CENTRO DE ENSINO UNIFICADO DE TERESINA – CEUT

CURSO DE CIÊNCIA DA COMPUTAÇÃO

ERNESTO CID BRASIL DE MATOS

UMA FERRAMENTA PARA GERAÇÃO AUTOMÁTICA DE TESTES FUNCIONAIS E PROTÓTIPOS DE INTERFACE, A PARTIR DE CASOS DE USO

TERESINA

2009

ERNESTO CID BRASIL DE MATOS

UMA FERRAMENTA PARA GERAÇÃO AUTOMÁTICA DE TESTES FUNCIONAIS E PROTÓTIPOS DE INTERFACE, A PARTIR DE CASOS DE USO

Monografia apresentada ao Centro de Ensino Unificado de Teresina – CEUT como um dos pré-requisitos para obtenção do grau de bacharel em Ciência da Computação.

Orientador: Thiago Carvalho de Sousa

TERESINA

2009

ERNESTO CID BRASIL DE MATOS

UMA FERRAMENTA PARA GERAÇÃO AUTOMÁTICA DE TESTES FUNCIONAIS E PROTÓTIPOS DE INTERFACE, A PARTIR DE CASOS DE USO

Monografia apresentada como exigência parcial para a obtenção do grau de Bacharel em Ciência da Computação, na área de concentração Engenharia de *Software*, à banca examinadora do Curso de Ciência da Computação, do Centro de Ensino Unificado de Teresina – CEUT

Aprovada em 03/12/2009

BANCA EXAMINADORA

Thiago Carvalho de Sousa
Universidade de São Paulo

Ricardo de Andrade Lira Rabêlo
Centro de Ensino Unificado de Teresina

Guilherme Amaral Avelino
Centro de Ensino Unificado de Teresina

A todos que, direta ou indiretamente,
ajudaram durante a trajetória até aqui.

Agradecimentos

A minha família, por todo o carinho e suporte durante esta jornada.

A todos os colegas, que durante o curso, acabaram tornando-se família.

Ao CEUT e todos os professores, pelo apoio durante a vida acadêmica.

A Remanso, por patrocinar parte deste projeto.

Ao Mestre Thiago, por estar sempre à disposição para orientação e conselhos (até mesmo aos domingos).

"A imaginação é mais importante que a ciência,
porque a ciência é limitada, ao passo que
a imaginação abrange o mundo inteiro."
– Albert Einstein

Lista de Figuras

Lista de Tabelas

Sumário

Resumo

Neste trabalho, temos como objetivo iniciar o desenvolvimento de uma plataforma de especificação de requisitos, que permita melhor rastreamento dos mesmos e seja capaz de gerar código fonte, a partir de documentos de especificação. Para isso, definimos uma linguagem natural controlada, que deve ser utilizada na definição de cenários de casos de uso, descrições de interface e glossários da aplicação. Essa linguagem foi especificada a partir de estudos de padrões de documentos de especificações de requisitos, seguindo, assim, o modelo adotado pela indústria de *software* atualmente. Após definida a linguagem, construímos um interpretador para ela, capaz de receber estas especificações e extrair informações destes documentos, para gerar código para um *framework* de testes funcionais, para aplicações *Web* e páginas HTML (*HyperText Markup Language*), representando as descrições de interface.

Palavras-chave: Engenharia de Requisitos. Testes de *Software*. Prototipação de *Software*. Automação.

Abstract

In this project we have the objective of initiating the development of a requirement specification platform which allows better requirements management and source code generation based on those specifications. To do so, we defined a natural controlled language which must be used to specify use case scenarios, user interface specifications and glossaries to an application. This language was specified based on studies about patterns on requirement specification documents, thus maintaining the pattern adopted by the software industry nowadays. After we defined the language, we have built an interpreter for it, capable of receiving these specifications and extracting information from them to generate code for a Web application functional testing framework and HTML (HyperText Markup Language) pages representing the user interface specifications.

Keywords: *Requirement Engineering. Software Testing. Software Prototyping. Automation.*

1 Introdução

Quando falamos de Engenharia de *Software*, e mais especificamente em Engenharia de Requisitos, os casos de uso podem ser considerados um padrão no que diz respeito à especificação de requisitos de *software*. A partir deles, os requisitos funcionais da aplicação podem ser descritos em cenários que demonstram a interação do usuário com a aplicação em questão.

Para manter a qualidade do *software* desenvolvido, é necessário testá-lo intensamente, para isto, existem diversas técnicas de teste. Dentre estas técnicas, uma é diretamente relacionada a especificações como os casos de uso: a aplicação de testes funcionais. Esta técnica tem como objetivo simular a interação do usuário com a aplicação, validando, desta forma, se a aplicação atende a requisitos especificados previamente.

Outra técnica bastante conhecida em ambientes de desenvolvimento de *software* é a utilizacão de protótipos da aplicação. Protótipos são geralmente utilizados para validação do *software* por parte do cliente, pois permitem que este tenha uma visão prévia de como será o *software* após finalizado, ainda em estágios iniciais do desenvolvimento, o que pode possibilitar correções de erros de maneira rápida, sem grandes impactos, evitando prejuízos por se desenvolver algo que foge às necessidades do cliente.

Uma vez que os requisitos fornecem informações preciosas para o desenvolvimento de testes funcionais e descrições de interfaces da aplicação, seria interessante gerá-los automaticamente a partir destes requisitos. Neste trabalho apresentamos uma ferramenta que, a partir de artefatos utilizados para descrição de requisitos (casos de uso, glossário e especificações de interface) escritos em uma linguagem natural controlada, trata de convertê-los em protótipos de interface em HTML e em testes funcionais para uma ferramenta de testes para aplicações *Web*.

Além disso, acreditamos que as técnicas de elicitação de requisitos ainda são muito precárias atualmente, pois se costuma documentar estes requisitos em arquivos de texto, que são difíceis de organizar, tendem a ficar desatualizados e obsoletos. Com este trabalho, iniciamos o desenvolvimento de uma plataforma de especificação de requisitos, que permite um melhor rastreamento e organização destes.

Outros objetivos do nosso trabalho são: redução do tempo de desenvolvimento da aplicação com o uso da geração de código fonte automatizada; aumento da coesão entre os requisitos e os testes funcionais e protótipos de interface, com a geração direta destes itens, a partir dos requisitos e definição de uma linguagem natural controlada para especificação de requisitos de *software*.

O foco inicial do projeto são os casos de uso que seguem o padrão CRUD (*Create*, *Retrieve*, *Update* e *Delete*), ou seja, cenários que tratam de criar, recuperar, atualizar e remover dados armazenados da aplicação presentes em algum repositório.

O trabalho está organizado da seguinte maneira: no primeiro capítulo apresentaremos conceitos necessários para o entendimento do restante do trabalho. Discutiremos sobre Engenharia de *Software*, Engenharia de Requisitos, Testes de *Software*, *ANTLR* (*ANother Tool for Language Recogntion*), dentre outros temas. No segundo capítulo apresentaremos alguns trabalhos relacionados, fazendo comparações entre aspectos positivos e negativos destes em relação a nossa abordagem. No terceiro capítulo apresentaremos a ferramenta que desenvolvemos, discutindo a metodologia empregada e ferramentas que utilizamos para desenvolvê-la. Discutiremos também sobre a arquitetura da ferramenta e mostraremos um exemplo de seu funcionamento. No quarto e último capítulo faremos as considerações finais, discutindo os resultados alcançados e sugerindo possíveis trabalhos futuros.

2 Contextualização e Conceitos Necessários

2.1 Considerações Iniciais

Neste capítulo apresentaremos conceitos necessários para o entendimento do restante do projeto. Discutiremos sobre Engenharia de *Software* e alguns de seus problemas, Engenharia de Requisitos, Casos de Uso, Testes de *Software*, dentre outros temas.

2.2 O que é Engenharia de *Software*?

Engenharia é a prática de empregar conhecimentos científicos ou empirícos à produção de algo. Desde os tempos remotos, o homem vem construindo edifícios, castelos, pontes etc. Inicialmente eles levavam anos para construí-los, geralmente gastavam mais material do que o necessário e cometiam vários erros. Com o passar do tempo e com a experiência adquirida, eles passaram a desenvolver construções mais estáveis, sólidas, com menos materiais e maximizando a utilização de mão de obra. Tudo isso foi conseguido após várias tentativas e erros, e através de estudo das técnicas empregadas, a fim de melhorar o processo de construção.

Softwares são aplicações programadas em uma máquina, um computador por exemplo, para desempenhar alguma tarefa, como resolver uma equação matemática, calcular coordenadas para um foguete ou até mesmo detectar possíveis pontos de inicidência de tumores cancerígenos em uma radiografia. Como podemos ver, as aplicabilidades de um *software* são várias, quase limitadas à criatividade humana, e costumam agregar enorme valor às mais diversas atividades. Eles estão sempre presentes no nosso cotidiano, em nossos inseparáveis *laptops* ou celulares e até mesmo discretamente instalados em aparelhos domésticos como microondas ou geladeiras.

Seguindo esta linha de pensamento, a Engenharia de *Software* é uma especialidade ou área da Ciência da Computação que estuda boas práticas, normas e diretrizes para a construção de *software*, de modo a desenvolvê-lo de maneira correta, dentro do prazo e do custo esperado, com máxima qualidade. A disciplina defende vários princípios que podem ser a chave para o sucesso de um projeto, ou fracasso, caso interpretadas erroneamente.

2.3 Crise do *Software*

O termo Crise do *Software* foi, primeiramente, citado em meados da década de 70, quando houve um rápido desenvolvimento dos *hardwares* em nível de complexidade. Máquinas mais poderosas podiam comportar programas mais complexos e eficientes, e para desenvolver esse tipo de programas, havia necessidade não somente de bons programadores, mas também de bons processos de desenvolvimento.

Inspirada na Engenharia Civil, surgiu o termo Engenharia de *Software*, uma ciência que deveria estudar boas práticas para desenvolver esses complexos programas da época, de forma a manter distantes problemas como: atrasos no projeto, projetos que extrapolavam o orçamento, *softwares* que não atendiam aos requisitos completamente ou que se tornavam impossíveis de serem mantidos ao longo de seu ciclo de vida.

2.4 *Chaos Report*

A indústria de *software* é conhecida por suas falhas e, muitas vezes, incapacidade de gerar lucros. Para demonstrar com estatísticas os diversos casos de sucesso ou insucesso em projetos de desenvolvimento de *software*, o *The Standish Group* emite relatórios com seus estudos sobre a quantidade de projetos que chegaram ao fim ou deixaram de ser concluídos, por algum motivo e os principais motivos que levaram esses projetos a serem cancelados. Os projetos são classificados em 3 tipos, seguindo os critérios descritos abaixo:

- *Projetos de sucesso*: o projeto foi concluído perfeitamente dentro do prazo, do orçamento e com todas as funcionalidades especificadas completas.
- *Projetos desafiados*: o projeto foi concluído e é operacional, mas transpôs o orçamento, foi entregue com atraso e as funcionalidades não estão exatamente como especificadas inicialmente.
- *Projetos arruinados*: o projeto foi cancelado.

De acordo com uma das pesquisas realizadas pelo *The Standish Group* (2004), dentre os projetos analisados: 34% obtiveram sucesso, 51% foram desafiados e 15% foram cancelados. Através desses dados, podemos notar que poucos são os projetos que conseguem ser terminados completamente, dentro de requisitos de tempo e dinheiro. Daí a

preocupação em se estudar novas práticas que possam aumentar as chances de um projeto obter o *status* de sucesso.

2.5 Métodos e Princípios

Como comentado anteriormente, a Engenharia de *Software* provê uma série de princípios e práticas que podem ajudar um projeto a alcançar sucesso. Existem várias abordagens de desenvolvimento de *software*, desde filosofias ágeis que pregam que "*software* funcionando na mão do cliente, o mais rápido possível, é o principal objetivo", a métodos mais formais em que o produto final não é tudo; é necessário fazê-lo de maneira eficaz, segura, com qualidade, documentada e obedecendo estritamente as especificações.

