Part I: Introduction

Today, the concept of software reuse has been widely accepted by the software industry. In the past, many experts had made a lot of efforts on study of software component reuse for projects, product lines, and organizations. Since software components are the parts for building software systems, creating highly reusable components is one of the important keys to the success of component-based software development. With the increase of available software components in today's commercial market, many software vendors have begun to develop large, complicated application systems using the component-based software engineering approach to reduce development cost and time. This trend drives a strong demand for component-based software engineering (CBSE) methodology, standards, and guidelines to help engineers and managers to conduct analysis, design, testing, and maintenance of software components and component-based programs.

Before discussing the basic problems and solutions in testing software components and component-based programs, we need to cover the fundamental concepts of components and component-based software. Part I serves this purpose. Chapter 1 provides the necessary background and concepts of software components, including definition, elements, properties, as well as differences between software components and traditional modules. Chapter 2 introduces software component testing, in terms of objectives, process, issues, challenges, and concerns. Chapter 3 helps readers learn the essentials of component-based software, including a development process and infrastructures. Chapter 4 presents a tutorial overview of testing component-based software, including basics, challenges and issues. Finally, Chapter 5 discusses the concepts of software testability for components and component-based programs.
Chapter 1 - Introduction to Software Components

In software industry, the concept of software components has been used for a long time to refer to the building parts of software programs. However, when we ask people what a software component is, we will get different answers because the software component concept has been in a vogue for some time. People have different understanding about software components, including component definition, elements, and properties. In this chapter, we introduce the concept of software component in component-based software engineering in comparison with traditional software modules.

We first review the evolution of software components from subroutine to reusable software components. Next, we discuss the basic concepts of components, including definition, basic elements, properties, and deliverables. Moreover, we examine the differences between software components in CBSE and software modules in traditional software development. Finally, we provide an overview of an engineering process for software components.

1.1 The Evolution of Software Components

In the early days of programming, programmers developed subroutines and constructed software systems based on these subroutines. To reduce programming efforts, subroutines were reused by programmers in a project team for a specific project during implementation [1]. This reuse, therefore, is the easiest practice of software reuse. In 1968, M. D. Mcilory introduced the concept of software components at NATO Software Engineering Conferences [2]. Since then, the development history of software component technology has gone through a number of stages.
From late 1970s to 1980s, engineers used structure-oriented software development methodology to partition application systems into a number of modules (or subsystems) based on functional requirements. After constructing modules (or subsystems), they integrated them to form application systems. During that time period, a lot of function libraries written in Fortran were developed as reusable software packages to support the development of scientific applications. These software packages could be considered as the first generation of software components. Programmers used these packages to develop scientific application programs. Although they understood the importance of component reuse, they did not know how to practice cost-effective software reuse due to a lack of disciplined component-based software engineering methodology.

In 1980s, the introduction of the object-oriented technology enabled software reuse in a broader scope, including the reuse of class analysis, design and implementation. Many object-oriented C++ class libraries were developed as reusable software packages [3]. Thus, object-oriented technology steered the evolution of component technology from *reusable function library* to *object-oriented class library*. In 1990s, many large corporations, such as GTE, IBM, Lucent Technologies, and HP launched enterprise-oriented software reuse projects to develop domain-specific business components for their product lines using the object-oriented technology [4].

In 1989, component technology advanced to its new era – *reusable middleware*, when the Object Management Group (OMG) began to standardize an open middleware specification to support distributed applications [5] and developed Common Object Request Broker Architecture (CORBA). Its major goal was to allow software
middleware vendors (like IONA) to provide common reusable middleware to support
distributed objects of application software interacting with each other over the network
without concerns of object location, programming language, operating system,
communication protocols, interconnections, and hardware platform. To provide higher-
level reusable components, the OMG also specified a set of CORBA Object Services
that define standard interfaces to access common distribution services, such as naming,
trading, and event notification.

Ten years later, with the wide deployment of commercial middleware components
in the real world, the evolution of component technology reached to its new stage –
*reusable application framework and platform*. In 1999, Sun Microsystems published
Enterprise JavaBeans (EJB) [7] as a software component model for developing and
deploying enterprise-wide business applications. The model provides component
producers, customers, end-users, and component model producers with an open
specification that can be used to develop application software systems employing
software components. EJB technology is designed to provide application developers
with a robust, distributed environment that is responsible for many complex features of
distributed computing, such as transaction management and multi-threading.

