Universidade de São Paulo Instituto de Matemática e Estatística

MONOGRAFIA

Tema: Estudo e desenvolvimento de novas funcionalidades do Java PathFinder

Orientadora: Ana Cristina Vieira de Melo (MAC – IME)

Aluno: Rodrigo Della Vittoria Duarte
Nusp:3098990

Índice

Capítulo 1: Introdução

Há algumas décadas, os computadores eram grandes máquinas que possuíam um custo de construção e manutenção elevado. Estas características tornavam o uso do computador restrito aos laboratórios de pesquisa de universidades ou de grandes corporações. Com o desenvolvimento de hardware dos últimos anos, foi possível diminuir o custo do computador facilitando assim o acesso da população em geral a esta ferramenta que pode ser útil em diferentes atividades.

Este aumento no uso do computador gerou a necessidade de novos softwares que pudessem ser mais simples de entender e fáceis de utilizar. Se, no início, o uso do computador era restrito a pesquisadores ou a pessoas que já tinham certo nível de conhecimento na área, atualmente o usuário não necessita de nenhum treinamento específico para seu uso.

Hoje em dia, essa demanda por software é ainda maior devido ao surgimento e o avanço da Internet. Com a Internet, podemos utilizar muitos softwares diferentes sem sair de casa[Xav07].Esta busca desenfreada por novos softwares pode causar problemas. Sabemos que o custo da produção de um software é elevado e muito difícil de se quantificar antes do desenvolvimento do software em si[Pre05]. Muito freqüentemente, o custo final do software é muito maior que o custo previsto. Além disso, temos a questão dos prazos, que a cada dia são menores e mais difíceis de se cumprir.

A computação é um campo muito extenso, abrigando várias áreas do desenvolvimento e uso do computador. Uma grande área da computação é a Engenharia de Software. A Engenharia de Software estuda as tecnologias e práticas utilizadas no desenvolvimento do software, desde sua especificação, passando por seu desenvolvimento até sua manutenção. Seus objetivos são o aumento da produtividade no desenvolvimento do software e da sua qualidade final. Logo, a elevação na demanda por software implica na necessidade de aprofundarmos os estudos em Engenharia de Software.

Segundo a Engenharia de Software, o desenvolvimento de um software envolve muitas etapas, e seu custo, obviamente, está dividido entre essas etapas. Uma das etapas que consome mais recursos no desenvolvimento do software é a de testes[Pre05]. Esta área é a responsável pela verificação de um software, ou seja, se o software atende aos requisitos previamente definidos. Ao contrário do que a princípio possa parecer, a etapa de testes deve estar presente durante todo o processo de desenvolvimento do software, pois quanto antes um erro for descoberto mais fácil será corrigi-lo. Essa onipresença aumenta os custos dessa etapa.

A tendência de se tentar diminuir os custos e se enquadrar nos prazos, comumente implica em descuidados com a etapa de testes. Isso pode fazer com que a qualidade do software final seja prejudicada. Com a qualidade prejudicada, não é raro encontrarmos erros no software e, sabemos como pode ser frustrante utilizar um software que não funciona corretamente. Ademais, corrigir um software em produção é muito mais difícil e consome muito mais recursos.

Um erro no software pode fazer com que uma operação incorreta seja executada ou que uma operação correta deixe de ser. Isso, além dos problemas supracitados, pode ocasionar problemas muito mais sérios. A vida de muitas pessoas pode depender da correção de um software. Por exemplo, um software que ajude a pilotar e detectar erros em um avião ou um software usado na medicina. Podemos perceber a catástrofe que seria se um desses software executasse uma operação inválida ou deixasse de executar uma operação. Por isso, a área de testes e validação é de suma importância no desenvolvimento do software.

A Engenharia de Software nos mostra que existem várias estratégias para testarmos um software. Dependendo da natureza do software e das ferramentas utilizadas para seu desenvolvimento, podemos e devemos escolher a melhor estratégia dentre elas[HLK97]. Assim, podemos identificar os erros de uma forma mais rápida, diminuindo o custo.

Estudos indicam que a área de testes exige de 30% a 40% do trabalho realizado no desenvolvimento de um software[Pre05]. Uma forma de se diminuir os custos e aumentar a confiabilidade do software é diminuindo a quantidade de testes a serem realizados, através do uso

de ferramentas que permitam automatizar parte dos testes.

Atualmente, muitos softwares são construídos no paradigma de orientação a objetos e poucos são as estratégias automatizadas de testes para esta classe de software. Além disso, o uso de estruturas de controle para tratamento de exceção está se tornando uma regra no desenvolvimento do software[SH00]. Testes automatizados que levem em conta essas estruturas de controle para tratamento de exceções são ainda mais raros.

Logo, podemos identificar o problema que desejamos enfrentar: diminuir os gastos no desenvolvimento de um software orientado a objetos(mais precisamente escrito em JAVA) com tratamento de exceções através da diminuição dos testes a serem realizados, sem diminuir a confiabilidade do software. Para resolver este problema, pensamos em usar um verificador de modelo para extrair informações acerca do comportamento de software afim de que a partir da análise dessas informações possamos diminuir os casos de testes a serem realizados para validar o software.

Este projeto é parte de um projeto maior. Seu objetivo final é a criação de uma ferramenta, um verificador de modelo, que extraia e organize as informações sobre o comportamento de um software JAVA, para que a análise posterior desses dados possa diminuir os testes realizados.

Já existe um verificador de modelo para aplicativos JAVA, o Java Pathfinder(1.1). Entretanto, esta ferramenta não apresenta os resultados obtidos de uma forma que possamos utilizá-los a fim de diminuirmos os casos de testes. Além disso, esta ferramenta não leva em consideração as estruturas de controle para tratamento de exceções, tão utilizadas atualmente. Assim, estendemos o Java Pathfinder, de tal forma que ele atenda às nossas necessidades.

1.1 Java Pathfinder

O Java Pathfinder é uma ferramenta desenvolvida no centro de pesquisas da NASA. Sua proposta é verificar um programa alvo em JAVA de todas as maneiras possíveis para checar se este viola alguma propriedade. Estas propriedades podem ser muito diversas, como ocorrência de *deadlock*, ou existência de exceções não tratadas. Se, em algum momento durante sua execução, o programa alvo violar alguma propriedade, o Java Pathfinder indica o caminho percorrido até a violação[Xav07].

Desenvolvido como um verificador de modelo(verifica se um modelo preserva uma propriedade), o Java Pathfinder funciona como uma JVM(Java Virtual Machine). JVM é um programa que carrega e executa os aplicativos JAVA, sendo responsável pela conversão dos *bytecodes* em código executável de máquina. Esta característica permite que aplicativos JAVA sejam executados nas mais diversas plataformas, bastando apenas a existência da Java Virtual Machine.

Um conjunto de *bytecodes* é o resultado de uma compilação de um programa JAVA, e esses *bytecodes* são usados pela JVM para executar o programa. Este é o mesmo procedimento usado pelo Java Pathfinder, ou seja, ele trabalha com os *bytecodes* do programa alvo.

Como um verificador de modelo, o Java Pathfinder apresenta uma vantagem em relação ao teste: a simulação do não determinismo. O Java Pathfinder pode retornar passos de sua execução até encontrar estados não explorados do programa alvo. Além disso, quando um estado já visitado é encontrado, o Java Pathfinder pode retroceder até encontrar estados não explorados para, a partir daí, continuar sua verificação.