Nesta seção, apresentaremos um arcabouço genérico para mostrar as diversas atividades exercidas no desenvolvimento de *software*. Ele pode ser considerado genérico, pois praticamente todas as atividades que serão apresentadas, estão presentes em qualquer processo de desenvolvimento. A Figura 1 ilustra esse arcabouço.

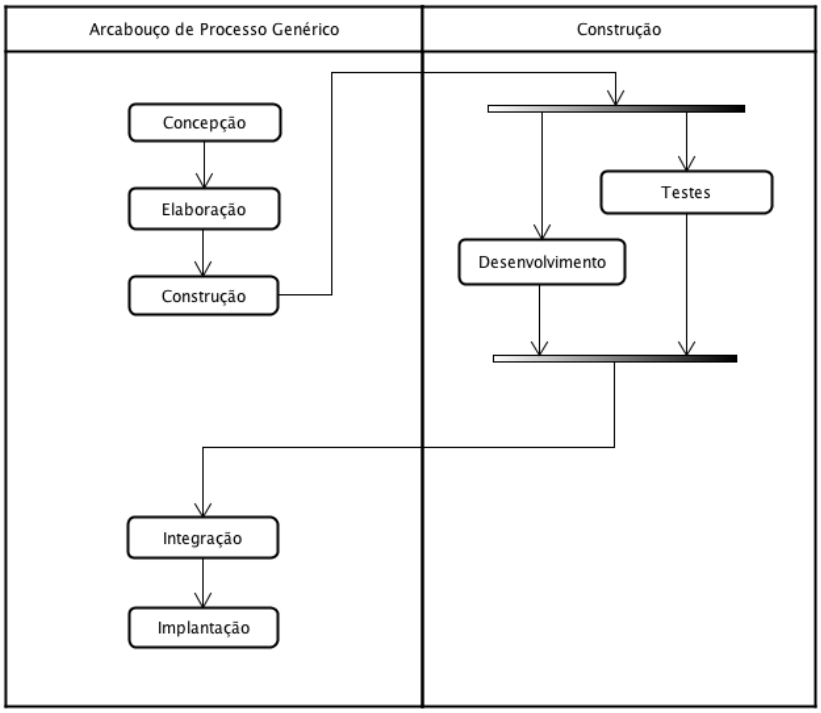

Figura 1: Arcabouço Genérico de um Processo de Desenvolvimento de *Software*.

A fase de concepção trata da especificação do produto. Nessa etapa, são verificadas as necessidades do produto. Geralmente é feito algum documento de especificação

de requisitos mais formal, muitas vezes na forma de contrato, onde serão listadas todas as funcionalidades que o *software* deverá apresentar. Além de funcionalidades, alguns requisitos não funcionais também podem ser elicitados, como carga de tráfego de dados que o sistema deve suportar, requisitos de segurança etc.

Durante a fase de elaboração, o *software* começa a tomar forma. É nessa fase que uma especificação mais precisa do *software* é feita, com diagramas da arquitetura, especificação de casos de uso, planejamento de iterações entre outras coisas. Alguns artefatos costumam ser produzidos nesta fase, como: diagramas de domínio, glossários, planos de integração etc.

Na etapa de construção, os desenvolvedores recebem os artefatos produzidos durante a etapa de elaboração, e implementam a ideia concebida. Considera-se uma boa prática a implementação de testes durante a codificação do *software*, dessa maneira, o desenvolvedor pode automatizar a execução desses testes, economizando tempo e energia com tarefas massantes como o *debugging* do código.

Após desenvolvida uma funcionalidade do *software*, ela deverá ser integrada à versão atual e estável do mesmo, de maneira que nada do que foi implementado anteriormente deixe de funcionar. Às vezes, existe também a necessidade de que o *software* se integre a outros já existentes, de maneira que possa compartilhar informações, serviços, tarefas e funções importantes com esses outros sistemas.

Quando o projeto alcançar o fim, ele deverá ser implantado no ambiente onde será utilizado. Isso pode envolver a configuração de servidores, migração de dados, instalação de *softwares* e treinamento das pessoas que utilizarão o sistema.

Como podemos ver, o processo de desenvolvimento é algo complexo e que envolve o trabalho de várias pessoas. Logo, necessita também de técnicas de gerenciamento para distribuição de tarefas, planejamento de metas e datas para cumprimento das mesmas. Essas tarefas são, geralmente, atribuídas a um Gerente de Projetos.

2.6 Engenharia de Requisitos

Para resolver um problema, não importando de que natureza, o desenvolvedor precisa conhecê-lo bem, sugerir alternativas, estar ciente de restrições. Na engenharia de *software*, o ramo que cuida dessas questões é a engenharia de requisitos.

A engenharia de requisitos trata de elicitar as necessidades do cliente, de maneira a ajudar a equipe de desenvolvimento a produzir o *software* requerido, sendo a base para qualquer projeto.

Devido a isso, Wiegers (2003) frisa que todos os *stakeholders* — os maiores interessados na questão — devem estar comprometidos em seguir um processo efetivo de coleta de requisitos. De acordo com o IEEE (1990), um requisito de *software* é:

- Uma condição ou funcionalidade necessária ao usuário para resolver um problema ou alcançar um objetivo;
- Uma condição ou funcionalidade que deve ser encontrada em um sistema, ou componente do sistema, para satisfazer um contrato, padrão, especificação ou outro documento formalmente imposto;
- Uma representação documentada de uma condição ou funcionalidade descrita nos itens anteriores.

A engenharia de requisitos pode ser considerada como uma ponte entre o cliente e a equipe de desenvolvimento, por onde serão transmitidas as informações necessárias aos desenvolvedores e por onde estes poderão tirar suas dúvidas. A pessoa responsável por controlar o fluxo de informações que passam por essa ponte é geralmente chamada de engenheiro de requisitos, também conhecido como analista de negócios ou analista de requisitos.

O engenheiro de requisitos deve estar em contato constante com o cliente, para absorver o máximo sobre o domínio do sistema que está ajudando a desenvolver, e transmitir o conhecimento adquirido ao restante da equipe. Ele também é responsável pelas tarefas de concepção, levantamento, elaboração, negociação, especificação, validação e gestão dos requisitos da aplicação. Explicaremos cada uma dessas tarefas, baseados principalmente em (PRESSMAN, 2006), nos parágrafos a seguir.

Durante a fase de concepção, é feito um estudo sobre a viabilidade do projeto, por meio do qual se procura avaliar se existe realmente a necessidade de um sistema para resolver o problema, ou se há um mercado em potencial, que justifique o desenvolvimento da aplicação.

Após a confirmação da viabilidade do projeto, precisaremos definir as funcionalidades do sistema. Para isso, devemos fazer o levantamento de requisitos, o qual é feito geralmente através de entrevistas com os interessados, requisitando que estes respondam

a algumas questões sobre a maneira como o sistema deve se comportar, para atender suas necessidades. Esse processo costuma ser complicado, pois os entrevistados costumam omitir informações, não propositalmente, mas muitas vezes por achar que elas são óbvias. O responsável pela entrevista deve extrair o máximo de informações do entrevistado, atentando aos mínimos detalhes.

Levantados os requisitos da aplicação, o engenheiro de requisitos deverá cuidar da elaboração destes. Nessa fase, ele deverá criar modelos que representem as funcionalidades requeridas pelo cliente, refinando-as. Esses modelos podem ser feitos através de documentos ou diagramas UML (*Unified Modeling Language*) por exemplo, e servirão de artefatos base para os desenvolvedores.

É comum que, durante o processo de refinamento dos requisitos, surjam alguns empecilhos que possam, por exemplo, causar conflitos entre funcionalides ou atrasos no tempo estipulado para o desenvolvimento do projeto. Por isso, o engenheiro de requisitos deve estar sempre preparado para negociar os requisitos com o cliente.

Com os requisitos definidos, o engenheiro de requisitos agora deverá especificá-los, através de alguma documentação, que servirá de guia durante o desenvolvimento. A especificação de requisitos também serve de fonte, para que se possa acompanhar a evolução dos requisitos e rastreá-los de maneira mais fácil.

Após produzida, essa especificação de requisitos deverá passar por uma validação, para que se possa avaliar sua completude e detectar possíveis erros. É aconselhado que todos os *stakeholders* participem do processo de validação, para que se tenha vários pontos de vista sobre o problema discutido.

O trabalho do engenheiro de requisitos não acaba quando a especificação é concluída e validada. Requisitos de *software* são muito voláteis, mudam constantemente de acordo com as necessidades do cliente, por isso ele deve cuidar da gestão dos requisitos já especificados, rastreando mudanças, atentando a possíveis conflitos ou ambiguidades etc.

É importante ressaltar que não devemos atribuir toda a responsalidade da elicitação de requisitos ao engenheiro de requisitos, pois o cliente também exerce um papel importantíssimo neste processo, devendo estar sempre à disposição, tentar entender o processo de desenvolvimento e colaborar o máximo possível.

Alguns artefatos são produzidos durante o levantamento de requisitos, geralmente algum tipo de documentação formal especificando as necessidades do cliente, e um con-

trato que sirva de garantia para ambas as partes. Outros exemplos de artefatos são: protótipos do sistema, glossários, especificações de casos de uso (explicados mais detalhadamente no próximo sub-tópico), diagramas de casos de uso etc.

Requisitos devem ser levados sempre a sério e é sempre bom investir em tempo, para analisá-los cuidadosamente. Se interpretados de maneira errada, algo como a já clássica brincadeira da construção do balanço (Figura 2), presente em várias bibliografias sobre engenharia de requisitos, pode acontecer.

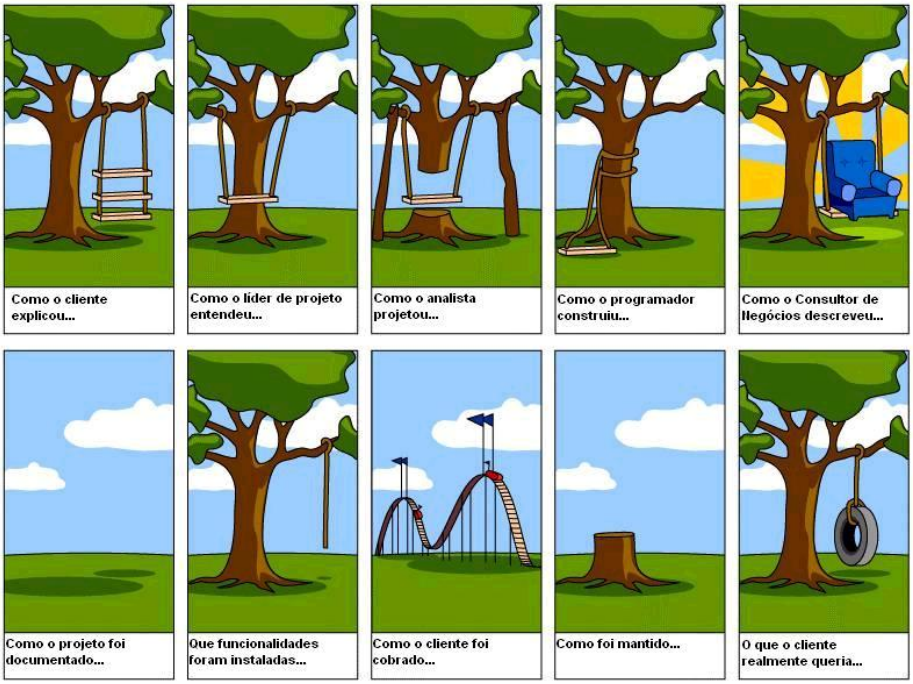

Figura 2: Problemas com a má comunicação de requisitos.
Fonte: http://amorimgabriel.files.wordpress.com/2009/10/projeto.jpg

2.6.1 Casos de Uso

Durante o desenvolvimento de um *software* existe sempre a necessidade de documentar as necessidades dos usuários e as funcionalidades que ele espera que a aplicação possua. Essa documentação serve de base para que os desenvolvedores possam projetar e implementar o produto, atingindo a meta principal: a satisfação do usuário.

