On the Modularity of Aspect-Oriented and Other Techniques for Implementing Product Lines Variabilities

Márcio de Medeiros Ribeiro[1], **Pedro Matos Jr.**[1], **Paulo Borba**[1], **Ivan Cardim**[1]

[1] Informatics Center – Federal University of Pernambuco
Caixa Postal 7851, 50740-540 – Recife – PE – Brazil

{mmr3,poamj,phmb,icc2}@cin.ufpe.br

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. Although Aspect-Oriented Programming (AOP) has been used to this purpose, we still need to identify in which situations it is suitable or not. We propose a catalog of variabilities at the source code level and a set of patterns for implementing them. After analyzing these patterns, we concluded that AOP is not the best technique to implement some kinds of variabilities from our catalog. We argue that our approach is useful to find the most suitable techniques to handle specific variabilities at the source code level.*

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 [Kolb et al. 2005]. Such an 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 [Anastasopoulos and Gacek 2001]. In addition, selecting the correct techniques to implement these variabilities might have considerable effects on the cost to evolve the SPL.

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] is a well known technique for implementing crosscutting concerns through a unit of modularity called aspect. Although AOP has been used to implement variabilities in SPLs [Alves et al. 2005], [Anastasopoulos and Gacek 2001] and [Anastasopoulos and Muthig 2004] argue that AOP is not always suitable for this task. Furthermore, recent empirical studies [Kulesza et al. 2006] have confirmed that AOP provides not only positive, but also negative effects on typical maintenance activities. In this context, a SPL developer must know whether AOP is suitable or not for implementing specific kinds of variabilities.

In this paper, we provide a preliminary catalog of kinds of variabilities. Such a catalog is code-centric, allowing SPL developers to choose more easily which technique to use when handling variabilities at the source code level. We then define patterns to address the implementation of these variabilities. For each of these patterns, we discuss the advantages and disadvantages with respect to the modularity that it provides. In this way, since each pattern uses a different technique, we are able to compare and discuss whether AOP might be a benefit for those SPL developers. The techniques used in this work are AOP, inheritance, mixins, and configuration files.

To derive our catalog and patterns, we analyzed two real and non-trivial SPLs of different domains (J2ME Games and Mobile Phone Test Cases). In both, we found the same kinds of variabilities, suggesting that they might be present in other domains. Afterwards, we have implemented these variabilities using the four aforementioned techniques. Notice that we analyzed not only source code, but also test cases. In this way, we observed that our catalog and patterns seem to be useful for both.

Our work is similar to others [Patzke and Muthig 2002, Tirila 2002, Coplien 2000], but with two main differences (representing our contributions):

- We present a code-centric catalog of kinds of variabilities and patterns to implement them. Such patterns are the first step towards defining a systematic decision model, which means that given a variability, we will be able to decide systematically which technique should be used for implementing it (Sections 2 and 3);
- When considering modularity, our work shows where in the source code AOP is suitable for implementing variabilities in SPLs. The results are presented in what follows: (i) AOP might be suitable, in accordance to [Alves et al. 2005]; (ii) AOP makes no difference when compared to other techniques; and (iii) AOP is not suitable (Section 4).

2. Towards a variabilities catalog

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 one handles the variabilities using conditional compilation[1], whereas the second handles them by using *if-else* statements. By the presence of duplicated code (present in the Mobile Phone Test Cases) and concerns not modularized (present in both), we concluded that such techniques are not sufficient for implementing those variabilities adequately.

In addition, we provide a preliminary catalog of kinds of variabilities extracted from those SPLs. 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.

2.1. J2ME Games

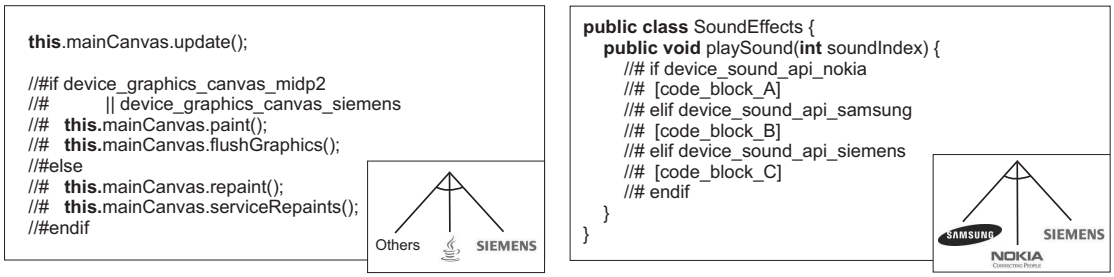
The first case study consists of a mobile game product line which can be instantiated for 17 device families and 6 different languages. The number of total possible instances for this product line is 152. The original implementations analyzed use only one technique (conditional compilation) to handle variabilities. This technique uses a special comment symbol (//#), indicating that a code line will be preprocessed. The code snippet to be compiled depends on variables used in *if directives*.

