Extracting and Evolving Mobile Games Product Lines

Vander Alves, Pedro Matos Jr., Leonardo Cole,
Paulo Borba, and Geber Ramalho

Informatics Center, Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife PE, Brazil
{vra,poamj,lcn,phmb,glr}@cin.ufpe.br

Abstract. For some organizations, the proactive approach to product lines may be inadequate due to prohibitively high investment and risks. As an alternative, the extractive and the reactive approaches are incremental, offering moderate costs and risks, and therefore sometimes may be more appropriate. However, combining these two approaches demands a more detailed process at the implementation level. This paper presents a method for extracting a product line and evolving it, relying on a strategy that uses refactorings expressed in terms of simpler programming laws. The approach is evaluated with a case study in the domain of games for mobile devices, where variations are handled with aspect-oriented constructs.

1 Introduction

There are several approaches for developing software Product Lines (PL) [5]: proactive, reactive, and extractive [13]. Since the proactive approach supports the full scope of products needed on the foreseeable horizon, it demands a high upfront investment and offers more risks; therefore, it may be unsuitable for some organizations, particularly for small to medium-sized software development companies with projects under tight schedules. In contrast, the other two approaches have reduced scope and therefore require a lower investment; they are incremental and thus can be more suitable for such organizations. An interesting possibility is to combine the last two approaches. But, to our knowledge, this alternative has not been addressed systematically at the architectural and at the implementation levels.

In all approaches, variability management must be addressed in the domain: while focusing on exploiting the commonality within the products, adequate support must be available for customizing the PL core in order to derive a particular PL instance. The more diverse the domain, the harder it is to accomplish this task. This, in some cases, may outweigh the cost of developing the PL core itself.

This paper addresses the issues of structuring and evolving product lines in highly variant domains. In particular, we present a method that relies on the combination of the extractive and the reactive approaches, by initially extracting variation from an existing application and then reactively adapting the newly

created PL to encompass other variant products. The method systematically supports both the extractive and the reactive tasks by defining refactorings that are derived from simple Aspect-Oriented Programming (AOP) [12] laws. Further, we evaluate our approach in the context of an industrial-strength mobile game product line.

Indeed, there are a number of techniques for managing variability from requirements to code level. Most techniques rely on object-oriented concepts. These techniques, however, are well-known for failing to capture crosscutting concerns, which often appear in highly variant domains. Mobile games, in particular, must comply with strict portability requirements that are considerably crosscutting, thereby suggesting AOP to handle variation, which is explored in our method.

The next section provides the background needed for describing our approach. The section briefly explains variability issues in the mobile games domain and also introduces AOP. Section 3 describes our approach, including its strategy and both extractive and reactive refactorings. The industrial case study evaluating the approach is presented in Section 4. We discuss related work in Section 5 and offer concluding remarks in Section 6.

2 J2ME Games and Aspects

Mobile games (and mobile applications, in general) must adhere to strong portability requirements. This stems from business constraints: in order to target more users, owning different kinds of devices, service carriers typically demand that a single application be deployed in a dozen or more platforms. Each platform generally provides vendor-specific Application Programming Interfaces (APIs) with mandatory or optional advanced features, which the developer is likely to use in order to improve game quality. In addition, devices have memory and display constraints, which further requires the developer to optimize the application. In either case, adapting the game for each platform is mandatory.

In this work, we focus on game development for mobile phones using J2ME's MIDP profile, which is targeted at mobile devices with constrained resources [14]. We analyze and manage the specific kinds of variations arising from platform variation, where *platform* means a combination of MIDP, vendor-specific API, and hardware constraints. Accordingly, some of the specific challenges for managing variation in this domain are the following: UI features (such as screen size, number of colors, pixel depth, sound, keypad display); available memory and maximum application size; different profile versions (MIDP 1.0 and MIDP 2.0); different implementation of the same profile; proprietary APIs and optional packages; known device-specific *bugs*; different idioms.

These specific kinds of variation tend to be considerably fine-grained such that they generally crosscut the game core and are tangled with other kinds of variation. This suggests AOP as a suitable candidate for modularizing these variations.

Aspect-oriented languages support the modular definition of concerns that are generally spread throughout the system and tangled with core features. These

are called crosscutting concerns and their separation promotes the construction of a modular system, avoiding code tangling and scattering.

