CSE 142

Testing, Debugging, and Program Design

Outline for Today

- How do we ensure that software works?
- Testing
  - Main methods
- Debugging
  - toString methods
- Some thoughts on design

Goals

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

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

- Some possible definitions
  - Does what the programmer wrote in the code
  - Works as intended
  - Does what the end user/customer wants/expects
- What do you think is the right definition?

Classes

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

Preconditions, Postconditions, and Invariants

- Still more precisely
  - Invariant – a property that is always true
  - Class invariant – a property of the class – aoften 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

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

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
  - 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