Object Oriented Concepts and Principles (Chapter 9)

There are many ways to look at a problem to be solved using a software-based solution. One widely used approach to problem solving takes an object-oriented viewpoint. The problem domain is characterized as a set of objects that have specific attributes and behaviors. The objects are manipulated with a collection of functions (called methods, operations, or services) and communicate with one another through a messaging protocol. Objects are categorized into classes and subclasses.

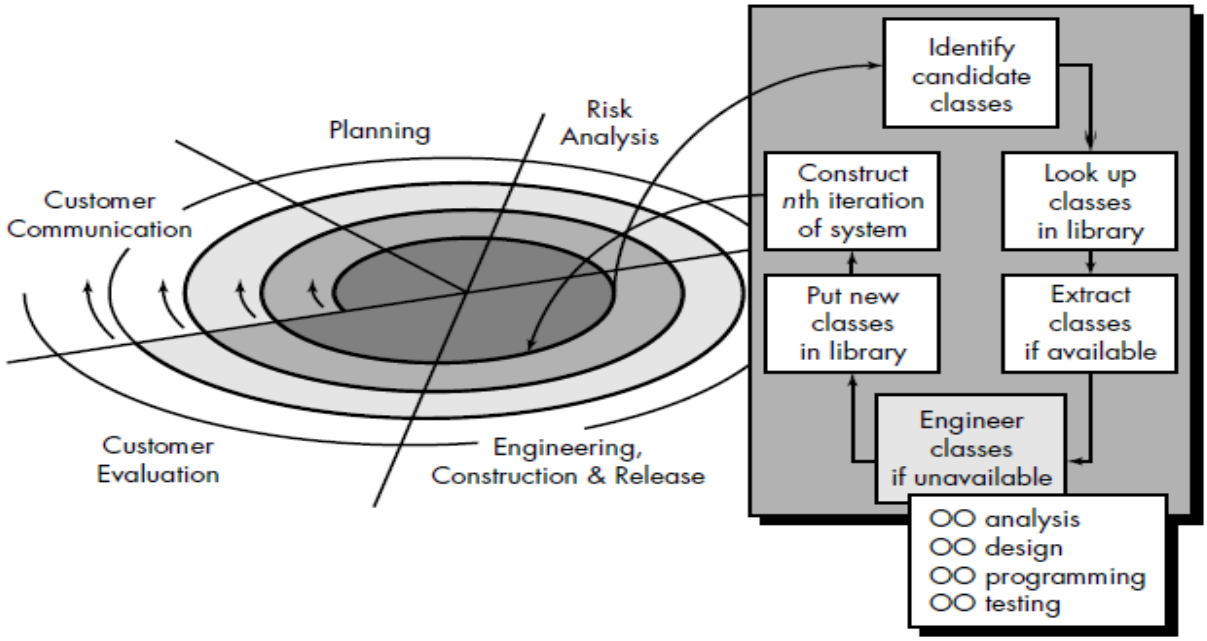
Object-oriented software engineering follows the same steps as conventional approaches. Analysis identifies objects and classes that are relevant to the problem domain; design provides the architecture, interface, and component- level detail; implementation (using an object-oriented language) transforms design into code; and testing exercises the object-oriented architecture, interfaces and components.
Object technologies lead to reuse, and reuse (of program components) leads to faster software development and higher-quality programs. Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer. In addition, object-oriented systems are easier to adapt and easier to scale (i.e., large systems can be created by assembling reusable subsystems).

THE OBJECT-ORIENTED PARADIGM

OO paradigm encompasses a complete view of software engineering. The benefits of object-oriented technology are enhanced if it is addressed early-on and throughout the software engineering process. Those considering object-oriented technology must assess its impact on the entire software engineering process. Merely employing object-oriented programming (OOP) will not yield the best results. Software engineers and their managers must consider such items as object-oriented requirements analysis (OORA), object-oriented design (OOD), object-oriented domain analysis (OODA), object-oriented database systems (OODBMS) and object-oriented computer aided software engineering (OOCASE).

