Pattern Oriented Approach for the Design of Frameworks for Software Productlines

Rajasree M S, Janaki Ram D, Jithendra Kumar Reddy
rajasree@cs.iitm.ernet.in, djram@lotus.iitm.ernet.in
Distributed & Object Systems Lab
Department of Computer Science & Engineering
Indian Institute of Technology, Madras, Chennai, India

Abstract

A productline is a collection of products that share a common set of features and allows for controlled variations. The architecture for any productline is not different from an application framework, i.e. a set of cooperating classes that make up a reusable design for a specific class of software. Frameworks facilitate the development process by partitioning the design into concrete and abstract classes. This paper describes a systematic approach to framework design by identifying the best design alternatives for the hot-spots in the design. Pattern Oriented Technique (POT) is used to come up with an initial design and this design can be modified in subsequent iterations. A set of pattern level measures based on an abstract model called pattern graph is used to quantify the tailorability of the framework.

Keywords: *Product Line Architecture (PLA), Pattern Oriented Technique (POT), Pattern Graph*

1 Introduction

A *Product Line Architecture* (PLA) is a design for a family of applications. The products in a software *Productline* share a common, managed set of features that satisfy specific needs of a market or mission. PLAs have become increasingly important in software development process since they attempt to capitalize the domain expertise of companies and facilitate large-scale reuse in a systematic way. Researchers have emphasized the need for treating software reuse as an important issue because of the information-rich nature of software assets [1]. It is interesting to note that in all other engineering disciplines, reuse is an integral part of good engineering design itself. Hence it is not necessary to treat the reuse aspect separately when systems are designed. This is in contrast to software design where reuse is mostly opportunistic. In order to facilitate planned reuse, sound scientific foundation that encompasses relevant design principles need to be transformed to working practical solutions. Only by means of this approach can software engineering mature itself as any other engineering discipline. The development methodology used for productline framework in our approach stresses these aspects by proposing a design-level reuse approach, using design patterns to model variabilities in PLAs.

PLAs are designed to amortize the effort of software design and development over multiple products. Hence it is very important to plan the design stages of the PLA evolution such that commonality in the products is abstracted out and provision is made for controlled variations. Since the up-front investment in the development of PLA is high compared to single product development, enough care should be taken to see that the architecture designed is capable of addressing the development of a large number of products with less development effort. The methodology adopted in this paper uses a pattern oriented approach for the design of a productline architecture. This is achieved by means of a set of carefully designed design level measures captured from an abstract level model of the design called a *Pattern Graph*. This model corresponds to the design patterns that are identified in the design. These measures can be used to specify the ease of adaptation of the design for various products in the productline.

The paper is organized as follows. Section 2 discusses the use of design patterns as variability mechanisms in frameworks. Section 3 discusses Domain Engineering. Section 4 explains the various steps in the development process. Section 5 briefs a few approaches related to productline development and Section 6 concludes the paper.

2 Use of Design Patterns As Variability Mechanisms in Frameworks

A *framework* is a collection of abstract classes that encapsulate common algorithms of a family of applications [8]. It makes up a reusable design for a specific class of software and provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations [9]. A developer can customize the framework to a particular application by sub-classing and composing instances of framework classes. Similar to productline, framework also addresses a domain or product family. Both provide guidelines as how to decompose a problem and design subsystems. The design of a framework comprises the identification of classes and their collaborations along with the realization of the shared invariants across various products in the productline. The hot-spots corresponding to the variabilities in the domain can be captured in the form of design patterns. Patterns are ideal for composing variability because they not only serve as structuring mechanisms, but also describe relationship within and between parts of a system. Thus framework serves as a generic application that allows creation of different applications of an application family (domain). The idea of using design patterns for the construction of frameworks is explored in [13], where design patterns are identified as micro- architectural elements of a framework. Also, patterns are seen as entities that are smaller and more abstract than frameworks.

3 Domain Engineering

Design patterns and frameworks speak from the solution end and object-oriented techniques like polymorphism, encapsulation and various forms of inheritance tend to focus on the problem end concentrating on a single application. As a result, in the former case, the system designer is left with a choice of selecting an "appropriate solution" for the problem at hand from the solutions that he has, whereas the latter technique provides solutions which are less general. Consequently both these techniques of reuse fail to provide as much as reuse as they are advertised about.

