After the analysis phase, the conceptual model is developed further into an object-oriented model using object-oriented design (OOD). In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. In a nutshell, a detailed description is constructed specifying how the system is to be built on concrete technologies.

The stages for object-oriented design can be identified as:

- Definition of the context of the system
- Designing system architecture
- Identification of the objects in the system
- Construction of design models
- Specification of object interfaces

### Object-Oriented System Design

#### System Design

Object-oriented system design involves defining the context of a system followed by designing the architecture of the system.

- **Context**: The context of a system has a static and a dynamic part. The static context of the system is designed using a simple block diagram of the whole system which is expanded into a hierarchy of subsystems. The subsystem model is represented by UML packages. The dynamic context describes how the system interacts with its environment. It is modelled using **use case diagrams**.

- **System Architecture**: The system architecture is designed on the basis of the context of the system in accordance with the principles of architectural design as well as domain knowledge. Typically, a system is partitioned into layers and each layer is decomposed to form the subsystems.

#### Object-Oriented Decomposition

Decomposition means dividing a large complex system into a hierarchy of smaller components with lesser complexities, on the principles of divide-and-conquer. Each major component of the system is called a subsystem. Object-oriented decomposition identifies individual autonomous objects in a system and the communication among these objects.

The advantages of decomposition are:

- The individual components are of lesser complexity, and so more understandable and manageable.
- It enables division of workforce having specialized skills.
- It allows subsystems to be replaced or modified without affecting other subsystems.

#### Identifying Concurrency

Concurrency allows more than one objects to receive events at the same time and more than one activity to be executed simultaneously. Concurrency is identified and represented in the dynamic model.

To enable concurrency, each concurrent element is assigned a separate thread of control. If the concurrency is at object level, then two concurrent objects are assigned two different threads of
control. If two operations of a single object are concurrent in nature, then that object is split among different threads.

Concurrency is associated with the problems of data integrity, deadlock, and starvation. So a clear strategy needs to be made whenever concurrency is required. Besides, concurrency requires to be identified at the design stage itself, and cannot be left for implementation stage.

**Identifying Patterns**

While designing applications, some commonly accepted solutions are adopted for some categories of problems. These are the patterns of design. A pattern can be defined as a documented set of building blocks that can be used in certain types of application development problems.

Some commonly used design patterns are:

- Façade pattern
- Model view separation pattern
- Observer pattern
- Model view controller pattern
- Publish subscribe pattern
- Proxy pattern

**Controlling Events**

During system design, the events that may occur in the objects of the system need to be identified and appropriately dealt with.

An event is a specification of a significant occurrence that has a location in time and space.

There are four types of events that can be modelled, namely:

- **Signal Event**: A named object thrown by one object and caught by another object.
- **Call Event**: A synchronous event representing dispatch of an operation.
- **Time Event**: An event representing passage of time.
- **Change Event**: An event representing change in state.

**Handling Boundary Conditions**

The system design phase needs to address the initialization and the termination of the system as a whole as well as each subsystem. The different aspects that are documented are as follows:

- The start-up of the system, i.e., the transition of the system from non-initialized state to steady state.
- The termination of the system, i.e., the closing of all running threads, cleaning up of resources, and the messages to be sent.
- The initial configuration of the system and the reconfiguration of the system when needed.
- Foreseeing failures or undesired termination of the system.

Boundary conditions are modelled using boundary use cases.

**Object Design**

After the hierarchy of subsystems has been developed, the objects in the system are identified and their details are designed. Here, the designer details out the strategy chosen during the system design. The emphasis shifts from application domain concepts toward computer concepts. The objects identified during analysis are etched out for implementation with an aim to minimize
Object design includes the following phases:

- Object identification
- Object representation, i.e., construction of design models
- Classification of operations
- Algorithm design
- Design of relationships
- Implementation of control for external interactions
- Package classes and associations into modules

**Object Identification**

The first step of object design is object identification. The objects identified in the object-oriented analysis phases are grouped into classes and refined so that they are suitable for actual implementation.

The functions of this stage are:

- Identifying and refining the classes in each subsystem or package
- Defining the links and associations between the classes
- Designing the hierarchical associations among the classes, i.e., the generalization/specialization and inheritances
- Designing aggregations

**Object Representation**

Once the classes are identified, they need to be represented using object modelling techniques. This stage essentially involves constructing UML diagrams.

There are two types of design models that need to be produced:

- **Static Models**: To describe the static structure of a system using class diagrams and object diagrams.
- **Dynamic Models**: To describe the dynamic structure of a system and show the interaction between classes using interaction diagrams and state–chart diagrams.

**Classification of Operations**

In this step, the operation to be performed on objects are defined by combining the three models developed in the OOA phase, namely, object model, dynamic model, and functional model. An operation specifies what is to be done and not how it should be done.

The following tasks are performed regarding operations:

- The state transition diagram of each object in the system is developed.
- Operations are defined for the events received by the objects.
- Cases in which one event triggers other events in same or different objects are identified.
- The sub–operations within the actions are identified.
- The main actions are expanded to data flow diagrams.

**Algorithm Design**
The operations in the objects are defined using algorithms. An algorithm is a stepwise procedure that solves the problem laid down in an operation. Algorithms focus on how it is to be done.

