Design Patterns

A Brief Review

Several slides from Sommerville 9th ed Ch. 7

What are Patterns?

- Patterns are *reusable solutions* in a *context*
  - Context: when, where, tradeoffs, lesson-learned
  - There are all kinds of patterns:
    - Design patterns
    - Analysis patterns – tend to be domain-specific interpretations
    - Process patterns (Coplien) – reusable business process segments
- Patterns capture domain experiences from master practitioners that solve real problems
- Patterns form a common vocabulary within a team or organization

Pattern elements

Name
- Meaningful pattern identifier

Problem description.
- The problem (obviously) and the context of the problem

Solution description.
- Template for a design solution that can be instantiated different ways

Consequences
- The results & trade-offs of applying the pattern.

Name: **Observer**.

Problem description
- Used when multiple displays of state are needed.

Solution description
- Separates the display of object state from the object itself.

Consequences
- Optimisations to enhance display performance are impractical.

Pattern Types (from GOF)

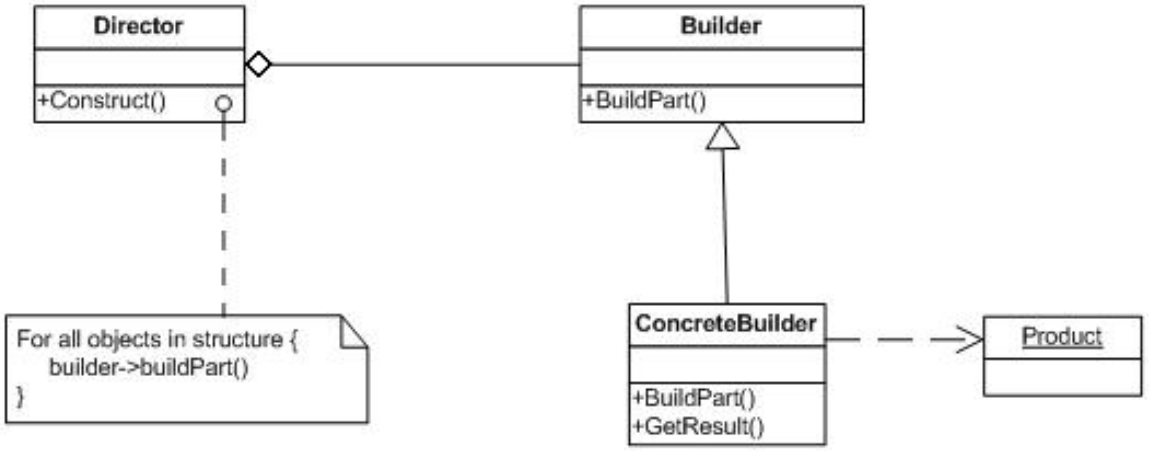
- Creational patterns
  - Decouple clients from knowing how an object is created
  - Examples: *Singleton* and *Abstract Factory*
- Behavioral patterns
  - Provides guidance on where to implement responsibilities
  - Examples: *Observer* and *Strategy*
- Structural patterns
  - Provides guidance on composing objects and classes
  - Example: *Decorator* and *Adapter*