The **After method call** is the first kind of variability discussed in this work. It occurs when some alternative or optional behavior might happen after a method call. Figure 1(a) 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 object is updated it must also be repainted. However, the sequence of methods invoked to repaint the canvas depends on the graphics API supported by the specific device (alternative feature[2]

[1]Technique that uses a source code preprocessor to conditonally compile blocks of code.

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

depicted in Figure 1(a): MIDP 2.0, Siemens, or other should be selected in the product line instance).



(a) After method call. (b) Whole Method body.

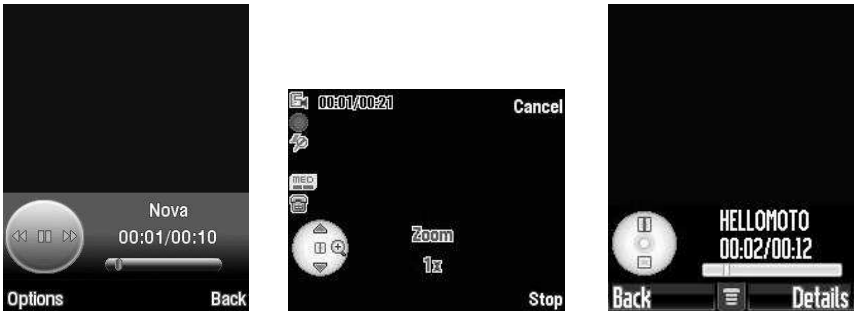
Figure 1. Variabilities found in the J2ME Games.

The **Whole method body** variability occurs when the whole body of a method differs among product line instances. Figure 1(b) illustrates an example of such variability. The *playSound* method is invoked whenever a sound is to be played. The parameter *soundIndex* holds the identifier of the sound to be played. The block of code that actually plays the desired sound varies depending on the sound API provided by the phone. This variability exists because phone manufacturers provide different proprietary APIs to deal with sounds. The alternative sound APIs are depicted in Figure 1(b). Only one sound API can be used in each product instance.

2.2. Mobile Phone Test Cases

The second case study analyzed in this work 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.

Figure 2 illustrates three Motorola mobile phones. Notice that *Phone A* does not provide the *stop* button whereas *Phone B* and *Phone C* do. On the other hand, *Phone B* provides the *camera landscape view*, differently from *Phone A* and *Phone C*.

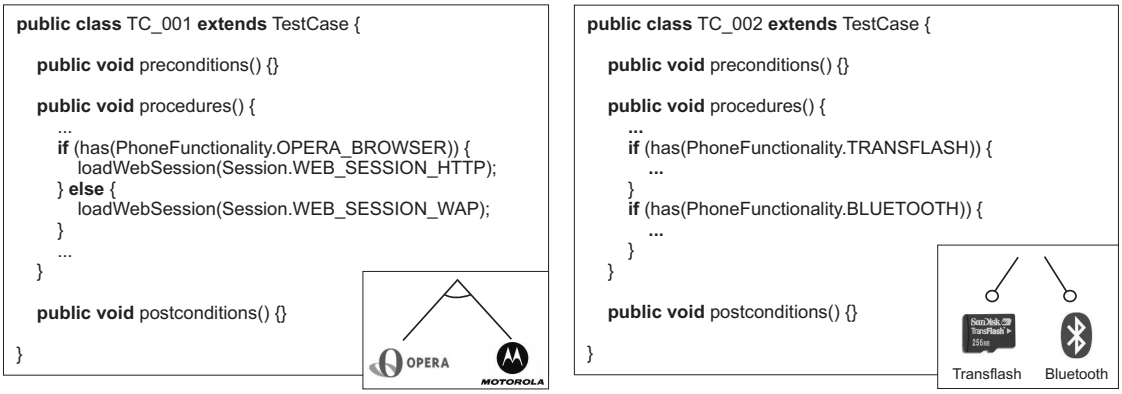


Phone A Phone B Phone C

Figure 2. Variabilities on Motorola Phones.

