Jacob Hidalgo

IS 683 Design Pattern HW

A design patterns are pieces of code In various languages that give us a reusable solution to very common problems in designing applications and software. A piece of code becomes a pattern after it is proven that it solves a problem, can be implemented in various scenarios, and the structure is identifiable to the user. One important feature to remember with a design pattern is that these are not exact solutions, as the usability of the patterns is almost infinite. These patterns are instead templates to how to solve a problem or improve the efficiency of our code.

Design patterns provide developers with a means to improve their overall software by having statements organized in a fashion to increase load time, and can reduce the overall file size. By using these patterns, you can prevent the smaller issues in the beginning of the development process instead of letting them accumulate and become a huge problem later on. The design patterns are also reviewed thoroughly by large groups of developers, so many patterns are proven throughout the community to be of great use. Finally, the design patterns are also updated as newer languages or even better methods are developed and tested to provide better solutions to problems.

A creational pattern is a design pattern that focus on the creation of objects and ensures that they are created in a certain way to improve efficiency. Objects that are created in a program in a typical manner may not be coded the best way, and these can bring problems later down the road when they are being processed by both the server side and client side of an application. Through the use of creational design patterns, these processes can be controlled through various types of patterns such as a constructor, factory, or prototype pattern. A constructor pattern is a special method to create a new object that can use and accept arguments. The constructor can then set values to the properties of the object and can also create methods to call the object. A factory pattern provides a generic interface for creating objects. This allows the developer to implement all of the details of the object, by basically creating an object schema. You can preset values and can create objects without having to create a “new” statement.

A structural pattern deals with how an object is composed, what values the object has, and what the relationships are with this object to another object. This helps in the fact that when a new part of the system is created or something new is being added, the entire system does not need to change. A popular structural pattern is a decorator. A decorator can add alternate values to an object that is already created, past the initial values that were set. The way I think of the structural patterns is similar to a traditional relational database, where the table’s fields (objects) have relational properties with other tables (foreign keys). If a new field needs to be entered into a table, you can create it, it’s relation to another field (if applicable) and you can assign values to pre-existing records (objects).

A behavioral pattern focuses on improving communication and translation of object’s values between different objects within a system. When things are not logically ordered in a system, various bugs and processing errors can occur. This is done through various patterns such as an observer pattern. This pattern works by having an object send a notification to the other objects. If the object does not care or want to hear the notification, it will stop listening to what the object is saying.

For my design pattern example, I decided to create a small student database that has a couple of students and attributes for those students. For this example, I used constructors, prototypes, and decorator patterns to create the data. The constructor and decorator patterns work well together because it gave me the ability to add values and attributes to pre-existing objects without having to edit the initial object values. In this case, I created a student object that asked for the student’s name, age, and grade. However, later on in the example, I added a new student and decided that it would be helpful to know the student’s gender. With the decorator, instead of editing the initial values created in the student object, I used a prototype with the decorator to create the value for gender. This allowed me to add the gender value to the new student and through the use of the prototype, I could also add it easily to the other pre-existing students before the gender value was added. The decorator patterns were created to tackle the initial problem of extending the values and capabilities that an object has from their initial purpose, which is what it is doing in a smaller scale in my example. Decorators can work in factory models, constructors, and other types of creational patterns. I chose these couple of patterns as they seem to work together the best through the example in the design pattern book, and to someone who is familiar with relational data basing, the concept seems very familiar as well.