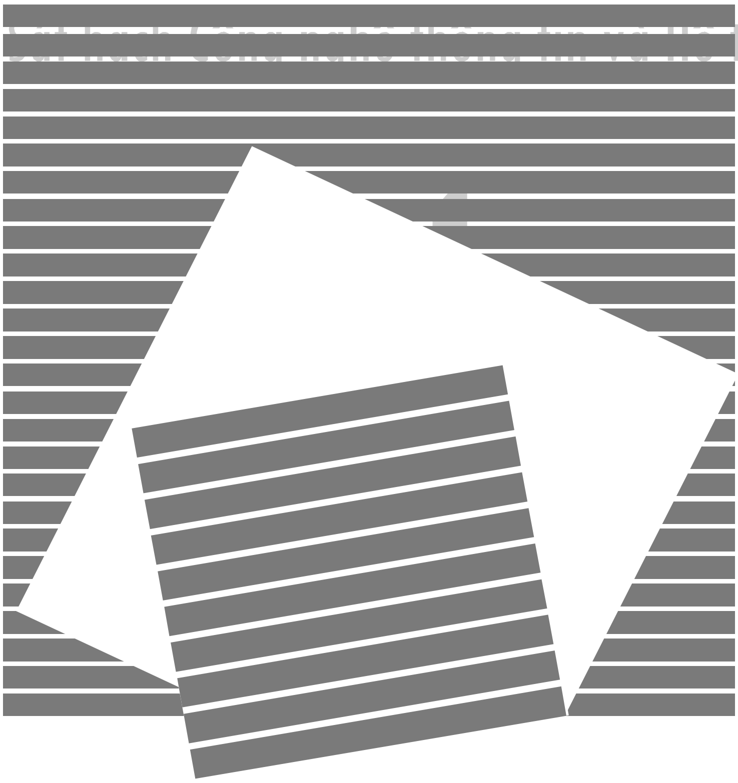
METI
Ministry of Economy,
Trade and Industry

Textbook for Software Design & Development Engineers

NO. 4 OBJECT-ORIENTED DEVELOPMENT



Second Edition

REVISED AND UPDATED BY

Japan Information Processing Development Corporation
Japan Information-Technology Engineers Examination Center

Textbook for Software Design & Development Engineers

No. 4 OBJECT-ORIENTED DEVELOPMENT

First edition first printed October 1, 2001
Second edition first printed August 1, 2002

Japan Information Processing Development Corporation
Japan Information-Technology Engineers Examination Center
TIME 24 Building 19th Floor, 2-45 Aomi, Koto-ku, Tokyo 135-8073 JAPAN

Table of Contents

1.Basic Concepts of Object-Orientation

2. UML

6. Implementation

7. Major Object-Oriented Techniques

1 Basic Concepts of Object-Orientation

Chapter Objectives

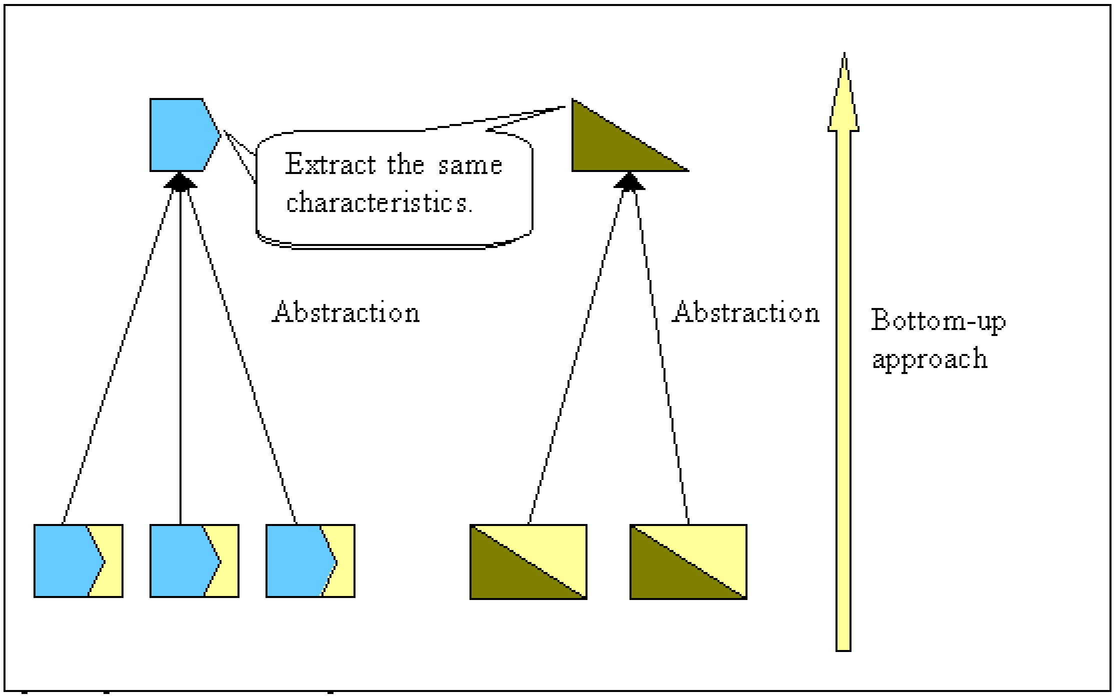
This chapter explains the basic concepts of the object-orientation, and also explains classes and objects, methods and messages, encapsulation and interfaces, association and multiplicity, inheritance, and polymorphism and dynamic binding.

1.1 Classes, Objects, and Attributes

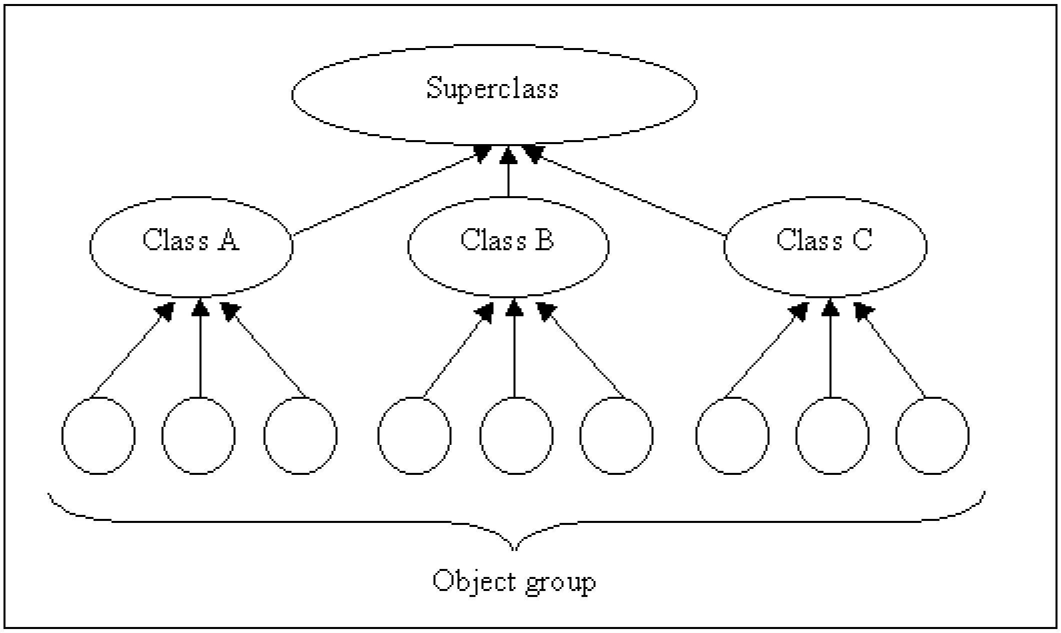
Classes, objects, and attributes are the basic concepts of the object-orientation. Attributes are information held by each object, an object is anything that models "things" in the real world, and a class is a description of the common characteristics of several objects. This section explains object abstraction, object templates, attributes, and methods.

1.1.1 Abstraction

Abstraction refers to the concept of identifying essential properties while simultaneously eliminating nonessential properties. Using this concept, things in the real world are abstracted; therefore, it is possible to build a stable model, to represent naturally an abstraction of reality and to express the reality, and to clarify the focus of analysis. The figure below shows an example of abstraction.

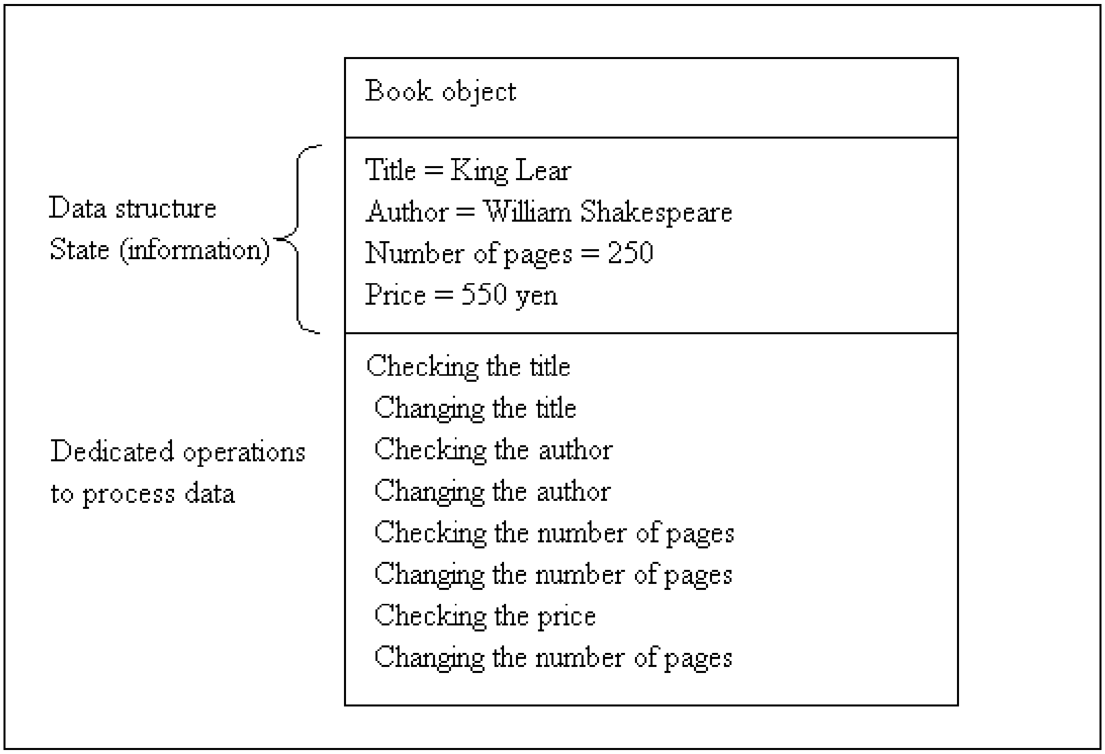


This figure shows the extraction of common characteristics from several objects. A description of the extracted common characteristics is called a class. A class at a higher level of abstraction is called a superclass. The relationships between classes and a superclass are shown in the following:



[Relationships between classes]

The model and design in the object-oriented paradigm are explained as follows.
This is a new thinking methodology for solving problems that is organized using concepts in the real world. The fundamental element is an object, which has both a data structure and behavior in a single entity. The object, which is an entity that can save a state (information), and which offers a number of operations (behavior) to either examine or affect this state. The following figure shows an example of an object.



Book object

Data structure
State (information)

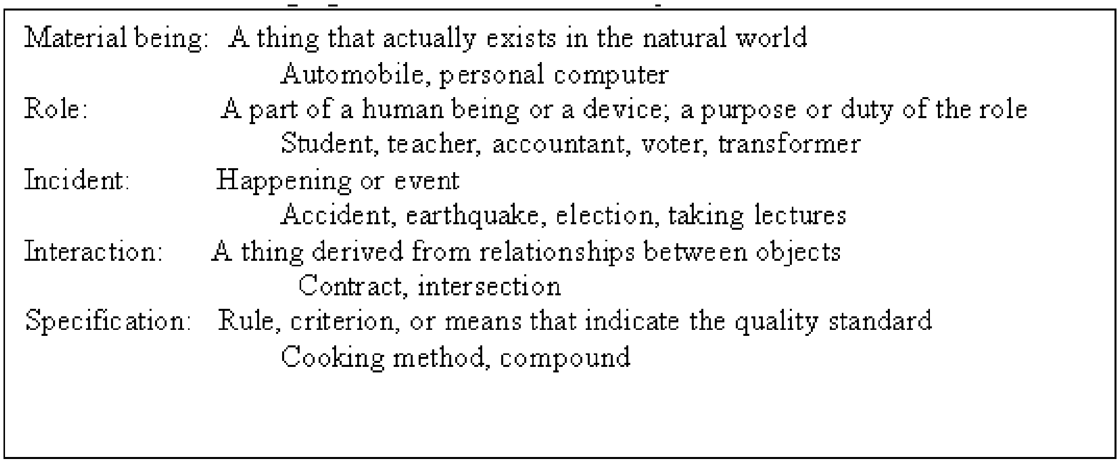
Title = King Lear
Author = William Shakespeare
Number of pages = 250
Price = 550 yen

Dedicated operations
to process data

Checking the title
Changing the title
Checking the author
Changing the author
Checking the number of pages
Changing the number of pages
Checking the price
Changing the number of pages

[Example of object]

Using object-orientation as a base, we model the system as a number of objects that interact. The following figure shows the kinds of objects.



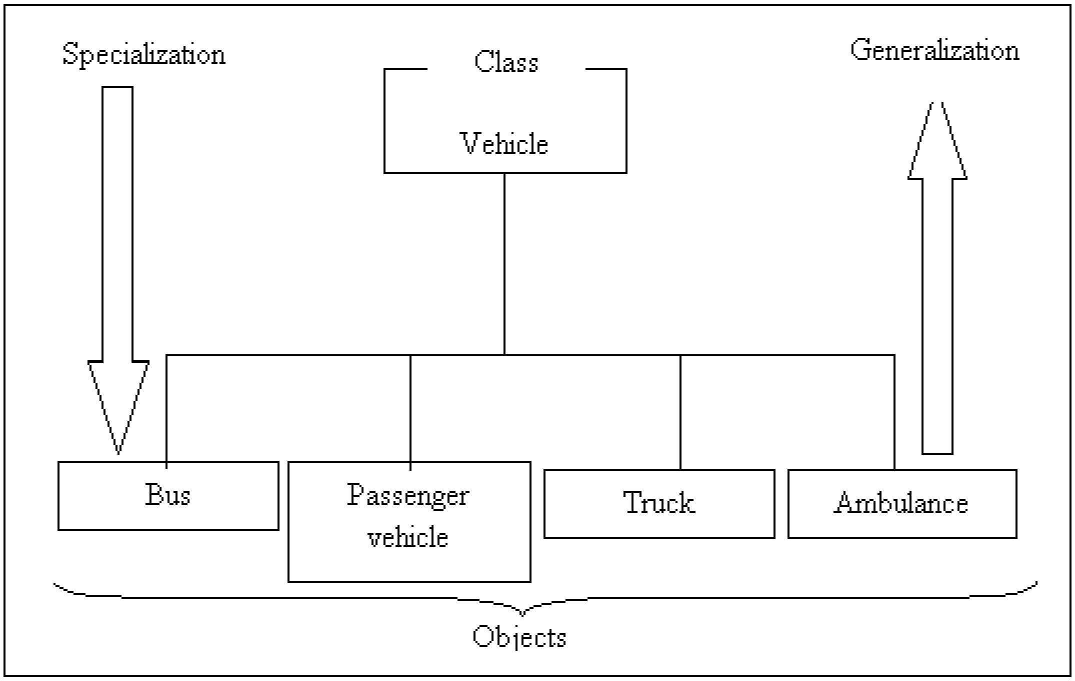
Material being: A thing that actually exists in the natural world
Automobile, personal computer
Role: A part of a human being or a device; a purpose or duty of the role
Student, teacher, accountant, voter, transformer
Incident: Happening or event
Accident, earthquake, election, taking lectures
Interaction: A thing derived from relationships between objects
Contract, intersection
Specification: Rule, criterion, or means that indicate the quality standard
Cooking method, compound

[Kinds of objects]

As shown in this figure, it is possible to avoid the duplicated development for programs having similar functions by considering the common characteristics in the initial step of the system development process.

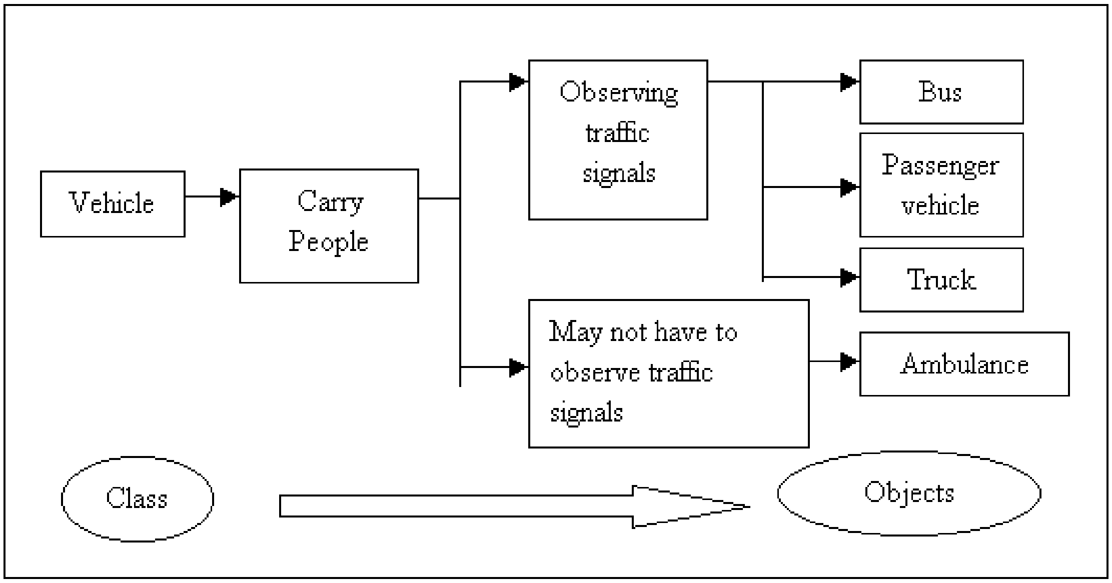
1.1.2 Object templates

The object template is a class expressed by abstracting objects having the same characteristics. The following figure shows an example of a class that clarifies the relationships between classes and objects.



[Relationships between a class and objects 1]

As shown in the figure, a class called “vehicle” is defined by abstracting characteristics common to each object. Conversely, the things related to the common characteristics called “vehicle” can be extended to specific objects such as “passenger vehicle” and “truck.” The following figure shows an example:

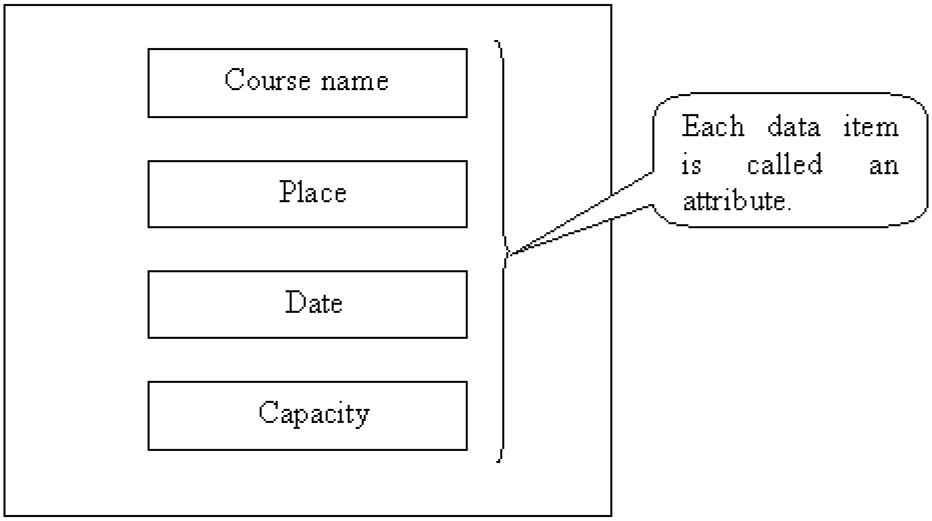


[Relationships between a class and objects 2]

The class is therefore assumed to be an object template.

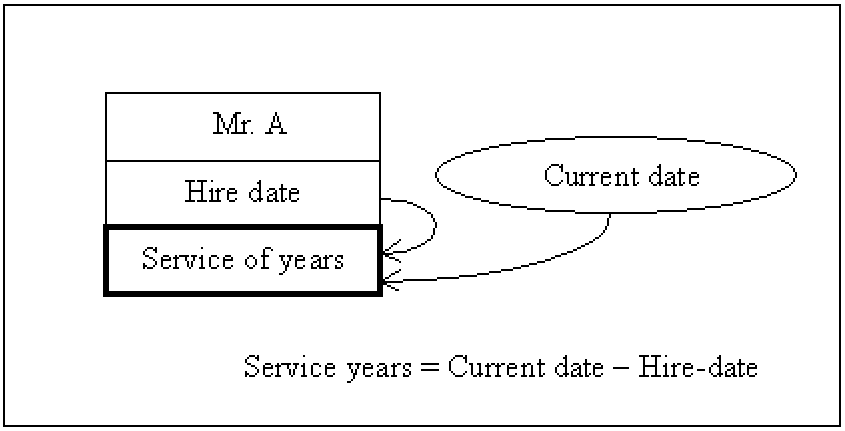
1.1.3 Attributes

An attribute means an internal state of an object, which is changed by an external action. This is explained in the following, using the object "Course" as an example. The "Course" object contains the course name, place, date, seating capacity, and so on. The configuration of the "course" object is shown in the following:



[Course object]

There is also a derived attribute related to the attribute. The derived attribute is an attribute derived from one or more attributes. The figure in the following shows an example of a derived attribute.



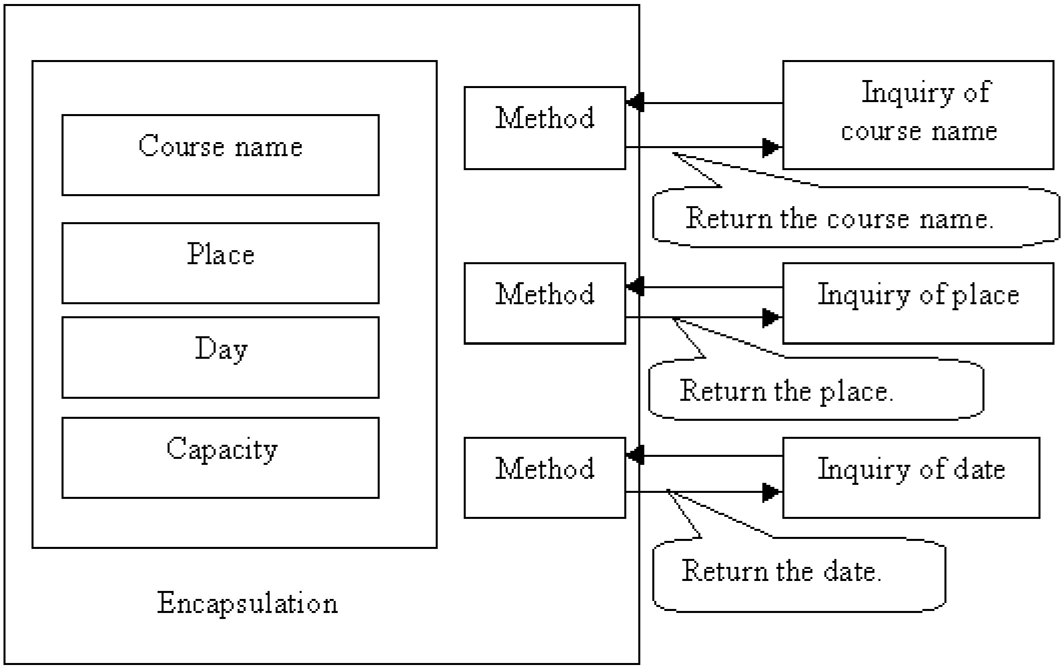
[Obtaining derived attribute]

As shown in this figure, "service years" obtained from the "hire date" and "current date" attributes is called a derived attribute.

When these attributes are changed from outside, consistency among the attribute may not be assured. Therefore, the attributes are hidden in the object so that they cannot be referred to directly from the outside. The object has a mechanism that reacts to an external signal. In other words, if a signal is given to inquire information that is used outside the object, an object returns the required information to the outside as a response. This mechanism is called a method.

1.1.4 Methods

A method, which is a mechanism of the object, is a function that responds to an external inquiry and takes an appropriate action for a specific request. The following figure shows an example of a method.

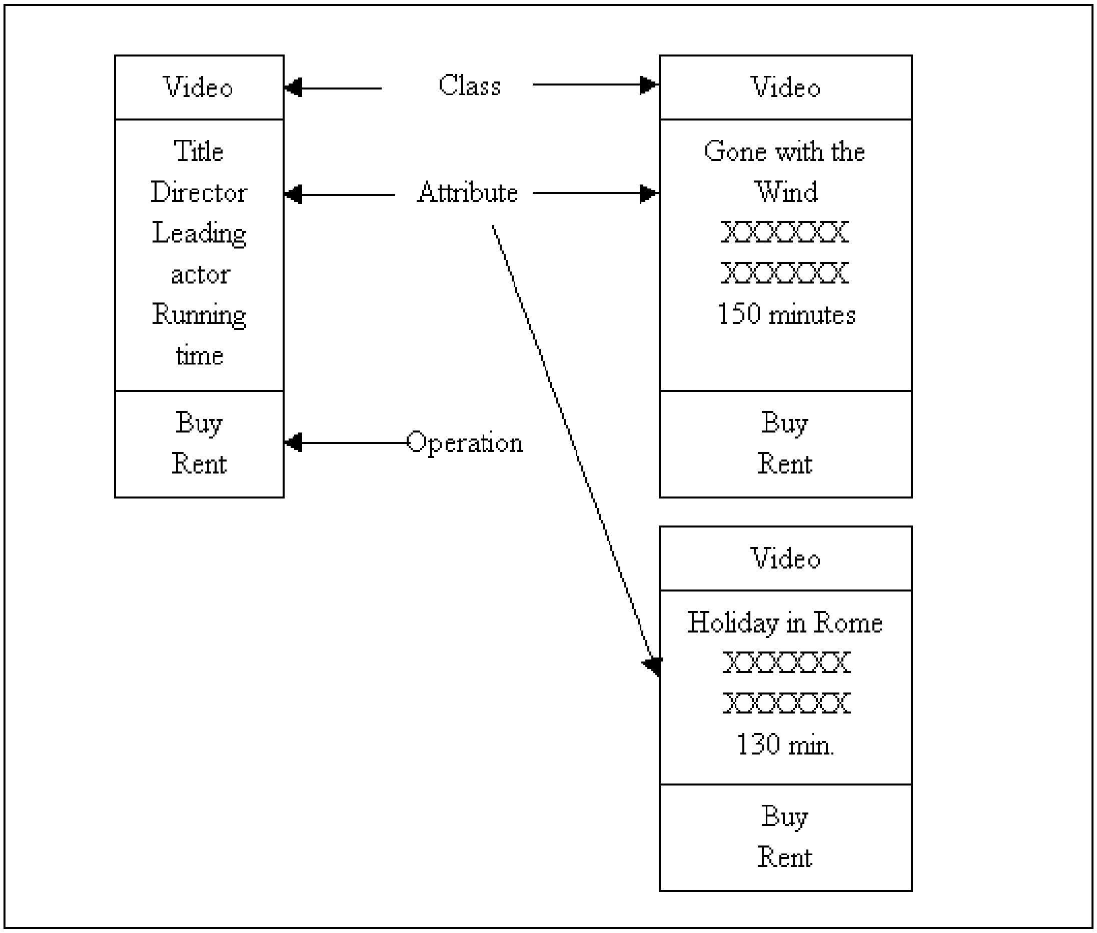


[Course object]

As shown in this figure, a method is a mechanism that returns the appropriate information to the outside in response to a specific external inquiry.

[Reference]

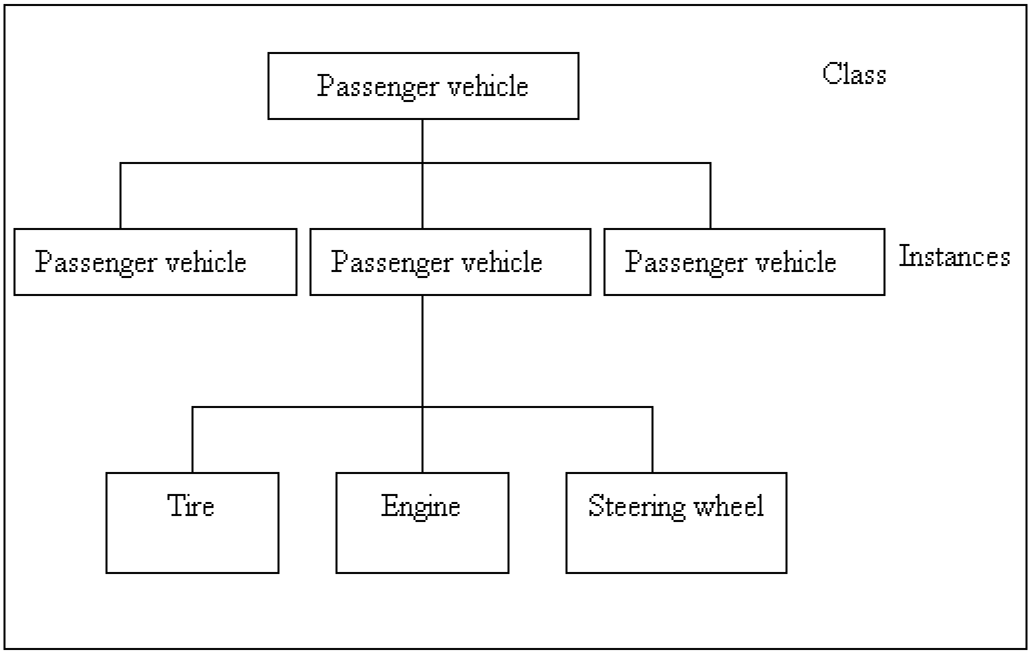
(1) Class diagrams
The class diagram is created on the basis of an object model. Using this class diagram, one can express a class structure, for example, a hierarchical relation or relationships between classes. The following figure shows an example of a class diagram.



[Example of a class diagram]

(2) Object diagrams
The object diagram shows objects in a system together with the relationships between objects. An example of an object diagram is shown in the following.

What is an object diagram?
Describes an instance.
The instance is an object created by assigning actual data to a class.
Used when there are particularly remarkable relationships.



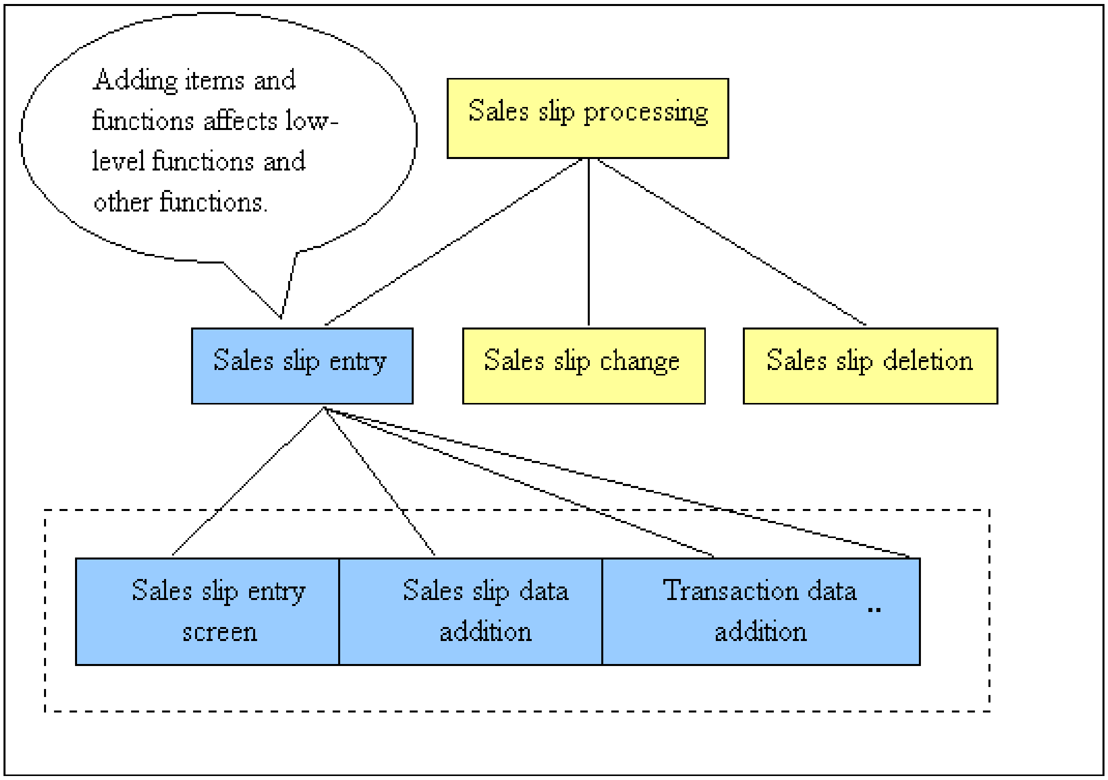
[Example of an object diagram]

1.2 Encapsulation and Interfaces

In the past, the mainstream technique in design was structured design techniques. However, various problems have been pointed out in this technique, and as a result object-orientation is now increasingly used in design and development. This section describes the problems of structured design techniques, and explains how these can be solved using the object-orientation. It also explains encapsulation and operation on attributes in the object-orientation.

1.2.1 From partitioning to integration of data and procedures

System design and development have been performed mainly using structured design technique, but various problems arise due to the characteristics of this technique. The causes are explained using examples as follows:



[Faults in structured design technique 1]

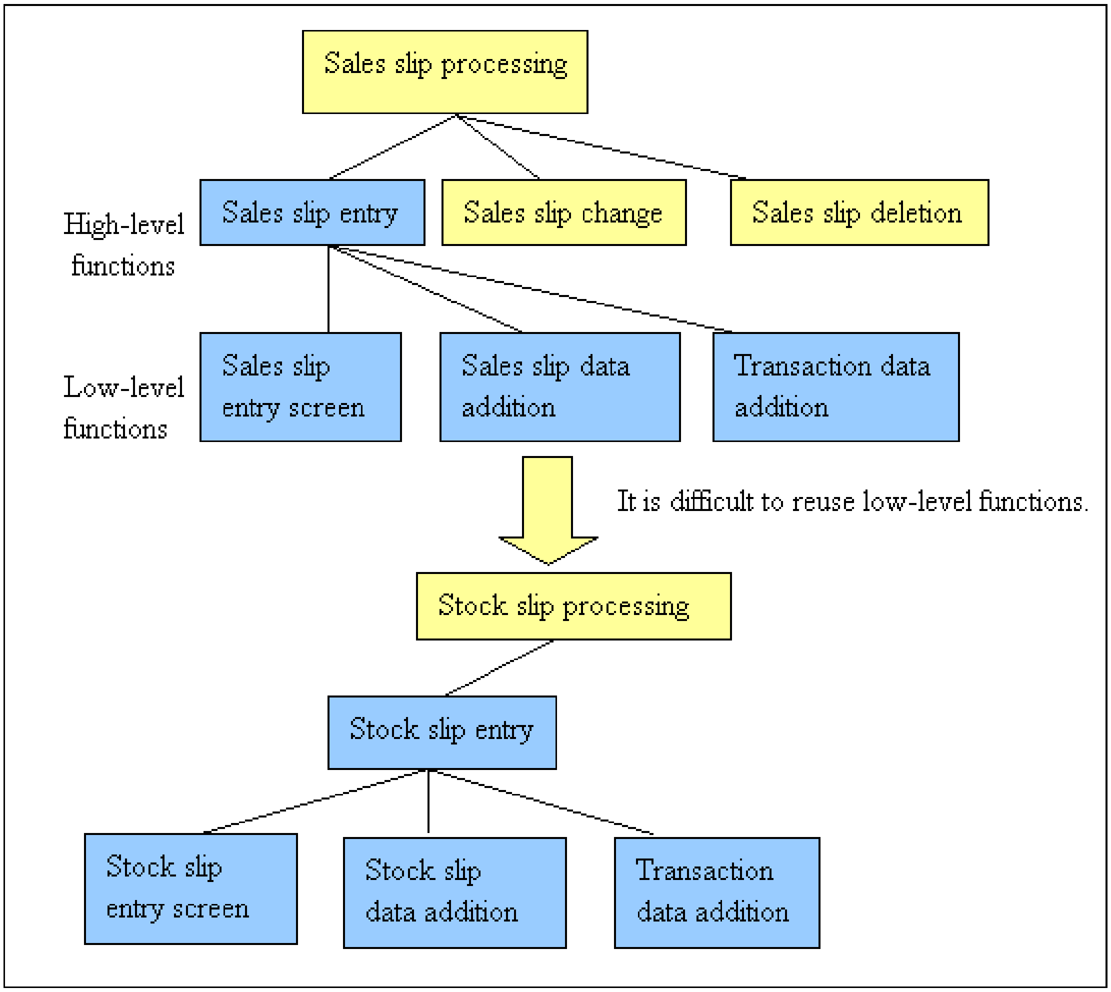
In this figure, there are the following problems:

(1) The changes to specifications of a function, which is a focused unit in the structured design techniques can easily happen.

(2) In the structured design technique, low-level functions are created to implement a high-level function. If the high-level function is altered, it may affect the low-level and other functions.

(3) In the structured design technique, low-level functions are created to satisfy a high-level function. Therefore, it is difficult to reuse existing functions.

For example, in the following figure, when the low-level functions that has "sales slip entry" as a high-level function are reused for another high-level function named “stock sales slip entry,” the low-level functions must be modified to implement the high-level function, “stock slip entry.”

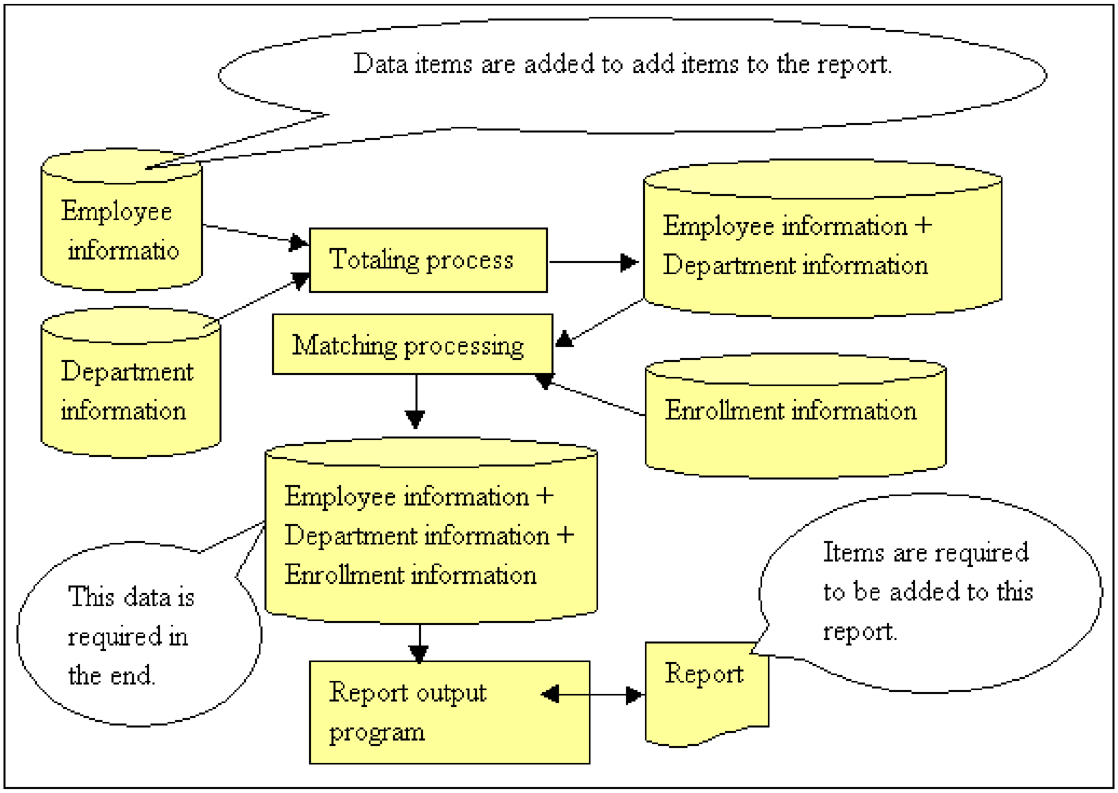


[Faults in structured design technique 2]

(4) Since the structured design technique is function-oriented, what should be done to implement a specific function, tend to be considered in the first place. As a result, data for which functions can be easily implemented are considered, and it becomes difficult to maintain data consistency.

(5) Because of the existence of the same information at different locations, changing the data structure may affect multiple programs in a system.

For example, as shown in the following figure, suppose that data items to be outputted to a report are added. In this case, the data items are added to employee information, but the totaled data may not match enrollment information in matching processing after totaling processing. The same data is placed at different locations; therefore, if the data structure is modified, it affects programs related to the data.

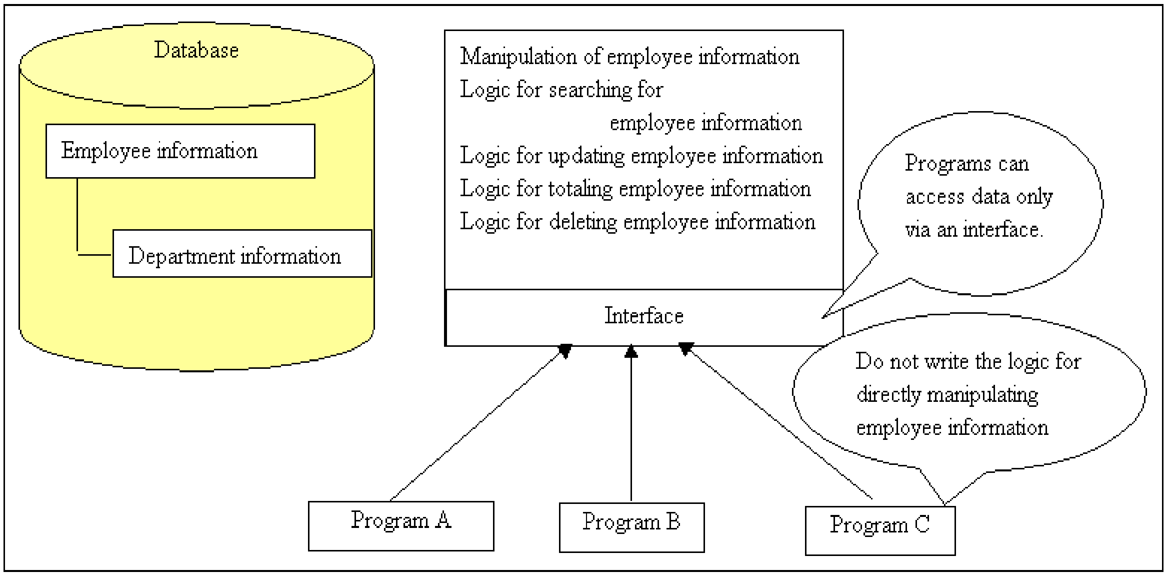

Data items are added to add items to the report.
Employee informatio
Totaling process
Employee information + Department information
Department information
Matching processing
Enrollment information
Employee information + Department information + Enrollment information
This data is required in the end.
Items are required to be added to this report.
Report
Report output program

[Faults in structured design technique 3]

These are the problems recognized in the structured design technique. The object-oriented paradigm models things in the real world as ones in the computer world, thus reducing inconsistency between the target world and the software structure. Each object has data, functions, and associations as information like things in the real world, so a model can be built using natural concepts. Each object is separated and independent, and has methods internally, which provides the following merits:

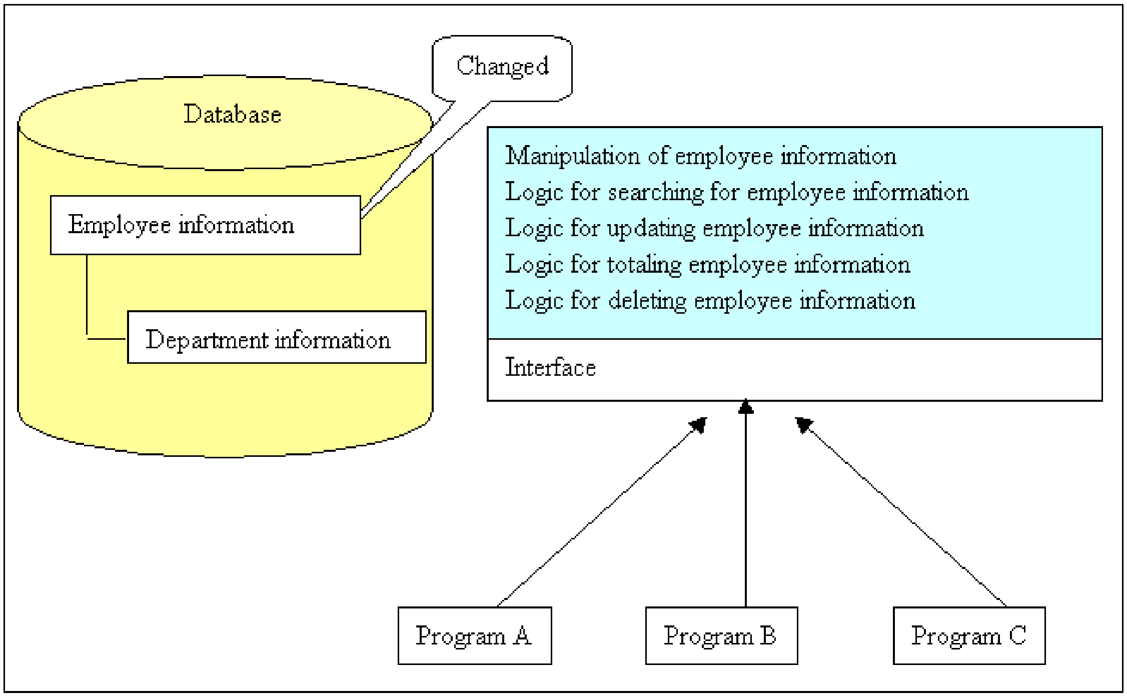
(1) Since the visualization of programs is possible, the development efficiency increases. The visualization of programs means that the characteristics and methods which objects have are easy to understand.
(2) Since a program is separated into objects as units which are independent of each other, it can be modified easily.
(3) Since each object has attributes and methods, which are encapsulated, a program can be handled as a component and easily be ported to another application. Encapsulation means to create a structure that puts together attributes and methods.

