AIPLE-IS: An Approach to Develop Product Lines for Information Systems Using Aspects

Rosana T. Vaccare Braga[1], Fernão S. Rodrigues Germano[1], Stanley F. Pacios[1], Paulo C. Masiero[1]

[1]Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
Department of Computing Systems
Caixa Postal 668 – 13560-970 – São Carlos – SP – Brazil

rtvb@icmc.usp.br, fernao@icmc.usp.br, stanley.pacios@gmail.com, masiero@icmc.usp.br

Abstract. *Product lines for information systems present variabilities both in non-functional and functional features. Aspects are being used successfully in the implementation of non-functional features, as they provide intuitive units for isolating the requirements associated to this type of features. However, aspects could also be used to implement some product line features that refer to functional requirements. Using that approach, the instantiation of specific products could be done by combining the desired aspects into the final product. In this paper, we propose an approach, named AIPLE-IS, to incrementally build a product line for information systems using aspects. The product line core is developed first, followed by the addition of optional features through aspects. A case study for a product line in the domain of information systems for psychology clinics is presented to illustrate the approach.*

1. Introduction

Object-oriented Programming (OOP) is an established programming paradigm, with well defined development processes, e.g. the Unified Process (Jacobson et al 99). On the other hand, Aspect-Oriented Programming (AOP) (Kiczales et al, 97; Elrad et al, 01) is a relatively new programming technique that has arisen to complement OOP, so the software community is still exploring it and evaluating its costs and benefits.

Research about concepts and languages for aspect orientation (AO) has already attained a mature stage. However, processes for AO are still topics under study (Baniassad et al, 06). Recent works by several authors (Pearce and Noble, 06; Griswold et al, 06; Apel et al, 06) have contributed to solve specific problems of different development phases. In particular, research focused on dealing with aspects in the early development stages of requirements engineering and architecture design are gaining more focus in the last few years (Baniassad et al., 2006).

The need for techniques that help design and develop better quality software in less time is one of the software engineering concerns. Many software products are developed for artifacts already specified and implemented using software reuse techniques. In this context, the software product line (SPL) approach appears as a proposal for software construction and reuse based on a specific domain (Bosch, 00).

This technique has already shown its value on OO development, and can both benefit AO development and benefit from it.

In this paper product line engineering is considered as the development of software products based on a core architecture, which contains artifacts that are common to all products, together with specific components that represent variable aspects of particular products. Product line commonalities and variabilities can be represented as system features (Kang et al, 90) and they can be related both to functional or non-functional software requirements. Thus, it is interesting to investigate how aspects can improve modularization of SPL parts, isolating interests and benefiting SPLs, allowing the creation of more pluggable and interchangeable features.

In this paper, we propose and approach for incrementally developing an SPL, in which aspects are used in a systematic way to ease the introduction of functional features in the SPL, without changing the remaining features. The approach has been created based on a concrete product line development, which refers to a psychology clinic control system. In brief, the motivation for developing this work is the need for processes and techniques for aspect-oriented analysis and design; the growing interest of the software community in early aspects; and the need for approaches to develop aspect-oriented product lines.

The remaining of this paper is organized in the following way. Section 2 gives an overview of the proposed approach, named AIPLE-IS. Section 3 presents the SPL core development in more details, while Section 4 describes the product creation phase. A case study to illustrate the approach is presented along sections 3 and 4. Section 5 discusses related work. Finally, Section 6 presents the conclusions and ongoing work.

2. Overview of the proposed approach

Our approach for Aspect-based Incremental Product Line Engineering for Information Systems (AIPLE-IS) is illustrated in Figure 1. It has two main phases: Core Development and Product Creation. The Unified Modeling Language –UML (Rational, 00) is used as a modeling notation, combined with artifacts from the Theme/Doc approach notation (Clarke et al, 05). In the first phase (*Core Development*), a domain analysis is done to identify both fixed and variant points of the domain. The fixed part is implemented in this phase and is here denoted as the SPL core assets, because they define the minimum features that a single product of the family will have. These core assets are implemented using aspects where necessary to ease the future inclusion of variant features in the subsequent phase, as explained in Section 3.

In the second phase (*Product Creation*) several iterations occur to develop specific features needed to produce SPL concrete products. Each increment will result in a set of features needed to obtain a particular product, but that can also be reused in other products. Aspect-oriented techniques are used whenever possible to isolate features into aspects. Products are obtained by composing aspects and base code according to specific requirements. This activity can be executed as soon as the core assets are implemented, as there may be products that consist only of basic functionalities, or it can be executed later by combining basic and optional features.

