GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

Event Notification and Method Encapsulation

Whether programming a standard desktop application with a windowing system, developing a web application using HTML and JavaScript or even building a complex distributed system of aggregated heterogeneous services, some form of event notification will invariably be required. As a software developer, you should already be familiar with ***event driven programming*** in GUI applications, where methods such as *onClick()* or *onMouseOver()* are implemented to provide notification to a running programme of user selections or gestures. Languages such as JavaScript and Visual Basic are based on ***event driven programming***. Even in complicated asynchronous systems, such as message queues, event notification services provide the glue that holds such an application together. All such systems are based on one or more implementations of the ***observer pattern***, which is also an essential component of the *Model-View-Controller* architecture.

Observer Pattern – Notify Me

Anytime there is a requirement for some form of event notification to be implemented, you should consider applying the observer pattern as a solution. The observer pattern enables the creation of a *one-to-many dependency*, so that when one object changes state, *all its dependents are notified and updated automatically*. The pattern is based on the publish-subscribe model where there is a 1:*n* relationship between a publisher (Subject) and a subscriber (Observer). The subject typically exposes methods such as ***register****(Observer o)*, ***unregister****(Observer o)* and ***update****(Object o)* that form a contract with client observers. Consider the example of a chat server application. The publisher (Subject) in this case is the chat server itself and the subscribers (Observers) are all the different client applications sending messages. When clients wishes to join a chat service, they first register with the service, thereby adding themselves as an observer to a list of other observers maintained by the chat server. When the chat server receives a message, it iterates over its list of observers (the 1:*n* relationship) and invokes a call-back method (an event) that notifies the client of a state change and then asks them to update. Note that this pattern is not just applicable to client-server interactions; it can be applied any time some model of event notification is needed.

There are ***two basic models of event notification*** that can be used:

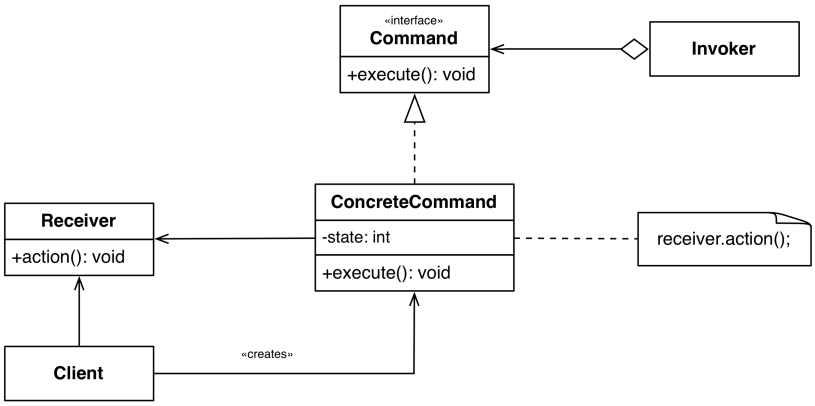
1. **Push:** When an update is received by the *subject* (chat server), it is responsible for pushing updates to all chat clients (*observers*). In a client-server environment, this requires that the *subject* knows the IP address and port number of each client listener.

2. **Pull:** It is the responsibility of each client (*observer*) to poll the chat server (*subject*) and ask if there have been any updates. This is a common approach in web and mobile application usage, where there is either no (or a limited only) mechanism for creating client-side sockets or network connectivity is sporadic and unreliable.

It is noteworthy that the an observer is typically a ***role*** that a class can play and, as such, the subject does not know anything about the list of observers it maintains, aside from their abstract interface methods, i.e. *observers are loosely coupled to the subject*. Therefore, any object that implements an observer interface can register to be notified by a subject. Because the pattern promotes the abstract coupling of observers to subjects, both can be reused independently of each other and changes to either subject or observer will not affect each other. It is also important to note that *observers should not depend on any specific order of notification*. The mechanism by which observers are maintained and notified is completely encapsulated inside the subject, which may use a list, set, map or any other 1:*n* data structure to manage its observers.

Encapsulating a Method Invocation

Sometimes we are presented with situations where a method is parameterised with an argument that relates to an enumeration of different potential actions. For example, when a chat client open a connection to a chat server, the set of actions may include *register()*, *unregister()* or *update()*. Alternatively, in a message queuing system that is capable to storing different types of requests, each request may warrant a different object to execute an action. A simple, but ungraceful, solution is to use an IF statement to check the type of request and determine who the request should be dispatched to. Using such an approach will require a change to the request receiver class each time a new action is added or an existing action removed. Moreover, it conflates and obfuscates the class receiving the request with a responsibility that is really not it own. A more robust solution will deal with the root issue, which is the variability or volatility of method request types. As we have already learned, when dealing with variability or volatility, encapsulation offers a mechanism for inducing stability into a design that will otherwise change often. A command pattern provides such stability by *encapsulating a request inside an object* (called a command object), enabling receiving methods to be parameterised with different requests. Command objects can also be used to queue requests and to implement undo operations in a stack.



The command pattern is composed of the following interacting classes:

- **Client**: responsible for creating a concrete command object and setting its receiver.
- **Receiver** knows how to perform the action needed to carry out a request. Any class can act as a receiver.
- **Invoker**: holds a command object and at some point asks the command to execute a request.
- **Command Interface**: Declares an interface for all commands. Declares a method like *execute()* which an invoker calls that asks a receiver to perform an action

- **Concrete Command**: defines a binding between an action and a receiver. An invoker makes a request by calling *execute()*. The concrete command carries out action by calling one or more methods on a receiver.

A client request is encapsulated in a command object and decouples an object making a request from an object that knows how to perform it. By encapsulating a receiver with an action or a set of actions. a command object is at the centre of this decoupling. Because the invocation is encapsulated, we can change the action without affecting the client in any way. The decoupling also confers upon a client the ability to make request without knowing the actual action that will be performed. An invoker makes the request of a command object by calling an *execute()* or similar method. This method invokes the action of the receiver, which implements the behaviour required by the client. In a client-server environment, the command pattern decouples the client from the server and provides the flexibility to change server-side classes. Moreover, as the command object encapsulates a method request in a single bulk operation, this has the further advantage of reducing number of network calls where a request encapsulates a multi-step operation. Note that the over-zealous application of the command pattern will lead to a proliferation of concrete command classes in an application.

Exercises

- Download the Zip archive titled "*Event Notification and Method Encapsulation - Chat Server Source Code*" on Moodle and extract the packaged files into a directory called ***chat-server***.

- Examine the interface *MessageObserver* and how its methods are implemented in the class *MessageObserverImpl*. *Explain how loose coupling is promoted between the observer and the subject. What are the implications of this design for the Open-Closed Principle (OCP)?*

- Examine the class *MessageBoard* and the instance variable *List<MessageObserver> observers* maintained by the class. Note how the methods *add()* and *update()* are implemented. *Explain how the subject supports 1:n dependencies.*

- Consider the *MessageCommand*, *MessageRequestImpl* and *MessageObserverImpl* classes. The first is an abstraction of a command and the latter two are concrete commands. Note how the *execute()* method of the concrete commands invokes the abstract methods of *Receiver*. The *MessageBoard* class acts as the *Receiver* (i.e. it knows how to execute the command). The class *Invoker* stores a command object and asks the instance of Command to execute a request.

- Consider how ***asynchronous communication*** between the client and server could be used to enhance the scalability of the design. *What changes should be made to accommodate this behaviour?*