Domain Engineering (DE) can be used to circumvent the above difficulty. It is an activity used for building reusable software artifacts. It comprises *Domain Analysis* (DA) and *Application Engineering* (AE). DA includes activities like identifying, collecting, organizing and representing relevant information in the domain. A key activity during this phase is a systematic analysis of commonality and variability that is prevalent in the productline . DA results in Domain Model which represents the sets of requirements common across all the products in the productline. AE process develops software products from software assets created by the DE process. In fact DE and AE are complementary, interacting parallel processes. Domain model forms a generic design for the productline. Domain analysis addresses a family of products whose context and requirements form the problem domain, and the solution forms the application domain.

4 Framework Development Steps

The steps that constitute the development of the framework are explained in the following sections. An iterative approach to design is envisaged in this paper. During each iteration, parameters which measure the ease of adaptation of the framework at compile time and run-time are measured using the approach suggested in [10]. This method enables to change the adaptability measures during each step in the development process.

4.1 Development of Feature Model

Feature Oriented Domain Analysis (FODA) [5] is carried out which generates the feature model. Feature model forms the infrastructure for the entire development since it depicts all the features by means of feature diagrams. Feature Diagrams contain hierarchies of feature trees with mandatory, optional and alternative features. Optional and mandatory features are also described as variation points [4]. Features are abstractions of requirements. Since requirements can be abstracted at multiple levels, variation points can occur at multiple levels in the development phases of the product and can address multiple levels of granularity. A particular requirement may apply to several features and a particular feature can be applied to fulfill more than one requirement (a n:n-relation) [11].

4.2 Identification of Classes, Responsibilities and Interactions

This step identifies the classes to be incorporated in the design and their responsibilities based on the feature model. The interactions among classes are also identified. Representation of features as suggested in [12] using UML notations can be used here. This model has the potential to depict a product in the productline by means of its features and its interactions using object collaborations since it combines UML and feature model.

4.3 Identification of Design Patterns at Abstract-level

During this step, related classes are grouped to form class groups and the interactions among these groups are identified. As a result, classes corresponding to each of the variable aspect of a feature fall in the same class group. These classes correspond to the realization of alternative, multiple or mutually exclusive options of a single feature. Once these class groups are formed, interactions among them are specified abstractly. Interaction among classes is the key idea of formation of a design pattern. Since we group related classes together, class group interactions form the basis of identifying patterns in an abstract manner. We do this because the model used for quantifying the attributes of the design called a *pattern graph* captures the fixed and variable parts of the design in an abstract manner [10]. Also, the variabilities inherent in the feature model will automatically be captured by means of these patterns since patterns inherently capture the variability in design.

4.4 Rough Design and Design Evolution Using Pattern Graph Design Measures

The patterns formed in the previous step are used as the starting point in the evolution of the design. A measurement model for pattern called a *pattern graph* is used for quantifying the design [10]. In a Pattern Graph, classes are represented as rounded rectangles, having three partitions, corresponding to template methods, hook methods and rigid methods. Hook methods are those which are declared in a class and defined in the same class. Methods which call at least one hook method are known as template methods. Rigid methods are those which are defined and declared in the same class. From the viewpoint of cost, reusability and maintenance, four key attributes for a pattern are identified in viz. *size, static adaptability* (SA), *dynamic adaptability* (DA) and *extendability* (EX). These attributes can be quantified.

The measures which we are mostly interested in productlines are the ease of tailorability of the architecture. SA and DA capture this. SA measures the ease with which a hook method can be defined in the subclass without worrying about other hook methods in the pattern. The higher this value, lesser the number of methods which we are forced to define when we try to define a hook method and the easier it is to adapt the design. SA is given by the following formula.

$$SA = \frac{\sum_{i=1}^{NC} H_i}{\sum_{i=1}^{NC} {H_i}^2}$$

Here, H corresponds to the number of hook methods in each class and NC is the total number of classes in the pattern graph. DA is a measure of the extent to which a patterns structure facilitates object composition which is given by the the formula:

$$DA = w_{DA,SCH}.SCH + w_{DA,CCH}.CCH + w_{DA,CWR}.CWR$$

Here, SCH and CCH represent the number of simple calls which have hook partitions at their target end and number of composite calls which have hook partitions of a class at their target end. CWR represents the number of classes with rigid methods only which do not call hook or template method.