A pattern is a (recurring) solution to a problem in a given context

Design problems

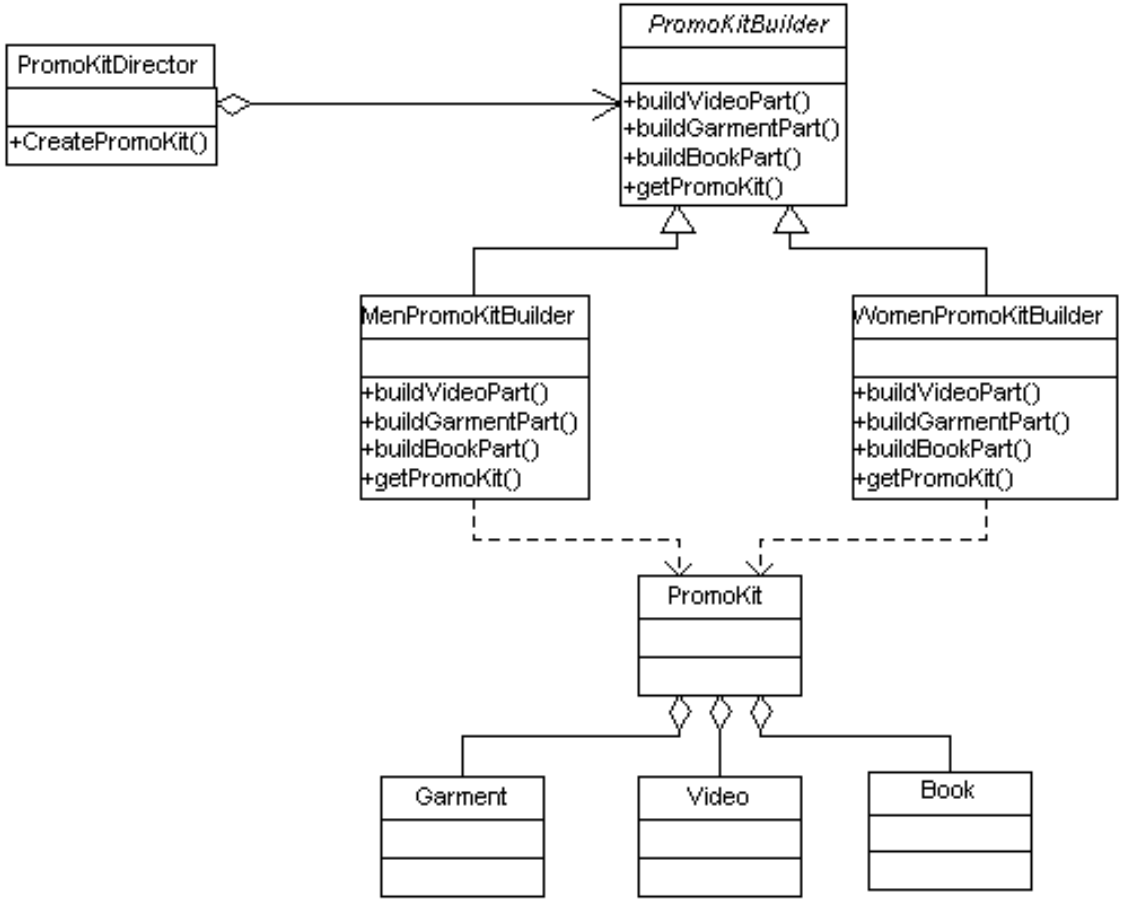
- Patterns give us a higher-level vocabulary by which to describe and discuss low-level solutions.
  - Instead of “write a class to manage the lifecycle of a certain type of objects”, say “create a *Factory* for them”
  - Other examples:
    - Tell several objects that the state of some other object has changed (Observer pattern).
    - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
    - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
    - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

CreationalPattern: Builder



- Goal of pattern is to produce a complex object
  - Often this "object" is a Façade or Composite
- Object model is not built "in one shot"
  - The creation process is itself complex, has many conditions, and may partially succeed or fail
- You often refactor a Builder from a Factory Method or Abstract Factory
- See also: Prototype

