# <u>Object Oriented Analysis and Design</u> (Chapter 10)

*Object-Oriented Analysis* (OOA) is the first technical activity that is performed as part of OO software engineering.
OOA—object-oriented analysis is based upon basic concepts: objects and attributes classes and members, wholes and parts. OOA is grounded in a set of five basic principles applied:
(1) The information domain is modeled; (2) function is described; (3) behavior is represented; (4) data, functional, and behavioral models are partitioned to expose greater detail; and (5) early models represent the essence of the problem while later models provide implementation details.

OOA begins with a description of use-cases—a scenario-based description of how actors (people, machines, other systems) interact with the product to be built. Class-Responsibility-Collaborator (CRC) modeling translates the information contained in use-cases into a representation of classes and their collaborations with other classes. The static and dynamic characteristics of classes are then modeled using a unified modeling language (or some other method).
The intent of OOA is to define all classes that are relevant to the problem to be solved—the operations and attributes associated with them, the relationships between them, and behavior they exhibit. To accomplish this, a number of tasks must occur:
1. Basic user requirements must be communicated between the customer and the software engineer.
2. Classes must be identified (i.e., attributes and methods are defined).
3. A class hierarchy must be specified.
4. Object-to-object relationships (object connections) should be represented.
5. Object behavior must be modeled.
6. Tasks 1 through 5 are reapplied iteratively until the model is complete.

## ☞ OBJECT-ORIENTED ANALYSIS

**The Booch method:** The Booch method (Grady Booch ) encompasses both a “micro development process” and a “macro development process.” The micro level defines a set of analysis tasks that are reapplied for each step in the macro process. Hence, an evolutionary approach is maintained. Booch’s OOA micro development process identifies classes and objects and the semantics of classes and objects and defines relationships among classes and objects and conducts a series of refinements to elaborate the analysis model.
**The Rumbaugh method:** James Rumbaugh and his colleagues developed the *object modeling technique* (OMT) for analysis, system design, and object-level design. The analysis activity creates three models: the object model (a representation of objects, classes, hierarchies, and relationships), the dynamic model (a representation of object and system behavior), and the functional model (a high-level DFD-like representation of information flow through the system).
**The Jacobson method:** Also called OOSE (object-oriented software engineering), the (Ivor Jacobson) method is a simplified version of the proprietary objectory method, also developed by Jacobson. This method is differentiated from others by heavy emphasis on the use-case—a description or scenario that depicts how the user interacts with the product or system.

Over the past decade, Grady Booch, James Rumbaugh, and Ivar Jacobson have collaborated to combine the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the ***Unified Modeling Language*** **(UML)**, has become widely used throughout the industry. UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pragmatic rules.

The syntax tells us how the symbols should look and how the symbols are combined. The syntax is compared to words in natural language; it is important to know how to spell them correctly and how to put different words together to form a sentence. The semantic rules tell us what each symbol means and how it should be interpreted by itself and in the context of other symbols; they are compared to the meanings of words in a natural language. The pragmatic rules define the intentions of the symbols through which the purpose of the model is achieved and becomes understandable for others. This corresponds in natural language to the rules for constructing sentences that are clear and understandable.

In UML, a system is represented using five different "views" that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams. The following views are present in UML:

- **User model view.** This view represents the system (product) from the user's (called *actors* in UML) perspective. The use-case is the modeling approach of choice for the user model view. This important analysis representation describes a usage scenario from the end-user's perspective.
- **Structural model view.** Data and functionality are viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.
- **Behavioral model view.** This part of the analysis model represents the dynamic or behavioral aspects of the system. It also depicts the interactions or collaborations between various structural elements described in the user model and structural model views.
- **Implementation model view.** The structural and behavioral aspects of the system are represented as they are to be built.
- **Environment model view.** The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

## ☞ DOMAIN ANALYSIS

Analysis for object-oriented systems can occur at many different levels of abstraction. OOA at a middle level of abstraction is called as d*omain analysis. Domain analysis* is performed when an organization wants to create a library of reusable classes (components) that will be broadly applicable to an entire category of applications.
Main Objective of domain analysis is to define a set of classes (objects) that are encountered throughout an application domain. These can then be reused in many applications.

