**Importance of modelling**

To know the importance of modeling let us assume that you are going to build a dog house, a house for your family and a high rise office for a client. In the case of a dog house you need minimal resources and the satisfaction of the dog is not that important.

In the case of building a house for your family, you need to satisfy the requirements of your family members and the amount of resources are non-trivial. In the case of building a high rise office, the amount of risk is very high.

Curiously, a lot of software development organizations start out wanting to build high rises but approach the problem as if they are knocking at a dog house. Sometimes you get lucky, If you have the right people at the right moment and if all the planets align properly, then you might get your team to create a software product that satisfies its users. This happens very rarely.

Unsuccessful software projects fail in their own unique ways, but all successful software projects are alike in many ways. There are many elements that contribute to a successful software organization; one common element is the use of modeling.

Modeling is a proven and well-accepted engineering technique. We build architectural models of houses and high rises to help their users visualize the final product.

Modeling is not only limited to the construction industry. Modeling is applied in the fields of aeronautics, automobile, picture, sociology, economics, software development and many more. We build models so that we can validate our theories or try out new ones with minimal risk and cost.

What, then, is a model? Simply put,

A model is a simplification of reality.

A good model includes those elements that have broad effect and omits those minor elements that are not relevant to the given level of abstraction. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

**Why do we model? There is one fundamental reason:**

We build models so that we can better understand the system we are developing.

Through modeling, we achieve four aims:

Models help us to visualize a system as it is or as we want it to be.

Models permit us to specify the structure or behavior of a system.

Models give us a template that guides us in constructing a system.

Every project can benefit from modeling. Modeling can help the development team better visualize the plan of their system and allow them to develop more rapidly by helping them build the right thing. The more complex your project, the more likely it is that you will fail or that you will build the wrong thing if you do any modeling at all.

**Q.1 : What is Generalization ?**

**Ans : Generalization**

**A generalization is a relationship between a general thing (called the superclass or parent)and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called an "is-a-kind of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Often but not always the child has attributes and operations in addition to those found in its parents. An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as shown in Figure. Use generalizations when you want to show parent/child relationships.**

**General class?**

Loosely speaking, a class which tells the main features but not the specific details. The classes situated at the top of the inheritance hierarchy can be said as General.

**Specific class?**

A class which is very particular and states the specific details. The classes situated at the bottom of the inheritance hierarchy can be said as Specific.

Relatively General Class: Money

Relatively Specific Class: Dollar, Euro, Rupees

Lemon, Orange are more Specific than Citrus

Banana, Apple are more Specific than Non-Citrus

Citrus, Non-Citrus are more Specific than Fruit

Fruit is most general class

Conversion of one class type to another class type?

We can convert references to one class type to another class type in Java. But for the conversion to happen the classes should be related with each other by the way of inheritance.

Therefore,

References for Vehicle and Bus can be type-casted to each other.

References for Vehicle and Car can be type-casted to each other.

References for Vehicle and Truck can be type-casted to each other.

References for Bus, Car and Truck can’t be type-casted to each other.

**Generalization**

Converting a subclass type into a superclass type is called ‘Generalization‘ because we are making the subclass to become more general and its scope is widening. This is also called widening or up casting. Widening is safe because the classes will become more general.

**For example, if we say Car is a Vehicle, there will be no objection. Thus Java compiler will not ask for cast operator in generalization.**

class Father {

public void work()

{

System.out.println("Earning Father");

}

}

class Son extends Father {

public void play()

{

System.out.println("Enjoying son");

}

}

class Main {

public static void main(String[] args)

{

// father is a superclass reference

Father father;

// new operator returns a subclass reference

father = (Father) new Son();

// which is widened using casting

// and stored in father variable

// Though casting is done but it is not needed

father.work();

// Uncomment next line to see the error

// father.play();

}

}

**Output:**

Earning Father

So, in widening or Generalization, we can access all the superclass methods, but not the subclass methods.

**Example: Now Suppose we override the superclass methods in sub class**

class Father {

public void work()

{

System.out.println("Earning Father");

}

}

class Son extends Father {

@Override

public void work()

{

System.out.println("Earning Son");

}

}

class Main {

public static void main(String[] args)

{

// father is the super class reference

Father father;

// new operator returns a subclass reference

father = (Father) new Son();

// which is widened using casting

// and stored in father variable

// Though casting is done but it is not needed

// subclass method is invoked

father.work();

}

}

**Output:**

Earning Son

-> E-commerce application -> Class diagram / Is-A relation , simple aggregation

Code -> simple class code with inheritance

**Q 2: What is Root and Leaf class?**

Ans : What is root class?

**Root class**

A root class inherits from no other class and defines an interface and behavior common to all objects in the hierarchy below it. All objects in that hierarchy ultimately inherit from the root class. A root class is sometimes referred to as a base class

**What class is a leaf?**

In class-based object-oriented programming languages, a leaf class is a class that should not be subclassed.

