UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

ANÁLISE DE TÉCNICAS COMBINADAS PARA A IMPLEMENTAÇÃO DE VARIAÇÕES EM LINHAS DE PRODUTO DE SOFTWARE

TRABALHO DE GRADUAÇÃO

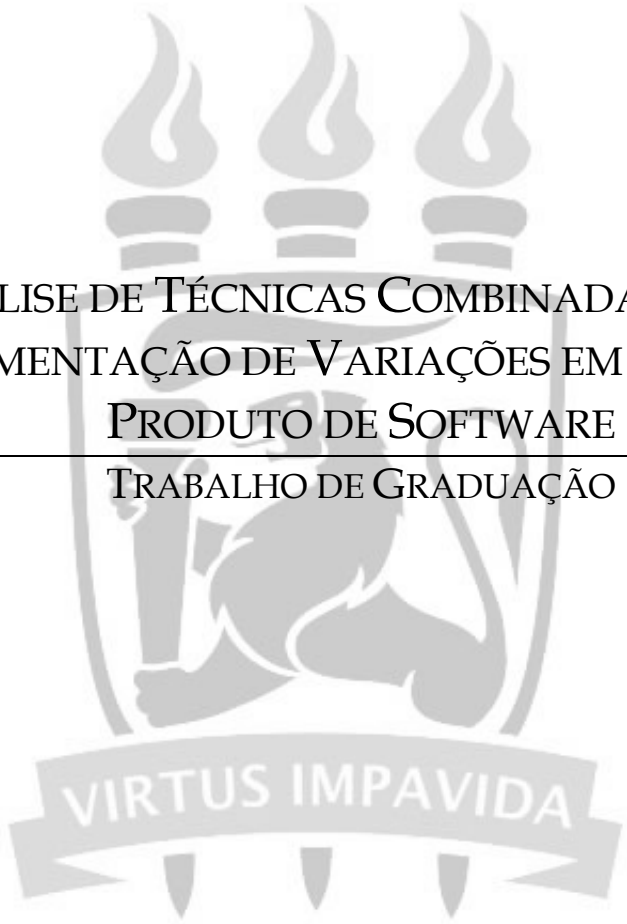
Danilo Cavalcanti Torres (dct@cin.ufpe.br)

Recife, Novembro de 2008.

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

TRABALHO DE GRADUAÇÃO



ANÁLISE DE TÉCNICAS COMBINADAS PARA A IMPLEMENTAÇÃO DE VARIAÇÕES EM LINHAS DE PRODUTO DE SOFTWARE

TRABALHO DE GRADUAÇÃO

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

Aluno: Danilo Cavalcanti Torres (dct@cin.ufpe.br)
Orientador: Paulo Henrique Monteiro Borba (phmb@cin.ufpe.br)

Recife, Novembro de 2008.

Agradecimentos

No final de mais uma difícil e árdua etapa da minha vida, deixo meus agradecimentos para todos aqueles que, diretamente ou indiretamente, contribuíram para a conclusão da minha graduação.

Como não poderia deixar de ser, em primeiro lugar agradeço ao Deus Todo Poderoso, detentor de toda a sabedoria e cujos pensamentos nem o mais sábio dos homens pode discernir.

Agradeço à minha família (especialmente meus pais e meu avô Cleodésio) que em todo tempo me incentivou e lutou para que eu tivesse acesso aos estudos.

Agradeço ao meu orientador, Prof. Paulo Borba pela paciência, respeito e exemplo como mestre, onde desde o primeiro período desta graduação mostrou ser uma pessoa e um profissional honroso.

Não poderia deixar de agradecer também ao Pedro Osandy Alves Matos Júnior, mestre do CIn, que com muita paciência me ajudou bastante no desenvolvimento deste trabalho, mesmo não estando em Recife.

Agradeço também a todos que fazem a família CIn, especialmente aos colegas de turma e de outras turmas com os quais compartilhei alegrias e preocupações.

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

Resumo

Cada vez mais sofisticados e acessíveis, os aparelhos de celular fazem surgir um grande e promissor mercado na indústria de entretenimento digital: os jogos móveis.

Das diversas etapas do desenvolvimento destes jogos, uma chama mais atenção pela sua complexidade: o *porting*. Isso se deve ao fato de que para um jogo poder ser comercializado, o mesmo deve estar disponível para um grande número de aparelhos, atendendo aos diversos tamanhos de tela e de memória, às diversas APIs dos fabricantes e idiomas, que o faz ser muito custoso. Para diminuir o gargalo deste processo, a indústria faz o uso do conceito de Linhas de Produtos de Software, que vem para automatizar esta etapa do desenvolvimento.

Este trabalho se propõe a realizar um estudo comparativo entre o uso de técnicas simples e combinadas em linhas de produtos de software expondo os resultados de acordo com métricas tradicionais de avaliação.

Palavras-chave: J2ME, Linhas de Produtos de Software, Herança, Compilação Condicional, Porting

Abstract

More and more sophisticated and accessible, mobile cell phones make grown a big and promissory market in digital entertainment industry: the mobile games.

On various stages of those games development, one contrast by its complexity: the porting. It's because for a game be commercialized, it must be available to a big number of devices, observing the to the different screen size and memory, the various manufacturer API and language, becoming it so expensive. To reduce the impact of this process, the industry makes use of Software Product Line concept, automating this stage of development.

This work has the goal of make a comparative study between the use of simple techniques and combined techniques in a software product line, showing the results through some traditional evaluate metrics.

Sumário

1. Introdução

Cada vez mais sofisticados e acessíveis, os aparelhos de celular fazem surgir um grande e promissor mercado na indústria de entretenimento digital: os jogos móveis. Projeções mostram que, até 2011, espera-se que sejam movimentados, em todo mundo, aproximadamente US$ 7 bilhões neste mercado [1].

Das diversas etapas do desenvolvimento destes jogos, uma chama mais atenção pela sua complexidade: o *porting*. Esta fase é responsável por adaptar o jogo para diversos tipos de celular, tendo em vista os diversos tamanhos de tela, quantidade de memória, a API implementada pelo fabricante do dispositivo, a capacidade de processamento dos aparelhos e o idioma do jogo. Sendo assim, para cada jogo devem ser geradas várias versões diferentes, podendo ter até mesmo mais de 2000 para um único jogo [2]. Para diminuir o overhead deste processo, a indústria faz o uso do conceito de Linhas de Produtos de Software, que vem para automatizar esta etapa do desenvolvimento.

Linhas de Produto de Software (LPS) são famílias de produtos de software que possuem uma série de requisitos em comum [3]. O uso de LPS se faz bastante comum no desenvolvimento de sistemas que tem de se adaptar a diferentes requisitos de diferentes clientes. Para isso é necessário apenas desenvolver uma LPS e gerenciar as instâncias do software geradas de acordo com as variações necessárias, reusando, dessa forma, componentes comuns a todas as instâncias. Um exemplo prático disto pode ser observado na maioria dos sistemas operacionais utilizados atualmente, onde várias versões são lançadas no mercado, como versões voltadas mais para escritórios, para empresas, para uso pessoal, dentre outros.