The first kind of variability discussed here concerns **Constant values**. It occurs whenever the value of a constant differs according to the selected product. Figure 3(a) illustrates an example of this situation. The parameter of the *loadWebSession* method varies

depending on the currently installed browser (alternative feature depicted in Figure 3(a): either *Opera* or a proprietary *Motorola* browser is selected in the product line instance). Although it is not shown, the *if-else* statement is crosscutting throughout the test case.



(a) Constant values. (b) After method execution.

Figure 3. Variabilities found in the Motorola Mobile Phone Test Cases.

The next kind of variability consists of an **After method execution**. Our example (Figure 3(b)) illustrates two optional features[3] implemented at the end of the *procedures* method. In 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 (in this case, features) is extremely important: changing it might break the test case. For example: suppose a test case that sets the alarm clock on a phone. The first step is to access the alarm application, and the second is to set it up. You can not do this in the reverse order.

2.3. Variabilities catalog

Due to space restrictions, we will not show in detail all the kinds of variabilities found in the SPLs analyzed (some of them are outlined in Table 1). Notice that the catalog is code-centric[4], being useful for SPL developers on the task of choosing a particular technique for a specific variability at the source code level.

Originally, all of these variabilities were implemented using conditional compilation and *if-else* statements. However, we have observed serious problems concerning modularity and duplicated code. The next section presents the patterns we created to address these problems. The construction of the patterns was based on many techniques: AOP, inheritance, mixins, and configuration files. This way, we are able to compare AOP with such techniques when implementing our kinds of variabilities in SPLs, allowing us to discuss the suitability of aspects in this context.

[3]Optional features are represented as open circles.

[4]In this case, we mean variabilities reflected in the source code.

Variability	J2ME Games	Motorola Framework
After / Before method call	X	X
After / Before method execution	X	X
Constant values	X	X
Conditional wrapping of method body	X	X
Whole method body	X	X
Interface implementation	X	-

Table 1. Preliminary catalog of kinds of variabilities.

3. Patterns for Implementing Variabilities

In this section, we provide some patterns[5] aiming at implementing[6] the kinds of variabilities previously discussed. We will not use conditional compilation and *if-else* statements in our patterns, since neither technique provides Separation of Concerns (SoC) when considering SPLs: there is no clear separation between core assets and variabilities.

3.1. After method call

Two patterns are proposed for this kind of variability:

- **AOP:** this implementation relies on *after advice*. Two or more aspects implement the variabilities separately. In the particular case of Figure 1(a), two aspects are required: one for each alternative feature;
- **Inheritance:** relies on the *Decorator* design pattern [Gamma et al. 1995]. For each alternative feature, a decorator is needed.

Both pattern implementations of this variability provide a better SoC when compared to the original conditional compilation implementation (Figure 1(a)). However, in both we observed a dependency between the modules that implement the variability alternatives (aspects or decorators) and the public interface of the class which contains the method to be intercepted. We observed that the AOP implementation is more modular because the aspects depend only on the signature of the advised method. On the other hand, the decorator pattern depends on the whole public interface of the decorated class.

3.2. Whole method body

We propose two patterns for this kind of variability: one using AOP and the other using inheritance. We explain and compare these approaches below:

- **AOP:** to implement this kind of variability with AOP, the method declaration is omitted from the class body. A set of aspects become responsible for introducing the missing method into the class using an *inter-type* declaration, where each aspect introduces a different version of such method (*Samsung*, *Nokia*, or *Siemens*, according to Figure 1(b)). Thus, exactly one of these aspects must be present on each product instance;
- **Inheritance:** relies on the classical *Strategy* design pattern [Gamma et al. 1995].

[5]Please, refer to `http://www.cin.ufpe.br/~poamj/patterns` for implementation details. Notice that such codes are proprietary, which means that only code snippets are provided.

[6]Our implementations are based on AspectJ for AOP and CaesarJ for Mixins.

