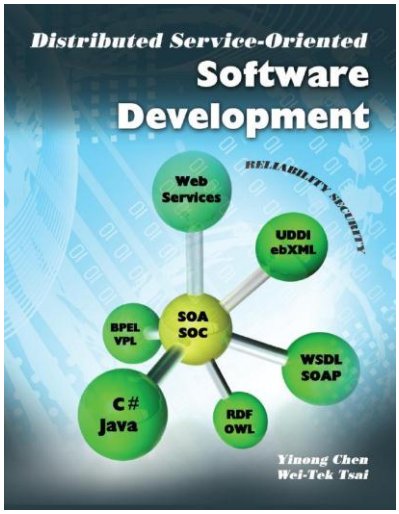
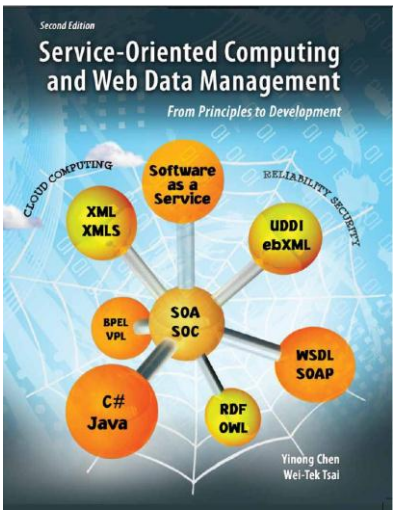
DISTRIBUTED SERVICE-ORIENTED SOFTWARE DEVELOPMENT





FIRST EDITION ***SECOND EDITION*** ***THIRD EDITION***

YINONG CHEN AND WEI-TEK TSAI

ARIZONA STATE UNIVERSITY

KENDALL/HUNT PUBLISHING COMPANY
4050 Westmark Drive Dubuque, Iowa 52002

ISBN 978-0-7575-5273-1

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Table of Contents (First Edition)

Preface

Software development has evolved for several generations from imperative, procedural, object-oriented, to distributed object-oriented paradigms. As the emergence of service-oriented computing, distributed software development is shifting from *distributed object-oriented development*, represented by CORBA (Common Object Request Broker Architecture) developed by OMG (Object Management Group) and Distributed Component Object Model (DCOM) developed by Microsoft, to *distributed service-oriented development*. Service-oriented computing and service-oriented software development have been adopted and supported by all major computer companies, including BEA, Google, HP, IBM, Intel, Microsoft, Oracle, SAP, and Sun Microsystems, and their technologies have been standardized by OASIS, W3C, and ISO.

Before we start to introduce what this book is about, let us first clarify three fundamental concepts: service-oriented architecture, service-oriented computing, and service-oriented software development.

Service-Oriented Architecture (SOA) is a distributed software architecture, which considers a software system consisting of a collection of loosely coupled services that communicate with each other through standard interfaces and protocols. These services are platform independent. Services can be published in public or private directories or repositories for software developers to compose their applications. As a software architecture, SOA is a conceptual model that concerns the organization and interfacing among the software components (services). It does not concern the development of operational software.

Service-Oriented Computing (SOC) refers to the computing paradigm that is based on the SOA conceptual model. However, SOC goes a step further to include not only the concepts and principles, but also the methods, algorithms, coding, and evaluation, which are a large part of the software development process.

Service-Oriented Development (SOD) concerns the entire software development cycle based on SOA concepts and SOC paradigm, including requirement, specification, architecture design, composition, service discovery, service implementation, testing, evaluation, deployment, and maintenance. SOD also involves using the current technologies and tools to effectively produce operational software.

We use "Distributed Service-Oriented Software Development" as the title of the book to compare with the widely used "Distributed Object-Oriented Software Development" approach, and to emphasize the fact that service-oriented software development is distributed naturally.

Not only is the software under development distributed in different computers in different locations, but also the development process is distributed, in the sense that the application builders, service brokers, and service providers are developers working independently in different locations, but following the same interfaces and standards. Furthermore, we have a chapter (Chapter Two) to discuss distributed computing in general and how SOA, SOC, and SOD fit into the framework of general distributed computing.

Recently, many SOA, SOC, and SOD books have been published in response to the growing requirements in these areas. These books fall into one of the three categories:

(1) high-level concepts and principles in SOA;

(2) one of the aspects of the SOC, such as BPEL, Ontology, or XML;

(3) SOD using a specific platform, such as Visual Studio .Net, Oracle SOA Suite, Java EE, or WebSphere. Most of these books are written by developers, and are largely focused on the language, platforms, and tools.

Different from the existing books, this book takes a balanced approach to teach all three topics of SOA, SOC, and SOD in one course, and covers a large portion of each topic in depth. The main concern of the book is to teach the SOA/SOC concepts, principles, and methods.

However, concepts, principles, and methods are not only explained in text and diagram, but also demonstrated in working code.

We believe that students can better understand concepts, principles, and methods, if they see a piece of working code that implements them.

We also introduce the cutting-edge technologies and tools that can be applied to develop operational software with reasonable size and functionality, such as an operational online bookstore, trading site, or a robotics program manipulating a real robot to traverse a maze with artificial intelligence. Such software can never be developed in a course assignment without the latest development tools and without using the services and components made available by professional service providers. Many exercises and at least one large project are given at the end of each chapter of the book for students to practice SOA and SOC concepts and to develop operational software.

This book covers SOA, SOC, and SOD topics in breadth and depth.

The book is based on the materials taught by the authors in CSE445/598 (Distributed Software Development) course in Computer Science and Engineering at Arizona State University every semester since Fall 2006. The CSE445 session is for seniors and the CSE598 session is for graduate students. The CSE598 also has an online session that is taught to students in the executive master's program in engineering. Many of these students are on the side of software project management. A part of advanced materials of the text was also taught in CSE 565

(Software Verification, Validation, and Testing). The objectives and outcomes of a course based on the text can include:

1. To develop an understanding of the software engineering of programs using concurrency and synchronization, with the following outcomes:

   * Students can identify the application, advantages, and disadvantages of concurrency, threads, and synchronization.

   * Students can apply design principles for concurrency and synchronization.

   * Students can design and write programs demonstrating the use of concurrency, threads, and synchronization.

2. To develop an understanding of the development of distributed software, with the following outcomes:

   * Students can recognize alternative distributed computing paradigms and technologies;

   * Students can identify the phases and deliverables of the software lifecycle in the development of distributed software;

   * Students can create the required deliverables in the development of distributed software in each phase of a software lifecycle;

   * Students understand the security and reliability attributes of distributed applications.

3. To develop an ability to design and publish services as building blocks of service-oriented applications, with the following outcomes:

   * Students understand the role of service publication and service directories;

   * Students can identify available services in service registries;

   * Students can design services in a programming language and publish services for the public to use.

4. To build skills in using a current technology for developing distributed systems and applications, with the following outcomes:

   * Students can develop distributed programs using the current technology and standards;

   * Students can use the current framework to develop programs and web applications using graphical user interfaces, remote services, and workflow.

This book is not for an introductory course in programming. Its main audiences are the seniors and graduate students in computer science and engineering, or software engineers with programming background. The readers are expected to be fluent in one of the object-oriented

programming languages such as C++, C#, and Java. Furthermore, students are expected to have understood basic software engineering principles.

The book consists of nine chapters and an appendix. Each chapter is a unit that can be taught in six to nine lecture hours, depending on the level of the detail the instructor wants to cover. They are

Chapter 1 Introduction to Distributed Service-Oriented Computing

Chapter 2 Distributed Computing with Multithreading

Chapter 3 Getting Started with Service-Oriented Software Development

Chapter 4 XML and Related Technologies

Chapter 5 Composition Languages for Service-Oriented Software Development

Chapter 6 Dependability of Service-Oriented Software

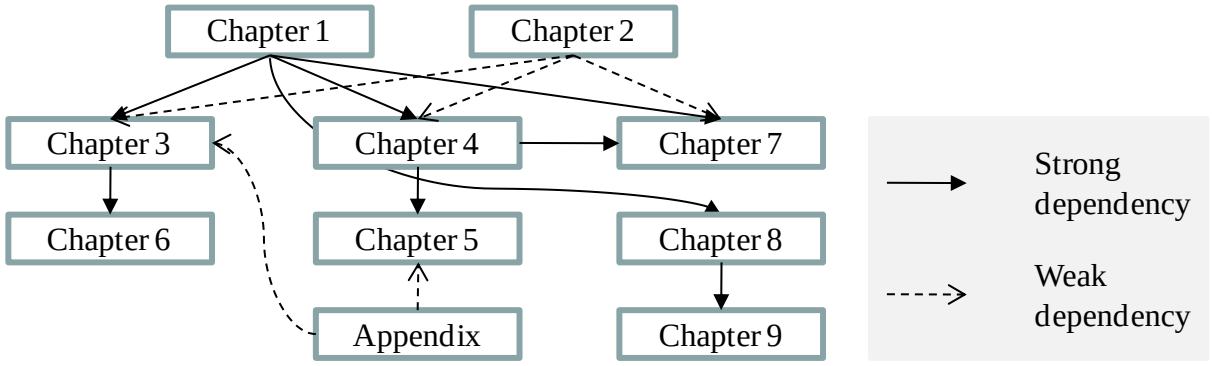
Chapter 7 Database and Ontology in Distributed Service-Oriented Software

Chapter 8 Service-Oriented Application Architecture

Chapter 9 A Mini Walkthrough of Service-Oriented Software Development

Appendix Tutorials on Component-Based and Service-Oriented Software Development

The dependency among the chapters is illustrated in the diagram below. Based on the dependency, a subset of the chapters can be selected to satisfy a set of course requirements.



This book is not intended to be a research monograph, but an undergraduate text for teaching senior and graduate students on SOA, SOC, and SOD. However, research students and working professionals may still find this book useful, because of its comprehensive and in-depth discussions of the state-of-the-art contents, cutting-edge technologies, and professional development tools. The book is based not only on the teaching experiences of the authors in these areas, but also on the understanding and expertise that the authors have accumulated in their research in these areas.

As SOA, SOC, and SOD are new and dynamic, the technologies and tools are evolving rapidly. Some of the materials may need to be updated soon after the print of the book. It is our intention to cover the latest concepts and technologies, and we must cut in at some point in this process. We have put more emphasis on the SOA and SOC concepts, principles, and methods, which are

relative stable compared to the SOD technologies and tools. We started to teach from the material of the book in Fall 2006. Large part of the development examples are initially based on .Net 2005. Now .Net 2008 is released. With little or no revision, we are able to test or convert all the examples into .Net 2008 before the print of the book. We expect the examples to work for the new editions of the tools in the future.

The tutorials in the appendix of the book are an important addition to the book. They provide full detail of Web application development discussed in Chapter Three and the robotics software development discussed in Chapter Five. On the other hand, the tutorials can be taught independently of the main text to students with no programming experience. In fact, the contents of the tutorials have been taught in a service-oriented computing course for high school students.

We like to thank many of our sponsors, supporters and colleagues in this project including Prof. Xiaoying Bai of Tsinghua University, Prof. Gary Bitter of Arizona State University, Prof. Farokh Bastani of University of Texas at Dallas, Prof. Kuo-Ming Chao of Coventry University, Dr. Shuyuan Chen of SAP, Dr. J. Y. Chung of IBM, Prof. Zhihui Du of Tsinghua University, Dr. K. W. Hwang of IBM, Prof. Kane Kim of University of California at Irvine, Prof. Y. H. Lee of Arizona State University, Prof. Yisheng Li of Fudan University, Prof. K. J. Lin of University of California at Irvine, and Dr. Raymond Paul of DoD OSD NII, Dr. Mary White of Arizona State University, Prof. S. S. Yau, Arizona State University, Prof. I-Ling Yen of University of Texas at Dallas. They contributed to our understanding of the materials. We also acknowledge the generous support from U.S. Department of Education and U.S. Department of Defense. Without their support, this book will not be possible. We also thank the teaching assistants and research assistants at Arizona State University involving Zhibin Cao, Calvin Cheng, Sandy Chow, Jay Elston, Qian Huang, Sheng Liu, Zheng Liu, Wu Li, Xin Sun, Jingjing Xu, Xinyu Zhou, and Peide Zhong. They validated many of the examples and assignments used in the book. Finally, we would like to thank our families for their support and understanding of taking on such a project while carrying out a full research and teaching load at the university.

Note for Instructors

All the homework assignments have been classroom-tested at Arizona State University. Furthermore, all the code presented in this book has been developed and tested. Contact the authors if you are interested in obtaining more materials in this book. This book also has a corresponding website at http://asusrl.eas.asu.edu/share/services/book/ where you can download resources related to this book. Instructor-only resources can be obtained by directly contacting the authors at {yinong, wtsai}@asu.edu.

Yinong Chen

Wei-Tek Tsai

May 2008

Chapter 1

Introduction to Distributed Service-Oriented Computing

This chapter introduces computer architecture, different computing paradigms, and particularly, the distributed computing paradigm and Service-Oriented Computing (SOC) paradigm.

1.1 Computer Architecture and Computing Paradigms

Software architectures and distributed software development are related to the computer system architectures on which the software is executed. This section introduces the computer architectures and various computing paradigms.