1.1. Problema

Especificamente no ambiente móvel, alguns fatores são fundamentais para o bom desempenho de uma LPS, como o tamanho final do *bytecode* e a qualidade do código tendo em vista a manutenção da linha gerada. Sendo assim, a LPS deve ser bem estruturada para que não seja implementada de forma a não trazer grandes benefícios no resultado final.

Algumas técnicas de programação ajudam na implementação das variações da linha, porém, devido a pouca informação e literatura a respeito, alguns programadores acabam por optar por alguma técnica que em determinada situação pode não ser a mais adequada. Este fato possui impacto direto na qualidade e eficácia da LPS.

Alguns critérios, métricas e *guidelines,* existentes em pouco número na literatura, podem auxiliar o programador na hora da decisão de qual técnica utilizar de acordo com cada situação.

1.2. Solução

Tendo em vista as diversas técnicas existentes, iremos analisar a viabilidade de implementarmos LPS usando técnicas combinadas, isto é, fazendo uso de varias técnicas simultaneamente.

Usando a ferramenta desenvolvida durante estudo feito por Pedro Osandy Alves Matos Júnior [4], cujo objetivo é analisar de forma comparativa diversas técnicas isoladamente, poderemos analisar os resultados obtidos pela implementação proposta nesse trabalho. Esta ferramenta, embora ainda sem muitos recursos, irão nos auxiliar a rodar métricas de qualidade e quantidade.

1.3. Objetivo

O objetivo deste estudo é apresentar, de forma comparativa, resultados de métricas de qualidade para duas formas diferentes de se implementar linhas de produtos de software, sendo a primeira utilizando técnicas isoladas (como compilação condicional ou herança) e a segunda utilizando combinações destas técnicas. Dessa forma, iremos colaborar com a literatura atual, ainda limitada neste assunto.

Para isso iremos usar como caso de estudo um jogo desenvolvido para dispositivos móveis, que originalmente utiliza a técnica de compilação condicional para implementação da LPS.

1.4. Estrutura deste Documento

Este documento está estruturado da seguinte forma:

- Capítulo 2 apresenta alguns conceitos importantes para o bom entendimento deste estudo. Alguns conceitos relacionados a Linhas de Produto de Software serão abordados, assim como algumas técnicas de variação utilizadas na indústria.

- Capítulo 3 expõe o nosso caso de estudo, sua arquitetura, seu código fazendo uso de LPS, suas diversas formas de implementação (de acordo com as técnicas escolhidas), além do código modificado para uso de técnicas combinadas

- Capítulo 4 expõe o nosso caso de estudo, sua arquitetura, seu código fazendo uso de LPS, suas diversas formas de implementação (de acordo com as técnicas escolhidas), além do código modificado para uso de técnicas combinadas

- Capítulo 5 conclui este estudo, discutindo em resumo os resultados obtidos, apresentando alguns trabalhos relacionados, pontos de melhoria para este trabalho, assim como possíveis trabalhos futuros.

2. Conceitos

Neste capítulo iremos apresentar algumas informações bastante relevantes para o bom entendimento deste estudo. Este capítulo está dividido da seguinte forma: na Seção 2.1 introduzimos conceitos relacionados a Linhas de Produto de Software e variações; Seção 2.2 complementa a seção anterior, discutindo algumas técnicas de variações que julgamos relevantes para o nosso estudo; finalmente na Seção 2.3 introduzimos alguns conceitos existentes no domínio dos jogos móveis.

2.1. Linhas de Produto de Software

De acordo com [5], Linha de Produto de Software é um conjunto de sistemas que compartilham um mesmo conjunto de requisitos, porém cada um com algumas variações que satisfazem as necessidades específicas de um segmento de mercado em particular.

A abordagem mais promissora de reuso de arquitetura é desenvolver uma arquitetura de linha de produto, que explicitamente conhece as variações e as partes em comum na família de sistemas que constituem a linha de produto [6].

A Figura 2.1 compara o custo acumulado necessário para desenvolver uma LPS com *n* instâncias diferentes, com *n* produtos de software independentes. A linha sólida representa o custo de desenvolvimento dos sistemas independentemente, enquanto a linha hachurada mostra o custo da engenharia da linha de produto. Inicialmente existe um custo para a implementação LPS (*Up-Front Investiment*), custo este não existente para a implementação independente. Este custo representa o tempo necessário para se estabelecer e configurar a linha.

Porém em um ponto do gráfico (*break-even point*) as curvas se encontram, indicando que o custo para ambas as implementações é o mesmo. Porém logo em seguida a linha tende a ter uma crescente vantagem.

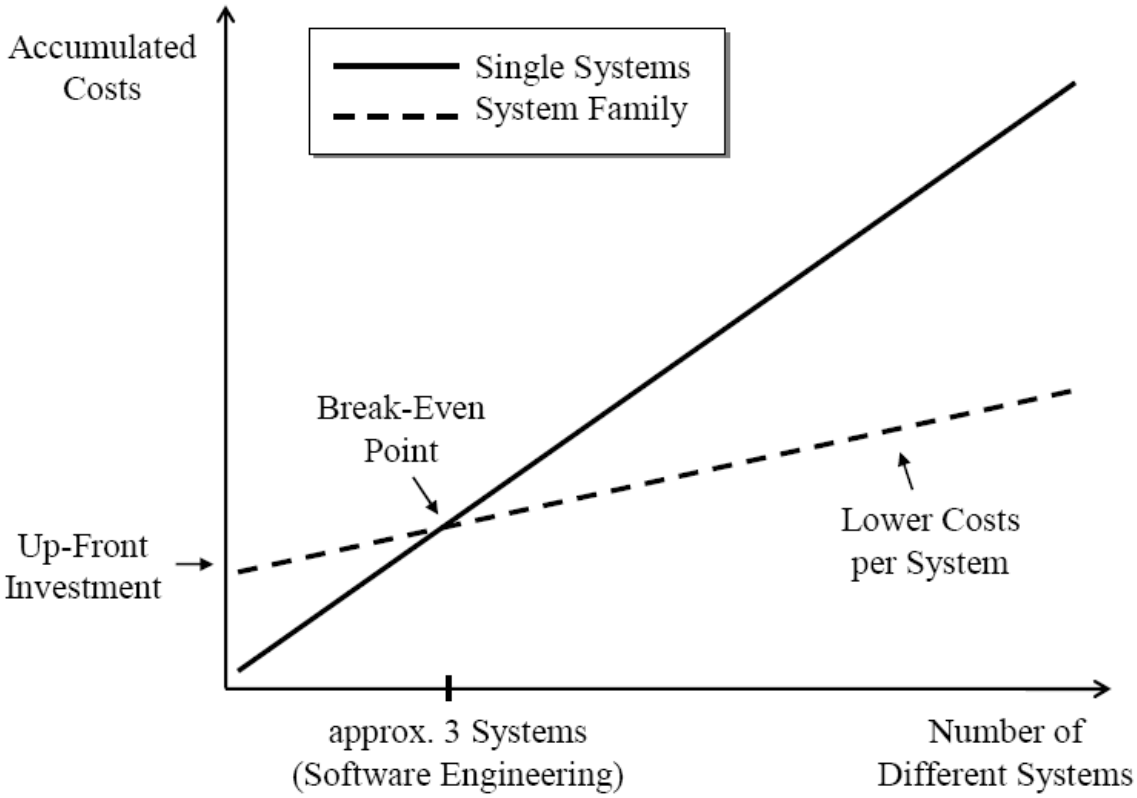


Fig. 2.1 Custo acumulado necessário para desenvolver uma LPS com n instâncias diferentes, com n produtos de software independentes [7]

O gráfico nos mostra que com até aproximadamente três sistemas, a implementação da LPS não se torna viável, visto que existem poucas variações.

