal, garantindo-se assim que um sistema de *software* é executado de acordo com os requisitos (Binder, 1999), (Kaner, 1997).

Apesar dos especialistas estarem em consenso quanto aos ganhos que uma boa estratégia de automatização de testes traz, esta é ainda uma área na qual não se tem um grande domínio dentro da indústria de *software*. Assim sendo, esta etapa é muitas vezes negligenciada, atuando-se muitas vezes sem se ter uma boa definição dos objetivos e das expectativas e sem a aplicação das técnicas mais apropriadas. Constando-se assim, um elevado número de casos de insucesso no que diz respeito à automatização de casos de teste (Bach, 1999) (Fewster, 2001) (Dustin, 1999).

2.2.1 Processo de teste

O objetivo num processo de desenvolvimento de *software* é construir um software com o menor número de defeitos possível e que responda de modo a satisfazer o cliente. De maneira a se conseguir detetar defeitos o mais cedo possível, faz-se uma verificação ao fim de cada fase do processo e uma avaliação de forma a garantir a qualidade do *software* a produzir. Sendo, que normalmente esta atividade é feita através de uma avaliação humana, muita das vezes está sujeita a falhas. Assim, como a fase de testes é a ultima no ciclo de vida de desenvolvimento, essa fase tem um elevado grau de importância, pois é aqui que se faz a ultima revisão de forma a se garantir a qualidade do *software*.

No processo de desenvolvimento de *software* são realizadas várias iterações, e o desenvolvimento de uma iteração pode influenciar iterações que já tenham sido realizadas e outras que ainda se vão realizar. É necessário por isso testar cada iteração de forma a não existirem inconsistências que afetem o que já foi feito ou que ainda se vai fazer.

Níveis de teste

O teste tem como objetivo detetar defeitos ao longo de todo processo de desenvolvimento de *software*. Para que isso aconteça são usados diferentes níveis de testes para cada fase do processo, destinando-se cada nível de teste a testar diferentes aspetos do sistema. Os níveis pelos quais os testes estão divididos são por teste unitário, teste de integração, teste de sistema e teste de aceitação. Estes níveis diferentes de teste têm objetivos diferentes no que diz respeito ao tipo de inconsistências que se pretendem encontrar.

- **Teste Unitário**
    - Teste Unitário: é um tipo de teste onde são testadas pequenas unidades do *software* desenvolvido, como pequenos excertos de código. Tem se o objetivo com este tipo de teste encontrar falhas de funcionamento numa pequena parte do sistema. Este teste normalmente é feito pelo programador do excerto de código. Depois dessas pequenas unidades serem testadas e não serem detetados erros, estas são consideradas para a fase de integração.
- **Teste de Integração**
    - Teste de Integração: este teste tem como objetivo encontrar falhas que provêm da integração, normalmente as falhas encontradas dizem respeito à transmissão de dados, ou a diferentes componentes que quando combinados, funcionam corretamente.
- **Teste de Sistema**
    - Teste de sistema: o teste de sistema consiste em testar o sistema por inteiro, procurando falhas tendo em conta os objetivos iniciais.
- **Teste de Aceitação**
    - Teste de aceitação: este teste é o ultimo a ser realizado, e tem como objetivo verificar se o sistema responde com qualidade aos requisitos definidos inicialmente e às necessidades do utilizador.
- **Teste de regressão**
    - Teste de regressão: como muitas das vezes são feitas mudanças no sistema depois de este estar concluído, os testes de regressão têm como objetivo garantir que essas mudanças não influenciaram o funcionamento do que foi

feito anteriormente. Assim, o teste de regressão tem como objetivo validar que tudo o que foi feito para trás não foi alterado e que as alterações feitas estão sem erros. No teste de regressão muitas das vezes são executados os casos de teste antigos, e depois dessa execução os resultados novos são comparados com os antigos, esperando-se que os resultados sejam iguais.

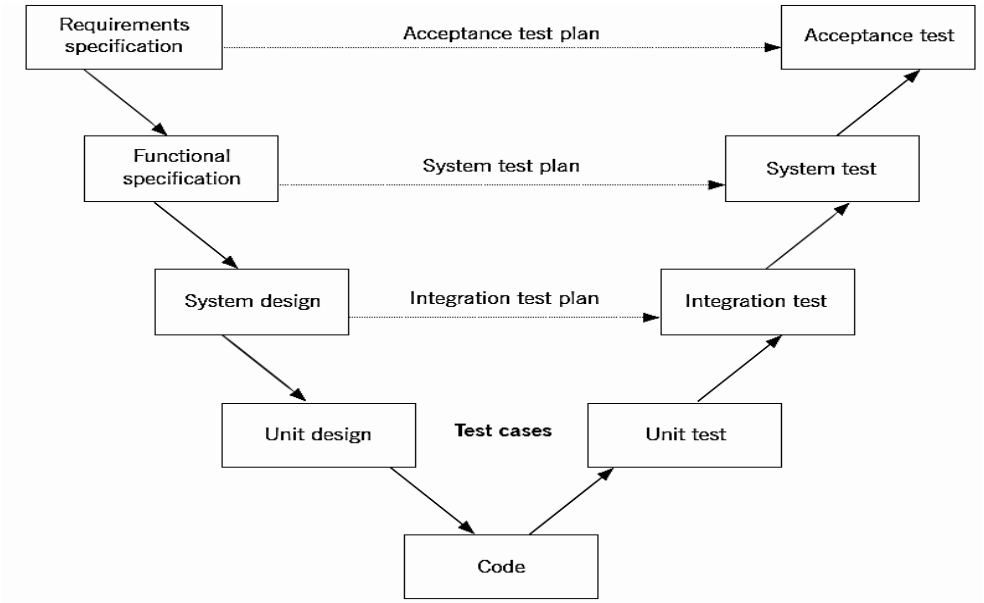


Figura 2.1 – Teste de Software, V-Model (Marick, 1999)

Plano de teste

A fase de teste está dividida em várias etapas. Normalmente essa fase começa com o plano de teste e termina com a etapa de aceitação do teste. O plano de teste consiste num documento para todo o projeto onde é definido o âmbito, a abordagem a ser tomada, agendamento dos testes, os itens de teste para todo o processo e as pessoas responsáveis por cada atividade. O planeamento da fase de testes pode ser feita antes do início do teste e em paralelo com a fase de codificação e conceção. Um plano de teste é elaborado tendo em conta o plano do projeto, documentação dos requisitos e documentação da conceção do sistema. O plano de projeto deve ter em conta se o plano de teste corresponde com o plano de qualidade determinado para o projeto e se o programa de testes corresponde ao plano de projeto. Os documentos de requisitos e de conceção são usados para se determinar quais as unidades de teste a serem feitas e quais as abordagens a serem tomadas durante a fase de testes.

Um plano de testes deve conter uma especificação da unidade de teste, recursos a serem testados, abordagens para testes, testes a entregar e agendamento e atribuição das tarefas.

Especificações de caso de teste

Para cada unidade de teste especificada no plano de testes tem de se fazer uma especificação de casos de teste separadamente. Tendo como base as técnicas e abordagens escolhidas para a fase de testes é feita uma especificação para casos de teste de cada unidade. As especificações definem todos os casos de teste a serem realizados, quais as entradas a serem usadas nos casos de teste, quais as condições a serem testadas e quais os resultados esperados para cada caso. A especificação de casos de teste é uma atividade muito importante no processo de teste. Pois se as especificações tiverem qualidade e derem resposta ao que realmente precisa de ser testado, isto faz com que o caso de teste construído tenha uma elevada qualidade. O facto de um caso de teste ter uma elevada qualidade garante que o *software* desenvolvido também é um produto de maior qualidade.

A validação de um caso de teste é feita através de uma revisão do caso de teste. O documento de especificação de caso de teste é revisto de uma forma formal com o objetivo de se conseguir certificar que todos os casos de teste são consistentes com o especificado no plano de teste.

A especificação de casos de teste também traz vantagens na escolha de um bom conjunto de casos de teste, quando é necessário fazer-se um teste de regressão, as especificações podem ser utilizadas como scripts, principalmente se o teste de regressão é feito manualmente.

Execução e análise de casos de teste

Na fase de especificação de casos de teste apenas é especificado o conjunto de casos de teste para cada unidade a ser testada. Para a execução de casos de teste pode ser necessário a criação de módulos de controlo. Os resultados obtidos da execução de casos de teste para cada unidade testada, determinam, depois de analisados, se o teste foi satisfatório ou não. Normalmente estes resultados são expressos na forma de relatórios, um relatório com a síntese do teste e um relatório de erros. O relatório com a síntese do teste é facultado para a parte de gestão de projeto, pois contem todos os passos sobre a execução do caso de teste, como por exemplo quantos casos de teste foram executados, quais os erros encontrados, quais os resultados obtidos, entre

outros. O relatório de erros, contem os detalhes de todos os erros encontrados, sendo que neste relatório os erros encontrados podem ser divididos por categorias, de forma a facilitar a análise e resolução de cada problema encontrado. A monitorização feita na fase de testes é bastante importante e deve ser realizada de uma forma cuidadosa, uma vez que nesta fase é despendido um grande esforço e custo, e é uma fase fulcral para garantir um produto de qualidade. A monitorização da fase de testes pode ser feita com recurso a métricas, como por exemplo, indicadores de esforço que são comparados com indicadores do mesmo tipo mas de projetos anteriores, permitem saber se o esforço num teste é o suficiente para que o teste seja realizado com qualidade.

Registo e rastreamento de defeitos

Em projetos de larga escala podem ser detetados muitos defeitos, que são encontrados por diferentes pessoas e em diferentes fases. Nem sempre quem encontra um defeito é a pessoa que o resolve, isto leva a que relatórios de defeitos tenham de ser feitos de uma forma séria e formal, no caso de esses relatórios serem feitos de forma negligente o tempo e esforço de resolução de um defeito pode acrescer de forma significativa. Assim, e de forma a se evitarem tais casos, o registo e rastreamento de defeitos ganha um peso considerável como uma das boas práticas na gestão de projetos.

De forma a simplificar o registo e rastreamento de testes, existem ferramentas onde a informação pode ser inserida e disponibilizada. Um defeito pode ser encontrado por qualquer pessoa a qualquer momento, depois de o defeito ser encontrado, este é registado num sistema de controlo de defeitos, a esse registo são anexadas informações sobre esse defeito, sendo este depois submetido. A tarefa de reparar o defeito é depois atribuída a uma pessoa, e esta é responsável pela análise e correção do defeito, passando este para o estado de resolvido. Depois de o defeito ser concertado, este, passa por testes, que normalmente são feitos por outra pessoa e só depois é dado como fechado. Pode-se afirmar assim, que um defeito tem um ciclo de vida que passa por três estados, submetido, resolvido e fechado.

Para facilitar a resolução de defeitos, estes podem ser gravados e catalogados por categorias. Existem diferentes formas de classificar defeitos, podendo ser classificados por defeitos de interface, funcional, ligação ou algoritmo, entre outros. Os defeitos podem ser ainda categorizados pela sua gravidade, podendo-se assim, se um defeito for classificado como grave,

reunir esforços para o resolver o mais rápido possível. Em engenharia de *software* na maioria das vezes as categorias de gravidade de um defeito são quatro: critica, maior, menor e superficial.

2.2.2 Tipos de teste

Segundo Molinari (2005), existem muitos tipos de teste, estes ganham novas classificações devido ao facto de existirem constantes mudanças de definição e execução do teste. Mas de uma forma geral pode-se definir os principais tipos de teste como:

- **Teste funcional**
  - O tipo de testes funcionais são aqueles onde são testadas as funcionalidades tendo em conta os requisitos do sistema. Este tipo de testes, verificam se as funcionalidades correspondem aquilo que foi definido no início do processo.
- **Teste não funcional**
  - Testes não funcionais são testes onde são avaliados aspetos que não têm em nada a ver com a funcionalidade do sistema. Aqui são avaliados aspetos como a eficiência, usabilidade, portabilidade e etc.
- **Teste estrutural**
  - O teste estrutural tem como objetivo avaliar o comportamento interno do sistema, normalmente consiste em avaliar o código, de forma a testar se está bem construído.

2.2.3 Fases da atividade de teste

A fase de teste é composta por atividades que são cruciais para o sucesso de implementação de um projeto de *software*. Esta etapa é bastante importante, pois se for uma etapa bem conseguida garante um bom teste ao sistema, o que garante que o sistema não tem erros e que se adequa tendo em conta os requisitos definidos no início do projeto. Em baixo são enumeradas as etapas, ou atividades segundo o SWEBOK (2004) com as respetivas definições (Kaner *et.al*, 1999):

- **Planeamento**
  - A fase de planeamento de teste consiste na coordenação das pessoas que vão atuar sobre esse processo e na gestão de infraestruturas e equipamentos

disponíveis para a realização dos testes. Nesta fase faz-se também o planeamento em relação ao tempo e esforço necessário para que o ambiente de teste esteja adequado ao projeto.

- **Geração de casos de teste**
    - A fase de geração de casos de teste baseia-se em definir qual o nível de teste a ser executado e quais as técnicas particulares do teste. Os casos de teste devem estar de acordo com as configurações pretendidas e devem incluir quais os resultados esperados para cada teste.
- **Desenvolvimento do ambiente de teste**
    - O ambiente utilizado para a realização do teste deve ter em conta a compatibilidade com as ferramentas existentes. Este ambiente deve facilitar o desenvolvimento dos casos de teste, bem como facilitar o registo e recuperação dos resultados esperados, scripts e outras componentes que envolvam o teste.
- **Execução**
    - A execução de testes deve conter documentados todos os passos feitos durante o teste, para que qualquer pessoa possa replicar os resultados, sendo que para tal, os testes devem ser realizados em conformidade com o que está nos documentos e deve ainda ser bem especificado qual a fase do *software* em teste.
- **Avaliação dos resultados de teste**
    - Nesta fase são avaliados os resultados do teste de forma a determinar o sucesso do teste. Caso não haja resultados inesperados, significa que o *software* tem o desempenho esperado, caso sejam encontradas falhas é necessário fazer-se uma análise sobre as mesmas, para se identificar qual o problema e qual a melhor forma para o corrigir (Poston, 1996).
- **Reportar problemas**
    - A atividade de teste pode ser registada de forma a se conseguir identificar quando é que um teste foi realizado, qual as partes abrangidas pelo teste, que configurações foram a base para o teste e outras informações relevantes. Os resultados de testes incorretos podem ser gravados para que possam ser acedidos mais tarde e corrigidos.
- **Rastreamento de erros**

- Falhas observadas durante os testes na maior parte das vezes estão relacionadas com falhas ou defeitos no *software*. Estas falhas podem ser analisadas de forma a se identificar qual o tipo de erro que originou determinado defeito e quando é que o erro foi introduzido. A informação gerada a partir desta fase é usada para observar quais os aspetos da engenharia de *software* que precisam ser melhorados e determina se a fase de análise e teste foram eficazes.

2.2.4 Técnicas para a geração de testes

Tabela de decisão

Tabelas de decisão são uma ferramenta onde são especificados processos que definem quais as ações que devem ser executadas, segundo um conjunto de condições. Uma tabela de decisão é construída tendo em conta uma área de condições, uma área de ações e segundo regras de decisão. Na área de condições são descritas quais as condições que devem ser verificadas para determinar as ações a serem tomadas. Na área de ações é mostrado qual as ações a serem realizados tendo em conta as condições que ocorreram. Por último as regras de decisão é onde são representadas as relações entre as condições e ações segundo uma regra definida. No âmbito de testes de *software* esta técnica tem utilidade, porque o testador pode explorar quais os efeitos de combinações entre os inputs e outros estados do software, garantindo assim uma correta implementação das regras de negócio.

Árvores de decisão

Árvores de decisão são diagramas em que se consegue identificar todas as possibilidades lógicas de uma sequência de decisões. Através de um esquema, as árvores de decisão mostram todas as ações possíveis e alternativas ao longo de um projeto. Uma árvore de decisão é desenhada na horizontal, definindo-se um ponto onde começa a árvore, em seguida esse ponto é ramificado tendo em conta o número de condições possíveis, depois de as condições serem expressas podem ser ainda ramificadas caso haja novas condições que derivem dessas, repetindo-se este processo até não existirem mais condições. Para a fase de testes de *software* as árvores de decisão têm quase o mesmo impacto que as tabelas de decisão, com a diferença de, dependendo do caso, facilitar a interpretação.

Teste de regressão

O teste de regressão faz com que seja testado todo o sistema sempre que é adicionada uma característica. Este tipo de testes é recomendado que sejam automatizados, de forma a ganhar-se tempo e não ter de se voltar a fazer todos os ciclos de testes de novo. Sempre que é adicionado um novo componente ao sistema, o sistema é modificado daí se fazer este tipo de teste para garantir que as fases já testadas não foram alteradas, garantindo-se assim que o sistema não fica com novos erros (Maia et al., 2009).

2.2.5 Abordagens de testes

Para Howden (1987), a etapa de testes de *software* é classificada em duas formas Em testes baseados no programa (*program-based testing*) ou testes estruturais e testes baseados nas especificações (*specification-based test*) ou testes funcionais. Estes dois tipos de teste também são conhecidos por teste caixa branca (*white-box test)* e teste caixa preta (*black-box test*). Nos testes de caixa branca a estrutura interna do programa, ou seja, o seu código é o alvo da execução dos casos de teste, conseguindo-se assim definir os caminhos possíveis no código do programa, dependendo da linguagem de programação utilizada. Os testes de caixa preta, ou testes funcionais, são elaborados com base na especificação do comportamento do *software*, este teste é independente do paradigma e da linguagem de programação usados, e geralmente é escrito numa linguagem natural, o que pode levantar questões de ambiguidade e de interpretação.

Testes de caixa preta são testes que têm como objetivo testar se o resultado de um *software* responde aos requisitos especificados para esse software. Nestes testes não se tem como objetivo testar como o programa foi feito, apenas se verifica se a resposta é a esperada. Nos testes de caixa preta são testados todos os requisitos funcionais do programa, tentando encontrar se funções incorretas, erros de interface, erros de performance ou erros de estrutura de dados. Para Pressman (2002) o primeiro passo para realização de testes caixa preta é compreender quais os módulos a serem testados, para que se consiga dar uma resposta aos requisitos esperados e em seguida, definir casos de teste que mostrem se a resposta aos requisitos especificados é a correta ou não.

Segundo Peters & Pedrycz (2001), o teste de caixa branca permite testar a estrutura interna do *software*. Este tipo de testes concentra-se em testar o código que compõe o *software*, com o objetivo de testar a estrutura e lógica do código produzido. Com as técnicas de testes caixa branca podem ser criados casos de teste de forma a se conseguir um controlo de fluxo de dados, testar caminhos independentes, testar todas as decisões lógicas e testar todos os ciclos dentro dos seus limites e intervalos. Conseguindo-se assim um produto sem erros e com uma maior qualidade (Pressman 2002).

