# Core Assets Development in Software Product Lines - Towards a Practical Approach for the Mobile Game Domain

**Leandro Marques do Nascimento[1,2], Eduardo Santana de Almeida[1,3], Silvio Romero de Lemos Meira[1,2]**

[1]Reuse in Software Engineering Group – RiSE, Recife, PE, Brazil

[2]Federal University of Pernambuco – UFPE, Recife, PE, Brazil

[3]Federal University of Bahia – UFBA, Salvador, BA, Brazil

{lmn2,srlm}@cin.ufpe.br, esa@rise.com.br

***Abstract.*** *Software Product Lines (SPL) approaches are gradually being adopted as a powerful strategy for achieving high productivity and increasing quality in software engineering. A particular domain where the adoption of such approach may bring relevant benefits is the mobile game domain given the big diversity of handsets and the large number of commonalities among these games. However, applying SPL approaches in such domain is not trivial because of some restrictions, such as reduced memory and application size. In this context, this work presents a practical approach to implement core assets in a SPL in the mobile game domain combining good practices from previous work and briefly describing a case study performed with three mobile games.*

## 1. Introduction

One of the key factors for improving quality, productivity and consequently reducing costs in software development is the adoption of software reuse – the process of creating software systems from existing software rather than building them from scratch [Krueger, 1992]. Several researches have been done describing techniques, methods and processes in the software reuse area [Almeida et al., 2007] and different efforts have been made to apply the concepts of this area in practice with successful results [Product Line Hall of Fame, 2008], including big companies such as Hewlett-Packard, Motorola and Bosch.

An approach commonly cited in the software reuse area is Software Product Line (SPL), which is defined as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*", [Clements and Northrop, 2002]. In other words, a SPL enables the instantiation of new applications based on a set of core assets, developed from the analysis of commonalities and variabilities of a specific domain or market segment.

In addition, a particular market segment that is in high-growth during the last years is the mobile applications market, especially the mobile games. According to iSuppli[1], the prediction is that the worldwide mobile gaming market will be worth $6.1

[1] iSuppli Corporation: Applied Market Intelligence: http://www.isuppli.com/

billion in 2010 – up from $1.8 billion in 2005. However, this market involves much more complex challenges, mainly because of the variety of handsets and manufacturers, besides many platform restrictions and different hardware configurations. For example, a game produced by Meantime[2] (our industrial partner), called *My Big Brother*, had to be deployed for almost *fifty* devices and the game had to support variations across those devices, such as: different screen sizes, different ways of handling HTTP protocol and different keypad configurations [Alves et al., 2005]. This scenario demands well defined software processes to build applications compatible with as many handsets as possible. Therefore, SPL can be a suitable option in this case.

In the SPL research field, different efforts have been made [Weiss, 1999] [Bayer et al., 1999] [Atkinson et al., 2000] [Clements and Northrop, 2002] [Pohl et al., 2005] [Gomaa, 2005] [Almeida, 2007] applying various techniques in different contexts. Despite all of this relevant work mentioned, there is still a lack of details in the phases of the SPL processes related to domain implementation (or core assets development), such as: how features can be mapped on components and correspondingly on code; how variability can be managed in code level; or guidelines to apply variability implementation techniques. The lack of details in domain implementation phases can make SPL approach unaffordable especially when taking into account the restrictions of mobile game domain, such as *memory size*, *processing power*, *different screen sizes* and *different API implementations*. Because of these characteristics involving the mentioned domain, the adoption of a SPL should be supported by as many details as possible at domain implementation phase.

Considering these aspects, the purpose of this work is to establish a practical approach for implementing assets in a SPL applied to mobile game domain and validate its applicability through a case study involving, firstly, three adventure mobile games developed with Java Micro Edition (ME) technology, and next, the production of a fourth game exploring the common features among them. This approach applies different techniques for variability implementation, providing guidelines for mobile game developers within a SPL context, once the focus of this work remains on domain implementation level.

This paper is organized as follows: Section 2 presents the related work, Section 3 discusses the proposed approach, which is subdivided in three phases: **Component Modeling**, **Component Implementation** and **Component Testing**. In the sequence, a brief description of the case study with a practical application of this approach is described in Section 4. Finally, Section 5 presents the concluding remarks and future work.

