**Software Design**

Table of Contents

[Software Design Levels 3](#_Toc83392778)

[Objectives of Software Design 4](#_Toc83392779)

[Software Design Principles 5](#_Toc83392780)

[SOLID 6](#_Toc83392781)

[Why Use SOLID? 6](#_Toc83392782)

[OOP Concepts 7](#_Toc83392783)

[Classes and Objects 7](#_Toc83392784)

[Encapsulation 7](#_Toc83392785)

[Message Passing and Association 8](#_Toc83392786)

[Abstraction 8](#_Toc83392787)

[Class Hierarchies and Inheritance 8](#_Toc83392788)

[Polymorphism 9](#_Toc83392789)

We initially gathered requirements from our clients and derived the specifications from there. The next step is to design a skeleton of the program to implement those specifications. This is where the **software design** phase comes in.

If we create a program which is impossible to modify and mould based on changing requirements, then that program will eventually become useless, even if it initially fits the requirements perfectly. Such a program is, in fact, worse than one that does not work at all but can be easily modified to get it to work.

To be able to create a program that can be easily modified, we need the design phase. Software design is a mechanism to **transform user requirements**, as described in the SRS, into a suitable form that helps programmers **implement** and **write code** for the software. Thus, we are moving from the **problem domain** into the **solution domain**.

In software design, we consider the system to be a set of components, or **modules**, each with their own behaviour and boundaries.

## Software Design Levels

The software design process has two levels:

1. **External Design** – This focuses on deciding which modules are needed, what their specifications are, what the relationships between them are and what the interfaces between them will be. This is also called **High-Level Design** (HLD).
2. **Internal Design** – This focuses on planning and specifying the internal structures and processing the details of each module. It is also called **Detail-Level Design** (DLD).

## Objectives of Software Design

1. **Correctness** – The software needs to be correct, as per the requirements.
2. **Completeness** – The software must have all the components required, such as data structures, modules, external interfaces, etc.
3. **Efficiency** – The software must use resources efficiently.
4. **Flexibility** – The software must be modifiable to adjust to changing needs.
5. **Consistency** – The software should not have any inconsistencies.
6. **Maintainability** – The software needs to be simple enough that it can be easily maintained by other designers.

## Software Design Principles

There are a few **principles** to software design that aim to help us handle the complexity of the design process. This will reduce the effort needed to design the software and reduce the risk of introducing errors in the design process.

The key software design principles we will be studying are:

1. **DRY** – Don’t Repeat Yourself means that each piece of code should occur exactly once in the system. This helps write scalable, maintainable and reusable code.
2. **KISS** – Keep it Simple, Stupid (mean) means that each small part of the software should be as simple as possible. We should avoid unnecessary complexities as much as we can.
3. **YAGNI** – You Ain’t Gonna Need It means that we should only implement something when we actually need it, not before.
4. **SOLID** – We will be looking into this in depth below.

## SOLID

In Object-Oriented Programming, **SOLID** is an acronym for **five design principles** used to make software understandable, flexible and maintainable. The principles are:

1. Single Responsibility Principle (SRP)
2. Open Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

We will be looking into each of these principles in depth later.

### Why Use SOLID?

Every programmer begins their life writing programs in single files. They do not yet know about the complexities that can arise from having all their code in a single file and the benefits of distributing the work.

Following SOLID can take a programmer from there to dividing everything into small parts and connecting everything up properly so that the program still works perfectly but is far easier to understand, debug and maintain.

## OOP Concepts

### Classes and Objects

A **class** is a template for objects. It defines the **properties** and **behaviours** of the objects and specifies valid and default values for properties.

An **object** is a member, or **instance**, of a class. It has a state in which all the properties have values that are explicitly defined or have the default value as specified by the class. Objects are generated by the class.

An application is designed at the **class level**. We are not concerned about specific objects. This is important to remember. The objects only become involved in the actual implementation.

### Encapsulation

**Encapsulation** is the ability to protect some properties of an object from external entities. Encapsulation is achieved if an object keeps its state **private** from other entities, i.e. they cannot directly access its properties. Instead, external entities can only call **public methods** that the class of the object defines. Thus, the object decides if and how external entities can access its properties.

With regards to accessibility, **attributes** can be **private**, meaning only the methods of the same class can access them, or **public**, meaning methods in any class can access them. **Methods** can be **public**, meaning they can be used by objects of any class, **private**, meaning they can be used only by objects of the same class, or **protected**, meaning they can be used by objects of the same class or a subclass.

### Message Passing and Association

Even though **methods** are defined by **classes**, the classes themselves are not the ones using those methods to communicate, it is the **objects**.

A **static diagram**, like a class diagram, shows different classes and the **logical associations** between those classes. It does not show the movement of **messages**.

An **association** between two classes means those classes **are able to communicate**. The actual communication, again, is performed by objects.

### Abstraction

**Abstraction** is an extension of encapsulation. Maintaining a large program’s code that is changing frequently can become very difficult. Abstraction aims to ease this.

In abstraction, each object exposes only a **high-level mechanism** for using it. It does not reveal the internal implementation details, only operations that are relevant for other objects. This means that one object can use another object without knowing details about it, thus making maintaining the code easier.

### Class Hierarchies and Inheritance

Objects are often similar, sharing common logic, but not exactly the same. To help in this situation, so that we do not end up creating hundreds of classes that are all very similar, we use **class hierarchies**. In fact, every class, except the Object class, has a parent, or **superclass**.

A class could have several ancestors. When defining the class, we also need to define any ancestors. Not doing so makes Object the default ancestor. In turn, every class can also potentially have descendants, or **subclasses**.

**Inheritance** is the process of making a class a descendant of another. This allows the subclass to use properties and methods that have been defined in its superclass, of course, given that the superclass allows it to do so. Essentially, we are **reusing code**.

Classes that inherit from a parent class are said to be **specializations** of the parent class, since they take what the parent class has and add on top of it. Classes that are parents in turn, are said to be **generalizations** of their subclasses.

### Polymorphism

**Polymorphism** refers to the ability of the same method to behave differently when used by different objects. For example, the print function can be called with both strings and with numbers and it will seemingly work the same, but the underlying implementation is actually different depending on the type of object we are printing.

If we inherit a method from a superclass and change its implementation, we are employing polymorphism. The specific type of polymorphism in this case is called **method overriding**. In method overriding, the method signature (name, parameters and return type) have to remain the same, but the code inside can change. The overridden method can have a narrower accessibility, but not a wider one, meaning a private method can be overridden and made public, but a public one cannot be made private.