Chris Cargile 09/19/2013
CSCI603 Assignment 2: “Upload Assignment: Design Patterns”

How Design Patterns Solve Design Problems

Object-oriented design patterns (OODP) attempts to lessen the burden of re-writing code by making software that is more reusable. By abstracting software into components which often have no real counterpart in the physical world, OODP provides a channel through which communications between various types of objects can be held to a contract requiring certain characteristics or behaviors be adhered to. An example of such is found in a design pattern concept that is key to OODP, which is polymorphism. Polymorphism allows the machine to associate requests from the client with objects in a manner that is agnostic of the type of object being called upon to execute an operation/request by mandating only that the object fulfill a certain contract known as its interface. In fact, to achieve the software concept of polymorphism and accommodate various design patterns, the most capable OOP languages will facilitate development with language functionality which includes support for interfacing and inheritance (multiple is better because it allows support for mixins).

Depending on where the system weighs in terms of trying to plan for reuse and its inclination towards trying to remain prepared for any change in circumstances, it may make more sense at times to use inheritance versus composition as a design pattern approach. While inheritance may make it easier to implement changes up-front by specifying the implementation of a behavior & class at compile-time, reducing the locations in which a developer may have to search to locate the class he desires to change to affect the new design consideration. However, while subclasses can override the operations of their parent, generated at compile-time simply by polymorphic-overrides (which take effect at run-time) ,composition remains a more advantageous approach to overriding, often times, because dynamic binding and run-time overrides afford the designer leverage in the struggle to ensure reusability by breaking the dependency between a subclass and its parent by freeing the subclass from the implementation details of its parent. Further, as composition helps to preserve encapsulation and focus being directed at a single task (GOF, 19), the Single Responsibility Principle of software design becomes more tractable. Nonetheless, if it is inevitable that subclassing through composition will not lend itself to a sufficiently functional concrete class, than a class founded on a combination of the two implementation OODP/software engineering mechanisms described above may be necessary and reveals that they are a compliment to one another.

Even with the abundance of tradeoffs and considerations already delaying development efforts, there remain additional exceptions to the ‘framework’ of rules mentioned above. The remaining OODP (or more strictly software engineering/language) mechanisms available for making code more reusable are delegation and parameterized typing (known in Java as generics).

There are multiple causes for redesign which are addressed and potentially solvable as discussed in the GOF’s “Design Patterns”, and upfront OODP considerations can circumvent future expenditures by warding off a need to redesign, in the first place, and by enhancing reusability upfront.