Extração de uma Linha de Produtos de Software na Área de Jogos para Celulares usando Programação Orientada por Features

Rogério Celestino dos Santos, Marco Túlio de Oliveira Valente

Instituto de Informática, PUC Minas
rogerio.celestino@gmail.com, mtov@pucminas.br

Abstract. *This paper documents an experience in extracting a software product line in the domain of mobile phone games. The described product line has been implemented using feature-oriented programming, as supported by the AHEAD system. The paper presents a quantitative and qualitative evaluation of the extracted product line and compares AHEAD with alternative technologies, notably with aspect-oriented programming.*

Resumo. *Descreve-se neste artigo uma experiência de extração de uma linha de produtos de software na área de jogos para celulares. Particularmente, a linha de produtos descrita no artigo foi implementada usando-se conceitos de programação orientada por features, tal como disponibilizados pelo sistema AHEAD. O artigo apresenta também uma avaliação quantitativa e qualitativa da linha de produtos extraída, bem como compara AHEAD com tecnologias alternativas, notadamente com programação orientada por aspectos.*

1 Introdução

Jogos digitais constituem aplicações cada vez mais importantes na indústria de software. 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 [1]. 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 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 [7]. 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. Do ponto de vista de implementação, diversas construções de programação podem ser usadas para apoiar a criação de LPS, incluindo construções mais antigas e bem conhecidas – como compilação condicional – até novas abstrações para modularização, como aspectos e *features* [11].

Neste artigo, descreve-se uma experiência de emprego de programação orientada por *features* – ou *feature-oriented programming* (FOP) [6, 12] – para extrair uma linha de produtos de software na área de jogos para celulares. Assim como desenvolvimento orientado por aspectos, FOP é considerada uma técnica moderna de modularização e separação de interesses, particularmente adequada para implementação de requisitos transversais heterogêneos [4]. Basicamente, na terminologia de FOP, uma *feature* representa um acréscimo na funcionalidade básica de um sistema. Assim, 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. A fim de viabilizar a implementação de uma LPS usando princípios e idéias básicas de FOP, utiliza-se no artigo o conjunto de ferramentas e linguagens conhecido por AHEAD [5]. A aplicação usada como estudo de caso é um jogo para celulares implementado em J2ME.

O restante deste artigo está organizado conforme descrito a seguir. Na Seção 2, são introduzidos conceitos básicos de programação orientada por *features*, de forma bastante influenciada pela terminologia e abstrações providas pela plataforma AHEAD. Na Seção 3, descreve-se a extração de uma LPS a partir da versão original do jogo Bomber. Apresenta-se o modelo de *features* da LPS proposta e descreve-se como *features* entrelaçadas no código original do jogo foram extraídas para módulos independentes, usando-se as construções e abstrações providas pela plataforma AHEAD. A Seção 4 avalia a linha de produtos gerada, segundo critérios quantitativos e qualitativos. Com base na experiência adquirida com o estudo de caso descrito no trabalho, a Seção 5 compara programação orientada por *features* com tecnologias alternativas para implementação de variabilidades em LPS, notadamente com programação orientada por aspectos. Por fim, a Seção 6 apresenta brevemente alguns trabalhos relacionados e a Seção 7 conclui o artigo.

2 Programação Orientada por Features Usando AHEAD

Em FOP, uma *feature* representa um acréscimo na funcionalidade básica de um programa. FOP advoga então que *features* devem ser tratadas como abstrações de primeira classe no projeto de sistemas e, como tal, devem ser implementadas em unidades de modularização independentes [6, 12]. 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. Além disso, deve ser possível combinar módulos que representam *features* de forma flexível, sem perder os recursos de verificação estática de tipos [11].

AHEAD (*Algebraic Hierarchical Equations for Application Design*) [5] é um conjunto de ferramentas que implementa os conceitos básicos de FOP, viabilizando o projeto de sistemas de acordo com os princípios descritos no parágrafo anterior. 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.

Por exemplo, suponha a classe `Buffer` da Figura 1 contendo um único inteiro e

métodos `get` e `set`[1]. Suponha ainda o refinamento mostrado na parte (b) dessa figura, o qual estende a classe `Buffer` com uma nova *feature*: a possibilidade de realizar um *undo* no *buffer*, restaurando o último valor armazenado no mesmo. Basicamente, esse refinamento consiste em: acrescentar um novo campo (`back`) na classe refinada (linha 2); acrescentar um novo método (`restore`) na classe refinada (linhas 3-5); e alterar o comportamento do método original `set`, fazendo com que ele salve o novo valor do buffer no campo `back` (linha 7), antes de prosseguir com sua execução original (linha 8).

```
1: class Buffer {
2:   int buf= 0;
3:   int get() {
4:     return buf;
5:   }
6:   void set(int x) {
7:     buf=x;
8:   }
9: }
```

(a)

```
 1: refines class Buffer {
 2:   int back= 0;
 3:   void restore() {
 4:     buf= back;
 5:   }
 6:   void set(int x) {
 7:     back= buf;
 8:     Super().set(x);
 9:   }
10: }
```

(b)

Figura 1. Exemplo de refinamento em AHEAD

Além da linguagem Jakarta, o ambiente AHEAD contém dois outros componentes: um compilador – chamado `composer` – responsável por combinar código de *features* com código base de um sistema, gerando assim um produto específico da LPS e uma linguagem para descrever combinações de *features* válidas. Essa linguagem permite que o `composer` detecte combinações inválidas ao se tentar gerar um determinado produto.

3 Estudo de Caso

Neste artigo, descreve-se uma experiência de extração de uma LPS na área de jogos para celulares. O objetivo do jogo escolhido como estudo de caso – chamado Bomber[2] – é 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 2.

No estudo realizado, uma abordagem extrativa foi empregada para gerar uma LPS [7]. Inicialmente, 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 visualmente.

- Som, incluindo efeitos sonoros implementados por meio da API padrão da plataforma J2ME ou usando APIs proprietárias de fabricantes de celulares. Essas

[1]Este exemplo de programação orientada por *features* foi extraído de [9].
[2]Disponível em: http://j2mebomber.sourceforge.net.

Figura 2. Telas do jogo Bomber

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 3. 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[3]); (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[3]).

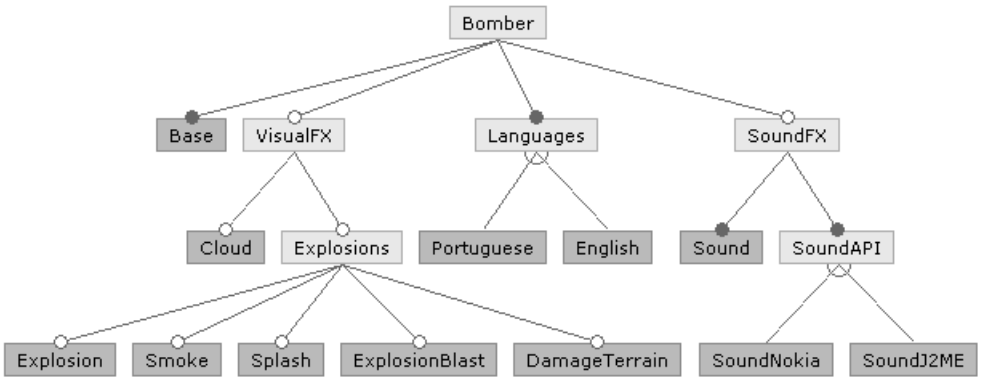

Figura 3. Modelo de Features

[3]Observe que este *ou* é exclusivo.

3.1 Extração da Linha de Produtos

A fim de extrair a LPS representada pelo modelo de *features* da Figura 3, 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 3 (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.

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 então migrado para refinamentos $R_1, R_2, \ldots, R_n$.

Descreve-se a seguir a aplicação desses passos em uma das classes do jogo Bomber. A Figura 4 mostra o código da classe `Bomb` após inserção de comentários indicando a presença de *features* (linhas 5-6 e 10-12). A Figura 5 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 uma chamada do método de mesmo nome da superclasse. Segundo, foi necessário extrair um método, chamado `destroy` (linhas 8-10), 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 4), a solução encontrada foi extrair um dos comandos internos do `if` para o método `destroy`. O método extraído é que será então refinado para reintroduzir o código dessas *features* na classe `Bomb`.

A Figura 6 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 `destroy` (linhas 6-9) da classe `Bomb`.

4 Avaliação

4.1 Avaliação Quantitativa

A versão original do sistema Bomber possui 42 classes, 9 interfaces e 7784 LOC. Já a linha de produtos extraída possui 8483 LOC (isto é, um acréscimo de cerca de 9% em relação à versão original). Esse acréscimo ocorre devido ao código extra demandado na declaração dos refinamentos (incluindo código para reimplementação da assinatura dos métodos refinados, para redirecionamento da execução para métodos originais, via comando `Super`, para chamada e implementação de métodos extraídos etc).

A Tabela 1 apresenta informações detalhadas sobre o número de linhas de código demandado para modularização de cada uma das *features* identificadas. Pode-se verificar

```
 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 4. Classe Bomb da versão original com comentários indicando *features*

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

Figura 5. Classe Bomb refatorada (com apenas funcionalidades básicas)

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

Figura 6. Refinamento que introduz a feature Sound na classe Bomb

nessa tabela que o código base do sistema representa cerca de 86% do código da linha de produtos (i.e. 7255/8483). Observa-se ainda que, em média, os módulos criados para implementação de *features* possuem 111.6 LOC. É importante também esclarecer que na versão original do sistema a característica `Sound` se encontra espalhada e entrelaçada pelos seus diversos módulos. Por esse motivo, os valores de `SoundJ2ME` e `SoundNokia` – isto é, dos refinamentos criados para modularização dessa característica – são nulos nessa versão.

Feature	LOC Original	LOC LPS	FCD
Base	7147	7255	-
English	50	95	1
Portuguese	0	0	1
Sound	53	148	8
SoundJ2ME	0	36	1
SoundNokia	0	39	1
Cloud	60	76	4
Explosion	15	61	7
ExplosionBlast	123	169	6
Smoke	197	326	5
Splash	108	158	7
DamageTerrain	31	120	7
Total	7784	8483	-
Média (excluindo Base)	57.9	111.6	4.36

Tabela 1. Métricas

Mostra-se também na Tabela 1 o valor da métrica chamada *Feature Crosscutting Degree* (FCD) [10], a qual representa o número de classes do código base que são refinadas pelo código modularizado de uma *feature*. Caso uma *feature* modifique a implementação de dois ou mais métodos de uma mesma classe, conta-se apenas uma vez essa classe no cálculo da métrica FCD. Como pode ser observado, em média, as *features* extraídas refinam 4.36 classes do programa base. Ou seja, as *features* implementadas apresentam um baixo grau de transversalidade, como já reportado em outras experiências de extração de LPS [4, 10].

4.2 Avaliação Qualitativa

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

- Configurabilidade: como esperado em uma LPS, considera-se que a principal vantagem da solução implementada neste trabalho é a facilidade com que se pode derivar diversos produtos. Na verdade, 768 diferentes versões do jogo Bomber podem ser geradas, bastando para isso selecionar corretamente os arquivos de *features* ao se chamar o combinador do ambiente AHEAD.

- Compilação em separado e verificação estática de tipos: 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.

- Reusabilidade: 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 6 referencia dois de seus métodos (`enterWater` e `destroy`) e um campo (`m_game_state`). Ou seja, existe um forte acoplamento entre os refinamento implementados no estudo de caso e as entidades por eles refinadas. Esse acoplamento prejudica a utilização desses refinamentos em outros sistemas.

4.3 Riscos à Validade do Estudo de Caso

O estudo de caso escolhido é um jogo de tamanho e complexidade médios. Acredita-se que sua escolha foi representativa para demonstrar as principais vantagens e desvantagens de processos de extração de *features* usando a plataforma AHEAD. Por outro lado, algumas situações mais complexas associadas a refatorações de *features* não foram detectadas na experiência realizada. Dentre elas, podemos citar: refinamentos na hierarquia de herança (por exemplo, uma *feature* exigir que uma classe do programa base estenda uma determinada classe); refinamentos que acessam variáveis locais de um método (já que refinamentos em AHEAD têm acesso apenas a atributos da classe refinada) e refinamentos que alteram o comportamento de outros refinamentos (ou então, um refinamento cuja implementação seja afetada pela presença ou não de um outro refinamento [9]). Em trabalhos futuros, pretendemos evoluir a linha de produtos extraída, de forma a analisar esses refinamentos mais complexos.

5 Comparações com Outras Tecnologias

Nesta seção, compara-se FOP com três tecnologias alternativas para implementação de LPS: herança, *mixins* e aspectos.

Herança: Em certo sentido, um refinamento em AHEAD é semelhante a extensão de uma classe por meio de herança. A principal diferença é que uma subclasse tem 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 iria requerer a implementação de múltiplas subclasses (uma para cada possível combinação de *features*). Além disso, seria necessário configurar o sistema para usar as subclasses corretas (por meio, por exemplo, de arquivos de configuração ou de técnicas de injeção de dependência).

Mixins: *Mixins* são subclasses cuja superclasse é especificada por meio de um parâmetro (cujo valor é informado quando se instancia o *mixin*) [3]. 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.

Aspectos: Aspectos – conforme definidos em AspectJ – 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, 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`).

Assim, 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 tem acesso direto ao ambiente da classe refinada, o que simplifica a sua sintaxe.

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, experiências mais realistas de uso de AspectJ têm demonstrado que tais requisitos não são tão comuns, notadamente no caso de implementação de LPS [4, 10].

6 Trabalhos Relacionados

São apresentados nesta seção alguns trabalhos que relatam experiências de extração de LPS. 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 controles de aeronaves [5]. No entanto, em geral, tais experiências de uso de FOP são descritas com menos detalhes que a descrição contida no presente artigo. Ou então, são baseadas no sistema de FOP chamado GenVoca (do qual AHEAD é uma evolução). Apel e Batory descrevem o uso de FOP na implementação de uma LPS na área de sistemas *peer-to-peer* [4]. De certa forma, no presente artigo, procurou-se também descrever a implementação de uma LPS usando FOP, porém em um outro domínio (jogos para celulares).

Existem ainda trabalhos que procuram aplicar aspectos na extração de linhas de produtos. Kastner, Apel e Batory documentam o uso de AspectJ para refatorar um sistema gerenciador de bancos de dados (Oracle Berkley DB) em uma linha de produtos [8]. A conclusão é que aspectos não constituem uma tecnologia encorajorada para realização dessa tarefa, implicando em diversos problemas de manutenibilidade e reusabilidade do código extraído. 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 [2]. Mostram ainda que alguns padrões de variações não podem ser migrados para aspectos.

7 Conclusões

Descreveu-se neste trabalho uma experiência de extração de uma LPS na área de jogos para celulares usando programação orientada por *features*, mais especificamente o sistema AHEAD. Como conclusão, pode-se afirmar que FOP/AHEAD é uma tecnologia interessante para implementação dos padrões de *features* mais comuns em LPS. Em geral,

as *features* implementadas possuem um comportamento heterogêneo, o qual não exige recursos de quantificação, nem designadores avançados de conjuntos de junção. Como trabalho futuro, pretende-se estender a LPS gerada, de modo a poder avaliar com mais precisão os benefícios e limites do sistema AHEAD. O objetivo final é disponibilizar uma LPS de referência para demonstrações e pesquisas envolvendo FOP.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG e CNPq.

Referências

[1] Vander Alves, Ivan Cardim, Heitor Vital, Pedro Sampaio, Alexandre Damasceno, Paulo Borba, and Geber Ramalho. Comparative analysis of porting strategies in J2ME games. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 123–132, 2005.

[2] Vander Alves, Alberto Costa Neto, Sergio Soares, Gustavo Santos, Fernando Calheiros, Vilmar Nepomuceno, Davi Pires, Jorge Leal, and Paulo Borba. From conditional compilation to aspects: A case study in software product lines migration. In *1st GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, pages 1–6, 2006.

[3] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, 2003.

[4] Sven Apel and Don Batory. When to use features and aspects?: a case study. In *5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 59–68, 2006.

[5] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *26th International Conference on Software Engineering (ICSE)*, pages 702–703, 2004.

[6] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *25th International Conference on Software Engineering (ICSE)*, pages 187–197, 2003.

[7] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley, 2001.

[8] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[9] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[10] Roberto Lopez-Herrejon and Sven Apel. Measuring and characterizing crosscutting in aspect-based programs: Basic metrics and case studies. In *10th International Conference on Fundamental Approaches to Software Engineering (FASE)*, March 2007.

[11] Roberto Lopez-Herrejon, Don Batory, and William R. Cook. Evaluating support for features in advanced modularization technologies. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 169–194. Springer-Verlag, 2005.

[12] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.