A Decision Model for Implementing Product Lines Variabilities

Márcio de M. Ribeiro
mmr3@cin.ufpe.br
Federal University of Pernambuco
Informatics Center
Recife, Pernambuco, Brazil

Pedro Matos Jr.
poamj@cin.ufpe.br
Federal University of Pernambuco
Informatics Center
Recife, Pernambuco, Brazil

Paulo Borba
phmb@cin.ufpe.br
Federal University of Pernambuco
Informatics Center
Recife, Pernambuco, Brazil

ABSTRACT

Software Product Lines (SPLs) encompass a family of software-intensive systems developed from reusable assets. One major issue during SPL development is the decision about which technique should be used to implement variabilities aiming at improving the Separation of Concerns (SoC) of the SPL. Although many works have addressed such issue by providing decision models to select techniques for implementing a given variability, their models are incomplete or inconsistent, since they do not provide quantitative analysis in their studies. In this paper, we have constructed an initial decision model based on both qualitative and quantitative analysis. It was constructed using variabilities found in two real SPLs and the results show that it might be helpful when considering variabilities of fine-grained source code.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.1 [**Programming Techniques**]: Miscellaneous—*Aspect-Oriented Programming, Mixin layers*

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 [10]. Such approach relies on core assets (representing the common artifacts present in products) and variabilities (representing the differences among the products). 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, 4, 5, 12] have analyzed the modularity provided by techniques for implementing variabilities in SPL. Although such studies provide decision models aiming at selecting techniques for implementing a given variability, neither address quantitative analysis, which means that only qualitative analysis are available. Notice that many researchers [7, 11, 13] might consider the aforementioned analysis incomplete or inconsistent. This way, in order to mitigate this lack, we provide in this paper a quantitative study for comparing techniques which implement variabilities in SPL. Our analysis focuses on Separation of Concerns (SoC), size, and coupling metrics.

Based on the results of our analysis, we have constructed an initial decision model (representing our main contribution). Thus, given a kind of variability, it is able to indicate which technique is suitable for implementing it. Differently of other approaches, our model is based on both qualitative and quantitative analysis. On the other hand, 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 to address them.

Our methodology is explained as follows. First, we describe kinds of variabilities encountered in two real and non-trivial SPLs of different domains (J2ME Games and Mobile Phone Test Cases). Given the different natures of the domains in which they were found, we believe that such variabilities are likely to occur in many other domains as well. Next, we use some patterns to address the implementation of these variabilities aiming at improving their modularity. Notice that such patterns are based on many techniques. Afterwards, we apply some software metrics in order to compare which pattern (i.e., technique) provided better modularity. Next, we analyze the advantages and disadvantages of the proposed techniques. Finally, we adapt the decision model to encompass such a new information.

The techniques analyzed in this work are: Aspect-Oriented Programming (AOP) [9], Inheritance using design patterns [6], Mixins [3], and Configuration Files.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

2. PRODUCT LINE VARIABILITIES

In this section we discuss some kinds of variabilities found in two real and non-trivial SPLs: J2ME Games and Mobile Phone Test Cases. The first handles the variabilities using conditional compilation, whereas the second handles them using *if-else* statements. By the presence of duplicated code and concerns not modularized, we concluded that such techniques are not suitable for implementing those variabilities.

The variabilities considered in this work are code-centric and their names are based exclusively on the locality of them in the source code. This way, it might be helpful on the task of finding the variability and even of choosing more easily which technique to use when handling variabilities at the source code level.

2.1 J2ME Games

The first case study consists of a mobile game product line which can be instantiated for 17 device families and 6 different languages.

The **Whole method body** is the first kind of variability discussed in this work. It occurs when the whole body of a method differs among product line instances. Figure 1 illustrates an example of such variability. The *playSound* method is invoked whenever a sound is to be played. The block of code that actually plays the desired sound varies depending on the phone sound API (alternative feature[1] depicted in Figure 1).

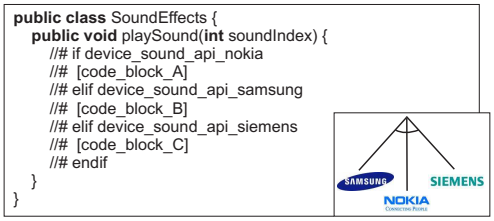

Figure 1: Whole Method body.

The **After method call** variability occurs when some alternative or optional behavior might happen after a method call. Figure 2 shows an example of such variability. 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 repaint depend on the graphics API supported by the device.

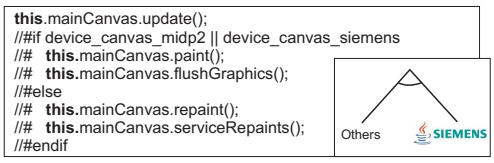

Figure 2: After method call.

