AOP-Driven Variability in Software Product Lines[1]

Vander Alves
Informatics Center
Federal University of Pernambuco – Brazil
vra@cin.ufpe.br

Abstract

Software Product Line has emerged as a promising process framework for developing a set of products scoped within a market segment and based on common artifacts. The challenge is to achieve higher productivity and quality standards. One of its core activities is variability representation and implementation, whereby specific product features are modeled, implemented, and then incorporated into the core product line architecture to instantiate the particular product. This paper proposes an approach to variation modeling and implementation, according to which we employ feature models and the Aspect-Oriented Paradigm to achieve an abstract and a modular view of variation, thereby promoting reuse and gains in productivity. We perform a case study to evaluate the approach in the domain of pervasive computing applications.

1. Introduction

Computational systems are becoming ubiquitous. By using a mobile phone with computational power, we can access and manipulate information almost everywhere and anywhere. Similarly, other electronic devices will gain or augment computational power. Indeed, the impact of information technologies in the society will increase significantly. Therefore, in this scenario, applications have to comply with high ever-increasing quality standards, specially availability and usability.

In order to meet these high quality standards, current applications must comply with a series of functional and non-functional requirements such as persistence, concurrency, distribution, and adaptability. This increases further the already complex task of developing these systems. Additionally, the development processes must be productive, and the resulting software must be extensible and reusable [4].

In order to meet the challenge of developing current applications, paradigms such as object orientation and software processes are used. The Object-Oriented Paradigm (OOP) is implemented by languages and relies on design and architectural patterns [5][7]. As a result, it offers more effective means to achieve reuse, thereby increasing productivity of future projects, and software maintenance. Object orientation, however, has some shortcomings, such as difficulty in modularizing systemic requirements, such as non-functional requirements and complex object protocols [17][18]. In order to overcome

[1] Joint work with Ayla Dantas and Paulo Borba. A shorter version was accepted as a poster at the *Second International Generative Programming and Component Engineering Conference (GPCE'03)*, September 22-25, 2003. Erfurt, Germany. Presented at the Workshop of Flexible and Adaptive Systems, a satellite event of SugarLoafPLoP 2003, August 12, Recife, Brazil.

these shortcomings, novel extensions of object orientation have been proposed, among which Aspect-Oriented Paradigm (AOP) is receiving increasing attention [13].

Software development processes also guide application development. These processes define activities to be carried out, the resulting artifacts, and the people to perform them. Processes thus help to reduce development complexity, promoting its predictability and reproducibility. Some shortcomings of existing processes are lack of implementation support [10]. As a result, reuse and extensibility, achieved during design, may be lost during the implementation, resulting in quality decrease of the final software. Some extensions of processes focusing on implementation are already being defined [1][14][19].

More recently, software processes are being generalized in meta processes called Software Product Lines (PL) [16], which focus on the development of a family of products targeting a specific market and based on a common base of artifacts. In a PL, there is a generic architecture which is common to all products in the line; this architecture is adapted for the creation of a particular product. In this process, variability modeling and implementation play a central role: the specific products differ in terms of these variations, and thus modeling and implementing them appropriately will translate into higher PL productivity.

Current approaches to variability modeling and implementation rely on employing purely object-oriented techniques [8][15]. As this paper will explain, higher levels of reuse in a PL may be achieved if complementary techniques are employed. In particular, our approach explores the use of feature models [12] and AOP in order to model and implement variations in a PL, respectively. In so doing, we expect a boost in productivity and in the quality of the member products. We perform an initial case study in the domain of pervasive computing applications in order to evaluate the approach.

The remainder of this paper is organized as follows. Section 2 gives an overview of possible approaches to variation modeling. Next, Section 3 provides a brief introduction to AOP and then details our approach and how it depends on this paradigm. Section 4 then describes the approach in the PL context. Finally, Section 5 summarizes contributions of this work, presents related work, and points to future research.

2. Variation Modeling

In a Product Line (PL) context, variation modeling plays a central role: the PL members are described in terms of specializations, configurations, and adaptations of the PL architecture.

In this section, we explore modeling notations for expressing variations in a product line, namely feature models, use case models, and class diagrams. The focus is on feature models, since our approach relies mostly on them and such models are not as in widespread industrial use as the other two models. Additionally, we will describe how

current processes typically employ these models. In Section 3, our approach will be presented.

2.1 Feature models

A feature is a distinguishable characteristic of a concept (system, component, and so on) that it relevant to some stakeholder of the concept. For example, it may be either a functional or non-functional requirement of the system.

Feature models represent the common and the variable features of concept instances and the dependencies between the variable features. A feature model consists of a feature diagram and some additional information, such as short semantic descriptions of each feature, rationales for each feature, stakeholders and client programs interested in each feature, priorities, and dependency rules.

