Journal of Software Engineering and Applications, **2011, 4, 283-292**
doi:10.4236/jsea.2011.45031 Published Online May 2011 (http://www.SciRP.org/journal/jsea)

Moving from Traditional Software Engineering to Componentware

Faisal Nabi[1], Roisin Mullins[2]

[1]E-commerce Security Research Group, Hazraat Baba Bullah Shah Research Center, Qasoor, Pakistan; [2]IT Research Group, School of Business, University of Wales Trinity Saint David, Wales, UK.
Email: nabifaisal@yahoo.com, r.mullins@tsd.ac.uk

Received March 10th, 2011; revised April 13th, 2011; accepted April 21st, 2011.

ABSTRACT

The field of software engineering and software technology is developing very fast. Perhaps as a consequence, there is seldom enough interest or opportunity for systematic investigation of how the underlying technology will actually perform. That is, we introduce new concepts, methods, techniques and tools—or change existing ones and emphasize their value. A major turn in software engineering leading to Componentware has dramatically changed the shape of software development and introduced interesting methods for the design and rapid development of systems which may provide cost-effective benefits. In this paper we will discuss Componentware, process model, architecture, principles and the drivers, advantages, disadvantage and reveal profound changes from the traditional software engineering approaches.

Keywords*: *Software, Component, Componentware, Component Infrastructure, Software Pattern, Software Architecture, Component-Based-Development

1. Introduction

The past 20 years has been a time of change in the field of IT as we see a succession of new technological inventions. The software engineering field has gone through several generations of introductions of methods and techniques for developing large, complex systems. The 1970s was the decade of structured programming and structured methods. The 1980s, was the decade of data modelling and techniques. And the 1990s have focused on methods and techniques to support prototyping, iterative development, and a variety of rapid application development (RAD) techniques based upon the object oriented (OO) paradigm. The past practices associated with programming using machine code, assembly languages, high level languages, fourth generation languages, and further on to CASE tools, appear to act to solve some technological problems and help us meet requirements and develop better systems. However, those in the profession recognize that previous approaches have not managed to minimize project failures and project shortfalls [1,2]. One of the key drivers for changes to the software engineering approaches is the pace of change that has swept through businesses across the world. This has accelerated a need for software engineers to respond to the need for better and improved systems that meet the changing requirements, more flexible design and allow for easier reconfiguration. Businesses have become reliant on using and integrating IT and view these tools as being core to their strategies, competitiveness and the value they create for customers and supplier. The business place high value on their IT and continue to invest in new systems but in return they expect results. The overall system requirements from businesses have increased considerably. Having this in mind the search for new engineering techniques, in order to meet new business needs, to some extent is anticipated but requires a discerning rethink about the stages of change.

There is a concern that much of traditional software approaches are misunderstood and many software designers fail to appreciate the capabilities of software approaches. Therefore, there is a need to move to a stage where the importance of software approaches are highly valued and better understood.

The signal of a software crisis on the increase in businesses became apparent decades ago, then with the move to object-oriented approaches (such as the Object Modelling Technique) which became standard they were deemed equally unsuccessful for corporate developers. These outcomes have been repeatedly confirmed by research. Three key factors that exacerbate the software

crisis are:

- Requirements change
- Commercial innovation
- Distributed computing

A significant part of the problem is the belief that all user expectations are achievable. While it seems apparent that user requirements for systems have increased much faster than corporate developers' capability to deliver. The problem is also not helped by the fact that requirements changes are more frequent as businesses jostle for position in the market and try to find ways to increase their competitive advantage by means of strategic corporate software.

Another confounding factor is the destabilizing force of accelerating technology innovation, both in commercial software and hardware platforms. Corporate developers have difficulty finding compatible configurations of software products, and are forced to upgrade configurations frequently as new products are released. Further, software maintenance due to technology upgrades is a significant corporate cost driver.

Due to dominance of the Internet for business and geographically diverse enterprises, distributed computing is an essential feature of many new applications. Today's highly distributed enterprises require heterogeneous hardware/software, decentralized legacy configurations, and complex communications infrastructure. The result is an environment with frequent partial system failures. Distributed computing reverses many key assumptions that are the basis for procedural and object-oriented software development.