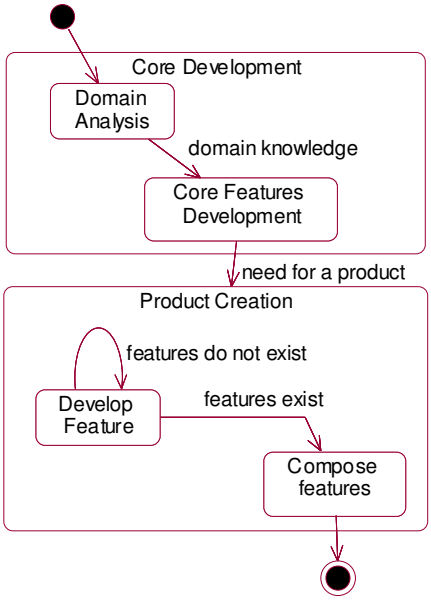

Figure 1. AIPLE-IS overview

3. Core Development

This phase aims at identifying and implementing the SPL core assets. It has two activities, as shown in Figure 1: domain analysis and core features development.

3.1. Domain Analysis

The domain analysis is conducted to capture the domain knowledge, i.e., to identify the functionality present in different applications of the same domain. This activity is extensive and, thus, is out of the scope of this paper to describe it in detail, as any existing domain analysis method could be used, such as those by Prieto-Diaz (1990) or Kang et al (1990). Gomaa (2004) also presents an approach to domain analysis that contains most of the good principles used by these authors, but his process is updated according to more recent notations.

The domain knowledge obtained in this phase should be properly documented to identify the SPL features, which can be mandatory, optional, or alternative. Mandatory features are those that should be present in all SPL members. Optional features can be present in one or more SPL members. Alternative features form a small set of features from which one or more are chosen to be part of an SPL member (exclusive alternative is also possible). The features model notation (Kang et al, 90) is used and a number is added to each feature to ease its future reference in subsequent phases. During domain analysis, it is important to discover mainly the mandatory features, and also those that are more likely to be needed later. More uncommon features are searched secondarily.

The domain analysis phase is outlined in Figure 2 (using BPMI notation (Arkin, 2002)), which shows its activities and artifacts produced. As it can be observed in the figure, the process starts with the study of one or more systems in the domain aiming at

creating, for each of them, a Features Document, a Requirements Document, a Features Model, a Conceptual Model and a Feature-Requirements Mapping. Those are named "individual" versions, i.e., each of them represents a single system. Other domain study activities can be used to help in the creation of these documents, as well as to eventually help in the domain analysis or even in AIPLE-IS subsequent phases.

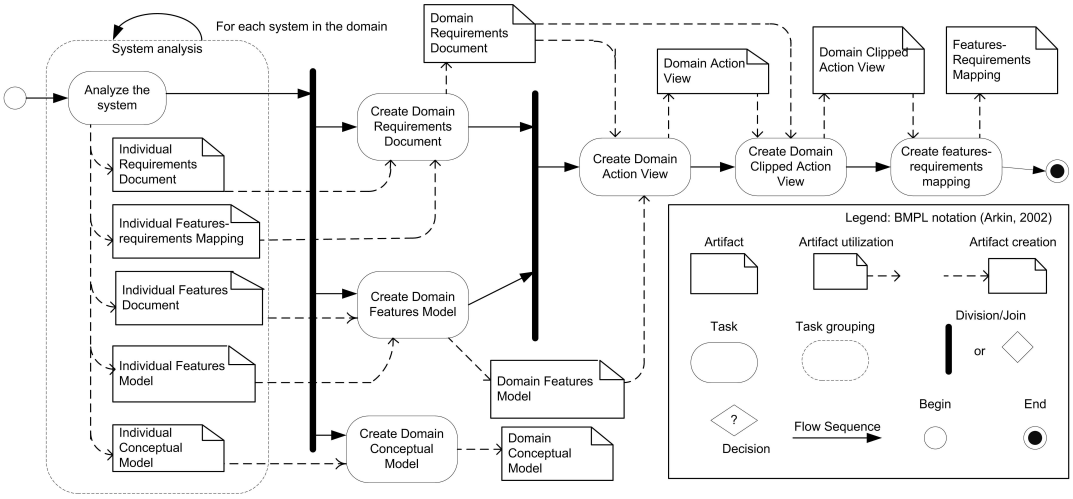

Figure 2. AIPLE-IS - Domain Analysis activities

In order to complete the domain analysis, a final version of each artifact is produced, named "domain" version, so that these domain versions encompass and organize all the content of the individual versions. Thus, a Domain Features Model is produced based on individual features documents and models, a Domain Conceptual Model is produced based on individual conceptual models, a Domain Requirements Document is produced based on individual requirements documents, and a Features-Requirements mapping is created based on individual features-requirements mappings. The other documents presented in Figure 2, as for example the action view and the clipped action view, are optionally created to help clarifying to which feature a requirement belongs to. They are based on the Theme approach notation proposed by Clarke (2005). The difference is that Clarke uses them to represent individual systems, while here they are used to represent the entire domain. In this phase we are not worried about which features are crosscutting or not.

To illustrate the usage of AIPLE-IS, we introduce an example implemented as part of a master thesis at ICMC-USP, where AIPLE-IS was used to develop part of a psychology clinic control system product line, here simply referred to as "PSI-PL". The possible products instantiated for this SPL are systems for managing several similar but different psychologist offices, psychologist hospitals, and other similar institutions. The PSI-PL domain analysis has been conducted based on the reverse engineering of three systems: the first is a private small psychologist office, and the other two are different hospitals (one private and one public) dedicated to attend psychology patients. The reverse engineering produced several individual models that were then used as basis to produce the domain model. A small part of the PSI-PL domain conceptual model and of the features model are shown in Figure 3 and Figure 4, respectively. These models illustrate several domain concepts, which can be mandatory or not (in the features model

of Figure 4, features with a filled circle are the mandatory ones, while those with a hallow circle are optional features). Figure 5 illustrates a small piece of the Domain Requirements Document (requirements were simplified to be presented here). Other artifacts obtained, such as the Domain Features Document, and the Features-Requirements Mapping are not shown here due to space restrictions.

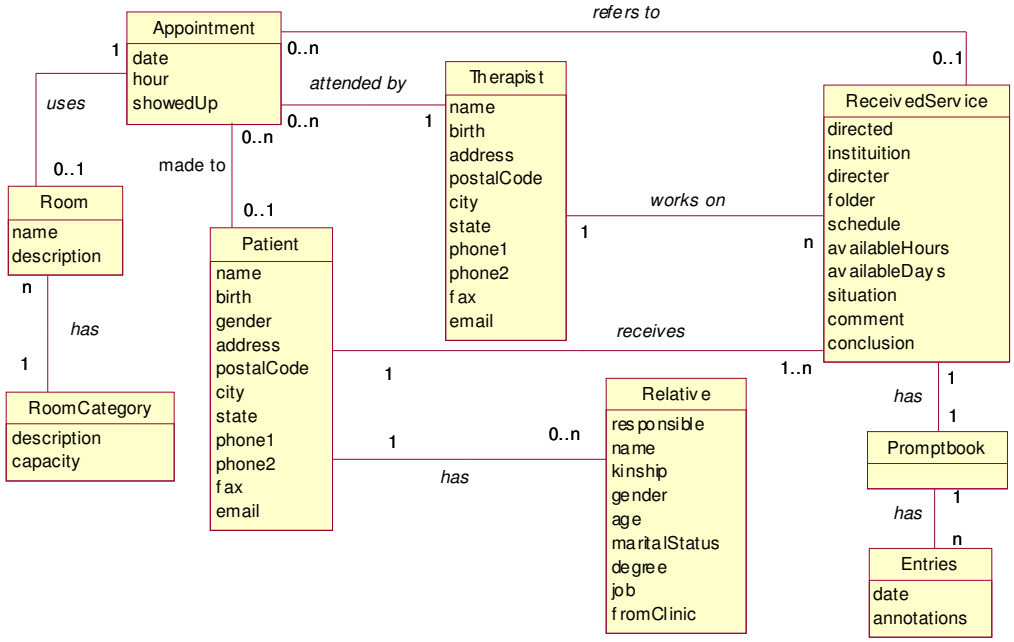

Figure 3. Partial Domain Model

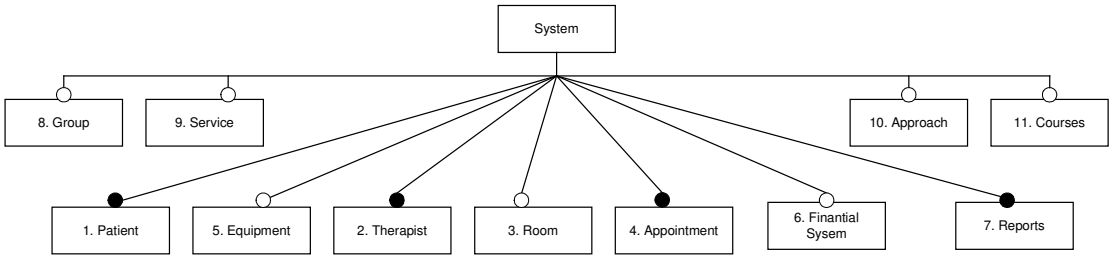

Figure 4. Partial Features Model

1 – The system should allow the inclusion, search, modification, and removal of patients from the clinic. Patients have the following attributes: name, birth date, address, zip code, state, phone, e-mail, identification document number.
7 – The system should allow the inclusion, search, modification, and removal of information about the service that the patient is receiving at the clinic, with the following data: therapist name, type of service, patient name, available dates/times, diagnosis, …
24 – The system should allow the inclusion, search, modification, and removal of appointments, containing the following data: patient name, therapist name, room, day/time scheduled, and service to be performed.
42 – The system should allow the management of information about the possible types of service offered by the clinic.

