Faculdade de Engenharia da Universidade do Porto

FEUP

Automatização do Processo de Testes de um Serviço Web e Móvel Multi-plataforma

Hugo André Gomes

Mestrado Integrado em Engenharia Informática e Computação

Orientador: João Carlos Pascoal de Faria (Professor)

28 de Junho de 2010

Automatização do Processo de Testes de um Serviço Web e Móvel Multi-plataforma

Hugo André Gomes

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo júri:

Presidente: Rui Filipe Lima Maranhão de Abreu (Professor Auxiliar Convidado)

Vogal Externo: Manuel Alcino Cunha Pereira Cunha (Professor Auxiliar)

Orientador: João Carlos Pascoal de Faria (Professor Auxiliar)

---

23 de Julho de 2010

Resumo

Actualmente, tem-se assistido ao crescimento na aposta de automatização de testes, especialmente em ambientes ágeis. No entanto, existem diversos entraves que dificultam a adopção destes processos.

O presente documento apresenta um estudo realizado na área da automatização de testes funcionais a aplicações web. Primeiramente é abordada a problemática da automatização de testes de um modo geral, seguida da particularização dos testes a nível funcional a aplicações web. A realização de testes funcionais a aplicações web encontra-se revestida de diversas particularidades que dificultam a sua correcta efectivação, nomeadamente a necessidade da execução de cenários de teste em diferentes *browsers* e plataformas, assim como a dificuldade na explicitação clara de cenários de teste, devido às características destas aplicações.

Como resultado foi construída uma plataforma de automatização de testes a aplicações web, cuja concepção e implementação se encontra devidamente documentada.

Abstract

Currently, there has been a growth in the automatization of tests, mainly in agile environments. However, there are a few obstacles that difficult the adoption of these processes.

This document presents a study in the automatization of functional tests to web applications. First, it is focused the problem of the automatization of tests in general, nexted by the particularization of tests on the functional level in web applications. The practice of functional tests in web applications is lined by several particularities that difficult its proper effectiveness, namely in the need of executing test sceneries in different browsers and platforms, as well as the difficulty in explaining clearly the test sceneries, due to the characteristics of these applications.

As a result, it has been built a automatization test platform for web applications, where the conception and implementation is well documented.

Agradecimentos

Quero expressar os mais sinceros agradecimentos ao meu orientador, o Professor João Carlos Pascoal de Faria, e à equipa da Cardmobili, especialmente ao Eng. Tiago Reis e à directora geral Helena Leite, pelo apoio, ideias e comentários ao longo de todo o trabalho realizado.

Queria também expressar os meus agradecimentos ao MIEIC, nas figuras do seu director, o Professor António Augusto de Sousa, e director do Departamento de Engenharia Informática, o Professor Raul Fernando de Almeida Moreira Vidal.

Não poderia deixar também de expressar um agradecimento muito sentido a todos os meus amigos e familiares, por todo o apoio e momentos proporcionados ao longo da realização da dissertação.

Hugo André Miranda Soares Ferreira Gomes

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

AJAX	*Asynchronous Javascript And XML*
API	*Application Programming Interface*
AUP	*Application Under Test*
BDD	*Behavior driven development*
CNL	*Controlled Natural Language*
COM	*Component Object Model*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
DP	*Driving Process*
FTDE	*Functional Test Development Environment*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
OLE	*Object Linking and Embedding*
QA	*Quality Assurance*
RC	*Remote Controller*
REST	*Representational State Transfer*
ROI	*Return On Investment*
SUP	*System Under Test*
TWA	*Test Writing Assistance*
UI	*User Interface*
WS	*Web Service*
WSDL	*Web Service Definition Language*
WWW	*World Wide Web*

Capítulo 1

Introdução

1.1 Enquadramento

A crescente difusão da Internet pela sociedade proporcionou o aparecimento de um novo mercado de aplicações de software. O desenvolvimento destas aplicações tenta acompanhar as necessidades e exigências dos seus utilizadores, proporcionando cada vez mais abordagens e novas funcionalidades. Todavia, o teste e a qualidade do produto é por vezes negligenciado, porque costumam acarretar custos monetários e/ou temporais avultados.

Actualmente, tem-se elevado a preocupação em assegurar a qualidade dos produtos de software [RWW+]. Embora o teste "tradicional" de aplicações possua os mesmos objectivos que o teste a Aplicações Web, as teorias de teste, bem como as suas técnicas, não podem ser aplicadas da mesma forma, dadas as particularidades das plataformas em que estas se situam. O ambiente Web possui particularidades que exigem um novo tipo de abordagem, por variadíssimas razões: a aplicação é utilizada pelos seus clientes em diferentes ambientes e plataformas (diferentes *browsers*, dispositivos móveis, etc.), o acesso à aplicação é realizado de um modo concorrente e distribuído e o estado da rede que estabelece a ligação entre o cliente e o servidor não se comporta sempre do mesmo modo.

Às circunstâncias supracitadas aliam-se os processos de desenvolvimento de software adoptados pelas equipas nesta área, que também diferem dos mais convencionais. Esta diferença deve-se principalmente à necessidade destes projectos terem que acompanhar o ritmo acelerado, dinâmico e multidisciplinar a que ocorrem as mudanças no ambiente Web, bem como a complexidade e integração inerente aos objectivos dos projectos Web [GM01], [Pre00].

A realização de testes funcionais a uma Aplicação Web, simulando a utilização de um cliente real, proporciona uma validação transversal de todo o sistema que a compõe. A

realização destes testes de um modo manual, particularmente em modelos de desenvolvimento iterativo, acarreta normalmente um enorme esforço, devido principalmente à grande repetição das operações implicadas [KNR09].

1.2 Motivação

O principal objectivo da Automatização de Testes consiste na optimização dos processos de criação e execução de casos de teste, contribuindo assim para a possibilidade de uma realização de testes mais rápida e frequente. No entanto, a realização de testes manuais continua a ser o modelo de testes mais adoptado [WA06]. Este facto é actualmente justificado com os inúmeros casos de insucesso ocorridos na opção do modelo de automatização, assim como pela difícil estimação no cálculo do retorno de investimento que se poderá obter com a adopção de uma estratégia de automatização [RW06], [LISA09].

O presente documento apresenta uma análise que tem como principais motivações a demonstração da possibilidade da adopção de uma estratégia de automatização de testes no seio de uma empresa em funcionamento co metodologias ágeis, nomeadamente a metodologia Scrum.

A Cardmobili é uma startup nascida na UPTEC, Pólo de Ciência e Tecnologia da Universidade do Porto, constituída por uma equipa de 10 elementos. Surge de uma ideia que se centra na desmaterialização dos cartões de fidelização.

Lançado em 2009, é um serviço móvel, disponível através de uma plataforma Web multi-*browser* e uma aplicação móvel multi-plataforma (iPhone, Windows Mobile, Blackberry RIM, Android, MIDP2.0/Java). O modelo de desenvolvimento ágil adoptado pela Cardmobili confere ao serviço um ciclo de actualizações (*releases*) frequentes, nas suas diferentes plataformas. A complexidade do serviço, a necessidade de uma disponibilidade permanente e o cariz de escalabilidade pensado para um universo de utilizadores (pessoas e empresas) à escala global, justificam plenamente o controlo de qualidade, como uma peça fundamental no processo de desenvolvimento.

1.3 Objectivos

Pretende-se que seja realizado um estudo na área de testes funcionais a aplicações web. Desta análise, deverá resultar a implementação de uma plataforma de testes (de preferência, tendo como ponto de partida uma ferramenta já existente) que agilize o processo de especificação de cenários de testes funcionais, execução destes e análise dos dados resultantes da sua execução. Em suma, uma plataforma que tem como finalidade a automatização do processo de teste da camada de *software* a nível funcional.

Devido às características de uma aplicação que se encontra a funcionar em ambiente Web, pretende-se que a execução dos cenários de teste seja realizada em múltiplos *browsers* e plataformas (possivelmente até em *browsers* em dispositivos móveis) [WX09]. Pretende-se que a execução de todo o conjunto de testes possa ser agendada sem obstruir o desenvolvimento do produto. Da execução deverá resultar a geração de um conjunto de relatórios, que possibilite a compreensão dos dados de teste. Actualmente, não existem quaisquer plataformas que respondam a estas necessidades.

A plataforma de testes deve facilitar um meio de especificação de testes flexível, de modo a que, face a uma alteração da interface da Aplicação Web, os *scripts* de teste possam ser executados após pequena ou nenhuma re-adaptação. A sua especificação deve ser efectuada numa linguagem que facilite a sua leitura, criação, edição e manutenção. A inclusão da plataforma, pressupõe também a formação de todos os elementos que compõem a equipa de desenvolvimento da Cardmobili.

Tendo em vista futuras necessidades, a plataforma de testes deverá ser extensível. A título de exemplo, deverá facilitar a integração de outros tipos de testes, tais como: desempenho, carga e segurança.

Deverá também ser realizado um estudo na área dos processos de automatização de teste de *software*. Consistirá num estudo e análise dos vários passos que compõem o ciclo de teste, aprofundando conhecimentos na área de práticas e técnicas que deverá também abranjer testes de sistema tradicionais, bem como testes a Aplicações Web, com o intuito da plataforma se integrar da melhor forma possível no processo de desenvolvimento de *software* da empresa.

1.4 Estrutura do Documento

O presente documento encontra-se dividido em seis capítulos. No primeiro serão dadas a conhecer as motivações, os objectivos propostos e o enquadramento da dissertação.

No capítulo seguinte, serão apresentados o estudo e investigação realizados na área da Automatização de Testes Funcionais a Aplicações Web. Em primeiro lugar, serão discutidos os problemas e potencialidades da Automatização de Testes. De seguida, será realizada uma apresentação de boas práticas na área da Automatização de Testes. Por fim, será apresentada a investigação sobre a realização de Testes Funcionais a Aplicações Web, assim como as ferramentas e técnicas existentes actualmente nesta área.

No terceiro capítulo, denominado Análise e Decisões de Desenvolvimento, será realizada uma análise extensiva e a escolha de uma das ferramentas apresentadas no capítulo anterior. Serão ainda definidas e apresentadas todas as decisões fundamentais à implementação da plataforma de automatização de testes.

De seguida, no capítulo Implementação de Solução Final, é apresentada a implementação realizada, sendo dada a conhecer a construção da plataforma de automatização de testes, assim como o seu funcionamento interno.

Nos dois últimos capítulos são apresentadas as Perspectivas de Desenvolvimento Futuro e as Conclusões. Em Perspectivas de Desenvolvimento Futuro são apresentados esboços de concepção de futuras extensões à plataforma construída. Por fim, em Conclusões são apresentadas as conclusões finais de toda a dissertação realizada.

Capítulo 2

Automatização de Testes Funcionais a Aplicações Web

Este capítulo visa documentar o estado actual do conhecimento e técnicas dos temas que o projecto aborda. Tem como principal objectivo expôr estudos, análises e testemunhos realizados na área de Automação de Testes e complementarmente na área de Testes Funcionais a Aplicações Web.

Primeiramente será dado destaque à Automatização de Testes. A exposição será realizada no enquadramento do projecto, ou seja, será dada relevância à interligação do tema com os Processos de Desenvolvimento Ágil e a área de Testes Funcionais a Aplicações Web. Posteriormente, serão apresentadas ferramentas que surgiram da investigação realizada na área do desenvolvimento de Testes Funcionais a Aplicações Web. Complementarmente, serão realizadas pequenas exposições das mesmas, explicitando, sempre que se justifique, técnicas a que cada uma recorre.

2.1 Vantagens e Obstáculos na Automatização de Testes

As metodologias de desenvolvimento ágil defendem o princípio da capacidade de entrega e posse de *software* em estado de pronto funcionamento, ao longo de todo o processo de desenvolvimento. Este princípio tem como principal objectivo conferir ao desenvolvimento um estado de *software* de testes e qualidade assegurada. De modo a tornar este princípio exequível é necessário provir o processo de uma camada constante de execução de testes, garantindo os padrões de qualidade desejados. Assim, a automatização do processo de teste, bem como a possível automatização de outros sub-processos (p.ex. *builds* automáticos, monitorização, etc), maximizará o tempo disponível da equipa para produção do *software*, fornecendo também um meio de não cair no estrangulamento do tempo

disponível única e exclusivamente na certificação do teste e qualidade [CG09].

Normalmente, quando se pensa em práticas de Automatização de Testes, pensa-se imediatamente apenas na economia de tempo como principal benefício. No entanto, existem muitas mais razões a ter em conta quando se equaciona a adopção destas práticas, assim como possíveis obstáculos e/ou impeditivos.

A maioria das vantagens advêm da reposta directa aos problemas/necessidades, que as práticas de automatização de testes têm como objectivo responder. Como apresentado em [KNR09], [HK06], [SJ04], [CG09], [LISA09], [WX09] e [BWK05], após um estudo e análise de implementações destas práticas em empresas de desenvolvimento de *software*, foram identificadas diversas vantagens que resultaram do uso de processos de automatização.

Em linhas gerais, as principais razões a ter em consideração são apresentadas na seguinte lista [CG09].

- ***Testes manuais exigem mais tempo***

  A constatação que o esforço despendido na execução manual de todos os testes agendados é superior ao desejado, é a mais básica das razões para a substituição por uma solução automatizada. O esforço costuma tornar-se evidente à medida que o *software* desenvolvido aumenta em funcionalidades e complexidade. Infelizmente, por norma as empresas costumam adoptar uma solução provisória, que por vezes se torna permanente, que consiste na limitação dos testes a realizar, restringindo-se ao avaliado como estritamente necessário.

  Em termos de esforço de tempo, a principal diferença entre uma opção automática e a manual é que a primeira é maioritariamente influenciada pelo número de testes a realizar, enquanto que a opção manual é maioritariamente influenciada pelo número de execuções de testes a realizar [RW06].

- ***A execução manual é mais propensa a falhas***

  A execução manual de testes torna-se rapidamente numa tarefa repetitiva, especialmente se se tratarem de testes funcionais. A repetição de passos bastante similares, partilhados entre diversos cenários de teste, tende a que sejam cometidos erros que resultarão invariavelmente na não identificação de problemas existentes no sistema [CG09].

  Do mesmo modo, face a uma dificuldade ou impossibilidade na reprodução manual de um determinado *bug* conhecido, uma pessoa é tentada à desistência da tarefa, ou então, apostando na insistência, ao consumo de um esforço superior ao idealmente estimado/esperado.

  A automatização de testes garante que todos os cenários de teste especificados irão ser executados exactamente da mesma forma, tornando o processo mais coerente.

- ***Mais tempo para a equipa fazer o seu trabalho (melhor)***

  A recuperação do tempo outrora despendido na tediosa execução manual dos testes, traduz-se numa maior quantidade disponível de tempo a ser empregado na exploração, compreensão e desenvolvimento de *software* [WX09].

  Quando conjugado com TDD (*Test-driven development*), promove a compreensão dos requisitos. Como referido em [KaJ06], a construção de testes segundo certos padrões, poderá servir como documentação de um sistema.

- ***A automatização maximiza a realização de testes de regressão***

  Testes de regressão são quaisquer testes que visam a verificação contínua de um produto em desenvolvimento. Este tipo de teste garante a detecção de problemas resultantes da integração de componentes, previamente testados, com outros componentes. Os erros detectados podem também resultar de modificações posteriores em componentes anteriormente testados, bem como de particularidades que imprimem alterações no comportamento do sistema em teste.

  O nome deste tipo de testes advém do uso da expressão "o sistema regrediu", aquando da detecção de erros por este tipo de testes.

  A presença de uma bateria de testes de regressão funciona como uma "rede de segurança", permitindo que sejam feitas alterações com uma maior margem de segurança, pois caso as alterações introduzam algum problema no sistema, este será provavelmente detectado [CG09].

  A automatização de testes de rsegressão alivia o esforço, que no caso da sua realização manual, cairia unicamente sobre os *testers* que teriam de se isolar quase exclusivamente na sua concretização.

- ***Feedback antecipado e frequente***

  Uma plataforma pronta a executar os testes proporciona a possibilidade de uma execução mais regular. Por conseguinte, no decorrer do processo de desenvolvimento, a equipa de desenvolvimento tem ao seu dispor um parecer constante e contínuo do estado de evolução. A detecção antecipada de problemas resultará em operações de correcção mais fáceis e menos custosas monetariamente que quando feitas tardiamente, dado que, por norma, a complexidade de um problema aumenta com o tempo.

  Num desenvolvimento iterativo, com *sprints* de desenvolvimento relativamente curtos, esta possibilidade torna-se uma mais valia, especialmente quando um dos objectivos é a capacidade de possuir, sempre que possível, *software* em estado pronto a usar.

- ***Um conjunto de testes pode servir como documentação***

  O uso de exemplos e guiões de utilização como documentação de desenvolvimento

é uma prática comum em metodologias ágeis. Acompanhar o desenvolvimento com um conjunto de testes funcionais descritivos irá oferecer uma documentação sempre actual e representativa do estado em que o sistema se encontra. Outra particularidade reside no comportamento "vivo" dos testes como documentação, na medida em que, por exemplo, a execução de um cenário de teste funcional apresenta não só um guião de uso, bem como uma demonstração real do funcionamento do sistema.

Esta prática tem como principal objectivo evitar o inconveniente de recorrer a uma constante actualização da documentação estática, que apoia o desenvolvimento. Esperando-se que ao aproveitar a informação existente num teste, combinando-a com a informação necessária para a documentação, o esforço despendido na criação e alteração de informação seja partilhado entre estas duas actividades.

- ***ROI(return on investment)***

  Todas as razões até agora referidas, em conjunto, significam a obtenção de um desempenho, que comparativamente a uma abordagem de realização manual, deverá ser superior. Normalmente, este argumento é fortemente apoiado com os principais pontos apresentados por [BWK05]:

  - A Automatização de Testes auxilia na concretização de ciclos de *relases* cada vez mais curtas, principalmente pelo decréscimo da fase de testes. Os testes podem ser realizados mais frequentemente, os *bugs* serão descobertos mais cedo e os custos de correcção serão menores.
  - Os *testers* estarão disponíveis para conceber mais e melhores testes, dada sua maior disponibilidade outrora perdida na execução manual de testes. Consequentemente, a qualidade e a extensão dos testes aumentará significativamente.

  No entanto, como referido em [RW06] e [Bac99], um cálculo comparativo viável é muito difícil de se realizar, devido aos inúmeros factores não tangíveis e implicitamente ocultos no processo, tornando o cálculo de ROI comparativo pouco verosímil.

A adopção de processos de automatização pode deparar-se perante obstáculos que impedirão a concretização das principais vantagens anteriormente referidas. Em 2001, numa actualização de um artigo [Bre01], Bret Pettinchord após experiências pessoais e profissionais em diversos projectos em variadas empresas, concluiu que as principais razões que contribuíam para a falha da integração de processos de automatização de testes eram:

- ***Dispensa de tempo de reserva na automatização***

  É frequente a adopção de processos de automatização com o único intuito de recuperação de tempo, outrora despendido na execução manual de testes. Daí que, adoptando uma solução de automação, se tenda a reservar apenas o mínimo tempo possível (p.ex. tempo de sobra e/ou reserva), resultando na construção de um conjunto de testes que não satisfaz todas as necessidades de teste do sistema.

- ***Objectivos mal definidos***

  Como anteriormente referido, existem muitas razões para adoptar métodos de automação de teste. Mas, por mais tentador que seja tentar concretizar todas ao mesmo tempo, torna-se mais exequível a tentativa de concretização das que respondem melhor às necessidades inerentes ao projecto. Pois diferentes projectos requerem diferentes soluções.

- ***Inexperiência***

  A introdução de alterações no processo de desenvolvimento sem formação prévia resulta numa utilização deficiente dos novos meios disponíveis. Ferramentas de automatização de testes não realizam os testes por si só, mas sim a sua correcta e cuidada utilização.

- ***Não aproveitamento do conhecimento adquirido em caso de abandono de processos de automatização***

  A aprendizagem acarreta, normalmente, um grande esforço, dadas as características particulares deste tipo de ferramentas, e em caso de uma desistência na implementação, o conhecimento adquirido dificilmente poderá ser aproveitado noutros campos.

- ***Adopção apenas em desespero de causa***

  A insatisfação generalizada com um processo demorado de testes é uma das causas mais recorrentes para a adopção de processos de automatização. No entanto, não significa que a solução mais indicada seja a automatização de tarefas. A decisão de implementação de processos de automatização reflecte muitas vezes um desejo irrealista e não uma proposta necessária e concretizável.

- ***Concentração do esforço na automatização em si, e não no teste***

  Tende a surgir uma relutância em pensar exclusivamente em testes, quando se tenta automatizá-los. A ideia de automatização é para a maioria muito mais interessante que a ideia de testar algo. Este desvio de atenção origina a criação de testes menos eficazes, empobrecendo assim a camada de testes.

- ***Foco em problemas técnicos***

  De uma perspectiva de engenharia, a automatização representa um desafio tecnológico bastante interessante. A resolução de muitos dos problemas que possam daqui surgir, pode consumir mais do que o tempo que futuramente se poderá vir a poupar.

Todavia, este levantamento surgiu no seio de desenvolvimento genérico de software. Mais tarde, em 2009, [CG09] apresenta uma nova lista, com base em experiências no seio de equipas que se regem pelos princípios das metodologias ágeis. À anterior, esta lista acrescenta estes novos elementos:

- ***Resistência por parte dos programadores***

Trabalhadores acostumados a ambientes de desenvolvimento mais tradicionais (onde um *developer* raramente cumpre actividades de teste) ou com pouca experiência em ambientes de desenvolvimento ágil, costumam mostrar mais resistência à execução de tarefas de teste de *software*. Como as metodologias ágeis são por norma significado de pequenas equipas de desenvolvimento, [1] a resistência e a não compreensão, mesmo que por um pequeno número de *developers*, poderão ser significativos para a falha dos processos de automatização adoptados.

- ***Curva de Aprendizagem***

  Formar uma equipa em automatização de testes é um processo penoso e difícil de realizar. Numa fase inicial, se a aprendizagem for fracamente planeada poder-se-á tornar irrealizável. Além disso, não basta por si só formar, o tempo de assimilação de todas as novas práticas e conhecimento associado é também essencial.

- ***Investimento Inicial***

  Automatização poderá implicar custos: aquisição de *hardware*, *software*, formação, construção de uma ferramenta à medida para uso interno, entre outros. A realização de estudos e análise de ferramentas, construção e experimentação de protótipos que forneçam dados que permitam fazer uma previsão credível são extremamente importante para decisões e cálculos comparativos. Estes cálculos auxiliarão na decisão da viabilidade e que possíveis retornos de investimento se poderão obter.

  Por tudo isto, uma avaliação errada pode induzir práticas desenquadradas e economicamente impraticáveis ou inviabilizar a adopção de práticas adequadas ao meio em questão.

- ***Legacy Code***

  A integração de processos de automatização em projectos com camadas de código dificilmente "testáveis" poderão impossibilitar ou dificultar a criação de testes de cariz automatizável. Por vezes, tornar-se-á viável semi-automatizar ou mesmo excluir os referidos componentes da camada de automatização de testes.

- ***Falta de Confiança***

  Ao longo da introdução dos processos de automatização, é usual surgirem diversos desafios que poderão abalar a confiança da equipa de desenvolvimento. A instalação deste tipo de sentimentos no seio de equipas pequenas resulta em desistência prematura, criando sérios obstáculos na adopção de processos de automatização.

- ***Hábitos Antigos***

  É pela experiência que se poderá efectuar adaptações e melhoramentos nos processos que se encontram em implementação. Porém, especialmente durante o período de

[1] Encontram-se documentadas diversas histórias de sucesso em ambientes pequenos de desenvolvimento (<10 trabalhadores), contrastando com o insucesso com equipas maiores (>20 trabalhadores) [BT03], [Bec99].

transição, é recorrente negligenciar o uso das novas ferramentas em detrimento das antigas, principalmente por conforto (hábitos antigos, mesmo sabendo que este não produzem os melhores resultados) [CG09].