Existem várias maneiras de se documentar os requisitos do usuário. Agilistas — seguidores de métodos ágeis (BECK; ANDRES, 2004) — costumam fazer isso em cartões,

contendo *user stories*, pequenas estórias geralmente escritas pelo próprio cliente, descrevendo uma funcionalidade da aplicação. Outros, que preferem metodologias mais formais, preferem fazer isso de maneira mais detalhada e escrevem longos documentos que tentam cobrir as necessidades do usuário, nos minímos detalhes. No final, ambos tem o mesmo objetivo: documentar as maneiras como os usuários irão interagir com o *software*, informando as ações dos mesmos e as repostas esperadas da aplicação.

A essa descrição de interações entre usuários e *software*, damos o nome de Casos de Uso. O conceito inicial de casos de uso foi feito por Jacobson (1992) e, de acordo com Cockburn (2001), podemos descrevê-los como contratos entre os *stakeholders*, que descrevem o comportamento esperado pelo sistema em diversas situações ou cenários.

Casos de uso podem ser representados através de documentos escritos em linguagem natural, diagramas, desenhos em papel etc. Apesar das inúmeras maneiras de representá-los, existe uma certa padronização sobre as partes que os compõem. Na página *http://www.usecases.org* pode se encontrar uma vasta documentação sobre casos de uso, o que inclui alguns exemplos de modelos que podem ser seguidos.

No geral, casos de uso são compostos por: atores, metas, pré-condições, cenários (estes podem ser divididos em: principais, alternativos e de exceção) e pós-condições. Nos parágrafos seguintes vamos discorrer sobre cada uma dessas partes.

Atores são entidades que irão interagir com o sistema, com o objetivo de alcançar alguma meta. Um ou mais atores podem participar de um caso de uso, podendo possuir diferentes objetivos. Uma curiosidade a se destacar é que atores não obrigatoriamente devem representar pessoas, eles podem ser outros sistemas, ou até mesmo um dispositivo periférico.

Um caso de uso geralmente representa uma meta do ator ao interagir com o *software*. Podemos citar como metas: cadastrar um cliente, consultar um item em estoque, validar o crédito de um consumidor etc. Alguns casos de uso podem possuir mais de uma meta, e para alcançá-las, às vezes é necessário que se cumpra alguma pré-condição. Logo, devemos declarar todas as pré-condições necessárias para a execução de um caso de uso em sua especificação. Suponhamos que um ator que representa o gerente de uma locadora de DVDs deseja registrar o aluguel de um filme. Poderíamos ter um caso de uso "Alugar DVD", que teria como meta "registrar o aluguel de um DVD", e pré-condições como: "O gerente deve estar autenticado no sistema" e "O cliente deve estar previamente cadastrado no sistema".

Nos cenários serão descritos o fluxo da interação entre ator e *software*. Entende-se como fluxo um conjunto de ações do ator feitas na aplicação. Alguns exemplos de ações são: preencher um campo de um formulário, clicar em um botão, selecionar uma opção. Além de ações do ator, podemos descrever também nos fluxos, respostas do *software* às ações do ator. Algumas possíveis respostas são: mensagens de confirmação, alertas de erros e notificações. Fluxos podem conter referências a outros fluxos, aos quais costuma-se dar o nome de sub-fluxos. Sub-fluxos podem ser do tipo alternativos ou de exceção. Fluxos alternativos geralmente são usados para expressar múltiplas opções descendentes de outro fluxo, enquanto fluxos de exceção expressam falhas.

Algumas vezes, após algum cenário, é necessário confirmar que alguma condição tenha sido atendida. Para representar essa condição, utilizamos as pós-condições. No exemplo da locadora de DVDs poderíamos acrescentar uma pós-condição em "Alugar DVD" que deve certificar que o *status* do DVD foi alterado para "alugado". Um exemplo de caso de uso é apresentado na Listagem 1.

```
Caso de uso: Efetuar saque.
Atores: Cliente.
Meta: O Cliente precisa efetuar um saque em sua conta.
Pré-condição: O Cliente precisa possuir uma conta no banco.
Cenário Principal - Sacar:
1. O cliente insere o cartão na ATM.
2. A ATM exibe o menu do sistema.
3. O cliente escolhe entre as opções:
- Conta Corrente (segue ao cenário alternativo: Sacar da Conta
Corrente).
- Poupança (segue ao cenário alternativo: Sacar da Poupança).
Cenário Alternativo: Sacar da Conta Corrente:
1. O cliente informa o valor.
2. A ATM requisita a senha.
3. O cliente informa a senha (Exceção: Senha Inválida).
4. A ATM libera o dinheiro.
Cenário Alternativo: Sacar da Conta Corrente:
1. O cliente informa o valor.
2. O cliente informa a variação.
3. A ATM requisita a senha.
4. O cliente informa a senha (Exceção: Senha Inválida).
5. A ATM libera o dinheiro.
Cenário de Exceção: Senha inválida:
1. A ATM exibe a mensagem: Senha inválida.
2. A ATM retorna ao menu principal.
```

Listagem 1: Exemplo de Caso de Uso.

Em Pressman (2006) o autor cita e estende algumas questões sugeridas por Jacobson (1992) as quais devem ser respondidas por um caso de uso, conforme listadas abaixo:

- Quem são os atores principais e os atores secundários?
- Quais são as metas dos atores?
- Que pré-condições devem existir antes de a história começar?
- Que tarefas ou funções principais são desempenhadas pelo ator?
- Que exceções deveriam ser consideradas quando a história é descrita?
- Que variações na interação dos atores são possíveis?
- Que informações do sistema o ator vai adquirir, produzir ou modificar?
- O ator terá de informar o sistema sobre alterações no ambiente externo?
- Que informações o ator deseja do sistema?
- O ator deseja ser informado sobre modificação inesperadas?

Um caso de uso que responde facilmente a essas perguntas, pode ser considerado razoavelmente bem especificado.

2.6.2 Glossário

Outro artefato que se costuma produzir durante o processo de engenharia de requisitos é o Glossário. Ele serve de guia para o desenvolvedor sobre os dados que persistirão na aplicação, pois contém regras para a utilização correta desses dados.

Algumas das regras definidas por glossários são as que dizem respeito ao tamanho máximo e mínimo dos dados, caracteres permitidos, atributos de uma entidade que devem ser obrigatoriamente mantidos, valores padrão etc. Outra grande utilidade deste artefato é descrever termos que fazem parte do domínio da aplicação e que são desconhecidos pelos desenvolvedores, eliminando problemas de interpretação.

Na Listagem 2 apresentamos um exemplo de glossário:

```
Glossário – Sistema de Locadora de DVDs
Entidades:
– Cliente:
    Descrição: entidade que representa um cliente da locadora de DVDs.
    Atributos:
      – Nome*: nome do cliente. Possui de 10 a 40 caracteres. Tipo
        texto.
      – CPF*: CPF do cliente. Possui 11 caracteres. Tipo texto.
      – Telefone: telefone do cliente. Possui 10 caracteres. Tipo
        numérico.
– DVD:
    Descrição: entidade que representa um DVD na locadora.
    Atributos:
      – Título: título do filme do DVD. Possui de 10 a 50 caracteres.
        Tipo texto.
      – Censura: idade da censura do DVD. Possui de 1 a 3 caracteres.
        Tipo numérico.
      – Status: define o status do DVD. Pode assumir os valores:
        NA PRATELEIRA, ALUGADO ou RESERVADO.
```

Listagem 2: Exemplo de Glossário.

2.7 Testes de *Software*

A atividade de testes está presente em praticamente qualquer ambiente de produção, pois desde automóveis a brinquedos precisam ser testados para assegurar um determinado nível de segurança e qualidade do produto. Com *software* não seria diferente, havendo vários métodos para testá-lo e assegurar sua qualidade. Ao longo dos anos, a atividade de testes tem se destacado cada vez mais como uma parte crucial do processo de desenvolvimento, ao ponto de surgirem filosofias como o *Test Driven Development* (TDD), que prega o desenvolvimento de testes antes mesmo de escrever o código propriamente dito (ASTELS, 2003).

Costumamos testar *software* com o objetivo de provar que o mesmo atende aos requisitos do cliente, e detectar falhas ou comportamentos indesejáveis (SOMMERVILLE, 2007). Em Pressman (2006), o autor discorre sobre algumas regras inicialmente defendidas por Myers (et al., 1979). Essas regras definem que um teste é um processo de execução de um programa com a finalidade de encontrar um erro (ou vários, quanto mais, melhor), e que um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.

A partir dessa premissa, podemos concluir que, para se testar um *software* devidamente, devemos desenvolver testes que tentem explorar falhas de maneiras inimagináveis, e que nos revelem a maior quantidade de erros possíveis.

Existem 3 principais métodos quando nos referimos a testes de *software*: testes de caixa preta (*black box*), caixa branca (*white box*) e caixa cinza (*gray box*). Os testes de caixa preta utilizam a abordagem do ponto de vista do usuário, por meio do qual não se conhece o funcionamento interno do sistema, apenas se sabe que, dada uma determinada entrada, espera-se que uma certa saída seja recebida. Este tipo de teste é geralmente aplicado com base na especificação dos requisitos do usuário, e tem o objetivo de verificar se estes requisitos foram atendidos.

É aconselhado que testes caixa preta sejam planejados previamente através de documentos denominados "Casos de Teste". Estes documentos contém especificações de como os testes aplicados sobre o *software* devem proceder.

Como complemento aos testes de caixa preta, temos os testes de caixa branca. Nestes, o analista de testes — a pessoa responsável pelos testes — tem conhecimento do funcionamento interno do *software*, e aplica os testes diretamente no código fonte. Testes de caixa branca são geralmente aplicados através de testes de unidade, que são testes escritos para verificação de pequenas porções do código fonte, geralmente classes ou métodos. Estes evitam grande parte dos *bugs* no *software*, principalmente se conseguem cobrir grande parte do seu código fonte, mas não devem ser utilizados como único critério de qualidade, e sempre que possível, devem ser complementados por outras formas de teste.

Podemos diferenciar os objetivos dos testes de caixa branca e preta da seguinte maneira: enquanto testes de caixa branca tem o objetivo de verificar se o programa está escrito corretamente, os testes de caixa preta tem o objetivo de verificar se o programa que escrevemos é o correto.

Ao mesclar os dois métodos anteriores, temos os testes de caixa cinza, que utilizam a vantagem do conhecimento do código fonte para descrever os testes em nível *black box*.

Podemos realizar testes de *software* com diversos objetivos, desde verificar se o código escrito provê o desempenho esperado pelo cliente, até testes de verificação da usabilidade das interfaces com o usuário. Citaremos a seguir alguns do principais tipos de teste de *software*:

- *Testes de Aceitação*: são testes caixa preta geralmente feitos pelo próprio cliente,

com os quais ele verificará se o *software* atende suas necessidades e se está pronto para utilização.

- *Testes de Integração*: consistem em verificar se diferentes unidades ou módulos de um *software* funcionam corretamente, quando postos para trabalhar em conjunto.
- *Testes de Unidade*: testam pequenas porções de código, como métodos e classes, verificando se estas funcionam corretamente.
- *Testes de Regressão*: verificam se funcionalidades de versões antigas do *software* continuam a funcionar corretamente após a inclusão de novas funcionalidades.
- *Testes de Recuperação*: são testes que forçam a falha do sistema para avaliar a sua capacidade de recuperação.
- *Testes de Estresse*: submetem o sistema a condições de utilização extremas, e verificam até que ponto ele pode surportar.
- *Testes de Desempenho*: verifica se o desempenho — o tempo que o sistema leva para responder as solicitações do usuário — do sistema atende as necessidades do cliente.

Como podemos notar, existem vários aspectos a serem considerados no teste de um *software*. Muitas vezes não há a possibilidade de se utilizar todos estes tipos de teste durante o desenvolvimento, mas devemos sempre executar o máximo de testes possíveis.

2.7.1 Testes Funcionais

Testes funcionais, também conhecidos como testes de aceitação, são testes caixa preta, com o objetivo de verificar se as necessidades do cliente são atendidas pelo *software*. De uma maneira mais formal, Wilson de Pádua (2003) define-os como testes projetados para verificar a consistência entre o produto implementado e os respectivos requisitos funcionais.