One issue of the AOP implementation pattern is that, using the Java compiler, the base code might not compile in the absence of aspects because of the missing method. An alternative to implement this kind of variability with AOP is to create a stub method on the base class and use an *around advice* on the execution of this method to introduce the variability code. The base code would not depend syntactically on the aspects, however a semantic dependency [de Medeiros Ribeiro et al. 2007] will still exist: an aspect must be present to implement the alternative variability, or the base code will be semantically incomplete.

The inheritance pattern can implement this kind of variability in a modular way. We propose the use of a well known design pattern which proved to be useful to handle this kind of variability.

[Hannemann and Kiczales 2002] proposed an implementation of the *Strategy* design pattern using AOP. However, further studies showed that although this implementation leads to better SoC, it can also lead to worse results in other attributes, such as size, coupling and cohesion [Garcia et al. 2005].

3.3. Constant values

Figure 3(a) illustrates the *Opera Browser* concern tangled with respect to the test case. For this variability, we propose two patterns:

- **AOP:** this implementation relies on *inter-type* declarations. This way, two aspects implement the constants' values (*WEB_SESSION_HTTP* and *WEB_SESSION_WAP*) for each browser separately. Therefore, the test case uses the constants introduced by the aspects as arguments of methods, eliminating the *if-else* statements of the test. In the particular case of Figure 3(a), the *loadWebSession* method would use a constant introduced by the aspect;
- **Configuration Files:** this implementation relies on a configuration file to provide the constants' values. Notice that in order to load the file, the test case must use some object responsible for reading it.

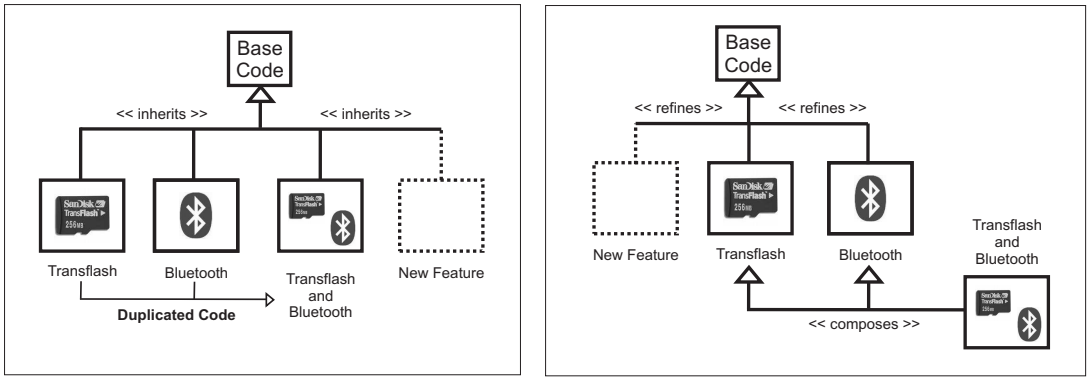
The AOP and Configuration Files approaches provide a suitable SoC: the variability in this case is implemented through aspects and files. The test case no longer handles variability using *if-else* statements. On the other hand, the main disadvantage of such approaches is the additional knowledge required of the tester regarding those new artifacts. In other words, the test now depends on aspects and external files to work correctly.

3.4. After method execution

We have implemented this variability using three patterns. The details are presented as follows:

- **Inheritance:** this implementation consists of overriding the *procedures* method (Figure 3(b)). Therefore, two classes inherits from *TC_002*. Each class overrides the aforementioned method and calls the super method followed by the specific concern code (*transflash* or *bluetooth*);
- **Mixins:** analogous to inheritance;
- **AOP:** two aspects are created aiming at crosscutting the *procedures* method using an *after advice*.

We have found problems in all three implementations. Inheritance does not enable feature compositions suitably: for phones with *transflash* and *bluetooth* it is necessary to create a new class which duplicates the source code of the two classes previously created. In addition, if we consider more than two features, the number of classes and compositions increases significantly (Figure 4(a)), being more difficult to maintain the whole application. In spite of mixins solving the problem of duplicated code when considering features composition, the increase on the number of classes and compositions problem remains (Figure 4(b)).



(a) After method execution using Inheritance.

(b) After method execution using Mixins.

Figure 4. After method execution: Inheritance x Mixins.

The AOP implementation relies on two aspects for implementing each feature, plus an extra aspect to declare the precedence among the features. This approach provides feature composition. Whenever a new feature must be considered, the aspects for that feature are written, and the existing precedence aspect is modified to take the new feature into consideration.