Meanwhile, as the latest incarnation of Component Object Model (COM) on the PC
platform, Microsoft’s COM+ [6] is the cornerstone of a framework of technologies
designed to support the development of large-scale distributed applications on Window
platform. Many experts believe that COM+ makes the development of scalable
Windows 2000 applications easier because COM+ runtime environment provides a
foundation for building scalable distributed enterprise applications.
Recently, several well-defined component-based development methods were published to support the development of component-based software systems. Cheesman and Daniel’s UML Components [8], D’Souza and Wills’ Catalysis [9], and Herzum and Sims’ Business Component Factory [10] are typical examples. These component-based methodologies provide engineers with well-defined processes, analysis and specification models, and engineering guidelines.

1.2 Why Is Software Component Reuse Important?

Although the concept of software component reuse was introduced in 1968, it has not been received much attention in the software industry and the academia until late 1980s. After mid 1980s, the fast increasing complexity of software application systems forced the software industry to look for cost-effective methods to construct complicated software systems in a short development life cycle to meet marketing needs. Software reuse based on quality components, therefore, has become popular in the academia and industry due to its advantage of reduced development cost and time. Companies, such as IBM, Lucent Technology, and GTE [4] launched enterprise-oriented software reuse projects to develop their in-house reusable component library. The major purpose was to speed up their production lines of application systems based on in-house reusable components, so they can keep pace with market forces and increase their competitive edge in a business market with dynamic needs and fast updated technology.

Since 1995, information technology and service market has grown rapidly due to the explosive popularity of the Internet and fast technology advances [11]. For example,
according to the Japan Information Service industry Association (JISA), the Japanese
information service market exceeded ten trillion yen (US$100 billion) in 1999. This
market change created a strong demand for new large-scale information service systems
and e-commerce applications over the Internet. To fulfill this demand, software market
and industry has to change in order to deliver large web-based information service
systems and e-commerce applications on a very tight schedule based on off-the-shelf
components. For instance, the Japanese software market structure has been changed from
custom software to shrink-wrapped package software and component-based software
[11]. Meanwhile, the Japanese software industry structure has also been changed from a
hierarchical structure to an open and competitive structure.

The tragic event of September 11, 2001 destroyed and damaged many business
application systems and infrastructures. This has driven a strong demand for setting up
new secured application systems and information services to replace out-of-date systems
in businesses and governments. Since most of them are global distributed information
service systems with complex structure, multi-platform, and strong requirements on
system service in reliability and scalability, it is very difficult or impossible for them to
develop new IT-based application systems and service infrastructures without using off-
the-shelf components. To reduce the development cost and time, they must use the
component-based software engineering approach to develop the new systems or to
upgrade the existing systems based on high quality reusable components.

1.3 What Is A Software Component?
What is a software component? You may get different definitions and answers from engineers, managers, testers, and researchers. One of the reasons is that the modern concept of software component is different from their traditional counterparts, although both capture and define the concept of "parts" of a software system. Not long ago, we were used to considering software modules as the default notion of software components. In recent years, as the component-based software engineering advances, the term of software component has been defined by a number of experts in different ways. This further contributed to misunderstandings and confusions about the notion of software component among engineers and managers.

Now let us review some of the important definitions of software components defined by the experts. One of the earliest definitions is given by Gready Booch [12].

A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.

This captures the idea that a reusable component is an encapsulated software module consisting of closely related object-oriented elements. Later, Clement Szyperski presented his well-known definition of software component in the 1996 European Conference on Object-Oriented Programming. It is stated below [13]:

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 third parties.

This definition is well accepted in the component-based software engineering community because it highlights the major properties of software components which do
not exist in traditional software modules, such as context independence, composite, deployable, and contracted interfaces. In 2000, a more broader, general notion of a component is defined by Alan W. Brown [14]:

*An independently deliverable piece of functionality providing access to its services through interfaces.*

The similar idea is also expressed in the UML's definition [15] below:

*A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.*

Recently, Bill Councill and George T. Heineman [16] gave a new definition to emphasize the importance of a consistent component model and composition standard in building components and component-based software.

*A software component is a software element that confirms to a component model and can be independently deployed and composed without modification according to a composition standard.*

According to them, a component model defines proper ways to construct software components, and support their interactions, composition and assembly. In addition, a component model also defines the mechanisms for component customization, packaging, and deployment.