Para a implementação da linha algumas técnicas são utilizadas atualmente na indústria. Especificamente para linhas de jogos móveis algumas destas são relevantes, as quais abordaremos a diante.

2.2. Compilação Condicional

Compilação condicional, ou pré-processamento, é uma técnica que permite especificar que determinados blocos de código serão incluídos na compilação de acordo com determinados parâmetros, normalmente especificados no processo de build. Principalmente na indústria de jogos móveis esta técnica se tornou padrão por gerar resultados mais eficazes para a linha de produto.

Para este estudo utilizamos o Antenna [8] que nos oferece um conjunto de diretivas que serão utilizadas na variação do código fonte. A Fig. 2.2 ilustra um trecho de código que faz uso destas diretivas.

```
//#ifdef device_screen_128x128
      public static final int SCREEN_WIDTH  = 128;
      public static final int SCREEN_HEIGHT = 128;
//#else
//# public static final int SCREEN_WIDTH  = 128;
//# public static final int SCREEN_HEIGHT = 117;
//#endif
```

Fig. 2.2 Bloco de código fazendo uso de diretivas de pré-compilação Antenna

Neste exemplo foram utilizadas as diretivas *//#ifdef, //#else* e *//#endif*. A primeira verifica se a *tag* de pré-compilação *device_screen_128x128* foi definida. Em caso afirmativo, o bloco de código posterior fará parte da compilação; em caso negativo, este irá ser comentado e o código após a diretiva *//#else* irá ser utilizado na compilação.

Sendo assim, através do uso de *tags* que representam as *features* do jogo, o código é tomado por linhas comentadas de pré-compilação que

acabam por comprometer a sua boa legibilidade em prol da performance e tamanho do *bytecode* final, sendo este último muito importante para o ambiente de jogos móveis.

2.3. Bytecode

Na linguagem de programação Java, todos os arquivos fontes são inicialmente escritos em arquivos de textos com a extensão .java. Estes arquivos são então compilados para arquivos .class pelo compilador *javac*. Um arquivo .class não contém código nativo do processador da máquina; ao invés, contém bytecode – a linguagem de máquina da Máquina Virtual Java (Java VM). O aplicativo *java* então roda sua aplicação com uma instância da Java VM [9].

No ambiente de jogos móveis implementados com J2ME [10], para cada instância da LPS é gerado um arquivo de aplicativo (arquivo jar) que representa o *bytecode* final da aplicação. Tendo os aparelhos limitações de memória e processamento, o tamanho deste arquivo é fundamental para o um bom alcance do jogo em vários aparelhos, melhorando a portabilidade do aplicativo.

2.4. Herança

Herança é uma das relações mais importantes no paradigma de orientação a objeto. Atua como mecanismo de expressar a relação entre duas classes (generalização), e é usada para especificar que uma classe é um tipo especial de uma outra classe (especialização). Os termos "generalização" e "especialização" são considerados normalmente como sinônimos de "herança" [12].

Considere um animal sendo representado pela classe chamada *Animal*, abstraindo se o mesmo é um mamífero, anfíbio ou réptil. Como cada grupo de animais possui características, comportamentos e atributos diferentes, eles podem ser representados por classes que estendem o comportamento geral de um animal. Isto é, novas classes irão surgir (*Mamifero, Anfibio, Reptil,*), classes estas que estendem *Animal*. A Fig. 2.3 abaixo ilustra segundo um diagrama UML [13] este exemplo.

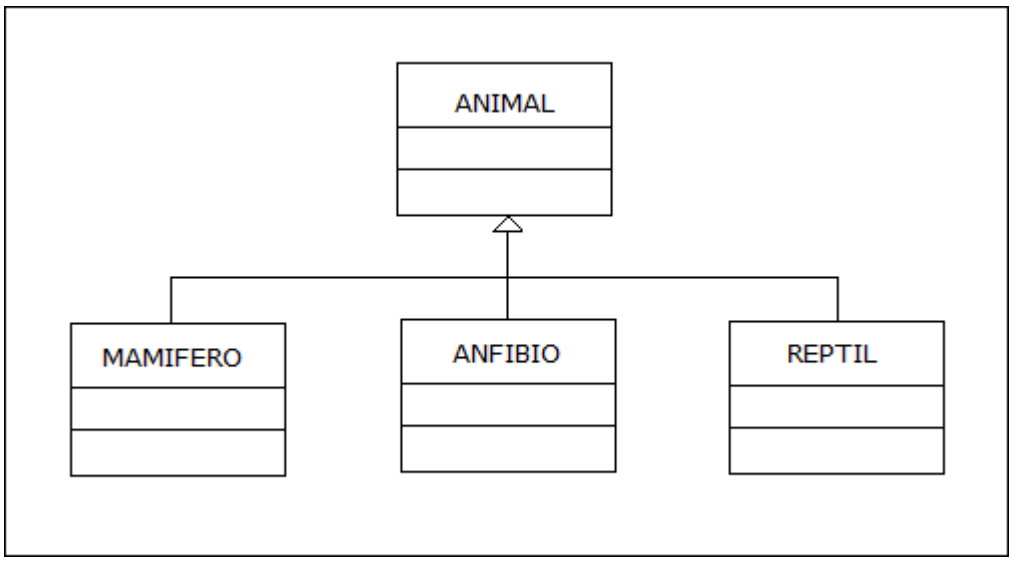


Fig. 2.3 Composição implementada com herança

Neste exemplo, a classe *Animal* é tida como a *superclasse*, enquanto as outras classes (*Mamífero, Anfibio, Reptil)* são chamadas de subclasses.

3. Implementação

Neste capítulo iremos explanar um pouco mais sobre nosso caso de estudo, o jogo BestLap, abordando sua implementação original, assim como as diversas formas de implementar suas variações.

3.1. Estudo de Caso

BestLap é um jogo móvel desenvolvido pela Meantime Mobile Creations e é atualmente distribuído da América do Sul, Europa e Ásia. São geradas várias versões deste mesmo jogo para 65 aparelhos diferentes, agrupados em 17 famílias. Cada família representa um conjunto de aparelhos compatíveis entre si rodando o mesmo código, isto é, uma mesma instância da LPS.