Todo esse processo é mais eficiente que a execução pura e simples de testes, pois, para atingir o mesmo nível de cobertura de um verificador de modelo, deveríamos construir muitos testes que exigiriam a execução do programa alvo diversas vezes desde o seu início.

Como o Java Pathfinder utiliza-se da construção de estados para realizar sua verificação, um programa alvo que seja muito grande pode gerar uma explosão de estados tornando a ferramenta ineficiente. Para contornar este possível problema, o Java Pathfinder dispõe de alguns mecanismos. Dentre estes mecanismos, destaca-se a possibilidade de configurar a estratégia de busca, onde

alguns filtros são aplicados de tal forma que alguns estados sejam interessantes verificar e outros não. Claramente, este procedimento diminui o número de estados a serem verificados, minimizando o problema.

Além disso, para reduzir o número de estados, o Java Pathfinder ainda recorre a outras estratégias: geradores de escolhas heurísticos que restringem a escolha de possíveis valores para teste de estado; redução de ordem parcial que é utilizada em programas concorrentes e visa analisar somente as operações que afetem outras *threads*; utilização da Java Virtual Machine nativa para executar algumas operações que dessa forma não teriam seu estado rastreado pelo Java Pathfinder.

O Java Pathfinder possui ainda: o VMListener, onde há métodos que são invocados quando há execução de alguma instrução de *bytecode* ou quando há um avanço ou retrocesso de estado; o Model Java Interface com o qual é possível a separação da execução do rastreamento de estados do Java Pathfinder com a execução da Java Virtual Machine nativa. Isto tudo visa proporcionar a extensão do Java Pathfinder. O Java Pathfinder ainda conta com arquivos de propriedades com os quais é possível interagir com o Java Pathfinder, ou seja, é possível utilizar esse arquivos para especificarmos como queremos o funcionamento do Java Pathfinder.

Somando todas essas características, é possível estender o Java Pathfinder para possamos contabilizar os fluxos de objetos visitados e, a partir desses dados, analisar o comportamento do programa alvo.

1.2 Objetivos do Trabalho

O objetivo deste trabalho é produzir uma ferramenta, o Java Pathfinder modificado, que permita extrair um grande conjunto de informação sobre o comportamento do programa alvo. A partir da análise deste conjunto de informação, podemos diminuir a quantidade de testes necessários para a validação do programa alvo.

Com a automatização de boa parte dos testes do programa alvo, podemos ter vários benefícios no desenvolvimento do software: diminuição do custo da fase de testes e por conseqüência diminuição do custo do projeto inteiro, aumento da qualidade do software, aumento da confiabilidade do software, etc.

Além de diminuir a quantidade dos testes necessários, os dados gerados pela nossa ferramenta possibilitam uma grande variedade de análises do comportamento do programa alvo. Com essas análises podemos extrair muitas informações sobre o programa alvo, o que pode ajudar não só a fase de testes, mas todas as outras fases do desenvolvimento. Mais ainda, essas informações podem ser usadas em diferentes campos de estudo da computação.

Capítulo 2: Conceitos Básicos

2.1 Objetos

Programação Orientada a Objetos é um termo muito usado atualmente, mas poucos têm o conhecimento de todos os aspectos que envolvem este conceito. Programação Orientada a Objetos é o nome dado a maneira de se estudar e desenvolver software baseado no comportamento e interação de objetos.

Para entendermos o que é um objeto, primeiro precisamos ter clara a noção de classe. Classe é uma abstração de um conjunto de objetos que possuem características semelhantes. É com a classe que definimos o comportamento e características dos objetos através dos métodos e atributos. Atributos são as características do objeto e métodos são suas habilidades. Assim, o objeto é uma instância da classe. Com os métodos, é possível ao objeto interagir com outros objetos respondendo a mensagens enviadas a ele. Mensagem é uma chamada a um método de um objeto. Tal método vai se comportar de acordo com a especificação da classe a qual pertence. Já os atributos do objeto permitem que o objeto mantenha suas características, desta forma os atributos armazenam o estado do objeto[Dei05].

Para simplificar o que vimos até aqui, vamos ver um exemplo. Podemos ter, em um software qualquer, um classe Carro. Esta classe pode ter como atributos: a cor, o ano e a velocidade. Já seus métodos seriam: `acelera`, `freia`, etc. Assim, poderíamos ter um objeto Ferrari, com a cor vermelha, o ano 2007 e a velocidade 0. Já um outro objeto Porsche teria outros atributos, como a cor preta, o ano 2006 e a velocidade 20. Então os métodos `acelera` e `freia` iriam agir diretamente na velocidade do objeto, a aumentando ou diminuindo respectivamente.

A Programação Orientada a Objetos ainda possui algumas características que a tornam uma ótima escolha na análise e desenvolvimento do software, dentre elas: Herança, Associação, Encapsulamento, Polimorfismo e Modularidade[Dei05].

Através da Herança podemos estender uma classe para outra, reaproveitando seus métodos e atributos, assim podemos diminuir a quantidade de código deixando o projeto mais enxuto e fácil de entender. No nosso exemplo, teríamos uma classe chamada Caminhão, que herdaria os atributos e métodos de Carro e que ainda poderia ter outros atributos (como número de eixos, comprimento, etc.) e outros métodos(como carrega ou descarrega). Outra característica que nos permite diminuir o código e reaproveitar método é a Associação com a qual um objeto pode utilizar recursos de outro objeto.

Além destas características temos ainda o Encapsulamento que permite que separemos os aspectos internos e externos do objeto. Assim, podemos restringir o acesso aos atributos, deixando apenas que os métodos possam manipulá-los. Assim os métodos seriam os aspectos externos e os atributos, os internos. Evitar o acesso direto aos atributos diminui os problemas e erros que o software pode gerar. No exemplo, somente os métodos `acelera` e `freia` poderiam alterar o atributo velocidade, tornando este atributo inacessível a outros objetos do sistema.

Polimorfismo é o mecanismo pelo qual é possível que objetos que estendem a mesma classe possam chamar métodos que possuam a mesma assinatura. Assinatura é o conjunto do nome do método, sua lista de parâmetros e retorno. Assim, podemos ter métodos especializados para cada classe, sendo que a decisão de qual método deve ser executado é feito em tempo de execução. No nosso exemplo poderíamos ter uma classe Ônibus, que como Caminhão herdaria os mesmo atributos e métodos de carro, mas poderíamos ter um método `acelera` especifico para Caminhão e outro para Ônibus.

Temos ainda a Modularidade, pois quando um objeto é criado, ele pode funcionar independentemente do resto do software, podendo ser usado em outros softwares ou substituído por outro objeto. Assim, se em algum momento tivéssemos uma outra classe Carro com, no mínimo, os mesmos métodos e atributos da anterior, mas melhor implementada, poderíamos substituí-la facilmente no sistema.

Com tantas características que facilitam e auxiliam na análise e desenvolvimento de software, já temos uma idéia do porque a Programação Orientada a Objetos é tão largamente utilizada hoje em dia e da importância do objeto no desenvolvimento do software. Muitas linguagens utilizam-se desse conceito, dentre elas: JAVA, C++,Php, Perl, Smalltalk, VB.net, ColdFusion, Python, etc.

