**Question 4A:** State design pattern

Describe the pattern using UML diagrams.

* What is the purpose?
  + Vi kender det også fra tråde, de kan f.eks. være i sovetilstand, wait tilstand, blokeret tilstand etc.
  + Man vil altså gerne undgå at have mange if-statements. Hvis f.eks. ens program skulle udvides, så skal man udvide antallet af if-statement – og det er træls. I stedet har vi forskellige nedarvningsklasser, som specificerer hvad det specifikke stadie gør. Når så et stadie automatisk ændres til et nyt, så skal metoderne automatisk følge med på det nye stadie.
  + Pointe: Hvis et objekt ændre sin ”indre” adfærd afhængig af det ”indre” stadie den er i, så er det en fordel at bruge state design pattern.

Disse stadiers opførsel specificeres i separate klasser, hvor hver klasse gør noget forskelligt. De nedarver fra samme interface, således objektet kun skal fokusere på fælles metoderne, men stadiet den er i bestemmer udkommet.

Man undgår således at have en masse if-statements, som først skal tjekke, hvilket stadie objektet er i

* What are the different parts involved?
  + De forskellige parter I design opsættet er derfor vores objekt, vores context, som har et stadie. Stadiet er et interface, som giver fælles metoder, til de som nedarver. Dem som nedarver, er så de forskellige mulige stadier, som contexten kan være i - 1 klasse til hver tilstand.
  + Når context bruger en statespecifik metode, så kalder den altså ned på nedarvningerne
  + Context kan sendes med som parameter i State klassen og ned til nedarvningerne, andre gange kan man have et variabel i nedarvningsklasserne af typen Context.
* Pros and cons:

Present your own example from an assignment, SEP2, or exercise

* Use UML and describe how your solution follows the pattern
* Show Java code – how the pattern is implemented.
* Show Java code for how the pattern can be used.

**Question 4B:** JUnit testing

Pointen med testing er at tjekke at ens kode virker, både med de forventede outcomes af ens kode, men også at teste det, som man ikke forventer skal ske.

Som vi så pænt har fået af vide: ”brugeren vil altid prøve at ødelægge ens program på de mærkeligste måder”, og derfor skal vi teste det!

Der er forskellige måder at teste på, og vi har arbejdet med 2 ting dette semester:

* + Scenario testing: Her tjekker man scenarier, features og user stories
  + Integration testing: Her tjekker man at specifikke klasser kommunikerer korret, f.eks. i en MVVM-struktur. Det er også test af integrationen af en database, eller en client-server connection

Unit testing: Man skriver kode som skal teste kode

* + Man kan teste helt ned til små dele, altså helt ned til enkelte metoder
  + Man tester ved at skrive et input i metoden, og man så også får det forventede output
  + Det er typisk mest spændende at bruge, på lidt mere kompliceret kode
  + Fordel: Når først man har skrevet sin unit testing kode, så kan man altid teste ens metoder, om de stadig virker, når man laver om i sine klasser – i stedet for at lave testene på ny

Describe the difference between black-box and white-box testing.

* Black-box (User POV): Vi har en “sort boks” som er vores software, og vi aner ikke hvad der er i boksen. Vi kan altså ikke teste metoderne heri, da vi ikke aner hvad der er deri, men vi kan teste funktionskravene udefra som brugere af softwaren.
  + Pointe: man giver softwaren et input, den giver et output, og så ser man om det giver det rigtige tilbage
* White-box(Progammer POV): Her kan vi se koden, og vi kan se hvor I koden, vi mener noget kan være kritisk, således vi kan lave vores test-cases mere specifikke
  + Man kan branch teste = man kan teste at ens outcome er true, man kan også teste om det samme outcome bliver false, ved de rigtige conditions. Man kan altså teste samme metode, og se om den giver de forventede svar i flere forskellige scenarier
* Unit test kan både være black- og white-box approach, men typisk er det white-box, da man kan teste ned i de små detaljer

