Simone Hanazumi

Ferramenta para Análise do Tratamento Excepcional de Objetos

São Paulo
2007

Simone Hanazumi

Ferramenta para Análise do Tratamento Excepcional de Objetos

Trabalho de Formatura Supervisionado apresentado para obtenção do título de Bacharel em Ciência da Computação pela Universidade de São Paulo.

Orientadora: Profa. Dra. Ana Cristina Vieira de Melo

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

São Paulo
2007

Agradecimentos

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), à Pró-Reitoria de Pesquisa da Universidade de São Paulo (PRP-USP) e ao Intituto de Matemática e Estatística da Universidade de São Paulo (IME-USP) pela Bolsa PIBIC Institucional[1] concedida para a realização deste trabalho.

À Profa. Dra. Ana Cristina Vieira de Melo, pela paciência e pelas valiosas orientações feitas para o andamento e conclusão do projeto.

A Paulo Roberto de Araújo França Nunes, Rodrigo Della Vittoria Duarte e Kleber da Silva Xavier, por suas colaborações, críticas e sugestões que ajudaram no desenvolvimento do trabalho.

Aos professores do IME-USP, cujos ensinamentos dados durante a graduação foram de grande importância para a realização deste trabalho.

Aos meus amigos, aos meus pais e à minha irmã, por seu apoio e incentivos constantes.

Por fim, agradeço a Deus por ter me dado forças para realizar e concluir este trabalho.

[1]Vigência: novembro de 2007 a outubro de 2008

Resumo

O trabalho desenvolvido é uma Iniciação Científica na área de *Engenharia de Software*. Ele consiste no desenvolvimento da OConGraX, software que estende uma ferramenta (OConGra [11, 12]) de modo que ela, ao analisar um programa escrito na linguagem Java [22], devolva o *OCFG* (*Object Control-Flow Graph*: Grafo de Fluxo de Controle de Objetos) correspondente com informações adicionais sobre os mecanismos de tratamentos de exceções empregados.

Um *OCFG* [2] é uma representação das classes, objetos e relacionamentos de um sistema. Os mecanismos para tratamentos de exceções [7, 17], por sua vez, são métodos utilizados para tratar uma ocorrência anormal no comportamento do programa. Ambos fornecem informações importantes para a análise comportamental de um sistema escrito em linguagem orientada a objetos, o que possibilita tornar o processo de testes mais ágil e eficiente.

Palavras-chave: testes estruturais, tratamento de exceções, análise do programa.

Sumário

Capítulo 1

Introdução

Testes são utilizados no desenvolvimento de um sistema para assegurar a qualidade do software [14]. Muitas das técnicas existentes para este fim utilizam-se da informação do fluxo de controle e execução do programa para melhor analisar o software e determinar, então, os tipos de testes a serem realizados.

Para essa análise, em especial nas linguagens OO (orientadas a objeto), devem ser utilizados critérios de teste [26, 20] que levem em consideração os efeitos das exceções [13, 4] e mecanismos de tratamentos de exceção [7, 17]. Estes mecanismos, além de aparecerem com grande freqüência na prática [21] por facilitarem o desenvolvimento de sistemas menos propensos a erros [1, 16], podem alterar completamente o comportamento de um programa em execução. Por isso, as exceções e os mecanimos utilizados para tratá-las devem ser testados, verificando assim como o sistema reage em situações excepcionais de execução.

Ademais, para auxiliar esse processo, devem ser construídas representações do fluxo de controle de programas com tratamento de exceção. Estas representações permitem uma melhor visualização da estrutura do sistema, facilitando a coleta de dados para uma seleção adequada de testes. Uma boa representação é o *CFG* (*Control-Flow Graph*: Grafo de fluxo de controle) [19], que permite visualizar os caminhos a serem percorridos pelas exceções durante a execução de um programa, e os locais onde elas são ou não tratadas. Uma variante deste grafo, utilizada para análise de programas OO, é o *OCFG* (*Object Control-Flow Graph*: Grafo de fluxo de controle de objeto) [2] que representa as classes, objetos e relacionamentos do sistema, mas não oferece a visualização referente ao tratamento de exceções.

Para possibilitar uma análise mais completa de um programa escrito em linguagem OO, seria necessário juntar as informações obtidas dos seus respectivos *CFG* e *OCFG*. Entretanto, a construção de tais representações é uma tarefa trabalhosa e bastante cara. Portanto, uma ferramenta que automatize tal procedimento facilitaria o processo de teste e manutenção do software.

1.1 Objetivos do Trabalho

Este trabalho teve como objetivo desenvolver um software, a OConGraX, que estende uma ferramenta de geração de um *OCFG*, a OConGra [11, 12]. A OConGraX recebe, por meio de uma interface gráfica, um software escrito na linguagem Java [22]. Em seguida, passa a analisá-lo de modo a devolver:

- Informações sobre o fluxo de objetos e exceções do programa (definição e uso dos objetos e exceções presentes no sistema);

- *OCFG* com os vértices e arestas adicionais provenientes da análise dos mecanismos de tratamento de exceção do software.

Com o uso da ferramenta desenvolvida neste trabalho, os dados necessários para a aplicação de critérios de testes [20] na análise de um sistema escrito em Java serão obtidos de forma rápida e eficiente. Além disso, a automatização dos procedimentos aqui citados representam uma diminuição do custo total de desenvolvimento de um software [26].

1.2 Organização dos Capítulos

Os capítulos estão organizados da seguinte maneira:

- **Parte Técnica:**
  - **Capítulo 2 - Conceitos Preliminares:** contém uma breve explicação da teoria necessária para o entendimento do trabalho.
  - **Capítulo 3 - OConGra:** descrição da ferramenta, cuja extensão foi realizada neste trabalho.
  - **Capítulo 4 - OConGraX:** descrição do trabalho realizado.
  - **Capítulo 5 - Conclusão:** apresentação dos resultados obtidos.
- **Parte Subjetiva:**
  - **Capítulo 6 - Desafios, Aprendizado e Planos futuros:** contém um relato da experiência que obtive na realização do trabalho e de como o conhecimento obtido nas disciplinas da graduação foi utilizado.

I. Parte Técnica

Capítulo 2

Conceitos Preliminares

Neste capítulo, serão apresentados os conceitos teóricos necessários para um melhor entendimento do trabalho aqui desenvolvido. Toda a teoria apresentada foi escrita baseada na linguagem Java [22].

2.1 Conceitos Básicos

Nesta seção, serão apresentados os conceitos introdutórios, importantes para compreender todo a motivação e desenvolvimento do projeto.

2.1.1 Objetos

Segundo Deitel e Deitel [3], podemos conceituar **objetos** como sendo componentes de software que modelam itens do mundo real de acordo com seus atributos e comportamentos (e.g., se considerarmos como objeto a "bola", ela poderá ter como atributos sua cor, forma e tamanho, e como comportamento, o fato de rolar).

Figura 2.1: Composição de um Sistema Java

Ainda segundo eles, os objetos com características em comum podem ser representados por uma **classe** (e.g., uma classe "Bola" poderia ser utilizada para representar os objetos "bola azul" e "bola vermelha"). As classes são, a grosso modo, os arquivos `.java` que compõem um sistema. Elas contém, além de seus objetos, a implementação de métodos. Estes métodos são os responsáveis por realizar tarefas sobre os objetos definidos no conjunto de classes do sistema.

***

Java [22] é uma linguagem orientada a objetos. Isto significa que os programadores que fazem uso desta linguagem concentram-se em criar classes e componentes, sendo que cada classe possui um conjunto de dados e métodos que manipulam esses dados. Os objetos, como descritos anteriormente, são instâncias da classe.

O ato de programar em linguagens OO recebe o nome de Programação Orientada a Objetos (POO). Realizar esse tipo de programação traz facilidades para implementação do projeto, uma vez que permite a reutilização de código e o uso de conceitos como encapsulamento de dados, herança e polimorfismo. Mais detalhes sobre a POO podem ser obtidas em [3, 23].

2.1.2 Exceções

As **exceções** [4, 6, 23, 26] são estruturas utilizadas para descrever e tratar uma situação anormal (evento excepcional) que ocorre durante a execução de um programa. No momento em que ocorre uma exceção, o fluxo de execução do aplicativo é transferido do local onde surgiu a exceção (lançamento da exceção) até um ponto que pode ser definido pelo programador (captura da exceção) [8], para que ela possa ser tratada. Esse tratamento consiste, em geral, num conjunto de instruções que permite ao programa recuperar-se da situação excepcional.

Apesar de erros e exceções dificilmente acontecerem durante a execução de programas, códigos que permitem a recuperação do sistema nessas ocorrências são bastante freqüentes, sobretudo em linguagens como Java, que possuem estruturas específicas para isso [19]. Isto pode ser explicado devido ao fato da utilização correta e consciente destes mecanismos fazer com que o programa seja mais robusto e tolerante a falhas, o que é essencial no mundo atual.

Tipos de Exceções

Na linguagem Java, as exceções podem ser classificadas [23, 26] em:

- **Síncronas**: ocorrem em pontos bem determinados do programa e são geradas por meio da avaliação de expressões, chamadas a métodos ou por meio de um comando *throw*.

  - Verificadas (*checked*): são verificadas pelo compilador[1]. Isto significa que o compilador garante que para cada exceção lançada existe uma estrutura que faz a sua captura.

  - Não Verificadas (*unchecked*): não são verificadas pelo compilador. Isto significa que estas exceções podem ser lançadas no código, sem que seja obrigatória a criação de uma estrutura para sua captura.

  - Implícitas: lançadas por uma chamada a uma sub-rotina ou pelo ambiente de execução.

  - Explícitas: lançadas pelo usuário por meio de um comando *throw*.

- **Assíncronas**: ocorrem em pontos não determinísticos do programa, geradas pela JVM[2] ou por outras *threads* em processos concorrentes por meio de um comando *stop()*.

[1] Compilador é o responsável por transformar o código Java em *bytecodes*. Estes *bytecodes* serão interpretados posteriormente pela máquina para executar o programa

[2] JVM é a abreviação de *Java Virtual Machine* ou Máquina virtual Java. A JVM é a responsável por interpretar os *bytecodes* (arquivos `.class` gerados na compilação dos arquivos `.java`) e os traduzir para um código executável pelo computador

Estruturas para Tratamento de Exceções

```
 1: try {
 2:  //bloco de código com ocorrência de exceção
 3: }
 4: catch (ExceptionType1 e1) {
 5:  //tratamento da exceção do tipo ExceptionType1
 6: }
 7: catch (ExceptionType2 e2) {
 8:  //tratamento da exceção do tipo ExceptionType2
 9: }
10: catch (Exception e3) {
11:  //tratamento de todos os tipos de exceções
12: }
13: finally {
14:  //código executado ao final do bloco try
15:  //ou ao final da seqüência de blocos try-catch,
16:  //independente da ocorrência de exceções em try.
17: }
```

Figura 2.2: Sintaxe da construção de tratamento de exceções em Java, baseado na figura do artigo de Sinha e Harrold [19]

Em Java, há quatro elementos principais utilizados para tratar exceções: *try*, *catch*, *throw* e *finally*.

Try define um bloco de código em que uma ocorrência excepcional pode acontecer. Este bloco pode conter um comando *throw*, e vir acompanhado de um conjunto de blocos definidos pelo elemento *catch* e de um bloco definido por *finally*.

Throw é utilizado para lançar uma exceção de forma explícita[3]. *Catch*, por sua vez, é um elemento que recebe como parâmetro um único tipo de exceção e define um bloco que a trata. Esta exceção é, em geral, lançada dentro de um bloco *try*, que pode estar no mesmo método ou não do bloco *catch*.

O bloco definido por *finally* é opcional, e caso exista, é o último elemento da seqüência de blocos *try-catch* ou vem logo após um bloco *try*. O código descrito nele (em geral, realizando liberação de recursos adquiridos no bloco *try* correspondente) é executado obrigatoriamente, independente da execução do programa ter tido uma ocorrência excepcional ou não. No caso do bloco *try* ser finalizado com um comando de transferência de controle (*return*, *break* ou *continue*), o bloco *finally* (se houver algum) sempre será executado.

Informações adicionais sobre estas estruturas podem ser obtidas em [3, 23].

2.1.3 Grafos

Grafos [18] são estruturas de dados compostas por vértices (ou nós) e arestas. São utilizadas para representar as informações e a maneira como elas se relacionam entre si (e.g., um grafo pode representar um mapa, sendo os vértices as cidades e as arestas, as estradas que fazem a ligação entre elas.).

Neste trabalho, serão utilizados os chamados grafos orientados ou dirigidos. Estes grafos possuem arestas que são representadas por setas, e indicam uma determinada direção a ser seguida de um vértice a outro. Para efeito de simplificação, utilizaremos apenas a palavra grafo para nos referirmos aos grafos orientados.

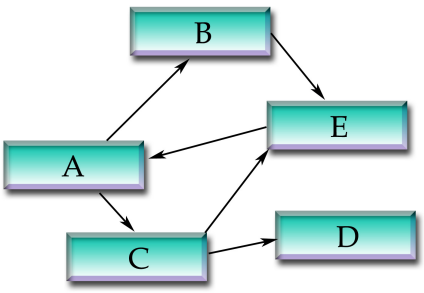


Figura 2.3: Exemplo de grafo orientado com 5 vértices e 6 arestas

[3] *Throw* pode ser utilizado também para lançar um método. Como este caso é pouco comum, ele não foi considerado neste trabalho

2.2 Testes

Testes [14] são utilizados para verificar se o programa está funcionando adequadamente. Eles são realizados durante o desenvolvimento de um software para assegurar sua corretude e robustez.

Há vários tipos de testes, sendo os principais:

- **Testes Funcionais (*Black-Box*)**: realizados para verificar se uma funcionalidade específica do programa é executada de forma correta.
- **Testes Estruturais (*White-Box*)**: relizados para verificar se o programa funciona corretamente analisando seu código e estrutura. São constituídos de três etapas:

  1. Montagem de um grafo de fluxo de controle baseado no código (estrutura) do programa. Este grafo de fluxo de controle é uma representação do funcionamento de um sistema, sendo os vértices as linhas com os comandos Java utilizados e as arestas os possíveis caminhos a serem percorridos durante a execução do programa.
  2. Obtenção dos pares def-uso (definição-uso) de objetos e exceções. Um par def-uso pode ser representado por **(a, b)**, em que **a** é a linha em que um objeto ou exceção é definido, e **b**, a linha em que um objeto ou exceção definido em **a** é usado no programa.
  3. Determinação dos caminhos a serem percorridos pelos testes, utilizando-se dos dados obtidos anteriormente. Em geral, estes caminhos são decididos de modo que todos os pares def-uso sejam exercitados pelo menos uma vez[4]. Isso garante que toda a estrutura do sistema seja testada.

Como pode-se perceber, a ferramenta desenvolvida neste trabalho (OConGraX) tem seu uso atrelado ao teste estrutural, por automatizar as primeiras duas etapas deste tipo de teste. Por isso, mais informações sobre as definições e usos de objetos e exceções e sobre o grafo de fluxo de controle serão dados a seguir. Informações adicionais sobre testes podem ser obtidos em [14].

2.3 Definição e Uso

Nesta seção serão explicados os conceitos de definição e uso de objetos e de exceções. Estes conceitos são de grande importância para a aplicação de critérios de teste [2, 20], e foram utilizados na implementação da OConGra [11, 12] e da ferramenta desenvolvida neste trabalho, a OConGraX.

2.3.1 Definição e Uso de Objetos

De acordo com Chen e Kao [2], podemos afirmar que os objetos são definidos e usados nas seguintes situações:

[4]Outros critérios podem ser utilizados para a determinação dos caminhos a serem testados, como os apresentados em [2, 20, 26]

Objeto definido: um objeto é dito definido quando o seu estado é iniciado ou alterado, ou seja, se pelo menos uma das seguintes condições for válida:

1. O construtor do objeto é invocado;
2. Um atributo é definido (tem seu valor alterado);
3. Um método que inicializa ou modifica o(s) atributo(s) é invocado.

Objeto usado: um objeto é dito usado quando pelo menos uma das seguintes condições for válida:

1. Um de seus atributos tem seu valor utilizado;
2. Um método que utiliza o valor de um de seus atributos é invocado;
3. O objeto é passado como parâmetro.

Este conceito de definição e uso de objetos foi utilizado na implementação da OConGra [11, 12]. No capítulo 3 desta monografia serão dados maiores detalhes sobre esta ferramenta.

Exemplo

```
 1: public class Bola {
 2:   int raio;
 3:   Color cor;
 4:
 5:   public Bola(int raio, Color cor) {
 6:     this.raio = raio;
 7:     this.cor = cor;
 8:   }
 9:
10:   public int getRaio() {
11:     return raio;
12:   }
13:
14:   public void setRaio(int raio) {
15:     this.raio = raio;
16:   }
17: }
```

```
1: public class Desenho {
2:
3:   public void desenhaBola(Bola bola, int x, int y) {
4:     //instruções para realização do desenho da bola
5:     //na posição (x,y).
6:   }
7: }
```

```
 1: public class Main {
 2:   static Bola b;
 3:
 4:   public static void main() {
 5:     b = new Bola(new Random().nextInt(), Color.red);
 6:     Desenho d = new Desenho();
 7:
 8:     if (b.raio() > 5) {
 9:       int aux = b.getRaio();
10:       b.setRaio((int)aux/2);
11:     }
12:     d.desenhaBola(b, 50, 100);
13:   }
14: }
```

Figura 2.4: Código Java (meramente ilustrativo) com definições e usos de objetos

Para exemplificar as situações em que um objeto é definido e/ou usado, utilizamos o código acima. A seguir, apresentamos as situações em que o objeto `b`, instância da classe `Bola`, é definido ou usado:

- Objeto `b` definido:
  - Caso 1: classe `Main`, linha 5;
  - Caso 2: classe `Bola`, linhas 6, 7 e 15;
  - Caso 3: classe `Main`, linha 10.
- Objeto `b` usado:
  - Caso 1: classe `Main`, linha 8;
  - Caso 2: classe `Main`, linha 9 (chamada ao método `getRaio()`);
  - Caso 3: classe `Main`, linha 12.

2.3.2 Definição e Uso de Exceções

O conceito de definição e uso de exceções que será mostrado aqui é uma simplificação do apresentado por Sinha e Harrold [20]:

Exceção definida: uma exceção é definida quando:

1. Um valor é atribuído a uma variável de exceção[5];
2. Uma variável de exceção é declarada como parâmetro de *catch*;
3. É lançada pelo comando *throw*.

Exceção usada: uma exceção é usada quando:

1. O valor de uma variável de exceção é acessado;
2. Uma variável de exceção é declarada como parâmetro de *catch*;
3. É lançada pelo comando *throw*.

Outro conceito importante, referente às exceções, é o de **par de definição-uso**. Segundo Sinha e Harrold [20], estes pares ocorrem quando:

1. Um valor é atribuído a uma variável de exceção (definição) e, posteriormente, a variável é lançada pelo comando *throw* (uso);
2. Num bloco *catch*, a variável de exceção declarada como parâmetro (definição) tem seu valor acessado nas instruções para seu tratamento (uso);
3. Uma variável de exceção lançada por um comando *throw* (definição) é pega e tratada num bloco *catch* (uso). Neste caso, o par só é formado se existe um caminho de execução que vai direto de *throw* até *catch*.

Tanto o conceito de definição e uso, quanto o de par de definição-uso de exceções, foi utilizado na implementação da OConGraX. Detalhes sobre esta ferramenta serão dados no capítulo 4.

[5]Uma variável de exceção é uma variável de programa cujo tipo declarado é um tipo de exceção. Um tipo de exceção, por sua vez, é uma classe de exceção (como, por exemplo, `IOException`). Em Java, a classe `Exception` é a superclasse de todas as demais classes de exceção, por isso, ao fazer `catch(Exception e)`, todas as exceções que chegam a esta parte do código são capturadas neste bloco

Exemplo

```
 1: public class C1 {
 2:   public void m1() {
 3:     Exception u = new Exception();
 4:
 5:     try {
 6:       throw u;
 7:     } catch (Exception e) {
 8:       System.out.println(e);
 9:     }
10:   }
11: }
```

Figura 2.5: Código Java (meramente ilustrativo) com definições e usos de exceções

No código acima, temos as seguintes situações em que as exceções são definidas e/ou usadas:

- Exceção definida:
  - Caso 1: classe `C1`, linha 3;
  - Caso 2: classe `C1`, linha 7;
  - Caso 3: classe `C1`, linha 6.
- Exceção usada:
  - Caso 1: classe `C1`, linha 8;
  - Caso 2: classe `C1`, linha 7;
  - Caso 3: classe `C1`, linha 6.

Também temos os seguintes pares de definição-uso de exceções:

- Caso 1: classe `C1`, linhas (3, 6);
- Caso 2: classe `C1`, linhas (7, 8);
- Caso 3: classe `C1`, linhas (6, 7).

2.4 Grafo de Fluxo de Controle (*CFG*)

O grafo de fluxo de controle é uma estrutura descrita em [2, 19] utilizada para representar o fluxo de execução de um programa. Neste grafo, temos que:

- Cada nó contém o número de uma linha do código do programa. Além do número, pode-se ter a linha inteira do código ou parte dela, para facilitar a visualização do grafo por parte do usuário;
- As arestas são orientadas e, quando cheias, indicam o fluxo de execução dentro de um único método; quando tracejadas, indicam que o fluxo de execução passa por um método distinto.

A seguir, um exemplo de grafo de fluxo de controle:

```
 1: package exemplo;
 2:
 3: public class SuperNo {
 4:   No raiz;
 5:   No folha;
 6:
 7:   public SuperNo() {
 8:     raiz = null;
 9:     folha = null;
10:   }
11:
12:   public void adicionaNo (No novoNo) {
13:     if (raiz == null) {
14:       raiz = novoNo;
15:       folha = raiz;
16:     }
17:     else {
18:       folha.prox = novoNo;
19:       folha = folha.prox;
20:     }
21:   }
22: }
```

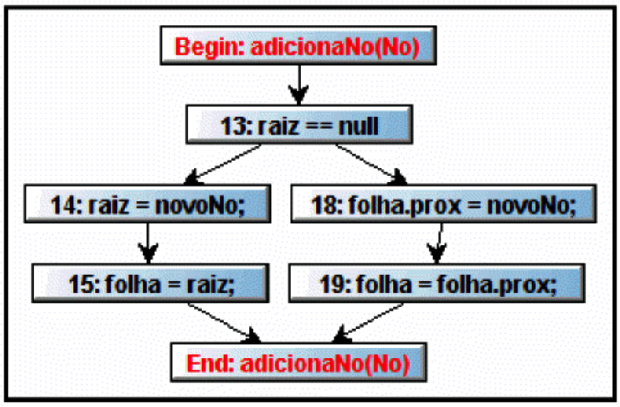


Figura 2.6: Exemplo de código Java (escrito por Paulo R. A. F. Nunes) e seu respectivo grafo de fluxo de controle. Como mostra a figura, os nós do grafo contêm, além do número da linha, seu conteúdo completo. Não há arestas tracejadas por se tratar de um grafo referente a um único método.

O grafo de fluxo de controle, junto com as definições e usos, é utilizado na aplicação de critérios de teste [2, 20]. Informações relativas à implementação e geração do grafo na OConGra [11, 12] e na OConGraX podem ser obtidas, respectivamente, nos capítulos 3 e 4.

Capítulo 3

OConGra

Neste capítulo, será apresentada a ferramenta OConGra (*Object Control Graph*) [11, 12], na qual foi baseado o trabalho. Os itens a serem abordados são:

- Descrição da OConGra.
- Especificações Técnicas e Implementação das Funcionalidades da ferramenta.
- Exemplo de uso.

3.1 Descrição

OConGra (*Object Control Graph*) é uma ferramenta desenvolvida por Nunes e Melo [11, 12] que, ao receber um sistema Java como entrada, retorna os seguintes dados ao usuário:

- Informações sobre as classes do sistema e seus respectivos objetos e métodos. Para cada método, são mostrados os parâmetros recebidos e definições e usos dos objetos nele existentes. Estas informações são apresentadas no *frame* principal do programa, em forma de árvore (`JTree`).
- Definição e uso dos objetos: uma listagem de todas as definições e usos dos objetos do sistema, separadas por método de classe.
- Grafo de fluxo de controle de objetos: a OConGra oferece uma visualização do grafo de fluxo de controle de um objeto de uma classe. Tanto a classe quanto o objeto podem ser escolhidos pelo usuário da ferramenta.

3.2 Especificações Técnicas

Nesta seção, serão apresentadas algumas especificações técnicas e dados relativos à implementação da ferramenta OConGra.

3.2.1 Linguagem e Ambiente de Desenvolvimento

A OConGra foi escrita em Java 1.4.2 [22], e desenvolvida utilizando-se o ambiente de desenvolvimento Eclipse [5]. Para a implementação da interface gráfica, foi utilizada a biblioteca *AWT* de Java.

3.2.2 *Frameworks*

Os *frameworks* são ferramentas utilizadas para auxiliar a codificação e desenvolvimento do programa. No caso da OConGra, dois *frameworks* foram usados: o *framework Recoder* [15] e o *JGraph* [9].

Recoder

Recoder [15] é um *framework* Java *open-source*[1] capaz de, dado um programa escrito nesta linguagem, realizar o *parsing*(análise de cada palavra e símbolo contidos nos arquivos fonte recebidos como entrada), transformar os arquivos fonte por meio da alteração de seus *bytecodes* (arquivos `.class`) e analisar o sistema léxica e sintaticamente. Este projeto foi inicialmente desenvolvido na *Universität Karlsruhe* (Alemanha) e *FZI Forschungszentrum Informatik Karlsruhe* (Alemanha), e atualmente seu desenvolvimento é realizado na *Växjö Universitet* (Suécia). Seu idealizador foi Dr. Andreas Ludwig e seus atuais desenvolvedores são Tobias Gutzmann, Mircea Trifu e Dr. Dirk Heuzeroth.

Figura 3.1: *Framework Recoder*

Este *framework* é utilizado para varrer os arquivos `.java` recebidos, de modo a obter dados necessários para determinar as definições e uso de objetos e construir o grafo de fluxo de controle de objetos (*OCFG*).

A versão do *Recoder* utilizada na OConGra foi a 0.73, lançada em 10 de maio de 2004.

JGraph

JGraph [9] é um *framework* utilizado para gerar e manipular grafos por meio da linguagem Java. É totalmente compatível com a biblioteca gráfica *Swing* do Java.

O projeto *JGraph* foi iniciado por Gaudenz Alder na *Swiss Federal Institute of Technology* (Suíça), em 2000. Tornou-se rapidamente um sucesso, sendo considerada a biblioteca de grafo Java *open-source* mais popular disponível.

Figura 3.2: *Framework JGraph*

A versão do *JGraph* utilizada na OConGra foi a 3.4.1, lançada em 11 de maio de 2004.

3.3 Funcionalidades da OConGra

As funcionalidades implementadas na OConGra são:

- Obtenção das definições e usos dos objetos do sistema Java recebido como entrada;
- Geração do grafo de fluxo de controle dos objetos.

[1]Um aplicativo *open-source* possui código aberto, ie, possui seu código disponível para *download*. Seu uso é gratuito.

3.3.1 Definição e Uso de Objetos

A programação do código para definição e uso de objetos foi feita utilizando o *framework Recoder* [15], e teve como base o conceito apresentado em 2.3.1.

As definições e usos dos objetos são determinadas no momento da leitura do código Java recebido como entrada, e armazenadas em dois vetores: um contendo apenas as definições dos objetos, e outro, apenas os usos.

Duas formas de visualização das definições e usos de objetos são oferecidas na OConGra: a primeira, aparece no *frame* principal; a segunda, aparece num *frame* específico para as definições e usos. Para facilitar a visualização, são fornecidas apenas uma listagem das definições e usos de um método da classe por vez, sendo este método escolhido pelo usuário. Um exemplo de como são estas visualizações pode ser visto em 3.4;

3.3.2 Grafo de Fluxo de Controle de Objetos (*OCFG*)

O grafo de fluxo de controle de objetos (*OCFG*) é uma variante do *CFG* (ver 2.4), e representa todos os caminhos possíveis a serem percorridos pelos objetos durante a execução de um programa em linguagem OO.

