Component Based Software Engineering

Association With

Patterns, Software Architecture,

Application Frameworks & Product Line Approach

By

R.Rajesh.

Abstract:

Component-based development has many potential advantages such as shorter time to market and lower prices. These advantages are especially attractive for customers, who often do not recognize the risks of lower reliability, possible problems with maintenance, etc. Many software companies are forced to use imported components in their products, but are not able to keep the development process under control.

Component-based development is still a process with lot of problems, not well defined either from theoretical or practical points of view. At this moment a light on the associations of CBSE with other methodologies can give a clear picture as to where it is leading us. In this paper the association of CBSE with Patterns, Software Architecture, Application Frameworks & Product Line Approach is discussed in detail.

Patterns:

Patterns involve a general description of a recurring solution to a recurring problem. But, in fact, a pattern does more than just identify a solution; it also explains why the solution is needed. A pattern also does the following:

· It solves a problem: Patterns capture solutions not just abstract principles or strategies.
· It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.
· The solution is not obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from principles. The best patterns generate a solution to a problem indirectly -- a necessary approach for the most difficult problems of design.
· It describes a relationship: Patterns do not just describe modules, but describe deeper system structures and mechanisms.
· A pattern has a significant human component.

"When trying to document a software solution/design, different kinds of patterns at different levels of abstractions can be used. *Architectural patterns* express a fundamental structural organization or schema for software systems.

It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships among them. *Design patterns* provide a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context. An *Idiom* is a low-level pattern specific to a programming language.

An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language. "

The difference between these three kinds of patterns is in their corresponding levels of abstraction and detail. Architectural patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide-sweeping implications, which affect the overall skeletal structure and organization of a software system.

Design patterns are medium-scale tactics that flesh out some of the structure and behavior of entities and their relationships. They do not influence overall system structure, but instead define micro-architectures of subsystems and components. Idioms are paradigm-specific and language specific programming techniques that fill in low-level internal or external details of a component's structure or behavior.

Software Architecture:

CBSE and software architecture (SA) are separate but related topics in software engineering research and practice. CBSE focuses on the realization of systems through integration of pre-existing components, while SA is concerned with the high-level organization and structure of systems in general. The current high interest in SA is mainly motivated by the possibility of managing complex software by using components. The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architectural design can support reuse in several ways. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components (or subsystems) and also frameworks into which components can be integrated. Such reusable frameworks may be domain-specific software architectural styles, component integration standards and architectural design patterns. Having a large amount of reusable components is the precondition for using reuse technology effectively.

Firstly, reusable information is domain specific. Reusability is not an isolated property of information; it depends on a particular problem and problem-solving context. Therefore, domain-oriented strategy should be adopted when identifying, acquiring and representing reusable information.

Secondly, cohesiveness and stability of domains, viz. knowledge about solutions and solution methods in domains are sufficiently cohesive and stable.

From an architectural perspective component-based system such as DCOM, CORBA, and JavaBeans define architectural styles that are predominantly object-oriented. In addition, they may support other forms of interaction such as event publish-subscribe.

Application Framework:

The purpose of an application framework is to enforce good design practices across all of the layers of a multi-tiered architecture. A good application framework is based on best practice architectural patterns, design patterns, re-usable components and libraries. A framework is more than just a pattern repository because it prescribes the interactions of the application at all layers and specifically, how it addresses the business problem.

Product Line Approach:

Product line approach can be simply defined as Domain Specific Engineering. It can be defined as a framework and discipline for engineering and manufacture of similar products. Domain is nothing but a set of processes, tools and assets of similar kind. Product line approach suggests to create a domain i.e. set of process, tools and assets for building similar kind of products. Always should institute and improve a product line business. Concentrate more on giving tailored products to the customers.

This approach is exclusively focusing on a market with customers who have similar needs. It is developing a product family and process for building similar customized products rapidly.

