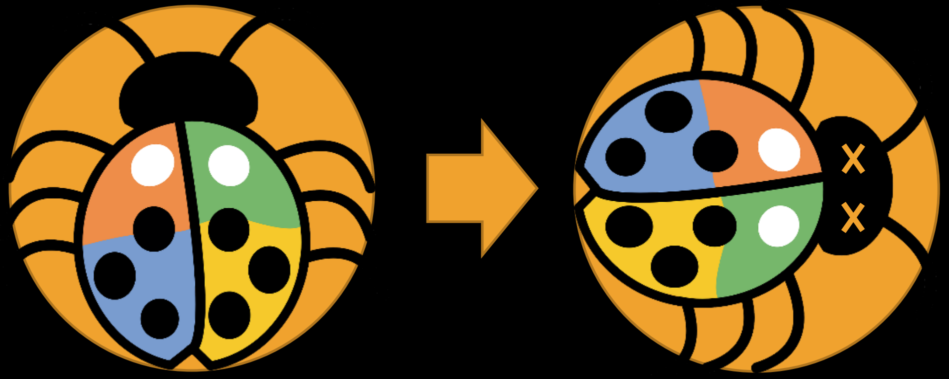
Unit Testing

Building Rock-Solid Software

SoftUni Foundation



SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents

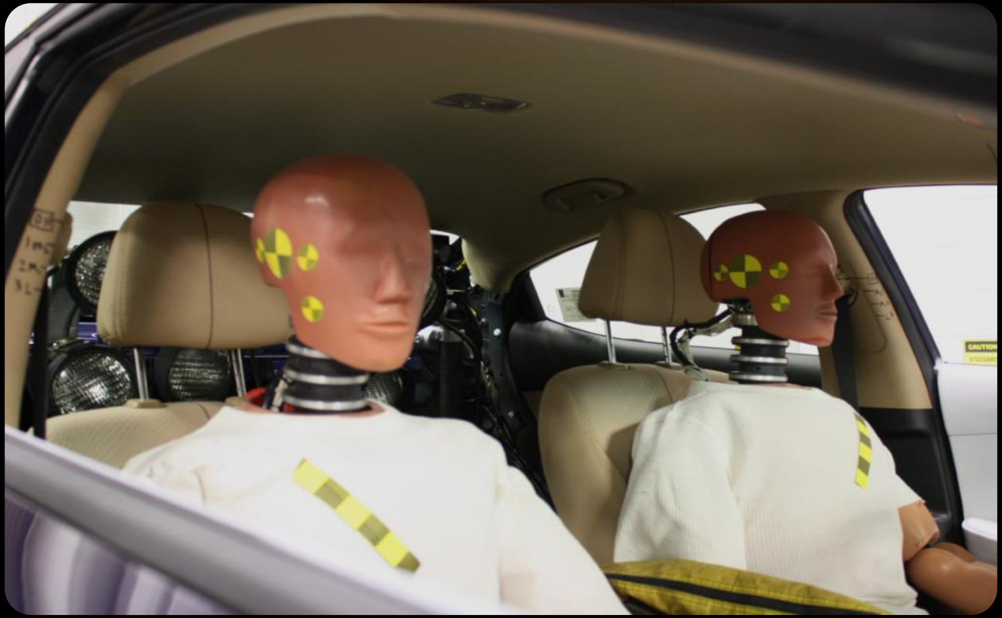
- What is **Unit Testing**?
- Unit Testing **Basics**
  - **3A** Pattern
  - Good Practices
- Unit Testing Frameworks - **JUnit**
- **Dependency Injection**
- **Mocking** and **Mock** Objects

Questions

SoftUni Foundation

sli.do

# JavaFundamentals



What is Unit Testing

Software Used to Test Software

Manual Testing

- Not **structured**
- Not **repeatable**
- Can't **cover** all of the code
- **Not** as **easy** to do as it should be