### 1. Reuse and Domain Analysis

Object-technologies are leveraged through reuse. *Benefits derived from reuse are consistency and familiarity. Patterns within the software will become more consistent, leading to better maintainability. Be certain to establish a set of reuse design rules" so that these benefits are achieved.*
***Reuse is the cornerstone of component-based software engineering.***

## 2. Domain Analysis Process

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain.

***Object-oriented domain analysis is the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.***

The goal of domain analysis is straightforward: to find or create those classes that are broadly applicable, so that they may be reused.

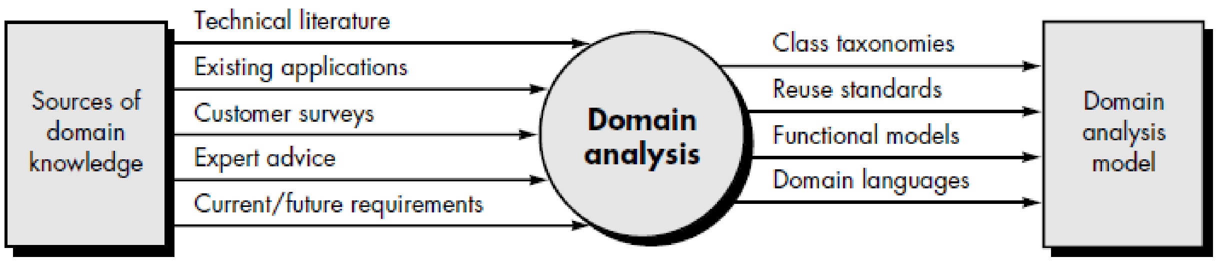


Fig: Input and Output for domain analysis

***Domain analysis may be viewed as an umbrella activity for the software process. By this we mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. The role of the domain analyst is to design and build reusable components that may be used by many people working on similar but not necessarily the same applications.***

Above figure illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain. In essence domain analysis is quite similar to knowledge engineering. The knowledge engineer investigates a specific area of interest in an attempt to extract key facts that may be of use in creating an expert system or artificial neural network. During domain analysis, *object* (and class) *extraction* occurs.

The domain analysis process can be characterized by a series of activities that begin with the identification of the domain to be investigated and end with a specification of the objects and classes that characterize the domain. Berard suggests the following activities:

- **Define the domain to be investigated.** To accomplish this, the analyst must first isolate the business area, system type, or product category of interest. Next, both OO and non-OO "items" must be extracted. OO items include specifications, designs, and code for existing OO application classes; support classes (e.g., GUI classes or database access classes); commercial off-the shelf (COTS) component libraries that are relevant to the domain; and test cases. Non-OO items encompass policies, procedures, plans, standards, and guidelines; parts of existing non-OO applications (including specification, design, and test information); metrics; and COTS non-OO software.
- **Categorize the items extracted from the domain.** The items are organized into categories and the general defining characteristics of the category are defined. A classification scheme for the categories is proposed and naming conventions for each item are defined. When appropriate, classification hierarchies are established.
- **Collect a representative sample of applications in the domain.** To accomplish this activity, the analyst must ensure that the application in question has items that fit into the categories that have already been defined. During the early stages of use of object-

technologies, a software organization will have few if any OO applications. Therefore, the domain analyst must "identify the conceptual (as opposed to physical) objects in each [non-OO] application."

- **Analyze each application in the sample.** The following steps are followed by the analyst:
  • Identify candidate reusable objects.
  • Indicate the reasons that the object has been identified for reuse.
  • Define adaptations to the object that may also be reusable.
  • Estimate the percentage of applications in the domain that might make reuse of the object.
  • Identify the objects by name and use configuration management techniques to control them.
  In addition, once the objects have been defined, the analyst should estimate what percentage of a typical application could be constructed using the reusable objects.

- **Develop an analysis model for the objects.** The analysis model will serve as the basis for design and construction of the domain objects.

## ☞GENERIC COMPONENTS OF THE OO ANALYSIS MODEL

To develop a "precise, concise, understandable, and correct model of the real world," a software engineer must select a notation that implements a set of generic components of an OO analysis model. Monarchi and Puhr define a set of generic representational components that appear in all OO analysis models.