## 2. Related Work

As it was already mentioned in the last section, several researches have been performed in the areas of software reuse and SPL. Most of the work published in these areas commonly present SPL based on the three essential activities: core asset development, product development, and management [Clements and Northrop, 2002]. These efforts can be characterized as model-based processes with common concepts such as: feature

[2] Meantime Mobile creations, available on http://www.meantime.com.br

modeling, domain (or product line) architecture, component modeling and composition and product instantiation based on the domain components. Examples of those processes are: PuLSE [Bayer et al., 1999], KobrA [Atkinson, et al., 2000], Software Engineering Institute's framework for SPL [Clements and Northrop, 2002], SPL Framework [Pohl et al., 2005] from Pohl, Bockle, and van der Linden, PLUS [Gomaa, 2005], and RiDE [Almeida, 2007].

On the other hand, other researchers take advantage of generative programming, such as FAST [Weiss, 1999], applied successfully in industrial cases such as Lucent Technologies and Avaya Labs.

All the previous work mentioned can be applied on different contexts, however, in the particular case of the mobile domain, adaptations in the processes should be done to make them address properly the restrictions of the domain. It happens mainly because some of the processes are too general or handle with technologies that are not well established in mobile domain, such as OSGi [OSGi Alliance, 2008]. Some efforts were made in order to adapt a SPL process to the mobile game domain, such as the GoPhone Project [Muthig et al., 2004], an adaptation of PuLSE and KobrA. This project did not consider common restrictions in the mobile domain, such as *screen size* and *different API implementations* according to different vendors.

In addition, another related work defines refactoring tools for extracting product lines from different versions of the same product, for instance, FLIP [Alves et al., 2008]. As a common practice in the mobile applications domain is to adapt an application for different handsets (porting), FLIP is intended to analyze the code of those different versions of the same application and then generate a product line based on aspect-oriented programming, mapping the cross-cutting concerns into aspects [Kiczales et al., 1997]. In the case of mobile game domain, for example, a cross-cutting concern can be *screen size*, because this characteristic of a handset affects different parts of the code at different levels. This kind of tool can be extremely useful when the product line is defined by a single product and different versions of it must be implemented to support different families of handsets, as the Abstraction Level 1 shown in Figure 1.

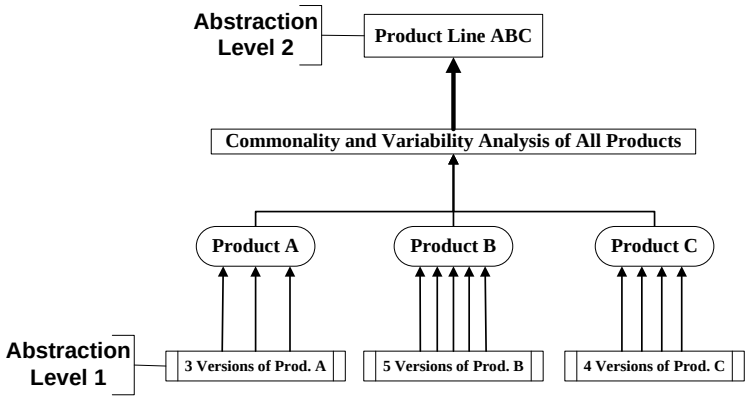


**Figure 1. Abstraction levels in software product lines applied to mobile game domain.**

However, not only is our approach intended to consider the different versions of the same product as potential variabilities to be used in other products, but it addresses

the commonality and variability of different products in the same domain at a higher level of abstraction, as Abstraction Level 2 shown in Figure 1.

## 3. A Practical Approach for Implementing Core Assets in SPLs applied to the Mobile Game Domain

Based on the good practices from different domain engineering and SPL processes [Almeida et al., 2005] added to our industrial experience, we structured an approach to cover the main specific aspects of mobile applications domain, as shown in Figure 2.

The approach is focused on the domain implementation phase of a SPL, abstracting the previous phases of domain analysis and design. The process is iterative, although the Figure 2 does not show it clearly to avoid complexity and misunderstandings.