Figure 5. Example of three domain requirements

3.2. Core Features development

The core features development aims at implementing all SPL common features. It includes activities such as defining the SPL core architecture, designing the software, implementing, and testing it. This is an extensive activity that presents many of the issues of developing a conventional system and, additionally, some more specific issues that arise due to the fact that we are developing a product line using aspects. Thus, we recommend the use of an object-oriented programming language that has an associated extension to support aspect-oriented characteristics. In the PSI-PL case study, Java and its aspect extension, AspectJ, have been used (AspectJ, 2006). MySQL relational database was used for objects persistence.

The definition of the SPL architecture depends on several factors related to the particular project issues, such as non-functional requirements that can influence on performance, flexibility, etc. For example, the architecture could be based on an object-oriented framework, on a set of components, or simply on a set of objects. In a lower level, it should be decided whether to use a layered-architecture, for example. These decisions involve the particular interests of the organization, so we consider this phase as being out of the scope of this paper. The PSI-PL architecture followed the MVC pattern (Buschmann et al., 1996).

After defining the architecture, the mandatory features are analyzed, designed, and implemented. The features model (produced in the previous phase) is a source for identifying the mandatory features. For example, in the PSI-PL case study, domain engineers have determined that the product line core base should consist of features Patient, Appointment, and Therapist. This is the minimum functionality expected from a member of the product line, probably used in small psychologist offices. AOP is used in this phase to provide a mechanism through which optional features are more easily introduced in the subsequent phase.

Even intending to isolate features, in certain moments they influence one another, as the final system expected behavior contains the interaction among features. The development of the features that influence other features is easier if the features that will be influenced are already designed and implemented. If they are not, the design of the influence is postponed until they appear in the design. So, a practical advice is to create first the features that are more independent of others. For example, in the PSI-PL case, requirement #1 of Figure 5 describes the Patient feature and it is easy to see that it is independent of other features, so it should be created first. The same is true for feature Therapist. On the other hand, as can be seen on requirement #24 of Figure 5, Appointment depends on both Patient and Therapist, so its creation should be delayed. To identify the influence among features, the clipped action view diagram developed in the analysis phase can be used, as exemplified in Section 4.1.

To ease the isolation of features and the identification of their dependencies, AIPLE-IS suggests the development in three steps involving analysis and design, as can be seen in Figure 6. Each step produces a part of the feature design. The first step creates the design part that deals with the feature interest more strictly and exclusively as possible, i.e., free of other features influence. To make this possible, the requirements associated to the feature are rewritten to withdraw any behaviors that might refer to other features. Then, the analysis and design proceed, creating use cases, class diagrams,

collaboration diagrams, etc., similarly to conventional OO analysis and design. Information about the feature is obtained from the artifacts resulting from domain analysis phase. The numbering present in the features model is used, together with the features-requirements mapping, to find the corresponding detailed requirements.

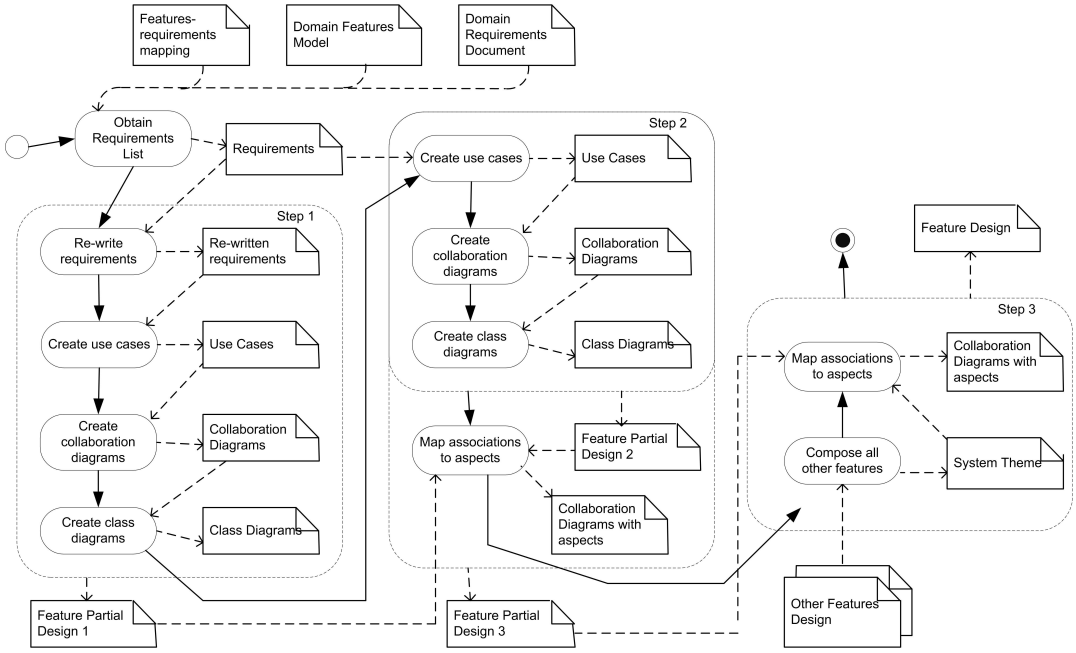

Figure 6. AIPLE-IS – Features Development activities

As an example, consider the design of the feature relative to requirement #24. It could be rewritten so that only Appointment is mentioned, i.e., citations to Patient and Therapist are withdrawn. The result is the requirement "The system should allow the inclusion, search, modification, and removal of appointments, containing the following data: room, day/time scheduled, and service to be performed". This requirement is used as basis for creating use cases, class diagrams, etc.

The second step creates the feature design part that exists due to the presence of other features. Here, the requirements are reviewed to consider those that have interests tangled and scattered in the requirements document. A refined design is created for the same feature, now considering the presence of other features. It is at this point that AOP begins to act. Once the elements to be added by other features are known, they are designed separately, allowing a modular implementation using AOP. Special annotations are done in the class diagrams and collaboration diagrams to denote the presence of aspects. We do not show this here due to space limitations, but any extension of UML to deal with aspects can be used. In the PSI-PL example, aspects are used in this step to create the associations between Appointment and Patient and between Appointment and Therapist. This aims at easing possible subsequent changes in associations, as explained in Section 4.1 and is according to the idea of relationship aspects introduced by Pearce and Nobel (2006).

The third step creates the feature design part that exists due to the new feature influence in the rest of the system. In order to find out exactly which influence the

feature causes in the rest of the system, it is required to have the design of the rest of the system. This is obtained using the composition of base themes (also according to the Theme approach). All existing features of the SPL are composed together to obtain this design, which is named as the "System" theme. Having the design of the System theme, it is possible to identify how the feature under development will be composed with the rest of the system. In the PSI-PL example, as we are beginning the development, we do not have a System yet, so this step is skipped. If we had a system, we would have to check how Appointment influences the rest of the system.

There are some guidelines that have to be followed to implement the features. They are summarized in Section 4.1, as they are common both to mandatory and optional features, and can be found elsewhere in more detail (Pacios et al, 06).

4. Product Creation

This phase aims at creating the concrete products and increasing the features repository. It has two activities, as shown in Figure 1: develop feature and compose features. The products are created on demand, and optional features are developed only when requested by a specific product.

4.1. Develop Feature

This activity is responsible for incrementally adding new features to the product line, using aspect techniques when appropriate, until all variable features identified in the domain analysis are developed. In fact, this phase can be extended as needed to add new features identified after the domain analysis, as part of the product line evolution. To implement a particular feature, the following general guidelines have been proposed (Pacios et al, 06):

- *G1 - New classes*: if a feature implies in the creation of one or more new classes, these should be implemented as conventional classes (with no need to use AOP);
- *G2 - New attributes and/or methods*: if the feature implies in the creation of new attributes or methods in existing classes, they could be introduced into the existing classes through intertype declarations, but other mechanisms could be used, for example, the properties pattern (Yoder et al., 2001);
- *G3 - Change in the behavior of existing methods*: if the feature existence implies in the modification of existing methods, this is done with aspects and advices;
- *G4 - New association*: if the feature implies in creating new associations between existing classes, or between a new class and an existing one, they are implemented with aspects, to guarantee the connectivity with the feature and its removal if necessary. N to 1 associations are generally implemented through an attribute included in one of the classes (the N side) to represent the reference to the other class (the 1 side). So, guideline G2 is applicable here.
- *G5 - Removal of existing associations*: if the presence of one feature requires removing one or more associations between existing classes (probably to add other different associations), then a mechanism is needed to remove them. To make that possible, the existing associations should have been included through aspects, so that just omitting the aspect that included it, is enough to remove the association.

As an example, consider the PSI-PL again. After the core features were implemented, it was decided to include the "Room" feature, which consists of allowing the control of rooms where appointments occur. This was an optional feature (see Figure 4), and it implied in the creation of two new classes, Room and RoomCategory. Room is associated to an existing class, Appointment. The implementation of this new feature was quite simple to execute using aspects. Following guideline G1, a new class, Room, was created, together with an aspect to introduce the new attribute (roomNumber) in the Appointment class to represent the association between Appointment and Room (G4).

More specific guidelines that need to be observed during the features implementation are summarized next. They are more suitable for the development of information systems, considering that in this work the three-tier architecture has been chosen (with an interface layer, a business application layer, and a persistence layer), and persistence is done using relational databases.

For each existing class of the business application layer that receives new attributes or methods, an aspect is created to: introduce new attributes and respective methods; supply the initial value for the new attributes; guarantee that the class methods that handle its attributes also handle the new attributes; and treat possible business rules associated to these new attributes.

To ease the introduction of new attributes and their treatment, meta-attributes can be used: one named "fields" and another named "values" (these can be vectors whose elements are strings with the fields and values names, respectively). The use of meta-attributes makes it possible for the aspects to introduce their new attributes in the corresponding meta-attribute, avoiding having to create an advice or intertype declaration to include new attributes. Functions that receive all object attributes by parameters, or that return all these attributes, are modified to receive and return objects of vector type. A particularly common case of functions of these types are the database query functions. On the other hand, by using meta-attributes the advantage of static variable checking is lost. Other possible solutions would be to use the Java language reflection or active object models (Yoder et al., 2001).

