Component Based Software Development Using Component Oriented Programming

Ruchi Shukla[1] and T. Marwala[2]

[1] Department of Electrical and Electronic Engineering Science,
University of Johannesburg,
Johannesburg,
South Africa
ruchishuklamtech@gmail.com
[2] Fac. of Engineering and Built Environment,
University of Johannesburg
tmarwala@uj.ac.za

Abstract. Software industries today are striving for techniques to improve the software developer's productivity, software quality and flexibility within the constraint of minimum time and cost. Component based software development (CBSD) is proving more suitable for the evolving environment of software industry. This paper demonstrates a sample application of component-oriented programming concepts for CBSD. Some of the potential risks and challenges in CBSD are also presented.

1 Introduction

The rapid growth in software and IT industry is leading to new software development paradigms, demanding faster delivery of software. Component based software (CBS) has recently started receiving attention among vendors, developers and IT organizations. There is a definite shift from structured programming written software for mainframe systems to object-oriented UML designed Smalltalk/C++ written client server software, to component based ADL designed C++/Java written N-tier distributed systems.

Reusability is the key issue today for building software systems quickly and reliably. The software system should be able to cope with complexity and adaptable to changing requirements by adding, removing and replacing the software components. Due to this component oriented programming has become the most popular programming technique [10]. With the emergence of component based software engineering (CBSE), component based software development (CBSD) has become an inevitable paradigm leading to less manpower/cost, increased quality, productivity and flexibility, reduced time to market, better usability and standardization.

According to Heineman and Councill: "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a component standard". A more widely accepted definition by Szyperski is: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties" [8].

The interface of a component consists of the specifications of its provided and required services. To specify these dependencies precisely, it is necessary to match the required services to the corresponding provided services. Component model defines how components can be constructed, assembled and deployed [12]. The current state of component models usage justifying the need for a component model selection framework was presented in [2]. However, the question arises as to how these CBS are developed for reusability and what are the metrics taken into consideration. Further, the success of CBSD using third-party components mainly depends on the selection of a suitable component for the intended application [3].

The rest of the paper is structured as follows: Section 2 and 3 present a brief overview of the CBSD process and the Component frameworks respectively. Section 4 demonstrates a basic example of CBSD employing component-oriented programming concept. Section 5 presents some critical risks and key challenges in CBSD while Section 6 concludes the work.

2 Component Based Software Development Process

The CBSD approach uses various similar components identified in various software systems from Commercial-off-the-shelf (COTS) components for large-scale software reuse. CBSD consists of the following major activities [5]:

(1) requirements analysis,
(2) software architecture selection and creation,
(3) component selection,
(4) integration, and
(5) component-based system testing.

The development cycle of a component-based system is different from the traditional (waterfall, spiral, iterative and prototype based) models. Figure 1 shows a comparison between the traditional waterfall model and the modern CBD process. The requirements gathering and design in the waterfall process model corresponds to finding and selecting components. Similarly, the implementation, test and release in the waterfall model are equivalent to creating, adapting and deploying the components, and maintenance corresponds to replacing the components [5].

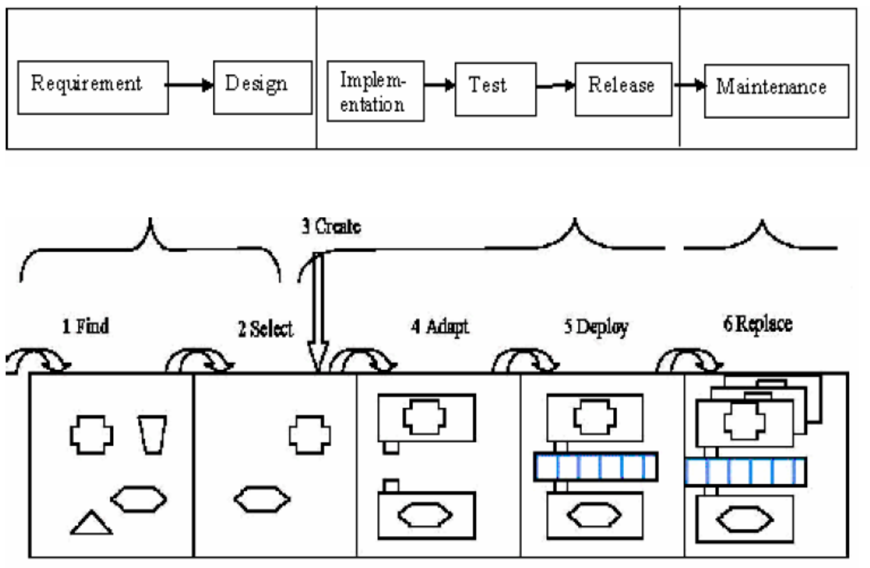


Fig. 1. Comparison of Waterfall Model Cycle With Component Based Development [5]

