

Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system’s object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, *“Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain”*.

The primary tasks in object-oriented analysis (OOA) are −

* Identifying objects
* Organizing the objects by creating object model diagram
* Defining the internals of the objects, or object attributes
* Defining the behavior of the objects, i.e., object actions
* Describing how the objects interact

The common models used in OOA are use cases and object models.

Object-Oriented Design

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include −

* Restructuring the class data (if necessary),
* Implementation of methods, i.e., internal data structures and algorithms,
* Implementation of control, and
* Implementation of associations.

Grady Booch has defined object-oriented design as *“a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”*.

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are −

* Bottom–up approach in program design
* Programs organized around objects, grouped in classes
* Focus on data with methods to operate upon object’s data
* Interaction between objects through functions
* Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object–oriented programming as *“a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships”*.

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling.

## Object Modelling

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modelling can be visualized in the following steps −

* Identify objects and group into classes
* Identify the relationships among classes
* Create user object model diagram
* Define user object attributes
* Define the operations that should be performed on the classes
* Review glossary

## Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.

Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”.

The process of dynamic modelling can be visualized in the following steps −

* Identify states of each object
* Identify events and analyze the applicability of actions
* Construct dynamic model diagram, comprising of state transition diagrams
* Express each state in terms of object attributes
* Validate the state–transition diagrams drawn

## Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modelling can be visualized in the following steps −

* Identify all the inputs and outputs
* Construct data flow diagrams showing functional dependencies
* State the purpose of each function
* Identify constraints
* Specify optimization criteria

## Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are −

* Feasibility Study
* Requirement Analysis and Specification
* System Design
* Implementation
* Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

### **Advantages/Disadvantages of Object Oriented Analysis**

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Focuses on data rather than the procedures as in Structured Analysis. | Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature. |
| The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system. | It cannot identify which objects would generate an optimal system design. |
| The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system. | The object-oriented models do not easily show the communications between the objects in the system. |
| It allows effective management of software complexity by the virtue of modularity. | All the interfaces between the objects cannot be represented in a single diagram. |
| It can be upgraded from small to large systems at a greater ease than in systems following structured analysis. |  |

### **Advantages/Disadvantages of Structured Analysis**

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA. | In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change. |
| It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems. | The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later. |
| The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel. | It does not support reusability of code. So, the time and cost of development is inherently high. |

# **OOAD - UML Analysis Model**

The Unified Modeling Language (UML) is a graphical language for OOAD that gives a standard way to write a software system’s blueprint. It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the structures and the relationships in a complex system.

## Brief History

It was developed in 1990s as an amalgamation of several techniques, prominently OOAD technique by Grady Booch, OMT (Object Modeling Technique) by James Rumbaugh, and OOSE (Object Oriented Software Engineering) by Ivar Jacobson. UML attempted to standardize semantic models, syntactic notations, and diagrams of OOAD.

## Systems and Models in UML

**System** − A set of elements organized to achieve certain objectives form a system. Systems are often divided into subsystems and described by a set of models.

**Model** − Model is a simplified, complete, and consistent abstraction of a system, created for better understanding of the system.

**View** − A view is a projection of a system’s model from a specific perspective.

## Conceptual Model of UML

The Conceptual Model of UML encompasses three major elements −

* Basic building blocks
* Rules
* Common mechanisms

### **Basic Building Blocks**

The three building blocks of UML are −

* Things
* Relationships
* Diagrams

### **Things**

There are four kinds of things in UML, namely −

* **Structural Things** − These are the nouns of the UML models representing the static elements that may be either physical or conceptual. The structural things are class, interface, collaboration, use case, active class, components, and nodes.
* **Behavioral Things** − These are the verbs of the UML models representing the dynamic behavior over time and space. The two types of behavioral things are interaction and state machine.
* **Grouping Things** − They comprise the organizational parts of the UML models. There is only one kind of grouping thing, i.e., package.
* **Annotational Things** − These are the explanations in the UML models representing the comments applied to describe elements.

### **Relationships**

Relationships are the connection between things. The four types of relationships that can be represented in UML are −

* **Dependency** − This is a semantic relationship between two things such that a change in one thing brings a change in the other. The former is the independent thing, while the latter is the dependent thing.
* **Association** − This is a structural relationship that represents a group of links having common structure and common behavior.
* **Generalization** − This represents a generalization/specialization relationship in which subclasses inherit structure and behavior from super-classes.
* **Realization** − This is a semantic relationship between two or more classifiers such that one classifier lays down a contract that the other classifiers ensure to abide by.