1.1.1 Computer Architecture

The computer architecture for a single-processor computer often refers to the processor architecture, which is the interface between software and hardware or the instruction architecture of the processor [Patterson 2004]. For a computer with multi-processors, the architecture often refers to the instruction and data streams. *Flynn's Taxonomy* [Flynn 1972] categorized computer architecture into four types:

- Single Instruction stream and Single Data stream (SISD), which refers to the simple processor systems;
- Single Instruction stream and Multiple Data streams (SIMD), e.g., the vector or array computers;
- Multiple Instruction streams and Single Data stream (MISD), e.g., fault-tolerant computer systems that perform redundant computing on the same data stream and voting on the results;
- Multiple Instruction streams and Multiple Data streams (MIMD), which refers to the systems consist standalone computer systems with their own memory and control, ALU, and I/O units.

The MIMD systems are often considered distributed systems, which have different areas of concerns, as shown in Figure 1.1. Distributed computing is about the principles, methods, and techniques of expressing computation in a parallel and/or distributed manner. Distributed software architecture concerns the organization and interfacing among the software components. Network architecture studies the topology and connectivity of network nodes. The network communication deals with the layers of protocols that allow the nodes to communicate with each other and understand the data formats of each other. Some studies use operating systems to differentiate distributed systems and networks. Distributed systems have coherent operating systems, while a set of network nodes has independent operating systems.

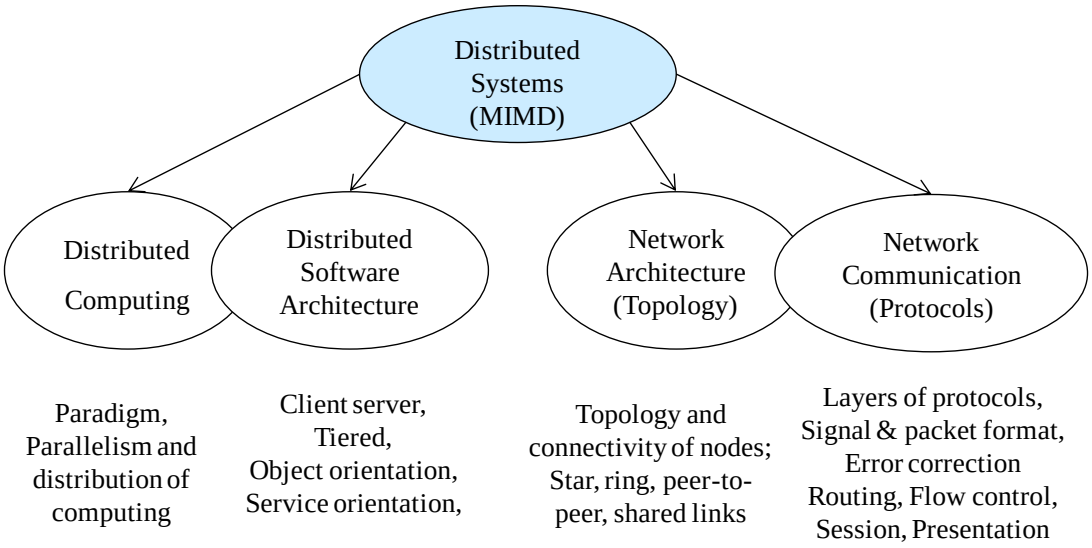


Figure 1.1 Distributed systems and networks

1.1.2 Software Architecture

The software architecture of a program or computing system is the structure, which comprises software components, the externally visible properties of those components, and the relationships between them [Bass 2003]. The design of software architecture does not mean to develop the operational software. Instead, it can be considered a conceptual model of the software, which is one of the development steps that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements;

(2) consider architectural alternatives at a stage when making design changes is still relatively easy;

(3) define the interfaces between the components;

(4) reduce the risks associated with the construction of the software.

It is important to design software architecture before designing the algorithm and implementing the software, because software architecture enables the communication between all parties (stakeholders) interested in the development of a computer-based system. The service-oriented architecture (SOA), which is a main topic of the book, explicitly involves three parties -- service providers, service brokers, and service requesters -- in the software architecture design, while each party conducts its algorithmic design and coding independently.

The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and on the ultimate success of the system as an operational entity.

1.1.3 Computing Paradigms

Numerous programming languages have been developed in history, but only several thousands of them are actually in use. Compared to natural languages that were developed and evolved independently, programming languages are far more similar to each other. They are similar to each other because of the following reasons. They share the same mathematical foundation (e.g., Boolean algebra, logic). They provide similar functionality (e.g., arithmetic, logic operations, and text processing). They are based on the same kind of hardware and instruction sets. They have common design goals: to find languages that make it simple for humans to use and efficient for hardware to execute. The designers of programming languages share their design experiences.

Some programming languages, however, are more similar to each other, while some other programming languages are more different from each other. Based on their similarities or the paradigms, programming languages can be divided into different classes. In programming language's definition, **paradigm** is a set of basic principles, concepts, and methods of how computation or algorithm is expressed. The major paradigms include imperative, OO, functional, logic, distributed, and SOC.

The **imperative**, also called the **procedural**, computing paradigm expresses computation by fully specified and fully controlled manipulation of named data in a step-wise fashion. In other words, data or values are initially stored in variables (memory locations), taken out of (read from) memory, manipulated in ALU (arithmetic logic unit), and then stored back in the same or different variables (memory locations). Finally, the values of variables are sent to the I/O devices as output. The foundation of imperative languages is the **stored program concept** based computer hardware organization and architecture (von Neumann machine) (see for example http://en.wikipedia.org/wiki/Von_Neumann_machine). Typical imperative programming languages include all assembly languages and earlier high-level languages like FORTRAN, Algol, Ada, Pascal, and C.

The **object-oriented** computing paradigm is basically the same as the imperative paradigm, except that related variables and operations on variables are organized into classes of **objects**. The access privileges of variables and methods (operations) in objects can be defined to reduce (simplify) the interaction among objects. Objects are considered the main building blocks of programs, which support the language features like inheritance, class hierarchy, and polymorphism. Typical OO programming languages include Smalltalk, C++, Java, and C#.

The **functional**, also called the **applicative**, computing paradigm expresses computation in terms of mathematical functions. Since we have been expressing computation in mathematical functions in many of the mathematical courses, functional programming is supposed to be easy to understand and simple to use. However, since functional programming is rather different from imperative or OO programming, and because most programmers first get used to writing programs in imperative or OO paradigm, it becomes difficult to switch to functional programming. The main difference is that there is no concept of memory locations in functional

programming languages. Each function will take a number of values as input (parameters) and produce a single return value (output of the function). The return value cannot be stored for later use. It must be used either as the final output or used immediately as the parameter value of another function. Functional programming is about defining functions and organizing the return values of one or more functions as the parameters of another function. Functional programming languages are mainly based on the lambda-calculus that will be discussed in Chapter Four. Typical functional programming languages include ML, SML, and Lisp/Scheme.

The **logic**, also called the **declarative**, computing paradigm expresses computation in terms of logic predicates. A logic program is a set of facts, rules, and questions. The execution process of a logic program is to compare a question to each fact and rule in the given fact and rulebase. If the question finds a match, then we receive a yes-answer to the question. Otherwise, we receive a no-answer to the question. Logic programming is about finding facts, defining rules based on the facts, and writing questions to express the problems we wish to solve. Prolog is the only significant logic programming language.

All these computing paradigms support both "programming-in-the-small" and "programming-in-the-large." The former emphasizes the development of program components or modules using basic programming constructs such as sequential, conditional branching, and looping constructs. The latter emphasizes developing large applications. Large applications often require more people and effort, and they are used in critical applications such as banking, e-business, embedded systems, e-government.

Another important paradigm is **component-based computing**. This paradigm emphasizes composing large applications based on pre-programmed components or modules. Components or modules are often pre-compiled program units, and they are linked into the application prior to the execution. Conceptually, component-based computing is not new. OO computing is widely considered to be component-based computing, where each class or object is a component. A namespace (a group of classes) can be considered a component, also. However, both of these views are tightly coupled with the specific definition of "class." Component-based computing can have a broader meaning, which allows any unit or module to be considered a component, and thus, can be consider a distinct paradigm different from OO computing. A component can be as small as an object and can be as large as an application, and a component is often well encapsulated. Thus, for some, SOC is really component-based computing, as services can be components. In their mind, SOC is essentially component-based computing but each component is specified using open standards.

Distributed computing involved computation executed on more than one logical or physical processors or computers. These units cooperate and communicate with each other to complete an integral application. The computation units can be functions (methods) in the component, components, or application programs. The main issues to be addressed in the distributed computing paradigms are concurrency, concurrent computing, resource sharing, synchronization, messaging, and communication among distributed units. Different levels of distribution leads to different variations. **Multithreading** is a common distributed computing technique that allows different functions in the same software to be executed concurrently. If the distributed units are at the object level, this is **distributed OO computing**. Some well-known distributed OO computing frameworks are CORBA (Common Object Request

Broker Architecture) developed by OMG (Object Management Group) and Distributed Component Object Model (DCOM) developed Microsoft.

Service-oriented computing (SOC) is another distributed computing paradigm. SOC differs from distributed OO computing in several ways:

- SOC emphasizes on distributed services (with possibly service data) rather than distributed objects;
- SOC explicitly separates development duties and software into service provision, service brokerage, and application building through service consumption;
- SOC supports reusable services in (public or private) repositories for matching, discovery and (remote or local) access;
- In SOC, services communicate through open standards and protocols that are platform independent and vendor independent.

Figure 1.2 summarizes the features of different computing paradigms.

It is worthwhile noting that many languages belong to multiple computing paradigms. For example, C++ is an OO programming language. However, C++ also includes almost every feature of C. Thus, C++ is also an imperative programming language, and we can use C++ to write C programs.

Java is more an OO language, that is, the design of the language put more emphasis on the object orientation. However, it still includes many imperative features. For example, Java's primitive type variables use value semantics and do not obtain memory from the language heap.

Lisp contains many non-functional features. Lisp and Scheme are functional programming languages, but they also contain many non-functional features such as sequential processing when input and output are involved

Prolog is a logic programming language, but its arithmetic operations use the imperative approach.

In summary, these computing paradigms often overlap with each other. For example, OO computing languages are often also imperative programming languages, and SOC languages such as C# and Java are also OO programming languages. Thus, a single programming language can be used to write programs in different computing paradigms. See [Chen 2006] for an introduction to these computing paradigms using C, C++, Scheme, and Prolog.

1.2 Distributed Computing and Distributed Software Architecture

In distributed computing, computation is distributed over multiple computing units (processors or computers), rather than confined to a single computing unit. Virtually, all large computing systems now are distributed, as the multi-core processor design is introduced.

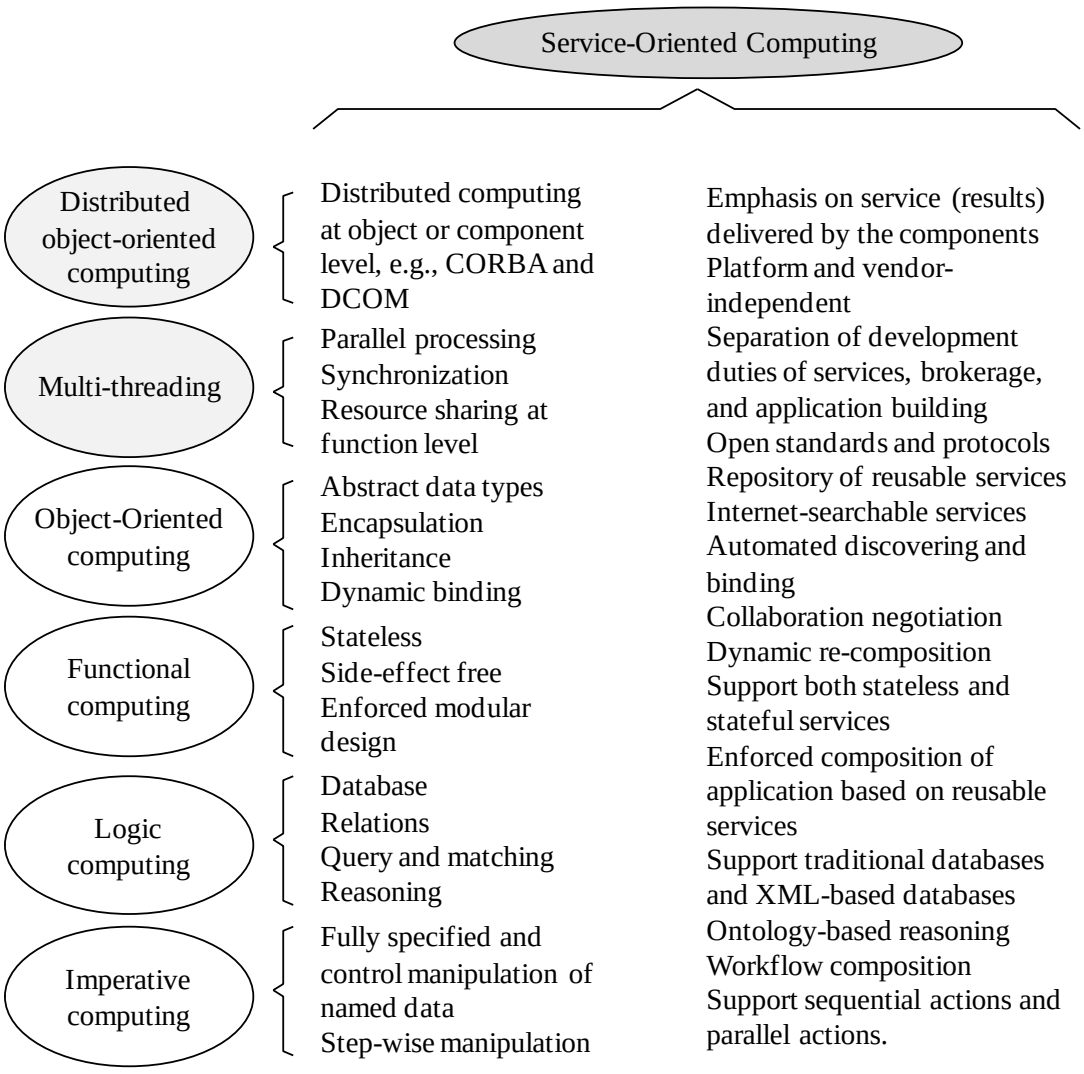


Figure 1.2 Features of different computing paradigms

1.2.1 Distributed Computing

Software architecture describes the system structure and functionality allocation over a number of logical or physical computing units. Having the right architecture for an application is essential to achieve the desired quality of service.

