Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior.

Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-oriented analysis (OOA) applies object-modelling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on what the system does, OOD on how the system does it.

Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object.

The implementation of "message sending" varies depending on the architecture of the system being modeled, and the location of the objects being communicated with.

Object-oriented analysis

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD). Analysis is done before the Design.

The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include

some kind of user interface mock-up. The purpose of object-oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer defined requirements.

Object-oriented design

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of how the system is to be built

Unified Modeling Language (UML)

Unified Modelling Language (UML) is the set of notations, models and diagrams used when developing object-oriented (OO) systems. UML is the industry standard OO visual modelling language. The latest version is UML 1.4 and was formed from the coming together of three leading software methodologists; Booch, Jacobson and Rumbaugh. UML allows the analyst ways of describing structure, behaviour of significant parts of system and their relationships.

It is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.
The Unified Modeling Language is commonly used to visualize and construct systems which are software intensive. A special language called Systems Modeling Language was designed to handle systems which were defined within UML 2.0.

The Unified Modeling Language is important for a number of reasons. First, it has been used as a catalyst for the advancement of technologies which are model driven, and some of these include Model Driven Development and Model Driven Architecture. Because an emphasis has been placed on the importance of graphics notation, UML is proficient in meeting this demand, and it can be used to represent behaviors, classes, and aggregation. While software developers were forced to deal with more rudimentary issues in the past, languages like UML have now allowed them to focus on the structure and design of their software programs. It should also be noted that UML models can be transformed into various other representations, often without a great deal of effort. One example of this is the ability to transform UML models into Java representations.

Many of these languages may be supported by OMG. The Unified Modeling Language has a number of features and characteristics which separate it from other languages within the same

category. Many of these attributes have allowed it to be useful for developers. In this article, I intend to show you many of these attributes, and you will then understand why the Unified Modeling Language is one of the most powerful languages in existence today.

The building blocks of UML

These are the fundamental elements in UML. Every diagram can be represented using these building blocks. The building blocks of UML contains three types of elements. They are:

1) Things (object-oriented parts of uml)

2) Relationships (relational parts of uml)

3) Diagrams

Things

A diagram can be viewed as a graph containing vertices and edges. In UML, vertices are replaced by things, and the edges are replaced by relationships. There are four types of things in UML. They are:

1) Structural things (nouns of uml – static parts)

2) Behavioral things (verbs of uml – dynamic parts)

3) Grouping things (organizational parts)

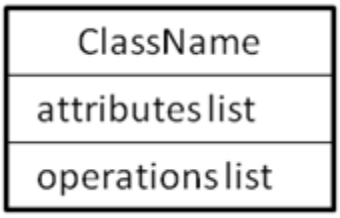
4) Annotational things (explanatory parts)

Structural things

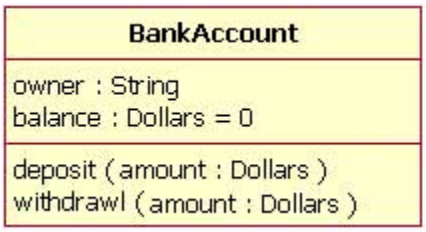
Represents the static aspects of a software system. There are seven structural things in UML. They are:

Class: A class is a collection of similar objects having similar attributes, behavior, relationships and semantics. Graphically class is represented as a rectangle with three compartments.

Graphical representation:



Example:



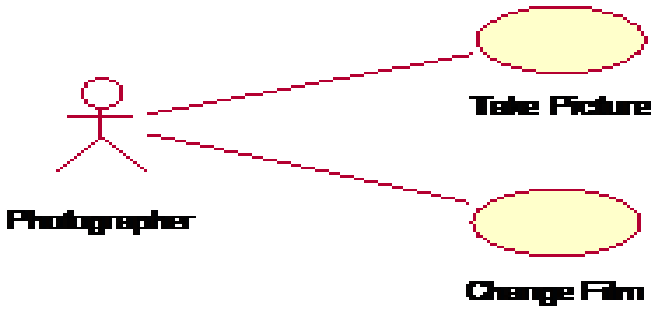
Interface: An interface is a collection of operation signatures and/or attribute definitions that ideally define a cohesive set of behavior. Graphically interface is represented as a circle or a class symbol stereotyped with interface.

