UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Thiago Leão Moreira

JMinimizer: Um Compactador de Aplicações Java.

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis, Novembro de 2004

JMinimizer: Um Compactador de Aplicações Java.

Thiago Leão Moreira

Esta dissertação foi julgada adequada para a obtenção do título de Bacharel em Ciência da Computação, e aprovada em sua forma final pela Coordenadoria do Curso de Bacharelado em Ciência da Computação.

Prof. Dr. José Mazzucco Junior

Banca Examinadora

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Prof. Dr. Luiz Carlos Zancanella

M. Sc. Tiago Stein DÁgostini

"A verdadeira função do homem é viver
e não apenas existir."
- Jack London

"A mim, pelas noite não dorminadas

e pelas baladas perdidas..."

Resumo

O crescente aumento da popularidade da linguagem de programação Java levou também a um aumento, no desenvolvimento de frameworks e bibliotecas utilitárias, que facilitam o desenvolvimento de aplicações. No entanto estes frameworks e bibliotecas aumentam o tamanho da aplicação final, podendo até ultrapassar em muitas vezes o tamanho da aplicação que efetivamente resolve o problema de negócio. Frameworks e bibliotecas geralmente são desenvolvidos para resolver ou tratar de problemas para uma grande quantidade de situações. No entanto aplicações são criadas para resolver um problema específico, sendo assim essas aplicações não utilizam todos os recursos que um framework ou uma biblioteca oferecem. É nessas funcionalidades inúteis para a aplicação específica que o JMinimizer fará e/ou removerá as mesmas.

Keywords: Java, J2ME, celular, PDA's.

Sumário

Resumo **v**

Sumário **vi**

Lista de Figuras **viii**

Lista de Tabelas **ix**

1 Introdução **x**

1.1 Motivação x

1.2 Justificativa xi

1.3 Objetivos xi

1.3.1 Específicos xii

2 Outras Ferramentas e trabalhos **xiii**

2.1 Obfuscadores xiii

2.1.1 Retroguard xiv

2.1.2 ProGuard xiv

2.2 Compactadores xv

2.2.1 SophiaCompress(Java) xv

2.2.2 Jax xv

2.2.3 Jazz xvi

3 Fundamentação Teórica **xvii**

3.1 A estrutura do bytecode xvii

3.2 BCEL xxi

4 Compactador de Aplicações Java **xxiii**

5 Implementação do Compactador de Aplicações Java **xxiv**

5.1 Tipos de depêndencia xxiv

5.2 Arquivo de configuração xxv

5.3 Etapa de analise xxvi

5.4 Transformação xxvii

5.5 Limpeza do ”pool”de constantes xxviii

5.6 Peristência dos resultados xxix

6 Estudo de caso **xxx**

7 Trabalhos futuros **xxxii**

7.1 Submeter artigo ao BYTECODE 2005 xxxii

7.2 Integração a um obfuscador xxxii

7.3 Desenvolver um plugin para Maven xxxii

7.4 Ampliar as formas de declarar métodos *entry point* xxxiii

7.5 Compatibilização com o J2SE 5.0 xxxiii

7.6 Testar diferentes compiladores xxxiii

7.7 Geração de relatórios xxxiii

7.8 Desenvolver uma interface gráfica xxxiv

8 Metodologia **xxxv**

9 Conclusão **xxxvii**

10 Código Fonte **xxxviii**

Referências Bibliográficas **c**

Lista de Figuras

3.1 formato de um arquivo .class xviii
3.2 estrutura de um campo xix
3.3 estrutura de um método xix
3.4 diagrama de classe de BCEL xxii

Lista de Tabelas

3.1 Compilação com e sem os atributos de depuração xx

6.1 Resultado da execução do JMinimizer nas classes do JMinimizer xxxi

Capítulo 1

Introdução

Este trabalho consiste do estudo e desenvolvimento de uma aplicação capaz de analisar uma outra aplicação Java[jav 03] e apartir dessa analise transformar a aplicação, de forma que seu comportamante não se modifique, tentando reduzir o seu tamanho original.

1.1 Motivação

A popularidade da linguagem de programação Java[jav 03], principalmente para pequenos dispositivos[j2m 03] (celulares, PDA's [1], smart phones, etc), resultou num aumento significante de bibliotecas utilitárias e frameworks que facilitam e aumentam a produtividade no desenvolvimento de aplicações para esta linguagem. Tais bibliotecas e frameworks são utilizados como infra-estrutura na solução de problemas específicos. Bibliotecas para logging, manipulação de documentos XML, construção de interfaces gráficas, frameworks para desenvolvimento WEB, para persistência de dados, etc... são algumas aplicações que estas bibliotecas de classes possuem. Infelizmente, o uso de bibliotecas de terceiros tem uma desvantagem: as bibliotecas que uma aplicação depende podem não estar presentes no ambiente de execução. Conseqüentemente, digamos que uma certa aplicação dependa do projeto Regexp (http://jakarta.apache.org/regexp/), biblioteca para avaliação de expressões regulares, os usúarios dessa aplicação deverão instalar Regexp no ambiemte de execução ou Regexp deverá ser disponibilizada junto com a aplicação. A primeira tática é muitas vezes desencorajada porque é incomoda e pode levar a erros de instalação (por exemplo, os usuário podem instalar uma versão imcompativel de Regexp gerando erros inesperados), já a segunda iniciativa - disponibilizar Regexp juntamente com a aplicação - geralmente

[1] Personal Digital Assistant

há um aumento significante do tamanho da aplicação. Em conseqüência do aumento do tamanho da aplicação também aumentará o tempo para se realizar o download (se for esta a forma de distribuição do aplicativo) e aumentará o espaço necessário para acomodar a aplicação no dispositivo. Este último é de suma importância quando desenvolvemos aplicações para a plataforma J2ME[2], onde alguns dispositivos alvos possuem somente 64 kilobytes de amazenamento para uma aplicação e um total de 512 kilobytes para todas as aplicações. Isto exposto verificamos que a utilização de bibliotecas de terceiros pode resolver o problema de desenvolver e depurar classes para a infra-estrutura e criar outros problemas relacionados ao armazenamento e ao tempo de obtenção da aplicação. No entanto este segundo, parece ser de mais fácil solução e não terá abordagem nesse projeto. Já o segundo - relacionado ao armazenamento da aplicação - é mais preocupante, visto que, sem espaço suficiente para armazenar a aplicação não podemos nem iniciar o download da mesma. Uma primeira alternativa de solução para o problema, seria aumentar a capacidade de armazenamento do aparelho/dispositivo ou adquirir um aparelho/dispositivo similar com maior capacidade de armazenamento. Mas se não for possível aumentar a capacidade de armazenamento ? Nem de trocar de aparelho/dispositivo ? Uma segunda solução seria tentar retirar do código final da aplicação e de suas dependências todo o tipo de informação e estrutura que não irá afetar a execução normal do aplicativo. E é nesta segunda solução que este trabalho de conclusão de curso se propoê em criar.

1.2 Justificativa

O desenvolvimento dessa dissertação contribuirá para a comunidade científica no esclarecimento da estrutura do *bytecode* Java[jav 03] e quais dessas estruturas podem ser removidas. A grande curiosidade que sempre tive em relação ao *bytecode* Java[jav 03], que é meu instrumento de trabalho, e a necessidade de desenvolver um trabalho de conclusão de curso para Universidade Federal de Santa Catarina me levaram a desenvolver essa dissertação.

1.3 Objetivos

Desenvolver uma ferramenta de desenvolvimento de aplicações Java[jav 03] que auxilie o *deployment* dessas aplicações na maior quantidade de dispositivos quem suportam a plataforma J2ME[j2m 03]

[2]Java 2 Micro Edition

1.3.1 Específicos

Esse projeto apresenta os seguintes objetivos específicos.

- Conhecimento
  - Consolidar meus conhecimentos na técnologia Java[jav 03].
  - Aprender as etapas de desenvolvimento de uma aplicação open source.
- Criar uma ferramenta capaz de diminuir o tamanho de uma aplicação Java[jav 03]

Capítulo 2

Outras Ferramentas e trabalhos

Nas pesquisas desenvolvidas foi observado que muitos trabalhos já foram realizados na busca por uma diminuição das aplicações Java[jav 03]. Alguns propondo uma reestruturação do bytecode java [HOR 98], outros sugerindo uma compactação diferente para os arquivos JAR [jar 04] [BRA 98], no entanto estes propoem mudanças estruturais muito porfundas, tanto nos arquivos class ou jars, tão profundas que só ferramentas especiais são capazes de ler estes tipos de arquivo. Porem também encontrei ferramentas que reduzem as aplicações Java sem modificar sua estrutura básica e assim, são perfeitamentes compatíveis com a especificação Java [jav 04] entre elas estão o Jax[jax 04], SophiaCompression[sop 04] entre outras. A seguir serão descritas brevemente cada uma das ferramentas que estudei.

2.1 Obfuscadores

Obfuscadores são ferramentas de grande utilização na distribuição de aplicações Java eles não impedem, mas dificultam a engenharia reversa que consiste em a partir do arquivos class obter um arquivo fonte. Além de obfuscarem os arquivos class essas ferramentas tem como efeito colateral a diminuição da aplicação final, por isso foram objetos de pesquisa dessa tese. Isto ocorre devido a troca dos nomes das classes e pacotes por nomes mais simples e compactos por exemplo, a classe original com nome *net.java.dev.jminimizer.JMinimizer* é renomeada para *a.b.c.d.A*, isto impacta significativamente na diminuição dos arquivos class.

2.1.1 Retroguard

O Retroguard é um obfuscador de bytecode, uma ferramenta projetada para substituir identificadores e atributos de compreensão fácil por humanos por strings sem sentido, tornando a engenharia reversa quase impossível. O resultado da execução do Retroguard é aplicações menores e com o código fonte protegido. Retroguard é distribuido sobre licensa GNU LGPL[1]. Algumas características:

1. redução do tamanho do bytecode Java (reduzindo 50% é possível, 20-30% é típico) levando à obtenção mais rápida das aplicações
2. projetado para ser fácilmente incorporado ao processo de desenvolvimento de aplicaçlões Java.
3. permite uma costumização completa do processo de obfuscação.
4. suporta multiplos *entry points*.
5. atual sobre arquivos JAR[jar 04].
6. obfuscação é controlado por uma linguagem de script flexível.
7. uma interface gráfica é provida para um simples gerenciamento de scripts.
8. usa massivamente sobrecarga de nomes de métodos e campos para um aumento da segurança.
9. gera unicamente bytecode Java verificado e completamente compatível com a especificação da máquina virtual Java.
10. atualiza o arquivo Manifest dos arquivos JAR, utilizando nomes obfuscados e automaticamente gerando menssagens sumário MD5 e SHA-1.

2.1.2 ProGuard

ProGuard é um otimizador e um obfuscador para bytecode Java. Ele pode detectar e remover classes, métodos, campos e atributos que não são usados. Também pode otimizar e remover instruções não usadas. Finalmente, ele renomeia classes, campos e métodos usando nomes curtos e sem sentido, resultando em arquivos JARs menores e de difícil execução de uma possível engenharia reversa.

[1]Lesser General Public License

2.2 Compactadores

Os compactadores são ferramentas com o mesmo intuito desse projeto, diminuir o tamanho da aplicação final. Foram encontradas algumas técnicas de compressão de aplicações Java, muitas delas sugerindo novas estruturas tanto para arquivos class e como para JAR's, além de outras que não modificam nenhum tipo de estrutura.

2.2.1 SophiaCompress(Java)

SophiaCompress(Java) é um compactador de aplicações Java, desenvolvido especialmente para o profile MIDP[2] 1.0 de J2ME[3] atulamente está na versão 2.0 e é desenvolvido por uma empresa japonesa a Sophia Cradle[sop 04], que além de produzir o SophiaCompress para Java também possui uma versão para Brew[bre 04]. SophiaCompress não é um software livre e seu uso é licensiado através do pagamento de licensas. Uma característica importante relacionada a esse projeto é que as transformações feitas no código não violam nenhuma especificação da linguagem Java bem como da JSR[4] 37. Algumas das características que SophiaCompress possui:

1. Diminuição dos nomes de classes, métodos e campos.
2. Compartilhamento de nomes de classes, métodos e campos.
3. Substituição de conjunto de instruções por mais curtos.
4. Inline de métodos.
5. Remoção de classes, métodos e campos que não são utilizados, inclusive dados sobre estas estruturas do Constant Pool.
6. Remoção de instruções não utilizadas.
7. Fundição de classes

2.2.2 Jax

O principal interesse do projeto Jax é a redução do tamanho das aplicações Java visando a redução do tempo de sua obtenção. Jax lê num arquivo class que constitui a aplicação

[2]Mobile Information Device Profile
[3]Java 2 Micro Edition
[4]Java Specification Request

Java, e executa uma analise na aplicação inteira para determinar os componentes, isto é, classes, métodos e campos da aplicação que devem ser mantidos para preservar o comportamento da aplicação. A seguir Jax aplica diversas transformações visando a redução do tamanho na aplicação, e cria uma arquivo JAR contendo a aplicação reduzida. As transformações incorporadas no Jax, atualmente incluem: remoção de atributos redundantes como LocalVariable e LineNumber, remoção de métodos e campos não utilizados, inline de métodos, nos casos que há redução do tamanho da aplicação, transformação da hierarquia de classes e renomeação de pacotes, classes, métodos e campos. Para aplicações com mais de 2.300 classes foram medidos reduções de até 90%, onde as grandes reduções, geralmente, ocorreram em aplicações baseadas em bibliotecas. Mas a média de redução observado é de 50%. Quando este projeto foi elaborado o projeto Jax já havia sido incorporado pelo produto WebSphere Studio Device Developer[wsd 04] e não possui mas link para sua obtenção.

2.2.3 Jazz

O formato de arquivo Jazz[BRA 98] pretende ser um substituto para o formato de arquivo JAR, quando for usado para armazenamento e distribuição de programas Java. Um arquivo Jazz possui uma compressão além do que é possível com um arquivo JAR. O tamanho reduzido do arquivo Jazz permite transmissões mais rápidas através da rede e ainda um benefício adicional de conservar espaço em disco. A compressão dos arquivos Jazz é obtidas através de uma combinação de diferentes métodos de compressão de dados, adaptados ao conjunto de características de uma coleção de classes Java. Apesar de a compressão ser maior dos arquivos Jazz, eles não seguem as especificações que um arquivo JAR segue, portanto só ferramentas ou JVM's especiais podem ler e descompactar um arquivo nesse formato. Atualmente esse formato de arquivo não é padrão para a tecnológia Java. Para se tornar padrão é necessário que toda a industria relacionada a tecnológia Java adote esse formato ou então que ferramentas e JVM's suportem os dois formatos. O primeiro passo para a incorporação desse formato na tecnológia Java é criar uma JSR no JCP[5], que é o orgão responsável pela padronização de tecnológias relacionadas à Java. Do contrário esse projeto permanecerá como uma pesquisa.

[5] Java Community Process

Capítulo 3

Fundamentação Teórica

Para poder realizar as transformações que este trabalho se porpoê em fazer é necessário antes, uma breve explicação da estrutura do objeto alvo dessa dissertação, o bytecode Java. O produto final do desenvolvimento de uma aplicação Java é um ou mais arquivos class, também chamados de bytecode. Cada arquivo com a extensão class representa uma classe ou interface na aplicação, tanto um como o outro possuem a mesma estrutura em arquivo, há somente uma flag diferenciando um do outro. Esses arquivos class são o resultado da compilação dos arquivos de código fonte, e sua estrutura é praticamente um mapeamento um para um da linguagem Java. É atravez desses arquivos class que foram feitas analises e transformações na sua estrutura para reduzir o tamanho das aplicações Java.

3.1 A estrutura do bytecode

Cada arquivo com a extensão class representa uma classe ou uma interface na linguagem Java. Esses arquivos são constituídos de um array de bytes.

A figura 3.1 exemplifica de maneira simples a organização e o significado de cada byte ou conjunto de bytes na estrutura do arquivo class. O significado das representações u2 e u4 são: u representando um byte sem sinal e o decimal (2 e 4) representado a quantidade de bytes. Por exemplo, o valor magic é composto dos quatros primeiros bytes sem sinais do arquivo class. A seguir serão exemplificados cada uma das estruturas que compõem o bytecode.

1. magic: este numero é fixo para qualquer bytecode e tem o valor 0xCAFEBABE. O objetivo desse identificador é previnir as JVMs de carregar outras coisas que não sejam classes Java.

2. minor_version: determina a menor versão que o bytecode suporta.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figura 3.1: formato de um arquivo .class

3. major_version: determina a maior versão que o bytecode suporta.

4. constant_pool_count: determina a quantidade de constantes disponíveis no pool de constantes.

5. constant_pool: estrutura que contem todas as constantes utilizadas no bytecode. Estas constantes podem ser Strings, ints, longs, doubles, etc.

