Design by Abstraction

CS151

Chris Pollett

Oct. 17, 2005.

Outline

- Slightly more on JUnit
- Design Pattern -- Singleton
- Refactoring
- Design Pattern -- Template Method

Slightly more on JUnit

- Last day, we gave the basic format of a JUnit test class and how to compile and use JUnit.
- Let’s look at an example of a test case

```
public void testRemove()
{
        LinkedList l = new LinkedList();
        for( int i = 1; i <= 7; i++)
        {
                l.insertLast(new Integer(i));
        } // creates a list with 1,2,3,4,5,6,7
        l.removeHead(); // list now 2,3,4,5,6,7
        l.removeLast(); // list now 2,3,4,5,6
        l.remove(2); // list now 2,3,5,6
        assertTrue(TestUtil.match(l, TestUtil.toIntegerArray(new
            int[]{2,3,5,6})); /* code for TestUtil is on page 238 but basically
                        matches the anonymous array against list contents*/
}
```

Design Patternss

- The book Design Patterns by [Gamma et al] define a list of common object oriented designs to solve programming problems which arose frequently.
- These were classified into three groups: creational patterns, structural patterns, and behavioral patterns.
- In specifying a pattern as in their book, one needs to give the following information:
  - Pattern name
  - Category: Creational, structural, or behavioral
  - Intent: a short description of the design issue or problem to be addressed.
  - Also known as: (optional) other names for the pattern
  - Applicability: Situations in which the pattern can be applied
  - Structure: UML diagram
  - Participant: list of classes and or objects involved in the pattern.

Design Pattern -- Singleton

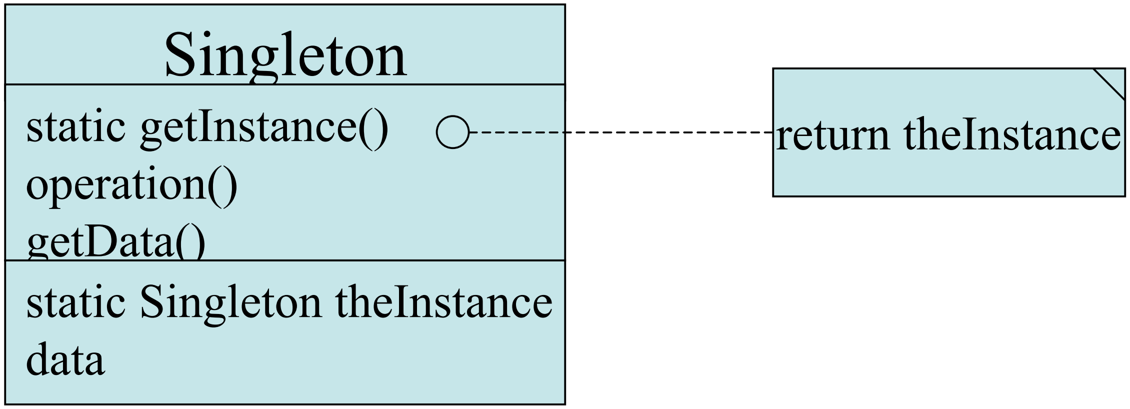
- We have already discussed the singleton pattern. Here's how it might be described according to the previous slides guidelines.

Design Pattern: Singleton

Category: Creational Design Pattern

Intent: Ensure that a class has only one instance and provide a global point of access to it.

Applicability: Use the Singleton pattern when there must be exactly one instance of a class and it must be accessible to clients from a well-known access point.



Participant: Singleton declares the unique instance of the class as a static variable, and defines a static method getInstance() for clients to access the unique instance. (could then give a code fragment)

Designing Generic Components

- A generic component is a set of classes or packages that can be extended or adapted, and reused in a variety of contexts.
- Along with using design patterns creating generic components is an important part of code reuse.
- They are also known as reusable components.
- We now discuss a way of finding generic components called refactoring.

Refactoring

- This consists of:
  - identifying code segments in a program that implement the same logic, often in the same exact code in different places (such code is hard to maintain).
  - Capture this logic in a generic component once.
  - Restructure the code so that every occurrence of the code segment uses the generic component.

Refactoring Method Invocation

- Rewrite:

```
class A
{
    void method1(…){//…
      step1(); step2();step3(); //….
    }
    void method2(…){//…
      step1(); step2();step3(); //….
    }
 //…
}
```

- As:

```
class A
{
    void computeAll()
    {step1(); step2();step3();}
    void method1(…){//…
      computeAll();
       //….}
    void method2(…){//…
       computeAll();
    //….
    }
 //…
}
```

Refactoring by Inheritance

- Might have two classes:

```
class A
{
    void method1(…){//…
      step1(); step2();step3(); //….
    }
//…
}
class B
{
    void method1(…){//…
      step1(); step2();step3(); //….
    }
//…
}
```

- Make a common class:

```
class Common
{
    void computeAll(…){
      step1(); step2();step3();}
}
class A extends Common
{
    void method1(…){//…
      computeAll() //….
    }
//…
}
class B extends Common
{
    void method1(…){//…
      computeAll() //….
    }
//…
}
```

Refactoring by Delegation

- Solves same problem as refactoring by inheritance, except now rather than have A and B extend Common, A and B each create an instance of Common c.

```
class A
{
     void method1(…){//…
        c.computeAll(); //….
     }
//…
}
```

```
class B
{
        void method1(…){//…
        c.computeAll();//….
        }
//…
}
```

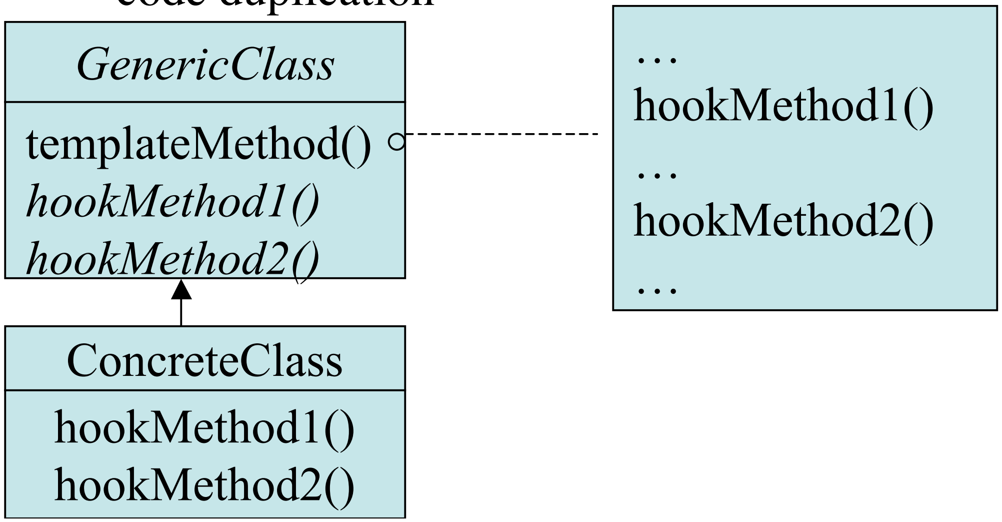
Design Pattern -- Template Method

Category: Behavioral

Intent: To define the skeleton of an algorithm in a method, deferring some steps to subclasses, thus allowing the subclasses to redefine certain steps of the algorithm

Applicability: The template method pattern should be used:

- to implement the invariant parts of an algorithm once and leave it to the subclasses to implement behavior that can vary
- to refactor and localize the common behavior among subclasses to avoid code duplication


GenericClass
templateMethod()
hookMethod1()
hookMethod2()
ConcreteClass
hookMethod1()
hookMethod2()
…
hookMethod1()
…
hookMethod2()
…