To simplify the concept of software components for our readers, we classify the notion of software components into three types: a) *reusable modules*, b) *reusable components*, and c) *composite building blocks*. They will be used in all chapters of this book. The detailed definitions are given below.
A reusable module is an independent and deliverable software part that encapsulates a functional specification and implementation for reuse by a third party.

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.

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.

Clearly, the concept of reusable modules given here comes from the traditional reuse idea, where functional module reuse is the major focus. It is a very useful concept for engineers and testers who deal with reusable software modules generated in a traditional software engineering process. The concept of reusable components given above emphasizes the quality production of reusable software parts with contracted interfaces. Since the component concept reflects the current adoption status of component technology in the real world, this notation is very useful for engineers, testers and managers who are involved with component-based software development projects. The given notion of composite building blocks not only focuses on component reuse but also emphasizes on component composition. The idea of building blocks, therefore, is very important for component developers and testers in a component vendor because it represents the current status and future trend of component technology.

\[1\] A component model defines the ways to construct components, and regulates the ways to integrate and assembly components.
1.4 Properties of Software Components in CBSE

Component properties refer to the essential characteristics of software components. In component-based software engineering, component properties can be classified into two groups: basic properties and advanced properties. Basic properties refer to the necessary features of a software component. Advanced properties refer to the distinct features offered in modern software components.

A software component has the following basic properties.

- **Identity** – Each component must be uniquely identifiable in its development environment as well as in its target deployment environment. Without this feature, large scale of component reuse is impossible. Current component technology ensures this by using a well-defined (or standard) naming scheme. CORBA, EJB and Microsoft DCOM are typical examples.

- **Modularity and encapsulation** - Software components results from partitioning of a software system. Each component encapsulates a set of closely related data elements and coherent functional logic to perform a specific task.

- **Independent deliverability** – Software components must be delivered as independent parts that can be replaced under certain conditions. Each part must play an independent functional role in a system, and support a specific task in a targeted operation environment. In traditional software development, the major deliverable of a reusable module is its program code. Today, in component-based software engineering, the deliverables of software components include well-defined
specification, interface document, user reference manual, source code, and executable code.

- **Contract-based interfaces** - An interface between software components can be viewed as a contract between a client of an interface and a provider of an implementation for the interface [13]. Each contract specifies what the client must do to use the interface to access a component service function. It also defines what kind of service and its implementation has to be provided by the provider to meet the service contract. This suggests that a component deliver its provided services only when its clients access the interface in a right way.

- **High quality** - Component quality is critical to component-based software because of the component reuse across different projects, product lines, and organizations. In the conventional software development, no dedicated quality control process is dedicated for assuring the quality of reusable modules. Quality checking for a software module is based on a traditional function validation process, including component unit testing and integration testing. Since defects in a reusable component could wreak havoc on all those software systems employing the component, today most component vendors use a rigorous quality control process to verify and validate the quality of constructed components. Effective reuse of high quality components not only reduces software development time but also increases software product quality.

- **Reusability** – Reusability of software components is the key to the success of component-based software engineering. Unlike traditional software modules, which typically have a very limited reuse scope, today’s software components provide
multiple-level granularities for reuse in a large scope. The reusable elements of a software component include its analysis specification, design and design patterns, source code, and executables. In addition, component deployment mechanisms and test support information (such as test cases and test scripts) are reusable items too. These artifacts can be reused across different organizations, projects, product-lines, or even by the public.

Modern software components, such as reusable components and building blocks, not only demonstrate these basic properties, but also possess the following distinct advanced properties.

- **Customizability and packaging** – This refers to the customization capability and packaging function which are built inside software components based on a well-defined approach. This feature ensures that software components can be customized and packaged to meet the requirements of a target application system. This feature extends the scope of component reusability. Software components with this property are known as *customizable and packageable components* because they can be configured and packaged by clients to select its meta-data, functional features, and interfaces.

- **Deployability** – Software components are known as deployable if a well-defined deployment approach has been built inside a component to support its deployment after they are customized, packaged, and installed in an operation environment. The deployment approach usually supports the creation of an executable instance of components and its necessary interactions with it.
• Multiple instances and polymorphism – A software component may have more than one deployed instance in an operational environment. Since all of the deployed instances implement the same component, they must share some common features and interfaces. However, they should be viewed as different instances because of their different feature customization, packaging selection, operational data, execution tasking or threading, and dynamic status in an operational environment.

