MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
SECRETARIA DA CIÊNCIA E TECNOLOGIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

EDUARDO SOARES ALBUQUERQUE

DETECÇÃO DE PADRÕES DE CÓDIGO JAVA NA PLATAFORMA ECLIPSE

Rio de Janeiro
2005

INSTITUTO MILITAR DE ENGENHARIA

EDUARDO SOARES ALBUQUERQUE

DETECÇÃO DE PADRÕES DE CÓDIGO JAVA NA PLATAFORMA ECLIPSE

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Alex de Vasconcellos Garcia – D.Sc.

Rio de Janeiro

2005

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 – Praia Vermelha
Rio de Janeiro - RJ CEP: 22290-270

A345 Albuquerque, Eduardo Soares
Detecção de padrões de código Java na plataforma Eclipse, Eduardo Soares Albuquerque.
- Rio de Janeiro: Instituto Militar de Engenharia, 2005.
88 p.:il, graf., tab.

Dissertação: (mestrado) – Instituto Militar de Engenharia, Rio de Janeiro, 2005.

1. Engenharia de Sistemas e Computação. 2. Linguagens de Programação. 3. Java (linguagem de programação de computador). I. Título. II. Instituto Militar de Engenharia.

CDD 005.13

INSTITUTO MILITAR DE ENGENHARIA

EDUARDO SOARES ALBUQUERQUE

DETECÇÃO DE PADRÕES DE CÓDIGO JAVA NA PLATAFORMA ECLIPSE

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Alex de Vasconcellos Garcia – D. Sc.

Aprovada em 14 de julho de 2005 pela seguinte Banca Examinadora:

_____

Prof. Alex de Vasconcellos Garcia – D. Sc. do IME – Presidente

_____

Prof. Edward Hermann Haeusler – D. Sc. da PUC-RIO

_____

Profª. Isabel Leite Cafezeiro – D. Sc. da UFF

_____

Prof. Ulf Bergmann – D. Sc. do IME

Rio de Janeiro

2005

AGRADECIMENTOS

Existem várias pessoas que tiveram uma importante participação na realização deste trabalho a quem gostaria de expressar minha imensa gratidão.

Ao meu pai e grande amigo Gumercino Gomes de Albuquerque, agradeço pelo carinho, incentivo, por batalhar tanto pelo melhor para mim e meus irmãos.

À minha querida mãe Wilma Soares Albuquerque, pelo apoio incondicional, carinho, amor e exemplo de vida.

Aos meus irmãos Juliana e Saulo pela amizade e companheirismo.

À família da minha noiva: Sr. Jonas, Dona Rita, Fabinha, Fabrícius, tio Jailson pelo apoio.

Aos meus companheiros de república, Marcelo, Handrick, Papel, Vladmir, Fabrício, CAndré, Arthur e Diogo pela amizade demonstrados durante esses anos de convivência.

Aos amigos que deixei em Ouro Preto, principalmente para república Consulado pela amizade e carinho recebidos sempre.

Ao professor Alex pela amizade, confiança, dedicação e competência com que conduziu a orientação deste trabalho.

Ao Instituto Militar de Engenharia, professores e funcionários do Departamento de Engenharia de Sistemas, e a todas as pessoas que, direta ou indiretamente, contribuíram para o desenvolvimento deste trabalho.

A CAPES, pelo suporte financeiro proporcionado durante estes dois anos, sem o qual seria impossível a realização desse sonho.

E finalmente, gostaria de agradecer a minha amada noiva Francesca Merigueti de Paula, pelo amor, carinho e por suportar os longos momentos de ausência durante todo esse tempo.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

API	APPLICATION PROGRAM INTERFACE
AST	ABSTRACT SYNTAX TREE
BCEL	BYTE CODE ENGINEERING LIBRARY
DOM	DOCUMENT OBJECT MODEL
EJB	ENTERPRISE JAVA BEANS
GUI	GRAPHIC USER INTERFACE
IDE	INTEGRATE DEVELOPMENT ENVIRONMENT
JDT	JAVA DEVELOPMENT TOOLING
JSP	JAVA SERVER PAGE
PDE	PLUG-IN DEVELOPMENT ENVIRONMENT
PDL	PATTERN DESCRIPTION LANGUAGE
XML	EXTENSIBLE MARKUP LANGUAGE

RESUMO

Este trabalho apresenta uma ferramenta para detecção desses padrões integrada à Plataforma Eclipse. A vantagem da ferramenta apresentada é implementar uma nova linguagem de descrição de padrões, definida neste trabalho, que permite que padrões sejam escritos de forma declarativa. Isso torna a ferramenta mais flexível que as ferramentas existentes. Nesta linguagem, os padrões são representados na forma de árvores, que são buscados pela ferramenta na AST do programa fonte. O algoritmo desenvolvido para fazer o casamento utiliza uma estratégia de clonagem de padrões, a fim de realizar o casamento de todas as instâncias dos padrões selecionados para detecção, percorrendo a AST do programa uma única vez. É apresentado ainda um estudo de caso realizado com a utilização da ferramenta sobre um sistema comercial, que mostra a flexibilidade, facilidade de uso e a eficiência da ferramenta em um projeto real.

ABSTRACT

This work presents a pattern detect tool integrated into the Eclipse Platform. The advantage of the presented tool is to implement a new language of description of patterns, defined in this work, that allows patterns to be written in a declarative way. That turns the most flexible tool than the existent tools. In this language, the patterns are represented in the form of trees, they are searched by the tool in AST of the program source. The algorithm developed to pattern match uses a strategy of clone patterns, in order to accomplish the pattern match of all the instances of the patterns selected for detection, transversing AST of the program an only time. It is presented a case study still accomplished with the use of the tool on a commercial system, that shows the flexibility, use easiness and the efficiency of the tool in a real project.

1. INTRODUÇÃO

Atualmente existem diversas ferramentas capazes de encontrar padrões sintáticos e até mesmo refatorar código fonte. Essas ferramentas possuem uma vasta gama de aplicações, como encontrar padrões (ou anti-padrões) de projeto, automatizar a manutenção evolutiva de acordo com métodos de orientação a objetos, extrair métricas a partir do código fonte e auxiliar no processo de inspeção de código fonte.

Durante o estudo destas ferramentas (que serão vistas no capítulo 2) observamos que a sua integração a um Ambiente de Desenvolvimento Integrado (IDE - *Integrated Development Environment*) torna-as mais prontamente utilizáveis, sendo facilmente adotadas pelos desenvolvedores que fazem uso do IDE. Observamos também que as ferramentas de origem acadêmica em geral deixam a desejar neste aspecto, o que prejudica sua usabilidade.

Constatamos ainda que uma deficiência da maior parte destas ferramentas, em especial as desenvolvidas fora da academia, é a dificuldade de especificar novos padrões. Para detectar-se um novo padrão estes devem ser tipicamente codificados em baixo nível. Desta forma tomamos como objetivo desenvolver uma ferramenta que atendesse aos seguintes requisitos:

- Especificação declarativa de novos padrões.
- Facilidade de inclusão dos novos padrões à ferramenta. Julgamos este requisito fundamental para a aplicabilidade da ferramenta em situações reais, que dependam da realidade do ambiente de desenvolvimento/execução, bem como da cultura da organização na qual será empregada.
- Integração da ferramenta com um IDE.

Finalmente, cabe ressaltar que a motivação do autor foi, desde o início, aplicar a ferramenta como um instrumento para auxiliar no processo de inspeção de código fonte. Desta forma, um segundo objetivo deste trabalho foi validar a aplicação da ferramenta como um instrumento para automatizar parte do processo de inspeção de código fonte.

Como o próprio título da dissertação diz, a linguagem para a qual se realizará a inspeção será a linguagem Java [JAVA] e a IDE utilizada para integração será o Eclipse [ECLIPSE].

1.1. INSPEÇÃO DE SOFTWARE

Com o aumento do tamanho e complexidade dos programas, o desenvolvimento e gerenciamento desses se tornaram também bastante complexos. Apesar dos vários métodos e ferramentas existentes para facilitar esses trabalhos, ainda assim existem grandes dificuldades em se explorar boas práticas no desenvolvimento, como por exemplo, o uso correto dos conceitos de orientação a objetos, que apesar de ter conceitos considerados relativamente simples, a sua aplicação não é necessariamente trivial. Devido a esses problemas, a Inspeção de *Software* tem alcançado um papel ainda mais relevante dentro do processo de desenvolvimento, o qual, cada vez mais precisa ser qualificado e padronizado, a fim de suportar organizações maiores e mais complexas de software.

O processo de inspeção foi introduzido por Michael Fagan [FAGAN, 1976]. Esse processo consiste em uma técnica formal para detecção e correção de “defeitos” em *softwares* através da análise de documentos. Estão envolvidos nesse processo o autor do documento, um moderador, um redator e um grupo de inspetores que farão a inspeção do documento em vários estágios dentro do processo. Dentre os vários benefícios da inspeção de *software* [FAGAN, 1976; FAGAN, 1986; SAPSOMBOON, 1999], podemos destacar:

1.Melhoria da qualidade do produto final.

2.Aumento da produtividade do desenvolvimento.

3.Diminuição dos custos de manutenção.

4.Melhoria do conhecimento técnico do programador.

5.Disseminação do conhecimento do projeto entre os membros da equipe.

6.Facilita a adoção de padrões de codificação

7.Melhora a avaliação do andamento do projeto.

8.Incentiva a integração da equipe.

A principal meta da inspeção é identificar defeitos nas primeiras fases do ciclo de desenvolvimento, tendo em vista os benefícios proporcionados em relação a esses defeitos serem encontrados tardiamente, o que acarretaria uma maior dificuldade em identificá-los e um maior custo em repará-los entre outros problemas. O resultado final é uma melhoria na qualidade do produto além de diminuição nos custos do mesmo. Esses benefícios são

geralmente aceitos tendo em vista vários estudos e casos de sucesso citados na literatura [FAGAN, 1986; MCCARTHY, 1996; RUSSELL, 1991; WELLER, 1993; PORTER, 1997].

Apesar dos benefícios, a inspeção de *software* encontra grande resistência e dificuldades de ser colocada em prática. Isso pode ser atribuído a vários fatores, dentre os quais, destacamos a necessidade de investimento em tempo e infra-estrutura para colocá-la em prática, embora esses investimentos sejam teoricamente compensados pelos benefícios. Outro fator de fundamental importância para que isso ocorra é o fato do processo de inspeção tradicional (formal) desprender um enorme esforço para realizá-lo, devido ao processo envolver uma cuidadosa revisão manual de todos artefatos envolvidos na construção de um *software*. Isso juntamente com o aumento do tamanho e complexidade dos *softwares* têm resultado em inspeções incompletas ou mesmo ruins além de um aumento no tempo desprendido para sua realização [MACDONALD, 1995].

1.2. AUTOMAÇÃO DA INSPEÇÃO DE CÓDIGO

A busca por uma maior qualidade tanto no processo de desenvolvimento quanto no produto final motivou a automatização de partes do processo de inspeção. Isso se justifica devido ao processo formal envolver um intenso trabalho manual, além de envolver a participação de um grupo de pessoas por um longo tempo. Automatizar partes do processo de inspeção torna-o mais eficiente, evita a não identificação de problemas ocorridos por falha humana na inspeção, além de adequá-lo a realidade atual, tornando viável sua prática. Um ponto importante a se destacar é que a inspeção automática não substitui a inspeção humana (manual), mas pode e deve complementá-la.

Originalmente, a inspeção de software foi motivada principalmente pela inspeção de código. A inspeção de código é a parte desse processo que averigua a conformidade do código escrito com a funcionalidade para que foi proposto, erros de programação e a não conformidade com a padronização adotada. A padronização do código-fonte, embora não seja fundamental para o sucesso um *software*, é uma prática cada vez mais adotada. Isso se deve a vários motivos, entre eles podemos destacar:

- O aumento na interação de todos envolvidos na construção do *software*.

- O aumento da complexidade e número de linhas de códigos.
- A disseminação de padrões e boas práticas de codificação, o que evita erros de programação.
- Facilidade de compreensão do código para eventuais manutenções e evoluções.

Algumas partes da inspeção de código, como por exemplo, a conformidade com padrões de codificação, não só podem, mas devem ser feitas através de uma inspeção automática. Isso porque este tipo de análise quando realizada por meio de *software* é mais eficiente, não permitindo falhas humanas durante a inspeção. Entretanto, outras atividades realizadas durante o processo de inspeção, como por exemplo, a verificação semântica de métodos e funções a fim de checar se estes realizam o que se propuseram a fazer, não podem ser automatizadas.

A idéia de automatizar parte do processo de inspeção motivou a criação de uma ferramenta com este fim integrada a uma IDE, diminuindo assim o trabalho manual realizado durante o processo bem como adequando este a realidade atual, fazendo com que esse processo seja mais viável de ser realizado, trazendo em conseqüência disso, os benefícios esperados de uma inspeção.

1.3. ORGANIZAÇÃO DA DISSERTAÇÃO

Os capítulos restantes estão organizados da seguinte maneira: o capítulo 2 descreve trabalhos relacionados à inspeção automática de código. O Capítulo 3 discorre sobre a plataforma Eclipse e suas principais características, dando uma breve introdução aos seus conceitos básicos, ao JDT (*Java Development Tooling*) e ao desenvolvimento de *plug-ins*. No capítulo 4 será apresentada a linguagem para representação de padrões e suas características. O capítulo 5 apresenta a arquitetura da ferramenta para detecção de padrões, o modelo utilizado para representação dos padrões e o algoritmo que realiza o casamento. No capítulo 6 é mostrado um estudo de caso realizado com a utilização da ferramenta e seus resultados. Finalmente, o capítulo 7 apresenta as conclusões finais, contribuições e trabalhos futuros.

2. TRABALHOS RELACIONADOS

Neste capítulo serão apresentados 4 trabalhos relacionados à inspeção automática de código, sendo o primeiro deles o único não relacionado com a linguagem Java.

2.1. LINT

Lint [JOHNSON, 1978] foi um dos primeiros trabalhos a propor uma solução para inspeção de código automática. Essa ferramenta trabalha na inspeção de códigos escritos na linguagem C e foi criada em 1976 por S. C. Johnson. Ela inspeciona o código fonte em busca de erros como: variáveis e funções não declaradas, variáveis não inicializadas, fragmentos de código inatingíveis, erros de tipos, *type casts*, entre outros.

A ferramenta é estruturada como dois programas e um *driver*. O primeiro programa é uma versão portável de um compilador C. Esse compilador faz a analise léxica e sintática, constrói e gerencia uma tabela de símbolos e monta árvores para expressões. Ao invés de criar um arquivo intermediário que serviria como entrada para o gerador de código, como em outros compiladores, o Lint produz um arquivo intermediário com referências ao uso, definição, e declaração de variáveis externas. As informações das variáveis locais a uma função ou a um arquivo são coletadas diretamente na tabela de símbolos ou examinando as árvores de expressões. O segundo programa lê o arquivo intermediário e verifica a consistência de todas as definições, declarações e usos das mesmas. Ou seja, o primeiro programa identifica problemas num contexto local e gera um arquivo com referências externas para ser utilizado pelo segundo programa que por fim detecta problemas de nível global envolvendo todos os arquivos de entrada da inspeção. O *driver* controla todo o processo de inspeção feito pelos dois programas.

2.2. CODEPRO ADVISOR

Das ferramentas apresentadas nesse capítulo, a CodePro Advisor [CODEPRO] é a única comercial. Outro aspecto importante a se mencionar, é que esta ferramenta também está integrada ao ambiente Eclipse e é uma das ferramentas mais completas para esse propósito. Além da inspeção de código (ou "auditoria de código" de acordo com a ferramenta), ela também oferece outras funcionalidades como cálculo de métricas, análise de dependência (entre projetos, pacotes e classes) e reparação de Javadoc [JAVA].

Especificamente na parte de inspeção de código, esta ferramenta conta com mais de 500 regras já pré-definidas. Essas regras representam vários tipos de abordagens, desde formatação do código-fonte até regras que identificam más praticas de programação. Além dessas regras, esta ferramenta permite que sejam criadas novas regras definidas pelo usuário. A criação de novas regras é baseada no modelo de extensão da plataforma Eclipse, ou seja, para criar uma regra, se faz necessário a extensão de "pontos de extensão" definidos pela ferramenta no ambiente Eclipse. Isso implica que o usuário que for definir uma nova regra, além de ter que escrever código Java e conhecer o modelo da AST, precisa ter conhecimento de desenvolvimento de *plug-ins* para o Eclipse.

2.3. PMD

PMD [PMD] é uma ferramenta *open source* para análise de código Java. Esta possui *plug-ins* para diversas ferramentas/IDE, como JDeveloper, Eclipse, JEdit, JBuilder, NetBeans, TextPad, Gel, etc. Ela identifica problemas como:

- Blocos `try`, `catch`, `finally` e `switch` vazios;
- Variáveis locais, parâmetros e métodos privados não utilizados;
- Comandos `if` e `while` vazios;
- Comandos `if` desnecessários;
- *Loops* `for` que poderiam ser *loops* `while`

Seu modelo é baseado na AST do código-fonte. Ela utiliza o JavaCC [JAVACC] para gerar o *parser* e o JJTree [JJTREE] para gerar a AST. A ferramenta conta com várias regras pré-definidas e permite que novas regras sejam incluídas. As novas regras podem ser escritas de duas maneiras: Java e XPath [XPATH].

Para escrever uma regra utilizando a linguagem Java, é necessário criar uma classe Java que estenda a `AbstractRule` do pacote `net.sourceforge.pmd`. Essa classe implementará os métodos `visit` para cada nó da AST de tal forma a identificar o padrão desejado. Por exemplo, a classe `WhileLoopsMustUseBraces` mostrada na FIG. 2.1 define uma regra para encontrar *loops* `while` sem chaves (bloco). Esta classe implementa o método `visit` para o nó `ASTWhileStatement`. Durante a visitação na AST, quando visita um nó desse tipo, este método é executado para identificação da regra.

```
public class WhileLoopsMustUseBracesRule extends AbstractRule {
    public Object visit(ASTWhileStatement node, Object data) {
        SimpleNode firstStmt = (SimpleNode)node.jjtGetChild(1);
        if (!hasBlockAsFirstChild(firstStmt)) {
            RuleContext ctx = (RuleContext)data;
            ctx.getReport().addRuleViolation(createRuleViolation(
                                        ctx, node.getBeginLine()));
        }
        return super.visit(node,data);
    }
    private boolean hasBlockAsFirstChild(SimpleNode node) {
        return (node.jjtGetNumChildren() != 0 &&
                (node.jjtGetChild(0) instanceof ASTBlock));
    }
}
```

FIG. 2.1. Classe WhileLoopsMustUseBraces.

Além dessa classe, para uma regra ser efetivamente incluída se faz necessário adicioná-la ao arquivo de regras, que é um arquivo XML com todas as regras. Neste arquivo, cada regra é definida com um nome, uma mensagem para apresentação ao usuário, o nome da classe que implementa essa regra, uma descrição e um exemplo de código que contenha essa violação. Para o exemplo acima, a FIG. 2.2 apresenta como ficaria o XML para representação dessa regra.

```
<rule name="WhileLoopsMustUseBracesRule"
      message="Avoid using 'while' statements without curly braces"
      class="net.sourceforge.pmd.rules.XPathRule">

      <description>
            Avoid using 'while' statements without using curly braces
      </description>

      <priority>3</priority>

      <example>
            <![CDATA[
                        public void doSomething() {
                              while (true)
                              x++;
                        }

            ]]>
      </example>
</rule>
```

FIG. 2.2. XML para identificação de uma regra.

