Rogério Celestino dos Santos

EXTRAÇÃO E EVOLUÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE NA ÁREA DE JOGOS PARA CELULARES USANDO PROGRAMAÇÃO ORIENTADA POR FEATURES

Dissertação apresentada ao Programa de Pós-Graduação em Informática como requisito parcial para obtenção do Grau de Mestre em Informática pela Pontifícia Universidade Católica de Minas Gerais.

Orientador: Marco Túlio de Oliveira Valente

Belo Horizonte
2009

FICHA CATALOGRÁFICA

Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

S237e

Santos, Rogério Celestino dos

Extração e evolução de linhas de produtos de software na área de jogos para celulares usando programação orientada por features. / Rogério Celestino dos Santos. – Belo Horizonte, 2009.

ix, 66f. : il.

Orientador: Marco Túlio de Oliveira Valente.

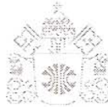
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais. Programa de Pós-graduação em Informática.

Bibliografia.

1. Engenharia de software – Teses. 2. Jogos por computador 3. Programação orientada a objetos (Computação) I. Valente, Marco Túlio de Oliveira. II. Pontifícia Universidade Católica de Minas Gerais. III. Título

CDU: 681.3.03

Bibliotecário: Fernando A. Dias – CRB6/1084



PUC Minas

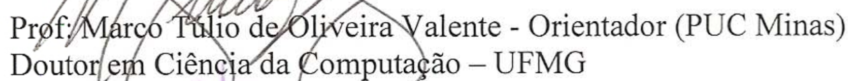
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

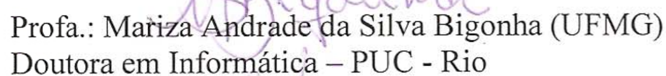
"Extração e Evolução de Linhas de Produtos de Software na Área de Jogos para Celulares usando Programação Orientada por Features"

Rogério Celestino dos Santos

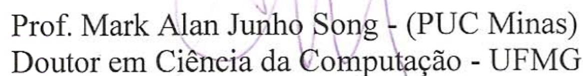
Dissertação defendida e aprovada pela seguinte banca examinadora:



Prof. Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Doutor em Ciência da Computação – UFMG



Profa.: Mariza Andrade da Silva Bigonha (UFMG)
Doutora em Informática – PUC - Rio



Prof. Mark Alan Junho Song - (PUC Minas)
Doutor em Ciência da Computação - UFMG

Belo Horizonte, 02 de abril de 2009.

Anel Rodoviário – KM 23,5, Rua Walter Ianni, 255 – São Gabriel / Belo Horizonte – MG / CEP: 31980-110
Fone: (31) 3439-5204 – Fax: (31) 3439-5204 – e-mail: mestradoinfsec@pucminas.br

À minha família,
minha namorada
e meus amigos.

Agradecimentos

Agradeço à minha família por me dar um grande apoio e esta oportunidade de concluir o mestrado.

Minha linda namorada por estar sempre ao meu lado e me apoiando em todos os momentos de minha vida. Sempre me alegrando na hora de estresse.

Ao meu orientador Professor Marco Túlio, um grande orientador, que teve paciência em sempre revisar meus textos encontrando os mesmos erros e me fez chegar a esse grande trabalho de mestrado. E me dando incentivos além do mestrado.

Ao Professor Ricardo Poley, pelas conversas interessantes e dicas importantes no decorrer do mestrado.

A Giovanna, secretária acadêmica, pela presteza e atenção a nós alunos.

Aos meus amigos e colegas da PUC.

A Professora Rosilane Mota, por ter me dado uma chance de lecionar no curso de jogos digitais da PUC.

E a todos professores pelos conhecimentos e amizades que obtive nesta fase de minha vida.

Resumo

Jogos para celulares apresentam desafios extras para seus desenvolvedores. Dentre tais desafios, provavelmente o mais importante consiste em prover suporte à grande variedade de dispositivos celulares existentes no mercado. Normalmente, isso requer que desenvolvedores de jogos tenham que gerar e manter versões de seus sistemas para diversas plataformas de celulares, de forma a lidar com características particulares dessas plataformas, incluindo APIs de desenvolvimento proprietárias e restrições de *hardware*, tais como tamanho do *display*, quantidade de memória, acessórios disponíveis, processamento etc. Assim, jogos para celulares constituem um domínio de aplicação promissor para desenvolvimento baseado em linhas de produto de software (LPS). Diversas construções de programação podem ser usadas para apoiar a criação de LPS, dentre elas pode-se mencionar compilação condicional, orientação por objetos, *mixins* e programação orientada por aspectos. Porém, uma nova solução para implementação de LPS, chamada de programação orientada por *features*, têm gerado grande interesse na comunidade científica. Descreve-se nesta dissertação uma experiência de extração e evolução de uma linha de produtos de software na área de jogos para celulares utilizando programação orientada por *features*. Particularmente, a linha de produtos descrita na dissertação foi implementada usando-se conceitos de programação orientada por *features* tal como disponibilizados pelo sistema AHEAD. A dissertação apresenta também uma avaliação quantitativa e qualitativa da linha de produto extraída e evoluída, bem como compara AHEAD com tecnologias alternativas, como compilação condicional, orientação por objetos, *mixins* e programação orientada por aspectos.

Palavras-chave: Engenharia de Software, Jogos por Computador, Programação Orientada a Objetos.

Abstract

Mobile phone games present extra challenges for softwares developers. Among these challenges, probably the most important is to provide support for the wide variety of mobile devices on the market. Normaly, this requires game developers to create and maintain versions of their systems for several mobile platforms in order to deal with particular characteristics of these platforms, including proprietary APIs and hardware restrictions (display size, amount of memory, available accessories, processing power etc). Thus, mobile games are a promising area for development based on software product lines (SPL). Several programming tecjnologies can be used to support the creation of SPL, among them we can mention conditional compilation, object-oriented programming, mixins and aspect-oriented programming. However, a new solution for the implementation of SPL, called feature-oriented programming, has generated great interest in the scientific community. This master dissertation documents an experience of extraction and evolution of a software product line for mobile games using feature-oriented programming. Particularly, the software product line described in the work has been implemented using the concepts of feature-oriented programming, as provided by the AHEAD system. This dissertation also presents a quantitative and qualitative assessment of the software product line extracted and evolved, and compares AHEAD with alternative technologies, such as conditional compilation, oriented-object programming, mixins and aspect-oriented programming.

Key-words: Software Engineering, Computer Games, Object-Oriented Programming.

Lista de Figuras

FIGURA 1 Variações de um jogo. 16

FIGURA 2 Atividades essenciais de uma LPS (CLEMENTS; NORTHROP, 2001). . . 20

FIGURA 3 Notações do DF (BATORY, 2005). 21

FIGURA 4 Diagrama de *Features* da LPS `Carro`. 21

FIGURA 5 Exemplo de Compilação Condicional 22

FIGURA 6 Espaço multidimensional dos requisitos de um carro. 24

FIGURA 7 Exemplo de AOP. 24

FIGURA 8 Classes e seus refinamentos. 25

FIGURA 9 Exemplo de refinamento em Jakarta. 26

FIGURA 10 Gramática da LPS `Carro`. 27

FIGURA 11 Exemplo de tela da ferramenta `guidsl`. 27

FIGURA 12 Tela do jogo Chuckie Egg 28

FIGURA 13 Fragmento de código da classe `Chuckie` 29

FIGURA 14 Implementação da *feature* `Sound` usando compilação condicional 30

FIGURA 15 Implementação da *feature* `Sound` usando aspectos 31

FIGURA 16 Implementação da *feature* `Sound` usando refinamentos 32

FIGURA 17 Telas do jogo Bomber 35

FIGURA 18 Diagrama de classes do Bomber. 37

FIGURA 19 Modelo de *Features* 38

FIGURA 20 Classe Bomb da versão original com comentários indicando *features* 40

FIGURA 21 Classe `Bomb` refatorada (com apenas funcionalidades básicas) 41

FIGURA 22 Menu de opções do jogo. 42

FIGURA 23 Métodos da classe `IntroGameHandler` que tratam o menu da versão original. 43

FIGURA 24 Classe`IntroGameHandler` refatorada. 44

FIGURA 25 Refinamento que introduz a *feature* `Sound` na classe `Bomb` 44

FIGURA 26 Refinamento que introduz a opção de som no menu 45

FIGURA 27 Exemplos de tela da *feature* `Cloud` no jogo `Bomber`. 45

FIGURA 28 Classe `ResourceManager` da versão original 46

FIGURA 29 Refatoração do método `loadLevel` da classe `ResourceManager` 46

FIGURA 30 Refinamento que introduz a *feature* `Splash` na classe `Bomb` 46

FIGURA 31 Refinamento que introduz a *feature* `ExplosionBlast` na classe `Bomb` ... 47

FIGURA 32 Refinamento que introduz a *feature* `DamageTerrain` na classe `Bomb` 47

FIGURA 33 Refinamento que introduz a *feature* `Cloud` na classe `Game` 47

FIGURA 34 Refinamento da interface `GameState` para o tratamento da *feature* `Cloud` 47

FIGURA 35 Refinamento da *feature* `Cloud` na classe `ResourceManager` 48

FIGURA 36 Refinamento do construtor *feature* `Smoke` na classe `FallingDebris` 48

FIGURA 37 Refinamento da classe `FallingDebris`. 48

FIGURA 38 Novo Diagrama de Feature. 51

FIGURA 39 Classe `MissionGameHandler` 52

FIGURA 40 Classe `MissionGameHandler` refatorada 52

FIGURA 41 Refinamento da *feature* `SpeedControl` 53

FIGURA 42 Refinamento da classe `Plane` 54

FIGURA 43 Refinamento da classe `Tank` na *feature* `MoveTank` 54

FIGURA 44 Refinamento da classe `Zeppelin` na *feature* `MoveZeppelin` 55

FIGURA 45 Refinamento da classe `Storage` na *feature* `ServiceRank` 56

FIGURA 46 Classe `List`. 60

FIGURA 47 Classe `ListWithSize`. 60

FIGURA 48 Evolução da classe `List`. 60

FIGURA 49 Exemplo de herança. 61

FIGURA 50 *Feature* obrigatória (`f1`) e opcionais (`f2` e `f3`) implementadas por meio de herança. 62

FIGURA 51 Classe `Bomb`. 62

FIGURA 52 Classe `BombDamageTerrain`. 62

FIGURA 53 Classe `BombSound`. 63

FIGURA 54 Classe `BombSoundDamageTerrain`. 63

FIGURA 55 Refinamento que insere o som na classe `Bomb`. 66

FIGURA 56 Aspecto que insere o som na classe `Bomb`. 66

FIGURA 57 *Mixin* para tratamento de som. 67

FIGURA 58 Criação da subclasse `SoundBomb` usando *mixin* `Sound`. 67

Lista de Tabelas

TABELA 1 Tamanho (em LOC) 33

TABELA 2 Comparação entre as tecnologias CC, AOP e FOP 34

TABELA 3 Número de linhas rotuladas para cada *feature*. 39

TABELA 4 Números de classes refinadas pelas *features*. 40

TABELA 5 Refatorações realizadas. 49

TABELA 6 Classes refinadas pelas novas *features*. 56

TABELA 7 LOC das *features*. 57

Sumário

1 INTRODUÇÃO 15

1.1 Visão Geral do Problema 15

1.2 Objetivos 16

1.3 Estrutura da Dissertação 17

2 LINHAS DE PRODUTO DE SOFTWARE 19

2.1 Diagrama de *Features* 21

2.2 Tecnologias Analisadas 22

2.2.1 Compilação Condicional 22

2.2.2 Programação Orientada por Aspectos 23

2.2.2.1 AspectJ 23

***2.2.3 Programação Orientada por* Features** 24

2.2.3.1 AHEAD 25

2.2.3.2 Regras de projeto 26

2.3 Estudo de Caso 28

2.3.1 Compilação Condicional 29

2.3.2 Aspectos 29

2.3.3 AHEAD 31

2.4 Avaliação 32

2.5 Comentários Finais 34

3 EXTRAÇÃO DE UMA LPS UTILIZANDO PROGRAMAÇÃO ORIENTADA POR FEATURES 35

3.1 Arquitetura do Sistema 35

3.2 Metodologia 36

3.3 Extração da LPS 39

3.3.1 Feature SoundFX 41

3.3.2 Feature VisualFX 43

3.4 Comentários Finais 49

4 EVOLUÇÃO DE UMA LPS UTILIZANDO PROGRAMAÇÃO ORIENTADA POR FEATURES 50

4.1 Evolução da LPS 50

4.1.1 Feature SpeedControl 53

4.1.2 Feature MoveTank 53

4.1.3 Feature MoveZeppelin 54

4.1.4 Feature ServiceRank 55

4.2 Comentários Finais 57

5 AVALIAÇÃO 58

5.1 Avaliação Qualitativa 58

5.2 Comparação com Orientação por Objetos 61

5.3 Comparação com Orientação por Aspectos 65

5.4 Comparação com *Mixins* 66

5.5 Comentários Finais 67

6 CONCLUSÃO 68

6.1 Visão Geral do Trabalho 68

6.2 Contribuições 69

6.3 Trabalhos Futuros 70

REFERÊNCIAS 71

1 INTRODUÇÃO

1.1 Visão Geral do Problema

Atualmente jogos digitais constituem aplicações cada vez mais importantes na indústria de software. Mesmo em momentos de crise, foi o mercado que teve mais crescimento (NPD...,). Um segmento relevante nesse mercado é aquele de jogos para dispositivos computacionais móveis, principalmente telefones celulares.

Apesar de mais simples que jogos para consoles e microcomputadores, jogos para celulares apresentam também desafios extras para seus desenvolvedores. Dentre tais desafios, provavelmente o mais importante consiste em prover suporte à grande variedade de dispositivos celulares existentes no mercado. Normalmente, isso requer que desenvolvedores de jogos tenham que gerar e manter versões de seus sistemas para diversas plataformas de celulares, de forma a lidar com características particulares dessas plataformas. Dentre essas características pode-se citar: APIs de desenvolvimento proprietárias, tamanho do *display*, quantidade de memória, acessórios disponíveis, processamento, dispositivos de entrada etc. Outras formas de variabilidades também podem ocorrer, como, por exemplo, no caso de diferentes máquinas virtuais criadas por cada fabricante de celular. Com isso, *bugs* podem ser encontrados em algumas plataformas e em outras não. Problemas também existem na distribuição de um jogo para celulares. Na maioria das vezes, a distribuição deve passar por uma operadora de telefonia celular, a qual exige alguns requisitos que os desenvolvedores devem seguir, tais como múltiplas linguagens, tamanho máximo de memória ocupado pela instalação etc.

Atualmente existem várias plataformas de desenvolvimento para celulares, dentre elas podemos citar Brew (BREW,), Symbian (SYMBIAN,), Android (ANDROID,) e J2ME (J2ME,). A escolha de J2ME nesta dissertação se deu pelo motivo de que é atualmente a plataforma mais utilizada para desenvolvimento para celulares e suporta quase todos modelos de celulares do mercado.

Para exemplificar o problema, imagine um jogo que deve ser distribuído para dez modelos de celulares que suportam duas linguagens (inglês e português), como apresentado na Figura 1. Nesse caso, será necessário criar vinte versões diferentes do jogo. Agora suponha, cinco jogos diferentes para cinquenta modelos diferentes de celulares e cinco línguas diferentes. Será necessário gerar 5 x 50 x 5 = 1.250 versões. Certamente, essa quantidade de versões representa um número muito difícil de se gerenciar.

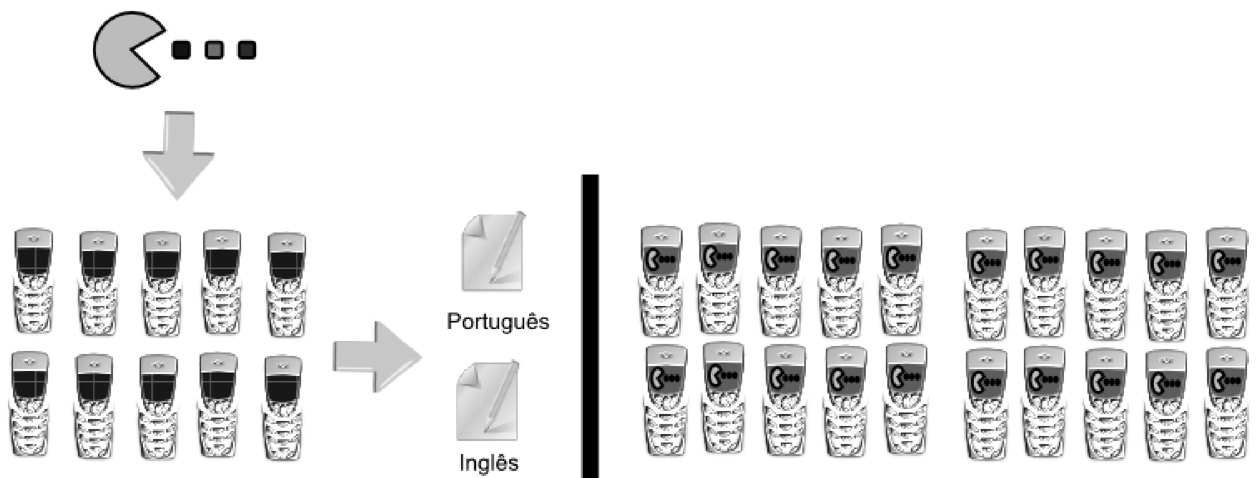


Figura 1: Variações de um jogo.

Assim, jogos para celulares constituem um domínio de aplicação promissor para desenvolvimento baseado em linhas de produto de software (LPS). Basicamente, essa abordagem de desenvolvimento propõe a derivação sistemática de produtos de software a partir de um conjunto de componentes e artefatos comuns (CLEMENTS; NORTHROP, 2001). Para tanto, advoga-se que engenheiros de software devem procurar identificar ao longo de todo processo de desenvolvimento pontos de variabilidade no núcleo de componentes e artefatos comuns, a partir dos quais possam ser derivados produtos específicos. No caso específico de jogos para celulares, essas variabilidades podem incluir o uso de uma API de desenvolvimento proprietária ou então *features* que devem existir em apenas algumas versões do sistema, como explosões, sons, animações mais complexas, *bugs* corrigidos em máquinas virtuais específicas ou traduções dos jogos.

Do ponto de vista de implementação, diversas construções de programação podem ser usadas para apoiar a criação de LPS. Dentre elas podemos mencionar compilação condicional (CC) (RITCHIE; KERNIGHAN, 1988; STROUSTRUP, 2000), orientação por objetos (OO), *mixins* (ANCONA; LAGORIO; ZUCCA, 2003) e orientação por aspectos (AOP) (KICZALES et al., 1997). Porém, recentemente uma nova solução para implementação de LPS chamada de programação orientada por *features* (*Feature-Oriented Programming*, FOP) (BATORY; SARVELA; RAUSCHMAYER, 2003; BATORY, 2004), têm gerado grande interesse na comunidade científica. Programação orientada por *features* é uma técnica de modularização que defende que sistemas devem ser sistematicamente construídos por meio da definição e composição de *features*, as quais são usadas para distinguir os sistemas de uma mesma família de produtos. Para implementação de sistemas utilizando FOP, a ferramenta mais utilizada atualmente chama-se AHEAD (*Algebraic Hierarchical Equations for Application Design*) (BATORY, 2004).

1.2 Objetivos

Essa dissertação tem como objetivo principal avaliar o uso de programação orientada por *features* para construção de LPS na área de jogos para celulares. Alguns passos foram seguidos para essa avaliação, como descrito a seguir:

- Comparação com outras tecnologias: inicialmente descreve-se uma comparação com três tecnologias para desenvolvimento de LPS: programação orientada por aspectos, compilação condicional e programação orientada por *features*. Nesse estudo de caso, realizou-se uma implementação de *features* utilizando essas três tecnologias em um jogo para celular, chamado Chuck Egg[1]. Com as *features* modularizadas, foram realizadas avaliações, baseadas em critérios qualitativos: como configurabilidade, modularidade, reusabilidade e simplicidade e quantitativos como, linhas de código.

- Extração de uma LPS: descreve-se uma experiência de extração de uma LPS na área de jogos para celulares, utilizando um jogo mais complexo (chamado Bomber[2]). Mais especificamente, descreve-se a metodologia utilizada nesse estudo de caso, as *features* propostas, as extrações realizadas e os problemas encontrados para geração de LPS baseada em FOP/AHEAD. Também foram utilizadas algumas métricas para análise da LPS extraída.

- Evolução de uma LPS: descreve-se uma experiência de evolução de uma LPS a partir das *features* extraídas do jogo Bomber. Na dissertação, apresentam-se essas novas *features*, a metodologia utilizada para evolução da LPS e alguns problemas encontrados.

- Avaliação: por fim avalia-se o uso de FOP/AHEAD na extração e evolução da LPS tratada na dissertação. São apresentadas também algumas comparações com tecnologias alternativas para construção de LPS.

1.3 Estrutura da Dissertação

O restante desta dissertação está organizado da seguinte maneira:

- Capítulo 2: realiza-se uma revisão bibliográfica, tratando de temas como LPS, CC, AOP, FOP e AHEAD. Além disso, apresenta-se um estudo comparando o uso de CC, AOP e FOP para implementação de LPS;

[1]Disponível em http://www.morgadinho.org/chuckie/
[2]Disponível em: http://j2mebomber.sourceforge.net.

- Capítulo 3: realiza-se a extração de uma LPS no domínio de jogos para celulares. Neste capítulo, descreve-se a metodologia utilizada, são apresentadas as *features* extraídas, o diagrama de *features*, exemplos de refinamentos realizados nas classes para implementação de *features* utilizando AHEAD e os principais problemas encontrados;

- Capítulo 4: realiza-se uma experiência de evolução da LPS extraída no capítulo anterior, isto é, são acrescentadas novas *features* nesta LPS. Apresenta-se o novo diagrama de *features*, os passos aplicados para realizar a evolução da LPS e problemas encontrados;

- Capítulo 5: realiza-se uma avaliação qualitativa e quantitativa da LPS proposta, incluindo comparações com tecnologias alternativas para implementação de LPS.

- Capítulo 6: conclui esta dissertação, apresentando suas principais contribuições e comparações com trabalhos relacionados. Para finalizar, apresentam-se propostas para trabalhos futuros que possam dar continuidade à pesquisa desenvolvida.

2 LINHAS DE PRODUTO DE SOFTWARE

Linhas de produto de software (LPS) é uma abordagem de desenvolvimento de software que propõe a derivação sistemática de produtos de software a partir de um conjunto de componentes e artefatos comuns (CLEMENTS; NORTHROP, 2001). Para tanto, advoga-se que engenheiros de software devem procurar identificar ao longo de todo processo de desenvolvimento pontos de variabilidade no núcleo de componentes e artefatos comuns, a partir dos quais possam ser derivados produtos específicos. O termo "linha de produto" é usado tradicionalmente na indústria para designar o desenvolvimento seqüencial de produtos, baseado em tarefas repetitivas, executadas sempre pelas mesmas pessoas. Como exemplo pode-se citar linhas de produtos comuns na indústria aeroespacial, automotiva e de componentes eletrônicos. De acordo com (CLEMENTS; NORTHROP, 2001), algumas propostas de desenvolvimento não podem ser caracterizadas como linhas de produto, por exemplo, bibliotecas de objetos, de componentes e de algoritmos.

Com o uso de LPS pode-se obter algumas vantagens como:

Redução no custo de desenvolvimento: artefatos reutilizados em vários sistemas implicam na redução do custo de desenvolvimento individual de cada sistema. Em outras palavras, elimina-se a necessidade de se desenvolver componentes desde o início.

Aumento de qualidade: os ativos do núcleo de uma LPS são reutilizados em vários sistemas. Desta maneira, eles são testados e revisados várias vezes. Ou seja, existe uma grande chance de detectar falhas e corrigi-las, melhorando assim a qualidade final do produto.

Redução do time-to-market: inicialmente o *time-to-market* de uma LPS é alto porque se deve primeiramente desenvolver os ativos do núcleo. Posteriormente, o *time-to-market* é reduzido, pois viabiliza-se que muitos componentes previamente desenvolvidos possam ser diretamente usados para geração de novos produtos.

Em LPS existem alguns conceitos importantes que dão sustentação ao modelo, incluindo: ativos de núcleo, desenvolvimento e domínio. Esses conceitos são descritos abaixo:

Ativos de núcleo: Os ativos são a base da linha de produto e correspondem a um conjunto de elementos customizáveis, utilizados na construção dos softwares produzidos (produtos). In-

cluso nos ativos estão, por exemplo componentes de software, modelos utilizados no processo, padrões de projeto utilizados pela equipe de desenvolvimento, documentação dos requisitos comuns à família de produtos, a arquitetura da linha de produtos, cronogramas etc. Dentre esses elementos, a arquitetura é o elemento chave e normalmente é estudada com mais profundidade do que os outros ativos.

Desenvolvimento do produto: também conhecido como engenharia de aplicação (*Application Engineering*), têm como objetivo a geração de produtos utilizando os ativos de núcleo, assegurando combinações corretas das variações de um produto de acordo com o especificado.

Gerenciamento: é uma atividade de gerenciamento técnico e organizacional na qual se realizam esforços para que a linha de produto não entre em colapso.

A Figura 2 demonstra a relação das três atividades essenciais na construção de LPS, na qual cada círculo representa uma atividade. Todas as três estão interligadas em um contínuo movimento de interatividade. Tanto o desenvolvimento dos artefatos de núcleo e o desenvolvimento do produto ocorrem no contexto da organização na qual a LPS está sendo utilizada. Porém, estas atividades devem ser sustentadas pelo gerenciamento, tanto técnico como organizacional. Não é aplicada nenhuma ordem entre as atividades. Os ativos de núcleo podem ser extraídos de um conjunto de produtos existentes. Existe um forte *feedback* do relacionamento entre essas três atividades, por exemplo: ao se usar os ativos de núcleo em um produto pode-se descobrir pontos específicos onde estes ativos devem ser aprimorados.

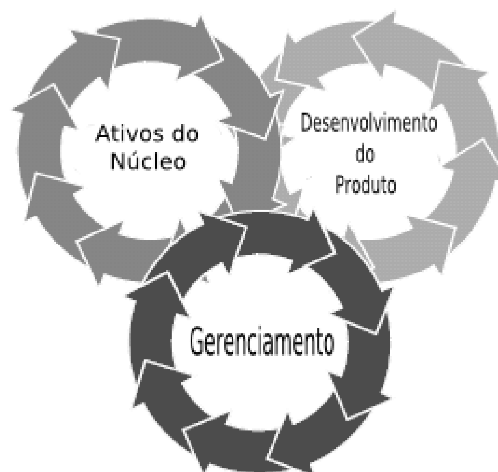


Figura 2: Atividades essenciais de uma LPS (CLEMENTS; NORTHROP, 2001).

2.1 Diagrama de *Features*

O diagrama de *features* (DF) é uma representação gráfica que descreve uma hierarquia de decomposição das *features* de uma LPS em *sub-features* (KANG et al., 1990; BATORY, 2003; JONGE; VISSER; IMPLLANG, 2002). Sendo assim, tem como objetivo apresentar todas as *features* de uma LPS e seus relacionamentos. O DF possui o formato de uma árvore, na qual os nós internos são *features* compostas e suas folhas as *sub-features*. *Sub-features* de uma *feature* composta podem ser obrigatórias, alternativas e opcionais.

A Figura 3 apresenta as notações que são utilizadas na construção de DFs. O **e** indica que todas as *features* filhas devem ser selecionadas, **alternativo** indica que deve ser escolhido exatamente uma das *features*, **ou** permite a escolha de uma ou mais *features*, **obrigatório** a *feature* é obrigatória e **opcional** a *feature* é opcional.

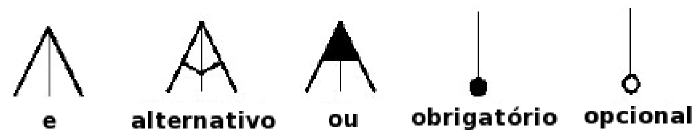


Figura 3: Notações do DF (BATORY, 2005).

A Figura 4 apresenta um exemplo de um diagrama de *features*, na qual mostra-se uma LPS `Carro`, representando diversas configurações possíveis de um carro. A LPS mostrada contêm as *features* `Transmissão`, `Chassi` e `Motor` como *features* obrigatórias. Já a *feature* `Ar` é opcional. A *feature* `Transmissão` possui duas *features* alternativas: `Manual` e `Automática`. A *feature* `Motor` contêm três *features* que podem ser selecionadas alternativamente ou combinadas, ou seja, o carro poderá ter o motor a diesel ou a gasolina ou a álcool ou então a combinação desses três tipos.

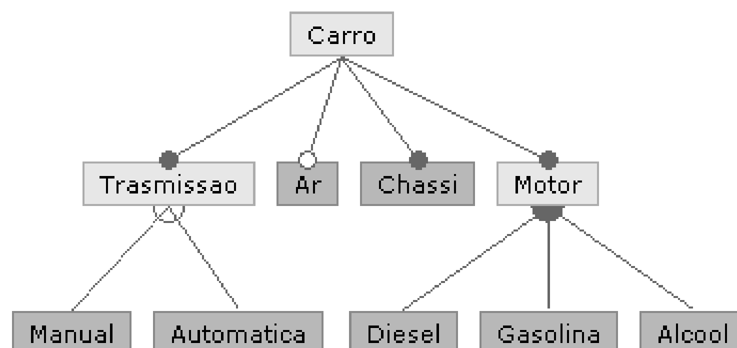


Figura 4: Diagrama de *Features* da LPS `Carro`.

2.2 Tecnologias Analisadas

Descreve-se nesta seção três tecnologias para implementação da LPS considerada nesta dissertação, a compilação condicional, a programação orientada por aspectos e a programação orientada por *features*.

2.2.1 Compilação Condicional

Compilação condicional (CC) é um mecanismo de implementação de variabilidades largamente utilizado em linguagens como C (RITCHIE; KERNIGHAN, 1988) e C++ (STROUSTRUP, 2000). Basicamente, diretivas de pré-processamento são usadas para delimitar linhas do código fonte que devem ser incluídas ou não em uma determinada versão de um sistema. Em Java, não existe suporte nativo a compilação condicional. No entanto, existem ferramentas de terceiros que dão suporte a essa técnica de implementação de variabilidades, como, por exemplo, a ferramenta Antenna[1]. Essa ferramenta utiliza um símbolo especial (//#) para indicar diretivas de compilação, por exemplo a diretiva (#if [expressão]). Essa diretiva é utilizada para que, quando a expressão for válida o código inserido abaixo desta diretiva seja incluído no código base do programa para ser compilado. Para identificar que um código faz parte de uma diretiva deve-se utilizar o símbolo especial (//@) no início da linha. Juntamente com a diretiva (#if) pode-se utilizar as diretivas (#else) e (#elif [expressão]). A diretiva (#else) será executada quando a expressão da diretiva (#if) for falsa. A diretiva (#elif [expressão]) será executada da mesma maneira do (#else), porém testará outra expressão antes de inserir o código ao programa.

A Figura 5 exemplifica a utilização dessas diretivas para delimitar o código responsável pela exibição de mensagens em um determinado sistema. Quando escolhida a diretiva `Portuguese`, a linha 2 é incluída no código. Por outro lado, a linha 4 somente será incluída no código e compilada quando a diretiva `English` for ativada em tempo de compilação.

```
1: //#if Portuguese
2: //@  public String Die="Morreu...";
3: //#elif English
4: //@  public String Die="Die...";
5: //#endif
```

Figura 5: Exemplo de Compilação Condicional

[1] Disponível em http://antenna.sourceforge.net

2.2.2 Programação Orientada por Aspectos

Programação orientada por aspectos (AOP) é uma técnica para separação de interesses transversais presentes no desenvolvimento de sistemas (KICZALES et al., 1997; TIRELO et al., 2004). AOP tem por objetivo modularizar decisões de projeto que não podem ser adequadamente definidas por meio de programação orientada por objetos. Isto se deve ao fato de que alguns requisitos violam a modularização natural do restante da implementação. Na programação orientada por aspectos, requisitos de sistemas são modelados por meio de classes, que implementam objetos do mundo real, e *aspectos*, que implementam requisitos transversais tais como `logging`, persistência, segurança e comunicação. Interesses transversais são interesses que estão espalhados e entrelaçados em diversos módulos de um sistema.

Em orientação por objetos, herança permite a criação de relações do tipo *é-um* e o uso de composição permite a criação de relações do tipo *é-parte-de*. Por exemplo, um objeto da classe `Carro` de um sistema *é-um* objeto da classe `Veículo`, supondo definida uma hierarquia de classes em que `Carro` é subclasse de `Veículo`. Nesse mesmo sistema, um objeto `Roda` *é-parte-de* um objeto `Carro`, se for considerada a composição das classes `Carro` e `Roda`. No entanto, certos elementos de projeto, tais como `Aerodinâmica` ou `Conforto`, não são definidos adequadamente por meio de herança ou composição, pois “atravessam” as decisões de implementação de diversos itens do carro. Tais requisitos são implementados em AOP por meio de *aspectos*.

A Figura 6 ilustra a relação entre alguns dos requisitos de um carro. Nesta representação, percebe-se que os componentes de um carro pertencem a uma dimensão e os requisitos transversais pertencem a uma dimensão distinta, cuja evolução idealmente deve ser independente das demais. Entretanto, em linguagens orientadas por objetos, a implementação destes elementos, naturalmente multidimensionais, deve ser feita em uma única dimensão: a dimensão de implementação dos requisitos funcionais. Em outras palavras, o espaço de requisitos, que é multidimensional, deve ser projetado no espaço de implementação que é unidimensional (TIRELO et al., 2004).

2.2.2.1 AspectJ

Dentre as linguagens com suporte a AOP, AspectJ é atualmente aquela mais madura e estável (KICZALES et al., 2001). AspectJ dá suporte a dois tipos de implementação de requisitos transversais: transversalidade dinâmica, que permite definir implementação adicional em pontos bem definidos do programa e transversalidade estática, que afeta as assinaturas estáticas das classes e interfaces de um programa Java.

No caso de transversalidade dinâmica, AspectJ oferece os seguintes recursos para modularização de interesses transversais: pontos de junção (*join points*), conjuntos de junção (*pointcuts*) e

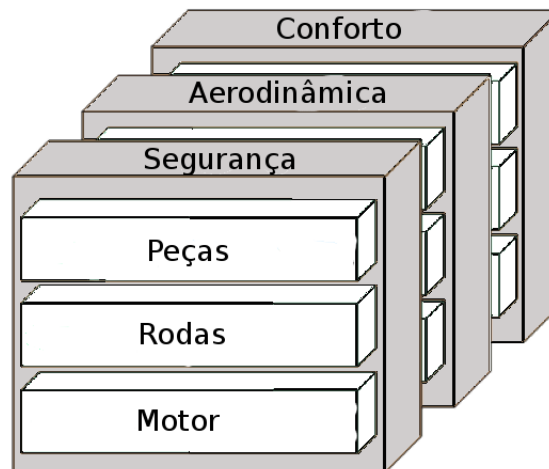


Figura 6: Espaço multidimensional dos requisitos de um carro.

adendos (*advices*). Pontos de junção são pontos da execução do programa. Exemplos de pontos de junção são chamadas de métodos de uma classe. Conjuntos de junção são construções que contêm pontos de junção e têm a função de reunir informações a respeito do contexto desses pontos. Adendos são blocos de código relativos aos requisitos transversais que devem ser executados em pontos de junção. Em AspectJ, é possível definir adendos que serão executados antes (`before`), após (`after`) ou no lugar de (`around`) pontos de junção. Em AspectJ também podem ser definidos os *aspectos*, os quais são semelhantes a classes, exceto pelo fato de possuírem como membros conjuntos de junção e regras de junção.

AspectJ permite selecionar pontos de junção em diferentes contextos como chamadas de métodos e construtores, execução de métodos e construtores, acesso a atributos, tratadores de exceções, definições baseadas no fluxo de controle e definições baseadas na estrutura léxica.

Por fim, como parte de recursos de transversalidade estática de AspectJ, pode-se, por exemplo, introduzir novos campos e métodos nas classes de um sistema. A Figura 7 exemplifica o uso de um aspecto em AspectJ que insere o atributo contendo um texto em português na classe `Base`.

```
1:public aspect Languages {
2: public final static String Base.Die="Morreu...";
3: ...
4:}
```

Figura 7: Exemplo de AOP.

2.2.3 Programação Orientada por *Features*

Programação orientada por *features* (*Feature-Oriented Programming*, FOP) é uma técnica para modularização de *features* que defende que sistemas devem ser sistematicamente construídos por meio da definição e composição de *features*, as quais são usadas para distinguir os

sistemas de uma mesma família de produtos (BATORY; SARVELA; RAUSCHMAYER, 2003; BATORY, 2004).

Em FOP, uma *feature* representa um acréscimo na funcionalidade básica de um programa. FOP advoga que *features* constituem abstrações de primeira classe no projeto de sistemas e, como tal, devem ser implementadas em unidades de modularização independentes. Ou seja, como tradicional em orientação por objetos, classes são usadas para implementar as funcionalidade básicas de um programa. As extensões, variações e adaptações dessas funcionalidades constituem *features*, as quais são implementadas em módulos sintaticamente independentes das classes do programa. Além disso, deve ser possível combinar módulos que representam *features* de forma flexível, sem perder os recursos de verificação estática de tipos. Tipicamente, *features* refinam outras *features* de forma incremental. O refinamento de *features* tem como objetivo encapsular múltiplos fragmentos das classes refinadas. Esses fragmentos podem realizar a inserção ou modificação de métodos, atributos e herança. Comparando com pacotes em Java, os quais encapsulam um conjunto de classes, refinamentos encapsulam fragmentos de múltiplas classes.

A Figura 8 apresenta um pacote com três classes `c1-c3` (na vertical) e os seus refinamentos `r1-r3` (na horizontal). As linhas tracejadas indicam que os refinamentos adicionam transversalmente comportamento extra nas classes `c1-c3`. Uma composição com todos os refinamentos `r1-r3` produz as classes `c1-c3` completas, ou seja, com todas as variações possíveis.

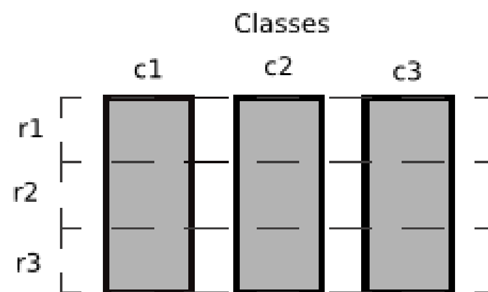


Figura 8: Classes e seus refinamentos.

2.2.3.1 AHEAD

AHEAD (*Algebraic Hierarchical Equations for Application Design*) (BATORY, 2004) é um conjunto de ferramentas que implementa os conceitos básicos de FOP, viabilizando o projeto de sistemas de acordo com os princípios descritos na seção 2.2.3. O principal componente desse ambiente é uma extensão de Java, chamada Jakarta (ou simplesmente Jak), que permite a implementação de *features* em unidades sintaticamente independentes. Por meio dessas unidades, chamadas de refinamentos, pode-se adicionar novos campos e métodos em classes do

programa base. Pode-se ainda adicionar comportamento extra em métodos já existentes.

O AHEAD Tool Suite (ATS) é um conjunto de ferramentas que permite tanto refinamentos de código fonte como de outros artefatos, como por exemplo, JavaDocs, *makefiles*, máquinas de estados etc. A principal ferramenta do ATS é o `composer`, o qual recebe a aplicação base e um conjunto de *features* como entrada e gera como saída uma aplicação base com a composição de suas *features*. Um arquivo `.jak` define os refinamentos, os quais são identificados pela palavra reservada `refines`. A Figura 9 mostra o uso de um refinamento que insere um atributo contendo uma mensagem em português na classe `Language`. Nessa figura, a palavra reservada `layer`

```
1: layer Portuguese;
2: public refines class Language {
3:  public  final static String Die="Morreu...";
4: }
```

Figura 9: Exemplo de refinamento em Jakarta.

identifica a *feature* a qual esse refinamento pertence. Em ATS, *features* são implementadas em diretórios, assim como pacotes em Java. Esses diretórios contêm *refinamentos* das classes do programa base. Sendo assim, uma composição de *features* é uma composição de diretórios. Podemos representar uma nova aplicação por um conjunto de equações em AHEAD. O operador especial • é usado para denotar uma composição de *features*. A seguir, são apresentados alguns exemplos destas equações que representam novos produtos da LPS:

1. jogoPort = Portuguese • Base

2. jogoEng = English • Base

A composição de *features* é aplicada da direita para esquerda. Ou seja, como exemplo, para gerar o produto `jogoPort` aplica-se a composição `Portuguese` na `Base`. Para compor então esse produto é utilizado a seguinte linha de comando:

```
$> composer --target=jogoPort Base Portuguese
```

O nome do produto é dado pelo parâmetro `--target` e a ordem da composição como dito anteriormente ocorre da direita para esquerda. Com isso, é gerado o diretório `jogoPort` com a versão do produto especificado. Os arquivos do produto `jogoPort` gerado via composição ainda estão no formato do Jakarta com a extensão `.jak`. Para compilar esses arquivos, é necessário que se converta os arquivos `.jak` para `.java`. Para isso, deve-se utilizar a ferramenta `jak2java`.

2.2.3.2 Regras de projeto

Um problema em FOP é garantir composições válidas e significantes de *features* (BATORY, 2005). Nem todas as combinações de *features* são válidas. Então, para tratar esse problema são

utilizadas regras de projeto (*design rules*) que são restrições de domínio específico por meio das quais são definidas regras de composições de *features*. Mais especificamente, utiliza-se uma *tree grammar* para definir combinações válidas de *features* (BATORY, 2005; KANG et al., 1990). Essas gramáticas requerem que cada símbolo terminal apareça em exatamente uma única produção. Além disso, elas utilizam iteração para expressar repetição, em vez de recursão. Por exemplo, `t+` representa uma ou mais repetições de `t`; já `t*` representa zero ou mais repetições de `t`. *Features* opcionais são indicadas entre colchetes. Por exemplo, `[t]` representa que a *feature* `t` é opcional.

A Figura 10 apresenta um exemplo de *tree grammar* para o DF mostrado na Figura 4. A gramática inicia com a produção raiz (`Carro`), a qual gera os símbolos `Transmissão`, `Ar`, `Chassi` e `Motor` que representam as *features* da LPS. Nota-se que `Ar` é uma *feature* é opcional. Como `Trasmissao` é uma *feature* composta, suas *sub-features* são representadas em um nova produção. Nota-se que essa produção utiliza o símbolo (|) para representar **ou** exclusivo, ou seja, pode-se utilizar somente a *feature* `Manual` ou a *feature* `Automatica`. O mesmo caso ocorre na produção `Motor`, porém como identificado na produção `Carro` as opções disponíveis de motor podem ser combinadas. Isto é, um carro *flex* pode possuir um motor a gasolina, álcool e diesel.

```
Carro : Transmissao [Ar] Chassi Motor+
Transmissao : Manual | Automatica
Motor : Diesel | Gasolina | Alcool
```

Figura 10: Gramática da LPS `Carro`.

ATS possui uma ferramenta GUI que permite que o usuário defina configurações para geração de um produto de modo visual, chamada de **guidsl** (BATORY, 2004). Essa ferramenta tem como entrada a gramática que representa a LPS. Um exemplo desta GUI com a gramática descrita anteriormente é apresentado na Figura 11. O usuário seleciona as *features* que deseja e a ferramenta habilita ou desabilita as *features* impedindo que sejam gerados programas incorretos. Dada as escolhas das *features*, pode ser gerado um arquivo `.equation` que contém os nomes das *features* que serão compostas utilizando o **composer**.

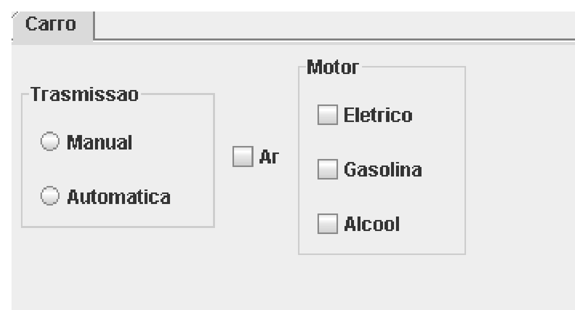


Figura 11: Exemplo de tela da ferramenta `guidsl`.

Existe também um *plugin* Eclipse, chamado FeatureIDE (LEICH et al., 2005), que dá su-

porte à construção e organização de linhas de produtos com o ATS. Esse *plugin* permite que se crie o DF da LPS e automaticamente a gramática desta LPS. Também é possível compilar todas as *features* selecionadas, de forma a gerar um produto final.

2.3 Estudo de Caso

Nesta seção demonstra-se o uso das três tecnologias descritas na seção 2.2 na modularização de uma *feature* tradicional em jogos para celulares: efeitos sonoros. Para isso, foi utilizado como estudo de caso uma implementação de código livre em J2ME do jogo Chuckie Egg[2]. Chuckie Egg é um jogo clássico da década de 80 cujo objetivo é fazer com que o personagem principal recolha a maior quantidade possível de ovos, desviando-se de galinhas que ficam protegendo os mesmos. Uma tela do jogo pode ser vista na Figura 12. A versão considerada possui 1519 linhas de código Java/J2ME e 15 classes.

Figura 12: Tela do jogo Chuckie Egg

A versão original do jogo Chuckie Egg não possui suporte a efeitos sonoros. Assim, para realização da demonstração proposta nesta seção, resolveu-se incorporar essa *feature* no sistema usando as três tecnologias para implementação de variabilidades descritas na Seção 2.2. Basicamente, foram inseridos sons com músicas de fundo durante todo o jogo e efeitos sonoros quando o personagem principal pula sobre uma das galinhas e quando ele captura algum objeto.

Como o objetivo é meramente demonstrativo, uma única classe do sistema será usada no restante desta seção para ilustrar as implementações realizadas. A classe escolhida, chamada `Chuckie`, é responsável pela manipulação do personagem principal, incluindo a implementação

[2]Disponível em http://www.morgadinho.org/chuckie/

de ações como pular, andar, subir escadas etc. Nesta classe, foi introduzido um efeito sonoro toda vez que o personagem principal executa um pulo.

Um fragmento do código da classe `Chuckie` é apresentado na Figura 13. Essa classe possui métodos que tratam os pulos do personagem principal, incluindo pulos em que ele permanece parado (linhas 6-8), pulos em que ele se move para a direita (linhas 9-11) e pulos em que ele se move para a esquerda (linhas 12-14). Utilizando as três tecnologias para implementação de variabilidades tratadas na seção, descreve-se nas subseções seguintes como esses métodos podem ser instrumentados para incorporar a *feature* de som proposta no estudo de caso.

```
 1: public class Chuckie extends GameSprite {
 2:   .....
 3:   public Chuckie(Image img, int x, int y) {
 4:     .....
 5:   }
 6:   public void jump() {
 7:     .....
 8:   }
 9:   public void jump_right() {
10:    .....
11:   }
12:   public void jump_left() {
13:    .....
14:   }
15: }
```

Figura 13: Fragmento de código da classe `Chuckie`

2.3.1 Compilação Condicional

Na Figura 14 apresenta-se a versão da classe `Chuckie` que usa-se CC para implementação da *feature* de Som. Inicialmente, usa-se uma diretiva de compilação para declarar um atributo do tipo `PlaySounds`, o qual faz o tratamento de som (linhas 2-4). Esse atributo é inicializado nas linhas finais do construtor da classe (linhas 8-10). Introduz-se também na classe um método que executa o som de pulo (linhas 12-20). Esse método foi criado para evitar repetições de código. Assim, nos métodos que implementam o pulo do personagem principal foram apenas inseridas chamadas ao método introduzido (linhas 22-24, 28-30 e 34-36). O som será executado logo no início do corpo desses métodos.

Assim, caso a *feature* `Sound` e sua respectiva diretiva de compilação sejam habilitadas em tempo de compilação, será gerada uma versão do sistema em que um efeito sonoro ocorre toda vez que o personagem principal realizar um pulo. Como pode ser observado, utilizando diretivas de compilação condicional, o código da *feature* fica entrelaçado no código base, aumentando seu tamanho e, portanto, dificultando o entendimento.

```
1: public class Chuckie extends GameSprite {
2:   //#if Sound
3:   //@ protected PlaySounds playJump;
4:   //#endif
5:   .....
6:   public Chuckie(...) {
7:     .....
8:     //#if Sound
9:     //@ playJump = new PlaySounds(...);
10:    //#endif
11:  }
12:  //#if Sound
13:  //@ protected void playJump() {
14:  //    try{
15:  //@     playJump.play();
16:  //@   } catch (Exception e) {
17:  //@     .....
18:  //@   }
19:  //@ }
20:  //#endif
21:  public void jump() {
22:    //#if Sound
23:    //@ playJump();
24:    //#endif
25:    .....
26:  }
27:  public void jump_right() {
28:    //#if Sound
29:    //@ playJump();
30:    //#endif
31:    .....
32:  }
33:  public void jump_left() {
34:    //#if Sound
35:    //@ playJump();
36:    //#endif
37:    .....
38:  }
39:  .....
40:}
```

Figura 14: Implementação da *feature* `Sound` usando compilação condicional

2.3.2 Aspectos

A Figura 15 apresenta um aspecto que modulariza a *feature* de som utilizando AspectJ. Este aspecto atua no código base da classe `Chuckie`. Inicialmente, o aspecto proposto introduz um atributo do tipo `PlaySounds` nessa classe (linha 2) e declara conjuntos de pontos de junção que interceptam os métodos de pulo da classe (linhas 3-11). O aspecto também declara um conjunto de junção que intercepta instanciações de objetos da classe `Chuckie` (linhas 12-13). Um adendo associado a esse conjunto de junção inicializa o atributo que foi introduzido na classe (linhas 14-16). O aspecto proposto contém também um segundo adendo, que executa o som de pulo antes dos pontos de junção que denotam execuções de métodos de pulo da classe

`Chuckie` (linhas 17-23).

Na implementação baseada em aspectos não foi necessário criar um método de execução de som, como no caso da implementação baseada em compilação condicional. O motivo é que o corpo desse método encontra-se contido no adendo das linhas 17-23. Além disso, na solução apresentada foi possível modularizar integralmente a *feature* de Som em um único aspecto, evitando-se assim o seu entrelaçamento no código base do sistema.

```
1: public aspect Sound {
2:   PlaySounds Chuckie.playJump;
3:   pointcut soundJump():
4:     execution (* Chuckie.jump());
5:   pointcut soundJumpDir():
6:     execution (* Chuckie.jump_right());
7:   pointcut soundJumpLeft():
8:     execution (* Chuckie.jump_left());
9:   pointcut soundJumps(Chuckie o):
10     (soundJump() || soundJumpDir() ||
11:    soundJumpLeft()) && target(o);
12:  pointcut soundJumpsCreate (Chuckie o):
13:    execution (Chuckie.new(..)) && target(o);
14:  after(Chuckie o): soundJumpsCreate (o){
15:    o.playJump= new PlaySounds(...);
16:  }
17:  before(Chuckie o): soundJumps(o){
18:    try {
19:      o.playJump.play();
20:    } catch (Exception e) {
21:     ...
22:    }
23:  }
24:}
```

Figura 15: Implementação da *feature* `Sound` usando aspectos

2.3.3 AHEAD

A Figura 16 mostra o refinamento da classe `Chuckie` responsável pela implementação da *feature* de Som. O refinamento mostrado é semelhante a uma classe de Java. Porém, utiliza-se a palavra reservada `refines` (linha 1), a qual identifica que o mesmo é um refinamento de uma classe já existente no sistema. O refinamento mostrado introduz na classe refinada um atributo para tratamento de som (linha 2). O construtor da classe base também é refinado para incluir a inicialização desse atributo (linhas 3-5). A exemplo da solução baseada em compilação condicional, também foi implementado um método, chamado `playJump`, responsável por executar o som de pulo (linhas 6-12). Por fim, para cada método que executa uma ação de pulo inclui-se uma chamada ao método `playJump` (linhas 14, 18 e 22). Nesses casos, para especificar que o código original do método base deve ser executado logo após o tratamento de som, utiliza-se a palavra reservada `Super` (linhas 15, 19 e 23).

```
 1: public refines class Chuckie {
 2:   protected PlaySounds playJump;
 3:   refines Chuckie(...){
 4:     playJump= new PlaySounds(...);
 5:   }
 6:   protected void playJump() {
 7:     try {
 8:       playJump.play();
 9:     } catch (Exception e) {
10:      .....
11:     }
12:   }
13:   public void jump() {
14:     playJump();
15:     Super.jump();
16:   }
17:   public void jump_right() {
18:     playJump();
19:     Super.jump_right();
20:   }
21:   public void jump_left() {
22:     playJump();
23:     Super.jump_left();
24:   }
25: }
```

Figura 16: Implementação da *feature* `Sound` usando refinamentos

2.4 Avaliação

Com base na experiência adquirida no estudo de caso descrito na Seção 2.3, apresenta-se a seguir uma avaliação das três tecnologias para implementação de variabilidades incialmente consideradas no trabalho. Essa avaliação é baseada nos seguintes critérios:

- Configurabilidade. Em desenvolvimento baseado em LPS, configurabilidade diz respeito à capacidade de se selecionar as *features* que serão incorporadas em um determinado produto (ou versão) da LPS. De acordo com esse critério, todas as três tecnologias permitem gerar de forma ágil versões do jogo Chuckie Egg com ou sem som. No caso de compilação condicional, basta ativar a diretiva de compilação utilizada para representar a *feature* Som. No caso de aspectos, basta combinar (ou não) o aspecto de som com o código base do sistema. O mesmo ocorre com o refinamento de som apresentado na Figura 16, que pode ser combinado ou não com sua classe base.

- Modularidade. Na implementação baseada em CC, o código responsável pela implementação da *feature* Som fica entrelaçado no código base do sistema conforme mostrado na Figura 14. Ou seja, não se obtém uma boa modularização e separação de interesses. Por outro lado, nas soluções baseadas em AOP e FOP, o código da *feature* Som é implementado em um módulo distinto. Com isso, não "polui-se" o código base com código relativo

à implementação de interesses que não são mandatórios no sistema, como é o caso de efeitos sonoros.

- Reusabilidade. Essa característica diz respeito à capacidade de se reutilizar o código de uma *feature* em outros sistemas. Segundo o estudo de caso realizado, trata-se de um ponto negativo de todas as três tecnologias analisadas. Usando compilação condicional, o código da *feature* não é implementado em um módulo distinto, que possa ser reusado em outros sistemas. Por outro lado, apesar disso ocorrer nas soluções baseadas em AOP e FOP, os novos módulos criados referenciam diretamente diversos elementos do código base do sistema. Em outras palavras, existe um acoplamento entre tais módulos e as classes do sistema Chuckie Egg. Por exemplo, no aspecto da Figura 15, existem referências a diversos membros da classe `Chuckie`, como os métodos `jump`, `jump_right` e `jump_left`. Referências semelhantes ocorrem no refinamento apresentado na Figura 16. A existência de tais acoplamentos impede a reutilização dos aspectos e refinamentos mostrados em novos sistemas.

- Simplicidade e facilidade de aprendizagem. Dentre as três tecnologias consideradas, compilação condicional é a mais simples e de mais fácil domínio. Por outro lado, com base na experiência realizada, considera-se que FOP/AHEAD é uma tecnologia mais simples que AOP/AspectJ. Por exemplo, em AHEAD não é necessário declarar conjuntos de junção, definir o tipo de adendo (`after`, `before` ou `around`), associar adendos a conjuntos de junção etc. Em vez disso, um refinamento tem acesso direto ao ambiente da classe refinada, o que simplifica a sua sintaxe.

- Tamanho do código. A Tabela 1 apresenta informações sobre o tamanho das três versões do jogo Chuckie Egg. Conforme pode ser observado nessa tabela, o maior acréscimo de linhas de código – em relação à versão original do jogo – ocorre na versão baseada em compilação condicional (+145 LOC), seguida pela versão baseada em FOP (+ 120 LOC) e depois pela versão baseada em AOP (+98 LOC).

	LOC	Acréscimo
Versão original	1519	–
Versão baseada em CC	1664	+ 145
Versão baseada em AOP	1617	+ 98
Versão baseada em FOP	1639	+ 120

Tabela 1: Tamanho (em LOC)

A Tabela 2 resume os principais resultados da avaliação realizada. Conforme pode ser observado nessa tabela, apesar de o jogo Chuckie Egg ser um sistema simples, pode-se concluir que existem benefícios no uso de FOP e AOP para implementação de variabilidades típicas de jogos para celulares. Essas vantagens são mais claras quando se compara FOP e AOP

com tecnologias mais tradicionais, e mais largamente utilizadas, como compilação condicional. Comparando-se especificamente FOP/AHEAD com AOP/AspectJ, considera-se que a principal vantagem do sistema AHEAD reside na simplicidade de sua linguagem para separação de interesses. Por outro lado, a principal desvantagem de AHEAD é a inexistência na linguagem de recursos de quantificação. Um refinamento sempre estende o comportamento de um método de uma classe do programa base. Já em AspectJ, recursos de quantificação permitem definir adendos que atuarão em múltiplos pontos de junção do programa base, o que é particularmente útil para implementação de requisitos transversais homogêneos, isto é, requisitos que requerem a implementação de um mesmo código em múltiplos pontos de um sistema.

	CC	AOP	FOP
Configurabilidade	+	+	+
Modularidade	-	+	+
Reusabilidade	-	-	-
Simplicidade	+	-	+/-
Tamanho do código	-	+	+/-

Tabela 2: Comparação entre as tecnologias CC, AOP e FOP

2.5 Comentários Finais

Nesta seção, foram descritas e comparadas três técnicas para modularização de variabilidades em jogos para celulares. Como estudo de caso, utilizou-se um jogo para celular implementado em J2ME (Chuckie Egg). Uma nova *feature* para tratamento de som foi acrescentada a esse jogo utilizando três diferentes tecnologias: compilação condicional, programação orientada por aspectos e programação orientada por *features*. Após a comparação realizada, fica claro que tecnologias modernas de implementação de variabilidades – como aspectos e refinamentos – oferecem ganhos importantes em relação a tecnologias mais tradicionais, como compilação condicional. Esses ganhos ocorrem basicamente em critérios como modularidade e tamanho final do código. No restante desta dissertação, descreve-se uma experiência de extração e evolução de uma LPS mais complexa e com um conjunto maior de *features*. A LPS a ser descrita também tem como objetivo modularizar *features* de um jogo para celular. Para sua implementação, usou-se programação orientada por *features*, tal como proposto pelo ambiente AHEAD. O objetivo final é documentar e avaliar os principais benefícios e limitações dessa tecnologia.

3 EXTRAÇÃO DE UMA LPS UTILIZANDO PROGRAMAÇÃO ORIENTADA POR FEATURES

Neste capítulo, descreve-se uma experiência de extração de uma LPS na área de jogos para celulares. Serão apresentados o estudo de caso, a metodologia utilizada, as *features* propostas, as extrações realizadas e os problemas encontrados. O jogo escolhido foi o Bomber[1]. O objetivo desse jogo é bem simples: pilotar um avião e lançar bombas em lugares demarcados, tendo que desviar de inimigos como aviões, *zeppelins*, tanques de guerra, submarinos etc. Bomber foi originalmente criado para celulares Nokia série 60. Posteriormente, o sistema foi portado para trabalhar com J2ME MIDP 2.0, sendo o código fonte dessa sua última versão aberto. Algumas imagens do jogo são mostradas na Figura 17.

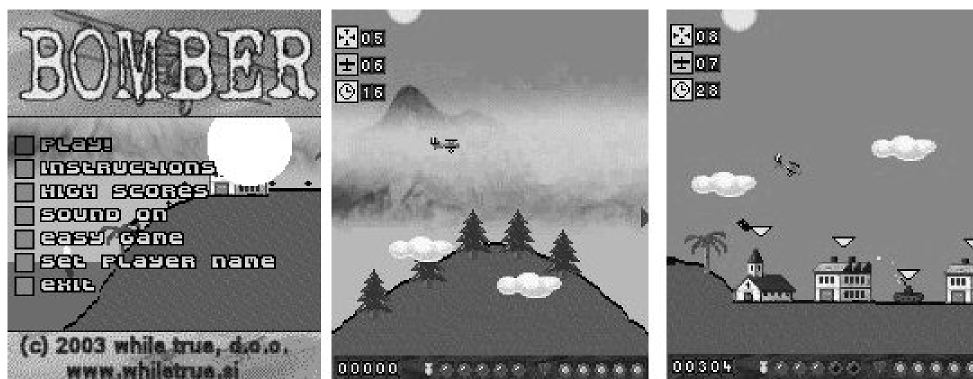


Figura 17: Telas do jogo Bomber

3.1 Arquitetura do Sistema

O jogo Bomber possui 7784 LOC sub-divididas em 9 interfaces, 2 classes abstratas e 40 classes, totalizando 51 classes. A seguir será apresentado uma breve descrição de algumas das principais classes do jogo:

- `BomberMIDlet`: classe inicial do jogo. Em J2ME é a classe responsável por tratar o ciclo de vida do aplicativo
- `ResourceManager`: classe responsável pelo tratamento de recursos do jogo, como, por exemplo, o carregamento de fases, imagens, sons etc

[1]Disponível em: http://j2mebomber.sourceforge.net.

- `GameObject`: classe base da qual a maioria dos objetos herdam e possuem alguns principais tratamentos, como, por exemplo, colisões entre os objetos
- `Game`: classe responsável por tratar o estado do jogo e possui todos os objetos do jogo a serem tratados
- `NokiaTerrain`: classe responsável por tratar e desenhar o terreno do jogo
- `Plane`: classe responsável pelo tratamento do avião do jogador
- `AIPlane`: classe responsável pelo tratamento de inteligência do avião inimigom, –movimenta o avião inimigo em busca do avião do jogador–
- `Building`: classe responsável pelo tratamento das construções que devem ser destruídas no jogo, como, por exemplo, as casas, igrejas, galpões etc
- `Debris`: classe responsável pelo tratamento das ruínas das construções
- `GameCanvas`: classe responsável pelo tratamento de entradas/saídas do jogo, como, por exemplo, o pressionamento de uma tecla, a visualização de uma imagem no display do celular etc
- `Storage`: classe responsável pelo armazenamento de dados do jogo, como, por exemplo, rankings, nomes dos jogadores etc
- `Smoke, Splash, Explosion`: tratam respectivamente efeitos de fumaça, efeitos de respingos e explosões
- `Zeppelin, Tank`: representam respectivamente o Zeppelin e o tanque

As classes descritas anteriormente são apresentadas no diagrama de classes da Figura 18.

3.2 Metodologia

No estudo realizado, uma abordagem extrativa foi empregada para gerar uma versão do jogo Bomber baseada em LPS (KRUEGER, 2002). Analisando as funcionalidades do sistema, foram identificados os seguintes conjuntos de *features*:

- Efeitos Visuais, incluindo imagens e efeitos de animação opcionais, como nuvens, explosões, danos ao terreno etc. Essas *features* são opcionais, visto que elas apenas enriquecem o jogo.

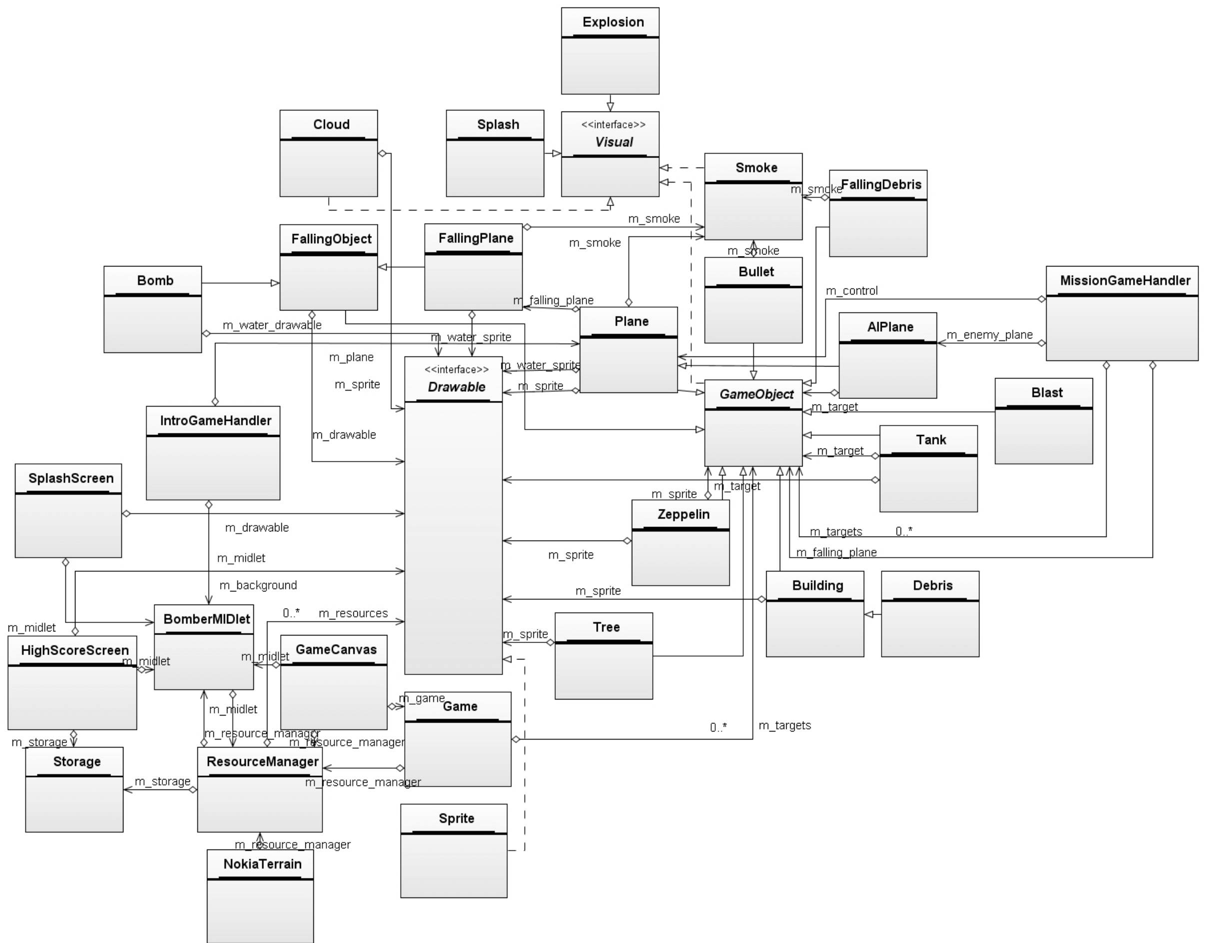


Figura 18: Diagrama de classes do Bomber.

- Som, incluindo efeitos sonoros implementados por meio da API padrão da plataforma J2ME ou usando APIs proprietárias de fabricantes de celulares. Essas *features* são opcionais, visto que é possível utilizar o jogo sem efeitos sonoros.

- Mensagens, incluindo exibição de mensagens em mais de uma língua. Nesse caso, pelo menos uma dessas *features* deve ser selecionada na geração de um produto, visto que mensagens informativas são fundamentais para entendimento do jogo.

- Base, contendo funcionalidades essenciais em qualquer instância do jogo. Essas funcionalidades foram definidas por exclusão, isto é, são todas as funcionalidades do sistema, exceto aquelas incluídas nos conjuntos anteriores.

O modelo de *features* da família de produtos proposta é mostrado na Figura 19. Nesse diagrama, define-se que em qualquer instância da linha de produto: (1) o conjunto de *features* `Base` é mandatório; (2) `VisualFX` é uma *feature* opcional, incluindo *subfeatures* responsáveis pela movimentação de nuvens e por efeitos visuais em explosões; (3) `Language` é uma *feature* mandatória, exigindo que se escolha exatamente uma de suas *subfeatures* (no caso, mensagens em português ou em inglês); (4) `SoundFX` é uma *feature* opcional; no entanto, caso essa `feature` seja selecionada, obrigatoriamente deve-se incorporar um conjunto de classes básicas para manipulação de som e classes específicas de uma API, no caso, J2ME ou Nokia.

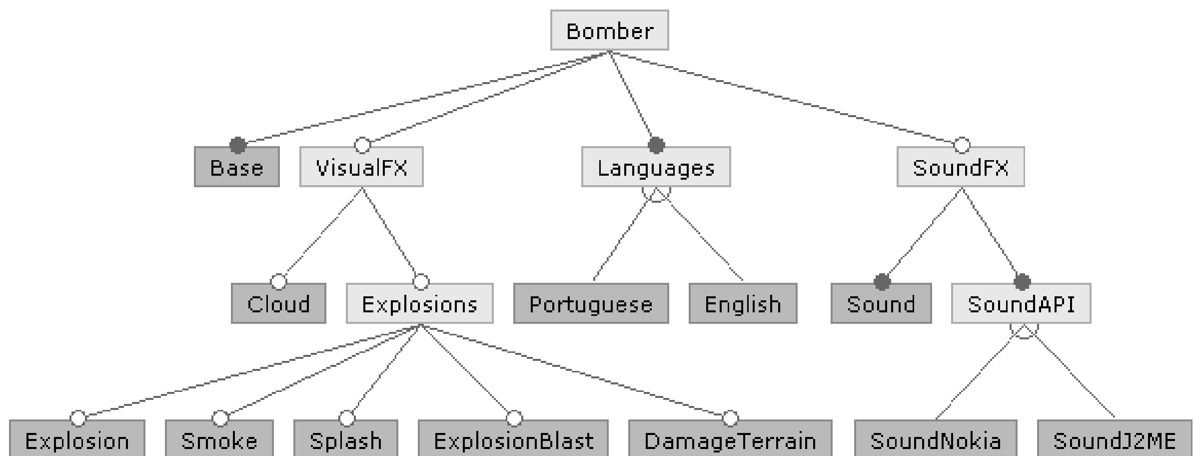


Figura 19: Modelo de *Features*

A fim de extrair a LPS representada pelo modelo de *features* da Figura 19, os seguintes passos foram seguidos:

1. Linhas do código fonte da versão original do sistema foram manualmente rotuladas – por meio de comentários – como pertencendo a um dos nodos folha do modelo de *features* da Figura 19 (de acordo com a funcionalidade do sistema implementada pela linha). A

única exceção são as linhas pertencentes ao módulo `Base`, as quais não foram rotuladas para não carregar o código com diversos comentários.

A Tabela 3 apresenta o número de linhas de código rotulada para cada *feature*. As *features* que não existem na versão original do sistema aparecem com o LOC zerado. Como pode ser observado nesta tabela, a `Base` corresponde a 91,8% das linhas de códigos da versão original do jogo.

Feature	LOC Original
Base	7147
English	50
Portuguese	0
Sound	53
SoundJ2ME	0
SoundNokia	0
Cloud	60
Explosion	15
ExplosionBlast	123
Smoke	197
Splash	108
DamageTerrain	31
Total	7784

Tabela 3: Número de linhas rotuladas para cada *feature*.

2. Refinamentos foram criados para modularizar a implementação de cada uma das *features* previstas na linha de produtos. Suponha, por exemplo, uma *feature* F cuja implementação original esteja espalhada pelas classes $C_1, C_2, \ldots, C_n$. Basicamente, o código não-modularizado de F foi extraído do código base do sistema e migrado para refinamentos $R_1, R_2, \ldots, R_n$. A Tabela 4 apresenta o número de classes que cada *feature* refina. Como pode ser observado nesta tabela, excetuando a `Base`, o número de classes refinadas pelas *features* propostas é relativamente baixo, variando de uma a oito classes.

3.3 Extração da LPS

Descreve-se a seguir a aplicação desses passos em uma das classes do jogo Bomber. A classe `Bomb` será utilizada como exemplo para demonstrar a extração tanto da *feature* `SoundFX`, como da *feature* `VisualFX`. A Figura 20 mostra o código da classe `Bomb` após inserção de comentários indicando a presença de *features* (linhas 5-6 e 10-12). A Figure 21 mostra o código dessa mesma classe após extração do código das *features* identificadas. Em relação ao código mostrado nessa figura, duas observações são relevantes. Primeiro, o método `enterWater` não mais existe na classe `Bomb`, visto que após a extração das *features* seu corpo se restringiu a

Feature	Classes Refinadas
Base	51
English	1
Portuguese	1
Sound	8
SoundJ2ME	1
SoundNokia	1
Cloud	4
Explosion	7
ExplosionBlast	6
Smoke	5
Splash	7
DamageTerrain	7

Tabela 4: Números de classes refinadas pelas *features*.

```
 1:class Bomb extends FallingObject {
 2:  ...
 3:  void enterWater() {
 4:    super.enterWater();
 5:    m_game_state.createSplash(...);                      // Splash
 6:    m_game_state.getResourceManager().sound(...);        // Sound
 7:  }
 8:  void onGround() {
 9:    if (m_state != DESTROYED) {
10:      m_game_state.createBlast(...);                     // ExplosionBlast
11:      m_game_state.getTerrain().crater(...);             // DamageTerrain
12:      m_game_state.getResourceManager().sound(...);      // Sound
13:      m_state = DESTROYED;
14:    }
15: }
16: ...
17:}
```

Figura 20: Classe Bomb da versão original com comentários indicando *features*

uma chamada ao método de mesmo nome da superclasse. Segundo, foi necessário extrair um método, chamado `setBombState` (linhas 8-10) na Figura 21, contendo apenas o código que indica que o estado da bomba agora é destruído (linha 9). Essa extração é necessária porque AHEAD não permite refinar comandos internos de um método, mas apenas o seu corpo. Como o código das *features* `ExplosionBlast`, `DamageTerrain` e `Sound` encontra-se entrelaçado em um comando `if` do método `OnGround` (linhas 10-12 da Figura 20), a solução encontrada foi extrair os comandos internos do `if` para o método `setBombState`. O método extraído será refinado para reintroduzir o código das *features* `ExplosionBlast`, `DamageTerrain` e `Sound` na classe `Bomb`.

Nas próximas seções serão demostradas as refatorações e refinamentos que foram realizadas especificamente para implantação das *features* `SoundFX` e `VisualFX`.

```
1: class Bomb extends FallingObject {
2:   ...
3:  void onGround() {
4:    if (m_state != DESTROYED) {
5:       setBombState();
6:    }
7:  }
8:  protected void setBombState() {
9:    m_state= DESTROYED;
10: }
11:   ...
12: }
```

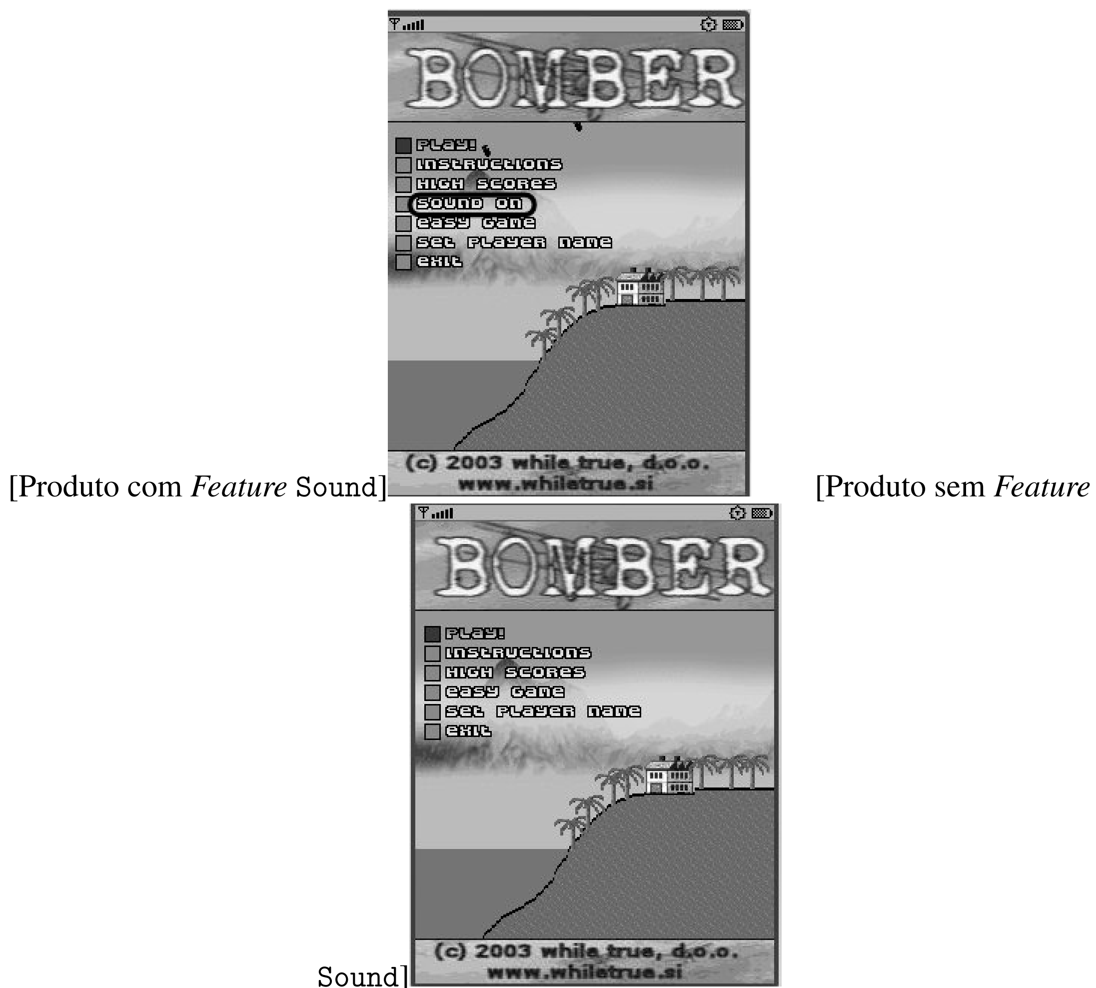
Figura 21: Classe `Bomb` refatorada (com apenas funcionalidades básicas)

3.3.1 *Feature* SoundFX

Discutiremos nessa seção as refatorações e refinamentos da *feature* `SoundFX`. Refatorações significativas relativas a essa *feature* foram necessárias, como, por exemplo, a opção no menu inicial do jogo para que o usuário possa desabilitar ou habilitar o som. Claramente, essa opção deve somente aparecer quando a *feature* `Sound` for selecionada para o produto. O menu do jogo com a *feature* `Sound` é apresentado na Figura 22a e o menu sem essa *feature* é apresentado na Figura 22b.

Para implementação do menu são utilizados basicamente dois métodos: um que captura a opção do menu que está sendo selecionada pelo usuário e outro método que faz a construção do menu na tela. O código desses dois métodos, pertencentes à classe `IntroGameHandler`, é apresentado na Figura 23. O método `handleCommands` (linhas 4-13) captura qual opção do menu foi selecionada. No exemplo, somente é apresentado o código que implementa a seleção da opção de som (linhas 8-10) no menu. Por fim, o método `buildMenu` (linhas 18-31) exibe o menu na tela inicial do jogo.

A primeira refatoração significante foi extrair o `switch` (linhas 6-12) da Figura 23 para um novo método e trocá-lo por `ifs`. A extração do código do `switch` para um novo método se deve ao fato de não ser possível o refinamento de um código interno a um método. A mudança do `switch` para vários `ifs` é necessária para que seja possível a inclusão de novas condições, como a condição de tratamento de som. Outra refatoração interessante foi do método `buildMenu`. Nota-se que em `buildMenu` cada opção do menu e armazenada em uma posição fixa de um vetor (linhas 15-25). Então, a simples retirada do código referente a *feature* `Sound` (linhas 19-20) causaria um problema na visualização do menu, pois o elemento da posição três não existirá nos produtos que não tiverem a *feature* `Sound` ativada. Para resolver esse problema, foi inicialmente criada uma classe `ArrayList` para tratar listas de objetos, já que em J2ME não existe esta classe. Feito isso, o vetor de strings foi substituído por um objeto do tipo `ArrayList`, permitindo que seja inserida uma opção do menu na ordem que se desejar.



[Produto com *Feature* `Sound`] [Produto sem *Feature* `Sound`]

Figura 22: Menu de opções do jogo.

A Figura 24 apresenta o código da classe `IntroGameHandler` pertencente a `Base` com seus métodos refatorados e com as *features* extraídas. O método `trataMenu` é um método que foi extraído para ser posteriormente refinado. Em `trataMenu`, o valor de `m_selected` – que anteriormente era comparado no `switch` (linha 6, Figura 23) – agora é utilizado para retornar uma string referente à opção do menu (linha 11) Figura 24. Com a string retornada, compara-se qual opção foi selecionada (linhas 12-16), já o método `buildMenu` contém o novo tratamento do menu com a utilização do `ArrayList`.

A Figura 25 apresenta parte do módulo responsável pela *feature* `Sound` na LPS proposta. O código mostrado contém um refinamento que introduz efeitos sonoros ao se executar os métodos `enterWater` (linhas 2-5) e `setBombState` (linhas 6-9) da classe `Bomb`.

O código da *feature* `Sound` que implementa a opção de som no menu do jogo é apresentado na Figura 26. Essa *feature* refina os métodos `buildMenu` (linhas 2-8) e `trataMenu` (linhas 9-178). Como afirmado anteriormente, após as refatorações é possível no método `buildMenu` escolher uma posição em que se deseja que apareça uma opção. Nesse refinamento, a posição 3 foi a escolhida para inserir uma opção no menu (linha 3). Posteriormente, o método `add` de

```
1:public class IntroGameHandler extends BasicGameHandler {
2: ...
3: private String[]  m_menu;
4: private void handleCommands(byte cmd) {
5:   try {
6:    switch (m_selected) {
7:    ...
8:    case 3: Storage s = m_game_state.getResourceManager().getStorage();
9:           s.setSoundEnabled(!s.getSoundEnabled());                      // Sound
10:          buildMenu(); break;
11:   }
12:  } catch(Exception e) {...}
13: }
14: private void buildMenu() {
15:   Storage s = m_game_state.getResourceManager().getStorage();
16:   m_menu[0] = Str.opt_menu_play;
17:   m_menu[1] = Str.opt_menu_instructions;
18:   m_menu[2] = Str.opt_menu_highscores;
19:   if (s.getSoundEnabled()) m_menu[3] = Str.opt_menu_sound_on;           // Sound
20:   else m_menu[3] = Str.opt_menu_sound_off;                              // Sound
21:   if (s.getDifficultyLevel() == 0) m_menu[4] = Str.opt_menu_level_easy;
22:   else if (s.getDifficultyLevel() == 1) m_menu[4] = Str.opt_menu_level_normal;
23:   else m_menu[4] = Str.opt_menu_level_hard;
24:   m_menu[5] = Str.opt_menu_set_player_name;
25:   m_menu[6] = Str.opt_menu_exit;
26: }
27: ...
28:}
```

Figura 23: Métodos da classe `IntroGameHandler` que tratam o menu da versão original.

`ArrayList` recebe como parâmetros uma string que descreve a opção do menu e a posição de inserção na lista (linhas 6-7). Assim, quando a *feature* `Sound` estiver selecionada em um produto, a opção correspondente aparecerá na posição 3 do menu como apresentado na Figura 22.

3.3.2 Feature VisualFX

Discutiremos nesta seção refatorações e refinamentos da *feature* `VisualFX`. Dentre as *features* dessa categoria, foram necessárias refatorações significativas para permitir a modularização da *feature* `Cloud`. Essa *feature* é responsável pelo tratamento das nuvens no jogo. A Figura 27a apresenta uma tela do jogo que possui a *feature* `Cloud` e a Figura 27b apresenta uma tela do jogo com a *feature* `Cloud` desabilitada.

Uma refatoração requisitada pela *feature* `Cloud` ocorreu na classe `ResourceManager`. Essa classe é responsável por gerenciar os recursos do jogo, como, por exemplo, sons, imagens, fases etc. A Figura 28 apresenta o método `loadLevel` (linhas 3-16) da classe `ResourceManager`, que é responsável por carregar as imagens referentes a uma fase do jogo gravadas em um arquivo. A refatoração desse método ocorreu no `switch` (linhas 8-14), trocando-o por `ifs`. Além disso, resultou na extração do método `createResource` apresentado na Figura 29 (linhas 11-

```
1:public class IntroGameHandler extends BasicGameHandler {
2: ...
3: protected ArrayList menu_list;
4: private void handleCommands(byte cmd) {
5:  try {
6:    trataMenu(cmd);
7:  }
8:  catch(Exception e){...}
9: }
10: protected void trataMenu(byte cmd) throws Exception {       // extracted method
11:  strSelected = (String)menu_list.element(m_selected);
12:  if(this.strSelected.equals(Str.opt_menu_sound_off)
13:    || this.strSelected.equals(Str.opt_menu_sound_on)){
14:    Storage s = m_game_state.getResourceManager().getStorage();
15:    s.setSoundEnabled(!s.getSoundEnabled());
16:    buildMenu();
17:  }
18:  ...
19: }
20: protected void buildMenu() {
21:   Storage s = m_game_state.getResourceManager().getStorage();
22:   menu_list.removeAll();
23:   menu_list.add(Str.opt_menu_play);
24:   menu_list.add(Str.opt_menu_instructions);
25:   menu_list.add(Str.opt_menu_highscores);
26:   if (s.getDifficultyLevel() == 0) menu_list.add(Str.opt_menu_level_easy);
27:   else if (s.getDifficultyLevel() == 1) menu_list.add(Str.opt_menu_level_normal);
28:   else menu_list.add(Str.opt_menu_level_hard);
29:   menu_list.add(Str.opt_menu_set_player_name);
30:   menu_list.add(Str.opt_menu_exit);
31: }
32: ...
33:}
```

Figura 24: Classe`IntroGameHandler` refatorada.

```
 1: refines class Bomb {
 2:   void enterWater() {
 3:     Super.enterWater();                        // proceeds with refined method
 4:     m_game_state.getResourceManager().sound(...);
 5:   }
 6:   void setBombState() {
 7:     Super.setBombState();                      // proceeds with refined method
 8:     m_game_state.getResourceManager().sound(...);
 9:  }
 10:}
```

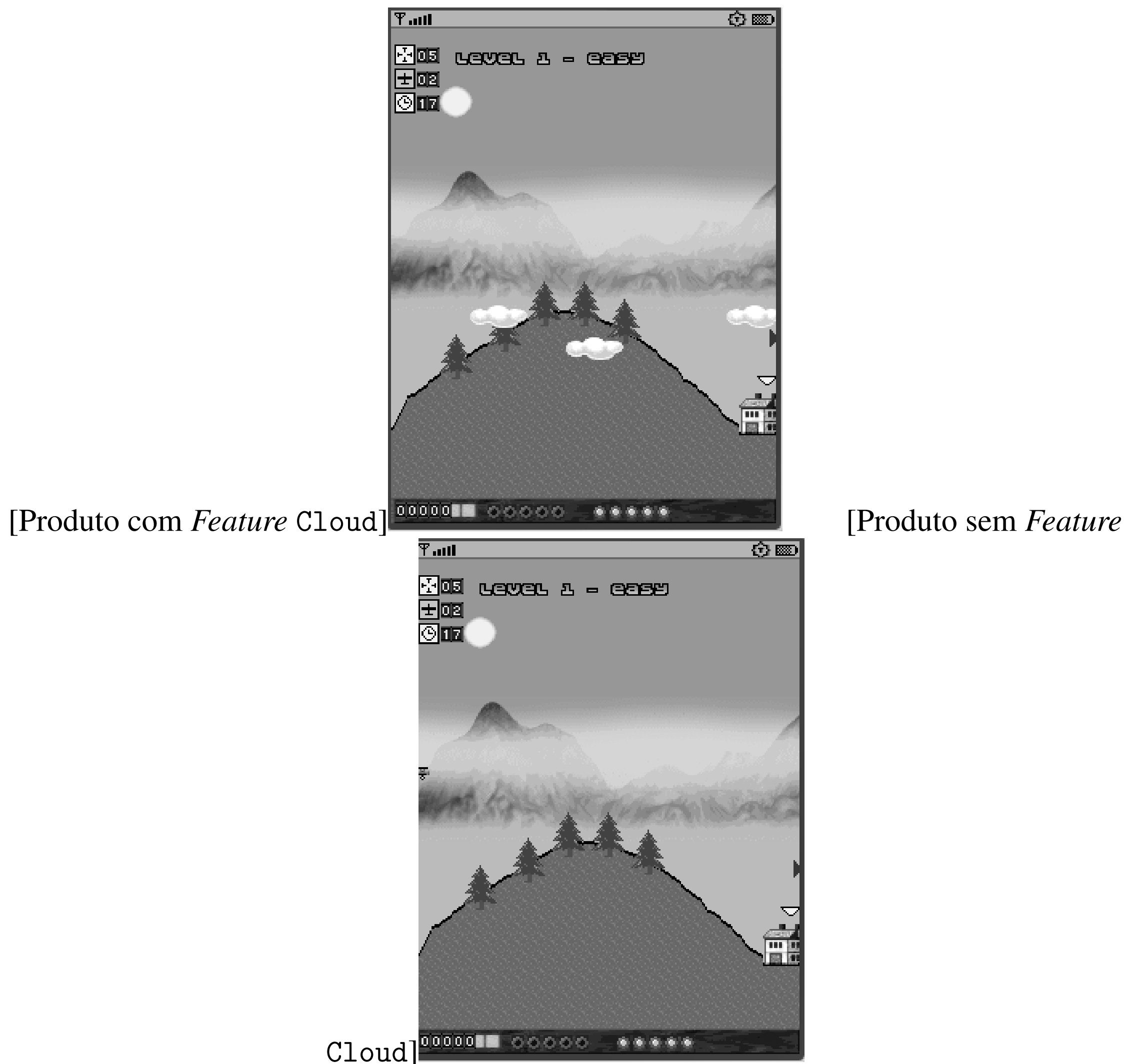
Figura 25: Refinamento que introduz a *feature* `Sound` na classe `Bomb`

16), o qual novamente utiliza-se de `ifs` para as comparações necessárias. Essa refatoração foi realizada para que as nuvens somente sejam tratadas quando a *feature* `Cloud` estiver habilitada.

Os refinamentos das *features* identificadas no código apresentado na Figura 20 serão apresentados a seguir. As Figuras 30, 31 e 32 apresentam respectivamente os refinamentos responsáveis pelas *feature* `Splash`, `ExplosionBlast` e `DamageTerrain`. O código da Figura 30 contém um refinamento que introduz os efeitos de respingos quando a bomba cai na água, isto

```
1:public refines class IntroGameHandler {
2: protected void buildMenu() {
3:   int posMenu=3;
4:   Super().buildMenu();
5:   Storage s = m_game_state.getResourceManager().getStorage();
6:   if (s.getSoundEnabled()) this.menu_list.add(Str.opt_menu_sound_on,posMenu);
7:   else this.menu_list.add(Str.opt_menu_sound_off,posMenu);
8: }
9: protected void trataMenu(byte cmd) throws Exception {
10:  Super().trataMenu(cmd);
11:  if(this.strSelected.equals(Str.opt_menu_sound_off)
12:     || this.strSelected.equals(Str.opt_menu_sound_on)) {
13:     Storage s = m_game_state.getResourceManager().getStorage();
14:     s.setSoundEnabled(...);
15:     buildMenu();
16:   }
17: }
18:}
```

Figura 26: Refinamento que introduz a opção de som no menu



[Produto com *Feature* `Cloud`] [Produto sem *Feature* `Cloud`]

Figura 27: Exemplos de tela da *feature* `Cloud` no jogo `Bomber`.

```
1:public class ResourceManager {
2: ...
3: public void loadLevel(GameState gs, int index) throws IOException {
4:   ...
5:   while(true) {
6:    ...
7:    byte type = dis.readByte();
8:    switch(type) {
9:      ...
10:      case 3: {                              // Cloud
11:        vis = gs.createCloud(...) break;
12:      }
13:    }
14:  }
15: }
16:}
```

Figura 28: Classe `ResourceManager` da versão original

```
1:public class ResourceManager {
2: ...
2: public void loadLevel(GameState gs, int index) throws IOException {
4:   ...
5:   while(true) {
6:    ...
7:    byte type = dis.readByte();
8:    createResource(type,...);
9:   }
10: }
11: protected void createResource(byte index,...) {
12:  Super().createResource(dis,gs,rgv,index);
13:  if(index==3)                                    // Cloud
14:   rgv.vis = gs.createCloud(...);
15:   ...
16:  }
17:}
```

Figura 29: Refatoração do método `loadLevel` da classe `ResourceManager`

é, quando é executado o método `enterWater` da classe `Bomb`. O código da Figura 31 contém um refinamento que introduz o efeito de explosão no terreno quando a bomba atinge o solo, isto é, quando executa-se o método `setBombState` da classe `Bomb`. Por fim, o código da Figura 32 contém um refinamento que introduz o efeito de dano no terreno quando a bomba também atinge o solo.

```
1:public refines class Bomb {
2: void enterWater() {
3:   super.enterWater();
4:   m_game_state.createSplash(...);
5: }
6:}
```

Figura 30: Refinamento que introduz a *feature* `Splash` na classe `Bomb`

A seguir será apresentado o refinamento da *feature* `Cloud`. Essa *feature* possui a classe `Cloud` que manipula as nuvens como, por exemplo, implementa a movimentação das nuvens

```
1:public refines class Bomb {
2: protected void setBombState() {
3:   super.setBombState();
4:   m_game_state.createBlast(...);
5: }
6:}
```

Figura 31: Refinamento que introduz a *feature* `ExplosionBlast` na classe `Bomb`

```
1:public refines class Bomb {
2: protected void setBombState() {
3:  super.setBombState();
4:  m_game_state.getTerrain().crater(...);
5: }
6:}
```

Figura 32: Refinamento que introduz a *feature* `DamageTerrain` na classe `Bomb`

no jogo. Essa classe foi totalmente extraída da `base`. A *feature* `Cloud` também refina outras classes da base. Por exemplo, ela introduz o método `createCloud` à classe `Game` (Figura 33). Esse método é responsável por criar os objetos de nuvens no jogo. A classe `Game` possui a interface `GameState` que também foi refinada, acrescentando o método `createCloud`. Esse refinamento é apresentado na Figura 34.

```
1:public refines class Game {
2: public Cloud createCloud(byte type, int x, int y, int vx) {
3:   Cloud ret_val = new Cloud(...);
4:   Super().addVisual(ret_val);
5:   return ret_val;
6: }
7:}
```

Figura 33: Refinamento que introduz a *feature* `Cloud` na classe `Game`

```
1:public refines interface GameState {
2:  Cloud createCloud(byte type, int x, int y,int vx);
3:}
```

Figura 34: Refinamento da interface `GameState` para o tratamento da *feature* `Cloud`

O refinamento que introduz o código referente ao tratamento da nuvem no método `createResource` é apresentado na Figura 35 (linhas 10-14). Como foi realizado nas refatorações anteriores, trocou-se o `switch` por um `if` (linhas 12-13). O método da superclasse também deve ser chamado para que sejam executados os outros tratamentos (linha 11). O método `getCloud` (linhas 2-9) é responsável por retornar o objeto que possui a imagem de uma nuvem. Esse método foi extraído da `base` e movido para o refinamento referente à *feature* `Cloud`.

A *feature* `Smoke` é uma *feature* que trata o efeito de fumaça no jogo. Ela também possui uma classe `Smoke` que foi totalmente extraída da `base`. Essa *feature* possui um refinamento interessante que ainda não foi apresentado, que atua sobre o construtor de uma classe. Para exemplificar essa forma de refinamento será utilizada a classe `FallingDebris`. A Figura 36

```
1:public refines class ResourceManager {
2:  Drawable getCloud(int index) {
3:    index = 5 + index;
4:    if(m_resources[index]==null) {
5:     DataInputStream dis = new DataInputStream(...);
6:     m_resources[index] = indexImage(index, dis, true);
7:    }
8:    return m_resources[index];
9: }
10: protected void createResource(byte index,...) {
11:   Super().createResource(index,...);
12:   if(index==3)                                      // Cloud
13:     vis = gs.createCloud(...);                      // Cloud
14: }
15:}
```

Figura 35: Refinamento da *feature* Cloud na classe ResourceManager

apresenta a classe FallingDebris original. Essa classe trata o efeito de ruínas quando um objeto é destruído por uma bomba. Entrelaçado a essa classe existe código relativo à *feature* Smoke (linhas 3-6,8-9). Nota-se que a construtora possui código referente a essa *feature* (linhas 8-9). Os códigos referentes a *feature* Smoke foram extraídos. Criou-se então um refinamento com o conteúdo somente referente à *feature* Smoke, conforme apresentado na Figura 37. O refinamento de construtores é uma exceção ao de métodos, pois se usa a palavra reservada refines (linha 5) para identificar que aquele construtor será refinado.

```
1:public class FallingDebris extends GameObject {
2:   ...
3:   protected Smoke m_smoke;                        //Smoke
4:   protected int  m_smoke_timer;                   //Smoke
5:   public static final int SMOKE_RATE = 100;       //Smoke
6:   public FallingDebris(...) {
7:   ...
8:   m_smoke = new Smoke(...);                       //Smoke
9:   m_game_state.addVisual(m_smoke);                //Smoke
10: }
11: ...
12:}
```

Figura 36: Refinamento do construtor *feature* Smoke na classe FallingDebris

```
1:public refines class FallingDebris {
2:  protected Smoke  m_smoke;
3:  protected int  m_smoke_timer;
4:  public static final int SMOKE_RATE = 100;
5:  refines FallingDebris(...) {
6:    m_smoke = new Smoke(...);
7:    Super().m_game_state.addVisual(m_smoke);
8: }
9: ...
10:}
```

Figura 37: Refinamento da classe FallingDebris.

A tabela 5 apresenta os tipos e as quantidades de refatorações realizadas. A seguir explica-

se cada tipo de refatoração realizado:

- Movimentações: indica a quantidade de métodos ou atributos que foram movidos da base para um refinamento;
- Extração de métodos: indica a quantidade de métodos que foram criados para a realização de um refinamento. Como exemplo, o método `setBombState` da Figura 21;
- Modificações em métodos: indicam mudanças no corpo de métodos, como, por exemplo, inserção de novos comandos, inserção ou retirada de um parâmetro etc;
- Modificações de visibilidades: indica mudança de visibilidade de `private` para `protected`. Essa forma de mudança é útil para que métodos ou atributos possam ser acessado pelas classes filhas;
- Movimentação de classes para *features*: indica movimentação de classes que realizavam tarefas específicas de uma *feature* para os respectivos refinamentos.

Tipo	**Quantidade**
Movimentações	37
Extração de métodos	25
Modificações em métodos	23
Modificações de visibilidades	3
Movimentação de classes para *features*	6
Total	94

Tabela 5: Refatorações realizadas.

3.4 Comentários Finais

Neste capítulo, descreveu-se uma experiência de extração de uma LPS na área de jogos para celulares utilizando FOP/AHEAD. Essa experiência foi apresentada com um estudo de caso, envolvendo o jogo Bomber. Mostrou-se a metodologia utilizada no estudo de caso, as *features* propostas, as extrações realizadas e os problemas encontrados. Mostrou-se também a arquitetura do jogo Bomber juntamente com seu diagrama de classes, o diagrama de *features* referente às *features* extraídas e os tipos de refatorações necessárias para permitir a extração de *features*.

Concluiu-se que o ambiente FOP/AHEAD foi capaz de modularizar as *features* identificadas no estudo de caso, utilizando uma linguagem simples, bem próxima a Java.

4 EVOLUÇÃO DE UMA LPS UTILIZANDO PROGRAMAÇÃO ORIENTADA POR FEATURES

Neste capítulo, descreve-se uma experiência de evolução da LPS descrita no capítulo anterior. Serão apresentadas as novas *features* criadas e o novo diagrama de *features*. As novas *features* criadas foram:

- `Spanish` : Tradução do jogo para a língua espanhola, sendo uma *feature* alternativa e exclusiva;
- `SpeedControl`: Permite que o jogador aumente ou diminua a velocidade do avião, sendo uma *feature* opcional;
- `MovingFX`: Novas animações no jogo, incluindo `MoveZepellin` (movimenta o *zeppelin*) e `MoveTank` (movimenta o tanque de um lado para o outro), sendo *features* opcionais;
- `ServiceRank`: Serviço de armazenamento *on-line* de *rankings* de pontuação do jogo. Utiliza *sockets* para comunicação com um servidor, sendo uma *feature* opcional;
- `SoundExplosion`: Essa *feature* é interligada com as *features* `Sound` e `Explosion`. Ela é responsável pelo controle do som quando ocorre uma explosão. Essa *feature* somente pode ser habilitada quando as outras duas *features* estiverem habilitadas ou se habilitada as outras duas também devem ser.

As *features* discutidas anteriormente são mostradas no DF apresentado na Figura 38. Essas *features* foram escolhidas por serem características interessantes e/ou comuns em jogos de celulares. Por exemplo, a *feature* `ServiceRank` pode utilizar *bluetooth* ou *wi-fi* para a comunicação com um servidor de rankings.

4.1 Evolução da LPS

Inicialmente descreve-se nesta seção algumas refatorações realizadas no código para que fosse possível incluir as novas *features* propostas. Mais especificamente, é apresentada a refatoração responsável por permitir a modularização da *feature* `SpeedControl`. Essa refatoração constitui na inserção de novos comandos do avião. A Figura 39 apresenta a classe que trata as missões do jogo.

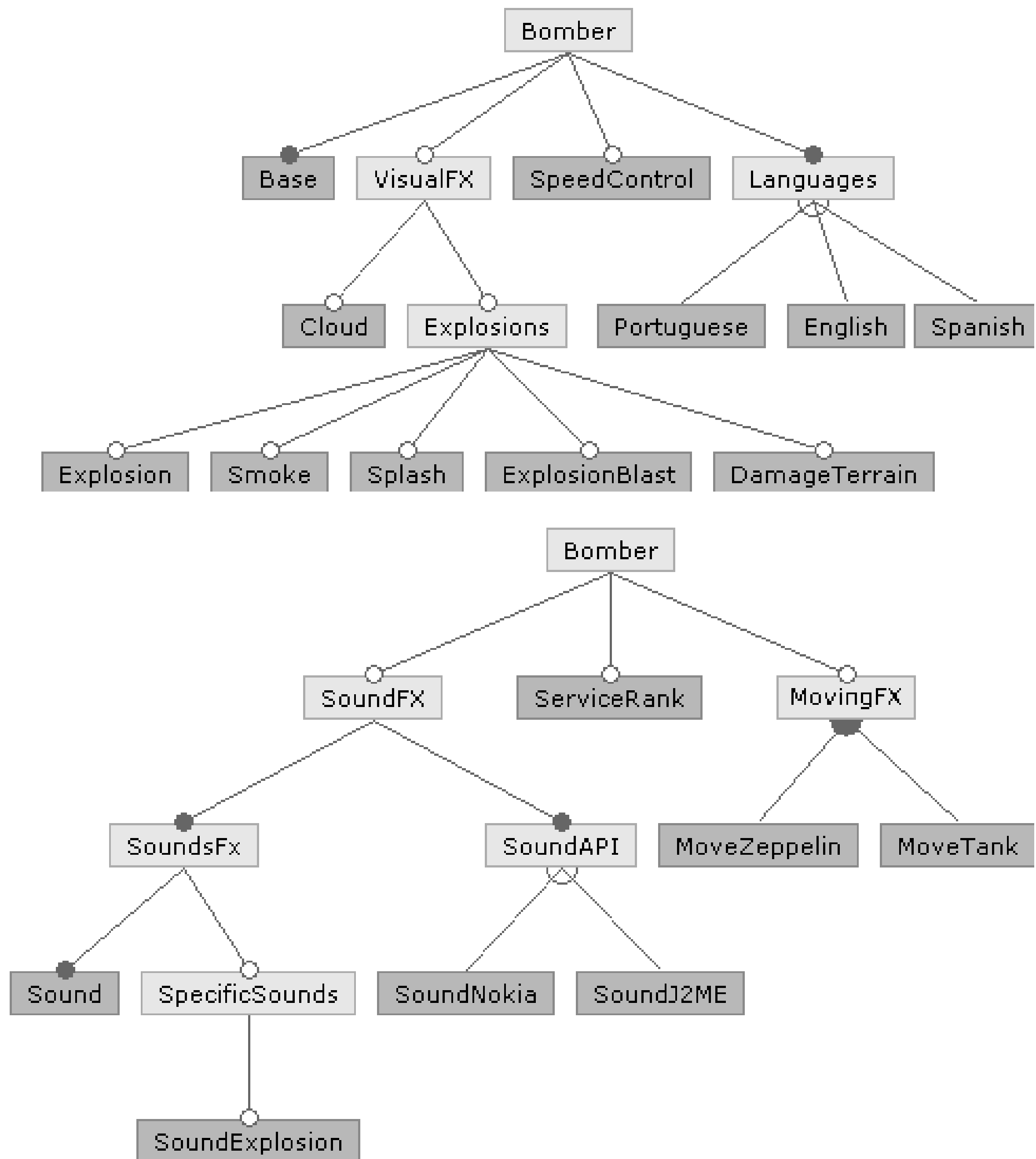


Figura 38: Novo Diagrama de Feature.

```
1:public class MissionGameHandler extends BasicGameHandler {
2: ...
3: public void event(byte type, boolean pressed) {
4:   if (...) {
5:     ...
6:   }
7:  else if (m_state == PLAYING) {
8:    switch(type) {
9:       case UP: m_control.setTurnUp(pressed); break;
10:      case DOWN: m_control.setTurnDown(pressed); break;
11:      case FIRE_A: m_control.setBomb(pressed); ... break;
12:      case FIRE_B: m_control.setFire(pressed); ...; break;
13:    }
14:  }
15: }
16:}
```

Figura 39: Classe `MissionGameHandler`

Essa classe possui um método que trata os controles do avião. Quando o jogador pressiona alguma tecla do celular, ela é capturada e, dependendo de seu valor, algumas operações são executadas (linhas 8-14). Os controles que são tratados nessa classe são: seta para cima (UP), move o avião para cima, seta para baixo (DOWN) , move o avião para baixo e os botões de tiro (A e B), disparam uma bomba ou uma bala. Para que seja possível inserir tratamentos de novos controles foi necessário extrair um novo método (`choiceControls`) que possui o tratamento dos controles, como apresentado na Figura 40 (linhas 12-19).

```
1:public class MissionGameHandler extends BasicGameHandler {
2: ...
3: public void event(byte type, boolean pressed) {
4:   if (...) {
5:     ...
6:   }
7:  else if (m_state == PLAYING) {
9:     choiceControls(type,pressed);                     // extracted method
10: }
11:}
12:void choiceControls(byte type,boolean pressed) {       // extracted method
13:  switch(type) {
14:    case UP: m_control.setTurnUp(pressed); break;
15:    case DOWN: m_control.setTurnDown(pressed); break;
16:    case FIRE_A: m_control.setBomb(pressed); ... break;
17:    case FIRE_B: m_control.setFire(pressed);... break;
18:  }
19: }
20:}
```

Figura 40: Classe `MissionGameHandler` refatorada

Após a refatoração foi criado um refinamento que introduz o tratamento dos novos comandos, conforme apresentado na Figura 41. Esse refinamento somente possui a introdução dos novos controles no método `choiceControls` (linhas 2-10). Por ter sido realizada uma refatoração em um `switch`, optou-se por manter o `switch` e não trocá-lo por um `if` como

ocorreu nas refatorações anteriores. Isso foi feito para demonstrar que também é possível introduzir novas escolhas em um `switch`. Neste caso, o método a ser refinado possui somente o comando `switch`. Então, para a inserção de novas escolhas no refinamento,utiliza-se um novo `switch`. Porém, são inseridas somente as novas escolhas e como escolha padrão, ou seja, caso nenhuma escolha tenha sido capturada, chama-se o método da base (linhas 4-6).

```
1:public refines class MissionGameHandler {
2: void choiceControls(byte type,boolean pressed){
3:   switch(type) {
4:     case LEFT: m_control.setTurnLeft(); break;
5:     case RIGHT: m_control.setTurnRight(); break;
6:     default: super.choiceControls(type,pressed); break;
7:   }
8: }
9:}
```

Figura 41: Refinamento da *feature* `SpeedControl`

Nas próximas sub-seções serão apresentados os refinamentos responsáveis pela inserção das novas *features*.

4.1.1 *Feature* SpeedControl

O refinamento apresentado na Figura 42 insere o tratamento de velocidade no avião representado pela classe `Plane`. Esse refinamento introduz um atributo que indica qual a aceleração do avião (linha 2) e o atributo de velocidade do avião (linha 3). Esse refinamento também refina o construtor da classe para que seja possível a inicialização do atributo (linhas 4-7) e introduz métodos que irão controlar a velocidade do avião como por exemplo, aumentando e diminuindo a sua velocidade (linhas 8-18). Esse métodos são chamados pelas novas opções de tecla (seta para esquerda e seta para direita), as quais estão todas no refinamento mostrado na Figura 41. O método `setTurnLeft` (linhas 8-14) Figura 42 é responsável pela diminuição da velocidade do avião sendo que a velocidade mínima é 10 (a imagem se movimenta a cada 10 pixels). A existência desse valor mínimo evita que o avião pare ou que tenha velocidade negativa. O método `setTurnRight` (linhas 15-18) é responsável pelo aumento de velocidade do avião. Toda vez que for executado, ele incrementa em 10 a velocidade do avião.

4.1.2 *Feature* MoveTank

A *feature* `MoveTank` somente possui um refinamento, apresentado na Figura 43. Esse refinamento tem como objetivo inserir na classe base `Tank` atributos necessários para movimentação do tanque durante o jogo (linhas 2-4). Tais atributos armazenam o sentido em que o tanque está se movendo, direita ou esquerda (linha 2), sua movimentação e quanto movimentar (linha 3) e a distância que movimentar (linha 4). Esse refinamento também adiciona comandos responsáveis

```
1:public refines  class Plane {
2:  private final int incrementAcel = 10;
3:  private final int min_engine_power;
4:  refines Plane(...) {
6:    min_engine_power = engine_power;
7:  }
8:  public void setTurnLeft() {
10:   if(m_engine_power>=10)
11:     m_engine_power-=incrementAcel;
12:   else
13:     m_engine_power=min_engine_power;
14: }
15: public void setTurnRight() {
16:   m_engine_power+=incrementAcel;
17: }
18:}
```

Figura 42: Refinamento da classe `Plane`

pela movimentação do tanque no método `Draw` (linhas 5-20). Tais comandos verificam se o tanque está se movimentando para esquerda ou direita (linhas 6-13) e se o tanque atingiu o máximo de movimentações (linhas 14-17). Os comandos introduzidos atualizam ainda a nova posição do tanque (linha 18) e chamam o método da base (linha 19).

```
1: public refines class Tank {
2:  private boolean right=true;
3:  private int mov,dist=50;
4:  private int quantDist=0;
5:  public void draw(Graphics g, int x, int y) {
6:    if(!right) {
7:      mov=-dist;
8:      quantDist--;
9:    }
10:   else {
11:     mov=dist;
12:     quantDist++;
13:   }
14:   if(quantDist>=50)
15:     right=false;
16:   if(quantDist<=0)
17:     right=true;
18:   m_bounds.getPos().x+=mov;
19:   super.draw(g,x,y);
20: }
21:}
```

Figura 43: Refinamento da classe `Tank` na *feature* `MoveTank`

4.1.3 *Feature* MoveZeppelin

A *feature* `MoveZeppelin` também possui somente um refinamento, o qual tem como objetivo inserir tratamento de movimento do zeppelin. Esse refinamento é apresentado na Figura 44.

Por meio desse refinamento são inseridos atributos necessários para o tratamento de movimentação do zeppelin (linhas 2-4). Mais especificamente, esses atributos armazenam o sentido no qual o zeppelin está se movendo, esquerda ou direita (linha 2), qual a posição do zeppelin (linha 3) e velocidade na qual ele se movimentará (linha 4). O construtor dessa classe foi refinado para que seja possível a inicialização desses atributos (linhas 5-9). O método `Draw` também foi refinado, para que seja tratada a movimentação do zeppelin (linhas 9-20). Caso o zeppelin esteja no início da tela à esquerda ele deve movimentar para direita (linha 10). Caso esteja na posição inicial, o zeppelin deve se movimentar para esquerda (linha 13). Caso o zeppelin tenha que ser movimentado para esquerda, ele receberá sua nova posição (linhas 16-17), senão ele recebe a nova posição para a direita (linhas 18-19).Por fim, chama-se o método da base para que seja desenhado o zeppelin em sua nova posição.

```
1: public refines class Zeppelin {
2:  private boolean esq=false;
3:  private int xini,yini;
4:  private int vel = 100;
5:  refines Zeppelin(...) {
6:    xini=Common.toFP(x);
7:    yini=Common.toFP(y);
8:  }
9:  public void draw(Graphics g, int x, int y) {
10:   if ( m_bounds.getPos().x<=0) {
11:     esq=false;
12:   }
13:   if (m_bounds.getPos().x>=xini) {
14:     esq=true;
15:   }
16:   if (esq)
17:     m_bounds.getPos().x-=vel;
18:   else
19:     m_bounds.getPos().x+=vel;
20:   Super().draw(g,x,y);
21:  }
22: }
```

Figura 44: Refinamento da classe `Zeppelin` na *feature* `MoveZeppelin`

4.1.4 *Feature* ServiceRank

Essa *feature* é responsável pelo armazenamento *on-line* de *rankings* de pontuação do jogo. Ela armazena e recupera em um servidor de *rankings* a pontuação e os dados dos jogadores. Ela possui duas novas classes que fazem o tratamento de envio de informações, nome e pontuação, por meio de *sockets*. Além disso, ela inclui um refinamento que atua sobre a classe `Storage`, que é a classe responsável pelo armazenamento e recuperação dos dados de *ranking* dos jogadores. Esse refinamento é apresentado na Figura 45. Esse refinamento declara um novo atributo responsável pelo tratamento das conexões com o servidor de *rankings* (linha 2). Um novo

método `createServer` (linha 3-6) é implementado para criar uma conexão com o servidor de *rankings*. Os métodos que sofrem refinamento são `save` e `readRecordStore`. No método `save` – responsável por salvar os *rankings* dos jogadores – são inseridos os códigos responsáveis por salvar o *ranking* em um servidor (linhas 7-11). O método `readRecordStore` é responsável por recuperar o *ranking* e em seu refinamento são inseridos comandos responsáveis por recuperar os dados de *ranking* em um servidor (linhas 12-19).

```
1: public refines class Storage {
2:  private ServerRank sr = null;
3:  private  void createServer() {
4:    if(sr==null)
5:      sr = new ServerRank();
6:  }
7:  public void save() throws Exception {
8:    createServer();
9:    ...
10:   sr.grava(dados);
11:  }
12:  protected void readRecordStore() throws Exception {
13:    ...
14:    dados=sr.dados();
15:    if(dados==null) {
16:      Super().readRecordStore();
17:      return;
18:    }
19:  }
20:}
```

Figura 45: Refinamento da classe `Storage` na *feature* `ServiceRank`

A seguir são apresentadas duas tabelas com análises referentes às novas *features*. A Tabela 6 apresenta medições com quantidades de classes refinadas e novas classes acrescentadas por cada nova *feature*. Pode ser visto que a *feature* `Spanish` possui somente uma classe refinada e nenhuma classe acrescentada; já a *feature* `ServiceRank` possui uma classe refinada e duas novas classes acrescentadas. Essa *feature* foi a única que demandou o acréscimo de novas classes.

Feature	Classes Refinadas	Novas Classes
Spanish	1	0
SpeedControl	2	0
ServiceRank	1	2
MoveZeppelin	1	0
MoveTank	1	0
SoundExplosion	1	0

Tabela 6: Classes refinadas pelas novas *features*.

A Tabela 7 apresenta o número de linhas de código (LOC) de cada nova *feature*, ou seja, o total de linhas que serão acrescidas no jogo com a escolha dessas novas *features*.

Feature	LOC
Spanish	50
SpeedControl	32
ServiceRank	163
MoveZeppelin	24
MoveTank	20
SoundExplosion	7

Tabela 7: LOC das *features*.

4.2 Comentários Finais

Neste capítulo, descreveu-se uma experiência de evolução da LPS descrita no Capítulo 3. Foram apresentadas as novas *features* criadas e o diagrama de *features* com o acréscimo dessas *features*. Mostrou-se algumas refatorações que foram realizadas no código para que fosse possível inclui-las. Foram apresentados também os refinamentos dessas *features* em FOP/AHEAD e como esses refinamentos modificam as classes do jogo. Mostrou-se o número de linhas de cada nova *feature* e o número de classes novas que cada *feature* acrescenta. Como observação final, pode-se afirmar que FOP/AHEAD também foi capaz de modularizar as novas *features* propostas para a LPS.

5 AVALIAÇÃO

Neste capítulo, realiza-se uma avaliação qualitativa da LPS proposta (Seção 5.1) e uma comparação com tecnologias alternativas de implementação, incluindo orientação por objetos (Seção 5.2) , orientação por aspectos (Seção 5.3) e *mixins* (Seção 5.4).

5.1 Avaliação Qualitativa

Do ponto de vista qualitativo, a solução proposta pode ser analisada de acordo com os seguintes fatores:

- Configurabilidade: essa característica define o quanto é flexível configurar, customizar e adaptar os componentes de um sistema (IEEE. . . , 1990). Como esperado em uma LPS, considera-se que a principal vantagem da solução implementada nesse trabalho é a facilidade com que se pode derivar diversos produtos. Na LPS extraída é possível gerar 384 diferentes versões do jogo Bomber como explicado pela seguinte fórmula:

  $$EX = \underbrace{2^6}_{\text{VisualFX}} * \underbrace{2}_{\text{Languages}} * \underbrace{3}_{\text{SoundFX}} = 384$$

  Na versão evoluída, é possível gerar 4.096 diferentes versões como explicado pela seguinte fórmula:

  $$EV = \underbrace{2^6}_{\text{VisualFX}} * \underbrace{2^2}_{\text{SoundFX}} * \underbrace{2}_{\text{SpeedControl}} * \underbrace{2}_{\text{ServiceRank}} * \underbrace{2^2}_{\text{MoveFX}} = 4.096$$

  Para gerar tais versões, basta definir corretamente os arquivos de *features* ao se chamar o combinador do ambiente AHEAD.

- Compilação em separado e verificação estática de tipos: compilação em separado é a capacidade de compilar cada módulo de um sistema de forma separada. Verificação estática de tipos é capacidade de o compilador realizar a verificação dos tipos de um programa. Na solução implementada, *features* são definidas em módulos independentes, os quais podem ser compilados de forma separada, sem perder os recursos de verificação estática de tipos de Java. Essa vantagem é importante principalmente em grandes sistemas de software.

Ela representa também um avanço em relação ao uso de diretivas de compilação condicional para implementação de variabilidades. Como já afirmado no Capítulo 2, com o uso de diretivas de compilação, quando se modifica uma parte de um programa é necessário que todo ele seja novamente compilado.

- Reusabilidade: é o grau de facilidade com que partes de um sistema podem ser reusados em outros sistemas ou LPS (IEEE..., 1990). Está relacionado à alta coesão e baixo acoplamento com outros módulos. Em AHEAD, um refinamento acrescenta comportamento a uma classe específica do programa base. Para isso, ele faz uso do nome dessa classe e também referencia seus membros. Por exemplo, para refinar a classe `Bomb`, o refinamento mostrado na Figura 2.9 referencia dois de seus métodos (`enterWater` e `setBombState`) e um campo (`m_game_state`). Ou seja, existe um forte acoplamento entre os refinamentos implementados nesta dissertação e as entidades por eles refinadas. Esse acoplamento prejudica o reúso desses refinamentos em outros sistemas e/ou LPS.

  Por outro lado, o que mais afeta o grau de coesão de uma classe é a presença de código entrelaçado, responsável por outros interesses (além do interesse principal da classe). Por exemplo, no estudo de caso realizado, a classe `Bomb` continha código referente à funcionalidade de som, o qual ficava entrelaçado com o código específico para tratamento da bomba. Ou seja, a coesão da classe era prejudicada. Então, com a extração da *feature* `Sound`, a classe `Bomb` ficou mais coesa, já que o código entrelaçado foi removido. Assim, aumentou-se também a possibilidade de reúso dessa classe em outros contextos, nos quais a funcionalidade de som não é necessária.

- Desenvolvimento em paralelo: designa a capacidade de implementar e evoluir os módulos de um sistema de forma independente (PARNAS, 1972). Em outras palavras, supondo que módulos possuem interfaces bem definidas eles podem ser implementados ao mesmo tempo, possivelmente por programadores distintos. Tanto na LPS estendida, como na evolução da LPS *features* adicionam comportamento extra em métodos do programa base. Assim, a princípio, programação orientada por *features* acrescenta uma nova dimensão ao problema de fragilidade de classes base, bastante conhecido em orientação por objetos (MIKHAJLOV; SEKERINSKI, 1998). Normalmente, em OO afirma-se que classes bases são frágeis porque mudanças em seus métodos podem afetar o correto funcionamento de classes derivadas. Por exemplo, suponha a classe `List` da Figura 46. Suponha agora uma classe `ListWithSize`, que estende a classe `List` com um atributo que irá armazenar o número de elementos da lista, conforme mostrado na Figura 47. Por fim, suponha uma mudança na classe `List` para torná-la mais eficiente, como apresentado na Figura 48. A melhoria implementada nessa classe ocorreu no método `addAll` (linhas 3-5). Agora esse método não chama mais o método `add` para inserir elementos na lista. Aparentemente essa nova versão parece ser compatível com a anterior. Porém, essa

mudança altera o comportamento da classe `ListWithSize`, pois o número de elementos da lista não será mais incrementado ao se chamar o método `addAll`.

```
1: public class List {
2:   ArrayList b;
3:   public List() {
4:     b = new ArrayList();
5:   }
6:   public void add(char x) {
7:     b.add(x);
8:   }
9:   public void addAll(ArrayList bs) {
10:    Iterator it = bs.iterator();
11:    while(it.hasNext()){
12:      char c=it.next().toString().charAt(0);
13:      add(c);
14:    }
15:  }
16:}
```

Figura 46: Classe `List`.

```
1: public class ListWithSize extends List {
2:  int size;
3:  public ListWithSize() {
4:    size=0;
5:  }
6:  public void add(char x) {
7:    size=size+1;
8:    super.add(x);
9:  }
10: public int size() {
11:   return size;
12: }
13:}
```

Figura 47: Classe `ListWithSize`.

```
1: public class List {
2:   ...
3:   public void addAll(ArrayList bs) {
4:     b.addAll(bs);
5:   }
6: }
```

Figura 48: Evolução da classe `List`.

No caso de FOP, mudanças em classes do programa base podem afetar também o comportamento de *features*. O principal motivo é que os refinamentos necessitam do método refinado para obter sua completa execução.

5.2 Comparação com Orientação por Objetos

Em certo sentido, um refinamento em AHEAD é semelhante a extensão de uma classe por meio de herança. Por exemplo, herança permite redefinir e introduzir novos métodos em classes como apresentado na Figura 49. Nessa figura, a classe `A` possui dois métodos: `f1` e `f2`. A classe `B` herda de `A`, redefinindo o método `f1` e inserindo um novo método `f3`.

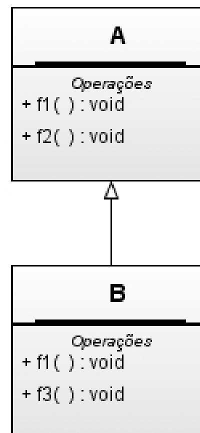


Figura 49: Exemplo de herança.

Uma das principais diferenças entre herança e AHEAD é o fato de uma subclasse ter um nome diferente da sua superclasse; já um refinamento não muda o nome da classe base. Assim, múltiplas combinações de *features* podem ser aplicadas sobre uma mesma classe base, em tempo de instanciação da LPS. Em OO, isso requer a implementação de múltiplas subclasses, uma para cada possível combinação de *features*, como apresentado na Figura 50. Nessa figura, o método `f1` representa uma *feature* obrigatória e `f2` e `f3` representam *features* opcionais. Se um sistema somente utilizar a *feature* `f1` será necessário instanciar objetos de `A`. Caso queira também utilizar a *feature* representada por `f2`, será necessário instanciar um objeto de `A1`. Caso queira utilizar apenas a *feature* `f3`, será necessário instanciar um objeto de `A2`. Por fim, se quiser utilizar as *features* `f2` e `f3`, será necessário instanciar um objeto de `A3`.

Em resumo, se em um sistema existem n métodos relacionados que representam *features* opcionais, serão necessárias $C_1^n + C_2^n + C_3^n + ...C_n^n$ subclasses para representar todas as combinações de *features*, onde C_m^n significa o número de combinações de n elementos, m a m.

Exemplo de herança: Descreve-se a seguir um exemplo de implementação de uma *feature* no jogo Bomber utilizando herança. Nesse exemplo, a classe `A` da Figura 50 será representada pela classe `Bomb`, a classe `A1` será representada pela classe `BombDamageTerrain`, a classe `A2` será representada pela classe `BombSound` e a classe `A3` será representada pela classe

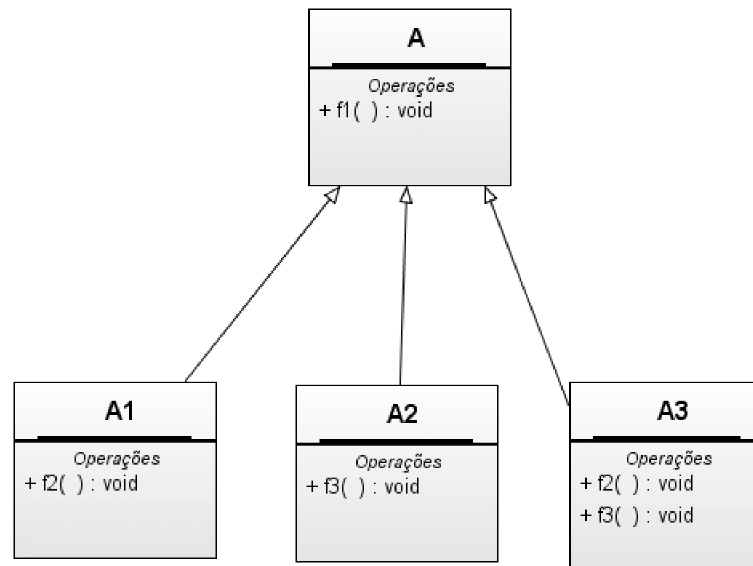


Figura 50: *Feature* obrigatória (`f1`) e opcionais (`f2` e `f3`) implementadas por meio de herança.

`BombSoundDamageTerrain`. O método redefinido é o `setBombState` (linhas 3-5, Figura 51), o qual define o estado da bomba como destruído. A Figura 51 apresenta a classe `Bomb` da base, a qual possui somente comandos obrigatórios. A Figura 52 apresenta a subclasse `BombDamageTerrain` que inclui a *feature* `DamageTerrain`. Essa classe redefine o método `setBombState` inserindo código requerido pela *feature* `DamageTerrain` (linha 4).

```
1: class Bomb extends FallingObject {
2:   ...
3:   protected void setBombState() {
4:     destroy();
5:   }
6: }
```

Figura 51: Classe `Bomb`.

```
1: class BombDamageTerrain extends  Bomb {
2:   protected void setBombState() {
3:     super.setBombState();
4:     m_game_state.getTerrain().crater(...);
5:   }
6: }
```

Figura 52: Classe `BombDamageTerrain`.

Um outro exemplo seria a inclusão da *feature* `Sound` na classe `Bomb`. Novamente torna-se necessário criar outra classe herdando de `Bomb`, como apresentado na Figura 53. Deve-se ainda incluir nessa nova classe o código referente à *feature* `Sound` (linha 4). Por fim, pode-se introduzir tanto a *feature* `Sound` quanto a *feature* `DamageTerrain`. Também nesse caso será necessário criar uma outra subclasse, como apresentado na Figura 54.

```
1: class BombSound extends Bomb {
2:   protected void setBombState() {
3:     super.setBombState();
4:     m_game_state.getResourceManager().sound(...);
5:   }
6: }
```

Figura 53: Classe BombSound.

```
1: class BombSoundDamageTerrain extends Bomb {
2:   protected void setBombState() {
3:     super.setBombState();
4:     m_game_state.getTerrain().crater(...);
6:     m_game_state.getResourceManager().sound(...);
7:   }
8: }
```

Figura 54: Classe BombSoundDamageTerrain.

Padrão de Projeto Decorador: O padrão de projeto Decorador é útil para inserir novos comportamentos em classe existentes (GAMMA et al., 1997). Esse padrão adiciona novos comportamentos em objetos dinamicamente, fornecendo uma alternativa mais flexível que o emprego de herança. Com ele é possível integrar novas *features* sem a explosão de subclasses. Para exemplificar, utilizaremos uma interface Channel, que tem a responsabilidade de enviar e receber dados pela rede, e as seguintes classes que implementam essa interface: TCPChannel (que utiliza o protocolo TCP para envio de dados), BufferChannel (que disponibiliza um *buffer* de dados) e ZipChannel (que permite compactação e descompactação de dados). Se for necessário um *buffer* associado ao TCPhannel, a instânciação de um objeto deverá ser feita da seguinte forma:

```
Channel= new BufferChannel (new TCPChannel());
```

Caso o TCPChannel necessite de compactar/descompactar os dados enviados/recebidos, a instânciação de um objeto deverá ser feita da seguinte forma:

```
Channel= new ZipChannel (new TCPChannel());
```

Ou então se for necessário tanto o *buffer* e a compactação, a instânciação de um objeto deverá ocorrer como mostrado a seguir:

```
Channel= new BufferChannel (new ZipChannel (new TCPChannel()));
```

Porém, a combinação de *features* não é automática como no caso do AHEAD, isso é, baseada em um arquivo de configuração externo à aplicação. Além disso, novamente as classes possuem nomes diferentes. Se uma *feature* possui refinamentos em várias classes, em AHEAD somente é necessário identificar qual *feature* deve ser acrescentada que todas as suas classes serão refinadas. Utilizando os decoradores isso não ocorre de forma automática.

Configuração e seleção de *features*: Uma desvantagem de soluções para implementação de *features* baseadas em orientação por objetos é a necessidade de configurar o sistema para usar a subclasse correta. Essa configuração pode ser feita, por exemplo, por meio de arquivos de configuração ou de técnicas de injeção de dependência (CHIBA; ISHIKAWA, 2005; FOWLER,). Arquivos de configuração são arquivos que armazenam configurações das aplicações, como exemplificado a seguir:

```
1: f2=true
2: f3=true
```

Esse arquivo define que as *features* `f2` e `f3` deverão ser incorporadas à aplicação base. Para utilizá-lo pode-se, por exemplo, utilizar uma fábrica de objetos do tipo `A` que retorne variações das subclasses de `A` de acordo com as *features* selecionadas, como apresentado no exemplo a seguir:

```
1: class AFactory {
2:   public static A getInstance() {
3:     ...
4:     if(f2 && !f3)
5:       return new A1();
6:     if(f3 && !f2)
7:       return new A2();
8:     if(f2 && f3)
9:      return new A3();
10: }
11:}
```

Essa fábrica retornará os objetos de acordo com as seleções descritas no arquivo de configuração (linhas 4-9). A instânciação do objeto deverá ser feita da seguinte forma:

```
A a = AFactory.getInstance();
```

Veja, no entanto, que a utilização de uma fábrica e de um arquivo de configuração não dispensa a criação de múltiplas subclasses (`A1`,`A2`,`A3`).

Princípio Aberto/Fechado: esse princípio advoga que entidades de software (classes, módulos, funções etc) devem estar abertas para extensões e fechadas para modificações (MEYER, 1988). Em outras palavras, novas funcionalidades devem ser adicionadas por meio de herança, em subclasses. Por outro lado, uma classe deve estar fechada e pronta para uso, de acordo com

a sua implementação atual. Como afirmado nesta dissertação, AHEAD/FOP tem como objetivo modificar, estender e acrescentar funcionalidades extras a classes base. Ou seja, com AHEAD/-FOP, classes devem estar abertas para extensões não somente por meio de herança, mas também por meio de refinamentos.

5.3 Comparação com Orientação por Aspectos

Aspectos – conforme definidos em AspectJ (KICZALES et al., 2001)– constituem inequivocamente uma tecnologia alternativa para implementação de refinamentos tal como propostos em AHEAD. Na verdade, aspectos são mais poderosos que refinamentos. Pode-se, por exemplo, alterar a hierarquia de classes do sistema, interceptar a execução de leituras e escritas em campos de classes, interceptar a execução de tratadores de exceções etc. Por outro lado, em AHEAD pode-se apenas introduzir novos campos e métodos em classes e alterar o comportamento de métodos já existentes de forma semelhante ao que se consegue em AspectJ por meio de adendos do tipo `around`. Resumindo, a linha de produtos descrita nesse artigo poderia ser extraída usando-se aspectos tal como implementados em AspectJ. No entanto, a principal vantagem de AHEAD reside exatamente na simplicidade de sua linguagem para separação de interesses. Basicamente, em AHEAD não é necessário declarar conjuntos de junção, definir o tipo de adendos (`after`, `before` ou `around`), associar adendos a conjuntos de junção etc. Em vez disso, um refinamento em AHEAD tem acesso direto ao ambiente da classe refinada, o que simplifica a sua sintaxe e o seu entendimento.

Por outro lado, a principal desvantagem de AHEAD é a inexistência na linguagem de recursos de quantificação. Um refinamento sempre estende o comportamento de um método de uma classe do programa base. Já em AspectJ, recursos de quantificação permitem definir adendos que atuarão em múltiplos pontos de junção do programa base, o que é particularmente útil para implementação de requisitos transversais homogêneos. No entanto, diversas experiências de uso de AspectJ têm demonstrado que tais requisitos não são tão comuns, notadamente no caso de implementação de LPS (APEL; BATORY, 2006; LOPEZ-HERREJON; APEL, 2007).

Para exemplificar as diferenças mencionadas anteriormente, será utilizada como base a classe `Bomb` apresentada na Figura 51. O refinamento que introduz o som nessa classe é apresentado na Figura 55. A Figura 56 apresenta um aspecto em AspectJ que introduz os comandos de som na classe `Bomb`. Nesse aspecto, são declarados conjuntos de junção (linhas 2-5) e então são criados os adendos (linhas 6-11) que introduzem os novos comandos na classe base. Nota-se que por serem dois tipos de som diferentes foi necessário criar dois adendos. Neste caso, o recurso de quantificação de aspectos não foi utilizado. Ou seja, cada um dos adendos implementados atua exatamente sobre um ponto do código base.

```
1: public refines class Bomb {
2:   void enterWater(){
3:     super.enterWater();
4:     m_game_state.getResourceManager().sound(SOUND_WATER_SPLASH);
5:   }
6:   protected void setBombState(){
7:     super.setBombState();
8:     m_game_state.getResourceManager().sound(SOUND_BOMB);
9:   }
10:}
```

Figura 55: Refinamento que insere o som na classe `Bomb`.

```
1: public aspect Sound {
2:   pointcut soundWater(Bomb o):
3:     execution (* Bomb.enterWater()) && target(o);
4:   pointcut soundState(Bomb o):
5:     execution (* Bomb.setBombState()) && target(o);
6:   after(Bomb o): soundWater(o){
7:     o.m_game_state.getResourceManager().sound(SOUND_WATER_SPLASH);
8:   }
9:   after(Bomb o): soundState(o){
10:    o.m_game_state.getResourceManager().sound(SOUND_BOMB);
11:  }
12:}
```

Figura 56: Aspecto que insere o som na classe `Bomb`.

5.4 Comparação com *Mixins*

Mixins são subclasses cuja superclasse é especificada por meio de um parâmetro, cujo valor é informado quando se instancia o *mixin* (ANCONA; LAGORIO; ZUCCA, 2003). Nesse sentido, *mixins* são também semelhantes a refinamentos em AHEAD. A principal diferença é que um *mixin* deve ser explicitamente instanciado, quando informa-se o nome de sua superclasse e o nome da subclasse a ser criada. Ou seja, *mixins* simplificam a criação de subclasses, mas não reduzem a explosão delas em LPS. Para exemplificar o uso de *mixins*, a Figura 57 apresenta um *mixin* – utilizando uma extensão de Java chamada JAM (ANCONA; LAGORIO; ZUCCA, 2003) – que possui os comandos de som que devem ser executados quando da explosão de uma bomba. Nesse *mixin*, a palavra reservada `inherited` (linhas 2-3) identifica que os métodos devem existir na classe pai. Então, para instanciar uma nova classe que utilize o `mixin` deve-se especificar de qual classe o `mixin` herdará. Essa especificação é apresentada na Figura 58. Nota-se que da mesma maneira que em OO, é necessário a criação de uma classe com outro nome. Ou seja, *mixins* não assumem o mesmo nome da superclasse, como ocorre no AHEAD.

```
1: mixin Sound {
2:  inherited void enterWater();
3:  inherited void setBombState();
4:  void enterWater() {
5:     super.enterWater();
6:     m_game_state.getResourceManager().sound(SOUND_WATER_SPLASH);
7:  }
8:  protected void setBombState() {
9:     super.setBombState();
10:    m_game_state.getResourceManager().sound(SOUND_BOMB);
11:  }
12: }
```

Figura 57: *Mixin* para tratamento de som.

```
1: class SoundBomb: Sound extends Bomb{}
```

Figura 58: Criação da subclasse `SoundBomb` usando *mixin* `Sound`.

5.5 Comentários Finais

Neste capítulo, realizou-se uma avaliação qualitativa da LPS proposta e uma comparação com tecnologias alternativas de implementação, incluindo orientação por objetos, orientação por aspectos e *mixins*. Mostrou-se que as principais vantagens do uso de FOP/AHEAD consistem na facilidade e flexibilidade com que se pode gerar produtos diferentes por meio de escolhas de *features*. Argumentou-se também que uma das principais vantagens de AHEAD é a simplicidade de sua linguagem para modularização de *features*.

6 CONCLUSÃO

6.1 Visão Geral do Trabalho

Um segmento relevante do mercado de jogos é o de jogos para dispositivos computacionais móveis, principalmente telefones celulares. Porém, jogos para celulares apresentam desafios extras para seus desenvolvedores. Dentre tais desafios, provavelmente o mais importante consiste em prover suporte à grande variedade de dispositivos celulares existentes no mercado. Assim, jogos para celulares constituem-se em aplicações promissoras para desenvolvimento baseado em linhas de produto de software (LPS).

Para implementação de LPS, diversas construções de programação podem ser usadas, incluindo compilação condicional (CC) (RITCHIE; KERNIGHAN, 1988; STROUSTRUP, 2000), orientação por objetos (OO), *mixins* (ANCONA; LAGORIO; ZUCCA, 2003) e programação orientada por aspectos (AOP) (KICZALES et al., 1997). Porém, recentemente uma nova solução para implementação de LPS, chamada programação orientada por *features* (*Feature-Oriented Programming*, FOP) (BATORY; SARVELA; RAUSCHMAYER, 2003; BATORY, 2004), tem gerado grande interesse na comunidade científica.

Nesta dissertação, foi realizado um trabalho de avaliação do uso de FOP/AHEAD na construção de LPS no domínio de jogos para celulares:

- Descreveu-se uma comparação com três tecnologias para desenvolvimento de LPS: programação orientada por aspectos, compilação condicional e programação orientada por *features*. *Features* foram implementadas utilizando essas três tecnologias em um jogo para celular. Por meio dessa primeira experiência, foi possível comparar o potencial de FOP/AHEAD na implementação de LPS. Assim, decidiu-se prosseguir com o trabalho, investigando o uso de FOP/AHEAD em um jogo de maior complexidade.

- Descreveu-se uma segunda experiência de extração de uma LPS na área de jogos para celulares, utilizando um jogo mais complexo.

- Descreveu-se uma experiência de evolução da LPS extraída a partir da versão original do jogo Bomber. Foram apresentadas as novas *features*, a metodologia utilizada para a evolução da LPS e alguns problemas encontrados.

- Avaliou-se o uso de FOP/AHEAD na extração e evolução da LPS tratada na dissertação. Foram descritas também comparações com tecnologias alternativas para construção de

LPS.

6.2 Contribuições

As principais contribuições deste trabalho são as seguintes:

- A metodologia e a experiência de extração e evolução da LPS tratada na dissertação claramente constituem um ponto de partida interessante para outros desenvolvedores interessados em migrar para uma abordagem de desenvolvimento baseada em linhas de produtos de software.
- O trabalho mostrou alguns benefícios de FOP/AHEAD. Dentre eles, pode-se citar:
  - A possibilidade de se gerar vários produtos de uma linha de produtos de software de uma maneira simples.
  - A possibilidade de compilar as *features* separadamente, sem perder os recursos de verificação estática de tipos de Java.
  - Refinamentos possuem o mesmo nome da classe pai, de forma diferente ao que se consegue em orientação por objetos por meio de herança.
  - Simplicidade de sua linguagem para separação de interesses.
  - Não é necessário declarar conjuntos de junção, definir o tipo de adendos (`after`, `before` ou `around`), associar adendos a conjuntos de junção etc.
  - O número de linhas de código da LPS extraída (e evoluída) é inferior àquele obtido quando se usa compilação condicional.
- No entanto, o trabalho também permitiu detectar algumas desvantagens de FOP/AHEAD, tais como:
  - Programação orientada por *features* acrescenta uma nova dimensão no problema de fragilidade de classes base (MIKHAJLOV; SEKERINSKI, 1998).
  - A inexistência em AHEAD de recursos de quantificação, isto é, um refinamento sempre estende o comportamento de um método de uma classe do programa base.
  - A existência de um forte acoplamento entre os refinamento implementados e as entidades por eles refinadas, o que prejudica o seu reúso em outros sistemas e/ou LPS.

Publicações: Os resultados desta dissertação deram origem às seguintes publicações:

- Rogério Celestino dos Santos; Marco Túlio de Oliveira Valente. Uma Comparação Preliminar entre Tecnologias para Implementação de Variabilidades em Jogos para Celulares.(SANTOS; VALENTE., 2008)
- Rogério Celestino dos Santos; Marco Túlio de Oliveira Valente. Extração de uma Linha de Produtos de Software na Área de Jogos para Celulares usando Programação Orientada por Features.(SANTOS; VALENTE, 2008)

6.3 Trabalhos Futuros

Com objetivo de investigar com mais profundidade o uso de AHEAD, pretende-se fazer novos estudos de caso (mais completos). Pretende-se também estudar construções de LPS utilizando outras plataformas de jogos como, por exemplo, consoles, *arcades*, *desktops* etc.

Uma outra linha de trabalho consiste em propor metodologias para extração de LPS, como, por exemplo técnicas de identificação *features*, técnicas de marcação de códigos referentes a *features* etc. Essa linha de trabalho pode resultar, por exemplo, no desenvolvimento de uma ferramenta para extração semi-automática de *features* em AHEAD.

REFERÊNCIAS

ALVES, V. Identifying variations in mobile devices. *Journal of Object Technology*, p. 47–52, 2004.

ALVES, V. et al. Comparative analysis of porting strategies in J2ME games. In: *21st IEEE International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2005. p. 123–132.

ALVES, V. et al. Refactoring product lines. In: *5th Generative Programming and Component Engineering (GPCE)*. [S.l.: s.n.], 2006. p. 201–210. ISBN 1-59593-237-2.

ALVES, V.; JR., P. M.; BORBA, P. An incremental aspect-oriented product line method for J2ME game development. *1st International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Vancouver, Canada, 2004.

ALVES, V. et al. Extracting and evolving mobile games product lines. In: *9th Software Product Lines Conference (SPLC)*. [s.n.], 2005. p. 70–81. ISBN 3-540-28936-4. Disponível em: <http://dblp.uni-trier.de/db/conf/splc/splc2005.htmlAlvesMCBR05>.

ALVES, V. et al. From conditional compilation to aspects: A case study in software product lines migration. In: *1st Aspect-Oriented Product Line Engineering (AOPLE)*. [S.l.: s.n.], 2006. p. 46–52.

ALVES, V. et al. Beyond code: Handling variability in art artifacts in mobile game product lines. *10th International Software Product Line Conference (SPLC)*, p. 124–132, 2006.

ANCONA, D.; LAGORIO, G.; ZUCCA, E. Jam—designing a java extension with mixins. *ACM Transactions on Programming Languages and Systems*, p. 641–712, 2003.

ANDROID. Disponível em `http://code.google.com/intl/android/android/` Acesso em 11 de fevereiro de 2009.

APEL, S.; BATORY, D. When to use features and aspects? A case study. In: *5th Generative Programming and Component Engineering (GPCE)*. [S.l.: s.n.], 2006. p. 59–68. ISBN 1-59593-237-2.

BATORY, D. A tutorial on feature oriented programming and product-lines. In: *25th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2003. p. 753–754. ISBN 0-7695-1877-X.

BATORY, D. Feature-oriented programming and the AHEAD tool suite. In: *26th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2004. p. 702–703.

BATORY, D.; LIU, J.; SARVELA, J. N. Refinements and multi-dimensional separation of concerns. *9th European software engineering conference (ESEC)*, ACM, p. 48–57, 2003.

BATORY, D.; SARVELA, J. N.; RAUSCHMAYER, A. Scaling step-wise refinement. *25th International Conference on Software Engineering (ICSE)*, p. 187–197, 2003.

BATORY, D. S. Feature models, grammars, and propositional formulas. In: *9th International Software Product Lines Conference (SPLC)*. [S.l.: s.n.], 2005. p. 7–20.

BREW. Disponível em `http://brew.qualcomm.com/` Acesso em 11 de fevereiro de 2009.

CALHEIROS, F. et al. Product line variability refactoring tool. *1st Workshop on Refactoring Tools (WRT)*, p. 33–34, 2007.

CAMARA, T. et al. Massive mobile games porting: Meantime study case. *5th Simpósio Brasileiro de Games (SBGAMES)*, 2006.

CHIBA, S.; ISHIKAWA, R. Aspect-oriented programming beyond dependency injection. *19th European Conference on Object-Oriented Programming (ECOOP)*, Springer, p. 121–143, 2005.

CLEMENTS, P.; NORTHROP, L. M. *Software Product Lines : Practices and Patterns*. [S.l.]: Addison-Wesley, 2001.

FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern*. Disponível em: `http://martinfowler.com/articles/injection.html` . Acesso em 21 janeiro 2009.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object Oriented Software*. [S.l.]: Addison-Wesley, 1997.

HERREJON, R. E. L. Understanding feature modularity in feature oriented programming and its implications to aspect oriented programming. *19th European Conference on Object-Oriented Programming (ECOOP)*, ECOOP, 2005.

IEEE Standard Glossary of Software Engineering Terminology. [S.l.], 1990.

J2ME. Disponível em `http://java.sun.com/javame/index.jsp` Acesso em 11 de fevereiro de 2009.

JONGE, M.; VISSER, J.; IMPLLANG, S. Grammars as feature diagrams. In: *Workshop on Generative Programming (WGP)*. [S.l.: s.n.], 2002. p. 23–24.

KANG, K. et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990. Technical Report.

KASTNER, C.; APEL, S.; BATORY, D. A case study implementing features using aspectj. *11th International Software Product Line Conference (SPLC)*, p. 223–232, 2007.

KICZALES, G. et al. An overview of AspectJ. In: *15th European Conference on Object-Oriented Programming (ECOOP)*. [S.l.]: Springer Verlag, 2001. (LNCS, v. 2072), p. 327–355.

KICZALES, G. et al. Aspect-oriented programming. In: *11th European Conference on Object-Oriented Programming (ECOOP)*. [S.l.]: Springer Verlag, 1997. (LNCS, v. 1241), p. 220–242.

KRUEGER, C. W. Easing the transition to software mass customization. In: *4th International Workshop on Software Product-Family Engineering (SPFE)*. [S.l.: s.n.], 2002. p. 282–293. ISBN 3-540-43659-6.

LEICH, T. et al. *Tool support for feature-oriented software development: FeatureIDE: an Eclipse-based approach*. [S.l.]: 20th International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)., 2005. 55–59 p. ISBN 1-59593-342-5.

LEICH, T.; APEL, S.; SAAKE, G. Using step-wise refinement to build a flexible lightweight storage manager. *9th East European Conference (EEC)*, p. 324–337, 2005.

LIU, J.; BATORY, D.; LENGAUER, C. Feature oriented refactoring of legacy applications. In: *28th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2006. p. 112–121. ISBN 1-59593-375-1.

LOPEZ-HERREJON DON BATORY, W. C. R. E. In: *Evaluating Support for Features in Advanced Modularization Technologies*. [S.l.]: 19th European Conference on Object-Oriented Programming (ECOOP), 2005. p. 169–194.

LOPEZ-HERREJON, R.; APEL, S. Measuring and characterizing crosscutting in aspect-based programs: Basic metrics and case studies. *1st International Conference on Fundamental Approaches to Software Engineering (FASE)*, p. 423–437, March 2007.

LOPEZ-HERREJON, R. E. The expression problem as product-line and its implementation in ahead. *Technical report, Department of Computer Sciences,University of Texas at Austin*, 2004.

MEYER, B. *Object-Oriented Software Construction*. [S.l.]: Prentice Hall, 1988.

MIKHAJLOV, L.; SEKERINSKI, E. A study of the fragile base class problem. In: *12th European Conference on Object-Oriented Programming (ECOOP)*. [S.l.: s.n.], 1998. p. 355–382.

MURPHY, G. C. et al. Separating features in source code: an exploratory study. In: *23rd International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2001. p. 275–284. ISBN 0-7695-1050-7.

NPD: Behind the Numbers. Disponível em `http://www.gamasutra.com/view/feature/3906/npd_behind_the_numbers_december_.php` Acesso em 10 de fevereiro 2009.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications. ACM*, ACM, New York, NY, USA, p. 1053–1058, 1972. ISSN 0001-0782.

PLEVA, G. Game programming and the myth of child's play. *Journal of Computing Sciences in Colleges (CSC).*, Consortium for Computing Sciences in Colleges, , USA, n. 2, p. 125–136, 2004. ISSN 1937-4771.

RIBEIRO, M. de M. et al. On the modularity of aspect-oriented and other techniques for implementing product lines variabilities. *1st Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*, p. 119–130, 2007.

RITCHIE, D. M.; KERNIGHAN, B. W. *The C Programming Language*. [S.l.: s.n.], 1988.

SAMPAIO, P. et al. Portando jogos em J2ME: Desafios, estudo de caso e diretrizes. *III Workshop Brasileiro de Jogos e Entretenimento Digital (WBJE)*, p. 1–7, 2005.

SANTOS, R.; VALENTE, M. T. Extração de uma linha de produtos de software na área de jogos para celulares usando programação orientada por features. In: *2nd Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*. [S.l.: s.n.], 2008. p. 1–10.

SANTOS, R. C. dos; VALENTE., M. T. de O. *Uma Comparação Preliminar entre Tecnologias para Implementação de Variabilidades em Jogos para Celulares.* 2008. 7th Simpósio Brasileiro de Games (SBGAMES), p. 84-87. (technical poster).

STROUSTRUP, B. *The C++ Programming Language*. [S.l.]: Addison-Wesley, 2000.

SYMBIAN. Disponível em `http://www.symbian.com/` Acesso em 11 de fevereiro de 2009.

TIRELO, F. et al. Desenvolvimento de software orientado por aspectos. *23st Jornada de Atualização em Informática (JAI)*, p. 57–96, 2004.

TRUJILLO, S.; BATORY, D.; DIAZ, O. Feature refactoring a multi-representation program into a product line. In: *5th Generative Programming and Component Engineering (GPCE)*. [S.l.: s.n.], 2006. p. 191–200. ISBN 1-59593-237-2.