2.2 Mobile Phone Test Cases

The second case study analyzed is a set of Mobile Phone Test Cases proprietary of Motorola Industrial. The test cases are part of a complex system that has the capability of testing families of Motorola mobile phones software.

[1]Alternative features are represented as an open arc.

The first kind of variability discussed here (**Method parameter**) occurs whenever the value of a method parameter differs according to the selected product. Figure 3 shows that the parameter of the *loadWebSession* method varies depending on the currently installed browser (alternative feature depicted in Figure 3: either *Opera* or a *Motorola* browser is selected in the product line instance).

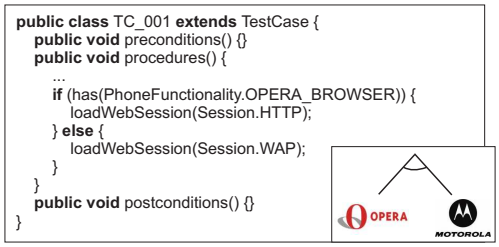

Figure 3: Method parameter.

The next kind of variability consists of an **End of method body**. Figure 4 illustrates two optional features[2] implemented at the end of the *procedures* method. This way, four instances of the product line are possible: (i) neither *Transflash* nor *Bluetooth* are present in the phone; (ii) both *Transflash* and *Bluetooth* are present in the phone; (iii) phones with only *Transflash*; (iv) phones with only *Bluetooth*. Notice that the order of execution of the steps is important: changing it might break the test case.

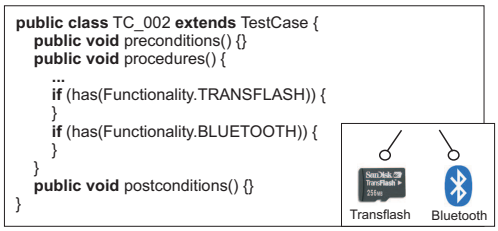

Figure 4: End of method body.

3. PATTERNS

In this section, we provide some patterns aiming at implementing the kinds of variabilities discussed in Section 2.

3.1 Whole method body

We propose two patterns for this kind of variability.

The **AOP** pattern relies on *inter-type* declarations. This way, a set of aspects become responsible for introducing the missing method (*playSound*) into the class, where each aspect introduces a different version of such method (*Samsung*, *Nokia*, or *Siemens*, according to Figure 1). Thus, exactly one of these aspects must be present on each product instance.

The second pattern (**Inheritance**) uses the classical *Strategy* design pattern [6].

3.2 After method call

Two patterns are proposed for this kind of variability.

The **AOP** pattern relies on *after advice*, where two or more aspects implement the variabilities separately. In the

[2]Optional features are represented as open circles.

particular case of Figure 2, two aspects are required: one for each alternative feature.

The **Inheritance** pattern uses the *Decorator* design pattern [6]. For each alternative feature, a decorator is needed.

3.3 Method parameter

For this kind of variability, we propose two patterns.

The first (**AOP**) relies on *inter-type* declarations. This way, two aspects implement the method parameter's values for each browser separately. Therefore, the test case uses the constants introduced by the aspects as parameters of methods, eliminating the *if-else* statements of the test. In Figure 3, the *loadWebSession* method would use as parameter the constant introduced by the aspect.

The **Configuration Files** pattern uses a file to provide the method parameter's values. In order to load the file, the test case must use some object responsible for reading it.

3.4 End of method body

We have implemented this variability using three patterns.

The **Inheritance** and **Mixins** patterns consist of overriding the *procedures* method (Figure 4). Therefore, two classes inherit from *TC_002*. Each class overrides the aforementioned method and calls the super method followed by the specific concern code (*Transflash* or *Bluetooth*).

The **AOP** pattern defines two aspects aiming at crosscutting the *procedures* method using an *after advice*. In addition, an extra aspect to declare the precedence among the features is needed.

4. METRICS

The metrics used in this work are presented here. Such metrics are based on [13]. They are divided in *SoC*, *size*, and *coupling*.

For SoC, we used the *Concern Diffusion over Components* (CDC) metric to measure concern diffusion. It counts the number of components which contributes to the implementation of a concern. We also measure the number of lines of code whose main purpose is to implement a concern, which is represented by the *Concern Lines of Code* (CLOC) metric. Moreover, we used *Number of Concerns per Component* (NCC) metric to count, for each component, the number of concerns it implements.

For size metrics, we considered *Lines of Code* (LOC) and *Vocabulary Size (VS)*, which count the number of source code lines (ignoring comments and blank lines) and the number of system components (classes, aspects, interfaces, and configuration files), respectively.