Present your own example from an assignment, SEP2, or exercise

* Give examples of a class or methods you want to test and describe what and how to test.
  + Branch testing: man tjekker flere if-statements og loops //ikke uddyb dette
  + Man kan tjekker ”Borders” altså hvis en metode har en afgrænsning, kan man før, under og efter afgræsningen, og se om det virker korrekt alle steder
    - Man tester ikke nødvendigvis for ALLE muligheder. Har man en range på 1-1000, tjekker man ikke samtlige 1000
      * Boundary value analysis: handler om boarders – tester boundaries
        + Start/End
        + Lower/Upper
        + Maximum/minimum
        + Just inside/just outside values
        + Empty/non-empty
        + Non-full/full
      * Equivalence partitioning: det data man smider ind i softwaren bliver fordelt i små mindre portioner, som hvor hver partition er ligeså bred som den anden. Hvis man f.eks. tilføjer flere tal i et array, og tester om det virker, så i stedet for at teste samtlige muligheder, så tager man enkelte, og antager resten også fungerer – f.eks tilføj min. 1 pizza til en orde, og max 10, så i stedet for at teste alt mellem 1 og 10, så tester man f.eks. 3, 5 og 7
      * Fordel ved de 2: man minimere antallet af nødvendige tests, det er en god rettesnor for, hvilke teste der er vigtige, og de er gode til programmer som skal bruge mange og store tal
* Relate this to equivalence partitioning, boundary value analysis, ZOMB+E and branch testing.
  + ZOMB+E = zero, one, many, boundary, exceptions
    - 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
* Show the class or method you are testing (Gerne I forhold til MVVM)
* Present your test strategy
* Show the JUnit test methods and relate it to your test strategy and describe how we can be sure that the tested method is correctly implemented
  + Junit = AAA struktur = Arrange, Act, Assert = man skal helst holde sig til disse tre, og ikke gentage dem I én test, da testen så bliver for stor.
  + Man skal teste så lidt som muligt pr test metode
  + Vi vil ikke have dependencies mellem testene – testene kan køre i forskellig rækkefølge, så det er ikke godt hvis én test først kan udføres efter en anden, da det kan ske, at det ikke bliver muligt
  + Arrange: man opretter de ting (objekter) man skal bruge, og så skal de efter Assert slettes igen, ved at bruge @Before og @After

Describe the difference between black-box and white-box testing.

Present your own example from an assignment, SEP2, or exercise

* Give examples of a class or methods you want to test and describe what and how to test.
* Relate this to equivalence partitioning, boundary value analysis, ZOMB+E and branch testing.
* Show the class or method you are testing
* Present your test strategy
* Show the JUnit test methods and relate it to your test strategy and describe how we can be sure that the tested method is correctly implemented

Talepapir:

State pattern:

* Man kan risikere, at ens program kan være i forskellige ”stadier”. Det kan f.eks. være, at hvis vi har en telefon, så kan den være i lydløs tilstand, eller på lyd. Det er 2 forskellige stadier. Vi kender det også fra tråde, de kan være i sleep tilstand, wait tilstand, blocked tilstand e.l.
* Altså har vi et objekt, som opfører sig forskelligt, alt efter hvilken tilstand den er i.
* Man kan sætte stadiet inde i objektet, i form af if-statements. Så skal vi tjekke hvilket stadie den gerne vil opnå, hver gang den skal ændres, og skal programmet udvides med flere stadier, så udvider vi også antallet af if-statements.
* Det har vi ikke lyst til, det er noget bøvl, og det tager lang tid.
* I stedet kan man lave klasser for hver muligt stadie objektet kan være i, hvor selve objektet kun kender til ét samlet interface, som stadierne skal nedarve fra. Metoderne i dette interface bliver herved state-specific, og ændre altså opførsel, alt efter hvilket stadie, der er taget i brug.
* UML: vi har en context klasse, altså vores program, som har en instans af State klassen. Context skal instans af state-klassen bestemmer således også start stadiet. Når Context så bruger de her State-specific metoder, så er det det nuværende stadie der bestemmer hvad der skal ske. Den kan f.eks. vælge at skifte stadie, og dette skal så specificeres i den klasse.
* På den måde bliver det også let at udvide programmet, i stedet for konstant at udvide if-statements hvor kriterierne konstant skal tjekkes.
* Context kan sendes med som parameter i State klassen og ned til nedarvningerne, andre gange kan man have et variabel i nedarvningsklasserne af typen Context.
* \*\*Præsenter UML og kode ift. context og states\*\*

JUnit

* Pointen med testing er at tjekke at ens kode virker, både med de forventede outcomes af ens kode, men også at teste det, som man ikke forventer skal ske.
* Her har vi både lært om scenario testing, integration testing og unittesting, og hver af disse kategoriserer sig under black-box testing, white-box testing eller for den sags skyld begge.
* Black-box testing kan man sige er user point of view. Man kan forestille sig, at man har en sort box, og det er vores program. Men vi kan ikke se ind i den sorte box, så den eneste måde at tjekke programmet på, er at man giver softwaren et input, den giver et output, og så ser man om det giver det rigtige tilbage. Man kan altså IKKE se selve funktionaliteten i koden, og teste direkte på denne.
* Det kan man tilgengæld med white-box testing. Dette er mere en programmørs point of view, da man kan se koden, og selv registrere, hvor man mener noget er kritisk, og det så skal testes. Dette kan man bl.a. gøre ved at branch teste, hvor man tester alle mulige branches, som en kode kan opnå. Det er altså både at tjekke om et outcome både er rigtig og forkert, ved at teste med forskellige værdier, således man får dækket alle tænkelige scenarier.
* Unit testing kan både være black- og white box, men vi bruger det typisk til white-box, så man kan teste ned i de helt små detaljer ved forskellige metoder
* KODE: Vi har tidligere fået givet en interface kaldet QueueADT, hvortil flere forskellige klasser har nedarvet fra denne, og herved givet funktionalitet til hver metode.

Hertil har jeg brugt JUnit til at teste, og funktionaliteten ved hver metode, har været korrekt.

Inde i QueueTest, f.eks. ved den første test, er testene sat således op, at vi har arrange, hvor objektet bliver oprettet, og dette er gjort i toppen, således denne metodes kan bruges ned igennem ved alle teste. Så er der act, hvor vi operere på objektet, og i den første test indsætter vi en kapasitet på køen. Assert som man kan bruge til sidst, validere at metoden under visse betingelser opfører sig korrekt. Eksempelvis i linje 35, der tester vi at der bliver throwet en IllegalArgumentException, når man indsætter et null objekt i køen. Tester man den, siger den korrekt, og tjekker man koden i klassen man tester, står der også at denne exception skal throwes.

Ned i gennem testene er der taget højde for

* Når man tester en klasse som denne, så er det oplagt at lave nogen boundary value analyse, hvor man tjekker de boarders der er ved listen. Det kan både være start og slut på listen, om man kan tilføje mere end man må, om man kan tilføje mindre end man må, om listen er tom eller fyldt, eller det modsatte.
* Man kan også sagtens lave equicalence partitioning, hvis listen f.eks. skulle blive meget stor, og man gerne vil tilføje ting på flere pladser, så kan man dele listens størrelse op i mindre partitions, og teste dem hver i sær.
* Begge har en fordel ved at man minimere antallet af nødvendige tests, det er en god rettesnor for, hvilke teste der er vigtige, og de er gode til programmer som skal bruge mange og store tal
* Hertil har mit udgangspunkt for testingen været ZOMB+E, som dækker over dem begge.
* Jeg har tjekket hvad der sker, når man indsætter et null objekt (linje 35), jeg har testet det samme med at indsætte ét enkelt element i metoden efter.
* Boundaries bliver testet, både ved linje 25, hvor kapaciteten hedder -1. Den melder dog at testen bestod her, og det kan derfor være en idé, at ændre metoden, så den få begrænsningen ved 0.  
  Det bliver testet i linje 97 at køen er tom, og i metoden efter tjekkes det når den ikke er tom. Det kommer derfor også omkring punkterne for boundary value analysis
* Flere steder tjekkes det også, om der opstår de forventede exceptions, f.eks. i linje 50, her tjekkes det at der blive throwet en exception, når man dequeuer fra en tom kø, samt tjekkes det at den ikke thrower exception efter man queuer et objekt, og deqeueue igen