The recognized technology for the software industry has been object-oriented programming (OO) and has been widely adopted by new corporate development projects mainly because OO is universally supported by software tool vendors. There are many programmers receiving training for object-oriented development (e.g. C++ and Java) as corporations create new strategic systems. This acclaimed approach will not satisfy all the engineering requirements and it is likely that these developers and corporations are likely to become the next generation of disillusioned participants in the software crisis. However, the organizations that endure and thrive with this technology must use it in sophisticated new ways, represented by componentware.

Therefore, it can be argued that Componentware elements could present a fundamental rethink and provide a focus for changes in systems thinking, software processes, and technology utilization.

The componentware approach could address the failure of the majority of projects to meet their deadlines, budget constraints, quality requirements and should perform well against the ever increasing costs associated with software maintenance. Software developers as reported in the literature [2] who have built software systems for business by assembling components already developed and prepared for integration suggest that there are four elements of Componentware:

- Component Infrastructures
- Software Patterns
- Software Architecture
- Component-Based Development

Mowbray and Malveau (2004) suggest that "Componentware technologies provide sophisticated approaches to software development that challenges outdated assumptions. Together these elements create a major new technology trend. Componentware represents as fundamental a change in technology as object-orientation and previous generations" and this new age of software development is illustrated in **Figure 1** [2].

In this paper we address the background leading to the need for a Componentware approach. The move from traditional approaches to the Componentware era, component technology, the drivers and factors of Componentware, advantages, disadvantage and conclusion.

2. The Move from Traditional Software Engineering to Componentware

The ubiquitous nature and widespread use of the personal computers and the Internet has created new markets, encouraged new users and developers and importantly makes computers a core part of education, business and society.

The acceptance that technology is a part of everyday life has placed an additional emphasis for change in the software industry. For example, new users as consumers expect the cost of software to reduce in order to match the ever-decreasing hardware price. Further, demand for new applications such as electronic commerce and groupware are high. However, conventional methodologies have not witnessed huge gains in productivity and quality. This is a further realization of the need to re-think the way software development should proceed in the future.

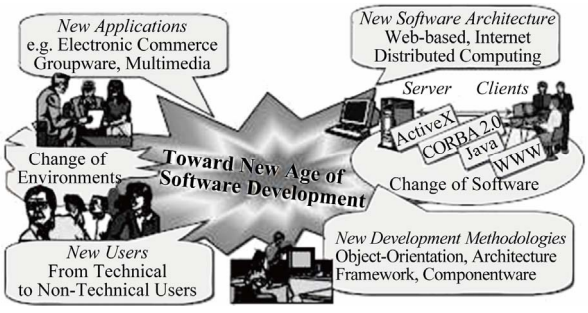

Figure 1. Change of environment and software.

Componentware has advantages in that they use components as the primary source and this would lead to a gain in productivity and quality. This is a view expressed by those from a matured engineering discipline [3]. As suggested by McIlroy, (1976) "making applications from software components has been a dream in software engineering community. My thesis is that the software industry is weakly founded, in part because of the absence of a software components sub industry. … A components industry could be immensely successful" [4]. However, it is clear that widespread reuse of software components in the software industry has not come about and a number of obstacles have been identified. However, it seems clear that the most fundamental problems are a lack of mechanisms to make components interoperable and there appears to be a lack of "reusable" components.

Since the early 1990's, approaches namely Componentware (CW) [5] or Component-Based Software Engineering (CBSE) has emerged [3,6-9]. The CBSE approach emphasized End-User Computing such as making possible to develop applications on the PCs. Further, the use of COTS (Commercial Off-The-Shelf) and Internet technology such as Web and Java-based technologies software promoted the CBSE approach for development business applications [10].

It is likely that if CBSE had been more widely accepted it could have opened up new possibilities for CBSE such as network distribution of components, and the reuse and interoperation of components over the Internet. In recent times, a few types of technologies have been widely deployed and are still evolving. They include ActiveX/DCOM from Microsoft [11], CORBA from OMG [12] and JavaBeans from SUN Microsystems [13].

2.1. What Differentiates CW/CBSE from the Conventional Software Reuse?

1) Conventional Software Reuse and CW/CBSE:

Although object-oriented technologies have promoted software reuse, there is a big gap between the whole systems and classes. To fill the gap, many interesting ideas have emerged in object-oriented software reuse in recent years. They include software architecture [14], design patterns [15], and frameworks [16].

2) CW/CBSE Approach: Componentware Engineering (CW)/Component-Based-Software Engineering (CBSE) as reported in the literature take different approaches, from conventional software reuse in the following ways.

a) Plug & Play: Component should be able to plug and play with other components or frameworks so that components can be composed at run-time without compilation.

b) Interface-centric: Components should separate the interface from the implementation and hide the implementation details so that they can be composed without knowing their implementation.

c) Architecture-centric: Components are designed on a pre-defined architecture so that they can interoperate with other components and/or frameworks.

d) Standardization: A component interface should be standardized so that they can be manufactured by multiple vendors and widely reused across the corporations.

e) Distribution through Market: Components can be acquired and improved though competition, being available on the market, and provide incentives to the vendors.

2.2. Process Model of CW and CBSE

CW/CBSE component-based software engineering allows the software development and delivery stages to evolve rather than follow a conventional approach and in doing so some parts of a system can be acquired from the component vendors and/or be outsourced to other organizations, some parts of software process can be done concurrently.

1) The architecture and the software process can be designed to allow software reuse, the software process should be reuse-oriented so that designers can reuse artifacts at different levels of abstraction along with software process.

2) **Figure 2** illustrates the conventional waterfall process and an example of CBSE process. As indicated in the figure the Componentware (CW)/CBSE process consists of two processes; component development and component integration. Since these two processes can be done by different organizations, these two processes can be concurrent. Unlike the conventional process, the CBSE process needs an additional process for component acquisition.

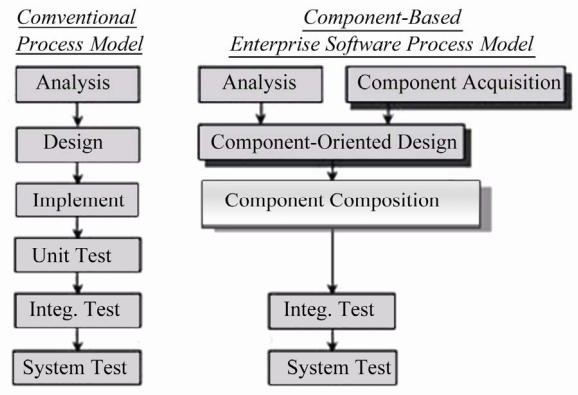

Figure 2. Conventional software process compared to component-based process model.

3. Component Technology

We will discuss these componentware technologies after a brief introduction to componentware's unique principles.

What is a Software Component?

A software component can be regarded as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard" [17].

A further example of a software component "is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by the third party" [3].

To simplify the concept of software components, we classify the notion of software components into three types:

1) Reusable Module/Unit is an independent and deliverable software part that encapsulates a functional specification and implementation for reuse by a third party.

2) A reusable component is an independent, deployable, and replaceable software unit that is reusable by a third party based on the unit's specification, implementation, and well-defined contracted interfaces.

3) A composite building block is a reusable component that is developed as a building part to conform a well-defined component model and the accompanying composition standards.

3.1. Component Architecture and Principles

When Component-based development is put into practice, it is necessary to divide the development into two parts that of supplier on one side and customer on the other side. In the organization the component developers are involved in the role of component supplier on one side and the developers of the software systems who are involved in assembly of these components have the role of component consumer on the other side.

This division can only be effective when a common basis is provided for suppliers and consumers of components. A component execution environment like COM or CORBA provide a partial solution. The main focus has been on technical issues concerning the communication between running software components. Therefore to aid the smooth process a component architecture is required to provide suppliers and customers with a set of rules, standards and guidelines that prescribe what components are and how they are assembled into software systems.

Component architecture therefore in our view is a set of principles and rules according to which a software system is designed and built with the use of components. The component architecture covers three aspects of a software system. These are:

1) Building blocks: The architecture specifies the type of building blocks systems are composed of.

2) Construction of the software system: The architecture specifies how the building blocks are joined together when developing an application. The architecture describes the role that building blocks play in the system.

3) Organization: Components are divided in categories based upon their functionality and these categories are placed in two dimensional layering.

