Observer & State

11/11/2009

Opening Discussion

- Do you have any questions about the reading for today?

Observer

- This behavioral pattern is used so that when the state of one object changes, numerous other objects can be notified of the change.
- The design for this includes an abstract Subject which can be observed, as well as multiple ConcreteSubjects and an Observer interface as well as ConcreteObservers.
- The Subjects allow Observers to be added and removed. The Observers simply have update methods.

Example

- GoF uses the example of several graphical objects that are linked to some data. For example, a spreadsheet, a bar graph, and a pie chart. When the data changes, all those observers should be updated.
- In Java, the GUI event listeners are all Observers. Just with slightly different terminology. The Listeners get called when something happens in the component that they are listening to.

Benefits and Drawbacks

- This pattern allows you to reuse Subjects and Observers independently.
- It provides an abstract coupling between the two which makes for a loose coupling between concrete implementations.
- Notifications are broadcast to all Observers. This allows flexibility with adding and removing the Observers.
- There are sometimes unexpected costs though. You have to be careful how often to alter a Subject if it has many listeners.

State

- This pattern allows an object to alter its behaviors when its internal state changes. In effect, it can appear to change class.
- For this pattern we have some type of Context class which is what the outside world sees. This class keeps a reference to a State object. The State interface has the methods that will vary depending on the state. We create a different subtype for each state. When the state of the context changes, we use a different State subtype.

Example

- GoF uses the example of a TCP connection. This type of object can have different states and depending on the state, it will behave differently when the use calls open or close.
- I have used this in my SwiftVis program when I have something like an enum where each option changes functionality. For examples, I have a filter that does coordinate conversions. The state indicates what type of conversion it does.

Benefits and Drawbacks

- This pattern localizes the behaviors for each state. This helps in two ways. First, it makes the code easier to edit and can make adding new states easier. Seconds, it prevents you from messing it up by having methods not properly check the state.
- This pattern makes state transitions very explicit and simple. All you do is make the state variable point to a new subtype of State.
- State objects can be shared (Flyweight).

Progress Reports

- Does anyone want to make a presentation of what they have done recently?