Style-specific techniques to design product-line architectures

Philippe Lalanda
Thomson-CSF Corporate Research Laboratory
Phone: 33 1 69 33 92 90
Email: lalanda@thomson-lcr.fr
Domaine de Corbeville
91404 Orsay, France

Abstract: The product-line approach is tremendously reuse promoting and permits effective capitalisation on a given domain. It is based on the design and exploitation of a product-line architecture. Techniques to guide the design of such architecture are today needed in order to make the product-line approach applicable in industrial settings.

In this paper, we argue that these techniques should be style-specific in order to provide the necessary level of guidance for application engineers and we take the example of the shared repository style. The approach introduced hereafter is one part of the ESPRIT project PRAISE[1] (Product-line Realisation and Assessment in Industrial SEttings).

Keywords: product-line architecture, design

1 Introduction

The industry of software-intensive systems is facing today both economical and technical challenges. On one hand, shrinking budgets and sharp competition require to reduce significantly development and maintenance costs, shorten lead time, and improve quality and predictability. On the other hand, the dramatic increase of the size and complexity of systems have brought considerable problems in terms of suitability, efficiency, scalability or portability. Component-oriented programming is therefore becoming increasingly important in many companies. The key paradigm of this approach is megaprogramming [Boehm and Scherlis, 1992], that is the ability to define a system by putting together software components. This approach raises the level of abstraction for designers and programmers, allows to amortise development costs on several systems, reduces lead-time by decreasing the amount of code to be produced, and improves quality through reuse of well proven software components.

Component-oriented programming raises several important issues. In particular, we believe that unguided assembly of software components is unlikely to produce applications meeting a set of pre-defined requirements. An integration framework, that is a blueprint, is necessary to allow a coherent assembly of software components. Product-line architectures provide such frameworks: they are defined in the context of a product-line and are primarily designed in order to drive the production of new applications in the product-line. The development of new applications is intended to be quicker, cheaper and of better quality. It is based on the reuse of software components that have been developed in order to fit the product-line architecture.

[1] PRAISE is supported by the European Commission under ESPRIT contract 28651 and is pursued by Thomson-CSF/LCR France, Bosch Germany, Ericsson Sweden and the European Software Institute Spain.

However, the design of product-line architectures remains a complex and poorly guided activity. We believe that techniques for the design of product-line architectures are today necessary in order to make the product-line approach possible in industrial settings. In this paper, we argue that these techniques should be style-specific in order to provide the necessary level of guidance for application engineers. The approach we introduce hereafter is one part of the ESPRIT project PRAISE (Product-line Realisation and Assessment in Industrial SEttings).

2 Product-line architectures

A product-line architecture is an abstraction: it is a specification of the high level structures of a family of applications. These structures reveal complementary facets of an architecture (static structure, dynamic structure, etc…) and contain elements like components, connections, data, processes [Bass *et al*, 1998].

A product-line architecture has to meet three fundamental requirements:

- It has to drive the architectural design of new applications in the product-line
- It has to facilitate the reuse of components at the product-line level
- It has to permit various analyses in order to assess the impact (cost, performance, etc…) of specific requirements for the development of new applications in the product-line

In order to meet these requirements, a product-line architecture has to capture both architectural commonalities and variabilities of a family of applications. Commonalities and variabilities may concern any element of an architecture. For example:

- The components: their presence in an architecture, the provided services, the requested services, their implementation, …
- The topology: the existence of connections between components, the nature of connections, their implementation, …
- The dynamics: the existence of threads, the components and services that are called in a thread, the relative priorities of the threads, …
- The physical environment: the number of machines and of CPU, the types of machines, …

The purpose of specifying commonalities is to set architectural constraints and to enforce their meeting when deriving new applications. This brings enough stability to allow the definition of reusable software artefacts at the product-line level like workplans, development structures or cost analyses. Architectural elements specified as common are invariant: they cannot be changed during the adaptation of the product-line architecture for the delivery of a new application. A modification would be damageable for two reasons:

- It would break the product-line homogeneity at the architectural level
- It would prevent the newly specified architecture to benefit from the software artefacts available at the product-line level

The purpose of specifying variabilities is to pinpoint the places in the architecture that have to be adapted. Encapsulating variability leads application engineers to the important places in the architecture and prevents them from changing elements that must be preserved. Additional guidance to resolve variability may be provided with the architecture. This is especially important for some non functional properties like performance or distribution which are generally application-specific.

