**Rumbaugh Methodology (OMT)**

The Rumbaugh methodology also known as OMT (Object Modeling Technique) is an approach used to develop manageable object-oriented systems and host object oriented programming. The purpose is to allow for class attributes, methods, inheritance, and association to be easily expressed. OMT is used in the real world for software modeling and designing.

According to Rumbaugh, there are several main reasons to utilize this modeling approach. One is to simulate entities before constructing them and another is to make communication with customers easier. Additionally, it help to reduce complexity through visualization.

**OMT consists of four stages:**

* Analysis
* Systems Design
* Object Design
* Implementation

**Additionally, OMT is always broken down into three separate parts. These parts are the:**

* An object model
* A dynamic model
* A functional model.

We will explain the bullets in greater detail later on.

**Notation/Diagramming Technique**

In OMT, abstract or concrete classes are represented as rectangles. Italicized font indicates abstraction while a normal font represents a concrete class.

| **CONCRETE** | ***ABSTRACT*** |
| --- | --- |

Lines indicate an association. There can be different multiplicities of association:

* A regular line indicates exactly one association
* Numbers above the line will indicate the multiplicity of associations
* A line with an open circle indicates optional multiplicity
* A line with a filled circle indicates many multiplicities
* A line with a diamond indicates an aggregation

**Note: Multiplicity indicates the number of objects that can participate in a relationship or the number of instances of one class as it relates to another. Also, an aggregation is a tighter-defined association.**

Additionally, objects are portrayed as rounded rectangles, the $ sign represents class operations or attributes, and a triangle in between associations represents a generalization or inheritance.

**Phases**

OMT consists of the following four phases:

**Analysis:** Assigns an object, dynamic and functional model to the design. Determines important properties and domain.
**Systems Design:** Outlines the basic systems structure of the program. Accounts for data storage and concurrency among other things.

**Object Design:** Classifies objects and determines operations and data structures. Inheritance and different associations are also checked.
**Implementation:** Conveys design through code.

**Models**

There are three main types of models:

**Object Model:** This model is the most stable and static of the models. Concerns itself with mainly classes and their associations with attributes. Divides the model into objects.
**Dynamic Model:** Concerns itself with the interaction between objects through events, states, and transitions.
**Functional Model:** Concerns itself with data flows, data storage, constraints, and processes

**Example**

Below is a simple example of a bank account. We've described the relationship and process of simple transactions between customers and bank accounts. We've also listed the instances and methods (only in the BankAccount class). And we have optional multiplicity of association between our transaction and bank account.

Take a look.



**Booch Methodology**

Booch's methodology focuses on object-oriented analysis and design and consists of five activities: conceptualization, analysis, design, evolution, and maintainance.

**Macro process:**

* **Conceptualization:** Establish requirements taking into account the perspective of the customer
* **Analysis:** Develop a model by defining object classes, their attributes, methods, and inheritance. Include associations, the dynamic part of a model.
* **Design:** Develop a structure/architecture where logical and physical details are discussed
* **Evolution:** As it relates to the implementation
* **Maintainance:** Maintainance following the delivery of the product

**Micro process:**

* Identify classes and objects
* Identify semantics as it relates to programming
* Identify interactions/relationships
* Note abstraction as it relates to interfaces and their implementation

The macro process is the general cycle that follows in Booch's method, the micro process emerges during the development or implementation of new features (classes, behaviors, etc.)

**Note: The methodology details six different models/diagrams: class diagrams, state transition diagrams, object diagrams, process diagrams, module diagrams, and interaction diagrams. Class and module diagrams are static while state transition diagrams are dynamic.**

**Example**

Below is a diagram illustrating a booch diagram as it relates to the types of bank accounts. We have a superclass BankAccount and we've listed its attributes below. As you can see, our classes SavingsAccount and CheckingsAccount inherit the superclass.


**Jacobson Methodology (OOSE)**

The Jacobson methodology, also known as Object-Oriented Software Engineering (OOSE) or even Objectory, is a method used to plan, design, and implement object-oriented software. The method is broken down into five parts: a set of requirements, an analysis, a design, an implementation, and a testing model. Uniquely, the methodology or OOSE utilizes use cases in its design. We will go into more detail about these use cases and the five parts later on.

According to the author, OOSE is meant to provide robustness to a software design making it more resilient, manageable, and maintainable.

**Components**

OOSE can be broken down into five separate components as we previously mentioned: a set of requirements, an analysis, a design, an implementation, and a testing model. These five parts correspond with certain verbs we can associate them with. The requirements model is expresses, the analysis model is structures, the design model realizes, the implementation model implements, and the test model verifies.

**Models:**
**Requirements:** Create problem domain object diagram (as they satisfy requirements) and specifies use case diagrams
**Analysis:** Analysis diagrams (these are similar to the ones we covered in the other methods)
**Design:** State transition diagrams and interaction diagrams (these are similar to the ones we covered in the other methods)
**Implementation:** Implementation of the model
**Testing Model:** Testing of the model

**Use Cases**

Use cases help us understand the how we want to design our system; more specifically, they are scenarios for helping us understanding the requirements of our system.

We can classify a use case as an interaction between a user and the system; hence "use" case. Broken down each use case has a goal and a responsibility: a goal for the user and a responsibility for the system to execute. Use cases can include non-formal code with no clear events flow, to clear events flow, to formal pseudo code. Additionally, use cases should be connected/related to one another with actors supporting the execution of the software and actors receiving the result of the execution in a chain of interconnected use cases.

Importantly, however a use case must contain these several elements:

* The how and when the use case begins and ends
* Governs interactions between the various actors - when the interaction occurs and what material is exchanged
* The how and when of data storage and data usage
* Exceptions
* The how and when of how the constraints of the problem domain are handled

As you can see, use cases have a emphasis on the **how** and the **when** of every feature and element of software. This helps improve the robustness of our system and its usability.

**Example**

Below is an example of a diagram illustrating a simplified ordering kiosk system. The direction of the arrow indicates what each use cases uses. The ordering kiosk relies on the customer for the order; similarly the receipt relies on the ordering kiosk for the order, who in turn the chef relies on. Our circles represent actors while our horizontal ovals represent the use cases.

