



## COMPUTING INTEGRITY

INCORPORATED

60 Belvedere Avenue  
Point Richmond, CA 94801-4023  
510.233.5400 Sales  
510-233.5444 Support  
510.233.5446 Facsimile



### **Object-Oriented Design Patterns: An Introduction<sup>1</sup> 20 January 2010 Thomas Mercer-Hursh, Ph.D.**

In reading about Object-Oriented Programming, one is likely to encounter frequent references to one or another “design pattern”. While the names of some of these patterns may seem to be intuitively clear in intent from the context, the full intent and purpose of the pattern is not likely to be appreciated without some background. It is the intent of this document to provide a very quick introduction to the subject for those working in Object-Oriented ABL.

What is a “design pattern?” The concept and form of design patterns was originated by Christopher Alexander<sup>2</sup> in the field of architecture, but has since been adopted in a number of other disciplines, particularly computer science and especially by those doing object-oriented programming<sup>3</sup>. The essential idea of a design pattern is that it is a formal statement of a common problem along with any conditions and a statement of a solution to the problem. It is not a specific piece of code, but rather the guidelines or recipe for how one can solve the problem. Much as one seeks in OO design to abstract the invariants of sets of problem space entities, patterns abstract the invariants in sets of problem solutions.

Patterns often have an aspect of obviousness about them; i.e., in reading the pattern one may think, “I’ve done that a million times”. However, even if familiar, providing the solution with a name and a formal statement promotes communication and can provide some rigor to what was previously a vague or poorly defined idea. Of course, not every pattern is obvious to every person, so they also have an important rôle in educating those new to the field and in reminding more experienced practitioners of options which they might forget if not recently used.

There are a number of books which have been published about patterns in computer science. Certainly the most famous of these is the so-called Gang of Four (GoF) book<sup>4</sup>. This book also contained early statements about some other core OO design principles including the notion of “programming to the interface” and is recommend reading for any programmer. For the OOABL programmer working on modern distributed architectures, one might also recommend Enterprise

---

<sup>1</sup> See also, Mercer-Hursh, Thomas (2010) *Object-Oriented Vocabulary: An Introduction* for brief definitions of the words used in these patterns.

<sup>2</sup> Alexander, Christopher (1977). *A Pattern Language*. Oxford University Press.

<sup>3</sup> While the same design patterns can actually be applied in almost any programming language or idiom, they have taken particular hold in object-oriented programming. This may be because there is a strong emphasis on re-use in OO and the essence of design patterns is re-use of knowledge. Also, Object Orientation is all about abstracting from the problem space, so one already has the conceptual mechanisms and infrastructure for design patterns, which are themselves abstractions of sets of problem solutions.

<sup>4</sup> Gamma, Helm, Johnson & Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Integration Patterns<sup>5</sup> as a seminal work for enterprise application architecture. The current document will briefly review the GoF patterns as an introduction to the subject. These patterns will be presented in three categories:

- Creational – patterns for creating objects;
- Structural – patterns for relationships between objects; and
- Behavioral – patterns for the behavior of objects.

## Creational Patterns

### Abstract Factory

In OO software, a Factory is the place in the code where an object is created. An Abstract Factory is a pattern in which the factories of related objects are combined into an abstract class to separate the creation of objects from their use. A client which needs an object of a particular type uses a generic interface to the Abstract Factory to request an object of a particular type and the Abstract Factory takes the responsibility for creating and delivering that object to the client. The Abstract Factory may be specialized to produce the same object sets with different implementations. E.g., one might have an Abstract Factory to produce documents such as OrderConfirmation, PickingSlip, and Invoice and a subclass of this Abstract Factory might produce different versions of these same documents for internal versus external use. A single interface would be used to request the documents and the correct form would be delivered according to which subclass had been instantiated.

### Builder

Builder is a pattern which addresses the problem of constructing complex objects. By decomposing the process of construction into multiple steps, one can use alternate implementations at each step in order to create a complex range of objects from a common basic pattern. Builder differs from Abstract Factory in that Abstract Factory is basically a one step process for related objects while Builder is a multi-step process for complex objects which may be composed of multiple subcomponents. The Builder pattern is typically implemented using an abstract Builder class to provide the interface, a ConcreteBuilder to provide the implementation, and a Director class to manage the structure.