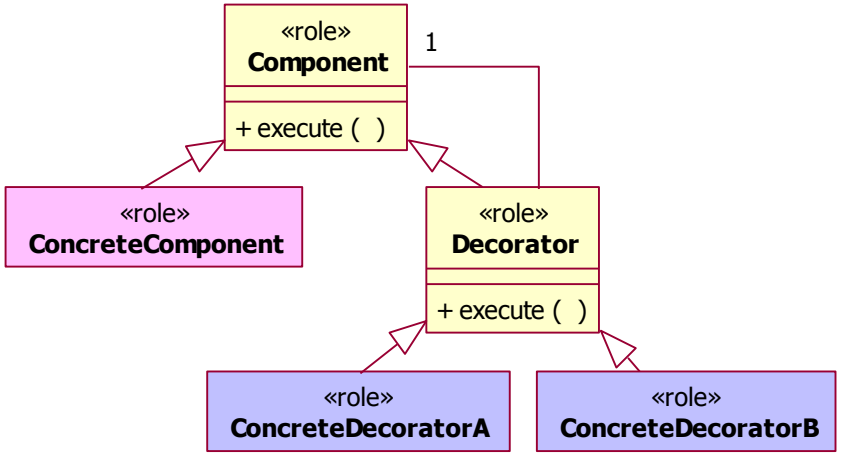
Builder Example



- From http://www.apwebco.com/gofpatterns/creational/Builder.html

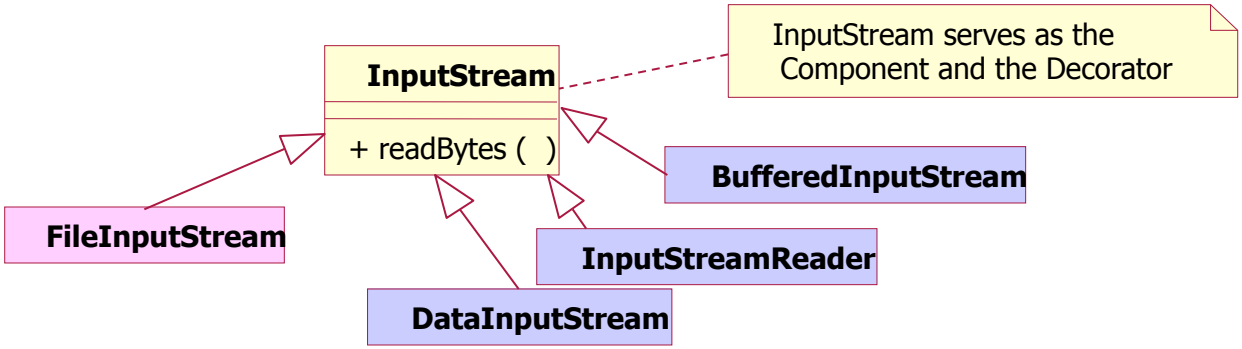
Structural Patterns: Decorator

- Intent: Attach additional responsibilities to an object dynamically.
  - Can easily add new decorations to existing system
  - Decorators provide an alternative to subclassing for extending functionality

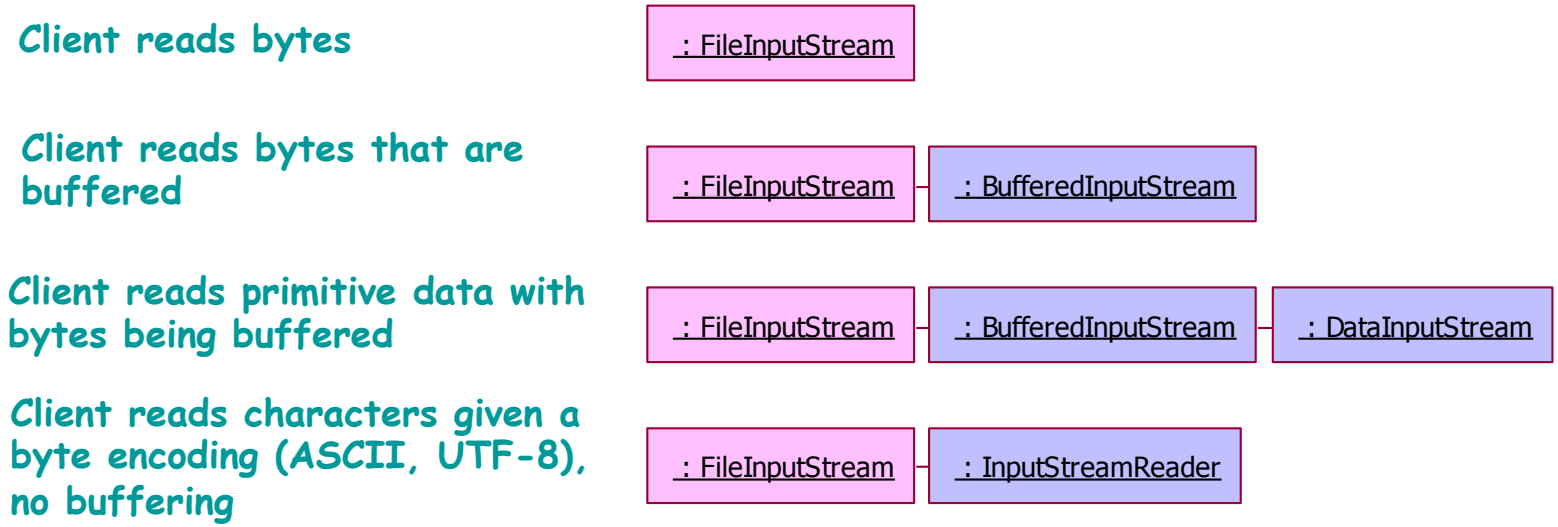


Decorator Example (Java Streams)

- Java provides rich set of stream classes for reading bytes and interpreting them
  - *Input stream* interface for reads bytes
  - *Buffered stream* read bytes from another stream & buffers
  - *Data stream* reads bytes & converts to primitive types
  - *InputStreamReader* reads bytes and converts to char



Decorator Example (Java Streams)



- Other Java technologies define extensions of InputStream so their streams can play in the decoration process

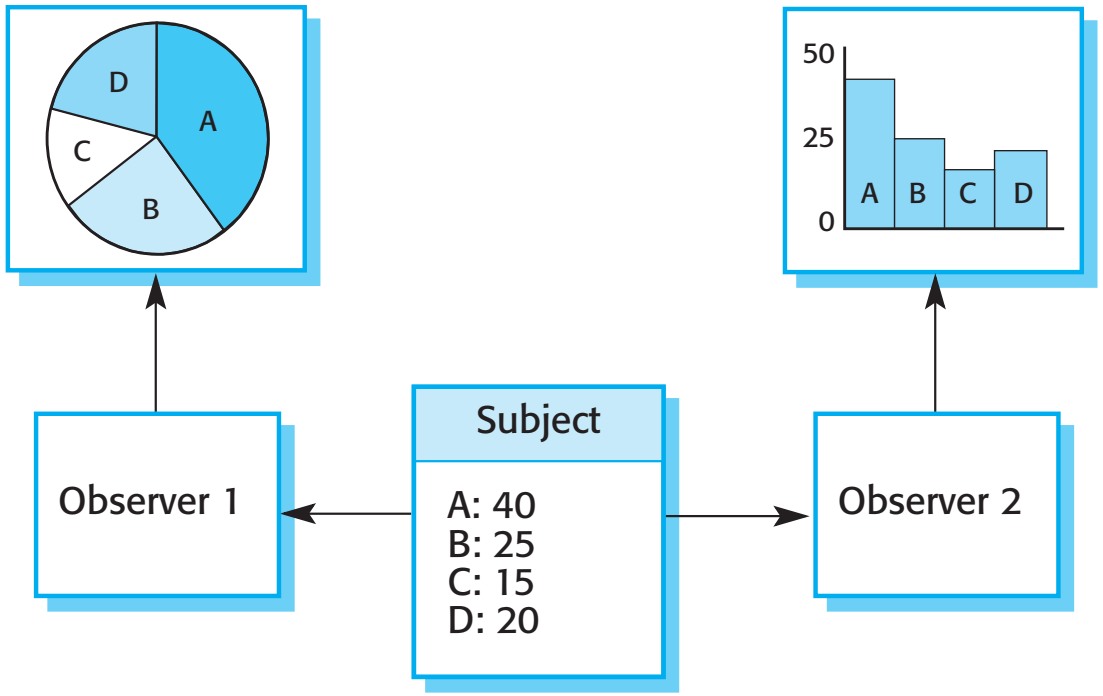
Deep-dive: Behavioral: The Observer pattern (1)

Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

The Observer pattern (2)

Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

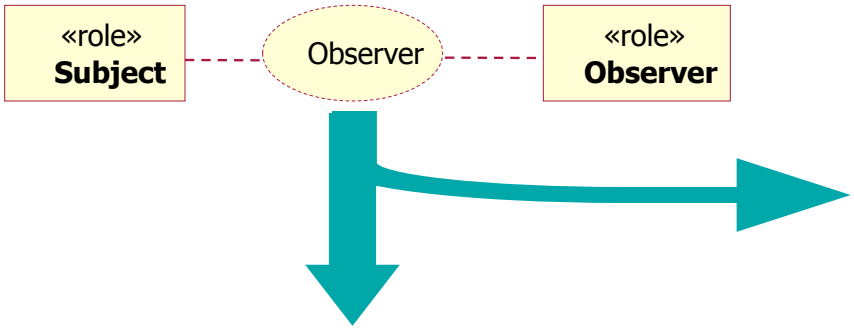
Multiple displays using the Observer pattern