3.2. Component vs Objects

As described by Szyperski C. (1998) a component's characteristic properties are "that it is a unit of independent deployment; it is a unit of third-party composition; and it has no observable state" [3]. This means that the component is independently deployable and it needs to be separated from the other components and the environment and it contains all its features.

In contrast to the properties characterizing components, an object's characteristic properties are that it is a unit of instantiation (it has a unique identity); it has state that can be persistent; and it encapsulates its state and behavior [3]. Again, several object properties follow directly. Since an object is a unit of instantiation, it cannot be partially instantiated. Further, since an object has an individual state, it also needs a unique identity so you can identify it, despite state changes, for the object's lifetime as shown in **Figure 3**.

Table 1 describes that although most components are composed of objects, there are fundamental differences between two structures. A Component can be differentiated from an object in that its "encapsulation" is guaranteed, that is there are no exposed implementation dependencies. An object might only be used within a single application, but a component has been designed with reuse in mind and can not assume much about the envi-

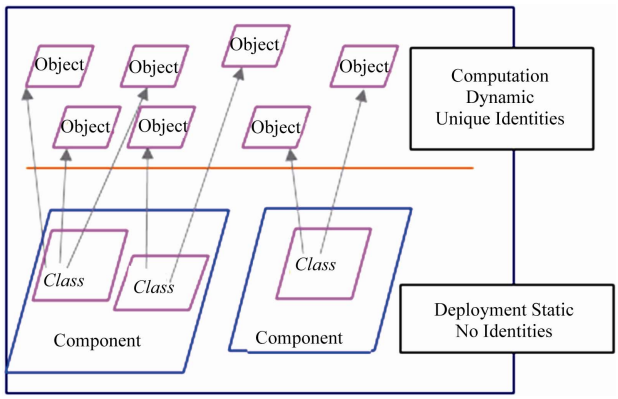

Figure 3. Components when activated create interacting objects.

Table 1. Summary of objects compared to components.

Objects	Components
Many Interdependencies	Few Interdependencies
Composed into Components	Composed into Services
Interface and Implementation Together	Separate Interface and Implementation

ronment in which it will be used. An object typically defines a much smaller portion of a problem space: an e-mail message, for example.

A component operates at a higher level, for example, for sending and receiving e-mail. A component typically contains a mechanism for configuring its operation in addition to its basic methods for performing its functions. This provides a means for a component to be adjusted to a range of different preferences, whereas an individual object dose not usually provides this range of configurability.

A component is often composed of a number of objects, which are designed to work together to provide specific functionality. Some component standards (such as EJB) also provide a recommended means to package their components for deployment (e.g, EJB's in jar file, containing objects themselves and a deployment description file).

Traditional Notions of Modules with Components: Module concepts do not normally support one aspect of Full-fledged components. For one, there are no persistent immutable resources that come with a Module, beyond that which has been hardwired as constant in the code. Resources parameterize a component and replacing these resources lets you version a component without needing to recompile; for example localization. Further, modification of resources may look like a form of a mutable component state. Since components are not supposed to modify their own resources (or their code), this distinction remains useful; resources fall into the same category as the complied code that forms part of a component. An additional aspect of the components that is not usually associated with modules is the configurability of dependencies.

3.3. Componentware

The term "Componentware" has been argued in the scientific community, and it is best considered as a reincarnation of object-orientated programming and other software technologies. There are four basic principles: encapsulation, polymorphism, late binding, and safety that distinguish componentware from previous technology generations.

Except for inheritance, the list overlaps with object-orientated approaches. The concept of a component means that inheritance does not have a fit/support, because it is loosely coupled and often distributed. **Figure 3**, clearly explains component's functionality by invoking other objects, and this is called delegations.

All components specification correspondence to its implementations. The specification states, its public interfaces with other components that define a component encapsulating a functional specification and implementation for reuse. Whereas, polymorphism supports reuse of component specifications that can be local or global standards for reuse throughout a system.

Componentware supports composition allowing integration and specifications from constituent components for building systems.

3.4. Component Infrastructures

It has been reported [18] that many global businesses including Microsoft, Sun Microsytems, IBM and the CORBA consortia have established significant componentware infrastructures through massive technology and marketing investments.