Encapsulation is explained next. Encapsulation means putting together the attributes of an object and methods which are operations on those attributes. In the encapsulation, the methods are made public to the outside while the attributes are hidden, which prevents external programs from directly handling the attributes. Two encapsulation examples are shown in the following:


Database
Employee information
Department information
Manipulation of employee information
Logic for searching for employee information
Logic for updating employee information
Logic for totaling employee information
Logic for deleting employee information
Interface
Programs can access data only via an interface.
Do not write the logic for directly manipulating employee information
Program A
Program B
Program C

[Encapsulation example 1]

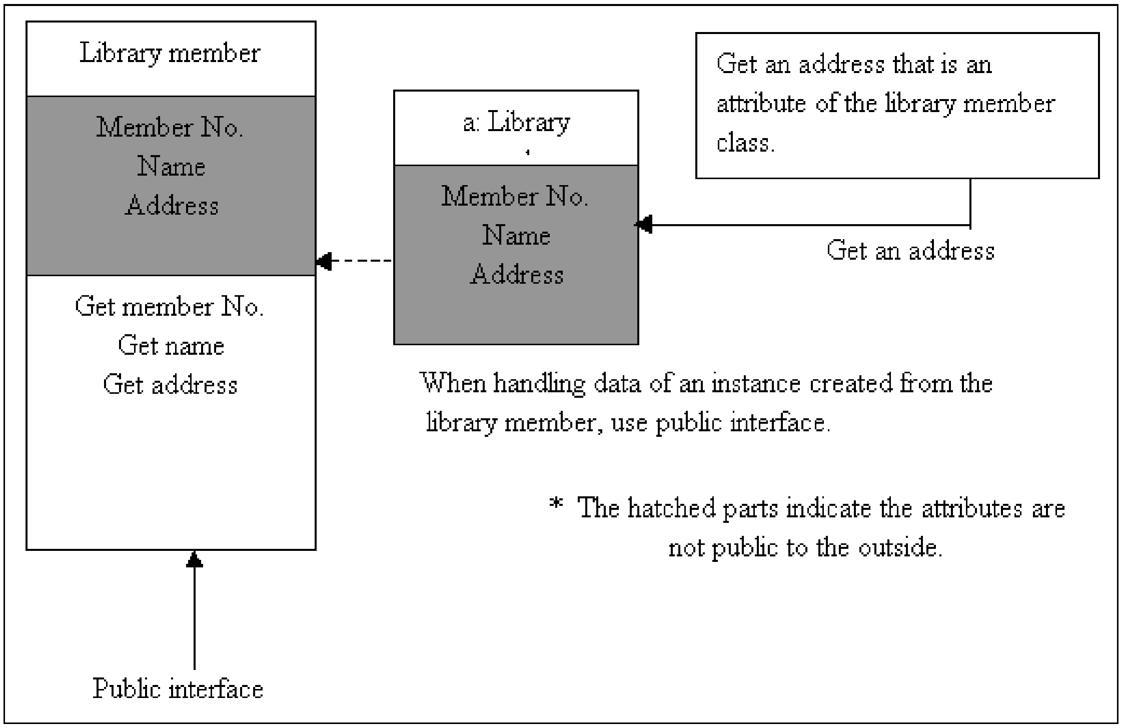
As shown in this figure, data is not public to the outside. Therefore, a method that is public to the outside must be used to process data. It is also important that no logic for manipulating the hidden data is described in external programs.


Changed
Database
Employee information
Department information
Manipulation of employee information
Logic for searching for employee information
Logic for updating employee information
Logic for totaling employee information
Logic for deleting employee information
Interface
Program A
Program B
Program C

[Encapsulation example 2]

As shown in this figure, even if employee information is changed, it does not affect programs A, B and C.

The following figure shows another example of encapsulation.

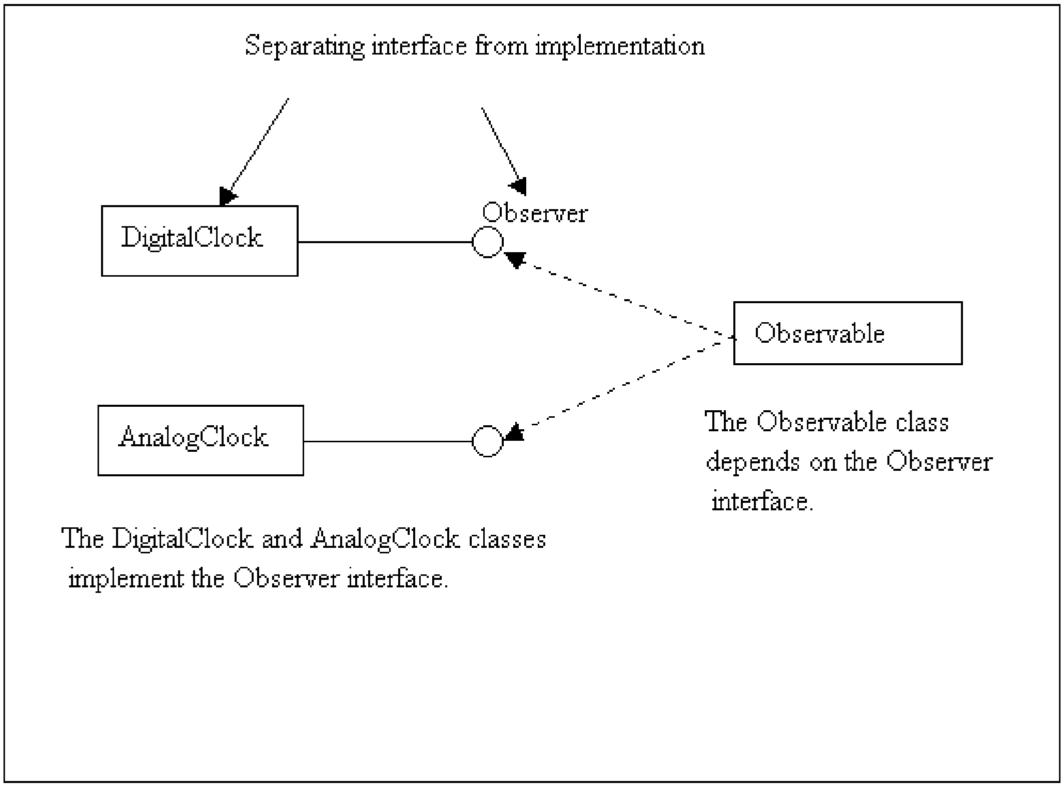

Library member
Member No.
Name
Address
Get member No.
Get name
Get address
a: Library
Member No.
Name
Address
Get an address that is an attribute of the library member class.
Get an address
When handling data of an instance created from the library member, use public interface.
* The hatched parts indicate the attributes are not public to the outside.
Public interface

[Encapsulation example 3]

As shown in this figure, encapsulation make public only operations without making public the data structure. In other words, user programs can process data by activating the public operations. Even if the data structure is changed, user programs are not affected.

The interface shown in the figure is explained here. The interface of an object is completely separated from the implementation. The interface means "a set of ways of using operations that are made public by the object for specific purposes".

The figure below shows an example of interface.


Separating interface from implementation
DigitalClock
Observer
Observable
AnalogClock
The Observable class depends on the Observer interface.
The DigitalClock and AnalogClock classes implement the Observer interface.

[Example of interface]

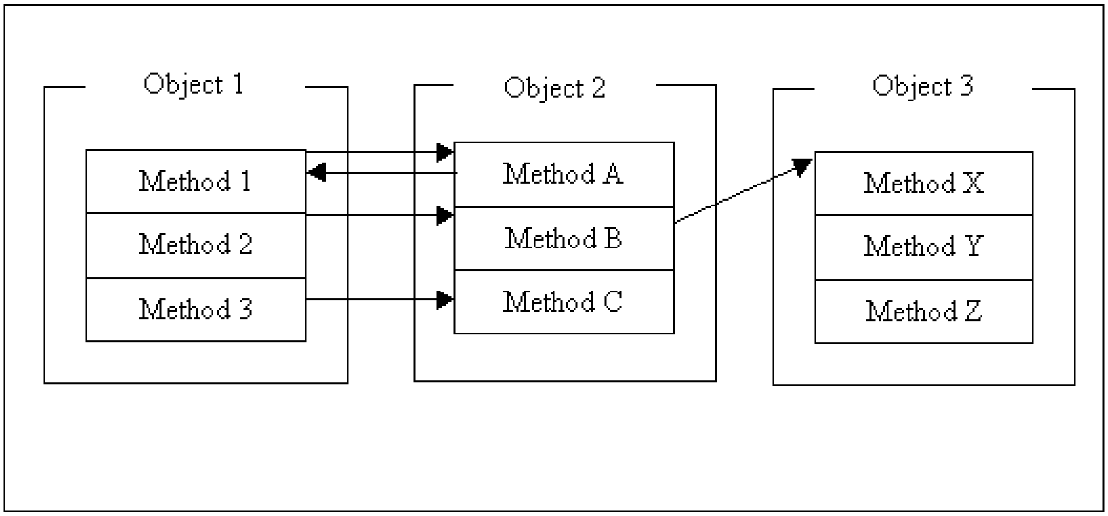
In this figure, only the Observer interface is public to the Observable class, and DigitalClock and AnalogClock classes, which implement the Observer interface, are hidden. The Observable class always requests operations to the Observer interface, so the Observable class need not recognize whether the implementation resides in DigitalClock or in AnalogClock class.

1.2.2 Method invocation

Method invocation is a means by which objects perform operations in collaboration while passing messages with each other. There are the following three patterns for the behavior of the message receiver:

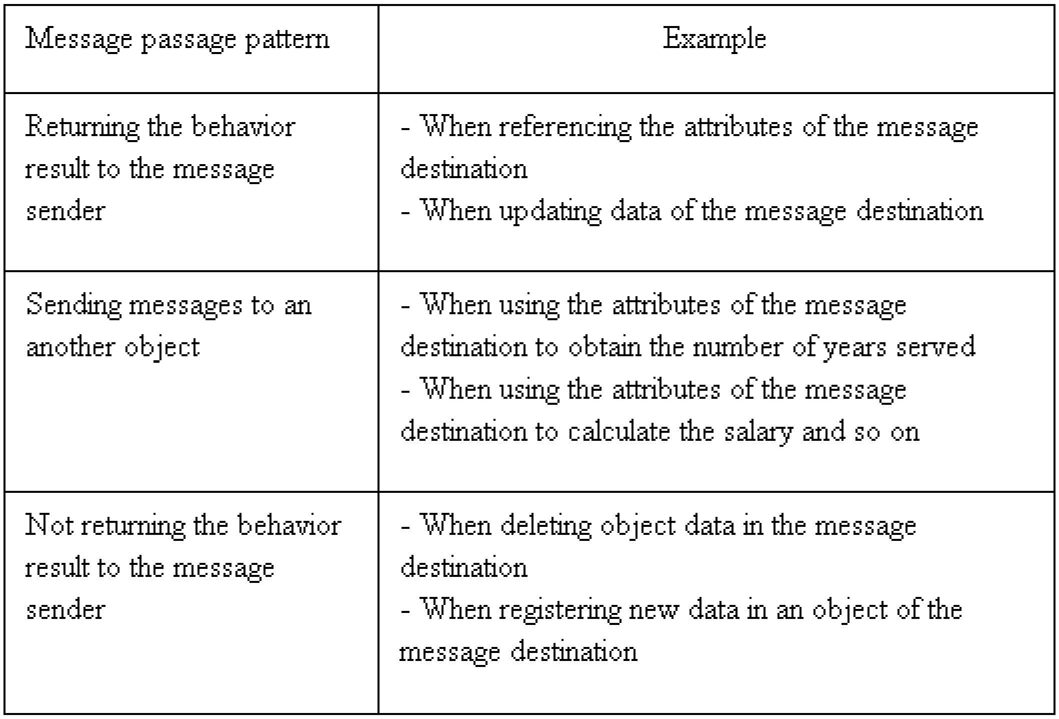
(1) Returning the result of the behavior for message to the message sender
(2) Sending a message to another object, because the receiver object by itself cannot complete the required behavior for the message
(3) Not returning the result of the behavior for the message to the message sender

The following figure shows these patterns.



[Method behavior patterns]

The following table gives examples of the method behavior.



Message passage pattern	Example
Returning the behavior result to the message sender	- When referencing the attributes of the message destination - When updating data of the message destination
Sending messages to an another object	- When using the attributes of the message destination to obtain the number of years served - When using the attributes of the message destination to calculate the salary and so on
Not returning the behavior result to the message sender	- When deleting object data in the message destination - When registering new data in an object of the message destination

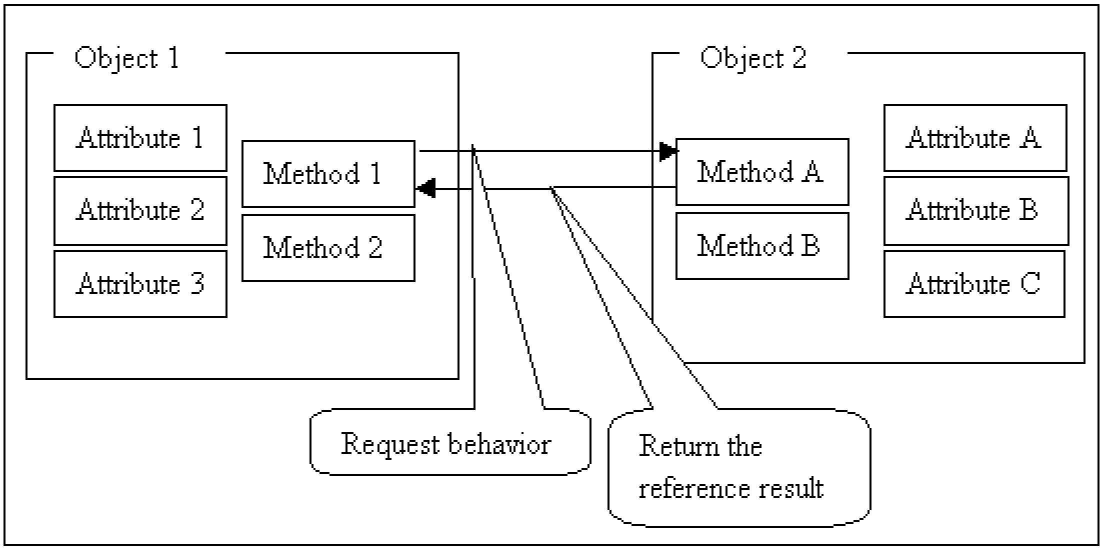
As described in the table, method invocation means calling a method in a target object, when referencing attributes of the object.

1.3 Association and Multiplicity

In the object-oriented paradigm, objects may be associated with others and use each other. This section explains the reference and association between objects and multiplicity.

1.3.1 Reference and association between objects

A reference is established between objects when one object passes a message to the other one to invoke a behavior. For example, when referencing or changing an attribute of another object, it is done by using its method. Such a mechanism that requests another object is called message passing. The following figure shows an example of message passing.

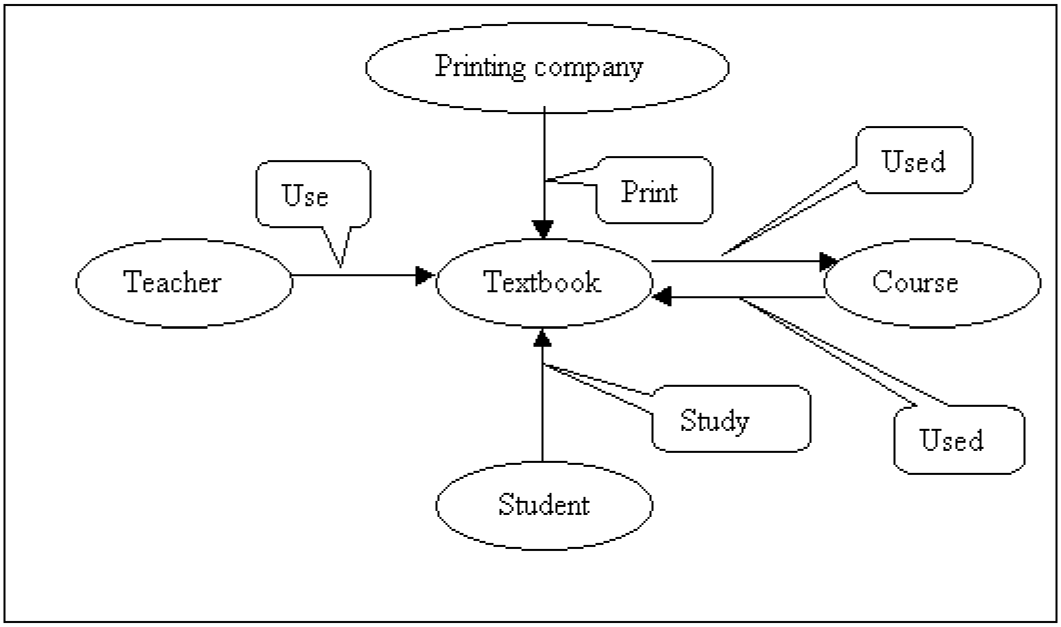


[Outline of message passing]

In this figure, object 2 contains method A for referencing the attributes of object 2 itself. When object 1 references the attributes of object 2, therefore, it must request the operation called attribute reference for method A defined in object 2. As a result, object 1 can reference the attributes of object 2.

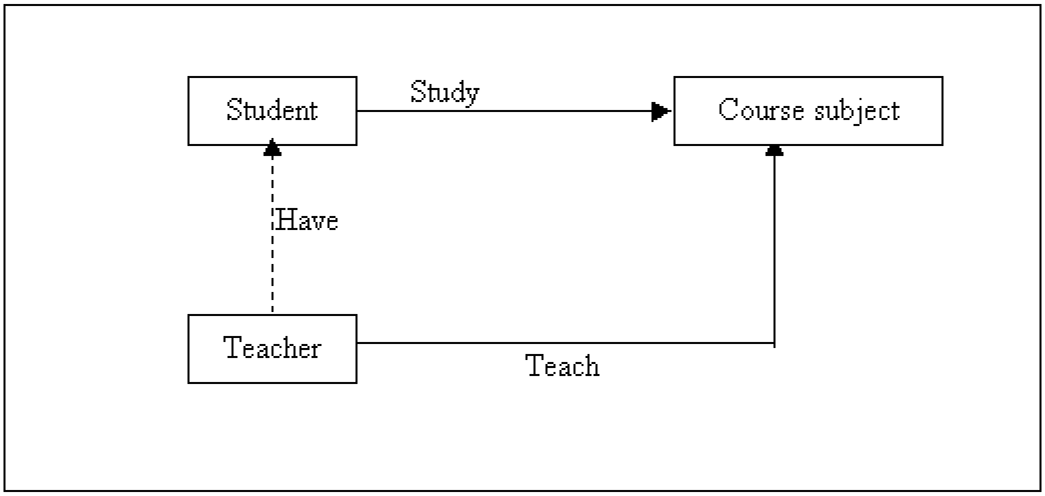
The association between objects means a temporary relationship that can be expressed with "object 1 uses object 2."

The following figure shows an example of an association between objects.



[Example of association]

In addition, there is a derived association as a form of the association. A derived association is one which can be obtained from another association. The following figure shows an example of a derived association.

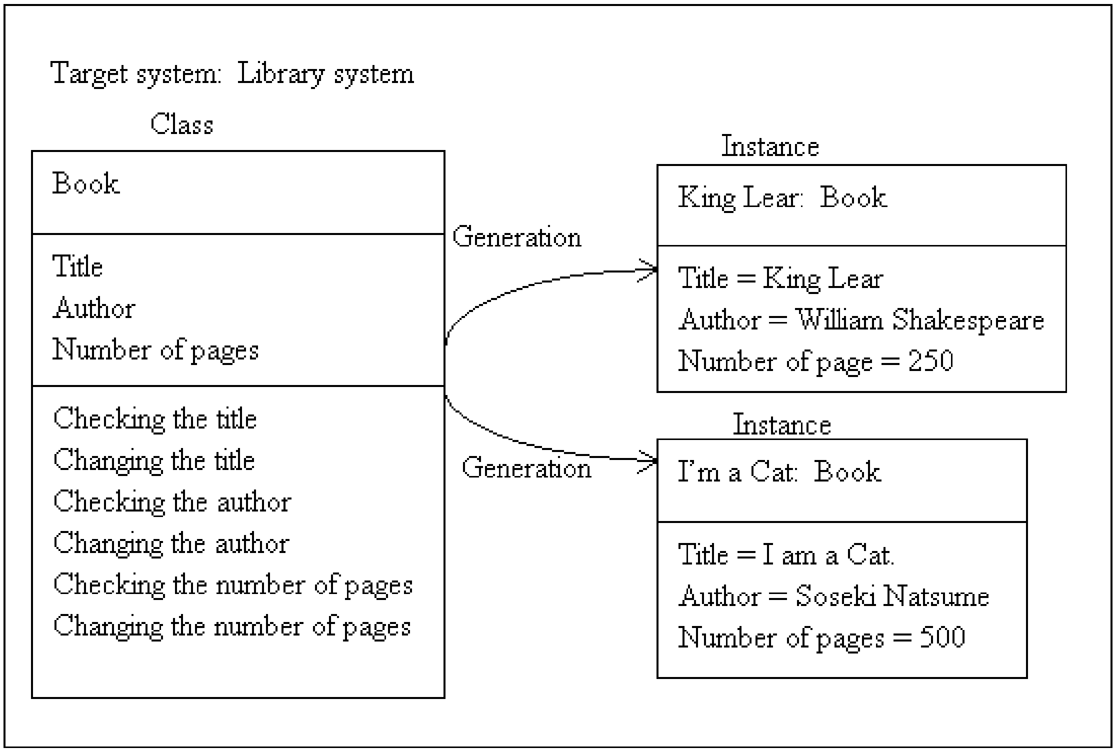


[Example of derived association]

In this figure, a new association between the student and teacher can be derived from existing associations between the student and course subject and between the teacher and course subject. Like this, an association that can be obtained from other associations is called a derived association.

1.3.2 Uses relationship

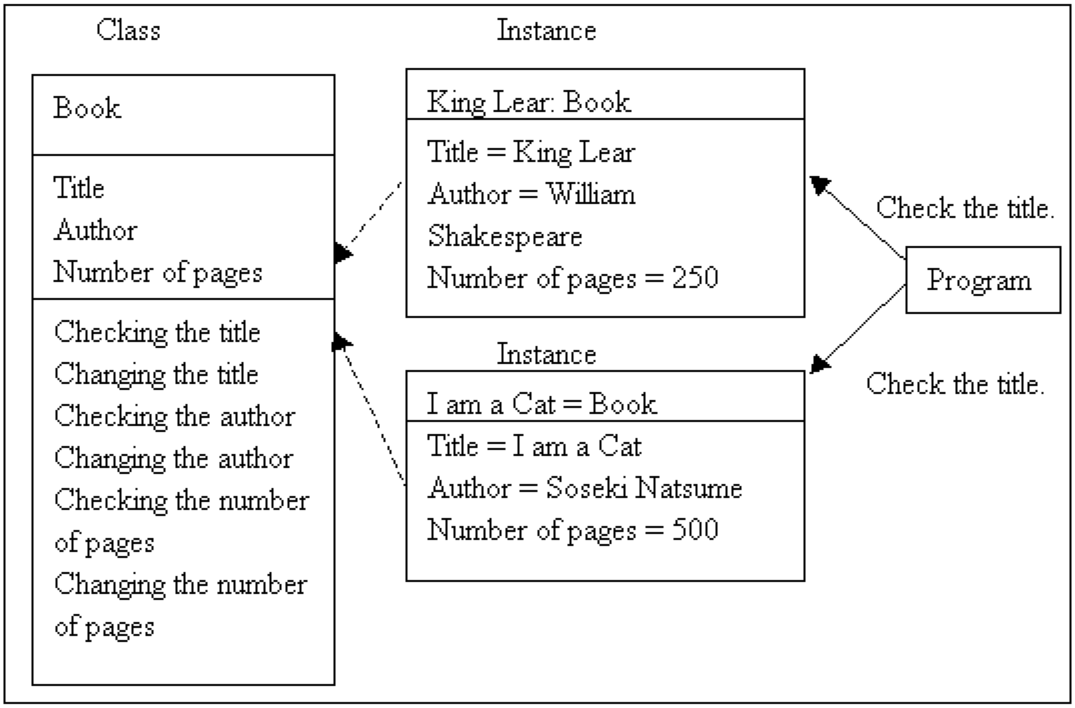
The use relationship indicates how many instances of one class can relate to each instance of another class. An instance is an object generated by giving actual data values to a class. The following figure shows an example of an instance.



[Example of instance]

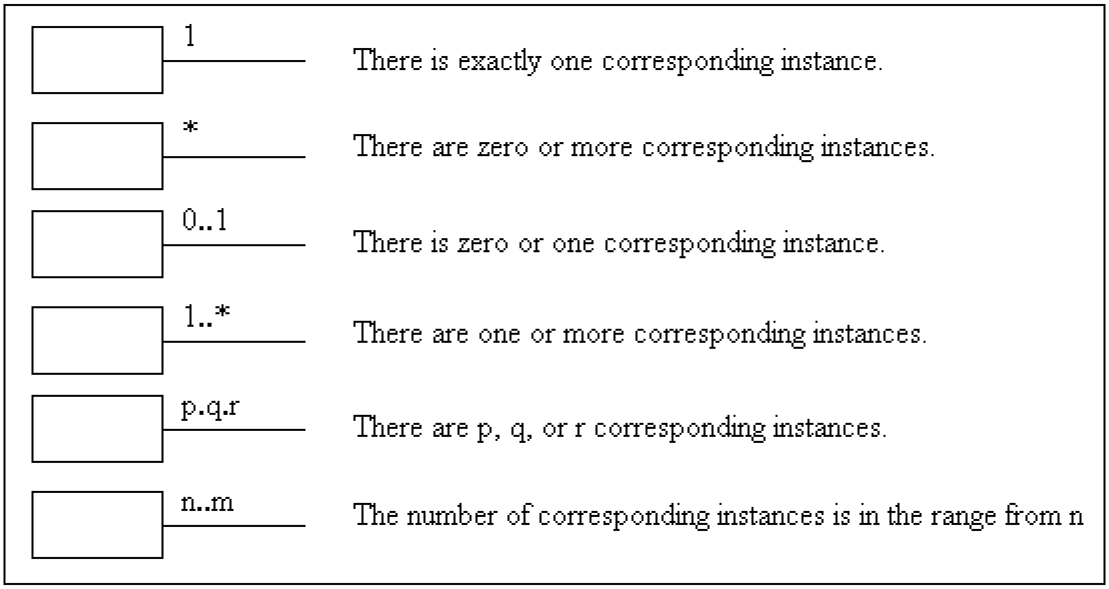
As shown in this figure, as instances are generated from the same class, they have information of the same structure. Attribute names and operations in a class are shared by multiple instances. When a program wants to know title information, it applies an operation to the instance created from the book class without directly applying an operation to the book class. The instance to which an operation is applied checks whether there is a target operation in the book class. If it is found, the instance executes the operation.

The following figure shows an example of this operation.



[Example of operation request for instance]

The following figures show the notation for multiplicity and an example.



There is exactly one corresponding instance.

There are zero or more corresponding instances.

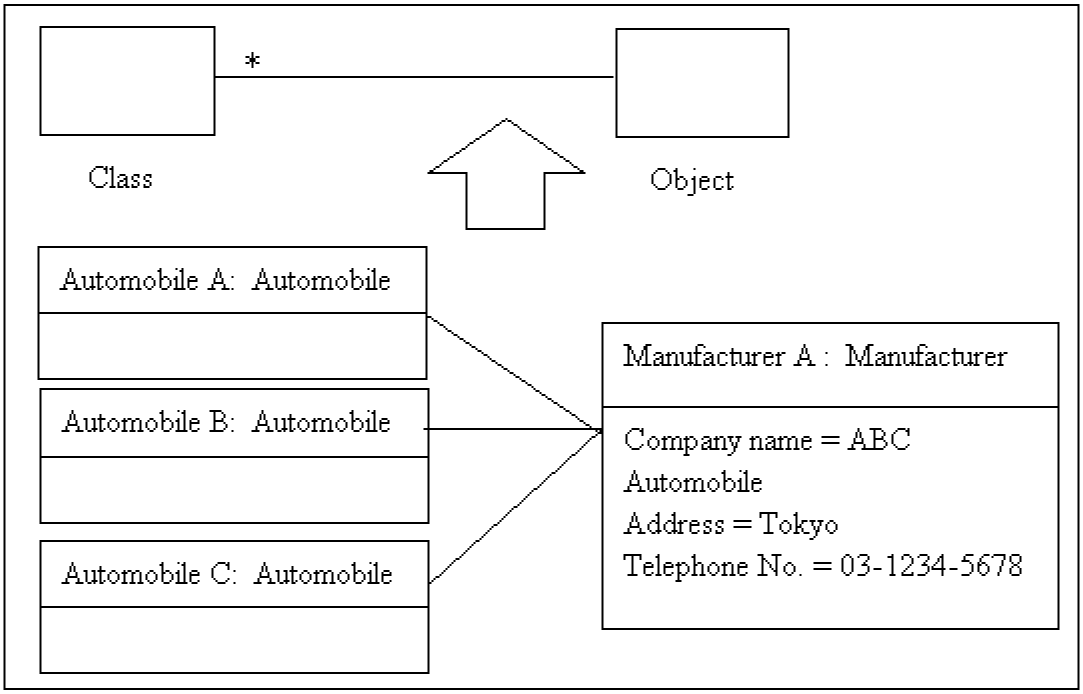
There is zero or one corresponding instance.

There are one or more corresponding instances.

There are p, q, or r corresponding instances.

The number of corresponding instances is in the range from n

[Notation of multiplicity]

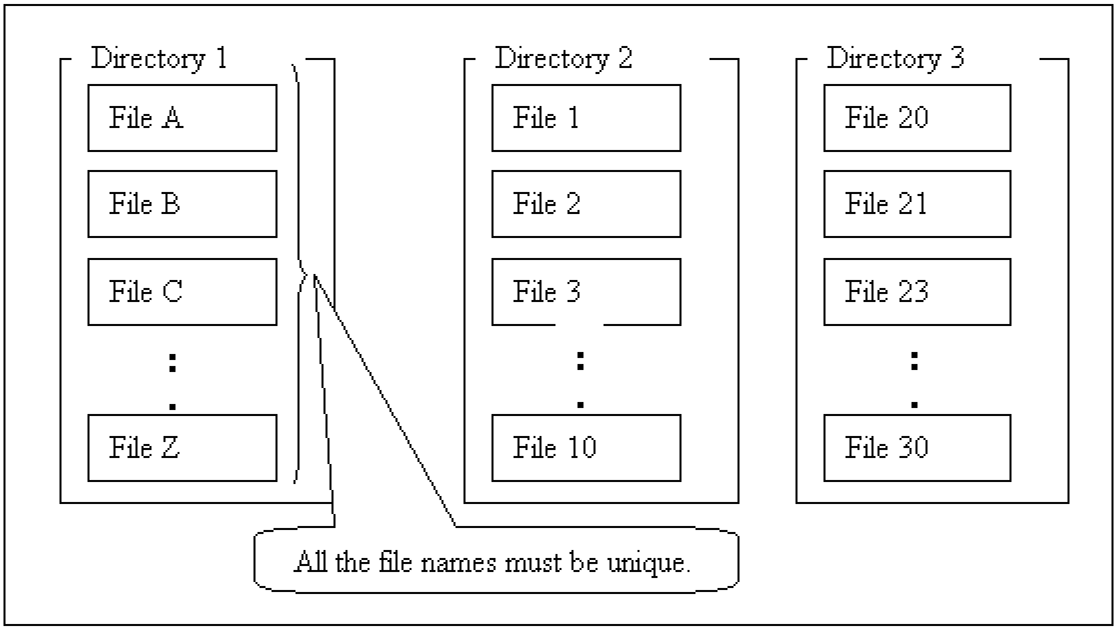

*
Class
Object
Automobile A: Automobile
Automobile B: Automobile
Automobile C: Automobile
Manufacturer A : Manufacturer
Company name = ABC
Automobile
Address = Tokyo
Telephone No. = 03-1234-5678

[Example of multiplicity notation]

In this figure, manufacturer A is related to automobiles A, B and C. This state is called multiplicity. In other words, multiplicity indicates how many objects of one class can relate to one object of another class.

There is an attribute called a qualifier, which is related with multiplicity. The qualifier is an attribute used to distinguish a specific object from multiple objects in the associated classes. The qualifier functions to reduce the multiplicity. For example, a qualifier is used to select a file in a specific directory.

The following figure shows an example of a qualifier.



[Example of delimiter]

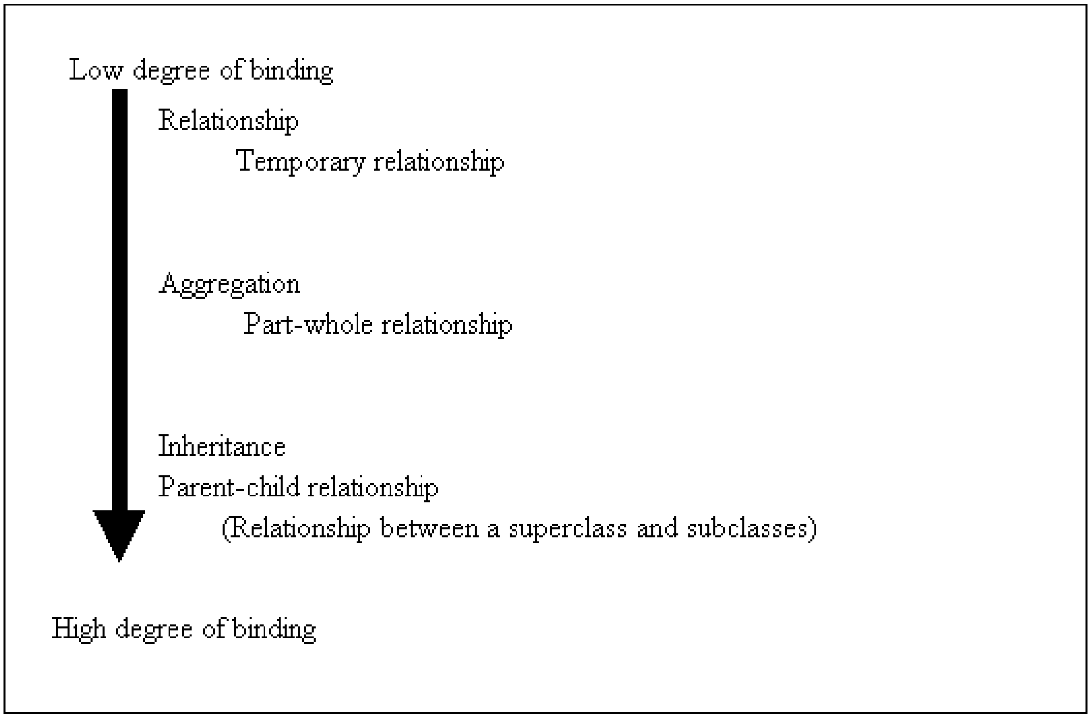
In this figure, a file that can be referenced is determined by specifying a directory. Only one file is obtained by specifying a directory and then defining the file name. In this example, the multiplicity between directory and file is reduced from “1 : multiple” to “1 : 1” by specifying a file name as a qualifier.

1.4 Inheritance and Aggregation

The object-orientation includes a relationship between classes and an relationship between objects. The relationship between classes is called is-a relationship, and one between objects is called part-of relationship.

1.4.1 Conceptual inclusion relationship between classes (is-a)

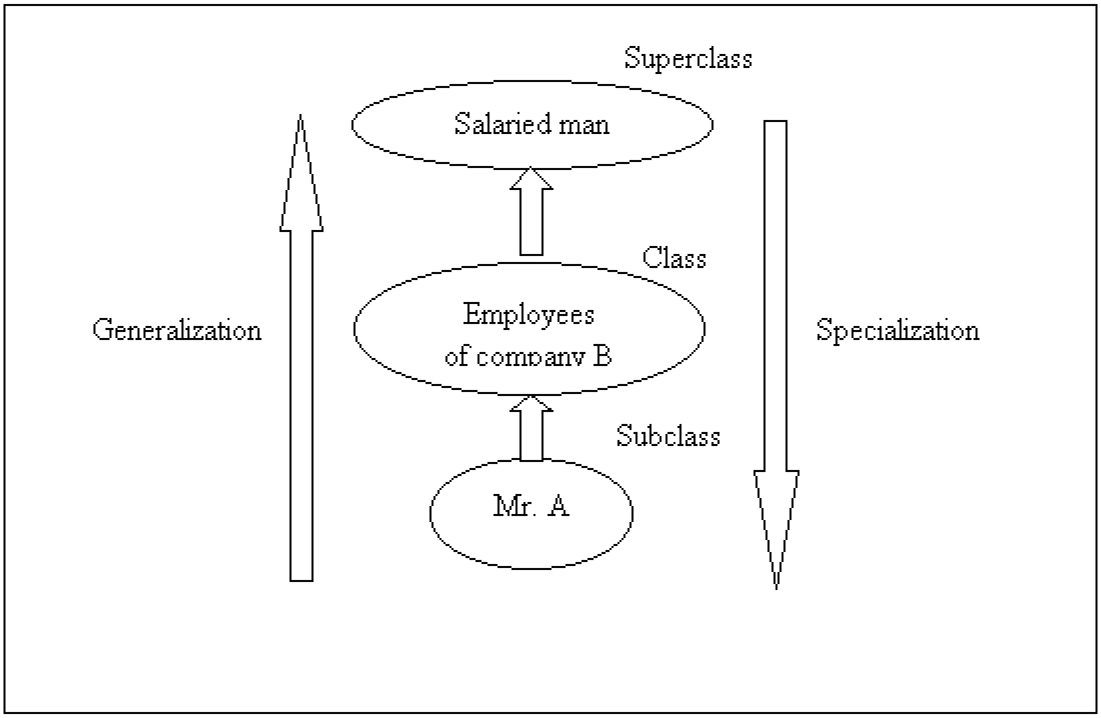
The is-a association means a parent-child association between classes. The following figure shows an example of an association between classes.



[Example of association between classes]

Relative to any specific class in a hierarchy, the classes above the class in the hierarchy are called superclass, and classes below the specific class are called subclasses. From this, the relationship "a subclass *is a* superclass" is derived. The creation of subclasses from superclass by refining the superclass is called'specializtion', and the relationship between a class and one or more refined or specialized version of it is called 'generalization'.

The following figure shows a simple example of an is-a relationship.



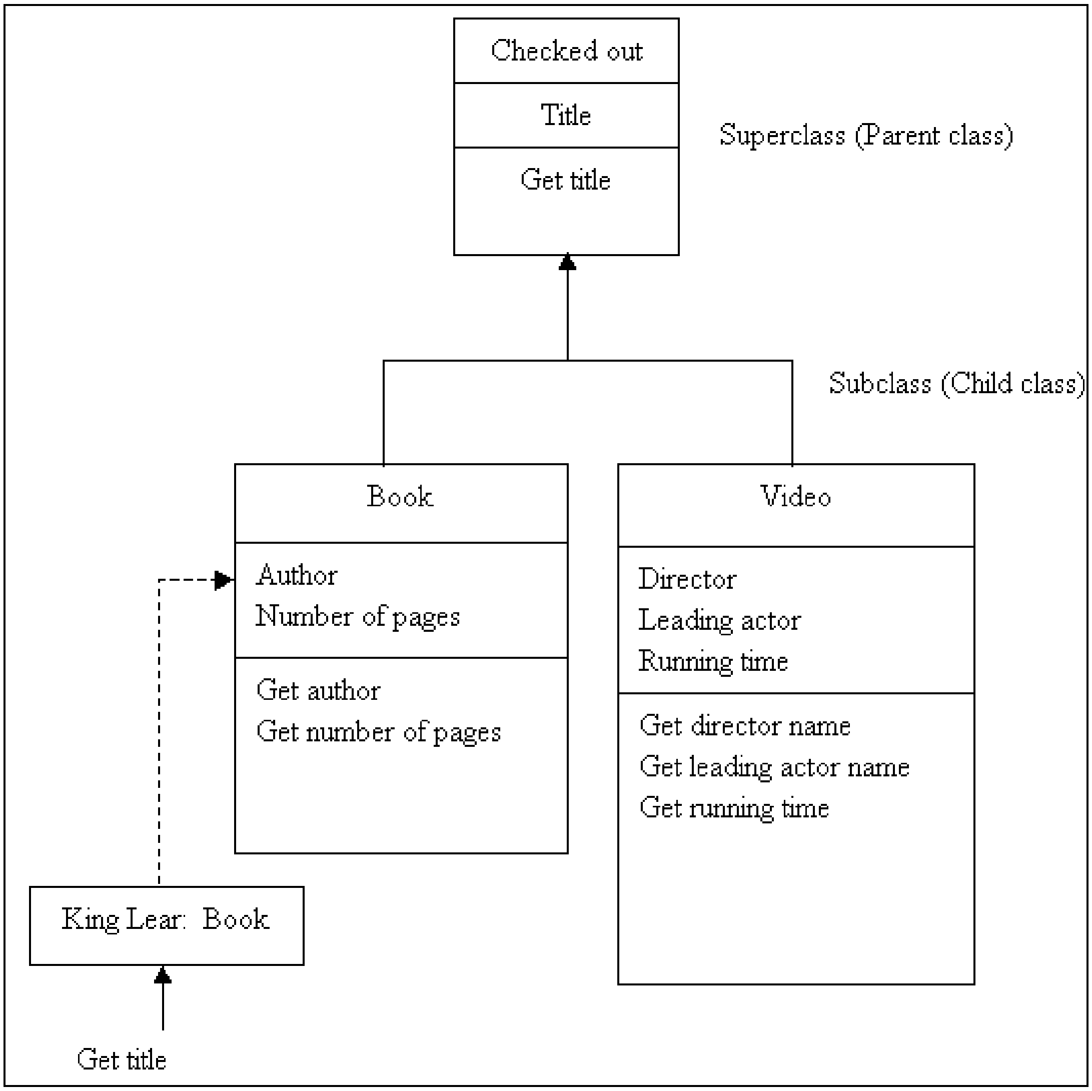
[Example of is-a relationship]

The is-a relationship establishes a logic, “A is B, and B is C; therefore, A is C.”
In this figure, this logic can be described as follows:
A is B. [***] Mr. A is an employee of company B.
B is C. [***] An employee of company B is a salaried man.
Therefore, A is C. [***] Mr. A is a salaried man.

A child class inherits all the characteristics of the parent class, and also it can have its own unique characteristics consisting of attributes and methods.

Next, inheritance is explained. Inheritance means that child classes inherit the characteristics of the parent class. The inheritance mechanism exists between classes. By using this mechanism, the characteristics defined in the parent class are inherited to child classes. There are two types of inheritances: single inheritance and multi-inheritance. Single inheritance means that a subclass inherits characteristics from only one superclass. Multi-inheritance means that a subclass inherits characteristics from multiple superclasses.