Estes testes são geralmente feitos pelo próprio cliente, o qual tentará desempenhar as atividades que ele planejava com o sistema, e verificará se obtém sucesso. Obviamente, como praticamente todas atividades do processo de desenvolvimento, a execução de testes funcionais pode ser automatizada por ferramentas especializadas. Uma das ferramentas para automação de testes funcionais mais conhecidas é o *Selenium*, que será apresentada posteriormente.

Ao aplicarmos testes de aceitação devemos verificar se a aplicação aceita as entradas corretas e responde corretamente a entradas erradas, se retorna valores corretos as ações do usuário, se os menus e janelas estão onde deveriam estar etc. Essas são algumas características que são avaliadas através de testes de aceitação. Características como desempenho do sistema, recuperação de falhas etc, não devem ser preocupações desse tipo de teste.

Para o desenvolvimento de bons testes funcionais é fundamental que os requisitos estejam bem definidos, de preferência em alguma especificação formal, fruto de um acordo entre o cliente e a equipe responsável pelo projeto. Especificações de requisitos completas são fundamentais para assegurar a qualidade dos testes funcionais.

2.8 Considerações Finais

Neste capítulo apresentamos conceitos necessários para o entendimento do projeto que desenvolvemos. Explicamos os fundamentos da Engenharia de *Software* através de um analogia a Engenharia Civil e mostramos o por quê de devermos buscar boas práticas de desenvolvimento, justificando através de dados apresentados no *Chaos Report*. Apresentamos também um arcabouço genérico para ilustrar o processo de desenvolvimento de *software* e fizemos uma pequena introdução à Engenharia de Requisitos e Testes de *Software*.

3 Trabalhos Relacionados

3.1 Considerações Iniciais

Neste capítulo discutiremos sobre trabalhos na área, desenvolvidos anteriormente por outros autores. Apresentaremos a abordagem destes autores apontando aspectos positivos que nos serviram de inspiração, e principalmente, destacando pontos que foram melhorados com a abordagem utilizada em nosso projeto.

3.2 Trabalhos Anteriores

Existem alguns trabalhos nesta área de pesquisa e acreditamos que alguns deles podem ser melhorados em certos aspectos. Nesta seção, apresentaremos alguns destes trabalhos, explicando as suas abordagens e as vantagens de nossa abordagem em relação aos mesmos.

Alguns destes estudos são relacionados à geração automática de testes baseados em requisitos, mas utilizam outras técnicas para a elicitação dos mesmos. Em (XIAOCHUN et al., 2008), os autores utilizam uma tabela de descrição de testes em que cada linha representa um passo do teste e cada coluna representa uma informação sobre o passo. Dentre as colunas temos: *Step*, representando o número do passo; *Keyword*, demonstrando a ação; *Object*, representando o objeto no qual a ação está sendo aplicada; *Value*, representando um valor que pode ser aplicado ao objeto e uma coluna para descrições extras.

Essa abordagem é similar à de nossa pesquisa, mas eles utilizam uma ferramenta comercial para executar os testes, o *Rational Functional Tester*, o que consideramos um fator não convidativo à utilização de sua ferramenta, dado o cenário *open source* que temos atualmente, com projetos livres tão bons quanto os proprietários.

Huang e Chen (2006) optaram por usar uma extensão do diagrama de atividades da UML (*Unified Modeling Language*) para a especificação de requisitos, declarando o fluxo dos cenários através de passos no diagrama. No entanto, acreditamos que essa abor-

dagem não segue um padrão da indústria de *software* atual, o que prejudica a aceitação.

Li e Miao (2008) escolheram utilizar o diagrama de casos de uso da UML, mas não o diagrama de casos de uso geralmente utilizado. Eles utilizaram o que chamaram de UCTM's (*Use Case Transition Models*) para criar uma árvore de possíveis cenários para o caso de uso. Depois de montada a árvore, eles utilizam essa estrutura para extrair prováveis casos de teste, mas não geram testes executáveis.

Essas abordagens utilizam métodos complexos para a especificação de requisitos, o que acreditamos, pode desestimular a colaboração do cliente no projeto. A utilização das abordagens destes trabalhos pode causar impactos em um processo de desenvolvimento de *software* real, por forçar a equipe responsável pelo projeto a aprender métodos incomuns.

Em nosso trabalho mantemos a utilização de casos de uso, um dos padrões mais usados para especificação de requisitos funcionais, adicionando apenas uma linguagem controlada em seu escopo. Por ser semelhante a uma linguagem natural, a linguagem controlada utilizada pela nossa ferramenta, oferece simplicidade, permitindo que clientes possam ajudar a equipe de desenvolvimento na elaboração dos requisitos e, consequentemente, dos testes para a aplicação.

Outra melhora que nosso trabalho apresenta em relação aos demais é a utilização de uma ferramenta mais moderna para execução dos testes funcionais: o *Selenium*. Este *framework* para testes de aplicações *Web*, permite que testes previamente especificados em alguma linguagem de programação, possam ser executados com apenas um comando.

Existem trabalhos relacionados também à geração de protótipos de interface. Em (ELKOUTBI et. al., 2006) os autores aplicam uma combinação de diagramas de colaboração, casos de uso e atividade da UML, suportados em especificações formais, para a geração de protótipos de interface da aplicação. Essa abordagem utiliza muitas e distintas fontes para a especificação, as quais acreditamos não ser amigável ao usuário.

Em (OLEK et. al., 2007) os autores desenvolveram uma ferramenta chamada *UC Workbench*, utilizada para a especificação de casos de uso e rascunhos de protótipos, para assim gerar um modelo de documentação. Através deste modelo, o usuário pode navegar através dos passos de um caso de uso e visualizar um rascunho de tela correspondente ao passo que ele escolheu. Entendemos que essa abordagem é muito boa sob o ponto de vista da usabilidade para o usuário, mas ela seria muito melhor se realmente gerasse protótipos funcionais destas telas, como propomos em nossa ferramenta.

3.3 Considerações Finais

Neste capítulo apresentamos alguns trabalhos desenvolvidos anteriormente que se relacionam de alguma forma ao nosso projeto. Contrastamos a abordagem destes trabalhos com a nossa abordagem, ressaltando pontos que serviram de motivação e pontos que acreditamos que podem ser melhorados de alguma maneira. Dentre estes aspectos que podem ser melhorados podemos citar:

- *Utilização de UML como método de especificação de requisitos:* o principal problema com a utilização de UML para definir requisitos é que, o padrão de meta-modelo utilizado, o XMI (*XML Metadata Interchange*), não é fortemente seguido por ferramentas CASE (*Computer-Aided Software Engineering*). Algumas ferramentas acabam implementando seu próprio padrão, o que pode dificultar a utilização de alguns destes trabalhos relacionados.

- *Geração de Casos de Teste apenas:* muitos dos projetos focam na criação de casos de teste apenas, dispensando a geração de testes executáveis, o que consideramos indispensável.

- *Utilização de ferramentas mais modernas:* alguns dos trabalhos relacionados utilizam ferramentas antigas, como o *HttpUnit*[1] por exemplo. Existem também casos que usam ferramentas proprietárias, o que pode ser um fator não convidativo a utilização do projeto. Em nosso trabalho, tentamos ao máximo utilizar ferramentas modernas e *open source*.

- *Aliar geração de protótipos e geração de testes:* como apresentado anteriormente, os trabalhos listados focam apenas em uma das vertentes, enquanto em nosso projeto, acreditamos que elas se complementam.

[1] http://httpunit.sourceforge.net/

4 Nossa Ferramenta

4.1 Considerações Iniciais

Neste capítulo discutiremos sobre a metodologia utilizada no desenvolvimento do projeto e ferramentas que utilizamos como suporte. Apresentaremos a arquitetura da nossa ferramenta componente a componente e finalmente mostraremos o funcionamento da ferramenta na prática.

4.2 Metodologia

Para desenvolver a ferramenta fizemos um estudo de modelos de especificação de casos de uso e, a partir destes modelos, estudados definimos o nosso próprio padrão de caso de uso. Nosso padrão é definido por uma linguagem natural controlada, que imita uma linguagem natural (o português), mas possui algumas restrições para torná-la mais formal.

Após definirmos nossa linguagem para especificação de casos de uso, criamos um interpretador para esta linguagem. O interpretador é responsável por traduzir os casos de uso recebidos como entrada em código para um *framework* de testes funcionais e em protótipos de interface em HTML.

Durante o desenvolvimento de nossa ferramenta, precisamos do suporte de outras para facilitar nosso trabalho. Visto que o nosso foco principal é a transformação de requisitos em testes, não necessitávamos desenvolver uma ferramenta para a aplicação dos testes, portanto utilizamos o *Selenium*.

Para interpretar os requisitos, cogitamos inicialmente a utilização de expressões regulares, mas como isso poderia ocasionar dificuldades na manutenção da ferramenta posteriormente, de acordo com sua evolução, preferimos utilizar uma ferramenta focada em construção de linguagens, ferramenta esta que é o ANTLR.

Nas próximas sub-seções apresentaremos estas ferramentas, explicando suas funcionalidades e o método de utilização.

4.2.1 *Selenium Framework*

Com a popularização das aplicações voltadas para o ambiente *Web*, várias ferramentas, com objetivo de dar suporte aos testes deste tipo de aplicação surgiram. Em meio a estes projetos, podemos citar: *Watir*[2], *HttpUnit*[3] e o *Selenium*[4]. Dentre essas ferramentas escolhemos utilizar o *Selenium*, um *framework open source*, portável e multiplataforma, que auxilia a execução de testes de aceitação em aplicações *Web*. Ele foi inicialmente desenvolvido pela *ThoughtWorks*[5] e hoje possui vários colaboradores importantes como *Google* e *Oracle*.

Optamos por esta ferramenta por permitir a utilização da linguagem Java no desenvolvimento dos testes, o que propicia melhor integração ao resto do projeto. Além disso, a comunidade responsável pelo seu desenvolvimento encontra-se bastante ativa, mantendo-o sempre atualizado e em constante evolução.

O *framework* é dividido em algumas ferramentas, dentre elas: *Selenium IDE*, *Selenium Remote Control*, *Selenium Grid*, *Selenium Core* e algumas outras, ainda em estados iniciais de desenvolvimento, como o *Selenium on Ruby/Rails* e o *CubicTest*. A Figura 3 ilustra de melhor maneira como a ferramenta é dividida, e a seguir descreveremos o funcionamento de algumas delas:

- *Selenium Core:* é um sistema para aplicação de testes feito em *JavaScript*. Serve de motor para o funcionamento de outras ferramentas do *framework* provendo as instruções de controle do *browser*.
- *Selenium IDE:* é uma IDE para desenvolvimento de testes *Selenium*, integrada ao navegador *Mozilla Firefox*[6], na forma de *plug-in*. Com ela é possível usufruir de todas as funcionalidades do *Selenium Core*, através de uma interface amigável. O *Selenium IDE* permite que o usuário grave os testes, faça *debugging*, utilize *autocomplete* e até mesmo exporte esses testes em forma de *scripts* para que possam ser utilizados posteriormente, ou editados manualmente.
- *Selenium Remote Control:* é uma API através da qual o desenvolvedor escreve testes na linguagem de sua preferência, para serem executados no *Remote Control Server*. Atualmente ela suporta as seguintes linguagens de programação: C#, Java, Perl,

[2]http://www.watir.com/
[3]http://httpunit.sourceforge.net/
[4]http://www.seleniumhq.org/
[5]http://www.thoughtworks.com/
[6]http://www.mozilla.com/firefox

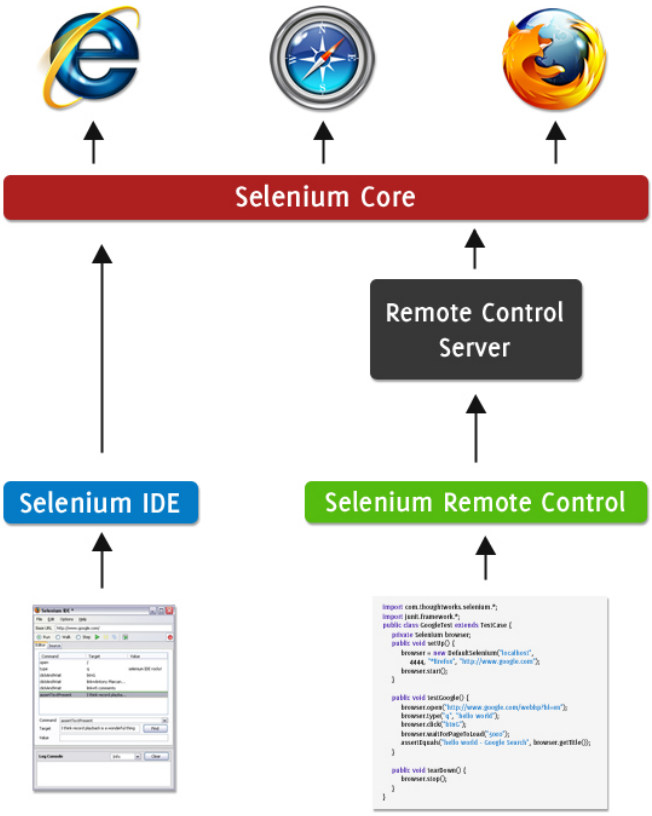

Figura 3: Arquitetura do *Selenium Framework*.

