**Change - The enemy of developers**

* No matter how well an application is designed, over time it will change or grow, otherwise it will die.

**Some reasons for change:**

* New functionality
* Migration to new DB
* Integration with other system – deal with different data format. E.g B2B interaction, OTA use data from suppliers or partners.

**How to cater for change?**

We can achieve reuse, create more flexible designs, and understand the object-oriented paradigm more thoroughly by studying and applying patterns.

But even patterns don’t serve as a guiding set of principles that are universally applicable, and with the proliferation of patterns over the past couple of years, simply finding the most appropriate pattern can be a daunting task. This begs some interesting questions.

* What are the fundamental principles of the object oriented paradigm?
* Is there a set of guiding principles that we can consistently and faithfully apply to help us create more robust systems?

**Reusable object-oriented Design Principles:**

1. **Identify aspects of your application that vary and separate them from what stay same.**

That is, take what varies and encapsulate it so that it won’t affect the rest of your code. So that later you can alter or extend the parts that vary without affecting to those that don’t.

This forms the basis of almost every design patterns. All patterns allow some part of a system to vary independently of others.

1. **Program to an ‘interface’, not an implementation**

The word interface is overloaded - Interface (set of methods and parameters).

Program to an interface actually means program to a supertype (via abstract class or interface).

When inheritance is used carefully (some will say *properly*), all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class.

*All* subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

* Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
* Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class (es) defining the interface.

This so greatly reduces implementation dependencies between subsystems.

1. **Favour Composition over Inheritance(Has-A over Is-A) aka Composite Reuse Principle (CRP)**

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition.

Class inheritance enables the implementation of one class in terms of another's. Reuse by subclassing is often referred to as white-box reuse. The term "white-box" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.

Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or *composing* objects to get more complex functionality.

Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes."

**Advantages of Inheritance**

* Class inheritance is defined statically at compile time and is straightforward to use.
* Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

**Disadvantages of inheritance**

* The implementations inherited from parent class cannot be changed at runtime since inheritance is defined at compile time.
* Parent classes often define at least part of their subclasses' physical representation. Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation". The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.
* Implementation dependencies can cause problems when reusing a subclass. In case any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something more appropriate. This dependency limits flexibility and ultimately reusability. One cure for this is to inherit only from abstract classes, since they usually provide little or no implementation.

**Object Composition**

* Object composition is defined dynamically at run-time through objects acquiring references to other objects.
* Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others. But there is a payoff.
* Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type.
* Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies.
* Favoring object composition over class inheritance helps to keep each class encapsulated and focused on one task. Application classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters.
* On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

1. **Classes should be opened for extension but closed for modification (Open Close Principle)**

A clever application design and the code writing part should take care of the frequent changes that are done during the development and the maintaining phase of an application. Usually, many changes are involved when a new functionality is added to an application. Those changes in the existing code should be minimized, since it's assumed that the existing code is already unit tested and changes in already written code might affect the existing functionality in an unwanted manner.

The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

Software entities like classes, modules and functions should be open for extension but closed for modifications.

1. **Strive for loosely coupled objects that interact**

Loosely coupled designs allow the construction of flexible object oriented systems that can handle change because they minimize the interdependency between objects.

1. **Principle of Least Knowledge – talk only to your immediate friends (Adapter) Law of Demeter**

For an operation O on a class C, only operations on the following objects should be called: itself, its parameters, objects it creates, or its contained instance objects.

Loosely coupled designs allow the construction of flexible object oriented systems that can handle change because they minimize the interdependency between objects.

The Principle of Least Knowledge (PLK) is also known as the Law of Demeter.

The basic idea is to avoid calling any methods on an object where the reference to that object is obtained by calling a method on another object. Instead, this principle recommends we call methods on the containing object, not to obtain a reference to some other object, but instead to allow the containing object to forward the request to the object we would have formerly obtained a reference to.

The primary beneﬁt is that the calling method doesn’t need to understand the structural makeup of the object it’s invoking methods upon.

1. **Dependency Inversion Principle**

In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes. A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes. Since the high level classes are defined in terms of others this seems the logical way to do it. But this is not a flexible design. What happens if we need to replace a low level class?

In order to avoid such problems an abstraction layer can be introduced between the high level classes and low level classes. Since the high level modules contains the complex logic they should not depend on the low level modules and that the new abstraction layer should not be created based on low level modules. The low level modules are created based on the abstraction layer.

According to this principle the way of designing a class structure is to start from high level modules to the low level modules:  
High Level Classes --> Abstraction Layer --> Low Level Classes

To sum up the Intent of dependency inversion is as follows:

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

The Dependency Inversion Principle formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level. In our own designs, attempting to couple at the abstract level can seem like overkill at times. Pragmatically, we should apply this principle in any situation where we’re unsure whether the implementation of a class may change in the future. But in reality, we encounter situations during development where we know exactly what needs to be done. Requirements state this very clearly, and the probability of change or extension is quite low.