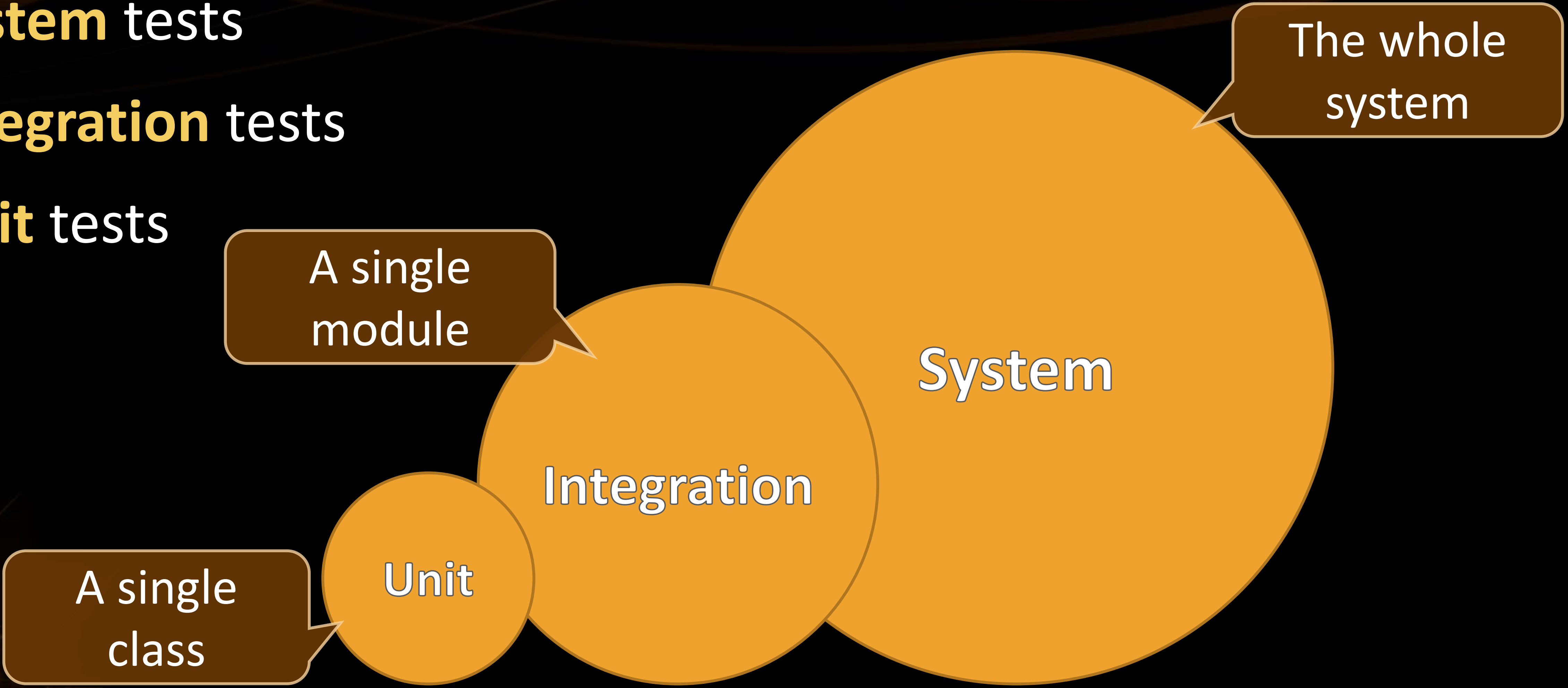
```
void testSum() {
  if (this.sum(1, 2) != 3) {
    throw new Exception("1 + 2 != 3");
}
```

Manual Testing (2)

- We need a **structured approach** that:
  - Allows **refactoring**
  - Reduces the **cost of change**
  - **Decreases** the number of **defects** in the code
- Bonus:
  - Improves **design**

