



CS1632, Lecture 11: Writing Testable Code

Wonsun Ahn

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- Give yourself something to test
- Move TUFs out of TUCs
- Make it easy to satisfy preconditions
- Make it easy to reproduce
- Make it easy to localize

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- Give yourself something to test
- Move TUFs out of TUCs
- Make it easy to satisfy preconditions
- Make it easy to reproduce
- Make it easy to localize

Segment Code

- Methods should perform one well-defined functionality

// Bad

```
public int getNumMonkeysAndSetDatabase(Database d) {  
    database = (d != null) ? d : DEFAULT_DATABASE;  
    setDefaultDatabase(database);  
    int numMonkeys = monkeyList.size();  
    return numMonkeys;  
}
```

- **Why?**

Refactor

```
public void setDatabase(Database d) {  
    database = (d != null) ? d : DEFAULT_DATABASE;  
    setDefaultDatabase(database);  
}  
  
public int getNumMonkeys() {  
    int numMonkeys = monkeyList.size();  
    return numMonkeys;  
}
```

- Why is this better?
 1. Simply more modular, better quality code
 2. But also easier to test --- no spurious dependency on Database d!

Key Ideas for Testable Code

- Segment code - make it modular
- **DRY (Don't repeat yourself)**
- Give yourself something to test
- Move TUFs out of TUCs
- Make it easy to satisfy preconditions
- Make it easy to reproduce
- Make it easy to localize

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 `ArrayList<Type>` is parameterized by `Type`
 - 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 only with different types

```
private ArrayList<Animal> animalList;

public int addMonkey(Monkey m) {
    animalList.add(m);
    return animalList.count();
}

public int addGiraffe(Giraffe g) {
    animalList.add(g);
    return animalList.count();
}

public int addRabbit(Rabbit r) {
    animalList.add(r);
    return animalList.count();
}
```

Refactor

```
// Animal is superclass of Giraffe, Monkey, Rabbit

private ArrayList<Animal> animalList;
public int addAnimal (Animal a) {
    animalList.add(a);
    return animalList.count();
}
```

Bad: What if there is no superclass?

```
// No superclass for List<Monkey>, List<Giraffe>, List<Rabbit>
public void addOne(List<Monkey> l, Monkey m) {
    l.add(m);
}

public void addOne(List<Giraffe> l, Giraffe g) {
    l.add(g);
}

public void addOne(List<Rabbit> l, Rabbit b) {
    l.add(b);
}
```

Refactor

```
// Use a generic method.  
// Pass List<T> where T can be any type.  
  
public <T> void addOne(List<T> l, T e) {  
    l.add(e);  
}
```

Replicated Code Could Be Internal To Methods!

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

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

```
// Both do basically the same thing
```

You Can DRY This Up, Too

```
// In one method...
```

```
String name = getName(db, id);
```

```
// Elsewhere, in another method...
```

```
String name = getName(db, id);
```

```
String getName(Database db, int id) {  
    // Enhancing this code will impact all calls  
    return db.find(id).get_names().first();  
}
```

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- **Give yourself something to test**
- Move TUFs out of TUCs
- Make it easy to satisfy preconditions
- Make it easy to reproduce
- Make it easy to localize

Give Yourself Something to Test

- By "something", I mean some state (usually a return value)
- Because state verification is always better than behavior verification
 - Checking program state after function call is the most *direct* way to verify it
 - Behavior verification is *brittle* because it delves into implementation details

Why Behavior Verification is Indirect

```
class Number {  
    private int val;  
    public void setVal(int v) { val = v; }  
}  
  
class Example {  
    public void setSquared(Number n, int v) {  
        n.setVal(v*v);  
    }  
}  
  
@Test void testSetSquared() {  
    Example ex = new Example();  
    Number n = Mockito.mock(Number.class)  
    ex.setSquared(n, 3);  
    Mockito.verify(n).setVal();  
}
```

