VII CONGRESSO DE INICIAÇÃO CIENTÍFICA DA UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

Pró-Reitoria de Pesquisa e Extensão

CNPq

PIBIC/CNPq/UFCG-2010

ESPECIFICAÇÃO E VERIFICAÇÃO DE REGRAS DE DESIGN EM PROGRAMAS JAVA E ASPECTJ

Solon Barbosa de Aguiar Neto[1], Rohit Gheyi[2]

RESUMO

O paradigma orientado a objetos (OO) modulariza as principais preocupações de um programa, mas existem algumas preocupações que se espalham pelo código ou que se misturam a outras preocupações. Então, o paradigma orientado a aspectos (AO) pode ser utilizado para modularizar essas preocupações, ditas preocupações transversais. Entretanto, existe uma forte dependência entre o código orientado a objetos e o orientado a aspectos. Essa dependência pode ser identificada, por exemplo, nos adendos (pontos de junção). Neste artigo apresentamos os primeiros passos na criação de uma linguagem e suporte ferramental (analisador estático) para expressar regras de design que ajudam a melhorar a modularidade entre aspectos e classes.

Palavras-chave: Desenvolvimento orientado a aspectos, regras de design, analisador estático

SPECIFICATION AND VERIFICATION OF DESIGN RULES IN JAVA AND ASPECTJ PROGRAMS

ABSTRACT

Aspect programming modularizes the main concerns regarding in the code. Some concerns, on the other hand, are spread throughout code and other unrelated concerns. This way, aspects may be used for modularizing these concerns, called traversed concerns. However, there is a tight dependency between OO code and AP. This dependency can be identified, for instance in junction points. In this paper we show the first steps on the creation of a language and tool support for expressing design rules that improve modularity between aspects and classes.

Keywords: Aspect programming, design rules, static analyzer

INTRODUÇÃO

O paradigma de orientação a aspectos (GREGOR KICZALES et al., 1997) surgiu da necessidade de se modularizar preocupações transversais. Entende-se por preocupações transversais aquelas que se espalham pelo código de uma aplicação e que também se misturam com outras preocupações. Dessa forma, orientação a aspectos (OA) surge como uma alternativa à modularização desse problema. Isso é feito em unidades chamadas de *aspectos*.

Basicamente um aspecto possui pontos de junção, que capturam partes específicas do código; e adendos, trechos de código ou informações que são adicionadas ao programa no ponto de junção especificado. Assim, um aspecto captura os diversos pontos de junção do código onde ele deve adicionar; sobrepor ou até mesmo substituir informações, através dos adendos descritos nele.

[1] Aluno do Curso de Ciência da Computação, Centro de Engenharia Elétrica e Informática, Departamento de Sistemas e Computação, UFCG, Campina Grande, PB, E-mail: solon@dsc.ufcg.edu.br

[2] Cientista da Computação, Professor. Doutora, Centro de Engenharia Elétrica e Informática, Departamento de Sistemas e Computação, UFCG, Campina Grande, PB, E-mail: rohit@dsc.ufcg.edu.br

Mas, para a declaração destes pontos de junção é necessário ter informações sobre o código orientado a objetos (OO). Informações como, por exemplo, o nome de uma classe; o tipo de retorno de um método; quais são os parâmetros de um método ou ainda se ele deve lançar exceções. Apenas quando estiver em posse dessas informações podemos declarar um ponto de junção e termos certeza de que os adendos que dependem destes pontos de junção irão trabalhar de forma correta. Isso garantirá a funcionalidade do programa.

Então, se por algum motivo tivermos a necessidade de alterar algo no código OO como, por exemplo, o nome de uma classe e tivermos um ponto de junção que referencia essa classe, a funcionalidade deste estará comprometida. Isso ocorre porque ele passará a capturar uma classe inexistente, já que o nome da classe foi mudado. Portanto, existe uma forte dependência entre o código orientado a objetos e o código orientado a aspectos. Mesmo que os aspectos modularizem as preocupações transversais, a modularidade das classes é quebrada, já que o programador OO deve estar ciente de quais partes de seu código são importantes para os pontos de junção declarados nos aspectos.

