Mer om testing

Kodeeksempel – Automatisk karakter

- Mål: Demonstrere hvordan TDD kan hjelpe med å naturlig bryte ned funksjonalitet i små testbare enheter, samt mer om Mocking

- La oss lage en metode for å automatisk beregne karakter for en gitt besvarelse
  - 90 - 100 poeng → A
  - 80 - 89 poeng → B
  - 60 - 79 poeng → C
  - 50 - 59 poeng → D
  - 40 - 49 poeng → E
  - Ellers F

Parameteriserte Enhetstester

- Når vi må ha mange test-scenarier innenfor en gitt enhetstest kan det bli tungvint skrive og opprettholde testene
- I slike tilfeller kan vi skille ut deler av testen og sende dem med som parametere (En parameterisert test)

- Vi går bare gjennom én måte å lage parameteriserte tester, men det finnes flere

Parameteriserte tester

- Koden under tester om forskjellige år er skuddår
  - Delbar på 4, med unntak hvis den er delbar på 100, men er det hvis den i motsetninger delbar på 400
  - Her vil det finnes mange scenarier som bør testes...

```
GregorianCalendar YearUtils = new GregorianCalendar();

@Test
public void testLeapYear() {
   Assertions.assertTrue(YearUtils.isLeapYear(2000)); // Divisible by 400
   Assertions.assertFalse(YearUtils.isLeapYear(1900)); // Divisible by 100 but not 400
   Assertions.assertTrue(YearUtils.isLeapYear(2004)); // Divisible by 4 but not by 100
   Assertions.assertFalse(YearUtils.isLeapYear(2001)); // Not divisible by 4
   // ...
   // ...
}
```

Parameteriserte tester

- Denne koden løser dette med en parameterisert test

- Test-scenariene skilles i en egen statisk metode - testLeapYearParameters()
  - «Stream of arguments» med året som skal testes og forventet resultat

- Testen defineres som en @ParameterizedTest()
  - name definerer en label for hvert scenarie
  - @MethodSource("testLeapYearParameters") definerer at testen skal ta argumentene i testLeapYearParameters() som parametere
  - Vi må definere parametere for testen med tilsvarende datatyper (int year, boolean expectedIsLeapYear)
- Act og Assert benytter parameterene for å abstrahere testing av alle scenarier
  - String.format() i assertEquals() er valgfritt for å definere output når en test feiler

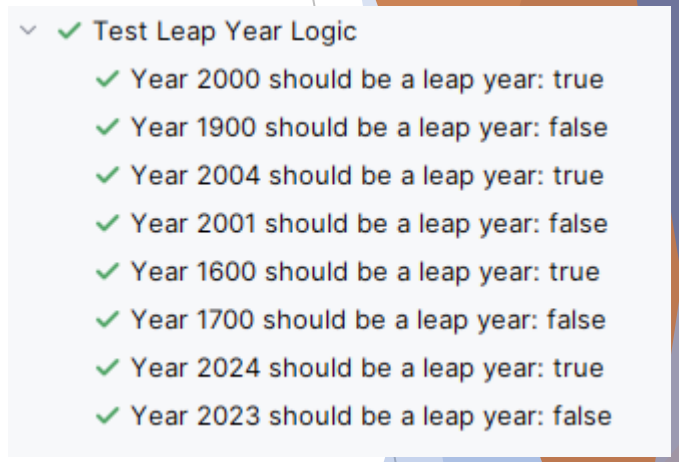
```
private static Stream<Arguments> testLeapYearParameters() {
    return Stream.of(
            Arguments.of(2000, true), // Divisible by 400
            Arguments.of(1900, false), // Divisible by 100 but not 400
            Arguments.of(2004, true),  // Divisible by 4 but not by 100
            Arguments.of(2001, false), // Not divisible by 4
            Arguments.of(1600, true),  // Divisible by 400
            Arguments.of(1700, false), // Divisible by 100 but not 400
            Arguments.of(2024, true),  // Divisible by 4 but not by 100
            Arguments.of(2023, false)  // Not divisible by 4
    );
}

@ParameterizedTest(name = "Year {0} should be a leap year: {1}")
@MethodSource("testLeapYearParameters")
@DisplayName("Test Leap Year Logic")
public void testLeapYear(int year, boolean expectedIsLeapYear) {
    // Act
    boolean actualIsLeapYear = YearUtils.isLeapYear(year);

    // Assert
    Assertions.assertEquals(expectedIsLeapYear, actualIsLeapYear,
            String.format("Year %d expected to be leap year: %b but got: %b",
                    year, expectedIsLeapYear, actualIsLeapYear));
}
```

Parameteriserte Enhetstester

- Fordeler
  - Vi definerer typisk act og assert bare én gang i testen (mens arrange og test-scenariene samles for seg selv)
  - Vi kan enkelt utvide med flere scenarier senere uten å endre testlogikken
  - Vi kan definere en egendefinert label for hvert scenario, som kan gjøre det enklere å feilsøke
- Ulemper
  - Mye mer overhead for å skrive testen
  - Særdeles mindre intuitivt å lese testene (Må hoppe mye frem og tilbake)
- Jeg vil anbefale å bli vant til å benytte parameteriserte tester, spesielt hvis det er mange test-scenarier



Kodeeksempel – Parameteriserte tester

- Gjøre om en av enhetstestene våre til en parameterisert test

- Skille ut scenarier i en egen statisk metode
- Definere testen som en @ParameterizedTest
- Definere og benytte parametere i act og assert delen av testen

Workshop

- Forsøk å følge TDD for en feature i prosjektet deres
  - Gjerne skriv kode sammen og diskuter
- Alternativt kan du skrive TDD for FizzBuzz
  - Tall som er delbare på 3 → «Fizz»
  - Tall som er delbare på 5 → «Buzz»
  - Tall som er delbare på både 3 og 5 → «FizzBuzz»
  - Ellers returner input-tallet (som String)
- Hvis du/dere står fast, spør

1. Skriv en (feilende) test
2. Få testen til å kjøre
3. Kjør testen
4. Få testen til å passere
5. Refaktorer
6. Tilbake til 1.