CS1632, Lecture 12: Writing Testable Code

Bill Laboon

LET'S DEFINE OUR TERMS, GENTLEMEN.

Testable Code

Code for which it is easy to write and perform tests, automated and manual, at various levels of abstraction, and track down errors when tests fail.

Good Code Is Not The Same As Testable Code

- Good code is testable code
- But not all testable code is good code
- So let’s learn to write good code and make it testable

Key Ideas for Testable Code

- Segment code - make it modular
- Give yourself something to test
- Make it repeatable
- DRY (Don’t repeat yourself)
- Use the dominant paradigm of the language

Segment Code

- Methods should be SMALL and SPECIFIC
- Do one thing and do it well

```
// Bad
public int getNumMonkeysAndSetDatabase(Database d) {
   if (d != null) {
      _database = d;
   } else {
      _database = DEFAULT_DATABASE;
   }
   setDefaultDatabase(_database);
   int numMonkeys = getNormalizedMonkeys();
   if (numMonkeys < 0) {
      numMonkeys = 0;
   }
   return numMonkeys;
}
```

Refactor

```
// Better
public void setDatabase(Database d) {
   if (d != null) {
      _database = d;
   } else {
      _database = DEFAULT_DATABASE;
   }
   setDefaultDatabase(_database);
}

public int getNumMonkeys() {
   int numMonkeys = getNormalizedMonkeys();
   if (numMonkeys < 0) {
      numMonkeys = 0;
   }
   return numMonkeys;
}
```

Give Yourself Something to Test

- Return values are worth their weight in gold!
  Easy to assert against
  Guaranteed to exist (in Java)

- Exceptions, modified accessible attributes, etc… something is better than nothing!

```
public void addMonkey(Monkey m) {
   if (m != null) {
      addMonkeyToMonkeyList(m);
   }
}
```

Refactor

```
// Better
public int addMonkey(Monkey m) throws NullMonkeyException {
   int toReturn = -1;
   if (m != null) {
      toReturn = addMonkeyToMonkeyList(m);
   } else {
      throw NullMonkeyException();
   }
   return toReturn;
}
```

Make It Repeatable

- Randomness or Dependence on External Data should be minimized

- Try to segregate PURE FUNCTIONS from SIDE-EFFECT-FUL CODE
    Pure functions = output depends ONLY on input, do nothing else
    Side effects = write to database, read a global variable, write to file system, etc.

Well, This Is Bad

```
public CrapsStatus rollDiceFirst() {
   // random throws of the dice
   int dieRoll1 = (new Die()).roll();
   int dieRoll2 = (new Die()).roll();
   int total = dieRoll1 + dieRoll2;
   switch (total) {
      case 2: case 3: case 12:
        return CRAPS_LOSE;
      case 7: case 11:
         return CRAPS_WIN;
      case 4: case 5: case 6: case 8: case 9: case 10:
         _firstRoll = total;
         return CRAPS_PLAY;
      default:
         return CRAPS_ERROR;
      }
   }
}
```

Refactor

```
// Better
public CrapsStatus rollDiceFirst(Die d1, Die d2) {
   // random throws of the dice
   int dieRoll1 = d1.roll(); // Can stub roll!
   int dieRoll2 = d2.roll();
   int total = dieRoll1 + dieRoll2;
   switch (total) {
      case 2: case 3: case 12:
        return CRAPS_LOSE;
      case 7: case 11:
         return CRAPS_WIN;
      case 4: case 5: case 6: case 8: case 9: case 10:
         _firstRoll = total;
         return CRAPS_PLAY;
      default:
         return CRAPS_ERROR;
      }
   }
}
```

Even Better

```
// Better
public CrapsStatus getCrapsStatus(int dieRoll1, int dieRoll2) {
   // Actual die rolls take place elsewhere
   // No need to double/stub!
   // Method is smaller and more focused
   int total = dieRoll1 + dieRoll2;
   switch (total) {
      case 2: case 3: case 12:
        return CRAPS_LOSE;
      case 7: case 11:
         return CRAPS_WIN;
      case 4: case 5: case 6: case 8: case 9: case 10:
         _firstRoll = total;
         return CRAPS_PLAY;
      default:
         return CRAPS_ERROR;
      }
   }
}
```