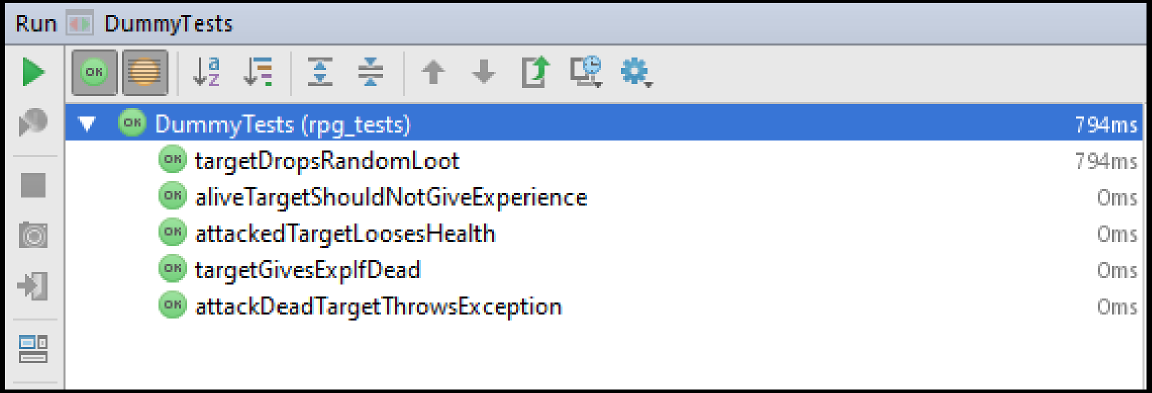
Automated Testing

- **System** tests
- **Integration** tests
- **Unit** tests

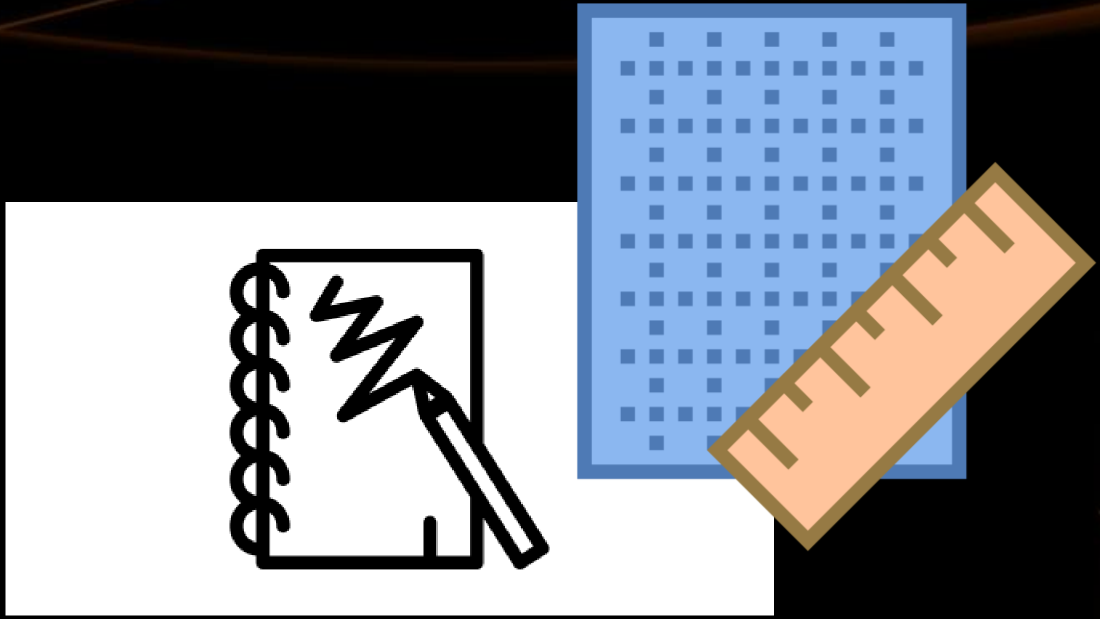


JUnit

- The first popular unit testing **framework**
- Most popular for Java development
- Based on Java, written by Kent Beck & Co.



```
DummyTests (rpg_tests)                    794ms
    targetDropsRandomLoot                 794ms
    aliveTargetShouldNotGiveExperience      0ms
    attackedTargetLoosesHealth              0ms
    targetGivesExpIfDead                    0ms
    attackDeadTargetThrowsException         0ms
```



Unit Testing Basics

How to Write Tests

Junit – Writing Tests

- Create new package (e.g. **tests**)
- Create a class for test methods (e.g. **BankAccountTests**)
- Create a **public void** method annotated with **@Test**

```
@Test
public void depositShouldAddMoney() {
  /* voodoo magic */
}
```

3A Pattern

- **Arrange** - Preconditions
- **Act** - Test a **single behavior**
- **Assert** - Postconditions

```
@Test
public void depositShouldAddMoney() {
  BankAccount account = new BankAccount();
  account.deposit(50);
  Assert.assertTrue(account.getBalance() == 50)
}
```

Each test should test a **single behavior**!

Exceptions

- Sometimes **throwing** an exception is the **expected behavior**

```
@Test(expected = IllegalArgumentException.class)
public void depositNegativeShouldNotAddMoney() {
  BankAccount account = new BankAccount();
  account.deposit(-50);
}
```

Assert

Arrange

Act

Problem: Test Axe

- Create a **`Maven`** project
- Add provided classes (**`Axe`**, **`Dummy`**, **`Hero`**) to project
- In **`test/java`** folder, create a package **`rpg_tests`**
- Create a class **`AxeTests`**
- Create the following tests:
  - Test if weapon **loses durability** after attack
  - Test attacking with a **broken weapon**

Solution: Test Axe

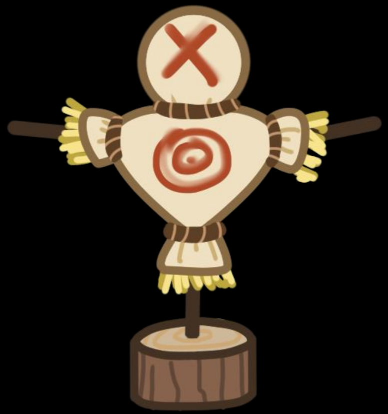
```
@Test
public void weaponLosesDurabilityAfterAttack() {
  // Arrange
  Axe axe = new Axe(10, 10);
  Dummy dummy = new Dummy(10, 10);
  // Act
  axe.attack(dummy);
  // Assert
  Assert.assertTrue(axe.getDurabilityPoints() == 9);
}
```

Solution: Test Axe (2)

```java
@Test(expected = IllegalStateException.class) // Assert
public void brokenWeaponCantAttack() {
  // Arrange
  Axe axe = new Axe(10, 10);
  Dummy dummy = new Dummy(10, 10);
  // Act
  axe.attack(dummy);
  axe.attack(dummy);
}
```

Problem: Test Dummy

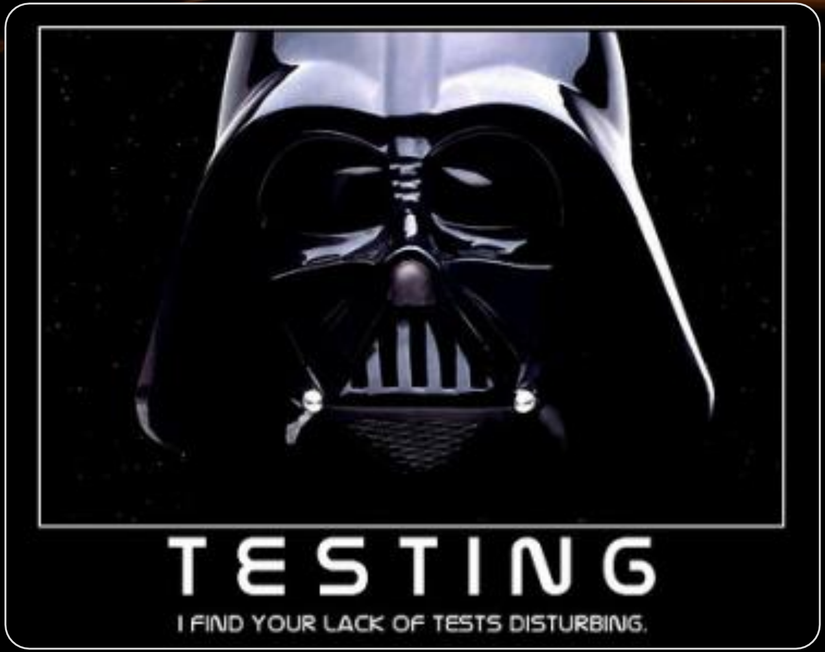
- Create a class **DummyTests**
- Create the following tests
  - Dummy **loses health** if attacked
  - Dead Dummy **throws exception** if attacked
  - Dead Dummy **can give** XP
  - Alive Dummy **can't give** XP



Solution: Test Dummy

```
@Test
public void attackedTargetLoosesHealth() {
  // Arrange
  Dummy dummy = new Dummy(10, 10);
  // Act
  dummy.takeAttack(5);
  // Assert
  Assert.assertTrue(dummy.getHealth() == 5);
}
// TODO: Write the rest of the tests
```

There is a better solution...



