

## A Brief Overview of Design Patterns

Xiannong Meng  
for  
CSCI 479  
Fall 2012

### What is a *design pattern*?

- A *design pattern* is a description of communicating objects and classes that are customized to solve a general design problem in a particular context.

### A simplistic example

- Problem: Read a sequence of employee data from a file until a particular employee ID is read.

```
f = open( 'input.data', 'rb' )
data = f.read( bytesOfData )
while data.id != IDToStop :
    store data in structure (e.g., table, tree)
    data = f.read( bytesOfData )
```

A "pattern" emerges:

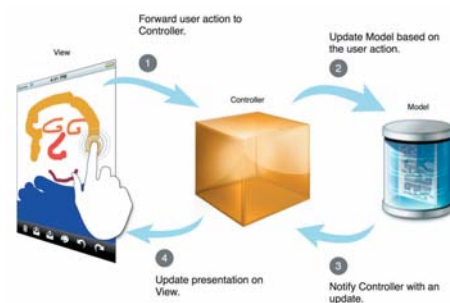
```
prime the condition
while the condition is true:
    do some work
    update the condition
```

### Two important facts w.r.t. DP

- Two very important pieces of information play critical role in the concepts of design patterns
  - MVC: Model-View-Control design pattern segregates the objects into three roles, *model*, *view*, and *control*. The pattern dates back to the days of *Smalltalk*, late 70's and early 80's. Still critically important today.
  - The publication of *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994), a.k.a. Gang of Four (GOF).

### Model in MVC

- Encapsulate data and basic behaviors in *model*
- Model objects maintain
  - Application data
  - The logics that manipulate these data
- View objects
  - Respond to user actions
  - Present to the user the data from the model
- Control objects act in between model and view



The diagram is from the book by Chung (2011)



## Issues affecting design

- *Programming To An Interface, Not An Implementation*
  - Design should concentrate on interface (behavior), not specific implementation
- *@protocol vs. Abstract Base Class (ABC) (in the terms of Objective-C), or interface vs. abstract class (in the terms of Java)*
  - Contrast between @protocol (Java interface) and ABC (Java ABC)
- *Object Composition vs. Class Inheritance*
  - Composition (black-box, properties hidden), inheritance (white-box, properties visible)

## Example of *behavior*

- We want to add books to a container that allows search, the details of the container is not the concern of the user (applications)

```
import bst as BookCase
#import avltree as BookCase
#import linkedlist as BookCase
myShelf = BookCase()
abook = read from the user
while abook != None:
    myShelf.add( abook )
    abook = read from the user
```

## Example of interface vs. ABC

```
public interface Walkable {
    void walk();
}

public abstract class Pet implements Walkable {
    protected String name;
    public Pet(String name) {this.name = name; }
    public void walk() {System.out.println("Pet's common walk!"); }
}

public class PetOwner implements Walkable {
    private String name;
    public PetOwner(String name) {this.name = name; }
    public void walk() {
        System.out.println(this.name + " : Exercise is good for me.");
    }
}

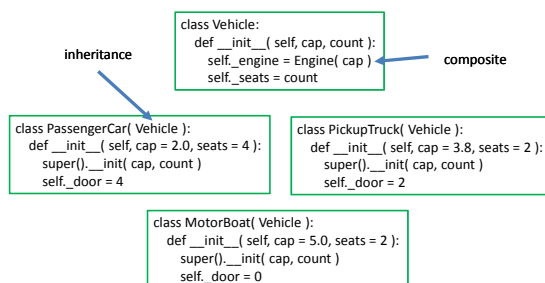
public class Dog extends Pet implements Walkable {
    public Dog(String name) { super(name); }
    @Override public void walk() {
        super.walk(); // Activate the parent's method before its own
        System.out.println("Dog " + this.name + " : Pant!!! Where's my leash!?!");
    }
}
```

## Applications that might use both

```
public class TestWalking {
    public static void sendWalking(Walkable walkObj) {
        walkObj.walk();
    }

    public static void main(String[] args) {
        PetOwner bob = new PetOwner("Tarzan"),
        jane = new PetOwner("Jane");
        Pet[] ourPets = {new Dog("Spot"), new Cat("Fluffy"), new Dog("Ceaser"),
        new Dog("Roadkill") };
        sendWalking(bob);
        sendWalking(jane);
        for (Pet p : ourPets)
            sendWalking(p);
    }
}
```

## Example of composite and inheritance



## Common software design patterns

- In three general groups (23 original GOF patterns)
  - *Creational patterns*: object creation mechanisms, trying to create objects in a manner suitable to the situation.
  - *Structural patterns*: design by identifying a simple way to realize relationships between entities.
  - *Behavioral patterns*: common communication patterns between objects and realize these patterns.

## Creational patterns

- **Abstract factory:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder:** Separate the construction of a complex object from its representation allowing the same construction process to create various representations.
- **Factory method:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Prototype:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Singleton:** Ensure a class has only one instance, and provide a global point of access to it.

## Structural patterns (1)

- **Adapter** (Wrapper or Translator): Convert the interface of a class into another interface clients expect.
- **Bridge:** Decouple an abstraction from its implementation allowing the two to vary independently.
- **Composite:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator:** Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.

## Structural patterns (2)

- **Facade:** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight:** Use sharing to support large numbers of similar objects efficiently.
- **Proxy:** Provide a surrogate or placeholder for another object to control access to it.

## Behavioral patterns (1)

- **Chain of responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

## Behavioral patterns (2)

- **Mediator:** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento:** Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
- **Observer** (or Publish/subscribe): Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
- **State:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

## Behavioral patterns (3)

- **Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template method:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

### An *iterator* example

- Iterator is almost a universal structure needed by all objects
- For example, your management asked your team to write a collection of programs to examine the performance of various search algorithms, using linear search, binary search, binary search trees, AVL trees, and hash table.
- Each of these structures needs an iterator.
- See the code example in `code/iterator`

### References

- Chung, C., (2011), *Pro Objective-C Design Patterns for iOS*, Apress, New York, NY.
- Freeman, E. & Freeman E., (2004), *Head First Design Patterns*, O'Rely, Sebastopol, CA.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J., (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.
- Wikipedia: *Software Design Pattern*, [http://en.wikipedia.org/wiki/Software\\_design\\_pattern](http://en.wikipedia.org/wiki/Software_design_pattern)