Introduction to Object oriented programming

Agenda

Key terms

LLD

> Low-level design (LLD) is a component-level design process that follows a step-by-step refinement process. This process can be used for designing data structures, required software architecture, source code and ultimately, performance algorithms. Overall, the data organization may be defined during requirement analysis and then refined during data design work. Post-build, each component is specified in detail

Procedural programming

> Procedural programming is a programming paradigm that uses a sequence of steps to solve a problem.

Object-oriented programming

> Object-oriented programming (OOP) is a programming paradigm that uses objects to model real-world things and aims to implement state and behavior using objects.

Abstraction

> Abstraction is the process of hiding the implementation details of a program from the user.

Encapsulation

> Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

Classes

> A class is a blueprint which you use to create objects.

Object

> An object is an instance of a class.

Why LLD

The goal of LLD or a low-level design (LLD) is to give the internal logical design of the actual program code. Low-level design is created based on the high-level design. LLD describes the class diagrams with the methods and relations between classes and program specs. It describes the modules so that the programmer can directly code the program from the document.

A good low-level design document makes the program easy to develop when proper analysis is utilized to create a low-level design document. The code can then be developed directly from the low-level design document with minimal debugging and testing. Other advantages include lower cost and easier maintenance

Ultimately, LLD has the following goals:

- Low level implementation details of a system
- organization of code
- write good software

What is a good software?

A good software is a software that is

- easy to maintain
- easy to scale.
- easy to extend

Maintainability

Software is not static. If you build a valuable product that works perfectly but is difficult to modify and adapt to new requirements, it will not survive in today's market. Maintainability is a long-term aspect that describes how easily software can evolve and change, which is especially important in today's agile environment.

The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment

ISO 25010 states that a highly maintainable software system must possess the following qualities:

- `Modularity` - The product is composed of discrete components such that a change to one component has minimal impact on other components.
- `Reusability` - The product makes use of assets that can be re-used in building other assets or in other systems.
- `Analyzability` - The impact of an intended change on the product, diagnosis of deficiencies, causes of failures or identification of the components that need to be changed can be analyzed effectively and efficiently.
- `Modifiability` - The product can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- `Testability` - The test criteria can be established effectively and efficiently, and the product can be tested to determine whether those criteria have been met.

Scenarios -

- How easy it is for a new developer to contribute to the product?
- Can I add a new feature to the product without impacting existing functionality?
- Can I get an insight into the product's performance?

Scalability

Software scalability is an attribute of a tool or a system to increase its capacity and functionalities based on its users' demand. Scalable software can remain stable while adapting to changes, upgrades, overhauls, and resource reduction.

Scenarios -

- You just expanded your business to North America and Europe. Will your software be able to handle the users?
- How will your application perform when it has multiple users using it at the same time?
- Does your application provide a good user experience?

Extensibility

> Extensibility is a software engineering and systems design principle that provides for future growth. Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The principle provides for enhancements without impairing existing system functions.

Scenarios -

1. Your application uses PayTM for payment, and you want to add a new payment method. It can be a very simple task based on how your software is structured.
2. You use AWS for deployment and now a client wants to use your application on their own server. Again, will your code be able to handle this?

3. If I change my backend code, will my frontend code be affected?

Extensibility vs Reusability

Extensibility and reusability have many emphasized properties in common, including low coupling, modularity and high risk elements' ability to construct for many different software systems, which is motivated by the observation of software systems often sharing common elements. Reusability together with extensibility allows a technology to be transferred to another project with less development and maintenance time, as well as enhanced reliability and consistency

Programming paradigms

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

There are two major types of programming paradigms:

- `Imperative` - an imperative program consists of commands for the computer to perform to change state e.g. C, Java, Python, etc.
- `Declarative` - focuses on what the program should accomplish without specifying all the details of how the program should achieve the result e.g. SQL, Lisp, etc.

We will be focusing on the imperative paradigm which has the two following subtypes.

Procedural programming

It is based on the concept of the procedure call. Procedures (a type of routine or subroutine) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

Think of all programming as managing the relationship between two fundamental concepts: state and behavior. State is the data of your program. Behavior is the logic.

Procedural Programming is based on implementing these two concepts separately. State is held in data structures. Behavior is held in functions (also known as procedures or subroutines). A procedural application therefore passes data structures into functions to produce some output.

