THE EVOLUTION OF COMPONENT BASED SOFTWARE ENGINEERING FROM THE TRADITIONAL APPROACH TO CURRENT PRACTICES

U. A. Khan
Department of IS, FCIT, King Abdul Aziz University, Jeddah, Kingdom of Saudi Arabia

ABSTRACT
Reusability is a phenomenon that has been long known. Initially in a low scale, at the levels of functions in the programs, and later in its alter avatar, in form of Component Based Software Engineering (CBSE). Reusability was the key factor around which CBSE revolved. But later, it was observed that even a more stringent factor was involved that marked CBSE as a pioneer in the Software Development Methodologies - Evolution of Requirements. Evolution of requirements is a natural phenomenon, in the life cycles of the Software Systems in the modern era. As requirements evolve, new features are added and existing features are modified or deleted. To conform to the requirement evolution, the current working version of the implementation also has to change in most cases, to support the changed needs. Where other methodologies felt short to embrace, CBSE seems to harness this fact. In this paper, I have compared the various approaches and methodologies from the traditional times. I then survey some of the popular approaches to CBSE, and provided a comparative discussion among those approaches considering the challenges faced by component based development.

I. INTRODUCTION
The development of Software Systems has undergone a radical change in the past few decades. In the past times, where traditional approaches were used, the Software was moderate in size as well as complexity. But, as the time progressed, it all changed. In the current scenario, the software is large in size, and the complexity has been drastically enhanced. The requirements and the functionality of the software evolve in a hurry. The traditional approaches have all but failed to deliver the expectations of the deadlines, budgets and quality for such large and complex software systems. Over the past some time, researchers have made it an aspiration to develop means of creating Software in a rapid, efficient and cost effective fashion. In their ambition to excel, the researchers re-defined the perception of the terms “Software Re-usability”, to the extent where the need for developing software units from the scratch was totally eliminated. The birth of the Component Based Software Engineering gave new directions to meet the challenges posed by the ever evolving Requirements in a Client Centric World. A whole new world of reusable components was aiding the developers in achieving their goals in a faster, more efficient and cost effective manners. It became possible for the developers to engineer new products with the existing ones with little effort and difficulty. The improvement in quality, increase in productivity, efficiency in the management of complex software, and wide variety of reusability [1] attributed to the huge success of Component based Software Engineering.

Despite the years and importance, Component Based Software Development (CBSD) is still young concept in the area of Software Engineering. There have been studies relating to the life cycle for CBSD conducted over a period of time, and some methodologies have been developed and employed in the industry. In this paper, I have reviewed the life cycle of the products being developed by Component Based Software Development approach, and try to provide a comparative study on the several popular lifecycles, discussing on their approaches and the differences that have been addressed by these approaches accordingly.
1. Comparison of the Data Centric vs the Procedural Approach for Creating Software’s

In general, most of the Application Software’s developed from the very beginning have been Client Server perspective. So it becomes worthwhile to compare the two popular software development life-cycle for developing current and future Client Server Software’s:

- Procedural centric models offer poor encapsulation / abstraction:
  - operations / procedural interfaces don't include testing and responses to requests
  - don't encapsulate data they operate on
  - some data operations have no procedural interface
- Procedures don't define who is responsible for data they use: who creates, who mutates, who queries who destroys? This make it difficult to re-use code
- structural models are not bound to operational model
- many low-level features of use of procedural interface are complex and are not factored out and abstracted in models, hence, the use of a client-server procedural API is synonymous with bottom-up development
- Data centric models offer a high-level of abstraction and encapsulation:
  - operations in model include testing and the responses to a request
  - encapsulate data they operate on
  - data can only be accessed via operations
- models define who is responsible for data they use - who creates, who mutates, who queries, who destroys
- Structural models are bound to operational model.
- Low-level generic features of the implementation are abstracted in the design models.

II. REVIEW OF SOME POPULAR SOFTWARE DEVELOPMENT METHODOLOGIES

With time, the complexity of the software increased exponentially. The effort to develop software's was also required to be compounded, and so, on this account the methodologies have to be developed too. The Software Development Life Cycles have to evolve in a fashion, so as to support the ever increasing complexity of the Software.

It is thus appropriate to generally examine different software life cycle models and point out their strengths and weaknesses before an alternative one is put forward.

The Waterfall model [2] has been long used by software engineers and has become the most prevalent software life cycle model. This model as shown in Figure 1, initially attempts to identify phases within software development as a linear series of actions, each of which must be completed before the next is commenced.

![Figure 1: Waterfall Model](image)

The Waterfall model is marked by the apparently neat, concise and logical ordering of the series of obvious phases, which must be followed in order to obtain the final software product. Refinements to this model consider that completion is seldom and that iteration back to a previous stage is likely to happen, but it takes no account of bottom-up development and prototyping.