Distributed computing often has to deal with multiple dimensions of challenges, including complexity, communication and connectivity, security and reliability, manageability, and unpredictability and nondeterministic behaviors. These challenges are well expressed in the following eight **fallacies of distributed computing**, proposed by Sun Microsystems fellows [http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing]:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.

4. The network is secure.
5. Topology does not change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous

The first four fallacies, called the fallacies of networked computing, were proposed by Bill Joy and Tom Lyon in 1991. Peter Deutsch added the next three, which are often referred to as Deutsch's seven fallacies. James Gosline added the eighth fallacy in 1997.

1.2.2 N-Tier Architecture

Similar to the OSI seven-layer network architecture, **distributed software architecture** often has a layered structure, in which components are organized in layers and refers to N-Tier Architecture. For example, complex business software can be organized in the following five-tier model:

1. **Presentation tier**: The layout of the Graphic User's Interface (GUI);
2. **Implementation of the presentation tier**: Program the GUI in certain programming language;
3. **Business logic tier**: Implementation of the business objects, rules, and policies;
4. **Data access tier**: Interfaces from the business logic to the databases;
5. **Data tier**: Databases.

The tiered design is well suited for distributed computing, with one tier or a number of adjacent tiers residing on one node of the distributed system. Another advantage is the flexibility in maintaining the system; the tiers can be modified relatively independently. For example, if tier 2, the implementation of the presentation, must be changed, none of the other tiers has to be changed from the logic point of view. The user can still use the same interfaces and the business logic can remain unchanged. From the programming point of view, the tier above may need to be changed if different user interfaces are offered at the modified tier.

Two-tier architecture and three-tier architecture are the most widely used distributed architecture. In the **two-tier architecture**, also known as **client-server architecture**, the application is modeled as a set of services that are provided by servers and a set of clients that use these services. Clients know of servers but servers do not need to know of clients. Both clients and servers are logical processes, which can reside on the same computer or on different computers. Figure 1.3 shows an example of the client-server architecture. The servers can form a federation, which backs each other up to provide dependable services to their clients. The federation is often transparent to the clients. Data services provided by databases are important to most business applications and the databases are part of the server in this architecture.

The client-server architecture can be further classified into **thin-client** and **fat-client** architectures. In the thin-client architecture, all of the application processing and data management are carried out on the server. The client is simply responsible for running the presentation software.

In the fat-client architecture, the software on the client implements the application logic and the interactions with the system user. The server is responsible for data management (database) only.

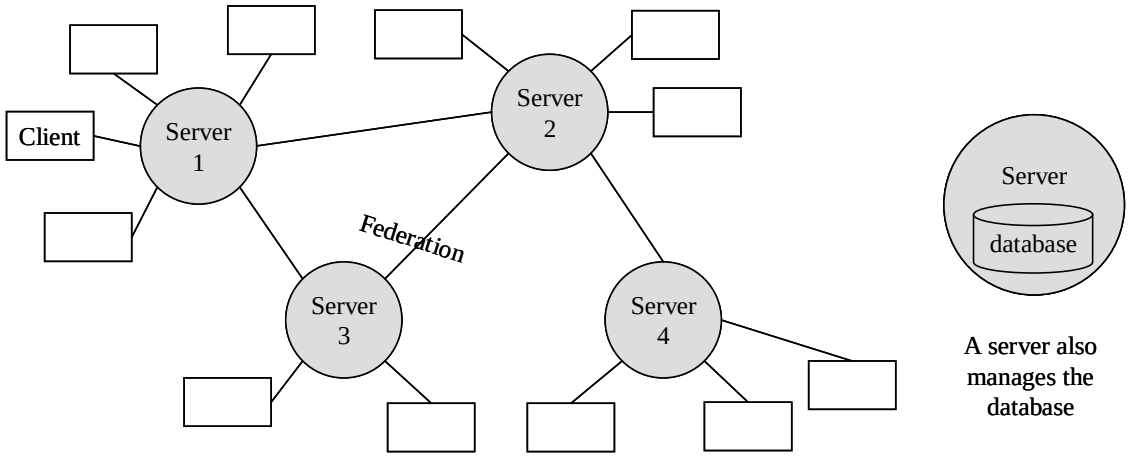


Figure 1.3 Client-server architecture, with the federation among the servers

Three-tier architecture consists of three layers shown in Figure 1.4. Each layer is executed on a separate processor. It is a more balanced approach, which allows for better performance than a thin-client approach and is simpler to manage than a fat-client approach. Three-tier architecture is a more scalable architecture -- as demands increase, extra servers can be added.

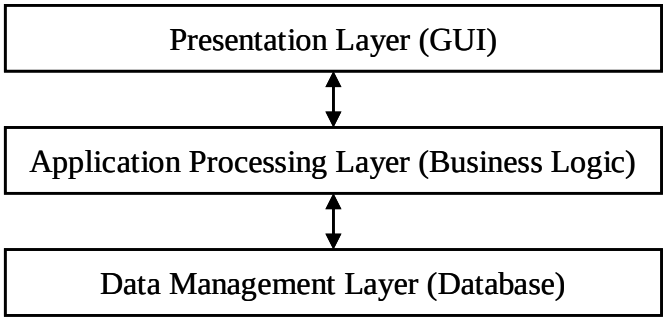


Figure 1.4 Three-tier architecture

Figure 1.5 shows an example of a three-tier Internet banking system, where the clients can include GUI of ATM (Automated Teller Machine), POP (Point Of Purchase), and Web access to the user account. The application-processing layer can reside in the bank's IT center, responsible for processing all the requests. The data, such as account information and balance, are stored in a different server managing the databases.

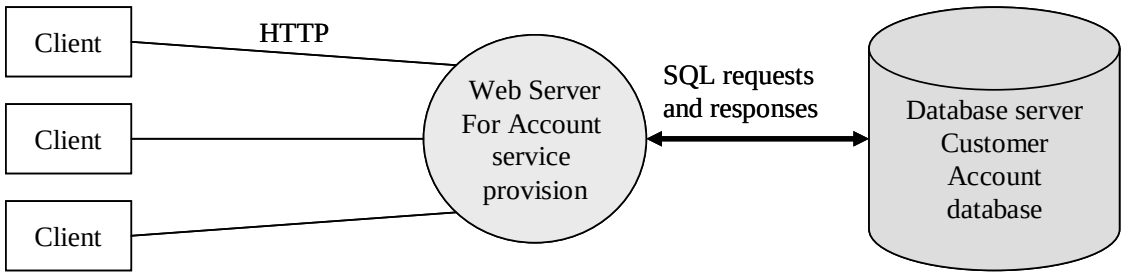


Figure 1.5 Example of a three-tier Internet banking system

The service-oriented architecture can be implemented as **four-tier architecture**, as shown in Figure 1.6(a), which consists of presentation layer, application layer, service repository layer, and data management. However, service-oriented architecture does not have to be tied to this architecture, where only adjacent tiers can communicate with each other. Figure 1.6(b) and (c) show two possible variations of implementing the SOA.

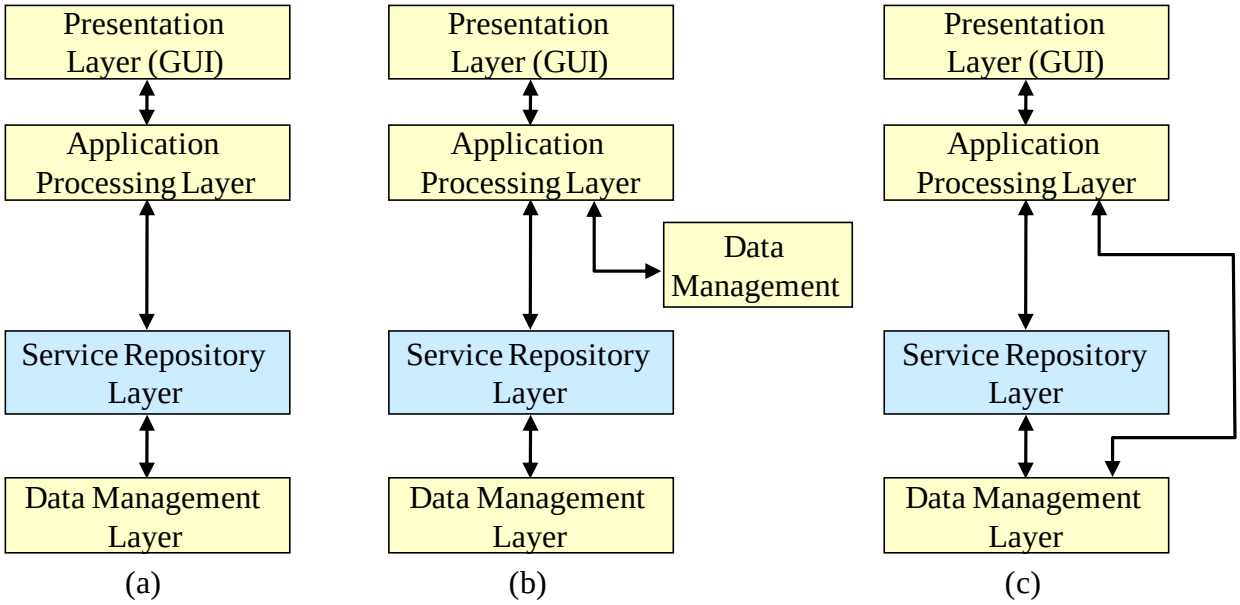


Figure 1.6 Four-tier architecture and its variations

1.2.3 Distributed Object Architecture

Different from the N-tier architecture, where the clients and servers are explicitly differentiated, the distributed object architecture makes no explicit distinction between clients and servers. Each distributable entity is an object that provides service to other objects and receives services from other objects.

Distributed object architecture is more generic in implementing different applications. However, it is more complex to design and to manage than the tiered architecture, because it allows the system designer to delay decisions on where and how services should be provided. In other words, it is an open system architecture that allows new resources to be added to the system as required. The system built on distributed object architecture is flexible and scalable. It is possible (e.g., written in the same language) to reconfigure the system dynamically with objects migrating across the network as required. As a logical model, distributed object architecture allows developers to structure and organize the system. In this case, developers can focus more on provision of the application functionality in terms of services and combinations of services.

The two major implementations of the distributed object architecture are CORBA (Common Object Request Broker Architecture) developed by OMG (Object Management Group) and Distributed Component Object Model (DCOM) developed by Microsoft.

In CORBA, object communication is through a middleware system called an Object Request Broker (ORB), also called software bus, as shown in Figure 1.7.

CORBA objects are comparable, in principle, to objects in C++, C#, and Java. The objects have a separate interface definition that is expressed using a common language IDL (Interface Definition Language), which is similar to C++. The interfaces of an object can be written in any language. A program translator can be used to translate the interface code, for example, in C++ and Java, into IDL code, and thus the objects written in different programming languages can communicate with each other. The ORB handles object communication through the stubs written in IDL. A service provider will make their service ports known as the IDL stubs. If a service requester calls a stub, the call will be translated to a call to the function of the service provider.

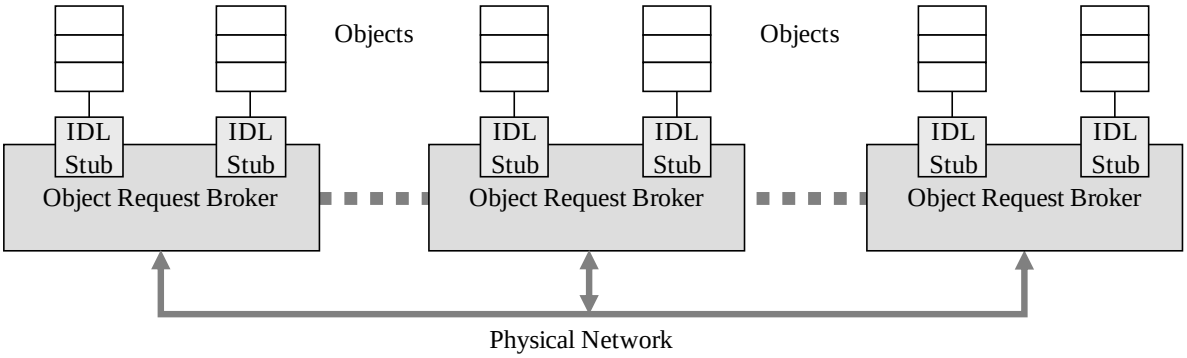


Figure 1.7 CORBA architecture

DCOM (Distributed Component Object Model) is the Microsoft's distributed software development framework before Visual Studio .Net. DCOM allows software components to distribute across several networked computers to communicate with each other. Initially, the distributed software development framework was called OLE (Object Linking and Embedding), a distributed object system. The framework evolved for several generations. It was extended into "Network OLE" and then to COM (Component Object Model) in 1993, which provides the communication capacity among objects. In Windows 2000, significantly extensions were made to COM and it was renamed COM+, before it evolved into DCOM. All technologies in DCOM were integrated into or replaced by Visual Studio .NET, which is an all-in-one OO, distributed, and service-oriented software development environment.