From a feature diagram of a concept, we can derive featural descriptions of the individual instances of a concept. In a PL context, a feature diagram represents the PL itself, whereas a featural description represents one member in the PL. Hereafter, we interpret features in this context.

A feature can be of one of the following types:

- Mandatory: this feature is present in all members of the PL;
- Optional: this feature may be present in a member of the PL;
- Alternative: exactly one of a set of features is present in a PL member;
- Or-feature: at least one of a set of features is present in a PL member.

A feature diagram is a tree, where each node is a *feature* of some type. The edges of the tree are interpreted together with the node type, and both indicate the configurability of the node, that is, whether the feature represented in the node can, must, or cannot be asserted of a particular PL member. In the diagram, a mandatory feature node is pointed to by a simple edge ending with a filled circle; an optional feature node is pointed to by a simple edge ending with an empty circle; the nodes of alternative features are pointed to by edges connected by an arc; the nodes of or-features are pointed to by edges connected by a filled arc.

Figure 1 depicts a feature diagram for a PL in the domain of Dictionary applications for embedded devices. *Dictionary* is the *root* feature of the PL. Features *Translation*, *Screens,* and *Search mechanism* are mandatory; feature *Dynamic customization* is optional; features *Dynamic screens, Colorized screens,* and *Internationalized screens* are or-features; *Server* and *Memory* are alternative features.

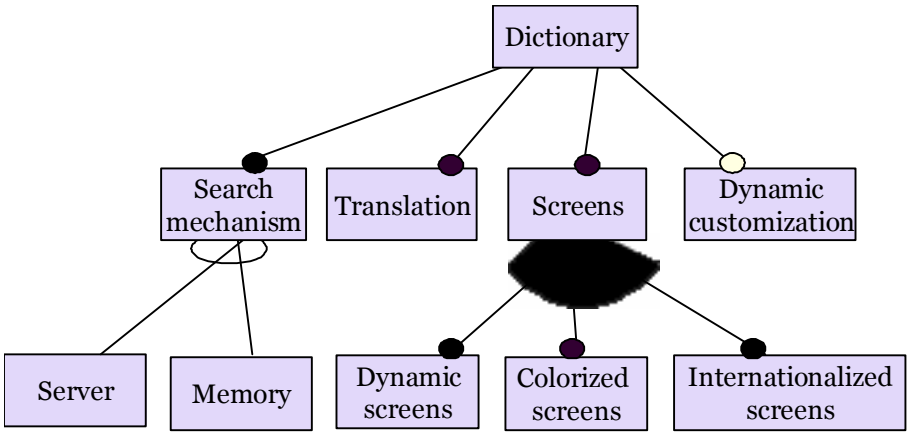

Figure 1. Feature diagram for Dictionary domain for embedded devices

Variability in feature diagrams is expressed using optional, alternative, and or-features. These features are referred to as *variable features*. The nodes to which variable features are attached are referred to as *variation points*. In Figure 1, all features except *Translation, Screens*, and *Search mechanism* are variable features; *Dictionary*, the *Screens* and the *Search mechanism* features are variation points, which are clearly pinpointed in the diagram.

We emphasize that a feature diagram models the configurability aspect of the PL, thereby leaving other aspects such as structural and behavioral relationships to other models. The very advantage of feature diagrams is that they avoid cluttering the configurability aspect with other aspects.

Some processes such as FeatuRSEB [9] are feature model centric, with this model serving as a concise synthesis of the variability and commonality represented in other models, specially the use case model.

2.2 Use Case models

The use case model defines what a system should do for its users. This model is used during requirements capture by the customers, end users, software engineers, and other stakeholders as they decide what the system should do.

In the use case model, users of the system are called *actors*. Each actor defines a distinct role assumed by a person or a machine interacting with the system. Each way an actor uses the system is a distinct use case. Each *use case* defines a set of interactions with the system. The use case model consists of actors and use cases, together with descriptions of their interactions and connections.

Variability, as proposed by Jacobson et. Al [11], is represented by the *extends* stereotype. Figure 2 shows a simple use case diagram in the same domain as the one used in Figure 1. There is one actor and five use cases, of which *ScreenSelection* is a variation point, and all others except *Translate* are variants. The variation point is indicated with a dot in the middle of the use case symbol.

