Component Based Software Engineering using Innovative Patterns

[1]Pranayanath Reddy Anantula, [2]Raghuram Chamarthi (Tejoraghuram)
[1]Dept. of IT, CVR College of Engineering, Ibrahimpatnam, Hyderabad, AP, India
IT Analyst, TATA Consultancy Services, Hyderabad, AP, India

Abstract
In today's world of rapidly advancing technology, for any kind of software product, quality is the prime concern. A software product is said to be a quality product if it justifies all the customer requirements. The quality of software can be categorized broadly into two ways, one quality of the design and the second quality of conformance. Quality of design encompasses requirements specification and design of the system and quality of conformance focuses on implementation. The quality of software is intricately connected to the underlying architecture because architecture is the base for further development of the project (Analysis, design and implementation). The work products of architecture refinement are the high level design, low level design, implementation and test cases. The software architecture is the structure of the system which comprises software components, their externally visible properties and the relationship among them. The software architecture is a meta-structure created based on the requirement and specifications. Architecture aids the software engineers to make early design decisions so that it will have a profound impact on software engineering activities that involve in success of the system. The major work to be done in modeling the architecture is to identify qualified, adaptable, updatable components and how these components are associated to build a new product. In the current work, we provide innovative patterns which help in developing the product using component based software engineering. Here we discuss how to create components by applying various innovative patterns like subtraction, multiplication, division, task unification and attribute dependency change patterns .We will also depict the results of applying these patterns to some of the software projects.

Keywords
Component Based Software Engineering, Innovative Patterns, Quality Attributes, and Software Architecture.

I. Introduction

Software architecture of a project is the structure of the system that encompasses software components, interfaces and relationships or interactions among them. It contains the global structure or a blue print of entire system. The architecture enables the software engineer to analyze the effectiveness of design to meet the requirements, consider alternative architectures, and reduce the risk associated with the construction of software. The principle of architecture evaluation of the system is to assure the quality. Software architecture must model the structure of a system in a way that the data and procedural components collaborate with one another. An architectural pattern is used in identifying the components required to be created from scratch or reuse from the repository of available components. The architecture pattern helps the software engineer to focus on individual aspect of the architecture rather than the entire architecture; patterns impose rules on the architectures via how the software should handle some of the aspects of functionality at the infrastructure level, address specific behavior issues within the context of the architecture.

A. Refine Architecture into Component

A component is a modular building block for the complete software. It is also a deployable and replaceable part of the system that encompasses implementation and exposes a set of interfaces. As software architecture is refined into software components the structure of the system begins to emerge. The components of the software architecture are derived from three sources they are application domain, infrastructure domain and interface domain. As we know that components are part of the software architecture, they must communicate and collaborate with other components and with entities that exists outside the boundaries of the software. The bond among the components is measured using cohesion and coupling factors. Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependency among modules [2]. For a good quality software project components should have high cohesion and low coupling factors. Each individual component in the software architecture should exhibit some of the properties like modularity, information hiding, functional independence, refinement, flexibility [11, 16, 18]. Component should exhibit a single or primitive solution and it should not depend on other components. It should also follow OCP open closed principle (A module/component should be open for extension but closed for modification).

Most of the software engineering efforts are spent in developing Component Based Software Engineering (CBSE). The component based software engineering includes identifying the right kind of components , their data structure, what all functionality should be present in the component, how they adapted the requirements, how to collaborate/composition of components and update components based of the feedback or future requirements [10].

Based on the requirements the software engineers will decide from the following questions which one to be chosen for each specific requirement.

1. Are Commercial Off The Shelf (COTS) components available?
2. Are in-house reusable components available to implement the requirement or not?
3. Whether the interfaces that are available are compatible within the architecture of the system to be built?

For reusable components the following process is followed [10, 13-14] as shown in the fig. 1.