There may be more than one algorithm corresponding to a given operation. Once the alternative algorithms are identified, the optimal algorithm is selected for the given problem domain. The metrics for choosing the optimal algorithm are:

- **Computational Complexity**: Complexity determines the efficiency of an algorithm in terms of computation time and memory requirements.
- **Flexibility**: Flexibility determines whether the chosen algorithm can be implemented suitably, without loss of appropriateness in various environments.
- **Understandability**: This determines whether the chosen algorithm is easy to understand and implement.

**Design of Relationships**

The strategy to implement the relationships needs to be chalked out during the object design phase. The main relationships that are addressed comprise of associations, aggregations, and inheritances.

The designer should do the following regarding associations:

- Identify whether an association is unidirectional or bidirectional.
- Analyze the path of associations and update them if necessary.
- Implement the associations as a distinct object, in case of many-to-many relationships; or as a link to other object in case of one-to-one or one-to-many relationships.

Regarding inheritances, the designer should do the following:

- Adjust the classes and their associations.
- Identify abstract classes.
- Make provisions so that behaviors are shared when needed.

**Implementation of Control**

The object designer may incorporate refinements in the strategy of the state–chart model. In system design, a basic strategy for realizing the dynamic model is made. During object design, this strategy is aptly embellished for appropriate implementation.

The approaches for implementation of the dynamic model are:

- **Represent State as a Location within a Program**: This is the traditional procedure-driven approach whereby the location of control defines the program state. A finite state machine can be implemented as a program. A transition forms an input statement, the main control path forms the sequence of instructions, the branches form the conditions, and the backward paths form the loops or iterations.
- **State Machine Engine**: This approach directly represents a state machine through a state machine engine class. This class executes the state machine through a set of transitions and actions provided by the application.
- **Control as Concurrent Tasks**: In this approach, an object is implemented as a task in the programming language or the operating system. Here, an event is implemented as an inter-task call. It preserves inherent concurrency of real objects.

**Packaging Classes**

In any large project, meticulous partitioning of an implementation into modules or packages is important. During object design, classes and objects are grouped into packages to enable multiple groups to work cooperatively on a project.
The different aspects of packaging are:

- **Hiding Internal Information from Outside View**: It allows a class to be viewed as a “black box” and permits class implementation to be changed without requiring any clients of the class to modify code.

- **Coherence of Elements**: An element, such as a class, an operation, or a module, is coherent if it is organized on a consistent plan and all its parts are intrinsically related so that they serve a common goal.

- **Construction of Physical Modules**: The following guidelines help while constructing physical modules:
  - Classes in a module should represent similar things or components in the same composite object.
  - Closely connected classes should be in the same module.
  - Unconnected or weakly connected classes should be placed in separate modules.
  - Modules should have good cohesion, i.e., high cooperation among its components.
  - A module should have low coupling with other modules, i.e., interaction or interdependence between modules should be minimum.

**Design Optimization**

The analysis model captures the logical information about the system, while the design model adds details to support efficient information access. Before a design is implemented, it should be optimized so as to make the implementation more efficient. The aim of optimization is to minimize the cost in terms of time, space, and other metrics.

However, design optimization should not be excess, as ease of implementation, maintainability, and extensibility are also important concerns. It is often seen that a perfectly optimized design is more efficient but less readable and reusable. So the designer must strike a balance between the two.

The various things that may be done for design optimization are:

- Add redundant associations
- Omit non-usable associations
- Optimization of algorithms
- Save derived attributes to avoid re-computation of complex expressions

**Addition of Redundant Associations**

During design optimization, it is checked if deriving new associations can reduce access costs. Though these redundant associations may not add any information, they may increase the efficiency of the overall model.

**Omission of Non-Usable Associations**

Presence of too many associations may render a system indecipherable and hence reduce the overall efficiency of the system. So, during optimization, all non-usable associations are removed.

**Optimization of Algorithms**

In object-oriented systems, optimization of data structure and algorithms are done in a collaborative manner. Once the class design is in place, the operations and the algorithms need to be optimized.

Optimization of algorithms is obtained by:

- Rearrangement of the order of computational tasks
• Reversal of execution order of loops from that laid down in the functional model
• Removal of dead paths within the algorithm

**Saving and Storing of Derived Attributes**

Derived attributes are those attributes whose values are computed as a function of other attributes \( base\text{\textunderscore}attributes \). Re-computation of the values of derived attributes every time they are needed is a time-consuming procedure. To avoid this, the values can be computed and stored in their computed forms.

However, this may pose update anomalies, i.e., a change in the values of base attributes with no corresponding change in the values of the derived attributes. To avoid this, the following steps are taken:

• With each update of the base attribute value, the derived attribute is also re-computed.
• All the derived attributes are re-computed and updated periodically in a group rather than after each update.

**Design Documentation**

Documentation is an essential part of any software development process that records the procedure of making the software. The design decisions need to be documented for any non-trivial software system for transmitting the design to others.

**Usage Areas**

Though a secondary product, a good documentation is indispensable, particularly in the following areas:

• In designing software that is being developed by a number of developers
• In iterative software development strategies
• In developing subsequent versions of a software project
• For evaluating a software
• For finding conditions and areas of testing
• For maintenance of the software.

**Contents**

A beneficial documentation should essentially include the following contents:

• **High-level system architecture**: Process diagrams and module diagrams
• **Key abstractions and mechanisms**: Class diagrams and object diagrams.
• **Scenarios that illustrate the behavior of the main aspects**: Behavioural diagrams

**Features**

The features of a good documentation are:

• Concise and at the same time, unambiguous, consistent, and complete
• Traceable to the system’s requirement specifications
• Well-structured