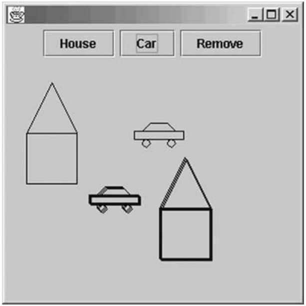
Introduction to Design Patterns

Three examples

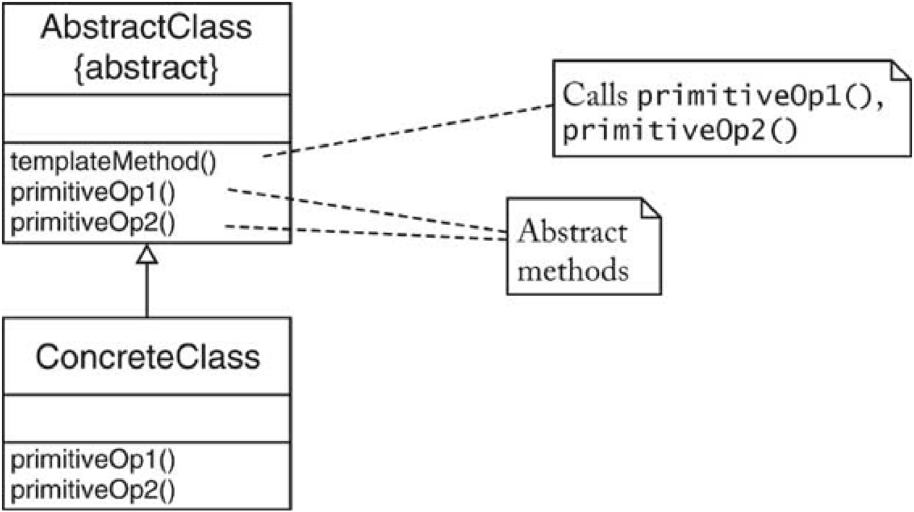
Design Patterns can be simple

- Highlighting a `Shape` in a GUI application
- Possible solution: Each class, such as `Car`, `House` implements a method called `highlight`
- **Problem**: Inconsistent
- **Solution**: In class `Shape`:

```
public void highlight() {
  translate(1,1);
  draw();
  translate(1,1);
  draw();
  translate(-2,-2);
}
```



Template Method



Template Method Context

- An algorithm is applicable for multiple types
- The algorithm can be broken down into primitive operations that may be different for each type
- The order of the primitive operations does not depend on the type

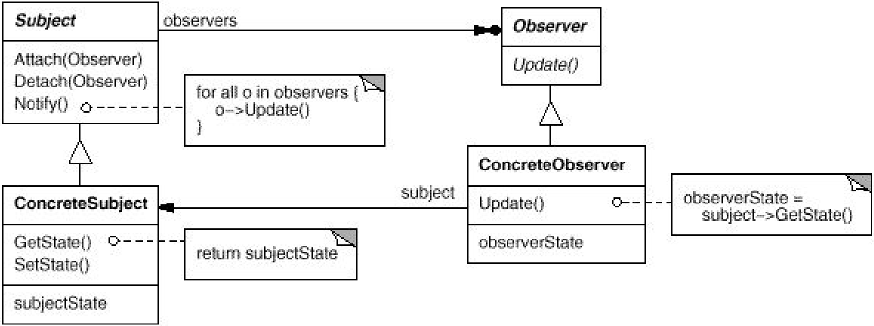
Template Method Solution

- Define an abstract superclass with a method for the algorithm and abstract methods for the primitive operations
- Algorithm calls primitive operations in right order
- Each subclass implements primitive operations but not the algorithm

Observer Pattern

- **Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Motivation** : Maintain consistency between related objects while avoiding tight coupling between their classes

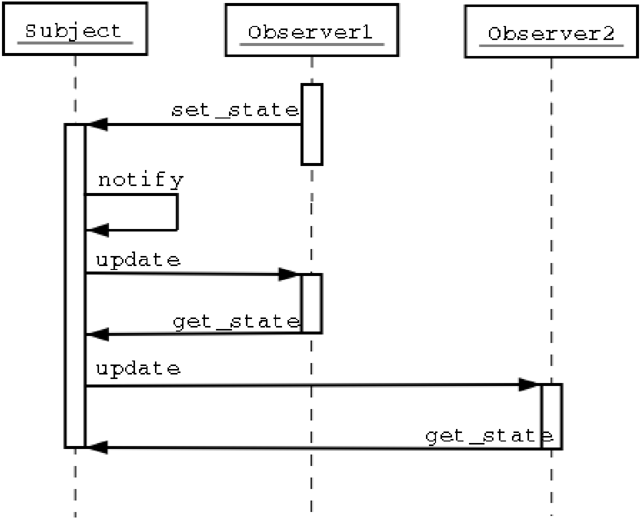
Observer Class Diagram



Observer - Participants

- Subject
  - Knows its observers
  - Provides interface for attaching, detaching and notifying its observers
- Observer
  - Defines an updating interface for observers
- Concrete subject
  - Stores state of interest to concrete observers
  - Notifies observers when state changes
- Concrete observer
  - Maintains a reference to its concrete subject
  - Stores state that corresponds to the state of the subject
  - Implements Observer updating interface

Observer Sequence Diagram



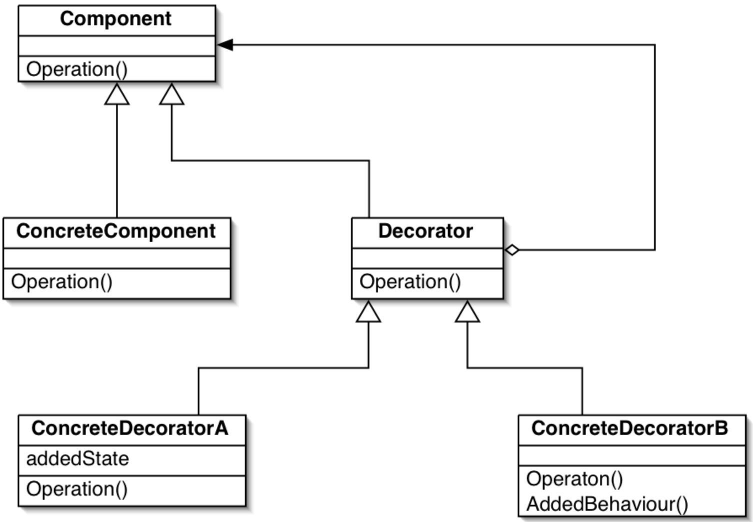
Observer - Consequences

- Abstract coupling between subject and observer
  - Permits changing number of observers dynamically
- Supports broadcast communication
- Can have observers depend upon more than one subject
- Need additional protocol to indicate what changed
  - Not all observers participate in all changes
- Dangling references when subject is deleted
  - Notify observers when subject is deleted

Decorator Pattern

- Intent: Attach additional responsibilities to an object dynamically
  - Provide a flexible alternative to subclassing for extending functionality
- Motivation: Want to add responsibility to individual objects not to entire classes
  - Add properties like border, scrolling, etc to any user interface component as needed

Decorator Class Diagram



Decorator Participants

- Component: defines the interface for objects that can have responsibilities added to them dynamically
- Concrete component: Defines an object to which additional responsibilities can be attached
- Decorator: Maintains a reference to a component object and defines an interface that conforms to Component
- Concrete decorator: Adds responsibilities to the component

Decorator Object Diagram

Decorator - Applicability

- Add responsibilities to individual objects dynamically and transparently
  - Without affecting other objects
- For responsibilities that can be withdrawn
- When subclass extension is impractical
  - Avoid combinatorial explosion of possible extensions
  - Class definition may be hidden or otherwise unavailable for subclassing

Decorator - Benefits

- More flexibility than static inheritance
  - Can add and remove responsibilities dynamically
  - Can handle combinatorial explosion of possibilities
- Avoids feature laden classes high up in the hierarchy
  - Pay as you go when adding responsibilities
  - Can support unforeseen features
  - Decorators are independent of the classes they decorate
  - Functionality is composed in simple pieces

Decorator - Liabilities

- From object identity point of view, a decorated component is not identical
  - Decorator acts as a transparent enclosure
  - Cannot rely on object identity when using decorators
- Lots of little objects
  - Often result in systems composed of many look alike objects
  - Differ in the way they are interconnected, not in class or value of variables
  - Can be difficult to learn and debug