*Static components* are structural in nature and indicate characteristics that hold throughout the operational life of an application. These characteristics distinguish one object from other objects.

*Dynamic components* focus on control and are sensitive to timing and event processing. They define how one object interacts with other objects over time.

***Static components do not change as the application is executed. Dynamic components are influenced by timing and events.***

The following components are identified:

- **Static view of semantic classes.** Requirements are assessed and classes are extracted (and represented) as part of the analysis model. These classes persist throughout the life of the application and are derived based on the semantics of the customer requirements.
- **Static view of attributes.** Every class must be explicitly described. The attributes associated with the class provide a description of the class, as well as a first indication of the operations that are relevant to the class.
- **Static view of relationships.** Objects are "connected" to one another in a variety of ways. The analysis model must represent these relationships so that operations (that affect these connections) can be identified and the design of a messaging approach can be accomplished.
- **Static view of behaviors.** The relationships just noted define a set of behaviors that accommodate the usage scenario (use-cases) of the system. These behaviors are implemented by defining a sequence of operations that achieve them.
- **Dynamic view of communication.** Objects must communicate with one another and do so based on a series of events that cause transition from one state of a system to another.
- **Dynamic view of control and time.** The nature and timing of events that cause transitions among states must be described.

## THE OOA PROCESS

The OOA process does not begin with a concern for objects. Rather, it begins with an understanding of the manner in which the system will be used—by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications. Once the scenario of usage has been defined, the modeling of the software begins. The sections that follow define a series of techniques that may be used to gather basic customer requirements and then define an analysis model for an object oriented system.

### 1. Use Cases

Use-cases are an excellent requirements elicitation tool, regardless of the analysis method that is used.

Use-cases model the system from the end-user's point of view. Created during requirements elicitation, use-cases should achieve the following objectives:

• To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.

• To provide a clear and unambiguous description of how the end-user and the system interact with one another.

• To provide a basis for validation testing.

During OOA, use-cases serve as the basis for the first element of the analysis model.

Using UML notation, a diagrammatic representation of a use-case, called a *use-case diagram*, can be created. Like many elements of the analysis model, the use-case diagram can be represented at many levels of abstraction. The use-case diagram contains actors and use-cases. *Actors* are entities that interact with the system. They can be human users or other machines or systems that have defined interfaces to the software.

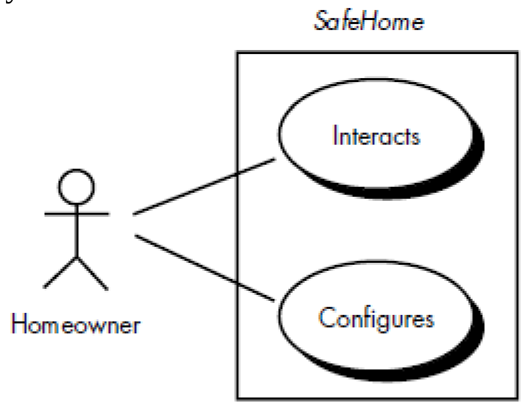
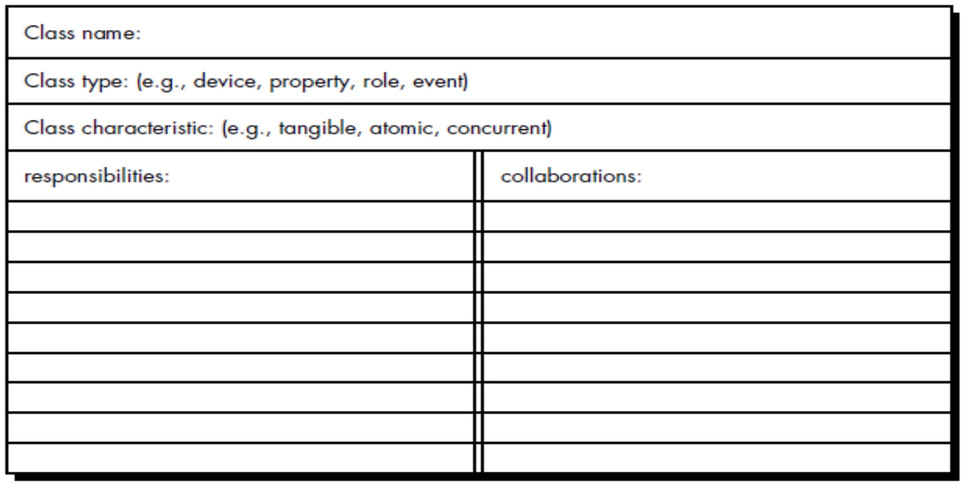