OO systems are engineered using an evolutionary process model. An evolutionary process model, coupled with an approach that encourages component assembly (reuse), is the best paradigm for OO software engineering. Referring to Figure given below, the component-based development process model has been tailored for OO software engineering.
The OO process moves through an evolutionary spiral that starts with customer communication. It is here that the problem domain is defined and that basic problem classes are identified. Planning and risk analysis establish a foundation for the OO project plan. The technical work associated with OO software engineering follows the iterative path shown in the shaded box. OO software engineering emphasizes reuse. Therefore, classes are “looked up” in a library (of existing OO classes) before they are built. When a class cannot be found in the library, the software engineer applies object-oriented analysis (OOA), object-oriented design (OOD), object-oriented programming (OOP), and object-oriented testing (OOT) to create the class and the objects derived from the class. The new class is then put into the library so that it may be reused in the future.

The object-oriented view demands an evolutionary approach to software engineering. As we will see that, it would be exceedingly difficult to define all necessary classes for a major system or product in a single iteration. As the OO analysis and design models evolve, the need for additional classes becomes apparent. It is for this reason that the paradigm just described works best for OO.



☞ OBJECT-ORIENTED CONCEPTS

Object-oriented = objects + classification + inheritance + communication

1. **Classes and Objects:** A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real world entity. An object encapsulates both data (attributes) and the functions (operation, methods, or services) that manipulate the data.

A class is a *generalized description* (e.g., a template, pattern, or blueprint) that describes a collection of similar objects. By definition, all objects that exist within a class inherit its attributes and the operations that are available to manipulate the attributes. A *superclass* is a collection of classes, and a *subclass* is a specialized instance of a class. The data abstractions (attributes) that describe the class are enclosed by a "wall" of procedural abstractions (called *operations, methods,* or *services*) that are capable of manipulating the data in some way. The only way to reach the attributes (and operate on them) is to go through one of the methods that form the wall. Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall). This achieves information hiding and reduces the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, they are cohesive; and because communication occurs only through the methods that make up the "wall," the class tends to be decoupled from other elements of a system. All of these design characteristics lead to high quality software.

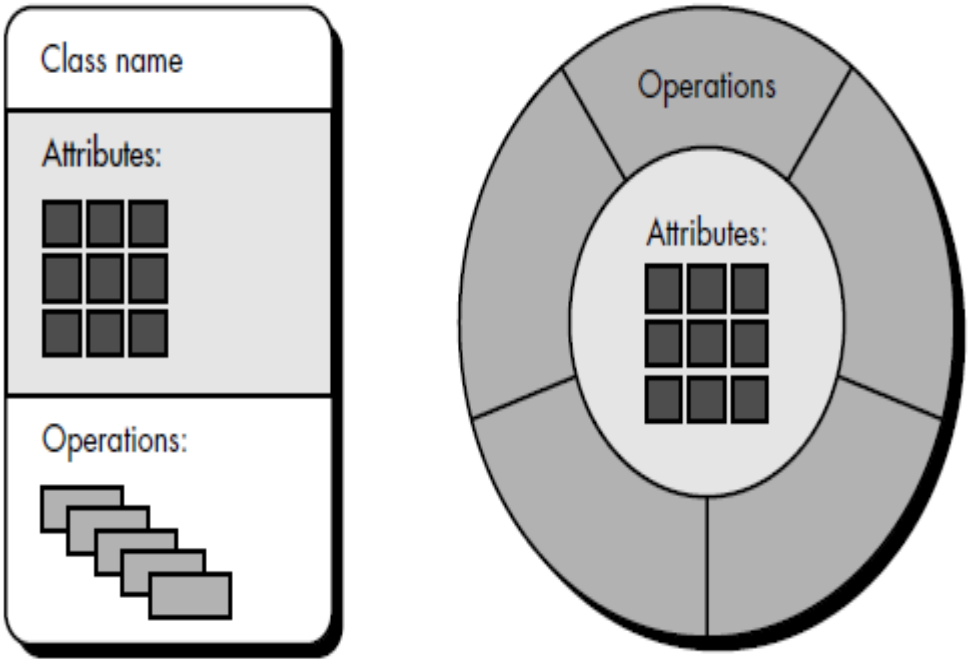


FIGURE An alternative representation of an object-oriented class

2. **Attributes :** Attributes are attached to classes and objects, and that they describe the class or object in some way.