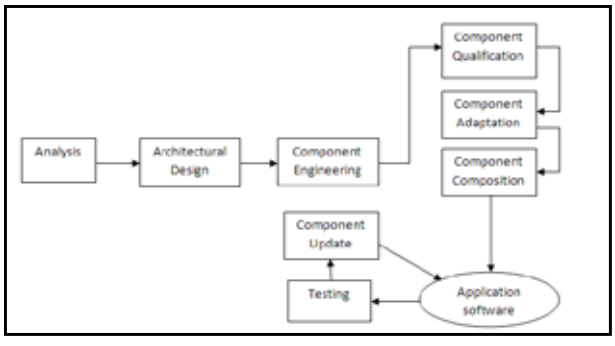

Fig. 1: A Process Model that Uses CBSE

1. Qualified components
It is assessed by the software engineers to ensure that not only functionality but performance, reliability, usability, testability and other quality factors conform to the requirements of the system to be built.

2. Adapted components
adapted to modify (also called mask or wrap) unwanted or undesirable characteristics.

3. Assembled components
integrated into an architectural style/pattern and interconnected with an appropriate infrastructure domain that allows the components to be collaborated, coordinated and managed effectively.

4. Updated components
replacing existing software as new versions of components become available [1].

B. Impact of CBSE on Quality, Productivity and Cost

1. Quality
In an ideal setting, a software component that is developed for reuse would be verified to be correct and would contain no defects. In practical, formal verification is not carried out routinely, and defects can and do occur. However, with each reuse, defects are uncovered and fixed, and a component's quality improves as a result. Over time, the component will become virtually defect free. It is fair to state that reuse provides a nontrivial benefit in terms of the quality and reliability for delivered software.

2. Productivity
When reusable components are applied throughout the software process, less time is spent creating the plans, models, documents, code, and data that are required to create a deliverable system. It follows that the same level of functionality is delivered to the customer with less input effort. Hence, productivity is improved. Although percentage productivity improvement reports are notoriously difficult to interpret, it appears that 40 to 50 percent reuse can result in productivity improvements in the 35–40 percent range [1].

3. Cost
The net cost savings for reuse are estimated by projecting the cost of the project if it were developed from scratch, and then subtracting the sum of the costs associated with reuse, and the actual cost of the software as delivered. The costs associated with reuse, include [1].

1. Domain analysis and modeling.
2. Domain architecture development.
3. Increased documentation to facilitate reuse.
4. Support and enhancement of reuse components.
5. Royalties and licenses for externally acquired Components.
6. Creation or use and operation of a reuse Repository.
7. Training personnel in design and construction for reuse.

Although costs associated with domain analysis and the operation of a reuse repository can be substantial, many of the other costs noted here address issues that are part of good software engineering practice, whether or not reuse is a priority.

II. Innovative Patterns
Here we discuss the five innovative patterns that have emerged from historical analysis of product development trends. Here we explain each pattern with the help of example and how each pattern can be applied to a software project. We have found that if these patterns are applied we can predict the market trend in advance and make changes accordingly.

A. Subtraction pattern
When a new product is introduced into market, the product tends to overcome the complexity of the old version product and their by adding some more interactive and innovative features that would enhance the performance and also user satisfaction.

As software project keep on updating, based on the market trends we need to modify the architecture design i.e., components of the system to meet the new requirements and in this process we need to develop the components such a way that little effort should be applied to modify or enhance the components. To do this software engineers must predict the changes in advance and create abstract operations which may be refined when needed instead of adding them when requirement occurs.

1. Example
Online web application that sell laptops through online. Users found it difficult to check the required features, price list and compare the model with other models. Due to this they were used to spend lot of time in navigating through different pages. These demerits were analyzed and came up with a web page where user can select the specifications, the range of price and also can compare the price and specifications with other modules in a single web page.

B. Multiplication pattern
In this pattern another copy of existing feature is made without changing the existing features of the components [5]. The objective of this pattern is to achieve quality change by adding extra add-on features to the existing one. When creating a component software engineer should analyze the features completely and come up with alternative strategies to achieve same functionality but with higher quality.

1. Example
In Online banking system if the user want to know the transaction details user need to give from date and to date to see all the transactions done in between the specified dates. And if user is interested to check the details of the specific transaction based on the amount he need to go through all the transactions to identify. To overcome this, an advance feature is provided which does the same functionality but it cut down the search criteria by asking the user to give the amount in a range. So that software will display only those transactions that falls under the given category.

