Building the Service Layer

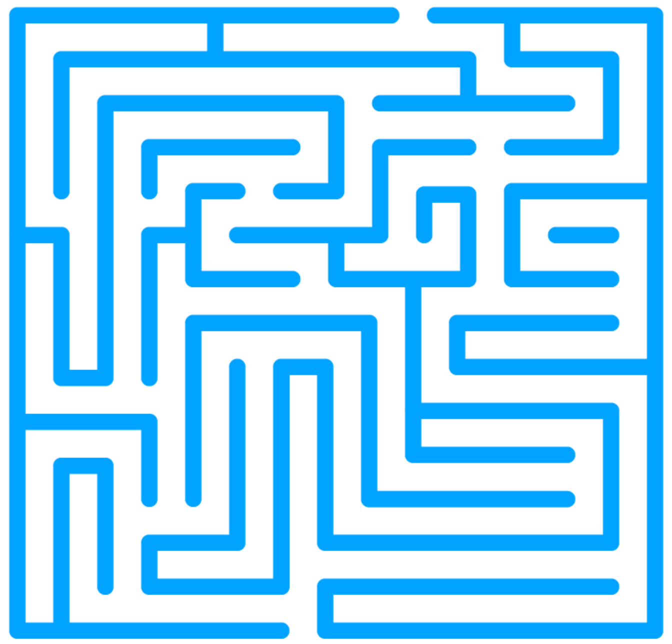
Esteban Herrera
Author

@eh3rrera | eherrera.net

Key Concepts for Testing the Service Layer

- ✓ Designing test scenarios
- ✓ Unit vs. integration tests
- ✓ Rich vs. anemic domain models
- ✓ Centralizing setup and teardown logic with JUnit

Challenges in Testing Business Logic



Starting point uncertainty

Edge case importance ambiguity

Dead-end test conflicts

Root Causes of Testing Challenges

Tightly coupled code

Overlapping tests with global state

Logic errors in design

Conflicting requirements

Designing Test Scenarios

ZOMBIES

Transformation Priority Premise (TPP)

ZOMBIES Acronym

Zero

One

Many (or more complex)

Boundary behavior

Interface definition

Exercise exceptional behavior

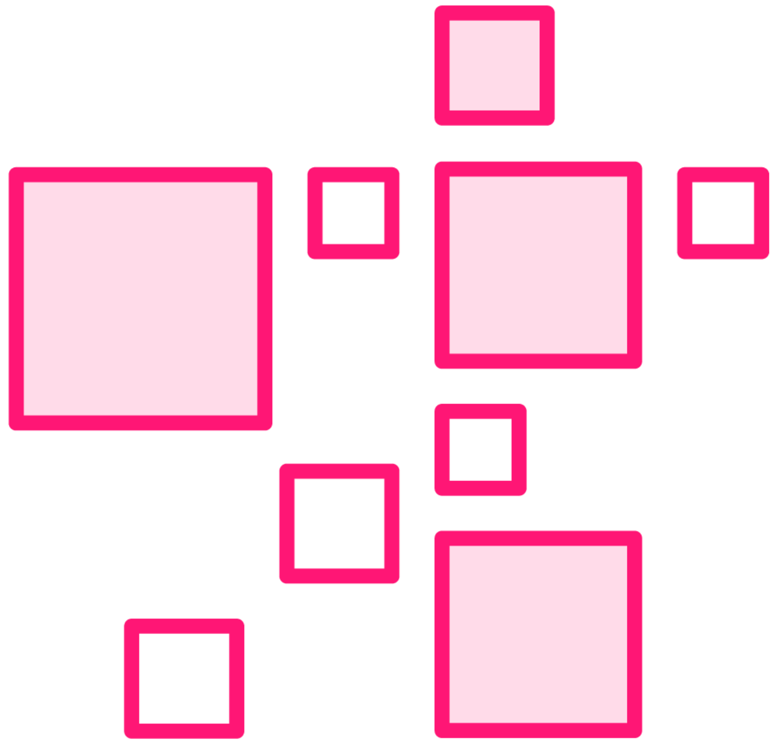
Simple scenarios, simple solutions

James Grenning: http://blog.wingman-sw.com/tdd-guided-by-zombies

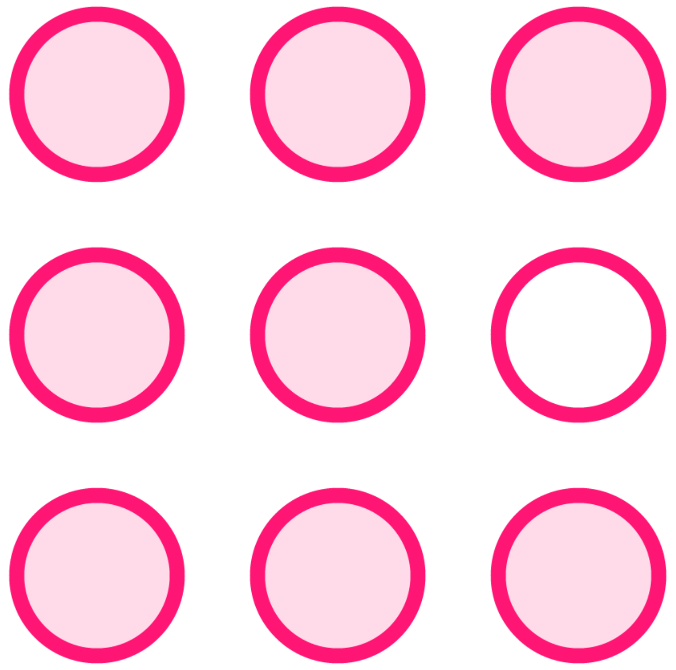
Transformation Priority Premise (TPP)