The Spiral model [3] makes software development more flexible and has been proposed mainly to speed up software development through prototyping. As shown in Figure 2, Prototyping is the process of building an incomplete piece of software that exhibits some of the most relevant aspects of the final software system. Prototyping provides constructive feedback to designers and potential users so that the system requirements can be clarified and refined early during software development. Evolutionary prototypes provide incremental software development, so that software systems may be gradually developed and tested, allowing major errors to be exposed and corrected early, which means that they are often cheaper to fix, but without effective management to control iterations, this process can degenerate into uncontrollable hacking.
Extreme Programming [4] is a deliberate and disciplined approach to software development, which is aimed to solve customer requirement change problem.

The methodology is designed to deliver the software your customer needs when it is needed. It is based on four essential values: communication, simplicity, feedback and courage. Unlike other traditional software development models, such as Waterfall, which conducts analysis, design, implementation and testing once in its long development cycle, or Spiral model, which has shorter, iterative development cycles, extreme programming is blending all these activities, a little at a time, throughout the entire software development life process. Figure 3, displays the four circles of Extreme Programming.

A growing number of companies in the software industry – including Microsoft – are following a process that iterates among design, building components, testing and getting feedback from customers as the product evolves. Many companies also ship preliminary versions of their products, incrementally adding features of functionality over time in various product releases [5]. Microsoft is different in the sense that it has introduced a structured hacker like approach in the development of large-scale software products; projects remain under control because teams of designers and testers frequently integrate and stabilize their improvements. It is truly an example of how to take advantage of the exploding demand for PC software and an effective way to deliver products to a “hungry” market. Its competitive strategy revolves around identifying mass markets quickly, introducing a product that is good enough to dominate the market rather than waiting until the product is perfect, then upgrading the product continuously.

The Twin Peaks model [6] also argues for a concurrent, iterative development of requirements and architecture during software development. It presents a partial and simplified version of the Spiral model that illustrates the distinct, yet intertwined activities of requirements engineering and architectural design. This model allows incremental development with the consequent management of risks, compromises architectural choices to accommodate existing commercial off-the-shelf software (COTS) solutions and because the model focuses on finer-grain development, it is more receptive to rapid changes as they occur. However, one of the main shortcomings of all these models is that none of them explicitly encourages reusability along all their phases. Therefore, a software life cycle model that emphasizes the importance of component reuse during software development is still in demand. Figure 4 displays a sample Twin Peak Model.

III. COMPONENT BASED SOFTWARE ENGINEERING

The basic unit of CBSE is a component. One widely accepted definition, due to Szyperski[7], is:
“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

According to a survey [8], a software component model defines what components are (Semantics), how components are defined, constructed, and represented (Syntax), and how components are composed, assembled, or deployed (Composition). A wide variety of different component models have been proposed and developed in the software engineering domain.

Two concepts are central to component-oriented development: the component; and, separately, the component model. There are various interpretations of these concepts; cf. e.g. [7] or [12]. The acceptation given to those concepts, and their realization in practice, largely influence the goodness of the approach, the fulfillment of all the application needs of interest, and ultimately, the opportunities for software reuse.

Using component-based software engineering (CBSE), applications are composed of reusable components with well defined interfaces and behavior. It has become a commonly used development paradigm. Using CBSE, software systems are built by composition of reusable building blocks called components. In addition to the services provided by a component, also the services required from other components or the environment are specified.

Figure 5: CBSD [13]

Broadly speaking, there are two types of component-based reuse: 1) The component based software systems that are built with assembling components already developed and prepared for the integration with no change, and 2) Using components already developed with changes to customize the need of the client. Reuse without change means simply selecting a component from a software component database, and dropping it into new software being developed. The cost of developing the component anew is zero! There are, in general, three types of component-based reuse in wide use.

One is reuse of most of existing software when developing the next version of the software. Typically, some 60-80 percent of the existing software gets to be reused in this situation. Another is “reuse” of thirty-party software, such as a sorting package, a database loader, etc. on the market or on the Internet as open source code. A third type of reuse is common functions available in programming language libraries, such as the math functions in the C Programming Library.

IV. COMPONENT BASED SOFTWARE ENGINEERING LIFE CYCLE
The development cycle of a component-based system is different from the traditional ones, i.e. the waterfall, spiral and prototype based models. Figure 6 shows a comparison between the traditional waterfall model and the component-based development model. The different steps in the Component Development Process are:

1. Find components which may be used in the system:
2. Select the components which meet the requirements of the system.
3. Alternatively, create a proprietary component to be used in the system.
4. Compose and deploy the components using a framework for components.
5. Replace earlier with later versions of components.

Somerville has provided a sequential approach for CBSD in [17] as displayed in Figure 7. There are mainly six steps which include:

Step 1: The user requirements are developed in outline rather than in detail as specific requirements limit the number of components that might be used.
Step 2: A complete outlined set of requirements are used to identify as many components as possible for reuse.
Step 3: Requirements are refined and modified so that they can comply with components.
Step 4: Architectural design is developed.
Step 5: After system architecture is designed, steps 2 and 3 may be repeated.
Step 6: Finally the components are integrated which turns into a complete system.