AspectJ [1] is the most widely used aspect-oriented extension to Java. Programming with AspectJ involves both aspects and classes to separate concerns. Concepts which are well defined with object-oriented constructs are implemented in classes. Crosscutting concerns are usually separated using units called aspects, which are integrated with classes through a process called weaving. Thus, an AspectJ application is composed of both classes and aspects. Therefore, each AspectJ aspect defines a functionality that affects different parts of the system.

Aspects may define pointcuts, advice and inter-type declarations. Pointcuts describe join points, which are sets of points of the program execution flow. Code to be executed at join points is declared as advice. Inter-type declarations are structures that allow the introduction of fields and methods into a class.

3 Method

Contrary to the proactive approach, which is more like the waterfall model, we rely here on a combination of the extractive and the reactive approaches. There are a number of reasons for this. First, small to medium-sized organizations, which still want to benefit from PLs, cannot afford the high cost incurred in adopting the proactive approach. Second, in domains such as mobile game development, the development cycle is so short that proactive planning cannot be completed. Third, there are risks associated in the proactive approach, because the scope may become invalid due to new requirements.

Our method first bootstraps the PL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the PL. Next, the PL scope is extended to encompass another product: the PL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a PL extension is used to add a new variant. The PL may react to further extension or refactoring.

The method is systematic because it relies on a collection of provided refactorings. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws [6]. These laws are appropriate because they are considerably simpler than most refactorings, involving only localized program changes, with each one focusing on a specific language construct.

3.1 Extraction

The first step of our method is to extract the PL: from one or more existing product variants, we extract a common core and corresponding product-specific

adaptation constructs. According to the variability nature of our domain, these constructs correspond to AspectJ constructs. The left-hand side of Figure 1 depicts this approach.

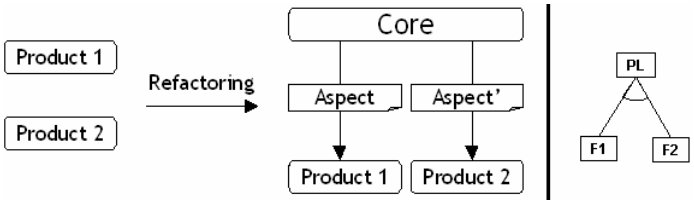


Fig. 1. Bootstrapping the Product Line

Product1 and *Product2* are existing applications in the same domain (for example, versions of a J2ME game for two platforms). *Core* represents the commonality within these applications. The core is composed with the *Aspect* and *Aspect'* aspects in order to instantiate the original products. These aspects thus encapsulate product-specific code.

The feature diagram [8, 4] for the PL is shown on the right-hand side of Figure 1. The diagram shows that the new PL is composed of two alternative subfeatures, *F1* and *F2*, representing *Product1* and *Product2*, respectively. The mapping between features and aspects is specified by a configuration knowledge mechanism [7], which imposes constraints on features and aspect combinations like dependencies, illegal combinations, and default combinations. Constraints involving only feature combinations are also specified in the feature model. The feature diagram is simple, since the PL has just been bootstrapped. However, as the PL evolves, either to accommodate more products or to explore further reuse opportunities, the diagram becomes more complex (Section 3.2).

In order to extract the variation within *Product1* and *Product2* — thus defining *Aspect* and *Aspect'*— we must first identify it in the existing code base. When more than one variant exists, diff-like tools provide an alternative. In either case, however, such a view is too detailed at this point. Indeed, the developer first needs to determine the general concerns involved. This could be described more concisely and abstractly with concern graphs, whose construction is supported by a specific tool [16]. Concern graphs localize an abstracted representation of the program elements contributing to the implementation of a concern, making the dependencies between the contributing elements explicit. Therefore, the actual first step in identifying these variations is to build a concern graph corresponding to known variability issues. In the case study described in Section 4, such issues would be the ones discussed in Section 2.