3 Designing a product-line architecture

When designing a product-line architecture, the purpose of a domain engineer is thus to specify common and variable architectural elements. The questions we try to answer are: what does this mean and how does he do ?

3.1 How to express commonalities

Specifying commonalities means expressing architectural invariant. The exact nature of this invariant actually depends on the degree of similarity between applications in a product-line.

Minimal specification can be obtained with an architectural style. A style is an abstract specification defining classes of architectures [Shaw and Garlan, 1996]. A style has no requirements about completeness and contains no domain information. It is however well suited to capture some commonalities in a product-line. For example, it can set the types of components and the topology that is allowable for the products architecture, the basic schemas of interaction between components and the minimum constraints that have to be met. Using a style to represent architectural commonalities clearly represents a minimalist approach: the stable part is not large enough to efficiently constrain application engineers. In addition, its level of genericity does not permit the definition of software artefacts at the product-line level.

In order to effectively constrain application engineers and to allow the definition of reusable software artefacts at the product-line level, we believe that commonalities should be expressed at a lower level and include domain information. More precisely, the invariant part of a product-line should contain the definition of:

- Standard software components and the way they are connected
- Standard execution modes
- Standard execution threads
- Standard mappings onto execution platforms

It is clear that the degree of standardisation is strongly dependent on the nature of the product-line. In product-lines where variability is important only a few of these elements may be specified.

3.2 How to express variabilities

Expressing variability in a product-line architecture means inserting information in order to pinpoint the places where adaptations are necessary and, if possible, provide some level of guidance to facilitate adaptations.

As indicated in [Perry, 1998], several techniques can be used in order to encapsulate variability in an architecture. The following are especially important:

- Under-specification
- Provisions
- Decision points

A way to express variability in a product-line architecture is to encapsulate architectural elements that vary and to leave them unspecified. The simplicity of this approach, that can be applied to any form of variability, is appealing. It clearly identifies the architectural places that need further refinement and thus provides some guidance for application engineers. It also leaves great flexibility to application engineers who are free to adapt the architecture as they wish. On the negative side, the amount of work for application engineers may still be important to fill in the unspecified parts. Application engineers

also have to conduct some analysis work in order to verify the conformance of their design/implementation with the requirements of the product-line.

Another way to express variability in a product-line architecture is to make provisions. This means that the architecture specifies more elements (components, services, modes, tasks, …) than necessary. At derivation time, application engineers have to select the adequate elements in order to meet the requirements of their application. This approach provides application engineers with all the elements needed to derive new architectures. The level of guidance is therefore advanced. However, the variability is not always clearly identified but rather diluted in the architecture. Choosing the right elements may then not be straightforward. In addition, constraints generally exist between provisioned elements: for example, some features may be incompatible, others may be complementary. These constraints have to be expressed in the product-line architecture in order to guide application engineers and to avoid non valid architectures.

Finally, another possibility is to encapsulate architectural elements that vary and to specify a list of possibilities or parameters. We call such a place a decision point. This approach is certainly the more advanced from the application engineers perspective. It pinpoints the architectural places where they have to intervene and provides a list of solutions to choose from. It also allows the definition of shareable software artefacts at the product-line level: the results and analyses provided by these artefacts can be explicitly expressed as a function of the decision points. As with provisions, constraints may also exist between decision points. Such constraints have to be expressed in order to guide application engineers.

4 Style-specific techniques

4.1 Our approach

In the previous section, we have presented a set of general techniques to capture commonalities and variabilities in a product-line architecture. Designing a product-line architecture requires to use all of these techniques. Depending on the characteristics of the product-line, techniques are preferable over others.

We believe that such techniques are too generic to provide effective guidance to domain engineers. A way to get more specific is to take into account architectural styles and to express what should be done to build a product-line architecture as a function of styles. It is today acknowledged that techniques or patterns to reach non functional properties like performance or adaptability in an architecture depend strongly on style, that is the types of allowed components, the way they communicate and cooperate, their execution model, etc. For example, techniques to increase performance of layered systems are different from those used for object-oriented systems. This is also true for the design of product-line architectures, the purpose of which is to reach high adaptability (which is definitively a non functional property).

