**DESIGN PATTERNS**

Are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

You can’t just find a pattern and copy it into your program, that way you can with off-the shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is more a high-level description of a solution. The code of the same pattern applied to different programs may be different.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

Here are the sections that are usually present in a pattern description:

* **Intent** of the briefly describes both the problem and the solution.
* **Motivation** further explains the problem and the solution the pattern makes possible.
* **Structure** of classes shows each part of the pattern and how they are related
* **Code** example in one of the popular programming languages makes it easier to grasp the idea behind the pattern

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

**Why should I learn patterns?**

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern.

* Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing pattern is still useful because it teaches you how to solve all sort of problems using principles of object-oriented design.
* Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that”, and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern.

**Classification of patterns**

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.

The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.

The most universal and high-level patterns are architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

Mainly patterns are classified as:

* **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
* **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
* **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

**Creational patterns.**

* **Factory method**
* **Abstract factory**
* **Builder**
* **Prototype**
* **Singleton**

**Structural patterns.**

* **Adapter**
* **Bridge**
* **Composite**
* **Decorator**
* **Facade**
* **Flyweight**
* **Proxy**

**Behavioral patterns.**

* **Chain of Responsibility**
* **Command**
* **Memento**
* **Observer**
* **Iterator**
* **Mediator**
* **State**
* **Strategy**
* **Template Method**
* **Visitor**

**FACTORY METHOD**

**Intent:** Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

Suggests that you replace direct object construction calls with calls to a special factory method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.

This pattern allows you to override the factory method in a subclass and change the class of products being created by the method.

There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

*Applicability*

* **Use it when you don’t know beforehand the exact types and dependencies of the objects your code should work with**.

Separates product construction code from the code that uses the product. Therefore, it’s easier to extend the product construction code independently from the rest of the code.

* **Use it when you want to provide users of your library or framework with a way to extend its internal components.**

Inheritance is probably the easiest way to extend the default behavior of a library or framework. But how would the framework recognize that your subclass should be used instead of a standard component itself.

* **Use it when you want to save system resources by reusing existing objects instead of rebuilding them each time.**

You often experience this need when dealing with large, resource-intensive objects such as database connections, file systems, and network resources.

*Pros*

You avoid tight coupling between the creator and the concrete products.

Single responsibility principle. You can move the product creation code into one place in the program, making code easier to support.

Open/closed principle. You can introduce new types of products into the program without breaking existing client code.

*Cons*

The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best-case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes.

**ABSTRACT FACTORY**

**Intent.** Is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

*Applicability*

* **Use it when your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products-they might be unknown beforehand, or you simply want to allow for future extensibility.** This pattern provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don’t have to worry creating the wrong variant of a product which doesn’t match the products already created by your app.

Consider implementing it when you have a class with a set of factory methods that blur its primary responsibility.

In a well-designed program each class is responsible only for one thing. When a class deals with multiple product types, it may be worth extracting its factory methods into a stand-alone factory class or a full-blown abstract factory implementation.

*Pros.*

You can be sure that the products you’re getting from a factory are compatible with each other.

You avoid tight coupling between concrete products and client code.

Single responsibility principle. You can extract the product creation code into one place, making the code easier to support.

Open/closed principle. You can introduce new variants of products without breaking existing client code.

*Cons*

The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

**BUILDER**

**Intent.** It lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

This pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

To create an object, you execute a series of these steps on a builder object. The important part is that you don’t need to call all the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

You can go further and extract a series of calls to the builder steps, you use to construct a product into a separate class called **director**. The director class defines the order which to execute the building steps, while the builder provides the implementation for those steps.

Having a director class in your program isn’t’ strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across the program.

In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

*Applicability*

* **Use it to get a rid of “telescopic constructor”.**

Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters. After applying the pattern, you don’t have to cram dozens of parameters into your constructors anymore, this is because, the pattern use only the steps that you really need to create objects.