Na OConGra, o grafo gerado é específico para uma classe e um objeto desta classe [2]. Cada nó do grafo resultante contém o número da linha percorrida e seu conteúdo. A cor das palavras contidas num nó podem variar da seguinte maneira:

1. **Verde**: se o objeto é definido no código daquela linha;
2. **Azul**: se o objeto é usado no código daquela linha;
3. **Roxo**: se o objeto é definido e usado no código daquela linha;
4. **Vermelha**: indica o nó de início e fim de um método;

As arestas do grafo são orientadas, e representadas por dois tipos:

1. **Arestas de Controle**: são representadas por flechas de linha cheia, e utilizadas para indicar o fluxo de controle de um método de um programa;
2. **Arestas Passa-Mensagem**: são representadas por flechas de linha tracejada, e utilizadas para indicar chamadas feitas pelo método analisado a outros métodos da mesma classe ou de classes distintas do sistema.

Com as informações obtidas no grafo e com as definições e usos dos objetos do sistema, é possível agilizar grande parte do processo de testes estruturais de programas escritos em linguagem OO (os procedimentos restantes seriam a obtenção dos pares de definição e uso e a definição dos critérios de testes a serem utilizados, para montagem dos casos a serem testados).

3.4 Exemplo de Uso

Nesta seção, serão mostrados *screenshots* da OConGra, quando esta recebe um código Java como entrada.

[2] Tanto a classe quanto o objeto podem ser especificados pelo usuário na interface da OConGra.

3.4.1 Código

O código aqui utilizado é baseado no exemplo apresentado por Chen e Kao em [2]:

```
 1: public class Frame {
 2:   Shape shape_obj;
 3:
 4:   public void draw_frame(){
 5:     shape_obj.draw();
 6:     if (shape_obj.get_param() > 10)
 7:       shape_obj.double1();
 8:       shape_obj.draw();
 9:   }
10:
11:   public Shape set_obj(int shape, int perimeter, int x, int y){
12:     if (shape == 1){
13:       shape_obj = new Square(perimeter, x, y);
14:     }
15:     else {
16:       shape_obj = new Circle(perimeter, x, y);
17:     }
18:     return shape_obj;
19:   }
20: }
```

```
 1: public abstract class Shape {
 2:   int perimeter;
 3:
 4:   public void set_param(int param){ perimeter = param;};
 5:
 6:   public int get_param(){ return perimeter;};
 7:
 8:   public void double1() { perimeter = perimeter * 2;};
 9:
10:   public abstract int draw();
11: }
```

```
 1: public class Circle extends Shape {
 2:   int x, y;
 3:   double radius;
 4:
 5:   public Circle(int x, int y, int init_perimeter){
 6:     this.x = x;
 7:     this.y = y;
 8:     perimeter = init_perimeter;
 9:     this.radius = init_perimeter /(2 * 3.14);
10:   }
11:
12:   public int draw(){
13:     System.out.println("Draw Circle");
14:     return 0;
15:   }
16: }
```

Figura 3.3: Código Java (parte 1) utilizado para exemplificar o uso da OConGra

```
 1: public class Square extends Shape {
 2:   protected int x, y;
 3:
 4:   public Square(int x, int y, int init_perimeter){
 5:     this.x = x ;
 6:     this.y = y ;
 7:     perimeter = init_perimeter;
 8:   }
 9:
10:   public int draw(){
11:     System.out.println("Draw Square");
12:     return 0;
13:   }
14: }
```

```
 1: public class Test {
 2:   static Frame frame_obj;
 3:
 4:   static public void Main (String[] args){
 5:     frame_obj = new Frame();
 6:     int shape = Integer.parseInt(args[0]);
 7:     int perimeter = Integer.parseInt(args[1]);
 8:     int x = Integer.parseInt(args[2]);
 9:     int y = Integer.parseInt(args[3]);
10:     frame_obj.set_obj(shape, perimeter, x, y);
11:     frame_obj.draw_frame();
12:   }
13: }
```

Figura 3.4: Código Java (parte 2) utilizado para exemplificar o uso da OConGra

3.4.2 Seleção do arquivo de entrada

Ao executar a OConGra, a primeira coisa a fazer é selecionar o arquivo a ser analisado pela ferramenta. Para isso, selecionamos `Main` → `Open Project` e clicamos num arquivo `.java` do pacote[3] desejado.

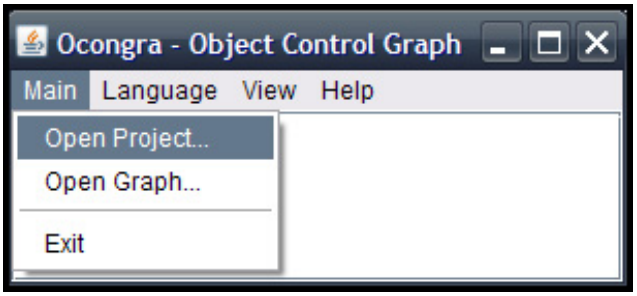


Figura 3.5: Para selecionar um arquivo, clicamos no menu da OConGra em `Main` → `Open Project`.

[3]Pacote é um conjunto de classes.

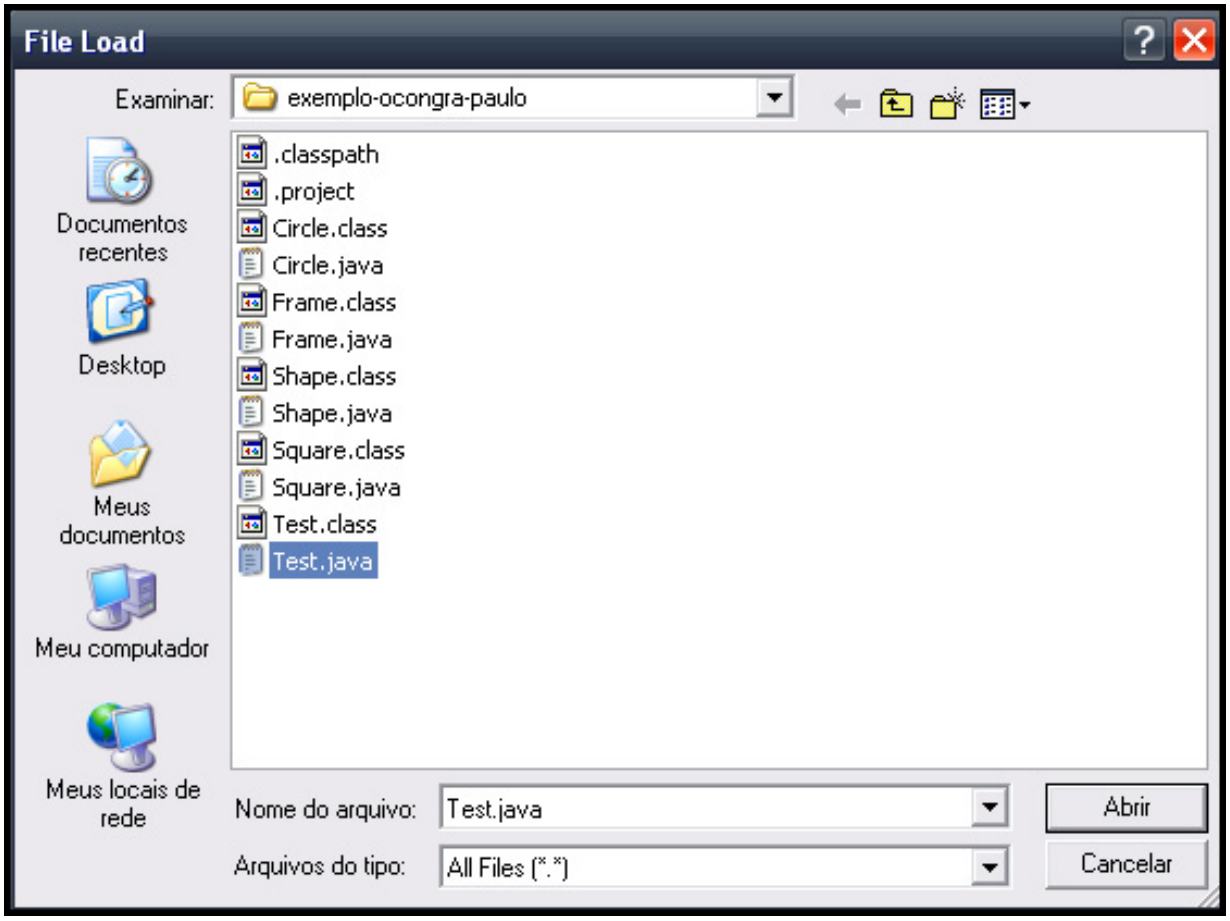


Figura 3.6: Seleção de um arquivo `.java` pertencente ao pacote desejado. No caso, selecionamos o arquivo `Test.java`, pertencente ao pacote `exemplo-ocongra-paulo`.

Após selecionar o arquivo, a OConGra realiza a leitura de todos os arquivos `.java` existentes no pacote, de forma recursiva. Ao mesmo tempo em que faz a leitura, ela determina as linhas em que há definição e uso de objetos nas classes do sistema, e exibe seu *frame* principal, com uma árvore (`JTree`) contendo informações relativas ao sistema Java passado como entrada.

***

Como dito na seção 3.1, a árvore apresenta a visualização das classes do pacote. No nosso caso, as classes identificadas são: `Frame`, `Shape`, `Circle`, `Square` e `Test`. Para cada classe, é possível visualizar os objetos e métodos que ela possui e, para cada método, os parâmetros recebidos e as linhas de definição e uso nele existentes.

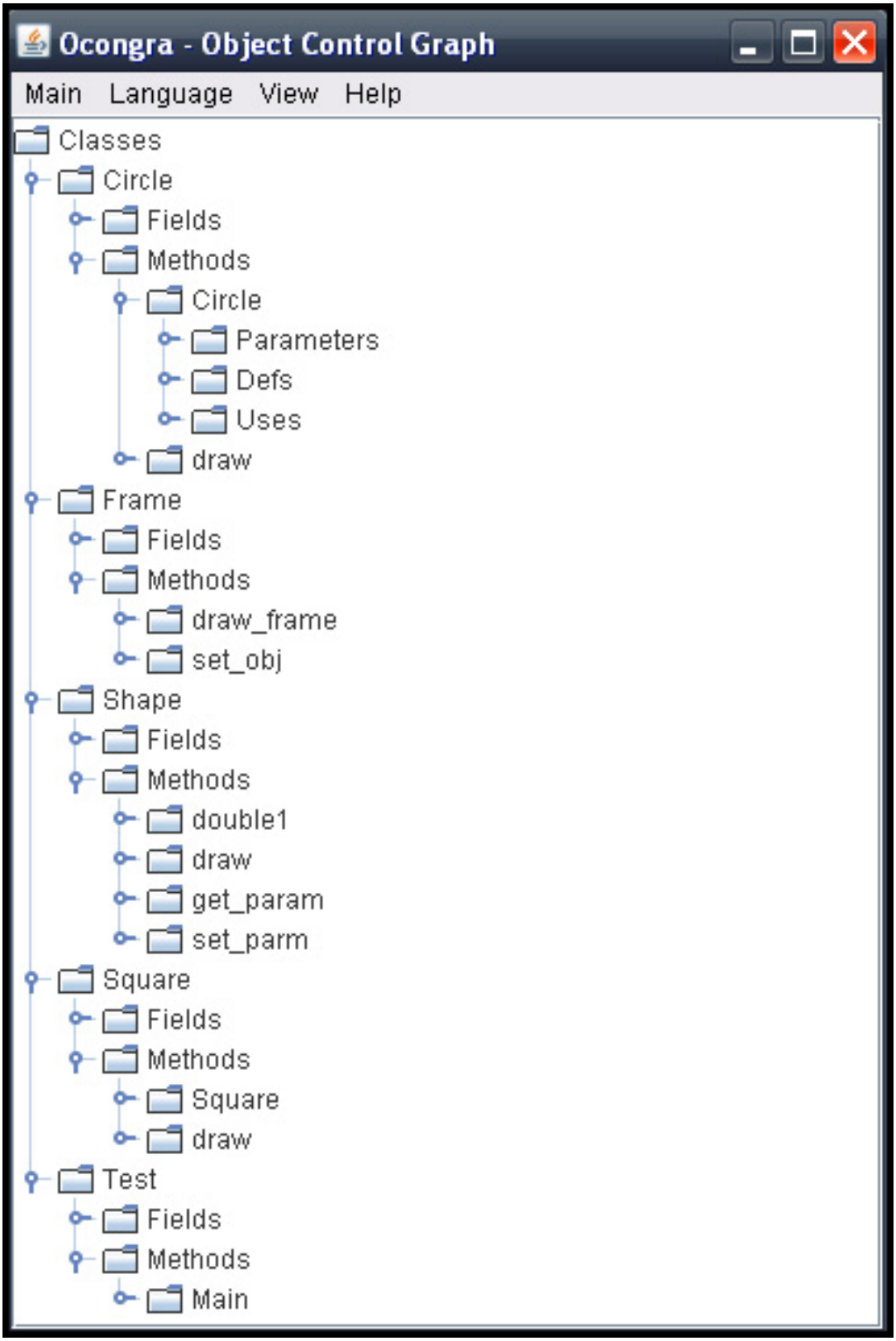


Figura 3.7: Frame principal da OConGra. Nele é exibida a árvore que contém as classes do pacote passado e informações específicas sobre cada uma das classes.

3.4.3 Visualização das Definições e Usos de Objetos

Uma outra opção para visualizar as definições e usos dos objetos, é abrir um outro *frame* que fornece somente a visualização das definições e usos. Neste novo *frame*, é possível selecionar uma classe do pacote e um método desta classe cujas linhas de definição e uso de objetos desejamos visualizar.

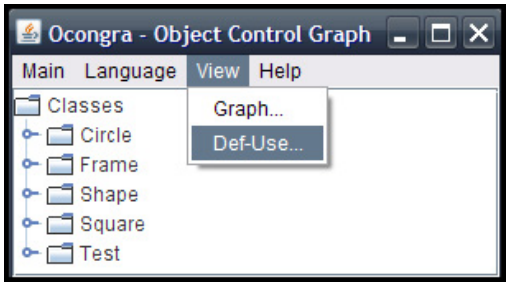


Figura 3.8: Para visualizar o *frame* de definição e uso de objetos, basta selecionar no menu: `View` → `Def-Use`.

