Remaining GRASP Patterns and Then GoF Patterns

Lect 19--20

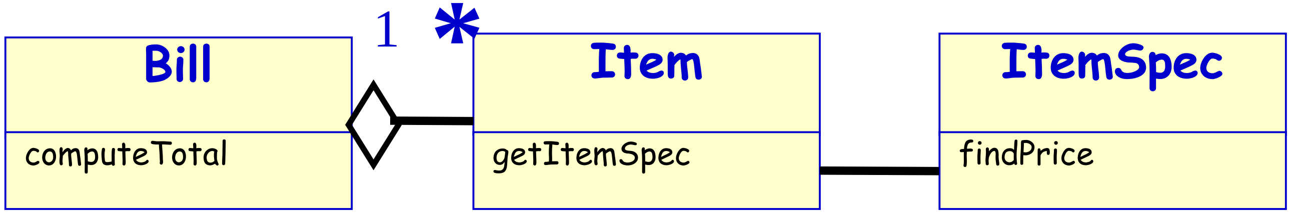
9-10-2023

GRASP Patterns

- GRASP patterns essentially suggest how to assign responsibilities to classes:
  - **Warning: Some Grasps tend to be vague and need to be seen as guidelines rather than solutions.**

- What is a responsibility?
  - **A contract or obligation of a class**
  - Responsibilities can include behaviour, data storage, object creation, etc.
  - Usually fall into two categories:
    - **Doing**
    - **Knowing**

Law of Demeter



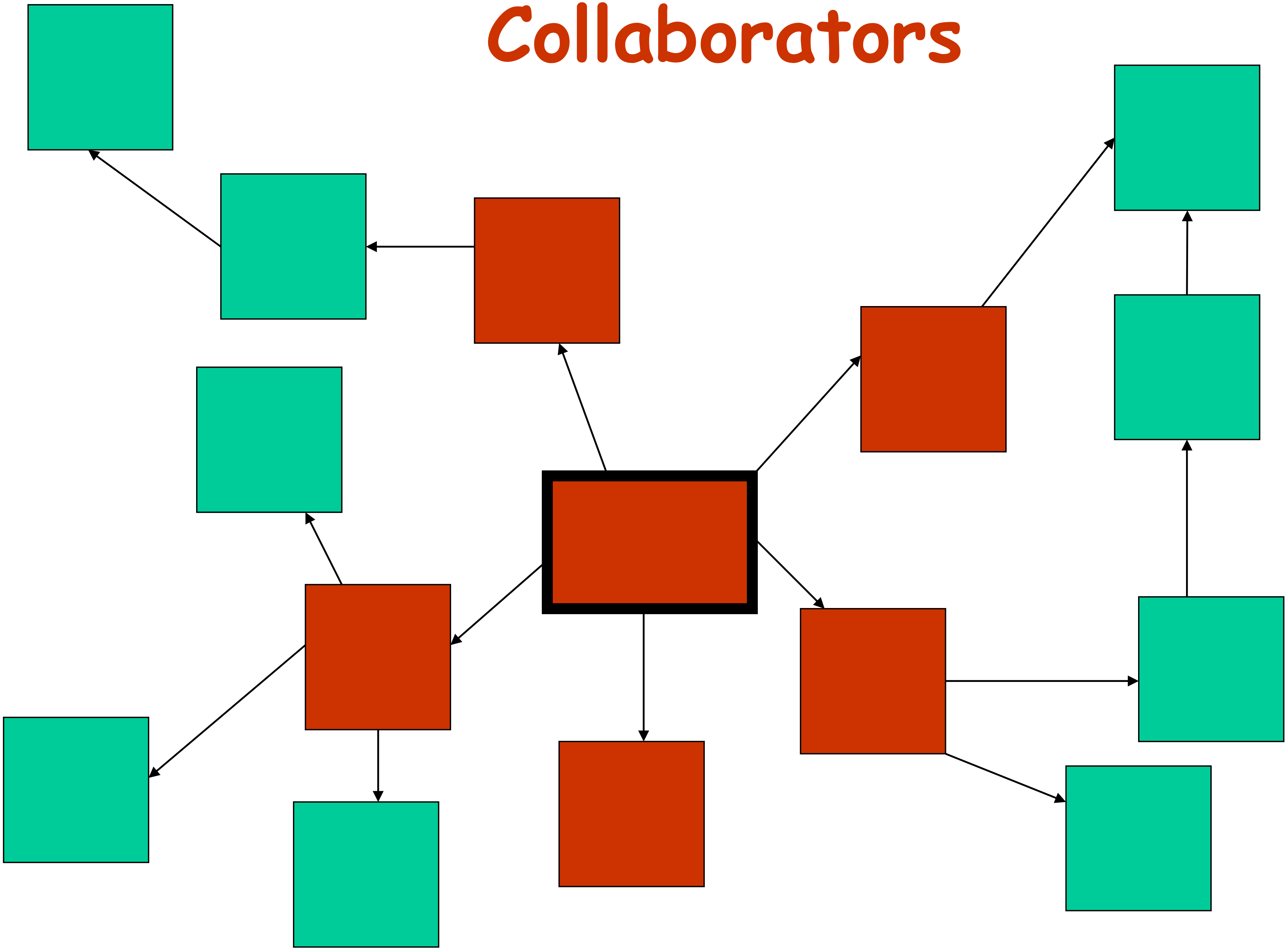
Problem:

- How to avoid a class from interacting with indirectly associated classes?

Solution:

- If two classes are not directly aware of each other:
  - Then these two classes should not directly interact.

Collaborators



Law of Demeter

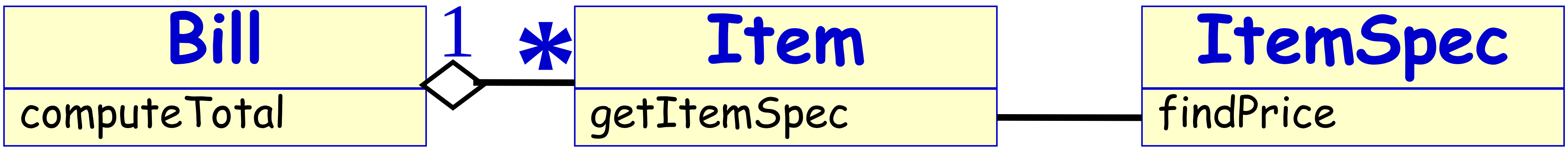
- In a method implementation, messages should only be sent to the following objects:
    - This object (or self)
    - An object parameter of the method
    - An object attribute of self
    - An object of a collection which is an attribute of self
    - An object created within the method

LD Violation: Hypothetical example

```
class A {
  private B b = new
 B();


  public void m() {
    this.b.c.foo();
  }
}
```

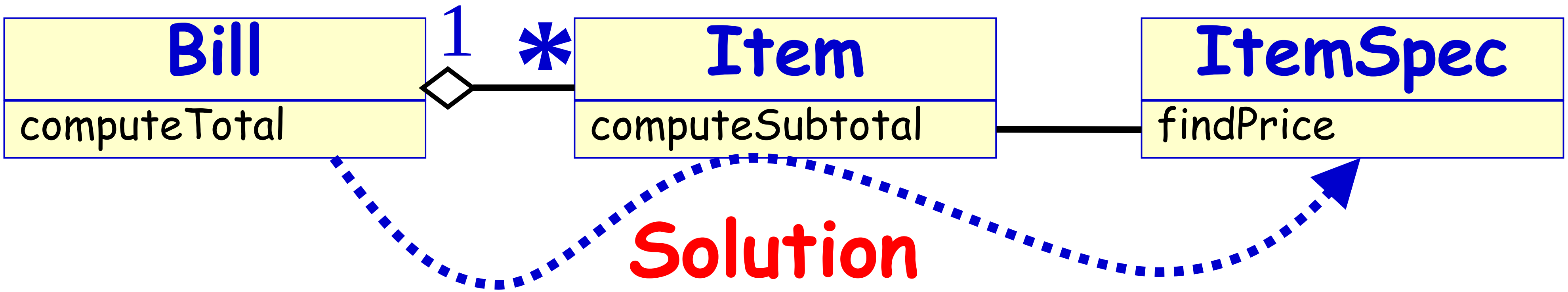
```
class B {
  C c;
}


class C {
  public void foo() {
  }
}
```

Law of Demeter: Check Violation



Bill
computeTotal
1
*
Item
getItemSpec
ItemSpec
findPrice

Violation: item.getItemspec().findprice()

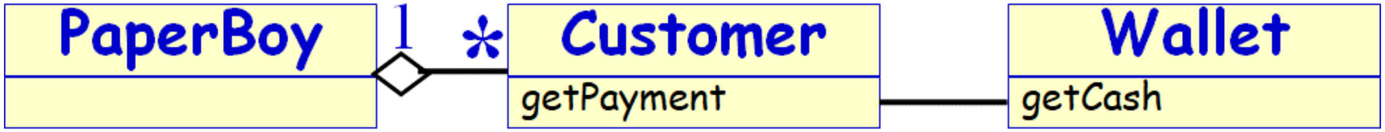


Bill
computeTotal
1
*
Item
computeSubtotal
ItemSpec
findPrice

Solution

LoD Programming Style

- In a class PaperBoy,
  - **do not use** customer.getWallet().getCash(due);
  - **rather use** customer.getPayment(due); and in customer.getPayment (due), use wallet.getCash(due)
- **Benefit:**



  - Easier analysis and maintainability
- **Tradeoff:**
  - More statements, but smaller/easier stmts
- **Experiments show significantly improved maintainability.**

The Law of Demeter: “Do not talk to strangers”

Don’t send messages to object refs returned from other method calls

```
public void movePlayer(int roll) {
  ...
  if (currentPlayer.square().isLastSquare()) {
    winner = currentPlayer;
  }
}
```

```
public void movePlayer(int roll) {
  ...
  if (currentPlayer.wins()) {
    winner = currentPlayer;
  }
}
```

Better

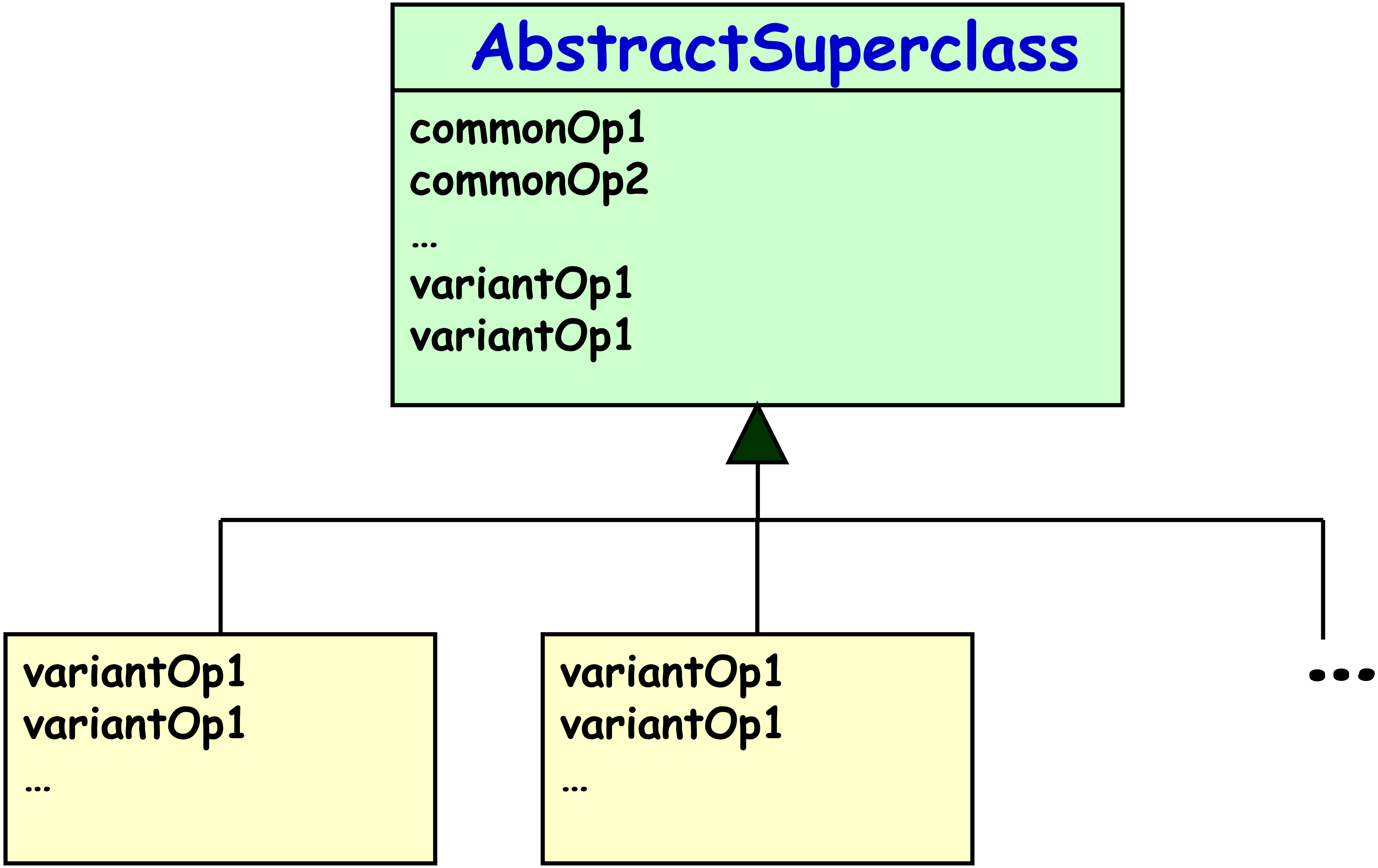
Law of Demeter: Final Analysis

- Reduces overall coupling in the design
- Adds a small amount of overhead in the form of indirect method calls

“The basic effect of applying the Law of Demeter is the creation of loosely coupled classes, whose implementation secrets are encapsulated. Such classes are fairly unencumbered, meaning that to understand the working of one class, you need not understand the details of many other classes.” Grady Booch

Polymorphism Pattern

- **Problem:**
  - How to handle alternative operations based on subtypes?
- **Solution:**
  - When alternate behaviours are selected based on the type of an object,
  - **Use polymorphic method call to select the required behaviour,**
  - **Rather than using if statement to test the type.**



AbstractSuperclass

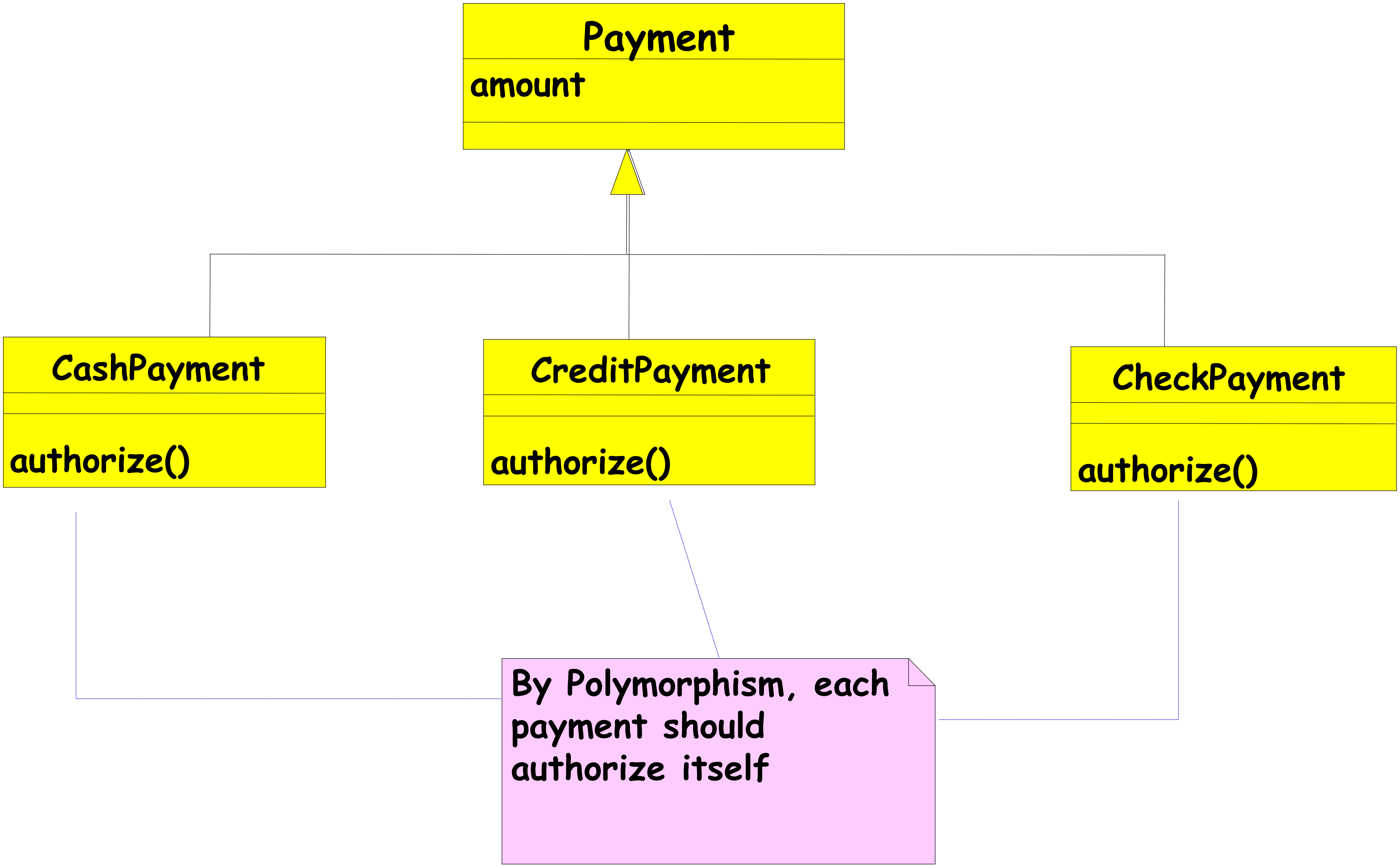
commonOp1
commonOp2
…
variantOp1
variantOp1

variantOp1
variantOp1
…

variantOp1
variantOp1
…

…

Polymorphism : Example



Payment

amount

CashPayment

authorize()

CreditPayment

authorize()

CheckPayment

authorize()

By Polymorphism, each payment should authorize itself

Polymorphism Pattern: Advantage

- Easily extendible as compared to using explicit selection logic…

GoF Patterns

What problems do GoF DPs solve?

- Good designs need to cope with change. Why?
  - Underlying technologies change
  - Business goals change
  - User expectations change
- How to cope with change?
  - Encapsulation
  - Inheritance
  - Polymorphism

GoF Pattern Classification

Creational	Structural	Behavioral
Abstract Factory	Adapter	Visitor
	Proxy	Command
Factory method	Bridge	Iterator
Builder	Composite	Mediator
Object pool	Decorator	Memento
Prototype	Façade	Observer
Singleton	Flyweight	State
Multiton		Template

Classification of Other Patterns

<table>
<tr><th>Concurrency</th><th>J2EE</th><th>Architectural</th></tr>
<tr><td>Active Object</td><td>Business Delegate</td><td>Layers</td></tr>
<tr><td>Balking</td><td>Composite Entity</td><td>Presentation-abstraction-control</td></tr>
<tr><td>Double checked locking</td><td>Composite View</td><td>Three-tier</td></tr>
<tr><td>Guarded suspension</td><td>Data Access Object</td><td>Pipeline</td></tr>
<tr><td>Monitor object</td><td>Fast Lane Reader</td><td>Implicit invocation</td></tr>
<tr><td>Reactor</td><td>Front Controller</td><td>Blackboard system</td></tr>
<tr><td>Read/write lock</td><td>Intercepting Filter</td><td>Peer-to-peer</td></tr>
<tr><td>Scheduler</td><td colspan="2">Model-View-Controller</td></tr>
<tr><td>Event-Based Asynchronous</td><td>Service Locator</td><td>Service-oriented architecture</td></tr>
<tr><td>Thread pool</td><td>Session Facade</td><td>Naked objects</td></tr>
<tr><td>Thread-specific storage</td><td>Transfer Object</td><td></td></tr>
</table>

GoF Patterns: A Classification

- **Creational patterns:**
  - When and how objects are instantiated?
- **Structural patterns:**
  - How objects are composed into larger groups?
- **Behavioral patterns:**
  - How are responsibilities distributed?

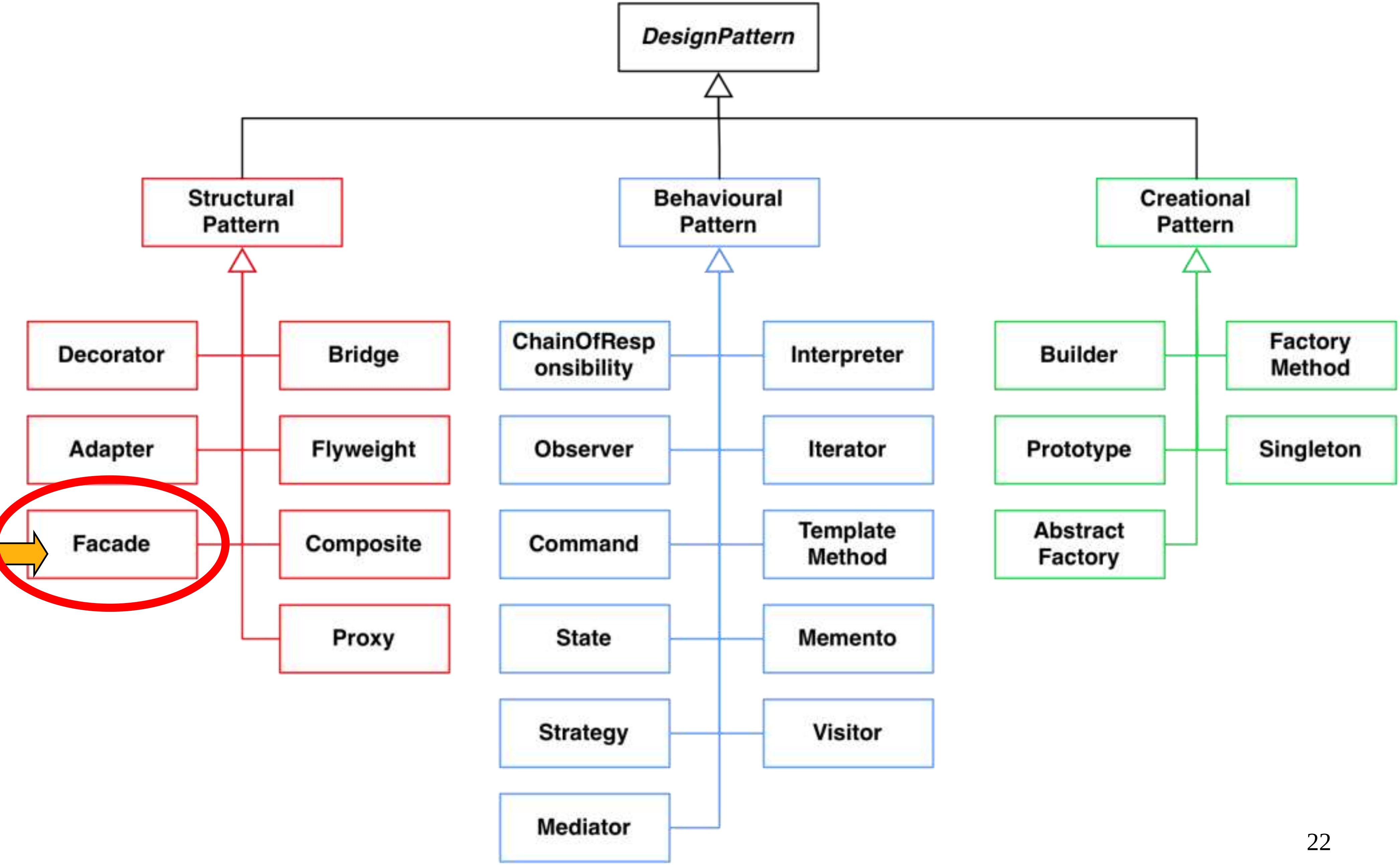
GoF Pattern Taxonomy

- **Structural Patterns**
  - Adapters, Bridges, Facades, and Proxies are variations of a single theme:
    - Reduce the coupling between two or more classes
    - **Introduce abstract classes to enable future extensions**

- **Behavioral Patterns**
  - Assignment of responsibilities among objects: Who does what?
  - Behavioral patterns concern cohesion and coupling.

