JUnit

Varför testbar kod?

Komplexa klasser blir snabbt oöverskådliga, och fel tenderar till att smyga in.

Om man testar sin klass olika metoder så säkerställer man att klassen ska göra vad man vill den ska göra.

Tester säkerställer även att metoder är små, dvs ingen metod ska ha så många rader att det blir otydligt vad den ska göra.

Om en metod ska spotta ut t.ex. en Double, så ska den med minsta antalet kodrader bearbeta variabler och returnera svaret.

En metod som först måste addera, sen beräkna med t.ex. Math.pow för att sen subtrahera innan den kan svara är således för stor då flera delar kan gå fel.

Bättre lösning på problemet är att bryta ut koden i tre metoder. En för addera, en för beräkna och en för att subtrahera.

På detta sätt får man skriva tre tester, ett test för varje metod, men då vet man exakt vad man vill ha för svar på varje metod och säkerställer att metoden genererar det svaret med de parametrarna man skickar in.

Typer av test

Det finns flera olika typer av test:

- Unit test
- Integration test
- Acceptence test
- System test

För att skapa en test klass, måste klassen sluta på antingen *Test.java eller *Tests.java

Konventionen är att testklassen ska ha samma namn som den klassen den ska testa och Test.

Klassen School blir då SchoolTest.java

Använder man Maven, skall alla testklasser läggas under src/test/java

Med kommandot mvn test körs enbart testklasser.

En testmetod skapas genom att annotera metoden med:
@Test
Public void.....

Det finns även andra typer av test, som:
@RepeatableTest(5)
Public void.....

vilket gör att ett test körs 5 gånger.

Om man har metoder som persisterar data till en databas,
Är det av förklarliga skäl inte lämpligt att testa metoden mot databasen.

Men metoden måste kunna testas ändå.

Man får då helt enkelt "Mocka" sin metod, vilket innebär att man simulerar en databas som enbart finns medans testerna körs.

H2 in-memory database är en vanlig databas att testa emot, men oftast räcker den eller andra in-memory databaser inte till.

Räddningen är då generiska testcontainrar.

Det är container som tar ner en Dockerfile med aktuell databas man vill ha.

När man kör testerna nu, så kommer först Maven hämta hem databasen i form av en container, sen kommer alla tester anropa denna container, för att därefter kastas.

Då är metoderna testade, produktionsdatabasen intakt och alla är nöjda.

Test Driven Development
TDD

TDD är ett sätt att utveckla sina program på.

Genom att noggrant planera programmet, ner i sina beståndsdelar, börjar man med att skriva tester för varje metod man tänkt skall finnas i klassen.

Genom att skriva testerna först, så vet man vad ens metoder ska göra och minimerar riskerna ännu mer att man skriver metoder med sidoeffekt.

Om man bygger med CI / CD, så kommer man få fel i CI pipe:n, om ett test failar, vilket gör attt man inte riskerar få ut ofärdig kod till Deployment.

Den stora nackdelen är att utvecklarna kan sitta fast ett par dagar med att skriva tester som sen kanske visar sig kommer få skrivas om ifall nya features tillkommer.