• Interoperability – Component interoperability refers to the component capability that supports communication and data exchange between components. Components are interoperable if they are developed based on standard communication channels and data exchange mechanisms defined in a component model. There are two types of component interoperability, local and remote. Local interoperability refers to component interoperability for components in a host-centered environment. However, remote interoperability refers to component interoperability between components over a network. Remote interoperability is based on remote method calls (RMCs), an extension of the concept of remote procedure calls (RPCs) that supports remote communications and data exchanges between components over a network.

• Composition – Components are composite if they are developed based on the standard composition methods to support component composition. In other words, the composite aggregation relationship between components is supported in an operational environment so that a component instance is able to create and destroy its aggregated parts (which could be other components). Supporting this composition relationship needs a uniform identification mechanism for composite parts so that the
transitive nature\textsuperscript{2} of the composition relationship between components can be easily managed. Since components are the building parts for the development of component-based software, their composition property is very important to support component composition.

- Component model conformity – This means that component realization follows a component model and its related standards to make sure that the component model standards and consistent mechanisms are used to support component interactions, composition, packaging, and deployment. In addition, before shipping to customers, a dedicated component quality control process must be in place to check all component deliverables based on the given requirements and pre-defined component standards.

1.5 Software Modules VS. Software Components In CBSE

What are the major differences between traditional software modules and software components in component-based software engineering? The previous sections discussed the major differences between modules and reusable components in terms of concepts, elements, deliverables, and properties. Let us now compare them from development perspectives. Table 1 summarizes the major differences.

In traditional software development, modules are developed by programmers (or engineers) to fulfill a specific functional part of a system. The major objective is to generate high quality modules with good modularity and well-structured function logic to make sure that they are easy to use, integrate, and maintain. Since software module development is usually conducted through a white-box construction process that focuses

\textsuperscript{2} Composition is transitive means that if C is a part of A, and E is a part of C, the E is also a part of A.
more on module functionality instead of being concerned with standard module interaction and composition, the constructed modules usually lack interoperability, standard interfaces, common packaging and deployment approaches. Moreover, conventional software modules (true even today) have been developed for a specific operation environment, and this further limits their reusability.