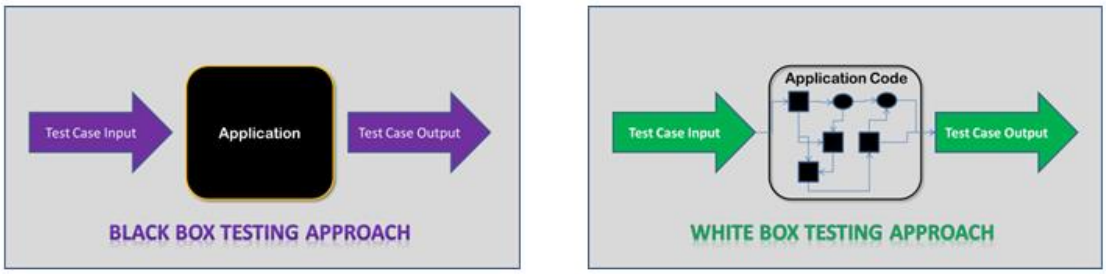


Figura 2.2 - Abordagens de testes Black Box e White Box (*in* Testing Whiz, 2011)

2.3 Geração automática de Casos de Teste

2.3.1 Definição de Caso de teste

Um caso de teste consiste, normalmente, em definir um identificador, quais as referências e especificações de *design*, pré-condições, eventos, ações a serem seguidas, entradas, saídas, resultado esperado e resultado real. Um caso de teste em engenharia de *software* é definido como um conjunto de condições ou variáveis que um testador determina, de forma a determinar se um sistema de software ou aplicativo está a funcionar e se funciona de forma correta ou não (IEEE, 1998).

Casos de teste são compostos por um conjunto de etapas, condições e entradas que são utilizados durante a fase de testes. O objetivo dos casos de teste é mostrar que um *software* passa ou não, no que diz respeito as funcionalidades e outros aspetos definidos no inicio do processo de conceção do software. Os casos de teste podem ser distintos, podendo estes ser do tipo funcionais, negativos, lógicos, físicos, de interface entre outros.

Um caso de teste permite manter o controlo sobre o teste de *software*, mas não existe um modelo formal que defina como deve ser escrito um caso de teste, no entanto, existem componentes

que estão sempre incluídas em todos os casos de teste que são criados. As componentes que todos os casos de teste criados têm são: um identificador do caso de teste, módulo do produto, versão do produto, histórico de revisões, objetivo, pressupostos, pré-condições, passos, resultados esperados, resultados observados e pós-condições.

Um bom caso de teste é aquele que tem uma grande probabilidade de encontrar o maior número de defeitos, no menor espaço de tempo e com a utilização do menor número de recursos possíveis. Não sendo possível garantir um programa sem qualquer erro, a qualidade de um *software* consegue-se ao se identificar um conjunto de casos de teste que possuam a probabilidade de descobrir o maior número de defeitos (Kansomkeat & Rivepiboon, 2003).

Um caso de teste não tem apenas o papel de avaliar se um programa tem erros, tendo em conta os resultados obtidos comparando-os com os resultados esperados, um caso de teste tem também o objetivo de auxiliar no tipo de entradas que um programa pode receber, conseguindo testar-se se no caso de uma entrada ser inserida de forma incorreta ou inválida o sistema responde como deveria.

A escolha dos casos de teste a serem usados é um dos pontos mais críticos da atividade de testes, pois como não se pode testar o programa num todo, a elaboração dos melhores casos de teste é fundamental para garantir um *software* de qualidade.

Os casos de teste podem ter diferentes objetivos, como tal existem três tipos de técnicas pelas quais os casos de teste são construídos. Os casos de teste podem, então, ser construídos com base em objetivos funcionais, estruturais ou baseados em erros, servindo a informação disponibilizada por estas técnicas para definir os diferentes casos de teste. A quando da técnica funcional os requisitos dos casos de teste são definidos tendo em conta a especificações funcionais que o programa deve ter. A técnica estrutural permite criar casos de teste que testem a implementação do programa. A técnica baseada em erros disponibiliza elementos para a criação de casos de teste que procurem erros comuns que normalmente acontecem durante o processo de desenvolvimento de um *software* (IEEE, 1988).

2.3.2 Geração de casos de teste a partir de requisitos funcionais

Existem várias abordagens distintas para construir casos de teste a partir de requisitos funcionais, no entanto a maior parte das organizações trabalha com requisitos nas línguas naturais, como por exemplo casos de uso.

Segundo Myers (2004), o objetivo da atividade de testes é revelar falhas no *software*, sendo para tal elaborados casos de teste com vista a revelar falhas. Um caso de teste é conjunto de entradas, condições de execução e resultados esperados, sendo que resultados esperados podem ser tanto mostrar que o sistema funciona ou que não funciona.

Para se produzir casos de teste que revelem o maior número de defeitos, a atividade de testes deve ser dividida em quatro etapas: planeamento, projetos de caso de teste, execução de casos de teste e avaliação de resultados (Myers, 2004). A etapa de planeamento diz respeito a conceção das partes de teste, do ambiente de teste e dos procedimentos operacionais que viabilizem os testes. A fase de projetos de caso de teste considera em que cenários que podem acontecer a quando a implementação dos casos de teste, dependendo das entradas, saídas e resultados esperados. A etapa de execução dos casos de teste é onde se observa o comportamento da implementação dos casos de teste no ambiente definido. A avaliação de resultados é a fase onde são analisados todos os dados que dizem respeito à interação e onde se faz uma avaliação de forma a se formar uma decisão sobre a fiabilidade e prestação do caso de teste.

Para Molinari (2010), a principal razão para se apostar na automatização de testes de *software* deve-se ao facto de ser necessário realizar cada vez mais testes em menos tempo e também o facto de as aplicações serem mais complexas e de se exigir uma maior qualidade do produto final.

A geração automática de testes a partir de requisitos é uma das vertentes da automatização de testes. Segundo Chen (2010), a escrita manual de casos de teste é ineficiente e demorada, como tal recomenda o uso de tecnologia para automatização dos casos de teste.

2.3.3 Testes manuais versus Testes automatizados

Um teste manual é aquele em que o *tester* (pessoa que executa o teste) executa o *software* manualmente com base na especificação dos testes, a qual detalha os casos e procedimentos de testes. Os testes realizados dessa forma, em geral, são cansativos e de difícil repetição. Já os testes automatizados, são aqueles feitos com auxílio de alguma ferramenta apropriada. Eles exigem mais tempo para implementar os testes, mas garantem mais segurança na manutenção e permitem que os testes sejam executados a qualquer instante.

Antes de decidir em automatizar ou não os testes, é preciso analisar a aplicação testada e quais os testes que serão construídos. A principal razão para automatização dos testes é a necessidade de reexecução dos mesmos. Supondo, por exemplo, que um *tester* projetou centenas de testes e optou por executá-los manualmente. Se em algum momento da execução da execução dos testes for descoberta uma falha, então os programadores deverão ser avisados. Além disso, dependendo da falha, a execução dos testes restantes deverá ser suspensa até que a mesma seja corrigida. Após a correção, como existem grandes hipóteses de que outras partes do *software* sejam afetadas, o *tester* terá que executar todos os testes novamente e novas falhas podem ser encontradas, repetindo-se o ciclo. Em um caso típico de testes, existe a necessidade de se executar casos de testes centenas e até milhares de vezes. Esse fator, normalmente, justifica a automatização dos testes.

A automatização de um conjunto de testes, geralmente, solicita mais esforço que a execução manual, mas quando automatizado, a sua execução é bastante simples, necessitando-se apenas de executar uma script ou o clicar em alguns botões (Neto *et al.* 2007). Assim, em situações como a descrita, automatizar os testes é a melhor maneira de economizar tempo e dinheiro.

Contudo, nem sempre é vantajoso automatizar os testes. Existem situações em que realizar os testes manualmente é mais adequado. Se a interface de utilizador da aplicação for mudar consideravelmente em um futuro próximo, por exemplo, então qualquer automatização que for feita vai precisar ser refeita. Além disso, quando o *software* for simples e não houver tempo suficiente, ou se for improvável a reutilização dos testes, então o teste manual continua a ser uma opção considerada.

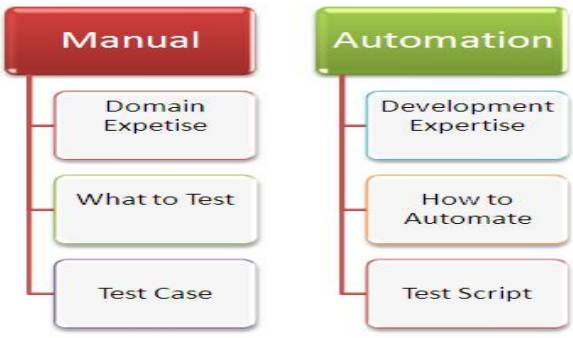


Figura 2.3 - Representação da diferença entre testes manuais e automáticos

2.3.4 Automatização de testes

Quando se fala em automatização é implícito que se pense em execução de uma determinada tarefa com uma maior rapidez. Na automatização de testes não é apenas a velocidade que aumenta, existem outros benefícios que proveem dessa prática. Ao se automatizarem os testes, estes dispensam uma supervisão, o que faz com que recursos sejam poupados. Outras das vantagens em automatizar testes influência a qualidade do produto a ser testado, pois em tarefas repetitivas feitas manualmente, muitas das vezes fruto do cansaço e enfades, tende-se a ignorar ou a negligenciar falhas, o que quando os testes são automáticos não acontece.

A automatização de testes pode ainda ser utilizada para aspetos que não podem ser testados manualmente, como por exemplo testes de *stress* ou de carga. Com recursos a ferramentas de automatização consegue-se simular casos que manualmente seriam impossíveis.

A automatização de testes veio alterar a forma como os testes são feitos. Num teste manual consegue-se determinar o comportamento do *software* com base no conhecimento que se tem. Num teste automático não se consegue determinar o comportamento com base no conhecimento, é necessário definir-se regras ou condições. É necessário definir o que é necessário testar e quais são os resultados esperados, devendo ser estas definições extremamente detalhadas.

Uma forma de reduzir os custos da atividade de testes é a partir da automatização dos testes usando alguma ferramenta apropriada. No caso dos testes funcionais, a maioria das ferramentas disponíveis se baseia na gravação de todas as ações executadas por um utilizador, gerando um script com os passos realizados. Dessa forma, as ações gravadas podem ser facilmente reexecutadas, permitindo assim a automatização do teste. Além disso, o script gerado pode ser facilmente alterado, permitindo modificações nos testes gravados sem a necessidade de reexecução do *software* e nova captura (Neto *et al*. 2007).

Um caso de teste consiste numa especificação que identifica um conjunto de interações do *software* que devem ser testadas. Essa especificação inclui os dados de teste, ou seja, as entradas e as saídas esperadas, e as condições sob as quais o teste deve ocorrer. O projeto de casos de teste é um dos grandes desafios do processo de teste de *software*, e pode ser tão desafiador quanto o projeto do próprio software (Pressman 2006). Como geralmente é impossível testar um programa exaustivamente, ou seja, testar todos os possíveis casos, o conjunto de casos de

teste deve ser o mais abrangente possível levando em conta qual o subconjunto dos casos possíveis tem maior probabilidade de exibir falhas.

A automatização da geração de casos de teste tem como objetivo tornar esse processo de identificação e geração mais ágil, menos propenso a erros humanos, mais rápido e menos dispendioso (Quentin, 1999). Em alguns tipos de testes, a repetição excessiva de procedimentos e a quantidade de iterações necessárias, pode se tornar impraticável para um humano realizar esses testes. O aparecimento de ferramentas de automatização pode acabar com o problema da limitação humana e também produzir uma informação com mais qualidade sobre um tipo de teste feito. Existe uma expectativa em torno da automatização de testes que muita das vezes não pode ser satisfeita. A automatização de testes é uma atividade que acrescenta custos a um projeto e que requer formação adequada para a realização da mesma quer em termos da própria atividade de automatização de testes, quer em termos do estudo da ferramenta a utilizar. O tempo e esforço para a criação de scripts automáticas é maior quando comparados com os testes manuais, sendo que existe uma diminuição de produtividade na fase de construção de testes automáticos. Mas tendo em atenção um estudo realizado em (Pretschner et al., 2005), onde se compara a capacidade de um conjunto de testes detetar erros de especificação e implementação, nota-se que, testes gerados automaticamente detetam um maior número de erros quando comparados com testes gerados manualmente tendo em conta os mesmos requisitos. Assim sendo a automatização é justificada, apesar do custo inicial ser maior, o resultado gerado futuramente compensa, pois ao serem detetados um maior número de erros, significa que o projeto vai ter uma melhor qualidade, o que trás benefícios a quando da conclusão do projeto.

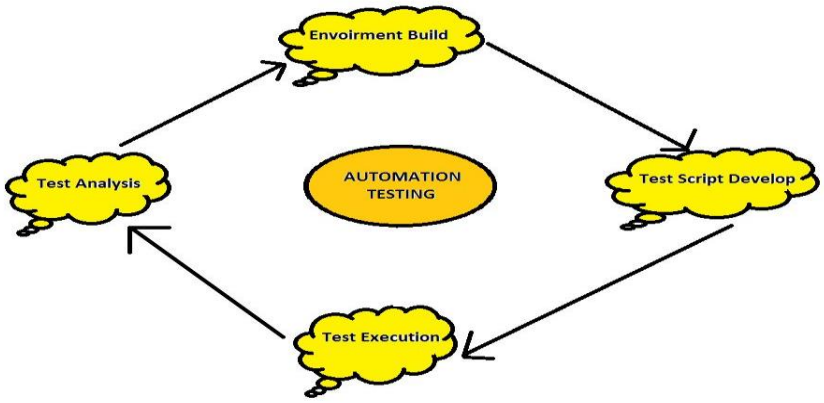


Figura 2.4 - Representação da automatização de testes

2.3.5 Técnicas de automatização de testes

Existem várias técnicas de automatização de testes, sendo as principais *record & playback*, programação por *scripts*, *data-driven* e *keyword-driven* (Fewster e Grahm, 1999). Neste ponto irá se fazer uma pequena descrição do que consiste cada uma destas técnicas.

Record & playback **-** consiste em, através da utilização de uma ferramenta de automatização de teste, gravar as ações realizadas por um utilizador sobre a interface gráfica de uma aplicação e converter essas ações em scripts de teste que podem ser executadas quantas vezes for desejado. De cada vez que este script for executado, as ações que foram gravadas são repetidas, como da primeira vez em que foi o utilizador que as executou. Para cada caso de teste é gravado um script completo que inclui os dados, procedimento e ações do teste sobre a aplicação. Para testes que sejam executados um baixo número de vezes, esta técnica é bastante boa, dado o facto de ser uma técnica simples e prática. No caso de serem tratados um grande número de testes automatizados, esta técnica já não é a melhor pois implicaria um elevado custo e dificuldades de manutenção, uma baixa reutilização e no caso de existir uma mudança no *software* ou de interface gráfica teriam de se alterar todos os scripts de teste.

Programação de *scripts* - é uma extensão da técnica descrita anteriormente. Nesta técnica as scripts de teste são alterados de forma a desempenharem um comportamento diferente do original durante a sua execução, isto consegue-se através da programação do *script*. Assim sendo é necessário conseguir-se a edição dos *scripts* de forma a poder programa-las, podendo os *scripts* teste serem alterados de forma a realizar uma maior quantidade de validações de resultados. A técnica de programação de scripts permite uma elevada reutilização, um elevado tempo de vida, melhor manutenção e uma maior robustez. Se por exemplo se altera uma interface gráfica, seria necessário apenas alterar algumas partes do *script* já criado. A grande desvantagem nesta técnica é a realização de um elevado número de scripts teste, pois para cada caso de teste deve ser também programado um *script* teste.

Data-driven - consiste em extrair os dados de teste, dos *scripts* de teste, que são especificadas em cada caso de teste, e guardá-las em arquivos separados dos *scripts* de teste. Assim sendo, os *scripts* de teste passam a conter apenas os procedimentos e as ações de teste, que muita das vezes são genéricos para um grande número de casos de teste. Consegue-se através desta técnica

que os *scripts* de teste apenas obtêm os dados de teste caso seja necessário pois estes dados de teste estão guardados no arquivo separado, que pode ser acedido diretamente pelo *script* de teste. Esta técnica permite que se possa apagar, modificar e adicionar dados de teste de uma forma fácil, permitindo ao projetista de teste apenas se preocupar em elaborar os arquivos com os dados e não com questões técnicas de automatização de teste.

Keyword-driven - tem como fundamento extrair, dos *scripts* de teste, o procedimento que representa a lógica de execução. Passando os *scripts* de teste apenas a conter ações específicas do teste sobre a aplicação, sendo estas identificadas por palavras-chave. As ações de teste são ativadas pelas palavras-chave a partir da execução de diferentes casos de teste. O procedimento de teste fica armazenado em um arquivo separado, na forma de um conjunto ordenado de palavras-chave e respetivos parâmetros. Nesta técnica as scripts de teste não mantêm os procedimentos no código, invocando-os diretamente dos arquivos de procedimento de teste, conseguindo-se assim a vantagem de facilmente se poder modificar, remover ou adicionar passos de execução no procedimento de teste sem necessidade de manutenção dos *scripts* de teste.

2.3.6 Melhores práticas para a automatização de casos de teste

Preparação para a automatização

É impossível automatizar um teste por completo, por isso é importante determinar quais os testes que a serem automatizados trariam mais benefícios ao projeto. Os principais testes candidatos à automatização são aqueles que exigem grandes quantidades de dados de entrada e aqueles que são executados com frequência. Testes que necessitem de intervenção humana são testes que não devem ser automatizados. As características que ajudam a definir se um teste é candidato a ser automatizado são:

- Se o teste for repetitivo;
- Se o teste tem como função avaliar condições de elevado risco;
- Se o teste é impossível ou demasiado caro para ser executado manualmente;
- Se são necessários vários dados de entrada para executar a mesma ação.

Com a automatização de testes que contêm características iguais às a cima descritas, os testadores ficam com mais tempo para realizar testes que tenham de ser feitos manualmente com uma maior profundidade.

Preparação da equipa e processos para a automatização

Quando se entra num processo de automatização de testes é necessário identificar e preparar uma equipa de testes e processos para a realização dos mesmos. A divisão de tarefas tendo em conta as características de cada individuo, pode ter um grande impacto no processo de automatização, quer em termos de qualidade do processo quer em termos de tempo de execução.

A realização de um plano de automatização ajuda a decidir quais os testes a automatizar, este plano pode conter quais os objetivos que se pretende com a automatização e uma descrição de quais os tipos de teste ou processos que serão automatizados. Feito este plano, identifica-se qual o conjunto inicial de testes a ser automatizado e ainda pode servir como guia para testes que sejam necessários realizar no futuro.

A criação de scripts para a automatização de testes devem ser feitas sobre casos de teste que já foram pensados e documentados e não de uma forma informal. A escolha da ferramenta certa para o processo também deve ser tomada em conta, pois dependendo de qual o processo de automatização a ferramenta pode ser uma ajuda ou não. Assim sendo um estudo sobre qual a melhor ferramenta para cada processo em concreto não deve ser negligenciado.

Implementação do teste automático

Aquando da implementação de um teste automático existe documentação que deve ser produzida de forma a garantir qualidade nesse processo. Não deve ser criado apenas um plano para a automatização, mas também deve ser criado um documento padrão de forma a se conseguir uma organização e uma estrutura para a forma como os *scripts* devem ser guardadas. Deve se ainda criar um documento com a listagem de todos os testes que foram automatizados e qual a finalidade de cada um.

Os *scripts* de teste criados é melhor que sejam pequenas e caso necessário sejam combinadas com outras de forma a conseguir-se testes mais complexos, pois a criação de scripts longas e

complexas são mais difíceis de analisar e atualizar. Os *scripts* criados devem ser de fácil reutilização, compreensíveis, sustentáveis e modulares.

