CS1530, LECTURE 18: DESIGN PATTERNS

Bill Laboon

WHAT ARE DESIGN PATTERNS?

- Common ways of performing tasks that are:
    - Above statement level
    - Below system level

TRIVIAL EXAMPLE - NULL CHECK

```
public int doSomething(Foo foo) {
   if (foo == null) {
       return -1;
   } else {
       return foo.calculate();
   }
}
```

DESIGN PATTERNS – A LANGUAGE FAILURE?

- In some sense, a design pattern can mean that a fundamental way of specifying behavior in a language is *missing*!
- Example: the null check pattern is not common in Rails.
- Why? There's a way to do it with one method on ANY object (see next slide).
- It was known to be a common pattern, and so built in to the language (or at least an extension of the language) to avoid programmers repeating themselves

“NULL CHECK” PATTERN IN RAILS

```
irb(main):008:0> a = nil
=> nil
irb(main):009:0> b = 1
=> 1
irb(main):010:0> b.negative?
=> false
irb(main):011:0> a.negative?
NoMethodError: undefined method `negative?' for nil:NilClass
  from (irb):11
  from /usr/local/lib/ruby/gems/2.3.0/gems/railties-5.0.0.1/
lib/rails/commands/console.rb:65:in `start'
 ... STACK TRACE ...
  from /usr/local/Cellar/ruby/2.3.1/lib/ruby/2.3.0/rubygems/
core_ext/kernel_require.rb:55:in `require'
  from -e:1:in `<main>'
irb(main):012:0> a.try(:negative?)
=> nil
```

ON THE OTHER HAND…

- We will always be discovering how to do things in our languages which were not thought of by the language specifiers
- Sometimes it is more trouble than it is worth to re-write our entire language every time we think of a better way to do things

ON USING PATTERNS

- Patterns are very useful!
    - Complexity reduced - abstractions already exist and are commonly known by developers
    - Reduce errors because common solutions are known and can be easily implemented
    - Provide heuristics (“rules of thumb”) by allowing us to think what patterns match what we need the software to do
    - Make for easier communication because we can discuss problems at higher level of abstraction

SPECIFIC DESIGN PATTERNS

- These are all from the popular "Gang of Four" book, *Design Patterns: Elements of Reusable Object-Oriented Software*
- If you are going to be a Java, C++, or C# developer, I recommend this book!
- Code Complete references some of these, but does not go into detail

LIST OF DESIGN PATTERNS WE WILL COVER

- Utility
- Singleton
- Factory
- Decorator
- Pool
- Iterator
- Strategy
- Null Object
- *Note that this is not a complete list!*

A NOTE ON PATTERN CODE EXAMPLES

- ☠️☢️⚠️☣️⚡
- NONE OF THE CODE IN THE FOLLOWING SLIDES IS THREAD-SAFE
- IT IS JUST EXAMPLE CODE
- IT WAS WRITTEN WITH AN EYE TOWARDS UNDERSTANDABILITY, NOT PERFORMANCE OR SAFETY
- USE AT YOUR OWN RISK

UTILITY PATTERN

- Perhaps the simplest pattern - a class which does not allow instantiation, and does not keep track of state
  - e.g. Math m = new Math()? Totally unnecessary
  - Make constructor private to avoid instantiation
- Useful for utility methods, e.g. tools, pure functions
- All static methods
- Has no state! Everything must be passed in as a param to a particular method

UTILITY EXAMPLE

```
public static class Math {

  private Math() {
      // Private constructor, can't instantiate
   }

   public static int square(int n) {
       return n * n;
   }

   public static int cube(int n) {
      return n * n * n;
   }
}
```

SINGLETON PATTERN

- Used when you want to ensure there is only one instance of a given class
- Can use lazy evalutation - not generated until needed
- Why use this instead of Utility? Can implement interfaces and state.
- Use for: database or other connections, any time you want to ensure a single instantiation

SINGLETON PATTERN EXAMPLE

```
// THERE CAN BE ONLY ONE

public class ProfessorLaboon {
   private static ProfessorLaboon _instance =
         null;

   private ProfessorLaboon() {  }

   public static ProfessorLaboon getInstance() {
      if (_instance == null) {
         _instance = new ProfessorLaboon();
      }
      return _instance;
   }

}
```

BENEFITS AND DRAWBACKS

- Drawbacks:
  - Can often just use static methods
  - Worries about thread safety
  - Makes unit testing more difficult, since global state introduced
- Benefits
  - Provide singular access to a single resource (e.g. log file)
  - Provides it as an object instead of in the static context

FACTORY PATTERN

- A class that exists to generate instances of other classes
- Different versions of a method return different classes
- Instantiation covered by the Factory
  - Thus, can hide instantiation details
  - For example, give me “something which implements this interface” - can change the specific class of the object sent back, hiding implementation details

FACTORY PATTERN EXAMPLE

```
public interface Bird {
   public void tweet();
}

class Duck implements Bird {
   public void tweet() { ... }
}

class Sparrow implements Bird {
   public void tweet() { ... }
}

class Cockatiel implements Bird {
   public void tweet() { ... }
}
```

FACTORY PATTERN EXAMPLE CODE, CONTINUED

```
class BirdFactory {
   public static getBird(String description) {
      if (description.equals("fluffy")) {
         return new Cockatiel();
      } else if (description.equals("small")) {
         return new Sparrow();
      } else {
         return new Duck();
      }
   }
}
```

DECORATOR

- Dynamically decorate an object from its base
- Generates a new decorated object
- This new object will have additional functionality “built in” to it
- We start with a base class, and “decorate it”

DECORATOR PATTERN EXAMPLE

```
interface Coyote {
   public void chase();
}

class SimpleCoyote implements Coyote {
    public void chase() {
      System.out.println("Chase!");
    }

}
```

DECORATOR PATTERN EXAMPLE, CONTINUED

```
abstract class CoyoteDecorator implements Coyote {

    protected Coyote _decoratedCoyote;

    public CoyoteDecorator(Coyote c) {
        _decoratedCoyote = c;
    }


    public void chase() {
        _decoratedCoyote.chase();
    }
}
```

DECORATOR PATTERN EXAMPLE, CONTINUED

```
class CoyoteRocketDecorator extends CoyoteDecorator {
    public CoyoteRocketDecorator(Coyote c) {
        super(c);
    }

    @Override
    public void chase() {
        super.chase();
        chaseOnRocket();
    }

    private void chaseOnRocket() {
       System.out.println("Chase on Rocket!");
    }

}
```

DECORATOR PATTERN EXAMPLE, CONTINUED

```
public class DecoratorDemo {
   public static void main(String[] args) {
      Coyote c1 = new SimpleCoyote();
      c1.chase();

      Coyote c2 = new CoyoteRocketDecorator(new SimpleCoyote());
      c2.chase();

      Coyote c3 = new CoyoteRocketDecorator(
        new CoyoteRollerSkatesDecorator(new SimpleCoyote()));
      c3.chase();


      Coyote c4 = new CoyoteRocketDecorator(
                  new CoyoteRocketDecorator (
                  new CoyoteRocketDecorator(
                  new CoyoteRollerSkatesDecorator(
                  new SimpleCoyote()))));
      c4.chase();
    }

}
```

DECORATOR PATTERN EXAMPLE, CONTINUED

```
class CoyoteRollerSkatesDecorator extends CoyoteDecorator {
    public CoyoteRollerSkatesDecorator(Coyote c) {
   super(c);
    }
    @Override
    public void chase() {
        super.chase();
        chaseOnRollerSkates();
    }
    private void chaseOnRollerSkates()
{ System.out.println("Chase on Roller Skates!"); }
```

POOL

- Store a bunch of objects around, can re-use
- Can create ahead of time
  - Lazy - generate objects only when necessary
  - **Eager - generate objects well beforehand**
- The items in the pool can be Threads, Objects, anything where you are more concerned about the time from when you need something to when you get it, than startup time

POOL EXAMPLE

```
public class RolexPool {

    private ArrayList _rolexes =
              new ArrayList<Rolex>();

    public Rolex getRolex() {
        if (_rolexes.isEmpty()) {
            // Generate some more Rolexes
        }
        // Get and return the first Rolex
    }

    public void returnRolex(Rolex r) {
       _rolexes.add(r);
    }

}
```

STRATEGY PATTERN

- Allows you to select which algorithm you want to use to achieve a similar results
- For example, in some circumstances, different sorting algorithms are better or worse
  - Concerned about worst-case scenarios? Use HeapSort
  - Concerned about average case? Use QuickSort
  - Working a small number of items? Use Insertion Sort
  - etc.

STRATEGY PATTERN EXAMPLE

```
public interface Sorter {
   public int[] sort(int[] intsToSort);
}

public class BubbleSortSorter Sorter {
    public int[] sort(int[] intsToSort) {
        // does bubble sort
    }
}

public class MergeSortSorter implements Sorter
    public int[] sort(int[] intsToSort) {
        // does merge sort
    }
}
```

STRATEGY PATTERN EXAMPLE, CONTINUED

```
public class Database {

    private Sorter _sorter = null;
    private int[] _storedInts = null;

    public Database(Sorter s, int[] someInts) {
        _sorter = s;
        _storedInts = someInts;

    }

    public int[] sortIntegersInDatabase() {
       _sorter.sort(_storedInts);
    }

}
```

NULL OBJECT PATTERN

- Instead of returning a null pointer reference, return an object with methods that don't do anything, or that return default values
- Avoids NullPointerExceptions - for a given type of object, we should never refer to any possible item as a “true” null, but rather our “null” version of the object

NULL OBJECT PATTERN EXAMPLE

```java
public interface Bird {
    public void chirp();
}

public class Turkey implements Bird {
    public void chirp() {
        System.out.println("Gobble gobble!");
    }
}

public class NullBird implements Bird {
    public void chirp() {
        // Do nothing

    }

}
```

NULL OBJECT PATTERN EXAMPLE, CONT'D

```
// Get the bird in the cage - may or may not
// exist
Bird b = cage.getBird();

// Old way
// if (b == null) {
//     // do nothing
// } else {
//    b.chirp();
// }

// With Null Object pattern
// No need to null check because methods just
// won't do anything
b.chirp();
```