Monografia de Conclusão do Curso de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Ocongra - Uma ferramenta de apoio ao teste Orientado a Objetos.

Aluno: Paulo Roberto de Araújo França Nunes
Orientadora: Ana Cristina Vieria de Melo

Índice

1 Introdução

1.1 Os problemas de teste

O teste de programas é uma fase complexa no desenvolvimento de software. Em sistemas complexos, os quais envolvem diversas atividades de produção, a probabilidade de encontrarmos falhas na programação é muito alta [5]. A atividade de testes [3] consiste de uma análise dinâmica do produto, sendo relevante para a identificação e eliminação de erros que persistem. Assim, exige-se muita cautela por parte do testador, pois qualquer alteração no sistema pode acarretar mudanças em outras partes, o que pode causar falhas.

O principal objetivo do teste é encontrar casos que descubram de maneira sistemática diferentes classes de erros e que tenham custos de tempo e de esforços mínimos. Com isso, técnicas e critérios de testes passam a ter papéis fundamentais na tarefa de teste.

1.2 O problema no teste Orientado a Objetos

Para diferentes paradigmas de desenvolvimento com diferentes características, são necessárias técnicas de testes que abordem essas características para que erros sejam descobertos.

O paradigma de Orientação a Objetos (OO) apresenta diferenças em relação ao de programação procedural. Requisitos de teste aplicáveis a sistemas procedurais tornam-se inadequados para os testes de sistemas Orientados a Objetos - os requisitos mudam de acordo com o paradigma.

Embora o teste OO apresente vantagens em relação ao teste procedimental, a realização de teste é ainda um dos maiores problemas encontrados na produção de códigos OO. As técnicas tradicionais, em geral, são insuficientes para detectar falhas. Em virtude dessas dificuldades, adaptações de técnicas tradicionais para testes de programas OO e novas estratégias de teste têm

sido bastante discutidas.

1.3 As técnicas existentes

Alguns critérios de teste existentes para o paradigma prodedural foram adaptados para o OO. Existem atualmente diversas técnicas para testes OO, como por exemplo, baseado em estado [2], teste incremental [4], todos os caminhos [1], baseado no fluxo de objetos [5]. O teste baseado em estados observa os valores armazenados na representação dos objetos. A técnica de teste enfatiza a parte hierárquica dos objetos a fim de testar grupos de classes relacionadas, reutilizando informações de testes das superclasses para direcionar os testes das subclasses. Já o critério de todos os caminhos faz diversas verificações nas alterações dos valores dos objetos. O teste baseado no fluxo de objetos será detalhado na Seção 2.

2 Uma técnica de teste OO

Estudos demonstram que é necessária uma avaliação global do sistema ao invés de análises e testes em pequenas partes do software [7]. A análise de fluxo de objetos segue uma estratégia de validação que se baseia na cobertura total de estados de transformação de cada objeto do sistema. Tais transformações são denotadas pela redefinição de valores para os atributos dos objetos através da passagem de mensagens entre eles.

Para monitorar o comportamento de cada um dos objetos são analisados todos os pares de definição e uso, ou seja, os locais onde cada objeto é definido ou usado.

- *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.

Em [6], Chen e Kao definem cinco passos para a construção de um grafo de fluxo de controle de objetos (GFCO) e identificação dos pares de definição-uso (*du*). O GFCO consiste basicamente na ligação entre as classes do sistema e seus métodos. Os cinco passos são os seguintes:

1. Criar o conjunto de definição e uso para cada método em cada classe. O conjunto de definição de um método M contém todos os atributos definidos por M e o conjunto de uso de M contém todos os atributos cujos valores são referenciados em M;
2. Construir o GFCO inicial (somente nós com classes e métodos);
3. Adicionar ao grafo as arestas intra-métodos. Baseado no conjunto de definição e no conjunto de uso criados no passo 1, identificam-se as ligações dentro de um mesmo método.
4. Adicionar ao grafo as arestas inter-métodos. Ainda baseado no conjunto de definição e no conjunto de uso criados no passo 1, identifica-se as ligações referenciadas em outros métodos.
5. Selecionar os pares *du*.

