CS 535 Object-Oriented Programming & Design
Fall Semester, 2000
Doc 17 Heuristics: Separating Abstractions
Contents

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 293-303 Observer Pattern

Object-Oriented Design Heuristics, Riel, Addison-Wesley, 1996, Heuristics 2.2, 2.8, 2.10, 3.4, 3.5, 4.6, 4.13

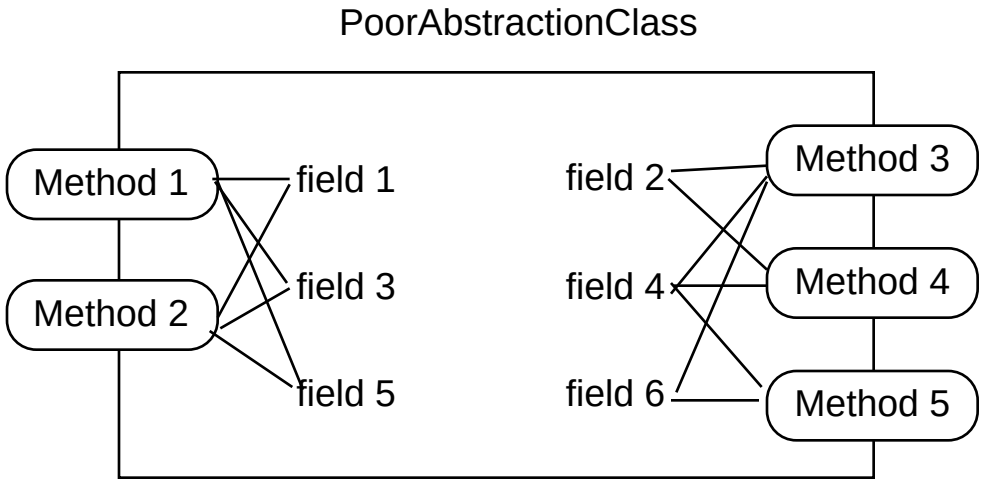
Too Many Abstractions

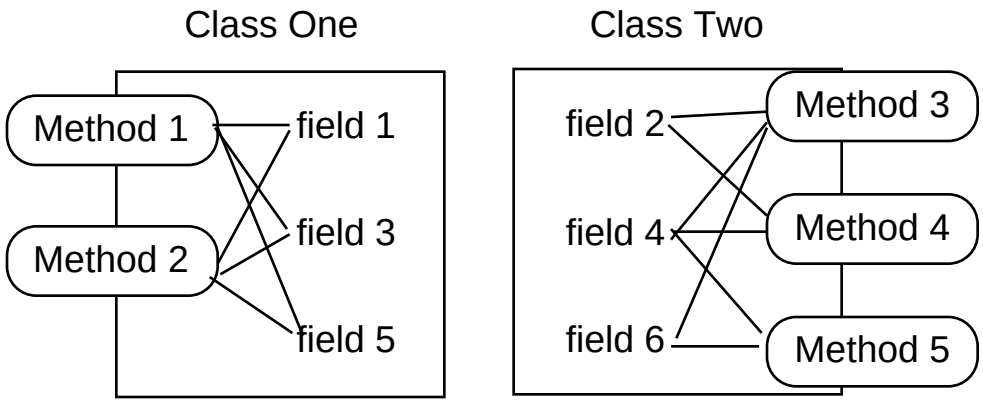
2.8 A class should capture one and only one key abstraction

2.10 Spin off nonrelated information into another class

3.4 Beware of classes that have too much noncommunicating behavior

4.6 Most of the methods defined in a class should be using most of the data members most of the time





Independence from Users

2.2 Users of a class must be dependent on its public interface, but a class should not be dependent on its users

4.13 A class must know what it contains, but it should not know who contains it.

3.5 In applications that consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface

Example

Build a Timer class (or alarm clock).

We need to be able to display the time remaining and notify an object when the alarm goes off.

How do we satisfy the above heuristics?

Basic Timer Class

Here is a class that works, but violates the heuristics. It talks directly to the screen.

```
public class Timer extends Thread {
   int secondsRemaining;

   public Timer( int minutes, int seconds ) {
      Assert.condition( ( minutes >= 0 ) && ( seconds >= 0 ));
      secondsRemaining = (minutes * 60) + seconds;
      setPriority( Thread.MAX_PRIORITY );
   }

   public String toString()  {
      int minutes = secondsRemaining / 60;
      int seconds = secondsRemaining % 60;

      return "" + minutes + ":" + seconds;
   }

   public void run() {
      try {
         while (secondsRemaining > 0 ) {
            sleep( 1000 );
            secondsRemaining--;
            System.out.println( this.toString() );
         }
         System.out.println( "Time to wake up" );
      } catch ( InterruptedException stopTimer ){ //stop the timer }
   }
}
```

Using the Timer class

```
public class Test  {
   public static void main( String args[] )  throws Exception
      {
      Timer clock = new Timer( 0, 8 );
      clock.start();
      }
```

How do we decouple the Timer class from the interface and the users?

Observer Pattern

Also know as Observer-Observable, dependants, and publish-subscribe. This basic idea is used in the model-view-controller (MVC). The model is the observable, the view and controller are the observer.

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

Applicability

Use the Observer pattern:

- When an abstraction has two aspects, one dependent on the other.
- When a change to one object requires changing others, and you don't 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.

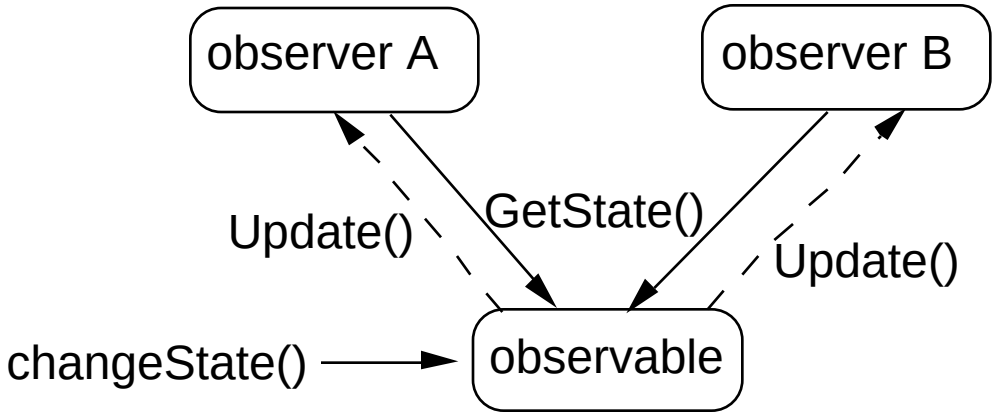
Participants

Observable (Subject, or model)
The model of the application
Maintains a list of dependents
When it changes state, it notifies its dependents

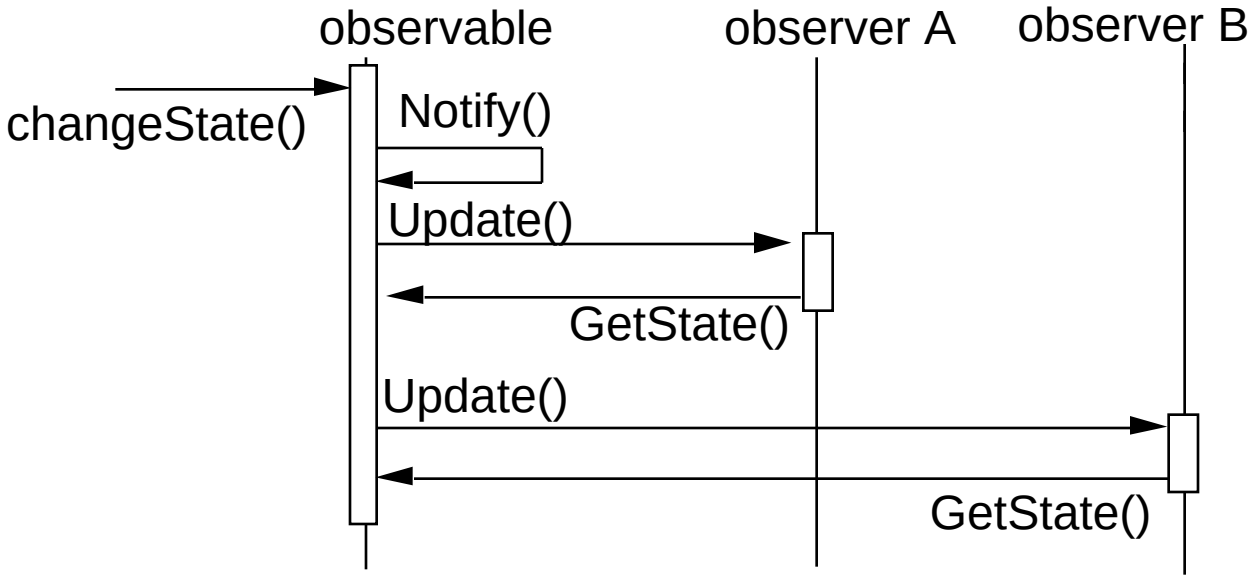
Observer
Object with interest in knowing when observable changes state

Collaborations

Object Diagram



Timing Diagram



Example Code and Java API

Java does include Observable and Observer in the JDK API. They are more complex than the implementation I use here. I did not use the built-in classes to avoid additional complexity. The goal here is to understand the concept not to cover details about Java API. See the JDK on-line documentation (java.util.Observable, java.util.Observer) and http://www.eli.sdsu.edu/courses/fall97/cs535/notes/observer/observer.html for more information about the standard Java classes.

Java's class Object has a notify() method which is related to threads. To avoid confusion with this method I will use the method name notifyObservers().

The implementation of Observable below does not have all the functionality of java.util.Observable. In particular, my Observable class does not handle the synchronization of threads as best it could. The notifyObservers method needs to be refined. See the source code of java.util.Observable or see notifyModelChanged() method on slide 4 of
http://www.eli.sdsu.edu/courses/spring98/cs696/notes/beanEvents/beanEvents.html

Observer

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

Observable

```
public class Observable{
   private ArrayList observerList = new ArrayList();

   public synchronized void addObserver( Observer myObserver) {
      observerList.add( myObserver);
   }

   public synchronized void removeObserver( Observer myObserver ) {
      observerList.remove( myObserver);
   }

   public synchronized void notifyObservers( String message ) {
      if (message == null )
         message = "";
      Iterator observers = observerList.iterator();

      while ( observers.hasNext() ) {
         Observer anObserver =(Observer) observers.next();
         anObserver.update( message );
      }
   }
}
```

Using the Observer/Observable
Timer

```
public class Timer extends Thread {
   public static final String TICK = "tick";
   public static final String ALARM = "alarm";

   Observable observers = new Observable();

   int secondsRemaining;

   public Timer( int minutes, int seconds ) {
      Assert.condition( ( minutes >= 0 ) && ( seconds >= 0 ));
      secondsRemaining = (minutes * 60) + seconds;
      setPriority( Thread.MAX_PRIORITY );
   }

   public void addObserver( Observer myObserver ){
      observers.addObserver( myObserver );
   }

   public String toString() {
      int minutes = secondsRemaining / 60;
      int seconds = secondsRemaining % 60;

      return "" + minutes + ":" + seconds;
   }

   public void run() {
      try {
         while (secondsRemaining > 0 ) {
            sleep( 1000 );
            secondsRemaining--;
            observers.notifyObservers( TICK );
         }
         observers.notifyObservers( ALARM );
      } catch ( InterruptedException stopTimer ) { //stop the timer
      }
   }
}
```

ASCIITimerView

```
public class ASCIITimerView implements Observer {
   Timer myTimer;

   public ASCIITimerView( Timer aTimer ) {
      myTimer = aTimer;
      myTimer.addObserver( this );
   }

   public void update( String message ) {
      if ( message.equals( Timer.TICK ) )
         System.out.println( "Time left: " + myTimer.toString() );
   }
}
```

Person

```
public class Person implements Observer {

   public void setAlarm( Timer aTimer ) {
      aTimer.addObserver( this );
   }

   public void update( String message ) {
      if ( message.equals( Timer.ALARM ) )
         wakeUp();
   }

   public void wakeUp() {
      System.out.println( "Mom, I am awake");
   }
}
```

Sample Usage

```
public static void main() {
   Timer clock = new Timer( 0, 8 );
   ASCIITimerView aView = new ASCIITimerView( clock );
   Person teenager = new Person();
   Person child = new Person();
   teenager.setAlarm( clock);
   child.setAlarm( clock);
   clock.start();
}
```

Output

```
Time left: 0:7
Time left: 0:6
Time left: 0:5
Time left: 0:4
Time left: 0:3
Time left: 0:2
Time left: 0:1
Time left: 0:0
Mom, I am awake
Mom, I am awake
```

Variants

- Use multiple observer sets for different types of events

  The Timer has two different type of events, tick and alarm. The observers must check for the type of event. If the observable contained more than one set of observers, the observers could register interest only in the events they need to know about. Java uses the term “listeners” to refer to this use of the pattern.

- Send a reference to the source of the event in the update message

  This will allow observers to observer more than one observable and permit them to not keep a reference to the observable.

For more variants see *Design Patterns: Elements of Reusable Object-Oriented Software.*

Modified Example

Observer

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

Observable

```
public class Observable {
   private ArrayList observerList = new ArrayList();

   public synchronized void addObserver( Observer myObserver ){
      observerList.add( myObserver );
   }

   public synchronized void removeObserver( Observer myObserver ) {
      observerList.remove( myObserver );
   }

   public synchronized void notifyObservers( Object message ) {
      Iterator observers = observerList.iterator();

      while ( observers.hasNext() ) {
         Observer anObserver =(Observer) observers.next();
         anObserver.update( message );
      }
   }
}
```

Timer

```
public class Timer extends Thread {
   Observable timeObservers = new Observable();
   Observable alarmObservers = new Observable();

   int secondsRemaining;

   public Timer( int minutes, int seconds ) {
      Assert.condition( ( minutes >= 0 ) && ( seconds >= 0 ));
      secondsRemaining = (minutes * 60) + seconds;
      setPriority( Thread.MAX_PRIORITY );
   }

   public void addTimeObserver( Observer myObserver ){
      timeObservers.addObserver( myObserver );
   }

   public void addAlarmObserver( Observer myObserver ){
      alarmObservers.addObserver( myObserver );
   }

   public String toString() {
      int minutes = secondsRemaining / 60;
      int seconds = secondsRemaining % 60;

      return "" + minutes + ":" + seconds;
   }

   public void run() {
      try {
         while (secondsRemaining > 0 ) {
            sleep( 1000 );
            secondsRemaining--;
            timeObservers.notifyObservers( this );
         }
         alarmObservers.notifyObservers( this );
      } catch ( InterruptedException stopTimer ) { //stop the timer
      }
   }
}
```

ASCIITimerView

```
public class ASCIITimerView implements Observer {
   public ASCIITimerView( Timer aTimer ) {
      aTimer.addTimeObserver( this );
   }

   public void update( Object aTimer ) {
      System.out.println( "Time left: " + aTimer.toString() );
   }
}
```

Consequences of Observer Pattern

Abstract coupling between Observable and Observer

The observable only knows that it has a lists of observers. It does not know the exact type of the observer. It just knows that the observer has an update method. Hence the coupling is abstract (between the observable and the interface observer) and minimal

Support for broadcast communication

The update message is a broadcast to all observers

Unexpected updates

Simple change one observable can cause numerous updates in the observers of that observable. Those observers may also be observable, so will send update messages to their observers. So one update message can trigger many updates, which can be expensive or distracting.

Updates can take too long

The observable can not perform any work until all observers are done

For more implementation details see: *Design Patterns: Elements of Reusable Object-Oriented Software*

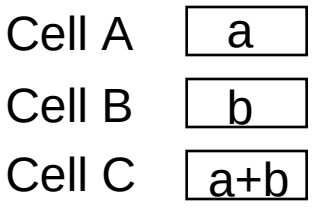
Spreadsheet Example

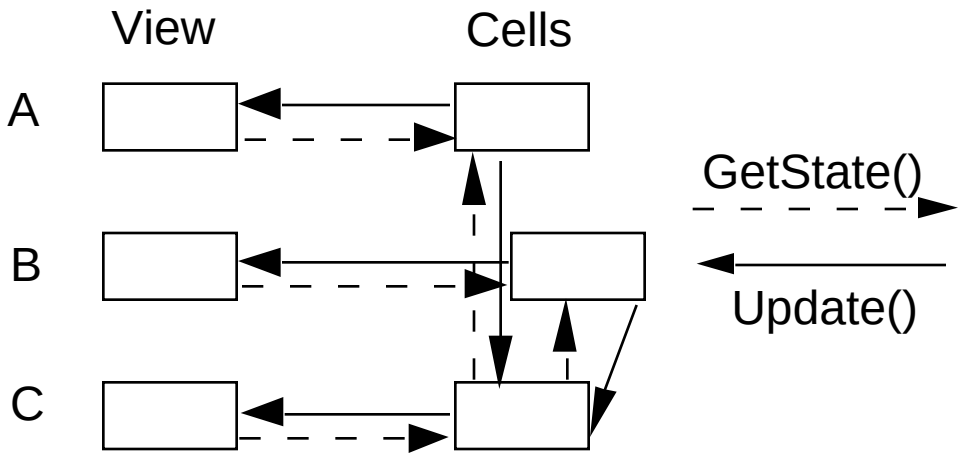
How to implement a spreadsheet?

Make SpreadsheetCells both observer and observable. It observers other SpreadsheetCells. When a cell changes it notifies its observers (SpreadsheetCells and SpreadsheetCellsViews) of the change

Make SpreedsheetCellView observers of SpreadsheetCells

How do we make cell C update it self and its view when the user changes the value of cell A?





When the user types a new value in view A, the view changes the value in cell A. Cell A then sends an update message to its observers, which includes cell C. Cell C then recomputes its sum, and sends an update message to its observers, which includes the view on cell C. The view then updates its display value.