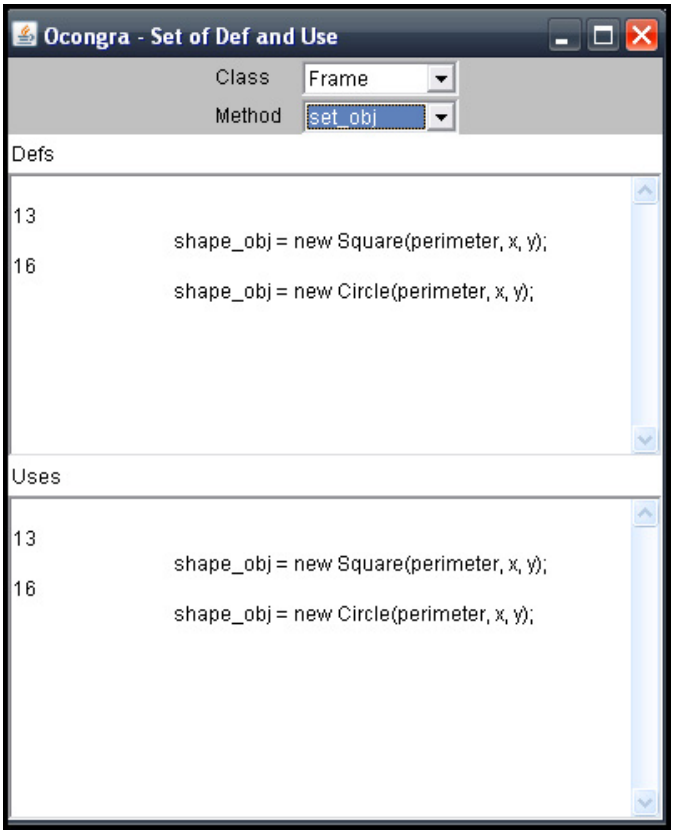


Figura 3.9: Visualização do *frame* de definição e uso de objetos exibido para o usuário. Neste caso, são exibidas as linhas em que objetos são definidos e usados no método `set_obj` da classe `Frame`

3.4.4 Visualização do Grafo de Fluxo de Controle de Objetos (*OCFG*)

Para visualizar o *OCFG*, é necessário abrir um novo *frame* da OConGra. Neste *frame*, o usuário pode, além de ver, editar o grafo, alterando o tamanho dos nós e suas posições.

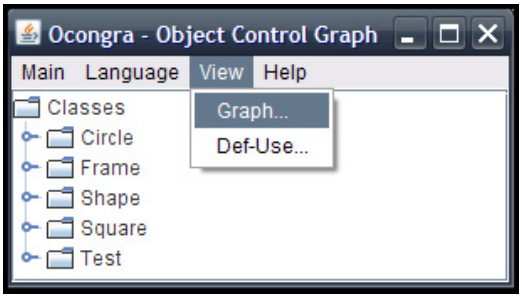


Figura 3.10: Para visualizar o grafo, basta selecionar no menu: `View` → `Graph`

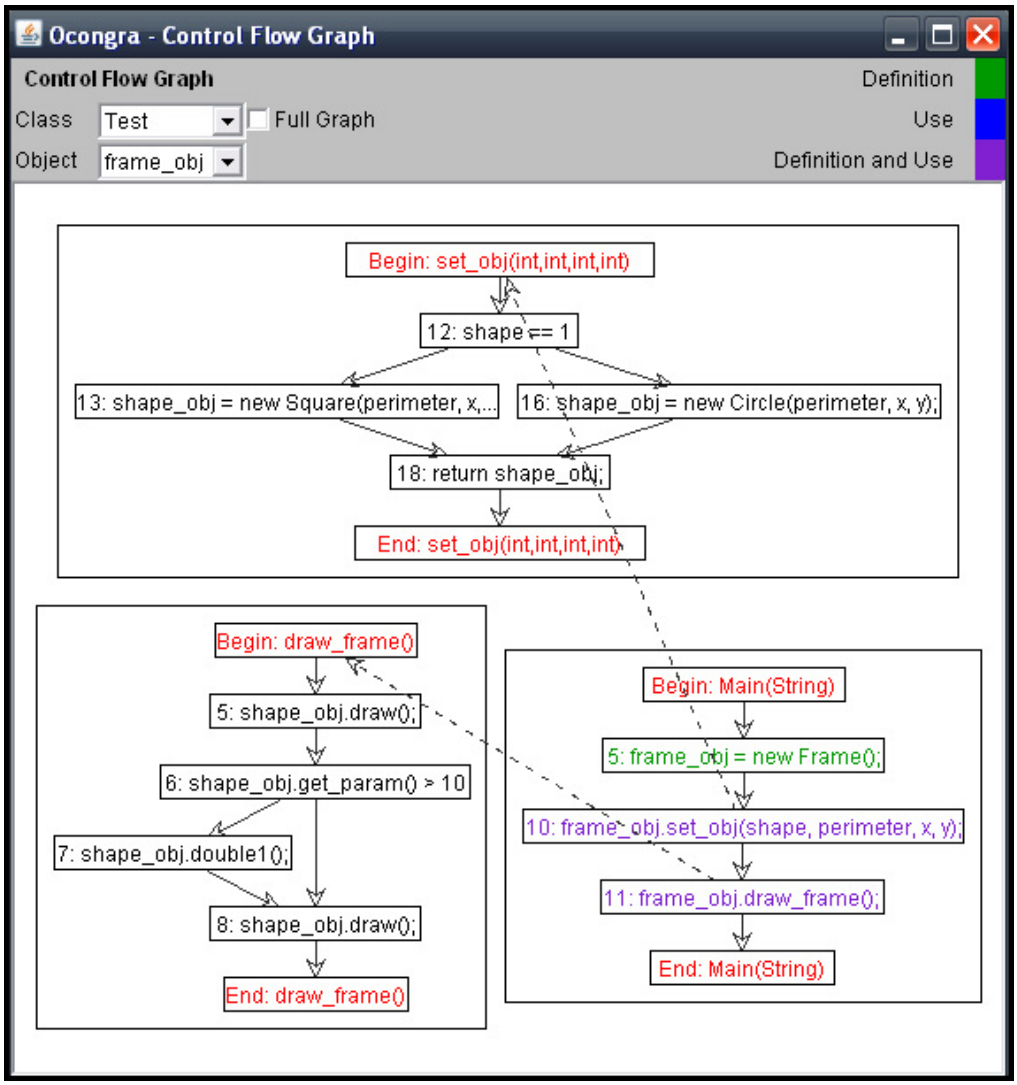


Figura 3.11: Visualização do grafo do objeto `frame_obj` da classe `Test`, com a opção "`Full Graph`" desmarcada (se esta opção estivesse marcada, todos os nós dos métodos visualizados no grafo apareceriam, e não apenas os mais relevantes, como no nosso caso). A legenda do grafo foi apresentada na seção 3.3.2.

Capítulo 4

OConGraX

Neste capítulo, será apresentada a OConGraX (*Object Control Graph with Exceptions*), software desenvolvido neste trabalho. Os seguintes itens serão abordados:

- Descrição da ferramenta.
- Implementação e Funcionalidades da ferramenta.
- Exemplo de uso.

4.1 Descrição

A OConGraX é uma extensão da OConGra [11, 12], ferramenta descrita no capítulo 3. O objetivo principal do software desenvolvido era manter as funcionalidades da OConGra (seção 3.3) e acrescentar as seguintes:

- Obtenção das definições e usos de exceções;
- Obtenção dos pares de definição e uso de exceções;
- Geração do *OCFG* com informações adicionais sobre os mecanismos de tratamento de exceção (seção 2.1.2) utilizados.

A adição destas novas funcionalidades visa a facilitar o processo de testes de programas escritos em linguagem OO com tratamento de exceção.

4.2 Especificações Técnicas

Nesta seção, relatarei detalhes sobre a implementação da OConGraX e suas funcionalidades.

4.2.1 Linguagem e Ambiente de Desenvolvimento

A linguagem utilizada para a implementação da OConGraX foi Java 6.0 [22], e o ambiente de desenvolvimento escolhido foi o Eclipse 3.2.2 [5]. A interface gráfica da OConGra foi modificada na OConGraX com o auxílio das novas funcionalidades da biblioteca *Swing* implementadas no Java 6.0.

4.2.2 *Frameworks*

Os *frameworks* utilizados aqui são os mesmos utilizados na OConGra (seção 3.2.2), só que em versões mais atuais. Nas próximas subseções, serão descritas as novas versões dos *frameworks* utilizados.

Recoder

A versão do *Recoder* [15] utilizada na OConGraX é a 0.81, lançada em 08 de maio de 2006. A principal novidade com relação à versão utilizada na OConGra seria o suporte à versão Java 5.0 ou superior. Esta foi uma atualização muito importante, pois a partir da versão 5.0, novas funcionalidades foram implementadas na linguagem, como novas funções para programação concorrente (pacote `java.util.concurrent`) e o acréscimo na sintaxe Java dos chamados "tipos genéricos".

Figura 4.1: ***Framework Recoder***

Podemos dizer que a adição dos "tipos genéricos" foi a principal razão da utilização de uma versão mais recente do *framework Recoder*. Isso porque os "tipos genéricos" acrescentaram novos elementos à sintaxe da linguagem, no caso, $<$ e $>$, na declaração de elementos como os dos pacotes `java.util.*`[1] (e.g., ao invés de escrever "`Vector v = new Vector();`" devo escrever agora "`Vector<Integer> v = new Vector<Integer>();`", restringindo assim o tipo de variável que esta coleção pode receber). Esta alteração fez com que os desenvolvedores do *framework Recoder* modificassem a ferramenta, de modo a considerar este novo símbolo na análise léxica feita pelo *framework*, evitando assim que o *Recoder* retornasse algum erro no caso de receber programas escritos com esta nova funcionalidade de Java.

JGraph

A versão utilizada na OConGraX é a 5.10.1.0, lançada em 07 de maio de 2007. Entre as vantagens que esta nova versão apresenta em relação à versão anterior utilizada na OConGra, estão: facilidades na implementação de funções que permitem salvar o grafo como um arquivo de imagem, maior compatibilidade com as funções da biblioteca *Java Swing* e facilidade na personalização do grafo, como utilização de cores e bordas.

Figura 4.2: ***Framework JGraph***

4.2.3 Nova Tecnologia estudada: a Linguagem XML

Uma das preocupações no desenvolvimento da OConGraX, era que os dados de definição e uso de objetos e exceções por ela gerados fossem compartilhados com outros aplicativos utilizados no processo de testes e verificação de programas. Para tornar isso possível, resolvemos utilizar a linguagem XML.

XML (*EXtensible Markup Language*: Linguagem Extensível de Marcação) [25, 24] é uma linguagem recomendada pela *W3C* (*World Wide Web Consortium*)[2] e é utilizada para estruturar, armazenar e enviar dados.

[1] Informações técnicas sobre este pacote podem ser obtidas em `http://java.sun.com/j2se/1.5.0/docs/api/java/util/package-summary.html`

[2] Informações sobre a W3C podem ser encontradas em: `http://www.w3.org/`

Sintaxe

A sintaxe utiliza-se de *tags*, estruturas de marcação cuja função é delimitar um bloco de dados do arquivo. Muitas vezes, os nomes das *tags* são usados para identificar os tipos de dados mantidos por ela, ou mesmo informar sobre a estrutura do arquivo. Em XML, não há *tags* pré-definidas. Portanto, os programadores podem definir as *tags* a serem utilizadas da maneira que melhor se adequar às suas necessidades.

Estrutura do arquivo

Num arquivo XML, a primeira linha contém a versão de XML utilizada e a codificação utilizada no documento. A segunda, contém a *tag* do chamado **elemento raiz**. Este elemento é obrigatório em qualquer documento escrito em XML.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<mensagem>
  <para>umaPessoa</para>
  <de>outraPessoa</de>
  <assunto>Reunião</assunto>
  <corpo>Marcada para o dia 08/10, às 10h.</corpo>
</mensagem>
```

Figura 4.3: Exemplo de arquivo escrito em XML.

Após o elemento raiz, são escritos os chamados **elementos filhos**, com seus respectivos dados e *tags*. Estes elementos, assim como o elemento raiz, podem ter elementos filhos (ou sub-elementos). É importante ressaltar o fato de que toda vez que uma *tag* é aberta, ela deve ser **fechada** e de acordo com a **ordem** em que foi aberta. Por exemplo, em XML, a seguinte linha estaria incorreta:

```
<a><b>algum dado</a></b>
```

O correto seria:

```
<a><b>algum dado</b></a>
```

Outro detalhe relativo às *tags* é que é feita diferenciação entre maiúsculas e minúsculas em seus nomes. Desta maneira, se a seguinte linha for escrita:

```
<TaG>outro dado</tag>
```

Ela estará incorreta pois não há o correto encadeamento de tags (“TaG” é diferente de “tag” e, portanto, o bloco de dados não é corretamente fechado).

Para ilustrar melhor a estrutura de um arquivo XML, observe a figura 4.3. De acordo com a primeira linha do arquivo, a versão de XML utilizada é a 1.0, e a codificação, a ISO-8859-1 (correspondente a línguas ocidentais). A segunda linha contém o elemento raiz do documento, no caso, “mensagem”. Os demais elementos descritos nas linhas seguintes (“para”, “de”, “assunto”, “corpo”) são os elementos filhos do elemento raiz.

Sax e *DOM*

Para realizar a manipulação de arquivos XML, a *W3C* especificou dois mecanismos: *SAX* e *DOM*.

SAX (***S**imple **A**PI for **X**ML*) [27, 10, 24] é uma API[3] utilizada, em geral, para manipulação de arquivos XML grandes. Isso acontece porque ela permite, por exemplo, a leitura de trechos específicos do documento, fazendo com que o uso da memória do computador seja menor, exigindo menos processamento. Apesar dessas vantagens, implementar funções utilizando *SAX* é uma tarefa complicada e trabalhosa.

DOM (***D**ocument **O**bject **M**odel*) [27, 24], diferentemente de *SAX*, possui uma implementação mais simples e intuitiva. Em compensação, ele faz um uso maior da memória do computador, por necessitar manipular o arquivo XML inteiro, e não apenas partes dele como com *SAX*. Por exemplo, para criar um arquivo XML utilizando *DOM*, é preciso definir toda a estrutura do arquivo e armazená-la em memória junto com os dados a serem inseridos no documento, para só depois realizar a escrita do arquivo propriamente dito.

Para a geração do arquivo XML na OConGraX, foi realizada a implementação com DOM. Detalhes sobre como as implementações são realizadas com *SAX* e *DOM* podem ser obtidas em [24].

4.3 Implementação

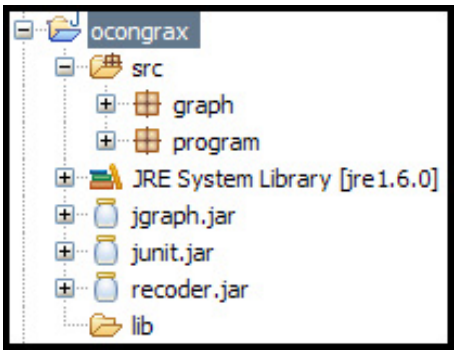


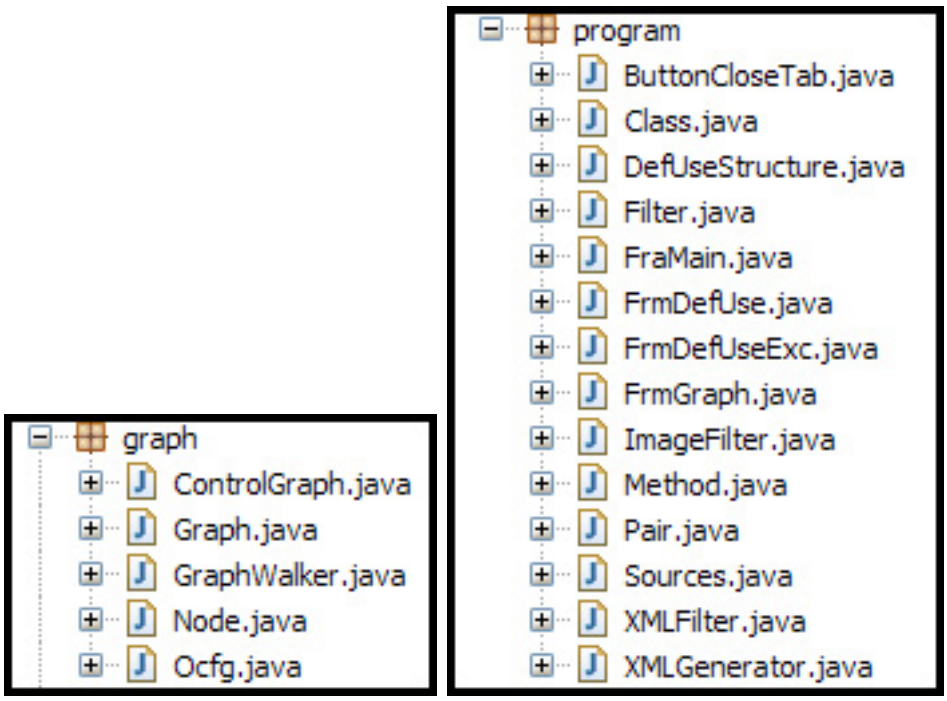
Figura 4.4: Pastas e Pacotes da OConGraX

O software OConGraX é constituído de dois diretórios: `src` e `lib`. Em `lib` estão os arquivos dos *frameworks* utilizados: *Recoder* e *JGraph* (o *framework JUnit* é utilizado pelo *Recoder*). Em `src`, estão os arquivos `.java` do programa agrupados em dois pacotes: `graph` e `program`.

No pacote `graph` estão as classes responsáveis por gerar o grafo de controle de fluxo de objetos. São elas as responsáveis por montar os nós e arestas do grafo, e apresentar o grafo de uma forma que facilite a visualização para o usuário.

No pacote `program` estão as classes que implementam a interface gráfica para interação do usuário com a OConGraX e as funcionalidades da ferramenta. Maiores detalhes sobre a implementação da interface gráfica e das funcionalidades serão dadas nas próximas seções.

[3]API (*Application Programming Interface*: Interface de Programação de Aplicativos) é um conjunto de padrões e métodos estabelecidos para utilização de um determinado software.



(a) Classes do pacote `graph` (b) Classes do pacote `program`

Figura 4.5: Classes da OConGraX

4.3.1 Interface Gráfica

Para melhorar a usabilidade da OConGraX, foi feita uma atualização da interface gráfica.

Na OConGra, como descrito no capítulo 3, a interface gráfica foi implementada utilizando:

- Biblioteca *AWT* de Java;
- Navegação por menus;
- *Frames* para cada visualização (definição e uso de objetos/*OCFG*).

Na OConGraX, foram feitas as seguintes alterações:

- Biblioteca *Swing* de Java;
- Utilização do `JSplitPane`, componente gráfico utilizado para dividir o painel de visualização em duas partes; o tamanho ocupado por cada uma das partes pode ser ajustado pelo usuário utilizando-se as setas presentes na barra divisória.
- Utilização de abas para navegar entre as visualizações do grafo e das definições e usos de objetos e exceções.

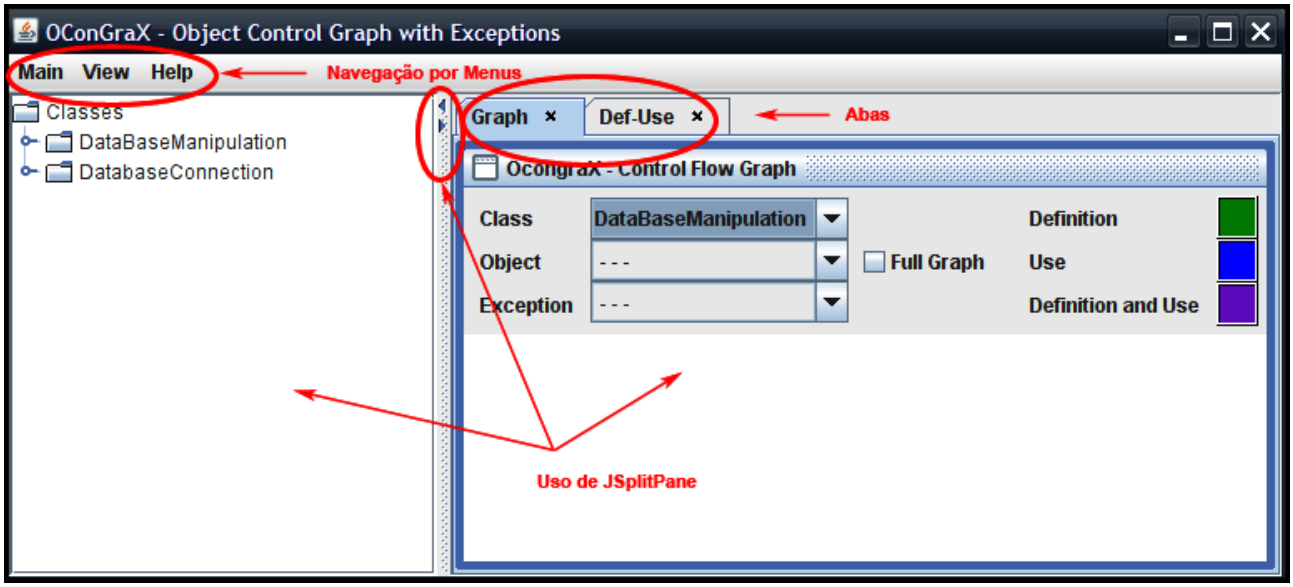


Figura 4.6: Interface Gráfica da OConGraX

O `JSplitPane` permite que o usuário veja a árvore de informações das classes do sistema Java recebido como entrada sempre que desejar. Além disso, o uso de abas para visualizar as definições e usos e o grafo torna a interação com a ferramenta mais ágil. Uma apresentação mais completa da interface será mostrada na seção 4.5.

4.4 Funcionalidades da OConGraX

A OConGraX, por ser uma extensão da OConGra [11, 12], possui as mesmas funcionalidades desta ferramenta (seção 3.3). Ademais, as seguintes funcionalidades foram acrescentadas à OConGraX:

- Obtenção das linhas de definições e usos das exceções;
- Obtenção dos pares de definição-uso de exceções;
- Permite salvar as linhas de definições e usos de objetos e exceções num arquivo XML;
- Geração do *OCFG* com informações sobre os mecanismos de tratamento de exceção utilizados no código recebido como entrada;
- Permite salvar o grafo como arquivo de imagem (`jpg`, `jpeg`, `gif`, `bmp`, `png`).

A seguir, será explicado com maiores detalhes a implementação de cada uma das funcionalidades citadas acima.

4.4.1 Definição e Uso de Exceções

A implementação da funcionalidade foi feita de acordo com o conceito apresentado na seção 2.3.2. Para sua realização, foi utilizado o aplicativo *Sourcerer* presente no *framework Recoder* [15]. Este aplicativo mostra como as estruturas do código são identificadas pelo *framework*, o que foi de grande ajuda para a obtenção das linhas de definição e uso de exceções.

A obtenção das definições e usos das exceções foi realizada de modo análogo a dos objetos, na OConGra: no momento que a leitura é realizada (por meio de uma varredura nas classes e métodos dos arquivos `.java` do sistema recebido como entrada), as definições e usos são armazenadas, respectivamente, num vetor de definições e num vetor de usos. Estes vetores armazenam todas as definições e usos de objetos e exceções do sistema.

A visualização das definições e usos das exceções é feita tanto na árvore, mostrada do lado esquerdo do frame da ferramenta, quanto na aba específica para visualização de definições e usos. Não há diferenciação nas linhas de definição e uso mostradas, ou seja, todas as linhas em que há definição e/ou uso de objetos e exceções num método (método este escolhido pelo usuário, na interface) são mostradas. Uma apresentação melhor deste fato poderá ser observada na seção 4.5.

A seguir, são mostrados trechos do código da OConGraX, responsáveis por obter e armazenar as definições e usos de objetos e exceções.

```
TreeWalker twClass = new TreeWalker(compUnit);
Class cls = null;

while (twClass.next()){
    ProgramElement pe = twClass.getProgramElement();
    pe.toSource();
    if (pe instanceof ClassDeclaration){
        ...
    }

    if (pe instanceof MethodDeclaration){
        Method met = null;

        String nm = ((MethodDeclaration)pe).getName();
        met = new Method(nm, cls.getFields(), (MethodDeclaration)pe,cls);
        met.getParameters(pe);

        met.setDefUse(cls, met, pe);
        ...
```

Figura 4.7: Trecho de código da classe `FraMain.java`, que mostra a varredura do arquivo `.java`. Para a realização desta varredura, é utilizada a estrutura do *framework Recoder* [15], `TreeWalker`, que permite navegar por cada elemento presente no código Java. O método responsável por armazenar as definições e usos é `setDefUse`

```
public void setDefUse(Class c, Method m, ProgramElement p) {
    Vector<Identifier> localExcDef = new Vector<Identifier>();
    TreeWalker twMethod = new TreeWalker(p);
    while (twMethod.next()) {
        ProgramElement pe = twMethod.getProgramElement();
        if (pe instanceof CopyAssignment)
            getCopy(c.getName(), (CopyAssignment) pe, localExcDef);
        else if (pe instanceof MethodReference)
            getMR(c.getName(), (MethodReference) pe);
        else if (pe instanceof LocalVariableDeclaration)
            getLocal(c.getName(), (LocalVariableDeclaration) pe, localExcDef);
        else if (pe instanceof Catch)
            getCatch(c.getName(), (Catch) pe);
        else if (pe instanceof Throw)
            getThrow(c.getName(), m, (Throw) pe, localExcDef);
        else if (pe instanceof Return)
            getReturn(c.getName(), (Return) pe);
    }
}
```

Figura 4.8: Método da classe `Method.java`, responsável por armazenar as definições e usos de objetos e exceções. Para o caso específico das exceções, são mais importantes os casos em que `pe` é "instância" de *Catch*, *Throw* e *LocalVariableDeclaration* (representa declarações locais de variáveis, no caso específico de exceções, declarações de variáveis de exceções). *MethodReference* (representa as chamadas de métodos) e *CopyAssignment* (representa as atribuições), além de *Return* são utilizados para a obtenção de definições e usos tanto de objetos quanto de exceções. Os casos em que pe é uma instância de *Try* ou *Finally* não são vistos, uma vez que tanto Try quanto Finally são estruturas usadas apenas para marcar o início de um bloco com instruções. E estas instruções podem conter as estruturas de exceções já mencionadas anteriormente.

Par de Definição-Uso de Exceções

Os pares de definição-uso de exceções foram implementados levando-se em consideração o conceito apresentado em 2.3.2. Com exceção do caso em que o par é formado por "*throw*" e "*catch*", cada um presente num método, a obtenção dos pares é feita no momento da leitura do arquivo `.java`, junto com a obtenção das linhas de definição e uso. Para obter o outro tipo de par, é feito um processamento após a obtenção de todas as definições e usos, métodos e classes do sistema. Neste processamento, são verificados os locais em que há chamadas de métodos que lançam exceções e, assim, são obtidos os pares restantes.

A visualização dos pares de definição-uso de exceções só é possível, no momento, no arquivo XML gerado pela OConGraX. Detalhes sobre este arquivo serão dados na próxima subseção.

4.4.2 Arquivo XML gerado pela OConGraX

Para poder compartilhar as informações de definições e usos de objetos e exceções, foi utilizada a linguagem XML [25, 24] (seção 4.2.3).

A estrutura do arquivo XML utilizada na OConGraX é mostrada ao lado. A versão de XML utilizada é a 1.0, e a codificação, ISO 8859-1, correspondente às linguas ocidentais.

O elemento raiz é "*elements*". Seus filhos são "*object*" e "*exception*":

O elemento "*object*" armazena o nome de um objeto. Possui os sub-elementos "*def*" e "*use*". Cada sub-elemento armazena, respectivamente, o número da linha em que o objeto identificado em "*object*" é definido e usado.

O elemento "*exception*" armazena o nome de uma exceção. Possui os sub-elementos "*pair*", "*def*" e "*use*". Em "pair", é armazenada a informação do número da linha em que a exceção identificada em "exception" é definida e o número da linha em que seu respectivo par é usado[4]. Já em "def" e "use", são armazenados, respectivamente, o número da linha em que a exceção identificada em "exception" é definida e usada.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <elements>
  - <object>
      <def>...</def>
      ...
      <use>...</use>
      ...
    </object>
    ...
  - <exception>
    - <pair>
        <def>...</def>
        <use>...</use>
      </pair>
      ...
      <def>...</def>
      ...
      <use>...</use>
      ...
    </exception>
    ...
</elements>
```

Figura 4.9: Estrutura do arquivo XML gerado pela OConGraX

Na OConGraX, é oferecida a opção de gerar o arquivo XML com os pares de definição-uso das exceções, ou sem os pares. Isto foi feito para dar maior flexibilidade para o usuário, no caso das informações dos pares não ser relevante para o critério de teste [2, 20] que ele deseja aplicar no software que ele deseja testar.