V. ISSUE FOR CBSD LIFECYCLE

Like any Software Development Life Cycle, there are issues related to CBSD Lifecycle as well. Some of these challenges are listed as below: [14, 15, 16]:

- Component behavior: System testing and maintenance is difficult for CBSD because:
  - Diverse tracking methods
  - No built-in tracking mechanism for third party components
  - No configuration function for clients.
- Difficulty in component error isolation: Trace formats and tracking mechanisms are different in components as different developers are involved.
- It causes inconsistency in tracking mechanism and trace massages which results in difficulty in error detection and isolation of components.
- Costly performance testing and tuning: Component’s performance information is usually not available to testers. So, system testers and integration engineers spend a lot of effort in this to trace out performance problems and responsible components.

VI. A COMPARISON OF CBSD MODELS

Several Models have been developed for assuring the efficient Software Development using Component Based Software Development approach. A brief discussion of some of them is given below. The V Model is an adaptation of the traditional waterfall model for modular system development. It defines a sequential process consisting of phases for requirements, system specification, system or architectural design, module design, implementation and testing. Implementation consists of coding for the individual modules, and coding for integrating the modules into the entire system using the architectural design for the system. An adaptation of the V Model
for CBD that does incorporate the bottom up nature of CBD is that of [18].

![Figure 8. Adaptation of the V Model for CBSD](image)

However, this adaptation really applies the V Model only to its system life cycle; there is no evidence of the V Model in its component life cycle.

Another model by the name of W Model has been used in software testing [19] and product line engineering [20] in the context of traditional (i.e. non-CBD) software engineering. [19] extends the V Model by adding a branch that integrates testing with debugging and code changes. [20] applies the V Model to domain engineering and application engineering in product lines.

In the context of CBD, the W Model is similar to standard CBD processes, e.g. [21], [22], [17], [23], in that they both contain separate life cycles for components and systems. However, unlike these processes, its component life cycle is the idealized one, which meets all the CBD desiderata in the literature. In particular the idealized component life cycle defines component composition in both component design and component deployment phases. This emphasis on composition results in compositional reasoning, which is an important property that is beneficial for practical system development, since it enables hierarchical system development and compositional reasoning.

Luiz Fernando Capretz, has introduced a new life cycle model for component-based development in [24]. He characterized software creation by change and instability. For this reason his proposed “Y” (Figure 9) software life cycle model considers overlap and iteration where necessary. The planned phases for the model are domain engineering, frame working, assembly, archiving, system analysis, design, implementation, testing, deployment and maintenance.

![Figure 9: The Y Model](image)

The main characteristic of this software life cycle model is the emphasis on reusability during software creation and evolution and the production of potentially reusable components that are meant to be useful in future software projects. Reusability implies the use of composition techniques during software development; this is achieved by initially selecting reusable components and assembling them, or by adapting the software to a point where it is possible to pick out components from a reusable library. It is also achieved through inheritance, which is naturally supported by an object-oriented approach [25]. Reusability within this life cycle is smoother and more effective than within the traditional models because it integrates at its core the concern for reuse and the mechanisms to achieve it.

Although the “Y” model initiated different new reusability aspects, still it needs expert knowledge to follow properly as it has no definite way to follow like top-down or bottom up. The research also concentrates on mainly component-based system development without considering any aspect of process for component development.
VII. CONCLUSION

In this paper, I surveyed several lifecycle processes starting from the traditional approach to component-based approach for software development and also provided a comparative discussion about those processes. As the technology of the future, CBSE faces many challenges today. The development life cycle is getting shorter, the software application domains expand rapidly, pressure on the software development costs is enormous and the requirements for integration of different domains are high. At the same time the technology in both software and hardware is changing extremely fast. And to compound it all, the requirements of the clients keep on evolving throughout the lifecycle of the development of the Software. The key to the survival is the capability to manage complexity and rapidly adapting to change. Without a possibility to reuse system components already developed it will be not possible to meet the ever evolving requirements. This means that the development process must change focus from programming intensive activities to reuse, integration, standards, management of complex and flexible structures, finding proper solutions, tradeoff analysis and marketing survey. The goal of CBSE is to standardize and formalize all disciplines supporting activities related to CBD. Success of the CBD approach depends directly on further changes in development procedures, tools, but also developers aptitude and organizational changes.

REFERENCES


<table>
<thead>
<tr>
<th>CBSD process by Sommerville</th>
<th>Y development process for CBSD</th>
<th>Y model for CBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements specifications</td>
<td>Requirements specifications are</td>
<td>Domain engineering phase is introduced to solve</td>
</tr>
<tr>
<td>are negotiated in the two phases of the lifecycle</td>
<td>are negotiated as an option, otherwise components are built</td>
<td>requirements negotiation problems</td>
</tr>
<tr>
<td>Only components are reused.</td>
<td>Only components are reused.</td>
<td>Not only individual components but also a collection of components of a particular domain or framework is also reused.</td>
</tr>
<tr>
<td>The lifecycle process is sequential.</td>
<td>The lifecycle process is sequential.</td>
<td>Not sequential. Can be utilized in top-down or bottom-up approach.</td>
</tr>
<tr>
<td>Lifecycle of the components are never mentioned.</td>
<td>Component lifecycle is considered as a separate process.</td>
<td>Component lifecycle is partially integrated with the CBSD lifecycle.</td>
</tr>
</tbody>
</table>

Table 1: Differences between CBSD Processes