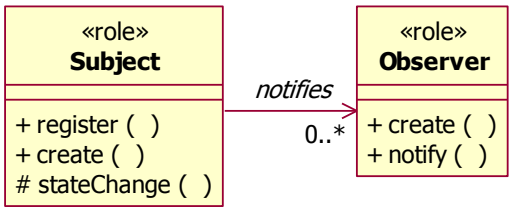
Behavioral: Observer Pattern

- *Defines a one-to-many dependency between a subject and observers. When the subject object changes state, all its observer objects are notified*

UML Collaboration

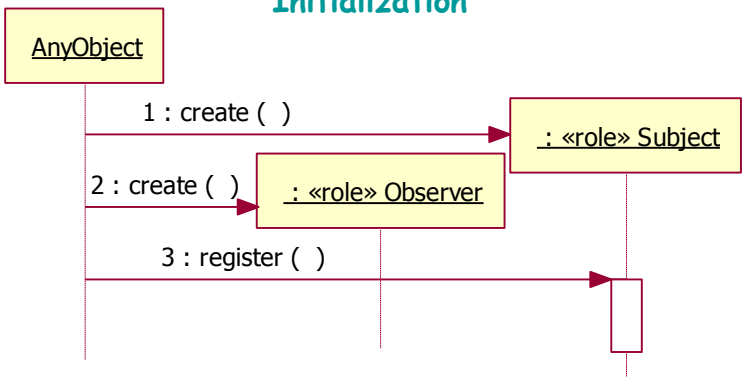


Structural View

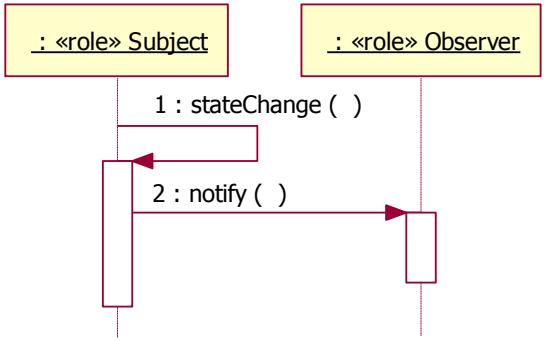


Behavioral Views

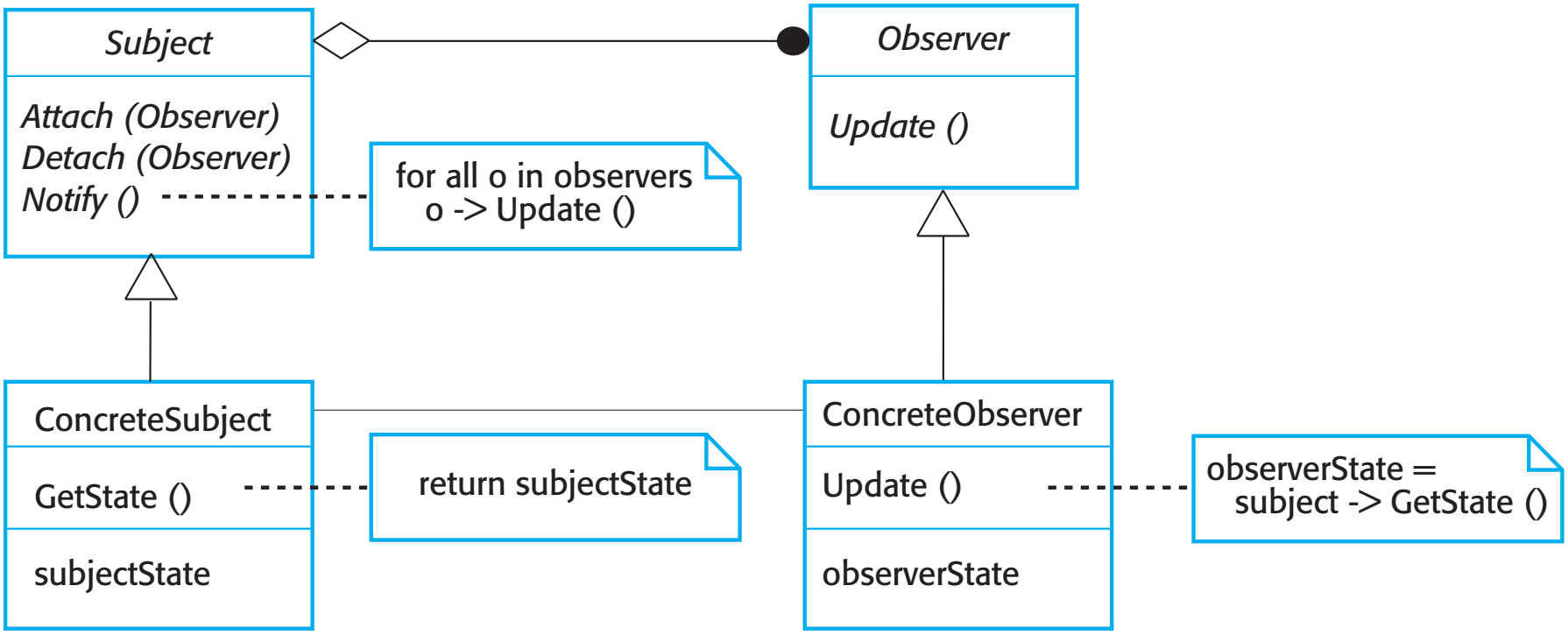
Initialization



Notifying observers

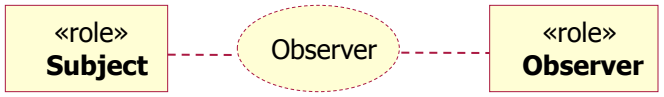


A UML model of the Observer pattern

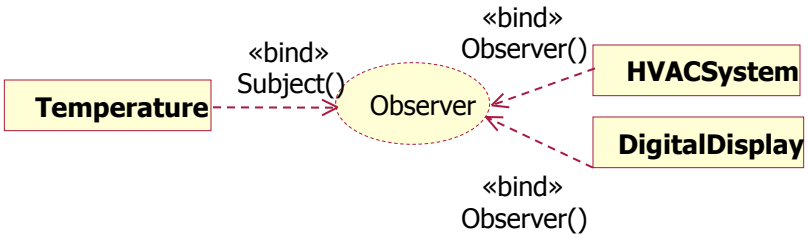


Applying the Observer Pattern

Collaboration Template



Instantiation




```
interface Observer {
    public abstract void notify();
}
```

```
class DigitalDisplay realizes Observer {
    public void notify() {
        //update digital display
    }
}
```

```
class Temperature {
    List<Observer> myObservers;

    public register (Observer newOne) {
        myObservers.add(newOne);
    }

    // Internally detect state change
    protected detectTemperatureChange() {
        for each observer in myObserver
            observer.notify();
    }
}
```

Participating in a pattern adds responsibilities to an object's implementation

Java Support for Observers

Java provides two types of Observer mechanisms:

- Listeners
  - Commonly associated with GUI applications
  - "Lightweight" in that the observed object is responsible for maintaining the list of listeners and notifying them.
- Observable/Observer
  - A class that allows itself to be watched must extend Observable
  - A class that watches an Observable implements Observer
  - Thoughts:
    - Basically provides an implementation of the Listener approach for you by providing methods on Observable like *addObserver*, *notifyObservers*, etc.
    - Since your class must extend Observable, it becomes tightly coupled to that inheritance hierarchy.

Design Patterns Wrap-up

Design pattern for *new implementations*

- Provide a means to rapidly assemble code
- DPs tend to favor delegation over inheritance

Design Patterns for already *existing code*

- Patterns define a "target" for refactoring!
- Several structural (Adapter, Composite, Façade) patterns and some Behavioral (Visitor, Mediator) help w/ integration

Patterns raise the vocabulary level of teams

- Less effort explaining parts of systems
- Less effort understanding code when we understand system's common patterns

References and Interesting Reading

- “A Pattern Language”, Christopher Alexander, 1977.
- “Design Patterns: Elements of Reusable Object-Oriented Software” Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- “Pattern-Oriented Software Architectures: A System of Patterns”, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal.
- “Analysis Patterns : Reusable Object Models”, Martin Fowler
- Mary Shaw and David Garlan, “Software Architecture: Perspective on an Emerging Discipline”