2.2 Exceções

Durante a execução de um software, pode ocorrer algum problema que impeça sua continuação. Chamamos este problema de exceção ou condição excepcional. Alguns exemplos de condições excepcionais são[Gak07]:

- Falha na conexão com o banco de dados;
- Acesso a arquivos inexistentes ou inválidos;
- Problemas com operações aritméticas como divisão por zero;
- Tentativa de acesso a um item inválido de uma lista;
- Falha na memória, que pode ser tanto acesso a um lugar restrito da memória ou simplesmente a falta de memória para terminar uma operação;
- Argumentos inválidos em chamadas para métodos.

Tratamento de Exceções é o mecanismo que permite o tratamento das condições excepcionais com a possível continuidade da execução do software. O uso deste tratamento permite que tenhamos um software mais robusto e com melhor tolerância as falhas, além disso, o Tratamento de Exceções tende a aumentar a legibilidade e a confiabilidade do software e a facilitar a sua manutenibilidade[Sun02].

Algumas linguagens não oferecem um mecanismo de tratamento de exceções embutidas na própria linguagem. Com isso, é necessário uma convenção para que haja um mecanismo de tratamento de exceções. Nesses casos, comumente quando uma exceção ocorre, altera-se alguma *flag* ou o método retorna algum valor que indica o problema. Esta maneira de tratar as exceções dificulta o desenvolvimento de software maiores. Os principais problemas encontrados são: diminuição da clareza e legibilidade do software devido à necessidade de checar essas *flags* ou o retorno das funções e a dependência do desenvolvedor em escrever essas verificações[Eck02].

Linguagens mais modernas, como o JAVA, apresentam estruturas para Tratamento de Exceções dentro da própria sintaxe. Isso evita muitos problemas e facilita nos testes e verificação do sistema. O Tratamento de Exceções tem sido cada vez mais utilizado para aumentar a qualidade do software.

1. Categorias de Exceções:

A linguagem JAVA possui duas categorias básicas de Exceções[Eck02]:

- Verificadas ou Checadas(*checked*): são as exceções que são verificadas pelo compilador, ou seja, para cada exceção deste tipo lançada, temos sua captura. Assim, essas exceções podem e devem ser tratadas pelo desenvolvedor;
- Não-Verificadas ou Não-Checadas(*unchecked*): são as exceções que não são verificadas pelo compilador, ou seja, não é necessária a captura deste tipo de exceção. Assim, o desenvolvedor pode ignorar este tipo de exceção.

Agora, podemos definir as principais classes de exceções em JAVA[Sun02]:

- *Exception*: é a classe das exceções checadas, assim, o compilador exige que as exceções dessa classe sejam tratadas;

- *RunTimeException*: nesta classe estão as exceções não-verificadas que representam erros do sistema, assim, o desenvolvedor tem o dever de rever o código e corrigir o problema;
- *Error*: também contém exceções não-verificadas, ou seja, que não necessitam do tratamento do desenvolvedor. Mas esta classe envolve os problemas mais sérios do sistema, que fatalmente levam a finalização do software.

Quando uma exceção(*exception*) ocorre, o método pode tratar a exceção ou, simplesmente lançar a exceção a quem chamou o método. Este método, que recebeu a exceção, não precisa necessariamente tratá-la, podendo somente efetuar um novo lançamento da exceção. Se este procedimento continuar e nenhum método tratar a exceção até o método *main*(), o software é finalizado de modo anormal exibindo a exceção gerada[Gak07].

A exceção também é um objeto, e quando uma exceção ocorre, o fluxo do sistema pode ser transferido até um ponto de interesse do desenvolvedor. Assim, através do lançamento da exceção, o desenvolvedor pode escolher tratar a exceção no momento que achar mais conveniente.

2. Comandos:

Para permitir o lançamento ou o tratamento das exceções, o JAVA fornece quatro elementos, a saber: *throw, try, catch* e *finally*.

O comando *throw* é o responsável pelo lançamento da exceção. Voltando ao nosso exemplo, se quiséssemos que o método `acelera` não tratasse suas possíveis exceções e somente as lançasse para outro método, teríamos:

```
public void acelera() throws ExceptionA, ExceptionB {
// Código do método, que pode gerar exceções do tipo A ou B
}
```

Assim, quando a exceção ocorresse, ela seria lançada ao método que chamou o método `acelera`. Este procedimento, de simplesmente lançar a exceção a outros métodos, não é recomendado pois modificações simples no código podem gerar grandes modificações no sistema todo. Por exemplo, se o método `acelera` sofrer uma alteração que acarrete no possível lançamento de uma ExceptionC, todo método que chama o método `acelera` e trata as exceções geradas por ele deverá ser alterado também.

Para evitar esse problema, podemos tratar a exceção, ao invés de lançá-la a outro método. Para isso utilizamos os elementos *try* e *catch.* O comando *try* especifica um bloco de código que pode gerar exceções. Já o comando *catch* nos permite definir que atitudes tomar quando determinada exceção é lançada. Podemos ter vários blocos *catch* para um mesmo bloco *try*, sendo que cada bloco *catch* fica responsável por um tipo específico de exceção.

O código dentro de um bloco *catch* só é executado se for lançada uma exceção dentro do bloco *try* do mesmo tipo definida no bloco *catch*. Assim, quando essa exceção ocorre, teremos um desvio do fluxo do sistema.

A seguir, voltamos ao exemplo anterior, mas agora não usaremos o comando *throw*, ou seja, não lançaremos a exceção a outro método, cuidando de tratá-la dentro do método `acelera`:

```
public void acelera() {
try{
// Código do método, que pode gerar exceções do tipo A ou B
}
catch(ExceptionA e1){
//Código para tratar a exceção deste tipo
}
catch(ExceptionB e2){
//Código para tratar a exceção do tipo B
```

```
}
```

Agora, suponhamos que é de nosso desejo alterar o código dentro do bloco *try* de tal forma que seja possível que uma exceção de um novo tipo seja lançada. Então, neste momento, só precisamos criar outro bloco *catch*, evitando assim, que essa alteração resulte em alterações em outros métodos:

```
public void acelera() {
try{
// Código do método, que pode gerar exceções do tipo A ou B
}
catch(ExceptionA e1){
//Código para tratar a exceção deste tipo
}
catch(ExceptionB e2){
//Código para tratar a exceção do tipo B
}
catch(ExceptionC e3){
//Código para tratar a exceção do tipo C
}
```

Finalmente, temos o comando *finnaly*, que define um bloco que será executado independentemente da geração ou não de uma exceção. Ou seja, os comandos dentro deste bloco serão executados incondicionalmente. No nosso exemplo, teríamos:

```
public void acelera() {
try{
// Código do método, que pode gerar exceções do tipo A ou B
}
catch(ExceptionA e1){
//Código para tratar a exceção deste tipo
}
catch(ExceptionB e2){
//Código para tratar a exceção do tipo B
}
catch(ExceptionC e3){
//Código para tratar a exceção do tipo C
}
finally{
//Código que será executado independentemente da existência ou não de exceções
}
```

O bloco *finally* é muito importante. Ele realmente é útil quando lidamos com arquivos ou banco de dados, já que permite fechar o arquivo ou terminar a conexão independentemente do correto funcionamento do método.

Programação Orientada a Objetos já é largamente utilizada no desenvolvimento de software e com o uso cada vez maior deste tipo de estrutura de Tratamento de Exceções, verificamos a necessidade de criar mecanismos de testes e validação automáticos para softwares que apresentem essas características.