Procedural code thinks of the actions that it has to perform as a series of steps. Imagine you want to transfer some money from one account to another. These are the following steps that you would take:

- Open the source account
- Withdraw the money
- Open the destination account
- Deposit the money in destination account

A procedural version of this program would be:

```
def transfer(source: int, destination: int, amount: int) -> None:

    source_account = get_account(source)
    update_account(source_account, -amount)

    destination_account = get_account(destination)
    update_account(destination_account, amount)

def get_account(number: int) -> dict:
    return list(filter(lambda account: account['number'] == number,
accounts))[0]

def update_account(account: int, delta: int) -> None:
    account['balance'] += delta
```

Object-oriented programming

Object-Oriented Programming is based on implementing the state and behaviour concepts together. State and behaviour are combined into one new concept: an Object. An OO application can therefore produce some output by calling an Object, without needing to pass data structures.

The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into objects that expose behavior (methods) and data (members or attributes) using interfaces. The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an "object", which is an instance of a class, operates on its "own" data structure.

Advantages of OO include the potential for information hiding: if a caller needn't pass any data structure, then the caller needn't be aware of any data structure, and can therefore be completely decoupled from the data format.

```
public class OopBankAccount {
    private Integer number;
    private Integer balance;

    public OopBankAccount(Integer number, Integer balance) {
        this.number = number;
        this.balance = balance;
    }

    void deposit(Integer amount) {
        this.balance  += amount;
    }

    void withdraw(Integer amount) {
        this.balance  += amount;
    }
```

```
    void transfer(OopBankAccount destination, Integer amount) {
        this.withdraw(amount);
        destination.deposit(amount);
    }

}
```

Advantages:

- Reusability: Through classes and objects, and inheritance of common attributes and functions.
- Security: Hiding and protecting information through encapsulation.
- Maintenance: Easy to make changes without affecting existing objects much.
- Inheritance: Easy to import required functionality from libraries and customize them, thanks to inheritance.

Disadvantages:

- Beforehand planning of entities that should be modeled as classes.
- OOPS programs are usually larger than those of other paradigms.
- Banana-gorilla problem - **You wanted a banana but what you got was a gorilla holding the banana and the entire jungle**

Abstraction

> the creation of abstract concept-objects by mirroring common features or attributes of various non-abstract objects or systems of study[3] – the result of the process of abstraction. is a mechanism which represent the essential features without including implementation details

Objects in an OOP language provide an abstraction that hides the internal implementation details. Similar to the coffee machine in your kitchen, you just need to know which methods of the object are available to call and which input parameters are needed to trigger a specific operation. But you don't need to understand how this method is implemented and which kinds of actions it has to perform to create the expected result.

Advantages of Abstraction:

- Abstraction is used to create a boundary between the application and the client code. This could help us to reduce the design and implementation complexity of software.
- In complex software, abstraction helps us separate responsibilities into software entities (classes, method, etc.) that only know the required functionality of each other but not how that functionality is implemented.
- Abstraction maximizes ease of use for the relevant information. In other words, the code with proper abstraction help programmers to quickly make sense of what that code does and what it is meant to be used for. It avoids code duplication and increases code reusability.
- Abstraction allows the programmer to simplify programming and shift focus from implementation details of actions toward the classes, available methods, etc. It will enable the user to avoid writing low-level code.
- It allows the programmer to change the internal implementation of methods or concrete classes without hampering the interface.

- Using abstraction, we can increase the code security as only relevant details will be provided to users.

Encapsulation

> encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods

Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access state values for all of the variables of a particular object. Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object.

Advantages of Encapsulation:

- `Hiding Data` - Users will have no idea how classes are being implemented or stored. All that users will know is that values are being passed and initialized.
- `More Flexibility` - Enables you to set variables as read or write-only. Examples include: setName(), setAge() or to set variables as write-only then you only need to omit the get methods like getName(), getAge() etc.
- `Easy to Reuse` - With encapsulation it's easy to change and adapt to new requirements.

Classes

A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). The user-defined objects are created using the class keyword.

When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

In order to create a class in Java, you need to use the class keyword.

```
class AsiaCup {
    private String name;
}
```

A class is a blueprint which you use to create objects. An object is an instance of a class - it's a concrete 'thing' that you made using a specific class. So, 'object' and 'instance' are the same thing, but the word 'instance' indicates the relationship of an object to its class.

You can create a new instance of a class by using the new keyword and the constructor of the class.

```
AsiaCup instance1 = new AsiaCup();
AsiaCup instance2 = new AsiaCup();
```

Changing the value of a member variable is done by using the dot operator (.) to access the member variable.

```
instance1.name = "Asia Cup 2022";
instance2.name = "Asia Cup 2023";

System.out.println(instance1.name); // Asia Cup 2022
System.out.println(instance2.name); // Asia Cup 2023
```

Reading List

- OOP vs Procedural vs Functional Programming
- Banana-gorilla problem
- Abstraction
- Abstraction in Python
- Abstraction in Javascript