**Q 3: Abstract classes and Operation .**

Ans: An abstract class is a class that is declared abstract —it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

**Java Abstract Class**

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class. For example,

// create an abstract class

abstract class Language {

// fields and methods

}

...

// try to create an object Language

// throws an error

Language obj = new Language();

An abstract class can have both the regular methods and abstract methods. For example,

abstract class Language {

// abstract method

abstract void method1();

// regular method

void method2() {

System.out.println("This is regular method");

}

}

To know about the non-abstract methods, visit Java methods. Here, we will learn about abstract methods.

**Java Abstract Method**

A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. For example,

abstract void display();

Here, display() is an abstract method. The body of display() is replaced by ;.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

// error

// class should be abstract

class Language {

// abstract method

abstract void method1();

}

**Example: Java Abstract Class and Method**

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

abstract class Language {

// method of abstract class

public void display() {

System.out.println("This is Java Programming");

}

}

class Main extends Language {

public static void main(String[] args) {

// create an object of Main

Main obj = new Main();

// access method of abstract class

// using object of Main class

obj.display();

}

}

**Output**

This is Java programming

In the above example, we have created an abstract class named Language. The class contains a regular method display().

**Q 4: Modeling with Generalization.**

Ans: Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

**Generalization**

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an “is – a – kind – of” relationship. For example, “car is a kind of land vehicle”, or “ship is a kind of water vehicle”.

**Q 5: MOdeling with Encapsulation.**

Ans: Encapsulation is the object model concept of including processing or behavior with the object instances defined by the class. Encapsulation allows code and data to be packaged together.

The definition of methods for a class is an integral part of encapsulation. A method is programming code that performs the behavior an object instance can exhibit. Calculating the age of a person would be an example of such behavior. The figure shows a way of looking at encapsulating the age method with an instance object. The code for the age method is "attached" to or encapsulated with the object rather than part of the application.

The notation for object model encapsulation is shown below. Methods are shown at the bottom of the class notation.

**Encapsulation**

Code and data were not always packaged together. At one time, for example, it was necessary to define an age calculation in each application or have a library that contained the age calculation routine. Having an age calculation or any routine, replicated in many applications may make it difficult to ensure that a change in the routine was made everywhere that it is used. Using a library improves this situation if use of the library is enforced. Nevertheless, with a library, you can never be sure which routine is supposed to be used with which data. It is entirely possible to execute the right code on the wrong data or the wrong code on the right data.

Object systems recognize which methods belong to which data. You cannot execute the right method on the wrong data as you could with library systems. The correct execution of methods is called dispatching, and it is handled by the object system.



**Encapsulation**: The process of providing a public interface to interact with the object while hiding other information inside the object. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. The main way that encapsulation helps reduce rippling effects of change is by keeping as many of the implementation details private to the class. By limiting the interface only to those members needed to use the class, many changes can be made to the implementation without affecting any code that uses the class.

**Q 6: Visibility of features.**

Ans: Public,Private,Protected.

**Q 7: Owner Scope**

Ans: In object-oriented programming, methods and variables have various scope. Scope means that the method or variable may or may not be directly accessible to other objects or classes. Classes that do not have instances may be accessible to the system.

**Class Scope:** Class variables and class methods are associated with a class. An instance of the class (object) is not required to use these variables or methods. Class methods cannot access instance variables or methods, only class variables and methods.

**Instance Scope:** Instance variables and instance methods are associated with a specific object. They can access class variables and methods.

**Private Scope:** Private variables and private methods are only accessible to the object they are contained in. So if something goes wrong with that, there is usually only one source file to look at. If you have a million lines of code in your project, but your classes are kept small, this can reduce your bug tracking effort by a large factor.

**Public Scope:** Public variables and public methods are accessible outside the object they are contained in, which for practical considerations means "potentially anywhere." If something goes wrong with a public field, the culprit can be anywhere, so in order to track down the bug, you may have to look at quite a lot of code

**Protected Scope:** Protected variables and protected methods are accessible by the class they are in and inheriting classes (subclasses).

Q 8: Relationships

Ans:



Composition

Two class diagrams. The diagram on top shows Composition between two classes: A Car has exactly one Carburetor, and a Carburetor is a part of one Car. Carburetors cannot exist as separate parts, detached from a specific car. The diagram on bottom shows Aggregation between two classes: A Pond has zero or more Ducks, and a Duck has at most one Pond (at a time). Duck can exist separately from a Pond, e.g. it can live near a lake. When we destroy a Pond we usually do not kill all the Ducks.

The UML representation of a composition relationship shows composition as a filled diamond shape on the containing class end of the lines that connect contained class(es) to the containing class.

Differences between Composition and Aggregation

**Composition relationship**

1. When attempting to represent real-world whole-part relationships, e.g. an engine is a part of a car.

2. When the container is destroyed, the contents are also destroyed, e.g. a university and its departments.

**Aggregation relationship**

1. When representing a software or database relationship, e.g. car model engine ENG01 is part of a car model CM01, as the engine, ENG01, maybe also part of a different car model.[9]

2. When the container is destroyed, the contents are usually not destroyed, e.g. a professor has students; when the professor leaves the university the students do not leave along with them.

Thus the aggregation relationship is often "catalog" containment to distinguish it from composition's "physical" containment.

**Q 9: Containers**

Ans: Java containers are based on a hierarchy of abstract data types shown in Figure 5.3. Being Java interfaces, these data types only contain method declarations; in other words, they do not contain any implementation code for the methods. The interface Collection considers a container to be a group of objects, without any assumptions about the uniqueness of the objects in the container. This interface declares methods common to all containers of types List and Set. These include such general-purpose methods like add to insert a new element into a container; clear to remove all the elements from a container; isEmpty to check whether a container is empty; iterator to return an iterator to a container; remove for removing a specific object from a container; removeAll for removing all the objects from a container that are specified via an argument container; and so on. A particularly noteworthy method declared in this interface is toArray. Since this method must be implemented by all containers of type Collection, it can be used to construct an array version of any of those containers.



**Q 10: Types and modeling with types**

Ans: Intention of object oriented modeling and design is to learn how to apply object -oriented concepts to all the stages of the software development life cycle.Object-oriented modeling and design is a way of thinking about problems using models organized around real world concepts. The fundamental construct is the object, which combines both data structure and behavior.

**Purpose of Models:**

* Testing a physical entity before building it
* Communication with customers
* Visualization
* Reduction of complexity

**Types of Models:**

There are 3 types of models in the object oriented modeling and design are: Class Model, State Model, and Interaction Model. These are explained as following below.

**Class Model:**

The class model shows all the classes present in the system. The class model shows the attributes and the behavior associated with the objects.

The class diagram is used to show the class model.The class diagram shows the class name followed by the attributes followed by the functions or the methods that are associated with the object of the class.Goal in constructing class model is to capture those concepts from the real world that are important to an application.

**State Model:**

State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states.Actions and events in a state diagram become operations on objects in the class model. State diagram describes the state model.

**Interaction Model:**

Interaction model is used to show the various interactions between objects, how the objects collaborate to achieve the behavior of the system as a whole.

The following diagrams are used to show the interaction model:

Use Case Diagram

Sequence Diagram

Activity Diagram

**Q 11: Interfaces and interface Modeling**

Ans: In UML modeling, interfaces are model elements that define sets of operations that other model elements, such as classes, or components must implement. An implementing model element realizes an interface by overriding each of the operations that the interface declares.

You can use interfaces in class diagrams and component diagrams to specify a contract between the interface and the classifier that realizes the interface. Each interface specifies a well-defined set of operations that have public visibility. The operation signatures tell the implementing classifiers what kind of behavior to invoke, but not how they should invoke that behavior. Many classifiers can implement a single interface, each one providing a unique implementation.

Interfaces support the hiding of information and protect client code by publicly declaring certain behavior or services. Classes or components that realize the interfaces by implementing this behavior simplify the development of applications because developers who write client code need to know only about the interfaces, not about the details of the implementation. If you replace classes, or components that implement interfaces, in your model, you do not need to redesign your application if the new model elements implement the same interfaces.

You can specify the following types of interfaces:

Provided interfaces: these interfaces describe the services that instances of a classifier (supplier) offer to their clients

Required interfaces: these interfaces specify the services that a classifier needs to perform its functions and to fulfill its own obligations to its clients

An interface typically has a name that reflects the role that it plays in an application. A common convention is to prefix the name of the interface with a forward slash to indicate that a model element is an interface.

As the following figures illustrate, the diagram editor displays an interface in the following ways:

Class rectangle symbol that contains the keyword «interface». This notation is also called the internal or class view.



An interface class diagram of an interface named Interface1.

Use the class shape when you need to model the details of the interface. Compartments in the class shape display information about the attributes, operations, and signal receptions of the interface.

Ball and socket notation, in which the implementation dependency from a classifier to the provided interface is displayed as a circle (ball) and the usage dependency from a classifier to the required interface is displayed as a half-circle (socket). This notation is also called the external view.

Provided interface (circle shape) Required interface (socket shape)

A component class diagram of a provided interface named Component1. A component class diagram of a required interface named Component1.





Use the ball and socket notation, also called the lollypop notation, when you need to model only that provided and required interfaces exist

**Q 12 : Roles , Interfaces and Roles**

Ans: A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

A role is the behavior of an entity participating in a particular context.

For example, Figure shows the interface Employee, whose definition includes three operations. There exists an association between the classes Person and Company in which context Person plays the role e, whose type is Employee. In a different association, the Person might present an entirely different face to the world. With this explicit type, the role the Person plays is more than just a name meaningful to the human reader of this diagram. In the UML, this means that the Person presents the role of Employee to the Company, and in that context, only the properties specified by Employee are visible and relevant to the Company.

Q 13 : Object persistance

Object Diagram

Additional Reference for Assessment.

<https://www.majyori.com/ooad/importance-of-modeling>