PHP, Python e Ruby. Um exemplo de código *Selenium Remote Control*, feito em Java, pode ser visto na Listagem 3, onde é feito um teste de consulta no *Google*. O código consiste em uma classe que estende obrigatoriamente *TestCase* e possui 3 métodos: método *setUp* que cuida das configurações iniciais antes de executar o teste, o método *testGoogle* que possui instruções do teste propriamente dito e o método *tearDown* que executa ações necessárias após a execução do teste.

- *Remote Control Server:* servidor responsável pela execução dos testes escritos em *Selenium RC*. Ao executarmos um teste escrito com esta API, o *Remote Control Server* cuida de iniciar uma sessão em um *browser*, executar as ações (clicar, preencher campos etc) especificados no teste sob a página, e encerrar a sessão no final da execução.

- *Selenium Grid:* dada a necessidade de se testar exaustivamente grandes aplicações, a execução de testes sequencialmente pode não ser viável. Para resolver este problema o *Selenium Grid* permite a execução de testes paralelamente em múltiplas máquinas, possibilitando até mesmo testes em ambientes heterogêneos.

```
import com.thoughtworks.selenium.*;
import junit.framework.*;

public class GoogleTest extends TestCase {

    private Selenium browser;

    public void setUp() {
        browser = new DefaultSelenium("localhost",
                  4444, "*firefox", "http://www.google.com");
        browser.start();
    }

    public void testGoogle() {
        browser.open("http://www.google.com/webhp?hl=en");
        browser.type("q", "hello world");
        browser.click("btnG");
        broswer.waitForPageToLoad("5000");
        assertEquals("hello world - Google Search",
                     browser.getTitle());
    }

    public void tearDown() {
        browser.stop();
    }
}
```

Listagem 3: Exemplo de código para um Teste *Selenium*.

4.2.2 ANTLR

Desenvolver interpretadores, compiladores, tradutores ou *parsers* para novas linguagens não é tarefa fácil, ela exige muito conhecimento e esforço do desenvolvedor, que muitas vezes conta com ferramentas pobres de recursos e de pouca ajuda. Para acabar com esse problema, surge o ANTLR, uma ferramenta que facilita o desenvolvimento desses tipos de aplicações, tornando a vida do desenvolvedor menos amarga.

De acordo com (PARR, 2007), o ANTLR é "...um sofisticado gerador de *parsers* que você pode usar para implementar interpretadores, compiladores, e outros tradutores para uma linguagem" (tradução livre do autor). A ferramenta foi criada pelo Prof. Ph.D Terence Parr, da *University of San Francisco*, começou a ser desenvolvida em 1989 e continua em evolução até a presente data.

Por ser de fácil utilização, poderoso, flexível, *open source* e possuir um bom ambiente para desenvolvimento da gramática, o ANTLR é largamente usado por programadores, principalmente para desenvolver *Domain Specific Languages* (DSLs). Uma DSL é uma linguagem para problemas de um domínio específico, como por exemplo, uma linguagem destinada à criação de jogos, que já incluiria em suas instruções básicas, comandos comuns à atividade de criação de jogos.

Para desenvolver um *parser* com ANTLR, devemos identificar as regras gramaticais da nossa linguagem, e especificá-las, geralmente utilizando o ambiente *ANTLR Works* (Figura 4). Este ambiente possui uma série de opções que facilitam o processo de desenvolvimento do *parser*, dentre elas: visualização de árvores gramaticais geradas, representações gráficas das regras, um interpretador para realização de testes, *debugger* integrado e até mesmo alguns comandos de refatoração.

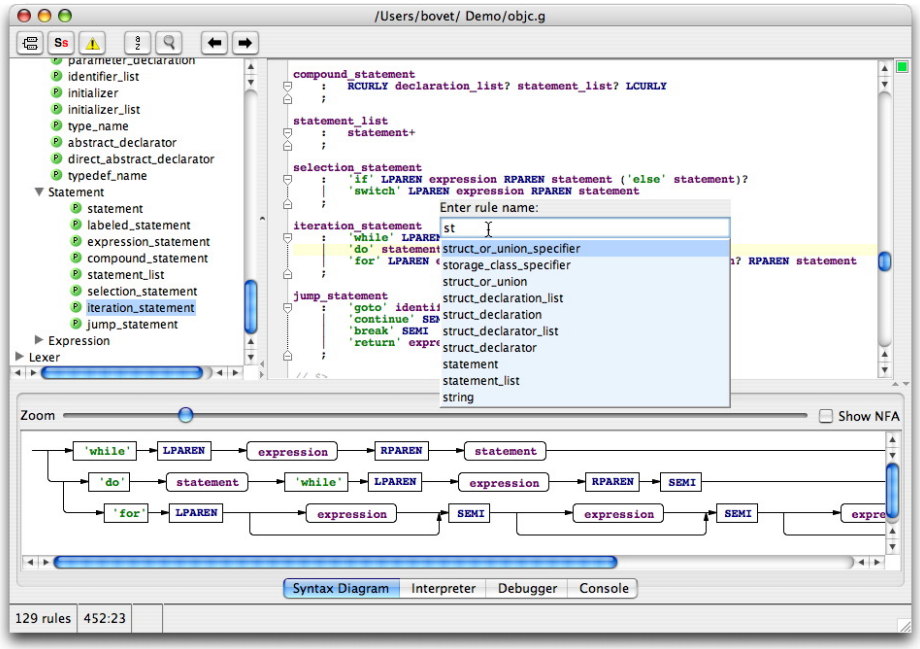

Figura 4: Editor do *ANTLR Works*.
Fonte: http://antlr.org/works/screenshots/editor.jpg

Devido a todas essas facilidades, o ANTLR permite que possamos desenvolver um *parser* de maneira rápida, sem muitas preocupações, de baixo nível e que atenda às necessidades da grande maioria dos desenvolvedores.

4.3 Arquitetura da Ferramenta

Nesta seção, mostraremos uma visão geral da arquitetura da ferramenta, discutindo o funcionamento de seus principais componentes.

Desenvolvemos uma ferramenta capaz de receber como entradas: um cenário de um caso de uso, um glossário de entidades do *software* e descrições da interface da aplicação para, então, traduzí-los em testes funcionais para o *Selenium* e protótipos de interface em HTML. A Figura 5 apresenta a divisão e interação entre os componentes da ferramenta. Mais detalhes sobre a implementação da ferramenta são encontradas no Apêndice 2.

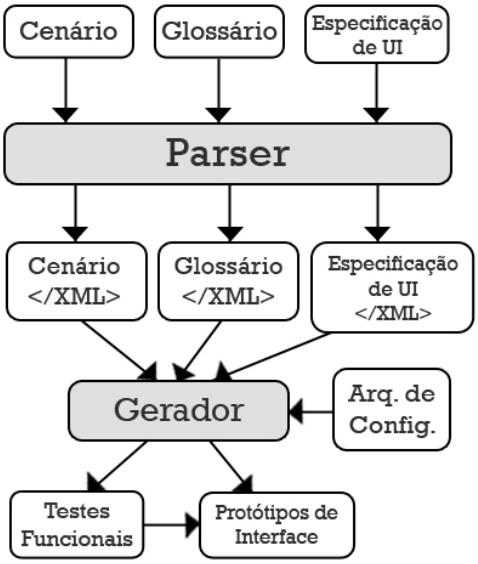

Figura 5: Arquitetura da Ferramenta.

Todos os artefatos que a ferramenta recebe como entrada – cenários, glossários e descrições de interface – devem ser escritos utilizando a linguagem natural controlada, definida pelo interpretador da ferramenta, que foi contruído utilizando o ANTLR. Mais detalhes sobre a implementação da gramática para esta linguagem são encontrados no Apêndice 1.

Depois de ler todos os artefatos, o *parser* irá gerar como saída arquivos XML (*Extensible Markup Language*) representando os cenários, glossários e interfaces da aplicação. Estes serão passados, juntamente com um arquivo de configuração, aos geradores da ferramenta. Finalmente, os geradores irão processar os arquivos XML e transformá-los em testes funcionais e páginas HTML.

Os testes funcionais gerados pela nossa ferramenta são testes para o *Selenium*, um *framework open source* que possui um conjunto de ferramentas para testar aplicações Web. Uma das mais conhecidas ferramentas – e a que utilizamos neste trabalho – é o *Selenium Remote Control*, geralmente chamado de *Selenium RC*.

O *Selenium RC* provê uma API para a programação de testes funcionais em várias linguagens de programação. Testes escritos utilizando esta API podem ser executados em um *Selenium Server*, que tratará de executar uma instância de um navegador e simular a interação do usuário com a página em questão.

Nos próximos tópicos explicaremos melhor cada componente da arquitetura. O funcionamento desses componentes será demonstrado utilizando o exemplo de um cenário simples, no qual o usuário deseja trocar sua senha na aplicação. Para fazer isto, o usuário precisa estar autenticado na aplicação, preencher um formulário contendo três campos ("Senha Atual", "Nova Senha" e "Repetir Nova Senha"), e então clicar em um botão "Trocar Senha".

4.4 Cenários

Os casos de uso escritos para a nossa ferramenta devem ser descritos a partir de uma linguagem natural controlada. Trabalhos realizados anteriormente (SCHWITTER, 2002) provam que esta abordagem pode ser bastante útil pela facilidade de compreensão, o que favorece a utilização da ferramenta por usuários comuns, não necessariamente especialistas em termos da computação.

A linguagem que criamos define a especificação de um cenário como uma sequência de passos, onde cada passo é formado por um ator, uma ação exercida por ele e um objeto que sofre a ação.

O ator representa o usuário que está utilizando a aplicação ou a própria aplicação. A ação é um verbo que representa ações como clicar, preencher, selecionar etc. Cada um desses verbos integrados à linguagem foi mapeado para representar uma chamada de método da API do *Selenium*. Desta forma, ao declararmos uma ação estamos na verdade especificando um método *Selenium* que será executado. A Tabela 1 apresenta algumas relações entre verbos e métodos da API *Selenium*.

Na Listagem 4, apresentamos um exemplo de como o cenário "Alterar Senha" se-

Action Verb	**Method**
marca (uma *checkbox/radio*)	*check(locator : String) : void*
clica (em um botão/*link*/*checkbox*/*radio*)	*click(locator : String) : void*
preenche (um campo de texto)	*type(locator : String, value : String) : void*
seleciona (um item em um *dropdown menu*)	*select(selectLocator : String, optionLocator : String) : void*

Tabela 1: Verbos e respectivos métodos na API *Selenium*.

ria especificado, utilizando a linguagem definida pela ferramenta. A estrutura do cenário possui um título que tem como objetivo identificar o cenário, uma pré-condição (opcional) que define alguns passos que deverão ser executados antes que o cenário propriamente dito possa ser executado, e um conjunto de passos que descrevem as ações que serão executadas no cenários.

Utilizamos a abordagem de cenários por estes descreverem bem como o usuário deverá utilizar o sistema e por ser de fácil entendimento tanto por parte dos desenvolvedores como por parte dos usuários. Em Carrol (1999) o autor enumera outras vantagens da utilização de cenários.

```
Cenario 1 - Alterar Senha:
Pre-condicao:
- Executar Login;
1. A aplicação exibe a seguite tela: "Alterar Senha"
2. Usuario preenche campo "Senha Atual"
3. Usuario preenche campo "Nova Senha"
4. Usuario preenche campo "Repetir Nova Senha"
5. Usuario clicar botao "Alterar"
```

Listagem 4: Exemplo de Especificação de Cenário.

