CHAPTER 4

COMPONENT-BASED SOFTWARE DEVELOPMENT SYSTEM

4.1 INTRODUCTION

Modern software systems become more and more large-scale, complex and difficult to control, resulting in high development cost, low productivity, unmanageable software quality and high risk to move to new technology [72]. Consequently, there is a growing demand for a new, efficient, and cost-effective software development paradigm. One of the most promising solutions today is the component-based software development approach. This approach is based on the idea that software systems can be developed by selecting appropriate off-the-shelf components and then assembling them with well-defined software architecture [71]. This new software development approach is very different from the traditional approach in which software systems can only be implemented from scratch.

Software components are easier than traditional programs to maintain, support, and modify for future requirements. The advantage of taking a server-centric approach [in the financial industry], is that we can define a business process, write it as a set of software components once, and then deliver it across multiple channels [15].

Components yield several advantages; perhaps their main benefit lies in faster application development, which, in turn, takes a product to
market faster and at a lower cost. Instead of having to research, write and test code, developers can buy pre-built components that have been written by experts in the field. Java or .NET based components, for example, can help an organization save an estimated 80 to 90 percent on development time by buying ready-made components [98].

Component-Based Software Development is an emerging discipline that is generating tremendous interest in the development of plug-and-play reusable software. This has led to the concept of ‘commercial off-the-shelf’ (COTS) components [33]. Component technology has become a central focus of software engineering in research and development due to its great success in market. Reusability is a key factor that contributes to this success [61]. Component-based software development (CBSD) is recognized today as an effective way to reduce development costs and time-to-market [68].

Component-based development has improved quality, increased productivity and effective management for complex software leading to wider range of usability [8][17]. Components-based development improves software development productivity and software quality by re-using the existing well-tested software components. The advantages of component based software development compared to traditional software development are given in Table 4.1[88]. The integration of software components has become one of the issues in software engineering [22]. Developing component-based systems becomes feasible due to the following:

- Enhancement in the quality and increase in variety of **Commercial Off-The-Shelf** (COTS) products.
- Economic pressures to reduce system development and maintenance costs.
- The emergence of effective component integration technology.
- The increasing amount of existing software in organizations that can be reused in new systems.

<table>
<thead>
<tr>
<th>Component-Based Software Development</th>
<th>Traditional Software Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building system from preexisting Components.</td>
<td>Building system from scratch.</td>
</tr>
<tr>
<td>Components and systems integrated from those Components are developed.</td>
<td>Software system is developed.</td>
</tr>
<tr>
<td>Component selection and evaluation are special lifecycle Phases.</td>
<td>In the lifecycle, there is no Special phase like that.</td>
</tr>
<tr>
<td>Much effort is required in the selection of components, testing &amp; verification phase</td>
<td>Much effort is required for system development</td>
</tr>
<tr>
<td>Reusability is the main theme.</td>
<td>Reusability usually not Considered</td>
</tr>
</tbody>
</table>

**Table 4.1 Component-based system versus Traditional software system**

When we look at developing software systems using components, it is the work of integrating the components with each other and the rest of the system that is the most important part of the component-based development process. Depending on the component model used and the actual components that are to be integrated, more or less work will have to be done in order to get all the parts of the system to function correctly together [47].
A component model is the framework, through which the components are integrated together to compose a system. They are also called component frameworks or component technologies. Well known component models are JavaBeans from Sun, COM/DCOM/COM+ from Microsoft, CORBA from OMG, and finally .NET which is Microsoft’s latest component model [18].

Integration of software components is an emerging concept in the software development process and it became the priority of many businesses, because business initiatives and mandates demand consistency, particularly for those IT systems with which customers and partners interface. Failure to achieve success in this endeavor results in longer production cycles, increased development costs, lack of trust on the part of customers and business partners, and, ultimately, loss of business [78].

Components must be integrated through some well-defined infrastructure. Interoperability is the key concept for integrating the components developed in different languages or in different platforms. All current integration broker suites are limited in their support for multiple component-models and platforms, being heavily biased towards development using particular component model such as EJB or J2EE (for most vendors) and COM /.NET for Microsoft based solutions. As such, current brokers are unable to easily compose distributed integrations across multiple component models already deployed within an enterprise.
4.2 INTEGRATION OF PLATFORMS

The Bridge between .NET and J2EE can be provided in any one of the following forms.

- Migration
- Portability
- Interoperability

*Mutation* provides the bridge between .NET and J2EE by rewriting the code from one to other. Migration for bridging .NET and J2EE involves more time and cost. Portability and reusability have been two main goals of software engineering since 1960s [27]. The key objective of application portability is to be able to deploy an application on different platforms. It allows an application to be used on a different platform or to be reused when the platform on which it is deployed is upgraded. Interoperability enables communication, data exchange, or program execution among various systems [97] in a way that requires the user to have little or no awareness of the underlying operations of those systems.

The *migration* is expensive in terms of time and money. Maintenance is complicated one, in migration. There are distinct advantages to interoperating with existing code, rather than migrating it [86]. Interoperability allows us to preserve the investment that already made in developing and stabilizing the code, familiarizing developers with it, and learning how to deploy and operate the code safely and effectively. In .NET and Java interoperable environment, even though *portability* is guaranteed, it requires that all software is written in Java [91]. Interoperability, on the other
hand, guarantees the use or access of other software regardless of what language it is coded in and what platform it is deployed on.

Even though middleware is available for assembling modules from disparate sources into a single application, it often does not form a complete bridge between components and may be inflexible as the application evolves. What is needed is the explicit design information that will forecast a more accurate, evolvable, and less costly integration solution implementation [21]. Thus in this work, the interoperability is used as a basis for integrating platforms J2EE and .NET.

4.3 INTEROPERABILITY

Interoperability is a quite familiar and ambiguous concept in computer science. Normally we look at it in two aspects—hardware and software. The interoperability problem associated with software is more complex and difficult to assure than hardware. With the popularization of Internet and distributed computing, all kinds of information with different format and structure in computer system are required to be integrated seamlessly and transparently. This requirement raises the issue of interoperability [69].

Many organizations already operate large enterprise environments based on either Java or Microsoft .NET. Hence if the organization perceives a need for a new application or addition to their current architecture, the automatic tendency is to start thinking in terms of the currently implemented environment [81]. Interoperability is a key requirement for many enterprises, allowing internal systems to work together and enabling businesses to connect to customers, external partners, and suppliers [45]. With ever increasing
requirements for efficiency, responsiveness, and cost cutting, interoperability is in great demand among modern IT developers.

Interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment or the model abstraction [51] [99]. Interoperability is one of the major challenges, particularly within component based software development environments. There are many aspects related to component interoperability, including syntactic agreements on method names, behavioral specifications of components, service access protocols, business domain knowledge and shared ontology, negotiation of Quality of Service and other non-functional properties.

By having interoperability, one vendor's implementation can be communicated seamlessly with other vendor implementation. J2EE simply does not recognize interoperability between implementations as a key requirement [42]. The challenge with this is that while imagining that companies wishing to integrate their supply chain, customer relationships, resource planning and other essential business processes will all use a single development platform J2EE or .NET. Even though it is a nice thought, reality teaches us that no one platform or operating system will be sufficient to meet the above needs of the customer. Interoperability between multiple platforms is vital.

Cross Language Independency or Language Interoperability is important for many reasons as highlighted in Reference [62]. The way that the industry moves forward and innovation happens is by new programming languages being created it would not be possible to stop but encourage them. Now to bring all of the languages under one umbrella, Language Interoperability is the only solution. Language Interoperability means
reusability of components in other sense. This reusability of components will save cost enormously. Language Interoperability means, one programming language, somehow uses something written in another language. Many different programming languages exist, so if a programming language supports language interoperability in a seamless way program development using that language is easier. If the support for language interoperability is clumsy or not present at all it makes program development harder.

4.4 INTEROPERABILITY BETWEEN COMPONENTS

As e-businesses discovered that distributed object applications were an efficient way to deliver information services, interoperability—how to get objects in different languages and operating system platforms to communicate with each other—became an immediate and overwhelming challenge [28].

Interoperability between components is one of the essential issues, since it enables the composition of reusable heterogeneous components developed by different people, at different times, and possibly with different uses in mind. Currently most object and component platforms, such as Common Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), or Enterprise Java Beans (EJB) provide the basic infrastructure for component interoperability at the lower levels, i.e., they sort out most of the “plumbing” issues. However, interoperability goes far beyond that; it also involves behavioral compatibility, protocol compliance and agreements on the business rules. For developing complex business applications, now the focus is on assembling components which are available on a local area network or on the net. These components must be localized and identified in terms of available services and communication protocol before making any request.
Interoperability Research Goals

In practice, given a collection of components to be composed, achieving interoperability consists of the following general steps:

1. Defining or selecting appropriate protocols and interfaces.
2. Implementing or adapting systems to employ the protocols and interfaces. This may require designing and implementing new systems, modifying or extending an existing system, or adding a new component.
3. Integrating and testing the interoperating components, may be done incrementally and requires refining of step 1 or 2, whenever mismatches between the systems are discovered.