2.3 Testes no paradigma de Orientação a Objetos

A Engenharia de Software nasceu para garantir uma mecanismo mais sistemático e controlado no desenvolvimento de software. Para isto, cria e estuda técnicas que auxiliam e norteiam todas as etapas do desenvolvimento de software. A Engenharia de Software se divide em algumas áreas, dentre as quais podemos citar: Requisitos de Software, Projeto de Software e Teste e Validação.

O Teste está distribuído durante todo o processo de desenvolvimento de software. Seu objetivo é verificar a existência de erros no software, a fim de que, ao final do desenvolvimento tenhamos uma software que atenda aos requisitos previamente levantados.

Dizemos que um software apresenta um erro, ou que um software falhou, quando ele apresenta um resultado ou comportamento inesperado pelo usuário. Um erro de software pode ocorrer por diversos motivos: falha na especificação, falha no algoritmo, falha na implementação do algoritmo, etc. Assim, podemos perceber que o erro pode surgir devido a qualquer etapa no desenvolvimento do software, tornando necessários os testes em todas as fases, não somente quando uma versão do software já esteja implementada.

Podemos imaginar que a área de testes não seria tão importante se tivéssemos uma equipe de desenvolvedores hábeis e experientes, entretanto, isto só não basta, já que existem sistemas muito grandes e com algoritmos e cálculos muito complexos, de tal forma que a área de testes deve estar separada da área de desenvolvimento. Além disso, antes de chegar ao desenvolvimento em si, o projeto de software já deve ter passado por muitas fases, todas elas devidamente testadas, pois não há desenvolvedor capaz de perceber e corrigir todos erros oriundos de fases anteriores, como a análise de requisitos por exemplo.

A qualidade de um software está intimamente ligada ao cuidado dispensado à área de testes[Pre05]. Podemos perceber que para realizar um teste bem feito durante todo o desenvolvimento do software, desde seus primeiros levantamentos de requisitos até sua manutenção, grande parte do tempo e do custo total do desenvolvimento do software pode ser consumido.

Para diminuirmos o custo e o tempo dispensados à área de testes podemos automatizar essa etapa, ao menos a parte de testes que cabe ao funcionamento do software em si. Quanto mais conseguirmos automatizar os testes, menos precisaremos nos preocupar em escrever manualmente casos de testes para os requisitos levantados. Além disso, com um sistema automático de teste do código, aumentamos a confiabilidade do sistema e liberamos esforço para outras áreas que também exigem esforços no desenvolvimento do software.

Para diferentes paradigmas de desenvolvimento de software, existem diferentes formas de testes. Os testes para sistemas procedurais são diferentes dos testes de sistemas baseados em orientação a objetos. Mesmo se fixarmos um paradigma de desenvolvimento, no nosso caso a Orientação a Objetos, existem várias formas de se testar um software. Além disso, precisamos levar em consideração a utilização do Tratamento de Exceções, que também deve ser testado.

A Engenharia de Software nos oferece vários métodos e estratégias para testar um software. Devemos, a partir da análise e estudo da natureza do software, identificar qual o melhor modelo de testes. O melhor modelo é aquele que, além de verificar corretamente todos os aspectos do software, consome menos recursos e tempo de desenvolvimento[Pre05].

2.3.1 Teste aliado à Verificação Formal

A área de Testes consome muitos recursos no desenvolvimento do software. Isto ocorre pela necessidade de escrevermos testes que cubram todo o software desenvolvido. Entretanto, esta maneira "força bruta" de se testar um software não é a única forma de garantirmos sua correção.

Uma forma de diminuirmos o custo de testes, é diminuir a quantidade de testes que devemos construir manualmente. Para isso, devemos encontrar outra forma de verificar partes do software para as quais não queremos escrever testes. A Verificação Formal nos permite que verifiquemos a correção de pontos do software sem a necessidade de escrevermos testes para estes pontos.

Com partes do software já verificados, podemos escrever testes apenas para os pontos do software não verificados, diminuindo o esforço na realização dos testes sem perder confiabilidade no software final.

Assim, com uma ferramenta automática de verificação de modelos, o Java Pathfinder, podemos extrair informações sobre o comportamento do software, a partir das quais podemos

reduzir os testes a serem realizados, diminuindo o tempo e esforço necessários para se validar o software.

Capítulo 3: Trabalho Realizado

3.1 Ferramentas e tecnologias estudadas

Para realizarmos este projeto, tivemos que aprofundar nossos estudos em algumas ferramentas e tecnologias. Entre essas ferramentas e tecnologias vamos explicar aqui aquelas que foram de maior utilidade durante o projeto.

3.1.1 Java Pathfinder

A primeira ferramenta que precisamos estudar foi o JavaPathfinder. Com o necessidade de implementar novas funcionalidades nesta ferramenta, procuramos conhecer profundamente todas as características relativas ao Java Pathfinder. Estudamos desde os objetivos que levaram ao seu desenvolvimento, passando por sua implementação, até a ferramenta final.

Ao estudarmos o funcionamento do Java Pathfinder, procuramos sempre atentar para todos conceitos utilizados na ferramenta e como utilizar esses conceitos para implementar o que necessitávamos para o desenvolvimento do nosso trabalho. Somente a partir do correto conhecimento da ferramenta é que pudemos estendê-la de tal forma que mantivemos suas características originais agregando novas funcionalidades que ampliam e complementam os resultados obtidos.

O próprio estudo do Java Pathfinder provocou o estudo de outros temas, já que seu desenvolvimento envolveu muitos conceitos e tecnologias diferentes. Assim, o estudo completo desta ferramenta foi o que consumiu maior tempo e exigiu o maior esforço no decorrer do trabalho, entretanto foi de suma importância para o desenvolvimento do projeto e para a obtenção dos resultados.

Com a clara noção das qualidades e deficiências do Java Pathfinder, pudemos dirigir nossos estudos às ferramentas e tecnologias que necessitaríamos para implementar todas as novas funcionalidades de acordo com os objetivos do trabalho.

3.1.2 OcongraX

A OcongraX é uma ferramenta desenvolvida em JAVA, que tem por objetivo auxiliar na execução de testes em sistemas JAVA. Para isso, recebe como entrada um sistema e devolve um grafo de fluxo de controle dos objetos e das exceções desse sistema além de um arquivo XML que contém os objetos e exceções do sistema, juntamente com os pares definição-uso destes elementos[NdM04] .

Com uma interface gráfica bem planejada e de fácil utilização, a OcongraX permite visualizar as estruturas das classes contidas no sistema bem como o grafo gerado pelo seu processamento.

A partir do arquivo XML gerado pela OcongraX, estendemos o Java Pathfinder para verificar quais pares definição-uso são cobertos pela verificação e quais não possuem essa checagem. Assim, para estes últimos é necessário a criação de testes manuais que verifiquem esses pares. Isto diminui a quantidade de testes criados manualmente necessários para a validação do sistema.

O par definição-uso de um objeto informa onde tal objeto é definido e onde é usado. Com esses pares em mãos, diminuímos o trabalho que teríamos para testar um software, concentrando nossos esforços nos testes de todos os pares definição-uso de todos os objetos. Além disso, podemos separar os testes pelas classes contidas no sistema, modularizando o trabalho e permitindo uma visão completa do sistema e de suas classes.

