# A Decision Model for Implementing Product Lines Variabilities

Márcio de M. Ribeiro
mmr3@cin.ufpe.br
Informatics Center - UFPE

Pedro Matos Jr.
poamj@cin.ufpe.br
Informatics Center - UFPE

Paulo Borba
phmb@cin.ufpe.br
Informatics Center - UFPE

## ABSTRACT

Software Product Lines (SPLs) encompass a family of software systems developed from reusable assets. One issue during SPL development is the decision about which technique should be used to implement variabilities and improve the Separation of Concerns (SoC) of the SPL. In this paper, we present an initial decision model based on both qualitative and quantitative analysis to guide developers on choosing suitable techniques for implementing SPL variabilities.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.1 [**Programming Techniques**]: Miscellaneous

## General Terms

Measurement, Experimentation

## Keywords

Software Product Lines, Software Metrics, Modularity

## 1. INTRODUCTION

Software Product Line (SPL) is a promising approach to improve the productivity of the software development process by reducing both cost and time of developing and maintaining increasingly complex systems. However, reasoning about how to combine both core assets and product variabilities is a very challenging task [2]. In addition, selecting the correct techniques to implement these variabilities might have considerable effects on the cost to evolve the SPL.

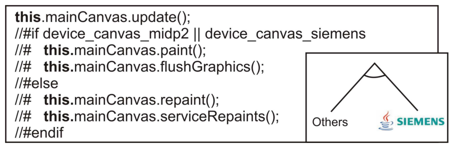
In this context, many studies [1, 2, 3, 4] have analyzed the modularity provided by techniques for implementing variabilities in SPL. Although they provide decision models to select techniques for a given variability, neither address quantitative analysis. In order to mitigate this lack, we provide in this paper a quantitative study for comparing techniques which implement variabilities in SPL.

According to the results of our analysis, we have constructed an initial decision model based on both qualitative and quantitative analysis. Thus, given a kind of variability, it is able to indicate which technique is suitable for implementing it. However, it is preliminar since (i) we have only used SoC, size, and coupling as comparison criteria; and (ii) we must analyze more kinds of variabilities and techniques.

## 2. PRODUCT LINES VARIABILITIES

In this section we discuss kinds of variabilities found in two real SPLs: J2ME Games and Mobile Phone Test Cases.

The first case study consists of a **J2ME Games** product line that can be instantiated for 17 device families. The kind of variability analyzed here is **After method call**. It occurs when some alternative or optional behavior happen after a method call (Figure 1). The *mainCanvas* object represents the area where the screen elements of the game are drawn. Every time the canvas is updated it must also be repainted. The canvas depend on the API supported by the device.



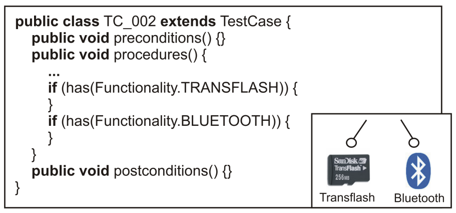
```
this.mainCanvas.update();
//#if device_canvas_midp2 || device_canvas_siemens
//#   this.mainCanvas.paint();
//#   this.mainCanvas.flushGraphics();
//#else
//#   this.mainCanvas.repaint();
//#   this.mainCanvas.serviceRepaints();
//#endif
```

**Figure 1: After method call.**

For this kind of variability, we propose two patterns. The first (**AOP**) relies on *after advice*, where two or more aspects implement the variabilities separately. The **Inheritance** pattern uses the *Decorator* design pattern. For each alternative feature, a decorator is needed.

The second case study analyzed is a set of **Mobile Phone Test Cases** proprietary of Motorola Industrial. The kind of variability discussed here is **End of method body**. Figure 2 depicts two optional features implemented at the end of the *procedures* method. Four instances of the product line are possible: neither *Transflash* nor *Bluetooth* are present in the phone; both *Transflash* and *Bluetooth* are present in the phone; phones with only *Transflash*; phones with only *Bluetooth*. Notice that the order of execution of the steps is important: changing it may break the test case.

We have implemented this variability using three patterns. The **Inheritance** and **Mixins** patterns consist of overriding the *procedures* method. Each subclass overrides it and calls the super method followed by the specific concern code (*Transflash* or *Bluetooth*). The **AOP** pattern defines two



```
public class TC_002 extends TestCase {
    public void preconditions() {}
    public void procedures() {
        ...
        if (has(Functionality.TRANSFLASH)) {
        }
        if (has(Functionality.BLUETOOTH)) {
        }
    }
    public void postconditions() {}
}
```

**Figure 2: End of method body.**

aspects (one for each feature). In addition, an extra aspect to declare the precedence among the features is needed.

## 3. EVALUATION

In this section, we evaluate our patterns quantitatively based on some metrics detailed in [5]. At the end, we outline our decision model according to the obtained results.