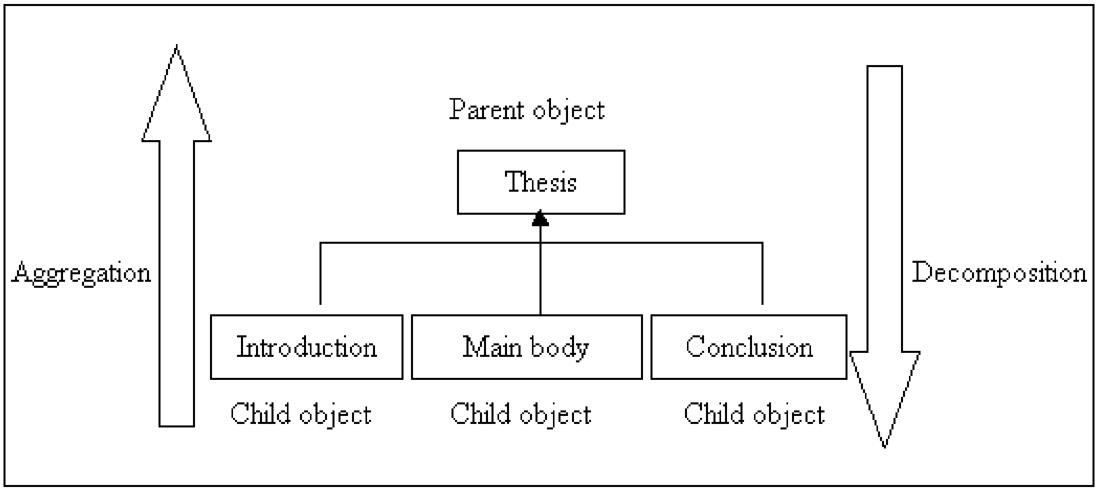
The following figure shows an example of inheritance.



[Example of inheritance]

1.4.2 Structured aggregation relationship between objects (part-of)

The part-of relationship is a parent-child relationship between objects. Relative to any specific object in a hierarchy, an object above the object in the hierarchy is called a parent object; an object below is called a child object. From this, the relationship "a child object *is a part of* the parent object" is derived. Partitioning a parent object to child objects is called decomposition. Obtaining a parent object from child objects is called aggregation. The following figure shows an example of a part-of relationship.

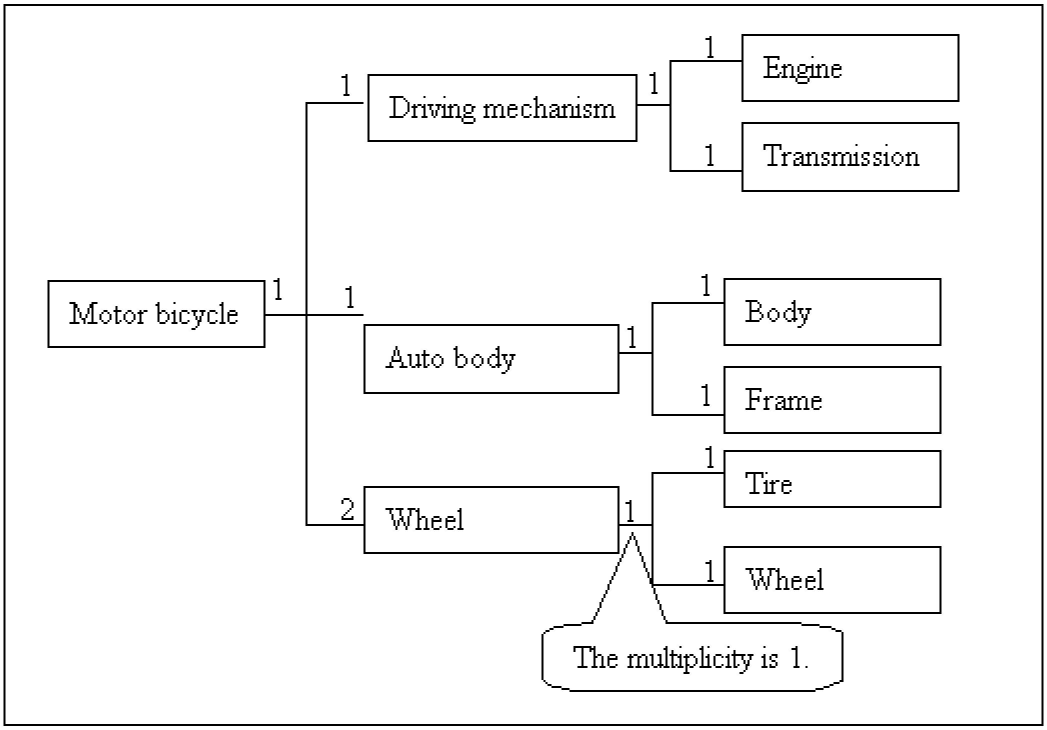


[Example of part-of relationship]

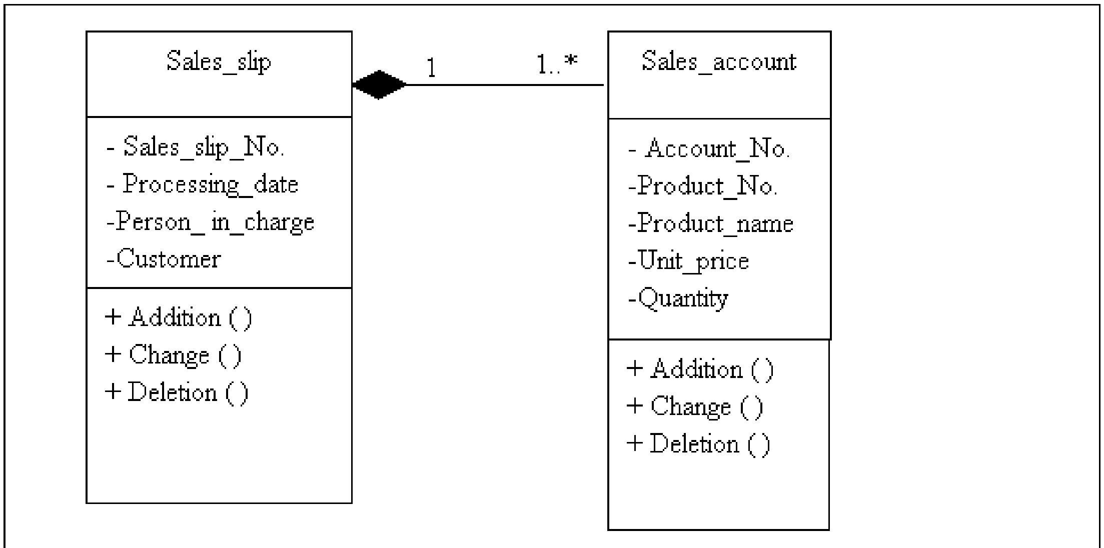
As shown in this figure, the thesis object consists of the introduction, main body, and conclusion. In other words, it can be expressed as "the conclusion is a part of the thesis." This expression can be changed to "the thesis has a conclusion"; therefore, the relationship between objects is also called "has-a relationship."

Next, composition, which is one form of aggregation, is explained. A composition has a characteristic relationship, "the existence of the partial object strongly depends on the existence of the entire object." When the dependence relationship is established, the contents cannot be changed.

The following figure shows an example of composition.



[Example of composition 1]



The sales slip consists of multiple sales accounts. Therefore the multiplicity is 1 in the sales slip that indicates the whole, and 1 or more in the sales account that indicates a part included in the whole.

[Example of composition 2]

As shown in this figure, the composition has a characteristic, “the multiplicity of the whole is always 1.”

1.5 Polymorphism and Binding

As explained in the chapter, messages are used for references, etc. between objects in the object-oriented paradigm. The object-oriented paradigm has two concepts: 1) polymorphism, which makes the messages clearer and more rationally use them and 2) binding. This section explains these concepts.

1.5.1 Polymorphism

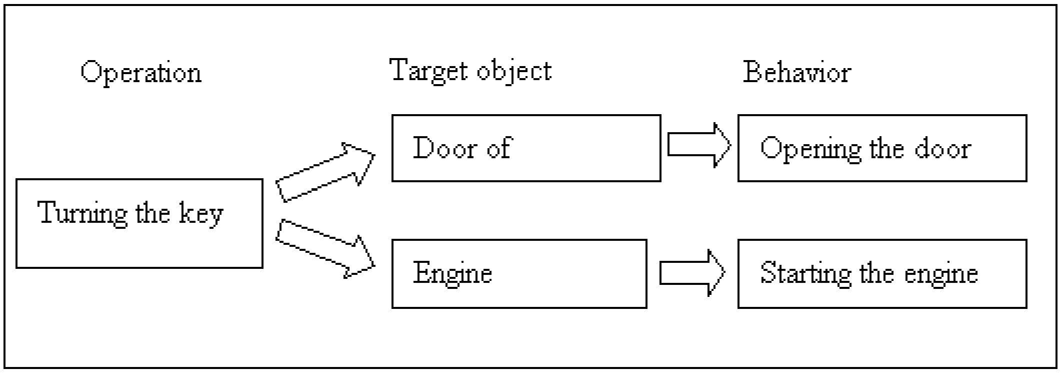
Polymorphism means to obtain the expected result by sending only one message, even if the message is sent to various objects. To reference the encapsulated data in an object, a procedure which is public to the outside must be called. A means for calling the procedure is a message. When sending a message to an object, use the following message expression:

object name (target object) + message (operation)

In the object-oriented paradigm, a target object is explicitly described in the message expression. Therefore only the information indicating the operation is necessary, and information about the target object is not necessary.

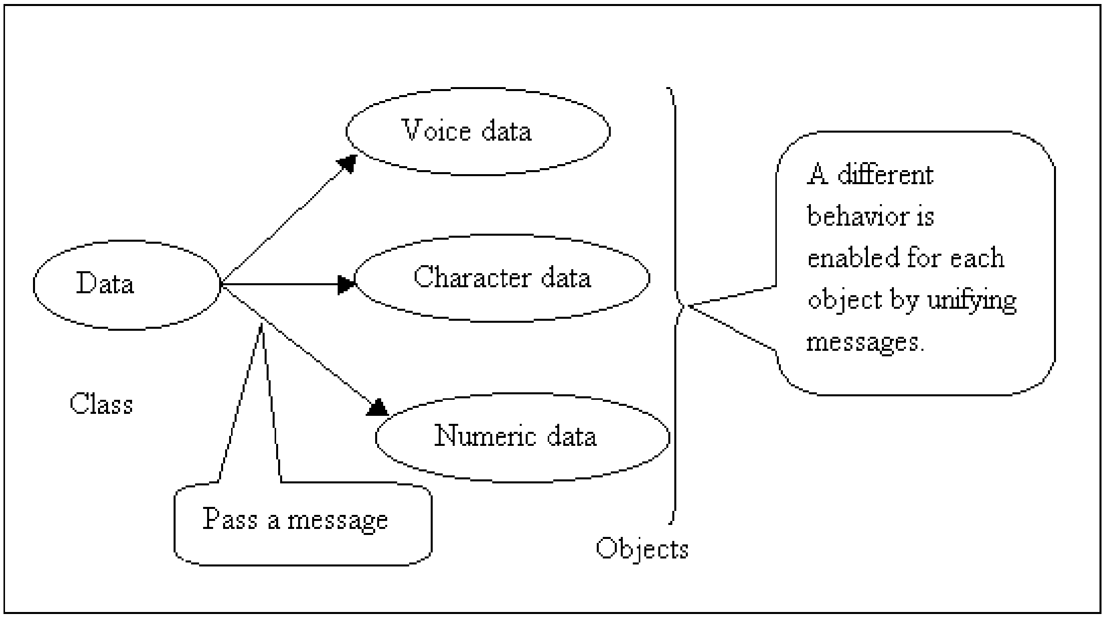
Therefore, when the target object is “door of automobile”, opeartion “turning the key” is interpreted as behavior “opening the door.” When the target object is “engine,” operation“turning the key” is interpreted as behavior “starting the engine.”

Like this, expressing many different operations with one message is called polymorphism.



In addition, there are basic operations such as data addition, change, and deletion. In this case, when target objects have different attributes or behaviors, a simple model can be obtained, because messages can be unified into these three.

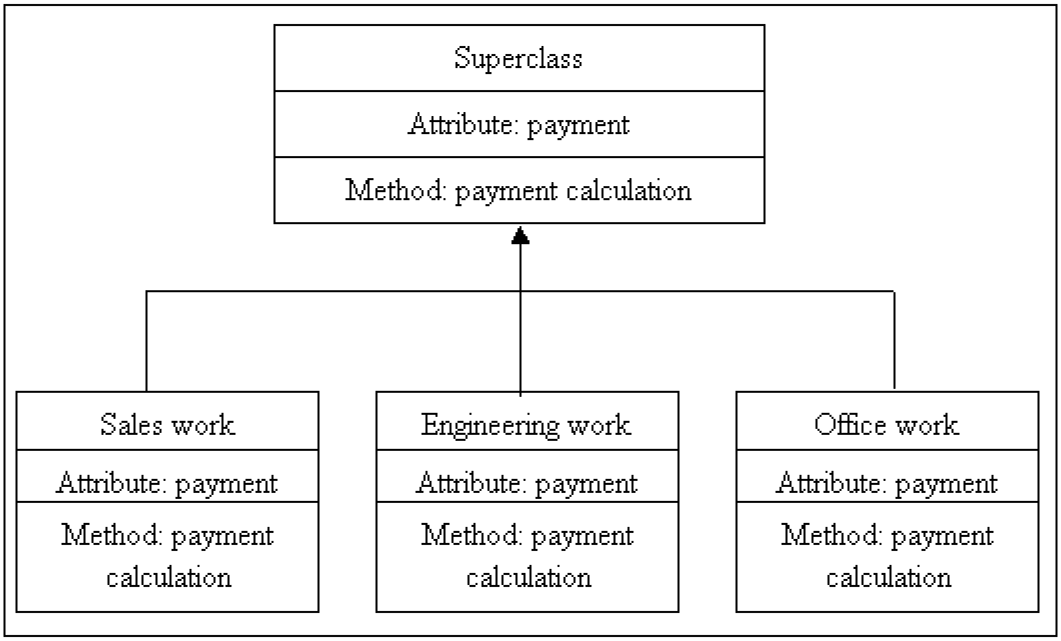
The following figure shows an example.


Voice data
Data
Character data
Class
Numeric data
Pass a message
Objects
A different behavior is enabled for each object by unifying messages.

[Example of basic operations for data]

As shown in this figure, even if the form for each data item is unknown, a different behavior is enabled for each different data item by sending an unified message. The merit of polymorphism is unifying message, and designing a complicated system via a simple interface.

The following figure shows an example of polymorphism between a superclass and subclasses.



[Example of polymorphism between superclass and subclasses]

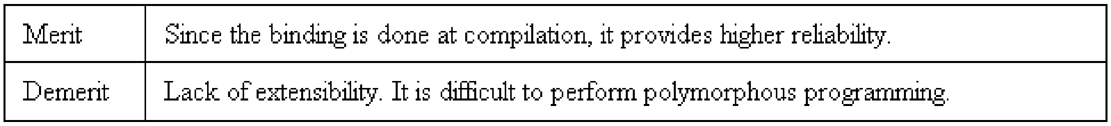
In this figure, the same operation is performed by all the subclasses. However, the actual procedure is different in the payment calculation by each subclass. This means that the actual calculation procedure can be left to a subclass if only the interface is know by the user when subclass is called by a subclass.

Like this, in the object-oriented paradigm, using polymorphism, a program having various information items can be described with a few functions. In other words, the object-oriented paradigm can express various events with a small vocabulary, and it is closer to natural languages than some programming languages.

1.5.2 Dynamic binding

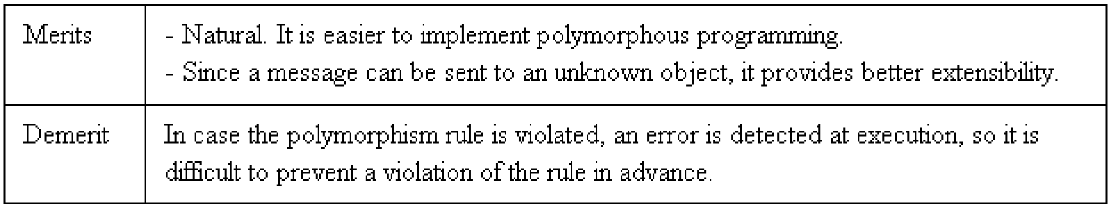
The binding between message and method means that an object receiving a message executes its behavior. Dynamic binding is a form of binding, and it means that all bindings are performed at execution of a program. Conversely, static binding means that all bindings are made at compilation of a program. The following tables summarize the merits and demerits of static and dynamic bindings.

[Static binding]



Merit	Since the binding is done at compilation, it provides higher reliability.
Demerit	Lack of extensibility. It is difficult to perform polymorphous programming.

[Dynamic binding]



Merits	- Natural. It is easier to implement polymorphous programming. - Since a message can be sent to an unknown object, it provides better extensibility.
Demerit	In case the polymorphism rule is violated, an error is detected at execution, so it is difficult to prevent a violation of the rule in advance.

As shown in this table, dynamic binding has merits and demerits. In the demerit row, the rule means a prerequisite “the message target object can always execute the operation.” Compared with static binding, dynamic binding provides high extensibility and generalization, but low reliability as shown.

Polymorphism and dynamic binding have been explained. Since using either of them alone is not particularly effective, both should be used together. The characteristics and merits of polymorphism are as follows.

(1) Characteristics

Polymorphism enables multiple different behaviors by sending the same message to different objects.

(2) Merit

When adding a new object, one need not rewrite a program by using the same protocol (message) as the existing one for the added object. This merit enhances the cognitive economical efficiency in the object-oriented paradigm. The cognitive economical efficiency means that one can easily understand the expression by polymorphism because there is similarity between the expression of polymorphism and that of the human language in recognition and understanding.

The characteristics and merits of dynamic binding are as follows:

(1) Characteristics

Dynamic binding enables the sending of a message regardless of the message target object.

(2) Merit

Dynamic binding offers good system extensibility and generalization.

Combining the merits of polymorphism and dynamic binding offers the following effects.

(1) A new object can easily be added to an existing system. The addition of a new object hardly affects existing programs.

(2) The maintainability and extensibility of the system are increased, because of item (1).

[Reference: Stereotypes of classes]

The stereotypes are specialized versions of classes classified by their meanings. For each class, only one stereotype can be specified. The following are a few examples of stereotypes.

(1) Entity class

This is a class that indicates a thing or the concept in the real world. It falls under a class recognized as a class at the early object-oriented stage or an entity in the ER diagram.

(2) Boundary class

This is used to manage communication used as an interface between the environment surrounding the system and the processing within the system. Examples are the screen used as a user interface and a communication protocol used as an interface with an external system. In other words, the boundary class is a class on the boundary between the system and the outside.

(3) Control class

This is a class holding a logic that cannot be assigned to the entity class. This class has behavior about the control particular to one or more use cases. The use cases are used to represent in the analysis process specific functions supplied by the system. It indicates a function at a larger level when viewed from outside.

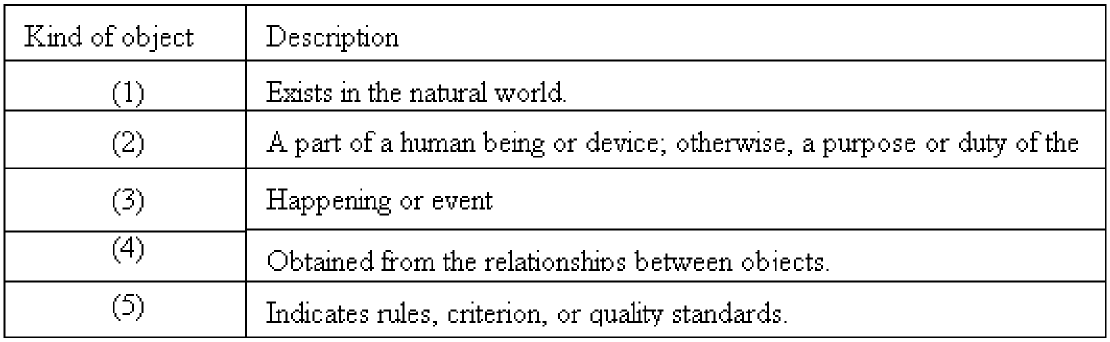
Exercises

1. For the object-oriented model and design, select the appropriate word or phrase from the set of answers for [***] in the following description.

The object-oriented paradigm is a new concept for solving problems using a model [(2)] on the basis of the concept of the [(1)]. The fundamental element is [(3)], and it has the [(4)] structure and its behavior in a single [(5)].

Answers
a. entity
b. data
c. program
d. real world
e. object
f. organized

2. In the table of kinds of objects below, select the appropriate word or phrase for [ ] from the set of answers.



Kind of object	Description
(1)	Exists in the natural world.
(2)	A part of a human being or device; otherwise, a purpose or duty of the
(3)	Happening or event
(4)	Obtained from the relationships between objects.
(5)	Indicates rules, criterion, or quality standards.

Answers
a. Incident
b. Specifications
c. Material being
d. Role
e. Interaction

3. Select the correct item(s) regarding abstraction.
(1) Abstraction means grouping objects.
(2) Abstraction means extracting the characteristics of one object.
(3) Abstraction is performed in the top-down approach.
(4) Abstraction means extracting the characteristics of multiple objects and obtaining the common ones.

4. Write one merit of using the concept of abstraction.

5. Select the appropriate word or phrase from the set of answers for [***] in the following description regarding attributes.

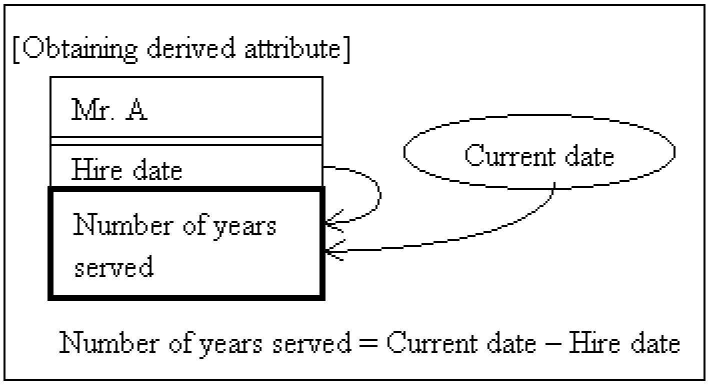
The attribute is [(1)] that is [(2)] by the action given from the [(3)] in an [(4)].

Answers

a. program
b. outside
c. changed
d. object
e. internal state

6. Select the appropriate word or phrase from the set of answers for [   ] in the following description regarding derived attributes.

The derived attribute is an attribute [(1)] from one or more [(2)] as shown below.


[Obtaining derived attribute]
Mr. A
Hire date
Number of years served
Current date
Number of years served = Current date – Hire date

Answers

a. Hire date
b. derived
c. Current date
d. attributes

7. Select the correct item(s) regarding methods.

(1) A method makes public the processing.
(2) A method does not respond to an external inquiry.
(3) A method is a function that responds to an external inquiry or performs the appropriate behavior.
(4) A method always performs a behavior by itself.

8. Select the appropriate word from the set of answers for [ ] in the following description regarding the demerits of structured design techniques.

(1) The modification of [(1)] frequently occurs in the [(2)] in the structured design techniques.

(2) In structured design techniques, a [(3)] function is created to implement a [(4)] function. If a [(4)] function is changed, it may [(5)] a [(3)] functions and others.

(3) Since the structured design technique is function-oriented, [(6)] that is easy to implement the functions tends to be prepared; therefore, maintaining the consistency of the [(6)] is difficult.

(4) Since the [(7)] information is placed at different locations, changing the data structure may affect multiple [(8)] in a system.

Answers

a. affect
b. low-level
c. specifications
d. programs
e. high-level
f. functions
g. same
h. data

9. Select all correct items regarding the object-oriented paradigm.

(1) Since the visualization of programs is possible, the development efficiency becomes higher.

(2) In the object-oriented paradigm, an object is created as a component, which is effective only in the system.

(3) In the object-oriented paradigm, a model can be built using natural concepts; it is not effective in manufacturing processes.

(4) Since a program is separated into independent objects, it can be modified easily.

10. Select the appropriate word or phrase from the set of answers for [ ] in the following description regarding encapsulation.

Encapsulation is a structure into which [(1)] and [(2)] are put together. In other words, encapsulation is [(3)] [(4)] and only the [(5)] is [(6)] to the outside. The external [(7)] can operate on [(8)] by activating the [(6)] operation.

Answers
a. data
b. operation
c. data structure
d. program
e. attributes
f. public
g. private
h.methods

11. Enter the appropriate word in [ ] in the following description regarding the merit of encapsulation.

The merit of encapsulation is hiding [(1)], so that it cannot be [(2)] directly from the outside. Therefore, when [(1)] is changed, the [(3)] does not affect the outside, that is, [(4)].

12. Enter the appropriate word in [ ] in the following description regarding the object diagram.

The object diagram describes an [(1)]. The [(1)] is [(2)] generated by assigning actual [(3)] to [(4)].

13. Select the appropriate word from the set of answers for [ ] in the following description regarding method invocation.

Method invocation is calling a [(2)] in a target object when a specific object references [(1)] of another one. This is used when objects proceed with processing [(3)], and it is performed by sending [(4)] to each other.

Answers
a. messages
b. attributes
c. method
d. in collaboration

14. Enter the appropriate word in [ ] in the following description regarding associations.

An association between objects is a [(1)] relationship that can be expressed as “object 1 [(2)] object 2.”

15. Enter the appropriate word or phrase in [ ] in the following description regarding derived associations.

The derived association is a relationship that can be obtained from [( )].

16. Select the appropriate word or phrase from the set of answers for [ ] in the following description regarding referencing between objects.

A reference between objects is established when an object accesses another one, by passing a [(1)] to another to activate a [(2)]. For example, when referencing or altering an attribute of another object, processing is executed using the [(3)] of the object. A mechanism that requests behavior for another object like this is called [(4)].

Answers
a. method
b. message passing
c. a message
d. behavior

17. Select the appropriate phrase from the set of answers for [ ] in the following description regarding instances.

The instance is [(1)] generated by assigning actual data to a class. The instance therefore has [(2)] in the same structure as that of the class. For the [(3)] and [(4)], the same thing as the class is [(5)] by multiple instances.

Answers
a. shared
b. information
c. attributes
d. an object
e. operations
f. inheritance

18. Enter the appropriate word in [ ] in the following description regarding multiplicity.

The multiplicity indicates how many objects in other classes are related for one [( )] in a specific class.

19. Select the appropriate word from the set of answers for [ ] in the following description regarding the use of the qualifier.

The qualifier is [(1)] used to distinguish a specific [(2)] from multiple [(2)] in the related classes, and it reduces [(3)].

Answers
a. multiplicity
b. object(s)
c. method
d. an attribute

20. Explain the is-a association.

21. Select the appropriate word from the set of answers for [ ] in the following description regarding inheritance.

The mechanism of inheritance, which exists between [(1)], inherits the [(2)] defined in the parent class to a child class. There are two types of inheritance: [(3)] and [(4)]. The [(3)] is inheriting from only one parent class to child classes. The [(4)] is inheriting from multiple parent classes.

Answers
a. multiple-inheritance
b. classes
c. characteristics
d. single inheritance

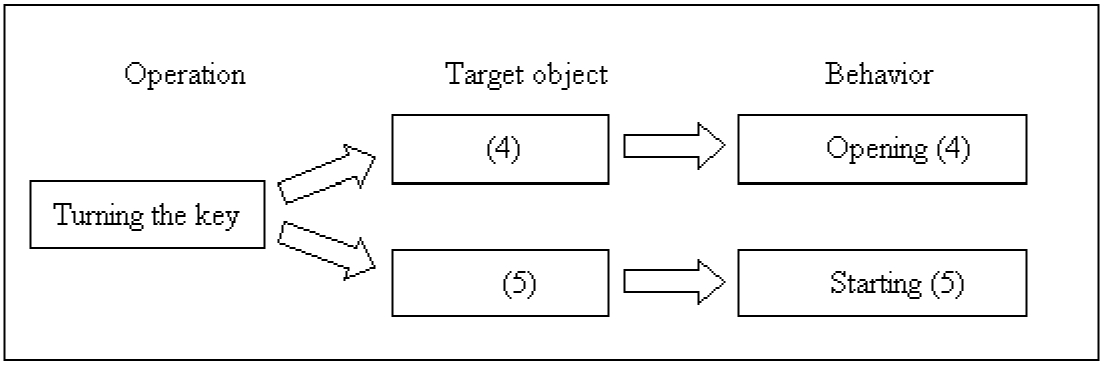
22. Explain the part-of association.

23. Enter the appropriate word or phrase in [ ] in the following description regarding the characteristic of composition.

- The existence of [(1)] strongly depends on the existence of [(2)].
- When [(3)] between the [(1)] and [(2)] is established, the contents cannot be changed.
- The [(4)] of the whole is always 1.

24. Select the appropriate word from the set of answers for [ ] in the following description and figure below about polymorphism.

Polymorphism means to obtain the expected [(1)] by sending only [(2)] even if [(3)] message is sent to various objects. The operation, “turning the key,” is shown below as an example.



Answers
a. the door
b. messages
c. result
d. corresponding
e. the same
f. the engine

25. Select the appropriate word or phrase for [ ] in the following description regarding dynamic binding.

Dynamic binding means that all bindings are performed during execution of a program. The merits of dynamic binding are that it is easy to do the [(1)], and that [(2)] is high, because [(3)] can be sent even to an object that is unknown. Like this, dynamic binding provides high [(2)] and [(4)], but [(5)] low.

Answers
a. reliability
b. extensibility
c. generality
d. messages
e. polymorphic programming

2 UML

Chapter Objectives

This chapter explains an overview of UML (graphical notation).

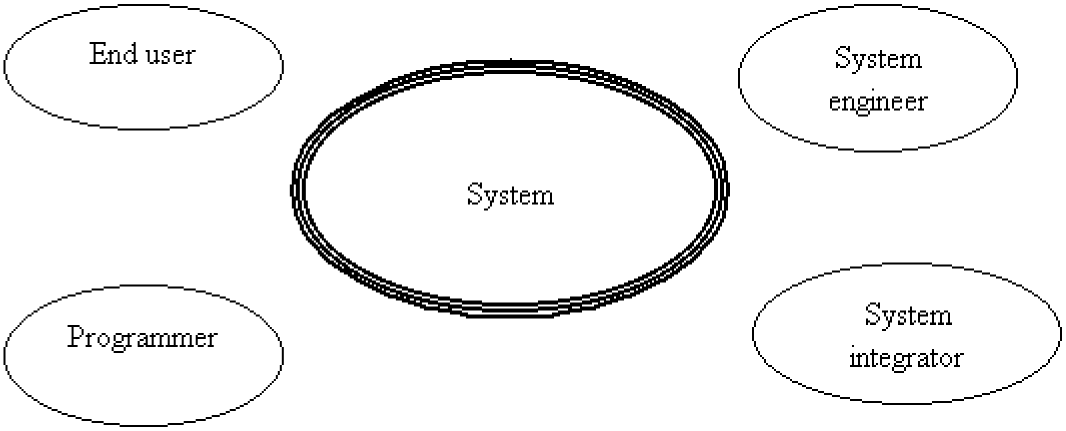
2.1 Views and Diagrams
2.2 Use Case Modeling
2.3 Static Model Description
2.4 Behavior Model Description

2.1 Views and Diagrams

The unified modeling language (UML) can be used in the software lifecycle from object-oriented analysis to object-oriented implementation. As an introduction to the UML concept, this section explains views and diagrams.

2.1.1 Views

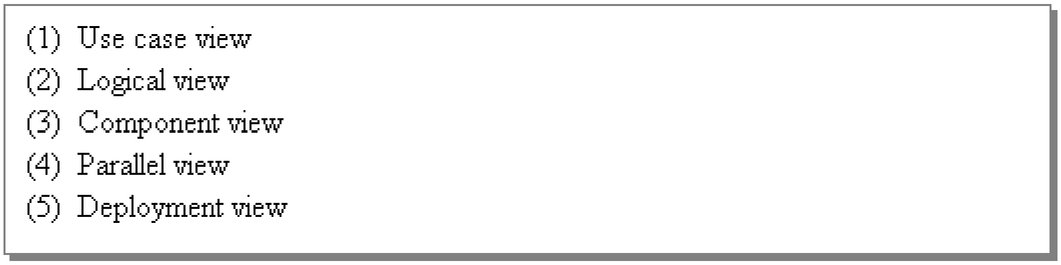
A system is developed by personnel, such as end users, system engineers, programmers, and system integrators.



The system development personnel define a system from various aspects to understand the system design in more detail. This kind of expression for a system model is called a view.

Several views are integrated into a system.

Views can be classified into five types:



(1) Use case view
(2) Logical view
(3) Component view
(4) Parallel view
(5) Deployment view

An architecture expresses in an easy-to-understand manner the important characteristics of an entire design.

This is very important for system understanding and evolution, organization of development, and promotion of reuse.

For each view, the functions described and developers intended for are explained in the following:

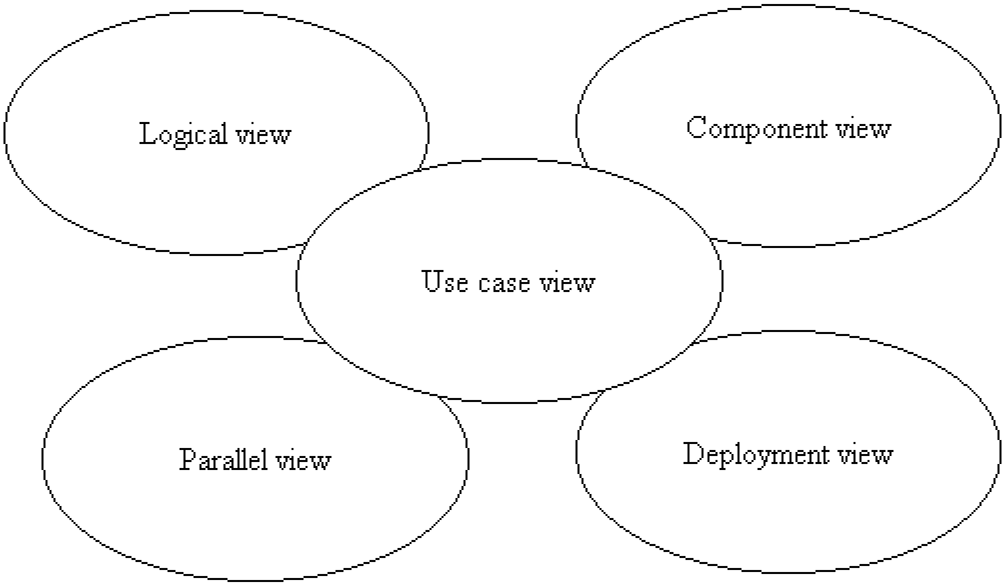
(1) Use case view
A use case view describes what the user requires from the system or the functions which the system supplies to the user. The use case is the core view, and the other views are created on the basis of this view. This view is for end users, system engineers, and programmers.

(2) Logical view
A logical view describes necessary system functions from static and dynamic aspects. This view is for system engineers and programmers.

(3) Component view
A component view describes the dependencies of source codes and binary codes. This view is for programmers.

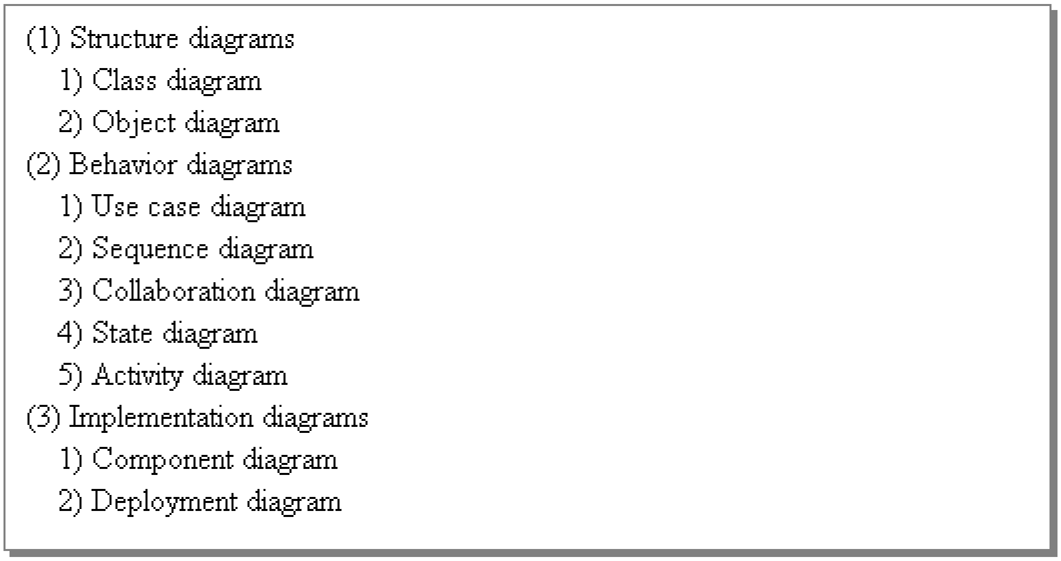
(4) Parallel view
A parallel view describes synchronous and asynchronous communication events in a parallel processing system. This view is for programmers and system integrators.

(5) Deployment view
A deployment view describes the physical location and connections of computers and other devices. This view is for integrators and testers.



2.1.2 Diagrams

A diagram is a graphical expression of view contents. The UML uses nine types of diagrams listed below. Diagrams can be classified into structure diagrams expressing system statistical structures, behavior diagrams expressing system behaviors, and implementation diagrams expressing components and deployment.



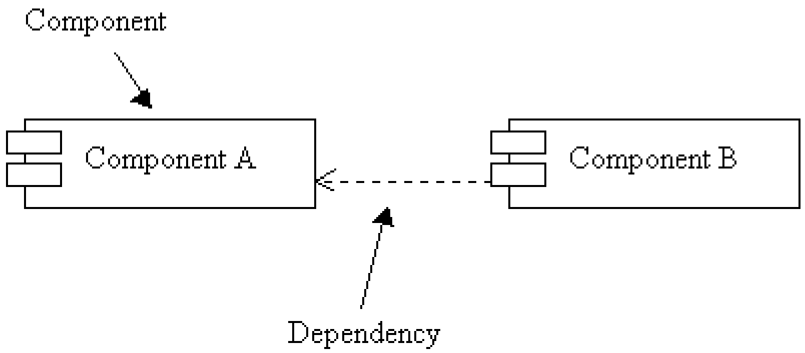
(1) Structure diagrams
   1) Class diagram
   2) Object diagram
(2) Behavior diagrams
   1) Use case diagram
   2) Sequence diagram
   3) Collaboration diagram
   4) State diagram
   5) Activity diagram
(3) Implementation diagrams
   1) Component diagram
   2) Deployment diagram

The diagrams are detailed in the following section. This section outlines each diagram.

(1) Structure diagrams
1) Class diagram
The class diagram shows the static structure of the target domain or system in terms of their relations, with the class.
2) Object diagram
An object diagram shows instance-level static structures.
(2) Behavior diagrams
1) Use case diagram
A use case diagram describes the method of using the system from an external point of view.
2) Sequence diagram
A sequence diagram describes how a set of objects interact in time.
3) Collaboration diagram
A collaboration diagram describes messages exchanged between collaborating objects with connections relationship between objects (collaboration contexts).
4) State diagram
A state diagram describes a set of the states of an object can have, transitions between states and the events that trigger the transitions, and actions accompanying transitions.
5) Activity diagram
An activity diagram describes several related activities in time order as a group of jobs or a series of processes.

(3) Implementation diagrams

1) Component diagram

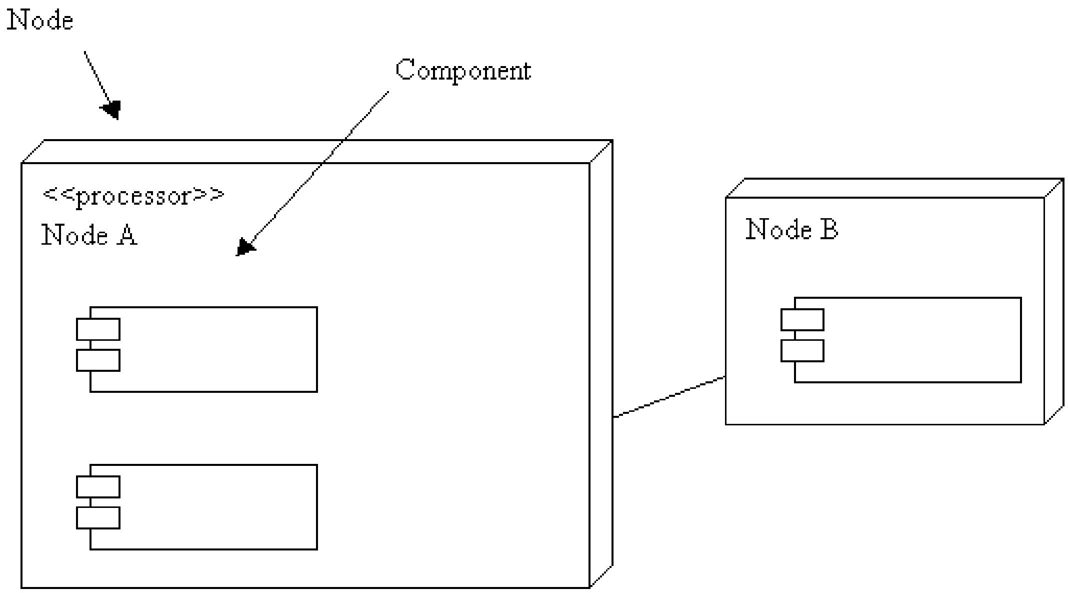


A component diagram shows the dependency between components like binary code's depending on source code, and clarifies the component change on other components.

A dependency means the relation of use between components. Dependency is indicated by a dotted arrow.

2) Deployment diagram

A deployment diagram expresses the deployments of software to hardware represented as nodes. Servers, clients, processors, and devices in a network are abstracted into nodes. The deployment diagram shows where processes and threads are allocated to nodes and where components are arranged. Communications and network connections between processes are expressed as relationships between nodes.



A cubic symbol represents a node. A stereotype can be used to emphasize the characteristic of the node. The stereotypes which are generally used are <<processor>> (that can execute programs) and <<device>> (terminal or other). A solid line between nodes indicates that these nodes can communicate with each other. This usually means direct hardware coupling.

A component represents a module operating at execution time.

UML-related Knowledge

(1) Agent-orientation

Agent-orientation is attracting attention as a next-ordinary object. Unlike general objects, this object has high-grade functions, such as intelligence, autonomy, adaptability, and portability. By integrating the object and artificial intelligence technologies, agent orientation will be achieved. By combining Internet technologies, a variety of solutions are provided.

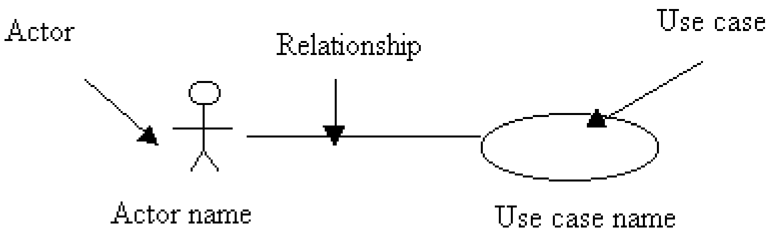
(2) Framework

A framework is a basic software framework provided for each specific domain or application as an expansible template. A framework does not work alone as an application, because it does not have concrete information. However, defining subclasses to fill the missing information completes the framework as an application.

2.2 Use Case Modeling

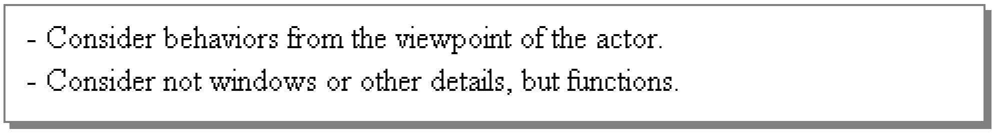
2.2.1 Use case diagrams

A use case diagram describes the method of using a system from an external point of view. A use case diagram is shown together with types of system users (actors). System requirements are encapsulated into use cases as the units, and are traced through the use case diagrams analysis models and design models. If a use case diagram is used, a series of works, such as system analysis, design, implementation, and testing, can be managed with relationships among them. Therefore, a use case can be used not only at the requirements phase itself, but in all the system development processes.



A human-like symbol represents an actor, and a use case is enclosed with a solid-line eclipse.