2.2 Boas Práticas na Automatização de Testes

Após estudo e análise, com base em inúmeros anos de experiência, Gerard Meszaros, Shaun M. Smith e Jennitta Andrea no *The Test Automation Manifesto* [MSA] definiram que testes automáticos devem ser:

- **Concisos** - Ser o mais simples possível
- **Auto Validativos** - Relatar os resultados da sua execução, sem intervenção humana
- **Repetíveis** - Multi-executáveis, sem intervenção humana [WX09]
- **Robustos** - Funcionam do modo especificado; a lógica da sua especificação não pode ser afectada por alterações no ambiente externo
- **Completos** - Verificam os requisitos para os quais foram concebidos
- **Imprescindíveis/Necessários** - Ser essenciais, nenhum teste pode ser dispensável
- **Claros** - Ser facilmente percebível
- **Eficientes** - A sua execução não deve demorar mais do que o tempo admitido
- **Especificos** - Cada teste deve ser concebido para testar uma determinada funcionalidade e não mais, facilitando a realização de "defect triangulation"[2]
- **Independentes** - Ser autónomo
- **Manuteníveis** - Ser de fácil interpretação, modificação e extensão
- **Rastreáveis** - Deve identificar que código, funcionalidade e restrição está a testar

Com o objectivo de completar os princípios mencionados, o *The Test Automation Manifesto* [MSA] inclui também um conjunto de práticas a seguir, aquando da construção de plataforma de automatização, bem como na altura da implementação e especificação de testes.

As práticas são agrupadas em quatro categorias, as três primeiras directamente relacionadas com as características dos princípios acima apresentados e um último grupo complementar.

2.2.0.1 Práticas de Legibilidade

As seguintes práticas relacionam-se com os seguintes princípios: "conciso", "imprescindível/necessário", "claro", "especifico" e "manutenível".

[2]Restrição iterativa até à identificação da fonte do problema detectado [Mes03].

Tests as Specfication

O teste deve especificar o comportamento que pretende validar com exactidão. Por outras palavras, deve explicitar o resultado correcto esperado e as condições que o devem originar.

Single Glance Readable

Um teste tem que ser conciso. Não deve ser requerido muito esforço para que o seu conteúdo seja percebido. O seu tamanho costuma ser um bom indicador, p.ex. se for necessário usar a *scroll bar* para ler o teste, é possível que esta prática não esteja a ser cumprida.

Intent Revealing Fixture

Deve ser dada relevância às pré-condições essenciais à correcta especificação do teste, aspectos não relevantes devem ser encapsulados. Na impossibilidade de encapsulamento deverá ser dada menor relevância na apresentação do teste.

Finder Methods

Na necessidade de criação de dados de teste, dever-se-á evitar uma definição "hard-coded". Aconselha-se o uso de métodos externos, que deverão ser responsáveis pela criação de dados. Com esta prática pretende-se que sejam obtidos testes claros e concisos.

Outcome Describing Verification Logic

A secção onde são especificadas as verificações (normalmente através de "asserts") deve ser o mais clara possível e menos susceptível de interpretações paralelas.

Single Condition Test

Um teste deve verificar uma e uma só condição simples (p.ex. um único cenário ou requisito). Cingir o domínio de cada teste possibilita a realização de "defect triangulation". Combinada com outras práticas, permite que o teste seja mais fácil de interpretar e manter.

Declarative Style

Deve ser dada preferência à concepção de testes que usem um estilo declarativo, descrevendo as condições e os resultados esperados. Contrastando com o estilo imperativo, onde em forma de receita, é indicado como e o que se pretende testar.

2.2.0.2 Práticas de Robustez

As práticas que compõem este grupo estão relacionadas com os seguintes princípios: "auto validativo", "repetível", "independente" e "manutenível".

Independent Tests

Todo o teste deve ser auto-contido. Quer isto dizer que a especificação não deve assentar em quaisquer pressupostos sobre o estado de outros testes.

Clean Slate Fixture

Um teste não deve depender directamente doutro qualquer teste, ou seja, deve construir todas as condições necessárias à sua realização.

Anonymous Creation Methods

O uso de métodos estáticos de criação de dados e/ou execução de tarefas simples e repetíveis, por parte do método principal que compõe o teste, previne problemas de conflitos usuais de partilha de dados na ocorrência de execução concorrente de diversos testes. Esta prática permite também reduzir o esforço na criação de novos testes, dado que este tipo de métodos fornece uma forma de reutilização de funcionalidades previamente implementadas.

Automated Test Cleanup

O uso de técnicas que automaticamente destroem o contexto criado pela execução de um teste, garante que cada teste será executado num ambiente controlado, coexistindo apenas com os dados especificados. A adopção desta prática reduz a complexidade na gestão de ambientes de teste.

SUT API Encapsulation

Abstrair a camada de especificação de testes das restantes camadas, permite realizar actualizações na plataforma e/ou framework evitando necessidades de realizar alterações nas especificações dos testes implementados. Esta prática contribui para a obtenção de testes legíveis e a concentração no que realmente importa na altura de implementar os testes propriamente ditos.

2.2.0.3 Práticas de Reutilização

Este grupo de práticas tem como objectivo ajudar no cumprimento dos seguintes elementos da lista de princípios apresentados: ”conciso”, ”claro” e ”manutenível” [SZ].

Reuse through Test Building Blocks

O corpo de métodos dos testes deve invocar métodos auxiliares, responsáveis pela execução de funcionalidades comuns e partilhadas, em vez da utilização de técnicas de ”inheritance” e ”override”.

Anonymous Creation Method

Já referido anteriormente em práticas de robustez.

Custom Assertions

Deve ser realizado um estudo prévio com o intuito de analisar o domínio das verificações que irão ser realizadas. A análise permitirá elaborar um conjunto de asserções características que deverão ser implementadas, numa camada externa à da especificação de testes. Posteriormente deverão ser invocadas pelos testes, reduzindo a complexidade presente nas especificações dos testes.

Parameterized Tests

Com o objectivo de se poder verificar o mesmo comportamento em diferentes circunstâncias devem ser especificados testes parametrizados. Assim, bastará executar o mesmo teste com diferentes dados para cobrir diferentes cenários.

Templated Framework Tests

A criação de um *template* poderá facilitar a criação e leitura de testes. O modelo adoptado poderá conter áreas bem definidas: preparação de dados, execução de passos, entre outros. Assim, bastará conhecer o modelo para realizar uma leitura do teste, mesmo que este não se conheça à partida.

Data-Driven Test Suite

Face à necessidade de execução de diversos testes em que a diferenciação assenta nos dados de teste é aconselhada a adopção de uma estratégia baseada em *Data-Driven Testing*. Bastará possuir um componente que, alimentado por uma base de dados de teste, executará diversas vezes o mesmo teste iterativamente, servindo para isso da prática *Parameterized Tests*.

2.2.0.4 Práticas Complementares

Round-Trip Test

Evitar a concepção de testes que possuam demasiadas especificidades, distribuindo-as em várias execuções sobre a mesma interface.

Pass-Thru Test

Validar interacções com outros SUT[3] recorrendo a componentes que simulem a sua presença, p. ex. uso de ”stubs” e ”mocks”.

Stub Out Slow

[3]System Under Test

Se um componente externo estiver a prolongar o tempo de execução de um teste, deverá ser substituído, recorrendo à prática *Pass-Thru Test*.

Stub Out Dependencies Beyond Control

Na presença de dependências externas, em que não se possua controlo directo, deve ser criado um "stub" de modo a realizar testes controlados a todas as interacções possíveis.

2.3 Automatização de Testes de Sistema

Na área de teste de software, costuma-se tradicionalmente separar os testes em quatro grupos: *unit tests*, *integration tests*, *system tests* e *acceptance tests*. Em [T06], é apresentada uma nova reorganização a ser usada dentro do contexto da Automatização de Testes, sugerida inicialmente por [Mes03]. Esta parte do princípio que dentro do âmbito da Automatização de Testes, a mesma abordagem pode ser usada a diferentes níveis. Entre outros, encontra-se o caso dos testes de sistema e dos testes de aceitação, que numa perspectiva de Automatização de testes pouco diferem entre si. Assim, segundo a nova abordagem de [Mes03] baseada na granularidade dos testes, existem três grupos: testes unitários, testes de componentes e testes de sistema.

Numa AUP[4] a porção mais pequena é a unidade(*unit*). Cada unidade possui uma API (a sua interface programática) que é usada na interacção com outras unidades no sistema. A API de cada unidade está então sujeita a testes, com o objectivo de verificar a sua funcionalidade. Devido à sua natureza, este tipo de testes são concebidos em contexto de automatização. Por conseguinte, actualmente existem imensas ferramentas e *frameworks* que suportam este tipo de testes, sendo a mais conhecida a família das *xUnit*.

Componentes podem diferir bastante dependo do seu contexto, mas poder-se-á afirmar que geralmente um componente é um conjunto de unidades, cujas interfaces (API's) interagem em conjunto. Com *component testing* será possível identificar em que componente(s) se encontram os erros, numa visão de interligação. Enquanto que com *unit testing* é possível identificar exactamente em que método(s) se encontram os erros.

Do ponto de vista da realização de testes, a diferença entre *component testing* e *system testing* é que o último propõe-se a realizar verificações das funcionalidades do SUT numa perspectiva autónoma e completa, enquanto que *component testing* realiza os testes de forma isolada dentro do sistema. Por outras palavras, *system testing* tem como objectivo realizar a verificação do sistema sobre a perspectiva do utilizador final (humano). Por conseguinte, este tipo de teste é realizado sobre as diversas interfaces de comunicação com o utilizador final, sendo a interface gráfica a mais comum.

[4]Application Under Test

Efectuar testes a nível da interface gráfica do sistema é *event-based* [SJ04], dado que as operações (acções) se enquadram no âmbito da simulação de um utilizador (p.ex. simulação *clicks* do rato) que despoletam eventos no sistema. Um componente essencial para a concretização da automatização deste tipo de testes é o uso de ferramentas capazes de interagir com uma interface gráfica [CG09].

Porém, existem certos testes sobre as *interfaces* com o utilizador que não devem (nem podem conceptualmente) ser automatizados, como é o caso dos testes de usabilidade. Um verdadeiro teste de usabilidade requer um utilizador real, da interacção deste com o sistema é que poderão ser geradas análises. Estas terão como base as reacções do utilizador, entrevistas posteriores, entre outras. Dada a natureza e os objectivos divergentes, são testes que terão irremediavelmente que ser efectuados "manualmente".

System testing pode ser dividido em várias categorias de testes. Como este documento se debruçará fundamentalmente numa das categorias (a de testes funcionais), serão deixadas de parte outras categorias de testes de sistema com grande potencial de automatização, nomeadamente os testes de carga e desempenho.

Por norma, os testes de carga e desempenho quando executados de forma automática reproduzem resultados em muito superiores a uma realização dos mesmos de um modo manual, como ainda proporcionam *ROI* muito mais atractivos. Não obstante, existem determinados casos deste tipo de testes que de modo manual não poderiam sequer ser efectivados.

2.4 Testes Funcionais a Aplicações Web

Os Testes Funcionais têm como principal objectivo verificar o correcto funcionamento do sistema em cenários específicos, através da simulação da interacção deste com um utilizador. Este tipo de testes ignora os mecanismos internos do sistema ou componentes, focando-se apenas nas respostas que o sistema apresenta aos seus utilizadores, tendo em vista as acções e dados introduzidos. Propõe-se, portanto, a verificar e validar o sistema, quanto aos seus requisitos funcionais [IEE90], [MRT08].

São testes do tipo "caixa preta", que têm como último objectivo testar a interacção entre todos os componentes e camadas do sistema, desde a interface até aos componentes de mais baixo nível, como por exemplo, a camada de acesso a dados.

O teste de software é uma das tarefas do desenvolvimento de software mais complicadas de concretizar com o sucesso desejado. O processo de testes a aplicações web tem-se demonstrado uma tarefa ainda mais complicada, dadas as suas peculiaridades [LF06]. Uma das principais dificuldades é a obrigação da execução destes em múltiplos *browsers*. Por outro lado, as aplicações web têm crescido em complexidade, possuem actualmente interfaces focadas na usabilidade, com muitas funcionalidades, a correr em *client-side*,

em contínua comunicação com componentes *server-side* [SPX08]. Outra característica desta área é o contínuo aparecimento de novas técnicas e estratégias de desenvolvimento, obrigando as ferramentas de teste a uma contínua evolução.

A execução de testes funcionais é muitas das vezes relegada para uma tarefa manual. No entanto, nos últimos anos têm surgido várias ferramentas, com o intuito de automatizar todo este processo, oferecendo técnicas de manipulação remota de *browsers* [HK06]. Uma extensa investigação permitiu identificar as funcionalidades típicas, arquitecturas utilizadas, bem como os objectivos em comum que cada uma das ferramentas deste tipo se compromete a cumprir.

Foi implementado um caso de teste para estudo, que poderá ser consultado no próximo capítulo, recorrendo a cada uma das ferramentas que de seguida serão apresentadas. A experimentação não foi realizada em algumas das ferramentas, por várias razões: falta ou escassa documentação e inexistência de versão de experimentação (nas ferramentas pagas). No entanto, foram estudadas documentações e apresentações disponibilizadas publicamente, assim como análises independentes realizadas a todas as ferramentas em questão.

Do estudo e implementação do caso de estudo realizado, foi possível identificar diferentes módulos funcionais, no desenvolvimento de testes funcionais a Aplicações Web:

- *Test Developer* - Módulo responsável por auxiliar na criação de *scripts* de teste. Normalmente, constituído pelos seguintes sub-módulos:
  - *Recorder* - Permite a criação automática de código do *script* de teste, com base numa sequência de passos executados num *browser* real
  - *Inspector*- Permite obter de uma forma automática ou assistida, a localização de elementos de uma página web.
- *Test Runner* - Módulo responsável pela execução do teste propriamente dito, recorrendo ao uso de um dos seguintes sub-módulos:
  - *Remote Executer* - Permite que o Teste Runner controle remotamente um *browser* real, e execute o teste nesse ambiente
  - *Browser Simulator* - Simula um *browser* real

A implementação e execução de um teste funcional com este tipo de ferramentas começa pela sua definição: quais os passos a executar, e que verificações devem ser feitas, durante o seu caminho de execução.

A implementação do caso de teste pode ser conseguida recorrendo a dois métodos:

- uso de um *Recorder*, que auxilia na obtenção do código
- escrita clássica de código

Ambos os métodos podem recorrer ao uso de uma ferramenta (*Inpector*), que auxília na definição de expressões que determinam a localização de um elemento (x)html, da aplicação web. Estas expressões podem ser escritas recorrendo a xpath, css, referenciação

de elemento por atributos e valores (id, name, entre outros), bem como a conjunção dos diferentes métodos, que é o caso, por exemplo, do uso do selector da *framework* jQuery.

Identificados os elementos da página, poderá ser possível implementar acções e validações ao conteúdo dos elementos. Normalmente, as acções sobre os elementos resumem-se à execução de *clicks*, preenchimento de campos de *input*, selecção de escolhas em elementos do tipo *combobox*, *radio*, entre outros. Como nas aplicações web actuais, as acções não se resumem apenas às já indicadas, é preciso também executar operações de *drag-and-drop*, despoletar eventos de *release-keys*, *scrolls* nas páginas, etc.

A implementação do *Recorder* é realizada de diversas formas. A estratégia mais usual é implementada em Javascript, que o *browser* interpreta e, paralelamente, com a página em que se pretende fazer os testes, grava as acções realizadas. Existem também implementações que consistem na criação de uma aplicação externa ao *browser*, que usando interface COM[5] ou outro tipo de comunicação inter-processos, grava as acções realizadas no *browser*. Por fim, existem também *Recorders* implementados como extensões para *browsers* específicos, mas que geram código independente da plataforma em que foi gerado. No entanto, esta estratégia insere um factor de repetição, quando é necessária a modificação de um teste, pois torna-se necessário realizar de novo a captura.

Obtido o código que implementa o teste, é necessário executá-lo. Dado que os *browsers* possuem diferentes interpretações dos componentes *client-side*, especialmente no que se refere a Javascript, esta execução é por norma realizada recorrendo à manipulação remota de diferentes *browsers*, os que os utilizadores das aplicações web mais usam. Menos eficaz, mas amplamente usado devido à sua simplicidade, é a utilização de um componente que tem como objectivo simular um *browser* real.

A manipulação de *browsers* reais é conseguida de diferentes maneiras. Uma assenta na ideia da construção de um *proxy*, pelo qual passarão as ligações dos *browsers*. O proxy injecta código em Javascript, que é responsável por executar os passos implementados nos *scripts* de teste. A outra estratégia consiste no uso ou construção de um componente que permitirá comunicar com um *browser* real. Este componente fornecerá uma interface que permite a recepção de instruções e a posterior execução destas no *browser* em questão.

O uso do *proxy* para injectar código em Javascript, permite que este possa ser utilizado em qualquer *browser* real, partindo do pressuposto que a implementação do componente em Javascript se encontra em conformidade com os interpretadores dos vários *browsers*. Por outro lado, um projecto desta natureza pressupõe numa vasta implementação, que com o tempo dificulta o seu desenvolvimento e manutenção. Outra grande desvantagem prende-se com o modo com que o Javascript é executado nos *browsers*, dado que o mais usual é este ser executado em ambiente de *sandbox*. Este ambiente acarreta inúmeras limitações,

[5]Component Object Model

quando o objectivo é simular as interacções de um utilizador real com a interface da aplicação.

Com o desenvolvimento de extensões próprias, obtêm-se implementações mais compactas e à medida de cada *browser*. No entanto, nem todos os *browsers* possuem uma arquitectura que permita o desenvolvimento de extensões. De referir também que esta solução obriga na maioria das vezes à re-implementação de funcionalidades comuns, para cada uma das extensões dos diferentes *browsers*.

2.5 Ferramentas de Automatização de Testes Funcionais a Aplicações Web

De seguida, irão ser apresentadas ferramentas que têm como principais funcionalidades o controlo remoto de *browsers*. Serão também apresentadas diversas ferramentas que se servem das anteriormente referidas, acrescentando outras funcionalidades que tentam responder às necessidades existentes no seio dos Testes Funcionais a Aplicações Web.

2.5.1 Selenium

Nome do Projecto: Selenium
Desde: 2004
Linguagens de Implementação dos Scripts de Teste: Java, Ruby, Groovy, Python, PHP, Perl, .NET (C#, VB.NET) e Javascript(usando a Rhino).
Licença: Apache License 2.0
Custos: Gratuita
Comunidade: Vasta e activa em vários canais de comunicação: fóruns, mailling lists e IRC
Documentação: Bastante completa e em contínua actualização
Plataformas: Corre em qualquer plataforma que possua disponível uma Java Virtual Machine
Sítio na Internet: http://seleniumhq.org/

Criada na ThoughtWorks[6], é provavelmente a ferramenta mais conhecida nesta área, encontra-se essencialmente dividida em três módulos independentes, mas que, quando usados em conjunto fornecem uma solução coesa e bastante completa: Selenium IDE, Selenium RC e Selenium Grid [HK06].

O Selenium IDE é o *Recorder*, que permite a criação assistida de *scripts* de teste. Esta ferramenta é executada apenas no Firefox, dado tratar-se de uma extensão para o mesmo. No entanto, o *script* gerado pode ser exportado para várias linguagens, podendo ser posteriormente executado em diferentes *browsers*, recorrendo ao Selenium RC.

[6] http://opensource.thoughtworks.com/released.html

O Selenium RC (Remote Controller) é o módulo que executa os *scripts* de teste. Os *scripts* podem ser executados em vários *browsers*, nas mais diversas plataformas. Encontra-se dividido em duas unidades principais:

- um servidor, escrito em Java, que executa e termina de forma autónoma instâncias de *browsers*(Internet Explorer, Mozilla Firefox, Safari, Google Chrome e Opera). Actua como um *proxy* HTTP que fica à escuta de sessões de teste. Assim que é dado início a uma sessão de teste, é indicado o *browser* onde se pretende executar o teste e, de seguida, são executados os passos presentes no *script* de teste transmitidos ao módulo (consultar diagramas presentes nas Figuras 2.1 e 2.2).
- um *thin client*, implementado em várias linguagens de programação, responsável por comunicar ao servidor Selenium RC o *script* a executar; dentro da ferramenta é conhecido como *Driving Process*.

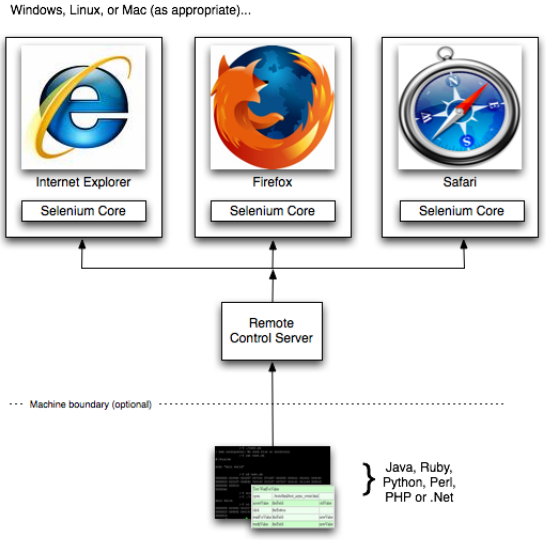

Figura 2.1: Diagrama Representacional do Selenium RC [Sel10]

O Selenium Grid nasceu da necessidade de acelerar o processo de execução de testes funcionais, dado que a execução sequencial consome muito tempo. A ideia é conseguir uma execução distribuída dos testes em diversos processos e plataformas, nos diversos *browsers* pretendidos. Esta execução permite reduzir o tempo consumido pela execução sequencial. Esta ferramenta comporta-se como um *HUB* que encaminha a execução de testes para os processos Selenium RC que se encontrem à escuta [Bru09].

Existe também um componente na estrutura Selenium, denominado Selenium Core, que é usado no Selenium IDE e Selenium RC. Este componente encontra-se escrito em Javascript e implementa toda a área responsável pela manipulação do *browser*. Por exemplo, o servidor Selenium RC após receber instruções de um cliente, traduz as mesmas para Selenese (comandos Selenium que correspondem a acções no *browsers*) e, através do uso

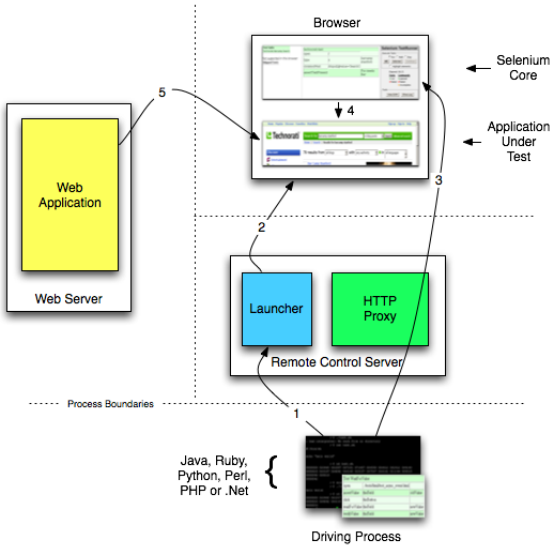

Figura 2.2: Diagrama de Arquitectura do Selenium RC [Sel10]

do Selenium Core, executa-as no *browser*; processo análogo no Selenium IDE, mas neste caso no sentido inverso, o Selenium Core intercepta as acções executadas no browser e tradu-las em Selenese.

Foi anunciado recentemente o desenvolvimento da versão 2 da ferramenta. Esta nova versão tem como principal ponto de viragem a fusão do projecto WebDriver com o Selenium.

O WebDriver era um projecto com o mesmo objectivo que a Selenium - automatizar os processos de testes funcionais a aplicações web. No entanto, a estratégia adoptada pelo projecto difere da Selenium. A WebDriver possui diversos componentes, denominados *Drivers*, para controlar, cada um, diferentes *browsers*. Por exemplo, para o Firefox o *Driver* é uma extensão. Esta estratégia, aliada aos fortes objectivos de ser uma *framework* com uma API intuitiva, de fácil exploração e aprendizagem, fez em pouco tempo do WebDriver uma opção bastante reconhecida [Sim07].

A equipa da Selenium procurava há já algum tempo uma maneira de resolver os problemas que advinham das restrições da *sandbox*, a que a Selenium Core estava sujeita, assim como das críticas que a API recebia frequentemente por parte dos utilizadores. Foi no WebDriver que a equipa da Selenium encontrou um aliado para a resolução deste problema. Simon Stewart, o criador da WebDriver, aceitou a proposta, e neste momento estão em processo de fusão. Segundo o anunciado, todos os *scripts*, escritos para os antigos WebDriver e Selenium, irão ser suportados nesta nova plataforma [Ste09].

