Uma Comparação Preliminar entre Tecnologias para Implementação de Variabilidades em Jogos para Celulares

Rogério Celestino dos Santos, Marco Túlio Valente
Instituto de Informática, PUC Minas
rogerio.celestino@gmail.com, mtov@pucminas.br

Resumo

Descreve-se e compara-se neste artigo um conjunto de tecnologias para implementação de variabilidades na área de jogos para celulares. Para isso, mostra-se um estudo de caso onde uma variabilidade de um jogo para celular foi implementada por meio de três tecnologias: compilação condicional, programação orientada por aspectos e programação orientada por *features*.

Keywords:: Linhas de produtos de software, Programação orientada por features, Programação orientada por aspectos, Compilação condicional, Jogos para celulares

1 Introdução

O desenvolvimento de jogos para dispositivos móveis é mais simples se comparado com o desenvolvimento para PCs e consoles. No entanto, jogos para dispositivos móveis devem ser adaptados aos diversos tipos de aparelhos celulares existentes. Normalmente, isso requer a manutenção de versões desses sistemas para diferentes 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, diferentes interfaces de rede etc).

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 and Northrop 2001]. Para tanto, 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 ou animações mais complexas.

Diversas tecnologias têm sido propostas para implementação de variabilidades em LPS, incluindo compilação condicional (CC), programação orientada por aspectos (AOP) [Kiczales et al. 1997] e programação orientada por *features* (FOP) [Batory et al. 2003]. Particularmente, neste artigo, descreve-se uma experiência de utilização dessas três tecnologias 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[1]. 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 1. A versão considerada do jogo Chuckie Egg possui 1519 linhas de código Java/J2ME e 15 classes.

Apresenta-se também no artigo uma avaliação qualitativa preliminar dos benefícios e desvantagens de cada uma das três tecnologias de implementação de variabilidades utilizadas no estudo de caso. O objetivo final da experiência e da avaliação apresentadas é subsidiar desenvolvedores de jogos que estejam interessados em utilizar em seus sistemas tecnologias mais modernas para implementação e modularização de variabilidades, como orientação por aspectos e orientação por *features*.

O restante deste artigo está organizado conforme descrito a seguir. Na Seção 2, descreve- se brevemente cada uma das três tecnologias de implementação de variabilidades adotadas no artigo (CC, AOP e FOP). Na Seção 3, descreve-se a implementação de uma *feature* responsável pela introdução de efeitos sonoros no jogo Chuckie Egg usando essas três tecnologias. A Seção 4 apresenta uma avaliação das tecnologias utilizadas no estudo de caso. A Seção 5 apresenta brevemente trabalhos relacionados e a Seção 6 conclui.

Figura 1: Tela do jogo Chuckie Egg

2 Tecnologias para Implementação de Variabilidades

Descreve-se resumidamente nesta seção as três tecnologias para implementação de variabilidades consideradas neste artigo.

Compilação condicional (CC) é um mecanismo de implementação de variabilidades largamente utilizado em linguagens como C e C++. 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[2]. Essa ferramenta utiliza um símbolo especial (//#) para indicar diretivas de compilação. A Figura 2 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 será compilada. Por outro lado, a linha 4 somente será 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 2: Exemplo de Compilação Condicional

Programação orientada por aspectos (AOP) é uma técnica para separação de interesses transversais presentes no desenvolvimento de sistemas [Kiczales et al. 1997]. Interesses transversais são interesses que estão espalhados e entrelaçados em diversos módulos de um sistema. Dentre as linguagens com suporte a AOP, AspectJ é atualmente aquela mais madura e estável [Kiczales et al.

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

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

2001]. Basicamente, AspectJ oferece os seguintes recursos para modularização de interesses transversais: *join points* (pontos da execução de um sistema orientado por objetos que podem ser instrumentados por meio de aspectos), *advices* (trechos de código executados em determinados *join points*) e aspectos, os quais são semelhantes a classes, exceto pelo fato de possuírem como membros designadores de conjuntos pontos de junção e *advices*. AspectJ permite ainda modificar as assinaturas estáticas das classes e interfaces de um programa Java. Por meio de aspectos pode-se, por exemplo, introduzir novos campos e métodos nas classes de um sistema.

Programação orientada por *features* (FOP) é uma outra técnica de modularização e separação de interesses, particularmente adequada para implementação de LPS [Batory et al. 2003]. Basicamente, na terminologia de FOP, uma *feature* representa um acréscimo na funcionalidade básica de um sistema. Ou seja, FOP 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. AHEAD (*Algebraic Hierarchical Equations for Application Design*) é um conjunto de ferramentas que implementam os conceitos básicos de FOP [Batory 2004]. 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 de um programa orientado por objetos. Pode-se ainda adicionar comportamento extra em métodos dessas classes.

3 Estudo de Caso

A versão original do jogo Chuckie Egg não possui suporte a efeitos sonoros. Assim, para realização da comparação descrita neste artigo, resolveu-se incorporar essa *feature* no sistema usando cada uma das três tecnologias para implementação de variabilidades descritas na Seção 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.

Por limitações de espaço, 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 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 3. 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 no artigo, 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 3: Fragmento de código da classe `Chuckie`

3.1 Compilação Condicional

Na Figura 4, apresenta-se a versão da classe `Chuckie` que utiliza CC para implementação da *feature* de Som. Inicialmente, utiliza-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 13-21). Esse método foi criado para evitar repetições de código. Assim, nos método que implementam o pulo do personagem principal foram apenas inseridas chamadas ao método introduzido (linhas 24-26, 30-32 e 36-38). O som será executado logo no início do corpo desses métodos.

