Recap

- We addressed the following items last week
- Notion of *design patterns*
  - **Composite**: compose objects in hierarchies
- UML Notation
  - Class Diagrams: Static relationship between classes

Objectives

- Continue talking about design patterns
  - Observer
  - Strategy
  - Singleton

Design Patterns

- Patterns catalog a large amount of accumulated knowledge about designing systems
- *Discovered* rather than *invented*
- Clever uses of language features to make behaviors reconfigurable at *runtime*

Design Pattern[1]

Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

[1] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

The Observer Pattern

- Way to implement a publisher/subscriber relationship between objects
- Classes Involved:
  - Observable, Publisher, Subject
  - Observer, Subscriber, Listener
- Picture mechanism in two ways: Listeners *listen* for changes in Subject **or** Subject *notifies* Listeners of changes

Observer[2]

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

[2] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

Example

- For our example we will think of the subject as a sensor and the listeners some objects that need to be updated when the sensor detects an event
- Sensor will have the responsibility of updating its listeners of changes
- Requirements:
  - Interface for all listeners
  - Interface for the Subject
  - Subject needs to keep track of its listeners
- Gives enough information to sketch out a basic class diagram

Ordered Interactions

- UML *Sequence Diagrams* allow us to model object interactions ordered in time
- Elements of Sequence Diagrams
  - Time runs down the page
  - Each object represented by a vertical line
  - *Actor* represented by a stick figure
  - Interactions between objects are represented by horizontal arrows
- In sequence diagrams only the order of interactions is shown

Simulating Events in a Separate Thread

- Create a new thread to create events randomly
- Implement the Runnable interface

```
@Override
public void run() {

int n = 20;

   try {
      while(!ended) {
         // System.out.format("Loop Number %d \n", n);
         n = n-1;
         if (n < 1) {
            ended=true;
         }

      Thread.sleep(1000);

      if (Math.random() > 0.6) {
         System.out.println("SensorEnv: Event! :)");
         sens.eventHappened();
      } else {
         System.out.println("SensorEnv: No Event! :(");
      }

      } // while
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

Main Client

- The `main` method in our client application creates the subject and listener objects
- A new thread is launched to simulate events

```
public class SensorClient {

   public static void main(String[] args) {

      SensorSubject sens = new SensorSubject();
      SensorListener o1 = new SensorListener("Listener 1",sens);
      SensorListener o2 = new SensorListener("Listener 2",sens);

      Thread t = new Thread(new SensorEnv(sens));
      t.start();
   }
}
```

Sensed and Listens interfaces

- Interfaces are defined for both the publishers and subscribers
- Sensor needs to be able to register listeners and notify them
- The Listeners have one method for being updated

```
public interface Sensed {
   public void addListener(Listens o);
   public void removeListener(Listens o);
   public void notifyListeners();
}

public interface Listens {
   public void update();
}
```

Java Collections


```
private Set<Listens> sensorListeners;
// ...
this.sensorListeners = new HashSet<Listens>();
```

- Java Collections contain a number of *interfaces* for data structures: `List`,`Map`,`Queue`,`Set`
- Uniqueness matters for `Set`
- With `new` need one of the implementations such as `HashSet` or `ArrayList`

Exercise with Observer

- A number of classes have been given for demonstrating the observer pattern
- Remaining classes that are needed are the concrete implementations of the concrete `SensorSubject` and `SensorListener`

```
SensorEnv: No Event! :(
SensorEnv: No Event! :(
SensorEnv: Event! :)
SensorSubject: 1 Events Have Happened!
SensorListener: Update detected by Listener 2!
SensorListener: Update detected by Listener 1!
SensorEnv: No Event! :(
SensorEnv: No Event! :(
SensorEnv: No Event! :(
SensorEnv: No Event! :(
SensorEnv: Event! :)
SensorSubject: 2 Events Have Happened!
SensorListener: Update detected by Listener 2!
SensorListener: Update detected by Listener 1!
SensorEnv: No Event! :(
SensorEnv: No Event! :(
SensorEnv: No Event! :(
```

Coupling and Cohesion[3]

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

General Design Principle

Aim for *low coupling* and *high cohesion*

- The Observer pattern demonstrates the power of loosely coupled designs
- Coupling refers to how much one object has to know about another object to interact
- The subject cares that listeners implement the `Listens` interface
- Cohesion refers to how much a class does a single thing

[3] Eric Freeman et al. *Head first design patterns*. O'Reilly Media, Inc., 2004.

Strategy Pattern Example

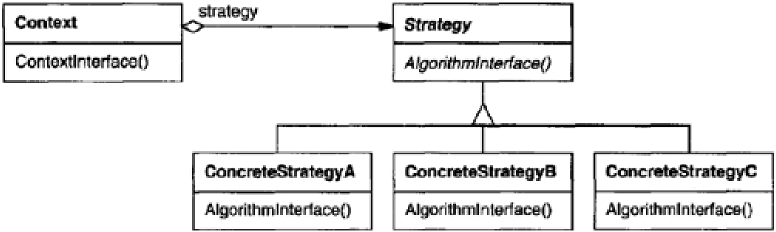
- *Strategy* is about being able to reconfigure algorithms at runtime
- Use encapsulation and delegation to make algorithms interchangeable
- In GoF, Strategy is motivated by an example of putting line breaks in text.
- We don't want to code the line break functionality in this class. We would like to make it more flexible

```
public class MyText {
    private String buff;

    public MyText(String buff) {
        this.buff = buff;
    }

    public String makeLineBreaks() {
        return // Implement linebreaking here
    }
}
```

Problems with a Simple Solution

- Linebreaking functionality baked into the `MyText` class
- Difficult/Awkward to change the linebreaking algorithm
  - Need to make changes to our main class
  - Switching between behaviors will require some fields and conditional/case statements
- Instead, abstract the algorithm into a separate interface

Strategy Design Pattern [4]



- An example of "favor composition over inheritance"
  - Get new linebreak behaviors by putting objects together, not subclassing
- Addresses the possibility of any future changes to the algorithm that may be needed

[4] Class diagram from Gamma, *Design patterns: elements of reusable object-oriented software*, op. cit.

- The code below shows how the `MyText` class might look if we used strategy to vary the line-breaking behavior

```
public class MyText {
    private String buff;
    private BreakBehavior b;

    public MyText(String buff, BreakBehavior b) {
        this.buff = buff;
        this.b = b;
    }

    public String makeLineBreaks() {
        return b.linebreak(buff);
    }

    public void setB(BreakBehavior b) {
        this.b = b;
    }
}
```

Linebreaks with Strategy (2)

- To encapsulate a particular algorithm, write a class that implements the behavior interface

```
import org.apache.commons.text.WordUtils;

public class SimpleBreakBehavior implements BreakBehavior {
    @Override
    public String linebreak(String s) {
        return WordUtils.wrap(s, 10);
    }

}
```

- We can write as many implementations of the algorithm as we need
- Easily change them at runtime using the setter method in the `MyText` class

```
public class NoBreakBehavior implements BreakBehavior {

    @Override
    public String linebreak(String s) {
        return s;
    }

}
```

Linebreaks with Strategy (4)

- The listing below shows how we might actually use the `MyText` class and change the runtime behavior

```
public class StrategyPatternExample {
    public static void main(String[] args) {
        MyText t = new MyText("This is a sentence.",
                              new SimpleBreakBehavior());

        System.out.println("---\nWith simple line breaks \n");
        System.out.println(t.makeLineBreaks());

        t.setB(new NoBreakBehavior());

        System.out.println("---\nWith no line breaks \n");
        System.out.println(t.makeLineBreaks());
    }
}
```

The Singleton Pattern

- The singleton pattern is used when we want to have a unique instance of a class and provide a global point of access to it.
- The pattern itself only consists of a single class
- Classic implementation of the singleton uses a private constructor and a static variable

Singleton

"Ensure a class has only one instance, and provide a global point of access to it." *GoF*

Restricting Object Creation

```
public class Singleton {
    //

    private Singleton() {}

    //
}
```

- The classic singleton implementation uses a `private` constructor
- What does this do?

Providing access to the singleton instance

```
public class Singleton {

    private Singleton() {}

    public static Singleton getInstance() {}
}
```

- What should be the contents of the `getInstance()` method?

The getInstance() method

```
public class Singleton {

    private Singleton() {}

    public static Singleton getInstance() {
        return new Singleton();
        }
}
```

- What *type* of static variable do we need, and how can we ensure that only one object can be created.

The Complete Singleton Implementation


```
public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
            }

        return uniqueInstance;
        }
}
```

- Here we have all of the basic elements of the singleton pattern

Issues with Singleton

- Uses
  - Can be useful if we need some kind of global state in an application
  - Factory patterns often employ a singleton
- Issues
  - Usefulness or appropriateness of singletons are debated
  - Sometimes called an *antipattern*
  - Issue with multi-threading

Dealing with multi-threading


```
public static synchronized Singleton getInstance() {
   if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
      }

   return uniqueInstance;
   }
```

- One option to deal with multi-threading problems is to declare the `getInstance()` method `synchronized`
- The synchronized keyword basically ensures that method calls are atomic