6. access_flags: este numero mascara os tipos de acesso que esta classes ou interface pode conter.

7. this_class: é o índice no pool de constantes que contem o nome da classe.

8. super_class: é o índice no pool de constantes que contem o nome da super classe.

9. interfaces_count: determina a quantidade de interfaces que esta classe implementa diretamente, ou o numero de interfaces que esta interface estende.

10. interfaces: um array contendo índices para os nomes das interfaces no pool de constantes, que este bytecode implementa ou estende.

11. fields_count: determina a quantidade de campos que esta classe ou interface possui.

12. fields: uma tabela que contem estruturas que representam um campo. A figura 3.2 representa essa estrutura.

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figura 3.2: estrutura de um campo

(a) access_flags: este numero mascara os tipos de acesso que este campo pode ter.

(b) name_index: índice no pool de constantes que contem o nome do campo.

(c) descriptor_index: índice no pool de constantes que contem a assinatura do campo.

(d) attributes_count: determina a quantidade de atributos que este campo possui.

(e) attributes: tabela que contem estruturas que representam os atributos deste campo.

13. methods_count: determina a quantidade de métodos que esta classe ou interface possui.

14. methods: tabela que contem estruturas que representam um método. A figura 3.3 representa essa estrutura.

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figura 3.3: estrutura de um método

(a) access_flags: este numero mascara os tipos de acesso que este método pode ter.

(b) name_index: índice no pool de constantes que contem o nome do método.

(c) descriptor_index: índice no pool de constantes que contem a assinatura do método.

(d) attributes_count: determina a quantidade de atributos que este método possui.

(e) attributes: tabela que contem estruturas que representam os atributos deste método.

15. attributes_count: determina a quantidade de atributos que esta classe ou interface possui.

16. attributes: tabela que contem estruturas que representam os atributos desta classe ou interface.

Conhecendo a fundo o bytecode foram observadas estruturas que podem receber transformações ou até mesmo serem retiradas, visando a diminuição do tamanho da aplicação. Na especificação da linguagem Java, exatamente na seção 4.7 de The JavaTM Virtual Machine Specification[**?**, **?**, javaspec] está explicito que algumas das estruturas que encontramos no bytecode podem ser facilmente retiradas sem mudança no comportamento do software. Tais estruturas são: SourceFile, LineNumberTable e LocalVariableTable, elas representam respectivamente, o nome do arquivo fonte a tabela do número das linhas no código fonte e a tabela de variáveis locais dos métodos. Vários compiladores fornecem meios, através de parametros, de, na hora da geração do bytecode, os atributos responsáveis pela depuração serem excluídos do arquivo class. Esta é uma técnica de compactação de código Java simples. Além desses três atributos de classe, existe

Tabela 3.1: Compilação com e sem os atributos de depuração

Classe/Interface	com depuração (bytes)	sem depuração (bytes)
net.java.dev.jminimizer.Analyser	8.686	7.991
net.java.dev.jminimizer.JMinimizer	3.527	3.310
net.java.dev.jminimizer.Transformer	16.591	15.353
net.java.dev.jminimizer.util.ClassUtils	3.770	3.485
net.java.dev.jminimizer.util.Repository	247	208
net.java.dev.jminimizer.util.Visitor	245	209

também um atributo que sinaliza ao desenvolvedor que a classe, o método ou o campo não deve ser utilizado, pois ele entrou em desuso, esse atributo é chamado de Deprecated. Geralmente quando uma estrutura é marcada como Deprecated outra estrutura assume o papel da depreciada. Esse

atributo não está explicitamente referenciado na especificação da linguagem Java como podendo ser removido, mas como ele é apenas um sinalisador para o desenvolvedor e não influênciando na execução da aplicação pode também ser removido ser problema da aplicação final.

3.2 BCEL

No entanto, mesmo conhecendo a estrutura do bytecode, sua manipulação através de um software não é fácil, pois como já dito anteriormente, um arquivo class é uma array de bytes. Visto essa dificuldade foi criado uma biblioteca de classes que facilitam a manipulação de um arquivo class. Essa biblioteca é chamada de BCEL que é o acronomo de *Byte Code Engineering Library*[bce 03], que visa oferecer aos seus usuários uma maneira conveniente de analisar, manipular e criar arquivos class. BCEL representa as classes ou interfaces contidas nos arquivos class por objetos[boo 03], com um nível elevado de abstração, que possuem todas as informações desses arquivos, como: métodos, campos, lista de instruções dos métodos, herança, etc ... A figura 3.4 representa o diagrama de classe da API[1] de BCEL, responsável por mapear as estrutura do array de bytes em objetos de fácil manipulação pelo desenvolvedor. Tais objetos podem ser lidos de um arquivo (ou de qualquer stream de entrada), serem modificados por algum programa e gravados em arquivos novamente (ou enviados a um stream de saída). Também pode-se criar classes ou interfaces do zero em tempo de execução.

BCEL também é util na aprendizagem sobre a Java Virtual Machine (JVM) e o formato dos arquivos class. Compiladores, otimizadores, obfuscadores, geradores de código e ferremantes de análise, vem utilizando BCEL com sucesso. Varios desses projetos podem ser consultados em http://jakarta.apache.org/bcel/projects.html. Dada a existência de BCEL não foi necessário implementar um leitor de arquivos class, que é de suma importância para o desenvolver desse trabalho. Necessário foi, aprender a trabalhar com as ferramentas e conhecer a API de BCEL.

---

[1] Application Programming Interface

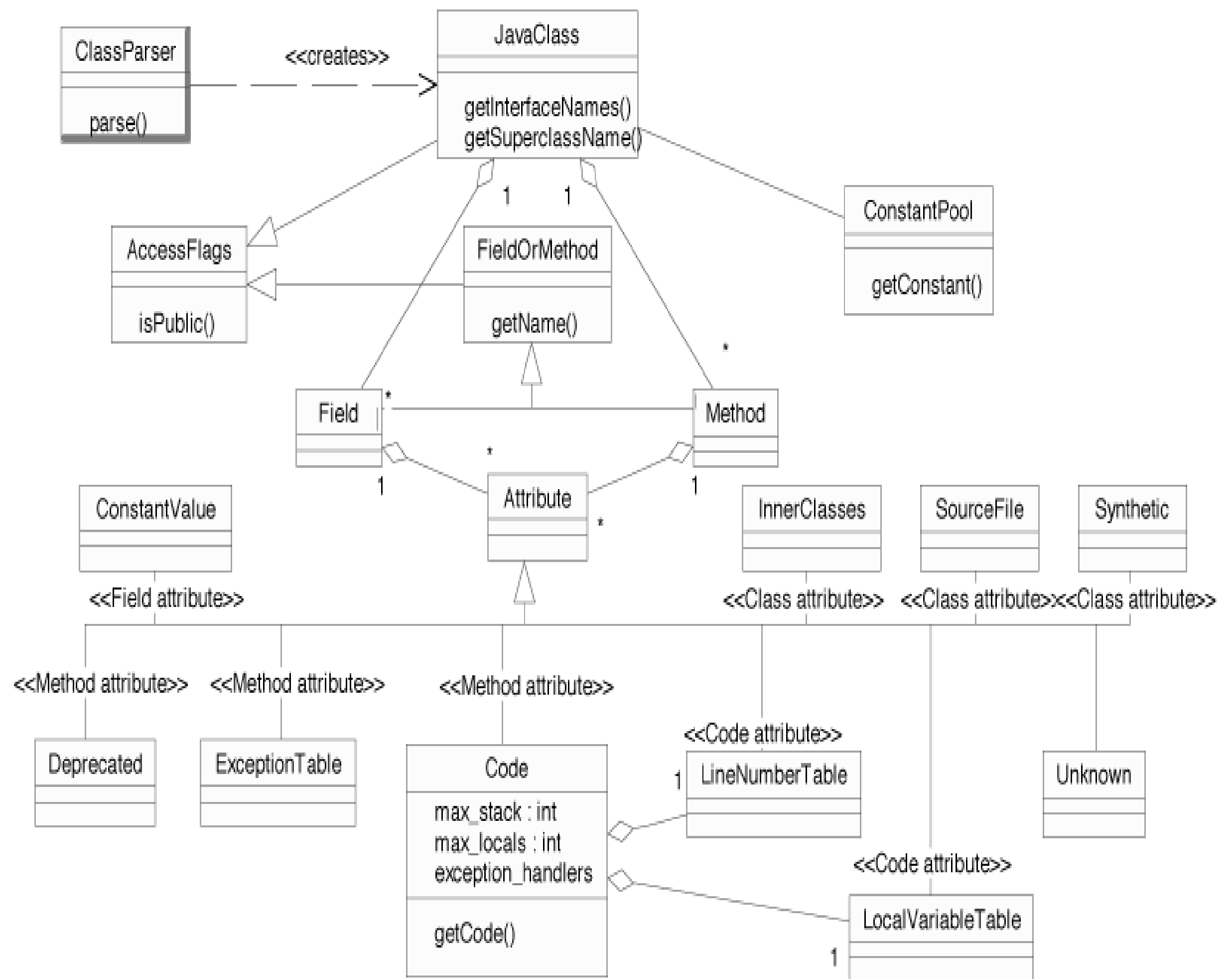

Figura 3.4: diagrama de classe de BCEL

Capítulo 4

Compactador de Aplicações Java

Atualmente a grande quantidade de biblietocas e frameworks para a linguagem Java facilitam e diminuem o tempo de desenvolvimento de aplicações para esta linguagem. Parsers XML, frameworks de persistenência, bibliotecas de logging, são exemplos de softwares já desenvolvidos e testados que são utilizados em larga escala por outras aplicações Java. No entanto estas bibliotecas e frameworks são projetadas para abrangerem a maior quantidade de situações que um desenvolvedor possa enfretar. Muitas vezes o desenvolvedor não utiliza todos os artifícios que uma bibliteca ou framework oferece, mas o código que não é utlizado também é disponibilizado juntamente com a aplicação final. Isso impacta na hora do usuário do aplicativo efetuar o download ou até impossibilita a instalação da aplicação por falta de espaço no dispositivo, este último aspecto aplicasse a plataforma J2ME. Como só parte da biblioteca ou framework será necessária a aplicação, a retirada da código não utilizado diminuiria o tempo de obtenção e ampliaria a gama de dispositivos capazes de executar a aplicação. Uma analise estática do código já compilado da aplicação, poderá nos fornecer as classes, métodos e campos que realmente fazem parte da aplicação e a partir desses dados eliminar tudo que não vier a ser realmente utilizado na execução do programa. Assim gerando um aplicação equivalente, no entanto, menor. É essa a finalidade do JMinimizer, eliminar tudo que não vier a ser utilizado na execução da aplicação.

Capítulo 5

Implementação do Compactador de Aplicações Java

5.1 Tipos de depêndencia

Um programa Java pode ter dois tipos de dependência relacionados a bibliotecas e frameworks de terceiros. O primerio tipo de dependência está vinculado ao ambiente em que a aplicação depois de pronta será executado, e nesse artigo a identificamos como dependência do tipo *runtime*. Suponhamos que estamos desenvolvendo um aplicativo para dispositivos móveis com suporte a Wireless Message API. Estes dispositivos possuem implementações das classes do pacote javax.wireless.messaging, sendo assim estas classes já estam disponiveis no ambiente de execução, no entanto para a compilação e para a analise estática do código elas são desconhecidas. Para a analise estática é preciso referencia-la, para que quando o JMinimizer começar a analiser a aplicação ele encontre todas as classes e interfaces que são referêncidas no código. O outro tipo de dependência não está relacionado ao ambiente de execução, no entanto também deve estar presente neste. Esse tipo de dependêcia é criada pelo desenvolvedor quando para solucionar problemas de infra estrutura tipo parsers XML, logging, persistência, este utiliza bibliotecas e/ou frameworks para resolve-los. Como esta dependência não está disponível no ambiente de execução ela deve ser disponibilizada juntamente com a aplicação. Aqui neste artigo a identificamos como dependência do tipo *program*. Dito isto já entedemos que as dependências do tipo *runtime* não precisam de nenhum tipo de tratamento, já que elas fazem parte do ambiente de execução. Já as dependências do tipo *program* podem e devem ser modificadas para diminuir o tamanho final da aplicação, já que elas devem ser disponibilizadas juntamento com o software.

5.2 Arquivo de configuração

O arquivo de configuração do JMinimizer possue secções para a devida declaração de quais bibliotecas fazem parte da dependência do tipo *runtime* e do tipo *program*.

Exemplo de dependêcia do tipo *program*.

```
<programClasspath>
        <directory path="./target/classes"/>
        <fileset directory="./lib">
                <file name="bcel-5.1.jar"/>
                <file name="jakarta-regexp-1.3.jar"/>
                <file name="commons-logging-1.0.3.jar"/>
                <file name="commons-cli-1.0.jar"/>
        </fileset>
        <fileset directory="./src/test/lib">
                <file name="commons-lang-1.0.1.jar"/>
        </fileset>
</programClasspath>
```

Também no arquivo de configuração é necessário declarar o *entry point* da aplicação. Geralmente o *entry point* é o método main, startApp (para Midlets) ou start (para Applets). Além do método *entry point* também é necessário declarar os métodos que serão chamados pelo ambiente de execução. Exemplo disto são os métodos startApp, pauseApp e destroyApp de um Midlet.

```
<class name="net.java.dev.jminimizer.Midlet">
        <method name="startApp">
                <arguments/>
                <return>
                        <void/>
                </return>
        </method>
        <method name="pauseApp">
                <arguments/>
                <return>
                        <void/>
                </return>
        </method>
        <method name="destroyApp">
                <arguments>
                        <primitiveType name="boolean"/>
                </arguments>
                <return>
                        <void/>
                </return>
        </method>
</class>
```

Existe a possibilidade também de declarar pontos de parada para o JMinimizer,

suponhamos que não há a necessidade de analisarmos classes do pacote **java.io**, basta para isso que declaremos no arquivo de configuração o seguinte trecho.

```
<notInspect>
        <pattern>java.io*</pattern>
</notInspect>
```

Feito isso todas as invocações de métodos de classes pertencentes ao pacote**java.io** não serão analisadas.

5.3 Etapa de analise

Tendo configurado as dependências os métodos que necessitam ser inspecionados e pontos de parada o JMinimizer irá método a método declarado inspecionar seu código a procura de novas invocações de métodos e acesso a atributos, tanto estáticos ou não. A medida que vai se achando novas invocações, essas chamadas de métodos e/ou atributos são adicionadas, se não pertencerem a um padrão de parada, à uma lista que contém uma única entrada para cada invocação de método ou acesso à atributo. No final do processo esta lista conterá todos os métodos e atributos que realmente compoêm o programa. Tanto métodos concretos, abstratos e nativos são adicionados a está lista, no entanto quando o JMinimizer encontra um método abstrato ou nativo ele não fará a inspeção do código, obviamente por este não o possuir. Durante este processo é verificado para cada novo método encontrado se este representa *_java.lang.Class.forName(java.lang.String className)* se sim o método que contêm a invocação deste método é adicionado a uma lista que será processada posteriormente e uma mensagem de alerta é enviada ao usuário informando-o que tal método possui invocação de *_java.lang.Class.forName(java.lang.String className).* Tudo isto é feito por que a linguagem Java suporta o carregamento dinâmico de classes. Dito isto, é necessário, para uma correta analise e tranformação do código, que o usuário declare no arquivo de configuração todas as classes que eventualmente poderão ser carregadas através da invocação do método que contêm a chamada à *_java.lang.Class.forName(java.lang.String className).* Finalizando o processo de analise, verificamos para todas as classes que foram encontradas, até então no processo, se estas classes possuem métodos que foram sobre escritos de suas classes e/ou interfaces pais. Se estas possuem métodos sobre escritos e que ainda não fazem parte da lista com todos os métodos da aplicação, estes serão adicionados a lista de métodos ainda não processados e o processo recomeçará. Ainda nessa etapa de analise é verificado para cada classe processada se está possue a invocação do método *pacote.NomeDaClasse.<cinit> ()V* que é o ”construtor”padrão

da classe. Esse método é invocado uma única vez após o carregamento da classe pela JVM. Ele é utilizado para setar valores a variáveis do tipo *static final*.

