UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

ADOÇÃO DE TECNOLOGIA PARA PORTE DE JOGOS MÓVEIS: O CASO DA MEANTIME

TRABALHO DE GRADUAÇÃO

Andrea Frazão de Menezes (afm3@cin.ufpe.br)

Recife, Janeiro de 2008

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

TRABALHO DE GRADUAÇÃO

ADOÇÃO DE TECNOLOGIA PARA PORTE DE JOGOS MÓVEIS: O CASO DA MEANTIME

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

Aluna: Andrea Frazão de Menezes (afm3@cin.ufpe.br)
Orientador**:** Geber Lisboa Ramalho (glr@cin.ufpe.br)
Co-orientador**:** Paulo Henrique Monteiro Borba (phmb@cin.ufpe.br)

Recife, Janeiro de 2008

Resumo

A atividade de porte demanda um esforço significativo dos desenvolvedores devido a vários desafios decorrentes da grande diversificação dos dispositivos móveis presentes no mercado, cada um contendo suas características específicas e limitações.

Atualmente, a técnica de porte mais utilizada na indústria para tratar as variações no código dos jogos móveis é a compilação condicional. Essa técnica, embora eficiente, traz algumas desvantagens, como por exemplo, deixa o código de uma solução de porte espalhado e entrelaçado com código de outras soluções, o que dificulta a legibilidade e o reuso. Dessa forma, as empresas de desenvolvimento de jogos móveis estão sempre buscando soluções de porte mais eficientes, escaláveis e menos custosas.

A Meantime Mobile Creations, em parceria com o C.E.S.A.R., UFPE/CIn, UFBA e UPE, com o apoio da Financiadora de Estudos e Projetos – FINEP, está trabalhando em um projeto de desenvolvimento de uma ferramenta, FLiP (Ferramenta para Linha de Produtos), que tem como objetivo facilitar o processo de porte de jogos móveis, utilizando conceitos de linha de produtos e programação orientada a aspectos.

Este trabalho realizou um estudo dos principais fatores relacionados ao sucesso da adoção da ferramenta de porte FLiP pela Meantime, trazendo um diagnóstico desses fatores, fazendo sugestões de melhorias para que o processo de adoção seja bem sucedido, e, por fim, sugerindo um roteiro de implantação, para minimizar os riscos e contemplar as melhorias sugeridas. Alguns destes resultados podem ser aplicados por uma empresa qualquer de desenvolvimento de jogos móveis, preocupada em adotar novas tecnologias para porte de aplicações, problema técnico central desta indústria hoje.

Palavras-Chave: Porte de jogos móveis, programação orientada a aspectos, linha de produtos de software, adoção de tecnologia.

Agradecimentos

Ao chegar ao fim de mais uma etapa de minha vida, deixo aqui os meus agradecimentos a todas as pessoas que, de alguma forma, deram a sua contribuição para que eu pudesse concluir a minha graduação.

Em primeiro lugar, agradeço à minha família pelo apoio dado durante todo o curso, e, sobretudo a este trabalho.

Agradeço ao meu orientador, Prof. Geber Ramalho por todo o apoio neste trabalho e ao meu co-orientador Prof. Paulo Borba pela orientação desde que passei a participar da equipe de desenvolvimento do FLiP.

Um agradecimento especial a todos os membros do FLiP e da Meantime. Sem a colaboração deles não teria sido possível a realização deste trabalho.

Agradeço a todos os professores e funcionários do CIn, aos colegas de turma, com os quais partilhei tantos trabalhos, tantas preocupações e alegrias.

E, por fim, meu agradecimento a UFPE, pela formação que recebi.

Sumário

1. Introdução

O mercado de jogos móveis é um dos mais promissores da indústria de entretenimento digital. Espera-se que, até 2011, aproximadamente US$ 7 bilhões sejam movimentados em todo o mundo neste mercado [Tercek, 2007].

Na medida em que o mercado de dispositivos móveis cresce, os fabricantes lançam um número cada vez maior de dispositivos diferentes, buscando atingir os mais diversos segmentos de mercado.

Desse contexto surge a necessidade das empresas de desenvolvimento de jogos móveis adaptarem suas aplicações para o maior número de dispositivos possível, criando múltiplas versões dos jogos otimizadas para cada dispositivo específico, em busca de maior lucratividade. Essa atividade de adaptação é chamada porte (*porting*).

A atividade de porte, que, em 2006, representou um terço do custo total do desenvolvimento de jogos móveis [Tercek, 2007], demanda um esforço significativo dos desenvolvedores devido a vários desafios decorrentes da grande diversificação dos dispositivos móveis presentes no mercado, cada um contendo suas características específicas e limitações [Cardim et al., 2005].

Atualmente, a técnica de porte mais utilizada na indústria para tratar as variações no código dos jogos móveis é a compilação condicional [Camara et al., 2006]. Essa técnica é bastante eficiente e evita que o código seja replicado, porém deixa o código de uma solução de porte espalhado e entrelaçado com código de outras soluções, o que dificulta a legibilidade e o reuso. Dessa forma, as empresas de desenvolvimento de jogos móveis estão sempre buscando soluções de porte mais eficientes, escaláveis e menos custosas.

A Meantime Mobile Creations [Meantime, 2007], empresa de desenvolvimento de jogos móveis, em parceria com o C.E.S.A.R., UFPE/CIn, UFBA e UPE, com o apoio da Financiadora de Estudos e Projetos – FINEP, está trabalhando em um projeto de desenvolvimento de uma ferramenta, FLiP (Ferramenta para Linha de Produtos), que tem como objetivo facilitar o processo de porte de jogos móveis, utilizando conceitos de linha de produtos [Clements et al., 2002] e programação orientada a aspectos (POA) [Kiczales, 1997]. A abordagem em que o FLiP se baseia faz uso de refatorações contínuas para isolar as variações do código da base comum a todas as aplicações armazenando-as em aspectos. A partir de um jogo desenvolvido independentemente para plataformas diferentes, chega-se a uma infra-estrutura de linha de produtos contendo a base do jogo e aspectos que representam as variações.

No entanto, a adoção de uma ferramenta de inovação, como o FLiP, por uma empresa, é um processo arriscado, que requer um estudo detalhado de impactos e benefícios.

Este trabalho tem como objetivo realizar um estudo dos principais fatores relacionados ao sucesso da adoção de uma ferramenta de porte por uma empresa de desenvolvimento de jogos móveis, trazendo um diagnóstico desses fatores, no caso da adoção da ferramenta FLiP pela Meantime, e sugestões de melhorias para que o processo de adoção seja bem sucedido. E, por fim, sugerir um roteiro de implantação, para minimizar os riscos e contemplar as melhorias sugeridas.

Este trabalho está organizado da seguinte forma: o capítulo 2 descreve a problemática que motiva este estudo, inicialmente abordando a adoção de ferramentas por empresas como problemática geral e o porte de jogos móveis como problemática específica e, por

fim, descreve os requisitos necessários a uma solução de porte para que esta seja adotada com sucesso por uma empresa de desenvolvimento de jogos. O capítulo 3 traz as principais soluções de porte citadas na literatura, descrevendo técnicas, modelos de organização e abordagens de porte e, por fim, traz uma comparação entre as diferentes soluções. O capítulo 4 descreve o processo de desenvolvimento e porte de jogos móveis utilizado, atualmente, pela Meantime. O capítulo 5 faz uma descrição da ferramenta FLiP, sua organização e funcionalidades. O capítulo 6 trata do diagnóstico da adoção do FLiP pela Meantime com relação aos requisitos listados no capítulo 2, além de trazer sugestões de melhorias e um roteiro de implantação da ferramenta. Por fim, o capítulo 7 apresenta as conclusões deste trabalho.

2. Problemática

Neste capítulo serão apresentados dois aspectos da problemática abordada neste estudo. Inicialmente, apresentaremos uma visão geral, que envolve a questão de adoção de tecnologia por empresas. Em seguida trataremos da problemática mais específica, que diz respeito à necessidade das empresas de desenvolvimento de jogos móveis de portar suas aplicações para a diversidade de dispositivos presentes no mercado. Por fim descreveremos os principais requisitos considerados necessários para que uma solução de porte seja adotada por uma empresa de desenvolvimento de jogos móveis com sucesso.

2.1. Problemática Geral: Adoção de Tecnologia

Nos dias atuais, a velocidade com que a competição tecnológica cresce força as empresas a adotarem novas posturas estratégicas e a conceberem novas formas de desenvolver tecnologia, buscando criar oportunidades de mercado para os seus produtos.

As revoluções tecnológicas podem ou não afetar significativamente empresas pertencentes aos setores industriais tradicionais, pois, na medida em que seus resultados e demandas são consideravelmente previsíveis, as incertezas ambientais são reduzidas [Bignetti, 2002].

Por sua vez, empresas de alta tecnologia, ou intensivas em conhecimento, vivem constantes situações de turbulência tendo, como exceção, momentos de estabilidade. Estas empresas são caracterizadas por introduzir inovações radicais e por possuírem produtos com ciclos de vida curtos, detendo assim um potencial de crescimento altíssimo, porém disputam mercados extremamente competitivos. Suas novas gerações de produtos exigem alto investimento em P&D para que possam ter condições de enfrentar o mercado competitivo em que se encontram. Aqueles que têm o poder de decisão dessas empresas procuram constantemente por novos mercados e novas oportunidades, o que inclui novas tendências tecnológicas [Bignetti, 1999].

Neste contexto, o processo de adoção de tecnologia é resultado de um processo de tomada de decisão e de interação social. Neste processo de competição tecnológica, uma tecnologia será escolhida vencedora, mesmo não sendo necessariamente a melhor [David, 1986; Arthur, 1996]. A solução final é muitas vezes, ao contrário do esperado – que o melhor vença – a do mais ousado. O processo de inovação é visto como um processo em espiral, no qual o desenvolvimento de uma nova tecnologia está vinculado a sua implementação e não como um processo seqüencial da pesquisa ao mercado [Bignetti, 2002].

Dessa forma, faz-se necessário realizar um estudo diagnóstico da adoção de uma tecnologia de inovação por uma empresa, seguido de proposições de adaptação de ambas as partes para que o processo de adoção tenha sucesso.

2.2. Problemática Específica: Porte de Jogos Móveis

Na medida em que o mercado de dispositivos móveis cresce, as empresas fabricantes de celulares buscam lançar um número cada vez maior de dispositivos com o objetivo de atender aos vários segmentos de mercado com diferentes necessidades e recursos financeiros, em um intervalo de tempo cada vez menor.

As operadoras e *publishers*, por sua vez, necessitam que o maior número de usuários possível tenha acesso aos jogos lançados, para que estes alcancem maior lucratividade. Dessa forma, é necessário desenvolver múltiplas versões dos jogos otimizadas para cada dispositivo específico [Sampaio et al., 2004] . Assim o desenvolvimento de jogos para celulares, que é usualmente considerado mais simples quando comparado com jogos para PCs e consoles, possuindo, inclusive, um ciclo de desenvolvimento geralmente mais curto [Cardim et al., 2005], passa a incorporar uma tarefa crítica, que é o porte.

Por definição, porte é o termo usado para descrever o processo de tradução de um software para que este possa ser executado em diferentes computadores ou sistemas operacionais além do que foi originalmente desenvolvido. Geralmente existem razões financeiras por trás da necessidade do porte, uma vez que tal atividade permite que o produto seja disponibilizado para um segmento maior de mercado.

A atividade do porte demanda um esforço significativo dos desenvolvedores devido a vários desafios decorrentes da grande diversificação dos dispositivos móveis presentes no mercado, cada um contendo suas características específicas e limitações [Cardim et al., 2005], tais como:

- Diferentes tamanhos de tela, números de cores, sons e *layouts* de teclados;
- Diferentes disponibilidades de memória de execução e armazenamento;
- Existência de APIs proprietárias e pacotes opcionais;
- *Bugs* específicos de cada dispositivo;
- Internacionalização (tradução do jogo para diversas línguas).

Java 2 Micro Edition (J2ME), versão da linguagem Java para dispositivos móveis, foi a primeira linguagem amplamente suportada por tais dispositivos. Sua arquitetura é dividida em uma série de configurações, perfis e pacotes opcionais, para que seja possível adaptar as aplicações criadas para a diversidade de dispositivos existentes.

Existem duas configurações possíveis: uma para dispositivos com maior capacidade computacional denominada CDC (*Connected Device Configuration*) e outra para dispositivos com menor capacidade computacional, denominada CLDC (*Connected Limited Device Configuration*). Uma configuração define o ambiente de execução e um conjunto de classes básicas, chamadas de core. Cada dispositivo se enquadra em uma configuração, dependendo de suas características de memória e poder de processamento e, assim, não se perde a capacidade de aparelhos mais sofisticados nem se torna inviável o desenvolvimento para aparelhos mais limitados.

Mesmo dentro de uma configuração, ainda existem diferenças grandes entre os dispositivos. Assim, foi criada a definição de perfil, que funciona como uma extensão de uma configuração, fornecendo uma série de APIs que serão utilizadas para o desenvolvimento das aplicações. Um perfil suporta uma categoria mais estreita de dispositivos dentro da estrutura de uma configuração escolhida. O MIDP (*Mobile*

Information Device Profile) é o perfil mais conhecido e bastante utilizando combinado com a CLDC.

A plataforma J2ME pode ser estendida adicionando-se APIs opcionais para o uso de tecnologias existentes ou emergentes tais como conectividade a banco de dados, multimídia, *Bluetooth* e *web services*. Além disso, muitos fabricantes provêem APIs proprietárias que também servem para estender as funcionalidades do J2ME padrão.

A princípio, todas as aplicações poderiam ser desenvolvidas utilizando apenas recursos comuns a todos os dispositivos, descartando o uso de inovações de perfis e configurações, bem como a utilização de pacotes opcionais e APIs proprietárias, o que reduziria bastante o esforço em torno da atividade de porte. Porém, isso causaria a perda do potencial dos dispositivos mais sofisticados além de que, em alguns casos, as operadoras requerem a inclusão de suas APIs proprietárias nos dispositivos que comercializam e exigem que os desenvolvedores utilizem estas bibliotecas.

Além disso, existem variações padrões que não se pode evitar, tais como: diferentes características dos dispositivos no que se refere à interface com o usuário, particularmente tamanho de tela, número de cores, sons e disposição do teclado, *bugs* conhecidos, disponibilidade de memória, tamanho máximo da aplicação e línguas internacionais.

A manutenção destas variações torna-se uma tarefa dispendiosa e demasiadamente passível de erros, já que o núcleo funcional comum da aplicação está disperso nas diferentes variações [Sampaio et al., 2004].

A Figura 2.1 mostra um jogo sendo executado em três plataformas diferentes. Para tanto, são necessárias três versões diferentes do mesmo jogo.

Figura 2.1: Diferentes dispositivos executando diferentes versões de um mesmo jogo.

O primeiro dispositivo, pertence à plataforma Nokia Serie 40 e possui restrições de memória maiores em relação aos outros dispositivos. O segundo dispositivo, pertencente à plataforma Nokia Série 60, dispõe de uma memória maior que os demais,

além de possuir uma tela maior. O terceiro dispositivo, pertencente à plataforma T720 da Motorola, possui uma tela muito pequena, além de não possuir um recurso chamado *flip*, que permite a criação do espelhamento de uma imagem durante a execução da aplicação, disponível apenas na API da Nokia [Vasconcelos, 2005].

Realizar as mudanças necessárias para fazer o porte da versão de uma plataforma para outra é um trabalho demorado e complexo. Além disso, deve-se considerar também, o aspecto da manutenção, já mencionado. Para diminuir este impacto, é necessário separar em módulos as variações específicas de cada um dos dispositivos, da parte comum a todos. A Programação Orientada a Objetos é bastante limitada para separar estas características, pois geralmente, o código de cada variação não está isolado, nem definido em um único trecho de código, e sim entrelaçado com outros códigos e disperso em diversas classes.

Na melhor das hipóteses, o produto é desenvolvido de uma forma escalável, de maneira que possa ser portado para outras plataformas facilmente, ainda oferecendo uma experiência positiva para o usuário. Desenvolvedores de jogos móveis tendem a construir sua arquitetura de jogos de forma que após uma versão ser completamente desenvolvida, soluções bem adaptadas possam ser construídas baseadas na versão original. A atividade do porte se tornou tão crítica no desenvolvimento de jogos móveis que atualmente existem empresas especializadas em prestar tal serviço [Tira Wireless, 2007].