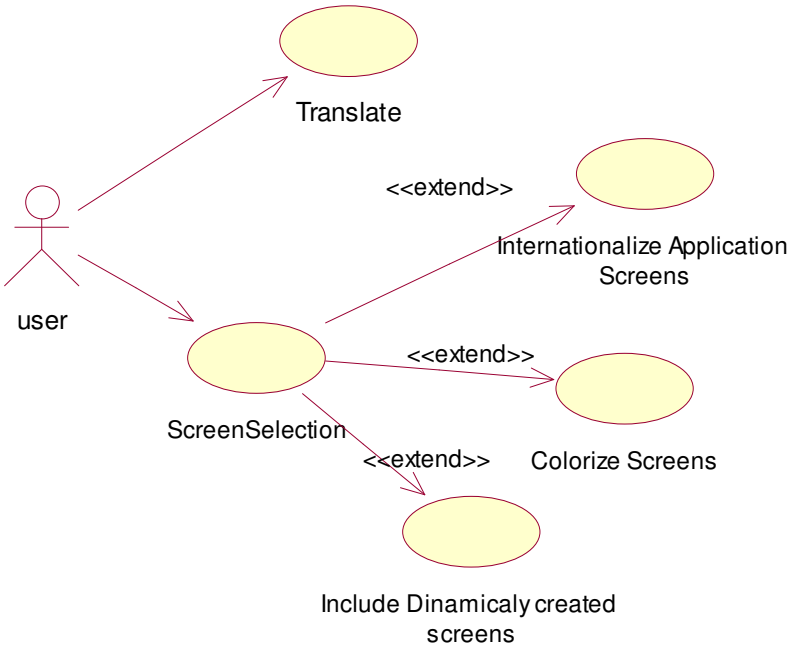

Figure 2. Use case diagram with one variation point and three variants.

Similarly to feature models, use case models can also describe a PL in an implementation-independent way. However, a use case model captures the system requirements from the user perspective (i.e., "operational requirements"), including only functional requirements, whereas the feature model organizes both functional and non-functional requirements from the domain engineer perspective, who is concerned with performing commonality and variability analysis.

In terms of expressive power of the notations employed by these two models, feature models allow representing the *configurability* of functionalities and non-functional requirements: whether a given feature is mandatory or optional, and whether a given set of features are alternative or or-features. On the other hand, use case models represent functionalities (use cases) and relationships among them, with generalization, include, and extend relationships. Although both models allow the representation of functional variability, the configurability nature of the former provides a concise and more abstract description of variation than the latter. Indeed, as Griss et. al. reported [9] the understandability of use case diagrams for domain engineers does not scale well for large systems and this model has a limitation in describing technical functionalities in some domains such as telecom.

Most use case-centered processes such as the Rational Unified Process (RUP) [10] are application engineering processes, instead of a full PL process, which also encompasses domain engineering processes, where feature models play a central role. As Section 3 will present in detail, our approach emphasizes the complementary use of these models.

2.3 Class diagrams

The UML notation has been extended with a stereotype, <<V>>, to denote variability [15]. In class diagrams, variability is expressed either as inheritance or aggregation. Figure 3 shows a small part of a detailed class diagram in the PL framework for the domain of applications introduced in Figure 1. Our goal here is only describing variability representation in class diagrams rather than the mapping between this model and the one from Figure 1. In fact, as Section 3 will present in detail, we do not employ class diagrams in representing variability. Therefore, the example in Figure 3 actually illustrates a possible design in which our approach is not employed.

