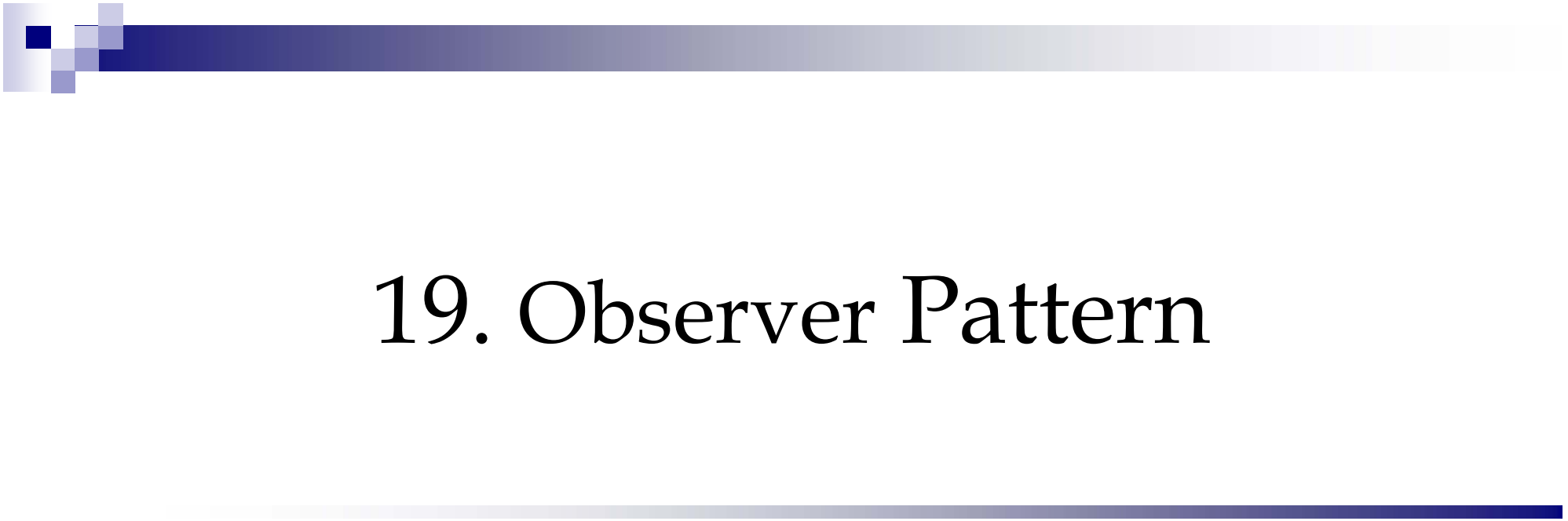
Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern University



19. Observer Pattern

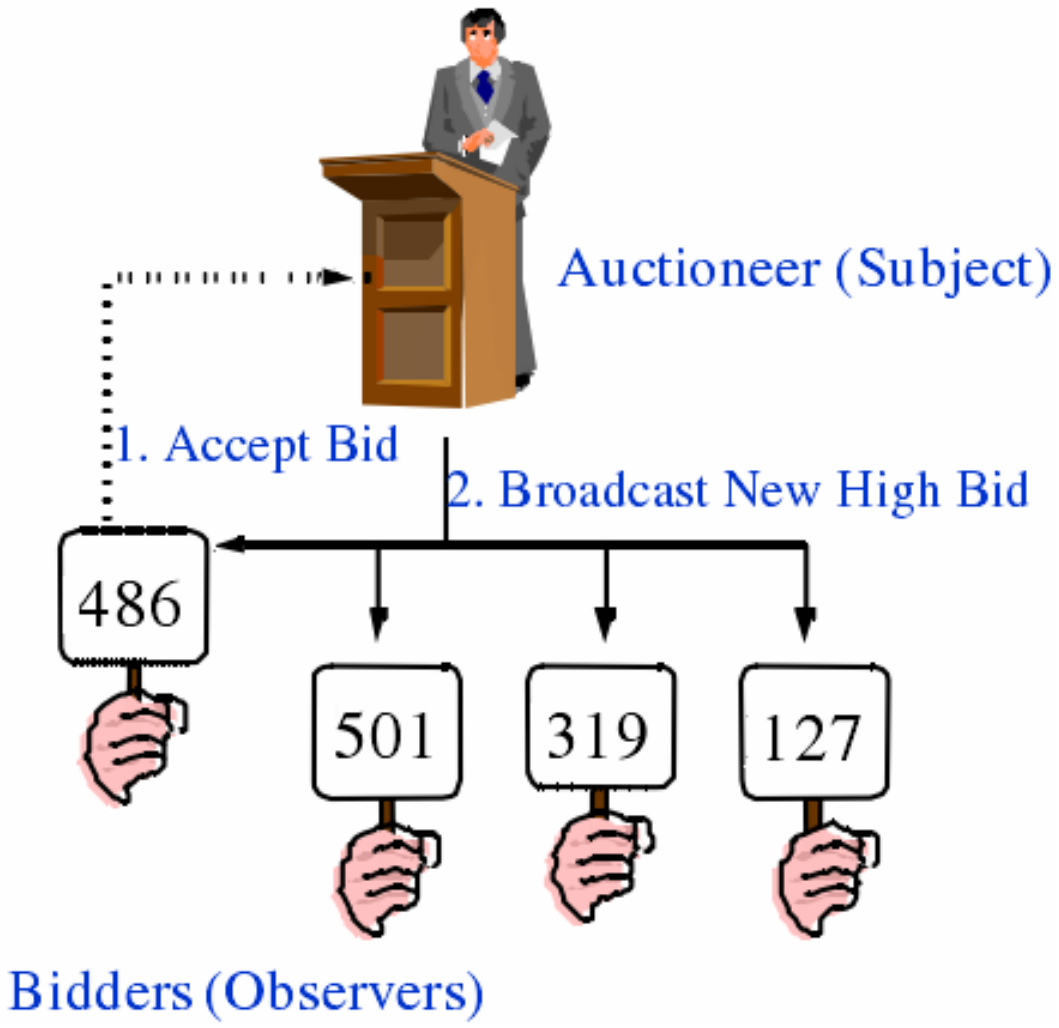
Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某，一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

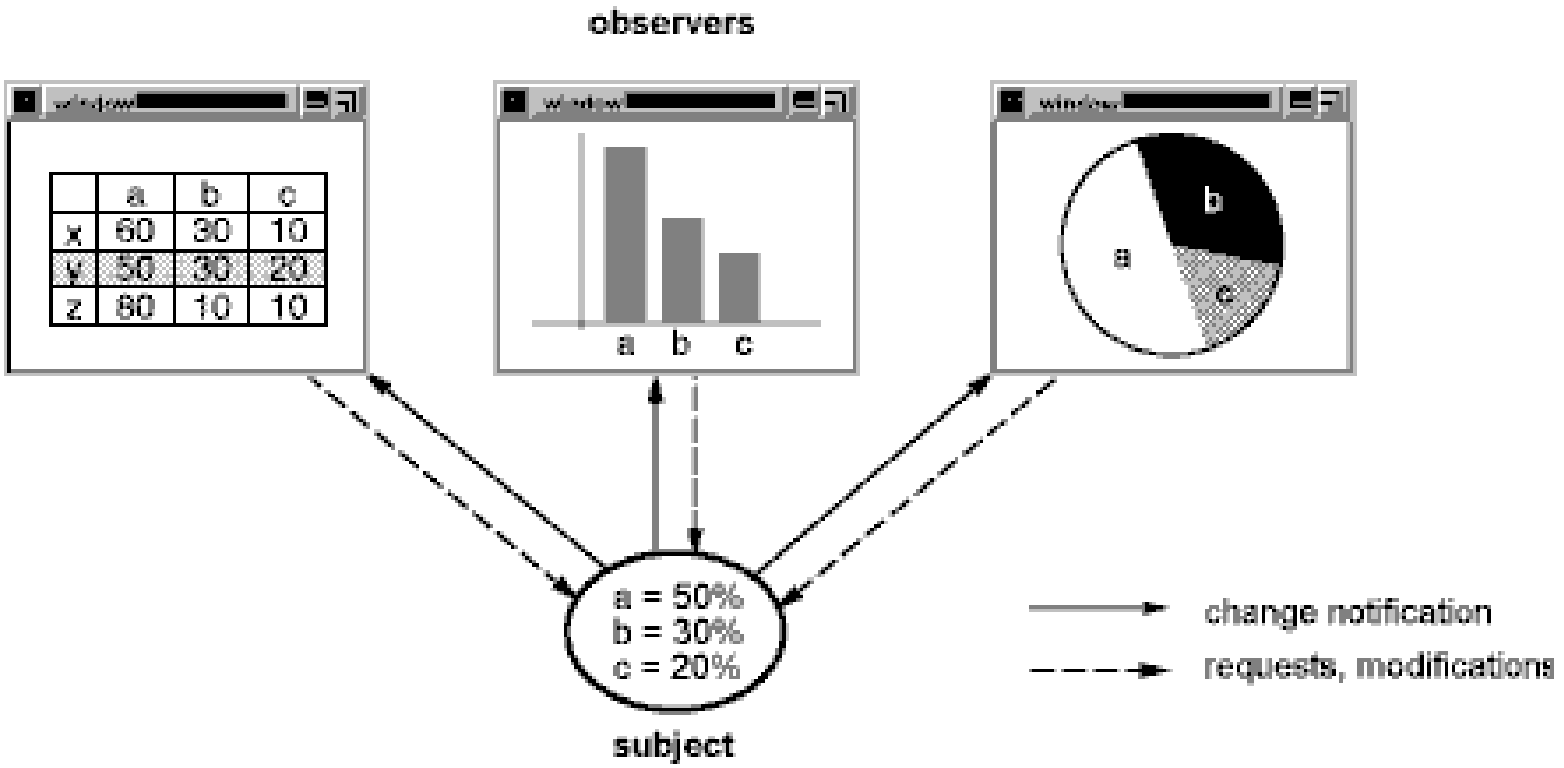
Intent

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

Example


Auctioneer (Subject)
1. Accept Bid
2. Broadcast New High Bid
486
501
319
127
Bidders (Observers)

Example


observers
a b c
x 60 30 10
y 50 30 20
z 60 10 10
a b c
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modifications

```
interface Auctioneer {
    public void attach(Bidder bidder);
    public void detach(Bidder bidder);
    public void clear();
    public void notifying();
    public void asking();
    public boolean accept();
    public Bidder currentBidder();
}

abstract class AbstractAuctioneer implements Auctioneer {
    protected List<Bidder> bidders;
    public AbstractAuctioneer() {
        bidders = new ArrayList<Bidder>();
    }
    public void attach(Bidder bidder) {
        if (!bidders.contains(bidders)) {
            bidders.add(bidder);
        }
    }
    public void detach(Bidder bidder) {
        bidders.remove(bidder);
    }
    public void clear() {
        bidders.clear();
    }
}
```

AuctioneerImpl Part 1

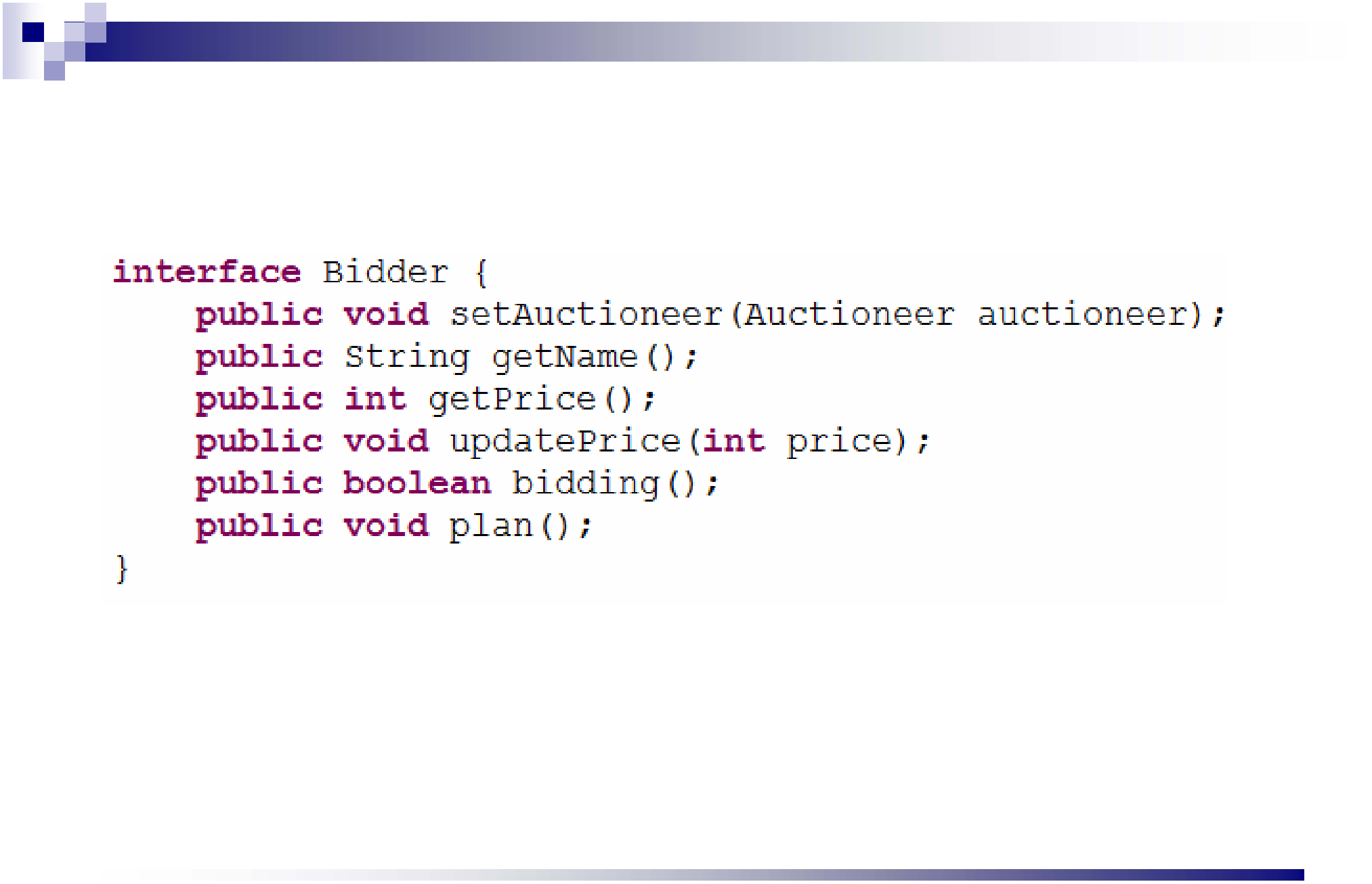
```
class AuctioneerImpl extends AbstractAuctioneer implements Auctioneer {
    private Bidder currentBidder;
    private int notifiedCount = 0;
    private int maxNotifiedCount = 3;
    public AuctioneerImpl(int initPrice, int maxNotifiedCount) {
        currentBidder = new BidderImpl("Init", 0, 0);
        currentBidder.updatePrice(initPrice);
        this.maxNotifiedCount = maxNotifiedCount;
    }
    public Bidder currentBidder() {
        return currentBidder;
    }
```

AuctioneerImpl Part 2

```
public void asking() {
    boolean biderChanged = false;
    for (Iterator<Bidder> it = bidders.iterator(); it.hasNext();) {
        Bidder bidder = it.next();
        boolean state = bidder.bidding();
        if (!state) {
            it.remove();
            System.out.println(bidder + " quit!");
        } else if (currentBidder.getPrice() < bidder.getPrice()) {
            currentBidder = bidder;
            biderChanged = true;
        }
    }
    if (!biderChanged) {
        notifiedCount++;
        System.out.println("Notified" + notifiedCount);
    }
}
```

AuctioneerImpl Part 3

```
    public void notifying() {
        for (Bidder bidder : bidders) {
            bidder.updatePrice(currentBidder.getPrice());
        }
    }
    public boolean accept() {
        if (notifiedCount >= maxNotifiedCount) {
            this.clear();
            System.out.println("Accept:" + currentBidder);
            return true;
        }
        return false;
    }
}
```

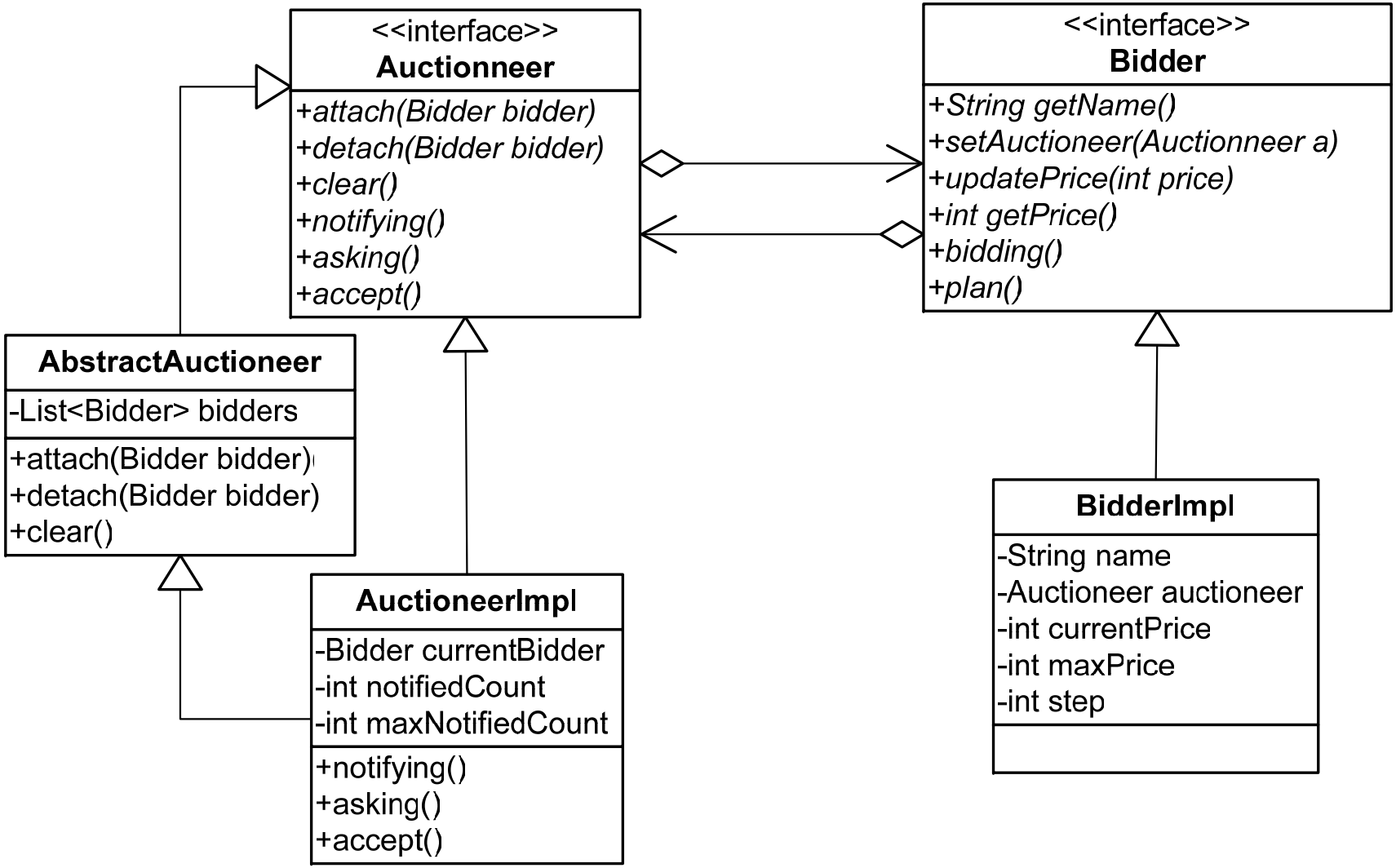

interface Bidder {
public void setAuctioneer(Auctioneer auctioneer);
public String getName();
public int getPrice();
public void updatePrice(int price);
public boolean bidding();
public void plan();
}

```
class BidderImpl implements Bidder {

    private String name;
    private Auctioneer auctioneer;
    private int currentPrice;
    private int maxPrice;
    private int step;

    public BidderImpl(String name, int maxPrice, int step) {
        this.name = name;
        this.maxPrice = maxPrice;
        this.step = step;
    }
    public void setAuctioneer(Auctioneer auctioneer) {
        this.auctioneer = auctioneer;
        auctioneer.attach(this);
    }
    public String getName() {
        return this.name;
    }
    public int getPrice() {
        return currentPrice;
    }
```

```
    public boolean bidding() {
        if (auctioneer == null) {
            return false;
        }
        if (auctioneer.currentBidder() == this) {
            return true;
        }
        if (currentPrice > maxPrice) {
            return false;
        }
        int price = currentPrice + step;
        currentPrice = price < maxPrice ? price : maxPrice;
        System.out.println(this);
        return true;
    }
    public void updatePrice(int price) {
        this.currentPrice = price;
    }
    // Use Strategy pattern or Template Method pattern here
    public void plan() {
        // Defining the bidding strategy dynamically
        // currentPrice, maxPrice, step
    }
    public String toString() {
        return this.name + ": " + this.currentPrice;
    }
}
```

```
public void testBidding() {
    Auctioneer auctioneer = new AuctioneerImpl(50, 3);
    Bidder tom = new BidderImpl("Tom", 100, 5);
    Bidder jack = new BidderImpl("Jack", 120, 10);
    Bidder marry = new BidderImpl("Marry", 150, 20);
    Bidder aike = new BidderImpl("Aike", 200, 20);
    tom.setAuctioneer(auctioneer);
    jack.setAuctioneer(auctioneer);
    marry.setAuctioneer(auctioneer);
    aike.setAuctioneer(auctioneer);
    while (!auctioneer.accept()) {
        auctioneer.notifying();
        auctioneer.asking();
    }
}
```

- Tom: 55
- Jack: 60
- Marry: 70
- Aike: 70
- Tom: 75
- Jack: 80
- Marry: 90
- Aike: 90
- Tom: 95
- Jack: 100
- Marry: 110
- Aike: 110
- Tom: 110 quit!
- Jack: 120
- Marry: 130
- Aike: 130
- Jack: 130 quit!
- Aike: 150
- Marry: 150
- Notified1
- Marry: 150
- Notified2
- Marry: 150
- Notified3
- Accept:Aike: 150

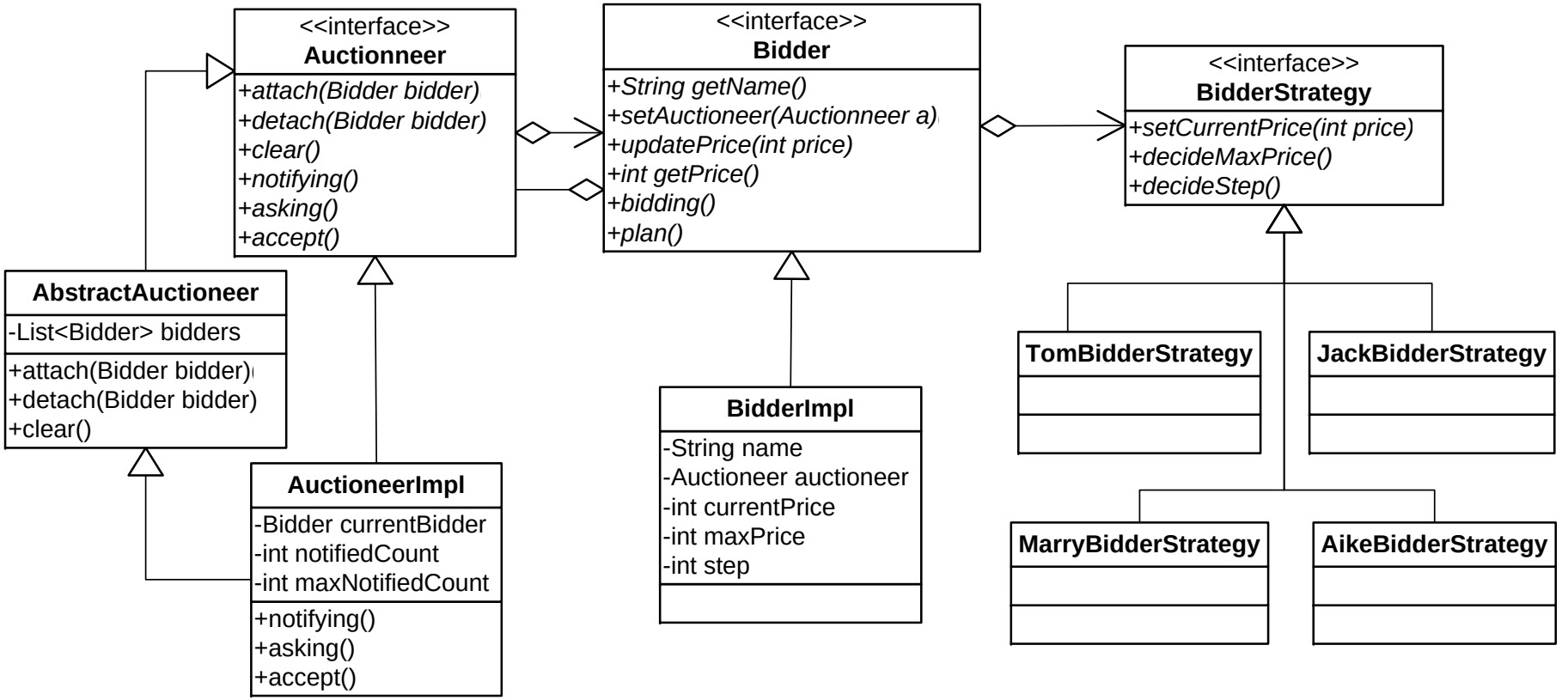

<<interface>>
Auctionneer
+attach(Bidder bidder)
+detach(Bidder bidder)
+clear()
+notifying()
+asking()
+accept()
<<interface>>
Bidder
+String getName()
+setAuctioneer(Auctionneer a)
+updatePrice(int price)
+int getPrice()
+bidding()
+plan()
AbstractAuctioneer
-List<Bidder> bidders
+attach(Bidder bidder)
+detach(Bidder bidder)
+clear()
AuctioneerImpl
-Bidder currentBidder
-int notifiedCount
-int maxNotifiedCount
+notifying()
+asking()
+accept()
BidderImpl
-String name
-Auctioneer auctioneer
-int currentPrice
-int maxPrice
-int step

```
interface BidderStrategy {
    public void setCurrentPrice(int price);
    public int decideMaxPrice();
    public int decideStep();
}
class TomBidderStrategy implements BidderStrategy {
    private int currentPrice;
    public int decideMaxPrice() {
        return currentPrice *2;
    }
    public int decideStep() {
        return currentPrice / 10;
    }
    public void setCurrentPrice(int price) {
        this.currentPrice = price;
    }
}
```

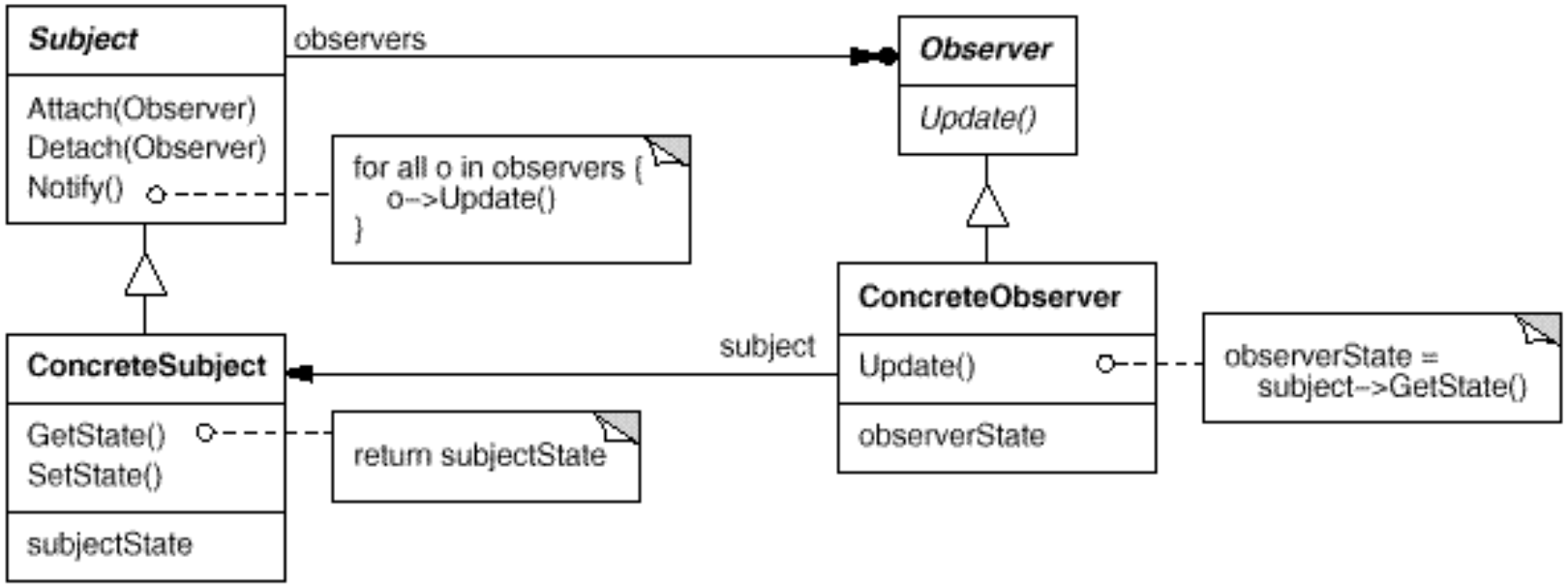
```
class StrategyBidder implements Bidder {
    private String name;
    private Auctioneer auctioneer;
    private BidderStrategy strategy;
    private int currentPrice;
    private int maxPrice;
    private int step;
    public StrategyBidder(String name, BidderStrategy strategy) {
        this.name = name;
        this.strategy = strategy;
    }
    public void setAuctioneer(Auctioneer auctioneer) {
        this.auctioneer = auctioneer;
        auctioneer.attach(this);
    }
    public String getName() {
        return this.name;
    }
    public int getPrice() {
        return currentPrice;
    }
```

```
    public boolean bidding() {
        if (auctioneer == null) {
            return false;
        }
        if (auctioneer.currentBidder() == this) {
            return true;
        }
        if (currentPrice > maxPrice) {
            return false;
        }
        int price = currentPrice + step;
        currentPrice = price < maxPrice ? price : maxPrice;
        System.out.println(this);
        return true;
    }

    public void updatePrice(int price) {
        this.currentPrice = price;
        plan();
    }
    public void plan() {
        strategy.setCurrentPrice(currentPrice);
        maxPrice = strategy.decideMaxPrice();
        step = strategy.decideStep();
    }
    public String toString() {
        return this.name + ": " + this.currentPrice;
    }
}
```

```
public void testStrategyBidding() {
    Auctioneer auctioneer = new AuctioneerImpl(50, 3);
    Bidder tom = new StrategyBidder("Tom", new TomBidderStrategy());
    // Bidder jack = new StrategyBidder("Jack",new JackBidderStrategy());
    // Bidder marry = new StrategyBidder("Marry",new MarryBidderStrategy());
    // Bidder aike = new StrategyBidder("Aike",new AikeBidderStrategy());
    tom.setAuctioneer(auctioneer);
    // jack.setAuctioneer(auctioneer);
    // marry.setAuctioneer(auctioneer);
    // aike.setAuctioneer(auctioneer);
    while (!auctioneer.accept()) {
        auctioneer.notifying();
        auctioneer.asking();
    }
}
```


<<interface>>
Auctionneer
+attach(Bidder bidder)
+detach(Bidder bidder)
+clear()
+notifying()
+asking()
+accept()
<<interface>>
Bidder
+String getName()
+setAuctioneer(Auctionneer a)
+updatePrice(int price)
+int getPrice()
+bidding()
+plan()
<<interface>>
BidderStrategy
+setCurrentPrice(int price)
+decideMaxPrice()
+decideStep()
AbstractAuctioneer
-List<Bidder> bidders
+attach(Bidder bidder)
+detach(Bidder bidder)
+clear()
AuctioneerImpl
-Bidder currentBidder
-int notifiedCount
-int maxNotifiedCount
+notifying()
+asking()
+accept()
BidderImpl
-String name
-Auctioneer auctioneer
-int currentPrice
-int maxPrice
-int step
TomBidderStrategy
JackBidderStrategy
MarryBidderStrategy
AikeBidderStrategy

Structure


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState

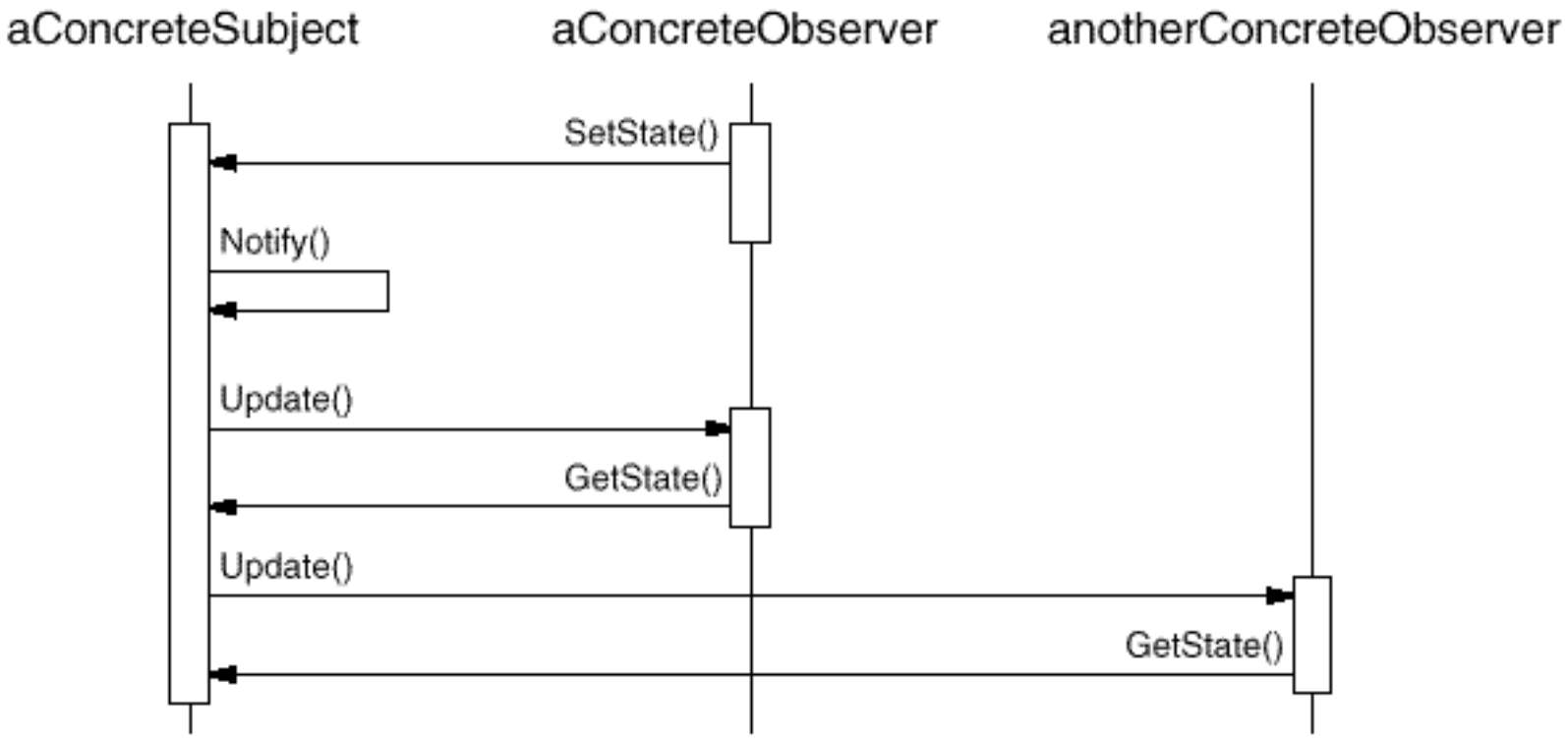
Participants

- Subject
  - Knows its observers. Any number of Observer objects may observe a subject.
  - Provides an interface for attaching and detaching Observer objects.
- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject
  - Stores state of interest to ConcreteObserver objects.
  - Sends a notification to its observers when its state changes.
- ConcreteObserver
  - Maintains a reference to a ConcreteSubject object.
  - Stores state that should stay consistent with the subject's.
  - Implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the ConcreteSubject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Collaborations



- Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject.

Consequences - advantages

- Abstract coupling between Subject and Observer.
  - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class.
  - The subject doesn't know the concrete class of any observer.
  - The coupling between subjects and observers is abstract and minimal.
- Subject and Observer belong to different layers of abstraction in a system. (DIP)
  - If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

Consequences - advantages

- Support for broadcast communication.
  - The notification that a subject sends needn't specify its receiver.
  - The notification is broadcast automatically to all interested objects that subscribed to it.
  - The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers.
  - This gives you the freedom to add and remove observers at any time.

Consequences – drawbacks

- Unexpected updates
  - Observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.
  - A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
  - Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious (伪造的) updates, which can be hard to track down.
- This problem is aggravated by the fact that the simple update protocol provides no details on what changed in the subject.

Consequences – drawbacks

- If a subject have many observers, it is time-costly to notify them all, especially when some method should be synchronized;
- If the observers depends each other circularly, the method will be invoked circularly (dead lock);
- Concurrence of the observers accessing the subject should be well considered ;
- The observers is only to know the subject is changed, but hard to know how a subject is modified.

Applicability

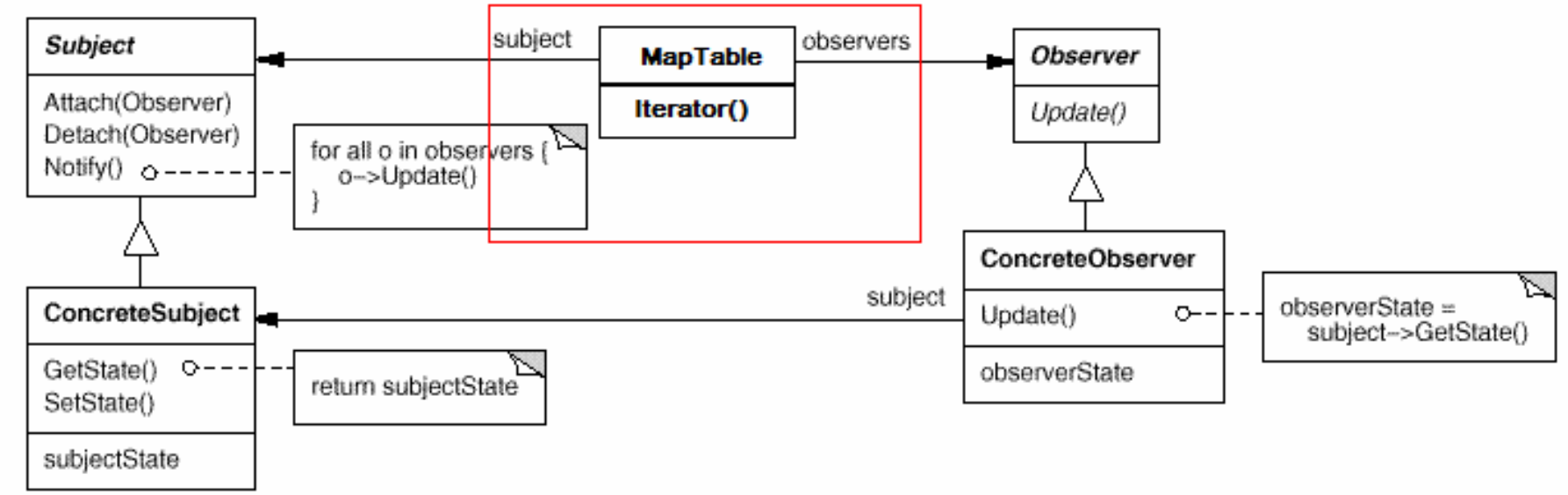
- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Implementation 1: Observing more than one subject

- It might make sense in some situations for an observer to depend on more than one subject.
  - For example, a spreadsheet may depend on more than one data source.
- It's necessary to extend the Update interface in such cases to let the observer know which subject is sending the notification.

Implementation 2: Mapping subjects to their observers.

- Store references to observers explicitly in the subject.
  - Such storage may be too expensive when there are many subjects and few observers.
- Using an associative look-up (hash table) to maintain the subject-to-observer mapping. (trade space for time)
  - This approach increases the cost of accessing the observers.


Subject
Attach(Observer)
Detach(Observer)
Notify()
subject
MapTable
Iterator()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
subject
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()

Implementation 3: Who triggers the update?

- Setting operations on Subject call notify after they change the subject's state automatically.
- Make clients (or one of observers) responsible for calling *Notify* at the right time.
  - Avoiding needless intermediate updates.
  - Clients have an added responsibility to trigger the update, clients might forget to call *Notify*.

Implementation 4: Deleting a subject or a observer

- Deleting a subject should not produce dangling references in its observers. vice versa.
- One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it.

Implementation 5: Making sure Subject state is self-consistent while notification

- It's important to make sure Subject state is self-consistent while calling *Notify*, because observers query the subject for its current state in the course of updating their own state.
  - Change state first, then notify it later;
  - Every changes should be notified;

Implementation 6: Avoiding observer-specific update protocols

- Different observers may interest different *Update,* the amount of information may vary widely.
  - Update without information: the update only be treat as an notification without state (data).
- Push model (Extreme condition): the subject sends observers detailed information about the change, whether they want it or not.
- Pull model (Extreme condition): the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

Push vs Pull

- Pull model
  - Emphasizes the subject's ignorance of its observers
  - May be inefficient, because Observer classes must ascertain what changed without help from the Subject.
- Push model
  - Assumes subjects know something about their observers' needs
  - Make observers less reusable. because Subject classes make assumptions about Observer classes that might not always be true.

Implementation 7: Specifying modifications of interest explicitly

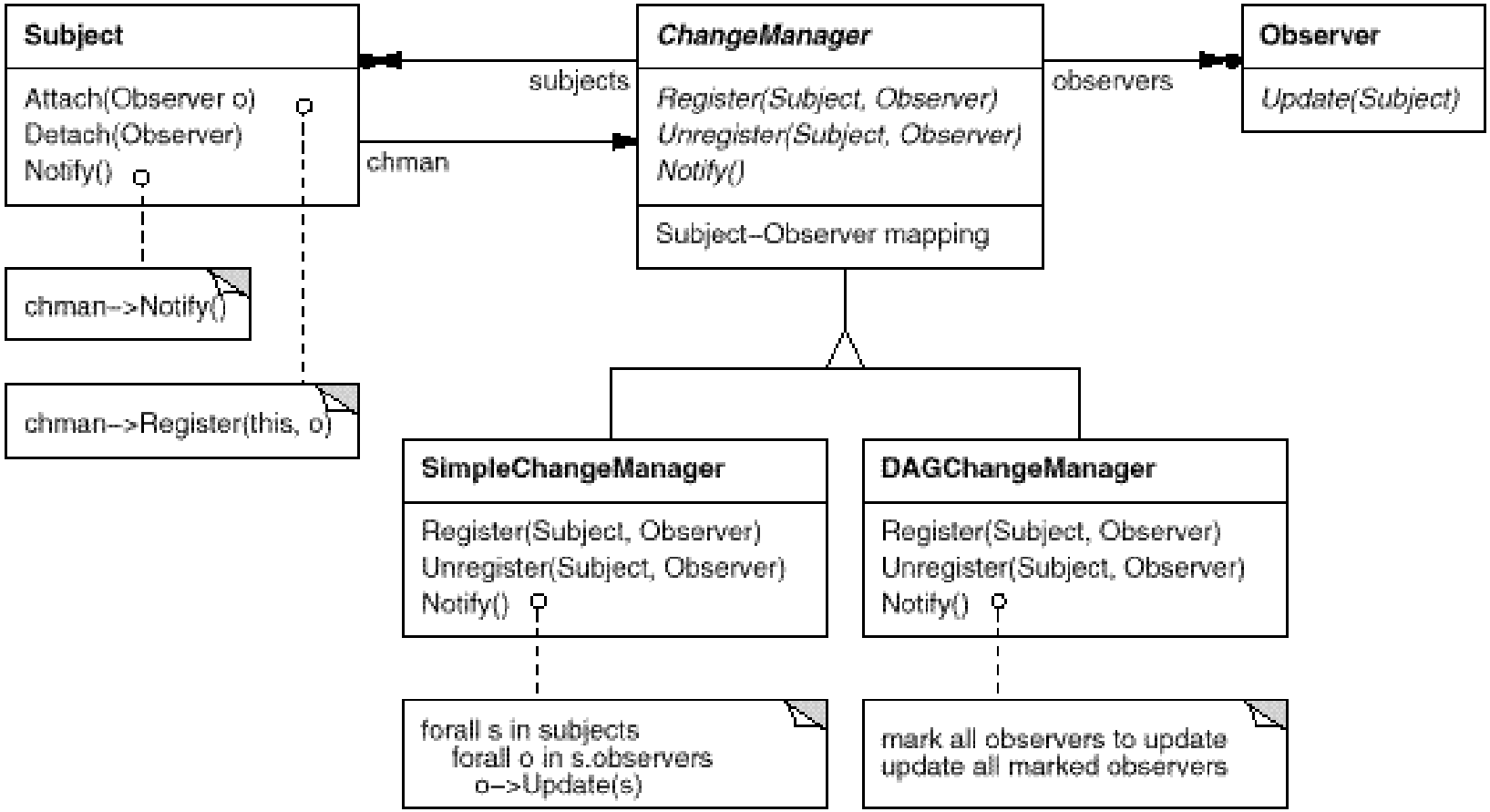
- Improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest.

```
public void attach (Observer observer ,Interest interest);
```

Implementation 8: Encapsulating complex update semantics.

- Dependency relationship between subjects and observers can be particularly complex.
  - If an operation involves changes to several interdependent subjects, you might have to ensure that their observers are notified only after *all* the subjects have been modified to avoid notifying observers more than once.
- An object that maintains these relationships might be required.

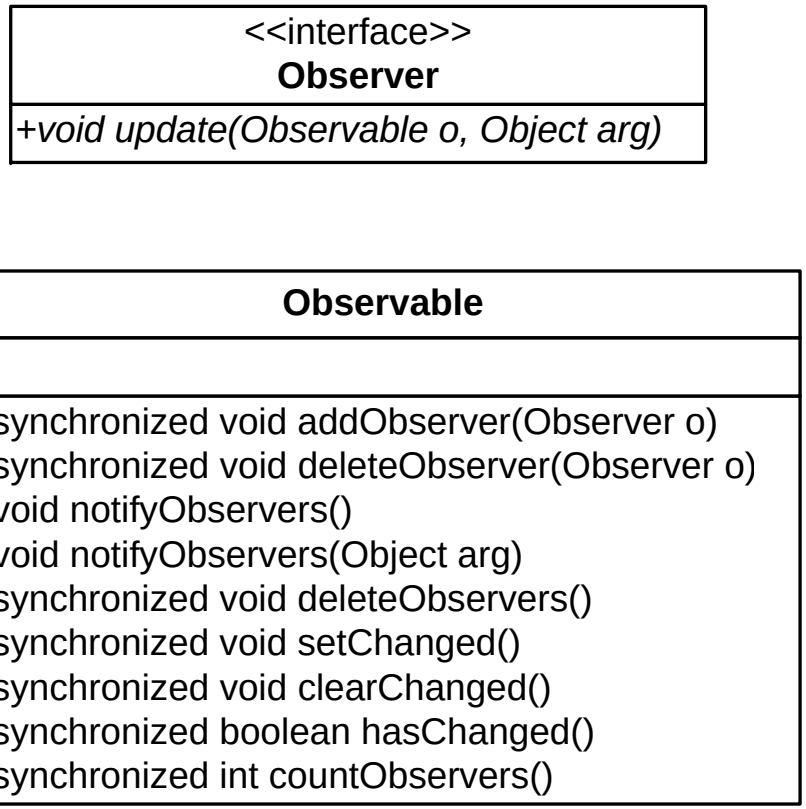
ChangeManager



ChangeManager

- It maps a subject to its observers and provides an interface to maintain this mapping. This eliminates the need for subjects to maintain references to their observers and vice versa.
- It defines a particular update strategy.
- It updates all dependent observers at the request of a subject.
- ChangeManager is an instance of the Mediator pattern

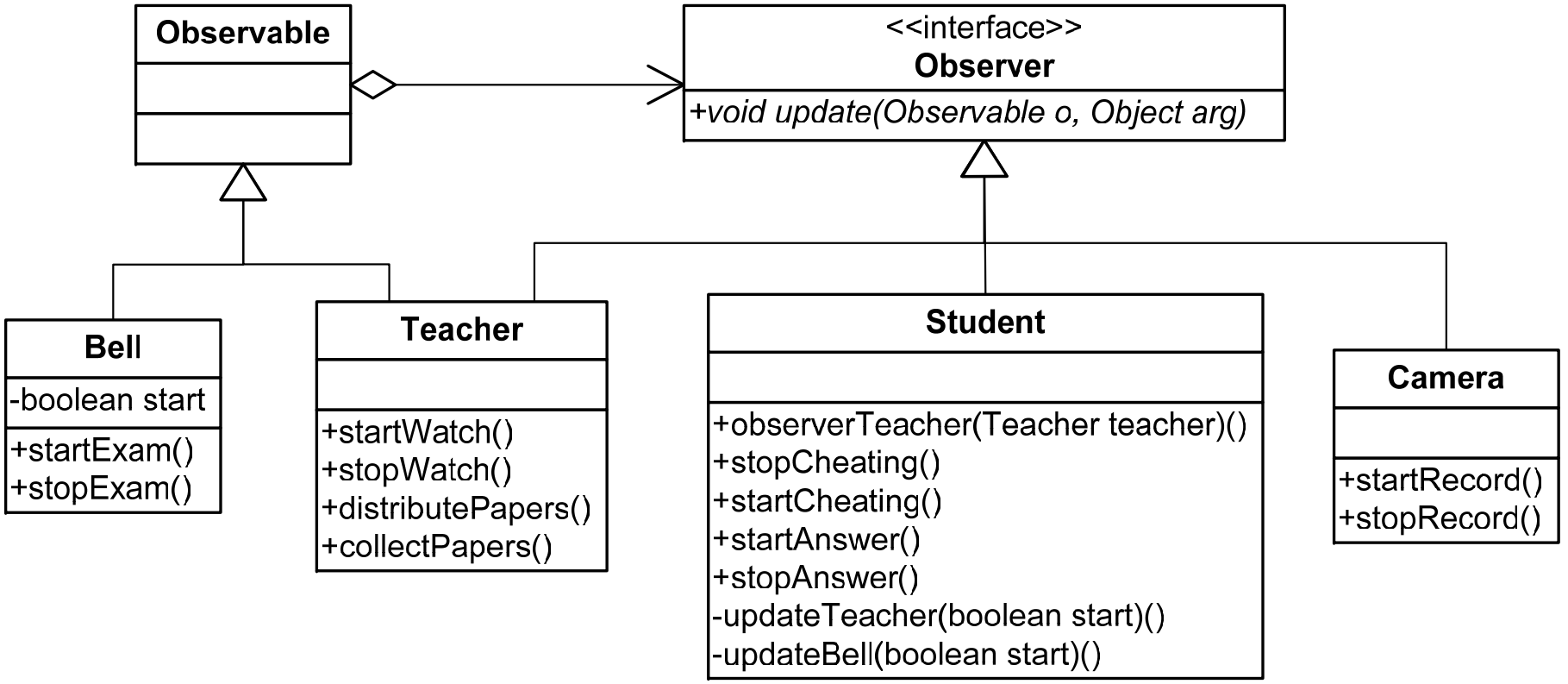
Example: java.util.Observer java.util.Observable



```
public class Observable {
    private boolean changed = false;
    private Vector obs;
    public Observable() {
        obs = new Vector();
    }
    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }
    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }
    public synchronized void deleteObservers() {
        obs.removeAllElements();
    }
    protected synchronized void setChanged() {
        changed = true;
    }
```

```
    protected synchronized void clearChanged() {
        changed = false;
    }
    public synchronized boolean hasChanged() {
        return changed;
    }
    public synchronized int countObservers() {
        return obs.size();
    }
    public void notifyObservers() {
        notifyObservers(null);
    }
    public void notifyObservers(Object arg) {
        Object[] arrLocal;
        synchronized (this) {
            if (!changed){
                return;
            }
            arrLocal = obs.toArray();
            clearChanged();
        }
        for (int i = arrLocal.length - 1; i >= 0; i--)
            ((Observer) arrLocal[i]).update(this, arg);
    }
}
```

Example using Observer and Observable



```
class Bell extends Observable {
    public void startExam() {
        System.out.println(this + ": start exam.");
        this.setChanged();
        this.notifyObservers(new Boolean(true));
    }
    public void stopExam() {
        System.out.println(this + ": stop exam.");
        this.setChanged();
        this.notifyObservers(new Boolean(false));
    }
    public String toString() {
        return "Bell";
    }
}
```

```
class Camera implements Observer {
    private String id;
    private Bell bell;
    public Camera(String id, Bell bell) {
        this.id = id;
        this.bell = bell;
        this.bell.addObserver(this);
        System.out.println(this + ": attach to " + bell + ".");
    }
    public void startRecord() {
        System.out.println(this + ": start record.");
    }
    public void stopRecord() {
        System.out.println(this + ": stop record.");
        this.bell.deleteObserver(this);
        System.out.println(this + ": detach from " + bell + ".");
        this.bell = null;
    }
    public void update(Observable o, Object arg) {
        boolean start = ((Boolean) arg).booleanValue();
        if (o == this.bell) {
            if (start) {
                startRecord();
            } else {
                stopRecord();
            }
        }
    }
    public String toString() {
        return id;
    }
}
```

```
class Teacher extends Observable implements Observer {
    private String id;
    private Bell bell;

    public Teacher(String id, Bell bell) {
        this.id = id;
        this.bell = bell;
        this.bell.addObserver(this);
        System.out.println(this + ": attach to " + bell + ".");
    }
    public void startWatch() {
        System.out.println(this + ": start watch.");
        this.setChanged();
        this.notifyObservers(new Boolean(true));
    }
    public void stopWatch() {
        System.out.println(this + ": stop watch.");
        this.setChanged();
        this.notifyObservers(new Boolean(false));
    }
```

```
    public void distributePapers() {
        System.out.println(this + ": distribute papers.");
    }
    public void collectPapers() {
        System.out.println(this + ": collect papers.");
        this.bell.deleteObserver(this);
        System.out.println(this + ": detach from " + bell + ".");
        this.bell = null;
    }
    public void update(Observable o, Object arg) {
        boolean start = ((Boolean) arg).booleanValue();
        if (o == this.bell) {
            if (start) {
                distributePapers();
            } else {
                collectPapers();
            }
        }
    }
    public String toString() {
        return id;
    }
}
```

```
class Student implements Observer {
    private List<Teacher> teachers;
    private Bell bell;
    private String id;

    public Student(String id, Bell bell) {
        this.teachers = new ArrayList<Teacher>();
        this.id = id;
        this.bell = bell;
        this.bell.addObserver(this);
        System.out.println(this + ": attach to " + bell + ".");
    }
    public Student observerTeacher(Teacher teacher) {
        teacher.addObserver(this);
        teachers.add(teacher);
        System.out.println(this + ": attach to " + teacher + ".");
        return this;
    }
    public void stopCheating() {
        System.out.println(this + ": stop Cheating.");
    }
    public void startCheating() {
        System.out.println(this + ": start Cheating.");
    }
    public void startAnswer() {
        System.out.println(this + ": start Answer.");
    }
```

```
    public void stopAnswer() {
        System.out.println(this + ": stop Answer.");
        this.bell.deleteObserver(this);
        System.out.println(this + ": detach from " + bell + ".");
        this.bell = null;
        for (Teacher teacher : teachers) {
            teacher.deleteObserver(this);
            System.out.println(this + ": detach from " + teacher + ".");
        }
        teachers.clear();
    }
    public void update(Observable o, Object arg) {
        boolean start = ((Boolean) arg).booleanValue();
        if (this.teachers.contains(o)) {
            updateTeacher(start);
        } else if (o == this.bell) {
            updateBell(start);
        }
    }
    private void updateTeacher(boolean start) {
        if (start) {
            stopCheating();
        } else {
            startCheating();
        }
    }
    private void updateBell(boolean start) {
        if (start) {
            startAnswer();
        } else {
            stopAnswer();
        }
    }
    public String toString() {
        return id;
    }
}
```

```
public void exam() {
    Bell bell = new Bell();

    Teacher teacherA = new Teacher("Teacher A", bell);
    Teacher teacherB = new Teacher("Teacher B", bell);

    new Camera("Camera A", bell);
    new Camera("Camera B", bell);

    new Student("Student 1", bell).observerTeacher(teacherA);
    new Student("Student 2", bell).observerTeacher(teacherB);
    new Student("Student 3", bell).observerTeacher(teacherA)
            .observerTeacher(teacherB);

    bell.startExam();
    teacherA.startWatch();
    teacherB.startWatch();
    teacherB.stopWatch();
    teacherA.stopWatch();
    bell.stopExam();
}
```

- Teacher A: attach to Bell.
- Teacher B: attach to Bell.
- Camera A: attach to Bell.
- Camera B: attach to Bell.
- Student 1: attach to Bell.
- Student 1: attach to Teacher A.
- Student 2: attach to Bell.
- Student 2: attach to Teacher B.
- Student 3: attach to Bell.
- Student 3: attach to Teacher A.
- Student 3: attach to Teacher B.
- Bell: start exam.
- Student 3: start Answer.
- Student 2: start Answer.
- Student 1: start Answer.
- Camera B: start record.
- Camera A: start record.
- Teacher B: distribute papers.
- Teacher A: distribute papers.
- Teacher A: start watch.
- Student 3: stop Cheating.
- Student 1: stop Cheating.
- Teacher B: start watch.
- Student 3: stop Cheating.
- Student 2: stop Cheating.
- Teacher B: stop watch.
- Student 3: start Cheating.
- Student 2: start Cheating.
- Teacher A: stop watch.
- Student 3: start Cheating.
- Student 1: start Cheating.
- Bell: stop exam.
- Student 3: stop Answer.
- Student 3: detach from Bell.
- Student 3: detach from Teacher A.
- Student 3: detach from Teacher B.
- Student 2: stop Answer.
- Student 2: detach from Bell.
- Student 2: detach from Teacher B.
- Student 1: stop Answer.
- Student 1: detach from Bell.
- Student 1: detach from Teacher A.
- Camera B: stop record.
- Camera B: detach from Bell.
- Camera A: stop record.
- Camera A: detach from Bell.
- Teacher B: collect papers.
- Teacher B: detach from Bell.
- Teacher A: collect papers.
- Teacher A: detach from Bell.

Extension: Delegation Event Model (DEM)

- Event : Event are encapsulated in a class hierarchy rooted at java.util.EventObject.
  - An event is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated.

Extension: Delegation Event Model (DEM)

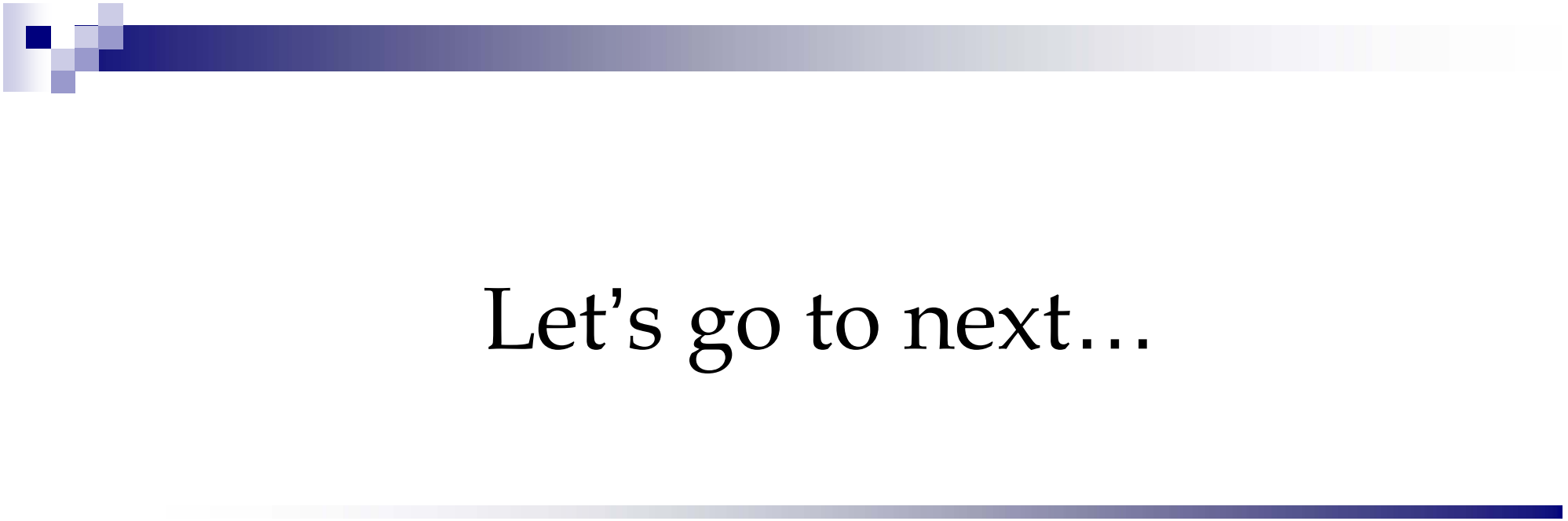
- Listener : A Listener is an object that implements a specific EventListener interface extended from the generic java.util.EventListener.
  - An EventListener interface defines one or more methods which are to be invoked by the event source in response to each specific event type handled by the interface.

Extension: Delegation Event Model (DEM)

- Source: An Event Source is an object which originates or "fires" events.
  - The source defines the set of events it emits by providing a set of set<EventType>Listener (for single-cast) and/or add<EventType>Listener (for mult-cast) methods which are used to register specific listeners for those events.

Extension 1: Delegation Event Model (DEM)

- DEM in AWT
- DEM in Servlet


Let’s go to next…