Clean TDD

Clean Code & Test Driven Development
Goedemorgen allemaal!

Mijn naam is Maarten Casteels en ben een Software Engineer bij Sopra Steria Group.

Vandaag beginnen we aan een avontuur in de wereld van softwareontwikkeling, waar we de kunst van het schrijven van schone, onderhoudbare code zullen verkennen, duiken in de rituelen van Test Driven Development (TDD), en de krachtige principes van SOLID zullen onthullen.

1. Wie van jullie heeft wel eens gewerkt met programmeercode die moeilijk te begrijpen was door hoe rommelig het was?
2. Heeft iemand ooit een fout in een programma gevonden nadat je dacht dat het klaar was?
3. Wie heeft er wel eens een stukje code geschreven dat later door iemand anders moest worden aangepast en heeft gehoord dat het moeilijk was om aan te passen?
4. Wie vindt het leuk om steeds opnieuw het wiel uit te vinden bij het programmeren?
5. Wie heeft ervaring met unit testing? Denk je dat het nuttig was?

De Reis naar Schone Code

Stel je voor dat je een tovenaar bent in een wereld waar magische spreuken geschreven worden als code. Elke spreuk moet helder, beknopt en krachtig zijn. Schone code is als deze magische spreuken, waarbij elke regel doordacht en duidelijk is, zonder verborgen trucs.

Intentie-verduidelijkende Namen

Gebruik namen die direct vertellen waarom ze bestaan, wat ze doen, en hoe ze gebruikt worden. Als iemand de naam leest, moeten ze niet hoeven raden wat het object doet of bevat. Dit is het fundament van schone code.

```
// Don't
int d; // elapsed time in days

List<Account> accountList = new ArrayList<>();

// Do
int elapsedTimeInDays;
int daysSinceCreation;
int fileAgeInDays;
```

```
List<Account> accounts = new ArrayList<>();
```

Uitspreekbare Namen

Goed gekozen namen in code zijn essentieel voor het begrijpelijk maken van de intentie achter de code. Uitspreekbare namen maken communicatie over software veel eenvoudiger. Het verschil tussen het bespreken van een klasse genaamd `CstmrRcrd` en een klasse genaamd `CustomerRecord` is significant. Het gebruik van volledige, duidelijke en uitspreekbare namen maakt code niet alleen toegankelijker voor anderen, maar ook voor jou zelf wanneer je maanden later terugkijkt op je eigen code. Laten we altijd streven naar duidelijkheid en eenvoud in onze benamingen!

```
// Don't
record CstmrRcrd(Person person) {}

// Do
record CustomerRecord(Person person) {}
```

Vermijden van Primitieve Obsessie

Primitieve obsessie treedt op wanneer we te veel leunen op primitieve datatypes om problemen op te lossen, zoals het gebruik van strings voor telefoonnummers of integers voor datums. Dit leidt vaak tot onnodige complexiteit en fouten. Door in plaats daarvan kleine objecten te gebruiken die gedrag vertonen, zoals `PhoneNumber` of `Date`, maken we onze code meer modulair, begrijpelijk en onderhoudbaar.

```
public boolean isCostHigherThanOtherCost(BigDecimal product1Cost, BigDecimal
product2Cost) {
    return product1Cost.compareTo(product2Cost) > 0;
}


public boolean isCostHigherThanOtherCost(Money product1Cost, Money product2-
Cost) {
    return product1Cost.isHigherThan(product2Cost);
}
```

Functies

Elke functie moet klein zijn en slechts één taak uitvoeren. Als je functies kort en gefocust houdt, worden ze gemakkelijker te lezen en te onderhouden. Dit helpt ook om ze gemakkelijker te testen en minder vatbaar voor fouten te maken.

Minimaliseren van Parameters

Een methode met te veel parameters kan moeilijk te begrijpen en te gebruiken zijn. Probeer functies te ontwerpen die minder parameters nodig hebben. Overweeg om objecten te gebruiken als je merkt dat een functie meer dan drie parameters nodig heeft.

Vermijden van Flags

Het gebruik van boolean flags als parameters voor functies kan vaak wijzen op meerdere verantwoordelijkheden binnen een functie. Overweeg om de functie te splitsen of andere ontwerppatronen te gebruiken om de behoefte aan flags te elimineren.

Commentaren

Commentaren moeten alleen gebruikt worden als dat nodig is. Goede code is meestal zelfverklarend en heeft minimale commentaren nodig. Gebruik commentaren om te verklaren 'waarom' iets gedaan wordt, niet 'wat' er gedaan wordt.

Foutafhandeling

Foutafhandeling is belangrijk, maar het moet niet de logica verstoren. Probeer foutafhandeling te isoleren en maak het een apart onderdeel van de routine. Dit maakt je code schoner en gemakkelijker te onderhouden.

