# CS1632:
# Writing Testable Code

Wonsun Ahn

# Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don’t repeat yourself)
- 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)
- 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. You are trying to do two things at the same time
public int getNumMonkeysAndSetDatabase(Database d) {
  database = (d != null) ? d : DEFAULT_DATABASE;
  setDefaultDatabase(database);
  int numMonkeys = monkeyList.size();
  return numMonkeys;
}
```

- **Why?**

# Refactor

```
// Good. Performs one thing: setting the database
public void setDatabase(Database d) {
  database = (d != null) ? d : DEFAULT_DATABASE;
  setDefaultDatabase(database);
}
// Good. Performs one thing: getting the number of monkeys
public int getNumMonkeys() {
  int numMonkeys = monkeyList.size();
  return numMonkeys;
}
```

1. More modular: better reusability, better readability, better maintainability, …
2. But also more testable: `getNumMonkeys()` no longer depends on `Database d`

# Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don’t repeat yourself)
- 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

# What's so Bad about Repeating Yourself?

- Leads not only to a bloated code base
  - Twice the amount of code to maintain
  - Twice the room for error

- But also less testable code
  - Twice the amount of testing you need to do
  - When a defect is found, bug fix must be replicated in all copies of code
    → Easy to forget a few, or apply the fix in a wrong way in a particular copy

# How do you DRY?

- For duplicate code, create a new method and call that instead:
  - Suppose you had the below two copies of code that are functionally identical:

```
// Copy 1 somewhere in source code
String name = db.where("user_id = " + id).get_names()[0];
// Copy 2 somewhere else in source code
String name = db.find(id).get_names().first();
```

  - DRY up the code by adding a new method `getName`

```
String getName(Database db,  int id) {
  // Enhancing this code will impact all calls
  return db.find(id).get_names().first();
}
// Copy 1 somewhere in source code
String name = getName(db, id);
// Copy 2 somewhere else in source code
String name = getName(db, id);
```

# How do you DRY?

- For two (or more) similar methods, merge them into one:
  - Suppose you had two methods `insertMonkey` and `addMonkey` similar in functionality:

```
public void insertMonkey(Monkey m) {
  animalList.add(m);
}
public int addMonkey(Monkey m) {
  animalList.add(m);
  return animalList.count();
}
insertMonkey(new Monkey());                // A use of insertMonkey
int count = addMonkey(new Monkey());       // A use of addMonkey
```

  - DRY up the code by merging the two methods into `addMonkey`

```
public int addMonkey(Monkey m) {
  animalList.add(m);
  return animalList.count();
}
addMonkey(new Monkey());                   // Changed to use addMonkey
int count = addMonkey(new Monkey());       // A use of addMonkey
```

# How do you DRY?

- What if two codes are functionally similar only with different types?
  - Happens frequently with object-oriented languages like Java or C++

- Make use of *polymorphism*
  - a.k.a. subclassing, subtyping, inheritance
  - Using a variable that can take on different types at runtime

# 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 Using Polymorphism

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

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

# How do you DRY?

- What if two codes are functionally similar only with different types?
  - Happens frequently with object-oriented languages like Java or C++
- Make use of polymorphism
  - a.k.a. subclassing, subtyping, inheritance
  - Using a variable that can take on different types at runtime
- Make use of *generic* classes and methods
  - Classes and methods that have parameterized types
  - E.g. Java `ArrayList<Integer>` is parameterized by type `Integer`
  - Language implementations: Java generics, C++ templates, …

# 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 Using Generics

```
// 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);
}
```

# Key Ideas for Testable Code

- Segment code - make it modular
- DRY (Don’t repeat yourself)
- 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)**

# Test-Unfriendly Features

- Feature that you typically *want* to fake using stubs
  - Feature takes too long to set up to work correctly
  - Feature takes too long to test (typically involving I/O)
  - Testing feature can cause unwanted side-effects

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

# Test-Unfriendly Constructs

- ☛ Methods that are *hard* to fake using *stubbing* or *overriding*
  - *Stubbing*: replacing a method in a mocked object using Mockito
  - *Overriding*: overriding a method in a “fake” class that subclasses real class

- Examples
  - Object constructors / destructors: impossible to override
  - Private methods: impossible to override
  - Final methods: impossible to override
  - Static methods: impossible to override or to stub
    (Impossible to stub since static methods are called on classes not objects)

# 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)
- 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 for testing

- 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 pass data using arguments, you will need less external data
   - With the original `getCatWeight`:
     ```
     Fish.weight = 5;
     int weight = getCatWeight(cat);
     ```
   - With refactored `getCatWeight`, no more global variable `Fish.weight`:
     ```
     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)
- 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 for testing
  - 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)
- 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 `Rooms` and stub `Room.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
  - Also what allowed us to mock the `Die` object for reproducibility
  - Has other software engineering benefits like *decoupling* the two classes (*Decoupled*: means it is easy to switch out one class for another)

# Dealing with Legacy Code


Image from https://goiabada.blog

# Dealing With Legacy Code

- Legacy code in the real world is seldom 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
- Now your project manage comes along and tells you to improve the code
  - Maybe refactor it to improve its performance
  - Maybe refactor it to make it more readable
  - Maybe even add a new feature
  - *Without breaking anything that worked before*
- Where do you even start?
  - Need to build a testing infrastructure to ensure nothing breaks
  - Problem is, legacy code was not written to be testable

# 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 "defective" behaviors that violate method specs may sometimes be used
    ☛ Must make sure that even these don't get *accidentally* modified
    ☛ If you modify them without modifying that use case, your software will break!

- 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

- *Seam* in software QA:
  - Place where two code modules meet where one can be switched for another
  - Without having to modify source code
- Why are seams important for pinning tests?
  - We want to pin down the behavior of each unit in legacy code
  - Seam is a place where fake objects can be injected to localize the unit tests
- Why is it important not to modify the source code?
  - The whole point of pinning tests is to test the legacy code *as is*
  - If we modify code, there is a danger that we may be changing behavior

# Look for Seams in your Legacy Code

- Example legacy code with no seam:

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

  ☛ Hard to unit test since we are forced to work with a real DB connection

- Example legacy code with seam:

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

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

# Look for Seams in your Legacy Code

- Does this really have no seam?

```
class Database
  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

- Does this really have no seam?

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

- Create a new “fake” class `FakeDatabase` for testing purposes

```
class FakeDatabase extends Database
  String read(String sql) {
    // Pretend we have a connection and return entry
    return “test database entry”;
  }
}
```

# Refactoring Legacy Code

1. Write pinning tests for the class(es) you will be refactoring
   - Make use of seams to enable unit testing
2. Refactor a method
3. Run pinning tests to make sure existing behavior did not change
4. Repeat Steps 2 and 3 for every method you want to refactor

# Now Please Read Textbook Chapter 16