Why Behavior Verification is Indirect

```
class Number {  
    private int val;  
    public void setVal(int v) { val = v; }  
}  
  
class Example {  
    public void setSquared(Number n, int v) {  
        n.setVal(v+v); // DEFECT!  
    }  
}  
  
@Test void testAddTen() {  
    Example ex = new Example();  
    Number n = Mockito.mock(Number.class)  
    ex.setSquared(n, 3);  
    Mockito.verify(n).setVal(); // STILL PASSES!  
}
```

Why Behavior Verification is Brittle

```
class Number {
    private int val;
    public void setVal(int v) { val = v; }
    public void setSquared(int v) { val = v*v; }
}

class Example {
    public void setSquared(Number n, int v) {
        n.setVal(v*v);
    }
}

@Test void testAddTen() {
    Example ex = new Example();
    Number n = Mockito.mock(Number.class)
    ex.setSquared(n, 3);
    Mockito.verify(n).setVal();
}
```

Why Behavior Verification is Brittle

```
class Number {
    private int val;
    public void setVal(int v) { val = v; }
    public void setSquared(int v) { val = v*v; }
}

class Example {
    public void setSquared(Number n, int v) {
        n.setSquared(v); // SAME THING as n.setVal(v*v)
    }
}

@Test void testAddTen() {
    Example ex = new Example();
    Number n = Mockito.mock(Number.class)
    ex.setSquared(n, 3);
    Mockito.verify(n).setVal(); // NO DEFECT BUT FAILS!
}
```

Refactor --- Give Yourself Something to Test

```
class Number {  
    private int val;  
    public void setVal(int v) { val = v; }  
}  
  
class Example {  
    public int setSquared(Number n, int v) {  
        int ret = v*v;  
        n.setVal(ret);  
        return ret;  
    }  
}  
  
@Test void testAddTen() {  
    Example ex = new Example();  
    Number n = Mockito.mock(Number.class);  
    assertEquals(9, ex.setSquared(n, 3));  
}
```

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- Give yourself something to test
- **Move TUFs out of TUCs**
- Make it easy to satisfy preconditions
- Make it easy to reproduce
- Make it easy to localize

No TUFs Inside TUCs

That is, no

Test-Unfriendly Features (TUFs)

inside

Test-Unfriendly Constructs (TUCs)

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
- ☛ Methods that are *hard* to fake using stubs or overriding

No TUFs Inside TUCs

- In other words ...
- Do not put code that you want to fake (TUFs) inside methods that are hard to fake (TUCs)

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- Give yourself something to test
- Move TUFs out of TUCs
- **Make it easy to satisfy preconditions**
- Make it easy to reproduce
- Make it easy to localize

Make it Easy to Satisfy Preconditions

- Dependence on external data == Bad
- What is external data?
 - Value of global variables
 - Value extracted from a global data structure
 - Value read from a file or database
 - Basically any value that you did not pass in as arguments
 - Also colloquially known as *side-effects*

Make it Easy to Satisfy Preconditions

// Bad

```
public float getCatWeight(Cat cat) {  
    int fishWeight = Fish.weight;  
    // Because the cat ate the fish  
    return cat.weight + fishWeight;  
}
```

- **Why?** `Fish.weight` is external data not in arguments.
 - Not good code in general: dependency embedded deep in implementation (Coders may modify `Fish.weight` and inadvertently impact `getCatWeight`)
 - Not good code for testing: easy to **miss** these in the **preconditions**

Refactor

// Better

```
public float getCatWeight(Cat cat, int fishWeight) {  
    return cat.weight + fishWeight;  
}
```

- Why? Now `fishWeight` is explicit in the arguments.
 - Nobody is surprised when `getCatWeight` changes when `fishWeight` changes
 - Easy to test: no preconditions at all!
- All values are passed through arguments
 - Otherwise known as a *pure function*
 - Why functional languages are easier to test and lead to fewer defects

Make it Easy to Satisfy Preconditions

- We have to access external data somewhere, where do we do it?
 1. If you make use of arguments, you will need less global variables
 - With the original `getCatWeight`:
`Fish.weight = 5;`
`int weight = getCatWeight(cat);`
With refactored `getCatWeight`:
`int weight = getCatWeight(cat, 5);`
 2. For the remaining external data
 - Segregate hard-to-test code with side-effects into a small corner
 - Keep as many methods *pure* as possible

