Testing / Enhetstesting - Introduksjon

Agenda

- Testing i software engineering
- Enhetstesting
- Equivalence partitions
- Skrive enhetstester med JUnit

Hva er testing i Software Engineering

- Det er overordnet fire "kategorier" av testing i Software Engineering (Sommerville, 2021):

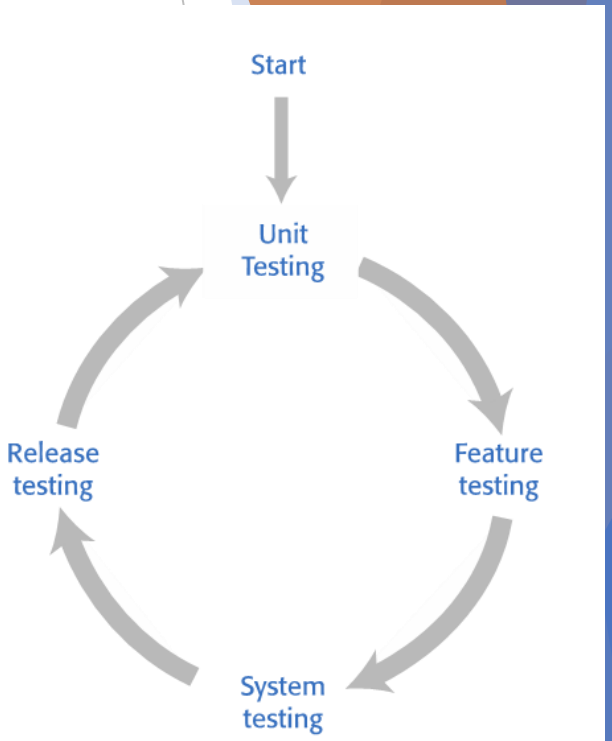
- Funksjonell testing
  - Teste (kode)funksjonalitet for å finne bugs og sjekke/bevise at ting funker som tiltenkt
- Brukertesting
  - Teste produktet med brukere for å finne ut hvor nyttig og "behagelig" produktet er å bruke
- Ytelsestesting
  - Teste om produktet er raskt nok og tåler belastning
- Sikkerhetstesting
  - Teste at produktet ikke kan misbrukes for uønskede effekter (mot leverandør og/eller brukere)

Avgrensning av testing i kurset

- Vi skal hovedsakelig forholde oss til **funksjonell testing** i dette kurset
  - Funksjonell testing kan gjøres forholdsvis enkelt og kan/skal gjøres under utvikling av produktkode
  - Brukertesting krever brukere å teste med. Nyttig, men kan bli mye overhead
  - Ytelsestesting er viktig når man faktisk skal selge et produkt. Det forventes ikke av dere.
  - Sikkerhetstesting: Større fokus i kurs som Teknologiprosjekt og Datasikkerhet i Utvikling og Drift
- Vi stopper dere ikke fra å prøve på de andre formene for testing
  - Men prioriter funksjonell testing

Funksjonell testing

- Typer funksjonalitet i et produkt
  - Fullstendig produkt → «Features» som sammen utgjør produktet → «Units/enheter» som sammen utgjør en Feature
- Unit/Enhet – En (typisk liten) bit med kode som har **én** klar oppgave
- Typer funksjonell testing
  - Enhetstesting – Teste at hver enhet i isoloasjon fungerer som tiltenkt
  - Integrasjonstesting – Teste at enheter fungerer når de jobber sammen
    - Kalles feature-testing hvis enhetene utgjør en hel feature
  - Systemtesting – Teste produktets features fungerer sammen som forventet
  - Release testing - Teste at produktet fungerer i tiltenkte bruksmiljøer (f.eks. på forskjellige datamaskiner)



(Sommerville, 2021)

Enhetstesting - Generelt

- Vårt fokus vil ligge mest på Enhetstesting

- Hva er egentlig en enhet? (Litt flertydig...)
  - Typisk en funksjon/metode
    - Kan også være en gruppe funksjoner/metoder eller en klasse
  - Har bare én oppgave - Kan ikke brytes ned mer
    - Er ikke egentlig en «enhet» dersom den kommuniserer med noe «variabelt» og utenfor seg selv
      - Filsystem, database, nettverk osv.

