



# CS1632, Lecture 12: Writing Testable Code

Wonsun Ahn

# *Testable Code*

---

- Code for which it is easy to perform tests
  - Whether automated and manual
  - Whether at unit level or system level
- ... and track down errors when tests fail

## 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
- Move TUFs out of TUCs

## 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 = _monkeyList.size();
  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 = _monkeyList.size();
  return numMonkeys;
}
```

## Give Yourself Something to Test

---

- Return values are worth their weight in gold!
  - Easy to assert against
- What if method does not have a return value?
  - Try adding a success/fail return value
  - Try throwing exceptions to indicate problems
  - Try modifying value of an accessible attribute
  - Always, something is better than nothing!

```
// Bad. No state to check for defects.  
public void addMonkey(Monkey m) {  
    if (m != null) {  
        _monkeyList.add(m);    // _monkeyList is private  
    }  
}
```

# Refactor

---

```
// Better. Can assert on return value.
public boolean addMonkey(Monkey m) {
    if (m != null) {
        _monkeyList.add(m);
        return true;
    }
    return false;
}
```

```
// Also better. Can catch exception and assert there.
public void addMonkey(Monkey m) throws NullMonkeyException {
    if (m != null) {
        _monkeyList.add(m);
    } else {
        throw NullMonkeyException();
    }
}
```

# Make It Repeatable

---

- Dependence on random or external data == Bad
  - Random data makes it impossible to repeat result (as you can imagine)
  - External data also makes repeating results very hard
- What is external data?
  - Value of global variables
  - Value extracted from a global data structure
  - Value returned from a database query
  - Value read from a file
  - Basically any value that you did not pass in as arguments or locally produce



# Not Repeatable Code

---

```
public Result playOverUnder() {  
    // random throw of the dice  
    int dieRoll = (new Die()).roll();  
    if (dieRoll > globalThreshold) {  
        return RESULT_OVER;  
    }  
    else if (dieRoll < globalThreshold) {  
        return RESULT_UNDER;  
    }  
    else {  
        return RESULT_AT;  
    }  
}
```

- **Two reasons why the result is not repeatable:**
  1. `dieRoll` will obtain a random value on every call
  2. `globalThreshold` may be different across calls
- **What if tester pre-sets `globalThreshold` before performing unit test?**
  - `globalThreshold` may be modified internally (e.g. by `Die.roll()`)

# Refactor

---

```
public Result playOverUnder(Die d) {  
    // random throw of the dice  
    int dieRoll = d.roll(); // Can stub roll()  
    if (dieRoll > globalThreshold) {  
        return RESULT_OVER;  
    }  
    else if (dieRoll < globalThreshold) {  
        return RESULT_UNDER;  
    }  
    else {  
        return RESULT_AT;  
    }  
}
```

- Now, a test double can be passed in for `Die` and `roll()` can be stubbed
- This type of refactoring is called *dependency injection*
  - Passing dependencies in as parameters to the tested method

## Even Better

---

```
public Result playOverUnder(int dieRoll) {  
    if (dieRoll > globalThreshold) {  
        return RESULT_OVER;  
    }  
    else if (dieRoll < globalThreshold) {  
        return RESULT_UNDER;  
    }  
    else {  
        return RESULT_AT;  
    }  
}
```

- Now, no need to even create a test double or stub!
- What about `globalThreshold`?
  - Inject that dependency too!

# The Best

---