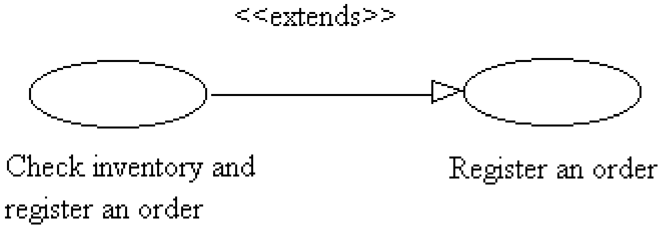
(1) Finding a use case
A use case represents a system function (a behavior viewed from outside the system)started by an actor for each job. The points of finding a use case are as follows:


- Consider behaviors from the viewpoint of the actor.
- Consider not windows or other details, but functions.

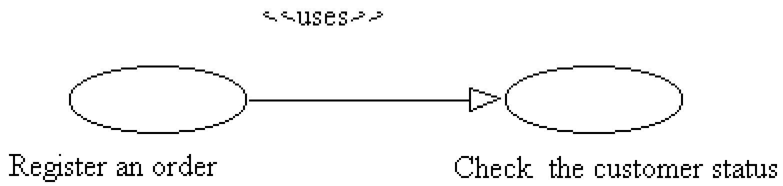
For an order management (telephone ordering system), a use case may be register an "order " or inquire an "order".

(2) Relationship between use cases
1) Extension: <<extends>>
This indicates a relationship between a general use case and a special use case giving its variation or exception. A blank-headed arrow is drawn from the special use case to the general use case.

The following example shows the use cases of "Checkinventry and register an order" and "Register an order entry."



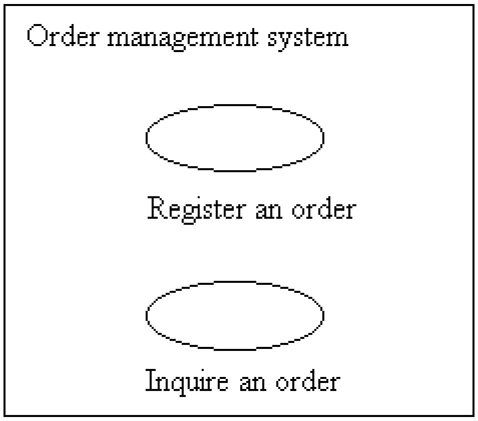
2) Use <<uses>>
This indicates a relationship where a use case internally calls and uses an instance of another use case. In the same way as for a subroutine call, a blank-headed arrow is drawn from the use case that calls the instance to the common use case.



The example below shows the use cases of "Register an order" and "the customer status check."

2.2.2 Systems

In a use case diagram, a group of use cases is enclosed with a large square to clarify the border (system borders) between use cases and actors, and a system name is indicated in the upper part inside the rectangle.

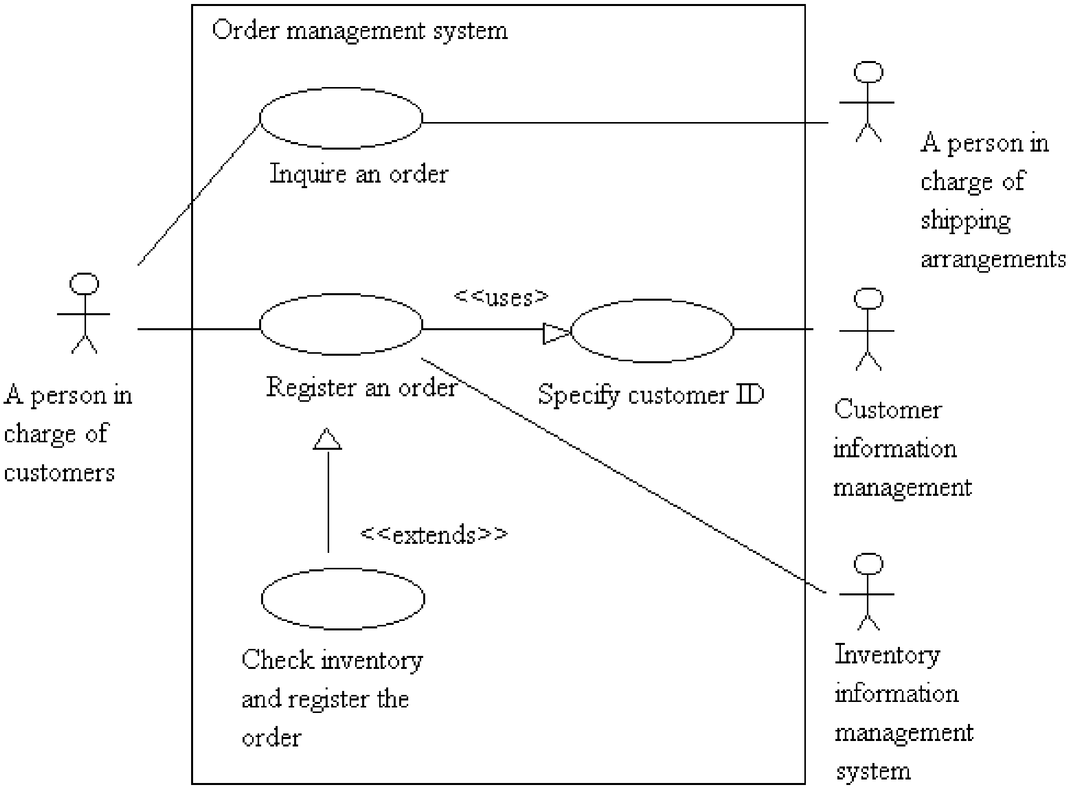


2.2.3 Actors

A use cases is activated by an actor. An actor actively exchanges information with the system or passively receives information from the system. The actor may be the end user, the hardware, or an external system (a system not to be developed).

In the order management system, for example, the person (actor) in charge of customers executes two use cases "Register an order" and "Inquire an order". The customer information management system actor and inventory information management system actor also get involved in the use case of "Register an order" and the actor in charge of shipping arrangement gets involved in that of "Inquire an order".

A use case diagram for the this example is as follows:


Order management system
Inquire an order
A person in charge of shipping arrangements
<<uses>
Register an order
Specify customer ID
A person in charge of customers
Customer information management
<<extends>>
Check inventory and register the order
Inventory information management system

In parallel with creating a use case diagram, documents are usually created for each use case. The documents are an outline describing the purpose of the use case in a few lines, an event flow, and a set of scenarios describing the contents of the use case in detail.

(1) Outline
The outline describes the purpose of the use case in a few lines.

(2) Event flow
The event flow describes in sentences when and how to start or end the use case and also interactions between the use case and actors.

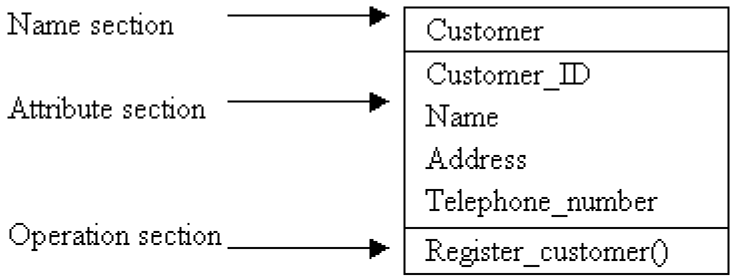
(3) Scenario
A scenario describes an example of the use case. Each use case is expressed as a set of scenarios.

2.3 Static Model Description

2.3.1 Class diagrams

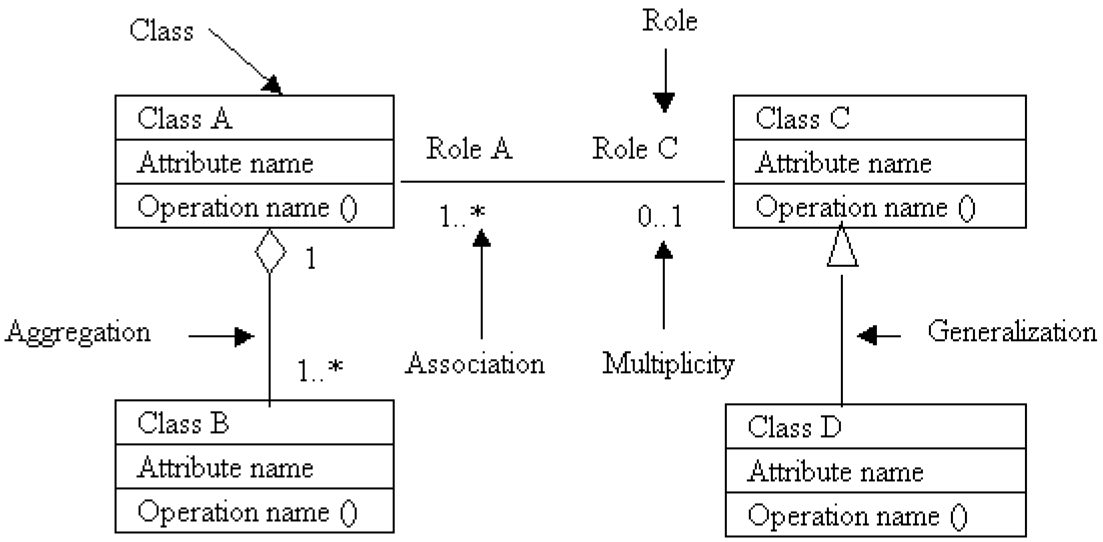
A class is an abstraction of a set of objects that have the same properties.
Class classification allows the systematic control of objects contained in a system. This also promotes understanding based on similarities and reuses of objects as software modules.

A class is represented as a rectangular icon partitioned into three sections delimited by horizontal lines. The name section at the top indicates the class name and properties; the attribute section in the middle lists the attributes; and the operation section at the bottom lists the operations. Either or both of the attribute and operation sections can be hidden.



A class diagram expresses a system static structure as classes and their relationships. A system structure is shown by describing a variety of relationships (association, aggregation, generalization, and dependency) between the classes.

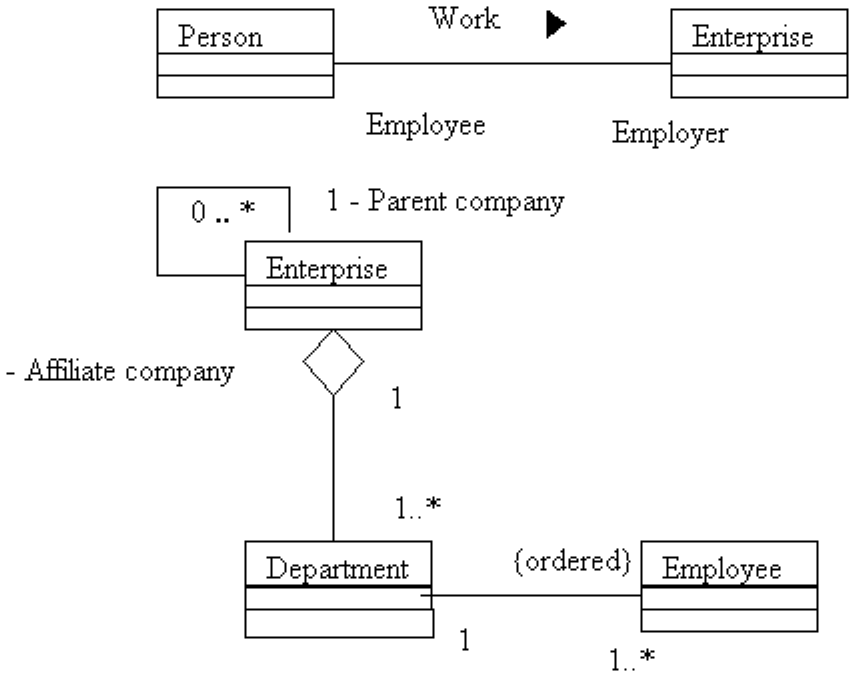
In a class diagram, several classes can be grouped into a package for system management. A package also becomes a component of the diagram.



2.3.2 Relationships

(1) Association

An association is a temporary relationship between classes. Classes do not always have relationships with each other. Here is an example of association qualifiers.



1) Association name

For an association name, an arbitrary name is indicated at the center of the association (do not confuse with a role name). A small black triangle can be added to specify the direction for reading names. (from the subject to an object).

2) Role name

A role name indicates the purpose, position, and role of a class in association with another class. This is described as a character string near the end of a line segment. A role name is optional but cannot be hidden.

3) Multiplicity

Multiplicity indicates the range of the number of instances (objects) that associated classes may have. This can be hidden. An integral range, including zero, is specified in the following format:

Lower limit..Upper limit

If an asterisk (*) is specified for the upper limit, the multiplicity does not have the upper limit. If one integral value is specified, the multiplicity is limited to the integral value only.

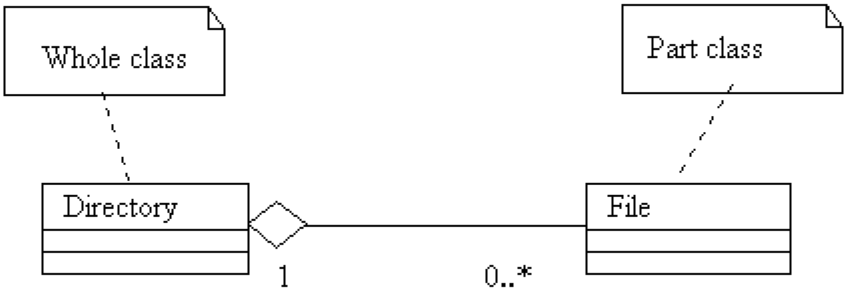
4) Visibility

For visibility, an indicator (+, #, -, or explicit keyword like [public]) is added before a role name to indicate how the object having the role looks like from the client of the other object.

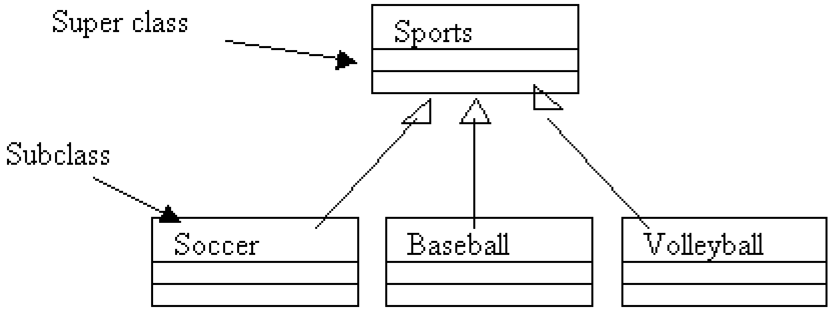
5) Ordering
Ordering is specified if the multiplicity is greater than 1. The element order is determined only when ordering is specified.
a) No ordering
A set of elements not in order is formed. This is the default.
b) Ordering
Elements are listed with no duplicate order numbers. Specify ordering or no ordering by an association property or in brackets (e.g. {ordered}).

6) Inductive visibility
Inductive visibility indicates that the class can (needs to) send a message to or reference information at the other related class.
An arrow is attached to the end of an association to indicate that inductive visibility is supported for the class to which the arrow points.

(2) Aggregation
Aggregation is a special form of association that is used when the relationship of the whole and part hold true between associated objects.
This association is described by adding a white diamond to the end of an association line segment.

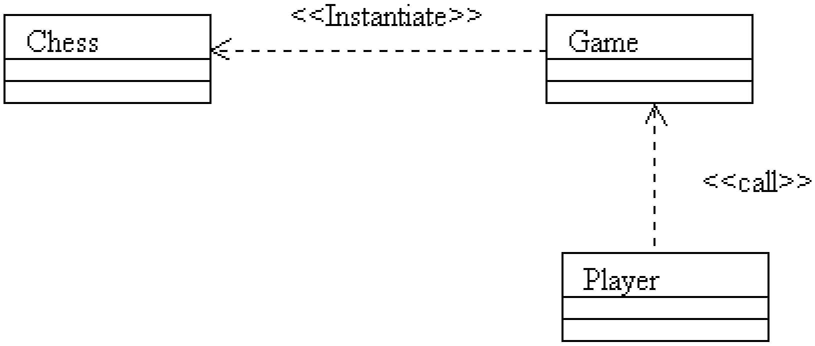


(3) Generalization
Generalization indicates a relationship of inheritance between general elements and special ones excluding them. Structures or behaviors common to one or more elements are shared as a general element (superclass) and element-unique factors are defined as special elements (subclasses).



A blank-headed solid-line arrow is drawn from a special element to a general element. For generalization between classes, an abstract class may be used.
An abstract class defines attributes or behaviors common among subclasses and is a class for a generalization hierarchy. An abstract class may not have any object.

(4) Dependency
Dependency indicates a semantic relation between two or more model elements. This means that a change to the target element (on which another depends) may necessitate a change to the source of dependency (element which depend on another).

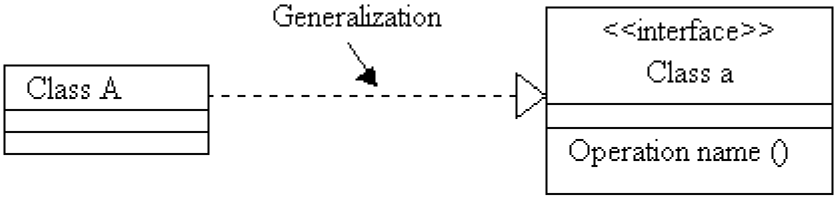


For dependency, a dotted line is drawn from the depending side to the dependent side between two or more model elements. The arrow may have an arbitrary name as its label.

2.3.3 Interfaces

An object is a basic unit for UML model expression, and its attributes or operations can be referenced or updated only by an indirect means called a message. In other words, an object is encapsulated and internal access to it is permitted only with a key called a message. An interface indicates what kind of key can make public the object. The functions provided by an object are available only through the interface. If only the interface is not changed, the object may be implemented in any way. In the object orientation, this characteristic enables the internal data structure or algorithm of an object easily changed without affecting other objects.

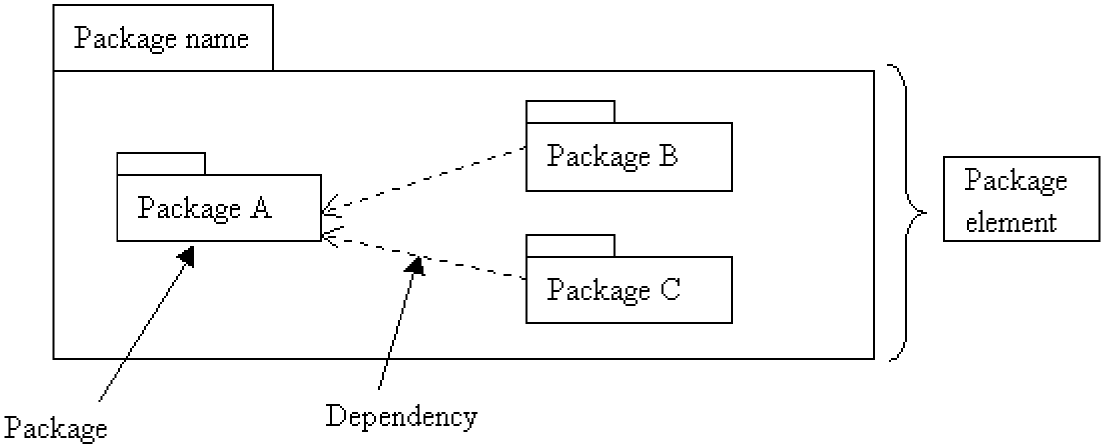
An interface only defines what is shown outside about implemented entities such as classes, and components, and does not have implementation. In other words, the operations specified by the interface must be provided classes supporting the interface. An interface may have a relationship of generalization. In this case, the interface class inherits operations from the high-level interface class.



To express an interface, the stereotype <<interface>> is added to the class icon. The attribute section is omitted because no attributes exist.

2.3.4 Packages

Model elements are grouped into a package. A package may have another package nested. Since each model element is owned by a single package, the package hierarchy has a tree structure.
A package serves as a base for configuration management, storage, and access control.



A package is expressed as a rectangle icon with another rectangle icon attached to the upper left like a tab (small rectangle).

A package name is indicated in a large rectangle. If the package contents are displayed in the large rectangle, the package name is enclosed in a tab.
Elements contained in the package are called package elements.
The visibility of each package element is indicated by adding one of the following symbols to the element name:

+: Public visibility
-: Private visibility
#: Protected visibility

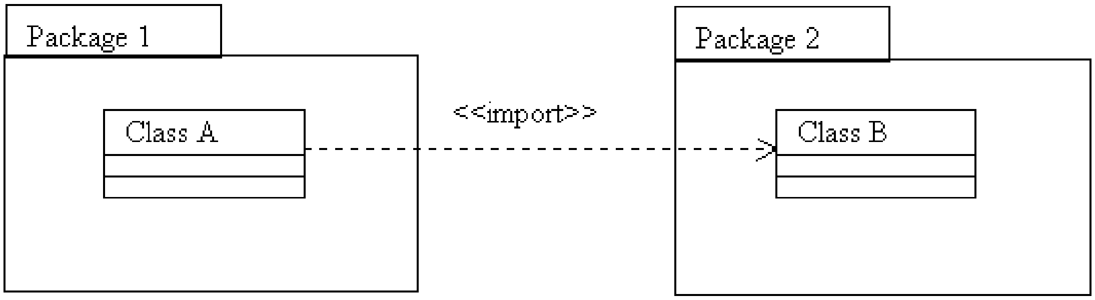
If a package element is a package, the visibility of an internal package element is determined from the visibility of the package and that of each element contained in the package (either visibility that is more limited).

A stereotype can be placed on a package name. The following are predefined in the UML:

<<facade>> Not holding any elements but referencing elements of another package
<<framework>> Mainly consists of patterns
<<system>> Representing all sets of models in the system
<<stub>> Providing the public sections of the package
<<topLevelPackage>> Representing the top-level package in the model

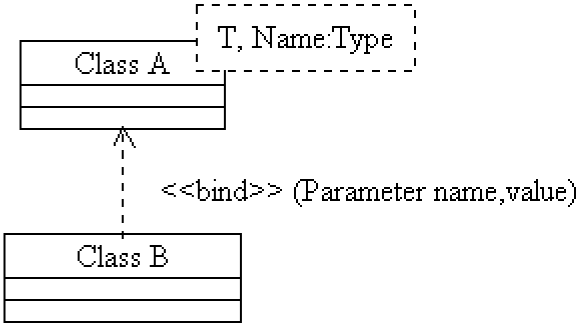
The tool function can be used to display only elements of a specified visibility level or to change the color or font for visibility.

For dependency between packages, a dotted arrow is drawn from the package containing the depending element (client) to that containing the dependent element (supplier). When referencing a class in a different package, the stereotype dependency <<import>> is drawn.



2.3.5 Template

Parameterized class is a concept used by C++ and other languages. This is a template class having one or more formal parameters that are not bound. An actual class is generated if specific values are given to the parameters for binding.



A parameterized class is expressed by overlaying a small dotted-line rectangle on the upper right corner of a class icon. A formal parameter listing of the class and its implementation type are described in the square. The formal parameters may also be expressed in the class.

The parameter syntax is as follows:

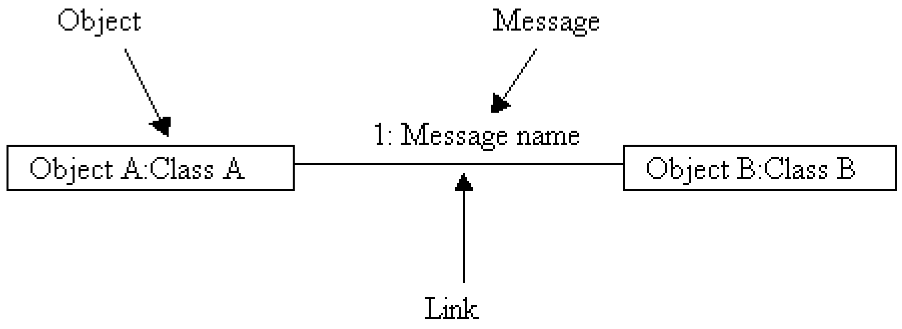
Name:Type
Name: The identifier of the parameter that is effective in the template
Type: Character string specifying a type to the parameter

For using a parameter as a reference, the stereotype <<bind>> is added to the arrow of dependency, and the actual parameter is written next in parentheses.

2.4 Behavior Model Description

2.4.1 Collaboration diagrams

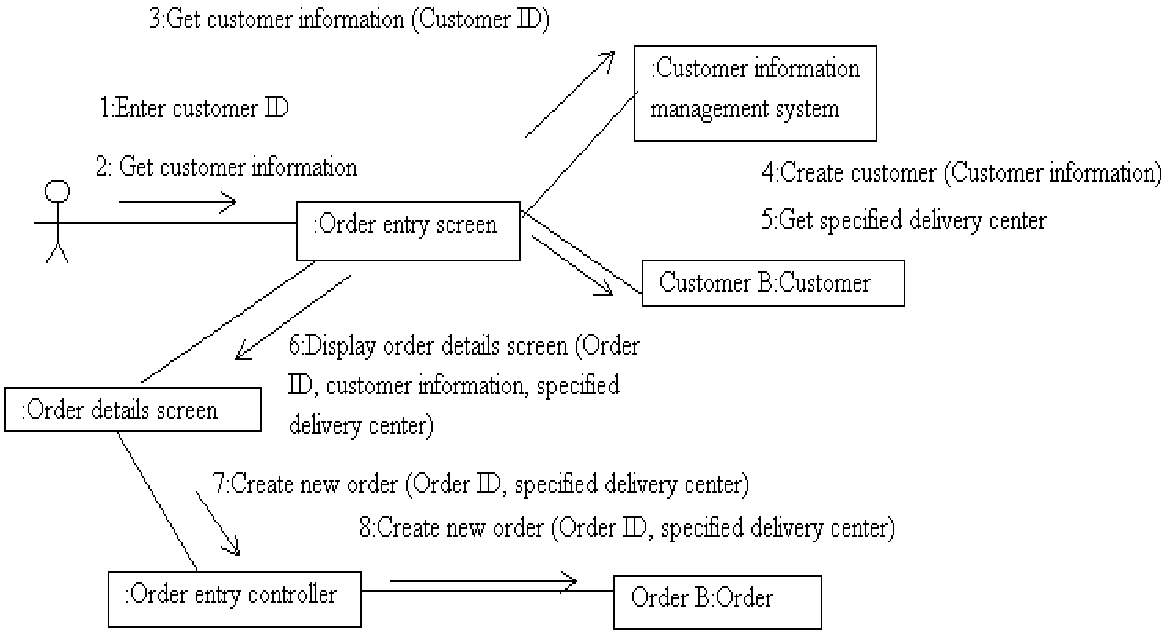
A collaboration diagram describes exchanges messages between the collaboration participating objects with connections between objects (collaboration contexts). This diagram clarifies a group of objects necessary for realizing a use case and their communication.



The participating objects are expressed as symbols with an underlined object name. The objects are coupled by a link, and the relationship of reference or connection is indicated for each object. An arrow is marked beside the link to represent a message flowing through the link, and a sequence number is assigned to the head of each message to indicate the message transmission order.

System behaviors can also be expressed in a sequence diagram. Compared with a sequence diagram, a collaboration diagram can express connections between objects in more detail for design, but may make tracing in order of time difficult.

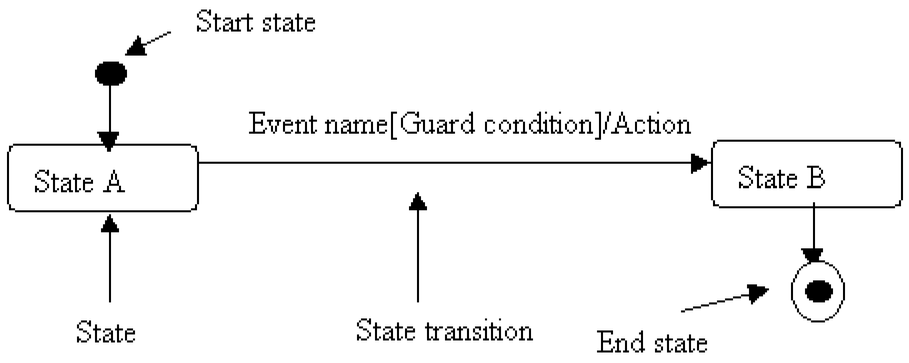
The collaboration diagram in the following indicates from when customer information is checked through the order details screen is displayed and an order class is created.


3:Get customer information (Customer ID)
:Customer information management system
1:Enter customer ID
2: Get customer information
4:Create customer (Customer information)
5:Get specified delivery center
:Order entry screen
Customer B:Customer
6:Display order details screen (Order ID, customer information, specified delivery center)
:Order details screen
7:Create new order (Order ID, specified delivery center)
8:Create new order (Order ID, specified delivery center)
:Order entry controller
Order B:Order

2.4.2 State diagrams

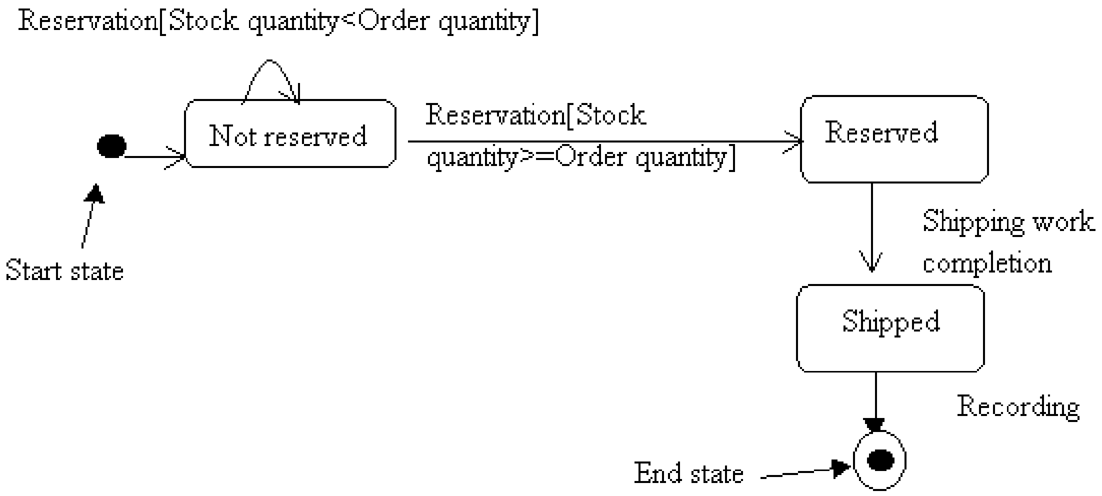
A state diagram describes object behaviors by a set of states which object may have, transitions between states and their trigger events or states, and actions accompanying transitions.

A collaboration or sequence diagram expresses an example of use case, but a state diagram expresses a general change pattern that an object belonging to each class may have.



A state is represented by a round box. State transition is indicated by a solid-line arrow between states. An event name is written beside the state transition. A guard condition is given when the transition is determined by whether or not the condition is satisfied. An action indicates state transition itself and operation accompanying the state itself. A state can also define activities that are executed consecutively at the starting state, at the ending state, and during the state.

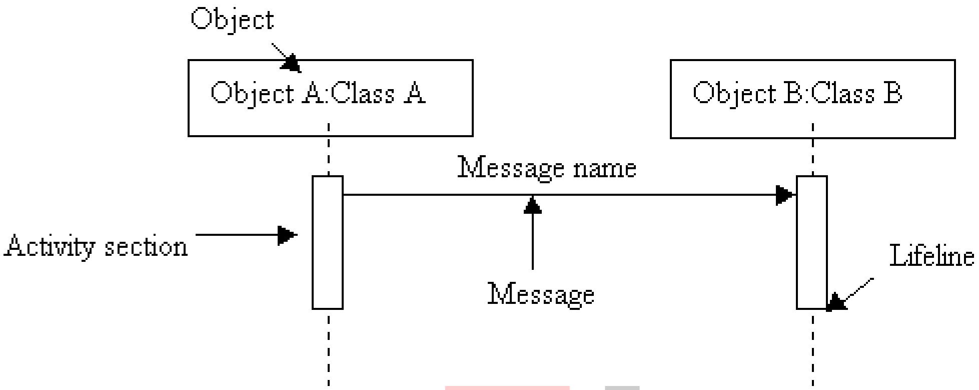
The following figure shows how to create an order object state diagram.


Reservation[Stock quantity<Order quantity]
Not reserved
Reservation[Stock quantity>=Order quantity]
Reserved
Start state
Shipping work completion
Shipped
Recording
End state

The first state after start is "Not reserved." Then the event "Reservation" occurs, but the state remains "Not reserved", because the reservation is not permitted, if the stock quantity is below the order quantity. The guard condition [Stock quantity<Order quantity] indicates this state. The event "Reservation" changes the state to "Reserved", if the stock quantity exceeds the order quantity. Then the event "Shipping work completion" changes the state to "Shipped" to finish the processing.

2.4.3 Sequence diagrams

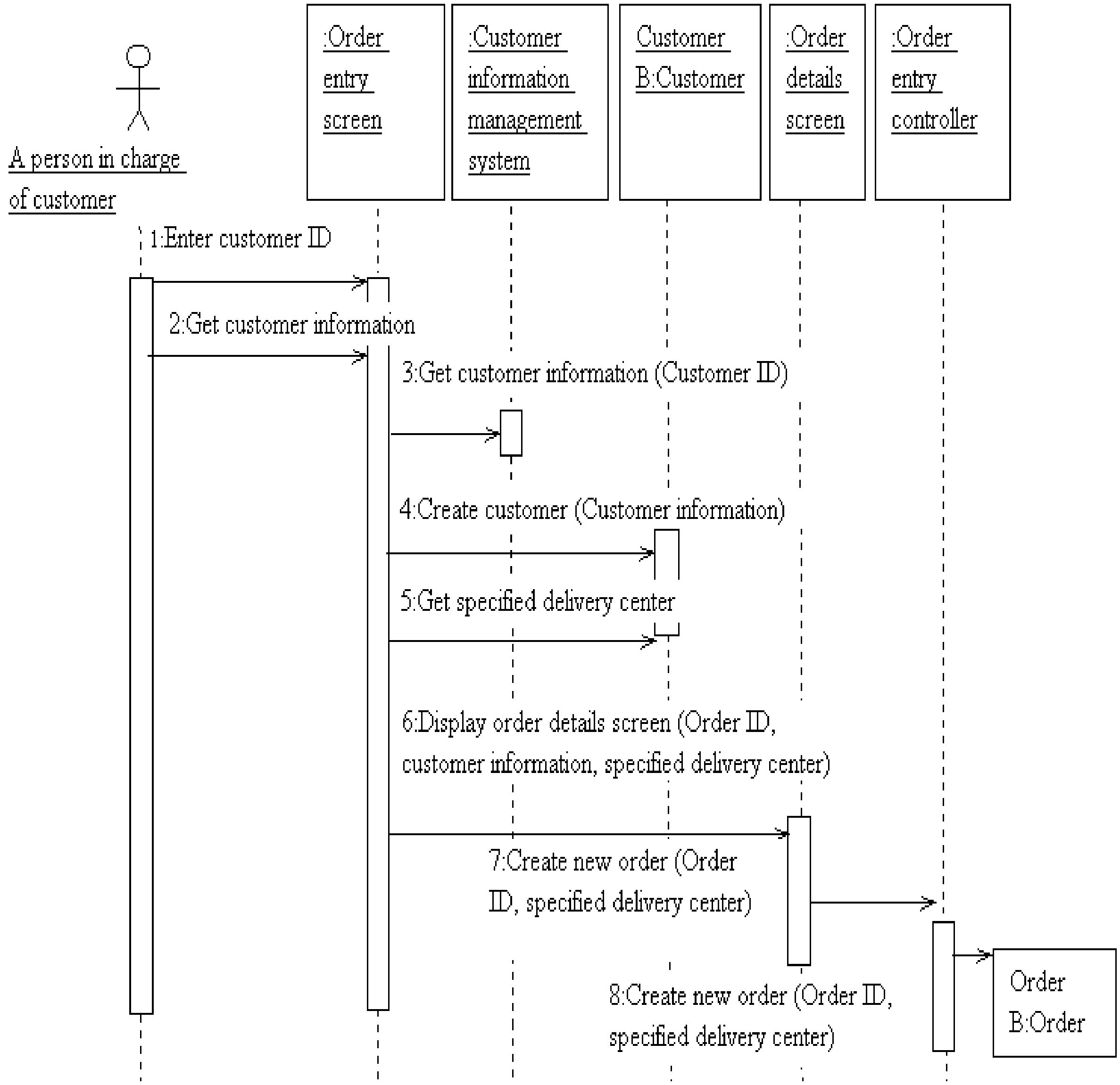
A sequence diagram is a kind of interaction diagram. This diagram indicates message transmission (interaction) events between collaboration participating objects in time order from the top. Since message transmission can be traced one at a time in time order, this diagram is useful for indicating the contents of collaboration specifically and is popular for analysis.



A sequence diagram is created on the basis of a use case diagram or scenario.

A vertical rectangle represents an object, and a solid-line arrow with a message name represents a message transmission event between objects. Message transmission events are listed sequentially from the top. Especially in analysis, each step of the scenario is described in text on the left side of a sequence diagram for message transmission. This makes it easy to set one scenario corresponding to a sequence diagram as an example of a use case.

The following use case diagram is from when customer information is checked through the order details screen is displayed and an order class is created.

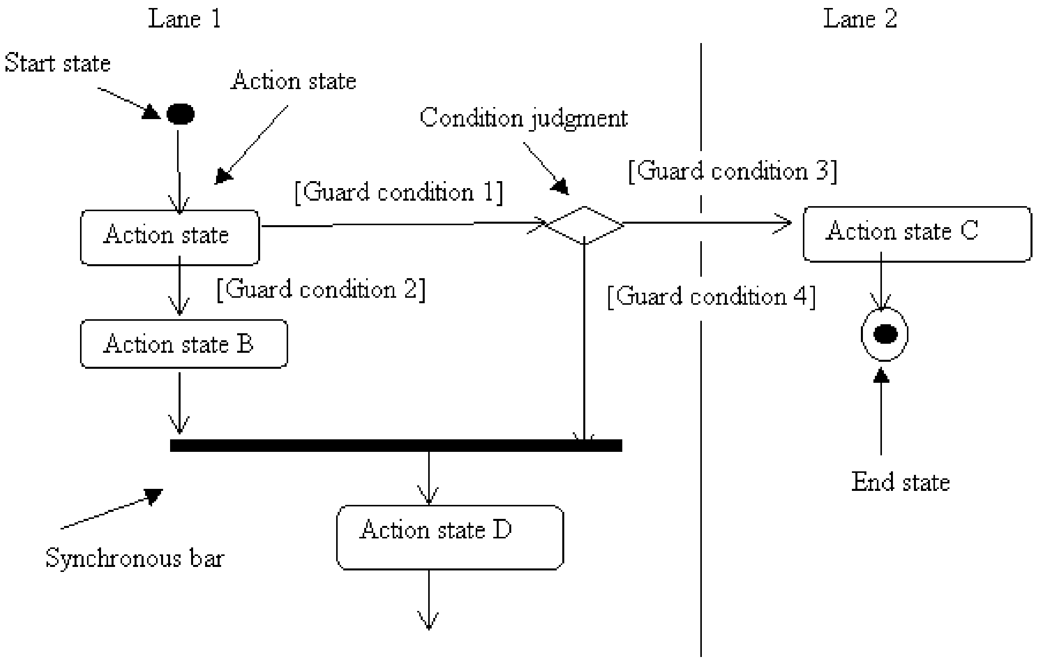

A person in charge of customer
:Order entry screen
:Customer information management system
Customer B:Customer
:Order details screen
:Order entry controller
1:Enter customer ID
2:Get customer information
3:Get customer information (Customer ID)
4:Create customer (Customer information)
5:Get specified delivery center
6:Display order details screen (Order ID, customer information, specified delivery center)
7:Create new order (Order ID, specified delivery center)
8:Create new order (Order ID, specified delivery center)
Order B:Order

2.4.4 Activity diagrams

An activity diagram expresses several related activities in time order as a group of jobs or a series of processing. An activity is a unit of work and is regarded as a kind of state. The fineness of activity is relative. An activity corresponds to a user, a task, system function (use case), or a method.

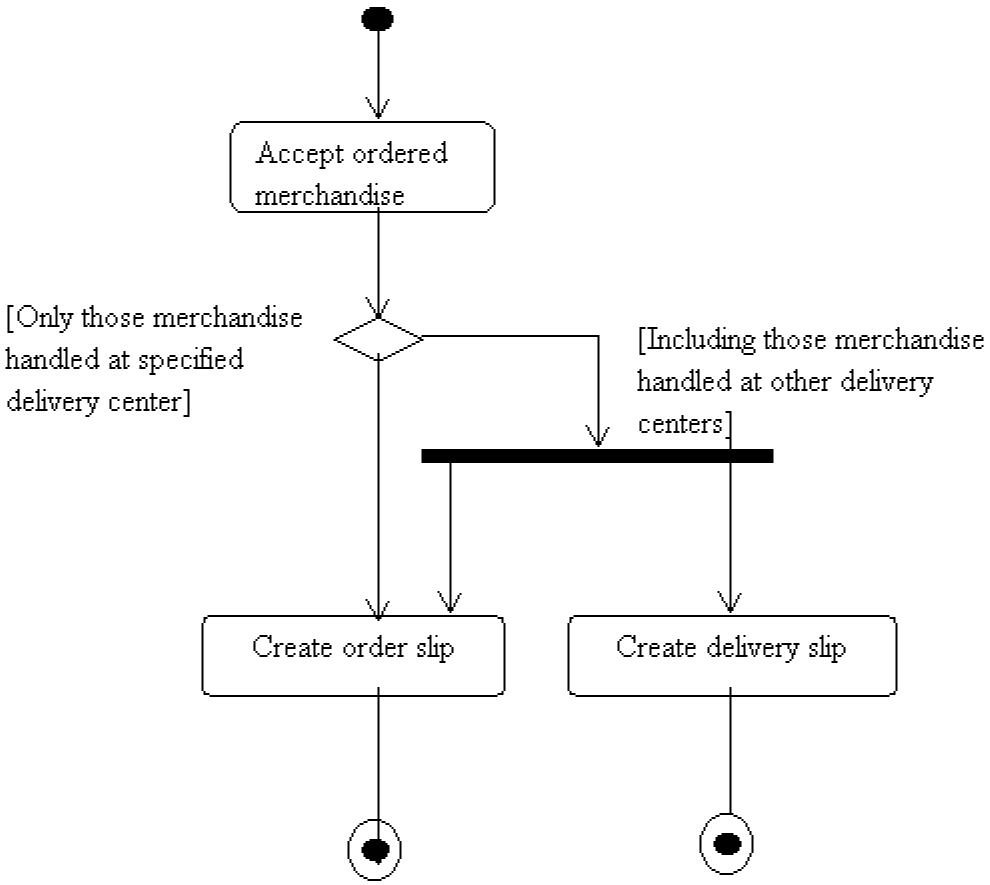
An activity diagram is used for the following descriptions:

- Business flow of specific work or organization
- Processing flow for one use case of the system
- Algorithm inside a method of an object



A round box represents an activity. An activity name is written in the round box. Activities are linked by arrows to indicate the work order. If transition to the next activity is permitted only after a certain condition is satisfied, a guard condition is added to the arrow. Condition judgment is regarded as a special activity and is expressed as a diamond. A bold bar represents the synchronization of all transitions input to the bar and also the parallelism of transitions if outputted from there. Lanes can be expressed to clarify a group of processing allocated to a specific object or subsystem.

The following activity diagram shows the event flow of a use case for order entry.


Accept ordered merchandise
[Only those merchandise handled at specified delivery center]
[Including those merchandise handled at other delivery centers]
Create order slip
Create delivery slip

Exercises

1. Read the explanation below and select an actor.

This order acceptance system also provides a function to accept orders through the Web. This means the system can accept orders in three ways, including by telephone and fax. When a telephone or fax order is received, the order receptionist enters the order information at the Web page screen. A customer may enter an order directly into the system through the Web page.

(1) Customer
(2) Order receptionist
(3) Web page
(4) Order
(5) Customer and order receptionist

2. Read the explanation below and select a use case.

When a purchase application is received, Yamada in the purchasing department of company O registers the application in the system, requests the vendor to send an order sheet by fax through the general affairs department, and makes a confirmation by telephone.

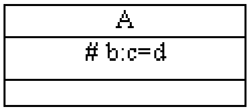
(1) Purchase system
(2) Purchase application
(3) Purchase application entry
(4) Order sheet transmission by fax
(5) Telephone confirmation

3. Select a characteristic of a use case diagram.
(1) Clarifies relationships between system provided functions and their users.
(2) Clarifies the class configuration of the system.
(3) Clarifies the data flow of the system.
(4) Clarifies the physical configuration of the system.
(5) Clarifies the status transition of the system.