5.4 Transformação

O resultado dessa etapa de analise é uma lista, sem entradas repetidas, com todos os métodos e campos que fazem realmente parte da aplicação. A partir dessa lista e de uma segunda lista com todas a classes que estão disponíveis como dependências do tipo *program* será feita transformações visando a diminuição do código necessário para a execução da aplicação. A classe que efetua a transformação implementa o padrão *Visitor*, assim sendo ela percorreá todas as classes que foram encontradas durante a etapa de analise e verificará para cada uma delas se esta possue métodos ou atributos que podem ser removidos. Se o método ou campo pode ser removido, então ele é removido, caso contrario, e o método pertença a lista de métodos que invocam *_java.lang.Class.forName(java.lang.String className)*, é feita uma verificação no seu código para identificar se a chamada do método *_java.lang.Class.forName(java.lang.String className)* foi implementada pelo desenvolvedor ou se foi um artifício usado pelo compilador para transformar a construção: `Class number= Number.class` numa chamada ao método *java.lang.Class.forName(java.lang.String className)*. Caso tenha sido o compilador que tenha produzido este código duas ações serão tomadas:

1. Será criado um método, na classe corrente em analise, que será responsável unica e exclusivamente a carregar classes oriundas da construção `Class number= Number.class`. Este método terá acesso publico e estático com a finalidade de todas as classes da aplicação terem acesso a ele. Esta ação é tomada uma única vez. Ela ocorre na primeira vez que for encontrado um código escrito pelo compilador com a finalidade de transformar em *bytecode* a construção `Class number= Number.class`. Assim que a ação se conclui o nome da classe em que foi adicionado o método é armazenado para que o 2**?** passo seja executado sem problemas.

2. Será modificado o método que invoca *_java.lang.Class.forName(java.lang.String className)* para que a partir de agora ele invoque o método que foi criado no passo anterior.

O passo seguinte na transformação é retirar os atributos *Deprecated, SourceFile, LineNumberTable, LocalVariableTable, Synthetic* das classes e ou interfaces e de seus membros (métodos e campos), caso no arquivo de configuração tenha sido declarado que deve ser feita uma compactação radical.

A execução desse passo deve só ser feita quando o software foi testado exaustivamente, tanto na sua forma original como na forma compactada, pois os atributos que foram removidos são utilizados para debugging e portanto a aplicação deve estar estável para sofrer uma compactação radical.

5.5 Limpeza do ”pool”de constantes

O passo seguinte é com certeza o mais importante e também o que levou mais tempo para ser implementado. Diz respeito a limpeza do ”pool”de constantes, que é uma seção do bytecode que contém dados que são utilizados para referênciar classes, métodos e campos da própria classe e de outras classes. A limpeza se faz necessária pois quando são removidos métodos e campos de uma classe, o ”pool”de constantes não é atualizado, ou seja, não são removidas as constantes que referenciavam o método e/ou campo, nem as referências das invocações de métodos que o método removido possuia. Visto que a atualização do ”pool”não é feita, a quantidade de bytes não removidos pode ser muito maior que a quantidade removida, por este motivo que está etapa da compactação é de suma importância, pois remover um método pela metade não é nada eficiente. No entanto, a operação de remover as constantes do ”pool”é estremamente complicada, pois as contantes são armazenadas através de indices num array e são referênciadas na estrutura do bytecode através do seu indice nesse array, portanto quando uma constante é removida tem-se duas alternativas: ou, é mantido o tamanho do array e no lugar da constante removida é inserida uma outra, no entanto com um tamanho muito reduzido, ou, é redimensionado o array para que contenha somente espaço para as constantes que realmente são necessárias ao programa e se atualize os indices das constantes nas outras estruturas do bytecode. A primeira versão funcional do JMinimizer contemplava a primeira alternativa na tentativa de limpar o ”pool”de constantes e seu desenvolvimento foi simples, pois quando era encontrada uma constante que podia ser removida ela era somente substituida por uma muito menor. Já a segunda versão do JMinimizer buscou redimensionar o ”pool”de constantes e atualizar as estruturas que referênciavam as constantes que não eram excluídas. A tarefa de redimensionar o ”pool”foi simples de executar, porém a atualização dos índices das constantes que não eram excluídas, foi o que tomou mais tempo e esforço para ser implementado, pois era necessário percorrer todas as estruturas do bytecode (métodos, campos, conjunto de instruções, manipuladores de exceções. . .) e atualizar o índice das constantes que essa estrutura referênciava.

5.6 Peristência dos resultados

Finalizando o processo temos a persistência da classe compactada e de todos os arquivos que estão disponíveis no classpath da dependência do tipo *program* e que não são arquivos do tipo *bytecode*, entre eles estão arquivos XML, figuras, etc. O programa final pode ser persistido num diretório ou em um arquivo do tipo jar[jar 04], essa configuração é feita no arquivo de configuração do projeto. Após este processo, é persistido o documento XML contendo todos os métodos e campos que foram retirados das classes, para cada classe que sofrer modificações é criado e persistido uma arquivo XML contendo estas entradas.

Capítulo 6

Estudo de caso

Inicialmente o projeto teve como alvo a plataforma J2ME, subdividindo-se em perfis e configurações. Contudo uma outra técnologia pode, facilmente, tirar proveito dos benefícios que o JMinimizer pode trazer, essa técnologia é Applet. Applets são aplicativos Java que são executados dentro dos navegadores de Internet, eles são embutidos nas páginas HTML[1] e quando o navegador encontra uma tag que indica a existencia de um Applet o navegador invoca uma máquina virtual Java para interpretar e renderizar o Applet na página HTML. Normalmente os applets são disponibilizados na forma de um arquivo jar[jar 04], e este pode ser relativamente grande e levar um tempo elevado para ser totalmente recebido pelo navegador que irá renderiza-lo. A grande vantagem que o JMinimizer trará neste caso é a diminuição do tempo de recebimento do arquivo jar[jar 04], visto que computadores geralmente não possuem problemas de armazenamento. Percebido isto, vi na técnologia applet um outro campo de utilização do JMinimizer. E foi nesse outro campo que o JMinimizer foi utilizado primeiramente. Bem, como todo estudante de ciências da computação que estuda e trabalha, eu também gosto de fazer alguns projetos temporários e foi num desses projetos que eu vi uma oportunidade de experimentar o JMInimizer. O projeto era um site de encontros que possuiria um chat para que os assinantes pudessem se encontrar e conversar. O chat seria uma versão mais simples dos famosos MSN Messeger[msn 04] e ICQ[icq 04]. A primeira versão realmente foi uma versão simples de seus inspiradores. No entanto, os proprietários do site decidiram oferecer algo mais elaborado aos seus assinantes. Decidiram que o chat deveria oferecer opções como trocar a cor da fonte das caixas de conversação e permitir que o usuário inserisse *emoticons*, tudo isso sem perder compatibilidade com a versão 1.1 do Java, que era a versão que os sistemas operacionais Windows 2000 possuiam embutidas. Para tal esforço, foram

[1] HyperText Mark-up Language

encontradas duas soluções iniciais: a primeira seria desenvolver o chat utilizando o framework de interface gráfica chamado Thinlet[thi 04], que propoe o desenvolvimento de interfaces gráficas baseadas em arquivos XML e é compativel com a versão 1.1 do Java. No entanto esse framework deixou a desejar quando comecei a tratar os eventos de teclado e por isso foi abandonado. A segunda opção era utilizar swing, mas ela foi rapidamente descartada devido a não existência de tal pacote na versão 1.1 do Java. A partir desse ponto iniciou-se uma pesquisa na Internet para que encontrasse um framework que suprisse nossa necessidade e tivesse compatibilidade com a versão 1.1 do Java. Com a ajuda dos sites de busca encontramos um projeto, antigo, mas que se encaixava perfeitamente nos requisitos que necessitavamos. Tal projeto é chamado de KFC[2], e está disponivel em http://openlab.jp/kyasu/, esse projeto é uma ”reescrita”dos componentes do pacote *java.awt* adicionando características que só foram desenvolvidas futuramente para os componentes do pacote *javax.swing*. No entanto, o projeto é grande para ser obtido via internet, cerca de 626 kilobytes, principalmente se considerarmos as conexões discadas. A partir desse momento encontrei uma grande chance de testar e aprimorar o JMinimizer. Os primeiros testes com o applet se mostraram falhos, já que a aplicação não funcionava como deveria. Isso era gerado por diversos fatores que foram arrumados ao longo do desenvolvimento do JMInimizer. Em 21/05/2004 foi gerado uma versão estável que analisava e transformava com sucesso o chat e mais ainda diminuia sensivelmente o tamanho da aplicação, tornando assim praticavel a distribuição da mesma pela internet.

Tabela 6.1: Resultado da execução do JMinimizer nas classes do JMinimizer

Método	bytes
Sem transformação	719.692
Com transformação	293.890

[2]Kazuki YASUMATSU’s Foundation Classes

Capítulo 7

Trabalhos futuros

Apesar de ter chegado à uma versão funcional e estável, ainda existem melhorias que podem ser adicionadas ao JMinimizer. A seguir estão listadas, em ordem crescente de prioridade, as ações que serão tomadas em relação ao projeto.

7.1 Submeter artigo ao BYTECODE 2005

Elaborar e submeter um artigo ao *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation*[wor 04] que ocorrerá em Edimburgo, Escócia no dia 9 de abril de 2005.

7.2 Integração a um obfuscador

A integração com um obfuscador traria além de dificultar a engenharia reversa, uma diminuição do código final da aplicação. Esse segundo aspecto é de grande importância visto que o objetivo desse trabalho é mesmo diminuir o tamanho das aplicações. Exposto isso fica claro que uma integração com um obfuscador aumentará o percentual de compactação de uma aplicação.

7.3 Desenvolver um plugin para Maven

Atualmente uma grande quantidade de aplicativos possuem integração ao Maven, e essa integração é feita através de plugins. Uma versão beta desse plugin para o JMinimizer já esta em testes no entanto a falta de tempo paralizou seu desenvoilvimento.

7.4 Ampliar as formas de declarar métodos *entry point*

Nos primeiros testes com aplicações reais surgiu a necessidade de declarar métodos que devem ser analisados de uma forma diferente da que a usual, por exemplo por pacotes. Essa melhoria traria diminuição no tempo de configuração do JMinimizer.

7.5 Compatibilização com o J2SE 5.0

Recentemente foi lançado a versão 5.0 da plataforma J2SE[1], que traz entre outras coisas o recurso de Anotações, que possibilita o desenvolvedor marcar métodos e campos para que outras ferramentas possam identifica-los e processa-los facilmente. No entanto esse novo recurso de Java impactou na modificação da estrutura do bytecode, portanto a atual versão do JMinimizer e também BCEL é imcompatível com esta versão da plataforma. Assim que BCEL for compatível com a nova plataforma será iniciado um esforço para, também, compatibilizar o JMinimizer a plataforma 5.0 do J2SE. Como foi adquirido grande conhecimento da biblioteca BCEL, tanto nas ferramentas como até mesmo no código fonte, me disponibilizei para ajudar na compatibilização do BCEL ao Java 5.0.

7.6 Testar diferentes compiladores

Atualmente os testes feitos com o JMinimizer foram realizados com o bytecode gerado por somente dois compiladores. O compilador fornecido pela Sun[sun 04] e o compilador embutido na IDE Eclipse[ecl 03]. No entanto existem muitos compiladores para a linguagem Java e é de extrema necessidade que para todos os compiladores que um desenvolvedor possa utilizar que o JMinimizer analise e transforme o bytecode de forma correta. Para isso há a necessidade de testar e homologar o JMinimizer com estes compiladores.

7.7 Geração de relatórios

A geração de relatórios é algo que está parcialmente implementado, visto que a geração de arquivos XML com os métodos e campos que são excluidos já é feita. No entanto

[1]Java 2 Standart Edition

arquivos XML são de difícil interpretação por humanos. Arquivos PDF[2] e HTML[3] são de melhor entendimento e para a geração destes arquivos basta uma simples transformação de um arquivos XML com uma folha de estilo XSL[4].

7.8 Desenvolver uma interface gráfica

O desenvolvimento de uma interface gráfica para o JMinimizer também esta planejado para ser desenvolvido, no entanto sua prioridade é baixa. Na interface gráfica seriam criados *wizards* para configurar o JMinimizer, para gerar relatórios em diversos formatos (PDF, HTML). A execução do JMinimizer também poderia ser feita através dessa interface, possibilitando até uma barra de progresso indicando as etapas de execução do JMinimizer.

[2]Portable Document Format
[3]HyperText Mark-up Language
[4]eXtensible Stylesheet Language

Capítulo 8

Metodologia

Desde que comecei a trabalhar como desenvolvedor Java, tive contato com inúmeros projetos *open-source*[1] e dentre estes observei que alguns utilizavam a métodologia de desenvolvimento de software chamada de XP[2], que me chamou a atenção. Como nunca havia desenvolvido software baseado nessa metodologia, resolvi aplica-la, no desenvolvimento do JMinimizer. É claro que não consegui ser um extremista e seguir ao pé da letra o que XP recomenda, mas algumas práticas que achei interessante adotei no desenvolvimento desse trabalho, entre elas estão: a confecção de testes unitários e a liberação de versões num período menor. A criação de testes unitários mostrou ser de grande importância nas etapas de desenvolvimento, principalmente quando as alterações no código fonte eram profundas, pois terminada as alterações eram executados os testes unitários e validados ou não as alterações feitas. Já a liberação de versões num período menor foi pouco utilizado, pois como não haviam "clientes"esperando pelo software não foi preciso liberar versões, no entanto quando foi obtido uma versão estável do JMinimizer, foram gerados marcos no controlador de versão do projeto, para que posteriormente essas versões estáveis pudessem ser obtidas, mesmo depois de alterados os códigos fontes do programa. Também inspirado nos projeos *open-source* e na minha vida profissional adotei ferramentas de desenvolvimento que se tornaram padrões na confecção de aplicações Java. Dentre todas, 6 programas foram essenciais. Inicando pelo sistema operacinal *Fedora Core 1 e 2*[fed 04], que é uma distribuição do sistema operacional Linux. O conjunto de ferramentas distribuídas pela Sun[sun 04] para desenvolvimento de aplicações Java chamando de J2SE SDK[3] também foi utilizado nas suas versões

---

[1]do inglês código-aberto

[2]eXtreme Programming

[3]Software Development Kit

1.4.2_02 e 1.4.2_4. A IDE[4] Eclipse[ecl 03] foi o ambiente de desenvolvimento utilizado para criar o JMinimizer, seu compilador embutido para Java também foi instrumento de uso dessa tese. O controlador de versão CVS[5][cvs 04], foi utilizado para manter de forma organizada as versões que eram geradas dia após dia do código fonte da aplicação. A versão 1.11.1p1 foi utilizado no servidor do CVS e as versão 1.11.15, 1.11.16 e 1.11.17 no cliente. Integrando todas essas ferramentas foi utilizado o Maven [mav 04] para gerenciar o projeto. Tarefas como compilar, executar os testes unitários, gerar documentação, métricas, relatórios, criar artefatos, gereciar o controlador de versão eram executadas através da ferramenta Maven que provê uma interface única para execução de todas essas tarefas.

[4]Integrated Development Environment

[5]Concurrent Versions System

Capítulo 9

Conclusão

O desenvolvimento de aplicações baseadas em bibliotecas de terceiros é ivariavelmente uma solução inteligente quando não queremos ou podemos desenvolver infra estrutura para nossa aplicação. Mas o impacto do tamanho da biblioteca no tamanho final da aplicação pode inviabilizar o projeto. A tese aqui desenvolvida criou uma ferramenta capaz de minimizar o impacto do uso de bibliotecas. Analisando o código estaticamente e removendo estruturas inúteis a aplicação oriundas dessas bibliotecas. Além disso essa ferramenta foi desenvolvida visando ser um projeto open-source, e atualmente está disponível em https://jminimizer.dev.java.net/.

Capítulo 10

Código Fonte

```
package net.java.dev.jminimizer.beans;

import org.apache.bcel.Repository;
import org.apache.bcel.classfile.Utility;
import org.apache.bcel.generic.ClassGen;
import org.apache.bcel.generic.MethodGen;
import org.apache.bcel.generic.Type;

/**
 * @author Thiago ãLeo Moreira <thiagolm@dev.java.net>
 *
 */
public class Method extends FieldOrMethod {
  /**
   * @param className
   * @param name
   * @param signature
   * @return
   */
  public static String toPattern(String className, String name,
      String signature) {
    StringBuffer buffer = new StringBuffer();
    buffer.append(className);
    buffer.append('.');
    buffer.append(name);
    buffer.append(signature);
    return buffer.toString();
  }

  private MethodGen method;

  /**
   * @param className
   * @param name
```

```
 * @param signature
 */
public Method(String className, String name, String signature) {
  super(className, name, signature);
}

/**
 * @param className
 * @param name
 * @param argumentClasses
 * @param returnClass
 */
public Method(String className, String name, String[] argumentClasses,
    String returnClass) {
  this(className, name, Utility.methodTypeToSignature(returnClass,
      argumentClasses));
}

/**
 * @return @throws
 *          ClassNotFoundException
 */
public org.apache.bcel.classfile.Method toClassFileMethod()
    throws ClassNotFoundException {
  ClassGen clazz = new ClassGen(Repository.lookupClass(className));
  //TODO lancar çãexeceo que ãno achou o metodo
  return clazz.containsMethod(name, signature);
}

/**
 * @return @throws
 *          ClassNotFoundException
 */
public MethodGen toMethodGen() throws ClassNotFoundException {
  if (method == null) {
    ClassGen clazz = new ClassGen(Repository.lookupClass(className));
    method = new MethodGen(this.toClassFileMethod(), className, clazz
        .getConstantPool());
  }
  return method;
}

/**
 * @return
 */
public String toPattern() {
  return Method.toPattern(this.getClassName(), this.getName(), this
      .getSignature());
}
```

```
  /**
   * @see java.lang.Object#toString()
   */
  public String toString() {
    StringBuffer buffer = new StringBuffer();
    Type ret = Type.getReturnType(this.getSignature());
    buffer.append(ret);
    buffer.append(' ');
    buffer.append(className);
    buffer.append('.');
    buffer.append(name);
    buffer.append('(');
    Type[] args = Type.getArgumentTypes(this.getSignature());
    for (int i = 0; i < args.length; i++) {
      buffer.append(args[i]);
      if (i != args.length - 1) {
        buffer.append(", ");
      }
    }
    buffer.append(')');
    return buffer.toString();
  }

}

package net.java.dev.jminimizer.beans;

import org.apache.bcel.generic.Type;

/**
 * @author Thiago ãLeo Moreira <thiagolm@dev.java.net>
 * @since Apr 13, 2004
 *
 */
public class Constructor extends Method {
  /**
   * @param className
   * @param signature
   */
  public Constructor(String className, String signature) {
    super(className, "<init>", signature);
  }

  /**
   * @param className
   * @param argumentClasses
   */
  public Constructor(String className, String[] argumentClasses) {
    super(className, "<init>", argumentClasses, "void");
```

```
  }

  /**
   * @see java.lang.Object#toString()
   */
  public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("new ");
    buffer.append(this.getClassName());
    buffer.append('(');
    Type[] args = Type.getArgumentTypes(this.getSignature());
    for (int i = 0; i < args.length; i++) {
      buffer.append(args[i]);
      buffer.append(", ");
    }
    if (args.length != 0) {
      buffer.delete(buffer.length() - 2, buffer.length());
    }
    buffer.append(')');
    return buffer.toString();
  }
}

package net.java.dev.jminimizer.beans;

import org.apache.bcel.generic.Type;

/**
 * @author Thiago ãLeo Moreira <thiagolm@dev.java.net>
 *
 */
public class Field extends FieldOrMethod {
  /**
   * @param className
   * @param name
   * @param signature
   */
  public Field(String className, String name, String signature) {
    super(className, name, signature);
  }

  /**
   * @see java.lang.Object#toString()
   */
  public String toString() {
    StringBuffer buffer = new StringBuffer();
    Type ret = Type.getReturnType(this.getSignature());
    buffer.append(ret);
    buffer.append(' ');
    buffer.append(this.getClassName());
```

```
    buffer.append('.');
    buffer.append(this.getName());
    return buffer.toString();
  }
}

package net.java.dev.jminimizer.beans;

import java.util.Comparator;

/**
 * @author Thiago ãLeo Moreira <thiago.leao.moreira@terra.com.br>
 *
 */
public abstract class FieldOrMethod {

  public static final Comparator COMPARATOR = new Comparator() {

    /**
     * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
     */
    public int compare(Object o1, Object o2) {
      FieldOrMethod fm1 = (FieldOrMethod) o1;
      FieldOrMethod fm2 = (FieldOrMethod) o2;
      return fm1.toCompare().compareTo(fm2.toCompare());
    }

  };

  protected String className;

  protected String name;

  protected String signature;

  /**
   *
   */
  public FieldOrMethod(String className, String name, String signature) {
    super();
    this.setClassName(className);
    this.setName(name);
    this.setSignature(signature);
  }

  /**
   * Returns <code>true</code> if this <code>FieldOrMethod</code> is the
   * same as the o argument.
   *
   * @return <code>true</code> if this <code>FieldOrMethod</code> is the
```

```
 *          same as the o argument.
 */
public boolean equals(Object o) {
  if (this == o) {
    return true;
  }
  if (o == null) {
    return false;
  }
  if (!(o instanceof FieldOrMethod)) {
    return false;
  }

  FieldOrMethod castedObj = (FieldOrMethod) o;
  return ((this.className == null ? castedObj.className == null
      : this.className.equals(castedObj.className))
      && (this.name == null ? castedObj.name == null : this.name
          .equals(castedObj.name)) && (this.signature == null ? castedObj.signature == null
      : this.signature.equals(castedObj.signature)));
}

/**
 * @return
 */
public String getClassName() {
  return className;
}

/**
 * @return
 */
public String getName() {
  return name;
}

/**
 * @return
 */
public String getSignature() {
  return signature;
}

/**
 * @see java.lang.Object#hashCode()
 */
public int hashCode() {
  return className.hashCode() ^ name.hashCode() ^ signature.hashCode();
}
```

```
  /**
   * @param string
   */
  public void setClassName(String string) {
    className = string;
  }

  /**
   * @param string
   */
  public void setName(String string) {
    name = string;
  }

  /**
   * @param string
   */
  public void setSignature(String string) {
    signature = string;
  }

  private String toCompare() {
    return className + " " + name + signature;
  }
}

package net.java.dev.jminimizer.util;

/**
 * @author Thiago ãLeo Moreira <thiago.leao.moreira@terra.com.br>
 * @since Apr 15, 2004
 *
 */
public class DisplayVisitor implements Visitor {
  /**
   *
   */
  public DisplayVisitor() {
    super();
  }

  /**
   * @see net.java.dev.jminimizer.util.Visitor#visit(net.java.dev.jminimizer.beans.Class)
   */
  public void visit(String className) {
    System.out.println(className.toString());
    System.out.println();
    System.out.println();
  }
```

```
  /**
   * @see net.java.dev.jminimizer.util.Visitor#finish()
   */
  public void finish() throws Exception {
    System.out.println("DisplayVisitor.finish");
  }
}

package net.java.dev.jminimizer.util;

import java.util.Set;

/**
 * @author Thiago ãLeo Moreira
 * @since Apr 16, 2004
 *
 */
public interface Repository extends org.apache.bcel.util.Repository {

  /**
   * Build a set with all classes that composite the program.
   *
   * @return
   */
  public Set getProgramClasses();

  /**
   * Build a set with all resources that composite the program.
   *
   * @return
   */
  public Set getProgramResources();

}

package net.java.dev.jminimizer.util;

import java.util.AbstractSet;
import java.util.Iterator;
import java.util.LinkedList;

/**
 * @author Thiago ãLeo Moreira <thiago.ãleo.moreira@terra.com.br>
 *
 */
public class InstructionSet extends AbstractSet {

  protected LinkedList list;

  /**
```

```
   *
   */
  public InstructionSet() {
    super();
    list = new LinkedList();
  }

  /**
   * @see java.util.Collection#size()
   */
  public int size() {
    return list.size();
  }

  /**
   * @see java.util.Collection#iterator()
   */
  public Iterator iterator() {
    return list.iterator();
  }

  /**
   * @see java.util.Collection#add(java.lang.Object)
   */
  public boolean add(Object o) {
    if (list.contains(o)) {
      return false;
    } else {
      list.add(o);
      return true;
    }
  }

}

package net.java.dev.jminimizer.util;

import java.io.File;
import java.net.URL;

import javax.xml.transform.Source;

import net.java.dev.jminimizer.beans.Method;

/**
 * @author Thiago ãLeo Moreira <thiago.leao.moreira@terra.com.br>
 *
 */
public interface Configurator {
```

```
/**
 * Returns a list of methods that must be inspect.
 *
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 * @return a array with methods that must be inspect.
 */
public Method[] getMethodsToInspect();

/**
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 *
 */
public URL[] getProgramClasspath();

/**
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 *
 */
public File getReportDirectory();

/**
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 *
 */
public Source getReportStyleSheet();

/**
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 *
 */
public URL[] getRuntimeClasspath();

/**
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 *
 */
public File getTransformationOutput();

/**
 * @author Thiago ãLeo Moreira
 * @since May 11, 2004
 * @return
 */
```

```
  public boolean inspect(Method method);

  /**
   * @author Thiago ãLeo Moreira
   * @since May 11, 2004
   *
   */
  public boolean inspect(String className);

  /**
   * @author Thiago ãLeo Moreira
   * @since Aug 04, 2004
   * @return
   */
  public boolean isDeepStripment();

}

package net.java.dev.jminimizer.util;

/**
 * @author Thiago ãLeo Moreira <thiago.leao.moreira@terra.com.br>
 *
 */
public interface Visitor {

  public void visit(String className) throws Exception;

  public void finish() throws Exception;

}

package net.java.dev.jminimizer.util;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.JarURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
```

```
import java.util.TreeSet;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

import org.apache.bcel.classfile.ClassParser;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.util.ClassPath;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * @author Thiago ãLeo Moreira
 * @since Apr 15, 2004
 *
 */
public class URLRepository implements Repository {
  private static final Log log = LogFactory.getLog(Repository.class);

  private URL[] pc;

  private Map programClasses;

  private URLClassLoader rc;

  private Map runtimeClasses;

  private Set programResources;

  /**
   * @param parent
   */
  public URLRepository(URL[] program, URL[] runtime) {
    this.rc = new URLClassLoader(runtime);
    this.pc = program;
    programClasses = new HashMap();
    programResources = new HashSet();
    runtimeClasses = new HashMap();
  }

  /**
   * @see org.apache.bcel.util.Repository#clear()
   */
  public void clear() {
    programClasses.clear();
    programResources.clear();
    runtimeClasses.clear();
  }

  /**
```

```
 * @see org.apache.bcel.util.Repository#findClass(java.lang.String)
 */
public JavaClass findClass(String className) {
  JavaClass jc = (JavaClass) programClasses.get(className);
  if (jc == null) {
    jc = (JavaClass) runtimeClasses.get(className);
  }
  return jc;
}

/**
 *
 */
private List findClassFromDirectory(File directory)
    throws MalformedURLException {
  List list = new LinkedList();
  if (directory.exists()) {
    File[] files = directory.listFiles();
    for (int i = 0; i < files.length; i++) {
      if (files[i].isDirectory()) {
        list.addAll(this.findClassFromDirectory(files[i]));
      } else {
        if (files[i].getName().endsWith(".class")) {
          //this is used to get classes that is compiled to be a
          // STUB
          if (files[i].getName().endsWith("_Stub.class")) {
            //this is a resource and don't have to be processed
            log.warn("Stubs: " + files[i]);
            programResources.add(files[i].toURL());
          } else {
            list.add(files[i]);
          }
        } else {
          programResources.add(files[i].toURL());
        }
      }
    }
  }
  return list;
}

/**
 *
 */
private List findClassFromJar(URL jar) throws IOException {
  List list = new LinkedList();
  JarURLConnection con = (JarURLConnection) jar.openConnection();
  JarFile file = con.getJarFile();
  Enumeration e = file.entries();
```

```
  while (e.hasMoreElements()) {
    JarEntry entry = (JarEntry) e.nextElement();
    if (entry.getName().endsWith(".class")) {
      //this is used to get classes that is compiled to be a STUB
      if (entry.getName().endsWith("_Stub.class")) {
        //this is a resource and don't have to be processed
        log.warn("Stubs: " + entry);
        programResources.add(new URL(jar, entry.getName()));
      } else {
        list.add(entry);
      }
    } else {
      if (!entry.isDirectory()) {
        programResources.add(new URL(jar, entry.getName()));
      }
    }
  }
  return list;
}

/**
 * @see org.apache.bcel.util.Repository#getClassPath()
 */
public ClassPath getClassPath() {
  return null;
}

/**
 *
 */
public Set getProgramClasses() {
  for (int i = 0; i < pc.length; i++) {
    String file = pc[i].getFile();
    String protocol = pc[i].getProtocol();
    if (protocol.equals("file")) {
      loadClassFromDirectory(new File(file));
    } else if (protocol.equals("jar")) {
      if (!file.endsWith("!/")) {
        throw new IllegalArgumentException(pc[i]
            + " must point to classpath and end with !/");
      }
      loadClassFromJar(pc[i]);
    }
  }
  Set classes = new TreeSet(String.CASE_INSENSITIVE_ORDER);
  classes.addAll(programClasses.keySet());
  return classes;
}
```

```
/**
 * @see org.apache.bcel.util.Repository#loadClass(java.lang.Class)
 */
public JavaClass loadClass(Class clazz) throws ClassNotFoundException {
  return this.loadClass(clazz.getName());
}

/**
 * @see org.apache.bcel.util.Repository#loadClass(java.lang.String)
 */
public JavaClass loadClass(String className) throws ClassNotFoundException {
  JavaClass jc = (JavaClass) this.findClass(className);
  if (jc == null) {
    jc = this.loadProgramClass(className);
    if (jc == null) {
      jc = this.loadRuntimeClass(className);
    }
  }
  return jc;
}

/**
 *
 * @param directory
 */
private void loadClassFromDirectory(File directory) {
  try {
    List classes = this.findClassFromDirectory(directory);
    Iterator i = classes.iterator();
    int start = directory.getAbsolutePath().length();
    while (i.hasNext()) {
      File file = (File) i.next();
      //extract the directory
      String clazz = file.getAbsolutePath().substring(start + 1);
      clazz = this.normalizeClass(clazz);
      if (!programClasses.containsKey(clazz)) {
        programClasses.put(clazz, this.parseClass(
            new FileInputStream(file), clazz));
      }
    }
  } catch (IOException e) {
    log.error("Never should be here!!!", e);
  }
}

/**
 *
 * @param jar
 */
```

```
private void loadClassFromJar(URL jar) {
  try {
    List list = this.findClassFromJar(jar);
    Iterator i = list.iterator();
    while (i.hasNext()) {
      JarEntry entry = (JarEntry) i.next();
      String clazz = this.normalizeClass(entry.getName());
      if (!programClasses.containsKey(clazz)) {
        programClasses.put(clazz, this.parseClass(new URL(jar,
            entry.getName()).openStream(), clazz));
      }
    }
  } catch (IOException e) {
    log.error("Never should be here!!!", e);
  }
}

/**
 *
 * @param className
 * @return @throws
 *         ClassNotFoundException
 */
private JavaClass loadProgramClass(String className)
    throws ClassNotFoundException {
  //program classes
  JavaClass jc = null;
  for (int i = 0; i < pc.length; i++) {
    URL url = null;
    String path = className;
    try {
      if (pc[i].getProtocol().equals("jar")) {
        path = path.replace('.', '/').concat(".class");
      } else {
        path = path.replace('.', File.separatorChar).concat(
            ".class");
      }
      url = new URL(pc[i], path);
    } catch (MalformedURLException e) {
      log.debug("Error on creating the URL", e);
      continue;
    }
    byte[] data;
    try {
      jc = this.parseClass(url.openStream(), className);
      programClasses.put(className, jc);
      break;
    } catch (IOException e) {
      log.debug("Error on reading the URL", e);
```

```
        continue;
      }
    }
    return jc;
  }

  /**
   *
   * @param className
   * @return @throws
   *         ClassNotFoundException
   */
  private JavaClass loadRuntimeClass(String className)
      throws ClassNotFoundException {
    //program classes
    JavaClass jc = null;
    //URL url= rc.getResource(className.replace('.',
    // '/').concat(".class"));
    InputStream in = rc.getResourceAsStream(className.replace('.', '/')
        .concat(".class"));
    if (in == null) {
      throw new ClassNotFoundException(className);
    }
    try {
      jc = this.parseClass(in, className);
      runtimeClasses.put(className, jc);
    } catch (IOException e) {
      log.debug("Error on reading the URL", e);
    }
    return jc;
  }

  /**
   *
   * @param clazz
   * @return
   */
  private String normalizeClass(String clazz) {
    //replace file separator per dot
    clazz = clazz.replace(File.separatorChar, '.');
    //remove from end the termination ".class"
    return clazz.substring(0, clazz.length() - 6);
  }

  /**
   *
   * @param in
   * @param className
   * @return @throws
```

```
   *          IOException
   */
  private JavaClass parseClass(InputStream in, String className)
      throws IOException {
    ClassParser parser = new ClassParser(in, className);
    JavaClass jc = parser.parse();
    jc.setRepository(this);
    return jc;
  }

  /**
   * @see org.apache.bcel.util.Repository#removeClass(org.apache.bcel.classfile.JavaClass)
   */
  public void removeClass(JavaClass clazz) {
    programClasses.remove(clazz.getClassName());
    runtimeClasses.remove(clazz.getClassName());
  }

  /**
   * @see org.apache.bcel.util.Repository#storeClass(org.apache.bcel.classfile.JavaClass)
   */
  public void storeClass(JavaClass clazz) {
    throw new RuntimeException("No sense in this class!!");
  }

  /**
   * @see net.java.dev.jminimizer.util.Repository#getProgramResources()
   */
  public Set getProgramResources() {
    return programResources;
  }
}

package net.java.dev.jminimizer.util;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;

/**
 * @author Thiago ãLeo Moreira
 * @since Apr 21, 2004
 *
 */
public class XMLErrorHandler extends DefaultHandler {

  private static final Log log = LogFactory.getLog(XMLErrorHandler.class);
```

```
/**
 * @see org.xml.sax.ErrorHandler#error(org.xml.sax.SAXParseException)
 */
public void error(SAXParseException exception) throws SAXException {
  String id = exception.getPublicId();
  if (id == null) {
    id = exception.getSystemId();
  }
  String location = "The xml file: " + id + " has a error at line: "
      + exception.getLineNumber() + " and column: "
      + exception.getColumnNumber() + " the error message is:\n";
  location += exception.getMessage();
  log.error(location);
  System.exit(0);
}

/**
 * @see org.xml.sax.ErrorHandler#fatalError(org.xml.sax.SAXParseException)
 */
public void fatalError(SAXParseException exception) throws SAXException {
  String id = exception.getPublicId();
  if (id == null) {
    id = exception.getSystemId();
  }
  String location = "The xml file: " + id
      + " has a fatal error at line: " + exception.getLineNumber()
      + " and column: " + exception.getColumnNumber()
      + " the error message is:\n";
  location += exception.getMessage();
  log.fatal(location, exception);
  System.exit(0);
}

/**
 * @see org.xml.sax.ErrorHandler#warning(org.xml.sax.SAXParseException)
 */
public void warning(SAXParseException exception) throws SAXException {
  String id = exception.getPublicId();
  if (id == null) {
    id = exception.getSystemId();
  }
  String location = "The xml file: " + id + " has a warning at line: "
      + exception.getLineNumber() + " and column: "
      + exception.getColumnNumber() + " the error message is:\n";
  location += exception.getMessage();
  log.warn(location);
}
```

```
  /**
   * @see org.xml.sax.EntityResolver#resolveEntity(java.lang.String,
   *      java.lang.String)
   */
  public InputSource resolveEntity(String publicId, String systemId)
      throws SAXException {
    return new InputSource(this.getClass().getResourceAsStream(
        "/resources/configuration.dtd"));
  }
}

package net.java.dev.jminimizer.util;

import java.io.File;
import java.net.URL;

import org.apache.bcel.Repository;

/**
 * @author Thiago ãLeo Moreira
 * @since Aug 3, 2004
 *
 */
public class Verifier {

  public static void main(String[] args) throws Exception {
    URL url;
    File file = new File(args[0]);
    if (file.isFile()) {
      url = new URL("jar:file:" + args[0] + "!/");
    } else {
      url = file.getAbsoluteFile().toURL();
    }
    Repository.setRepository(new URLRepository(new URL[] { url },
        new URL[0]));
    String[] argsTemp = new String[args.length - 1];
    for (int i = 0; i < argsTemp.length; i++) {
      argsTemp[i] = args[i + 1];
    }
    org.apache.bcel.verifier.Verifier.main(argsTemp);
  }

}

package net.java.dev.jminimizer.util;

import java.util.Set;

import net.java.dev.jminimizer.beans.Method;
```

```
import org.apache.bcel.Repository;
import org.apache.bcel.generic.ClassGen;
import org.apache.bcel.generic.FieldGen;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * @author Thiago ãLeo Moreira
 * @since Apr 13, 2004
 *
 */
public class ClassUtils {

  private static final Log log = LogFactory.getLog(ClassUtils.class);

  public static Method findMethod(Configurator configurator,
      Set classesUsedByProgram, String className, String name,
      String signature) throws ClassNotFoundException {
    ClassGen clazz = new ClassGen(Repository.lookupClass(className));
    org.apache.bcel.classfile.Method method;
    Method m = null;
    do {
      className = clazz.getClassName();
      if (configurator.inspect(className)
          && classesUsedByProgram.add(className)) {
        //used just to relax de user
        log.info("Analysing class: " + className);
      }
      method = clazz.containsMethod(name, signature);
      if (method != null) {
        m = new Method(clazz.getClassName(), method.getName(), method
            .getSignature());
        log.debug("Method find: " + m);
        return m;
      } else {
        String[] interfaces = clazz.getInterfaceNames();
        for (int i = 0; i < interfaces.length; i++) {
          m = ClassUtils.findMethod(configurator,
              classesUsedByProgram, interfaces[i], name,
              signature);
          if (m != null) {
            log.debug("Method find: " + m);
            return m;
          }
        }
      }
      clazz = new ClassGen(Repository.lookupClass(clazz
          .getSuperclassName()));
    } while (!clazz.getClassName().equals("java.lang.Object"));
```

```
    return m;
  }

  public static FieldGen findField(String className, String name)
      throws ClassNotFoundException {
    ClassGen clazz = new ClassGen(Repository.lookupClass(className));
    org.apache.bcel.classfile.Field field;
    do {
      field = clazz.containsField(name);
      if (field != null) {
        log.debug("Field find: " + clazz.getClassName() + "." + field);
        return new FieldGen(field, clazz.getConstantPool());
      }
      clazz = new ClassGen(Repository.lookupClass(clazz
          .getSuperclassName()));
    } while (clazz != null);
    return null;
  }

  public static String normalize(String string) {
    StringBuffer buffer = new StringBuffer(string);
    char[] chs = { '.', '(', ')', '$' };
    for (int i = 0; i < chs.length; i++) {
      ClassUtils.normalize(chs[i], buffer);
    }
    return buffer.toString().replace('*', '.');
  }

  private static void normalize(char c, StringBuffer buffer) {
    String ch = "" + c;
    int i = buffer.indexOf(ch);
    while (i != -1) {
      buffer.insert(i, '\\');
      i = buffer.indexOf(ch, i + 2);
    }
  }

}

package net.java.dev.jminimizer.util;

import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
```