### **Diagrams**

A diagram is a graphical representation of a system. It comprises of a group of elements generally in the form of a graph. UML includes nine diagrams in all, namely −

* Class Diagram
* Object Diagram
* Use Case Diagram
* Sequence Diagram
* Collaboration Diagram
* State Chart Diagram
* Activity Diagram
* Component Diagram
* Deployment Diagram

### **Rules**

UML has a number of rules so that the models are semantically self-consistent and related to other models in the system harmoniously. UML has semantic rules for the following −

* Names
* Scope
* Visibility
* Integrity
* Execution

### **Common Mechanisms**

UML has four common mechanisms −

* Specifications
* Adornments
* Common Divisions
* Extensibility Mechanisms

### **Specifications**

In UML, behind each graphical notation, there is a textual statement denoting the syntax and semantics. These are the specifications. The specifications provide a semantic backplane that contains all the parts of a system and the relationship among the different paths.

### **Adornments**

Each element in UML has a unique graphical notation. Besides, there are notations to represent the important aspects of an element like name, scope, visibility, etc.

### **Common Divisions**

Object-oriented systems can be divided in many ways. The two common ways of division are −

* **Division of classes and objects** − A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.
* **Division of Interface and Implementation** − An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

### **Extensibility Mechanisms**

UML is an open-ended language. It is possible to extend the capabilities of UML in a controlled manner to suit the requirements of a system. The extensibility mechanisms are −

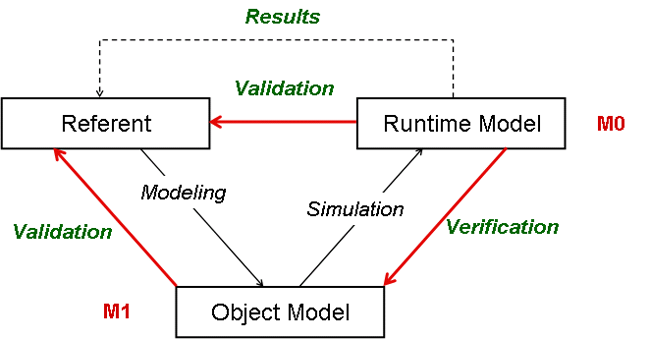
* **Stereotypes** − It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones.
* **Tagged Values** − It extends the properties of UML building blocks.
* **Constraints** − It extends the semantics of UML building blocks.

Software Engineering  
The Analysis Workflow

**Agenda**

* Modeling and Simulation Model
* Analysis Artifacts - Metamodel
* [Analysis Workflow Detail](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/analysisWorkflow.html#Analysis_Workflow_Detail)
* [Architectural Analysis](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/analysisPackages.html#Architectural_Analysis)
* [Analyze Use Case](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/analysisWorkflow.html#Analyze_a_Use_Case_)
* Analysis Model - Rules of Thumb
* [Analyze Classes](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/analysisWorkflow.html#What_are_analysis_classes)
  + A Paradigm for Finding Analysis Objects and Classes
  + CRC Modeling
  + Finding Classes by Using RUP Stereotypes
* [Analyze Use Case Realizations](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/useCaseAnalysis.html)

**Modeling and Simulation Model**



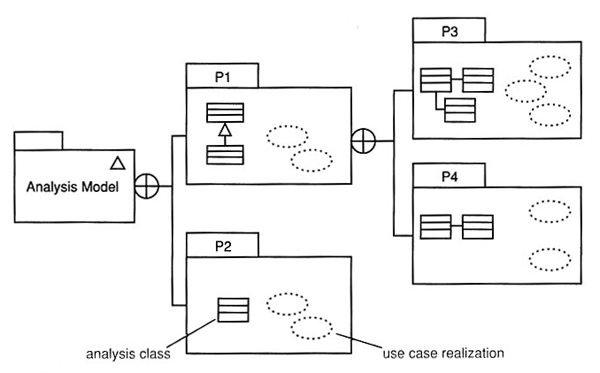
* A class is a descriptor (template) for a set of objects that have the same *features* (attributes, operations, methods, relationships, and behavior).
* An interaction is a unit of behavior of a context classifier, for example a use case. The classifier provides the context for the interaction.

