ECE450 – Software Engineering II

Today: **Design Patterns VII**
Observer, Command, and Memento

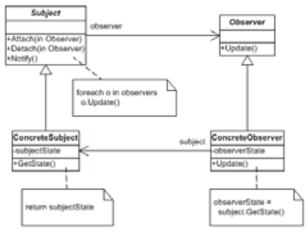
Keeping on top of things

- Many modern software applications need to maintain consistency with data stored or modified elsewhere
  - A stock trading application that responds to fluctuations in the stock market
  - A web-based email client that lets you know when you received new messages
  - An instant messenger client that keeps track of the status of your contacts in real time
  - A chart in a spreadsheet that changes whenever its data source changes
  - ...a long etcetera...
- How can we design an application that ensures this data consistency *without making the classes tightly coupled*?
  - For example, the data objects in the spreadsheet should not need to know that you have a chart
    - You could have a table, or a reference to the data in a text editor, or a different kind of chart, and the spreadsheet should not need to differentiate between all these types
- Ideas?

Publish-subscribe mechanism

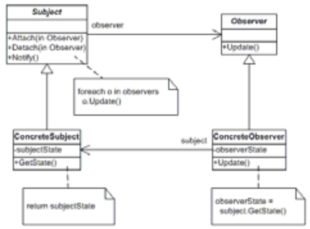
- The key to solve this problem is to identify that there are two elements: A **subject** and an **observer**
  - A subject may have any number of dependent observers
  - All observers will be notified whenever the subject undergoes a change in state
  - The subject, then, is the “publisher” of notifications...
  - ...and it sends these notifications to all of its “subscribers”, without worrying about what those subscribers are
- The *Observer* pattern
  - *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*

Structure and participants



- Subject
  - Knows its observers. Any number of Observer objects may observe a subject
  - Provides an interface for attaching and detaching Obsever objects
- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject

Structure and participants



- ConcreteSubject
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes
- ConcreteObserver
  - Maintains a reference to a ConcreteSubject object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer updating interface to keep its state consistent with the subject's

Applicability

- Use the Observer pattern in any of the following situations:
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
  - When a change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify other objects without making assumptions about who these objects are
    - In other words, you don't want these objects tightly coupled
- Using Java? You can use its Observer and Observable interfaces
  - ...but they'll consume an inheritance dimension
- Also applicable as user interface event "listeners"
  - For example, you want to know when the user moves the mouse or presses a key...
  - ...so you "listen" to events applying the Observer pattern

Consequences

- The Observer pattern lets you vary subjects and observers independently
  - It lets you add observers without modifying the subject or other observers
- Abstract coupling between Subject and Observer
  - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class
  - The subject doesn't know the concrete class of any observer
- Support for broadcast communication
  - Unlike an ordinary request, the notification that a subject sends need not specify its receiver
    - It is broadcast automatically to all subscribed objects
  - You can add and remove observers at any time; it is up to the observer to handle or ignore notifications

Implementation

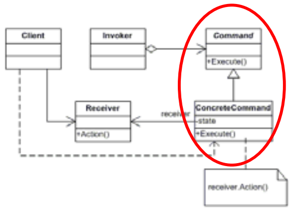
- Push- and pull-models of observing
  - So far we have described a "pull" model: Subjects notify observers that they have been modified, but do not specify what the modification was
    - Each observer is responsible for "pulling" the specific information they are interested in
  - The "push" model consists of having the subject send observers detailed information about the change, whether they want it or not
  - The pull model emphasizes the subject's ignorance of its observers, but may be inefficient
  - The push model is more efficient, but compromises reuse, as the Subject classes make assumptions about Observer classes that might not always be true

Moving on: Requests can be objects too!

- First problem: Sometimes it's necessary to issue requests to objects without knowing anything about the operations being requested or the receiver of the request
  - For instance, user interface toolkits that include objects like buttons and menus –the system does not know what each will do, but it knows that they'll do *something*
- Second problem: Implementing **undo**
  - Still the nightmare of many architects
  - Can make or break your software's usability
- Both problems can be solved with one pattern: ***Command***
  - *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations*

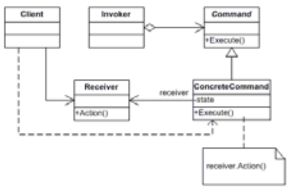
Structure and Participants

- Command
  - Declares an interface for executing an operation
- ConcreteCommand
  - Defines a binding between a Receiver object and an action
  - Implements Execute by invoking the corresponding operation(s) on Receiver



Structure and Participants

- Client
  - Creates a ConcreteCommand object and sets its receiver
- Invoker
  - Asks the command to carry out the request
- Receiver
  - Knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver



Applicability

- Use the Command pattern when you want to…
  - Parameterize objects by an action to perform
  - Specify, queue, and execute requests at different times
    - A Command object can have a lifetime independent of the original request
  - Support undo
    - The Command's Execute operation can store state for reversing its effects in the command itself
    - The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute
    - Executed commands are stored in a history list
    - Unlimited-level undo and redo is achieved by traversing the list backwards and forwards calling Unexecute and Execute, respectively
  - Support logging changes so that they can be reapplied in case of a system crash
    - Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation
  - Structure a system around high-level operations built on primitives operations
    - Common in information systems that support **transactions**
  - Note that you can construct **macros** of commands using the Composite pattern

Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it
- Commands are first-class objects. They can be manipulated and extended like any other object
- You can assemble commands into a composite command (again, through Composite)
- It's easy to add new Commands, because you don't have to change existing classes

Implementation

- Avoiding error accumulation in the undo process
  - Hysteresis can be a problem (that is, undo -> redo leading to slightly different state)
  - Errors can accumulate as commands are executed, unexecuted, and reexecuted repeatedly
  - It may be necessary to store more information in the command to ensure objects are restored to their original state
  - The Memento pattern (coming up!) can be applied to give the command access to this information without exposing the internals of other objects
- What happens if you try to undo an operation such as Print()?
  - You can't unprint!
  - Could do nothing
  - ...or raise an exception...
  - ...or provide an isUndoable method that acts like an impermeable barrier...
    - and optionally clear the history at each un-undoable call

And now for a discussion on *Memento*!

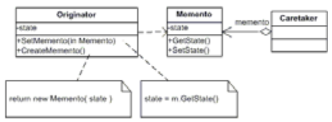
(You must see it if you haven't –one of the smartest and most creative movies ever!)

The **other** Memento...

- Sometimes it's necessary to record the internal state of an object
  - Implementing checkpoints
  - Undo mechanisms
- You must save state information somewhere so that you can restore objects...
  - ...but objects normally encapsulate some or all of their state, making it inaccessible to others!
  - Exposing this state would violate encapsulation
- Memento:
  - *Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later*
- A memento is an object that stores a snapshot of the internal state of another object –the memento's **originator**.
  - The undo mechanism will request a memento from the originator
  - The originator initializes the memento with information of its current state
  - Only the originator can store and retrieve information from the memento

Participants and structure

- Memento
  - Stores internal state of the Originator object
  - Protects against access by objects other than the originator
- Originator
  - Creates a memento containing a snapshot of its current internal state
  - Uses the memento to restore its internal state
- Caretaker
  - Is responsible for the memento's safekeeping
  - Never operates on or examines the contents of a memento



Applicability

- Use the Memento pattern when…
  - A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
  - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Consequences

- Preserving encapsulation boundaries
  - Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator
- Using mementos might be expensive
  - They will incur in considerable overhead if Originator must copy large amounts of information to store the memento or if clients create and return mementos to the originator often
- Defining narrow and wide interfaces
  - It may be difficult in some languages to ensure that only the originator can access the memento's state
    - In general, you want to make Memento an internal class to the originator, so that the Caretaker does not know how to read it