Segundo Parnas (1979), um bom desenvolvimento modular tem por conseqüência três principais benefícios para o projeto. São eles: compreensão; fácil capacidade de mudança e desenvolvimento em paralelo. O programa é decomposto em unidades menores chamadas de *módulos*. Cada um desses módulos pode ser desenvolvido e compreendido separadamente. Entretanto, no atual contexto, a modularidade entre OO e OA é comprometida pela quebra da modularidade das classes.

Desta forma, podemos identificar a necessidade de se especificar diretrizes ou regras que tanto os programadores OO quanto os OA devam seguir para que o funcionalidade do programa não seja comprometida. Essas regras também devem ajudar a melhorar a modularidade do projeto. Elas seriam a principal ponte entre classes e aspectos, tendo em vista essas características que Sullivan (SULLIVAN et al., 2005) propôs para o uso de regras de design. Essas regras visam diminuir o acoplamento entre classes e aspectos. A diferença é que as regras de Sullivan eram descritas em uma linguagem natural.

Seguindo este trabalho, Griswold (2005) propôs um modo de especificar estas regras de design utilizando a própria linguagem de AspectJ (KICZALES, 2001). Aspectj é uma linguagem de programação do paradigma orientado a aspectos. Usando declarações *error* e *warning*, essas regras poderiam ser automaticamente checadas, mas usar a própria linguagem AspectJ para criar estas regras pode ser de difícil declaração, além de escalar para poucas regras. Em outras palavras, teríamos regras bastante limitadas por usarmos a própria linguagem para essas declarações.

Tendo observado estas limitações, Doseá (2007) propôs uma linguagem para a especificação das regras de design. Essas regras de design especificariam quais requisitos os desenvolvedores OO e OA deveriam seguir para permitir um desenvolvimento em paralelo e com baixo acoplamento. Essa linguagem, entretanto, não era suficientemente expressiva e não possuía uma semântica bem definida, além de não possuir um suporte ferramental para a checagem automática dessas regras.

MATERIAL E MÉTODOS

Abordagem

Esse projeto teve como objetivo definir uma linguagem formal para a especificação de regras de design, assim como desenvolver métodos para a checagem automática dessas regras utilizando a API *reflection* de Java. Então, assim como Java possui interfaces, apresentamos uma interface para preocupações transversais. Ou seja, uma *xpi* (interfaces para programas entrecortantes) que foi representada pela linguagem que definimos.

Vale lembrar que o suporte ferramental é de grande importância para a linguagem, pois não só tivemos uma linguagem própria para especificar as regras de design, como esta checagem facilitará o trabalho dos desenvolvedores do projeto. Isso é feito ao exibir alertas ou mensagens sobre as regras de design. Também é alertado se essas regras estão sendo seguidas de forma correta. Isso serve como um guia em todas as etapas do projeto. A figura 1 apresenta a abordagem seguida.

Portanto, especificamos a sintaxe da linguagem seguindo o formalismo de Backus-Naur e utilizamos o Java Compiler Compiler (javaCC). JavaCC é um analisador sintático para criar a gramática desta linguagem. Com esse objetivo definido, também avaliamos as vantagens e desvantagens da linguagem proposta em comparação com outras abordagens; a compreensão e que regras podemos especificar e, por fim, a expressividade de nossa linguagem.

Implementação da Linguagem/ Analisador Estático

Após a revisão bibliográfica e o estudo do paradigma AO escolhemos utilizar o JavaCC para a implementação da gramática da linguagem, tendo em vista a análise estática para o suporte ferramental. A especificação da linguagem foi feita em paralelo ao desenvolvimento da análise das regras de design.

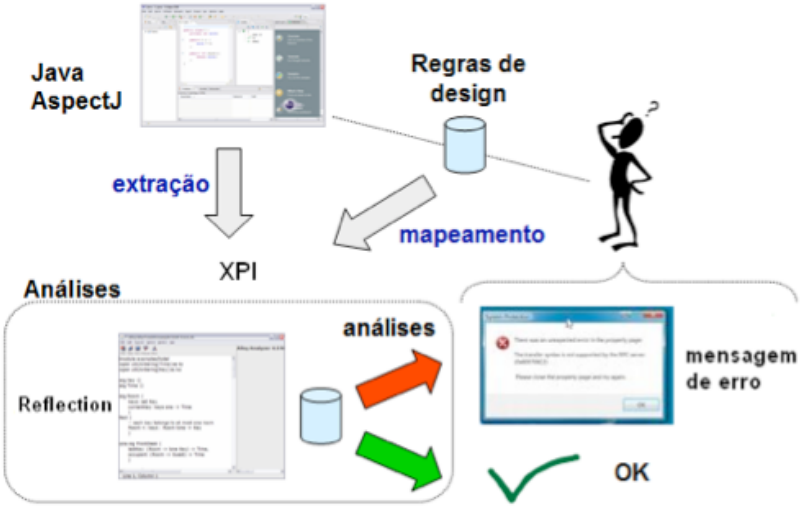

Figura 1: Técnica Proposta

Assim, sempre que uma nova regra era expressa na linguagem, também era desenvolvida a ferramenta que checaria a mesma regra.

Utilizamos a API *reflection* de Java para implementar as regras. Ela nos permite analisar o comportamento do programa. Neste projeto, reflection é utilizado para descobrir informações que são declaradas na regras de design, como, por exemplo, a existência de uma classe; quais métodos essa classe possui; se ela tem algum tipo de herança; as interfaces que ela implementa etc.

RESULTADOS E DISCUSSÃO

Para avaliar a linguagem proposta, especificamos as regras de design catalogadas na revisão bibliográfica. Nesse caso, criamos a xpi para o exemplo das figuras geométricas, que é um exemplo clássico no estudo do paradigma AO. Apresentamos na figura 2.

```
xpi DisplayUpdate {
        class c1 = public class Tela ;
        interface i1 = public interface FiguraGeometrica;
        method m1 = public void mover(int, int);
        method m2 = public static void atualizar () ;
        rules : c1 ;
        i1;
        c1 implements i1;
        m1 methodOf i1;
        !m2 methodOf c1;
}
```

Figura 2: Interface para aspectos

Nessa xpi há a declaração de quatro variáveis. Duas delas correspondem a uma classe *public* de nome `Tela` e uma interface pública, de nome `FiguraGeometrica`. As outras duas variáveis correspondem aos métodos públicos sem retorno: `mover` e `atualizar`, sendo este também estático. Após a declaração dessas variáveis temos as regras de design da xpi. Definimos a existência da classe `Tela` e da interface `FiguraGeometrica`. Além disso, definimos que a classe `Tela` implementa a interface `FiguraGeometrica`; que o método `mover` é declarado nesta interface e, por fim, que o método `atualizar` não deve ser existir na classe `Tela`.

Outras xpis estão sendo criadas para os padrões de projeto propostos pelo grupo Gang-of-Four (GoF) e implementados em AspectJ. Em todos os estudos de caso, sempre quando foi necessário, refinamos a nossa linguagem desenvolvida ao adicionar novas regras ou corrigir as regras existentes.

De acordo com os estudos da revisão bibliográfica foram identificadas várias regras de design. Podemos citar, como exemplo, regras existenciais. A partir dessas regras, definimos se o projeto deve ou não possuir uma classe ou interface; ou ainda se um método ou atributo existe no escopo de uma determinada classe. Por exemplo, um ponto de junção que captura a execução de um método que saque uma quantia *x*, que está na classe `ContaBancaria` pode nos dar vários exemplos de regras existenciais. Elas podem ser vistas na figura 3.

```
public class ContaBancaria {
   ...
   public boolean sacar (double quantia) {
     ...
   }
}

public aspect CPMF {
   ...
   after() returning(): execution
   (boolean ContaBancaria.sacar(double) {
     ...
    }
}
```

Figura 3: Exemplos de regras existenciais.

Neste exemplo várias operações foram omitidas. Apresentamos apenas os principais pontos para o entendimento do exemplo. Diante dessa classe e desse aspecto, observamos que este captura um ponto de junção da classe `ContaBancaria` depois de retornar da execução de um método público de retorno *boolean* cujo nome é `sacar` e que possui um único parâmetro do tipo *double*. Ao extrair as regras de design segundo o aspecto deste exemplo tivemos as seguintes regras:

- Deve haver uma classe de nome `ContaBancaria`;
- Deve haver um método público de retorno *boolean* com o nome sacar e um único parâmetro de tipo *double*. Esse método pertencerá à classe `ContaBancaria`;

Seguindo essa idéia pudemos identificar outras regras como a existência de uma interface; de uma classe; de um atributo; de um construtor ou método específico, assim como poderíamos definir regras nas quais uma classe deve implementar uma interface ou definir herança entre classes.