Once the concern graph is constructed, the developer should analyze the variability pattern within that concern. Depending on the pattern, a refactoring may be applied in order to extract it from the core. By analyzing applications

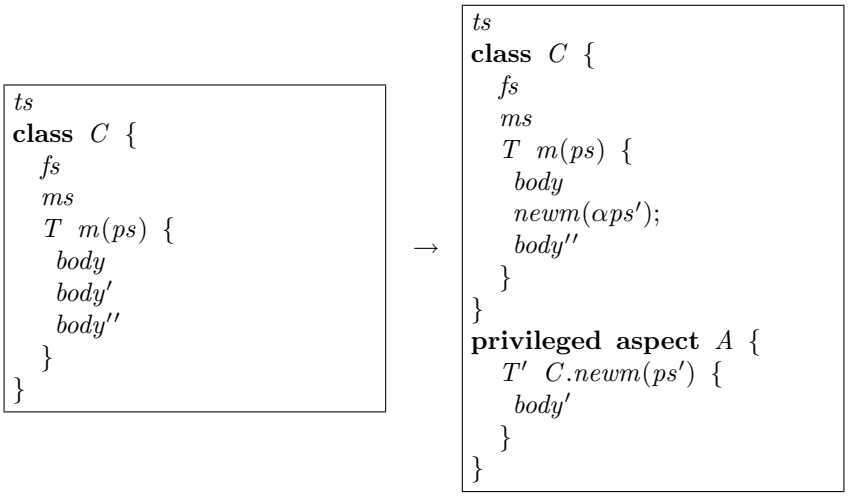
in the domain of mobile games, we observed a number of recurring variability patterns, for which the corresponding refactorings are listed in Table 1.

Table 1. Summary of Refactorings

Refactoring	Name
1	Extract Method to Aspect
2	Extract Resource to Aspect - after
3	Extract Context
4	Extract Before Block
5	Extract After Block
6	Extract Argument Function
7	Change Class Hierarchy
8	Extract Aspect Commonality

Some of the refactorings in Table 1, such as *Change Class Hierarchy*, are coarse-grained; others, such as *Extract Argument Function*, are fine-grained; some, such as *Extract Method to Aspect*, have medium granularity. Part of their names refers to an AspectJ construct that encapsulates the variation. For example, the *Extract Method to Aspect* refactoring is intended to extract the variant part of a concern, appearing in the middle of a method body, into AspectJ's inter-type declaration construct. Such declaration can then be implemented according to the specific variant. The refactoring structure is shown next:

Refactoring 1 ⟨Extract Method to Aspect⟩



provided

- A cannot be defined in ts;
- $body'$ does not change more than one local variable;

- A does not introduce any field to C with the same name of a C field used in $body'$.

On the left-hand side, $body'$ denotes the variability to be extracted. On the right-hand side, such variability is extracted into aspect A's inter-type declaration; thus a different aspect may provide a different variant implementation with that construct. We denote the set of type declarations (classes and aspects) by *ts*. Also, *fs* and *ms* denote field declarations and method declarations, respectively. Finally, we use α preceding a list of parameters to denote only the names of those parameters.

The refactoring provides preconditions to ensure that the program is valid after the transformation. Another use of the preconditions is to guarantee that the transformation preserves behavior. Refactoring 1 has preconditions arising from simpler transformations and refactorings, whose composition yields the whole refactoring.

The first precondition guarantees validity: since the refactoring creates an aspect A, such aspect cannot be defined in *ts*. For the second precondition, as we rely on the *Extract Method* refactoring [9], we need a precondition stating that the piece of code extracted into its own method does not change more than one local variable. Otherwise, the extracted code would need to return two values, and that would not be possible. Regarding the third precondition, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class, according to the AspectJ semantics. Hence, it is possible to declare a private field as a class member and as an inter-type declaration at the same time using the same name. As a consequence, transforming a member method that uses this field into an inter-type declaration implies that the method now uses the aspect inter-typed field. This leads to a change in behavior. A precondition is thus necessary to avoid this problem.

As mentioned, the application of the refactoring creates a new aspect (A), which is related to a variant concern. Further application of other refactorings may refine A, incorporating additional elements of this concern, possibly using other constructs such as pointcuts and advice. In fact, except for Refactorings 1 and 7, all the others in Table 1 deal with pointcuts and advice constructs. A slight variation of Refactoring 1 would consider the pre-existence of aspect A in order to make the refactoring available for repeated applications. Additionally, even though aspect A is privileged, this constraint can be removed later, after moving, with intermediate refactorings, other pieces of the variant concern into the aspect, for example by using the *Extract Resource to Aspect* refactoring.

