**Design Patterns**

* **Introduction:**

All the design patterns are classified into following three categories.

- Creational

- Structural

- Behavioural

**Creational:**

Creational patterns deals with the process of creation of objects and classes.

**Structural:**

Structural patterns deals with how classes and objects are arranged or composed.

**Behavioural:**

Behavioural patterns describe how classes and objects interact and communicate with each other.

* **Creational Patterns:**

Creational patterns deals with the process of creation of objects and classes.

Following are the patterns we are going to cover:

Builder

Simple Factory

Factory Method

Prototype

Singleton

Abstract Factory

Object Pool

* **Builder Pattern:**

What problem builder design pattern solves?

**“Class constructor requires lot of information”**

If you need the object is immutable.

Note: The object values should not change once the object is created.

Example:



The problem with above class is it has many parameters and most of them are with same type.

This will be confusing for the user, they have to know the order of it and there is a chance to pass shipCode value to shipVolume as both are same data type.

Builder pattern helps all these issues.

Another problem Builder pattern solves:

When you have many number of parts (Address, Role) objects to assemble the User object, then we can go for Builder pattern.

**“Objects that need other objects or ‘parts’ to construct them”**

****

What is Builder?

We have a complex process to construct an object involving multiple steps, then builder design pattern can help us.

In builder we remove the logic related to object construction from client code and abstract it in separate class.

UML of builder:

A screenshot of a cell phone

Description automatically generated

Role - Product:

Final complex object that we want to create.

Role – Builder:

Provides interface for creating “parts” of the product.

This is responsible for creating parts of the product and building the final object.

Builder will have a method to check the details of created parts.

Role – Concrete Builder:

Constructs parts and assembles final product.

Keeps track of product it creates.

Role – Director:

Uses builder to construct object.

Knows the steps and their sequence to build product.

**Builder implementation steps:**

* We start by creating a builder
  + Identify the “parts” of the product and provide methods to create those parts.
  + It should provide a method to “assemble” or build the product/object.
  + It must provide a way / method to get fully built object. Optionally builder can keep reference to an product it has built so the same can be returned in future.
* A director can be a separate class or client can play the role of director.

Builder – Implementation and Design Consideration

* You can easily create an immutable class by implementing builder as an inner static class. You will find this type of implementation used quite frequently even if immutability is not a main concern.

Design consideration:

* The director role is rarely implemented as separate class, typically the consumer of the object instance or the client handles that role.
* Abstract builder is not required if “product” itself is not part of any inheritance hierarchy. You can directly create concrete builder.
* If you are running into a “too many constructor arguments” problem then it’s a good indication that builder pattern may help.

Builder Examples:

* The java.lang.StringBuilder class as well as various buffer classes in java.nio package like ByteBuffer, CharBuffer are often given as examples of builder pattern.
* In my opinion they can be given as examples of builder pattern, but with an understanding that they don’t match 100% GoF definition. These classes do allow us to build final object in steps, providing only a part of final object in one step. In this way they can be thought of as an implementation of builder pattern.
* So a StringBuilder satisfies the intent / purpose of builder pattern. However as soon as we start looking at structure of the StringBuilder things start to fall apart. GoF definition also states that, builder has potential to build different representations of product interface using same steps.
* There is another great example of builder pattern in Java-8. The java.util.Calendar.Builder is a builder.

Comparison with Prototype:

|  |  |
| --- | --- |
| Builder | Prototype |
| We have complex constructor and builder allows us to work with that. | Prototype allows altogether skip using constructor |
| We can create a builder as separate class and so it can work with legacy code. | In Java this pattern works using clone method, and needs to modify existing code so may not work with legacy code. |

Builder Pitfalls:

* A little bit complex for new comers mainly because of ‘method chaining’, where builder methods return builder itself.
* Possibility of partially initialized object; user code can set only a few or none of properties using withXXX methods and call build(). If required properties are missing, build method should provide suitable defaults or throw exception.

Summary:

* Think of builder pattern when you have a complex constructor or an object is built in multiple steps.