DRY - Don’t Repeat Yourself

- Don’t copy and paste code from one section of your program to another
- Don’t have multiple methods with the same or similar functionality
- Try to have “generic” methods (not language-specific, but in Java, try to use Generics)

Bad

```
public int addMonkey(Monkey m) {
   if (m != null) {
      _animalList.add(m);
   }
   return _animalList.count();
}
public int addGiraffe(Giraffe g) {
   if (g != null) {
      _animalList.add(g);
   }
   return _animalList.count();
}
public int addRabbit(Rabbit r) {
   if (r != null) {
      _animalList.add(r);
   }
   return _animalList.count();
}
```

Refactor

```
// Animal is superclass for Giraffe,
// Monkey, and Rabbit

public int addAnimal(Animal a) {
    if (a != null) {
        _animalList.add(a);
    }
    return _animalList.count();
}
```

Ensure That You Don't Have Multiple Methods Doing The Same Thing

```
public int addUpArray(int[] x) {
   int toReturn = 0;
   for (int j=0; j<x.length; x++) {
      toReturn += x[j];
   }
   return toReturn;
}
// elsewhere in codebase..
public int arrayTotal(int[] a) {
   int toReturn = 0;
   int c = 0;
   while (++c < a.length) {
      toReturn = toReturn + a[c];
   }
   return toReturn;
}
```

Why?

- Twice as much room for error
- Bloated codebase
- Perhaps slightly different behavior (look closely at previous code!)
- Harder to find errors
- Which one to use?

Replicated Code Could Be Internal To Methods!

```
// In one method…
String name = db.where("user_id = " +
    id_num).get_names[0];

// Elsewhere, in another method…
String name =
    db.find(id).get_names().first();
```

You Can DRY This Up, Too

```
public static String getName(Database db,
int id) {
   // Add in guard code, try..catch, etc.
   // Can all be here in one place
   return db.find(id).get_names().first();
}

// In one method…
String name = getName(db, id);

// Elsewhere, in another method…
String name = getName(db, id);
```

Use the dominant paradigm of the language

- Java is object-oriented - program in an object-oriented way

- Will help you with making stubs, doubles, mocks, segregating code, etc.

Procedural Style

```
public static int rollDie(Random r) {
   return r.nextInt(6) + 1;
}

public static void main(String[] args) {
   Random rng = new Random(args[0]);
   int dieRoll1 = rollDie(rng);
   int dieRoll2 = rollDie(rng);
   boolean keepPlaying = true;
   while (keepPlaying) {
      ...
```

In An Object-Oriented Language, Write Object-Oriented Code

```
public class Die {
   Random _rng = null;
   public Die() {
      _rng = new Random();
   }
   public Die(int seed) {
      _rng = new Random(seed);
   }
   public int roll() {
      return r.nextInt(6) + 1;
   } r.nextInt(6) + 1;

}
```

Languages Are Designed The Way They Are For a Reason

YOU CAN PROGRAM JAVA IN A FUNCTIONAL WAY

or a procedural way

or a logical way

or a constraint-based way

BUT IT MIGHT BE AS WEIRD, DIFFICULT-TO-USE AND DIFFICULT-TO-UNDERSTAND AS THE FONTS ON THIS SLIDE

The SOLID Principles

- A mnemonic for “five key principles” of good object-oriented design
  - Single Responsibility Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

Single Responsibility Principle

A class should have a single responsibility.

That responsibility should be entirely encapsulated by the class.

Bad “S”

```
// What is Stuff’s single responsibility?

public class Stuff {
    public void printMemo() { ... }
    public int numCats(String breed) { ... }
    public String getName() { ... }
    public void haltSystem(int exitCode) { ... }
}
```

Better “S”

```
public class Cat {
   public String getName() { ... }
   public String getBreed() { ... }
   public Currency getRentalCost() {...}
   public int rent() { ... }
}

public class RentACatSystem {
   public void startSystem() { ... }
   public void haltSystem(int exitCode) { ... }
   public void forceShutdown() { ... }

}
```

Single Responsibility Principle

Describe the class. If you can't do it without using "and", you are probably violating the Single Responsibility principle.

