## Del 3: Spørsmål

Her skal dere besvare noen spørsmål. Dere vil bli vurdert på hvorvidt dere evner å

reflektere rundt egne valg, forståelsen dere viser for objektorientert programmering og

pensum i emnet. Dere skal besvare følgende spørsmål:

1.Hvilke deler av pensum i emnet dekkes i prosjektet, og på hvilken måte? (For

eksempel bruk av arv, interface, delegering osv.)

2.Dersom deler av pensum ikke er dekket i prosjektet deres, hvordan kunne dere brukt disse delene av pensum i appen?

* Delegering (kunne ha hatt movementpatternet i en brikkeklasse)
* Arv (vi valgte å ikke bruke vanlig arv)
* Abstrakte metoder (kunne definert return typer uten å ha felt)
* Observert-observatør

3.Hvordan forholder koden deres seg til Model-View-Controller-prinsippet? (Merk:

det er ikke nødvendig at koden er helt perfekt i forhold til Model-View-Controller

standarder for å få full uttelling på dette spørsmålet. Det er mulig (og bra) å

reflektere rundt svakheter i egen kode)

Mindre logikk ved å tegne basert på brettet slik det er representert i Game. Fremdeles er all logikken i forhold til det å tegne brikkene på riktig sted. Blitt kvitt rokade og en passent. Flytter kun det som trengs å endres, ikke hele brettet -> kjører fortere, men krever mere logikk.

4.Hvordan har dere gått frem når dere skulle teste appen deres, og hvorfor har

dere valgt de testene dere har? Har dere testet alle deler av koden? Hvis ikke,

hvordan har dere prioritert hvilke deler som testes og ikke? (Her er tanken at

dere skal reflektere rundt egen bruk av tester)

Alle metoder som er public, men på en måte som tester private metoder også. Alle metoder skal kunne kalles utenom GUI uten at det forårsaker ugyldig tilstand eller oppførsel. Siden det er sjakk, så er det umulig å kunne teste alt. Datastrukturene har kuka oss over. Mange metoder er avhengig av at andre metoder kjører riktig. Game er vanskelig å teste siden en må manuelt sette opp sjakkposisjoner -> må anta at enkelte metoder funker som de skal. Svakhet ved koden. Hashmapet til checklegalmoves er kjernen til alle andre metoder, så den må være korrekt, ellers blir all annen oppførsel feil. Svære metoder, så har trukket det ut til andre klasser. Ikke enkle metoder som gettere og settere. Har testet alle deler dekket av prosjektbeskrivelse. Tester ved startposisjonen og enkelte spesifikke posisjoner. Alt funker fra start og under spesifikke tilfeller.

5.Har dere møtt på noen utfordringer i løpet av prosjektet? Hva ville dere gjort

annerledes en annen gang?

Datastrukturer i henhold til innkapsling. Hvordan en kan sende komplekse strukturer mellom objekter. Skal ting valideres selv om det ikke kommer til å feile gjennom bruk av GUI, så vidt vi har skjønt, ja. Utfordringer med å løse sjakkmatt og patt. Kanskje valgt et mindre komplisert prosjekt, da hadde det vært enklere å kunne implementere objektorienterte prinsipper etter hvert som vi lærte de, siden det er mindre kode å holde styr på. Velge andre datastrukturer. Tenkte nøyere igjennom hvordan klasser burde interagere, og minimere passering av data. Delt brettet i 64 og hatt 1d-array kunne ført til enklere tester. Problemer med liveshare, ikke kunne kjørt kode. Brukte at det ikke var innkapsla som feature, brukte at det er referanser til den samme strukturen, så vi kunne sjekke og endre på brettet mellom objekter.

1.Hvilke deler av pensum i emnet dekkes i prosjektet, og på hvilken måte? (For

eksempel bruk av arv, interface, delegering osv.)

