Programming and Algorithms II

Lecture 9: Observer

Nicolas Wu

nicolas.wu@bristol.ac.uk

Department of Computer Science
University of Bristol

Observer Problem

Observer Problem

- Let's suppose we want to implement a News Cafe, where people come to read their favourite newspapers …

Observer Problem

- Implement a Newspaper
- Implement a Reader
- Implement the NewsCafe

Observer Problem

Try #1: The Newspaper

```
import java.util.*;

class Newspaper {
  private String name;
  private List<String> articles;

  Newspaper(String name) {
    articles = new ArrayList<String>();
    this.name = name;
  }

  void addArticle(String article) {
    articles.add(article);
  }

  String getArticle(int issue) {
    String article = name + " : ";
    if (issue > 0 && issue <= articles.size())
      return (article + articles.get(issue - 1));
    else
      return (article + " : Bad issue number!");
  }

  int getIssue() {
    return (articles.size());
  }
}
```

We'll pretend that articles are just a list of Strings

Adding an article makes a whole new issue

You can get an article by asking for the right issue

The current issue number

Observer Problem

Try #1: The Reader

A reader has a name, and reads a paper

```
class Reader {
  String name;
  Newspaper newspaper;

  Reader(String name, Newspaper newspaper) {
    this.name = name;
    this.newspaper = newspaper;
  }

  void readArticle() {
    int issue      = newspaper.getIssue();
    String article = newspaper.getArticle(issue);
    System.out.println(name + " reads: " + article);
  }
}
```

A reader always reads the article that's hot off the press

Observer Problem

Try #1: The NewsCafe

This will just hold our entry point ... it's not really OOP

```
class NewsCafe {
  public static void main(String[] args) {
    Newspaper times = new Newspaper("The Times");
    Reader tom = new Reader("Tom", times);
    Reader jack = new Reader("Jack", times);

    times.addArticle("Stormy weather!");

    tom.readArticle();
    jack.readArticle();

    times.addArticle("Students on strike!");

    tom.readArticle();

    times.addArticle("Hottest day!");

    tom.readArticle();
    jack.readArticle();
    jack.readArticle();

  }
}
```

Our newspaper and readers are created

Oh! A new article!

Quick read!

More news!

Read more! (Jack didn't drop by the cafe today)

Poor Jack never got to read the article about "Students on Strike", in fact, he had to read the "Hottest day" story twice.

Observer Problem

Q.How can we resolve these problems?

- The problem with our code so far:
  - They might miss an article if they don't read on time

A. We could fix this by adding a readArticle() after every addArticle()

  - They might read the same article multiple times if the news didn't update
  - Tom and Jack can only ever read one newspaper each

A. We could add a list of newspapers

Observer Problem

- Add readArticle() after every addArticle()
- Modify Reader to have a list of newspapers

Observer Problem

Try #2: The Reader

```
import java.util.*;

class Reader {
  String name;
  List<Newspaper> newspapers;

  Reader(String name) {
    this.name = name;
    newspapers = new ArrayList<Newspaper>();
  }

  void readArticle(Newspaper newspaper) {
    int issue      = newspaper.getIssue();
    String article = newspaper.getArticle(issue);
    System.out.println(name + " reads: " + article);
  }

  void readArticles() {
    for (Newspaper newspaper : newspapers) {
      readArticle(newspaper);
    }
  }

  void subscribe(Newspaper newspaper) {
    newspapers.add(newspaper);
  }
}
```

We now store a list of newspapers

Here's how to read one newspaper

Go through each newspaper subscription and read it

Here's how we subscribe

Observer Problem

```
class NewsCafe {
  public static void main(String[] args) {
    Reader tom = new Reader("Tom");
    Reader jack = new Reader("Jack");

    Newspaper times = new Newspaper("The Times");
    tom.subscribe(times);
    times.addArticle("Stormy weather!");
    tom.readArticles();

    Newspaper guardian = new Newspaper("The Guardian");
    tom.subscribe(guardian);
    jack.subscribe(guardian);

    guardian.addArticle("Bad news!");
    tom.readArticles();
    jack.readArticles();

    guardian.addArticle("Good news!");
    tom.readArticles();
    jack.readArticles();

    times.addArticle("Stormy weather!");
    tom.readArticles();
  }
}
```

Try #2: The NewsCafe

Subscribing is neat!

Hm, but the rest of this code isn't great

Q. What goes wrong here?

Observer Problem

```
class NewsCafe {
  public static void main(String[] args) {
    Reader tom = new Reader("Tom");
    Reader jack = new Reader("Jack");

    Newspaper times = new Newspaper("The Times");
    tom.subscribe(times);
    times.addArticle("Stormy weather!");
    tom.readArticles();

    Newspaper guardian = new Newspaper("The Guardian");
    tom.subscribe(guardian);
    jack.subscribe(guardian);

    guardian.addArticle("Bad news!");
    tom.readArticles();
    jack.readArticles();

    guardian.addArticle("Good news!");
    tom.readArticles();
    jack.readArticles();

    times.addArticle("Stormy weather!");
    tom.readArticles();
  }
}
```

Try #2: The NewsCafe

Subscribing is neat!

Hm, but the rest of this code isn't great

Q. What goes wrong here?

A. Tom has to re-read some Times articles if the Guardian publishes more often

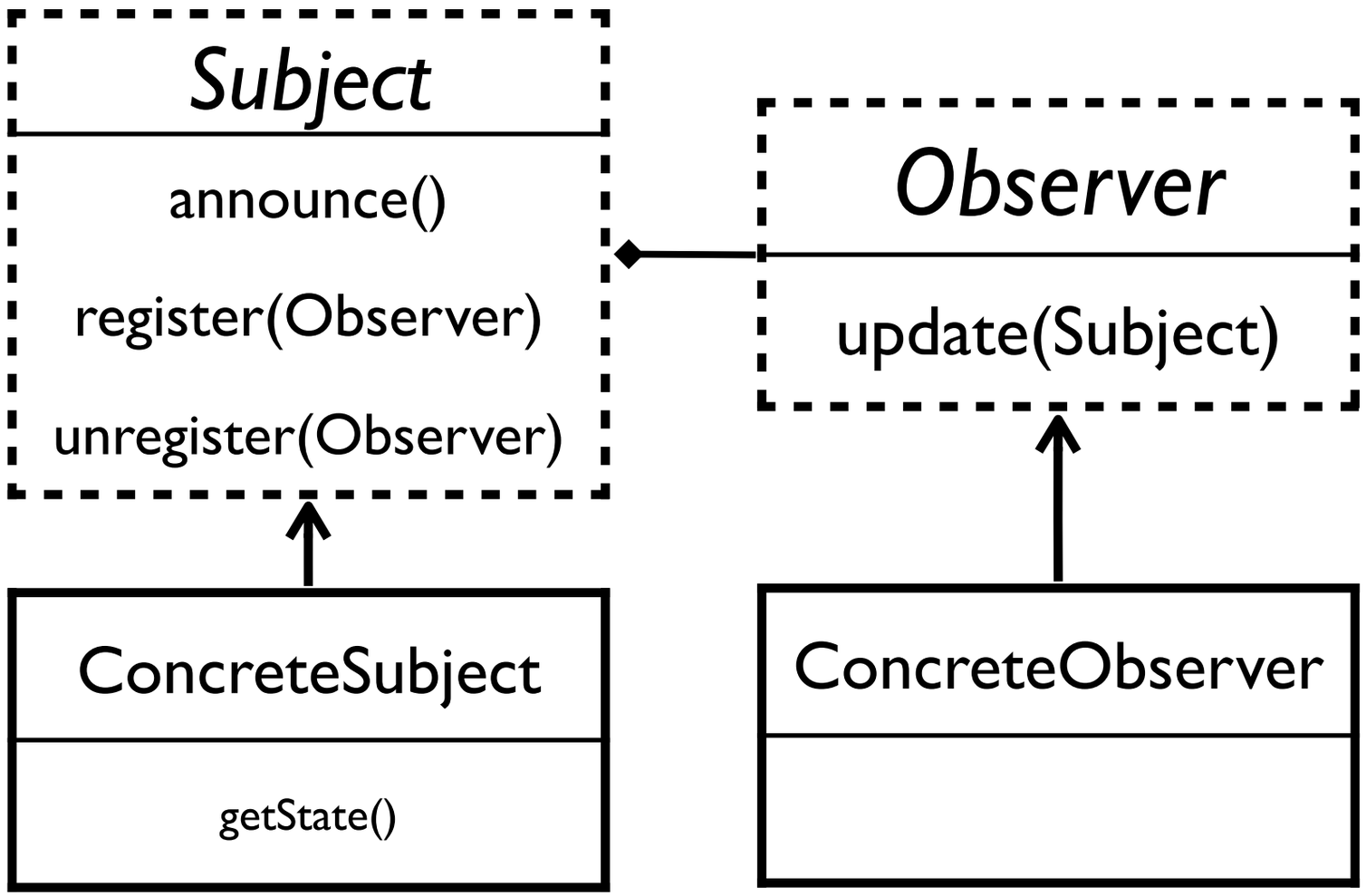
A. It's a bit fiddly to have to add readArticles() after each addArticle(). In fact, we even have to make sure we tell the right reader to read!

Observer Pattern

Observer Pattern

- GoF: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically
- Example: Someone makes a move on a board game, so all other players must be notified so they can update

Observer Pattern



- A *subject* can *register* or *unregister* several *observers*
- When the *subject* is ready to *notify*, it should *update* each registered *observer*

Observer Pattern

- Add register() and unregister() to Newspaper, the *Subject*
- Add announce() to the Newspaper
- Add update() to Reader, the *Observer*
- Modify NewsCafe to use registrations

Observer Pattern

```
import java.util.*;

class Newspaper {
  private String name;
  private List<String> articles;
  private Set<Reader> readers;

  Newspaper(String name) {
    articles = new ArrayList<String>();
    readers  = new HashSet<Reader>();
    this.name = name;
  }

  void addArticle(String article) {
    articles.add(article);
    announce();
  }

  String getArticle(int issue) {
    String article = name + " : ";
    if (issue > 0 && issue <= articles.size())
      return (article + articles.get(issue - 1));
    else
      return (article + " : Bad issue number!");
  }

  int getIssue() {
    return (articles.size());
  }

  void register(Reader reader) {
    readers.add(reader);
  }

  void unregister(Reader reader) {
    readers.remove(reader);
  }

  void announce() {
    for (Reader reader : readers) {
      reader.readArticle(this);
    }
  }
}
```

We announce a new article when one is added!

The announcement tells the readers to get reading

Observer Pattern

```
import java.util.*;

class Reader {
  String name;
  List<Newspaper> newspapers;

  Reader(String name) {
    this.name = name;
    newspapers = new ArrayList<Newspaper>();
  }

  void readArticle(Newspaper newspaper) {
    int issue      = newspaper.getIssue();
    String article = newspaper.getArticle(issue);
    System.out.println(name + " reads: " + article);
  }

  void readArticles() {
    for (Newspaper newspaper : newspapers) {
      readArticle(newspaper);
    }
  }

  void subscribe(Newspaper newspaper) {
    newspapers.add(newspaper);
    newspaper.register(this);
  }
}
```

We'll read articles from a particular newspaper

We need only register ourselves with the Newspaper

Observer Pattern

```
class NewsCafe {
  public static void main(String[] args) {
    Reader tom = new Reader("Tom");
    Reader jack = new Reader("Jack");

    Newspaper times = new Newspaper("The Times");
    tom.subscribe(times);
    times.addArticle("Stormy weather!");

    Newspaper guardian = new Newspaper("The Guardian");
    tom.subscribe(guardian);
    jack.subscribe(guardian);

    guardian.addArticle("Bad news!");
    guardian.addArticle("Good news!");
    times.addArticle("Good weather!");
  }
}
```

Wow, we actually **removed** code here: the NewsCafe is now a slick reading machine!

Observer Pattern

- You might notice that there's an Observer *interface* and Observable *class* in Java

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

Q. I smell something fishy, can you?

```
class Observable {
  void addObserver(Observer o);
  protected void clearChanged();
  int countObservers();
  void deleteObserver(Observer o);
  void deleteObservers();
  boolean hasChanged();
  void notifyObservers();
  void notifyObservers(Object arg);
}
```

Observer Pattern

- You might notice that there's an Observer *interface* and Observable *class* in Java

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

```
class Observable {
  void addObserver(Observer o);
  protected void clearChanged();
  int countObservers();
  void deleteObserver(Observer o);
  void deleteObservers();
  boolean hasChanged();
  void notifyObservers();
  void notifyObservers(Object arg);
}
```

Q. I smell something fishy, can you?

A. Observable really shouldn't be a class: it should be an **interface**

By locking ourselves into a class, we must fear the deadly diamond of death: interfaces are more reusable!

Observer Pattern

- You might notice that there's an Observer *interface* and Observable *class* in Java

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

```
class Observable {
  void addObserver(Observer o);
  protected void clearChanged();
  int countObservers();
  void deleteObserver(Observer o);
  void deleteObservers();
  boolean hasChanged();
  void notifyObservers();
  void notifyObservers(Object arg);
}
```

Q. I smell something fishy, can you?

A. Observable really shouldn't be a class: it should be an **interface**

By locking ourselves into a class, we must fear the deadly diamond of death: interfaces are more reusable!

Q. Admittedly, our NewsCafe suffers from this too: does this matter?

Observer Pattern

- You might notice that there's an Observer *interface* and Observable *class* in Java

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

```
class Observable {
  void addObserver(Observer o);
  protected void clearChanged();
  int countObservers();
  void deleteObserver(Observer o);
  void deleteObservers();
  boolean hasChanged();
  void notifyObservers();
  void notifyObservers(Object arg);
}
```

Q. I smell something fishy, can you?

A. Observable really shouldn't be a class: it should be an **interface**

By locking ourselves into a class, we must fear the deadly diamond of death: interfaces are more reusable!

Q. Admittedly, our NewsCafe suffers from this too: does this matter?

A. Not really: we were concerned with an instance of observer pattern, not a general framework

Push vs Pull Observer

Push vs Pull Observer

- So far we've been looking at a *pull* observer, pattern, where the *Reader* had to *pull* the articles from the *Newspaper*
- Another variation is the *push* observer pattern, where the subject *pushes* articles
- Ultimately, they achieve the same effect, but it's worth understanding them both, since there are different trade-offs

Push vs Pull Observer

Pull

```
class Subject { …
  void announce() {
    for (Observer observer : observers) {
      observer.update(this);
    }
  }
}
```

```
class Observer { …
  void update(Subject subject) {
      State state = subject.getState();
      …
  }
}
```

The observer **pulls** the state from the subject

- ➕ Observers extract only the state they require
- ➖ The Observer might be out of Sync when it does a pull

Push

```
class Subject { …
  void announce() {
    for (Observer observer : observers) {
      observer.update(getState());
    }
  }
}
```

The subject **pushes** the state to the observer

```
class Observer { …
  void update(State state) {
      …
  }
}
```

- ➖ More state than required might be passed around
- ➕ Every Observer receives up-to-date changes immediately