Uma definição de um objeto ocorre quando o construtor do objeto é invocado, quando um atributo deste objeto é alterado ou quando um método que atualize um atributo deste objeto é invocado. O uso de um objeto ocorre quando um de seus atributos tem seu valor utilizado, um método que utilize um atributo deste objeto é invocado ou quando o objeto é passado como

parâmetro para outro método.

Vale salientar que, a lista de pares definição-uso é um subconjunto do produto cartesiano do conjunto das definições pelo conjunto dos usos do objeto. Nem todo uso refere-se a toda definição. Tendo a lista de definições e usos do objeto, a OcongraX cria os pares definição-uso a partir da análise da seqüência de instruções do sistema analisado, fornecendo no arquivo somente os pares de definição-uso válidos do sistema.

Como foi dito anteriormente, uma exceção é um tipo de objeto, portanto pode ter seus pares de definição-uso definidos como um objeto qualquer. Assim, desenvolvemos uma ferramenta, no caso o Java Pathfinder alterado, que verifica não só os objetos contidos no sistema como também as exceções e estrutura do tratamento excepcional do sistema.

3.1.3 XML

É cada vez mais comum que sistemas completamente diferentes tenham que se comunicar entre si. Isto pode tornar-se um problema na medida que a falta de padrão para esta comunicação pode causar perda de dados ou interpretação errada dos dados. No nosso projeto, precisamos ler o arquivo XML, gerado pela OcongraX, que contém os objetos e exceções do sistema juntamente com os pares de definição-uso além de gerar novos arquivos XML.

XML, a abreviação de Extensible Markup Language(linguagem extensível de marcação), é uma linguagem que permite o armazenamento e a troca de dados entre vários sistemas de uma forma simples e rápida, liberando o desenvolvedor de se preocupar com algumas questões envolvendo os dados que ficam a cargo do padrão da linguagem[Tit07]. Na realidade, podemos ver XML como um conjunto de regras ou diretrizes que permitem estruturar os dados em formatos de texto. Seguindo o padrão XML é fácil gerar e recuperar os dados.

Para entender melhor o que é a XML, vamos ver como ela surgiu. Por volta dos anos 70, a IBM criou uma linguagem para armazenar e manipular a imensa quantidade de informação que dispunha sobre diversos temas. Essa linguagem chama-se GML(General Markup Language). A partir daí, a ISO(International Organization for Stardadization, uma entidade que trata de normalizar mecanismos de processos dos mais diversos campos) trabalhou para criar uma normalização da GML, dando origem a SGML(Standard General Markup Language).

Além do desenvolvimento da SGML, durante os anos 90, tivemos o aparecimento da HTML(Hyper Text Language Markup) que surgiu para ser utilizada pela web. Esta linguagem, por ser muito simples, acabou sendo usada muito rapidamente pelos mais diversos setores sem que houvesse uma padronização de sua estrutura.

Assim, para reunir a capacidade de armazenamento e manipulação de dados da SGML e a simplicidade da HTML surgiu a XML. A organização responsável pelo aparecimento da XML foi a W3C(World Wide Web Consortium). A XML, como a HTML, pode ser interpretada pelos *browsers*, mas a rigidez de seu padrão impede alguns problemas da HTML no que se refere ao tratamento da informação[Tit07].

Deste modo, a XML apresenta algumas características muito úteis para armazenar e compartilhar informação. A XML permite a separação do conteúdo da formatação, a criação de *tags* sem limitação, a criação de arquivos para verificação de sua estrutura, etc[Tit07].

Tag é uma estrutura de marcação que possui uma marca de início e outra de fim, utilizada em XML como delimitador de conteúdo. O próprio nome da *tag* pode ser utilizado para informar sobre o tipo de dado mantido por ela ou sobre a estrutura do próprio arquivo XML.

Com tantas vantagens em relação as linguagens existentes, a XML está sendo largamente usada para comunicação entre sistemas, migração de dados e aplicações web. A seguir, temos um exemplo de um arquivo em XML, onde mantemos a informação da classe Carro, bem como dos objetos Ferrari e Porsche e seus atributos, usados em exemplos anteriores:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<title>Monografia</title>
```