4. Evaluation

Our evaluation focuses on software modularity. In order to assess the modularity of our patterns, we have used Design Structure Matrixes (DSMs) [Baldwin and Clark 2000]. In this section, we show an evaluation whenever the AOP pattern is not the best solution.

4.1. Background

Design Structure Matrixes (DSMs) are used to visualize dependencies among *design parameters*. These parameters correspond to any decision that needs to be made along the product design.

Design parameters may have different abstraction levels. In software engineering, some design decisions are related to process development, language, code/architectural style, and so forth. Moreover, if we consider implementation as design activities, software components like classes, interfaces, packages, and aspects should be represented as design parameters.

The notion of dependency arises whenever a design parameter depends on another. When considering DSMs, each design parameter appears both in the row headings and the columns headings of the matrix.

Figure 5 represents software components as parameters in a DSM. A mark in row B, column A represents that component B depends on component A. In the same way a X in row A, column B represents that component A depends on component B. Whenever this mutual dependency occurs, we have an example of *cyclical dependency*, which implies that both components can not be independently addressed. In other words, their parallel development is compromised.

	A	B	C
A		x	
B	x		x
C			

Figure 5. Example of dependencies in a DSM.

4.2. Evaluating implementation patterns using DSMs

In order to compare the patterns, a DSM was constructed for each one. The notion of dependency considered in this paper consists of explicit references between classes and aspects. Moreover, we also consider *Semantic Dependencies* [Neto et al. 2007]. Such dependencies are not syntactically defined in the code, so that there is no explicit reference between classes and aspects.

Our previous work [de Medeiros Ribeiro et al. 2007] concluded that AOP constructs aimed to support crosscutting modularity might actually break class modularity. This basically occurs by the existence of semantic dependencies.

Figure 6(a) and 6(b) regard the **Whole method body** variability. Figure 6(a) illustrates the DSM of the AOP pattern. Notice that there are cyclical dependencies: aspects depend on the *SoundEffects* class to introduce the missing method and the class depends on the aspects to work correctly (the latter is a semantic dependency: there is no explicit reference about the aspect in the class). Therefore, such components can not be independently addressed, which compromises their parallel development. On the other hand, Figure 6(b) shows the DSM for the inheritance pattern. Such implementation does not create cyclical dependencies, allowing parallel development and better modularity.

Design Parameters		1	2	3	4
SoundEffects.java	1		x	x	x
SamsungPlayerAspect.aj	2	x			
NokiaPlayerAspect.aj	3	x			
SiemensPlayerAspect.aj	4	x			

(a) AOP pattern.

Design Parameters		1	2	3	4	5
IPlayer.java	1					
SoundEffects.java	2	x				
SamsungPlayer.java	3	x				
NokiaPlayer.java	4	x				
SiemensPlayer.java	5	x				

(b) Inheritance pattern.

Figure 6. Patterns of the *Whole method body* variability.

Figures 7(a) and 7(b) illustrate the DSMs of the **Constant values** variability. Both are identical and cyclical: in the first, the test case depends on the aspect to introduce the

constants. On the other hand, the aspect references the test case in a pointcut expression. When considering configuration files, the problem of cyclical dependencies remains: developers of both class and configuration file must be aware of each other, for defining attributes names, for example. We do not considered the infrastructure to load the file (for example, an instance of the *java.util.Properties* class) in the DSM, since it is not susceptible to change frequently.

Design Parameters		1	2
TC_001.java	1		x
ConstantsAspect.aj	2	x	

(a) AOP pattern.

Design Parameters		1	2
TC_001.java	1		x
ConfigurationFile	2	x	

(b) Config. Files pattern.

Figure 7. Patterns of the *Constant values* variability.

Table 2 summarizes our conclusions about which pattern to choose according to the kind of variability.

After call		**Whole method body**		**Constant values**		**After execution**		
AOP	Inheritance	AOP	Inheritance	AOP	Conf. Files	AOP	Inheritance	Mixins
X			X	X	X	X		

Table 2. Patterns choices regarding modularity.

5. Related Work

[Anastasopoulos and Gacek 2001] 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 neither a catalog of kinds of variabilities nor patterns as we do. Nevertheless, the work compares the techniques using attributes such as traceability, scalability, and binding time. In contrast, we only analyzed the modularity attribute.

