# CS1632:
# Writing Testable Code

Wonsun Ahn

# Key Ideas for Testable Code

- DRY (Don’t repeat yourself)
- Create seams in your code
- Make testing easy to reproduce
- Move TUFs out of TUCs

# Key Ideas for Testable Code

- DRY (Don’t repeat yourself)
- Create seams in your code
- Make testing easy to reproduce
- Move TUFs out of TUCs

# 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

# DRYing Copy-And-Paste Code

- Suppose you had the below two copies of code that are functionally identical:

```
// Copy 1 somewhere in source code
String one = db.find(1).get_names().first();
// Copy 2 somewhere else in source code
String two = db.find(2).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 one = getName(db, 1);
// Copy 2 somewhere else in source code
String two = getName(db, 2);
```

# DRYing Duplicate Methods

- If you have two similar methods `insertMonkey` and `addMonkey`:

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

- DRY up the code by just keeping `addMonkey`:

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

# What if methods differ only in parameter types?

- Happens frequently with object-oriented languages like Java or C++
- Make use of *polymorphism*
  - a.k.a. subclassing, subtyping, inheritance

# addMonkey, addGiraffe, addRabbit do the same thing

```
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();
}
```

# DRYing by Using Polymorphism

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

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

# No superclass for Car, Banana, Rabbit. What to do?

```
public void addCar(List<Car> l, Car car) {
  l.add(car);
}
public void addBanana(List<Banana> l, Banana banana) {
  l.add(banana);
}
public void addRabbit(List<Rabbit> l, Rabbit rabbit) {
  l.add(rabbit);
}
```

# DRYing by Using Generics

```
// Generic method, where T is a paramterized type.
public <T> void addSomething(List<T> l, T e) {
   l.add(e);
}
```

- Now we can add any type of object:

```
List<Car> carList = new ArrayList<>();
addSomething(carList, new Car());
```

# Key Ideas for Testable Code

- DRY (Don’t repeat yourself)
- Create seams in your code
- Make testing easy to reproduce
- Move TUFs out of TUCs

# What are Seams?

- *Seam* in software QA:
  - Place where two objects meet where one object can be switched for another

- Why are seams important for testing?
  - Seam is a place where mock objects can be injected for unit testing

- *Dependency injection*: Creating seams in your software
  - By passing in dependent objects as arguments, rather than creating internally
  - Allows you to pass in mock objects for the purposes of testing
  - Has other software engineering benefits like *decoupling*

# Creating a seam in a class

```
// 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()`.

# Creating a seam in a class

```
// Good
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 pass in mock `Rooms` into `House` constructor.

# Creating a seam in a method

- Example 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 code with seam, after dependency injection:

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

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

# Key Ideas for Testable Code

- DRY (Don’t repeat yourself)
- Create seams in your code
- Make testing easy to reproduce
- Move TUFs out of TUCs

# 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;
   }
}
```

# Perform Depedency Injection on Die Object

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

- Now you can mock Die and stub d.roll():

```
Die d = Mockito.mock(Die.class);
Mockito.when(d.roll()).thenReturn(6);
playOverUnder(d);
```

# Key Ideas for Testable Code

- DRY (Don’t repeat yourself)
- Create seams in your code
- Make testing easy to reproduce
- Move TUFs out of TUCs

# 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

- TUCs for Stubbing:
  - Static methods: Only instance methods of mock objects can be stubbed

- TUCs for Overriding:
  - Final methods: impossible to override by fake subclass
  - Object constructors / destructors: also impossible to override

# 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)

# 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 performance or readability
  - 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** before refactoring
  1. Expected behavior may even be unknown because of lack of documentation
  2. Even if expected behavior is known, we still want to pin down existing behavior
     - ☛ Even if existing behavior violates documentation, we want to keep that behavior (We don't know enough to judge whether the documentation or code is wrong)
     - ☛ Mindset is to not break something that has been working so far!

- Look for *seams* to mock TUFs for easy testing
  - E.g. mock a database to easily test hard to recreate edge and corner cases

# Refactoring Legacy Code

1. Write pinning tests for the class(es) you will be refactoring
   - Make use of seams to fake TUFs
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