Outra forma de definir uma regra é através de expressões XPath. Escrever regras desta maneira faz com que seu código seja mais condensado. Para regras simples, é fácil escrever regras através de expressões XPath tornando o código mais legível. Entretanto, para regras mais complexas o uso desta representação é de difícil formulação e compreensão. Ainda para o exemplo acima, a FIG. 2.3 mostra como ficaria uma expressão XPath que representaria essa regra.

```
//WhileStatement[not(Statement/Block)]
```

FIG. 2.3. Expressão XPath para while sem bloco.

2.4. FINDBUGS

FindBugs [FINDBUGS] é uma ferramenta *open source* para identificação de *bugs* em programas Java. Esta é formada por duas partes principais, *FindBugs engine* e *FindBugs front-end*. A última versão da ferramenta (0.9.1) conta com 3 *front-ends*: uma *task* para o Ant, um *plug-in* para o Eclipse e uma aplicação Java *stand-alone*.

A principal diferença entre a FindBugs e as outras ferramentas apresentadas é que esta busca instâncias de *bugs patterns* em código compilado, ou seja, em *bytecode*. Para cada bug, existe um detector que é implementado usando BCEL [BCEL, 1994] (*Byte Code Engineering Library*) que permite a análise, criação e manipulação de classes Java. A distribuição atual da ferramenta conta com mais de 50 detectores e está disponível em http://findbugs.sourceforge.net/. A princípio a arquitetura da ferramenta não permite a inclusão de novos padrões para identificação de *bugs*. Mas, por ser um software livre e de código aberto, nada impede o usuário de incluí-los.

2.5. OUTRAS FERRAMENTAS

Existem outras ferramentas voltadas à detecção de padrões Java, dentre as quais podemos destacar as ferramentas *JCosmo* [EMDEN, 2002], *CoffeeStrainer* [BOKOWSKI, 1999], CheckStyle [CHECKSTYLE] e a *Hammurapi* [HAMMURAPI]. Algumas destas plataformas permitem que o usuário defina novas regras, no entanto, estas devem ser programadas em Java.

Os trabalhos JaTS [CASTOR, 2001] e JPearl [MAIA, 2002] são voltados para a refatoração de programas Java, e também apresentam uma forma declarativa de descrever de padrões. Enquanto JaTS e JPearl procuram simplificar a tarefa de descrição de padrões aproximando estes da sintaxe concreta de Java, o presente trabalho, procura aproximar o usuário da sintaxe abstrata, permitindo gerar facilmente padrões a partir da AST do programa, incluindo dados sensíveis ao contexto usados para decorar a AST, como, por exemplo, o tipo de expressões.

3. ECLIPSE

Este capítulo tem como objetivo apresentar a Plataforma Eclipse [ECLIPSE, 2003] e alguns de seus conceitos. Inicialmente será apresentado um breve histórico da plataforma. Em seguida é mostrada a arquitetura geral da ferramenta e de dois dos principais componentes.

3.1. INTRODUÇÃO AO ECLIPSE

Eclipse é uma plataforma universal para integração de ferramentas para diversos fins. O projeto foi lançado em novembro de 2001 quando a IBM doou o código fonte do *Websphere Studio Workbench* no qual havia investido 40 milhões de dólares. A partir daí, formou-se o *Eclipse Consortium* para gerenciar e continuar o desenvolvimento da plataforma. Fazem parte desse consórcio várias das principais empresas de tecnologia mundial como: Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft, Webgain, Serena, Sybase e Fujitsu. A Plataforma Eclipse foi projetada para as seguintes funcionalidades [ECLIPSE, 2003]:

- Suporte a construção de uma variedade de ferramentas para desenvolvimento de aplicação.
- Suporte a ferramentas para manipular tipos de conteúdo arbitrários (por exemplo, HTML, Java, C, JSP, EJB, XML, etc).
- Facilitar integração de diferentes tipos de ferramenta num mesmo ambiente.
- Suporte GUI e ambientes de desenvolvimento de aplicação.
- Executar em vários sistemas operacionais.

A plataforma propriamente dita não fornece muita funcionalidade para usuários finais. O diferencial da plataforma é que ela favorece o desenvolvimento rápido de recursos integrados com base no modelo de *plug-ins* e fornece um modelo de UI (interface com o usuário) comum para trabalhar com diferentes ferramentas num mesmo ambiente. Uma outra vantagem decorre do fato da plataforma poder ser executada em vários sistemas operacionais, fazendo

com isso que os *plug-ins* possam ser executados de forma inalterada em qualquer um dos sistemas operacionais suportados.

3.2.ARQUITETURA

A plataforma Eclipse é estruturada em torno do conceito de pontos de extensão (*extension points*). Os pontos de extensão são locais bem definidos no sistema onde outras ferramentas (denominadas *plug-ins*) podem contribuir com funcionalidades. Cada subsistema na plataforma pode ser estruturado como um ou mais *plug-ins* que implementam alguma funcionalidade. Os *plug-ins* podem definir seus próprios pontos de extensão ou simplesmente adicionar extensões aos pontos de extensão de outros *plug-ins*. Um *plug-in* é escrito em Java, mas podendo também incluir imagens, arquivos de ajuda, bibliotecas etc. Uma aplicação complexa para o Eclipse pode ser baseada em vários *plug-ins* e pontos de extenção. O EclipseSDK conta com dois *plug-ins* principais:

- JDT – *Java Development Tools* (suporte ao desenvolvimento Java) [JDT]
- PDE – *Plug-in Development Environment* (suporte ao desenvolvimento de *plug-ins*)

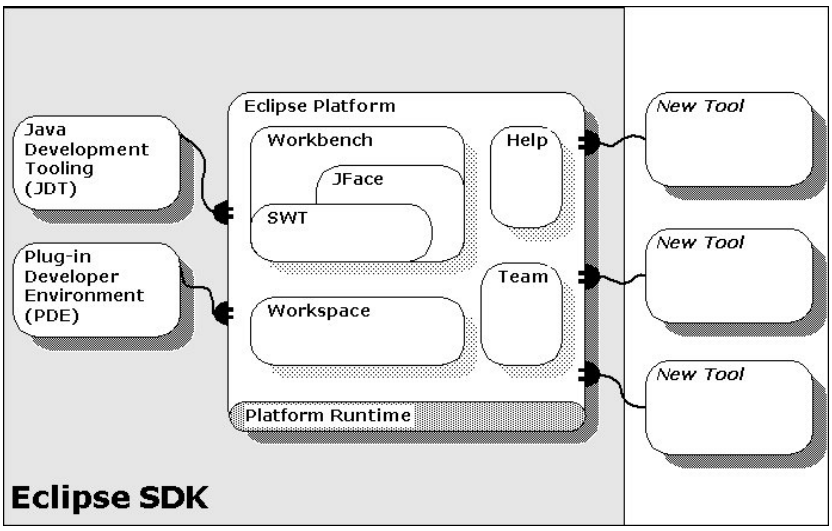


FIG. 3.1. Arquitetura da Plataforma Eclipse.

3.3. JAVA DEVELOPMENT TOOLING (JDT)

O JDT é um conjunto de *plug-ins* que permite que os usuários gravem, compilem, testem, depurem e editem programas escritos na linguagem Java. O JDT utiliza vários pontos de extensão e estruturas de plataforma. Os *plug-ins* que compõem o JDT agregam comportamentos específicos da linguagem Java ao modelo genérico de recursos de plataforma e contribui com *views*, editores, perspectivas e ações específicas para o desenvolvimento em Java na *Workbench*.

Esses *plug-ins* fornecem uma API (*Application Program Interface*) e vários pontos de extensão que permitem escrever outros *plug-ins* que interagem com programas e recursos Java dentro da plataforma Eclipse. Entre as funcionalidades que podem ser desenvolvidas utilizando-se o JDT, destacam-se:

- Manipular programas Java, criar novos projetos, gerar código Java, detectar problemas no código.
- Executar um programa Java da plataforma de maneira programática.
- Incluir novas funções e extensões no próprio IDE Java.

O JDT é formado por três componentes principais:

- **JDT Core** - módulo principal, responsável pela infraestrutura para manipular e compilar códigos Java.
- **JDT UI** - módulo de interface com usuário provida pela IDE (para Java). UI do JDT (`org.eclipse.jdt.ui`) é o *plug-in* que implementa as classes da interface com o usuário específicas do Java que manipulam elementos Java. Os pacotes no UI do JDT implementam as extensões específicas do Java para a *Workbench*.
- **JDT Debug** – módulo de suporte a depuração de um programa. Específico para a linguagem Java.

3.3.1. JDT CORE

O JDT Core (`org.eclipse.jdt.core`) é o *plug-in* que define os elementos Java da API. Essa API é composta por vários pacotes que fornecem acesso os objetos do modelo Java e à infraestrutura da IDE para suporte a Java. Os pacotes do JDT Core incluem:

- **`org.eclipse.jdt.core`** - define as classes que descrevem o modelo Java.
- **`org.eclipse.jdt.core.compiler`** - define a API para a infra-estrutura do compilador.
- **`org.eclipse.jdt.core.dom`** - suporta AST (*Abstract Syntax Trees*) que pode ser utilizada para examinar a estrutura de um programa Java (Compilation Unit).
- **`org.eclipse.jdt.core.eval`** - suporta a avaliação de trechos de código.
- **`org.eclipse.jdt.core.jdom`** - suporta DOM (*Document Object Model*) [XML]. Esse modelo pode ser usado para percorrer a estrutura de uma *Compilation Unit*.
- **`org.eclipse.jdt.core.search`** - suporta a busca de elementos Java na *workbench* que correspondem a uma determinada descrição segundo sua API.
- **`org.eclipse.jdt.core.util`** - fornece classes de utilitário para manipulação de arquivos *.class* e elementos de modelos Java.

O modelo Java é o conjunto de classes que representam os objetos associados à criação, edição e construção de um programa Java. As classes desse modelo estão definidas no pacote `org.eclipse.jdt.core`. O JDT utiliza um modelo hierárquico de objetos em memória para representar a estrutura de um programa Java. A tabela a seguir mostra alguns dos principais elementos desse modelo.

TAB. 3.1. Principais elementos do JDT.

Elemento	**Descrição**
`IJavaModel`	Representa o elemento Java raiz, correspondente à área de trabalho. Ele é o pai de todos os projetos Java.
`IJavaProject`	Representa um projeto Java na área de trabalho.
`IPackageFragmentRoot`	Representa um conjunto de fragmentos de pacote e mapeia os fragmentos para um recurso subjacente que é uma pasta, ou um arquivo JAR.
`IPackageFragment`	Representa a parte da área de trabalho que corresponde a um pacote inteiro ou a uma parte do pacote.
`ICompilationUnit`	Representa um arquivo fonte Java (*.java*).
`IPackageDeclaration`	Representa uma declaração de pacote em uma unidade de compilação

	(`ICompilationUnit`).
`IImportContainer`	Representa a coleção de declarações de importação de pacote em uma unidade de compilação.
`IImportDeclaration`	Representa uma única declaração de importação de pacote.
`IType`	Representa um tipo de origem dentro de uma unidade de compilação ou um tipo binário dentro de um arquivo “.class”.
`IField`	Representa um campo dentro de um tipo.
`IMethod`	Representa um método ou construtor dentro de um tipo.

Alguns desses elementos implementam a interface `IOpenable` e podem ser abertos e navegados na *Package Explorer View* e na *Outline View*. A FIG. 3.2 e FIG. 3.3 mostram esses elementos na plataforma.

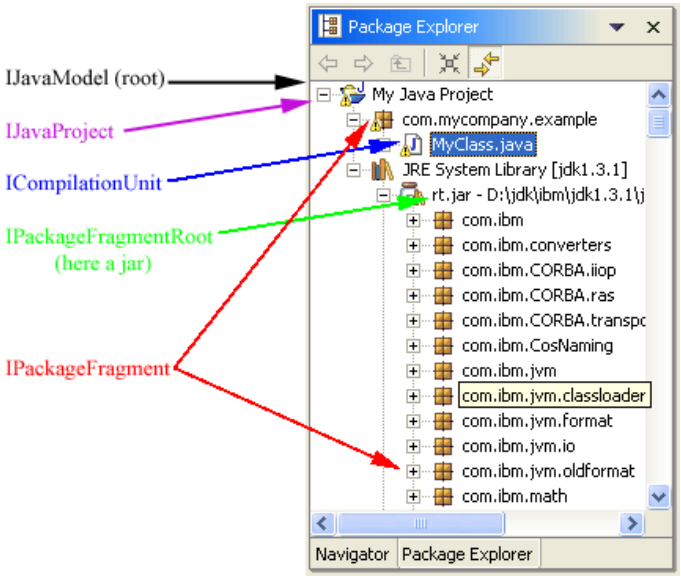


FIG. 3.2. Package Explorer View.

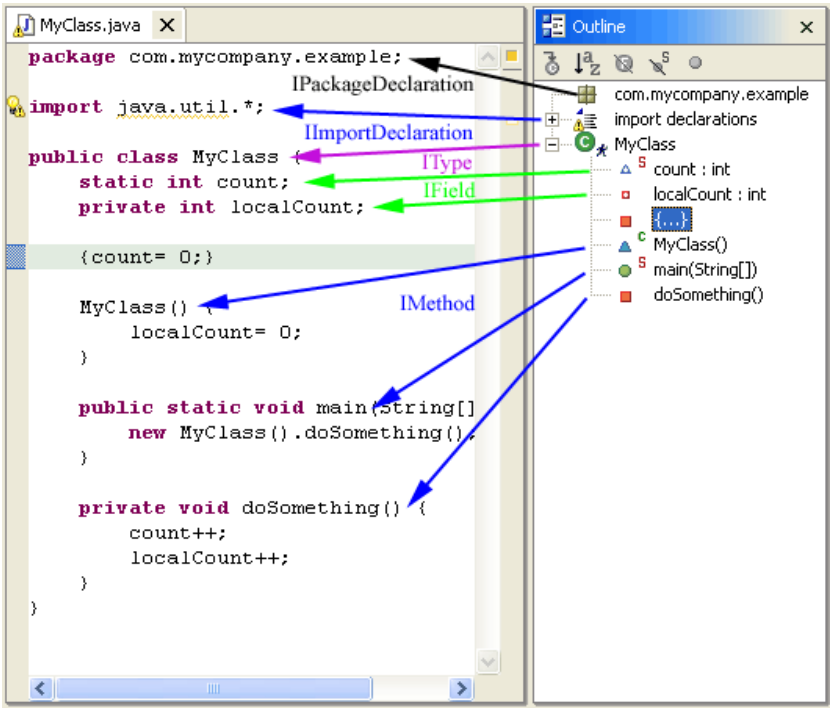


FIG. 3.3. Outline View.

O JDT Core através do pacote `org.eclipse.jdt.core.dom` disponibiliza um modelo de classes Java para representação do código fonte. Este modelo é representado através de uma *Abstract Syntax Tree* (AST) completa que pode ser construída, consultada e manipulada pela API do JDT. As principais classes desse pacote são: `AST`, `ASTParser`, `ASTNode` e `ASTVisitor`. A classe `AST` funciona como uma fábrica (*factory*) e como gestora de nós da *Abstract Syntax Tree*. A classe `ASTParser` faz o *parser* de código fonte Java para criar uma AST. A `ASTNode` é a superclasse (abstrata) de todos os tipos de nós da AST. Cada nó da AST representa um fragmento do código fonte que são essencialmente um nome, um tipo, uma expressão, um comando ou uma declaração. Os tipos de nós da AST são agrupados através de cinco classes abstratas:

- Expressões – `Expression`
- Nomes – `Name` (que é um subtipo de expressão)
- Comandos – `Statement`
- Tipos – `Type`
- Declarações – `BodyDeclaration`

As classes que representam os nós concretos são descritas na API do JDT [JDT].

A classe `ASTVisitor` é uma classe que implementa o *Design Pattern Visitor* [GAMMA, 1994] para percorrer a AST. Para cada tipo nó da AST, existem dois métodos (`visit(ASTNode)` e `endVisit()`) que fazem a visita de um nó na descida e subida respectivamente.

3.4. PLUG-IN DEVELOPMENT ENVIRONMENT (PDE)

O *Plug-in Development Environment* (PDE) é uma ferramenta que foi projetada para auxiliar os desenvolvedores na criação, teste, depuração e desenvolvimento de *plug-ins* na plataforma Eclipse. Como dito anteriormente, ele também é um *plug-in* dentro da *workbench* da plataforma e não pode ser executado separadamente. O PDE fornece um conjunto de contribuições na plataforma (exibições, editores, perspectivas, etc.) que juntas, organizam o processo de desenvolvimento de *plug-ins*.

Um *plug-in* é composto por uma classe responsável em gerenciar seu ciclo de vida, um arquivo XML de manifesto (*manifest file*) que sempre possui o nome "*plugin.xml*" e as classes que realmente implementam suas funcionalidades. Durante a inicialização, o Eclipse lê todos os manifestos de *plug-ins* implantados no diretório `ECLIPSE_HOME/plugins`. As informações relativas a todos *plug-ins* encontrados são armazenadas em um registro que fica disponível para todos outros *plug-ins* em tempo de execução. Os *plug-ins* não são carregados na inicialização do Eclipse, permitindo que o tempo de abertura do Eclipse seja constante, independente do número de *plug-ins* instalados. Os *plug-ins* e suas respectivas classes só são carregados quando estritamente necessário (*lazy loading*).

3.4.1. CRIANDO UM PLUG-IN

Para exemplificar a construção de um *plug-in*, os passos a seguir mostram como criar uma contribuição na Plataforma Eclipse através do *Hello IME* que é um *plug-in* que estende a

plataforma através de uma *action* (ação) na *workbench*. Essa ação poderá ser executada pelo menu ou pela barra de ferramentas do ambiente.

Passo 1 – Criar um *Plug-in Project*

Para se criar um *plug-in*, o primeiro passo é criar um *Plug-in Project*. O Eclipse através do PDE já oferece toda a estrutura para esta tarefa. O caminho no menu para executar essa tarefa é: `File->New->Project->Plug-in Project`. Quando feito isso, será solicitada várias informações referentes ao *plug-in*: nome do projeto, nome do *plug-in*, versão, ID, *provider*, etc (FIG. 3.4).

