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

```java
// 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

```java
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

Be a code anti-natalist!

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"

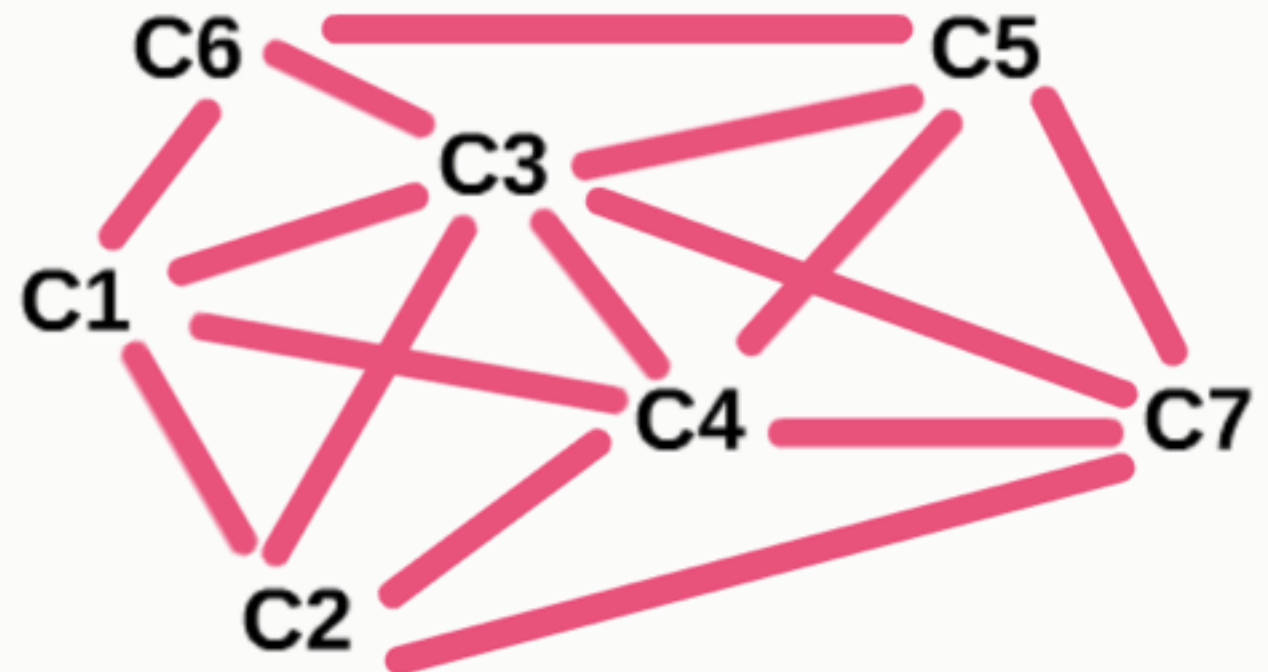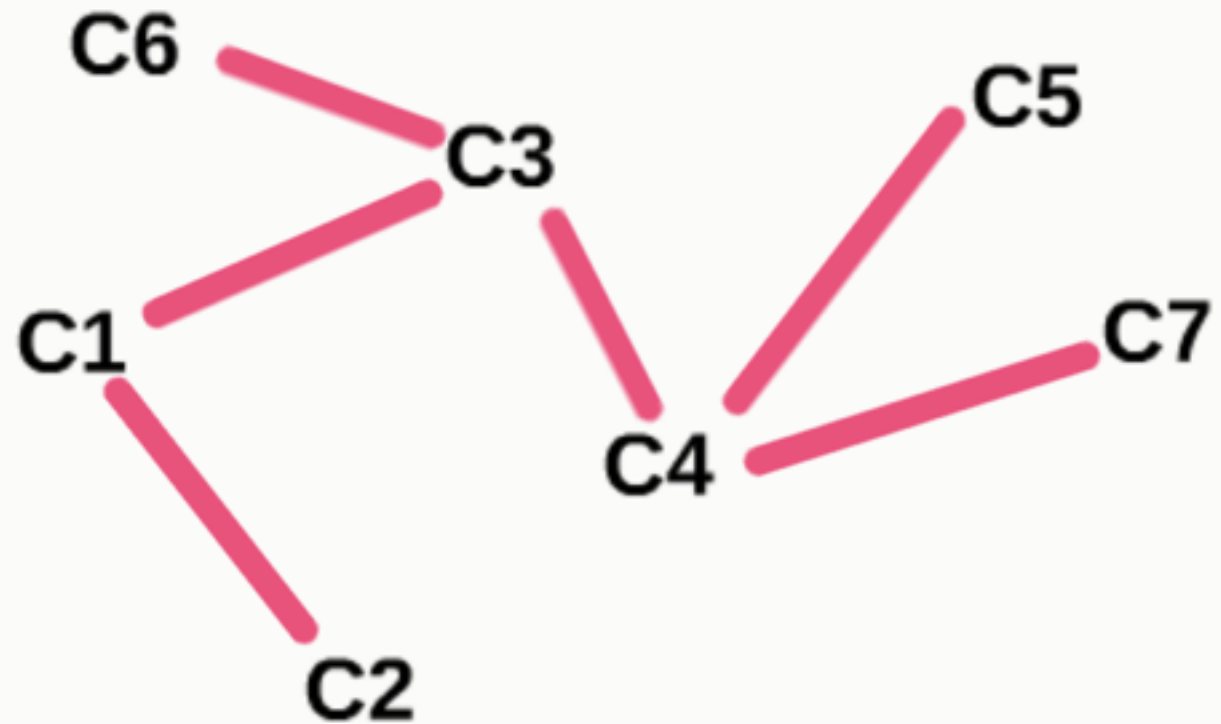Never call a method on an object you got from another call.

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