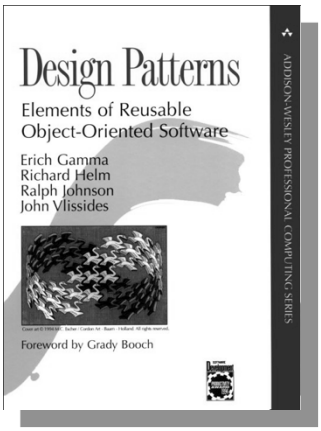
Design Patterns

Outline

- What is a Design Pattern?
- Why study Design Patterns?
- Describing Design Patterns
- Catalog of Design Patterns
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Design Pattern Examples
  - Singleton – Creational
  - Façade – Structural
  - Observer – Behavioral

What is a Design Pattern?

- A design pattern "…names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design." [GoF 95]
- A design pattern is a proven solution to a recurrent problem in a context.
- An effective, reusable, proven structure/communication solution for a given object-oriented design problem.



Why study Design Patterns?

- Reuse existing, high-quality solutions to commonly recurring problems.
- Establish common terminology to improve communications within teams.
  - Shifts the level of thinking to a higher perspective.
- Improve team communications and individual learning.
- Improved modifiability and maintainability of code
  - Design patterns are time-tested solutions
- Adoption of improved object-oriented design strategies
  - Encapsulation and information hiding
  - Design to interfaces
  - Favor composition over inheritance

Describing Design Patterns

- Pattern name and classification
- Intent
  - Design issue/problem being addressed
- Structure
  - A graphical representation (in UML) of the classes in the pattern
- Participants
  - Classes participating in the pattern and their responsibilities
- Collaborations
  - How participants collaborate to carry out their responsibilities
- Consequences
  - Trade-offs and results of using the pattern
- Implementation
  - Techniques, hints, or pitfalls to be aware of when implementing
  - Sample code

Catalog of Design Patterns

- The Gang of Four (Gamma, Helm, Johnson, and Vlissides) did the early work on design patterns.
- GoF described a structure within which to catalog and describe design patterns.
- GoF cataloged 23 patterns.
- It is important to note that GOF did not create these patterns.
- They identified patterns that already existed in high-quality designs.

Catalog of Design Patterns

Creational	Structural	Behavioral
Factory Method	Adapter	Chain of Responsibility
Abstract Factory	Bridge	Command
Builder	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	**Facade**	Mediator
	Flyweight	Memento
	Proxy	**Observer**
		State
		Strategy
		Template Method
		Visitor

Catalog of Design Patterns

- Creational Patterns
  - Abstract the instantiation process
  - Define classes to handle object creation
- Structural Patterns
  - Define how classes and objects are composed to form larger structures.
  - Describe ways to compose objects to realize new functionality.
- Behavioral Patterns
  - Concerned with algorithms, flow of control, and assignment of responsibilities between objects
  - Describe how a group of objects cooperate to perform a task.

Singleton Pattern

- Pattern Category – Creational
- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.
- Problem addressed
  - Ensuring that a class is instantiated only once, and that the resulting object is readily accessible.
- Solution
  - Make the class itself responsible for instantiation and knowing whether it has been instantiated.

Singleton Pattern

- Implementation
  - Add a private static member of the class that refers to the desired object (initially, it is null).
  - Add a public static method that instantiates this class if this member is null (and sets this member's value) and then returns the value of this member.
  - Set the constructor's status to private or protected so that no one can directly instantiate this class and bypass the static constructor mechanism.
- Consequences
  - Controlled access to sole instance.
  - Easily adapted to permit a fixed number of instances greater than one.
  - Easily extended to provide a family of such access-controlled subclasses with different functionality.

Singleton Pattern

- Example in Java

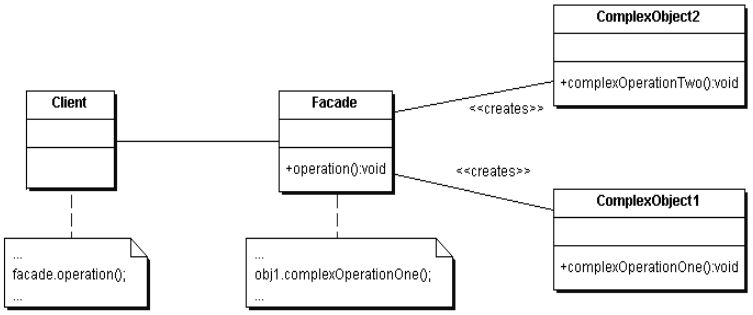
```
public class Customer {
    private static Customer instance;

    private Customer() {}

     public static Customer getInstance() {
       if (instance == null)
          instance = new Customer();
       return instance;
     }
}
```


Facade Pattern

- Pattern Category – Structural
- Intent
  - Provide a unified interface to a set of interfaces in a subsystem.
  - Facade defines a unified higher-level interface that makes the subsystems easier to use.
- Problem addressed
  - Using design patterns often leads to a complex system of many small components which may be daunting for the casual user. It would be nice if there were a way to provide a simple interface for the basic functionality that is needed most often.
- Solution
  - Create a Facade class that encapsulates the basic functionality of the system by bundling together common operations.

Facade Pattern

- Graphical representation



Facade Pattern

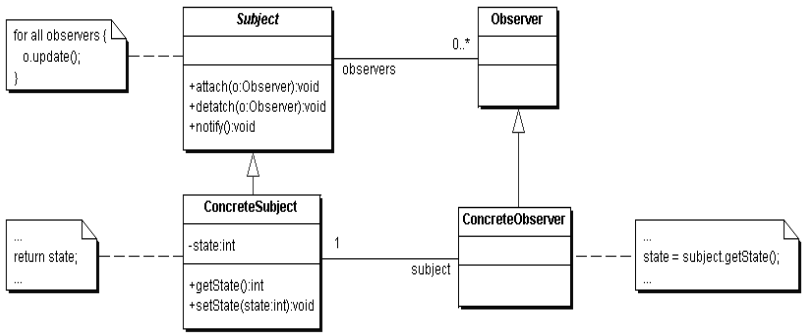
- Consequences
  - Simplifies use of system
    - Shields users from subsystem components, limiting the number of objects that the user must deal with
  - Does not limit "power users"
    - Serves as a convenience layer for general users, without limiting access to subsystem components for those that desire or need more customizable uses
- Facade pattern can be applied when
  - You do not need to use all the functionality of a complex system
  - You want to encapsulate or hide the original system
  - You want to extend the functionality of the original system
  - The cost of writing the new class (facade) is less than the cost of everybody learning how to use the original system

Observer Pattern

- Pattern Category – Behavioral
- Intent
  - Define a one-to-many dependency between objects so that when one object (*subject*) changes state, all its dependents (*observers*) are notified and updated automatically.
- Problem
  - You need to notify a ***varying*** list of objects that an event has occurred.
- Solution
  - Observers delegate the responsibility for monitoring for an event to a central object – the subject
- Use the Observer pattern when the
  - List of objects that need to be notified of an event changes or is somehow conditional.
  - Subject cannot anticipate every object that might need to know about the event.

Observer Pattern

- Graphical representation



Observer Pattern

- Implementation
  - Step 1
    - Make the observers behave in the same way
    - To do this: all observers implement the same interface
      - update() method
  - Step 2
    - Observers are responsible for knowing what they are to watch for
    - To do this: Have the observers register themselves with the subject
    - Subject provides two methods:
      - attach(observer)
      - detach(observer)
  - Step 3
    - The subject notifies the observers when the event occurs
      - notify() method
  - Step 4
    - Observers get the information from the subject
- The Observer pattern aids flexibility and keeps things decoupled.

Observer Pattern

- Support for observer pattern in Java
  - java.util.Observable class
  - java.util.Observer interface

```
public class Observable {
   public void addObserver(Observer o);
   public void deleteObserver(Observer o);
   public void notifyObservers();
   public void notifyObservers(Object arg);
   public void deleteObservers();
   protected void setChanged();
   protected void clearChanged();
   public boolean hasChanged();
   public int countObservers()
}
```

Observer Pattern

- java.util.Observer interface

```
public interface Observer {
   void update(Observable o, Object arg);
}
```