2.3. Requisitos Necessários a uma Solução de Porte

Devido às restrições do domínio de jogos para dispositivos móveis, o mercado extremamente competitivo, o ciclo de desenvolvimento curto e o número crescente de dispositivos-alvo do porte, uma série de requisitos são necessários a uma solução de porte para que esta seja adotada com sucesso por uma empresa de desenvolvimento de jogos móveis. A seguir descreveremos os principais entre estes requisitos.

2.3.1. Requisitos Técnicos

Consideramos técnicos os requisitos que envolvem restrições do domínio de aplicações para dispositivos móveis, aspectos inerentes ao produto final, isto é, aqueles requisitos que dizem respeito a restrições que devem ser atendidas para que uma aplicação para dispositivos móveis seja executada satisfatoriamente. Entre estas restrições destacamos:

- Tamanho da aplicação: os dispositivos móveis possuem, no geral, limitações grandes de memória e, dessa forma, as aplicações desenvolvidas para estes dispositivos possuem restrições rigorosas com relação ao tamanho máximo que podem atingir.
- Alocação de Memória: é mais uma restrição do domínio de jogos móveis. Ainda que alguns dispositivos mais sofisticados possuam capacidade de alocação de memória maior que a média, e que sejam uma tendência os dispositivos evoluírem no sentido de ter mais memória, as aplicações também evoluem em complexidade e geralmente utilizam o limite de recursos disponíveis. Assim, um constante cuidado deve ser tomado com relação a este fator.

Dessa forma é requisito de uma solução de porte não gerar *overhead* no tamanho da aplicação final e nem na alocação de memória.

2.3.2. Requisitos de Implantação

Consideramos requisitos relacionados à implantação aqueles que envolvem aspectos e atividades ligados à implantação da solução de porte, tais como:

- Aquisição de ferramentas: ao adotar uma nova solução, uma empresa precisa adquirir licenças para utilizar as ferramentas envolvidas nesta solução. Algumas ferramentas possuem licenças de uso muito caras, tornando impraticável seu uso por empresas de pequeno ou médio porte.
- Adaptação de tecnologias existentes: quando uma empresa adota uma nova solução, é necessário analisar se esta irá substituir ou trabalhar em conjunto com outras ferramentas já utilizadas pela empresa. Caso esta solução vá trabalhar em conjunto com as ferramentas existentes, adaptações devem ser realizadas para integrar as ferramentas existentes com a nova solução.
- Adaptação do processo de desenvolvimento: neste caso, de forma semelhante à questão da adaptação de tecnologias existentes, é necessário saber se a nova solução irá substituir ou trabalhar em conjunto com as tecnologias existentes. Caso vá substituir as ferramentas existentes, um novo processo de desenvolvimento deverá ser criado. Caso esta solução vá trabalhar em conjunto com as existentes, o processo de desenvolvimento deve ser adaptado.
- Curva de aprendizado e treinamentos: toda solução, que envolve ferramentas e tecnologias novas, possui uma curva de aprendizado associada a ela. Para amenizar essa curva de aprendizado treinamentos podem ser oferecidos.

É desejado que uma solução de porte não esteja associada a altos custos de aquisição de ferramentas, que a adaptação das tecnologias e processos existentes possa ocorrer de forma menos impactante possível e que a curva de aprendizado não seja demasiadamente longa.

2.3.3. Resultados Esperados

Ao adotar uma nova solução, uma empresa espera obter resultados melhores do que os que obtém com a tecnologia utilizada antes da adoção. Alguns resultados positivos que uma empresa de desenvolvimento de jogos móveis espera de uma solução de porte são, por exemplo:

- Produtividade: entende-se por produtividade a relação entre os resultados obtidos por uma empresa e os recursos utilizados. O conceito de produtividade está relacionado com a capacidade de uma empresa de gerar produtos no seu processo produtivo.
- Escalabilidade: indica a habilidade da solução de manipular uma porção crescente de trabalho de forma uniforme.

- Legibilidade do código: o código deve ser escrito de maneira clara, de forma que possa ser facilmente compreendido.

Uma solução de porte adotada por uma empresa deve proporcionar: maior produtividade, isto é, um aumento na produção da empresa com relação aos recursos utilizados; escalabilidade, neste caso, as ferramentas envolvidas na solução devem manter-se operando de forma eficiente na medida em que novos dispositivos alvo de porte são adicionados à linha de produção da empresa; legibilidade de código, pois não é interessante que o código gerado seja demasiadamente poluído, o que dificulta a compreensão e manutenção.

3. Soluções de Porte Existentes

Atualmente o processo de porte é realizado de maneira ineficiente, pois separar o código das características específicas dos dispositivos do código comum a todas as plataformas não é uma tarefa trivial, já que tais características encontram-se espalhadas pelo código fonte e, geralmente, estão entrelaçadas com outras características [Alves, 2004].

Podemos encontrar, na literatura, algumas soluções para o problema do porte, porém apenas algumas foram validadas na indústria [Cardim et al., 2005].

Neste capítulo descreveremos as principais soluções de porte citadas na literatura, classificadas em três grupos: técnicas de porte, que abrange as tecnologias utilizadas para tratar as variações do porte; modelos de organização, que diz respeito à forma de organizar as variações e artefatos em geral, resultantes do porte; e abordagens de porte, que trata de estratégias de realização do porte. Por fim, faremos uma comparação entre as soluções apresentadas, com relação aos requisitos necessários a uma solução de porte enumerados na seção 2.3.

3.1. Técnicas de Porte

Nesta seção apresentaremos as principais técnicas de porte citadas na literatura, descrevendo a sua utilização, vantagens e desvantagens.

3.1.1. Compilação Condicional

Esta técnica consiste em identificar pontos de variação entre as plataformas alvo e, utilizando compilação condicional, apoiada por ferramentas de pré-processamento, isolar o código específico de cada plataforma do código comum a todas.

Existem várias ferramentas que dão suporte a recursos de pré-processamento, como Antenna [Antenna, 2007] e J2ME Polish [J2ME Polish, 2007], onde diretivas definem uma compilação condicional do código escrito para todas as plataformas de maneira que apenas os trechos de código pertencentes às diretivas especificadas sejam compilados.

Um estudo de caso do porte do jogo My Big Brother [Cardim et al., 2005] nos traz alguns exemplos de tratamento de variações utilizando compilação condicional. A listagem 3.1 mostra um trecho de código resultante do tratamento de variações de tamanho de tela.

```
class Resources {...
        //#ifdef SCREEN_SIEMENS
        public static final int SCREEN_HEIGHT = 80;
        //#elifdef SCREEN_N40
        public static final int SCREEN_HEIGHT = 128;
        //#elifdef SCREEN_N60
        public static final int SCREEN_HEIGHT = 208;
        ...
        //#endif
        ...
    }
```

Listagem 3.1: Tratamento de variações de tamanho de tela.

A listagem 3.2 mostra o tratamento de variação de mapeamento de teclado.

```
class GameController {...
        public static int BOARD_SOFT_LEFT = 0;
        public static int BOARD_SOFT_RIGHT = 0; ...
        static{
        //#ifdef KEYS_C650
        BOARD_SOFT_LEFT = -21;
        BOARD_SOFT_RIGHT = -22;
        //#elifdef KEYS_T720
        BOARD_SOFT_LEFT = -6;
        BOARD_SOFT_RIGHT = -7; ...
        //#elifdef KEYS_V300
        BOARD_SOFT_LEFT = -21;
        BOARD_SOFT_RIGHT = -22;
        //#elifdef KEYS_SIEMENS
        BOARD_SOFT_LEFT = -1;
        BOARD_SOFT_RIGHT = -4;
        //#endif
        } ...
}
```

Listagem 3.2: Tratamento de variação de mapeamento de teclado.

Esta técnica é muito utilizada atualmente. É bastante eficiente, uma vez que evita que o código seja totalmente replicado para que se comece a portar para outra plataforma e, além de tratar das variações entre as plataformas, trata também das novas *features* que possam ser inseridas em uma determinada versão do jogo. Outra vantagem é que não há *overhead* no tamanho final da aplicação. Entretanto, o uso de compilação condicional prejudica consideravelmente a legibilidade do código, além de não ser escalável, pois para adicionar um novo dispositivo alvo de porte é necessário codificar as variações referentes a estes em vários locais espalhados no código.

3.1.2. Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) foi criada no fim da década de 1990, nos laboratórios da Xerox, por Gregor Kiczales e sua equipe. A POA trata de um problema específico: capturar unidades consistentes de um sistema de software que as limitações dos modelos de programação tradicionais deixam espalhados por diversos pontos do sistema.

Em sistemas desenvolvidos utilizando Programação Orienta a Objetos (POO) é comum que certos requisitos fiquem definidos em vários trechos de código (requisitos transversais), em vez de ficarem apenas em um bloco de código ou em uma determinada classe como é desejado. Isso é, muitas vezes, um fator negativo, pois prejudica a legibilidade e manutenção do código. Em alguns casos, a funcionalidade além de espalhada pelo código, está entrelaçada com o código de outras funcionalidades. Devido a restrições de linguagem e, pela própria arquitetura da implementação, existem ocasiões em que não é possível modularizar alguns requisitos utilizando apenas recursos de POO.

Algumas implicações deste problema são [Vasconcelos, 2005]:

- Redundância: muitos fragmentos de código iguais, ou semelhantes, ocorrem em diversos pontos do código;
- Forte acoplamento: os métodos das classes que implementam requisitos transversais geram dependência nos métodos de outras classes;
- Fraca coesão: métodos contêm instruções que não estão diretamente relacionadas às funcionalidades que eles implementam;
- Dificuldade de compreensão, manutenção e reutilização: como a implementação do requisito transversal depende do código espalhado pela aplicação, sua compreensão, manutenção e reutilização ficam comprometidas.

Dessa forma a POA traz uma solução para este problema, complementando POO, na medida em que coloca a implementação espalhada de um requisito transversal (*crosscutting concern*) dentro de uma unidade chamada aspecto.

AspectJ [Kiczales et al., 2001] é uma extensão da linguagem Java orientada a aspectos.

Para compreender o funcionamento da POA e de AspectJ é necessário entender alguns conceitos fundamentais detalhados a seguir:

- *Join points*: pontos bem definidos na execução de um programa. Podem ser uma chamada de método ou construtor, acesso a um atributo, tratamento de exceção, inicialização de classes e objetos.
- *Pointcuts*: construção da linguagem para identificar *join points*. Têm como principal finalidade expor o contexto de um *join point* para um *advice*.
- *Advice*: especifica o código que será executado quando certo *join point* for atingido. *Advices* usam *pointcuts* para interferirem na execução do programa, executando algum trecho de código antes ou depois do *join point* referenciado, ou ainda substituindo a execução deste por outro.
- *Intertype declarations*: são estruturas que tornam possível introduzir novos atributos, métodos e construtores, dentro de uma classe já existente.
- Aspectos: entidades compostas de *pointcuts* e *advices* que são combinados com classes Java, a fim de modificar o comportamento destas.

Tendo em vista que as variações presentes nos dispositivos móveis geralmente são transversais ao código base da aplicação, isto é, estão espalhadas no código da aplicação e entrelaçadas com o código de outras variações, a utilização de aspectos para isolar as variações dos dispositivos móveis é uma solução atrativa, uma vez que todo o código relativo a uma variação poderia estar em apenas um aspecto, o que facilitaria sua legibilidade, manutenção e reuso.

Uma desvantagem do uso de aspectos em jogos para dispositivos móveis é o *overhead* gerado no tamanho da aplicação.

3.1.3. Program Transformation

Consiste na utilização de uma técnica para gerenciar variações, existentes nas diferentes versões de um mesmo jogo, sistematicamente ao mesmo tempo em que preserva as características comuns.

Em um estudo de caso realizado [Cardim et al., 2005] foi utilizado *program transformation* para realizar o porte do jogo *The Rain of Fire*. Foi utilizado o *Java Transformation System* (JaTS) [JaTS, 2001], uma ferramenta capaz de aplicar transformações definidas pelo usuário a um programa Java. A solução consistiu em extrair o código de uma plataforma abstrata (o *core*) dos códigos das versões existentes, de forma que este core contivesse todas as *features* comuns a todas as versões. Em seguida o código para cada plataforma seria gerado aplicando transformações no *core*.

Inicialmente foi necessário realizar comparações entre os códigos existentes a fim de se separar o *core* das variações. Em seguida foram definidos pares de transformações (um conjunto de mudanças no código fonte), para cada variação: uma que deveria transformar o código específico de uma plataforma no *core* e outra transformação que deveria gerar o código específico a partir do *core*. Então, foram identificados padrões de código que estiveram presentes em todas as ocorrências de cada variação. E, finalmente, são criados os *templates* JaTS para casar esses padrões no código fonte e realizar as transformações.

Esta solução se mostrou eficiente e simples. Uma vez que as transformações JaTS agem diretamente no código fonte, o código para cada plataforma é legível e localizado. Fazendo uso das transformações é possível traçar um caminho de porte de uma plataforma para qualquer outra, através do *core*. No entanto existem algumas desvantagens: os padrões de casamento do JaTS possuem flexibilidade limitada, e os *templates* utilizados para estas transformações dependem parcialmente do código do *core*. Dessa forma, a evolução do core, potencialmente levaria à evolução dos *templates*.

3.2. Modelos de Organização

Nesta seção descreveremos duas formas de organização dos artefatos envolvidos no porte de jogos móveis. A primeira é uma forma ad-hoc de organização sem regras explícitas. Enquanto a segunda determina regras bem definidas de organização e uso desses artefatos.

3.2.1. Ad-hoc

Algumas empresas possuem repositórios de variações e bibliotecas de componentes reusáveis e fazem uso de ambos na criação de novos produtos. Porém, muitas vezes, não existem regras bem definidas que especifiquem como esse reuso deve ocorrer, nem existe relação entre o repositório de variações e os componentes reusáveis, que indiquem como estes estão associados. Assim, essas empresas organizam as variações e artefatos em geral resultantes do porte de jogos móveis de maneira ad-hoc.

3.2.2. Linha de Produtos

Na medida em que a demanda por softwares cresce em vários domínios com padrões de qualidade crescentes e ciclos de desenvolvimento menores, linha de produtos de software vem emergindo como uma promessa, não apenas de aumentar a qualidade e diminuir os custos do desenvolvimento de softwares, mas também de reduzir o *time-to-market.*

Uma linha de produtos de software consiste em um conjunto de produtos desenvolvidos a partir do mesmo conjunto de artefatos e focados em um domínio específico. De fato, migrando para uma abordagem de linha de produtos, uma organização pode aumentar sua competitividade em um domínio específico de desenvolvimento de software. Ao mesmo tempo, adotar esta abordagem requer comprometimento organizacional e técnico [Alves, 2004].

Uma linha de produtos representa uma família de software em determinado domínio de aplicação que possuem características em comum e que são desenvolvidos tendo a mesma base (*core*). Uma linha de produtos de software é composta por um conjunto de artefatos reusáveis e um conjunto de regras especificando como estes artefatos devem ser gerenciados.

O processo que parte de uma linha de produtos e gera um software específico é chamado de instanciação. Mais especificamente, a instanciação consiste em, a partir da composição e customização de alguns dos artefatos reusáveis e do eventual desenvolvimento de novos artefatos, dar origem a um produto específico [Carmo, 2005].

Existem várias abordagens para se desenvolver uma linha de produtos de software. Podemos citar: abordagem proativa, reativa e extrativa [Krueger, 2001]. Na abordagem proativa a organização projeta e implementa uma nova linha de produtos para dar suporte a todo o escopo de produtos previsto. Na abordagem reativa, a organização desenvolve incrementalmente uma linha de produtos existente quando ocorre a demanda de um novo produto ou novos requisitos em produtos existentes. Na abordagem extrativa, a organização extrai produtos existentes para uma linha de produtos.

Uma vez que a abordagem proativa demanda um grande investimento inicial e oferece mais riscos, ela pode ser inadequada para algumas organizações, em particular para pequenas e médias empresas de desenvolvimento de software com projetos que possuem cronogramas restritos. Em contraste, as outras duas abordagens possuem escopo reduzido, requerem baixo investimento e dessa forma podem ser mais adequado para tais organizações [Alves et al., 2007].

A utilização de linhas de produtos para gerenciar as variações dos dispositivos no desenvolvimento e porte de jogos móveis é uma solução que pode ser associada a outras soluções como pré-processamento, programação orientada a aspectos, *program transformation*.

Entre os estudos realizados mais recentemente podemos encontrar indicações de que a utilização de linha de produtos de software é uma opção para reduzir os custos de desenvolvimento e agilizar o processo de porte, levando os produtos para o mercado em um tempo satisfatório [Alves, 2004].