4. Select an appropriate combination for drawing a relationship of generalization.

(1) Type class and implementation class
(2) Interface class and implementation class
(3) Superclass and subclass
(4) Whole set class and part class

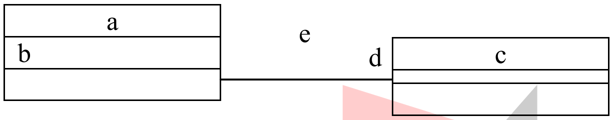
5. Select an appropriate explanation for the following figure:



(1) The type of attribute b is c.
(2) The default value of attribute b is c.
(3) The type of attribute c is b.
(4) The default value of attribute c is d.

6. Select an inappropriate explanation regarding the use of a class diagram.
(1) Shows classes and their static relationships.
(2) Shows class attributes and operations.
(3) Shows the physical file partitioning of a class.
(4) Shows the abstraction of a class.
(5) Shows the logical partitioning of a class.

7. Select an attribute(s) for the following figure:

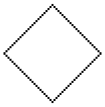


(1) a and c
(2) b
(3) d
(4) e

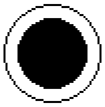
8. Select the most appropriate explanation for a collaboration diagram.
(1) Shows the lifecycle of a specific object.
(2) Shows the static structure of a class.
(3) Shows the collaboration of objects about a scenario.
(4) Shows a combination of components.

9. Select the starting state icon.

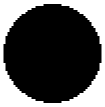
(1)



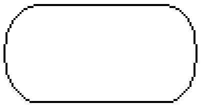
(2)



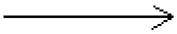
(3)



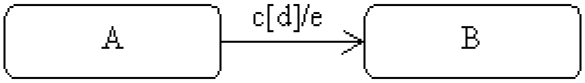
(4)



(5)



10. Select an appropriate combination of c, d, and e in the state chart below.

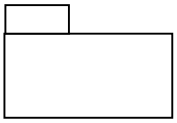


(1) c: Class d: Event e: Activity
(2) c: Guard condition d: Event e: Activity
(3) c: Event d: Guard condition e: Action
(4) c: Action d: Event e: Guard condition
(5) c: Pre-condition d: Guard condition e: Post-condition

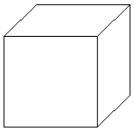
11. Select an item that does not become a component in a component diagram.
(1) Source code file
(2) Binary file
(3) Execution file
(4) Network node

12. Select a component icon.

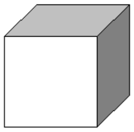
(1)



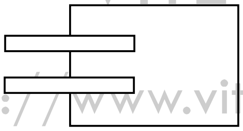
(2)



(3)



(4)



13. A developed program was modified. Select an appropriate diagram to see which file should be recompiled.
(1) Class diagram
(2) Sequence diagram
(3) Component diagram
(4) Deployment diagram
(5) Collaboration diagram

3 Object-Oriented Development Process

Chapter Objectives

This chapter explains how to proceed with the object-oriented software development process.

3.1 Characteristics of the Object-Oriented Development Process

The conventional development process is called waterfall development where processing advances to the next step after completing one step.

In the new development process called iterative development, however, a system is divided into small functional units and constructed by repeating the units. This is called iterative development.

(1) Small development unit

A system is divided into small functional units. A small development unit represents the development steps of each functional unit, from requirements definition to testing.

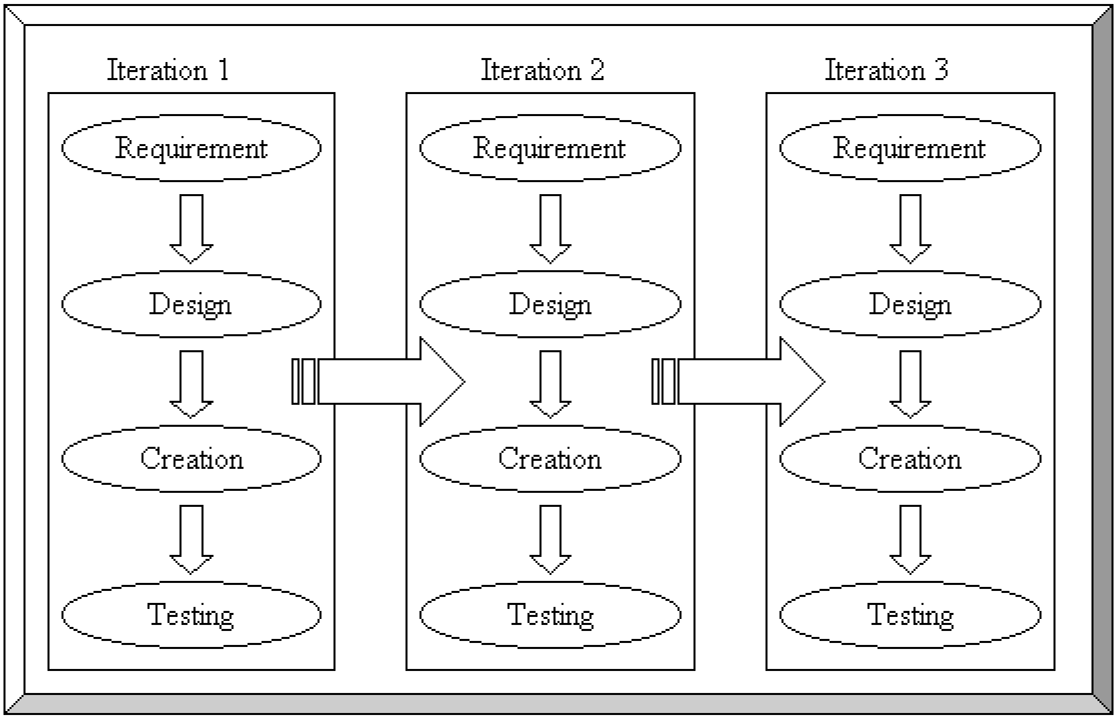
(2) Mini-project iteration

The contents of iterative development are formed as a mini-project of waterfall development, and a developer can experience a series of iterative development steps through a mini-project. This increases the chance of finding project-related problems in the initial stage.

(3) Architecture-centric

Architecture-centric means to verify and confirm important functions to determine an architecture at the initial stage of iterative development. This can reduce architecture-related risks in subsequent development, because the system is developed based on the determined architecture. This makes it possible to realize a stable architecture concept.

Here is an example of the iterative development process.



[Iterative development process]

3.1.1 Use case-driven development

(1) Use case

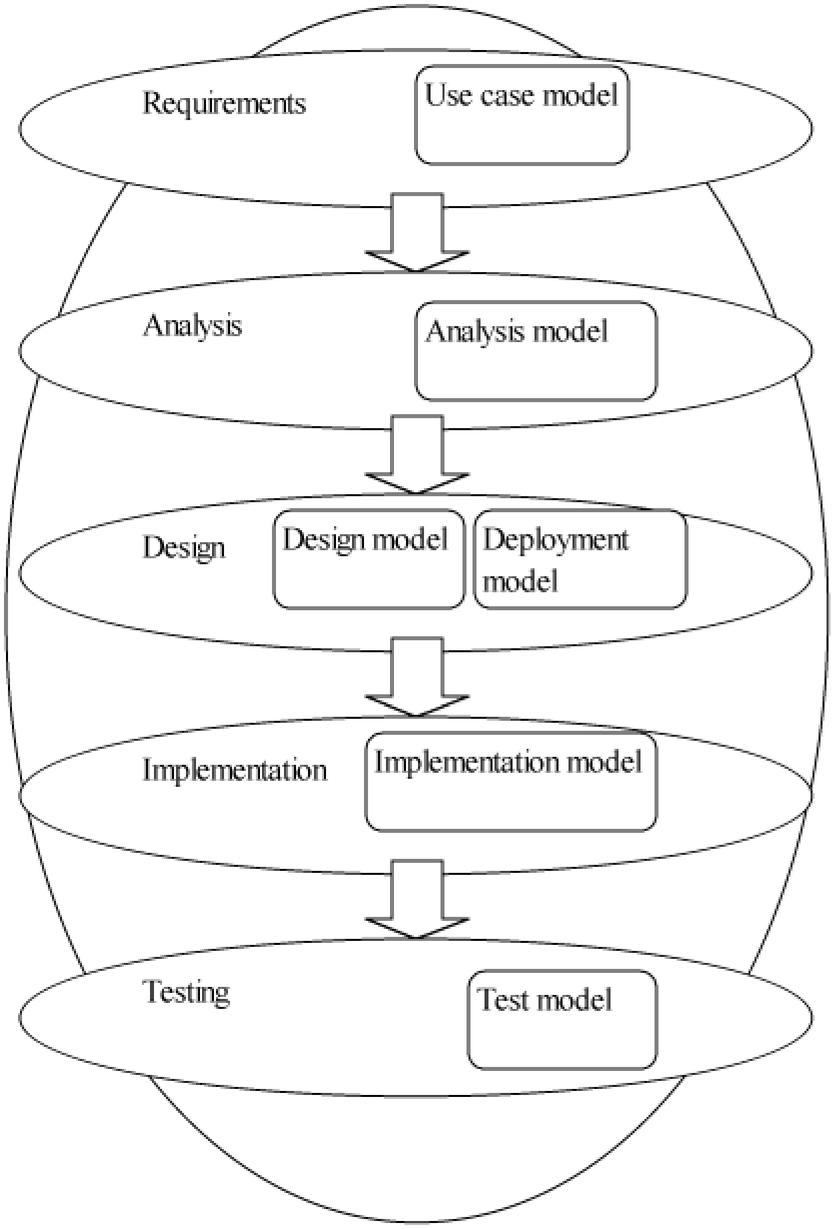
A use case is a series of operations to identify behaviors and functions from the viewpoint of the system user for system execution. Priority is given not only to useful system functions, but also functions for providing valuable results to the user.

(2) Use case-driven development

In use case-driven development, the development process involves gradually constructing a system by following workflows based on use cases.

Use case-driven development divides a use case into basic units for system-executed functions. Therefore, the analysis, design, implementation, testing, and user document creation should be conducted for each use case to ensure that the user requirements are properly satisfied.

The next figure shows use case-driven development.



Use case-driven development consists of a series of workflows from requirements definition to testing. The ellipses in the background indicate how a use case is linked to the work flows.

3.1.2 Architecture-centric

(1) What is a software architecture?

A software architecture specifically expresses the most important static and dynamic aspects of a target system examined from various angles.

An architecture clarifies the important characteristics of an entire design. Since this is greatly affected by many factors, such as the software execution platform, reusable components, existing system, extra functions, and reliability, the process to focus on the flexibility for changes, reusability, and ease of understanding during the process is important.

(2) Definition of architecture

An architecture can be outlined from what is not related to the use case. However, in the software a function corresponds to a use case and the shape its architecture, and these two mutually affect each other. Therefore, it is necessary to understand an outline of use cases.

Selected use cases should be individually defined in detail and clarified as components, classes, or subsystems.

Defining use cases in detail clarifies the details of the architecture and improves the precision of many use cases. The use case definition process should be continued until the architecture becomes stable.

(3) Significance of architecture-centric

In a software system that is large in scale and complex, its architecture is very important for system understanding and evolution, organization of development, and promotion of reuse.

1) Understanding system

For system development, all those concerned should understand the system. Keep in mind that a system has complicated behaviors and environmental and technical complexities.

Since a system is easy to change, changes may cause confusion. To avoid this confusion, an architecture-centric model should be developed.

2) Organizing development

A large software development organization increases the amount of communication among developers. A good architecture enables interfaces to be defined clearly for minimizing the amount of communication required. In other words, an interface is an efficient means of communication among developers.

With a stable interface, the software programs of two parties can run independently. An appropriate architecture will lead to an appropriate interface between subsystems.

3) Promotion of reuse

Reuse makes it impossible to find appropriate components for the problem domain and the architecture. Considering component connections satisfies system requirements and produces a use case model.

The role of an architecture is to define a reusable subsystem for use by a developer. A good architecture enables a developer to find reusable elements and components easily.

4) Promotion of system evolution

After implemented a system or even during development, a system should evolve according to changes in the surrounding environment. To allow evolution, a system must be easy to change.

A system should be flexible for changes and capable of gradual evolution.

3.1.3 Component-centric

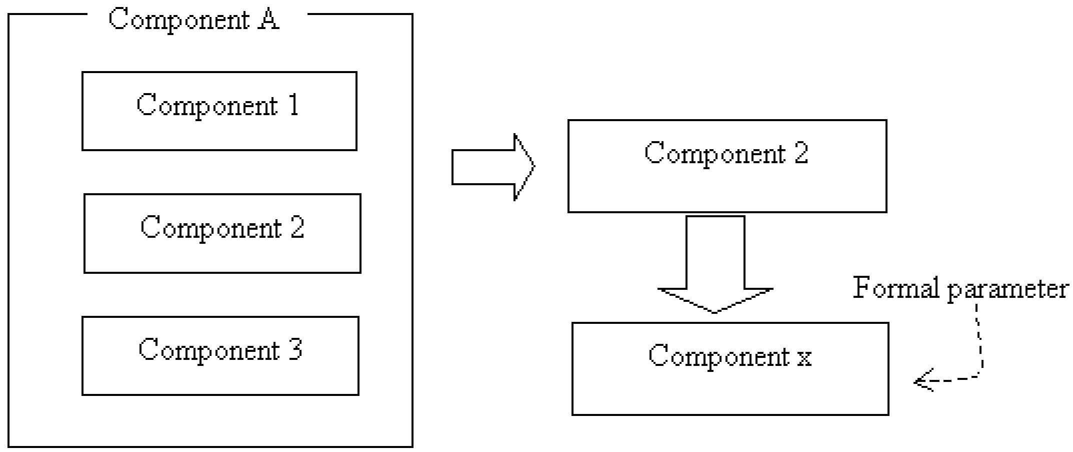
A component is a reusable class or class library developed using an object-oriented technology.

(1) Reusing components

Reusing components means that a component that can be reused is once disassembled and its reusable subcomponents are used to develop target software.

This has been used as macro parts and function libraries.

(2) Decomposing components



Component decomposition means to parameterize portions to be changed in a component to reuse the subcomponent.

The example decomposes component A and reuses component 2.

Component x has been made by parameterizing the portions to be changed in component x.

When reusing component x, specify changes as formal parameters.

3.1.4 Iterative and incremental development

Incremental development does not develop an entire system at a time, but develops use case-divided functions step-by-step. However, the progress of the entire project is based on an iterative workflow.

For an iterative workflow, a project is divided into four phases. Each phase is explained next.

(1) Inception phase

In the inception phase, the goals, scope, and direction of a project are studied to see whether the project may be started. The work period and contents differ greatly depending on the scale and characteristics of the project.

In this phase, use cases are listed and candidate architectures are created to study the contents, and judge the feasibility.

(2) Elaboration phase

The elaboration phase is the most important phase together with the construction phase. The construction phase is prepared for detailing use cases and establishing an architecture baseline. Important use cases selected from the listed ones are implemented to verify the architecture.

In this phase, the construction target and method are understood with some models to solve problems in the project. The following models are used:

- Domain model

For some kinds of problem domains, a domain model may not be created or created by other than the development team.

- Conceptual model

A conceptual model represents customer requirements. This model clarifies functional requirements, scenarios, users, quality requirements, performance requirements, and priorities.

At the end of this phase, about two thirds or more of the top level architecture should be clear.

- Specifications model

A specifications model represents architecture. This model clarifies main components, interfaces, and deployment.

At the end of this phase, it is desirable that all main use cases and about two thirds or more of the total use cases have been made clear.

It is important that these models should not be laboriously detailed, spending more time than necessary.

In the elaboration phase, processing is repeated several times to verify the architecture. To reduce iterations, skills in the programming language and development environment should be have been enhanced.

The iteration results may be or may not be used in the next construction phase.

The project speed can be measured by iterations in the elaboration phase to help estimation and planning in the next construction phase.

(3) Construction phase

In the construction phase, software is constructed on the basis of a model or plan created in the elaboartion phase. In a project, this phase takes the most time and effort.

A conceptual model, specifications model, or iteration plan are detailed and changed frequently as the work progresses. From the results, a new model called an implementation model is created in the construction phase. Necessary information is incorporated into the implementation model to realize the architecture specified in the specifications model.

Tests are also performed in this phase.

Programs created by iterations are integrated continuously. Module-level tests are

performed before integration, and system-level tests after integration. If codes that do not pass the tests are integrated, the project will become disordered. To prevent this, testcases must be created together with a test program.
Even after the program passes the test, the testcases can be used for later testing, if not discarded but stored together with the program.

- Detailing the conceptual and specifications models
The conceptual and specifications models created in the elaboration phase need to be sufficiently detailed by iterations to create an implementation model.
Requirements and restrictions are checked and determined in detail.
The things to be entered in the implementation model should not be entered in the conceptual or specifications model. The implementation model is created while the conceptual and specifications models are detailed or changed.

(4) Transition phase
The purposes of the transition phase are:
- To enhance the quality
- To enhance the performance
- To add optional functions
- To prepare a user document
The transition phase is terminated at the formal release to the user.
Lastly, a project document is created to prepare for project completion. The document should describe problems found in the project, main artifacts, necessary maintenance information, and measured metrics (e.g. lines of code, the number of operators, and the number of loops) records.

Exercises

1. Which is not one of the three characteristics of the object-oriented software process?
(1) Architecture-centeric
(2) Activity modeling
(3) Iterative and incremental development
(4) Use case-driven development

2. Which is an appropriate explanation of use case-driven development?
(1) The development process is carried out along a certain workflow.
(2) Use-case-driven development is affected by the software execution platform, reusable components, and existing system.
(3) The system is divided into subsystems or other units for phased development.
(4) The most important static and dynamic aspects of the system are expressed specifically.

3. Which is not correct about the significance of architecture?
(1) All those concerned should understand the development system.
(2) A larger software development organization increases the communication work between developers. Therefore, an interface should be clarified to minimize the effort.
(3) Promoting reuse makes it impossible to identify problem areas and appropriate components for setting up an architecture.
(4) The system should evolve according to environmental changes even in the middle of development or during operations.

4. The following descriptions are about iterative and incremental phases. Select the corresponding phases from the answers below.
(1) The construction phase is prepared for detailing use cases and establishing an architecture baseline.
(2) The purposes of this phase are to enhance the quality, to enhance the performance, to add optional functions, and to prepare a user document.
(3) The goals, scope, and direction of a project are studied to see whether the project may be started.
(4) Software is created on the basis of a model or plan created in the elaboration phase.

Answers
a. Inception phase
b. Elaboration phase
c. Construction phase
d. Transition phase

4 Analysis

Chapter Objectives

This chapter explains the analysis techniques based on object orientation, and how the modeling language UML (unified modeling language) is used in the analysis process.

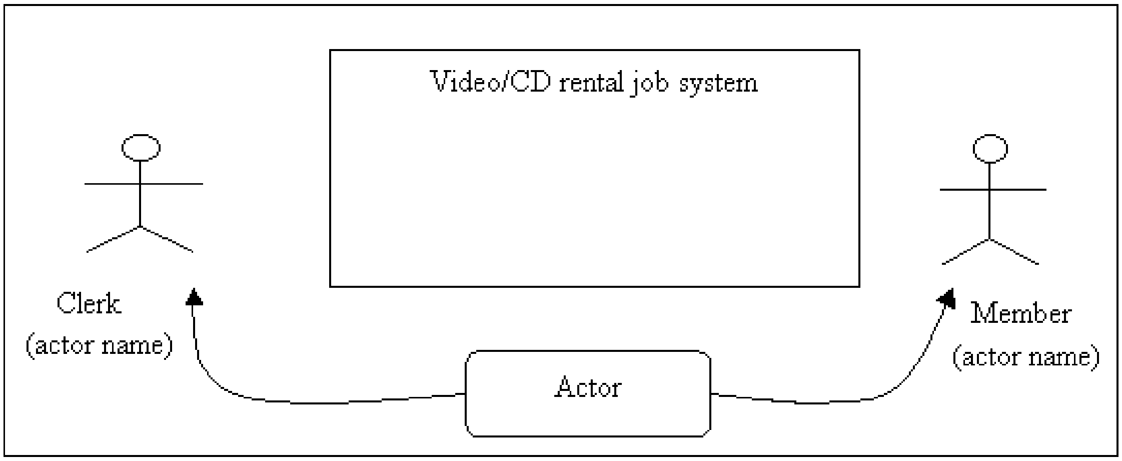
4.1 Requirements Analysis

In the requirements analysis, the system requirements of the user are clarified and the positioning of the system to be developed in the user business is decided. The important thing at this point is the orientation and elaboration for the system to be developed. The requirements analysis is conducted iteratively while the user and developer conduct a review. This section explains the requirement analysis method, using the use case as an example.

4.1.1 Actors

The actor refers to the user who actually uses the system and to other systems and hardware that are related to the system. If the actor is a person, it represents the abstracted role of the person, instead of the user himself or herself. This enables the same person to play multiple actor roles.

The following figure shows an example of an actor:



[What is an actor?]

4.1.2 Use case

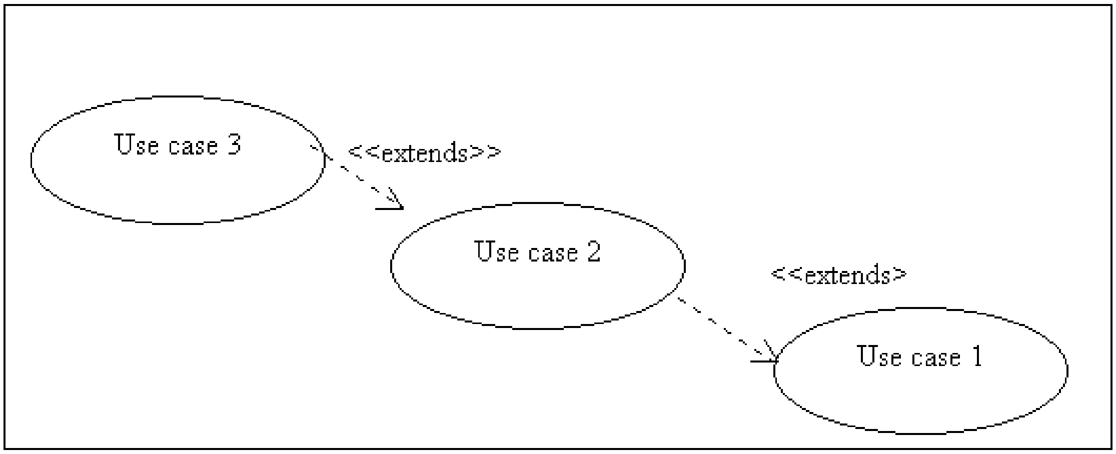
The use case refers to what is obtained by modeling the relationships between the actor and the system and by abstracting a job of some unit performed by the user utilizing the system. The use case is a function in the system started by an actor and is described by a simple sentence such as "Lend a video". Features of the use case are that it is started by an actor and is a complete function for each job of some unit performed by the actor. There are the following types of relationships between use cases: the extension relationship, use relationship, and grouping.

These relationships are explained below.

(1) Extension relationship

The extension relationship refers to the inheritance relationship between use cases. This means that a use case extends another use case.

The following figure shows an example of an extension relationship.



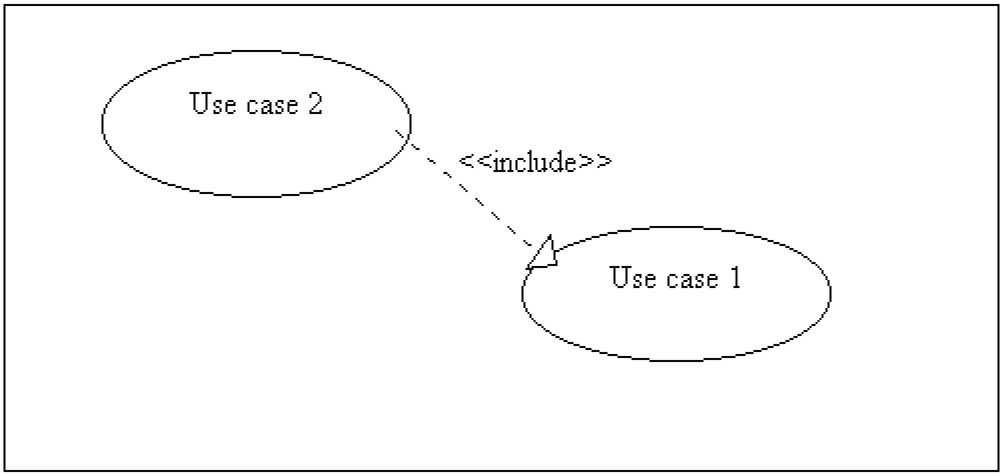
[Extension relationship between use cases]

The figure shows that the use case 2 is extended by the use case 1. It also shows that the use case 3 is extended by the use case 2. "<<extends>>" is used as the stereotype to be attached to the arrows.

(2) Use relationship

The use relationship refers, like the extension relationship, to the inheritance relationship between use cases. The difference from the extension relationship is that all functions available in use cases must be fully used in the use relationship. However, this does not necessarily mean that functions in the using use case must be used in accordance with its sequence. It means that functions of the using use case and those of the used use case can be mixed.

The following figure shows an example of a use relationship.



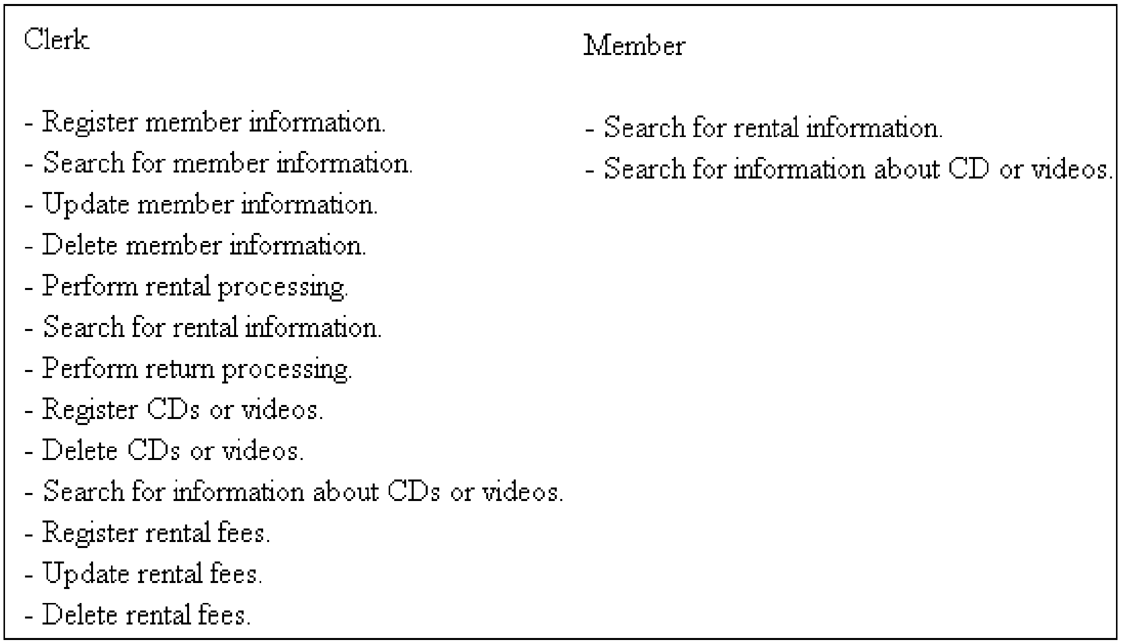
[Use relationship between use cases]

The figure shows that the use case 2 uses the use case 1. Like the extension relationship, an arrow is drawn from a using use case to a used use case, and a white triangle is marked on the used use case side. The stereotype to be used here is "<<include>>".

(3) Grouping

Grouping means to group functions shown in the use case diagram by their similarity or their close relationship.

The following describes processes that can be executed by actors using sentences as use cases:



Clerk

- Register member information.
- Search for member information.
- Update member information.
- Delete member information.
- Perform rental processing.
- Search for rental information.
- Perform return processing.
- Register CDs or videos.
- Delete CDs or videos.
- Search for information about CDs or videos.
- Register rental fees.
- Update rental fees.
- Delete rental fees.

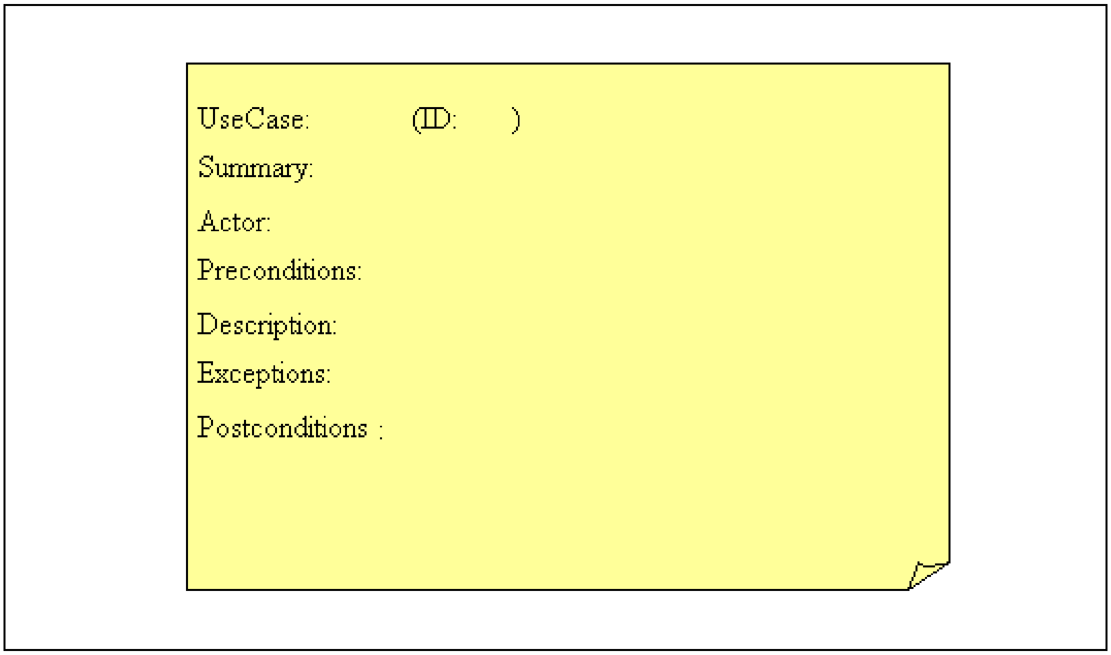
Member

- Search for rental information.
- Search for information about CD or videos.

[Examples of the use case in the video/ CD rental job system]

The use case description describes the interaction between the use cases and actors. In UML, nothing is specified that actually shows exchanges assumed between the outside of the system and use cases.

The general format for the use case description is shown in the following.



UseCase: (ID:)

Summary:

Actor:

Preconditions:

Description:

Exceptions:

Postconditions :

[General format for the use case

The following explains the contents to be described in the figure:

(1) UseCase
Describe the usecase name.

(2) Summary
Describe the role to be played by the use case.

(3) Actor
Describe what the actor is for the use case.

(4) Preconditions
Describe preconditions for starting the use case.

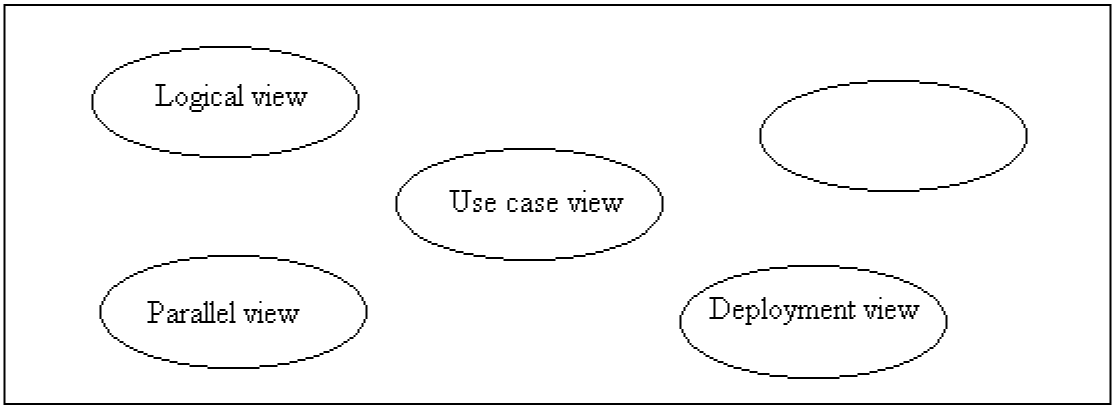
(5) Description
Describe normal process performed by the use case.

(6) Exceptions
Describe exception process in the use case.

(7) Postconditions
Describe the resulting state after executing the use case.

4.1.3 Use case model view

The use case view describes requirements of the actor to the system and the functions provided by the system to the actor. This view is the central view in system development. Contents of the use case view are reflected in other views, which are used to describe other aspects. Thus, if the use case view is changed, other views are also affected. The following explains the use case view and four other views, with a figure showing their relationship.



[Relationship between the use case view and other views]

(1) Logical view
The logical view describes the internal mechanism of the system of how to realize the functions extracted in the use case. This view is mainly used by the designer and developer.

(2) Component view
The component view describes the dependency between software components. The following types of software components are available:

1) Program source code
2) Binary code obtained by compiling source code
3) Executable program obtained by linking all binary code required for execution

The component view is mainly used by the developer.

(3) Parallel view
The parallel view describes synchronization and asynchronization in parallel processing of processes and threads. The purposes of the parallel view are as follows:

1) Efficient use of resources such as CPU
2) Handling synchronization of communication between processes or threads executed in parallel

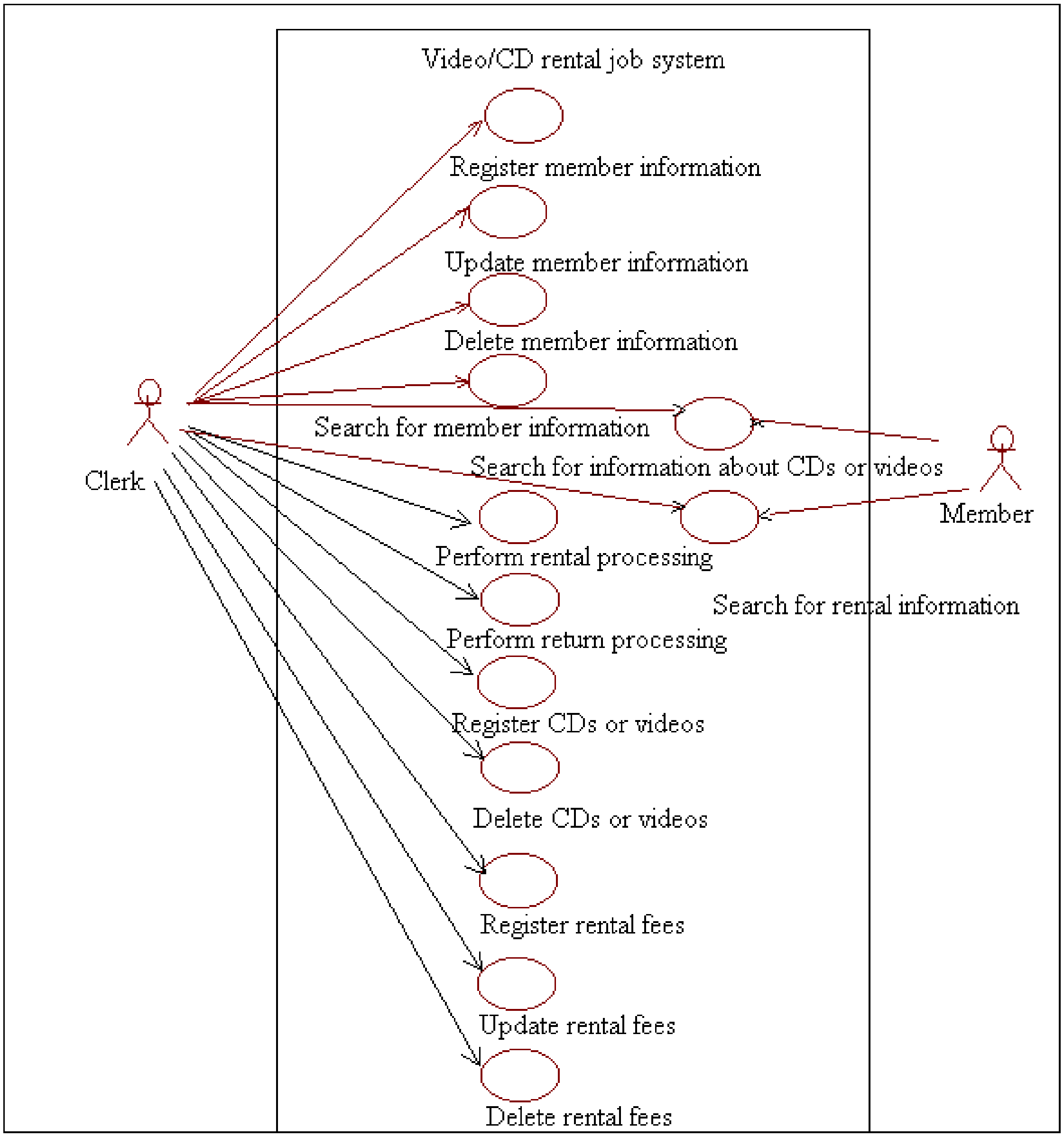
The parallel view is mainly used by the designer.

(4) Deployment view

The deployment view describes the physical layout of computers and devices and their relationships. The deployment view is mainly used by the designer.

As explained, the use case describes the actor requirements and the functions provided by the system. As an actual example of use case, the following shows a use case diagram:

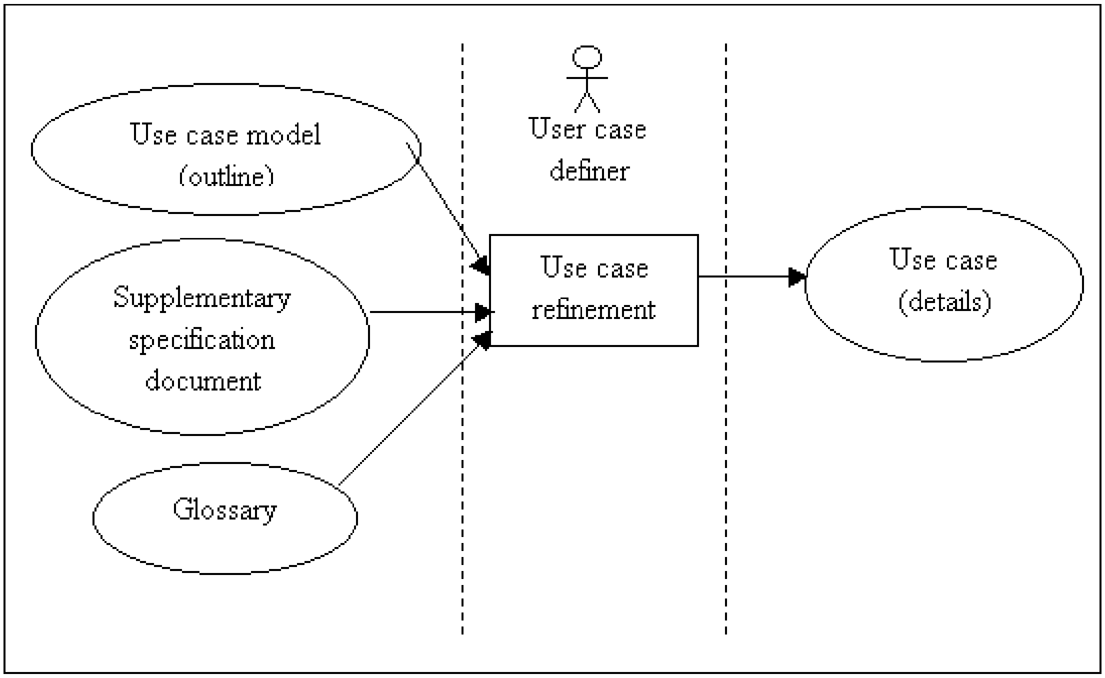
The following figure shows an "example of a use case in the video/CD rental business system".



Video/CD rental job system
Register member information
Update member information
Delete member information
Search for member information
Search for information about CDs or videos
Clerk
Member
Perform rental processing
Search for rental information
Perform return processing
Register CDs or videos
Delete CDs or videos
Register rental fees
Update rental fees
Delete rental fees

[Use case diagram]

The refinement of the use case is described next. The purpose of refining the use case is to describe in detail how to start and end the use case and the event flow including interaction between the system and actor.



[Use case refinement]

In the figure, it is important for the use case definer to proceed with work while keeping direct contact with the use case users. Specifically, perform the following:

(1) Interview with the use case users.

(2) Summarize the use case users'opinions about the use case of.

(3) Request the use case users to recheck the use case definition.

4.2 Domain Analysis

Domain analysis is used to verify the feasibility of the system again based on the results of requirements analysis by the use case. It is important to conduct the verification from the viewpoint of the system. This section explains not only the domain object model creation and input-output model design, but also the class diagram, interaction diagram, event flow diagram, and package diagram used for domain analysis.

4.2.1 Domain object model

A domain object refers to an object that plays an important role in the target domain. A domain object model can be created by the following procedure:

(1) Collect and build a glossary.
A glossary is a collection of terms used for system development. Since a term often directly leads to an object name or operation name in object-oriented development, the glossary becomes more important as the system becomes bigger.

In the domain analysis process, domain specific terms and other main terms are defined. It is important at this point to accurately define terms that tend to be vague. Typical examples are "user" and "staff". Terms can be defined, using the "staff" of a library as an example, as follows:

- Staff
A staff person with a librarian qualification who is authorized to lend books

(2) Find objects in the domain and their attributes.
First, find domain objects according to the following procedures:

1) Find the name, responsibility, and role of each object.
Search the scenario that describes the use case and the behaviors of the system for domain objects. Since domain objects play dominant roles in the domain, they must be clearly distinguishable and should exist for a long time in the life cycle of the system. Then, find the responsibility and role for each domain object candidate.

The following table lists them, taking the library as an example:

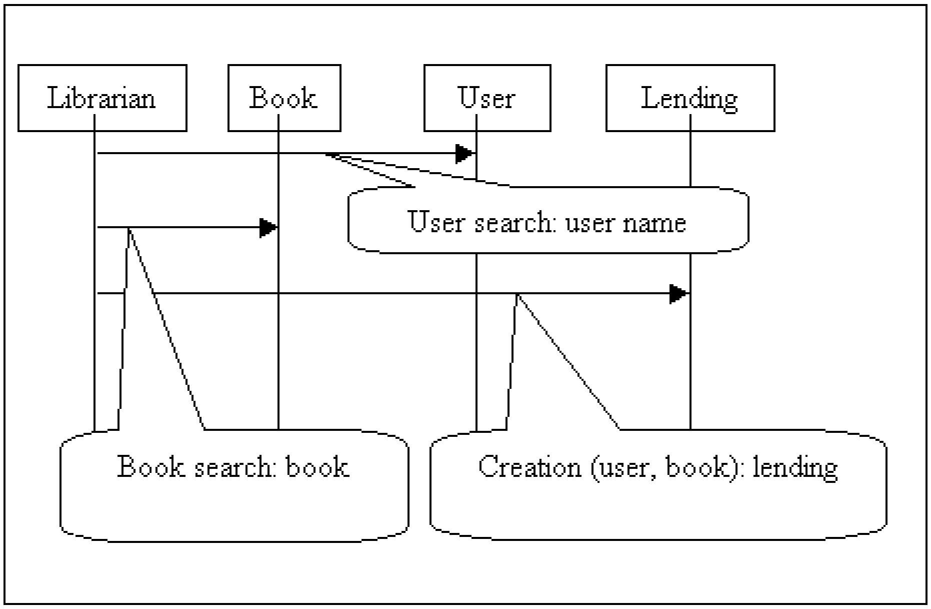
[Domain object candidates and their responsibilities and roles]

Domain object candidate	Responsibility and role
User	Entity to borrow books
Librarian	Entity to handle lending and returning books
Lending	Entity to process and manage book lending

2) Classify objects and find relationships and inclusions

Classify domain objects found in 1) and then find relationships and inclusions among domain objects. Create a sequence diagram to decide which messages to exchange in which sequence. The sequence diagram shows the sequence of message flows among objects.

An example of a sequence diagram is shown in the following, taking book lending in the library as an example.