Ways of using UML

1. UML class diagrams as **sketch** - result of requirement analysis
2. UML class diagrams as **blueprint** - result of a design process based on class sketches
3. UML as **programming language** - UML diagrams are compiled directly to executable code

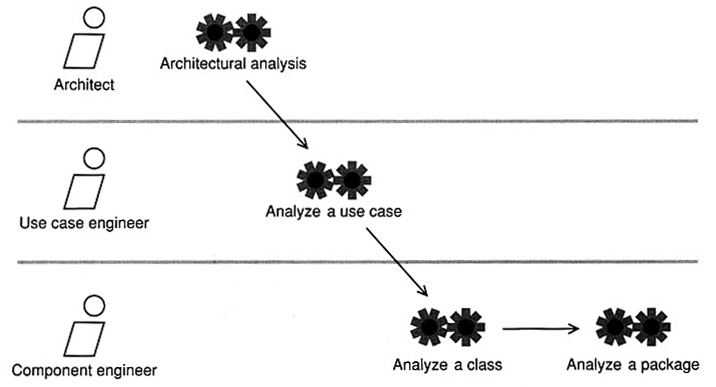
**Analysis Artifacts - Metamodel**

Analysis modeling is strategic as we are trying to model the system's **essential behavior**.



* In the analysis workflow, two key artifacts are produced
  + analysis classes - these model key concepts of the business domain;
  + use case realizations - these illustrate how instances of the analysis classes can interact to realize behavior specified by a use case.

**Analysis Workflow Detail**

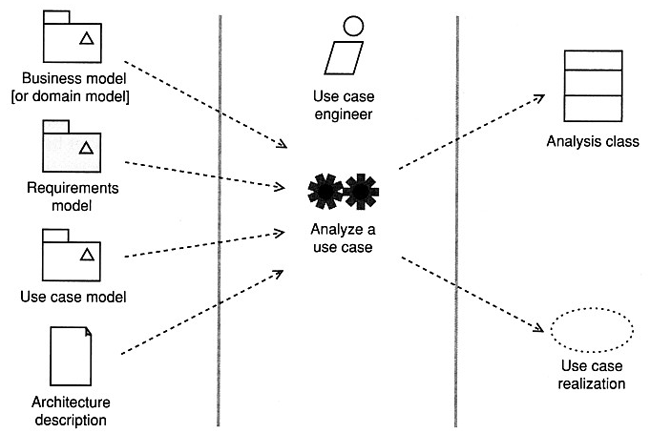


[**Architectural Analysis**](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/analysisPackages.html#Architectural_Analysis)

* Gain understanding and manage strategic system complexity
* Determine the package structure: application-specific vs. application-general packages
* Find analysis packages

**Analyze a Use Case**

"Analyze a use case" involves creating analysis classes and use case realization.



**Analysis Model - Rules of Thumb**

* The analysis model is always in the **language of the business** (project glossary describes the vocabulary of the business domain)
* Create models that "**tell a story**". Each diagram produced should elucidate same important part of the system's desired behavior.
* Concentrate clearly on capturing the **big picture**. Don't get bogged down in the details of how the system will work-there is plenty of time for this in design
* Distinguish clearly between the **problem domain** (the business requirements)  and the **solution domain** (detailed design consideration). Always focus on **abstractions** that exist in the problem domain.
* Always try to **minimize coupling**. Each association between classes  creates coupling between them.
* Explore **inheritance** *if*there seems to be a natural and compelling hierarchy of abstractions. In analysis never apply inheritance just to reuse code. Inheritance is the strongest form of coupling between classes.
* Always ask, "Is the model useful to all the stakeholders?" There's nothing worse than producing an analysis model that is ignored by the business users or the designers and developers. **Yet this happens all too often.**
* Finally-keep the **model simple**! Remember, **inside every complex analysis model is a simple analysis model**struggling to get out.

**Abstractions**

Abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time.

Abstraction in SE is the facility to define objects that represent abstract "experts" that can perform work, report on and change their state, and "communicate" with other objects in the system. The term data *encapsulation* refers to the hiding of expert *structure*, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, and standardizing the way that different data types interact. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called *polymorphism*. When it proceeds in the structure of the types or classes, structuring them to simplify a complex set of relationships, it is called *delegation* or *inheritance*.

**What are analysis classes?**

* **Analysis classes** model important aspects of problem domain such as "customer" or "product".
  + represent a crisp abstraction in the problem domain
  + should map on to real-world business concepts (and carefully named accordingly)
* An analysis class should map in a clear and unambiguous way to real world business concepts.
* Finding the analysis class is the key to OO analysis and design.
  + Analysis classes only have key attributes and very high-level responsibilities.
  + The name of analysis class should indicate its intent.
  + Responsibilities describe cohesive sets of operations.

**Anatomy of an Analysis Class**

* Analysis classes only have **high-level set of attributes** and very **high-level responsibilities.**
* They **indicate** the attributes that the resultant design classes will **probably** have.
* Analysis class operations specify, at high-level, the key services that the class must offer. In design they will become actual implantable operations.

