SWEN 262

Engineering of Software Subsystems

Observer Pattern

Invaders From Space™

1. Invaders from Space™ is a game that challenges the player to fight off waves of invading aliens using a spaceship.
    a. *The player uses left and right arrow keys to move their spaceship in the corresponding direction.*
    b. *They use the spacebar to fire a single shot at the encroaching aliens.*
    c. *If at any time the player presses the "ESC" key, the game pauses and they are prompted to quit. Pressing the "y" key will quit the game. Pressing the "ESC" key again will unpause.*
2. The game should be as responsive as possible, and so each time the user presses one of the control keys, the game should immediately respond in some way.

There are a number of ways that the program can check for user input. Some are better than others...

Polling

```
while(true) {
  try {
    Thread.sleep(INTERVAL);
  } catch(InterruptedException e) {}

  switch(lastKeyPressed()) {
    case LEFT_ARROW:
      moveLeft(); break;
    case RIGHT_ARROW:
      moveRight(); break;
    case SPACE_BAR:
      fire(); break;
    case ESC:
      togglePause(); break;
    case Y:
      checkForQuit(); break;
  }
}
```

One possible technique is ***polling***.

When using polling, execution is suspended for some ***interval***, after which the program checks to see if any significant events have occured.

Once any event(s) have been handled, the program suspends execution again until the next interval elapses.

Q: What are the potential drawbacks of this implementation?

A: What happens if more than one event occurs between intervals? What if no events occur?

Drawbacks of Polling

While sometimes polling is unavoidable, there are many drawbacks.

- *Missed Events - if the polling interval is too long, it is possible that more than one event will occur between intervals.*
- *Wasted Cycles - if the polling interval is too short, processing time is wasted when the thread wakes up to check for events and none have occurred.*
- *Duplicate Events - it is possible that the same event may be handled more than once, e.g. a single press of the left arrow moves the ship 2 or more times.*

Let’s take a look at an alternative to polling.

Observing a Subject

Begin by defining an `interface` to be implemented by any ***observers*** that should be notified whenever a key is pressed on the keyboard.

```
public interface KeyListener {
  public void keyPressed(KeyEvent event);
}
```

In this example, a `KeyEvent` class is used to encapsulate the details about the event (e.g. which key was pressed, how many times, etc.).

Create a second `interface` to be implemented by any ***subject*** on which a key may be pressed and that should notify any registered ***observers***.

```
public interface Component {
  void register(KeyListener listener);
  void deregister(KeyListener listener);
  void notify(KeyEvent event);
}
```

Implement the Subject Interface

```
public class PlayArea implements Component {
  private List<KeyListener> listeners =
    new ArrayList<>();

  void register(KeyListener listener) {
    listeners.add(listener);
  }

  void deregister(KeyListener listener) {
    listeners.remove(listener);
  }

  void notify(KeyEvent event) {
    for(KeyListener listener : listeners) {
      listener.notify(event);
    }
  }
}
```

Implement a ***concrete subject*** so that interested ***observers*** can register to be notified when an event occurs.

The ***observers*** are typically kept in a data structure such as a list or a set.

When an event does occur, e.g. the user presses a key on the subject, the ***concrete subject*** should iterate over the ***observers*** and notify them.

The ***concrete subject*** should call this method to notify its ***observers*** *immediately* any time a key is pressed.

Implement the Observer Interface

```
public class PlayerActionHandler
                    implements KeyListener {
  public void keyPressed(KeyEvent event) {
    switch(event.getKeyCode()) {
      case LEFT_ARROW:
        moveLeft(); break;
      case RIGHT_ARROW:
        moveRight(); break;
      // and so on...
    }
  }
}
```

Implement a ***concrete observer*** that includes the code that should be executed each and every time a key is pressed.

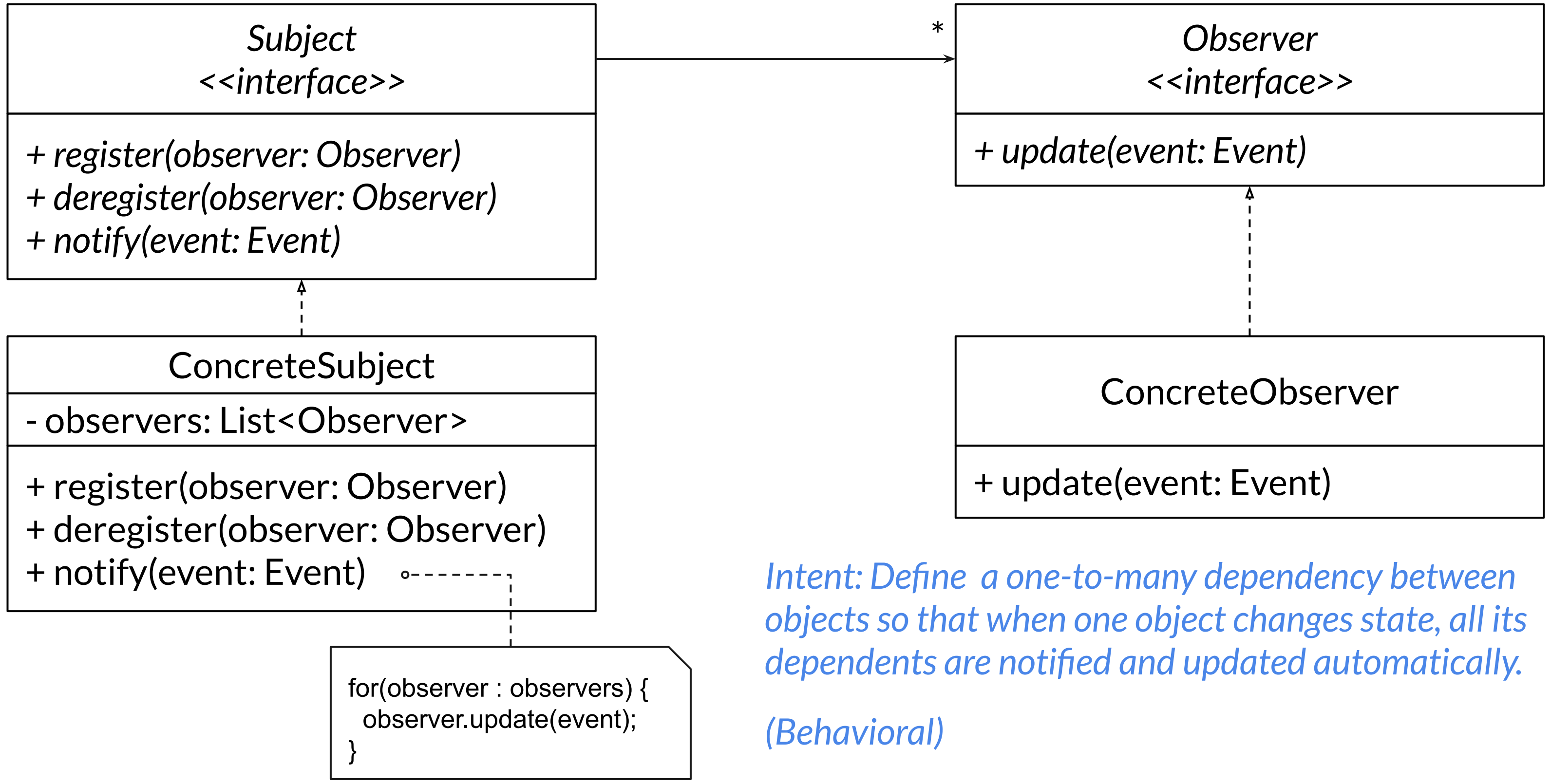
The `KeyEvent` encapsulates the information about the event (i.e. which key was pressed). The ***concrete subject*** will call this method on any registered ***observer*** exactly once each time a key is pressed.

```
KeyListener listener =
  new PlayerActionHandler();

Component playArea = new PlayArea();
playArea.register(listener);
```

Finally, create an instance of the ***concrete observer*** and register it with the ***concrete subject***.

The ***concrete subject*** will notify the ***concrete observer*** *immediately* whenever a key is pressed. No polling needed!

GoF Observer Structure Diagram



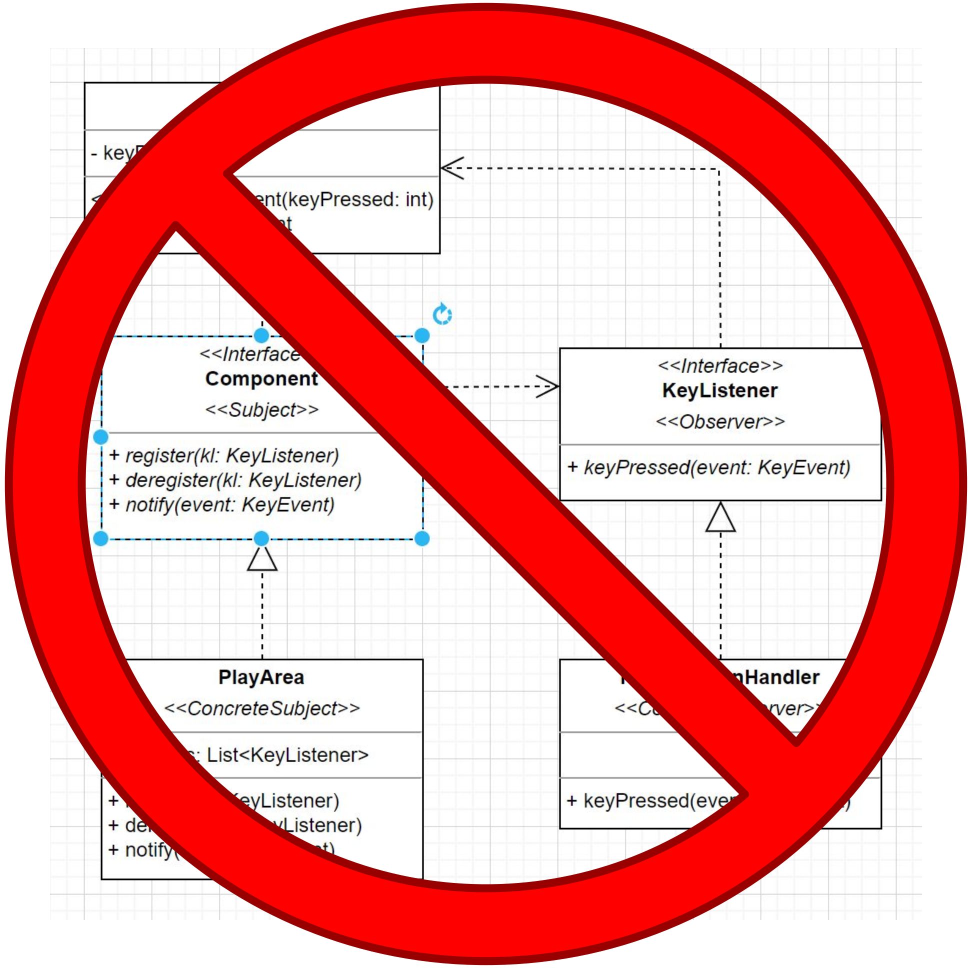
```
for(observer : observers) {
  observer.update(event);
}
```

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

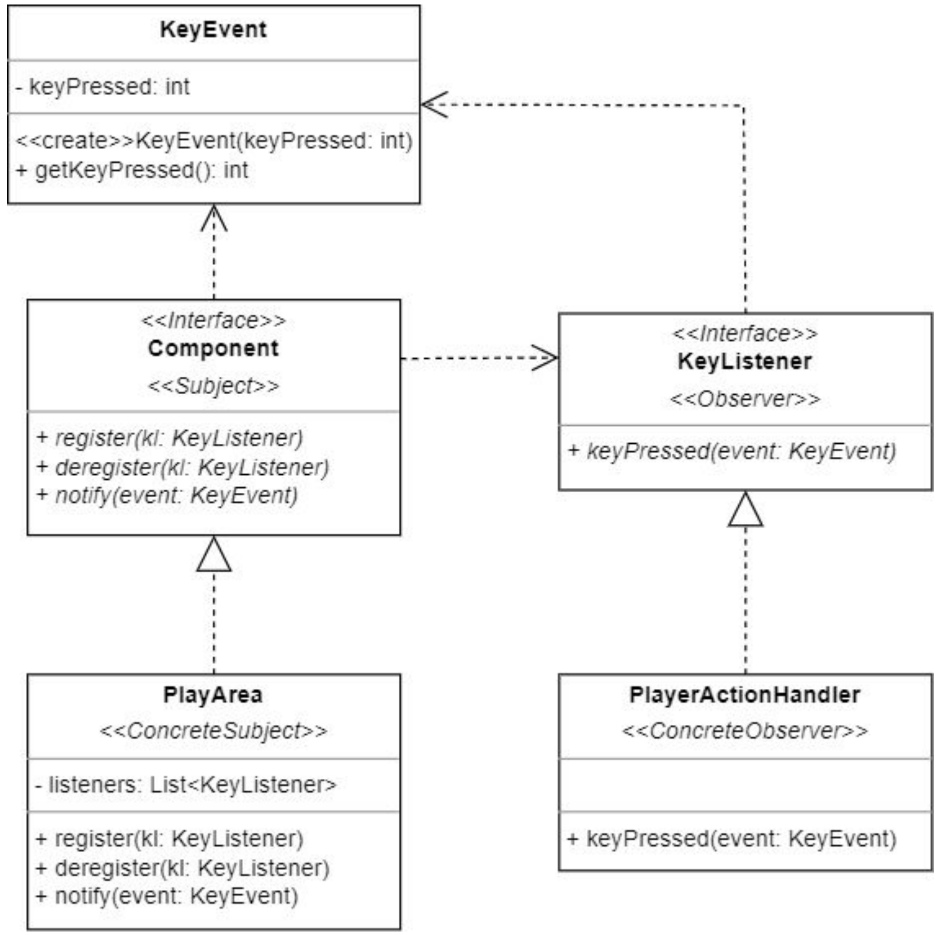
(Behavioral)

Diagrams should be drawn with a professional drawing tool like draw.io or Lucidchart and saved as an image.

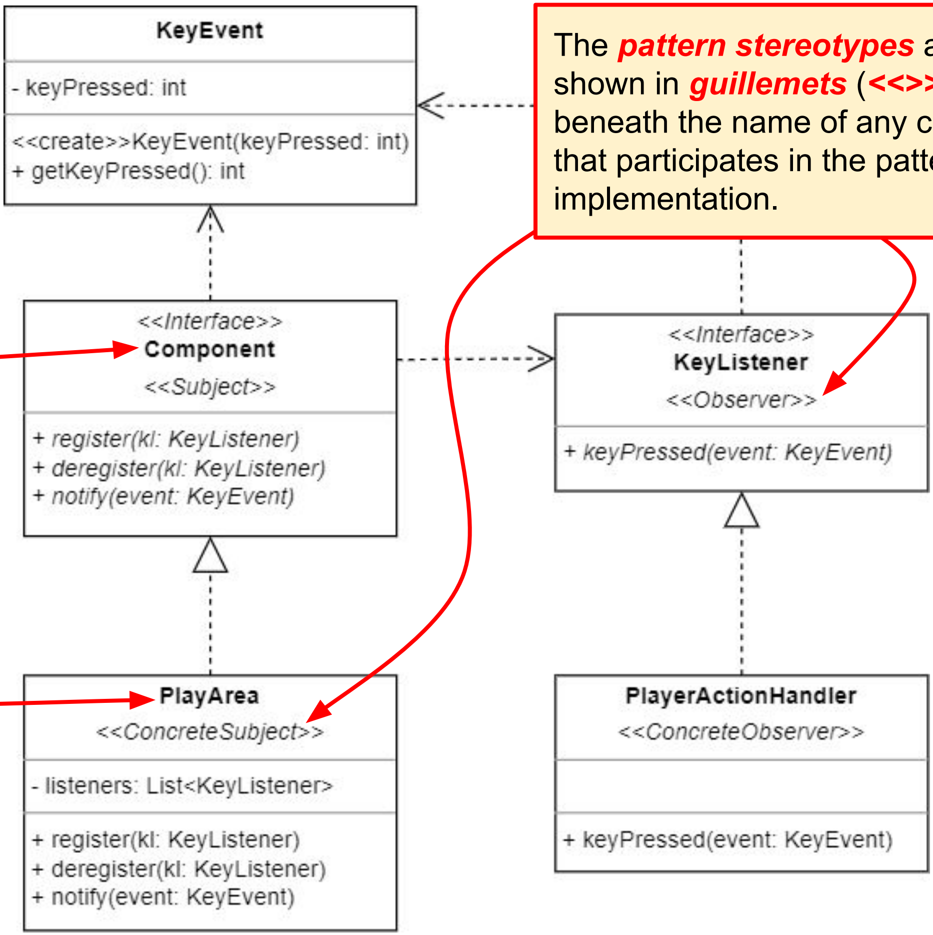
Screenshots are sloppy, lazy, and look unprofessional.



UML Class Diagram


KeyEvent
- keyPressed: int
<<create>>KeyEvent(keyPressed: int)
+ getKeyPressed(): int
<<Interface>>
Component
<<Subject>>
+ register(kl: KeyListener)
+ deregister(kl: KeyListener)
+ notify(event: KeyEvent)
<<Interface>>
KeyListener
<<Observer>>
+ keyPressed(event: KeyEvent)
PlayArea
<<ConcreteSubject>>
- listeners: List<KeyListener>
+ register(kl: KeyListener)
+ deregister(kl: KeyListener)
+ notify(event: KeyEvent)
PlayerActionHandler
<<ConcreteObserver>>
+ keyPressed(event: KeyEvent)

UML Class Diagram


KeyEvent
- keyPressed: int
<<create>>KeyEvent(keyPressed: int)
+ getKeyPressed(): int
<<Interface>>
Component
<<Subject>>
+ register(kl: KeyListener)
+ deregister(kl: KeyListener)
+ notify(event: KeyEvent)
<<Interface>>
KeyListener
<<Observer>>
+ keyPressed(event: KeyEvent)
PlayArea
<<ConcreteSubject>>
- listeners: List<KeyListener>
+ register(kl: KeyListener)
+ deregister(kl: KeyListener)
+ notify(event: KeyEvent)
PlayerActionHandler
<<ConcreteObserver>>
+ keyPressed(event: KeyEvent)

The ***pattern stereotypes*** are shown in ***guillemets*** (**<<>>**) beneath the name of any class that participates in the pattern implementation.

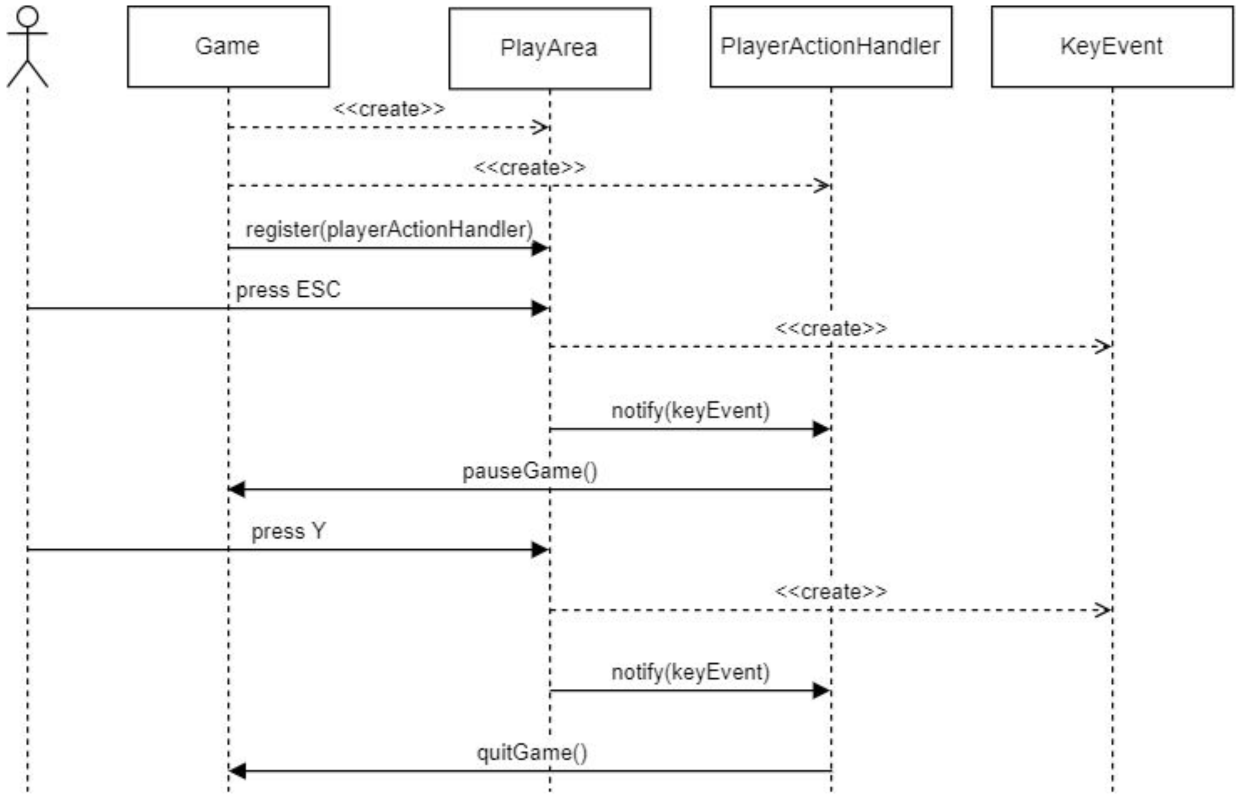
Note that each class has a ***context specific*** name appropriate for the game.

GoF Pattern Card

Name: *Player Action Subsystem*		GoF Pattern: *Observer*
Participants		
Class	Role in Pattern	Participant's Contribution in the context of the application
Component	*Subject*	*Defines the interface for any class that can be observed for key presses. This is most likely to be a GUI component of some kind, like a panel.*
PlayArea	*ConcreteSubject*	*The GUI component that displays the play area including space ship, aliens, etc. This component will have focus during play, and so will generate an event at any time that the user presses a key while the game is running.*
KeyListener	*Observer*	*The interface for any class that should be notified when the user presses a key on an observed subject. There may be several such listeners in the game.*
PlayerActionHandler	*ConcreteObserver*	*Interprets key presses from the player into actions in the game, i.e. left arrow moves the space ship left, right arrow moves right, and so on.*
Deviations from the standard pattern: *None*		
Requirements being covered: *1a. Ship movement, 1b. Firing weapons, 1c. Pause/quit, 2. Responsive to player input.*		

Sequence Diagram

A sequence diagram illustrating the "quit game" feature described in the requirements.



Push vs. Pull Notifications

The examples shown here use ***push notifications***.

- *The information about the change in the state of the subject is encapsulated as an event.*
- *The event is sent to the registered observers.*

Alternatively, ***pull notifications*** may be used instead.

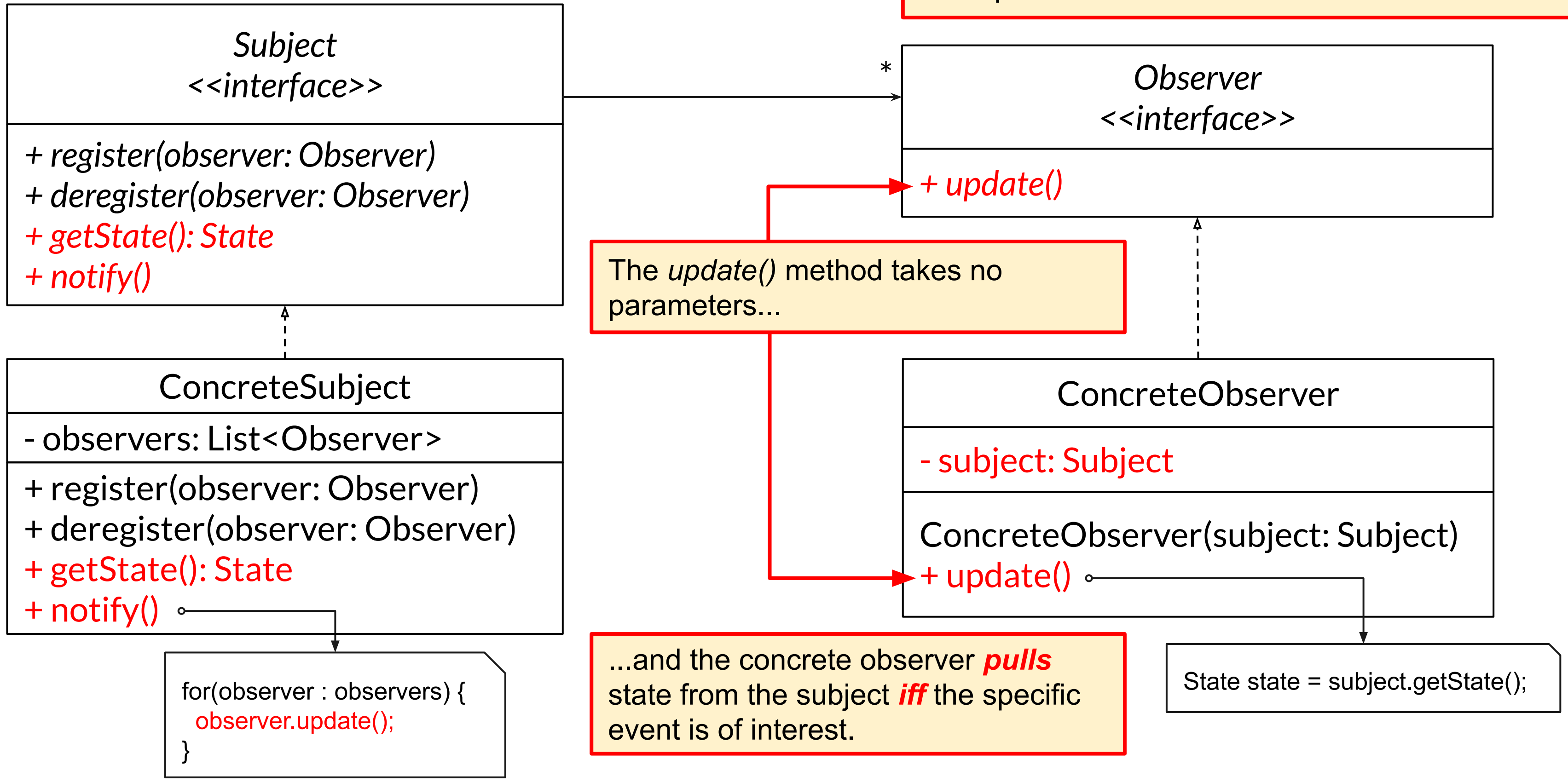
- *The subject notifies the observers that a specific event has occurred, but does not send the details about the event to the observers.*
- *If the observer is interested in the specific type of event that has occurred, it will request the details from the subject.*
- *This is useful if there are many different types of events that may occur, and individual observers may only be interested in a subset of the events.*

```java
public class ConcreteObserver
                  implements Observer {

  private Subject subject;

  public ConcreteObserver(Subject subject) {
    this.subject = subject;
  }

  public void update() {
    State state = subject.getState();
  }
}
```

In this alternative implementation, the update method takes no parameters and the concrete observer needs a reference to the subject.

Observer (Pull Notifications)

In this alternative, the observers are notified that *something* changed, but not the specifics. The intent remains the same.


Subject
<<interface>>
+ register(observer: Observer)
+ deregister(observer: Observer)
+ getState(): State
+ notify()
*
Observer
<<interface>>
+ update()
ConcreteSubject
- observers: List<Observer>
+ register(observer: Observer)
+ deregister(observer: Observer)
+ getState(): State
+ notify()
ConcreteObserver
- subject: Subject
ConcreteObserver(subject: Subject)
+ update()

```
for(observer : observers) {
  observer.update();
}
```

```
State state = subject.getState();
```

The *update()* method takes no parameters...

...and the concrete observer ***pulls*** state from the subject ***iff*** the specific event is of interest.

Observer

There are several *consequences* to implementing the observer pattern:

- *Abstract coupling is used so that the Concrete Subject is not coupled with Concrete Observers (and vice versa).*
- *Updates a broadcast to any interested observers.*
- *Cascading updates may occur if/when one observer modifies the subject.*

Things to Consider

1. How does Observer affect the overall cohesion in the system?
2. The coupling?
3. What other design principles might observer make better or worse?
4. When might it not be appropriate to use Observer?