A combinação das abordagens reativa e extrativa para desenvolver uma linha de produtos se enquadra a realidade de uma empresa de desenvolvimento de jogos móveis. Particularmente um processo de extração incremental de linha de produtos para jogos

móveis, utilizando programação orientada a aspectos foi proposto [Alves et al., 2004]. Mais detalhes desse processo serão dados a mais adiante.

3.3. Abordagens

Nesta seção descreveremos duas abordagens de porte de jogos móveis, sendo a primeira uma abordagem artesanal e a segunda uma proposta de extração de linha de produtos a partir de jogos existentes.

3.3.1. Abordagem Incremental

A abordagem incremental consiste na tentativa de portar a aplicação individualmente para as plataformas desejadas. A aplicação é desenvolvida para um dispositivo e a partir desta versão modelo faz-se incrementalmente o porte para as outras plataformas.

Alguns estudos de caso realizados [Menon, 2005; Cardim et al., 2005] levantam os principais pontos a serem levados em consideração ao realizar este tipo de porte. Nos dois casos citados o porte foi feito a partir de uma aplicação inicialmente desenvolvida para a família de dispositivos Nokia série 60, que possui especificações bastante superiores à média da época em que as aplicações foram desenvolvidas, tendo como alvo a família de dispositivos Nokia série 40 que possui pelo menos duas grandes limitações: tamanho máximo da aplicação de 64 KB e memória *heap* com capacidade de 200 KB.

O primeiro ponto levantado diz respeito ao tamanho da aplicação. Para reduzir o tamanho da aplicação, a fim de que esta possa ser executada por um dispositivo com menor poder de processamento, pode-se remover *features* opcionais que não comprometam a jogabilidade da aplicação, simplificar animações e interface, remover imagens que não sejam necessárias, redimensionar as imagens para tornar os elementos do jogo proporcionais ao tamanho da tela do dispositivo alvo, reestruturar o código de maneira a diminuir o tamanho do *bytecode* gerado.

O segundo ponto diz respeito à utilização da memória *heap*. Nos dois casos citados acima, devido a um *bug* nos dispositivos da série 60, que impede que o *garbage collector* libere totalmente a memória ocupada por um objeto de imagem, optou-se por adotar uma política de alocação de imagens que carrega todas as imagens que seriam utilizadas no jogo no início da aplicação e as deixa na memória enquanto a aplicação estiver rodando. Como a memória *heap* de tais dispositivos é suficientemente grande, não há problema em tal técnica. No entanto, o mesmo não acontece com os dispositivos da série 40 que possuem, conforme dito anteriormente, uma limitação de memória *heap*. Dessa forma foi necessário mudar a política de alocação de imagens. Uma vez que tais dispositivos não possuem o *bug* da plataforma da série 60, foi possível utilizar uma política de alocação que mantém na memória apenas as imagens que estão sendo utilizadas.

Os estudos de caso resultaram em algumas recomendações de boas práticas para o porte incremental de aplicações móveis.

Foi possível concluir que não é uma boa prática desenvolver a aplicação inicial para um dispositivo com maior poder de processamento e maior capacidade de memória, tendo como alvo do porte um dispositivo com mais restrições. A razão é que o esforço para

fazer a aplicação ser executada satisfatoriamente em um dispositivo com restrições maiores é muito grande. Para realizar as adaptações, no caso remover funcionalidades, imagens etc. é necessário realizar um estudo detalhado do dispositivo alvo e é requerido um tempo considerável.

Uma boa prática seria desenvolver o jogo inicialmente para um dispositivo com mais restrições e, após um modelo ter sido desenvolvido, poder-se-ia realizar o porte para dispositivos mais sofisticados, acrescentando funcionalidades opcionais. Dessa forma, quando se está desenvolvendo um jogo para um dispositivo móvel, é necessário ter em mente o ambiente restrito para o qual se está desenvolvendo.

No entanto, ainda que a aplicação seja desenvolvida para um dispositivo com mais restrições existem ainda variações que não se podem evitar, tais como tamanho de tela, *input* de teclado, som, fontes, etc.

Uma recomendação para lidar com essas variações é modularizá-las, isto é, separar o código relativo a cada um desses aspectos de modo que estes possam ser facilmente identificados e adaptados ao dispositivo alvo.

Ainda que experiências de porte de aplicações resultem em boas práticas para facilitar atividades futuras de porte incremental, existe ainda um problema em tal abordagem. Para cada dispositivo portado, o código da aplicação é replicado, ou seja, existe um código distinto para cada família de dispositivos, o que traz uma conseqüência negativa para a manutenção da aplicação: um defeito encontrado ou uma melhoria que seja realizada deve ser implementada em todas as versões do código, gerando um esforço extra para os desenvolvedores e, ocasionalmente, inconsistência entre as versões.

3.3.2. Abordagem Incremental de Extração de Linha de Produtos Utilizando Orientação a Aspectos

Uma abordagem propõe a utilização programação orientada a aspectos e linha de produtos para tratar as variações relativas ao porte [Alves et al., 2004]. É proposto um processo extrativo incremental para estruturar uma Linha de Produtos a partir de jogos para dispositivos móveis existente, utilizando refatorações contínuas e programação orientada a aspectos para isolar variações do código comum a todas as aplicações. Utilizando esta abordagem é possível fatorar efetiva e modularmente variações em aspectos e posteriormente compô-las com a base da aplicação.

Conforme dito anteriormente existem diversas abordagens para se desenvolver uma linha de produtos de software: proativa, reativa e extrativa. Uma vez que a abordagem proativa suporta todo o escopo do produto, esta requer de antemão um grande investimento e oferece mais riscos. Em contraste, as outras duas abordagens têm escopo reduzido e assim requerem um investimento menor. Elas são incrementais e desta forma são mais adequadas para empresas pequenas ou de médio porte.

Neste contexto, foi proposto um método que combina a abordagem extrativa e reativa, que, inicialmente, extrai variações de uma aplicação existente e depois, reativamente, adapta a linha de produtos recentemente criada para abranger outras variações de produtos. A combinação dessas duas abordagens foi escolhida por diversos motivos. Primeiramente, pequenas e médias empresas, que podem se beneficiar do uso de linhas de produtos, geralmente não podem pagar pelos altos custos da abordagem proativa. Em

segundo lugar, em domínios como o desenvolvimento de jogos móveis, o ciclo de desenvolvimento é tão curto que o planejamento proativo não pode ser completado. Em terceiro lugar, existem riscos associados à abordagem proativa, uma vez que o escopo pode se tornar inválido em detrimento da adição de novos requisitos.

Existe um grande número de técnicas para gerenciar variações dos requisitos ao nível de código. A maior parte destas técnicas faz uso de conceitos de orientação a objetos. Entretanto, estas técnicas são conhecidas por falharem na tentativa de capturar interesses transversais que, freqüentemente, aparecem em domínios de grande variação [Alves et al., 2005]. Jogos móveis, em particular, precisam atender aos requisitos de portabilidade, que são consideravelmente transversais, o que sugere o uso de programação orientada a aspectos para lidar com as variações.

A abordagem proposta é incremental: a partir de um jogo desenvolvido independentemente para plataformas diferentes, chega-se a uma infra-estrutura de linha de produtos contendo a base do jogo e aspectos que representam as variações.

Primeiramente, o método cria a linha de produtos e depois a desenvolve com uma abordagem reativa. Inicialmente, pode haver um ou mais produtos independentes, que serão refatorados para que se obtenham as variações que irão formar a linha de produtos. Em seguida, o escopo da linha de produto é estendido para abranger outro produto: a linha de produtos reage para acomodar a nova variação.

O primeiro passo é extrair a linha de produtos a partir de um ou mais produtos existentes, extraindo a base comum e construindo a correspondência das variações específicas de cada produto. De acordo com a natureza do domínio essa correspondência pode ser feita utilizando POA, conforme é mostrado na Figura 3.1.

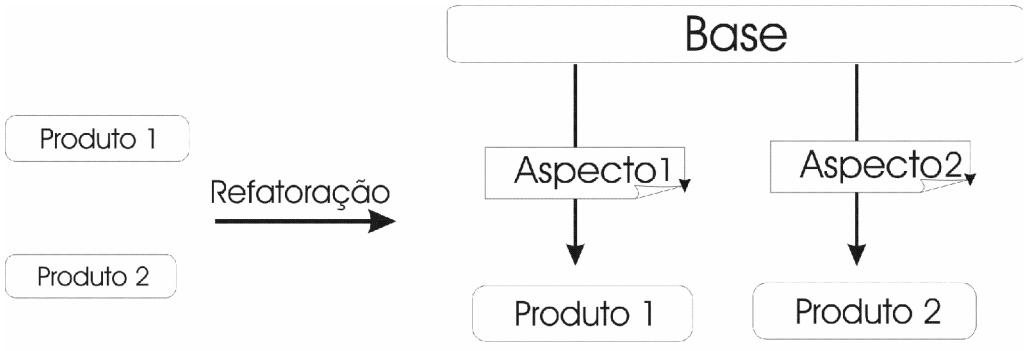

Figura 3.1: Extração da linha de produtos.

Produto 1 e Produto 2 são aplicações existentes do mesmo domínio. Base (*core*) representa a parte comum a essas aplicações. A Base será combinada com Aspecto1 ou Aspecto2 para instanciar os produtos originais. Estes aspectos encapsulam o código específico de cada produto.

O mapeamento entre *features* da linha de produtos e aspectos é especificado por um mecanismo de *configuration knowledge* [Czarnecki et al., 2000], que impõe restrições nas combinações entre *features* e aspectos como dependências, combinações ilegais e combinações padrão. Restrições envolvendo apenas combinações de *features* são especificadas no *feature model*.

O diagrama de *features* para a linha de produtos, conforme mostrado na Figura 3.2, é composto de duas *subfeatures* alternativas F1 e F2, representando Produto 1 e Produto 2, respectivamente.

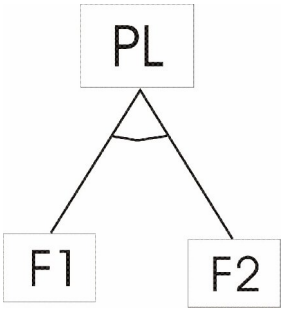

Figura 3.2: Diagrama de *features* da linha de produtos.

Após a linha de produtos ter sido criada, esta pode ser desenvolvida para abranger produtos adicionais. Neste processo, um novo aspecto é criado para adaptar a base à nova variação como mostra a Figura 3.3.

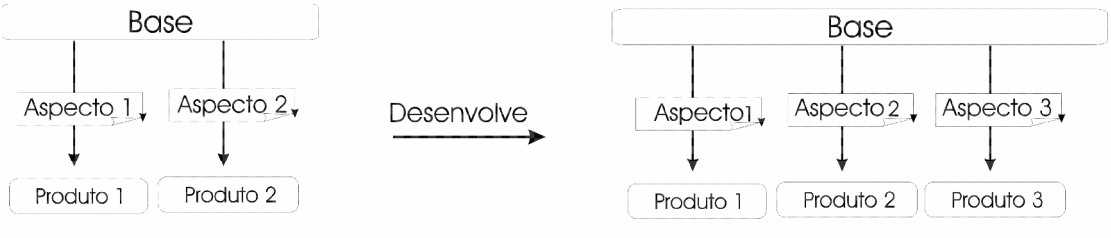

Figura 3.3: Desenvolvimento da linha de produtos.

Além disso, uma nova *feature* é adicionada à linha de produtos para representar o novo produto e o *configuration knowledge* é atualizado para mapear a nova *feature* ao novo aspecto, como mostra a Figura 3.4.

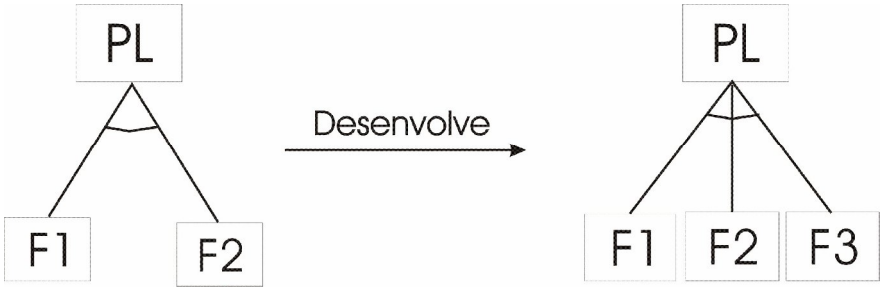

Figura 3.4: Desenvolvimento do diagrama de *features*.

Conforme mostra a Figura 3.5, a partir de uma implementação inicial para uma plataforma, pode-se obter uma versão para outra plataforma, extraindo para aspectos as partes do código que correspondem às características específicas da primeira, obtendo como produto dessa operação o código comum às duas – a base - e um conjunto de

aspectos correspondentes às características específicas da primeira plataforma – os primeiros artefatos da linha de produtos. Novos aspectos são criados para implementar as características específicas da segunda plataforma. Ao final, tem-se uma linha de produtos com dois produtos instanciáveis [Alves et al., 2005].

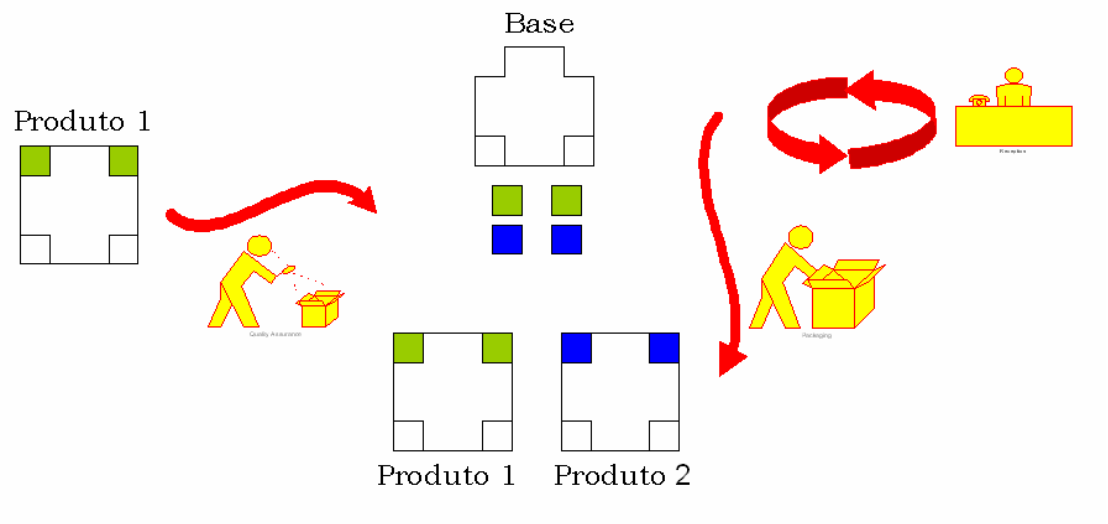

Figura 3.5: Linha de Produtos.

Quando for necessário fazer alterações de requisitos, altera-se apenas o código base ou um aspecto referente a um determinado produto. Quando se quer instanciar um novo produto apenas combina-se a base com o aspecto referente ao produto desejado. Para adicionar um novo produto à linha de produtos, é necessário, apenas, implementar novos aspectos que atuam sobre o código base.

3.4. Comparação Entre as Soluções

Nesta seção faremos comparações entre as soluções de porte apresentadas neste capítulo com relação aos requisitos necessários a uma solução de porte enumerados na seção 2.3. É importante salientar que os requisitos de implantação relativos à adaptação de tecnologias e adaptação de processo não foram considerados, pois é necessário ter conhecimento das tecnologias e processos utilizados pela empresa que está adotando a solução para se avaliar o esforço.

3.4.1. Técnicas de Porte

Em relação aos requisitos técnicos, especificamente à restrição de tamanho da aplicação, a técnica de compilação condicional não causa impacto. O mesmo não acontece com a programação orientada a aspectos, pois a inserção de aspectos no código gera um *overhead* no tamanho final da aplicação. Em relação à técnica *program transformation* não foram realizados testes suficientes para chegar a resultados conclusivos. O mesmo ocorre com relação à alocação de memória, embora se acredite que o uso das técnicas apresentadas não cause impacto neste requisito.

Em relação à aquisição de ferramentas, as três técnicas apresentadas trazem resultados satisfatórios, pois existem ferramentas livres de suporte a compilação condicional, como o Antenna e o J2ME Polish, programação orientada a aspectos, como AspectJ e *program transformation*, como o JaTS. Em relação à curva de aprendizado, devido ao uso de tecnologias amplamente utilizadas no mercado, acreditamos que compilação condicional e programação orientada a aspectos sejam mais interessantes do que *program transformation.*

No que diz respeito aos resultados esperados, acreditamos que todas as técnicas apresentadas tragam resultados satisfatórios em relação à produtividade, o mesmo não acontece em relação à escalabilidade, onde a técnica de programação orientada a aspectos traz um melhor resultado na medida em que isola soluções de porte em um aspecto, facilitando o reuso, enquanto a compilação condicional deixa uma mesma solução espalhada pelo código e entrelaçada com outras soluções. No caso de *program transformation*, conforme dito anteriormente os templates gerados são dependentes do código da base e, dessa forma, a evolução da base naturalmente leva a evolução dos templates.

Por fim, em relação à legibilidade de código, a compilação condicional, ao contrário das demais técnicas que isolam o código das variações em arquivos distintos, deixa o código das soluções de porte espalhado pelo código fonte e entrelaçado com outras soluções, o que dificulta a legibilidade.

A Tabela 3.1 apresenta as técnicas de porte e seus respectivos resultados com relação aos requisitos especificados na seção 2.3, segundo nossa análise.

Técnica	Requisitos Técnicos		Requisitos de Implantação		Resultados Esperados		
	Tamanho da aplicação	Alocação de Memória	Aquisição de Ferramentas	Curva de Aprendizado	Produtividade	Escalabilidade	Legibilidade
Compilação Condicional	Bom	Não conclusivo	Bom	Bom	Bom	Ruim	Ruim
Programação Orientada a Aspectos	Ruim	Não conclusivo	Bom	Bom	Bom	Bom	Bom
Program Transformation	Não conclusivo	Não conclusivo	Bom	Ruim	Bom	Ruim	Bom

Tabela 3.1: Técnicas de Porte.

3.4.2. Modelos de Organização

Em relação aos requisitos técnicos, os modelos de organização descritos não causam impactos, pois não exercem influencia diretamente no código da aplicação. Dizem respeito, apenas a forma de organização e utilização das soluções de porte e dos artefatos gerados.

No que diz respeito aos requisitos de implantação, o modelo de linha de produtos não é ideal quando se trata da aquisição de ferramentas, pois um estudo realizado com as

ferramentas de linha de produto existentes no mercado mostra que as ferramentas de código livres não disponibilizam todas as funcionalidades necessárias para a utilização de linha de produtos no porte de jogos móveis sendo, assim, necessária a aquisição de ferramentas proprietárias. Já o modelo ad-hoc não especifica a utilização de ferramentas.

Em relação à curva de aprendizado, o modelo de linha de produtos tende a ser mais interessante, pois o que diferencia o reuso de artefatos em uma linha de produtos de software de outras formas de reuso é que, em uma linha de produtos, os artefatos são específicos para uma única família de produtos, o que torna o processo de instanciação mais simples. Além disso, em uma linha de produtos, o processo de instanciação de cada produto específico é detalhadamente documentado, isto é, não só existe um conjunto de artefatos reusáveis, mas também um conjunto de instruções de como gerar produtos específicos a partir desses artefatos. Enquanto em um modelo ad-hoc não há essa especificação, o que pode tornar o processo de instanciação mais difícil.

A facilidade do reuso no modelo de linha de produtos tende a trazer maior produtividade e escalabilidade ao processo de instanciação.

Assim como nos requisitos técnicos, os modelos de organização não causam impacto direto na legibilidade do código.

A Tabela 3.2 apresenta os modelos de organização e seus respectivos resultados com relação aos requisitos especificados na seção 2.3, segundo nossa análise.

Modelo de Organização	Requisitos Técnicos		Requisitos de Implantação		Resultados Esperados		
	Tamanho da aplicação	Alocação de Memória	Aquisição de Ferramentas	Curva de Aprendizado	Produtividade	Escalabilidade	Legibilidade
ad-hoc	Não influencia	Não influencia	Bom	Ruim	Ruim	Ruim	Não influencia
Linha de Produtos	Não influencia	Não influencia	Ruim	Bom	Bom	Bom	Não influencia

Tabela 3.2: Modelos de Organização.

3.4.3. Abordagens de Porte

Em relação ao tamanho da aplicação e à alocação de memória a abordagem incremental pode ter um resultado insatisfatório, em casos como os descritos nos estudos de caso citados, onde um jogo é inicialmente desenvolvido para um dispositivo mais poderoso e precisa ser portado para um dispositivo com maiores restrições. No caso da abordagem extrativa que utiliza linha de produtos e aspectos, a restrição de tamanho da aplicação também é um problema, pelo mesmo motivo do overhead gerado no tamanho da aplicação pela inserção de aspectos no código. Em relação à alocação de memória não foram realizados testes utilizando esta abordagem, embora acreditemos que a inserção de aspectos não cause impactos neste requisito.

No que diz respeito a aquisição de ferramentas, a abordagem extrativa possui o problema de aquisição de ferramentas proprietárias para o gerenciamento da linha de produtos. Já a abordagem incremental não requer o uso de ferramentas específicas.

O oferecimento de treinamentos deve ser suficiente para reduzir a curva de aprendizado das duas abordagens.

Em relação aos resultados esperados, especificamente em relação à produtividade e à escalabilidade, a abordagem extrativa traz vantagens quando comparada com a abordagem incremental, que requer um trabalho artesanal de codificação individual do porte para cada versão do jogo. Em relação à legibilidade as duas abordagens têm resultado satisfatório.

A Tabela 3.3 apresenta as abordagens de porte e seus respectivos resultados com relação aos requisitos especificados na seção 2.3, segundo nossa análise.

Abordagem	Requisitos Técnicos		Requisitos de Implantação		Resultados Esperados		
	Tamanho da aplicação	Alocação de Memória	Aquisição de Ferramentas	Curva de Aprendizado	Produtividade	Escalabilidade	Legibilidade
Incremental	Ruim	Ruim	Bom	Bom	Ruim	Ruim	Bom
Extrativa utilizando linha de produtos e aspectos	Ruim	Não Conclusivo	Ruim	Bom	Bom	Bom	Bom

Tabela 3.3: Abordagens de Porte.

4. Processo da Meantime

A Meantime Mobile Creations é uma empresa de desenvolvimento de jogos para dispositivos móveis. Como todas as empresas desse ramo, a Meantime enfrenta o problema do porte de suas aplicações para a diversidade de dispositivos móveis existentes no mercado.

Inicialmente, a Meantime realizava o porte de suas aplicações utilizando a abordagem incremental, descrita na seção 3.1, onde a aplicação é desenvolvida para um dispositivo e depois o seu código é replicado e adaptado para os demais dispositivos, o que traz problemas como replicação de código, difícil manutenção, uma vez que um erro encontrado após o porte teria que ser corrigido em cada uma das versões criadas, além de ser uma abordagem artesanal, já que cada versão do jogo é produzida individualmente, impossibilitando que seja aplicada em larga escala.

A partir da experiência adquirida do porte de suas aplicações, a Meantime desenvolveu uma plataforma – MG2P (*Meantime Game Porting Platform*) – que dá suporte ao porte massivo de jogos móveis [Camara et al., 2006].

A plataforma MG2P é atualmente utilizada pela Meantime e seu funcionamento será detalhado neste capítulo.

4.1. MG2P

O MG2P é uma plataforma desenvolvida pela Meantime, que inclui um conjunto de técnicas, ferramentas e artefatos para prover generalização e, acima de tudo, escalabilidade para o processo de desenvolvimento e porte de jogos móveis. Na medida em que a empresa ganhou mais experiência, foi possível desenvolver esta solução que, no caso particular da Meantime, tornou mais eficiente o processo de porte.

A utilização do MG2P se baseia em três pilares: uma base de dados do domínio de dispositivos móveis, uma arquitetura base – MBA (*Meantime Basic Architecture*) e um sistema de *build* – MBS (*Meantime Build System*). A seguir, detalharemos cada um desses pilares.

4.1.1. Base de Dados do Domínio de Dispositivos Móveis

A partir da experiência adquirida em suas atividades prévias de porte e analisando as variações e similaridades dos dispositivos, a Meantime adotou uma abordagem de abstração da grande quantidade de dispositivos existentes em famílias de dispositivos que possuem características em comum. Estas famílias não são necessariamente iguais às famílias especificadas pelos fabricantes. A categorização criada pela Meantime é baseada em critérios relevantes à atividade do porte.

As principais variações foram identificadas e classificadas da seguinte forma:

- Variações específicas dos dispositivos: diferenças dos próprios dispositivos, com tamanho de tela, *input* de teclado, etc.;
- Variações de características de jogo: presença de APIs específicas do jogo;

- *Known issues*: *bugs* gerais encontrados em mais de um dispositivo;
- Variações gerais: suporte a várias línguas e fontes gráficas;
- Variações de características: presença de *features* próprias de jogo.

Depois de identificadas e classificadas, estas variações foram mapeadas, de forma que possibilitou a agregação da maior parte dos dispositivos de cada fabricante em famílias, onde cada uma destas é uma combinação de características semelhantes. Para cada família foi especificado um dispositivo para ser o representante. Geralmente esse representante é o dispositivo que possui maior restrição de memória e processamento e maior número de *known issues*. Dessa forma, quando o porte é feito para uma família, tendo como alvo este dispositivo representante, pode-se assumir que, se o jogo funciona neste dispositivo com mais restrições, provavelmente funcionará nos demais.

Essa abordagem propõe a existência de um único código fonte base compartilhado por todas as diferentes versões do jogo. As variações foram mapeadas em diretivas de pré-processamento, de forma que o código relativo a todas as variações de todas as famílias encontra-se em um único código base, sendo separados, apenas, por estas diretivas de pré-processamento. A tabela 4.1 mostra um exemplo de diretivas de pré-processamento relativas a variações de tamanho da tela e uso de uma API específica. A Tabela 4.2 lista algumas diretivas de pré-processamento utilizadas para definir uma família de dispositivos fabricados pela Nokia.

Categoria	Sub-categoria	Variação	Diretiva
Específica do dispositivo	Tamanho de tela	128x117	device_screen_128x117
		128x128	device_screen_128x128
		130x130	device_sreen_130x130
		128x142	device_screen_128x142
		128x149	device_sreen_128x149
		...	...
Características de jogo	Uso de API de Tiled Layer	Meantime API	tiledlayer_api_meantime
		MIDP 2.0 API	tiledlayer_api_midp2
		Siemens Game API	tiledlayer_api_siemens

Tabela 4.1: Diretivas de pré-processamento.

ID da família	Diretivas utilizadas
NOK1	device_screen_128x128
	device_keys_nokia
	device_graphics_canvas_nokiaui
	device_graphics_transform_nokiaui
	game_sprite_api_meantime
	device_sound_api_nokia

Tabela 4.2: Diretivas de pré-processamento para a família NOK1.

O mapeamento das variações do domínio em diretivas de pré-processamento é feito em um arquivo Excel. Inicialmente o uso deste tipo de arquivo era satisfatório, pois existia

uma quantidade menor de variações. Atualmente as variações de mais de 500 dispositivos agrupados em cerca de 49 famílias são mapeadas neste arquivo, o que torna difícil a visualização do mapeamento como um todo, assim como sua manutenção.

4.1.2. Meantime Basic Architecture (MBA)

O resultado da implementação do domínio é a arquitetura básica (MBA). Esta arquitetura engloba as variações básicas relativas ao porte, que são identificadas através das diretivas de pré-processamento nas quais foram mapeadas.

A idéia básica da arquitetura é ser um guia para os desenvolvedores e ajudar a produzir um código que siga o padrão adotado pela empresa. Todos os *design patterns*, classes e recomendações vêm de experiências do passado adquiridas pelos desenvolvedores. A arquitetura foi criada para acelerar o desenvolvimento inicial, facilitar o porte e evitar erros comuns durante as fases de *design* e desenvolvimento.

O desenvolvedor inicia o desenvolvimento do jogo a partir de uma versão estável da arquitetura. Cria o código base do jogo, que será desenvolvido de forma a abranger as variações específicas do jogo de todos os dispositivos. Se uma nova variação (que não está mapeada) é identificada durante o desenvolvimento do jogo, esta deve ser analisada e, se for o caso, incorporada a base de dados do domínio como uma nova diretiva de pré-processamento e a MBA deve ser atualizada.

A arquitetura acelera o desenvolvimento, uma vez que o desenvolvedor pode focar apenas nas características do jogo, deixando para a arquitetura a responsabilidade de tratar das demais variações.

Por outro lado, a utilização de código pré-processado dificulta a legibilidade do código fonte, uma vez que existe uma grande quantidade de código comentada relativa às mais diversas variações. Além disso, a solução de uma variação de porte encontra-se espalhada nas diversas classes da arquitetura e entrelaçada com o código das outras variações. Por esse motivo torna-se uma tarefa não-trivial reusar uma solução de porte presente na arquitetura.

O fato de a arquitetura ter sido desenvolvida como um único código fonte que contempla todas as variações gerais dos dispositivos alvo da empresa é ainda outro fator dificultador do reuso. Esta solução foi adotada devido às restrições de tamanho de código fonte de uma aplicação para dispositivos móveis, que impede a utilização em larga escala de recursos de orientação a objeto.

Estes fatores dificultam não só a manutenção da arquitetura, na medida em que a correção de um *bug* ou uma melhoria pode acarretar mudanças em vários trechos de código, como também o reuso das soluções de porte em um framework externo, por exemplo. Neste caso seria necessário encontrar os trechos de código relativos a esta solução e copiá-los e colá-los no destino, o que não é uma solução ideal.

4.1.3. Meantime Build System (MBS)

O MBS é um sistema de *build*, baseado no Ant [Ant, 2007] e Antenna, desenvolvido pela Meantime com o objetivo de possibilitar a compilação do código e o empacotamento dos recursos de um jogo durante a fase de *deployment*.

Para cada família do domínio é criado um arquivo de propriedades que lista todas as diretivas de pré-processamento relativas a esta família, assim como o caminho dos recursos que deverão ser usados para gerar o produto final. O MBS utiliza as diretivas de pré-processamento especificadas no arquivo de propriedades para pré-processar o código base. Na fase de pré-processamento as partes do código pertencentes a uma família são selecionadas e as demais são comentadas para que o código seja compilado.

Para empacotar a aplicação, o MBS utiliza as informações dos recursos contidas no arquivo de propriedades para selecioná-los corretamente para uma determinada família.

Os arquivos de propriedades são escritos de forma a conter o mínimo de informações específicas de um jogo possível, para que possam ser reusados por outras aplicações com poucas mudanças. Por outro lado, a composição desses arquivos é feita de forma manual e não existem regras explicitas para tal atividade. Não existe nenhum relacionamento entre a composição desses arquivos e o mapeamento do domínio, o que pode levar a problemas de inconsistência de informações, já que a documentação e a manutenção são feitas diretamente nos artefatos.

5. Ferramenta para Linha de Produtos (FLiP)

O FLiP – Ferramenta para Linha de Produtos – é uma ferramenta que tem como objetivo o apoio ao processo de desenvolvimento de linhas de produtos para jogos móveis, a partir da extração de características de produtos existentes, e a automatização da instanciação dos produtos gerados a partir da linha de produtos, buscando facilitar o porte deste tipo de aplicação para os diversos tipos de dispositivos presentes no mercado. A implementação do FLiP se baseia na abordagem descrita na seção 3.3.2.

O FLiP foi desenvolvido como uma extensão do Eclipse [Eclipse, 2007], um *framework open source* para construção de programas. Originalmente o Eclipse foi desenvolvido como uma IDE para desenvolvimento Java, porém, atualmente, é utilizado para diversos propósitos, devido à facilidade com que pode ser estendido, através da adição de novos *plugins*.

O FLiP foi implementado como um conjunto de *plugins* para o Eclipse, de forma que novos *plugins* podem ser facilmente adicionados estendendo-o. Detalhes da extensibilidade do FLiP serão dados adiante.

Neste capítulo faremos uma descrição dos módulos que compõem o FLiP, suas respectivas responsabilidades e funcionalidades.

5.1. Módulos e Funcionalidades

O FLiP é dividido em três módulo: FLiPEx, FLiPG e FLiPC. A Figura 5.1 mostra os módulos do FLiP no cenário da extração de uma linha de produtos a partir de um produto existente.

