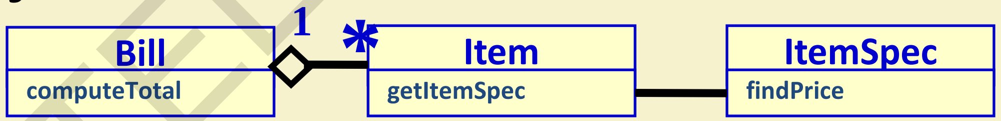
Law of Demeter

Problem:

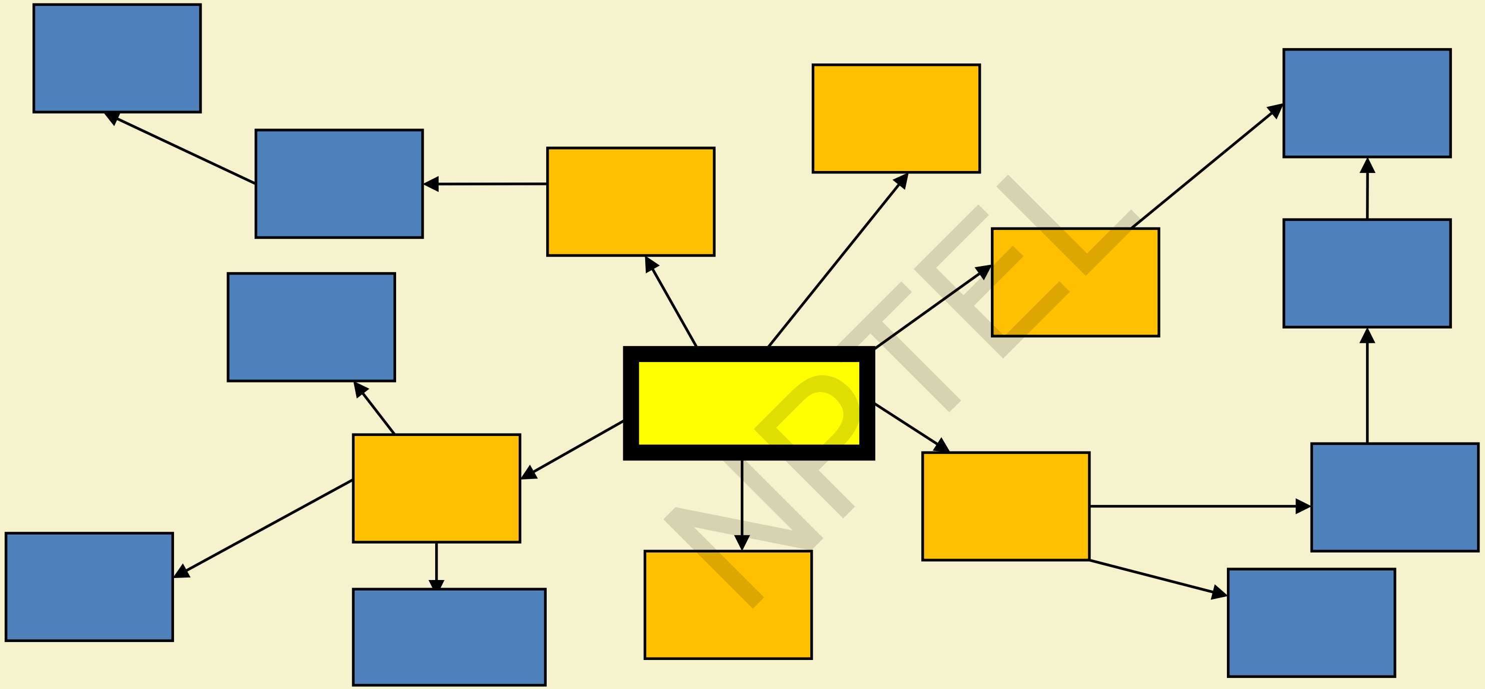
- How to avoid a class from interacting with indirectly associated objects?



Solution:

- If two classes have no other reason to be directly aware of each other:
  - Then the two classes should not directly interact.

Collaborators



Law of Demeter

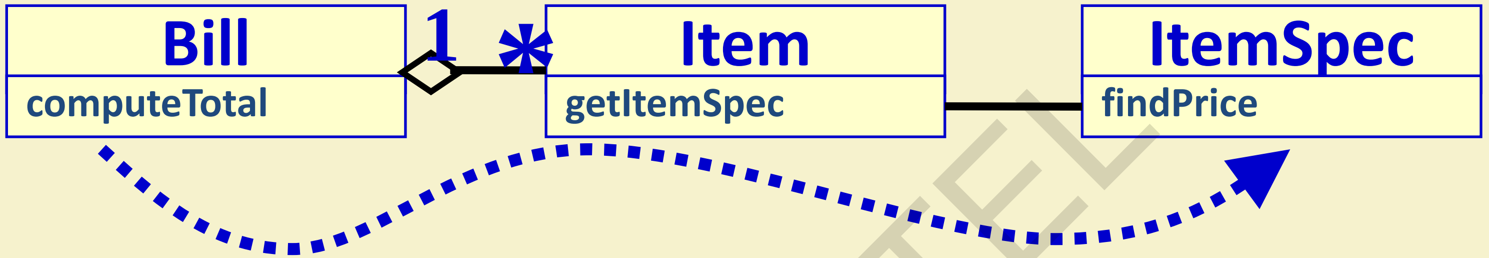
- In a method, calls should only be made to the methods of following objects:
  - **This object (or self)**
  - **An object parameter of the method**
  - **An object attribute of self**
  - **An object created within the method**

A Hypothetical example

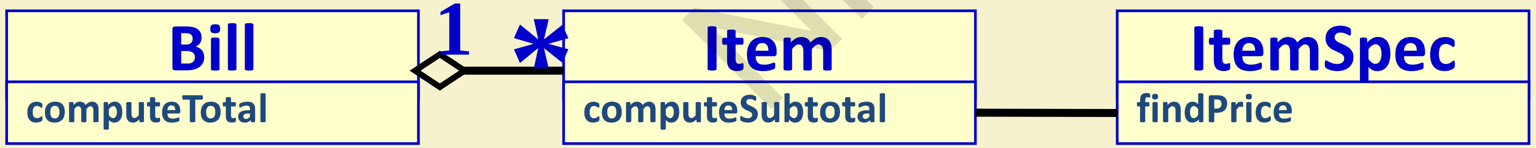
```
class A {
  private B b = new B();

  public void m() {
    this.b.c.foo();  // High
                       coupling: bad
  }
}
```

```
class B {
  C c;
}
class C {
  public void foo() {
  }
}
```

Law of Demeter: Example



Violation: item.getItemspec().findprice()



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
- **Tradeoff:**
  - More statements, but simpler statements
- Experiments show significantly improved maintainability.

Law of Demeter: Final Analysis

- Reduces coupling between classes
- Adds a small amount of overhead in the form of indirect method calls

"The basic effect of applying this Law 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

- **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 behaviour,**
  - **Rather than using if statement to test the type.**

Polymorphism Pattern Advantage

- Easily extendible as compared to using explicit selection logic

GoF Design Patterns

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

What Problems do GoF DPs solve?