[Alves et al. 2005] 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, the proposed method relies only on AOP refactorings. On the other hand, our work proposes patterns that use different techniques (not only AOP). We have also found some kinds of variabilities to which AOP may not be suitable.

[Coplien 2000] proposes a method called Multi-Paradigm Design. This method consists of analyzing commonalities and variabilities of a SPL (application domain analysis) and implementation techniques (solution domain analysis). Further, the commonalities and variabilities are mapped on the available techniques (solution domain). 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, our patterns make the task of defining refactorings easier, which means that the implementation of a code variability might migrate from one technique to another. His work does not consider AOP.

[Tirila 2002] also provides a study on how to implement variabilities in SPLs. Although he did not use AOP as a technique for implementing variabilities, Frames and Frameworks are considered. In addition, he provides an evaluation based on scope, flexibility, and efficiency.

6. Concluding Remarks

This paper presented a preliminary catalog at source code level of kinds of variabilities extracted from two real and non-trivial SPLs of different domains (J2ME Games and Mobile Phone Test Cases). Since they are from completely different domains, we believe that such variabilities should be present in other domains as well. Such catalog is code-centric, which allows SPL developers to choose more easily which technique to use when handling variabilities at the source code level.

Besides, we provided patterns aiming at implementing such variabilities. These patterns are based on many techniques: AOP, inheritance, mixins, and configuration files. Advantages and disadvantages regarded to modularity provided by each pattern were discussed. In this way, since each pattern uses a different technique, we were able to analyze whether AOP might be a benefit or not for SPLs developers when considering modularity.

Differently of existing works, we have analyzed not only source code, but also test cases. In this way, we observed that our catalog and patterns seem to be useful for both.

Based on DSM analysis, we observed that to implement some kinds of variabilities from our catalog, AOP might not be suitable or makes no difference when compared to other techniques.

As future works, we intend to use software metrics in order to validate our qualitative analysis, calculating the coupling, cohesion, and Net Options Value (NOV) of our patterns' implementations. Afterwards, we should be able to implement a systematic decision model based on both qualitative and quantitative analysis. Additionally, parameters such as binding time, scalability and others will be analyzed.

7. Acknowledgments

We would like to thank CNPq, a Brazilian research funding agency, for partially supporting this work. In addition, we thank SPG[7] members (especially Alberto Costa Neto) for feedback and fruitful discussions about this paper. We also thank Silvia Sampaio, Anne Ximenes, and Gisele Leal from Motorola Industrial for helping us on the Mobile Phone Test Cases.

References

Alves, V., Jr., P. M., Cole, L., Borba, P., and Ramalho, G. (2005). Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Prod-*

[7] `http://www.cin.ufpe.br/spg`

uct Line Conference (SPLC'05), volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer-Verlag.

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

Anastasopoulos, M. and Muthig, D. (2004). An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In *ICSR*, pages 141–156.

Baldwin, C. Y. and Clark, K. B. (2000). *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press.

Coplien, J. (2000). *Multi-Paradigm Design*. PhD thesis, Vrije Universiteit Brussel, Etterbeek, Belgium.

de Medeiros Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A. C., Borba, P., and Soares, S. (2007). Analyzing Class and Crosscutting Modularity with Design Structure Matrixes. In *Proceedings of the 21th Brazilian Symposium on Software Engineering (SBES'07)*. To appear.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A. (2005). Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, New York, NY, USA. ACM Press.

Hannemann, J. and Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 161–173, New York, NY, USA. ACM Press.

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

Kolb, R., Muthig, D., Patzke, T., and Yamauchi, K. (2005). 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. IEEE Computer Society.

Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., and Lucena, C. (2006). 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. IEEE Computer Society.

Neto, A. C., de Medeiros Ribeiro, M., Dósea, M., Bonifácio, R., Borba, P., and Soares, S. (2007). Semantic Dependencies and Modularity of Aspect-Oriented Software. In *Proceedings of 1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering (ICSE'07)*, page 11, Minneapolis, Minnesota, USA. IEEE Computer Society.

Patzke, T. and Muthig, D. (2002). Product Line Implementation Technologies. Technical Report 057.02/E, Fraunhofer Institut Experimentelles Software Engineering.

Tirila, A. (2002). Variability Enabling Techniques for Software Product Lines. Master's thesis, Tampere University of Technology, Tampere, Finland.