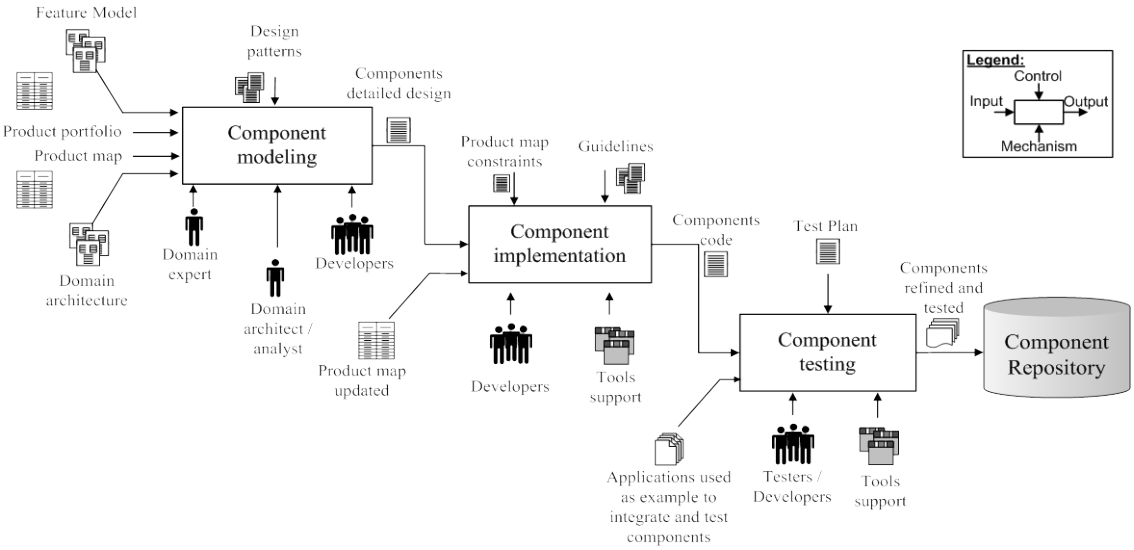


**Figure 2. Overview of the proposed SPL approach for the mobile game domain focused on domain implementation phase.**

Three distinct phases can be highlighted in the process: **Component Modeling**, **Component Implementation**, and **Component Testing**. From the former steps of domain analysis and domain design four mandatory artifacts are expected to be provided because they are used as input for the domain implementation and also throughout the three phases. These artifacts are:

**Product portfolio**. It describes all the product families. In this approach, it is considered that the product families are the target handsets in which the game must run. Each family has a base member, which is used as a reference for the entire family. Table 1 demonstrates an example of product portfolio.

**Product map**. It holds information about main capabilities of products, including the restrictions inherent to the mobile domain. It also maps the reference members of the families from product portfolio to the main capabilities of products. Table 2 shows an example of a product map. It describes the devices' general capabilities considering their main configurations, shown in the table as *Variation Levels*. It is important distinguish the handset capabilities from features, which are

represented in the domain feature model. In addition, Table 3 shows an example of how to map families of handsets to their respective main capabilities.

**Table 1. Example of product portfolio including priorities among families of handsets.**

| Manufacturer #1: Motorola | | | | | | | |
|---|---|---|---|---|---|---|---|
| **FAMILY** | **BASE** | **COMPATIBILITY** | **SCREEN** | **MIDP** | **HEAP** | **JAR** | **MP3** |
| MOTO_1 | **i830** | i830; i265; i275; i560 | 130x130 | 2.0 | 1.1 Mb | 4 Mb | ☑ |
| MOTO_2 | **i850** | i850; i870; i880 | 176x206 | 2.0 | 4 Mb | 4 Mb | ☑ |

| PRIORITY | MEMBER | BASE | COMMENTS |
|---|---|---|---|
| **1** | MOTO_2 | **i850** | Very well sold in Europe and USA |
| **2** | MOTO_1 | **i830** | Market: Nextel Brazil next trimester |

**Table 2. Example of product map.**

| Capability | Variation Level #1 | Variation Level #2 | Comments |
|---|---|---|---|
| **Screen Size** | 130x130 | | |
| | 176x206 | | Actual size of 176x220, but available size of 176x206 |
| | 240x320 | | |
| **Key Mapping** | Nokia | | |
| | Motorola | | |
| | SonyEricsson | | |
| **Sound** | Allocation Mode | Pre-allocate | At most 8 players at a time |
| | | On demand | |
| | Player instance | Block | Player in a separated block |
| | | Thread | Sound player in a thread |
| **Known issues** | Network | UDP problem | |
| | Memory | No Garbage Collection | Memory is not deallocated |

**Table 3. Example of mapping handset families to their main capabilities in the product map.**

| Member | Capability | Comments |
|---|---|---|
| **MOTO_1** | **Screen Size: 176x206** | |
| | **Keypad Motorola codes** | Key codes described in family's properties file |
| | **Sound on demand** | If the allocated players number reach 6, they must be closed before opening a new one |
| | **Player instance in block** | |
| | **No Garbage collection** | This problem implies in reducing the number |

**Domain architecture**. Modeled in terms of a well accepted notation, such as UML, the domain architecture is used to guide the development of new applications in the target domain. It holds information about the initial structure of application code and models the SPL extension points.

**Feature model**. Described in terms of a visual model and based on well accepted notation, such as FODA [Kang et al., 1990], this artifact shows the relationships among domain features. The feature model can also be used to guide on identifying components and refining product map/portfolio.

In order to properly apply the approach proposed in this work, it is highly recommended that all the artifacts' information is fully provided, following the given examples. Java Micro Edition (JME) is adopted as the base technology for all examples.