Distributed object architecture is a predecessor of SOC. It has many characteristics of SOC. The significant improvements and achievements made in SOC include:

- All major computer companies have agreed on the SOC standards, protocols, and interfaces for creating interoperable services, which are platform and language independent. In the case of distributed object architecture, CORBA and DCOM have similar functionality and goals; however, the systems developed in the two environments are not interoperable, and DCOM is also platform dependent.
- SOC has explicitly separated the duties of development: The service providers develop services, the service requesters build the application using existing services, and service brokers publish the services and facilitate the matching and discovery of services. In distributed object architecture, there is no explicit separation of duties, and there are no external mechanisms for service publication and discovery.

- The Web service implementation of SOC makes use of the pervasive internet infrastructure to deliver the services, while allowing using local area networks to build private SOC applications using the same technologies and standards.

Multithreading is the basic distributed computing model, which allows the parallel computing units to be specified by the programmer at the function and class levels, which are executed as independent operating processes and are running on the same processor or on different processors, depending on the operating system's scheduling and dispatching. Communication, resource sharing, and synchronizations among the threads are managed by the programmer. Chapter Two will cover multithreading in detail.

1.3 Service-Oriented Architecture and Computing

1.3.1 Basic Concepts and Terminologies

A **service** is the interface between the service producer (or provider) and the consumer. The producer (also called provider) of a computing service is the person who develops the computer program (or the computer that runs or hosts the program) for others to use; while a service consumer is a person or a computer program that uses a service. From the producer's point of view, a service is a function module that is well-defined, self-contained, and does not depend on the context or state of other functions. These services can be newly developed modules or just modules wrapped around existing legacy programs to give them new interfaces.

From the application builder or service consumer's point of view, a service is a unit of work done by a service provider to achieve desired results. Different from an application, a **service** normally does not have the human user's interface. Instead, it provides Application Programming Interface (API) so that the service can be called (invoked) by an application or another service. For human users to use a service, a user interface needs to be added. A service with a user interface is an **application**.

The discovery of services by service consumers can be facilitated by service brokers. A service broker allows a service producer to publish their service definitions and interfaces, and at the same time allows a service consumer to search its database to discover the desired services.

An important feature of SOC is to divide the software development into three parties (stakeholders): service requesters or consumers, providers, and brokers. This three-party structure adds significant flexibility to the software system structure, and supports a new approach of software development: composition.

Service-Oriented Architecture (SOA) is a distributed software architecture, which considers a software system consisting of a collection of loosely coupled services that communicate with each other through standard interfaces, such as WSDL (Web Services Description Language) interface and via standard message-exchanging protocols such as SOAP (Simple Object Access Protocol). These services are autonomous and platform independent. They can reside on different computers and make use of each other's services to achieve their own desired goals and end results. Software in SOA should be developed and maintained by three independent parties, service requester (application builders), service brokers, and service providers. Service providers develop services and publish them in service brokers, while the service requesters

discover the services via service brokers using the available services to compose their applications. As the same services can be published by many service providers, the service requesters can dynamically discover new services and bind them into their applications at runtime, as better services are discovered.

Service-Oriented Computing (SOC) refers to the computing paradigm that is based on the SOA conceptual model. SOC includes the concepts, principles, and methods that represent computing in three parallel processes: service development, service publication, and application composition using services that have been developed. The essential difference between SOA and SOC is that SOA is a conceptual model that does not concern the algorithmic design and implementation to create operational software, while SOC involves a large part of the software development life cycle from requirement, problem definition, conceptual modeling, specification, architecture design, composition, service discovery, service implementation, and testing, to evaluation. As a result, SOA is more of the concern from the application builders (service requesters), while SOC is concerned by all three parties of the SOC software development.

Service-Oriented Development (SOD) refers to the entire software development cycle based on SOA concepts and SOC paradigm, including requirement, problem definition, conceptual modeling, specification, architecture design, composition, service discovery, service implementation, testing, evaluation, deployment, and maintenance, which will lead to operational software.

In the literature, SOA is often extended to include the meaning of the SOC, and thus, SOA and SOC are used interchangeably, particularly when the specific differences between SOA and SOC are not the concern of the discussion. On the other hand, SOC is often extended to include the meaning of SOD, particularly when the specific differences between SOC and SOD are not the concern of the discussion. Thus, in this book, we will use SOC for SOA and SOD as well, to simplify the use of terminology, if the differences among them are not the concern of the discussion.

Figure 1.8 illustrates the relationship between SOA, SOC, and SOD. The dotted circle shows the coverage of this book.

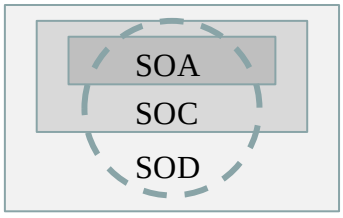


Figure 1.8 SOA, SOC, and SOD

We use "Distributed Service-Oriented Software Development" as the title of the book to contrast the widely used Distributed Object-Oriented Software Development approach, and to emphasize the fact that service-oriented software development is distributed in nature. Not only the software under development is distributed in different computers in different locations, but also the development process is distributed in the sense that the application builders, service brokers, and service providers are developers working independently in different locations, but following the same interfaces and standards. Furthermore, we have Chapter Two to discuss

distributed computing in general and how SOA, SOC, and SOD fit into the framework of general distributed computing.

Web services (**WS**) are services accessible over the Web. Web services-based computing is a specific implementation of SOC. It is perhaps the most widely known SOC example; however, other SOC implementations are also possible. Web services support SOC, and have a set of enabling technologies including WSDL, SOAP, and XML. XML is the standard for data representation; SOAP enables remote invocation of services across network and platforms. WSDL is used to describe the interfaces of services. UDDI (Universal Description Discovery and Integration) and ebXML (electronic business eXtensible Markup Language) are used to publish Web services, which enable publishing, searching, and discovery, manually and programmatically. More standards and protocols are being included in the WS technology set every day. Web services have several technical aspects:

- Services are functional building blocks. Multiple services can form a composite service and the composite service becomes a new building block. However, the code of a Web service does not need to be imported and integrated into the application. Instead, a service runs at the service provider's site and is loosely coupled with the application using messages. Thus, the service does not have to been written in the same programming language and does not have to be developed or running on the same platform.
- Services are software modules that can be identified by URL (Uniform Resource Locator) and whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts.
- Web services are often described by WSDL, accessed by the protocol SOAP over HTTP. With an added human interface, a single service or a composite service can form a Web application. Web services are normally accessed by computer programs, while Web applications are accessed by human users using a Web browser.

Composition is a key concept in SOC, which uses available services to compose a composite service or an application. Two composition methods are proposed and realized: Orchestration and choreography. In **orchestration**, a central process, which can be a service itself, takes control over the involved services and coordinates the execution of different operations. The involved services communicate with the central process only. Orchestration is useful for private business process. BPEL (Business Process Execution Language) is the major composition language that supports orchestration. In **choreography**, there is no central coordinator. Each service involved can communicate with any partners; choreography is useful for public business process and allows dynamic composition. WS-CDL (Web Services Choreography Description Language) is a composition language that supports choreography.

Service-Oriented Infrastructure (SOI): This term can have two meanings. The first meaning refers to the hardware and software support for SOC, as SOC involves many new kinds of operations not commonly used in traditional computing such as publishing, discovery, policy-based governance, orchestration, and choreography. For example, if the number of services is huge, the search algorithm needs to be efficient, with a good caching mechanism. Otherwise, a significant amount of time will be spent on discovery. Another example is the policy governance mechanism. As policies need to be enforced at runtime, the enforcement mechanism needs to be efficient and run at the real time as the application is running. As some

of the SOC operations can be quite expensive, it is quite logical that some of these operations should be executed by hardware or supported by hardware to save cost and time. This is particularly true if the SOC system needs to be used in mission-critical real-time systems.

Another meaning of SOI is that a hardware system can be organized in a service-oriented manner like a software system. An example of this kind of SOI is now being developed by Intel in their SOI group. The principal idea is to treat computing components, memory components, and networking components as virtual services. Essentially they are treating these hardware components as services like software services, and they control these hardware services like software services in a service-oriented manner. Intel calls this PaaS (Platform as a Service) so to compare the SaaS (Software as a Service) concept. In this way, a hardware system can be composed and re-composed like a software system, and managed like an SOC system. Another interesting implication is that once a hardware system is organized in an SOI manner, hardware is constructed as re-composable services, which allow hardware components to be replaced or upgraded without stopping the operation of the system. This means that current fault-tolerant computing techniques can be seamlessly integrated into the architecture design. This will be a research topic for the future.

Web 2.0 is the proposed next generation of Web or internet. The core concepts include users as *active* contributors (rather than just passive observers), peer collaboration, collective intelligence, moving the computing platform from desktop to the Web, user-centric computing, and service orientation. One well-known example is the Wikipedia, where millions of user participated in writing an online encyclopedia. This approach has been particularly successfully as the Wikipedia has become a popular way for people to learn. Note that the Wikipedia Company has only seven employees, yet it has produced millions of pages of knowledge, and almost all the knowledge is contributed by users. This is an excellent example how massive collaboration can create something that is of great value. This book has many citations to Wikipedia, which is a witness that the materials in the Wikipedia are indeed useful. The approach of conducting business using Web 2.0 is now called Wikinomics [http://en.wikipedia.org/wiki/Wikinomics]. Numerous organizations are now trying to duplicate this approach in creating something of great value.

Semantic Web. Semantic Web is defined by W3C, which provides a vision for the future of the Web. The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. The idea is to give information explicit meaning, to make it possible for Web services to automatically process and integrate information available on the Web. Semantic Web is now also called Web 3.0 [http://en.wikipedia.org/wiki/Web_3], as the name Web 2.0 has been used. However, Web 3.0 is currently more mature than Web 2.0 and may be formally deployed before Web 2.0. Semantic Web has been a popular research area for a number of years.

Ontology. The word "ontology" comes from philosophy, where it means a systematic explanation of being. In computer science, ontology is defined to be the formal specification of the terms and objects in a domain and the relationships among them. One of the principal relationships is classification. Often an ontology system defines a vocabulary of terms (words), their meanings (semantics), their interconnections (e.g., synonym and antonym), and rules of inference (reasoning), which is used in the semantic Web projects as the main means of implementation.

Service-Oriented Databases (SODB). As SOC became popular, the database technologies also become relevant. SOC applications use XML-based data and message, which have tree-structures, whereas traditional databases consist of tables of rows and columns. There are several approaches to address the mismatch between data structures.

The first approach is to use traditional databases and an adapter to convert the XML-based data and message to and from data of tables in the traditional databases. This is the current business practice in this area.

The second approach is to encode data in the XML format and store the XML files as database. The main challenge is to design and implement efficient XML-based query language to retrieve data from, and store data into, the XML database. The XQuery language has been defined by W3C to serve this purpose.

The third approach is to encapsulate the existing database management systems such as relational database systems as service, and develop related services so that an SOA application can talk to the database system. Those related services are called **information services**.

Ontology can also serve as a database for SOC applications. In fact, an XML database can be viewed as a simplified ontology system.

1.3.2 Service-Oriented Computing

In traditional software development paradigms, the developer takes the requirements, converts them into specification, and then translates the specification into the executable that meets the requirements. Several approaches are available to translate the specification into an operational system, including the waterfall model, incremental development, object-oriented computing (OOC), and component-based computing. Each approach has its own engineering processes and techniques.

SOC is a new paradigm that evolves from the OOC and component-based computing by splitting the developers into three independent but collaborative parties: the **application builders** (also called **service requesters**), the **service brokers,** (or **publishers**), and the **service developers** (or **providers**). The responsibility of the service developers is to develop software services with standard interfaces. The service brokers publish or market the available services. The application builders find the available services through service brokers and use the services to develop new applications. The application development is done via discovery and composition rather than traditional design and coding. In other words, the application development is a collaborative effort from the three parties.

Services are platform-independent and loosely coupled so that services developed by different providers can be used in a composite service. Many standards have been developed to ensure the interoperability among services. However, the competition is fierce. Only the best services can survive because, for a given known service requirement, for example, password encryption and “add-to-cart” services, many providers can implement and publish the same service for application builders to use in their applications.

In SOC, individual services are developed independently based on standard interfaces. They are submitted to service brokers. The application builders or service requesters search, find, bind, test, verify, and execute services in their applications dynamically at runtime. Such a

service-oriented architecture gives the application builders the maximum flexibility to choose the best service brokers and the best services. Figure 1.9 shows a typical service-oriented architecture, its components, and the process of registering and requesting a service. The components and steps shown in the diagram are explained as follows:

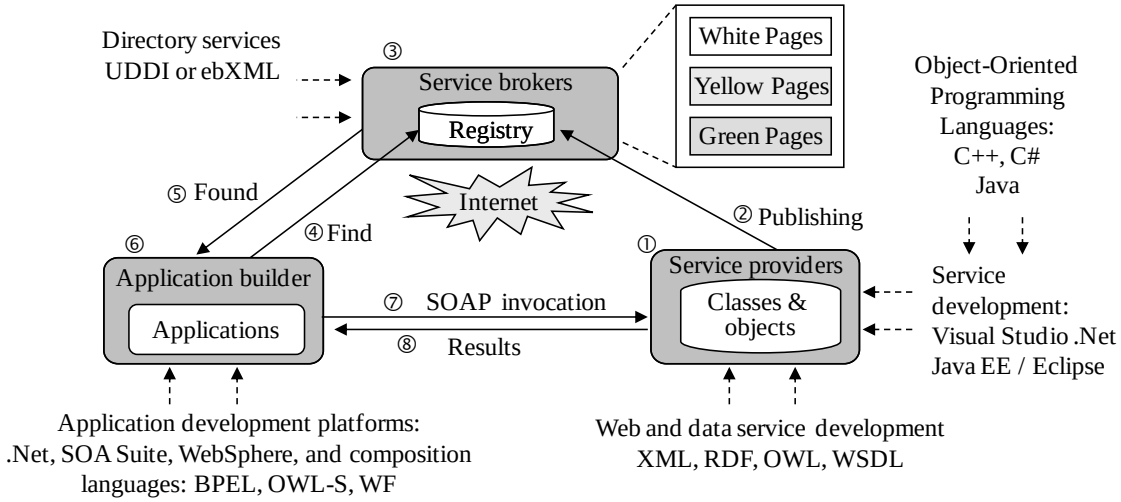


Figure 1.9 A typical service-oriented architecture

1. The Web services providers develop software components, corresponding to classes and objects in OOC to provide different services using programming languages such as C++, C#, and Java. Service-oriented software development environments like .Net, J2EE, and the Eclipse.
2. The service providers register the services to a service broker and the services are published in the registry.
3. Current service brokers use UDDI or ebXML standards that provide a set of standard service interfaces for registering and publishing Web services. For UDDI, the information needed for registering a service includes: (1) White Pages information: Service provider's name, identification, for example, the DUNS number, and contact information. (2) Yellow Pages information (business category): industry type, product type, service type, and geographical location. (3) Green Pages information: technical detail how other Web services can access (invocate) the services, such as APIs (Application Programming Interfaces). UDDI's White and Yellow Pages are an analogy to the telephone White and Yellow pages. The UDDI standard supports directory only, while and ebXML supports both directory and repository.
4. An application builder looks up, through the Internet, the broker's service registry, seeking desired services and instructions on how to use the services. The ontology and standard taxonomy in the service broker can help automatic matching between the requested and registered services.
5. Once the service broker finds a service in its registry, it returns the service's details (service provider's binding address and parameters for calling the service) to the application builder.

6. The application builder uses the available services to compose the required application. This is higher level programming using service modules to construct larger applications. In this way, the application builders do not have to know low-level programming. With the help of an application development platform, the application code can be automatically generated based on the constituent services. The current application development platforms include like .Net, J2EE, SOA Suite, ActiveBPEL, and WebSphere from IBM, which can support high-level composition of applications using existing services.
7. The code of services found through a broker resides in a remote site, normally in the service provider's site, or in the service broker if service repository is offered by the broker. SOAP invocation can be used to remotely access the services.
8. The service in the service provider's site directly communicates with the application and delivers service results.

1.3.3 Object-Oriented Computing versus Service-Oriented Computing

SOC is different from Object-Oriented Computing (OOC) in many ways, even though SOC evolves from OOC, and they may look similar. In the past, some mistakenly thought that OOC is not much different from procedural computing, because traditional procedural languages already have the concept of data abstraction such as structure, which is similar to class, and procedures, which is similar to methods. Even though OOC may look similar to traditional computing, the fact that designers think in terms of classes and objects fundamentally change their way of thinking. As a result, many new concepts and methods emerge in OOC, such as design patterns, inheritance, dynamic binding, polymorphism, design hierarchy, and UML (Unified Modeling Language).

Similarly, SOC is different from OOC, because now designers will think in terms of services, workflows, service publishing, discovery, application composition using reusable services, and policy governance. These concepts are indeed different from OOC.

Furthermore, services can be available on the web or in a private repository, and an application can use runtime search to discover new services and bind the service into the application. The application builder may not need to buy and install the **service component** (the software that provides the service); instead, the application can access the service component remotely and pay for the service used. Software upgrade will become easy, because once the service components are upgraded, the new services will be immediately available to the applications, saving significant cost of un-installing and re-installing software on client computers. Software will be charged based on the extent of use. Thus, users will not have to pay for unnecessary software. In other words, SOC provides a new model of software application, instead of buy-install-and-use, SOC provides a new model of use-and-pay.

The SOC also has a significant impact on the system structure, dependability attributes, and mechanisms, such as system reliability, security, system reconfiguration, and re-composition. These mechanisms will be drastically different from OOC. For example, instead of static composition (with dynamic creation of objects and dynamic binding) in OOC, SOC allows dynamic composition in real time and at runtime using services just discovered, and with knowledge of the service interface only. Because new services will be discovered at runtime, SOC also needs a runtime ranking and selection mechanism based on runtime interoperability

evaluation, testing, and other criteria. In case of system failures or requirement changes, the SOC also needs a distributed reconfiguration and re-composition strategies. Such strategies will be rather different for OOC.

In OOC, it is necessary to develop the code manually, even though some forms of dynamic binding can be used. The current OOC dynamic binding mechanism allows polymorphism, that is, methods that belong to a family of classes can replace each other at runtime. Yet SOC allows an unrelated service to replace an existing service as long as the new service has the same WSDL specification.

In SOC, a faulty service can be easily replaced by another standby service by a DCS (Dynamic Composition Service). The DCS is also a service that can be monitored and replaced. The key is that each service is independent of other services, and thus, replacement is natural. Only the affected services will be shut down. This approach allows the mission-critical application to proceed with minimum interruption.

Although SOC shares certain concepts and technologies with OOC, such as component design and component reuse, the innovation in SOC is significant. Figure 1.10 contrasts the main technologies and the development methodologies between the two paradigms.

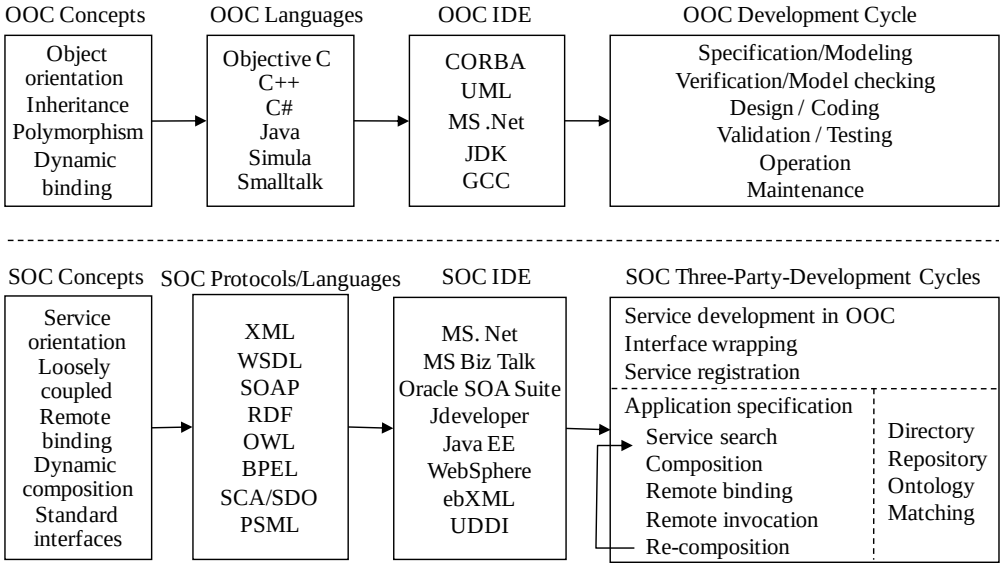


Figure 1.10 OOC and SOC concepts and technologies

Table 1.1 elaborates the comparison between OOC paradigm and SOC paradigm in terms of major features in the software development process.

1.3.4 Service-Oriented Enterprise

A **Service-Oriented Enterprise** (SOE), proposed by Intel researchers and standardized by OASIS, is a stack of technologies that implement and expose the business processes through an SOA system. SOE provides a framework for managing the business processes across an SOA landscape. At its core, the SOE is a system structure that supports core enterprise computing. An enterprise is not just an individual system. In fact, it is more than all the systems within a business unit, but across a large corporation. For example, a computer system for an army unit

in a given state is not an enterprise system, but a DoD (Department of Defense) system that controls and commands a major DoD function is an enterprise system. A supply chain system for a major retail store, such as Wal-Mart or Target, is another example of an enterprise system. Thus, an enterprise system is much larger than an individual system, and it may consist of hundreds of systems residing in multiple states or nations. A SOE is a system that supports the enterprise-wide operations.

Table 1.1 OOC versus SOC

Features	Object-Oriented Computing	Service-Oriented Computing
Methodology	Many methodologies are available to develop OO programs.	In addition, SOC involve service discovery, architecture, application composition, and software monitoring.
Cooperation among developers	Development is by a single team responsible for entire life cycle. Cooperation is among software engineers working on require-ment, designers, coding, and QoS.	Development is delegated to three independent parties: application builder, service provider, and service broker. Cooperation is among these three parties.
Abstraction	Abstract data type (class) and encapsulation of data and methods within a program.	Abstraction is at the service (including workflows) and architecture levels.
Code reuse	Inheritance allows code reuse within one application or within one platform. OO design patterns and application frameworks can be used to promote software reusability.	Services can be shared to promote reusability. Service brokers with ontology information enable systematic sharing of services.
Dynamic binding	Associating names to variables and methods at runtime.	Can dynamically allocate remote service required through the service directory.
Re-composition	Often it is necessary to determine and import the components at design time.	Can remove remote services, and find and add newly available services through the service directory.
Component communi-cation and interface	Importation of component code and integration at compilation time. Often this is platform and language dependent.	Remote invocation without importing the code. Platform and language independent. Open standard protocols ensure interoperability from different vendors.
System maintenance	Users need to maintain and/or upgrade their hardware and software regularly.	Hosting software needs to be maintained by provider, but services may be maintained by third parties.
Reliability	Software reliability can be obtained via testing and	Application reliability depends on the reliability of application software as well as

	reliability modeling. Fault-tolerant software can be designed with redundant components.	the reliability of services used, and possibly also the reliability of involved service brokers. Software reliability can be obtained with collaboration and contributions from all the parties. Fault-tolerant software can be designed with redundant services.

As an enterprise-wide system, the traditional elements of SOA, that is, searching, discovery, interfacing, and service invocation, are not the focus of SOE, even though they are the common elements shared by the participating systems. These elements describe how to construct services and how to use services. They do not describe how sets of services support enterprise business processes or how atomic services function within an enterprise.

The central challenge facing the SOE is to design service-oriented business processes within an enterprise in such a way that the process is visible and manageable end-to-end. As the number of services available within the enterprise increases, the execution pattern becomes increasingly difficult to define and to track. An SOE is still a relative young research area within SOC, which itself is a young discipline at this time.

Figure 1.11 shows an example of the layers in an SOE with composite e-business applications and Web services as its foundation. The top layer of SOE is the configurable business logic. The next layer is the ebSOA (SOA for electronic business), which is a standard for service broker, including both registration and repository. The next layer is the Service-Oriented Management (SOM), which implements the non-functional features such as fault-tolerant computing, reliability, security, and policies. Service-Oriented Infrastructure (SOI) provides virtual services that represent the services that can be provided by hardware components. For example, Intel is developing this layer to map its hardware layer resources, including computing resources, memory resources, networking resources, devices, sensors, and actuators, to the service-oriented above architectures. The bottom layer is the hardware devices that perform the required tasks.

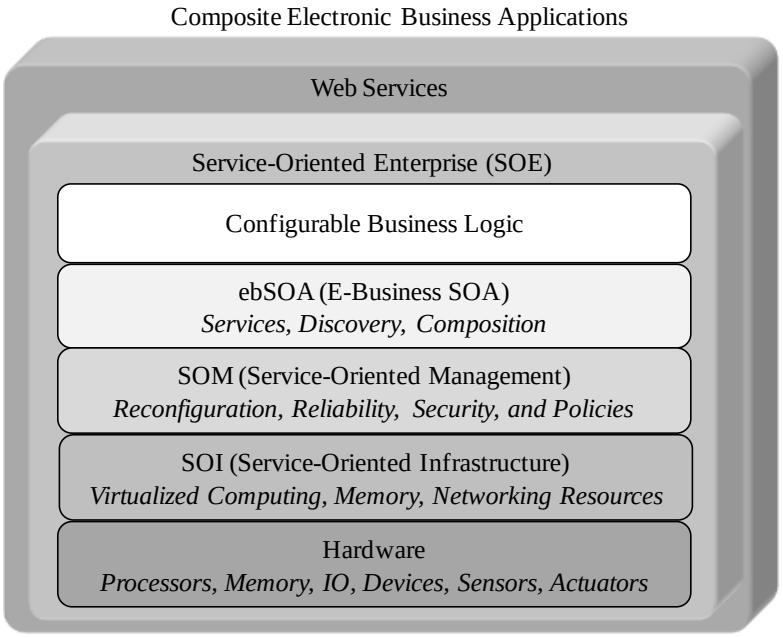


Figure 1.11 SOE Framework

1.3.5 Service-Oriented System Engineering

Service-Oriented System Engineering (SOSE) is a combination of system engineering, software engineering, and service-oriented computing. It suggests developing service-oriented software and hardware under system engineering principles, including requirement, modeling, specification, verification, design, implementation, testing (validation), operation, and maintenance. Current research and practice on SOC are largely focused on functionality and protocols of SOC software. As SOC moves into mission-critical applications, as well as the entire computing and communication infrastructure moves to SOC, SOSE issues need to be addressed.

Table 1.2 lists typical SOSE techniques in each development phase. Many of the techniques are collaborative. For example, test cases may be contributed in a collaborative manner by all three parties. The service provider can provide sample unit test cases for the service broker and service requestors to reuse. The service broker can provide its own test cases via specification-based test case generation tool, and the broker may even make the tool available for all the parties. The application builder can examine the sample test cases by the service broker, apply the test case generation tool provided by the service broker, and even contribute its own application test cases.

Table 1.2 Different SOSE techniques

Development phase	SOSE techniques
Collaborative Specification & modeling	Service specification languages, model-driving architecture, ontology engineering, and policy specification.
Collaborative Verification	Dynamic completeness and consistency checking, dynamic model checking, and dynamic simulation.
Collaborative Design	Ontology engineering, dynamic reconfiguration, dynamic composition and re-composition, dynamic dependability (reliability, security, vulnerability, safety) design
Collaborative Implementation	Automatic system composition and code generation
Collaborative Validation	Dynamic specification-based test generation, group testing, remote testing, monitoring, and dynamic policy enforcement
Collaborative Run-time Evaluation	Dynamic data collection and profiling, data mining, reasoning, dependability (reliability, security, vulnerability, etc) evaluation
Collaborative Operation and Maintenance	Dynamic reconfiguration and re-composition, dynamic re-verification and re-validation

Even though we mainly use software to illustrate SOSE, the same can be applied to hardware and networks. Major computer companies are developing SOI and SON (Service-Oriented Networks) to support SOC applications at this time. They will need to develop the related SOSE techniques.

While the basic engineering principles remain the same, the way they are applied will be different in the SOC paradigm. Specifically, most engineering tasks will be done on the fly at runtime in a collaborative manner. Because systems will be composed at runtime using existing services, many engineering tasks need to be performed without complete information and with significant information available just in time before application. In this way, SOSE in some way may be drastically different from traditional engineering where engineers have complete information about the system requirements and thorough analyses can be performed even before system design is started.

SOC is a new paradigm for computing and thus new engineering techniques need to be developed to make SOC software and hardware dependable, reliable, safe, and secure. SOSE techniques are different from traditional system engineering techniques even though the basic engineering principles such as mathematics remain the same. Due to the dynamic features such as runtime composition and re-composition, new applications may not be evaluated by traditional system engineering because many components may be dynamically discovered and composed, and their source code may not be available. Thus, dynamic runtime system engineering techniques need to be applied.

1.4 Service-Oriented Software Development and Applications

1.4.1 Traditional Software Development Processes

Software development processes define the steps of development that leading to high quality software. Several processes have been proposed and applied, including waterfall, iterative, object-oriented, and component-based development processes. Object-oriented and component-based software development processes are similar; Figure 1.12 shows a possible process. Both development processes require to decompose the system to be developed into components, to develop the code of the components first, and then to use the components to build the applications. Object-oriented development process is a more specific approach than the component-based approach, which is defined by a set of specific features, such as encapsulation, inheritance, polymorphism, and dynamic binding. General speaking, object-oriented development is certainly component-based. However, component-based development may or may not be object-oriented.

1.4.2 Service-Oriented Software Development

Traditional computing paradigms affect mainly the design (algorithms) and implementation (programming) phases in the software development process. SOC affects the entire software development process as well as the cycle of the software. To better understand the impacts, let us first examine the unique features of SOC software:

- Self-contained and self-describing: Services are published through service brokers and the published services contain sufficient information for other services to discover, match, bind, and invoke remotely and at runtime.
- Reconfigurable and Re-composable: A newly discovered service can be composed into an existing service in two different ways: reconfiguration and re-composition.
- Reconfiguration: An existing service can be replaced by a new service satisfying the same function specification. Reconfiguration is performed when a service is faulty or becomes unavailable.
- Re-composition: In a SOC system, the user could change the specification of a service at runtime theoretically, resulting a re-composition; during which, new services could be included in a composite service and existing services could be excluded.
- Dynamic verification: The dynamically modified specification must be dynamically verified to assure the required properties of the specification.
- Dynamic validation: The dynamically reconfigured or re-composed service must be dynamically validated (tested) to assure it meeting the specification.
- Dynamic evaluation: The dynamic reconfiguration and re-composition may lead to structural change of a service and the attributes (reliability, security, safety, and performance) must be dynamically evaluated.

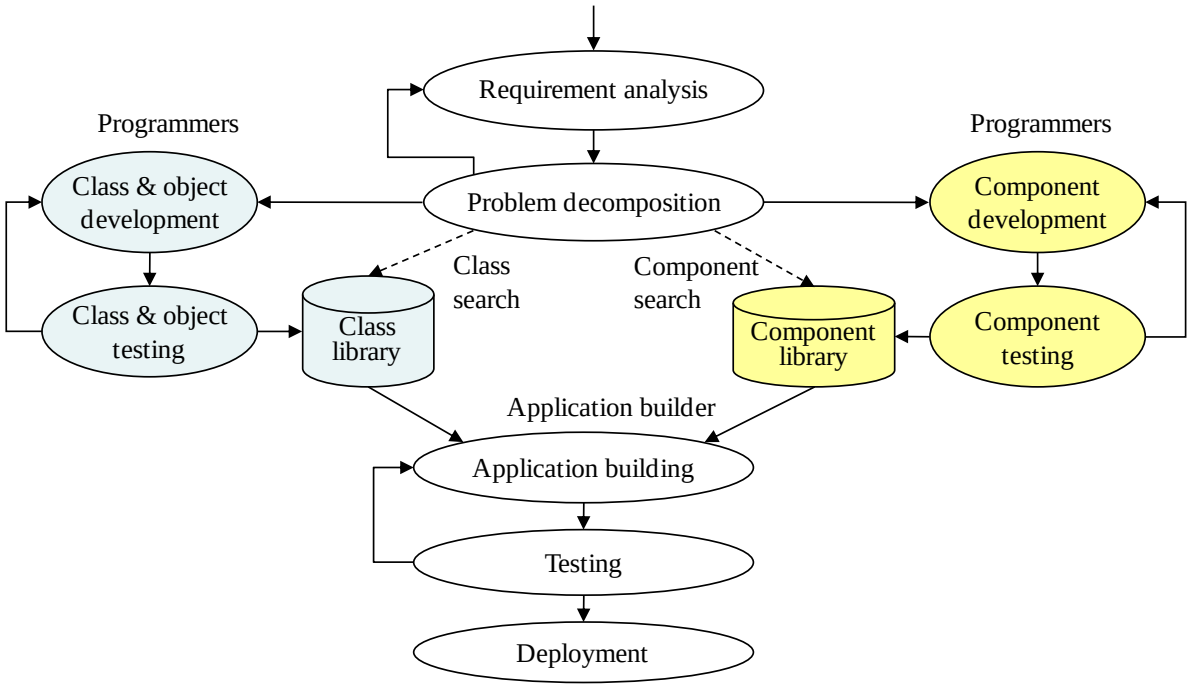


Figure 1.12 Object-oriented and component-based software development processes

In traditional software development process, the entire process is often managed by the same organization of developers. The new service-oriented software development is divided into three parallel processes: Service development, service publishing into the service brokers, and application building (composition).

The services are of two kinds: atomic and composite. An atomic service is an object with standard interface. Thus, the development of atomic services is not much different from that of the object-oriented software development. The main difference is that an object normally needs to be integrated into the application written in the same programming language; whereas an atomic service can reside on a remote computer and can be invocated by applications written in different programming languages. Thus, the interface of an atomic service must be designed following certain predefined standards. The interface must contain the description of the functions of the service and the technical detail of invocating the service, so that the service can be discovered and can be properly invocated by other programs. WSDL (Web Service Description Language) is major languages used to describe the interfaces of services and SOAP (Simple Object Access Protocol) is used to transport messages between services. An atomic service can either be developed from scratch or be a wrapped service from an existing software component.

The development of composite services is different from that of traditional software development process. Although traditional software development allows the construction of larger components from smaller components, the construction is static and manual. The construction of composite service can be static and manual. However, it can also be dynamic and automatic, that is, a service can be composed at runtime when a required service does not exist and need to be composed from the existing services. Existing services include those services that are published through service brokers. Once a service is composed, the composite service can be published as a new service for future service or application composition. An SOC

application has little difference from a composite service. The former has a GUI for human users to access, while the latter has programmatic interfaces exclusively for computer programs (applications or services) to access.

The development processes in OOC and in SOC are elaborated in Figure 1.13. Typically, an OOC application is developed by the same team in the same language (as shown on the left part of the figure), while an SOC application is developed by using pre-developed services developed by independent service providers. To find the required services, the application builder looks up the service directories and repositories. If a service cannot be found, the application can publish the requirement or develop the service in-house. Service providers can develop services based on their own requirement analysis, or look up the requirement published in the directories.

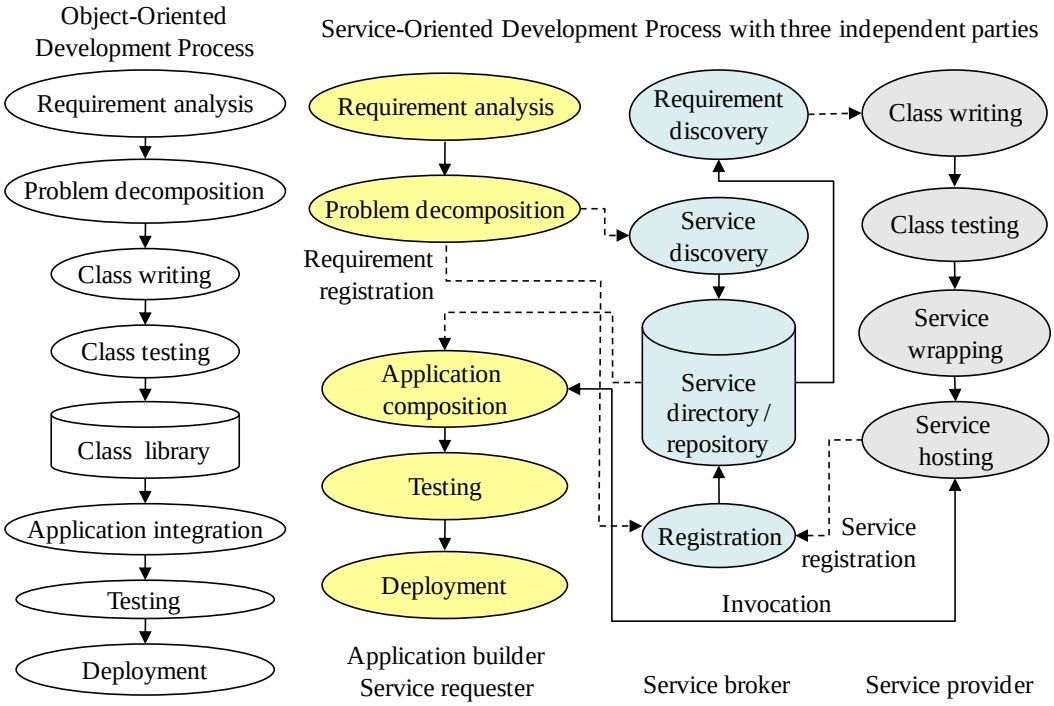


Figure 1.13 Object-oriented versus service-oriented software development process

Like traditional software development, SOC software development process starts with the requirement analysis and definition. Figure 1.14 shows a typical requirement definition steps. At the end of the requirement, the system to be developed will be more formally modeled and specified in a modeling and specification language.

The rest of the application building process is significantly different from the traditional software development. Application builders use the existing services published by service brokers to build application. In this process, the application builder can focus on their business logic, instead of programming tasks. If the existing services cannot meet an application's functional requirement, the application builder can construct a composite service to meet the requirement. Figure 1.15 outlines the steps of software composition process from application builder's perspective.