This same guideline can be applied to include associations between classes. The association is represented by a reference from one class to the other. So, a field can be added in both vectors to deal with the referential attribute. The additional methods necessary to handle the new attributes are included through intertype declarations.

The interface layer has to reflect the modifications that occur in the application business classes that they represent. In the particular case of information systems, most application classes have a corresponding graphical user interface (GUI) class, which might need new widgets placed on them due to the inclusion of new attributes. So, a mechanism to introduce these widgets in GUI classes is needed. A possible solution is to divide the construction of the GUI screen in several parts, so that it is easy to identify pointcuts where to place the new widgets and the respective treatment. For example, the GUI creation method should have at least four parts: create variables, initialize variables, position the widgets on the screen, and treat events related to the widgets. Thus, an aspect can be created for each GUI class, and advices can be used to introduce the treatment of the new attributes in each method.

Returning to the PSI-PL example, in terms of its GUI, after introducing the new "Room" feature, it is necessary to include an additional widget so that the final user can choose the room where the appointment is scheduled. This can also be done with an aspect, which adds this widget and corresponding behavior to the GUI class responsible for entering the appointment data.

So, this first evolution of the PSI-PL produced an increment that allows the creation of two different products: simple psychology office and office with room's allocation. The second iteration to evolve the PL considered that a patient can be scheduled not only to one therapist, but to a service that is performed by a therapist. This implies that one patient can be registered in several different services offered by a hospital, each of which is attended by a different therapist. For example, he or she can participate in a career advice therapy and in a couple therapy, so that different appointments are made for them. Service is an optional feature of PSI-PL (see Figure 4).

To design this feature, initially its requirements are rewritten to withdraw any behavior that do not belong to Service itself, as for example requirement #7 of Figure 5 is re-written as "The system should allow the inclusion, search, modification, and removal of information about the service received at the clinic, with the following data: type of service, available dates/times, diagnosis, …". This is enough to develop a complete design for the Service feature itself, without the influence of other features (Patient and Therapist in this case). Then, the artifacts Action View and Clipped Action View, obtained in the domain analysis, are used to help visualizing which part of the functionality results from the influence of other features. Figure 7 (a) shows the Action View corresponding to the Service feature, described in requirements 7 and 42, but also mentioned in requirement 24 (which deals with Appointment). It can be observed that features Patient and Therapist affect the Service feature directly through requirement 7.

When the features-requirements mapping was built, it was decided that the Service feature is dominant in requirement 7, so Patient and Therapist features will affect the Service feature. This decision is reflected in the clipped action view of Figure 7 (b). In requirement 24, the dominant feature is Appointment, as service is just one more detail in its main goal, which is to make an appointment. The creation of the clipped action view is the right moment to review decisions related to features-requirements relationships. The clipped action view indicates that Service will influence Appointment.

Finally, to finish the Service feature design, a comparison is done with the rest of the system to detect any intersections. In this case, this intersection is empty, as all classes are new and thus should be implemented simply using OO classes.

Regarding the organization of the product line code, to improve reuse of the features separately, code artifacts (such as classes and aspects) that refer to one feature should be put together in one or more packages. New classes can be placed in a separate package, and a package could be created for each new association among different features. That way, it is easier to reuse the new classes or just the associations. A features-packages mapping can be created to ease the subsequent features composition.