C. Division pattern
Division pattern is used to split the existing product into many component modules. The specialty of this pattern is that each module preserves the characteristics of the whole product. Each component should be able to operate individually even other components are not operating properly i.e., the component preserves the functionality with in it and there is no dependency with other components [4].

1. Example

Google known for its wide range of usage all over the globe had all the utilities integrated into a single domain. Some of the utilities were maps, videos, igoogle, search engine, mail, images etc. Now they have been separated into individual modules. The main advantage of this pattern is that even if the "google.com" is down, the user can still browse the various areas using their individual modules.

D. Task Unification Pattern

Assign a new task to an existing product, thereby fusing two tasks in a single task. The basic rationale is to binding the related features into a single component instead of two separate components [4]. If two components are having most of the similar attributes it is better to combine the two components and make single component. The use of this pattern is that if any modification is required, it is sufficient to do in one component. In the earlier case where the same kind of functionality presents in two components we need to analyze and do the modification in the relevant component or do the same kind of modification twice.

1. Example

The video cutter player which is used for playing the video or audio and also editing the media file. In this player they have merged the media player and video cutter features into one product their by providing a greater customer satisfaction and ease of use.

E. Attribute dependency change Pattern

This pattern mainly involves dependent relationship between attributes of a product and attributes of its immediate environment. Pattern can be made more adaptable to the given environment. One can also create dependencies that exist between two unrelated attributes of a single pattern [5-6]. The attribute dependency pattern often generates what later seem like inevitable patterns.

1. Example

Windows Media Player is mainly used for playing audio and video files. This media player identifies the file format and plays the file accordingly. That is if the extension is .mp3 then it identifies that it is a audio file and plays the file in the audio format. If the extension is avi then it identifies that it is a video file and plays it in the video format. In this way it adapts to the given environment and satisfies the needs of the customer. Applying innovative patterns to component based software engineering:

Step 1: Collect scenarios and analyze them from user point of view.

Step 2: Elicit requirements, constraints and environment description.

Step 3: Describe the architecture style/pattern that has been chosen to address the scenarios and requirements.

Step 4: Identify the quality attributes relevant to the architecture.

Step 5: Apply the innovative patterns one by one iteratively.

Step 6: Refine the architecture based on innovative patterns.

Step 7: Evaluate quality attributes by considering each attribute in isolation attributes assess includes reliability, performance, security, maintainability, flexibility, testability, portability, reusability and interoperability.

Step 8: Generate the result and submit to customer for feedback.

Step 9: Repeat step 4 to 7 if required based on the customer feedback.

III. Results and Discussion

Some of the quality attributes are evaluated and compared with the traditional approach and innovative pattern approach.

Let us look at three unique and distinct projects, namely:

1. Financial management with 5000 functional points
2. Health care with 4000 functional points
3. Online reservation system with 2000 functional points

The result of the comparison is presented below in detail with the following parameters.

The graph plotted below shows the relationship between Reliability, Maintainability, Adaptability, and Testability (in terms of percentage on Y axis) and function points (X axis) and presents a comparison between the traditional approach and innovative pattern approach.

A. Reliability

Reliability is defined as the amount of time that the software is available for use. It is also defined as the probability that it will perform a required functionality under some conditions for a period of time. The probability of success, R (t), plus the probability of failure, F (t), is always unity. Expressed as a formula:

F (t) + R (t) = 1 or, F (t) =1 - R (t).

It was seen that the reliability of the system was higher in the pattern approach than in traditional approach. Statistic report is as shown in the fig. 2.