[Example of the sequence diagram in book lending]

The arrows in the figure represent messages and their format is as follows:

Message name (parameters): return value

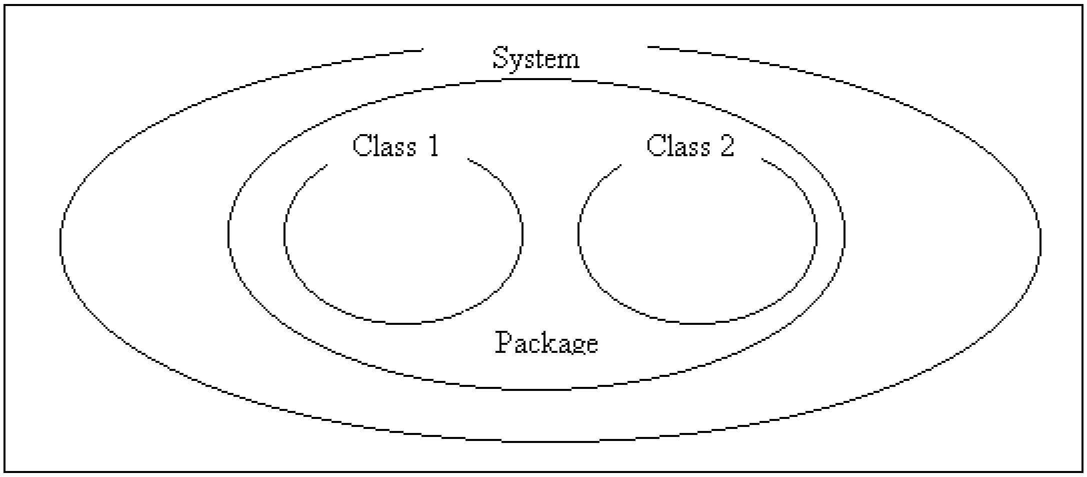
It is clear from the figure how the librarian searches for the user and books, and then lends the books to the user.

(3) Extract classes contained in the domain model.

The domain model represents the domain in which the system to be created will be used. Namely, extract classes that exist in the domain.

(4) Divide the system into packages.

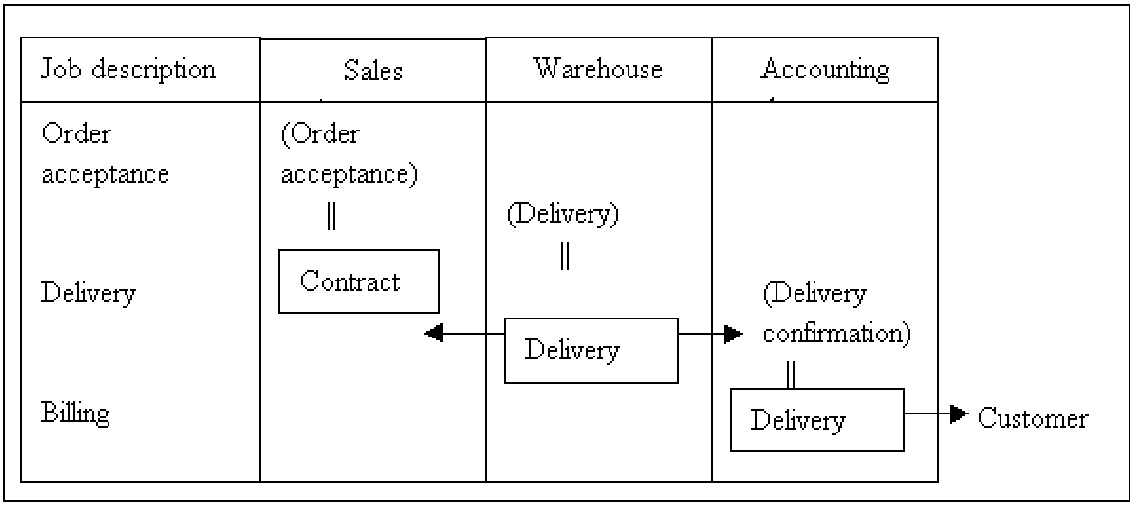
A package is a unit of grouping based on the class role or meaning. Classifying classes into groups improves maintainability of the model. The concept of package is shown in the following.



[Concept of package]

(5) Conduct the workflow analysis.

The workflow analysis is a method to analyze functions that make up a job by making a schematic representation of the job flow. This enables problems, such as unnecessary work in the job, to be detected. There is no specific way of drawing diagrams for the workflow analysis. The following shows an example of a diagram for workflow analysis.



[Example of a workflow diagram]

4.2.2 Input-output model design

The input-output model design means designing view objects. Examine the required items closely before designing the actual screens and forms that are directly associated with the user. Clarify which items are to be displayed or printed. View objects are obtained by organizing such items for each input-output operation.

(1) Purpose of introducing the view object
Extract view objects from the use case

Grasp rough input-output data for view objects, instead of concrete input/output data.

System development using GUI has become commonplace in recent years, and the GUI creation function of programming languages has been extended as greater importance is placed on the user interface. By using such excellent programming languages or dedicated software, the user interface can now be designed and implemented relatively easily. As a result, it is now possible to develop the logic part and to design and develop the user interface, such as the screen, separately by assigning separate development teams.

When the user interface is developed separately, the substances of input-output data are hidden, which increases the independence of the logic part and the user interface. This reduces the effects of changes to the logic part on the user interface.

This is the purpose of introducing view objects.

(2) Correspondence with the domain object attributes
View objects have input-output data extracted in the use case description work as their attributes. The substances of the view object attributes become domain object attributes. Use the view object attributes in the user interface design, and use the domain object attributes, which are substances, in the logic design.

Thus, the designer must recognize the correspondence between view object attributes and domain object attributes.

The following shows an example of a table of correspondences between view objects and domain objects.

[A table of correspondences between view objects and domain objects]

View object name	Video/CD lending status view	
View object attributes	Domain object attributes	Direction
Member_No.	member_information.member_No	Storage
Commodity_No.	commodity_information.commodity_No	Reference
Date	lending_detail.dending_date	Reference
	lending_detail.expected_return_date	Reference
	return_detail.return_date	Storage
Fee	commodity_information.rental_fee	Reference
	lending_detail.payment	Reference
	return_detail.late_fee	Storage

In the corresponding table, the view object names and their attributes are described. Also describe domain object attributes, which are substances, to clarify the correspondences.

For example, the "date" attribute of "video/CD lending status view" indicates that the "lending date" attribute of "lending detail" and "return date" attribute of "return_detail" are substances. The storage indicates that a view object attribute is stored as a domain object attribute. The reference indicates that a domain object attribute is reflected in a view object attribute.

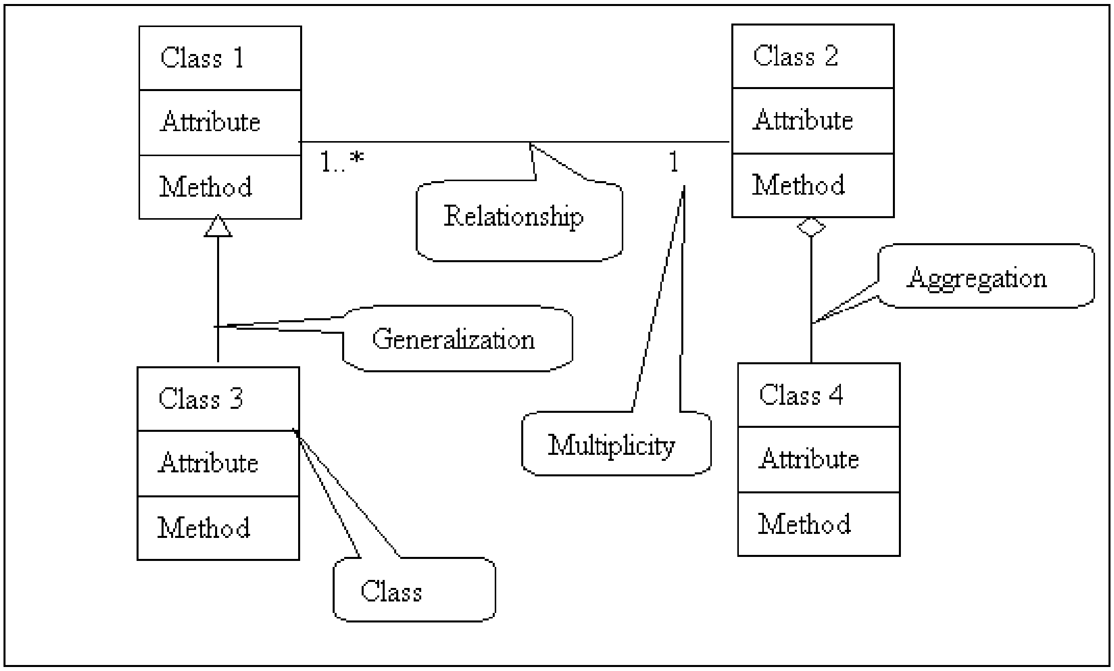
Even if no view object is introduced, it is necessary to grasp to which domain object the input-output data extracted in the use case description corresponds.

4.2.3 Class diagrams, interaction diagrams, event flow diagrams, and package diagrams

(1) Class diagrams

The class diagram describes the static structure of classes existing in a system. The static structure refers to the relationships between the classes and the attributes and operations held by the classes. The class diagram is used to view the problem domain logically and statically.

The following figure shows an example of a class diagram.



[Example of a class diagram]

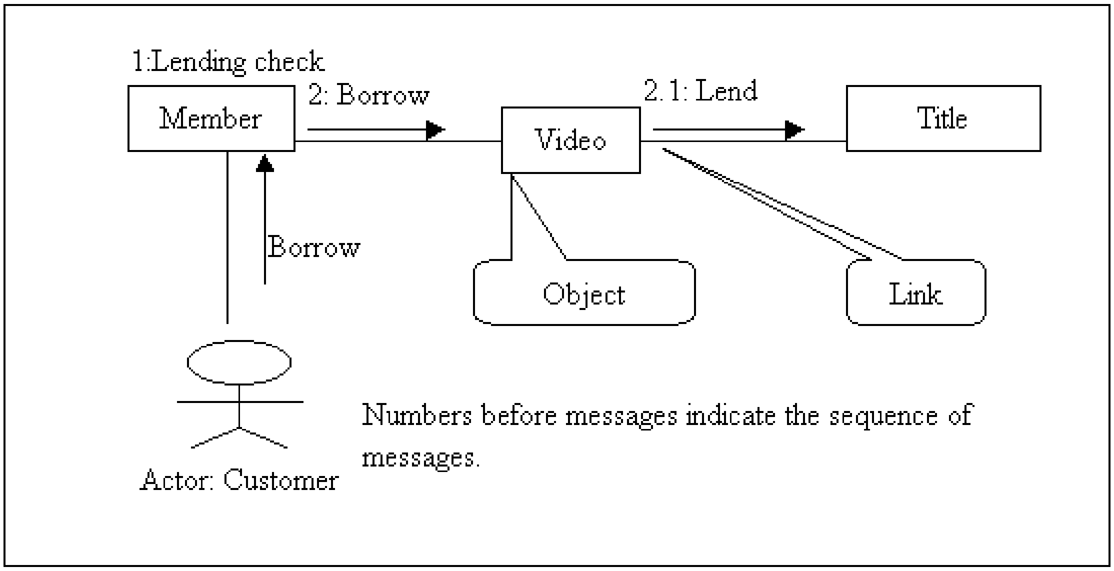
(2) Interaction diagrams

The interaction diagram represents behaviors of a related group of objects. It also shows how a use case is implemented by the message flow and system. An interaction diagram is a collaboration diagram or a sequence diagram.

The following explains each of these.

1) Collaboration diagrams
The collaboration diagram represents the relationships among interacting objects and focuses on the layout.

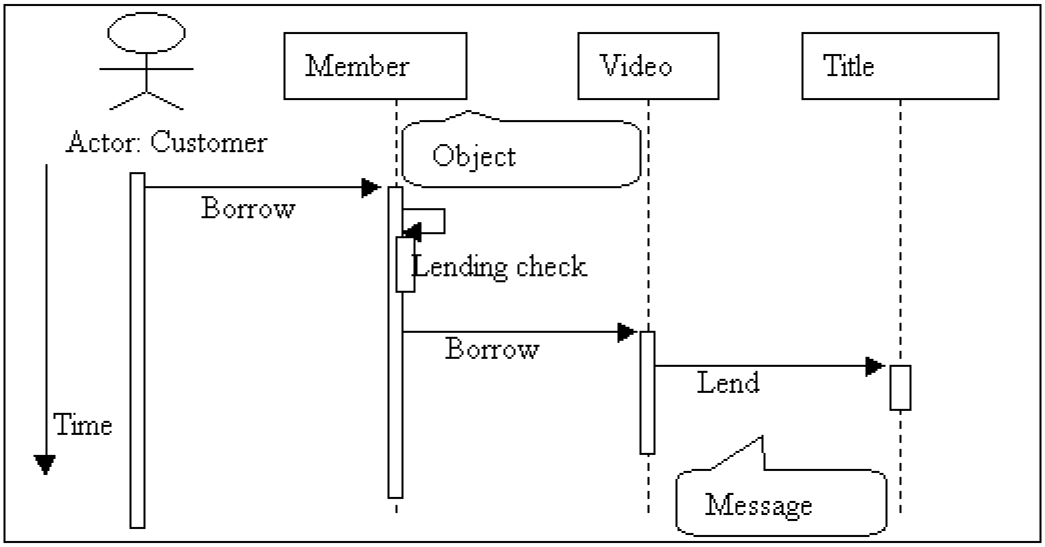
The following figure is an example of a collaboration diagram:



[Example of a collaboration diagram]

2) Sequence diagrams
The sequence diagram represents message sending and receiving among objects in time. By using the sequence diagram, the flow of messages can be grasped intuitively.

The following figure is an example of a sequence diagram.



[Example of a sequence diagram]

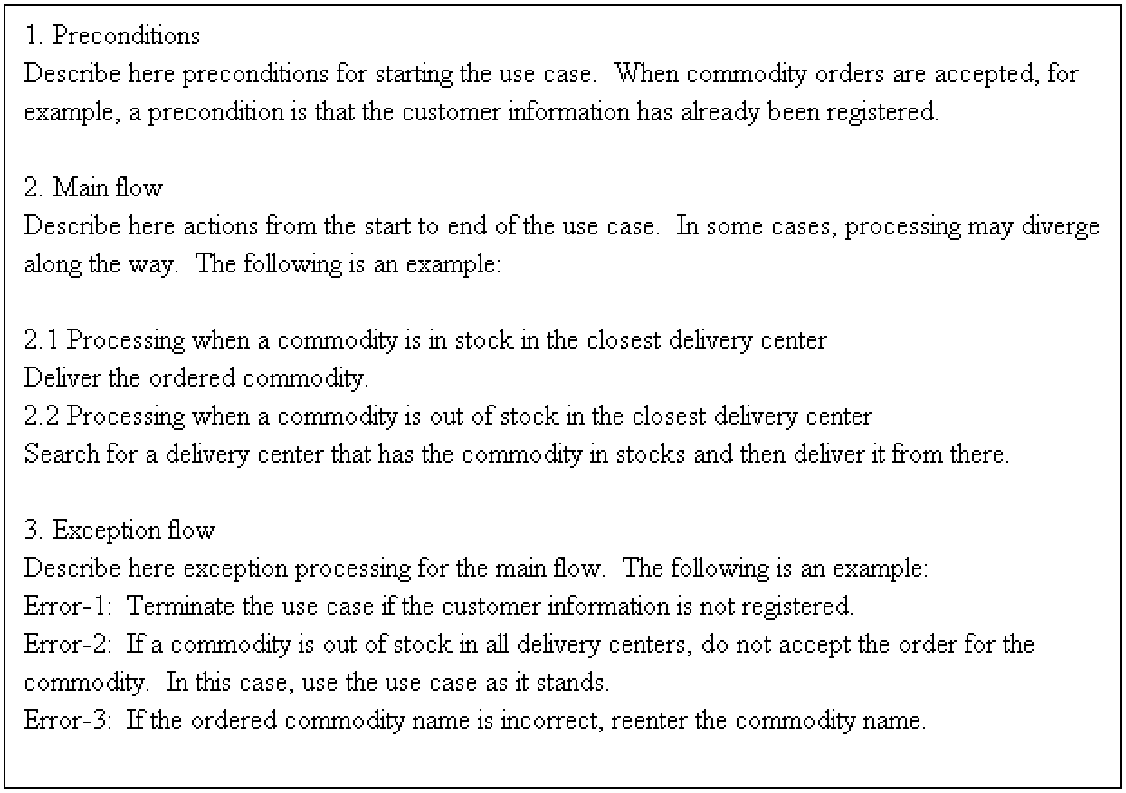
(3) Event flow diagrams

The event flow diagram describes a sequence of actions for a use case. That is, the event flow represents not only what the system does when some use case is executed, but also the interactions between the system and actors. This is represented in writing. The important points are as follows:

1) Avoid overly-detailed description of the user interface.

2) Describe exception processing as an exception flow.

The following shows an example of an event flow diagram.



1. Preconditions

Describe here preconditions for starting the use case. When commodity orders are accepted, for example, a precondition is that the customer information has already been registered.

2. Main flow

Describe here actions from the start to end of the use case. In some cases, processing may diverge along the way. The following is an example:

2.1 Processing when a commodity is in stock in the closest delivery center
Deliver the ordered commodity.
2.2 Processing when a commodity is out of stock in the closest delivery center
Search for a delivery center that has the commodity in stocks and then deliver it from there.

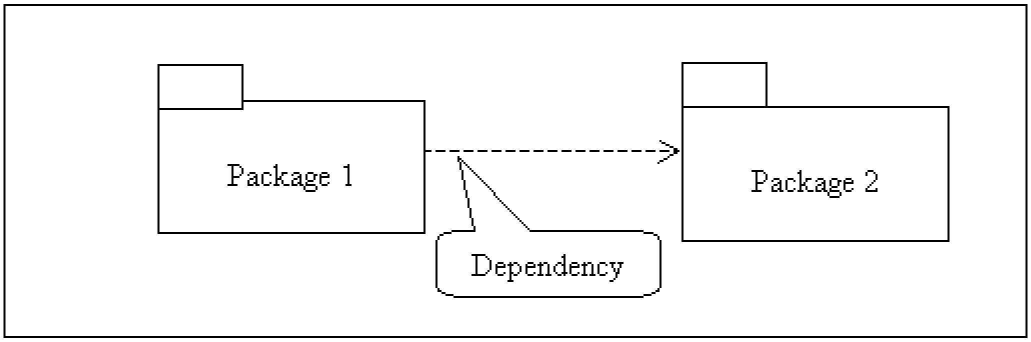
3. Exception flow

Describe here exception processing for the main flow. The following is an example:
Error-1: Terminate the use case if the customer information is not registered.
Error-2: If a commodity is out of stock in all delivery centers, do not accept the order for the commodity. In this case, use the use case as it stands.
Error-3: If the ordered commodity name is incorrect, reenter the commodity name.

[Example of an event flow description]

(4) Package diagrams

The package diagram is a variation of the class diagram. It represents the dependency among packages. A package refers to a grouping of model elements. The following figure illustrates an example of a package diagram.



[Example of a package diagram]

As shown in the figure, packages are mutually related through dependency. When using a package diagram, be careful so that the dependency is not bi-directional.

Exercises

1. Select a suitable term from the set of choices for [***square***] in the following sentences regarding the actor:

   [***(1)***] refers to the user who actually uses the system or other systems and hardware that are related to the system. If [***(1)***] is a person, it represents the [***(2)***] role of the person, instead of the user himself or herself. This enables the [***(3)***] to play multiple [***(4)***] of [***(1)***].

Answers:
a. abstracted
b. actor
c. same person
d. roles

2. Select a suitable from the set of choices term for [***square***] in the following sentences regarding the use case:

   The use case refers to what is obtained by [***(1)***] the relationships between the [***(2)***] and [***(3)***], and [***(4)***] a job of some unit performed by the user utilizing the [***(3)***]. It is a [***(5)***] in the [***(3)***] started by an [***(2)***] and is described by a [***(6)***] sentence. Features of the use case are that it is started by an [***(2)***] and is a [***(7)***] [***(5)***] for each job of some unit performed by the actor.

Answers:
a. abstracting
b. actor
c. function
d. simple
e. complete
f. system
g. complex
h. modeling

3. Select a suitable term from the set of choices for [***square***] in the following sentences regarding the extension relationship of use case:

The extension relationship refers to the [***(1)***] relationship between use cases. [***(2)***] is used as the [***(3)***] to be used in the extension relationship. The [***(3)***] is used to [***(4)***] the meaning indicated by the rectangles or lines in the diagrams.

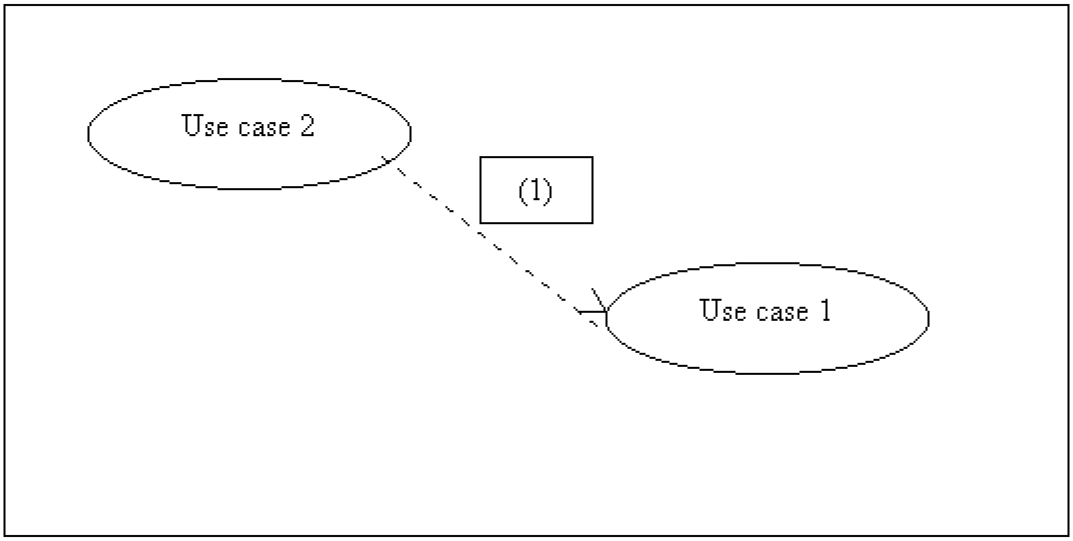
Answers:
a. clarify
b. stereotype
c. dependency
d. inheritance
e. <<extends>>

4. Select a suitable term from the set of choices for [***square***] in the following sentences regarding the use relationship of use case:

The use relationship refers, like the [***(1)***] relationship, to the [***(2)***] relationship between [***(3)***]. The difference from the [***(1)***] relationship is that all functions available in [***(3)***] must be fully used in the use relationship. The [***(4)***] to be used here is [***(5)***].

Answers:
a. stereotype
b. aggregation
c. extension
d. <<include>>
e. inheritance
f. <<include>>
g. class
h. use cases

5. Fill in with a suitable term [***square***] of the following figure relating to the use case invocation.



[Use case invocation]

6. Select a suitable term from the set of choices for [***square***] in the following sentences regarding the use case view:

The use case view describes requirements of the [***(1)***] to the [***(2)***] and [***(3)***] provided by the [***(2)***] to the [***(1)***]. This view is the central view in system development. Other views are [***(4)***], [***(5)***], [***(6)***], and [***(7)***]. Contents of the use case view are reflected in these views, which are used to describe other aspects. Thus, if the use case view is changed, other views are also [***(8)***].

Answers:
a. logical review
b. system
c. actor
d. parallel view
e. affected
f. deployment view
g. functions
h. component view

7. Fill in [***square***] with suitable terms in the following description of the logical view.

The logical view describes the [***(1)***] of the system of how to realize the [***(2)***] extracted in the use case. This view is mainly used by the [***(3)***] and [***(4)***].

8. Select suitable terms from the set of choices for [***square***] in the following sentences regarding the component view:

The component view describes the [***(1)***] between [***(2)***]. The following types of [***(2)***] are available:

1) Program [***(3)***]
2) [***(4)***] obtained as a result of compiling [***(3)***]
3) [***(5)***] obtained by linking all [***(4)***] required for execution

The component view is mainly used by the [***(6)***].

Answers:
a. dependency
b. source code
c. executable program
d. software components
e. binary code
f. designer
g. developer

9. Select a suitable term for [***square***] in the following sentences regarding the parallelism view from the set of choices. The same answer may be selected multiple times.

The parallelism view describes synchronous and [***(1)***] processing in parallel processing of [***(2)***] and threads. The purposes of the parallelism view are as follows:

1) Efficient use of resources such as [***(3)***]
2) Handling synchronization of communication between [***(2)***] or threads executed in parallel

The parallelism view is mainly used by the [***(4)***].

Answers:
a. developer
b. processes
c. CPU
d. asynchronous
e. designer

10. Fill in [***square***] with suitable terms in the following description of the deployment view.

The deployment view describes the physical layout of [***(1)***] and [***(2)***] and their relationships. The deployment view is mainly used by the [***(3)***].

11. Fill in [***square***] with suitable terms in the following description of tasks to be carried out when defining the use case.

- [***(1)***] the use case users.
- Summarize [***(2)***] about the use case of the use case users.
- Make a request to [***(3)***] the use case definition to the use case users

12. Select suitable terms from the set of choices for [***square***] in the following explanation about the domain object model. The same answer may be selected multiple times.

A domain object refers to an [***(1)***] that plays an important role in the [***(2)***]. A domain object model can be created as follows:

(1) Collect and build a [***(3)***].
(2) Find [***(4)***] in the domain in question and their [***(5)***].
(3) Extract classes contained in the [***(6)***].
(4) Divide the system into [***(7)***].
(5) Conduct the [***(8)***].

Answers:
a. workflow analysis
b. glossary
c. target domain
d. domain model
e. object
f. attributes
g. packages

13. Select a suitable term from the set of choices for [***square***] in the following sentences regarding the class diagram:

The class diagram describes the [***(1)***] structure of [***(2)***] existing in a system. The [***(1)***] structure refers to the relationships between [***(2)***] and the [***(3)***] and [***(4)***] held by [***(2)***]. The class diagram is used to view the problem domain [***(6)***] and [***(1)***] by focusing on [***(5)***] of the system.

Answers:
a. logically
b. functions
c. classes
d. attributes
e. static (statically)
f. operations

14. Select suitable terms from the set of choices for [***square***] in the following sentences regarding the interaction diagram:

The interaction diagram represents behaviors of a related [***(1)***]. It also shows how a [***(2)***] is implemented by the [***(3)***] flow and system. The examples of interaction diagram are a [***(4)***] and a [***(5)***].

Answers:
a. use case
b. sequence diagram
c. group of objects
d. message
e. collaboration diagram

15. Fill in [***square***] with suitable terms in the following notes when describing an event flow diagram.

1) Avoid detailed description of the [***(1)***].
2) Describe exception processing as an [***(2)***].

16. Select a suitable term from the set of choices for [***square***] in the following sentences regarding the package diagram:

The package diagram is a variation of the [***(1)***]. It represents the [***(2)***] among packages. A package refers to a [***(3)***] of model elements. When using a package diagram, be careful so that the [***(2)***] is not [***(4)***].

Answers:
a. bi-directional
b. grouping
c. dependency
d. uni-directional
e. class diagram

17. Fill in [***square***] with suitable terms in the following description of the collaboration diagram.

The collaboration diagram represents the [***(1)***] among [***(2)***] objects. It attaches primary importance to the [***(3)***].

18. Select suitable terms for [***square***] in the following sentences regarding the sequence diagram from the set of choices:

The sequence diagram represents [***(2)***] transmission [***(1)***] as a [***(3)***]. By using the sequence diagram, the flow of [***(2)***] can be grasped [***(4)***].

Answers:
a. message(s)
b. a time series
c. intuitively
d. among objects

19. Fill in [***square***] with suitable terms in the following description of the workflow analysis.

The workflow analysis is a method to analyze [***(1)***] that make up a [***(2)***] by making a [***(3)***] of the [***(2)***] flow . This enables [***(4)***] ,such as unnecessary work in the [***(2)***] , to be detected.

20. Select suitable terms for [***square***] in the following sentences regarding the use case detailing from the set of choices:

The purpose of detailing the use case is to describe in detail [***(1)***]/[***(2)***] the use case and the [***(3)***] including [***(4)***] between the [***(5)***] and [***(6)***] in detail.

Answers:
a. event flow
b. how to start
c. system
d. interaction
e. how to end
f. actor

5 Design

Chapter Objectives

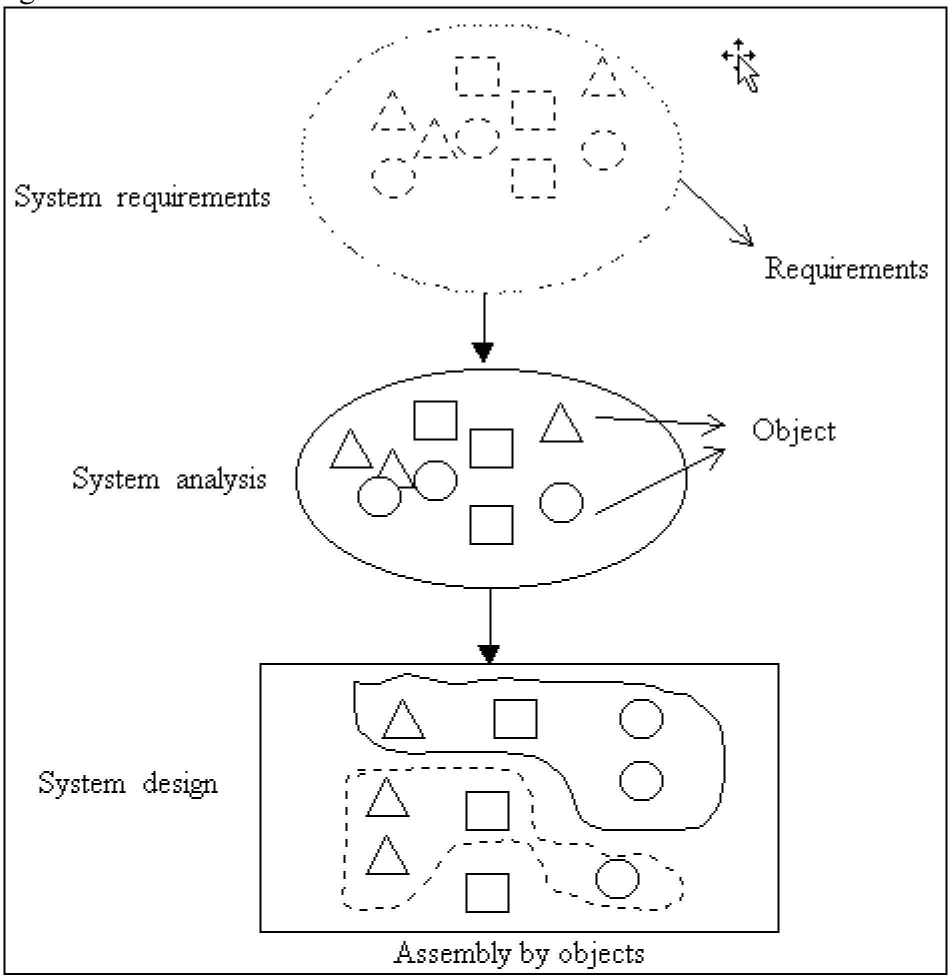
This chapter explains the design using the object-oriented methodology for the design process.

5.1 Architecture Definition

In the architecture definition, the system configuration and job restructuring are defined based on outputs from the analysis phase.

5.1.1 System component design

In system development based on object orientation paradigm, assembly as shown in the figure is carried out:



System implementation based on object orientation paradigm

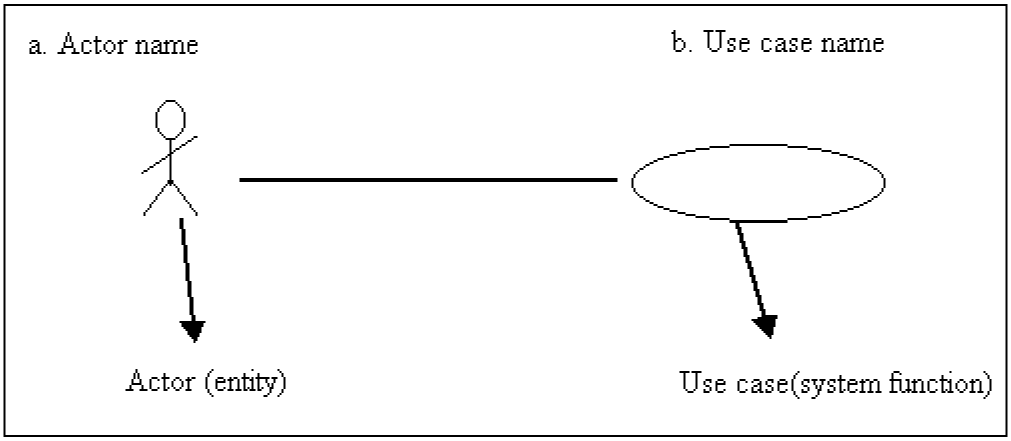
In system development based on object orientation paradigm, objects are extracted from the system requirements. Components that can be reused are created wherever possible, by making extracted objects more independent through encapsulation.

A system is constructed by combining extracted objects in this way.

5.1.2 Consistency between use cases and system components

How a system operates and reacts when viewed from outside is represented by the "use case diagram." This shows how a system should function (use case) and its external environment (actor). The use case diagram shows the system from the end user's viewpoint, thus enabling mutual understanding of the requirements through communication with end users and specialists in the area.

By using the use case diagram, the boundary between the inside and outside of a system can be clarified.



<Use case diagram>

a) Actor
- Represents the roles to be played by the system user.
- Actively exchanges information with the system and passively receives information from the system.
- Humans, hardware, and external systems are the actors.

b) Use case
- Models interactions between the actor and system.
- Started by the actor and perform some function of the system.
- Abstraction of one unit job to be performed by the system user using the system
- A collection of all use cases indicates everything about how a system is used.
- Each model of (analysis, design, implementation, and testing) can be associated and managed from the use case viewpoint.
- "Outline" and "Event flow" are use case documents.

Use cases are used to analyze requirements. Normally, a document is created for each use case simultaneously, when a use case diagram is created. The use case documents describe the purpose of use case in a few lines and a job event flow.

Because development requirements may be summarized with the user based on the document, the document must be written using terms that the user can understand.

The event flow describes when and how a use case starts and ends, and interaction between the actor and use case in writing. At this point, the user interface need not be described in detail, because usage of the user interface must be rewritten each time it is changed. Also describe exceptions as an exception flow.

Documentation of the flow when a use case is performed is called a "scenario". The scenario is an important material for actually considering objects and classes. Two types of scenario are available.

The "basic scenario" corresponds to cases where some use case operates without problems, and the "secondary scenario" complements this basic scenario.

(1) Scenario
- Instance of the use case
- Description of the actual flow in writing when the use case is executed
- Each use case can be represented as a set of multiple scenarios.
- Source from which required classes and objects are extracted
- The system behavior can be represented specifically through examples.

(2) Types of scenario
- Basic scenario: Case where the use case operates without problems
- Secondary scenario: Exception to the basic scenario

A scenario can be written by describing the flow of processing for one use case in sentences with actual names and values.

In the analysis phase, mainly create a basic scenario for each use case. Then, when no new behavior is found, the creation has been completed. In the design phase, the secondary scenario is also needed for considering error handling.

While conducting scenario analysis, ensure that common points identified in the system, omissions in the use case description, and others are reflected in the use case models and documents.

5.2 Component Design

In the component design, components contained in subsystems are designed in more detail based on the outputs of the architecture definition, and also classes contained in each component are designed.

5.2.1 Design of a class in the component (Class diagram refinement and design of an interface between classes in the component)

(1) Class diagram refinement

The purpose of modeling relationships between objects in more detail is to deepen the understanding of the essential meaning of modeling targets. However, detailed modeling may make the model closer to the implementation code.

Making the model closer to the implementation code means whether unique implementation code can be created from the model. Clarifying the class attribute types, operation arguments, and return values is refinement that makes the model closer to the implementation code.

One advantage of modeling based on the class diagram is that the most important and stable concept structures from the target problems can be shared among the development team and users as an object model. This allows the whole team to recognize "requirements", "purposes", and "class design", which are the most important processes in software development, as static structures with less change. It is very important that all development staff share and recognize a stable model created in a virtual world.

(2) Design of an interface between classes in the component

The class provider needs to consider carefully the ease of use or understanding of a class. If this is neglected, the class, even if it is created, may not be used or may be used wrongly.

By specifying an interface, common points can be found in operations among multiple classes, and a collection of such classes can be defined as a superclass.

Common operations are called an interface (class library protocol). A class created to specify an interface often becomes an abstract class, and polymorphism is implemented by inheriting each class from this class.

5.2.2 Class design

(classes, class attributes, behaviors, relationships and aggregation, and extraction for generalization and refinement)

(1) Classes

There are similarities among objects, and information can be organized by focusing on object similarities. A class is "a thing" that is used to extract commonality among many objects and classify them.

The following items are needed to explain the class:

- Class purpose
- Class group (category name and package name to which the class belongs)
- Class applicable scope (description about which platforms allow the use of the class)
- Class usage (explanation about how to use the class, such as whether inheritable or not)
- Object diagram (attributes and public methods that require explanation)
- Related classes (explanation about other classes that are closely related)
- Conditions for using the class (conditions or constraints assumed for using the class)
- Attribute explanation
- Method explanation
- Examples of using the class (sample source code)

(2) Class attributes

Attributes are either public or private, and private attributes are hidden from outside the class. The following explains only attributes that are needed to use the classes, instead of all attributes of the classes. Such attributes may contain private attributes.

The following items are needed to explain attributes:

- Attribute name
- Attribute type
- Attribute meaning
- Values that can be taken by the attribute
- Lifecycle of the attribute (describe when to initialize and when to discard, if necessary)

(3) Class behaviors

Class behaviors are described in the class explanation. Methods are either public or private. Private methods are implemented only to realize class requirements and are not made public. In contrast, public methods have the class requirements themselves as their method names, and their names are made public.

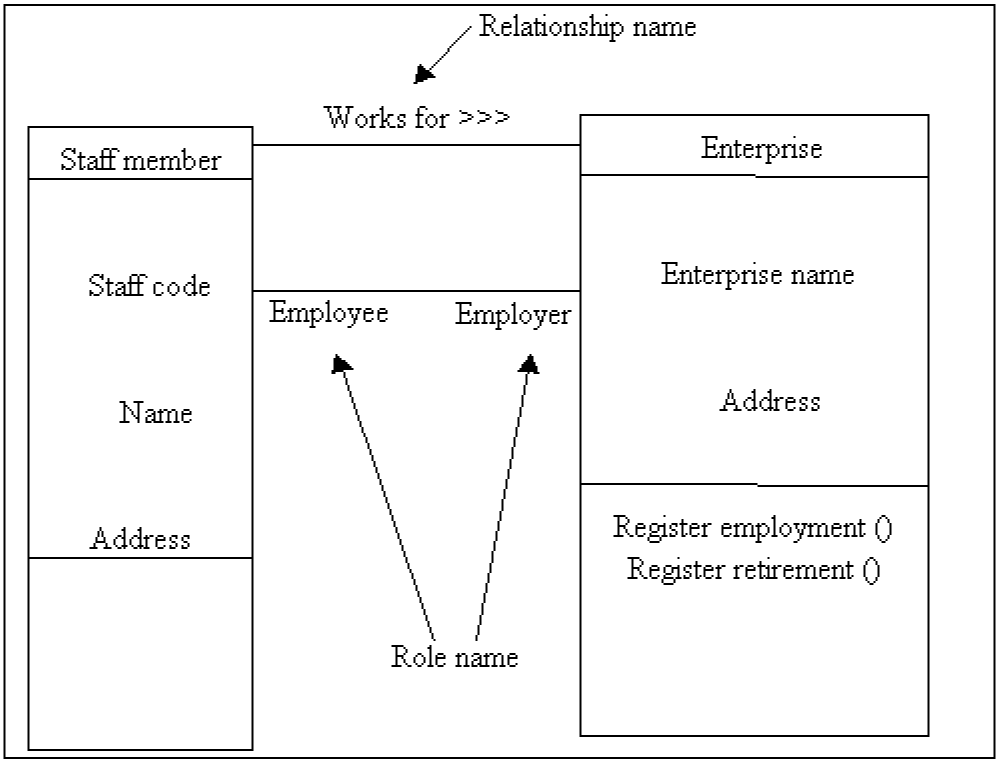
The following items are needed to explain methods:

- Method name
- Explanation about the method parameters and return value
- Functional explanation (explanation about functions to be provided to the user and exception processing)

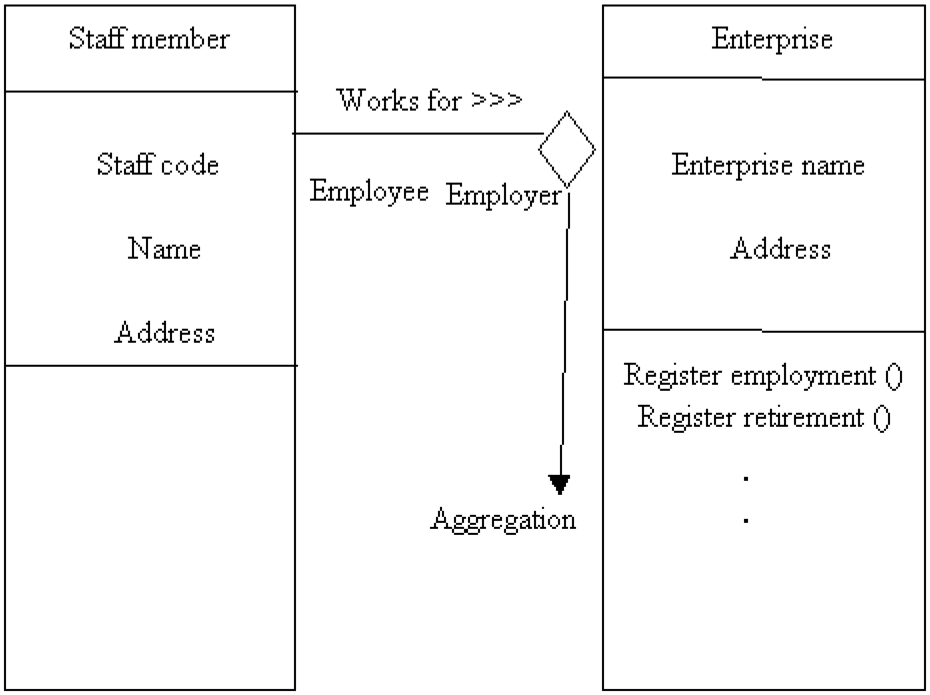
- Explanation about preconditions
- Explanation about conditions for using the method
- Explanation about whether overriding by inheritance is possible or not
- Examples of using the method

(4) Relationships and aggregation
The relationship name is used in the following figure to give a certain meaning to some link between classes. The role of each object in a relationship is shown. Viewed from the employee, for example, the enterprise has the role as an employer. Viewed from the enterprise, on the other hand, the staff has the role as an employee. If neither relationship name nor role name can be described, only the meaning "there is some relationship" can be represented between the enterprise and staff. The relationship name and role name clarify the meaning of the actual relationship (link) of an object created from two classes. The relationship name sets a certain criterion for the relationship of "what type of link exists" instead of vague relationships.

Information exchanged between classes linked by relationships is restricted by the relationship name to some degree.



The aggregation indicates "whole - part" and is a kind of relationship. In the following figure, the aggregation is indicated by placing a small diamond at the end of association line and means the relationship structure that the enterprise (whole) is made up of staff (parts).
The meaning of the whole and part has been added to the relationship between the enterprise and staff members in the previous figure. Aggregation can tell a model user which is the whole. As the number of classes shown in the class diagram increases, this technique to indicate the whole becomes more important.

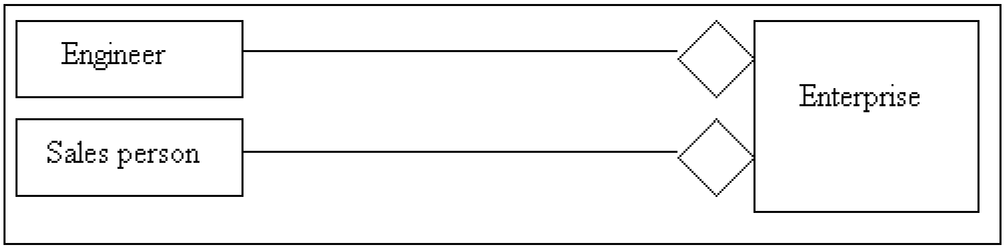


(5) Extraction for generalization, and refinement
"Generalization" is a modeling technique. To understand generalization, it is important to carefully consider the act of "analyzing things."