Indeed, after applying Refactoring 1, there may remain other variabilities to be extracted from the core. The strategy is to apply the refactorings in Table 1 repeatedly, such that the product line core and the variant aspects are built progressively. Section 4 illustrates this with a case study.

3.2 Evolution

Once the product line has been bootstrapped, it can evolve to encompass additional products. In this process, a new aspect is created to adapt the core to the

new variant. Moreover, a new feature is added to the feature diagram in order to represent the new product, and the configuration knowledge is updated to map the new feature to the new aspect (Figure 2).

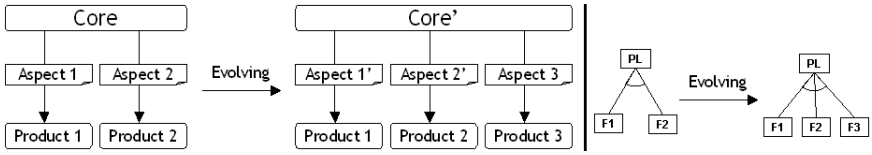


Fig. 2. Evolving the Product Line

The refactorings in Table 1 can also be used for evolution. As Figure 2 also indicates, the core itself may evolve because features common to *Product1* and *Product2* might not be shared by *Product3*. This may trigger further adaptation of the previously existing aspects, too. However, AspectJ tools can identify parts of the core on which these previous aspects depend, and some refactorings are also aspect-aware [10], thereby minimizing the need to revisit such previous aspects.

Another evolution scenario involves restructuring the product line to explore commonality within aspects. Such commonality would not be in the core when it is not shared by all products, but only by a subset. The feature diagram is also changed to show the commonality extraction (Figure 3). The existing commonality is extracted from *F1* and *F2* and is represented as a new optional feature, *F12*. Further, the feature model is augmented with the constraint that *F1* and *F2* depend on *F12*, and the configuration knowledge with the mapping of *F12* to *Aspect12*. An alternative approach would not update the feature model, but then the configuration knowledge would have to map *F1* to $\{Aspect1'', Aspect12\}$ and *F2* to $\{Aspect2'', Aspect12\}$. The former alternative should be used when it is meaningful to have the *F12* feature; the latter when the extracted commonality is meaningful only at the code level.

Figure 3 can become more complex with the addition of new platforms and identification of reusable aspects. However, constraints in the feature model as well as the configuration knowledge (the mapping of features to aspects) limit aspect combinations, thereby providing support for scalability.

4 Method Evaluation

We performed a case study to evaluate variability in J2ME games, which are mainstream mobile applications of considerable complexity in comparison with other mobile applications. In particular, we investigated how the same game GM was adapted to run in three platforms (P_1, P_2, and P_3)[1]. P_1 relies solely on

[1] The actual names are not relevant here.

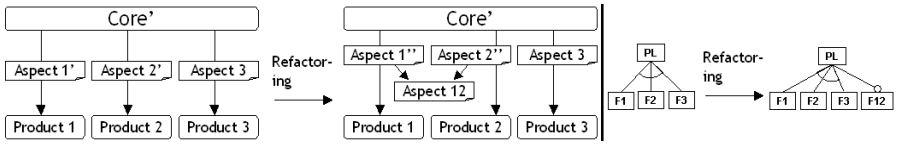


Fig. 3. Refactoring the Product Line

MIDP 1.0, whereas P_2 and P_3 rely on MIDP 1.0 and a proprietary API. GM is a game currently offered by service carriers in South America and Asia.

The variability issues within these products are as follows: optional images, proprietary API, application size limit, screen dimensions, and additional keys. One important remark is that these features are not independent. Indeed, application size constrains other features, such as optional images and additional keys.

In order to evaluate our approach, we created a PL implementation of the three products and then compared the PL version with the original implementation of these products. To create and evolve the PL, we first identified the variabilities (such as optional images) with concern graphs and then moved their definition to aspects using the *Extract Resource to Aspect* refactoring. In another step, we addressed method body variability within the platforms. Accordingly, we made extensive use of the *Extract Method to Aspect* refactoring. The *Extract After Block* and *Extract Before Block* refactorings were used when the variant code appeared at the end or beginning of the method body. On the other hand, the *Extract Context* refactoring was used when the variation surrounded common code, representing a context to it. The *Extract Argument Function* refactoring was used when variation appeared as an argument for a method call. Finally, we used the *Change Class Hierarchy* refactoring to deal with class hierarchy variability.