As a consequence, our current goals are the following:

- Identify with our business units the domains that could benefit from a product-line approach
- Identify the architectural styles used in the selected domains and describe them as architectural patterns
- Provide style-specific techniques to guide the design of product-line architectures in the selected domains

4.2 Example

A very popular style that has been identified in our business units is the repository style [Shaw and Garlan, 1996]. In software systems based on the repository style, components communicate via a shared memory called a repository. The repository represents the only vector of communication for the system components. When a component produces some information that is of interest for other components, it stores it in the shared repository. The other components will retrieve it there if need be.

The repository style structures a software system into :

- Independent software components containing domain knowledge, called domain components. Components do not know each other. They are defined by their inputs and outputs, and have no knowledge of which components have produced the data they use, and which components will use their outputs.

- A structured repository accessible by every component (read and write accesses). This repository stores all the data that need to be exchanged by components during system execution.

- A control component which purpose is to activate and coordinate the domain components. It takes the form of a more or less sophisticated scheduler.

We have described this style as an architectural pattern [Lalanda, 1998]. We have also defined a set of actions to be performed in order to design a product-line architecture in domains where architectures are based on the repository style. For space consideration, we do not present all the actions to be performed (they are presented in detail in [PRAISE, 1999]).

Major actions are the following:

- Standardisation of the shared repository

- Definition of reusable domain components

- Definition of a standard control component

The standard repository should be defined as a structured memory (through the definition of data-structures) capable of storing any kind of data exchanged in the applications of the product-line. Most data should be common while others are specific to a few applications. Defining a standard repository is generally a tedious task since it may involve a large number of variables, often described heterogeneously across applications.

Standard domain components are related to the standard repository. The format of their inputs and outputs have to be in conformance with the data types defined in the repository. Defining standard components requires to specify a standard functional decomposition of applications. Since decomposition may be variable among a group of applications, choices have therefore to be made. Some domain components are specified as necessary while others are specified as optional (their purpose is to meet application-specific requirements as opposed to common requirements).

In our experiments, the control component is structured into prioritised tasks. Standardising the control component thus means standardising these tasks. Some are always present in an application; others are optional. In any case, they have to be defined in conformance with the standard domain components since their purpose is to activate and coordinate the activities of these standard domain components. Implementing a standard control component encapsulating variability in terms of tasks to be performed may be a difficult task. There is also no standard way to express some assumptions that are made in the tasks about the components services (about performance for example), and then to verify that these assumptions are met when integrating new domain components.

5 Conclusion

The product-line approach is tremendously reuse promoting and permits effective capitalisation on a given domain. It is based on the design and exploitation of a product-line architecture. Techniques to guide the design of such architecture are today needed in order to make the product-line approach applicable in industrial settings. The design of product-line architectures is especially hard for two reasons. First, the development of software architectures for large and complex systems raises challenges that are not well tackled today. This, of course, applies to product-line architectures. Second, product-line architectures are explicitly built to be reused. This requires to anticipate future evolutions and to provide application developers with means and guidance to adapt the architecture to their specific needs.

In this paper, we have been developing the idea that techniques to build product-line architectures should be style-specific in order to provide the necessary level of guidance to domain engineers. As part of the PRAISE project, we are currently defining techniques to build product-line architectures for several architectural styles, including the shared repository style.

6 References

[Boehm and Scherlis, 1992] B.W. Boehm and W.L. Scherlis, Megaprogramming, Proceedings of the DARPA Software Technology Conference, April 1992.

[Bass *et al*, 1998] L. bass, P. Clements and R. Kazman. Software Architecture in Practice. Addison-Wesley, 1998.

[Shaw and Garlan, 1996] D. Garlan and M. Shaw. Software Architecture: Perspectives on an emerging discipline. Prentice-Hall, 1996.

[Perry, 1998] D. Perry. Generic architecture descriptions for product-lines, Second International Workshop on Development and Evolution of Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, 1998.

[Lalanda, 1998] P. Lalanda. Shared repository pattern. Proceedings of Pattern Languages of Programming, Monticello, Illinois, 1998.

[PRAISE, 1999] Document P28651-D2.2, PRAISE project, http://www.esi.es/Projects/Reuse/Praise/