2.4 Modelação por DSL (Domain-Specific Languages)

2.4.1 Conceito de DSL (Domain-Specific Languages)

Linguagens específicas de domínio (DSL – Domain-Specific Language) são linguagens utilizadas para um propósito específico, que servem para resolver problemas relacionados com um domínio em concreto (Cook, 2007). As DSLs têm como objetivo reduzir a complexidade de problemas. Em termos de modelação o uso de DSLs permite a utilização de soluções na linguagem e nível de abstração em que o domínio esta inserido, o que trás vantagens em relação a linguagens de propósito geral como é caso da *Unified Modeling Language* (UML) (Chavarriaga, 2013). Os benefícios que as DSLs trazem podem ser bastantes. Com o uso de linguagens específicas de domínio alterações feitas em requisitos podem ser representadas como alterações no modelo, conseguindo-se assim uma mais rápida implementação. Alterações na plataforma tecnológica podem ser introduzidas e manipuladas na etapa de integração do padrão, sem que seja alterada a representação modelada. O volume do código a ser atualizado baixa consideravelmente e os erros de código podem ser corrigidos através do gerador de código (Cook, 2007).

As DSLs podem ser textuais ou gráficas, normalmente as DSLs textuais são usadas para *input* enquanto as gráficas são usadas par output, pois facilitam a visualização de resultados. Para a implementação de DSLs é necessário seguir algumas abordagens. Uma das abordagens tem como principio especificar uma gramática usada para depois criar um *parse* de conversão. Esta abordagem deve ser realizada por alguém especializado na área, pois é uma tarefa complicada onde podem acontecer vários erros, devido ao facto de a gramática poder ser ambígua. Outra das abordagens consiste em utilizar propriedades de uma determinada linguagem de forma a serem utilizadas pelas DSLs, como por exemplo a utilização de classes para criar configurações específicas para a resolução de problemas. Uma outra abordagem aparece com a utilização de XML (eXtensible Markup Language), que é um formato para a geração de linguagens de marcação, que permite definir a representação de elementos e contem uma grande variedade de bibliotecas e ferramentas para processar documentos num formato específico.

2.4.2 Processo de criação de DSLs

O processo de criação de uma DSL é diferente do processo usado pelas linguagens de programação tradicionais (Spinellis, 2001). Apesar de no início algumas DSLs terem sido desenhadas como linguagens genéricas de programação (Wirth, 1974), e implementadas como interpretadores usando técnicas tradicionais de implementação de linguagens de programação, o processo de realização de DSL é bastante diferente daqueles que implementam linguagens genéricas. As DSLs são parte de um sistema abrangente e implementadas para um domínio específico. Para diminuir a dificuldade e o esforço atribuídos ao desenho e implementação de uma DSL, começaram a aparecer estratégias de reutilização, que ajudam a resolver problemas no desenho e que podem ser aplicadas a problemas similares, sendo designados por padrões (Gamma et al., 1995) (Coplien e Schmidt, 1995).

Segundo Deursen (1996), a criação de uma DSL tem de seguir um conjunto de etapas, as quais são descritas a baixo:

- **Análise:** a etapa de análise consiste em definir qual o papel que a DSL vai ter no domínio da aplicação onde vai ser criada. É nesta fase que deve ser recolhido e registado o conhecimento do domínio inserido, com o objetivo de passar este conhecimento para elementos semânticos e operações que são possíveis de definir numa linguagem computacional.
- **Implementação:** a fase da implementação deve assegurar que todas as definições semânticas definidas na etapa anterior são implementadas e deve-se nesta fase também desenvolver um compilador, para que se faça a tradução entre as DSLs criadas e as definições semânticas obtidas da etapa da análise.
- **Utilização**: esta etapa consiste em dar uso a DSL criada tendo em conta o domínio para o qual ela foi construída.

2.4.3 Testar uma DSL

Como uma DSL é uma linguagem que pode ser criada por qualquer pessoa e para um qualquer domínio, o teste de uma DSL é uma ação difícil e problemática, pois é necessário fazer uma abordagem sobre todos os aspetos da implementação da DSL. Entre estes aspetos a testar, estão

os aspetos de sintaxe, restrições, tipo de sistema (*collection*) e semântica de execução. Para facilitar o teste de todos esses aspetos, faz-se uma separação dos testes em diferentes categorias (Voelter, 2010):

- **Testar sintaxe**
    - O teste de sintaxe de uma DSL consiste em os desenvolvedores tentarem escrever um grande conjunto de programas e analisarem se esses programas podem ser expressos com a linguagem criada. Se essa passagem não se concretizar, isso significa que a sintaxe está imcompleta. Também se pode tentar escrever programas errados e verificar-se se os erros são detetados e se as mensagens de erro são reportadas.
- **Testar restrições**
    - O processo de testar restrições tem como objetivo garantir que as mensagens de erro são anotadas nos elementos corretos do programa caso esses elementos tenham restrições ou tipos de erro. Este processo é bastante importante sobre tudo para linguagens com restrições complexas.
- **Testar semântica**
    - Teste de semântica consiste em escrever afirmações que vão contra a execução do programa. Por exemplo criar uma DSL com base no entendimento daquilo que o programa deverá fazer, depois gerar o programa numa representação executável e escrever manualmente os testes unitários tendo em conta o comportamento do programa gerado e aquilo que se espera que a DSL faça.
- **Verificação formal**
    - A verificação formal, pode ser usada como complemento ao teste de semântica. A diferença existente entre a verificação formal e o teste de semântica é que no caso do teste de semântica, é especificado um cenário de execução em particular para cada caso teste, enquanto na verificação formal todo o programa é verificado de uma só vez. No teste de semântica para se ter uma verificação de todo o modelo é necessário cria-

se e executarem-se vários testes, o que pode ser bastante trabalhoso e podem ser deixados cenários de parte.

- **Testar serviços do editor**
    - Testar os serviços do editor traz alguns desafios, segundo o autor existem três maneira de abordar esta temática. Uma das abordagens consiste em o editor fornecer APIs especificas que ligam a aspetos de interface do utilizador de forma a facilitar os testes destes. Outra das abordagens consiste em utilizar ferramentas de teste de interface do utilizador genéricos de forma a simular a escrita no editor e assim verificar-se o comportamento que dai resulta. Por fim, podem se isolar os aspetos algorítmicos do comportamento do editor em módulos separados e em seguida fazer-se um teste unitário a cada um dos diferentes módulos.
- **Testar adequação da linguagem**
    - Uma DSL só tem utilidade se conseguir representar aquilo que é pretendido. Defende-se que uma DSL tem que dar cobertura total ao domínio onde vai atuar. De forma a validar a DSL há um conjunto de questões que devem ser colocadas, como por exemplo :se o domínio onde a DSL vai ser inserida é o domínio certo; se a DSL tem capacidade para cobrir todo o domínio; no caso de a DSL cobrir todo o domínio, ver se as abstrações são as adequadas para a finalidade do modelo; analisar se os utilizadores da DSL apreciam a sua notação e se a usam de forma eficiente.

      Para dar uma resposta a estas questões têm de ser feitas revisões manuais e validações aos requisitos, sendo que para tal uma DSL deve ser desenvolvida de forma incremental e iterativa.

2.4.4 Vantagens do uso de DSL

Com o uso de Linguagens Especificas de Domínio a programação torna-se mais fácil e mais confiável, pois consegue-se uma redução da distância conceptual entre o espaço do problema e a linguagem usada para exprimir o problema (UTCAT, 2006). A quantidade de código a ser produzido é menor e de mais fácil geração, sendo que ao se criar uma DSL que cumpra todos

os requisitos para ser considerada de qualidade, os problemas de domínio são reduzidos pois as soluções são expressas na mesma linguagem do domínio onde se está envolvido.

Alguns autores defendem algumas vantagens do uso de DSLs:

- Segundo Ladd e Ramming (1994), os programas criados com recurso a uma DSL são concisos, auto documentados e têm um elevado nível de reutilização.
- Para Deursen, as DSLs permitem a conservação e reutilização de conhecimento de um determinado domínio (Deursen *et al*., 2000).
- Permitem a validação e otimização ao nível de domínio e não da programação (Menon *et* Pingali, 1999).
- As DSLs permitem a reutilização no desenho de *software*, através do uso de geração automática de programas (Kieburtz, 2001).

2.4.5 Desvantagens do uso de DSL

O uso de DSL não traz apenas vantagens para quem as adota, são identificados também alguns aspetos menos positivos, caso haja, pontos que não sejam ponderados ou sejam negligenciados. Os maiores problemas das DSLs são sobre tudo referentes à sua criação, pois esta é uma etapa que envolve a compreensão do problema, isto quer dizer que se o problema não for bem compreendido existem grandes hipóteses de a DSL criada, conter irregularidades (UTCAT).

(Deursen et al., 2000), defende que também existe na fase da conceção questões que podem serem consideradas como desvantagens:

- Custo do desenho, implementação e manutenção das DSL;
- Dificuldade em criar uma DSL que seja a mais adequada no âmbito onde vai ser utilizada;
- Deursen defende que existe uma diminuição de eficiência quando se compara uma DSL com software escrito manualmente numa linguagem de programação genérica;
- Existe uma dificuldade na oscilação entre a criação de DSL e as de linguagem de programação genérica;
- Custo de formação dos utilizadores, pois estes passam a utilizar uma sintaxe com a qual não estão familiarizados.

2.5 Conclusão

A revisão de bibliografia feita neste capítulo permite adquirir conhecimento sobre a objetividade deste trabalho. Como se verifica no estado da arte, este projeto engloba vários conceitos de diferentes áreas.

A realização deste capítulo permite criar uma ponte de ligação entre os vários temas e áreas abordados. O foco deste capítulo foi feito em relação aos testes de *software*, casos de teste e formas de automatização e também sobre linguagens específicas de domínio.

Em relação aos testes de *software* percebe-se quais os tipos de teste existentes e quais as técnicas e abordagens a serem tomadas durante essa etapa do processo de desenvolvimento de um sistema. Já em ralação à temática de casos de teste o estudo serviu essencialmente para perceber como é realizada esta forma de especificação de requisitos, quais as vantagens que trás e como pode ser feita a sua automatização com o intuito de melhorar e tornar mais rápido o processo de teste de *software*.

Por ultimo foi feito um estudo sobre as linguagens especificas de domínio, como é que estas são criadas e como é que podem ser integrados em um ambiente e também qual a vantagem que a sua utilização por parte dos profissionais e empresas pode trazer.

Em suma a realização deste capítulo foi importante para se conseguir ter noções sobre os conceitos bases para a realização deste trabalho.

3 Derivação de Casos de Teste a partir de Requisitos

3.1 Introdução

Neste capítulo são apresentadas algumas tecnologias e ferramentas analisadas e utilizadas ao longo do projeto de dissertação, tendo em conta o objetivo de conseguir gerar cenários de teste de forma automática a partir de requisitos, especificados através de casos de uso. Para a realização deste projeto havia já certas tecnologias e ferramentas definidas à partida, visto serem as utilizadas pela organização em que este projeto foi desenvolvido.

Assim sendo são apresentadas as tecnologias e linguagens nas quais são representados os requisitos e como estes são representados, bem como as abordagens estudadas e utilizadas para se conseguir a automatização dos casos de teste. É explicado também neste capítulo o processo de mapeamento feito para que a partir dos casos de uso especificados se conseguissem cenários de teste viáveis.

3.2 UML e MDD

Para o desenvolvimento do projeto de geração de casos de teste de forma automática a partir de requisitos existiram duas tecnologias fundamentais para a sua realização, o UML *(Unified Modeling Language*) (Booch *et al*., 2005) e o MDD (*Model-Driven Development*) (Schmidt, 2006). O primeiro consiste numa linguagem de modelação que permite uma descrição de forma estruturada e compreensível no desenvolvimento de *software*, enquanto o segundo consiste numa técnica de desenvolvimento de software com base em modelos.

3.2.1 UML (Unified Modeling Language)

O UML é uma linguagem de modelação que auxilia o desenvolvimento de *software*. Está linguagem permite aos programadores conseguirem uma melhor visualização e projeção das funcionalidades e das características o que sistema a ser desenvolvido deve ter, bem como as comunicações existentes entre as várias componentes do sistema.

Em geral, o UML permite a criação de diagramas padronizados de forma a mostrar-se com base numa notação gráfica como será o sistema a ser desenvolvido. Como um modelo UML é construído com base em diagramas, muitas das vezes confunde-se um modelo UML com um diagrama de UML, mas um diagrama de UML é uma representação gráfica da informação de um modelo UML, podendo existir um modelo sem que para isso tenham de existir os diagramas desse modelo.

Esta linguagem de modelação foi criada tendo como base um conjunto de melhores práticas de engenharia, com o objetivo de facilitar a especificação, documentação, estruturação e visualização lógica do desenvolvimento de um sistema informático. Existem vários tipos de diagramas para representar diferentes aspetos de um sistema, segundo o paradigma UML (Booch *et al.*, 2005). Alguns dos principais diagramas são os seguintes:

- **Diagramas de Casos de Uso** – este tipo de diagramas são usados com o objetivo de descrever e definir requisitos funcionais de um sistema. Estes diagramas são constituídos por atores, sistema a ser modelado e podem conter outros casos de uso. Os atores representam as entidades externas ao sistema que interagem com este, esta interação é feita através de relações de associação, que representam determinada ação que um ator vai ter com o sistema.

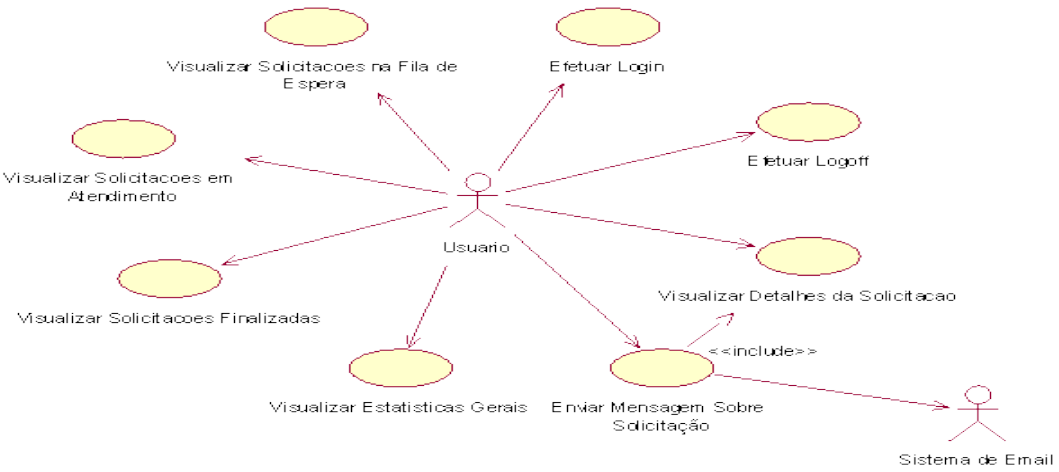


Figura 3.1 - Exemplo de um Diagrama de Casos de Uso

- **Diagramas de Classes** – os diagramas de classes representam a estrutura de classes do sistema. As classes de um sistema têm diferentes relações entre elas, como por exemplo

de associação, dependência, especialização, entre outras. Cada classe tem uma série de atributos e operações que esta realiza. Este diagrama ajuda a perceber qual a estrutura de um sistema, conseguindo-se ter a perceção daquilo que o sistema trata e necessita para corresponder aquilo que é esperado.

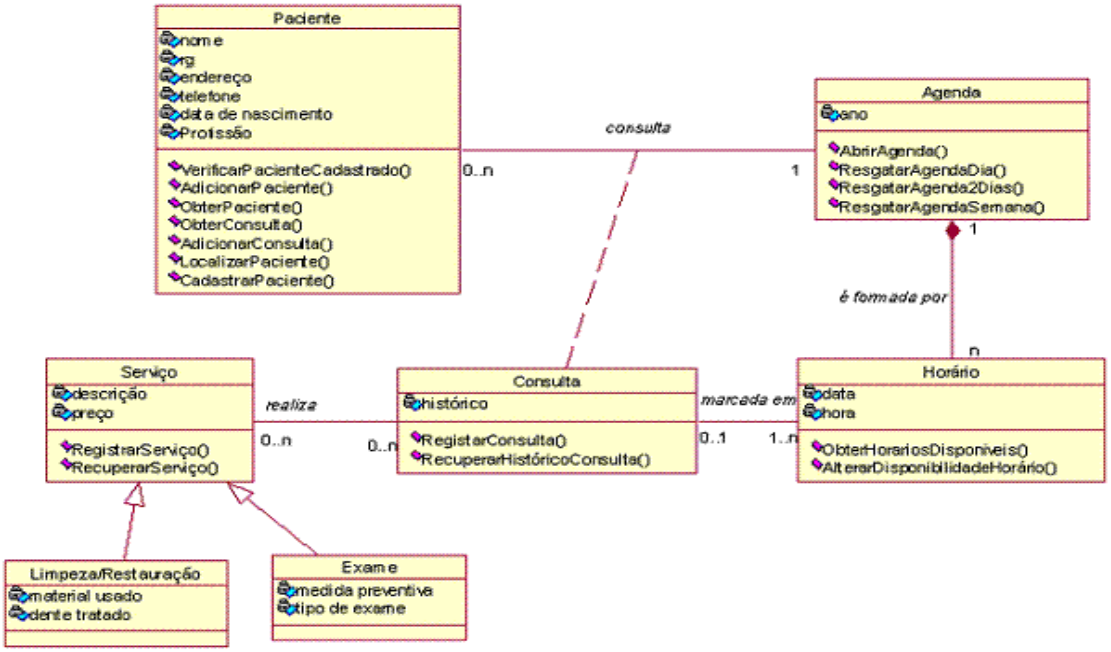


Figura 3.2 - Exemplo de um Diagrama de Classes

- **Diagramas de Objetos** - o diagrama de objetos é uma variação do diagrama de classes, tendo como objetivo mostrar quais os objetos que são instanciados de uma determinada classe. Este tipo de diagramas é bastante útil para exemplificar diagramas de classes que

sejam muito complexos, visto eles representarem todas as relações existentes entre as classes.

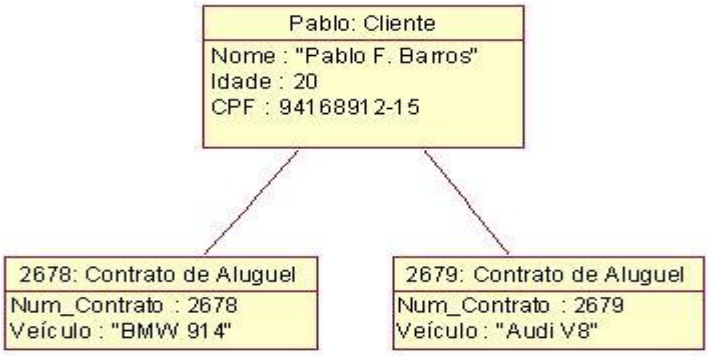


Figura 3.3 - Exemplo de um Diagrama de Objetos

- **Diagramas de Estado** – diagramas de estado são um auxílio à compreensão das classes que constituem um sistema. Nestes diagramas é feita uma demonstração de todos os estados possíveis que os objetos de uma dada classe podem ter e quais os eventos que levam a tais estados. Estes diagramas podem não ser elaborados para todas as classes que constituem um sistema, mas sim para aquelas classes em que é conhecido qual o seu comportamento e onde é afetado esse comportamento pelos diferentes estados do sistema. Assim este tipo de diagramas identifica o ciclo de vida de um determinado objeto, sistema e subsistema identificando quais os eventos que afetam determinados estados ao longo do tempo.