De TDD Uitdaging

Nu, met onze spreuken netjes georganiseerd, hoe zorgen we ervoor dat ze werken zoals bedoeld? Hier komt TDD om de hoek kijken. Denk aan TDD als het testen van je toverspreuken in een gecontroleerde omgeving voordat je ze in het echt gebruikt.

Waarom Tests Schrijven?

Tests schrijven helpt ons om te bevestigen dat onze magie—onze code—precies doet wat we verwachten dat het doet onder verschillende omstandigheden. Dit geeft ons het vertrouwen om nieuwe functies te bouwen en bestaande code te refactoren zonder de bestaande functionaliteit te breken.

Het Schrijven van Tests met JUnit 5

JUnit 5 is de nieuwste generatie van de populaire testing framework voor Java. Het biedt nieuwe mogelijkheden die testen niet alleen makkelijker maken, maar ook krachtiger.

Basis van een JUnit 5 Test

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
class MathOperationsTest {

    @Test
    void testAdd() {
        assertEquals(2, Math.add(1, 1), "1 + 1 should equal 2");
    }
}
```

Deze eenvoudige test controleert of de som van 1 en 1 gelijk is aan 2.

Geavanceerd Testen met JUnit 5

JUnit 5 biedt een scala aan nieuwe annotaties en functionaliteiten die je tests krachtiger en flexibeler maken.

Voorbeeld van `@RepeatedTest` en `@BeforeEach`

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.RepeatedTest;
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import java.util.Random;

class RandomGeneratorTest {

    private Random random;

    @BeforeEach
    void setUp() {
        random = new Random();
    }

    @RepeatedTest(5)
    void testRandom() {
        assertNotEquals(random.nextInt(), random.nextInt(), "Random numbers
should not be the same");
    }
}
```

Dit voorbeeld toont het gebruik van `@BeforeEach` om een frisse instantie van `Random` op te zetten voor elke test, en `@RepeatedTest` om de test meerdere keren uit te voeren.

AssertJ: Vloeiende Assertions voor Java

AssertJ biedt een rijke en intuïtieve set van assertion methodes, waardoor tests makkelijker te lezen en te schrijven zijn.

Voorbeeld van AssertJ Gebruik

```
import static org.assertj.core.api.Assertions.assertThat;

class ListTest {

    @Test
    void testListContains() {
        assertThat(Arrays.asList("apple", "banana", "orange"))
            .as("Check if the list contains the expected fruits")
            .contains("banana")
            .doesNotContain("grape");
    }
}
```

Dit voorbeeld illustreert hoe AssertJ gebruikt kan worden om te verifiëren dat een lijst bepaalde elementen bevat en andere niet, met duidelijke foutmeldingen.

De Red-Green-Refactor Cyclus

We beginnen met een kleine test die faalt (Rood), schrijven dan minimale code om de test te laten slagen (Groen), en verbeteren vervolgens de code zonder de functionaliteit te veranderen (Refactor). Deze cyclus helpt ons om robuuste en onderhoudbare code te bouwen.

De SOLID Stenen van het Kasteel

Als ons rijk groeit, bouwen we kastelen met onze spreuken—dit zijn onze softwareapplicaties. De fundamenten van deze kastelen zijn de SOLID-principes.

Single Responsibility Principle (SRP)

Elke klasse moet één reden hebben om te veranderen. Dit helpt om de klassen klein en beheersbaar te houden, met minder redenen om bugs te introduceren.

Open-Closed Principle (OCP)

Software-entiteiten zoals klassen, modules en functies moeten open staan voor uitbreiding, maar gesloten voor wijzigingen. Dit bevordert het gebruik van interfaces en abstracte klassen.

Liskov Substitution Principle (LSP)

Objecten in een programma moeten vervangbaar zijn met instanties van hun subtypen zonder dat dit het programma verstoort. Dit zorgt voor een sterke contractuele compatibiliteit tussen basis- en afgeleide klassen.

Interface Segregation Principle (ISP)

Geen klasse moet gedwongen worden om interfaces te implementeren die ze niet gebruiken. Dit minimaliseert de afhankelijkheden tussen klassen.

Dependency Inversion Principle (DIP)

Modules op hoog niveau mogen niet afhankelijk zijn van modules op laag niveau. Beide moeten afhankelijk zijn van abstracties. Dit bevordert een ontwerp waarbij de details afhankelijk zijn van beleid, niet andersom.

Epiloog

Terwijl we onze toverkunsten verbeteren met elk van deze technieken en principes, bouwen we niet alleen software, we bouwen een legende. Elke regel code, elke test die we schrijven, draagt bij aan ons verhaal, een verhaal van helderheid, duurzaamheid en creativiteit in codering.

Extra's:

Meer informatie kan je terug vinden in het boek Clean Code: A Handbook of Agile Software Craftsmanship.

– Workshop Materiaal
– Blog Ordina JWorks
– Ordina Stages