4.4.3 Grafo gerado pela OConGraX

O grafo gerado pela ferramenta é o *OCFG* (seção 3.3.2) com informações adicionais sobre os mecanismos de tratamento de exceção. Na OConGraX, o grafo gerado é referente a um objeto de uma classe (como no caso da OConGra) ou a uma exceção da classe. Tanto a classe quanto o objeto/exceção podem ser escolhidos pelo usuário.

Da OConGra, foi mantida a possibilidade de edição do grafo pelo usuário. Foi mantida também a opção "*Full Graph*", que quando marcada, faz com que sejam visualizados todos os nós dos métodos representados pelo grafo, e não apenas os mais relevantes para o objeto/exceção da classe em questão.

Apesar de ter se baseado no artigo de Sinha e Harrold [19], a implementação não seguiu à risca a notação sugerida no texto.

[4]No caso do par (uso) pertencer a outro método, é armazenado não somente o número da linha em que ele se encontra, mas também seu nome e a classe em que ele pertence

Representação dos Nós

Para diferenciar os nós que contém linhas pertencentes a blocos definidos por mecanismos de tratamento de exceção (*try*, *catch*, *finally*), ou que contenham o comando *throw*, eles foram pintados em tom alaranjado. Os demais nós passaram a ser coloridos em tom de azul. As cores das linhas presentes nos nós seguem o padrão definido na seção 3.3.2, com o acréscimo da cor magenta, utilizada para marcar o início e o fim de um bloco *Finally*.

Arestas

As **arestas de controle** (representadas por setas desenhadas com linha cheia) mostram os caminhos com os fluxos de execuções possíveis dentro de um mesmo método. No caso de ligar nós que contenham algum elemento pertencente a uma estrutura de tratamento de exceção, as arestas passam a ser laranjas, e não mais pretas.

As **arestas passa-mensagem** (representadas por setas desenhadas com linha tracejada) mostram os caminhos com os fluxos de execuções possíveis entre métodos distintos, que podem estar inclusive em classes diferentes. Com o acréscimo do caso das exceções, as arestas tracejadas passaram a ligar não apenas chamadas a métodos existentes no código recebido como entrada, mas também os nós *throw* e *catch* que formam pares de definição-uso. O uso das cores é análogo ao feito para as arestas de controle.

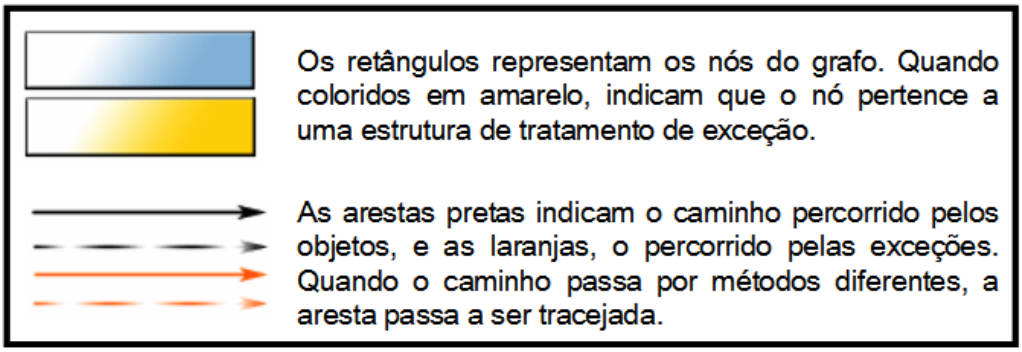


Figura 4.10: Notação utilizada para o grafo

Uma nova funcionalidade referente ao grafo implementada na OConGraX, é a possibilidade de salvá-lo como um arquivo de imagem. Isto foi possível graças ao uso da nova versão do *framework JGraph* [9].

Permitir salvar o grafo como arquivo de imagem dá ao usuário a chance de utilizar o grafo e as informações nele contidas em outros aplicativos de testes e verificação de programas, além de permitir que o grafo seja impresso.

4.5 Exemplo de Uso

Nesta seção serão apresentadas *screenshots* da OConGraX, de modo a ilustrar seu funcionamento ao receber um código Java com mecanismos de tratamento de exceção.

4.5.1 Código

O código utilizado neste exemplo é uma adaptação do apresentado por Sinha e Harrold em [19]:

```
1 public class DataBaseManipulation {
2   DatabaseConnection dbConn =
                  new DatabaseConnection();
3   String sqlCmd;
4   Array row;
5
6   void dbUpdateOperation () {
7     dbConn.open();
8     try {
9       row = dbConn.select(sqlCmd);
10      sqlCmd = updateFields(row);
11      dbConn.update(sqlCmd);
12    }
13    catch (UpdateException ue) {
14      showMessage(ue);
15    }
16    finally {
17      dbConn.close();
18    }
19  }
20 }
```

```
1 public class DatabaseConnection {
2
3   void update(String cmd)
                    throws UpdateException {
4     UpdateException u;
5     int status;
6
7     if ((status = executeCmd(cmd)) == 0) {
8       u = new UpdateException();
9       throw u;
10    }
11  }
12 }
```

Figura 4.11: Código Java usado para exemplificar o funcionamento da OConGraX

4.5.2 Seleção do projeto Java

Para selecionar o projeto Java a ser analisado pela OConGraX, basta clicar em `Main` → `Open Project`.

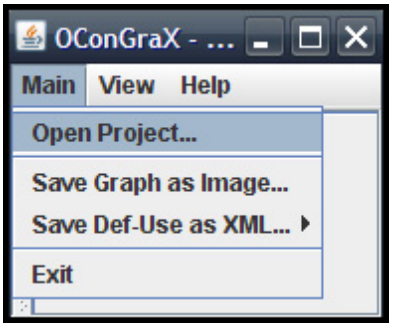


Figura 4.12: Para abrir um projeto Java na OConGraX, vá ao menu e clique em `Main` → `Open Project`

Após o clique, aparecerá uma caixa de diálogo onde o usuário selecionará o diretório que contém os pacotes com os arquivos `.java` do projeto desejado.

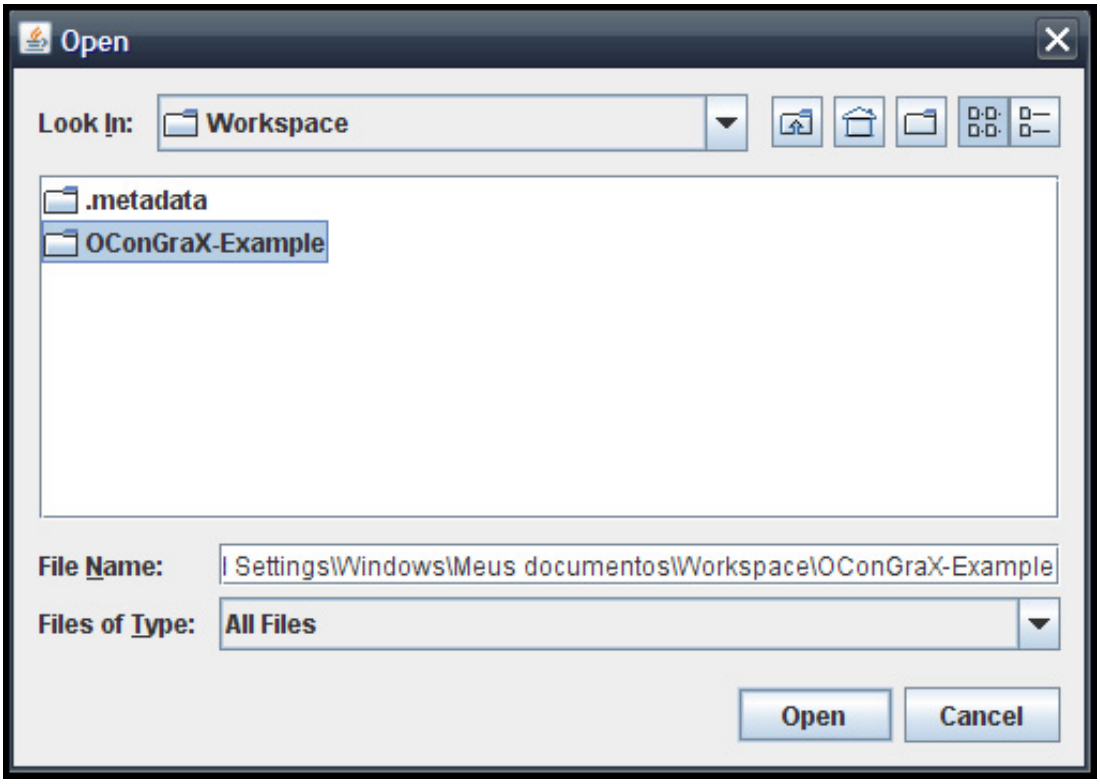


Figura 4.13: Seleção do diretório com os pacotes do projeto Java. No caso, o diretório é `OConGraX-Example`

Selecionado o diretório, a OConGraX realiza a leitura do código recebido e devolve a árvore com informações sobre as classes, métodos, objetos e exceções dos pacotes do projeto Java.

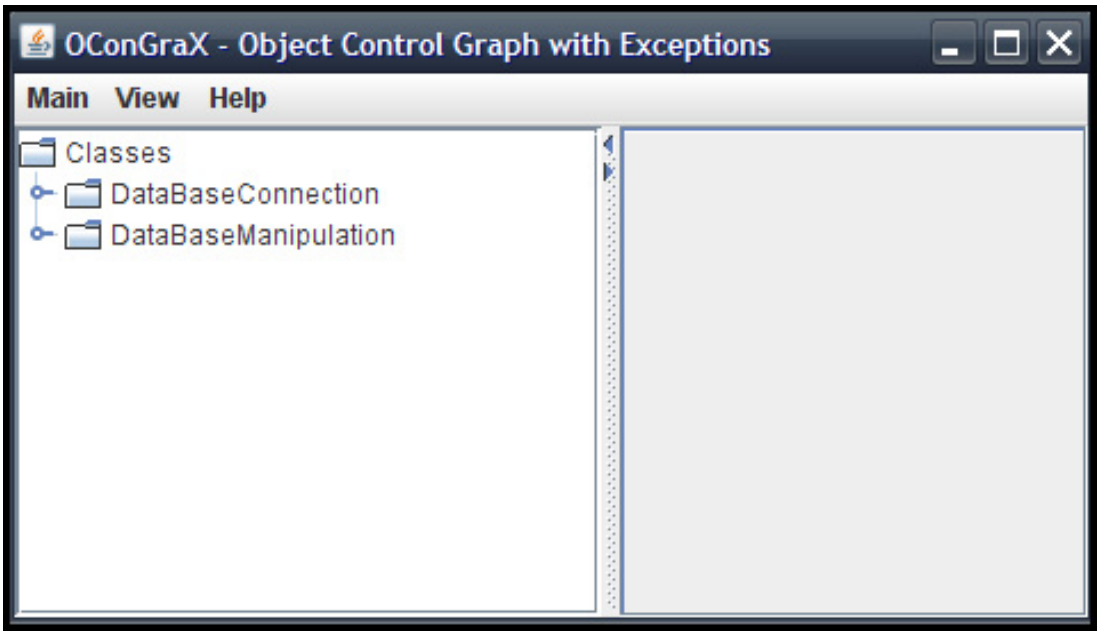


Figura 4.14: Visualização da OConGraX após o projeto Java ser aberto

4.5.3 Visualização das Definições e Usos de objetos e exceções

Para visualizar as definições e usos de objetos e exceções de um método, basta selecionar esta opção no menu.

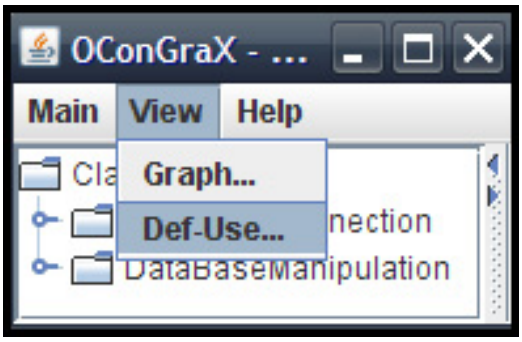


Figura 4.15: Para ver a aba de Definições e Usos, vá em `View` → `Def-Use`

Em seguida, será aberta a aba de Definições e Usos. Para visualizar as linhas em que elas ocorrem, selecione um método da classe desejada.

