Design Patterns: The Builder Pattern

The design pattern I chose to analyze in this article is the “Builder” pattern within the Java language. The builder pattern is used to create an object made up of other objects. It allows users to “separate the construction of a complex object so that the same construction process can create different representations [1].” This is useful when the user wants the creation of specific parts to be independent of the main object. The pattern allows a “client object to construct a complex object by specifying only its type and content, being shielded from the details related to the object’s representation [2].” The builder pattern is a creational pattern, meaning that it “deals with object creation mechanisms, and trying to create objects in a manner suitable to the situation [1].”

Some objects require complex assembly, or sometimes an “application may need to create the elements of a complex aggregate [1].” To relieve the client of the burden of having to build their own objects, the builder pattern can “provide the mechanism for building these complex objects [2].” In order to achieve this, in Java, the builder pattern is implemented using four major participant classes. The Product class represents the final product object being built. The Builder class “specifies an abstract interface for creating parts [2]” of the final product object. The ConcreteBuilder class implements the Builder interface and “puts together parts of the product while providing an interface for saving the product [2].” Finally, the Director class “constructs the complex object” using the Builder interface. “When the client calls the main method of the application, it initiates the Builder and Director class [2].” The Director class typically receives a Builder object as a parameter from the client. Thus, it is “responsible for calling the appropriate methods of the Builder class [2].” This allows the final product object to be built in an efficient manner.

A simple example that can be used to explain how all these components work in conjunction with one another is a vehicle manufacturer. The manufacturer has a set of parts, from which he can build a car, truck, or motorcycle. In this case, the person building the vehicle will play the role of the Builder class. He “specifies the interface for building any of the vehicles, using the same set of parts and a different set of rules for every different type of vehicle [2].” For each of the objects under construction, “the ConcreteBuilders will be the builders [2].” Obviously, the vehicle built takes the role of the final product object, while the “Director is the manufacturer [2].” Since all vehicles must have certain components such as an engine, brakes, and headlights, the builder design pattern can help simplify the client’s process of creating a vehicle.

The builder pattern is extremely useful in reducing complexity of creating a product object by “parsing a complex representation, and creating one of several targets [1].” My stance on this topic is that due to its ability to reduce complexity, the builder pattern is an appropriate pattern to use in this context; however, one must make the distinction between other similar patterns. Another creational pattern very similar to the builder is the Abstract Factory pattern. For the builder, the “Builder class is instructed on how to create the object and then asked for it, but the way the class is put together is up to the Builder class [2].” In the abstract factory case, “the client uses the factory’s methods to create its own objects [2].” The abstract factory pattern is very simple. The principle difference between the two is that the abstract factory pattern “requires the entire object to be built in a single call, with all the parameters passed in on a single line [3].” Thus, the builder is beneficial when the product object cannot be produced in a single step.

I believe that the simplicity of the abstract factory pattern can be used within the builder pattern to make it even more powerful. “Sometimes creational patterns are complimentary: Builder can use one of the other patterns to implement which components get built [4].” This will allow clients to make use of both patterns and take advantage of what both of them have to offer.

References

[1] A. Shvets. (2015). Design Patterns Explained Simply. [On-line]. Available: https://sourcemaking.com/design_patterns/builder [June 18, 2016].

[2] “Builder Pattern.” Internet: http://www.oodesign.com/builder-pattern.html [June. 16, 2016].

[3] “Builder Design Pattern in Java.” Internet: https://myjavalatte.wordpress.com/tag/builder-pattern-vs-factory-pattern/, Oct. 22, 2014 [June. 18, 2016].

[4] “What is the difference between Builder Design Pattern and Factory Design Pattern?” Internet: http://stackoverflow.com/questions/757743/what-is-the-difference-between-builder-design-pattern-and-factory-design-pattern, Apr. 16, 2009 [June. 18, 2016].