2.5.2 Watir

Nome do Projecto: Watir (segundo os seus criadores, lê-se Water)
Desde: 2002

Linguagens de Implementação dos Scripts de Teste: Ruby, mas existem migrações da ferramenta para outras linguagens: WatiN para .NET(C#, VB.NET, F# e IronPython) e Watij para Java.
Licença: BSD License
Custos: Gratuita
Comunidade: Vasta e activa em vários canais de comunicação: fóruns, mailling lists e IRC
Documentação: Bastante completa e em contínua actualização
Plataformas: Corre em qualquer plataforma que possua disponível uma versão de Ruby
Sítio na Internet: http://watir.com/

Watir, do acrónimo "Web Application Testing in Ruby", inicialmente possibilitava apenas a manipulação remota do *Internet Explorer*. Actualmente é capaz de realizar o mesmo para outros *browsers*, tais como o Mozilla Firefox, Google Chrome, Safari e Opera [Wat10].

A ferramenta tira proveito da implementação, em Ruby, do protocolo OLE[7] implementado sobre uma componente recorrendo à arquitectura COM[8], para manipular o Internet Explorer [Cha09]. Com o crescimento, foram criados outros componentes que recorrem a outros métodos. O controlo do Mozilla Firefox é conseguido recorrendo ao uso do jSSH, uma extensão da Mozilla, que pode ser instalada posteriormente ou incluída na altura da sua compilação, que escuta a partir de um ligação TCP/IP pedidos remotos de execuções a realizar na aplicação (Figura 2.3) [SQF08].

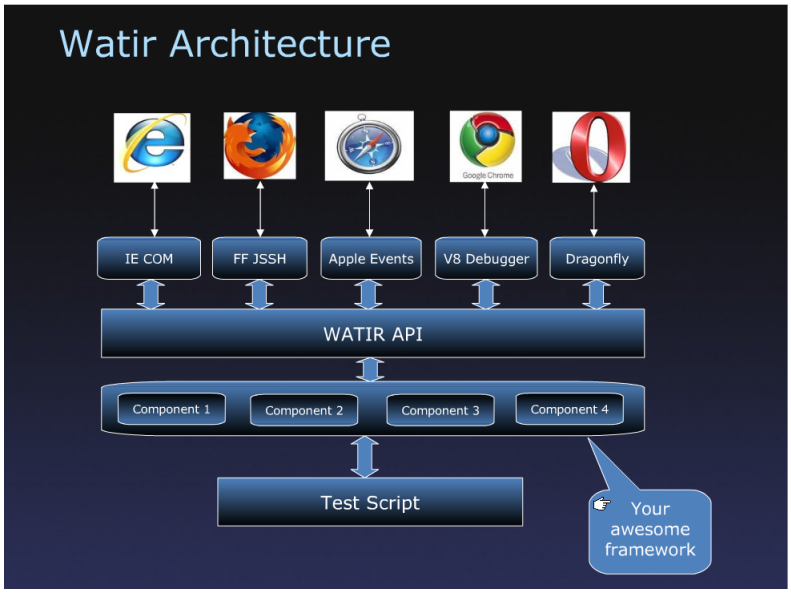

Figura 2.3: Watir - Diagrama de Arquitectura [Sai10]

[7]Object Linking and Embedding
[8]Component Object Model

Não possui quaisquer outros componentes, como por exemplo um *Recorder*. Contudo, existem projectos independentes que, recorrendo ao uso da Watir, implementam esta e outras funcionalidades.

O seu uso não pressupõe conhecimento de técnicas de localização de elementos (tais como: xpath, css, entre outros), dado que a sua API fornece um conjunto de métodos, que permitem realizar pesquisas de elementos recorrendo a variadíssimos filtros. No entando, é possível, se desejado em conjunção com os métodos referidos, aplicar expressões nas tradicionais técnicas de localização de elementos na DOM de uma página web. Devido a esta particularidade, aliada à simplicidade na leitura de código em Ruby, os *scripts* de teste em Watir são fáceis de construir (mesmo manualmente) e manter.

2.5.3 Windmill

Nome do Projecto: Windmill
Desde: 2006
Linguagens de Implementação dos Scripts de Teste: Python, JavaScript e, fruto de uma migração, Ruby
Licença: Apache License 2.0
Custos: Gratuita
Comunidade: Vasta e activa em vários canais de comunicação: mailling list e especialmente IRC
Documentação: Completa e em contínua actualização
Plataformas: Corre em qualquer plataforma que possua disponível uma versão de Python
Sítio na Internet: http://www.getwindmill.com/

A Windmill foi construída para ajudar a equipa de QA (Quality Assurance) do projecto Chandler Server Web UI (Cosmo)[9], dado que o desenvolvimento deste era caracterizado por possuir ciclos de *releases* curtos, e não pretendiam negligenciar a qualidade do mesmo [Ada10]. Pela natureza do aparecimento da ferramenta e dadas as necessidades actuais do teste a aplicações web, a Windmill foi criada com os seguintes objectivos:

- permitr a automatização dos testes funcionais em aplicações web
- possuir boas capacidades de *debug* de *scripts* de teste
- facilitar a escrita e manutenção de *scripts* de teste
- manter o projecto Windmill aberto a submissões e participações externas, fomentando uma cultura de apoio e desenvolvimento colaborativo

Esta ferramenta é maioritariamente construída em Javascript, mas possui também parte do código em Python.

Em Python, encontra-se implementada a secção responsável pela execução dos *browsers* (consultar diagrama presente na Figura 2.4). Recorrendo à estratégia de uso de um *proxy*

[9] http://hub.chandlerproject.org/

HTTP, inclui no *browser* a sua componente em Javascript. Esta componente, não só é responsável por executar o papel de *Recorder* como também o papel de *Test Runner*. O *Recorder* do Windmill é bastante intuitivo e permite a serialização do código do teste gerado para duas linguagens distintas: Javascript e Python. Possui também um método bastante eficaz de obtenção assistida de expressões que exprimem a localização de um elemento na DOM de uma página. O *Test Runner*, por sua vez, tendo um script de testes a executar, é capaz de o executar em modo assistido ou totalmente automático. Ambos os componentes, dada a sua implementação em Javascript, podem ser utilizados em diferentes *browsers*, nomeadamente Mozilla Firefox, Safari, Internet Explorer, Google Chrome e Opera [Win10].

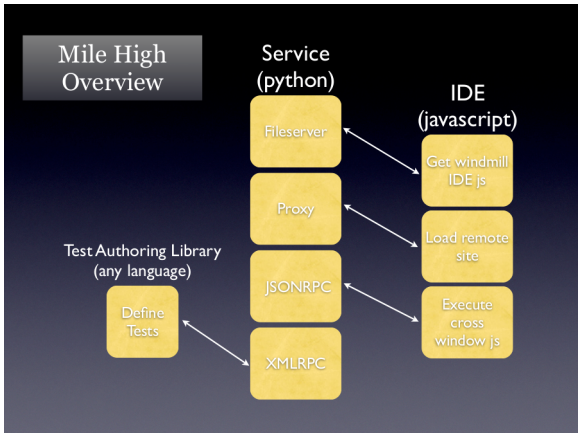

Figura 2.4: Windmill - Diagrama de Arquitectura [Ada08]

2.5.4 Sahi

Nome do Projecto: Sahi
Desde: 2005
Linguagens de Implementação dos Scripts de Teste: Linguagem da própria ferramenta
Licença: Apache License 2.0
Custos: Gratuita
Comunidade: Razoável e activa por mailling list e fórum oficial da ferramenta
Documentação: Completa e em contínua actualização
Plataformas: Corre em qualquer plataforma que possua disponível uma Java Virtual Machine
Sítio na Internet: http://sahi.co.in/w/

A Sahi é uma ferramenta que tem por objectivo facilitar a gravação e execução de *scripts* de teste. Outras funcionalidades principais incluem: possibilidade de execução em vários *browsers*, execução paralela de *scripts* de teste, reporte detalhado de execuções e funcionalidades de *debug*.

Esta ferramenta é implementada recorrendo a um servidor de *proxy* HTTP que, à semelhança de outros, injecta o seu componente escrito em Javascript, responsável pelas funcionalidades de criação assistida (*Recorder*) e execução (*Test Runner*) de testes.

Os *scripts* de teste na Sahi, são implementados numa linguagem própria [Sah10b]. A execução dos testes poderá ser conseguida de modo automático e independente. Opcionalmente, usando as potencialidades de *multi threading* do Java, os testes podem ser executados em paralelo [Sah10a].

2.5.5 MaxQ

Nome do Projecto: MaxQ
Desde: 2004
Linguagens de Implementação dos Scripts de Teste: Jython
Licença: BSD License
Custos: Gratuita
Comunidade: Pequena e pouco activa
Documentação: Reduzida
Plataformas: Corre em qualquer plataforma que possua disponível uma Java Virtual Machine
Sítio na Internet: http://maxq.tigris.org/

A MaxQ é uma ferramenta que permite a criação assistida de testes, colocando-se entre o *browser* e o servidor da aplicação web a testar. A execução de testes não é realizada num *browser*, dado que a MaxQ possui um componente que visa apenas simular o comportamento de um *browser* real (consultar diagrama presente na Figura 2.5).

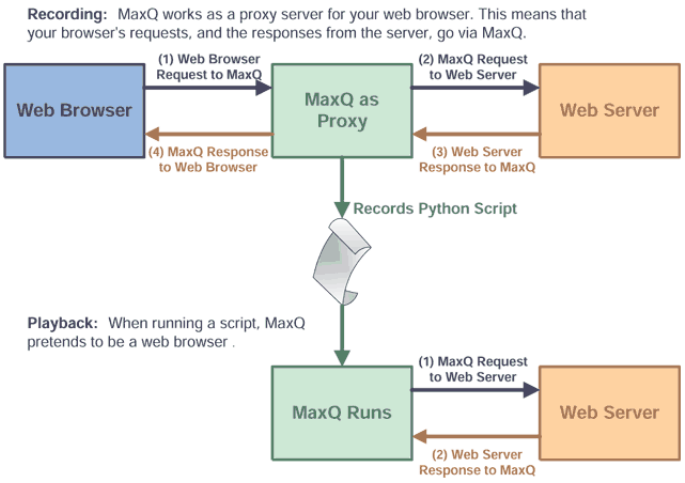

Figura 2.5: MaxQ - Diagrama de interacções [Max10]

Com o objectivo de trazer legibilidade e consequentemente maior facilidade na criação e manutenção de *scripts* de teste, a MaxQ usa Jython como linguagem de implementação

de testes. Jython é uma implementação de Python feita em Java, e como o núcleo desta ferramenta é feito em Java, a interligação entre ambos é conseguida de uma forma coesa.

2.5.6 SWAT

Nome do Projecto: SWAT
Desde: 2008
Linguagens de Implementação dos Scripts de Teste: FitNesse
Licença: GNU General Public License (GPL)
Custos: Gratuita
Comunidade: Pequena e pouco activa
Documentação: Razoável
Plataformas: Windows 2000/XP/Vista/7
Sítio na Internet: http://ulti-swat.sourceforge.net/

SWAT, do acrónimo Simple Web Automation Toolkit, é uma ferramenta de testes apenas disponível para Windows. É capaz de executar testes em múltiplos *browsers*: Mozilla Firefox, Internet Explorer, Safari e Google Chrome.

Ao contrário da maioria das outras ferramentas do mesmo género, o editor funciona também como *Recorder*. O editor é uma aplicação independente (implementada em C#), que procede ao controlo remoto do *browser*, recorrendo à mesma estratégia usada pela Watir [SWA10].

Os *scripts* de teste desta ferramenta são escritos usando a sintaxe do FitNesse, que se encontra integrado no projecto. O FitNesse é um ferramenta que integra num só sistema funcionalidades de teste e uma wiki. Os *scripts* de teste são páginas wiki, que são posteriomente visualizados ou executados. A SWAT usa a componente da wiki do FitNesse como modo para descrever e apresentar os *scripts* de testes funcionais [Fit10].

2.5.7 Apodora

Nome do Projecto: Apodora
Desde: 2007
Linguagens de Implementação dos Scripts de Teste: IronPython
Licença: GNU General Public License (GPL)
Custos: Gratuita
Comunidade: Muito pequena
Documentação: Reduzida
Plataformas: Windows 2000/XP/Vista/7
Sítio na Internet: http://www.apodora.org/

Criada na ACULIS[10], a Apodora é uma ferramenta com uma abordagem diferente de todas as ferramentas deste género. A ferramenta liga-se a um SGBD[11], onde armazena a informação das páginas onde se irão realizar os testes. A estas entidades na Base de Dados, associam-se os elementos presentes nas páginas (e respectiva informação), relevantes nas interacções nos testes a realizar [Apo10].

Na criação dos *scripts* de teste, os elementos presentes na base de dados são referenciados, em vez de realizar a localização *on-demand* como em outras ferramentas. Os *scripts* de teste são escritos em IronPython, uma implementação de Python escrita em C#, com o objectivo de trazer legibilidade aos *scripts* de teste. Estes testes poderão ser executados apenas em dois *browsers*: Internet Explorer e Mozilla Firefox.

2.5.8 WebAii

Nome do Projecto: WebUI Test Studio
Desde: 2009
Linguagens de Implementação dos Scripts de Teste: C# e VB.NET
Licença: Proprietária
Custos: $2,499
Comunidade: Razoável
Documentação: Completa e em contínua actualização
Plataformas: Windows 2000/XP/Vista/7
Sítio na Internet: http://telerik.com/products/web-testing-tools/webui-test-studio-features.aspx

A WebUI Test Studio é instalada sobre o Visual Studio da Microsoft, acrescentando-lhe funções de gestão de testes, tais como: gestão de casos de uso, criação assistida e edição de *scripts* de teste e execução remota de testes em diferentes *browsers*. Um dos principais objectivos da ferramenta é potenciar e flexibilizar o processo de criação de testes de funcionalidades em aplicações, que usam a tecnologia ASP.NET AJAX. A ferramenta encontra-se perfeitamente interligada com as restantes da empresa, o conjunto de ferramentas RadControls.

A ferramenta integra num só ambiente, o do Visual Studio, as funcionalidades de *Recorder* e de *Test Runner*. De realçar que o *Recorder* usa um sistema *point and click* para facilitar a localização de elementos, e que a partir de um *script* de teste, é possível gerar um esquema representando a *storyboard* do mesmo. O *Test Runner* usa uma componente de abstracção de *browsers*, que pertence à WebAii Testing Framework[12], permitindo que os testes possam ser executados no Internet Explorer, Mozilla Firefox e Safari. O WebAii Testing Framework, também da Telerik, é um projecto *open-source* com uma API com o objectivo de facilitar a escrita e execução de testes. Uma solução criada nesta *framework*

[10] http://www.aculis.com/
[11] Sistema de Gestão de uma base de dados
[12] http://www.telerik.com/products/web-testing-tools/webaii-framework-features.aspx

pode ser integrada com vários sistemas de testes unitários .NET: VS unit testing, NUnit, MbUnit e XUnit [Art09].

2.5.9 iMacros

Nome do Projecto: iMacros Scripting Edition/iMacros PRO Edition
Desde: 2006
Linguagens de Implementação dos Scripts de Teste: Windows Scripting Host, Visual Basic 6, Visual Basic .NET, C#, Java, Perl, Python, C++, ASP, PHP, ASP.NET
Licença: Proprietária
Custos: $199- $499
Comunidade: Vasta e activa em vários canais de comunicação
Documentação: Completa e em contínua actualização
Plataformas: Windows 2000/XP/Vista/7
Sítio na Internet: http://www.iopus.com/imacros/

Esta ferramenta foi inicialmente criada com o intuito de facilitar a criação de soluções de *web automation* e *web scraping*. Com o rápido crescimento, especialmente na automatização de pequenas tarefas sobre aplicações web (encontra-se no top 10 das extensões mais populares do Firefox nas categorias *bookmark, social and sharing, web data, alerts, widgets e web development*), a ferramenta alargou o seu leque de funcionalidades para a área de *Web Minning* e *Web Testing*. Com estes novos objectivos, foram lançadas duas versões diferentes: iMacros Scripting Edition e a iMacros PRO Edition [iMa10b].

A iMacro serve-se do Component Object Model para controlar remotamente os dois *browers* que suporta, o Mozilla Firefox e o Internet Explorer. O código dos *scripts* de teste pode ser escrito em qualquer linguagem que suporte essa tecnologia, sendo que as mais aconselhadas pelos criadores da ferramenta são: Windows Scripting Host, Visual Basic 6, Visual Basic .NET, C#, Java, Perl, Python, C++, ASP, PHP, ASP.NET.

Adicionalmente, a ferramenta permite, recorrendo a uma tecnologia denominada por DirectScreen, controlar elementos fora do espectro (x)html, tais como: Java applets, Adobe Flash, Adobe Flex, Microsoft Silverlight e controlos ActiveX [iMa10a].

2.5.10 actiWATE

Nome do Projecto: actiWATE
Desde: 2004
Linguagens de Implementação dos Scripts de Teste: Java
Licença: Licença própria[13]
Custos: Gratuita
Comunidade: Muito reduzida e pouco ou nada activa

[13]http://www.actiwate.com/license.html

Documentação: Reduzida
Plataformas: Corre em qualquer plataforma que possua disponível uma Java Virtual Machine
Sítio na Internet: http://www.actiwate.com/

A actiWATE é uma ferramenta constituida por módulos, em que um ainda se encontra por implementar e outros dois já estão totalmente implementados e prontos a utilizar (ver Figuras 2.6, 2.7, 2.8).

O módulo principal é o responsável por fornecer a API para implementar os testes, bem como outros dois complementares: o *Logger* e *Emulated Browser*. Todo esta *framework*, é compatível com jUnit, permitindo assim utilizar outras aplicações de teste compatíveis com jUnit. É também compatível com a Log4j, um subprojecto da Apache de *Logging* (ver Figuras 2.6, 2.7 e 2.8).

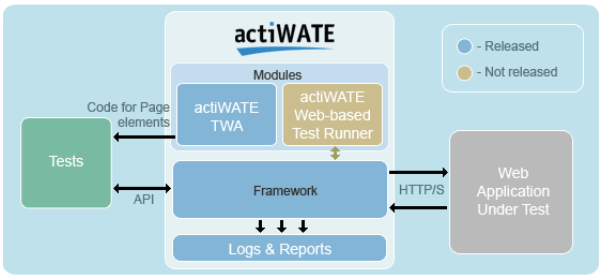

Figura 2.6: Diagrama de Componentes da actiWATE [act10]

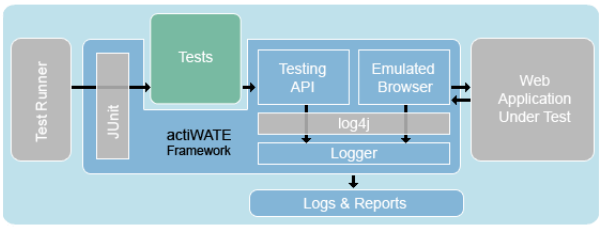

Figura 2.7: actiWATE Framework [act10]

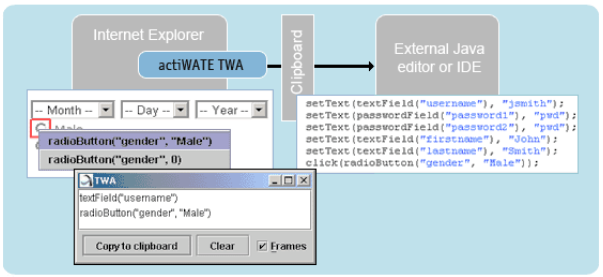

Figura 2.8: actiWATE TWA [act10]

Recentemente foi também construído um componente *Recorder*, chamado actiWATE TWA, do acrónimo *Test Writing Assistant*, que funciona como *plug-in* para o Internet

Explorer.

O *Test Runner* da actiWATE executa os testes recorrendo ao módulo *Emulated Browser*, que fornece as mesmas funcionalidades de um *browser* real, executando até o Javascript presente nas páginas. Encontra-se em construção um novo componente que permitirá executar os testes de forma remota sobre *browsers* reais, à semelhança de outras ferramentas deste género.

2.5.11 Squish

Nome do Projecto: Squish for Web
Desde: 2006[14]
Linguagens de Implementação dos Scripts de Teste: Python, JavaScript, Perl e Tcl
Licença: Proprietária
Custos: 2600eur ou 3800eur, dependo da extensão da compra
Comunidade: Reduzida
Documentação: Suporte activo se licença regularizada
Plataformas: Windows (NT, 2000, XP, Vista, 7), Linux e derivados Unix, Mac OS X
Sítio na Internet: http://www.froglogic.com/products/squish/web.php

A froglogic[15] é a detentora desta ferramenta, que possui várias versões: Squish for Qt, Squish for Java, Squish for Mac, Squish for 4Js, Squish for Tk, Squish for XView e a Squish for Web. Todas estas versões partilham em comum o objectivo de realizar testes funcionais a aplicações em diferentes plataformas e paradigmas.

Os *scripts* de teste podem ser escritos manualmente ou em modo assistido (recorrendo a um *Recorder*), em várias linguagens de programação: Python, JavaScript, Perl e Tcl. O *Test Runner* permite executar os testes em diferentes *browsers*, em diferentes ambientes: Internet Explorer, Mozilla Firefox, Safari e Konqueror [Fro09].

2.5.12 TestMaker

Nome do Projecto: TestMaker Community / TestMaker Enterprise
Desde: 2006
Linguagens de Implementação dos Scripts de Teste: Python, JavaScript, Perl e Tcl
Licença: GNU General Public License v2 ou Licença Proprietária
Custos: 2600eur ou 3800eur, dependo da extensão da compra
Comunidade: Razoável
Documentação: Suporte activo se licença regularizada
Plataformas: Corre em qualquer plataforma que possua disponível uma Java Virtual Machine

[14] http://www.froglogic.com/pg?id=NewsEvents&category=51
[15] http://www.froglogic.com

Sítio na Internet: http://www.pushtotest.com/

A PushToTest lançou duas versões diferentes da TestMaker: a TestMaker Community e a TestMaker Enterprise. A TestMaker Community contém um conjunto de funcionalidades com o objectivo de manter e realizar testes, bem como fornecer um suporte de gestão das mesmas. Por outro lado, a TestMaker Enterprise engloba a versão Community, e adiciona-lhe funcionalidades no que respeita a uma realidade de negócio de maior escala e envolvendo testes a outros conjuntos de tecnologias, nomeadamente SOA e outros elementos não (x)html nativos (p.ex. Flax e Flex) [Pus10b].

A TestMaker integra várias ferramentas da área de testes, erigindo-as conjuntamente numa única plataforma. As tecnologias que a integram são: soapUI[16], Selenium, HTMLUnit[17], Glassbox[18] e SpikeSource TestGen4Web[19].

A execução de testes neste sistema integrado funciona de modo distribuído. Os testes na TestMaker são agrupados em conjuntos *scripts* chamados *Test Agents*. Os *scripts* podem ser escritos em Java, .NET, Jython, Groovy, PHP e Ruby. A sua execução ocorre num ou mais *TestNodes* da ferramenta, cada teste é acompanhado pelo *PTTMonitor*, que observa o seu *backend*, à medida que o teste vai sendo executado. No fim de uma execução, os resultados são apresentados, facilitando a identificação da raíz de possíveis problemas de desempenho ou *bugs* no código [Pus10a].

Opcionalmente, o *TestRunner* do TestMaker suporta o Selenium Remote Control, assim como o Selenium Grid.

2.5.13 Bad Boy

Nome do Projecto: Bad Boy

Desde: 2003

Linguagens de Implementação dos Scripts de Teste: Linguagem própria

Licença: Licença própria[20]

Custos: $45

Comunidade: Reduzida

[16]Ferramenta de desenvolvimento de testes a Web Services: SOAP/WSDL e REST. A interface da soapUI é conhecida por ser de fácil aprendizagem e por acelerar o processo de desenvolvimento de testes. A soapUI inclui diversas funcionalidades de desenvolvimento, tais como: serviço de alojamento e implementação de *Mocks* e ferramentas de migração de WSDL. Na versão Pro, são incluídas mais funcionalidades, de que se destacam as de análise de cobertura de testes e um editor de *scripts* de testes com auto-completação. Sítio na Internet: http://www.pushtotest.com/products/soapui

[17]É um *browser* feito em Java sem interface, com o objectivo de simular um *browser* real. O projecto usa o motor do Rhino como interpretador de Javascript. Sítio na Internet: http://htmlunit.sourceforge.net/

[18]Glassbox usa AOP (*Aspect Oriented Programming*) para poder monitorizar a execução de servidores aplicacionais web em Java. Entre os vários tipos de monitorização, a ferramenta observa o estado do uso da memória, *threads* e ligações à base de dados. Sítio na Internet:http://www.pushtotest.com/products/glassbox

[19]A TestGen4Web é uma extensão para o Mozilla Firefox que permite realizar tarefas de *Record* e execução de testes. Sítio na Internet: http://developer.spikesource.com/wiki/index.php/Projects:TestGen4Web

[20]http://www.badboy.com.au/index/license

Documentação: Suporte activo se licença regularizada
Plataformas: Microsoft Windows
Sítio na Internet: http://www.badboy.com.au/

O Bad Boy incorpora o Internet Explorer como instância do seu sistema, comportando-se como um componente externo. Todas as acções executadas neste componente são monitorizas e controladas. A ferramenta cria automaticamente código que traduz as acções realizadas, que poderá posteriomente ser exportado para um *script* de teste. A execução dos testes é realizada unicamente no Internet Explorer. Durante a execução, a ferramenta continuará a sua monitorização, apresentando todos os detalhes associados às acções realizadas.

2.5.14 WET

Nome do Projecto: WET
Desde: 2005
Linguagens de Implementação dos Scripts de Teste: Ruby
Licença: BSD license(WET Core) e LGPL2.1(WET UI)
Custos: Gratuita
Comunidade: Razoável
Documentação: Completa e em contínua actualização
Plataformas: Windows 98/Me/2000/XP/Vista/7 com Ruby e .NET 2.0 instalados
Sítio na Internet: http://wet.qantom.org/

O WET começou por ser um simples *plugin* do Watir, que com o crescimento de utilização, evoluiu para uma ferramenta autónoma.

Possui dois componentes principais, o WET Core e o WET UI. O Core é responsável pela maior parte da execução dos testes, enquanto que o UI, construído em .NET, assiste o *tester* na criação de testes.

2.5.15 StoryTestIQ

Nome do Projecto: StoryTestIQ
Desde: 2006
Linguagens de Implementação dos Scripts de Teste: FitNesse
Licença: GNU General Public License (GPL)
Custos: Gratuita
Comunidade: Razoável
Documentação: Completa e em contínua actualização
Plataformas: Corre em qualquer plataforma que possua disponível uma Java Virtual Machine

Sítio na Internet: http://storytestiq.solutionsiq.com/

A StoryTestIQ é a ferramenta que alia a Selenium e a FitNesse numa só ferramenta. Como anterioramente referenciado, a FitNesse integra numa ferramenta as te funcionalidades de teste e uma wiki. Os *scripts* de testes são páginas de wiki, que são posteriomente visualizados e/ou executados em ambiente Selenium.

Capítulo 3

Análise e Decisões de Desenvolvimento

Neste capítulo será apresentado o primeiro leque de decisões tomadas, tendo em conta o enquadramento e necessidades da empresa em que a plataforma se insere.

Primeiramente será realizada uma análise às ferramentas apresentadas no capítulo anterior, com a finalidade de escolher a que melhor responderá às necessidades da plataforma de automatização de testes.

Posteriormente serão dadas a conhecer as decisões tomadas, acompanhadas pelos requisitos e necessidades que se propõem resolver. De modo a melhor enquadrar a plataforma de testes, tomar-se-á o processo manual de testes funcionais da empresa como ponto de partida para as primeiras ideias de solução. Este capítulo incluirá também diagramas de arquitectura, actividades e *mockups* de interface.

3.1 Análise e Escolha da Ferramenta

Após uma investigação de índole mais teórica, apresentada no capítulo anterior, tendo como base as tecnologias e o estado das ferramentas usadas na área dos Testes Funcionais a Aplicações Web, passou-se a uma fase mais prática. Para esta fase de experimentação, foram seleccionadas as ferramentas que melhor se enquadravam e as que potenciavam uma futura evolução.

3.1.1 Caso de Teste de Experimentação

Para o efeito, foi criado um cenário de teste de relativa complexidade, a ser implementado em cada uma das ferramentas seleccionadas. Com este ensaio, pretendia-se também obter uma maior aproximação deste paradigma de teste, assim como identificar pontos

fracos e pontos fortes das diversas ferramentas.

O cenário é um hipotético teste funcional ao caso de uso "Registo de Cartões". A especificação do cenário de teste segue o seguinte guião:

1. Aceder à página principal da Cardmobili.
2. Iniciar sessão com o utilizador de teste com o Nome de Utilizador: "**email_da_conta**"e Password: "**palavra-passe_da_conta**".
3. Terminada a fase de autenticação, navegar até onde se encontram listados todos os cartões suportados pelo sistema da Cardmobili, carregando no *link*: "**All Cards**".
4. Na caixa de pesquisa, escrever o nome do cartão "**myzoncard**".
5. Usando a funcionalidade que realiza a pesquisa em tempo de escrita, realizar um compasso de espera pela sua realização, e então no cartão "**myZONcard**", carregar no botão de adição (contém a descrição: "**Add card to your account.**").
6. A acção irá despoletar a abertura do elemento "**Add card**" para o nome do cartão inserido.
7. Nos campos de edição "**Name**"e "**Card number**", escrever respectivamente: "**Hugo Gomes**" e "**NUM_CARTAO**".
8. Carregar no botão "**save**", de modo a confirmar a adição do cartão.
9. Aguardar resposta e confirmar a apresentação da mensagem de operação concretizada com sucesso: "**Card successfully added to your account. Keep registering cards or check your cards in your 'My cards' tab**".

3.1.2 Implementação do Caso de Teste

Este teste foi implementado com sucesso ou com relativo sucesso na maioria das ferramentas apresentadas no capítulo anterior. Não foi possível implementar em algumas das ferramentas, por impossibilidade de aquisição da ferramenta ou por se ter revelado impossível implementar ou executar o teste, conforme o especificado.

As implementações melhor conseguidas podem ser consultadas nos Anexos A, B, C, D e E, respectivamente usando as seguintes ferramentas: Selenium, Selenium2, Watir, Windmill e Sahi.

Com o objectivo de experimentar o comportamento conjunto das ferramentas com algumas *frameworks* de testes, optou-se por usar também algumas das *frameworks* de teste mais conhecidas.

3.1.3 Principais Problemas Identificados

Foram identificados diversos problemas, bem como as áreas que requerem maior esforço, durante o processo de desenvolvimento de Testes Funcionais a Aplicações Web. Alguns dos referidos problemas têm origem na própria arquitectura da ferramenta, enquanto que outros provaram ser comuns no desenvolvimento deste tipo de testes.

Um dos principais problemas manifesta-se quando se pretende determinar a localização de elementos de uma página Web. Nesta operação pode recorrer-se a um leque, mais ou menos variado, de estratégias. As mais usadas são as expressões XPath, CSS, localização por valores de conteúdo ou atributos de elementos da página e o modo nativo de referenciamento de elementos na DOM com *Javascript*. A estas acrescentam-se estratégias que permitem a junção das várias técnicas mencionadas, ou então a criação de código que analisa iterativamente porções de código (x)html da página, realizando a localização de elementos, recorrendo por exemplo a expressões regulares.

Sempre que se constrói uma expressão de localização, pretende-se que esta funcione mesmo após uma remodelação da página e/ou alteração na geração desta, por outras palavras, a expressão deve ser flexível. O actual uso de *web frameworks* que possibilitam o desenvolvimento de um modo integrado, relegando a atribuição de id's a elementos, para responsabilidade destas, dificulta ou impossibilita o uso de algumas estratégias de localização, como é o caso da atribuição dinâmica de id's aos elementos.

Outro problema identificado, depreende-se com a própria natureza da plataforma das Aplicações Web, que se manifesta normalmente por flutuações na velocidade de obtenção de conteúdo pela rede. Normalmente, para contornar este problema, tende-se a acordar um valor para o intervalo de tempo que se deve tolerar, na obtenção de conteúdo pela rede. Existem duas categorias principais de eventos que requerem a obtenção de conteúdo pela rede: obtenção do conteúdo completo de uma página (p.ex. abertura de uma página pela primeira vez), obtenção "parcial" (p.ex. conteúdo obtido por AJAX[1]). Uma vez que actualmente as Aplicações Web possuem inúmeras rotinas, em *background*, que se valem de AJAX para obtenção de conteúdo, torna-se impraticável esperar por uma espécie de estado de quietude numa Aplicação Web (ou seja, quando já não há garantidamente mais conteúdo por receber). Assim, e numa perspectiva do utilizador, recorrendo ao valor de tempo tolerado já referido, é costume especificar testes funcionais em Aplicações Web realizando uma espera de rotina pelo aparecimento do elemento em questão (de notar no código apresentado as rotinas de *waitForCondition* na Selenium, *WebDriverWait.until* na Selenium2, *wait_until* na Watir e *waits.forElement* na Windmill).

[1] Asynchronous Javascript And XML

Algumas das ferramentas que foram experimentadas, implementam o componente responsável pelo controlo do *browser* quase inteiramente em Javascript. Este é injectado no *browser*, quando os testes se começam a executar, funcionando como *middle-code* entre a Aplicação Web e o *browser*. As instruções deste tipo de componentes, por se encontrarem em ambiente de *sandbox* no *browser*, não conseguem simular, de um modo inteiramente fiel, as acções de um utilizador real. Esta limitação traduz-se essencialmente, na maioria das vezes, na dificuldade em despoletar eventos relacionados com o teclado, com a *scroll bar*, envio de ficheiros, controlo de janelas *pop-up*, comunicações HTTPS[2], entre outros.

Por fim, falta referir um defeito presente em quase toda as ferramentas que se preende com a "postura" face à perspectiva de uma utilização real de uma Aplicação Web. Por outras palavras, refiro-me a interacções com elementos Web, mesmo que estes não se encontrem visíveis para o utilizador, que a maioria das ferramentas autoriza. Ou seja, neste caso particular, se um elemento só se mostrar visível para o utilizador após a realização de um *hover* sobre um outro elemento ou se este se encontrar escondido por outro elemento, a interacção com o elemento em questão deveria falhar. Nesta área, apenas a Selenium2 se comporta da maneira mais correcta (de notar no código exemplo a método *hover*[3], assim como o método isDisplayed[4]).

3.1.4 Conclusões e Escolha das Ferramentas

De seguida, será apresentada uma tabela com classificações às ferramentas que foram experimentadas, tendo em atenção diversas variáveis. Os valores de classificação encontram-se no intervalo de 1 a 5, sendo 5 o melhor.

Ferramentas	Browsers	Extensões	API	Aprendizagem	Estável	Maturidade	Automatização	Total
Selenium2	5	4	3	4	4	4	4	4
Selenium	4	4	4	3	4	5	4	4
Watir	5	4	4	4	4	3	3	4
Windmill	5	3	4	4	4	3	3	4
Sahi	5	3	2	3	3	3	3	3
SWAT	2	2	3	4	3	2	3	3
Apodora	2	2	2	2	2	3	3	2
WebAii	4	3	3	2	3	4	3	3
iMacros	2	3	2	3	2	3	2	2
actiWATE	1	3	3	3	3	2	2	3
WET	3	2	3	3	2	2	3	3
StoryTestIQ	5	4	2	4	3	2	3	3

Tabela 3.1: Classificação de Ferramentas

As ferramentas que têm um componente implementado em *Javascript* de controlo de acções, têm mais propensão à compatibilidade com diversos *browsers*, embora não seja garantido, sendo prova deste facto o código que não tenha sido criado tendo em vista os diferentes interpretadores de Javascript presentes nos diferentes *browsers*. Das ferramentas

[2]HTTP Secure

[3]http://selenium.googlecode.com/svn/trunk/docs/api/java/org/openqa/selenium/RenderedWebElement.html#hover

[4]http://selenium.googlecode.com/svn/trunk/docs/api/java/org/openqa/selenium/RenderedWebElement.html#isDisplayed()

experimentadas, as que suportam a execução num maior número de *browsers* são as seguintes: Selenium, Selenium2, Watir, Windmill, Sahi e StoryTestIQ. É importante referir os recentes desenvolvimentos na Selenium2, que apontam para um futura compatibilidade com *browsers* de plataformas móveis, como é o caso do iPhone[5] e do Android[6].

Parte do potencial de extensibilidade das ferramentas analisadas encontra-se intrinsecamente ligado com as linguagens e/ou plataformas usadas na sua construção, isto porque certas linguagens possuem um leque mais abrangente de ferramentas com potencial interesse de adopção que outras. Além disso, certas linguagens facilitam também o desenvolvimento de extensões. Por tudo isto e também pelas API das ferramentas que fornecem explicitamente meios de as extender, a Selenium, a Selenium2, a Watir e a StoryTestIQ foram as que obtiveram melhor classificação neste ponto.

A implementação do mesmo caso de teste nas diversas ferramentas permitiu identificar quais as que possuíam melhor API e melhor curva de aprendizagem. Assim e conjuntamente, a Selenium, a Selenium2, a Watir, a Windmill e a WebAii demonstraram possuir melhor ambiente de desenvolvimento, bem como uma API mais completa e intuitiva para este tipo de testes.

A maturidade e estabilidade das ferramentas revelaram estar fortemente relacionadas com as comunidades envolvidas, tempo de existência da ferramenta e empresas/projectos que as adoptaram. A título de exemplo, a Google e a Oracle encontram-se envolvidas com a Selenium e o Facebook e o Yahoo com a Watir.

Dada a dimensão da comunidade Selenium, foram já desenvolvidas externamente inúmeras extensões, sendo que muitas delas potenciazam enormemente o factor de automatização da ferramenta (é o caso do projecto Selenium Grid, já referido no capítulo no Estado da Arte).

Por conseguinte, tendo em conta todos os pontos referidos, a Selenium2 (mesmo estando em *beta*) será a ferramenta adoptada para a realização de controlo remoto dos diversos *browsers*.

3.2 Proposta de Desenvolvimento

A plataforma de automatização de testes deverá acompanhar o *tester* durante todo o processo de testes, desde a concepção até à análise dos resultados, passando obviamente pelos passos intermédios de especificação, implementação e execução.

A análise e experimentação anteriormente apresentadas permitiram identificar os pontos mais críticos para a automatização e aceleração da realização de Testes Funcionais a Aplicações Web. Assim, e nesta linha de raciocínio decidiu-se dividir o processo de

[5]http://code.google.com/p/selenium/wiki/IPhoneDriver
[6]http://code.google.com/p/selenium/wiki/AndroidDriver

testes em sub-processos semi-independentes: especificação de cenários de testes funcionais, definição de *locators* (identificadores de elementos numa página web) que irão operar nos testes, especificação de *suites* de execução, execução de testes e leitura/análise das execuções.

Resolveu-se optar por uma abordagem que maximizasse o factor de independência entre todos estes sub-processos. Esta decisão é apoiada pelo facto de assim uma operação de modificação/actualização afectar apenas pontualmente parte do processo (p.ex. a alteração da localização de um elemento não obriga a redefinição dos cenários de testes funcionais onde esse elemento é utilizado). Por outras palavras, a aposta numa abordagem pragmaticamente dinâmica visa a anulação de todos os pontos susceptíveis de repetição de operações. Em suma, pretende-se minimizar sempre que conveniente todos os processos que se apresentem como uma potencial perda de tempo sem valor tangível. A proposta de desenvolvimento não se caracterizará como sendo totalmente dinâmica, dado o receio que com uma separação excessiva se criem demasiados sub-processos, tornando a aprendizagem e concretização da plataforma, por parte dos seus utilizadores, demasiado complicada.

Em [And07], Jennitta Andrea introduz o conceito de um FTDE (*Functional Test Development Enviroment*), alimentado pelo universo dos mais modernos IDE's, como é o caso do IntelliJ e Eclipse, que fornecem um meio integrado de desenvolvimento. Actualmente, o meio de desenvolvimento de testes funcionais resume-se ao uso intervalado de editores de texto para escrita dos testes e uso de terminais para a sua execução. A ideia passa por criar novas sinergias, resultantes da junção das actuais potencialidades que integram os actuais IDE's, com as soluções de testes funcionais actuais e futuras.

Assim, tendo em mente as vantagens de um possível FTDE, e por o Eclipse ser um dos IDE's mais completo e aliado com o facto de este ser o ambiente de desenvolvimento adoptado na Cardmobili, equacionou-se a construção da plataforma de automatização de testes como um possível *plugin* ao Eclipse. Deste modo, não se está a falar apenas de um processo integrado de testes, mas da inclusão deste no próprio ambiente de desenvolvimento de *software*.

3.2.1 Arquitectura

Tendo em vista a concretização dos primeiros princípios até aqui apresentados, procedeu-se ao esboço e concepção da arquitectura que deverá suportar toda a plataforma de automatização de testes.

No diagrama presente na Figura 3.1, podem ser identificados três componentes principais:

- **Componente de Especificação**
- **Componente de Geração de Código e Compilação**
- **Componente de Execução**

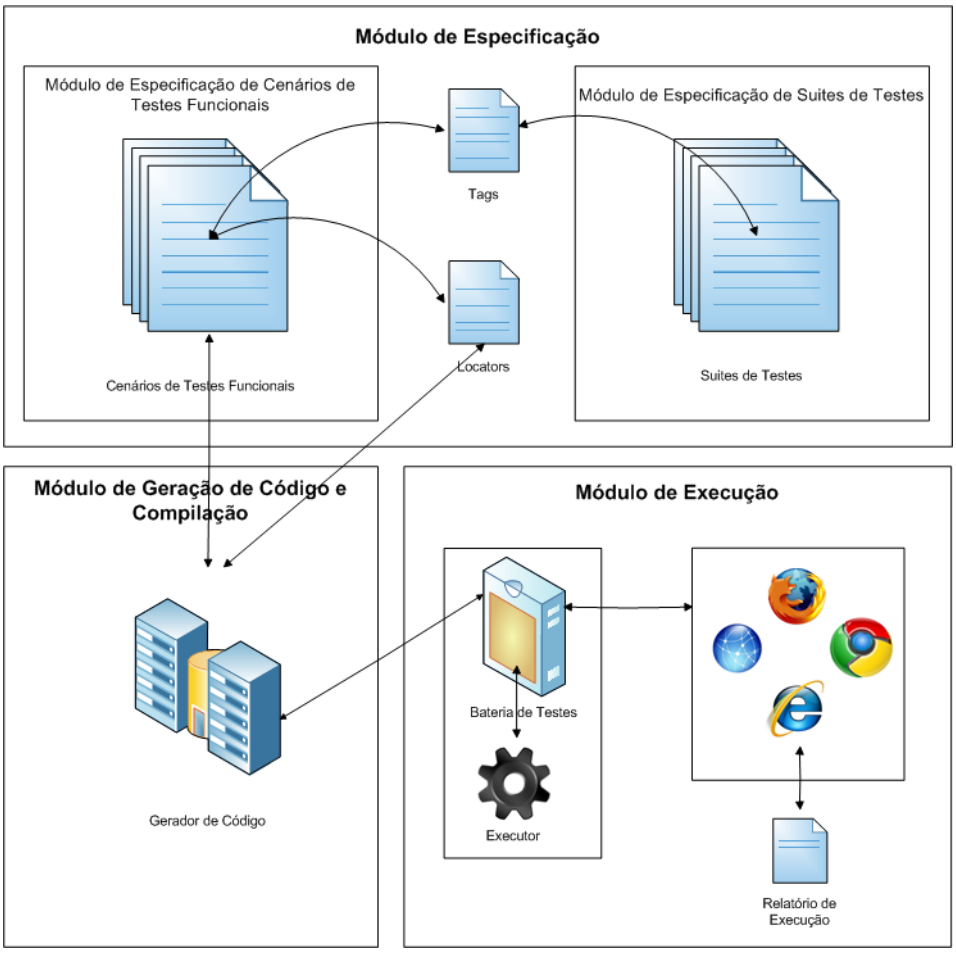

Figura 3.1: Diagrama de Arquitectura

O Componente de Especificação detém a lógica da concepção abstracta de cenários de testes funcionais e das *suites* de execução de testes. Este componente é constituído por dois módulos principais:

- **Módulo de Especificação de Cenários de Testes Funcionais** - Este módulo é responsável pela implementação do editor de criação de cenários de testes funcionais; a especificação deverá ser o mais abstracta possível, não devendo limitar a realização dos testes a um qualquer modelo de execução.
- **Módulo de Especificação de Suites de Execução de Testes** - Uma suite de execução define um conjunto de cenários de testes a realizar numa determinada execução, p.ex. todos os testes a um determinado módulo funcional de *software*, testes a funcionalidades implementadas numa determinada *release*, entre outros. O conjunto de testes deve ser constituído pela definição explícita de inclusão e/ou exclusão de cenários de teste específicos ou pela inclusão e/ou exclusão implícita de testes pertencentes a *tags*/grupos.

Este componente possui outros dois módulos: "*Locators*" e "*Tags*".

Em "*Locators*" encontram-se definidos todos os elementos da Aplicação Web que são referenciados nos testes. A concepção deste módulo como uma unidade independente, permite criar a separação da lógica dos elementos da especificação dos cenários de testes funcionais. Assim, sempre que for necessário modificar/actualizar a definição da localização de um elemento será apenas necessário operar num único sítio, evitando uma actualização repetitiva resultante da consequente replicação da informação, caso não existisse esta separação.

Uma "*Tag*" (ou grupo) é uma palavra que poderá caracterizar um cenário de teste funcional. A caracterização de cenários de teste permitirá que sejam criados grupos, segundo as suas *tags*. Deste nodo, palavras-chave como "*release* 4", "área de cliente", "sessões", "crítico" poderão criar grupos úteis que auxiliarão na definição de *Suites* de Execução de Testes.

O Componente de Geração de Código e Compilação é responsável pela transformação das especificações dos cenários de testes em código executável. A implementação gerada por este componente recorrerá à Selenium2 e à TestNG (uma das mais completas *framework* de testes do Java), para a criação de uma bateria de testes independente e auto-executável. Além disso, o componente deverá ser primeiramente implementado de uma forma abstracta, pois caso seja futuramente necessário mudar de ferramenta responsável pela controlo remoto do *browsers*, será apenas necessário mudar o domínio de conversão de instruções.

O Componente de Execução é o módulo que suporta as funcionalidades de execução de uma bateria de testes. Uma execução precisará apenas que lhe seja fornecida a especificação de uma *Suite* de Execução de Testes. Aplicará a lógica de filtros dos testes a executar, irá detectar os *browsers* presentes na máquina onde a bateria de testes se encontra, e então executará os testes. Finda a execução, a bateria deverá gerar relatórios de resultados. Os relatórios deverão conter toda a informação de execuções: que testes passaram, quanto tempo demoraram, que testes falharam (juntamente com possíveis indicadores para a razão da falha), entre outros dados relevantes. Adicionalmente, deverão ser anexados, por teste e *browser*, *screenshots* que apresentarão sequencialmente a execução dos cenários de teste. Os *screenshots*, além de fornecerem informação visual dos testes, poderão também servir para a detecção de possíveis problemas a nível da renderização da Aplicação Web.

3.2.2 Fluxo de Actividades

Com o objectivo de melhor expor o processo de construção e execução de testes, usando a plataforma a implementar, apresentar-se-ão de seguida diversos diagramas de actividade. Com o decorrer desta apresentação, aumentar-se-á progressivamente o detalhe dos ciclos de actividades propostos para a plataforma.

Primeiramente será dada uma visão geral das actividades disponibilizadas pela plataforma, depois serão apresentadas com maior detalhe as actividades implícitas neste primeiro diagrama: edição de cenários de testes funcionais, edição de *suites* de execução de testes, execução de testes e complementarmente a estas principais serão apresentadas as actividades paralelas.