### Factory Method

The essence of the Factory Method pattern is to isolate object creation into a method. This can provide several benefits. If the class has subclasses, then each subclass of the base class can override this method to provide the appropriate implementation for that subclass. If there are multiple ways in which the class can be implemented, then using multiple Factory Methods can provide clarity in naming. E.g., if one might create an Order from either database information or from information received in an XML method, one could provide an overloaded constructor in Order with these variations in parameters or one could provide Factory Methods CreateOrderFromDB and CreateOrderFromXML which would make the purpose and context of each clear and unambiguous.

### Prototype

In the Prototype pattern, a new object is created via a clone of an existing object rather than by creating a new object from scratch. The object provided for cloning is the prototype or prototypical

---

<sup>5</sup> Hohpe, Gregor & Woolf, Bobby (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

instance. This pattern avoids the need for subclasses because a single creator can produce multiple objects based on different prototypes and is useful when the cost of creating a clone is inherently less expensive than the cost of creating a new object. Thus, one might, for example, clone an existing object, apply some series of transactions, and then replace the original object with the modified one.

### **Singleton**

In the Singleton pattern, one restricts instantiation of a given class to a single object. This is useful when either the nature of the class is such that only one instance should exist, e.g., where the object performs some coordination rôle among other objects. Singletons are often used to introduce global state into a system, but this can be overused or abused; consequently, there are some who regard Singleton as an anti-pattern<sup>6</sup>. Singleton is really only relevant when the problem space dictates that only one entity can exist at a time but in the problem solution flow of control there are multiple opportunities for creating the entity.

## **Structural Patterns**

### **Adapter aka Wrapper**

An Adapter or Wrapper translates one interface into another so that two objects which could not otherwise work together based on the original interface can cooperate based by means of the Adapter. The Object Adapter pattern is used at runtime via a class which translates the properties and methods of the original interface to the desired properties and methods. In this usage, the Adapter contains an instance of the class which is being adapted, hence the term Wrapper. The Class Adapter pattern uses a class which inherits or implements both the original and desired interface. In languages such as OOABL, which does not support multiple inheritance, the desired interface must be implemented as a pure interface.

### **Bridge**

The purpose of the Bridge pattern is to decouple the abstract part of a problem from the implementation part of a problem. In that, it can be thought of as simply doing the correct decomposition of the problem space, but as it is a common requirement, it merits identification as a pattern. A classic example of the use of the Bridge pattern is any component in which there is a portion which is device or system dependent and a portion which is independent or invariant, e.g., a device driver where there is a common portion that applies to all devices and defines the common interface and a device specific portion for the particular hardware.

### **Composite**

The Composite pattern allows a group of objects to be treated in the same way as a single object by “composing” objects into a tree structure to represent a single entity. It is used when one wishes to treat individual objects and groups of objects in the same way, i.e., any operation you can perform on the individual you can also perform on the group and expect some kind of equivalent behavior. Thus, Composite provides for a tree structure in which both the nodes and the leaves share the same interface, or at least most of the same interface. If one relaxes the requirement for a common interface, Composite can be seen as a pattern which allows one to navigate hierarchical structures containing both composed and atomic data.

---

<sup>6</sup> Anti-Pattern: A design pattern which is widely used, but is ineffective or counterproductive in practice.

## Decorator

The Decorator pattern allows new or additional behavior to be added to an existing object dynamically. Decorator can be seen as an alternative to subclassing except that subclassing changes behavior at compile time and affects all instances of a class while Decorator occurs at run time and affects only a single instance. Thus, when there is a wide range of alternate behaviors which can occur in unpredictable or unstructured combinations, the subclass structure may be complex, unwieldy, and fragile, but Decorator allows composing any desired combination of behavior dynamically, notably during a particular object state rather than for the entire session. The classic example of Decorator is adding a scroll bar to a window when the content width exceeds the window width. That functionality is needed only in that state, so separating the functionality into a separate class simplifies the window class and encapsulates that functionality in its own class. Another good example is providing complex access control logic, which might be common to many classes, in a Decorator so that the base class need not implement that logic.

## Façade

In the Façade pattern, one object provides a simpler or separated interface to one or more other objects. It can be used to simplify a complex interface, to provide a level of separation between the client and the object or objects providing a service, or to make transparent the fact that multiple classes are serving the client. The intent of Façade is similar to Adapter, but Adapter exists specifically to connect two incompatible interfaces while Façade exists to convert a complex interface into a simple one for ease of use and design. One may have multiple Façades for the same collection of objects, each with a specific focus or purpose. Façade is also useful for providing a uniform interface when multiple different behaviors or objects might be involved, each with their own interface. For example, a data access layer might use uniform Façade object to provide a standard simple interface, but in one instance the Façade might connect to a local data source and in another it might connect to a remote one. One can think of Façade as similar to the process of defining an interface for a class by focusing on the invariants of the Responsibility corresponding to that class.

## Flyweight

A Flyweight is an object which minimizes memory usage by sharing as much data as possible with other similar objects. It is appropriate when one needs a large number of instances of an object, but instantiating an individual proper object for each instance would use an unacceptable amount of memory. The shared data is typically placed into external data structures and passed to the object when it needs to be used. The Flyweight pattern has three parts. First, one must divide the object into intrinsic and extrinsic parts. The extrinsic part is passed to the intrinsic part to provide a complete object. Second, one makes the extrinsic part a Value Object, i.e., it has no behavior and therefore uses little memory. Third, one needs a unique object Factory to combine the extrinsic and intrinsic parts to create a unique instance.

## Proxy

A Proxy, in its most general form, is a class functioning as an interface to something else. It is a local representative of some other resource which is expensive or impossible to duplicate. The Proxy is not transforming the interface, but merely providing a local instance. Proxies have several forms and purposes including a Protection Proxy to control access, Remote Proxy to hide details of a remote connection, and **Lazy Instantiation Pattern** in which the Proxy functions to take the place of a large or expensive object until or when it is needed.

## Behavioral Patterns

### Chain of Responsibility

The Chain of Responsibility pattern consists of a source of command objects and a series of processing objects. Each processing object contains logic for the types of command objects that it can handle, and how to pass off those that it cannot to the next processing object in the chain. New processing objects can also be added to the end of this chain. A series of processing objects is chained together and a command passed to the head of the chain. Each processing objects decides whether or not it can handle the command. If it can, it does so; if it can't, it passes it to the next processing objects in sequence. This simplifies the dispatch mechanism, since only one object for the head of the chain is required and is very flexible in allowing new sets of processing objects to be composed according to need and context. This pattern promotes loose coupling.

### Command

In the Command pattern, a command class represents and encapsulates all the information necessary to perform a certain operation, e.g., call a method, at a later time. A *client* object builds a Command object containing the necessary information; an *invoker* object decides when the Command will be executed; and a *receiver* object is the object containing the method to be executed. An example of the use of the Command pattern is multi-level undo in which each of the user's instructions is preserved as a Command object in a stack. If the user indicates a need for undo, the Command is popped off of the stack and its `undo()` method is run. Another example is a wizard where one often collects multiple pages of information into Command objects and then passes these to the invoker when the user has completed filling out the wizard.

### Interpreter

The purpose of the Interpreter pattern is to evaluate sentences in a formal language by associating a class with each symbol or token in the language and arranging these in a composite pattern tree structure. Interpreter does not cover the parsing of the input into tokens, only the representation of the parsed structure ready for evaluation. It is primarily used for evaluating specialized languages such as SQL or a special communications language.

### Iterator

The intent of the Iterator pattern is to provide an object which can be used to traverse some aggregate structure or collection, abstracting the details of the implementation of the structure. Typically, an Iterator provides methods for testing whether elements are available, advancing to the next or next Nth position, and accessing the value at the current position, but does so independent of the structure of the collection. By the use of an Iterator, the nature of the collection can be changed without changing the logic of the client using the collection. Also, Iterators allow multiple clients to access the same collection at the same time, each Iterator providing a pointer to the current element rather than that pointer being provided by the collection.

### Mediator

The purpose of the Mediator pattern is to provide a class which manages the communication among many objects in an application so that individual classes need not be linked directly, but can be linked through the Mediator. This reduces coupling among the classes in the application and provides a central point for modification of the communication structure. Thus, communication

paths can be varied independent of the classes which are communicating. Classes which use the Mediator for communication are sometimes called Colleagues.

### **Memento**

The purpose of the Memento pattern is to provide the ability to undo or rollback the state of an object to a prior state. Two objects are involved in this pattern, the *originator* and the *caretaker*. The originator is an object which has some state to which one would like to return. The caretaker is the object which has some action which it wishes to perform on the originator, but which wishes to have the ability to return to the prior state, e.g., if the action fails. The caretaker object requests a Memento object from the originator, containing the required state information. The caretaker never examines the Memento object, but merely holds it for the desired scope. If the action needs to be undone, the caretaker returns the Memento to the originator and the originator restores its state.

### **Observer**

In the Observer pattern a subject object maintains a list of Observer objects which have an interest in the state of the subject object. When specified aspects of that state change, the subject notifies the Observers of the change, typically via running a method on the Observer. This is a simple version of the more generalized notion of **Publish/Subscribe**. In Observer there is a direct connection between the subject and the Observer and the notification is synchronous, versus **Publish/Subscribe** in which the communication may be mediated by other objects and the notification may be asynchronous.

### **State**

The intent of the State pattern is to enable an object to act in different ways according to its internal state. Thus, in some respects the object appears from the outside to be multiple object types, but rather than changing from one object type to another, this change in behavior is connected to the internal state of a single object. This is useful when the object has a continuous rôle, but the behavior required at different times needs to vary. E.g., a cursor on a screen has a continuous existence, but its function at different times varies according to the way in which it is used. E.g., when the cursor is dragged across some text with the mouse button depressed, it switches from a pointing state to a marking state, enabling cut and paste and other behaviors which do not apply in the pointing state.

### **Strategy**

In the Strategy pattern, multiple algorithms are encapsulated, each in their own object with an interchangeable interface and can be substituted one for the other in use by an object which needs the category of algorithm without any specific coupling to any given specific algorithm. For example, one might have a series of algorithms for calculating travel time based on the mode of transportation – foot, car, public transit, etc. A routing object could then select the appropriate Strategy object from this set and have that object calculate the travel time for a particular journey based on user preference or other inputs. Strategy objects tend not to have state since they merely compute an algorithm based on inputs.

### **Template Method**

In the Template Method pattern, a class is defined which provides the abstract steps in an algorithm and subclasses of that class implement these steps to provide specific instances of the algorithm. This pattern is used when there is a general pattern applied in a number of specific ways so that the generalization structure reflects the unified aspects of the structure in the superclass and the differing

aspects are reflected in the subclasses. In some versions, the Superclass has only abstract steps, but in others all common parts of the algorithm are provided in the Superclass. Template Method differs from Strategy in that Strategy has no assumption that the alternate Strategies are similar in structure.

### **Visitor**

The purpose of the visitor pattern is to provide a mechanism for separating an algorithm from the object structure in which it operates, thus allowing the same algorithm to be applied to multiple object types and multiple algorithms to be applied to any one object on a dynamic basis. The essence of the implementation of this pattern is that an object provides a method which will accept objects of the visitor type and which will then execute a `visit()` method on that object to apply the algorithm to the object's structures with the object passed as an argument to the Visitor class. The Visitor typically contains code which applies different actions according to the class of the argument. This pattern improves Separation of Concerns by encapsulating a general class of behavior separate from the specific instances where that behavior might be used. Some feel that Visitor is overused since it implies a combinatorial interaction lattice between objects and algorithms, which does not often occur in most problem spaces. This leads to questionable practices such as “no-op” visits.

### **Summary**

This concludes the summary of the 23 GoF patterns. It is, at best, a brief summary and the reader is encouraged to further reading to gain a more complete understanding of these and other patterns, especially since many patterns have significant context about when they are and are not seen as appropriate to use and there is a diversity of opinion on their virtue and proper application. Moreover, the GoF patterns, which are the most famous of the OO pattern sets, are by no means the only patterns available and there are many books and many, many web pages with additional patterns. Other sets may supplement the GoF patterns with entirely new ideas, recharacterize GoF patterns in ways the author finds more intuitive or useful, or combine GoF and/or other patterns into more complex patterns.

It is important to recognize that patterns are not rules, but ideas about how to solve problems. They are not to be slavishly applied whenever the opportunity presents, but rather are intended to provide inspiration and a mnemonic aid for associating a familiar problem with a corresponding solution. If a pattern *almost* fits a problem, but not quite, then one should feel free to adapt the pattern to fit the requirements, possibly by borrowing elements from another pattern. Some problems will seem to fit multiple patterns and one must choose which provides the best solution for the purpose at hand. Their intent is to make design and programming easier, not to provide rules to limit solutions. Patterns can also be very helpful in communicating ideas.