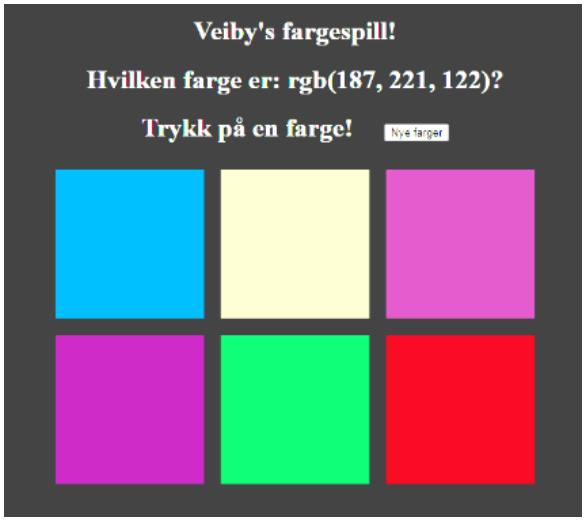
Dokumentasjon OOP-prosjektoppgave

Kandidatnummer: 10223

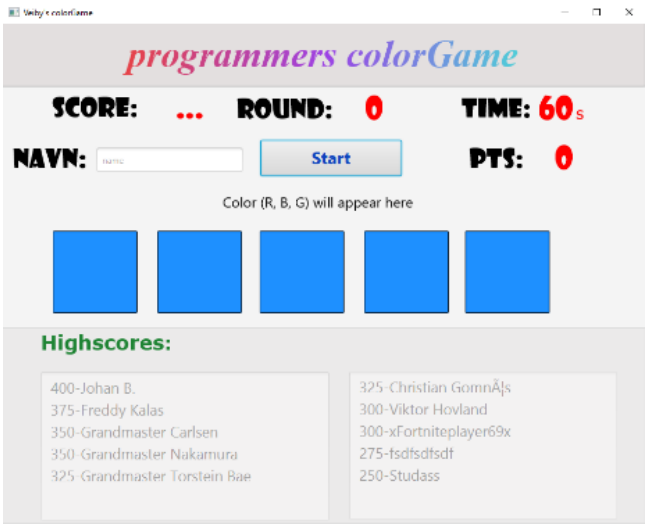
Del 1: Beskrivelse

Som prosjektoppgave har jeg laget et spill i det objekt-orienterte programmeringsspråket; Java. Spillet har jeg kalt for *programmers colorGame*, dette fordi spillet går ut på å gjette hvilken farge som samsvarer med den oppgitte RGB-koden. Spillet implementerer et felt for å skrive navn, og en startknapp. Det gis også informasjon om hvor mye tid man har igjen, hvor mange poeng man har, og hvilken runde man er på. Når man har fullført alle fem rundene, eller tiden har gått ut, vil man få oppgitt scoren. Denne avgjør om man klarer å komme seg inn på rekord-listen, som viser de ti beste scorene. Når runden er fullført, vil det også bli mulighet for å spille igjen.

Dette prosjektet har vært veldig spennende for meg. Dette fordi jeg lagde et spill i html, css og javascript da jeg gikk på ungdomsskolen. Appen jeg har laget kan ses på som en videreutvikling av det gamle front-end-spillet, dette til tross for at designet, logikken og kodespråket er fullstendig forandret. Jeg har kun hentet inspirasjon til hva jeg skal lage, og ikke hentet noe logikk fra den gamle appen. Som beviser legger jeg vet html, css og js filene til den gamle appen.