```
<type>Exemplo de Xml</type>
<class>
	<classname>Carro</classname>
	<method>acelera</method>
	<method>freia</method>
	<object>
		<name>Ferrari</name>
		<cor>Vermelho</cor>
		<ano>2007</ano>
		<velocidade>0</velocidade>
	</object>
	<object>
		<name>Porsche</name>
		<cor>Preto</cor>
		<ano>2006</ano>
		<velocidade>20</velocidade>
	</object>
</class>
```

3.1.4 SAX e DOM

Para realizar operações com arquivos XML, o W3C especificou dois mecanismos: SAX e DOM. SAX permite fazer um percurso da seqüência de elementos do XML e DOM é utilizado para criação do documento XML na memória, assim podemos executar qualquer tipo de operação que quisermos sobre os dados do arquivo XML. Assim, fizemos uso dessas ferramentas tanto para ler quanto para criar nossos arquivos XML[Tec00].

SAX é a abreviação de Simple API for XML. Originalmente o SAX foi implementado em JAVA, que é o que utilizamos, mas atualmente diversas linguagens já dispões desta API. API (Application Programming Interface) é um conjunto de padrões e rotinas estabelecidos para a utilização das funcionalidades de determinado software.

A grande vantagem de SAX é sua eficiência, já que não permite uma navegação completa pelos elementos do XML. Assim o uso de memória e do processador é minimizado durante a utilização do software. Entretanto, sua utilização requer a implementação de grande parte de código o que dificulta seu uso pelo desenvolvedor[Tec00].

DOM ou Document Object Model, por sua vez permite uma navegação e manipulação total dos dados do XML, para isso utiliza-se de mais memória e processamento que o SAX. Em contra-partida, o código necessário para utilização de seus recursos é muito menor e mais simples. Isso implica num software mais legível e enxuto, o que permite uma melhor manutenibilidade[Tec00].

Assim, através do JAVA, juntamente com o SAX e o DOM, é possível ler arquivos XML, assim como criá-los e manipulá-los de uma forma simples e que permite uma interação com outros sistemas de uma maneira eficiente, minimizando erros provenientes de possíveis problemas decorrentes da falta de padronização na comunicação entre sistemas.

A utilização destas ferramentas, permitiu que a leitura do arquivo proveniente da OcongraX fosse feito, além de que tornou possível a criação dos arquivos XML de saída que contém informação de muita importância sobre o comportamento do programa alvo, assim como de suas classes e estrutura. Além disso, esses arquivos podem ser utilizados por qualquer software de maneira simples, ou podem até serem lidos por pessoas, já que a estrutura do XML permite essa compreensão.

3.2 Atividades realizadas

No desenvolvimento do projeto, muitas atividades foram realizadas, de acordo com os objetivos propostos e com as dificuldades encontradas. O produto final obtido é o Java Pathfinder alterado, capaz de extrair informações do comportamento do software.

3.2.1 Geração dos arquivos com todas as instruções executadas

Como o Java Pathfinder monitora a execução de um programa alvo, nossa primeira preocupação foi alterá-lo para que gerasse um arquivo contendo todas as instruções executadas do programa alvo. Assim, a partir deste arquivo poderíamos verificar quais pontos do programa foram checados e quais pontos ainda necessitariam de testes.

Após essa alteração, analisando o arquivo gerado percebemos que sua compreensão não era trivial. Isso ocorreu pois o Java Pathfinder cria *threads* para auxiliá-lo na execução do programa alvo e essa informação aparecia embaralhada no nosso arquivo de saída. Então, decidimos criar um arquivo de saída para cada *thread* do Java Pathfinder, assim teríamos um pouco mais de organização dos dados referentes a execução do programa.

Ainda assim, percebemos que simplesmente informar a instrução executada do programa alvo pela *thread* no respectivo arquivo de saída não nos fornecia informação suficiente para entendermos o comportamento do Java Pathfinder. Isto porque quando temos várias *threads*, a execução do programa passa de uma *thread* para outra, assim, a ordem das instruções que apareciam nos arquivos não era exatamente a ordem das instruções executadas. Então, decidimos incluir nos arquivos, além das instruções executadas, a ordem em que elas foram executadas, para podermos analisar as instruções executadas na ordem em que foram executadas.

3.2.2 Geração dos arquivos com os pares de definição e uso

Além destes arquivos contendo a execução do programa alvo, gostaríamos de produzir algo mais profundo para auxiliar na etapa de testes do desenvolvimento de um software. Para isso decidimos utilizar o arquivo de saída gerado pela OcongraX bem como todos conceitos já vistos dos pares definição e uso de objetos.

Neste caso, a primeira atividade foi estudar a estrutura e a informação contida no arquivo XML gerado pela OcongraX. A partir daí, implementamos no Java Pathfinder a leitura deste arquivo, deste modo possuímos, em tempo de execução do Java Pathfinder, toda informação referente aos pares de definição e uso dos objetos contidos no programa alvo.

Neste ponto, pensamos em comparar a execução do programa alvo realizada pelo Java Pathfinder com toda a listagem dos pares de definição e uso obtidas através da leitura do arquivo XML gerado pela OcongraX. Como o arquivo XML gerado pela OcongraX contém o número da linha em que foi definido e usado o objeto e não a instrução em si, precisamos verificar se o Java Pathfinder e a OcongraX possuíam a mesma numeração de linha, ou seja se tratam da mesma maneira linhas em branco, linhas de comentário, etc. Assim, teríamos certeza que a linha 'x' do programa alvo para a OcongraX possui a mesma instrução que a linha 'x' do programa alvo para o Java Pathfinder. É evidente o problema que teríamos se as duas ferramentas numerassem as linhas de maneira diferente.

Deste modo, resolvemos gerar dois arquivos XML de saída com basicamente a mesma estrutura do arquivo XML proveniente da Ocongra. Um arquivo contém todos os pares de definição e uso que a execução do Java Pathfinder cobriu e outro arquivo contém a lista dos pares que não foram cobertos, seja por não ter passado pela linha de definição, pela de uso ou por nenhuma das duas linhas. Para isso, não precisamos definir no Java Pathfinder quais linhas continham definição ou uso de objetos, analisando somente se a execução passou pelo número de linha informada pelo arquivo XML da Ocongra.

Agora, com esses arquivos gerados, podemos reduzir o número de testes necessários para se validar o programa alvo, pois precisamos verificar só os pares não cobertos pelo Java Pathfinder.

3.2.3 Atualização e leitura dos arquivos de propriedades

Por fim, utilizamos os arquivos de propriedades característicos do Java Pathfinder para definirmos seu funcionamento para essas novas funcionalidades. Ou seja, através da leitura das propriedades contidas nesses arquivos podemos:

- Optar pela criação dos arquivos de saída com as instruções executadas e sua ordem;
- Informar o nome do arquivo XML de entrada(arquivo gerado pela OcongraX);
- Informar os nomes dos arquivos XML de saída(arquivos com os pares de definição e uso que foram cobertos ou não pela execução do Java Pathfinder).

Vale salientar que os arquivos de saída com as instruções executadas são criados com o nome da *thread* a que essas instruções se referem. Mais ainda, se o arquivo XML de entrada não for informado, os arquivos de saída não conterão informações e, se o nome de algum arquivo XML de saída não for informado, ele não será gerado. Assim podemos, escolher se queremos que o Java Pathfinder gere só o arquivo XML com os pares cobertos, só o arquivo com os pares não cobertos ou os dois.

Para realizar todas essas alterações no Java Pathfinder, procuramos alterar o mínimo possível do código original do Java Pathfinder. Para isso, utilizamos a noção de *listener* já explicada, que nos permitiu interagir com o Java Pathfinder sem alterar suas características iniciais. Ou seja, para termos todas essas inovações basta incluir este *listener* que criamos no Java Pathfinder e atualizar os arquivos de propriedades.

3.3 Resultados e produtos obtidos

Além da própria ferramenta Java Pathfinder alterada, podemos considerar como resultados obtidos os arquivos gerados. Para analisar os arquivos de saída que contém as instruções executadas, vamos ver um pequeno exemplo:

```
public class DeadlockException {
    static Lock lock1;
    static Lock lock2;
    static int  state;

    public static void main(String[] args) {
        lock1 = new Lock();
        lock2 = new Lock();
        Process1 p1 = new Process1();
        Process2 p2 = new Process2();
        p1.start();
        p2.start();
    }

    public static void geraException() throws Exception{
        int valor = (int) (Math.random()*10);
        if(valor%2 == 0)
            throw new Exception();
    }
}

class Process1 extends Thread {
    public void run() {
        DeadlockException.state++;
        synchronized (DeadlockException.lock2) {
            synchronized (DeadlockException.lock1) {
                DeadlockException.state++;
            }
        }
    }
}

class Process2 extends Thread {
```

```
    public void run() {
        DeadlockException.state++;
        try {
            DeadlockException.geraException();
        } catch(Exception e) {
            synchronized (DeadlockException.lock2) {
                synchronized (DeadlockException.lock1) {
                    DeadlockException.state++;
                }
            }
        } finally {
            synchronized (DeadlockException.lock1) {
                synchronized (DeadlockException.lock2) {
                    DeadlockException.state++;
                }
            }
        }
    }
}
```

Para este programa alvo, a execução do Java Pathfinder gera três arquivos de saída:

- main.txt:

```
0         lock1 = new Lock();
1 class Lock {
2         lock1 = new Lock();
3         lock2 = new Lock();
4 class Lock {
5         lock2 = new Lock();
6         Process1 p1 = new Process1();
7 class Process1 extends Thread {
8         Process1 p1 = new Process1();
9         Process2 p2 = new Process2();
10 class Process2 extends Thread {
11         Process2 p2 = new Process2();
12         p1.start();
13         p2.start();
14     }
                        .
                        .
                        .
                        .
```

- Thread-0.txt:

```
15         DeadlockException.state++;
16         synchronized (DeadlockException.lock2) {
17             synchronized (DeadlockException.lock1) {
18                 DeadlockException.state++;
19             synchronized (DeadlockException.lock1) {
20         synchronized (DeadlockException.lock2) {
21     }
43                 DeadlockException.state++;
44             synchronized (DeadlockException.lock1) {
45         synchronized (DeadlockException.lock2) {
46     }
68                 DeadlockException.state++;
69             synchronized (DeadlockException.lock1) {
70         synchronized (DeadlockException.lock2) {
71     }
90                 DeadlockException.state++;
```