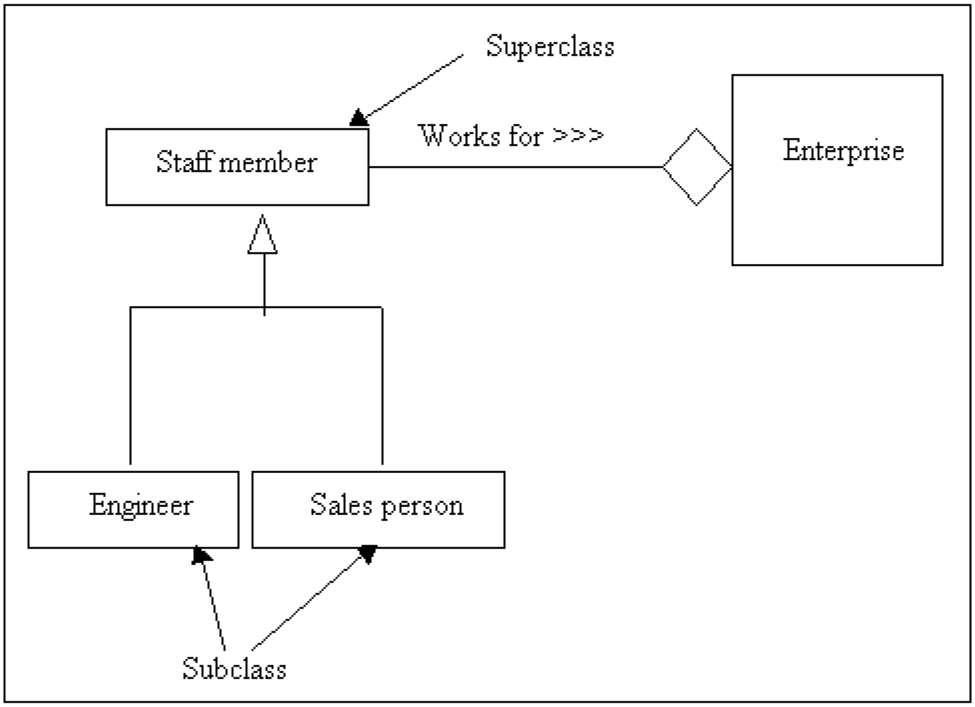
For example, analyze here "The enterprise has engineers and sales persons."

First, classify "things" from the above sentence.
- Enterprise
- Engineer
- Sales person

The following figure shows the modeling using the relationships of the enterprise, engineer, and sales person.



Organize information based on the previous figure. Since the enterprise employs engineers and sales persons, modeling using generalization including the relationship between the staff and enterprise is represented as shown in the following figure.



5.3 Detailed Design

In the detailed design, design is carried out in more detail so that classes contained in the component can be implemented based on the output of the component design.

5.3.1 Refining class structure

Software requires maintenance to meet changes in customer requirements and market demands. If data is embedded in the program, it is necessary to identify which parts of the program to modify and also to check whether other modules will be adversely affected by modifications.

If, however, software is structured so as to reference data, the program itself needs no change even if data values change. By making software independent of data, the software is easier to maintain.

As the number of classes in an object increases, it becomes necessary to group classes into categories related to the classes. It is important, when classifying objects, to model a hierarchical structure that begins with the most general items at the top level and go down to specific types of objects. How to model must be formulated with a class diagram for modeling.

The class structure based on execution performance, maintenance, etc. must be refined, keeping the those points in mind.

5.3.2 Considering reusability

Extract classes that provide common functions in the program to be implemented, in order to obtain a reusable class structure. The viewpoint of using components and that of component development are needed.

5.3.3 Considering object persistence

One of the features of the object-oriented database management system (ODBMS) is the support of extendable types.

In ODBMS, objects of all types can be made persistent.

During execution of a program, some objects are persistent objects that exist permanently, and other objects are non- persistent objects that are created temporarily.

In ODBMS, object identifiers are saved regardless of the object storage location (location independence), attribute value (value independence), structure change (structure independence) of relationship, etc.

It is necessary to distinguish between persistent objects and non- persistent objects to examine the execution efficiency, storage timing of persistent objects, and timing for disposing non- persistent objects.

Exercises

1. Select the most suitable sequence when the object-oriented system design is divided into processes of a to d from the group of choices below:

a. Standardization of object modeling and objects
b. Encapsulation design
c. Job process modeling
d. Control design

Answers:
(1) a-b-c-d
(2) a-c-b-d
(3) c-a-b-d
(4) c-a-d-b
(5) c-d-a-b

6 Implementation

Chapter Objectives
This chapter explains the implementation phase.

6.1 Implementation

In the implementation, the system is, based on design results, implemented in terms of components, such as the creation of source code and compilation into the executable file format. The system architecture has already been determined in the design phase.

The main purpose of implementation is to make a system whose detailed design is completed executable on computers. The implementation focuses on the following points:

(1) System integration plan required for each iteration
This implements a system by taking an incremental approach in which small manageable steps are repeated.

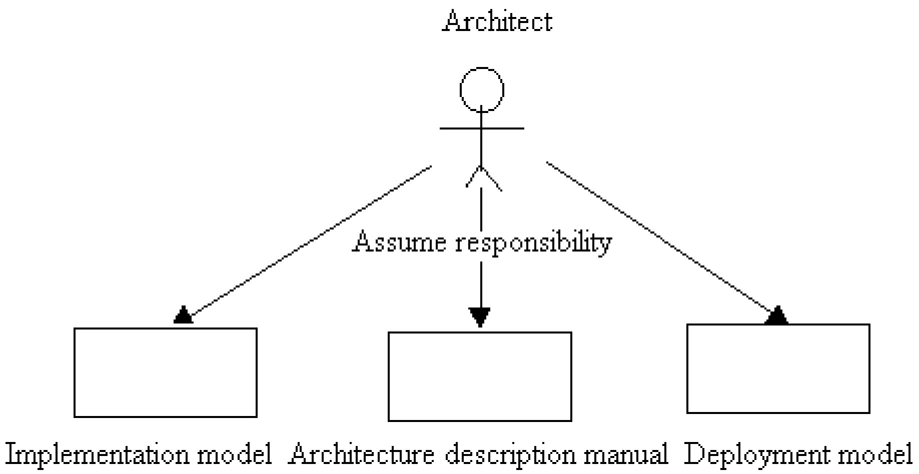
(2) System distribution
This refers to mapping of executable components to nodes of the deployment model. Mapping is performed based on active classes found in the design phase.

(3) Implementation of design classes and design subsystems found in the design phase
This implements design classes as file components containing source code.

(4) Unit test of components and their integration
Compile and link components to integrate them into one or multiple executable files. Then, pass them to the integration test and system test.

The following diagrams show the relationships of how to carry out implementation, and the workers (jobs that can be assigned to personnel) and the products which are involved.

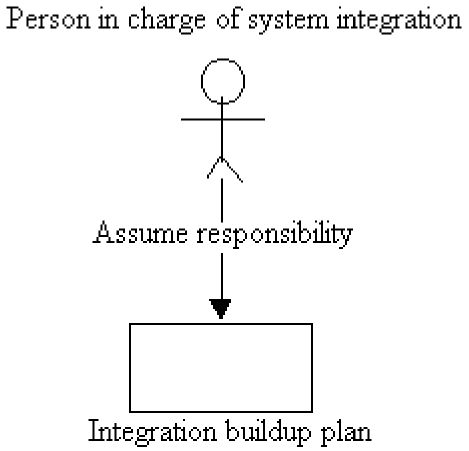
(1) Architect
-The architect is in charge of guaranteeing that the overall implementation model is correct, consistent, and easy to read with respect to its integrity.



(2) Person in charge of system integration

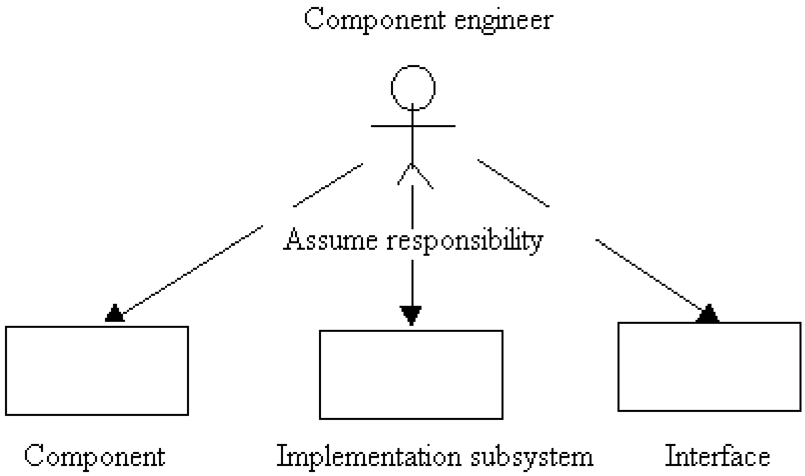
- The person in charge of system integration is responsible for creating an executable architectural baseline by integrating components corresponding to subsystems.
- Architectural baseline

Baseline released at the end of the elaboration phase, focusing on the system architecture



(3) Component engineer

- The component engineer is in charge of defining and maintaining the source code of one or multiple file components, to check whether functions are implemented correctly in each component.

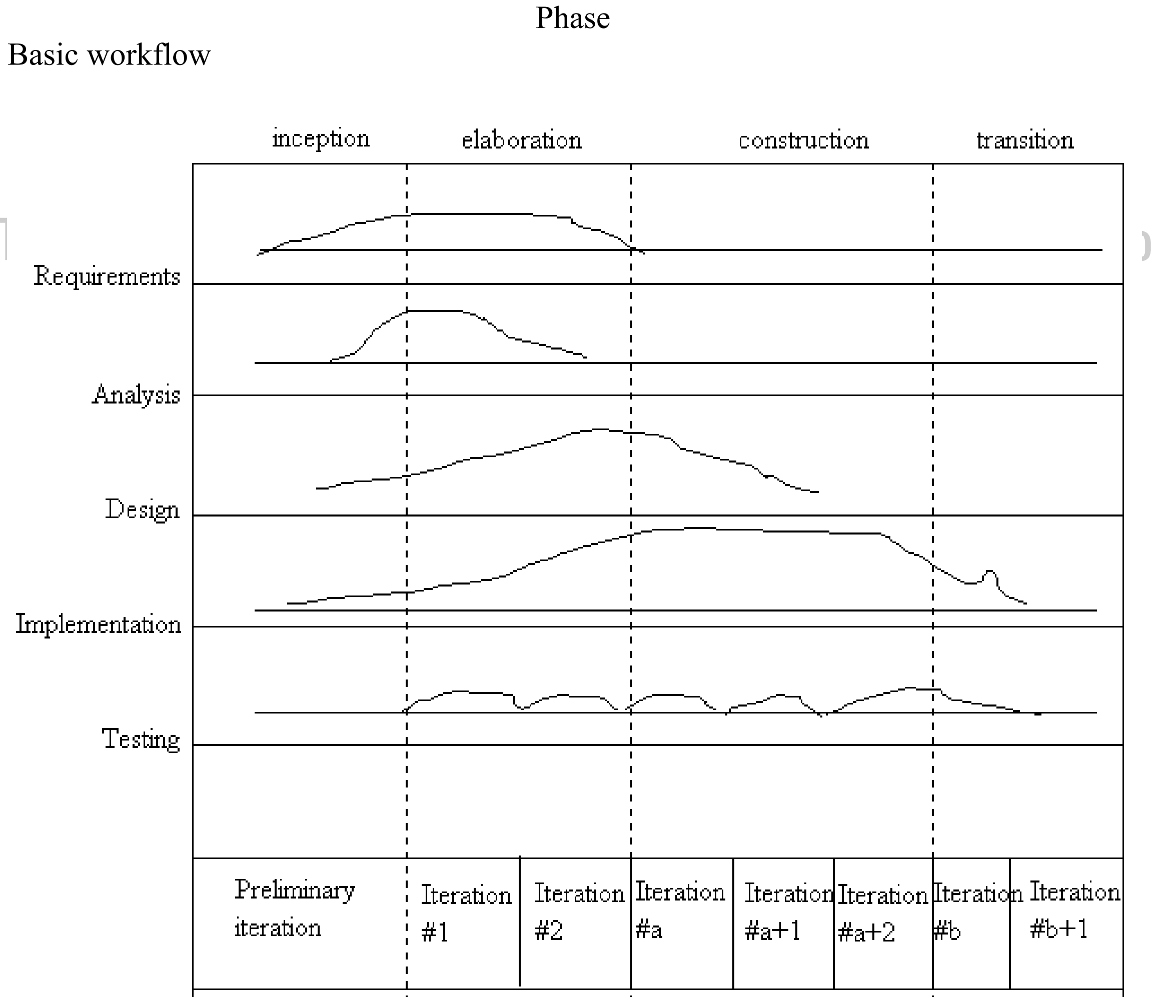


The following figure represents the transition of labor in the implementation workflow. As is evident from the figure, the implementation consumes most resources in the iterations of the construction phase.

Also when creating an architectural baseline that can be executed in the elaboration phase and when handling defects found in the beta release of a system in the transition phase, the weight for the implementation work becomes high (a projecting part in the transition section).

- Beta release

Distribution of a product under development to the users



6.1.1 Selecting the programming language

Here, define the pattern that specifies the description format of the programming language.

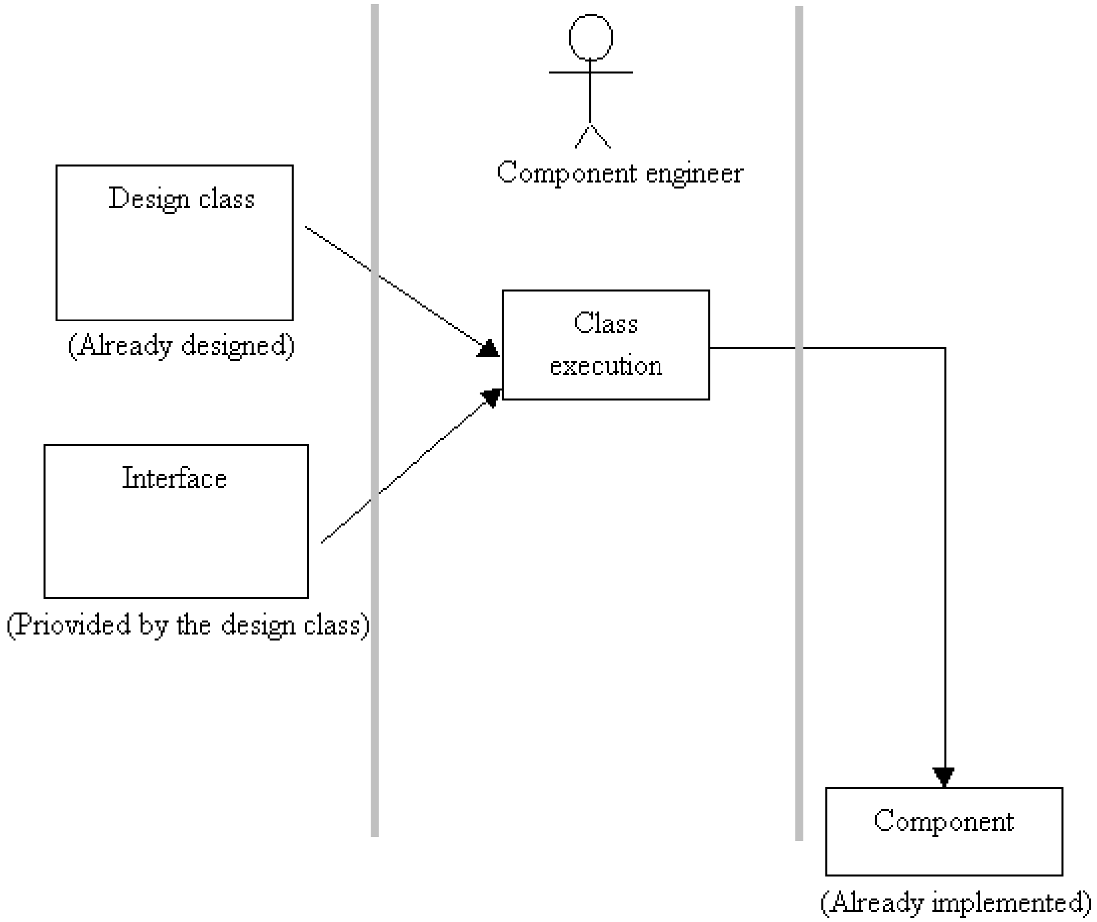
Object-oriented technology includes various object-oriented languages.

C++, Smalltalk, Eiffel, Objective-C, and Java are typical object-oriented languages.

6.1.2 Implementing classes

The purpose of class implementation is to implement a design class in a file component.

The following diagram shows the input and results of class implementation.



Take the following four steps to implement a design class as a file component:

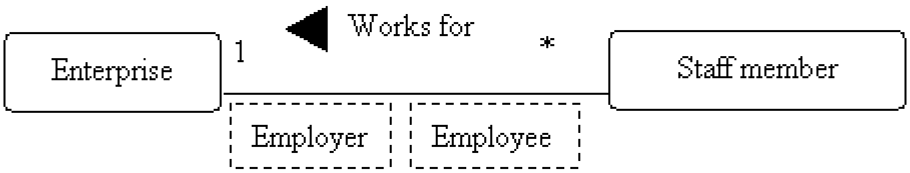
(1) Create an outline of the file component
The code that implements a design class is classified as a file component. Thus, it is necessary to create an outline of the file component that implements the design class and to decide the scope of coverage of the class. Normally, multiple design classes are implemented in a single file component.

However, the outlining method of the file component is restricted by the modularization approach and prescription of files of the programming language to be used.

When using Java, for example, one Java file component is created for each class implementation. The selected file component needs to support the compilation, installation, and maintenance of the system.

(2) Create code from the design class
In the design, many details of the design classes and their relationships are described using the syntax of the programming language. Part of the source code that implements classes can be easily created. This is useful for operations and attributes of classes and relationships in which classes participate. Because, however, only the operation signature (name and parameters of behavior characteristics) is created, implementation is needed.

Note that it is quite difficult to create code from relationships and aggregation defined in the design model. The creation method depends largely on the programming language to be used.
A relationship refers to a link between instances created from a class.
A relationship can be represented in more detail by using the relationship name and role name



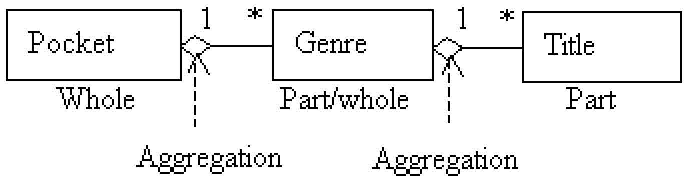
1) Relationship name: Clarifies the meaning by giving a name.
In the figure,

- [***] in "Works for" ◆ indicates the direction, instead of the ownership.

- Means that "staff member works for the enterprise."

2) Role name: Role played by the class
In the figure,
- The staff member seen from the enterprise is an employee, because he or she plays the role of an employee.
- The enterprise seen from the staff member is an employer, because it plays the role of an employer.

The aggregation is the specialization of relationships and indicates the relationship between the whole and parts.



- Multiple title classes are aggregated to one genre class.
- Multiple genre classes are aggregated to one pocket book class.

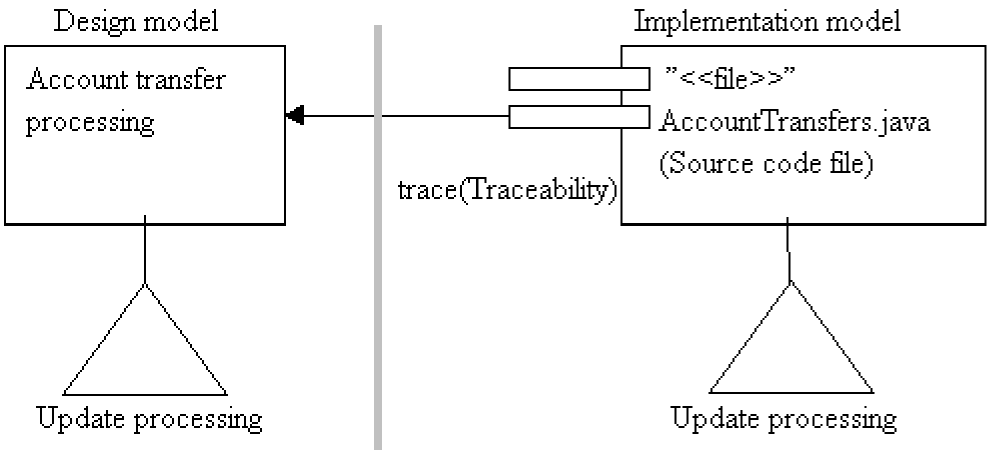
(3) Implementation of operations as methods
Operations defined in the design classes, as long as such operations are not "virtual operations (abstract operations)" implemented in a subtitle, must all be implemented. The term "method" is used to represent this implementation of operations. Examples of methods in a file component are: Java methods, Visual Basic methods, C++ member functions, etc.

To implement an operation, it is necessary to select appropriate algorithms and data structures (local variables of the method) to be supported and to code action required by the algorithms. In the design of a design class, methods may be specified by a natural language or pseudo-code. Design methods must be used as inputs of this step.

(4) Components that provide interfaces
A resultant component must provide the same interface as that of the class to be implemented.

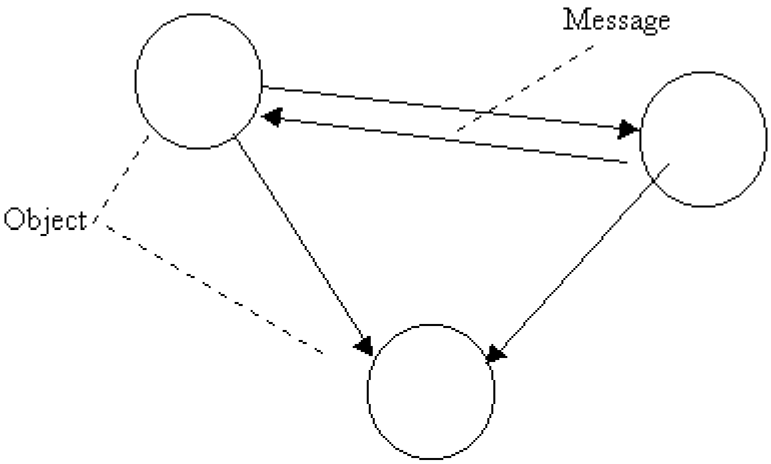
The following figure shows a component that provides the same interface as that of the class to be implemented.

For example, the design class "account transfer processing" provides "transfer processing" in its interface. The component "AccountTransfer.java" that implements the class "account transfer processing" also provides the "transfer processing" interface.



6.1.3 Implementation of methods

Implement by the member functions the procedures to do execution for messages.



Information exchanged between objects is called messages. Upon receipt of a message, an object starts a method to process the message. In actual programming, an operation to send a message to an object is replaced by executing a method of the receiver object.

Because a method implementation is a function (member function), the method may have a return value. In such a case, the return value is a message sent from the receiver object to the sender object. That is, messages are sent and received, if the method has a return value.

Two special member functions called the constructor and destructor are included in the object:

(1) Constructor
This is a function to initialize an object

(2) Destructor
This is a function used to release objects that are no longer used in C++.
In Java, the function corresponding to the constructor is called the finalizer.

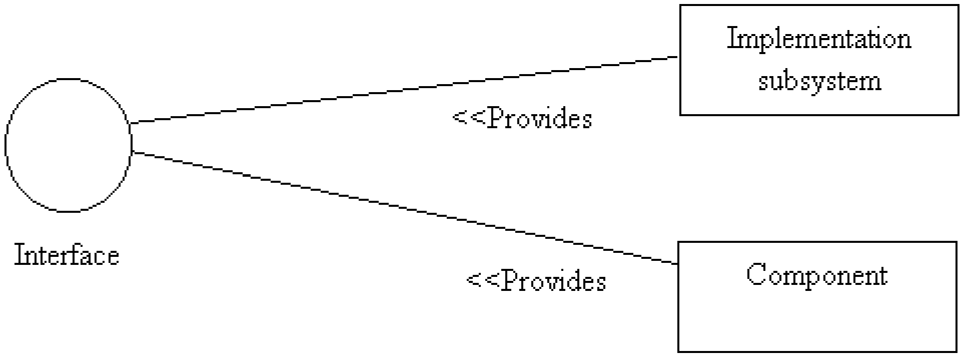
These functions are called just after creating an instance or just before discarding it, to initialize or clear a member function.

6.1.4 Implementing user interfaces

To implement a user interface, first use the interface for operations to be provided by the design class or design subsystem, in the implementation model. Then, specify the operations to be implemented in the component and implementation subsystem.

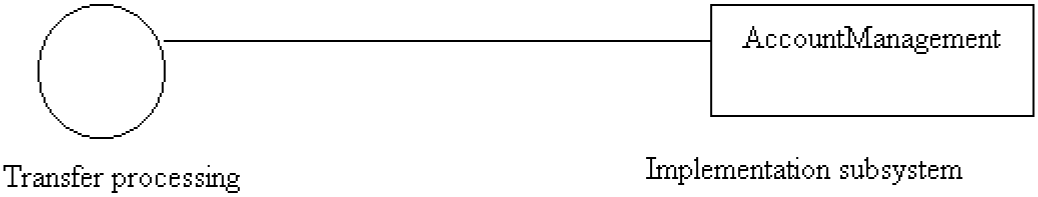
The following figure shows the relationship, which is a key to the interface. The component and implementation subsystem can have "use dependency" with respect to the interface.

A component that provides an interface needs to implement all operations defined in the interface correctly. An implementation subsystem that provides an interface must contain the component or subsystem that provides the interface.



The following figure shows an implementation subsystem that provides an interface.

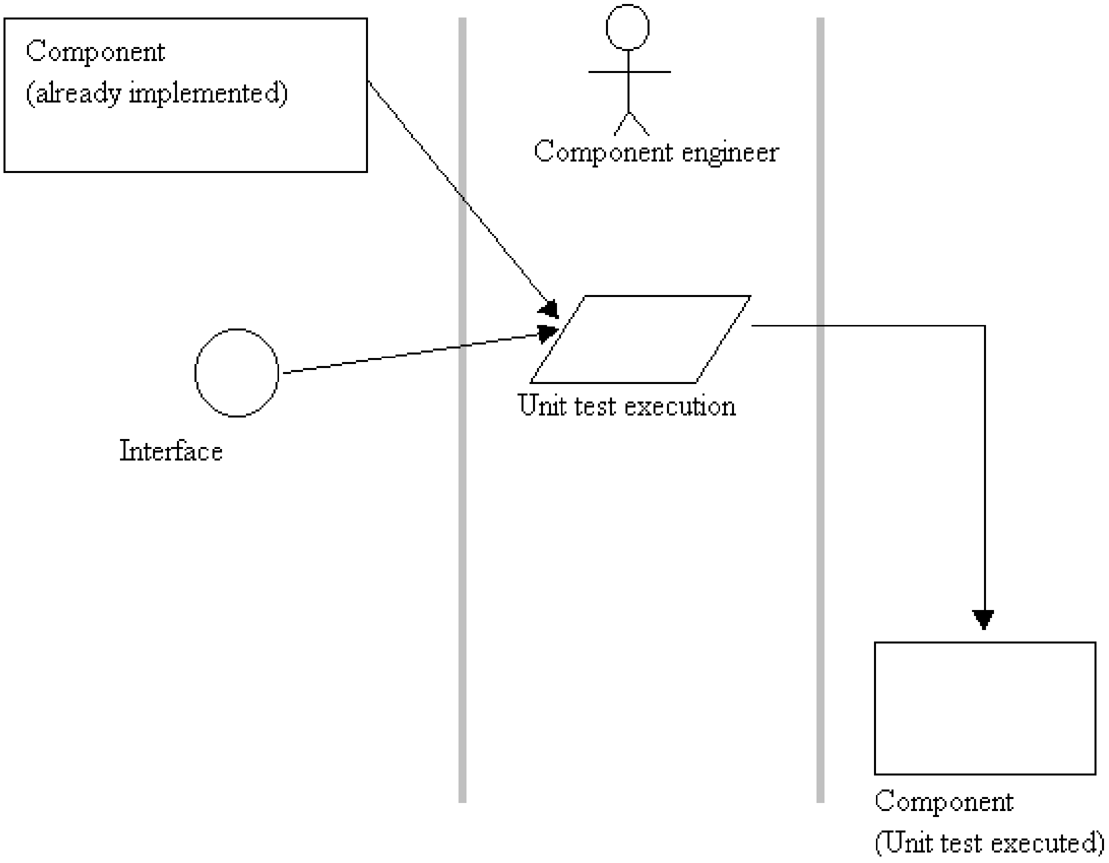
The Bank system has an implementation subsystem named "AccountManagement (Java package)" and this subsystem provides the interface "transfer processing."



6.1.5 Creating test data

Verify the results of implementation. Test methods are: the unit test in which each of the implemented components is tested individually; the integration test which checks whether multiple integrated components behave normally; and the system test.

(1) Test each of the implemented components individually.
The following figure shows the test input and results.



In the unit test, the following tests are targeted and performed:

1) Specification test
In this test, the unit's behaviors that can be observed externally are verified. This is called the black box test. The specification test verifies only the component behaviors without considering how component behaviors are implemented in the component. Thus, what kind of output is returned is verified, when specific input is given to the component and the test is performed in a specific state. The range of input/state/output is divided into equivalence classes.

"Equivalence class" refers to the input/state/output that provide a series of values at which object behaviors can be regarded as the same. By testing the components with combinations of the equivalence classes of input/state/output, the same effect as the test by all the individual combinations can be obtained.

2) Structure test
The structure test verifies the internal implementation of a unit. This is called the white box test. This test checks whether each component operates internally as intended.

The component engineer must test all codes in the structure test. This test must be executed at least once for each of the codes. Paths in the code can be divided into general paths, critical paths, unknown paths among paths that mediate algorithms, and paths relating to high risks. Path tests must also be performed due to their importance.

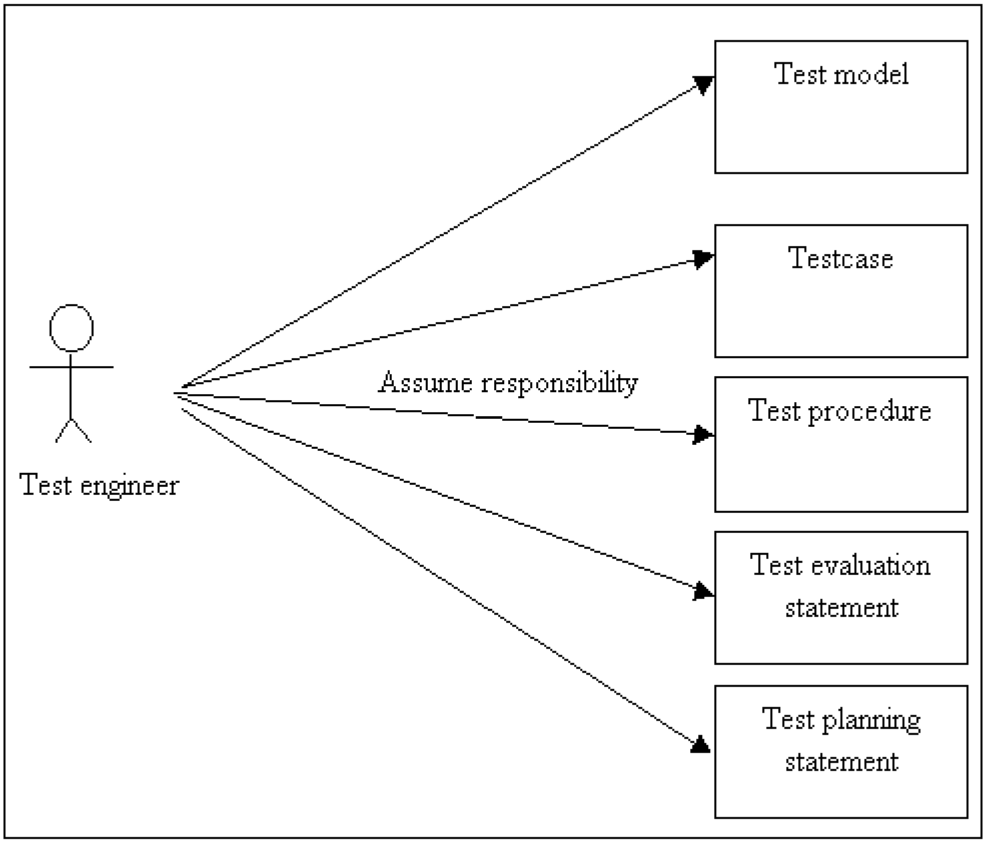
(2) Tests that verify implementation results should focus on the following points:
1) Test plan required for each iteration
The integration test is needed for all builds created in iterations and only for the last iteration in the system test.

2) Test design and implementation
Create the test cases that specify what test to perform and the procedure that specifies how to perform the test. If the test can be automated, create an executable test component.

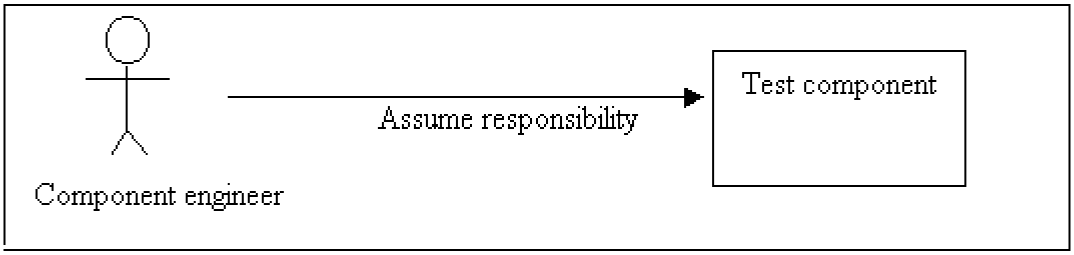
3) Execution of various tests and systematic handling of test results
Test any build again in which a defect has been found. Correct any serious defects by going back to the basic flows, such as the design and implementation.

The following three figures show the workers and artifacts involved in the tests.

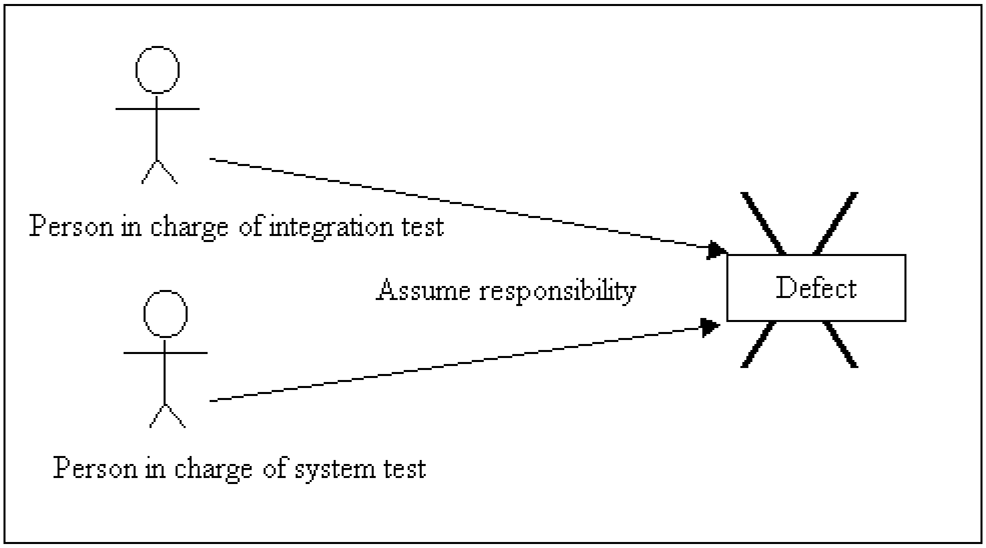
(1)



(2)



(3)


Person in charge of integration test
Assume responsibility
Defect
Person in charge of system test

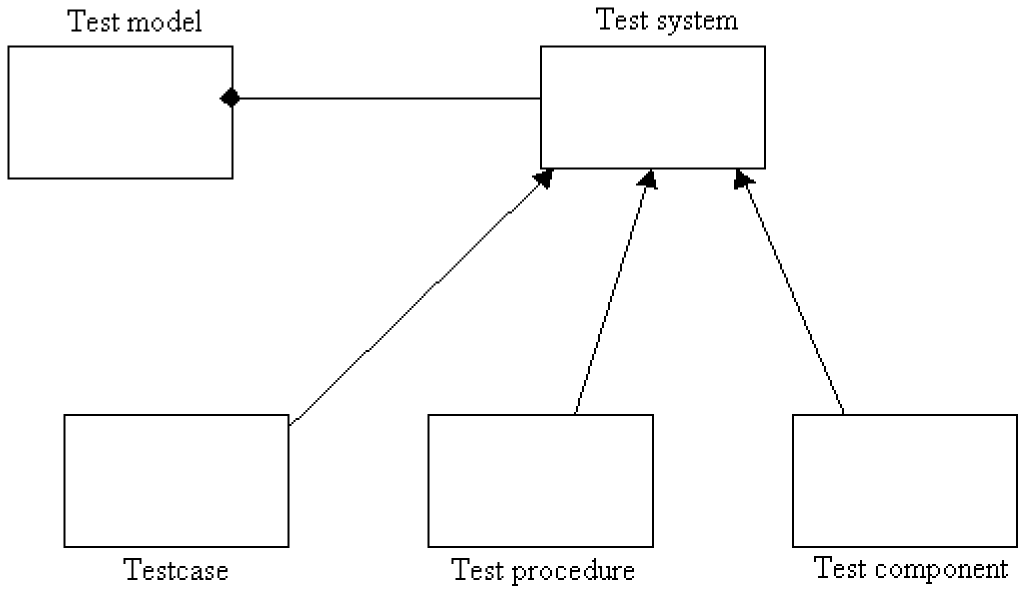
A test plan is created when the system responsibility assignment is decided in the inception phase.

The integration test and system test are performed to correct defects detected in the initial operation and the regression test is performed iteratively. In the last stage of development, many regression tests are performed. Like the following items, the test model must be maintained while developing it:

- Remove old testcases (corresponding test procedures and test components).
- Modify part of the testcases for the regression testcases.
- Create a new case for the preceding build.

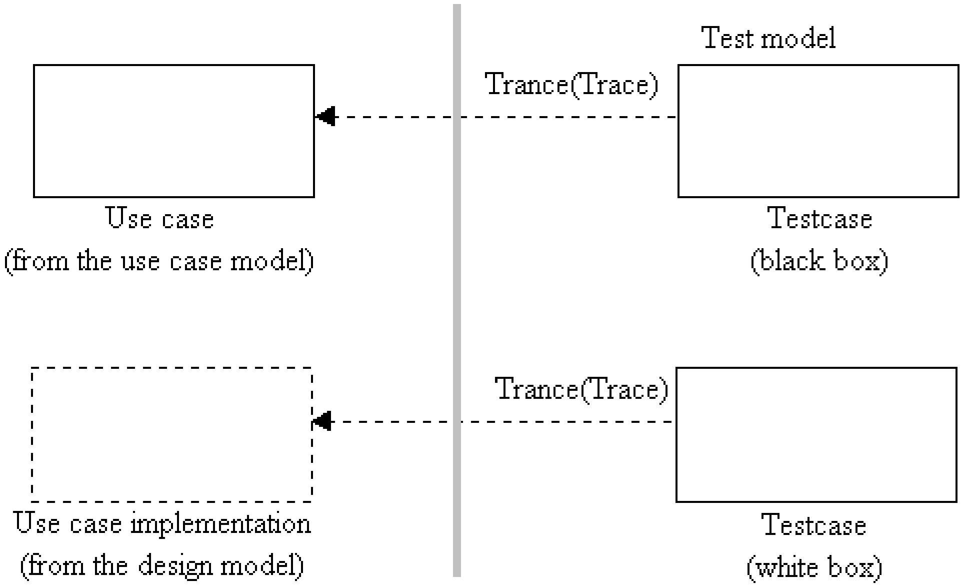
The test model is a method to test executable components (builds) in the implementation model by the integration test or system test.

The following figure shows a test model:



The test model is a set of testcases, test procedures, and test components. If there are many test-cases, test procedures, and test components, use packages in the model so that they can be handled more easily.

The following figure shows some test cases:



The testcase can be taken from use cases in the use case model or "use case implementation" in the design model. It is thus possible to trace how the test case was created.

The following cases are used for the test case:

(1) Method to test use cases or specific scenarios
This test case verifies whether preconditions specified in the use case are satisfied as a result of interaction between the actor and the system, and the sequence of actions specified in the use case is followed.

Because the test case based on a use case is a black box test of a system, the test checks external behaviors of the system.

(2) Method to test "use case implementation - design" and specific scenarios
This test verifies interaction between components that implement a use case.

Because the test based on "use case implementation" is a white box test of a system, the test checks internal interaction of components of a system.

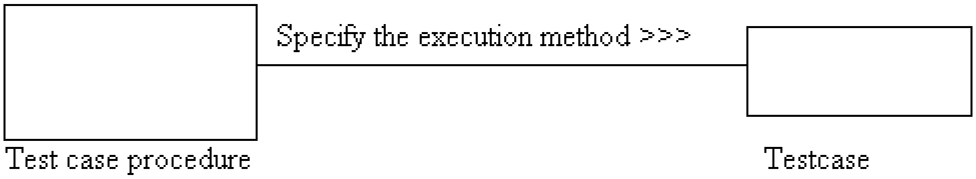
The following explains the test cases that specify the tests of the whole system:
(1) Installation test
The installation test checks whether a system works correctly when installed on the customer's hardware.

(2) Configuration test
The configuration test checks whether a system works correctly when the system configuration such as the network configuration changes.

(3) Negative test
The negative test terminates a system abnormally to reveal weaknesses of the system. The test engineer performs tests in ways that are not designed such as a test in an incorrect network configuration, with insufficient hardware capacity, and with overload.

(4) Stress test
The stress test checks problems that may arise in a system when resources are insufficient or resource contention occurs.

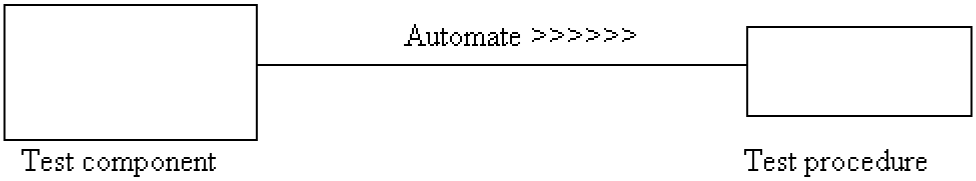
The test procedure is a method to perform one or multiple test cases or a part of them. The following figure shows a test procedure:



Examples of the test procedure are a procedure to perform testcases manually and an operation procedure of a test automation tool for creating executable test components. It is more effective if a test procedure is reused for multiple testcases or multiple test procedures are reused for one testcase.

One or multiple test procedures or a part of them is automated in the test component, and executable components (builds) in the implementation model are checked in the integration test or system test.

The following figure shows a many-to-many relationship between test components and test procedures:



The following explains the integration test case.

The integration test case is used to verify whether components interact correctly after integrating components into the build.
- Identify the testcases for each build.
- Identify the test procedures to specify the execution method of testcases.

The system testcase is explained in the following.

The system test is used to verify whether a system functions correctly as a whole. In each system test, a combination of use cases instantiated under different conditions is tested.

The test designer decides the priorities of use case combinations to develop system testcases, paying attention to the following points:
- Are parallel operations required?
- Are operations likely to be performed in parallel?
- Are operations likely to affect one another if they are performed in parallel?
- Are multiple processors involved?
- Are system resources, such as processes, processors, databases, and communication software, used frequently in complex ways?

Most system test cases can be found by focusing on the event flow of use cases and special requirements (performance requirements).

7 Major Object-Oriented Techniques

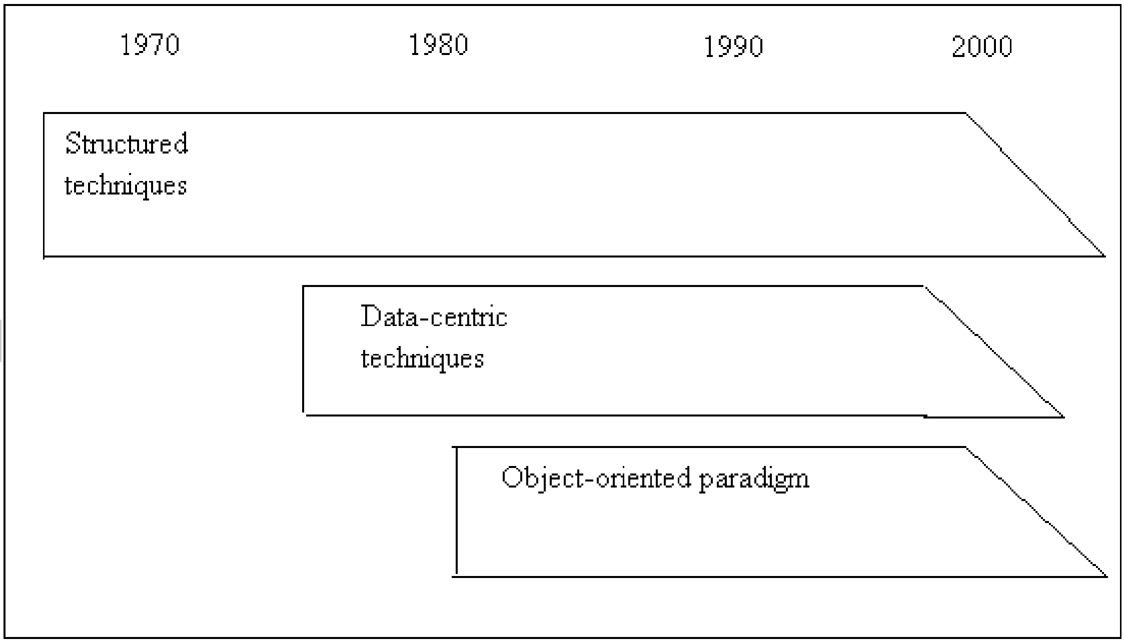
Chapter Objectives

This chapter provides knowledge about methodologies, techniques, and languages within the scope of the object-oriented paradigm.