Most of the research so far has focused on the development and use of components within the following two development paradigms

1. Rapid application driven (RAD) paradigm where visual tools are used to create user interface and associate components with elements identified in interface, suitable for small to medium sized systems [13].

2. Object-oriented analysis and design (OOAD) paradigm where development takes place based on a conventional software life cycle and is oriented to the tasks and relevant objects, including their interactions, generalization and composition.

Here, we represent the OOAD approach from three aspects, i.e. functional, behavioural and structural, corresponding to use case, communication diagram and class diagram respectively. A component includes several use cases. A communication diagram is used to obtain the object usages and depict the dynamic behaviour of each use case. A set of participated classes are also specified by the communication diagram. We can extract the structural relationships among the objects from the class diagram. The relationship between classes and components is shown in Figure 2.

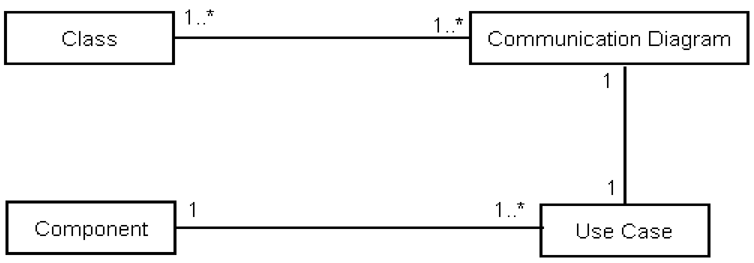


Fig. 2. Relationship Between Classes And Components

The process of component identification begins by clustering related objects and making them reusable. Each clustered object identified from clustering object approach are relating to their corresponding classes for allocating candidate components. Finally, in the last phase, we can identify reusable components from refining the candidate component [7, 9, 14].

3 Component Frameworks

A framework is a set of constraints on components and their interactions. Three standardized component frameworks are: CORBA, COM/DCOM, JavaBeans/EJB. The distributed version of COM is the DCOM. In the year 2002 .NET was released, which presents a platform-independent target for software development. It relies mainly on software component and the COP paradigm. EJB architecture is another component based architecture for developing and deploying component objects.

Any component can exhibit varying degree of distribution, modularity and independence of platform or language. Mapping of components is done on the following 3-dimensional space [4]:

1. Monolithic systems (0,0,0) – non-distributed, non modular, language dependent and platform dependent.
2. VB components (0,1,0) - neither distributed, nor language independent.
3. CORBA – distributed and language independent but the underlying components often remain platform dependent.

Java components are cross platform in scope. Wrapping a platform independent language such as Java with language independent middleware such as CORBA would yield a component worthy of (1,1,1) status [4].

4 Component Oriented Programming

Component-oriented programming (COP) enables programs to be constructed from reusable software components, following certain predefined standards including interface, versioning, deployment and connections [15]. COP as against OOP includes Polymorphism + Real late binding + Real and Enforced encapsulation + Interface inheritance + Binary reuse. COP allows various kinds of reuse including white-box reuse and black-box reuse. White-box reuse means that the source of a software component is made available and can be studied, reused, adapted, or modified. Black-box reuse is based on the principle of information hiding.

Example-1 demonstrates how to write a .NET serviced component that implements the IMessage interface and displays a message with "Hello Component" in it when the interface's ShowMessage( ) method is called [15].