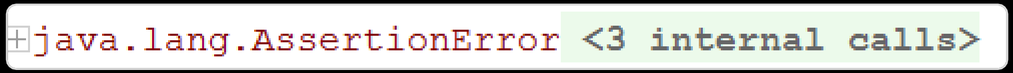
Unit Testing Best Practices

How to Write Good Tests

Assertions

- `assertTrue()` vs `assertEquals()`
  - `assertTrue()`

```
Assert.assertTrue(account.getBalance() == 50);
```

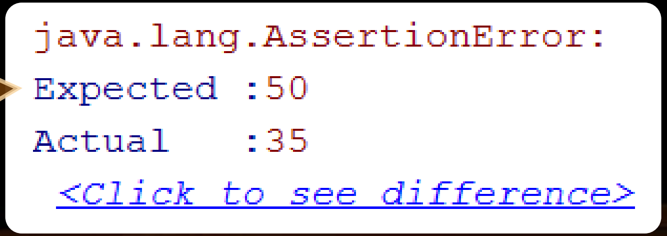


```
java.lang.AssertionError <3 internal calls>
```

  - `assertEquals(expected, actual)`

```
Assert.assertEquals(50, account.getBalance());
```

Better description when expecting value



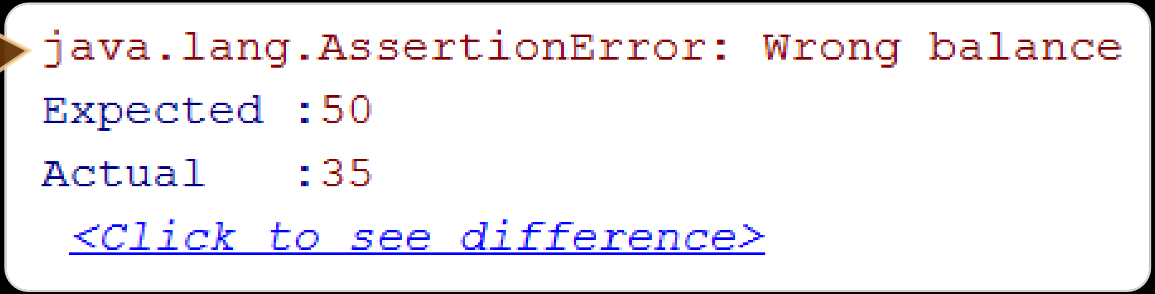
```
java.lang.AssertionError:
Expected :50
Actual   :35
 <Click to see difference>
```

Assertion Messages

- Assertions can **show messages**
  - Helps with **diagnostics**
- **Hamcrest** is useful tool for test diagnostics

```
Assert.assertEquals(
    "Wrong balance", 50, account.getBalance());
```

Helps finding the problem



```
java.lang.AssertionError: Wrong balance
Expected :50
Actual   :35
 <Click to see difference>
```

Magic Numbers

- Avoid using magic numbers (use **constants** instead)

```
private static final int AMOUNT = 50;
@Test
public void depositShouldAddMoney() {
  BankAccount account = new BankAccount();
  account.deposit(AMOUNT);
  Assert.assertEquals("Wrong balance",
                 AMOUNT, account.getBalance());
}
```

@Before

- Use **@Before** annotation

```
private BankAccount account;
@Before
public void createAccount() {
   this.account = new BankAccount();
}
@Test
public void depositShouldAddMoney() { /… }
```

Executes before each test

Naming Test Methods

- Test names
  - Should use **business domain terminology**
  - Should be **descriptive** and **readable**

```
incrementNumber() {}
test1() {}
testTransfer() {}
```

✗

```
depositAddsMoneyToBalance() {}
depositNegativeShouldNotAddMoney() {}
transferSubtractsFromSourceAddsToDestAccount() {}
```

✓

Problem: Refactor Tests

- Refactor the tests for **Axe** and **Dummy** classes
- Make sure that
  - **Names** of test methods are **descriptive**
  - You use **appropriate assertions** (assert equals vs assert true)
  - You use **assertion messages**
  - There are **no magic numbers**
  - There is no **code duplication** (Don’t Repeat Yourself)

