Interceptors

A Design Pattern

The goal of this document is to provide an introduction to the Interceptor Design Pattern, its purpose, how it works, its primary uses and a brief comparison of it to other design patterns. Diagrams to aid in the description are also provided.

Robert Dabrowski
9/30/2013

Description

The interception pattern is literally a pattern for writing code that intercepts other code. More technically put, implementations of this pattern intercept specific method calls or events that take place at runtime in a system and replace whatever was about to happen with the registered interceptor version. From a developer's perspective, this pattern allows new implementations, services and behaviors to be introduced into a system at any time without the original system requiring modifications. This addition is supposed to be transparent and happen automatically when specified events occur. (Buschmann 109)

Purpose of Interceptors

The purpose of the interceptor pattern is to provide a means for adding additional services to a system in a transparent fashion. One might use an interceptor for a variety of things, from replacing algorithms to adding or changing behavior altogether. Ultimately the goal is to allow for modifications to existing code by addition rather than modification. Interceptors do just that; when an interceptor is loaded, it is literally inserted at an "interception point" that could be at any location in a given codebase. This is a step beyond simply creating a new class that follows an interface because of how powerful this insertion is. (Buschmann pg 110-120)

To allow for smooth integration, a few key rules must be followed by any implementation of the interceptor pattern: the core architecture must be initially designed to accept more interceptors without any modifications to the core, additions should not affect existing features or other unrelated components, and the ability to monitor and control behavior of a framework should be made available. Coincidentally, following these rules provides a better design in general because they promote change by addition and protection of existing code. The true purpose of interceptors is to provide a powerful means of changing by addition without modification. (Buschmann pg 110-120)

The most pertinent cause for using interceptors is the knowledge that more services, algorithms, specific implementations or internal features will be introduced to a system and also the fact that the exact number additions is completely unknown. It is also important to know where interception points should be located, although this requires significant foresight. However, if specific locations for changes are not known, the frameworks necessary for interceptors can still be

built and are well worth the risk of wasting time with initial design. The interceptor pattern has the power to allow additions to existing frameworks without any modification, and thus is used where the places of modification are sometimes known, but the volume of changes is not. (Daigneau 196) (Buschmann pg 110-120)

How They Work

In this section, I will explain different ways that the interceptor pattern works. In each paragraph, I will include more of the things that complicate and add power to interceptors, as there are different but similar versions of interceptors that may be implemented. Grossly oversimplifying the interceptor pattern, I have provided the illustration below. It shows that interceptors plug themselves in-between operations in an existing framework. The X signifies what the interceptor replaced and the partial disconnection of the interceptor signifies the transparency of the interception that allows it to work right into the existing system.

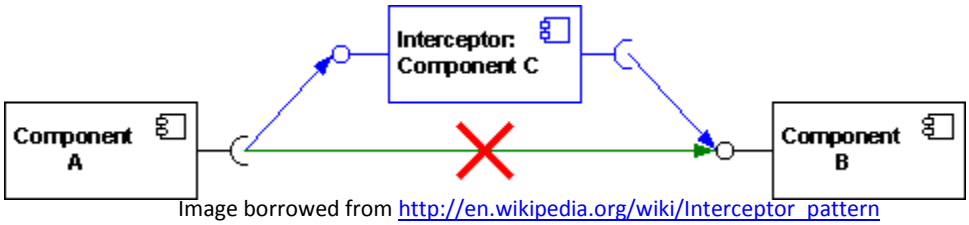


Image borrowed from http://en.wikipedia.org/wiki/Interceptor_pattern

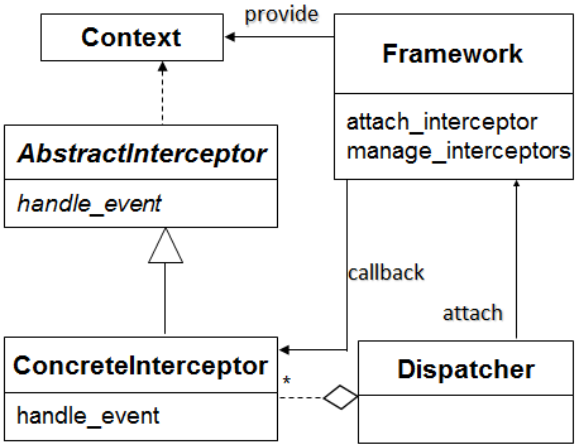
The interceptor pattern starts with a system, usually a client and server system. Next are interception points within that system; places where interceptors will someday do their work. These points are marked by special annotations, which Java recognizes, that mark both the interceptors and interception points. At runtime, these interception points trigger events that can be used to start an interceptor. The interceptor itself will take the place of some service or other interior feature of the system and return control when it finishes. In reality, it is more of a transparent substitution of the interceptor into the system that won't require any modifications to the original system. (Daigneau 197)(Buschmann pg 110-129)(Hansen 13)

To organize interceptors, a dispatcher class is often created to handle the registration and removal of interceptors. The dispatcher also handles picking a particular interceptor and calling on it when events are triggered. To help with organization, dispatchers are typically implemented as singleton objects to create a central interaction point for interceptors. A common feature to add to dispatchers is scheduling; dispatchers can be implemented to take multiple interceptors, register

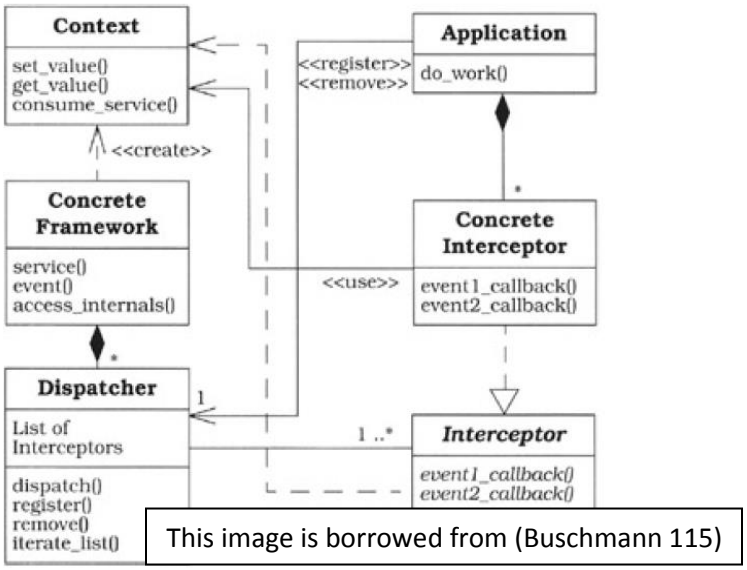
them in order and use them in that order when events happen. Sequences of interceptors or interceptor reuse can also be implemented in this way. (Buschmann pg 119-129)(Oracle)

Another construct used with interceptors is known as a context object. These objects are often passed into interceptors to provide information about the event that triggered the interception. These context objects can be generated by the dispatcher, or even earlier by the system itself. In addition to providing context about the event, references to objects in the main system may also be provided so that the interceptor may have the ability to do work on the system itself. (Buschmann pg 121-129)(Hansen 13)

The UML diagrams below show two implementations of the interceptor pattern. They each have concrete interceptors building off of either an interface or abstract class to make them uniform. It technically does not matter for the pattern, but it is a stronger design to make the interceptors to the interface. Also shared between these two versions is the fact that many interceptors may be used with a single dispatcher and all interceptors are capable of using context objects.



This image is borrowed from (Schmidt 103)



This image is borrowed from (Buschmann 115)

The differences between these two UMLS are that the implementation on the right splits up the application and the concrete framework, where the UML on the right only has a framework. This difference is also not very important to the interceptor pattern, so long as interceptors are registered with the dispatcher and are called properly when events trigger them. It should also be noted from these UML diagrams that different interactions between the classes may exist; for example, interceptors may know about dispatchers, but it isn't required. (Buschmann pg 115-130)(Schmidt 102-105)

Real-World Value

The true value of the interceptor pattern is in the fact that a system prepared for interceptors may take them at any time and in any amount by addition rather than modification. This means that long after release, a software package may be updated simply by adding a few files. The following example will demonstrate this value.

A good example of a typical use for the interceptor pattern is a client-server system. The reason for this is the fact that the system will need many updates, but it is unknown how many or exactly where. This example will be a simple example with the client being a user of a multiplayer game system, with the game hosted on the server. For this example, the game will be Strategy because it is familiar. During gameplay, the client will be sending messages to the server that represent moves that the player wants to make and the server will send back messages that say the result of the move or errors resulting from the move. It will be the client's job to take move requests from a real player and send that move to the server. It will be the server's job to validate and keep track of moves throughout the game for any players involved.

With these behaviors established and the knowledge that the server will have to be responsible for different things for different versions of the game, I have decided to use the interceptor pattern to aid with modifications to the server. The next step is to identify all the variability points in the server and also label these as interception points using the correct Java annotations. Also, the Strategy factory should be modified to be able to register interceptors into the system. A dispatcher that interceptors are registered with will also be necessary. Finally, context objects will also need to be developed mainly with the details about the move and possibly with references to the internal framework of the system. Those references could include access to the board configurations, piece details and rules for different versions of Strategy.

Later when more versions of Strategy are necessary, interceptors may be registered with the server and new game mechanics may be added. Additionally, the server will not require any modifications to accept these additions, which is the key to the success of interceptors. This means that new features and versions can be added with almost no development costs, which makes interceptors invaluable to this client-server system.

Similar Patterns

The primary example for picking interceptors over other patterns is the fact that once the initial work of setting up a framework to allow for an interceptor, any interceptor may be added and no modifications to the original code will be necessary. Interceptors can also be used to abstract services out of an implementation to allow for higher code reuse or modularity. They are supposed to not have any effects on the original design of the framework architecture. Finally, I might also choose interceptors because of how unique they are; anyone who may further develop system in the future will see interceptors and how specific they are, and thus have strong guidelines for consistent development. I must note that Buschmann considers interceptors to be "orthogonal to most other patterns presented in" his book, which is a downside because it is hard to jump into studying interceptors from any other design pattern, making it harder to implement (Buschmann 109).

Although the documentation I found did not clearly point to a counterpart for the interceptor, I would argue that the state pattern could be called a very basic interceptor pattern. My main reasoning for this is the potentially dynamic nature introduced by both patterns. The state pattern allows for different implementations of the same interface to be used as conditions vary during run time. The same is true for interceptors; interceptors may be registered and unregistered as the application/framework sees fit. Each of these patterns can allow varying behavior at run time. The problem with the state pattern is the main framework has to be updated to include the different states, or parameterized to accept any classes that follow the original interface. Interceptors, however, are much more powerful. As long as interception points are declared at the first creation of the framework, interceptors may be added at any time. They also may be designed to have little power over the system to full control of the framework through context objects. To further the difference between the two patterns, interceptors may actually be piggy-backed on each other to create "pipelines", where the state pattern does not inherently have this property (Daigneau 196). While it is true that the state pattern is capable of chaining concrete states together to form complex behavior, dynamic scheduling of states and can have full system control like interceptors, the primary difference of transparency still remains. Interceptors are transparent in that the rest of the system does not have to know an interception is taking place, where states must be passed in as parameters or hardcoded into the framework. Ultimately, interceptors provide a lot more power to a system than the state pattern.

Special Note

There actually are different java libraries for interceptors, and some do require modification of the original source code while others do not. However, the idea behind the pattern remains the same.

My Critique of Interceptors after Implementation

When reading about the implementation of context objects to use with interceptors, I thought they were really cool with how much power they handed out. I thought it was neat the way they could have anything from a simple string explaining the command to having references to the internal workings of a server. In my implementation, I just passed around objects as parameters instead of packaging them up into a context object, which does the same job. This was when I realized how dirty and dangerous interceptors are; in order for them to fully do their jobs, most will need references to critical pieces of the server and will be passed those references even if they don't need them (in most cases). With interceptors so easy to load into a program, things get very dangerous because interceptors from unknown and untrusted sources can get full control of the whole system BY DESIGN. This tension can be eased by restricting the number of references given in as context, but is still a concern.

Works Cited

Buschmann, F., Henney, K., & Schmidt, D. C. (2007). *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages.* Wiley.

Daigneau, R. (2012). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Upper Saddle River: Addison-Wesley.

Hansen, F. O. (n.d.). *Architecture and Design of Distributed Dependable Systems.* Retrieved September 30, 2013, from http://kurser.iha.dk/ee-ict-master/tiardi/Slides/ARDI6-POSA2-Interceptor.pdf

Oracle. (2013). *Overview of Interceptors*. Retrieved September 30, 2013, from EE 6 Tutorial: http://docs.oracle.com/javaee/6/tutorial/doc/gkigq.html

Schmidt, D. (n.d.). Pattern-Oriented Software Architectures . Nashville, Tennessee: Vanderbilt University.