Assim, caso a *feature* `Sound` e sua respectiva diretiva de compilação sejam habilitadas, 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:  .....
13:  //#if Sound
14:  //@ protected void playJump() {
15:  //    try{
16:  //@     playJump.play();
17:  //@   } catch (Exception e) {
18:  //@     .....
19:  //    }
20:  //@ }
21:  //#endif
22:
23:  public void jump() {
24:    //#if Sound
25:    //@ playJump();
26:    //#endif
27:    .....
28:  }
29:  public void jump_right() {
30:    //#if Sound
31:    //@ playJump();
32:    //#endif
33:    .....
34:  }
35:  public void jump_left() {
36:    //#if Sound
37:    //@ playJump();
38:    //#endif
39:    .....
40:  }
41:  .....
42:}
```

Figura 4: Implementação da feature Som usando compilação condicional

3.2 Aspectos

A Figura 5 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 *advice* associado a esse conjunto de junção inicializa o atributo que foi introduzido na classe (linhas 14-16). O aspecto proposto contém também um segundo *advice*, 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 *advice* 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 5: Implementação da feature Som usando aspectos

3.3 AHEAD

A Figura 6 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).

4 Avaliação

Com base na experiência adquirida no estudo de caso descrito na Seção 3, apresenta-se a seguir uma avaliação das três tecnologias para implementação de variabilidades 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.

```
 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 6: Implementação da feature Som usando refinamentos

  O mesmo ocorre com o refinamento de som apresentado na Figura 6, 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 4). 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 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 5, 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 6. A existência de tais acoplamento 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 *advices* 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 bastante simples, pode-se concluir que existem benefícios inequívocos 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

5 Trabalhos Relacionados

Em um trabalho anterior, documentamos uma experiência de extração de uma LPS na área de jogos para celulares [Santos and Valente 2008]. As principais diferenças para o presente trabalho são as seguintes: (1) a experiência foi realizada com um outro jogo (chamado Bomber[3]); (2) a experiência envolveu uma única tecnologia para implementação de variabilidades (programação orientada por *features*). Assim, no presente trabalho, procurou-se de forma integrada – isto é, utilizando o mesmo sistema alvo – mostrar os princípios de funcionamento, os benefícios e as principais limitações de duas outras tecnologias: programação orientada por aspectos e compilação condicional.

Batory cita o emprego de programação orientada por *features* na implementação de famílias de produtos envolvendo gerenciadores de bancos de dados, protocolos de redes, controladores de robôs e sistemas de controle de aeronaves [Batory 2004]. No entanto, tais experiências foram baseadas no sistema de FOP chamado GenVoca (do qual AHEAD é uma evolução). Alves et al. descrevem um conjunto de estratégias para migrar uma LPS implementada usando compilação condicional para aspectos, usando para isso um jogo para celular [Alves et al. 2006]. Mostram ainda que alguns padrões de variações não podem ser migrados para aspectos. Kastner, Apel e Batory descrevem o uso de AspectJ para refatorar um sistema gerenciador de bancos de dados (Oracle Berkley DB) em uma linha de produtos [Kästner et al. 2007]. Assim como Alves, apontam também algumas limitações de aspectos quando usados com esse objetivo.

Com objetivo similar ao deste artigo, Lopez-Herrejon, Batory e Cook avaliam diversas tecnologias para implementação de *features*, incluindo não apenas AspectJ e AHEAD, mas também outros sistemas e linguagens de programação, como Hyper/J, Jiazzi e Scala [Lopez-Herrejon et al. 2005]. No entanto, a avaliação realizada é baseada em um LPS hipotética, consistindo de uma calculadora com diferentes operações (adição, subtração etc) e formatos de números (números inteiros, decimais etc). Por outro lado, no presente artigo procurou-se demonstrar e comparar o uso de tecnologias para implementação de variabilidades utilizando um sistema simples, mas real, no domínio de jogos para celulares.

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

6 Considerações Finais

Neste artigo, realizou-se uma comparação entre 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*. Na 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. Como trabalho futuro, pretende-se estender a LPS gerada no artigo, investigando novas *features*. Pretende-se também utilizar essas técnicas para modularizar *features* típicas de jogos para consoles e PCs.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG e CNPq.

Referências

ALVES, V., NETO, A. C., SOARES, S., SANTOS, G., CALHEIROS, F., NEPOMUCENO, V., PIRES, D., LEAL, J., AND BORBA, P. 2006. From conditional compilation to aspects: A case study in software product lines migration. In *1st GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 1–6.

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

BATORY, D. 2004. Feature-oriented programming and the AHEAD tool suite. In *26th International Conference on Software Engineering (ICSE)*, 702–703.

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

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, Springer Verlag, vol. 1241 of *LNCS*, 220–242.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, Springer Verlag, vol. 2072 of *LNCS*, 327–355.

KÄSTNER, C., APEL, S., AND BATORY, D. 2007. A case study implementing features using AspectJ. In *11th International Software Product Line Conference (SPLC)*, 223–232.

LOPEZ-HERREJON, R., BATORY, D., AND COOK, W. R. 2005. Evaluating support for features in advanced modularization technologies. In *19th European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag, vol. 3586 of *LNCS*, 169–194.

SANTOS, R., AND VALENTE, M. T. 2008. Extração de uma linha de produtos de software na área de jogos para celulares usando programação orientada por features. In *II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*, 1–10.