A cobertura de todos os pares *du* aumenta a probabilidade de se encontrar falhas no programa.

Por isso, é necessário identificar dois tipos de pares de objetos: intra-métodos (referenciados dentro de um mesmo método) e inter-métodos (usados ou definidos fora do método). Para identificar esses pares é utilizado o grafo de fluxo de controle de objetos. Assim torna-se muito mais fácil a visualização do sistema e também a identificação dos pares *du*.

Porém, a criação manual dos grafos está sujeita a falhas. Assim, há uma desestimulação para a aplicação da técnica devido às dificuldades. Como vimos anteriormente, um dos objetivos dos testes é encontrar falhas com o mínimo de esforço e o mínimo de custo possível. Diante da necessidade de automatização dessa tarefa tomamos como objetivo desta pesquisa a criação da ferramenta Ocongra (Object Control Graph): *Uma ferramenta de apoio ao teste Orientado a Objetos*. A ferramenta disponível faz com que os responsáveis pelos testes se concentrem na criação de casos de testes, ao invés de gerarem os grafos manualmente, tornando assim a tarefa de testes menos demorada e trabalhosa [11].

3 Uma ferramenta de apoio

A Ocongra [11] recebe como entrada um sistema (escrito em Java) e tem como saída o grafo de de fluxo controle de seus objetos. O usuário, através de interface gráfica, designa o local onde estão localizados os arquivos fonte do sistema. Após a análise de cada classe do sistema, a ferramenta produz o grafo e o conjunto de pares *du*.

3.1 Geração do Grafo

A construção do GFCO de um dado sistema obedece aos mesmos passos sugeridos em [6]. Dessa forma, a ferramenta foi desenvolvida com base nesses passos. A seguir serão descritas as fases executadas pela Ocongra para a geração do GFCO e o conjunto de pares de definição e uso.

1. Leitura e Identificação do Projeto: o usuário especifica o local do projeto. Aqui, o programa recebe como entrada de dados um sistema composto por classes e identifica cada classe, seus métodos e heranças. Os diretórios serão identificados pelo usuário e a rotina identifica as classes métodos e suas relações no sistema.
2. Construção do GFCO com as classes-mãe: baseado nos resultados da fase anterior, será construído o grafo contendo a relação entre cada classe do sistema lido na entrada. Assim, essas classes terão suas chamadas de métodos denotadas.
3. Identificação das definições-usos através do GFCO: com o grafo definido, as mudanças de estado dos objetos poderão ser analisadas com maior precisão. Analisando o grafo, o software identificará cada definição-uso utilizada pelas classes do sistema;
4. Atualização do GFCO baseada nas definições-usos: a identificação das definições e usos (passo anterior) possibilita a implementação de alterações no grafo obtido no passo 2, criando assim um novo grafo.
5. Geração gráfica dos resultados obtidos: o grafo encontrado após as análises do sistema é desenhado e exibido de forma gráfica ao usuário. Para exibir na tela os grafos obtidos utilizamos o framework JGraph [10].

5 Arquitetura da Ferramenta

O sistema está sendo desenvolvido em Java com a utilização da ferramenta Eclipse [8]. Possui aproximandamente 3500 linhas de código. Pressupõe-se que a entrada da ferramenta seja um conjunto de arquivos .java que compõem um sistema livre de erros de sintaxe ou compilação.

A arquitetura da Ocongra (Figura 1) consiste de uma interface gráfica, um Parser Java e um Gerador de Grafo. O usuário informa a localização de todos os arquivos do sistema. A ferramenta então aciona o Parser Java e com o resultado obtido aciona o Gerador do Grafo que devolve o

GFCO do sistema lido.

O Parser Java processa os arquivos .java e armazena uma estrutura de dados organizada de forma a facilitar a geração do GFCO.
O Gerador do Grafo percorre a estrutura criada pelo parser e, seguindo os passos citados anteriormente, constrói o GFCO.