These component infrastructures are described as dominant infrastructures extending beyond industry segments and of course they share similarities. A key feature is their ability to be mutually interoperable and they are becoming mainstream development platforms.

For example [18] "Microsoft has been promoting the Component Object Model (COM) and follow-on products for several years. COM is a single-computer component infrastructure. OLE and ActiveX define componentware interfaces based upon COM. In theory, the Distributed Component Object Model (DCOM) extends the capabilities of COM over networks and the Internet". It has been reported that there has been a remarkable shift in businesses such as Microsoft to migrate to componentware and to continue to promote componentware in the future.

3.4.1. Software Patterns

Software patterns are often simply described as the design tools, knowledge reuse and describe the manner of modelling business systems. These are described by Mowbray, T. J. and R. C. Malveau, (1997) [19,20] as "comprising a common body of software knowledge which can be applied across all component infrastructures. The most famous category of software patterns, called design patterns, comprises proven software design ideas which are reused by developers". Other important categories of patterns include analysis patterns and anti-patterns [20]. The software patterns are important in understanding design concepts of componentware and are important for specifications and implementation. It is expected that software patterns will ameliorate many of the common problems reported when developing ob-

ject-oriented systems.

3.4.2. Software Architecture

Software architectures are concerned with design context, including planning, maintenance, operations, stable interface specifications, and the design and reuse of components. These stable architecture designs present interesting opportunities for distributed projects teams.

3.5. Component-Based Development

There are a number of advantages and factors that stimulate interest in component-based development. According to Brown, (1996) the following factors have contributed to make the component-based development approach become so popular [7].

- The need to automate the software development process.
- The economic push moving many organizations towards greater use of available commercial solutions.
- The style and architecture of the applications being developed has significantly changed. There has been a major change from centralized mainframe-based applications towards multi-tiered applications.
- A set of infrastructure standards and supporting technologies has emerged during the last few years.
- Domain-specific libraries and frameworks are starting to appear.
- Leading companies within the software industry have announced component-based development strategies.
- The web infrastructure is maturing. While there is a great deal of chaos and competing technology, there is also a basic shape to the infrastructure that allows collections of independent developed software applications to be searched, remotely invoked, communicated, and share data.

Component-Based Development Roles: Component-Based-Development is a term that describes the process of creating applications from existing components and it involves more than just deployment and creates the units of functionality, which are later assembled into an application [21]. The component-based development process includes reuse and is an iterative, incremental development process and requires specialist technical roles.

These componentware roles are described as "component system architect, component framework architect, component programmer and component assembler/application developer and these roles are different from object-oriented approaches which are more reliant on management decision making.

3.6. The Component-Based Development Process

Component-based development process appears as two different activities.

1) The component-based application development process.

2) The component development process respectively.

The component-based application development process focuses on assembly software components that supply user services driven by specific business requirements. Whereas the component development process focuses on acquiring, wrapping and building the actual components.

Component Application Development

Many authors are in agreement that the component application development process considers the design phase as a major aspect as it captures specification, assembly selection of components and integration. According to Brown (1996) "the development of component-based systems introduces fundamental changes in the way systems are acquired, integrated, deployed and evolved. Rather than the classic waterfall approach to system development, systems are designed by examining existing components to see how they meet the system requirements. This is followed by an iterative process of refining the requirements to match the existing components, and deciding which of those components that can best be integrated to provide the necessary functionality. Finally, the system is engineered by assembling the selected components using locally developed code". It is further suggested that [7] "the real difference between traditional application development and component based application development takes place within the design phase, and it is in the design that you begin paying attention to how the objects you have identified, during the analysis phase, can be grouped into components".

These component states are best described by Brown (1996) as [7] (**Figure 4**):

- **"Gathered or off-the-shelf components (COTS)**

Consists of those identified as being of potential interest, and requires a process of investigating those components which may come from a variety of local and remote sources. At this stage little may be known about a component's characteristics. The information available may simply be its name, its parameters, and some idea of its required operating environment.

- **Qualified components**

Have discovered interfaces so that possible source of conflict and overlap among components has been identified. Discovery can take many forms, but usually involves detailed analysis of any available component documentation or specifications, discussions with component developers and users, and trial execution of the component in different settings.