Fig. 3.1 Telas capturadas do jogo BestLap

É um jogo casual de corrida onde o jogador tenta alcançar o melhor tempo em uma única volta para então ser classificado para a *pole position.* O *score* do jogador depende do tempo da volta e dos bônus adquiridos durante o jogo. As melhores pontuações são guardadas. Opcionalmente, a pontuação pode ser enviada para um servidor, sendo comparada com o ranking dos outros jogadores. A Fig. 3.1 mostra algumas telas do nosso caso de estudo.

O jogo foi feito usando a plataforma J2ME juntamente com a técnica de compilação condicional para implementar as variações do produto. Para este trabalho abordaremos também outras técnicas utilizadas.

3.2. Implementação Original

A implementação original do jogo possui 14 classes e faz uso da técnica de Compilação Condicional, técnica esta padrão da indústria visto que gera melhores resultados no tamanho final do jogo, característica indispensável para aplicativos móveis, visto que muitos aparelhos possuem limitações de memória.

Fig. 3.2 ilustra um trecho de código da classe *GameMenu*, demonstrando como a técnica de compilação condicional é utilizada.

```
public void setType(int menuType) {
    this.menuType = menuType;
    PREVIOUS_KEY_CODE = MainCanvas.UP_PRESSED;
    NEXT_KEY_CODE = MainCanvas.DOWN_PRESSED;

    switch (this.menuType) {
        // #if feature_arena_enabled
        case POST_SCORE_MENU:

            this.menuTitlePosY = GameMenu.MENU_TITLE_POS_Y;
    *       this.message = null;
            this.firstOptionPosY = GameMenu.MAIN_MENU_OPTIONS_INITIAL_POS_Y;
            this.gapBetweenOption = MAIN_MENU_GAP_BETWEEN_OPTIONS;
            break;
        // #endif
        case END_RACE_MENU:
            this.optionIndex = 0;
            options = this.endRaceMenuCommands;
            leftSoftKey = GameMenu.COMMAND_ID_SOFTKEY_CONFIRM;
            rightSoftKey = GameMenu.COMMAND_ID_SOFTKEY_CONFIRM;
            break;
    }
    this.setMenuOptions(options, leftSoftKey, rightSoftKey);
}
```

Fig. 3.2 Código da classe *GameMenu* usando diretivas de pré-compilação

No código acima, retirado do método *setType* da classe *GameMenu,* caso a tag de pré-compilação *feature_arena_enabled* não seja definida no processo de build da aplicação, o bloco de código destacado será comentado, não sendo então compilado. Este fato interfere diretamente no tamanho do bytecode final do jogo, fazendo com que o mesmo se reduza.

Esta variação especificamente indica se o jogo terá ou não o requisito A*rena,* indicando se a pontuação do jogador será enviada ao servidor para ser visto juntamente com o progresso dos outros jogadores em um site. Porém outras variações mais gerais, como tamanho da tela do aparelho, podem ser encontradas no código como mostra a Fig. 3.3 logo abaixo

```
// #if device_screen_128x128
/** Position X of the left softkey */
public static final int SOFTKEY_COMMAND_LEFT_POS_X = 3;
/** Position Y of the softkeys */
public static final int SOFTKEY_COMMAND_POS_Y = 112;
// #elif device_screen_128x117
// #
// # /** Position X of the left softkey */
// # public static final int SOFTKEY_COMMAND_LEFT_POS_X = 3;
// #
// # /** Position Y of the softkeys */
// # public static final int SOFTKEY_COMMAND_POS_Y = 101;
// #endif
```

Fig. 3.3 Variações dos atributos referentes ao tamanho da tela do aparelho

Da mesma forma, dependendo da definição da variavel de pré-compilação *device_screen_128x128,* os valores das variáveis *SOFTKEY_COMMAND_LEFT_POS_X* e *SOFTKEY_COMMAND_POS_Y* irão variar. Como este jogo está disponível para 65 aparelhos diferentes, existem várias outras combinações destes pares de variáveis para atender aos diferentes tamanhos de tela.

Com estes trechos de código, podemos ter uma idéia do quão tomado por linhas comentadas de pré-compilação se torna o código final do jogo. Este é um fator que degrada a boa legibilidade do código. Porém o que faz esta técnica ser a mais utilizada pela indústria atualmente é o fato destas muitas linhas de código de pré-compilação serem comentários JAVA, sendo então desconsideradas pelo compilador *javac*, não influenciando no tamanho final do *bytecode* [11].

3.3. Implementação usando Herança

Nesta implementação do jogo, a técnica usada para implementar as variações foi herança. Muitas subclasses estendem outras classes que

pertencem a API J2ME (por exemplo MIDlet), mas este tipo de herança não é o foco desta seção, visto que não é utilizada como meio de implementar as variações da LPS.

Mas para esta implementação casos de herança estratégica foram geradas para implementar as variações da linha, tendo a preocupação de não prejudicar a qualidade e os requisitos de cada *build.* A Fig. 3.4 ilustra um caso onde herança foi utilizada para implementar uma variação da linha.

```
public class GameMenuArena extends GameMenu {

    // #if feature_arena_enabled
    public void changeMenuType() {
        if (this.menuType == GameMenu.POST_SCORE_MENU) {
            this.menuTitlePosY = GameMenu.MENU_TITLE_POS_Y;
            this.message = null;
            this.firstOptionPosY = GameMenu.MAIN_MENU_OPTIONS_INITIAL_POS_Y;
            this.gapBetweenOption = GameMenu.MAIN_MENU_GAP_BETWEEN_OPTIONS;
        } else {
            super.changeMenuType();
        }
    }
    // #endif
```

Fig. 3.4 Exemplo de herança na implementação de uma variação da LPS

O bloco de código exposto na Fig. 3.4 representa a classe *GameMenuArena* que estende a classe *GameMenu* e sobrescreve o método *changeMenuType()* da sua superclasse. Neste exemplo, caso a tag de pré-compilação *feature_arena_enabled* estiver definida, uma verificação será feita e caso não seja satisfeita, o método da superclasse será chamado.

Esta abordagem pode ser interessante pelo fato de que isolamos todo o conteúdo relacionado à tag *feature_arena_enabled*, onde caso a condição exposta seja satisfeita, todo o método da superclasse não precisará ser acessado e caso a diretiva não seja definida, o bloco de código mostrado na

Fig. 3.4, antes pertencente ao método da superclasse, não será compilado, diminuindo o tamanho final do bytecode.