If any other technology is chosen, those artifacts should be reviewed to reflect the possible restrictions of the technology. Figure 1 also shows five different roles present in the approach: domain expert, domain analyst, domain architect, developer and tester.

### 3.1. Component Modeling

The main goal of this phase is to obtain the component detailed design (or component specification). For this purpose, the domain architect is the most important role, responsible for analyzing the architecture and modeling the identified components. This work does not focus on component identification based on domain architecture and feature model, but it can be observed as an important research field [Almeida, 2007.

During component modeling, the architect must take into consideration all information described in product portfolio and product map, because these artifacts may bring up some restrictions that must be addressed in components' internal design, such as, a family of handsets with reduced memory available.

Before defining the internal component structure, the domain architect must decide which implementation technique will be used to handle the variation levels (or variation points) in product map (example in Table 2). One of the most common implementation techniques applied to mobile domain is conditional compilation [Alves et al., 2005]. This technique can be easily applied to cut off unnecessary code and reduce the final application size. Although conditional compilation tags can reduce the code readability, there are tools to perform refactoring steps [Alves et al., 2008] helping on remaining the code readable. The use of conditional compilation tags does not exclude the use of any other variability implementation technique, such as, inheritance, delegation or aspect-oriented programming (AOP).

In order to add the conditional compilation tags for each respective variation level, the example product map and also the mapping of handset families and their main capabilities are refined.

To start performing the component specification, the KobrA approach [Atkinson, et al., 2000] is used as a reference and adapted. KobrA approach was chosen to be adapted because it describes a component using two levels of abstraction (internal and external point of views) and this makes it easier for developers and component users (integrators) to understand the component behavior. A KobrA is based on specification and realization models. Both models are composed by a general set of UML models: specification – functional model, behavioral model, structural model and decision model; realization – interaction model, execution model, structural model and decision model.

In this approach, the use of textual models is discouraged, such as the decision model, and only the use of a structural model is suggested. Doing it this way may avoid misunderstandings with textual representations and maintenance problems, if the textual model has to reflect the large number of handset families. On the other hand, the use of the structural model is encouraged, which is the base for component implementation, combined with `<<variant>>` stereotype to indicate a variant part of the component internal design. The domain feature model is used to help on identifying where the `<<variant>>` stereotype should be applied. Product portfolio and product map help while designing components to make them complaint to the game domain restrictions.

Moreover, the components can contain any of the conditional compilation tags listed in product map. In this situation, the component internal design should be complemented with additional information referent to which tags are being used by any specific component. Figure 3 shows an example of component internal design with `<<variant>>` stereotype and additional information of tags being used by the component.

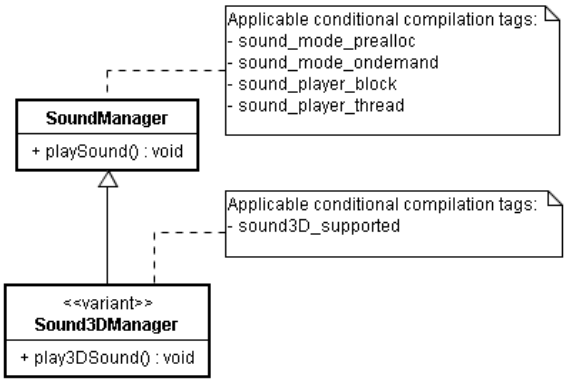


**Figure 3. Example of component internal design with `<<variant>>` stereotype and additional information of the applicable conditional compilation tags.**

## 3.2. Component Implementation

Once the components internal design has been accomplished, the developers are responsible for implementing these components using information from components' internal design (structural models), product portfolio descriptions, product map constraints and domain feature model.

As the developers have participated in the Component Modeling phase, they are supposed to be familiarized with internal component design and so can start components implementation. The major concern at this point is to code the domain restrictions and variability within the components specified using the already known techniques for variability implementation in product lines [Anastasopoulos and Gacek, 2001]. These techniques are: *conditional compilation*, *aggregation* / *delegation*, *inheritance*, *dynamic class loading*, *properties*, *static libraries*, *overloading*, *parameterization*, *design patterns*, *aspect-oriented programming (AOP)*, *reflection* and *dynamic link libraries (DLLs)*. Among these techniques, only two of them, reflection and dynamic link libraries (DLLs), are not applicable to mobile domain because the current versions of MIDP/CLDC 2.1/1.1 respectively, still do not support them.

In the sequence, for each applicable technique mentioned, guidelines and good practices of programming are presented through example scenarios to help developers during component implementation. Detailed information about these guidelines and example scenarios with respective code snippets can be found in [Nascimento, 2008].