- **Adapted components**

Have been amended to address potential sources of

conflict. Typically, simple scripts are written as a buffer between user request and component actions. The buffer can be used to provide default information to the components, eliminate access to unwanted component behavior, and act as an insulation layer for the replacement of components. The figure implies a kind of component wrapping.

- **Assembled components**

Have been integrated via some form of common infrastructure. For consistent component assembly the infrastructure must consist of both a physical communication infrastructure such as messaging infrastructure and a set of conceptual agreements such as naming convention that embody the shared semantics among the components. This infrastructure will support component assembly and coordination, and differentiates architectural assembly from ad hoc glue.

- **Updated components**

Have been replaced by newer versions, or by different components with similar behavior and interfaces. This aspect of component assembly must be expected and well supported as an essential activity through appropriate definition of component interfaces and controlled interaction among components".

Further, a key aspect of component development is to design for component reuse.

3.7. Architectural Design of a Component-Based System

The importance of a stable, solid and dynamic system architecture is critical to large-scale systems and is of utmost importance for component-based systems. According to Bass (1998) architecture is, "an abstract view distinct from the details of implementation, algorithm, and data representation" [22]. Architecture is, increasingly, becoming a crucial part of a software organization's business strategy because it is a reusable asset in itself that can be reapplied to subsequent systems.

Therefore, it has increased value for the business and presents opportunities for new business strategies.

A common view is that component-based development requires a planned and disciplined approach to the architecture of the system being built. According to Clements (1995) "purchasing components at random will result in a collection of mismatched parts that will have no hope of working in unison" [23]. This has been further advocated by Garlan (1995), as outlined in [24] who points out that "the reason why even carefully considered sets of components may be unlikely to successfully operate with each other is that designers of software components make assumptions that are often subtle and undocumented about the ways in which the components will interact with other components, or the expectations about services or behaviours of those other components". It has been suggested that component models do not solve all architectural mismatch problems.

However, there is a component-based development approach known as layered systems and this approach "is based on dividing into different groups (layers) based on what kind of services the components are offering, that is the characteristics of the components". This approach suggests greater efficiency for development and maintenance projects.

The unified principle of the layered system architecture approach is that a component at a particular layer is allowed to make use only of components at the same or next lower layer.

Thus, components at each layer are insulated from change when components at distant layers are replaced or

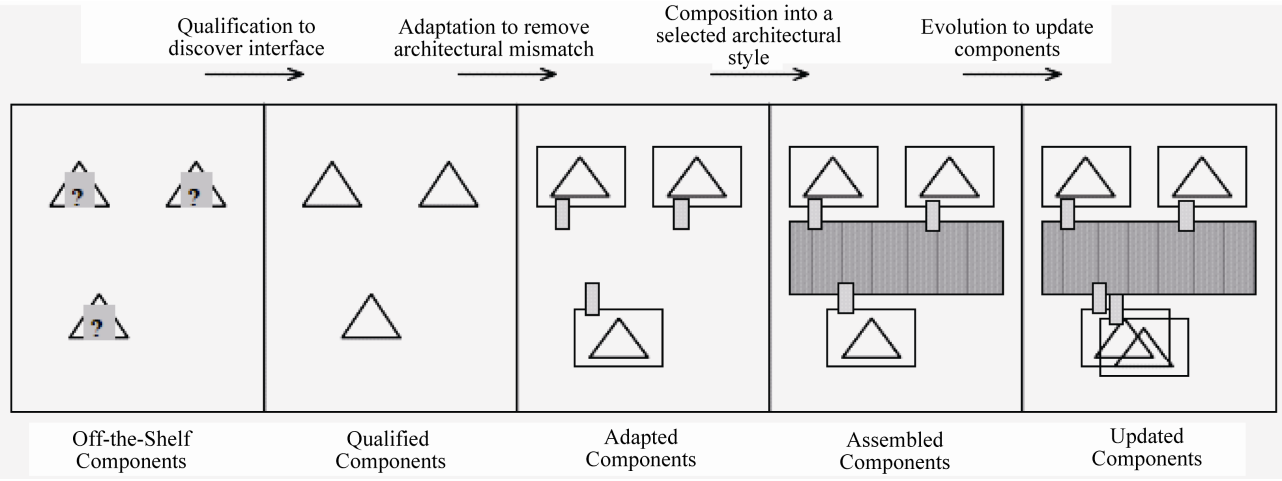

Figure 4. Reference model for component assembly [7].