As mentioned in Section 3.1, in order to better identify and understand some variations, we can use concern graphs, which are created iteratively by querying a model of the program, and by determining which elements (class, methods, and fields) and relationships returned as part of the queries contribute to the implementation of the concern. The querying process starts with a *seed* [16], usually a class found with a lexical tool. From this class, the remaining elements are added with tool support. For example, the concern graph C for the optional images concern (oi) in P_1 would be as follows:

$$C_{p1,oi} = (V_{p1,oi}, V^*_{p1,oi}, E_{p1,oi}),\ V^*_{p1,oi} = \varnothing$$

$$V_{p1,oi} = \left\{ \begin{array}{l} \mathit{Resources}, \mathit{GameScreen}, \mathit{Resources.dragonRight}, \\ \mathit{Resources.loadImages}(), \mathit{GameScreen.wakeEnemy}() \end{array} \right\},$$

$$E_{p1,oi} = \left\{ \begin{array}{l} (reads, GameScreen.wakeEnemy(), Resources.dragonRight), \\ (writes, Resources.loadImages(), Resources.dragonRight), \\ (declares, Resources, Resources.dragonRight), \\ (declares, Resources, loadImages()), \\ (declares, GameScreen, wakeEnemy()) \end{array} \right\},$$

The set $V_{p1,oi}$ describes the vertices (classes, methods, attributes) partially implementing the concern. Set $V^*_{p1,oi}$ consists of vertices (classes, methods) solely dedicated to the concern implementation. Finally, set $E_{p1,oi}$ groups edges relating elements from the previous sets.

During the evolution of the PL to include P_3, we had to deal with the *load images on demand* concern. This concern was specific to this platform, as it had constrained memory and processing power. To implement this concern, we had to define a method for each screen that could be loaded. Before a screen was loaded, the corresponding method was called. In contrast, in P_1 and P_2 implementations, the images were loaded only once, during game start-up. In this case, there was only one method that loaded all the images into memory. This situation illustrates the scenario in Figure 2.

We addressed this by applying a sequence of *Extract Method* refactorings in the core to break the single method loading all images into finer-grained methods loading images for each screen; the call of this single method was then moved from the core to P_1's and P_2's aspects, and the calls to such smaller methods were moved to P_3's aspect by the *Extract Before Block* refactoring.

Another evolution scenario took place when we realized that some commonality existed between P_1 and P_2 with respect to the *flip* feature[2]: these two platforms are from the same vendor and share this feature, which is not shared by P_3, from another vendor. Therefore, the flip feature is isolated in the corresponding aspects of P_1 and P_2, but it would be useful to extract this commonality into a single module. In fact, we were able to factor this out into a single generic aspect with the *Extract Aspect Commonality* refactoring, thus illustrating the scenario in Figure 3.

After creation and evolution of the PL, we analyzed code metrics. Table 2 shows the number of Lines of Code (LOC) for each product in the original implementation, in contrast with the PL implementation. We calculate the LOC of a PL instance as the sum of the core's LOC and the LOC of all aspects necessary to instantiate this specific product.

Table 2. LOC in original and PL implementations

Original Implementation				**PL Implementation**				
P_1	P_2	P_3	Total	Core	P_1	P_2	P_3	Total
2965	2968	3143	9076	2549	3042	3047	3210	4405

[2] Proprietary graphic API allowing an image object to be drawn in the reverse direction, without the need for an additional image.

Table 2 shows that LOC is slightly higher when comparing each PL instance with the corresponding product in the original implementation. This is caused by the extraction of methods and aspects, which increase code size due to new declarations. On the other hand, there is a 48% reduction in the total LOC of the PL implementation, when compared to the sum of LOCs of the single original versions. This was possible because the core, which represents 57% of the PL LOC, is reused in all instances, thus eliminating most of code repetition occurring when there are three independent implementations. Another factor that contributes to the reduction in PL LOC is the existence of reusable aspects.

Another analyzed metric was the packaged application (jar files) sizes of the original and of PL implementations (Table 3). The jar files include not only the bytecode files, but also every resource necessary to execute the application, such as images and sound files. The jar file size is a very important factor in games for mobile devices, due to memory constraints.