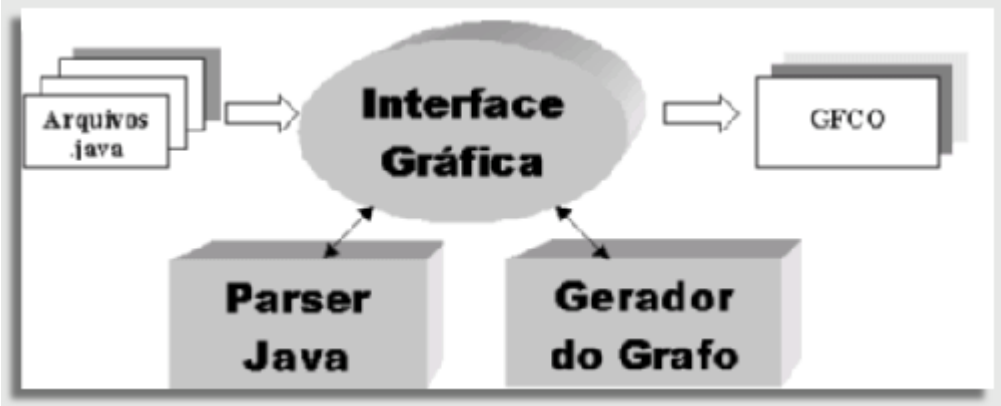

Figura 1 - Arquitetura da ferramenta desenvolvida

6 Um Exemplo de Uso

Para exemplificar o funcionamento da ferramenta, consideremos a aplicação [1] ilustrada na Figura 2. A classe *Shape* tem duas classes derivadas: *Square* e *Circle*. Alguns dos métodos desta aplicação não foram exibidos, pois não são relevantes pra o demonstração do critério (método *draw* por exemplo).

A classe *Frame* contém um objeto 'shape_obj' *Shape* - como pode ser visto na Figura 2 - que é utilizado para exibir os gráficos criados.

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

Figura 2.1 - Um programa Java simples - Classe Frame.

```
1  public abstract class Shape{
2      int perimeter;
3
4      public void set_parm(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 }
```

Figura 2.2 - Um programa Java simples - Classe Shape.

```
1  public class Square extends Shape{
2
3      protected int x,y;
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 }
```

Figura 2.3 - Um programa Java simples - Classe Square.

```
1  public class Circle extends Shape{
2
3      int x,y;
4      double radius;
5
6      public Circle(int x, int y, int init_perimeter){
7          this.x=x;
8          this.y=y;
9          perimeter=init_perimeter;
10         this.radius = init_perimeter /(2*3.14);
11     }
12
13     public int draw(){
14         System.out.println("Draw Circle");
15         return 0;
16     }
17 }
```

Figura 2.4 - Um programa Java simples - Classe Circle.

```
1  public class teste{
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 2.5 - Um programa Java simples - Classe teste.

Inicialmente, o usuário informa a localização do projeto Java (Figura 3). Em seguida a ferramenta exibe a estrutura de todas as classes identificadas: nome das classes e suas heranças, seus respectivos métodos e quais os números de linhas em que eles aparecem (Figura 4).

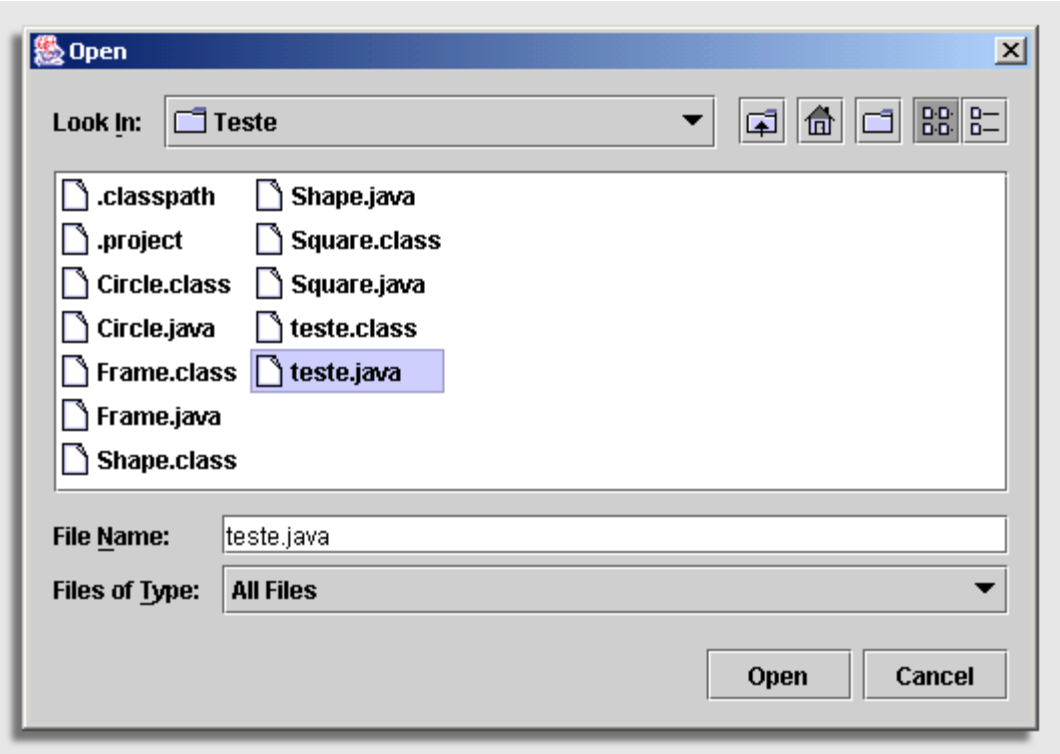

Figura 3 - Tela de abertura e indicação do projeto Java a ser analisado.

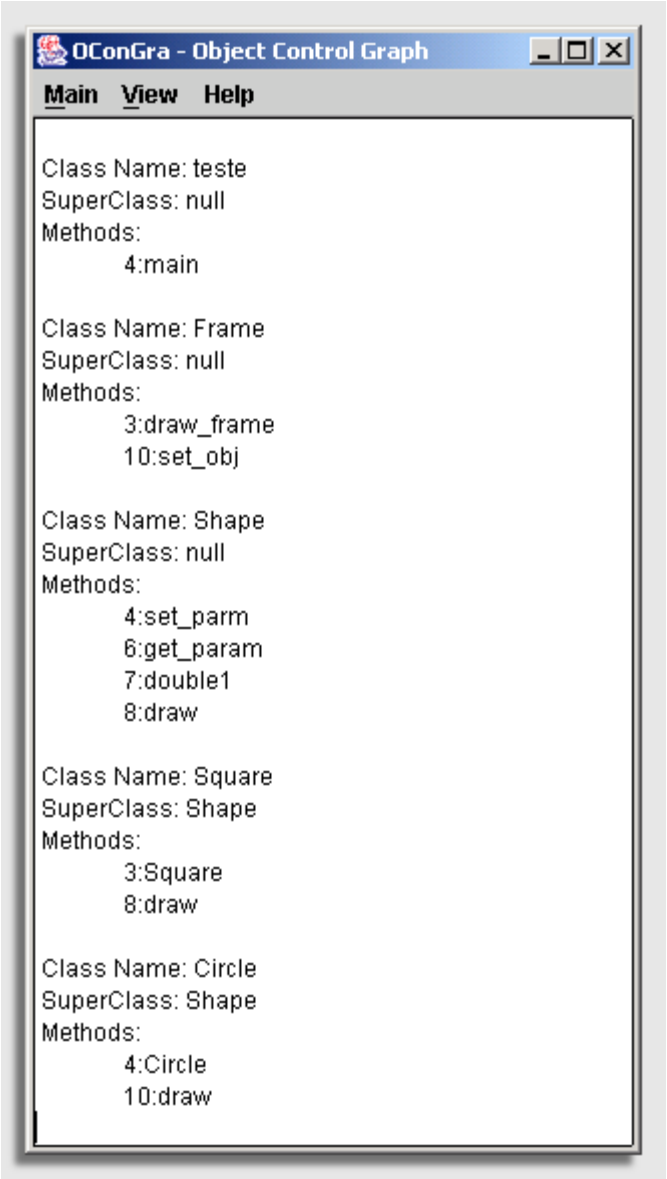

Figura 4 - Visualização de informações das classes identificadas.

A Figura 4 mostra as definições e os usos presentes no método *draw_frame* da classe *Frame*, conforme enunciado anteriormente. Cada linha que possui um objeto é verificada a ocorrência de uma definição ou de um uso.

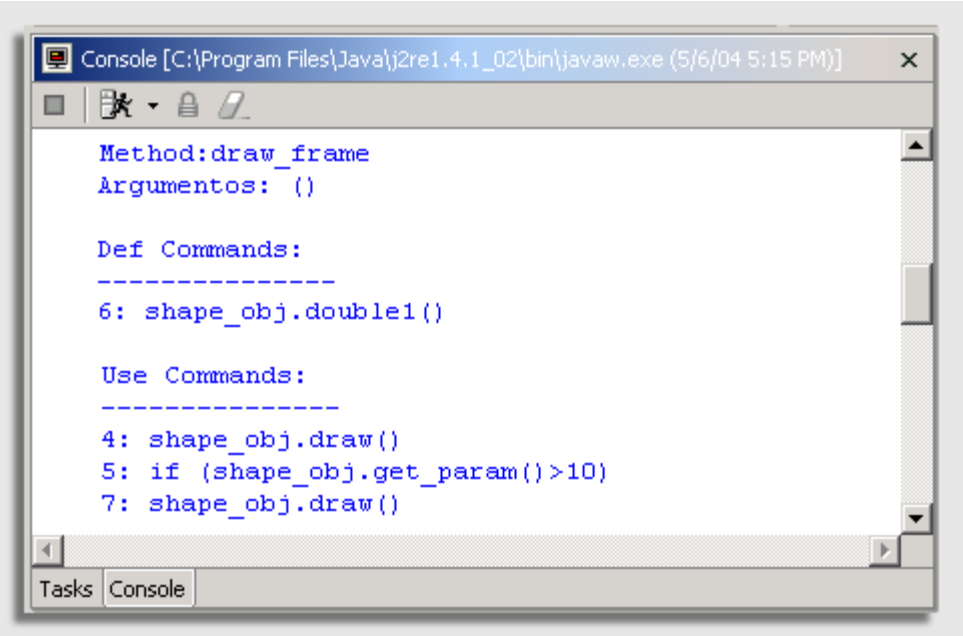

*Figura 4 - Definições e Usos do método **draw_frame** da classe **Frame**.*

Já a Figura 5 mostra o grafo correspondente ao exemplo. Após identificar a estrutura do sistema, o programa constrói esta saída gráfica.

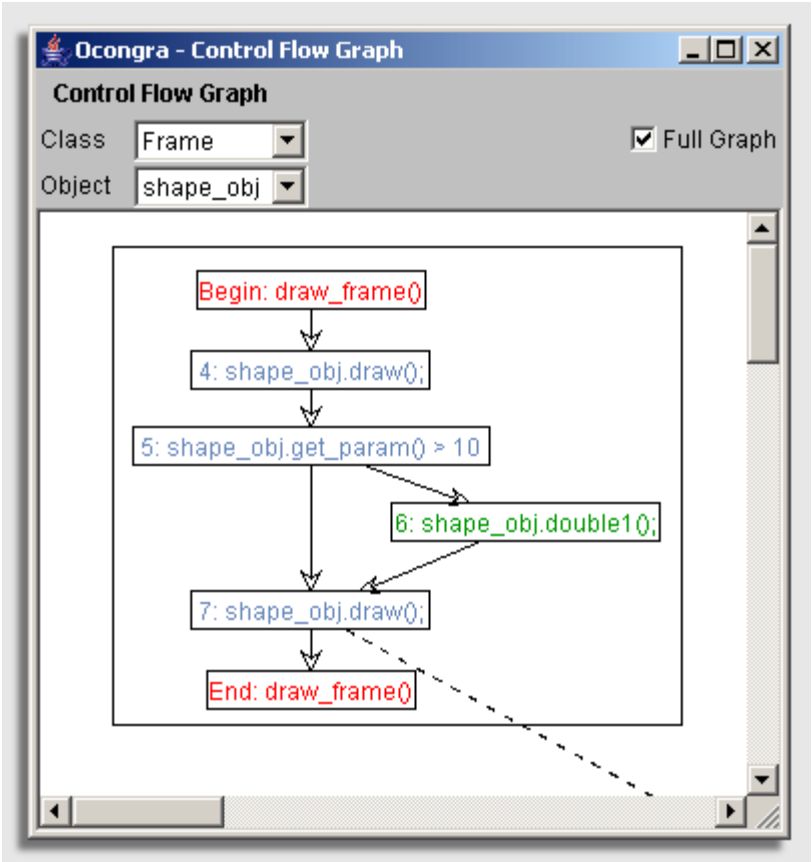

Figura 5 - Grafo de fluxo de controle da aplicação.

Após identificar as definições-uso e o grafo, é necessário exercitar os pares pelo menos uma vez. Para isso são criados os seguintes testes:

- (1, 11, 2, 2) - exercita os pares (12, 4), (12, 5), (12, 6), (6, 7) da classe *Frame*

- (1, 3, 2, 2) - exercita os pares (12, 4), (12, 5), (12, 6), (13, 7) da classe *Frame*
- (2, 15, 2, 2) - exercita os pares (15, 4), (15, 5), (15, 6), (6, 7) da classe *Frame*
- (2, 5, 2, 2) - exercita os pares (15, 4), (15, 5), (15, 6), (15, 7) da classe *Frame*

Os quatro testes acima exercitam todas as instanciações do objeto 'shape_obj'.

7 Aprendizado com o desenvolvimento do projeto

Participar do desenvolvimento de um sistema que aplicasse a fundo o conteúdo (ou pelo menos parte dele) visto no curso foi uma experiência muito interessante. Por exemplo, eu nunca imaginaria que algum dia eu aplicaria busca binaria fora de um EP!! Brincadeiras a parte, o aprendizado maior foi lidar com o problema de entregar as fases no prazo e ainda ter que estudar as disciplinas. A forma de organizar o tempo e identificar as prioridades foi um grande desafio no decorrer da pesquisa. Além disso, muitos assuntos pouco explorados no curso (como o próprio tema da pesquisa - teste de software) foram estudados exaustivamente. Assim, a importância dos testes e de que forma eles afetam diretamente a qualidade final de um produto foi ganhando destaque na minha mente. Como no curso isso é visto supercialmente, a inciação científica foi de extremo valor para o aprendizado. Mas, o melhor de todo o desenvolvimento foi ter participado do *18° Simpósio Brasileiro de Engenharia de Software*, onde, além de apresentar o presente trabalho, foram vistos trabalhos de outras faculdades e foi possível aprender muito, tanto academicamente como pessoalmente.

7.1 Relação entre Iniciação Científica e Disciplinas do Curso

O desenvolvimendo do software foi baseado em tópicos vistos na disciplina de *Engenharia de Software*. Toda a documentação, a metodologia de desenvolvimento, técnicas de testes fazem parte dos tópicos dessa disciplina. Uma das técnicas estudadas em aula foi a de grafo de controle de fluxo de objeto, a qual é tema desta pesquisa. Foram necessários estudos mais profundos sobre a técnica utilizada e pesquisas sobre aplicações mais específicas. Além disso, outras técnicas de testes também foram pesquisadas com a finalidade de aprimorar a aplicação da mesma.

A manipulação da linguagem de programação Java é estudada em *Laboratório de Programação II*. A base fornecida foi essencial para a implementação de alguns algoritmos. Com o decorrer do projeto, o conhecimento da linguagem é aprimorado ao surgirem situações nas quais são necessárias pesquisas de novas soluções.

Para localizar as palavras, foi necessário implementar algumas buscas de padrões e utilizar algumas já implementadas na própria linguagem. Estudos realizados na disciplina de *Linguagens Formais e Autômatos* auxiliaram muito na resolução do problema.

As estruturas utilizadas para armazenar o conteúdo dos arquivos '.java' e também o grafo, são modificações das estruturas vistas nas disciplinas *Estrutura de Dados* e *Algoritmos em Grafos*.

Com a utilização de frameworks desenvolvidos por outros grupos foi necessário extender o conhecimento sobre suas vantagens e maneiras de otimizar a aplicação de reutilização de componentes, visto na disciplina de Tópicos de Programação Orientada a Objetos.

Infelizmente, a área de engenharia de software e técnicas para elaboração e contrução de softwares não são muito aprofundadas no curso. Poderia haver mais disciplinas obrigatórias relacionadas a padrões de desenvolvimento. Não no sentido de "ensinar a programar" ou "ensinar a produzir software". Mas a variedade de formas de desenvolvimento existentes e a carência de técnicas consolidadas poderiam ser exploradas.