Interoperability is not a problem; it’s a software quality [24]. The problem in achieving this quality is explained in the following section.

4.5 PROBLEMS AND CHALLENGES IN INTEROPERABILITY

Following are the problems in Interoperability in Component based Systems [58]:

1. COTS software is usually delivered as black box components with limited specification making it difficult to predict how the components behave under different conditions.
2. There is a general lack of methods for mapping user requirements to component based architecture.
3. Components are packaged and delivered in many different forms (Example: function libraries, off-the shelf applications and frameworks).

4. Component framework offer varying features (example: component granularity, tailorability, platform support, distributed system support and interoperability).

5. Most component integration processes suffer from inflexibility by lack of component evaluation schemes. This problem is often compounded by lack of interoperability standards between component frameworks and adequate vendor support.

6. Generally majority of COTS software are not tailorable or “plug and play”. Significant effort may be required to build wrappers and the “glue” between components in order to evolve the applications or tailor components to new situations. As the system evolves these wrappers must be maintained.

If software components are written in different programming languages, communication between the components can become problematic. The study of multilanguage interoperability attempts to bridge the communication barriers that arise between software components due to language differences.

A significant shortcoming of such approach is their lack of seamlessness, or transparency of interoperability. Components may require heavy modification in order to interoperate via these approaches; and after integration, the components may be riddled with interoperability-based code, adversely affecting their readability, maintenance, and cohesion [20].
Whenever we intend to exchange data between different systems, we may have to confront the following challenges [61]:

- **Primitive data type mappings**: Even though the same data type may exist in both languages, it cannot be guaranteed that they will map exactly. This is especially true with floating point numbers and strings.
- **Non-existent data types**: It is possible that a data type in one language does not exist in the other. Typical examples are the specialized data types that represent collections of elements, such as vectors.
- **Complex data types**: Complex data types that are composed of other data types have to be exposed to the other party so that the proper mapping can be created. Extensive testing must be carried out to assure that complex data types are not creating any problem.

4.6 INTEROPERABILITY STANDARDS

Without standards there will be no interoperability. A major reason for difficulty with interoperability in the past was a lack of agreed upon and adopted standards for such interoperation. Now there are two well-defined and mature standards in existence that can help alleviate interoperability pains. They are the Internet Inter-Orb Protocol (IIOP) and The Simple Object Access Protocol (SOAP) [91].

- **The Internet Inter-Orb Protocol (IIOP)**

  Recognizing the need for interoperability, the Object Management Group (OMG) defined the Common Object Request Broker Architecture
CORBA) in the early 1990s. The OMG is a consortium of more than 700 companies, including Microsoft. As part of the CORBA specification, the OMG defined the protocol requirements formally known as the General Inter-ORB Protocol (GIOP).

Unfortunately, CORBA implementations from different vendors had difficulty in implementing interoperable applications. A few years later, in an attempt to guarantee interoperability between different CORBA implementations, the OMG defined IIOP, which is a concrete realization of the GIOP specification. In other words, IIOP is an implementation of the GIOP specification over TCP/IP. All CORBA 2.0-compliant object request brokers (ORBs) must support IIOP, so that they are interoperable.

- **The Simple Object Access Protocol (SOAP)**

Recognizing the need for a better solution to the interoperability problem, Microsoft, IBM, and Develop Mentor came together in the late 1990s to create the first version of SOAP. SOAP was originally an acronym for simple Object Access Protocol. SOAP 1.1 is the messaging protocol used by J2EE web services and is the defacto standard for web services [76]. The following points are noteworthy about SOAP:

- Whereas IIOP, ORPC, and Java Remote Method Protocol (JRMP) are binary protocols, SOAP is a text-based protocol that uses XML. Using XML for data encoding gives SOAP some unique capabilities. For example, debugging applications based on SOAP is much easier because reading XML is easier than reading a binary stream of raw numbers. And because all the information in SOAP is
in text form, SOAP is much more firewall-friendly than IIOP, ORPC, or JRMP (used in Java Remote Method Invocation, or RMI).

- Because it is based on a vendor-agnostic technology, namely XML (and HTTP), SOAP appeals to all vendors including Microsoft, Sun, IBM, and others.
- Recognizing the fact that SOAP is text-based and thus by definition verbose, communications using binary protocols such as IIOP will in almost all cases outperform those that use SOAP as the underlying protocol.