**Conditional Compilation**. It enables control over the code segments to be included or excluded from a program compilation defined by pre-processing symbols. This technique is largely used in mobile domain, mainly because it takes out of the application scope all unnecessary code leaving the application as compact as possible to be deployed to different handsets with completely different capabilities. The major weak point of this technique is related to maintenance, once the code may become hard

to read and also not compilable in some environments. To mitigate this issue, support tools can be used to manage conditional compilation tags and then make it easier to build applications in the environment they are being deployed. Conditional compilation can become a powerful technique to be used when correctly combined with others techniques and good programming practices. Throughout the description of the other variability implementation techniques next, conditional compilation will be also explored, showing how to combine this technique with the others.

*Example Scenario for Conditional Compilation*. One common situation where conditional compilation can be used is when it is needed to add an optional feature to a specific product in the product line without increasing the other products size. For example, if one specific customer needs to add its logo on the first screen game of the game, usually called “splash” screen. The code responsible to instantiate the logo image and paint it on screen should be embraced by conditional compilation tags. Doing it this way, will not cause the other products to have its final size increased.

**Aggregation/Delegation**. Aggregation is an object oriented technique which enables objects to virtually support any functionality by forwarding requests they can normally not satisfy to so-called delegation objects which provide the requested services. Delegation is commonly used with facade design pattern.

*Example Scenario for Aggregation/Delegation*. This technique is usually necessary when a given object has different functionalities and needs to delegate some services to another object. For example, if it is needed to isolate *sound* component and delegate specific functionality to a *sound controller* and a *player controller*. The suggested approach to use Aggregation/Delegation is to combine it with conditional compilation tags and isolate these tags in delegated object, embracing its whole method bodies. Try to maintain the tags only inside these methods. Other methods that call the delegated one must not be embraced by the tags to avoid spreading tags all over the code. The following code snippet (double columned) represents delegation technique combined with conditional compilation tags.

```
public class SoundFacade {
  public void playSnd(String filename) {
    // No tags in here
    SoundCtrl.playSnd(filename);
  }
}
```

```
class SoundCtrl {
  static void playSnd(String filename) {
    // Delegated method body embraced by
    // tags
    //#ifdef sound_player_block
      //sound code 1
    //#elif sound_player_thread
      //sound code 2
    //#endif
  }
}
```

**Inheritance**. It is used to assign basic functionality to super classes and extensions to subclasses. Inheritance should be used carefully in mobile domain because the basic functionality present on super classes has to be actually applicable for all subclasses to avoid unused methods.

*Example Scenario for Inheritance*. Consider a game where projectiles are shot in the direction of the player and they can make a straight or curve path, according to the handset screen size in which the game will run. An abstract super class *Projectile* and

two subclasses, *Fireball* and *Missile*, are defined. In this typical situation, it is recommended that the entire subclasses' bodies are embraced by conditional compilation tags according to the target handset screen size.

**Dynamic Class Loading**. It is an automatic function of Java and allows loading a class in memory only at the first time the class is going to be used. To avoid unnecessary classes loaded in memory, *factory* design pattern can be used to manage objects in memory according to the specific product of the SPL.

*Example Scenario for Dynamic Class Loading*. One typical situation where dynamic class loading is applied is when a method call of some specific class is made inside an "*if*" block. If no other methods of that class had been previously called, then the class has not been loaded into memory yet. Therefore, to save application size resources, the "*if*" block must be replaced by conditional compilation tags.

**Properties**. Properties files can be used as a powerful technique to group all characteristics of a given device. For example, the set of conditional compilation tags, the name of resources folder, the set of all handsets in that specific family, all these information can be in a property file to be read during application deployment or even at runtime to execute certain functions.

*Example Scenario for Properties*. A common example of use of properties file is when the mobile application needs to be translated to different languages. All application texts are stored in a properties file.

**Static Libraries**. One of the most powerful ways to apply software reuse is to maintain a static library of methods/functions and/or components. The library can contain mathematical functions, network connections manager components, image processing functions, among others, and it can be properly applied to a mobile applications context without restrictions. Tools can be used to cut off unused methods in the deployment process.

*Example Scenario for Static Libraries*. Mounting a static library in the mobile game domain is usually necessary when the required functionalities are not present in the native API, for example, *Calendar* class to manipulate dates or complementary String and Image manipulation.

**Overloading**. This is a useful technique to reuse a method name by changing its signature. Just as parameterized methods, overloaded method bodies should be embraced by conditional compilation tags.

**Parameterization**. The behavior of the method being called is determined by the values of the parameters that are being set. Parameterization should be used carefully, once it can generate code that represents OO anti-patterns. To not increase the complexity of the parameterized method, it should delegate the correspondent functionality to a specific method and this last is responsible for executing the requested functions.