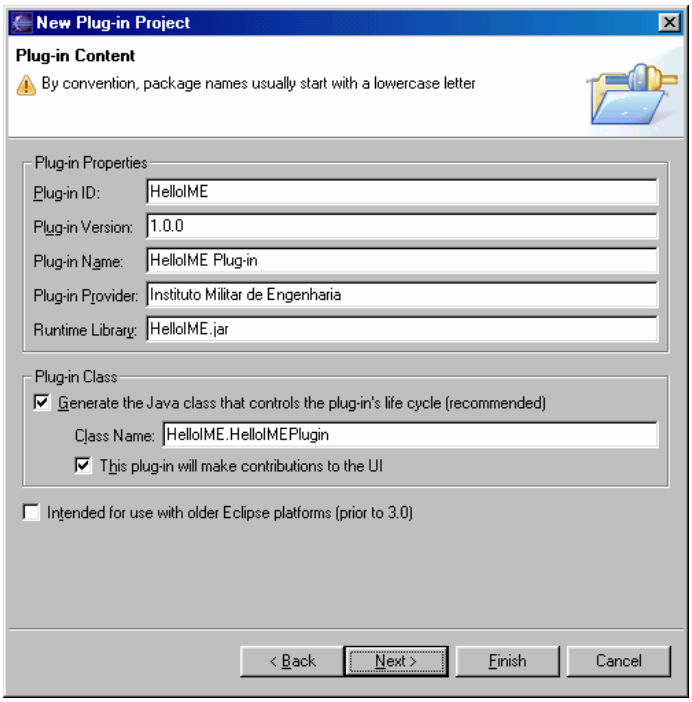


FIG. 3.4. Criação de um *Plug-in Project.*

Passo 2 – Definir como *o plug-in* vai interagir com o Eclipse

Antes de começar a escrever código para realmente implementar o *plug-in* é necessário determinar como ele se integrará ao Eclipse. Isso é feito definindo qual ponto de extensão será utilizado para a integração. Para esse exemplo, será inserido um botão na barra de ferramentas

para executar uma *action*, e o ponto de extensão que será usado para isso é o `org.eclipse.ui.actionSets`.

Passo 3 – Criar o arquivo de manifesto

Quando se cria um *Plug-in Project*, o PDE cria um arquivo de manifesto padrão com as informações fornecidas para criação do projeto. Esse arquivo precisa ser editado para definir quais pontos serão estendidos, para definir outros pontos de extensão (se for o caso), para definir qual a classe principal, etc. A figura abaixo mostra o arquivo de manifesto para nosso exemplo.

```
<plugin
   id="HelloIME"
   name="HelloIME Plug-in"
   version="1.0.0"
   provider-name="Insituto Militar de Engenharia"
   class="HelloIME.HelloIMEPlugin">

   <runtime>
      <library name="HelloIME.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
   </requires>

   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Hello IME Action Set"
            visible="true"
            id="HelloIME.actionSet">
         <menu label="&amp;amp;IME" id="imeMenu">
            <separator name="imeGroup"/>
         </menu>
         <action
               label="&amp;amp;Hello IME Action"
               icon="icons/sample.gif"
               class="HelloIME.actions.HelloIMEAction"
               tooltip="Hello IME"
               menubarPath="imeMenu/imeGroup"
               toolbarPath="imeGroup"
               id="HelloIME.actions.HelloIMEAction">
         </action>
      </actionSet>
   </extension>
</plugin>
```

FIG. 3.5. Arquivo de manifesto (*plugin.xml*).

A primeira parte do arquivo de manifesto é a declaração do *plug-in* com várias informações (nome, ID, classe principal, *provider*, etc). Um ponto importante a ser destacado é que o ID é a forma com que a plataforma identifica o *plug-in*. Ele deve ser único. Caso existam dois ou mais *plug-ins* com mesmo ID, haverá um conflito. O elemento `runtime` especifica para a plataforma onde estão as classes que implementam o *plug-in*. `requires` é o elemento no qual estão especificados todos os *plug-ins* que esse *plug-in* tem dependência.

Nesse exemplo estaremos usando a extensão `actionSets` do *plug-in* `org.eclipse.ui`. O elemento `extension` é o elemento que realmente descreve a integração com a plataforma. Nele, através do atributo `point`, é especificado qual ponto de extensão está sendo "estendido". Como dito anteriormente, o ponto é o `org.eclipse.ui.actionSets` do *plug-in* `org.eclipse.ui`.

Passo 4 – Implementar a interface IWorkbenchWindowActionDelegate

Para programar realmente a funcionalidade do *plug-in* para o ponto de extensão em questão, precisa-se criar uma classe que implementa a interface `IWorkbenchWindowActionDelegate`. O nome da classe e a estrutura de pacotes do projeto devem coincidir com o que foi declarado no arquivo de manifesto. A figura abaixo mostra um exemplo de implementação para essa funcionalidade.

```
package HelloIME.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

public class HelloIMEAction implements IWorkbenchWindowActionDelegate {
  private IWorkbenchWindow window;

  public HelloIMEAction() {}

  public void run(IAction action) {
    MessageDialog.openInformation(window.getShell(),
                    "HelloIME Plug-in",
                    "Hello Instituto Militar de Engenharia !!!");
    }

  public void init(IWorkbenchWindow window) {
    this.window = window;
  }
}
```

FIG. 3.6. Classe que implementa IWorkbenchWindowActionDelegate.

Passo 5 – Executar o *plug-in*

Um conceito importante no PDE é o conceito de instâncias da *workbench*, de *host* e de *tempo de execução*. Ao inicializar a *workbench*, você vai utilizá-la para trabalhar com projetos que definem os *plug-ins* que estiverem sendo construídos. A instância da *workbench* que está executando enquanto desenvolve um *plug-in* utilizando o PDE é chamada de *host*. Os recursos disponíveis nessa instância virão exclusivamente dos *plug-ins* que estiverem instalados com seu aplicativo. Para testar um *plug-in* deve-se executá-lo como uma *Run-time Workbench*, lançando outra instância da *workbench*, a de tempo de execução. Essa ocorrência conterá os mesmos *plug-ins* que a ocorrência *host*, mas também terá o *plug-in* que se deseja testar. O *launcher* do PDE cuidará da combinação do *plug-in* que está sendo "*executado*" com os *plug-ins* de *host* e da criação da instância de tempo de execução.

A FIG. 3.7 mostra a *workbench* de tempo de execução para o exemplo que foi construído. Ao clicar no menu `IME -> Hello IME Action` uma caixa de dialogo é lançada na tela.

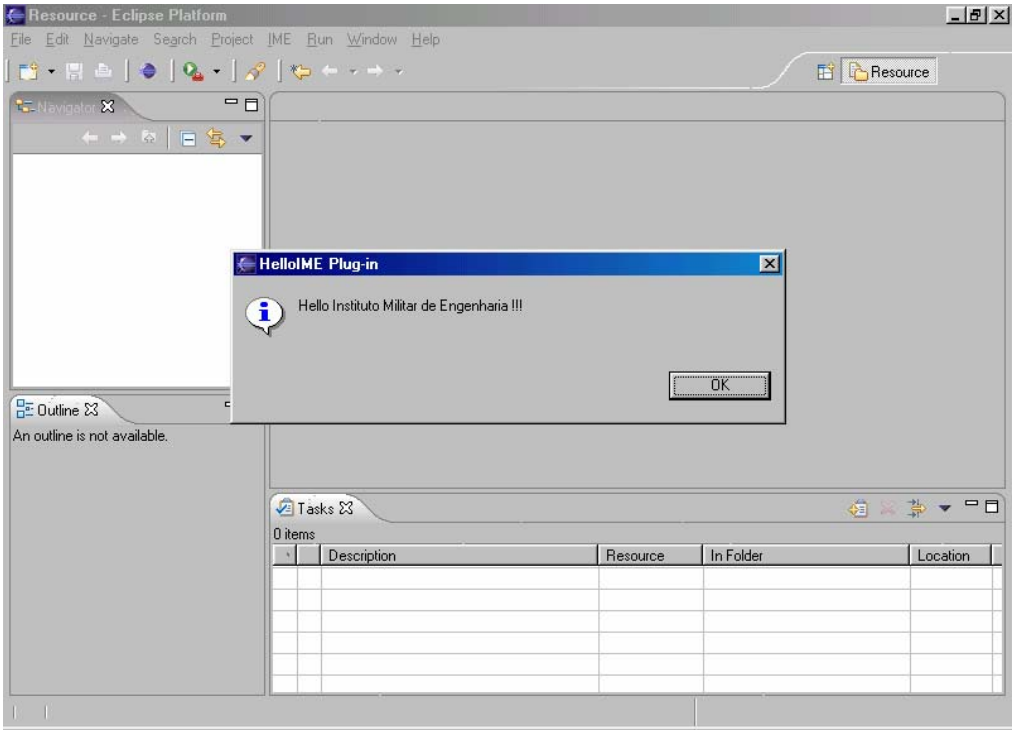


FIG. 3.7. Run-time Workbench.

4. A LINGUAGEM PARA DESCRIÇÃO DE PADRÕES

O principal diferencial da ferramenta desenvolvida é a utilização de uma linguagem que representa padrões de forma simples e objetiva. Neste capítulo será apresentada a linguagem para representação de Padrões de Código Java, a qual foi chamada de PDL – *Pattern Description Language*. Essa linguagem tem sintaxe XML [*XML*] fazendo com que os padrões sejam estruturados na forma de árvores. O trabalho de detecção destes padrões em um programa é simplesmente encontrar homomorfismos entre estes padrões e a AST do programa. No decorrer desse capítulo, será apresentada uma visão geral da linguagem PDL, os elementos que compõe a linguagem, suas principais características e vários exemplos ilustrativos de códigos PDL.

4.1. REPRESENTAÇÃO DE UM PADRÃO EM PDL

Como dito anteriormente, padrões PDL serão descritos através de documentos XML onde cada elemento do documento corresponderá a um nó da AST do programa, ou seja, cada nome de *tag* de um elemento corresponderá ao nome de um nó na árvore sintática. O modelo de AST utilizado será o modelo da Plataforma Eclipse no pacote *org.eclipse.jdt.core.dom* [*JDT*]. Esse modelo possui um conjunto de classes que modelam códigos fontes de programas Java. Entre essas classes, a `AST` é a responsável por representar a árvore de derivação do programa, que pode ser obtida através da classe `ASTParser` que faz a análise sintática do código fonte Java e retornando a AST. Cada nó da AST é um objeto de uma subclasse de `ASTNode`, ou seja, a `ASTNode` é superclasse de todos os nós da AST que representam um nome, um tipo, uma expressão, um comando ou uma declaração. A FIG. 4.1 (b) é uma representação gráfica do modelo da AST da Plataforma Eclipse para o código mostrado na FIG. 4.1 (a).

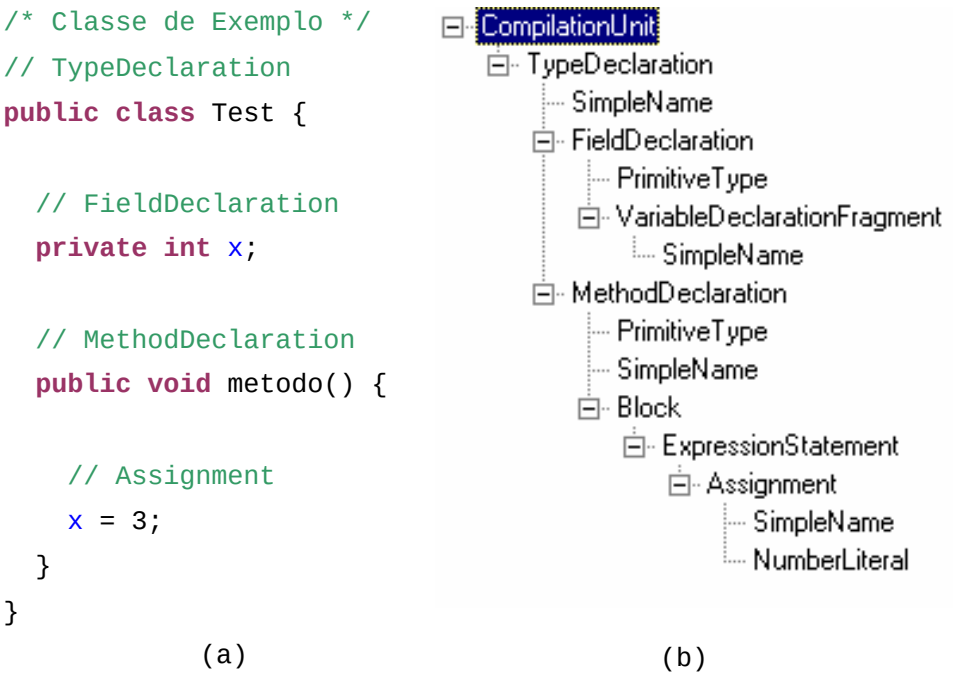


FIG. 4.1. (a) Classe em Java. (b) Esquema da AST gerada para o código (a).

É relativamente simples escrever padrões PDL estritamente sintáticos quando se conhece o modelo da AST. Para isso, basta criar elementos XML com o nome de cada nó da AST, colocando-os numa hierarquia desejada. Por exemplo, para o nó `MethodDeclaration`, o elemento XML correspondente será: `<MethodDeclaration> ... </MethodDeclaration>` ou simplesmente `<MethodDeclaration/>` caso não seja necessário ao padrão a descrição interna desse nó. A FIG. 4.2 mostra um padrão PDL e sua representação gráfica. Esse padrão representa um método que possui um '*if*' dentro de um '*while*'. (As regras de casamento do padrão PDL com a AST serão descritas no capítulo 5)

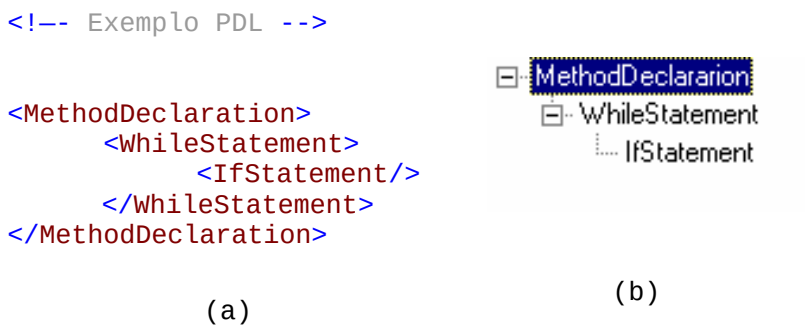


FIG. 4.2. (a) Padrão em PDL. (b) Esquema para representação do padrão (a).

Podemos interpretar os padrões PDL como árvores e o algoritmo de busca como a procura de um homomorfismo entre a árvore que representa o padrão e a árvore sintática do programa. Alternativamente, podemos interpretar o padrão PDL como uma forma sentencial, com um não terminal único do lado esquerdo, que é a raiz do padrão, e com os não-terminais do lado direito sendo as folhas do padrão.

4.2. ATRIBUTOS EM PDL

Cada nó da AST possui atributos que podem ser expressos em PDL. Esses representam pontos semânticos de um programa Java. Por exemplo, atributos que representam o tipo para uma expressão, o nome de uma determinada variável, modificadores (`public`, `static`, `final`) de um determinado método, etc. A FIG. 4.3 mostra um padrão PDL que usa o atributo `isPrivate` para o elemento `<MethodDeclaration>`.

```
<!—- Exemplo PDL -->

<MethodDeclaration isPrivate=”true”>
    <IfStatement/>
    <ForStatement/>
</MethodDeclaration>
```

FIG. 4.3. Exemplo de código PDL com atributo.

Um exemplo de código-fonte contendo o padrão descrito acima e a respectiva AST pode ser visto na FIG. 4.4. Observe que, para haver casamento do padrão, é necessário que a visibilidade do método seja `private`.

```java
public class Figura44 {

	//MethodDeclaration
	private void metodo01() {

		//IfStatement
		if(true);

		//ForStatement
		for(;;);

	}
}
```

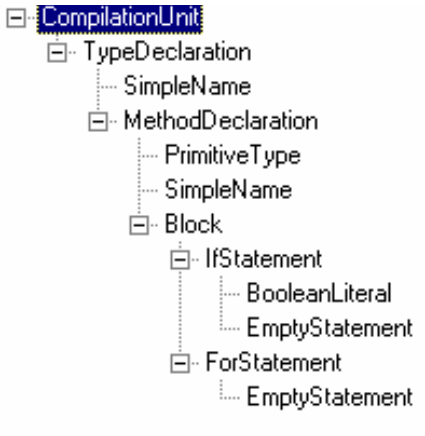


FIG. 4.4. Exemplo de código Java e sua AST que contém o padrão da FIG.4.3.

Os atributos PDL são representados como atributos XML, um identificador, o operador '=' e o valor do atributo entre aspas duplas. Em PDL, o valor dos atributos são expressões regulares as quais, serão casadas com o valor do atributo da AST. A FIG. 4.5 mostra um padrão PDL com o atributo *name*. Esse padrão representa declarações de métodos que iniciam com letra maiúscula.

```
<MethodDeclaration   name="[A-Z].*"
                     isContructor="false"
                     _Mark="Metodos nao devem começar
                            com letra Maiuscula"
/>
```

FIG. 4.5. Código PDL com atributo *name*.

Para exemplificar uma instância desse padrão, a FIG. 4.6 mostra um trecho de código Java com duas declarações de métodos. O primeiro método (`MetodoIniciadoComMaiuscula`) é uma instância desse padrão e o segundo (`metodoIniciadoComMinuscula`) não pois seu nome não é reconhecido pela expressão regular `[A-Z].*`.

```java
...
private void MetodoIniciadoComMaiuscula() { ...}
...
private void metodoIniciadoComMinuscula() { ...}
...
```

FIG. 4.6. Código Java com métodos iniciados com letra maiúscula e minúscula.

4.3. ATRIBUTOS ESPECIAIS

Além dos atributos simples já presentes na AST, PDL possui outros atributos chamados de “especiais”. Esses atributos são representados da mesma maneira que os outros, com a diferença de iniciarem com o caracter ‘_’ (sublinhado). Esses atributos são: `_Mark` e o `_Bind, _InParentSubTree, _MaxLevelSubTree e o _Not.`

4.3.1. ATRIBUTO _MARK

O atributo `_Mark` tem a finalidade de inserir uma marca no ambiente Eclipse. Ele é utilizado pela ferramenta de detecção para inserir marcas na *Problems View* do Eclipse toda vez que um determinado padrão for detectado. Um padrão pode gerar várias marcas, desde que seja respeitada a restrição de uma marca por elemento de um arquivo PDL. O valor atribuido a cada marca será o texto de descrição na *Problems View*. As FIG .4.7 e FIG. 4.8 mostram respectivamente, um exemplo de código PDL que usa esse atributo e a *Problems View* com esse padrão detectado. Quando o usuário clica numa marca, o fragmento de código onde essa marca foi detectada é selecionado no editor Java da Plataforma Eclipse.

```
<CompilationUnit>
    <ImportDeclaration isOnDemand="true"
                       _Mark="Importe Classes e nao Packages"
    />
</CompilationUnit>
```

FIG. 4.7. Código PDL com o atributo _Mark.

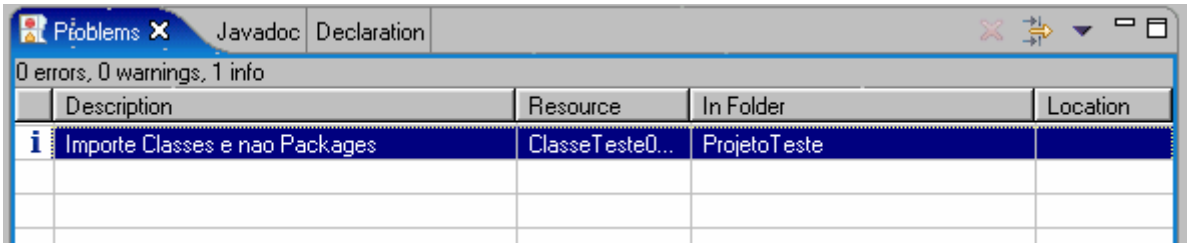


FIG. 4.8. *Problems View* com uma marca.