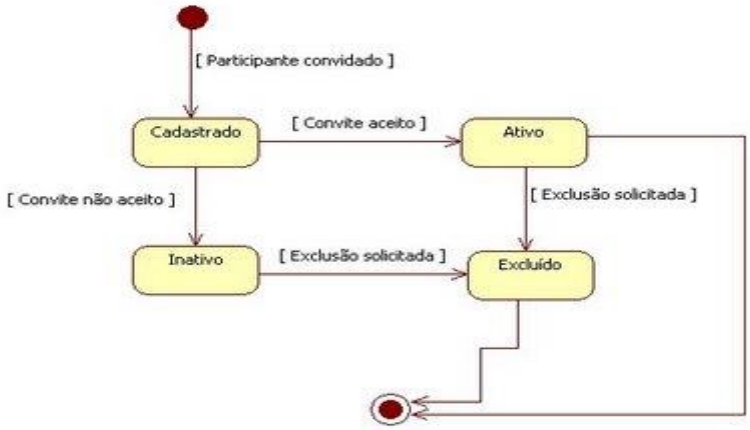


Figura 3.4 - Exemplo de um Diagrama de Estado

- **Diagramas de Sequência** – num diagrama de sequência são demonstrados a colaboração e relação existente entre um determinado número de objetos que constituem um sistema. É neste tipo de diagramas que se percebe qual a sequência de mensagens que são trocadas entre objetos para um determinado evento. Consegue-se identificar nestes diagramas o que acontece durante o ciclo de vida de um evento, podendo saber-se o que acontece em determinado momento. Os diagramas de sequência são elaborados com recurso a dois eixos, um vertical e outro horizontal, sendo que o eixo vertical corresponde ao tempo de vida de determinada atividade representada, enquanto o eixo horizontal representa os objetos envolvidos numa atividade e quais as mensagens que são trocadas entre esses objetos.

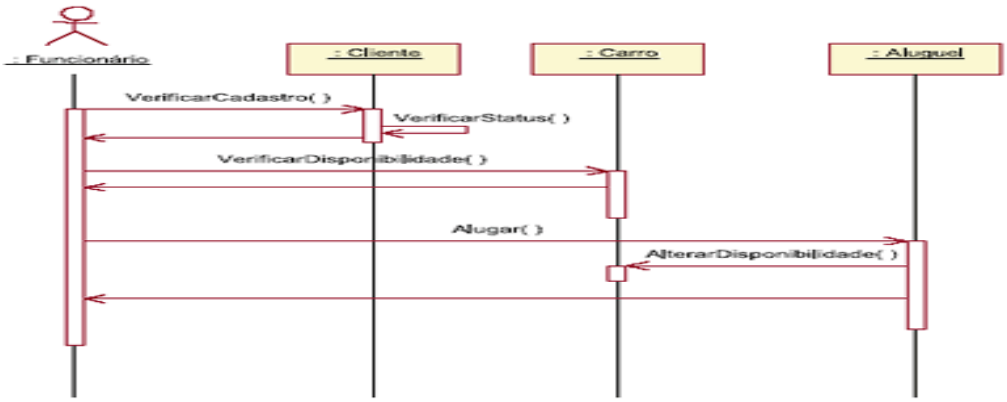


Figura 3.5 - Exemplo de um Diagrama de Sequência

- **Diagramas de Atividade** – nos diagramas de atividade é focada e representada a implementação de uma operação e quais as suas atividades numa instância de um objeto. O objetivo destes diagramas é identificar as ações que são executadas e quais os resultados que derivam dessas ações em relação ao estado dos objetos. Os diagramas de atividade podem ser elaborados tendo em vista diferentes propósitos como, identificar o trabalho que é executado quando uma ação é realizada, identificar as operações internas de um determinado objeto, mostrar como é que um determinado número de

ações relacionadas podem ser executadas e quais os objetos afetados por estas e mostrar como é que uma instância de um objeto pode ser executada.

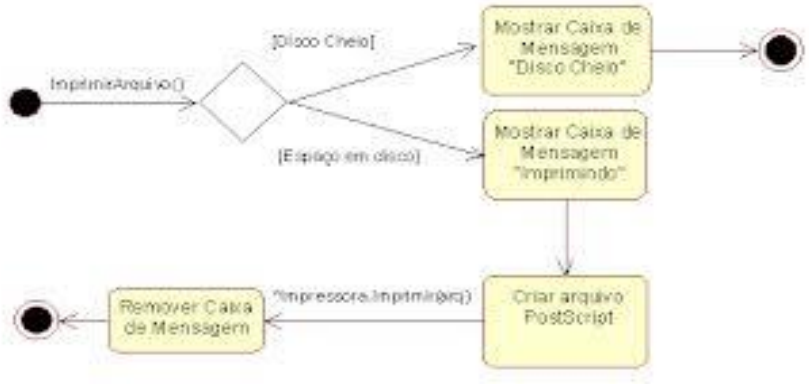


Figura 3.6 - Exemplo de um Diagrama de Atividade

3.2.2 Casos de Uso

Em engenharia de *software*, um caso de uso é uma lista de passos que define as interações entre um ator e um sistema de forma a alcançar um objetivo. Com recurso a casos de uso consegue-se descrever a unidade funcional de um sistema, conseguindo-se identificar os requisitos que são necessários para que o sistema corresponda ao espectável. Cada caso de uso deve apenas descrever unicamente uma só funcionalidade ou objetivo esperado pelo sistema. Assim sendo, quando a criação de um sistema é usual construir vários casos de uso. De forma a facilitar uma melhor compreensão do que é espectável em cada funcionalidade que se pretende representar, muitas das vezes os casos de uso são representados na forma de diagramas, conseguindo-se assim uma melhor compreensão sobre quais os atores que cada caso de uso contem e também qual os tipos de relação existente entre cada um deles.

Quando se cria casos de uso deve-se ter em conta que apenas são descritas funcionalidades que são pretendidas que o sistema realize e não a forma como elas vão ser realizadas, não sendo identificado nos casos de uso qualquer funcionamento sobre o estado interno do sistema que implementa essa funcionalidade, nem a forma como esta será implementada.

Na construção de um sistema informático, cada caso de uso diz respeito a uma característica do sistema, conseguindo-se assim, através da criação dos vários casos de uso, uma visão geral das funcionalidades e requisitos que serão desenvolvidos para a criação do sistema.

Construção de um Caso de Uso

Para construir um caso de uso existe um conjunto de regras e definições que devem ser seguidas de forma a se ter a certeza que um caso de uso está corretamente construído e compreendido. O *standard* usualmente utilizado para a construção de um caso de uso e que facilita a sua compreensão, consiste na elaboração das seguintes secções:

- **Nome**: o nome do caso de uso é o indicador que informa a que atividade este se refere. Idealmente esse nome deve ser composto por um verbo e um substantivo.
- **Iterações**: nesta secção é descrito o estado do caso de uso à medida que este vai evoluindo.
- **Sumário**: o sumário consiste num pequeno resumo que explica em que consiste o caso de uso.
- **Pré-condições**: dentificação de condições que devem ser verificadas quando se dá inicio ao use case. Estas pré-condições são ações anteriores que devem ser respondidas e validades de forma a se dar início ao caso de uso atual.
- **Fluxo de Eventos**: nesta secção são descritos os eventos que vão sendo realizados ao longo do ciclo de vida do caso de uso. Normalmente estes eventos têm uma sequência lógica do sistema e são numerados de forma a se compreender qual a sequência a seguir.
- **Fluxos Alternativos**: fluxos alternativos são uma sequência de eventos que são alternativos aos eventos descritos na secção do fluxo de eventos.
- **Pós-condições**: pós-condições consistem numa descrição do estado do sistema após ser executado o caso de uso.
- **Regras de negócio**: esta secção é reservada para a adição de informação relativamente a restrições, regras de negócio e informações sobre a organização para a qual o sistema vai ser desenvolvido.
- **Notas**: nesta secção pode ser acrescentada informação adicional que possa ser importante e que seja relativa ao caso de uso, mas que não seja especificada em nenhuma secção anterior.
- **Autor e Data**: informação de quais o autor ou autores responsáveis pela construção do caso de uso e das datas das várias versões revistas.

Vantagens em especificar casos de uso

A adoção de casos de uso para a identificação de requisitos num projeto de desenvolvimento de um sistema informático, vem terminar com a muita documentação que era produzida para esse propósito e que nem sempre conseguia a abrangência de todo o sistema, falhando-se muitas das vezes na identificação dos requisitos necessários. No desenvolvimento de sistemas é bastante fácil adicionar ou remover casos de uso tendo em conta as alterações que são feitas ao sistema. Uma das principais razões para o aumento desta técnica para se fazer levantamento de requisitos é o facto de os casos de uso serem de fácil compreensão para qualquer interveniente no âmbito do negócio. As vantagens identificadas na adoção desta abordagem são as seguintes:

- A quando da especificação de um caso de uso é fácil identificar quais os requisitos funcionais necessários que o sistema vai necessitar;
- Os casos de uso podem ser reutilizados dentro de um projeto;
- O facto de serem identificados e especificados fluxos alternativos ajuda na identificação de comportamentos adicionais que podem aumentar a robustez do sistema a ser desenvolvido;
- Com a implementação dos casos de uso é mais fácil fazer-se o planeamento em termos de esforço e tempo que o projeto pode vir a demorar;
- São especificações de fácil compreensão, por isso facilitam a comunicação entre todos os elementos que fazem parte do projeto;
- A elaboração de casos de uso ajuda a identificar a relação entre cada um dos requisitos encontrados e o âmbito de negócio, conseguindo-se identificar qual o contexto em que cada requisito está inserido;
- Permitem que os requisitos sejam definidos com um contexto e com um sentido, pois são definidos de uma forma sequencial e percetível em cada nível de execução em que o requisito está presente.

Limitações dos casos de uso

Apesar de todas as vantagens descritas a cima a especificação de casos de uso no desenvolvimento de *software* tem as suas limitações. Assim e apesar da especificação de casos

de uso ser bastante eficiente, principalmente quando se trata em identificar requisitos funcionais, são encontradas as seguintes limitações:

- Esta abordagem não é eficiente quando se pretende fazer o levantamento de requisitos não-funcionais, ou seja, características que não estão relacionadas diretamente com as funcionalidades e sim com a qualidade do sistema a ser desenvolvido;
- Apesar de um caso de uso seguir um *template* a quando da sua conceção, a compreensão desse caso de uso vai depender em muito da qualidade e clareza da informação de quem o elabora;
- Apesar da interpretação de um caso de uso ser relativamente fácil, é necessário passar-se por um processo de aprendizagem de forma a conseguir-se fazer uma interpretação correta daquilo que está elaborado no caso de uso.

3.2.3 Use Case Model

Um *Use Case Model*, ou modelo de caso de uso, consiste no fornecimento de informações detalhadas sobre o comportamento do sistema a ser desenvolvido, conseguindo identificar todas as formas esperadas de utilização do sistema. Este modelo é constituído por um conjunto de diagramas (casos de uso, sequência, atividades, estados, entre outros) que representam e descrevem as interações dos utilizadores com o sistema. Através de um modelo de casos de uso consegue-se a identificação de requisitos que o sistema deve conter em termos de funcionalidades de forma a dar uma resposta ao espectável. Neste modelo não é descrita a forma como o sistema deve funcionar internamente, mas sim em termos de interação com os utilizadores, podendo estes utilizadores serem pessoas ou outros sistemas que interajam com aquele a ser construído (Jacobson, 2011).

Com a adoção de um modelo de caso de uso no desenvolvimento de um sistema informático, consegue-se o seguinte:

- Consegue-se que as equipas de desenvolvimento cheguem a acordo sobre as funcionalidades e características que são necessárias ao sistema a desenvolver;
- Consegue-se estabelecer um limite entre os objetivos esperados e o sistema, conseguindo-se assim uma visão completa do sistema quer em termos de atores que interagem com este, quer também em termos de funcionalidade que este deve ter;

- Com a adoção da abordagem de modelos de caso de uso consegue-se uma gestão de requisitos do sistema mais fácil e mais ágil.

Os diagramas que compõem um modelo de casos de uso podem ser concebidos ou tratados tendo vários níveis de detalhe. Os três níveis de detalhe que um modelo de caso de uso deve tratar são: estabelecer valor, estabelecer limites ao sistema e estruturação.

Estabelecer valor: o primeiro passo as ser dado a quando da construção de um modelo de casos de uso consiste em identificar quais os atores mais importantes e quais as funcionalidades primárias do sistema, pois são estes que fornecem valor ao sistema. Sendo este nível de detalha um pouco mais suave, pode ser muito apropriado principalmente para projetos em que se pretenda adicionar funcionalidades a sistemas já existentes, onde não há grande acréscimo de valor em modelar toadas as funcionalidades que o sistema tem ou faz.

Estabelecer os limites do sistema: como descrito antes, os principais atores e casos de uso de um sistema capturam as principais funcionalidade e qual o principal valor do sistema a ser construído. Porém estes podem não ser suficientes para determinar o completo funcionamento que o sistema deve ter, assim, nestes casos pode ser necessário adicionar ao modelo atores e casos de uso secundários que apoiem um funcionamento mais eficaz do sistema. Assim este nível de detalhe é bastante útil a quando da modelação para a construção de um novo sistema.

Estruturado: como um modelo de casos de uso, muitas das vezes contem informações redundantes, como por exemplo sequências comuns, a estruturação do modelo de casos de uso é uma forma de lidar com esses inconvenientes. Para sistemas grandes e complexos, principalmente em sistemas que são usados para fornecer funcionalidades semelhantes dentro diferentes contextos, uma estruturação do modelo de casos de uso pode facilitar uma melhor compreensão, podendo levar a que se encontre elementos que possam ser reutilizáveis e outros que estejam em excesso e que possam ser eliminados.

Assim sendo, enquanto um modelo de casos de uso mostrar de forma clara o valor que o sistema a criar ou a atualizar vai fornecer, então a modelação está a ser feita como deve ser. Na modelação existem alguns cuidados a serem tomados quando do acréscimo de detalhe ao modelo, pois deve ser garantido que o limite estabelecido para o sistema é respeitado e que os

detalhes adicionados acrescentam valor ao sistema, tornando-o mais eficiente. Caso contrário não deve ser acrescentado detalhe ao modelo.

3.2.4 Desenvolvimento Orientado a Modelos (MDD)

Em todos os ramos de engenharia a construção de modelos é fundamental para a compreensão de um problema complexo e conseguir descobrir possíveis soluções que consigam contornar esses problemas. Assim em engenharia informática pode-se ter grande benefício com a adoção de modelos e técnicas de modelação no desenvolvimento de sistemas. No processo de desenvolvimento de sistemas, modelos eram usados como algo com um papel mais secundário, servindo sobre tudo para a simplificação e compreensão do problema em termos de documentação. Com a abordagem *Model-Driven Development* (MDD), o papel dos modelos e da modelação no processo de desenvolvimento de *software* adquiriu um novo e vantajoso papel. Abordagens de desenvolvimento orientadas a modelos foram concebidas com o intuito de aproveitar sobre tudo o facto da tecnologia que suporta os modelos ter evoluído conseguindo-se processos de automatização (Schmidt, 2006). Em MDD o foco principal no desenvolvimento de *software* passam a ser os modelos, o que faz com que se consigam conceitos mais perto do domínio do problema e não tanto relacionados com a tecnologia de implementação a ser utilizada. Assim, consegue-se um melhor entendimento sobre o problema de domínio ao qual se deve dar uma resposta e como assim se consegue uma abordagem mais independente em termos tecnológicos fica também facilitada a mudança evolutiva que possa existir ou que seja necessária de ser feita. Uma das premissas existentes na abordagem de desenvolvimento orientada a modelos é que se consiga a geração de programas automaticamente a partir dos modelos correspondentes. E como o processo de modelação e as tecnologias de automatização foram evoluindo ao longo do tempo, esse passo, agora, é possível.

Consistindo o desenvolvimento orientado em modelos na geração automática de código a partir de modelos, apenas se pode dizer que se consegue todos os benefícios da abordagem se se conseguir gerar programas completos e não apenas esqueletos de código e modelos que executam automaticamente ao nível computacional. Conseguindo alcançar estes objetivos tem se programas mais fiáveis, pois consegue-se que uma linguagem de modelação assuma o papel de uma linguagem de implementação. Este processo é alcançável através de técnicas e ferramentas que funcionam como geradores de código que produzem o código proveniente de

modelos de forma automática e com a mesma eficiência, ou até mais, que o código feito de forma artesanal.

Como garantir modelos de qualidade

Em engenharia modelos foram criados com o objetivo de reduzir o risco e de ajudar a compreender melhor um problema e quais as possíveis soluções antes de se passar a parte da implementação. De forma a se conseguir modelos de melhor qualidade, existem cinco características que estes devem ter (Selic, 2003). A **abstração** é considerada uma das mais importantes, pois um modelo é sempre uma representação reduzida de um sistema que este representa. Esta característica permite que a essência do sistema a ser desenvolvido seja mais facilmente compreendida, pois são ocultados detalhes que sejam irrelevantes sobre um determinado ponto de vista, reduzindo a complexidade resultante da cada vez maior sofisticação dos sistemas de *software*. A **compreensão** é outra das características que um modelo deve ter. Um modelo deve ser facilmente compreendido, quase que de uma forma intuitiva, oferecendo assim um atalho e reduzindo a quantidade de esforço intelectual necessário para a sua compreensão. Esta característica trás vantagens pois deixa de ser necessária a análise detalhada e intelectualmente desgastante de quando se tenta compreender um determinado sistema com base em linguagens de programação. Outra das características que um bom modelo deve ter é a **precisão**, devendo um modelo fornecer uma representação precisa e exata das caraterísticas importantes do sistema a ser modelado. **Prevenção**, um modelo deve ser útil para prever corretamente propriedades que o sistema a ser desenvolvido deve conter. De realçar que esta característica é dependente da precisão que o modelo tem e da forma de modelação adotada. Por ultimo um modelo deve ser económico, isto significa que a construção e análise de um modelo deve ter um custo mais baixo que outro tipo de metodologias de desenvolvimento.

3.3 Análise do Modelo de Requisitos

O modelo de requisitos utilizado para a realização deste projeto foi o já utilizado na organização, usando-se assim exemplos reais e formas reais da especificação dos requisitos

utilizados para a realização deste projeto. Em seguida será explicado quais as ferramentas, técnicas e formas utilizadas para a descrição dos requisitos.

3.3.1 Ferramenta de especificação de requisitos

Yakindu *Requirements* (itemis, 2014) é um *software* para gestão de requisitos, com esta ferramenta consegue-se descrever os requisitos para um software com precisão e com riscos minimizados. Para a definição dos vários elementos de uma especificação de requisitos incluindo casos de uso, objetos de negócio e atores, interfaces e interfaces gráficas e fluxos de página bem como um editor de linguagem especifica, permitindo assim evitar-se erros desde o inicio do projeto e saber rapidamente onde se está, através da gestão de requisitos.

