Introduction to OOP and UML

This article introduces aspects of OOP relevant to system design, with a focus on encapsulation, abstraction, polymorphism, and inheritance.

We live in a world of objects and behaviors. A dog barks at a cat. A passenger drives a car. When working with software that relates to our world, it makes sense to define our code in terms of the objects and behaviors we are working with.

The introduction of *object-oriented programming*, or *OOP*, was a major shift in software design, allowing us to define our software components as objects. Before OOP, the main language type was *imperative programming*, in which blocks, functions, and files were the main methods of component organization. OOP added new levels of structure on top of imperative programming and introduced four pillars to assist in creating reusable and extensible designs: abstraction, encapsulation, inheritance, and polymorphism.

In this article, we will explain how these pillars influence system design and describe using *Unified Modeling Language*, or *UML*, to represent objects and their relationships.

UML: Describing an Object and Its Relationships

To describe how an object is going to work in our system, we should not need to see every line of code. The names of classes, methods, and attributes should be able to convey the purpose of an object. Working with our objects in high-level diagrams allows us to reason about how our components will work together without having implemented them, or knowing all of the technical details.

The UML base class diagram was designed to document objects at the name, method, and attribute level. Below is an example of a UML class diagram. At the top is the class name, followed by the attributes, and then the methods:

These diagrams are meant to convey what an object is for, as well as how it is going to be interacted with. The diagrams can also give a developer some idea of how it should be implemented.

Multiple Choice Question
Which is **NOT** included in a class UML diagram?
Implementation details
Attributes
Methods
Class name
👏
Correct! UML diagrams represent a class at a high-level, leaving the implementation details up to the code.

System Diagrams and Flowcharts

We can also represent objects at a system level, displaying the relationships between objects. Take a look at the following example of a *system diagram*:

This diagram describes how information flows between the main pieces of the application at a high level. The `Website` receives user requests, which it forwards to a `Server`. The `Server` processes those requests, turning them into `Database` queries. The `Database` sends information back to the `Server`, which packages content to return to the `Website`. These kinds of diagrams can help gain a bird's-eye view of how a complicated system works.

Designing OOP Classes

These diagrams give us a nice way to represent our classes, but how do we ensure these classes are well designed?

Let's start from the main pillars of OOP:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

Abstraction is hiding the low-level details and representing an idea at a high level. When we design an object's methods that will be used by other objects, we want to abstract what the methods will do at the highest level possible.

This allows other classes to only depend on the “big picture” of a class, which is less likely to change over time.

Encapsulation

Encapsulation is highly related to abstraction. Encapsulation means that an object should manage its internal state, exposing only behaviors to be used by other objects. Essentially, only the high-level abstract behaviors of an object should be accessible to other objects, and access to an object’s internals should be restricted.

Inheritance

Inheritance means that an object inherits characteristics of its parent object. Inheritance allows us to implement behaviors shared by subclasses in the parent object, and leave the implementation of behavioral differences to a child object.

Polymorphism

By defining classes using inheritance, we can reuse the aspects of objects that stay the same. With our next pillar, *polymorphism*, we can define the differences between subclasses. Polymorphism allows a class to overwrite the inherited behavior of its superclass. Polymorphism allows the behavior of particular superclass methods to be left up to the child class implementation.

Multiple Choice Question
Which pillar of Object Oriented Programming would be described as hiding an object’s internal implementation details behind higher-level methods?
Polymorphism
Attributes
Inheritance
Encapsulation
👏
Correct!

Review

In this article, we discussed the role that OOP plays in system design as well as the basics of representing OOP systems in UML and system diagrams. We further elaborated on the meaning and purpose of abstraction, encapsulation,

inheritance, and polymorphism. As you read and write more OOP code, you'll see how these tools and concepts help software developers design systems that are more responsive to change.