To measure coupling, we considered the *Coupling between Components* (CBC) metric, which counts, for each component, the number of other components to which it is coupled. This metric considers coupling dimensions in AOP: accesses to aspect methods and attributes defined by inter-type declarations, and the relationships between aspects and classes or other aspects defined in the pointcuts. We also used the *Depth of Inheritance Tree* (DIT) metric to measure coupling. This metric counts how far down the inheritance hierarchy a class or aspect is declared.

5. EVALUATION

In this section, we evaluate our patterns quantitatively based on the metrics discussed in Section 4. In addition, we also discuss the advantages and disadvantages of each pattern, since our decision model is based on qualitative analysis as well. At the end, we illustrate our decision model instantiated according to the obtained results.

5.1 Patterns evaluation

The metrics for **Whole method body** patterns are shown in Table 1. The CBC value shown in the table represents the sum of CBC metric values from all components. The CDC metric illustrates that both **Inheritance** and **AOP** patterns provide a suitable SoC. However, size and coupling metrics show that the **AOP** pattern provides a smaller ($LOC_{AOP} < LOC_{Inheritance}$; $VS_{AOP} < VS_{Inheritance}$) and less coupled ($CBC_{AOP} < CBC_{Inheritance}$) and scattered ($CDC_{AOP} < CDC_{Inheritance}$) solution.

Whole method body		Inheritance	AOP
CLOC	Siemens	44	41
	Nokia	37	36
	Samsung	32	31
CDC	Siemens	2	1
	Nokia	2	1
	Samsung	2	1
LOC	Commonalities	86	57
	Variabilities	113	108
VS		6	4
CBC		9	3

Table 1: Whole method body patterns.

A similar conclusion of the **After method call** patterns is illustrated in Table 2. The comparison of **Inheritance** and **AOP** patterns for both **Whole method body** and **After method call** variabilities shows that the patterns differ on coupling and size metrics. Size metrics is higher for **Inheritance** patterns, because they require an additional interface and a factory class, in order to instantiate the correct interface implementation. For this reason, the sum of both CDC and CBC metrics also increases.

Despite it is not represented in the source code, there is an implicit interface between aspects and the base code. If we consider such interface, the results of the metrics would change. Notice that we did not considered these interfaces since the CBC metric calculates only explicit artifacts. In fact, interfaces between aspects and classes might be explicitly defined [8], but they are out of the scope of this work. Moreover, the **AOP** patterns do not require a factory class, because aspects are implicitly instantiated by the weaver. The developer only have to choose the correct set of aspects for each product build. This work may be automated by build tools and scripts.

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 2: After method call patterns.

Table 3 illustrates the results of the metrics for the **Method**

parameter patterns. Two patterns are analyzed: **AOP** and **Configuration Files**.

Both patterns provide a suitable SoC since the variabilities are addressed by aspects and files, which means that the test case no longer handles variability using *if-else* statements.

The *LOC* necessary to implement the *Variabilities* using the **AOP** pattern is almost two times bigger when comparing to the **Configuration Files** pattern (notice that $LOC_{Variabilities} = CLOC_{Opera} + CLOC_{Motorola}$). The *LOC* (for *Commonalities*) and *CBC* metrics of the **AOP** pattern are slightly smaller, since the **Configuration File** pattern relies on an additional class for loading the file. However, it is important to note that such infrastructure (for example, an instance of the *java.util.Properties* class) is not susceptible to change frequently.

Method parameter		AOP	Conf. Files
CLOC	Opera	7	4
	Motorola	7	4
CDC	Opera	1	1
	Motorola	1	1
LOC	Commonalities	72	74
	Variabilities	14	8
VS		3	3
CBC		2	3

Table 3: Method parameter patterns.

Table 4 illustrates the metrics for the **End of method body** patterns. For these patterns, consider that *DIT* represents the maximum *DIT* value found among all components. 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 (Figure 5). In other words, the pattern does not address SoC adequately. *CLOC* and *LOC* show how the amount of lines responsible for implementing the variabilities is bigger when comparing to the **Mixins** and **AOP** patterns.

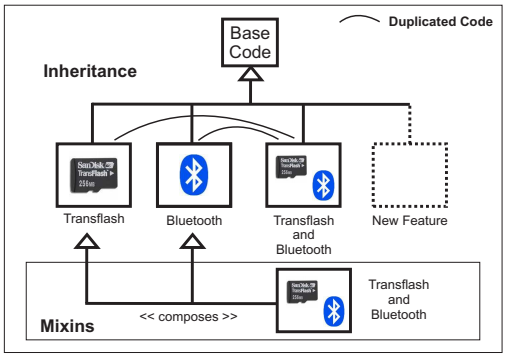

Figure 5: Inheritance *versus* Mixins.