<table>
<thead>
<tr>
<th>Perspectives</th>
<th>Software components in CBSE</th>
<th>Traditional software modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development purpose</td>
<td>Build high quality software parts that can be reused by third parties for a long term.</td>
<td>Construct quality functional parts for a specific project</td>
</tr>
<tr>
<td>Development objective</td>
<td>Develop high quality software parts with standard contract-based interfaces according to a standard component model.</td>
<td>Develop high quality modules with good modularity and well-structured logic</td>
</tr>
<tr>
<td>Creators</td>
<td>Component development engineers</td>
<td>Programmers or software engineers</td>
</tr>
<tr>
<td>Users</td>
<td>Application system developers and integration engineers</td>
<td>Application programmers</td>
</tr>
<tr>
<td>Development process</td>
<td>A dedicated component engineering process</td>
<td>No dedicated component development process</td>
</tr>
<tr>
<td>Design and analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Think in terms of reusable components.</td>
<td>• Think in terms of functions, or sub-functions</td>
<td></td>
</tr>
<tr>
<td>• Focus on black-box and interfaces</td>
<td>• Focus on white-box design and coding</td>
<td></td>
</tr>
<tr>
<td>• Conduct domain analysis and modeling</td>
<td>• Use an ad hoc domain model</td>
<td></td>
</tr>
<tr>
<td>• Follow a component model.</td>
<td>• Apply structure-oriented or object-oriented analysis and design methodology</td>
<td></td>
</tr>
<tr>
<td>• Apply UML or a similar methodology for analysis &amp; design.</td>
<td>• Based on a specific technology under an implicit environment.</td>
<td></td>
</tr>
<tr>
<td>• Use design patterns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Implement the design in variety of explicit context environments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Unique identification in an operational environment</td>
<td>• Unique identification in a particular application program or system</td>
<td></td>
</tr>
<tr>
<td>• Contract-based interfaces specified in a standard format</td>
<td>• Independent program unit and compiled unit</td>
<td></td>
</tr>
<tr>
<td>• Modular and encapsulation of functional logic and data</td>
<td>• Customer-built interfaces specified in an ad hoc format</td>
<td></td>
</tr>
<tr>
<td>• Reusability</td>
<td>• Functional modularity</td>
<td></td>
</tr>
<tr>
<td>• Packaging and deployability</td>
<td>• Implicit dependency on specific technology and a operational system environment</td>
<td></td>
</tr>
<tr>
<td>• Interoperability* (for building blocks)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Composition* (for building blocks)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Component model conformity* (for building blocks)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standards</td>
<td>A component model and standards which stipulate consistent component interaction mechanisms, and standard packaging and deployment approaches.</td>
<td>No component model and consistent approaches to support component interoperation, packaging, and deployment.</td>
</tr>
<tr>
<td>Reuse granularity</td>
<td>Component abstracts, reusable function/class libraries, packages or modules, frameworks, platforms, application servers</td>
<td>Procedures/subroutines, classes, modules (or a group of classes or subroutines), or subsystems</td>
</tr>
<tr>
<td>Reuse scopes</td>
<td>Large reuse scopes encompassing different projects, product lines, organizations, or even the programming endeavors of the general public.</td>
<td>Limited reuse scope for a specific project, product line, or an organization.</td>
</tr>
<tr>
<td>Deliverables</td>
<td>Component specification, design, source code, executables, interfaces, user reference manual, quality control metrics and report. (Note: component design and source code are not delivered to clients.)</td>
<td>Module specification, design, source code and executables, as well as user reference manual.</td>
</tr>
<tr>
<td>Quality Control</td>
<td>A rigorous quality control process in place to control the quality of reusable software components.</td>
<td>Only system quality control process is used. No rigorous quality control process for reusable software modules.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only system quality control process used; no rigorous quality control process employed for assuring the quality of individual reusable software modules.</td>
</tr>
</tbody>
</table>
In component-based software engineering, software components are developed by component engineers as highly reusable parts for the third-party usage across product lines and organizations. The major objective is to create reusable and building blocks with contract-based interfaces to make sure that they are highly reusable, interoperable, easily packaged, and deployable. Component development is a reuse-centered development process focusing on component interaction and composition. Component engineers perform this process based on a component model, the companion component standards, and a consistent approach to support component interactions, customization, packaging, and deployment. Before the component development process begins, component domain analysis and modeling is performed first. During component development, engineers use the component-oriented unified modeling language (UML) to define components by specifying component use cases, object-oriented structures, and dynamic behaviors. Unlike traditional module development where module quality control is conducted in an ad hoc manner, a component development process in CBSE uses a well-defined component quality control process to verify and control component quality based on a component model and its companion component standards. Moreover, a component development process could produce reusable parts in diverse granularity, including reusable components, libraries, packages, frameworks, platforms, and application servers.

What are the differences between software classes and software components? A class in a program is a type of class objects. Each class is a program unit that encapsulates a set of functions and data attributes. Hence, it is a programming concept.
However, in the component world, a class could be viewed as a reusable component without deployment support because it possesses the common properties of a software component, including identity, encapsulation, reuse, polymorphism, and composition. However, classes are not composite building blocks because no standard models and mechanisms are involved to support class interactions, packaging, and deployment, although class templates allow programmers to customize classes in meta-data types and functions.

1.6 Engineering Process For Software Components

A well-defined software engineering process plays an important role in software project management. It enables project managers and engineers to control and manage software projects in a systematic manner. Similarly, building software components needs a well-defined component engineering process to control component development activities. A software component development process involves six groups of people.

- Component engineers, who perform component analysis, design, and implementation according to a given component model and its companion standards.
- Technical managers, who are in charge of component development projects by managing and coordinating different project activities.
- Component test engineers, who perform functional validation and performance evaluation for generated software components.
- Quality assurance personnel, who define component test plan, quality control process and standards, and quality evaluation metrics for software components. Meanwhile, they also control the component quality process to ensure the conformity of all constructed components to a pre-defined or selected component model.
• Component technical supporters, who are assigned to perform software maintenance tasks for components after they have been shipped to customers.
• Component technical writers, who are responsible for user documentation.

Figure 1.1 An Engineering Process for Software Components

Figure 1.1 shows a component engineering process for software components. It is an iterative process consisting of the following seven phases.

• Analysis – In the component analysis phase, all component requirements (including functional/non-functional requirements, data and objects, user cases, and interfaces) are collected, modeled, and specified based on a well-defined methodology, such as unified modeling language (UML)[17]. The result of this phase is a component specification document.