É uma ferramenta de fácil utilização, que não exige um treino excessivo para a sua utilização. Os benefícios do uso de este *software* são ter uma rápida vista sobre o risco e consequências de uma mudança de requisitos, menores custos através da mudança de requisitos, menor tempo despendido na documentação de defeitos, desenvolvimento mais rápido e com menos erros e menor tempo de realização do projeto.

As suas principais funções são fornecer especificações gráficas e escritas, fazer uma validação de requisitos e gerar documentação de requisitos automaticamente.

Esta ferramenta é utilizada principalmente por analistas e engenheiros de *software*.

É uma ferramenta que permite o desenvolvimento de sistemas orientados a modelos. Com o uso de modelos, consegue um processo de desenvolvimento integrado e aumenta a qualidade e sustentabilidade do projeto. É uma ferramenta baseada na plataforma Eclipse, podendo assim integrar-se perfeitamente com ambientes baseados nesta plataforma.

Permite especificar aplicações complexas em pouco tempo e manter os efeitos e riscos de mudança de requisitos ao longo de todo o processo de desenvolvimento.

Com esta ferramenta consegue-se especificações que são compreensíveis a todos os grupos alvo, requisitos para soluções de TI (Tecnologias de Informação) podem ser formulados precisamente e visualizar o processo de negócio. Permitindo assim que todos os envolvidos no projeto se compreendam, isto é, todos falam sobre a mesma coisa.

Pode-se ainda através de esta ferramenta criar especificações de requisitos de forma mais rápida e mais fácil. A criação manual de relatórios é eliminada e a pesquisa e navegação através de documentos criados é facilitada. Análise que antes eram feitas à mão, como por exemplo quantos casos de uso um objeto específico de negócio tem, é feito agora automaticamente. Diagramas e visualizações, como casos de uso, são gerados automaticamente através de descrições.

Utilizando a ferramenta Yakindu consegue-se gerar especificações sem erros, pois este oferece um alargado conjunto de assistentes e verificações de sintaxe e de semântica que simplifica a recolha de requisitos, e reduz os erros de especificação. Se um erro se arrasta, este é identificado logo desde de início, fazendo com que os custos de correção sejam o menor possível.

Fornece indicações sobre o custo e esforço durante todo o processo, sendo que as especificações criadas são completas, consistentes e inequívocas. Da documentação resultam estimativas do esforço concreto e podem ser feitas declarações sobre a duração e o custo da realização de cada projeto.

Consegue-se uma identificação dos esforços necessários e qual o impacto esperado. Assegura-se assim que os requisitos de um sistema e as suas alterações possam ser seguidos durante todo o processo.

Na imagem abaixo é representado um esquema sobre o funcionamento da ferramenta Yakindu, onde se demonstra que a ferramenta diminui a complexidade e manipula um modelo a partir do sistema, dos processos e da arquitetura, conseguindo gerar artefactos como código, documentação, testes e documentação técnica.

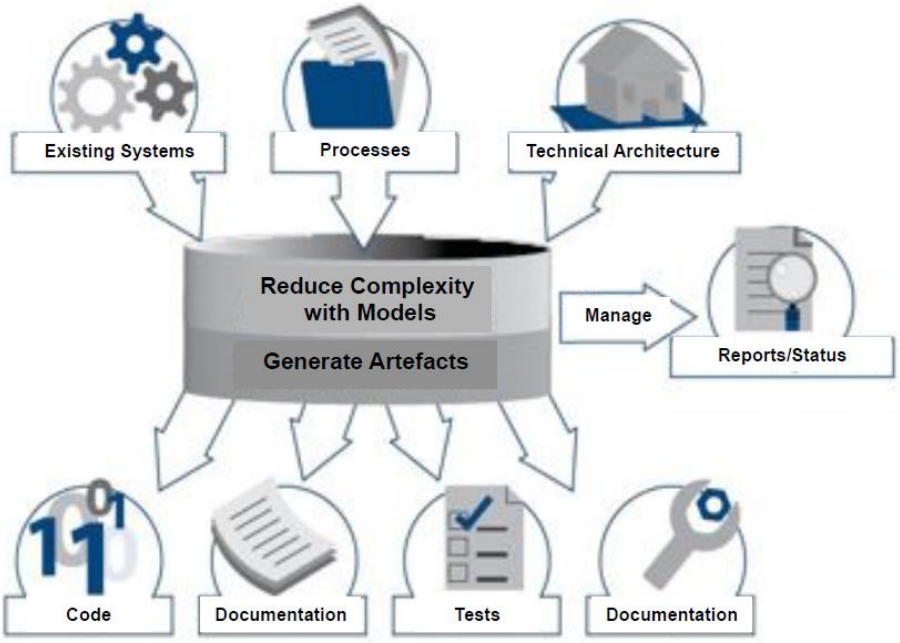


Figura 3.73 - Funcionalidades da Ferramenta Yakindu Requirements (itemis, 2014)

3.3.2 Especificação de Casos de Uso na ferramenta

A definição de casos de uso na ferramenta Yakindu Requirements consiste na descrição de um conjunto de passos que representam o fluxo da informação.

Na definição de casos de uso existem um conjunto de instruções que devem ser seguidas. No contexto da organização também existem algumas regras internas que foram definidas a quando da definição de casos de uso.

- **usecase** – A instrução usecase sinaliza o inicio da definição de um caso de uso. Esta instrução conte um <**id**> que corresponde a uma identificação do caso de uso e uma <**descrição**> onde é feita uma abreviada descrição do propósito do caso de uso. Um *usecase* é sempre terminado com a instrução <**end usecase**>.
- **goals** – Nesta instrução deve ser indicado de forma clara qual o objetivo do caso de uso.
- **requirements** – Aqui são identificados quais os requisitos a que o caso de uso satisfaz.
- **preconditions** – Caso exista uma ou mais pré-condições para o caso de uso, estas devem ser aqui descritas. A descrição é feita em texto livre podendo ainda à frente da descrição

ser adicionada a instrução <**requires**> que indica uma dependência com outro caso de uso.

- **labels** – Labels são valores de texto livre, que podem servir de identificador para o caso de uso.
- **actors** – Aqui são introduzidos quais os atores que interagem com o caso de uso.
- **entities** – São objetos de negócio que são utilizados no caso de uso.
- **pages** – Esta instrução permite identificar quais as páginas e os screens que o caso de uso utiliza.
- **basic flow** – Nesta instrução inicia-se a definição dos vários passos que o caso de uso vai ter. Após a definição de todos os passos esta instrução termina com a instrução <**end flow**>.
- **step** – A instrução step marca o início de um passo. Esta instrução é seguida de um número e de uma descrição. A instrução step pode ser seguida de outras instruções. Uma delas é a instrução <**by**> seguida de um ator, que identifica qual o ator responsável por aquele passo, também pode ser seguido pela instrução <**using**> seguida de uma entidade, que informa quais as entidades usadas no passo e ainda pela instrução <**on**> seguida de um screen, esta instrução atribui um screen ao passo em questão. Um step pode ainda conter uma série de acções:
    - **continue with step <nº do passo>** - com esta instrução consegue-se saltar para um passo, podendo esse salto ser feito para para um passo posterior ou um anterior.
    - **invokes usecase <ID>** - nesta instrução permite a invocação de outro caso de uso num determinado passo. Deve ser seguido da instrução <**continue with step**> caso se pretenda continuar em outro passo.
    - **alternatives** – esta instrução permite a identificação de alternativas que um determinado passo possa ter. As alternativas devem ser iniciadas pela instrução <**alternatives**>, seguido de instruções do tipo:
        - **if** "condição" **then** <acção> e terminar com a instrução **else if** que pode conter uma condição ou não.
    - **Postcondition** – corresponde ao estado de retorno que o caso de uso vai ter. Assim é indicado nesta instrução aquilo que acontece no final do caso de uso.
    -

Na figura 3.8 é representado como é feita a especificação de casos de uso na ferramenta *Yakindu Requirements*. Esta ferramenta permite a criação de páginas, podendo posteriormente essas paginas serem associadas a casos de uso. Os casos de uso criados são ficheiros que do tipo *usecases*, já as páginas são ficheiros do tipo uiapplication. Na figura 3.9 é representada a especificação de uma página.

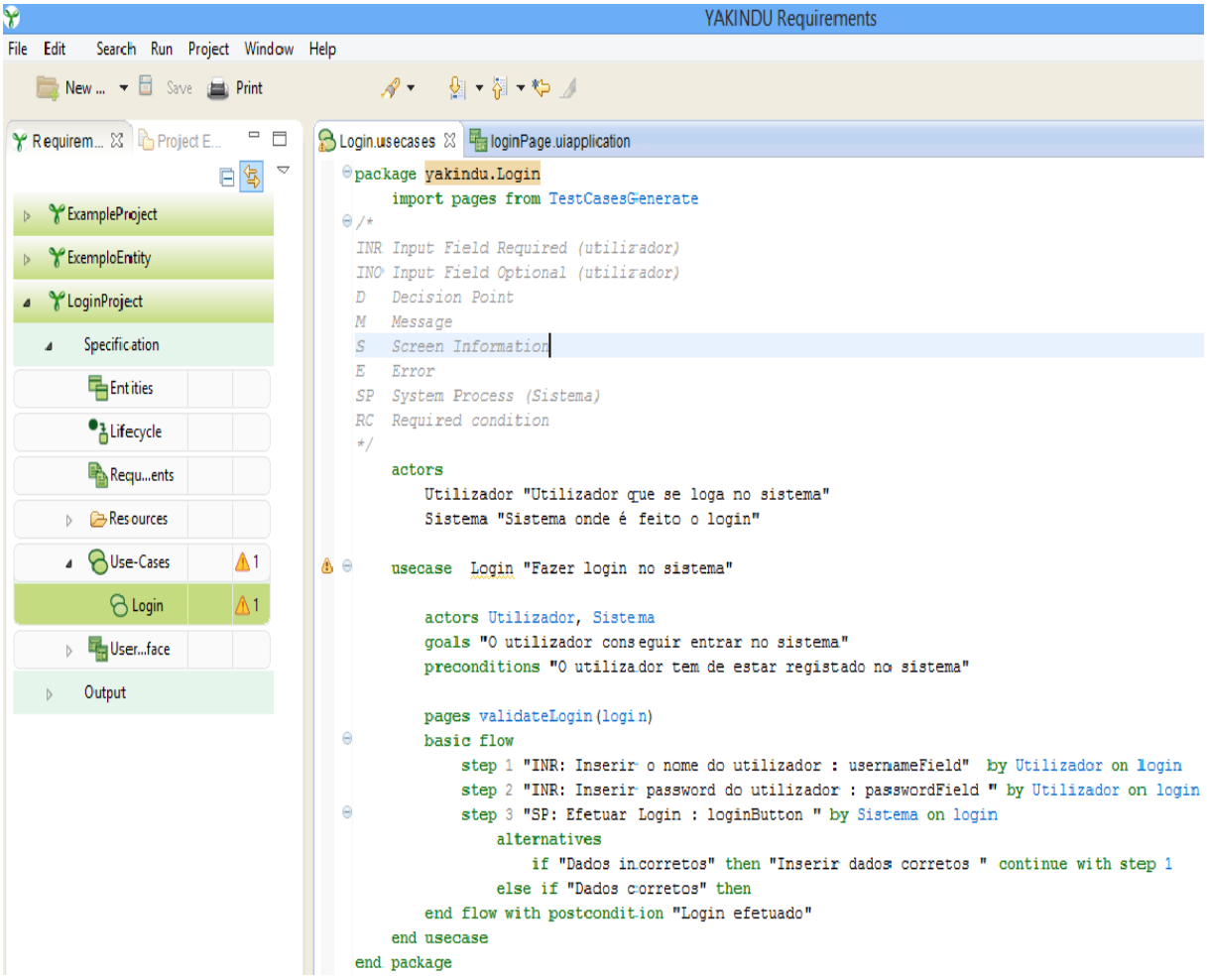


Figura 3.8 - Exemplo da especificação de um caso de uso no Yakindu

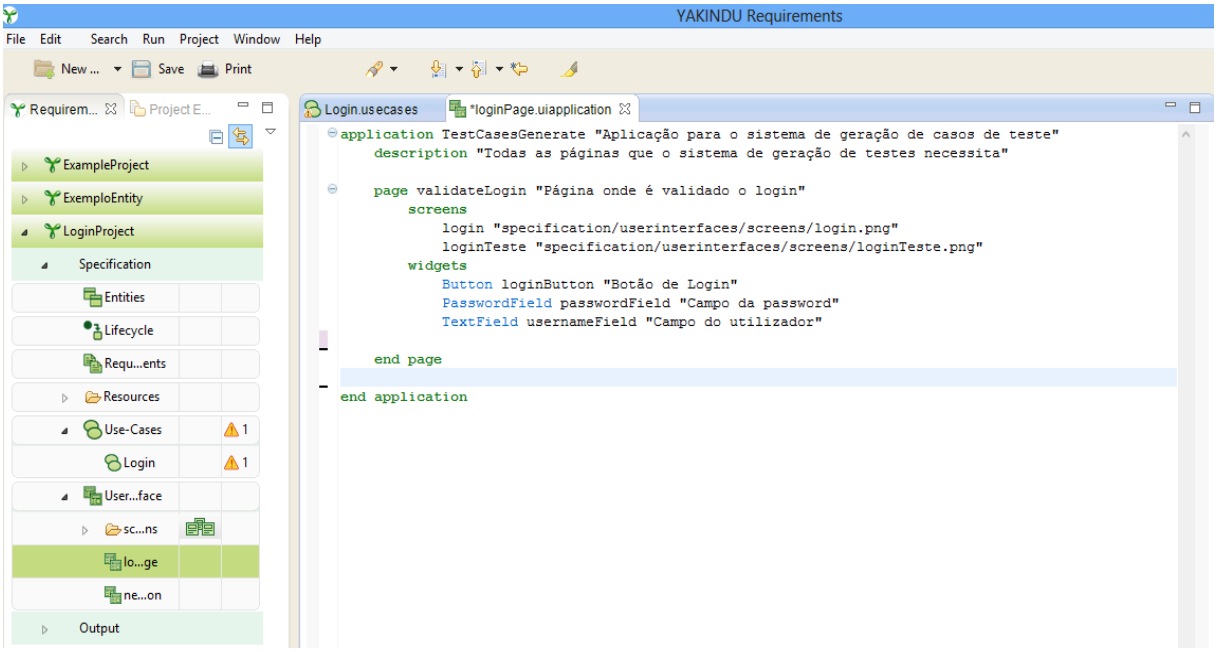


Figura 3.94 - Exemplo da especificação de uma página no Yakindu

3.3.3 Modelo dos Casos de Uso

O modelo de casos de uso que é carregado e utilizado segundo a abordagem MDD (Model-Driven Development), está representado na figura a baixo. O modelo utilizado consiste num conjunto de classes, em que cada classe contem um dado número de atributos e é responsável pela realização de certas operações.

Tendo como base o modelo de casos de uso, conseguindo-se assim acesso as suas classes, conseguiu-se identificar qual a classe que responsável por determinada operação e quais os

atributos utilizados, sendo este processo fundamental para o mapeamento ser feito, com vista à geração de cenários de teste.

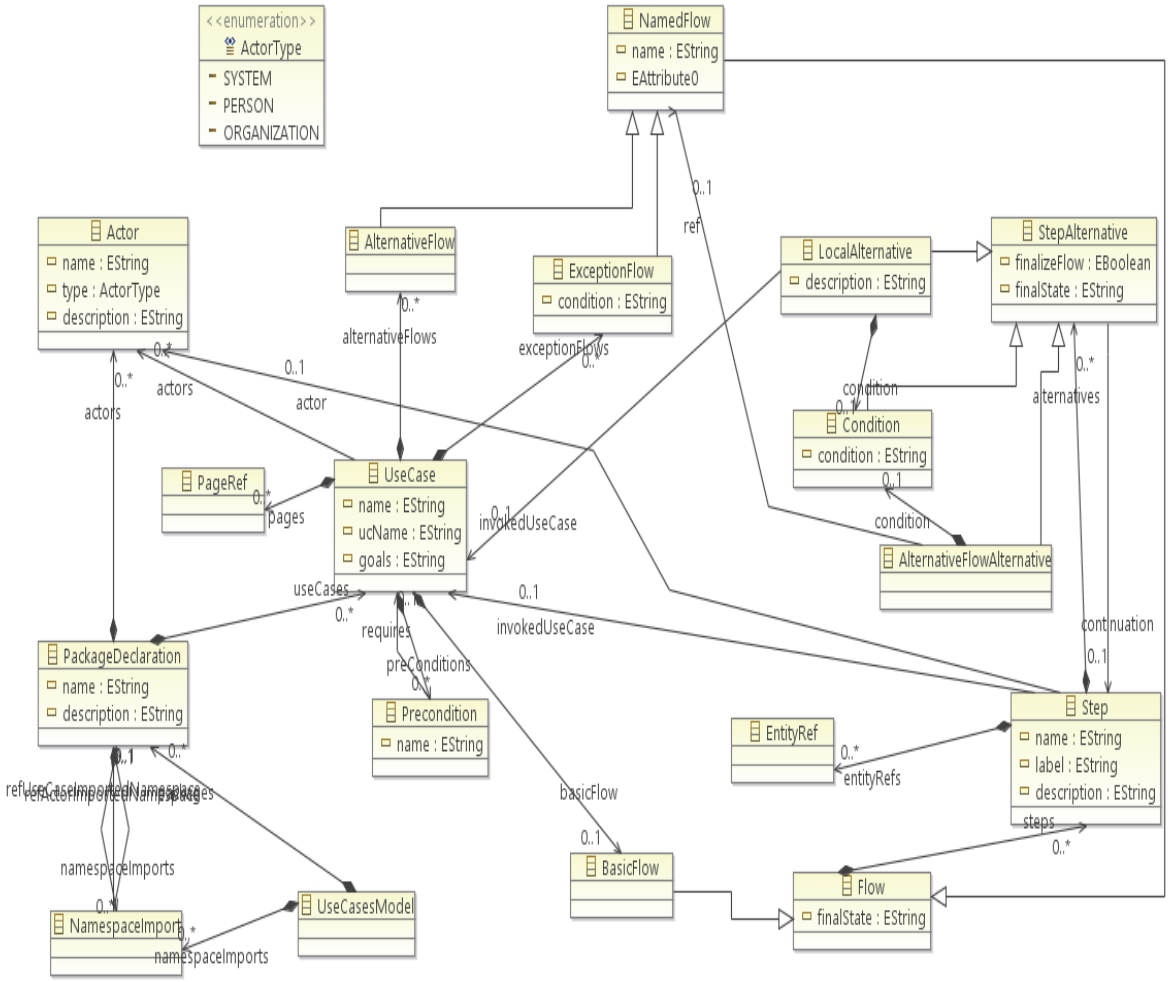


Figura 3.10 - Modelo referente aos use cases

3.4 Derivação de Casos de Teste

3.4.1 Mapeamento de Casos de Uso para a DSL de Testes

A tabela 1 tem como objetivo ajudar a compreender como foi feito o processo de mapeamento que teve de ser realizado no desenvolvimento do trabalho. Assim, classes e atributos especificados no modelo de casos de uso vão dar origem a outros atributos da DSL de testes que contem os cenários de teste correspondentes a cada caso de uso analisado.