Although the metrics show that the **Mixins** pattern is slightly better than the **AOP** (see *CLOC* and *LOC* metrics), we concluded that it occurs only for two features. If we consider one more feature (*Infrared*, for example), the number of classes and compositions increases significantly when using the **Inheritance** and **Mixins** patterns, being

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
NCC	Class	2	-	-
	Mixins	-	2	-
	Extra Aspect	-	-	2
LOC	Commonalities	82	82	82
	Variabilities	101	59	61
VS		4	4	4
CBC		3	4	4
DIT		1	2	-

Table 4: End of method body patterns.

hard to maintain the whole application. In this case, instead of defining only one class for features composition, the developer must implement four: *Transflash & Bluetooth*, *Transflash & Infrared*, *Bluetooth & Infrared* and *Transflash & Bluetooth & Infrared*. Notice that because of the last case the *DIT* metric would be 3 for the **Mixins** pattern. The *NCC* metric is the same for all patterns: *Class*, *Mixins*, and *Extra Aspect* implement the *Transflash* and *Bluetooth* concerns ($NCC = 2$). However, in the *Infrared* scenario, both **Inheritance** and **Mixins** patterns would have four classes where $NCC > 1$. In other words, these four components do not separate the three aforementioned concerns. The *CDC* and *VS* metrics also increase in this scenario: before *Infrared*, $CDC_{Transflash} = CDC_{Bluetooth} = 2$ and $VS = 4$; after *Infrared*, $CDC_{Transflash} = CDC_{Bluetooth} = CDC_{Infrared} = 4$ and $VS = 8$.

The **AOP** pattern does not have such scalability problem because of the *Extra Aspect* responsible for declaring the precedence among 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 (in the *Infrared* scenario, $VS = 5$). The *CDC* metric remains the same for all features.

5.2 Decision Model

Our decision model is outlined in Table 5. The model is preliminar since we have only used SoC, size, and coupling as comparison criteria. Performance, traceability, and maintainability will be addressed in future works. Besides, we need to analyze more kinds of variabilities and techniques to implement them.

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

Table 5: Our Preliminar Decision model.

6. RELATED WORK

[1] proposes a method to address the creation and evolution of SPLs focusing on the implementation and feature

level. The method first bootstraps the SPL and then evolves it with a reactive approach. Such a method relies on a collection of provided refactorings at both the code and feature model levels. Although this work provides a framework for comparing variability implementation techniques, it does not encompass quantitative analysis. On the other hand, the work considers additional techniques such as frames and program transformation systems.

[2] claims that little attention has been given on how to deal with variabilities in SPLs at the source code level. To this end, they examine various implementation approaches with respect to their use in a product line context. However, this work provides only a qualitative analysis. Nevertheless, the work compares the techniques using attributes such as SoC, traceability, scalability, and binding time.

[4] proposes a method called Multi-Paradigm Design. This method consists of analyzing commonalities and variabilities of a SPL and implementation techniques. Our approach differs from his work because we rely on a set of implementation patterns that are code-centric and more fine-grained than the domain analysis solution that he proposes, allowing SPL developers to choose more easily which technique to use when handling variabilities at the source code level. Moreover, he does not consider metrics in his study.

[7, 11, 13] use software metrics for comparing AOP with other techniques regarding the modularity and maintainability provided by them. However, they do not analyze the impact of these techniques when implementing variabilities in SPLs. The metrics used here is based on these works.

7. 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. Thus, given a kind of variability, our model is able to indicate which technique is suitable for implementing it.

In order to construct such model, we have analyzed two real and non-trivial SPLs of different domains (J2ME Games and Mobile Phone Test Cases). In both, we have found common kinds of variabilities at source code level. Since they are from completely different domains, we believe that such variabilities should be present in other domains as well. Next, we provided patterns aiming at implementing such variabilities. Advantages and disadvantages as well as a quantitative study (based on SoC, size, and coupling metrics) of the patterns were presented. According to the results, we adapted the decision model to encompass the best patterns.

As future works, we intend to use performance, traceability, maintainability, and binding time in our decision model. Additionally, we need to take into consideration more kinds of variabilities and techniques to implement them.

8. 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 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer-Verlag, September 2005.

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

[3] CaesarJ. CaesarJ Project, 2007. Last visit: July, 2007.

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

[5] M. de Medeiros Ribeiro, P. M. Jr., P. Borba, and I. 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)*, October 2007. To appear.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[7] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 36–74. Springer, 2006.

[8] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242, 1997.

[10] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society.

[11] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of the 22th IEEE International Conference on Software Maintenance (ICSM'06)*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

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

[13] 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 17th Brazilian Symposium on Software Engineering (SBES'03)*, pages 19–34, October 2003.