Murad Abdi

Professor Woolfolk

SYCS 363

April 22, 2022

The principles of object-oriented design are important to understand when programming. There are three main classifications of patterns: creational, structural, and behavioral. These classifications each contain numerous sub-designs. Knowing these different design patterns is important for any programmer, as they provide a way to flexibly and efficiently create code. In this essay, I will examine two designs from each of these classifications and discuss their unique advantages and drawbacks.

Creational patterns involve creating objects with great flexibility. Two examples of creational patterns are singletons and prototypes. A prototype is a fully initialized instance that can be copied or cloned, while a singleton is a class of which only a single instance can exist.

The singleton design pattern involves a class that is responsible for creating a single object and providing a global access point to it, all while ensuring that no other copies of the object are created. This is advantageous because it allows an object to be used repeatedly without having to create multiple copies of it, filling up bandwidth on a redundant task. However, its drawbacks include the difficulty it faces with unit testing because of its global state and multithreaded environments because of the precautions that have to be taken to ensure that the object won’t be created multiple times. Singletons are useful in tasks where mitigating redundancy can save a lot on resources like memory. For example, if a streaming platform like Twitch needed to reinitialize a connection between the video broadcaster and a video receiver for every frame, the latency would be magnitudes higher than what it currently is.

The prototype design pattern is useful when we want to instantiate an object but see that it will take too many resources or cost resources that aren’t available, like memory. We still need to use this object, so what we can do is create a new object from an existing copy that still satisfies the needs of our current project. Here’s an example: all webpages have some kind of CSS stylesheets. Whenever a web page is opened, the stylesheet is rendered onto it, giving it the look that it has. This stylesheet is a file that is downloaded the first time you open a webpage, and is saved to your browser’s local memory. This reduces the need to re-download the file from the server every time you open the webpage, which saves bandwidth on the server’s end and allows you to enjoy a faster user experience. If you clear your browser’s local memory, the file will have to be re-downloaded again and the next time you open the web page will be a bit slower, but just once. If browsers didn’t apply the prototype design pattern to stylesheets in this manner, you would experience slower web page speeds all the time. This is what makes prototypes so useful. On the technical side, one drawback of the prototype design pattern is that it hides the concrete implementation information of the class with abstraction. An advantage is that it grants independence from an existing object, allowing us to implement a new object without relying on an old one.

Structural patterns concern the relationships between objects. Two examples of structural patterns are bridges and flyweights. A bridge is a way to connect two pieces of software so they can communicate with each other, while a flyweight is a lightweight object that can be used in place of a heavier object.

The flyweight is a structural design pattern that is useful for reducing memory use and increasing the number of objects that can be created in the process. It does this by caching data used by various objects and sharing that data between them, effectively limiting redundancy. A good analogy for the flyweight design pattern is that it operates like a factory--flyweight objects themselves are immutable, so it's the pattern that handles all the real assembly work. Whenever an object is to be created, the flyweight first checks its repository of previously created objects to see if it already exists. If it does, it returns the existing object. If it does not, it creates a new object and saves it for future reference. Flyweights are great objects that would be needed very often but bad for a set of objects that won't be created more than once or twice. Creating the first of an object with a flyweight design pattern is slower than just trying to create the object by itself because the data of it also has to be stored in a data structure that the flyweight can traverse. Here's an example: if Apple's iPhone manufacturers receive an order to create an additional 1000 gray iPhones, all they need to do is let the assembly line run and change nothing, since gray iPhones are the most common color and they already have the materials for it. If they receive an order for a color that they've never manufactured before like orange, they'll have to stop the assembly line, make room for a machine part that will add that color, then start the assembly line back up. Altering the assembly line just to create the first orange iPhone is a slower process than if you were to just set out to color an orange iPhone by hand, but once the assembly line is up and running again, it will be able to manufacture orange iPhones faster than a human could, saving valuable time and money.

The bridge design pattern is a structural pattern that is useful when you need to separate abstractions from their implementations. In short, an abstraction is the intended functionality of an object while the implementation is the wiring and plumbing behind the wall that make it work. Here's an example: when we go inside our homes, we're concerned with having the lights on when we want them to and having a temperature that feels comfortable. We hardly consider the vents that move cold air and hot air around or the light bulbs producing the light around us.

These are the implementations. We just feel the temperature and see a well-lit room, which are the abstractions. The bridge design pattern enables modularity that allows for flexible implementations that can be applied to different things, while abstractions can be reached with many different implementations. The heating and cooling system you have in your car is completely different and far more compact than the one present in homes, but the abstraction is the same--you feel a temperature that's just right. The bridge design pattern is great because it allows implementations to be hidden from clients and interfaces, but using it would be a bad idea if you want the implementation to be accessible and a close part of the abstraction.

Behavioral patterns consider the way objects interact with each other. Two examples of behavioral patterns are command and interpreter. Command is a pattern that allows you to issue requests to an object without knowing the details of how the object will fulfill the request. Interpreter is a pattern that allows you to translate one representation of data into another.

The command design pattern is useful when a request is passed to a program but the object that should be executed is not immediately clear. Under the command pattern, a request is encapsulated within an object, essentially creating a "trigger" that can be used for different uses by different objects. An example is a light switch--a on/off light switch is only a light switch because it is wired to the fixtures, but it can be wired to anything else that uses an on/off switch. The switch itself can be changed to a slider or a button or back to a switch, but its core instruction doesn't change. It can toggle the state of compatible electronics from on to off because they all understand the same instruction.

The interpreter design pattern is useful for creating grammatical representations for programming languages that allow machines to quickly translate them from one format to another, all while retaining the core instruction at hand. A great example of this design pattern is the Java compiler, which interprets the Java source code into machine-readable byte code that the JVM (Java Virtual Machine) can understand. This pattern isn’t widely used and its best use cases are limited to language development, giving it a strong advantage in that field with few applicable uses elsewhere.

All design patterns serve unique purposes and can sometimes overlap. Understanding the principles of these can help one choose the best pattern for your needs.