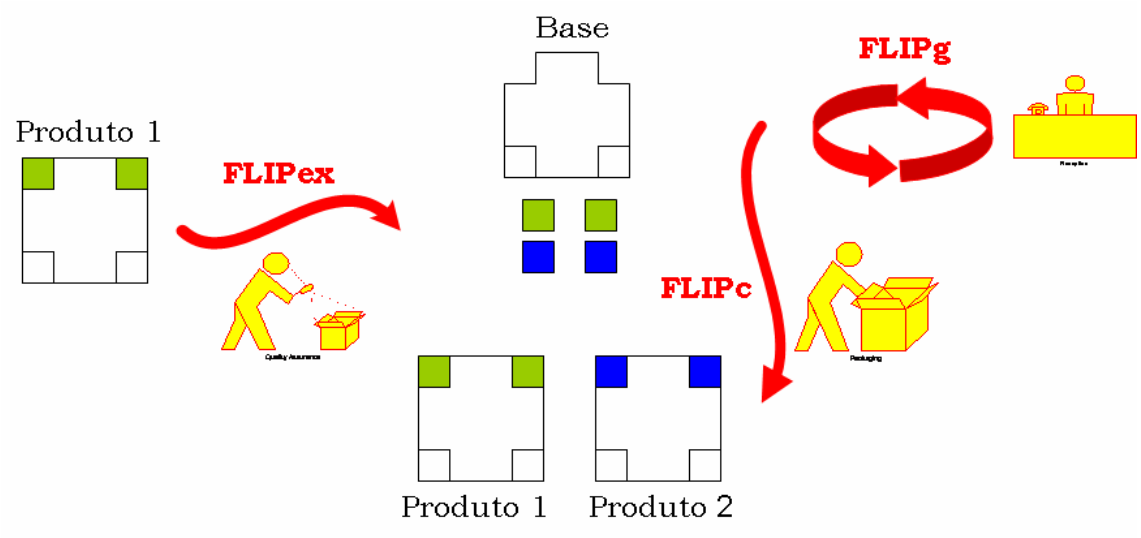

Figura 5.1: Módulos do FLiP.

A seguir descreveremos os três módulos da ferramenta enfatizando suas responsabilidades e funcionalidades.

5.1.1. FLiPEx

O FLiPEx é o módulo responsável pelas funcionalidades relativas às transformações de código que serão executadas através do FLiP. Uma transformação de código é dividida em três componentes:

- Técnica: descreve qual o tipo de destino será dado ao código selecionado no arquivo de origem. Hoje existem duas técnicas implementadas no FLiP: uma que leva o código selecionado na classe Java para um aspecto e outra que transforma o código Java em código pré-processado.
- Extrator: são responsáveis por executar as refatorações, removendo o código do arquivo origem, executando as transformações e transferindo-o para o arquivo destino. Extratores devem estar associados a alguma técnica.
- Validador: são responsáveis por verificar se um extrator pode ou não ser utilizado para extrair o código selecionado, checando se pré-condições de um extrator são satisfeitas. Um validador deve estar associado a um extrator, que por sua vez, pode possuir um ou vários validadores.

O FLiPEx foi implementado como um *plugin* do Eclipse de forma a facilitar a adição de novos componente.Desta forma são disponibilizados pontos de extensão para que novas técnicas, extratores e validadores possam ser acoplados.

5.1.1.1. Funcionalidades:

O FLiPEx compreende as funcionalidades relativas às transformações de código que serão executadas pelo usuário através do FLiP. Como principais funcionalidades do FLiPEx temos:

- Extrações de código Java ou pré-processado para aspectos: consiste na seleção de um trecho de código e posterior extração deste para um, ou mais, aspectos, utilizando a linguagem AspectJ. As extrações implementadas baseiam-se no estudo de variações existentes em jogos móveis [Alves et al., 2006], no qual foram realizadas extrações do código das variações de dois jogos da Meantime, BestLap e Juggling, para aspectos. As extrações implementadas são:
    - Extração de constantes e atributos;
    - Extração de super classe;
    - Extração de interface;
    - Extração de método;
    - Extração de membros da classe;
    - Extração de bloco estático;
    - Extração de declaração de *import*;
    - Extração de código utilizando *after-execution*;
    - Extração de código utilizando *around-execution*;
    - Extração de código utilizando *before-execution;*

- Extração de código utilizando *after-cal;l*
- Extração de código utilizando *before-call;*
- Extração de código utilizando *after-se;t*
- Extração de código utilizando *before-se;t*
- Extração de código utilizando *after-initialization;*
- Extração de código utilizando *before-initialization;*
- Extração de *if* envolvendo todo o corpo do método.

Os validadores que checam as pré-condições necessárias para realizar estas extrações se baseiam nas leis de transformação de programas AspectJ enunciados por Cole [Cole, 2005]. A Figura 5.2 mostra a tela de seleção de técnica e extrator que serão utilizados para realizar uma extração utilizando o FLiPEx.

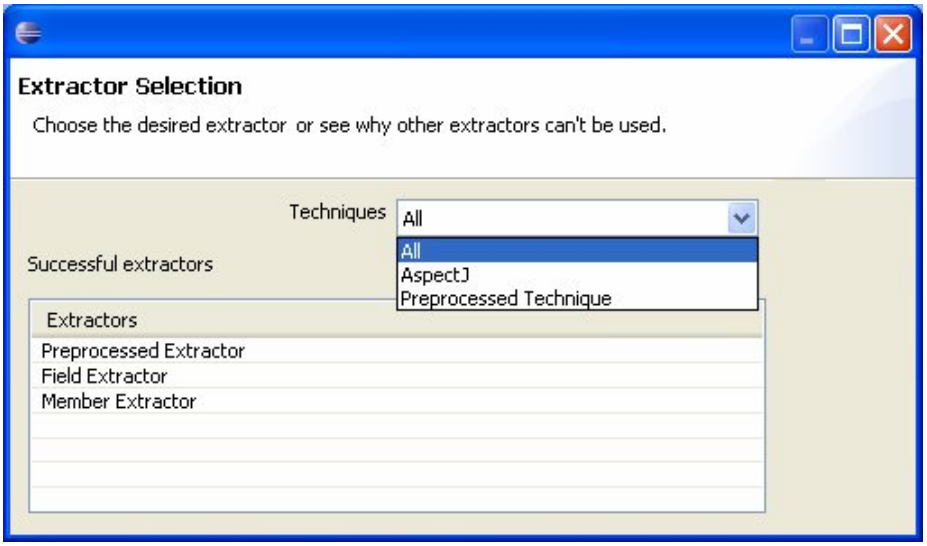

Figura 5.2: Seleção de técnica e extrator.

A Figura 5.3 mostra a lista de extratores que podem ser utilizados para realizar a refatoração do trecho de código selecionado e a lista de extratores que falharam, assim como o motivo da falha.

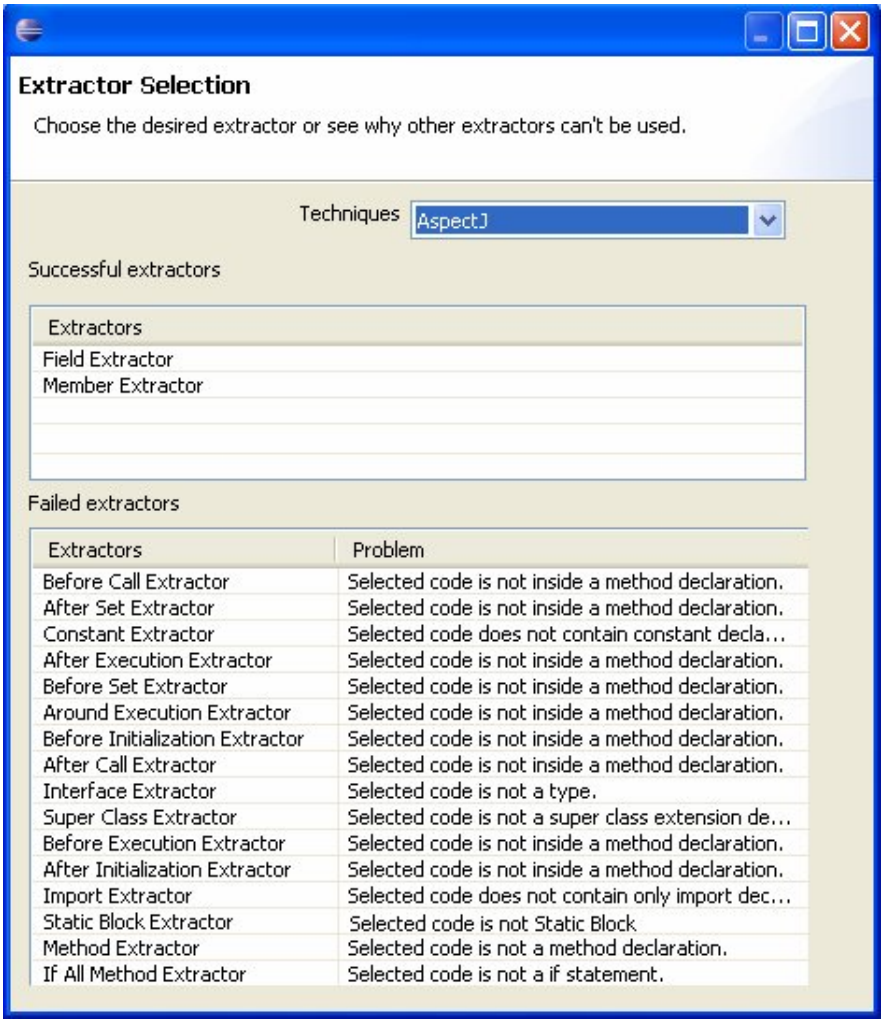

Figura 5.3: Lista de extratores disponíveis.

- Extração de código Java para código pré-processado: é uma funcionalidade complementar à técnica de extração de código Java ou código pré-processado para aspectos. Percebeu-se a necessidade de utilizar uma outra técnica em conjunto com a técnica de extração para aspectos, para contemplar os casos onde seria necessário realizar extrações que programação orientada a aspectos não contempla.
- Extração *batch* de código pré-processado para aspectos: através desta funcionalidade é possível selecionar um projeto, ou uma classe Java, que contenha código pré-processado e solicitar que o FLiP busque todo o código por diretivas de pré-processamento e realize a seleção automáticas dos trechos de código pré-processados para realizar a extração.

Algumas funcionalidades secundárias foram implementadas buscando facilitar o uso da ferramenta. Entre elas podemos citar:

- Suporte a clonagem de extrações: permite a associação de um trecho de código a um conjunto de *features* clones. São consideradas *features* clones aquelas que são alternativas entre si, isto é, são *features* alternativas de uma mesma *feature* pai. A Figura 5.4 mostra um conjunto de *features* alternativas (128x128,

128x117, 130x130) que possuem a mesma *feature* pai (Screen Size). Estas *features* estão relacionadas a possíveis tamanhos de telas de dispositivos móveis.

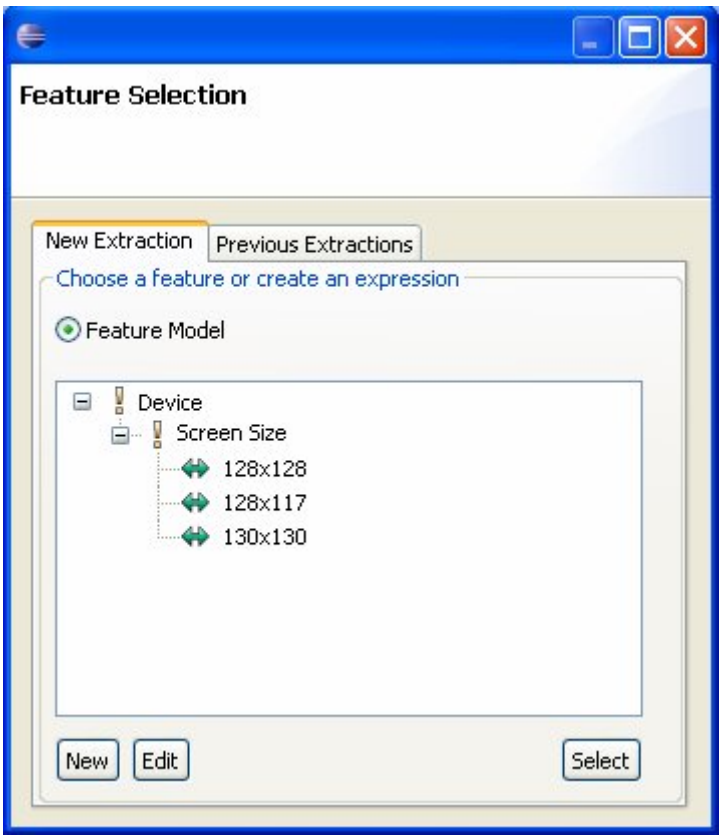

Figura 5.4: *Feature* alternativas.

A Figura 3.5 mostra a tela do FLiP onde o usuário tem a opção de selecionar uma *feature* ou uma lista de clones.

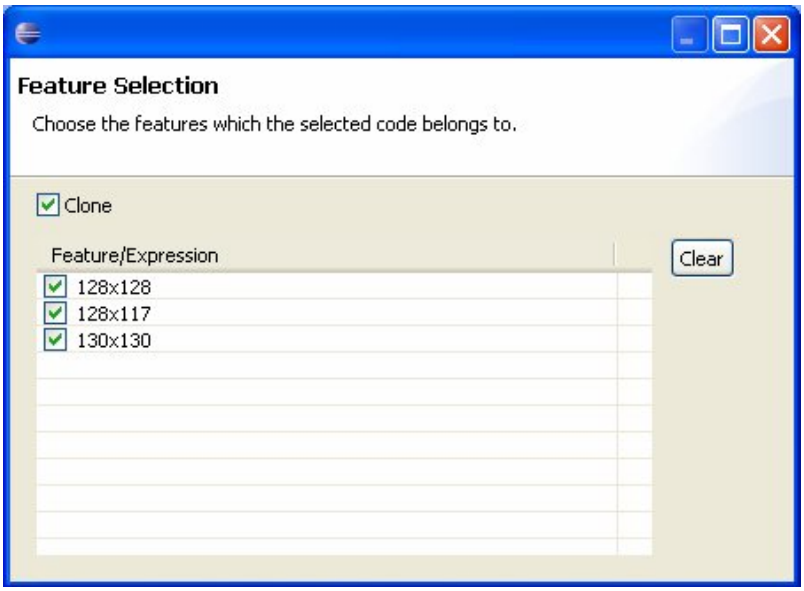

Figura 5.5: Lista de *features*.

- Suporte a várias extrações utilizando a mesma configuração: tem como objetivo tornar mais ágil o processo de extrair diversos trechos de código para um mesmo conjunto de pares de *features* e aspectos. Quando um trecho de código é extraído, o usuário tem a opção de associá-lo a uma configuração que já tenha sido realizada, poupando-o de selecionar as *fetures* e os respectivos aspectos a cada nova extração. A Figura 3.6 mostra a tela do FLiP onde o usuário pode optar por utilizar uma configuração previamente realizada.

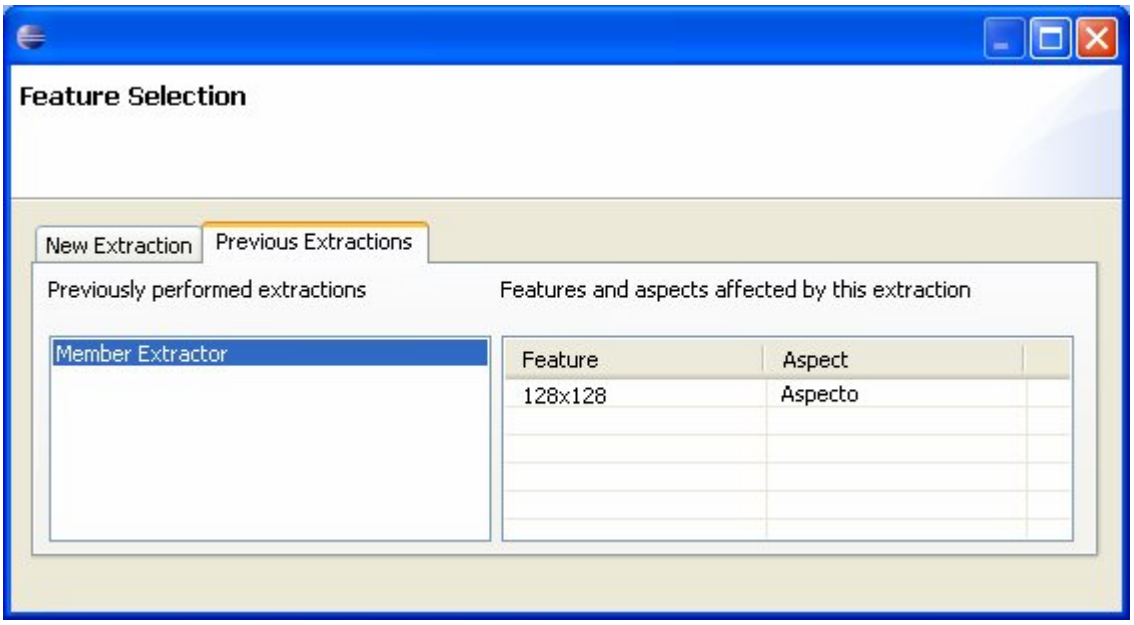

Figura 5.6: Seleção de configuração previamente utilizada.