Graphical representation:

Use Case: A use case is a collection of actions, defining the interactions between a role (actor) and the system. Graphically use case is represented as a solid ellipse with its name written inside or below the ellipse.

Graphical representation:

Example:

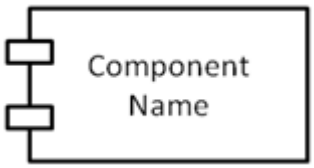


Collaboration: A collaboration is the collection of interactions among objects to achieve a goal. Graphically collaboration is represented as a dashed ellipse. A collaboration can be a collection of classes or other elements.

Graphical representation:

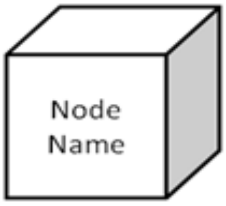
Component: A component is a physical and replaceable part of a system. Graphically component is represented as a tabbed rectangle. Examples of components are executable files, dll files, database tables, files and documents.

Graphical representation:

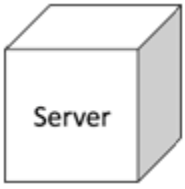


Node: A node is a physical element that exists at run time and represents a computational resource. Graphically node is represented as a cube. Examples of nodes are PCs, laptops, smartphones or any embedded system.

Graphical representation:

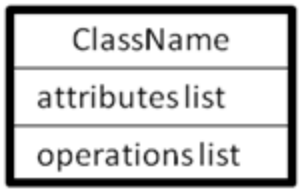


Example:



Active Class: A class whose objects can initiate its own flow of control (threads) and work in parallel with other objects. Graphically active class is represented as a rectangle with thick borders.

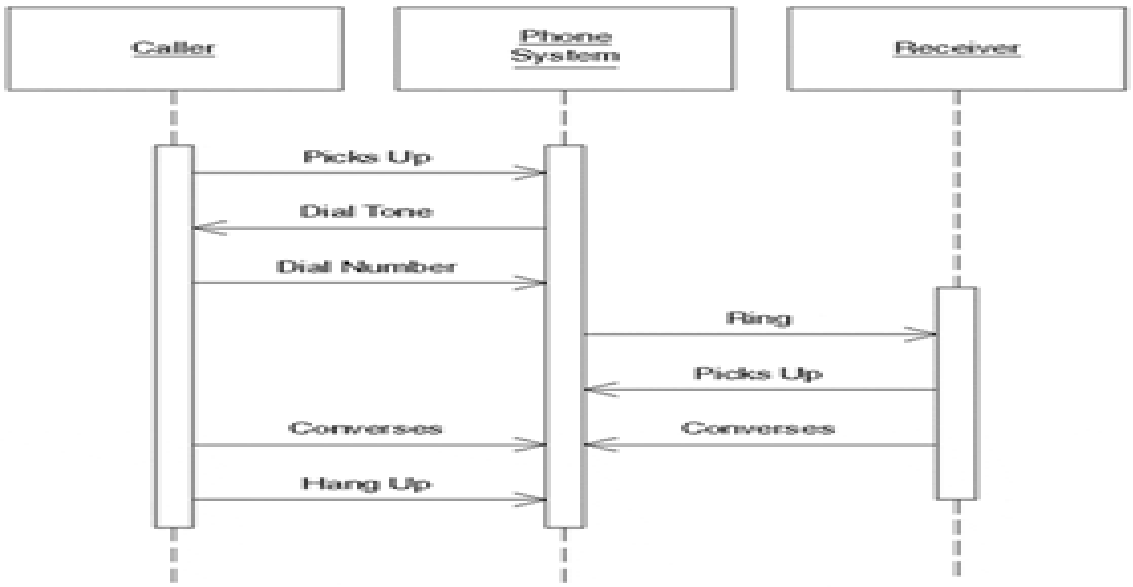
Graphical representation:



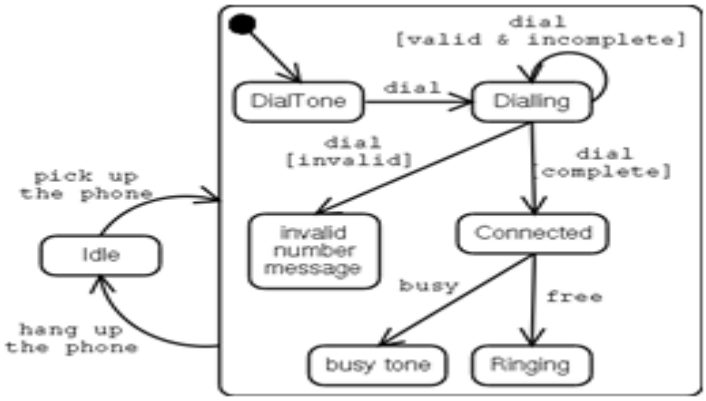
Behavioral Things

Represents the dynamic aspects of a software system. Behavior of a software system can be modeled as interactions or as a sequence of state changes.

Interaction: A behavior made up of a set of messages exchanged among a set of objects to perform a particular task. A message is represented as a solid arrow. Below is an example of interaction representing a phone conversation:



State Machine: A behavior that specifies the sequences of states an object or interaction goes through during its' lifetime in response to events. A state is represented as a rectangle with rounded corners. Below is an example of state machine representing the states of a phone system:

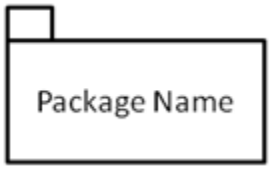


Grouping Things

Elements which are used for organizing related things and relationships in models.

Package: A general purpose mechanism for organizing elements into groups. Graphically package is represented as a tabbed folder. When the diagrams become large and cluttered, related are grouped into a package so that the diagram can become less complex and easy to understand.

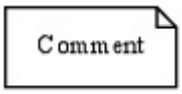
Graphical representation:



Annotational Things

Note: A symbol to display comments. Graphically note is represented as a rectangle with a dog ear at the top right corner.

Graphical representation:

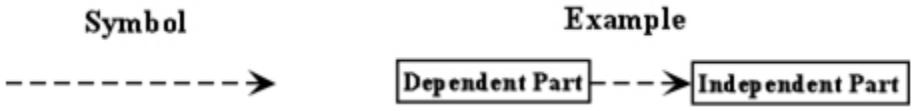


Relationships

The things in a diagram are connected through relationships. So, a relationship is a connection between two or more things.

Dependency: A semantic relationship, in which a change in one thing (the independent thing) may cause changes in the other thing (the dependent thing). This relationship is also known as “using” relationship. Graphically represented as dashed line with stick arrow head.

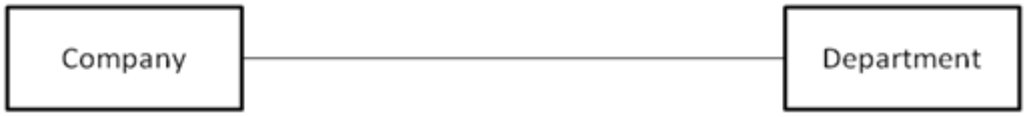
Graphical representation:



Example:

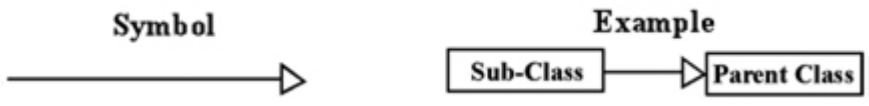
Association: A structural relationship describing connections between two or more things. Graphically represented as a solid line with optional stick arrow representing navigation.

Example:

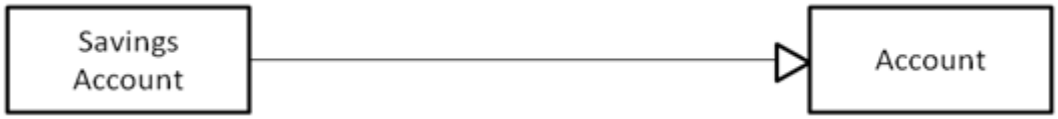


Generalization: Is a generalization-specialization relationship. Simply put this describes the relationship of a parent class (generalization) to its subclasses (specializations). Also known as “is-a” relationship.

Graphical representation:



Example:



Realization: Defines a semantic relationship in which one class specifies something that another class will perform. Example: The relationship between an interface and the class that realizes or executes that interface.

Graphical representation and Example: