From Building Blocks to Projects

CS151

Chris Pollett

Oct. 10, 2005.

Outline

- Design and Implementation of Classes
- Contracts and Invariants
- Design by Contract

Public and Helper Classes

- There are two kinds of classes those for general use (**public classes**), and those that are used solely to help implement other classes (**auxiliary or helper classes**).
- Usually only the former are declared public.
- Public classes live in a file with the same name. i.e., Point is defined in Point.java
- A helper class which only helps one public class, say A, in a package usually is defined in the same file as A. For example, Node in a LinkedList implementation of the List interface.
- If a helper class supports several public classes in a package it may reside in a file by itself.

Class Members

- The order of the class members does not matter in Java.
- However, it is good practice to order classes so other programmers can easily follow your code.
- In particular, for this class we'll use the order:

```
public class AClass
{
    <public constants (public static final fields)>
    <public constructors>
    <public accessors -- methods to access, not modify object state>
    <public mutators -- methods which modify object state>
    <non-public fields>
    <non-public auxiliary methods or nested classes>
}
```

Design Guidelines

- Avoid nonfinal public fields, except when a class is final and the field value can be set to anything (unconstrained).
  - To read an attribute X, use a method getX() (accessor)
  - To set the value of an attribute X, use a method setX(value) (mutator)
- Completeness of the Public Interface
  - Make sure your class has a complete set of accessors and mutators, remembering people might subclass your class in unexpected ways.
- Separate Interface From Implementation
  - If the functionality of a class can be implemented in several different ways, create an interface first, then implement that.

Documenting Source Code

- Java has a facility called **javadoc** to make it easy to generate documentation along with the development of the source code.
- A javadoc comment looks like /** some comment */ (notice the two *'s). It also might include special tags of the form @some_tag.
- At the command line one can type:

javadoc name_of_java_program.java

and javadoc will strip out these comments and generate a nicely formulated HTML page.

- You should use javadoc comments:

(1) at the start of a class

```
/**
        Description of some class

        @author Chris Pollett
        @version 2005.10.09
        @since JDK 1.0
*/
```

(2) before each public method.

```
/**
        Description of method
        @param variable_name - what variable used for
        …
        @return - what is return by this method
        @throws - some exception thrown by the method
        @see - some related method (optional)
*/
```

Contracts of Methods

- Method declarations in an interface only define the *types* of the methods not their *behaviors*.
- A *contract* of a method specifies its behavior.
- Contracts are often specified informally or not at all. This can lead to the following problems:
  - Incompleteness or silence on some aspects of behavior
  - Ambiguity and multiple interpretations
  - Contradictions with other contracts
- A formal contract specification language like Larch, Z, or VDM can sometimes solve these problems.
- For this class, we will take a simplified approach specifying contracts with pre- and post-conditions.

More on Contracts

- A *precondition* is a boolean expression which must hold when a method is invoked.
- A *postcondition* is a boolean expression which must hold when a method returns.
- In a javadoc comment before a method we could list these conditions with the made up tags @pre and @post.

```
/**
    @pre precondition_1
    …
    @pre precondition_n
    @post postcondition_1 …
*/
```

Example

```
/**
      Returns true if and only if the list is empty

      @pre true
      @post @result <=> size() > 0
      @post @nochange
*/
public boolean isEmpty()
```

- @result means the return value, @nochange means the state of the object is unchanged by this method. => mean implies, <=> means if and only if

More Sophisticated Contracts

- When specifying a mutator, one often needs to distinguish between the values of expression before the method was applied.
- We'll write this as:

  Expression @pre
- To specify contracts involving collections like lists it also useful to be able to quantify over the elements in the list. We'll do this with the notations like:

  @forall x: *Range @ Expression*

  @exists x: *Range @ Expression*

  Here Range must be of the form (a) [m..n] to indicate values between m and n, (b) an expression which evaluates to a Collection, Enumeration, or Iterator, or (c) a class name.

A More Sophisticated Example

```
/**
   Inserts a new element at the head of the list

   @pre item != null
   @post size() == size()@pre + 1
   @post item@pre == element(0)
   @post @forall k : [1 .. size() - 1]
                  @ element(k-1)@pre == element(k)
*/
public void insertHead(Object item);
```

Invariants of Classes

- A state of an object is *transient* if one or more of its methods are being executed.
- A state of an object is *stable* if it has been initialized (by calling one of its constructors), but none of its methods are currently executing.
- An *invariant* of a class is a formally specified condition that always holds on any object of the class whenever it is in a stable state.
- Given the invariants of a class, an object of the class is in a *well-formed* state if the invariants hold on that state.

Specifying Invariants

- The key characteristics of the doubly linked list representation in the book can be captured by:
  1. If the list is empty, both head and tail must be null.
  2. If the list is not empty, the head field points to the first node in the list, and the tail points to the last node.
  3. The count field should equal the number of nodes reachable by following the next link from the head of the list.
  4. For each reachable node, the following two things hold: (a) prev of next and next of prev point to the current node; (b) prev of the head and next of tail are null.
- One could write a little function _wellformed() to check these conditions. Then in our javadoc comment for the whole LinkedList class we could have the line:

  @invariant _wellformed()

Preserving Invariants

- Given a class, for the implementation to preserve the class' invariants it must:
  - **Establish invariants by public constructors** -- each public constructor must imply as a postcondition that each of the invariants holds
  - **Preserving invariants by public methods** -- each public method of the class can be assumed to have the invariants as a precondition, and the invariants must be implied by postconditions of the method.

Assertions

- An *assertion* is a boolean condition at a given location of a program which should be true whenever the flow of execution reaches that location.
- Since JDK 1.4, Java supports assertions with the syntax:

  assert *Assertion*;
- If the Assertion evaluates to false a AssertionError exception is thrown; true assertions have no effect.
- Assertions can be used to check pre- and post-conditions as well as invariants.
- Assertions on preconditions should be at the start of the method, for postconditions assertions should be at the return points of the methods and for invariants there should assertions at both the entry and exit points.
- To compile code with assertion you need to use:

  javac -source X.Y filename.java

  java -ea filename

  where X.Y is 1.4 or 1.5

Example

```
/**
        Returns the first element of the list.

        @pre !isEmpty()
        @post @result == element(0)
*/
public Object head()
{
        assert !isEmpty();
        Object result = (head != null ? head.item : null);
        assert result == element(0);
        return result;
}
```

Design By Contract

- One would like it to be the case that anytime a class C is implemented/extended by some subclass S. That S can be freely used wherever something of type C is called.
- To ensure things behave correctly at run-time, we need for S to honor the contract of C. This is the idea of designing by contract. Specifically,
  - we need that the precondition of each method of S to be no stronger than the precondition of that method in C.
  - we need that the postcondition of each method of S to be at least as strong as the postcondition of that method in C.