* Abstrakte klasser (Piece) -> gikk vekk fra arv **x**
* Iterator (BoardTilesIterator) x
* Comparator (lambda uttrykk) x
* Lambda streams x
* Innkapsling (Deep copy) x
* Polymorfisme (Piece) x
* Abstraksjon (Movementpatterns) x
* Unntakshåndtering x
* Validering

**2.Dersom deler av pensum ikke er dekket i prosjektet deres, hvordan kunne dere brukt disse delene av pensum i appen?**

* Delegering (kunne ha hatt movementpatternet i en brikkeklasse)
* Arv (vi valgte å ikke bruke vanlig arv)
* Abstrakte metoder (kunne definert return typer uten å ha felt)
* Observert-observatør

Alle metoder som er public, men på en måte som tester private metoder også. Alle metoder skal kunne kalles utenom GUI uten at det forårsaker ugyldig tilstand eller oppførsel. Siden det er sjakk, så er det umulig å kunne teste alt. Datastrukturene har kuka oss over. Mange metoder er avhengig av at andre metoder kjører riktig. Game er vanskelig å teste siden en må manuelt sette opp sjakkposisjoner -> må anta at enkelte metoder funker som de skal. Svakhet ved koden. Hashmapet til checklegalmoves er kjernen til alle andre metoder, så den må være korrekt, ellers blir all annen oppførsel feil. Svære metoder, så har trukket det ut til andre klasser. Ikke enkle metoder som gettere og settere. Har testet alle deler dekket av prosjektbeskrivelse. Tester ved startposisjonen og enkelte spesifikke posisjoner. Alt funker fra start og under spesifikke tilfeller.

## Del 1 - Beskrivelse

Appen vår består av et sjakkspill med grafisk brukergrensesnitt med mulighet for å lagre et pågående parti, som en senere kan laste inn og fortsette på. Flytting av brikker skjer ved å klikke på en brikke og klikke på feltet en vil flytte den til. Vi forhåndsberegner alle lovlige trekk, og de blir vist som sirkler på brettet etter en har klikket på en brikke. Algoritmen vår tar hensyn til alle regler, og den avslutter spillet både ved sjakk matt og uavgjort ved patt. Ved bondeforvandling skriver en inn den brikken en vil erstatte bonden med i et tekstfelt.

## Del 2

I sjakk er det naturlig å uttrykke brikkene, brettet og rutene på brettet som objekter. Originalt hadde vi alle disse klassene, men vi valgte å droppe brettet på grunn av at vi støtte på problemer med innkapsling.

//**Abstrakte klasser**

Brikkene har en overordnet abstrakt klasse Piece som de individuelle brikkeklassene arver fra. Dette er et godt eksempel på hvordan polyformisme kan spare utviklingstid. Grunnen til at vi valgte å gjøre Piece til en abstrakt klasse i stedet for å bruke vanlig arv er fordi dette ikke er en egen brikke som eksisterer i spillet, og klassen skal aldri konstrueres som et eget objekt.

I tillegg til at subklassene arver metoder, så er en annen fordel med at en bruker ikke kan konstruere et Piece objekt at konstruktøren til Piece kan ha skjulte parametere som blir fylt ut med forhåndsbestemte verdier i konstruktøren til brikkeklassene. Piece klassen vår inneholder en ID som blir brukt i kontrolleren til å hente riktig bilde for å vise på brettet. IDen til en brikke kan dermed fylles inn anonymt i konstruktøren i brikkeklassene som sikrer at den alltid vil være korrekt, og vi kan dermed bruke en generell metode for alle Piece-klassene for å hente ut IDen.

**//Iterator**

I sjakk må en ha en måte å representere rutene for å kunne gjøre endringer som å flytte brikker fra og til dem og hente ut data. Vi endte opp med å bruke kartesiske koordinater i form av en int array. Vi lagde brettet vårt ved å lage en todimensjonal array av Tile-objekter, og kan dermed bruke koordinatene til å hente ut Tile-objektet vi ønsker å endre på.

I grafikken representerte vi brettet med et GridPane med ImageViews i hver celle. For å kunne flytte på bildene til brikkene, måtte vi ha en måte å korrespondere et objekt i Tile-arrayet til ImageViewet på den tilsvarende posisjonen i GridPanet. Løsningen vår er at IDen til et ImageView er koordinatene på GridPanet. Siden en JavaFX ID er en streng, gjorde vi at Game-klassen (som inneholder Tile-arrayet) implementerer Iterable, og lagde en iterator som returnerer koordinatene som en streng på samme form som FX IDene. Vi bruker så denne til å hente ut ImageViews i GridPanet met getChildren-metoden.

**//Comparator**

Siden vi forhåndsberegner de lovlige trekkene til alle brikkene, må vi ha en datastruktur for å representere dette. Vi valgte et HashMap, hvor nøkkelen er koordinaten til Tilen en brikke står på og verdien er en ArrayList med koordinatene til alle Tilene den kan flytte til.

I testingen måtte vi kunne sammenligne en stilling som er lagd på forhånd med en som blir generert av Game, og vi støtte da på problemer siden et HashMap ikke har innbyrdes ordning av elementene. Vi lagde derfor en Comparator i et lambdauttrykk for å sortere elementene etter koordinatene til nøklene.

//**Lambdastreams**

Det var få steder der det var aktuelt å bruke lambdastreams. I spill skjer endring av variabler sine verdier i stor grad gjennom kontrollflyten til programmet, som blir bestemt av algoritmer som er spesifikke til spillet sin funksjon. Det er dermed sjeldent at en får bruk for metodene lambdastreams tilbyr. Der vi fikk bruk for det var av filtrering av variabler som oppfylte visse krav.

**//Abstraksjon**

Det beste eksempelet vi har på abstraksjon i koden vår er en klasse som heter MovementPatterns. Denne klassen har logikken for å returnere koordinatene til Tilesene en brikke kan flytte til gitt Tilen den står på, posisjonen til resten av brikkene på brettet og bevegelsesmønsteret sitt. Den har en metode, moveHandler, som du gir den nevnte informasjonen til som argumenter. moveHandler kaller så på metoden som returnerer de mulige trekken (denne klassen tar ikke hensyn til at enkelte trekk kan sette kongen i sjakk, det blir beregnet av en annen klasse.) Brukeren som kaller på moveHandler trenger ikke å ta hensyn til hvordan den beregner de mulige trekkene til en brikke, de gir bare den nødvendige informasjonen og får tilbake resultatene.

**//Innkapsling**

Vi brukte mye tid på å sikre innkapsling. Dette var hovedsakelig fordi vi valgte datastrukturer for enkelte deler av programmet som ikke egnet seg for innkapsling hvis de skulle bli brukt av andre klasser. Det fremste eksempelet på dette er brettet vårt. Det består, som nevnt, av et todimensjonalt array av Tiles, altså Tile[][] I Java. Hver Tile har så enten en eller ingen brikker. På grunn av at alle interaksjoner med objekter blir gjort ved referanser i Java, må en forsikre at et objekt ikke endrer på et annet objekt sitt data direkte, men på en kopi. Dette lar seg lett gjøre på objekter som er immutable, eller samlinger av objekter som er immutable. Da kan en bruke en teknikk som heter «defensive copy.» Et eksempel på dette er:

return new ArrayList<int>(et felt eller variabel i objektet som er en ArrayList<int>);

Den ArrayListen objektet får i retur kan det endre på uten å gjøre endringer på det andre objektet sin ArrayList.

Problemer oppstår når en skal returnere objekter eller samlinger av objekter som er mutable, i vårt tilfelle Tile-arrayet med Piece objekter. Både Tile og Piece er mutable objekter, så selv om en returnerer en defensive copy, vil den bestå av referanser til de samme Tilene og Piecene fra det andre objektet, som bryter innkapsling.

Måten vi kom oss rundt dette på, er ved å bruke en teknikk som heter deep copying. Vi gjøre dette ved å bruke Serializable interfacet, som lar deg lage en kopi et objekt som vil eksistere et annet sted i minnet. Et objekt som mottar en deep copy kan dermed manipulere denne kopien uten å gjøre endringer på objektet den mottar kopien fra sine data.

**//Unntakshåndtering**

Unntakshåndtering har blitt brukt på to måter i sammenheng med filbehandling. Den ene bruken er at en må håndtere unntak når en bruker filbehandling i Java. Den andre er at vi har egne metoder for å validere strengen vi henter ut, og de utløser unntak som brukes i kontrolleren for å skrive informasjon til brukeren om at en feil har oppstått.

**//Validering**

En fordel med et spill som sjakk, som bruker et grafisk grensesnitt, er at det er umulig for brukeren å gi feil innputt. Brukeren kan kun klikke på brikker som har mulige trekk og kan kun utføre ett av de trekkene, og de trekkene vil alltid være innenfor brettet sine grenser. Så lenge den underliggende koden for å finne lovlige trekk og utføre flyttingen fungerer som den skal, vil spillet alltid kjøre som forventet, og en trenger i liten til ingen grad å implementere unntakshåndtering.

I henhold til spesifikasjonene som er oppgitt i oppgaven, skal en dog ta hensyn til at noen som utvider eller bruker koden ikke skal kunne gi feil input i public metoder. Vi har dermed innført validering av input i de metodene der det er aktuelt.

**2.Dersom deler av pensum ikke er dekket i prosjektet deres, hvordan kunne dere brukt disse delene av pensum i appen?**

**//Abstrakt metode**

I Piece klassen brukte vi ingen abstrakte metoder. Vi kunne ha brukt en abstrakt metode for som getter av brikken sin ID. Hver subklasse sin definisjon hadde hatt IDen sin som en konstant i return-uttrykket. Da hadde vi sluppet å ha IDen som et felt.

**//Arv**

Som nevnt i avsnittet om abstrakte klasser, så gav det mer mening å bruke det i stedet for vanlig arv, siden et Piece objekt ikke skal konstrueres. Brikkene er også det eneste naturlige stedet å bruke arv.

**//Delegering**

Et sted vi kunne ha brukt delegering er i Piece. Vi måtte i så fall ha brukt vanlig arv i stedet for en abstrakt klasse. I stedet for å ha bevegelsmønsteret til brikkene samlet i sin egen klasse, kunne de ha vært inni sin respektive Piece subklasse. Piece klassen har et felt som tar inn et objekt av en subklasse og en metode for å hente mulige trekk for en brikke. Denne metoden kaller på den samme metoden for subklasse-objektet den har som felt.

**//Observert-observatør**

Et sted det kunne ha egnet seg å bruke observert-observatør-teknikken, er for å utføre metodekall i samband med endringer i spilltilstanden.

**3.Hvordan forholder koden deres seg til Model-View-Controller-prinsippet? (Merk: det er ikke nødvendig at koden er helt perfekt i forhold til Model-View-Controller standarder for å få full uttelling på dette spørsmålet. Det er mulig (og bra) å reflektere rundt** **svakheter i egen kode)**

Koden vår forholder seg i all hovedsak i henhold til MVC-prinsippet. All logikken som ligger i kontrolleren er for å hente innputt og oppdatere grafikk og tekst. Innputt kommer i formen av klikk på brettet og tekst.

Klikk på brettet henter koordinaten til ruten, som så blir sendt videre til Game. Game gir tilbake informasjon om brikken har lovlige trekk, som blir brukt til å tegne sirkler på de respektive rutene, og setter kontrolleren i en tilstand hvor den flytter grafikken til brikken og fjerner sirklene hvis brukeren klikker på en av de lovlige rutene.

Hvis brukeren skal lagre et spill blir strengen med navnet på filen de har skrevet inn sendte til SaveBoardState som håndterer lagringen og gir tilbake informasjon om suksess eller feil som blir vist i grafikken. Ved innlastning gjør kontrolleren det samme, men oppdaterer grafikken basert på data den får fra Game når det objektet har oppdatert den interne tilstanden til spillet.

Kontrolleren får altså bare nødvendig informasjon for å kunne endre på grafikken, og har ingen logikk for å endre på spilltilstanden direkte.

En svakhet ved kontrolleren er at den oppdaterer kun grafikken på de brikkene som skal flyttes på i et gitt trekk. Det gjør at det er en del logikk for endring av grafikken for en passent og rokade, siden de trekkene flytter to brikker samtidig. Den alternative metoden er at for hvert trekk tegner den hele brettet på nytt etter at Game har oppdatert det interne brettet. Metoden med å kun endre på det som er nødvendig gir raskere kjøretid, men gir økt kompleksitet i kontrollerkoden, mens den andre metoden er saktere, men krever mindre kode.

**4.Hvordan har dere gått frem når dere skulle teste appen deres, og hvorfor har dere valgt de testene dere har? Har dere testet alle deler av koden? Hvis ikke, hvordan har dere prioritert hvilke deler som testes og ikke? (Her er tanken at dere skal reflektere rundt egen bruk av tester)**

Vi har ikke testet gettere og settere, og vi antar at IO-metoder til Java fungerer som de skal. Vi har testet alle metoder som er public, både for å forsikre om at de utfører oppgaven de gjør korrekt og at de utløser unntak ved feil innputt eller hvis de blir kalt i en ugyldig tilstand. En utfordring med testene, er at programmet er sjakk. På grunn av kompleksiteten til sjakk, er det umulig å teste alle tilfeller.

De datastrukturene vi har valgt egner seg ikke godt til testing. (litt usikker på hva spesifikt det er med de, så gjerne fyll ut her)

Vi har også mange metoder som nytter metoder i andre klasser. Det blir dermed vanskelig å teste metoder i vakuum siden de er avhengige av at metodene de kaller på fungerer slik som de skal. Et godt eksempel på dette er Game. Det er tungvint å sette opp sjakkposisjoner manuelt, så en er avhengig av at metodene som gjør det for en fungerer hvis en skal kunne skrive testene uten å bruke urimelige mengder med arbeid på dem.

**5.Har dere møtt på noen utfordringer i løpet av prosjektet? Hva ville dere gjort annerledes en annen gang?**

Datastrukturer i henhold til innkapsling. Hvordan en kan sende komplekse strukturer mellom objekter. Skal ting valideres selv om det ikke kommer til å feile gjennom bruk av GUI, så vidt vi har skjønt, ja. Utfordringer med å løse sjakkmatt og patt. Kanskje valgt et mindre komplisert prosjekt, da hadde det vært enklere å kunne implementere objektorienterte prinsipper etter hvert som vi lærte de, siden det er mindre kode å holde styr på. Velge andre datastrukturer. Tenkte nøyere igjennom hvordan klasser burde interagere, og minimere passering av data. Delt brettet i 64 og hatt 1d-array kunne ført til enklere tester. Problemer med liveshare, ikke kunne kjørt kode. Brukte at det ikke var innkapsla som feature, brukte at det er referanser til den samme strukturen, så vi kunne sjekke og endre på brettet mellom objekter.

En av utfordringene har vært kompleksiteten til sjakk i seg selv. På grunn av denne kompleksiteten har det blitt mye kode. Det har gjort det vanskelig å kunne endre på koden etter hvert som vi har lært pensum utover semesteret, siden det ville innebære å måtte endre på store deler av koden.

Å velge et mindre komplisert prosjekt hadde sannsynligvis gjort det enklere å forholde seg til og implementere de objektorienterte prinsippene og teknikkene. Spesielt med tanke på at vi måtte ha innført de fortløpende etter hvert som vi lærte de gjennom semesteret.

i visste heller ikke at det å sende rundt dype datastrukturer gjør at de ulike objektene gjør endringer på de samme dataene, noe som gjorde innkapsling til en utfordring. Det å endre på måten de ulike objektene sender informasjon mellom hverandre hadde vært noe vi ville ha gjort annerledes.

Vi hadde også problemer med VSCode sin liveshare funksjon. Personen som ikke hostet kunne ikke kjøre eller debugge koden og så heller ikke resultatetene når hosten kjører koden.