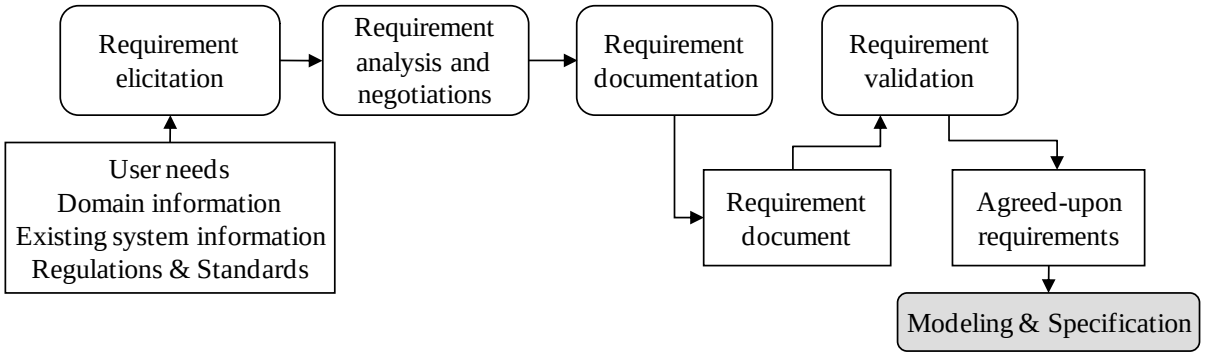


Figure 1.14 Requirement Development Process

In Figure 1.15, we separate the data and ontology specification from the functional specification. In SOC, to facilitate the dynamic composition and re-composition, it is recommended to separate data such as policies, rules, and configuration parameters from the functional specification. Storing these data in an ontology or a configuration file allows them to be modified and to take effect at runtime without stopping the program. Policy-based computing is a good example of such separation.

The functional specification and data/ontology specification are verified using traditional verification techniques, such as model checking. Test cases can be generated from the specifications based on either the functionality or process flow in the specification.

Once the workflow is verified, the remote services need to be discovered or developed separately if no existing services are available. Once all services are bound into the workflow, the workflow becomes executable in the given environment, such as a simulation environment. The application will be tested in the simulation environment before being deployed into the field environment or a more realistic environment in which execution data can be collected for various analyses. If semantic information, such as policies, is stored in the ontology, the execution can be validated by the ontology or the policies. Based on the validation and evaluation, the system can be reconfigured by binding to different services at runtime. The requirement can be revised too. In this case, the system needs to be stopped to manually revise the models and specifications.

1.4.3 Applications of Service-Oriented Computing

As a general-purpose computing paradigm, SOC can be applied in any domains where OOC can be applied. Especially, OOC can be considered as a part of SOC. Every OOC application can be theoretically considered as an SOC application. However, in many situations, SOC provides unique advantages.

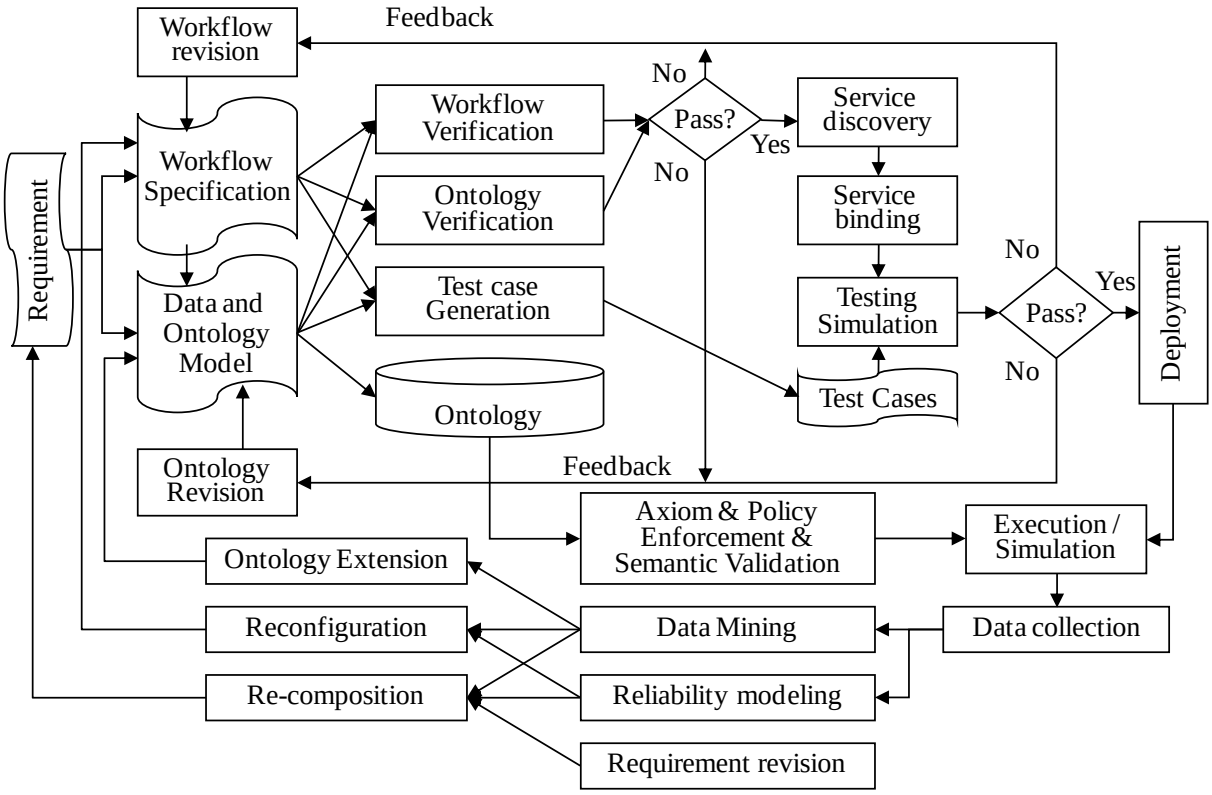


Figure 1.15 Service-Oriented Application Development Process

Electronic business has been the stronghold of SOC, where many services are dynamic and have to be remote and over the internet. For example, a travel agency has to remotely invoke the services offered from the airliners, hotels, and car rentals. It is not doable to import the code of the services into the local server of the travel agency. Similarly, building an online bookstore requires to access the services from multiple parties, including banks, publishers, and freighters. The other emerging application areas include banking, healthcare, and e-government, where the services from different divisions are loosely coupled to provide collaborative services to their customers.

Robotics and embedded computing are traditional application fields where control programs are an integral part of the device. The introduction of SOC into this field makes it more flexible in accomplishing the mission of a robot or an embedded system. Instead of preloading the entire control program to the system, parts of the programs are implemented as remote services. The modification of the remote services can change the behavior and the course of the application without interrupting its execution. This feature is particularly attractive because the robot or the embedded system may have been in a location that is not physically reachable.

Many manufacturing processes today are controlled by computers. The introduction of SOC software in the processes makes the modification of the process much easier and more efficient.

Figure 1.16 shows a part of the SOC research and application projects at Arizona State University. The development of SOC software and hardware is the core of the research and applications. Concepts, principles, models, techniques, methods, tools, and frameworks have been developed to support the applications in a number of areas, including e-business, industrial

process control, command and control, embedded systems, robotics, bio/medical information system, and ontology-based education systems.

Many of the topics will be covered in this book, not only at the conceptual level, but also at the development and implementation levels.

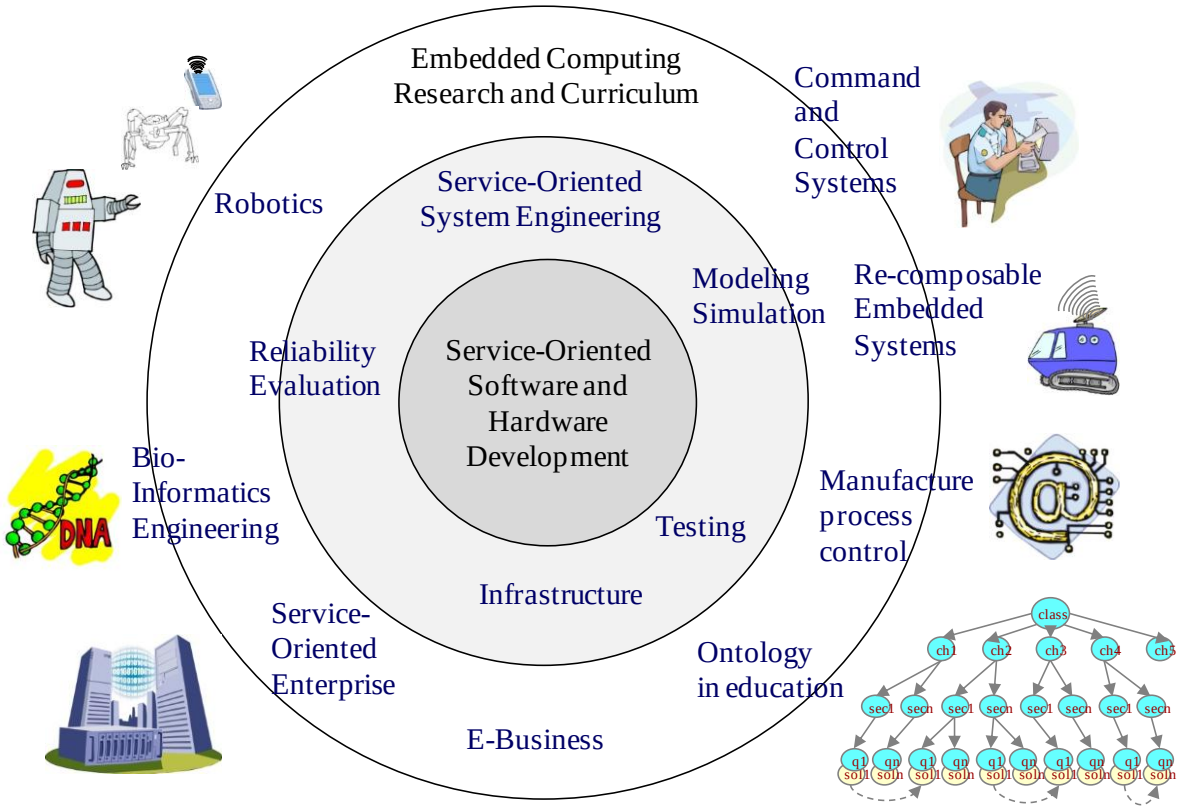


Figure 1.16 SOC research and application at Arizona State University

1.5 Discussions

While SOC/SOA has been under developed for the last ten years and has been adopted by all major computer and software companies such as BEA, HP, IBM, Microsoft, Intel, Oracle, Sun Microsystems, and SAP, as well as government agencies such as US Department of Defense, British Healthcare System, multiple Canadian provincial governments, and the State of Arizona. Many felt that SOA is relatively young and much work is needed. Specifically, SOA critics have pointed out several issues for improvement. For example, one issue is that SOA lacks of a commonly agreed definition. Some people felt that SOA is not well defined and thus it is difficult to characterize SOA. For example, at Wikipedia, the following definition is stated for SOA [http://en.wikipedia.org/wiki/Service-orientation]:

- **"Service-oriented Architecture** (**SOA**) is an architectural design pattern that concerns itself with defining loosely-coupled relationships between producers and consumers. While it has no direct relationship with software, programming, or technology, it is often confused with an evolution of distributed computing and modular programming."

This definition is not good enough for SOA, because this description also fits OO computing. An OO program can also be loosely coupled. In fact, loose coupling is one of the principal attributes of OO software. Furthermore, OO computing can be distributed computing and certainly it is one of common modular programming techniques. Some key SOA attributes, such as separation of definition from implementation, have also been used in OO software, as a class interface definition has been separated from its implementation. In fact, the concept of separating definition from implementation has been attempted for over 30 years in the computing history, including data abstraction and procedural abstraction. Thus, this concept is certainly not new or unique. No wonder that in the same page, the collective authors state:

- "There is no widely agreed upon definition of SOA other than its literal translation. It is an architecture that relies on service-orientation as its fundamental design principle. In an SOA environment, independent services can be accessed without knowledge of their underlying platform implementation. These concepts can be applied to business, software and other types of producer/consumer systems."

In other words, even thousands of authors around the world who are active in SOA could not agree on the SOA definitions, as Wikipedia is edited and contributed by their active readers.

Some SOA definitions are are based the common SOA protocols used. For example, if a software program uses XML, WSDL, OWL, BPEL and/or other protocol or standards, then it is an SOA software. This definition is still not good enough, because these SOA protocols are constantly being updated and revised. It is even possible that later versions of these protocols will have little resemblance to previous versions, as the SOA history certainly can testify that several SOA protocols have been completely replaced by newer protocols. Specifically, BPEL has replaced several SOA composition languages before.

Some SOA authors also use SOA properties as definitions. However, this is not good enough either, specifically, because some often touted SOA properties are actually not available at this time. For example, dynamic composition is often an important characteristic of SOA. However, this feature is not available in a practical SOA environment yet. In other words, it is still a research topic. Most of the SOA tools today actually use *static* composition, that is, selecting services at the design time rather than at runtime dynamically. Thus, defining SOA by dynamic composition is not appropriate at this time. Furthermore, as SOA progresses, other SOA characteristics will emerge, defining SOA by current SOA properties will prove to be too restrictive.