Tabela 1 - Mapeamento de casos de uso para a FeatureTest

FeatureTest	Use Case
Package name	Package name
Feature ID: caso haja mais que um usecase é adicionado o id desse usecase	UseCase ID
FeatureName	UseCaseName
FeatureDescription	Goals do UseCase
ScenarioName	UseCaseName com a incrementação de um algarismo a cada cenário diferente que é gerado.
ScenarioDescription	Goals do UseCase
Assuming	PreCondition: sendo que aqui terá de ser adicionado qual o useCase que é requerido como préCondition, sendo a descrição de esse UC referente ao assuming na feature.
then	Índice 1 da Label do Step do BasicFlow ou da alternativa
and	Índice 1 da Label do Step do BasicFlow ou da alternativa
finally	Índice 1 da Label do último Step do BasicFlow ou da alternativa
at	Índice 2 da Label do Step com a adição do prefixo correspondente à ação

Na empresa existem algumas regras usadas internamente a quando da especificação de casos de uso. Assim o mapeamento foi feito de forma a respeitar as regras existentes. Assim a quando da definição de um step na sua label este deve conter três informações separadas entre elas por ":". A primeira informação da label é um termo definido pela empresa e que permitem identificar o tipo de ação daquele passo, segunda informação é a descrição do passo e a terceira é uma introdução feita para a geração da Feature DSL que indica qual o widget da página usado naquele step. A quando da introdução de um screen no step correspondente é adicionado de forma automática ao caso de uso as páginas relativas a cada screen.

3.4.2 Descrição dos elementos gerados na DSL de Testes

A **feature** diz respeito ao caso de uso para o qual os cenários de teste são gerados. Tem o nome da feature e uma pequena descrição.

O **url** da feature é o url da página onde o teste é iniciado.

A **description** é onde é indicado a finalidade do caso de uso que vai ser testado.

Nos cenários que são gerados a partir do caso de uso esses "**scenario's**" contêm um nome, uma descrição do cenário em questão, o **assuming** que consiste nas pré-condições do caso de uso, o **then**, os **and's** e o **finally** são os passos que vão ser testados sendo que o é gerado à frente da instrução **at** é o tipo de acção que diz respeito aquele passo.

Em relação ao steps gerados na feature o significado destes é o seguinte:

- Os steps contêm um nome que identifica o step e uma descrição. Através do nome do step, como este no seu início, já identifica o tipo de acção, como set para texto, **click** para um botão e **validate** para fazer uma validação.
- Na expressão **var** que é especificada quando o nome do step tem o prefixo set é feita uma pesquisa numa tabela contida num ficheiro CSV e procura nessa tabela se aquele atributo existe. A instrução **check** faz uma verificação, em uma DSL já existente, onde estão definidos os tipos de elementos e o retorno que cada um desses elementos tem. A operação assign é uma operação de atribuição, onde é inserido o valor do **assign** no atributo que contem o identificador descrito na instrução **id**.

- Quando um step consiste numa ação de **click**, o processo é o mesmo da instrução **assign** descrita anteriormente, ou seja, é atribuída a ação de um botão ao elemento que for identificado.
- Quando um step contem a operação **validate**, esta operação consiste em verificar se a ação identificada nesse step consegue chegar à página de destino. No caso do exemplo, essa validação era feita se depois da ação realizada se chegasse à página pessoal do utilizador.

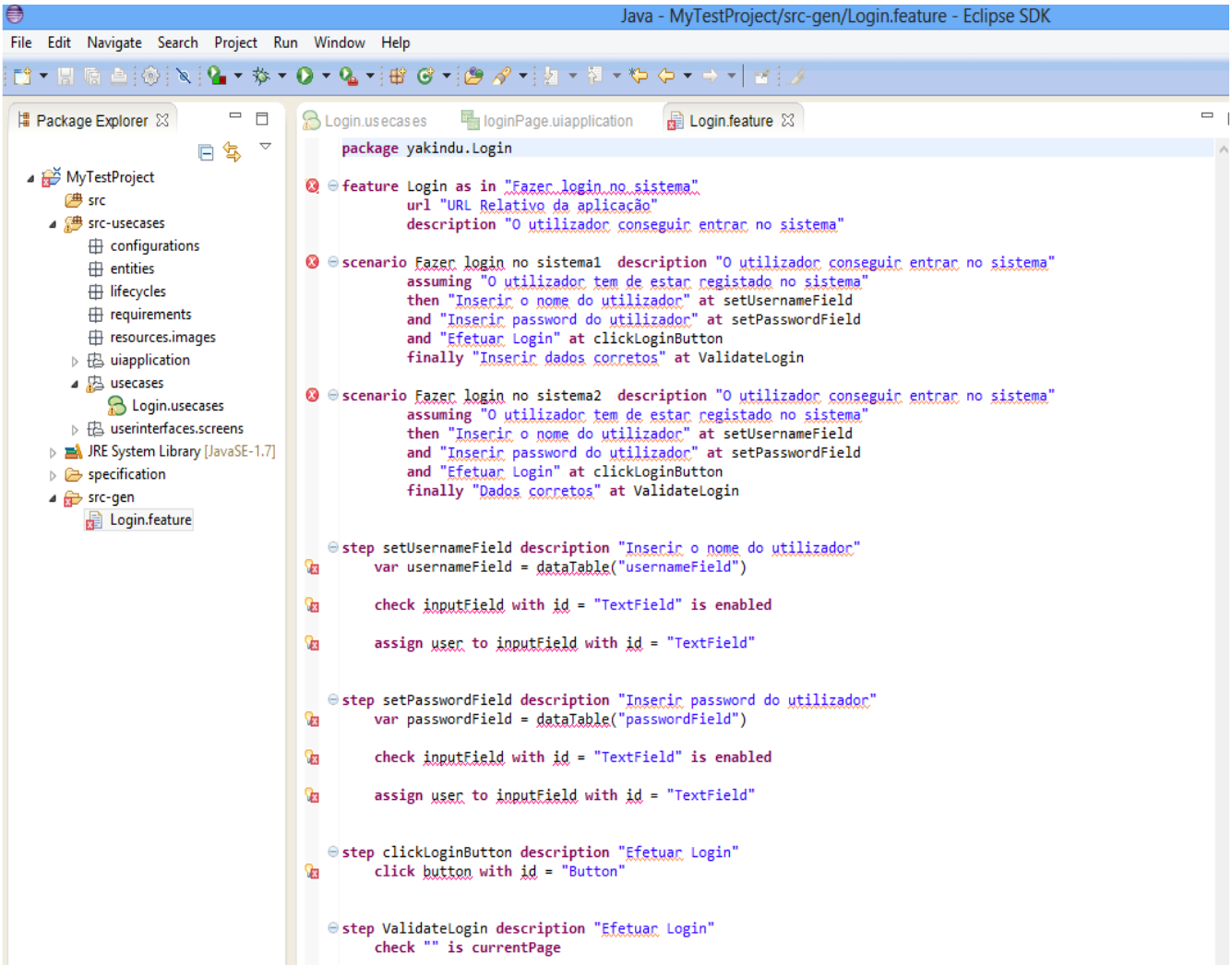


Figura 3.11 - Exemplo da Feature de teste gerada

3.5 Conclusão

Com a realização do estudo das ferramentas e tecnologias existentes para a realização do trabalho de geração de cenários de teste a partir de requisitos de uma forma automática, sendo que como este projeto foi realizado em âmbito profissional, existiram ferramentas e abordagens que tiveram de ser utilizadas.

Na área da especificação de requisitos a abordagem UML e a utilização de casos de uso era já a utilizada na organização, sendo foi feita uma análise da forma como os requisitos eram definidos e de forma a ir ao encontro do pretendido houve alterações que tiveram de ser feitas nessa definição. Quanto à geração da DSL de testes, e com recurso as técnicas de desenvolvimento orientado a modelos, fez-se com que através do modelo de uma linguagem se conseguisse gerar outra linguagem que utiliza propriedades da primeira.

Quando se fala em automatização, tem-se como objetivo que algo seja efetuado sem contributo humano, ou que seja feito de uma forma autónoma. Para a geração de cenários de teste ser realizada de forma automática houve a necessidade de ser criado um gerador com recurso à tecnologia Xtend, sendo esta etapa explicada de uma forma mais abrangente no capítulo seguinte.

Em suma, efetuou-se neste capítulo uma análise sobre as ferramentas utilizadas para o desenvolvimento do trabalho bem como técnicas de desenvolvimento em projetos de software. Como o objetivo do trabalho engloba a área de testes e de requisitos foram estudadas também as ferramentas e normas utilizadas na organização para a especificação de requisitos e dos testes.

4 Geração Automática de Casos de Teste

4.1 Introdução

No seguinte capítulo será apresentado o estudo de caso realizado no departamento de Research & Design da empresa i2S – S.A., que tem como objetivo a geração de casos de casos de teste de uma forma automática a partir de requisitos já especificados.

Para a realização deste estudo de caso foi necessário fazer uma análise sobre as ferramentas e tecnologias usadas na organização, quer ao nível da especificação de requisitos quer também ao nível de tecnologias que a empresa decidiu, com base no seu processo de desenvolvimento, que deveriam ser usadas para a realização do estudo.

Com base e de acordo com o que foi definido anteriormente foi desenvolvido um protótipo que com recurso a tecnologias como DSLs (Linguagens Especificas de Domínio) e abordagens de desenvolvimento orientada a modelos (MDD – *Model-Driven Development*) se conseguisse a criação de uma DSL de testes de forma automática que identifica quais os casos de teste a serem realizados para um determinado caso de uso.

4.2 DSL, Ecore e EMF

Como foi discutido no capítulo 3, para se fazer a derivação de casos de teste a partir de requisitos, é necessário não só compreender e mapear a especificação dos requisitos, neste caso essa especificação é feita a partir de casos de uso, e compreender a forma como essa derivação vai ser feita e o que vai gerar, mas é também necessário analisar tecnologias e ferramentas que permitam ir de encontro ao objetivo.

Como a realização deste trabalho segue uma abordagem de desenvolvimento orientada a modelos e ao uso de linguagens específicas de domínio (DSL), foi necessário o estudo das tecnologias Xtend e Xtext em relação às DSLs e as tecnologias EMF e Ecore em relação aos

modelos. De salientar que todas estas tecnologias são integradas no ambiente de desenvolvimento Eclipse Framework.

4.2.1 DSL de Testes

Quando se tem como objetivo a criação de uma linguagem para um domínio específico é necessário recorrer a tecnologias e ferramentas que permitam essa criação. Para a criação da DSL de testes usada neste trabalho foram usadas duas tecnologias que estão integradas no ambiente de desenvolvimento Eclipse. Essas duas tecnologias são o Xtend e o Xtext, sendo explicado a seguir quais as características e potencialidades que essas tecnologias têm e como funcionam.

Xtend

Xtend é uma linguagem flexível e expressiva proveniente do Java. Esta linguagem de programação em questões de sintaxe e semântica tem os princípios da linguagem Java mas com aspetos melhorados (Bettini, 2013). Esta linguagem apresenta funcionalidades adicionais e integra ainda recursos de programação funcional. É compilada a partir de código Java permitindo assim a integração de todas as bibliotecas Java. A sua linguagem e o seu ambiente de desenvolvimento fazem parte do ambiente Eclipse.

As vantagens que a linguagem Xtend produz em relação ao Java são:

Extensão de métodos – a extensão de métodos permite adicionar novos métodos, sem modificar os métodos já existentes.

Expressões lambda – permitem que se tenha uma sintaxe mais concisa para funções literais.

Anotações ativas – estas permitem que os desenvolvedores participem no processo de tradução de código Xtend para código Java com recurso a livrarias, o que se torna útil quando é requerido pelo Java a escrita manual de padrões.

Sobrecarga de operadores – permite fazer bibliotecas mais expressivas, existindo operadores que não são fixos, esses operadores não estão limitados às operações em certos tipos. Assim é

permitido redefinir os operadores para qualquer tipo aplicando um método de assinatura correspondente.

Expressões Switch – a expressão switch em Xtend é diferente da usada em Java, pois não há nenhuma regra em que seja permitido apenas avaliar-se um caso no máximo e o switch pode ser usado para qualquer referência e não apenas para algumas.

Dispatch múltiplo – normalmente, a resolução de um método é feita estaticamente quando é feita a compilação e as chamadas ao método são feitas através de argumentos estáticos, no contexto de extensão de métodos esta abordagem não consegue dar uma resposta, o facto de Xtend permitir que se faça dispatch múltiplo permite que haja um comportamento polimórfico que dá resposta a essas situações.

Expressões de modelo – modelos permitem uma concatenação de String legível. As expressões de modelo permitem que se possa ter uma abrangência a várias linhas e expressões podem ser alojadas para depois serem avaliadas.

Não há declarações – na linguagem Xtend tudo é uma expressão que tem um tipo de retorno. Como não existem demonstrações a escrita de código é feita de maneira mais interessante.

Na linguagem Xtend é fornecido um largo leque de opções para tornar os métodos de extensão disponíveis, conforme o que foi dito anteriormente. Xtend fornece ainda uma biblioteca com varias classes utilitárias e métodos estáticos. Estes métodos estáticos são disponibilizados automaticamente em código Xtend para poderem ser utilizados como métodos de extensão. As classes utilitárias visam melhorar as funcionalidades de tipos padrão e de coleções.

Xtext

Xtext é uma *framework* que permite a criação de DSLs. Esta *framework* é baseada numa abordagem recursiva. Xtext faz parte do ambiente de desenvolvimento integrado (IDE) Eclipse e com recurso a uma gramática enriquecida gera um editor baseado neste IDE, um *parser* e árvore de sintaxe abstrata com base em EMF Ecore. Fornece ainda API's e linguagens para definir os objetivos, restrições e geradores e ainda aspetos adicionais com o ambiente de desenvolvimento que não podem ser originados automaticamente a partir da gramática definida (Bettini, 2013). Para se verificar se um tipo é válido, este tem de ser calculado. Em Xtext as restrições, transformações e geradores são implementados, por defeito, em Java, ou então em

Xtend, que é uma linguagem muito parecida com Java que faz parte do Xtext, como explicado anteriormente. Esta *framework* não fornece suporte para a definição de um tipo de sistema (*collection*), apenas permite a implementação de regras escritas em Xtend como parte da validação. Como as linguagens se estão a tornar mais complexas é necessário criar suporte para a implementação de um tipo de sistema (*collection*) para que os utilizadores tenham regras de escrita concisas. Assim sendo, este tipo de *freamwork* deve dar resposta de forma a serem expressas regras de forma concisa, suportar casos comuns de regras definidas, mas não impedir o utilizador de implementar casos mais sofisticados manualmente, reportar especificações que estejam incompletas ou que se referem a conceitos que não existem. Deve existir integrabilidade com a *framework* de validação da linguagem desenvolvida para que os erros de digitação possam ser apresentados junto com os erros de restrições e por último suportar a fase de teste e de *debugging* das regras de escrita.

Esta *framework* permite a implementação de linguagens específicas de domínio (DSL's) de uma forma rápida cobrindo completamente todos os aspetos de infraestrutura de uma linguagem, desde a implementação, passando pelo gerador de código até chegar ao interpretador.

Uma das coisas mais surpreendentes do Xtext é que para se começar uma implementação de uma DSL, é apenas necessário fazer uma especificação de gramática semelhante ao ANTLR. Não sendo assim necessária nenhuma preocupação com regras de construção da AST. A criação AST é feita de forma automática, assim como o analisador lexical.

Apesar de a *framework* Xtext trazer todas estas implementações padrão, e que normalmente são as necessárias para atender às necessidades, estas implementações podem ser personalizadas pelo programador.

4.2.2 EMF (Eclipse Modeling Framework)

O EMF é uma *framework* de modelação e geração de código para a construção de ferramentas e outras aplicações baseadas em um modelo de dados estruturado. A partir de uma especificação do modelo descrito no XMI, o EMF fornece ferramentas e suporte de execução para a produção de um conjunto de classes Java para o modelo, um conjunto de classes de adaptadores que permite a visualização e edição do modelo com base em comandos e ainda um editor. Os

modelos podem ser especificados usando anotações Java, documentação XML ou ferramentas de modelação. Mais importante que tudo o EMF fornece a base para a interoperabilidade com outros ferramentas e aplicações que também sejam baseadas em EMF (Gronback, 2013).

O EMF é uma tecnologia que compõem a plataforma Eclipse e permite a manipulação de modelos com base nos meta-modelos correspondentes. O meta-modelo em que o EMF se baseia, e que será aprofundado mais à frente, é o meta-modelo Ecore. O EMF utiliza um subconjunto de mapeamentos do meta-modelo para Java que foram otimizados para manipulação de memória, o que torna, a tecnologia mais eficiente e mais simples de ser utilizada.

No desenvolvimento orientado a modelos (MDD), o *Eclipse Modeling Framework* (EMF) está atualmente a tornar-se uma referência. Com recurso a esta *framework* consegue-se descrever modelos de classes e gerar código java que suporta a criação, alteração, armazenamento e carregamento de instâncias de um modelo. Tendo em conta isto, percebe-se que o EMF faz uma unificação das tecnologias Java, XML, UML que permite no caso de se definir uma transformação a partir da abordagem EMF, essa transformação seja aplicável a outras tecnologias.

O EMF proporciona uma modelação e uma geração de código para aplicações Eclipse que sejam baseadas em modelos de dados estruturados. A *framework* Eclipse EMF pode ser usada para modelar um modelo de domínio, fazendo com que haja uma distinção entre o meta-modelo e o modelo real. O meta-modelo descreve a estrutura do modelo. Um modelo é, então, o exemplo deste meta-modelo. EMF fornece uma estrutura *pluggable* para armazenar as informações do modelo, já o padrão utiliza XMI para persistir a definição do modelo.

Como já dito anteriormente, o EMF permite criar meta-modelos através de diferentes meios, como XMI, anotações Java e XML. A descrição feita a seguir vai mostrar como com o uso de ferramentas EMF se consegue criar um modelo EMF.

Uma vez que um meta-modelo EMF é especificado pode-se gerar as classes de implementação de Java correspondentes ao modelo. O EMF fornece a possibilidade de estender o código gerado de uma forma mais segura. Consegue-se um modelo explícito do domínio ajudando a proporcionar uma visibilidade clara do modelo. Funcionalidades de notificação no caso de o modelo ser alterado também compõem o EMF, bem como, geração de interfaces para criar

objetos, ajudando assim a manter uma aplicação limpa das classes implementadas individualmente. Outra das vantagens é o facto de se poder gerar código Java a partir do modelo em qualquer ponto do tempo.

As três partes fundamentais que compõem o EMF são:

Ecore – é um meta-modelo que é o núcleo da *framework* EMF. Este meta-modelo permite descrever e suportar modelos na sua execução, incluindo notificações de alteração, dar suporte a persistências com padrão XMI e uma API muito eficiente para a manipulação de objetos EMF genéricos.

EMF.Edit – é uma *framework* que inclui classes genéricas reutilizáveis para a construção de editores para modelos EMF. Esta *framework* fornece:

- Classes de conteúdos e fornecedores de etiquetas e suporta as propriedades fonte e outras classes de conveniência que permitem a exibição dos modelos EMF usando uma área de trabalho padrão (JFace), vistas e folhas de propriedades.

- Uma *framework* de comando que inclui um conjunto de classes para implementação de um comando genérico para a construção de editores que suportam o desfazer e refazer de forma totalmente automática.