- Enhetstesting – Vi sjekker at forskjellige inputs gir de resultatene vi forventer
  - Det vi forventer er basert på tiltenkt funksjonalitet

Enhetstesting – Equivalence partions

- For å sjekke at enheter fungerer som forventet må vi teste et gjennomtenkt utvalg inputs
  - Begrepet «Equivalance partions»
- Vi bør teste for ...
  - Et variert utvalg av riktige inputs (normalt bruk)
    - Bekrefter at «det fungerer som det skal»
  - Et utvalg av inputs som kan føre til feil eller som ikke skal aksepteres (unormalt bruk)
    - Kontrollerer at unormalt/uakseptabelt bruk blir håndert på en god måte
- Det er typisk ikke mulig eller fornuftig å teste alle mulige inputs
  - Vi må teste et utvalg som generelt representerer forskjellige «grupper» av inputs

Enhetstesting – Equivalence partions

```
def namecheck (s):

        # Checks that a name only includes alphabetic characters, - or
        # a single quote. Names must be between 2 and 40 characters long
        # quoted strings and -- are disallowed

        namex = r"^[a-zA-Z][a-zA-Z-']{1,39}$"
        if re.match (namex, s):
                if re.search ("'.*'", s) or re.search ("--", s):
                        return False
                 else:
                        return True
        else:
                return False
```

(Sommerville, 2021)

Enhetstesting – Equivalence partions

Correct names 1
The inputs only includes alphabetic characters and are between 2 and 40 characters long.

Correct names 2
The inputs only includes alphabetic characters, hyphens or apostrophes and are between 2 and 40 characters long.

Incorrect names 1
The inputs are between 2 and 40 characters long but include disallowed characters.

Incorrect names 2
The inputs include allowed characters but are either a single character or are more than 40 characters long.

Incorrect names 3
The inputs are between 2 and 40 characters long but the first character is a hyphen or an apostrophe.

Incorrect names 4
The inputs include valid characters, are between 2 and 40 characters long, but include either a double hyphen, quoted text or both.

(Sommerville, 2021)

Enhetstesting – Equivalence partitions
Gode retningslinjer (1)

- ***Test edge cases***
  If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.

- ***Force errors***
  Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.

- ***Fill buffers***
  Choose test inputs that cause all input buffers to overflow.

- ***Repeat yourself***
  Repeat the same test input or series of inputs several times.

(Sommerville, 2021)

Enhetstesting – Equivalence partitions Gode retningslinjer (2)

- ***Overflow and underflow***
  If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.

- ***Don't forget null and zero***
  If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.

- ***Keep count***
  When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.

- ***One is different***
  If your program deals with sequences, always test with sequences that have a single value.

(Sommerville, 2021)

Lage Enhetstester

- **Hver enhetstest er en bit med kjørbar kode definert av enhetens utvikler**

- En enhetstest er basert på å sammenligne to verdier (Er de like?)
  - Den ene verdien er resultatet av enhetens kjøring
  - Den andre er verdien utvikleren forventer av enhetens kjøring

Lage Enhetstester

- Verdisammenligning gjør feilsøking ganske enkelt
  - Hvis like verdier → «Yippi! Enheten funker!»
  - Hvis ulike verdier → «Ok. Noe er feil med enheten og må fikses.»
  - En enhet er typisk bare noen få linjer med kode
  - Vi kan typisk også se nøyaktig hvilken input som medbringer feilen

- Merk: En enhetstest må være deterministisk for å være nyttig (Samme input gir alltid samme resultat)
  - Hvis ikke er det vanskelig å feilsøke...
  - Hvis det som testes er basert på noe tilfeldig/uforutsigbart, må dette gjøres statisk i testen

Enhetstester - Rammeverk

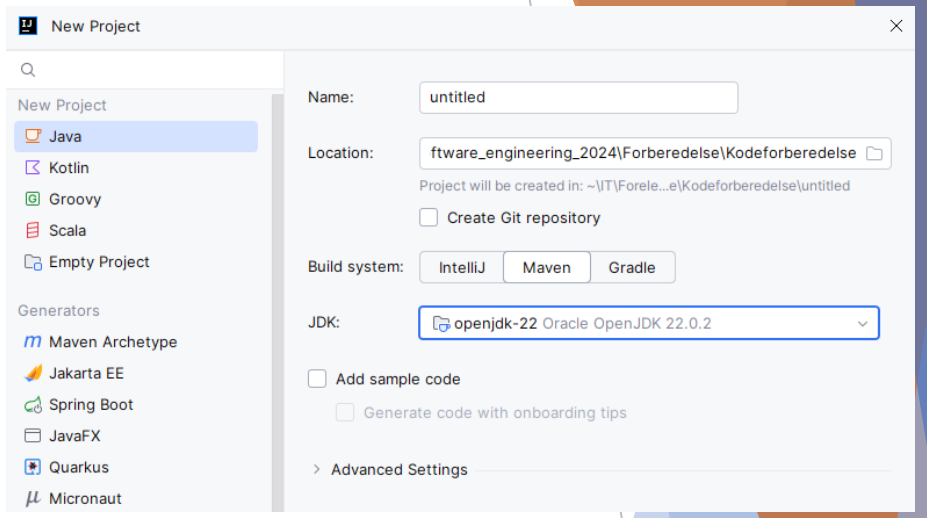
- Java – JUnit
- Python – PyTest
- .NET – NUnit
- Javascript – Mocha