Tivemos como objetivo declarar essas regras. Aliado a isso e visto que aspectos modularizam preocupações transversais que se combinam com o código orientado a objetos, identificamos a necessidade de se especificar apenas regras de design sobre as limitações OO nos quais os aspectos trabalharão. Portanto, não existem regras existenciais sobre os componentes presentes nos aspectos.

Assim como identificamos regras simples como as existenciais, existem regras mais complexas. O padrão de projeto *Singleton* é um bom exemplo dessas regras. Nesse padrão, só deve haver um único objeto de um determinado tipo instanciado por vez. Logo, por definição, todos os seus construtores devem ser privados e a única maneira de acessar esses objetos é através do método estático `getInstance`, que retorna um objeto da interface *Singleton*. Então, poderíamos definir as seguintes regras de design para o padrão de projeto *Singleton*:

- Deve haver uma interface de nome *Singleton*;
- Para todas as classes que implementam a interface *Singleton,* todos os construtores dessas classes devem ser privados e todas as classes devem ter o método `getInstance`, com os qualquer parâmetro, que retorne um objeto do tipo *Singleton*;

Dessa forma, compreendemos que, assim como existem regras específicas mais simples de serem declaradas e checadas, também existem regras globais, mais complexas e que exigem um esforço adicional para serem checadas, mas que adicionam um grande poder à linguagem.

Analisador Sintático

Para definir nossa linguagem utilizamos o JavaCC como meio de criar a gramática da linguagem, com suas próprias palavras reservadas e expressões próprias para a linguagem. Grande parte das regras de produção foram reaproveitadas da gramática 1.5 de Java, visto que nosso objetivo é especificar regras de design para o código OO e que para isso precisamos de grande parte da sintaxe de Java.

A partir dos estudos observados, nossa linguagem especifica uma interface para aspectos na qual tivemos um conjunto de variáveis e declaramos as regras de design do projeto. Na figura 4 descrevemos a estrutura de nossa linguagem juntamente com algumas regras da BNF.

```
xpi name {
   (Constructor id = <signature> |
    Field id = <signature> |
    Method id = <signature> |
    Class id = <signature> |
    Interface id = <signature>)*

   rules
     (existanceRule ::= var |
      interfaceRule ::= var "implements" var |
      inheritanceRule ::= var "extends" var |
      methodRule ::= var "methodOf" var |
      constructorRule ::= var "constructorOf" var |
      attributeRule ::= var "in" var |
      callRule ::= var "call" var)*
 }
```

Figura 4: Estrutura da interface para aspectos.

Na parte superior da linguagem mostrada na figura 4, declaramos os elementos da mesma (classes, métodos, construtores, interfaces, atributos). Em cada uma, colocamos a assinatura da mesma forma que declaramos em Java. O objetivo destas declarações é utilizá-las nas regras definidas na parte de baixo da figura 4. As regras são expressas a partir do conjunto de variáveis e das palavras reservadas de nossa linguagem.

Basicamente a linguagem produzida subdividi-se em duas partes: a declaração de variáveis e a declaração das regras de design. A declaração das variáveis é necessária para a posterior declaração das regras, pois estas usam as variáveis como base para a sua definição. Já a declaração das regras define as regras que serão checadas pelo analisador estático.

Declarações

Nossa linguagem permite declarar as seguintes variáveis (cada declaração é seguida de um exemplo):