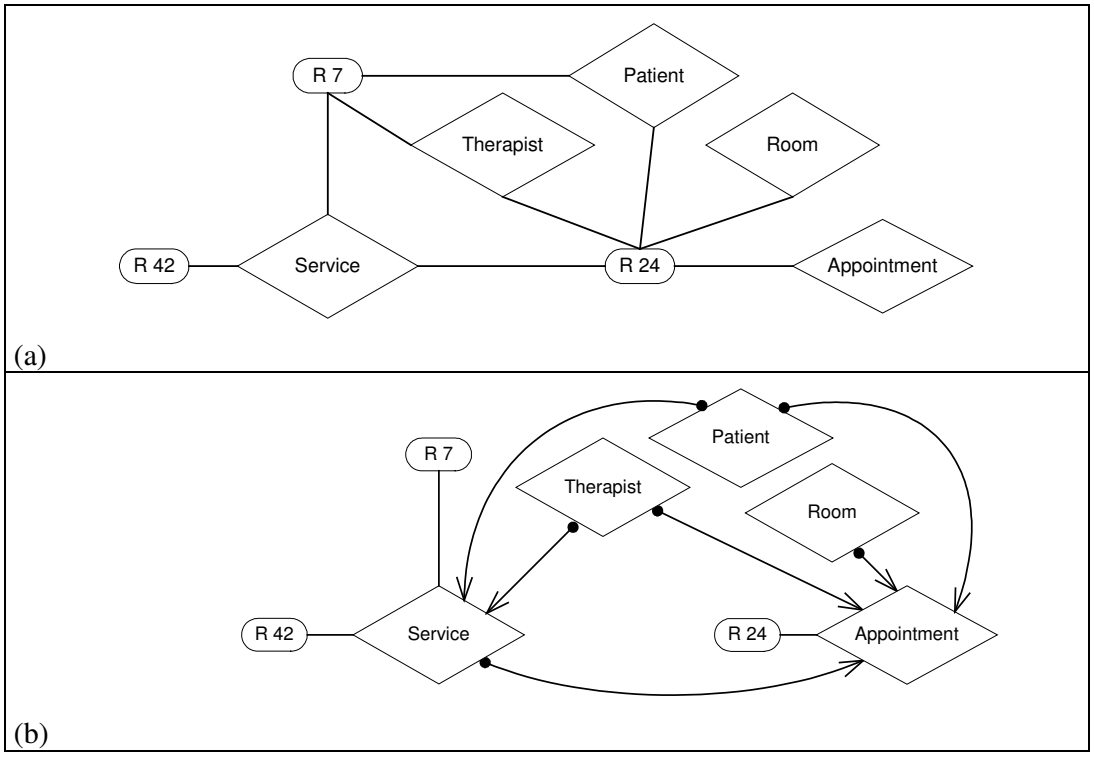

Figure 7. Action View (a) and Clipped Action View (b) for Service Feature

4.2. Compose Features

In this phase, concrete products are instantiated from the product line. The software engineer needs to choose the variabilities of a particular product and then use composition rules to join the base code with the code of each optional feature implemented in the product line. As aspects are used in the product line development, a special tool may be necessary to make this composition. In this work, the AJC compiler and byte-code weaver for AspectJ and Java were used.

Each feature chosen to be part of the product must be present in the features model. Then, the features-packages mapping is consulted to determine if the chosen feature requires other features. Thus, the total set of features that must be present on the product is obtained. Among these features, it is verified which are already implemented in the repository. Features that are still not implemented must be so (see Section 4.1), as they are developed on demand for the product instantiation.

Besides implementing the features, in the PSI-PL example it was necessary to implement the main interface of the system, which is not associated to any feature, but is required to allow the access to the system functionalities. It was also necessary to implement a persistence layer to persist objects into the MySQL relational database.

Several product instantiations were done to evaluate the proposed approach. The combination process was done manually, and several different products were obtained by combining the base code with the optional features produced so far.

5. Related Work

Several works have been proposed that relate aspect-oriented programming and product line development. Alves et al. (2004) describe an incremental process for structuring a PL from existing mobile device game applications, in which continuous refactoring of existing code is done using aspect-oriented programming to isolate commonality from variability within the products. In our approach, several refactorings mentioned by Alves et al. can be used as a means of isolating the aspects from the original code, but we do not impose the existence of a code, as our reverse engineering step aims at extracting knowledge about the domain instead of code itself. Another relevant difference here is that our work considers features in a high granularity level (for example Appointment is a feature) while in Alves work a feature could be a very fine-grained characteristic of a game, for example the way images are loaded.

Loughran et al. (2004) propose an approach that joins framing and aspect-oriented techniques to integrate new features in product lines. This allows parameterization and reconfiguration support for the feature aspects. The framing technology has the disadvantage of being less intuitive, besides needing previous knowledge of the subsequent features. An advantage is to parameterize aspects at runtime. Although these techniques are applicable only in the coding phase, we plan to investigate how they could fit our process.

Apel et al. (2006) propose the integration of aspects and features at the architectural level through aspectual mixing layers (AMLs), also aiming at incremental software development. Our approach uses the same idea of integrating aspects and features at the architectural level, and our composition of core features and optional features can be thought of as a way, although very simplistic, of having the same effects of using AMLs. However, we do not make use of special language constructs and we do not treat the problem of aspects dependency and precedence.

Mezini and Ostermann (2004) present an analysis of feature-oriented and aspect-oriented modularization approaches with respect to variability management for product lines. Their discussion about weaknesses of each approach indicate the need for appropriate support for layer modules to better attend the development of software for product families, improving reuse of isolated layers. Even using AspectJ in our approach, we try to make it flexible to allow the use of other languages that support AOP and overcome AspectJ limitations.

Although all these approaches have points in common with our approach, our focus is on proposing a systematic process through which product lines for information systems can be built using AOP, so parts of these approaches can be incorporated into our process. For example, crosscut programming interfaces (XPIs) (Griswold at al, 2006) could be used to decouple aspect code from the advised code; and languages like Caesar or the concept of AMLs could be used instead of simply using AspectJ.

6. Concluding Remarks

AIPLE-IS allows incremental development of product lines using aspects. It is composed of a sequence of short, well-defined and reproducible steps, with well-defined

activities and artifacts. It considers the product line features as the main application assets, so the features are isolated, encapsulated, and designed with aspect orientation.