- Må typisk importeres/settes som en dependency for å kunne benytte

- Dere står frie til å velge språk/rammeverk i prosjektet, men...
- Forelesningseksempler vil benytte JUnit i Java

JUnit – Oppsett av prosjekt

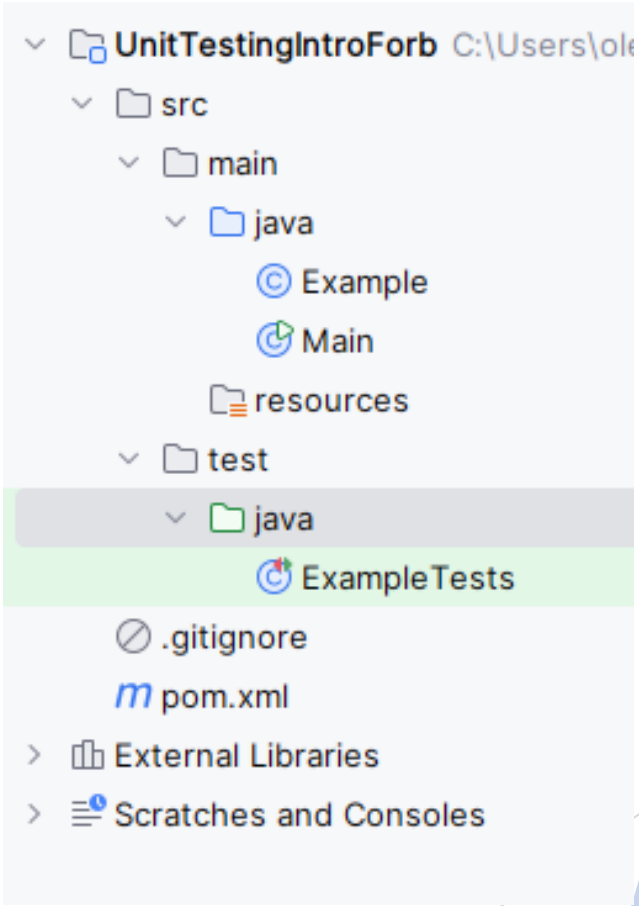
- IntelliJ IDEA
- Krever et byggeverktøy
  - Jeg benytter Maven
    - Litt mer lettvektig enn Gradle
- Vi må legge til JUnit som en dependency i filen pom.xml




```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.2</version>
  </dependency>
</dependencies>
```

JUnit – Prosjekt struktur

- To viktige mapper under src (generert av Maven):
  - main/java → Vanlige Java-klasser
  - test/java → Java-klasser for enhetstesting
- Det er vanlig å lage en test-klasse for hver Javaklasse som har testbar funksjonalitet
  - Men ikke et direkte krav...
  - Navngis typisk "<Orginalt klassenavn>Tests"
- Merk at test-klassers navn **MÅ** slutte på «Test» eller «Tests» for å kunne tolkes i enkelte sammenhenger



JUnit – Skrive Enhetstester

- Gitt denne (elendige) enheten for passordsjekking
  - Hva burde vi teste? (Equivalence partitions)

```
public class Example {
    public boolean scuffedPasswordCheck(String password) {
        if (password.equals("Password12345")) {
            return true;
        }
        else return false;
    }
}
```

- 2 ting...
  - Passord er riktig → true
  - Passord er feil → false

JUnit – Skrive Enhetstester

- Vi lager en test-klasse med et tilsvarende navn som den originale (her ExampleTests)

- En gitt test er definert som en void-metode
  - Annortert med @Test
  - Navnet på metoden er valgfritt, men det kan være lurt å navngi den etter hva som skal testes
  - @DisplayName() er også valgfritt men hjelper for oversikt i kjøreresultater, samt lesbarhet av kode

```
public class Example {
    public boolean scuffedPasswordCheck(String password) {
        if (password.equals("Password12345")) {
            return true;
        }
        else return false;
    }
}
```

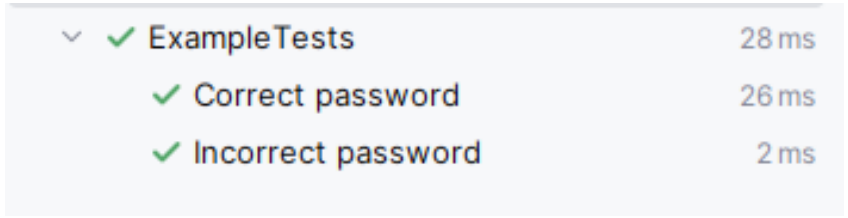
```
public class ExampleTests {



}
```

JUnit – Skrive Enhetstester

- Tester defineres av «Assertions»
  - Sammenligning av verdier
  - Assertions.assertEquals() sjekker at to verdier er like
    - Første verdi representerer det som er forventet
    - Andre verdi er det faktiske resultatet
  - Hvis resultatet er en boolean verdi, kan du benytte
    - Assertions.assertTrue()
    - Assertions.assertFalse()
  - Slikt som assertThrows() finnes også...
    - Teste exceptions




```java
public class Example {
    public boolean scuffedPasswordCheck(String password) {
        if (password.equals("Password12345")) {
            return true;
        }
        else return false;
    }
}
```

```java
public class ExampleTests {

    @Test
    @DisplayName("Correct password")
    public void scuffedPasswordCheckCorrect() {
        Example example = new Example();

        Assertions.assertEquals(true, example.scuffedPasswordCheck("Password12345"));
        Assertions.assertTrue(example.scuffedPasswordCheck("Password12345"));
    }

    @Test
    @DisplayName("Incorrect password")
    public void scuffedPasswordCheckIncorrect() {
        Example example = new Example();

        Assertions.assertEquals(false, example.scuffedPasswordCheck("wrongpassword"));
        Assertions.assertFalse(example.scuffedPasswordCheck("wrongpassword"));
        Assertions.assertFalse(example.scuffedPasswordCheck("bingus"));
    }
}
```

Arrange, Act, Assert

- Når vi skriver tester er det fordelaktig å følge en standardisert struktur
- En test kan konseptuelt deles opp i tre deler
  - Arrange
    - Oppsett av ressurser som er nødvendig for funksjonaliteten som testes
  - Act
    - Utføre funksjonaliteten som testes og ta imot resultatet
  - Assert
    - Verifiser resultatet opp mot hva vi forventer

Arrange, Act, Assert

```
@Test
@DisplayName("Correct password")
public void scuffedPasswordCheckCorrect() {
    //Arrange
    Example example = new Example();

    // Act
    boolean result = example.scuffedPasswordCheck("Password12345");

    // Assert
    boolean expected = true;
    Assertions.assertEquals(expected, result);
    Assertions.assertTrue(result); // Better alternative
}
```

Merk at bruk av assertEquals() er unødvendig her men illustrer at vi kan skille expected som en egen variabel

Arrange, Act, Assert

Det er lov å gjøre Act og Assert samtidig så lenge det forblir oversiktlig

```
@Test
@DisplayName("Correct password")
public void scuffedPasswordCheck() {
    // Arrange
    Example example = new Example();

    // Act and Assert
    Assertions.assertEquals(true, example.scuffedPasswordCheck("Password12345"));
    Assertions.assertTrue(example.scuffedPasswordCheck("Password12345")); // Better alternative

}
```

Merk igjen, assertEquals() er fremdeles unødvendig her, men er med som et eksempel

JUnit – Skrive Enhetstester

- Lag en unik test for hver equivalence partion
  - Mange assertions som tester mye forskjellig gjør det vanskelig å se hva som faktisk passerer/feiler
  - Det er motsetning fornuftig å ha flere assertions som tester samme type resultat
- Hver test bør være beskrivende så det er lett å forstå hva den gjelder
  - Metodenavn
  - Display name

Kodeeksempel – Enhetstester

- Sette opp prosjekt med Maven
- Legge til JUnit som en dependency i pom.xml

- Opprette en klasse InformationChecker
  - Opprette en metode controlAge() for kontrollere at en gitt alder er mellom 0 og 120

- Opprette en test-klasse InformationCheckerTests
  - Opprette passende test-metoder ut ifra controlAge() sin funksjonalitet

- Se på testing av void-metoder og exceptions

Fordelene med testing

- Vi bekrefter for oss selv at jobben er ferdig
- Knerter bugs tidlig
  - Hvis ikke må man lete hardere senere …
- Opprettholdbarhet
  - Senere endringer på en enhet kan ødelegge tidligere definert funksjonalitet
  - Testene informerer oss hvis noe ryker
- Tester er et godt supplement til dokumentasjon
  - Vi kan se hvordan enheten er ment til å fungere
  - Beviser samtidig at enheten fungerer slik den skal (Vi kan stole på den)