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- Give yourself something to test
- Move TUFs out of TUCs
- Make it easy to satisfy preconditions
- **Make it easy to reproduce**
- Make it easy to localize

Make it Easy to Reproduce

- Dependence on random data == Bad
 - Random data makes it impossible to reproduce result

// Bad

```
public Result playOverUnder() {  
    // random throw of the dice  
    int dieRoll = (new Die()).roll();  
    if (dieRoll > 3) {  
        return RESULT_OVER;  
    }  
    else {  
        return RESULT_UNDER;  
    }  
}
```

Refactor

```
public Result playOverUnder(Die d) {  
    int dieRoll = d.roll();  
    if (dieRoll > 3) {  
        return RESULT_OVER;  
    }  
    else {  
        return RESULT_UNDER;  
    }  
}
```

- **Why better?** Now you can mock Die and stub d.roll():
Die d = Mockito.mock(Die.class);
Mockito.when(d.roll()).thenReturn(6);
playOverUnder(d);

Even Better

```
public Result playOverUnder(int dieRoll) {  
    if (dieRoll > 3) {  
        return RESULT_OVER;  
    }  
    else {  
        return RESULT_UNDER;  
    }  
}
```

- Why better? Now don't even have to mock or stub anything!
`playOverUnder(6);`

Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don't repeat yourself)
- Give yourself something to test
- Move TUFs out of TUCs
- Make it easy to satisfy preconditions
- Make it easy to reproduce
- **Make it easy to localize**

Make it Easy to Localize

// Bad

```
public class House {  
    private Room bedroom;  
    private Room bathRoom;  
    public House() {  
        bedroom = new Room("bedRoom");  
        bathRoom = new Room("bathRoom");  
    }  
    public String toString() {  
        return bedroom.toString() + " " + bathRoom.toString();  
    }  
}
```

- **Why? No way to mock bedroom or bathRoom and stub toString().**

Refactor

```
public class House {  
    private Room bedroom;  
    private Room bathRoom;  
    public House(Room r1, Room r2) {  
        bedroom = r1;  
        bathRoom = r2;  
    }  
    public String toString() {  
        return bedroom.toString() + " " + bathRoom.toString();  
    }  
}
```

- **Now we can easily mock and stub:**

```
Room bedroom = Mockito.mock(Room.class);  
Room bathRoom = Mockito.mock(Room.class);  
House house = new House(bedroom, bathRoom);
```

Make it Easy to Localize

- This is called *dependency injection*
- *Dependency injection*: Passing a dependent object as argument rather than building it internally
 - Makes testing easier by allowing you to mock that object
 - Has other software engineering benefits like decoupling the two classes

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
- The real world is seldom so tidy
 - Code is often written hurriedly under pressure, with no consideration for testing
 - Often there is no documentation and you aren't even sure how the code works
- Where do you even 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:

```
String read(String sql) {  
    DatabaseConnection db = new DatabaseConnection();  
    return db.executeQuery(sql);  
}
```

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

- Example with seam:

```
String read(String sql, DatabaseConnection db) {  
    return db.executeQuery(sql);  
}
```

☛ Easy to unit test by passing a mock db and stubbing db.executeQuery

Look for Seams in your Legacy Code

- Does this really have no seam?

```
String read(String sql) {  
    DatabaseConnection db = new DatabaseConnection();  
    return 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 Database
{
    String read(String sql) {
        DatabaseConnection db = new DatabaseConnection();
        return db.executeQuery(sql);
    }
}
```

- Now create a new class DatabaseForTesting

```
class DatabaseForTesting : public Database
{
    String read(String sql) {
        // Pretend we have a connection and return an entry
        return "test database entry";
    }
}
```

- Use DatabaseForTesting for testing purposes

Dealing With Legacy Code

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

Now Please Read Textbook Chapter 16