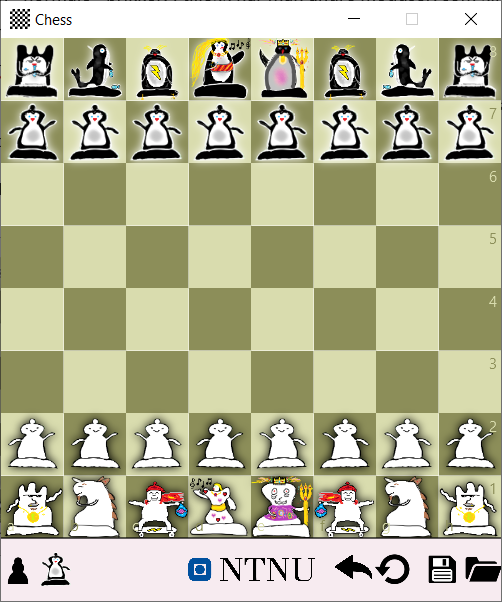
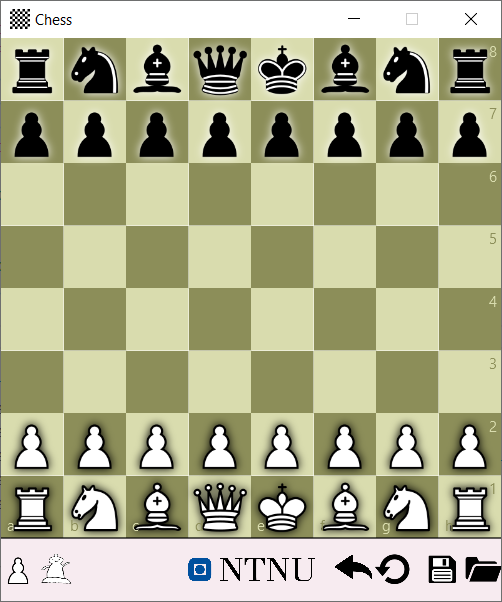
# Del 1.

Jeg har valgt å lage sjakk. Programmet følger alle sjakkregler, dvs. standard bevegelse på brikker, rokkering, oppgradering av bønner, En-Passant, sjakk og sjakkmatt. Jeg har implementert drag-and-dropp funksjonalitet slik at bevegelser av brikker er brukervennlig. Brukeren kan lagre spillet og gjenoppta det senere. Jeg har også implementert en tilbakeknapp, slik at brukeren kan gå tilbake en tur. Bondeforvandling har et unikt brukergrensesnitt som lar brukeren velge hvilken brikke bonden skal oppgraderes til.

