More Class Design

Lecture 11, Fri Jan 29 2010

borrowing from slides by Paul Carter and Wolfgang Heidrich

http://www.cs.ubc.ca/~tmm/courses/111-10
Reminders

- Assignment 1 due Wed 5pm
- TA office hours in DLC
- Check your ugrad email account regularly (or forward to active account)
  - grade info will be sent there
Exam

- Midterm reminder: Mon Feb 8, 18:30 - 20:00
  - FSC 1005
  - exam will be one hour, extra time is just in case needed
  - I'll discuss coverage next time

- DRC: Disability Resource Center
  - academic accommodation for disabilities
  - forms due one week before exam (Monday!)
  - http://students.ubc.ca/access/drc.cfm
Recap: Public vs Private

- **public** keyword indicates that something *can* be referenced from outside object
  - can be seen/used by client programmer
- **private** keyword indicates that something *cannot* be referenced from outside object
  - cannot be seen/used by client programmer
Recap: Designing Classes

- Blueprint for constructing objects
  - build one blueprint
  - manufacture many instances from it
- Consider two viewpoints
  - client programmer: want to use object in program
    - what public methods do you need
  - designer: creator of class
    - what private fields do you need to store data
    - what other private methods do you need
Public vs. Private Example

```java
public class Die {
    ...
    public int roll()
        ...
    private void cheat(int nextRoll)
        ...
    }
```
Public vs. Private Example

Die myDie = new Die();

int result = myDie.roll(); // OK
myDie.cheat(6);           //not allowed!
Unified Modeling Language

- Unified Modeling Language (UML) provides us with mechanism for modeling design of software
  - critical to separate design from implementation (code)
  - benefits of good software design
    - easy to understand, easy to maintain, easy to implement
- What if skip design phase and start implementing (coding)?
  - code difficult to understand, thus difficult to debug
- We’ll use UML class diagrams represent design of our classes
- Once the design is completed, could be implemented in many different programming languages
  - Java, C++, Python,...
### UML Visual Syntax

- + for public, - for private
- fields above, methods below

<table>
<thead>
<tr>
<th>Classname</th>
<th>fields</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ field: type</td>
<td></td>
<td>+ Classname()</td>
</tr>
<tr>
<td>- field: type</td>
<td></td>
<td>+ method(): return type</td>
</tr>
<tr>
<td>+ Classname()</td>
<td></td>
<td>+ method(param1 type,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>param2 type): return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- method(): return type</td>
</tr>
</tbody>
</table>
UML for Die

UML diagram for Die class we designed

<table>
<thead>
<tr>
<th>Die</th>
</tr>
</thead>
<tbody>
<tr>
<td>- sides: int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Die()</td>
</tr>
<tr>
<td>+ setSides(numSides: int): void</td>
</tr>
<tr>
<td>+ roll(): int</td>
</tr>
</tbody>
</table>
Separation and Modularity

- Design possibilities
  - Die and RollDice as separate classes
  - one single class that does it all
- Separation allows code re-use through modularity
  - another software design principle
- One module for modeling a die: Die class
- Other modules can use die or dice
  - we wrote one, the RollDice class
- Modularization also occurs at file level
  - modules stored in different files
  - also makes re-use easier
Control Flow Between Modules

- Last week was easy to understand control flow: order in which statements are executed
  - march down line by line through file
- Now consider control flow between modules

Client code

```java
int rollResult;
myDie.setSides();
rollResult = myDie.roll();
```

Die class methods

```java
public int roll()
{
    ...
}

public void setSides()
{
    ...
}
```
Designing Point: UML

- class to represent points in 2D space
Implementing Point

public class Point {

}
Formal vs. Actual Parameters

- **formal** parameter: in declaration of class
- **actual** parameter: passed in when method is called
  - variable names may or may not match
- if parameter is primitive type
  - call by value: value of actual parameter copied into formal parameter when method is called
  - changes made to formal parameter inside method body will not be reflected in actual parameter value outside of method
- if parameter is object: covered later
Scope

- Fields of class are have **class scope**: accessible to any class member
  - in `Die` and `Point` class implementation, fields accessed by all class methods
- Parameters of method and any variables declared within body of method have **local scope**: accessible only to that method
  - not to any other part of your code
- In general, scope of a variable is block of code within which it is declared
  - **block** of code is defined by braces `{ }`
Commenting Code

Conventions
- explain what classes and methods do
- plus anywhere that you've done something nonobvious
  - often better to say why than what
    - not useful
      int wishes = 3; // set wishes to 3
    - useful
      int wishes = 3; // follow fairy tale convention
javadoc Comments

- Specific format for method and class header comments
  - running javadoc program will automatically generate HTML documentation

- Rules
  - /** to start, first sentence used for method summary
  - @param tag for parameter name and explanation
  - @return tag for return value explanation
  - other tags: @author, @version
  - */ to end

- Running
  - % javadoc Die.java
  - % javadoc *.java
/**
   Sets the die shape, thus the range of values it can roll.
   @param numSides the number of sides of the die
   */
public void setSides(int numSides) {
    sides = numSides;
}

/**
   Gets the number of sides of the die.
   @return the number of sides of the die
   */
public int getSides() {
    return sides;
}
/** Die: simulate rolling a die
 * @author: CPSC 111, Section 206, Spring 05-06
 * @version: Jan 31, 2006
 *
 * This is the final Die code. We started on Jan 24,
 * tested and improved in on Jan 26, and did a final
 * cleanup pass on Jan 31.
 */
Cleanup Pass

- Would we hand in our code as it stands?
  - good use of whitespace?
  - well commented?
    - every class, method, parameter, return value
  - clear, descriptive variable naming conventions?
  - constants vs. variables or magic numbers?
  - fields initialized?
  - good structure?
  - follows specification?

- ideal: do as you go
  - commenting first is a great idea!

- acceptable: clean up before declaring victory
Key Topic Summary

Borrowed phrasing from Steve Wolfman

- Generalizing from something concrete
  - fancy name: abstraction
- Hiding the ugly guts from the outside
  - fancy name: encapsulation
- Not letting one part ruin the other part
  - fancy name: modularity
- Breaking down a problem
  - fancy name: functional decomposition
Reading Assignment Next Week

- Chap 4.3-4.5 re-read