*Example Scenario for Parameterization*. One example where the parameterization can be a good solution is when there is a method responsible for playing different types of sounds according to the content type passed as parameter. As it was mentioned, in order to avoid complexity, the method checks the parameter and then delegates the operation for each specific method. In summary, the use of

parameterization technique is recommended to be combined with delegation and conditional compilation tags. The following code snippet illustrates this scenario.

```
class Sound {
  public void playFile(int contentType,
      String file) {
    switch (contentType) {
        case CONTENT_TYPE_MIDI:
            this.playMidi(file);
            break;
        case CONTENT_TYPE_MP3:
            this.playMp3(file);
            break;
    }
  } // Class body continues
  private void playMidi(String file){
    //#ifdef sound_type_mid
      /*method body*/
    //#endif
  }
  private void playMp3(String file){
    //#ifdef sound_type_mp3
      /*method body*/
    //#endif
  }
}
```

**Design Patterns**. As it was already mentioned, some design patterns may be used in conjunction with other techniques to manipulate variability at code level, such as: *factory*, *delegate*, *facade*, among others.

*Example Scenario for Design Patterns*. Typically in the mobile game domain, factory design pattern can be used with dynamic class loading to provide lazy instantiation of resources in memory. Mobile games usually make use of many images and frequently needs to load and unload image resources in memory. The following code snippet shows and example of factory pattern with lazy instantiation of images in memory. The dispose method provides a way to unload resources from memory.

```
public class ImageFactory {
  private Image menuBackground;
  private Image fireball;
  public Image getMenuBackground() {
    if (this.menuBackground == null) {
      // Lazy image instantiation
    }
  return this.menuBackground;
  }
  public Image getFireball() {
    if (this.fireball == null) {
      // Lazy image instantiation
    }
  return this.fireball;
  }
  private void disposeImages() {
    this.fireball = null;
    this.menuBackground = null;
  }
}
```

**Aspect-Oriented Programming (AOP)**. Applying AOP to mobile product lines can be an interesting approach to reduce development efforts while porting applications mainly because the major restrictions of mobile domain are related to cross-cutting concerns, such as, display size. AOP demands the use of tools to extract aspects from different versions of the same application and produce product lines [Alves et al., 2008].

Besides the described techniques, one approach that is commonly mentioned in combination with reuse and SPL research fields is OSGi [Almeida et al., 2008]. However, the Java Specification Request (JSR) 232, which enables OSGi platform for the mobile environment, has not got many adopters, mainly because, deep changes would be necessary in the MIDlet life cycle manager to enable OSGi and let two different JME applications share the same resources, such as a shared component library. For now, few JSR-232 capable handsets have been announced by the current main manufacturers making the usage of OSGi in the mobile applications environment not practical.

Based on component specification (structural model) as mentioned in the last section, and also on the product portfolio and product map descriptions considering the conditional compilation tags, the developers can implement the reusable components (SPL core assets) following the guidelines described in this section. After implementation, developers should update components' documentation with the following information: groups of conditional compilation tags of each component and the respective dependencies among them; memory usage; total component size in JAR file. This documentation may be very useful during component maintenance and SPL expansion to deploy new products based on the developed infrastructure.

### 3.3. Component Testing

Once the components have been implemented, they must be tested to ensure they are working properly. The main inputs for this phase are: the components code, example applications used to integrate and test those components and a Test Design (TSTD) document. An example application can be either an instantiated product of the SPL or a simple application developed to test component basic functionality. This information about how a specific component is going to be tested should be provided in the TSTD.

The TSTD should be structured based on the information of product portfolio, product map, domain feature model and requirements, usually represented as a Game Design Document in a game project. It is suggested that the tests are focused on the base members of the families, once it is not viable to test all the members of all families. It is implicit that if a component is working fine in the base member of the family, it will work properly on the other members of that family. At least, it is what is expected from the handset manufacturers when they produce different handsets of the same family and usually with the same hardware platform.

In order to better organize the TSTD, it is suggested that two different documents are produced: one responsible for holding domain tests, which can be used to test different components across different handset families; and other responsible for testing each application derived from the SPL. Detailed information about the testing approach can be seen in [Nascimento, 2008], as the main focus of this paper in on component implementation phase. Other research efforts have been made specifically focused on component testing, such as one in RiSE[3] Group shown in [Silva et al., 2009].

