Software Quality: Program Specification, Design, Testing, Debugging

Outline

- How do we ensure that software works?
- We will carry this topic out over a couple of lectures

Main Topics (More than One Lecture!)

- Code reuse
- Specification
  - Java Interfaces
  - Pre and Postconditions
- Design
- Testing
  - Main methods
    The mysterious "static" unveiled
  - Test harnesses
- Debugging
  - toString methods

Goals as Software Developers

- ***Design*** software to increase the chances it works properly, and can be debugged and modified effectively and efficiently
  (a hard problem)
- ***Verify*** that software works *correctly*
  (whatever that means)
- ***Diagnose and Fix*** problems effectively
  (figuring out a systematic way to approach this)

What Does It Mean for Software to be "Correct"?

- Some possible definitions
  - Does what the programmer wrote in the code
  - Does what the end user/customer wants/expects
- What do *you* think is the right definition?
  1. ...
  2. ...
  3. ...
  - ...

Common Standard Definition

- "Software is correct if it operates according to its specification"
- A "specification" is a description of what the software should do
- Problem: specifications are hard to come by
  - May be informal
  - May be incomplete or incorrect
  - May be so complete and detailed they are hard to understand
  - May change over time

Fact o' Life

- Fact o' Life: Software gets built and deployed even in the absence of good specifications
- How?
  - Common sense
  - Precedent
  - Common programming practice
  - Peer advice
  - User consultation
  - Inspired guesswork...

Reusing Code

- Goal: increase correctness of a new system by reusing code already tested
- Example: libraries of all kinds
- Internal reuse:
  - Using a method or a class at different points of the program, instead of writing out the code from scratch each time
  - (Joe) Hummel's Law: "If you find your some writing the same code twice, stop and figure out how to do it right."
- *Generalize* whenever practical

  Write code once that can be used many times

A Generalization Example

- methodA sorts an array of Strings
  - alphabetic order, using String *compareTo* method
- methodB sorts an array of BankAccounts
  - order determined by comparing the balances:

  if (ba1[x].getBalance() <= ba2[y].getBalance()...
- The code for the two methods is largely the same
  - Parameter types are different
  - The only algorithmic difference:

    How to tell if one thing is <, ==, or > the other
- Goal: write one method that works for both

A Brilliant Idea (which almost works)

- Write a compareTo method for BankAccount!
- The code for comparing BankAccounts is then exactly like the code for comparing Strings!

- Stumper: what type is used for the array??
- For example, if you leave the type as
  - *void sort (String[ ] array)*
  - Java will not allow you to pass in a BankAccount[ ]

    Java knows the types are different

    Java knows that a cast from one to the other won't work

Another Brilliant Idea

- Make the parameter an array of Objects

  void sort (Object[ ] array) {...

  void insert(Object[ ] array, int n, Object newVal) {...
  - Every BankAccount is an Object
  - Every String is an Object
  - The parameter passing problem is solved
- But... Java now complains about *compareTo* in the *insert* method
  - *CompareTo* is not a method of *Object*
- Needed: a way to
  - 1. use a common type for BankAccount and String
  - 2. assure Java that the objects in the array will have a *compareTo* method

Solution: An Interface

- The word "interface" is a common one in computing
- Here we are using a narrow, technical, Java meaning for the word
- *An Interface defines a set of methods*
  - method signatures ONLY – no statements
- Any class can choose to "implement" an Interface
  - This obligates the class to fully implement each method prescribed by the Interface
  - The class can implement as many additional methods as it wants
  - A class can implement as many different interfaces as it needs
  - Textbook: 15.1.4-15.1.4

The Interface for our Example

- *compareTo* is the method that the sort method needs to call, so..
  - 1. Define an Interface which specifies the compareTo method
  - 2. Modify the sort method signature to show that the array must implement that Interface
  - Make sure that both *BankAccount* and *String* both implement that Interface

    All this requires is that each class implements a *compareTo* method
- Problem solved!
  - When Java sees a call to *sort*, it can check that the objects in the array satisfy the Interface.

More Concretely...

- "*Comparable*" is the name we choose for the Interface
- *Comparable* is defined with syntax similar to class definition syntax:

```
public interface Comparable {
    int compareTo(Object otherObj);
}
```

- It turns out the *Comparable* interface is already defined like this in Java
  - saving us the trouble of having to define it
- It turns out that the standard *String* class already implements *Comparable*
  - To verify this, visit the JavaDoc page for *String*

Interfaces: The Final Magic

- We can declare objects of this type:

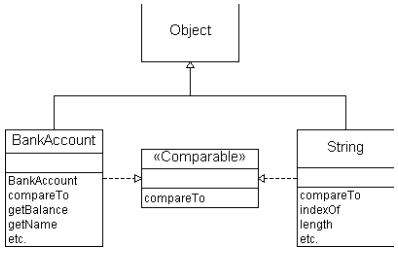
```
Comparable obj1;
```

  - means *obj1* will refer to some object which implements the Comparable interface
- The magic: *obj1* can refer either to a *BankAccount* or a *String*!
- More magic: *Comparable[]* can refer to either a *BankAccount* array or a *String* array!!
- Final form of the method signatures:

```
public void sort(Comparable[] array)
public void insert(Comparable[] array, int pos, Object newValue)
```

- Final magic: our code works now not just with String and BankAccount, but *any* objects that implement *Comparable*.

Class Relationships: Final Picture



Correctness and Specifications at the Java Level

- The unit of programming in Java is the class
- What does it mean for a class implementation to be correct?
- Informally, "everything works", provided constructors and methods are used with suitable arguments
- More precisely,
  1. A newly constructed object has an appropriate state
  2. If given suitable arguments, each method works properly, returns the right result, and leaves the object in an appropriate (possibly updated) state
- "Works properly" takes us back to the specification problem...

Specifications at the Java Level

- Specifications are often given as comments in the code
- Java programmers typically use JavaDoc conventions when writing major comments
  - Allows the comments to be extracted into a standard, widely understood format
- A particular case of code specifications is especially important: the "invariant"
  - Invariants are things which must be true if the program is correct
  - Invariants are sometimes described in comments, and sometimes incorporated into the code

Commonly Identified Invariants

Invariant in general – a property that is always true

- *Class invariant* – a property of the class – often about its state – that is always true
  - (except, possibly, momentarily while related things are being updated)
- *Precondition* – a property of a method that is required to be true for the method to be able to execute correctly
  - ("property" used in the English sense, not the technical sense of a property – instance variable – of an object)
- *Postcondition* – a property of a method that is guaranteed to be true after the method has executed, provided its preconditions were satisfied when it was called

Class Invariant Example: CreditCard Class

```
/** Representation of a single credit card */
public class CreditCard {
    // instance variables
    private String name;        // account holder's name
    private int number;         // account number
    private double limit;       // credit limit, limit >= 0.0 always
    private double balance;     // current account balance;
                                //    0.0 <= balance <= limit always
    ...
```

- The constraints on limit and balance are examples of class invariants
- Class invariants are normally not explicit in the Java code, but they are needed to understand the class – so include them in comments

Postcondition Example: CreditCard Constructor

```
public class CreditCard {
    ...
    /** Construct a new CreditCard with the given name, account number, and
     *  credit limit, with an initial balance of 0.0
     *  Postcondition: class invariants are true with name = given name,
     *      number = given number, limit = given limit, and balance = 0.0
     */
    public CreditCard(String name, int number, double limit) {
      this.name = name;
      this.number = number;
      this.limit = limit;
      this.balance = 0.0;
    }
    ...
```

- **Footnote: specifying the postcondition in this much detail is normally overkill, since the comment describes the parameters, and correct constructors/methods can be assumed to preserve the class invariants**
  (but it illustrates the point of what a postcondition is)

Precondition/Postcondition Example

```
public class CreditCard {
    ...
    /** Add amount to this credit card's balance, provided the limit is big enough.
     *  Return true if successful, return false otherwise (limit too small).
     *  precondition: amount >= 0.0
     *  postcondition: if amount+balance <= limit, increase balance by amount
     *                 and return true, otherwise do nothing and return false
     */
    public boolean charge(double amount) {
      if (balance + amount <= limit) {
        balance = balance + amount;
        return true;
      } else {
        return false;
      }
    }
```


What if the Precondition is not True?

- **This can only happen for two reasons:**
  - **Client code uses inappropriate arguments**
  - **Bugs in the class implementation**
- **How do we react?**
  - **Really covered in CSE143. Preview....**
  - **Error in client code: generate an exception**
    (like NullPointerException, MethodNotFound, ...)
  - **Bug: use assertions to catch problem during debugging**

Designing Methods

- **Invariants and Comments are valuable forms for method specifications**
- **But... who decides what methods and classes the system should have?**
- **Given a problem, there are usually many ways it can be divided into smaller parts such as methods and classes**
- **We focus here on method design: deciding which methods to define and how they fit together**
- **Typical issues:**
  - **One method or a number of smaller ones?**
  - **What should the parameters and return values be?**
  - **What instance variables are used and how?**

Testing

- Now we know how we want it to work, how do we decide if it is working?
- Goal – verify that the implementation is "correct"
- Procedure
  - Figure out what to test and what sample data to use

    Do this before or while coding
  - Run tests and compare with expected results

Test Cases

- Can't test everything – way too many possible cases
- Try to test "important" cases
  - "Typical" cases
  - Edge cases – 0, 1, many
  - "Incorrect" cases – how does the code cope with bad data?
- Goal is to find a set of cases that covers all possibilities
  - Use representative data to cover each set of similar values

Example: Fahrenheit to Celsius

- Suppose we want to test code for the conversion

  celsius = 5.0/9.0 * (fahrenheit – 32.0)
- Suggest some input values and expected output
  - Try to get complete coverage with as few cases as you can

Unit Testing vs System Testing

- TBS

Debugging – What If Something's Wrong?

- Effective debugging – a controlled experiment
  - Form hypothesis of what might be happening
  - Figure out how to gather information to verify or refute
  - Run experiments
  - Repeat until solved
- Goal is to systematically find bugs
  - What works?
  - Where do things go wrong?
  - What is happening? How can we fix it?
- Avoid random hacking – you'll just make things worse(!)

Gathering Debugging Information

- Simplest method: insert "System.out.println(*stuff*);" at interesting points
  - Figure out things you expect, then print out the actual values and compare
- Works great for basic types and objects (int, double, char, boolean, String)
- Would like to also be able to print objects to see important things about their state

```
System.out.println(checking);
```

  - Default Java prints memory address (mostly meaningless)
  - But – we can make our classes smarter so we get something useful when we print an object

Method toString

- A class can contain a toString method
- Whenever an object is used where a String is needed (in println, for example), the class's toString method (if present) is used to produce a suitable string
- toString specification (can use in any appropriate class)

```
/** Return a String representation of this object */
public String toString() { … }
```


toString Example

```
/** Return a string representation of this CreditCard */
public String toString() {
    String description = "CreditCard[name = " + name + ", number = " + number
                         + ", balance = " + balance + ", limit = " + limit + "]";
    return description;
}
```

Running Tests – main Methods

- So far we've run tests by typing statements into DrJava's interactions window

```
CreditCard plastic = new CreditCard("I. M. Broke", 80195, 5000.00);
System.out.println(plastic);
plastic.charge(4950.00);
System.out.println(plastic);
```

- A test should be packaged so it can be run repeatedly
- How? Method *main*
  - This is also the standard Java way of packaging the "starter" code needed to create initial objects and run a program

Method main

- For our purposes, create a new class with a single method that looks like this

```
public class ClassName {
    /** main method – specify what it does */
    public static void main(String[ ] args) {
        code for main method goes here
    }
}
```

  - "public static void main(String[ ] args)" has to be typed exactly like that
  - *"ClassName"* is whatever you want (Main, Test1, TestChargeMethod, …)
  - *"code for main method goes here"* is the same kinds of statements you'd type in DrJava's interactions window

    Except that ";" is required after every statement

Example Test Program for CreditCard

```
/** Test CreditCard charge method */
public class Main {
    /** test program for charge */
    public static void main(String[ ] args) {
        CreditCard plastic = new CreditCard("I. M. Broke", 80195, 5000.00);
        System.out.println(plastic);
        plastic.charge(4950.00);
        System.out.println(plastic);
    }
}
```

- Once this is compiled in DrJava, you can run it by typing

  java Main

  in the interactions window

  (use the name of the class instead of Main if it is different)

Designing Software for Quality

- There are many ways to divide the parts of a system into separate classes
- Key idea: Each class should do one thing well
- Key idea: ***Information Hiding***
  - Implementation details in different classes should be independent of and hidden from each other
  - Use public interfaces and references to other objects – don't rely on private information that can change without notice
- Hard to get right – takes experience and redesign – but makes testing, debugging, and modification much easier when done well

Coupling and Cohesion

- Specific concepts to talk about design quality
  - Qualitative, hard to measure, but useful
- ***Cohesion*** – the degree to which a class *completely* encapsulates a *single* notion
  - Maximize this
  - If a class is doing more than one thing, split it into separate classes
- ***Coupling*** – the degree to which a class interacts with and depends on other classes
  - Minimize this

Summary

- Building quality software is not easy
  - Need good design to start
    - Coupling, cohesion
    - Reuseble parts promotes quality
  - Need to check that things work as expected
    - Designing and implementing test cases
  - Need to effectively diagnose and fix any problems
    - Debugging
- Worth the effort to try to get these things right
  - Higher-quality software, built faster, tested and debugged with less grief, happier customers