Table 2. Difference between component-based software and traditional software model for component assembly [26].

Perspective	Component-Based Software	Traditional Software
Objective	To efficiently develop reliable, maintainable, reusable software systems	To develop efficient, reliable software systems
Infrastructure	Beside operating systems and programming languages, certain component models are needed. For instance NET/COM +, EJB, or CORBA	Only need the support of operating systems and programming languages
Process	Component-based software engineering process – includes many unique activities such as component selection, customization, composition	Traditional software engineering process – includes requirements analysis, design, coding and testing
Roles	Component providers Component users	Usually, only one development team participates in the entire process.
Portability	Relatively easy to achieve, because the adopted component model and component architectures will reconcile many incompatibility and interoperability issues.	Difficult to achieve as the entire engineering process is often for one specific environment
Interoperability	1. Communications among components will go through interfaces and are managed by component model. Component model can manage some of the interoperability issues for you. 2. When combining varied components, interoperability issue is more serious	Very often, traditional software systems are developed in a restricted context, and there are less interoperability issues. Once there are, it is hard to achieve as modules and subsystems are context dependent
Maintainability	1. When producer provides newer version, system can be easily upgraded 2. Less control over how and when components are maintained 3. May introduce more overhead when source code is not available 4. Versioning is handled by component providers independently, so potential conflicts may exist	1. When maintaining the software, any modification may require reconstruction of the entire system 2. Have full controll of any maintenance activity
Reusability	Reusability can occur at any different level: component, component architecture	Difficult to reuse; for some OO systems, the reuse is limited to individual classes
Scalability	Easy to scale: the scalability can be automatically managed by component model	Hard to scale as any change in the code may require changes to all relevant modules or subsystems
Structure	Loosely coupled Often distributed	Usually tightly coupled and centralized system
Code availability	Some component source code may not be available	Source code is usually available
Customization	1. Many components are general purpose components and need customization before composition 2. Introspection mechanism provided by the component systems to explore available systems	When integrating modules into the final product, no customization is needed

modified" [24]. Additionally, according to Clements (1996) the study and practice of software architecture is still immature. Rigorous techniques need to be developed to describe software architecture so that they can be analyzed to predict and assume non-functional system properties [25] and because a solid and dynamic architecture is so critical to component-based systems, the architecture-based activities need to be more precisely defined and supported with processes and tools, and need to be smoothly incorporated into existing development processes.

4. The Drivers of Componentware

There are many deficiencies in the traditional software engineering approaches that have led to an acceptance to move to componentware which allows rapid "Assembly" of parts (Components), the development of parts as "reusable entities", maintenance and upgrading of systems by customizing and replacing such parts. A comparison of component-based software and traditional software is presented in **Table 2**.

The move to Componentware also reveals that businesses are forced to find ways to address the problems with traditional approaches. The common failure such as, the majority of projects unable to meet their deadline, budget, quality requirement and the continued increase in the costs associated with software maintenances can not continue and needs to be addressed by adopting a new

approach.

The traditional systems objective is "to deliver an efficient and reliable software system. Reusability, maintainability and ability to evolve are characteristics that are not among the top priorities. Nevertheless, as software systems get larger and larger, and more and more complex, these characteristics become more critical [26].

Finally Componentware needs to address the following objectives: shorter time to market, lower cost, better reusability, higher reliability, and maintainability if it is to be accepted by the software industry and software developers.

5. Disadvantage of Componentware

It has been suggested that "the component users do not have full control of the entire engineering process, which can introduce difficulties in component composition, selection, testing, and maintenance, as well as many other activities" [26]. Therefore more thought needs to be given to understand how the separate roles and distinct working patterns of designers and users can adjust to utilizing the innovative capability offered by the Componentware approach.

6. Conclusions

The move to greater use of Componentware is the next major software technology trend. At the present time the uptake has not been widespread as developers prepare for the transition.

It is readily available for commercial exploitation. The move to Componentware has been widely accepted and actively supported by major vendors, including Microsoft, Sun, IBM, and the CORBA vendor consortia.