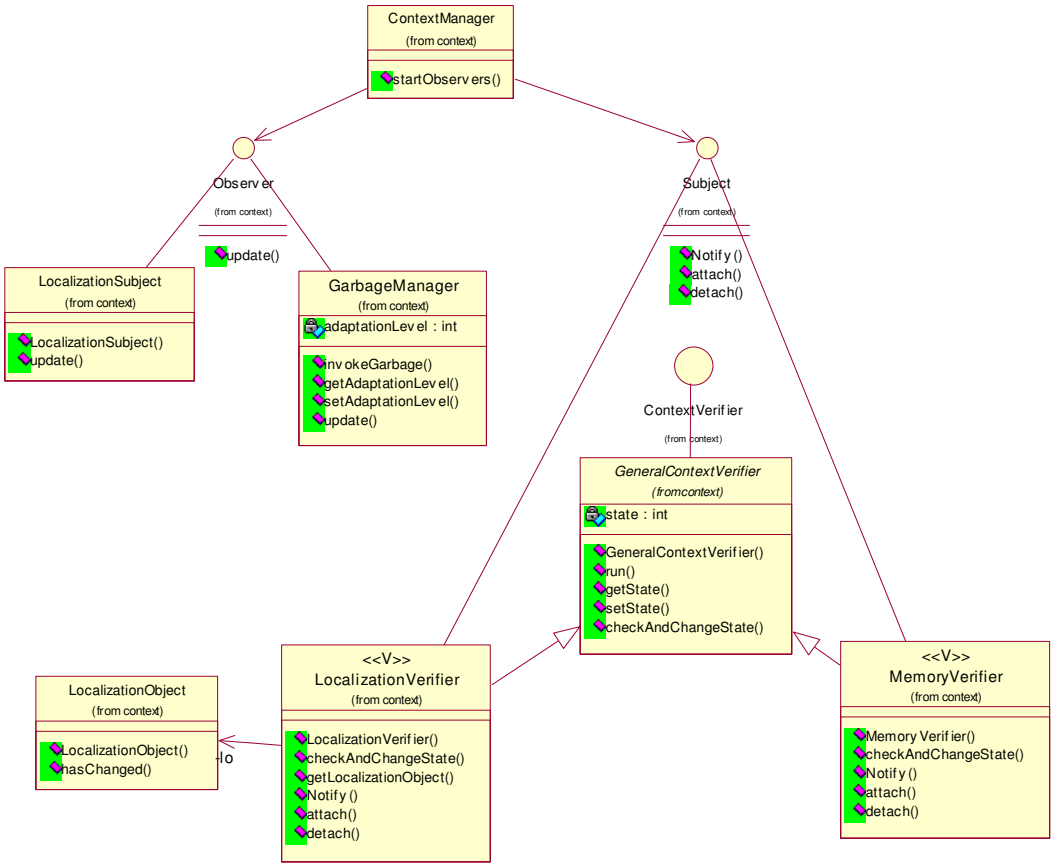

Figure 3. Variability in class diagrams.

GeneralContextVerifier is a *variation point*, and *LocalizationVerifier* and *MemoryVerifier* are *variants*. This is a simple diagram, just aiming to illustrate the

variation notation, but in industrial-strength applications, the diagram becomes a cluttering of concerns, involving structural and configuration information.

The previous subsection explained that, for the purpose of representing variation of functionalities, feature models are more abstract than use case models. We now contrast feature models with class diagrams in the context of variation representation.

First, we consider the following variation description: a variation point in a PL has three variable features, and at least one of these features is present in each PL member. This is an or-feature, which can be described concisely by a feature diagram, as illustrated in Figure 4. In particular, we note that the representation makes no attempt to describe the particular combinations of features for a particular product.

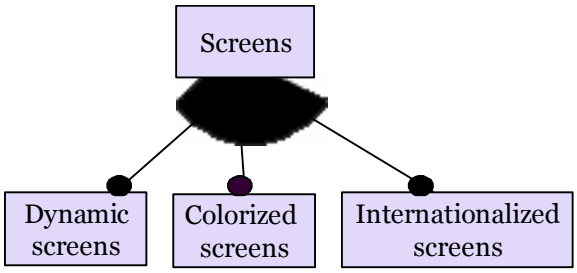

Figure 4. Variation point in a feature diagram. Variable features are or-features. A PL member presents at least one of the sub-features.

Next, we consider a purely object-oriented representation of this variability in class diagrams. According to the language for variability representation in such diagrams, the variation is restated as follows: a variation point in a PL has three variants, and at least one of these variants is present in each PL member. Figure 5 shows this purely OO representation in a class diagram. We note that inheritance is used extensively in order to explore the possible ways in which the variants compose. This leads to the problem of behavior and state fragmentation. For instance, a class such as *Colorized Screens* is defined in 3 places (*Colorized, ColorizedDynamic,* and *InternationalizedColorized*); the same is true for *Dynamic Screens* and *Internationalized Screens*.

The fragmentation makes it difficult to identify the reusable elements. It could be alleviated with multiple inheritance, but the diagram would become more complex and thus harder to reuse. Therefore, the object-oriented representation of variation in class diagrams is not as abstract and as concise as that of feature models; additionally, at the design level, another mechanism should be employed in order to avoid the fragmentation problem and promote higher levels of reuse. In the next section, we show one such alternative, namely aspects.

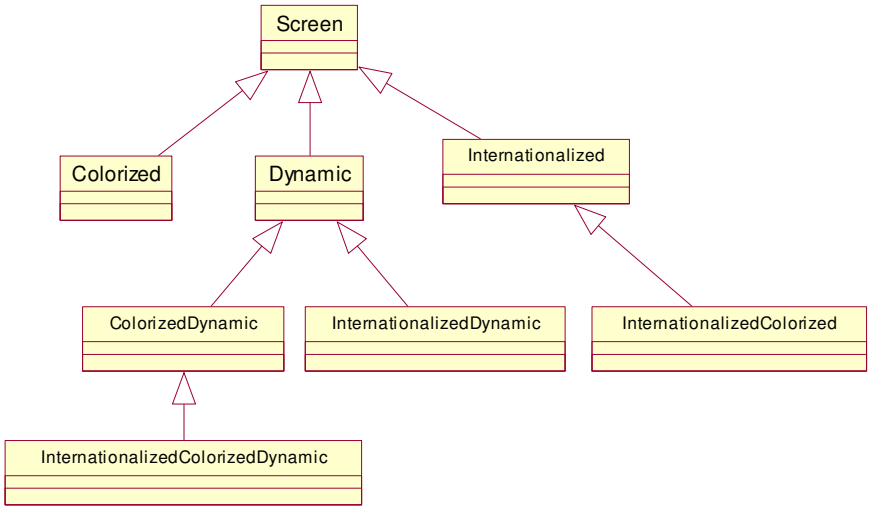

Figure 5. Variation point implementation

3. AOP Driven Variability

In most software development processes, variability modeling is performed by employing some combination of use case and class diagrams. These diagrams have been used successfully in the development of a single product. Nevertheless, as mentioned in the previous section, these modeling techniques have some shortcomings in a PL context, where a higher level of reuse is desired. In this section, we present a complementary method for handling variability in PLs at a more abstract level and thus promote higher reuse levels. Our approach relies on employing the Aspect-Oriented Paradigm (AOP). We start by describing concisely this paradigm and then we detail our approach. Finally, we provide some initial evaluation of the approach.

3.1 Aspect-Oriented Paradigm

Objects fail to modularize certain concerns. For example, object-oriented design and implementation of non-functional requirements such as persistence, distribution, synchronization, and logging may lead to tangled code, where these concerns are spread throughout the code [17]. This problem may also arise with some functional requirements, in particular those involving elaborate protocols among objects [18]. Although design and architectural patterns mitigate this problem, it seems that it cannot be completely eliminated, which thus suggests an inherent limitation of objects.

The Aspect-Oriented Paradigm (AOP) [13] addresses this problem. An aspect is a module representing a crosscutting concern, encapsulating its representation in a single unit, rather than allowing its scattered representation throughout design elements and source code. The locations affected by the aspect are referred to as *join points*, which are

described concisely and abstractly by *pointcuts*, at which specific behavior and structure can be inserted by *advice* and *introduction* statements, respectively.

Current practice in AOP has shown that it has successfully led to modular designs and implementations of crosscutting concerns [20]. This paradigm is intended to be used as a complement to the object-oriented approach, rather than instead of it. Indeed, AOP languages like AspectJ are extension of OO languages such as Java. Additionally, design and architectural patterns may also be used to structure aspects, which can also be composed into frameworks [6]. The approach described in this paper combines these two paradigms to achieve better quality and productivity goals in PL development.

3.2 Strategy

Use case models may not scale with clarity for larger systems and they lack feature expressiveness for some domains. On the other hand, waiting to model variability directly in class diagrams by employing solely object-oriented mechanisms may lead to poor reuse levels like the one depicted in Figure 5.

We propose a complementary approach, where use case and class diagrams are used together with the more abstract feature models, with which we are able to consider different options of variation point implementation, including others than object-oriented techniques. In fact, we postulate that variability is inherently an aspect, and thus the core of our strategy is modeling and implementing variation points as aspects. Use case and class diagrams are still used, but in modeling the PL common functionalities and architecture. Figure 6 shows an overview of our process proposal.

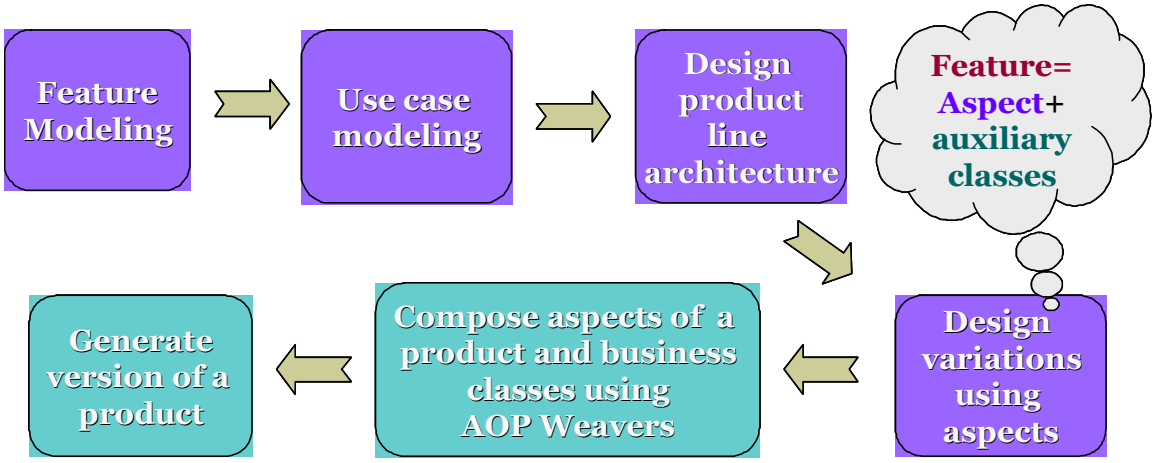

Figure 6. Process proposal overview

First, Feature Modeling defines the features of the product line members and identifies common features among all products as well as the variable ones for the members. This essentially corresponds to scoping the PL, and the most important artifact generated is the

PL feature model. An example of this model and the artifacts created in the next steps are represented as views in Figure 7.