Example-1: A simple .Net component

```
using System;
using System.EnterpriseServices;
namespace MyNamespace
{
public interface IMessage
{
void ShowMessage( );
}
public class MyComponent:ServicedComponent,IMessage
{
public MyComponent( ) //Default Constructor
{
}
public void ShowMessage( )
{
Console.WriteLine("Hello Component! ");
}
}
}
```

4.1 Registering Assemblies

Before adding the serviced components to a COM+ application, we need to register their assembly with COM+ [11]. This can be done using the RegSvcs.exe command line utility. The code then has to be signed with a cryptographic key. Open the *AssemblyInfo.cs* file, and at the bottom, insert the full path to the key against the AssemblyKeyFile entry [6]. The step is shown below:

```
C:\MyNamespace\MyNamespace\bin\Debug>regsvcs MyNamespace.dll
Installed Assembly:
Assembly: C:\MyNamespace\MyNamespace\bin\Debug\MyNamespace.dll
Application: MyNamespace
TypeLib: C:\MyNamespace\MyNamespace\bin\Debug\MyNamespace.tlb
C:\MyNamespace\MyNamespace\bin\Debug>
```

Now, the component is installed. If we open Component Services (Start - Settings - Control Panel - Administrative Tools - Component Services) and navigate down through the tree to COM+ Applications, we can see our newly installed application Figure 3.

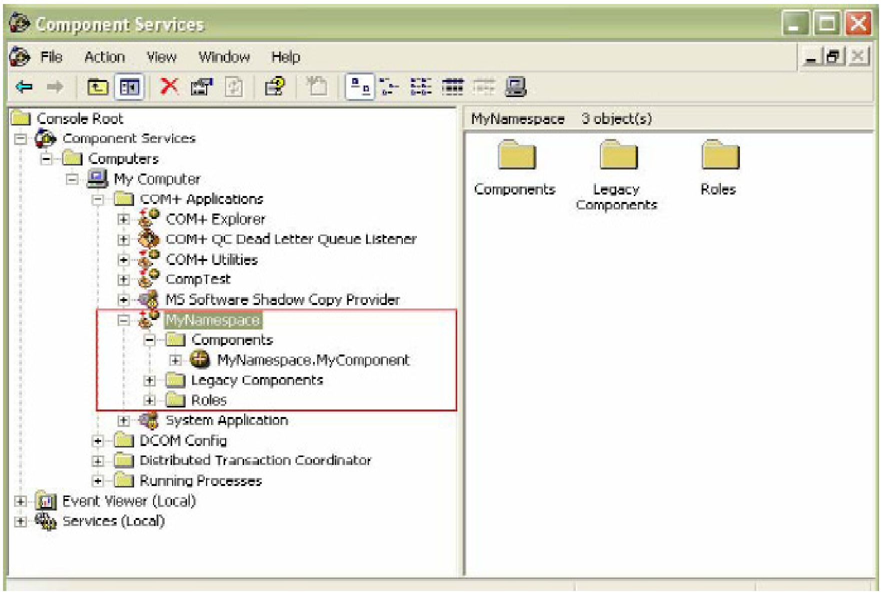


Fig. 3. Component Services

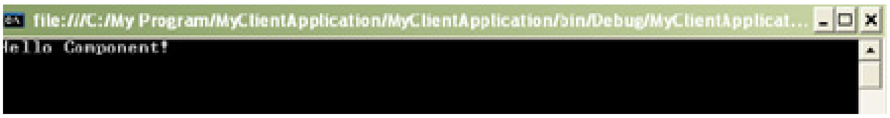
4.2 The Client Application

After building the serviced component library, we can create a client application. Then we can write the code to instantiate a new MyClient instance, and invoke the method ShowMessage ( ) [15]. The client code is shown below:

Example-2: A simple client application

```
using System;
using System.EnterpriseServices;
using MyNamespace;
namespace MyClientApplication
{
class MyClient
{
static void Main(string[ ] args)
{
MyComponent mycom = new MyComponent( );
mycom.ShowMessage( );
Console.ReadLine( );
}
}
}
```

The output is as shown in Figure 4.


file:///C:/My Program/MyClientApplication/MyClientApplication/bin/Debug/MyClientApplicat...

```
Hello Component!
```

Fig. 4. Client Application

5 Risks and Challenges in CBSD

Risks

- Changing nature of modern day software products and robustness in CBD.
- Reusable components are not designed and not coded from scratch.
- Development based on in-house, multi-origin reusable, 3rd party COTS software or open source software components.
- Few empirical studies that investigate how to use and customize COTS-based development processes for different project contexts [3].
- Component interfaces are defined by models with less information for functional testing.
- Instability in CBSD standards.
- Reliability of CBS.
- A usage of web based COTS leads to system security threats.
- Architectural risks of CBS and its agility with respect to software architecture, quality and maintenance.
- CB risk analysis should adopt similar principles of encapsulation and modularity as CBD methods (ISO, 2009a, b).
- Generalization of components for future reuse.