Solution: Refactor Tests

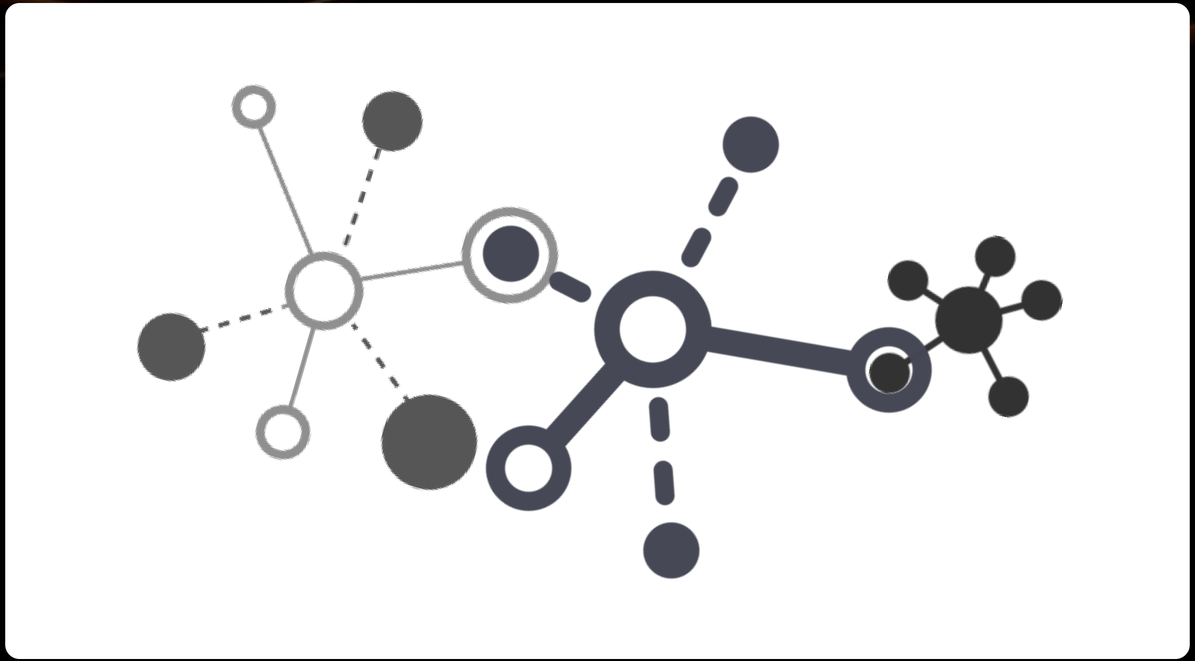
```java
private static final int AXE_ATTACK = 10;
private static final int AXE_DURABILITY = 10;
private static final int DUMMY_HEALTH = 10;
private static final int DUMMY_XP = 10;
private Axe axe;
private Dummy dummy;
@Before
public void initializeTestObjects() {
  this.axe = new Axe(AXE_ATTACK, AXE_DURABILITY);
  this.dummy = new Dummy(DUMMY_HEALTH, DUMMY_XP);  }
```

Solution: Refactor Tests (2)

```
@Test
public void weaponLosesDurabilityAfterAttack() {
  this.axe.attack(this.dummy);
  Assert.assertEquals("Wrong durability",
       AXE_DURABILITY,
       axe.getDurabilityPoints());  }
@Test(expected = IllegalStateException.class)
public void brokenWeaponCantAttack() {
  this.axe.attack(this.dummy);
  this.axe.attack(this.dummy);  }
```

Unit Testing Basics

Live Exercises in Class (Lab)



Dependencies

Isolating Behaviors

Coupling and Testing

- Consider testing the following code:
  - We want to test a **single behavior**

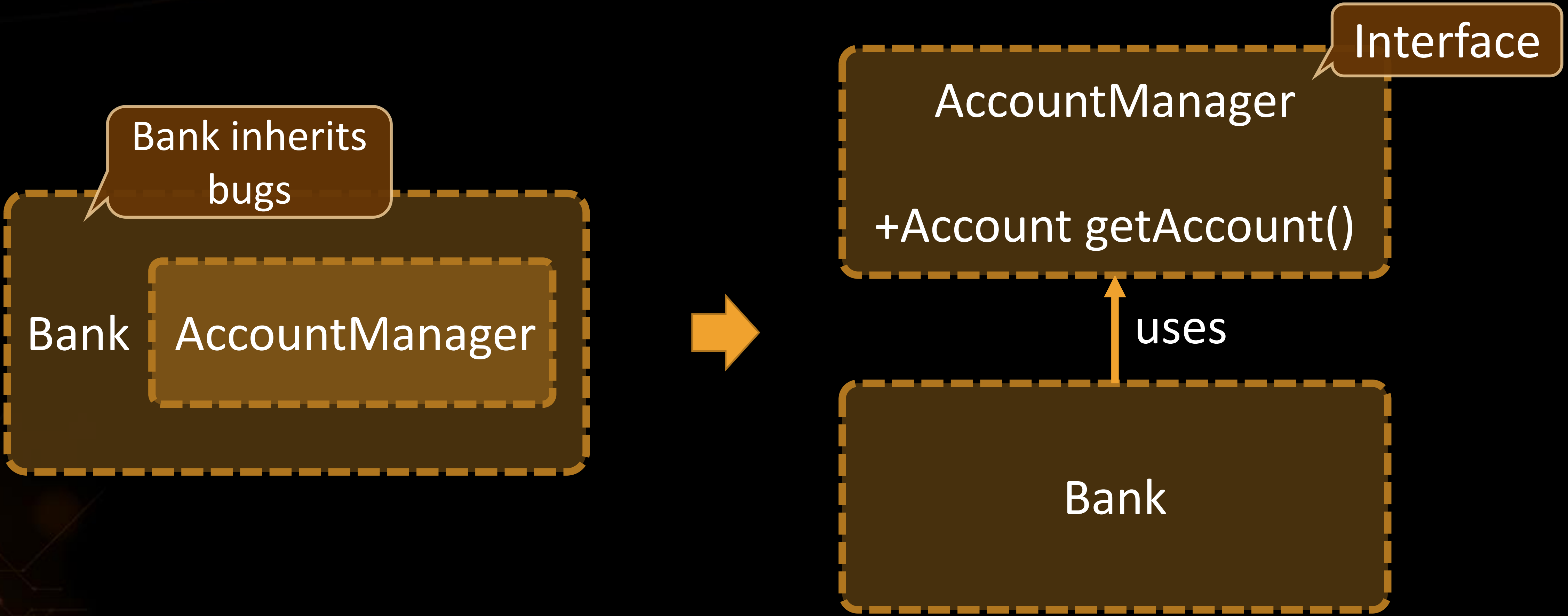
```java
public class Bank {
  private AccountManager accountManager;
  public Bank() {
    this. accountManager = new AccountManager();
  }
  public AccountInfo getInfo(String id) { … }
}
```

Concrete Implementation

Bank **depends** on AccoutManager

Coupling and Testing (2)

- Need to find solution to **decouple classes**



Dependency Injection

- Decouples classes and **makes code testable**

```
interface AccountManager {
  Account getAccount();
}
public class Bank {
  private AccountManager accountManager;
  public Bank(AccountManager accountManager) {
    this. accountManager = accountManager;
  }
}
```

Using Interface

Independent from Implementation

Injecting dependencies

Goal: Isolating Test Behavior

- In other words, to **fixate** all **moving parts**

```
@Test
public void testGetInfoById() {
  // Arrange
  AccountManager manager = new AccountManager() {
    public Account getAccount(String id) { … }
  }
  Bank bank = new Bank(manager);
  AccountInfo info = bank.getInfo(ID);
  // Assert…  }
```

Anonymous class

Fake interface implementation with fixed behavior

Problem: Fake Axe and Dummy

- Test if hero **gains XP** when **target dies**
- To do this, first:
  - Make **Hero** class **testable** (use **Dependency Injection**)
  - Introduce **Interfaces** for Axe and Dummy
    - Interface Weapon
    - Interface Target
  - Create test using a **fake Weapon** and **fake Dummy**

Solution: Fake Axe and Dummy

```
public interface Weapon {
  void attack(Target target);
  int getAttackPoints();
  int getDurabilityPoints();   }
```

```
public interface Target {
  void takeAttack(int attackPoints);
  int getHealth();
  int giveExperience();
  boolean isDead();
}
```