8 Dificuldades Encontradas

Naturalmente, dificuldades apareceram no decorrer de cada etapa do projeto, as quais contribuiram muito para o aprendizado e também para o amadurecimento do sistema como um todo.

Para demonstrar melhor as dificuldades encontradas, dividiremos conforme o módulo de desenvolvimento do projeto.

1. Leitura e Identificação do Projeto:
   O reconhecimento dos elementos da linguagem Java pertencentes a um arquivo '.java' apresentou dificuldades, principalmente nas definições da própria linguagem. Por exemplo, identificar os comandos dos métodos (juntamente com suas utilizações) e classificá-los devidamente (declaração de variável, chamada de função ou métodos, comando condicional, atribuição, etc).
   Mas, com a utilização do framework Recoder [9], os elementos foram facilmente reconhecidos e utillizados na ferramenta. A dificuldade agora passou a ser readaptar as funcionalidades já implementadas para o novo pacote;
2. Construção do GFCO com as classes-mãe:
   Seguir cada comando da rotina lida e decidir qual o caminho seguir, e assim montar a estrutura de dados adequada, até agora, foi o ponto de maior complexidade no decorrer do projeto. Praticamente é necessário seguir a ordem dada pela codificação. Em alguns casos são necessárias muitas rotinas para tratar um único comando. O desenvolvimento de uma estrutura de dados que armazenasse os super nós, nós e arestas tomou bastante tempo, principalmente, por ser uma parte que afeta diretamente as demais fases do projeto;
3. Identificação das definições-usos através do GFCO:
   Basicamente, o desenvolvimento do algoritmo para identificação dos pares e juntamente alguma validação para o algoritmo foi a maior dificuldade na construção deste módulo.
4. Atualização do GFCO baseada nas definições-usos:
   Tratar os casos particulares de atributos (ou nomes de classes, métodos) que possuem palavras reservadas como subpalavras, atributos, métodos, etc, que eventualmente apareçam dentro de algumas strings (que no código fonte apareceriam entre aspas duplas ou simples) tem grau de complexidade elevado. O principal problema foi identificar esta característica e assim implementar uma rotina que as contemplassem;
5. Geração gráfica dos resultados obtidos:
   O framework utilizado não disponibiliza rotinas de posicionamento automático na tela. Assim, foi necessário implementar esta rotina para que não houvessem sobreposição dos grafos (vértices e suas arestas).

Embora todas essas fases tenham sido finalizadas com sucesso, aproximadamente na metade da pesquisa, eu via o tempo passar e um software desenvolvido com inúmeras restrições (principalmente com relação ao parser Java) e incontáveis bugs. Felizmente, conversando com amigos do curso, encontramos um framework que resolvia quase todos os meus problemas. Bem, quase todos. Algumas adaptações foram necessárias para que pudesse ser utilizado de forma correta no projeto.

Assim, depois de muito pesquisar e perguntar aos mais experientes nos devidos assuntos, o resultado aos poucos foi se definindo e com isso, as dificuldades foram uma a uma superadas.

9 Conclusões

Além da geração do grafo, a ferramenta auxilia a geração de dados para testes: a cobertura total dos casos de definição e uso de objetos (segundo a técnica [9]). A automatização desta fase de testes reduz consideravelmente o tempo de teste, visto que a geração manual dos grafos é complexa.

O apoio ao teste é a sua utilização prioritária. Porém, podemos utilizar os grafos resultantes no entendimento e análise de programas. Como o grafo das classes são gerados e a definição e uso dos objetos são identificados, a análise sobre a relação de dependência entre objetos é facilitada assim como a redefinição dos objetos. Podemos também com o auxílio do grafo verificar pontos no programa em que há maior fluxo de informações, por exemplo.

Pretendemos ainda estender a ferramenta para auxiliar a geração de casos de teste. Segundo os estudos realizados até o momento, temos a expectativa de obter uma geração semi-automática de casos de teste. No exemplo visto, o programa poderia sugerir as entradas para exercitar os pares *du* encontrados.

Também pretendemos desenvolver uma extensão da ferramenta que permita a análise e geração do GFCO para outras linguagens OO.

Com relação ao curso de Ciência da Computação, foi de muita qualidade e de grande utilidade para que os objetivos dessa pesquisa fossem atingidos. Talvez, se o enfoque em engenharia de software fosse maior, muitas das coisas que até agora tenho dificuldades (mesmo depois de muito pesquisar) pudessem ter sido melhor exploradas e consequentemente o aprendizado seria maior. Porém, as dificuldades fazem parte do processo e apenas contribuem para o crescimento pessoal, técnico e profissional.