4.3.2. ATRIBUTO _BIND

O atributo `_Bind` é utilizado para ligar atributos entre dois ou mais elementos. Esse atributo possui dois parâmetros, um identificador e o atributo a ser ligado. Esses parâmetros são separados por vírgula dentro do valor atribuído para esse atributo. A FIG. 4.9 mostra um exemplo de código PDL com esse atributo onde os atributos `identifier` dos nós `SimpleName` são ligados. Essa ligação é uma restrição simples, fazendo com que esses elementos somente sejam casados se possuirem o mesmo valor para os atributos ligados.

```
<TypeDeclaration>
  <FieldDeclaration>
    <VariableDeclarationFragment>
        <SimpleName _Bind="X,identifier" />
    </VariableDeclarationFragment>
  </FieldDeclaration>
  <MethodDeclaration _Mark="Nome do metodo coincide
                            com atributo da classe">
      <SimpleName _Bind="X,identifier"
                  _MaxLevelSubTree="1" />
  </MethodDeclaration>
</TypeDeclaration>
```

FIG. 4.9. Código PDL com o atributo _Bind.

Especificamente para o exemplo da FIG. 4.9, os atributos *identifier* dos elementos `SimpleName` ligados devem possuir o mesmo valor, ou seja, o identificador da variável declarada (atributo da classe) deve coincidir com o nome do método. A FIG. 4.10 mostra um exemplo de código Java que possui uma instância do padrão acima e sua AST. Note que o casamento se dá para o segundo método (`nomeIgual`) e não existe casamento para o primeiro método (`nomeDiferente`), apesar de sintaticamente ser igual.

```
public class ExemploDoBind {
      int nomeIgual;
      public int nomeDiferente() {...}
      public boolean nomeIgual() {...}
}
```

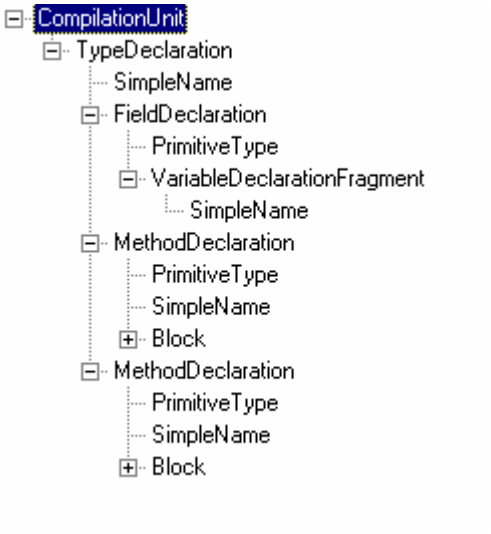


FIG. 4.10. Código Java para contendo um casamento com o padrão da FIG.4.9.

4.3.3. ATRIBUTO _INPARENTSUBTREE

Como dito anteriormente, cada nó da AST do modelo `org.eclipse.jdt.core.dom` possui atributos que podem ser utilizados diretamente no código PDL. Além desses atributos simples, vários nós da AST também possuem atributos que representam suas sub-árvores. Esses atributos são do tipo `ASTNode` e são utilizados em PDL através do `_InParentSubTree`, que faz com que um determinato elemento PDL esteja restrito a uma sub-árvore do elemento pai. A FIG. 4.11 mostra um código PDL que usa o `_InParentSubTree` para restringir o elemento `VariableDeclaration` a sub-árvore `thenStatement` do elemento `IfStatement`.

```
<!-- Exemplo PDL -->

<MethodDeclaration isFinal="false">
     <IfStatement>
         <VariableDeclarationStatement
                              _InParentSubTree="thenStatement"/>
     </IfStatement>
</MethodDeclaration>
```

FIG. 4.11. Código PDL com o atributo especial _InParentSubTree.

O trecho de código Java da FIG. 4.12 mostra a declaração de um método com dois *ifs* aninhados e a AST correspondente. Para esse caso existe apenas uma instância do padrão da FIG. 4.11, que é o *if* mais interno. Isso porque o comando de declaração de variável (`VariableDeclarationStatement`) para o primeiro *if* está na sub-árvore `elseStatement` (o segundo filho `Block` do `IfStatement`) e não na `thenStatement` (primeiro filho `Block`), o que não ocorre com o *if* mais interno.

```
...
public void metodoNaoFinal() {
      if(true) {
            naoEhEsseIf = true;
      } else {
            if (true) {
                  boolean ehEsseIf;
            }
      }
}
...
```

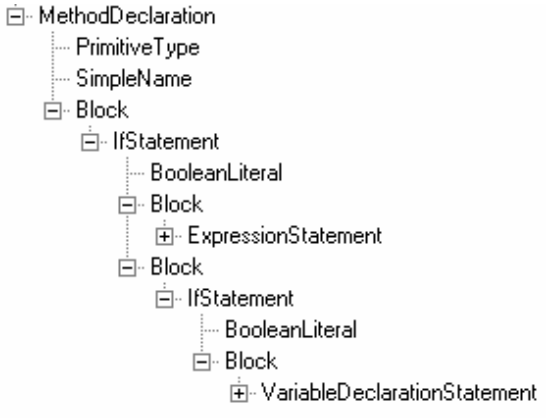


FIG. 4.12. Código Java e sua AST para exemplo do atributo especial _InParentSubTree.

4.3.4. ATRIBUTO _MAXSUBTREELEVEL

O atributo `_MaxLevelSubTree` é utilizado para restringir um elemento a um determinado nível na sub-árvore do elemento pai, isto é, a uma determinada altura na AST. A FIG. 4.13 mostra um exemplo de código PDL onde o elemento `EmptyStatement` está restrito ao nível imediatamente abaixo (nível 1) do elemento pai.

```
<MethodDeclaration>
  <IfStatement _Mark="If com corpo vazio">
      <EmptyStatement _MaxSubTreeLevel="1" />
   </IfStatement>
</MethodDeclaration>
```

FIG. 4.13. Código PDL com o `_MaxLevelSubTree`.

A FIG. 4.14 mostra um fragmento de código Java que contém o padrão acima. Para esse exemplo, existe uma única instância do padrão que ocorre com o casamento do elemento `IfStatement` mais interno (que satisfaz a restrição do atributo `_MaxSubTreeLevel`). Se não houvesse o atributo `_MaxSubTreeLevel` para elemento `EmptyStatement`, haveria duas instâncias desse padrão nesse fragmento de código (os dois `IfStatement` com o `EmptyStatement`).

FIG. 4.14. Código Java e sua AST para o exemplo do `_MaxLevelSubTree`.

4.3.5. ATRIBUTO _NOT

Para representar padrões em PDL em que um determinado elemento não deva existir na AST, faz-se o uso do atributo _Not para esse elemento. A FIG. 4.15 mostra um padrão PDL onde o elemento `Block`, filho imediato do `ForStatement` não possui elementos filhos. Esse padrão representa comandos *for* com o corpo vazio. A FIG. 4.16 mostra um exemplo de código Java contendo esse padrão. Repare que o segundo `for` não casaria, pois possui um elemento (`ReturnStatement`) como filho.

```
<MethodDeclaration>
  <ForStatement _Mark="For com corpo vazio">
    <Block _MaxSubTreeLevel="1">
        <ASTNode _Not="true" />
    </Block>
  </ForStatement>
</MethodDeclaration>
```

FIG. 4.15. Padrão PDL com o atributo _Not.

```
public void metodo() {
      ...
      //ForStatement
      for(;i>0;) {
         //Nenhum elemento
      }
      ...
      for(;;) {
         //Elemento ReturnStatement
         return;
      }
      ...
}
```

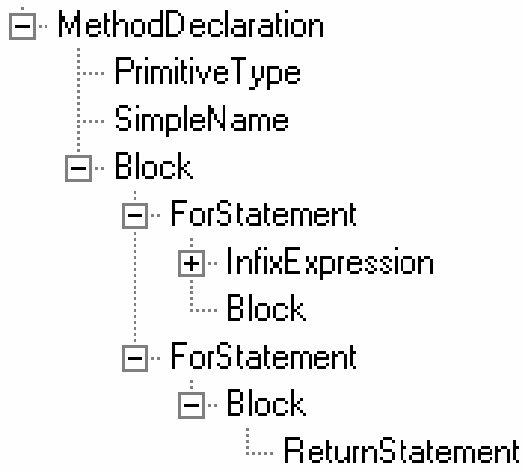


FIG. 4.16. Código Java e sua AST para o exemplo do elemento _Not.

4.4. ELEMENTO OR

Afora os elementos que representam um nó da AST, PDL possui mais um tipo de elemento, o *'Or'* (`<Or>`). Esse elemento realiza a função "ou" entre elementos. A FIG. 4.17 mostra um exemplo de padrão PDL que usa `<Or>` para representar o padrão que define elementos `TryStatement` dentro de *loops* (*for* ou *while*). Cada filho imediato do elemento `Or` é uma opção de casamento para o algoritmo de detecção de padrões.

```
<MethodDeclaration>
  <Or>
    <ForStatement _Mark="Evite blocos 'try' dentro de loops">
        <TryStatement />
    </ForStatement>
    <WhileStatement _Mark="Evite blocos 'try' dentro de loops">
        <TryStatement />
    </WhileStatement>
  </Or>
</MethodDeclaration>
```

FIG. 4.17. Exemplo de código PDL com Or.

Um exemplo de código-fonte contendo uma instância desse padrão (casando o `ForStatement`) e a respectiva AST pode ser visto na FIG. 4.18.

```
public class ElementoOr {

      public void metodoCasandoFor() {

            for(;;) {
                  try {
                  } catch (Exception e) {
                  }
            }
      }
}
```

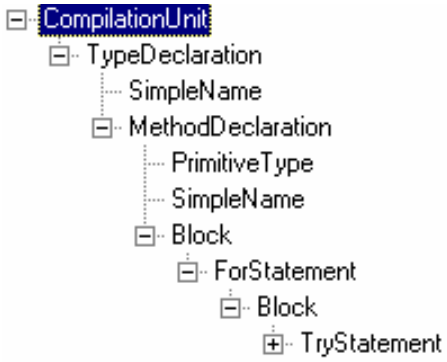


FIG. 4.18. Código Java para exemplo do elemento Or.

5. FERRAMENTA DE DETECÇÃO DE PADRÕES

Este capítulo apresenta a ferramenta para detecção de padrões, que foi desenvolvida totalmente integrada ao ambiente de desenvolvimento integrado (IDE) Eclipse. Conforme características desse IDE descritas no capítulo 3, a ferramenta foi desenvolvida como um *plug-in* para essa plataforma.

A escolha da Plataforma Eclipse como IDE foi motivada por várias características presentes nesse IDE, entre elas, pelo seu suporte ao desenvolvimento em Java, pela sua portabilidade de Sistema Operacional, por ser extensível, *Open Source* e pela grande aceitação que vem tendo junto a comunidade de desenvolvedores (segundo pesquisa da The Middleware Company, 34% utilizam o ambiente Eclipse [MIDDLEWARE RESEARSH, 2004]). Um outro fator importante para essa escolha foi o suporte dado através de APIs para obteção e manipulação de ASTs de código Java dentro da plataforma.

O grande diferencial da ferramenta descrita nesse capítulo em relação às outras ferramentas citadas no capítulo 2 é a possibilidade da fácil definição e inclusão de novos padrões. Um outro diferencial da ferramenta é a integração com um IDE, possibilitando que a inspeção seja feita sem que o desenvolvedor necessite trocar o seu ambiente de trabalho para realização da mesma.

5.1. ARQUITETURA DA FERRAMENTA

A ferramenta foi desenvolvida como um *plug-in* para a plataforma Eclipse. Ela é composta basicamente de um extrator de modelo para a linguagem PDL, algoritmo para detecção de padrões, uma visão para AST do código fonte Java (*AST View*) e uma visão onde os padrões a serem detectados são selecionados (*Pattern Select View*). A FIG. 5.1 mostra o ambiente Eclipse com a ferramenta integrada.

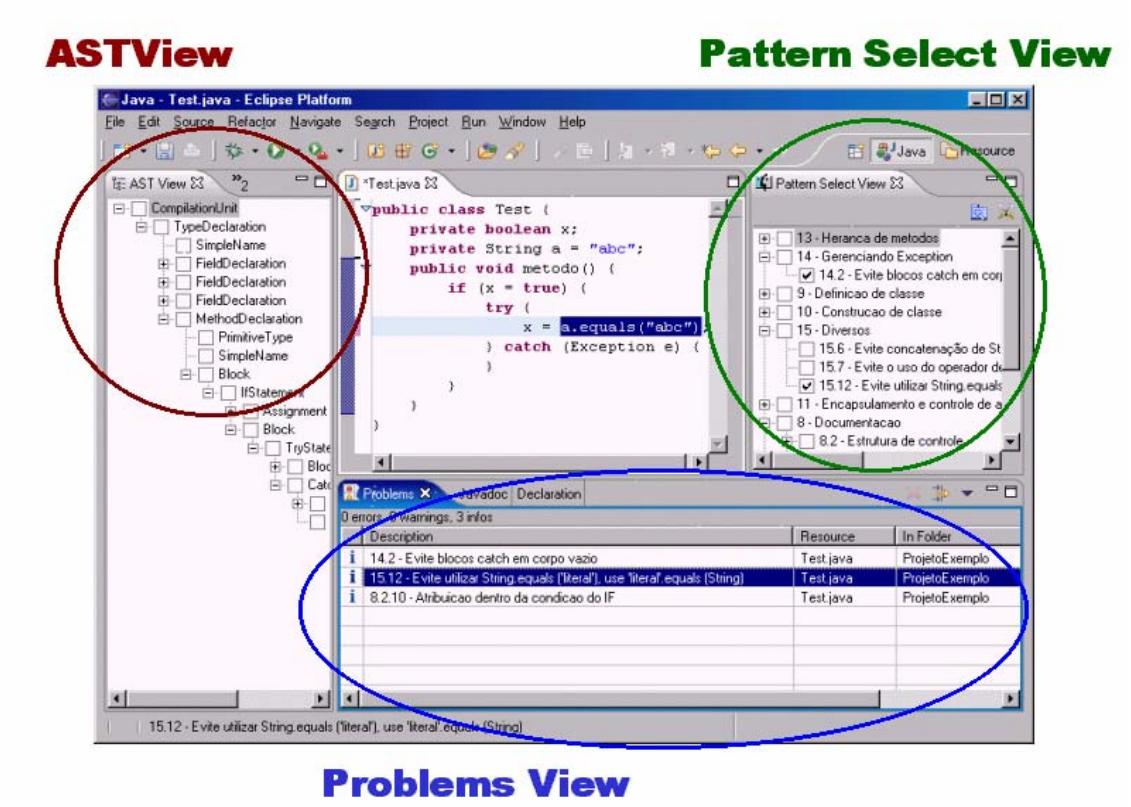


FIG. 5.1. Ferramenta integrada ao Eclipse.

Para realizar a detecção dos padrões, inicialmente são selecionados padrões na *Pattern Select View*. Após isso a ferramenta obtém a AST do programa fonte através de chamada de função do JDT (componente do EclipseSDK), em seguida faz o *parse* dos arquivos PDL selecionados para obter sua representação em memória (*PatternModel*). A partir daí, o algoritmo de detecção é chamado, recebendo como parâmetros a AST do programa Java e os *PatternModel* de cada padrão selecionado para detecção. A FIG. 5.2 ilustra a arquitetura da ferramenta.

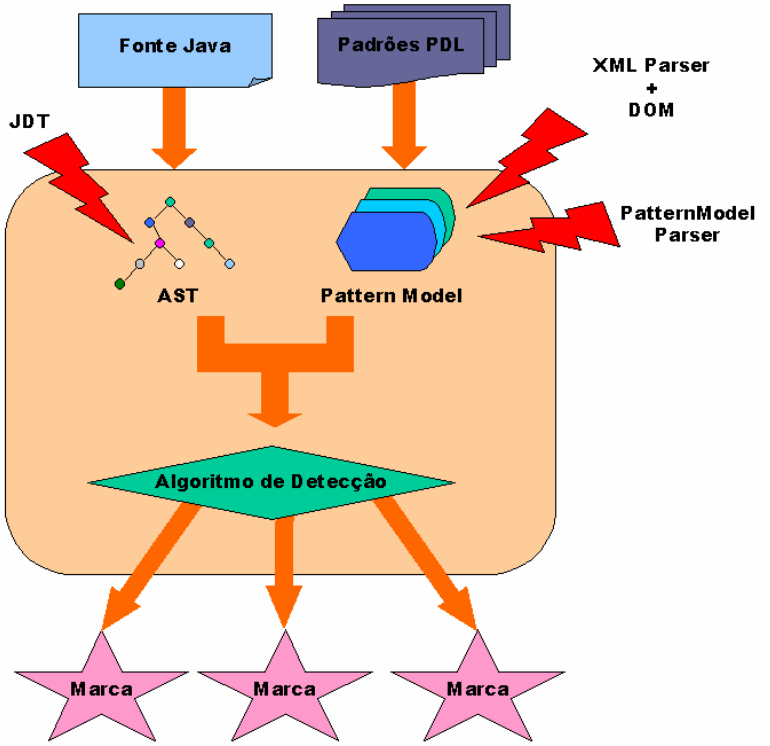


FIG. 5.2. Arquitetura da ferramenta.

5.2. MODELO DE REPRESENTAÇÃO DOS PADRÕES PDL

Padrões PDL são escritos como documentos XML válidos. Para a ferramenta de detecção de padrões esses documentos são mapeados em objetos segundo um modelo de classes. Para fazer esse mapeamento, inicialmente é utilizado um *Parser* XML para obter a representação em memória do documento XML. Esse *parser* retorna um objeto `Document` que é uma representação hierárquica dos elementos XML e através da API DOM esse objeto pode ser percorrido e manipulado como uma árvore.

A partir daí, essa árvore DOM (*Document Object Model*) é percorrida e mapeada em objetos que representam o modelo de padrões definido pela ferramenta, chamdado de "*PatternModel*". Esse modelo também tem a estrutura de uma árvore e é formado por três classes:

`PatternTree` – representa um padrão (árvore do padrão)

`PatternNode` – representa um elemento do padrão (nó da árvore)

`ForeignNode`. – modela a dependência entre nós (e padrões)

5.3. ALGORITMO DE DETECÇÃO

A idéia geral do algoritmo para detecção de padrão é simplesmente encontrar homomorfismos entre estes padrões e a AST do programa. Conforme ilustrado na FIG. 5.2, o algoritmo de detecção recebe como entradas uma AST e uma coleção de padrões (*PatternModel*) a serem detectados. O algoritmo pode ser considerado eficiente no sentido de detectar todas ocorrências dos padrões da coleção percorrendo a AST uma única vez, independentemente do número de padrões e do número de ocorrências destes. Para percorrer a AST, é utilizado um modelo (`Visitor`) que estende o modelo de visitação da AST definido pela plataforma Eclipse. Para cada sub-árvore da AST, a visitação é feita da seguinte maneira:

- Visita a raiz na descida
- Visita recursivamente cada sub-árvore filha na ordem
- Visita a raiz na subida

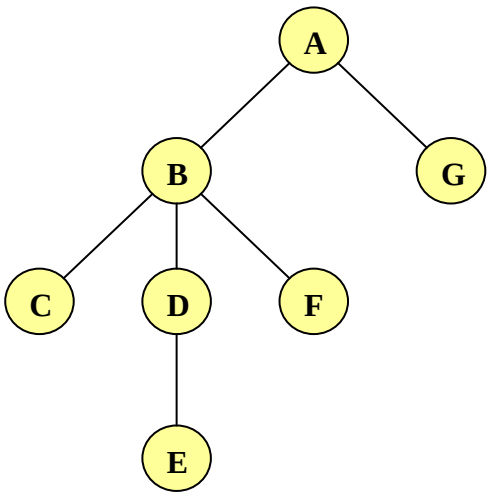


FIG. 5.3. Exemplo de uma árvore.