Other code smells:

1. Many methods
2. Many attributes
3. Difficult to comprehend what class does
4. Methods don't seem related

Open / Closed Principle

Classes should be open for extension, but closed to modification.

In other words, add features by subclassing, not adding code.

Once complete, code modification in a given class should generally not occur except to fix defects.

Open / Closed Principle

```
public class Printer {
    private void formatDocument() { ...
}
    public void printDocument() { ... }
}
```

Now let’s say we want to add a way to print PDFs. One way would be to add a method:

```
    public void printToPDF()  { ... }
```

But this is a violation of the “O”!

Better

```
abstract class Printer {
   private void formatDocument() { ... }
}

public class PhysicalPrinter extends Printer {
   public void printDocument() { ... }
}

public class PdfPrinter extends Printer {
   public void printDocument() { ... }
}
```

The Open/Closed Principle

If your classes keep getting bigger with each commit, you may be violating the Open/Closed Principle.

This helps us because once a class is done, it’s done. Modifying classes incessantly is a recipe for regression errors.

Liskov Substitution Principle

A class B which is a subclass of class A, should implement any method in A while meeting all invariants.

Liskov Substitution Principle

```
// What’s wrong with this?
public class Circle {
   public Location loc;
   public Color color;
   public double radius;
}

public class Square extends Circle {
    public double length;
    public double height;
}
```

Better

```
public class Shape {
   Location loc;
   Color color;
}

public class Rectangle extends Shape {
    public double length;
    public double height;
}

public class Circle extends Shape {
    public double radius;
}
```

Interface Segregation Principle

Clients should not depend on methods that they do not use.

In practice, this means lots of small interfaces, not one big one.

Interface Segregation Principle

```
public interface BankInterface {
    public void transferMoneyIntraBank();
    public void transferMoneyInterBank();
    public void allocateMortgage();
    public void transferMortgage();
    public void setupHeloc();
    public void withdrawCash();
    public void depositCheck();
    public void depositCash();
    public void authenticate();
    public Bank[] getBankBranches();
    public Employee[] getBankEmployees();
}
```

Better

```
public interface AtmInterface {
    public void withdrawCash();
    public void depositCash();
    public void depositCheck();
    public void authenticate();
}
```

Interface Segregation Principle

If you find yourself not using all of the methods of an interface, consider splitting up the interfaces for different roles.

Otherwise, there is more room for error and the code becomes more difficult to understand.

Be a code anti-natalist!

Dependency Inversion Principle

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle

```
public class Aviary {
   public void buyCockatiel();
   public void buyGreyParrot();
   public void buyYellowBelliedSapSucker();
}
```

Better

```
public class Aviary {
    public void buyBird(Bird b);
}

public abstract class Bird {
…
}
public class Cockatiel extends Bird {
…
}
```

Dependency Inversion Principle

Attributes need to be interfaces (or an abstract class)

All class packages connect through interfaces

Concrete classes are final classes; all subclasses should derive from abstract classes

As a side effect of that, any concrete method should not be overridden, only abstract methods

The Downside

OVERENGINEERING

See “Enterprise FizzBuzz” for this principle run amok

Avoid Leaky Abstractions

If you abstract something, you should not need to know implementation details.

If I know I have a Bird class, I can tell it to fly(). I should not have to say:

```
Bird b = new Eagle();
if (b instanceof Hummingbird) {
   flapWingsFast();
} else if (b instanceof Seagull) {
   glide();
} // … etc.
```

Leaky Abstractions Are Bad

Abstract this unnecessary knowledge away. The user of the class does not want to - and should not - know about it.

```
public class Hummingbird extends Bird {
    public void fly() {
        flapWingsFast();
    }
}

public class SeaGull extends Bird {
    public void fly() {
        glide();
    }
}
```

Leaky Abstractions Are Inevitable

The Law of Leaky Abstractions:
“All non-trivial abstractions, to some degree, are leaky.” - Joel Spolsky

TCP is abstracted away to be a reliable connection - until it isn’t.