3. Operations, Methods, and Services:

An object encapsulates data (represented as a collection of attributes) and the algorithms that process the data. These algorithms are called operations, methods, or services1 and can be viewed as modules in a conventional sense. Each of the operations that are encapsulated by an object provides a representation of one of the behaviors of the object. For example, the operation *GetColor* for the object **automobile** will extract the color stored in the color attribute. The implication of the existence of this operation is that the class **automobile** has been designed to receive a stimulus (we call the stimulus a *message*) that requests the color of the particular instance of a class. Whenever an object receives a stimulus, it initiates some behavior. This can be as simple as retrieving the color of automobile or as complex as the initiation of a chain of stimuli that are passed among a variety of different objects. In the latter case, consider an example in which the initial stimulus received by object 1 result in the generation of two other stimuli that are sent to object 2 and object 3. Operations encapsulated by the second and third objects act on the stimuli, returning necessary information to the first object. Object 1 then uses the returned information to satisfy the behavior demanded by the initial stimulus.

4. Messages: Messages are the means by which objects interact. A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed. An object is requested to perform one of its operations by sending it a message telling the object what to do. The receiver [object] responds to the message by first choosing the

operation that implements the message name, executing this operation, and then returning control to the caller.
Messages and methods [operations] are two sides of the same coin. Methods are the procedures that are invoked when an object receives a message.
The interaction between messages is illustrated schematically in Figure below. An operation within a sender object generates a message of the form

Message: [destination, operation, parameters]

where *destination* defines the *receiver object* that is stimulated by the message, *operation* refers to the operation that is to receive the message, and *parameters* provides information that is required for the operation to be successful.

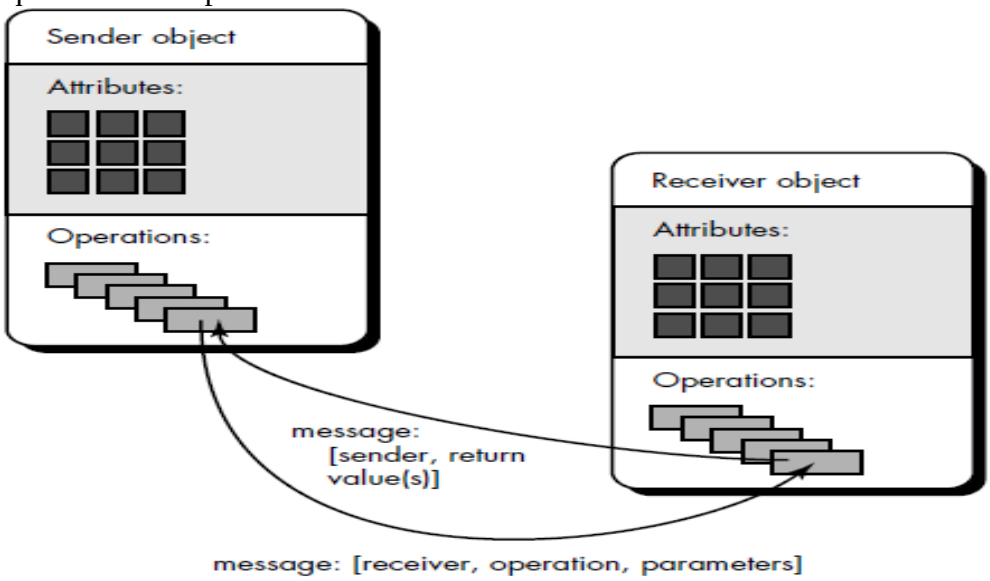


Figure: message passing between objects

Considering the objects shown in figure below an example of message passing within an OO system can be demonstrated. Four objects, **A, B, C,** and **D** communicate with one another by passing messages. For example, if object **B** requires processing associated with operation *op10* of object **D**, it would send **D** a message of the form message: [**D**, *op10*, {data}]
As part of the execution of *op10*, object **D** may send a message to object **C** of the form message: (**C**, *op08*, {data})
Then **C** finds *op08*, performs it, and sends an appropriate return value to **D**. Operation *op10* completes and sends a return value to **B**.