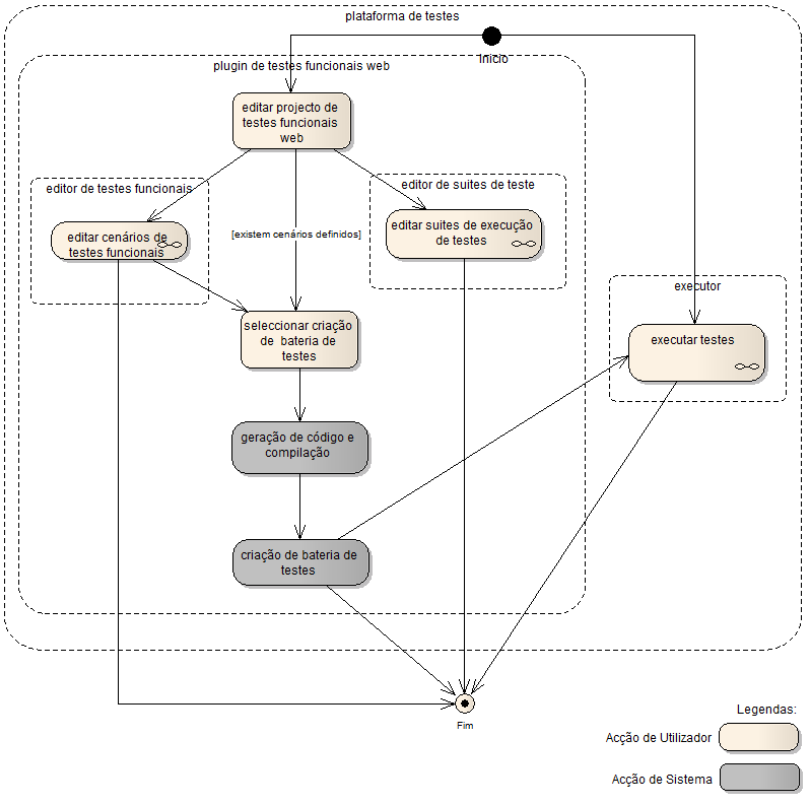

Figura 3.2: Diagrama de Actividade - *Plugin* (Plataforma de Automatização de Testes Funcionais a Aplicações Web)

No diagrama de actividades presente na Figura 3.2, podem ser identificados os principais componentes da plataforma, cada um com a actividade fundamental que suportam e que justifica a sua existência. No diagrama encontram-se representadas duas grandes regiões atómicas de actividades: o "*plugin* de testes funcionais" e a "bateria de testes". No *plugin* de testes funcionais o utilizador tem ao seu dispor as funcionalidades de edição de cenários de testes funcionais, edição de *suites* de execução de teste e geração e construção de uma bateria de testes, para consequente execução.

Como habitual no Eclipse, a plataforma de testes instanciará um projecto no IDE que, de acordo com uma estrutura, organizará toda a informação que alimenta o processo de testes. Recorrendo às funcionalidades já presentes no Eclipse, no que respeita ao Controlo de Versões, herdar-se-á a camada de gestão de desenvolvimento de testes usada no desenvolvimento de *software*. Deste modo, existirão revisões de testes identificadas por

release e por tempo, entre outros factores. Por conseguinte, haverá uma sincronização dos testes com o desenvolvimento de *software*, significando p.ex. a criação de pares: revisão do produto de *software* e revisão da bateria de testes.

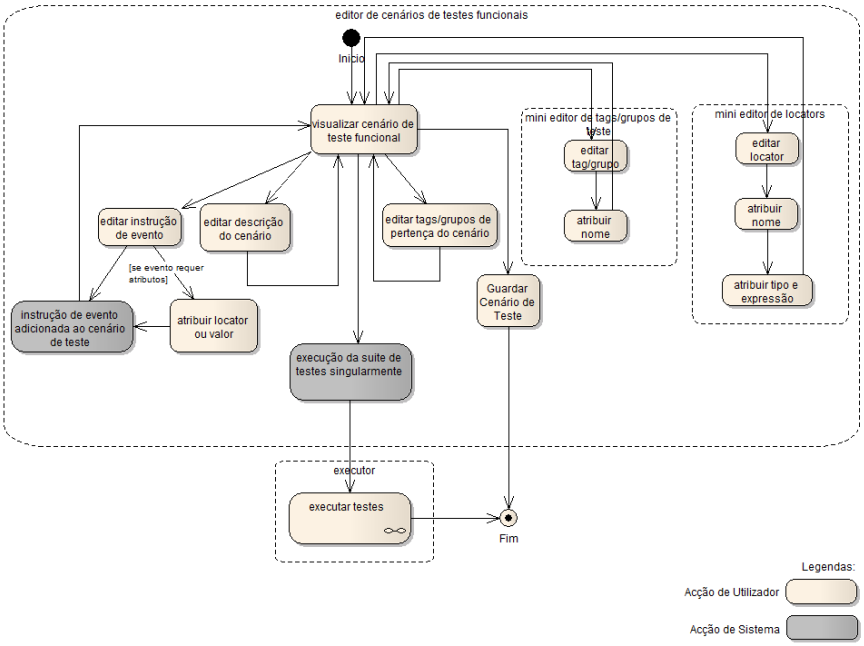

Figura 3.3: Diagrama de Actividade - Especificação de Cenários de Testes Funcionais

No editor de cenários de testes funcionais a Aplicações Web (Figura 3.3), o utilizador poderá editar iterativamente os eventos que constituirão um cenário de testes. Paralelamente, poderá também introduzir informação complementar aos eventos (valores de entrada, localização de elementos (*locators*), entre outros). Um utilizador poderá também caracterizar cenários de teste recorrendo às já referidas "*tags*", assim como escrever uma descrição textual auxiliar sobre o teste.

A criação e edição da informação relativa aos *locators* e *tags* poderá ser realizada nos mini-editores auxiliares que estarão presentes face-a-face com o editor principal de cenários de teste.

Ainda neste editor, um utilizador poderá pedir a execução do cenário de teste que se encontra a editar. Esta operação consistirá basicamente na execução de uma *suite* constituída apenas pelo cenário em questão.

A utilização do editor de suites de execução de testes consistirá apenas nas indicações de inclusão e/ou exclusão de cenários de testes específicos ou de conjuntos de cenários de teste caracterizados por uma determinada *tag*.

Similarmente ao editor de cenários de teste, um utilizador poderá pedir que seja realizada a execução da suite que se encontra a editar. Adicionalmente, poderá também pedir

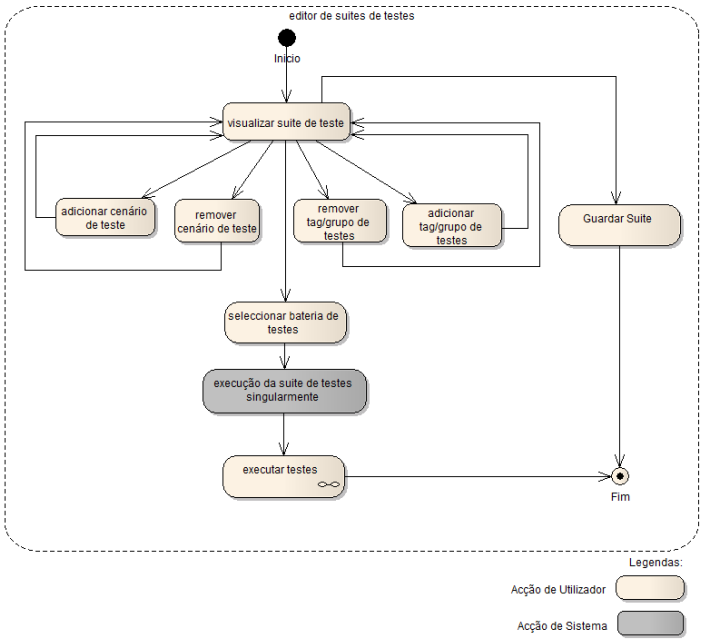

Figura 3.4: Diagrama de Actividade - Especificação de Suites de Execução de Testes

a execução de todas as suites de teste presentes no projecto de testes.

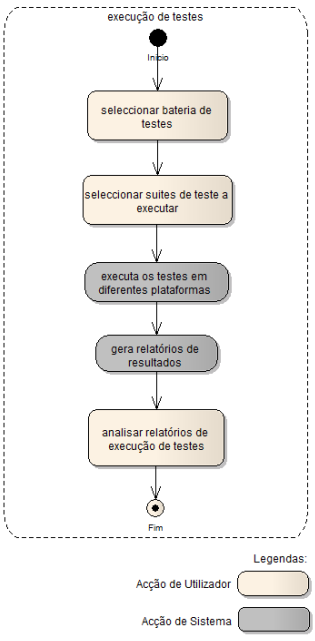

Figura 3.5: Diagrama de Actividade - Execução de Testes

Assim que uma bateria de testes é gerada e compilada, ser-lhe-á apenas necessário indicar, por meio de uma filtragem resultante da especificação da *suite* de execução fornecida, que cenários de teste deve executar. Antes da execução dos testes propriamente dita,

deverão ser identificados que *browsers* o sistema onde será realizada a execução possui, de modo a direccionar a realização dos testes para essas plataformas.

Da execução de testes resultarão diversos dados, que serão compilados num relatório. Este deverá apresentar em que *browsers* foram realizados os testes, quais as execuções que falharam e quais as que foram concretizadas com sucesso. A informação será complementada com as razões da falha na execução de testes, bem como a criação de uma galeria sequencial de *screenshots* de cada execução de um teste.

3.2.3 *Mockups* de interfaces

Definidas as principais iterações e a arquitectura geral da plataforma, passou-se ao estudo de *mockups* de interfaces para os vários componentes. Com o objectivo de facilitar a aprendizagem e a agilização no uso da plataforma de testes, resolveu-se apostar na concepção das interfaces.

Serão apresentados os *mockups* para as interfaces do Editor de Cenários de Testes Funcionais a Aplicações Web (Figura 3.6), Mini-Editor de *locators* (Figura 3.7), Mini-Editor de *tags* (Figura 3.8) e Editor de *Suites* de Execução de Testes (Figura 3.9), assim como os *mockups* para o Modelo de Relatório de Execução de Testes (Figura 3.10) e o Modelo de Apresentação de Sequência de *Screenshots* de Execução de Testes (Figura 3.11).

A plataforma definirá duas novas perspectiva do Eclipse[7], próprias para o Editor de Cenários de Testes Funcionais e o Editor de *Suites* de Execução de Testes. Ambas as perspectivas terão o editor propriamente dito ao centro e à sua esquerda o clássico explorador do projecto do Eclipse. No caso do Editor de Cenários de Testes existirá à direita uma vista para os mini-editores de *Locators* e *Tags*.

No Editor de Cenários de Teste a construção será realizada por meio de escolhas semi-assistidas, em que o utilizador escolherá que tipos de eventos constituirão o teste (recorrendo ao uso de *combobox*'s). No fim, o utilizador irá obter, em jeito de CNL[8], uma apresentação textual do teste. Os eventos poderão requerer o preenchimento de argumentos, que poderão ser dados simples ou então *locators*.

O Editor de *Suites* de Execução de Testes apresentará duas listas diferentes: a dos cenários de testes e a das *tags* presentes no projecto de teste. Destas listas, o utilizador poderá definir a sua *Suite* de Execução de Testes, recorrendo a acções de inclusão e/ou exclusão.

Após a execução de todos os testes será gerado um Relatório de Resultados. Este documento listará os *browsers* em que foram realizados os testes. Para cada *browser* serão listados os resultados dos testes, em forma tabular. A tabela incluirá diversa informação, como é o caso do tempo de execução do teste, se foi executado com sucesso, e em caso de falha qual a sua razão. Complementarmente, para cada execução dos testes, o utilizador

[7] http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html
[8] Controlled Natural Language

poderá visualizar a galeria sequencial de *screenshots*.

De seguida, serão listados os referidos *mockups*. Sempre que se mostrar necessário, será realizada a legendagem dos mesmos.

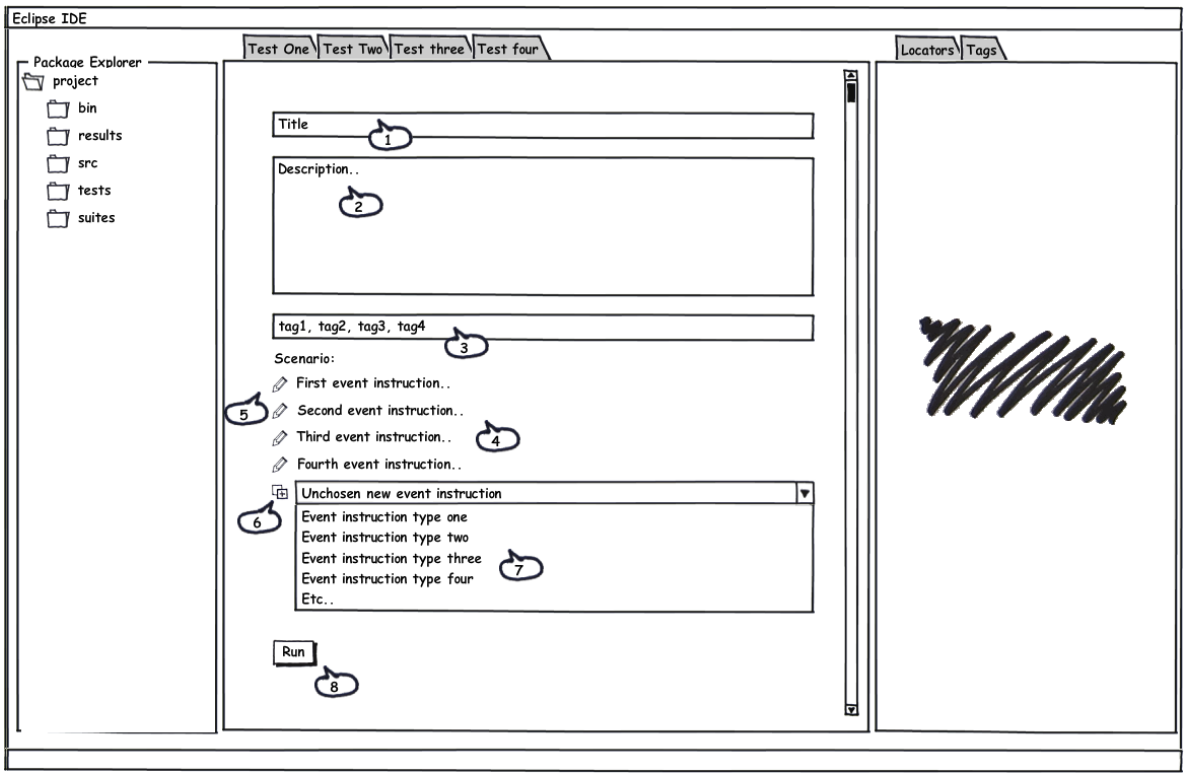

Figura 3.6: *Mockup* de interface - Editor de Cenários de Testes Funcionais a Aplicações Web

Legendagem (Figura 3.6):

1. Definição do Título do Cenário de Teste
2. Descrição textual do Cenário de Teste
3. *Tags* que caracterizaram o Cenário de Teste
4. Botão de Edição de Instruções de Eventos do Cenário
5. Eventos que constituem o Cenário de Teste em Edição
6. Botão de adição de novo evento
7. Escolha do tipo de evento
8. Botão de Execução

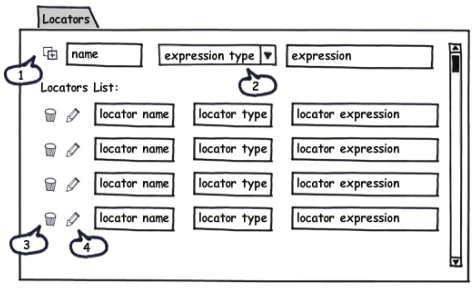

Figura 3.7: *Mockup* de interface - Mini-Editor de *locators* de elementos em Aplicações Web

Legendagem (Figura 3.7):

1. Botão de adição de novo *locator* ao projecto
2. Campos de definição de um *locator*
3. Botão de remoção de um *locator*
4. Botão de edição de dados de um *locator*

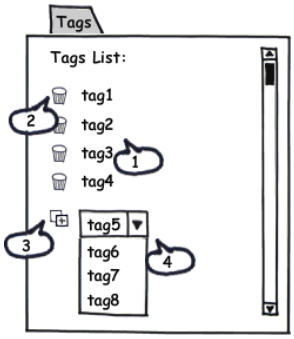

Figura 3.8: *Mockup* de interface - Mini-Editor de *tags* de Testes

Legendagem (Figura 3.8):

1. Listagem de *tags* que caracterizam o Cenário de Testes em edição
2. Remover *tag* da listagem
3. Adicionar a *tag* seleccionada na *combo-box* à listagem
4. *Combo-box* editável, com todas as *tags* existentes no projecto de testes. Podem ser criadas novas *tags* ao projecto na *combobox*, bastando para isso escrever a nova *tag* desejada.

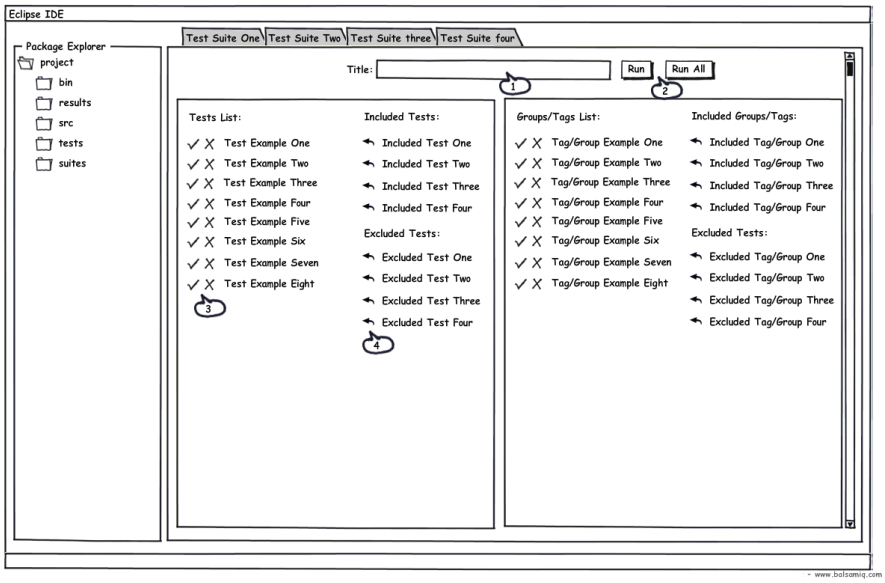

Figura 3.9: *Mockup* de interface - Editor de *Suites* de Execução de Testes

Legendagem (Figura 3.9):

1. Título da *Suite* de Execução de Testes
2. Botões que accionam a execução da *suite* em edição ou de todas as *suites* presentes no projecto
3. Botão de inclusão e exclusão de Testes e/ou *Tags*/Grupos
4. Botão de remoção de referência de Testes e/ou *Tags*/Grupos na *suite* em edição

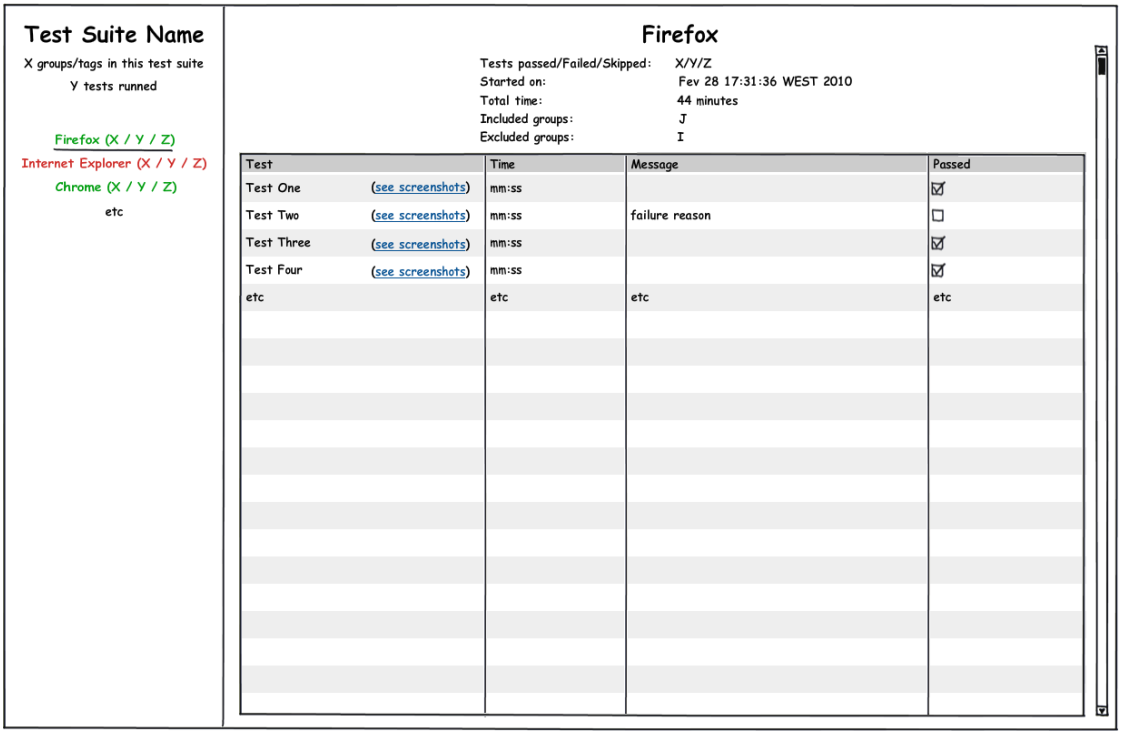

Figura 3.10: *Mockup* de Modelo de Relatório de Execução de Testes

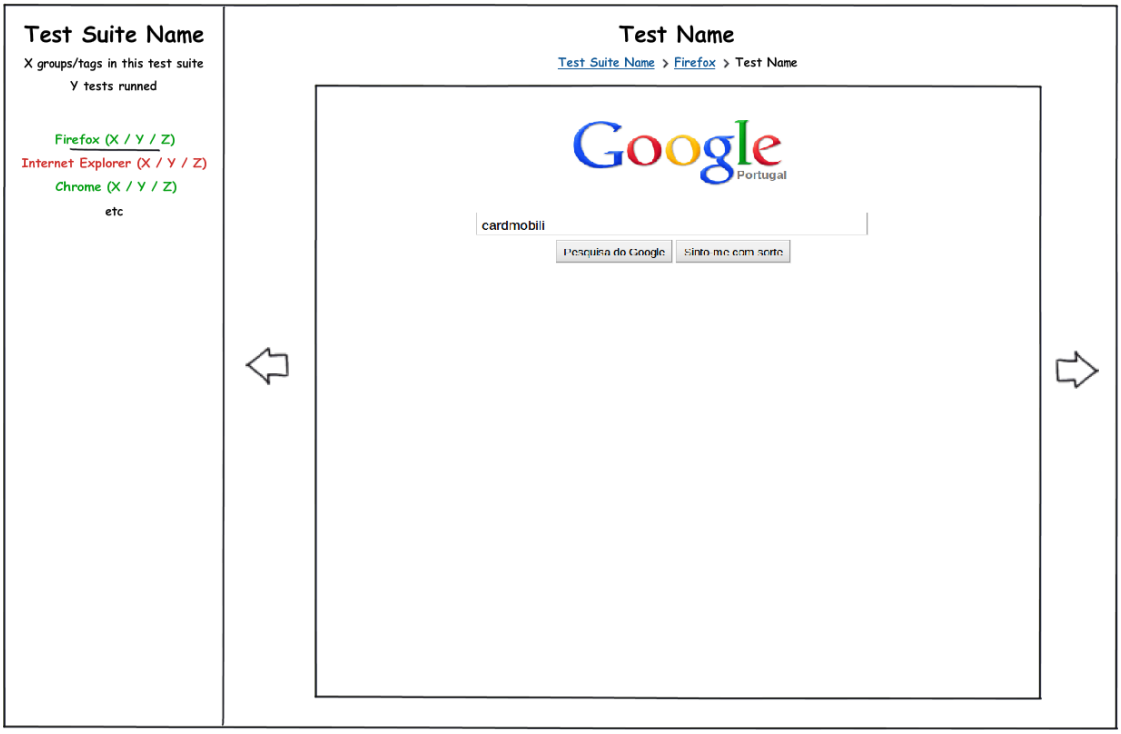

Figura 3.11: *Mockup* de Modelo de Apresentação de Sequência de *Screenshots* de Execução de Testes

Capítulo 4

Implementação de Solução Final

Neste capítulo serão apresentadas as principais etapas de implementação da solução final da plataforma de testes.

Começou-se por implementar o Editor de Cenários de Testes Funcionais a Aplicações Web, assim como as principais estruturas de dados que suportariam o *plugin*, os *wizards* de construção de um projecto para o *plugin* em questão e os *wizards* de criação de cenários de teste.

Seguidamente, passou-se à construção do gerador de código. Chegada a esta fase, já se procedeu à experimentação real de execução de testes funcionais. Foram experimentadas e analisadas abordagens e possíveis ferramentas de auxílio, e assim que se obteve uma estabilidade satisfatória no modelo de código gerado, procedeu-se à implementação do Editor de *Suites* de Execução de Testes.

Estando a base das *suites* completamente definida, procedeu-se a afinações finais no gerador de código para que a bateria de testes pudesse ser usada em qualquer plataforma e que a execução pudesse gerar relatórios com resultados com o tipo de informação pretendida.

Antes de qualquer desenvolvimento, foi implementado em Eclipse um novo tipo de projecto que deveria representar um *workspace* da Plataforma de Automatização de Testes a Aplicações Web.

Um projecto deste tipo é constituído pelas seguintes pastas:

- **bin** - compilação mais recente do código da bateria de testes gerado
- **jars** - baterias de testes compiladas e executáveis
- **results** - relatórios de execuções realizadas a partir do Eclipse
- **src** - código fonte gerado a partir da especificação dos testes
- **suites** - especificações de *suites* de execução de testes
- **tests** - especificações de cenários de testes funcionais

Um projecto contém dois ficheiros na raiz do projecto:

- **locators.xml** - definição de *locators* (ver exemplo de conteúdo no Anexo F)
- **tags.xml** - listagem de *tags* (ver exemplo de conteúdo no Anexo G)

4.1 Editor de Cenários de Testes Funcionais a Aplicações Web

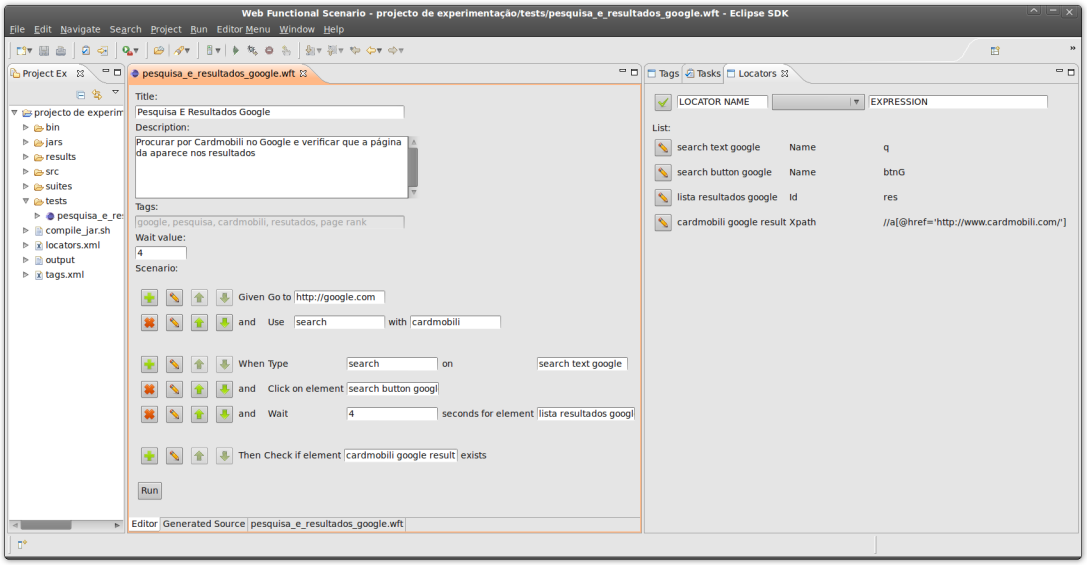

Figura 4.1: *Screenshot* de Editor de Cenários de Testes Funcionais a Aplicações Web

A construção deste editor foi precedida por uma análise que consistia na escolha de um modelo de como deveria ser representado e apresentado um cenário. Das diversas possibilidades estudadas, a escolha recaiu sobre um padrão de definição de cenários normalmente usado em BDD (*Behavior Driven Development*) [Dan10a].

4.1.1 Behavior Driven Development

O *Behavior Driven Development* é uma técnica que tem como principal objectivo encorajar a colaboração entre os *developers*, QA (*Quality Assurance*) *testers* e elementos não técnicos da empresa. Tal como o nome sugere, esta técnica concentra a sua atenção na obtenção de uma clara compreensão do comportamento que o *software* deve reproduzir.

Testes escritos segundo os critérios BDD, devem ser escritos em linguagem natural, de forma que estes sejam compreendidos (ou mesmo escritos) por qualquer elemento da empresa. O uso da linguagem natural aliada à adopção de um modelo de escrita orientada à descrição do comportamento e funcionalidades que um pedaço de *software* deve conter, maximiza e centraliza a atenção no propósito para o qual o *software* é criado, reduzindo ao mínimo a existência de detalhes técnicos.

Estas descrições são normalmente concebidas recorrendo à criação de exemplos que descrevem o comportamento, tendo como base cenários demonstrativos. Segundo Dan North, em [Dan10b], a estrutura recomendada segue o seguinte padrão:

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
  And [some more context]...
When  [event]
Then  [outcome]
  And [another outcome]...

Scenario 2: ...
```

Figura 4.2: Exemplo de estruturação do padrão BDD

Como pode ser observado, começa-se por identificar (em "*Narrative*") a funcionalidade ("*feature*"), quem usufruirá dela ("*role*") e a que necessidade pretende responder ("*benefit*").

Geralmente o único modo de se visualizar o comportamento de um produto de *software* é a partir da sua interface que se traduz normalmente numa *GUI*. A estrutura serve-se de um modelo de descrição de cenários para demonstrar como é que o *software* deve funcionar.

Este é o modelo que será adoptado para a descrição dos cenários de testes funcionais para a plataforma. Na especificação de cenários, Dan North recomenda que a descrição deva ser efectuada recorrendo ao uso dos termos "*Given*", "*When*" e "*Then*".

As instruções *Given* devem definir o contexto do cenário, p.ex. dados, área do *software* em que o texto irá ser executado, entre outros.

Em *When* devem ser definidos todos os eventos que irão efectivar as acções que o cenário de teste deve verificar. As instruções devem ser simples e em conjunto validar a funcionalidade em teste.

Por fim, em *Then* devem ser realizadas as validações dos dados resultantes da execução dos eventos descritos no cenário de teste.

Com este modelo em mente, decidiu-se criar um domínio de eventos Web, que deverá conter todas as acções susceptíveis de ser realizadas numa Aplicação Web. Como a Web está em constante mudança, a implementação do domínio deverá facilitar a introdução de novas acções, bem como a possibilidade de acções compostas.

4.1.2 Domínio de Acções em Aplicações Web

O domínio criado divide-se em conjuntos de eventos simples, eventos compostos e extensões complementares. Um evento simples representa uma única acção realizada por um

utilizador numa Aplicação Web, enquanto que acções compostas representam conjuntos de duas ou mais acções simples. As extensões complementares representam possíveis rotinas auxiliares na realização dos testes. No protótipo foram implementadas duas extensões deste tipo: uma que permite verificar se todas as imagens foram bem carregadas e renderizadas no *browser* e outra que verifica se alguma hiperligação presente numa pagina Web aponta para um local na Web inexistente (ou seja, o erro HTTP 404).

No protótipo encontra-se implementado o seguinte domínio de acções Web (são apresentados o identificador, frase representativa no editor e seu significado):

- ***GOTO***

  Frase no Editor: *The GOTO statement represents the instruction*
  Acção que representa a escrita de um endereço num *browser* e consequente pedido de navegação para o mesmo.

- ***USE_VAR***

  Frase no Editor: *Use [VAR] with [VALUE]*
  Definição de uma variável com um dado associado.

- ***SUCESSION_OF***

  Frase no Editor: *Sucession of [SCENARIO]*
  Instrução com a função similar à de um *Helper* que chama um outro teste, de modo a conseguir o estado que se obtém com a execução do mesmo. É exemplo disso a chamada do Cenário de Teste a um iniciação de sessão numa Aplicação Web, para a colocação do teste em construção no estado de autenticado na aplicação.

- ***BACK***

  Frase no Editor: *Back is pressed*
  Acção que representa a selecção do botão Retroceder do *browser*.

- ***FORWARD***

  Frase no Editor: *Forward is pressed*
  Acção que representa a selecção do botão Avançar do *browser*.

- ***REFRESH***

  Frase no Editor: *Refresh is pressed*
  Acção que representa a selecção do botão Actualizar do *browser*.

- ***WAIT_FOR***

  Frase no Editor: *WAIT [NUMBER] seconds for element [LOCATOR]*
  Rotina que espera pela renderização do "LOCATOR" até ao máximo de "NUMBER" segundos; em caso de insucesso, o teste falha.

- ***ASSERT_EXISTS***

  Frase no Editor: *Check if element [LOCATOR] exist*

Asserção que tem por objectivo verificar a existência de um elemento numa página Web.

- ***ASSERT_NOT_EXISTS***

  Frase no Editor: *Check if element [LOCATOR] do not exist*
  Asserção que tem por objectivo verificar a não existência de um elemento numa página Web.

- ***ASSERT_TEXT_EQUAL***

  Frase no Editor: *Check if element [LOCATOR]'s text is equal to [VALUE]*
  Asserção que tem por objectivo verificar se o conteúdo do texto de um elemento numa página Web é igual a "VALUE".

- ***ASSERT_TEXT_NOT_EQUAL***

  Frase no Editor: *Check if element [LOCATOR]'s text is not equal to [VALUE]*
  Asserção que tem por objectivo verificar se o conteúdo do texto de um elemento numa página Web não é igual a "VALUE".

- ***ASSERT_ATTR_EQUAL***

  Frase no Editor: *Check if attribute [ATTR] of [LOCATOR] is equal to [VALUE]*
  Asserção que tem por objectivo verificar se o conteúdo de um atributo de um elemento numa página Web é igual a "VALUE".

- ***ASSERT_NOT_ATTR_EQUAL***

  Frase no Editor: *Check if attribute [ATTR] of [LOCATOR] is not equal to [VALUE]*
  Asserção que tem por objectivo verificar se o conteúdo de um atributo de um elemento numa página Web não é igual a "VALUE".

- ***ASSERT_IS_VISIBLE***

  Frase no Editor: *Check if element [LOCATOR] is visible*
  Asserção que tem por objectivo verificar se um elemento numa página Web se encontra visível para o utilizador.

- ***ASSERT_NOT_VISIBLE***

  Frase no Editor: *Check if element [LOCATOR] is not visible*
  Asserção que tem por objectivo verificar se um elemento numa página Web não se encontra visível para o utilizador.

- ***ASSERT_ITEM_SELECTED***

  Frase no Editor: *Check if the selected item [LOCATOR] is selected*
  Asserção que tem por objectivo verificar se o elemento numa página Web se encontra seleccionado de entre os vários elementos de uma lista.

- ***CLICK***

  Frase no Editor: *Click on element [LOCATOR]*
  Acção que representa um *click* sobre o elemento indicado.

- ***HOVER***

  Frase no Editor: *Mouse over on [LOCATOR]*
  Acção que representa a passagem do rato sobre o elemento indicado.

- ***TYPE***

  Frase no Editor: *Type [VALUE] on [LOCATOR]*
  Acção que representa a escrita de "VALUE" no elemento indicado.

- ***SEND_KEYS***

  Frase no Editor: *Send keys [KEY COMBINATIONS] to [LOCATOR]*
  Acção que representa a selecção de uma combinação de teclas sobre o elemento indicado.

- ***CLOSE_BROWSER***

  Frase no Editor: *Close browser*
  Acção que representa o fecho de um *browser*.

- ***OPEN_BROWSER***

  Frase no Editor: *Open browser*
  Acção que representa a inicialização de um novo *browser*.

- ***DELETE_ELEMENT_TEXT***

  Frase no Editor: *Delete text of element [LOCATOR]*
  Acção que representa a eliminação de texto que se encontra escrito no elemento indicado, por meio de uso da tecla *backspace*.

- ***SELECT_ITEM***

  Frase no Editor: *Select item [ITEM] on element [LOCATOR]*
  Acção que representa a selecção de um item contido numa lista do elemento indicado.

- ***CLICK_AND_WAIT***

  Frase no Editor: *Click on element [LOCATOR] and wait [TIME] seconds*
  Acção composta que representa um *click* sobre o elemento indicado, seguido da espera dos segundos indicados.

- ***WAIT***

  Frase no Editor: *Wait [TIME] seconds*
  Acção que representa a espera do tempo indicado.

- ***TYPE_AND_WAIT***

  Frase no Editor: *Type [VAR] on element [LOCATOR] and wait [TIME] seconds*
  Acção composta que representa a escrita de "VALUE" no elemento indicado, seguido da espera dos segundos indicados.

- ***CHECK_IMAGES***

  Frase no Editor: *Check if all images are alive*

Extensão complementar que, por meio de uma rotina, verifica se todas as imagens (elementos <img>) presentes numa página Web foram carregadas com sucesso.

- ***CHECK_LINKS***

  Frase no Editor: *Check if all links are alive*
  Extensão complementar que por meio de uma rotina, verifica se todas as hiperligações (elementos <a>) presentes numa página Web apontam para um local válido na Web (erro HTTP 404).

A selecção no editor é realizada recorrendo a uma *combobox*, contendo as frases referidas, como pode ser observado no *screenshot* presente no Anexo H.

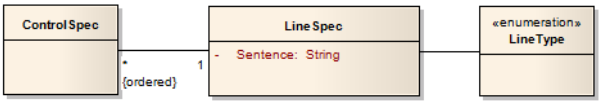

Figura 4.3: Classe LineSpec

As acções encontram-se definidas recorrendo à classe *LineSpec* (Figura 4.3 [1]) que, de um modo abstracto, define o formato de um elemento do domínio de acções Web. Um elemento é instanciado pela definição dos dados do formato de *LineSpec*: frase de descrição, tipo de frase (identificador/*enum LineType*), e controlos que o definem (juntamente com os argumentos necessários).

4.1.3 Estrutura de Dados de Cenário de Testes Funcionais

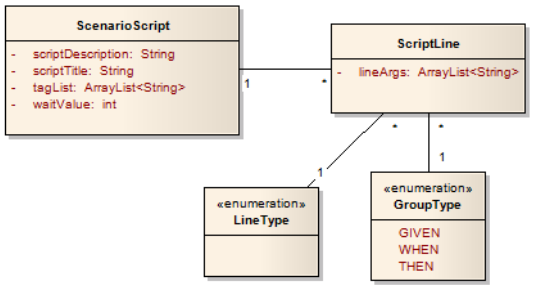

Figura 4.4: Classe ScenarioScript

A estrutura de dados que armazena toda a informação necessária à especificação completa de um Cenário de Testes Funcionais a Aplicações Web pode ser consultada na Figura 4.4. Foi também criada uma especificação para que um Cenário de Testes fosse facilmente serializado (carregado e exportado) para um ficheiro XML. A título de exemplo é de seguida apresentada a serialização de um Cenário de Teste.

[1] o *enum LineType* e a class *ControlSpec* não se encontram totalmente desenvolvidos por razões de leitura, devido ao seu tamanho

```
<scenario>
   <wait>4</wait>
   <title>Pesquisa E Resultados Google</title>
   <description>Procurar por Cardmobili no Google e
        verificar que a pagina da aparece nos resultados </description>
   <tags>
      <tag>google</tag>
      <tag>pesquisa</tag>
      <tag>cardmobili</tag>
      <tag>resutados</tag>
      <tag>page rank</tag>
   </tags>
   <lines>
      <line group="GIVEN" type="GOTO">
         <args>
            <arg>http://google.com</arg>
         </args>
      </line>
      <line group="GIVEN" type="USE_VAR">
         <args>
            <arg>search</arg>
            <arg>cardmobili</arg>
         </args>
      </line>
      <line group="WHEN" type="TYPE">
         <args>
            <arg>search</arg>
            <arg>search text google</arg>
         </args>
      </line>
      <line group="WHEN" type="CLICK">
         <args>
            <arg>search button google</arg>
         </args>
      </line>
      <line group="WHEN" type="WAIT_FOR">
         <args>
            <arg>4</arg>
            <arg>lista resultados google</arg>
         </args>
      </line>
      <line group="THEN" type="ASSERT_EXISTS">
         <args>
            <arg>cardmobili google result</arg>
         </args>
      </line>
   </lines>
</scenario>
```

Figura 4.5: Exemplo de Serialização de um Cenário de Teste

4.2 Geração de Código

Com os Cenários de Teste implementados e armazenados em estruturas de dados criadas para o efeito, procedeu-se à implementação do Componente de Geração de Código.

Para o efeito, foram criadas duas classes principais (Figura 4.6[2]):

- **Translater** - Classe abstracta que define todos os métodos que a implementação de um tradutor deve definir. Cada método possui um tipo de acção do domínio de acções Web correspondente.

[2]No diagrama as classes *Translater*, *TranslaterSelenium* e *ScenarioScript* foram simplificadas devido a questões de legibilidade

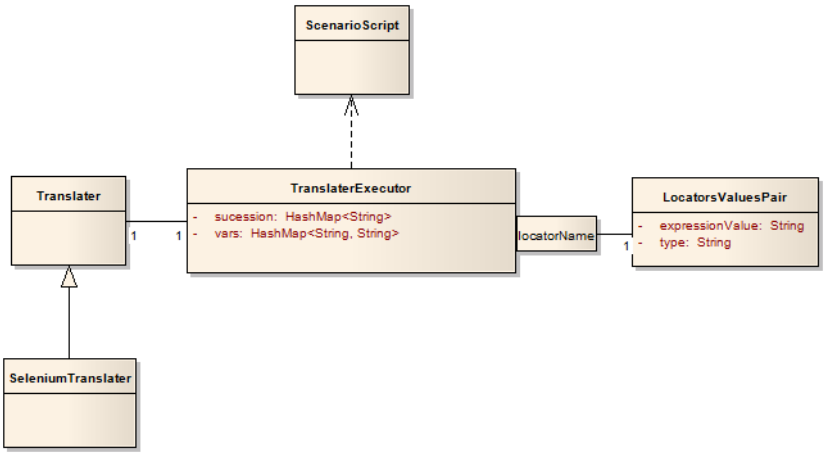

Figura 4.6: Classes Translater e TranslaterExecutor

- **TranslaterExecutor** - Classe que, servindo-se de uma implementação da classe abstracta *Translater*, executa a geração de código para todos os cenários de testes especificados num projecto.

Dado que a escolha da ferramenta que permite o controlo remoto de um *browser* recaiu sobre a Selenium2, foi criada a classe SeleniumTranslater que extende a classe Translater, fazendo *override* a todos os métodos necessários à implementação de um tradutor.

Foi criado um modelo programático de testes que combina a Selenium2 com a *framework* de testes TestNG [BS07], estruturado como ilustrado na Figura 4.7:

No diagrama apresentado podem ser identificadas diversas classes. A maioria são criadas estaticamente e definem o ambiente de execução que acompanhará as classes que serão geradas dinamicamente pelo gerador de código. Este módulo gera uma classe por cenário de teste presente no projecto, atribuindo-lhe como nome o título do cenário correspondente. Estas classes geradas extendem a classe WebDriverTestBase e são etiquetadas com a anotação *Test*[3] do TestNG. Esta anotação define no TestNG, um método como sendo um teste susceptível de ser executado.

A classe WebDriverTestBase é constituída por diversas métodos, etiquetados com anotações *BeforeMethod*[4] do TestNG. Cada um destes métodos inicia uma implementação da classe WebDriver[5]. As implementações aquando da escrita deste documento em uso são as seguintes: HtmlUnitDriver[6], ChromeDriver[7], InternetExplorerDriver[8] e FirefoxDriver[9].

[3]http://testng.org/javadoc/org/testng/annotations/Test.html
[4]http://testng.org/javadoc/org/testng/annotations/BeforeMethod.html
[5]http://selenium.googlecode.com/svn/trunk/docs/api/java/org/openqa/selenium/WebDriver.html
[6]http://code.google.com/p/selenium/wiki/HtmlUnitDriver
[7]http://code.google.com/p/selenium/wiki/ChromeDriver
[8]http://code.google.com/p/selenium/wiki/InternetExplorerDriver
[9]http://code.google.com/p/selenium/wiki/FirefoxDriver

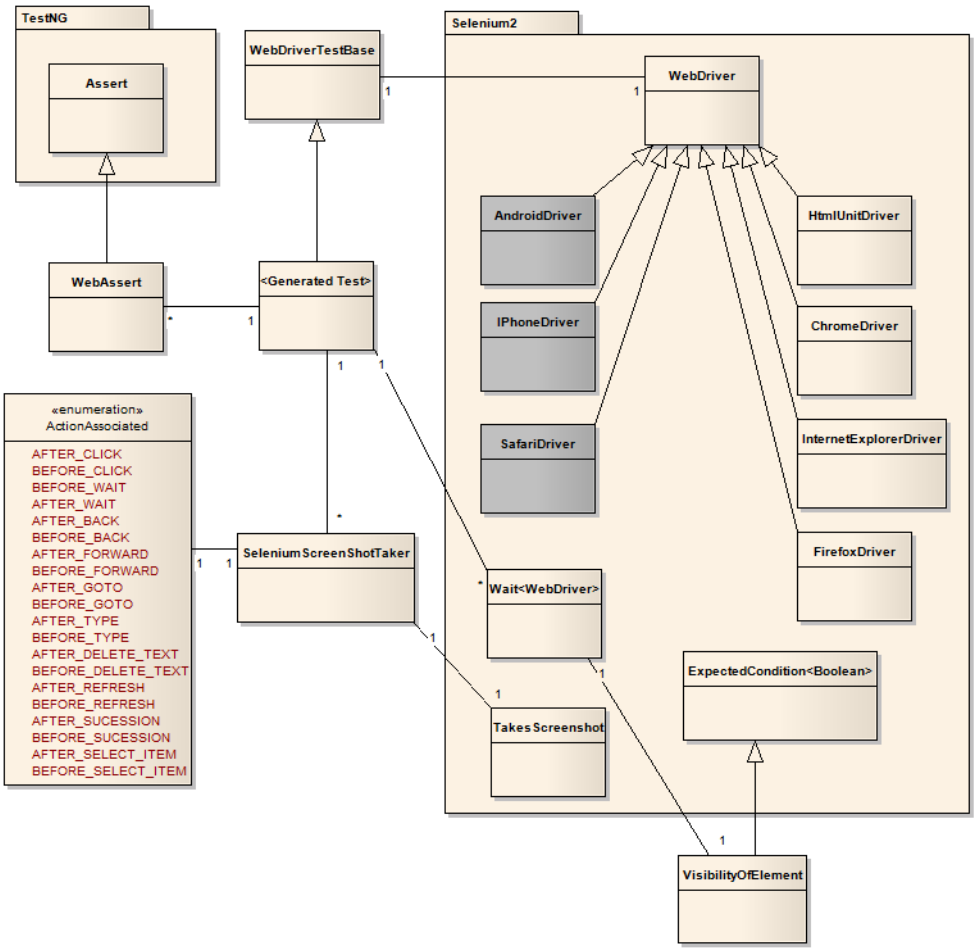

Figura 4.7: Estruturação do Módulo resultante do Gerador de Código

No entanto, irão ser incluídas as seguintes implementações, actualmente em desenvolvimento pela equipa da Selenium2: AndroidDriver[10], IPhoneDriver[11] e SafariDriver[12].

Um teste gerado tem ao seu dispor diversas funcionalidades implementadas pelas seguintes classes: WebAssert, SeleniumScreenShotTaker e Wait<WebDriver>.

A WebAssert define um novo leque de asserções, vocacionadas para o ambiente Web, implementadas de propósito para serem incluídas na plataforma de testes, combinadas com as funcionalidades da Selenium2. Entre outras, encontram-se disponíveis os seguintes métodos: AssertElementExists, AssertElementNotExists, AssertTextEqual, AssertTextNotEqual, AssertAttrEqual, AssertAttrNotEqual, AssertItemSelected e AssertItemNotSelected. Optou-se por não criar este tipo de asserções de um modo genérico, à semelhança da classe Translater, porque algumas ferramentas (como a Watir e a Windmill) já oferecem

[10]http://code.google.com/p/selenium/wiki/AndroidDriver
[11]http://code.google.com/p/selenium/wiki/IPhoneDriver
[12]http://github.com/mfazekas/safaridriver

este tipo de funcionalidade. Assim, decidiu-se construir como parte integrante da implementação conjunta em Selenium2 e TestNG (extendendo a classe Assert[13] desta *framework* de testes).

A classe SeleniumScreenShotTaker encapsula os métodos de captura de *screenshots* em qualquer *browser*, por meio do seu WebDriver correspondente. Todo o *screenshot* possui uma acção associada, identificada na enumeração ActionAssociated.

Por forma a auxiliar a concretização de esperas controladas pela presença de elementos numa página Web, foi criada a classe VisibilityOfElement. Esta classe servirá de método de verificação para a classe Wait<WebDriver>[14].

4.3 Editor de *Suites* de Execução de Testes

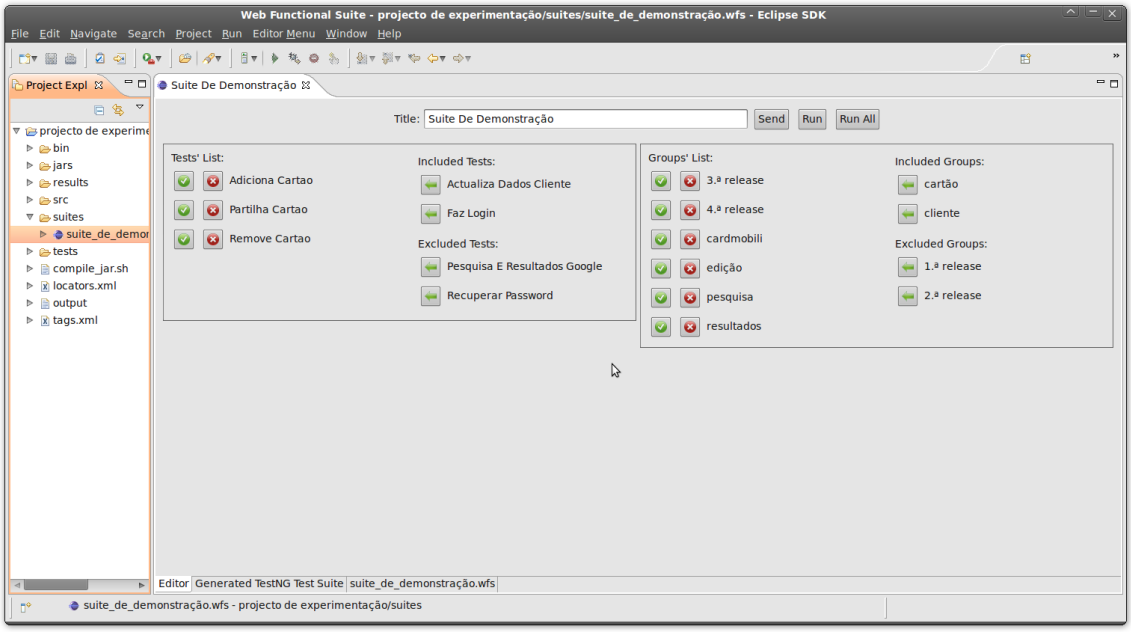

Figura 4.8: *Screenshot* de Editor de Suites de Execução de Testes

Neste editor, o utilizador define apenas a constituição de uma execução de testes por meio de inclusão e/ou exclusão de cenários de teste específicos ou pela inclusão e/ou exclusão implícita de testes pertencentes a *tags*/grupos.

A estrutura de dados que suporta este editor é bastante simples. Consiste apenas num conjunto de quatro listas, onde estão definidas as inclusões e exclusões de cenários

[13] http://testng.org/javadoc/org/testng/Assert.html
[14] http://selenium.googlecode.com/svn/trunk/docs/api/java/index.html

de teste e *tags*/grupos, respectivamente. As suites de execução de testes são serializadas (carregado e exportado) para um ficheiro XML, conforme o seguinte exemplo:

```
<suite>
   <scenarios_included>
      <scenario_included>Actualiza Dados Cliente</scenario_included>
      <scenario_included>Faz Login</scenario_included>
   </scenarios_included>
   <scenarios_excluded>
      <scenario_excluded>Pesquisa E Resultados Google</scenario_excluded>
      <scenario_excluded>Recuperar Password</scenario_excluded>
   </scenarios_excluded>
   <groups_included>
      <groups_included>cartao</groups_included>
      <groups_included>cliente</groups_included>
   </groups_included>
   <groups_excluded>
      <groups_excluded>1 release</groups_excluded>
      <groups_excluded>2 release</groups_excluded>
   </groups_excluded>
</suite>
```

Figura 4.9: Exemplo de Serialização de uma *Suite* de Execução de Testes

Na presença de uma bateria de testes, esta definição será interpretada e usada para a realização de operações de teoria dos conjuntos, de modo a filtrar os quais cenários de teste a executar.

4.4 Execução de Testes e Relatórios de Resultados

A implementação foi realizada tendo como base o motor de execução de testes do TestNG[15], adoptando-o e extendendo-lhe funcionalidades.

Uma bateria de testes gerada e compilada pela plataforma desenvolvida pode ser usada integrada com o Eclipse ou externamente em modo independente. A execução requer apenas que lhe seja indicada a *Suite* ou a localização de uma pasta que contenha diversas *Suites* de Execução de Teste.

De seguida, uma rotina detecta quais os *browsers* instalados na máquina em que a execução se processará. A detecção de *browsers* está implementada para funcionar em três ambientes diferentes: Microsoft Windows, Mac OS X e nas diversas distribuições de Linux. Esta funcionalidade tem como objectivo eleger que *Webdrivers* da Selenium2 irão ser usados na execução de testes. Caso não sejam detectados quaisquer *browsers* reais instalados, a execução irá ser realizada usando apenas o HtmlUnit.

Assim que é realizada a eleição das plataformas de execução, é processado o algoritmo que filtra os cenários de teste que irão ser executados, tendo em conta as operações de teoria de conjuntos. Esta funcionalidade foi implementada em *BeanShell*[16], dada a possi-

[15] http://testng.org/javadocs/org/testng/TestNG.html
[16] http://www.beanshell.org/

bilidade de assim se poder extender funcionalidades no TestNG. O *script* em BeanShell é gerado dinamicamente em *runtime*, de acordo com a especificação da *suite*, para então ser injectado no motor do TestNG.

Chegado a este passo, a execução de testes usa a bateria de testes para a sua concretização. Por outras palavras, a preparação constituída pela eleição dos *browsers* e a geração da implementação dinâmica da teoria de conjuntos da especificação da *suite* para execução, é fornecida ao motor do TestNG que se encontra incluído na bateria de testes. Depois, os testes são executados em modo sequencial, *browser* a *browser* e então teste a teste. A execução de testes do TestNG gera documentação (Relatórios de Execução) que, de modo a apresentar a informação desejada, é também extendida pela plataforma. Esta extensão inclui dois pontos fundamentais: os *screenshots* descritivos assim como a inclusão de informação auxiliar como é o caso de certas asserções (*CHECK_LINKS* e *CHECK_IMAGES*). O modelo de implementação foi pensado de modo a organizar as execuções por *browsers*, de modo a que esta organização seja reflectida no correspondente Relatório de Execução (Figura 4.10).

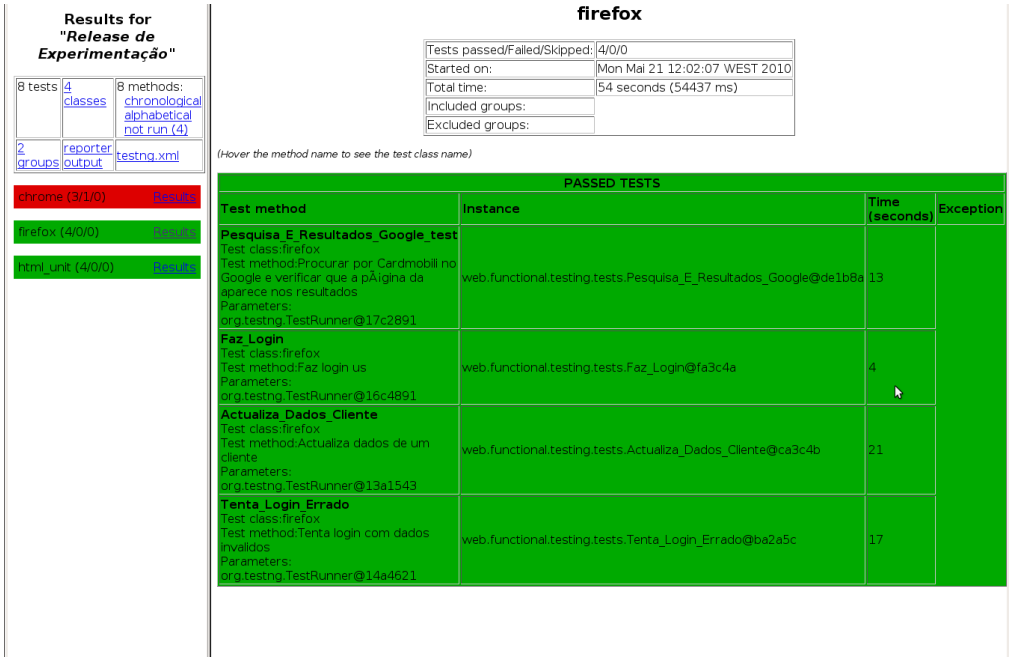

Figura 4.10: *Screenshot* de exemplo de Relatório de Resultados

Os Relatórios de Resultados são armazenados na pasta *Results* (execução a partir do Eclipse) ou na pasta onde a bateria de testes de encontra (execução independente). É criada uma pasta com o nome da *Suite* de Execução com a data e hora da execução. Dentro desta pasta encontram-se páginas em (x)html com o conteúdo dos relatórios propriamente ditos, assim como diversas pastas, cada uma identificada com o nome dos diferentes cenários de teste executados. Os referidos *screenshots* encontrar-se-ão neste local, dentro de

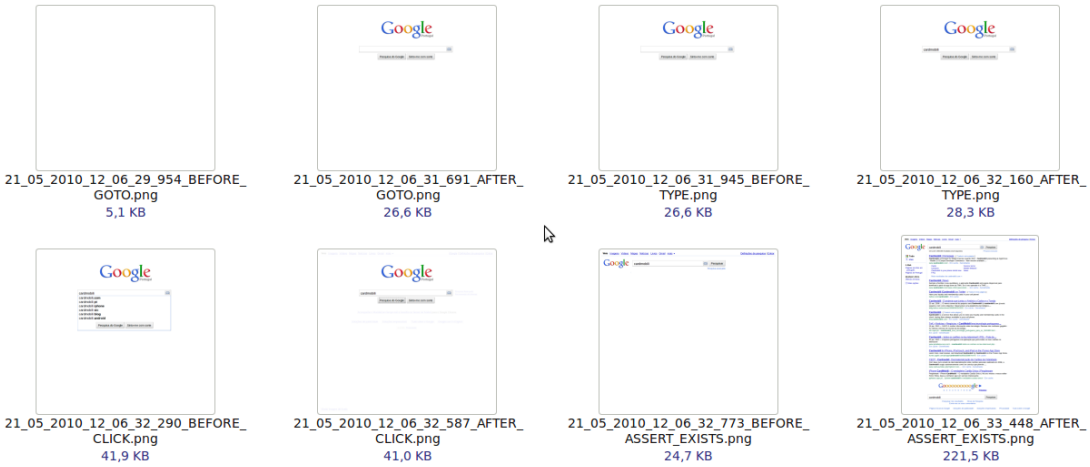

Figura 4.11: Galeria de Miniaturas de *Screenshots* resultante da execução de um Cenário de Teste

pastas identificadas com o nome do *browser* em que foram capturados os *screenhosts*. Os *screenshots* encontram-se ordenados por ordem de captura, permitindo assim a visualização na ordem em que foram realizados os eventos correspondentes (ver Figura 4.11).

Capítulo 5

Perspectivas de Desenvolvimento Futuro

Neste capítulo serão apresentadas futuras ideias e desenvolvimentos, acompanhados por esboços de concepção preliminares.

As ideias de desenvolvimento que irão ser apresentadas têm como principais objectivos:

- dotar a plataforma de uma arquitectura de execução de testes que reproduza de maneira mais fiel possível o ambiente das Aplicações Web
- melhoramento de sub-processos da plataforma de automatização de testes.

Para tal, foram pensadas quatro ideias principais que poderão ser alvo de implementação futura:

- Sistema Distribuído de Execução de Testes
- Dados de Teste
- Criador de *locators*
- Execução de Testes com recolha de *logs* da Aplicação Web

5.1 Sistema Distribuído de Execução de Testes

A experiência de utilização de uma Aplicação Web pode ser drasticamente afectada pela velocidade de recepção de conteúdo, falhas na ligação (originando *timeouts*), entre outros factores. De forma a envolver estes factores na execução de testes, pensou-se em distribuir a execução por diferentes plataformas e locais do globo terrestre. Deste modo, poder-se-ia verificar se, p.ex. a utilização da Aplicação Web por parte de um utilizador japonês é similar à de um utilizador português, se a Aplicação Web funciona dentro dos limites temporais estabelecidos, etc.

Assim, resolveu-se conceber um modo de expansão do componente de execução de testes da plataforma até aqui construída.

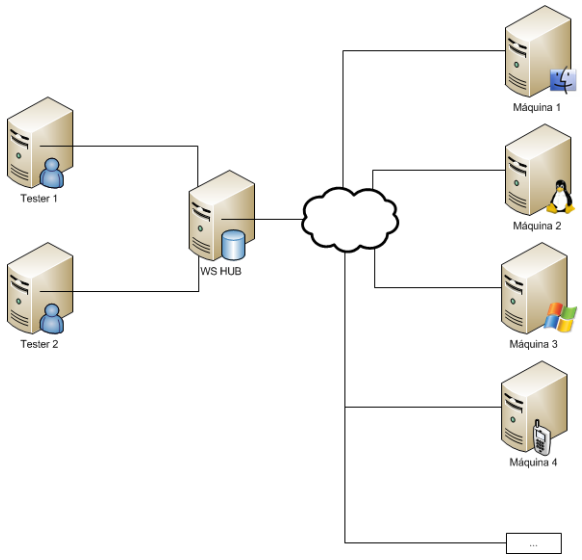

Figura 5.1: Diagrama de Arquitectura do Sistema Distribuído de Execução de Testes

A solução proposta introduz dois novos tipos de elementos no componente de execução: o WS HUB e as Máquinas de Testes.

O WS HUB (Web Services Hub) funciona como centro de gestão de execuções remotas. *Testers* poderão desenvolver os cenários de teste, experimentá-los nas suas máquinas e assim que estiverem prontos, gerar uma bateria de testes e enviá-la para a WS HUB. No WS HUB estarão alojados todas as baterias de testes geradas, relatórios de execução de testes e informação sobre as máquinas de testes presentes no sistema.

A comunicação decorreria através de uma camada de *Web Services*, que disponibilizaria funcionalidades para as duas outras entidades do sistema: as máquinas dos *testers* e as máquinas executoras de testes.

Entre as diversas funcionalidades, destacam-se as que permitiriam a marcação de execução de testes (tendo como variáveis as plataformas onde decorreriam, possível agendamento, qual a bateria de testes a usar e a localização da máquina onde deveriam ocorrer), consulta das máquinas disponíveis no sistema e respectiva informação, consulta de relatórios de execução de testes realizados, assim como a consulta do estado das máquinas de teste.

Na perspectiva das máquinas de teste, estariam disponíveis funcionalidades que permitiriam o registo de uma nova máquina de testes no sistema (com toda a informação relevante à sua caracterização na plataforma), a consulta de pedidos de execução a realizar e o envio de relatórios de resultados de execução de testes.

5.2 Dados de Teste

Outro aspecto que poderia aumentar a cobertura dos testes no sistema, seria a possibilidade de usar dados nos cenários de teste de um modo dinâmico. A proposta que se apresenta de seguida propõe a adopção de uma nomenclatura de referenciação de dados. A referenciação apontará para uma estrutura similar a uma *HashTable*, onde actuaria como *key*. A cada *key* estaria associada uma lista de valores de dados de teste.

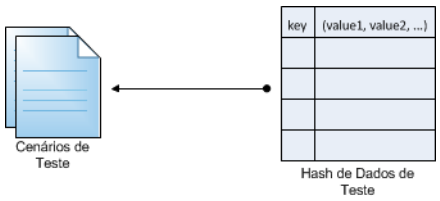

Figura 5.2: Diagrama de Arquitectura do Sistema Distribuído de Execução de Testes

A nomenclatura seguiria um padrão, em que o próprio nome indicava a validade dos dados que referenciaria. Esta estratégia cobriria também o problema de ter que testar a Aplicação Web em diferentes línguas. A título de exemplo, a referenciação do nome de um cartão seria feita usando os nomes: cartao.nome.valido e cartao.nome.invalido. O gerador de código, consoante as referenciações, processaria diferentes usos das asserções (p.ex. *AssertEqual*/*AssertNotEquals*).

A inclusão desta estratégia obrigaria também a modificações na geração dos relatórios de execuções de teste, de modo a apresentar os dados que foram usados nas execuções dos cenários de teste.

5.3 Criador de *locators*

A processo de criação de *locators* pode revelar-se bastante moroso, especialmente quando se deseja que a expressão seja ao mesmo tempo flexível e exacta.

A ideia de uma ferramenta que assistisse o utilizador na criação das expressões poderia eliminar este problema. Actualmente existem diversas ferramentas que contemplam este tipo de funcionalidade, que funcionam como extensões para *browsers*. No entanto, não o realizam do modo desejado e de acordo com as especificidades da actual plataforma de automatização de testes.

A solução passaria pela adopção e adaptação de uma destas ferramentas. Na adaptação seriam implementadas estratégias de cálculo de expressões xpath relativas, expressões de identificação de elementos apenas por id, texto de uma hiperligação, entre outras. Das expressões sugeridas pela ferramenta, o utilizador escolheria a que mais se adequaria, aliviando-o do esforço da construção da mesma.

Uma investigação preliminar permitiu que fossem seleccionadas algumas das ferramentas susceptíveis de serem escolhidas: FirePath[1] para o Firebug[2] do Firefox, XPather para o Firefox[3], XPath Checker[4] para o Firefox e a Dom Inspector[5] também para o Firefox.

5.4 Execução de Testes com recolha de *logs* da Aplicação Web

Testes de Caixa Cinzenta são testes que combinam técnicas de Caixa Branca e Caixa Preta. São testes que são executados na perspectiva do utilizador à semelhança de Testes de Sistema, mas com a diferença de ocorrer recolha de informação interna do sistema (antes, durante e depois da realização dos testes) [Pat05]. Todavia, não contempla operações de alteração de dados internos do sistema, na medida em que o utilizador não possui uma interface que lhe permita executar este tipo de operações.

A utilização deste tipo de técnicas de teste facilitaria o processo de resolução de *bugs*, por poderem fornecer mais e melhor informação sobre o problema.

Uma das maiores fontes de informação sobre o funcionamento de um sistema encontra-se nos *logs* criados pelo próprio sistema. O módulo de execução da plataforma de testes poderia então, valendo-se desta informação, enriquecer a informação apresentada nos relatórios de execução de teste.

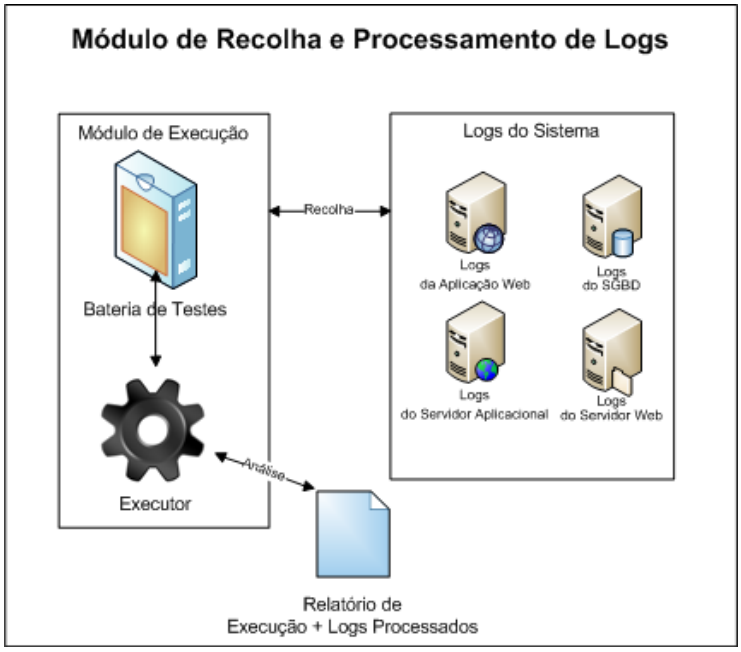

Figura 5.3: Processo de recolha de *Logs* do Sistema em Teste

[1] http://code.google.com/p/firepath/
[2] http://getfirebug.com/
[3] http://xpath.alephzarro.com/
[4] http://code.google.com/p/xpathchecker/
[5] https://developer.mozilla.org/En/DOM_Inspector

Ao actual módulo de execução de testes seria dado acesso aos diversos *logs* produzidos pelo sistema que alimenta a Aplicação Web, bem como aos *logs* produzidos pela própria Aplicação Web. Esta operação poderia ser conseguida recorrendo ao uso de uma ferramenta, como é o caso do rsync[6], que sincronizaria os diversos ficheiros de *log* com uma cópia dos mesmos, a que o módulo de execução teria acesso de leitura.

Após a referida recolha, o módulo de execução de testes poderia filtrar a informação nos *logs* recorrendo a heurísticas temporais, restringindo assim a informação a anexar aos relatórios de execução de testes.

[6] http://samba.anu.edu.au/rsync/

Capítulo 6

Conclusões

Uma plataforma é uma ferramenta ou um conjunto de ferramentas que se encontram ao dispor de uma equipa, acompanhando-a no processo de desenvolvimento adoptado. O principal objectivo é o melhoramento de desempenho, sem alterar nem redefinir o processo de desenvolvimento.

A realização de testes é um dos processos mais importantes no desenvolvimento de *software*. Em processos iterativos, caracterizados por um desenvolvimento em pequenos ciclos de implementação, o teste adquire maior presença ao longo do processo, devido à necessidade deste acompanhar o seu desenvolvimento, devendo ser também realizado iterativamente. Ou seja, à medida que novas implementações vão sendo realizadas e antigas actualizadas, irão sendo criados, actualizados e executados os respectivos testes. Esta abordagem introduz no processo de desenvolvimento um factor de repetição, que se não for abordado de um modo adequado, poderá prejudicar os processos globais de produção. A realização manual (concepção e execução) de testes no seio de pequenas equipas sob um desenvolvimento iterativo, à medida que o projecto cresce, vai tomando contornos de impraticabilidade devido ao crescente esforço associado. Por conseguinte, nasce a necessidade de agilização destes processos, com ferramentas que de alguma forma automatizem os processos, aliviando a equipa de desenvolvimento do referido esforço.

O estudo, análise e trabalho realizados surgem no seio de um empresa revestida com estas características. A Cardmobili fornece um serviço que se encontra disponível através de uma aplicação Web multi-*browser* e uma aplicação móvel multi-plataforma (iPhone, Windows Mobile, Blackberry RIM, Android, MIDP2.0/Java). O principal objectivo do trabalho realizado, que se debruça na automatização de testes, consiste na criação de uma plataforma que deverá auxiliar na optimização dos processos de criação e execução de testes sobre a Aplicação Web do serviço disponibilizado pela empresa.

O estudo e análise extensivas possibilitaram a construção de uma base de conhecimento, que auxiliou a concepção da plataforma. A investigação debruçou-se em primeiro lugar nas vantagens/potencialidades e desvantagens/obstáculos na adopção de processos de automatização de testes. Seguidamente foram estudados problemas habituais, bem como investigações anteriores e recomendações de boas práticas na área. Por fim, foi também realizado um estudo na área de testes funcionais a Aplicações Web. Foram estudadas arquitecturas de teste, técnicas e ferramentas existentes.

A construção de uma plataforma de automatização de testes é um projecto de *software* por si só, logo o desenvolvimento, o uso, manutenção e actualização são revestidos dos mesmos problemas de um usual projecto de *software*. Os requisitos, arquitecturas e decisões foram feitos tendo em atenção os estudos e investigação realizada na área.

Tomando por base o conceito de FTDE (Functional Test Development Environment), foi implementado um *plugin* para o Eclipse, com o objectivo de fornecer um ambiente integrado de desenvolvimento de testes. Quando usado por equipas que utilizem o Eclipse como ambiente de desenvolvimento para o próprio projecto, integra-se com esse mesmo ambiente, apresentando-se como um único ambiente. A plataforma de testes permite automatizar a realização de testes, desde a concepção até à execução e análise de resultados. Durante a implementação foram utilizadas práticas de automatização estudadas na altura da investigação, integrando-as também na própria plataforma, conduzindo assim um utilizador da plataforma, no cumprimento e utilização destas no desenvolvimento de testes. O cumprimento destas práticas faz com que os testes sejam mais claros, mais robustos e mais fáceis de manter.

A plataforma é constituída por quatro componentes principais: Editor de Especificação de Cenários de Teste Funcionais a Aplicações Web, Editor de *Suites* de Execução de Teste, Módulo de Geração de Código e Módulo de Execução de Testes.

O primeiro editor permite que sejam desenvolvidas especificações abstractas de cenários de teste funcionais a Aplicações Web. A especificação assenta na utilização do padrão BDD de apresentação de cenários, na utilização de um domínio de acções Web criado para o efeito e pela utilização de uma estruturação de representação de elementos de páginas Web, permitindo assim que elementos não técnicos, envolvidos no desenvolvimento de *software*, possam também participar na realização de testes.

Da análise de ferramentas de testes funcionais, foi escolhida uma que permitisse o controlo remoto de um *browser* de acordo com as necessidades do projecto. A partir das especificações de cenários de testes funcionais poderão ser geradas as respectivas implementações. Foi criado um módulo de geração de código abstracto, posteriormente usado para a implementação de um gerador de código que recorre à integração programática da ferramenta seleccionada (a Selenium2). Esta geração de código cria um ambiente de execução de testes, que aliado com a *framework* TestNG, permite a geração de baterias

de testes independentes do *plugin* em si, podendo ser posteriormente usadas na execução dos cenários de teste em diversas plataformas (Windows, Mac OS X e Linux).

Podem ainda ser especificadas *suites* de execução de testes, que são construídas recorrendo à inclusão e/ou exclusão explícita de cenários de testes ou implícita, recorrendo ao conceito de *tags*/grupos de cenários de teste (que definem conjuntos de cenários de teste), criando assim a possibilidade de execuções de testes especificas. Valendo-se de uma *suite* de execução de testes, o módulo de execução de testes, irá detectar os *browsers* instalados na máquina onde se realiza a execução e então, usando uma bateria de testes, efectuará os diversos cenários de testes funcionais nos respectivos *browsers*.

A execução de testes irá dar origem à geração dos respectivos relatórios. Cada relatório contém a informação relevante sobre os cenários de teste realizados e complementarmente um conjunto sequencial de *screenshots*, representativos da execução do cenário a que se encontram associados.

A plataforma encontra-se actualmente preparada para realização de cenários de teste em plataformas móveis, estando apenas dependente do lançamento da versão funcional dos *drivers* de controlo da Selenium2 para o iPhone e Android.

Devido à forte componente de extensibilidade e adaptabilidade da plataforma, foram igualmente realizados diversos esboços para futuros desenvolvimentos que a plataforma poderá sofrer, com o intuito de a melhorar e tornar mais completa. A possibilidade de executar cenários de teste em diversas plataformas em máquinas presentes em diversos locais do globo, poderá ser efectuada com o alargamento do Módulo de Execução da plataforma para um Sistema Distribuído de Execução de Testes. Foram também construídos modelos conceptuais da implementação de módulos de dados de teste e de assistência na criação de *locators* de elementos de páginas Web. Por fim, foi também pensada a adopção do conceito testes de caixa cinzenta, com a construção preliminar da conceptualização de recolha de dados de *logs* da Aplicação Web em teste.

Os objectivos e resultados esperados com esta dissertação foram rigorosamente cumpridos, o que se traduz num aumento substancial da capacidade da realização de testes ao seu serviço web disponibilizado pela empresa .

As investigações realizadas nas áreas da automatização de testes e de testes funcionais a aplicações web, permitiu reavaliar e melhorar o desempenho nas fases de realização de testes, dentro dos processos de desenvolvimento de *software* da empresa.

A solução construída prova as potencialidades na adopção do conceito de FTDE (Functional Test Development Environment). O conceito foi extendido ao ponto em que com a adopção do padrão BDD, os elementos não técnicos possam também realizar testes a Aplicações Web. A plataforma permite num único ambiente (que pode ser interligado com o ambiente de desenvolvimento), efectivar todo o processo de testes a Aplicações Web. Possuindo camadas que permitem simplificar as diversas operações que constituem

o processo de testes, sendo possível conceber, manter, executar e analisar os resultados das respectivas execuções, dentro de um mesmo ambiente integrado que a plataforma suporta.

Referências

[act10] actiWATE. actiwate - architecture overview, Janeiro 2010. http://www.actiwate.com/architecture_overview.html.

[Ada08] Christian Adam. Windmill Slide Presentation 2008, 2008. http://www.getwindmill.com/features/presentations.

[Ada10] Christian Adam. Windmill 0.2 Released, 2010. http://blog.chandlerproject.org/2007/10/18/windmill-02-released/.

[And07] Jennitta Andrea. Envisioning the Next Generation of Functional Testing Tools. *Ieee Software*, 2007.

[Apo10] Apodora. Apodora | an opensource functional test automation solution, Janeiro 2010. http://www.apodora.org/about.php.

[Art09] Inc. ArtOfTest. *AUTOMATION DESIGN CANVAS 2.0 USER GUIDE*. ArtOfTest, 2009.

[Bac99] James Bach. Test Automation Snake Oil. *Bach*, pages 1–6, 1999.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, first edition, 1999.

[Bre01] Pettichord Bret. Seven Steps to Test Automation Sucess. *Star West*, 2001.

[Bru09] Wichmann Dennis Bruns Andreas, Kornstädt Andreas. Web Application Tests with Selenium. *IEEE*, pages 88–91, 2009.

[BS07] Cédric Beust e Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[BT03] Boehm e Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[BWK05] S. Berner, R. Weber e R.K. Keller. Observations and lessons learned from automated testing. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 571–579, 2005.

[CG09] Lisa Crispin e Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009.

[Cha09] Taolun Chai. Automated Universal Testing and Tutoring System for Web Application. *Architecture*, 2009.

[Dan10a] North Dan. Introducing BDD, Junho 2010. http://blog.dannorth.net/introducing-bdd/.

[Dan10b] North Dan. What's in a Story?, Junho 2010. http://blog.dannorth.net/whats-in-a-story/.

[Fit10] Fitnesse. Fitnesse - frontpage, Janeiro 2010. http://www.fitnesse.org/.

[Fro09] Froglogic. *Squish® for Web - Data Sheet*. froglogic, Alemanha, 2 edition, 2009.

[GM01] A. Ginige e S. Murugesan. Web engineering: an introduction. *IEEE Multimedia*, 8(1):14–18, 2001.

[HK06] A. Holmes e M. Kellogg. *Automating Functional Tests Using Selenium*. IEEE, 2006.

[IEE90] IEEE, editor. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, New York, NY, USA, 1990.

[iMa10a] iMacros. Directscreen wiki, Janeiro 2010. http://wiki.imacros.net/DS.

[iMa10b] iMacros. imacros wiki, Janeiro 2010. http://wiki.imacros.net/.

[KaJ06] T. Karamat e a.N. Jamil. Reducing Test Cost and Improving Documentation In TDD (Test Driven Development). *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)*, pages 73–76, 2006.

[KNR09] Eun Ha Kim, Jong Chae Na e Seok Moon Ryoo. Test Automation Framework for Implementing Continuous Integration. *2009 Sixth International Conference on Information Technology: New Generations*, pages 784–789, Abril 2009.

[LF06] Giuseppe A. Di Lucca e Anna Rita Fasolino. Testing WebBased applications:The state of the art and future trends. *Information and Software Technology*, 48(12), 2006.

[LISA09] Lucas Lima, Juliano Iyoda, Augusto Sampaio e Eduardo Aranha. Test case prioritization based on data reuse an experimental study. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 279–290, Outubro 2009.

[Max10] MaxQ. Recording and Running, 2010. http://maxq.tigris.org/docs/record.html.

[Mes03] Gerard Meszaros. Agile regression testing using record & playback. *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03*, page 353, 2003.

[MRT08] Alessandro Marchetto, Filippo Ricca e Paolo Tonella. A case study-based comparison of web testing techniques applied to AJAX web applications. *International Journal on Software Tools for Technology Transfer*, 10(6):477–492, Dezembro 2008.

[MSA] Gerard Meszaros, Shaun M Smith e Jennitta Andrea. The Test Automation Manifesto. pages 73–81.

[Pat05] Ron Patton. *Software Testing (2nd Edition)*. Sams, 2nd edition, 2005.

[Pre00] R.S. Pressman. What a tangled Web we weave [Web engineering]. *IEEE Software*, 17(1):18–21, 2000.

[Pus10a] PushToTest. Building tests of rich internet applications using ajax and selenium, Janeiro 2010. http://www.pushtotest.com/docs/testmaker-help/testmaker-tutorials/building-tests-of-rich-internet-applications-using-ajax-and-selenium/tutorial-all-pages.

[Pus10b] PushToTest. The value of pushtotest open-source and commercial, Janeiro 2010. http://www.pushtotest.com/products/comparison.

[RW06] Rudolf Ramler e Klaus Wolfmaier. Economic perspectives in test automation. *Proceedings of the 2006 international workshop on Automation of software test - AST '06*, page 85, 2006.

[RWW+] Rudolf Ramler, Edgar Weippl, Mario Winterer, Wieland Schwinger e Josef Altmann. A Quality-Driven Approach to Web Testing.

[Sah10a] Sahi. Running multiple tests in batch mode · , 2010. http://sahi.co.in/w/Running+multiple+tests+in+batch+mode.

[Sah10b] Sahi. Sahi Scripting Basics - Part 1 ·, 2010. http://sahi.co.in/w/sahi-scripting-basics.

[Sai10] Venkat Sai. WATIR - Automated testing that doesn't hurt', 2010. http://www.slideshare.net/sai_venkat/watir.

[Sel10] Selenium. Selenium-rc — selenium v1.0 documentation, Janeiro 2010. http://svn.openqa.org/fisheye/browse/~{}raw,r=2395/selenium/selenium-website/src/main/rst/.build/html/05_selenium_rc.html.

[Sim07] Stewart Simon. GTAC 2007: Web Driver, 2007. http://www.youtube.com/watch?v=tGu1ud7hk5I&feature=player_embedded.

[SJ04] Yanhong Sun e Edward L. Jones. Specification-driven automated testing of GUI-based Java programs. *Proceedings of the 42nd annual Southeast regional conference on - ACM-SE 42*, page 140, 2004.

[SPX08] Bernard Stepien, Liam Peyton e Pulei Xiong. Framework testing of web applications using TTCN-3. *International Journal on Software Tools for Technology Transfer*, 10(4):371–381, Agosto 2008.

[SQF08] Mate' Sztipanovits, Kai Qian e Xiang Fu. The automated web application testing (AWAT) system. *ACM Southeast Regional Conference*, 2008.

[Ste09] Huggins Jason Stewart Simon. GTAC 2009 - Selenium: to 2.0 and Beyond!, 2009. http://www.youtube.com/watch?v=RQD4EzWI4qk.

[SWA10] SWAT. ulti-swat - home, Janeiro 2010. http://ulti-swat.wikispaces.com/.

[SZ] State Street e Technology Zhejiang. A Test Automation Solution on GUI Functional Test 1.

[T06] Harri T. Data-Driven and Keyword-Driven Test Automation Frameworks. 2006.

[WA06] Tom Wissink e Carlos Amaro. Successful Test Automation for Software Maintenance. *2006 22nd IEEE International Conference on Software Maintenance*, pages 265–266, Setembro 2006.

[Wat10] Watir. Platforms ■ Watir, 2010. `http://watir.com/platforms/`.

[Win10] Windmill. Windmill Documentation, 2010. `http://trac.getwindmill.com/`.

[WX09] Xinchun Wang e Peijie Xu. Build an Auto Testing Framework Based on Selenium and FitNesse. *2009 International Conference on Information Technology and Computer Science*, Julho 2009.

Anexo A

Selenium: Implementação de Caso de Teste