Para definir qual classe o código irá chamar (*GameMenuArena* ou *GameMenu*) uma terceira classe foi criada, chamada *GameMenuFactory,* cujo conteúdo é mostrado na Fig. 3.5 logo abaixo.

```
public class GameMenuFactory {

    public static GameMenu createGameMenu() {
        // #if feature_arena_enabled
        return new GameMenuArena();
        // #else
        // # return new GameMenu();
        // #endif

    }

}
```

Fig. 3.5 Classe *GameMenuFactory* direcionando as chamadas ao *GameMenu*

Sendo assim, chamadas ao *GameMenu* passará antes pela classe *GameMenuFactory* que irá verificar se a diretiva *feature_arena_enabled* está definida, podendo então direcionar para a subclasse *GameMenuArena*, ou para sua a superclasse *GameMenu.*

3.4. Implementação com Técnicas Combinadas

Analisaremos agora como se deu nosso trabalho em desenvolver uma nova implementação do jogo usando técnicas combinadas para implementar as variações. Isto é, iremos fazer uso de duas técnicas simultaneamente, com o intuito de obtermos melhores resultados para a nossa linha de produção. Para nosso estudo utilizaremos as duas técnicas já

vistas anteriormente. Iremos expor duas implementações possíveis usando estas técnicas, com foco pouco diferente entre si.

3.4.1. Implementação 1

De forma geral iremos mesclar as características das duas técnicas. Faremos uso de herança, gerando novas subclasses a partir de outras (superclasses), e todo o código que representa estas subclasses será envolvido por diretivas de pré-compilação. Isto fará com que, caso a diretiva não esteja definida, o custo em tamanho desta subclasse seja zero, não impactando no tamanho final da aplicação. O diagrama UML abaixo ilustra o impacto desta decisão para a classe *GameMenu*.

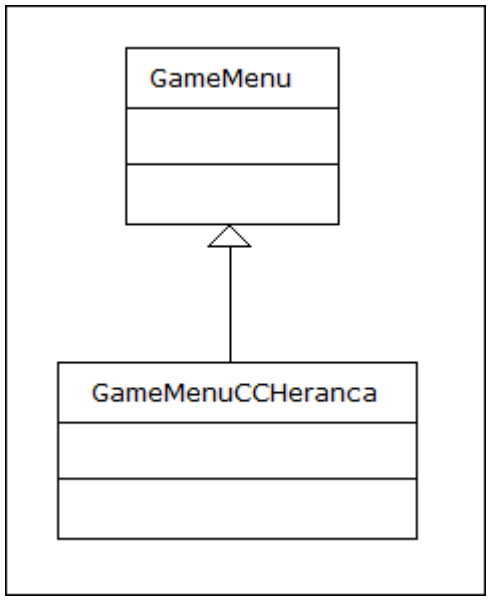


Fig. 3.6 Nova classe *GameMenuCCHeranca* herdando de *GameMenu*

Iremos transferir para estas subclasses todo código relativo a uma determinada diretiva, isto é, ao invés de termos em uma mesma classe várias diretivas, como já mostrado na Fig. 3.3, teremos várias subclasses envolvidas pelas suas respectivas diretivas. Todas estas ficarão em um

mesmo arquivo, visto que as diretivas devem ser disjuntas entre si, isto é, a seleção de uma delas implica na não seleção das outras, como mostrado na Fig.3.7.

```
//#if device_screen_128x128
public class GameMenuCCHeranca extends GameMenu {

    /** Gap between selection and option */
    public final int GAP_SELECTOR = 5;

    /** Gap between options used to main menu */
    public final int MAIN_MENU_GAP_BETWEEN_OPTIONS = 5;

}
//#elif device_screen_128x160
//# public class GameMenuCCHeranca extends GameMenu {
//# /** Vertical position of menu's title */
//# public static final int MENU_TITLE_POS_Y = 38;
//#
//# /** Vertical position of main menu options */
//# public static final int MAIN_MENU_OPTIONS_INITIAL_POS_Y = 70;
//# }
//#endif
```

Fig. 3.7 Conteúdo da classe *GameMenuCCHeranca*

Porém o fato de mover os atributos afetados pelas diretivas para a subclasse *GameMenuCCHeranca,* acabou gerando erros de compilação na classe *GameMenu*, visto que estes atributos movidos são usados por alguns métodos dentro desta última classe. Movendo então estes métodos para a subclasse geramos então outro problema, o de termos códigos duplicados, visto que estes métodos não sofrem alteração de acordo com a definição ou não das diretivas. Buscando resolver então este outro problema, tivemos que criar uma nova classe chamada *GameMenuCommon,* contendo estes métodos e estendendo a classe *GameMenuCCHeranca,* como mostra o diagrama abaixo. Desta forma, as requisições que antes eram feitas para a classe *GameMenu,* agora serão feitas para a classe *GameMenuCommon.*

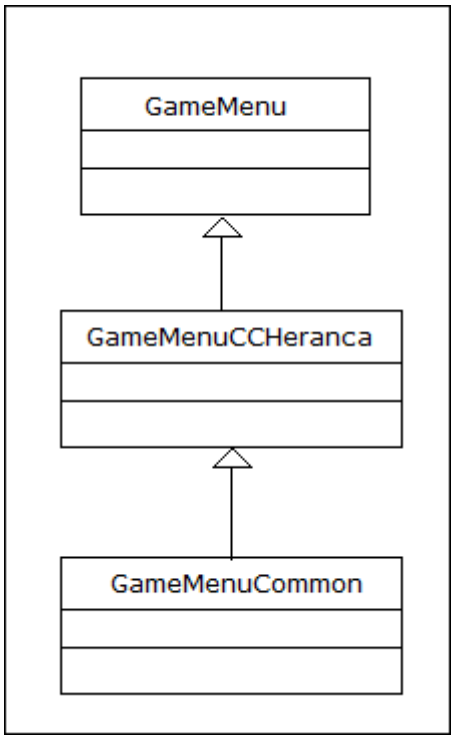


Fig. 3.8 Nova classe *GameMenuCommon* herdando de *GameMenuCCHeranca*

3.4.2. Implementação 2

Nesta segunda implementação iremos utilizar herança de forma mais estratégica em partes do código. Para isso iremos focar em uma única *feature* que tenha certa relevância, procurando aperfeiçoar ao máximo o código.

Observamos a diretiva *feature_arena_enabled* bem presente em todo o código que indica se o jogo conterá a opção Arena, sendo então utilizada em nosso estudo. Esta *feature* foi escolhida por representar 14.74% do total de linhas de código na versão original do BestLap e por ser uma *feature* importante na LPS, por estar presente em 15 instâncias da linha

Utilizando a mesma classe da implementação anterior (*GameMenu)*, iremos demonstrar o processo reproduzido nesta segunda implementação.