• Design – In this phase, engineers conduct component design based on component analysis results from the previous phase. In component-based software engineering, component design includes three tasks. The first task is to conduct the design for functional logic and data objects, define related algorithms, and make trade-off decisions on technology and operation environment. The next task is to follow a selected component model, and work on component realization by providing data exchange mechanisms for component communication. The last is to define consistent approaches to support component packaging, and deployment. The outcome of this phase is a component design document.

• Coding – This is the component implementation phase in which a specific technology and programming language are used to implement the component design within
specific operational environments. It is common to see a component implemented for more than one implementation. Each implementation depends on a specific technology set and an operational environment. For a customizable component, multiple implementations may be provided, for the customer to choose from, to support the same functional feature but with different data structures, algorithms, and business logic.

- **Testing** – The major task of this phase is to validate a generated software component based on the given component specification and design. During this phase, component testers perform white-box and black-box testing to uncover component design and implementation errors. Since software components are the final products, other types of testing must be conducted, such as component usage testing, and component performance testing. The result of this phase is a component test plan, test design document, test bed, and test report.

- **Packaging and deployment** – This phase is required only when a component provides packaging and deployment capabilities. Testers and component quality assurance personnel must execute and validate different packaging facilities, customization features, and deployment approaches to check if they work fine under each configuration setting in every specific operational environment.

- **Maintenance** – This phase begins after shipping first version of a software component. During this phase, the component is enhanced according to customer requests, and bugs are fixed based on customer reports.
S. D. Lee at al [17] present an UML-based component engineering methodology. They discuss the detailed tasks and modeling methods involved in component analysis and design. In modern component technology, software component construction must follow a well-defined component model and standards. To achieve this, a component reference model (see Figure 1.1) is needed to support the engineering activities in all phases. A well-defined quality assurance process is required to control and monitor the quality components generated from a component development process. Information about component quality control can be found in Part II of this book.

### 1.7 Basic Elements of Software Components

What are the basic elements of a software component? What are the deliverables of a software component development process? Here we discuss the answers to these questions based on a comparative view of reusable modules, components, and composite building blocks.

A **reusable software module** usually includes five basic elements. The first is module specification, which presents the module's function at an abstract level. The second is module interfaces, which specify how the module interacts with others. The third is module design, which records the design decisions on internal class objects and functional logic. The fourth refers to module implementation, which includes the generated source code and executable code based on an implicitly specified operation environment, including computer hardware platform and operating system. The last is module validation, which refers to test harnesses and testing facilities, such as test drivers, and test stubs. Figure 1.2(a) shows all deliverables from software module
development. Among them, only the first two items are visible to its clients unless its source code is designed for reuse.

Similar to a reusable module, a **reusable component** also has its five elements: specification, design, implementation, interfaces, and validation support. However, there are two differences. The first difference is about the interfaces. A module has function-oriented interfaces, while a component has contract-based interfaces. Component interfaces can be classified into two types: a) *import interfaces*, through which the component obtain services from other components, and b) *supply interfaces*, through which other components could obtain services from the component. The other difference is related to operation environment. Module design and implementation frequently depends on an implicit operation environment and technology. However, component design and implementation must always be performed based on explicit operation environments and technology.

As shown in Figure 1.2(b), a reusable component has two extra elements than a reusable module. The first is the *deployment support* that refers to a deployment facility, such as an installation and configuration package, which makes it a deployable product. The other refers to its quality control results, such as quality control documents, metrics, or a certification report. This suggests that a well-defined quality control process and standards are in place to control and monitor the quality of components.

**Figure 1.2 Component Deliverables**
As shown in Figure 1.2 (c), a composite building block contains a distinct element, known as component model realization, which refers to the selection and realization of a component model to support component customization, packaging, and deployment. This element conforms to the component development based on a well-defined component model and the concomitant standards [13], including includes the selection and realization of approaches and mechanisms for customization, packaging, and deployment. Similar to a reusable component, a composite building block also consists of its specification, design, implementation, and interfaces. If a composite building block is customizable, then for the same component specification, it may contain more than one design and implementation for the involved functional features and interfaces based on different technology and operational environments.

1.8 References


**Footnote**

[1] A component model defines the ways to construct components, and regulates the ways to integrate and assembly components.

[2] Composition is transitive means that if C is a part of A, and E is a part of C, the, E is also a part of A.