Chapter 1

Definition of a Software Component and Its Elements

Bill Councill
George T. Heineman

1.1 Introduction

The goal of this chapter is to rigorously define terms that describe the best practices of component-based software engineering (CBSE). We will develop and describe in detail the term *software component* and its constituent elements to provide clear, unambiguous, and rational meanings to the terms used to describe CBSE. You will find some terms used here for the first time. The reason for this is simple: many terms in software engineering emerged without precise definitions in publication and were subsequently used without reflection or scientific review. We avoid the use of any terms that, although popular, are not rigorously defined, are circularly referenced, or are simply

descriptions of software engineering phenomena. When nonrigorous terms are used we rely on the engineering sciences, particularly industrial and civil engineering, which require precise definitions because of the demands for safety and public welfare inherent in their disciplines.

For too long a so-called engineering discipline has not trained its practitioners in engineering practices, not subjected its students to internships and then tested their knowledge and expertise, and not prevented untrained and undisciplined employees from taking jobs of authority. Meanwhile, project managers are not engineering-trained software engineers or managers with full knowledge of the complete life cycle. We are at a crossroads. Project managers must know the intricacies of design, development, testing, and component management to successfully manage the assembly of complex components and their even more complicated maintenance. These managers must also be aware of the need for second- and third-party certification and contractual relationships.

1.1.1 Background

We first define a set of basic terms required to explain the characteristics assigned to the definitions. *Software* is constructed to execute on a general-purpose von Neumann computing device (henceforth, a *machine*). A software element contains sequences of abstract program statements that describe computations to be performed by a machine. A software element is *machine-executable* if: (1) the machine directly executes the program statements or (2) a machine-executable *interpreter* directly understands the program statements and the machine directly executes the interpreter.

The *source code* for software is the set of *machine-readable* files containing program statements written in a programming language. These statements are either compiled into *machine-executable* statements using a compiler or executed by an interpreter.

1.2 Definition

A software component simply cannot be differentiated from other software elements by the programming language used to implement

the component. The difference must be in how software components are used. Software comprises many abstract, *quality* features, that is, the degree to which a component or process meets specified requirement (IEEE Std 610.12-1990). For example, an efficient component will receive more use than a similar, inefficient component. It would be inappropriate, however, to define a software component as "an efficient unit of functionality." Elements that comprise the following definition of the term software component are described in the "Terms" sidebar.

> 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 composition standard.
>
> A *component model* defines specific interaction and composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model.
>
> A *software component infrastructure* is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

These definitions demonstrate the important relationship between a software component infrastructure, software components, and a component model.

1.2.1 Interaction Standard

One underlying concept of a component is that it has clearly defined interfaces. We pattern our definition for the term *interface* after the object composition model for Reference Model of Open Distributed Processing (RM-ODP). A joint effort of international standards bodies, the RM-ODP describes a framework and supporting infrastructure that enables the integration of distribution, internetworking, interoperability, and portability of applications ([Raymond, 1995] [ISO/ITU Open Distributed Processing—Reference Model—Part 2: Foundations, 1995b]. An *interface standard* is the mandatory requirements employed and enforced to enable software elements to directly interact with other software elements. An interface standard declares what can comprise an interface.

Terms	
Standard	An object or quality or measure serving as a basis to which others should conform, or by which the accuracy or quality of others is judged (by present-day standards). This term includes proprietary vendor and producer standards as well as national and international standards produced by recognized standards bodies.
Software element	A sequence of abstract program statements that describe computations to be performed by a machine.
Interface	An abstraction of the behavior of a component that consists of a subset of the interactions of that component together with a set of constraints describing when they may occur. The interface describes the behavior of a component that is obtained by considering only the interactions of that interface and by hiding all other interactions.
Interaction	An action between two or more software elements.
Composition	The combination of two or more software components yielding a new component behavior at a different level of abstraction. The characteristics of the new component behavior are determined by the components being combined and by the way they are combined.