- **Class**: variável que identifica uma classe. Deve declarar sua visibilidade, modificadores (se é abstrata, visibilidade etc.) e o respectivo nome da classe.
  Exemplo: class c = **public class** ContaBancaria
- **Interface**: variável que identifica uma interface. Deve declarar sua visibilidade, modificadores, (se é abstrata etc.) e o respectivo nome da interface. Existem poucas diferenças entre as variáveis de *classe* e de *interface*, mas é necessário diferenciá-las pois há diferenças na checagem das regras de cada uma. Além disso, foi uma decisão deste projeto ter separado ambas.
  Exemplo: interface i = **public interface** Conta
- **Method**: variável que identifica um método. Deve declarar sua visibilidade, modificadores (se é abstrato, estático etc), tipo de retorno do método, parâmetros, se lança exceções e o seu nome.
  Exemplo: method m = **public boolean** sacar(**double**)
- **Constructor**: variável que identifica um construtor. Deve declarar sua visibilidade, modificadores, parâmetros, se lança exceções, além do respectivo nome do construtor.
  Exemplo: constructor ctr = **public** ContaBancaria(**String**, **String**, **int**)
- **Attribute**: variável que identifica um atributo. Deve declarar sua visibilidade, modificadores, tipo, além de seu respectivo nome.
  Exemplo: attribute atr = **private String** ID

Por fim, nossa linguagem possui palavras reservadas especiais que dão uma maior expressividade à declaração das variáveis. Por exemplo, a palavra * significa qualquer nome, tipo ou modificador. Já a palavra .. significa um número qualquer de parâmetros. Poderíamos estar interessados em todos os métodos de privados, com qualquer retorno, qualquer nome e ainda qualquer parâmetro. Assim, expressaríamos esta variável da seguinte forma: method m = private * * (..).

Numa primeira abordagem, nossa linguagem não utilizava estas variáveis e com isso encontramos alguns problemas ou limitações na criação das regras de design. Portanto, foi sugerido o uso de variáveis para posterior criação das regras. Note que essas declarações apenas auxiliam na criação das regras de design. Sozinhas elas não têm poder algum. A decisão de usar variáveis para a linguagem foi tomada para facilitar o reuso destas em diferentes regras, assim como diminuir o tamanho das regras criadas.

Regras

A declaração das regras usa as variáveis descritas anteriormente na linguagem. Nessa parte da xpi são criadas as regras que serão checadas pelo analisador estático. Podemos definir como regras: regras existenciais (aquelas que definem a existência de uma classe interface); regras de herança/interfaces (que declaram se uma classe estende de outra ou se uma classe implementa uma interface); regras locais (que definem em qual classe ou interface um método ou atributo deve estar - construtor no caso de classes) e, por fim, se um método deve chamar outro método.

Assim como podemos definir a existência dessa regras, podemos declarar o oposto: proibir existência de uma classe, interface etc. Definimos o operador "!" (*not*) para declarar que um projeto não deve ter uma determinada classe ou que um método não pode chamar outro método. O operador not se estende para todas as regras declaradas neste projeto.

As seguintes regras de design podem ser expressas na nossa linguagem, onde a palavra `var` refere-se a uma variável declarada anteriormente. Para todas as regras, podemos tanto definir a existência daquela regra como também proibir a existência da mesma. Isso é feito ao usar o operador not no início da declaração. As regras presentes no projeto são:

- Regra que define a existência ou não de uma classe ou interface: `var`.
- Regra que define ou não se uma classe estende outra classe ou se uma interface estende ou não outra interface: `var extends var`.
- Regra que define ou não se uma classe implementa uma interface: `var implements var`.
- Regra que define se um método pertence ou não a uma classe ou interface: `var methodOf var`.
- Regra que define e um atributo pertence ou não a uma classe: `var in var`.
- Regra que define se um construtor pertence ou não a uma classe: `var constructorOf var`.
- Regra que define se um método chama ou não outro método: `var call var`.

O analisador sintático checa cada uma das variáveis quando uma regra é criada. Logo, se por algum motivo declararmos que um método estende uma classe, o projeto acusaria que não é possível fazer essa declaração e que houve um erro de sintaxe. Dessa forma, garantimos que apenas as variáveis do tipo correto são associadas a uma determinada regra.

CONCLUSÃO

O desenvolvimento da linguagem foi bastante produtivo tendo visto várias aplicações de assuntos estudados pelo bolsista na disciplina de Teoria da Computação. Embora inicialmente tenhamos seguido uma outra abordagem sem declaração de variáveis e bastante semelhante a abordagem de Doseá, não houve problemas em mudar a sintaxe da linguagem para a atual. Isso ocorreu apesar dessa tarefa ter consumido um tempo considerável.

Temos uma boa solução para assegurar a qualidade de nossa linguagem à medida que ela evolui. Testes de unidade que checam o maior número possível de regras, assim como dão suporte ferramental no desenvolvimento e checagem das regras são usados. A cada nova regra acrescentada à linguagem os testes são rodados. Isso assegura que as regras já criadas continuam funcionando de forma correta.