AOSD techniques facilitated the SPL features implementation. With the encapsulation supplied by AO, features become more cohesive, easier to be combined, and reusable. The very nature of aspect-oriented programming is responsible for easing features combination. OOAD techniques have been combined with AOSD to optimize design with separation of concerns. The approach has integrated AOSD with AO implementation, i.e., it supplies the basis for creating the whole product design, and also AO implementation techniques (guidelines) that can be used aiming at a good code with high cohesion and low coupling, reinforcing reusability.

The guidelines for AO implementation can also be used independently of other approach activities. However, it is more guaranteed to have a good code having a good design. This also means that the approach deals with the aspects problem in the development initial phases. For example, the requirements are already grouped by features. This problem is being largely discussed nowadays in the software community (e.g. Baniassad et al, 06).

Although the incremental nature of AIPLE-IS is an advantage, the code can become more complex with the introduction of new features that may involve modification of associations among existing features. This causes the code to have less intuitive meaning, which can be a disadvantage and imply in more difficult maintenance. Ongoing work is being done to tackle with this problem.

To help instantiating products, application generators could be used. A master dissertation research is being conducted with this goal. Captor (Shimabukuro et al, 06) is an application generator that automatically builds applications based on high level specification languages persisted in a repository. It is being extended to allow aspect oriented composition. Finally, other case studies should be performed to validate AIPLE-IS with other examples, possibly in other domains.

References

Alves, V., Matos Jr, P., and Borba, P. (2004) “An Incremental Aspect-Oriented Product Line Method for J2ME Game Development”, Workshop on Managing Variability Consistently in Design and Code (in conjunction with OOPSLA´2004).

Apel, S., Leich, T., Saake, G. (2006) Aspectual Mixin Layers: Aspects and Features in Concert. In: Proc. of International Conference on Software Engineering, p. 122-131.

Arkin, A., 2002. Business Process Modeling Language (BPML), Version 1.0. http://www.bpmi.org/ (last access: december, 2006)

AspectJ. The AspectJ Project. Disponível para acesso na URL: http://eclipse.org/aspectj/, em 10/11/2006.

Baniassad, E. L. A., Clements, P., Araújo, J., Moreira, A., Rashid, A.; Tekinerdogan, B., 2006. Discovering Early Aspects. In IEEE Software, v. 23, n 1, p. 61-70.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7.

Buschmann F. et al., 1996. Pattern-oriented software architecture: A System of Patterns, Wiley.

Clarke, S.; Baniassad, E. L. A., 2005. Aspect Oriented Analysis and Design. Addison-Wesley Professional, ISBN 0321246748.

Elrad, T.; Filman, R. E.; Bader, A., 2001. Aspect Oriented Programming, Communications of the ACM, 44(10), October.

Gomaa, H., 2004. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley.

Griswold, W. G.; Shonle, M.; Sullivan, K.; Song, Y.; Tewari, N.; Cai, Y.; Rajan, H., 2006. Modular Software Design with Crosscutting Interfaces. IEEE Software, vol. 23, no. 1, p 51-60.

Jacobson, I.; Booch, G.; Rumbaugh, J., 1999. The Unified Process. IEEE Software (May/June).

Kang, K., et al., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Kiczales, G. Lamping, J. Menhdhekar, A. Maeda, C. Lopes, C. Loingtier, J. M. Irwin, J., 1997. Aspect-oriented programming. In: Proc. of the European Conference on Object-Oriented Programming, Springer-Verlag, p. 220–242.

Loughran, N., Rashid, A., Zhang, W., and Jarzabek, S. (2004) "Supporting Product Line Evolution with Framed Aspects". Workshop on Aspects, Components and Patterns for Infrastructure Software (held with AOSD 2004).

Mezini, M. and Ostermann, K. (2004),Variability Management with Feature-Oriented Programming and Aspects, Foundations of Software Engineering (FSE-12), ACM SIGSOFT.

Pacios, S. F.; Masiero, P. C.; Braga, R. T. V., 2006. Guidelines for Using Aspects to Evolve Product Lines. In: III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, p.111-120.

Pearce, D. J.; Noble, J., 2006. Relationship aspects. In: Proc. of the 5th international conference on Aspect-oriented software development, p. 75-86.

Prieto-Diaz, R.; Arango, G., 1991. Domain analysis and software system modeling. IEEE Computer Science Press Tutorial.

Rational, C., 2000. Unified Modeling Language. Available at: http://www.rational.com/uml/references (last access: December, 2006).

Shimabukuro, E. K.; Masiero, P. C.; Braga, R. T. V., 2006. Captor: A Configurable Application Generator, Proceedings of Tools Session of the 20th Simpósio Brasileiro de Engenharia de Software, p.121-128 (in Portuguese).

Yoder, J.W.; Balaguer, F.; Johnson, R., 2001. Architecture and Design of Adaptive Object Models. SIGPLAN Not. 36, p. 50-60.