4.5 Glossário

O glossário possui informações sobre as entidades presentes na aplicação e seus respectivos atributos. Dentre estas informações podemos citar: tamanhos de atributos, atributos obrigatórios, tipos de dados etc. Ele será utilizado sempre que a ferramenta necessitar de alguma dessas informações, como por exemplo, ao gerar os protótipos de interface, o gerador necessita saber a quantidade de caracteres máximos de um determinado campo.

No exemplo dado (Listagem 5), temos um glossário composto por apenas umas entidade: um Usuário. A descrição de uma entidade no glossário é composta por uma breve descrição de seu significado e uma lista de atributos.

Em nosso exemplo, os atributos de um usuário são: um *login*, que é do tipo texto, obrigatório (representado pelo *) e um tamanho máximo de 20 caracteres; um *e-mail*, que é de um tipo *e-mail* pré-definido; uma senha, que é obrigatória, do tipo senha e tamanho máximo de 8 caracteres.

```
Entidade - Usuário
Descricao: Representa um usuário no sistema.
Atributos:
- Login*, TEXTO(20);
- E-mail, EMAIL;
- Senha*, SENHA(8);
```

Listagem 5: Exemplo de Especificação de Glossário.

4.6 Especificações de Interface

Quase todo caso de uso possui uma interface com a qual o usuário irá interagir. Pensando nisso, criamos uma linguagem natural controlada simples para especificar estas interfaces. Um exemplo de interface descrita utilizando esta linguagem é apresentada na Listagem 6.

```
Nome da Página: Alterar Senha
- Senha "Senha Atual" {Usuario.Senha} Escrita
- Senha "Nova Senha" {Usuario.Senha} Escrita
- Senha "Repetir Nova Senha" {Usuario.Senha} Escrita
- Botao "Alterar"
```

Listagem 6: Exemplo de Especificação de Interface.

Uma interface é descrita por um conjunto de componentes HTML como: *inputs*, *checkboxes*, botões, *drop down menus* etc. No exemplo dado, a página "Alterar Senha" possui três componentes *input* do tipo senha e um botão "Alterar". Cada componente está anotado com a palavra-chave "Escrita", que indica que este campo pode receber um valor. A ferramenta também suporta a anotação "Leitura", que indica que um campo é do tipo *read only*.

Cada componente é também anotado com uma referência a um item no glossário. Por exemplo, "{Usuario.Senha}" significa que o dado componente se refere ao atributo Se-

nha da entidade Usuario. Desta forma, a ferramenta pode encontrar todas as informações complementares sobre o atributo em questão, como as citadas no tópico anterior.

4.7 Interpretador e Arquivos XML

O *parser* é o componente da ferramenta responsável por receber especificações de cenários, descrições de interface e glossários escritos em nossa linguagem natural controlada e convertê-los em um formato mais legível ao gerador. O formato escolhido foi o XML, que é um padrão mantido pelo *World Wide Web Consortium* [7], com o objetivo de prover uma forma para representação de dados de maneira simples, estruturada e generalista. Este padrão permite também que os arquivos possam ser utilizados por outra ferramenta posteriormente.

Cada arquivo XML contém todas as informações de seu respectivo artefato, mas de uma forma mais estruturada. Desta forma, o gerador pode acessar e trabalhar com os dados extraídos mais facilmente. Na Listagem 7 apresentamos um exemplo de arquivo XML gerado a partir do cenário "Alterar Senha".

```
<scenario>
  <name>Alterar Senha</name>
  <precondition>
    <execute>Login</execute>
  </precondition>
  <step>
    <stepId>1</stepId>
    <actor>Sistema</actor>
    <action>
      <display>Alterar Senha</display>
    </action>
  </step>
  <step>
    <stepId>2</stepId>
    <actor>Usuario</actor>
    <action>
      <fill>Senha Atual</fill>
    </action>
  </step>

  ...
```

[7] http://www.w3.org

```
  <step>
    <stepId>5</stepId>
    <actor>Usuario</actor>
    <action>
      <click>Alterar</click>
    </action>
  </step>
</scenario>
```

Listagem 7: Arquivo XML gerado a partir da especificação de cenário.

O XML que representa o cenário organiza todas as informações relevantes sobre este agrupadas em "nós". Existe um nó principal chamado *scenario* que possui outros nós filhos, representando a estrutura de um cenário. O nó *name* é utilizado para definir o nome do cenário, desta forma servindo como identificador. Temos também um nó chamado *precondition* que define ações que devem ser executadas antes do cenário principal iniciar (no exemplo dado, a ferramenta deve executar o cenário "Login" antes de iniciar o "Alterar Senha"). Após este, temos uma lista de nós *step*, que são compostos por um nó *stepId* que funciona como identificador para o nó; o nó *actor*, representando o ator que está atuando no passo; e o nó *action* que contém nós específicos, representando o tipo de ação executada sobre um item da aplicação. Como podemos ver em nosso exemplo, existem três tipos de ações: exibir, preencher e clicar.

Na Listagem 8, apresentamos outro exemplo de arquivo XML, desta vez representando uma interface da aplicação. O XML chega a ser auto explicativo, ele mapeia todos os componentes HTML que aparecerão na página. Temos um nó principal *ui-specification* com dois nós filhos: um *title*, que agrega um nome; um *html-component*, que possui um tipo e alguns atributos, como tamanho máximo do campo. Os valores para o título, e nomes dos campos são provenientes de descrição da interface, enquanto atributos como valores máximo de um campo são extraídos do glossário.

```
<ui-specification>
    <title>Alterar Senha</title>
    <html-component>
      <password>
        <name>Senha Atual</name>
        <length>8</length>
      </password>
    </html-component>

    ...

    <html-component>
```

```
      <button>
        <name>Alterar</name>
      </button>
    </html-component>
</ui-specification>
```

Listagem 8: Arquivo XML gerado a partir de uma descrição de interface.

4.8 Arquivo de Configuração

O arquivo de configuração possui propriedades complementares, necessárias para que a ferramenta possa trabalhar corretamente. Podemos utilizá-lo para configurar arquivos de estilização como arquivos CSS (*Cascading Style Sheet*), para enumerar os itens do menu da aplicação etc.

4.9 Gerador

O gerador é o componente da ferramenta responsável por ler os arquivos XML e traduzí-los em código real. Nós desenvolvemos dois geradores: o primeiro traduz XML's com descrições de interfaces em páginas HTML e o segundo traduz XML's de cenários em testes funcionais para o *Selenium*.

O gerador de interfaces recebe o XML contendo as descrições da mesma, lê a lista de componentes, e gera um arquivo HTML equivalente. Se passarmos o XML da Listagem 8 para o gerador, ele criará uma página HTML contendo três *inputs* do tipo senha e um botão chamado "Alterar".

```
<html>
<head>
  <meta http-equiv="Content-type"
  content="text/html; charset=utf-8">
  <title>Alterar Senha</title>
  <link rel="stylesheet" href="/css/style.css"
  type="text/css" media="screen"
  title="no title" charset="utf-8">
</head>
<body>
  <label for="senha_atual">Senha Atual</label>
  <input type="password" name="senha_atual"
  id="senha_atual" maxlength="8">
  <br>
```

```
  ...

  <input type="button" name="alterar"
  value="Alterar">
  <br>
</body>
</html>
```

Listagem 9: Exemplo de página HTML gerada.

A Listagem 9 mostra a página HTML gerada. Cada nó *html-component* presente no arquivo XML é convertido em uma *tag* HTML, com seus atributos preenchidos com valores presentes no nó. Informações complementares como o caminho para o arquivo CSS são provenientes do arquivo de configuração. A visualização da página gerada é mostrada na Figura 6.

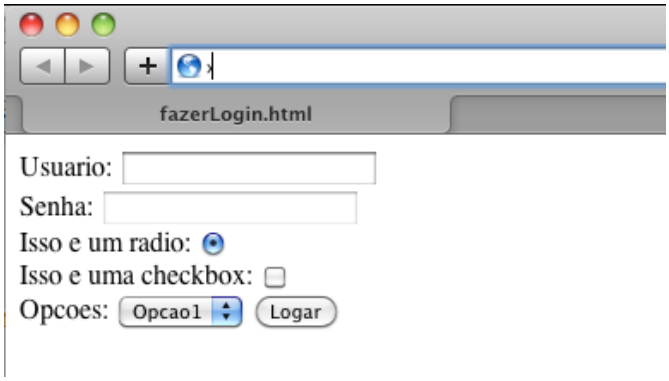

Figura 6: Página HTML gerada pela ferramenta.

Da mesma forma, o gerador de testes funcionais extrai do arquivo XML os nós *step* e os traduz em chamadas de métodos da *Selenium Remote Control API*. Cada método é definido por um verbo mapeado previamente, por exemplo, o verbo "clicar" corresponde a uma chamada do método *click* no *Selenium*.

Os componentes nos quais as ações serão aplicadas são definidos no conteúdo do nó que representa a ação. O identificador para o componente deve ser idêntico ao informado nas especificações de interface. Os valores que serão informados em um campo são, em alguns casos, gerados aleatoriamente, e em outros, são valores padrões pré-definidos.

Se considerarmos o arquivo XML apresentado na Listagem 7, o gerador irá gerar como saída o seguinte trecho de código *Selenium*:

```
import junit.framework.TestCase;
import com.thoughtworks.selenium.*;

public class AlterarSenhaTest extends TestCase {

  private Selenium browser;

  public void setUp() {
    browser = new DefaultSelenium("localhost",
        4444, "*firefox",
        "http://localhost:8080/TestApplication");
    browser.start();
  }

  public void testChangePassword()
                throws InterruptedException {
    browser.showContextualBanner();
    browser.open("/");

    browser.type("name=Senha Atual",
                          "valor aleatorio aqui");
    browser.type("name=Nova Senha",
                          "valor aleatorio aqui");
    ....

    browser.click("Alterar");
  }

  public void tearDown() {
    browser.stop();
  }
}
```

Listagem 10: Código *Selenium* gerado a partir da especificação de cenário.

Quando executamos um destes testes, o servidor do *Selenium* irá executar uma instância do navegador e carregar a página referente ao cenário. Como o cenário "Alterar Senha" possui uma pré-condição, o *Selenium* irá executar primeiramente o cenário da pré-condição (no caso o cenário "Login") antes de qualquer coisa. Depois disso, a sequência principal de passos do cenário será executada, ou seja, o teste irá preencher os campos "Senha Atual", "Nova Senha", irá repeti-la e, então, clicar no botão "Alterar", exatamente como o usuário iria fazer, e verificar se algum erro ocorre. Como exemplo de erro podemos citar o seguinte: suponha que o desenvolvedor esqueceu de colocar, na página, o campo "Repetir Nova Senha"; ao executar o teste, este notaria a ausência do campo e lançaria um erro, alertando que um campo está faltando na página.

4.10 A Ferramenta na Prática

A ferramenta foi desenvolvida como um *plug-in* (CLAYBERG e RUBEL, 2008) para a IDE *Eclipse*[8], desta forma ela propicia maior integração com o restante do projeto da aplicação, visto que o uso de múltiplas ferramentas para desenvolvimento é suscetível à perda de foco do desenvolvedor.

O *plug-in* que desenvolvemos possui editores especiais para a edição de cenários, glossários e especificações de interface escritos em nossa linguagem natural controlada. Os editores possuem funcionalidades de destaque e *auto complete* para palavras reservadas, como mostrado na Figura 7. Detalhes sobre a instalação do *plug-in* podem ser encontrados no Apêndice 3.

