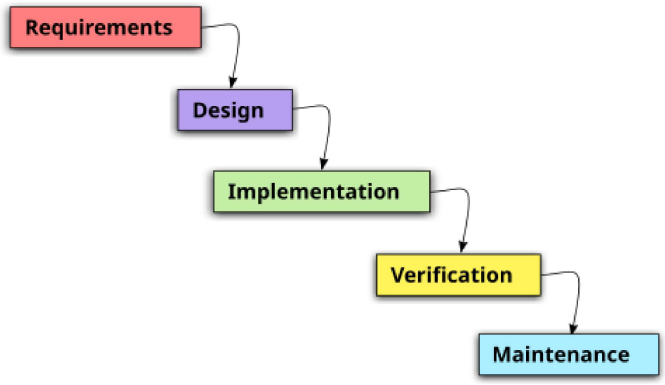
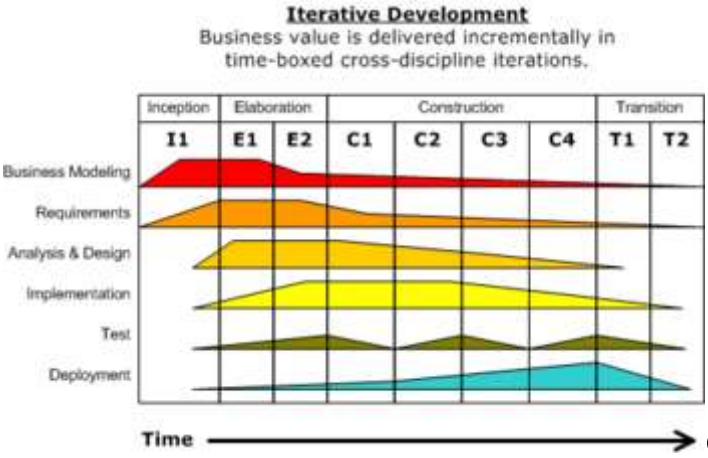
Object-oriented analysis and design (**OOAD**) is an approach to analyzing and designing a computer-based system by applying an object-oriented mindset and using visual modeling throughout the software development process. It consists of object-oriented analysis (OOA) and object-oriented design (OOD) – each producing a model of the system via object-oriented modeling (OOM). Proponents contend that the models should be continuously refined and evolved, in an iterative process, driven by key factors like risk and business value.

OOAD is a method of analysis and design that leverages object-oriented principals of decomposition and of notations for depicting logical, physical, state-based and dynamic models of a system. As part of the software development life cycle OOAD pertains to two early stages: often called requirement analysis and design.[1]



The Waterfall Model.



OOAD is conducted in an iterative and incremental manner, as formulated by the Unified Process.

Although OOAD could be employed in a waterfall methodology where the life cycle stages as sequential with rigid boundaries between them, OOAD often involves more iterative approaches. Iterative methodologies were devised to add flexibility to the development process. Instead of working on each life cycle stage at a time, with an iterative approach,

work can progress on analysis, design and coding at the same time. And unlike a waterfall mentality that a change to an earlier life cycle stage is a failure, an iterative approach admits that such changes are normal in the course of a knowledge-intensive process – that things like analysis can't really be completely understood without understanding design issues, that coding issues can affect design, that testing can yield information about how the code or even the design should be modified, etc.[2] Although it is possible to do object-oriented development in a waterfall methodology, most OOAD follows an iterative approach.

The object-oriented paradigm emphasizes modularity and re-usability. The goal of an object-oriented approach is to satisfy the "open–closed principle". A module is open if it supports extension, or if the module provides standardized ways to add new behaviors or describe new states. In the object-oriented paradigm this is often accomplished by creating a new subclass of an existing class. A module is closed if it has a well defined stable interface that all other modules must use and that limits the interaction and potential errors that can be introduced into one module by changes in another. In the object-oriented paradigm this is accomplished by defining methods that invoke services on objects. Methods can be either public or private, i.e., certain behaviors that are unique to the object are not exposed to other objects. This reduces a source of many common errors in computer programming.[3]

Object-oriented analysis

Common models used in OOA are the use case and the object model. A use case describes a scenario for standard domain functions that the system must accomplish. An object model describes the names, class relations, operations, and properties of the main objects. User-interface mockups or prototypes can also be created to help understanding.[4]

Unlike with OOA that organizes requirements around objects that integrate processes and data, with other analysis methods, processes and data are considered separately. For example, data may be modeled by entity–relationship diagrams, and behaviors by flowcharts or structure charts.

Artifacts

Outputs of OOA are inputs to OOD. In an iterative approach, and an output artifact does not need to be completely developed to serve as input to OOD. Both OOA and OOD can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot. OOA artifacts include:

Conceptual model

A conceptual model captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

Use case

A use case is a description of sequences of events that lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.

System sequence diagram

A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

User interface documentation

Optional documentation that shows and describes the look and feel of the user interface.

Relational data model

If an object database is not used, a relational data model should usually be created before the design since the strategy chosen for object–relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the OOD artifacts in parallel, and the growth of an artifact can stimulate the refinement of other artifacts.

Object-oriented design

OOD, a form of software design, is the process of planning a system of interacting objects to solve a software problem. A designer applies implementation constraints to the conceptual model produced in OOA. Such constraints could include the hardware and software platforms, the performance requirements, persistent storage and transaction, usability of the system, and limitations imposed by budgets and time. Concepts in the analysis model which is technology independent, are mapped onto implementing classes and interfaces resulting in a model of the solution domain, i.e., a detailed description of *how* the system is to be built on concrete technologies.[5]

OOD activities include:

- Defining objects and their attributes
- Creating class diagrams from conceptual models
- Using design patterns[6]
- Defining application frameworks
- Identifying persistent objects/data
- Designing the object relation mapping if a relational database is used
- Identifying remote objects

OOD principles and strategies include:

Dependency injection

Dependency injection is that if an object depends upon having an instance of some other object then the needed object is "injected" into the dependent object; for example, being passed a database connection as an argument to the constructor instead of creating one internally.

Acyclic dependencies principle

The acyclic dependencies principle is that dependency graph of packages or components (the granularity depends on the scope of work for one developer) should have no cycles. This is also referred to as having a directed acyclic graph.[7] For example, package C depends on package B, which depends on package A. If package A depended on package C, you would have a cycle.

Composite reuse principle

The composite reuse principle is to favor polymorphic composition of objects over inheritance.[6]

Artifacts

Sequence diagram

Extend the sequence diagram to add specific objects that handle the system events. A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

Class diagram

A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, its attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.