Second, PL functionalities are modeled in more detail using use case diagrams. The use case view is not shown in Figure 7 due to space limitations. Third, the product line architecture is designed according to use case realizations of mandatory features. This activity is mostly an object-oriented modeling step, making use of architectural and design patterns to achieve a flexible architecture, which is reused throughout all PL members. An important property of this architecture is the absence of variation points.

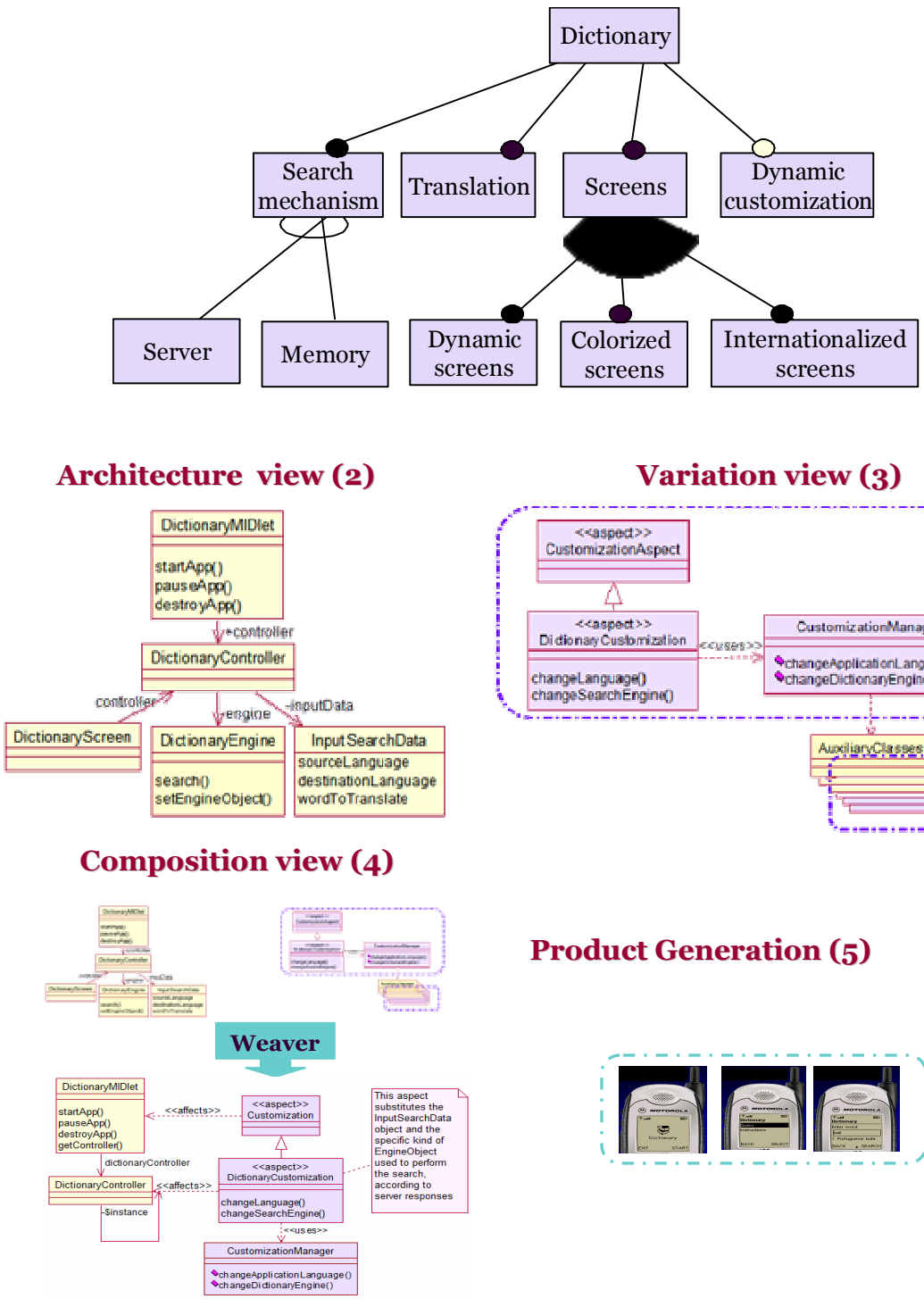

Figure 7. Artifact view of proposed process. Only a piece of each model is shown. The use case view is not shown in this figure due to space limitations

Fourth, variations previously identified during Feature Modeling are modeled as aspects. In particular, an aspect stereotype is created to represent a variable feature as an aspect in a UML class diagram. The aspect is shown together with any auxiliary classes in the variation view, which ultimately crosscuts the architecture view, but is maintained independently from it. This independence is possible due to the modular representation of crosscutting concerns provided by aspects. Use of design patterns [7] is also recommended in the variation view.

Fifth, we collect the aspects corresponding to a desired configuration of variable features, the auxiliary classes they use, the product line architecture, and compose them automatically using AOP weavers [13]. This is guided by a production plan, which specifies how to compose these components. In particular, abstract pointcuts are implemented in concrete aspects in order to augment the PL architecture with the structure and behavior necessary to implement the variable features selected for a particular product. Finally, code is ready to be processed, the application is packaged, deployed, and tested.

3.3 Evaluation

The proposed process described previously has been used in an initial case study in the domain of pervasive computing device applications. In particular, we have addressed the domain of dictionary-like applications for mobile information devices, such as mobile phones. All figures presented previously are from this domain.

The execution of the case study consisted of the following activities: 1) implementing functional variation using aspects; 2) implementing functional variation using design patterns; 3) implementing functional variation with OO techniques but without design patterns. Developers implemented both functional and non-functional variations represented by Figure 1. In all activities, code size was measured, and developer's opinions about the method were recorded. No development time was tracked during this initial study.

Code size for activity 1 tended to be 20% smaller than for activities 2 and 3. This and developer's remarks about the process suggest that variability representation in the feature model and their representation and implementation with aspects provide a more abstract and understandable view of the domain even when the model grows in size. Additionally, according to developer's opinion, the mapping of variable features to aspects promotes a clear separation of the reusable core PL architecture from the configurable aspects, thereby increasing modularity and promoting a more automatic configuration of aspects specific to a certain PL member. These characteristics suggest an improvement in application development over a conventional process.

Indeed, this is an initial evaluation only, according to the scope of this work. Further steps include using more precise and not subjective metrics, such as development time and reuse-related metrics, scaling the development team to a larger group, deriving more

products from the same PL, and experimenting with other domains. We plan to address games and multimedia applications next.

4. Process documentation

The process described in the previous section is in fact a refinement of the Framework for Software Product Line Practice [16]. In this section, we describe our process proposal within the PL context.

The first three activities in our process proposal, namely *Feature Modeling*, *Design Product Line Architecture*, and *Design Variations Using Aspects* (Figure 6), focus on the development of reusable assets. Accordingly, feature models define the PL scope, each particular product being a feature description of this model; the PL architecture is the core skeleton reused among specific products; the variations are modeled independently from the core architecture as aspects and can be considered reusable components which are composed according to a production plan in order to implement a specific product configuration. Therefore, in the PL Framework, these activities refine the general *Core Asset Development* activity.

The remaining activities in our proposed process, namely *Compose Aspects and Business Classes Using AOP Weavers* and *Generate Version of a Product*, focus on the instantiation of a specific PL member, thereby refining the *Product Development* activity of the framework. These relations are illustrated in Figure 8.

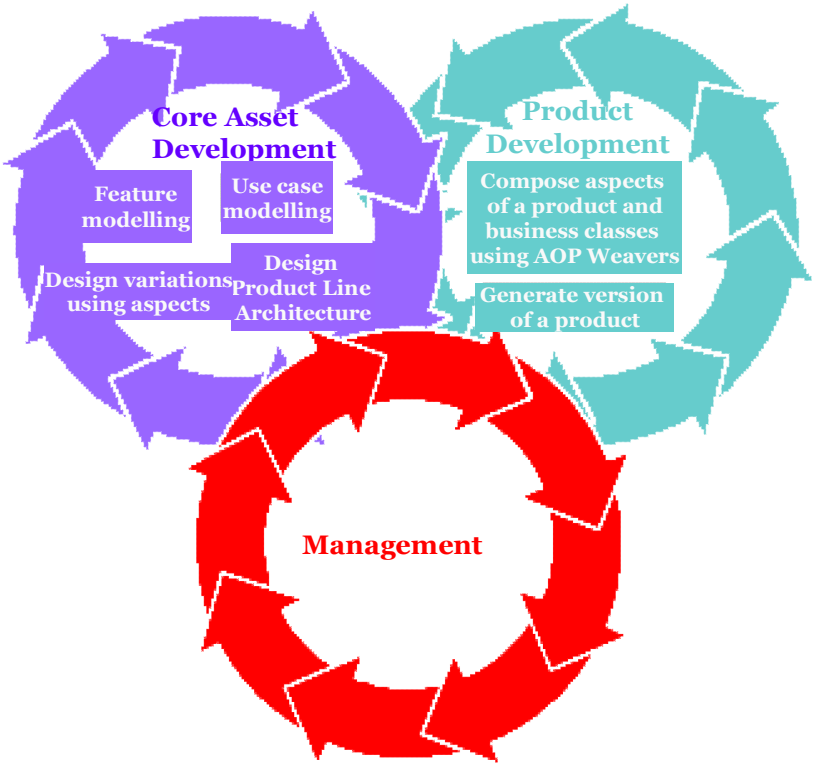

Figure 8. Proposed refinement of the PL framework.

The arrows indicate that each major activity is iterative and that the interaction between Core Asset Development and Product Development occurs in either way: assets created in the former are used in the latter; products created in the latter may lead to the development of new reusable assets. The Management activity is out of the scope of this work.

5. Conclusion

This work has proposed a refinement of the PL Framework. We have presented the combined use of feature models and AOP in order to model and implement PL variability, respectively. Variability modeling with feature models is concise and promotes better opportunities for reuse. Additionally, the mapping of variable features to aspects allows a modular description and implementation of the configurability of these variations, which the core PL architecture is oblivious of. Further, creating a particular PL product becomes a more automatic task, since a specific selection of product variable features maps to aspects previously implemented during Core Asset Development and these are weaved automatically with the core architecture to generate the final product during Product Development. By using these modeling techniques, we expect to boost reuse and productivity in the PL context.

Our approach is complementary rather than alternative to object orientation and some well established models. Indeed, we still employ use case and class diagrams; besides, design and architectural patterns are essential for defining the core PL architecture and for variation modeling with aspects.

According to the work of Bachmann et al. [3], we also notice the importance of explicitly representing variation and indicating architecture locations for which change has been planned. Our approach, however, indicates such locations in the aspects view (which is independent of the architecture view), whereas their work indicates such locations within the PL architecture itself. Some alternative approaches to variation modeling and implementation in a PL context rely solely on object orientation. Morisio et al. [15] extended UML with a stereotype to model variation. Gimenes et. al [8] propose an object-oriented framework encompassing both commonality and variation representation in the Workflow Management System domain. Our research relies mostly on feature models and aspects, instead. Anastasopoulos et. al [2] give an overview of alternatives for product line implementation. Among other alternatives, they also consider an AOP approach, but no integration with features models is suggested as we present here.

We have initially considered a relevant domain, namely pervasive computing applications, where there is a significant demand for feature variation, and a high time-to-market pressure. Future work includes using more precise metrics to validate our results, and experimenting with other domains such as games and multimedia applications.

6 References

[1] V. Alves, P. Borba. An Implementation Method for Distributed Object-Oriented Applications. In *XV Brazilian Symposium on Software Engineering,* pages 161-176, Rio de Janeiro, 2001.

[2] M. Anastasopoulos, C. Gacek. *Implementing Product Line Variability*. Symposium on Software Reusability, 2001. ACM Press.

[3] F. Bachmann, L. Bass. *Managing Variability in Software Product Lines.* Symposium on Software Reusability, 2001. ACM Press.

[4] G. Booch. The limits of software. *Software Development Forum*, PARC, Palo Alto, CA, 2002.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996

[6] M. Fleury, F. Reverbel. *The JBoss Extensible Server.* ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro. 2003.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] I. Gimenes, F. Lazilha, E. Junior, R. Halmeman. A Component-Based Product Line Architecture for Workflow Management Systems *(in Portuguese).* Third Brazilian Workshop on Component-Based Development. São Carlos, September, 2003.

[9] M. Griss, J. Favaro, M. d'Alessandro. Integrating Feature Modeling with the RSEB. *Proceedings of the Fifth International Conference on Software Reuse. IEEE Computer Society* Press, 1998.

[10] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[11] I. Jacobson, M. Griss, P. Johnson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[12] K. Kang et al. Feature-Oriented Domain Analysis Feasibility Study. Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[13] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28 (154), December 1996.

[14] T. Massoni. Um Processo de Software com Suporte para Implementação Progressiva. Dissertação de mestrado. Centro de Informática – Universidade Federal de Pernambuco, Fevereiro, 2001.

[15] M. Morisio, G. Travassos, M. Startk. Extending UML to Support Domain Analysis. In Proceedings of IEEE International Conference on Automated Software Engineering. 2000, p. 321-324.

[16] L. Northrop. A Framework for Software Product Lines, 2002. http://www.sei.cmu.edu/plp/framework.html

[17] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal. Specifying subject-oriented composition. TAPOS, 2(3):179-202, 1996. Special Issue on Subjectivity in OO Systems.

[18] H.Ossher, Petri Tarr. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In *International Conference on Software Engineering,* pages 698-688. ACM, 1999.

[19] S. Soares. Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos. Dissertação de mestrado. Centro de Informática – Universidade Federal de Pernambuco, Fevereiro, 2001.

[20] S. Soares, E. Laureano, P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA'02, Object-Oriented Programming Systems Languages and Applications.* ACM Press, 2002.