$w_{DA,SCH}, w_{DA,CCH}$ and $w_{DA,CWR}$ are the weights which is a measure of the number of times an object corresponding that particular class is modified or replaced. These values can be assessed only after studying the system over a period of time. So, during the design stage we can assume a value of 1. A detailed treatment of these measures is available in [10].

Thus SA and DA provide the ease of adaptation of the framework. Once these measures are identified, the design can be modified based on the requirements of the domain by refactoring it.

5 Related Work

PLAs have been recognized by software community long way back when McIlroy initiate ed them early in 1969 [2]. Information-hiding principle by Parnas encodes a module's commonalities as its interface and variabilities as a module's secrets [3]. RSEB (*Reuse-driven Software Engineering Business*) [4] addresses development of application product families taking into account their organizational and technical issues . RSEB and FODA [5] are integrated in *FeatuRSEB* [6]. This reuse-oriented model serves as a catalog to link use cases, variation points, reusable components and configured applications. The importance of scoping in software productlines and a methodology to achieve this is suggested in [7]. None of these approaches suggest a measurement approach to improve the design. That is where our approach significantly differs.

6 Conclusions

Design patterns are popular in software design for capturing the experience of designers. The approach used in this position paper describes how OO designs can be composed from design patterns. Pattern graph abstraction proves to be powerful in modeling productline architectures since it accounts for the variabilities in the architecture by means of template and hook methods. Also, it is important to see that the hook methods capture variabilities that are present in the architecture. The effect of one hook method on other hook methods is taken care of in the measurement model. Hence, the design measurement approach takes into consideration the effect of one variability on the other. In other words, the effect of various interacting features in the design is captured in this measurement approach. Another important aspect in our approach is the concept of adjustable weights which can play a major role in the domain of architectural evolution.

References

[1] A. Mili, Sh. Yacoub, E. Addy, H. Mili, "Toward an Engineering Discipline of Software Reuse," *IEEE Software*, pp. 22–30, September/October 1999.

[2] D. McIlroy, "Mass Produced Software Components," *Software Engineering: Report on a Conference by the Nato Science Committee*, pp. 138–150, October 1968.

[3] D. L. Parnas, "On The Criteria to be Used in Decomposing Systems into Modules," *Communications of ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[4] M. L. G. I. Jacobson and P. Johnson *Software Reuse Architecture, Process and Organization for Business Success*, Addison Wesley 1997.

[5] Kang K. S. Cohen Hess J. Nowak W. and Peterson S *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report No. CMU/SEI-90-TR-21 Software Engineering Institute Carnegie Mellon University Pittsburgh, PA, 1990.

[6] M. Griss, J.Favaro, M. d. Alessandro, "Integrating Feature Modeling With RSEB," in *Proceedings of the Fifth International Conference on Software Reuse*, pp. 76–85, IEEE Computer Society, 1998.

[7] K. Schmid, "Multi-Staged Scoping for Software Product Lines," in *Proceedings of Software Product Lines: Economics, Architectures and Implications, Workshop No.15 at 22 nd International Conference of Software Engineering (ICSE 2000)*, (Limerick, Ireland), pp. 19–22, June 10 th 2000.

[8] R. Johnson, B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, June/July 1988.

[9] E. Gamma, R.Helm, R.Johnson, J.Vlissides *Design Patterns, Elements of Reusable Object Oriented Software*, Addison Wesley 1995.

[10] D. Janaki Ram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S.V.G.K. Raju and A. Ananda Rao, "An Approach for Pattern Oriented Software Development Based on a Design Handbook ," *Annals of Software Engineering*, vol. 10, pp. 329–358, 2000.

[11] M. S. Jilles van Gurp, Jan Bosch, "On the Notion of Variability in Software Product Lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Institute of Electrical and Electronics Engineers, Inc., 2001.

[12] H. Gomaa, "Modeling Software Product Lines With UML," in *Proceedings of Software Product Lines: Economics, Architectures and Applications, Workshop No. 3 at 23 rd International Conference of Software Engineering (ICSE 2001)*, (Toronto, Ontario, Canada), pp. 27–31, May 13 2001.

[13] Ralph E. Johnson, "Frameworks = Components + Patterns ," *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, 1997.