A minimal form of an analysis class consists of the following:

* **Name** - this is mandatory
* **Attributes** - attributes names are mandatory although only important subset of candidate attributes may be modeled at this point
* **Operations** - in analysis, operations might just be very high-level statements of the responsibilities of the class. Operations parameters and return values are only shown where they are important for understanding the model.
* **Visibility** - generally not shown
* [**Stereotypes**](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/modules/uml/index.html#Extensibility_Mechanisms) - may be shown if they enhance the model
* [**Tagged values**](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/modules/uml/index.html#Extensibility_Mechanisms)- may be shown if they enhance the model

**A Paradigm for Finding Analysis Objects and Classes**

A class describes the features of set of objects. Every object is an instance of exactly one class. Finding the right classification scheme is one of the keys to successful OO analysis.

* Identify the ***nouns present***, and create objects representing them. Then find the ***verbs relating to these nouns***(actions) and implement them by adding methods to the objects.
* Leads to larger class hierarchies than we might want

**Instead**

* Use in analysis***CRC modeling,***however in [design](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/designWorkflow.html) use[***commonality* and *variability***](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/designWorkflow.html#Commonality_and_Variability_Analysis)oriented modeling as a primary approach than looking for at just nouns and verbs.

**CRC (Class, Responsibility, Collaboration) Modeling**

[Original CRC paper](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/papers/oopsla89.html)

CRC is brainstorming technique in which you capture on sticky notes the important things in the problem domain

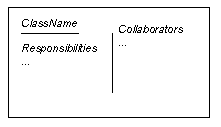
Difficulties with OOP:

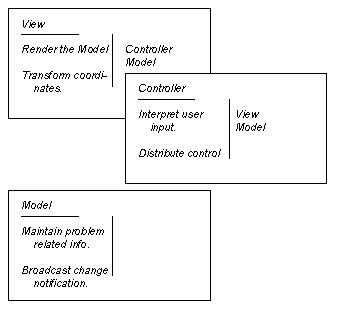
* Give-up the global knowledge of control that is possible with procedural programming (global variables, unnecessary pointers, inappropriate reliance on the implementation of other objects)
* Rely on the local knowledge of objects to accomplish their tasks
* Shift from process, data flow, and data store to class, responsibility, and collaboration

CRC card - a standard index card divided into three sections showing the name of the class, the responsibilities of the class, and the collaborators of the class.

* Finding classes
  + Use a **use case model** as a primary approach than looking for at just nouns and verbs.
  + Look for anything that interacts with the system or is part of the system.
  + Ask yourself, "Is there a customer?"
  + Follow the workflow (scenario) of your use cases.
  + Look for the three to five main classes immediately.
  + Create a new card for a class immediately.
  + Use one or two words to describe the class.
  + Class names are singular.
* Responsibilities identify problems to be solved.
  + Ask yourself what class knows.
  + Ask yourself what the class does.
  + If you've identified a responsibility, ask yourself what class it belongs to.
  + Sometimes we identify responsibilities that we won't implement, and that's OK.
  + Classes will collaborate to fulfill many of their responsibilities.
* Collaborators (senders - experts looking for other experts) will send or be sent messages in the course of satisfying responsibilities.
  + Collaboration occurs when a sender needs information that it doesn't have.
  + Collaboration occurs when a sender needs to modify information that it doesn't have.
  + There will always be at least one initiator of any given collaboration.
  + Sometimes the collaboration does the bulk of the work.
  + New responsibilities may be created to fulfill the collaboration.
  + Don't pass the buck.

**CRC Cards**



Model, view, controller (MVC) Paradigm  


**CRC Process**

1. Explain that this is a true [brainstorm](https://pja.mykhi.org/mgr/blokowe/INN/sorcersoft.org/io/uml/brainstorm.html)
2. Explain the CRC modeling technique
3. Interactively perform the following modeling steps:
   * Find classes
   * Find responsibilities
   * Define collaborators
   * Define use cases in your CRC terms
   * Rearrange the cards
   * Prototype
4. Perform use-case scenario

* Stakeholder involvement is essential for CRC success.
* Important business concepts generally become classes.

**Finding Classes by Using RUP Stereotypes**

* It can be complementary to CRC modeling
* Three distinct types of analysis class can be distinguished
  + **<<entity>>** - a class that is used to model persistent information about something
    - value classes
    - cut across many use case
    - are manipulated by control classes
    - provide info to, and accept info from, boundary classes
    - represent key things managed by the system (e.g., Customer)
    - are often persistent
  + **<<control>>** - a class that encapsulates use-case-specific behavior
  + **<<boundary>>** - a class that mediates interaction between the system and its environment  
    - communicate with external actors
    - user interface classes - that interface between the system and humans
    - system interface classes - that interface with other systems
    - device interface classes - that  interface with external devices (sensors)