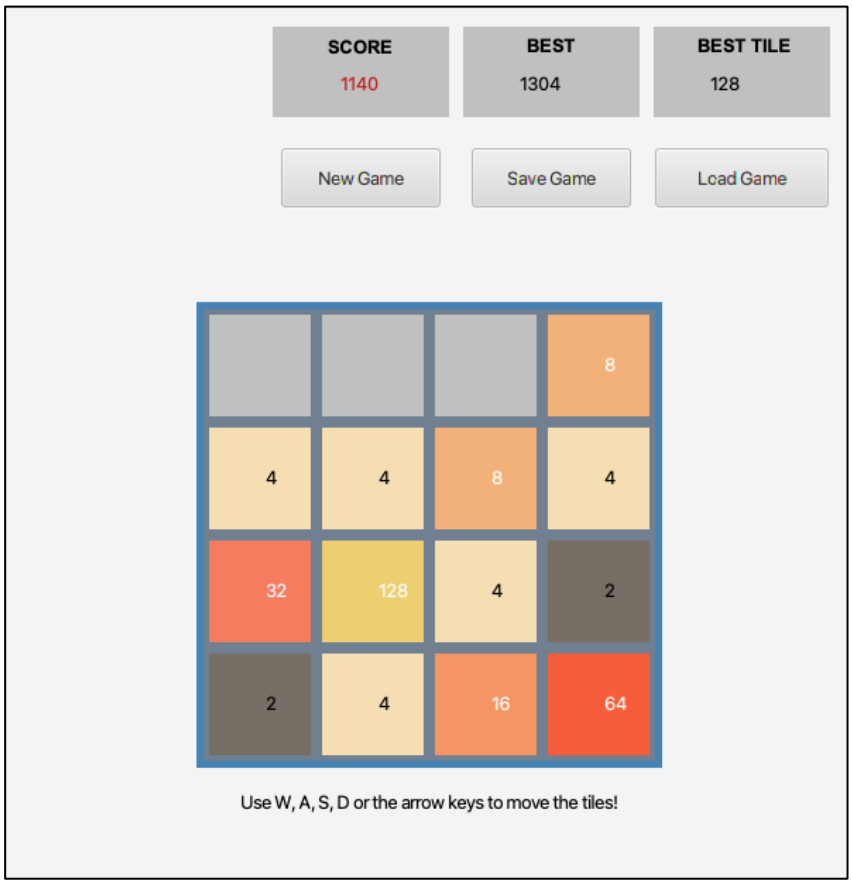
PROSJEKTOPPGAVE

TDT 4100 – Objektorientert programmering

NTNU

En modell av puslespillet 2048

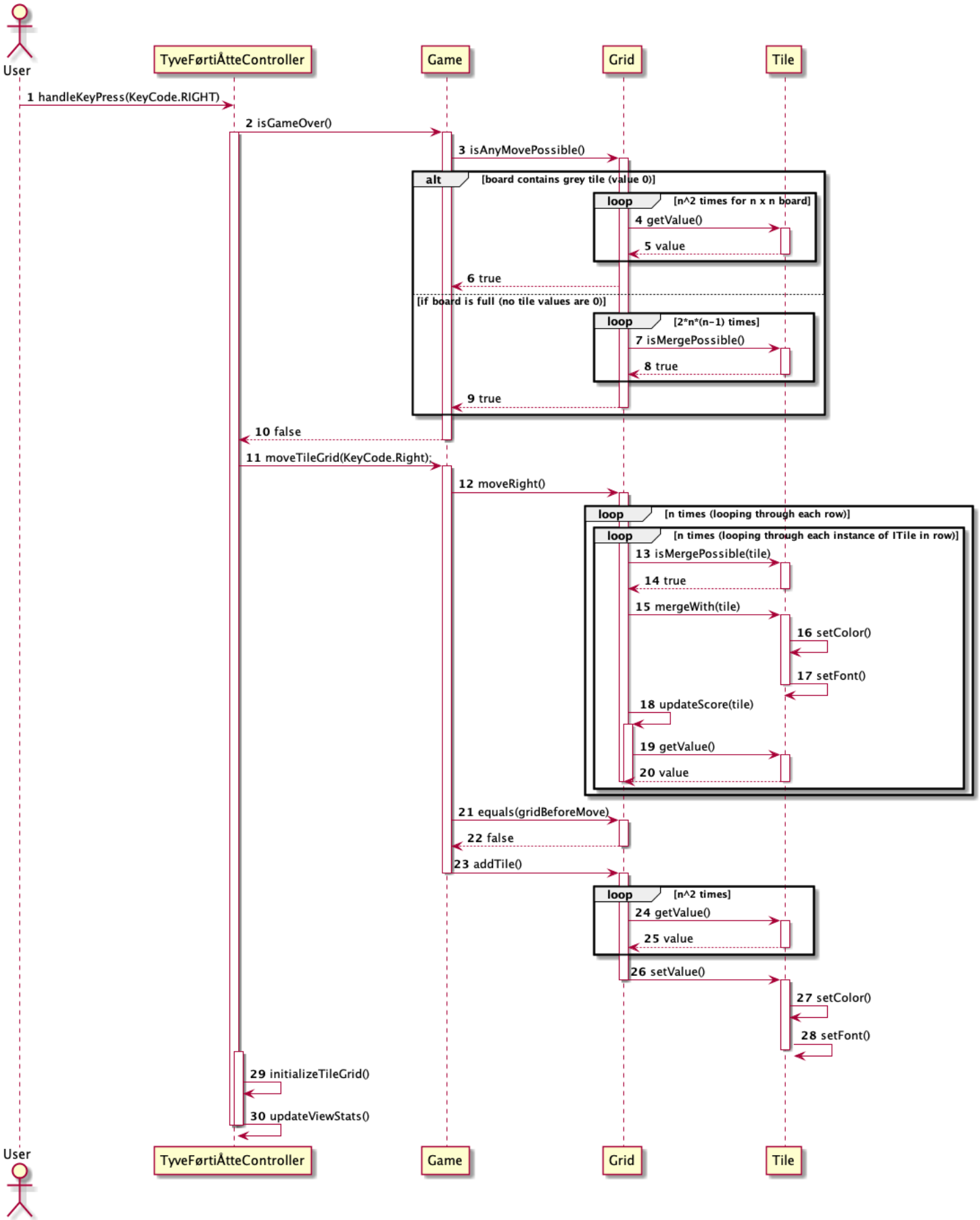


V2022

Del 1: Kort beskrivelse av appen og hva den gjør

2048 er et rutenettbasert puslespill, hvor hver rute enten representerer en toerpotens eller er grå (uten tall). Målet med spillet er å slå sammen tilfeldig genererte ruter med samme tall for å skape nye ruter med større tall, slik at en til slutt oppnår 2048-ruten (2^11). Da har en vunnet. Versjonen jeg har laget er inspirert av originalen, gitt ut av Gabriele Cirulli i 2014. Rutenettet er kvadratisk med 16 ruter totalt. En starter med to tilfeldig genererte ruter. Tilfeldig genererte ruter dukker opp på tilfeldige grå ruter uten tall og har verdien 2 eller 4 (0.5/0.5 sannsynlighet). For hvert trekk kan brukeren flytte brettet til høyre, venstre, opp eller ned. Da flyttes alle rutene i valgt retning og naboruter med samme tall som støter sammen skaper en ny rute med dobbelt så høy verdi. Eksempelvis vil sammenslåing av 4 og 4 skape en ny rute med verdi 8. En får poeng hver gang en slår sammen to ruter, der summen av tallene på rutene blir lagt til i scoren. En får ikke poeng for ruter som blir tilfeldig generert. Spillet er over når rutenettet har 16 tall-ruter, hvorav ingen av dem har mulighet til å slå seg sammen.

Del2: Diagram som viser en interessant/viktig del av appen



Siden 2048 er et rutenett basert spill med mye logikk som skjer i bakgrunnen, så er det mer hensiktsmessig å vise en interessant del av appen ved å demonstrere hva som skjer under kjøretid. For dette trengte jeg et dynamisk diagram, som er grunnlaget for valget mitt av et sekvensdiagram.

Sekvensdiagrammet en kan se ovenfor illustrerer oppførselen ved et spill som er allerede startet, der brukeren trykker piltast høyre. Antagelser på forhånd er at spillet ikke kan tapes eller vinnes ved to vilkårlige neste trekk og at det er tre eller flere ledige ruter. Dette er et typisk scenario som forekommer i stor grad av kjøretiden til programmet. Bildet på forsiden er et slikt scenario.

Del3:

1. Hvilke deler av pensum som dekkes og på hvilken måte

Koden min består av grunnklassene Tile (med tilhørende grensesnitt ITile), Grid og Game. En *tile* har en farge, font og verdi. Tileklassen implementerer ITile grensesnittet som har metodene som er relevante for resten av klassene. Tile inneholder altså data-grunnlaget som brukes når ruter vises i viewet, samt et snev av logikk for sammenslåing av ruter. Tanken bak ITile grensesnittet er å gi mulighet til videre utvidelse eller endring av programmet, om en eksempelvis vil ha en annen farge på rutene, eller generere 3er potenser istedenfor 2er potenser for verdien. Klassene vil da kunne bruke de samme metodene som før gjennom ITile, men appen vil oppføre seg annerledes avhengig av hvilken instans ITile er. Innmaten til ITile instanser kan med andre ord variere i stor grad, men vil ikke påvirke de andre klassene

Grid-klassen inneholder logikken i programmet. Her kalkuleres det bla. hvordan det genereres en tilfeldig rute på en tilfeldig plass ved hjelp av lambda streams og Predicate-interfacet. Metoden som flytter og slår sammen rutene til høyre, moveRight, sorterer array-et ved å sortere hver rad ved hjelp av at ITile arver fra Comparable, som i Tile klassen gjør at jeg kan overskrive compareTo metoden, slik at den sorterte listen passer med logikken. For moveLeft, moveDown og moveUp brukes små hjelpemetoder i tillegg til moveRight for å gjenbruke kode.

Videre har Grid klassen metoder for å sjekke om det er flere mulige trekk ved fullt rutenett, holder styr på scoren en får underveis som en gjør trekk og finner høyeste verdi til stede på brettet. Disse metodene inneholder en del logikk og knyttes direkte til array-et med ITile instanser. Metodene brukes derimot i Game klassen. Dette er et fint eksempel på bruk av delegering og kommer tydelig frem i sekvensdiagrammet eksempelvis i sekvens nr. 2-10. Med andre ord, så er det game som «styrer spillet» i modellen, men delegerer all logikk til Grid, som i enkelte tilfeller delegerer videre til ITile.

Validering brukes også hyppig, spesielt i konstruktørene til Tile og Grid. Da en må ta hensyn til ugyldige tallverdier og tile arrays som ikke er fullstendige. For dette har jeg blant annet en statisk metode i game klassen som inneholder en liste med gyldige tallverdier

2. Hvilke deler av pensum som ikke dekkes og hvordan disse kunne blitt brukt

Målet for prosjektet mitt har vært å skrive en implementasjon av 2048 spillet ved å skrive effektiv, robust og oversiktlig objektorientert kode. Dersom jeg skulle videreutviklet appen, ville jeg prøvd å implementere en abstrakt Tile klasse som tar i bruk arv. Jeg ville da presentert brukeren for flere valg når en starter spillet, som å velge å ha grønne, blå eller røde fargekontraster på brikkene. En kunne også valgt å generere tilfeldige 3er potenser istedenfor 2er potenser (kanskje både 2er og 3er potenser på et brett også hadde vært kult). For å realisere noe slikt, ville jeg trengt 6 forskjellige tile klasser. Her kan en naturligvis spare seg for unødvendig kode ved å ta i bruk en sentral objektorientert mekanisme, nemlig arv. Eksempelvis et abstrakt tile klasse som 2er og 3er potenser arver fra, som de ulike fargekontrastene igjen vil arve fra. Eksempelvis om en vil ha 3er potenser med blå fargekontraster. En ville da fått arv i tre ledd og spart mye kode. Arv ville blitt enda mer essensielt dersom vi hadde hatt flere fargekontraster. Det er vel gjerne derfor arv er såpass anerkjent i objektorientert kode. Ved å implementere arv legger en også en solid grunnstein for enda videre utvidelse av appen.

En annen tenkt implementasjon er å kunne ha negative 2er potenser, der man vil få to blanke ruter hvis man slår sammen -2 og 2. Da ville jeg bare måtte endret på addTile metoden til Grid klassen, med andre ord kunne jeg arvet fra Grid klassen, overskrevet kun én metode og

slippet å skrive en helt ny Grid klasse med identiske metoder utenom én (jeg måtte overskrevet noen metode i Tile klassen også).

3.Hvordan koden forholder seg til Model-View-Controller prinsippet

Viewet viser endringer i tilstanden til spillet, tar imot tekst input for å lagre tilstanden til et spill, og har en liste med filer som representerer lagrede spill. Kontroller klassen tar imot input fra viewet i form av KeyCodes, tekst input og Button events. Kontrolleren har feltene Game og ISaveHandler. Input som er relevant for tilstanden til spillet sender kontrolleren vider til Game. Videre henter kontrolleren henter ut alle tilstandsendringer som angår spillet fra Game klassen

Modellen består av SaveHandler, Tile, Grid og Game. Det er et 1-1 forhold mellom Grid og Game klassen, videre har grid koblinger til 16 ulike ITile instanser. Game har alle metodene som kontrolleren trenger, Grid inneholder logikken i appen og Tile inneholder data og noe logikk. Til sammen utgjør Game, Grid og Tile et flott samspill som håndterer tilstandsendringer i viewet. Det som gjenstår av modellen, er da ISaveHandler som håndterer henting og skriving fra og til fil.

4. Hvordan jeg har gått frem ved testing av appen og hvorfor jeg har valgt testene jeg har

Jeg startet med å teste Tile klassen. Her ligger fokuset i testene på om jeg har en validering i Tile som sikrer mot ugyldige tall og at logikken er riktig. Testene går relativt i dybden, noe som for meg var viktig, da fargen og fonten er bestemt av verdien.

Tilsvarende testes det at konstruktøren i Grid validerer input verdiene. Dette testes fordi at Grid baserer seg på at parameteren gridList skal ha en 2-dimensjonal array (4 x 4) der hver indeks har en ITile instans. Dersom ikke hver indeks har ITile instanser, eller at arrayen er på 4 x 4 form, så vil appen krasje. Videre testes det at scoren oppdateres med verdien til den resulterende tile etter sammenslåing av to tiles for at en skal være sikker på at viewet skal vise riktig informasjon underveis i spillet. Om moveRight ikke gjør riktige kalkulasjoner (slår ikke sammen tiles, flytter i feil retning) vil dette forvirre brukeren når rutene beveger seg i vilkårlige retninger når høyre var forventet.

De fleste metodene i Game testes implisitt når en tester Grid klassen grunnet delegering. Konstruktørene testes, slik at en vet at spillet en starter har riktig tilstand, en tester static metoden som inneholder de gyldige verdiene til rutene i brettet (toer potenser 0 - 2048). Siden disse verdiene brukes til validering både Grid og i Tile er det viktig at disse er riktige.

Angående henting og skriving fra og til fil så testes det at gyldige filer går fint å lastes inn, at ugyldige filer og filer med ugyldig data ikke vil laste inn og utløse unntak istedenfor. Det testes også at lagring av et spill skjer uten problemer.

5. Utfordringer i løpet av prosjektet

Jeg opplevde flere ganger at det jeg stod fast. Det er jeg glad for, fordi det har lært meg mye nytt i flere aspekter av koding. Det har vært en lærerik prosess. Neste gang ville jeg brukt post-it lapper og planlagt prosjektet bedre før jeg satte i gang.