**Talepapir State Design pattern + JUnit**

**State Design Pattern**

* Man kan risikere, at ens program har forskellige ”stadier”
* F.eks. en telefon = lydløs tilstand eller lyd
  + 2 forskellige stadier
* Kender også fra tråde
  + Sleep
  + Wait
  + Blocked
  + Og lignende
* Altså har vi et objekt
  + Opfører sig forskelligt, alt efter hvilken tilstand
* Man kan sætte stadiet inde i objektet, i form af if-statements
  + Så skal vi tjekke, hvilket stadie, den vil opnå, hver gang den ændres
* Skal programmet udvides med flere stadier
* Flere if-statements
* BØVL nej tak – det kan blive stort
* I stedet kan man lave klasser for hvert mulig stadie, objektet kan være i
  + Selve objektet kun kender til ét samlet interface
  + Stadierne nedarver fra
* Metoderne i interface bliver herved state-specific
  + Ændrer opførsel alt efter stadie der tages i brug
* De forskellige parter i design opsættet er derfor
  + Vores object = context
    - Har et stadie
  + Stadier er et interface, som giver fælles metoder, til nedarver
  + Dem som nedarver, er de forskellige mulige stadier, som context kan være i
  + 1 klasse til hver tilstand
* Når context bruger state-specific metode
* Kalder nedarvninger
* Context kan sendes med som parameter i state klassen
* Andre gange kan man have et variabel i nedarvningsklasserne af typen Context

**JUnit**

* Pointen med JUNIT
  + Tjekke ens kode virker
    - Både forventet
    - Også test om det giver fejl ved uforventet
* Vi har lært
  + Scenario-testing
  + Integration testing
  + Unit testing
    - Hver kategoriseres i black/white box testing – enten én eller begg
* Black-box testing kan man sige er bruge POV
  + Forestille sig en sort kasse = vores program
  + Man kan ikke se i den sorte kasse
    - Eneste måde at tjekke programmet, er ved at give softwaren et input
    - Software giver output
      * Se om det er det rigtige man får tilbage
    - MAN KAN ALTSÅ IKKE SE SELVE FUNKTIONALITETEN I KODEN
* Det kan man tilgengæld ved white-box testing
  + Dette er programmørens POV
    - Se kode = selv registrere, hvor noget er kritisk, og teste dette
  + Kan bl.a. gøres ved branch testing, hvor man tester alle mulige branches, som koden kan opnå
    - Gøres ved = tjekke om outcome både er forkert/rigtigt,
    - Teste med forskellige værdier
      * Man skal sørge for at få alle tænkelige scenarier dækket
* Unit-testing kan både være black- og white box
  + Typisk white, så man kan teste ned i små detaljer i koden
* Det kan derfor være smart både at tjekke boundaries, altså afgrænsninger i form af
  + BONDARY VALUE ANALYSIS
    - * + Start/End
        + Lower/Upper
        + Maximum/minimum
        + Just inside/just outside values
        + Empty/non-empty
        + Non-full/full
* Har man en stor range, f.eks. at en liste indeholder 1000 elementer, så tjekker man ikke samtlige 1000
  + Equivalence partitioning: bryder data op i mindre lige store bider, og tester på disse, og antager at resten er korrekt
* Fordel: Minimere unødvendige teste = God rettesnor
* ZOMB+E:
  + - Zero: hvad sker der hvis der er 0 elementer i en liste, eller man indsætte 0, alt der inkludere 0
    - One: Hvad med hvis man indsætter 1, eller 1 element (ligesom zero)
    - Many: hvad sker der hvis der er mange
    - Boundary: basically boundary value analysis
    - Exceptions: Vi tjekker vores exceptions; bliver de thrown som vi forventer
      * Der er typisk en boarder i mellem 0 og 1, og man bruger altså noget boundary value analysis, i forhold til f.eks. empty/non-empty
      * Many kan dække over equivalence partitioning, da man tester mange partitions, som skal dække flere ting

KODE TIL JNUIT

* Givet interface QueueADC
* Forskellige klasser nedarver
* Hertil brugt JUnit til at teste funktionalitet ved hver metode har været korrekt
* Inde i QueueTest, f.eks. ved den første test
  + Setup = Arrange = objekt oprettes
    - Metode laves i toppen, til at oprette objekt
    - Når metoden bliver kaldt i test, bliver objektet operettet
  + Act = operere på objektet
    - Første test indsætter vi kapacitet i køen
  + Assert = det sidste = validere at metoden under visse betingelser opfører sig korrekt
  + Eksempel linje 35 = tester om der bliver throwet exception
    - Tester man den, siger den korrekt
      * Hvis kode fra klassen
* Når man tester en klasse som denne, oplagt at lave boundary value analysis
  + Tjekker boarders ved listen
  + Start slut på listen
  + Må man tilføje mere, må man fjerne mindre
  + Er listen fyldt, er listen tom
* Udgangspunkt i testningen har været ZOMB+E, som dækker over dette
* Z (zero) = linje 30
* O (one) = linje 67
* M (many) = linje 77
* B (boundaries) = linje 45
* E (exceptions) = Linje 36