## 4. Case Study

In order to evaluate the described approach, we performed a case study following the organization proposed in [Wohlin et al., 2000]. We used three games of the same domain from Meantime: *Monga*, *American Dad – Roger's Escape* and *Zaak*. Based on the commonalities and variabilities of these three games, we defined a SPL, using the approach described in this paper, and produced a fourth game, called *Smart Escape*. The basic domain architecture has been implemented with 6489 non-commented source lines of code (NCSLOC) distributed in 277 methods.

[3] RiSE – Research in Software Engineering Group – www.rise.com.br


**Figure 4. Screenshots of the games in case study. A) Monga. B) American Dad – Roger's Escape. C) Zaak. D) Smart Escape.**

The fourth game derived from SPL has basic common features from all of the three previous games but adds a different way of interaction. The main goal of the game is to reach the door placed at the top of the screen without being noticed by the enemies spread through the floors. No form of weapon is used, unlike the three other games.

We used the GQM [Basili et al., 1994] approach in the case study to validate our process efficiency regarding the quality of core components. Thus, we defined three metrics: *Component Complexity* – uses cyclomatic complexity [McCabe, 1976]; *Domain Restrictions Management* – indicates the percentage of components with domain restrictions mapped to code level; *Traceability* – indicates the percentage of components specifications that can be mapped to domain features and to code (respective conditional compilation tags). We defined null hypotheses for the three metrics according to GQM, as summarized in Table 4.

**Table 4. Summary of Metrics and their respective null hypotheses.**

| Metric | Null Hypotheses |
|---|---|
| **Component complexity**<br>- McCabe Cyclomatic Complexity (CC) | CC >= 21 |
| **Domain Restrictions Management**<br>- $N_{FA}$: %Product families that have the restriction of application size mapped to code level<br>- $N_{FS}$: %Product families that have the restriction of different screen sizes mapped to code level<br>- $N_{FI}$: %Product families that have the restriction of different API implementations mapped to code level | $N_{FA}$ <= 50%<br>$N_{FS}$ <= 50%<br>$N_{FI}$ <= 50% |
| **Traceability**<br>- $T_{CF}$: %Components that can be mapped to domain features<br>- $T_{CC}$: %Components that can be mapped to code (respective conditional compilation tags) | $T_{CF}$ <= 70%<br>$T_{CC}$ <= 70% |

After collecting the metrics of the case study, all the null hypotheses could be rejected. Detailed information about the whole case study, including complete explanation of all null hypotheses, can be found in [Nascimento et al., 2008]. At the end, four components were developed, each of them with an average of 397 NCSLOC. The results of the case study have shown that the approach can be suitable for the mobile domain and some lessons were learned.

## 4.1. Lessons Learned

One of the strongest points of this approach is the facility of applying conditional compilation tags in code, mainly when the product map explicitly maintains all information about handsets capabilities and their respective tags. On the other hand, the use of these tags can become problematic if the number of product families grows too much. In our case, we did not face this problem because we used 3 product families from 3 major mobile phone manufacturers.

The main threat that could be identified in the case study execution is related to the staff. As the staff in this study has previous experiences in the mobile games, it may be difficult to find another group with the same characteristics to execute a new study.

Considering the values used for null hypotheses in case study, we realized that it may be necessary to perform new case studies to calibrate these values because we had no reference value from previous studies at project start. Considering specifically the *Component Complexity* metric value, we also realized that it may be slightly affected due to the number of conditional compilation tags. The calibration of these values may reveal that it is necessary to use some adjustment factor to component complexity metric in terms of the number of tags.

During project execution, it was noticed that a tool for managing all conditional compilation tags would be very useful to automatically deploy different products according to different families.

## 5. Concluding Remarks and Future Work

Software product lines are being explored in different domains and contexts with many successful cases. In addition, a market segment that is drawing industry's and academia's attention is the mobile game domain, mainly because of its high-growth. This domain presents specific characteristics, especially because of the great diversity of handsets and the need of ubiquitous applications running in as many handsets as possible. Thus, some efforts have been made to apply SPL approach to mobile game domain. However, the current work in SPL area does not address properly the mentioned characteristics of the domain, mainly in domain implementation phase. In this paper, we described a practical approach to develop core assets in a SPL providing details at code level and then performed a case study with three mobile games. The case study results have shown that the approach can be suitable for the mobile domain.

As future work, we are planning to evolve this approach and take into consideration the principles of Domain-Specific Languages (DSLs). Moreover, we plan to increase the number of product families and consequently the number of domain restrictions to perform a case study in a more challenging scenario.

## Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[4]), funded by CNPq/FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08 and Brazilian Agency (CNPq process number 475743/2007-5).

## References

Almeida, E. S. (2007) "RiDE: The RiSE Process for Domain. Engineering". Ph.D. Thesis, Brazil.