```
public Result playOverUnder(int dieRoll, int threshold) {  
    if (dieRoll > threshold) {  
        return RESULT_OVER;  
    }  
    else if (dieRoll < threshold) {  
        return RESULT_UNDER;  
    }  
    else {  
        return RESULT_AT;  
    }  
}
```

- Now, this method has become a pure function
  - Pure function: function where result is computed purely from parameters
  - A pure function is by definition always repeatable
- Try to make your functions pure functions whenever possible
  - Segregate hard-to-test code with side-effects into a small corner

# DRY - Don't Repeat Yourself

---

- Don't copy and paste code
- Don't have multiple methods with similar functionality
- Make use of “generic” classes and methods
  - Classes and methods that have parameterized types
  - E.g. Java `List<Type>` is parameterized by `Type` so that user can make a list of integers (`List<Integer>`) or list of strings (`List<String>`) using the same class
  - Language implementations: Java generics, C++ templates, ...

## Why DRY?

---

- Twice as much room for error
- Bloated codebase
- A bug fix or enhancement must be replicated on all copies of the code – another source of error

## Bad: Replicated code but with different types

---

```
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();  
}
```



## Bad: Two copies of very similar code (but slightly different)

---

```
public int addUpArray(int[] x)
{
    int toReturn = 0;
    for (int j=0; j<x.length; j++)
        { 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;
}
```

## 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

---

```
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

---

- When in Rome, do as the Romans do
- Automated testing frameworks are geared towards the dominant programming paradigm for that language
- Java is an Object Oriented Programming (OOP) language
  - Program in an OOP way!
  - It will allow you to use test doubles, stubs, mocks, ...

# 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 OOP Language, Write OOP Code

---

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

# Don't make life hard for the tester

---

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

# 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
- ☛ Code that you typically *want* to fake using stubs

## Examples of Test-Unfriendly Constructs

---

- Private methods
- Final methods
- Final classes
- Class constructors / destructors
- Static methods
- ☛ Code that is *hard* to fake using stubs

## No TUFs Inside TUCs

---

- In other words ...
- Do not put code that you want to fake inside that is hard to fake

# Dealing with Legacy Code



Image from <https://goiabada.blog>

# Dealing With Legacy Code

---

- In most classes, you had it easy
  - You either wrote greenfield code (that is, code from scratch)
  - Or modified 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 under pressure, with no consideration for testing, let alone any testing code
  - Often there is little to no documentation and you aren't even sure how the legacy code even works
- Where do you start?

# Start by Writing Pinning Tests

---

- *Pinning Test*: A test done to pin down existing behavior
  - Note: existing behavior may be different from expected behavior
  - Want to pin down all behavior, bugs and all, before modifying
  - Even obscure corner case behaviors may sometimes be used
    - ☛ Must make sure these don't get accidentally modified
- Pinning tests are typically done using unit testing
  - Where do I look for places where I can unit test?
  - Look for seams!

## Look for Seams in your Legacy Code

---

- *Seams*: Places where *behavior* can be modified without modifying *code*

- Example with no seam:

```
void executeSql(String sql) {  
    DatabaseConnection db = new DatabaseConnection();  
    db.executeSql(sql);  
}
```

- ☛ Hard to unit test since we need a working DB connection

- Example with seam:

```
void executeSql(String sql, DatabaseConnection db) {  
    db.executeSql(sql);  
}
```

- ☛ Easy to unit test by passing a test double DB connection

# Look for Seams in your Legacy Code

---

- Does this really have no seam?

```
void executeSql(String sql) {  
    DatabaseConnection db = new DatabaseConnection();  
    db.executeSql(sql);  
}
```

- Maybe it does, if you look closely enough!



# Look for Seams in your Legacy Code

---

- Suppose you have this legacy class

```
class LegacyClass
{
    virtual void executeSql(String sql) {
        DatabaseConnection db = new DatabaseConnection();
        db.executeSql(sql);
    }
}
```

- Now create a new class JustForTestingClass

```
class JustForTestingClass : public LegacyClass
{
    virtual void executeSql(String sql) {
        DatabaseConnection db =
Mockito.mock(DatabaseConnection.class);
        // Stub db to specify whatever behavior you want
        db.executeSql(sql);
    }
}
```

- Use JustForTestingClass for testing purposes

## Dealing With Legacy Code

---

- After pinning down behavior, you can slowly start refactoring the code
- Leave the codebase better than when you found it.
- Don't sink into the Swamp of Sadness.