
Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

# Otimizando Compiladores De AspectJ Para Java ME

Fernando Henrique Calheiros Lopes

Trabalho de Graduação

Recife
22 de Agosto de 2007

Universidade Federal de Pernambuco
Centro de Informática

Fernando Henrique Calheiros Lopes

# Otimizando Compiladores De AspectJ Para Java ME

*Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.*

Orientador: *Prof. Dr. Paulo Henrique Monteiro Borba*

Recife
22 de Agosto de 2007

*A minha família*

# Agradecimentos

*"Tu deviens responsable pour toujours de ce que tu as apprivoisé."*
—ANTOINE DE SAINT-EXUPÉRY

Esse trabalho marca o fim de uma jornada de cinco anos e uma das fases mais importantes da minha vida. Vou aproveitar esse espaço para agradecer a todos que, direta ou indiretamente, contribuíram nessa jornada.

Agradeço aos meus pais, por tudo que já fizeram por mim e por terem me propiciado a oportunidade de realizar este curso longe deles, e por terem sempre me apoiado e me incentivado.

A todos que contribuíram de alguma forma para elaboração deste trabalho. A Paulo, pela orientação, e a Vander e Vilmar pelas revisões e sugestões. A todos do projeto *FLiP*, os membros atuais e os que saíram do projeto.

A todos que fazem o Centro de Informática da UFPE, professores e funcionários.

A todos os meus amigos, os próximos e os distantes. Em especial a Alexandra, por ter me apoiado nos momentos mais difíceis do último ano, a Emannuel, por ter sido minha consciência externa, a Sylvia, por ter estado sempre lá pra me ouvir, e a todo o pessoal do lendas.

*Life is too important to be taken seriously.*
—OSCAR WILDE

# Resumo

Este trabalho se propõe a estabelecer os motivos por trás do *overhead* que AspectJ introduz no *bytecode* das classes Java, no contexto de uma Linha de Produtos de Sofware de um jogo *Java ME*. São apresentados os motivos do *overhead* e são propostas melhorias para os compiladores *ajc* e *abc*, para diminuir o *overhead*. Duas das melhorias propostas são implementadas no *abc* e o ganho obtido com as mesmas é apresentado.

**Palavras-chave:** Java ME, Linhas de Produtos de Software AspectJ, Programação Orientada a Aspectos, Compiladores, Otimização

# Abstract

This work intends to establish the reasons behind the overhead that AspectJ introduces in Java classes' bytecodes, in the context of a Java ME game Software Product Line. The motives behind the overhead are presented and improvements to the ajc and the abc compilers are proposed, to diminish this overhead. Two of the proposed improvements are implemented in the abc compiler and the gain obtained with them is presented.

**Keywords:** Java ME, Software Product Lines, AspectJ, Aspect-oriented Programming, Compilers, Optimization

# Sumário

# Lista de Listagens

# Lista de Tabelas

# Lista de Algoritmos

CAPÍTULO 1

# Introdução

*"If we don't believe in freedom of expression for people we despise, we don't believe in it at all."*

—NOAM CHOMSKY

O desenvolvimento de jogos móveis é um dos campos mais promissores da indústria de entretenimento digital. Projeções mostram que, até 2011, espera-se que sejam movimentados, em todo mundo, aproximadamente US$ 7 bilhões com jogos móveis [Ter07].

Um dos principais problemas do desenvolvimento de jogos móveis é o *porting* [ACV+05]: para uma empresa atuar nesse mercado é necessário que os jogos desenvolvidos pela mesma estejam disponíveis para uma grande variedade de aparelhos que possuem características, recursos e API's distintas. Cada produto gerado para atender uma família de aparelhos de um mesmo jogo é chamado uma instância desse jogo.

O processo de *porting*, que, em 2006, representou um terço do custo total do desenvolvimento de jogos [Ter07], consiste na adaptação de um jogo a fim de que ele possa ser executado em diferentes aparelhos, que envolve a identificação dos pontos de variação (pontos no código ou em artefatos onde o jogo varia entre uma plataforma e outra) e o tratamento dos mesmos.

A técnica mais utilizada pela indústria no processo de *porting* para tratar as variações no código dos jogos é o uso de compilação condicional [CLG+06] que deixa o código entrelaçado com os trechos de diferentes produtos no mesmo arquivo, mas tem a vantagem de não adicionar *overhead* ao tamanho dos jogos.

Outras técnicas estão sendo analisadas para substituir compilação condicional. Uma das técnicas mais promissoras para tratamento de variações é o uso de *Programação Orientada a Aspectos* (*AOP*, do inglês *Aspect Oriented Programming*) [KLM+97, ANS+06], através de AspectJ [KHH+01], uma extensão à linguagem Java.

O uso de AspectJ como mecanismo de inserção de variações em jogos em *Java ME* está sendo posto em prática pelo projeto *FLiP* (projeto de pesquisa financiado pela FINEP), que visa a construção de um conjunto de plugins para o Eclipse que promovem suporte ferramental ao uso de *Linhas de Produtos de Software* [CN02] no desenvolvimento de jogos em *Java ME*.

Um empecilho para o uso de AspectJ, como mecanismo de inserção de variações no código de jogos desenvolvidos em *Java ME* [Mic07] é o *overhead* no tamanho do código gerado pelos compiladores AspectJ. No contexto de jogos móveis esse *overhead* é proibitivo, pois boa parte dos aparelhos usados no mercado (por exemplo, no Brasil) tem limitações de memória. Mesmo que a tendência do mercado seja os aparelhos evoluírem para ter mais memória, as aplicações

também evoluem em complexidade e sempre ficam no limite, portanto uma diminuição deste *overhead* pode ser um diferencial para a criação de aplicações mais ricas.

Um exemplo simples desse *overhead* é que cada aspecto gera um arquivo .class próprio, visto que o processo de compilação de AspectJ gera classes Java compatíveis com as especificação da JVM [LY99]. No contexto de jogos móveis, onde o espaço disponível em memória de armazenamento e de execução é consideravelmente restrito, a adição de um .class pode restringir o número de features do jogo devido ao menor espaço disponível após a introdução dos mesmos. Trabalhos recentes [ACH+05b, Kuz04, Cor07] mostram que ainda há espaço para a otimização da geração de código AspectJ.

Otimizadores de *bytecode* [Pro07, Ret07] possuem passos de redução, onde o tamanho do *bytecode* de jogos em *Java ME* é diminuído consideravemente, tais como remover classes e atributos não usados, diminuir o nome de classes, atributos e métodos, entre outros.

Mesmo com o uso de otimizadores de código, o uso de AspectJ ainda deixa um *overhead* proibitivo. Se os pontos onde os compiladores de AspectJ são ineficientes, no sentido de gerar código mais compacto, forem identificados e modificações forem realizadas para melhorar a geração de código, o uso de AspectJ como mecanismo de inserção de variações de jogos em *Java ME* se tornaria uma opção viável.

É importante ressaltar que as otimizações propostas neste trabalho podem ser não são necessariamente aplicáveis em todos os contextos. Serão propostas otimizações com o objetivo de otimizar o código gerado de AspectJ quando o mesmo está sendo utilizado para inserir variações em Linhas de Produtos de Software, ou seja, quando não há muita quantificação. A pequena quantificação significa que as variações que estão sendo inseridas por AspectJ são inseridas em poucos pontos, normalmente cada código sendo inserido apenas em um ponto.

## 1.1 Objetivos deste estudo

A pesquisa apresentada aqui tem por finalidade melhorar a geração de código AspectJ para torná-la viável para o uso do mesmo como mecanismo de inserção de variações em jogos *Java ME*.

As contribuições deste trabalho são:

- Identificação de *overheads* na geração de código dos compiladores de AspectJ
- Implementação de melhorias em um dos compiladores
- Identificação de boas práticas no uso de AspectJ como mecanismo de inserção de variações em jogos *Java ME*

## 1.2 Metodologia

Para atingir os objetivos dessa pesquisa serão executadas cinco etapas: análise de tamanho, inspeção, discussão, implementação de otimizações, coleta de resultados e conclusão.

Na análise de tamanho será feita a comparação entre código gerado pelos compiladores *ajc* [Teaa, Teab] e *abc* [ACH+04, ACH+05a] em um jogo real desenvolvido por um parceiro industrial, a Meantime [Mea07]. Esta análise apresentará uma comparação do tamanho dos

*builds* das instâncias da versão original do jogo, ou seja, apenas com compilação condicional, com os de uma versão do jogo com variações extraídas para aspectos e compilada com o *ajc* e com o *abc*.

O resultado dessa análise permitirá uma quantificação do *overhead* que o uso de AspectJ adiciona no produto final e a escolha do compilador aonde serão feitas as otimizações.

A inspeção consistirá numa análise do código gerado por cada tipo de construção utilizada para tratar as variações existentes, para descobrir o motivo por trás do *overhead* de AspectJ.

Na discussão, os resultados levantados na análise serão utilizados para propor melhorias na geração de código de AspectJ. Nesta etapa também será feita a prova de que as melhorias propostas escolhidas para implementação preservam o comportamento do código gerado. Essa prova é de grande importância, uma vez que sem a mesma não há como garantir que as otimizações são válidas.

A coleta de resultados consistirá numa repetição da análise apenas com o compilador otimizado, para comparação de tamanho dos *jar*'s gerados antes e depois da implementação das melhorias e quantificação da diminuição no tamanho do código.

Na conclusão será feita uma apreciação dos resultados obtidos, serão levantadas as lições aprendidas e boas práticas identificadas durante a realização desta pesquisa e serão apresentadas as oportunidades de trabalhos futuros.

## 1.3 Escopo Deste Estudo

A ferramenta de extração de código do *FLiP*, chamada *FLiPEx* [CBS+07], permite ao usuário selecionar código Java no editor do Eclipse e extraí-lo para aspectos, através de *refactorings* existentes [Col05].

Para a definição do conjunto de *refactorings* que seriam oferecidos no *FLiPEx*, foi feita uma análise em 2 jogos desenvolvidos pela *Meantime*, essa análise determinou quais construções de AspectJ são necessárias para extrair as variações existentes em ambos os jogos e, também, quais variações não podem ser extraídas utilizando AspectJ [ANS+06].

AspectJ é uma linguagem que possui muitas construções, algumas delas, como o uso de *pointcuts cflow*, não podem ser utilizadas em *Java ME* devido ao uso de reflexão, recurso que não está disponível nos celulares que utilizam *Java ME*.

Este trabalho vai limitar-se ao estudo das construções de AspectJ identificadas pelo *FLiP* como necessárias para extrair a maior parte das variações identificadas nos 2 jogos avaliados, ou seja, as . O uso destas construções com recursos de Java 5 não será avaliado, uma vez que estes recursos não estão disponíveis em *Java ME*.

Como nos 2 jogos avaliados pelo projeto *FLiP* a quantificação foi mínima, ou seja, os códigos apenas eram inseridos em um determinado ponto, as otimizações propostas estarão visando atender a esse caso. Em casos em que trechos de código idênticos são inseridos em várias partes com o uso de AspectJ, ou seja, quando há maior quantificação, as otimizações aqui propostas (e também algumas já disponíveis nos próprios compiladores) não são recomendadas, uma vez que podem levar a um aumetno do tamanho do código.

## 1.4 Organização do trabalho

O Capítulo 2 apresenta os principais conceitos utilizados no trabalho. O Capítulo 3 apresenta a análise de tamanho, comparando o tamanho dos *builds*. O Capítulo 4 apresenta a análise individual das construções de AspectJ que estão no escopo do trabalho. O Capítulo 5 apresenta as otimizações propostas, a intuição por trás das mesmas, quais serão implementadas, detalhes da implementação e a prova de que o comportamento se mantém após suas aplicações. O Capítulo 6 apresentará os resultados obtidos com a aplicação das otimizações para quantificação dos ganhos obtidos. Por fim, o Capítulo 7 apresenta as conclusões do trabalho e oportunidades de trabalhos futuros.

CAPÍTULO 2

# Conceitos

*"To avoid situations in which you might make mistakes may be the biggest mistake of all."*
—PETER MCWILLIAMS

Neste capítulo serão apresentados os principais conceitos utilizados neste trabalho.

## 2.1 AspectJ

*Programação Orientada a Aspectos* permite a implementação de interesses transversais (cross-cutting concerns) de uma forma modular. A implementação mais difundida de *AOP* é a linguagem AspectJ.

AspectJ é uma extensão orientada a aspectos de Java. Por ser uma extensão de Java, o código gerado por programas AspectJ é *bytecode* compatível com a especificação da JVM [LY99] Essa compatibilidade é possível porque os compiladores de AspectJ transformam as construções específicas de AspectJ em código Java puro, toda as construções de AspectJ que afetam classes Java modificarão o códgio destas classes utilizando construções de Java. O processo de modificação das classes Java chama-se *weaving*.

Documentos explicando mais profundamente a linguagem AspectJ podem ser encontrados em [Tea05a, Tea05b]. Aspectos podem definir *pointcuts*, *advices* e *inter-type declarations*.

### 2.1.1 *Pointcuts*

*Pointcuts* definem um conjunto de pontos no fluxo de execução de um programa onde um trecho de código pode ser inserido (*advices*). Estes pontos são chamados de *join points*.

Os *join points* suportados por AspectJ são:

- *Method call* - quando um método é chamado (não inclúi chamadas via *super*).
- *Method execution* - quando o corpo de um método é executado.
- *Constructor call* - quando um objeto é instanciado e o construtor deste objeto é chamado (não inclúi chamadas de construtores via *super* ou *this*).
- *Constructor execution* - quando o corpo de um constructor é executado, após a sua chamada *this* ou *super*.
- *Static initializer execution* - quando o inicializador estático de uma classe executa.

- *Object pre-initialization* - antes do código de inicialização de um objeto de uma classe particular rodar. Isso inclui o tempo entre o início do seu primeiro constructor invocado e o início do construtor da sua classe pai.

- *Object initialization* - Quando o código de inicialização do objeto para uma classe particular roda. Isso inclui o tempo entre o retorno do construtor da sua classe pai e o retorno do seu primeiro construtor invocado.

- *Field reference* - quando um atributo não constante é referenciado.

- *Field set* - quando um atributo tem seu valor alterado.

- *Handler execution* - quando um tratador de exceção é executado.

- *Advice execution* - quando o corpo de um *advice* executa.

Como mencionado, *pointcuts* descrevem conjuntos de *join points*. Para a definição de pointcuts são utilizados designadores de *pointctus*. Por exemplo, call(Signature) identifica chamadas de método ou de construtor que casem com o padrão especificado por Signature. Designadores de *pointcut* podem ser combinados com os operadores lógicos &&, || e !.

### 2.1.2 Advices

Os *advices* especificam código para ser executado e quando ele deve ser executado. As escolhas possíveis são antes (before), após (after) ou no lugar (around) do join point capturado.

O *advice before* executa alguns comandos antes do join point capturado. O *advice after* possui três formas: *after returning* executa o código do *advice* apenas quando o *join point* executa com sucesso, opcionalmente ele também pode expor o valor de retorno; *after throwing* é usado para executar um *advice* apenas se o *join point* capturado levantar uma exceção específica, ele também pode, opcionalmente, expor a exceção; o último caso é o *advice after*, ele executa o código do *advice* não importanto se o join point foi executado com sucesso ou não.

O *advice* around pode executar alguns comandos antes e após o join point, usando ou não uma chamada a proceed, que permite a execução do join point capturado. Dessa forma, o join point pode ser completamente sobrescrito se proceed não for chamado. O comando proceed pode também alterar os valores do contexto exposto ao *advice*.

### 2.1.3 Inter-type Declarations

Além de moficar a dinâmica de execução do programa, um aspecto pode modificar sua estrutura estática. Os mecanismos de modificação estática providos por *inter-type declarations* são:

- introdução de novos atributos ou métodos a classes e interfaces existentes,

- modificação da hierarquia de classes, indicar que uma classe estende outra classe ou implementa uma dada interface.

### 2.1.4 ajc

O Compilador *ajc* é a implementação oficial de AspectJ. É um compilador integrado com o ambiente de desenvolvimentod o Eclipse através do *plugin* AJDT (*AspectJ Development Tools*, uma extensão do JDT, *Java Development Tools*).

O *ajc*, por ser uma extensão do compilador do JDT, também possui suas principais características, ou seja, o *ajc* também é um compilador incremental, é capaz de gerar um programa executável a partir de código que possua erros, ignorando esses trechos para uma etapa posterior de compilação incremental.

O principal artigo descrevendo o funcionamento do *ajc* é o [HH04], onde é explanado o processo de *weaving* de *advices*.

### 2.1.5 abc

O *abc* é um compilador acadêmico que implementa a linguagem AspectJ. Ele possui algumas diferenças entre sua implementação e a do *ajc* [1], tal como não suportar os recursos de Java 5.

Os principais objetivos do *abc* são a extensibilidade do compilador, que possibilita a adição de novas construções à linguagem, de novas otimizações, etc, e a geração de código eficiente.

O abc utiliza o Polyglot[ref] como *front-end* da compilação, para a geração da árvore sintática, e o *Soot* [VRGH+00, VRHS+99] *como back-end*, para geração e otimização de código. O código é gerado a partir da árvore sintática para uma das representações intermediárias do *Soot*, a *Jimple* [VRH98].

Jimple é uma representação de bytecode tipada, compacta e que utiliza *3-address code*, ou seja, é possível saber o tipo de cada expressão ou variável, o número de instruções de *Jimple* é consideravelmente menor do que o de *bytecode*, porém mantendo o mesmo poder representativo, e toda instrução é o mais simples possível, a maioria sendo da forma x = y op z [S.M97].

## 2.2 Linhas de Produtos

*Linha de Produtos de Software* (do inglês *Software Product Lines*) [CN02] é um *framework* de processos que focam no desenvolvimento de uma família de produtos visando um mercado específico e baseados numa base comum de artefatos.

Numa linha de produtos, há uma arquitetura genérica que é comum a todos os produtos da linha; essa arquitetura é adaptada para a criação de um produto particular. Cada produto da linha é definido a partir de uma seleção de *features*, atributos que caracterizam as funcionalidades do produto. Essas seleções de Variabilidade num produto é o conjunto de pontos onde ele difere dos outros produtos da linha. Existem vários níveis de variabilidade: de código, de recursos, de arquitetura, etc. O tipo de variabilidade que é focado neste trabalho é a variabilidade de código.

Numa linha de produtos de jogos *Java ME*, a tecnologia mais usada pela indústria para tratar variabilidade de código é o uso de pré-processamento. Mais recentemente o uso de

[1] http://abc.comlab.ox.ac.uk/differences

AspectJ como mecanismo de inserção de variabilidades no código tem sido investigado pela academia [ANS+06].

### 2.2.1 Pré-processamento

Pré-processamento de código é uma técnica que permite especificar que determinados trechos de código somente serão incluídos numa compilação quando determinados parâmetros, normalmente especificados no processo de *build*, sejam verdadeiros.

Pré-processamento é comumente utilizado em programas *C* e *C++* mas com a necessidade de incluir ou não determinados trechos de código *Java ME* em aplicações móveis, o uso de pré-processamento virou a técnica padrão da indústria para tratamento de pontos de variação no código.

Na Listagem 2.1 é mostrado como o pré-processamento se apresenta no código fonte:

**Listagem 2.1** Exemplo de pré-processamento.

```
1 //#if device_graphics_canvas_nokiaui
2 public class MainCanvas extends FullCanvas {
3 //#elif device_graphics_canvas_midp2 || device_graphics_canvas_siemens
4 //# public class MainCanvas extends GameCanvas {
5 //#elif sku_id_sel
6 //# public class MainCanvas extends Canvas implements CommandListener {
7 //#else
8 //# public class MainCanvas extends Canvas {
9 //#endif
```

No exemplo acima, quando a especificação de um *build* tiver selecionado a *tag* após o *//#if*, o trecho de código logo abaixo da mesma entrará descomentado para compilação. Quando esta *tag* não estiver selecionada o código entre ela e o *//#elif* será comentado.

Através do uso de *tags* que representam as *features* do jogo, os desenvolvedores de jogos *Java ME* tratam as variações de código dos diferentes aparelhos de uma maneira que sacrifica a legibilidade do código em prol da performance, uma vez que pré-processamento não adiciona nenhum overhead no código final. Uma vez que cada *build* é composto por uma seleção de *features*, um *build* de um jogo *Java ME* é uma instância da linha de produtos.

## 2.3 *Bytecode*

*Bytecode* é o formato de código utilizado pela linguagem Java quando compilada. Cada *bytecode* possui exatamente um *byte*, e um programa Java compilado é uma uma sequência de instruções de *bytecode*. Vários trechos de *bytecode* serão mostrados neste trabalho.

A especificação completa do formato é apresentada em [LY99]. Para uma maior facilidade na leitura dos bytecodes dos programas mostrados neste trabalho, o *plugin* para o Eclipse do projeto ASM [Kul07] foi utilizado para obter uma representação legível dos bytecodes.

CAPÍTULO 3

# Análise de Tamanho

*"The Difficult is that which can be done immediately; the Impossible that which takes a little longer."*
—GEORGE SANTAYANA

O primeiro passo para analisar o *overhead* de AspectJ é quantificá-lo. Foi realizada uma comparação do tamanho dos *builds*, ou seja, dos arquivos *jar*, de ambas as versões do jogo, a original, que utiliza compilação condicional para tratar as variações, e a versão com grande parte das variações extraídas para aspectos. Visto que é padrão na indústria a obfuscação do código após a compilação, o tamanho comparado será o do *jar* gerado pelo obfuscador open source *ProGuard* [Pro07].

Visando atender o maior número de celulares, e assim obter um maior lucro, cada jogo é desenvolvido para uma família de celulares. Uma família de celulares é um agrupamento de celulares de um fabricante, que possuem características similares. Cada família possui um aparelho referência, onde, quando um jogo for desenvolvido e estiver rodando sem falhas neste aparelho, o mesmo *build* poderá ser usado nos outros aparelhos da família.

O jogo analisado foi desenvolvido para 15 famílias de celulares, abrangendo dezenas de aparelhos.

A Tabela 3.1 apresenta o tamanho, em *bytes*, de cada *build* do jogo original, com compilação condicional, compilado com o compilador da JDK, o *javac*, com o *ajc* e com o *abc*.

Um fato interessante que pode-se perceber, é que usando o compilador *abc*, todos os *builds* apresentaram um tamanho ligeiramente menor do que o compilado com o *javac*. Esse resultado se deve ao fato do *abc* utilizar o *Soot* tanto para geração quanto para otimização do código, e este possui uma série de otimizações que não estão presentes no compilador padrão da JDK. O tamanho dos builds com o *ajc*, entretanto, foi maior do que com o *javac*, este resultado se deve ao fato de que o compilador *ajc* não foi desenvolvido visando um compilador otimizado, e sim um compilador que pudesse ser integrado ao ambiente de desenvolvimento do Eclispe, tendo como prioridade funcionalidades como compilação incremental e não uma geração de código compacto e otimizado.

Já na versão do jogo que utiliza aspectos para inserção de variações, a diferença entre o *ajc* e o *abc* é maior, sendo, na média, de 1301 *bytes* (aumento de 1.5%), o que é uma diferença considerável no domínio. A Tabela 3.2 apresenta o tamanho dos builds com os compiladores *ajc* e *abc*.

O *abc* apresenta-se como claro vencedor na geração de código com aspectos, o que o leva a ser o compilador escolhido para a etapa de implementação deste trabalho. Porém, essa escolha

| *Build ID* | *javac* | *ajc* | *abc* |
|---|---|---|---|
| MOT1 | 72.922 | 73.012 | 72.512 |
| MOT2 | 85.350 | 85.425 | 84.939 |
| MOT3 | 72.957 | 73.049 | 72.544 |
| S40 | 65.339 | 65.448 | 64.913 |
| S40M2 | 73.023 | 73.117 | 72.563 |
| S40M2V3 | 72.524 | 72.614 | 72.115 |
| S60M1 | 85.587 | 85.681 | 85.190 |
| S60M2 | 84.506 | 84.592 | 84.106 |
| SAM1 | 66.570 | 66.689 | 66.246 |
| SAM2 | 80.543 | 80.667 | 80.251 |
| SE02 | 84.367 | 84.446 | 83.958 |
| SE03 | 72.604 | 72.698 | 72.146 |
| SE04 | 72.545 | 72.633 | 72.132 |
| SIEM3 | 73.267 | 73.355 | 72.854 |
| SIEM4 | 72.500 | 72.592 | 72.078 |
| Média | 75.640 | 75.735 | 75.236 |
| Diferença em relação ao javac | 0% | 0,1246% | -0,5338% |

**Tabela 3.1** Tamanho (em bytes) da versão original do jogo (sem aspectos)

não exclui o *ajc* da inspeção do bytecode gerado pelas construções usadas para implementar as variações. A contribuição deste trabalho no que diz respeito ao *ajc* será a identificação dos motivos pelo qual o código do mesmo se apresenta maior do que o *abc* e o *javac*.

A Tabela 3.3 apresenta os tamanhos dos *builds* da versão sem aspectos do jogo, compilada com o *abc*, visto que o mesmo apresentou melhor desempenho que o *javac*, junto com os tamanhos dos *builds* da versão com aspectos, compilada com o *abc* e com o *ajc*.

Como pode-se ver, a diferença média entre a versão original e a versão com aspectos compilada com o *abc* é de 8.416 bytes (aumento de 11%), o que é uma diferença que torna inviável o uso de AspectJ como mecanismo de inserção de variações de código.

No capítulo seguinte serão analisados os motivos por trás deste aumento, para serem propostas otimizações que visem reduzir esse *overhead*.

| *Build ID* | *ajc* | *abc* |
|---|---|---|
| MOT1 | 82.832 | 81.534 |
| MOT2 | 95.258 | 93.985 |
| MOT3 | 87.410 | 85.956 |
| S40 | 74.367 | 72.950 |
| S40M2 | 81.951 | 80.668 |
| S40M2V3 | 81.744 | 80.526 |
| S60M1 | 95.695 | 94.085 |
| S60M2 | 94.770 | 93.122 |
| SAM1 | 73.178 | 72.024 |
| SAM2 | 85.688 | 84.544 |
| SE02 | 92.900 | 91.719 |
| SE03 | 81.610 | 80.433 |
| SE04 | 81.527 | 80.383 |
| SIEM3 | 83.012 | 81.754 |
| SIEM4 | 82.356 | 81.101 |
| Média | 84.953 | 83.652 |
| Diferença em relação ao ajc | 0% | -1,5314% |

**Tabela 3.2** Tamanho (em bytes) da versão do jogo com aspectos

| *Build ID* | Original (*abc*) | Com aspectos (*ajc*) | Com aspectos (*abc*) |
|---|---|---|---|
| MOT1 | 72.512 | 82.832 | 81.534 |
| MOT2 | 84.939 | 95.258 | 93.985 |
| MOT3 | 72.544 | 87.410 | 85.956 |
| S40 | 64.913 | 74.367 | 72.950 |
| S40M2 | 72.563 | 81.951 | 80.668 |
| S40M2V3 | 72.115 | 81.744 | 80.526 |
| S60M1 | 85.190 | 95.695 | 94.085 |
| S60M2 | 84.106 | 94.770 | 93.122 |
| SAM1 | 66.246 | 73.178 | 72.024 |
| SAM2 | 80.251 | 85.688 | 84.544 |
| SE02 | 83.958 | 92.900 | 91.719 |
| SE03 | 72.146 | 81.610 | 80.433 |
| SE04 | 72.132 | 81.527 | 80.383 |
| SIEM3 | 72.854 | 83.012 | 81.754 |
| SIEM4 | 72.078 | 82.356 | 81.101 |
| Média | 75.236 | 84.953 | 83.652 |
| Diferença em relação ao original | 0% | 12,9153% | 11,1861% |

**Tabela 3.3** Comparação (em bytes) da versão original com a versão com aspectos

CAPÍTULO 4

# Inspeção individual

*"Millions long for immortality who don't know what to do with themselves on a rainy Sunday afternoon."*

—SUSAN ERTZ

A inspeção das construções de AspectJ utilizadas nos aspectos do jogo tem por objetivo identificar porque os *builds* que utilizam AspectJ para inserir as variações de código apresentam essa diferença considerável de tamanho, para no capítulo seguinte serem propostas melhorias para diminuir essa diferença.

São analisados os *bytecodes* gerados por aspectos que utilizam as construções de AspectJ presentes no jogo analisado e pelas classes afetadas por estes aspectos. Por motivos de claridade os exemplos serão minimalísticos, contendo cada um apenas o mínimo de construções de AspectJ necessárias para cada tipo de variação do jogo.

## 4.1 Classe, Interface e Aspecto vazios

Visto que serão analisados vários exemplos de construções de AspectJ, é interessante que primeiro sejam identificadas as diferenças entre a compilação de aspectos, classes e interfaces vazias em ambos os compiladores. Uma inspeção do *bytecode* gerado por ambos os compiladores com esses tipos vazios já pode identificar padrões de ineficiência na geração de código para exemplos mais elaborados.

O primeiro código analisado é o da compilação da classe vazia da Listagem 4.1.

**Listagem 4.1** Classe Vazia.

```
1 public class BlankClass {
2 }
```

O *bytecode* gerado pelo compilador *ajc* é mostrado na Listagem 4.2:

**Listagem 4.2** *Bytecode* de uma classe vazia compilada pelo *ajc*.

```
1  public class BlankClass {
2    public <init>() : void
3     L0 (0)
4      LINENUMBER 2 L0
5      ALOAD 0: this
6      INVOKESPECIAL Object.<init>() : void
7      RETURN
8     L1 (4)
9      LOCALVARIABLE this BlankClass L0 L1 0
10     MAXSTACK = 1
```

```
11      MAXLOCALS = 1
12  }
```

O código gerado pelo *ajc* para uma classe vazia apresenta apenas o construtor implícito que existe em toda classe, quando um não é declarado explicitamente pelo programador. Foi verificado que este código é idêntico ao gerado pelo compilador *javac*.

O *bytecode* desta mesma classe gerado pelo *abc* é apresentado na Listagem 4.3:

**Listagem 4.3** *Bytecode* da um classe vazia compilada pelo *abc*.

```
1   public class BlankClass {
2
3     public <init>() : void
4      L0 (0)
5       LINENUMBER 2 L0
6       ALOAD 0
7      L1 (2)
8       LINENUMBER 2 L1
9       INVOKESPECIAL Object.<init>() : void
10      RETURN
11      MAXSTACK = 1
12      MAXLOCALS = 1
13
14    static <clinit>() : void
15     L0 (0)
16      LINENUMBER 2 L0
17      RETURN
18      MAXSTACK = 0
19      MAXLOCALS = 0
20  }
```

Diferentemente do *ajc* e do *javac*, o compilador *abc* gerou um método <clinit>() vazio, quando a existência do mesmo não era necessária, uma vez que este método é utilizado para a inicialização de atributos estáticos, que não se apresentam nesta classe.

O código da Listagem 4.4 foi utilizado para verificar o *bytecode* de uma interface vazia, gerado por ambos os compiladores.

**Listagem 4.4** Interface vazia.

```
1   public interface BlankInterface {
2   }
```

Com o *ajc*, o classfile da interface apresenta um corpo vazio dentro da declaração da mesma, como podemos ver na Listagem 4.5:

**Listagem 4.5** *Bytecode* da uma interface vazia compilada pelo *ajc*.

```
1   public abstract interface BlankInterface {
2   }
```

Já o *abc*, novamente cria um método <clinit>() vazio desnecessariamente, uma vez que a interface não possui nenhum atributo estático, como podemos ver na Listagem 4.6.

**Listagem 4.6** *Bytecode* da uma interface vazia compilada pelo *abc*.

```
1   public abstract interface BlankInterface {
2     static <clinit>() : void
3      L0 (0)
4       LINENUMBER 2 L0
5       RETURN
```

```
6       MAXSTACK = 0
7       MAXLOCALS = 0
8   }
```

O código da Listagem 4.7 foi utilizado para verificar o *bytecode* gerado de um aspecto vazio.

**Listagem 4.7** Aspecto vazio.

```
1   public aspect BlankAspect {
2   }
```

O *bytecode* gerado pelo *ajc* é apresentado na Listagem 4.8:

**Listagem 4.8** *Bytecode* da um aspecto vazio compilado pelo *ajc*.

```
1   public class BlankAspect {
2
3     ATTRIBUTE org.aspectj.weaver.WeaverState : unknown
4
5     ATTRIBUTE org.aspectj.weaver.SourceContext : unknown
6
7     ATTRIBUTE org.aspectj.weaver.Aspect : unknown
8
9     ATTRIBUTE org.aspectj.weaver.WeaverVersion : unknown
10
11    private static Throwable ajc$initFailureCause
12
13    public final static BlankAspect ajc$perSingletonInstance
14
15    static <clinit>() : void
16      TRYCATCHBLOCK L0 L1 L2 Throwable
17     L0 (0)
18      LINENUMBER 2 L0
19      INVOKESTATIC BlankAspect.ajc$postClinit() : void
20     L1 (2)
21      GOTO L3
22     L2 (4)
23      ASTORE 0
24      ALOAD 0
25      PUTSTATIC BlankAspect.ajc$initFailureCause : Throwable
26     L3 (8)
27      RETURN
28      MAXSTACK = 1
29      MAXLOCALS = 1
30
31    public <init>() : void
32     L0 (0)
33      LINENUMBER 2 L0
34      ALOAD 0: this
35      INVOKESPECIAL Object.<init>() : void
36      RETURN
37     L1 (4)
38      LOCALVARIABLE this BlankAspect L0 L1 0
39      MAXSTACK = 1
40      MAXLOCALS = 1
41
42    public static aspectOf() : BlankAspect
43    ATTRIBUTE org.aspectj.weaver.AjSynthetic : unknown
44     L0 (0)
45      LINENUMBER 1 L0
46      GETSTATIC BlankAspect.ajc$perSingletonInstance : BlankAspect
47      IFNONNULL L1
48      NEW NoAspectBoundException
49      DUP
```

```
50      LDC "BlankAspect"
51      GETSTATIC BlankAspect.ajc$initFailureCause : Throwable
52      INVOKESPECIAL NoAspectBoundException.<init>(String,Throwable) : void
53      ATHROW
54     L1 (9)
55      GETSTATIC BlankAspect.ajc$perSingletonInstance : BlankAspect
56      ARETURN
57      MAXSTACK = 4
58      MAXLOCALS = 0
59
60    public static hasAspect() : boolean
61    ATTRIBUTE org.aspectj.weaver.AjSynthetic : unknown
62     L0 (0)
63      LINENUMBER 1 L0
64      GETSTATIC BlankAspect.ajc$perSingletonInstance : BlankAspect
65      IFNULL L1
66      ICONST_1
67      IRETURN
68     L1 (5)
69      ICONST_0
70      IRETURN
71      MAXSTACK = 1
72      MAXLOCALS = 0
73
74    private static ajc$postClinit() : void
75    ATTRIBUTE org.aspectj.weaver.AjSynthetic : unknown
76     L0 (0)
77      LINENUMBER 1 L0
78      NEW BlankAspect
79      DUP
80      INVOKESPECIAL BlankAspect.<init>() : void
81      PUTSTATIC BlankAspect.ajc$perSingletonInstance : BlankAspect
82      RETURN
83      MAXSTACK = 2
84      MAXLOCALS = 0
85  }
```

Podemos perceber que são criados 2 atributos, 5 métodos, e 4 ATTRIBUTES no *classfile* resultante. Os 4 ATTRIBUTES no início do *classfile* não são atributos da classe, e sim meta-dados que a especificação da JVM prevê que compiladores podem criar para adicionar informações adicionais aos *classfiles*.

O atributo ajc$perSingletonInstance representa a instância *Singleton* do aspecto e o atributo ajc$initFailureCause guarda o motivo da falha da inicialização da classe do aspecto, caso algum problema aconteça durante a inicialização do mesmo.

O método <clinit>() é o método de inicialização de classe, o que esse método faz na classe do aspecto é chamar o método ajc$postClinit(), que por sua vez cria a instância *Singleton* do aspecto e a guarda no atributo ajc$perSingletonInstance. O método <init>() é o método padrão de Java que contém o corpo do construtor de cada classe, visto que esse aspecto não possui construtor, um corpo padrão foi gerado pelo *ajc*. Por fim, os métodos aspectOf() e hasAspect() retornam, respectivamente, a instância do aspecto e um booleano que indica se o aspecto possui ou não uma instância diferente de null.

Como pode-se perceber, um aspecto completamente vazio gerou um *classfile* de tamanho considerável, com 2 atributos estáticos, 3 métodos e 4 ATTRIBUTES, desconsiderando os métodos padrões de inicialização de classe e de objeto de Java.

O mesmo aspecto compilado com o *abc* gera o bytecode apresentado na Listagem 4.9.

**Listagem 4.9** *Bytecode* da um asoecto vazio compilada pelo *abc*.

```
1  public class BlankAspect {
2
3    public final static BlankAspect abc$perSingletonInstance
4
5    private static Throwable abc$initFailureCause
6
7    public <init>() : void
8     L0 (0)
9      LINENUMBER 2 L0
10     ALOAD 0
11    L1 (2)
12     LINENUMBER 2 L1
13     INVOKESPECIAL Object.<init>() : void
14     RETURN
15     MAXSTACK = 1
16     MAXLOCALS = 1
17
18   public static aspectOf() : BlankAspect throws NoAspectBoundException
19     GETSTATIC BlankAspect.abc$perSingletonInstance : BlankAspect
20     ASTORE 0
21     ALOAD 0
22     IFNULL L0
23     ALOAD 0
24     ARETURN
25    L0 (6)
26     NEW NoAspectBoundException
27     DUP
28     LDC "BlankAspect"
29     GETSTATIC BlankAspect.abc$initFailureCause : Throwable
30     INVOKESPECIAL NoAspectBoundException.<init>(String,Throwable) : void
31     ATHROW
32     MAXSTACK = 4
33     MAXLOCALS = 1
34
35   public static hasAspect() : boolean
36     GETSTATIC BlankAspect.abc$perSingletonInstance : BlankAspect
37     IFNULL L0
38     ICONST_1
39     IRETURN
40    L0 (4)
41     ICONST_0
42     IRETURN
43     MAXSTACK = 1
44     MAXLOCALS = 0
45
46   static <clinit>() : void
47     TRYCATCHBLOCK L0 L1 L2 Throwable
48    L0 (0)
49     INVOKESTATIC BlankAspect.abc$postClinit() : void
50    L1 (2)
51     GOTO L3
52    L2 (4)
53     PUTSTATIC BlankAspect.abc$initFailureCause : Throwable
54    L3 (6)
55     LINENUMBER 2 L3
56     RETURN
57     MAXSTACK = 1
58     MAXLOCALS = 0
59
60   private static abc$postClinit() : void
61     NEW BlankAspect
62     DUP
63     INVOKESPECIAL BlankAspect.<init>() : void
64     PUTSTATIC BlankAspect.abc$perSingletonInstance : BlankAspect
```

```
65      RETURN
66      MAXSTACK = 2
67      MAXLOCALS = 0
68  }
```

Como pode ser observado, o *bytecode* gerado *abc* possui campos e métodos equivalentes aos gerados pelo *ajc*, sem, entretanto, possuir qualquer meta-dado adicional na forma de ATTRIBUTES.

O tamanho do *classfile* para o aspecto gerado pelo ajc foi de 1126 *bytes*, contra 743 *bytes* do compilado pelo *abc*. Por motivos de brevidade e simplificação da leitura do código gerado, estes métodos e atributos do aspecto não serão mais exibidos, exceto quando o uso de alguma construção implique numa modificação dos mesmos. O código dos métdos <init>() e <clinit>() também serão omitidos, respeitando a mesma regra de exceção dos métodos dos aspectos.

## 4.2 Inter-type Declarations

### 4.2.1 Atributos

A primeira construção de AspectJ analisada é a introdução de atributos via *Inter-type Declarations*, da forma:

- [*Modifiers*] Type OnType.Id;
- [*Modifiers*] Type OnType.Id = Expression;

A principal diferença entre uma *Inter-type Declaration* de atributos para uma declaração de atributos simples de Java é o tipo alvo do atributo, ou seja, onde ele será inserido. Na gramática acima representado por OnType.

Ambas as formas foram identificadas na análise feita pelo projeto *FLiP*, portanto ambas serão inspecionadas.

#### 4.2.1.1 Sem Inicialização

Os código das duas Listagens 4.10 e 4.11 abaixo apresentam, respectivamente, uma classe sem nenhum atributo ou método, e um aspecto que introduz um atributo nesta classe, sem inicializá-lo.

**Listagem 4.10** Classe vazia onde será inserido um atributo sem inicialização.

```
1  public class BlankClass {
2  }
```

**Listagem 4.11** Aspecto que insere um atributo em uma classe, sem inicializá-lo.

```
1  public aspect AttributeAspect {
2          public int BlankClass.x;
3  }
```

A Listagem 4.12 apresenta o *bytecode* da classe BlankClass, após a compilação com o *ajc*:

**Listagem 4.12** *Bytecode* da classe após a inserção do atributo sem inialização (*ajc*).

```
1  public class BlankClass {
2    ATTRIBUTE org.aspectj.weaver.WeaverState : unknown
3
4    ATTRIBUTE org.aspectj.weaver.WeaverVersion : unknown
5
6    public int x
7
8    public <init>() : void
9     L0 (0)
10     LINENUMBER 3 L0
11     ALOAD 0: this
12     INVOKESPECIAL Object.<init>() : void
13     ALOAD 0: this
14     INVOKESTATIC
15          AttributeAspect.ajc$interFieldInit$AttributeAspect$BlankClass$x(BlankClass) : void
16     RETURN
17    L1 (6)
18     LOCALVARIABLE this BlankClass L0 L1 0
19     MAXSTACK = 1
20     MAXLOCALS = 1
21 }
```

Foi observado que o atributo foi inserido na classe alvo, e que além da adição do mesmo, foi criada uma chamada a um método estático da classe do aspecto. No *classfile* do aspecto, o uso da construção *Inter-type Field Declaration*, resultou na criação de 3 métodos e dois 1 novo ATTRIBUTE:

**Listagem 4.13** *Bytecode* dos dispatchers e do método de inicialização do atributo (*ajc*)

```
1   ATTRIBUTE org.aspectj.weaver.TypeMunger : unknown
2
3   public static ajc$interFieldInit$AttributeAspect$BlankClass$x(BlankClass) : void
4   ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
5    L0 (0)
6     LINENUMBER 4 L0
7     RETURN
8     LOCALVARIABLE ajc$this_ BlankClass L0 L0 0
9     MAXSTACK = 0
10    MAXLOCALS = 1
11
12  public static ajc$interFieldGetDispatch$AttributeAspect$BlankClass$x(BlankClass) : int
13  ATTRIBUTE org.aspectj.weaver.EffectiveSignature : unknown
14    ALOAD 0
15    GETFIELD BlankClass.x : int
16    IRETURN
17    MAXSTACK = 1
18    MAXLOCALS = 1
19
20  public static ajc$interFieldSetDispatch$AttributeAspect$BlankClass$x(BlankClass,int) : void
21  ATTRIBUTE org.aspectj.weaver.EffectiveSignature : unknown
22    ALOAD 0
23    ILOAD 1
24    PUTFIELD BlankClass.x : int
25    RETURN
26    MAXSTACK = 2
27    MAXLOCALS = 2
```

Percebe-se que cada método possui um meta-dado na forma de ATTRIBUTE. O primeiro método, o que é invocado no construtor da classe BlankClass possui no corpo apenas um *statement* return, ou seja, esse método efetivamente não realizada nada. Os outros 2 métodos criados são *dispatchers* para o acesso e atribuição de valor ao atributo adicionado à classe BlankClass.

Todo o acesso a esse atributo é feito por estes 2 métodos.

Com o *abc*, entretanto, a única adição à classe BlankClass é o atributo em si, como podemos ver na Listagem 4.14.

**Listagem 4.14** *Bytecode* da classe após a inserção do atributo sem inialização (*abc*).

```
1  public class BlankClass {
2
3    public int x
4
5    /* .. */
6  }
```

O *classfile* de BlankClass não possui nenhuma referência ao aspecto AttributeAspect, cujo *classfile* é idêntico ao do modelo apresentado no início desta seção.

#### 4.2.1.2 Sem inicialização

Os código das Listagens 4.15 e 4.16 apresentam, respectivamente, uma classe sem nenhum atributo ou método, e um aspecto que introduz um atributo nesta classe e o inicializa.

**Listagem 4.15** Classe vazia onde será inserido um atributo com inicialização.

```
1  public class BlankClass {
2  }
```

**Listagem 4.16** Aspecto que insere um atributo em uma classe, com inicialização.

```
1  public aspect AttributeAspect {
2          public int BlankClass.x = 1;
3  }
```

O *bytecode* do *classfile* de BlankClass gerado pelo *ajc* é idêntico ao que foi gerado no aspecto que introduz o mesmo campo, porém sem inicializá-lo. Já o *bytecode* do aspecto apresentou um método diferente, como podemos ver na Listagem 4.17.

**Listagem 4.17** *Bytecode* do método de inicialização do atributo (*ajc*).

```
1   public static ajc$interFieldInit$AttributeAspect$BlankClass$x(BlankClass) : void
2   ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3    L0 (0)
4     LINENUMBER 4 L0
5     ALOAD 0: ajc$this_
6     ICONST_1
7     INVOKESTATIC AttributeAspect.ajc$interFieldSetDispatch$AttributeAspect$BlankClass$x(
8           BlankClass,int) : void
9     RETURN
10   L1 (5)
11    LOCALVARIABLE ajc$this_ BlankClass L0 L1 0
12    MAXSTACK = 2
13    MAXLOCALS = 1
```

O método que não realizava nenhuma operação na versão do aspecto sem a inicialização do atributo inter-type agora realiza a tarefa de atribuir o valor da expressão de inicialização ao mesmo. Na linha 6 a instrução ICONST_1 coloca o valor '1' na pilha, para, em seguida, o o método *dispatcher* de modificação do valor do atributo o retirar para usar como parâmetro para efetuar a atribuição.

Percebe-se um grande nível de redireção de código, algo que poderia ser feito diretamente na classe BlankClass requeriu 2 chamadas de método para ser realizado.

Com o *abc*, onde sem a inicialização a class BlankClass não apresentava nenhuma referência ao aspecto, agora possui uma chamada estática no seu construtor, como podemos ver na Listagem 4.18.

**Listagem 4.18** *Bytecode* do construtor da classe onde o atributo foi inserido (*abc*).

```
1  public <init>() : void
2   L0 (0)
3    ALOAD 0
4   L1 (2)
5    INVOKESPECIAL Object.<init>() : void
6    ALOAD 0
7    INVOKESTATIC AttributeAspect.fieldinit$24(BlankClass) : void
8    RETURN
9    MAXSTACK = 1
10   MAXLOCALS = 1
```

No *classfile* do aspecto foram criados os 2 métodos, mostrados na Listagem 4.19.

**Listagem 4.19** Bytecode dos métodos de inicialização do atributo (abc).

```
1  public static fieldinit$24(BlankClass) : void
2    ALOAD 0
3    ALOAD 0
4    INVOKESTATIC AttributeAspect.init$x$18(BlankClass) : int
5    PUTFIELD BlankClass.x : int
6    RETURN
7    MAXSTACK = 2
8    MAXLOCALS = 1
9
10 public static init$x$18(BlankClass) : int
11  L0 (0)
12   ICONST_1
13  L1 (2)
14   IRETURN
15   MAXSTACK = 1
16   MAXLOCALS = 1
```

O método init$x$18()funciona como *placeholder* do código da expressão de inicialização, retornando o valor da mesma. O método invocado por BlankClass, fieldinit$24(), tem a mesma função do método criado pelo *ajc*, inicializar o atributo, porém, diferentemente do *ajc*, o *abc* não gera *dispatchers* para o acesso e modificação do atributo, o método de inicialização faz a modificação diretamente no atributo.

#### 4.2.1.3 Constantes

Constantes são um caso especial de *Inter-type Declaration*, no código Java original elas são atributos de tipos primitivos, com os modificadores static final. Constantes são amplamente utilizadas em *Java ME* porque cada ocorrência de uso de uma costante cujo valor é conhecido em tempo de compilação é substituída pelo valor literal da constante. No final o código não possuirá um atributo real, nem instruções de acesso ao valor deste atributo, o que é bastante vantajoso em termos de espaço.

Um problema que ocorre com o uso de *Inter-type Declarations* para inserir constantes é que elas não serão realmente constantes, os compiladores de AspectJ inserem um atributo es-

tático na classe, ao invés de substituírem as ocorrências pelo valor literal. Além de aumentar o tamanho do código, o que acontecia no código dos jogos é que algumas constantes eram utilizadas em construções *switch* em vários cases. A especificação de Java dita que valores de *cases* devem ser literais conhecidos em tempo de compilação.

Foi identificada uma maneira alternativa de inserção de constantes em classes com AspectJ que resolve ambos os problemas. A solução consiste na criação de uma interface interna no aspecto, que vai conter as constantes e fazer com que a classe que contia a constante implemente essa interface, através da construção declare parents de AspectJ.

A classe e o aspecto das Listagens 4.20 e 4.21 demonstram o uso deste artifício:

**Listagem 4.20** Classe onde será inserida uma constante.

```
1  package constant;
2
3  public class BlankClass {
4
5          public int constant() {
6                  return CONSTANT;
7          }
8  }
```

**Listagem 4.21** Aspecto que insere uma constante numa classe Java.

```
1  package constant;
2
3  public aspect ConstantAspect {
4          public interface Constants {
5                  public static final int CONSTANT = 1;
6          }
7          declare parents : BlankClass implements Constants;
8  }
```

Este código compilado com o *ajc* não adiciona nenhum atributo a BlankClass, e o código do método constant() é mostrado na Listagem 4.22.

**Listagem 4.22** *Bytecode* do método constant() (*ajc*).

```
1    public constant() : int
2    ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3     L0 (0)
4      LINENUMBER 6 L0
5      ICONST_1
6      IRETURN
7     L1 (3)
8      LOCALVARIABLE this BlankClass L0 L1 0
9      MAXSTACK = 1
10     MAXLOCALS = 1
```

Como pode ser observado, o uso da constante foi substituído por seu valor literal, através da instrução ICONST_1. O *classfile* do aspecto segue o modelo apresentado no início desta seção, e um terceiro *classfile* é gerado, o da interface que contém a constante. Esse *classfile* é mostrado na Listagem 4.23.

**Listagem 4.23** *Bytecode* da interface declarada no aspecto (*ajc*).

```
1  public abstract interface ConstantAspect$Constants {
2    ATTRIBUTE org.aspectj.weaver.WeaverState : unknown
3
```

```
4   ATTRIBUTE org.aspectj.weaver.WeaverVersion : unknown
5   INNERCLASS ConstantAspect$Constants ConstantAspect Constants 1545
6
7   public final static int CONSTANT = 1
8 }
```

Compilando com o *abc*, o *classfile* de BlankClass não possui nenhum atributo, e o código do método contant() é idêntico ao gerado pelo *ajc*. O código do aspecto também segue o modelo apresentado anteriormente para aspectos vazios, e o *bytecode* da interface apenas difere no gerado pelo *ajc* pela ausência de meta-dados e pela presença de um <clinit>() com corpo vazio.

### 4.2.2 Extensão de Classe

AspectJ provê uma construção que permite alterar a hierarquia de classes de um programa, fazendo uma classe estender outra, a sintaxe desta construção é a seguinte:

- declare parents: TypePattern extends Type;

A semântica dela é que classes que casam com o padrão TypePattern vão estender a classe especificada por Type.

As Listagens 4.24, 4.25 e 4.26 apresentam, respectivamente, duas classes vazias, e um aspecto que faz com que a classe BlankClass estenda a classe BlankSuperclass.

**Listagem 4.24** Classe vazia cuja hierarquia será alterada para estender uma outra classe.

```
1 package hierarchy.classextension;
2
3 public class BlankClass {
4
5 }
```

**Listagem 4.25** Classe vazia que será a classe pai.

```
1 package hierarchy.classextension;
2
3 public class BlankSuperclass {
4
5 }
```

**Listagem 4.26** Aspecto que faz uma classe vazia estender outra classe vazia.

```
1 package hierarchy.classextension;
2
3 public aspect HierarchyChanger {
4         declare parents : BlankClass extends BlankSuperclass;
5 }
```

Em ambos os compiladores, o *bytecode* gerado é essencialmente idêntico ao gerado pelo compilador do JDK caso a declaração de extensão estivesse em BlankClass diretamente. A diferença entre o código das classes Java gerados pelo *ajc* e pelo *abc* é que no último cada classe contém um método <clinit>() vazio, o que não ocorre com o *ajc*. O classfile dos aspectos novamente é idêntico ao modelo apresentado no início da seção.

### 4.2.3 Implementação de Interface

AspectJ provê uma construção que permite alterar a hierarquia de classes de um programa, fazendo uma classe estender outra, a sintaxe desta construção é a seguinte:

- declare parents: TypePattern implements TypeList;

A semântica dela é que classes que casam com o padrão TypePattern vão implementar as interfaces especificadas por TypeList.

O código das Listagens 4.27, 4.28 e 4.29 apresentam, respectivamente, uma classe vazia, uma interface vazia e um aspecto que faz com que a classe vazia implemente a interface vazia.

**Listagem 4.27** Classe vazia cuja hierarquia será alterada para implementar uma interface.

```
1 package hierarchy.interfaceimplementaion;
2
3 public class BlankClass {
4 }
```

**Listagem 4.28** Interface vazia.

```
1 package hierarchy.interfaceimplementaion;
2
3 public interface BlankInterface {
4 }
```

**Listagem 4.29** Aspecto que faz uma classe vazia estender outra classe vazia.

```
1 package hierarchy.interfaceimplementaion;
2
3 public aspect HierarchyChanger {
4         declare parents : BlankClass implements BlankInterface;
5 }
```

O mesmo padrão apresentado pela extensão de classe foi observado na implementação da interface, o *classfile* gerado pelo *ajc* era essencialmente idêntico à uma versão deste exemplo sem o aspecto (BlankClass explicitamente implementando BlankInterface). Já com o *abc* apresenta novamente a criação de <clinit>() vazio na classe BlankClass.

### 4.2.4 Métodos

AspectJ provê uma construção que permite adicionar um método numa classe alvo, tanto métodos concretos como abstratos podem ser inseridos. A sintaxe para cada caso é a seguinte:

- $[Modifiers]\ TypeOnType.Id(Formals)\ [ThrowsClause]$ { Body }
- abstract $[Modifiers]\ TypeOnType.Id(Formals)\ [ThrowsClause]$;

O *FLiP* identificou apenas ocorrências do primeiro caso, de inserções concretas de método em classes concretas. Nas Listagens 4.30 e 4.31 podemos ver um exemplo do uso desta construção.

**Listagem 4.30** Classe vazia onde será inserido um método.

```
1 public class BlankClass {
2 }
```

**Listagem 4.31** Aspecto que insere um método concreto em uma classe vazia.

```
1 public aspect IntertypeMethod {
2         public int BlankClass.x() {
3                 return 0;
4         }
5 }
```

O que o aspecto IntertypeMethod faz é adicionar na classe BlankClass um método público chamado x(), que retorna um inteiro.

Compilando esse exemplo com o *ajc* temos o método apresentado na Listagem 4.32 adicionado na classe BlankClass.

**Listagem 4.32** *Bytecode* do método x() na classe BlankClass (*ajc*).

```
1 public x() : int
2   L0 (0)
3   LINENUMBER 1 L0
4   ALOAD 0
5   INVOKESTATIC IntertypeMethod.ajc$interMethod$IntertypeMethod$BlankClass$x(BlankClass) : int
6   IRETURN
7   MAXSTACK = 1
8   MAXLOCALS = 1
```

Percebe-se novamente aqui a redireção para o aspecto, através da chamada a um método estático da classe do aspecto IntertypeMethod, que efetivamente executa o corpo de x(). Na Listagem 4.33 são mostrados os dois métodos que são criados no *classfile* do aspecto.

**Listagem 4.33** *Bytecode* dos métodos de implementação e *dispatcher* no aspecto (*ajc*).

```
1  public static ajc$interMethod$IntertypeMethod$BlankClass$x(BlankClass;) : int
2  ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3  ATTRIBUTE org.aspectj.weaver.EffectiveSignature : unknown
4   L0 (0)
5    LINENUMBER 5 L0
6    ICONST_0
7    IRETURN
8   L1 (3)
9    LOCALVARIABLE ajc$this_ BlankClass; L0 L1 0
10   MAXSTACK = 1
11   MAXLOCALS = 1
12
13 public static ajc$interMethodDispatch1$IntertypeMethod$BlankClass$x(BlankClass;) : int
14 ATTRIBUTE org.aspectj.weaver.EffectiveSignature : unknown
15   ALOAD 0
16   INVOKEVIRTUAL BlankClass.x() : int
17   IRETURN
18   MAXSTACK = 1
19   MAXLOCALS = 1
```

O primeiro método contém o corpo real de x(), e o segundo método é um *dispatcher* do método x(), ou seja, quando, no código, o método x() for invocado, o método real a ser invocado será o *dispatcher*, que por sua vez chamará o método que se encontra na classe BlankClass, que por sua vez chama o método que contém a sua implementação no aspecto.

Percebe-se um alto nível de redireção, toda chamada do método x() vai resultar em 3 chamadas de métodos aninhadas.

Com o *abc*, perecebe-se que essa redireção também está presente, apesar de acontecer em um nível menor, o corpo do método x() no *classfile* de BlankClass é mostrado abaixo:

**Listagem 4.34** *Bytecode* do método x() na classe BlankClass (*abc*).

```
1 public x() : int
2   ALOAD 0
3   INVOKESTATIC IntertypeMethod.x(BlankClass) : int
4   IRETURN
5   MAXSTACK = 1
6   MAXLOCALS = 1
```

Ele também chama um método do aspecto, a diferença do *abc* para o *ajc* é que o método chamado já é o método que contém a implementação de x():

**Listagem 4.35** *Bytecode* do método de implementação no aspecto IntertypeMethod (abc).

```
1 public static x(BlankClass) : int
2  L0 (0)
3   LINENUMBER 5 L0
4   ICONST_0
5  L1 (2)
6   LINENUMBER 5 L1
7   IRETURN
8   MAXSTACK = 1
9   MAXLOCALS = 1
```

Toda ocorrência de chamada ao método x() da classe BlankClass é feita diretamente na própria classe, sem o uso de um *dispatcher*.

## 4.3 *Advices*

*Advices* são construções de AspectJ que permitem que um bloco de código seja executado antes (*before*), depois (*after*) ou no lugar (*around*) de um determinado ponto na execução de um programa, estes pontos são chamados de *join points*. AspectJ permite a criação de expressões que permitem a captura destes pontos no programa, estas expressões são chamadas de *pointcuts*.

A sintaxe do uso de *advices* é apresentada na gramática abaixo:

- $[strictfp]\,AdviceSpec\,[throwsTypeList]$ : Pointcut Body

onde AdviceSpec é um dos abaixo:

- before( Formals )
- after( Formals ) returning [ ( Formal ) ]
- after( Formals ) throwing [ ( Formal ) ]
- after( Formals )
- Type around( Formals )

O projeto *FLiP* identificou a necessidade do uso de advices para a inserção de alguns pontos de variação dentro de métodos. Portanto, código resultante do uso de advices, usando os *pointcuts* identificados como necessários pelo *FLiP*, será analisado para a identificação do *overhead* adicionado pela utilização dos mesmos.

Durante a análise, foi observado que o ponto onde o código do *advice* vai ser inserido antes, depois, ou no lugar não influenciou a maneira da inserção deste código. Por exemplo, a inserção do código de um *advice after call* se distingue de um *advice after set* apenas pelo *join point alvo*.

Logo, será mostrado um exemplo de cada tipo de *advice*, *after*, *before* e *around*.

### 4.3.1 Acesso a atributos privados

Uma vez que o uso de *advices* permite (quando o aspecto é declarado como *privileged*) o acesso a atributos ou métodos privados da classe exposta por this() ou target(), existe um *overhead* associado ao acesso a estes membros privados. Uma vez que eles são privados, não há como acessálos diretamente, portanto os compiladores criam métodos *accessors* para acessá-los.

**Listagem 4.36** Classe com atributo e método privados.

```
1  public class ClassWithMethods {
2          private int x;
3
4          private void someMethod() {
5          }
6          /*
7           * Outros métodos omitidos
8           */
9  }
```

Utilizando a classe da Listagem 4.36 para exemplificar esse *overhead*, se tivéssemos *advices* em aspectos *privileged* que acessassem o atributo x e o método somemethod(), haveria a criação dos seguintes métodos pelo compilador ajc:

**Listagem 4.37** *Bytecode* dos *accessors* (*ajc*).

```
1   public static ajc$privFieldGet$Aspect$ClassWithMethods$x(ClassWithMethods) : int
2    L0 (0)
3     LINENUMBER 1 L0
4     ALOAD 0
5     GETFIELD ClassWithMethods.x : int
6     IRETURN
7     MAXSTACK = 1
8     MAXLOCALS = 1
9
10  public static ajc$privFieldSet$Aspect$ClassWithMethods$x(ClassWithMethods,int) : void
11   L0 (0)
12    LINENUMBER 1 L0
13    ALOAD 0
14    ILOAD 1
15    PUTFIELD ClassWithMethods.x : int
16    RETURN
17    MAXSTACK = 2
18    MAXLOCALS = 2
19
20  public ajc$privMethod$Aspect$ClassWithMethods$someMethod() : void
21   L0 (0)
22    LINENUMBER 1 L0
```

```
23      ALOAD 0
24      INVOKESPECIAL ClassWithMethods.someMethod() : void
25      RETURN
26      MAXSTACK = 1
27      MAXLOCALS = 1
```

Na Listagem 4.37 os dois primeiros métodos são métodos *accessors* do atributo x, para que o advice poder acessar seu valor ou modificá-lo. O último método é um *accessor* para o método someMethod(), para que o advice também possa invocá-lo.

O *abc* também cria métodos accessors para o acesso de membros privados, como podemos observar na Listagem 4.38:

**Listagem 4.38** *Bytecode* dos *accessors* (*abc*).

```
1  public get$accessor$x$16() : int
2    ALOAD 0
3    GETFIELD ClassWithMethods.x : int
4    IRETURN
5    MAXSTACK = 1
6    MAXLOCALS = 1
7
8  public set$accessor$x$17(int) : int
9    ALOAD 0
10   ILOAD 1
11   PUTFIELD ClassWithMethods.x : int
12   ILOAD 1
13   IRETURN
14   MAXSTACK = 2
15   MAXLOCALS = 2
16
17 public accessor$someMethod$15() : void
18   ALOAD 0
19   INVOKESPECIAL ClassWithMethods.someMethod() : void
20   RETURN
21   MAXSTACK = 1
22   MAXLOCALS = 1
```

Porém há uma diferença entre o *ajc* e o *abc*, o *ajc* cria métodos estáticos que recebem um objeto da classe de onde se deseja acessar membros privados, enquanto o *abc* cria métodos de instância na mesma, o que diminui um pouco o código, já que não há a necessidade de passar o objeto como parâmetro para o método.

Podemos ver que o acesso a membros privados das classes Java por aspectos adiciona um *overhead* considerável, 2 métodos para cada atributo privado acessado e um método para cada método privado acessado.

### 4.3.2 *After*

O *advice* utilizado para inserir variações após a chamada de métodos é o *after call*, o código mostrado nas Listagem 4.39 e 4.40 apresenta um exemplo onde uma classe possui 3 métodos, affectedMethod(), callee() e someMethod(). Neste exemplo a variação será uma chamada a someMethod() após a chamada de callee() no método affectedMethod(), logo, o *advice* vai inserir essa chamada após a chamada de callee().

**Listagem 4.39** Classe afetada por *after call*.

```
1  public class ClassWithMethods {
```

```
2           public void affectedMethod() {
3                   this.callee();
4           }
5
6           public void callee() {
7           }
8
9           public void someMethod() {
10          }
11  }
```

**Listagem 4.40** Aspecto com *after call*.

```
1   public aspect AfterCall {
2           after(ClassWithMethods cthis) :
3                   call(public void ClassWithMethods.callee())
4                   && this(cthis)
5                   && withincode(public void ClassWithMethods.affectedMethod()) {
6                   cthis.someMethod();
7           }
8   }
```

Esse *advice* faz um *binding* do objeto de execução atual no contexto, ou seja, o que this referencia, à variável cthis, para que ele possa ser acessado dentro do corpo do *advice*, portanto, a chamada cthis.someMethod() será inserida após a chamada de calee() dentro de affectedMethod(). O que garante essa inserção no ponto correto são os *pointcuts* utilizados no *advice*, *call* especifica que o *advice* vai ser executado após a chamada do método especificado e *withincode* especifica que essa chamada de método se encontra no método especificado.

Na Listagem 4.41 vemos o *bytecode* do método affectedMethod() da ClassWithMethods compilado com o *ajc*, já que o mesmo foi alterado pelo aspecto:

**Listagem 4.41** *Bytecode* do método afetado pelo *after call* (*ajc*).

```
1   public affectedMethod() : void
2     ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3       TRYCATCHBLOCK L0 L1 L2 Throwable
4       L3 (0)
5       LINENUMBER 6 L3
6       ALOAD 0: this
7       L0 (2)
8       INVOKEVIRTUAL ClassWithMethods.callee() : void
9       L1 (4)
10      GOTO L4
11      L2 (6)
12      ASTORE 1
13      INVOKESTATIC AfterCall.aspectOf() : AfterCall
14      ALOAD 0: this
15      INVOKEVIRTUAL AfterCall.ajc$after$AfterCall$1$be448d7e(ClassWithMethods) : void
16      ALOAD 1
17      ATHROW
18      L4 (13)
19      NOP
20      INVOKESTATIC AfterCall.aspectOf() : AfterCall
21      ALOAD 0: this
22      INVOKEVIRTUAL AfterCall.ajc$after$AfterCall$1$be448d7e(ClassWithMethods) : void
23      NOP
24      L5 (19)
25      LINENUMBER 7 L5
26      RETURN
27      L6 (21)
28      LOCALVARIABLE this ClassWithMethods L3 L6 0
```

```
29      MAXSTACK = 2
30      MAXLOCALS = 2
```

Pode-se perceber que foi inserido um bloco try-catch ao redor da chamada do método alvo, e a chamada ao método que implementa o *advice* se encontra no bloco catch(Throwable), e também após o mesmo.

O *overhead* da inserção deste advice foi enorme, mas ele se deve a uma implementação ingênua do advice, uma vez que se tivesse sido utilizado o parâmetro returning na definição do *advice*, a inserção de um bloco try-catch não ocorreria, uma vez que a execução do corpo do *advice* só ocorreria em caso de sucesso na execução do *join point* especificado.

O *bytecode* do método invocado na instância do aspecto é mostrado na Listagem 4.42.

**Listagem 4.42** *Bytecode* do método de implementação do *after call* (*ajc*).

```
1    public ajc$after$AfterCall$1$be448d7e(ClassWithMethods) : void
2    ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3    ATTRIBUTE org.aspectj.weaver.Advice : unknown
4     L0 (0)
5      LINENUMBER 7 L0
6      ALOAD 1: cthis
7      INVOKEVIRTUAL ClassWithMethods.someMethod() : void
8     L1 (3)
9      LINENUMBER 8 L1
10     RETURN
11    L2 (5)
12     LOCALVARIABLE this AfterCall L0 L2 0
13     LOCALVARIABLE cthis ClassWithMethods L0 L2 1
14     MAXSTACK = 1
15     MAXLOCALS = 2
```

Como podemos ver, na linha 7 o método someMethod() é invocado, o que o advice especificava que deveria ser feito após a chamada de callee() em affectedMethod().

O mesmo exemplo compilado com o abc apresenta o mesmo comportamento de criação de um bloco try-catch, porém não há a criação de um método com o corpo do *advice* na classe do aspecto.

**Listagem 4.43** *Bytecode* do método afetado pelo *after call* (*abc*).

```
1   public affectedMethod() : void
2       TRYCATCHBLOCK L0 L1 L1 Throwable
3       ACONST_NULL
4       ASTORE 2
5      L0 (2)
6       LINENUMBER 6 L0
7       ALOAD 0
8      L2 (4)
9       LINENUMBER 6 L2
10      INVOKEVIRTUAL ClassWithMethods.callee() : void
11      GOTO L3
12     L1 (7)
13      ASTORE 1
14      ALOAD 2
15      IFNONNULL L4
16      INVOKESTATIC AfterCall.aspectOf() : AfterCall
17      POP
18     L4 (13)
19      LINENUMBER 7 L4
20      ALOAD 0
21     L5 (15)
22      LINENUMBER 7 L5
```

```
23      INVOKEVIRTUAL ClassWithMethods.someMethod() : void
24      ALOAD 1
25      ATHROW
26     L3 (19)
27      INVOKESTATIC AfterCall.aspectOf() : AfterCall
28      POP
29     L6 (22)
30      LINENUMBER 7 L6
31      ALOAD 0
32     L7 (24)
33      LINENUMBER 7 L7
34      INVOKEVIRTUAL ClassWithMethods.someMethod() : void
35      RETURN
36      MAXSTACK = 1
37      MAXLOCALS = 3
```

Como podemos perceber na Listagem 4.43 que a chamada a someMethod() se encontra no próprio método afetado, porém, apesar de não criar um método com o corpo do *advice*, ainda fica no código chamadas desnecessárias ao método aspectOf(), que retorna a instância do aspecto.

na Listagem 4.44 é mostrada uma versão mais inteligente do *advice*, utilizando o parâmetro returning, para especificar que o código do *advice* somente será executado quando a chamada de callee() retornar com sucesso:

**Listagem 4.44** Aspecto com *after returning call*.

```
1  public aspect AfterCall {
2          after(ClassWithMethods cthis) returning :
3                  call(public void ClassWithMethods.callee())
4                  && this(cthis) {
5                  cthis.someMethod();
6          }
7  }
```

A Listagem 4.45 apresenta o *bytecode* de affectedMethod() compilado com o *ajc*.

**Listagem 4.45** *Bytecode* do método afetado por um *after returning call* (*ajc*).

```
1  public affectedMethod() : void
2  ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3   L0 (0)
4    LINENUMBER 6 L0
5    ALOAD 0: this
6    INVOKEVIRTUAL ClassWithMethods.callee() : void
7    INVOKESTATIC AfterCall.aspectOf() : AfterCall
8    ALOAD 0: this
9    INVOKEVIRTUAL AfterCall.ajc$afterReturning$AfterCall$1$be448d7e(ClassWithMethods) : void
10   NOP
11  L1 (7)
12   LINENUMBER 7 L1
13   RETURN
14  L2 (9)
15   LOCALVARIABLE this ClassWithMethods L0 L2 0
16   MAXSTACK = 2
17   MAXLOCALS = 1
```

Como podemos ver na Listagem 4.45, essa versão é bem menor do que a anterior, visto que não foi criado um bloco try-catch e só há uma chamada a aspectOf() e ao método que implementa o *advice*.

Na Listagem 4.46 podemos ver que com o *abc* tivemos a mesma melhoria.

**Listagem 4.46** *Bytecode* do método afetado por um *after returning call* (*abc*).

```
1   public affectedMethod() : void
2    L0 (0)
3     LINENUMBER 6 L0
4     ALOAD 0
5    L1 (2)
6     LINENUMBER 6 L1
7     INVOKEVIRTUAL ClassWithMethods.callee() : void
8     INVOKESTATIC AfterCall.aspectOf() : AfterCall
9     POP
10   L2 (6)
11    LINENUMBER 7 L2
12    ALOAD 0
13   L3 (8)
14    LINENUMBER 7 L3
15    INVOKEVIRTUAL ClassWithMethods.someMethod() : void
16    RETURN
17    MAXSTACK = 1
18    MAXLOCALS = 1
```

Mas novamente se encontra presente a chamada desnecessária a aspectOf().

#### 4.3.2.1 *Set/Initialization*

A inserção de variações após a modificação do valor de um atributo também foi utilizada, assim como a inserção de variações no fim do corpo de um construtor.

A primeira pode ser feita com a utilização do *pointcut set*:

- set(FieldPattern)

A segunda, pode ser feita com a utilização do pointcut initialization:

- initialization(ConstructorPattern)

Como foi mencionado, o ponto após o qual a variação será inserida não influi na forma geral de inserção do *advice*. Logo, os exemplos com estes 2 *pointcuts* serão omitidos.

### 4.3.3 *Before*

O *advice* utilizado para inserir variações no início do corpo de um método é o *before execution*, o código das Listagens 4.47 e 4.48 apresentam um exemplo onde uma classe possui 3 métodos, affectedMethod(), e setBoolean(). Neste exemplo a variação será uma chamada a setBoolean() no início do método affectedMethod(), logo, o *advice* vai inserir essa chamada no início do corpo de affectedMethod().

**Listagem 4.47** Classe afetada por *before execution*.

```
1  public class ClassWithMethods {
2
3          private boolean bool;
4
5          public void affectedMethod() {
6          }
7
8          public void setBoolean(boolean bool) {
9                  this.bool = bool;
10         }
11 }
```

**Listagem 4.48** Aspecto com *before execution*.

```
1  public aspect BeforeExecution {
2          before(ClassWithMethods cthis) :
3                  execution(public void ClassWithMethods.affectedMethod(..))
4                  && this(cthis) {
5                  cthis.setBoolean(false);
6          }
7  }
```

Este *advice* fará com que uma chamada ao método setBoolean() passando *false* seja executada no ínicio do método affectedMethod().

Com o *ajc* temos o *bytecode* apresentado na Listagem 4.49 para o método affectedMethod():

**Listagem 4.49** *Bytecode* do método afetado por um *before execution* (ajc).

```
1  public affectedMethod() : void
2  ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3   L0 (0)
4    LINENUMBER 9 L0
5      INVOKESTATIC BeforeExecution.aspectOf() : BeforeExecution
6      ALOAD 0
7      INVOKEVIRTUAL
8          BeforeExecution.ajc$before$BeforeExecution$1$c60adbbd(ClassWithMethods) : void
9     L1 (4)
10     RETURN
11     LOCALVARIABLE this ClassWithMethods L1 L1 0
12     MAXSTACK = 2
13     MAXLOCALS = 1
```

No início do método há uma chamada a aspectOf() para a obtenção da instância do aspecto, e em seguida é invocado um método na classe do aspecto, o *bytecode* deste método é apresentado na Listagem 4.50.

**Listagem 4.50** *Bytecode* do método de implementação do *before execution* (*ajc*).

```
1  public ajc$before$BeforeExecution$1$c60adbbd(ClassWithMethods) : void
2  ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3  ATTRIBUTE org.aspectj.weaver.Advice : unknown
4   L0 (0)
5    LINENUMBER 7 L0
6    ALOAD 1: cthis
7    ICONST_0
8    INVOKEVIRTUAL ClassWithMethods.setBoolean(boolean) : void
9   L1 (4)
10   LINENUMBER 8 L1
11   RETURN
12  L2 (6)
13   LOCALVARIABLE this BeforeExecution L0 L2 0
14   LOCALVARIABLE cthis ClassWithMethods L0 L2 1
15   MAXSTACK = 2
16   MAXLOCALS = 2
```

Como pode ser observado, este método contém a implementação do *advice*, fazendo a chamada do método setBoolean().

Com o *abc* temos o *bytecode* apresentado na Listagem 4.51 para affectedMethod():

**Listagem 4.51** *Bytecode* do método afetado por um *before execution* (*abc*).

```
1  public affectedMethod() : void
2    INVOKESTATIC BeforeExecution.aspectOf() : BeforeExecution
3    POP
```

```
4   L0 (2)
5    LINENUMBER 7 L0
6    ALOAD 0
7   L1 (4)
8    LINENUMBER 7 L1
9    ICONST_0
10  L2 (6)
11   LINENUMBER 7 L2
12   INVOKEVIRTUAL ClassWithMethods.setBoolean(boolean) : void
13   RETURN
14   MAXSTACK = 2
15   MAXLOCALS = 1
```

Novamente foi feito o *inlining* do corpo do *advice* no método afetado e percebe-se também que uma chamada desncessária a aspectOf() foi deixada no código.

### 4.3.4 *Around*

*Advices around* permitem a execução do corpo do *advice* no lugar do *join point* capturado pelo *pointcut* utilizado no mesmo. O exemplo das Listagens 4.52 e 4.53 é essencialmente idêntico a uma das utilizações de *around* num dos jogos analisados pelo projeto *FLiP*. Um método que fazia a comparação entre duas teclas possuía uma variação que foi inserida utilizando *around*.

**Listagem 4.52** Classe afetada por *around execution*.

```
1  public class ClassWithMethods {
2          public boolean compare(int key1, int key2) {
3                  return key1 == key2;
4          }
5  }
```

**Listagem 4.53** Aspecto com *around execution*.

```
1  public aspect AroundExecution {
2          boolean around(int key1, int key2) :
3                  execution(public boolean compare(int,int))
4                  \&\& args(key1, key2) {
5                  boolean temp = proceed(key1, key2);
6                  return temp || key1 == -key2;
7          }
8  }
```

O *advice* inicialmente guarda em temp o valor da execução normal do método compare(), que é executado através da chamada a proceed(), e em seguida faz um *OR* do resultado com uma segunda comparação, que é a variação.

Este exemplo, compilado com o ajc, resulta no *bytecode* apresentado na Listagem 4.54.

**Listagem 4.54** *Bytecode* do método afetado por um *around execution* (*ajc*).

```
1  public compare(int,int) : boolean
2  ATTRIBUTE org.aspectj.weaver.MethodDeclarationLineNumber : unknown
3   L0 (0)
4    LINENUMBER 1 L0
5    ILOAD 1
6    ISTORE 3
7    ILOAD 2
8    ISTORE 4
9    ALOAD 0
10   ILOAD 3
```

```
11    ILOAD 4
12    INVOKESTATIC AroundExecution.aspectOf() : AroundExecution
13    ILOAD 3
14    ILOAD 4
15    ACONST_NULL
16    INVOKESTATIC ClassWithMethods.compare_aroundBody1$advice(ClassWithMethods,
17                    int, int, AroundExecution, int, int, AroundClosure) : boolean
18    IRETURN
19    MAXSTACK = 7
20    MAXLOCALS = 5
```

Podemos ver que o corpo do método compare() agora contém apenas a chamada ao método que contém a implementação do *advice*, todas as instruções anteriores são apenas a configuração dos parâmetros a serem passados para o método de implementação do *advice*.

**Listagem 4.55** *Bytecode* do *placeholder* do método afetado por um *around execution* (*ajc*).

```
1  private final static compare_aroundBody0(ClassWithMethods,int,int) : boolean
2   L0 (0)
3    LINENUMBER 5 L0
4    ILOAD 1: key1
5    ILOAD 2: key2
6    IF_ICMPNE L1
7    ICONST_1
8    IRETURN
9   L1 (6)
10   ICONST_0
11   IRETURN
12  L2 (9)
13   LOCALVARIABLE this ClassWithMethods L0 L2 0
14   LOCALVARIABLE key1 int L0 L2 1
15   LOCALVARIABLE key2 int L0 L2 2
16   MAXSTACK = 2
17   MAXLOCALS = 3
```

O método compare_aroundBody0(), mostrado na Listagem 4.55 é o método onde o código original de compare() foi colocado, para que as chamadas a proceed() no corpo do *advice* sejam feitas a esse método.

**Listagem 4.56** *Bytecode* do método de implementação de um *around execution* (ajc).

```
1  private final static compare_aroundBody1$advice(ClassWithMethods , int, int, AroundExecution,
2                                                  int, int, AroundClosure) : boolean
3   L0 (0)
4    LINENUMBER 107 L0
5    ILOAD 4
6    ILOAD 5
7    ALOAD 6
8    ASTORE 8
9    ISTORE 9
10   ISTORE 10
11   ALOAD 0: this
12   ILOAD 10
13   ILOAD 9
14   INVOKESTATIC ClassWithMethods.compare_aroundBody0(ClassWithMethods,int,int) : boolean
15   ISTORE 7
16  L1 (12)
17   LINENUMBER 108 L1
18   ILOAD 7
19   IFNE L2
20   ILOAD 4: temp
21   ILOAD 5
22   INEG
```

```
23    IF_ICMPEQ L2
24    ICONST_0
25    IRETURN
26   L2 (21)
27    ICONST_1
28    IRETURN
29   L3 (24)
30    LOCALVARIABLE this AroundExecution L0 L3 0
31    LOCALVARIABLE key1 int L0 L3 1
32    LOCALVARIABLE key2 int L0 L3 2
33    LOCALVARIABLE ajc_aroundClosure AroundClosure L0 L3 3
34    LOCALVARIABLE temp boolean L1 L3 4
35    MAXSTACK = 3
36    MAXLOCALS = 11
```

O método de implementação do *advice*, mostrado na Listagem 4.56 contém inicialmente uma chamada ao método que contém o corpo original de compare(), seguida da comparação que foi adicionada no *advice*.

O objeto do tipo AroundClosure é usado quando o corpo do advice usa *inner classes*, mais detalhes sobre a motivação por trás deste objeto podem ser encontrados em [HH04, Cor07].

Percebe-se que um advice around gerou um *overhead* maior do que o dos *advices before* e *after*.

Este *overhead* é consideravelmente menor com o *abc*. O bytecode do método compare() compilado com o *abc* é apresentado na Listagem 4.57.

**Listagem 4.57** *Bytecode* do afetado por um *around execution* (*abc*).

```
1   public compare(int,int) : boolean
2     INVOKESTATIC AroundExecution.aspectOf() : AroundExecution
3     POP
4     ILOAD 1
5     ILOAD 2
6     IF_ICMPNE L0
7     ICONST_1
8     ISTORE 0
9     GOTO L1
10   L0 (8)
11    ICONST_0
12    ISTORE 0
13   L1 (11)
14    ILOAD 0
15   L2 (13)
16    IFNE L3
17   L4 (15)
18    ILOAD 1
19   L5 (17)
20    ILOAD 2
21   L6 (19)
22    INEG
23   L7 (21)
24    IF_ICMPNE L8
25   L3 (23)
26    ICONST_1
27   L9 (25)
28    ISTORE 2
29    GOTO L10
30   L8 (28)
31    ICONST_0
32   L11 (30)
33    ISTORE 2
34   L10 (32)
35    ILOAD 2
```

```
36  L12 (34)
37    IRETURN
38    MAXSTACK = 2
39    MAXLOCALS = 3
```

Ao contrário do *ajc*, o *abc* fez *inlining* do corpo do advice, que está contido no próprio método compare().

Além dos *classfiles* do aspecto e da classe Java, foi criado 1 *classfile* adicional, que não é referenciado nem pelo aspecto, nem ela classe Java. A Listagem 4.58 apresenta o bytecode do mesmo.

**Listagem 4.58** *Bytecode* da interface gerada por um *around execution* (abc).

```
1  public interface Abc$proceed$AroundExecution$around$9 {
2    public abstract abc$proceed$AroundExecution$around$9(int, int, int, int, int, int) : boolean
3  }
```

Provavelmente é algum código que era utilizado por uma geração de *advices around* antiga e que não é mais utilizado.

## 4.4 Resumo da Inspeção

Aqui serão resumidos os pontos identificados em cada exemplo analisado:

### 4.4.1 Particularidades de cada compilador

- *abc*
  - Sempre cria um método <clinit>() nas classes mesmo quando não é necessária a existência do mesmo.
- *ajc*
  - Adiciona muita meta-informação nos *classfiles*.

### 4.4.2 *Inter-type Declarations*

- Atributos
  - Sem inicizaliação
    - *ajc*
      - 3 métodos criados no aspecto: inicialização do atributo (vazio), 2 dispatchers para acesso ao atributo.
      - adiciona uma chamada no construtor da classe alvo a um método no aspecto que faz a inicialização do atributo inserido, que no caso é vazio.
    - *abc*
      - nenhum overhead adicionado no aspecto e na classe Java, e esta não possui referência ao aspecto.

- Com Inicialização
    - *ajc*
        - 3 métodos criados no aspecto: inicialização do atributo, 2 *dispatchers* para acesso ao atributo.
        - adiciona uma chamada no construtor da classe alvo a um método no aspecto que faz a inicialização do atributo inserido, que atribui o valor utilizando o *dispatcher* de atribuição.
    - *abc*
        - adiciona uma chamada no construtor da classe alvo a um método no aspecto que faz a inicialização do atributo inserido, que atribui o valor utilizando o *dispatcher* de atribuição.
        - adiciona uma chamada no construtor da classe alvo ao método de inicialização do atributo.

- Constantes
    - *ajc*
        - Uso das constantes substituído por seu valor literal, não há atritutos novos inseridos na classe Java.
        - Um *classfile* gerado para a interface que contém as costantes.
    - *abc*
        - Mesmo comportamento do *ajc*.

- Método
    - *ajc*
        - Cria 3 métodos: um na classe alvo, que chama o método na classe do aspecto que contém a implementação, e um *dispatcher* no aspecto.
    - *abc*
        - Cria 3 métodos: um na classe alvo, que chama o método na classe do aspecto que contém a implementação, e um *dispatcher* no aspecto.

- Extensão de Classe
    - *ajc*
        - Nenhum *overhead* adicionado na classe Java ou no aspecto, a classe Java apenas possui, de novo, a declaração de extensão da superclasse.
    - *abc*
        - Mesmo comportamento do ajc.

- Implementação de Interface

- *ajc*
  - Nenhum *overhead* adicionado na classe Java ou no aspecto, a classe Java apenas possui, de novo, a declaração de implementação da interface.
- *abc*
  - Mesmo comportamento do *ajc*.

CAPÍTULO 5

# Discussão

*"Podes ser somente uma pessoa para o mundo, mas para alguma pessoa tu és o mundo."*

—GABRIEL GARCIA MARQUÉZ

A partir da análise realizada no Capítulo anterior pode-se sugerir as seguintes otimizações para o compilador *abc*:

- Não relacionada a aspectos
  - Remover <clinit>() vazios
- *Inter-type Declarations*
  - Atributos
    - Sem inicialização
      - Não necessita de otimizações.
    - Com inicialização
      - Mover a inicialização dos atributos para a classe alvo.
    - Constantes
      - Não necessita de otimização.
    - Método
      - Colocar o corpo de métodos *inter-type* na própria classe alvo
    - Mudança de hierarquia
      - Não necessita de otimização.
- Advices
  - Remover métodos accessors quando puderem ser os membros puderem ser acessados diretamente (o *advice* está *inline*).
  - Remover chamadas desnecessárias a aspectOf().

Uma vez que a obsfucadores de código, como o ProGuard [Pro07], possuem o recurso de remover de um *build* classes que não estejam sendo referenciadas, quando nenhuma classe Java referenciar um *classfile* de um aspecto, o mesmo não será incluído no *build*. Esse foi o principal ponto levado em consideração para a escolha das otimizações a serem implementadas. As duas otimizações escolhidas foram:

- Remover chamadas desnecessárias a aspectOf()
  - Uma vez que essas chamadas sejam removidas, aspectos que contém apenas *advices* não vão ser referenciados pelas classes Java afetadas, logo, serão removidos do *build*.
  - É importante notar que os métodos *accessors* ficam nas classes Java e não referenciam os aspectos.
- Mover o corpo de métodos *inter-type* para o método na classe alvo
  - A classe java não possuirá mais referência ao aspecto que inseriu o método, podendo o aspecto ser removido do *build*.

## 5.1 Mover o corpo de métodos *inter-type* para o método na classe alvo

### 5.1.1 Notação

As leis de programação aqui mostradas estabelecem uma equivalência entre programas AspectJ dado que algumas restrições sejam respeitadas. A estrutura de cada lei consiste de três partes: lado esquerdo, lado direito e pré-condições. Os dois primeiros são *templates* de programas equivalentes. A terceira parte indica condições que devem prevalecer para garantir a equivalência entre os programas.

O conjunto de declarações de tipos (classes e aspectos) é denotado por *ts*. Além disso, *pcs* e *as* denotam um conjunto de declarações de declarações de *pointcuts* e uma lista de declarações de *advices*, respectivamente.

É usado $\sigma$*(C.m)* para denotar a assinatura de um método *m* da classe *C*, incluindo o seu tipo de retorno e sua lista formal de parâmetros. Além disso, *context* é usado para denotar a lista de parâmetros do *advice*, incluindo o objeto executado (mapeado num parâmetro chamado *cthis*), e os parâmetros do método (*ps*). É usado *bind(context)* para denotar a expressão de designadores de *pointcut* que fazem *bind* dos parâmetros do *advice* (*this*, *target*, e *args*). As leis sempre expõem o contexto máximo disponível.

Existem diferentes pré-condições dependendo da direção em que a lei é usada. Isso é representado por setas, onde o símbolo ($\leftarrow$) indica que a pré-condição deve se manter quando aplicando a lei da direita para a esquerda. Similarmente, o símbolo ($\rightarrow$) indica que a pré-condição deve se manter quando aplicando a lei da esquerda para a direita. Finalmente, o símbolo ($\leftrightarrow$) indica que a pré-concição deve se manter em ambas as direções.

### 5.1.2 Prova de Corretude

Inicialmente, devemos provar que o que essa otimização se propõe a fazer preserva o comportamento do sistema.

Uma vez que o *abc* faz *inlining* de *advices*, deixando uma chamada desnecessária a aspectOf(), será provado apenas para um tipo de *advice*, provas similares podem ser desenvolvi-

das para os outros tipos de *advices*. Vamos provar a corretude da otimização usando o exemplo de um *advice before execution*.

De acordo com [Col05], a lei de *refactoring* 2, mostrada abaixo, apresenta a equivalência entre um trecho de código que está no início de um método numa classe Java, e o mesmo trecho sendo inserido através de um *advice before execution* de AspectJ.

**Law 1.** ⟨Add Before-Execution (original)⟩

$$
\begin{array}{l}
ts \\
\textbf{class}\ \ C\ \ \{ \\
\quad fs \\
\quad ms \\
\quad T\ \ m(ps)\ \ \{ \\
\quad\quad body' \\
\quad\quad body \\
\quad \} \\
\} \\
\textbf{privileged}\ \ \textbf{aspect}\ \ A\ \ \{ \\
\quad pcs \\
\quad bars \\
\quad afs \\
\}
\end{array}
\quad = \quad
\begin{array}{l}
ts \\
\textbf{class}\ \ C\ \ \{ \\
\quad fs \\
\quad ms \\
\quad T'\ \ m(ps)\ \ \{ \\
\quad\quad body \\
\quad \} \\
\} \\
\textbf{privileged}\ \ \textbf{aspect}\ \ A\ \ \{ \\
\ \ \textbf{before}(context)\ \ : \\
\quad\quad \textbf{execution}(\sigma(C.m))\ \ \&\& \\
\quad\quad bind(context)\{ \\
\quad\quad\quad body'[cthis/\textbf{this}] \\
\ \ \} \\
\}
\end{array}
$$

**provided**

- $\rightarrow$ $body'$ does not declare or use local variables; $body'$ does not call `super`;
- $\leftarrow$ $body'$ does not call `return`;
- $\leftrightarrow$ `A` has the lowest precedence on the join points involving the signature $\sigma(C.m)$; There is no designator `within` or `withincode` capturing join points inside $body'$.

O que está enunciado em *provided* são as pré-condições para que esta relação de equivalência seja verdadeira. Uma vez que estas pré-condições sejam atendidas, podemos ver que o efeito de utilizar um *advice before execution* com o intuito de inserir um corpo denotado por body' no início do método T m(ps) é equivalente ao deste corpo estar efetivamente no início do método.

Porém, a partir dos exemplos mostrados na análise, vemos que o que temos quando compilamos com o *abc* na verdade é o apresentado abaixo:

**Law 2.** ⟨Add Before-Execution (*abc*)⟩

$$
\boxed{\begin{array}{l} ts \\ \textbf{class}\ C\ \{ \\ \quad fs \\ \quad ms \\ \quad T\ \ m(ps)\ \ \{ \\ \quad\quad A.aspectOf(); \\ \quad\quad body' \\ \quad\quad body \\ \quad \} \\ \} \\ \textbf{privileged aspect}\ A\ \{ \\ \quad pcs \\ \quad bars \\ \quad afs \\ \} \end{array}} \quad = \quad \boxed{\begin{array}{l} ts \\ \textbf{class}\ C\ \{ \\ \quad fs \\ \quad ms \\ \quad T'\ \ m(ps)\ \ \{ \\ \quad\quad body \\ \quad \} \\ \}\ \textbf{privileged aspect}\ A\ \{ \\ \ \textbf{before}(context)\ \ : \\ \quad\quad \textbf{execution}(\sigma(C.m))\ \ \&\& \\ \quad\quad bind(context)\{ \\ \quad\quad\quad body'[cthis/\textbf{this}] \\ \ \ \} \\ \} \end{array}}
$$

Nota-se que a única diferença entre este e o anterior é a presença da chamada a aspectOf(). Como mencionado no Capítulo anterior, o que esse método faz é criar a instância *Singleton* do aspecto, caso ela não tenha sido criada ainda, e retorná-la. É um método sem efeitos colaterais para o programa, visto que quando a instância do aspecto for realmente necessária ela será criada, logo, a chamada a esse método pode ser removida, uma vez que quando é feito o *inlining* dos *advices*, nenhum método é chamado na instância do aspecto.

Uma vez que essa chamada seja removida, teremos a primeira relação de equivalência, visto que a a diferença entre as duas é a presença da chamada a aspectOf(), com isto está provado que a otimização proposta preserva o comportamento. Provas similares podem ser feitas para todos os outros tipos de *advices*.

### 5.1.3 Implementação

As chamadas desnecessárias ao método que retorna a instância do aspecto se apresentam no *bytecode* na forma de uma instrução *staticinvoke* seguida de uma instrução *pop*, uma vez que o retorno do método aspectOf() é colocado na pilha, se ele estiver seguido por uma instrução *pop*, isto significa que o valor de retorno do mesmo não está sendo utilizado, e ambas as instruções podem ser removidas. Abaixo é mostrado um trecho de *bytecode* que exemplifica este padrão.

**Listagem 5.1** Padrão em *bytecode* de chamada desnecessária a aspectOf().

```
1 INVOKESTATIC AspectClass.aspectOf() : AspectClass
2 POP
```

Uma vez que a otimização seria implementada no compilador *abc*, para a integração de uma otimização que remove essas chamadas haveria duas opções para a implementação da mesma:

- Descobrir o ponto no compilador onde se faz o *inlining* dos *advices* e, neste ponto, remover as chamadas a aspectOf().
- Implementar um transformador de código usando o framework *Soot*, e inserir o mesmo após a execução de todas as otimizações realizadas pelo *abc*.

Como o código do abc contém milhares de linhas de código e não é de fácil entendimento, a segunda opção foi a escolhida.

As transformações do Soot são feitas não diretamente em *bytecode*, mas sim numa linguagem intermediária chamada Jimple [VRH98]. O fato de Jimple ser *3-address code* é importante, pois ele indicará como o padrão mostrado anteriormente em *bytecode* será representado em Jimple. Abaixo é mostrada parte da gramática de Jimple:

*stmt* $\longrightarrow$ *assignStmt* | *identityStmt* | *gotoStmt* | *ifStmt* | *invokeStmt* | *switchStmt* | *monitorStmt* | *returnStmt* | *throwStmt* | *breakpointStmt* | *nopStmt*;

Podemos ver que *statements* em Jimple podem ser dos tipos mostrados acima. O tipo que interessa pra essa otimização é o invokeStmt. Como pode ser visto no *bytecode*, a invocação de aspectOf é seguida de um pop, ou seja, não se trata de um assignStmt, e sim de um invokeStmt.

*invokeStmt* $\longrightarrow$ invoke *invokeExpr*;

Como podemos ver acima, invokeStmt é uma instrução de invoke seguida por uma invokeExpr, que pode ser dos 4 tipos abaixo:

*invokeExpr* $\longrightarrow$ specialinvoke local.m($imm_1$...,$imm_n$) |

interfaceinvoke local.m($imm_1$...,$imm_n$) | virtualinvoke local.m($imm_1$...,$imm_n$) |

staticinvoke m($imm_1$...,$imm_n$);

Logo, o objetivo do transformador será encontrar os invokeStmt que invocam o método aspectOf() e os remover dos métodos onde se encontram. O algoritmo para realizar esta tarefa é o seguinte:

```
   Input: Classes compiladas pelo abc
   Output: Classes otimizadas
 1 foreach c no conjunto de classes do
 2     foreach método m da classe c do
 3         foreach statement s do método m do
 4             if s é um invokeStmt then
 5                 if a expressão de s for uma StaticInvokeExpr then
 6                     N ⟵ nome do método de s;
 7                     T ⟵ tipo da classe onde N está declarado;
 8                     R ⟵ tipo de retorno de M;
 9                     if N = "aspectOf" E T = R then
10                         remover s de m;
11                     end
12                 end
13             end
14         end
15     end
16 end
```

**Algoritmo 1**: Algoritmo para remoção de chamadas desnecessárias a aspectOf()

As transformações que atuam em corpos de métodos no Soot são classes que estendem a classe BodyTransformer. O método abstrato que deve ser implementado para realizar uma transformação é o internalTransform(). A implementação deste método mostrada abaixo implementa o algoritmo descrito acima, uma vez que ele será executado para cada método em cada classe compilada:

**Listagem 5.2** *Bytecode* de x() após otimização.

```
1  protected void internalTransform(Body body, String phaseName,
2                  Map options) {
3  PatchingChain methodStatements = body.getUnits();
4  Iterator statementsIterator = methodStatements.snapshotIterator();
5
6  while (statementsIterator.hasNext()) {
7          Stmt statement = (Stmt) statementsIterator.next();
8
9          if (statement instanceof JInvokeStmt) {
10                 JInvokeStmt assignStatement = (JInvokeStmt) statement;
11                 InvokeExpr invokeExpression =   assignStatement.getInvokeExpr();
12                 if (invokeExpression instanceof StaticInvokeExpr) {
13                                 SootMethod invokeMethod = invokeExpression.getMethod();
14                                 if (isAspectSingletonAccess(invokeMethod)) {
15                                         methodStatements.remove(statement);
16                                 }
17                         }
18                 }
19         }
20 }
21
22 private boolean isAspectSingletonAccess(SootMethod invokeMethod) {
23         return invokeMethod.getName().equals("aspectOf")
24                         && invokeMethod.getReturnType().equals(
25                                 invokeMethod.getDeclaringClass().getType());
26 }
```

Esta transformação efetivamente remove todas as ocorrências desnecessárias a aspectOf() de todas as classes compiladas.

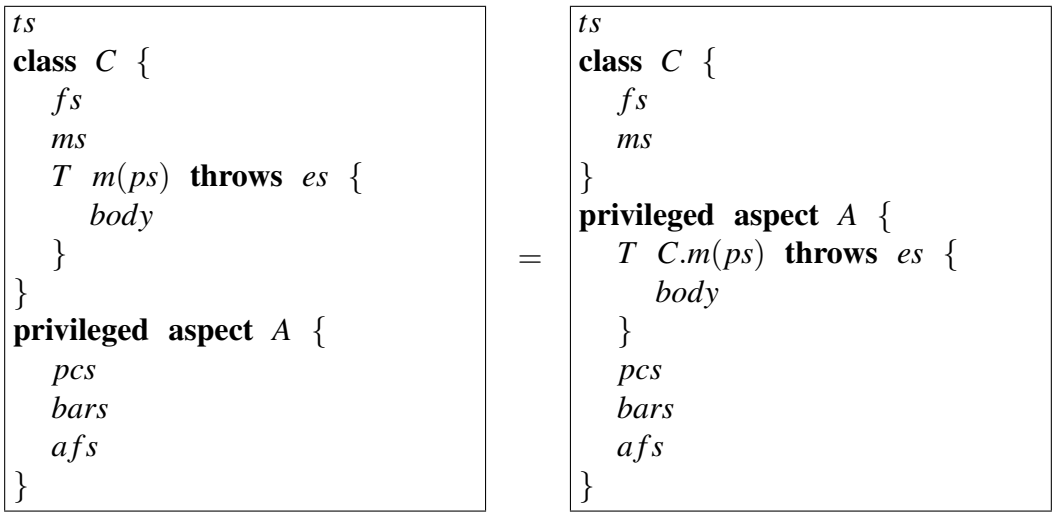
## 5.2 Remoção de chamadas desnecessárias a aspectOf()

### 5.2.1 Prova de Corretude

Inicialmente, devemos provar que o que essa otimização se propõe a fazer preserva o comportamento do sistema.

De acordo com [Col05], a lei de *refactoring* 19, mostrada abaixo, apresenta a equivalência entre um método que está diretamente numa classe Java, e o mesmo método sendo inserido por AspectJ.

**Law 3.** ⟨Move Method to Aspect⟩



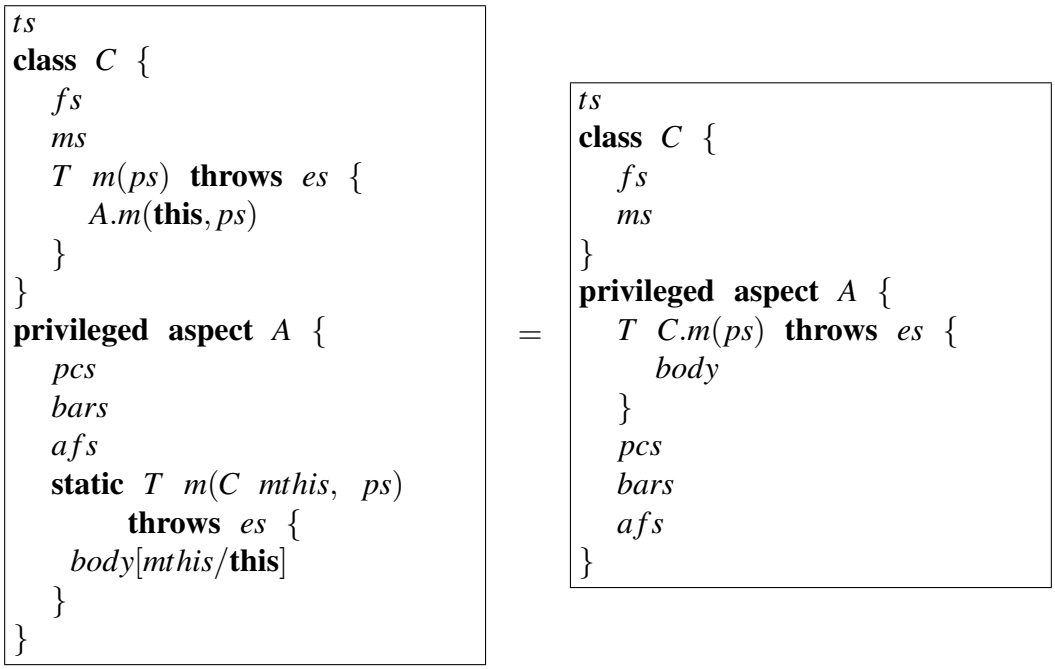

**provided**

↔ A does not introduce any field to C with the same name of a C field used in *body*.

O que está descrito em *provided* são as pré-condições para os dois lados da relação serem equivalentes. Como podemos observar, o efeito de um método ser inserido por *Inter-type Declaration* equivale ao método, com o mesmo corpo, inserido diretamente na classe Java.

Esta não é a estrutura identificada na compilação do *abc*. Uma vez que temos:

**Law 4.** ⟨Move Method to Aspect (abc)⟩



Ou seja, a inserção de um método via *inter-type declaration* está sendo equivalente à criação de um método na classe alvo, chamado de método alvo, e um método no aspecto, que contém o corpo que desejávamos inserir na classe, chamado de método de implementação. Além dos parâmetros especificados no método *inter-type*, o método de implementação recebe um objeto do tipo da classe alvo, e toda referência a *this* feita em *body* é substituída pelo nome do parâmetro, denotado por *mthis*.

Pode-se fazer uma analogia com o *refactoring Extract Method* [FBB+99], o que está presente no código seria um *refactoring Extract Method* que extraiu o corpo do método *m* da classe *C* para um método. A diferença para o *Extract Method* original, é que o original extrai para um método de instância na própria classe, enquanto que o que temos aqui é uma extração para um método estático em outra classe. Esse método estático recebe os parâmetros originais de *m* e *this*, que representa o objeto que está em execução, para acesso aos seus atributos e métodos de instância. Desfazer esse *refactoring* faria com que o corpo do método fosse inserido no lugar devido, ou seja, no próprio método *m* da classe *C*.

Para transformarmos essa representação criada pelo *abc* na primeira, mais simples, o que deve ser feito é o *inlining* do conteúdo deste método de implementação estático no ponto onde ele é invocado, ou seja, no método alvo. Feitos os devidos ajustes de parâmetros, para que *body* referencie o *this* da classe *C* após o *inlining*, teremos efetivamente a primeira relação de equivalência.

Com isso provamos que a otimização proposta preserva o comportamento do sistema.

### 5.2.2 Implementação

Ao contrário do *ajc*, o *abc* não apresenta um padrão de nomeclatura nos nomes dos métodos na classe do aspecto que indique o propósito de cada um.

Através de uma expressão regular simples poderíamos identificar se a chamada de método sendo investigada tem como alvo um método de implementação de um *Inter-type Method Declaration*, uma vez que ela sempre começaria com "ajc$intertype".

Para contornar este problema, a maneira mais simples identificada foi utilizar o recurso de aplicação de *tags* do Soot, para, modificando o compilador *abc*, adicionar uma *tag* específica que identifique o método desejado.

Após um estudo do código do *abc*, foi identificado que o método abaixo da classe InterTypeAdjuster realiza a criação do método de implementação e o método alvo, que é inserido na classe alvo:

**Listagem 5.3** Método que cria os métodos de *Inter-type Declarations*.

```
1 private void addMethod(IntertypeMethodDecl imd ) {
2                 SootMethod implMethod = addImplMethod(imd);
3                 addTargetMethod(imd,implMethod);
4 }
```

Após a criação de uma classe de *tag*, a classe InterTypeMethodImplementation (uma classe que implementa a interface soot.tagkits.Tag), o código de addMethod() foi modificado para colocar a tag no método de implementação. O código moficado é mostrado abaixo:

**Listagem 5.4** Método que cria os métodos de *Inter-type Declarations* (alterado).

```
1  private void addMethod( IntertypeMethodDecl imd ) {
2          SootMethod implMethod = addImplMethod(imd);
3          implMethod.addTag(new InterTypeMethodImplementationTag());
4          addTargetMethod(imd,implMethod);
5  }
```

A classe implementada para realizar transformações de *inlining* do corpo de método estáticos foi a classe abstrata AbstractStaticInvokeInliner, que estende BodyTransformer.

O algoritmo principal da transformação encontra-se no método internalTransform:

**Listagem 5.5** Método que faz implementa o algoritmo 2.

```
1  protected void internalTransform(Body body, String phaseName, Map options) {
2          PatchingChain methodStatements = body.getUnits();
3          Iterator statementsIterator = methodStatements.snapshotIterator();
4          SootMethod currentMethod = body.getMethod();
5          SootClass currentClass = currentMethod.getDeclaringClass();
6
7          while (statementsIterator.hasNext()) {
8                  Stmt statement = (Stmt) statementsIterator.next();
9
10                 if (statement instanceof JInvokeStmt) {
11                         JInvokeStmt invokeStatement = (JInvokeStmt) statement;
12                         InvokeExpr invokeExpression = invokeStatement.getInvokeExpr();
13                         if (invokeExpression instanceof StaticInvokeExpr) {
14                                 tryToInline(body, currentMethod, currentClass, statement,
15                                         invokeExpression);
16                         }
17                 }
18         }
19 }
```

A primeira versão da classe tratava apenas do *inlining* de invokeStmt, ou seja, não fazia *inlining* de invocações de métodos estáticos feitas em assignStmt, que sabe-se que seria um recurso utilizado para a aplicação de outras otimizações. O algoritmo executado por esta classe é o seguinte:

**Input**: Classes compiladas pelo *abc*
**Output**: Classes otimizadas

```
1  foreach c no conjunto de classes do
2      foreach método m da classe c do
3          foreach statement s do método m do
4              if s é um invokeStmt then
5                  if a expressão de s for uma StaticInvokeExpr then
6                      if s atender às condições de inlining then
7                          substituir s pelo do corpo de m;
8                      end
9                  end
10             end
11         end
12     end
13 end
```

**Algoritmo 2**: Algoritmo genérico para *inlining* de métodos estáticos.

O método tryToInline executa o último passo do algoritmo acima, e é mostrado na Listagem 5.6:

**Listagem 5.6** Método que faz *inlining* de invocações de métodos estáticos.

```
1  private void tryToInline(Body body, SootMethod currentMethod,
2          SootClass currentClass, Stmt statement, InvokeExpr invokeExpression) {
3          SootMethod invokeMethod = invokeExpression.getMethod();
4          if (shouldBeInlined(currentMethod, currentClass, statement,
5                          invokeMethod)) {
6                  SiteInliner.inlineSite(invokeMethod, statement, body
7                                  .getMethod());
8                  invokeMethod.getDeclaringClass().removeMethod(invokeMethod);
9          }
10 }
```

O que ele faz é chamar o método shouldBeInlined(), que retorna verdadeiro quando invokeExpr analisado se qualifica para *inlining*. O método shouldBeInlined() é abstrato, permitindo que as classes concretas determinem as condições que devem ser verdadeiras para que o *inlining* ocorra. Quando esse método retorna verdadeiro, o método inlineSite() da classe SiteInliner do Soot é utilizado para fazer o *inlining*.

**Listagem 5.7** Método abstrato que determina condições de *inlining*.

```
1  protected abstract boolean shouldBeInlined(SootMethod currentMethod,
2          SootClass currentClass, Stmt statement, SootMethod invokeMethod);
```

Além de do método abstrato que checa as condições específicas de cada otimização, a classe AbstractStaticInvokeInliner provê o método isSafeToInline(), mostrado na Listagem 5.8.

**Listagem 5.8** Método que diz se é segura a realização do *inlining*.

```
1  protected boolean isSafeToInline(SootMethod currentMethod,
2          SootClass currentClass, Stmt statement, SootMethod invokeMethod) {
3          return invokeMethod.getActiveBody() instanceof JimpleBody
4                          && InlinerSafetyManager.ensureInlinability(invokeMethod,
5                                          statement, currentMethod, "safe");
6  }
```

Esse método faz uso da classe InlineSafetyManager do Soot, que verifica se o corpo de um método pode ser *inlined* no local do statement especificado, no método também especificado.

Com isso, a implementação concreta do transformador de remoção de chamadas desnecessárias a aspectOf() ficou pequena e clara. Um único método teve que ser sobrescrito, o método abstrato shouldBeInlined(), onde nele especificamos as condições que devem ser atingidas para que o método seja *inlined*. A classe que realiza a otimização é mostrada na Listagem 5.9.