```
1  public class SeleniumTest {
2   private Selenium selenium;
3   @Test
4   public void addCard() {
5          this.selenium = (Selenium) new DefaultSelenium("localhost", 4444, "firefox", "http://cardmobili.com/");
6          this.selenium.start();
7          this.selenium.open("/home.xhtml");
8           try {
9                   selenium.type("loginForm:loginUsernameInput", "email");
10                  selenium.type("loginForm:loginPasswordInput", "password");
11                  selenium.click("//form[@id='loginForm']/div[1]/div[3]/p/a/span[2]");
12                  selenium.waitForPageToLoad("30000");
13
14                  selenium.click("link=All cards");
15                  selenium.waitForPageToLoad("40000");
16
17                  selenium.type("cardManipulationForm:searchString", "myzoncard");
18                  selenium.keyUp("cardManipulationForm:searchString", "d");
19
20                  selenium.waitForCondition("var value = selenium.isElementPresent('//*[@class='cardGallery']//p[contains(text(),'myZONcard')]')
21                                                  and contains(@style, 'visibility: visible')]'); value == true", "80000");
22
23                  selenium.click("css=a.cardItemRegister");
24                  selenium.waitForCondition("var value = selenium.isElementPresent('//*[@class='modalContent']//h3[contains(text(),'myZONcard')]')
25                                                  and contains(@style, 'visibility: visible')]'); value == true", "80000");
26
27                  String[] allFields = selenium.getAllFields();
28                  ArrayList<String> textFields = new ArrayList();
29                  for(String field : allFields) {
30                          if(field.matches("^cardRegisterForm.*")) {
31                                  textFields.add(field);
32                          }
33                  }
34                  selenium.type(textFields.get(0), "Hugo Gomes");
35                  selenium.type(textFields.get(1), "NUM CARTAO");
36                  selenium.click("//*[@id='cardRegisterForm']//*[text()='save']");
37
38                  selenium.waitForCondition("var value = selenium.isElementPresent('//*[contains(text(), 'Card successfully added to your account.'
39                                                  and contains(@style, 'visibility: visible')]'); value == true", "80000");
40                  selenium.isTextPresent("Card successfully added to your account.");
41
42            } catch (SeleniumException e) {
43             fail(e.getMessage());
44            } catch (InterruptedException e) {
45                 e.printStackTrace();
46          }
47    }
48  }
```

Anexo B

Selenium 2: Implementação de Caso de Teste

```
1  public class Login_na_Cardmobili extends WebDriverTestBase {
2          @Test
3           public void Login_na_Cardmobili_test(ITestContext context) {
4                  driver.navigate().to("http://cardmobili.com");
5                  driver.findElement(By.xpath("//*[@class='email']//input")).sendKeys("email");
6                  driver.findElement(By.xpath("//*[@class='passwd']//input")).sendKeys("password");
7                  driver.findElement(By.partialLinkText("ligar-me")).click();
8
9                  new WebDriverWait(driver, 8).until(new VisibilityOfElementLocated(By.partialLinkText("All Cards")));
10                 driver.findElement(By.partialLinkText("All Cards")).click();
11                 driver.findElement(By.xpath("//*[@class='searchOptions']//input")).sendKeys("myzoncard");
12
13                 new WebDriverWait(driver, 8).until(new VisibilityOfElementLocated(By.xpath("//*[@class='cardGallery']//p[text()='myZONcard']")));
14                 driver.findElement(By.xpath("//*[@class='cardGallery']//p[text()='myZONcard']")).click();
15
16                 ((RenderedWebElement) driver.findElement(By.xpath("//*[@class='cardThumb']//img"))).hover();
17                 driver.findElement(By.xpath("//*[@class='cardItemRegister']")).click();
18
19                 new WebDriverWait(driver, 8).until(new VisibilityOfElementLocated(By.xpath("//*[@id='cardRegisterForm']//div[2]//span//input")));
20                 driver.findElement(By.xpath("//*[@id='cardRegisterForm']//div[2]//span//input")).sendKeys("Hugo Gomes");
21                 driver.findElement(By.xpath("//*[@id='cardRegisterForm']//div[2]//span//input")).sendKeys("NUM CARTAO");
22                 driver.findElement(By.linkText("guardar")).click();
23
24                 new WebDriverWait(driver, 8).until(new VisibilityOfElementLocated(By.xpath("//*[contains(text(), " +
25                                                                                             "'Card successfully added to your account.')]")));
26                 Assert.assertEquals("Card successfully added to your account.", driver.findElement(By.xpath("//*[contains(text(), " +
27                                     "'Card successfully added to your account.')]")).getText();
28                 return;
29         }
30 }
```

Anexo C

Watir: Implementação de Caso de Teste

```
1  class TestExample < Test::Unit::TestCase
2    def test_add_card
3      browser = Watir::Browser.new
4      browser.goto("http://www.cardmobili.com")
5
6      browser.text_field(:id, 'loginForm:loginUsernameInput').value = "email"
7      browser.text_field(:id, 'loginForm:loginPasswordInput').value = "password"
8      browser.link(:title, "Login").click
9
10     wait_until {browser.text.include? "All cards"}
11
12     browser.link(:text, "All cards").click
13     browser.text_field(:id, 'cardManipulationForm:searchString').value = "myzoncard"
14     browser.text_field(:id, 'cardManipulationForm:searchString').fire_event('onkeyup')
15
16     wait_until {browser.text.include? "myZONcard"}
17     browser.link(:class, 'cardItemRegister').click
18
19     wait_until {browser.text.include? " Add card"}
20     browser.text_field(:xpath, "//*[@id='cardRegisterForm']").text_fields[0].value ="NUM CARTAO"
21     browser.text_field(:xpath, "//*[@id='cardRegisterForm']").text_fields[1].value ="Hugo Gomes"
22     browser.links.each do |l|
23       l.click if l.text =~/^save.*/i
24     end
25
26     wait_until {browser.text.include? "Card successfully added to your account"}
27     assert(browser.text.include? "Card successfully added to your account.")
28   end
29 end
```

Anexo D

Windmill: Implementação de Caso de Teste

```
def test_addcard_winmill():
  client = WindmillTestClient(__name__)
    client.type(text=u'email', id=u'loginForm:loginUsernameInput')
    client.type(text=u'password', id=u'loginForm:loginPasswordInput')
    client.click(xpath=u"//form[@id='loginForm']/div[1]/div[3]/p/a/span[2]")

    client.waits.forPageLoad(timeout=u'20000')
    client.waits.forElement(jquery=u'("a:contains(\'All cards\')")[0]', timeout=u'8000')
    client.click(jquery=u'("a:contains(\'All cards\')")[0]')

    client.waits.forPageLoad(timeout=u'20000')
    client.type(text=u'myzoncard', id=u'cardManipulationForm:searchString')
    client.waits.forElement(jquery=u'("#cardGallery p:contain(\'myZONcard\')")[0]', timeout=u'4000')

    client.click(jquery=u'("a.cardItemRegister")[0]')
    client.waits.forElement(jquery=u'("#cardRegisterForm a:contain(\'save\')")[0]', timeout=u'8000')
    client.type(jquery=u"('#cardRegisterForm :input:not(:hidden)')[0]",  text=u'Hugo Andr\xe9 Gomes')
    client.type(jquery=u"('#cardRegisterForm :input:not(:hidden)')[1]",  text=u'NUM CARTAO')
    client.click(jquery=u'("#cardRegisterForm a:contain(\'save\')")[0]')

    client.waits.sleep(milliseconds=u'8000')
    client.asserts.assertTextIn(jquery=u"('.globalMessagesSuccess h4')",
    validator=u"Card successfully added to your account. \\
    Keep registering cards or check your cards in your 'My cards' tab.")
```

Anexo E

Sahi: Implementação de Caso de Teste

```
_setValue(_textbox("loginForm:loginUsernameInput"), "email");
_setValue(_password("loginForm:loginPasswordInput"), "password");
_click(_span("Login"));
_click(_link("All cards"));
_setValue(_textbox("cardManipulationForm:searchString"), "myzoncard");
_wait(4000);
_click(_byClassName("cardItemRegister", "a"));
_wait(4000);
_setValue(jQuery.noConflict()('#cardRegisterForm :input:not(:hidden)')[0], "Hugo
_setValue(jQuery.noConflict()('#cardRegisterForm :input:not(:hidden)')[1], "NUM
_click(_link(/^cardRegisterForm.*/));
_assertEqual(jQuery.noConflict()('.globalMessagesSuccess h4').text(),
"Card successfully added to your account.Keep registering cards or
check your cards in your 'My cards' tab.", "Mensagem de Cartão
Criado com Sucesso");
_click(_link("My cards"));
_click(_link('myZONcard'));
<browser>
function getValues(i){
var values = [];
jQuery.noConflict()('#cardRegisterForm :input:not(:hidden)').each(function(j, it
values[j] = jQuery.noConflict()(item).val();
});
return values[i];
}
</browser>
_assertEqual(getValues(0), "Hugo André Gomes", "Nome no Cartão");
_assertEqual(getValues(1), "NUM CARTAO", "Número no Cartão");
```

Anexo F

Exemplo de *locators.xml*

```
<existingLocators>
   <updated>2010-05-21_12_01_54_779</updated>
   <locators>
      <locator>
         <name>search text google</name>
         <type>NAME</type>
         <expression>q</expression>
      </locator>
      <locator>
         <name>search button google</name>
         <type>NAME</type>
         <expression>btnG</expression>
      </locator>
      <locator>
         <name>lista resultados google</name>
         <type>ID</type>
         <expression>res</expression>
      </locator>
      <locator>
         <name>cardmobili google result</name>
         <type>XPATH</type>
         <expression>//a[@href=&apos;http://www.cardmobili.com/&apos;]</expression>
      </locator>
   </locators>
</existingLocators>
```

Figura F.1: Exemplo de locators.xml

Exemplo de *locators.xml*

Anexo G

Exemplo de *tags.xml*

```
<existingTags>
   <updated>2010-05-21 12:07:49:597</updated>
   <tags>
      <tag>cardmobili</tag>
      <tag>pesquisa</tag>
      <tag>resultados</tag>
      <tag>cartao</tag>
      <tag>cliente</tag>
      <tag>edicao</tag>
      <tag>1 release</tag>
      <tag>2 release</tag>
      <tag>3 release</tag>
      <tag>4 release</tag>
   </tags>
</existingTags>
```

Figura G.1: Exemplo de tags.xml

Exemplo de *tags.xml*

Anexo H

Screenshot de *combobox* de selecção de acções de domínio Web

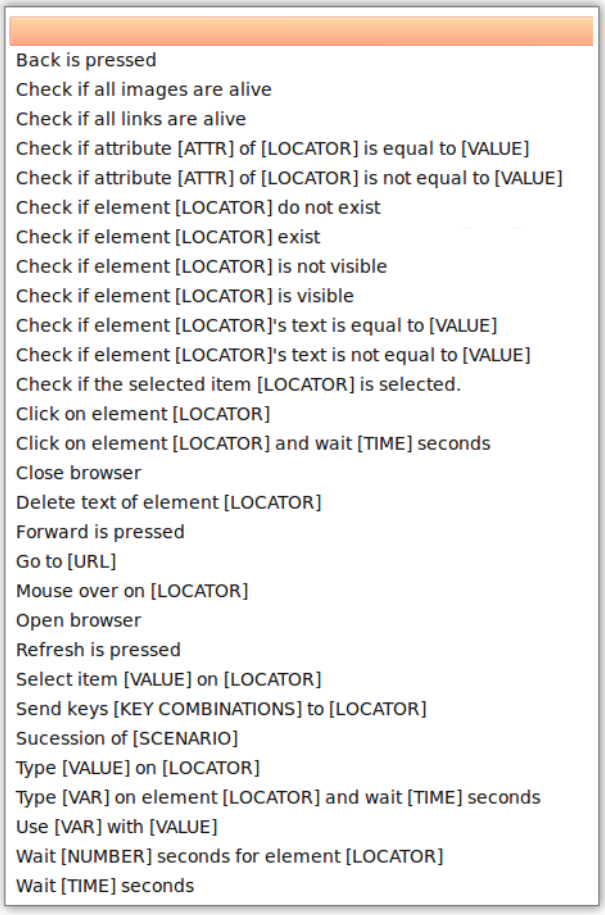

Figura H.1: *Screenshot* de *combobox* de selecção de acções de domínio Web