# 1

# INTRODUCTION

COPYRIGHTED MATERIAL

***Objectives of This Chapter***

- Define COP and its major objectives
- Discuss the importance of COP
- Define components and their characteristics
- Distinguish hardware components and software components
- Investigate the differences between OOP and COP
- Introduce component-based software engineering

## 1.1 WHAT IS COP?

Throughout this book COP stands for *component-oriented programming*.

Programming is an activity of constructing computer programs that are a sequence of instructions describing how to perform certain tasks with computers. Programming can be classified or described in terms of the major techniques, concepts, or facility used for constructing computer programs. Different computers require different kinds of programming techniques. The Pascal's machine built in 1642 by Blaise Pascal (1623–1662), for instance, could only be programmed by operating gears and cranks. For all modern electronic computers, however, certain programming languages have to be used to program the computers. If a specific technique, say $X$, is being used to program computers, we will say this is *X-oriented programming*. Below is a brief discussion of different programming paradigms toward component-oriented programming.

*Component-Oriented Programming*, by Andy Ju An Wang and Kai Qian
ISBN 0-471-64446-3 Copyright © 2005 John Wiley & Sons, Inc.

*Gear-Oriented Programming*
Similar to the Pascal's machine mentioned above, for Charles Babbage's difference engine in 1822, programming means to change gears, cogs, and wheels. Through these physical motion and arrangement of gears, a new computation could be performed. Obviously, this kind of computer programming required both mental intelligence and physical capability as well.

*Switch-Oriented Programming*
For the first electronic computer ENIAC (Electronic Numerical Integrator And Computer) in 1942, programming means presetting switches and rewiring the entire system for each new "program" or calculation. This could be a tedious work because ENIAC had 6000 multiposition switches connecting a multitude of sockets with a veritable forest of jumper cables.

*Procedure-Oriented Programming*
With the help of high-level programming languages, programmers have been liberated from manipulating hardware details and the chores of machine languages. Programming languages such as Fortran, Pascal, and C encourage programmers to think in terms of procedures or functions. This was highlighted by structured programming, a top–down design and stepwise refinement design approach. In order to develop a computer program, the programmer would first think about what functionality should be implemented. Each functional requirement will be implemented eventually as a collection of procedures or functions.

*Object-Oriented Programming*
Object-oriented programming (OOP) encourages programmers to think in terms of data types. Programs are constructed around data types and data-type hierarchies. The fundamental building blocks in an object-oriented program are classes and objects. An object is a packet of information stored in a chunk of computer memory. Every object is associated with a data type, and the data type determines what can be done to an object. All programming languages have built-in data types, such as the integer data type and the character data type. Typically, programmers define data types and data-type hierarchies themselves so that they can describe individual entities to come up naturally in their applications.

*Aspect-Oriented Programming*
Aspect-oriented programming (AOP) introduces a new technology for separation of concerns in software development. AOP makes it possible to modularize crosscutting aspects of a system. Like objects, aspects may arise at any stage of the software life cycle, including requirements specification, design, implementation, and so on. Common examples of crosscutting aspects are design or architectural constraints, systemic properties or behaviors (e.g., logging and error recovery), and features. AOP blends support for many different kinds of modularity, including block structure, object structure, inheritance as well as crosscutting.

A key intuition underlying AOP is that simple hierarchies are not rich enough to capture complex structures. Therefore, AOP tries to explore a variety of mechanisms that make it possible to view and implement a system from multiple perspectives. Toward this end, a new programming language called *AspectJ* was designed and implemented. AspectJ is a seamless aspect-oriented extension to the Java programming language, enabling the clean modularization of crosscutting concerns such as error checking

and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multiobject protocols.

*Component-Oriented Programming*
Component-oriented programming enables programs to be constructed from prebuilt software components, which are reusable, self-contained blocks of computer code. These components have to follow certain predefined standards including interface, connections, versioning, and deployment.

Components come in all shapes and sizes, ranging from small application components that can be traded online through component brokerages to the so-called large grain components that contain extensive functionality and include a company's business logic. In principle, every component is reusable independent of context, that is to say, it should be ready to use whatever from wherever.

COP is to develop software by assembling components. While OOP emphasizes classes and objects, COP emphasizes interfaces and composition. In this sense, we could say that COP is an interface-based programming. Clients in COP do not need any knowledge of how a component implements its interfaces. As long as interfaces remain unchanged, clients are not affected by changes in interface implementations.

## 1.2 WHY IS COP IMPORTANT?

How do other more mature industries or engineering disciplines deal with the development of complex systems?

Building systems out of components is a natural part of engineering systems. The automotive industry, for instance, develops very complex cars using components of every size from a tiny screw to complex subsystems such as engines and transmissions. The modern automotive factory has become more of a system integrator than a manufacturer. It is easy to name many other industries and engineering disciplines making effective use of components by a rigorous set of standards that define interoperability. For hundreds of years, industries have adopted component standards for interchangeable parts and streamlined assembly tools to speed the development of highly complex products.

It is the Industry Revolution that dramatically changed the nature of production in which machines replaced tools, steam and other energy sources replaced human or animal power, unskilled workers replaced skilled workers, and massive products made by machines and assemble lines replaced handmade items.

In the software industry, however, the products are still mainly handmade items. The productivity is low, the quality is not guaranteed, and projects are mostly overrun. The phenomenon has been called "software crisis."

As hardware technology advances, the cost of developing a computer application is mainly on software part. The key issues of software engineering include how to create quality software effectively. Components are widely seen by software engineers as an important technology to address the "software crisis." The Industrial Revolution of software will occur through component-based software engineering.

There are a number of important reasons why COP is important. It provides a higher level of abstraction. There are an increasingly large number of reusable component libraries that assist in the development of applications for various domains.

There are three major goals of COP: conquering complexity, managing change, and reuse.

1. *Conquering Complexity* We are living in a complex world at information explosion age. According to a research team at UC Berkeley [SIMS 2000], "The world produces between 1 and 2 exabytes of unique information per year, which is roughly 250 megabytes for every man, woman, and child on earth. An exabyte is a billion gigabytes, or $10^{18}$ bytes." In computer science, the size and complexity of software have been increasing significantly. Table 1.1 shows the comparison of some operating systems in terms of their sizes. The first string in each box is the version; the second is the size measured in lines of source code, where $K = 1000$ and $M = 1{,}000{,}000$. Comparisons within a column have real meaning, while comparisons across columns do not [Tanebaum 2001]. Fortunately, COP provides an effective way to deal with the complexity of software: divide and conquer.
2. *Managing Change* Change is inherent in software engineering. The user requirements change, specifications change, personnel change, budget changes, technology changes, and so on. One of the fundamental software engineering principles is to emphasize the importance of managing change. It is important to place primary

**TABLE 1.1. Software Size Changes Over Time**

| Year | AT&T | BSD | MINIX | Linux | Solaris | Win NT |
|---|---|---|---|---|---|---|
| 1976 | V6 9K | | | | | |
| 1979 | V7 21K | | | | | |
| 1980 | | 4.1 38K | | | | |
| 1982 | Sys III 58K | | | | | |
| 1984 | | 4.2 98K | | | | |
| 1986 | | 4.3 179K | | | | |
| 1987 | SVR3 92K | | 1.0 13K | | | |
| 1989 | SVR4 280K | | | | | |
| 1991 | | | | 0.01 10K | | |
| 1993 | | Free 1.0 235K | | | 5.3 850K | 3.1 6M |
| 1994 | | 4.4 Lite 743K | | 1.0 165K | | 3.5 10M |
| 1996 | | | | 2.0 470K | | 4.0 16M |
| 1997 | | | 2.0 62K | | 5.6 1.4M | |
| 1999 | | | | 2.2 1M | | |
| 2000 | | Free 4.0 1.4M | | | 5.8 2.0M | 2000 29M |

emphasis during architecture and design on the dependencies between the components, and on the management of those dependencies.

COP provides an effective way to follow the software engineering principle of dealing with change: planning for change, design for change, and building for change. Components are easy to adapt to new and changing requirements. Software engineers have come to the consensus that the best way of dealing with constant changes is to build systems out of reusable components conforming to a component standard and plug-in architecture.

3. *Reuse* Software reuse allows to design and implement something once and to use it over and over again in different contexts. This will realize large productivity gains, taking advantage of best-in-class solutions, the consequent improved quality, and so forth.

   There are different levels of software reuse. Source code copy, for instance, is the lowest level of reuse. Procedural function libraries are a better form of reuse than source code copy, but not extensible. Class libraries are a better form of reuse, and they are extensible. However, it requires a lot of understanding before classes can be reused. Moreover, it supports only white-box reuse; clients will be affected if internals of classes changed. For instance, in an OOP language such as C++ or Java, derived classes are coupled to the base class implementation. Changes in any of the base classes in the inheritance hierarchy would break derived classes. Furthermore, this level of reuse is language specific; no reuse across code in other languages.

   COP supports highest level of software reuse because it allows various kinds of reuse including white-box reuse, gray-box reuse, and black-box reuse. White-box reuse means that the source of a software component is made available and can be studied, reused, adapted, or modified. Black-box reuse is based on the principle of information hiding. The interface specifies the services a client may request from a component. The component provides the implementation of the interface that the clients rely on. As long as the interfaces remain unchanged, components can be changed internally without affecting clients. Gray-box reuse is somewhere in between white-box reuse and black-box reuse.

As the size and complexity of software systems grows, the identification and proper management of interconnections among the pieces of the system becomes a central concern. COP provides a manageable solution to deal with the complexity of software, the constant change of systems, and the problems of software reuse. COP is now the de facto paradigm for developing large software systems, for example, enterprise-scale distributed applications, N-tier Web applications, and Web services.

## 1.3 WHAT IS A COMPONENT?

Software engineering has made great progress during the last 30 years with many innovative technologies invented and applied including structured programming, CASE technology, and object-oriented technology. However, the essential style of developing software maintains the same: the programmers write code line by line until it finishes. Such a preindustry style of production is the key factor for the unmatch between hardware productivity and software productivity. The revolutionized changes in software

development, like the industrial revolution to classical engineering disciplines, have to be introduced:

- The handcraft fashion of software development should be replaced by engineering methods.
- The line-by-line-coding style should be replaced by component-based development.
- The syntax-based programming should be replaced by assembling components based on their interface and semantics.

Obviously, such a revolutionary change would not occur until we could discover systematic, formal approaches to component-based software development.

Software components are defined in various ways from similar and different points of view. [Brown 1998] presented four definitions of a software component, summarizing the First International Workshop on CBSE in April 1998:

1. A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. (*Philippe Krutchen, Rational Software*)
2. A runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime. (*Gartner Group*)
3. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition. (*Clemens Szyperski*)
4. A business component represents the software implementation of an "autonomous" business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system. (*Wojtek Kozaczynski, SSA*)

Discussion about component definitions can also be found in many recent conferences and publications [Koza 1999; Parrish 1999; Wang 2000; Yacoub 1999; Fischer 2002; Fukazawa 2002]. In this book, we will use the following definition for a software component:

> A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface.

From this definition, a component is a program or a collection of programs that can be compiled and made executable. It is self-contained; thus, it provides coherent functionality. It is self-deployable so that it can be installed and executed in an end user's environment. It can be assembled with other components so that it can be reused as a unit in various contexts. The integration is through a component's interface, which means that the internal implementation of a component is usually hidden from the user.

Component technologies complying with the above definition include JavaBeans and enterprise Java Beans (EJB) from Sun Microsystems, COM (Component Object

Model), DCOM (Distributed Component Object Model), and .NET components from Microsoft Corporation, and CORBA (Common Object Request Broker Architecture) components from the Object Management Group. We will discuss all these components along with OSGi (Open Service Gateway Initiative) component model and Web service components in this book.

## 1.4 HARDWARE COMPONENTS AND SOFTWARE COMPONENTS

For the last 50 years, the technology for producing computer hardware has radically changed. Chips of integrated circuits (IC) and very large scale integrated circuits (VLSI) make the basic building blocks from which faster and less expensive computers can be assembled. In hardware engineering, component-based approach is widely utilized to build new products through prebuilt hardware components. There are some driving factors for component-based hardware design:

- The complexity and capability of hardware devices are increasing exponentially.
- The productivity and time-to-market requirements are getting more stringent.
- Synthesis from very high-level descriptions does not provide adequate quality of results.

Hardware engineers achieve required design productivity by assembling reusable blocks of intellectual property (IP) such as microprocessors, DSP, encryption/decryption chips, and so on. Recently, silicon capability has enabled integration of entire systems on a single die. The component-based approach has increased productivity and reliability of end products since each chip component has been well tested and fabricated massively. However, there was no dramatic change in software production. Each new software product required that software designers and programmers start from scratch and produce program code line by line until the program was finished. The big progress of software development for the last 50 years is from producing a program line by line using machine code to producing a program line by line using high-level programming languages. High-level programming languages offer several important benefits that improve programming productivity thanks to compiler techniques. Nevertheless, the line-by-line fashion of software production is still dominating the current practice and becoming the bottleneck of rapid developing quality software with low cost. What are the major barriers of applying component-based approach to software engineering? What are the differences between hardware components and software components?

Logically, computer hardware and software are equivalent in terms of their computability. That is, any operation performed by software can also be built directly into the hardware. On the other hand, any instruction executed by the hardware can also be simulated in software. Typical hardware components are ICs including medium-scale integration (MSI), large-scale integration (LSI), and VLSI circuits. These chip-level components are possible because of logic circuits that have well-defined input–output functions that do not require interaction with outside world directly. The communications among hardware components are supported by a system bus. A software component, on the other hand, is usually difficult to be expressed as simply a function from an input domain to an output codomain. Moreover, some software components

must interact with users or other physical plant in the environment; thus, they cannot be treated simply as black boxes like their hardware counterparts.

It is important to note that there is a well-defined computing model, that is, Boolean algebra, for gate-level design in building hardware components. Boolean algebra provides an economical way of describing the function of digital circuitry. Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function through algebraic laws.

What is the computing model for component-based software engineering? Obviously, a similar computing model is greatly in need for component-based software development, like Boolean algebra for component-based hardware engineering. As a matter of fact, Hoare logic [Hoare 1969], among others, provides a formalism, verifying the correctness of programs based on the program structure and their pre- and postconditions. As we move from statement-level programming or line-by-line coding to component-based development, we are in need of a formalism capturing the behavior of components and assuring the correctness of component interactions. We will discuss this issue extensively in Chapter 2.

Furthermore, we have to note that the comparison of software components to hardware components is limited for two reasons. The first is that unlike memory chips – which must be physically reproduced each time, that is, producing components with slight variations – producing software components is an exact duplicate each time. The second is that a software component's environment is likely to change for each new installation.

Finally, there is no temporal aspect, such as wear or chemical deterioration, to consider in software components [Mason 2002]. This difference is important especially when we consider the quality assurance issues for component-based development. For instance, the reliability for a hardware component is a function of four factors:

- errors in design
- errors in manufacture
- physical defects
- chemical and physical wear

The reliability for a software component, however, is a function of only one factor: errors in design (for a software component, here we consider its implementation as a part of design), because software does not wear out, nor, in most senses, are there manufacturing errors.

## 1.5 FROM OOP TO COP

### 1.5.1 Language View of COP

A program is an assembly of basic programming elements. In a machine language, the basic programming elements are 0s and 1s. Assembly languages use simple "words" as basic programming elements to abstract binary strings. Higher-level languages have made coding closer to human readable. The basic programming elements in a typical higher-level programming language are functions, also called *procedures* or *subroutines*. In an OOP (object-oriented programming) language, an object is a basic

programming element encapsulating data and behavior. Therefore, OOP is a way of partitioning functionality in modeling a problem or process into its constituent objects. The data abstraction provides a way of organizing data with associated operations, allowing classes or objects to be reused easily in another program. In COP, the basic programming elements are components, self-contained and self-deployable entities encouraging higher level of software reuse.

### 1.5.2 Software Engineering View of COP

OOP was claimed to support encapsulation, inheritance, and polymorphism. However, it never reached its goal because implementation inheritance violated true encapsulation. Furthermore, objects or classes are not self-deployable. We could distinguish COP from OOP in terms of various software engineering aspects as below:

- COP is interface-based, while OOP is object-based.
- COP is a packaging and distribution technology, while OOP is an implementation technology.
- COP supports high-level reuse, while OOP supports low-level reuse.
- COP, in principle, can be written in any language, while OOP is bound to OO languages.
- COP has loosely coupled components, while OOP has tightly coupled objects dependent on each other through inheritance implementation.
- COP has large granularity components, while OOP has objects as fine-grained units of composition.
- COP supports multiple interfaces, and interface-oriented design, while OOP does not provide clear relationship of interfaces among superclasses and subclasses.
- COP supports more forms of dynamic binding and dynamic discovery, while OOP provides limited support for object retrieval and runtime composition mechanisms.
- COP has better mechanisms for third-party composition, while OOP has limited forms of connectors (method invocation).
- COP provides more support for higher-order services (security, transactions, etc.), while OOP has limited sets of supported services such as security, transactions, and so on.
- COP components are designed to obey rules of the underlying component framework, while OOP objects are designed to obey OO principles.

Table 1.2 gives a brief summary of commonality and differences among structured programming (SP), OOP, and COP.

In terms of composing capability, SP is the lowest, OOP is high, and COP is very high. Two different implementation units in SP can never be interchangeable. In OOP, however, two different objects implementing the same specification are interchangeable. Yet, in COP, two different components with different specifications are interchangeable, as long as they satisfy those interface requirements for all client components. For instance, one server component may replace another one even if it has a different specification, as long as its specification includes the same interfaces the client components require.

**TABLE 1.2. Comparison of COP with OOP and SP**

| Capabilities | SP | OOP | COP |
|---|---|---|---|
| *Divide and Conquer*<br>• Manage complexity<br>• Break a large problem down into smaller pieces | Yes | Yes | Yes |
| *Unification of Data and Function*<br>• A software entity combines data and the functions processing those data<br>• Improves cohesion | | Yes | Yes |
| *Encapsulation*<br>• The client of a software entity is insulated from how that software entity's data is stored or how its functions are implemented<br>• Reduces coupling | | Yes | Yes |
| *Identity*<br>• Each software entity has a unique identity | | Yes | Yes |
| *Interface*<br>• Represents specification dependency<br>• Divides a component specification into interfaces<br>• Restricts inter-component dependency | | | Yes |
| *Deployment*<br>• The abstraction unit can be deployed independently | | | Yes |

## 1.6 COMPONENT-BASED SOFTWARE ENGINEERING

Sometimes, COP and CBSE (component-based software engineering) are interchangeable in literature. However, CBSE is a more general term, taking COP as only a part of it:

$$\text{CBSE} = \text{COA} + \text{COD} + \text{COP} + \text{COM},$$

where COA, COD, and COM represent component-oriented analysis, component-oriented design, and component-oriented management respectively. CBSE promises to accelerate software development and to reduce costs by assembling systems from prefabricated software components. Designing, developing, and maintaining components for reuse is, however, a very complex process, which places high requirements not only for the component functionality and flexibility but also for the development organization. CBSE covers many software engineering disciplines and different techniques, which still have not been fully defined, explained, and exploited either from theoretical or practical points of view.

In traditional software engineering, software development process consists of a sequence of activities or stages, namely, analysis, design, programming, testing, and integration. In CBSE, the main development stages become analysis, design, provision, and assembling. That is to say, the traditional programming, testing, and integration activities are replaced in CBSE by component provision and component assembling.

From the viewpoint of components, there are two kinds of activities in CBSE: *developing for reuse* (DF) and *developing with reuse* (DW). For DF, the development effort could be organized following traditional software engineering approaches, with the emphasis of component standards. For instance, each component provides two kinds of interfaces: (1) *provided interface*, which defined the public services this component will provide and (2) *required interface*, which specifies the services this component requires in order to work properly. For DW, software component search and retrieval have now become crucial activities for building applications.

From the viewpoint of engineering process, components could be classified into five different forms [Cheesman 2001]:

1. *Component Specification* This form represents the specification of a unit of software that describes the behavior of a set of *component objects* and defines a unit of implementation. Behavior is defined as a set of *interfaces*. A component specification is realized as a *component implementation*.
2. *Component Interface* The *interface* form presents a definition of a set of behaviors that can be offered by a *component object*.
3. *Component Implementation* The *implementation* form is a realization of *component specification*, which is independently deployable. This means it can be installed and replaced independently of other components. It does not mean that it is independent of other components – it may have many dependencies. It does not necessarily mean that it is a single physical item, such as a single file.
4. *Installed Component* The *installed* form is an installed (or deployed) copy of a *component implementation*. A component implementation is deployed by registering it with the runtime environment. This enables the runtime environment to identify the *installed component* to use when creating an instance of the component or when running one of its operations.
5. *Component Object* A *component object* is an instance of an *installed component*. This is a runtime concept. Similar to OOP, a component object in COP is an object with its own data and a unique identity, which performs the implemented behavior. An *installed component* may have multiple *component objects* (which require explicit identification) or a single one (which may be implicit).

## 1.7 SUMMARY

- With the advances of technology in computer hardware and software, the field of computer programming has grown from machine-code programming, via object-oriented programming, to component-oriented programming.
- Software components support black-box reuse.

- Software components encapsulate functionality and provide services through well-defined interfaces.
- Components are interchangeable software parts.
- A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface.
- Logically, hardware components and software components are equivalent, but COP needs fundamental theory and supporting tools to match the advance of hardware engineering.
- Compared with structured programming and OOP, the major features of COP include interfaces and independent deployment.
- There are two kinds of activities in component-based software engineering: development for reuse and development with reuse.
- From the viewpoint of software engineering, a component can take different forms: specification, interface, implementation, installed, and running object.

## 1.8 SELF-REVIEW QUESTIONS

**1.** The essential characteristics of a software component include
   **a.** composable
   **b.** interface
   **c.** deployable
   **d.** all of the above

**2.** Which of the following is common in hardware and software components?
   **a.** Binary form
   **b.** Unique return value
   **c.** Reusable
   **d.** Having Boolean logic as computing model

**3.** Which of the following could be a component?
   **a.** A macro in C
   **b.** A template in C++
   **c.** A block in Smalltalk
   **d.** A device bean

**4.** The major problem of Clemens' definition for a component is:
   **a.** Composability should not be a characteristic of components.
   **b.** Independence should not be a characteristic of components.
   **c.** Components are not in binary form.
   **d.** It does not reflect various forms of components.

**5.** Which of the following is false?
   **a.** Type definitions and Ada generics both can be components.
   **b.** Procedures, classes, modules, or even entire applications could be components.

**c.** Insisting on independence and binary form for components could allow for multiple independent vendors and robust integration.

**d.** Composition of components enables prefabricated things to be reused by rearranging them in ever-new composites.

**6.** CBSE is about

**a.** how to create reusable components

**b.** how to create software products with reusable components

**c.** how to develop software with component requirements specification, provisioning, and assembling

**d.** all of the above

**7.** Explain why "Object orientation has failed but component software is succeeding."

**a.** Object orientation has a 20-year history.

**b.** Component-based approach is new and promising.

**c.** Object technology tends to ignore the aspects of economies and markets. Objects, by themselves, are not independently deployable. Components are independent units of deployment.

**d.** Object-oriented programming languages are too hard to use.

**8.** Explain why "Component standards are essential for component markets to develop and thrive."

**a.** With standards, we could build a general-purpose component.

**b.** A component standard specifies the necessary interfacing of certain components as is needed to allow clients and vendors to work together.

**c.** The interplay of components is not a problem of software engineering.

**d.** A component does not need many clients to be economically viable.

**9.** A common feature among SP, OOP, and COP is

**a.** unification of data and function

**b.** divide and conquer for managing complexity

**c.** encapsulation for information hiding

**d.** interface specification

**10.** A unique feature for COP (which SP or OOP does not possess) is

**a.** unification of data and function

**b.** divide and conquer for managing complexity

**c.** encapsulation for information hiding

**d.** interface specification

**11.** Which of the following has the highest composability, that is, the capability of one entity being integrated with other entity?

**a.** SP

**b.** OOP

**c.** COP

**d.** XP (extreme programming)

### Keys to Self-Review Questions

**1.** d **2.** c **3.** d **4.** d **5.** a **6.** d **7.** c **8.** b **9.** b **10.** d **11.** c

## 1.9 EXERCISES

**1.** When we discuss component-oriented programming, we require that a component is in binary form and to be independently deployable. Discuss why these requirements are essential for component-oriented programming.

**2.** Which of the following could be components and why?
- **a.** A procedure
- **b.** A class
- **c.** A module
- **d.** A function
- **e.** An object
- **f.** A subroutine
- **g.** A UML diagram
- **h.** A testing case
- **i.** An event
- **j.** Type declaration
- **k.** C macros
- **l.** C++ template
- **m.** Smalltalk blocks
- **n.** A protocol
- **o.** A package in Java
- **p.** An assembly in. NET

**3.** List three major goals for component-oriented programming.

**4.** List different forms of components along with brief explanations.

**5.** What are the two kinds of developing activities in component-based software engineering?

## REFERENCES

[Brown 1998] Brown, A. W. and Wallnau, K. C. "The current state of CBSE," *IEEE Software*, Sept./Oct.: 37–46, 1998.

[Cheesman 2001] Cheesman, J. and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.

[Fischer 2002] Fischer, B. Deduction-Based Software Component Retrieval, Ph.D. thesis, *http://ase.arc.nasa.gov/people/fischer/papers/phd.html*, 2002.

[Fukazawa 2002] Fukazawa, Y., Washizaki, H., and Yamamoto, H. Software Component Metrics, *http://www.fuka.info.waseda.ac.jp/Project/CBSE/metrics.html*, 2002.

[Hoare 1969] Hoare, C. A. R. "An axiomatic basis for computer programming," *Communications of ACM*, 12(10): 576–580, 1969.

[Koza 1999] Kozaczynski, W. "Composite nature of component," *1999 International Workshop on Component-Based Software Engineering*, *http://www.sei.cmu.edu/cbs/icse99/papers*, 1999.

[Mason 2002] Mason, D. V. Probabilistic Program Analysis for Software Component Reliability, Ph.D. thesis, *www.sarg.ryerson.ca/~dmason/thesis.pdf*, 2002.

[Parrish 1999] Parrish, A., Dixon, B., and Hale, D. "Component based software engineering: a broad based model is needed," *1999 International Workshop on Component-Based Software Engineering*, *http://www.sei.cmu.edu/cbs/icse99/papers/16/16.htm*, 1999.

[SIMS 2000] School of Information Management and Systems. University of California at Berkeley, How Much Information? *http://www.sims.berkeley.edu/research/projects/how-much-info/summary.html*, 2000.

[Tanebaum 2001] Tanebaum, A. *The Modern Operating Systems*, 2nd ed., Prentice Hall, 2001.

[Wang 2000] Wang, J. A. "Towards component-based software engineering," *Journal of Computing Sciences in Colleges*, 16(1): 177–189, 2000.

[Yacoub 1999] Yacoub, S., Ammar, H., and Mili, A. "Characterizing a software component," *1999 International Workshop on Component-Based Software Engineering*, *http://www.sei.cmu.edu/cbs/icse99/papers/34/34.htm*, 1999.