**Listagem 5.9** Classe que faz *inlining* do método de implementação de um método *Inter-type*.

```
1  public class InterTypeMethodInliner extends AbstractStaticInvokeInliner {
2          private static InterTypeMethodInliner instance = new InterTypeMethodInliner();
3
4          public static InterTypeMethodInliner v() {
5                  return InterTypeMethodInliner.instance;
6          }
7
8          protected boolean shouldBeInlined(SootMethod currentMethod,
9                          SootClass currentClass, Stmt statement, SootMethod invokeMethod) {
10                 boolean isInterTypeMethodImplementation = invokeMethod
11                         .getTag(InterTypeMethodImplementationTag.name) != null;
12                 boolean isMethodCallInsideOwningClass = invokeMethod
13                         .getDeclaringClass() == currentClass;
```

```
14                  boolean isSafeToInline = super.isSafeToInline(currentMethod,
15                          currentClass, statement, invokeMethod);
16
17                  return isInterTypeMethodImplementation
18                                  && !isMethodCallInsideOwningClass
19                                  && isSafeToInline;
20          }
21  }
```

É padrão do Soot que os transformadores sejam *Singletons*, e que o método que retorna a instância dos mesmos se chame v(), este padrão foi seguido no desenvolvimento do transformador.

A primeira verificação feita no método é se o método investigado possui a *tag* IntertypeMethodImplementationTag, uma vez que a geração dos métodos de implementação foi alterada no *abc* para colocar esta *tag* nos mesmos, apenas os métodos desejados possuirão esta *tag*, se não possuírem, serão desconsiderados.

A segunda verificação checa se a chamada não está sendo feita dentro do próprio aspecto, para evitar *inlining* de chamadas recursivas. A última verificação é a realizada pelo Soot para saber se é seguro fazer o inlining do método desejado no ponto específico onde ele está sendo invocado.

Este método, então, realiza as checagens necessárias para o *inlining* de métodos de implementação de declarações de método *inter-type*.

CAPÍTULO 6

# Resultados

*"Obstacles are those frightful things you see when you take your eyes off your goal."*
—HENRY FORD

Neste Capítulo serão mostrados os resultados obtidos com a aplicação das otimizações implementadas e descritas no Capítulo anterior. Inicialmente será apresentado o ganho obtido com cada otimização individualmente, e, posteriormente, com ambas as otimizações habilitadas.

## 6.1 Remoção de chamadas desnecessárias a aspectOf()

Utilizando o exemplo do *advice after returning* da análise, das Listagens 4.39 e 4.44, vamos mostrar o resultado da aplicação da otimização implementada. A Listagem 6.1 mostra o conteúdo do método affectedMethod() compilado com a otimização.

**Listagem 6.1** *Bytecode* de affectedMethod() após otimização.

```
1  public affectedMethod() : void
2     L0 (0)
3      LINENUMBER 5 L0
4      ALOAD 0
5     L1 (2)
6      LINENUMBER 5 L1
7      INVOKEVIRTUAL ClassWithMethods.callee() : void
8     L2 (4)
9      LINENUMBER 7 L2
10     ALOAD 0
11    L3 (6)
12     LINENUMBER 7 L3
13     INVOKEVIRTUAL ClassWithMethods.someMethod() : void
14     RETURN
15     MAXSTACK = 1
16     MAXLOCALS = 1
```

O corpo do *advice* foi inserido no lugar correto, e não há mais a chamada desnecessária a apsectOf(), deste modo a classe afetada não possui nenhuma referência ao *classfile* do aspecto.

## 6.2 Mover o corpo de métodos inter-type para o método na classe-alvo

Utilizando o exemplo de método *inter-type* das Listagens 4.30 e 4.31, vamos apresentar o resultado da aplicação da otimização implementada.

**Listagem 6.2** *Bytecode* de x() após otimização.

```
1 public x() : int
2         ICONST_0
3         IRETURN
4         MAXSTACK = 1
5         MAXLOCALS = 1
```

O conteúdo do método que antes se encontrava num método na classe do aspecto, agora se encontra efetivamente no método inserido na classe, e não há nenhuma referência ao *classfile* do aspecto no *classfile* da classe BlankClass.

## 6.3 Tamanho dos Builds

Aplicando apenas a primeira otimização, a Tabela 6.1 apresenta a melhoria no tamanho dos *builds* comparando a versão com aspectos sem a otimização e a com a otimização aplicada.

| *Build ID* | *abc* | *abc* + otimização 1 |
|---|---|---|
| MOT1 | 81.534 | 78.713 |
| MOT2 | 93.985 | 91.136 |
| MOT3 | 85.956 | 82.405 |
| S40 | 72.950 | 69.862 |
| S40M2 | 80.668 | 78.734 |
| S40M2V3 | 80.526 | 78.315 |
| S60M1 | 94.085 | 91.203 |
| S60M2 | 93.122 | 90.560 |
| SAM1 | 72.024 | 69.270 |
| SAM2 | 84.544 | 82.109 |
| SE02 | 91.719 | 89.218 |
| SE03 | 80.433 | 77.913 |
| SE04 | 80.383 | 77.890 |
| SIEM3 | 81.754 | 79.116 |
| SIEM4 | 81.101 | 78.731 |
| Média | 83.652 | 81.012 |
| Diferença em relação ao não otimizado | 0% | -3,1559% |

**Tabela 6.1** Tamanho (em bytes) da versão com aspectos sem otimização e da versão com a otimização 1

Com a aplicação apenas da segunda otimização, a Tabela 6.2 apresenta a melhoria obtida.

Pode-se perceber que a melhoria não é tão grande quanto à da primeira otimização, isso se deve ao fato de que a quantidade de método *inter-type* nos aspectos do jogo ser inferior à quantidade de *advices*. Como podemos ver na Tabela 6.3 a aplicação das duas otimizações, o ganho obtido é maior do que com cada uma individualmente.

Por fim, é comparado o tamanho da versão com aspectos com as duas otimizações, com o tamanho do jogo original, sem aspectos e compilada com o *abc*, que foi o compilador que

| *Build ID* | *abc* | *abc + otimização 2* |
|---|---|---|
| MOT1 | 81.534 | 80.426 |
| MOT2 | 93.985 | 92.845 |
| MOT3 | 85.956 | 84.631 |
| S40 | 72.950 | 71.541 |
| S40M2 | 80.668 | 79.175 |
| S40M2V3 | 80.526 | 79.400 |
| S60M1 | 94.085 | 92.953 |
| S60M2 | 93.122 | 91.923 |
| SAM1 | 72.024 | 71.665 |
| SAM2 | 84.544 | 84.190 |
| SE02 | 91.719 | 90.674 |
| SE03 | 80.433 | 79.006 |
| SE04 | 80.383 | 79.337 |
| SIEM3 | 81.754 | 80.654 |
| SIEM4 | 81.101 | 79.979 |
| Média | 83.652 | 82.560 |
| Diferença em relação ao não otimizado | 0% | -1,3054% |

**Tabela 6.2** Tamanho (em bytes) da versão com aspectos sem otimização e da versão com a otimização 2

gerou o menor tamanho para esta versão do jogo, e do jogo com aspectos, também compilado com o abc, mas sem as otimizações. A Tabela 6.4 apresenta o resultado final das otimizações comparado com a versão original do jogo.

Pode-se perceber que ainda há uma diferença grande entre a versão sem aspectos para a otimizada com aspectos, mas a diminuição dessa, porém o ganho obtido com a aplicação das otimizações foi uma diminuição de 48% no *overhead* que é adicionado ao jogo pelo uso de aspectos. A soma do ganho obtido com cada otimização separadamente foi quase idêntica ao valor do ganho obtido com as duas otimizações, com uma difereça de apenas 200 bytes a menos na versão com ambas as otimizações.

Apesar de não terem sido feitas medições de performance, é esperado que haja uma melhoria na performance dos jogos, uma vez que não se vai mais estar chamando desnecessariamente o método que retorna a instância do aspecto e que também não vai mais haver um nível de profundidade de chamadas de método quando forem chamados métodos inseridos por *inter-type declarations*.

| *Build ID* | *abc* | *abc* + 2 otimizações |
|---|---|---|
| MOT1 | 81.534 | 77.299 |
| MOT2 | 93.985 | 89.738 |
| MOT3 | 85.956 | 80.597 |
| S40 | 72.950 | 68.457 |
| S40M2 | 80.668 | 76.955 |
| S40M2V3 | 80.526 | 76.905 |
| S60M1 | 94.085 | 89.815 |
| S60M2 | 93.122 | 88.775 |
| SAM1 | 72.024 | 68.908 |
| SAM2 | 84.544 | 81.744 |
| SE02 | 91.719 | 88.196 |
| SE03 | 80.433 | 76.504 |
| SE04 | 80.383 | 76.843 |
| SIEM3 | 81.754 | 77.732 |
| SIEM4 | 81.101 | 77.335 |
| Média | 83.652 | 79.720 |
| Diferença em relação ao não otimizado | 0% | -4,7004% |

**Tabela 6.3** Tamanho (em bytes) da versão com aspectos sem otimização e da versão com as duas otimizações

| *Build ID* | original | com aspectos (*abc*) | com aspectos (otimizado) |
|---|---|---|---|
| MOT1 | 72.512 | 81.534 | 77.299 |
| MOT2 | 84.939 | 93.985 | 89.738 |
| MOT3 | 72.544 | 85.956 | 80.597 |
| S40 | 64.913 | 72.950 | 68.457 |
| S40M2 | 72.563 | 80.668 | 76.955 |
| S40M2V3 | 72.115 | 80.526 | 76.905 |
| S60M1 | 85.190 | 94.085 | 89.815 |
| S60M2 | 84.106 | 93.122 | 88.775 |
| SAM1 | 66.246 | 72.024 | 68.908 |
| SAM2 | 80.251 | 84.544 | 81.744 |
| SE02 | 83.958 | 91.719 | 88.196 |
| SE03 | 72.146 | 80.433 | 76.504 |
| SE04 | 72.132 | 80.383 | 76.843 |
| SIEM3 | 72.854 | 81.754 | 77.732 |
| SIEM4 | 72.078 | 81.101 | 77.335 |
| Média | 75.236 | 83.652 | 79.720 |
| Diferença em relação ao original | 0% | 11,181% | 5,9599% |

**Tabela 6.4** Tamanho (em bytes) da versão original do jogo comparada com a versão com aspectos sem e com otimizações.

CAPÍTULO 7

# Conclusão

*"Be the change you wish to see in the world."*
—MAHATMA GANDHI

Neste texto, apresentou-se um estudo da geração de código dos 2 principais compiladores de AspectJ: o *AspectJ Compiler* (*ajc*), e o *AspectBench Compiler* (*abc*). Nesse estudo, foram identificadas ineficiências na geração de código de ambos os compiladores para um subconjunto de construções de AspectJ, aquelas identificadas como necessárias pelo projeto *FLiP* para a implementação da maior parte das variações de código de 2 jogos *Java ME*.

A análise realizada identificou o *abc* como o compilador que gerava o menor código de AspectJ. Além disso, a análise apontou os pontos onde cada compilador poderia ser otimizado para uma geração de código mais compacto e eficiente. Foi listado um conjunto de otimizações possíveis de serem implementadas, e, deste conjunto, duas foram escolhidas e implementadas, baseado na intuição de que as mesmas trariam uma maior diminuição do *overhead*.

É importante ressaltar novamente que estas otimizações podem ser válidas apenas no contexto de uma Linhas de Produtos de Software, onde normalmente cada código inserido por por AspectJ vai para apenas um lugar específico do código Java.

Para o jogo utilizado neste trabalho, o uso das duas otimizações levou a uma redução de 48% do *overhead* causado pelo uso de AspectJ. Essa redução é um avanço para tornar o uso de *AspectJ* como mecanismo de inserção de variações de código em jogos *Java ME* uma opção mais atrativa do que pré-processamento.

Como trabalhos futuros, enumera-se:

- provar que as outras otimizações propostas não implementadas preservam o comportamento;
- implementar as outras otimizações não implementadas;
- analisar as construções de AspectJ que não estavam no escopo deste estudo;
- implementar as otimizações no *ajc*;
- analisar o impacto de construções que interferem umas com as outras;
- avaliar o impacto das otimizações propostas com *benchmarks* de performance e memória.

O teste das otimizações numa linha de produtos de jogos *Java ME* é um bom indicador de que as mesmas podem trazer grandes resultados num ambiente de alta variabilidade, como é o domínio analisado.

Este estudo mostrou que pequenas otimizações podem levar a um ganho que, no contexto de jogos móveis, pode ser um fator decisivo para a escolha do uso de AspectJ como mecanismo de inserção de variabilidades de código.

# Referências Bibliográficas


[ACH+04] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Building the abc aspectj compiler with polyglot and soot. Technical report, The abc Group, October 2004.

[ACH+05a] Pael Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible aspectj compiler. *TAOSD*, 2005.

[ACH+05b] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising aspectj. *PLDI*, 2005.

[ACV+05] Vander Alves, Ivan Cardim, Heitor Vital, Pedro Sampaio, Alexandre Damasceno, Paulo Borba, and Geber Ramalho. Comparative analysis of porting strategies in j2me games. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 123–132, September 2005.

[ANS+06] Vander Alves, Albeto Costa Neto, Sérgio Soares, Gustavo Santos, Fernando Calheiros, Vilmar Nepomuceno, Davi Pires, Jorge Leal, and Paulo Borba. From conditional compilation to aspects: A case study in software product lines migration. *1st Workshop on Aspect-Oriented Product Line Engineering, in conjunction with 5th ACM International Conference on Generative Programming and Component Engineering*, October 2006.

[CBS+07] Fernando Calheiros, Paulo Borba, Sérgio Soares, Vilmar Nepomuceno, and Vander Alves. Product line variability refactoring tool. *1st Workshop on Refactoring Tools (WRT07), in conjunction with the 21st European Conference on Object-Oriented Programming*, pages 33,34, July 2007.

[CLG+06] Tarcisio Camara, Rodrigo Lima, Rangner Guimarães, Alexandre Damasceno, Vander Alves, Pedro Macedo, and Geber Ramalho. Massive mobile games porting: Meantime study case. *Brazilian Symposium on Computer Games and Digital Entertainment - Computing track*, 2006.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[Col05] Leonardo Cole. Deriving refactorings for aspectj. Master's thesis, Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil, February 2005.

[Cor07] Eduardo Santos Cordeiro. Otimizações na compilação de adendos de contorno em programas orientados por aspectos. Master's thesis, Universidade Federal de Minas Gerais, 2007.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. *AOSD*, 2004.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. *ECOOP*, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP*, 1997.

[Kul07] Eugene Kuleshov. *ASM 3.0 A Java bytecode engineering library*, 2007. http://download.forge.objectweb.org/asm/asm-guide.pdf.

[Kuz04] Sascha Kuzins. Efficient implementation of around-advice for the aspectbench compiler. Master's thesis, Oxford University, 2004.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition, 1999. Disponível em http://java.sun.com/docs/books/vmspec/index.html.

[Mea07] Meantime. *Meantime Mobile Creations*, 2007. http://www.meantime.com.br.

[Mic07] Sun MicroSystems. *Java Platform Mobile Edition*, 2007. http://java.sun.com/j2me.

[NCD03] Nathaniel Nystrom, Michael R. Clarkson, and Chris Dutchyn. Polyglot: An extensible compiler framework for java. *CC'03*, 2622:138–152, 2003.

[Pro07] ProGuard. *ProGuard, java class file shrinker, optimizer and obfuscator*, 2007. http://proguard.sourceforge.net.

[Ret07] RetroGuard. *RetroGuard for Java Bytecode Obfuscation*, 2007. http://www.retrologic.com/retroguard-main.html.

[S.M97] Steven S.Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Teaa] AspectJ Team. *Guide for Developers of the AspectJ Compiler and Weaver*. Disponível no módulo docs do código-fonte do compilador ajc.

[Teab] AspectJ Team. *Página oficial do projeto AspectJ e do compilador ajc*. http://www.eclipse.org/aspectj, Última visita em Agosto de 2007.

[Tea05a] AspectJ Team. *The AspectJ development environment guide*, 2005. http://www.eclipse.org/aspectj/doc/released/devguide/index.html.

[Tea05b] AspectJ Team. *The AspectJ development kit developerś notebook*, 2005. http://www.eclipse.org/aspectj/doc/next/adk15notebook/index.html.

[Ter07] Robert Tercek. First decade of mobile games. *Presentation at GDC Mobile*, 2007.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? *Compiler Construction*, pages 18–34, 2000. htp://www.tirawireless.com.

[VRH98] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, McGill University, July 1998.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. *Proceedings of CASCON*, pages 125–135, July 1999.