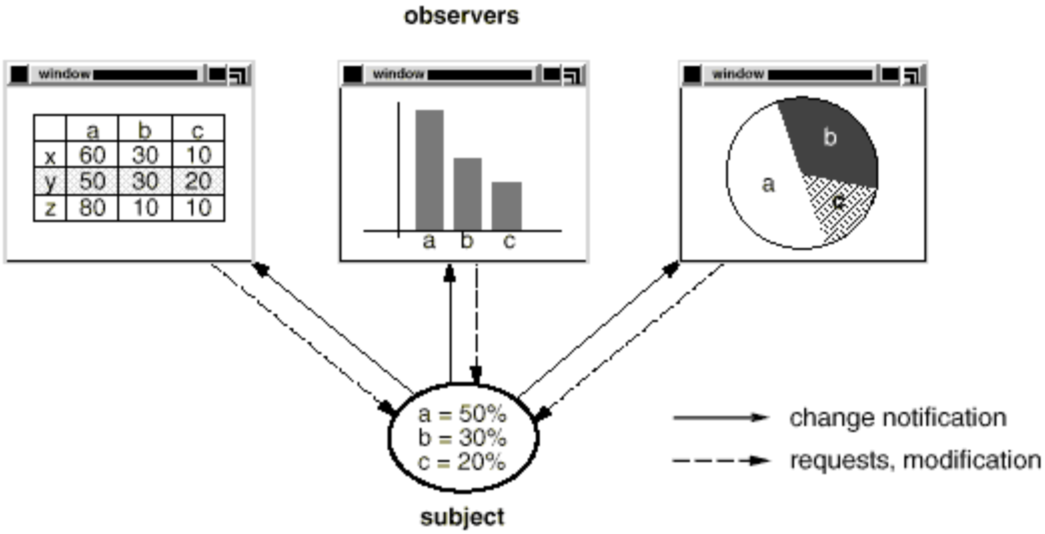
CMPE 202

Gang of Four Design Patterns

Observer

Motivation

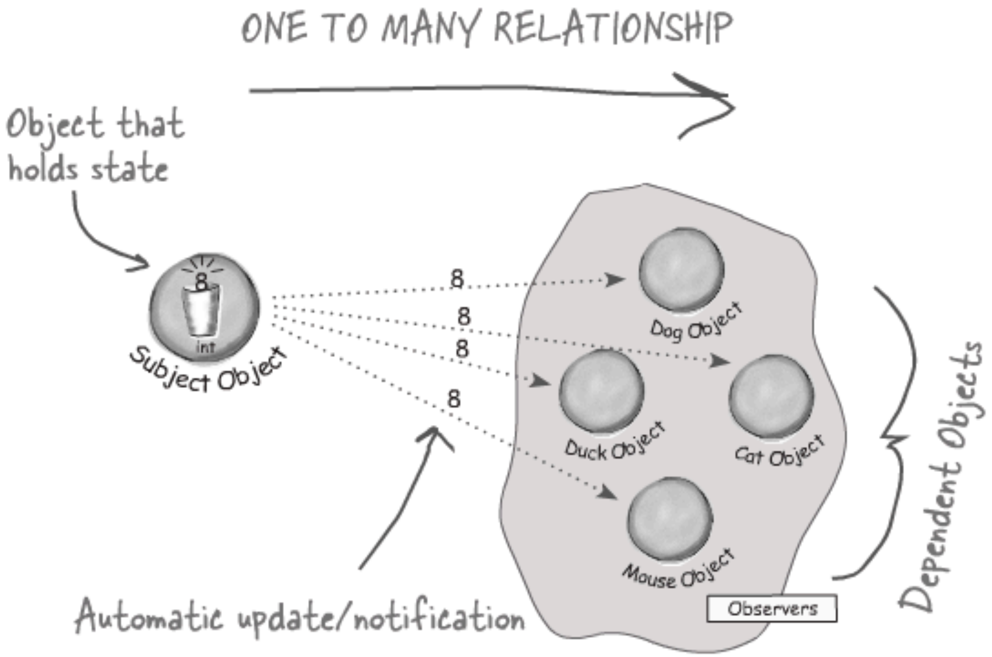
- Would like to decompose problem into classes, but doing so can sometimes introduce inconsistencies.
- Want to maintain consistency without tight coupling



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

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

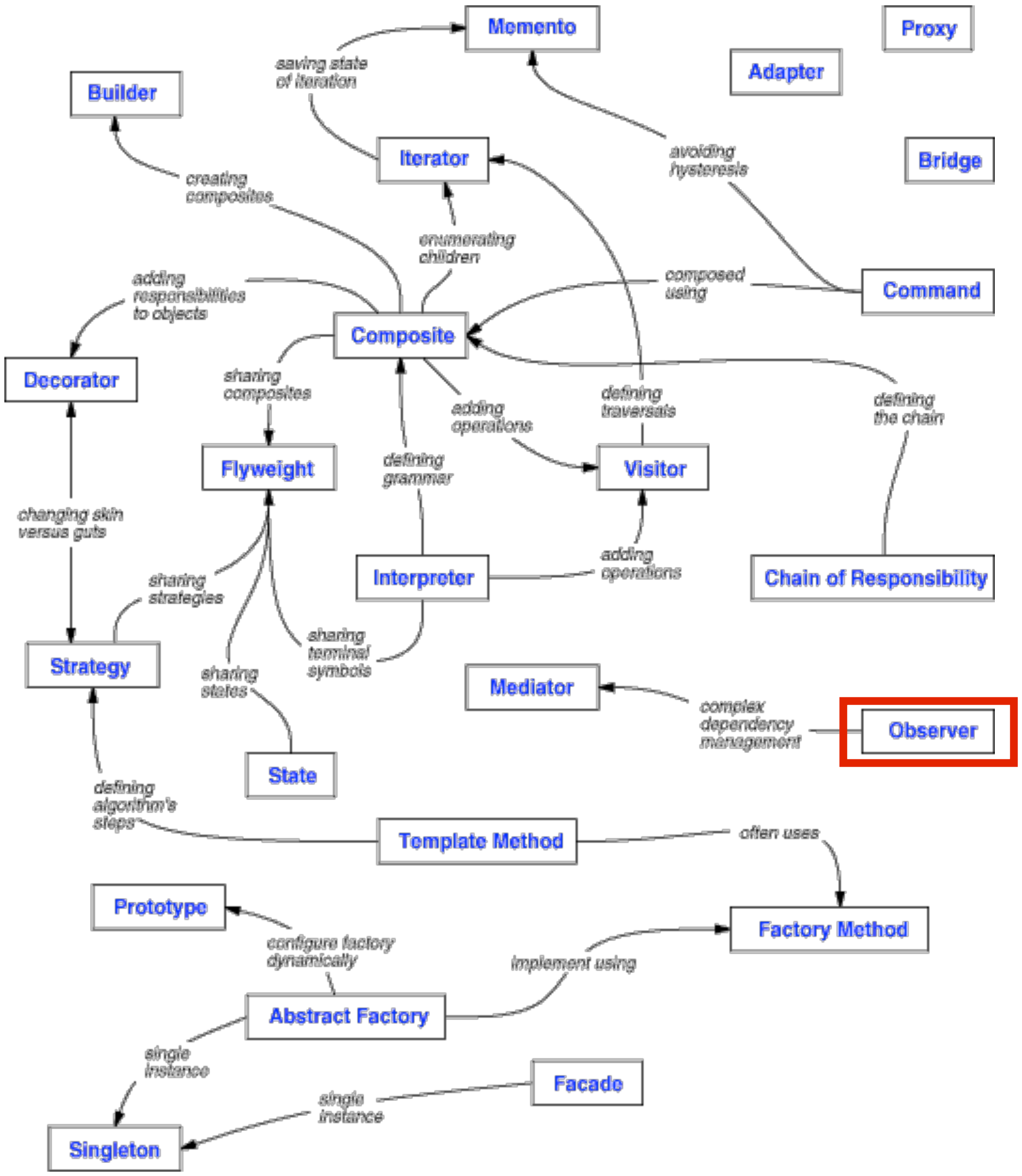
Let's relate this definition to how we've been talking about the pattern:

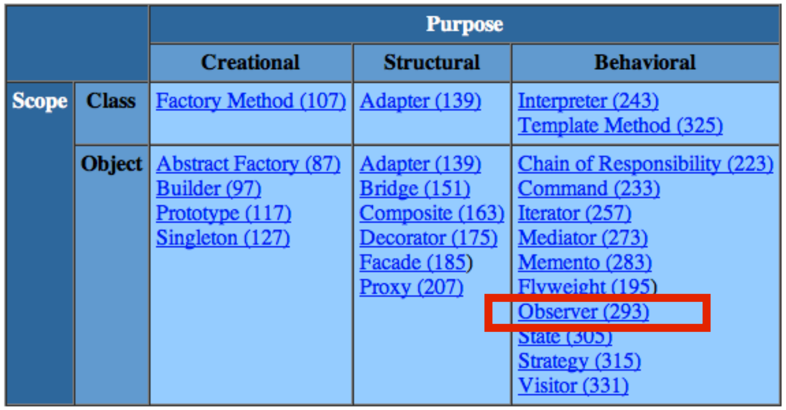


The Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

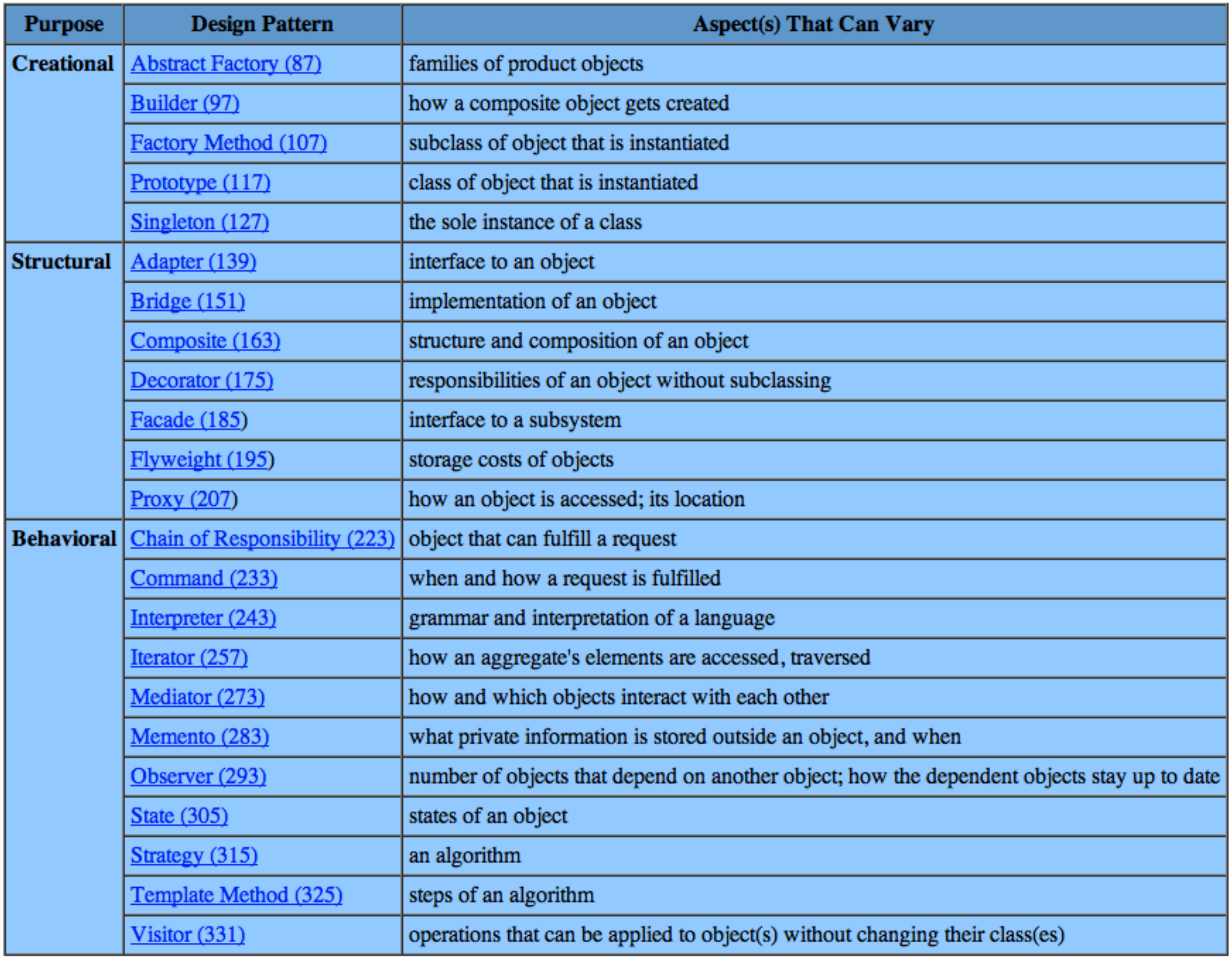
Observer





		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Design Pattern Catalog



Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

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

Applicability

Use the Observer pattern in any of the following situations:

- 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.

Participants

- **Subject** (Interface)
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.

- **Observer** (Interface)
  - 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.

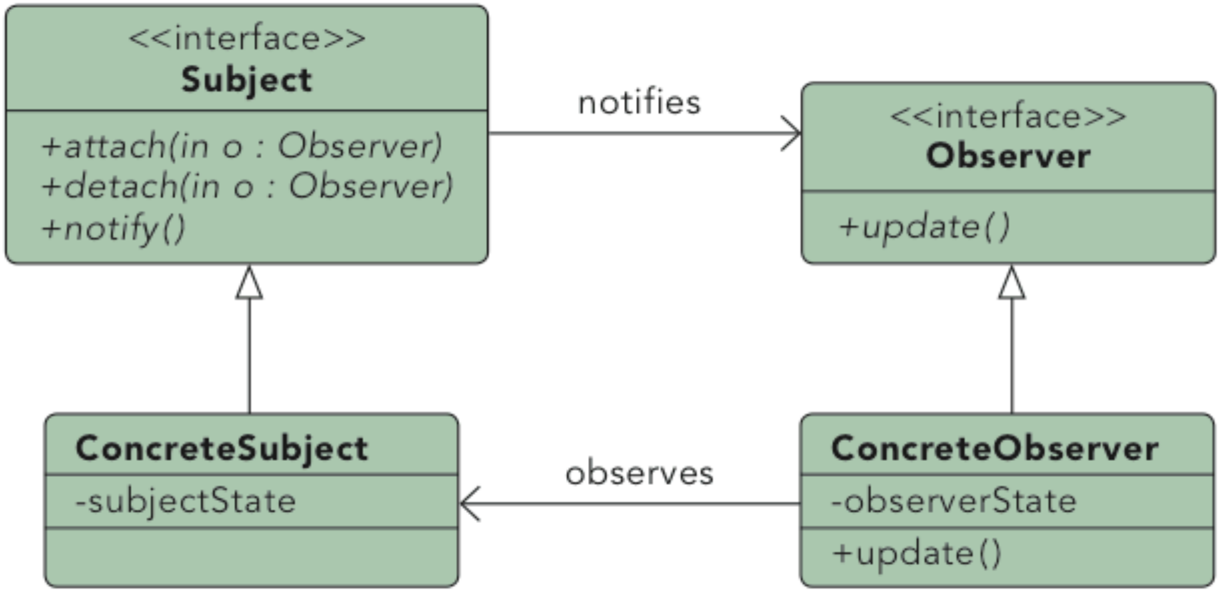
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

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 concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

OBSERVER

Object Behavioral

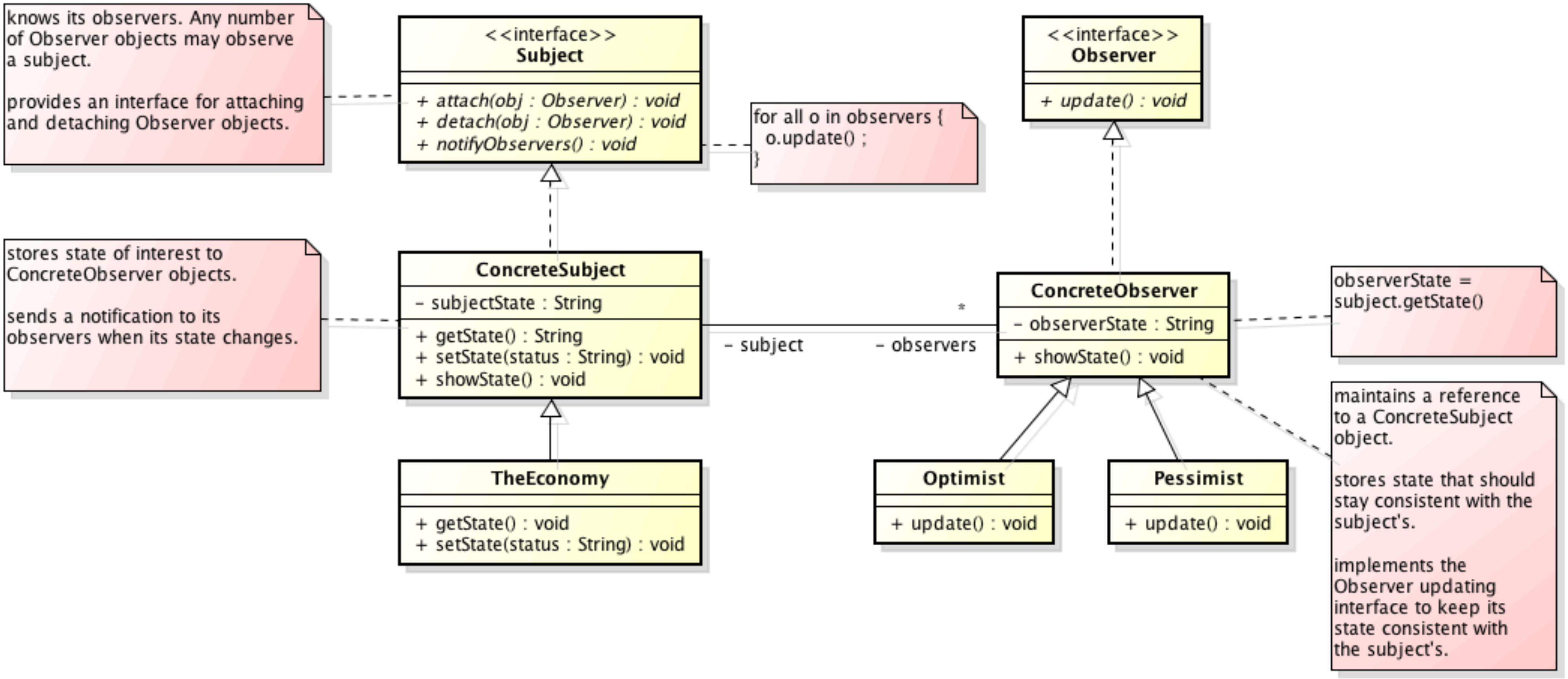


Purpose

Lets one or more objects be notified of state changes in other objects within the system.

Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.



knows its observers. Any number of Observer objects may observe a subject.

provides an interface for attaching and detaching Observer objects.

```
for all o in observers {
  o.update() ;
}
```

stores state of interest to ConcreteObserver objects.

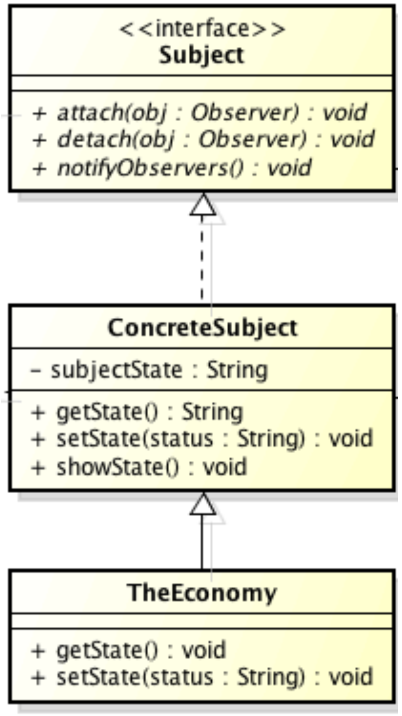
sends a notification to its observers when its state changes.

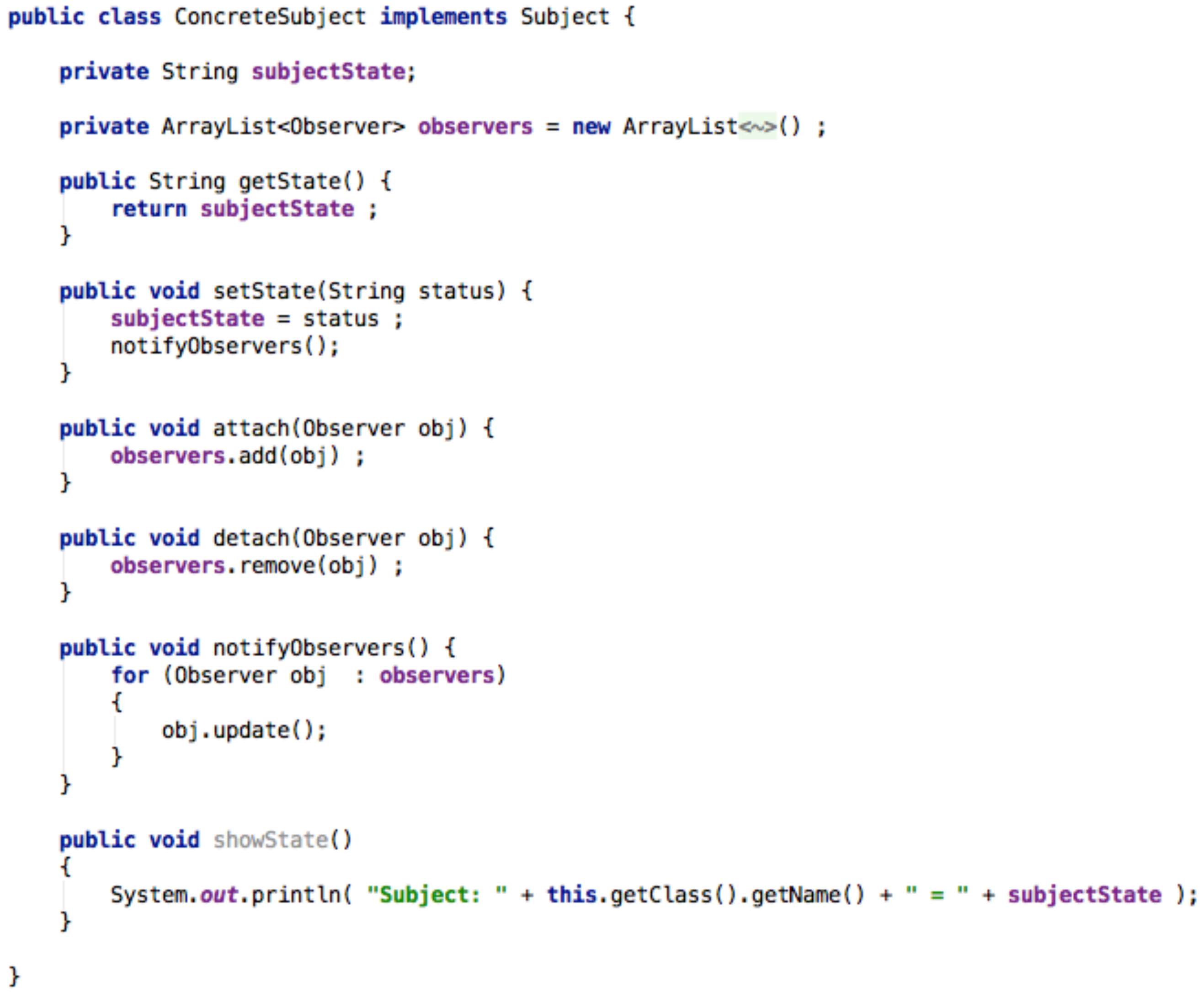
observerState = subject.getState()

maintains a reference to a ConcreteSubject object.

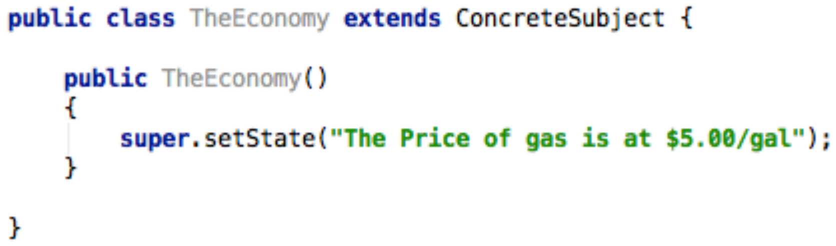
stores state that should stay consistent with the subject's.

implements the Observer updating interface to keep its state consistent with the subject's.

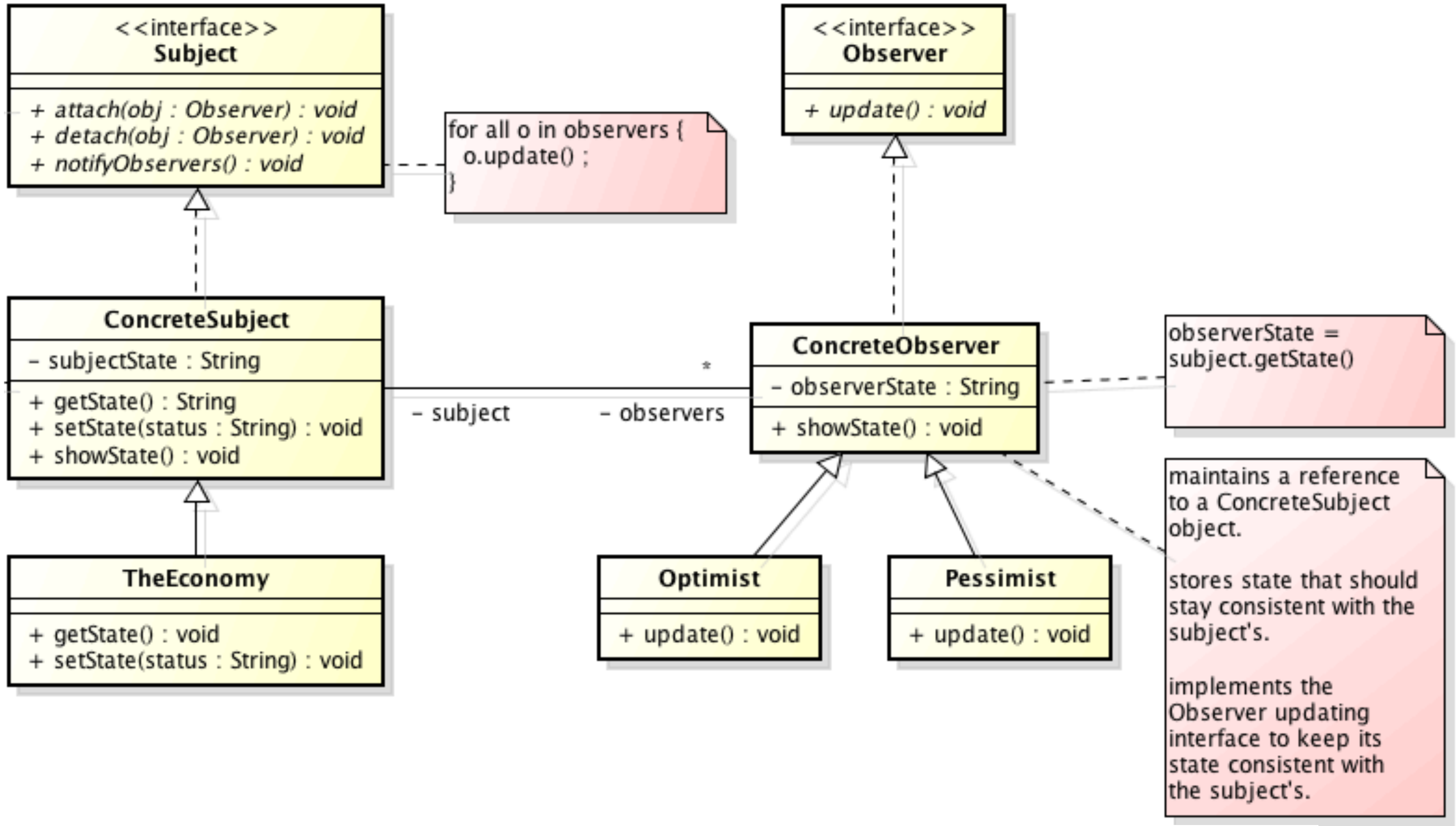

<<interface>>
Subject
+ attach(obj : Observer) : void
+ detach(obj : Observer) : void
+ notifyObservers() : void
ConcreteSubject
- subjectState : String
+ getState() : String
+ setState(status : String) : void
+ showState() : void
TheEconomy
+ getState() : void
+ setState(status : String) : void



```java
public class ConcreteSubject implements Subject {

    private String subjectState;

    private ArrayList<Observer> observers = new ArrayList<~>() ;

    public String getState() {
        return subjectState ;
    }

    public void setState(String status) {
        subjectState = status ;
        notifyObservers();
    }

    public void attach(Observer obj) {
        observers.add(obj) ;
    }

    public void detach(Observer obj) {
        observers.remove(obj) ;
    }

    public void notifyObservers() {
        for (Observer obj  : observers)
        {
            obj.update();
        }
    }

    public void showState()
    {
        System.out.println( "Subject: " + this.getClass().getName() + " = " + subjectState );
    }

}
```



```java
public class TheEconomy extends ConcreteSubject {

    public TheEconomy()
    {
        super.setState("The Price of gas is at $5.00/gal");
    }

}
```



<<interface>>
Subject

+ *attach(obj : Observer) : void*
+ *detach(obj : Observer) : void*
+ *notifyObservers() : void*

for all o in observers {
  o.update() ;
}

<<interface>>
Observer

+ *update() : void*

ConcreteSubject

- subjectState : String

+ getState() : String
+ setState(status : String) : void
+ showState() : void

- subject

*

- observers

ConcreteObserver

- observerState : String

+ showState() : void

observerState =
subject.getState()

TheEconomy

+ getState() : void
+ setState(status : String) : void

Optimist

+ update() : void

Pessimist

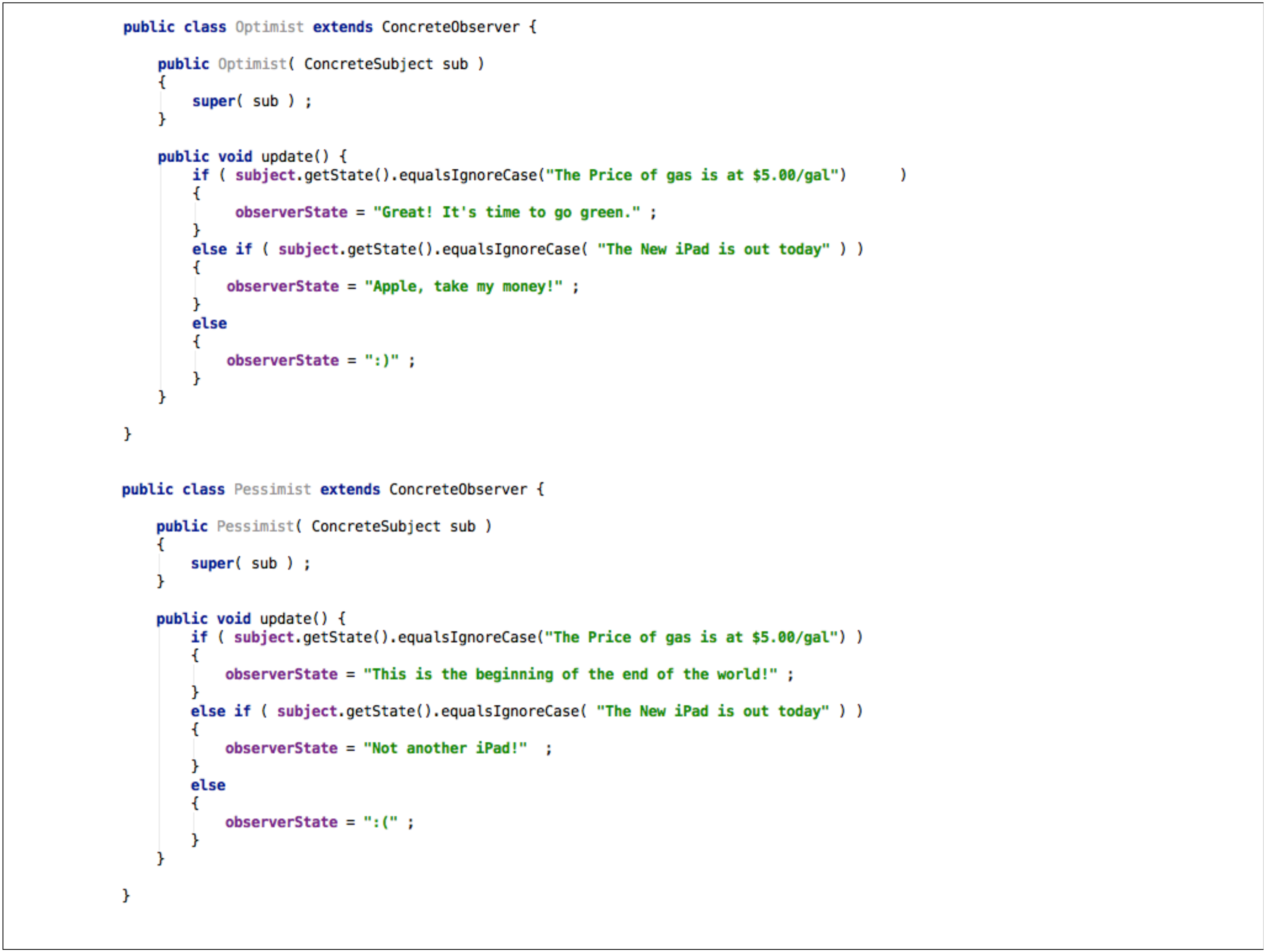
+ update() : void

maintains a reference to a ConcreteSubject object.

stores state that should stay consistent with the subject's.

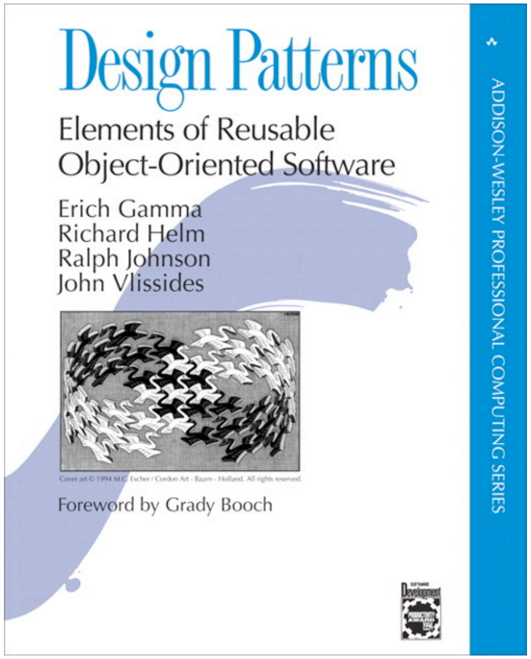
implements the Observer updating interface to keep its state consistent with the subject's.

```
public class ConcreteObserver implements Observer {

    protected String observerState;
    protected ConcreteSubject subject;

    public ConcreteObserver( ConcreteSubject theSubject )
    {
        this.subject = theSubject ;
    }

    public void update() {
        // do nothing
    }

    public void showState()
    {
        System.out.println( "Observer: " + this.getClass().getName() + " = " + observerState );
    }

}
```



```
public class Optimist extends ConcreteObserver {

    public Optimist( ConcreteSubject sub )
    {
        super( sub ) ;
    }

    public void update() {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal")      )
        {
             observerState = "Great! It's time to go green." ;
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Apple, take my money!" ;
        }
        else
        {
            observerState = ":)" ;
        }
    }

}


public class Pessimist extends ConcreteObserver {

    public Pessimist( ConcreteSubject sub )
    {
        super( sub ) ;
    }

    public void update() {
        if ( subject.getState().equalsIgnoreCase("The Price of gas is at $5.00/gal") )
        {
            observerState = "This is the beginning of the end of the world!" ;
        }
        else if ( subject.getState().equalsIgnoreCase( "The New iPad is out today" ) )
        {
            observerState = "Not another iPad!"  ;
        }
        else
        {
            observerState = ":(" ;
        }
    }

}
```

Resources for this Tutorial



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

A Brain-Friendly Guide

Head First Design Patterns

Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory pattern is probably wrong

Discover the secrets of the Patterns Guru

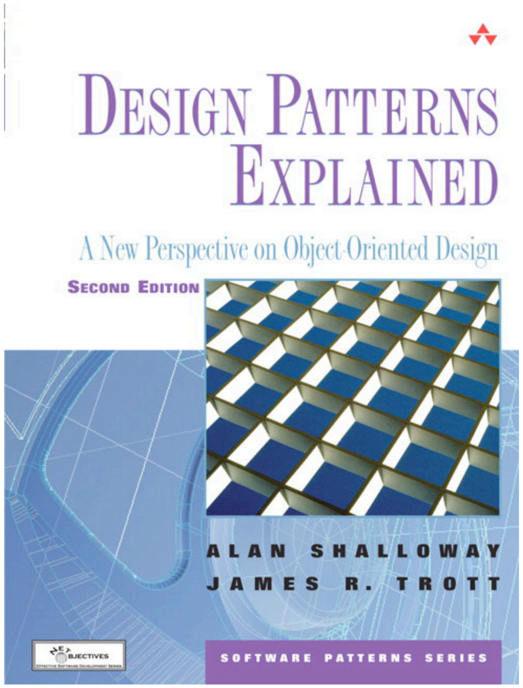
Load the patterns that matter straight into your brain

Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

See why Jim's love life improved when he cut down his inheritance

O'REILLY®

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates



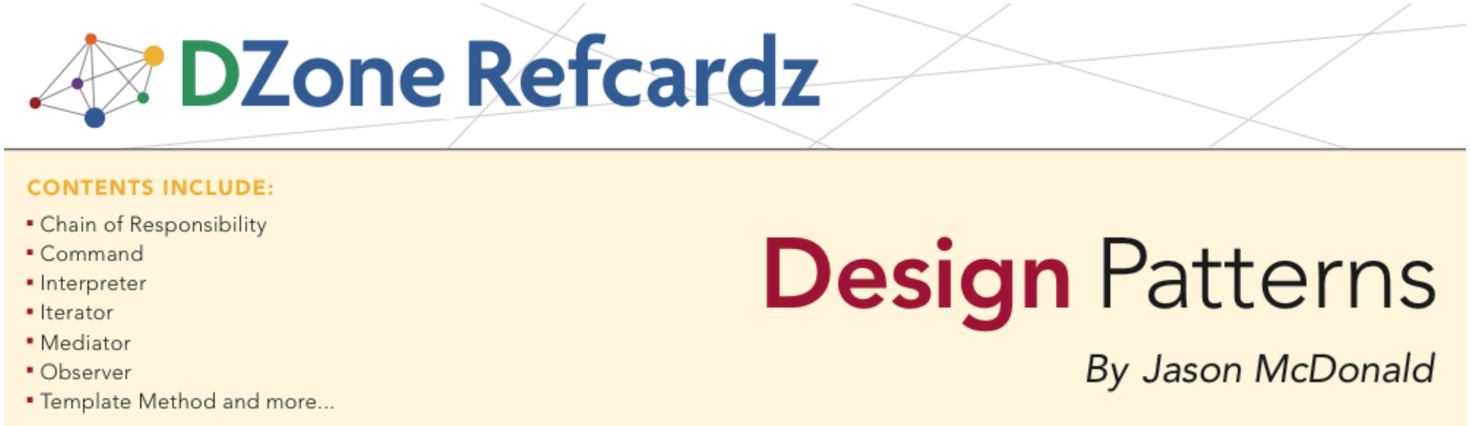
Design Patterns Explained

A New Perspective on Object-Oriented Design

Second Edition

Alan Shalloway
James R. Trott

Software Patterns Series



DZone Refcardz

CONTENTS INCLUDE:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Observer
- Template Method and more...

Design Patterns

By Jason McDonald