- **Creational Patterns**
  - Separate a system from how its objects are created
  - **Increase the system's flexibility in terms of object creation.**

Taxonomy of GoF Patterns (23 Patterns)

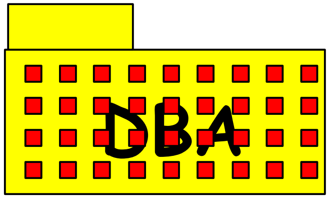


Facade Pattern

Facade Pattern

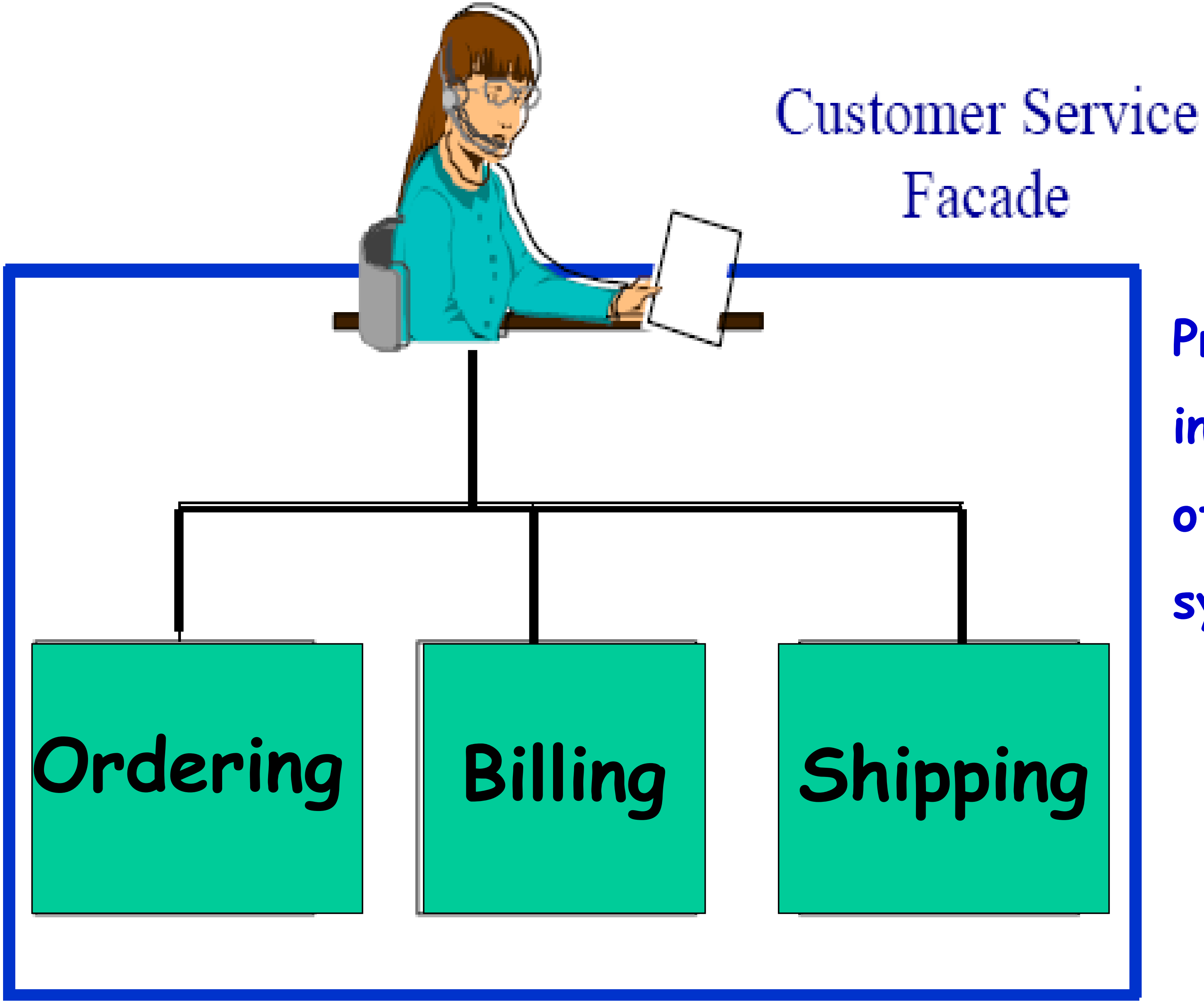
Problem: **How to avail services from a large number of classes present in a package?**

- **Context:** A package is a cohesive set of classes:
  - Example: RDBMS interface package



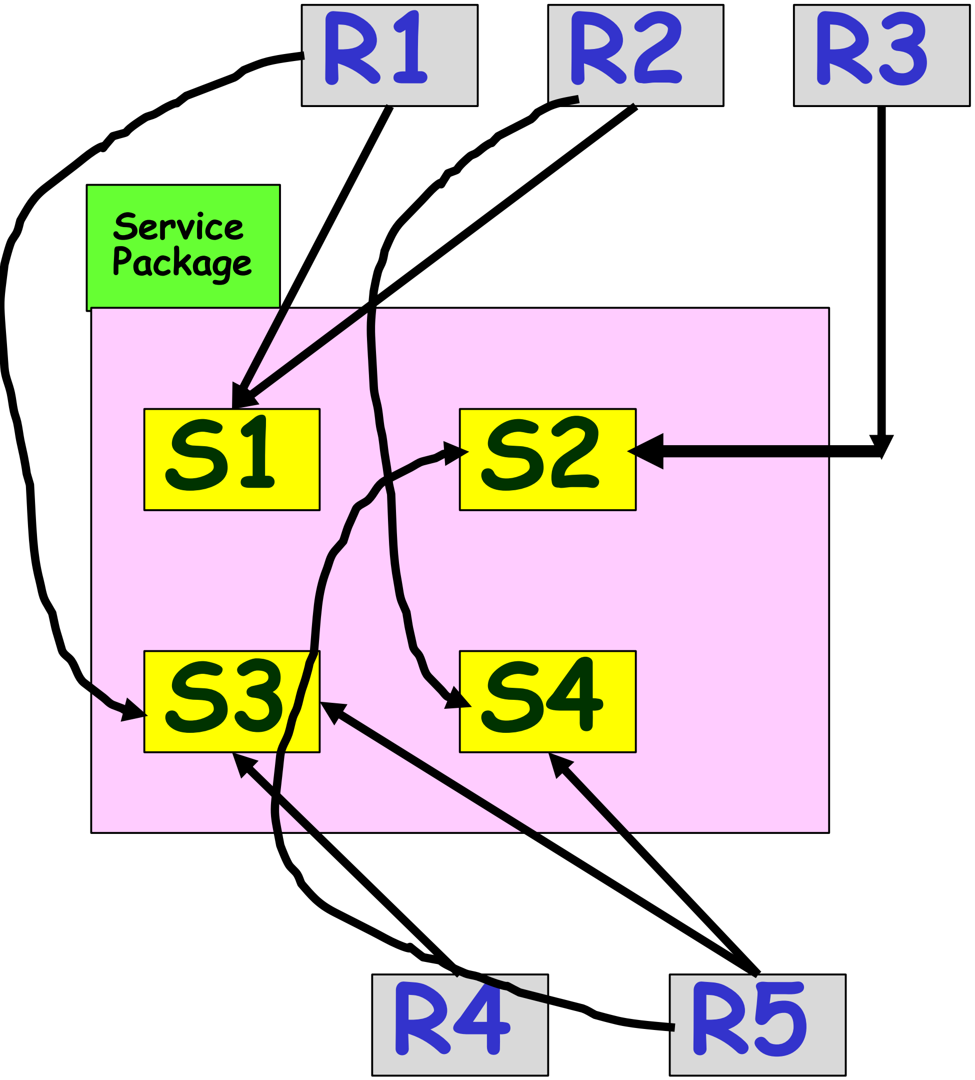
- **Solution**: A facade class (e.g. DBfacade) is created to provide a common interface to the services offered by the package.

Facade (Non software example)

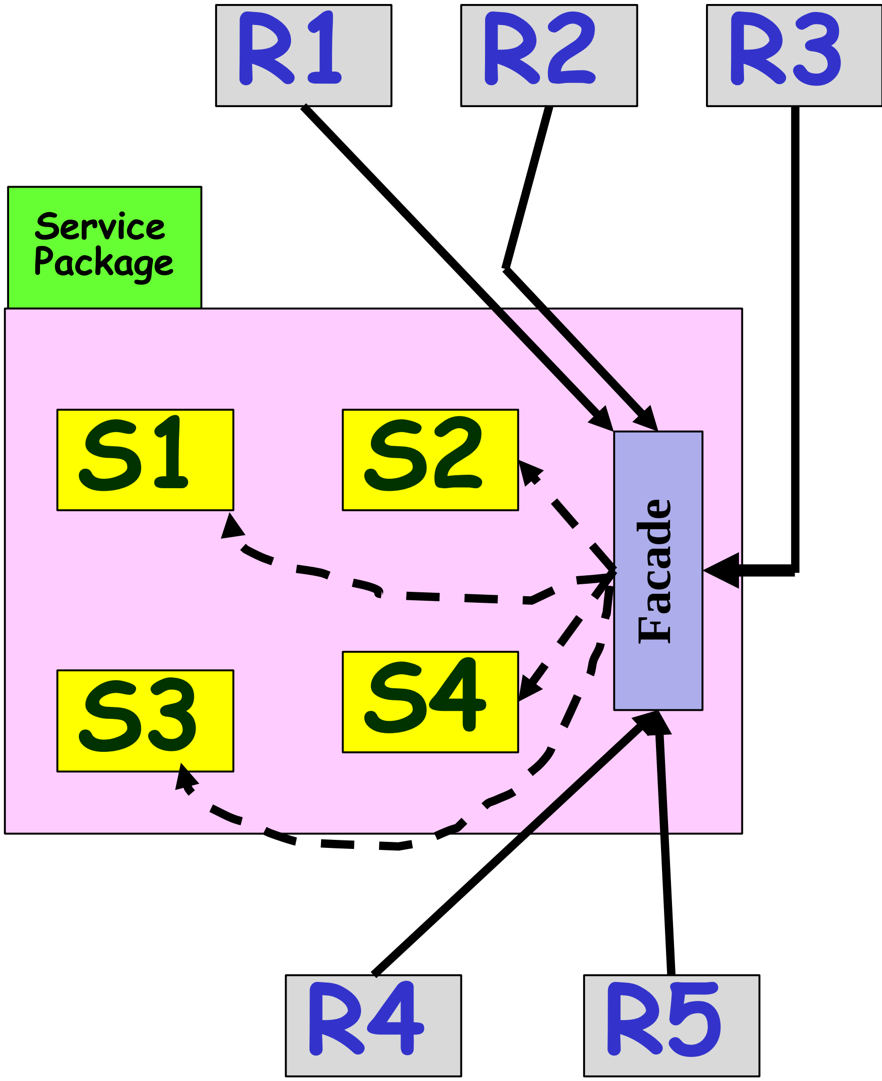


Provides a unified interface to a set of classes in a system.

Overview of Facade

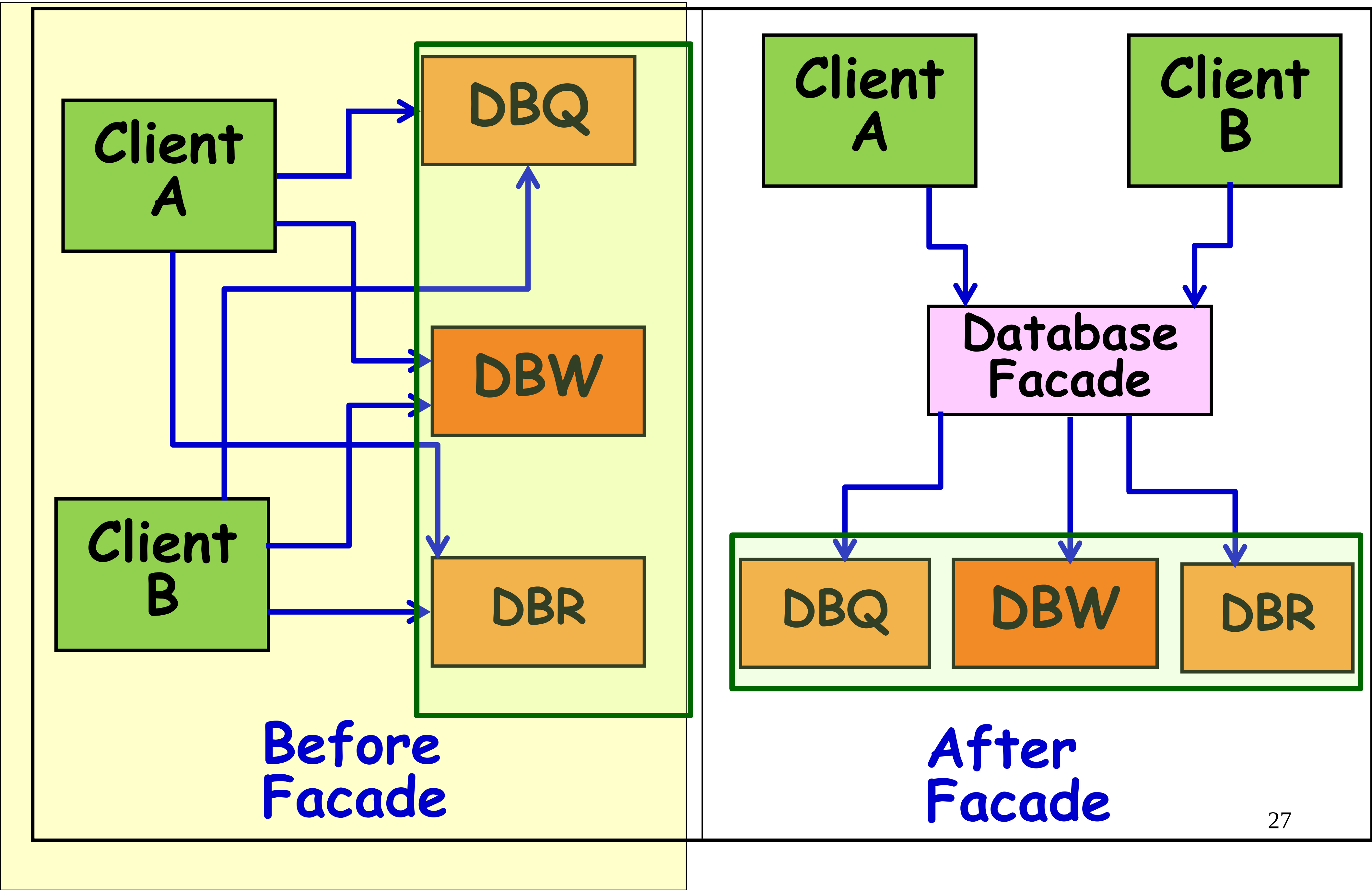


(a) Service invocation without a façade

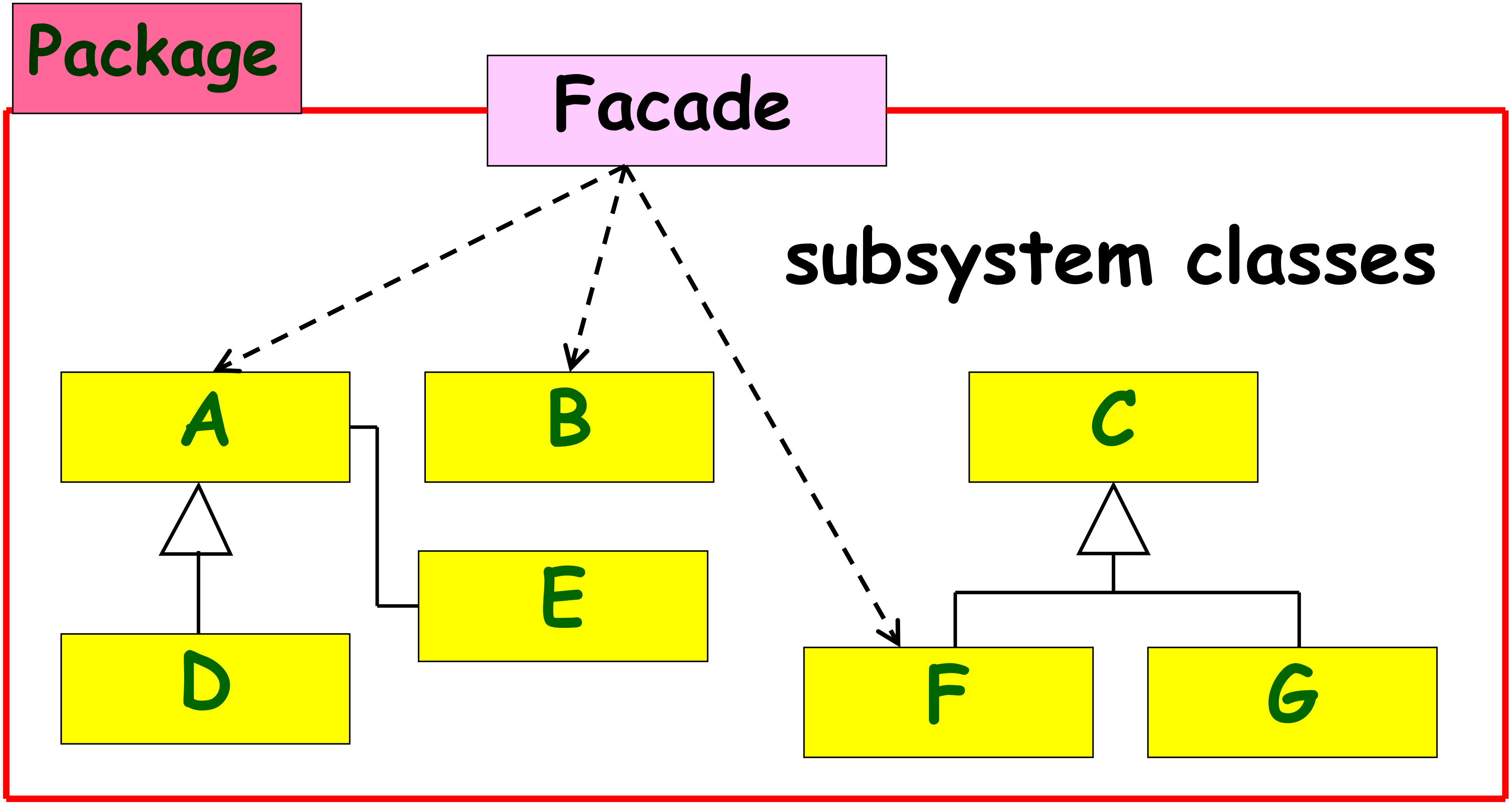


(b) Service invocation using a façade class

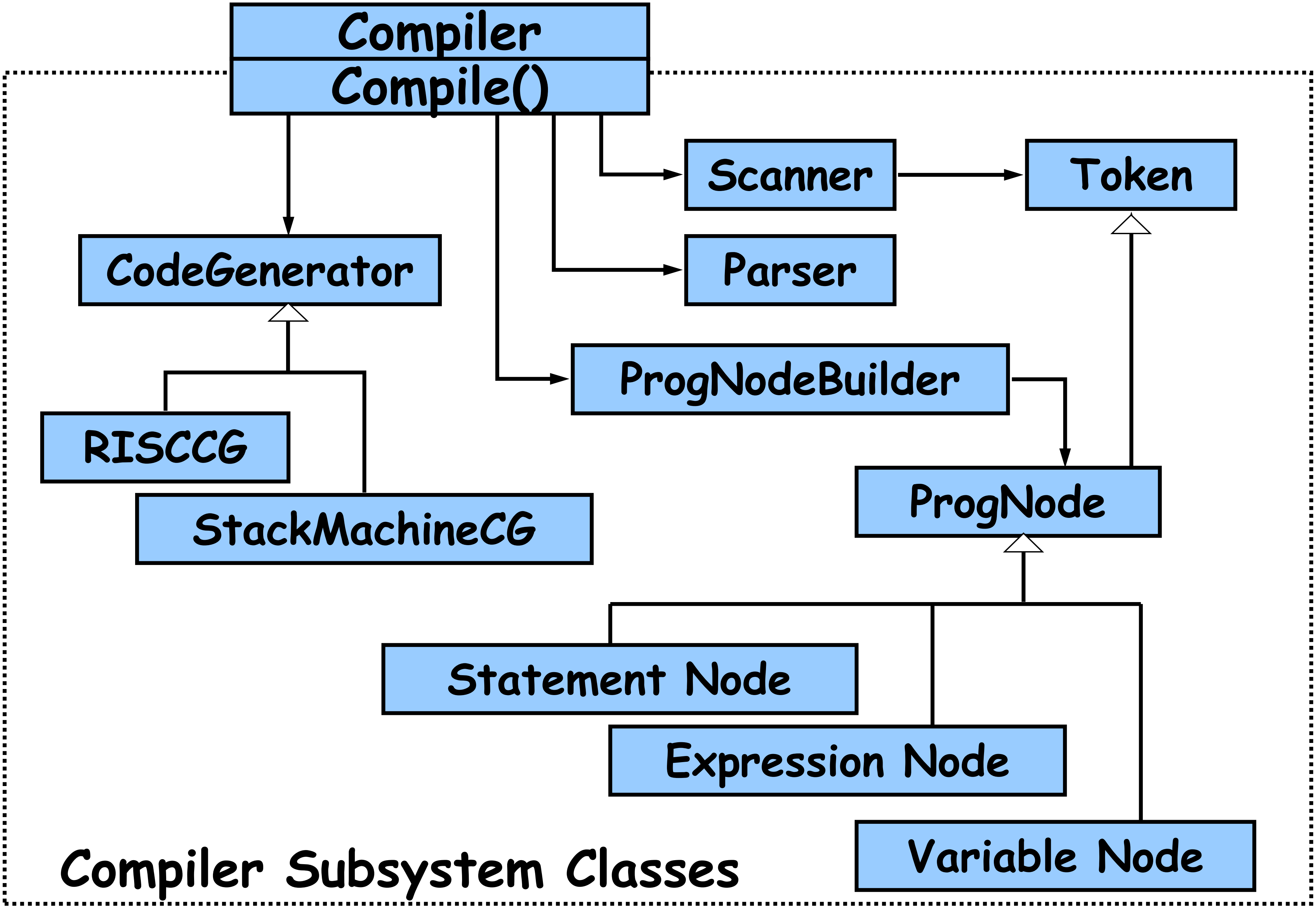
Before and After Using Facade



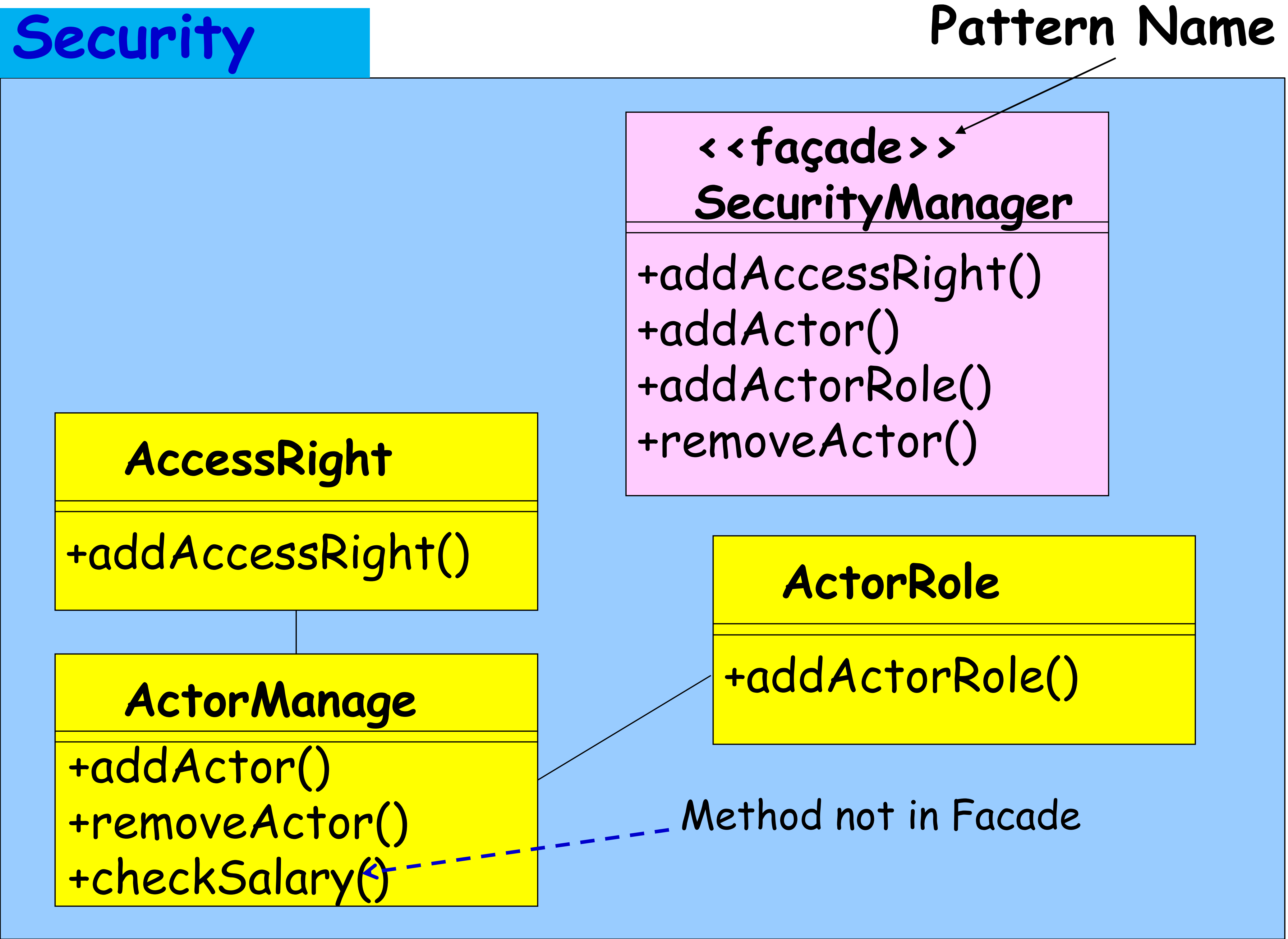
Facade: Structure



Facade Pattern: Example



Security



Pattern Name

<<façade>>
SecurityManager

+addAccessRight()
+addActor()
+addActorRole()
+removeActor()

AccessRight

+addAccessRight()

ActorRole

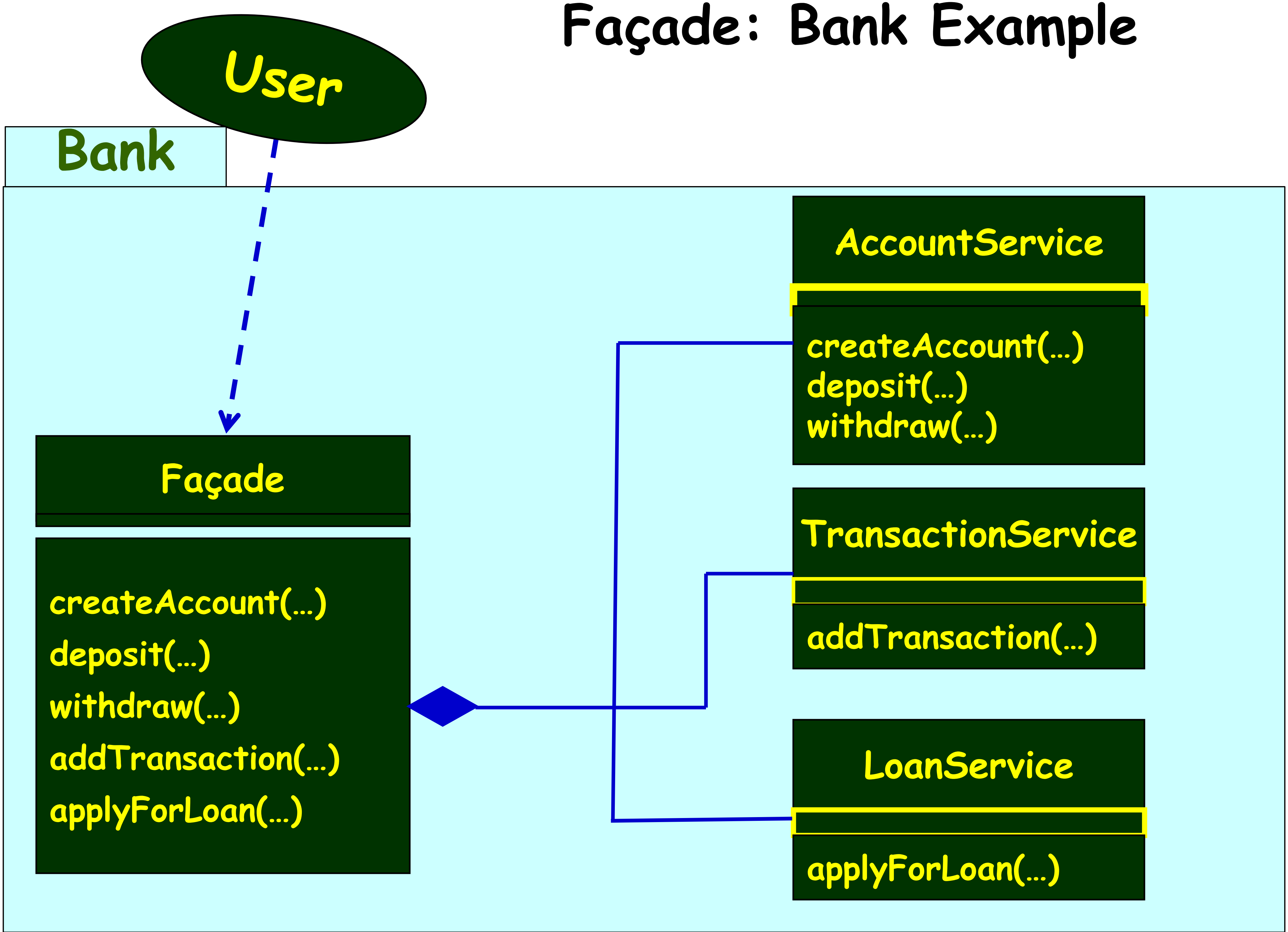
+addActorRole()

ActorManage

+addActor()
+removeActor()
+checkSalary()

Method not in Facade

Façade: Bank Example

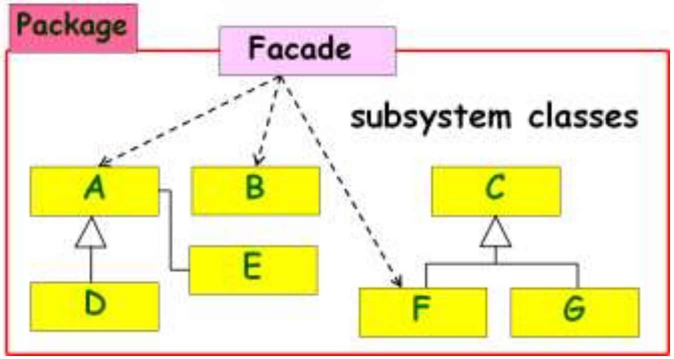


Façade: Bank Example

```
public class BankFaçade {
    private AccountService
    accountService;
    private TransactionService
    transactionService;
    private LoanService loanService;
    public addAccount(Account account) {
     accountService.addAccount(account);
    }
    public deposit(int accountId, float
    amount) {
     accountService.deposit(accountId,
    amount);
    }
    public withdraw(int accountId,
    float amount) {
     accountService.withdraw(accountId
    , amount);
    }
    public addTransaction(Transaction
    tx) {
     transactionService.addTransaction(
    tx);
    }
    public applyForLoan(Customer cust,
    LoanDetails loan) {
     loanService.apply(cust, loan);
    }
}
```

Discussions on Façade Pattern

- Clients communicate only with the Façade:



  - Façade forwards each request to the appropriate subsystem object(s).
  - Results in lowering overall coupling; clients need to know only about the Façade.

- Façade often does little more work than simply delegating requests to other classes inside the package!

Discussions on Façade Pattern

- Although subsystem objects perform actual work:
  - **Façade may have to translate between subsystem interfaces.**

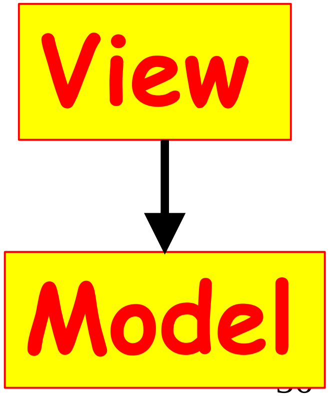
- Clients are aware of only the Façade:
  - Don't have to directly access the other objects in the package.

Model-View Separation Patterns

Model-View Separation Patterns --- Background

- Symptom of Poorly designed GUIs:
  - Classes with a large and incoherent set of responsibilities:
  - Encompass: GUI, Listening to events, domain logic, application logic, etc...

- We should *separate the concerns.*
  - Model and view objects
  - How should they communicate?

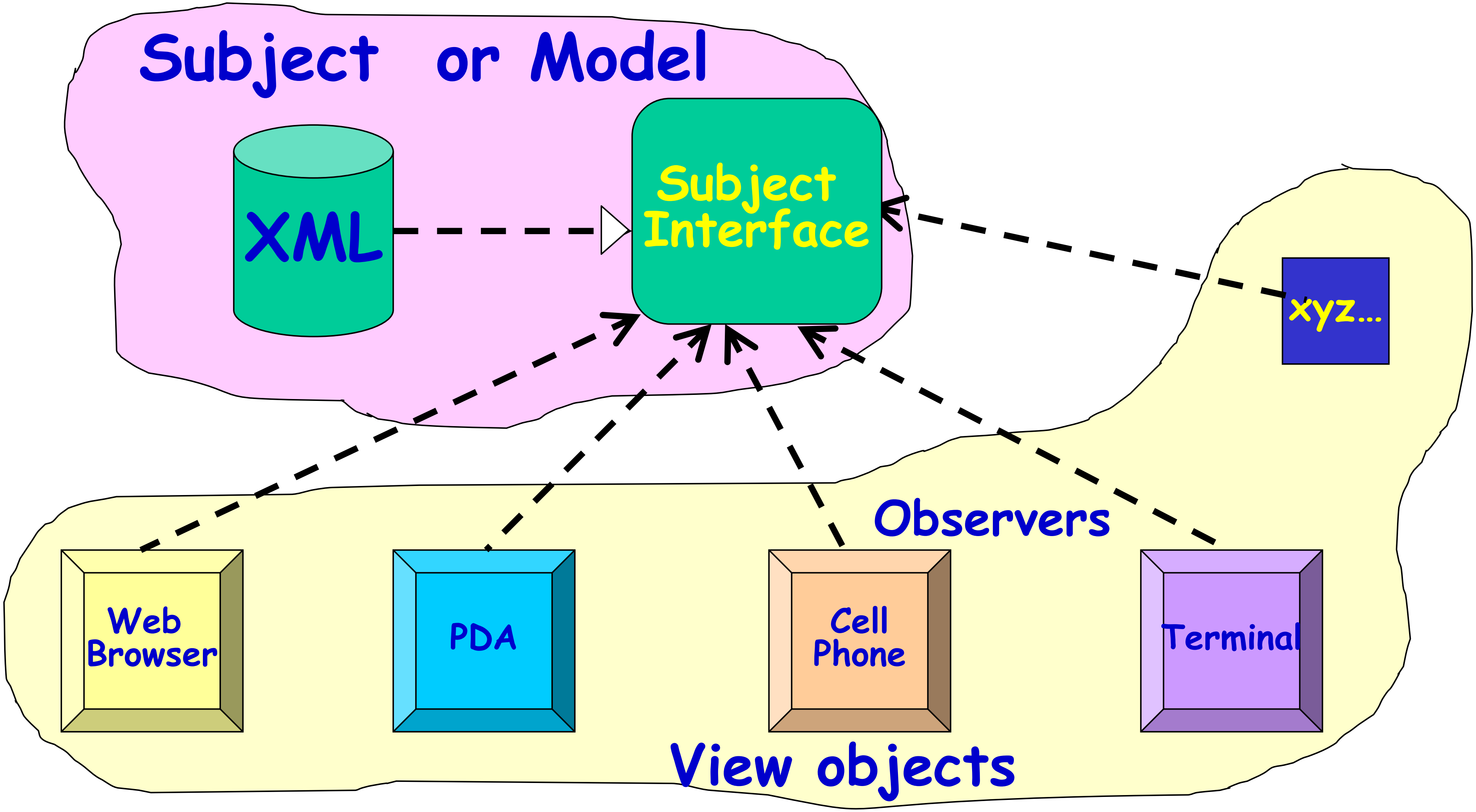


Model View Separation Patterns

- **Problem:**
  - How should the non-GUI classes communicate with the GUI classes and vice versa?

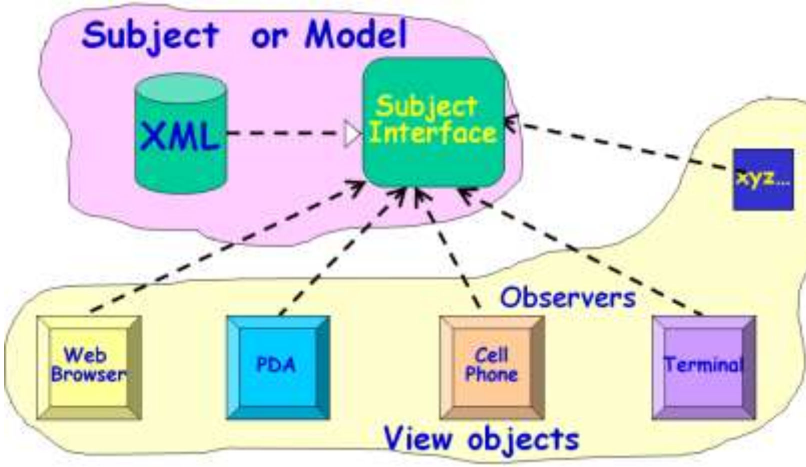
- **Solution:**
  - Several solutions exist --- each appropriate in a different context.

Simplistic Model View Communication



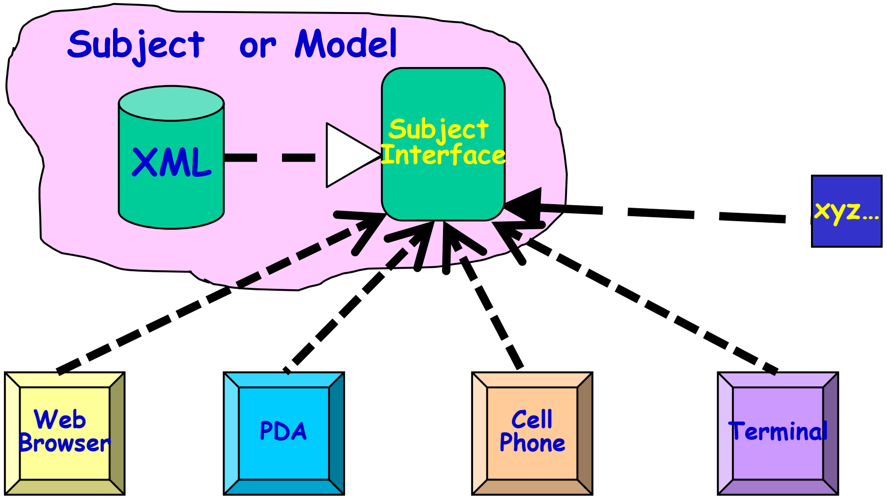
Model View Separation Patterns: Simple Solution Overview

- Model objects should not invoke services of view objects…
  - Vice-versa is OK, Why?
  - View objects are transient
  - Reuse, extension, maintenance are easier.
  - A change to a view object does not require change to the model object.



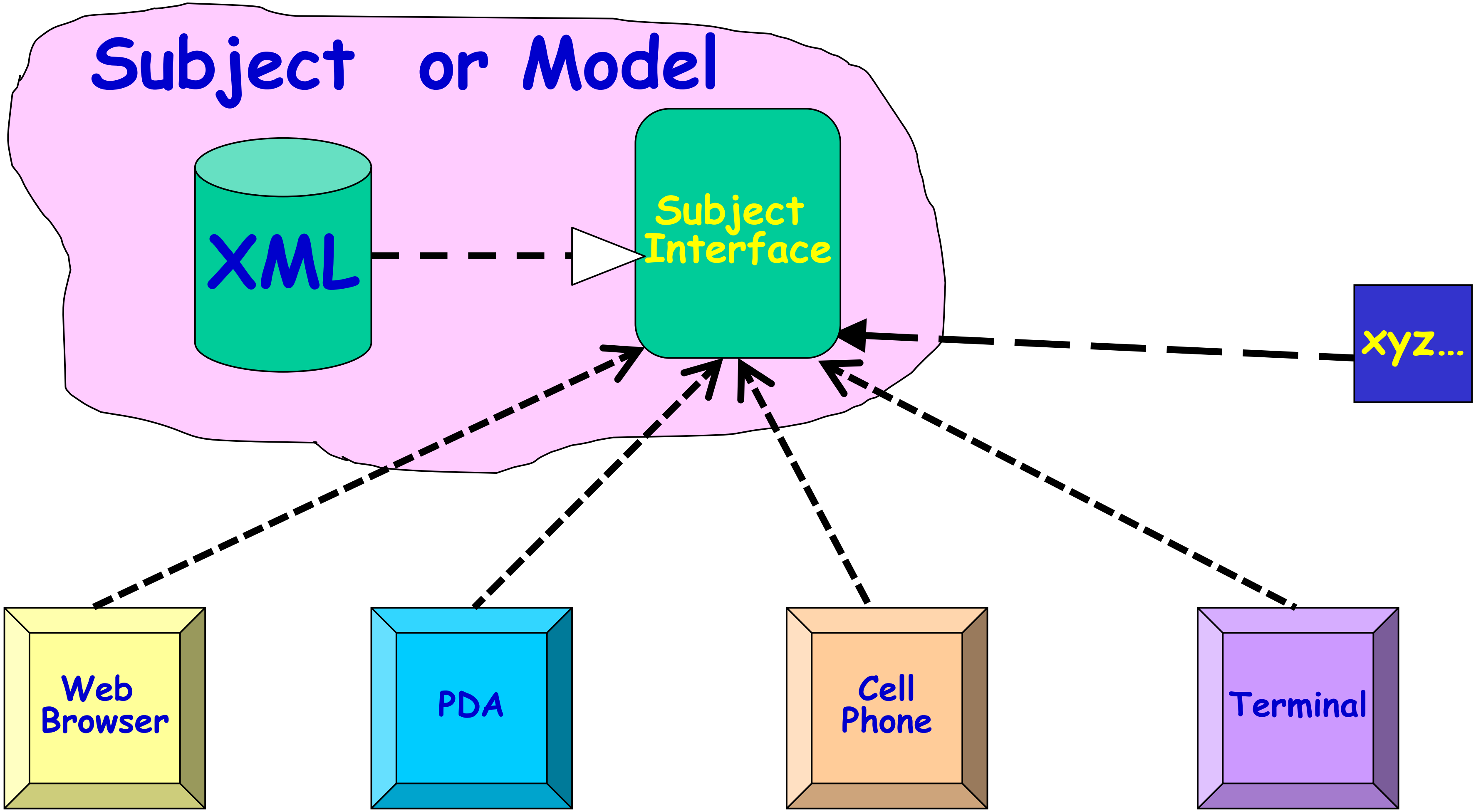
Solution 1: Pull from Above

- **Most obvious solution:**
  - View (boundary) objects directly invoke model objects.



- **Problem:**
  - Views can become inconsistent
  - View objects do not know when model state changes
  - How do users change the model?

Obvious Solution



Data is polled by various observers

Contexts in Which Pull from Above Does Not Work

- **Data changes asynchronously:**
  - Events in a simulation experiment, stock market alerts, network monitor events...
- **Some clients can change the model:**
  - In a computer game, a mouse event by a player to make a move.

Model-View Separation Solutions

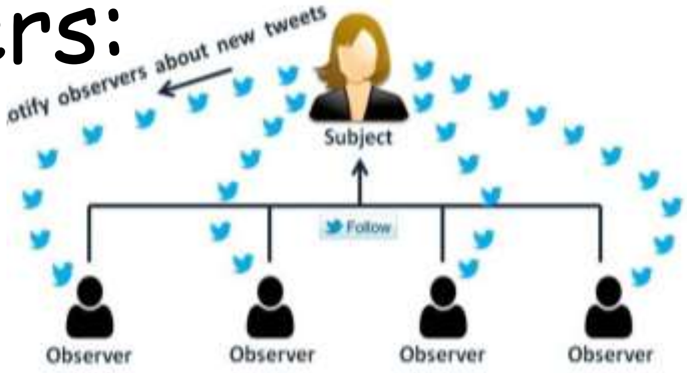
- **Pull from above:**
  - Does not work when model changes asynchronously
- **Observer pattern: (GoF)**
  - Multiple independent observers
- **Model-View Controller Pattern:**
  - Multiple independent observers
  - Multiple independent controllers
- **Publish-subscribe pattern:**
  - Large number of observers
  - Multiple models elements
  - Different observers interested in different classes of events

Observer Pattern: Problem

- When a model object changes state asynchronously and is accessed by several view objects:
  - **How should the interactions between the model and the view objects be structured?**
- **Solution:** Define a one-to-many dependency, so that when model changes state, all of its dependents are notified and updated automatically.

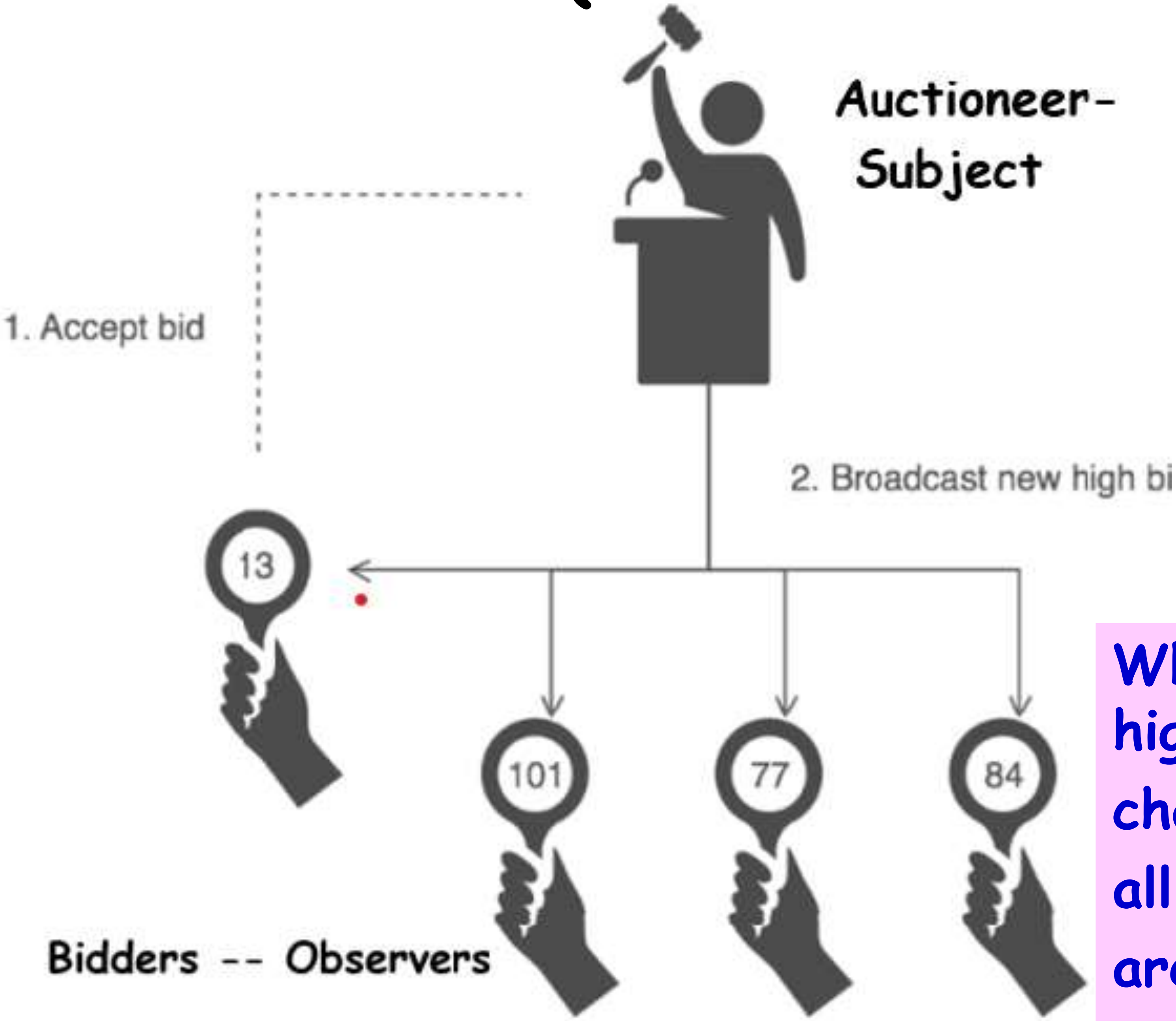
Observer Pattern: Context

- There could be many observers:
  - Also the number of observers may vary dynamically
  - **Each observer may react differently to the same notification…**



- Subject (model) should be as much decoupled from the observers as possible:
  - **Allow observers to change independently of the subject…**

Observer (Non software example)

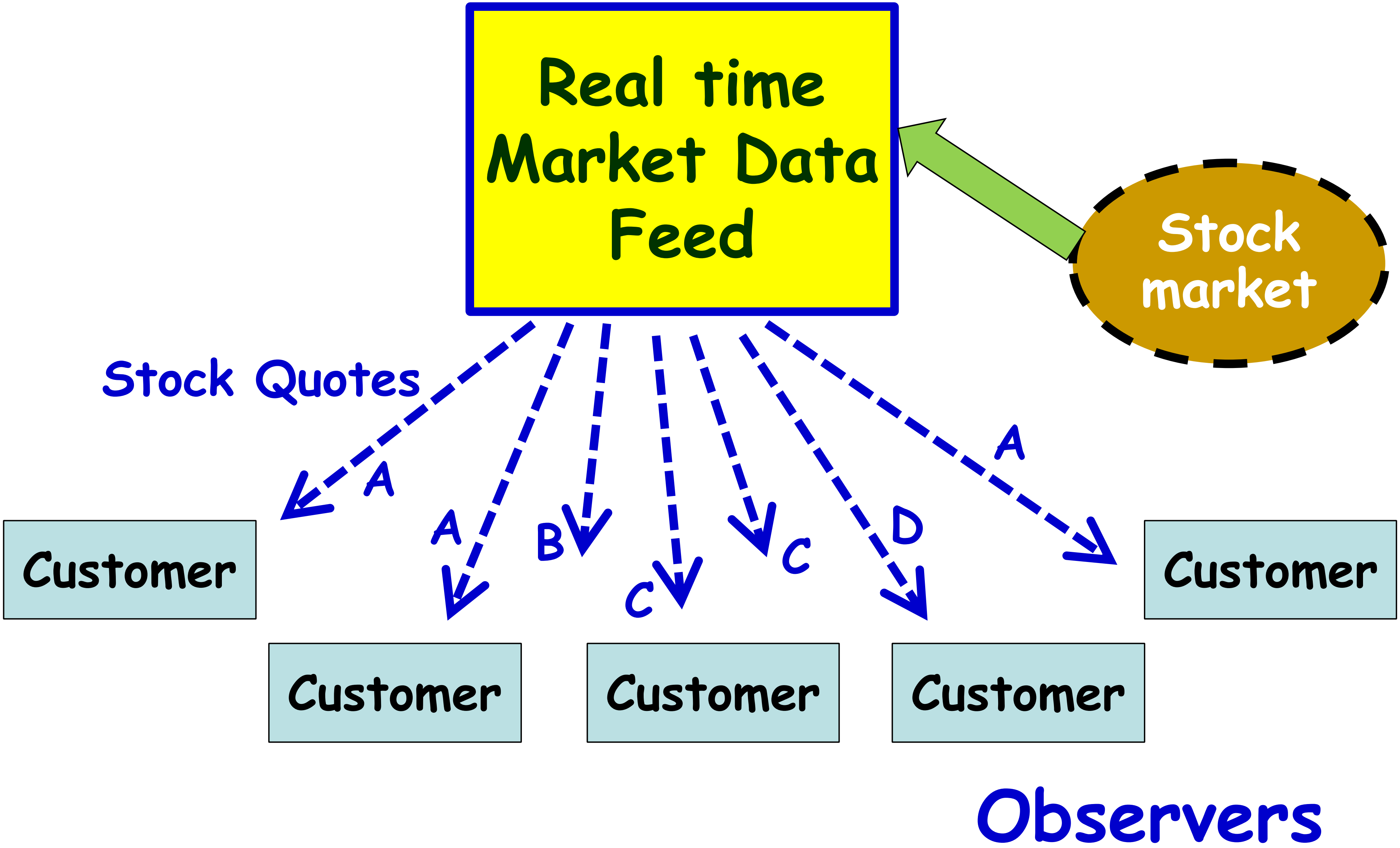


Auctioneer-Subject

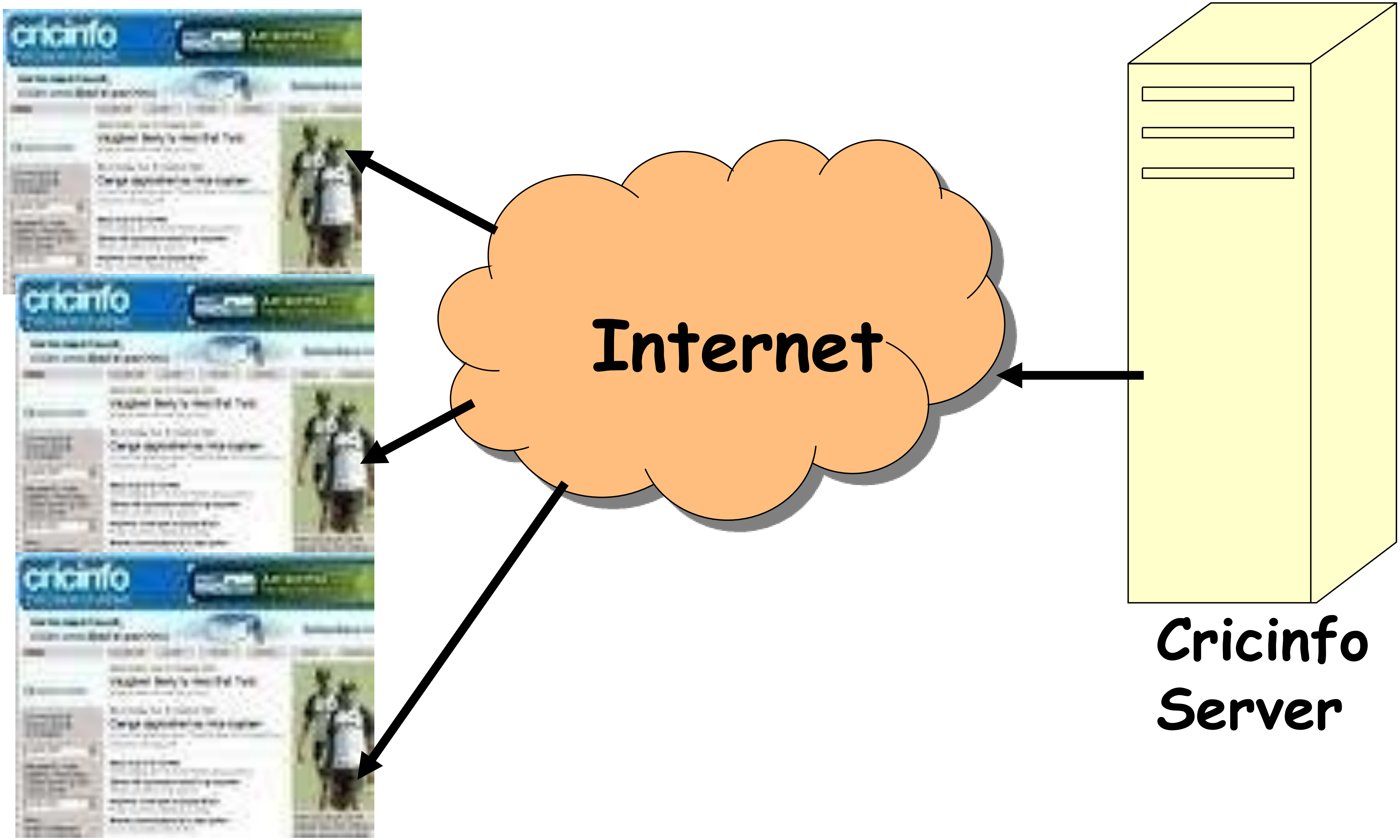
Bidders -- Observers

When present highest bid object changes its state, all its dependents are notified.

Example 1: Stock Quote Service



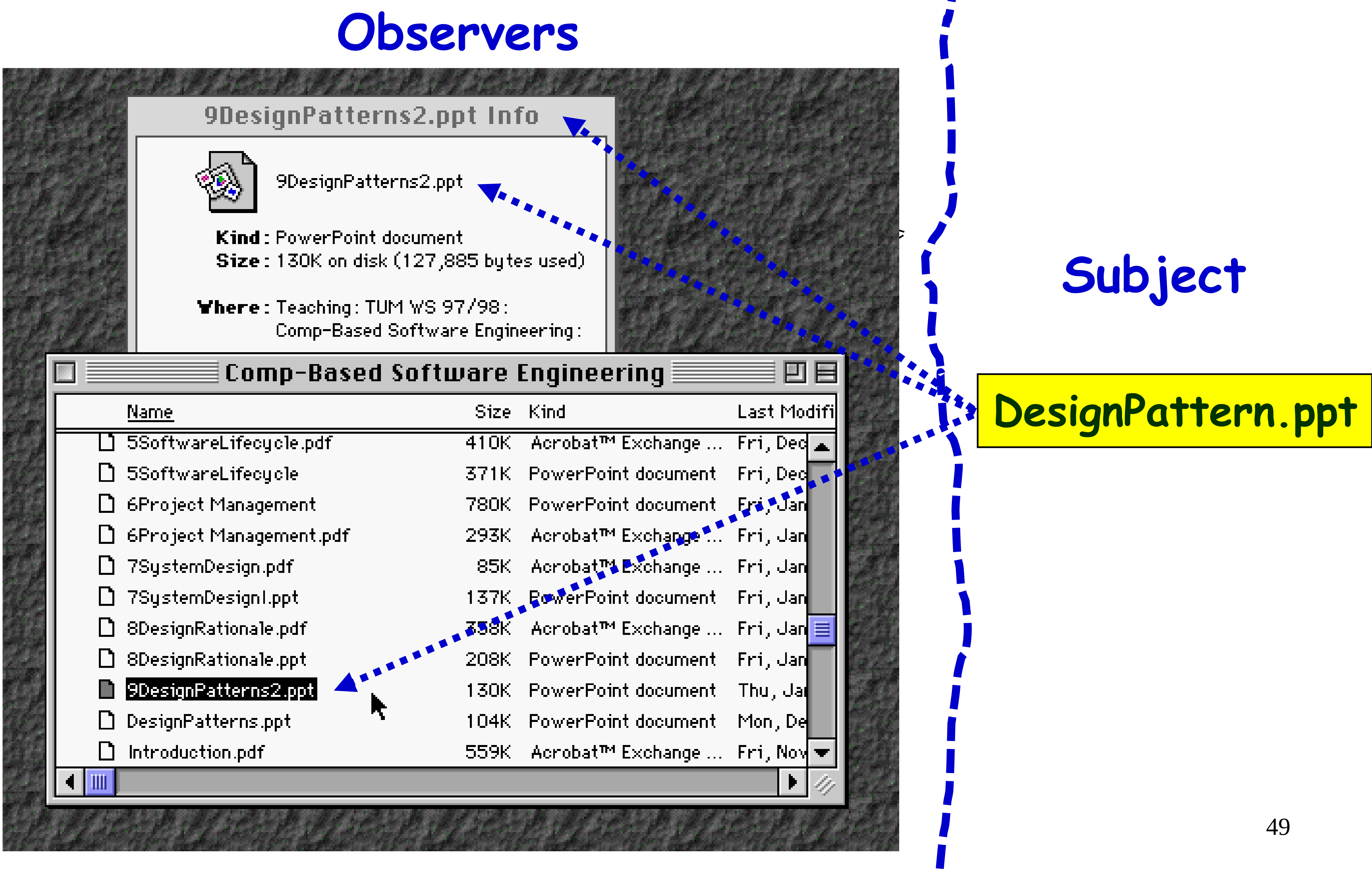
Example 2: Cricket Match



Internet

Cricinfo Server

Example 3: File Listing



9DesignPatterns2.ppt Info

9DesignPatterns2.ppt

Kind : PowerPoint document
Size : 130K on disk (127,885 bytes used)

Where : Teaching : TUM WS 97/98 :
Comp-Based Software Engineering :

Comp-Based Software Engineering

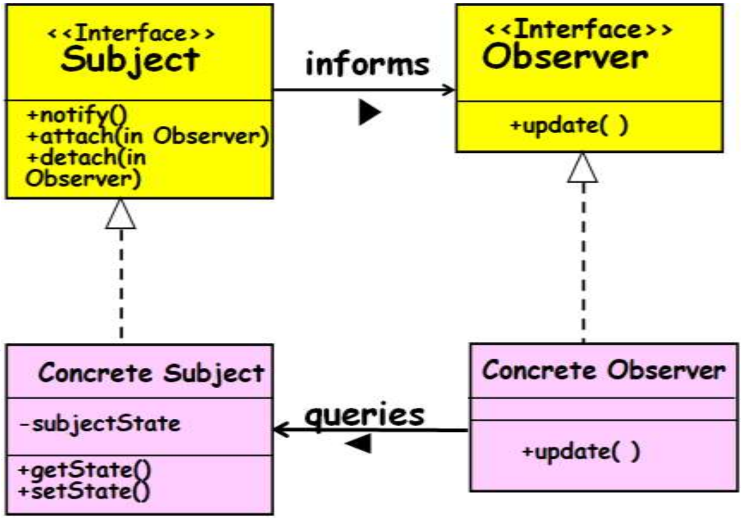
Name	Size	Kind	Last Modifi
5SoftwareLifecycle.pdf	410K	Acrobat™ Exchange ...	Fri, Dec
5SoftwareLifecycle	371K	PowerPoint document	Fri, Dec
6Project Management	780K	PowerPoint document	Fri, Jan
6Project Management.pdf	293K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesign.pdf	85K	Acrobat™ Exchange ...	Fri, Jan
7SystemDesignI.ppt	137K	PowerPoint document	Fri, Jan
8DesignRationale.pdf	358K	Acrobat™ Exchange ...	Fri, Jan
8DesignRationale.ppt	208K	PowerPoint document	Fri, Jan
9DesignPatterns2.ppt	130K	PowerPoint document	Thu, Jar
DesignPatterns.ppt	104K	PowerPoint document	Mon, De
Introduction.pdf	559K	Acrobat™ Exchange ...	Fri, Nov

Observer Pattern

- When observable (model) changes state:
  - All dependent objects are notified --- these objects then update themselves.
  - **Allows for consistency between related objects without tightly coupling the classes.**

Observer Pattern: Solution

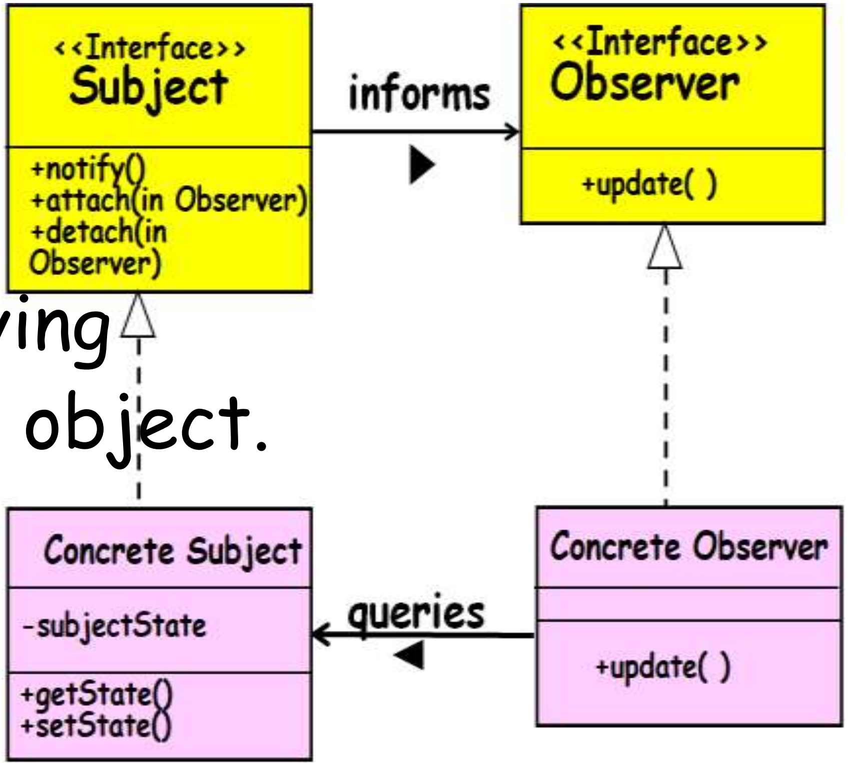
- Observers need to register themselves with the model object.
  - The model object maintains a list of the registered observers.



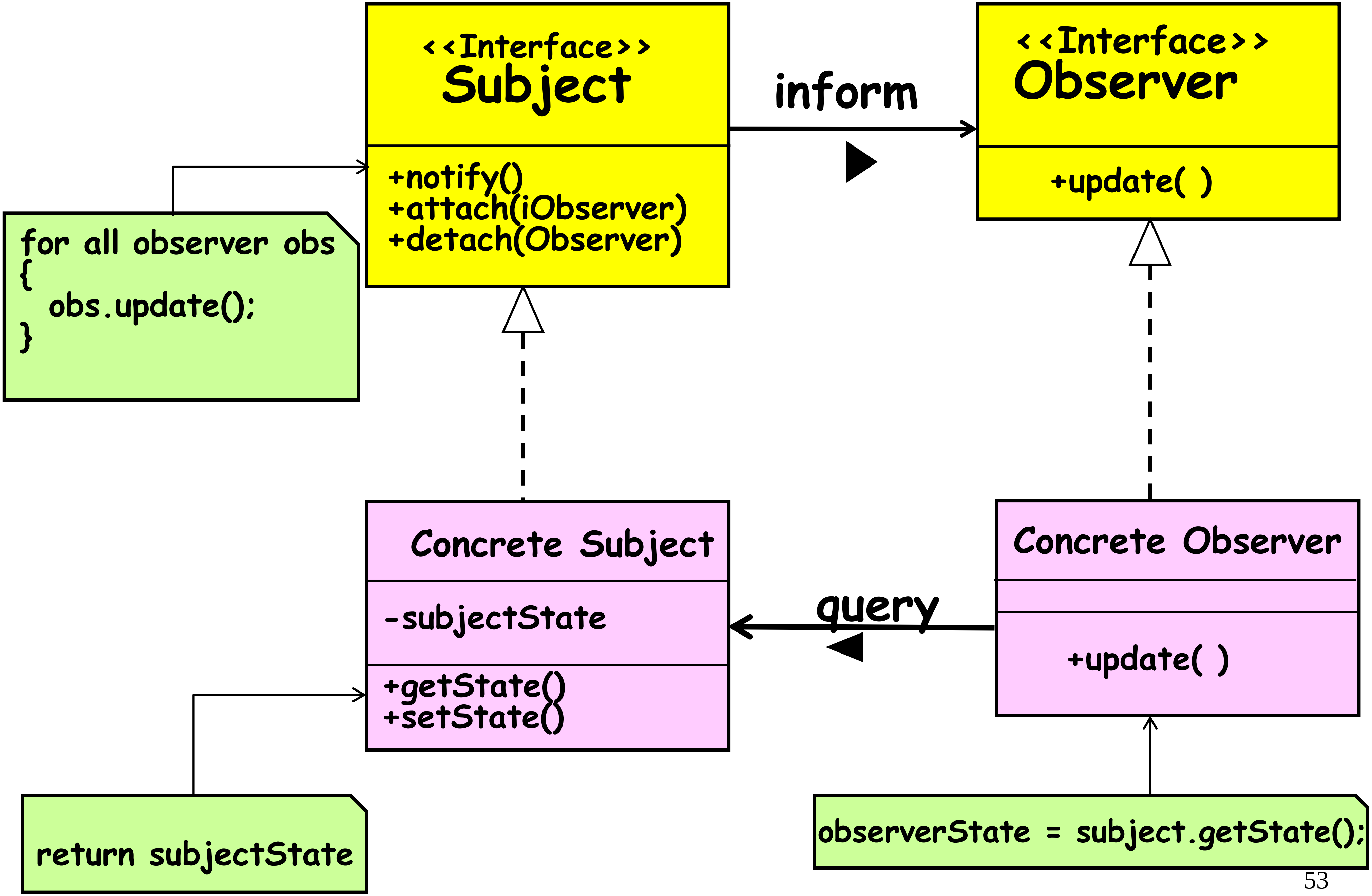
- When a change occurs to a model object:
  - Model notifies all registered observers.
  - Subsequently, each observer queries the model object to get any specific information about the changes.

Key Players

- **Subject Interface**
  - Knows its observers - provides interface for attaching/detaching subjects
- **Observer Interface**
  - Defines an interface for notifying the subjects of changes to the object.
- **ConcreteSubject**
  - Inplements subject interface to send notification to observers when state changes
- **ConcreteObserver**
  - Implements Observer interface to keep state consistent with subject



Structure of Observer Pattern



<<Interface>>
Subject

+notify()
+attach(iObserver)
+detach(Observer)

inform ▶

<<Interface>>
Observer

+update( )

```
for all observer obs
{
  obs.update();
}
```

Concrete Subject

-subjectState

+getState()
+setState()

query ◀

Concrete Observer

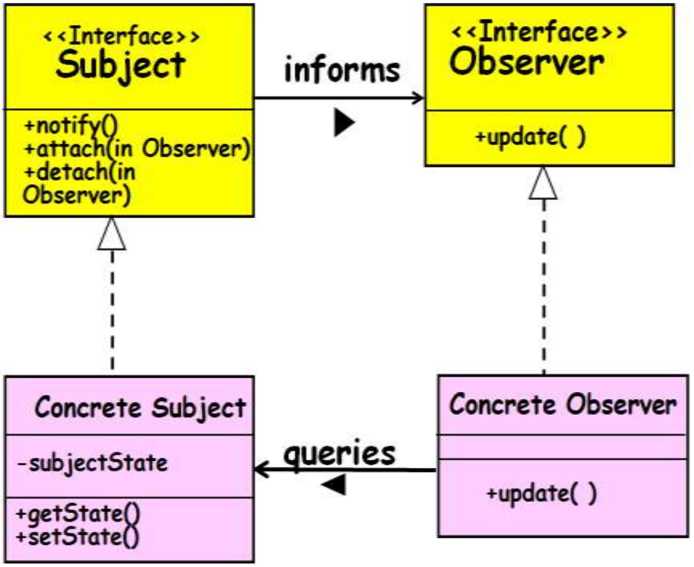
+update( )

return subjectState

observerState = subject.getState();

Observer Pattern

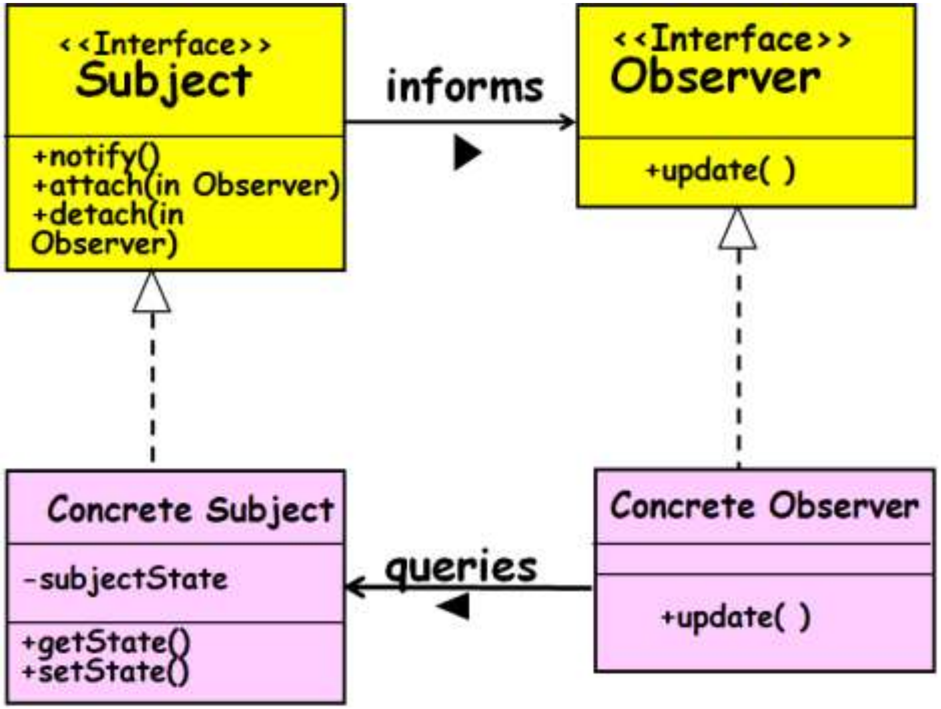
- Defines a many-to-one dependency between objects:
  - **When subject changes state, all its dependents are notified and they update themselves.**



- Effectively decouples the subject from the observer:
  - But, subject has little information about needs of the observer.
  - Can result in excessive notifications.

Working of Observer Pattern

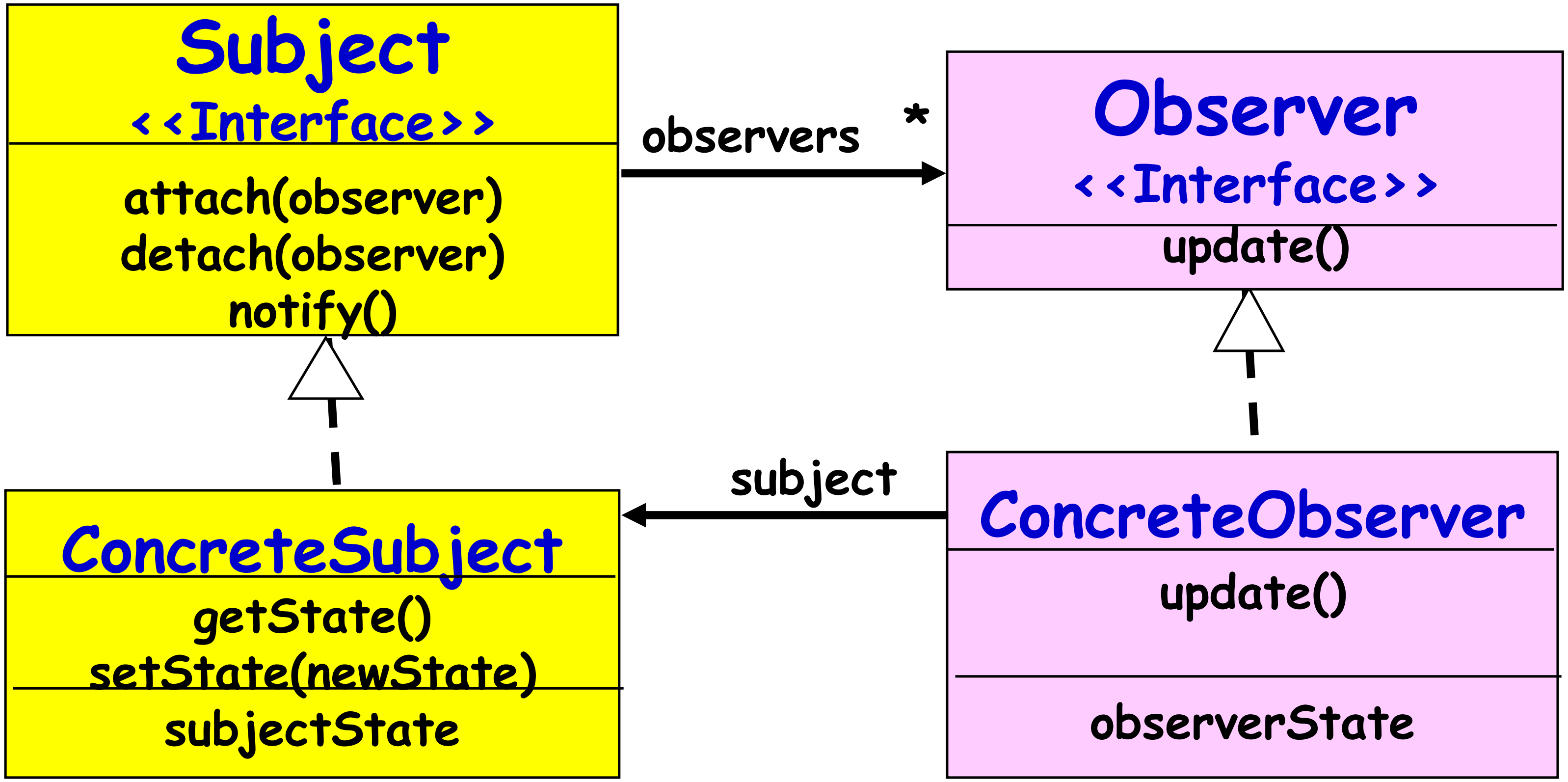
- ConcreteSubject notifies its observers:
  - Whenever a change makes state inconsistent with the observers.



- After being informed of change:

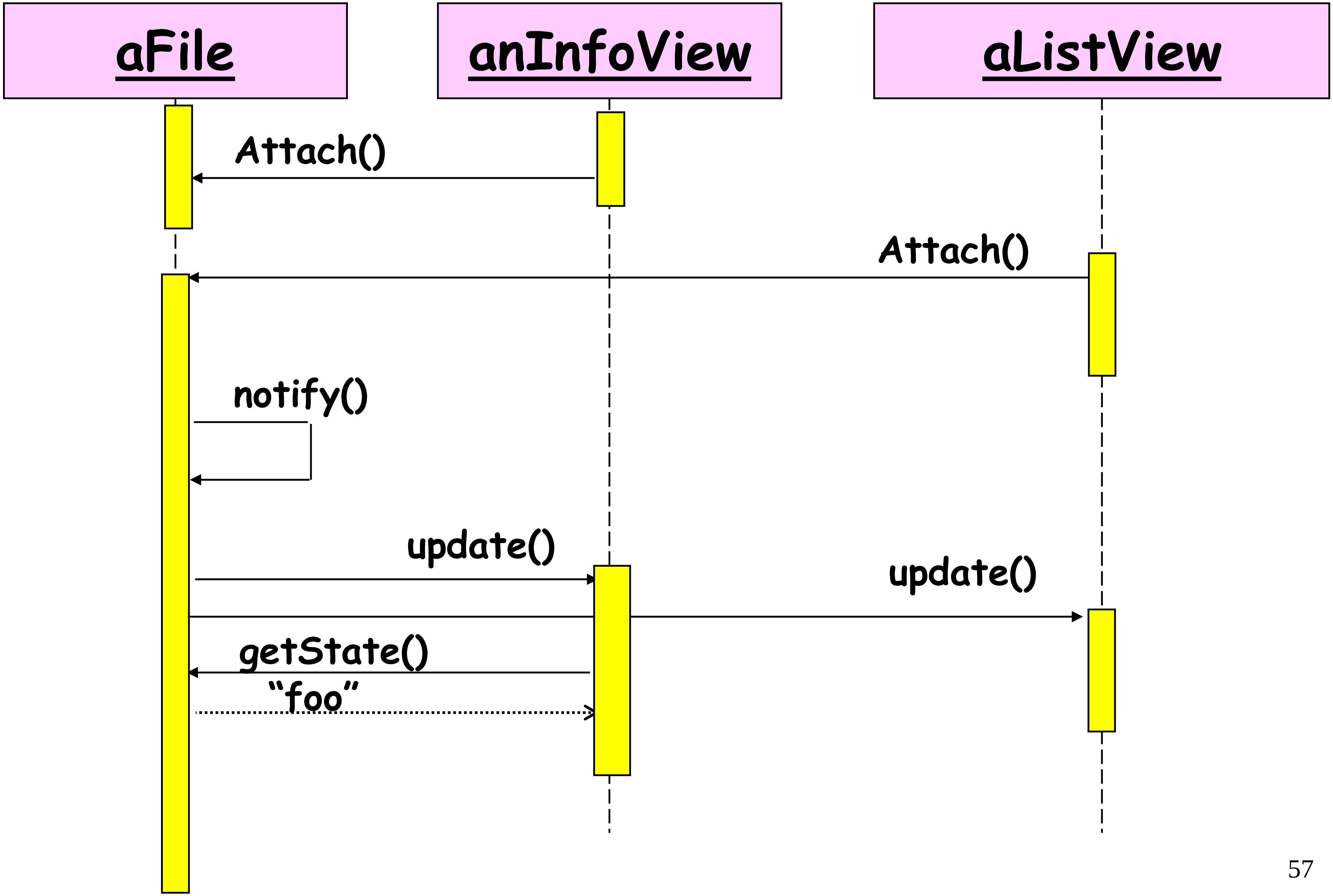
– A ConcreteObserver queries the subject to reconcile its state with subjects.

Observer Pattern



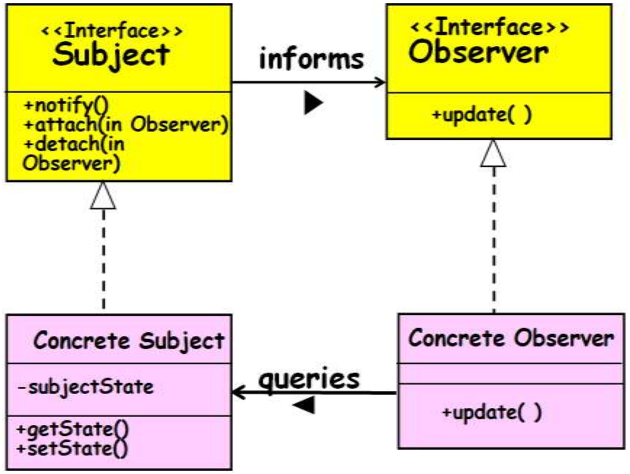
- The Subject represents the actual state,
  - The Observers represent different views of the state.

Animated Sequence diagram



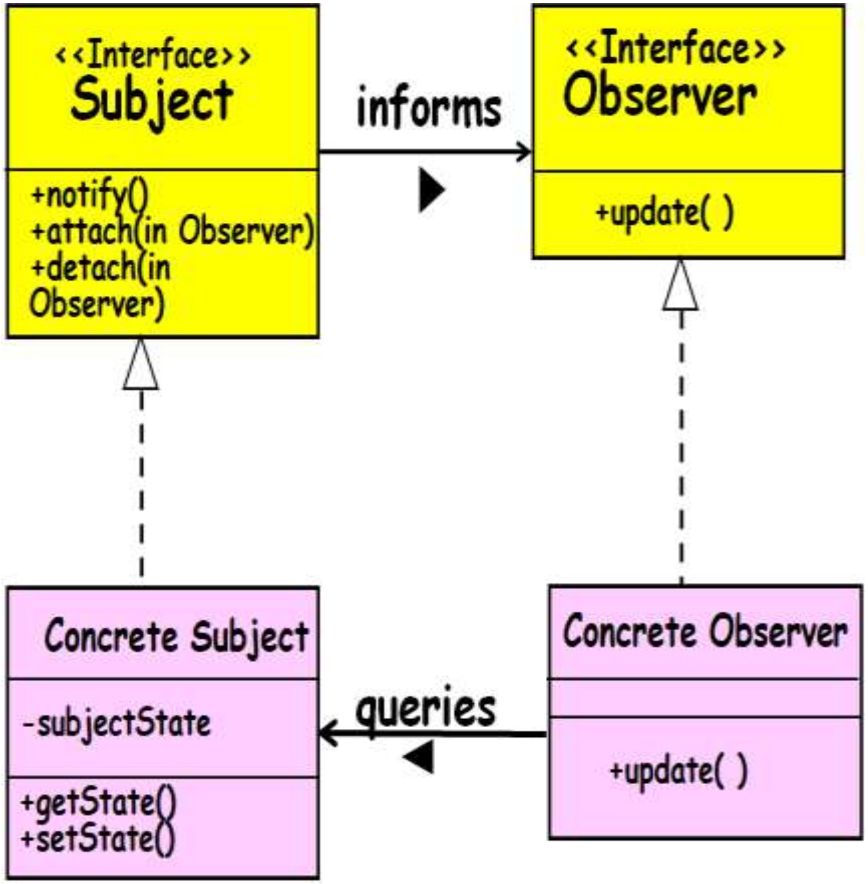
Activities of Update

- ConcreteObserver.Update():
  - Check for exceeding some threshold, etc.
  - Repaint the user interface
  - These are operations that might have cluttered up a domain class (Subject).



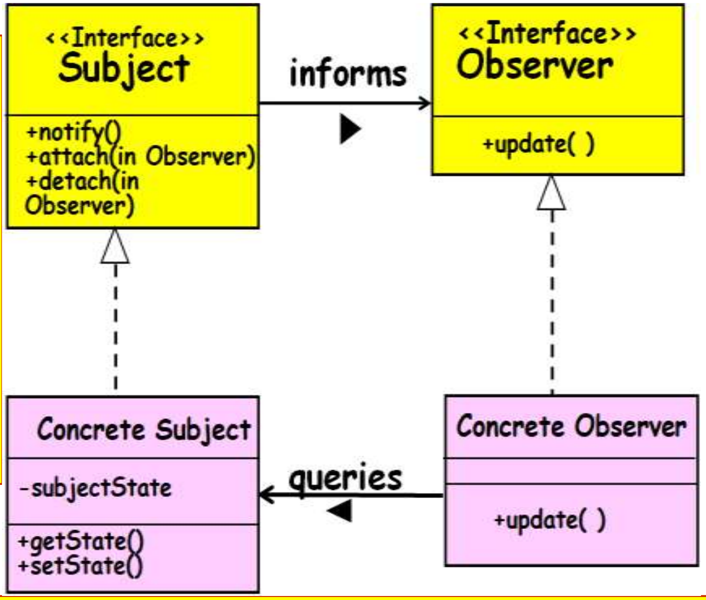
Observer Implementation

- A Subject class usually maintains:
  - An **ArrayList** of ids of registered Observers
  - Makes it easier when any random object wants to attach or detach

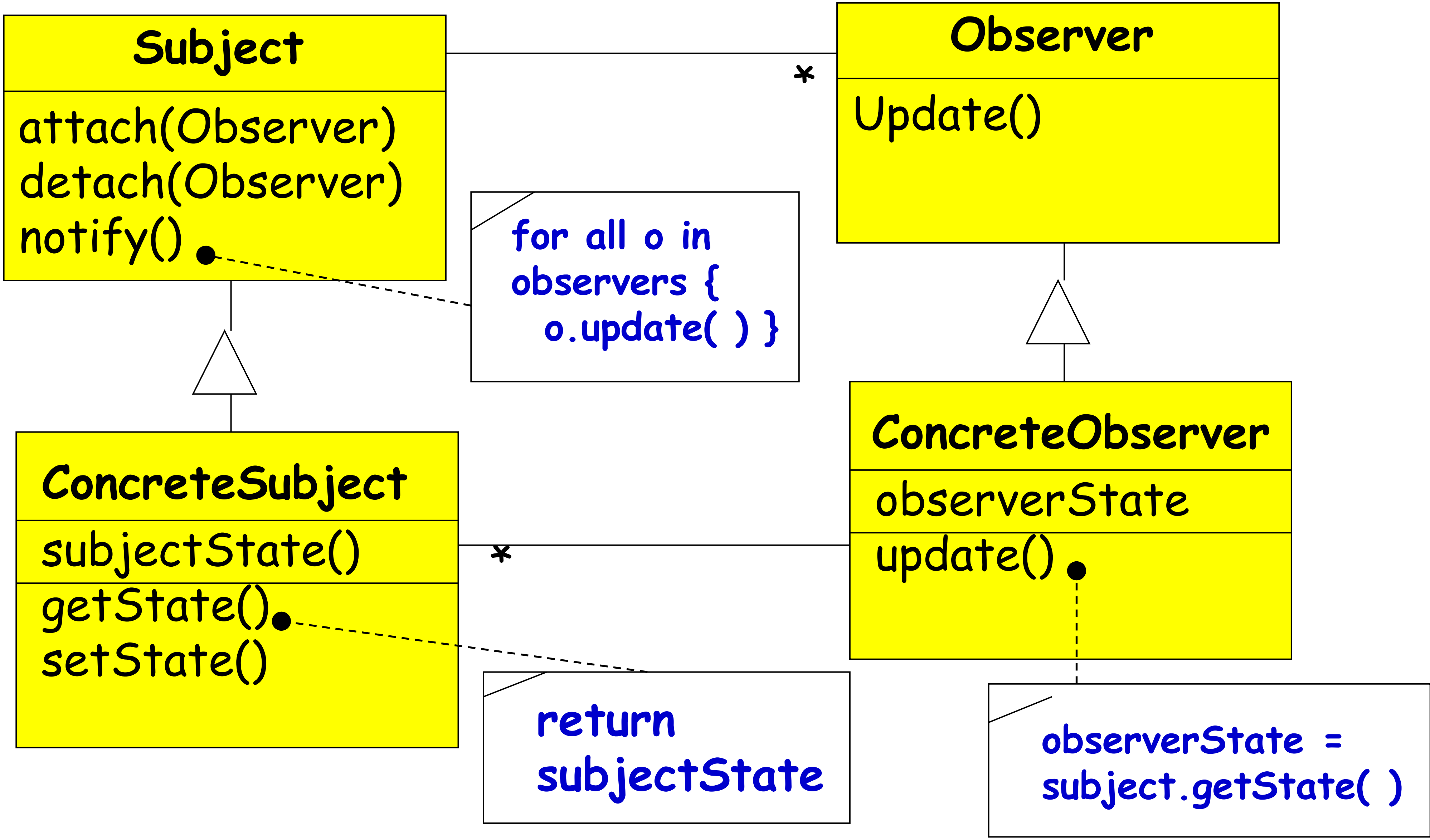


Observer Pattern Implementation in Java

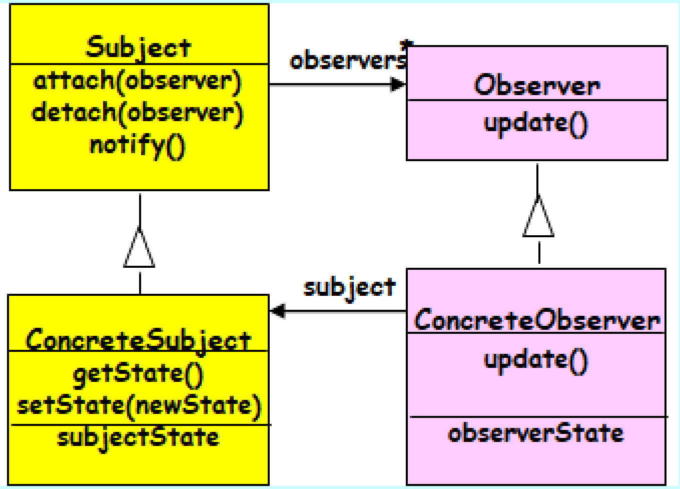
```
public interface Observer {

    public void update(Subject o );

}
```



```
Public interface Subject {
        public void addObserver(Observer o);
        public void removeObserver(Observer o);
        public String getState();
        public void setState(String state);
}
```


Subject
attach(Observer)
detach(Observer)
notify()
for all o in
observers {
o.update( ) }
*
Observer
Update()
ConcreteSubject
subjectState()
getState()
setState()
*
ConcreteObserver
observerState
update()
return
subjectState
observerState =
subject.getState( )

```
class ConcreteObserver implements Observer {
    private String state = "";
    public void update(Subject o) {
        state = o.getState();
        System.out.println("Update received from Subject, state
                                            changed to : " + state);}
    }

  class ConcreteSubject implements Subject {
        private List observers = new ArrayList();
        private String state = "";

        public String getState() {
               return state;}

        public void setState(String state) {
              this.state = state;
              notifyObservers();}
```



```
  public void addObserver(Observer o) {
      observers.add(o);}









}
```

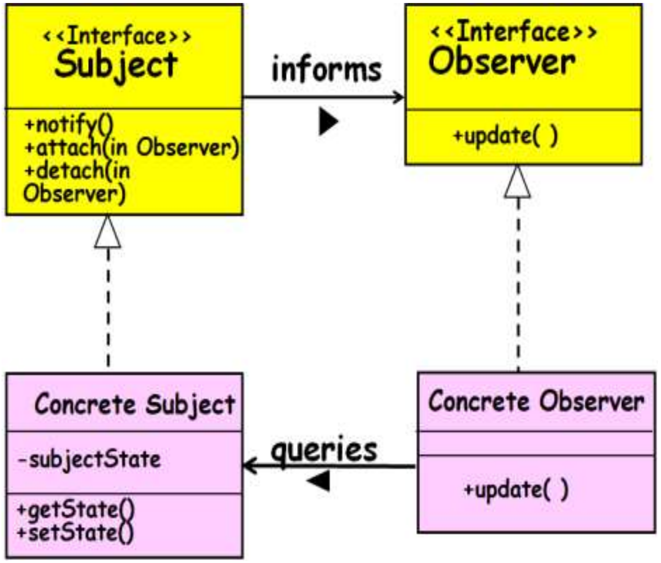
Inbuilt Java Observers

- Support for the Observer pattern is built into Java language.

```
interface Observer {
    void update(Observable o, Object arg);
} //java.util.Observer

class Observable {
    void addObserver(Observer o) { … }
    void deleteObserver(Observer o) { … }
    void notifyObservers(Object arg) { … }
} //java.util.Observable base-class
```

Java Observer/Observable

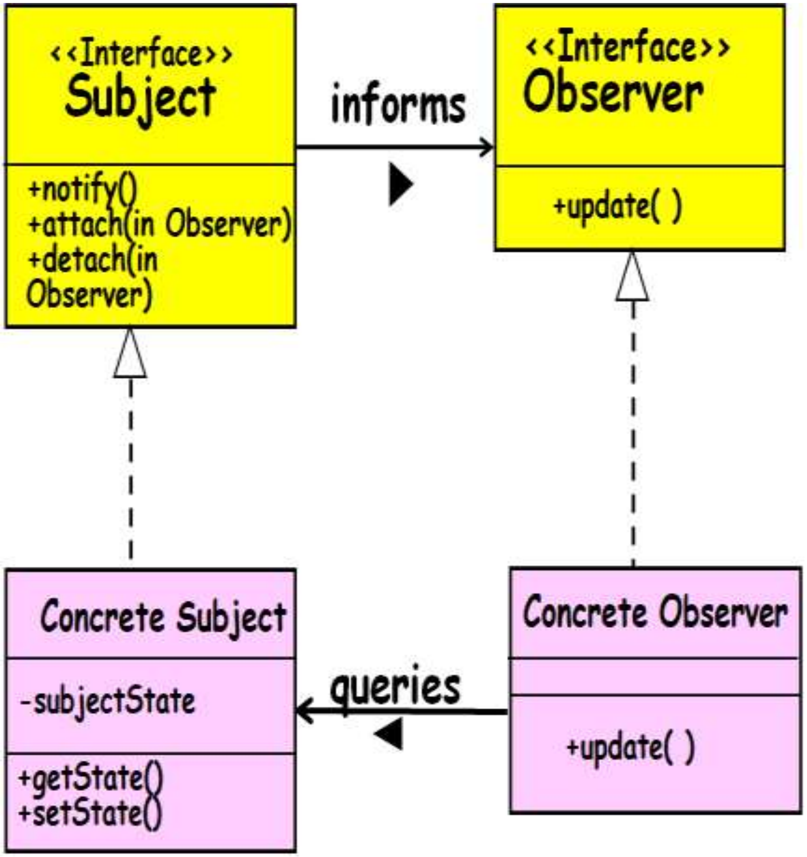
- Observers register with Observable
- Observable recognizes events and calls specified method of observers
- **Java Observer**
  - Interface class, implemented by observers
- **Java Observable**
  - Concrete Class



Java Observer Class

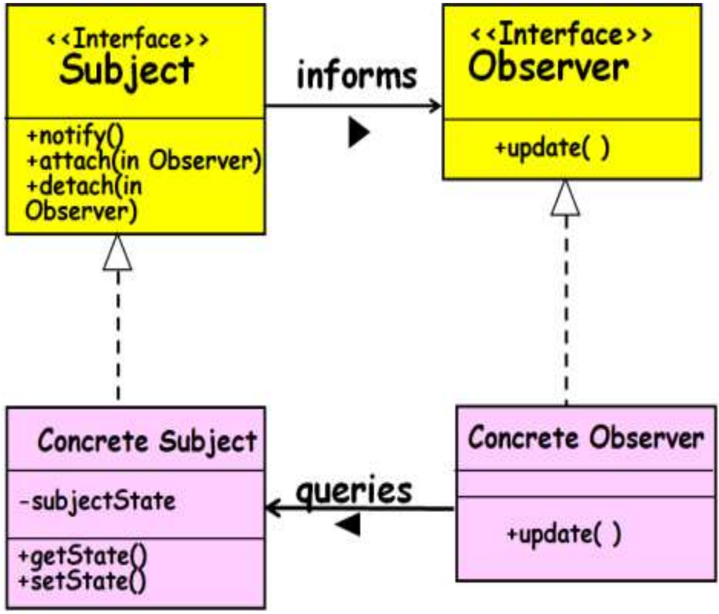
- Must implement an update() method which the Observable object x will call:
  - **update(Observable x, Object y)**

- When Observable object x calls the update() method for a registered observer:
  - It passes itself as a parameter and also a data object y…



Java Observable Class

- Needs to be extended
- Inherited methods from Observable
  - **addObserver(Observer z)**
    - Adds the object z to the observable's list of observers
  - **notifyObserver(Object y)**
    - Calls the update() method for each of the observers in its list of observers, passing y as the data object



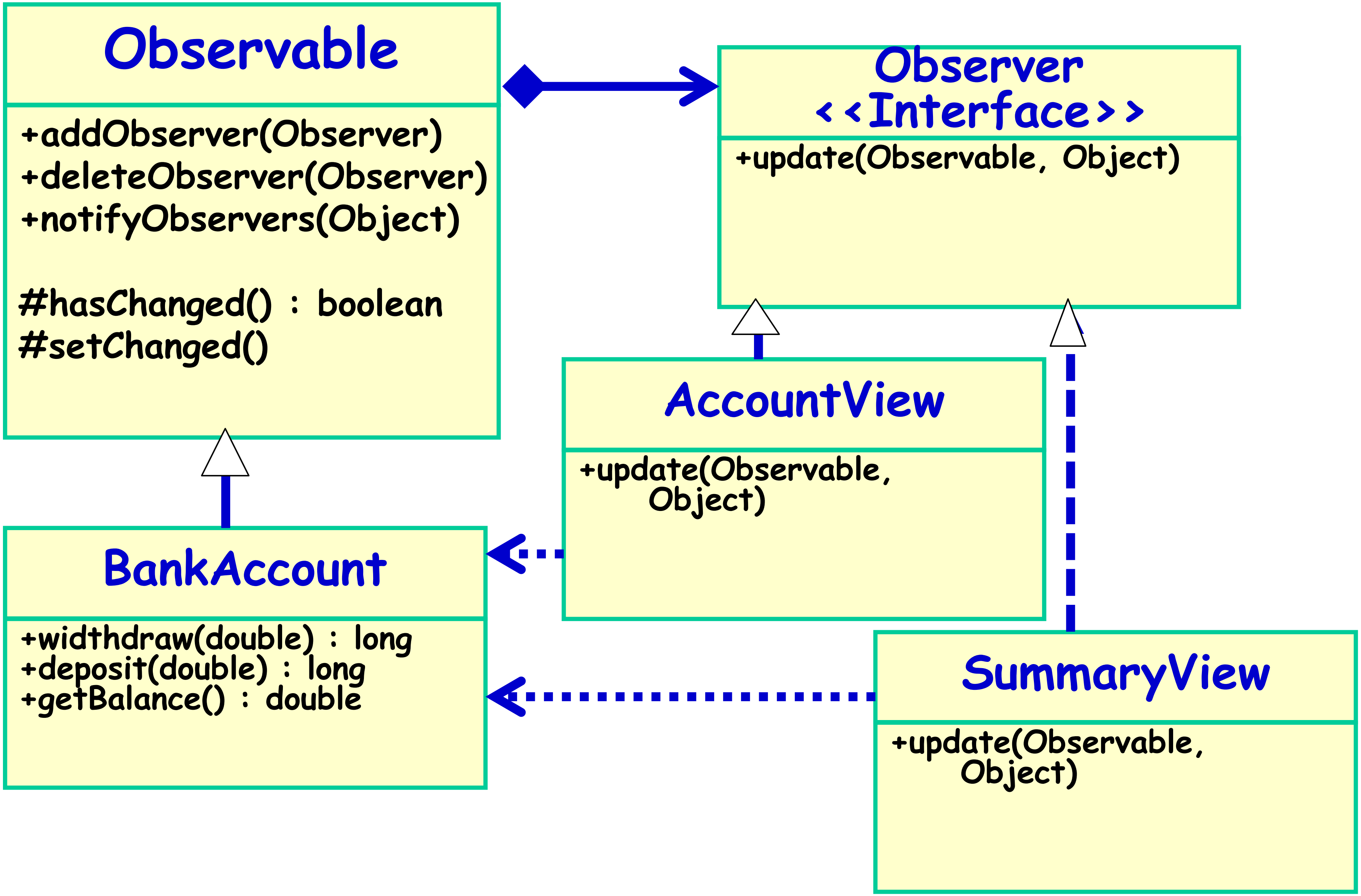
Java Support for Observer Pattern

```
interface Observer {
        void update (Observable sub, Object arg)
 }
```

Subject.

```
class Observable {
          public void addObserver(Observer o) {}
          public void deleteObserver (Observer o)   {}
          public void notifyObservers(Object arg) {}
         public boolean hasChanged() {}
 …
}
```

Bank Example: Observer Class Diagram

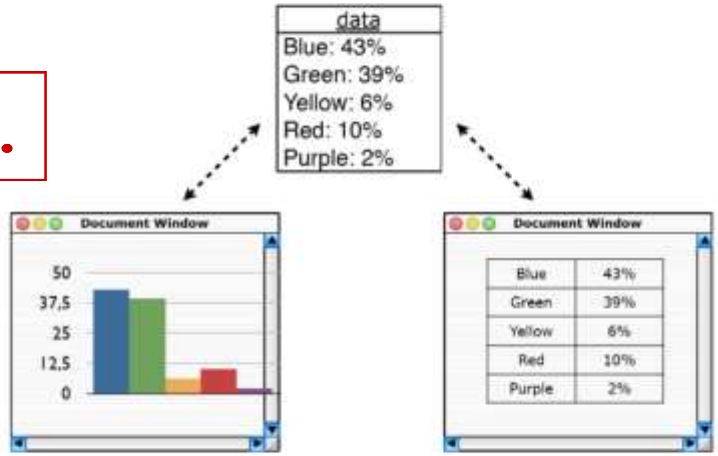


Observer Pattern- Implementation Example 2

```
public  PiChartView implements Observer {
        void update(Observable sub, Object arg) {
           // repaint the pi-chart
}       }
```

A Concrete Observer.

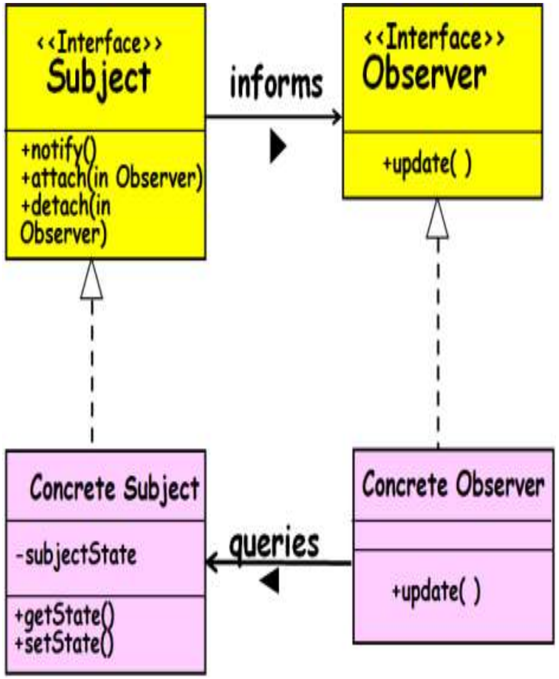
```
class StatsTable extends Subject{
     public boolean hasChanged() {
          // override to decide when it is considered changed
     }
}
```



Blue	43%
Green	39%
Yellow	6%
Red	10%
Purple	2%

Observer Pattern - Consequences

- Effectively removes coupling between subject and observer:
  - subject need not know concrete observers
- Support for broadcast communication:
  - All observers are notified
- **Supports asynchronous updates:**
  - Observers need not know when updates occur



Observer Pattern - Consequences

- Consequences
  - + **Modularity:** subject and observers may vary independently
  - + **Extensibility:** can define and add any number of observers
  - + **Customizability:** different observers provide different views of subject
- Implementation Issues
  - – Subject-observer mapping
  - – Dangling references

Some Limitations of Observer Pattern

- **Model objects incur substantial overhead:**
  - To handle registration of the observers,
  - To respond to queries from observers.
- **Performance concerns:**
  - Especially when the number of observers is large…

Observer Pattern

- **Indications:**
  - Asynchronous update of model elements
  - Observer and model states need to be consistent all the time
  - Number of observers can change dynamically

Observer

- **Contra-Indications:**
  - Static data
  - Fixed viewers

Java Observer Deprecated! Why?

- **It does not provide a rich enough event model for applications!**
  - For example, supports only the notion that something has changed, but does not convey any information about what has changed
- **Not threadsafe!**
- Programmers often make use of **PropertyChangeListener** instead:
  - Which itself is an implementation of Observer pattern…

```
public class PCLBlog {
    private String blogPostTitle;
    private PropertyChangeSupport support;
    public PCLBlog() {
        support = new PropertyChangeSupport(this);
    }
    public void addPropertyChangeListener(PropertyChangeListener pcl) {
        support.addPropertyChangeListener(pcl);
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl) {
        support.removePropertyChangeListener(pcl);
    }
    public void setBlogPostTitle(String value) {
        support.firePropertyChange("blogPostTitle", this.blogPostTitle, value);
        this.blogPostTitle = value;
    }
}
```

Can Skip!

Home Work

- **Provide class design for:**
- **Blog application:** Any one can enrol by providing e-mail. Any blog posted is communicated to the registered users.
- **Cricket application:** Any one can enrol by providing e-mail. Any development (runs, out, over etc) posted is communicated to the registered users.

Hint

```
public class Blog extends Observable {
    private List<Observer> subscribers = new ArrayList<>();
    public void addBlogPost(String blogPostTitle) {
        notifyObservers(blogPostTitle);
    }
    public void notifyObservers(String blogPostTitle) {
        for (Observer subscriber : subscribers) {
            subscriber.update(this, blogPostTitle);
        }
    }
    public void registerObserver(Observer newSubscriber) {
        this.subscribers.add(newSubscriber);
    }
    public void removeObserver(Observer previousSubscriber) {
        this.subscribers.remove(previousSubscriber);
    }
}
```

MVC: History

- As compared to the Observer pattern:
  - More appropriate in certain situations.
- First developed by Xerox PARC for Smalltalk-80.
- Used by the Application Kit system in NeXTstep.
  - Used by the Cocoa APIs for Apple's OS X.
- It is the recommended structural framework for developing J2EE applications.

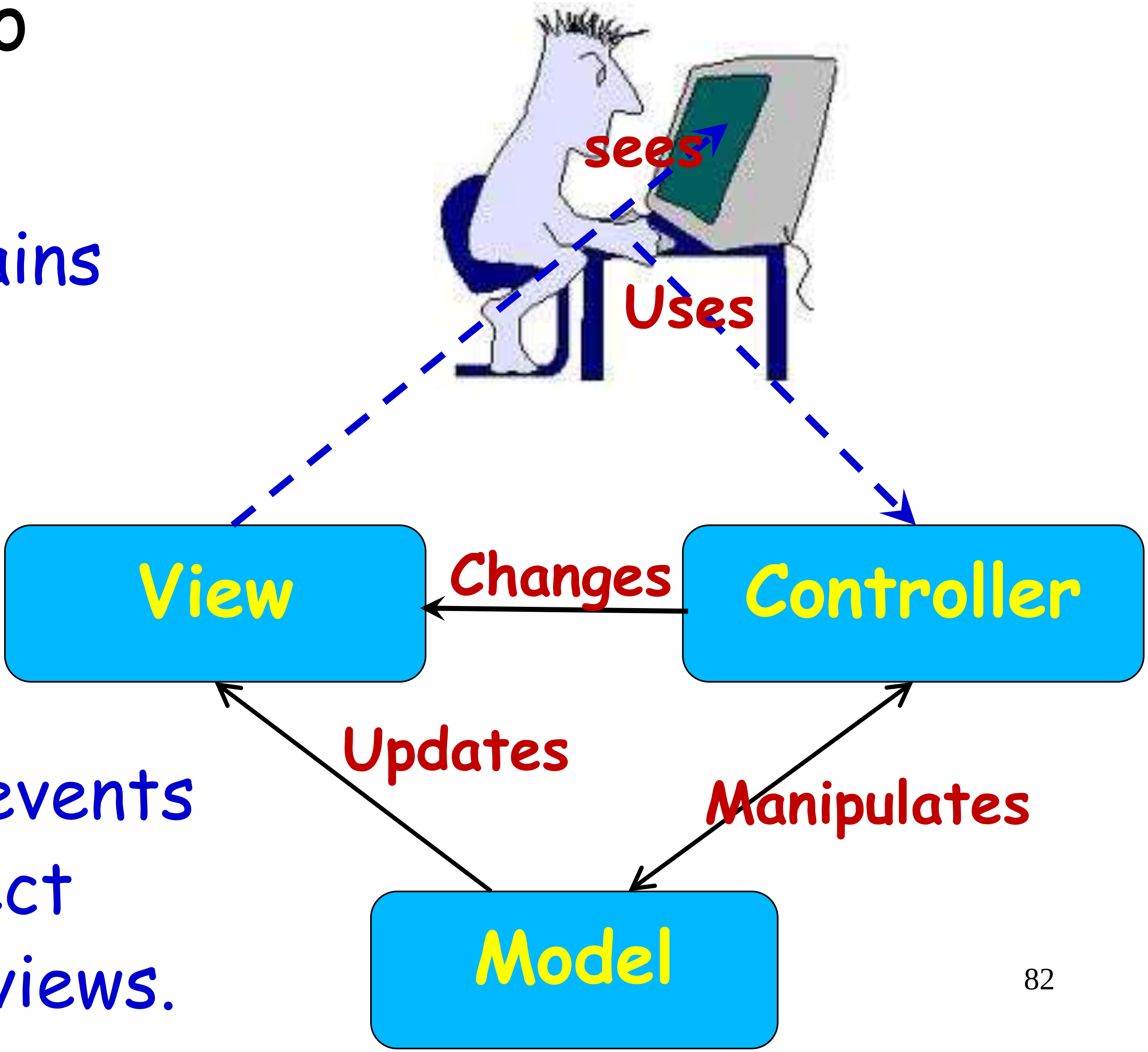
Subsequently…

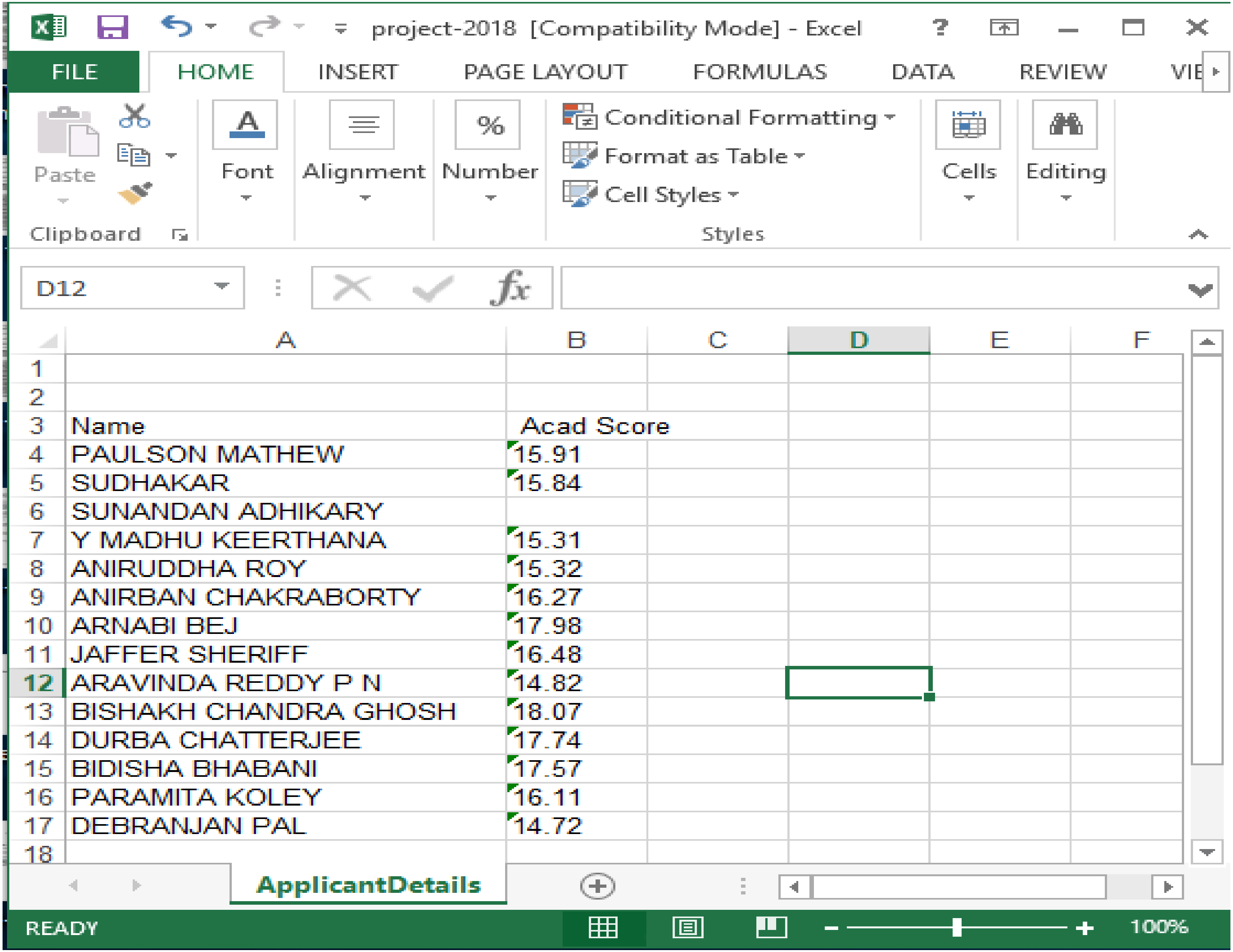
- A heavily used architectural design pattern…

- The main theme of:
  - **Struts**
  - **Ruby on Rails**
  - **Java Server Faces (JSF)**

Model-View-Controller (MVC) Pattern

- Breaks an application into three parts:
  - **Model** maintains data (State),
  - **View** displays the data.
  - **Controller** handles user events that may affect the model or views.

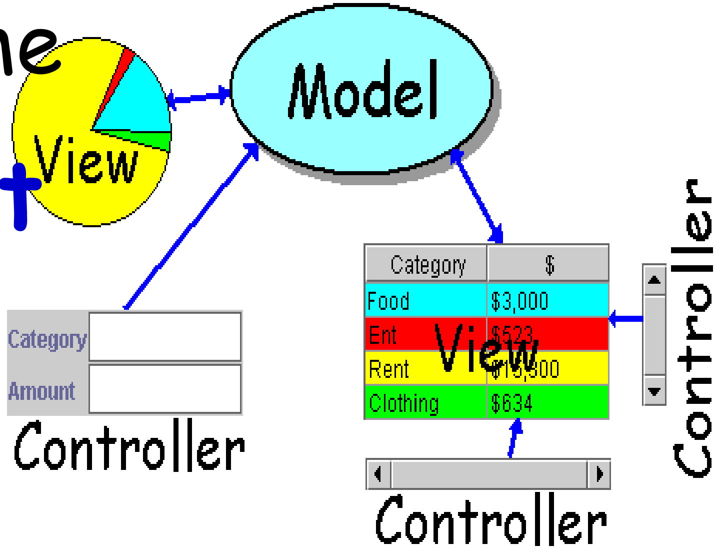




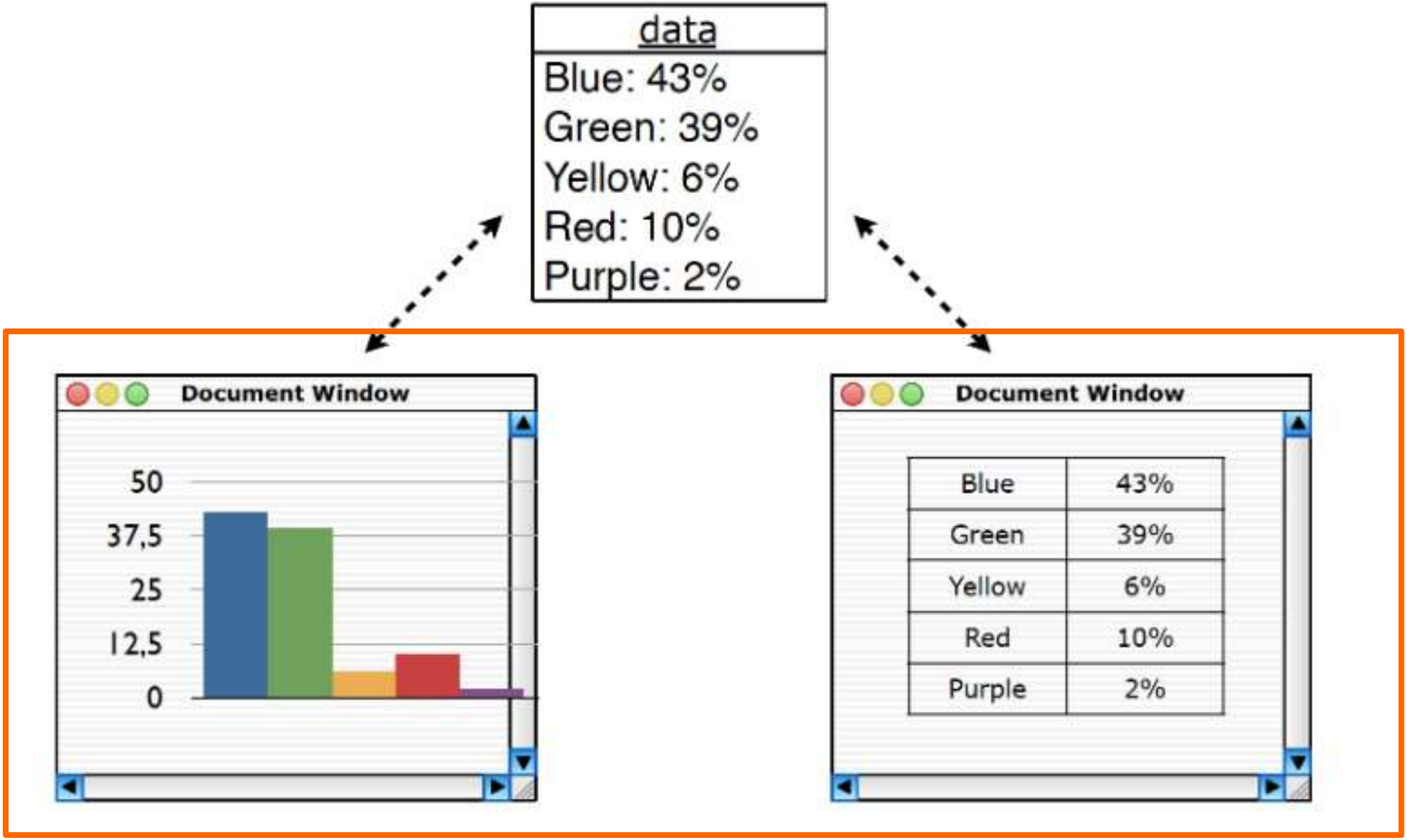
	A	B	C	D	E	F
1						
2						
3	Name	Acad Score				
4	PAULSON MATHEW	15.91				
5	SUDHAKAR	15.84				
6	SUNANDAN ADHIKARY					
7	Y MADHU KEERTHANA	15.31				
8	ANIRUDDHA ROY	15.32				
9	ANIRBAN CHAKRABORTY	16.27				
10	ARNABI BEJ	17.98				
11	JAFFER SHERIFF	16.48				
12	ARAVINDA REDDY P N	14.82				
13	BISHAKH CHANDRA GHOSH	18.07				
14	DURBA CHATTERJEE	17.74				
15	BIDISHA BHABANI	17.57				
16	PARAMITA KOLEY	16.11				
17	DEBRANJAN PAL	14.72				
18						

MVC Architecture

- A typical application includes:
  - **Model: Maintain application data,**
    - Data in a spreadsheet, or
    - state of a chess game -- positions of pieces
  - **View: Present output**
    - Data in table or pie chart or bar chart
    - Graphical view of chess game
  - **Control: Handle user input**
    - key presses
    - mouse click on controls



Dependency Among MVC Elements



data

Blue: 43%

Green: 39%

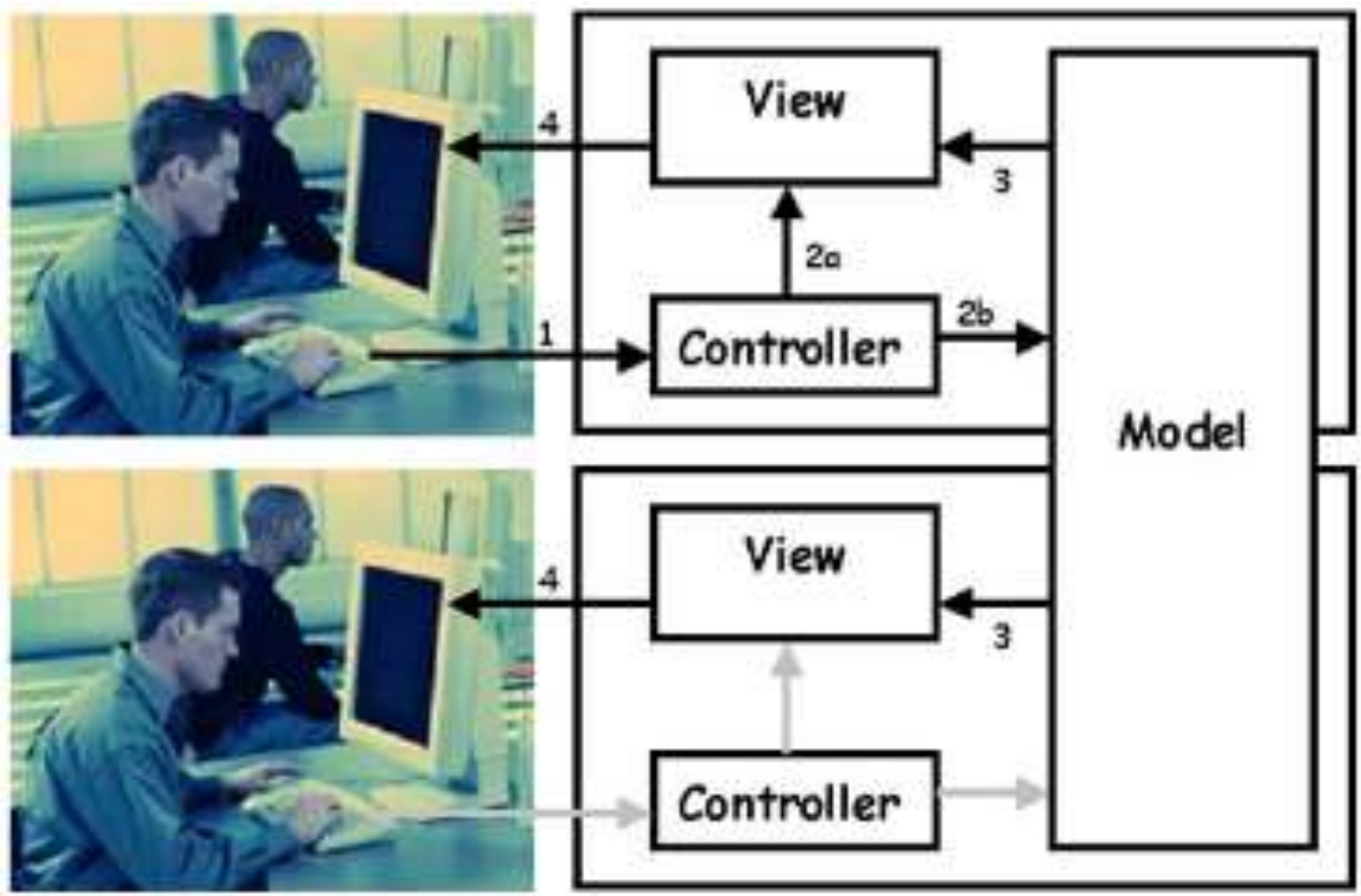
Yellow: 6%

Red: 10%

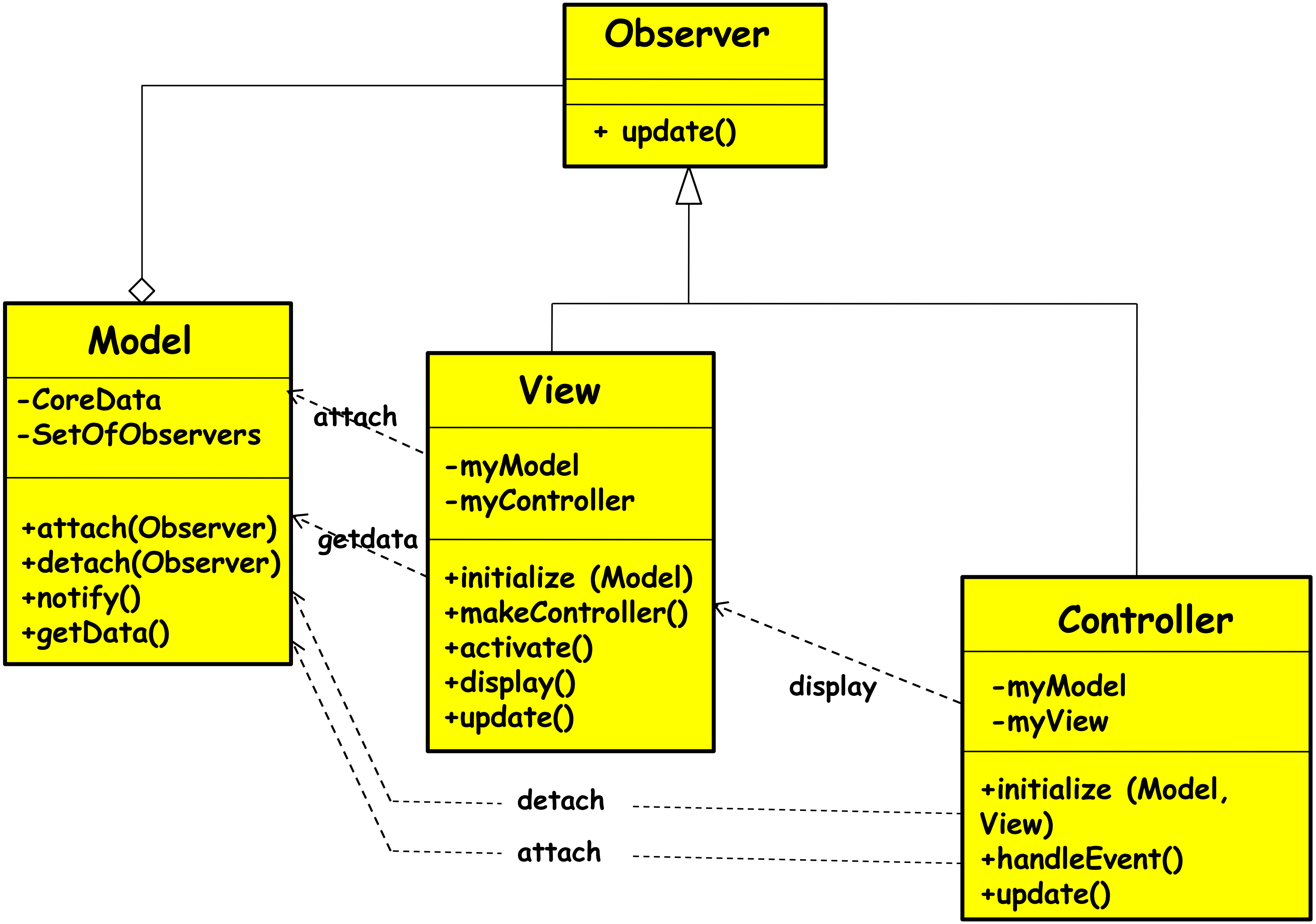
Purple: 2%

Blue	43%
Green	39%
Yellow	6%
Red	10%
Purple	2%

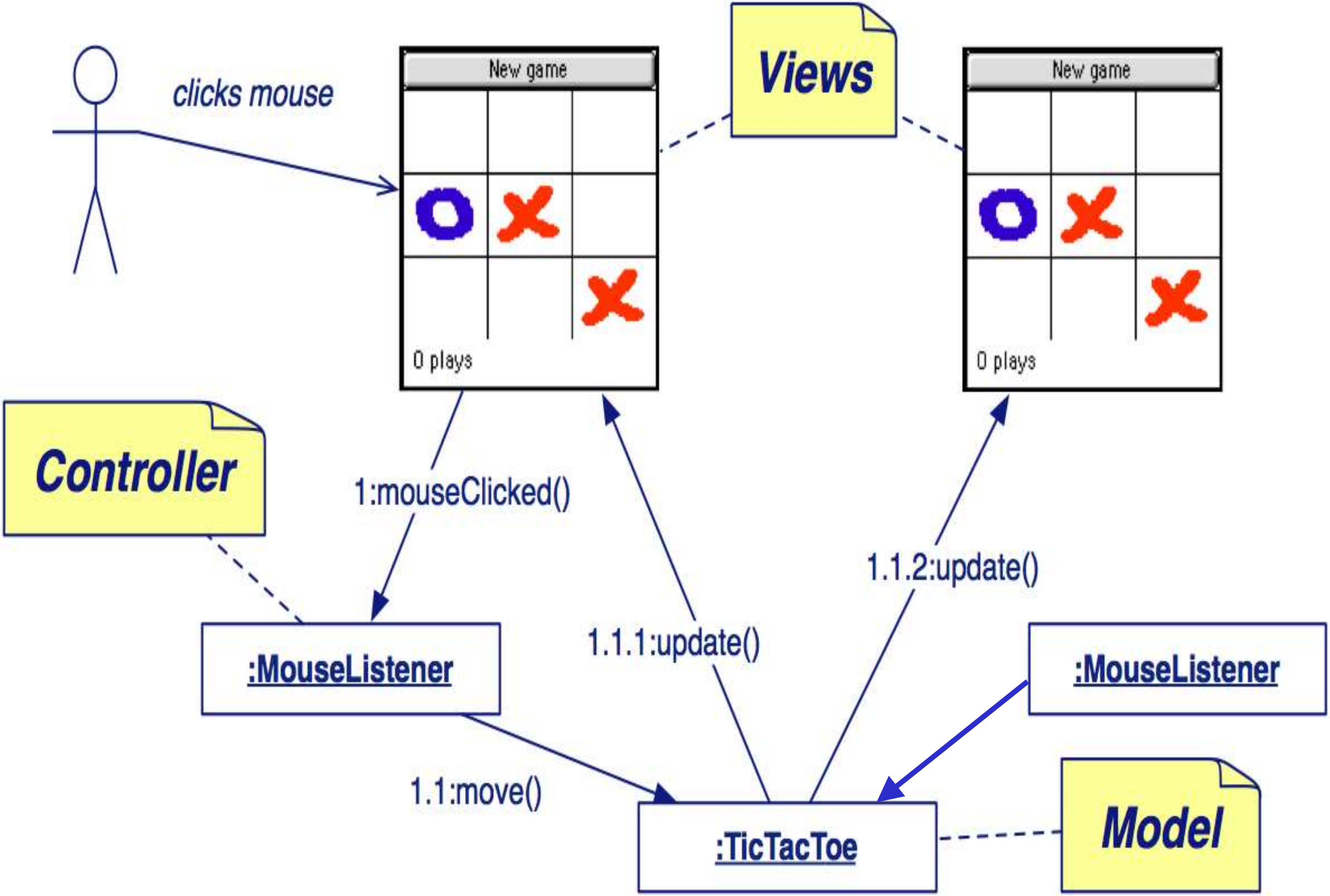
Multiple Views and Controls



MVC incorporates Observer Pattern!



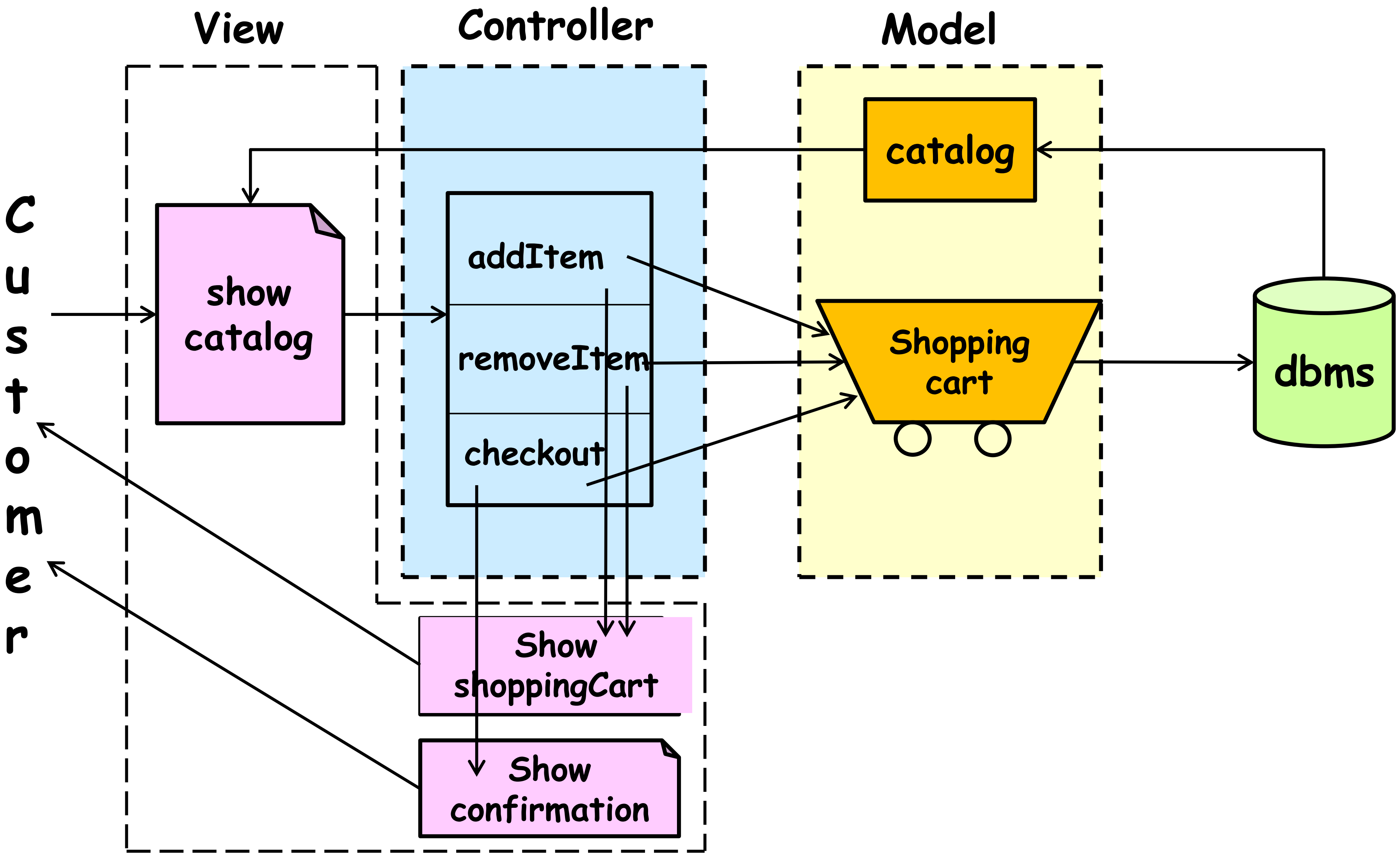
Model-View-Controller: Example

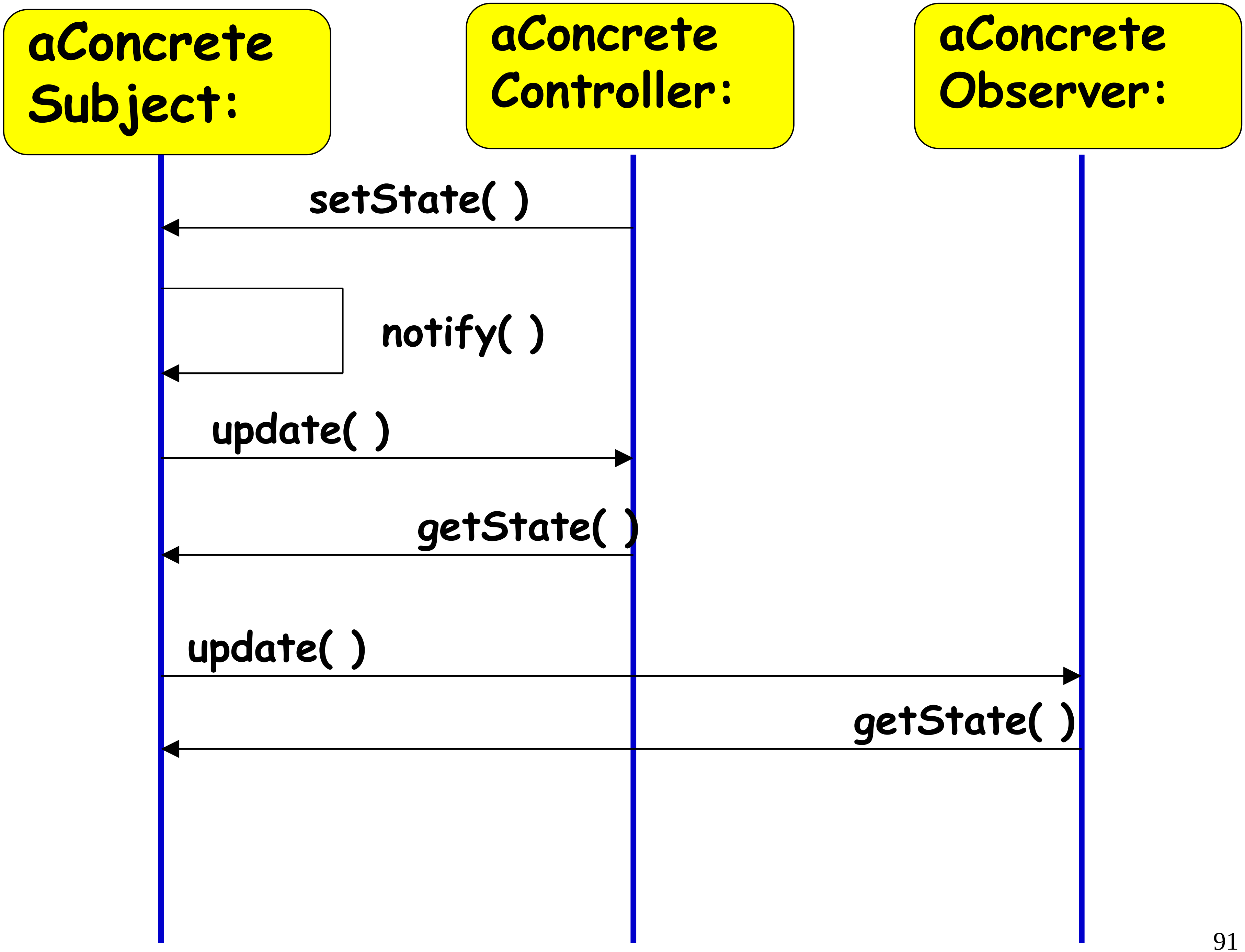


MVC in Web Application Development

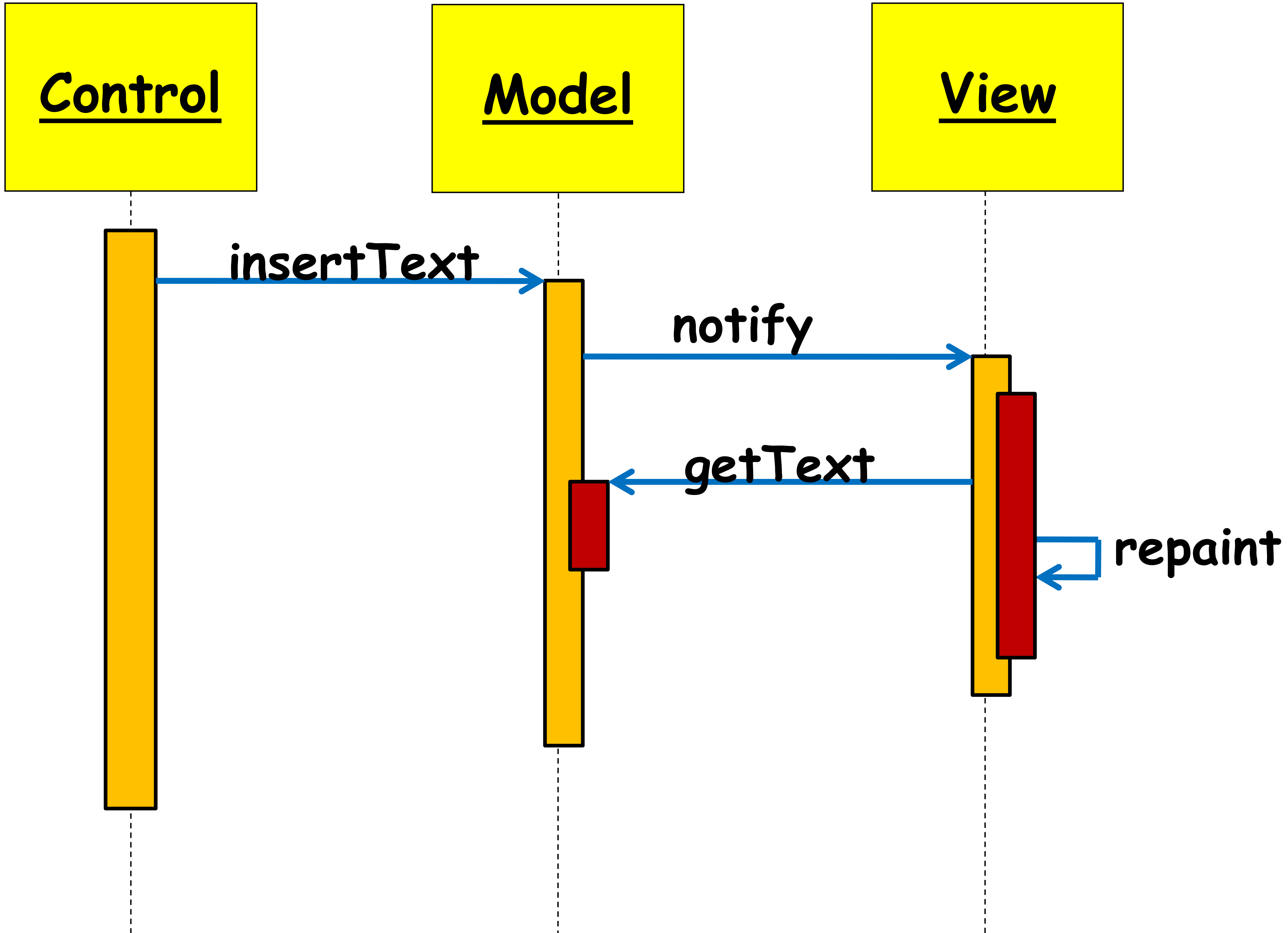
1. **Model**: handles business logic
   - Data retrieval & manipulation
   - Including checking "Business Rules"
2. **View**: user interface (UI) component
   - Front-end generation: html in browser
   - **The *LOOK* of the application**
3. **Control**: deals creating with UI events
   - Triggered by user using the UI
   - **The *FEEL* of the application**

MVC: Another Example




aConcrete Subject:
aConcrete Controller:
aConcrete Observer:
setState( )
notify( )
update( )
getState( )
update( )
getState( )

MVC Example: Word Processor

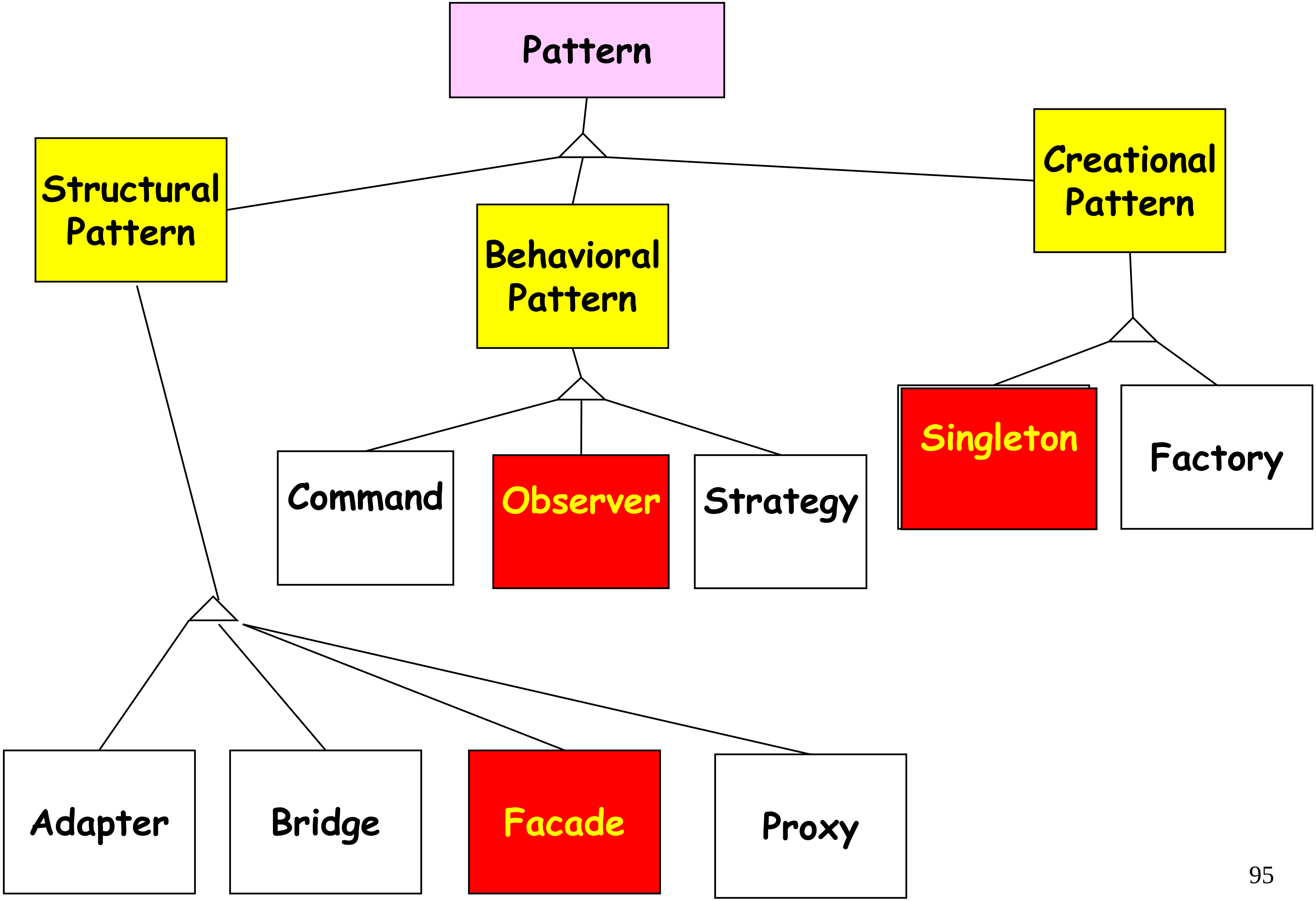


MVC Pattern in Java Swing

Swing uses the model-view-controller architecture (**MVC**) as the fundamental design behind each of its components. Essentially, **MVC** breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves. The model encompasses the state data for each component. **Robert Eckstein**

Singleton Pattern

Recollect: Pattern Taxonomy



Creational Patterns

- Abstracts out the instantiation process.
- Make system independent of creation of new objects:
  - Objects created
  - Possibly composed
- Encapsulate knowledge about which concrete classes the system uses:
  - Hide how instances of these classes are created and put together

Singleton Pattern: Problem

- Ensure that a class has only a single instance.

- Provide a global point of access to it.

Singleton Pattern: Example 1

- **Problem: A single object should maintain an application's configuration.**
- Many objects may wish to read and update the configuration:
  - **How to restrict only one instantiation?**
  - **A singleton class will ensure that all of the application's objects can get the same copy of the game's configuration…**

Singleton Pattern: Example 2

- **Problem:**
  - How to control access to resources such as database connections or sockets?

- **Example:** You have a license for only one connection for your database:
  - ***A Singleton makes sure that only one connection is made.***
  - If you buy more licenses or use a JDBC driver that allows multithreading, the Singleton can be easily adjusted to allow more connections (Multiton).

Singleton: Example 3

- Suppose you need a counter that gives out unique numbers (e.g. token numbers in a restaurant):
  - First, the counter object needs to be unique.
  - A Singleton counter can generate the numbers and synchronize access.

More Singleton Examples

- One session manager per user session
- One file system
- One shopping basket per customer
- One logger
- One configuration object
- One account per user,
- Abstract factory and factory method, etc.

The Singleton Pattern

- A Singleton is accessed by many objects in an application:

  - **Therefore should be easily and globally accessible --- Provide a global point of access to the object.**

  - **Ensure that only one instance of a class is created ---Allow multiple instances when required.**

Singleton Solution: Main Idea

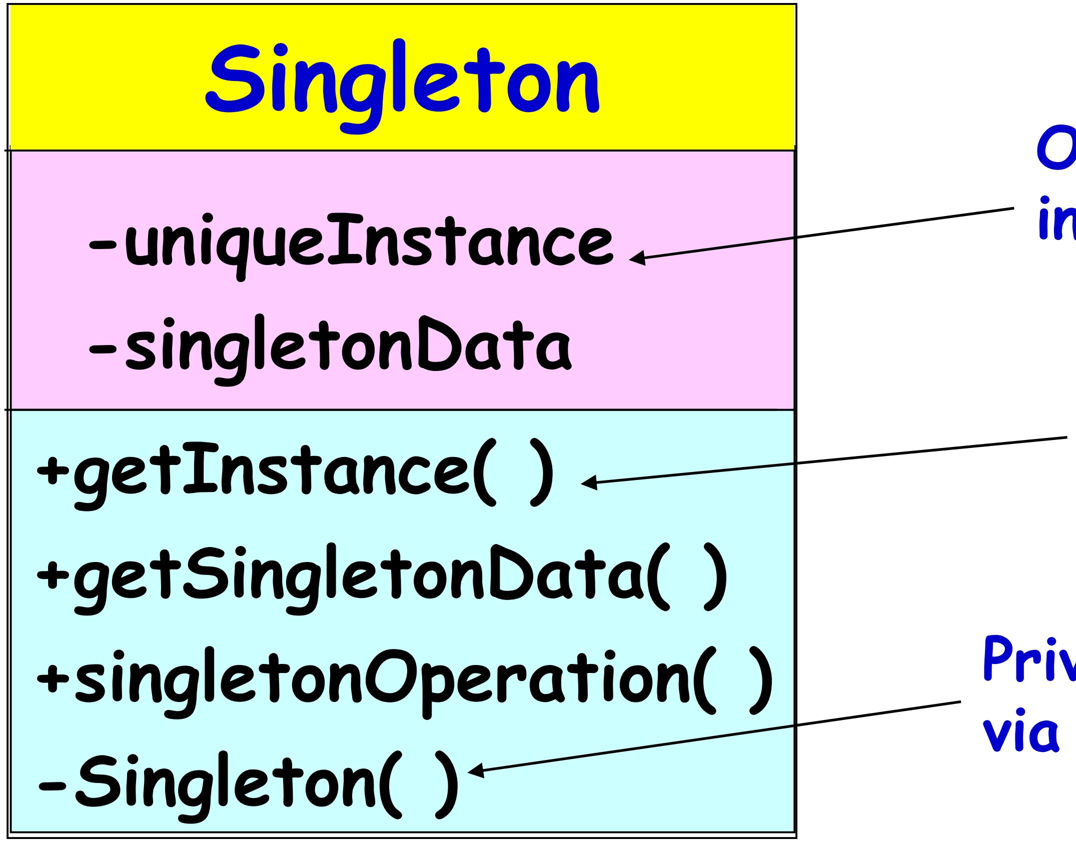
- Create an object with operation:
  - **getInstance()**
- When the class is first accessed:
  - Singleton instance is created and the object's identity is returned.
- On subsequent calls to getInstance():
  - No new instance is created,
  - Id of existing object is returned.

The Singleton Pattern

- A Singleton class itself is responsible for keeping track of its sole instance.
  - **How?** --- intercept all requests to create new objects.

- **Singletons maintain a static reference to the sole singleton instance:**
  - Return a reference to that instance from a static instance() method.

Singleton Structure



Singleton

-uniqueInstance

-singletonData

+getInstance( )

+getSingletonData( )

+singletonOperation( )

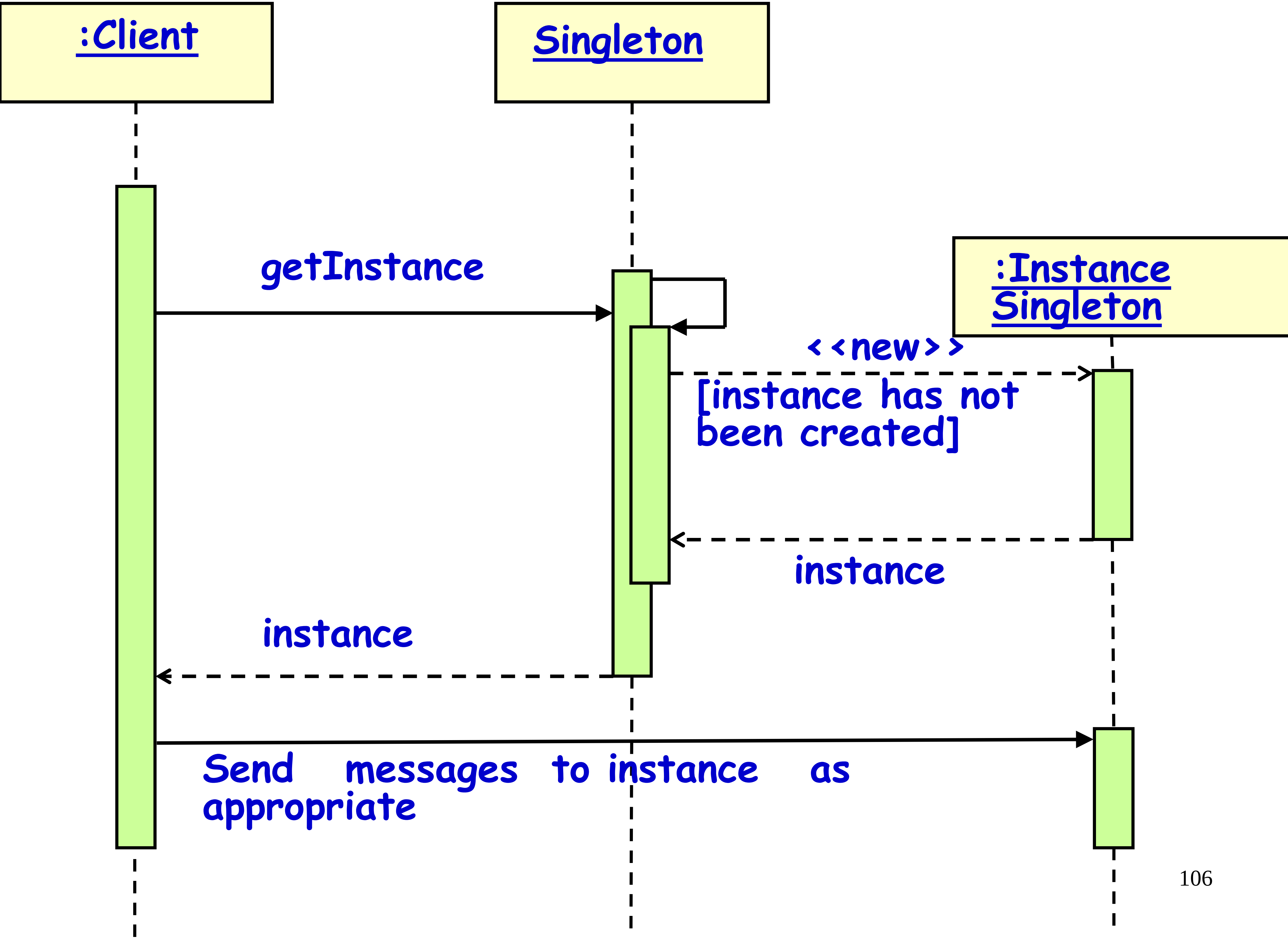
-Singleton( )

Object identifier for singleton instance, class scope i.e. static

Returns object identifier for unique instance

Private constructor only accessible via getInstance()

```
getInstance( ) {
    if ( uniqueInstance == null )
    { uniqueInstance = new Singleton( ); }
    return uniqueInstance;
}
```



:Client

Singleton

:Instance Singleton

getInstance

<<new>>

[instance has not been created]

instance

instance

Send messages to instance as appropriate

Singleton: Example Code

```
Public Class Singleton {
  private static Singleton uniqueInstance = null;

   private Singleton( ) { .. } // private constructor
   public static Singleton getInstance( ) {
      if (uniqueInstance == null)
            uniqueInstance = new Singleton();
      return uniqueInstance;
  }
}
```

Exercise 1

- Objects in a certain application:
  - Get a Database Manager: **DBMgr.getDBMgr();**
- It needs to be ensured that:
  - There is only one Database Manager object.
  - Need to disallow creation of more than one object of this class.

Exercise 1 -- Solution

```
class DBMgr {
   private static DBMgr pMgr=null;
    Private DBMgr() { } // No way to create outside of this Class
    publicstatic DBMgr getDBMgr() { // Only way to create.
   if (pMgr == NULL)  pMgr = new DBMgr();
   return pMgr;  }
   public Connection getConnection();    }
```

Usage:

```
DBMgr dbmgr =  DBMgr.getDBMgr();
//Created first time called
```