Some defines SOA software as a collection of services. However, this definition is too loose. If so, what is the definition of a service? Does a service have a state? Is a service passive, autonomous, thin, or fat? Some people say that a service should be a *fat* service, that is, a service has many supporting facilities and tools, and it can be even more autonomous like a software agent. This definition looks interesting and makes a software service more intelligent and probably more useful than a traditional "passive" service. However, this definition actually makes the current SOA infrastructure almost invalid, as they do not support "intelligent" services yet. The current SOA infrastructure does not support those common SOA operations such as composition, deployment, governance, modeling and interoperability, on this kind of "intelligent" or autonomous services. Unless a new SOA infrastructure framework is developed, it is difficult to support those autonomous services using the current SOA infrastructure.

We prefer the definition from OASIS. According to the SOA reference model specification, SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with, and use capabilities to produce desired effects consistent with measurable preconditions and expectations. The SOA reference model specification bases its definition of SOA around the concept of "needs and capabilities," where SOA provides a mechanism for matching needs of service consumers with capabilities provided by service providers.

OASIS also has a definition of service. A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. Moreover, a service has service description, visibility, interaction, real-world effect, execution context, and contract and policy. However, this definition is too loose, because it can fit a passive or thin service, as well as fat and intelligent service.

Using these definitions, SOA approach essentially allows a person to publish software components following some standards, and allows others to discovery and reuse. Note carefully that the above definition does not say that only software services can be published and discovered. In fact, numerous things such as workflows, collaboration templates, application templates, data, data schema, policies, test scripts, and user interfaces can be published, discovered, and reused by others, as listed in Table 1.3.

Table 1.3 SOA Publishable Items

Reusable artifacts	Description
Methods (or Services)	Basic building blocks in SOA, and allows software development by composition.
Workflows	They specify the execution sequence of a workflow with possibly multiple services. They allow rapid SOA application development.
Application templates	These specify entire applications with their workflows and services. They allow rapid SOA application development.
Data, data schema, and data provenance	Data and associated data schema such as messages produced during SOA execution can be published and discovered.
Policies	Policies are used to enforce SOA execution and can be published for reuse.
Test scripts	Consumers, producers, and brokers can publish test scripts to be used in verification by other parties.
Interfaces	GUI design can be used and linked at runtime to facilitate dynamic SOA application with changeable interfaces.

Thus, potentially SOA can publish and reuse not only software services, but also other software artifacts such as workflow, policies, and data. Let us attempt a working definition of SOA:

- An SOA is an approach for software construction, verification, validation, maintenance, and evolution that involve specification, implementation, and publication of software artifacts such as services, workflows, collaboration patterns, and application templates following certain open interoperability standards. This approach develops software by composition with reusable software artifacts.

This working definition excludes an agent to be a service, but allows centralized and distributed SOA, as well as code to be mobile. This definition allows various web service protocols to be used as a part of open interoperability standards, but it does not mention any specific protocols. In this way, all kinds of protocols, including future protocols, can be included as a part of SOA. Thus, various open interoperability standards for service specification (such as WSDL), workflow language (such as BPEL), and collaboration specifications (such as CPP/CPA) can be used. At the same time, as these standards can be updated or even replaced in future, while the working definition does not need to be updated. Of course, the working definition of SOA can be updated and be changed from time to time, as we understand SOA more in the future.

Many outstanding books and papers that cover SOA are now available. Most of them are more suitable for working professionals. The standard organizations OASIS and W3C have developed most SOA related standards and reference models. Furthermore, as SOA is started mainly from computer industry, instead of from academia, one should search and navigate the SOA websites from the major industry players, the most notable ones including BEA, HP, IBM, Microsoft, Oracle, SAP, and Sun Microsystems. Readers can also find a large amount of SOA materials at DoD sites and DoD conference proceedings as DoD is one of the earliest adopters of SOA. Many DoD engineers and contractors have worked on SOA and they have gained significant experience. Due the relative youth of SOA, many concepts and ideas are expressed in white papers or web blogs.

Many universities around the world (mainly in Asia, Australia, America, Europe) also offer SOA courses, but as SOA is a wide area, different topics are actually covered in them. Most of these classes have offered their materials on the Web and readers can search their websites for information.

1.6 Exercises and Projects

Name: _____

Date: _____

1. Multiple choice questions. Choose one answer in each question only. Choose the best answer if multiple answers are acceptable.

1.1 Which of the followings are fallacies of distributed systems?

(A) Latency is zero.

(B) Bandwidth is infinite.

(C) The network is secure.

(D) Topology doesn't change.

(E) All of them of fallacies.

1.2 Generally speaking, a service is an interface between the

(A) service provider and the service broker.

(B) service requester and the service broker.

(C) Yellow Pages and the Green Pages.

(D) producer and the consumer.

1.3 What architecture is a tiered architecture?

(A) Client-server architecture

(B) CORBA

(C) Service-oriented architecture

(D) DCOM

1.4 What concept is least related coding?

(A) Service-oriented architecture

(B) Service-oriented computing

(C) Service-oriented software development

(D) object-oriented programming

1.5 What entity does not belong to the three-party model of SOC software development?

(A) Service provider

(B) Service broker

(C) Application builder

(D) End user of software

1.6 What is the most significant difference between the Distributed Object Architecture (DOA) (for example CORBA and DCOM) and the Service-Oriented Architecture (SOA)?

(A) SOA software has better modularity.

(B) SOA software does not require code-level integration among the services.

(C) DOA software has better reusability.

(D) DOA software better supports cross-language integration.

1.7 What concept is lest related to the application composition?

(A) BPEL

(B) choreography

(C) orchestration

(D) Code integration.

1.8 XML is

(A) an object-oriented programming language.

(B) a service-oriented programming language.

(C) a database programming language.

(D) a standard for data representation.

1.9 What protocol enables remote invocation of services across network and platforms?

(A) XML

(B) SOAP

(C) WSDL

(D) UDDI

1.10 Which of the followings is/are the proposed features of Web 2.0?

(A) Software as operational services.

(B) Users are treated as co-developers.

(C) Use loosely coupled and easy-to-use services to compose applications.

(D) Use services and data from multiple external sources to create new services and applications.

(E) All above

2. What are SOA, SOC, SOD, SOE, SOI, and SOSE? Briefly state their definitions based on your understanding.

Answer:

3. What are the main differences between requirement analyses in the OOC paradigm and in the SOC paradigm?

   Answer:

4. What are the major benefits of separating an application builder from the service providers?

   Answer:

5. What are the main techniques in SOSE (service oriented system engineering)? For each technique, write one or two sentences to describe its purpose.

   Answer:

6. Compare and contrast the traditional software development process and the Service-oriented software development process. For each step of the development, write a paragraph to describe the purposes, responsibilities, functions of the step.

   Answer:

7. What is a service registry? What is a service repository? What are their differences?

   Answer:

8. An electronic travel agency needs to be developed. What is your responsibility if you are:

8.1 a service provider?

   Answer:

8.2 a service broker?

   Answer:

8.3 an application builder?

   Answer:

9. You plan to invent a unique online game.

9.1 Describe what you must do as an application builder and what you can expect the service providers to do for you.

9.2 Describe your invention idea and list everything you must do as an application builder.

9.3 List everything that you can possibly find through service brokers.

Answer:

10. List a few application areas where you believe SOC is a better fit than OOC. State your reasons and justifications.

Answer:

11. What are the impacts of SOC paradigm to the IT market and to computer science graduates?

Answer:

12. This is an open problem. Search on the Internet to find a Web service testing tool. Download their reports and white papers; and write a half page summary about the tool.

A Service-Oriented Computing Workshop

As SOC is a young discipline, students will learn a great deal by doing their own research on SOC. One way to facilitate the research is to organize a workshop within the class. Specifically, each student needs to submit a paper to the workshop organized by the instructor and the teaching assistants. A sample call-for-papers is given below.

CALL FOR PAPERS

Workshop on Introducing Service-Oriented Computing (WISOA)

Scope – Workshop on Introducing Service-Oriented Computing (WISOA) serves as an initial meeting for participants of distributed service-oriented software development course at Arizona State University to exchange results and visions on all aspects of Service-Oriented Computing (SOC), Service-Oriented Architecture (SOA), and Service-Oriented System Engineering (SOSE). Starting with this new paradigm and their realization in Web Services (WS), WISOC covers all areas related to architecture, semantics, language, protocols, dependability, reliability, security, discovery, composition, publishing, testing and evaluation, interoperability, business process, as well as the deployment and experience of real service-oriented systems.

Topics of Interests – WISOC invites state-of-the-art survey submissions on all topics related to service-oriented computing, including (but not limiting) to the followings:

- Service Orientation Concepts and Definitions
- Service Modeling and Specification
- Service Requirements Engineering
- Service Semantics and Ontology
- Services and Business Processes
- Services, Components, and Agents
- Service-Oriented Design Patterns
- Service-Oriented Development Processes and Methods
- Service Publishing, Discovery, and Invocation
- Service Composition, Interoperability, Coordination, Orchestration, and Chaining
- Service Reputation and Trust
- Intelligent Selection, Service Brokering, and Service Level Agreement and Negotiation
- Services and Legacy Systems
- Service-Oriented Enterprise Architecture
- Service-Oriented System Implementation and Deployment
- Service-Oriented Verification, Testing, and Evaluation
- Service QoS, Dependability, Reliability and Performance
- Service Policy Management
- Service Infrastructures
- Service Privacy, Confidentiality, and Security
- Service Oriented Real-Time and Embedded Systems
- Service on Peer-to-Peer Network
- Service-Oriented embedded systems
- Service on Grid Network

This project consists of the following activities. The total number of points each student can obtain is 100. Ten percent of the papers will receive 10 bonus points as the best paper award.

1. The paper: 80 points

The points will be awarded based on the instructor's evaluation, as well as the peer evaluation, according to following evaluation questions, with 10 points for each question:

1) The paper is relevant to one of the focus areas given in the call for papers.
2) The paper has well defined questions to address, and the materials are coherent and consistent.

3) The paper clearly presents the ideas, and is easy to read.
4) The paper is technically sound and correct.
5) The paper is interesting and informative, which makes the reviewers feel it is useful to read.
6) The abstract and the summary, which summarize the paper well at the beginning and at the end, are concise.
7) The paper effectively uses diagrams and/or tables to present the ideas.
8) The paper closely follows the IEEE conference paper format and the given guidelines in the call for papers.

If the paper is a team work, the workload must be divided equally among the team members. It must be made clear which sections are written by (the responsibility of) which member. The reviewers may give different scores to different team members based on the sections and the paragraph which the members are responsible for.

2. Peer Evaluation: 10 points

Each student will act as a reviewer and will review three papers and submit three review reports. The quality of the review reports will be evaluated by the instructor. Up to 10 points will be awarded.

3. Improvement of the paper based on the review reports: 10 points.

The authors of each paper must improve the paper based on the comments in the review reports. The part of changes made must be shown in "Track changes" in MS Word**.** You can turn on the track changes in the Tool menu. Resubmit the paper after the revision. The instructor and the teaching assistants will check if the improved paper addresses the comments given by the reviewers. A camera-ready copy must be submitted and the papers will be published in an electronic form.

Previous workshop proceedings are available at the website:

http://www.public.asu.edu/~ychen10/teaching/cse445/index.html

Typical Components of Technical Papers/Reports

Title

Author(s)

Affiliation(s)

Abstract

Summary, important issues and results, assuming the readers have not read the full report.

Introduction

This section may cover background information, related work, the purposes of this writing this paper, outline of the paper, and so forth.

The main sections

They may contain several or all of the following components.

- **Overview**, including the architecture of the system;
- **Model Development**: explore a few models — model refinements, include graphic, equations, and so forth;
- **Procedure** (the steps are you going to use to complete this design, assumptions);
- **Design of experiment, simulation, implementation**;
- **Discussion of Results**: the numerical and graphic results and from models, upper and lower limits.

Summary/Conclusions

Summary of the work and the important results, assuming the readers have read the full report;

Acknowledgements

Who have helped the authors in preparing the research and on what issues?

References

List the all the references that you have based your work on, related to, referred to, and so on. Each reference you have listed must be cited in the paper. List the references in IEEE proceedings reference format.

Appendices (if any)

For example, Excel spreadsheet, diagrams, and extra explanations.

Other issues: Include page numbers, cite the reference where the content is based on, related to, and referred to. Follow the required format.

Review Form

Workshop on Service-Oriented Computing (WSOC)

Paper ID:

Paper Title:

1. Numerical Evaluation

Scale: (0-2) Strongly disagree, (3-4) Weakly disagree, (5-6) Marginal, (7-8) Weakly agree, (9-10) Strongly agree

Evaluation questions:

1) The paper is relevant to one of the focus areas given in the call for papers (0-10).
2) The paper has well defined questions to address and the materials are coherent and consistent (0-10).
3) The paper clearly presents the ideas and is easy to read (0-10).
4) The paper is technically sound and correct (0-10).
5) The paper is interesting and informative, which makes the reviewers feel useful it is to read (0-10).
6) The abstract and the summary, which summarize the paper well at the beginning and at the end, are concise (0-10).
7) The paper effectively uses diagrams and/or tables to present the ideas (0-10).
8) The paper closely follows the IEEE conference paper format and the given guidelines in the call for papers (0-10).

2. Detailed Comments

Please supply detailed comments to support each of your scores. You may also indicate any errors you have found. The length of the comments must be between 15 and 30 lines.