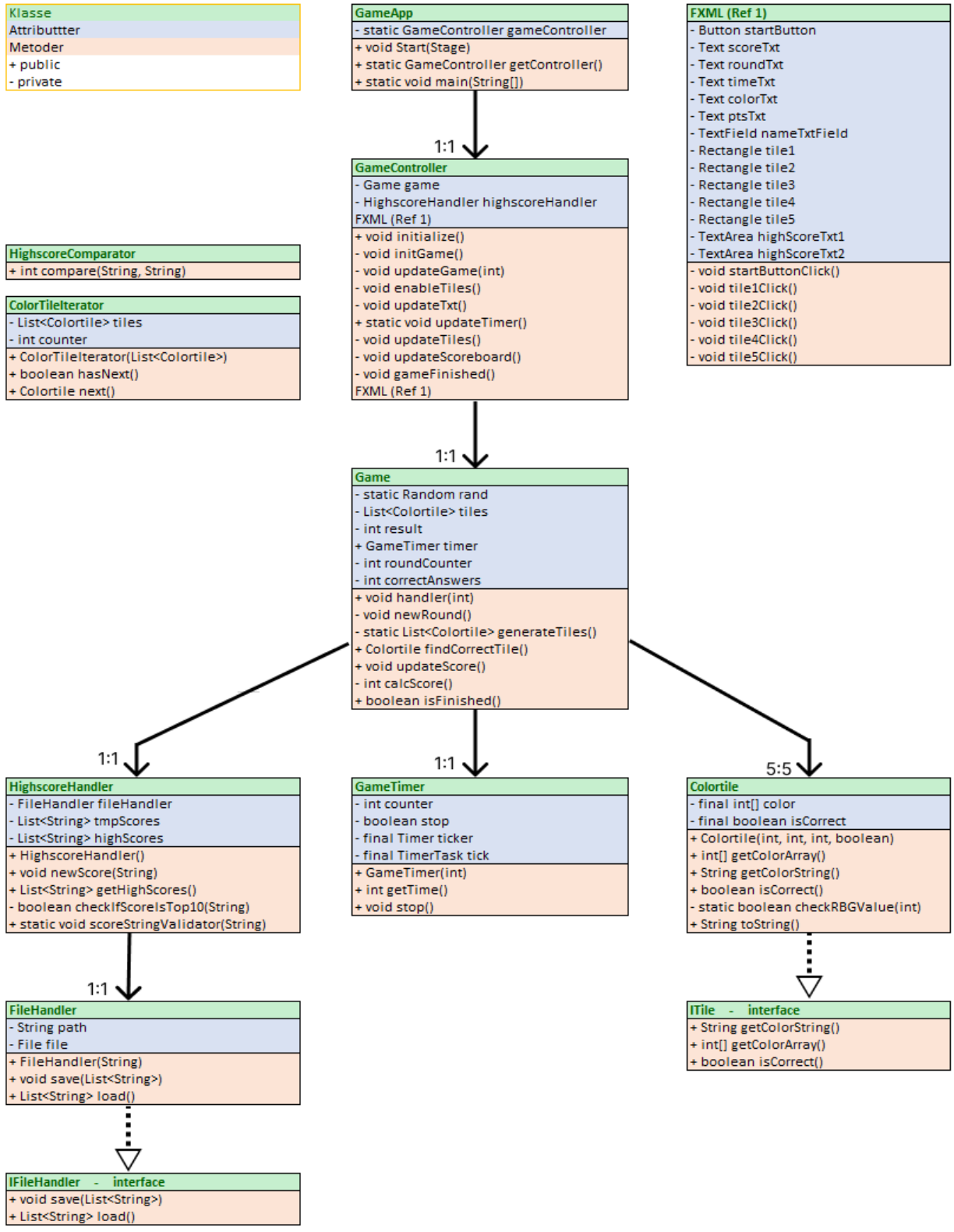
Figur 1 Gammel front-end applikasjon



Figur 2 Prosjektoppgave

Del 2: Diagram

Jeg har valgt å vise et klassediagram (figur 3). Dette fordi applikasjonen har en del klasser, og det er ikke intuitivt hvordan de ulike klassene henger sammen. Diagrammet fungerer både som en representasjon av klassestrukturen, og en pekepinn på hvordan applikasjonen fungerer.



Figur 3 Klassediagram prosjektoppgave

Del 3: Spørsmål

1: Hvilke deler av pensum er dekket?

Når jeg bestemte meg for hvordan jeg skulle lage oppgaven tok jeg utgangspunkt i prosjektbeskrivelsen, og den tematiske pensumoversikten (pensum) fra blackboard.

Figur 4 HighscoreHandler-klasse

Det første og andre punktet i pensum angår klasser, instanser, metoder, konstruktører, «this», innkapsling, interaksjon mellom objekter og standardmetoder ved innkapsling. Alt dette er brukt hyppig igjennom hele applikasjonen:

Klassen «HighscoreHandler.java» (figur 4) inneholder alle punktene nevnt over. Dette er en klasse, og når den skal brukes så må det lages en instans av den. Den har ulike metoder, som for eksempel konstruktøren. I konstruktøren lager den en ny instans av en annen klasse, kalt «FileHandler». For å kunne peke til denne instansens av HighscoreHandlers opprettede instans av FileHandler-klassen vil man bruke nøkkelordet «this».