A component supports a *provided interface* if the component contains an implementation of all operations defined by that interface. The interface hides the component implementation. A component needs a *required interface* if the component requests an interaction defined in that interface and the component expects some other soft-

ware element to support that interface. A component may be unable to provide an interface if one of its required interfaces is unfulfilled. A component should ideally deploy with descriptive information that completely specifies all provided and required interfaces.

Software elements interact with a component using the component's clearly defined and documented interfaces. An interaction standard defines the elements of an interface. If the component can perform its function only by interacting with other software elements, all explicit context dependencies should be clearly specified in the component's documentation. An interaction standard is actually a superset of the interface standard previously discussed. The interaction standard covers both direct and indirect interactions that may exist between components.

A component may have an *explicit context dependency* on the operating system, a software component, or some other software element. An *interaction standard* specifies the type of explicit context dependencies a component may have. Another form of explicit context dependency occurs when a component must execute on a computer with a specific clock speed to achieve its performance objective. If the component must interact with a hardware device the component uses application programming interfaces provided by the operating system or an interface provided by the component model implementation. In both cases the descriptive information for the component must clearly define the explicit context dependency. To enable reuse and interconnection of components, component producers and consumers often agree on a set of interfaces before the components are designed. These mutual agreements can lead to standardized interfaces.

1.2.2 Composition Standard

For independent deployment a component must be clearly separate from an operating system and other components. Thus, the component encapsulates the necessary data and algorithms to perform its tasks. The way in which a component is deployed is determined by the component model and typically involves three steps.

1. Installing the component in preparation for its use.

2. Configuring the component and perhaps the operating system where the component will be executed to make the component available.
3. Instantiating the component for use.

The source code for a software component is the full set of machine-readable software files (containing procedures and modules) and machine-executable files (containing run-time libraries and pre-compiled object code) required to package the software component into a machine-readable software element. A software component may be packaged in binary form to:

- Protect the proprietary intellectual property of the software component producer because it is nearly impossible to reverse-engineer the source code for a component from its binary packaged form
- Decrease installation and deployment costs
- Reduce explicit context dependencies (the consumer, for example, must have Gnu C++ compiler version 2.8.1)

The component producer will decide whether the source code should be deployed with the component. It is possible that a consumer or third-party certifier will require access to the source code.

A composition standard defines how components can be composed to create a larger structure and how a producer can substitute one component to replace another that already exists within the structure. We pattern our definition for the term *composition* in the **Terms** sidebar after RM-ODP.

In addition to an interface description, the component producer should provide sufficient descriptive documentation to enable a component consumer to assemble the component into a target application. Third-party certifiers will also use the documentation to verify the process used to develop the component and ensure that the final product fulfills the specifications. The component producer or the third-party certification organization will decide the most appropriate form for the documentation, that is, whether to store it with the component, in either source or binary format, or provide it separately. The forms of documentation generally deemed most advantageous to component consumers are

- Business rules
- Business processes
- Functional requirements
- Nonfunctional requirements
- Use case scenarios
- Design documentation using Unified Modeling Language diagrams and Object Constraint Language
  - Preconditions
  - Postconditions
  - Design contracts

1.2.3 Component Model

A component model operates on two levels. First, a component model defines how to construct an individual component. For example, Microsoft's Component Object Model (COM) requires each COM component to provide an `IUnknown` interface. Second, a component model can enforce global behavior on how a set of components in a component-based system will communicate and interact with each other. A component model enables composition by defining an interaction standard that promotes unambiguously specified interfaces. A component can be composed with another component or other software element (for example, legacy code) by creating assembled or integrated connections respectively.

The component model defines the permitted mechanisms for creating assembled or integrated connections. D'Souza and Wills (1999) observe that "plug-in compatibility" only succeeds if a component can accurately declare its expectations of the other component to which it is connected. The process of assembly may be as complicated as necessary to achieve the goal of precise specifications. We use the term *assembly* to include the many different forms in which components compose, such as wrapping, static and dynamic linking, and "plug-and-play".

The component model may define customization mechanisms that describe how components can be extended without modification. We treat customization as an advanced form of interaction. A component model may also define mandatory component properties, such as cod-

ing formats, documentation standards, or obligatory producer-independent interfaces.

