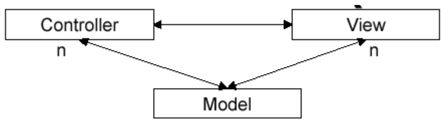
# Design Patterns

## Design Patterns

- A design pattern is a template solution that developers have refined over time to solve a range of recurring problems
- Generally object-oriented focus
  - Name that uniquely identifies the pattern
  - Problem description that describes situations it can be used
  - Solution stated as a set of classes and interfaces
  - Consequences that describes tradeoffs and alternatives

## Model-View-Controller (MVC)

- Archetypical example of a design pattern
- Three components
  - Model : Encapsulates system data and operations on the data
  - View : Displays data obtained from the model to the user
  - Controller : Handles events that affect the model or view



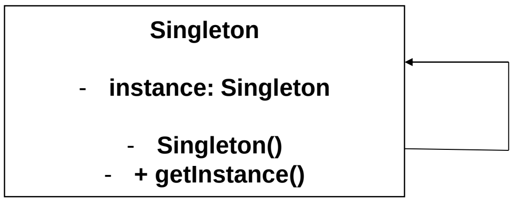
- Separating user interface from computational elements considered a good design practice

## Exercise

- Consider a program that displays an analog clock;  what could correspond to the model, view, and controller?

## Singleton

- Sometimes it is important to have only one instance of a class, e.g., a window manager or ID generation system
- The singleton class ensures that only one instance of a class is created



## Singleton

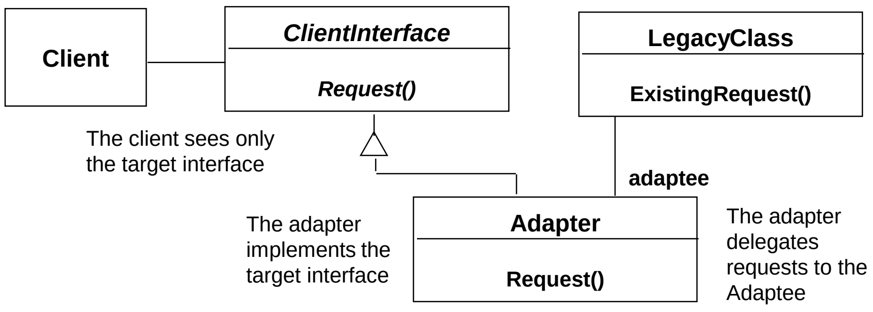
```
class Singleton
{
        private static Singleton instance;
        private Singleton()
        {
        ...
        }

        public static synchronized Singleton getInstance()
        {
                if (instance == null)
                        instance = new Singleton();

                return instance;
        }
        ...
        public void doSomething()
        {
                ...
        }
}
```

Usage: Singleton.getInstance().doSomething()

## Adapter Pattern

- "Convert the interface of a class into another interface clients expect."
- The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent. We will only cover object adapters (and call them therefore simply adapters)

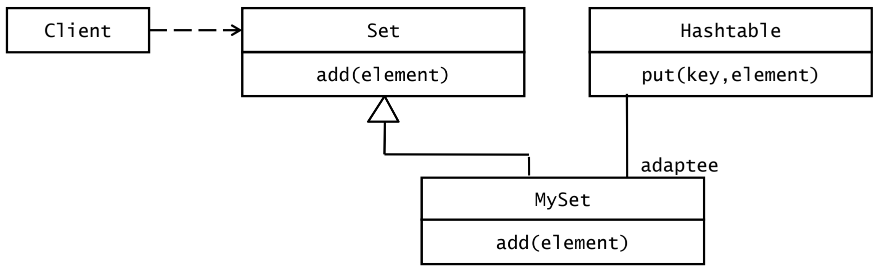
## Adapter pattern


Client
ClientInterface
Request()
LegacyClass
ExistingRequest()
The client sees only the target interface
adaptee
The adapter implements the target interface
Adapter
Request()
The adapter delegates requests to the Adaptee

- Delegation is used to bind an **Adapter** and an **Adaptee**
- An **Adapter** class implements the **ClientInterface** expected by the client. It delegates requests from the client to the **LegacyClass** and performs any necessary conversion.
- **ClientInterface** could be a Java interface, or an abstract class

## Adapter Pattern

- Example: Implementing a set using a hashtable (e.g. if Java had no set class but does have a hashtable class)



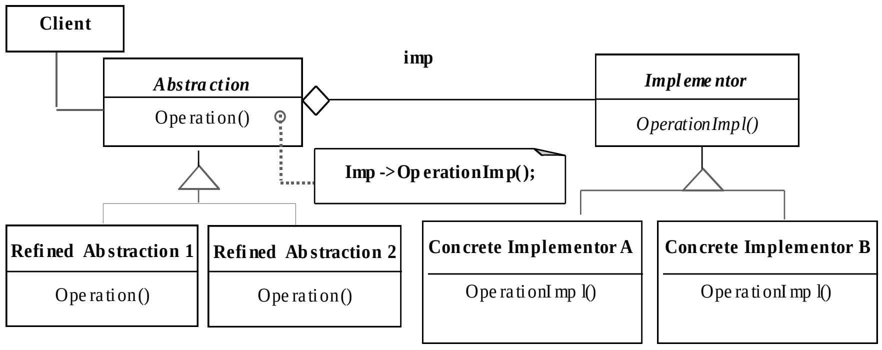
## Exercise

- Our client code uses a Calculator library with an Add method that takes two integers and returns the sum. We upgraded the library and now it takes two floats. Rather than change all the code in the client show using UML how the Adapter pattern could be used instead.

## Bridge Pattern

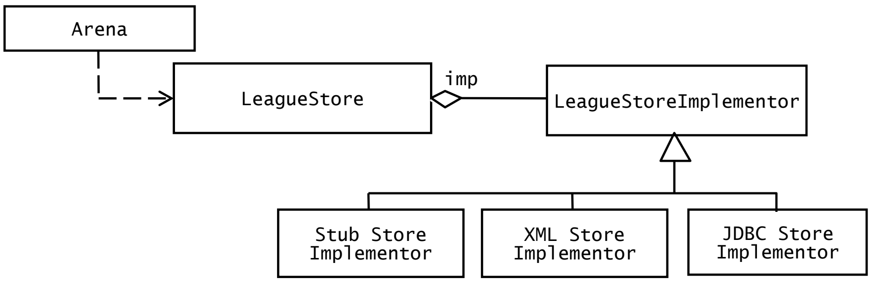
- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”. (From [Gamma et al 1995])
- The bridge pattern is used to provide multiple implementations under the same interface.
  - Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- Also known as a Handle/Body pattern.
- Allows different implementations of an interface to be decided upon dynamically.

## Bridge Pattern



## Bridge Pattern Example

- Abstracting how to perform database activity for storing tournaments



## Adapter vs Bridge

- Similarities:
  - Both are used to hide the details of the underlying implementation.
- Difference:
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they're designed (reengineering, interface engineering).
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
    - Green field engineering of an "extensible system"
    - New "beasts" can be added to the "object zoo", even if these are not known at analysis or system design time.

## Exercise

Draw the UML diagram for this pseudocode and identify the pattern

```
class Main
  Names n = new Names()
  n.add("Myra Mains")
  n.add("Terry Aki")
  n.add("Stu Pidd")


class Names
  private List namelist = new ArrayList()
  // private List namelist = new LinkedList()
  void add(string name)
      namelist.add(name)
  int count()
      return namelist.count


 interface List
    void add(string name)
    int count()
```

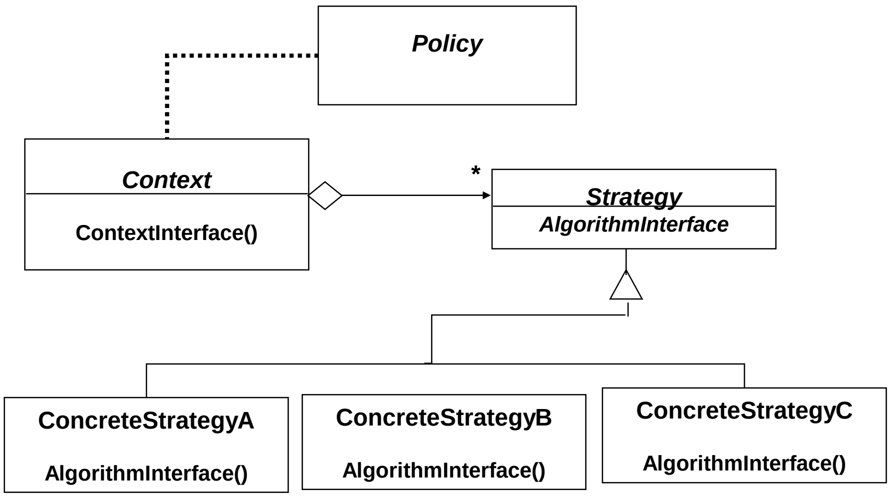
```
class ArrayList implements List
    private data[]
    void add(string name)
        data[i] = name
         …
    int count()
        return size
```

```
class LinkedListList implements List
    private Node next
    void add(string name)
        head.data = name
        head.next = new Node()
         …
    int count()
        return nodeCount
```

## Strategy Pattern

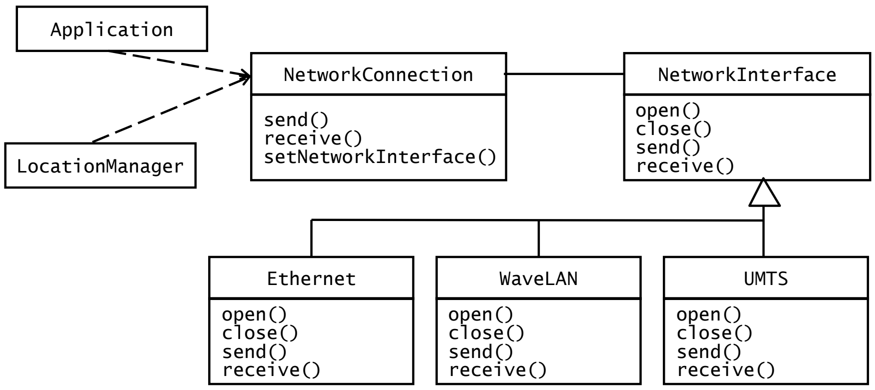
- The Strategy Design Pattern is similar to the Bridge pattern, but context drives selection of which implementation to use
- Consider a mobile application that needs to switch its wireless protocol based upon context
  - Bluetooth
  - 802.11B
  - Mobile phone network

## Strategy Pattern



**`Policy` decides which `Strategy` is best given the current `Context`**

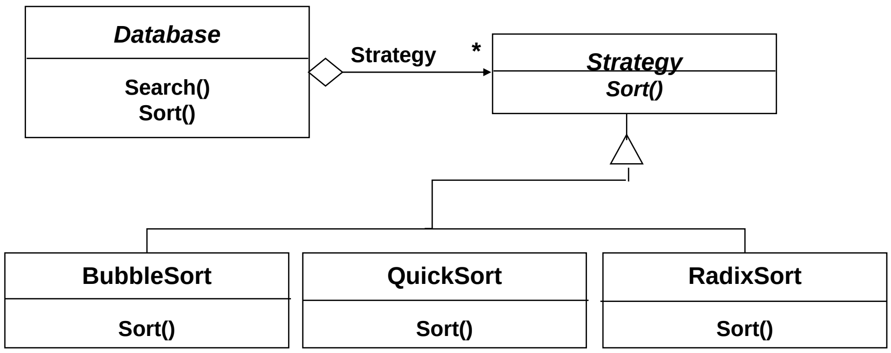
## Strategy Example



LocationManager configures NetworkConnection with a specific NetworkInterface based on the current location.

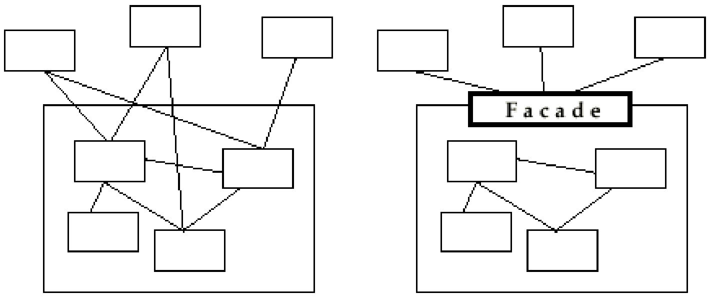
Application uses send/receive independent of concrete interface.

## Applying a Strategy Pattern in a Database Application



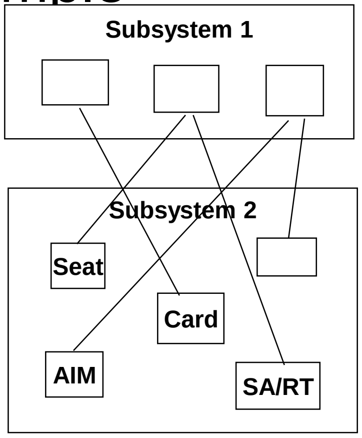
## Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide  a closed architecture



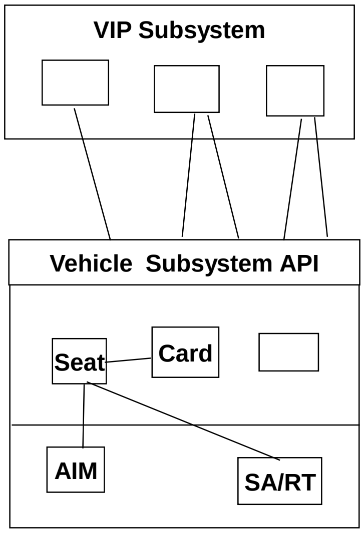
## Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is “Ravioli Design”
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code



## Realizing an Opaque Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a facade is used the subsystem can be used in an early integration test
  - We need to write only a driver



## Factory Pattern

- Most common factory pattern is the Abstract Factory
- Here we cover a similar Factory pattern that is commonly used
- The basic idea is instead of directly creating a "product" via new, ask the factory object for a new product, providing the information about the type of object needed
  - client uses the products as abstract entities without knowing about their concrete implementation

## Factory Motivation

- Consider a pizza store that makes different types of pizzas

```
Pizza pizza;

if (type == CHEESE)
        pizza = new CheesePizza();
else if (type == PEPPERONI)
        pizza = new PepperoniPizza();
else if (type == PESTO)
        pizza = new PestoPizza();

pizza.prepare();
pizza.bake();
pizza.package();
pizza.deliver();
```

This becomes unwieldy as we add to our menu

This part stays the same

Idea: pull out the creation code and put it into an object that only deals with creating pizzas - the PizzaFactory

## Factory Motivation

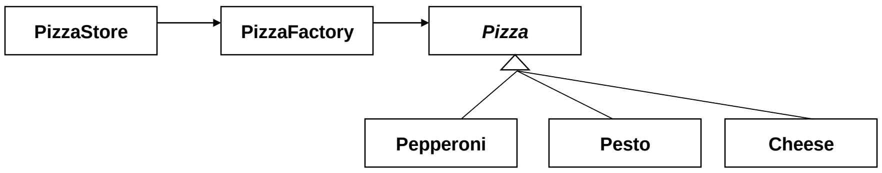
```
public class PizzaFactory
{
   public Pizza createPizza(int type)
   {
     Pizza pizza = null;
     if (type == CHEESE)
         pizza = new CheesePizza();
     else if (type == PEPPERONI)
         pizza = new PepperoniPizza();
     else if (type == PESTO)
         pizza = new PestoPizza();
     return pizza;
   }
}
```

Replace concrete instantiation with call to the PizzaFactory to create a new pizza →

Now we don't need to mess with this code if we add new pizzas

```
Pizza pizza;
PizzaFactory factory;

...

pizza = factory.createPizza(type);

pizza.prepare();
pizza.bake();
pizza.package();
pizza.deliver();
```

## Pizza Classes

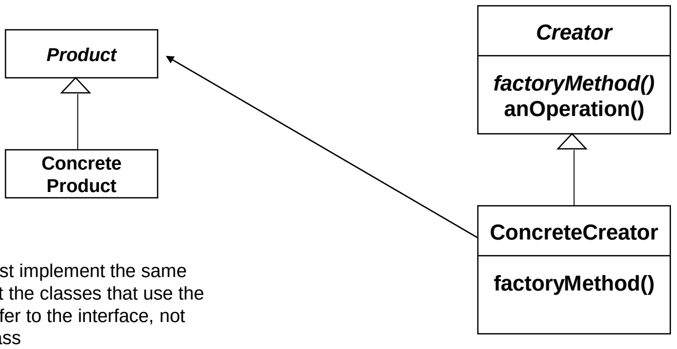


This is not quite the formal Factory pattern, to do so we would need an abstract PizzaFactory class.

First, the pattern:

## Factory Pattern

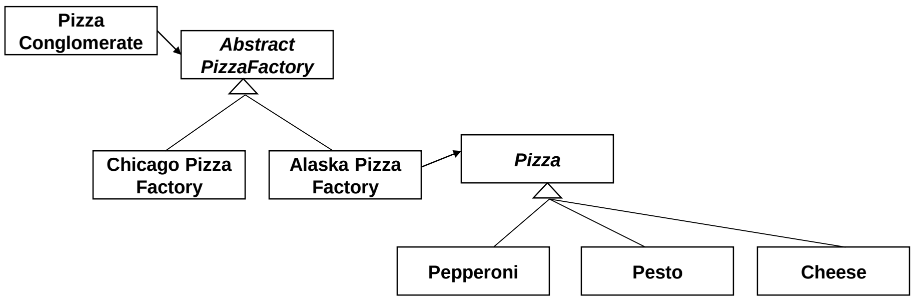
The Creator class contains implementations for all methods to manipulate products, except for creating them via factoryMethod

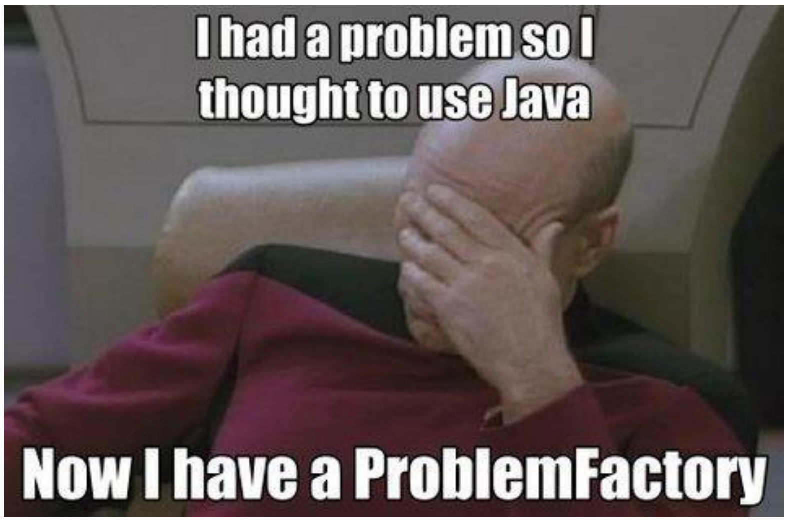


All products must implement the same interface so that the classes that use the products can refer to the interface, not the concrete class

The ConcreteCreator is the only class that can create concrete products returned by factoryMethod()

## Pizza Factory Classes




I had a problem so I thought to use Java
Now I have a ProblemFactory

## Null Object Pattern

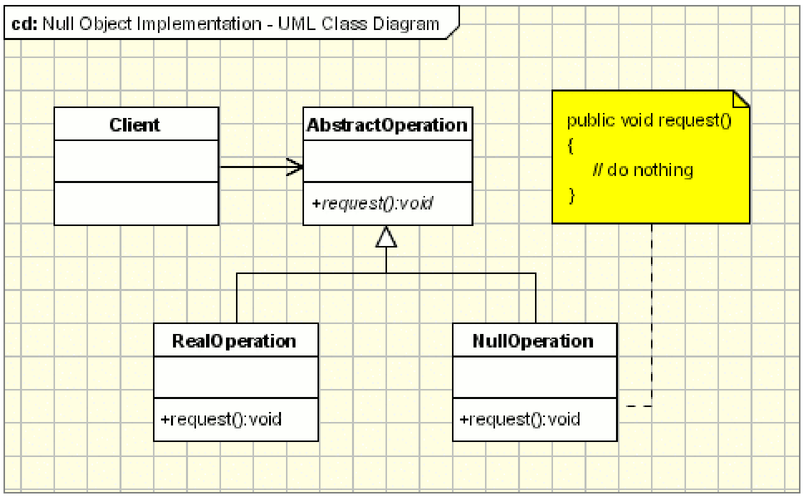
- Motivation
  - Depending on the context, a system may or may not need to invoke some functionality
  - Ex:

```
 if (logger != null)
{
   logger.callStuff();
     . . .
}
```

  - Idea: Introduce a Null object with non-behavior

```
logger.callStuff();
```

## Null Object Pattern



## Command Pattern: Motivation

- Say you have a remote control with three buttons
  - You would like to be able to walk around and press the buttons to turn on/off different devices
  - However, each device you want to control has a different interface for the power command
    - Ceiling Fan: OnOff();
    - Garage Door: OpenClose();
    - Television: TogglePower();

## Command Pattern Motivation

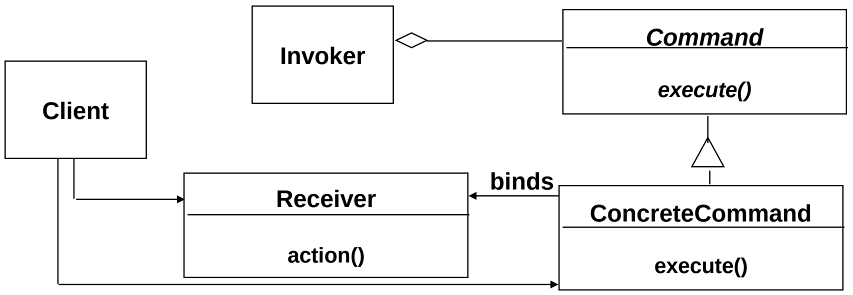
- Approach that works but very static:

```
if (buttonPress == 0)
       TogglePower();   // TV
else if (buttonPress == 1)
       OpenClose();     // Garage
else if (buttonPress == 2)
       OnOff();         // Fan

Etc.
```

More flexible and easier to use: Create an object, the **command object**, that encapsulates the desired request, and have the user invoke the request from the command object. In this case we may have 3 command objects in an array:

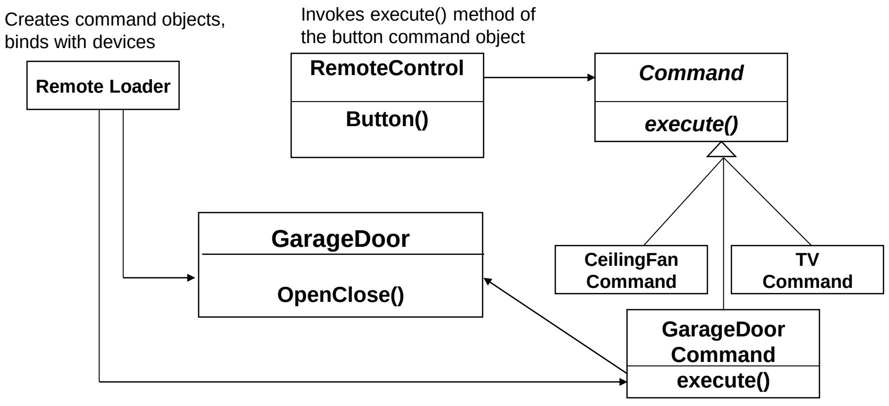
**Button[buttonPress].execute();**

## Command pattern



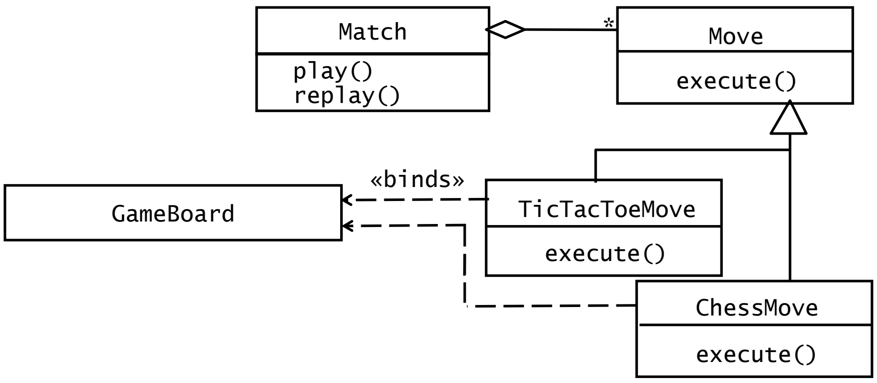
- **Client** creates a **ConcreteCommand** and binds it with a **Receiver.**
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
- The **Invoker** has the responsibility to do the command (“execute” or “undo”)**.**

## Command Pattern for Remote



execute() for each concrete command would use delegation to the corresponding device, e.g.
garagedoor.OpenClose()
or tv.TogglePower()

## Applying the Command design pattern to Game Matches



Match only calls Move, which executes, undoes, stores commands

## Command pattern Applicability

"Encapsulate a request as an object, thereby letting you
- parameterize clients with different requests,
- queue or log requests, and
- support undoable operations."

- Uses:
  - Undo queues, can add now since each command is sent through a command object and we can create a history of commands within this object
  - Database transaction buffering

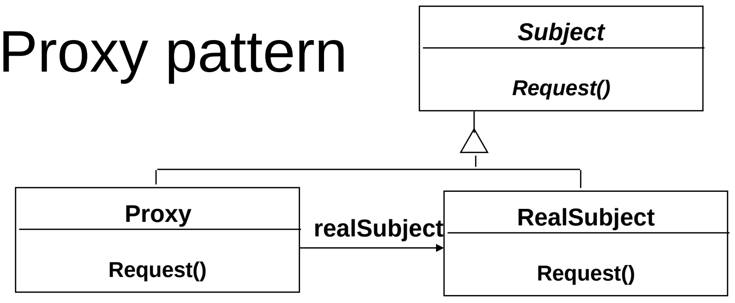
## Proxy Pattern: Motivation

With you as my Proxy, I'll be able to triple the amount of lunch money I can extract from friends!

# Proxy Pattern

- A proxy acts as an intermediary between the client and the target object
  - Why? Target may be inaccessible (network issues, too large to run, resources…)
- The proxy object has the same interface as the target object
  - The proxy has a reference to the target object and forwards (delegates) requests to it
- Useful when more sophistication is needed than a simple reference to an object (i.e. we want to wrap code around references to an object)

## Proxy pattern



- Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject.**
- Delegation is used to catch and forward any accesses to the **RealSubject** (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.

## Example: Virtual proxy

- Say your application needs to sometimes load and display large images
  - Expensive to load an image each time
- Virtual proxy
  - One instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object.

## Image Proxy (1 or 3)

```
interface Image {
   public void displayImage();
}

class RealImage implements Image {
   private String filename;
   public RealImage(String filename) {
       this.filename = filename;
       System.out.println("Loading   "+filename);
   }
   public void displayImage() { System.out.println("Displaying "+filename); }
}
```

## Image Proxy (2 of 3)

```
class ProxyImage implements Image {
   private String filename;
   private RealImage image = null;

   public ProxyImage(String filename) { this.filename = filename; }
   public void displayImage() {
       if (image == null) {
           image = new RealImage(filename); // load only on demand
       }
       image.displayImage();
   }
}
```

## Image Proxy (3 of 3)

```
class ProxyExample {
   public static void main(String[] args) {
       ArrayList<Image> images = new ArrayList<Image>();
       images.add( new ProxyImage("HiRes_10GB_Photo1") );
       images.add( new ProxyImage("HiRes_10GB_Photo2") );
       images.add( new ProxyImage("HiRes_10GB_Photo3") );

       images.get(0).displayImage(); // loading necessary
       images.get(1).displayImage(); // loading necessary
       images.get(0).displayImage(); // no loading necessary; already done
       // the third image will never be loaded - time saved!
   }
}
```

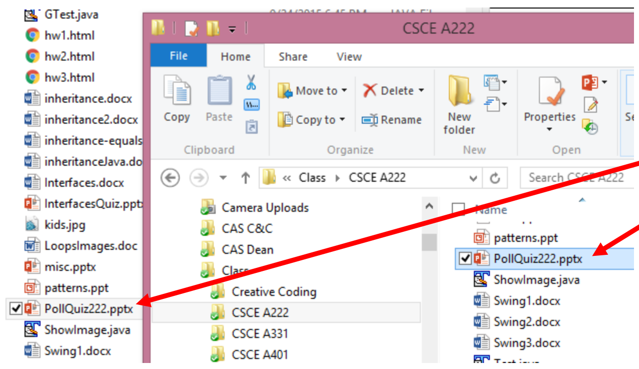
## Observer pattern

- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
- Also called "Publish and Subscribe"

- Uses:
  - Maintaining consistency across redundant state
  - Optimizing batch changes to maintain consistency

## Observer pattern (continued)

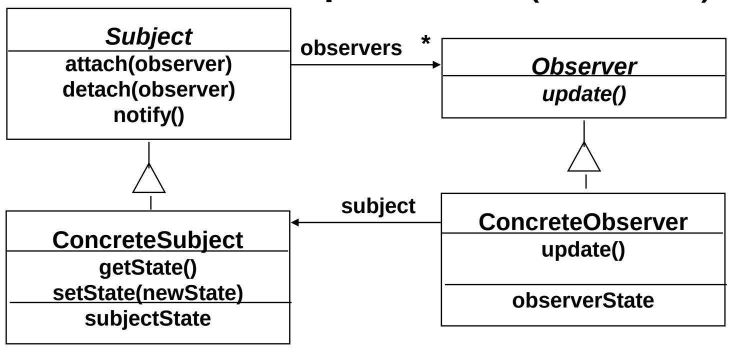
**Observers** **Subject**



patterns.ppt

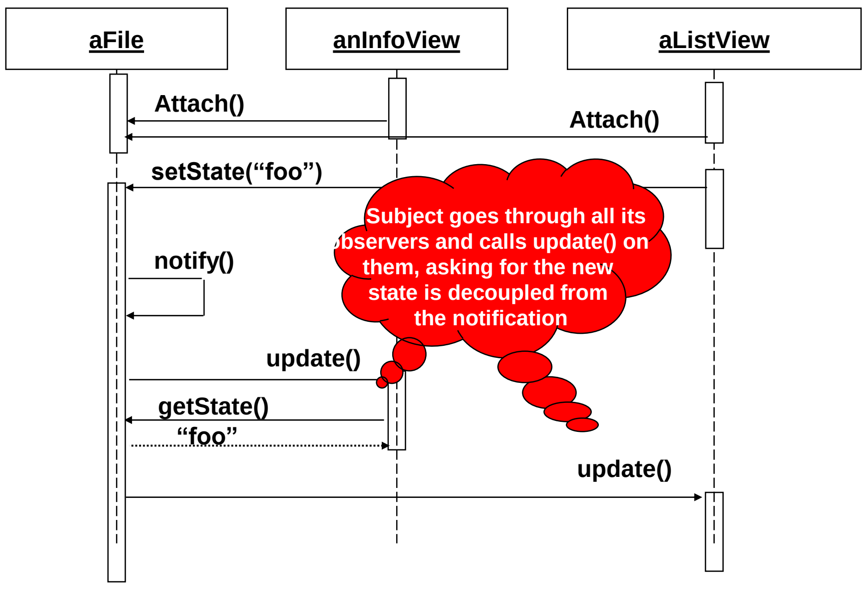
change name to foo.pptx

# Observer pattern (cont'd)



- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can  be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) ***not*** an interface.

Sequence diagram for scenario:
Change filename to "foo"


aFile
anInfoView
aListView
Attach()
Attach()
setState("foo")
notify()
update()
getState()
"foo"
update()
Subject goes through all its observers and calls update() on them, asking for the new state is decoupled from the notification

## Which Design Pattern Applies?

| Phrase | Design Pattern |
|---|---|
| “Manufacturer independence”, “Platform Independence” | Abstract Factory |
| “Must comply with existing interface”, “Must reuse existing component” | Adapter |
| “Must support future protocols” | Bridge |
| “All commands should be undoable”, “All transactions should be logged” | Command |
| “Policy and mechanisms should be decoupled”, “Must allow different algorithms to be interchanged at runtime” | Strategy |

## Conclusion

- Design patterns
  - Provide solutions to common problems.
  - Lead to extensible models and code.
  - Can be used as is or as examples of interface inheritance and delegation.

- Design patterns solve all your software engineering problems ☺