SQL is an abstraction of a database, but if performance matters, how you construct your query also matters. For that you need to know how the data is stored.

The Law of Demeter*

"Tell, don't ask"

Never call a method on an object you got from another call.

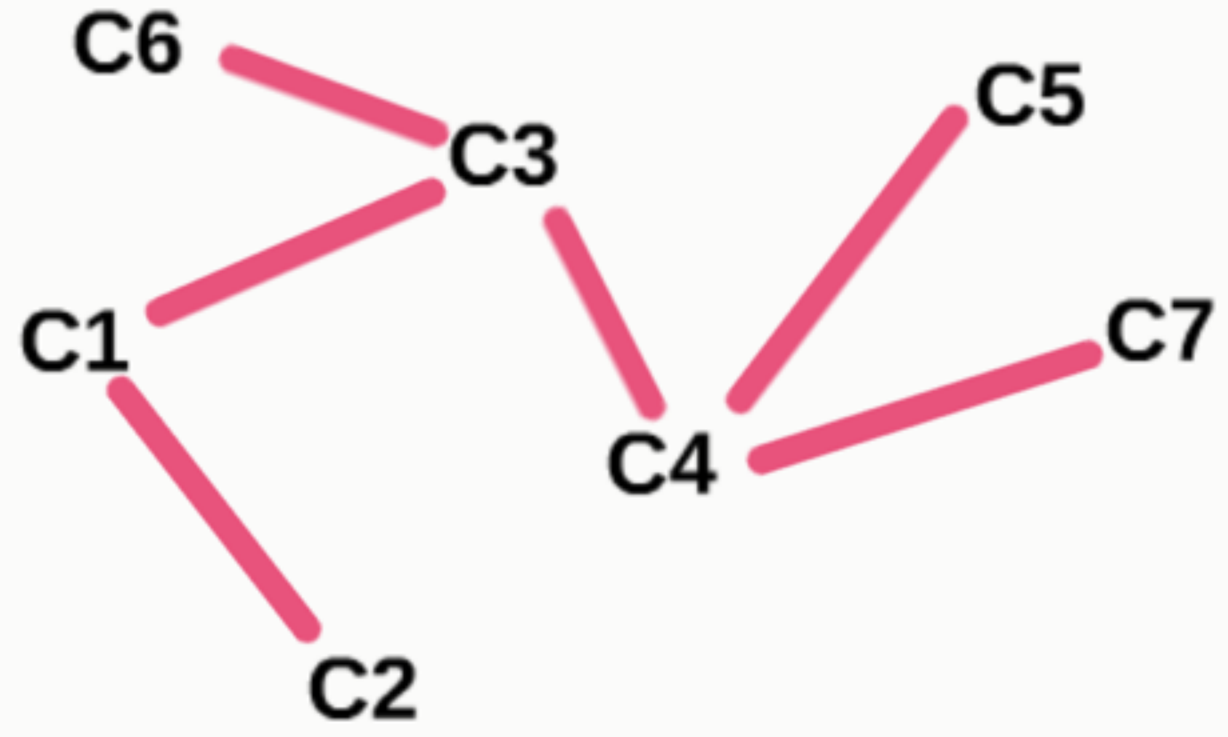
The Law of Demeter

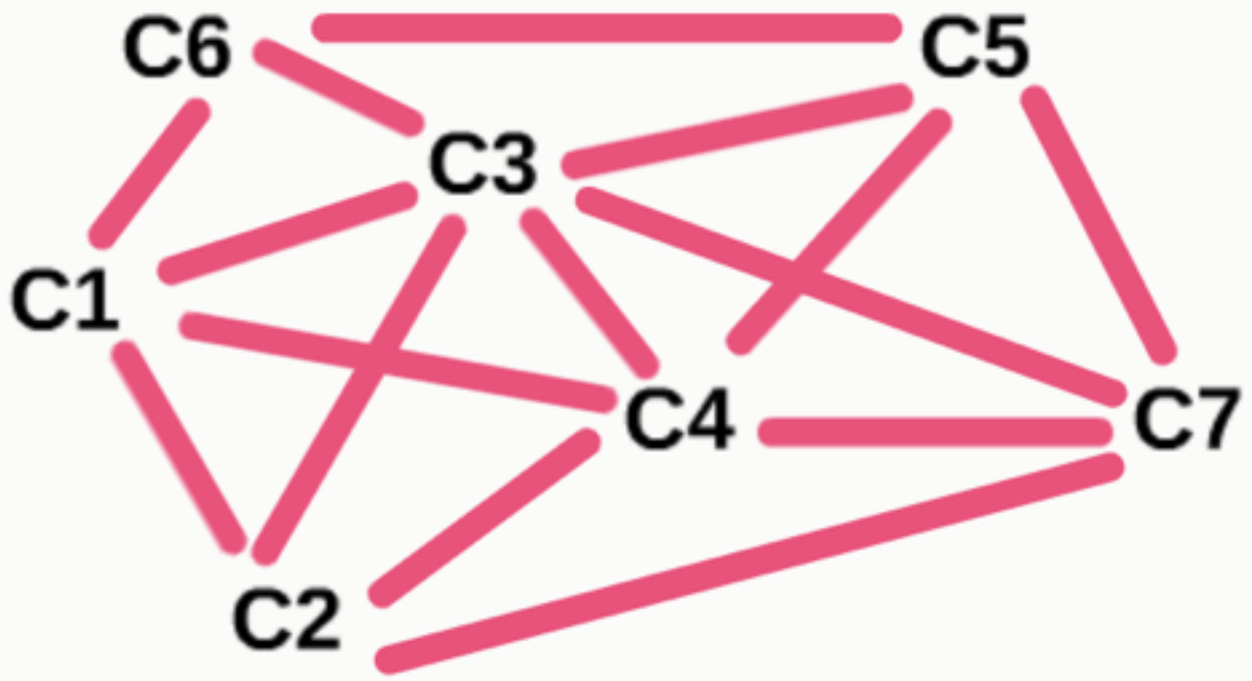
```
String pigLatinName =
    database.getTable(“Users").lookup(id).translate("Pig
       Latin”);
```

Better

```
String pigLatinName = PigLatinizer.pig_latinize(name);
```

Which looks easier to refactor?





Only Applies to Different Classes!

```
// In this case, every return value is a String.
// The Law of Demeter applies to CLASS dependencies,
// NOT objects.

String foo =
   "  BLAH ".toLowerCase().substring(2).replace('a',
      ‘b’).trim();
```

The Law of Demeter

If you have a long line of dot-whatevers, you may be violating the Law of Demeter.

However, if every return value applies to the same class, you are not violating it.

No TUFs Inside TUCs

That is, no

Test-Unfriendly Features

inside

Test-Unfriendly Constructs

Examples of Test-Unfriendly Features

- Printing to console
- Reading/writing from a database
- Reading/writing to a filesystem
- Accessing a different program or system
- Accessing the network

Examples of Test-Unfriendly Constructs

- Private methods
- Final methods
- Final classes
- Class constructors / destructors
- Static methods

Provide (or Look For) Seams

Seams are places where behavior can be modified without modifying code

Make these common!

The rule of “making more methods” can be considered a special subclass of this rule.

Example

```
// SEAM
public void printDoc(Printer p, args) {
    p.print(args);
}

// NO SEAM
public void printDoc2() {
    Printer p = new Printer(DEFAULT_ARGS);
    p.print();
}
```

Dealing With Legacy Code

In most classes, you are either writing greenfield code (that is, code from scratch) or code that your professor wrote to make it easy on you (even though it may not always look like that).

The real world is seldom so tidy.

Code is often written hurriedly and under pressure. Modifying it later is difficult.

Dealing With Legacy Code

Best advice:

Leave the codebase better than when you found it.

You can try adding tests! You can try refactoring!

Make more seams!

Move TUFs out of TUCs!

Don’t sink into the Swamp of Sadness.

Dealing With Legacy Code

“Working With Legacy Code” by Michael Feathers is an excellent resource.

Summary: http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf

Testable Java: http://www.objectmentor.com/resources/articles/TestableJava.pdf