Agradecimentos

Ao IME pelo suporte fornecido e ao CNPq pelo apoio financeiro.
A Leandro C. Prudente pelo fornecimento dos fontes utilizados como exemplo nesta monografia.
A prof. Ana pelo incentivo em todo o projeto e pela ajuda na revisão dos textos ;).
A todos que de alguma forma colaboraram para a conclusão deste projeto.

E principalmente ao Pai.

"Porque és a minha ajuda, canto de alegria à sombra das tuas asas".
Sl 63:7

10 Referências

[1] CHAIM, M. L.. Depuração de Programas Baseada em Informação de Teste Estrutural. *PhD thesis*, Faculdade de Engenharia Elétrica e de Computação da Universidade de Campinas, Novembro/2001.

[2] PINTO, I. M. and PRICE, A. M. A.. Um sistema de apoio ao teste de programas orientados a objetos com uma abordagem reflexiva. In *12° Simpósio Brasileiro de Engenharia de Software*, pages 87-102, 1998.

[3] BARBOSA, Ellen F., VINCENZI, Auri M. R. and MALDONADO, José C.. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. In *12° Simpósio Brasileiro de Engenharia de Software*, pages 103-120, 1998.

[4] HARROLD, M. J., MCGREGOR, J. D. and FITZPATRICK, K. J.. Incremental testing of object-oriented class structures. In *international Conference on Software Engineering, pages 68-80. IEEE Computer Society Press, 1992.*

[5] PRESSMAN, R. S.. *Software Engineering*. McGraw-Hill, 2001.

[6] CHEN, M. and KAO, H. M. Testing Object-Oriented Programs - An integral aproach. In *Proccedings of the Tenth International Symposium on Software Reliability Engineering. IEEE Computer Society Press, 1999.*

[7] ALEXANDER, Roger T. and OFFUTT Jefferson A.. Criteria for Testing Polymorphic Relationships. In *11th International Symposium on Software Reliability Engineering. pages 15-23, 2000.*

[8] PROJETO ECLIPSE. Disponível no site Eclipse, URL: http://www.eclipse.org. Consultado em 15/08/2003.

[9] PROJETO RECODER. Disponível no site Sourceforge, URL: http://recoder.sourceforge.net. Consultado em 16/05/2004.

[10] PROJETO JGRAPH. Disponível no site Sourceforge, URL: http://jgraph.sourceforge.net. Consultado em 07/06/2004.

[11] NUNES, Paulo R. A. F. e MELO, Ana C. V.. Ocongra - Uma ferramenta para geração de Grafos de Fluxo de Controle de Objetos. In *18° Simpósilo Brasileiro de Engenharia de Software - Sessão de Ferramentas*. SBC, 2004.

---

1 Exemplo retirado do Artigo [6]