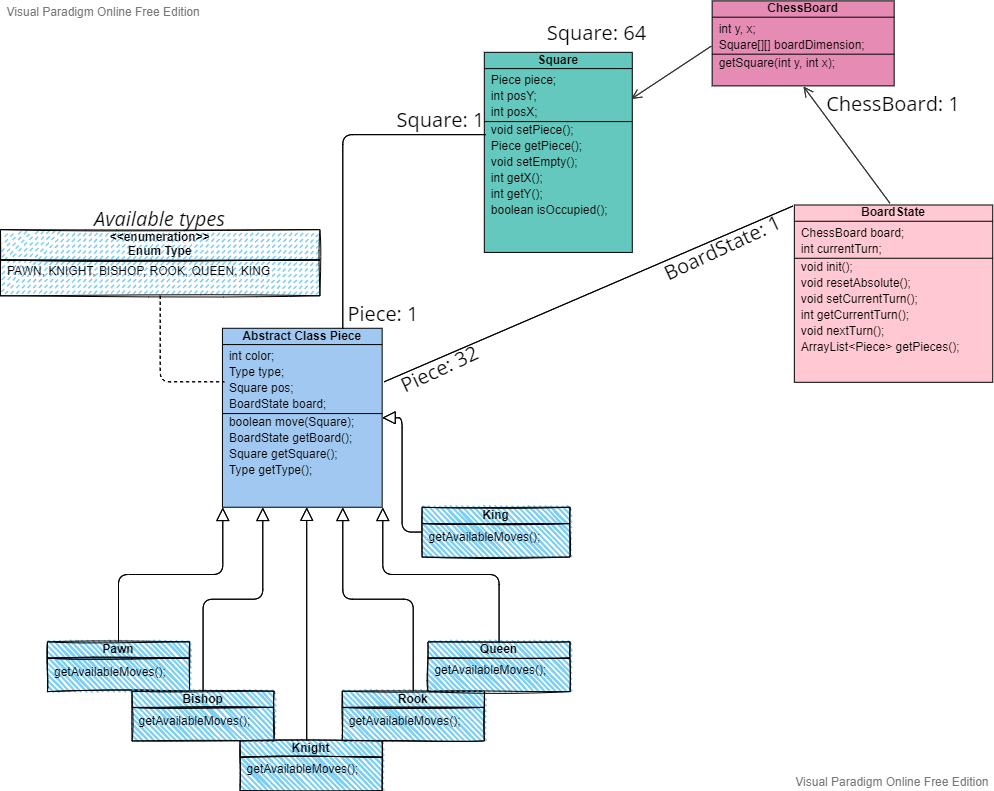
Jeg har også implementert forskjellige stiler å spille sjakk i. Programmet inneholder en standard stil (se figur 1), med «normale» brikker. I tillegg har man andre moduser, som gir sjakkbrikkene sine egne stiler (se figur 2). I disse stilene har hver brikke to versjoner av seg selv som de veksler mellom (two-frame-animasjon) på et tilfeldig intervall for å få dem til å se mer «levende» ut. Denne funksjonen er bare for gøy, og har ingen reell påvirkning på selve sjakk spillet, regelsettet eller brukergrensesnittet.

Ellers har vi en rekke andre mindre funksjoner som nytt-spill- funksjonalitet og meny, og en kul sluttskjerm.

Figur 2: Sjakk med egen stil på brikker

Figur 1: Standard sjakkoppsett

# Del 2.



Figur 2: Viser klassediagram av sjakkprogrammet i sin starttilstand

Klasediagrammet (figur 2) viser «back-end»-delen av koden til sjakkprogrammet. Dette er en simplifisert representasjon av hvordan de forskjellige objektene påvirker og anerkjenner hverandre. Diagrammet antar at man nettopp har startet spillet, altså at det eksisterer 32 brikker på brettet. Pensum om klassediagram dekker ikke Enums (så vidt jeg vet), så var ikke sikker på hvordan jeg kunne implementere dette inn i diagrammet.

Grunnen for at jeg valgte klassediagram er at det kan gi en enkel «overview» av hvordan programmet henger sammen, uten at man blir overveldet med informasjon. Sjakkprogrammet er komplisert, og inneholder mange metoder og attributter. Skulle jeg brukt noen av de andre diagrammene hadde de blitt ekstremt store og krevende å lage.

Det er viktig å notere seg at jeg ikke har tatt med alle attributter og metoder, men heller de som er viktigst for å forstå hvordan programmet henger sammen.

Diagrammet ble laget på [*denne*](https://online.visual-paradigm.com/diagrams/solutions/free-class-diagram-tool/) nettsiden.

# Del 3.

## Spørsmål 1.

Sjakk er et stort program, så naturligvis dekker det også store deler av pensum. Det er mye delegering av arbeid, da det er mange klasser som sender og gir informasjon til andre klasser. Vi har f.eks en Square klasse, som blir brukt i ChessBoard, som igjen blir brukt i BoardState, som blir brukt av Piece, ChessController og mer.