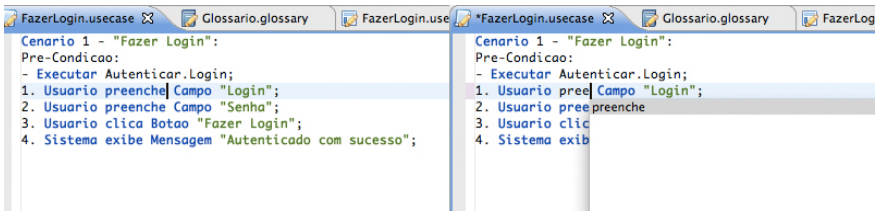

Figura 7: Editores do *Plug-in*.

Para utilizar a ferramenta, sugerimos que o usuário mantenha a estrutura de seu projeto como mostrado na Figura 8, organizando cenários, especificações de interface e glossários, separadamente. Os arquivos de cenários, especificações de interface e glossários deverão possuir extensão "*.usecase*", "*.userinterface*" e "*.glossary*", respectivamente.

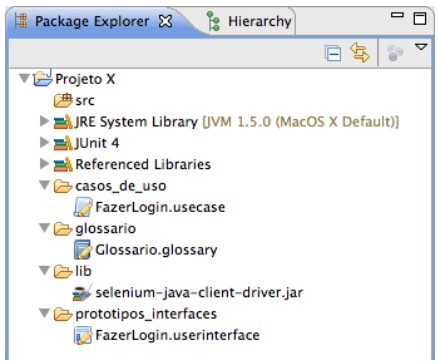

Figura 8: Estrutura recomendada para o projeto.

[8]http://www.eclipse.org

Dependendo do editor que estiver aberto no momento, um botão específico para interpretá-lo irá aparecer na *toolbar* do *Eclipse*. Este botão só aparecerá se, e somente se, um editor para um dos arquivos estiver aberto, e somente o botão para o respectivo tipo de arquivo aparecerá, como ilustrado na Figura 9.

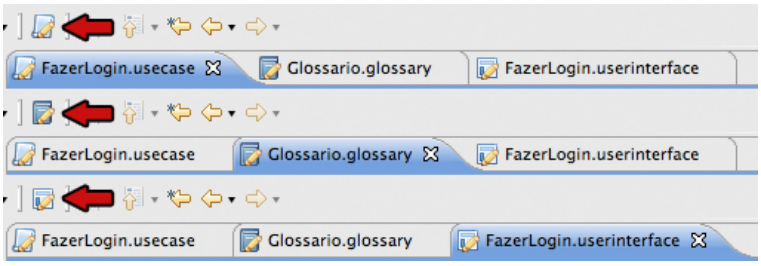

Figura 9: Botões para interpretação dos artefatos.

Ao clicar no botão para interpretação de um glossário, uma estrutura de diretório para arquivos XML será criada, e dentro deste diretório, será gerado um arquivo XML representando o glossário em questão (Figura 10).

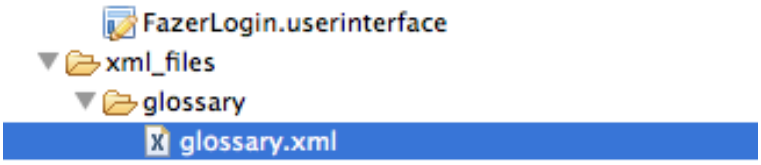

Figura 10: Estrutura de diretório XML para glossários.

Algo semelhante acontece quando o usuário clica no botão para interpretar uma descrição de interface. A ferramenta irá: criar o diretório para arquivos XML, caso o mesmo não tenha sido criado anteriormente; criar um diretório para as páginas HTML geradas; gerar uma página HTML equivalente a descrição de interface (Ver Figura 11).

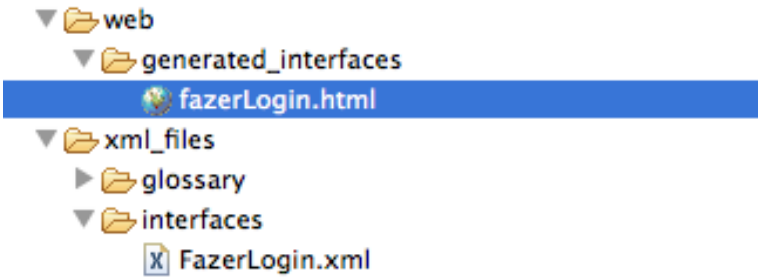

Figura 11: Estrutura de diretório XML e Web para interfaces.

O botão para interpretar cenários funciona da mesma forma, mas no lugar de um diretório para páginas HTML, será criado um pacote *generatedTests* na pasta *src* do projeto, e um teste para o cenário dentro deste pacote (Figura 12).

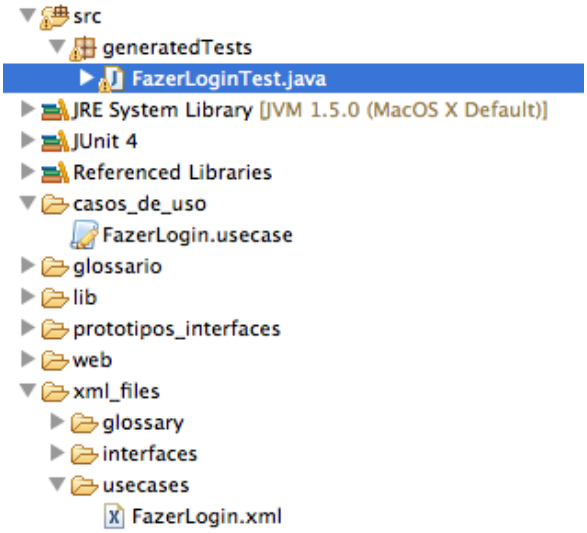

Figura 12: Estrutura de diretório XML e pacote para testes.

Depois de gerado o código, o desenvolvedor deve se responsabilizar por resolver dependências como: inserir as bibliotecas do *Selenium* no *buildpath* do projeto, iniciar o *Selenium Server* e, finalmente, executar os testes.

4.11 Considerações Finais

Neste capítulo discutimos sobre a metodologia empregada durante o projeto e ferramentas que utilizamos como apoio. Apresentamos a arquitetura da ferramenta, mostrando a fundo cada componente e artefatos de entrada e saída. Finalizamos o capítulo mostrando o funcionamento da ferramenta na prática, ilustrando as funcionalidades do *plug-in* para o *Eclipse*.

5 Conclusões

5.1 Considerações Iniciais

Neste capítulo apresentamos as conclusões alcançadas com este trabalho, listamos nossas contribuições, aspectos que ainda precisam ser melhorados e sugerimos alguns trabalhos futuros para o projeto.

5.2 Conclusões do Projeto

A ferramenta que desenvolvemos coloca agilidade e formalidade lado a lado e traz os melhores aspectos de ambos os mundos. Nós aumentamos a velocidade do processo de desenvolvimento ao oferecermos uma ferramenta para a automação da geração de testes funcionais e protótipos de interface, enquanto aplicamos o formalismo necessário para que requisitos possam prover testes funcionais e interfaces mais fiéis e confiáveis com o usuário. A ferramenta também segue o método de desenvolvimento dirigido a casos de uso, o que pode ser considerado um padrão em projetos de *software* atualmente.

Além do mais, a simplicidade de nossa linguagem natural controlada permite ao cliente colaborar com a tarefa de especificação, integrando-o à equipe de desenvolvimento, o que resulta em menos problemas de comunicação e o encoraja a seguir o projeto mais de perto. Outro aspecto que estimila a colaboração do cliente é a geração de protótipos de interface. Estes protótipos permitem que o cliente possa fazer críticas ao projeto em estágios iniciais de desenvolvimento, evitando assim mau entendimentos por parte dos desenvolvedores em relação aos requisitos.

Existem poucas ferramentas práticas relacionados à geração de testes funcionais e/ou protótipos de interface para aplicações *Web* e este trabalho vem a aumentar esse número. Dentre as principais vantagens de nosso trabalho em relação a projetos anteriores, podemos citar o método simplificado de especificação dos requisitos e a utilização de ferramentas mais modernas como o *ANTLR* para definição da linguagem e o *Selenium* para execução dos testes funcionais.

Existem alguns pontos em nosso trabalho que podem ser melhorados. O principal deles é o teste em um ambiente real de desenvolvimento. Sem este teste não podemos avaliar as vantagens ou desvantagens que a ferramenta pode trazer a ambientes de desenvolvimento.

Outro ponto que pode ser melhorado diz respeito ao escopo do projeto. Inicialmente focamos apenas em cenários do padrão CRUD (ou seja, casos de inclusão, leitura, atualização e remoção de registros em um *software*). Acreditamos que a pesquisa tende a evoluir a partir do momento que passarmos a abordar cenários mais complexos que fujam desse escopo.

5.3 Trabalhos Futuros

A ferramenta se encontra atualmente em uma versão beta e ainda pode ser melhorada em diversos aspectos. Nesta seção, enumeramos alguns trabalhos que achamos interessante serem realizados futuramente.

- *Suporte a cenários mais complexos:* já que a ferramenta tem como foco inicial cenários do padrão CRUD, devemos melhorá-la adicionando suporte a cenários mais complexos, como casos que podem derivar múltiplos subcenários ou casos envolvendo cenários que ocorram paralelamente. Isto pode ser feito com a extensão da linguagem controlada e com melhorias na implementação do gerador.

- *Geração inteligente de entradas para os testes:* no estado atual, o gerador atribui valores aleatórios as entradas utilizadas durante os testes, salvo alguns casos, onde utiliza-se valores pré-definidos. Seria interessante se estas entradas apresentassem valores significativos, desta forma economizariamos tempo na edição dos testes gerados e teriamos testes ainda melhores. Para solucionar este problema poderíamos utilizar dados do glossário para extrair valores relevantes para estas entradas, como por exemplo: se um campo possui uma restrição de valores entre 1 e 100, o gerador deveria gerar valores dentro e fora deste escopo, para aumentar o escopo dos testes.

- *Geração de múltiplos casos de teste:* outra funcionalidade interessante seria a geração de um conjunto de casos de teste para aplicação. Já que podem ocorrer vários tipos de situação dentro de um cenário, como por exemplo, dentro de um cenário de autenticação no sistema, o usuário poderia: 1) Preencher os campos “Usuário” e “Senha”, e em seguida clicar em “Autenticar”; 2) Preencher apenas o campo “Usuário” e clicar em “Autenticar”; 3) Preenchar apenar o campo “Senha” e clicar em “Autenticar”. Se a ferramenta puder gerar testes para o máximo de situações

possíveis, teremos maior cobertura de testes para aplicação e conseguiremos detectar uma maior quantidade de erros. Isto poderia ser feito através da combinação de passos de um cenário, gerando uma árvore de possíveis subcenários presentes em um cenário principal.

- *Melhorias na interface e na usabilidade da ferramenta:* a interface da ferramenta foi desenvolvida como um *plug-in* para a IDE *Eclipse*, que é um dos ambientes de desenvolvimento mais utilizados atualmente. No entando, existem desenvolvedores que não utilizam esta IDE, então acreditamos que seja interessante desenvolver uma interface que possa atrair estes desenvolvedores também. Uma ferramenta desacoplada de uma IDE tiraria um pouco da integração com o restante do projeto, mas seria uma forma mais genérica de utilizar a ferramenta, permitindo que outros desenvolvedores a utilizassem. Outra melhoria em relação à interface seria adicionar funcionalidade de *drag and drop* – arrastar e soltar componentes – para composição das interfaces com o usuário.

- *Estudo dos impactos da ferramenta em um projeto de* software *real:* talvez o mais importante dos trabalhos futuros seja este. Através do teste em um projeto real poderemos quantificar a redução de tempo e trabalho que a geração automática de testes e interfaces pode trazer, além de verificar a melhoria na qualidade dos testes produzidos depois da adoção da ferramenta. Outro fator que poderá ser observado é a usabilidade da ferramenta para o cliente, verificando se o mesmo pode utilizá-la e compreendê-la o suficiente para ajudar na especificação de requisitos.

- *Suporte a outras línguas:* a linguagem natural controlada, desenvolvida neste trabalho, é baseada no Português. Seria interessante melhorar a arquitetura da ferramenta para que se possa integrar suporte a outras linguagens naturais controladas, baseadas em outras línguas.

5.4 Considerações Finais

Neste capítulo listamos as conclusões alcançadas com este projeto, apresentando aspectos positivos e aspectos que podem ser melhorados. Em relação a estes aspectos que podem ser melhorados, apresentamos uma lista de trabalhos futuros e possíveis formas de implementá-los.