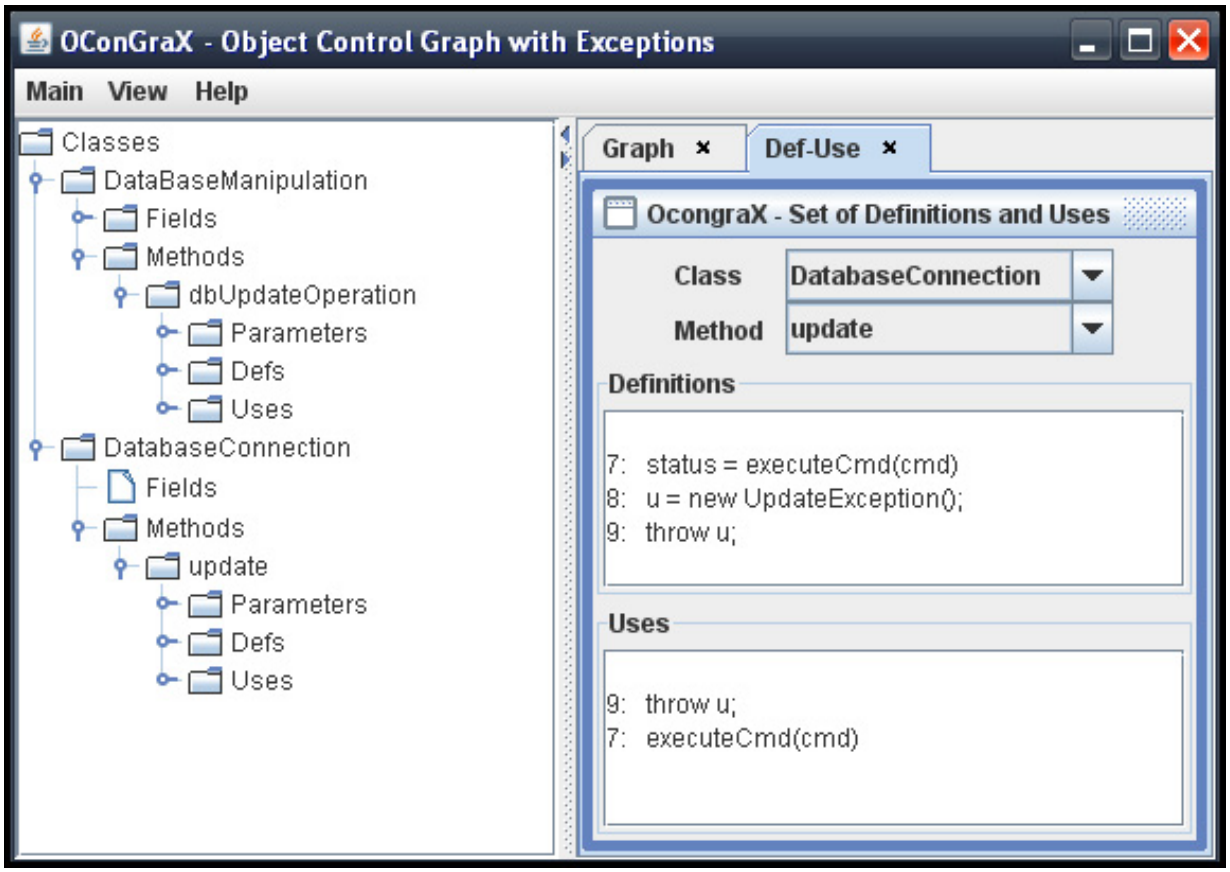


Figura 4.16: Visualização da aba de Definições-Usos, com as linhas de def-uso de objetos e exceções do método `update` da classe `DataBaseConnection`. No caso deste *screenshot*, a aba do Grafo já estava aberta, mas é importante verificar que é possível fechar a aba sempre que quiser, clicando no '*x*' existente ao lado do nome da aba.

4.5.4 Visualização do Grafo gerado pela OConGraX

Para visualizar o grafo, basta selecionar esta opção no menu. A aba do Grafo aparecerá em seguida, e sua visualização será gerada assim que o usuário selecionar a classe e o objeto (ou exceção) desejado.

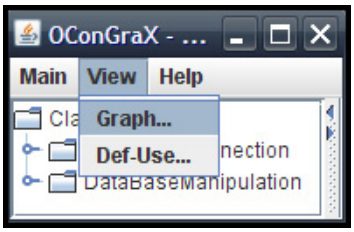


Figura 4.17: Para visualizar o grafo, selecione `View` → `Graph` no menu

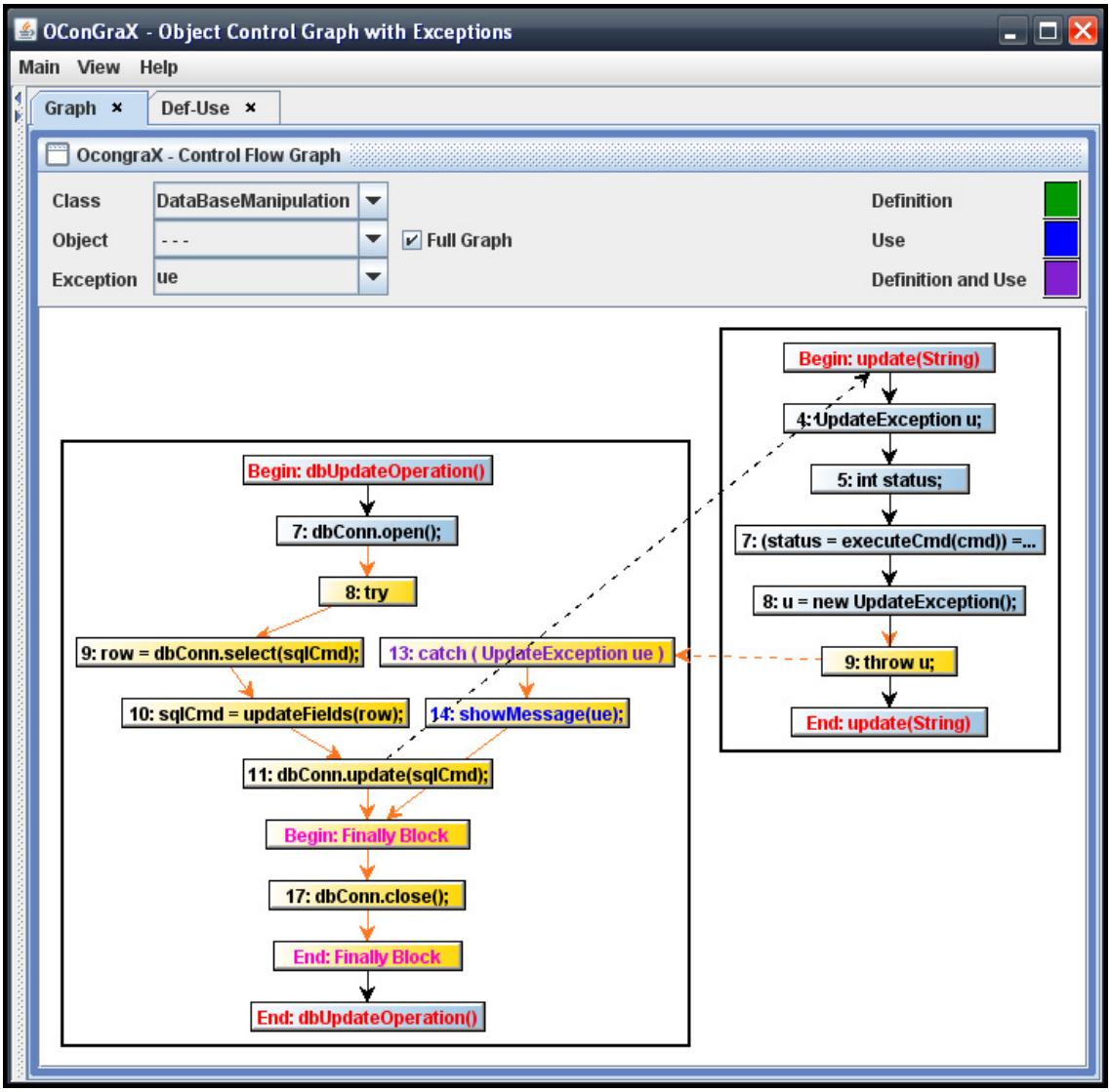


Figura 4.18: Visualização do grafo referente à exceção `ue` da classe `DataBaseManipulation`, com a opção "*Full Graph*" marcada. A legenda do grafo é explicada na seção 4.4.3

4.5.5 Salvando o grafo

Se após visualizar o grafo o usuário desejar salvá-lo como arquivo de imagem, basta ele ir no menu e clicar em `Main` → `Save Graph as Image`.

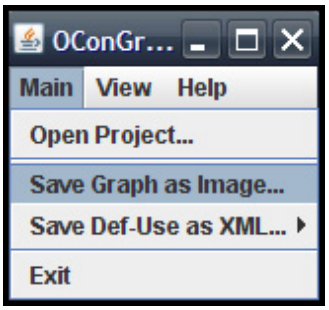


Figura 4.19: Para salvar o grafo, selecione `Main` → `Save Graph as Image`

Uma caixa de diálogo aparecerá. Nela, o usuário pode selecionar o diretório em que o arquivo será salvo e o nome do arquivo. Caso nenhuma extensão seja colocada no nome do arquivo, a OConGraX automaticamente gera um arquivo `jpg`. Senão, o usuário pode colocar qualquer extensão do conjunto suportado pela ferramenta.

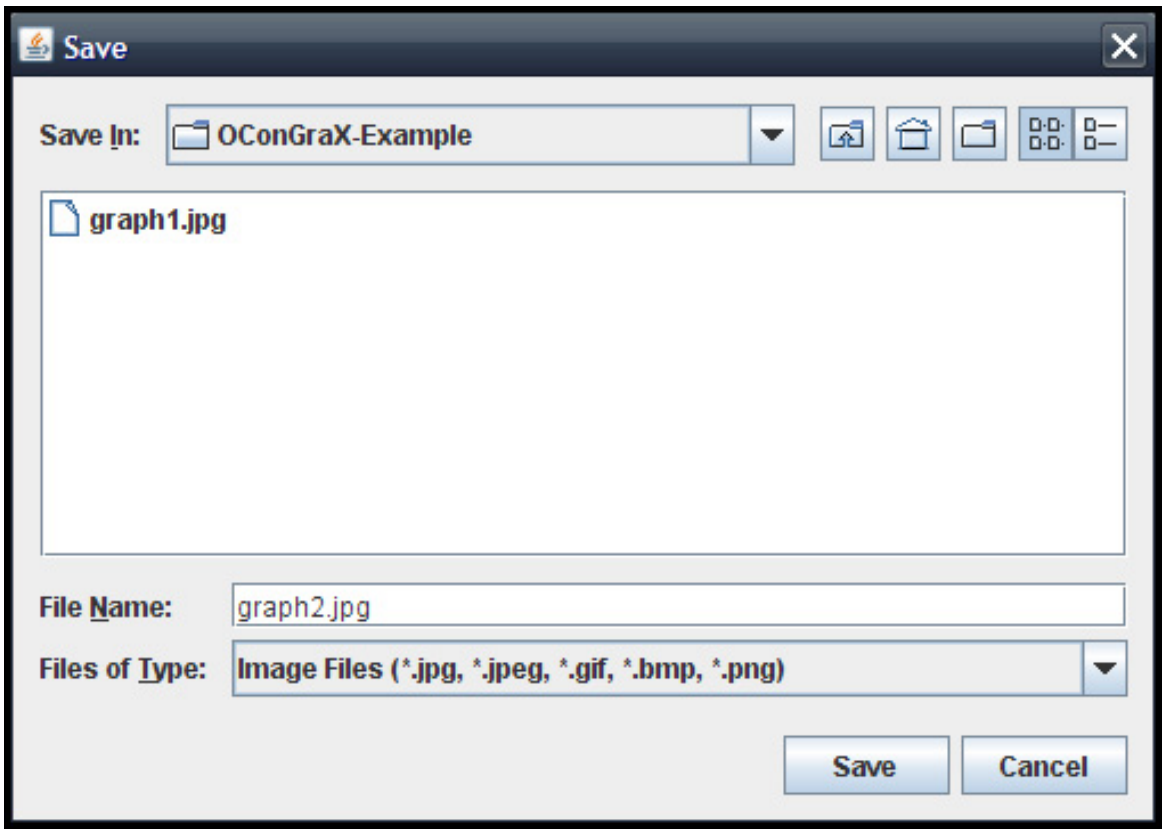


Figura 4.20: Caixa de Diálogo que aparece para salvar o grafo como arquivo de imagem. As extensões de arquivo suportadas pela ferramenta são listadas na linha abaixo do local em que se escreve o nome do arquivo a ser salvo.

4.5.6 Salvando as Definições e Usos num arquivo XML

Para salvar as definições e usos num arquivo XML, selecione no menu: `Main` → `Save Def-Use as XML`. Aparecerá duas opções: salvar as definições e usos com os pares de definição-uso de exceções ou sem os pares. Neste exemplo, a opção selecionada é a de salvar as definições e usos com os pares de def-uso de exceções.

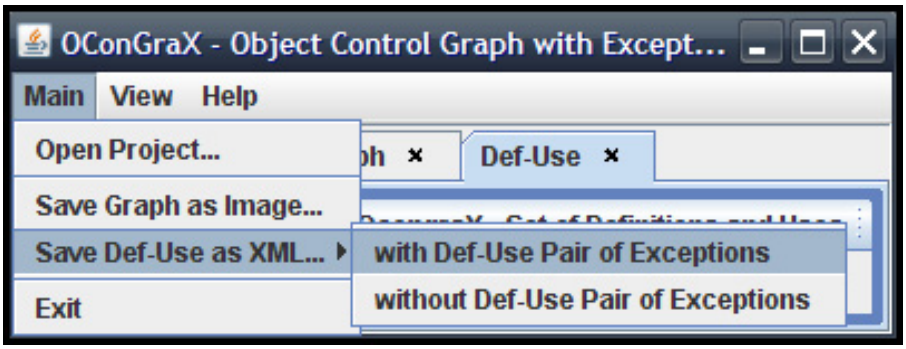


Figura 4.21: Selecionando no menu a opção de salvar as definições e usos de objetos e exceções, mais os pares de definição-uso de exceções, num arquivo XML

Selecionada a opção, aparecerá a caixa de diálogo, onde o usuário definirá o nome e a localização do arquivo XML a ser gerado pela OConGraX.