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;

import net.java.dev.jminimizer.beans.Constructor;
import net.java.dev.jminimizer.beans.Method;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.regexp.RE;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.helpers.DefaultHandler;

/**
 * @author Thiago ãLeo Moreira <thiago.leao.moreira@terra.com.br>
 * @since May 11, 2004
 */
public class XMLConfigurator implements Configurator {

  public static final String ELEMENT_DIRECTORY = "directory";

  public static final String ELEMENT_FILE = "file";

  public static final String ELEMENT_FILESET = "fileset";

  public static final String ELEMENT_PATTERN = "pattern";

  public static final String ELEMENT_PROGRAM_CLASSPATH = "programClasspath";

  public static final String ELEMENT_RUNTIME_CLASSPATH = "runtimeClasspath";

  public static final String ELEMENT_URL = "url";

  private static final Log log = LogFactory.getLog(XMLConfigurator.class);

  private boolean deepStripment;

  private Set methods;

  private Map primitives;

  private Set programClasspath;

  private RE reNotInspect;

  private File reportDiretory;
```

```
private Source resportStyleSheet;

private Set runtimeClasspath;

private File transformationDiretory;

/**
 *
 */
public XMLConfigurator(File file) throws Exception {
  super();
  methods = new HashSet();
  this.initPrimitives();
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setValidating(true);
  DocumentBuilder builder = factory.newDocumentBuilder();
  DefaultHandler handler = new XMLErrorHandler();
  builder.setErrorHandler(handler);
  builder.setEntityResolver(handler);
  Document document = builder.parse(file);
  this.normalize(document);
  Element root = document.getDocumentElement();
  String temp = root.getAttribute("deepStripment");
  if (temp != null) {
    deepStripment = temp.equals("true");
  } else {
    deepStripment = false;
  }
  programClasspath = this.buildClasspath(root,
      XMLConfigurator.ELEMENT_PROGRAM_CLASSPATH);
  runtimeClasspath = this.buildClasspath(root,
      XMLConfigurator.ELEMENT_RUNTIME_CLASSPATH);
  methods = this.buildMethodsToInspect(root
      .getElementsByTagName("inspect"));
  reNotInspect = this.buildNotInspectPattern(root
      .getElementsByTagName("notInspect"));
  transformationDiretory = this.buildTransformationOutput((Element) root
      .getElementsByTagName("output").item(0));
  reportDiretory = this.buildReportOutput((Element) root
      .getElementsByTagName("report").item(0));
}

private File buildReportOutput(Element eOutput) {
  File file;
  if (eOutput == null) {
    file = new File("report");
  } else {
    String path = ((Element) eOutput.getFirstChild())
```

```
        .getAttribute("path");
    file = new File(path);
  }
  file.mkdirs();
  return file;
}

/**
 *
 * @param eOutput
 * @return
 */
private File buildTransformationOutput(Element eOutput) throws IOException {
  File output;
  if (eOutput == null) {
    output = new File("out.jar");
    if (!output.exists()) {
      output.createNewFile();
    }
  } else {
    Element element = (Element) eOutput.getFirstChild();
    if (element.getNodeName().equals("file")) {
      output = new File(element.getAttribute("name"));
      if (!output.exists()) {
        output.createNewFile();
      }
    } else {
      output = new File(element.getAttribute("path"));
      if (!output.exists()) {
        output.mkdirs();
      }
    }
  }
  return output;
}

/**
 *
 * @param root
 * @param elementName
 * @return
 */
private Set buildClasspath(Element root, String elementName) {
  Set classpath = new HashSet();
  NodeList list = root.getElementsByTagName(elementName);
  for (int i = 0; i < list.getLength(); i++) {
    Element eClasspath = (Element) list.item(i);
    classpath.addAll(this.buildURLs(eClasspath
        .getElementsByTagName(XMLConfigurator.ELEMENT_URL)));
```

```
    classpath.addAll(this.buildDirectorys(eClasspath
        .getElementsByTagName(XMLConfigurator.ELEMENT_DIRECTORY)));
    classpath.addAll(this.buildFileset(eClasspath
        .getElementsByTagName(XMLConfigurator.ELEMENT_FILESET)));
  }
  return classpath;
}

/**
 * @param className
 * @param constructorElement
 */
private Method buildConstructor(String className, Element constructorElement) {
  String[] args = this.getArgumentClasses(constructorElement
      .getFirstChild().getChildNodes());
  Constructor m = new Constructor(className, args);
  log.debug("Constructor builded: " + m);
  return m;
}

/**
 * @param lDirectory
 * @return
 */
private Set buildDirectorys(NodeList lDirectory) {
  Set directories = new HashSet();
  for (int i = 0; i < lDirectory.getLength(); i++) {
    Element eDirectory = (Element) lDirectory.item(i);
    File directory = new File(eDirectory.getAttribute("path"));
    if (directory.exists()) {
      try {
        directories.add(directory.toURL());
      } catch (MalformedURLException e) {
        log.error("Error creating url: " + directory, e);
      }
    } else {
      log.warn("The directory not exist: " + directory
          + " and don't was add to classpath!");
    }
  }
  return directories;
}

private Set buildFileset(NodeList lFileset) {
  Set files = new HashSet();
  for (int i = 0; i < lFileset.getLength(); i++) {
    Element eFileset = (Element) lFileset.item(i);
    File directory = new File(eFileset.getAttribute("directory"));
    if (directory.exists()) {
```

```
        NodeList lFile = eFileset.getChildNodes();
        for (int j = 0; j < lFile.getLength(); j++) {
          Element eFile = (Element) lFile.item(j);
          File file = new File(directory, eFile.getAttribute("name"));
          if (file.exists()) {
            try {
              files.add(new URL("jar:" + file.toURL() + "!/"));
            } catch (MalformedURLException e) {
              log.error("Error creating url: " + file, e);
            }
          } else {
            log.warn("The file don't exist: " + file
                + " and don't was add to classpath!");
          }
        }
      } else {
        log.warn("The directory not exist: " + directory
            + " and don't was add to classpath!");
      }
    }
    return files;
  }

  /**
   * @param className
   * @param methodElement
   * @return @throws
   *         ClassNotFoundException
   */
  private Method buildMethod(String className, Element methodElement) {
    Node argumentsElement = methodElement.getFirstChild();
    Element returnElement = (Element) argumentsElement.getNextSibling()
        .getFirstChild();
    String[] args = this.getArgumentClasses(argumentsElement
        .getChildNodes());
    String ret = this.getReturnClass(returnElement);
    Method m = new Method(className, methodElement.getAttribute("name"),
        args, ret);
    log.debug("Method builded: " + m);
    return m;
  }

  /**
   * @param root
   */
  private Set buildMethodsToInspect(NodeList lInspect) {
    Set methods = new HashSet();
    for (int i = 0; i < lInspect.getLength(); i++) {
      Element eInspect = (Element) lInspect.item(i);
```

```
      NodeList lClass = eInspect.getChildNodes();
      for (int j = 0; j < lClass.getLength(); j++) {
        Element eClass = (Element) lClass.item(j);
        String className = eClass.getAttribute("name");
        NodeList lConstructors = eClass
            .getElementsByTagName("constructor");
        for (int k = 0; k < lConstructors.getLength(); k++) {
          Element eConstructor = (Element) lConstructors.item(k);
          methods.add(this.buildConstructor(className, eConstructor));
        }
        NodeList lMethods = eClass.getElementsByTagName("method");
        for (int k = 0; k < lMethods.getLength(); k++) {
          Element eMethod = (Element) lMethods.item(k);
          methods.add(this.buildMethod(className, eMethod));
        }
      }
    }
    return methods;
  }

  /**
   * @param element
   */
  private RE buildNotInspectPattern(NodeList lNotInspect) {
    StringBuffer buffer = new StringBuffer();
    for (int i = 0; i < lNotInspect.getLength(); i++) {
      Element eNotInspect = (Element) lNotInspect.item(i);
      NodeList lPattern = eNotInspect.getChildNodes();
      for (int j = 0; j < lPattern.getLength(); j++) {
        String temp = lPattern.item(j).getFirstChild().getNodeValue()
            .trim();
        if (temp.startsWith("*") && temp.endsWith("*")) {
          buffer.append(ClassUtils.normalize(temp));
        } else if (temp.startsWith("*")) {
          buffer.append(ClassUtils.normalize(temp + "$"));
        } else if (temp.endsWith("*")) {
          buffer.append(ClassUtils.normalize("^" + temp));
        }
        buffer.append('|');
      }
    }
    if (buffer.length() != 0) {
      buffer.deleteCharAt(buffer.length() - 1);
    } else {
      buffer.append("\\b9876546321\\b");
    }
    log.debug("Pattern compiled: " + buffer.toString());
    return new RE(buffer.toString());
  }
```

```
/**
 * @param lURL
 */
private Set buildURLs(NodeList lURL) {
  Set urls = new HashSet();
  for (int i = 0; i < lURL.getLength(); i++) {
    Node nURL = lURL.item(i);
    String url = nURL.getFirstChild().getNodeValue();
    try {
      urls.add(new URL(url));
    } catch (MalformedURLException e) {
      log.error("Error creating url: " + url, e);
    }
  }
  return urls;
}

/**
 * @param list
 * @return @throws
 *          ClassNotFoundException
 */
private String[] getArgumentClasses(NodeList list) {
  return this.getClasses(list);
}

/**
 * @param list
 */
private String[] getClasses(NodeList list) {
  String[] classes = new String[list.getLength()];
  for (int i = 0; i < list.getLength(); i++) {
    Element element = (Element) list.item(i);
    String name = element.getNodeName();
    if (name.equals("primitiveType")) {
      classes[i] = (String) primitives.get(element
          .getAttribute("name"));
    } else {
      classes[i] = element.getAttribute("name");
    }
  }
  return classes;
}

/**
 * @see net.java.dev.jminimizer.util.Configurator#getMethodsToInspect()
 */
public Method[] getMethodsToInspect() {
```

```
    return (Method[]) methods.toArray(new Method[0]);
}

/**
 * @see net.java.dev.jminimizer.util.Configurator#getProgramClasspath()
 */
public URL[] getProgramClasspath() {
    return (URL[]) programClasspath.toArray(new URL[0]);
}

/**
 * @see net.java.dev.jminimizer.util.Configurator#getReportDirectory()
 */
public File getReportDirectory() {
    return reportDiretory;
}

/**
 * @see net.java.dev.jminimizer.util.Configurator#getReportStyleSheet()
 */
public Source getReportStyleSheet() {
    return resportStyleSheet;
}

/**
 * @param element
 */
private String getReturnClass(Element element) {
    String name = element.getNodeName();
    if (name.equals("void")) {
        return Void.TYPE.getName();
    } else if (name.equals("primitiveType")) {
        return (String) primitives.get(element.getAttribute("name"));
    } else {
        return element.getAttribute("name");
    }
}

/**
 * @see net.java.dev.jminimizer.util.Configurator#getRuntimeClasspath()
 */
public URL[] getRuntimeClasspath() {
    return (URL[]) runtimeClasspath.toArray(new URL[0]);
}

/**
 * @see net.java.dev.jminimizer.util.Configurator#getTransformationOutput()
 */
public File getTransformationOutput() {
```