EMF.Codegen – uma das grandes vantagens da utilização da *framework* EMF é a facilidade de geração de código com tudo o que é necessário para construir um editor completo para um modelo EMF. Este inclui uma interface gráfica onde podem ser especificadas as opções de geração e quais os geradores a serem invocados. A parte da geração aproveita o componente JDT (*Java Devolpment Tooling*) do Eclipse.

4.2.3 Ecore

A *framework* EMF é composta por um meta modelo (Ecore) para os modelos que são descritos suportando esses modelos em termos de tempo de execução, inclui notificações de alteração,

suporte à persistência com padrão XMI de serialização e ainda disponibiliza uma API que pode ser usada de uma forma muito eficiente para manipulação de objetos EMF genéricos.

A afirmação feita anteriormente que o EMF é composto por um meta-modelo não é completa, pois na realidade a *framework* EMF é composta por dois meta-modelos, um é o modelo Ecore e outro é o modelo Genmodel. O meta-modelo Ecore contem as informações sobre as classes que estão definidas, já o meta-modelo Genmodel contem informações adicionais como CodeGeneration, que por exemplo contem as informações relativas aos caminhos e arquivos. O meta-modelo Genmodel contem ainda parâmetros de controlo sobre como o código deve ser gerado.
A principal caraterística que o meta-modelo Ecore adiciona é que este modelo permite definir elementos diferentes:

EPackage: é uma componente do Ecore que permite a organização de classes e de tipo de dados.

EClass: representa uma classe, que tem zero ou mais atributos e zero ou mais referências;

EAttribute: representa um atributo que tem um nome e um tipo;

EReference: representa uma extremidade de uma associação entre duas classes. Contem uma *flag* que indica se existe uma contenção e uma classe de referência para a qual aponta.

EDataType: representa qual o tipo do atributo.

A composição do modele Ecore contem um objeto raiz que representa todo o modelo. Este modelo tem filhos que representam os packages, cujos filhos representam as classes, enquanto os filhos das classes representam os atributos dessas mesmas classes.

Componentes Ecore

Como explicado anteriormente o Ecore é um meta-modelo do EMF, sendo este meta-modelo bastante flexível, podendo ser gerados modelos Ecore de acordo com o modelo que um utilizador esteja e dependendo da camada arquitetural onde este está inserido.

Os componentes Ecore estão relacionados de acordo com esta hierarquia representada na figura 4.1. Neste meta-modelo o EObject é a raiz do modelo, sendo todos os outros componentes filhos deste, significando que todos os outros componentes herdam as propriedades do tipo EObject.

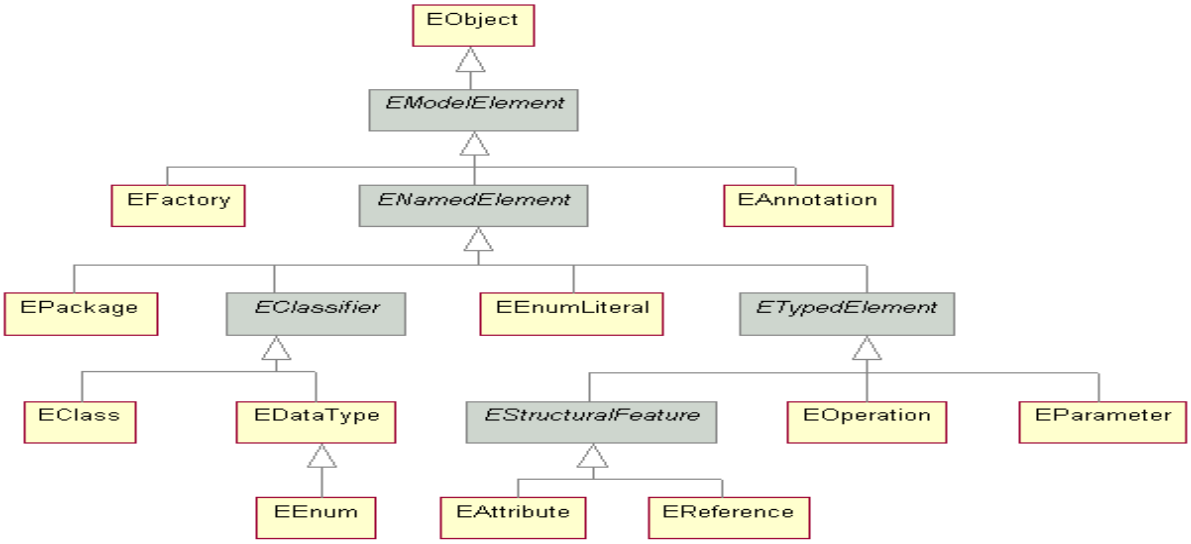


Figura 4.1 - Componentes do meta-modelo Ecore

O Ecore consiste numa linguagem que permite a construção de outras linguagens e que tem conceitos como class, existindo assim herança e propriedades. As propriedades contêm atributos que podem ser definidos tais como nome, multiplicidade e o tipo. O meta-modelo ecore não é mais do que uma versão simplificada de uma modelação por classes em UML. Na

figura 4.2 demonstra-se as relações, atributos e operações de todos os componentes pertencentes ao Ecore.

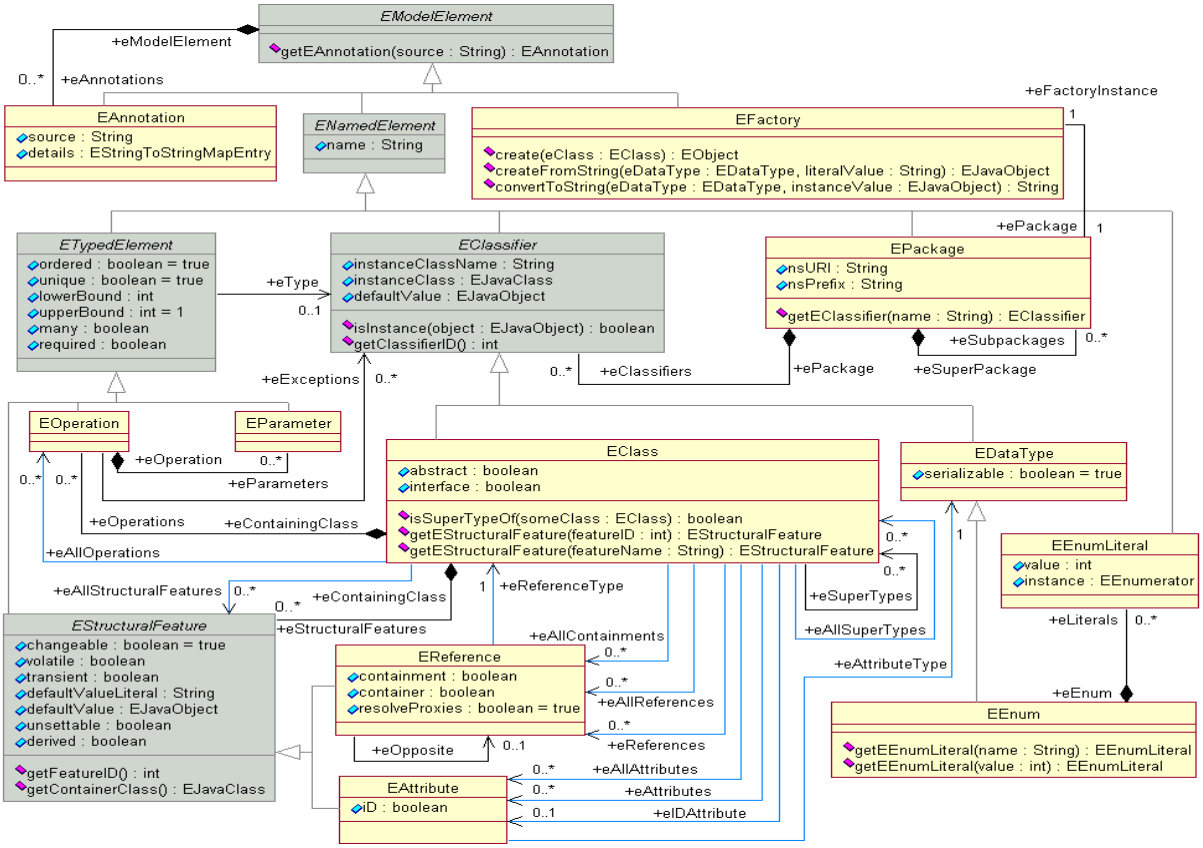


Figura 4.2 - Relacionamento entre os componentes do meta-modelo Ecore

4.3 Carregamento do modelo de requisitos

Nesta secção será explicada a implementação da abordagem de desenvolvimento orientada a modelos (MDD), introduzida na secção 3.3 deste documento, e como o Eclipse Modeling Framework (EMF) auxiliam essa tarefa. A presente secção pretende detalhar todos os aspetos realizados para conseguir carregar o modelo de casos de uso usado na organização para a especificação de requisitos e como esse modelo é transformado em um modelo de classes, de

forma a conseguir-se explorar esse modelo e utilizar as informações necessárias para a criação da DSL de testes.

Inicialmente é apresentada a forma como se consegue carregar o modelo de casos de uso a partir de um ficheiro que contenha a extensão usecases e posteriormente é mostrado como com a ajuda do Ecore do EMF se consegue transformar esse modelo numa estrutura de classes.

4.3.1 Carregamento do modelo de especificação de requisitos

Para a geração da Test Feature que identifica de forma automática quais os casos de teste a serem realizados para cada caso de uso, o primeiro passo a ser realizado consiste no carregamento do conteúdo do modelo que diz respeito à especificação de requisitos para uma estrutura de classes.

O carregamento deste Modelo é realizado com recurso da tecnologia EMF (*Eclipse Modelimg Framework*).

Este carregamento dos modelos é feito através de um *resourceSet*, que a partir de um diretório carrega o modelo de todos os ficheiros com a extensão .usecases e .uiapplication.

```
public static void loadResources(IFolder sourceFolder) {
    try {
        for (IResource sourceResource : sourceFolder.members()) {
            if (sourceResource.getType() == IFolder.FILE) {
                if (sourceResource.getFileExtension().equals(
                        "uiapplication")
                        || sourceResource.getFileExtension().equals(
                                "usecases")) {
                    Resource resource = getResourceSet().getResource(
                            URI.createFileURI(sourceResource.getFullPath()
                                    .toString()), true);
                    getResourceSet().getResources().add(resource);
                }
            } else {
                loadResources(sourceFolder.getFolder(sourceResource
                        .getName()));
            }
        }
    } catch (CoreException e) {
        e.printStackTrace();
    }
}
```

Figura 4.35 - Método responsável pelo carregamento dos ficheiros

Os modelos deste tipo de ficheiros são adicionados ao mesmo *resourceSet*, conseguindo-se obter o conteúdo dos modelos de cada ficheiro. O modelo de casos de uso (.usecases) e das páginas (.uiapplication) são dois modelos diferentes, mas que têm dependências entre si.

```
private static XtextResourceSet resourceSet;

public static XtextResourceSet getResourceSet() {
    if (resourceSet == null) {
        Injector injector = new UseCasesStandaloneSetup()
                .createInjectorAndDoEMFRegistration();
        resourceSet = injector.getInstance(XtextResourceSet.class);
        resourceSet.addLoadOption(XtextResource.OPTION_RESOLVE_ALL,
                Boolean.TRUE);
    }
    return resourceSet;

}
```

Figura 4.4 - Método que permite a "transformação" do conteúdo do ficheiro para um modelo

O método exposto na figura 4.5 mostra como é conseguido o carregamento do modelo de casos de uso.

```
public static UseCasesModel loadModel(IResource sourceResource) {

    UseCasesModel result = null;

    try {
        Resource resource = getResourceSet().getResource(
                URI.createFileURI(sourceResource.getFullPath().toString()),
                true);
        result = (UseCasesModel) resource.getContents().get(0);

    } catch (Exception e) {
        e.printStackTrace();
    }

    return result;

}
```

Figura 4.5 - Método responsável pelo carregamento do modelo de casos de uso

Estes modelos são carregados com base na tecnologia Ecore da *framework* EMF e são transformados em uma estrutura de classes com as suas dependências e atributos. Podendo-se passar a utilizar o modelo como uma estrutura de classes.

Através do *Eclipse Modeling Framework* SDK, depois de selecionado um ficheiro que contenha a especificação dos casos de uso da ferramenta Yakindu Requirements, consegue-se com o auxílio do Ecore a criação do modelo com base em diagramas Ecore.

Depois de se ter os diagramas do modelo no tipo .ecore o próximo passo é gerar o modelo, com o auxilio do genmodel. Depois disso tem-se o modelo pretendido. Na verdade são criados dois tipos de modelo, um do tipo .ecore e outro do tipo .genmodel.

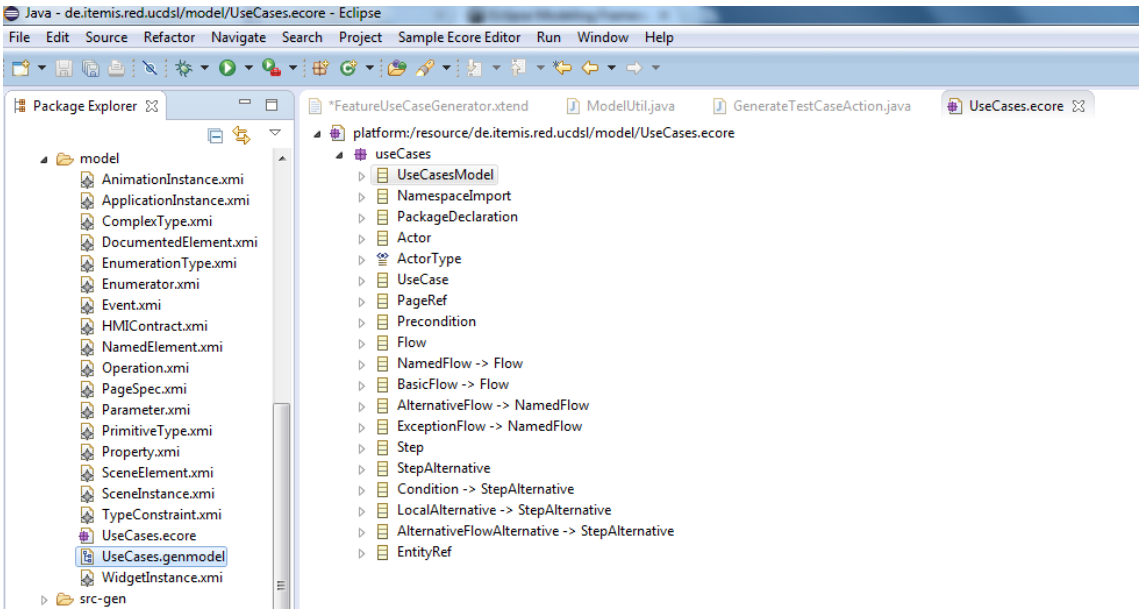


Figura 4.6 - Modelo de classes do tipo ecore

O passo seguinte consiste em gerar as classes do modelo. Para isso seleciona-se a raiz do modelo que contem a extensão .genmodel e com recurso da *framework* EMF gera-se as classes Java do modelo.

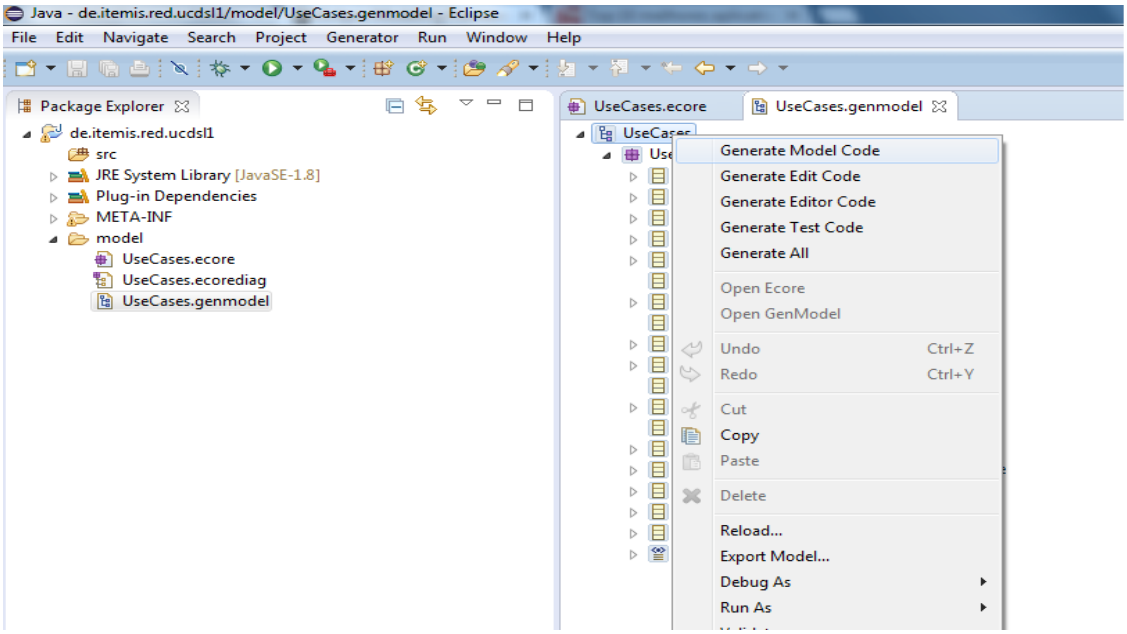


Figura 4.7 - Geração de classe Java a partir de um modelo ecore

Como demostrado na figura a baixo na pasta usecases que faz parte da pasta src são geradas as classes Java do modelo e também as classes de implementação, passando assim de um modelo de classes para uma estrutura de classes em Java que podem ser acedidas, alteradas e utilizadas com o proveito necessário para o objetivo a atingir.

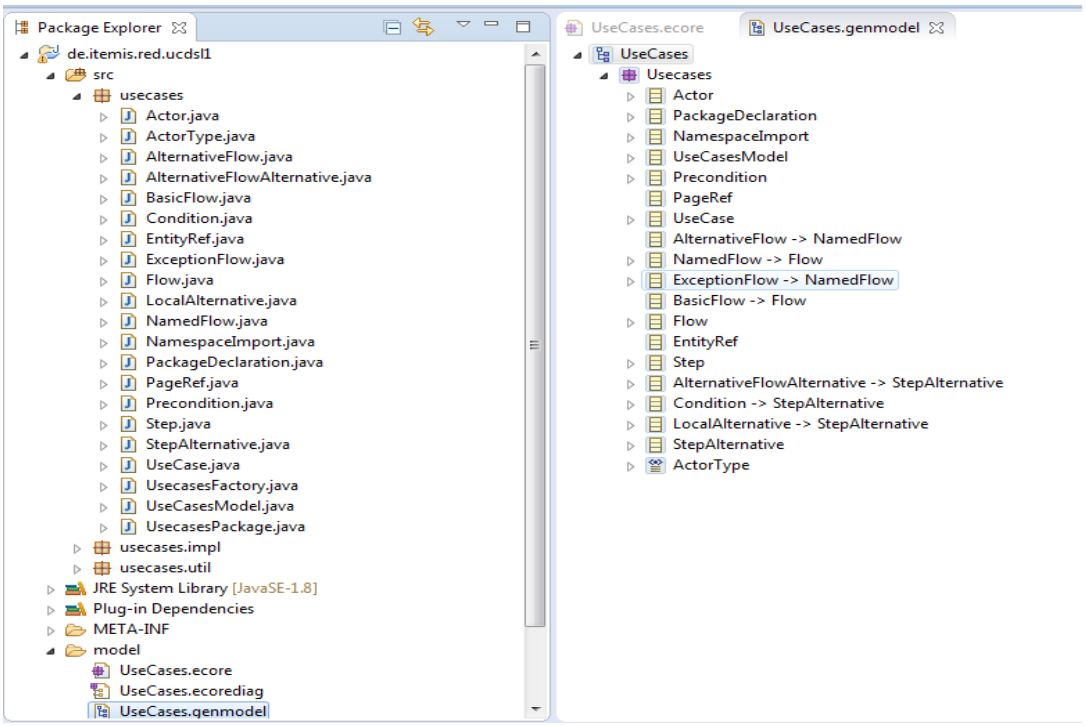


Figura 4.8 - Modelo de classes Java gerado

4.3.2 Algoritmo para a criação da Test Feature

Foi criado um algoritmo para a criação da DSL Test Feature. O que este algoritmo faz, basicamente, é percorrer todo o caso de uso e à medida que encontra as caraterísticas do caso de uso guarda essa informação na forma estruturada das classes Java. Depois de essa informação

estar guardada é feito um mapeamento com as propriedades que a Test Feature que representa os casos de teste se pretende que contenha.