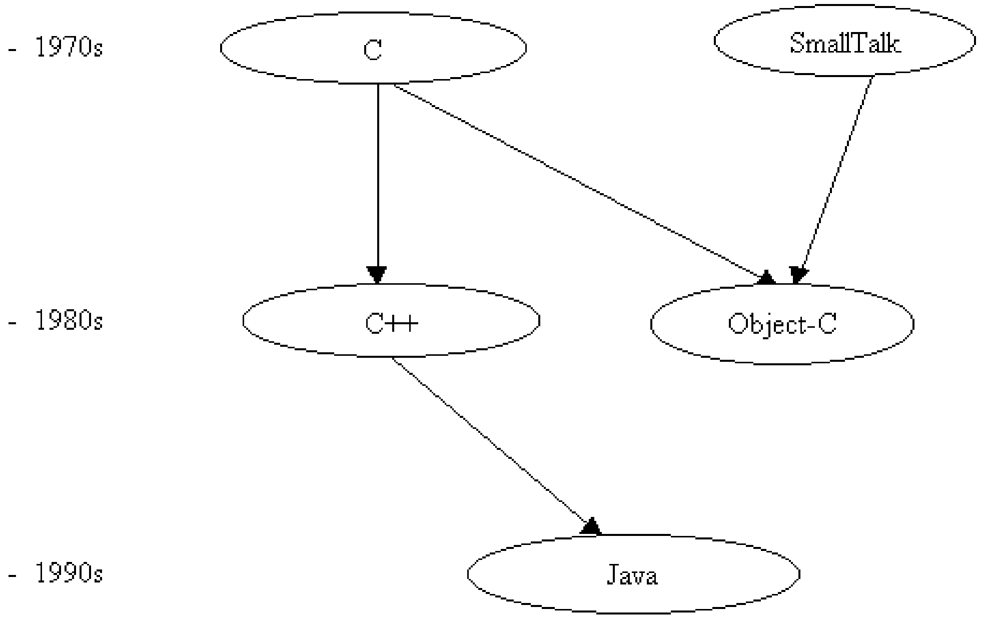
7.1 Object-Oriented Methodologies

This section explains the object-oriented methodologies. An object-oriented methodology is a generic name of the concept, work procedures, and document notation for developing an object-oriented system. Since 1970, various methodologies have been proposed.

The following figures show the progress of analysis and design techniques, as well as the history of the object-orientation.



[Progress of analysis and design techniques]



1970s…C language and SmallTalk were developed.
1980s…C++ and Object-C were developed to include the object-oriented paradigm in C.
1990s…Java, which enhances object orientation of C++, was developed.

[History of object-oriented paradigm]

This section explains the object-oriented analysis methods, using the OMT, Booch, Coad & Yourdon, Shlaer & Mellor, and Catalysis methods as examples.

7.1.1 OMT method

The OMT method is a methodology that covers the range from the analysis process to the design and implementation process in software development based on the object-oriented paradigm. It has two functions: 1) modeling the relation of data structures on the basis of the analysis process and 2) the development process goes on repetitively. The OMT method consists of the following three phases:

- Analysis
- System design
- Object design

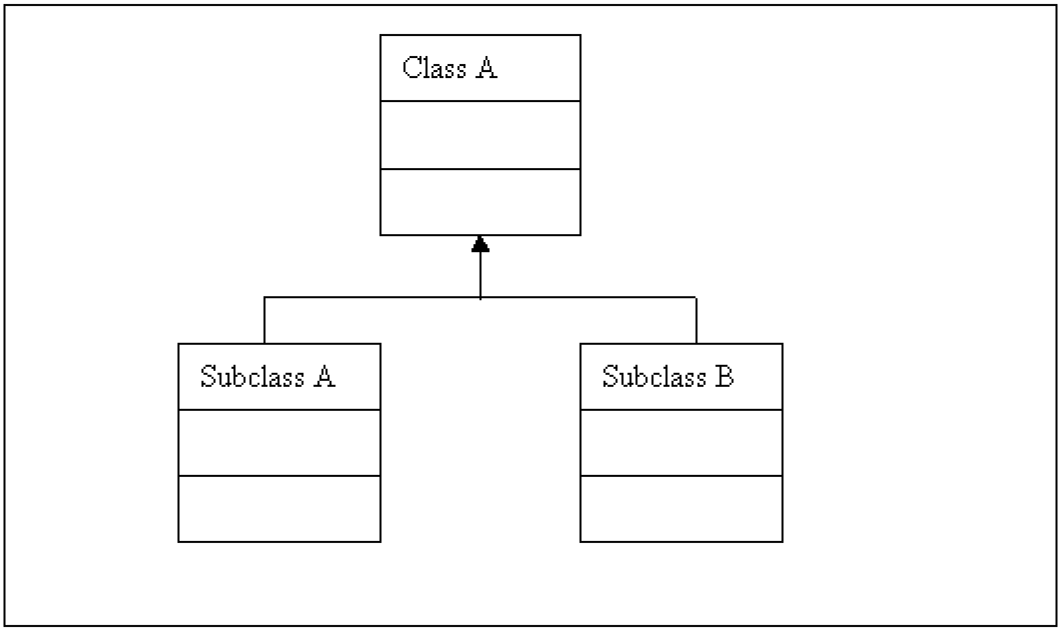
(1) Analysis

The following three models are developed in the analysis phase: object model, dynamic model, and function model.

1) Object model

The object model expresses data with object diagrams, regarding a system as an object. The object diagram shows the relationship between classes and objects.

The following figure shows an example of the object model.



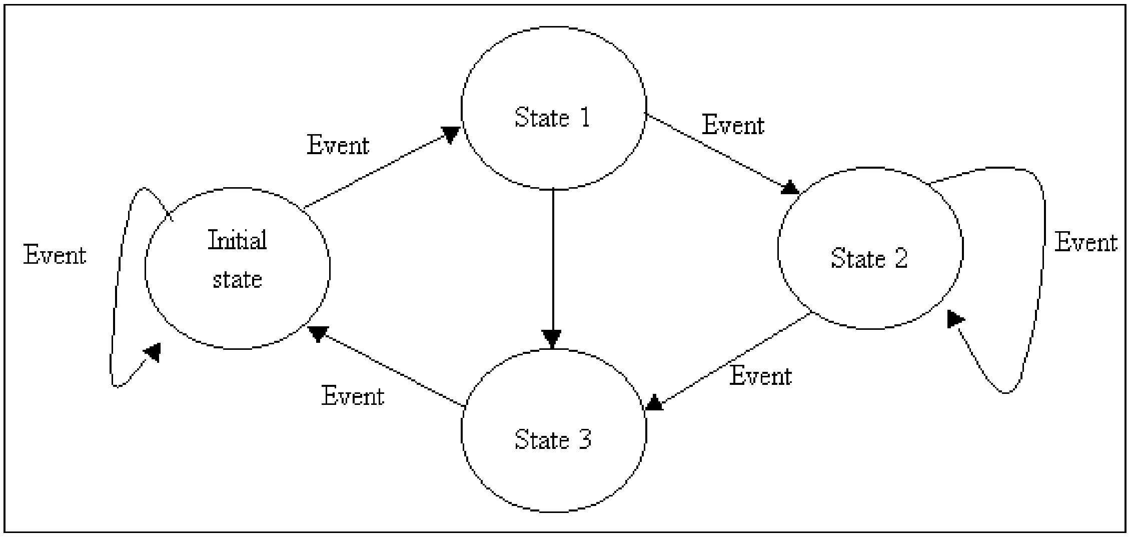
[Example of object model]

2) Dynamic model

The dynamic model indicates based on a state transition diagram how an object value, which is expressed as a state, changes against a specific external stimulus.

The model therefore explains how an object reacts to an event and how object interacts.

The following figure shows an example of such a state transition diagram.

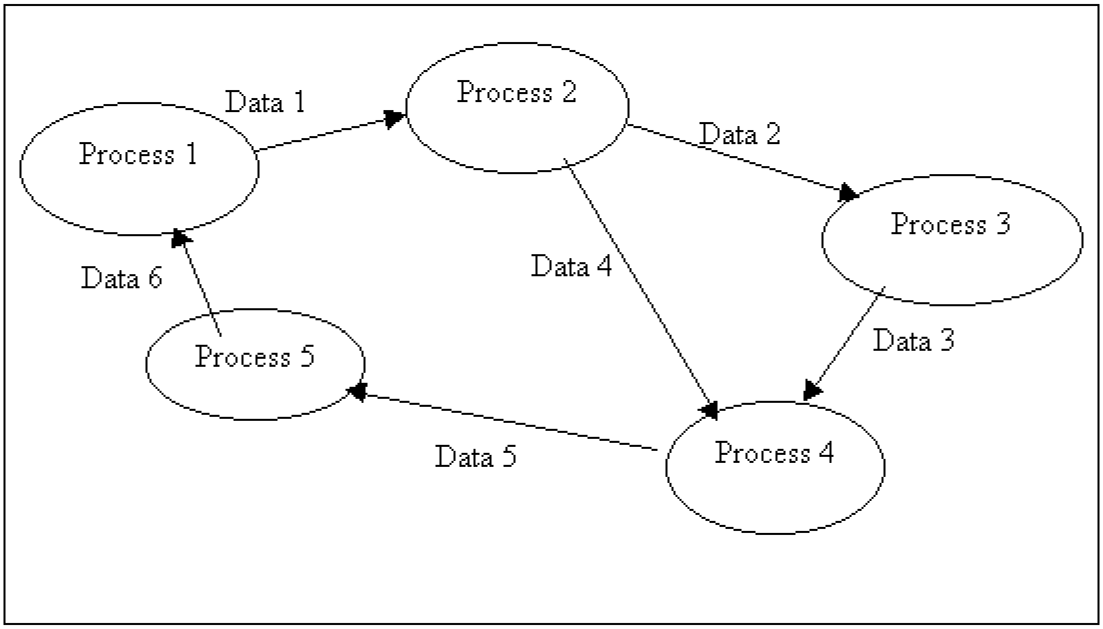


[Example of state transition diagram]

This figure indicates how the object state changes from the initial state when an event occurs.

3) Function model
The function model describes the change of an object value and applicable constraints using a data flow diagram.

The following figure shows an example of the data flow diagram.



[Example of data flow diagram]

This figure shows how the data value changes from data item 1 as it go through process 2 to process 5, beginning with process 1.

In the analysis process, a simple and accurate model of the real world is created. In this process, the focus is on the question “what should be done?”; therefore, one need not consider the implementation and applicable restrictions, such as “how should it be processed?”

(2) System design
In the system design process, divide a system into subsystems, and determine an interface between subsystems. In this case, one must consider the initialization, termination, and fault conditions of a program.

(3) Object design
In the object design process, follow the system analysis procedure to obtain a detailed model for implementation.

7.1.2 Booch method

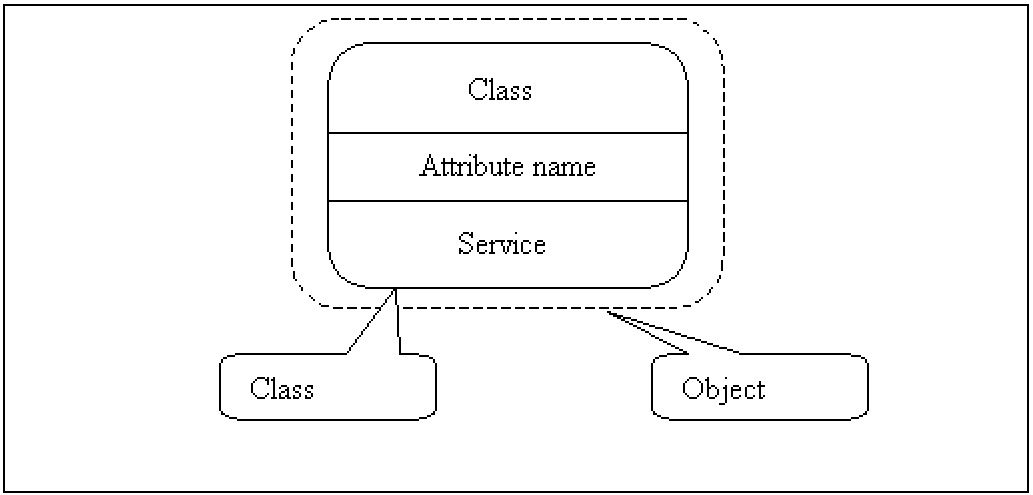
The Booch method specifies the applicable points of view for documentation in detail. This method provides a consistent means for reflecting the characteristics of the object-oriented paradigm on the design process. This analysis method was developed while taking into account implementation in a programming language such as C++.

7.1.3 Coad & Yourdon method

The Coad & Yourdon method expresses system structure and operations using class and state transition diagrams. This section explains how a document is to be created in an analysis process based on the Coad & Yourdon method.

(1) Class diagram

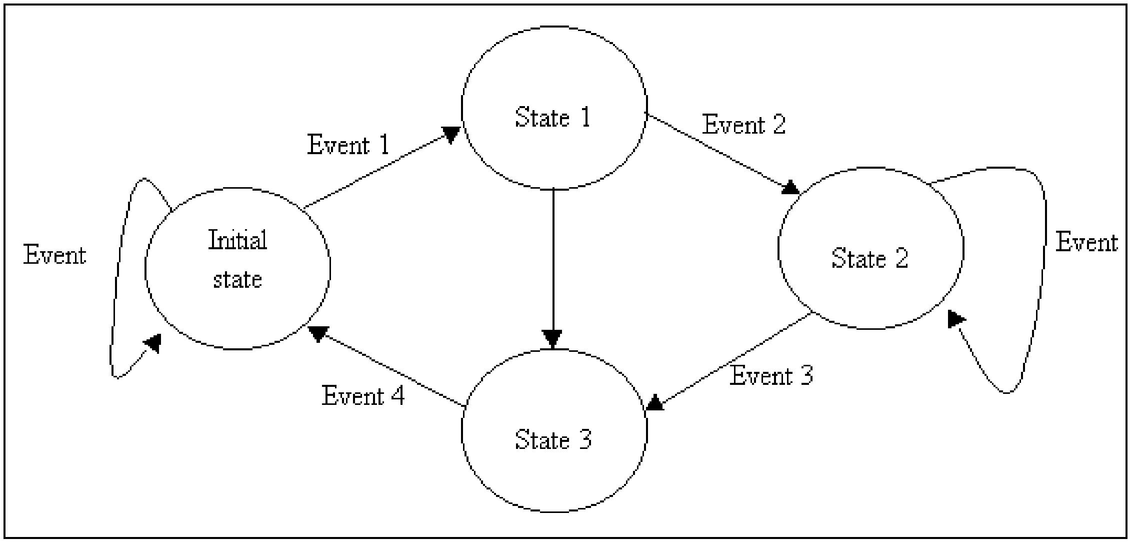
The following figure shows an example of the notation for classes and objects.



[Example for class and object notation]

As shown in this figure, the Coad & Yourdon method expresses an object with a dotted-line round square and a class with a solid-line round square. In this figure, “service” stands for a method of the object.

(2) State transition diagram
The following figure shows a state transition diagram.



[Example of state transition diagram]

This figure indicates how the object changes to state 1 after event 1 occurs for the object in initial state.

7.1.4 Shlaer & Mellor method

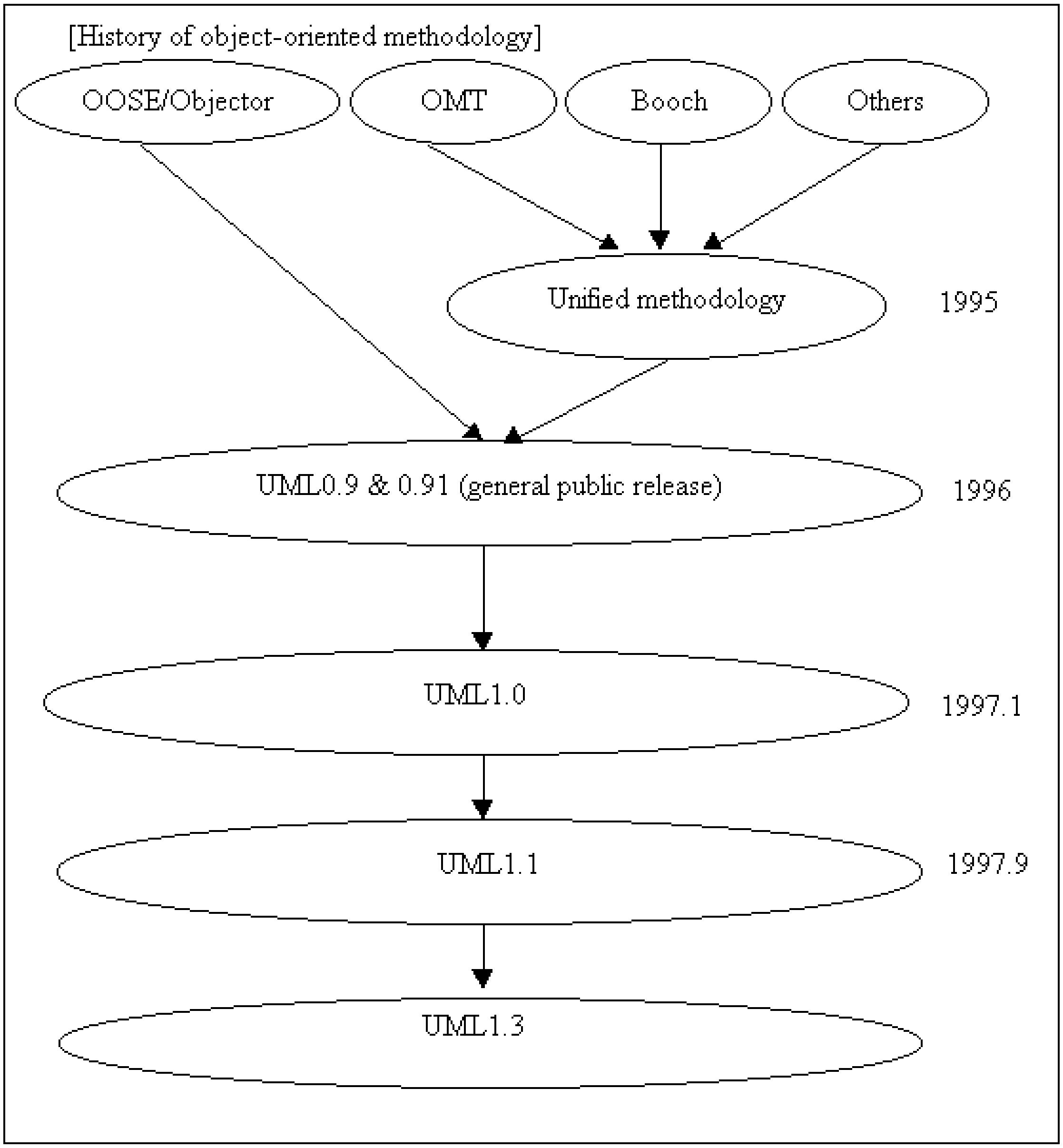
The Shlaer & Mellor method is an analysis method proposed by S. Shlaer and S. Mellor in 1988 and 1992. This analysis method uses an information model as notation method for a model. This method is characterized by its close relationship with conventional modeling and structured techniques.

7.1.5 Catalysis method

The catalysis method is an analysis method proposed by Desmond D'Souza of ICON Computing, Inc. and Alan Cameron Wills of Trireme International, Ltd.. This method adopts the component-oriented paradigm and framework. Therefore, it is applicable for systems with a scope covering up the entire system of an enterprise. This method enables seamless process execution in the system from design to implementation, when the system is implemented using distributed object techniques, common object request broker architecture (CORBA) and the distributed component object model (DCOM).

[Reference: Progress of object-oriented methodology]
Various object-oriented methodologies have been proposed since 1972 when Smalltalk was developed. Since the methodologies varied depending on purposes, a unified method was created later on by combining useful parts, aiming at global standardization. As a result of these efforts, the unified modeling language (UML) was developed as a worldwide standard of the model notation for the object-oriented paradigm.

The following figure illustrates the development of the object-oriented methodology.



This figure shows the transition in the development of the object-oriented methodology. The OMT method, Booch method, and so on were developed as the first object-oriented methodologies. However, since there were differences between the respective model descriptions and types, and an attempt was made to unify those methodologies. As a result, a unified methodology was created in 1995. However, it was deemed impossible to standardize methodologies globally, and only the model notation method, UML, has become as a worldwide standard.

7.2 Distributed Objects

The term distributed object means an object which exists on a network. This section explains how to use distributed objects, using CORBA, RMI, and DCOM as examples.

7.2.1 CORBA

CORBA (Common Object Request Broker Architecture) is a set of specifications used to mutually transfer information between objects running on different computers on the network. CORBA supplies an infrastructure for calling an object (distributed object) on a network from a client. The following are characteristics of CORBA.

(1) Portability

Portability means that an application configured on a specific CORBA can be ported to another CORBA without modification.

(2) Interoperability

Interoperability means that applications on different CORBAs can be connected to one another using the common protocols named "General Inter-ORB Protocol (GIOP)" and "Internet Inter-ORB Protocol (IIOP)."

7.2.2 RMI

RMI (Remote Method Invocation) means a mechanism that enables communication between Java programs via a network. Using RMI, information can be transferred between distributed applications on the network. Therefore, one can configure an environment in which the entire network can be used as one virtual computer.

7.2.3 DCOM

DCOM (Distributed Component Object Model) is an extended version of COM (Component Object Model), an object-to-object communication protocol developed by Microsoft for the use on networks.

7.3 Languages

The programming languages for the object-oriented paradigm are called object-oriented languages. Object-oriented languages are an extended version of languages that support the abstract data type for data abstraction. This section explains the typical object-oriented languages, Smalltalk, C++, and Java.

7.3.1 Smalltalk

Smalltalk is an original programming language that is based on the object-oriented paradigm developed by Xerox, in the U.S.A. in 1972. This language is not only a development language, but also a comprehensive language covering all operating system (OS) functions, language functions, and application functions. Smalltalk is an interactive language that was greatly influenced by the functional programming language Lisp. Smalltalk has the following three characteristics:
(1) Simple syntax
(2) No type concept
(3) Modeling completely with objects

7.3.2 C++

C++ is the result of adding object-oriented functions to C. C++, which is based on C, can be used on all systems for which C is suitable. It is therefore an object-oriented language with a wide range of applications. C++ has the following three characteristics:

(1) Strong typing
In C++, objects that are included in one another must have the same type. In other words, if an object of a specific type is included in an object of another type, an error occurs.

(2) Concept of classes
C++ is a version of C that supports the object-oriented paradigm. Therefore, C++ includes the concept of classes.

(3) Multiple inheritance is possible.
C++ enables multiple inheritance, where data is inherited from multiple base classes during the implementation of classes. However, frequent use of multiple inheritance may make inheritance relations ambiguous.

7.3.3 Java

Java is a pure object-oriented language developed by Sun Microsystems. All programs are described by defining classes. Java provides various functions with a simple syntax. Unlike C++, Java disregards compatibility with other programming languages, and is therefore an object-oriented language with a high level of completeness. Java has the following four characteristics:

(1) Java is a pure object-oriented language.
Since Java disregards compatibility with other programming languages, it is purely an object-oriented language.

(2) Java has an extensive class library
Most Java standard functions are already provided in the library.

(3) Java is platform-independent
During compilation, a Java program is converted to byte code which can be executed on all platforms that support Java. Byte code means that Java programs are written in the form of byte-type binary codes that are transformed into the executable code for the respective machine.

(4) Java assures safety
Java programs do not have the concept of pointers and Java's byte code is strongly typed. For this reason, programs can be verified before execution. This assures safety in execution.

7.4 Object-Oriented Databases

The relational database proposed in 1970s was widely adopted because of the useful features it provides. It has the following three characteristics:

(1) Simple models
- A relational database is a set of real-world objects that are expressed as a table.

(2) Data independency
- In a relational database, the model description is independent of the physical implementation method.
- In a relational database, data is independent of the logical structure.

(3) Non-procedural inquiry language
- In a relational database, the required data is defined, not the procedure for data handling.

However, relational databases have the following disadvantages:

(1) A relational database does not have sufficient capability for modeling extensive parts of the real world.
- The available data types and data structures for a relational database are limited.
- In a relational database, it is difficult to express the relationship between data items.

(2) Program development and management environment are weak.
- Relational databases do not account for a harmonic relationship between programming and database languages.
- Relational databases does not have strong part creation and encapsulation functions.

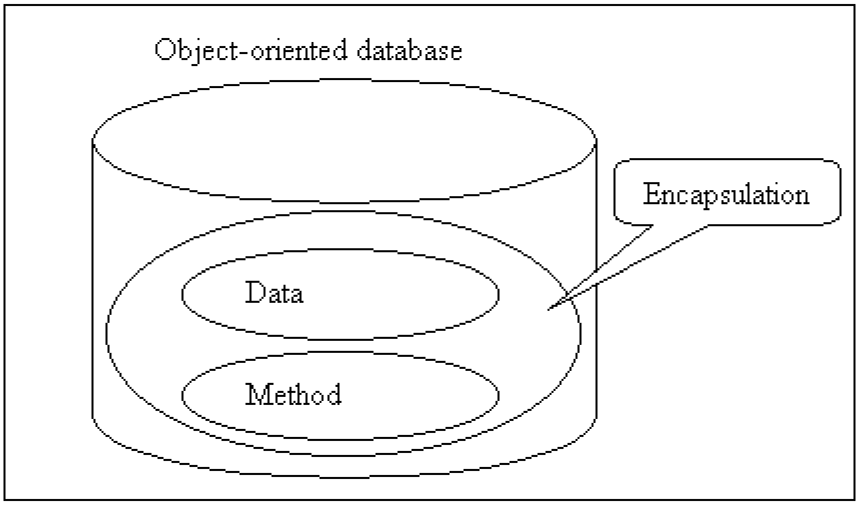
To solve these problems, object-oriented databases were developed. This section explains object-oriented databases.

7.4.1 OODB

OODB (object-oriented database) is one of the new generation databases. Whereas a relational database stores numeric values and characters as data, OODB stores objects. The mechanism that enables the addition, deletion, and update of objects is called object-oriented database management system (OODBMS). The set of data items collected by the OODBMS is called the object-oriented database.

In an object, data and a operating procedures are encapsulated.

The following figure shows an example of an object in an object-oriented database.



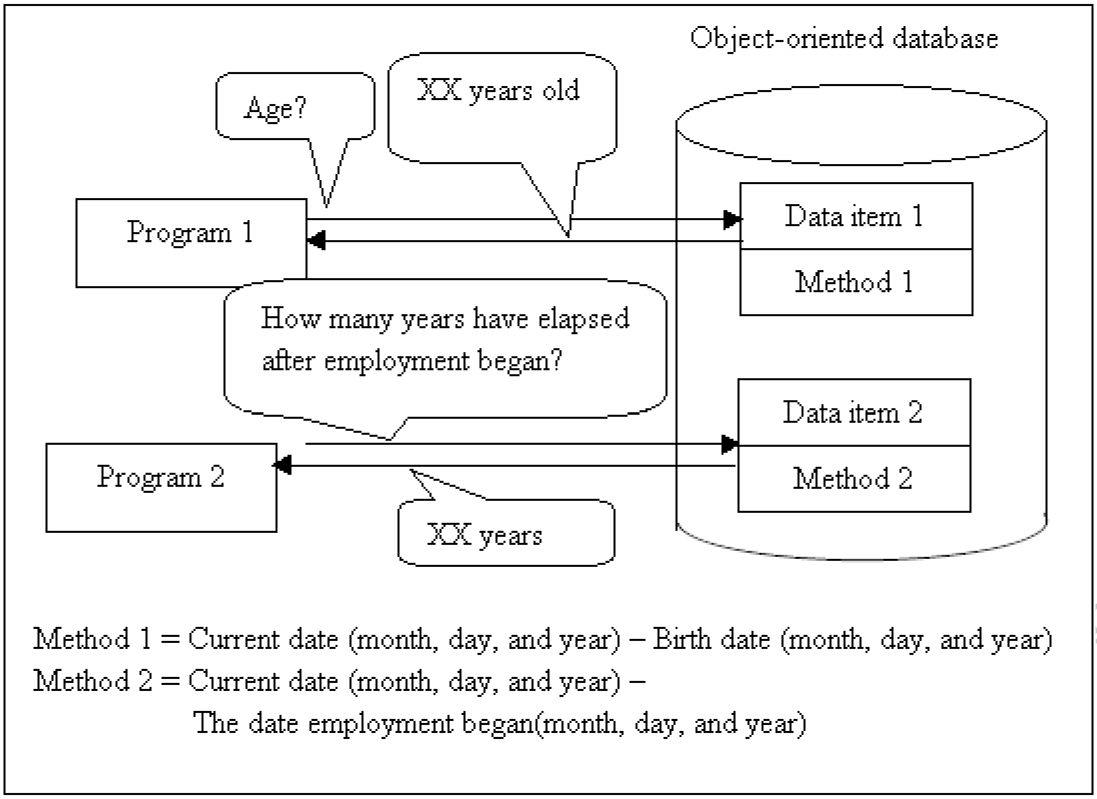
[Example of object]

The data in this figure includes still images, animations, diagrams, and voice data in addition to numeric values and characters.

Since OODB provides many different data structures, the system can handle complicated data structures that would be difficult to handle with conventional databases. OODB enables mixture of multiple data structures in a database.

In OODB, as shown in this figure, data and method are encapsulated, which means that a value that would have to be obtained by calculation in a conventional system can be obtained by a method.

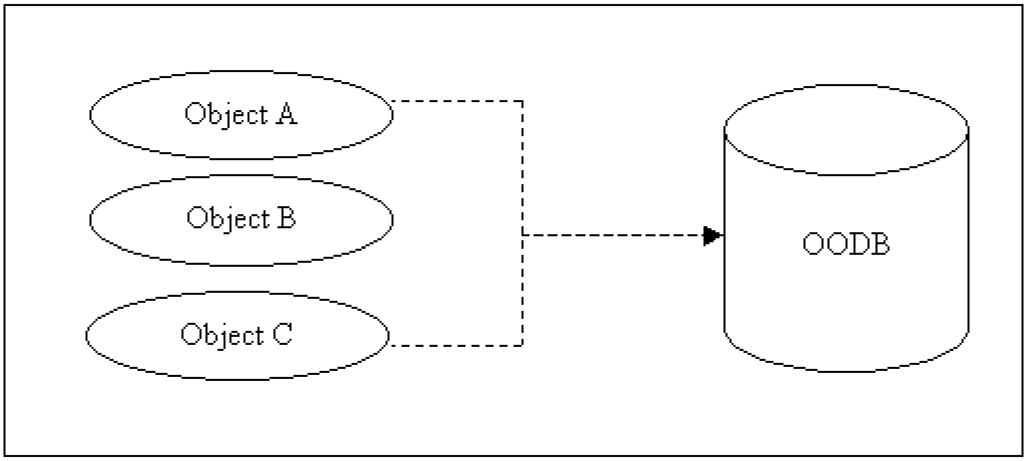
The following figure shows how to obtain data using a method.


Object-oriented database
Age?
XX years old
Program 1
Data item 1
Method 1
How many years have elapsed after employment began?
Data item 2
Method 2
Program 2
XX years
Method 1 = Current date (month, day, and year) – Birth date (month, day, and year)
Method 2 = Current date (month, day, and year) –
The date employment began(month, day, and year)

[Obtaining a value using a method]

Another characteristic of OODB is that the user does not have to recognize data reading and writing operations. This is because the used object is stored in a database by OODBMS after the program terminates. An object automatically stored in a database is called a persistent object, and an object not stored in a database is called a temporary object.

The following figure shows the relationship between persistent and temporary objects.

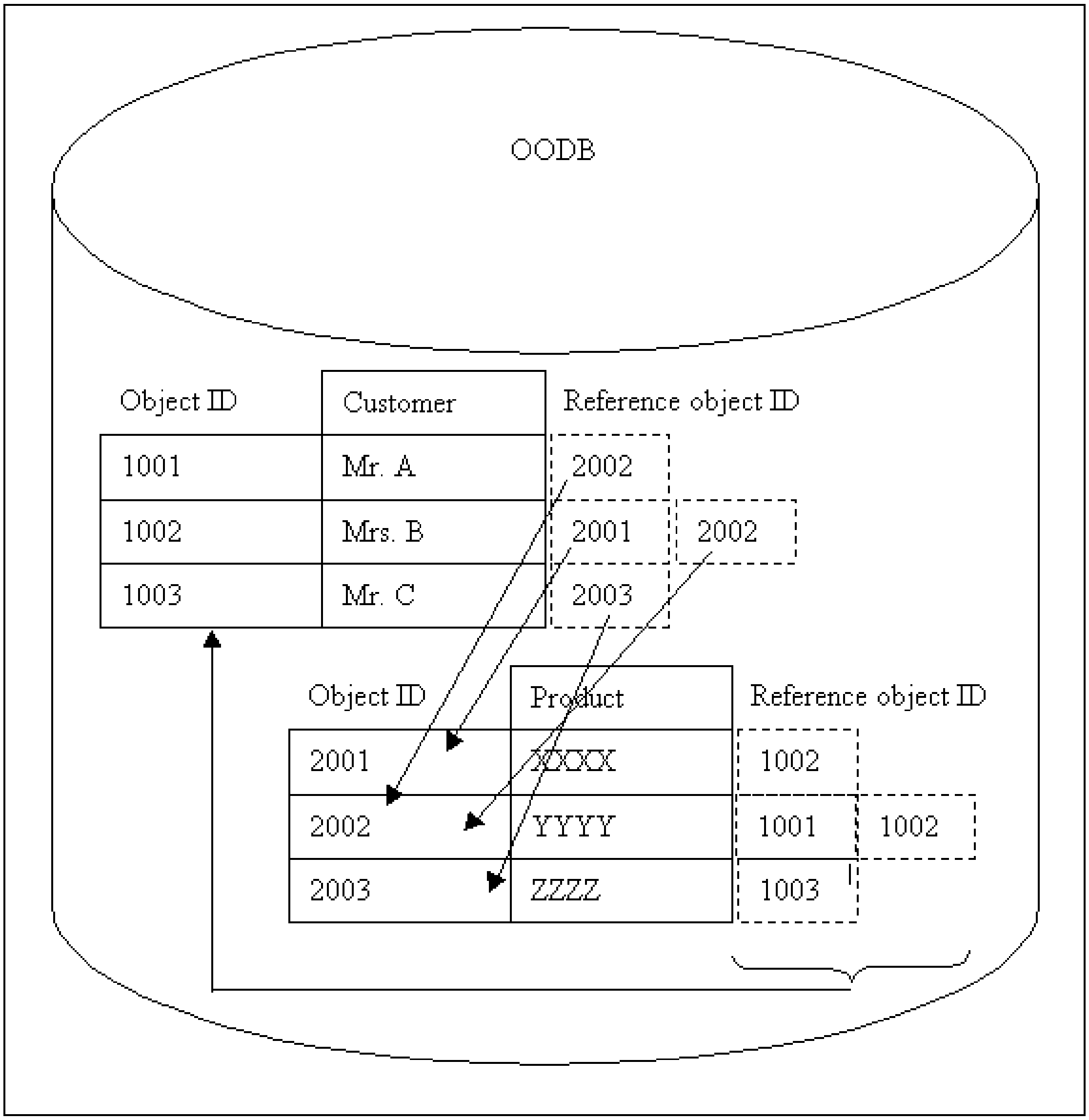


[Persistent and temporary objects]

In this figure, objects A and C are persistent objects, and object B is a temporary object.

The data read operations of OODBMS are explained in the following. Objects stored in OODB are related to one another by the respective object IDs. In this case, one object can have multiple reference object Ids. In other words, relationships of n : m can be reflected in the database. Since the reference object IDs are assigned by OODBMS, the user does not have to consider them.

The figure below shows the data read operation of OODBMS:



Object ID	Customer	Reference object ID	
1001	Mr. A	2002	
1002	Mrs. B	2001	2002
1003	Mr. C	2003	

Object ID	Product	Reference object ID	
2001	XXXX	1002	
2002	YYYY	1001	1002
2003	ZZZZ	1003	

[Data read operation of OODBMS]

The characteristics of OODB are summarized in the following:

(1) OODB can store data that cannot be stored in conventional databases. For example, still images and sound data, in addition to numeric values and characters.

(2) Since OODB enables handling of various data structures, multiple data structures can be mixed in a database.

(3) In OODB, data and method are encapsulated, and data can therefore be obtained using an embedded method.

(4) In OODB, the user does not have to consider data read and write operations.

Exercises

1. The following statement is about the OMT method in the object-oriented methodology. Fill in the appropriate term by selecting from the set of choices below.

The OMT method is a methodology that [(1)] the range from [(2)] to the [(3)] and [(4)] in software development based on the object-oriented paradigm. It has two features: 1) modeling the relation of [(5)] on the basis of [(1)] and 2) [(6)] the development process.

Answers

a. analysis process
b. data structures
c. design
d. implementation
e. reflecting
f. support

2. In the following statement of CORBA characteristics for distributed objects, fill in the appropriate term.

- [(1)] means that an application configured on a specific CORBA can be ported to another CORBA without modification.
- [(2)] means that applications on different CORBAs can be connected to one another using the common protocols named "General Inter-ORB Protocol (GIOP)" and "Internet Inter-ORB Protocol (IIOP)."

3. Fill in the appropriate word(s) in the following statement about RMI in distributed objects.

RMI (remote method invocation) means the specifications used to enable [(1)] between Java programs via [(2)].

4. In the following statement about the object-oriented language, Smalltalk, fill in the appropriate phrase by selecting from the set of choices below.

Smalltalk is an original program language based on the object-oriented paradigm developed by [(1)], in the U.S.A. in 1972. Smalltalk is an [(2)] that was greatly influenced by the [(3)], Lisp.

Answers

a. interactive language
b. Xerox
c. functional language

5. In the following statement about C++ as an object-oriented language, fill in the appropriate phrase from the set of choices below.

C++ is the result of adding [(1)] facilities to [(2)]. C++, which is based on [(2)], can be used on all systems for which [(2)] is suitable. It is therefore an [(1)] language with a wide [(3)].

Answers

a. range of applications
b. C language
c. object-oriented

6. In the following description about Java as an object-oriented language, fill in the appropriate term by selecting from the set of choices below.

Java is a [(1)] object-oriented language developed by [(2)]. All programs are described by defining [(3)]. Java supplies various functions with a simple syntax. Unlike C++, disregards [(4)] with other programming languages, and is therefore an [(5)] with a high [(6)].

Answers

a. compatibility
b. object-oriented language
c. Level of completeness
d. Sun Microsystems
e. class
f. pure

7. In the following statement about object-oriented databases, fill in the appropriate phrase from the set of choices below.

OODB (object-oriented database), is a [(1)][(2)]. Whereas a relational database stores [(3)] and [(4)] as data, OODB stores [(5)] only. The mechanism that enables the addition, deletion, and update of objects in this OODB is called [(6)]. The set of data items collected by the OODBMS is called the [(2)].

Answers

a. object-oriented database
b. objects
c. new generation
d. characters
e. numeric values
f. object-oriented database management system

8. Fill in the appropriate words in the following explanation about the characteristics of object-oriented databases.

- OODB can store data that cannot be stored in conventional databases. For example, [(1)] and [(2)], in addition to numeric values and characters.
- Since OODB enables handling of various [(3)], multiple [(3)] can be mixed in a database.
- In OODB, data and [(4)] are [(5)], and data can therefore be obtained using an built-in [(4)].
- In OODB, the user does not have to [(6)] data read and write operations.

9. In the following statement about the relationship between objects in object-oriented database, fill in the appropriate phrase by selecting from the set of choices below.

Objects stored in OODB are related to one another by the respective [(1)]. In this case, one object can have multiple [(2)]. In other words, relationships of [(3)] can be reflected in the database.

Answers

a. object IDs

b. n : m

c. reference object IDs

10. In the following statement about data read and write operations in object-oriented databases, fill in the appropriate words by selecting from the set of choices below.

In [(1)], the used object is stored in a database by OODBMS after [(2)] terminates. An object automatically stored in a database is called [(3)], and an object not stored in a database is called [(4)].

Answers

a. the program

b. a temporary object

c. an object-oriented database

d. a persistent object

Answers for No.4 Exercises

Answers for Chapter 1 (Basic Concepts of Object Oriemtation)

1. (1) d (2) f (3) e (4) b (5) a

2. (1) c (2) d (3) a (4) e (5) b

3. (4)

4. Abstraction considers the common characteristics of objects. Duplicated development can be avoided by using this approach in the initial step of the system development process.

5. (1) e (2) c (3) b (4) d

6. (1) b (2) d

7. (3)

8. (1) c (2) f (3) b (4) e (5) a (6) h (7) g (8) d

9. (1), (4)

10. (1) e (2) h (3) g (4) c (5) b (6) f (7) d (8) a

11. (1) data (2) operated (4) change (4) (application) program

12. (1) instance (2) an object (3) data (4) class

13. (1) b (2) c (3) d (4) a

14. (1) temporary (2) uses

15. other association

16. (1) c (2) d (3) a (4) b

17. (1) d (2) b (3) c (4) e (5) a

18. object

19. (1) d (2) b (3) a

20. The is-a association is a parent-child relationship between classes.

21. (1) b (2) c (3) d (4) a

22. The part-a association is a parent-child relationship between objects.

23. (1) part object (2) Whole object (3) dependence relationship (4) multiplicity

24. (1) c (2) b (3) e (4) a (5) f

25. (1) e (2) b (3) d (4) c (5) a

Answers for Chapter 2 (UML)

1. (5) Customer and order receptionist

2. (3) Purchase application registration

3. (1) Clarifies relationships between system provided functions and their users.

4. (3) Superclass and subclass

5. (1) The type of attribute b is c.

6. (3) Shows the physical file partitioning of a class.

7. (2) b

8. (3) Shows the collaboration of objects for a scenario.

9. (3)

10. (3) c: Event d: Guard condition e: Action

11. (4) Network node

12. (4)

13. (3) Component diagram

Answers for Chapter 3 (Object-oriented Development Process Overview)

1. (2)

2. (1)

3. (3)

4. (1) b (2) d (3) a (4) c

Answers for Chapter 4 (Analysis)

1. (1) b, (2) a, (3) c, (4) d

2. (1) h, (2) b, (3) f, (4) a, (5) c, (6) d, (7) e

3. (1) d, (2) e, (3) b, (4) a

4. (1) c, (2) e, (3) h, (4) a, (5) f

5. <<include>>

6. (1) c, (2) b, (3) g, (4) a, (5) d, (6) f, (7) h, (8) e

7. (1) internal mechanism, (2) functions, (3) designer, (4) developer

8. (1) a, (2) d, (3) b, (4) e, (5) c, (6) g

9. (1) d, (2) b, (3) c, (4) e

10. (1) computers, (2) devices, (3) designer

11. (1) Interview, (2) opinions, (3) recheck

12. (1) e, (2) c, (3) b, (4) e, (5) f, (6) d, (7) g, (8) a

13. (1) e, (2) c, (3) d, (4) f, (5) b, (6) a

14. (1) c, (2) a, (3) d, (4) e, (5) b ((4) and (5) may be interchanged)

15. (1) user interface, (2) exception flow

16. (1) e, (2) c, (3) b, (4) a

17. (1) interaction, (2) relationships, (3) layout

18. (1) d, (2) a, (3) b, (4) c

19. (1) functions, (2) job, (3) schematic representation, (4) problems

20. (1) b, (2) e, (3) a, (4) d, (5) c, (6) f

Answers for Chapter 5 (Design)

1. (3)

Answers for Chapter 6 (Implementation)

(No questions for this chapter)

Answers for Chapter 7 (Major Object-Oriented Techniques)

1. (1) f (2) a (3) c (4) d (5) b (6) e

2. (1) portability (2) interoperability

3. (1) communication (2) network

4. (1) b (2) a (3) c

5. (1) c (2) b (3) a

6. (1) f (2) d (3) e (4) a (5) b (6) c

7. (1) c (2) a (3) e (4) d (5) b (6) f

8. (1) still image (2) sound (3) data structures (4) methods (5) encapsulated (6) consider

9. (1) a (2) c (3) b

10. (1) c (2) a (3) d (4) b

Index