```
    return transformationDiretory;
  }

  /**
   *
   *
   */
  private void initPrimitives() {
    primitives = new HashMap();
    primitives.put("boolean", Boolean.TYPE.getName());
    primitives.put("byte", Byte.TYPE.getName());
    primitives.put("short", Short.TYPE.getName());
    primitives.put("char", Character.TYPE.getName());
    primitives.put("int", Integer.TYPE.getName());
    primitives.put("float", Float.TYPE.getName());
    primitives.put("long", Long.TYPE.getName());
    primitives.put("double", Double.TYPE.getName());
  }

  /**
   * @see net.java.dev.jminimizer.util.Configurator#inspect(net.java.dev.jminimizer.beans.Method)
   *
   */
  public boolean inspect(Method method) {
    return this.inspect(method.toPattern());
  }

  /**
   * @see net.java.dev.jminimizer.util.Configurator#inspect(java.lang.String)
   */
  public boolean inspect(String pattern) {
    boolean b = !reNotInspect.match(pattern);
    log.debug("Inspect " + pattern + " ? = " + pattern);
    return b;
  }

  /**
   * @param node
   */
  private void normalize(Node node) {
    NodeList list = node.getChildNodes();
    for (int i = 0; i < list.getLength();) {
      Node item = list.item(i);
      if ((item.getNodeType() == Node.TEXT_NODE && item.getNodeValue()
          .trim().length() == 0)
          || item.getNodeType() == Node.COMMENT_NODE) {
        node.removeChild(item);
      } else {
        i++;
```

```
        this.normalize(item);
      }
    }
  }

  /**
   * @see net.java.dev.jminimizer.util.Configurator#isDeepStripment()
   */
  public boolean isDeepStripment() {
    return deepStripment;
  }

  public boolean preserveResource(String resource) {
    return true;
  }

}

package net.java.dev.jminimizer;

import java.util.Set;

import org.apache.bcel.Constants;
import org.apache.bcel.classfile.Attribute;
import org.apache.bcel.classfile.Code;
import org.apache.bcel.classfile.CodeException;
import org.apache.bcel.classfile.ConstantClass;
import org.apache.bcel.classfile.ConstantDouble;
import org.apache.bcel.classfile.ConstantFieldref;
import org.apache.bcel.classfile.ConstantFloat;
import org.apache.bcel.classfile.ConstantInteger;
import org.apache.bcel.classfile.ConstantInterfaceMethodref;
import org.apache.bcel.classfile.ConstantLong;
import org.apache.bcel.classfile.ConstantMethodref;
import org.apache.bcel.classfile.ConstantNameAndType;
import org.apache.bcel.classfile.ConstantPool;
import org.apache.bcel.classfile.ConstantString;
import org.apache.bcel.classfile.ConstantUtf8;
import org.apache.bcel.classfile.ConstantValue;
import org.apache.bcel.classfile.Deprecated;
import org.apache.bcel.classfile.ExceptionTable;
import org.apache.bcel.classfile.Field;
import org.apache.bcel.classfile.InnerClass;
import org.apache.bcel.classfile.InnerClasses;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.LineNumber;
import org.apache.bcel.classfile.LineNumberTable;
import org.apache.bcel.classfile.LocalVariable;
import org.apache.bcel.classfile.LocalVariableTable;
import org.apache.bcel.classfile.Method;
```

```
import org.apache.bcel.classfile.Signature;
import org.apache.bcel.classfile.SourceFile;
import org.apache.bcel.classfile.StackMap;
import org.apache.bcel.classfile.StackMapEntry;
import org.apache.bcel.classfile.Synthetic;
import org.apache.bcel.classfile.Unknown;
import org.apache.bcel.generic.ClassGen;
import org.apache.bcel.generic.ConstantPoolGen;
import org.apache.bcel.generic.FieldGen;
import org.apache.bcel.generic.MethodGen;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * @author Thiago ãLeo Moreira
 * @since Jul 8, 2004
 *
 */
public class ConstantPoolCleanerVisitor implements
    org.apache.bcel.classfile.Visitor {

  private static final Log log = LogFactory
      .getLog(ConstantPoolCleanerVisitor.class);

  private ConstantPoolGen newPool;

  private ConstantPoolGen oldPool;

  private Set classesUsedByProgram;

  private boolean deepStripment;

  public ConstantPoolCleanerVisitor(boolean deepStripment,
      Set classesUsedByProgram) {
    newPool = new ConstantPoolGen();
    this.classesUsedByProgram = classesUsedByProgram;
    this.deepStripment = deepStripment;
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitCode(org.apache.bcel.classfile.Code)
   */
  public void visitCode(Code obj) {
    log.trace("Visiting method attribute of type Cope: nothing done!!!");
    throw new RuntimeException(
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitCode(org.apache.bcel.classfile.Code)");
  }
```

```
/**
 * @see org.apache.bcel.classfile.Visitor#visitCodeException(org.apache.bcel.classfile.CodeException)
 */
public void visitCodeException(CodeException obj) {
  log
      .trace("Visiting code attribute of type CopeException: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitCodeException(org.apache.bcel.classfile.CodeException)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantClass(org.apache.bcel.classfile.ConstantClass)
 */
public void visitConstantClass(ConstantClass obj) {
  log.trace("Visiting Constant of type Class: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantClass(org.apache.bcel.classfile.ConstantClass)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantDouble(org.apache.bcel.classfile.ConstantDouble)
 */
public void visitConstantDouble(ConstantDouble obj) {
  log.trace("Visiting Constant of type Double: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantDouble(org.apache.bcel.classfile.ConstantDouble)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantFieldref(org.apache.bcel.classfile.ConstantFieldre
 */
public void visitConstantFieldref(ConstantFieldref obj) {
  log.trace("Visiting Constant of type FieldRef: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantFieldref(org.apache.bcel.classfile.ConstantFieldref)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantFloat(org.apache.bcel.classfile.ConstantFloat)
 */
public void visitConstantFloat(ConstantFloat obj) {
  log.trace("Visiting Constant of type Float: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantFloat(org.apache.bcel.classfile.ConstantFloat)");
```

```
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantInteger(org.apache.bcel.classfile.ConstantInteger)
 */
public void visitConstantInteger(ConstantInteger obj) {
  log.trace("Visiting Constant of type Integer: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantInteger(org.apache.bcel.classfile.ConstantInteger)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantInterfaceMethodref(org.apache.bcel.classfile.Cons
 */
public void visitConstantInterfaceMethodref(ConstantInterfaceMethodref obj) {
  log
      .trace("Visiting Constant of type InterfaceMethodref: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantInterfaceMethodref(org.apache.bcel.classfile.ConstantInterfaceMethodref)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantLong(org.apache.bcel.classfile.ConstantLong)
 */
public void visitConstantLong(ConstantLong obj) {
  log.trace("Visiting Constant of type Long: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantLong(org.apache.bcel.classfile.ConstantLong)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantMethodref(org.apache.bcel.classfile.ConstantMetho
 */
public void visitConstantMethodref(ConstantMethodref obj) {
  log.trace("Visiting Constant of type Methodref: nothing done!!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitConstantMethodref(org.apache.bcel.classfile.ConstantMethodref)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitConstantNameAndType(org.apache.bcel.classfile.ConstantName
 */
public void visitConstantNameAndType(ConstantNameAndType obj) {
  log.trace("Visiting Constant of type NameAndType: nothing done!!!");
  throw new RuntimeException(
```

```
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitConstantNameAndType(org.apache.bcel.classfile.ConstantNameAndType)");
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitConstantPool(org.apache.bcel.classfile.ConstantPool)
   */
  public void visitConstantPool(ConstantPool obj) {
    log.trace("Visiting ConstantPool: nothing done!!!");
    throw new RuntimeException(
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitConstantPool(org.apache.bcel.classfile.ConstantPool)");
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitConstantString(org.apache.bcel.classfile.ConstantString)
   */
  public void visitConstantString(ConstantString obj) {
    log.trace("Visiting Constant of type String: nothing done!!!");
    throw new RuntimeException(
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitConstantString(org.apache.bcel.classfile.ConstantString)");
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitConstantUtf8(org.apache.bcel.classfile.ConstantUtf8)
   */
  public void visitConstantUtf8(ConstantUtf8 obj) {
    log.trace("Visiting Constant of type Utf8: nothing done!!!");
    throw new RuntimeException(
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitConstantUtf8(org.apache.bcel.classfile.ConstantUtf8)");
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitConstantValue(org.apache.bcel.classfile.ConstantValue)
   */
  public void visitConstantValue(ConstantValue obj) {
    log.debug("Visiting field attribute of type ConstantValue");
    obj.setNameIndex(newPool.addConstant(oldPool.getConstant(obj
        .getNameIndex()), oldPool));
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitDeprecated(org.apache.bcel.classfile.Deprecated)
   */
  public void visitDeprecated(Deprecated obj) {
    log.debug("Visiting field/method attribute of type Deprecated");
    obj.setNameIndex(newPool.addConstant(oldPool.getConstant(obj
```

```
        .getNameIndex()), oldPool));
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitExceptionTable(org.apache.bcel.classfile.ExceptionTable)
 */
public void visitExceptionTable(ExceptionTable obj) {
  log.debug("Visiting code attribute of type ExceptionTable");
  obj.setNameIndex(newPool.addConstant(oldPool.getConstant(obj
      .getNameIndex()), oldPool));
  int[] exceptions = obj.getExceptionIndexTable();
  for (int i = 0; i < exceptions.length; i++) {
    exceptions[i] = newPool.addConstant(oldPool
        .getConstant(exceptions[i]), oldPool);
  }
  obj.setExceptionIndexTable(exceptions);
  obj.setConstantPool(newPool.getConstantPool());
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitField(org.apache.bcel.classfile.Field)
 */
public void visitField(Field obj) {
  log.debug("Visiting field");
  obj.setNameIndex(newPool.addConstant(oldPool.getConstant(obj
      .getNameIndex()), oldPool));
  obj.setSignatureIndex(newPool.addConstant(oldPool.getConstant(obj
      .getSignatureIndex()), oldPool));
  Attribute[] attributes = obj.getAttributes();
  for (int i = 0; i < attributes.length; i++) {
    attributes[i].accept(this);
  }
  obj.setConstantPool(newPool.getConstantPool());
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitInnerClass(org.apache.bcel.classfile.InnerClass)
 */
public void visitInnerClass(InnerClass obj) {
  log.debug("Visiting innerClass");
  String className = ((ConstantClass) oldPool.getConstant(obj
      .getInnerClassIndex())).getBytes(oldPool.getConstantPool())
      .replace('/', '.');
  if (classesUsedByProgram.contains(className)) {
    newPool.addConstant(oldPool.getConstant(obj.getInnerClassIndex()),
        oldPool);
    int index = obj.getInnerNameIndex();
    if (index != 0) {
      newPool.addConstant(oldPool.getConstant(index), oldPool);
```

```
      }
      index = obj.getOuterClassIndex();
      if (index != 0) {
        newPool.addConstant(oldPool.getConstant(index), oldPool);
      }
    }
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitInnerClasses(org.apache.bcel.classfile.InnerClasses)
   */
  public void visitInnerClasses(InnerClasses obj) {
    log.debug("Visiting innerClasses");
    int size = newPool.getSize();
    InnerClass[] innerClasss = obj.getInnerClasses();
    for (int i = 0; i < innerClasss.length; i++) {
      innerClasss[i].accept(this);
    }
    if (size != newPool.getSize()) {
      obj
          .setNameIndex(newPool
              .addUtf8(Constants.ATTRIBUTE_NAMES[Constants.ATTR_INNER_CLASSES]));
      obj.setConstantPool(newPool.getConstantPool());
    }
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitJavaClass(org.apache.bcel.classfile.JavaClass)
   */
  public void visitJavaClass(JavaClass obj) {
    log.trace("Visiting JavaClass: nothing done!!");
    throw new RuntimeException(
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitJavaClass(org.apache.bcel.classfile.JavaClass)");
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitLineNumber(org.apache.bcel.classfile.LineNumber)
   */
  public void visitLineNumber(LineNumber obj) {
    log.trace("Visiting code attribute of type LineNumber: nothing done!!");
    throw new RuntimeException(
        "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
        "visitLineNumber(org.apache.bcel.classfile.LineNumber)");
  }

  /**
   * @see org.apache.bcel.classfile.Visitor#visitLineNumberTable(org.apache.bcel.classfile.LineNumberTable)
   */
```

```
public void visitLineNumberTable(LineNumberTable obj) {
  log
      .trace("Visiting code attribute of type LineNumberTable: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitLineNumberTable(org.apache.bcel.classfile.LineNumberTable)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitLocalVariable(org.apache.bcel.classfile.LocalVariable)
 */
public void visitLocalVariable(LocalVariable obj) {
  log
      .trace("Visiting code attribute of type LocalVariable: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitLocalVariable(org.apache.bcel.classfile.LocalVariable)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitLocalVariableTable(org.apache.bcel.classfile.LocalVariable
 */
public void visitLocalVariableTable(LocalVariableTable obj) {
  log
      .trace("Visiting code attribute of type LocalVariableTable: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitLocalVariableTable(org.apache.bcel.classfile.LocalVariableTable)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitMethod(org.apache.bcel.classfile.Method)
 */
public void visitMethod(Method obj) {
  log.debug("Visiting method");
  Attribute[] attributes = obj.getAttributes();
  for (int i = 0; i < attributes.length; i++) {
    if (attributes[i] instanceof Synthetic) {
      attributes[i].accept(this);
    }
    //only for methods that is native or abstract
    if (attributes[i] instanceof ExceptionTable
        && (obj.isAbstract() || obj.isNative())) {
      attributes[i].accept(this);
    }
    if (attributes[i] instanceof Deprecated) {
      attributes[i].accept(this);
    }
  }
```

```
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitSignature(org.apache.bcel.classfile.Signature)
 */
public void visitSignature(Signature obj) {
  log.trace("Visiting signature: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitLocalVariableTable(org.apache.bcel.classfile.LocalVariableTable)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitSourceFile(org.apache.bcel.classfile.SourceFile)
 */
public void visitSourceFile(SourceFile obj) {
  log
      .debug("Visiting field/method's attribute of type SourceFile: nothing done!!");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitSynthetic(org.apache.bcel.classfile.Synthetic)
 */
public void visitSynthetic(Synthetic obj) {
  log.debug("Visiting field/method's attribute of type Synthetic");
  obj.setNameIndex(newPool
      .addUtf8(Constants.ATTRIBUTE_NAMES[Constants.ATTR_SYNTHETIC]));
  obj.setConstantPool(newPool.getConstantPool());
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitUnknown(org.apache.bcel.classfile.Unknown)
 */
public void visitUnknown(Unknown obj) {
  log.trace("Visiting Unknown attribute: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitUnknown(org.apache.bcel.classfile.Unknown)");
}

/**
 * @see org.apache.bcel.classfile.Visitor#visitStackMap(org.apache.bcel.classfile.StackMap)
 */
public void visitStackMap(StackMap obj) {
  log.trace("Visiting code's attribute of type StackMap: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitStackMap(org.apache.bcel.classfile.StackMap)");
}
```

```
/**
 * @see org.apache.bcel.classfile.Visitor#visitStackMapEntry(org.apache.bcel.classfile.StackMapEntry)
 */
public void visitStackMapEntry(StackMapEntry obj) {
  log
      .trace("Visiting code's attribute of type StackMapEntry: nothing done!!");
  throw new RuntimeException(
      "Not implemented: net.java.dev.jminimizer.ConstantPoolCleanerVisitor." +
      "visitStackMapEntry(org.apache.bcel.classfile.StackMapEntry)");
}

public JavaClass cleanUpClassGen(ClassGen classGen) {
  String className = classGen.getClassName();
  oldPool = classGen.getConstantPool();
  ClassGen newClassGen = new ClassGen(className, classGen
      .getSuperclassName(), classGen.getFileName(), classGen
      .getAccessFlags(), classGen.getInterfaceNames(), newPool);
  Attribute[] attributes = classGen.getAttributes();
  for (int i = 0; i < attributes.length; i++) {
    attributes[i].accept(this);
  }
  Method[] methods = classGen.getMethods();
  for (int i = 0; i < methods.length; i++) {
    //TODO used because a bug in bcel in method MethodGen.copy(String,
    // ConstantPoolGen)
    if (methods[i].isAbstract() || methods[i].isNative()) {
      methods[i].setNameIndex(newPool.addConstant(oldPool
          .getConstant(methods[i].getNameIndex()), oldPool));
      methods[i].setSignatureIndex(newPool.addConstant(oldPool
          .getConstant(methods[i].getSignatureIndex()), oldPool));
    } else {
      MethodGen methodGen = new MethodGen(methods[i], className,
          oldPool);
      methodGen = methodGen.copy(className, newPool);
      if (deepStripment) {
        methodGen.stripAttributes(deepStripment);
        attributes = methodGen.getAttributes();
        for (int j = 0; j < attributes.length; j++) {
          if (attributes[j] instanceof Synthetic
              || attributes[j] instanceof Deprecated) {
            methodGen.removeAttribute(attributes[j]);
          }
        }
      }
      methods[i] = methodGen.getMethod();
    }
    methods[i].accept(this);
    newClassGen.addMethod(methods[i]);
```

```
    }
    Field[] fields = classGen.getFields();
    for (int i = 0; i < fields.length; i++) {
      if (deepStripment) {
        attributes = fields[i].getAttributes();
        for (int j = 0; j < attributes.length; j++) {
          if (attributes[j] instanceof Synthetic
              || attributes[j] instanceof Deprecated) {
            FieldGen fieldGen = new FieldGen(fields[i], oldPool);
            fieldGen.removeAttribute(attributes[j]);
            fields[i] = fieldGen.getField();
          }
        }
      }
      fields[i].accept(this);
      newClassGen.addField(fields[i]);
    }
    if (deepStripment) {
      attributes = newClassGen.getAttributes();
      for (int i = 0; i < attributes.length; i++) {
        if (attributes[i] instanceof SourceFile
            || attributes[i] instanceof Deprecated) {
          newClassGen.removeAttribute(attributes[i]);
        }
      }
    }
    return newClassGen.getJavaClass();
  }

}
```

```
package net.java.dev.jminimizer;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

import net.java.dev.jminimizer.beans.Field;
import net.java.dev.jminimizer.beans.Method;
import net.java.dev.jminimizer.util.ClassUtils;
import net.java.dev.jminimizer.util.Configurator;
import net.java.dev.jminimizer.util.InstructionSet;
import net.java.dev.jminimizer.util.Visitor;

import org.apache.bcel.Repository;
```

```
import org.apache.bcel.classfile.Attribute;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.Synthetic;
import org.apache.bcel.generic.ClassGen;
import org.apache.bcel.generic.ConstantPoolGen;
import org.apache.bcel.generic.FieldInstruction;
import org.apache.bcel.generic.Instruction;
import org.apache.bcel.generic.InstructionHandle;
import org.apache.bcel.generic.InvokeInstruction;
import org.apache.bcel.generic.LoadClass;
import org.apache.bcel.generic.MethodGen;
import org.apache.bcel.generic.ObjectType;
import org.apache.bcel.util.InstructionFinder;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * @author Thiago ãLeo Moreira <thiagolm@dev.java.net>
 *
 *
 *
 *
 */
public class Analyser {

  private static final Log log = LogFactory.getLog(Analyser.class);

  protected Set classes;

  protected Configurator configurator;

  protected Set methodsThatUseClassForName;

  protected Set notProcessedMethods;

  protected net.java.dev.jminimizer.util.Repository repository;

  /**
   * @param configurator
   * @param repository
   * @throws ClassNotFoundException
   */
  public Analyser(Configurator configurator,
      net.java.dev.jminimizer.util.Repository repository)
      throws ClassNotFoundException {
    super();
    this.configurator = configurator;
    this.repository = repository;
    this.notProcessedMethods = new HashSet();
```

```
    this.classes = new TreeSet(Collections.reverseOrder());
    this.methodsThatUseClassForName = new HashSet();
    Repository.setRepository(this.repository);
  }

  /**
   * @param methods
   * @param superClass
   * @param className
   * @param usedMethods
   */
  private void analiseMethodFromSuperClass(List methods,
      JavaClass superClass, String className, Set usedMethods) {
    classes.add(superClass.getClassName());
    for (int i = 0; i < methods.size();) {
      org.apache.bcel.classfile.Method mcf = (org.apache.bcel.classfile.Method) methods
          .get(i);
      Method m = new Method(className, mcf.getName(), mcf.getSignature());
      ClassGen scg = new ClassGen(superClass);
      if (mcf.getName().equals("<cinit>")
          && mcf.getSignature().equals("()V")) {
        notProcessedMethods.add(m);
        methods.remove(mcf);
        log.trace("Find a class initializer: " + m);
        continue;
      }
      if (mcf.getName().equals("<init>")) {
        methods.remove(mcf);
        log.trace("Find a default constructor: " + m);
        continue;
      }
      if (scg.containsMethod(mcf.getName(), mcf.getSignature()) != null
          && !usedMethods.contains(m)) {
        notProcessedMethods.add(m);
        methods.remove(mcf);
      } else {
        i++;
      }
    }
  }

  /**
   * @param method
   * @throws ClassNotFoundException
   */
  private void analiseMethodThatUseClassForName(Method method)
      throws ClassNotFoundException {
    MethodGen mg = method.toMethodGen();
    if (method.getName().equals(Transformer.SYNTHETIC_METHOD_NAME)) {
```

```
      Attribute[] a = mg.getAttributes();
      for (int i = 0; i < a.length; i++) {
        //if method was the method that centralize calls to
        // java.lang.Class.forName(java.lang.String className) do
        // nothing.
        if (a[i] instanceof Synthetic) {
          return;
        }
      }
    }
    methodsThatUseClassForName.add(method);
    log
        .debug("Method that contains a call to java.lang.Class.forName(java.lang.String className): "
            + method);
  }

  /**
   * @param method
   * @param usedMethods
   * @return @throws
   *            ClassNotFoundException
   */
  public void analyse(Method method, Set usedMethods)
      throws ClassNotFoundException {
    if (!usedMethods.contains(method)) {
      String className = method.getClassName();
      if (configurator.inspect(className) && classes.add(className)) {
        //used just to relax de user
        log.info("Analysing class: " + method.getClassName());
      }
      usedMethods.add(method);
      MethodGen methodGen = method.toMethodGen();
      if (methodGen.isNative()) {
        log.trace("Native method: " + method.toString());
        return;
      }
      if (methodGen.isAbstract()) {
        log.trace("Abstract method: " + method.toString());
        return;
      }
      LoadClass[] loadClasses = this.findLoadClassInstructions(methodGen);
      ConstantPoolGen pool = methodGen.getConstantPool();
      for (int i = 0; i < loadClasses.length; i++) {
        if (loadClasses[i] instanceof InvokeInstruction) {
          InvokeInstruction instruction = (InvokeInstruction) loadClasses[i];
          Method methodTemp = new Method(instruction
              .getClassName(pool), instruction.getName(pool),
              instruction.getSignature(pool));
          if (usedMethods.contains(methodTemp)) {
```

```
          continue;
        }
        methodTemp = ClassUtils.findMethod(configurator, classes,
            instruction.getClassName(pool), instruction
                .getName(pool), instruction
                .getSignature(pool));
        if (methodTemp.getName().equals("forName")
            && methodTemp.getClassName().equals(
                "java.lang.Class")) {
          this.analiseMethodThatUseClassForName(method);
        }
        // test if this method should be analyse
        if (configurator.inspect(methodTemp)) {
          this.analyse(methodTemp, usedMethods);
        } else {
          usedMethods.add(methodTemp);
        }
      } else if (loadClasses[i] instanceof FieldInstruction) {
        FieldInstruction fieldInstruction = (FieldInstruction) loadClasses[i];
        className = fieldInstruction.getClassName(pool);
        usedMethods
            .add(new Field(className, fieldInstruction
                .getName(pool), fieldInstruction
                .getSignature(pool)));
        this.buildHierachyTree(className);
      } else {
        ObjectType objectType = loadClasses[i]
            .getLoadClassType(pool);
        //used to cacth casts from object to array of primitive
        // types (int[], char[] long[])
        if (objectType != null) {
          className = objectType.getClassName();
          //used to cacth new, multinewarray, checkcast and
          // instanceof
          this.buildHierachyTree(className);
        }
      }
    }
  }
}

private void buildHierachyTree(String className)
    throws ClassNotFoundException {
  if (configurator.inspect(className) && classes.add(className)) {
    //used just to relax de user
    log.info("Analysing class: " + className);
    JavaClass javaClass = repository.loadClass(className);
    JavaClass superJavaClass = javaClass.getSuperClass();
    while (superJavaClass != null) {
```

```
        className = superJavaClass.getClassName();
        superJavaClass = superJavaClass.getSuperClass();
        this.buildHierachyTree(className);
      }
      String[] javaClasses = javaClass.getInterfaceNames();
      for (int i = 0; i < javaClasses.length; i++) {
        this.buildHierachyTree(javaClasses[i]);
      }
    }
  }

  /**
   * @param methods
   * @param usedMethods
   * @throws ClassNotFoundException
   */
  public void analyse(Method[] methods, Set usedMethods)
      throws ClassNotFoundException {
    for (int i = 0; i < methods.length; i++) {
      this.analyse(methods[i], usedMethods);
    }
    this.analyseOverridedMethods(usedMethods);
    int size = notProcessedMethods.size();
    if (size != 0) {
      methods = new Method[size];
      notProcessedMethods.toArray(methods);
      notProcessedMethods.clear();
      this.analyse(methods, usedMethods);
    }
    log
        .debug("Quantity of methods that call java.lang.Class.forName(java.lang.String className): "
            + methodsThatUseClassForName.size());
  }

  /**
   * @param usedMethods
   * @throws ClassNotFoundException
   */
  private void analyseOverridedMethods(Set usedMethods)
      throws ClassNotFoundException {
    String[] cls = (String[]) classes.toArray(new String[0]);
    for (int i = 0; i < cls.length; i++) {
      this.analyseOverridedMethods(cls[i], usedMethods);
    }
  }

  /**
   * @param className
   * @param usedMethods
```

```
 * @throws ClassNotFoundException
 */
private void analyseOverridedMethods(String className, Set usedMethods)
    throws ClassNotFoundException {
  JavaClass jc = repository.loadClass(className);
  List methods = new ArrayList(Arrays.asList(jc.getMethods()));
  int size = methods.size();
  JavaClass superClass = jc.getSuperClass();
  while (superClass != null && methods.size() != 0
      && configurator.inspect(className)) {
    this.analiseMethodFromSuperClass(methods, superClass, className,
        usedMethods);
    superClass = superClass.getSuperClass();
  }
  JavaClass[] is = jc.getAllInterfaces();
  for (int i = 0; i < is.length && methods.size() != 0
      && configurator.inspect(className); i++) {
    this.analiseMethodFromSuperClass(methods, is[i], className,
        usedMethods);
  }

}

/**
 * @param method
 * @return
 */
private LoadClass[] findLoadClassInstructions(MethodGen method) {
  Set instructions = new InstructionSet();
  InstructionFinder finder = new InstructionFinder(method
      .getInstructionList());
  Iterator i = finder.search("LoadClass");
  while (i.hasNext()) {
    InstructionHandle[] ih = (InstructionHandle[]) i.next();
    if (ih.length != 1) {
      throw new RuntimeException("Just one instruction must return!");
    } else {
      Instruction instruction = ih[0].getInstruction();
      instructions.add(instruction);
    }
  }
  return (LoadClass[]) instructions.toArray(new LoadClass[0]);
}

/**
 * @return Returns the methodsThatUseClassForName.
 */
public Set getMethodsThatUseClassForName() {
  return methodsThatUseClassForName;
```

```
  }

  /**
   * @param visitor
   */
  public void receiveVisitor(Visitor visitor) throws Exception {
    Iterator iter = classes.iterator();
    while (iter.hasNext()) {
      String className = (String) iter.next();
      visitor.visit(className);
    }
  }
}

package net.java.dev.jminimizer;

import java.io.File;
import java.util.HashSet;
import java.util.Set;

import net.java.dev.jminimizer.util.Configurator;
import net.java.dev.jminimizer.util.Repository;
import net.java.dev.jminimizer.util.URLRepository;
import net.java.dev.jminimizer.util.Visitor;
import net.java.dev.jminimizer.util.XMLConfigurator;

import org.apache.commons.cli.BasicParser;
import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;

/**
 * @author Thiago ãLeo Moreira
 * @since Apr 18, 2004
 *
 */
public class JMinimizer {

  /**
   *
   * @param args
   * @throws Exception
   */
  public static void main(String[] args) throws Exception {
    CommandLineParser parser = new BasicParser();
    CommandLine cl = null;
    ;
```

```
    try {
      cl = parser.parse(getOptions(), args);
    } catch (ParseException e) {
      HelpFormatter hf = new HelpFormatter();
      hf.printHelp("net.java.dev.jminimizer.JMinimizer", getOptions());
      System.exit(0);
    }
    File file = (File) cl.getOptionObject('c');
    if (!file.exists()) {
      System.out.println("Configuration file not found: " + file);
      System.exit(0);
    }
    Configurator configurator = new XMLConfigurator(file);
    Repository repo = new URLRepository(configurator.getProgramClasspath(),
        configurator.getRuntimeClasspath());
    System.out.println("Analysing...");
    Analyser an = new Analyser(configurator, repo);
    Set usedMethods = new HashSet();
    an.analyse(configurator.getMethodsToInspect(), usedMethods);
    System.out.println("Transforming...");
    Set methodsThatUseClassForName = an.getMethodsThatUseClassForName();
    Visitor visitor = new Transformer(configurator, repo, usedMethods,
        methodsThatUseClassForName);
    an.receiveVisitor(visitor);
    visitor.finish();
  }

  /**
   *
   * @return
   */
  private static Options getOptions() {
    Options opts = new Options();
    //configFile option
    Option op = new Option("c", "config", true, "Configuration file.");
    op.setType(File.class);
    op.setRequired(true);
    opts.addOption(op);
    return opts;
  }
}