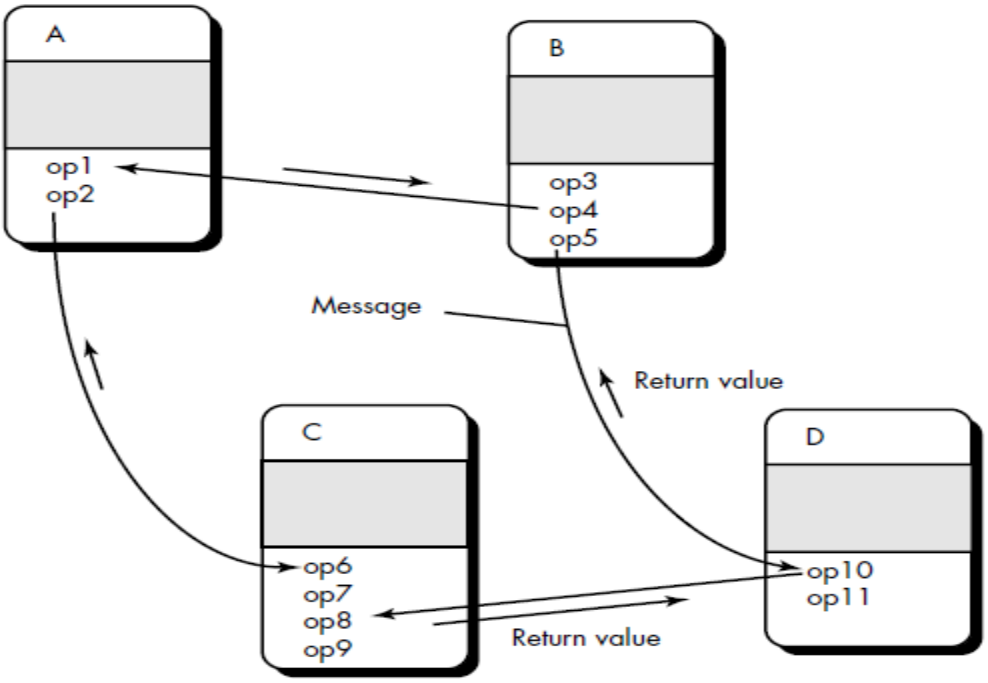


fig: Message passing example

5. Encapsulation, Inheritance, and Polymorphism

Basic three characteristics of object-oriented systems make them unique from structured systems. As we have already noted, the OO class and the objects spawned from the class encapsulate data and the operations that work on the data in a single package. This provides a number of important benefits: (*advantages of Object Oriented Architecture*)

• The internal implementation details of data and procedures are hidden from the outside world (information hiding). This reduces the propagation of side effects when changes occur.

• Data structures and the operations that manipulate them are merged in a single named entity—the class. This facilitates component reuse.

• Interfaces among encapsulated objects are simplified. An object that sends a message need not be concerned with the details of internal data structures. Hence, interfacing is simplified and the system coupling tends to be reduced.

Inheritance is one of the key differentiators between conventional and OO systems.

Inheritance provides a means for allowing subclasses to reuse existing superclass data and procedures; also provides mechanism for propagating changes

A subclass **Y** inherits all of the attributes and operations associated with its superclass, **X**. This means that all data structures and algorithms originally designed and implemented for **X** are immediately available for **Y**—no further work need be done. Reuse has been accomplished directly. Any change to the data or operations contained within a superclass is immediately inherited by all subclasses that have inherited from the superclass. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system. It is important to note that, at each level of the class hierarchy, new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy.

Polymorphism is a characteristic that greatly reduces the effort required to extend an existing OO system.
Polymorphism is a mechanism that allows several objects in a class hierarchy to have different methods with the same name (instances of each subclass will be free to respond to messages by calling their own version of the method)
Polymorphism enables a number of different operations to have the same name. This in turn decouples objects from one another, making each more independent.
The ability of different objects to respond, each in its own way, to identical messages is called *polymorphism*.

