Writing testable code

Examples for OOP

Why does the code matter?

- Tests can only be as good as the code
- Maintainable code means maintainable tests
- SOLID principles can improve code quality and therefore test quality in Java
- We will look at three examples

Is this code testable?

```
class ClassThatDoesEverything {
    private int base;
    private int meaningOfLife;
    /* dozen other vars */
    public ClassThatDoesEverything(int base, ...) {
        this.base = base;
        /* set remaining vars */
    }
    /* 500 lines of code */
    public int addAndCalcLifeMeaning(int val) {
        this.base += val;
        /* 100s of lines of other stuff */
        this.meaningOfLife = 42;
    }
}
```

Is this code testable?

```
class ClassThatDoesEverything {
    private int base;
    private int meaningOfLife;
    /* dozen other vars */
    public ClassThatDoesEverything(int base, ...) {
        this.base = base;
        /* set remaining vars */
    }
    /* 500 lines of code */
    public int addAndCalcLifeMeaning(int val) {
        this.base += val;
        /* 100s of lines of other stuff */
        this.meaningOfLife = 42;
    }
}
```

We have to alter every test if we modify our class signature!

Complex methods = complex tests

Single Responsibility Principle

- The “keep it simple stupid”-rule
- One task, one method or class

```
class Counter {
    int base;
    public Counter(int base) {
        this.base = base;
    }
    public int add(int val) {
        this.base += val;
        return base;
    }
}
```

Single Responsibility Principle

- The “keep it simple stupid”-rule
- One task, one method or class

Pruned Constructor!

```
class Counter {
    int base;
    public Counter(int base) {
        this.base = base;
    }
    public int add(int val) {
        this.base += val;
        return base;
    }
}
```

Easy to test!

Is this code testable?

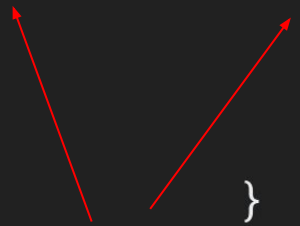
```
class IntCounter {
    int base;
    public IntCounter(int base) {
        this.base = base;
    }
    public int add(int val) {
        this.base += val;
        return base;
    }
}
```

```
class FloatCounter {
    float base;
    public IntCounter(float base) {
        this.base = base;
    }
    public float add(float val) {
        this.base += val;
        return base;
    }
}
```

Is this code testable?

```
class IntCounter {
    int base;
    public IntCounter(int base) {
        this.base = base;
    }
    public int add(int val) {
        this.base += val;
        return base;
    }
}
```

```
class FloatCounter {
    float base;
    public IntCounter(float base) {
        this.base = base;
    }
    public float add(float val) {
        this.base += val;
        return base;
    }
}
```



Existing tests impacted if swap, change or add more counters!

Open–closed principle

- Use polymorphism!
- Makes code easily extendable with minimal impact on existing tests
- Allows us to mock/stub the interface instead of each implementation!

```
interface Counter<T> {
    public T add(T val);
}

class IntCounter implements Counter<Integer> {
    /* ... */
    public Integer add(Integer val){
        this.base = base.add(val);
        return base;
    }
}
```

Is this code testable?

```
interface Inventory {/* ... */}
class InventoryImpl {/* ... */}
class Store {
    private Inventory inventory;
    public Store() {
        this.inventory = new InventoryImpl();
    }
}
```

Is this code testable?

```
interface Inventory {/* ... */}
class InventoryImpl {/* ... */}
class Store {
    private Inventory inventory;
    public Store() {
        this.inventory = new InventoryImpl();
    }
}
```

Tests on Store will test Inventory as well!

Dependency inversion principle

- Declare dependencies in the constructor!
- Easily test only Store by stubbing/mocking inventory

```
interface Inventory {/* ... */}
class Store {
    private Inventory inventory;
    public Store(Inventory inventory) {
        this.inventory = inventory;
    }
}
```

Dependency inversion principle

- Declare dependencies in the constructor!
- Easily test only **Store** by stubbing/mocking **Inventory**

Remember “Open-Closed” - use interfaces!

```
interface Inventory {/* ... */}
class Store {
    private Inventory inventory;
    public Store(Inventory inventory) {
        this.inventory = inventory;
    }
}
```

Eliminate “new”!