Referências

ASTELS, D. ***Test-Driven Development:*** *A Practical Guide.* Prentice Hall, 2003.

BECK, K.; ANDRES, C. ***Extreme Programming Explained:*** *Embrace Change.* Addison-Wesley, 2004.

CARROL, J. M. ***Five Reasons for Scenario-Based Design***. 32nd Hawaii International Conference on System Sciences, 1999.

CLAYBERG, E.; RUBEL, D. ***Eclipse Plug-ins***. 3. ed. Addison-Wesley, 2008.

COCKBURN, A. ***Writing Effective Use Cases***. 1. ed. Addison-Wesley, 2001.

ELKOUTBI, M.; KHRISS, I.; KELLER, R. K. ***Automated Prototyping of User Interfaces Based on UML Scenarios***. *Automated Software Engineering*, 2006.

FILHO, W. de P. P. **Engenharia de Software:** Fundamentos, Métodos e Padrões. LTC, 2003.

HUANG, C.; CHEN, H. ***A Tool to Support Automated Testing for Web Application Scenario***. The IEEE International Conference on Systems, Man and Cybernetics, 2008.

IEEE. ***IEEE Standard Glossary of Software Engineering Terminology***. IEEE Computer Society Press, 1990.

JACOBSON, I. ***Object-Oriented Software Engineering:*** *A Use Case Driven Approach.* 1. ed. Addison-Wesley, 1992.

LI, L.; MIAO, H. ***An Approach to Modeling and Testing Web Applications Based on Use Cases***. 2008 International Symposium on Information Science and Engineering, 2008.

MYERS, J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. ***The Art of Software Testing***. Wiley, 1979.

OLEK, L.; MICHALIK, B.; NAWROCKI, J.; OCHODEK, M. ***Quick Prototyping of Web Applications***. Balancing Agility and Formalism in Software Engineering, 2007.

PARR, T. ***The Definitive ANTLR Reference:*** *Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

PRESSMAN, R. S. **Engenharia de Software**. 6. ed. McGraw-Hill, 2006.

SCHWITTER, R. ***English as a Formal Specification Language***. 13th International Workshop on Database and Expert System Applications, 2002.

SOMMERVILLE, I. **Engenharia de Software**. Addison Wesley, 2007.

The Standish Group. ***Chaos Chronicles***. Standish Reports, 2004.

WIEGERS, K. E. ***Software Requirements***. 2. ed. Microsoft Press, 2003.

XIAOCHUN, Z.; BO, Z.; JUEFENG, L.; QIU, G. ***A Test Automation Solution on GUI Functional Test***. The IEEE International Conference on Industrial Informatics, 2008.

APÊNDICES

Apêndice 1 - Gramáticas

Neste apêndice apresentamos o código da gramática utilizada na nossa ferramenta, compátivel com ANTLR v3.

Gramática para Glossários:

```
glossary = entityList

entityList = entity+

entity = 'Entidade' '-' entityName ':' 'Descricao:' text* ';'
'Atributos:' attributeList

entityName = (ID)+

text = (WHITESPACE | ID | '"' | ',' | '/')+

attributeList = (attribute)+

attribute = '-' ID STAR? ',' attributeType ';'

attributeType = textType | INTEGER | FLOATING_POINT

textType = TEXT '(' INT ')'

FLOATING_POINT = 'PONTO FLUTUANTE'
INTEGER        = 'INTEIRO'
CPF            = 'CPF'
CNPJ           = 'CNPJ'
LIST           = 'LISTA'
TEXT           = 'TEXTO'
GRID           = '#'
MULTIPLA       = 'MULTIPLA'
DATA           = 'DATA'
STAR           = '*'
ID             = (('a'..'z'|'A'..'Z')+ INT? ('a'..'z'|'A'..'Z')*)+
INT            = '0'..'9'+
FLOAT          = ('0'..'9')+ '.' ('0'..'9')+
NEWLINE        = '\r'? '\n'
WHITESPACE     = (' '|'\t')+
```

Listagem 11: Gramática para o Glossário.

Gramática para Descrições de Interface:

```
uiPrototype = 'Interface_com_Usuario' '-' '"' multiWordID '"' ':'
htmlComponentList

htmlComponentList = (htmlComponent)+

htmlComponent = select | field | button | radioButton | checkBox

attributeReference = '{' attributeid=attributeId '}'

attributeId = multiWordID '.' (multiWordID | attributeId)

field = 'Campo' FIELD_TYPE '"' multiWordID '"'
attributeReference writeOrReadOnly ';'

button = 'Botao' '"' multiWordID '"' 'Acao' '"' multiWordID '"' ';'

radioButton = 'Radio' '"' multiWordID '"' attributeReference
writeOrReadOnly ';'

checkBox = 'Checkbox' '"' multiWordID '"' attributeReference
writeOrReadOnly ';'

select = 'Select' '"' multiWordID '"' attributeReference
writeOrReadOnly ';'

attributeReferenceList = attributeReference (',' attributeReference)*

multiWordID = (ID)+

writeOrReadOnly = 'Escrita' | 'Leitura'

FIELD_TYPE  = 'Tipo_Texto' | 'Tipo_Senha' | 'Tipo_Arquivo'
ID          = ('a'..'z'|'A'..'Z'|'ç'|'Ç'|'Á'|'á'|'Â'|'â'|'Ã'|'ã'|
              'É'|'ú'|'Ú'|'ê'|'Ê')+
INT         = '0'..'9'+
NEWLINE     = '\r'? '\n'
WHITESPACE  = (' '|'\t')+
```

Listagem 12: Gramática para Descrições de Interface.

Gramática para Cenários de Casos de Uso:

```
scenario = 'Cenario' INT '-' '"' multiWordID '"' ':'
preCondition? stepList

preCondition = 'Pre-Condicao:' preConditionStepList

preConditionStepList = (preConditionStep)+

preConditionStep = '-' systemAction ';'

systemAction = 'Executar' scenarioReference

scenarioReference = multiWordID '.' multiWordID

stepList = (step)+

step = stepId ACTOR ACTION item ';' | stepId multiplePaths

stepId = INT '.'

multiplePaths = 'Ha' INT 'caminhos' ':' pathList

pathList = (path)+

path =  ID ')' ACTOR ACTION item ':' pathStepList

pathStepList = (pathStep)+

pathStep = pathStepAction ';' | pathStepAsserts ';'

pathStepAction = '-' ACTOR ACTION item

pathStepAsserts = '-' ACTOR 'valida' item

attributeReference = '{' attributeid=attributeId '}'

attributeId = multiWordID '.' (multiWordID | attributeId)

attributeReferenceList = attributeReference (',' attributeReference)*

item = 'Link' '"' multiWordID '"' 'sessao' multiWordID |
  'Campo' '"' multiWordID '"' |
  'Select' '"' multiWordID '"' |
  'Botao' '"' multiWordID '"' |
  'Formulario' |
  'Atributos' |
```

```
  'Mensagem' '"' multiWordID '"'

multiWordID = (ID)+

ACTION      = 'clica'|'seleciona'|'preenche'|'inclui'|'apaga'|
              'exibe'|'valida'
ACTOR       = 'Usuario' | 'Sistema'
ID          = ('a'..'z'|'A'..'Z'|'ç'|'Ç'|'Á'|'á'|'Â'|'â'|'Ã'|'ã'|
               'É'|'ú'|'Ú'|'ê'|'Ê')+
INT         = '0'..'9'+
NEWLINE     = '\r'? '\n'
WHITESPACE  = (' '|'\t')+
```

Listagem 13: Gramática para os Cenários de um Caso de Uso.

Apêndice 2 - Estrutura do Código da Ferramenta

Neste apêndice apresentaremos estrutura do código da ferramenta já que a mesma será de código aberto, a disposição para colaboração de qualquer interessado. O *download* do código do projeto pode ser feito através de repositório aberto, provido pelo Google Code, a partir da página: http://code.google.com/p/waatgenerator/ .

A ferramenta foi dividida em projetos menores para melhor organização do código. Cada um destes projetos menores foi então exportado para um arquivo *.jar*, que depois é importado para o *classpath* dos componentes que precisam utilizá-lo. Estes projetos menores também estão divididos em projetos de *plug-ins* para o *Eclipse* e projetos para o funcionamento da ferramenta em si, como interpretadores e geradores.

Os projetos estão divididos da seguinte maneira:

- *br.com.ceut.waat.glossaryeditor*: projeto de *plug-in* para o *Eclipse* contendo o código necessário para gerar o editor especial para glossários da ferramenta, que fornece destaque e *auto complete* de palavras reservadas.
- *br.com.ceut.waat.interfaceeditor*: projeto de *plug-in* para o *Eclipse* contendo o código necessário para gerar o editor especial para descrições de interface da ferramenta, que fornece destaque e *auto complete* de palavras reservadas.
- *br.com.ceut.waat.usecaseeditor*: projeto de *plug-in* para o *Eclipse* contendo o código necessário para gerar o editor especial para cenários de um caso de uso, que fornece destaque e *auto complete* de palavras reservadas.
- *br.com.ceut.waat.menu*: projeto de *plug-in* para o *Eclipse* contendo o código necessário adicionar as funcionalidades da ferramenta na interface da IDE. Possui também todos os métodos de tratamentos de eventos, que serão disparados quando o usuário clicar em um dos botões da ferramenta.
- *WaatGlossaryCore*: projeto contendo o código necessário para a interpretação de um arquivo de glossário. Possui classes responsáveis por: ler estes arquivos e gerar uma representação XML do mesmo; ler um arquivo XML e transformá-lo em um objeto com a estrutura de glossário.
- *WaatInterfaceCore*: projeto contendo o código necessário para a interpretação de um arquivo de descrição de interface. Possui classes responsáveis por: ler estes arquivos e gerar uma representação XML do mesmo; ler um arquivo XML e transformá-lo em um objeto com a estrutura de descrição de interface.

- *WaatScenarioCore*: projeto contendo o código necessário para a interpretação de um arquivo de cenário de caso de uso. Possui classes responsáveis por: ler estes arquivos e gerar uma representação XML do mesmo; ler um arquivo XML e transformá-lo em um objeto com a estrutura de cenário.

- *WaatInterfaceGenerator*: projeto contendo o código necessário para gerar páginas HTML a partir de objetos representando descrições de interface.

- *WaatScenarioGenerator*: projeto contendo o código necessário para gerar testes funcionais para *Selenium* a partir de objetos representando cenários de um caso de uso.

- *WaatGrammars*: projeto contendo o código das gramáticas de cenário, glossário e descrição de interfaces para o ANTLR.

Apêndice 3 - Configurando a Ferramenta

Para a utilização do *plug-in* recomendamos que o mesmo seja instalado na versão 3.5 ou superior do *Eclipse*. Para instalá-lo o usuário deve seguir os seguintes passos:

1. Fazer o *download* do *plug-in* em: http://code.google.com/p/waatgenerator/ ;
2. Selecionar *Help* no menu do *Eclipse*;
3. Em *Help*, selecionar *Install new Software...* ;
4. Uma janela abrirá, nela o usuário deverá selecionar o botão *Add*;
5. O usuário deverá então informar um nome qualquer para o recurso (Ex: *Waat Generator*) e deverá clicar no botão *Archive*;
6. O usuário deverá informar o arquivo que fez *download* pelo site;
7. Após informado o arquivo, o usuário deverá marcar todos os recursos que aparecerão na janela de instalação de *plug-ins*;
8. Depois de selecionados os recursos o usuário deverá proceder com a instalação através dos botões *Next*, e aceitar o termo de uso quando solicitado;
9. No final do processo de instalação o *Eclipse* deverá ser reiniciado.

Após a instalação da ferramenta o usuário também deverá configurar as bibliotecas do *Selenium* para que a mesmo possa funcionar corretamente. Para isso, deverá ser feito o *download* do arquivo *selenium-java-client-driver.jar* que pode ser encontrado no pacote *Selenium RC* a partir da página: http://seleniumhq.org/download/ . Por último, o arquivo deverá ser adicionado ao *classpath* do projeto do usuário.