```
91              synchronized (DeadlockException.lock1) {
92          synchronized (DeadlockException.lock2) {
93      }
112                 DeadlockException.state++;
113              synchronized (DeadlockException.lock1) {
114          synchronized (DeadlockException.lock2) {
115      }
132                 DeadlockException.state++;
                        .
                        .
                        .
                        .
```

- Thread-1.txt:

```
22          DeadlockException.state++;
23              DeadlockException.geraException();
24          int valor = (int) (Math.random()*10);
25          if(valor%2 == 0)
26              throw new Exception();
27          } catch(Exception e) {
28              synchronized (DeadlockException.lock2) {
29                  synchronized (DeadlockException.lock1) {
30                      DeadlockException.state++;
31                  synchronized (DeadlockException.lock1) {
32              synchronized (DeadlockException.lock2) {
33          }
34          } finally {
35              synchronized (DeadlockException.lock1) {
36                  synchronized (DeadlockException.lock2) {
37                      DeadlockException.state++;
38                  synchronized (DeadlockException.lock2) {
39              synchronized (DeadlockException.lock1) {
40          }
41      }
42          DeadlockException.state++;
47          DeadlockException.state++;
48              DeadlockException.geraException();
                        .
                        .
                        .
                        .
```

Os arquivos Thread-0.txt e Thread-1.txt não foram colocados por completo para não sobrecarregar a monografia.

Outros resultados obtidos com o nosso projeto são os arquivos XML com os pares de definição e uso dos objetos do programa alvo. Para exemplificar esta funcionalidade, vamos imaginar um outro exemplo que tenha como arquivo XML de entrada(gerado pela OcongraX) o indicado abaixo:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<elements>
    <object>
        <classname>name1</classname>
        <name>teste</name>
        <pair>
            <def>5</def>
            <use>11</use>
        </pair>
        <pair>
```

```
            <def>78</def>
            <use>93</use>
        </pair>
        <def>5</def>
        <def>78</def>
        <use>11</use>
        <use>93</use>
    </object>
    <object>
        <classname>name2</classname>
        <name>teste2</name>
        <pair>
            <def>14</def>
            <use>23</use>
        </pair>
        <def>14</def>
        <use>23</use>
    </object>
    <exception>
        <classname>name1</classname>
        <name>teste3</name>
        <pair>
            <def>18</def>
            <use>36</use>
        </pair>
        <pair>
            <def>21</def>
            <use>42</use>
        </pair>
        <def>18</def>
        <def>21</def>
        <use>36</use>
        <use>42</use>
    </exception>
</elements>
```

Após a leitura deste arquivo e a verificação das linhas executadas ou não pelo Java Pathfinder temos como saída:

- Arquivo com os pares cobertos:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<elements>
    <object>
        <classname>name1</classname>
        <name>teste</name>
        <pair>
            <def>5</def>
            <use>11</use>
        </pair>
    </object>
    <object>
        <classname>name2</classname>
        <name>teste2</name>
        <pair>
            <def>14</def>
            <use>23</use>
        </pair>
    </object>
    <exception>
        <classname>name1</classname>
```

```
        <name>teste3</name>
        <pair>
            <def>18</def>
            <use>36</use>
        </pair>
    </exception>
</elements>
```

- Arquivos com os pares não cobertos:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<elements>
    <object>
        <classname>name1</classname>
        <name>teste</name>
        <pair>
            <def>78</def>
            <use>93</use>
        </pair>
    </object>
    <exception>
        <classname>name1</classname>
        <name>teste3</name>
        <pair>
            <def>21</def>
            <use>42</use>
        </pair>
    </exception>
</elements>
```

Estes arquivos agora, fornecem toda informação necessária de uma forma estruturada para uma análise mais profunda sobre quais testes precisam ser feitos para cobrir todo o programa alvo. Sem a utilização de nossa ferramenta, seria necessário escrever testes que cobrissem todos os pares de definição e uso contidos no arquivo XML de saída da OcongraX, para validarmos o programa alvo. Com o uso do Java Pathfinder, diminuímos o número de testes necessários, já que precisamos apenas de testes que cubram os pares de definição e uso não cobertos pela verificação do Java Pathfinder. Como nossa ferramenta divide os pares em dois arquivos XML, um contendo os pares cobertos e o outro os pares não cobertos, precisamos de testes apenas para os pares contidos no arquivo XML de saída que contém os pares não cobertos. Diminuindo assim a quantidade de testes necessários para a validação do programa alvo.

3.4 Conclusões

A área de Testes e Validação é de extrema importância no desenvolvimento de software. Uma forma de diminuir os custos e aumentar a qualidade do software é automatizando os testes. É cada vez mais comum o desenvolvimento de software com tratamento de exceções, apesar disso são raras as formas automáticas de se testar este tipo de software.

O Java Pathfinder, uma ferramenta desenvolvida pela NASA, tem por objetivo verificar se um programa alvo satisfaz ou não um conjunto de propriedades. Através da implementação de novas funcionalidades no Java Pathfinder e com o uso de um arquivo de saída da OcongraX, foi possível gerar listagens da execução do programa alvo além de arquivos XML que contêm todos os pares de definição e uso dos objetos do programa alvo, separados em pares que foram verificados pelo Java Pathfinder e pares que não foram verificados.

Com esses arquivos, é possível fazer um trabalho que diminua os casos de testes necessários para a validação do programa alvo, já que apenas precisamos de testes que verifiquem os pares de definição e uso não verificados pelo Java Pathfinder. Assim, com menos testes a serem realizados, podemos diminuir o tempo gasto nesta fase e, por conseqüência o custo, além de que aumentamos a

confiabilidade do software produzido uma vez que com a automatização de boa parte dos testes, diminuímos o número de erros provenientes da falha humana na elaboração ou execução dos testes.

Desta forma, elaboramos uma ferramenta que permita automatizar os testes de programas JAVA, atingindo assim o objetivo proposto para nosso trabalho. Além disso, com todas as informações geradas pelo Java Pathfinder acerca do comportamento do software, é possível a realização de vários outros tipos de análise sobre o software.

Bibliografia

[Bin00] Robert Binder. Testing Object-Oriented Systems: Models, patterns and tools. Addison-Wesley, 2000. [Blo01] Joshua Bloch. Effective Java: Programming Language Guide. Addison Wesley, 2001.

[DDH72] O. J. Dahl, E. Dijkstra, and C. A. R. Hoare. Structured programming. New York:Academic Press, 1972. 48

[Dei05] Deitel Harvey M., Deitel Paul J. Java Como Programar. Prentice Hall Nacional, 6a. Edição, 2005

[DMRR00] David D´eharbe, Anamaria Martins Moreira, Leila Ribeiro, and Vanderlei Moraes Rodrigues. Introdução a métodos formais: Especificação, semântica e verificação de sistemas concorrentes. RITA, 7(1):7–48, 2000.

[Eck02] Bruce Eckel. Thinking in Java. Prentice-Hall, 3rd edition, 2002.

[Gak07] Luciana Setsuko Gakiya. Classificação e Busca de Componentes com Tratamento de Exceções. Projeto de Dissertação de Mestrado, 2007.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. Addison-Wesley, 3rd edition, 2005.

[HLK97] Pei Hsia, Xiaolin Li, and David C. Kung. Augmenting data flow criteria for class testing. In Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative research, 1997.

[Int06] International Software Testing Qualifications Board. Standard Glossary of terms used in Software Testing, 2006.