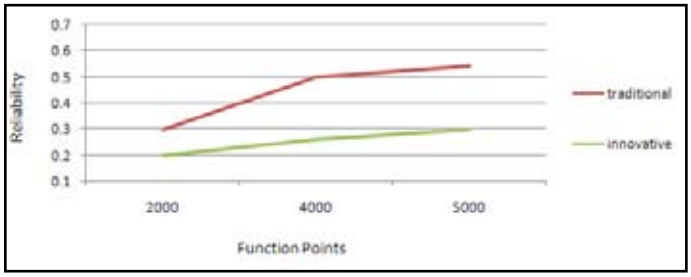

Fig. 2: Graph Showing Variation In Applying Traditional And Innovative Pattern For Reliability

B. Maintainability

Maintainability is defined as a percentage of total cost that requires maintaining the product. Learning from the past in order to improve the ability to maintain systems, or improve reliability of systems based on maintenance experience. It was seen that the maintainability of the system using patterns was easier and significant when compare to traditional approach. This was because of the application of subtraction, division and task unification patterns. Statistic report is as shown in the fig. 3.

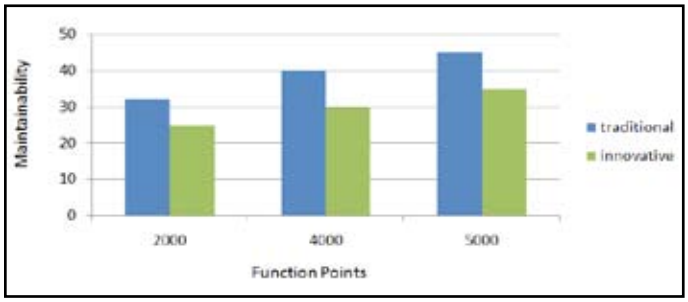

Fig. 3: Graph Showing Variation In Applying Traditional And Innovative Pattern For Maintainability

C. Adaptability

It is the quality feature provided by the system. Adaptability represents the degree of ease with which modifications can be accommodated. The tasks will be distinct, the end-users will be heterogeneous, operational environment will change and their competences and expectations will evolve. Here again it is impossible for developers to anticipate all possible requirements modifications. Thus, the dynamics of changing conditions shifts the customization process of the system's characteristics from the development phase to its usage and operation phase because the time needed for a professional development is too short or the new features are too costly. Due to subtraction and task unification patterns adaptability is much higher when compared to traditional approach. Statistic report is as shown in the fig. 4.

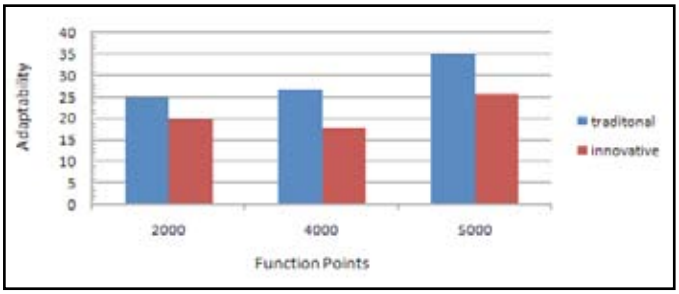

Fig. 4: Graph Showing Variation In Applying Traditional And Innovative Pattern For Adaptability

D. Testability

Testing a program to ensure that it perform its intended function [7]. Assuming that good test coverage is applied and most of the defects are uncovered and fixed before the product is released. This ensures that customers will report minimum number of defects. The chance of achieving customer satisfaction with such kind of products is much higher. If a product is having probability of lesser defects, than a lot of money is spent on maintaining and supporting a product after its development will be less. Hence testability is one of the important attribute to the maintainability of any software product. It was observed that the project that was created using innovative pattern lead to higher testability. Statistic report is as shown in the fig. 5.

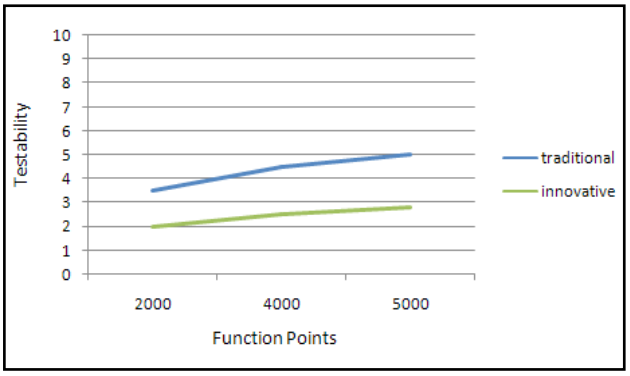

Fig. 5: Graph Showing Variation In Applying Traditional And Innovative Pattern For Testability

From the above evaluation of software quality attributes we observed how the application of innovative patterns in component based software engineering brought a variation in their values and this resulted in a significant improvement in performance.

