Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Observer Pattern: Intent

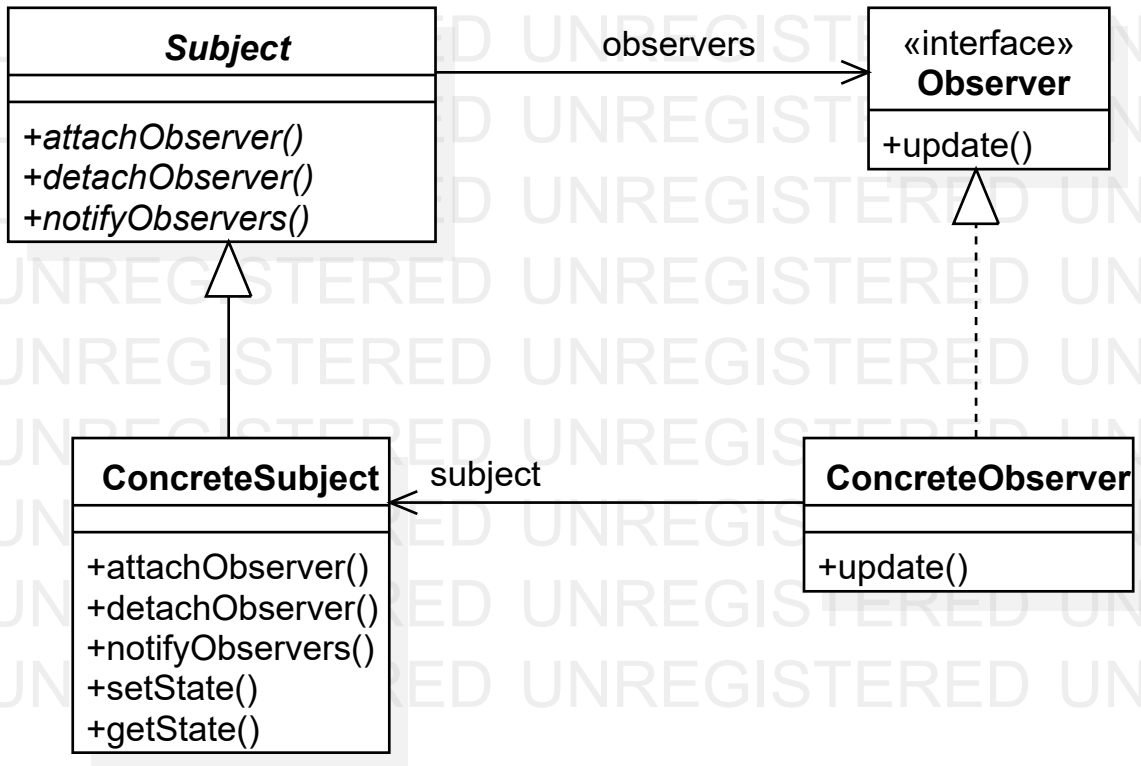
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Also known as Dependents, Publish-Subscribe

Observer Pattern

- The observer design pattern is a pattern where a subject keeps a list of observers.
- Observers rely on the subject to inform them of changes to the state of the subject.

Observer Pattern: Structure



Observer Pattern: Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.

Observer Pattern: Participants (contd.)

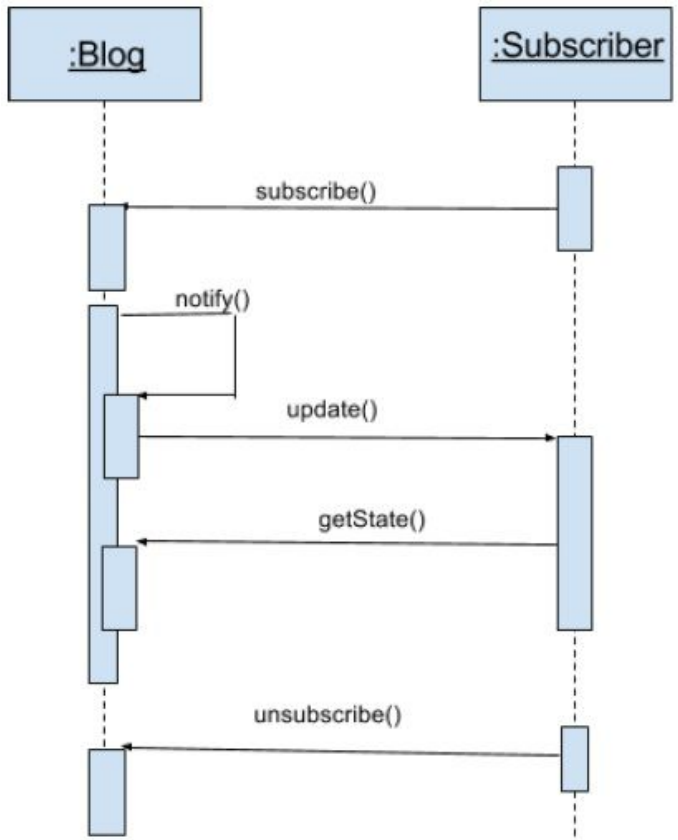
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.

- **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.

Observer Pattern: Example

- Imagine you have subscribed to a blog, and would like to receive notifications of any changes made to the blog.

- The sequence diagram for this example might look as below:



Observer Pattern: Example (contd.)

- A sequence diagram for observe patterns will have two major roles: the subject (the `blog`) and the observer (a `subscriber`).
- In order to form the subject and observer relationship, a `subscriber` must subscribe to the `blog`.
- The `blog` then needs to be able to notify subscribers of a change.
- The `notify` function keeps `subscribers` consistent, and is only called when a change has been made to the `blog`.

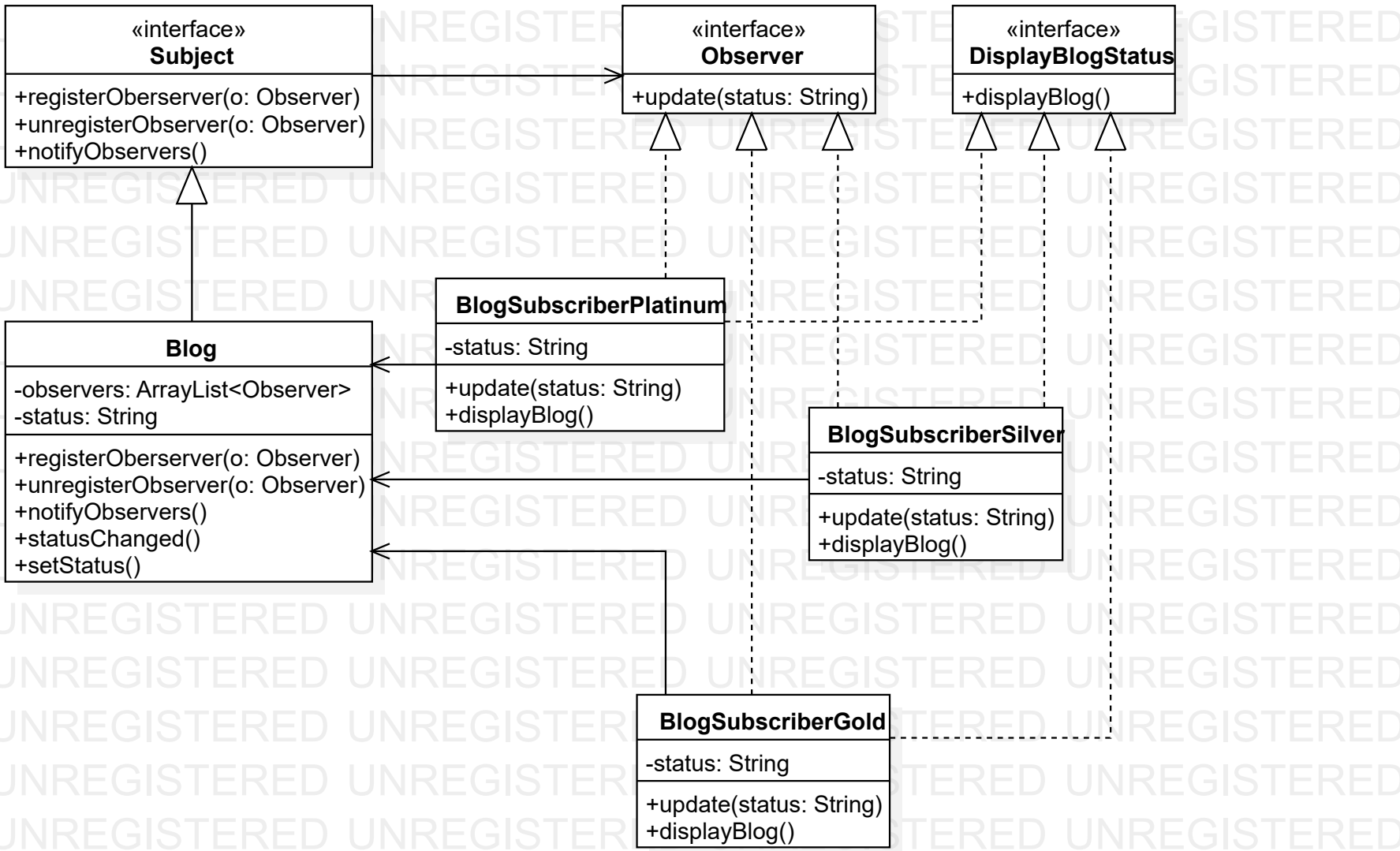
Observer Pattern: Example (contd.)

- If a change is made, the `blog` will make an update call to update subscribers.
- Subscribers can get the state of the `blog` through a `getState` call.
- It is up to the `blog` to ensure its subscribers get the latest information.
- To `unsubscribe` from the blog, subscribers could use the last call in the sequence diagram.
- `unsubscribe()` originates from the `subscriber` and lets the `blog` know the `subscriber` would like to be removed from the list of observers.

Observer Pattern: Example (contd.)

- The `Subject` superclass has three methods: `registerOberserver`, `unregisterObserver`, and `notifyObservers`.
- These are essential for a `subject` to relate to its observers.
- A `subject` may have zero or more observers registered at any given time.
- The `Blog` subclass would inherit these methods.

Observer Pattern: Example (contd.)



Observer Pattern: Example (contd.)

```
public interface Subject {
    public void registerOberserver(Observer o);
    public void unregisterObserver(Observer o);
    public void notifyObservers();
}
```

Observer Pattern: Example (contd.)

```
import java.util.ArrayList;

public class Blog implements Subject {

    private ArrayList<Observer> observers;
    private String status;

    public Blog() {
        observers = new ArrayList<Observer>();
    }
    public void registerOberserver(Observer o) {
        observers.add(o);
    }
    public void unregisterObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }
```

Observer Pattern: Example (contd.)

```
//Blog (contd.)

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(status);
        }
    }

    public void statusChanged() {
        notifyObservers();
    }

    public void setStatus(String status) {
        this.status = status;
        statusChanged();
    }
}
```

Observer Pattern: Example (contd.)

- The `Blog` class is a subclass of `Subject`, which will inherit and implement the `registerOberserver`, `unregisterObserver`, and `notifyObservers` methods.
- The `registerOberserver` method adds an observer to the list of observers.
- The `unregisterObserver` method removes an observer from the list.
- The `notify` method calls update upon each observer on the list.

Observer Pattern: Example (contd.)

- The `Observer` interface would be as below:

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

Observer Pattern: Example (contd.)

- The `Observer` interface only has the `update` method.
- An `observer` must have some way to `update` itself.
- The `Subscriber` class implements the `Observer` interface, providing the body of an `update` method so a subscriber can get what changed in the `blog`.

Observer Pattern: Example (contd.)

- The `Observer` interface makes sure all observer objects behave the same way.
- There is only a single method to implement, `update()`, which is called by the subject.
- The subject makes sure when a change happens, all its observers are notified to update themselves.
- In this example, there are three Subscriber that implements the `Observer` interface.
- This update method is called when the blog notifies the subscriber of a change.

Observer Pattern: Example (contd.)

```
public class BlogSubscriberPlatinum implements Observer,
     DisplayBlogStatus {
    private String status;

    public void update(String status) {
        this.status = status;
        displayBlog();
    }

    public void displayBlog() {
        System.out.println("Now displaying blog for Platinum
            subscribers: "+status);
    }
}
```

Observer Pattern: Example (contd.)

```
public class BlogSubscriberGold implements Observer,
     DisplayBlogStatus {
    private String status;

    public void update(String status) {
        this.status = status;
        displayBlog();
    }

    public void displayBlog() {
        System.out.println("Now displaying blog for Gold
            subscribers: "
               +status.substring(0, 8));
    }
}
```

Observer Pattern: Example (contd.)

```
public class BlogSubscriberSilver implements Observer,
     DisplayBlogStatus {
    private String status;

    public void update(String status) {
        this.status = status;
        displayBlog();
    }

    public void displayBlog() {
        System.out.println("Now displaying blog for Silver
            subscribers: "
               +status.substring(0, 8));
    }
}
```

Observer Pattern: Example (contd.)

```
public interface DisplayBlogStatus {
    public void displayBlog();
}
```

Observer Pattern: Example (contd.)

- Now test the communication between `Blog` and `Subscribers`.

```
public class ObserverBlogSubscriber {
    public static void main(String[] args) {
        BlogSubscriberPlatinum p = new
            BlogSubscriberPlatinum();
        BlogSubscriberSilver s = new BlogSubscriberSilver();

        Blog bp = new Blog();

        bp.registerOberserver(p);
        bp.registerOberserver(s);
        bp.setStatus("Observer Pattern");

        bp.unregisterObserver(s);
        BlogSubscriberGold g = new BlogSubscriberGold();
        bp.registerOberserver(g);
        bp.setStatus("Observer Pattern");
    }
}
```

Observer Pattern: Summary

- Observer design patterns save time when implementing a system. If many objects rely on the state of one, the observer design pattern has even more value.
- Instead of managing all observer objects individually, the subject manages them, and ensures observers are updating themselves as needed.
- This behavior pattern is typically used to make it easy to distribute and handle notifications of changes across systems in a manageable and controlled way.