5.1.2. FLiPG

O FLiPG é o módulo responsável pelas funcionalidades de criação e gerenciamento do *feature model* e manutenção dos artefato associados às *features*. Não faz parte do escopo do FLiP implementar ferramentas de *feature model* e *configuration knowledge*. Assim foi utilizada uma ferramenta externa, o Pure::Variants [Pure::Variants, 2007], uma ferramenta de suporte à criação e gerenciamento de linhas de produtos, para armazenar as *features* que serão criadas e utilizadas pelo FLiP e os artefatos criados pelos usuários, assim como as associações entre os artefatos e as *features*.

5.1.2.1. Funcionalidades:

O FLiPG interage com a ferramenta de *feature model* provendo ao usuário as funcionalidades necessárias para a criação e gerenciamento de *features* e interage com a ferramenta de *configuration knowledge* provendo funcionalidades de associação dos artefatos criados a partir das extrações às *features*. Entre estas funcionalidades podemos citar:

- Criar e atualizar *feature*: o usuário pode criar uma *feature* e adicioná-la ao *feature model* ou atualizar uma *feature* no momento em que está realizando uma extração. A Figura 5.7 mostra a tela de criação de nova *feature* através do FLiPG.

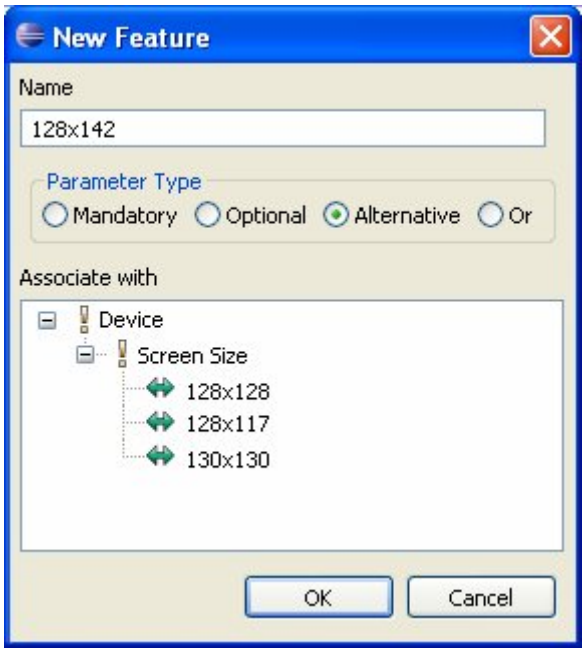

Figura 5.7: Tela de criação de nova *feature*.

- Associar artefato a uma *feature*: ao criar um artefato a partir da extração de um trecho de código o usuário deve associar este artefato a uma *feature*, indicando, dessa maneira, que este artefato será incluído na montagem de um produto caso esta *feature* seja selecionada na instanciação do produto.
- Associar artefato a uma expressão de *features*: o usuário pode associar um artefato gerado através de uma extração a uma expressão de *features*. Essa expressão indica quais *features* devem, ou não, estar selecionadas para que o artefato seja incluído na montagem de um produto. Esta associação é armazenada pelo *configuration knowledge*.
- Selecionar instância da linha de produtos: o usuário poderá selecionar uma instância da linha de produtos, selecionando *features* que irão compor o seu produto. Essa instância representa um produto que poderá ser gerado utilizando o FLiPC, conforme será explicado mais adiante. A Figura 5.8 mostra a seleção de uma instância da linha de produtos.

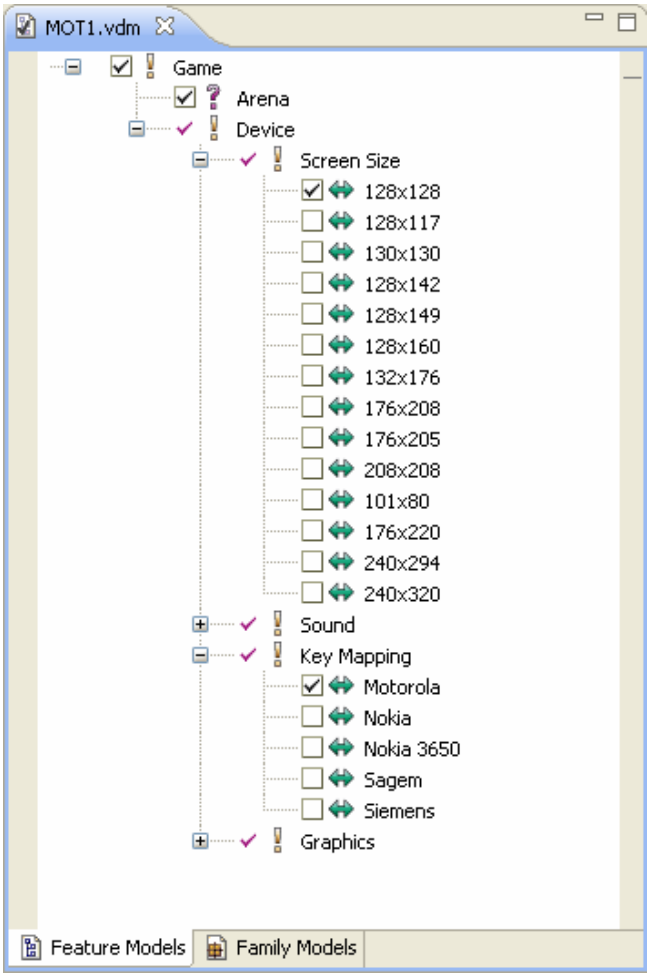

Figura 5.8: Instanciação de produto.

- Gerar especificação de *builds*: esta funcionalidade permite ao usuário gerar especificações de *builds* para todas as instâncias da linha de produtos, através da geração automática dos arquivos de propriedades que servem como entrada para o sistema de *build*. Esta geração automática é feita a partir da inclusão dos aspectos, associados às *features* selecionadas em uma instância de produto, no arquivo de propriedades.

Assim como o FLiPEx, o FLiPG também foi implementado em forma de *plugin* para o Eclipse e disponibiliza pontos de extensão para que seja possível acoplar outras ferramentas de *feature model* e *configuration knowledge* de acordo com a preferência do desenvolvedor.

5.1.3. FLiPC

O FLiPC é o módulo responsável pela composição dos artefatos de acordo com a instanciação dos produtos. Para que o produto final seja construído conforme desejado, é necessário acoplar uma ferramenta de *build* ao FLiPC de acordo com a necessidade do usuário. Na atual implementação do FLiP a ferramenta de *build* utilizada é o MBS (*Meantime Build System*).

Assim como o FLiPEx e o FLiPG, o FLiPC foi implementado como um *plugin* do Eclipse e disponibiliza ponto de extensão para que a ferramenta de *build* desejada possa ser acoplada.

O FLiPC possui uma única funcionalidade que é a execução da geração de *builds*. Após selecionar uma instância de produto da linha de produtos, o usuário pode executar a geração de *builds* que fará uma chamada a ferramenta de *build* que esteja acoplada ao FLiPC para, dessa forma, gerar o produto final da instanciação.

5.2. Extensibilidade

Conforme dito anteriormente, o FLiP foi criado com o objetivo de ser extensível. Para tanto, foi desenvolvido na forma de um conjunto de *plugins* para o Eclipse, fornecendo pontos de extensão para que ferramentas e componentes adicionais sejam facilmente acoplados. Os pontos de extensão disponibilizados pelo FLiP são:

5.2.1. Pontos de extensão do FLiPEx

- Técnicas: ponto de extensão para adição de novas técnicas de extração;
- Extratores: ponto de extensão para adição de novos extratores às técnicas;
- Validadores: ponto de extensão para adição de novos validadores para os extratores.

5.2.2. Pontos de extensão do FLiPG

- Ferramenta de *feature model*: ponto de extensão para o acoplamento de uma ferramenta de *feature model*;
- Ferramenta de *configuration knowledge*: ponto de extensão para o acoplamento de uma ferramenta de *configuration knowledge*.

5.2.3. Pontos de extensão do FLiPC

- Ferramenta de *build*: ponto de extensão para o acoplamento de uma ferramenta de *build*.

5.2.4. Pontos de extensão da GUI

- Extratores: ponto de extensão que permite ao desenvolvedor adicionar páginas, no fluxo de telas do wizard principal do FLiP, relacionadas a um extrator;

- *Constraint Dialog*: ponto de extensão que permite o acoplamento de um *contraint dialog*, componente de GUI responsável por criar restrições entre as *features* e armazená-las no *feature model*. Este componente é mostrado no momento em que o usuário requisita a edição de uma *feature* e clica o botão “editar restrições” que só estará disponível caso haja algum *dialog* implementando este ponto de extensão.

6. Diagnóstico e Sugestões

6.1. Processo de diagnóstico

O processo de diagnóstico deste estudo foi realizado com base no conhecimento adquirido da ferramenta, nos onze meses em que trabalhei no desenvolvimento do FLiP e nas entrevistas que realizei com a diretoria da Meantime e com membros da equipe de desenvolvimento, incluindo o líder técnico. Tais entrevistas tiveram o objetivo de definir os requisitos relevantes para a empresa que uma solução de porte deve contemplar, além de adquirir um conhecimento mais profundo do processo de desenvolvimento da Meantime, para que fosse possível sugerir um roteiro de implantação do FLiP.

6.2. Diagnóstico e Sugestões

Nesta seção faremos um diagnóstico do FLiP com relação aos requisitos necessários a uma solução de porte descritos na seção 2.3. Em alguns casos, faremos sugestões de mudanças para que o FLiP se torne mais adequado à Meantime.

6.2.1. Requisitos Técnicos

6.2.1.1. Tamanho da Aplicação

A primeira restrição do domínio de jogos móveis que abordaremos diz respeito ao tamanho máximo que uma aplicação pode atingir.

A inserção de aspectos no código de jogos móveis gera um *overhead* no tamanho do código gerado pelos compiladores AspectJ. Mesmo com o uso de otimizadores de código [ProGuard, 2007; RetroGuard, 2007] – aplicativos que possuem passos de redução de código, onde o tamanho do *bytecode* das aplicações é reduzido consideravelmente, através de mudanças no código tais como remoção de classes e atributos não utilizados, diminuição dos nomes das classes, atributos, métodos, etc. - o overhead gerado é proibitivo no contexto de jogos móveis, onde o espaço disponível em memória de armazenamento e de execução é consideravelmente restrito.

Um estudo realizado [Lopes, 2007] identifica o overhead na geração de código dos compiladores de AspectJ e descreve a implementação de otimizações em um compilador. Neste estudo foi realizada uma comparação do tamanho dos *builds* (arquivos .jar) de duas versões de um jogo desenvolvido em Java ME: a versão original, que utiliza compilação condicional e uma versão contendo grande parte das variações extraídas para aspectos.

Na geração dos *builds* da versão contendo aspectos, foram utilizados os compiladores ajc [ajc, 2007] e abc [Avgustinov et al., 2004]. O primeiro é a implementação oficial de AspectJ. É um compilador integrado com o ambiente de desenvolvimento Eclipse através do *plugin* AJDT (*AspectJ Development Tools*, uma extensão do JDT, *Java Development Tools*). O abc é um compilador acadêmico que implementa a linguagem

AspectJ e tem como principais objetivos extensibilidade, que possibilita a adição de novas construções à linguagem, de novas otimizações, etc. e a geração de código eficiente.

A tabela 6.1 apresenta os resultados obtidos do tamanho dos *builds* da versão sem aspectos do jogo, compilada com o abc, visto que este apresentou melhor desempenho que o javac, e os tamanhos dos *builds* da versão que contem aspectos, compilada utilizando o abc e o ajc. Para cada versão foram realizados *builds* para diversas famílias de dispositivos.

Build ID	Original (abc)	Com aspectos (ajc)	Com aspectos (abc)
MOT1	72.512	82.832	81.534
MOT2	84.939	95.258	93.985
MOT3	72.544	87.410	85.956
S40	64.913	74.367	72.950
S40M2	72.563	81.951	80.668
S40M2V3	72.115	81.744	80.526
S60M1	85.190	95.695	94.085
S60M2	84.106	94.770	93.122
SAM1	66.246	73.178	72.024
SAM2	80.251	85.688	84.544
SE02	83.958	92.900	91.719
SE03	72.146	81.610	80.433
SE04	72.132	81.527	80.383
SIEM3	72.854	83.012	81.754
SIEM4	72.078	82.356	81.101
Média	75.236	84.953	83.652
Aumento	0%	12,92%	11,19%

Tabela 6.1: Tamanho dos *builds* da versão original e da versão contendo aspectos.

Como é possível perceber, a diferença média entre a versão original e a versão contendo aspectos compilada com o abc é de 8.416 bytes (aumento de 11%), o que torna inviável a utilização de tais *builds*.

Na segunda etapa do estudo, algumas otimizações foram implementadas no compilador abc, buscando a diminuição do overhead gerado. A Tabela 6.2 apresenta o resultado das otimizações, comparando a versão original, a versão contendo aspectos antes das otimizações e a versão contendo aspectos após as otimizações.

Build ID	Original (abc)	Com aspectos (abc)	Com aspectos (otimizado)
MOT1	72.512	81.534	77.299
MOT2	84.939	93.985	89.738
MOT3	72.544	85.956	80.597
S40	64.913	72.950	68.457
S40M2	72.563	80.668	76.955
S40M2V3	72.115	80.526	76.905
S60M1	85.190	94.085	89.815
S60M2	84.106	93.122	88.775
SAM1	66.246	72.024	68.908
SAM2	80.251	84.544	81.744
SE02	83.958	91.719	88.196
SE03	72.146	80.433	76.504
SE04	72.132	80.383	76.843
SIEM3	72.854	81.754	77.732
SIEM4	72.078	81.101	77.335
Média	75.236	83.652	79.720
Aumento	0%	11,19%	5,96%

Tabela 6.2: Tamanho dos *builds* da versão original, da versão contendo aspectos sem otimizações e da versão contendo aspectos com otimizações.

Pode-se perceber que ainda existe uma diferença entre a versão original e a versão contendo aspectos otimizada, porém as otimizações trouxeram uma diminuição de 48% no overhead em relação a versão contendo aspectos sem otimizações.

O FLiP utiliza o compilador abc, com as otimizações sugeridas em [Lopes, 2007], como compilador de AspectJ, buscando a redução do overhead gerado no código final pela adição de aspectos ao código.

6.2.1.2. Alocação de Memória

A alocação de memória é mais uma restrição do domínio de jogos móveis.

A utilização do FLiP no desenvolvimento das aplicações não deve interferir neste ponto, uma vez que a introdução de aspectos é a única mudança de implementação de código que será feita e, neste caso, a aplicação final é resultante da fusão entre dois núcleos - a base (*core concerns*) implementada em Java e os interesses transversais (*crosscutting concerns*) implementados em AspectJ – sem resultar em qualquer alteração com relação a alocação de objetos na memória.

O processo responsável por combinar esses dois núcleos, é chamado combinação aspectual (*aspect weaving*). Tal processo antecede a compilação e gera um código intermediário na linguagem de componentes (Java) capaz de produzir a operação desejada, ou de permitir a sua realização durante a execução do programa. Dessa forma, as classes referentes ao código do negócio nos sistemas não sofrem qualquer alteração para suportar a programação orientada a aspectos [Winck et al., 2006].

No entanto, nenhum estudo foi efetivamente realizado no sentido de verificar o *footprint* de memória das aplicações originais e após a inserção dos aspectos. Para chegar a

conclusões mais precisas, propomos a realização de um estudo semelhante ao realizado em [Lopes, 2007], onde uma comparação seria feita entre as aplicações originais, utilizando compilação condicional e as aplicações utilizando aspectos.

6.2.2. Requisitos de Implantação

6.2.2.1. Aquisição de Ferramentas

Ao adotar uma nova ferramenta, a empresa precisa adquirir licenças para usá-la. Algumas ferramentas possuem licenças de uso muito caras, tornando impraticável sua adoção por empresas de pequeno ou médio porte.

O FLiP é uma ferramenta de código aberto sob licença LGPL de software livre, isto significa que o FLiP pode ser usado livremente e pode ser utilizando em conjunto com outros softwares livres ou proprietários. Dessa forma, o uso do FLiP não acarretaria custos de licença para a Meantime, porém, conforme dito anteriormente, o FLiP necessita de ferramentas auxiliares para o seu funcionamento pleno. Estas ferramentas são: ferramenta de *feature model*, ferramenta de *configuration knowledge* e ferramenta de *build*.