IV. Conclusion

The objective of applying innovative patterns to component based software engineering is to assure high quality software product to customer. The collaboration of components is essential to expose the specific limitations of the architecture and user requirements. An exhaustive literature survey was take-up to derive how innovative patterns have an edge over traditional approach and it was proved that applying innovative patterns improve the quality of software in terms of making architecture simple and by predicting future enhancements/modifications in advance. Each innovative pattern which we have discussed has some distinct feature like the subtraction pattern is used to eliminate the redundant and useless features and add new features. The multiplication pattern is used by making a similar copy of a feature with some more advance enhancements. The division pattern is used to split the complex component into simpler components. The task unification pattern merges the distinct tasks that can be served as a single component and finally the attribute dependency change. Using these patterns we added finer granularity to the component based software engineering approach. All these patterns were applied to the some of the software projects and it was observed that there was a significant improvement in the design and the quality attributes of the software project. These patterns can be applied to any kind of software or hardware product universally.

References

[1] Roger S Pressman,"Software Engineering A Practitioner's Approach 6th Edition", Mc Graw Hill, 2010.

[2] Bachmann F.Bass L, Klein M, Shelton C.,"Designing software Architectures to achieve quality attribute requirements", IEEE Proceedings 2005, pp. 53-165.

[3] Bass Len, Paul Clements, Rick Kazman,"Software Architecture in Practice", Second Edition, Addison-Wesley, Boston, 2003.

[4] Jacob Goldenberg, Yoram Louzoun, Sortin Solomon, David Mazursky,"Finding your innovation sweet spot", Harvard Bussiness Review, 2003, pp. 1-11.

[5] Sankar Ram N., Paul Rodrigues,"Innovative patterns for finding enhanced solution to your architecture", International Journal of Computer Science and Network Security, South Korea, Vol, 8, No. 7, pp. 314- 318, 2008.

[6] Sankar Ram N., Paul Rodrigues, "Architectural patterns for finding your innovative sweet spot", Journal of Convergence Information Technology, Korea, Vol. 3, No. 4, pp. 54-58, 2008.

[7] I. Sommerville.,"Software Engineering (6th Edition). Addison-Wesley, 2001.

[8] R. Pooley, P. Steven.,"Using UML: Software Engineering with Objects and Component", Addison-Wesley, 1999.

[9] Adler, R.M.,"Emerging Standards for Component Software", Computer, vol. 28, no. 3, 1995, pp 68–77.

[10] Brown, A.W., K.C. Wallnau,"Engineering of Component Based Systems", Component-Based Software Engineering, IEEE Computer Society Press, 1996, pp. 7–15.

[11] Clements, P.C., "From Subroutines to Subsystems: Component Based Software Development", American Programmer, vol. 8, No. 11, 1995.

[12] Gogru, A., M. Tanik,"A process Model for Component-Oriented Software engineering", IEEE software Engineering, Vol .20, No. 2, 2003, pp. 34-41.

[13] Whittle, B.,"Models and Languages for Component Description and Reuse", ACM Software Engineering Notes, Vol. 20, No. 2, 1995, pp. 76–89.
[14] Tracz, W.,"Third International Conference on Software Reuse-Summary", ACM Software Engineering Notes, vol. 20, No. 2, 1995, pp. 21–22.
[15] Linthicum, D.S.,"Component Development (a Special Feature)", Application Development Trends, 1995, pp. 57–78.
[16] Heineman G., W. Councill,"Component Based Software engineering", Addision-Wesely, 2001.
[17] N. Haghpanah, S. Moaven, J. Habibi, M. Kargar, S. H. Yeganeh, "Approximation algorithms for software component selection problem", In APSEC, pp. 159–166, IEEE Computer Society, 2007.
[18] Arvinder Kaur, Kulvinder Singh Mann,"Component Selection for Component based Software Engineering", International Journal of Computer Applications, Vol. 2, No.1, May 2010.