Brikkene bruker (extends) en abstrakt klasse, Piece, og arv blir derfor brukt. Alle de forskjellige Piece-underklassene påvirker også hverandre. Basert på plassering, farge, om de har beveget på seg, hvor kongen befinner seg osv. så påvirker det hvor de forskjellige brikkene har lovt til å bevege seg. Det er derfor stor interaksjon mellom objekter. Innkapsling blir også brukt slik at metoder som bare blir brukt internt er private.

Enums blir brukt for å bestemme hvilken type brikke en Piece er. Lambdauttrykk blir brukt for drag-and-drop mechanics. Det blir også brukt private-classes for two-frame-animasjon. Er usikker på om dette er pensum eller ikke. I tillegg blir grunnleggende pensum som if-setninger, loops osv. brukt i sjakkprogrammet.

## Spørsmål 2.

Jeg finner ikke noe pensum som programmet mitt ikke bruker.

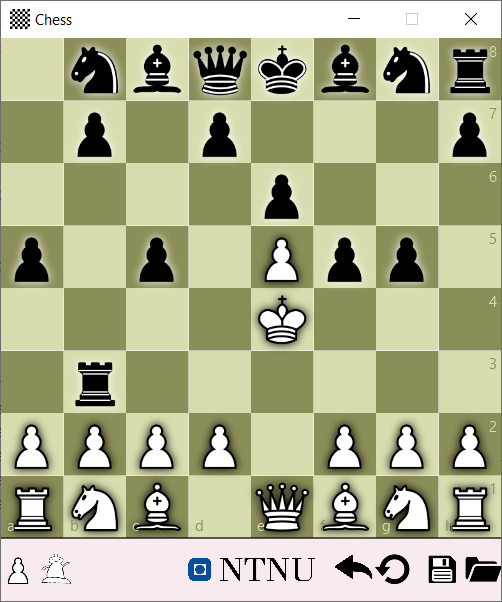
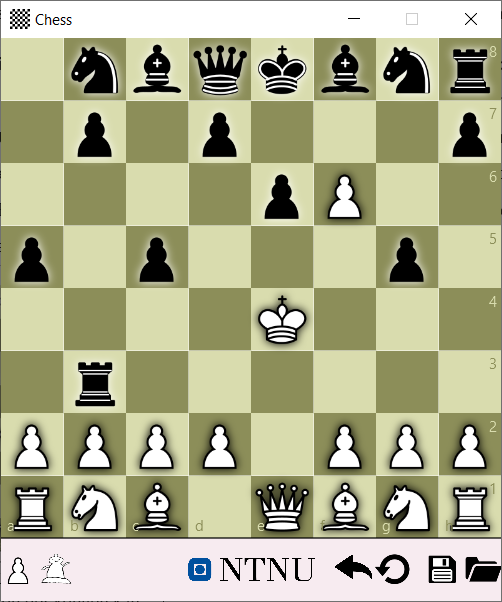
## Spørsmål 3.

Programmet klarer å følge Model-View-Controller-prinsippet nokså bra, men ikke perfekt. Piece-klassen er en miks mellom flere deler, da den inneholder logikk i hvordan en brikke kan bevege seg, men er også ansvarlig for den synlige drag-and-drop logikken. Dette er ikke vanskelig å fikse, da drag-and-dropp kan bli plassert inn i ChessController klassen. Men jeg syns det er mer intuitivt og lettere å lese koden hvis drag-and-drop mechanics for Piece faktisk befinner seg i Piece klassen, selv om det strengt tatt ikke følger Model-View-Controller-prinsippet. For meg er det viktigere at koden er lett å lese, enn at den følger alle prinsipper eller er så optimalisert som mulig.

## Spørsmål 4.

Det er vanskelig å teste om sjakk faktisk fungerer. Det er utrolig mange kombinasjoner man kan spille ut et sjakkspill, og jeg kan ikke vær sikker på at programmet mitt vil fungere 100% korrekt hver eneste gang. Dette var noe jeg var forberedt på når jeg begynte med prosjektet.