- no code at all → null
- null → constant
- constant → constant+
- constant → scalar variable
- statement → statements
- unconditional → if
- scalar variable → array
- array → container
- statement → recursion
- if → while
- expression → function
- variable → assignment
- add a case (or else) statement
- add complex algorithm

Two Methods for Testing the Service Layer



Integration tests



Pure unit tests

Integration Test

```java
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {MyService.class, MyRepo.class})
// SpringJUnitConfig(classes = {MyService.class, MyRepo.class})
public class MyServiceIntegrationTest {
    @Autowired
    private MyService myService;
    @Autowired
    private MyRepo myRepository;

    // Your integration test methods
}
```

Unit Test

```
@ExtendWith(MockitoExtension.class)
public class MyServiceUnitTest {
    @InjectMocks
    private MyService myService;
    @Mock
    private MyRepo myRepository;

    @Test
    public void testMethod() {
        when(myRepo.someMethod()).thenReturn("someValue");
        // ...
    }
}
```

Organizing Business Logic

Rich domain model
(data and behavior)

Anemic domain model
(data)

Rich Domain Entity

```java
public class Order {
    private List<OrderLine> lines;
    private boolean isShipped;

    public void ship() {
        if (isShipped) {
            throw new IllegalStateException("Order already shipped.");
        }
        // logic to ship the order ...
        isShipped = true;
    }
}
```

Anemic Domain Entity

```
public class Order {
    private List<OrderLine> lines;
    private boolean shipped;
}
public class OrderService {
    public void shipOrder(Order order) {
        if (order.isShipped()) {
            throw new IllegalStateException("Order already shipped.");
        }
        // logic to ship the order ...
        order.setShipped(true);
    }
}
```

Rich Service, Anemic Entity

```java
public class OrderService {
    public void shipOrder(Order order) {
        if (order.isShipped()) {
            throw new IllegalStateException("Order is already shipped.");
        }
        // logic to ship the order ...
        order.setShipped(true);
    }
}

public class Order {
    private List<OrderLine> lines;
    private boolean shipped;
    // Getters and setters
}
```

Anemic Service, Rich Entity

```java
public class Order {
    private List<OrderLine> lines;
    private boolean isShipped;

    public void ship() {
        if (isShipped) {
            throw new IllegalStateException("Order is already shipped.");
        }
        // logic to ship the order ...
        isShipped = true;
    }
}

public class OrderService {
    public void processOrder(Order order) {
        // some workflow logic ...
        order.ship();
        // some more workflow logic ...
    }
}
```

Anemic Service and Entity, Rich Domain Class

```java
public class OrderService {
    private OrderLogic orderLogic = new OrderLogic();

    public void processOrder(Order order) {
        // some workflow logic ...
        orderLogic.shipOrder(order);
        // some more workflow logic ...
    }
}

public class OrderBusinessLogic {
    public void shipOrder(Order order) {
        if (order.isShipped()) {
            throw new IllegalStateException("Order is already shipped.");
        }
        // logic to ship the order ...
        order.setShipped(true);
    }
}

public class Order {
    private List<OrderLine> lines;
    private boolean shipped;
}
```

Centralizing Setup and Teardown in JUnit Tests

@BeforeEach

@BeforeAll

@AfterEach

@AfterAll

Centralizing Setup and Teardown in JUnit Tests

```java
public class MyServiceTest {
    @BeforeAll
    static void initAll() {
        // Run once before any test is executed
    }

    @BeforeEach
    void init() {
        // Run before each test
    }

    @Test
    void testMethod() {
        // Test logic
    }

    @AfterEach
    void tearDown() {
        // Run after each test
    }

    @AfterAll
    static void tearDownAll() {
        // Run once after all tests are executed
    }
}
```

Nested Classes in JUnit Tests

```java
public class MyServiceTest {
    @BeforeAll
    static void initAll() { System.out.println("Before all tests"); }

    @BeforeEach
    void init() { System.out.println("Before each test in outer class"); }

    @Test
    void testMethod() { /* Test logic */ }

    @Nested
    class MyServiceTestScenario {

        @BeforeEach
        void init() { System.out.println("Before each test in nested class"); }

        @Test
        void nestedTestMethod() { System.out.println("Test in nested class"); }
    }
}
```

Setting up the Test Class

Creating a Ticket

Assigning an Agent to a Ticket

Resolving a Ticket

Closing a Ticket

Updating Tickets

Getting a Ticket by ID

Filtering Tickets

Up Next:

Building the Persistence Layer