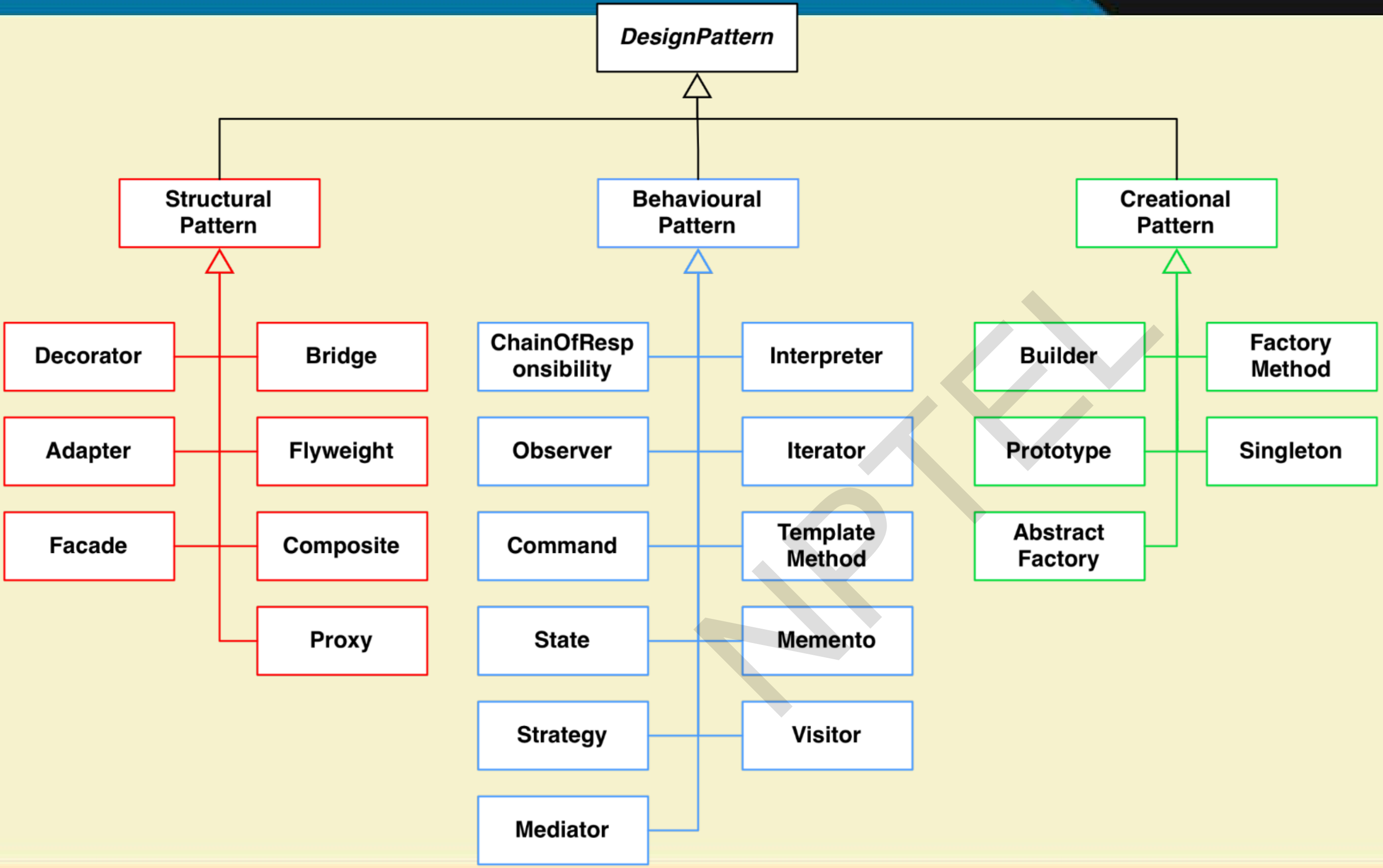
Types of GoF Patterns

- **Creational patterns:**
  - How objects are created?
- **Structural patterns:**
  - How classes or objects are composed into larger groups?
- **Behavioral patterns:**
  - How responsibility is 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**
    - Encapsulate complex structures
- **Behavioral Patterns**
  - Assignment of responsibilities between objects: Who does what?
  - Behaviorial patterns concern cohesion and coupling.
- **Creational Patterns**
  - The goal is to provide a simple abstraction for a complex instantiation process.
  - Make the system independent from the way its objects are created, composed and represented.

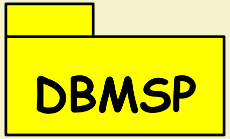
Taxonomy of GoF Patterns (23 Patterns)



Facade Pattern

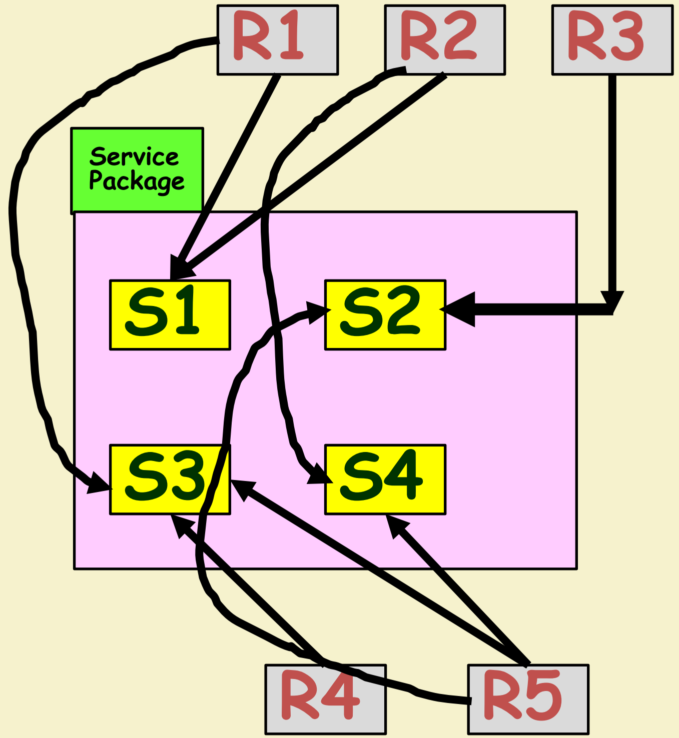
Problem: **How to access a large number of classes (having complex internal interactions) in a package?**