Almeida, E. S., Santos, E. C. R., Alvaro, A., Garcia, V. C., Lucrédio, D., Fortes, R. P. M., Meira, S. R. L. (2008) "Domain Implementation in Software Product Lines Using OSGi". In: *7th International Conference on Composition-Based Software Systems* (ICCBSS), Spain.

Almeida, E. S., Alvaro, A., Garcia, V.C., Mascena, J.C.C.P., Burégio, V.A.A., Nascimento, L.M., Lucrédio, D., Meira, S.R.L. (2007) "C.R.U.I.S.E: Component Reuse in Software Engineering". C.E.S.A.R e-book, Available on http://cruise.cesar.org.br/, Brazil, accessed in June, 2009.

[4] INES - http://www.ines.org.br

Almeida, E. S., Alvaro, A., Lucredio, D., Garcia, V.C., Meira, S.R.L. (2005) "A Survey on Software Reuse Processes". In: *IEEE International Conference on Information Reuse and Integration* (IRI), USA, IEEE Press.

Alves, V., Cardim, I., Vital, H., Sampaio, P., Damasceno, A., Borba, P., Ramalho, G. (2005) "Comparative Analysis of Porting Strategies in J2ME Games". In: *21st IEEE International Conference on Software Maintenance* (ICSM'05), pp. 123-132.

Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., Borba, P. (2008) "FLiP: Managing Software Product Line Extraction and Reaction with Aspects". In: *12th Software Product Line Conference* (SPLC'2008), Ireland, pp. 354-354.

Anastasopoulos, M., Gacek, C. (2001) "Implementing Product Line Variabilities". In *Symposium on Software Reusability: Putting Software Reuse in Context*, Canada, pp. 109-117, ACM Press.

Atkinson, C., Bayer, J., Muthig, D. (2000) "Component-Based Product Line Development: The KobrA Approach". In: *1st Software Product Line Conference (SPLC)*, USA, pp. 289-309.

Basili, V. R., Caldiera, G., Rombach, H. D. (1994) The Goal Question Metric Approach, *Encyclopedia of Software Engineering*, Vol. 02, pp. 528-532.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J. (1999) "PuLSE: A Methodology to Develop Software Product Lines". In: *Symposium on Software Reusability* (SSR), USA, pp. 122-131.

Clements, P., Northrop, L. (2002) "Software Product Lines: Practices and Patterns". Addison-Wesley, pp. 608.

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

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A. (1990) "Feature-Oriented Domain Analysis (FODA) Feasibility Study". (Technical Report CMU/SEI-90-TR-21), Software Engineering Institute.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.M., Irwin J. (1997) "Aspect-Oriented Programming". In: *11th European Conference on Object-Oriented Programming* (ECOOP 1997), Finland, Lecture Notes in Computer Science 1241, Springer-Verlag, pp. 220-242.

Krueger, C.W. (1992) "Software Reuse". ACM Computing Surveys, Vol. 24, No. 02, pp. 131-183.

McCabe, T. J. (1976) "A Complexity Measure". IEEE Transactions on Software Engineering, pp. 308-320.

Muthig, D., John, I., Anastasopoulos, M., Forster, T., Doerr, J., Schmid, K. (2004) "GoPhone - A software product line in the mobile phone domain". (Technical Report, 025.04/E), Fraunhofer IESE.

Nascimento, L. M., Almeida, E. S., Meira, S. R. L. (2008) "A Case Study in Software Product Lines - The Case of the Mobile Game Domain". In: *34th IEEE Euromicro Conference on Software Engineering and Advanced Applications* (SEAA), Italy.

Nascimento, L. M. (2008) "Core Assets Development in Software Product Lines - Towards a Practical Approach for the Mobile Game Domain". M.Sc. Dissertation, Brazil.

OSGi Alliance (2008) Available on http://www.osgi.org. Accessed in April, 2009.

Pohl, K., Bockle, G., van der Linden, F. (2005) "Software Product Line Engineering: Foundations, Principles and Techniques". Springer, pp. 468.

Product Line Hall of Fame (2009). Available on http://www.sei.cmu.edu/ productlines/plp_hof.html, accessed in May, 2009.

Silva, F. R. C., Almeida, E. S., Meira, S. R. L. (2009) "An Approach for Component Testing and Its Empirical Validation". In: *24th Annual ACM Symposium on Applied Computing* (SAC), USA.

Weiss, D. M., Lai, C. T. R. (1999) "Software Product-Line Engineering: A Family-Based Software Development Process". Addison-Wesley, pp. 426.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., Wesslén, A. (2000) "Experimentation in Software Engineering: An Introduction". Kluwer Academic Publishers, pp. 204.