package net.java.dev.jminimizer;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URL;
```

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.jar.Attributes;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.jar.Manifest;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import net.java.dev.jminimizer.beans.Method;
import net.java.dev.jminimizer.util.Configurator;
import net.java.dev.jminimizer.util.Repository;
import net.java.dev.jminimizer.util.Visitor;

import org.apache.bcel.Constants;
import org.apache.bcel.classfile.Attribute;
import org.apache.bcel.classfile.ConstantUtf8;
import org.apache.bcel.classfile.Field;
import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.LocalVariable;
import org.apache.bcel.classfile.Synthetic;
import org.apache.bcel.generic.ARETURN;
import org.apache.bcel.generic.ATHROW;
import org.apache.bcel.generic.BasicType;
import org.apache.bcel.generic.ClassGen;
import org.apache.bcel.generic.CodeExceptionGen;
import org.apache.bcel.generic.ConstantPoolGen;
import org.apache.bcel.generic.DUP;
import org.apache.bcel.generic.GETSTATIC;
import org.apache.bcel.generic.IFNONNULL;
import org.apache.bcel.generic.Instruction;
import org.apache.bcel.generic.InstructionFactory;
import org.apache.bcel.generic.InstructionHandle;
import org.apache.bcel.generic.InstructionList;
import org.apache.bcel.generic.InstructionTargeter;
import org.apache.bcel.generic.LDC;
import org.apache.bcel.generic.LocalVariableGen;
import org.apache.bcel.generic.MethodGen;
import org.apache.bcel.generic.ObjectType;
import org.apache.bcel.generic.POP;
import org.apache.bcel.generic.TargetLostException;
import org.apache.bcel.generic.Type;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;

/**
 * @author Thiago ãLeo Moreira <thiagolm@dev.java.net>
 * @since Apr 16, 2004
 *
 */
public class Transformer implements Visitor {

  private static final Log log = LogFactory.getLog(Transformer.class);

  public static final String SYNTHETIC_METHOD_NAME = "SYNTHETIC$JMINIMIZER";

  private Set classesInProgramClasspath;

  private Set classesUseByProgram;

  private String classThatContainsTheNewMethod;

  private Configurator configurator;

  private Set methodsThatUseClassForName;

  private JarOutputStream out;

  private Repository repository;

  private Set usedMethods;

  /**
   * @param configurator
   * @param repository
   * @param usedMethods
   * @param methodsThatUseClassForName
   * @throws IOException
   */
  public Transformer(Configurator configurator, Repository repository,
      Set usedMethods, Set methodsThatUseClassForName) throws IOException {
    super();
    this.configurator = configurator;
    this.repository = repository;
    this.usedMethods = usedMethods;
    this.methodsThatUseClassForName = methodsThatUseClassForName;
    this.classesInProgramClasspath = repository.getProgramClasses();
    this.classesUseByProgram = new HashSet();
    File output = configurator.getTransformationOutput();
    if (output.isFile()) {
      out = new JarOutputStream(new FileOutputStream(output, false)) {
```

```
      /**
       * @see java.util.zip.ZipOutputStream#close()
       */
      public void close() throws IOException {
        this.closeEntry();
      }
    };
  }
}

/**
 * @param className
 * @param pool
 * @return
 */
private MethodGen createMethod(String className, ConstantPoolGen pool) {
  int accessFlag = Constants.ACC_STATIC | Constants.ACC_PUBLIC;
  Type returnType = Type.getType("Ljava/lang/Class;");
  InstructionList code = new InstructionList();
  InstructionFactory factory = new InstructionFactory(pool);
  Instruction i = InstructionFactory.createLoad(Type.STRING, 0);
  InstructionHandle startPC;
  startPC = code.append(i);
  i = factory.createInvoke("java.lang.Class", "forName", returnType,
      new Type[] { Type.STRING }, Constants.INVOKESTATIC);
  InstructionHandle endPC = code.append(i);
  i = InstructionFactory.createReturn(returnType);
  code.append(i);
  i = InstructionFactory.createStore(Type.OBJECT, 1);
  InstructionHandle handlerPC = code.append(i);
  i = factory.createNew(new ObjectType("java.lang.NoClassDefFoundError"));
  code.append(i);
  code.append(new DUP());
  i = InstructionFactory.createLoad(Type.OBJECT, 1);
  code.append(i);
  i = factory.createInvoke("java.lang.Throwable", "getMessage",
      Type.STRING, new Type[0], Constants.INVOKEVIRTUAL);
  code.append(i);
  i = factory.createInvoke("java.lang.NoClassDefFoundError", "<init>",
      Type.VOID, new Type[] { Type.STRING }, Constants.INVOKESPECIAL);
  code.append(i);
  code.append(new ATHROW());
  MethodGen mg = new MethodGen(accessFlag, returnType,
      new Type[] { Type.STRING }, new String[] { "className" },
      SYNTHETIC_METHOD_NAME, className, code, pool);
  CodeExceptionGen ceg = mg.addExceptionHandler(startPC, endPC,
      handlerPC, new ObjectType("java.lang.ClassNotFoundException"));
  mg.setMaxStack();
```

```
    mg.setMaxLocals();
    int x = pool.addConstant(new ConstantUtf8("Synthetic"), pool);
    mg.addAttribute(new Synthetic(x, 0, null, pool.getConstantPool()));
    return mg;
  }

  /**
   * @param jc
   * @throws IOException
   */
  private void dump(JavaClass jc) throws IOException {
    String classFile = jc.getClassName().replace('.', File.separatorChar)
        .concat(".class");
    OutputStream stream;
    if (out == null) {
      File file = new File(configurator.getTransformationOutput(),
          classFile);
      File directory = file.getParentFile();
      if (directory != null) {
        directory.mkdirs();
      }
      stream = new FileOutputStream(file);
    } else {
      out.putNextEntry(new JarEntry(classFile));
      stream = out;
    }
    log.debug("Dumping class: " + classFile);
    jc.dump(stream);
  }

  public void finish() throws IOException {
    URL[] programsClasspath = configurator.getProgramClasspath();
    Set programResources = repository.getProgramResources();
    Iterator iterator = programResources.iterator();
    if (out != null) {
      Set duplicateEntries = new HashSet();
      while (iterator.hasNext()) {
        URL resource = (URL) iterator.next();
        String resourceString = resource.toString();
        for (int i = 0; i < programsClasspath.length; i++) {
          String programClasspathString = programsClasspath[i]
              .toString();
          if (resourceString.startsWith(programClasspathString)) {
            resourceString = resourceString
                .substring(programClasspathString.length());
            if (resourceString.equals("META-INF/MANIFEST.MF")) {
              break;
            }
            if (duplicateEntries.contains(resourceString)) {
```

```
          log
              .warn("The entry "
                  + resourceString
                  + " already exist. The file "
                  + resource.toString()
                  + " will not be adds to generated program!!!");
          break;
        }
        duplicateEntries.add(resourceString);
        out.putNextEntry(new JarEntry(resourceString));
        InputStream in = resource.openStream();
        byte[] data = new byte[1024];
        int lengthOfDataRead = 0;
        while ((lengthOfDataRead = in.read(data)) != -1) {
          out.write(data, 0, lengthOfDataRead);
        }
        in.close();
        break;
      }
    }
  }
  out.putNextEntry(new JarEntry("META-INF/MANIFEST.MF"));
  this.writeManifest(out);
  out.finish();
} else {
  while (iterator.hasNext()) {
    URL resource = (URL) iterator.next();
    String resourceString = resource.toString();
    for (int i = 0; i < programsClasspath.length; i++) {
      String programClasspathString = programsClasspath[i]
          .toString();
      if (resourceString.startsWith(programClasspathString)) {
        resourceString = resourceString
            .substring(programClasspathString.length());
        if (resourceString.equals("META-INF/MANIFEST.MF")) {
          break;
        }
        int lastIndex;
        if (programClasspathString.startsWith("file:")) {
          lastIndex = resourceString
              .lastIndexOf(File.separatorChar);
        } else {
          lastIndex = resourceString.lastIndexOf('/');
        }
        File directory = new File(configurator
            .getTransformationOutput(), resourceString
            .substring(0, lastIndex));
        if (directory.mkdirs()) {
        }
```

```
          File file = new File(directory, resourceString
              .substring(lastIndex + 1));
          if (file.exists()) {
            log
                .warn("The file "
                    + resourceString
                    + " already exist. The file "
                    + resource.toString()
                    + " will not be adds to generated program!!!");
            break;
          }
          InputStream in = resource.openStream();
          OutputStream out = new FileOutputStream(new File(
              directory, resourceString
                  .substring(lastIndex + 1)));
          byte[] data = new byte[1024];
          int lengthOfDataRead = 0;
          while ((lengthOfDataRead = in.read(data)) != -1) {
            out.write(data, 0, lengthOfDataRead);
          }
          in.close();
          out.close();
          break;
        }
      }
    }
    File directory = new File(configurator.getTransformationOutput(),
        "META-INF");
    directory.mkdirs();
    File file = new File(directory, "MANIFEST.MF");
    FileOutputStream out = new FileOutputStream(file);
    this.writeManifest(out);
    out.close();
  }
}

private void writeManifest(OutputStream out) throws IOException {
  Manifest manifest = new Manifest();
  Attributes attributes = manifest.getMainAttributes();
  attributes.putValue("Manifest-Version", "1.0");
  //TODO put the version of actual JMinimizer
  attributes.putValue("Created-By", "JMinimizer alpha-2");
  manifest.write(out);
}

/**
 * @param pool
 * @param className
 * @return
```

```
 */
private InstructionList buildInstructionListWithCallToNewMethod(
    ConstantPoolGen pool, String className) {
  InstructionList list = new InstructionList();
  InstructionFactory i = new InstructionFactory(pool);
  list.append(i.createInvoke(className, SYNTHETIC_METHOD_NAME,
      new ObjectType("java.lang.Class"), new Type[] { Type.STRING },
      Constants.INVOKESTATIC));
  return list;

}

/**
 * @param document
 * @param method
 * @return
 */
private Element reportMethod(Document document,
    org.apache.bcel.classfile.Method method) {
  Element eMethod = document.createElement("method");
  eMethod.setAttribute("name", method.getName());
  Element eArguments = document.createElement("arguments");
  Type[] args = Type.getArgumentTypes(method.getSignature());
  for (int j = 0; j < args.length; j++) {
    Element eArg;
    if (args[j] instanceof BasicType) {
      eArg = document.createElement("primitiveType");
    } else {
      eArg = document.createElement("classType");
    }
    eArg.setAttribute("name", args[j].toString());
    eArguments.appendChild(eArg);
  }
  Element eReturn = document.createElement("return");
  Type type = Type.getReturnType(method.getSignature());
  if (type.getType() == Constants.T_VOID) {
    eReturn.appendChild(document.createElement("void"));
  } else {
    Element temp = null;
    if (type instanceof BasicType) {
      temp = document.createElement("primitiveType");
    } else {
      temp = document.createElement("classType");
    }
    temp.setAttribute("name", type.toString());
    eReturn.appendChild(temp);
  }
  eMethod.appendChild(eArguments);
  eMethod.appendChild(eReturn);
```

```
    return eMethod;
  }

  private void transformBySynthetic(MethodGen method, ConstantPoolGen pool,
      InstructionList code) {
    InstructionHandle start = code.getStart();
    InstructionHandle end = code.getEnd();
    InstructionHandle temp = start.getNext();
    InstructionFactory factory = new InstructionFactory(pool);
    code.insert(temp, this.buildInstructionListWithCallToNewMethod(pool,
        classThatContainsTheNewMethod));
    InstructionHandle newEnd = code.insert(temp, new ARETURN());
    try {
      code.delete(temp, end);
    } catch (TargetLostException e) {
      InstructionHandle[] targets = e.getTargets();
      for (int d = 0; d < targets.length; d++) {
        InstructionTargeter[] targeters = targets[d].getTargeters();
        for (int j = 0; j < targeters.length; j++) {
          if (targeters[j] instanceof CodeExceptionGen) {
            method
                .removeExceptionHandler((CodeExceptionGen) targeters[j]);
          }
          if (targeters[j] instanceof LocalVariableGen) {
            LocalVariableGen lvg = (LocalVariableGen) targeters[j];
            //method.removeLocalVariable(lvg);
            LocalVariable lv = lvg.getLocalVariable(pool);
            if (lv.getStartPC() == 0) {
              lvg.setEnd(newEnd);
            } else {
              method.removeLocalVariable(lvg);
            }
          }
        }
      }
    }
  }

  /**
   * @param method
   * @param pool
   * @return @throws
   *         Exception
   */
  private org.apache.bcel.classfile.Method transform(MethodGen method,
      ConstantPoolGen pool) throws Exception {
    Attribute[] attrs = method.getAttributes();
    InstructionList code = method.getInstructionList();
    InstructionHandle[] ins = code.getInstructionHandles();
```

```
    boolean isCompilerArtificios = false;
    //test if this method is Synthetic
    for (int i = 0; i < attrs.length; i++) {
      if (attrs[i] instanceof Synthetic
          && !method.getName().equals(SYNTHETIC_METHOD_NAME)) {
        this.transformBySynthetic(method, pool, code);
        isCompilerArtificios = true;
      }
    }
    //test if this method has a sequence os instructions write by compiler
    for (int j = 0; j < ins.length; j++) {
      if ((j < (ins.length - 4))
          && ins[j].getInstruction() instanceof GETSTATIC
          && ins[j + 1].getInstruction() instanceof DUP
          && ins[j + 2].getInstruction() instanceof IFNONNULL
          && ins[j + 3].getInstruction() instanceof POP
          && ins[j + 4].getInstruction() instanceof LDC) {
        this.transformByInstructionSequence(ins[j + 5], ins[j + 14],
            code, pool, method);
        isCompilerArtificios = true;
      }
    }
    if (!isCompilerArtificios) {
      log
          .warn("The method "
              + new Method(method.getClassName(), method
                  .getName(), method.getSignature())
              + " has a call to java.lang.Class.forName(java.lanag.String className)!!");
    }
    return method.getMethod();
  }

  private void transformByInstructionSequence(InstructionHandle start,
      InstructionHandle end, InstructionList code, ConstantPoolGen pool,
      MethodGen method) throws TargetLostException {
    InstructionHandle temp = start;
    code.insert(start, this.buildInstructionListWithCallToNewMethod(pool,
        classThatContainsTheNewMethod));
    temp = start.getNext().getNext().getNext();
    try {
      code.delete(temp, end);
    } catch (TargetLostException e) {
      InstructionHandle[] handles = e.getTargets();
      for (int i = 0; i < handles.length; i++) {
        InstructionTargeter[] targets = handles[i].getTargeters();
        for (int j = 0; j < targets.length; j++) {
          if (targets[j] instanceof CodeExceptionGen) {
            CodeExceptionGen ceg = (CodeExceptionGen) targets[j];
            method.removeExceptionHandler(ceg);
```

```
          }
        }
      }
    }

    try {
      code.delete(start);
    } catch (TargetLostException e) {
      InstructionHandle[] handles = e.getTargets();
      for (int i = 0; i < handles.length; i++) {
        InstructionTargeter[] targets = handles[i].getTargeters();
        for (int j = 0; j < targets.length; j++) {
          if (targets[j] instanceof CodeExceptionGen) {
            CodeExceptionGen ceg = (CodeExceptionGen) targets[j];
            method.removeExceptionHandler(ceg);
          }
        }
      }
    }
    method.setConstantPool(pool);
    method.setInstructionList(code);
  }

  /**
   * @see net.java.dev.jminimizer.util.Visitor#visit(net.java.dev.jminimizer.beans.Class)
   */
  public void visit(String className) throws Exception {
    if (className.indexOf("Stub") != -1) {
      System.out.println(className);
    }
    if (classesInProgramClasspath.contains(className)) {
      classesUseByProgram.add(className);
      Document document = DocumentBuilderFactory.newInstance()
          .newDocumentBuilder().newDocument();
      Element eClazz = document.createElement("class");
      eClazz.setAttribute("name", className);
      document.appendChild(eClazz);
      JavaClass javaClass = repository.findClass(className);
      ClassGen cg = new ClassGen(javaClass);
      org.apache.bcel.classfile.Method[] ms = javaClass.getMethods();
      log.debug("Cleaning class: " + className);
      for (int i = 0; i < ms.length; i++) {
        Method m = new Method(className, ms[i].getName(), ms[i]
            .getSignature());
        //Deleting methods
        if (!usedMethods.contains(m)) {
          log.debug("Removing method: " + m);
          cg.removeMethod(ms[i]);
          if (true) {
```

```
                eClazz.appendChild(this.reportMethod(document, ms[i]));
            }
        } else {
            if (methodsThatUseClassForName.contains(m)) {
                org.apache.bcel.classfile.Method mc = m
                        .toClassFileMethod();
                ConstantPoolGen pool = cg.getConstantPool();
                if (classThatContainsTheNewMethod == null) {
                    classThatContainsTheNewMethod = m.getClassName();
                    cg.addMethod(this.createMethod(m.getClassName(),
                            pool).getMethod());
                }
                cg.replaceMethod(mc, this.transform(m.toMethodGen(),
                        pool));
            }
        }
    }
    //Deleting fields
    Field[] fields = javaClass.getFields();
    for (int i = 0; i < fields.length; i++) {
        net.java.dev.jminimizer.beans.Field field = new net.java.dev.jminimizer.beans.Field(
                className, fields[i].getName(), fields[i]
                        .getSignature());
        if (!usedMethods.contains(field)) {
            log.debug("Removing field: " + field);
            cg.removeField(fields[i]);
        }
    }
    log.debug("Cleaning up the constant pool of class: "
            + cg.getClassName());
    ConstantPoolCleanerVisitor visitor = new ConstantPoolCleanerVisitor(
            configurator.isDeepStripment(), classesUseByProgram);
    javaClass = visitor.cleanUpClassGen(cg);
    this.dump(javaClass);
    if (eClazz.hasChildNodes()) {
        File directory = configurator.getReportDirectory();
        directory.mkdirs();
        File report = new File(directory, className + ".xml");
        javax.xml.transform.Transformer trans = TransformerFactory
                .newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.INDENT, "yes");
        trans.transform(new DOMSource(document), new StreamResult(
                report));
    }
  }
 }

}
```

Referências Bibliográficas

[bce 03] Disponível em <http://jakarta.apache.org/bcel>. Acesso em: Agosto.

[boo 03] Disponível em <http://java.sun.com>. Acesso em: Agosto.

[BRA 98] BRADLEY, Q.; HORSPOOL, R.; VITEK, J. **JAZZ: An efficient compressed format for Java archive files**.

[bre 04] Disponível em <http://brew.qualcomm.com/brew/en/>. Acesso em: Julho.

[cvs 04] Disponível em <http://www.cvshome.org>. Acesso em: Outubro.

[ecl 03] Disponível em <http://www.eclipse.org>. Acesso em: Novembro.

[fed 04] Disponível em <http://fedora.redhat.com>. Acesso em: Outubro.

[HOR 98] HORSPOOL, R. N.; CORLESS, J. Tailored compression of java class files. [S.l.], v.28, p.1253–1268, Outubro, 1998.

[icq 04] Disponível em <http://www.icq.com>. Acesso em: Fevereiro.

[j2m 03] Disponível em <http://java.sun.com/j2me>. Acesso em: Agosto.

[jar 04] Disponível em <http://java.sun.com/docs/books/tutorial/jar/>. Acesso em: Agosto.

[jav 03] Disponível em <http://java.sun.com>. Acesso em: Agosto.

[jav 04] Disponível em <http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html>. Acesso em: Abril.

[jax 04] Disponível em <http://www.research.ibm.com/jax/>. Acesso em: Agosto.

[mav 04] Disponível em <http://maven.apache.org>. Acesso em: Abril.

[msn 04] Disponível em <http://messenger.msn.com>. Acesso em: Fevereiro.

[sop 04] Disponível em <http://www.s-cradle.com/english/products/compress>. Acesso em: Julho.

[sun 04] Disponível em <http://www.sun.com>. Acesso em: Outubro.

[thi 04] Disponível em <http://www.thinlet.com>. Acesso em: Março.

[wor 04] Disponível em <http://profs.sci.univr.it/ spoto/Bytecode05/>. Acesso em: Novembro.

[wsd 04] Disponível em <http://www-306.ibm.com/software/wireless/wsdd>. Acesso em: Outubro.