Para a árvore da FIG. 5.3, a visitação será feita da seguinte maneira:

- Visita o nó A na descida
- Visita o nó B na descida
- Visita o nó C na descida
- Visita o nó C na subida
- Visita o nó D na descida
- Visita o nó E na descida

- Visita o nó E na subida
- Visita o nó D na subida
- Visita o nó F na descida
- Visita o nó F na subida
- Visita o nó B na subida
- Visita o nó G na descida
- Visita o nó G na subida
- Visita o nó A na subida

O modelo de um padrão também possui a estrutura de uma árvore. A classe `PatternTree` representa a árvore de um padrão. Essa classe é composta pelos nós e um ponteiro (chamado de `current`) para o próximo nó a ser casado. Além disso, cada nó do padrão guarda uma referência ao nó da AST com qual está casado.

Iniciada a busca dos padrões, a cada visita a um nó na descida da AST, o algoritmo compara esse nó da AST com `current` de cada `PatternTree` da coleção de padrões. Se a comparação for bem sucedida para algum `current`, é feita uma cópia desse objeto inserindo-o na coleção de padrões a serem detectados. A finalidade dessa cópia é encontrar todas as instâncias de um mesmo padrão na AST. Assim, cada nó casado gera um novo padrão na coleção de padrões a serem detectados, este padrão é posteriormente descartado, conforme será visto na explicação do algoritmo. Apesar da duplicação de padrões afetarem o desempenho do algoritmo de casamento, para situações reais, esse fato não implica em uma degradação significativa no desempenho do algoritmo, pois o tamanho do padrão é pequeno em relação à AST. O algoritmo percorre a AST uma única vez, portanto, fixando-se o padrão, a complexidade do algoritmo é linear em relação ao tamanho da AST.

Os objetos para os quais a comparação foi bem sucedida são então atualizados. Se o nó do padrão tem um filho, o `current` passa a ser o filho. Caso não tenha, a atualização do `current` será feita na subida desse nó, passando o `current` para o próximo irmão, tio, etc. A cada visita na subida a um nó da AST é verificado se algum casamento está completo. A visita na subida também checa se algum padrão não pode mais ser encontrado, ou seja, o atributo nó da AST casado com o pai do elemento `current` é o nó que está sendo visitado. Nesses casos, esses padrões são retirados da coleção de padrões a serem detectados.

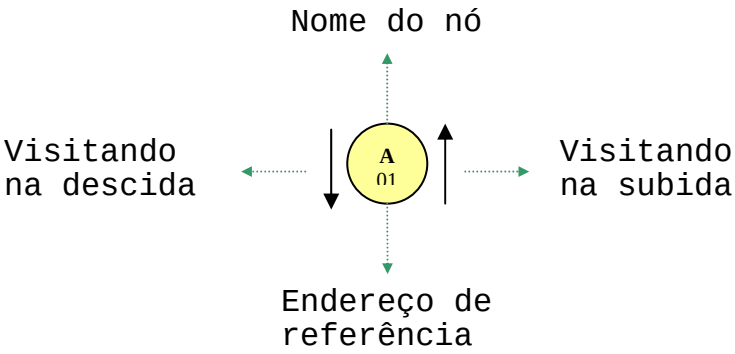


FIG. 5.4. Representação de um nó da AST.

Para ilustrar o funcionamento do algoritmo, as FIG. 5.6 até FIG. 5.17 mostram o que acontece a cada visita na AST. Parra facilitar o entendimento, nesse exemplo serão utilizados representações didáticas da AST e do modelo do padrão (FIG. 5.4, FIG. 5.5 respectivamente).

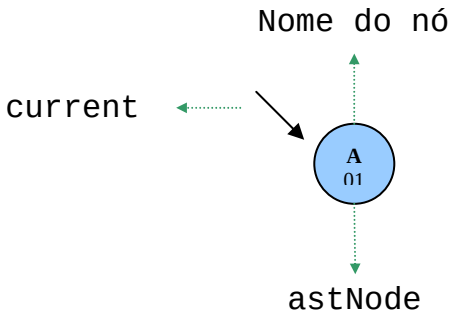


FIG. 5.5. Representação didática de um nó do padrão.

Inicialmente temos somente a AST e um único padrão na coleção:

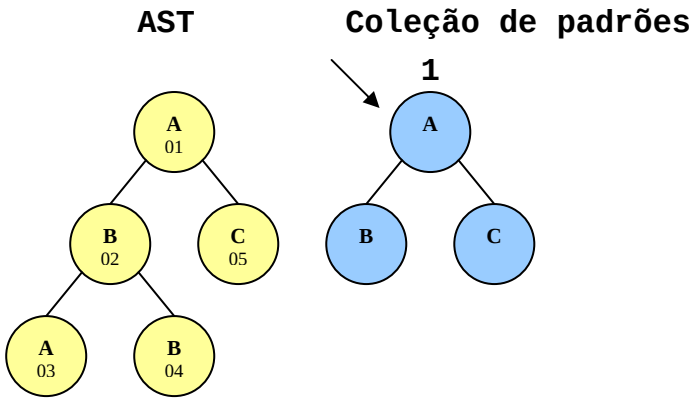


FIG. 5.6. Estado inicial da detecção.

Visitando na descida o nó A (01). O `current` casa com o nó, faz-se uma cópia desse padrão (padrão 2) e sua atualização (`current`):

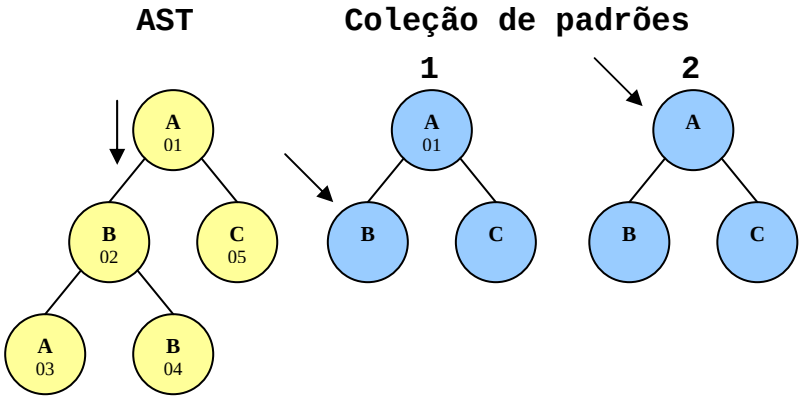


FIG. 5.7. Visita na descida do nó A (01).

Visitando na descida o nó B (02). O `current` do padrão 1 casa com o nó, então, faz-se uma cópia desse padrão (padrão 3). A sua atualização não é feita, pois esse elemento não tem filhos.

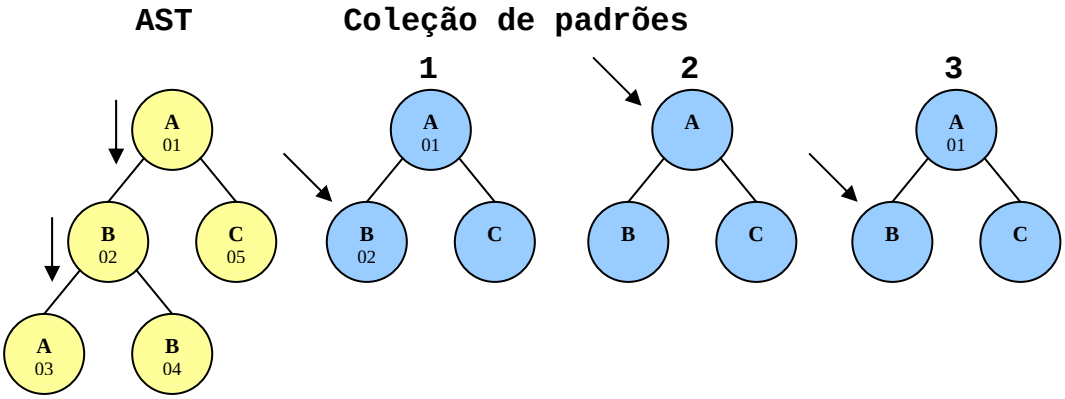


FIG. 5.8. Visita na descida do nó B (02).

Visitando na descida o nó A (03). O `current` do padrão 2 casa com o nó. Faz-se uma cópia desse padrão (padrão 4) e sua atualização:

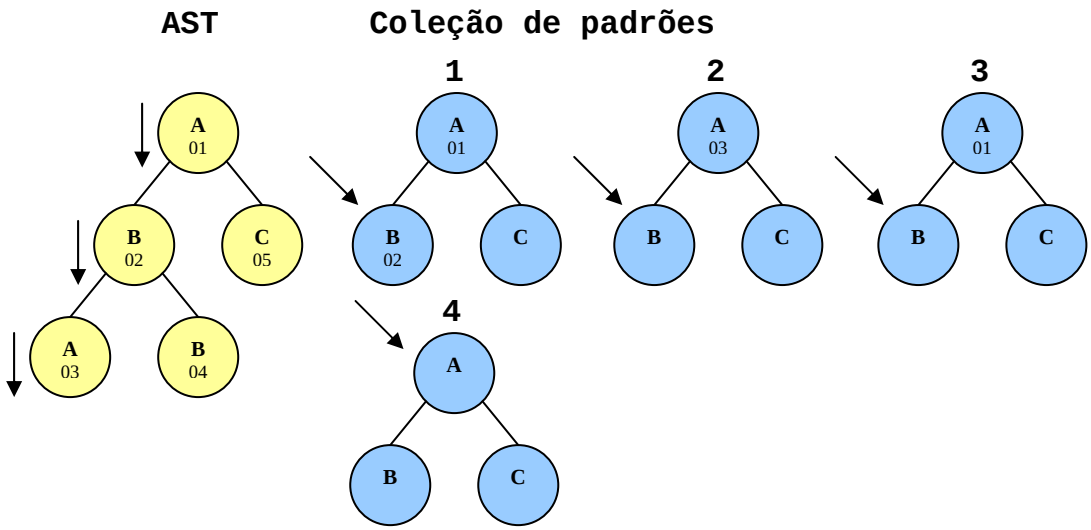


FIG. 5.9. Visita na descida do nó A (03).

Visitando na subida o nó A (03). A verificação de padrões encontrados não detecta nenhum padrão. A verificação de padrões que não podem mais ser encontrados detecta o padrão 2, que é removido da coleção de padrões.

.

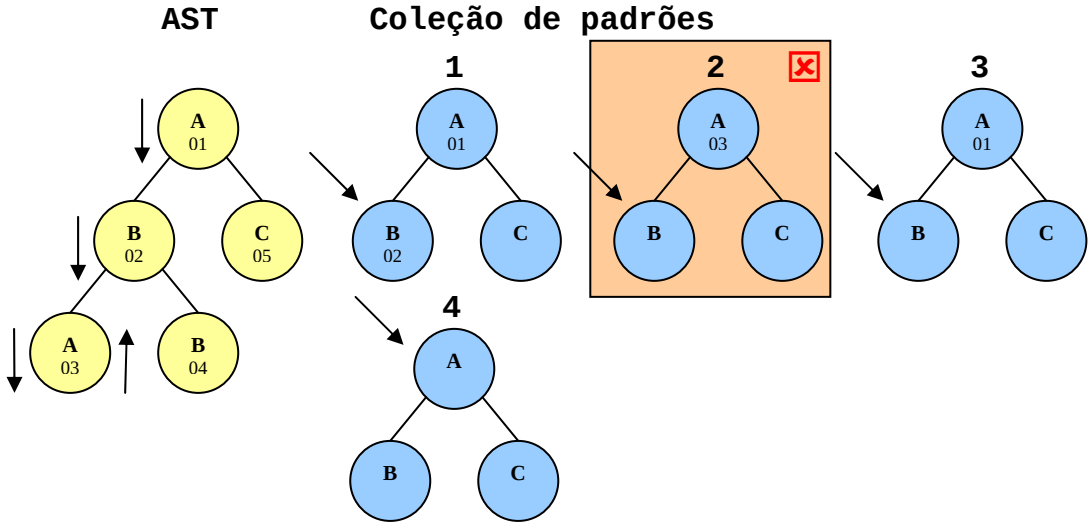


FIG. 5.10. Visita na subida do nó A (03).

Visitando na descida o nó B (04). O `current` do padrão 3 casa com o nó, então, faz-se uma cópia desse padrão (padrão 5). A sua atualização não é feita, pois o elemento casado não possui filho.

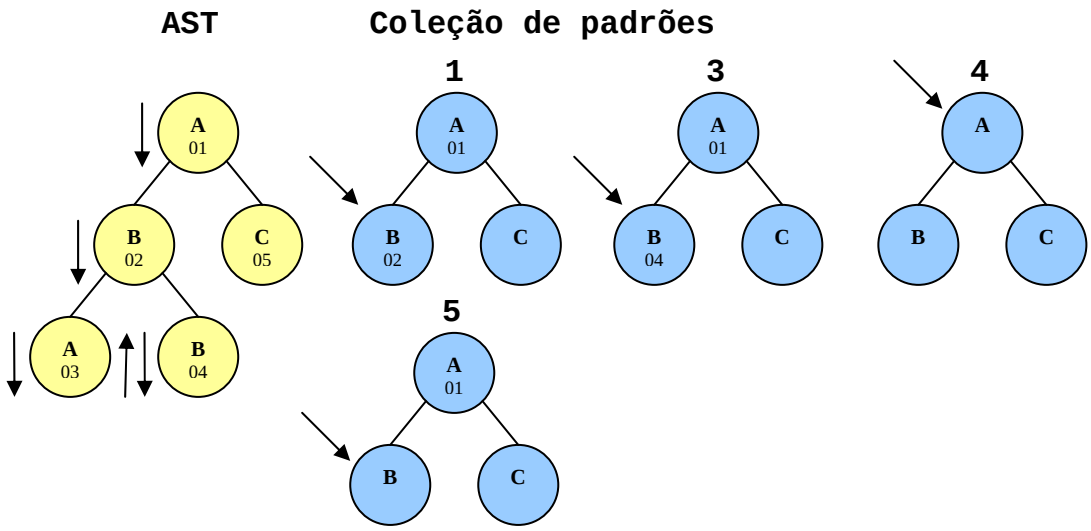


FIG. 5.11. Visita na descida do nó B (04).

Visitando na subida o nó B (04). A verificação de padrões encontrados não detecta nenhum padrão. A verificação de padrões que não podem mais ser encontrados não detecta nenhum padrão. É feita a atualização do `current` do padrão 3:

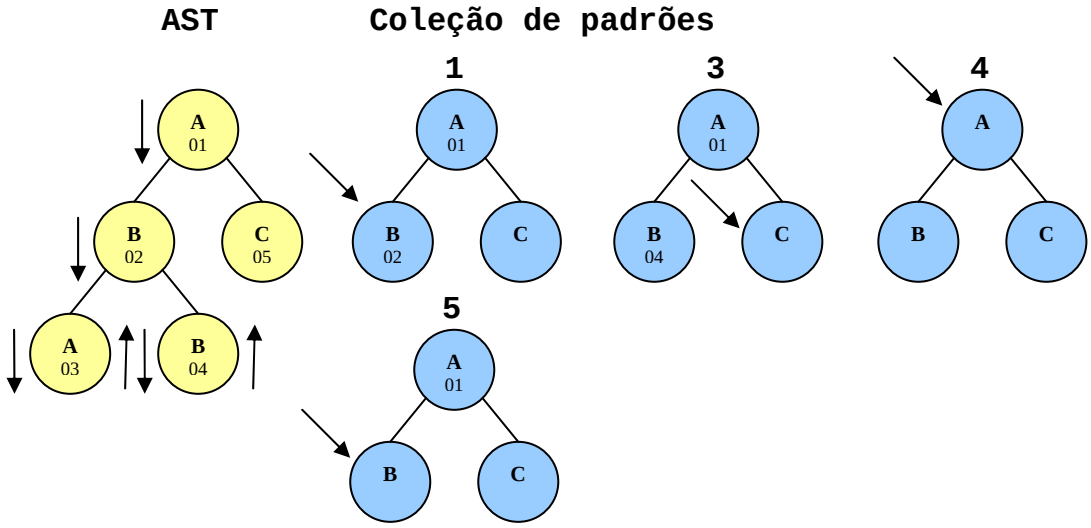


FIG. 5.12. Visita na subida do nó B (04).

Visitando na subida o nó B (02). A verificação de padrões encontrados não detecta nenhum padrão. A verificação de padrões que não podem mais ser encontrados não detecta nenhum padrão. É feita a atualização do `current` do padrão 1:

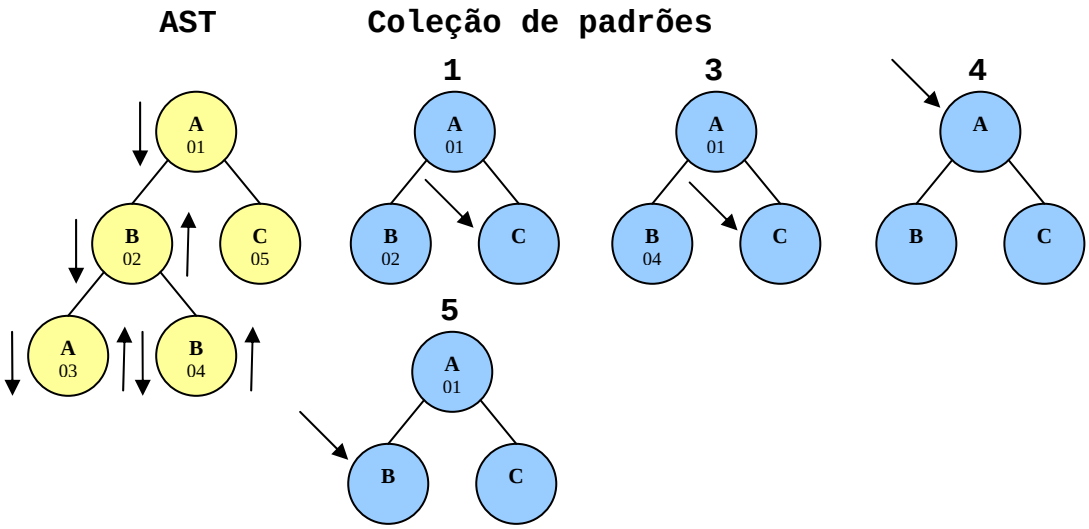


FIG. 5.13. Visita na subida do nó B (02).

