Abstract Factory

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.
- Supporting new kinds of products requires extending the factory interface, which involves changing the Abstract Factory class and all of its subclasses.

Design Patterns

Toukir Ahammed
Lecturer
Institute of Information Technology (IIT)
University of Dhaka
Email: toukir@iit.du.ac.bd

Factory

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 the 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.
- Clients might have to subclass the Creator class just to create a particular Concrete Product object
- The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

Singleton

Pros

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it's requested for the first time.
- Controlled access to sole instance
- Permits a variable number of instances.

Cons

- Violates the Single Responsibility Principle. The pattern solves two problems at the time.
- The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

Singletons are Pathological Liars

Drawbacks of the Singleton

Prototype

Pros

- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.
- You get an alternative to inheritance when dealing with configuration presets for complex objects.

Cons

- Cloning complex objects that have circular references might be very tricky.

Builder

Pros

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

Cons

- The overall complexity of the code increases since the pattern requires creating multiple new classes.

Adapter

Pros

- *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

Cons

- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest of your code.

Adapter

Class Adapter

- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- lets Adapter override some of Adaptee’s behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

Object Adapter

- lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Facade

Pros

- It promotes weak coupling between the subsystem and its clients. You can isolate your code from the complexity of a subsystem.
- It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It doesn't prevent applications from using subsystem classes if they need to. Thus, you can choose between ease of use and generality.

Cons

- A facade can become a god object coupled to all classes of an app.

Proxy

Pros

- You can control the service object without clients knowing about it.
- You can manage the lifecycle of the service object when clients don’t care about it.
- The proxy works even if the service object isn’t ready or is not available.
- *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.

Cons

- The code may become more complicated since you need to introduce a lot of new classes.
- The response from the service might get delayed.