The most important aspects of componentware are not the choice of technologies, but how these are applied. Successful adoption of componentware must include the reuse of software patterns, the planning of software architecture, and the establishment of component-based development teams and recognition of specific roles.

The move to Componentware is an exciting opportunity to advance software development and address the shortcomings of outdated software approaches.

This paper seeks to clarify that if Componentware is adopted it should enables modern business practices and expectations in ways that could not be achieved with previous approaches especially when business have to face the challenges of requirements change and rapid commercial innovation. There is good evidence to support that Componentware delivers the benefits of software reuse and enables outsourcing to distributed project teams which is of high importance and in addition it creates new value for businesses.

REFERENCES

[1] B. Dahlbom and L. Mathiassen, "The Future of Our Profession," *Communications of the ACM*, Vol. 40, No. 6, 1997, pp. 499-519. doi:10.1145/255656.255706

[2] T. J. Mowbray and R. C. Malveau, "Software Architect Bootcamp," Prentice Hall PTR, London, 2004.

[3] C. Szyperski, "Component Software: Beyond Object-Oriented Programming," ACM Press, Addison-Wesley, New York, 1998.

[4] M. D. McIlroy, "Mass-Produced Software Components, Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering)," Van Nostrand Reinhold, New York, 1976.

[5] J. Udell, "Componentware," *Byte*, Vol. 19, No. 5, 1994, pp. 46-56.

[6] M. Aoyama, "Componentware: Building Applications with Software Components," *Information Processing Society of Japan*, in Japanese, Vol. 37, No. 1, 1996, pp. 71-79.

[7] A. Brown, "Component-Based Software Engineering," IEEE CS Press, Washington DC, 1996.

[8] D. Kiely, "Are Components the Future of Software?" *IEEE Computer*, Vol. 31, No. 2, 1998, pp. 10-11.

[9] J. Sametinger, "Software Engineering with Reusable Components," Springer-Verlag, Berlin, 1997.

[10] K. Wallnau and D. Carney, "COTS Products and Technology Evaluation: Concepts and Pragmatics," *20th International Conference on Software Engineering Tutorial*, Kyoto, 19-25 September 1998.

[11] "Microsoft," 1998. http://www.microsoft.com/activex/

[12] "OMG," 1998. http://www.omg.org

[13] A. Thomas, "Enterprise JavaBeans: Server Component Model for Java," White Paper, 1997. http://www.javasoft.com/products/ejb/.

[14] M. Shaw and D. Garlan, "Software Architecture, Perspectives on an Emerging Discipline," Prentice Hall, London, 1996.

[15] E. Gamma and R. Helm, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, New York, 1995.

[16] M. E. Fayad and D. C. Schmidt,"Object-Oriented Application Frameworks," *Communications of the ACM*, Vol. 40, No. 10, 1997, pp. 32-38. doi:10.1145/262793.262798

[17] B. Councill and G. T. Helneman, "Component-Based Software Engineering: Putting the Pieces Together," Addison-Wesley, New York, 2001.

[18] "Componentware," 2010. http://www.phptr.com

[19] T. J. Mowbray and R. C. Malveau, "Corba Design Patterns," John Wiley & Sons, New York, 1997.

[20] W. J. Brown, R. C. Malveau, H. W. McCormick and T. J. Mowbray, "Antipatterns: Refactoring Software, Architectures, and Projects in Crisis," John Wiley & Sons, New York, (1998).

[21] N. Michael, "Java Frameworks & Components: Accelerate Your Web Application Development," Cambridge University Press, Cambridge, 2003.

[22] L. Bass, P. Clements and R. Kazman, "Software Architecture in Practice," Addison-Wesley, New York, 1998.

[23] P. E. Clements, "From Subroutines to Subsystems: Component-Based Software Development," *American Programmer*, Cutter Information Corporation, 1995.

[24] D. Garlan, R. Allen and J. Ockerbloom, "Architectural Mismatch," *Proceedings of the* 17*th International Conference on Software Engineering*, Washington, 24-28 April, 1995.

[25] P. E. Clements, "Software Architecture: An Executive Overview," Software Engineering Institute, 1996.

[26] J. Z. Gao, H. S. J. Tsao and Y. Wu, "Testing and Quality Assurance for Component-Based Software," Artech House Computer Library, Boston-London, 2005.