Como trabalho futuro, o principal objetivo é aumentar a expressividade da linguagem; criar regras mais complexas e que tenham um escopo maior. Para isso, investigaremos mais estudos de caso como modelo para a criação e checagem destas regras. Além disso, outro ponto de fundamental importância é a validação deste projeto. Dessa forma, continuaremos com o estudo dos padrões de projeto e a criação de xpis para os mesmos. Isso servirá para avaliar o grau de modularidade dos padrões de projeto implementados utilizando aspectos com e sem regras de design auxiliando estes.

É importante lembrar que para cada regra é necessário adicionar a checagem desta regra ao analisador estático, então o projeto continuará com o trabalho paralelo de propor novas regras e checá-las. Além disso, serão adicionados novos testes de unidade para estas regras.

AGRADECIMENTOS

Agradeço ao orientador deste projeto, Rohit Gheyi, pelos ensinamentos e ajuda ao longo de todo o projeto e ao CNPq pela bolsa de Iniciação Científica.

REFERÊNCIAS BIBLIOGRÁFICAS

BORBA, P. *et al*. Algebraic Reasoning for Object-Oriented Programming. **Science of Computer Programming**, v.52, p.53–100, Aug. 2004.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines:** Practices and Patterns. Addison-Wesley, 2001.

CODE, S. **Semmle Code**. Disponível em:< http://semmle.com/ >. Acesso em: 30 jun. 2009.

COLE, L.; BORBA, P. Deriving refactorings for AspectJ. **AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development**, New York, p. 123–134, 2005.

CZARNECKI, K. *et al*. Feature models are views on ontologies. **Proceedings of the 10th International on Software Product Line Conference**, Washington, p. 41–51, 2006.
FOWLER, M. **Refactoring:** Improving the Design of Existing Code. Addison- Wesley. 1999.

GAMMA, E. *et al*. **Design Patterns:** Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994.

GHEYI, R. **Basic laws of object modeling**. 2004. Dissertação (Mestrado em Ciência da Computação), Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, 2004.

GHEYI, R. **A Refinement Theory for Alloy**. 2007. Tese (Doutorado em Ciência da Computação), Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, 2007.

GHEYI, R.; MASSONI, T.; BORBA, P. A complete and minimal set of algebraic laws for feature models. **Simpósio Brasileiro de Linguagens de Programação (SBLP)**, Fortaleza, Brasil, Aug. 2008.

GHEYI, R.; MASSONI, T.; BORBA, P. Algebraic laws for feature models. **Journal of the Brazilian Computer Society**, 2009.

JACKSON, D. **Software Abstractions:** Logic, Language and Analysis. MIT press, 2006.

JACKSON, D.; SCHECHTER, I.; SHLYAHTER, I. Alcoa: the alloy constraint analyzer. **22nd International Conference on Software Engineering,** p. 730–733, 2000.

JACKSON, M. **Problem frames:** analyzing and structuring software development problems. Addison-Wesley Longman Publishing, 2001.

JUnit. Junit. Disponível em: < http://www.junit.org >. Acesso em: Abr. 2010.

KICZALES, G. *et al*. Getting started with AspectJ. **Communications of the ACM**, p.59–65, 2001.

KICZALES, G. *et al*. Aspect-oriented programming. **Proceedings European Conference on Object-Oriented Programming**, Berlin and New York, vol. 1241, p. 220–242, 1997.
MORGAN, C.; VOLDER, K.; WOHLSTADTER, E. A static aspect language for checking design rules. **Proceedings of the 6th international conference on Aspect-oriented software development**,New York, p. 63–72, 2007.

COSTA NETO, A. *et al*. Semantic dependencies and modularity of aspect-oriented software. **ICSE Workshop on Assessment of Contemporary Modularization Techniques**, Minneapolis, May 2007.

OPDYKE, W. **Refactoring Object-Oriented Frameworks**. 1992. Tese (Doutorado em Ciência da Computação), University of Illinois at Urbana-Champaign, 1992. Disponível em: < http://gotdotnet.com/team/fxcop/ >. Acesso em: 30 Abr. 2007.