Visitando na descida o nó C (05). O `current` do padrão 1 e 3 casam com o nó, então, faz-se uma cópia desses padrões (padrão 5 e 6 respectivamente). A sua atualização não é feita, pois os elementos casados não possuem filho.

:

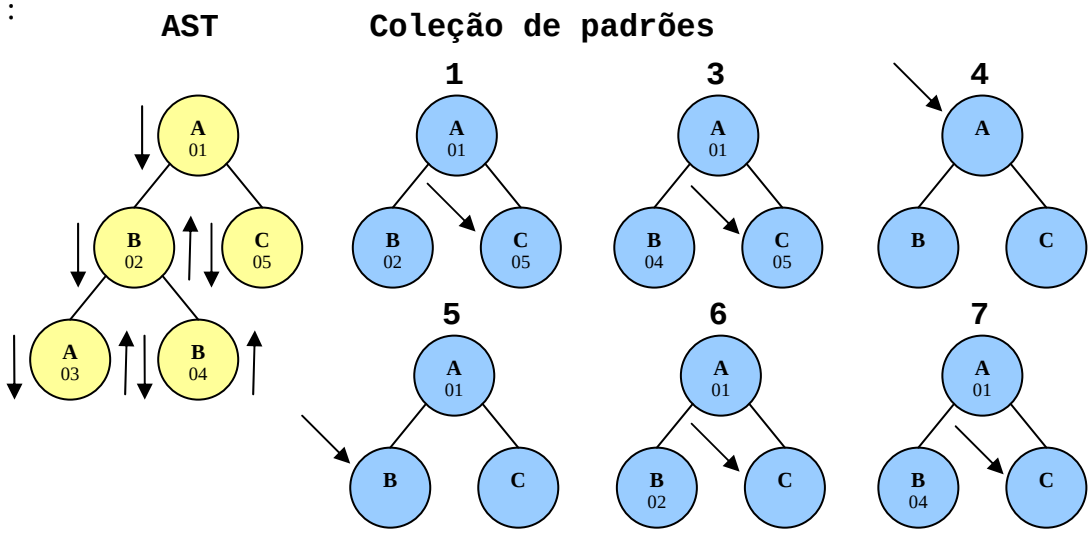


FIG. 5.14. Visita na descida do nó C (05).

Visitando na subida o nó C (05). A verificação detecta dois padrões encontrados (1 e 3). Esses padrões são armazenados fora da estrutura e retirados da coleção. A verificação de padrões que não podem mais ser encontrados não detecta nenhum padrão.

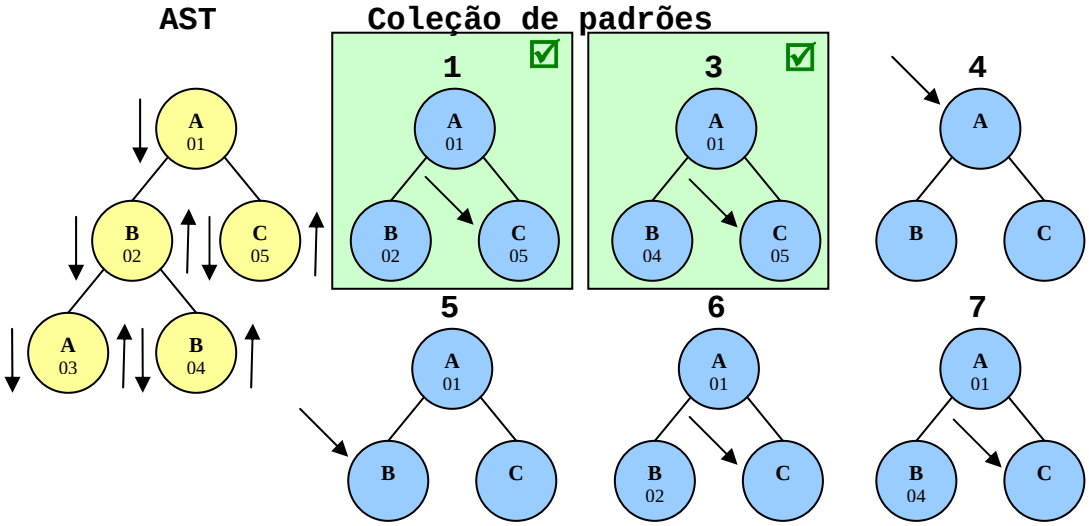


FIG. 5.15. Visita na subida do nó C (05).

Visitando na subida o nó A (01). A verificação de padrões encontrados não detecta nenhum padrão. A verificação de padrões que não podem mais ser encontrados detecta três padrões. Estes são retirados da coleção de padrões.

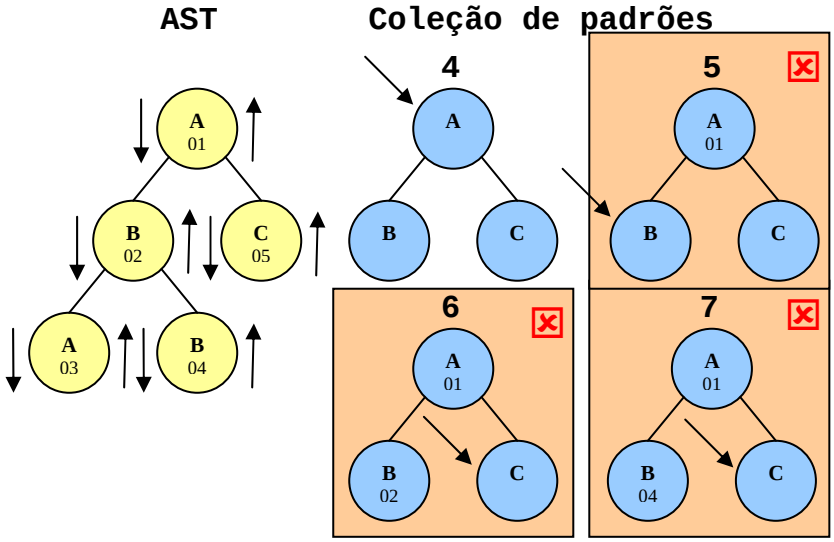


FIG. 5.16. Visita na subida do nó A (01).

A visitação termina restando na coleção de padrões somente os padrões originais, ou seja, os padrões que inicialmente foram selecionados para detecção. Para esse exemplo, restou na

coleção apenas único padrão (padrão 4). A FIG.5.17 mostra as duas ocorrências desse padrão na AST.

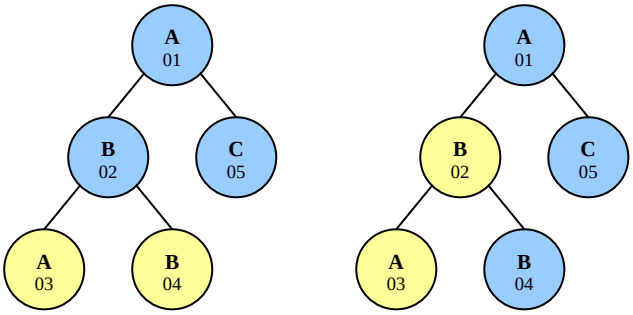


FIG. 5.17. Ocorrências encontradas.

A FIG. 5.18 mostra um exemplo de código Java (à esquerda) e um exemplo de padrão escrito em PDL (à direita). Esse padrão representa um método que possui um comando `if` com comando `return` dentro.

```
/* Classe de Exemplo */

// TypeDeclaration
public class ClasseTest {


  // MethodDeclaration
  public void metodo() {
      if (true) {
         if (true)
              return;
      }
}
```

```
<!—- Exemplo PDL -->

<MethodDeclaration>
   <IfStatement _Mark="if com return">
         <ReturnStatement/>
     </IfStatement>
</MethodDeclaration>
```

FIG. 5.18. Exemplo de Código Java e padrão PDL.

Ao executar o algoritmo de detecção para o padrão e código Java da figura acima, serão encontradas duas instâncias desse padrão. Isso acontece porque o algoritmo casa o nó `IfStatement` do padrão com as duas ocorrências do `IfStatement` na AST, conforme ilustrado na FIG.5 .19.

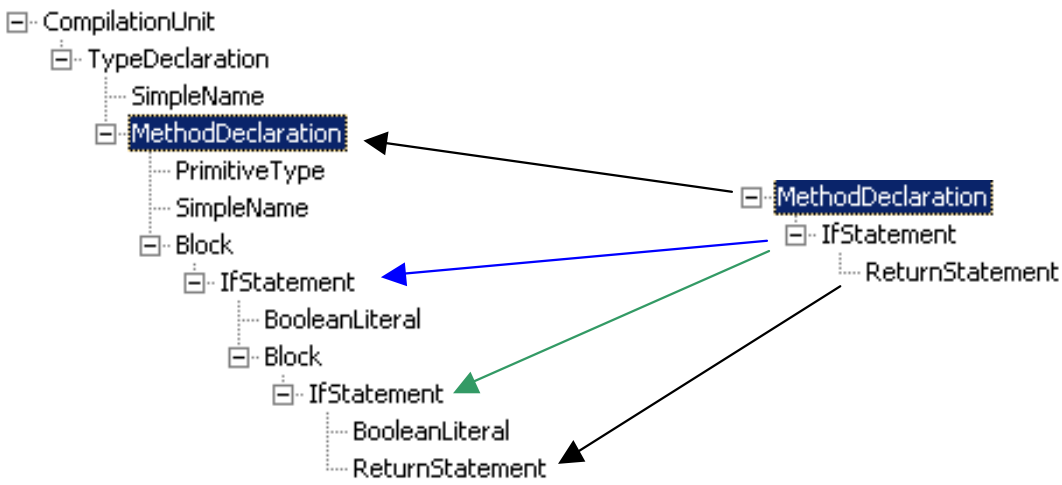


FIG. 5.19. Ocorrências encontradas para o `IfStatement`.

5.4. ALGORITMO COM ALTERNATIVO “OR”

O alternativo “Or” representado pelo elemento especial `<Or>` da linguagem PDL tem como objetivo realizar a função “ou” entre dois ou mais elementos em um padrão. Cada filho imediato de um nó “Or” é uma alternativa para o casamento. Didaticamente podemos visualizar um padrão com alternativo como sendo vários padrões simples onde cada filho imediato de um elemento “Or” forma um novo padrão. A FIG. 5.20 mostra um exemplo que em que o padrão mais a esquerda pode ser “visualizado” como dois outros padrões.

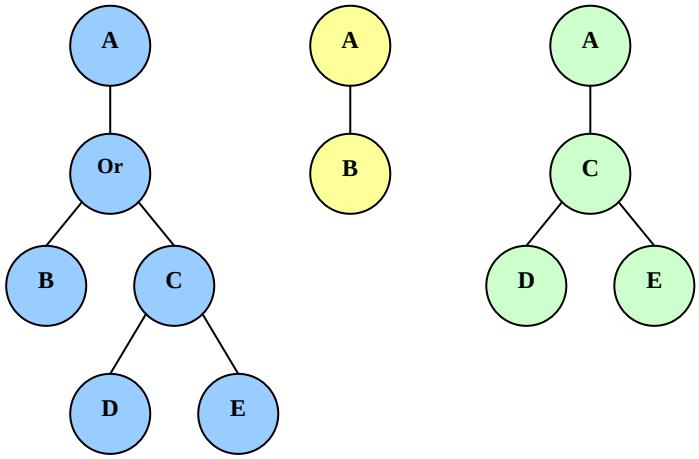


FIG. 5.20. Padrão com “Or” e seus correspondentes.

Para fazer o casamento de um padrão com alternativo são necessárias várias outras checagens no algoritmo descrito na sessão 5.3. Toda visita na descida em que há um casamento, testa se o próximo elemento é um filho do casado e é um `<Or>`. Em caso afirmativo, gera-se uma cópia de todo o padrão para cada filho setando o `current` como cada filho, respectivamente. Após as cópias geradas, retira-se o padrão original da coleção deixando somente as cópias. Para cada visita na subida, se o `current` está casado e o próximo elemento é um `<Or>`, gera-se também as cópias de mesma maneira que na visita na descida. Uma outra checagem que é feita na subida é se está subindo pelo elemento pai do `<Or>`. Neste caso, retira-se esse padrão da coleção, pois ele não pode mais ser encontrado.

5.5. CASAMENTO DE ATRIBUTOS

O casamento dos atributos é feito durante o casamento de um nó, ou seja, para que haja o casamento de um nó se faz necessário o casamento dos atributos. Os atributos “simples”, os que não iniciam com o caracter sublinhado (‘_’) são casados como expressões regulares. Cada um desses atributos possui um correspondente na AST.

Os casamentos de cada atributo especiais são feitos separadamente através de uma chamada de função para cada atributo especial. Para o casamento `_InParentSubTree`, obtém-se o nó da AST correspondente a sub-árvore do elemento “pai” onde o elemento a ser casado deve estar. Após isso, é feita a comparação desse nó da AST com o atributo `astNode` dos ancestrais (pai, avô, bisavô, etc) até o nó raiz ou até que esteja fora do limite definido pelo atributo `_MaxSubTreeLevel` (quando existente). Se a comparação for bem sucedida para algum ancestral, o elemento está na sub-árvore correta e o casamento é feito.

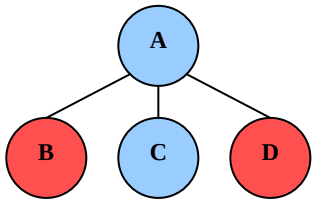
O atributo especial `_Mark` é o único que não tem influência no casamento. A chamada de função que realiza o “casamento” desse atributo retorna sempre `true` (casamento bem sucedido) e apenas cria um marcador a ser inserido na plataforma Eclipse (um marcador na *Problems View*).

O atributo `_Not` tem a finalidade de representar padrões em PDL em que um determinado elemento não deva existir na AST para que haja o casamento. A idéia do casamento de um padrão com `_Not` é transformá-lo em vários padrões onde um desses padrões (padrão

principal) é o padrão que se deseja encontrar e os outros (padrões dependentes) são padrões que não podem ser encontrados para que o casamento do padrão principal tenha sucesso. Por exemplo, a FIG. 5.21 (a) mostra o código de um padrão com dois nós que tem o atributo `_Not` que está representado graficamente pela FIG. 5.21 (b), onde os nós com `_Not` estão de cor diferente dos demais nós do padrão.

```
<A>
    <B _Not="true" />
    <C />
    <D _Not="true" />
</A>
```

(a)



(b)

FIG. 5.21. Padrão com atributo especial `_Not`.

Esse padrão se transforma em três outros padrões (mostrados na FIG. 5.22), um padrão principal que deve ser encontrado para que haja o casamento e os padrões dependentes que não podem ser encontrados. Uma observação importante é que caso seja encontrado algum padrão dependente, este só anula o padrão principal caso tenham os mesmos casamentos anteriores à transformação do padrão realizada ao encontrar algum elemento com atributo `_Not` em questão. Essa "dependência" é representada no modelo de padrões pela classe `ForeignNode`. Essa classe mantém a relação em os nós (e o padrões) do padrão principal e seus padrões dependentes. Através dessa relação é possível detectar um real casamento de um padrão principal e detectar quando esse padrão ou seus dependentes não podem ser mais encontrados para então removê-los da coleção de padrões a serem detectados.

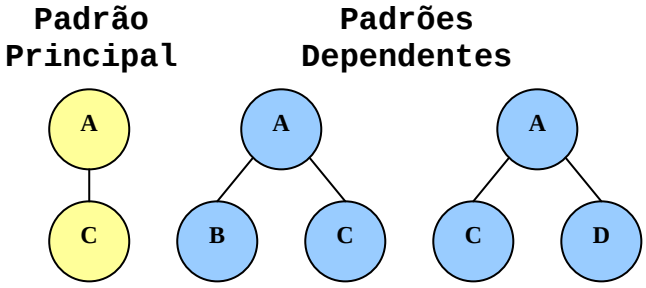


FIG. 5.22. Resultado da transformação.

6. ESTUDO DE CASO

Nesse capítulo será apresentado um estudo de caso no qual a ferramenta em questão é aplicada em um sistema real. Esse estudo tem o objetivo de avaliar a aplicabilidade da ferramenta para um caso real.

O sistema utilizado nesse estudo é um sistema desenvolvido para WEB, escrito em Java, com 67 classes totalizando 14484 linhas de código[1] (não foram computadas as linhas de código JSP, HTML, XML, PL-SQL). Também será apresentado neste capítulo alguns dos padrões utilizados e os resultados obtidos.

6.1.ABORDAGEM PARA O ESTUDO

Os mecanismos de definição de padrão desenvolvidos são flexíveis o bastante para a codificação de padrões visando diversos benefícios mencionados na seção 1.1 do capítulo 1. No entanto seria muita ambição para um primeiro estudo de caso definir padrões de código que agregassem diversos objetivos. Portanto, optou-se por definir padrões focados no benefício 6 (Facilitar a adoção de padrões de codificação).

Com este intuito, foram codificados trinta e nove padrões, cada um deles representa a não conformidade com algum item da recomendação de codificação disponibilizada pela *HotWork Solution* [HOTWORK, 2004]. A recomendação de codificação da *HotWork* é adotada como parte do padrão de codificação da empresa que desenvolveu o projeto utilizado neste estudo de caso.

6.2. PADRÕES DE CODIFICAÇÃO

Padrões de codificação são regras que ajudam a desenvolver e melhorar o estilo e estrutura de programas. Essas regras focam em erros comums cometidos pelos

[1] Não será apresentada nenhuma parte do código do sistema conforme acordo com a empresa que cedeu o sistema para o estudo.

desenvolvedores que afetam a qualidade do código. É importante lembrar que a qualidade de um software está diretamente ligada ao custo de manuntenção, performace e portabilidade, onde através do uso de padrões de codificação pode-se previnir e/ou detectar problemas reduzindo o tempo gasto em manutenção, depuração, otimização, etc. Entre os benefícios diretos proporcionados pelos padrões de codificação, destacam-se:

- Aumento de qualidade do código fonte.
- Aumento da legibilidade do código.
- Permite mudanças com o mínimo de impacto no código.
- Melhoria do conhecimento técnico do programador.
- Aumento de Produtividade.

6.3.PADRÕES UTILIZADOS

Para esse estudo foram implementados em PDL trinta e nove padrões de codificação da *HotWork Solution* [HOTWORK, 2004]. Conforme mencionado anteriormente, esses padrões foram adotados como parte dos padrões de codificação da empresa que desenvolveu o projeto utilizado. Esses padrões foram agrupados e numerados de acordo com suas abordagens. A seguir são mostrados os padrões e suas respectivas descrições. No apêndice, estão os códigos PDL para cada um desses padrões.

Número de classes por arquivo – Padrão 7.2.2

Em geral, deve ser somente uma classe por arquivo. Isso permite que o arquivo fonte combine com o nome da classe, portanto seja mais fácil de localizar o arquivo fonte. Dessa forma, classes relacionadas devem ser colocadas no mesmo package, mas não no mesmo arquivo.

Conseqüência: Manutenibilidade.

Especificar os tipos de import – Padrão 7.2.3

Importe classes e não pacotes. Isso reduz a chance de erro quando duas classes têm o mesmo nome, e utiliza-se a classe errada.