[NdM04] Paulo Roberto Nunes and Ana C. V. de Melo. Ocongra – uma ferramenta para geração de grafos de controle de fluxo de objetos. In Jaelson F. B. de Castro, editor, Anais do 18o. Simpósio Brasileiro de Engenharia de Software - SBBD-SBES, Brasília-Distrito Federal, Brasil, oct 2004.

[Pre05] Roger S. Pressman. Software Engineering: A practitioner's approach. McGraw-Hill, 6th edition, 2005.

[Pru04] Leandro Cesar Prudente. Um estudo sobre teste versus verificação formal de programas java. Master's thesis, IME-USP, Março 2004.

[SH00] S. Sinha and M. J. Harrold. Analysys and testing of programs with exception handling constructs. IEEE Transactions on Software Engineering, 26(9), September 2000. 49

[Sun02] SUN MICROSYSTEMS PRESS - Java Programming Language SL-275 – California USA, 2002.

[Tec00] Technacal Articles and Tips, disponível na Internet em http://java.sun.com/developer/TechTips/2000/tt0627.html, 2000.

[Tit07] Ed Tittel. XML. Bookman, 1a. edição, 2007.

[Xav07] Kleber da Silva Xavier. Estudo sobre Redução do Custo de Testes através da utilização de Verificação de Componentes Java com Tratamento de Exceções. Qualificação de Mestrado, 2007.

Capítulo 4: Parte Subjetiva

4.1 Desafios e frustrações encontrados

Ao desenvolvermos um trabalho de formatura supervisionado, freqüentemente nos deparamos com imprevistos que se tornam verdadeiros desafios e situações que nos levam a alguma frustração. Somente isto já nos garante um aprendizado de grande valia em nosso futuro, talvez tão valioso quanto o aprendizado dos conceitos e tecnologias utilizados no decorrer do projeto.

Este trabalho baseia-se em implementar melhorias e novas funcionalidades a um software já existente. Isto significa que não participei da concepção e desenvolvimento da ferramenta inicial. Esta característica me levou ao maior desafio do trabalho: compreender perfeitamente todos os aspectos da ferramenta, desde o código em si até os conceitos e tecnologias aplicadas para seu desenvolvimento. Assim gastei um tempo considerável no entendimento da ferramenta.

Esta situação ocasionou o primeiro sentimento de frustração, pois enquanto meus colegas já tinham alguns resultados em seus trabalhos, eu continuava, a primeira vista, no mesmo ponto inicial.

Um outro desafio encontrado decorre do tempo disponível para a realização do trabalho. Como surgiram muitas idéias para melhorar a ferramenta, nem todas puderam ser implementadas a tempo, então o desafio foi escolher as melhores ou mais importantes idéias e, certas vezes, até unir duas idéias para que efetivamente pudesse produzir algo relevante.

Uma frustração inerente a natureza do trabalho é que, de certo modo, o uso da ferramenta é restrita a aqueles que detém algum conhecimento em Engenharia de Software ou que trabalham com estas tecnologias. Talvez fosse mais gratificante se a ferramenta final desenvolvida fosse algo de uso cotidiano ao usuário comum.

4.2 Lista das disciplinas mais relevantes para o trabalho

Certamente, todas disciplinas oferecidas dão sua contribuição para a formação do aluno. Dentre as disciplinas cursadas destaquei aquelas que julgo mais importantes ao longo do desenvolvimento do trabalho:

- MAC0332 – Engenharia de Software

  Com esta disciplina pude entrar em contato com os conceitos de Engenharia de Software, bem como identificar a grande importância da Engenharia de Software na computação em geral.

- MAC0340 – Laboratório de Engenharia de Software

  Nesta disciplina aprendi como funciona a Engenharia de Software na prática e percebi a importância dos testes no desenvolvimento do software.

- MAC0211 – Laboratório de Programação I

  Laboratório de Programação I ensinou a trabalhar em projetos maiores daqueles propostos na grande maioria das disciplinas da grade.

- MAC0242 – Laboratório de Programação II

  Esta disciplina ofereceu a oportunidade de conhecer a orientação a objetos e trabalhar com JAVA.

- MAC0442 – Programação Orientada a Objetos

  Nesta disciplina conheci os conceitos mais profundos de orientação a objetos que foram muito úteis no desenvolvimento do trabalho.

- MAC0338 – Análise de Algoritmos

É com esta disciplina que aprendi a avaliar a qualidade de um algoritmo. Com os conceitos aqui estudados pude melhorar a ferramenta desenvolvida no trabalho.

4.3 Observações sobre a aplicação de conceitos estudados nas disciplinas do curso

O conceito estudado para o desenvolvimento do projeto que seria muito útil em diversas disciplinas do curso é o XML. Como permite a manutenção de informação de uma maneira simples e eficiente, o XML poderia ser utilizado em muitas disciplinas em que projetos maiores precisam ser criados. Principalmente naqueles projetos em que cada pessoa ou grupo fica responsável por uma parte do projeto e é necessário a comunicação entre software de diversos grupos diferentes.

Poderíamos usar o XML ainda para criar uma padronização dos resultados de todos os projetos em todas as disciplinas, assim o aluno poderia manter de forma simples e eficiente um certo histórico de tudo que produziu no período, além do que facilitaria o aprendizado do aluno, pois este teria mais tempo para se preocupar com o desenvolvimento do projeto em si e em aprender os conceitos envolvidos, em vez de se preocupar com o formato dos relatórios finais a serem entregues junto com o código desenvolvido.

De mesmo modo, se todo arquivo de entrada fosse em XML, muito do tempo gasto para leitura e validação dos dados poderia ser realocado na resolução do problema proposto, permitindo uma maior compreensão do aluno em relação ao tópico estudado. Assim, com o uso do XML, fica evidente que o uso do SAX e do DOM seriam necessários para ajudar nessa manipulação dos arquivos XML.

4.4 Futuro

Tendo em vista a conclusão do curso, o próximo e natural passo a ser dado para continuar a atuar nessa área seria o início da pós-graduação. Assim, poderia ampliar e focar os estudos em Engenharia de Software de tal forma que o que hoje parece um desafio se torne em pouco tempo em realização.

Computação é uma área em que a novidade é uma constante. Além disso, a computação se divide em várias outras áreas que podem ser muito diferentes entre si. Engenharia de Software, em particular é uma dessas áreas e uma das maiores, senão a maior. Portanto, para avançar nesses estudos é necessário muita dedicação e muito tempo, o que a pós-graduação pode começar a oferecer. Obviamente, que os estudos não terminariam na pós-graduação, mas seria uma ótima opção a seguir neste momento.

Além disso, é cada vez mais recorrente que as empresas de tecnologia, cientes da importância da Engenharia de Software, invistam fortemente neste setor. Então, é possível continuar atuando nesta área na iniciativa privada.

Pensando exclusivamente no projeto, um possível caminho a seguir seria melhorar a interface do Java Pathfinder, pois apesar de ser um ferramenta incrível, sua interface deixa a desejar. Esta dificuldade ao se trabalhar com o Java Pathfinder pode ser um empecilho para sua larga utilização. Ainda, poderia pensar em ampliar a área de aplicação do Java Pathfinder para outras linguagens. Obviamente isso acarretaria em muitas mudanças na ferramenta, o que exigiria muito tempo e esforço de desenvolvimento.