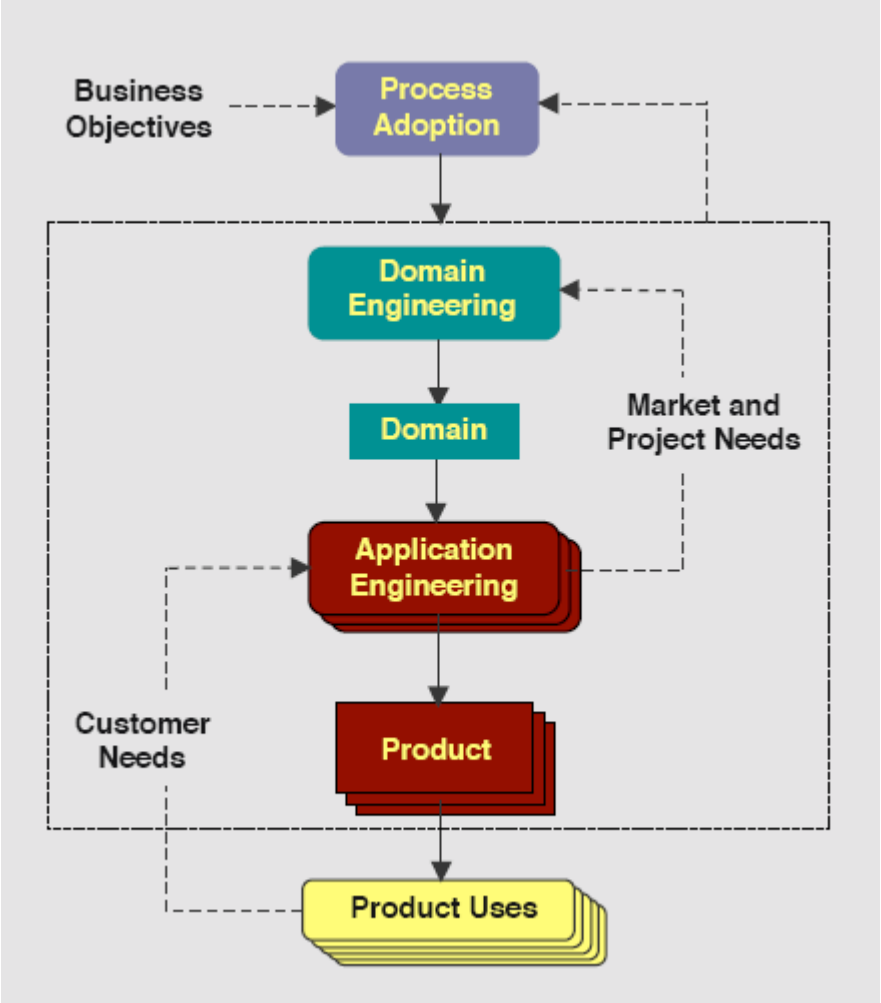

Chart depicting the Product Line Approach.

The Product Line Business Motivations are:

- Gain competitive advantage by being more responsive to diversity and change in customer and market needs.
- Increase "process capability" (achievable productivity and product quality) by focusing improvement efforts on a set of similar products

The logical association between Product line approach and CBSE can be effectively embossed at *reuse*. CBSE mostly concentrates on component reuse which is also the back bone for product line approach. So the components are usually part of product line approach.

The notion of software reuse has been around for a time and got broad application, such as subroutines reuse, generic classes reuse, and compilers reuse etc. The introduction of the notion of software component provides the technical foundation for software reuse. Consequently makes software reuse gain more universal attention. The component production and reuse is the key for software industry to develop and form scale-economics.

In general, component is the system element that can be distinguished definitely; software component is the relatively independent constitutional element that has certain contents in software systems. Because the software component being talked about at present mainly concerns about its reuse content, software component mostly denotes reusable software component, viz. system constitutional element that has relatively independent functions and can be reused by a number of software systems. In the following discussion, except for special indication, the component we talk about all denotes reusable software component.

Having a large amount of reusable components is the precondition for using reuse technology effectively. Firstly, reusable information is domain specific. Reusability is not an isolated property of information; it depends on a particular problem and problem-solving context. Therefore, domain-oriented strategy should be adopted when identifying, acquiring and representing reusable information.

