Observer Design Pattern

A Lecture for KMP's COMP 401 Class

Aaron Smith

October 25, 2018

Adapted from Ketan Mayer-Patel's Lecture Slides

What are design patterns?

Design patterns are techniques

for **organizing your code**

that are often used in the real world

Common Design Patterns

- Iterator
- Factory
- Singleton
- Decorator
- Observer / Observable
- Model-View
- Model-View-Controller
- And more…

Review: Decorator

Goal

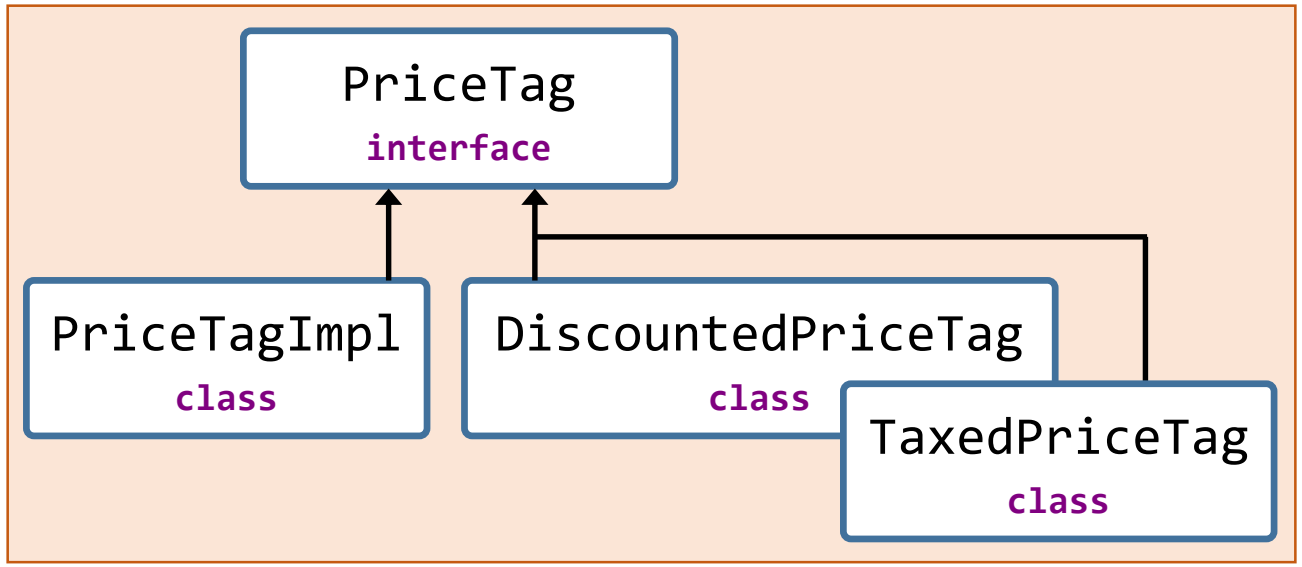
- Want to **add optional features** (decorations) to objects of a specific interface
- The added features **do not depend** on interface implementation details

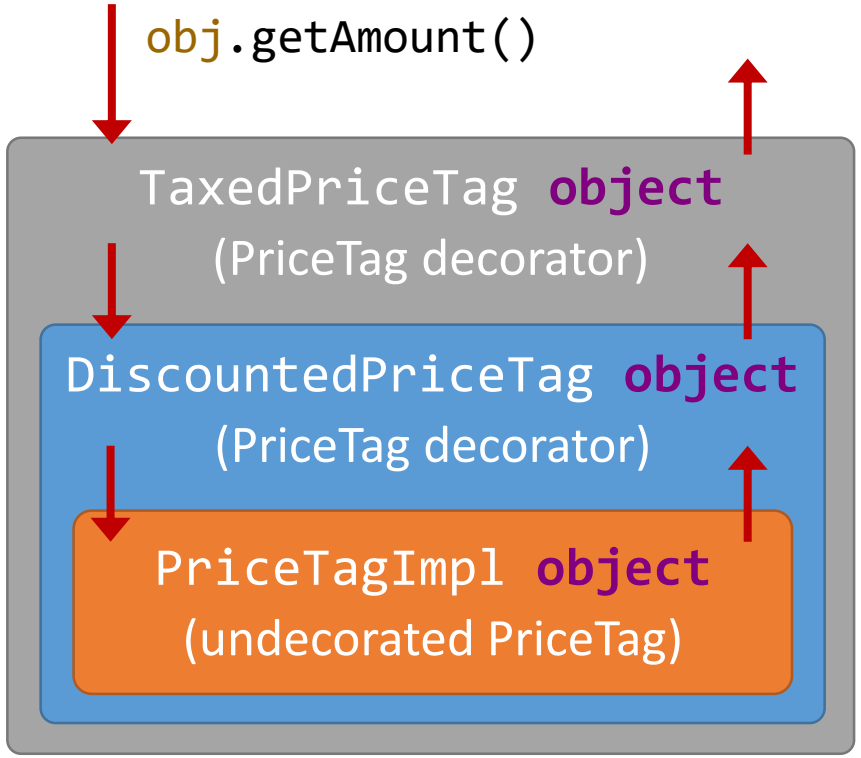
Technique

- Wrap (encapsulate) an existing instance of the interface inside a decorator class, which also implements the interface
- Forward (delegate) interface methods to the wrapped object
- Add features before or after delegation

Review: Decorator Example

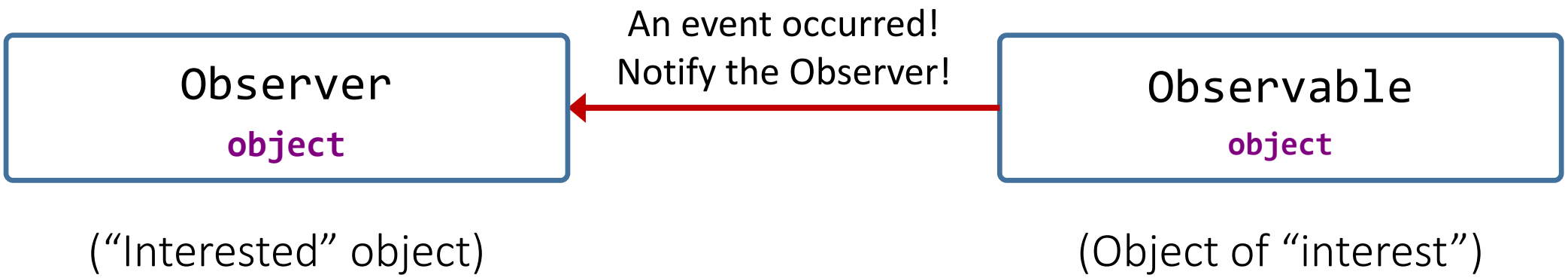
```
public interface PriceTag {
  double getAmount();
  void   setAmount(double amount);
}
```





Observer Design Pattern

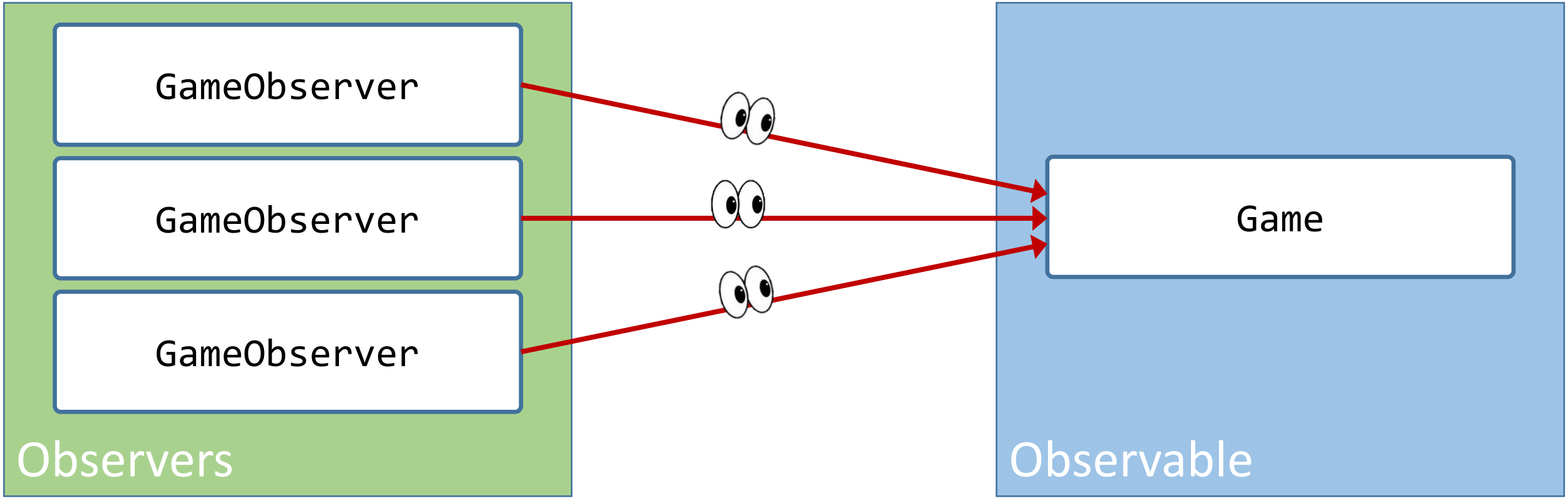
Situation: *Something* happens inside one object that another object must respond to.



Use Cases

- User Interfaces
  - Web programming with JavaScript (COMP 426)
  - Graphical User Interfaces (GUIs) like AWT and Swing
- Asynchronous Programming
  - Also known as “event-based” programming
  - Events may be induced by hardware
- Building block for other design patterns
  - Model-View
  - Model-View-Controller

Registering as an Observer



The `GameObserver` objects **register** as observers of the `SportsGame` object, waiting for an **event** to occur

Q: What *is* an event, anyway?

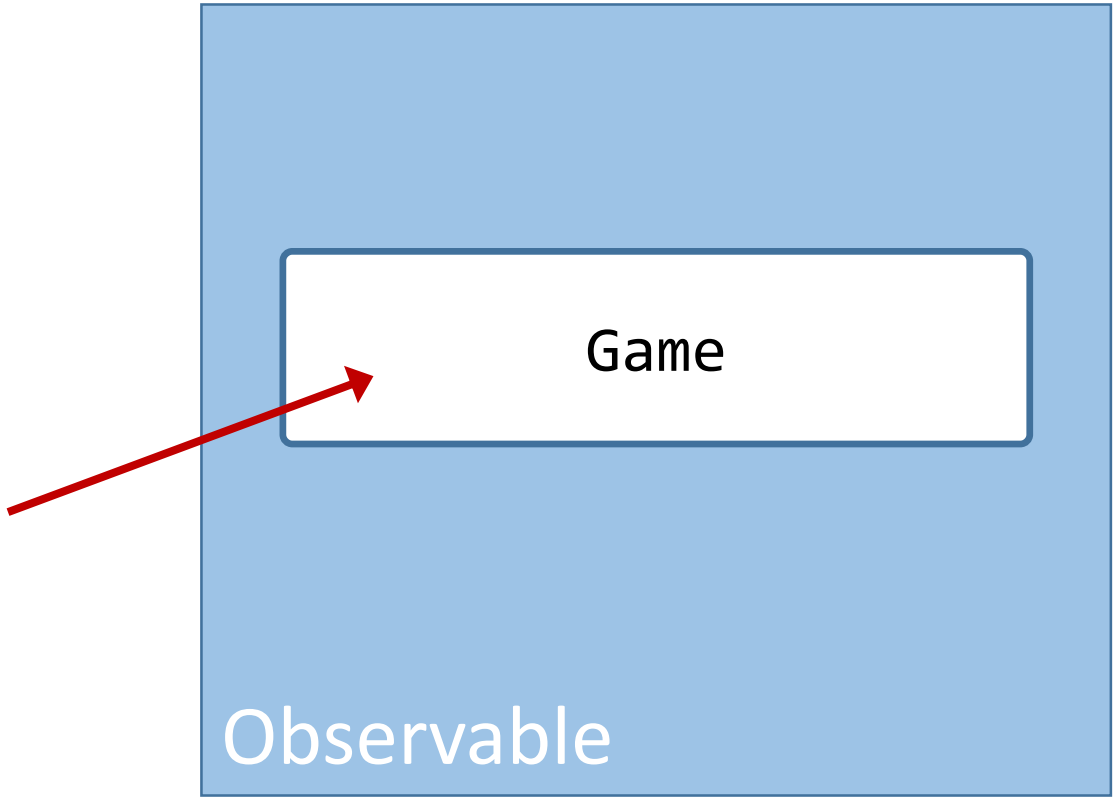
A: A <u>state change</u> inside an **observable** object

Events may be induced by:

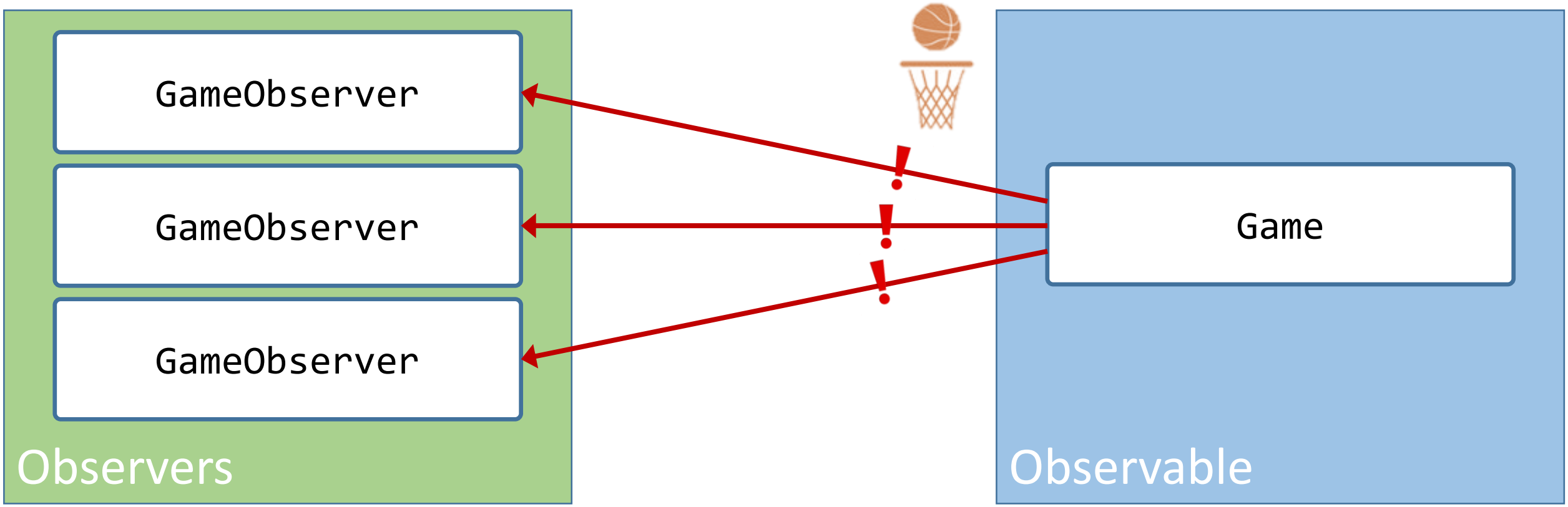
- User interaction with an on-screen UI component
- Hardware (sensors, buttons, etc.)
- Changing data

One commonality:

Events happen <u>inside</u> the **observable** object



Notifying When an Event Occurs



The `Game` object **notifies** all registered `GameObserver` objects whenever an **event** occurs

Execution Sequence

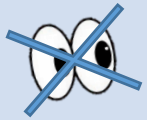
Observers register with the observable object

An event occurs inside the observable object

The observable object notifies all observers

The observers might react to the event

Observers can deregister to stop observing



Glossary

Observer, Listener	• An object interested in an event
Observable, Subject, Dispatcher	• The object that causes the event to happen
Event, Action	• A state change that may cause other objects to react
Dispatch, **Update**, **Notify**	• The act of letting observers know that an event occurred
Register, Listen	• The act of observing an event
Deregister	• The act of no longer observing an event

Job of an **Observable** Object

1. Notify all observer objects when an event occurs

```
void notifyObservers();
```

2. Store a list of all observer objects

```
private List<Observer> observers;
```

3. Allow observer objects to register and deregister themselves

```
void addObserver(Observer o);
void deleteObserver(Observer o);
```

Basic **Observable** Class

```java
public class Observable {
  private List<Observer> observers;

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

  void deleteObserver(Observer o) {
    observers.remove(o);
  }

  void notifyObservers() {
    for (Observer o : observers) {
      o.update();
    }
  }
}
```

Encapsulates a `List` of `Observer` objects

Allows `Observer` objects to be **added** or **removed**

Calls a method on all `Observers`, allowing them to respond

Basic **Observable** Class

```
public class Observable {
  private List<Observer> observers;

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

  void deleteObserver(Observer o) {
    observers.remove(o);
  }

  void notifyObservers() {
    for (Observer o : observers) {
       o.update();
    }
  }
}
```

`notifyObservers()` should be called every time an event occurs

Job of an Observer Object

1. Respond to events

```
void update();
```

2. Register and deregister to receive updates about events

```
observable.addObserver(observer);
observable.deleteObserver(observer);
```

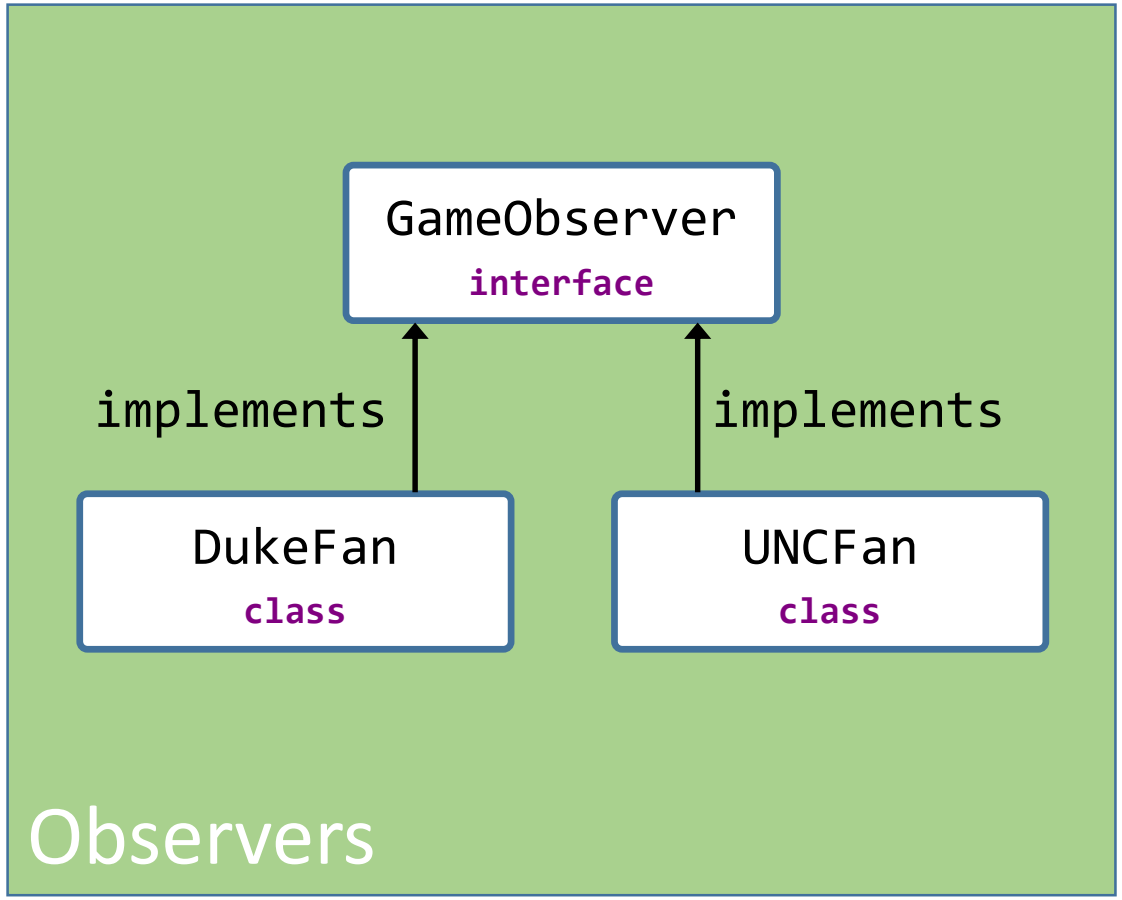
Basic Observer Interface

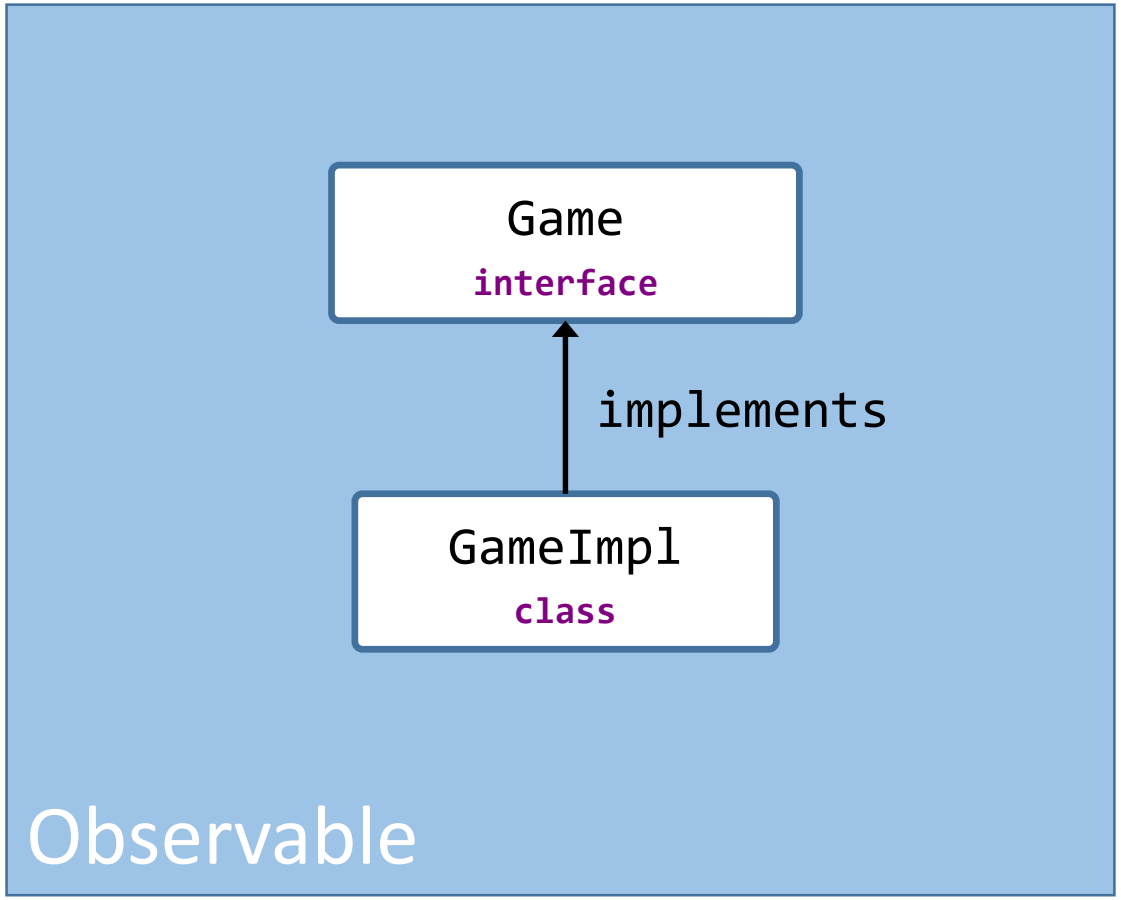
```
public interface Observer {
  void update();
}
```

`update()` is called by the `Observable` object to notify that an event occurred

The code inside `update()` specifies how the `Observer` should respond to an event

Example


GameObserver
interface
implements
implements
DukeFan
class
UNCFan
class
Observers


Game
interface
implements
GameImpl
class
Observable

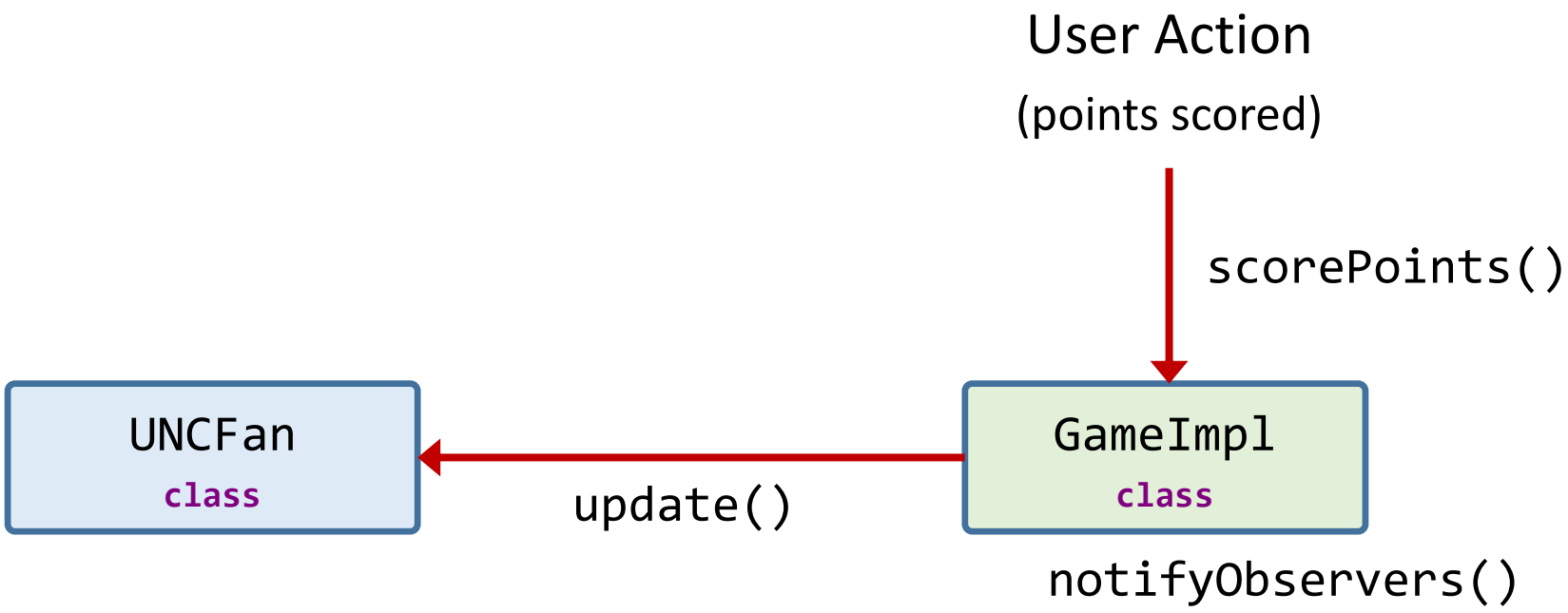
Code Version 1:
Basic Observer

Game.java
GameImpl.java

GameObserver.java
DukeFan.java
UNCFan.java

Main.java

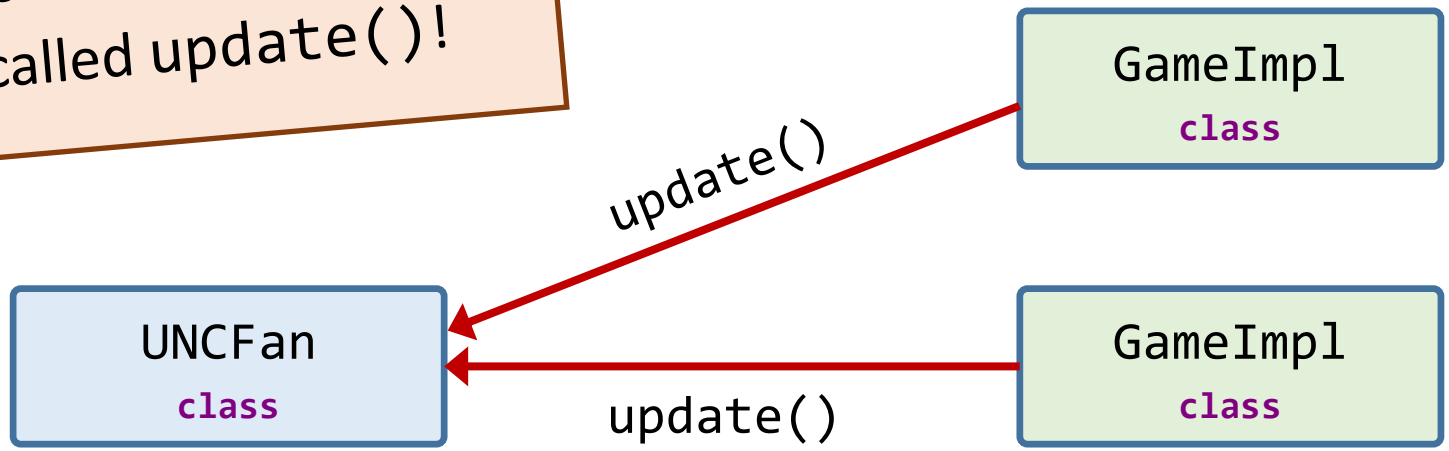
Limitation of First Example



What if a fan (observer) wants to be updated about more than one game (observable) at a time?

Multiple Games

Need a way to determine which `GameImpl` called `update()`!



What if a fan (**observer**) wants to be updated about more than one game (**observable**) at a time?

Supporting Multiple Games

```java
public interface Observer {
  void update(Observable o);
}
```

```java
public class Observable {

  // ...

  void notifyObservers() {
    for (Observer o : observers) {
      o.update(this);
    }
  }
}
```

An `Observable` is passed into `update()`

Supporting Multiple Games

```java
public interface GameObserver {
  void update(Game g);
}
```

```java
public class GameImpl implements Game {

  // ...

  void notifyObservers() {
    for (GameObserver o : observers) {
      o.update(this);
    }
  }
}
```

A Game is passed into update()

Code Version 2: Multiple Games

GameImpl.java

DukeFan.java, UNCFan.java

Main.java

Passing Context to the Observer

```
public interface Observer {
  void update(Observable o, Info i);
}
```

```
public class Observable {

  // ...

  void notifyObservers(Info i) {
    for (Observer o : observers) {
      o.update(this, i);
    }
  }
}
```

Contextual information may now be passed into `update()`

Passing Context to the Observer

```
public interface Observer {
  void update(Observable o, Info i);
}
```

Observable that produced the event

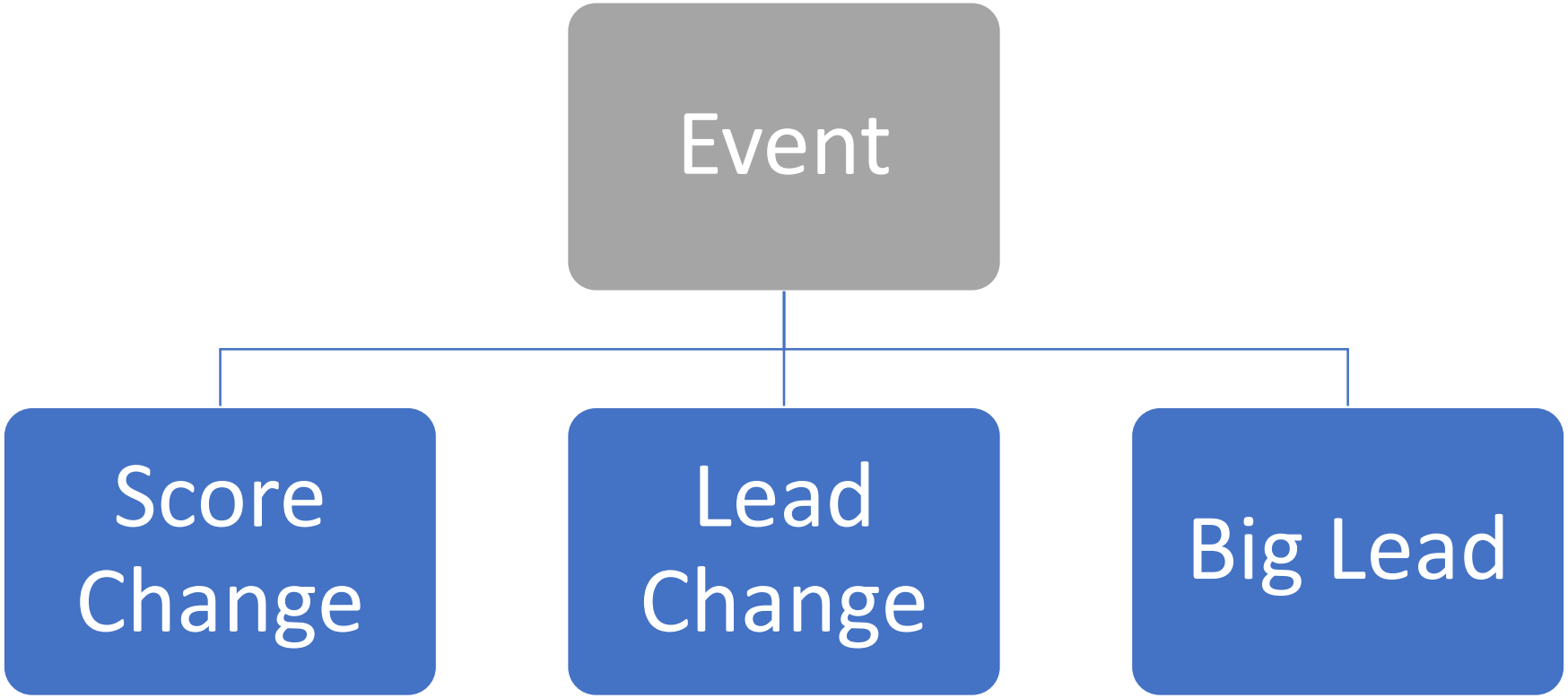
Event circumstances

```
public class Observable {

  // ...

  void notifyObservers(Info i) {
    for (Observer o : observers) {
      o.update(this, i);
    }
  }
}
```

Different Event Types



Passing Context to the Observer

```java
public interface Observer {
  void update(Observable o, Info i);
}
```

Package event info inside this object

```java
public class Observable {

  // ...

  void notifyObservers(Info i) {
    for (Observer o : observers) {
      o.update(this, i);
    }
  }
}
```

GameEvent Object

```java
public interface GameObserver {
  void update(Game o, GameEvent e);
}
```

```java
public interface GameEvent {
  String getType();
  String getWhoScored();
}
```

```java
public class GameImpl {

  // ...

  void notifyObservers(GameEvent e) {
    for (Observer o : observers) {
      o.update(this, e);
    }
  }
}
```

Code Version 3:
Passing Event Information

GameEvent.java, GameEventImpl.java

GameImpl.java

DukeFan.java, UNCFan.java

Can we refactor the Observable code into its own class?

Java tried it!

Java Built-In Observer / Observable

java.util.Observer

```
public interface Observer {
  void update(Observable o, Object arg);
}
```

java.util.Observable

```
public class Observable {
  void addObserver(Observer o);
  void deleteObserver(Observer o);
  void deleteObservers();
  void notifyObservers(Object arg);
}
```

Java ***used to*** provide a skeleton **Observer interface** and **Observable class** in java.util

Java Built-In Observer / Observable

java.util.Observer

```
public interface Observer {
  void update(Observable o, Object arg);
}
```

java.util.Observable

```
public class Observable {
  void addObserver(Observer o);
  void deleteObserver(Observer o);
  void deleteObservers();
  void notifyObservers(Object arg);
}
```

It was deprecated in Java 9

Why?

- Inflexible observable class
- Required unsafe contravariant casts

Aside: Default Implementations In Interfaces

Interfaces may now contain:

1. `static` method implementations
2. `static` named constants
3. `default` method implementations

Since Java 8

4. `private` method implementations

Since Java 9

Note: All `default` method implementations are `public`

Code Example 4: Default Implementations

Point.java