The metrics for the **After method call** patterns are depicted in Table 1. The $CDC$ metric shows that both **Inheritance** and **AOP** patterns provide a suitable SoC. However, the size and coupling metrics show that the **AOP** pattern provides a smaller ($LOC_{AOP} < LOC_{Inherit.}$; $VS_{AOP} < VS_{Inherit.}$) and less coupled ($CBC_{AOP} < CBC_{Inherit.}$) and scattered ($CDC_{AOP} < CDC_{Inherit.}$) solution. Size metrics are higher for the **Inheritance** pattern since it requires an additional interface and a factory class for instantiating the correct interface implementation. For this reason, the sum of both $CDC$ and $CBC$ metrics also increases. In contrast, the **AOP** pattern does not require a factory class, since aspects are implicitly instantiated by the weaver.

| After method call | | Inheritance | AOP |
|---|---|---|---|
| CLOC | Paint | 11 | 11 |
| | Repaint | 11 | 11 |
| CDC | Paint | 2 | 1 |
| | Repaint | 2 | 1 |
| LOC | Commonalities | 476 | 450 |
| | Variabilities | 22 | 22 |
| VS | | 6 | 4 |
| CBC | | 11 | 3 |

**Table 1: After method call patterns.**

Table 2 illustrates the metrics for the **End of method body** patterns. As illustrated, the **Inheritance** pattern does not enable feature compositions suitably: for phones with both *Transflash* and *Bluetooth*, it is necessary to create a new class which duplicates the source code of the two classes responsible for implementing each feature separately. $CLOC$ and $LOC$ show how the amount of lines responsible for implementing the variabilities is bigger when comparing to the **Mixins** and **AOP** patterns.

Although the metrics show that the **Mixins** pattern is slightly better, it occurs only for two features. If one more feature (*Infrared*) is added, the number of classes increases significantly when using the **Inheritance** and **Mixins**. In this case, instead of defining one class, the developer must implement four: *Transflash*/*Bluetooth*, *Transflash*/*Infrared*, *Bluetooth*/*Infrared* and *Transflash*/*Bluetooth*/*Infrared*. The $NCC$ metric is the same for all patterns: *Class*, *Mixins*, and *Extra Aspect* implement the *Transflash* and *Bluetooth* concerns. However, in the *Infrared* scenario, both **Inheritance** and **Mixins** patterns would have four classes where $NCC >$ 1. Thus, they do not separate the concerns. The $CDC$ and $VS$ metrics also increase in this scenario: $CDC_{Transflash} = CDC_{Bluetooth} = CDC_{Infrared} = 4$ and $VS = 8$.

| End of method body | | Inheritance | Mixins | AOP |
|---|---|---|---|---|
| CLOC | Bluetooth | 36 | 36 | 36 |
| | Transflash | 21 | 21 | 21 |
| | Both | 44 | 2 | 4 |
| CDC | Bluetooth | 2 | 2 | 2 |
| | Transflash | 2 | 2 | 2 |
| LOC | Commonalities | 82 | 82 | 82 |
| | Variabilities | 101 | 59 | 61 |
| NCC | | 2 | 2 | 2 |
| VS | | 4 | 4 | 4 |
| CBC | | 3 | 4 | 4 |

**Table 2: End of method body patterns.**

The **AOP** pattern does not have such scalability problem because of the *Extra Aspect* which declares the precedence of the features (only this component have $NCC > 1$). Whenever a new feature must be considered, the aspect for that feature is written, and the existing precedence aspect is modified to take the new feature into consideration ($VS = 5$). The $CDC$ metric remains the same for all features.

Our decision model is finally outlined in Table 3.

| Kinds of Variabilities | | Technique |
|---|---|---|
| After method call, Whole method body | | AOP |
| Method parameter | | Config. Files |
| End of method body | 2 features | Mixins or AOP |
| | 3 or more | AOP |

**Table 3: Our Preliminar Decision model.**

## 4. CONCLUDING REMARKS

This paper presented a preliminary decision model based on both qualitative and quantitative analysis. Such model might help SPL developers when selecting techniques for implementing a given variability at source code level.

As future work, we intend to use performance and scalability in our model. Further, we will consider more kinds of variabilities and techniques to implement them.

## 5. REFERENCES

[1] V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the SPLC'05*, volume 3714 of *LNCS*, pages 70–81. Springer-Verlag, September 2005.

[2] M. Anastasopoulos and C. Gacek. Implementing Product Line Variabilities. In *Proceedings of the SSR'01*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[3] J. Coplien. *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, Etterbeek, Belgium, July 2000.

[4] T. Patzke and D. Muthig. Product Line Implementation Technologies. Technical Report 057.02/E, Fraunhofer Institut Experimentelles Software Engineering, October 2002.

[5] C. Santánna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: A Assessment Framework. In *Proceedings of the SBES'03*, pages 19–34, October 2003.