Testprosessen min har vært å få mange av vennene mine til å spille sjakk på programmet mitt, i tillegg til å gå gjennom merkelige og mer uvanlige situasjoner for å se om alt fungerer som planlagt. Det var noen ganger jeg støttet på mindre vanlige situasjoner der programmet ikke fungerte som det skulle. For eksempel var det en periode der sjakkprogrammet ikke kunne se at En-Passant var et valid trekk for å hindre sjakkmatt (se figur 4 og figur 5). Etter jeg oppdaget dette, fikset jeg opp i feilen.



Figur 4: Viser en posisjon der kongen kan unngå sjakkmatt bare ved bruk av En-Passant (bonden på f5 var på f7 turen før).

Figur 5: Viser hvordan man kan bruke En-Passant for å unnslippe sjakkmatt.

En funksjon som er vanskelig å teste er remis. Siden det er vanskelig å fort finne en stilling som gir remis, er jeg ikke sikker på om remis-logikken min dekker alle muligheter og har ikke klart å teste den skikkelig.

Ellers har jeg lagd en rekke JUnit tester som tester brikkenes funksjonalitet for bevegelse og sjakk-«detection». Det er vanskelig å lage tester ang. front-end (drag-and-drop, animasjoner, osv.), så alle testene ang. bare back-end kode.

## Spørsmål 5.

Å lage sjakk var utrolig lærerikt, og naturligvis var det flere utfordringer jeg møtte på (de fleste relatert til JavaFX). Det første problemet var å lære seg hvordan JavaFX fungerte. Selv om det har blitt dekket i pensum og forelesninger, var det mye som ikke hadde blitt nevnt eller snakt om. ImageView er et eksempel på dette. Ettersom alle brikkene mine er bilder, brukte jeg lang tid på å finne ut av hvordan jeg kunne få dem opp på skjermen og interagere med dem. Drag-and-drop funksjonen tok spesielt lang tid.

Jeg sleit også med å implementere Snap-To-Grid funksjonalitet, men fant videoer som [denne](https://www.youtube.com/watch?v=ME6WfnR6zys) som hjalp meg med det. Et annet problem jeg møtte på ang. JavaFX var hvordan man kan åpne en ny fane uten at man kan interagere med den forrige. Jeg endte opp med å finne ut av Modality og ownership av stages som hjalp meg med dette problemet. Jeg sleit også med å finne ut av hvordan jeg kunne sende informasjon til kontrollere. F.eks. når jeg lagde oppgraderingsmenyen sleit jeg med å finne ut av hvordan jeg kunne se hvilken bonde som skulle bli oppgradert. Jeg sleit også med å koble de korrekte brikkene til de korrekte ImageView-ene, og kunne lagre dette mtp. saving og loading.

Ellers sleit jeg med å implementere noen mer uvanlige sjakkregler som En-Passant og rokkering. En-Passant var spesielt vrient når det kom til saving og loading, og når det var nødvendig for å unngå sjakk/sjakkmatt (som nevnt tidligere).

Sjakk og sjakkmatt tok også en del tid å implementere. Det er mange forskjellige kombinasjoner og hendelser man må tenke på. Kan kongen bevege seg? Kan en annen brikke angripe brikken for å hindre sjakk? Kan en annen brikke blokkere brikken for å hindre sjakk? Hvis ja til noen av de to forrige, vil kongen fortsatt være truet fra en annen brikke?

En simplifisert forklaring av min metode for å løse dette, er å gå gjennom alle brikker og simulerte alle mulige trekk dem kan ta, for å så se om kongen fortsatt var i sjakk. Hvis den fant et trekk som ikke ville sette kongen i sjakk, betyr det at man ikke er i sjakkmatt. Det finnes kanskje en bedre måte å implementere dette på, men dette var den beste måten jeg kunne komme på.