**Fig:** High level Use-case

### 2. Class-Responsibilities-Collaborator Modeling

Once basic usage scenarios have been developed for the system, it is time to identify candidate classes and indicate their responsibilities and collaborations. *Class-Responsibility-Collaborator* (CRC) modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.
CRC model may make use of actual or virtual *index cards.* The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does". *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility. In general, collaboration implies either a request for information or a request for some action.



| | |
|---|---|
| Class name: | |
| Class type: (e.g., device, property, role, event) | |
| Class characteristic: (e.g., tangible, atomic, concurrent) | |
| responsibilities: | collaborations: |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

**Fig:** A CRC model Index card

## 3. Define structure and hierarchies

Once classes and objects have been identified using the CRC model, the analyst begins to focus on the structure of the class model and the resultant hierarchies that arise as classes and subclasses emerge. Using UML notation, a variety of class diagrams can be created. *Generalization/specialization* class structures can be created for identified classes.
Structure representations provide the analyst with a means for partitioning the CRC model and representing that partitioning graphically. The expansion of each class provides needed detail for review and for subsequent design.

## 4. Define Subjects and Subsystem

An analysis model for a complex application may have hundreds of classes and dozens of structures. For this reason, it is necessary to define a concise representation that is a digest of the CRC and structure models just described. When a group of all classes collaborate among themselves to accomplish a set of cohesive responsibilities, they are often referred to as *subsystems* or *packages* (in UML terminology). Subsystems or packages are abstractions that provide a reference or pointer to more detail in the analysis model. When viewed from the outside, a subsystem can be treated as a black box that contains a set of responsibilities and that has its own (outside) collaborators. A subsystem implements one or more *contracts* with its outside collaborators. A contract is a specific list of requests that collaborators can make of the subsystem. Subsystems can be represented with the context of CRC modeling by creating a subsystem index card. The subsystem index card indicates the name of the subsystem, the contracts that the subsystem must accommodate, and the classes or (other) subsystems that support the contract. Packages are identical to subsystems in intent and content but are represented graphically in UML.

## OBJECT-ORIENTED DESIGN

Object-oriented design transforms the analysis model created using object-oriented analysis into a design model that serves as a blueprint for software construction. The design of object oriented software requires the definition of a multilayered software architecture, the specification of subsystems that perform required functions and provide infrastructure support, a description of objects (classes) that form the building blocks of the system, and a description of the communication mechanisms that allow data to flow between layers, subsystems, and objects. Object-oriented design accomplishes all of these things.
OOD is divided into two major activities: system design and object design. System design creates the product architecture, defining a series of “layers” that accomplish specific system functions and identifying the classes that are encapsulated by subsystems that reside at each layer. In addition, system design considers the specification of three components: the user interface, data management functions, and task management facilities. Object design focuses on the internal detail of individual classes, defining attributes, operations, and message detail.
An OO design model encompasses software architecture, user interface description, data management components, task management facilities, and detailed descriptions of each class to be used in the system.

## DESIGN FOR OBJECT-ORIENTED SYSTEMS

For object-oriented systems, we can also define a design pyramid, the four layers of the OO design pyramid are:
**The subsystem layer** contains a representation of each of the subsystems that enable the software to achieve its customer-defined requirements and to implement the technical infrastructure that supports customer requirements.
**The class and object layer** contains the class hierarchies that enable the system to be created using generalizations and increasingly more targeted specializations. This layer also contains representations of each object.
**The message layer** contains the design details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.
**The responsibilities layer** contains the data structure and algorithmic design for all attributes and operations for each object.

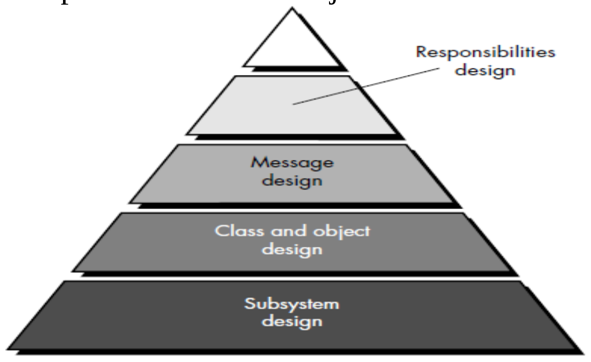


**Fig:** OO Design Pyramid

The design pyramid focuses exclusively on the design of a specific product or system. It should be noted, however, that another "layer" of design exists, and this layer forms the foundation on which the pyramid rests. The foundation layer focuses on the design of *domain objects* (called *design patterns)*. Domain objects play a key role in building the infrastructure for the OO system by providing support for human/computer interface activities, task management, and data management. Domain objects can also be used to flesh out the design of the application itself.

## ☞ Translating an OOA model into an OOD model

Figure Below illustrates the relationship between the OO analysis model and design model that will be derived from it. The subsystem design is derived by considering overall customer requirements (represented with use-cases) and the events and states that are externally observable (the object-behavior model). Class and object design is mapped from the description of attributes, operations, and collaborations contained in the CRC model. Message design is driven by the object-relationship model, and responsibilities design is derived using the attributes, operations, and collaborations described in the CRC model.

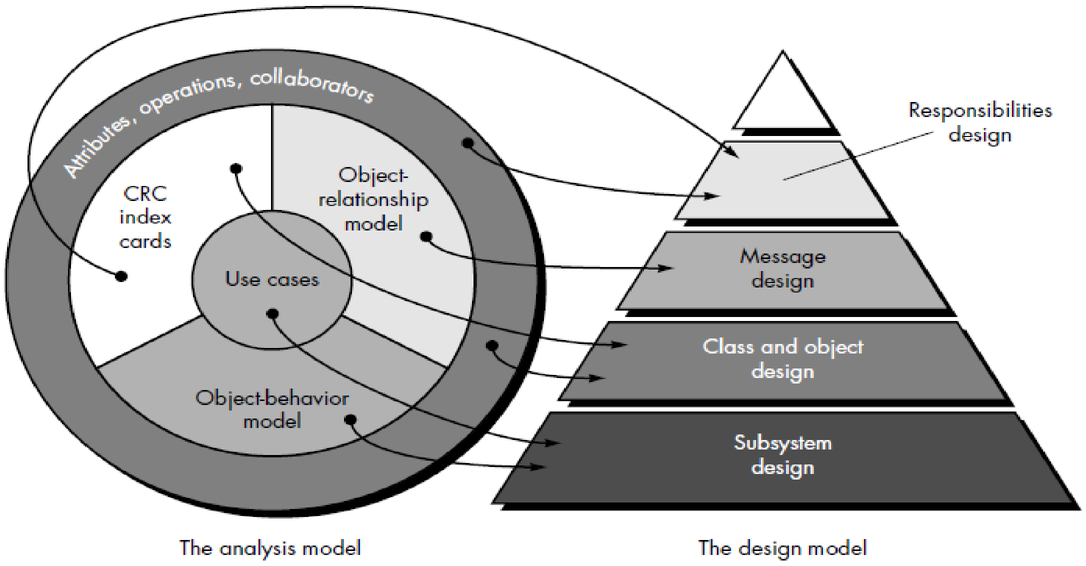


Fig: Translating OOA to OOD

## ☞ THE SYSTEM DESIGN PROCESS

## ☞ THE OBJECT DESIGN PROCESS

Object design is concerned with the detailed design of the objects and their interactions. It is completed within the overall architecture defined during system design and according to agreed design guidelines and protocols. Object design is particularly concerned with the specification of attribute types, how operations function, and how objects are linked to other objects.
It is at this stage that the basic concepts and principles associated with component level design come into play. Local data structures are defined (for attributes) and algorithms (for operations) are designed.

### 1. Object Descriptions

A design description of an object (an instance of a class or subclass) can take one of two forms:
(1) P*rotocol description* that establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs when it receives the message or
(2) I*mplementation description* that shows implementation details for each operation implied by a message that is passed to an object. Implementation details include information about the object's private part; that is, internal details about the data structures that describe the object’s attributes and procedural details that describe operations. The protocol description is nothing more than a set of messages and a corresponding comment for each message.

### 2. Designing Algorithms and Data Structures

A variety of representations contained in the analysis model and the system design provide a specification for all operations and attributes. Algorithms and data structures are designed using an approach that differs little from the data design and component-level design approaches discussed for conventional software engineering. An algorithm is created to implement the specification for each operation. In many cases, the algorithm is a simple computational or procedural sequence that can be implemented as a self-contained software module. However, if the specification of the operation is complex, it may be necessary to modularize the operation. Conventional component-level design techniques can be used to accomplish this. Data structures are designed concurrently with algorithms. Since operations invariably manipulate the attributes of a class, the design of the data structures that best reflect the attributes will have a strong bearing on the algorithmic design of the corresponding operations.
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.

### 3. Program Components and Interfaces

An important aspect of software design quality is modularity; that is, the specification of program components (modules) that are combined to form a complete program. The object-oriented approach defines the object as a program component that is itself linked to other components (e.g., private data, operations). But defining objects and operations is not enough. During design, we must also identify the interfaces between objects and the overall structure (considered in an architectural sense) of the objects. Although a program component is a design abstraction, it should be represented in the context of the programming language used for implementation. To accommodate OOD, the programming language to be used for implementation should be capable of creating the following program component.

## ☞ DESIGN PATTERNS

Design patterns allow the designer to create the system architecture by integrating reusable components
The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution.
There are many recurring patterns of classes and communicating objects in many object oriented systems. These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.
Throughout the OOD process, a software engineer should look for every opportunity to reuse existing design patterns (when they meet the needs of the design) rather than creating new ones.

### 1. Describing a Design Pattern

Mature engineering disciplines make use of thousands of design patterns. For example, a mechanical engineer uses a two-step, keyed shaft as a design pattern. Inherent in the pattern are attributes (the diameters of the shaft, the dimensions of the keyway, etc.) and operations (e.g., shaft rotation, shaft connection). An electrical engineer uses an integrated circuit (an extremely complex design pattern) to solve a specific element of a new problem. All design patterns can be described by specifying the following information:
• the name of the pattern
• the intent of the pattern
• the "design forces" that motivate the pattern
• the solution that mitigates these forces
• the classes that are required to implement the solution
• the responsibilities and collaboration among solution classes
• guidance that leads to effective implementation
• example source code or source code templates
• cross-references to related design patterns
The design pattern name is itself an abstraction that conveys significant meaning once the applicability and intent are understood. *Design forces* describe the data, functional, or behavioral requirements associated with part of the software for which the pattern is to be applied. In addition forces define the constraints that may restrict the manner in which the design is to be derived. In essence, design forces describe the environment and conditions that must exist to make the design pattern applicable. The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems. These attributes represent characteristics of the design that can be searched (e.g., via a database) so that an appropriate pattern can be found. Finally, guidance associated with the use of a design pattern provides an indication of the ramifications of design decisions.

### 2. Using Patterns in Design

In an object-oriented system, design patterns10 can be used by applying two different mechanisms: inheritance and composition. Inheritance is a fundamental OO concept. Using inheritance, an existing design pattern becomes a template for a new subclass. The attributes and operations that exist in the pattern become part of the subclass.
*Composition* is a concept that leads to aggregate objects. That is, a problem may require objects that have complex functionality (in the extreme, a subsystem accomplishes this). The complex object can be assembled by selecting a set of design patterns and composing the appropriate object (or subsystem). Each design pattern is treated as a black box, and communication among the patterns occurs only via well-defined interfaces.

Software Engineers suggest that object composition should be favored over inheritance when both options exist. Rather than creating large and sometimes unmanageable class hierarchies (the consequence of the overuse of inheritance), composition favors small class hierarchies and objects that remain focused on one objective. Composition uses existing design patterns (reusable components) in an unaltered form.