Challenges

- Challenges in component configuration during integration.
- Many complicated CBSD process models have been proposed, but no step-by-step guidance for implementing them is available [2].
- Limited knowledge about current industrial OTS selection practices.
- Effort estimation and fault identification.
- Relevant or new suite of software metrics and effort estimation and costing models for component assemblies are still not available.
- Challenges in multi-language component selection and integration.
- Lack of formal component selection methods and non availability of documentation impacting the component integration.
- Challenges of using semi-formal techniques like UML and formal techniques like VDM, Z and B in the early stages of component development.
- Challenges to check the presence of virus due to non availability of source code.
- Prediction of system behavior from component behavior.
- The maintenance process of CBS from system to product to component level.

Complexities involved in the development of an effort estimation tool for CBSD

- Relatively new concept, hence limited published metrics and available tools [1].
- Information about system artefacts, relationships and dependencies can be obscure, missing, or incorrect as a result of continued changes to the system.
- Changes must conform or be compatible with an existing architecture, design and code constraints.
- Multi-language, multi-platform software.
- Real time (dynamic/probabilistic/adaptive) components and cost drivers.

6 Conclusions

This paper demonstrates by means of a simple example, the use of component concepts and COP based software development method. The basic aim was to orient the programmers towards using an innovative software development approach for real life projects. This approach is a beginning of seamless support and better integration of the development tools, runtime, component services, and the component administration environment. However, the use of COTS components comes with its own challenges and risks which need to be analysed before arriving at a decision to use them for CBSD.

Acknowledgments. The financial support offered to RS by the FEBE of the University of Johannesburg, Johannesburg is gratefully acknowledged.

References

1. Aris, H., Salim, S.S.: Issues on the application of component oriented software development: Formulation of research areas. Inform. Tech. J., 1–7 (2008)
2. Aris, H., Salim, S.: State of component models usage: Justifying the need for a component model selection framework. I. Arab J. Inform. Tech. 8(3), 310–317 (2011)
3. Ayala, C., Hauge, O., Conradi, R., Francha, X., Li, J.: Selection of third party software in Off-the-shelf-based software development-An interview study with industrial practitioners. J. Syst. Softw. 84, 620–637 (2011)
4. Brereton, P., Budgen, D.: Component-based systems: A classification of issues. Comput. 33(11), 54–62 (2000)
5. Crnkovic, I.: Component-based software engineering – New challenges in software development. In: 25th International Information Technology Interfaces (ITI) Conference, Cavtat, Croatia (2003)
6. `http://www.codeproject.com/Articles/6736/A-Very-Simple-Persistent-Cache-in-a-COM-Component` (accessed March 31, 2012)
7. Kim, S.D., Chang, S.H.: A systematic method to identify software components. In: 11th Asia-Pacific Software Engineering Conference, Seoul, South Korea, pp. 538–545 (2004)
8. Lau, K.K., Wang, Z.: Software component models. Trans. Softw. Engg. 33(10), 709–724 (2007)

9. Lee, S.D., Yang, Y.J., Cho, E.S., Kim, S.D., Rhew, S.Y.: COMO: A UML based component development methodology. In: 6th Asia Pacific Software Engineering Conference, Takamatsu, Japan, pp. 54–61 (1999)
10. Liu, Y., Cunningham, H.C.: BoxScript: A component-oriented language for teaching. In: 43rd ACM Southeast Conference, Kennesaw, USA (2005)
11. Lowy, J.: Component services, `http://ondotnet.com/pub/a/dotnet/excerpt/com_dotnet_ch10/index.html?page=3` (accessed March 31, 2012)
12. Mahmood, S., Lai, R., Kim, Y.S.: Survey of component-based software development. IET Softw. 1(2), 57–66 (2007)
13. Panfilis, S.D., Berre, A.J.: Open issues and concerns on Component Based Software Engineering. In: 9th International Workshop on Component-Oriented Programming, Oslo, Norway (2004)
14. Sook, M., Cho, E.S.: A component identification technique from object-oriented model. Springer, Heidelberg (2005)
15. Wang, A.J.A., Qian, K.: Component-oriented programming. John Wiley & Sons (2005)