



CS1632: Writing Testable Code

Wonsun Ahn

Key Ideas for Testable Code

- DRY (Don't repeat yourself)
- Create seams in your code
- Move TUFs out of TUCs

Key Ideas for Testable Code

- DRY (Don't repeat yourself)
- Create seams in your code
- 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

```
// Animal is superclass of Giraffe, Monkey, Rabbit
public void addMonkey(List<Animal> l, Monkey m) {
    l.add(m);
}

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

public void addRabbit(List<Animal> l, Rabbit r) {
    l.add(r);
}
```

DRYing by Using Polymorphism

```
// Animal is superclass of Giraffe, Monkey, Rabbit  
  
public int addAnimal (List<Animal> l, Animal a) {  
    l.add(a);  
}
```

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 parameterized type.  
public <T> void addItem(List<T> l, T item) {  
    l.add(item);  
}
```

- Now we can add any type of object:

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

Key Ideas for Testable Code

- DRY (Don't repeat yourself)
- Create seams in your code
- 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 / fake objects can be injected for testing
- *Dependency injection*: Creating seams in your software
 1. Pass in dependent objects as arguments, rather than creating internally
 2. Inject mock objects when testing emulating behavior you want to test
- Has other software engineering benefits like *decoupling*
 - Easy to replace modules with other modules that are faster / more reliable / ...

Dependency Injection for Unit Testing

// 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 for unit testing House.

Dependency Injection for Unit Testing

// 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 create House with mock Rooms with stubbed toString().

Dependency Injection for Faster Testing

- Example code with no seam:

```
int countEntries(String sql) {  
    DatabaseConnection db = new DatabaseConnection();  
    List<Entry> entries = db.executeSql(sql);  
    return entries.size();  
}
```

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

- Example code with seam, after dependency injection:

```
int countEntries(String sql, DatabaseConnection db) {  
    List<Entry> entries = db.executeSql(sql);  
    return entries.size();  
}
```

☛ Now we can inject a mock db and all queries are super fast!

Dependency Injection for Deterministic Testing

- 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;  
    }  
}
```

Dependency Injection for Deterministic Testing

// 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);
assertEquals(RESULT_OVER, playOverUnder(d);

Key Ideas for Testable Code

- DRY (Don't repeat yourself)
- Create seams in your code
- 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
- Feature needs complex setup to work correctly
 - Needs set up of complex software / hardware
- Feature involves long latencies
 - Talking to a remote service over a slow network
 - Accessing data from slow data storage
- Feature can cause unwanted side-effects
 - Mess with customer database in current deployment

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 inherits from 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 child class
 - Object constructors / destructors
 - In any OOP language, child constructor always calls parent constructor implicitly
 - Enforced by language runtime to ensure parent members are also initialized with child

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 manager 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

Start by Writing Pinning Tests

- *Pinning Test*: A test done to pin down **existing behavior** before refactoring
 1. Expected behavior may be unknown because of lack of documentation.
 2. Even if expected behavior is known, we still want to pin down existing behavior.
 - ☛ Existing behavior is prioritized because documentation can go stale.
(Mindset is to not break something that has been working so far.)
 - ☛ Can worry about whether to update documentation or existing behavior later.
- Look for *seams* to mock TUFs for easy testing
 - E.g. Mock a database to easily recreate testing scenarios for tested feature

Refactoring Legacy Code

1. Write pinning tests for the class you will be refactoring
 - Observe existing behavior and encode them into assertions
 - Make use of seams to fake TUFs
2. Refactor the class
3. Run pinning tests to make sure existing behavior did not change
4. Repeat Steps 1 and 3 for every class you want to refactor

Now Please Read Textbook Chapter 16