4.4 Geração de casos de teste

Na secção seguinte é apresentada a ultima etapa prática do trabalho. Depois de se ter o modelo de casos de uso carregado para uma estrutura de classes, que possa ser acedida e utilizada e depois de estar feito o mapeamento entre a informação retirada do modelo de casos de uso para a *feature* de testes que se pretende gerar, como foi explicado na secção anterior, falta apenas criar a estrutura que essa *feature* deve ter e a forma como é gerada.

4.4.1 Criação de Templates para identificação dos casos de teste

Após a estrutura de classes dos casos de uso estar carregada para uma estrutura de classes e feito o mapeamento entre os atributos dos casos de uso e das páginas de forma a dar origem à Test Feature onde são identificados os cenários de teste para um determinado caso de uso, falta definir qual a estrutura que a Test Feature deve ter. Assim com recurso à tecnologia Xtend são criados os *templates* que serão gerados e que estruturam a *feature* de acordo com o que é pretendido. A criação desses *templates* é mostrada nas figuras a baixo.

Como dito anteriormente a criação da estrutura da feature é feita através de templates criados em Xtend, sendo que essa estrutura consiste em vários templates diferentes dentro de um

template geral. Assim, são criados vários tipos de templates, sendo que todos farão parte de um único template, como representado na figura 4.9.

```
def generateFeature(PackageDeclaration pd, UseCase useCase, String url, Step step) '''
    «generatePackageName(pd)»

    «generateFeatureCode(useCase, url)»

    «generateScenarios(useCase)»

    «stepsFeatures.map[generateStepFeature].join»
'''
```

Figura 4.9 - Template da feature de testes

A função representada na figura 4.10 mostra de que forma é criado o *template* que diz respeito ao PackageName. Num *template* o que está dentro de (‘’’) é aquilo que aparece na *feature* enquanto que o que esta entre << >> são os atributos que se pretende de cada objeto.

```
def generatePackageName(PackageDeclaration pd) '''
    package «pd.name»
'''
```

Figura 4.10 - Template do PackageName

Na figura 4.11 está representado o template que define o cabeçalho da feature. Para obter esta informação é necessária a invocação de dois objetos, o objeto UseCase e o objeto String. O objeto UseCase é necessário para retirar a informação sobre o nome do caso de uso, a sua identificação e o objetivo. O objeto String que é um objeto nativo da programação por objetos

e que não precisa de ser criado pelo utilizador permite a manipulação de um conjunto de caracteres.

```
def generateFeatureCode(UseCase uc, String url) '''
    feature «uc.name» as in "«uc.ucName»"
            url "«"URL Relativo da aplicação"»"
            description "«uc.goals»"
'''
```

Figura 4.11 - Template que define a feature

Para a criação de cenários, visto que cada caso de uso pode ter um ou mais cenários, é necessário criar um método que faça essa gestão. Para se conseguir essa gestão, é necessária a criação de uma nove classe com o nome de Scenario, onde são apenas especificados os atributos que um cenário contem. Depois de criada essa classe o método de geração de cenários invoca os

atributos necessários da classe UseCase e atribui aos vários cenários existentes as propriedades correspondentes. O método representado na figura 4.12 é o responsável por essa gestão.

```
def generateScenarios(UseCase uc) {

    var scneariosStr = ""

    var newScenario = new Scenario()
    newScenario.assuming = uc.preConditions.get(0).name
    newScenario.descriptionScenario = uc.goals
    newScenario.nameScenario = uc.ucName

    getScenarios(uc.basicFlow.steps, newScenario, null, uc)
    for (Scenario scenario : scenarios) {
        scneariosStr = scneariosStr.concat(
            generateScenarioCode(scenario.nameScenario, scenario.descriptionScenario, scenario.assuming,
                scenario.condition, scenario.finallyScenario).toString)
    }
    return scneariosStr
}
```

Figura 4.12 - Método que faz a gestão dos vários cenários

Na figura 4.13 é representado o método responsável pela criação do template que diz respeito à identificação do ou dos vários cenários existentes.

```
def generateScenarioCode(String nameScenario, String descriptionScenario, String assuming, String condition,
    String finallyScenario) '''
    scenario «nameScenario»  description "«descriptionScenario»"
            assuming "«assuming»"
            then "«condition»
            finally "«finallyScenario»

'''
```

Figura 4.13 - Template que define o Cenário

4.4.2 Gerador de Cenários de Teste

A última etapa do processo de geração de casos de teste a partir de casos de uso é a criação de um gerador para que esse processo seja realizado de uma forma automática. O que este gerador faz é utilizar o modelo carregado, o algoritmo de mapeamento de casos de uso para a Test Feature e os *templates* criados para que esta seja gerada quando essa ação for pretendida.

Foi criada uma action na *framework* Eclipse para que depois de escolhida uma pasta que contenha pelo menos um ficheiro do tipo .usecases e depois de selecionada a pasta executa-se essa ação para que o processo de geração automática seja realizado.

```
public void run(IAction action) {
    try {
        IProject project = getSelectedProject();
        sourceFolder = getSourceFolder(project);
        targetFolder = getOrCreateTargetFolder(project);

        System.out
                .println("Pasta de Origem: " + sourceFolder.getFullPath());
        System.out.println("Pasta de Destino: "
                + targetFolder.getFullPath());

        ModelUtil.loadResources(sourceFolder);
        IFolder sourceUsecasesFolder = sourceFolder.getFolder("usecases");

        generateAllUsecasesToFeatureDSL(sourceUsecasesFolder, targetFolder);

    } catch (CoreException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    System.out.println("Saiu");
}
```

Figura 4.14 - Método "gerador" da action definida

Depois de gerada a *feature* de testes, está será executada numa *framework* chamada *Selenium*. Será nesta *framework* que serão executados posteriormente os casos de teste que são identificados na feature que é gerada.

Em relação à *framework*, o Selenium é uma *framework* de testes para *software*. Esta *framework* permite gravar e reproduzir testes sem que seja necessário a criação de scripts. Fornece uma DSL de testes que permite escrever testes em vários tipos de linguagens de programação diferentes. Com o Selenium a execução dos testes é feita a partir dos *browsers web*.

O ambiente de desenvolvimento do Selenium permite a criação de scripts manualmente de modo a ser fornecido um suporte complementar. O Selenese, que é a linguagem de criação de

scripts no Selenium, fornece um conjunto de comandos que são executados em um *browser* e recuperam dados das páginas resultantes.

4.5 Conclusão

Neste capítulo foi efetuado em primeiro lugar um estudo sobre os processos de desenvolvimento da organização e quais as tecnologias utilizadas na mesma, bem como as ferramentas necessárias para a realização do caso de estudo.

Compreender a forma como os casos de uso são definidos na organização e compreender o modelo da ferramenta (Yakindu Requirements) onde estes são definidos foi fundamental para se poder fazer o mapeamento de uma forma mais correta, sendo que mesmo assim, para a correta geração de casos de teste de uma forma automática a especificação de requisitos teve de sofrer algumas alterações em relação a forma como é feito atualmente.

Concluído este capítulo, pode afirmar-se que o objetivo pretendido é alcançável sendo que para tal, a forma como os requisitos são definidos tem de sofrer pequenas alterações, ou não se consegue um mapeamento correto para a geração da DSL de Testes. Assim sendo conseguiu-se alcançar o espectável de provar que com recurso ao desenvolvimento orientado a modelos e a linguagens específicas de domínio a automatização é possível.

5 Conclusões

Neste último capítulo é encerrada a dissertação de mestrado. Apos a realização e descrição de todos os passos realizados neste trabalho faz-se neste capítulo uma descrição sobre os resultados obtidos, sobre as limitações que a realização deste projeto teve. Faz neste capítulo ainda uma pequena observação, sobre alternativas que poderiam ser realizadas e sobre o trabalho futuro que esta dissertação pode suportar.

5.1 Conclusões finais

O processo de testes de *software*, como dito anteriormente neste documento, é um dos processos mais complexos e "caros" para uma organização. Ao se darem conta deste acontecimento, cada vez mais organizações estão a apostar em formas de melhorar e reduzir os custos desse processo.

O caminho a seguir para que a qualidade dos testes seja melhor e o tempo de execução de todo o processo seja reduzido, passa pela automatização, mesmo que parcial, desse processo.

A realização deste projeto tem como objetivo a automatização da geração de cenários de teste a partir de requisitos especificados em casos de uso, é uma etapa do processo de testes que sendo feita de uma forma manual, dependendo da dimensão do projeto, pode ser bastante demorada e custosa de ser feita. Esta etapa ao ser automatizada, consegue que o tempo de identificação dos cenários de teste seja bastante reduzido e também se consegue uma identificação mais viável e menos propensa a erros, uma vez que esta etapa ao ser feita manualmente, está dependente da experiencia, traqueio e concentração do responsável por essa tarefa, que é a mesma coisa que dizer, está dependente do fator humano sendo a probabilidade de existência de erros e falhas muito mais elevada em relação a ferramentas de automatização.

Em suma, ao ser criada a ferramenta de geração de casos de teste de forma automática, não se está só a ter ganhos em relação a tempos de identificação e execução, mas também em termos de qualidade na identificação dessa etapa de testes, sendo a probabilidade de geração de casos de teste com erros ou falhas ser muito reduzida.

5.2 Limitações

Como este projeto de dissertação foi realizado no âmbito organizacional, houveram limitações que foram levantadas por isso. A principal limitação encontrada foi o facto de muitas das ferramentas utilizadas para a realização do projeto eram ferramentas já utilizadas pela organização e que já estavam implementadas no processo organizacional, não podendo estas serem alteradas.

O facto de a ferramenta de especificação de requisitos utilizada pela organização, Yakindu Requirements, que é uma ferramenta desenvolvida e suportada pela **itemis AG**, conter uma licença de utilização na organização e a qual a empresa não estava interessada em alterar, trouxe algumas limitações. Assim, e apesar de através da framework EMF conseguir-se ter acesso ao modelo utilizado pela ferramenta para a especificação de casos de uso e poder-se gerar as classes java desse modelo, não é permitido a alteração desse modelo, não se podendo assim criar classes ou criar atributos em classes que fossem necessários para dar uma resposta ao pretendido e necessário para a execução do projeto.

5.3 Trabalho futuro

O trabalho realizado no âmbito desta dissertação permitiu concluir que a automatização do processo de testes, permite não só reduzir o tempo e custos desse processo mas também garante testes de maior qualidade e menos falhas.

Foi mostrado também neste trabalho que o desenvolvimento orientado a modelos (MDD) e a criação de linguagens específicas de domínio (DSL) são duas abordagens que facilitam muito, não apenas o processo de testes, mas também podem ser uteis para outros processos de desenvolvimento de *software* e organizacionais.

Em relação ao trabalho desenvolvido, o trabalho futuro pode passar pela extensão do modelo de casos de uso de forma a conseguir-se criar classes ou adicionar atributos que responderiam às necessidades pretendidas e que melhorariam e facilitariam o processo de mapeamento da

especificação de requisitos para a *feature* de teste gerada. Caso se consiga realizar a extensão do modelo, a maior limitação descrita na alínea anterior é ultrapassada, pois consegue-se um modelo de acordo com o que é pretendido.

Referências

AGUIAR, A. (2003) - A minimalist approach to framework documentation, PhD thesis.

BACH, J. (1999) - Test automation snake oil. In 14 International Conference and Exposition on Testing Computer Software (1999).

BERNDTSSON, M. et al. (2008) - Thesis Project - A Guide for Students in Computer Scienceand Information Systems, Springer.

BETTINI, L. (2013) – Implementing Domain-Specific Languages with Xtext and Xtend, 2013.

BINDER, R. V. (1999) - Testing object-oriented systems – Models, patterns, and tools, 1999.

BOOCH, G. et al. (2005) – The Unified Modeling Language User Guide, 2nd Editon, 2005.

CHAVARRIAGA, E.; MACÍAS, J.A. (2013) - A model-driven approach to building-driven approach to building modern semantic web-based user interfaces. Advances in Engineering Software, vol. 40, n. 12, pg. 1329-1334 (2013)

CHEN, Li (2010) - A Simulation Study on Some Search Algorithms for Regression Test Case.

COOK, S.; JONES, G.; KENT, S.; WILLS, A. C. (2007) - Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional (2007).

COPLIEN, J.; SCHMIDT, D. (1995) - Pattern Languages of Program Design. Addison-Wesley, (1995).

CWALINA, K.; ABRAMS, B. (2005) - Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries. Addison-Wesley Professional.

DEURSEN, A. et al. (2000) - Domain-Specific Languages: An Annotated Bibliography. In ACM SIGPLAN Notices 35(6), pg. 26-36.

DUSTIN, E. (1999) - Lessons in test automation. In STQE – The Software Testing & Quality Engineering Magazine (1999).

FEWSTER, M.; GRAHM, D. – Software Test Automation. Addison-Wesley, (1999).

FEWSTER, M. (2001) - Common mistakes in test automation. In Proceedings of Fall Test Automation Conference (2001).

GAMMA, E. et al. (1995) - Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, (1995).

GRONBACK, R. (2013) – Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, 2013.

HEYNER, A.; MARCH, S.; PARK, J.; RAM, S. (2004) - Design science in information systems research. MIS quarterly, 28(1), pg.75-105.

HOWDEN, W. E. (1987) - Functional Program Testing and Analysis, McGraw-Hill, New York (1987).

IEEE (1998). IEEE standard for software test documentation. New York: IEEE. ISBN 0-7381-1443-X.

ITEMIS, 2014 – www.itemis.com

JACOBSON, I. et al. (2011) – Use Case 2.0: The Guide to Succeeding with Use Cases, 2011.

KANER, C. (1997) - Improving the maintainability of automated test suites. In Proceedings of the International Quality Week (1997).

KANER C.; Falk, J. and Nguyen, Q. (1999) - Testing Computer Software, second ed., John Wiley & Sons (1999), Chaps. 1, 2, 5-8, 11-13, 15.

KANSOMKEAT, S.; RIVEPIBOON, W. (2003) – Automated-generating test case using UML statechart diagrams. In Proc. SAICSIT (2003), pg. 296-300.

KIEBURTZ, D. (2001) – Retrospective on Formal Methods. In school of Science an Engineering, OHSU.

KNUTH, D. E. (1984) - Literate programming. The Computer Journal, 27(2), pg. 97–111.

KRAMER, D. (1999) - Api documentation from source code comments: a case study of javadoc. In Proceedings of the 17th annual international conference on Computer documentation, pg. 147–153.

LADD, A.; RAMMING, J. (1994) - Two application languages in software production. In USENIX Very High Level Languages Symposium Proceedings, pg. 169-178.

MAIA, C.L.B., CARMO, R.A.F., FREITAS, F.G., CAMPOS, G.A.L., SOUZA, J.T. (2009) - A Multi-Objective Approach for the Regression Test Case Selection Problem.

MARICK, B. (1999) – New Models for Test Development. Testing Foundations, 1999.

MENON, V.; PINGALI, K. (1999) - A case for source-level transformations in MATLAB. Proceedings of the second USENIX Conference on Domain-Specific Languages. USENIX Association, pg. 53-66.

MOLINARI, L. (2005) - Testes de Software Produzindo Sistemas Melhores e Mais Confiáveis. 2ª Edição. São Paulo-SP (2005).

MOLINARI, L. (2010) - Inovação e Automação de Testes de Software. 1ª Edição. São Paulo-SP (2010).

MYERS, G. (2004) - The Art of Software Testing. 2 ª ed. John Wiley (2004).

NETO, P.; RESENDE, R.; PADUA, C. (2007) - Requirements for Information Systems Model-Based Testing, in Proceeding of ACM Symposium on Applied Computing (SAC'07), Seoul, Korea.

PEFFERS, K. et al. (2008) - A design science research methodology for information systems research.Journal of Management Information Systems, 24(3), pg. 45-77.

PETERS, J. F.; PEDRYCZ, W. (2001) - Engenharia de Software: Teoria e Prática. Rio de Janeiro (2001).

POSTON, R. M. (1996) - Automating Specification-Based Software Testing, IEEE (1996).

PRESSMAN, R. S. (2002) - Engenharia de Software. 5. Ed. Rio de Janeiro: McGraw-Hill (2002).

PRESSMAN, R.S. (2006) - Engenharia de Software 6 Edição, 2006.

PRETSCHNER, A. et al. (2005) - One evaluation of model-based testing and its automation. In ICSE ’05: Proceedings of the 27th international conference on Software engineering, pages 392–401, New York, NY, USA (2005).

QUENTIN, Geoff (1999) – Automated software testing: Introduction, management and performance (1999) pg. 283-284.

SCHMIDT, D. C. (2006) – Model-Driven Engineering, 2006.

SELIC, B. (2003) – The Pragmatics of Model-Driven Development, 2003.

SHARMA, M.; MALL, R. (2009) – Automatic generating of test specifications for coverage of system state transitions. Information and software Technology (2009), pg. 418-432.

SMITH, M. (2001) - Towards modern literate programming, Project Report HONS 10/01, Department of Computer Science, University of Canterbury, Christchurch, New Zealand

SPINELLIS, Diomidis (2001) - Notable design patterns for domain specific languages. Journal of Systems and Software, 56(1), pg.91-99 (2013).

SWEBOK (2004) - Guide to the Software Engineering Body of Knowledge 2004 Version.

UTCAT (2006) - University of Texas Center for Agile Technology. In http://www.cat.utexas.edu/dsl.html

VAN DEURSEN, A.; HEERING, J.; KLINT, P. (1996) - Language Prototyping: An Algebraic Specification Approach: Vol. V. World Scientific Publishing Co., Inc. (1996).

VAN HEESCH, D. (2002) - Doxygen - a documentation system for c++, java and other languages.

VOELTER, M. (2010) - DSL Engineering – Designing, Implementing and Using Domain-Specific Languages.

WIRTH, Niklaus (1974) - On the design of programming languages. In Jack L. Rosenfeld, editor, Information Processing 74: Proceedings of IFIP Congress 74, pg. 386-393, Stockholm, Sweden.