Conseqüência: Legibilidade, Manutenibilidade.

Tamanho dos métodos - Padrão 7.2.4

Os métodos devem ser o menor possível, para possibilitar legibilidade e manutenibilidade. Para isso, os métodos públicos devem ter no máximo 2 níveis de aninhamento de blocos. Os demais, no máximo 3.

Conseqüência: Legibilidade, Manutenibilidade.

Não declare múltiplas variáveis – Padrão 7.2.5

Não declare múltiplas variáveis em uma única declaração. Declaração de múltiplas variáveis em uma única declaração é confuso.

Conseqüência: Legibilidade.

Evitar assinatura de métodos com muitos parâmetros – Padrão 7.2.6

Uma grande quantidade de parâmetros indica complexidade para chamar objetos e deve ser evitado. Propomos uma limitação de 5 (cinco) parâmetros recebidos por um método.

Conseqüência: Eficiência, Manutenibilidade.

Todos os *switch statements* devem conter um *default case* – Padrão 8.2.2

O uso do *default* no *case* faz o fluxo de controle explícito quando não existe uma combinação para o *label* em questão. Isso torna mais fácil para os desenvolvedores determinarem quando um *label* usado no case deve ser adicionado ou removido inadvertidamente.

Conseqüência: Legibilidade, Manutenibilidade.

Coloque constantes no lado esquerdo das comparações – Padrão 8.2.8

Uma digitação comum enquanto escrevendo código é usar o operador "=" ao invés de "==" em operações de igualdade. Colocando constantes no lado esquerdo da comparação irá fazer com que o compilador de uma mensagem de erro.

Conseqüência: Manutenibilidade.

Declare *loops for* com uma condição e um incremento – Padrão 8.2.9

Um *loop for* deve ter uma condição e um incremento para facilitar a legibilidade do código.

Conseqüência: Legibilidade.

Evitar atribuições dentro de uma condição *if* – Padrão 8.2.10

Deve-se evitar um *statement if* com atribuição na sua expressão de decisão. Isso torna o código menos legível.

Conseqüência: Legibilidade.

Evitar *statements for* com corpo vazio – Padrão 8.2.11

Statements for que são seguidos imediatamente por ponto e vírgula ou com seu bloco vazio não são uma boa pratica de programação.

Conseqüência: Manutenibilidade.

Atribuição a variáveis de controle dentro do corpo de um *for* – Padrão 8.2.12

Uma variável de controle de um *for* deve somente ser modificada na inicialização e expressão de controle do *statement*. Modificando-a no corpo do *loop* torna o código difícil de entender e pode contribuir para erros.

Conseqüência: Legibilidade, Manutenibilidade.

Evitar *statements if* com corpo vazio – Padrão 8.2.13

Um *statement if* estiver seguido imediatamente por um ponto e vírgula, ou com seu bloco vazio, não é uma boa prática de programação.

Conseqüência: Manutenibilidade.

Constantes devem ter nomes somente com maiúsculas – Padrão 8.3.1.1

Colocar os identificadores de constantes com letras maiúsculas ajuda na legibilidade do código.

Conseqüência: Legibilidade.

Variáveis devem ter o nome iniciado com uma letra minúscula – Padrão 8.3.1.2

Colocar os identificadores de variáveis e atributos da classe iniciando com letra minúscula ajuda na legibilidade do código.

Conseqüência: Legibilidade.

Pacotes devem ter o nome completamente minúsculo – Padrão 8.3.1.3

Os pacotes devem ter o nome completamente formado por letras minúsculas e sem acentuação.

Conseqüência: Legibilidade.

Variáveis de *loop for* devem ser identificadas como i, j, k – Padrão 8.3.3

Usando os nomes i, j, k para variáveis de controle de um *for* (quando do tipo inteiro) é uma convenção que é resultado de tradição. O uso desses nomes deve ser uniforme por todos os códigos do projeto, tornando assim fácil e rápido a identificação de variáveis de controle de um *for*.

Conseqüência: Legibilidade, Manutenibilidade.

Classes e Interfaces devem iniciar com letra Maiúscula – Padrão 8.4.1

Os nomes de classes e interfaces devem iniciar por uma letra maiúscula. Isso ajuda na legibilidade do código.

Conseqüência: Legibilidade.

Use 'L' ao invés de 'l' para expressar constantes do tipo *long* – Padrão 8.5.2

Constantes inteiras são *long* se você colocar no fim da declaração o "L" ou "l". É preferível usar o "L" ao invés de "l", pois o "l" pode ser confundido com o numero um.

Conseqüência: Legibilidade.

Todos os nomes de métodos devem iniciar com uma letra minúscula – Padrão 8.6.1

Comece o nome do método com letra minúscula e coloque letra maiúscula para todas as palavras subseqüentes. Isso torna o código mais legível.

Conseqüência: Legibilidade.

A referência *super* é equivalente ao uso de *casting* com *this* – Padrão 9.6

Dada a superclasse B e a subclasse D, uso de *super* em D é o mesmo que a expressão `((B)this)`, mas deve usar-se o *super*.

Conseqüência: Legibilidade, Manutenibilidade.

Evite o uso de parâmetros que conflite com atributos da classe – Padrão 9.7

O uso de parâmetros em métodos com o mesmo nome de atributos da classe causa confusão e pode provocar erros lógicos.

Conseqüência: Legibilidade, Manutenibilidade.

Não inicialize atributos estáticos dentro do construtor – Padrão 10.1.2

Atributos estáticos podem somente ser inicializados uma única vez na classe. Isto é feito quando a classe é carregada. Atribua um valor inicial para a declaração ou inclua um bloco estático caso seja necessário algum processo para obter o valor inicial.

Conseqüência: Eficiência.

Evite construtores públicos com mais de 2 níveis de aninhamento – Padrão 10.1.3.1

Construtores são chamados freqüentemente. Portanto eles devem ser eficientes. Use construtores para inicialização de construtores. Funções extensivas devem ser deixadas para ações realizadas por outros métodos que são explicitamente chamados após a construção.

Conseqüência: Legibilidade, Eficiência.

Classe que apenas empacota um grupo de métodos e atributos estáticos deve ter construtor privado – Padrão 10.1.3.2

Classe que não possui atributos não-estáticos não precisam ser instanciadas, portanto, deve ter seu construtor privado.

Conseqüência: Manutenibilidade, Eficiência.

Maximize o uso de encapsulamento de atributos – Padrão 11.2

Crie todos os atributos como privados e faça métodos de acesso para eles. Isso permite uma maior flexibilidade para mudanças de implementação.

Conseqüência: Legibilidade, Manutenibilidade.

Evite mais de dois níveis de aninhamento de classes – Padrão 11.6

Aninhamento em mais de dois níveis de classes podem ser difícil de compreenção, portando, isso deve ser evitado.

Conseqüência: Legibilidade, Manutenibilidade.

Não retorne *null* quando o tipo de retorno é um *array* – Padrão 13.2.1

Não retorne *null* para as referências de *array*. É preferível retornar sempre um *array* com tamanho zero. Isso evita *NullPointerException* em vários casos.

Conseqüência: Legibilidade, Manutenibilidade.

Evite blocos *catch* com corpo vazio – Padrão 14.2

Blocos *catch* com o corpo vazio são perigosos, pois podem estar escondendo alguma exceção.

Conseqüência: Manutenibilidade.

Evite concatenação de *String* com operador '+=' – Padrão 15.6

Concatenação de *String* podem refletir em impactos na performance. *Strings* são objetos imutáveis, dessa forma, a concatenação resulta na criação de objetos temporários. Uma solução para isso é utilizar o *java.lang.StringBuffer*.

Conseqüência: Eficiência.

Evite o uso excessivo do operador de negação – Padrão 15.7

O operador de negação prejudica a legibilidade do código. Evite o uso no máximo três vezes esse operador dentro de um método.

Conseqüência: Legibilidade.

Evite usar *String.equals('literal')*, use *'literal'.equals(String)* – Padrão 15.12

Deve-se evitar uso de um objeto String chamando seu método *equals*. Isso porque essa chamada pode disparar uma exceção de *NullPointerException* para o caso dessa referência ser nula (*null).*

Conseqüência: Manutenibilidade.

Evite criar objetos String chamando *new String()* para literais – Padrão 16.1

Copiar um literal para dentro de um objeto *String* "gasta" tempo e é redundante e deve ser evitado.

Conseqüência: Eficiência.

Evite o uso de *Date[]*, use *long[]* – Padrão 16.3

O objeto Date contém muitos campos, e com isso utiliza muito espaço. Caso seja necessário utilizar um *array* de objetos Date, substitua pelo uso de um *array* de *long*, onde somente serão armazenados os valores *long* que representam as datas.

Conseqüência: Eficiência.

Evitar *static collections* – Padrão 16.4

Objetos que representem coleções estáticas (*Vector*, *Hashtable*, etc) podem armazenar vários outros objetos, fazendo com que possa ocorrer *memory leaks*. Quando são colocados vários objetos em uma coleção estática, esses objetos serão referenciados por essa coleção por toda a "vida" do programa.

Conseqüência: Manutenibilidade.

Evite chamar métodos dentro da condição de um *loop* – Padrão 16.6

A menos que o compilador otimize-o, a condição do *loop* será calculada para cada iteração sobre o *loop*. Se o valor de uma condição não sofre mudança, ele será executado mais rápido do que a chamada de um método fora do *loop*.

Conseqüência: Eficiência.

Uso do operador condicional '*if (cond) return; else return;*' – Padrão 16.7, 16.8

O operador condicional para uma simples expressão que executa um ou outro *statement*, baseado em uma expressão pode ser substituído por uma forma compacta `((cond) ? return; : return;)`.

Conseqüência: Legibilidade.

Evite chamadas de métodos *synchronizd*' em um *loop* - Padrão 16.10

Métodos *synchronized* são “custosos”, a invocação desses métodos dentro de um *loop* não é recomendada e deve ser evitada.

Conseqüência: Eficiência.

Coloque blocos *try/catch* fora dos *loops* – Padrão 16.11

Evite o uso de blocos *try/catch* dentro de *loops*. Isso pode representar perda de performance.

Conseqüência: Eficiência.

6.4.RESULTADOS

Após a realização da aplicação da ferramenta de inspeção na identificação das não conformidades com os padrões implementados, obtivemos o resultado apresentado na TAB. 6.1. Os resultados obtidos nos permitem constatar que os padrões atingiram plenamente o objetivo a que se propunham, a saber, identificar não conformidades com o padrão de codificação. Além disso, apesar do foco neste objetivo particular, os padrões foram capazes de localizar questões sensíveis no código fonte, como a existência de blocos *catch* vazios, que provocam a existência de exceções silenciosas, ou ainda a presença excessiva de chamada a métodos sincronizados dentro de *loops*, o que causa degradação na performance. Apesar de ser uma prática recomendada, o padrão 8.2.8 não fora incorporada na cultura da equipe desenvolvedora, daí a existência de 144 instâncias desse padrão.

TAB. 6.1. Resumo dos padrões encontrados.

Padrões Encontrados	Conseqüência	Total de ocorrências	Número de classes com o padrão
Padrão 16.10	E	26	4
Padrão 15.6	E	5	1
Padrão 10.1.3.1	L, E	3	1

Padrão 14.2	M	1	1
Padrão 16.11	E	2	2
Padrão 8.2.10	L	1	1
Padrão 16.6	E	15	9
Padrão 7.2.4	L, M	23	5
Padrão 8.2.8	M	144	24
Padrão 8.2.2	L, M	4	2
Padrão 7.2.6	E, M	3	3
Padrão 8.4.1	L	24	24
Padrão 8.6.1	L	4	1
Padrão 7.2.5	L	5	2
Padrão 8.2.13	M	2	2
Padrão 8.2.9	L	1	1
Padrão 15.12	M	4	1

Conseqüência: L – Legibilidade

M – Manutenibilidade

E – Eficiência

7. CONCLUSÕES

Conforme mostrado no início dessa dissertação, o crescimento do tamanho e complexidade dos programas juntamente com a dificuldade de realizar um trabalho de inspeção motivou o surgimento de várias ferramentas que automatizam parte do processo de inspeção. Durante o estudo destas ferramentas foram constatadas deficiências, como a dificuldade em se definir padrões e a não integração com um ambiente de desenvolvimento. Essas deficiências prejudicam estas ferramentas tanto em sua usabilidade, por forçar o usuário a trocar seu ambiente de desenvolvimento para realizar uma inspeção, como dificultando a definição e inclusão de padrões específicos para uma organização.

Baseado nisso, apresentamos uma ferramenta de inspeção de código que permite:

- Especificação declarativa de novos padrões.
- Facilidade de inclusão dos novos padrões à ferramenta.
- Integração da ferramenta com um IDE.

No trablho foi especificada uma linguagem para representação de padrões de código (PDL - Pattern Description Language). Essa linguagem possui sintaxe XML, permitindo que padrões sejam declarados por meio de elementos XML que representam nós da AST do código-fonte. A ferramenta desenvolvida para inspeção de código detecta padrões escritos na linguagem PDL através de um algoritmo para o casamento de padrões. O algoritmo criado para fazer o casamento utiliza uma estratégia de clonagem de padrões a fim de realizar o casamento de todas as instâncias dos n padrões selecionados para detecção, percorrendo a AST do programa uma única vez.

Nessa ferramenta, a inclusão de novos padrões torna-se mais simples que nas demais ferramentas em questão. Isso porque não é necessário programar na linguagem Java, ter conhecimento em desenvolvimento de *plug-ins*, além de não precisar que os novos padrões sejam compilados. Sendo suficiente inclusão dos mesmos diretamente na ferramenta.

Outro diferencial da ferramenta desenvolvida foi a integração com o IDE Eclipse, permitindo que a inspeção seja realizada no mesmo ambiente de trabalho em que ocorre o desenvolvimento. Dessa maneira, não é necessário que o desenvolvedor mude seu ambiente de desenvolvimento para realizar uma inspeção, facilitando e motivando a execução da mesma.

Assim, mostramos uma ferramenta de inspeção de código que possui um método simples e poderoso de representação de padrões sintáticos, o que a torna mais flexível do que as ferramentas atuais.

O estudo de caso confirmou a grande facilidade de desenvolvimento de padrões, bem como comprovou a utilidade da ferramenta para inspeção de código Java. Além disso, ficou claro durante o estudo de caso que a integração da ferramenta com o ambiente Eclipse a tornou uma ferramenta de alta produtividade e de fácil uso. Também é bom dizer que esse estudo superou as expectativas, pois mesmo na primeira aplicação prática da ferramenta, ficou claro o impacto sobre a qualidade de um sistema real.

7.1. TRABALHOS FUTUROS

Entre os trabalhos já em desenvolvimento no momento estão a elaboração de um experimento para medir o ganho de produtividade e de qualidade com o uso da ferramenta e a extensão da ferramenta para outras linguagens (ainda no ambiente Eclipse).

Além disso, existem ainda algumas melhorias a serem feitas na ferramenta para que esta se torne um produto, permitindo que esta possa ser adotada por empresas de desenvolvimento. Entre essas melhorias estão:

- *Wizard* para geração de novos padrões.
- Aplicar a inspeção sob vários arquivos simultaneamente.
- Permitir a definição que padrões que envolvam mais um arquivo de código-fonte.
- Melhorar o tratamento de erros para padrões especificados erroneamente.
- Classificação dos padrões em níveis de prioridades, possibilitando que se faça a inspeção somente para determinados níveis pré-selecionados.
- Criar uma tela de configuração para o *plug-in.*
- Criar uma versão para distribuição.

Em um futuro próximo planeja-se estender a PDL com condições oriundas de análise de fluxo de controle e análise de fluxo de dados, o que permitirá a especificação de padrões mais expressivos.

8. REFERÊNCIAS BIBLIOGRÁFICAS

AHO, ALFRED, SETHI, RAVI, ULLMAN, JEFFREY. **Compilers Principles, Techniques and Tools**, Addison-Wesley Publishing Company Março de 1998.

BCEL. **The Byte Code Engineering Library**, http://jakarta.apache.org/bcel/, 2004. In Proceedings of the ACM SIGSOFT '94 Symposium

BOKOWSKI, BORIS. **Coffeestrainer: Statically-checked constraints on the definition and use of types in Java**. *In Proceedings of ESEC/FSE'99*. Springer-Verlag, Setembro de 1999.

CASTOR, F. E BORBA, P. **A Language For Specifying Java transformations**, V Simpósio Brasileiro de Linguagens de Programação, Curitiba, Brazil, p.236-251, 2001.

CHECKSTYLE. **A Development Tool to Help Programmers Write Java Code**, disponível em: http://checkstyle.sourceforge.net. Último acesso: abril de 2005.

CODEPRO. **CodePro Advisor on-line documentation**. Disponível em: http://www.instantiations.com/codepro/ws/docs. Último acesso: fevereiro de 2005.

ECLIPSE 2003. **Eclipse Platform Technical Overview**, Object Technology International, fevereiro de 2003. Disponível em: http://www.eclipse.org/whitepapers/eclipse-overview.pdf. Último acesso: fevereiro de 2005.

ECLIPSE. **Site oficial do projeto Eclipse**. Disponivel em: http://www.eclipse.org. Último acesso: maio de 2005

EMDEN, EVA VAN E MONEEN, LEON. **Java Quality Assurance by Detecting Code Smells.** *In Procceedings of the 9th Working Conference on Reverse Engineering*, Outubro de 2002.

FAGAN, Michael E. **Advances in Software Inspection**, IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, Julho de 1986, pp.744-751.

FAGAN, Michael E. **Design and Code Inspections to Reduce Errors in Program Development**, IBM System Journal, Vol. 15, No. 3, 1976, pp.182-211.

FINDBUGS. **FindBugs** Manual, Disponível em: http://findbugs.sourceforge.net/manual/index.html. Último acesso: fevereiro de 2005.

GAMMA, R. HELM, JOHNSON, R., VLISSIDES, J. **Design Patterns - Elementsof Reusable Object-Oriented Software**. Addison-Wesley, 1994.

GARCIA, ALEX. **Um Modelo Categórico para Tradução entre Linguagens de Programação**. Tese de Doutorado, PUC-Rio, 2000.

HAMMURAPI. **Hammurapi Documentation**, Disponível em: http://www.hammurapi.org/content/Hammurapi.1.html. Último acesso em: fevereiro de 2005.

HOTWORK. **Java Programming Guidelines**. Fevereiro de 2004. Disponível em: http://hotwork.sourceforge.net/hotwork/manual/guidelines/code-guidelines-user-guide.html. Último acesso em: Fevereiro de 2005.

JAVA. **Site oficial da linguagem Java**. Disponível em: http://java.sun.com. Último acesso: fevereiro de 2005.

JAVACC. **Site oficial do JavaCC**. Disponível em: https://javacc.dev.java.net/. Último acesso: fevereiro de 2005.

JDT. **JDT Plug-in Developer Guide**. Disponível em: http://www.eclipse.org/documentation/pdf/org.eclipse.jdt.doc.isv_3.0.1.pdf. Último acesso em: fevereiro de 2005.

JJTREE. **Site oficial do JJTree**. Disponível em: https://javacc.dev.java.net/doc/JJTree.html. Último acesso: fevereiro de 2005.

JOHNSON, S.C. **Lint, a C program checker**, In Unix Programer's Manual, volume 2A, capítulo 15, páginas 292-303. Bell Laboratories, 1978.

