PROGRAMMING ADVANCED

(J)UNIT TESTING

WAAROM UNIT TESTEN?

- om kwaliteit te garanderen
- om werkende code niet per ongeluk kapot te maken
- om onze assumpties te valideren
- voor een beter design van onze code
- voor onze eigen gemoedsrust
- om (sneller) naar huis te kunnen ‘s avonds
- om sneller fouten en regressies(dingen die plots niét meer werken) te vinden
- maar ook… om een stuk “vreemde” code beter te leren begrijpen

WAT IS HET?

- in isolatie testen van 1 of enkele blokken van de functionaliteit
- focus op die specifieke code
- evt. ook op het geheel(integratie-test)

UNIT TESTEN SCHRIJVEN OM CODE BETER TE BEGRIJPEN

- (unit) testen zijn het perfecte middel om een api die je gebruik te leren gebruiken
- test je veronderstellingen aan de hand van code

UNIT TESTEN SCHRIJVEN ALVORENS BESTAANDE CODE AAN TE PASSEN

- (unit) testen zijn het perfecte middel bestaande code aan te passen
  - zorg eerst dat je een stuk ongeteste code hebt
  - schrijf een test en zorg dat de test slaagt
  - pas de code aan volgens de nieuwste vereisten
  - zorg dat de test blijft slagen

UNIT TESTEN SCHRIJVEN IMPLICEERT EEN CLEAN DESIGN

- Door unit testen te schrijven, zal je code mooier in elkaar steken. Het is namelijk heel moeilijk om code te schrijven voor een klasse die lelijk in elkaar zit
- Het gaat ook hand in hand met het SINGLE RESPONSIBILITY PRINCIPLE en met SEPARATION OF CONCERNS

SOORTEN TESTEN

- unit test : afzonderlijke class
- functional test: stuk functionaliteit
- integration(e2e) test: gehele systeem

TEST DRIVEN DEVELOPMENT

- Doel: (propere) code schrijven die werkt
- Drie stappen:
  - schrijf een test voor nieuwe functionaliteit (die faalt)
  - pas de code aan zodat de test slaagt
  - controleer of alle testen blijven slagen
  - herhaal dit proces
- LIVE CODING VOORBEELD

TEST DRIVEN DEVELOPMENT

- Uitdagingen
  - neemt iets meer tijd in beslag
  - … maar bespaart tijd én kopbrekers tijdens het bug-fixen
- het is geen silver bullet voor élk probleem
- coverage van reeds bestaande code

HOE DOEN WE DIT?

- Er zijn een heel aantal test frameworks voorhanden
- 1 van de voornaamste spelers is JUnit
- we beschrijven onze scenarios, en hoe we verwachten dat onze code reageert op deze scenarios

LIFECYCLE

- volgorde onbepaald
- elke test wordt uitgevoerd in isolatie
- Annotaties
  - @Test, @Test(expected=..,timeout=..)
  - @Before, @After
  - @BeforeClass, @AfterClass
  - @Ignore

ASSERTIONS

- vergelijkt het resultaat met hetgeen wat we verwachten
- als alle assertions slagen is onze test geslaagd
  - assertEquals, assertTrue, assertNull, assertNotNull, assertFalse, ..

DEMO

be.pxl.les2.api_begrijpen

ISOLEREN VAN HET TE TESTEN OBJECT

- Het te testen object kan dependencies hebben op andere objecten.
  - deze objecten willen we niet aanmaken en configureren om al onze test-paden te dekken
  - dit lossen we op door stubs(doet niets) of mocks(doet wat je het zegt) te maken
- Mocks & Stubs (aparte klasse)
- Alternatief: Mockito en EasyMock voor het makkelijk aanmaken van mocks

JUNIT MOCK EN STUB

- interface vereist
- véél varianten in mocks, dus veel te veel code te schrijven
- maintenance nightmare
- zie ToegangsControleMetJunitMocksTest.java

MOCKITO MOCK

- geen interface vereist
- enkel de code schrijven die je nodig hebt
- flexibel
- zie ToegangsControleMetMockitoMocksTest.java

DEMO

ToegangsControleMetJunitMocksTest.java en ToegangsControleMetMockitoMocksTest.java

TESTING EXCEPTIONS

- Als onze code in bepaalde gevallen een exception gooit, kunnen we dit ook testen

```
@Test(exception=FooException.class)
public void testIfCodeThrowsMyException(){
  testedObject.doSomethingThatThrowsMyException();
}
```

- Als we in de @Test-annotatie géén exception meegeven, zal de test falen, aangezien een niet-verwachte exception gesmeten wordt

TESTING EDGE CONDITIONS AKA BOUNDARIES

- Bij het schrijven van testen is het heel belangrijk dat je je edge-condities test. Dit zijn de randgevallen, bvb. test de functie `isMeerderjarig(int leeftijd){return leeftijd > 18;}`
  - `assertTrue(isMeerderjarig(22));`
  - `assertFalse(isMeerderjarig(9));`
  - `assertTrue(isMeerderjarig(18))` -> FALSE

DEMO

CalculatorPowerOfTwoTest

GEPARAMETERISEERDE TESTEN

- Junit laat het toe om geparameteriseerde testen te schrijven
- Zo kan je snel je code testen met verschillende waarden
- @RunWith(Parameterized.class)
- constructor met argumenten
- @Parameterized.Parameters
- zie PrimeNumberCheckerTest.java

MOCKITO

- http://mockito.org/
- @RunWith(MockitoJUnitRunner.class)
- @Mock
- @InjectMocks

POWERMOCK

- (!) Veelgevraagd tijdens een job interview: Hoe test je static methods
  - deze zijn moeilijk om te testen (net omdat ze static zijn)
    - soms kan je niet anders, omdat je bvb een static method van een aangeleverde library gebruikt
- Powermock maakt het mogelijk om deze alsnog te testen
- https://github.com/jayway/powermock

FLUENT API'S

- http://joel-costigliola.github.io/assertj/
- Fluent testing
  - korter bij de natuurlijke taal dus leesbaarder
- Makkelijk om collections te testen
  - zie MovieCharacterAssertJTest.java

DEMO

MovieCharacterAssertJTest.java

INTEGRATIE MET JE ONTWIKKEL-OMGEVING

- IntelliJ
- Eclipse

INTEGRATIE MET MAVEN

- mvn test
- mvn test -Dtest=TestName

WIL JE NOG MEER WETEN?

- De referentie: Test-Driven Development (Kent Beck) https://www.bol.com/nl/p/test-driven-development/1001004001812742/?country=BE
- https://app.pluralsight.com/library/courses/java-unit-testing-junit/exercise-files