Unlike IIOP, which represents message data in a binary format called the Common Data Representation (CDR), SOAP uses XML for its transfer syntax [65]. Transfer syntax is the format that a protocol uses to transfer data in a message from sender to receiver. Getting data from sender to receiver requires marshaling, and it is well accepted that marshaling overhead can degrade middleware performance and scalability significantly [30]. The marshaling required for CDR is of medium complexity and SOAP marshaling is far more complicated, due to its use of XML as its transfer syntax, which is time consuming to parse. Even though the popularity of XML is driving rapid improvements in XML parsing engines, the complexity of SOAP's transfer syntax versus CDR means that SOAP will always be more expensive to marshal and demarshal.

In addition to the above, IIOP is better than SOAP due to the following [48]:

- Transparent reuse of existing servers
- Tight coupling
- Object-level granularity
- Efficiency
4.7 INTEROPERATION BETWEEN J2EE AND .NET

Today there are three main component architectures competing within the software development market [83]:

- **Microsoft .NET** - This new standard development platform is on top of the Microsoft Windows operating systems. Windows is currently dominating the client desktop environment worldwide.

- **CORBA** - A well established, non-proprietary, independent integration platform. CORBA has a long track record for integrating legacy systems.


Despite the popular view that CORBA and J2EE (specifically, EJB’s) are competing against one another, both can smoothly interoperate because of EJB's adaptation of the IIOP protocol. The real gap exists between the vendor independent server-side technologies (CORBA and EJB) on the one side, and Microsoft's .NET on the other side. There are number of ways to implement J2EE and .NET interoperability:

- Web services.
- Runtime bridges such as Borland’s Janeva, Intrinsyc’s J-Integra and JNBridge’s JNBridgePro.
- Message-oriented middleware such as IBM MQseries, Microsoft Message Queue (MSMQ), BEA MessageQ, and Tibco Enterprise Message Server.
- A shared database
- Integration brokers such as IBM MQSeries Integrator, Mercator Commerce Broker, Microsoft BizTalk Server, and webMethods Enterprise Services Platform.

Although the current trend for achieving interoperability is through Web Services, they also have their share of limitations. Web Services are great when it comes to integrate the heterogeneous loosely coupled systems, but they have no support for remote object references. In practice, they are stateless and closer to a remote method call than to a distributed object system. Furthermore, SOAP and XML are by no means a compressed format and tend to be quite verbose.

- IIOP is a binary protocol and in most cases it outperforms on communications front comparing to SOAP [91].
- SOAP is mostly used over HTTP, which is a connectionless protocol. IIOP is connection-oriented and allows a richer interaction model that uses stateful services and additional protocol-based services such as load balancing and fault tolerance.

Thus IIOP is better than SOAP for interoperability. In this work, IIOP .NET is used, which is a channel based on IIOP.

4.8 IIOP .NET

As Windows and Java enterprise solutions continue to grow, companies increasingly emphasize interoperability to extend the existing software infrastructure and application investments. For application
architects, the challenge is to find strategies to integrate and extend existing deployments without ‘rip and replace’ options or costly application rewrites.

IIOP .NET is a .NET remoting channel based on the IIOP protocol, the same used by Java's RMI/IIOP. IIOP is part of the CORBA standard. IIOP .NET acts as an ORB (a CORBA object request broker): it makes objects defined in .NET application accessible to other remote ORBs, and vice-versa [66]. Java RMI/IIOP implements a subset of the CORBA type system (due to some limitations in Java's type system) and roughly provides the same features as IIOP .NET for the J2EE platform.

![Diagram of IIOP-based distributed object system]

**Figure 4.1 Overview of an IIOP-based distributed object system**

Using IIOP .NET is almost as simple as using .NET built-in remoting.

Number of tools had been developed for providing interoperability between J2EE and .NET component frameworks. The familiar and efficient tools like j-Integra and Janeva support IIOP, IIOP is better than SOAP for interoperability. So an automated tool called PIS is designed based on IIOP .NET, which is a .NET remoting channel based on IIOP protocol, and explained in Chapter 6.
4.9 CONCLUSION

This chapter dealt with the meaning of interoperability and the issues associated with the existing interoperable solutions. Further it demonstrated the need for evolving new interoperable solution to take away interoperability pains. Description of the interoperability technologies such as Web Services, J-integra and Janeva are made in the following chapter.