Secondly, cohesiveness and stability of domains, viz. knowledge about solutions and solution methods in domains are sufficiently cohesive and stable. So cohesiveness of specification and implementation knowledge in a domain makes it possible to solve a large number of problems through a finite, relatively small set of reusable information. Stability of domain makes it possible to reuse the captured information again and again over extended periods of time.

From the knowledge of these two aspects we can draw the conclusion: the components acquired in domain engineering are large numbers and have high reusability. More than that, the domain-oriented software architecture acquired in domain engineering is more specific and easier to manipulate than the general software architecture. Therefore, domain engineering is the main approach to acquire component and architecture.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. Architectural design can support reuse in several ways. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components (or subsystems) and also frameworks into which components can be integrated. Such reusable frameworks may be domain-specific software architectural styles, component integration standards and architectural design patterns.

As observed, an important trend has been the desire to exploit commonality across multiple products. Two specific manifestations of that trend are improvements in our ability to create product lines within an organization and the emergence of domain-specific architectural standards for cross-vendor integration. With respect to product lines, a key challenge is that a product line approach requires different methods of development. In a single-product approach the architecture must be evaluated with respect to the requirements of that product alone. Moreover, single products can be built independently, each with a different architecture. In particular, there must be an up-front (and on-going) investment in developing a reusable architecture that can be instantiated for each product. Like product line approaches, domain-specific architectural standards for cross-vendor integration provide frameworks that permit system developers to configure a wide variety of specific systems by instantiating that framework. But more importantly, such standards support the integration of parts provided by multiple vendors.

From an architectural perspective component-based system such as DCOM, CORBA, and JavaBeans define architectural styles that are predominantly object-oriented. In addition, they may support other forms of interaction such as event publish-subscribe.

Having a large amount of reusable components is the precondition for using reuse technology effectively. Firstly, reusable information is domain specific. Reusability is not an isolated property of information; it depends on a particular problem and problem-solving context. Therefore, domain-oriented strategy should be adopted when identifying, acquiring and representing reusable information. Secondly, cohesiveness and stability of domains, viz. knowledge about solutions and solution methods in domains are sufficiently cohesive and stable. So cohesiveness of specification and implementation knowledge in a domain makes it possible to solve a large number of problems through a finite, relatively small set of reusable information. Stability of domain makes it possible to reuse the captured information again and again over extended periods of time.

From the knowledge of these two aspects we can draw the conclusion that the components acquired in domain engineering are large numbers and have high reusability. More than that, the domain-oriented software architecture acquired in domain engineering is more specific and easier to manipulate than the general software architecture. Therefore, domain engineering is the main approach to acquire component and architecture.

However, component integration standards typically go beyond architectural modeling by providing run-time infrastructure and (in many cases) considerable support for generating code from more abstract descriptions.

Conclusion:

As we have seen, though the concepts of Patterns, Software Architecture, Application Framework and Product Line approach are separate and important in their own way yet are tailored into each other from a Component Based Software Engineering point of view. All these four components plus Component Based Software Engineering are independent yet inseparable and give the best of software engineering when activated and implemented together.

References:

1. Overview Component-based Software Engineering State of the Art Report. Ivica Crnkovic, Magnus Larsson.

2. Component-Based Software Engineering: New approach in managing large-scale software Seminar Hörsalen ABB Corporate Research Kopparlunden.

3. Viewpoints and Frameworks in Component-Based Software Design P.S.C. Alencar, D.D. Cowan, T. Nelson

4. Different aspects of Component Based systems. Magnus Larsson.

5. On the Definition of Concepts in Component Based Software Development Zeynep Kiziltan, Torsten Jonsson, and Brahim Hnich

6. Architectural Styles in Component-Based Software Engineering.
   Frank Lüders

7. Reuse-Based Software Production Technology Yang Fuqing, Wang Qianxiang, Mei Hong, Chen Zhaoliang

8. Software Architecture. David Garlan

9. Sotware Architecture and Product Lines. Pearsons Education.