1.2.4 Component Model Implementation

The component model implementation is the dedicated set of executable software elements necessary to support the execution of components within a component model. An operating system (OS) could embed the component model implementation but that would only further complicate the OS and might restrict the applicability of the component model. The component model implementation is typically a thin layer that executes on top of an OS. Multiple OSs can port this layer to ensure maximum applicability of the component model.

The interaction standard for a component model determines whether an interface must be registered with the component model implementation prior to use. Interfaces are typically defined by using an interface definition language (IDL) and registered with an interface repository associated with the component model implementation. The composition standard for a component model determines whether a component must be registered with the component model implementation prior to its use. The vendor of a component model implementation may provide tools such as an IDL compiler to support the development of components.

The component model implementation makes it possible to execute components that conform to the component model. The Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA), for example, functions in an open distributed processing system using Object Request Brokers, software applications that execute on OSs such as Microsoft Windows or UNIX. CORBA is an open standard, which means that the OMG promotes the standard but is not a component producer that provides the component model implementation. When the producer of a component model implementation is also the designer of the component model, the component model could be proprietary (that is, only one component model implementation is available).

1.2.5 Summary

Our definitions and descriptions of components, component models, component infrastructures, and component model implementations do not explain how software engineers should design and construct components. As software engineers we must only promote new technologies that provide state-of-the-art design and deployment methodologies for software systems to support immediate empiricism. Eventually Engineering scrutiny will eventually be applied to design and deployment methodologies.

One more definition must be added at this point to differentiate software development and the component life cycle. In a traditional software development life cycle developers are often analysts, designers, and developers. A project has a well-defined beginning, when requirements are elicited, and a well-defined ending, when the final software system is delivered. Component production is different. Considerably more time is devoted to business rules, business process modeling, analysis, and design. Much less time is spent in development while testing occurs throughout. Accordingly, we introduce the following definition

> The *component-based software life cycle* (CSLC) is the life cycle process for a software component with an emphasis on business rules, business process modeling, design, construction, continuous testing, deployment, evolution, and subsequent reuse and maintenance.

The analysis and design phases for a CSLC are significantly longer than during a traditional software development life cycle. At least one verification activity is conducted at the end of each CSLC phase. While recommended in various software engineering guides and standards, verification is an absolute necessity throughout the analysis and design to ensure a successful construction and unit testing phase. During development unit testing, using the "one best way," (see Chapter 37) is implemented and tested. Testing against each component will be performed separately. Software testers participate in integration and system testing cooperatively with all team members. Maintenance is designed into the component for evolution that may occur years away.

The component model implementation that supports interaction and composition must also enable engineers to create domain-specific software component infrastructures whose components interact to

realize the functionality and behavior of a desired system during the CSLC. The next section describes our vision concerning how to successfully integrate component-based technology into existing software development processes.

1.3 The Master Software Development Plan

1.3.1 Background

Software engineering literature commonly defines the term software component by describing "software component framework" or "software architecture." Indeed, it is often difficult to describe the details of a smaller part without referring to the greater whole to which it ultimately belongs. After an exhaustive literature review we discovered that authors generally use the terms *architecture* and *framework* (often without definition) interchangeably and in a variety of dissimilar ways. Both terms, however, are used solely within the software engineering community and not in other engineering disciplines.

Software architecture and frameworks are essentially synonymous with the elicitation of the highest levels of design elements for software systems. Instead of creating more confusion by trying to differentiate these terms and their complementary theories, we describe a simpler concept, that of increasingly discrete and detailed design. We start with the concept of a master software development plan borrowed from the well-established practice within industrial engineering.

1.3.2 Definition

In any meaningful engineering endeavor a lead engineer or component-based software project manager (hereafter, engineer) establishes the scope of the project, generally according to an RFP, identifies the performance specifications of the finished product, and determines how to validate the success of the requested product. When establishing the product's scope the lead engineer divides the work into *subprojects*, self-contained processes of analysis and design that produce elements that will be incorporated into the final design. The elements from all subprojects are managed by subproject engineers and custom-

ized to implement an integrated design that will satisfy the performance specifications. In CBSE and software reuse we use the engineering term *performance specification* rather than requirements specification because the term foreshadows the need for decomposing a problem into subproblems to be solved, while requirements are often global in scope and can't be reused easily.

> A *performance specification* defines the functional requirements for a product, the environment in which it must operate, and any interface and interchangeability requirements. It provides criteria for verifying compliance, but it does not state methods for achieving results. (*www.dsp.dla.mil/documents/sd-15.html*).

In the field of software engineering the master software development plan created by the lead engineer ensures the success of the project and its subprojects.

> A *master software development plan* describes the methods adopted by the lead software engineer or manager for a system's composition and interaction. It is a conceptual plan that defines the boundaries of the system, its elements, interactions, and constraints on these elements and interactions. The master software development plan consists of a global design that identifies discrete and manageable subprojects.

1.3.3 Description

The lead engineer is responsible for ensuring the success of the project by adapting existing models to fit the needs of the software under development. Traditional professional engineering practices require that the processes necessary for each project are determined by the lead engineer based upon the performance specifications of the project. While nationally or internationally prescribed organizational processes are helpful—such as the Software Engineering Institute's Capability Maturity Modeling and the International Standards Organization (ISO) 9001—the lead engineer is responsible for the project and subprojects. Engineers are also responsible for compliance with laws and regulations as well as industry standards.

Realization of a comprehensive and frequently complex master software development plan results in the design of a software component infrastructure. The component infrastructure embodies the

design decisions and trade-offs of the project. The lead engineer ensures that descriptive documentation is developed that precisely describes the interactions among components in the software component infrastructure. Through increasingly detailed design the component infrastructure is refined and a component model is selected. The final implemented components in the infrastructure will conform to this component model. The component model implementation enables the interaction and composition of the components in the software component infrastructure

The lead engineer may select an existing component model, design and implement a proprietary component model, or develop a component model implementation for an existing component model. The component model implementation will support the execution of the components in the software component infrastructure. Lead engineers do not work in isolation; they rely on the judgment of subproject engineers and the analysis and design team. Therefore, the choice is based on best practices, according to the professional assessment of the lead engineer—as influenced by engineering and design staff—to satisfy the original design goals of the desired system within existing budget and staffing constraints.

The lead engineer may choose to design a software component infrastructure that includes components that conform to different component models. In this case the lead engineer must ensure the interoperability of the respective component model implementations of these component models.

Because of the different types of performance specifications a system must satisfy (such as efficiency, use of resources, and integration with legacy data), software engineers, analysts, and designers benefit from "visualizing" any proposed software system from a variety of viewpoints. The software component infrastructure that supports the master software development plan can be analyzed from multiple viewpoints to evaluate global performance requirements. We derive our viewpoints from the RM-ODP standard (ISO/ITU Open Distributed Processing—Reference Model—Part 2: Foundations, 1995b). The RM-ODP framework is composed of five viewpoints that provide sets of concepts, structures, and rules. These viewpoints are used to specify ODP systems. They are also applicable to component-based systems

- *Enterprise* (purpose, scope, and policies) – An enterprise specification contains policies determined by the organization rather than imposed on the organization by technology choices.
- *Information* (semantics of information and information processing) – The individual components must share a common understanding of the information they communicate when they interact.
- *Computational* (functional decomposition) – The computational specification describes how the overall functionality of an application is distributed between the components.
- *Engineering* (infrastructure support) – The engineering specification describes the design that enables communication between components.
- *Technology* (choices of technology for implementation) – The technology specification describes the implementation of the system and the information required for testing.

More information is available about each of the viewpoints in the ISO/IEC *Overview* document (ISO/ITU Reference Model of Open Distributed Processing - Part 1: Overview, 1995a).

We have been influenced by the RM-ODP definitions of composition, interfaces, and viewpoints because the reference model is an international standard developed jointly by ISO and ITU-T to support the distribution, internetworking, interoperability, and portability of ODP. Although developed to provide a "big picture" to organize pieces of an object-oriented ODP system into a coherent whole, the RM-ODP is designed as an abstract standard because it carefully avoids prescribing an implementation.

Many sets of viewpoints are available for decomposing, analyzing, and distributing systems' functions and components. Incorporating a set of concepts and rules for each of the viewpoints, the RM-ODP provides a method to specify systems from particularly important software engineering viewpoints. The RM-ODP is appealing because it offers languages for analyzing and resolving the requirements of businesses for software based on these viewpoints, although the languages are object-specific. We find that CBSE, in general, has similar needs, and this book provides many chapters to help you understand its complexities.

1.4 The Big Picture

Figure 1-1 graphically demonstrates the relationships among the master software development plan and software components, software component models, software component model implementations, and software component infrastructures. The lead engineer is responsible for selecting the component model that can best implement the logical design as described by the software component infrastructure created for the master software development plan. The design elements produced by the subprojects must interact effectively and efficiently to achieve the overall project goal and software engineers must determine how to map the logical elements into physical components. The RM-ODP viewpoints can help engineers make successful decisions for the producer or consumer. Therefore, actual implementation of the system becomes an engineering endeavor.

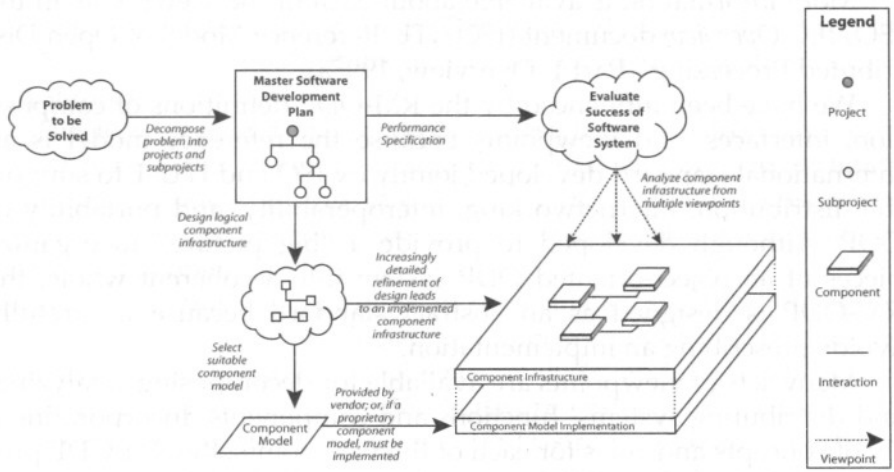

Figure 1-1: *The big picture.*

As a small example, consider a component producer that has won a competitive bid for designing and deploying a component-based general ledger system. The lead engineer first decomposes the project into its constituent subprojects, such as inventory, accounts receivable, and merchandise distribution. The business or contracting organization may already have a legacy inventory system. Therefore, a sub-

project engineer is assigned the task of designing a component that effectively wraps the legacy system and provides an interface that conforms to the component model used by the other system components. Then the lead engineer would probably schedule a design inspection with everyone involved in the initial design to verify that all components interact and integrate logically according to the master software development plan.

The component infrastructure in Figure 1-1 is the result of the design process followed by the project engineers and design team. Through increasingly detailed refinement the software component infrastructure is implemented and forms the basis for the desired software system.

1.5 Conclusion

One of our goals in creating this book was to develop a set of rigorous definitions that clarify many of the terms used in the CBSE literature. CBSE is a challenge even for experienced software project team members because of the lack of rigorous definitions, the use of confusing and circular definitions, and descriptions passing for definitions. We have ensured that all the chapters in this book conform to the definitions presented in this chapter. The term software component has the same meaning in this chapter as in every other chapter in this book. Achieving these definitions was more than an academic exercise_it was a necessary step in defining the emerging field of CBSE. We have devoted considerable energy in developing these definitions and we expect you will find them extremely useful as you develop your CBSE vocabulary and skills.