Matthew Kachar

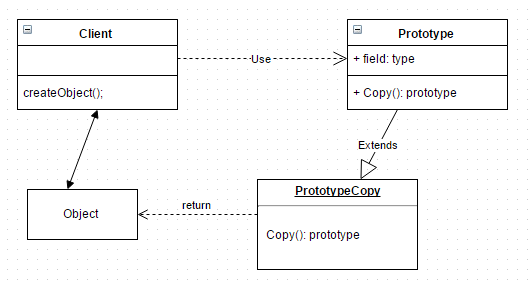
COSC 2100

4/30/15

Design Patterns

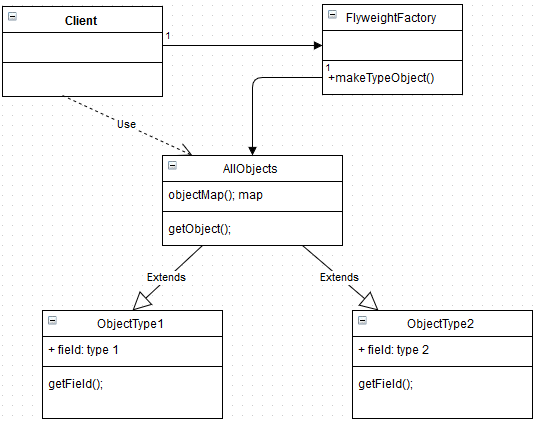
Utilizing specific software design patterns can be very beneficial during development, saving the programmer time by offering a template for unique problems. And certain design patterns can easily be tailored to handle any anticipated expansions or maintenance. Most situations fall into one of three basic categories, each of which has several subcategories. The main three categories are creational, structural and behavioral design patterns.

Creational design patterns contain a function for object creation, while maintaining flexibility by allowing certain components of the object to be instantiated by other classes. For instance, the builder design pattern was proposed to deal with complex construction of an object. Instead of dealing with large constructors and unwieldy parameters, a builder object is created which will then merge other objects to create the final product. Each object represents a different option during the construction process and often utilizes interfaces for flexibility. Some object creation can be so extensive that the process becomes costly. For such situations, the prototype creational design pattern can be implemented, which uses a copy of an existing object as a mold for the construction of new objects. Moreover, every object also has a copy feature which allows it to also be used as a potential prototype for the creation of any new object.



Thus, only the changes that need to be made to the new object can be made without constructing an object from the ground up.

Structural design patterns rely on class relationships to provide not just the framework of the program, but much of the function as well. To a certain degree, structural designs yield significant latitude for customization of interfaces. Much like the builder creational design pattern, the composite structural design pattern utilizes the construction of complex objects by means of primitive objects. The difference being composite design allows whole groups of objects, no matter what their significance, to be accessed without exception—through these established relationships. A good example of this might be a resize or zoom function for an application where the user executes a command to objects in an entire selection or window. It might be necessary to use this composite design for such a function since it would involve having a selection object that contains the other selected objects, all of which have the ability to be manipulated. The flyweight structural design pattern is another style of implementation for complex objects or large quantities of objects. In addition to relationships between other objects, some of the common properties of each object become objects themselves. Since these properties do not need to be instantiated, because a common method can instead point to one object to provide data for a large group of objects, space can be saved within the program.



With the flyweight design pattern every new object is organized by specific properties within a group and those properties become part of another class object. Another interesting structural design pattern is the decorator design. With this design class functions can be extended during runtime. For instance, when a window is opened it may not require scroll bars, but sometimes a horizontal or vertical scroll bar function needs to be added to the window while the program is running or when a new window is opened. The window’s implementation would utilize a decorator class that would add a particular subclass’ function to the superclass object based on some parameters.

When flexible communication is required between objects without concern for rigid class structure, behavioral design patterns are used. The construction of objects with some behavioral designs is entirely dependent on the construction of other objects or implementations. The template method is an example of such a pattern. To illustrate, consider the construction of a car. Certain parts of the car need to be built before others can be installed, such as the wheels, which must have achassis to be mounted on. Thus, the order of implementation is important and the connection between objects depends on this order. The mediator pattern is another method which leaves lose connection between objects but is much more flexible. A mediator object is the only connection between two objects and therefore, properties of the objects can change freely without impacted other objects directly. In fact a lot of the behavioral design patterns work from a similar concept were direct connection is removed. The advantages this type of design are clear with the memento pattern. This design allows objects to store a particular configuration as a restoring point while being manipulated. With the memento pattern a caretaker and originator object are created. The originator is the object itself and the caretaker is the object in a desired configuration which the originator can return to. Any undesired interaction consequences with the originator can be undone.

Each of these different design patterns has its advantages, but employing several designs in the development of large complex software is perhaps the best way to create robust, yet intuitive, applications. And much of these smaller subcategories of design patterns can undoubtedly be coupled with other patterns seamlessly. It is not difficult to see how the builder pattern and mediator pattern are somewhat related or could perhaps be used to synthesize a new design type.