Nesta classe alguns blocos de código estão envolvidos pela diretiva citada, como mostrado na Fig.3.9, podendo então ser movidos para a nova classe a ser criada (*GameMenuArena*). O diagrama da Fig. 3.10 mostra a nova subclasse.

```
public void show(int type){
    this.setType(type);
    int[] options = null;
    int leftSoftKey = 0;
    int rightSoftKey = 0;

    switch(type){
        case GameMenu.MAIN_MENU:
            this.optionIndex = 0;
            options = this.mainMenuCommands;
            leftSoftKey  = GameMenu.COMMAND_ID_SOFTKEY_CONFIRM;
            rightSoftKey = GameMenu.COMMAND_ID_SOFTKEY_EXIT_CONFIRM;
            break;

        case GameMenu.ENABLE_SOUND_MENU:
            leftSoftKey  = GameMenu.COMMAND_ID_SOFTKEY_SOUND_ON;
            rightSoftKey = GameMenu.COMMAND_ID_SOFTKEY_DECLINE_SOUND;
            break;
//#if feature_arena_enabled
        case GameMenu.POST_SCORE_MENU:
            this.optionIndex = 0;
            options = this.postScoreMenuCommands;
            leftSoftKey  = GameMenu.COMMAND_ID_SOFTKEY_CONFIRM;
            rightSoftKey = GameMenu.COMMAND_ID_SOFTKEY_GO_MAIN_MENU;
            break;
//#endif
     }

    this.setMenuOptions(options, leftSoftKey, rightSoftKey);
}
```

Fig. 3.9 Bloco de código da classe *GameMenu*

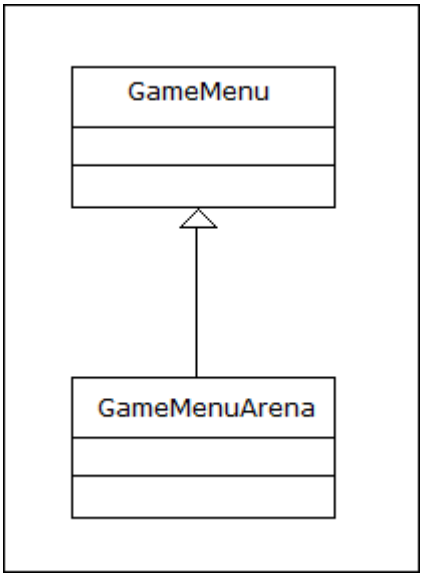


Fig. 3.10 Nova classe *GameMenuArena* herda de *GameMenu*

Na Fig.3.11 mostramos então o conteúdo da nova subclasse, para onde movemos o bloco de código relativo à *feature* Arena. Foi criado um novo método na superclasse (*gameMenuType()*) cujo corpo é todo o *switch* mostrado anteriormente, porém sem o *case* envolvido pela diretiva. Nesta nova subclasse este método é sobrescrito para a execução inicial do *case* movido.

```
//#if feature_arena_enabled
public class GameMenuArena extends GameMenu {

    public void changeMenuType() {
        if (this.menuType == GameMenu.POST_SCORE_MENU) {
            this.menuTitlePosY = GameMenu.MENU_TITLE_POS_Y;
            this.message = null;
            this.firstOptionPosY = GameMenu.MAIN_MENU_OPTIONS_INITIAL_POS_Y;
            this.gapBetweenOption = MAIN_MENU_GAP_BETWEEN_OPTIONS;
        }
        else {
            super.changeMenuType();
        }
    }
}
//#endif
```

Fig. 3.11 Nova classe *GameMenuArena* herda de *GameMenu*

A chamada para a classe *GameMenu* irá sofrer alterações também, visto que, caso a *feature* Arena esteja definida, *GameMenu* não deve ser chamada, e sim a sua subclasse *GameMeuArena,* como ilustrado da Fig.3.12.

```
        //Create the game menu
//#if feature_arena_enabled
        this.gameMenu = new GameMenuArena();
//#else
//#     this.gameMenu = new GameMenu();
//#endif
```

Fig. 3.12 Inicialização do menu do jogo

4. Análise

Neste capítulo analisaremos os resultados obtidos com o nosso estudo, detalhando e comparando com as outras especificações já mencionadas anteriormente.

Um dos fatores críticos para obtenção dos resultados se deve à ausência de ferramentas de processamento de métricas que atendam ao objetivo deste trabalho.

Foi encontrado um estudo para a obtenção destes resultados no qual o autor, Pedro Osandy Alves Matos Junior mestre em Ciências da Computação pela Universidade Federal de Pernambuco, desenvolveu para auxílio a sua dissertação. Não se trata de uma ferramenta já estruturada, mas sim de um *script* desenvolvido em JAVA que irá gerar resultados para cada métrica já pré-estabelecida [4].

4.1. Métricas

Nesta sessão descreveremos as métricas utilizadas no nosso trabalho. Iremos explanar o significado de cada uma, assim como qual a sua importância para nosso estudo. Segue cada uma abaixo.

Bytecode Size (BS) – Tamanho do Bytecode: Esta métrica refere-se ao tamanho do arquivo final (arquivo jar) de cada instância da LPS. O arquivo jar contém não somente classes compiladas ou aspectos, mas também qualquer arquivo necessário para rodar a aplicação, assim como imagens, arquivos de som, e arquivos de configuração. O tamanho do jar é calculado após a aplicação de algum otimizador de bytecode ou ferramenta obfuscador, isto é, medimos apenas o tamanho dos arquivos que irão

realmente ser distribuídos para o usuário final. Esta métrica é classificada como de eficiência, isto é, podemos inferir quão eficiente é determinada implementação. Esta técnica é muito relevante no contexto de jogos móveis, devido a grande restrição de memória que grande parte dos aparelhos possuem.

As próximas quatro métricas classificam-se como métricas de complexidade. São utilizadas para inferir a complexidade dos artefatos de código fonte. Uma LPS com muitos componentes, contendo muitas linhas de código fonte é mais difícil de se manter que uma mais simples.

Lines of Code (LOC) – Linhas de Código: Calcula o número de linhas de código para cada *implementation asset*. Comentários e linhas em branco são desconsiderados.

Number of Attributes (NOA) – Número de Atributos: Calcula o número de atributos para cada *implementation asset*. Para aspectos é considerado tanto atributos de aspectos e atributos inter-type definidos dentro do aspecto. Para arquivos de configuração, cada entidade é considerada como um atributo.

Number of Operations (NOO) – Número de Operadores: Similar a métrica NOA, esta calcula o número de operadores (métodos e *advices* para aspectos) para cada *implementation assets.*

Weighted Operations per Component (WOC) – Quantidade de Operações por Componente: Mede a complexidade de um componente, em termos de suas operações (métodos ou *advices*). Nós consideramos que o número de parâmetros formais de uma operação como um fator de

complexidade, isto é, operações com mais parâmetros formais são mais complexos que outro com menos parâmetros. Desta forma, denotamos a complexidade da operação *O*, *c* da seguinte forma: $c = q + 1$, onde *q* é a quantidade de parâmetros de *O*.

As próximas duas métricas são relacionadas como métricas de modularidade e independência entre os módulos da LPS.

Coupling between Components (CBC) – Acoplamento entre Componentes: Calcula, para cada componente (*implementation asset),* o número de outros componentes com os quais está acoplado. Nós consideramos que o componente *A* está acoplado ao componente *B,* se a declaração de *A* referencia *B* de alguma forma.