MACDONALD, F., MILLER, J., BROOKS, A., ROPER, M., WOOD, M. **A review of tool support for software inspection**, Proc. 7th Int. Workshop on CASE, 1995.

MAIA, M. E OLIVEIRA, A. **JPearl – Uma linguagem para descrição de reestruturações em programas Java**. VI Simpósio Brasileiro de Linguagens de Programação. Rio de Janeiro, Brasil, 2002.

MCCARTHY, P., PORTER, A. et al. **An experiment to assess cost-benefits of inspection meetings and their alternatives: A pilot study**. Proceedings of the International Metrics Symposium, Berlin, IEEE CS Press: 100-111, 1996.

MENEZES, PAULO B. **Linguagens Formais e Autômatos**, Editora Sagra Luzzato, 3ª Edição, 2000.

MIDDLEWARE RESEARCH. **J2EE Application Server Survey**, Junho de 2004.

PMD. **Site oficial do PMD**. Disponível em: https://pmd.sourceforge.net. Último acesso: fevereiro de 2005.

PORTER, A., SIY, H. P. et al. **An experiment to assess the cost-benefits of code inspections in large scale software development**, IEEE Transactions on Software Engineering 23(6): 329-46, 1997.

RUSSEL, Glen W. **Experience with Inspections in Ultralarge-Scale Developments**, IEEE Software, Vol. 8, No. 1, Janeiro de 1991, pp.25-31.

SAPSOMBOON, Bordin. **Software Inspection and Computer Support**, Tese de Doutorado - University of Pittsburgh, School of Information Sciences, Junho de 1999.

WELLER, Edward F. **Lessons from Three Years of Inspection Data**, IEEE Software, Vol. 10, No. 5, Setembro de 1993, pp.38-45.

XML. **XML Specifications**. Official Web Site for XML. Disponível em: http://wwww3c.org/XML. Último acesso: fevereiro de 2005.

XPATH. **Site oficial do XPath**. Disponível em: http://www.w3.org/TR/xpath. Último acesso: maio de 2005.

9. APÊNDICES

9.1. APÊNDICE 1: CÓDIGO PDL DOS PADRÕES

Padrão 7.2.2

```
<CompilationUnit _Mark="7.2.2 - Mais de uma Classe no mesmo arquivo">
    <TypeDeclaration />
    <TypeDeclaration />
</CompilationUnit>
```

Padrão 7.2.3

```
<CompilationUnit>
    <ImportDeclaration isOnDemand="true"
                       _Mark="7.2.3 - Importe Classes e nao Packages" />
</CompilationUnit>
```

Padrão 7.2.4

```
<CompilationUnit>
  <Or>
    <MethodDeclaration isPublic="true"
                       _Mark="7.2.4 - Metodos publicos devem ter
                              no maximo 2 niveis de aninhamento">
      <Block>
        <Block>
          <Block>
            <Block />
          </Block>
        </Block>
      </Block>
    </MethodDeclaration>

    <MethodDeclaration isPublic="false"
                       _Mark="7.2.4 - Metodos nao publicos devem ter no
                              maximo 3 niveis de aninhamento">
      <Block>
        <Block>
          <Block>
            <Block>
              <Block />
            </Block>
          </Block>
```

```
            </Block>
          </Block>
        </MethodDeclaration>
    </Or>
</CompilationUnit>
```

Padrão 7.2.5

```
<CompilationUnit>
    <Or>
        <FieldDeclaration _Mark="7.2.5 - Mais de uma atributo no mesmo
                                 Statement">
            <VariableDeclarationFragment />
            <VariableDeclarationFragment />
        </FieldDeclaration>
        <VariableDeclarationStatement _Mark="7.2.5 - Mais de uma variável
                                              no mesmo Statement">
            <VariableDeclarationFragment />
            <VariableDeclarationFragment />
        </VariableDeclarationStatement>
    </Or>
</CompilationUnit>
```

Padrão 7.2.6

```
<MethodDeclaration _Mark="7.2.6 - Metodo com mais de 5 parametros">
      <SingleVariableDeclaration _MaxSubTreeLevel="1"
                                 _InParentSubTree="parent" />
      <SingleVariableDeclaration _MaxSubTreeLevel="1"
                                 _InParentSubTree="parent" />
      <SingleVariableDeclaration _MaxSubTreeLevel="1"
                                 _InParentSubTree="parent" />
      <SingleVariableDeclaration _MaxSubTreeLevel="1"
                                 _InParentSubTree="parent" />
      <SingleVariableDeclaration _MaxSubTreeLevel="1"
                                 _InParentSubTree="parent" />
      <SingleVariableDeclaration _MaxSubTreeLevel="1"
                                _InParentSubTree="parent" />
</MethodDeclaration>
```

Padrão 8.2.2

```
<SwitchStatement _Mark="8.2.2 - Todos os switch statements devem
                        conter um default case">
    <SwitchCase _Not="true" isDefault="true" />
</SwitchStatement>
```

Padrão 8.2.8

```
<TypeDeclaration>
  <Or>
    <InfixExpression operator="==" _Mark="8.2.8 - Contante deve ficar do
                                          lado esquerdo da comparacao">
        <SimpleName _InParentSubTree="leftOperand" />
        <BooleanLiteral _InParentSubTree="rightOperand" />
    </InfixExpression>
    <InfixExpression operator="==" _Mark="8.2.8 - Contante deve ficar do
                                          lado esquerdo da comparacao">
        <SimpleName _InParentSubTree="leftOperand" />
        <CharacterLiteral _InParentSubTree="rightOperand" />
    </InfixExpression>
    <InfixExpression operator="==" _Mark="8.2.8 - Contante deve ficar do
                                          lado esquerdo da comparacao">
        <SimpleName _InParentSubTree="leftOperand" />
        <NullLiteral _InParentSubTree="rightOperand" />
    </InfixExpression>
    <InfixExpression operator="==" _Mark="8.2.8 - Contante deve ficar do
                                          lado esquerdo da comparacao">
        <SimpleName _InParentSubTree="leftOperand" />
        <NumberLiteral _InParentSubTree="rightOperand" />
    </InfixExpression>
    <InfixExpression operator="==" _Mark="8.2.8 - Contante deve ficar do
                                           lado esquerdo da comparacao">
        <SimpleName _InParentSubTree="leftOperand" />
        <StringLiteral _InParentSubTree="rightOperand" />
    </InfixExpression>
   </Or>
</TypeDeclaration>
```

Padrão 8.2.9

```
<CompilationUnit>
  <Or>
      <ForStatement hasInitializer="false"
                    _Mark="8.2.9 - For deve ter um inicializador" />
      <ForStatement hasUpdaters="false"
                    _Mark="8.2.9 - For deve ter um update" />
  </Or>
</CompilationUnit>
```

Padrão 8.2.10

```
<IfStatement>
    <Assignment _InParentSubTree="expression"
                _Mark="8.2.10 - Atribuicao dentro da condicao do IF" />
</IfStatement>
```

Padrão 8.2.11

```
<MethodDeclaration>
  <Or>
    <ForStatement _Mark="8.2.11 - For com corpo vazio">
        <EmptyStatement _MaxSubTreeLevel="1"
                        _InParentSubTree="parent" />
    </ForStatement>
    <ForStatement _Mark="8.2.11 - For com corpo vazio">
      <Block _MaxSubTreeLevel="1" _InParentSubTree="parent">
          <ASTNode _Not="true" />
      </Block>
    </ForStatement>
  </Or>
</MethodDeclaration>
```

Padrão 8.2.12

```
<MethodDeclaration>
  <ForStatement _Mark="8.2.12 - For com variavel de controle
                       atualizada no corpo">
      <Assignment _MaxSubTreeLevel="1" _InParentSubTree="parent" />
      <SimpleName identifier=".*" _Bind="X,identifier" />
    <Block _MaxSubTreeLevel="1" _InParentSubTree="parent">
      <Assignment>
```

```
            <SimpleName _Bind="X,identifier"
                _InParentSubTree="leftHandSide" />
        </Assignment>
      </Block>
   </ForStatement>
</MethodDeclaration>
```

Padrão 8.2.13

```
<MethodDeclaration>
   <Or>
      <IfStatement _Mark="8.2.13 - If com corpo vazio">
          <EmptyStatement _MaxSubTreeLevel="1"
                          _InParentSubTree="parent" />
      </IfStatement>
      <IfStatement _Mark="8.2.13 - If com corpo vazio">
         <Block _MaxSubTreeLevel="1" _InParentSubTree="parent">
            <ASTNode _Not="true" />
         </Block>
      </IfStatement>
   </Or>
</MethodDeclaration>
```

Padrão 8.3.1.1

```
<FieldDeclaration isFinal="true" _Mark="8.3.1.1 - Constantes devem ter nomes
                                        somente com maiusculas">
   <VariableDeclarationFragment>
       <SimpleName identifier=".*[a-z].*" _InParentSubTree="name" />
   </VariableDeclarationFragment>
</FieldDeclaration>
```

Padrão 8.3.1.2

```
<CompilationUnit>
   <Or>
      <VariableDeclarationStatement _Mark="8.3.1.2 - Variavel deve ter o
                                           nome comecado com minuscula">
         <VariableDeclarationFragment>
            <SimpleName identifier="[A-Z].*" _InParentSubTree="name" />
         </VariableDeclarationFragment>
      </VariableDeclarationStatement>
      <SingleVariableDeclaration _Mark="8.3.1.2 - Variavel deve ter o nome
```

```
                                        comecado com minuscula">
        <SimpleName identifier="[A-Z].*" _InParentSubTree="name" />
    </SingleVariableDeclaration>  </Or>
</CompilationUnit>
```

Padrão 8.3.1.3

```
<CompilationUnit>
  <PackageDeclaration _Mark="8.3.1.3 - Pacotes devem ter o nome
                              completamente minusculo">
      <SimpleName identifier="[A-Z].*" />
  </PackageDeclaration>
</CompilationUnit>
```

Padrão 8.3.3

```
<CompilationUnit>
  <Or>
    <ForStatement _Mark="8.3.3 - For com nome de variavel de controle
                          int diferente de (i,j,k)">
      <VariableDeclarationExpression _InParentSubTree="initializer">
        <VariableDeclarationFragment>
          <SimpleName identifier="[^i-k]" resolveTypeBinding="int"
                          _InParentSubTree="name" />
        </VariableDeclarationFragment>
      </VariableDeclarationExpression>
    </ForStatement>
    <ForStatement _Mark="8.3.3 - For com nome de variavel de controle
                          int diferente de (i,j,k)">
      <Assignment _InParentSubTree="initializer">
        <SimpleName identifier="[^i-k]" resolveTypeBinding="int"
                      _InParentSubTree="leftHandSide" />
      </Assignment>
    </ForStatement>
  </Or>
</CompilationUnit>
```

Padrão 8.4.1

```
<TypeDeclaration name="[^A-Z].*"
                    _Mark="8.4.1 - Classes e Interfaces devem
                              comecar com letra Maiuscula" />
```

Padrão 8.5.2

```
<NumberLiteral token=".*l" _Mark="8.5.2 - Usar L ao inves de l para long" />
```

Padrão 8.6.1

```
<MethodDeclaration name="[A-Z].*" isContructor="false"
                    _Mark="8.6.1 - Metodo deve comecar com letra Minuscula" />
```

Padrão 9.6

```
<CastExpression _Mark="9.6 - A referencia super eh equivalente
                          ao uso de casting com this">
    <ThisExpression />
</CastExpression>
```

Padrão 9.7

```
<TypeDeclaration>
  <FieldDeclaration>
    <VariableDeclarationFragment>
        <SimpleName identifier=".*" _Bind="X,identifier" />
    </VariableDeclarationFragment>
  </FieldDeclaration>
  <Or>
    <MethodDeclaration _Mark="9.7 - Nome do metodo ou parametro
                                coincide com atributo da classe">
        <SimpleName _Bind="X,identifier" _MaxSubTreeLevel="1"
           _InParentSubTree="parent" />
    </MethodDeclaration>
    <MethodDeclaration _Mark="9.7 - Nome do metodo ou parametro
                                coincide com atributo da classe">
      <SingleVariableDeclaration>
          <SimpleName _Bind="X,identifier" />
      </SingleVariableDeclaration>
    </MethodDeclaration>
  </Or>
</TypeDeclaration>
```

Padrão 10.1.2

```
<CompilationUnit>
  <FieldDeclaration isStatic="true">
    <VariableDeclarationFragment>
      <SimpleName identifier=".*[a-z].*" _InParentSubTree="name"
                  _Bind="X,identifier" />
    </VariableDeclarationFragment>
  </FieldDeclaration>
  <MethodDeclaration isConstructor="true" _Mark="10.1.2 - Nao inicialize
                                                 atributos static no contrutor">
    <Assignment>
      <SimpleName _InParentSubTree="leftHandSide"
                  _Bind="X,identifier" />
    </Assignment>
  </MethodDeclaration>
</CompilationUnit>
```

Padrão 10.1.3.1

```
<MethodDeclaration isConstructor="true" isPublic="true"
                   _Mark="10.1.3.1 - Contrutores publicos podem ter no
                          maximo 2 niveis de aninhamento entre comandos">
  <Block>
    <Block>
      <Block>
        <Block />
      </Block>
    </Block>
  </Block>
</MethodDeclaration>
```

Padrão 10.1.3.2

```
<CompilationUnit>
  <Or>
    <TypeDeclaration>
      <FieldDeclaration _Not="true" isStatic="false" />
      <MethodDeclaration _Not="true" isStatic="false" />
      <MethodDeclaration isPublic="true" isConstructor="true"
        _Mark="10.2.3.2 - Classe que apenas empacota um grupo de
        metodos e atributos estaticos deve ter construtor privado" />
    </TypeDeclaration>
```

```
      <TypeDeclaration>
          <FieldDeclaration _Not="true" isStatic="false" />
          <MethodDeclaration isPublic="true" isConstructor="true"
             _Mark="10.2.3.2 - Classe que apenas empacota um grupo de
             metodos e atributos estaticos deve ter construtor privado" />
          <MethodDeclaration _Not="true" isStatic="false" />
      </TypeDeclaration>
   </Or>
</CompilationUnit>
```

Padrão 11.2

```
<FieldDeclaration isPublic="true"
    _Mark="11.2 - Maximize o uso de encapsulamento de atributos(private)" />
```

Padrão 11.6

```
<TypeDeclaration>
   <TypeDeclaration>
       <TypeDeclaration _Mark="11.6 - Evite mais de dois niveis de
                                   aninhamento de classes" />
   </TypeDeclaration>
</TypeDeclaration>
```

Padrão 13.2.1

```
<MethodDeclaration returnTypeIsArray="true"
                    _Mark="13.2.1 - Retorne um array vazio e nao null">
   <ReturnStatement>
       <NullLiteral />
   </ReturnStatement>
</MethodDeclaration>
```

Padrão 14.2

```
<CatchClause _Mark="14.2 - Evite blocos catch em corpo vazio">
   <Block>
       <ASTNode _Not="true" />
   </Block>
</CatchClause>
```

Padrão 15.6

```
<InfixExpression operador="+="
                    _Mark="15.6 - Evite concatenacao de String com +=">
    <Expression resolveTypeBinding="java.lang.String"
        _InParentSubTree="leftOperand" _MaxSubTreeLevel="1" />
</InfixExpression>
```

Padrão 15.12

```
<TypeDeclaration>
  <MethodInvocation _Mark="15.12 - Evite utilizar String.equals ('literal'),
                                  use 'literal'.equals (String)">
      <SimpleName _InParentSubTree="expression"
          resolveTypeBinding="java.lang.String" />
      <SimpleName _InParentSubTree="name" identifier="equals" />
      <StringLiteral _InParentSubTree="arguments,0" />
  </MethodInvocation>
</TypeDeclaration>
```

Padrão 16.1

```
<VariableDeclarationFragment _Mark="16.1 - Evite usar a keyword 'new' quando
                                criar objetos do tipo String para armazenar literais">
   <SimpleName _InParentSubTree="name" resolveTypeBinding="java.lang.String" />
  <ClassInstanceCreation argumentsSize="1">
      <StringLiteral _InParentSubTree="arguments,0" />
  </ClassInstanceCreation>
</VariableDeclarationFragment>
```

Padrão 16.3

```
<VariableDeclarationStatement>
    <ArrayType _InParentSubTree="type" />
  <VariableDeclarationFragment _Mark="16.3 - Evite o uso de 'Date[]',
                                          use 'long[]'">
      <SimpleName _InParentSubTree="name"
          resolveTypeBinding="java.util.Date\[\]" />
  </VariableDeclarationFragment>
</VariableDeclarationStatement>
```

Padrão 16.4

```
<FieldDeclaration _Mark="16.4 - Evite 'static' collections">
   <VariableDeclarationFragment>
        <SimpleName _InParentSubTree="name"
           resolveTypeBinding="java.util.Collection" />
   </VariableDeclarationFragment>
</FieldDeclaration>
```

Padrão 16.6

```
<MethodDeclaration>
   <Or>
      <ForStatement _Mark="16.6 - Evite chamar metodos dentro do
                                  statement de condicao de um loop">
          <MethodInvocation _InParentSubTree="expression" />
      </ForStatement>
      <WhileStatement _Mark="16.6 - Evite chamar metodos dentro do
                                    statement de condicao de um loop">
          <MethodInvocation _InParentSubTree="expression" />
      </WhileStatement>
   </Or>
</MethodDeclaration>
```

Padrão 16.7, 16.8

```
<IfStatement  _Mark="16.7 - Uso do operador condicional if
                        (cond) return else return">
   <Block _InParentSubTree="thenStatement">
        <ASTNode _Not="true" />
        <ReturnStatement />
        <ASTNode _Not="true" />
   </Block>
   <Block _InParentSubTree="elseStatement">
        <ASTNode _Not="true" />
        <ReturnStatement />
        <ASTNode _Not="true" />
   </Block>
</IfStatement>
```

```
<IfStatement _Mark="16.8 - Uso do operador condicional
                    if (cond) a = b; else a = c">
  <Block _InParentSubTree="thenStatement">
      <ASTNode _Not="true" />
      <Assignment />
      <ASTNode _Not="true" />
  </Block>
  <Block _InParentSubTree="elseStatement">
      <ASTNode _Not="true" />
      <Assignment />
      <ASTNode _Not="true" />
  </Block>
</IfStatement>
```

Padrão 16.10

```
<MethodDeclaration>
  <Or>
    <ForStatement _Mark="16.10 - Evite chamadas de metodos
                          'synchronized' em um loop">
      <MethodInvocation isSynchronized="true" />
    </ForStatement>
    <WhileStatement _Mark="16.10 - Evite chamadas de metodos
                    'synchronized' em um loop">
      <MethodInvocation isSynchronized="true" />
    </WhileStatement>
  </Or>
</MethodDeclaration>
```

Padrão 16.11

```
<MethodDeclaration>
  <Or>
    <ForStatement _Mark="16.11 - Coloque 'try_catch' fora dos loops">
      <TryStatement />
    </ForStatement>
    <WhileStatement _Mark="16.11 - Coloque 'try_catch' fora dos loops">
      <TryStatement />
    </WhileStatement>
  </Or>
</MethodDeclaration>
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo