Software Development (cs2500)

Lecture 22: Interfaces and Polymorphsm

M. R. C. van Dongen

November 11, 2013

Outline

- We study the `Comparable` interface.
  - This is the basic interface for comparing things.
- We study callback methods.
  - Callbacks are the basis for a common design pattern.
  - The pattern is called the *Observer Design Pattern*.
  - Using this pattern, GUI applications can respond to events:
    - Pop up menu when a button is clicked;
    - Scroll text when scrollbar is moved;
    - Read key from keyboard when key is pressed;
    - Move mouse cursor when mouse is moved;
    - ...

Sorting

- Many applications require sorting.
  - Sorting a student list by surname or ID;
  - Sorting a dictionary alphabetically;
  - Sorting an index;
  - ...
- With a polymorphic sorting method, you could reuse them all.
  - You implement the sorting method in terms of an interface;
  - The method uses polymorphic variables to do the work.
- But how do you compare all these different objects?

The `Comparable` Interface

- In `Java` you compare objects with `compareTo( )`.
- This method is defined in the `Comparable` interface:
  - `public int compareTo( Object that );`
  - Should return negative value if `this` is smaller;
  - Should return positive value if `that` is smaller;
  - Should return zero it `this` and `that` are incomparable.

Additional Requirements

for all a, b, and c we must have:

Sign	◻ Sign of `a.compareTo( b )` should be equal to sign of `- b.compareTo( a );` ◻ Required.
Transitivity	◻ If `a.compareTo( b ) < 0 && b.compareTo( c ) < 0` then `a.compareTo( c ) < 0;` ◻ If `a.compareTo( b ) == 0`, then the signs of `a.compareTo( z )` and `b.compareTo( z )` should be equal; ◻ Required.
Consistency	◻ `(a.compareTo( b ) == 0) == a.equals( b ).` ◻ Recommened.

Ensuring these may be difficult for different object types.

Case Study

- Let’s implement `compareTo( )` for a `BankAccount` application.
- To compare two `BankAccount` objects, we simply compare their balances.

How to Implement Comparable

Without Polymorphism

Java

```java
public class BankAccount implements Comparable {
    private double balance;

    // omitted

    /**
     * Compare this instance with another instance of this class.
     * <bf>Note: should only be used to compare instances of this class.</bf>
     * @param other The other instance.
     * @return a negative value if this is less significant than other;
     *         a positive value is other is less significant than this;
     *         zero otherwise.
     */
    @Override
    public int compareTo( Object other ) {
        final BankAccount that = (BankAccount)other;

        return (this.balance < that.balance) ? -1 :
               (that.balance < this.balance) ? +1 : 0;
    }
}
```

Comparing Polymorphic Types

- The `BankAccount` class compared `BankAccounts` objects.
- It casted to `BankAccount` and then compared `balance` attributes.
- For general *polymorphic* instances this also works.
- For some application you can even implement `compareTo` with a polymorphic variable that corresponds to an interface.

Example

With Polymorphism

Java

```
public interface Animal extends Comparable {
    public String getName( );
}
```

Example Continued

With Polymorphism

Java

```
public class ConcreteAnimal implements Animal {
    // omitted

    @Override
    public int compareTo( Object that ) {
        return compareTo( this, (Animal)that );
    }

    public static int compareTo( final Animal first, final Animal second ) {
        return first.getName( ).compareTo( second.getName( ) );
    }
}
```

Example Continued

With Polymorphism

Java

```
public class Cat implements Animal {
    // omitted

    @Override
    public int compareTo( Object that ) {
        return ConcreteAnimal.compareTo( this, (Animal)that );
    }
}
```

Example Continued

With Polymorphism

Java

```
public class Dog implements Animal {
    // omitted

    @Override
    public int compareTo( Object that ) {
        return ConcreteAnimal.compareTo( this, (Animal)that );
    }
}
```

The Observer Design Pattern

- The *observer pattern* is a commonly used design pattern.
- It defines a one-to-many object dependency.
- The dependency ensures that the object's dependents are automatically updated when the object's state changes [Gamma et al. 2008].
- AKA Dependents, Publish-Subscribe [Freeman, and Freeman 2005, Pages 44–78], and Event-Listener.

The Observer Pattern (Continued)

The Source of the News: A Newspaper

- There is one `Subject`.
- There are zero or more `Observer`s.
- An `Observer` can be attached to the `Subject`.
- An `Observer` can be detached from the `Subject`.
- If the `Subject`'s state changes it updates all its `Observer`s.
  - This is done by calling each `Subject`'s `update( )` method.

The Observer Pattern (Continued)

Potential Readers

- There is one `Subject`.
- There are zero or more `Observers`.
- An `Observer` can be attached to the `Subject`.
- An `Observer` can be detached from the `Subject`.
- If the `Subject`'s state changes it updates all its `Observers`.
  - This is done by calling each `Subject`'s `update( )` method.

The Observer Pattern (Continued)

Subscribe as Reader to the Newspaper

- There is one `Subject`.
- There are zero or more `Observers`.
- An `Observer` can be attached to the `Subject`.
- An `Observer` can be detached from the `Subject`.
- If the `Subject`'s state changes it updates all its `Observers`.
    - This is done by calling each `Subject`'s `update( )` method.

The Observer Pattern (Continued)

Unsubscribe as Reader to the Newspaper

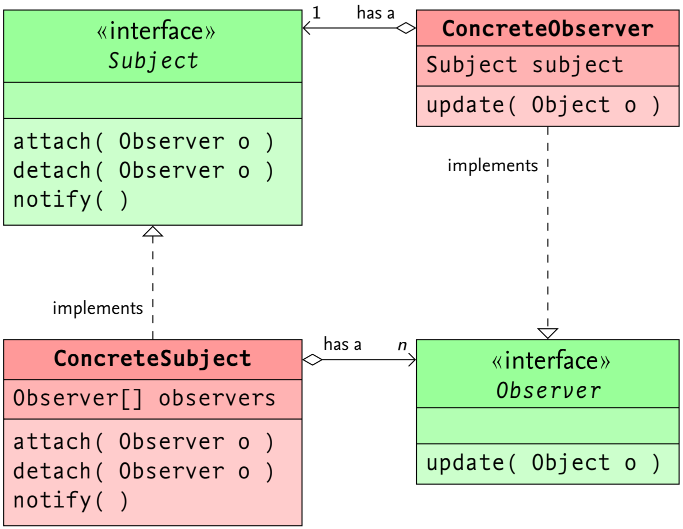
- There is one `Subject`.
- There are zero or more `Observers`.
- An `Observer` can be attached to the `Subject`.
- An `Observer` can be detached from the `Subject`.
- If the `Subject`'s state changes it updates all its `Observers`.
  - This is done by calling each `Subject`'s `update( )` method.

The Observer Pattern (Continued)

Inform the Readers about News

- There is one `Subject`.
- There are zero or more `Observers`.
- An `Observer` can be attached to the `Subject`.
- An `Observer` can be detached from the `Subject`.
- If the `Subject`'s state changes it updates all its `Observers`.
  - This is done by calling each `Subject`'s `update( )` method.

Class Diagram of the Observer Pattern



Case Study

- Let’s implement an example.
- We have a newspaper and readers of the newspaper.
- The readers can subscribe and unsubscribe.
- The newspaper informs the subscribers about new newsitems.

Implementing the Interfaces

Java

```
public interface Subject {
    // Subscribe to this newspaper.
    public void attach( Observer subscriber );
    // Unsubscribe from this newspaper.
    public void detach( Observer subscriber );
    // Notify this newspaper of a news event.
    public void notify( String event );
}
```

Java

```
public interface Observer {
    // Inform this subscriber about a published event.
    public void update( String event );
}
```

A Concrete Observer

Java

```
public class ConcreteObserver implements Observer {
    // The name of the subscriber.
    final String name;

    public ConcreteObserver( final String name ) {
        this.name = name;
    }

    // Inform this subscriber about a published event.
    @Override
    public void update( final String event ) {
        System.out.println( name + " reading: " + event );
    }

    @Override
    public String toString( ) {
        return name;
    }
}
```

A Concrete Subject

Java

```
public class ConcreteSubject implements Subject {

    // The name of this newspaper.
    private final String name;
    // The subscribers of this newspaper.
    private final ArrayList<Observer> subscribers;

    public ConcreteSubject( final String name ) {
        subscribers = new ArrayList<Observer>( );
        this.name = name;
    }

    @Override
    public String toString( ) {
        return name;
    }

    // omitted

}
```

A Concrete Subject (Continued)

Java

```
public class ConcreteSubject implements Subject {

    // omitted

    @Override // Subscribe a new customer.
    public void attach( final Observer subscriber ) {
        System.out.println( subscriber + " subscribed to " + this );
        subscribers.add( subscriber );
    }

    @Override // Unsubscribe an existing customer.
    public void detach( final Observer subscriber ) {
        System.out.println( subscriber + " unsubscribed from " + this );
        subscribers.remove( subscriber );
    }

    @Override // Inform this newspaper about hot news item.
    public void notify( final String news ) {
        // Inform all subscribers about the news item.
        System.out.println( this + " got news item: " + news );
        for( Observer subscriber : subscribers ) {
            subscriber.update( news );
        }
    }

}
```

The Main Class

Java

```java
public class Main {
    public static void main( String[] args ) {
        final Subject eolas = new ConcreteSubject( "Eolas" );
        final Subject examiner = new ConcreteSubject( "Examiner" );

        final Observer john = new ConcreteObserver( "John" );
        final Observer jane = new ConcreteObserver( "Jane" );
        final Observer eoin = new ConcreteObserver( "Eoin" );

        examiner.attach( john );
        examiner.attach( eoin );
        examiner.attach( jane );
        eolas.attach( jane );

        eolas.notify( "Assignment 2 handed back this Wednesday." );
        examiner.notify( "100 Jobs to be created by Indeed.com." );

        examiner.detach( jane );

        examiner.notify( "No news today." );
    }
}
```

Sample Output

Unix Session

```
$
```

Sample Output

Unix Session

```
$ java Main
```

Sample Output

Unix Session

```
$ java Main
John subscribed to Examiner
Eoin subscribed to Examiner
Jane subscribed to Examiner
Jane subscribed to Eolas
Eolas got news item: Assignment 2 handed back this coming Wednesday.
Jane reading: Assignment 2 handed back this coming Wednesday.
Examiner got news item: 100 Jobs to be created by Indeed.com.
John reading: 100 Jobs to be created by Indeed.com.
Eoin reading: 100 Jobs to be created by Indeed.com.
Jane reading: 100 Jobs to be created by Indeed.com.
Jane unsubscribed from Examiner
Examiner got news item: Sorry folks: No news today.
John reading: Sorry folks: No news today.
Eoin reading: Sorry folks: No news today.
$
```

For Friday

- Study [Horstmann 2013, Sections 8.4–8.5].
- We postpone [Horstmann 2013, Section 8.6 and further] until next year.
    - If you're interested, read [Horstmann 2013, Section 8.6 and further].

Acknowledgements

- This lecture corresponds to[Horstmann 2013, Sections 8.4 and 8.5].
- [Freeman, and Freeman 2005, Pages 44–78]
- Gamma et al. [2008] is the Bible of all Design Patterns.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.