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

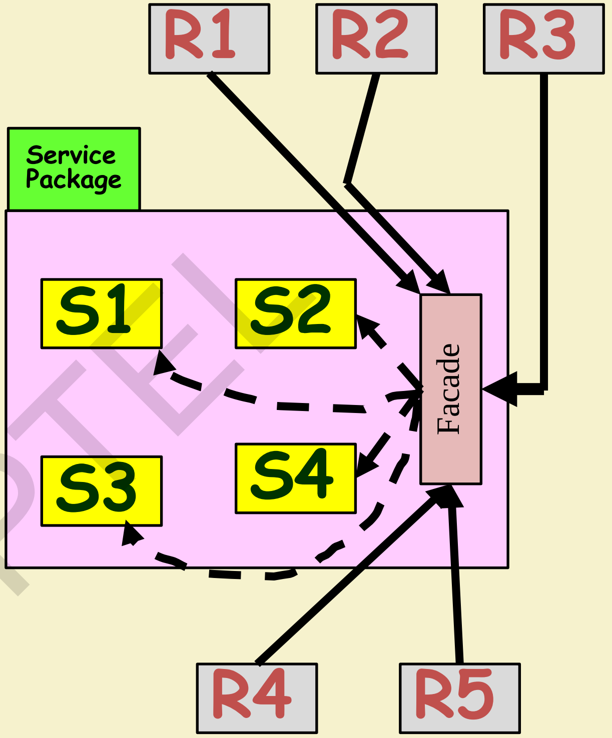


- **Solution**: Create a facade class (e.g. DBfacade) d to provide a common interface to the services of the package.

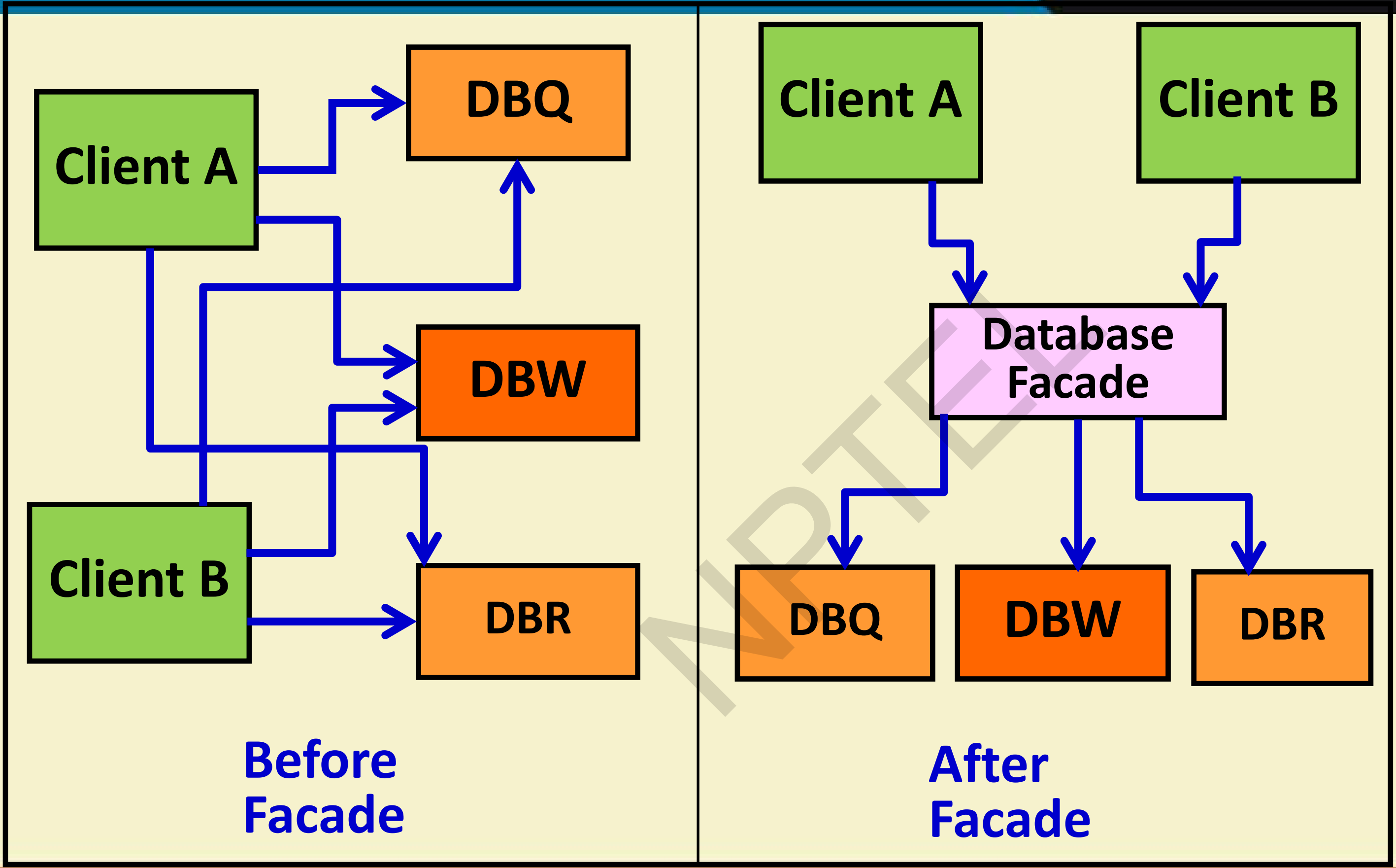
Facade



(a) Service invocation without a façade

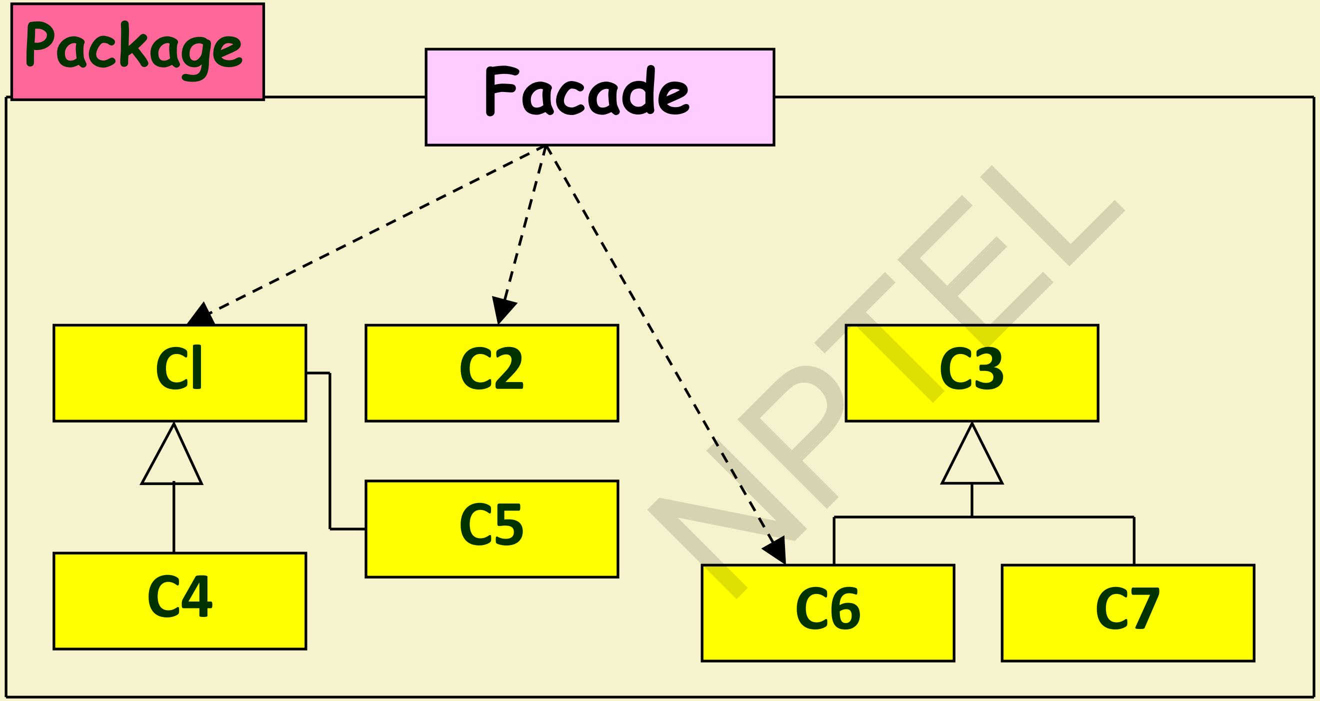


(b) Service invocation using a façade class

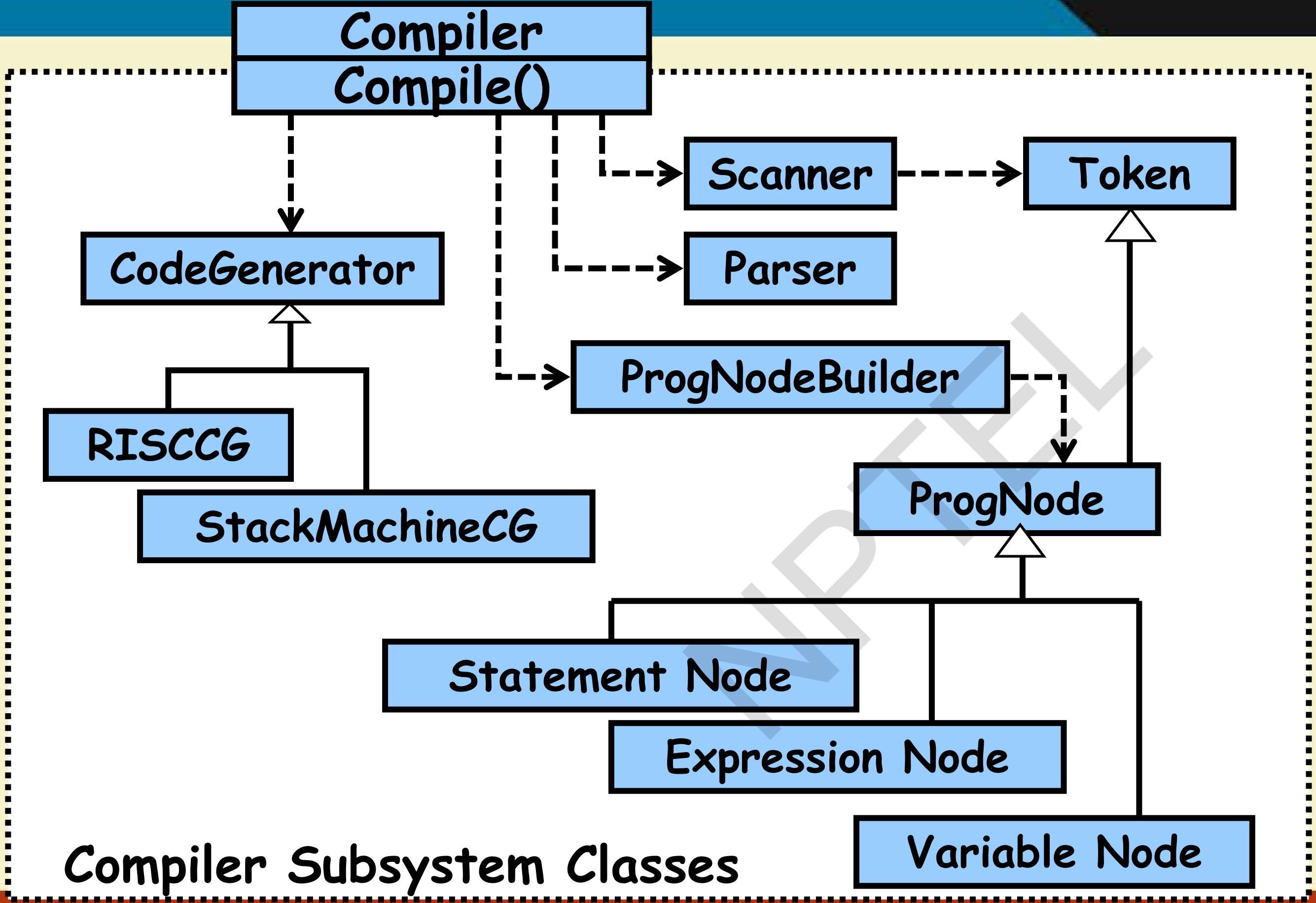


Before and After Using Facade

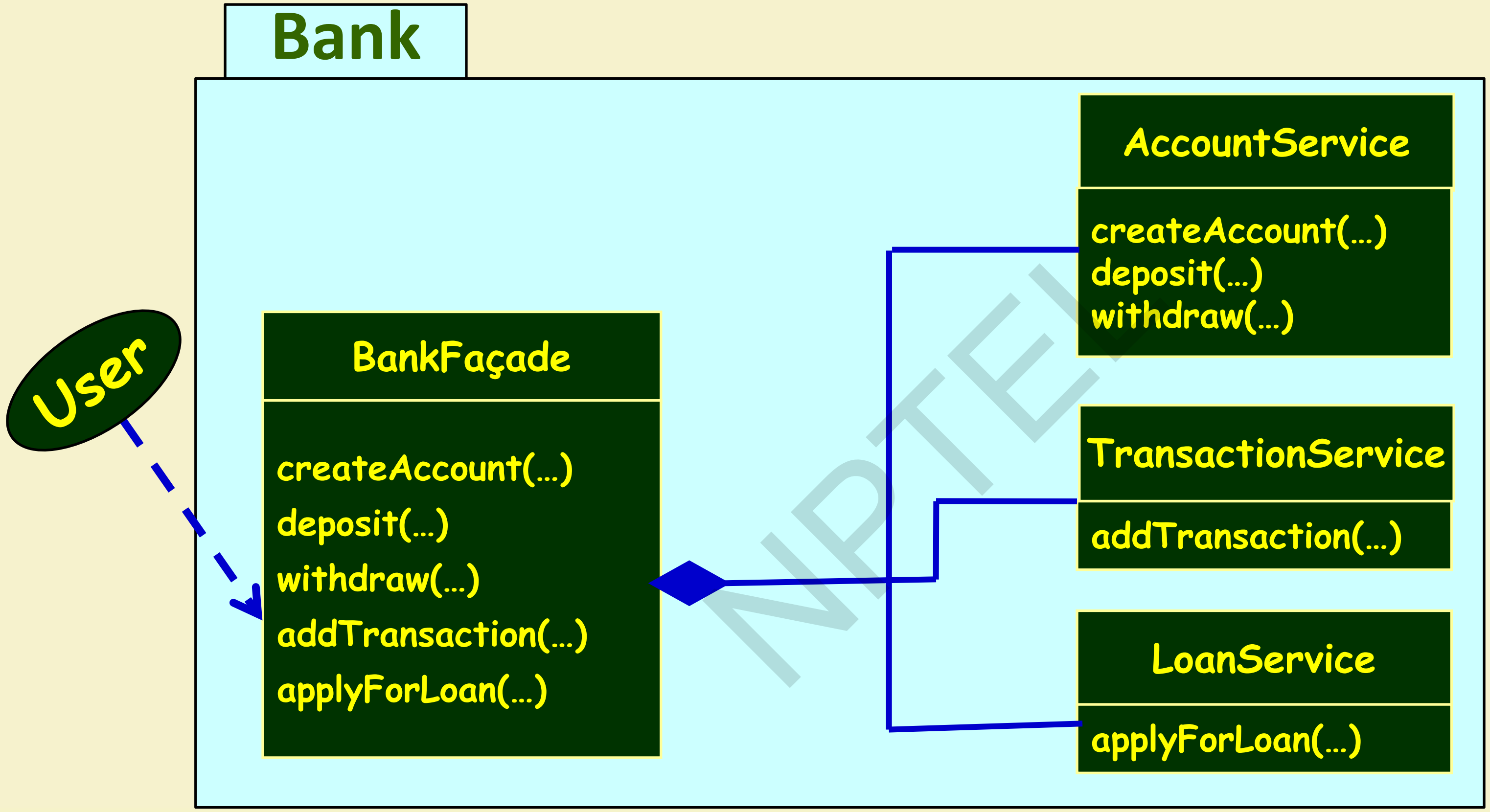
Facade Structure



Facade Pattern: Example



Façade: Bank Example



…

Façade Code: 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(accou
    ntId, amount);

    }

    public addTransaction(Transaction
    tx) {
          transactionService.addTransac
    tion(tx);

    }

    public applyForLoan(Customer cust,
    LoanDetails loan) {
          loanService.apply(cust, loan);

    }

}
```

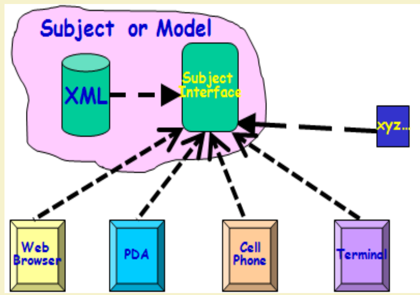
Discussions on Façade Pattern

- Clients communicate with the subsystem by sending requests to Façade:
  - **Façade forwards them to the appropriate subsystem object(s).**
  - **Promotes low coupling; clients know only about the Façade.**
- **Often, does little more than just 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 that use the Façade:
  - Don’t have to access its subsystem objects directly.

Observer Pattern

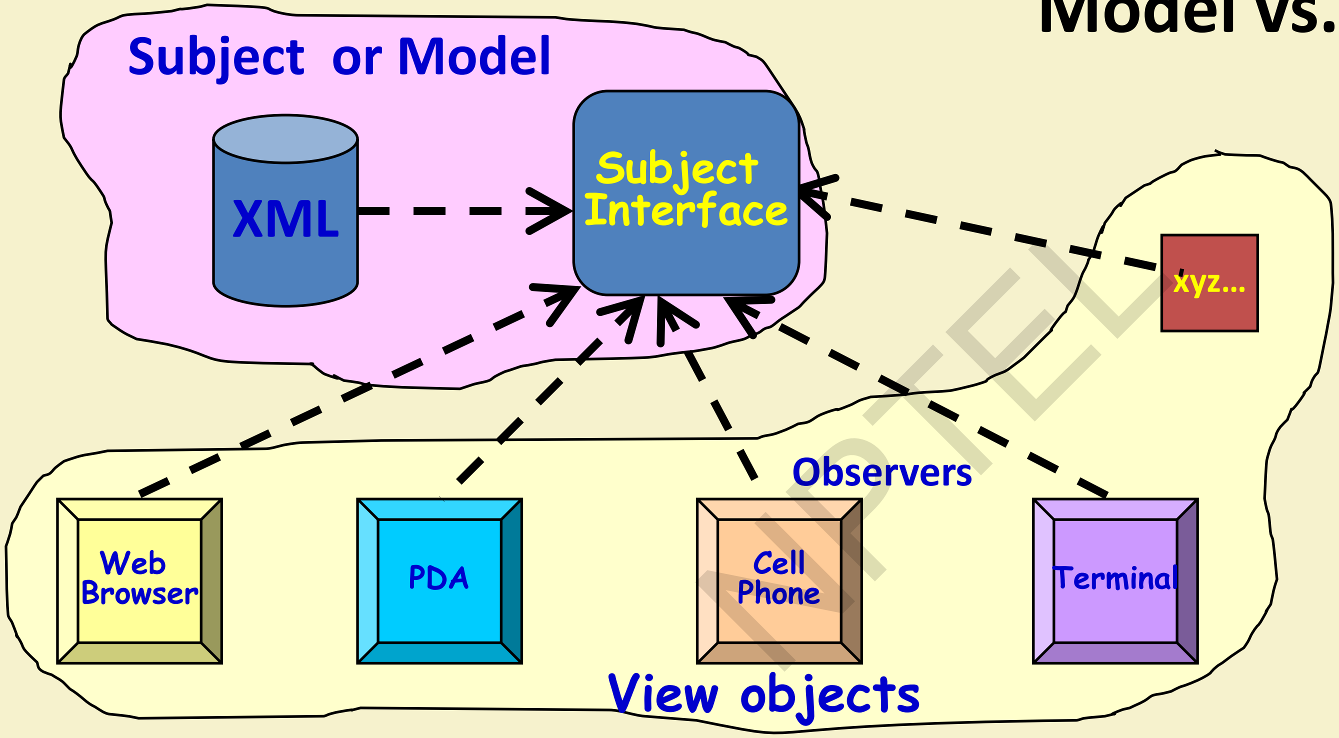


- **Problem:**
  - When a model object changes state asynchronously and is accessed by several view objects:
    - **How should the interactions between the model and the view objects be structured?**
- **Solution: Define a one-to-many dependency, so that when model changes state, all of its dependents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

Observer Pattern --- 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
  - But, how should they communicate?

Model vs. View Objects

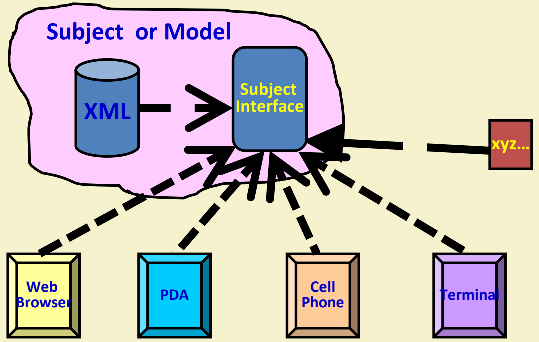


Model View Separation Patterns: Solution Overview

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

Solution 1: Pull from Above

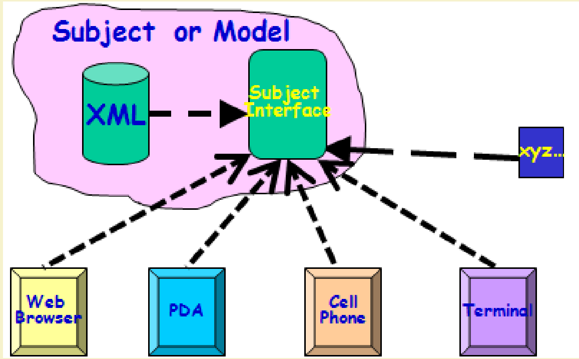
- **Most obvious solution:**
  – View (boundary) objects invoke model objects.
- **Problem:**
  – Views can become inconsistent
  – View object does not know when model state changes



Context in Which Pull from Above Does Not Work

- **Data changes asynchronously**
  - An event in a simulation experiment, stock market alert, network monitor, etc.
  - A mouse event causes change to a shape.
- **Dynamic set of observers**

Observer Pattern: Context



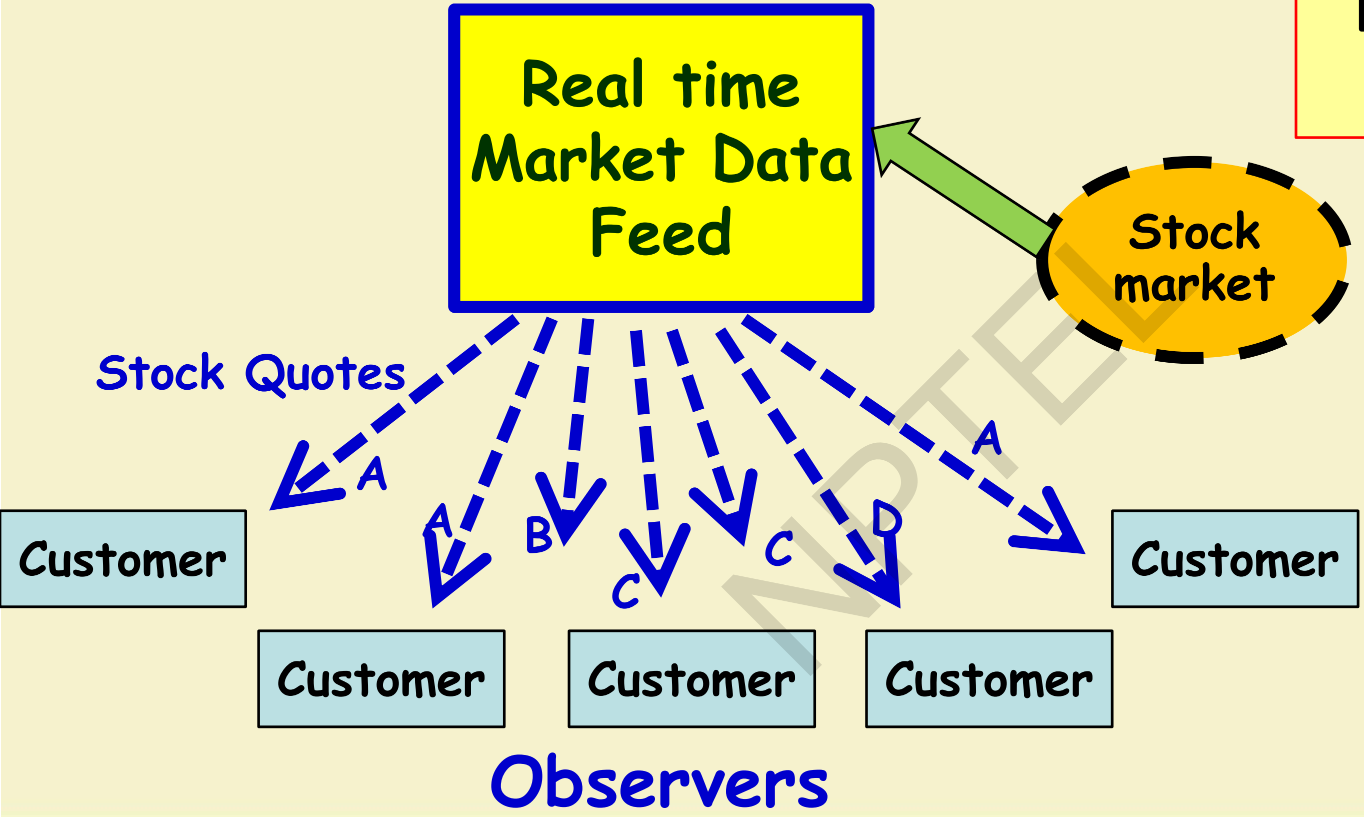
- There could be many observers
  - Also the number of observers may vary dynamically
  - **Each observer may react differently to the same notification**
- Subject (model) should be as much decoupled from the observers as possible:
  - Allow observers to change independently of the subject

Observer -- Non software example

15
17
BID
21
BID
33
BID

When an object changes its state, all its dependants are notified.

Example: Stock Quote Service



Observer Pattern: Main Idea

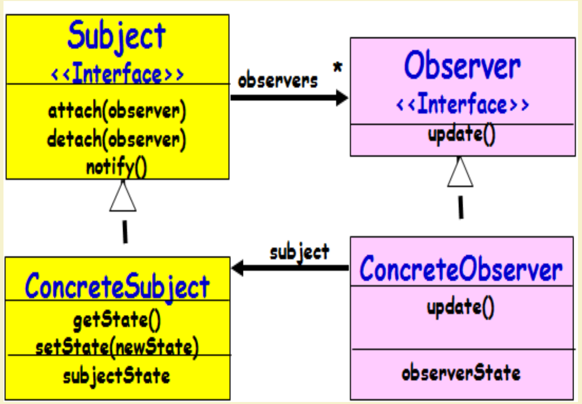
- When observable (model) changes state:
  - All dependent objects are notified --- these objects update themselves.
  - **Allows for consistency between related objects without tightly coupling the classes.**

Observer Pattern: Solution

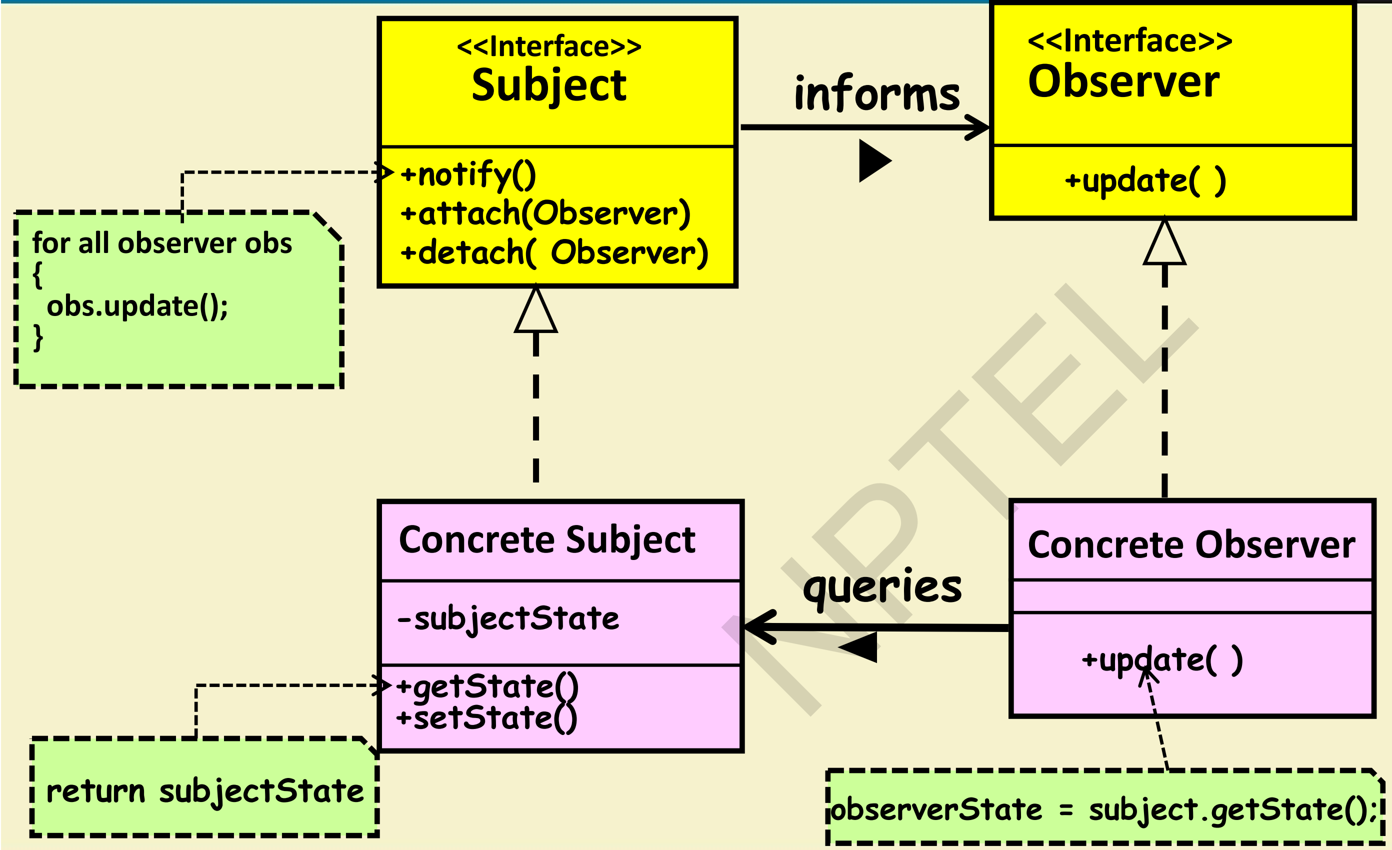
- Observers should register themselves with the model object.
  - **The model object maintains a list of the registered observers.**
- When a change occurs to a model object:
  - Model notifies all registered observers.
  - Subsequently, each observer queries the model object to get any specific information about the changes.

Key Players

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



Structure of Observer Pattern

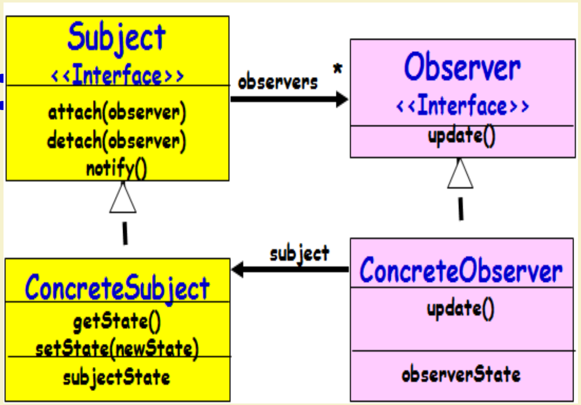


Observer Pattern

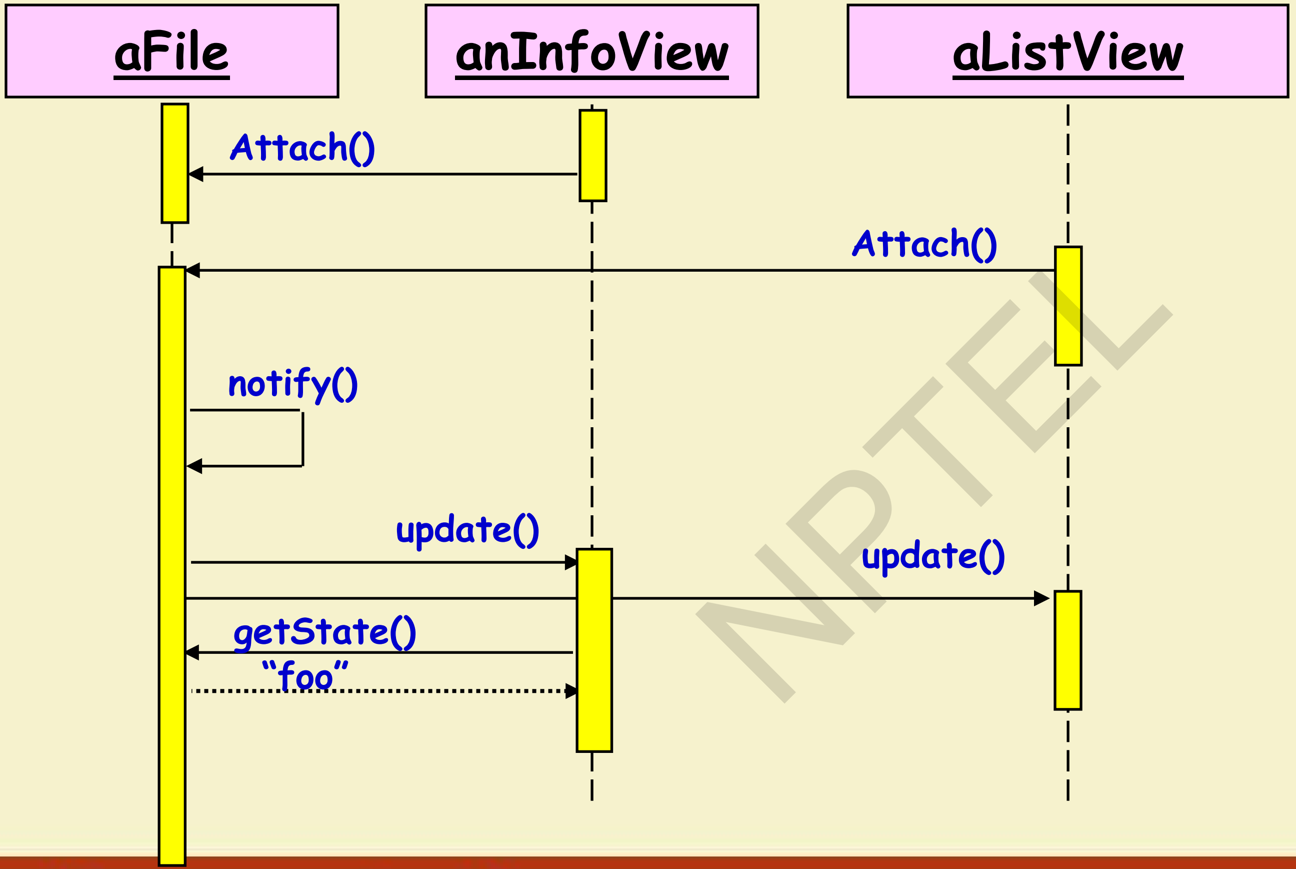
- Defines a one-to-many dependency between objects:
  - **When one object changes state, all its dependents are notified and updated automatically.**
- Used to decouple the subject from the observer:
  - Since the subject has little information about needs of the observer.
  - Can result in excessive notifications.

Working of Observer Pattern

- ConcreteSubject notifies its observers:
  - Whenever a change makes its state inconsistent with the observers.
- After being informed of change:
  - A ConcreteObserver queries the subject to reconcile its state with subjects.



Observer: Sequence diagram



Activities of Update

- ConcreteObserver.Update():
  - Repaint a user interface
  - Check for exceeding thresholds, etc.
  - These are operations that might clutter up a domain class (Subject).

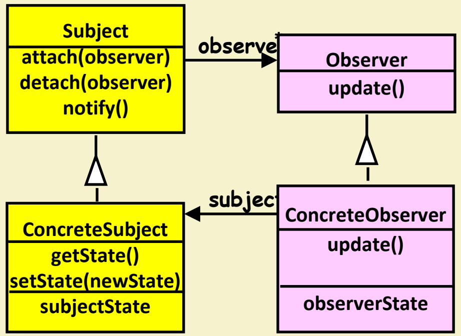
Observer Java Implementation

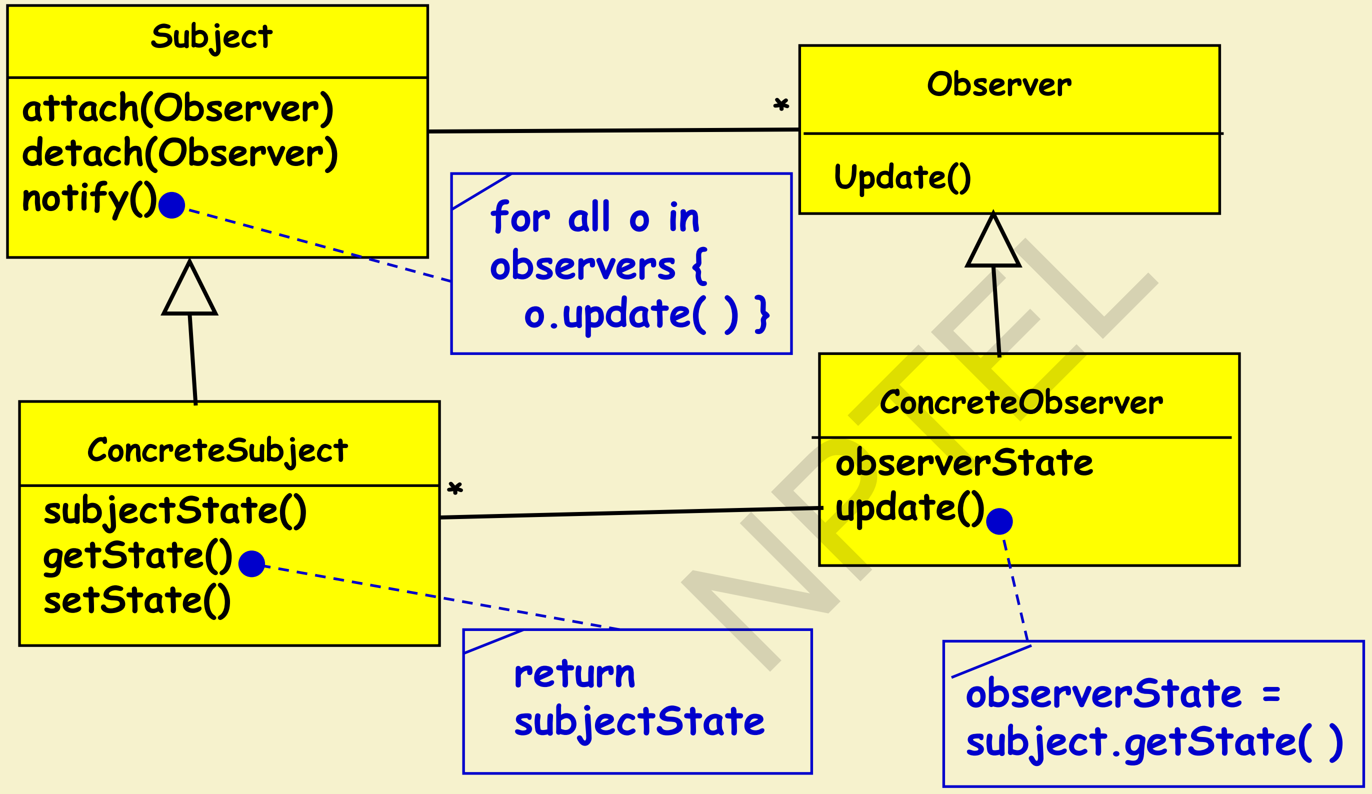
- A Subject class usually maintains:
  - An ArrayList of ids of the registered Observers
  - Makes it easier when a random object wants to attach or detach

Observer Pattern Java Implementation

```
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()
*
Observer
Update()
for all o in observers { o.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 chnged 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);}

  public void removeObserver(Observer o) {
    observers.remove(o);}

  public void notifyObservers() {
      Iterator i = observers.iterator();
      while (i.hasNext()) {
             Observer o = (Observer) i.next();
             o.update(this);}
}
```

Inbuilt Java Observers

```
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**
  - Class, extended by concrete Observable

Java Observer Class

- Implementers must have 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 an identifier and also a data object y

Java Observable Class

- Extended by concrete Observables
- 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 observer 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() {}
 ...
}
```

Observer Pattern- Implementation Example 1

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

A Concrete Observer.

```
class StatsTable extends Observable{
      public notify() {
      …
      }
}
```

Observer Pattern - Consequences

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

- **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**
  - – **Dangling references**
  - – Registering modifications of interest explicitly

Limitations of Observer Pattern

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

Observer Pattern: Final Words

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

Observer: Final Words

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

Java Observer Deprecated!

- They don't provide a rich enough event model for applications!
  - For example, they support only the notion that something has changed, but they don't convey any information about what has changed
- Not threadsafe!
- Make use of **PropertyChangeListener** instead:
  - Which itself is an implementation of Observer pattern

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.