Na atual implementação do FLiP, a ferramenta de *build* com a qual este encontra-se integrado é o MBS – ferramenta de *build* da Meantime – porém, a ferramenta de *feature model* e *configuration knowledge* utilizadas – o Pure::Variants – é uma ferramenta proprietária. Esta ferramenta foi escolhida após uma análise das ferramentas presentes no mercado e a conclusão de que o Pure::Variants possui todas as funcionalidades necessárias ao FLiP.

Uma solução para esse problema seria o desenvolvimento de uma ferramenta de *feature model* e *configuration knowledge* de código aberto, que contemplasse as funcionalidades necessárias ao funcionamento do FLiP. Ou ainda, a adaptação de uma ferramenta livre existente de forma a complementar suas funcionalidades para as necessidades do FLiP.

Foram realizados estudos da viabilidade de adaptação de duas ferramentas livres o ToolDAy (*Tool for Domain Analysis*) [ToolDAy, 2007] e o GenArch (*Generative Architecture Plugin*) [Cirilo, 2007], porém o esforço necessário para realizar tais adaptações foge ao escopo do projeto de desenvolvimento do FLiP.

6.2.2.2. Adaptação de Tecnologias e Processos Existentes

Quando uma empresa adota uma nova ferramenta, é necessário analisar se esta irá substituir ou trabalhar em conjunto com outras ferramentas já utilizadas pela empresa. Caso a ferramenta vá substituir as existentes, um novo processo de desenvolvimento deverá ser criado. Caso esta ferramenta vá trabalhar em conjunto com as existentes, o processo de desenvolvimento deve ser adaptado e adaptações também deverão ser realizadas para integrar as ferramentas existentes com a nova.

No caso da Meantime, o FLiP passará a atuar em conjunto com algumas das ferramentas já utilizadas. Assim adaptações para a integração devem ser feitas.

Durante o desenvolvimento do FLiP, conforme dito anteriormente, o MBS foi adaptado de forma a gerar os produtos de acordo com as especificações do FLiP. Porém, para a implantação, é necessário, ainda, realizar a migração da planilha de dispositivos da Meantime, para a ferramenta de *feature model* e a adaptação da MBA para o uso de aspectos.

É necessária, também, a adaptação do processo de desenvolvimento como um todo. Para tanto, sugerimos um roteiro de implantação gradativo, que diminui os riscos e impactos. Este roteiro será descrito na seção 6.3.

6.2.2.3. Curva de Aprendizado e Treinamentos

Toda nova ferramenta possui uma curva de aprendizado associada a ela. Para determinar esta curva de aprendizado é necessário levar em consideração a dificuldade de utilização da ferramenta e das tecnologias envolvidas no seu uso e o conhecimento do usuário dessas tecnologias.

O FLiP foi desenvolvido buscando ser totalmente intuitivo para usuários familiarizados com porte de jogos móveis. Além disso, algumas funcionalidades específicas desse domínio foram adicionadas, buscando melhorar a usabilidade.

No que diz respeito às tecnologias envolvidas no FLiP, os desenvolvedores da Meantime já devem estar familiarizados com conceitos de linha de produtos, uma vez que alguns desses conceitos já são utilizados pela empresa, mesmo que informalmente, como *feature model*, instanciação de produtos, etc. Em relação à programação orientada a aspectos, acreditamos que desenvolvedores que utilizam orientação a objetos, não terão dificuldades em assimilar.

O oferecimento de treinamentos da ferramenta de linha de produtos utilizada, programação orientada a aspectos e funcionalidades específicas do FLiP, para a equipe de desenvolvedores da Meantime, deve ser suficiente para acelerar satisfatoriamente a curva de aprendizado.

6.2.3. Resultados Esperados

6.2.3.1. Produtividade

Com relação aos resultados esperado de uma solução de porte, é importante dizer que o FLiP tem como objetivo melhorar a produtividade do porte de jogos móveis, na medida que propõe uma melhor organização do *feature model* da empresa – que atualmente é armazenado em um arquivo Excel de difícil visualização e manutenção – e através da automatização da geração dos arquivos de propriedades, que, atualmente, é feita de forma manual. Após a implantação do FLiP, todo o *feature model* estará mapeado em uma ferramenta de gerenciamento de linha de produtos, com interface amigável, além de prover suporte à manutenção. Com relação aos arquivos de propriedades, a geração automática a partir de regras bem definidas reduzirá os erros, pois atualmente, estes arquivos são gerados manualmente e não existe nenhuma especificação contendo regras que indiquem como gerá-los.

6.2.3.2. Escalabilidade

Outro objetivo do FLiP é escalabilidade do porte, através da automatização da geração dos arquivos de propriedades e do fácil reuso das soluções de porte, codificadas isoladamente em aspectos. Dessa forma, ao adicionar uma nova família ao grupo de famílias alvo do porte é possível reusar as variações já armazenadas em aspectos de outras famílias e, caso seja necessário, deve-se apenas codificar em aspecto, ou em Java e posteriormente extrair para um aspecto, as *features* específicas dessa família e, por fim, criar um modelo de instanciação para ela. Quando o MBS for acionado, automaticamente todos os arquivos de propriedades serão gerados a partir dos modelos de instanciação de produto especificados.

6.2.3.3. Legibilidade do Código

Outro resultado esperado de uma solução de porte é a legibilidade do código. Com relação a esse aspecto, devemos lembrar que a programação orientada a aspectos tem como proposta o isolamento de interesses transversais, que geralmente encontram-se espalhados pelo código e entrelaçados com o código de outros interesses, em arquivos chamados aspectos. Dessa forma é possível fazer um reuso maior desses trechos de códigos, antes espalhados e entrelaçados. No caso da Meantime, as variações, até então isoladas através de diretivas de pré-processamento, estarão codificadas em aspectos, após a implantação do FLiP. Além do fácil reuso que o aspecto provê para essas soluções de porte, contamos também com uma maior legibilidade do código, uma vez que todo o código de variações, anteriormente comentado em diretivas de pré-processamento, estará organizado em aspectos.

Por outro lado, devemos considerar a legibilidade do código por parte de pessoas externas à equipe de desenvolvimento da empresa. Nesse caso, se a pessoa não tiver conhecimento em programação orientada a aspectos, o código pode tornar-se ilegível.

Existem empresas especializadas em realizar o porte de jogos móveis para diversos dispositivos e, em algumas situações, é conveniente para uma empresa de desenvolvimento de jogos móveis contratar os serviços dessas empresas. No caso de a Meantime contratar os serviços de uma empresa especializada para portar uma de suas aplicações, desenvolvida após a adoção do FLiP, a legibilidade do código pode ser um problema, se a empresa não estiver familiarizada com a utilização de aspectos, e dessa forma, não for possível realizar o porte do produto.

Uma solução para este problema seria a criação da ferramenta Anti-FLiPEx, que teria como finalidade desfazer as extrações realizadas utilizando a ferramenta FLiPEx, ou seja, com o Anti-FLiPEx seria possível inserir código em uma classe Java a partir de um aspecto. Detalhes de implementação do Anti-FLiPEx fogem ao escopo deste estudo e por isso não serão descritos.

A implementação do Anti-FLiPEx solucionaria o problema de legibilidade do código por empresas externas, porém acreditamos que, após investir em uma ferramenta de automatização de porte, a Meantime não utilizará serviços de empresas especializadas para portar suas aplicações.

6.3. Roteiro de Implantação

Conforme dito anteriormente, para a adoção do FLiP pela Meantime será necessário realizar adaptações nas ferramentas utilizadas atualmente e no processo de desenvolvimento da Meantime. Identificamos que serão necessárias adaptações no *feature model*, na arquitetura e na ferramenta de *build*. Quanto ao processo serão necessárias adaptações no processo de desenvolvimento de um novo jogo e no processo de porte de um jogo existente.

Considerando que as adaptações da ferramenta de *build* foram feitas ainda no período de desenvolvimento do FLiP, propomos um roteiro de implantação da ferramenta, onde as adaptações do *feature model* e da arquitetura sejam feitas gradativamente, de forma a causar o menor impacto possível na adoção da ferramenta e que as adaptações necessárias do processo sejam mínimas.

6.3.1. Adaptação da Arquitetura e do Feature Model

Inicialmente, um conjunto de variações da arquitetura deve ser selecionado. Como exemplo, podemos citar *debug*, *bugs* específicos de dispositivos e variações de API. Então uma dessas variações é escolhida (preferencialmente a que tiver menos impacto na aplicação) e todos os trechos de código relativos a esta variação, *debug*, por exemplo, serão extraídos para um ou mais aspectos, a critério do desenvolvedor, e estes aspectos serão associados a uma *feature*, que deverá ser criada no *feature model*. Após todos os trechos de código relativos a *debug* terem sido extraído da arquitetura, o mesmo processo deve ser aplicado ao segundo grupo de variações, como,por exemplo, *bugs* específicos de dispositivos. Assim, após aplicar este processo a todos os grupos de variações da arquitetura, teremos a arquitetura e o *feature model* adaptados. A partir daí já será possível adicionar e remover facilmente estas *features* no momento da instanciação do produto e todo esse processo pode ser feito na velocidade que a Meantime julgar conveniente. A Figura 6.1 ilustra o processo de adaptação da arquitetura e de criação do *feature model*.

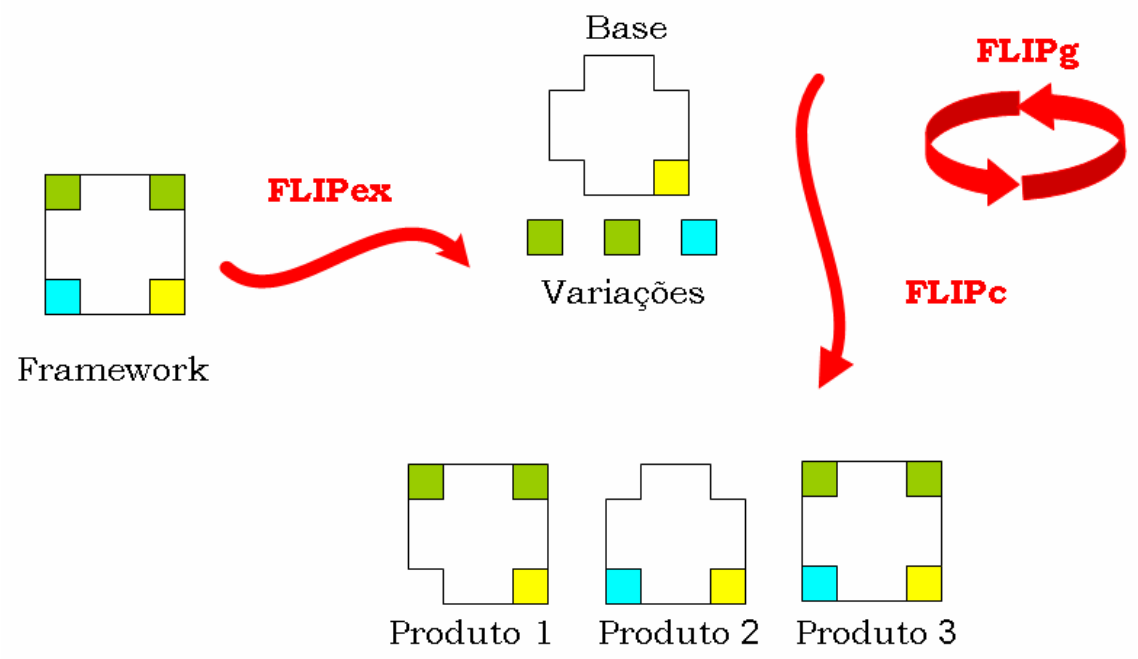

Figura 6.1: Processo de adaptação da arquitetura e de criação do *feature model*.

A partir deste ponto já será possível criar um novo jogo ou portar um jogo existente utilizando o FLiP de fato.

6.3.2. Processo de Criação de um Novo Jogo

No processo de criação de um novo jogo, identificamos dois cenários que requerem procedimentos distintos.

6.3.2.1. Utilizando o Framework da Empresa

Este é o cenário mais comum em uma empresa de jogos. Trata do desenvolvimento de um novo jogo utilizando o framework da própria empresa. Envolve um processo com duas fases: desenvolvimento do jogo para as famílias base e porte para as demais famílias.

Inicialmente, deve-se fazer uma cópia do *feature model* da arquitetura ou cria-se um *subfeature model* contendo apenas as *features* que serão utilizadas no projeto em questão.

Então, dá-se início ao desenvolvimento do jogo para uma família de dispositivos. Na medida em que o código é desenvolvido, as variações vão sendo identificadas e extraídas para aspectos e associadas às *features* do *feature model*. Caso seja necessário, novas *features* vão sendo criadas.

Após o fim do desenvolvimento do jogo para uma família, o desenvolvedor deve criar aspectos com o código relativo às *features* das demais famílias base, e associá-los a *features* existentes no *feature model*.

Ao final da fase de desenvolvimento do jogo para as famílias base, inicia-se a fase de porte para as demais famílias. Nesta fase, instâncias de produtos referentes a cada uma das 49 famílias são criadas, a partir da composição do código base, com a combinação entre as *features* extraídas do framework e as *features* específicas desse projeto, criadas na fase anterior.

A Figura 6.2 mostra as duas fases do processo de criação de um novo jogo utilizando o framework da empresa.

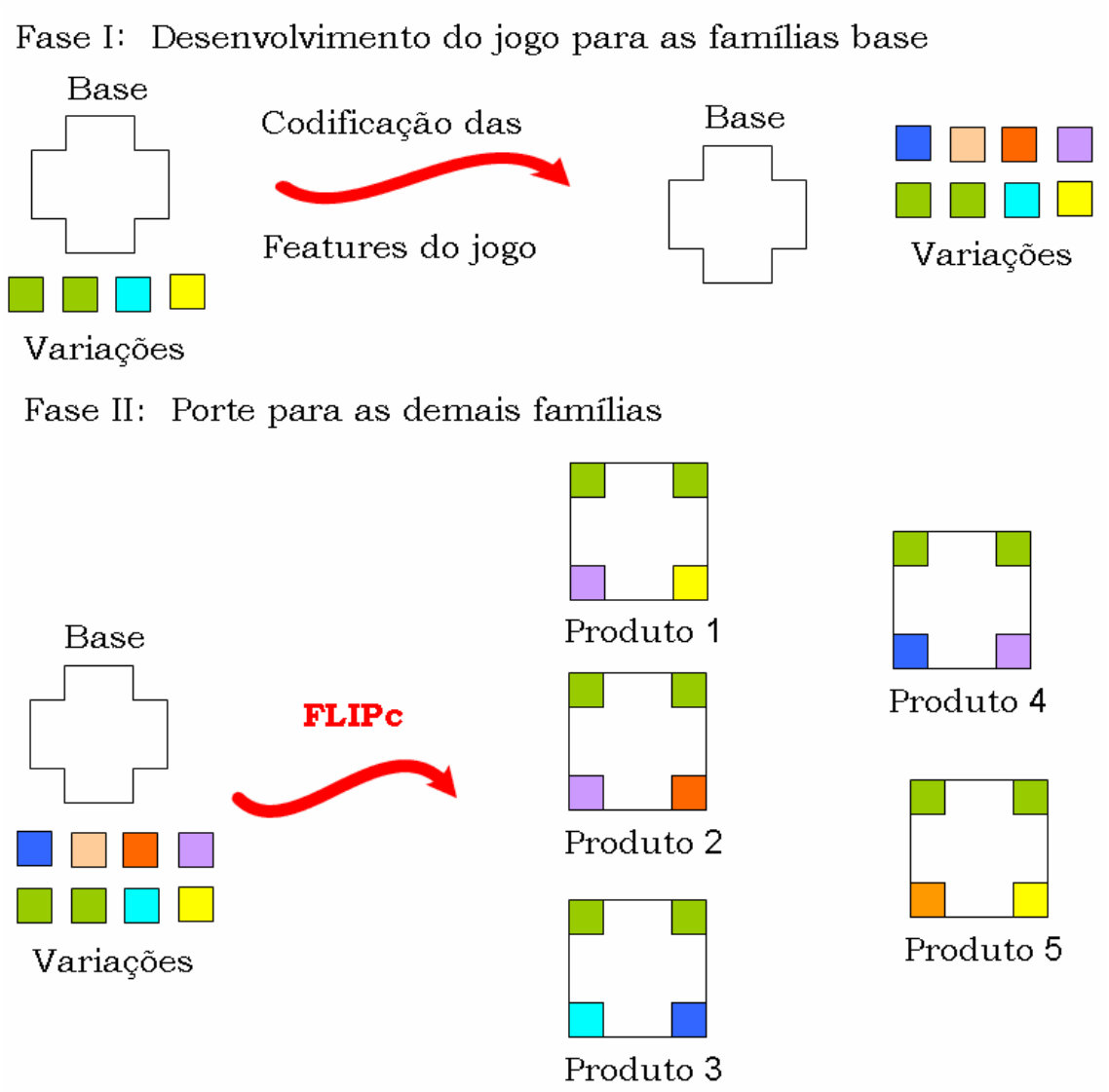

Figura 6.2: Processo de criação de um novo jogo utilizando o framework da Meantime.