Polymorphism results from the fact that every class lives in its own name space. The names assigned within a class definition won't conflict with names assigned anywhere outside it. This is true both of the instance variables in an object's data structure and of the object's methods:

- Just as the fields of a C structure are in a protected name space, so are an object's instance variables.
- Method names are also protected. Unlike the names of C functions, method names aren't global symbols. The name of a method in one class can't conflict with method names in other classes; two very different classes could implement identically named methods.

The main benefit of polymorphism is that it simplifies the programming interface. It permits conventions to be established that can be reused in class after class. Instead of inventing a new name for each new function you add to a program, the same names can be reused.

☞ IDENTIFYING THE ELEMENTS OF AN OBJECT MODEL

The elements of an object model—classes and objects, attributes, operations, and messages were each defined and discussed in the preceding section. But how do we go about identifying these elements for an actual problem? The following sections present a series of informal guidelines that will assist in the identification of the elements of the object model.

1. Identifying Classes and Objects:

We can begin to identify objects by examining the problem statement or by performing a "grammatical parse" on the processing narrative for the system to be built. Objects are determined by underlining each noun or noun clause and entering it in a simple table. Objects can be:
• *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
• *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
• *Occurrences* or *events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
• *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.

• *Organizational units* (e.g., division, group, and team) those are relevant to an application.
• *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
• *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or in the extreme, related classes of objects.

2. Specifying Attributes: Attributes are chosen by examining the problem statement, looking for things that fully define an object and make it unique. Attributes describe an object that has been selected for inclusion in the analysis model.
In essence, it is the attributes that define the object—that clarify what is meant by the object in the context of the problem space. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the object **player** would be quite different than the attributes of the same object when it is used in the context of the professional baseball pension system. In the former, attributes such as name, position, batting average, fielding percentage, years played, and games played might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.
To develop a meaningful set of attributes for an object, the analyst can again study the processing narrative (or statement of scope) for the problem and select those things that reasonably "belong" to the object.

3. Defining Operations: Operations define the behavior of an object and change the object's attributes in some way. More specifically, an operation changes one or more attribute values that are contained within the object. Therefore, an operation must have "knowledge" of the nature of the objects attributes and must be implemented in a manner that enables it to manipulate the data structures that have been derived from the attributes. Although many different types of operations exist, they can generally be divided into three broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, and (3) operations that monitor an object for the occurrence of a controlling event.

4. Finalizing the Object Definition: The definition of operations is the last step in completing the specification of an object. Operations were culled from a grammatical parse of the processing narrative for the system. Additional operations may be determined by considering the "life history" of an object and the messages that are passed among objects defined for the system.
The generic life history of an object can be defined by recognizing that the object must be created, modified, manipulated or read in other ways, and possibly deleted. For the **system** object, this can be expanded to reflect known activities that occur during its life. Some of the operations can be ascertained from likely communication between objects.

☞ MANAGEMENT OF OBJECT-ORIENTED SOFTWARE PROJECTS

OO projects require as much or more management planning and oversight as conventional software projects. Modern software project management can be subdivided into the following activities:

1. Establishing a common process framework for a project.

2. Using the framework and historical metrics to develop effort and time estimates.

Lorenz and Kidd suggest the following set of project metrics:

- **Number of scenario scripts.** A *scenario script* (analogous to use-cases) is a detailed sequence of steps that describe the interaction between the user and the application. Each script is organized into triplets of the form
  {**initiator,** *action,* **participant**}
  where **initiator** is the object that requests some service (that initiates a message); *action* is the result of the request; and **participant** is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.
- **Number of key classes.** *Key classes* are the "highly independent components" that are defined early in OOA. Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.
- **Number of support classes.** *Support classes* are required to implement the system but are not immediately related to the problem domain. Examples might be GUI classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout the recursive/parallel process.
  The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.
- **Average number of support classes per key class:** Key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be much simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.
- **Number of subsystems.** A *subsystem* is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

3. Establishing deliverables and milestones that will enable progress to be measured.
4. Defining checkpoints for risk management, quality assurance, and control.
5. Managing the changes that invariably occur as the project progresses.
6. Tracking, monitoring, and controlling progress.

(Note: For details refer to Book, Software Engineering - Roger S Pressman 5th edition, Chapter 20)