Solution: Fake Axe and Dummy (2)

```
// Hero: Dependency Injection through constructor
public Hero(String name, Weapon weapon) {
  this.name = name;       /* Hero: Dependency Injection */
  this.experience = 0;    /* through constructor */
  this.weapon = weapon; }
```

```
public class Axe implements Weapon {
  public void attack(Target target) { … }
}
```

```
// Dummy: implement Target interface
public class Dummy implements Target { }
```

Solution: Fake Axe and Dummy (3)

```
@Test
public void heroGainsExperienceAfterAttackIfTargetDies() {
  Target fakeTarget = new Target() {
    public void takeAttack(int attackPoints) { }
    public int getHealth() { return 0; }
    public int giveExperience() { return TARGET_XP; }
    public boolean isDead() { return true; } };

                                              //Continues on next slide…
```

Solution: Fake Axe and Dummy (4)

```
//…


  Weapon fakeWeapon = new Weapon() {
    public void attack(Target target) {}
    public int getAttackPoints() { return WEAPON_ATTACK; }
    public int getDurabilityPoints() { return 0; } };


  Hero hero = new Hero(HERO_NAME, fakeWeapon);
  hero.attack(fakeTarget);
  // Assert…
}
```

Fake Implementations

- Not **readable**, cumbersome and boilerplate

```
@Test
public void testRequiresFakeImplementationOfBigInterface() {
  // Arrange
  Database db = new BankDatabase() {
    // Too many methods…
  }
  AccountManager manager = new AccountManager(db);
  // Act & Assert…
}
```

Not suitable for big interfaces

Mocking

- Mock objects **simulate behavior** of real objects
  - **supplies data** exclusively for the test - e.g. **network** data, **random** data, **big** data (database), etc.

```
@Test
public void testAlarmClockShouldRingInTheMorning() {
  Time time = new Time();
  AlarmClock clock = new AlarmClock(time);
  if(time.isMorning()) {
    Assert.AssertTrue(clock.isRinging());
  } }
```

Test will pass only in the morning!

Mockito

- Framework for mocking objects

```
@Test
public void testAlarmClockShouldRingInTheMourning() {
  Time mockedTime = Mockito.mock(Time.class);
  Mockito.when(mockedTime.isMorning()).thenReturn(true);
  AlarmClock clock = new AlarmClock(mockedTime);
  if(mockedTime.isMorning()) {
    Assert.AssertTrue(clock.isRinging());
  }
}
```

Always true

Problem: Mocking

- Include `Mockito` in the project dependencies
- Mock fakes from previous problem
- Implement Hero **Inventory**, holding unequipped weapons
  - method - `Iterable<Weapon> getInventory()`
- Implement Target giving random weapon upon death
  - field - `private List<Weapon> possibleLoot`
- Test Hero killing a target getting loot in his inventory
- Test Target drops random loot

Solution: Mocking

```
@Test
public void attackGainsExperienceIfTargetIsDead() {
  Weapon weaponMock = Mockito.mock(Weapon.class);
  Target targetMock = Mockito.mock(Target.class);
  Mockito.when(targetMock.isDead()).thenReturn(true);
  Mockito.when(targetMock.giveExperience()).thenReturn(TARGET_XP);
  Hero hero = new Hero(HERO_NAME, weaponMock);

  hero.attack(targetMock);
  Assert.assertEquals("Wrong experience", TARGET_XP, hero.getExperience());
}
```

Solution: Mocking (2)

- Create `RandomProvider` Interface
- Hero method
  - `attack(Target target, RandomProvider rnd)`
- Target method
  - `dropLoot(RandomProvider rnd)`
- Mock weapon, target and random provider for test

Summary

- **Unit Testing** helps us build **solid code**
- **Structure** your unit tests – **3A Pattern**
- Use **descriptive names** for your tests
- Use different **assertions** depending on the situation
- **Dependency Injection**
  - makes your classes **testable**
  - **Looses coupling** and **improves design**
- **Mock** objects to **isolate tested behavior**

Unit Testing

SoftUni Foundation

xssoftware

SmartIT

NETPEAK
SEO and PPC for Business

SUPERHOSTING.BG

INDEAVR
Serving the high achievers

telenor

SOFTWARE GROUP

INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE

Questions?

https://softuni.bg/courses/

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License