Lack of Cohesion in Operations (LCOO) – Falta de Coesão nas Operações: Esta métrica mede a falta de coesão de um component (classe ou aspect), através da quantidade de entidades usada nas operações em cada *implementation asset***.**

Esta última métrica está relacionada à separação de conceitos nos componentes, podendo ser identificado problemas de dispersão de confusão entre os conceitos implementados.

Number of Concerns per Component (NCC) – Número de Conceitos por Componente: Calcula o número de diferentes conceitos que um componente implementa. Um valor alto para esta métrica é um sinal de que a implementação dos diferentes conceitos está confusa no componente.

4.2. Resultados

Nesta sessão iremos expor alguns resultados das métricas das implementações já expostas anteriormente assim como os resultados da versão original do jogo implementada usando compilação condicional.

A Tabela 4.1 mostra o valor de algumas métricas para a implementação original do BestLap, onde todas as variações, inclusive a feature Arena, são implementadas com a técnica de compilação condicional.

De todas as classes listadas na tabela, apenas as classes *MainCanvas*, *MIDletController*, *Resources*, *GameScreen*, *MainScreen*, *GameMenu* e *NetworkFacade* contém algum bloco de código relacionado a *feature* Arena. *NetworkFacade* é a única classe que foi criada apenas com o propósito de atender à *feature* Arena. As demais classes não possuem ligação direta com Arena, mas são de alguma forma afetadas por esta *feature.*

O tamanho do *bytecode* final para uma das instâncias da linha que utilizam a *feature* Arena (S40) foi de 65701 bytes enquanto para outra instância da linha que não possui Arena (SAM1) foi de 66406 bytes.

Classe	NOA	NOO	WOC	CBC	LCOO
ArrayList	3	10	17	5	45
BVGAnimator	23	13	30	17	78
BytePNG	10	15	30	14	105
GameMenu	146	22	31	10	231
GameScreen	244	59	93	16	1711
LevelManager	46	22	33	16	231
MainCanvas	68	29	52	20	406
MainScreen	64	34	75	15	561
MIDletController	7	13	14	14	78
NetworkFacade	28	13	21	21	78
Resources	308	49	115	26	1176
Screen	18	43	119	24	903
SoundEffects	20	12	20	23	66
Sprite	24	12	18	6	66

Tabela. 4.1 Métricas para LPS BestLap usando a técnica de compilação condicional

Neste próximo passo iremos expor os resultados das métricas referentes à Implementação 1 já descrita neste estudo. A Tabela 4.2 mostra o valor de algumas métricas para cada classe desta implementação, incluindo as novas classes adicionadas com a aplicação da técnica.

Classe	NOA	NOO	WOC	CBC	LCOO
ArrayList	3	10	17	5	45
BVGAnimator	23	13	30	17	78
BytePNG	10	15	30	14	105
GameMenu	95	18	23	9	153
GameMenuCCHeranca	48	1	1	4	0
GameMenuCommon	4	4	8	7	6
GameScreen	244	59	93	16	1711
LevelManager	46	22	33	16	231
MainCanvas	68	29	52	20	406
MainScreen	64	34	75	15	561
MIDletController	7	13	14	14	78
NetworkFacade	28	13	21	21	78
Resources	308	49	115	26	1176
Screen	18	43	119	24	903
SoundEffects	20	12	20	23	66
Sprite	24	12	18	6	66

Tabela. 4.2 Métricas para LPS BestLap para Implementação 1

Analisando os valores obtidos, podemos perceber que houve um aumento na complexidade da implementação do BestLap. Isto é refletido em um pequeno aumento de uma unidade no NOA, NOO e no WOC e um aumento de 11 unidades em CBC. Houve uma diminuição considerável na LCOO, diminuindo 72 unidades.

É interessante ser notificado também o fato de que a quantidade de linhas na classe *GameMenu* foi reduzido em 1204 unidades, pois muitos atributos e métodos relacionados as *features* que indicam o tamanho da tela foram movidos para a nova subclasse gerada.

Iremos agora expor os resultados das métricas referentes à Implementação 2 descrita neste trabalho.

A Tabela 4.3 mostra o valor de algumas métricas para cada classe desta implementação, incluindo as novas classes adicionadas com a aplicação da técnica. Lembrando que esta implementação busca abordar o código relacionado à *feature* Arena.

Classe	NOA	NOO	WOC	CBC	LCOO
ArrayList	3	10	17	5	45
BVGAnimator	23	13	30	17	78
BytePNG	10	15	30	14	105
GameMenu	146	23	32	10	253
GameMenuArena	0	1	1	2	0
GameScreen	244	60	96	16	1770
LevelManager	46	22	33	16	231
MainCanvas	68	29	52	20	406
MainScreen	64	35	76	15	595
MainScreenArena	0	1	1	2	0
MIDletController	7	13	14	14	78
NetworkFacade	28	13	21	21	78
Resources	308	49	115	26	1176
Screen	18	43	119	24	903
SoundEffects	20	12	20	23	66
Sprite	24	12	18	6	66

Tabela. 4.3 Métricas para LPS BestLap para Implementação 2

Analisando os valores obtidos, podemos perceber que houve novamente um aumento na complexidade da implementação do BestLap. Isto é refletido em um pequeno aumento nos valores das métricas NOO, WOC e CBC. Nesta situação houve um aumento na LCOO, aumentando 178 unidades.

A quantidade de linhas na classe *GameMenu* foi reduzido em apenas 9 unidades, pois algumas linhas de código foram transferidas para a nova subclasse gerada.

4.3. Avaliação

Analisando estes resultados podemos perceber que a implementação com compilação condicional ainda consegue ter menos impacto na complexidade da LPS.

Mas a técnica de compilação condicional, mesmo com todos esses atributos e preferência, não provê modularidade para as *features* como Arena, usada em nosso estudo. Nesta versão da LPS todas as linhas de código relacionadas a uma determinada *feature* também estão relacionadas a outros conceitos do BestLap, definidos entre diretivas de pré-compilação. Por esta mesma razão a Implementação 1 e Implementação 2geram melhores resultados com relação à modularidade de conceitos. Usando compilação condicional não temos a idéia de modularidade de conceitos. Em ambas as implementações criamos entidades cujos propósitos eram de apenas implementar a *feature* Arena, o que torna estas versões mais fácil de manter o código fonte relacionado a esta *feature*

Todas estas métricas vistas e discutidas até o momento possuem apenas méritos qualitativos do código fonte. Iremos analisar agora questões mais quantitativas que, como já falado anteriormente neste estudo, é de

grande importância no contexto de jogos móveis. Para isso iremos analisar os resultados gerados para a métrica BS, que indica o tamanho final do *bytecode*. Para cada instância da LPS iremos rodar esta métrica e, a partir da média entre todas elas, iremos analisar os resultados podendo então julgar o impacto causado pelas duas implementações.