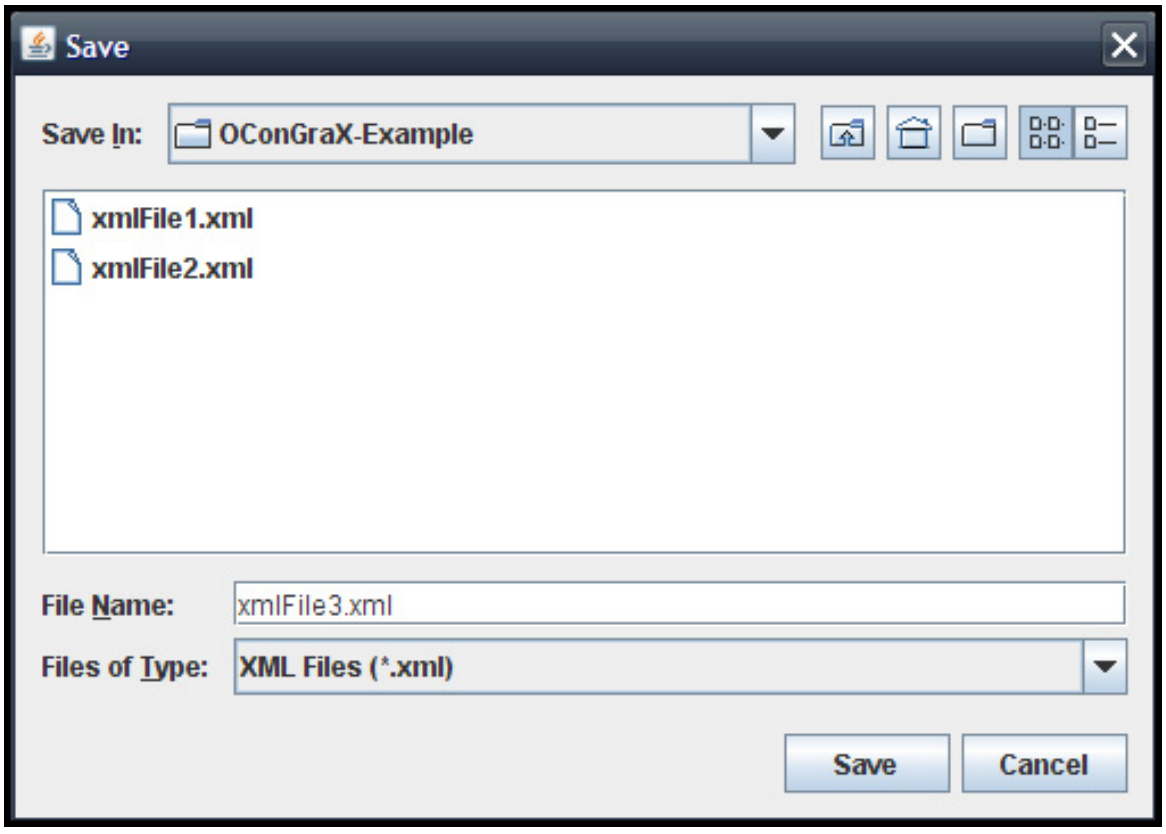


Figura 4.22: Caixa de Diálogo que aparece para determinar o nome e a localização do arquivo XML a ser gerado pela ferramenta

A seguir, um exemplo parcial do arquivo XML gerado pela OConGraX.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <elements>
  - <object>
      <classname>...</classname>
      <name>...</name>
      <def>...</def>
      ..
      <use>...</use>
      ..
    </object>
    ..
  - <exception>
      <classname>DataBaseManipulation</classname>
      <name>ue</name>
    - <pair>
        <def>13</def>
        <use>14</use>
      </pair>
      <def>13</def>
      <use>13</use>
      <use>14</use>
    </exception>
    ..
  </elements>
```

Figura 4.23: Exemplo parcial do arquivo XML gerado pela ferramenta

Capítulo 5

Conclusão

Neste capítulo, escreverei sobre os resultados finais deste trabalho, e sobre como a utilização da ferramenta nele desenvolvida pode facilitar o processo de testes de software escritos em linguagem Java.

5.1 Resultados

A OConGraX, produto final deste trabalho, realiza todas as funcionalidades descritas no início desta monografia (Capítulo 1). Suas principais características, somadas as funcionalidades da OConGra, são:

- Interface Gráfica de fácil interação com o usuário;
- Leitura e Análise de um Projeto Java, de modo a devolver:
  - Definições e Usos de Objetos e Exceções;
  - Pares de definição-uso de Exceções;
  - Grafo de Fluxo de Controle de Objetos (*OCFG*) com informações sobre as estruturas de tratamento de exceção (*try*, *catch*, *throw*, *finally*) utilizadas no projeto Java recebido como entrada.
- Salvar os dados de Definições e Usos de Objetos e Exceções num arquivo XML;
- Salvar o grafo gerado como arquivo de imagem.

5.2 Utilização do software desenvolvido

Com a OConGraX, informações como as definições e usos de objetos e exceções e o grafo de fluxo de controle de objetos com informações sobre exceções são obtidas automaticamente. Isto facilita muito a realização de testes estruturais em programas OO (como dito em 2.2), pois a automatização destes passos representa uma significativa diminuição nos custos do processo de testes do programa. Diminuindo-se os custos, testes como os que verificam o comportamento do programa em situações excepcionais deixam de ser negligenciados, o que acarreta no desenvolvimento de programas menos propensos a erros e mais confiáveis.

Além disso, permitir que os dados gerados sejam armazenados em arquivos representa uma grande vantagem, pois dessa forma é possível compartilhar as informações da OConGraX, reutilizando-as em outros aplicativos desenvolvidos com a finalidade de testar e verificar programas escritos em Java.

II. Parte Subjetiva

Capítulo 6

Desafios, Aprendizado e Planos futuros

Neste capítulo, relatarei minha experiência de fazer um trabalho de Iniciação Científica: os desafios, frustrações, e as displinas do curso que me forneceram o conhecimento necessário para a realização do projeto. Também falarei sobre meus planos para o futuro, tanto para a ferramenta aqui desenvolvida, quanto para minha vida acadêmica.

6.1 Desafios e Frustrações

Durante o desenvolvimento deste trabalho, posso dizer que enfrentei dois desafios principais: o primeiro era o de entender o código e a lógica do programa a ser estendido (OConGra); o segundo, foi o de conciliar o trabalho de conclusão de curso com os estudos e tarefas das demais disciplinas cursadas neste ano de 2007.

Para passar por estes dois desafios, pude contar com a ajuda da minha orientadora, professora Ana Cristina Vieira de Melo, e de seus outros orientandos que desenvolvem trabalhos na mesma linha de pesquisa (área de testes, de Engenharia de Software): Kleber da Silva Xavier e Rodrigo Della Vittoria Duarte. Desde o início do desenvolvimento do trabalho, em janeiro deste ano até sua conclusão, mantivemos reuniões quinzenais para verificar o andamento do trabalho que cada um estava desenvolvendo e trocar idéias e sugestões para eventuais problemas que surgiam.

Uma outra grande ajuda que recebi foi do próprio autor da OConGra, Paulo Roberto de A. F. Nunes, com quem mantive contato durante o ano inteiro. Graças a isso, o entendimento do código da ferramenta foi mais rápido, o que me permitiu programar com maior rapidez a extensão da OConGra, OConGraX.

Apesar de ter conseguido superar os dois desafios citados, não consegui concluir completamente tudo o que eu havia planejado inicialmente, como acrescentar uma documentação para o usuário, implementar os pares de definição-uso de objetos e corrigir alguns "bugs" da OConGra. E estas tarefas incompletas seriam as minha únicas frustrações.

6.2 Disciplinas

As disciplinas que cursei ao longo da minha graduação, e que eu considero como as mais importantes para o desenvolvimento deste trabalho são:

- **MAC0122 - Princípios de Desenvolvimento de Algoritmos:** por ser a disciplina que fornece a base para escrever algoritmos de maior complexidade.
- **MAC0323 - Estruturas de Dados:** por apresentar formas de representação de dados mais eficientes e uma lógica mais aprofundada de algoritmos, o que foi muito útil paraa implementação do código desenvolvido no trabalho.
- **MAC0211 - Laboratório de Programação I:** por apresentar técnicas de implementação iniciais para grandes projetos, e a linguagem LaTeX 2_ε de edição de textos, utilizada nesta monografia.
- **MAC0242 - Laboratório de Programação II:** por apresentar técnicas mais avançadas de implementação para projetos grandes de programação.
- **MAC0328 - Algoritmos em Grafos:** por ser a disciplina em que obtive o conhecimento teórico para manipular e construir grafos. Tal conhecimento pôde ser aplicado na prática para a resolução de um problema real.
- **MAC0332 - Engenharia de Software:** por apresentar a teoria e os métodos de desenvolvimento de software. Para este trabalho, o conhecimento obtido nesta disciplina na parte de testes foi fundamental.
- **MAC0441 - Programação Orientada a Objetos:** por apresentar técnicas de programação em linguagens orientadas a objeto. Isto foi muito útil, uma vez que o código foi todo escrito numa linguagem OO, Java.
- **MAC0446 - Princípios de Interação Homem-Computador:** por apresentar métodos e técnicas para design de interface, de modo a facilitar a interação do usuário com o software. Com o conhecimento obtido nesta disciplina, realizei alterações para melhorar a usabilidade da ferramenta desenvolvida neste trabalho.

6.3 Planos Futuros

6.3.1 Ferramenta: OConGraX

Apesar do produto final, OConGraX, ter atingido todas as metas pretendidas no início deste trabalho, sempre é possível melhorar. Por isso, acredito que os novos objetivos a serem alcançados com a ferramenta seriam:

- Melhoramento na interface e na usabilidade;
- Refatoração do código;
- Acréscimo de novas funcionalidades, como a implementação dos pares de definição-uso de objetos e de uma nova extensão para que ela possa ser utilizada em programas escritos em outras linguagens OO que não Java.

6.3.2 Vida Acadêmica

A experiência de realizar uma Iniciação Científica que resultou neste trabalho foi muito enriquecedora e estimulante. Por isso, pretendo continuar a trabalhar na área de Engenharia de Software sob a orientação da professora Ana Cristina Vieira de Melo, possivelmente numa pós-graduação (Mestrado) aqui no IME-USP.

Referências Bibliográficas

[1] Bloch, Joshua: *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

[2] Chen, Mei Hwa e Kao, Howard M.: *Testing Object-Oriented Programs – An Integrated Approach*. In *ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering*, página 73, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0443-4.

[3] Deitel, Harvey M. e Deitel, Paul J.: *Java: Como Programar*. Bookman, 2004.

[4] Eckel, Bruce: *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002, ISBN 0131002872.

[5] Eclipse. Disponível em: <`http://www.eclipse.org/`>. Acesso em: 02 mar. 2007.

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

[7] Garcia, Alessandro F., Rubira, Cecília M. F., Romanovsky, Alexander, e Xu, Jie: *A comparative study of exception handling mechanisms for building dependable object-oriented software*. The Journal of Systems and Software, 59(2):197–222, 2001. Disponível em: <`http://citeseer.ist.psu.edu/garcia01comparative.html/`>. Acesso em: 22 mar. 07.

[8] Gosling, James, Joy, Bill, Steele, Guy, e Bracha, Gilad: *The Java Language Specification*. Addison-Wesley, 2005.

[9] JGraph. Disponível em: <`http://www.jgraph.com/`>. Acesso em: 21 out. 2007.

[10] Microsoft Corporation: *What is SAX?* Disponível em: <`http://msdn2.microsoft.com/en-us/library/ms754682.aspx`>. Acesso em: 19 nov. 2007.

[11] Nunes, Paulo Roberto de Araújo França: *Ocongra - Uma ferramenta de apoio ao teste Orientado a Objetos*. Monografia. Disponível em: <`http://www.ime.usp.br/~prnunes/monografia.html`>. Acesso em: 21 out. 2007.

[12] Nunes, Paulo Roberto de Araújo França e Melo, Ana Cristina Vieira de: *OConGra - Uma Ferramenta para Geração de Grafos de Controle de Fluxo de Objetos*. In *Anais do 18º Simpósio Brasileiro de Engenharia de Software*, Brasília, Brasil, 2004. 18º SBES.

[13] Pereira, David Paulo: *Um* framework *para coordenação do tratamento de exceções em sistemas tolerantes a falhas*. Dissertação de Mestrado, Universidade de São Paulo, São Paulo, Brasil, 2007.

[14] PRESSMAN, Roger S.: *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001, ISBN 0072496681.

[15] RECODER. Disponível em: <`http://recoder.sourceforge.net/`>. Acesso em: 21 out. 2007.

[16] ROBILLARD, Martin P. e MURPHY, Gail C.: *Analyzing exception flow in Java programs*. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, páginas 322–337, London, UK, 1999. Springer-Verlag, ISBN 3-540-66538-2.

[17] ROMANOVSKY, Alexander, DONY, Christophe, KNUDSEN, Jørgen Lindskov, e TRIPATHI, Anand: *Exception Handling in Object Oriented Systems*. Lecture Notes in Computer Science, 1964:16–, 2001. Disponível em: <`http://citeseer.ist.psu.edu/romanovsky01exception.html`>. Acesso em: 22 mar. 07.

[18] SEDGEWICK, Robert: *Algorithms in C, Part 5: Graph Algorithms*. Addison-Wesley, 2001.

[19] SINHA, Saurabh e HARROLD, Mary Jean: *Analysis of Programs with Exception-Handling Constructs*. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, página 348, Washington, DC, USA, 1998. IEEE Computer Society, ISBN 0-8186-8779-7.

[20] SINHA, Saurabh e HARROLD, Mary Jean: *Criteria for Testing Exception-Handling Constructs in Java Programs*. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, página 265, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0016-1.

[21] SINHA, Saurabh e HARROLD, Mary Jean: *Analysis and Testing of Programs with Exception Handling Constructs*. IEEE Trans. Softw. Eng., 26(9):849–871, 2000, ISSN 0098-5589.

[22] SUN MICROSYSTEMS: *Java Technology*. Disponível em: <`http://java.sun.com/`>. Acesso em: 10 nov. 2007.

[23] SUN MICROSYSTEMS: *The Java Tutorials*. Disponível em: <`http://java.sun.com/docs/books/tutorial/`>. Acesso em: 02 out. 2007.

[24] SUN MICROSYSTEMS: *JDC Tech Tips: June 27, 2000*. Disponível em: <`http://java.sun.com/developer/TechTips/2000/tt0627.html`>. Acesso em: 19 nov. 2007.

[25] W3 SCHOOLS: *XML Tutorial*. Disponível em: <`http://www.w3schools.com/xml/default.asp`>. Acesso em: 19 nov. 2007.

[26] XAVIER, Kleber da Silva: *Estudo sobre Redução do Custo de Testes através da Utilização de Verificação de Componentes Java com Tratamento de Exceções*. Qualificação de Mestrado, 2007.

[27] XML.ORG: *XML.org - Resources*. Disponível em: <`http://www.xml.org/xml/resources_cover.shtml`>. Acesso em: 19 nov. 2007.