6.3.2.2. Utilizando um Framework Externo

Este cenário trata do desenvolvimento de um jogo pela empresa utilizando um framework externo. É um cenário menos comum, porém possível de acontecer, quando a empresa está desenvolvendo um projeto para um cliente que faz esta exigência.

Neste cenário, caso haja interesse por parte da empresa, é possível realizar o processo de extração de *features* no framework que será utilizado pelo projeto. Porém esta atividade

pode custar tempo e dinheiro. Assim, é necessário haver um estudo de viabilidade e custo-benefício para identificar se a atividade deve ser desenvolvida ou não.

Em caso positivo, o mesmo processo descrito na seção 6.3.1 deve ser aplicado ao framework utilizado no projeto.

Após o processo de extração do framework, ou caso esse processo não seja feito, deve-se executar o processo descrito na seção 6.3.2.1.

6.3.3. Processo de Porte de um Jogo Existente

No processo de porte de um jogo existente, identificamos dois cenários que requerem procedimentos distintos.

6.3.3.1. Jogo da Empresa

Este cenário descreve a atividade de porte de um jogo desenvolvido pela empresa antes da implantação do FLiP.

Inicialmente, cria-se um *subfeature model*, a partir do *feature model* original do framework, que será utilizado no projeto em questão.

Em seguida, deve-se extrair as *features* específicas do jogo para aspectos, associá-las a *features*, que devem ser adicionadas ao *feature model*.

O próximo passo é codificar as *features* de jogo da nova família, extraí-las para aspectos, associá-los a *features* que devem ser adicionadas ao *feature model*.

Para finalizar cria-se uma instância do produto a partir da composição do código base com a combinação das *features* extraídas previamente do framework e as *features* codificadas para a nova família.

A Figura 6.3 mostra processo de porte de um jogo desenvolvido utilizando o framework da empresa, porém antes da implantação do FLiP.

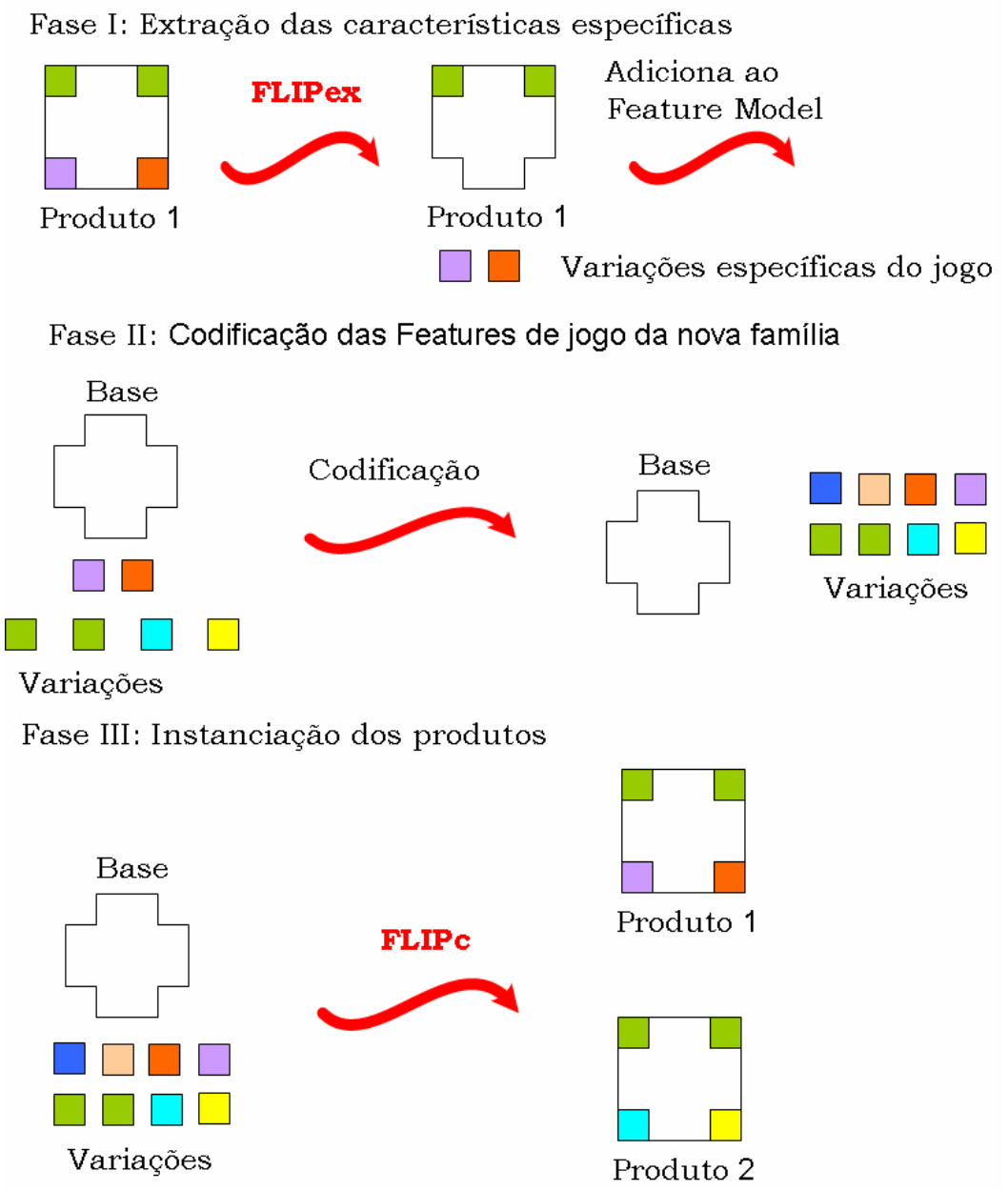

Figura 6.3: Processo de porte de um jogo desenvolvido pela Meantime antes da adoção do FLiP.

6.3.3.2. Jogo de Outras Empresas (Fábrica de Porte)

Este cenário trata da possibilidade da Meantime, após adotar uma ferramenta de automatização de porte poderosa como o FLiP, se tornar uma fábrica de porte, isto é, a empresa receberia jogos desenvolvidos por outras empresas, para portar para outras plataformas.

Para tanto seria aplicado um processo semelhante ao descrito na seção 6.3.2.1, porém seria necessário extrair o código do produto a fim de se obter um código base e não apenas as *features* específicas de jogo, uma vez que o produto não foi desenvolvido utilizando o framework da empresa, como acontecia no caso descrito na seção 6.3.2.1.

7. Conclusão

Neste trabalho apresentamos os requisitos considerados necessários a uma solução de porte para que esta possa ser adota por uma empresa de desenvolvimento de jogos móveis com sucesso. Foi realizada, também, uma análise comparativa entre as principais soluções de porte citadas na literatura com relação a tais requisitos.

A partir dos resultados obtidos foi possível concluir que dentre as soluções de porte de jogos móveis analisadas, a que envolve a extração de linha de produtos a partir dos jogos existentes da empresa, utilizando programação orientada a aspectos para tratar as variações do código, se mostrou a mais eficiente.

Tal solução busca maior produtividade e escalabilidade no processo de porte e legibilidade do código fonte ao isolar as variações, geralmente espalhadas pelo código, em aspectos, o que torna o código mais limpo e fácil de compreender, além de possibilitar um maior reuso das soluções de porte. A organização das variações em um *feature model* também proporciona uma melhor visualização e manutenção da linha de produtos.

Porém, esta solução não atende a todos os requisitos especificados neste estudo, como necessários a uma solução de porte, apresentando deficiências nos aspectos de aquisição de ferramentas - não foi encontrada uma ferramenta de suporte a linha de produtos livre que disponibilizasse todas as funcionalidade necessárias a esta solução- e de geração de *overhead* no tamanho do código final do jogo, devido à inserção de aspectos, que causa um aumento no tamanho do código proibitivo no domínio de jogos móveis, onde o espaço disponível em memória de armazenamento e de execução é consideravelmente restrito.

Após uma análise da solução de porte utilizada por uma empresa de jogos móveis, a Meantime, e da apresentação da solução de porte da ferramenta FLiP, foi possível realizar um diagnóstico da adoção desta ferramenta.

A partir daí, percebemos que o problema de geração de *overhead* no tamanho do código final do jogo pode ser solucionado através da utilização de um compilador AspectJ otimizado.

Quanto à aquisição de ferramentas, sugerimos a implementação de uma ferramenta de suporte à linha de produtos de código aberto que contemple todas as funcionalidades necessárias ao FLiP. Ou ainda que adaptações sejam feitas em uma ferramenta de código aberto existente.

Para diminuir o impacto da adoção do FLiP e das adaptações necessárias ao seu funcionamento, sugerimos um processo de implantação gradativo, onde tais adaptações sejam realizadas de forma incremental até que o ambiente de desenvolvimento esteja completamente adaptado para que, a partir daí, seja possível desenvolver novos jogos ou portar jogos existentes utilizando o FLiP.

Referências Bibliográficas

[ajc, 2007] AspectJ Team, 2007. Página oficial do projeto AspectJ e do compilador ajc.http://www.eclipse.org/aspectj.

[Alves et al., 2007] Alves, V.; Matos Jr, P.; Cole, L.; Vasconcelos, A., Borba ,P. and Ramalho, G. Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming. Transactions on Aspect-Oriented Software Development (TAOSD): Special Issue on Software Evolution, 2007.

[Alves et al., 2006] Alves, V.; Costa Neto, A.; Soares, S.; Santos, G.; Calheiros, F.; Nepomuceno, V.; Pires, D.; Leal, J.and Borba, P. From Conditional Compilation to Aspects: A case study in software product lines migration. 1st Workshop on Aspect-Oriented Product Line Engineering, in conjunction with 5th ACM International Conference on Generative Programming and Component Engineering, October 2006.

[Alves, 2004] Alves, V. Identifying Variations in Mobile Devices. In Young Researchers Workshop at the GPCE'04, Vancouver, Canada, October 2004.

[Alves et al., 2005] Alves, V.; Matos, P.; Cole, L.; Borba, P.; Ramalho, G. Extracting and Evolving Mobile Games Product Lines. In 9th International Software Product Line Conference (SPLC'05), volume 3714 of Lecture Notes in Computer Science, September 2005. Springer.

[Alves et al., 2004] Alves, V.; Matos, P.; Borba, P. An Incremental Aspect-Oriented Product Line Method for J2ME Game Development. In Workshop on Managing Variability Consistently in Design and Code at OOPSLA'04, Vancouver, Canada, October 2004.

[Ant, 2007] Ant, Apache, 2007. http://ant.apache.org.

[Antenna, 2007] Antenna, 2007. http://antenna.sourceforge.net.

[Arthur, 1996] Arthur, W.B. Increasing Returns and the New World of Business. Harvard Business Review, p. 101-109, 1996.

[Avgustinov et al., 2004] Avgustinov, P.; Christensen, A.S.; Hendren, L.; Kuzins, S.; Lhoták, J.; Lhoták, O.; Moor, O. de.; Sereni, D.; Sittampalam, G.and Tibble, J. Building the ABC Aspectj Compiler with Polyglot and Soot. Technical report, The abc Group, October 2004.

[Bignetti, 2002] Bignetti, L.P.O. Processo de Inovação em Empresas Intensivas em Conhecimento. Revista de Administração Contemporânea, vol. 6, nº 3, Setembro/Dezembro de 2002.

[Bignetti, 1999] Bignetti, L.P. Strategic Actions and Innovation Practices in Knowledge-based industries. Montreal, 1999. 494 f. Thesis (Ph.D. in Administration), École des Hautes Études Commerciales.

[Camara et al., 2006] Câmara, T.; Lima, R.; Guimarães, R.; Damasceno, A.; Alves, V.; Macedo, P. and Ramalho, G. Massive Mobile Games Porting: Meantime Study Case. Brazilian Symposium on Computer Games and Digital Entertainment - Computing track, 2006.

[Cardim et al., 2005] Alves, V.; Cardim, I.; Carmo, V.; Sampaio, P.; Damasceno, A.; Borba, P.; Ramalho, G. Comparative Analysis of Porting Strategies in J2ME Games. In 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 2005. IEEE Press.

[Carmo, 2005] Carmo, H.V. do. Técnicas para Construção de Linha de Produtos de Jogos Móveis. Trabalho de Graduação, Centro de Informática, Universidade Federal de Pernambuco, Recife, Agosto 2005.

[Cirilo, 2007] Cirilo, E.; Kulesza , U.; Lucena, C.J. P. de. GenArch: Uma Ferramenta baseada em Modelos para Derivação de Produtos de Software. Anais da Sessão de Ferramentas do SBCARS 2007.

[Clements et al., 2002] Clements, P. and Northrop, L. Software Product Lines. Practices and Patterns. Addison-Wesley, 2002.

[Cole, 2005] Cole, L. Deriving Refactorings for Aspectj. Master's thesis, Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil, February 2005.

[Czarnecki et al., 2000] Czarnecki,K. and Eisenecker, U. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[David, 1986] David, P. Understanding the Economics of QWERTY: the necessity of history. In: PARKER, W. (Ed.). Economic History and the Modern Economist. New York: Basil Blackwell, 1986. p. 30-49.

[Eclipse, 2007] Eclipse, Eclipse.org, 2007. http://www.eclipse.org

[JaTS, 2001] JaTS - Java Transformation System, F. U. of Pernambuco. http://www.cin.ufpe.br/˜jats/, 2001.

[J2ME Polish, 2007] J2ME Polish, 2007. http://www.j2mepolish.org.

[Kiczales, 1997] Kiczales, G.; Lamping, J.; Mendhekar A.; Maeda, C.; Lopes, V.; Loingtier, J.; Irwin, J. Aspect-Oriented Programming. In European Conference on Object Oriented Programming, ECOOP 97, LNCS 1241, pages 220 242, Finland, June 1997.

[Kiczales et al., 2001] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. and Griswold, W.G. An overview of aspectj. ECOOP, 2001.

[Krueger, 2001] Krueger, C. Easing the Transition to Software Mass Customization. In Proceedings of the 4th International Workshop on Software Product-Family Engineering, pages 282–293, 2001.

[Lopes, 2007] Lopes, F.H.C. Otimizando Compiladores de AspecJ Para Java ME. Trabalho de Graduação, Centro de Informática, Universidade Federal de Pernambuco, Recife, Agosto 2007.

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

[Menon, 2005] Menon, H. Portability Analysis in Mobile Gaming Using J2ME. Master's thesis, Lane Department of Computer Science and Electrical Engeneering, Morgantown, West Virginia.

[ProGuard, 2007] ProGuard. ProGuard, Java Class File Shrinker, Optimizer and Obfuscator, 2007. http://proguard.sourceforge.net.

[Pure::Variants, 2007] Pure::Variants, Pure Systems, 2007. http://www.pure-systems.com.

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

[Sampaio et al., 2004] Sampaio, P.; Damasceno, A.; Alves, V.; Ramalho, G.; Borba, P. Porting Games in J2ME: Challenges, Case Study, and Guidelines. In III Brazilian Workshop on Games and Digital Entertainment, October, 2004, Curitiba, Brasil, Agosto 2005.

[Tercek, 2007] Tercek, R. First Decade of Mobile Games. Presentation at GDC Mobile, 2007.

[Tira Wireless, 2007] Tira Wireless. TiraJump, 2007. http://www.tirawireless.com/jump/.

[ToolDAy, 2007] ToolDAy, Rise, 2007. http://www.rise.com.br.

[Vasconcelos, 2005] Vasconcelos, A.T. Ferramenta para Construção de Linha de Produtos no Eclipse. Trabalho de Graduação, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2005.

[Winck et al., 2006] Winck, D.V.; Goetten Jr, V. AspectJ: Programação Orientada a Aspectos com Java. São Paulo: Novatec, 2006.

Assinaturas

Recife, 24 de janeiro de 2008.

_____

Geber Lisboa Ramalho (Orientador)

_____

Paulo Henrique Monteiro Borba (Co-orientador)

_____

Andrea Frazão de Menezes (Aluno/Autor)