Instância	Compilação Condicional	Implementação 1	Implementação 2
S40	65701	66610	66526
S40M2	73380	74221	75300
S40M2V3	72876	73165	73665
SAM1	66918	67699	64856
SAM2	80893	79838	78838
SE01	71508	72419	72358
SE03	72960	73804	74880
SE04	72892	73185	73685
SIEM2	72936	73790	74856
SIEM4	72847	71643	73643

Tabela. 4.4 Métrica BS para diferentes implementações da LPS BestLap

Como esperado a implementação usando a técnica de compilação condicional conseguiu alcançar os menores tamanhos do *bytecode* final para as instâncias da nossa LPS.

Um fato relevante a ser observado é que o resultado da Implementação 2, cujo código aborda a *feature* Arena, é menor que o resultado para a implementação usando compilação condicional para algumas instâncias da linha, chegando a ter uma redução de até 2000 bytes, o que para o contexto móvel pode representar grande valor. Estes resultados chegam a ser melhores que os obtidos com a implementação deste mesmo jogo usando Aspectos [14], cujos resultados foram obtidos por Pedro Osandy em seu estudo feito [4].

Estas instâncias (como a SAM1 e a SAM2) possuem algo em comum que os fazem ter estes resultados distintos, que é fato deles não possuírem a *feature* Arena.

Isto se deve ao fato de que esta implementação moveu linhas de código relacionadas à *feature* Arena para as novas subclasses geradas e, caso a diretiva relacionada à Arena não esteja definida, estas subclasses não serão incluídas no pacote final do jogo, visto que está completamente envolvida pela diretiva de pré-compilação, como nos mostra a Fig.3.11.

Dependendo do contexto e ferramentas de geração de *builds* a partir de uma LPS de determinada empresa de aplicativos móveis, pode-se optar por gerar builds de determinadas instâncias da linha usando certa técnica e utilizar outras técnicas para a implementação de outras instâncias. Com isso poderíamos absolver os melhores resultados variando as técnicas para gerar cada instância da linha.

5. Conclusões

Neste trabalho apresentamos alguns critérios, atributos e métricas para mensurar a qualidade de uma Linha de Produto de Software. Ao final podemos obter mais informações a respeito de que técnicas usar para a implementação da nossa linha.

Inicialmente contextualizamos o nosso estudo para o ambiente de aplicativos móveis, mais especificamente para o jogo móvel BestLap criado pela empresa Meantime Mobile Creations. Através da literatura atual percebemos que a técnica mais utilizada atualmente pela indústria é a de compilação condicional, técnica esta que "suja" o código com linhas de pré-compilação, fazendo uso de diretivas, variáveis definidas geralmente antes do processo de build que definem quais instâncias da LPS serão geradas.

Como objetivo deste trabalho implementamos duas novas versões deste jogo fazendo uso de técnicas combinadas, isto é, utilizando duas técnicas simultaneamente. Para nosso estudo fizemos uso da técnica de compilação condicional e de herança. Para a primeira nova versão utilizamos estas técnicas focando na *feature* Tamanho de Tela, que informa qual o tamanho da tela do aparelho final. Para a segunda implementação focamos nas linhas de código que estavam relacionadas à *feature* Arena que informa se o build conterá ou não a opção de enviar a pontuação do jogador para o servidor.

Finalmente rodamos algumas métricas de qualidade e de quantidade para cada nova implementação, assim como para a versão original do jogo implementada usando compilação condicional. Verificamos a importância de cada métrica, destacando a relevância da métrica Bytecode Size (BS) que informa o tamanho do *bytecode* final do jogo.

5.1. Contribuições

As contribuições deste estudo foram:

- Introdução de mais uma possível implementação de uma LPS utilizando técnicas combinadas;
- Implementação de uma solução para uma linha de produto de software utilizando compilação condicional juntamente com herança;
- Exposição de resultados através de métricas qualitativas e quantitativas para tomadas de decisão a cerca de qual técnica utilizar em uma LPS;
- Geração de novas instâncias da linha para o jogo BestLap com diminuição considerável do tamanho do *bytecode*;

5.2. Trabalhos Relacionados

Em [4] é apresentado um conjunto de critérios de avaliação e métricas para avaliar atributos de qualidade para linhas de produto de software. O autor define alguns tipos de variações, oferecendo um conjunto de padrões de implementação para cada tipo de variação.

São avaliadas algumas métricas, divididas de acordo com o critério de avaliação, como complexidade do código fonte, modularidade, separação de conceitos, e efeiciência na geração do *bytecode*. Foi gerado então um catálogo de tipos de variações que poderrão auxiliar trabalhos futuros.

Um trabalho recente [15] mostra, através de uma abordagem prática, novas formas de implementação de linhas de produto. Nele são detalhados a modelagem, implementação e criação de testes para cada novo componente. O estudo também introduz o conceito de técnicas combinadas para a implementação da linha.

Outro trabalho recente [25] foca na implementação de variações em casos de teste automatizados. O trabalho apresenta um modelo de decisão para auxiliar os líderes técnicos ou desenvolvedores a escolher mecanismos para reestruturar variações de teste em LPS. Pode-se com isto então, aumentar a modularidade das variações de testes e também a remoção de códigos duplicados.

5.3. Trabalhos Futuros

Listamos abaixo possíveis trabalhos futuros que possam surgir após este estudo.

- Aplicação da solução sugerida em um caso real a ser utilizado na indústria;
- Implementação de um framework para gerar versões de todas as instâncias da LPS, podendo ser utilizadas diversas técnicas para certos conjuntos de instâncias pré-definidos;
- Melhoramento da ferramenta utilizada para rodar as métricas, tornando-a mais amigável e flexível para adaptação a outros projetos.

Referências

[1] Robert Tercek. First decade of mobile games. Apresentação no *GDC Mobile*, 2007.

[2] Tarcísio Câmara. Portando Jogos Mobile em Escala. Apresentação no *SBGAMES 2006*.

[3] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001.

[4] MATOS JÚNIOR, Pedro Osandy Alves. Analisys of Techniques for Implementing Software Product Lines Variabilities. Dissertação de Mestrado

[5] (Clements & Northrop, 2001) P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001, pp. 608

[6] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley, 2005, pp. 701

[7] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[8] Antenna: An ant-to-end solution for wireless java, 2008. Last visit: March, 2008

[9] Sun Microsystems, http://java.sun.com/

[10] The Java ME Plataform, http://java.sun.com/javame/

[11] Java: Como Programar, Harvey M. Deitel

[12] Arthur J. Riel, *Object-Oriented Design Heuristics*, Addison Wesley

[13] Object Management Group – UML, http://www.uml.org/

[14] Márcio de Medeiros Ribeiro, Pedro Matos Jr., Paulo Borba, and Ivan Cardim. On the Modularity of Aspect-Oriented and Other Techniques for Implementing Product Lines Variabilities. In Proceedings of the 1st Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'07), in conjunction with the 21th Brazilian Symposium on Software Engineering (SBES'07), October 2007

[15] NASCIMENTO, L. M., Core Assets Development in Software Product Lines - Towards a Practical Approach for the Mobile Game Domain, Agosto 2008