Table 3. Jar size (kbytes) in original and PL implementations

	Original Implementation		**PL Implementation**	
	size	reduced size	size	reduced size
P_1	61,9	58,5	97,0	67,9
P_2	61,7	57,3	97,6	61,8
P_3	56,1	52,4	93,5	56,7
Total	179,8	168,2	288,2	186,3

We can notice a jar size increase from original versions to PL instances. The reason for this is the overhead generated by the AspectJ weaver on the bytecode files. We also noticed that very general pointcuts intercepting many join points can lead to greater increases in bytecode file sizes. This considerably influenced us in the definition and use of the refactorings. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [2]. The reduced size of each original version and PL instance are shown in Table 3.

5 Related Work

Prior research also evaluated the use of AOP for building J2ME product lines [3]. We complement this work by considering the implementation of more features in an industrial-strength application, explicitly specifying the refactorings to build and evolve the PL, and raising issues in AspectJ that need to be addressed in order to foster widespread application in this domain.

AOP refactorings have also been described elsewhere [15, 11]. The former proposes a catalog for object-to-aspect and aspect-to-aspect refactorings, whereas the latter provides an abstract representation of object-to-aspect refactorings as

roles. However, their use in the PL setting is not explored, and the refactorings format follows the imperative style [9]; in contrast, our approach is template-oriented, abstract, concise, and thus does not bind a specific implementation, which could be done, for instance, with a transformation systems receiving as input refactoring templates.

Concern graphs provide a more concise and abstract description of concerns than source code [16]. We rely on concern graphs to identify variant features. Once the concern is identified, we extract it into an aspect and may further revisit it during PL evolution.

In previous work, a language-independent way to represent variability is provided, and it is shown how it can be used to port J2SE applications to a J2ME product line [17]. Our approach differs from such work because, although ours relies on language-specific constructs, it has the advantage of not having to specify join points in the base code.

6 Conclusions

We present a method for creating and evolving product lines combining the reactive and extractive approaches. Our method uses a set of refactorings, which can be extended when necessary. These refactorings can be derived from a combination of programming laws that allow us to better understand these refactorings and increase the confidence that they are correct. Our refactorings rely on AOP to modularize crosscutting concerns and to generalize the implementations of these concerns in order to increase code reuse. Constraints in the feature model and in the configuration knowledge limit aspect combination and thus promote scalability of the process.

Our evaluation with an existing mobile game shows that we can benefit from extensive code reuse and easily evolve the PL to encompass other products while still maintaining code reliability. It also shows that the sequence of applied refactorings must be strategically chosen. This strategy can be influenced by some factors like desirable reuse level and application size restrictions. Although the evaluation is in the mobile game domain, we argue that the method and the issues addressed here are valid for mobile applications in general, of which mobile games are representative. We also believe that other highly variant domains could benefit from our method.

Acknowledgements

We thank Meantime Mobile Creations for granting us access to the game used in our case study and for fruitful discussions with lead developers Alexandre Damasceno and Pedro Sampaio. We also appreciate valuable feedback from the members of SPG, André Santos, Jeff Gray, the anonymous referees, and our shepherd Pierre America. This work was partially supported by CNPq and FACEPE.

References

1. *AspectJ project*. http://www.eclipse.org/aspectj/, 2005.
2. *ProGuard*. http://proguard.sourceforge.net, 2005.
3. M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 2004.
4. T. Bednasch, K. Czarnecki, U. Eisenecker, and M. Lang. *Captain Feature*. https://sourceforge.net/projects/captainfeature/, 2005.
5. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
6. L. Cole and P. Borba. Deriving refactorings for AspectJ. In *AOSD'05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 123–134, 2005.
7. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
8. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *SPLC*, pages 266–283, 2004.
9. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.
10. Oberschulte C. Hanenberg S. and Unland R. Refactoring of aspect-oriented software. In *Net.ObjectDays*, Erfurt, Germany, September 2003.
11. J. Hannemann, G. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, 2005.
12. 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.
13. C. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, pages 282–293, 2001.
14. Sun Microsystems. *JSR-37 Mobile Information Device Profile (MIDP)*. http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html, 2000.
15. M. Monteiro and J. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, 2005.
16. M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.
17. W. Zhang, S. Jarzabek, N. Loughran, and A. Rashid. Reengineering a PC-based system into the mobile device product line. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, 2003.