Vi ser også at alle metoder og attributter er innkapslet ved bruk av enten «private» eller «public». Forskjellen er at ved bruk av «private», kan kun instansen av klassen kalle attributten/metoden. Ved «public» kan attributten/metoden bli kalt utenfor klassen selv. Det er ofte vanlig å gjøre alle attributter «private», for så å lage «getters» og «setters» -metoder for å hente ut/endre verdiene. Klasser er naturligvis alltid «public», konstruktører er også for det meste «public», med noen spesialunntak. Det går også an å gjøre en attributt/metode «static». Dette betyr bare at attributtet/metoden ikke er påvirket av instansen av klassen. M.a.o den kan kalles uten at man har en instans av klassen. Et eksempel på dette ser vi på første linje etter «try»-statementet i figur 4.

Det neste punktet i pensum omhandler objektoppførsel og grensesnitt. Objektoppførsel kan tolkes som mye, men jeg antar det angår hvordan klasser oppfører seg som objekter. Alle klasser er egentlig underklasser av hovedklassen «objekt», man kunne skrevet «extends objekt» i alle klassene sine, uten at det hadde gjort noe forskjell. På denne måten oppfører alle klasser seg som underklasser av andre «super»-klasser. Men det finnes også grensesnitt, disse er ikke objekter men mere retningslinjer for hvordan en klasse skal se ut. Når jeg skulle bestemme meg for hvordan flisene i applikasjonen skulle se ut lagde jeg først et interface («ITile»), som jeg implementerte i «Colortile»-klassen. Klassen har dermed fått en rekke retningslinjer til hvordan den skal se ut. Dette kan brukes til å generalisere utseende til klasser.

Figur 5 GameApp-klassen

Et annet punkt som kommer lenger ned er bruk av arv og abstrakte klasser. Her vil jeg først si at det er ikke mulig å lage en applikasjon i java uten å bruke arv. Man arver alltid fra en klasse, om den klassen er «object» eller en annen. Men jeg har brukt arv et spesifikt sted, det er i klassen «GameApp» (figur 5). Her bruker jeg nøkkelordet «extends» for å vise at

min klasse arver fra «Application»-klassen. Dette gjør at jeg kan skrive «GameApp.launch(args)» uten å ha definert den i selve «GameApp»-klassen. Klassen arver alle metodene fra «Application»-klassen.

«Testing» er det neste punktet på pensum. Jeg valgte å bruke JUnit-tester for å prøve ut all viktig og komplisert oppførsel i applikasjonen. Hvordan og hva som ble testet vil bli utdypt i et senere avsnitt.

```
//Save
public void save(List<String> highScores) {

    //Write every hs as a line in txt document (file)
    try (PrintWriter writer = new PrintWriter(this.file);) {
        for (String string : highScores) {
            writer.println(string);
        }
        writer.flush();
    } catch (Exception e) {
        System.err.println(e);
    }

}
```

Figur 6 Unntakshåndtering (try-cathch)

Det neste punktet på pensum er unntakshåndtering. Dette ble brukt når jeg skulle lese/skrive fra og til fil (figur 6). Det er mye som kan gå feil, og det er lurt å bruke (try-catch) eller (try-with-resources) når man jobber med fil. Dette for å passe på at filen alltid blir lukket etter ferdig, og for å håndtere «exeptions» som kan oppstå underveis. Det ble også brukt unntakshåndtering andre steder, hvor jeg brukte «throws» (figur 7).

```
//  Validate input-score-format
public static void scoreStringValidator(String elem) throws NumberFormatException {
    Integer.parseInt(elem.split( regex: "-")[0]);
}
```

Figur 7 Unntakshåndtering throws

Under seksjonen «Java-teknikker» i pensum, finner vi en del punkter som ikke er implementert i applikasjonen min, dette fordi ikke alle teknikkene gir mening å implementere. Teknikkene vi finner er:

Synlighetsmodifikatorene. Dette er utdypt i et tidligere avsnitt.

Wrapper-klasser. Disse er blitt brukt til plant annet å gjør om int til String og omvendt.

Collection-rammeverket. List<String> blir brukt som format for highscore-listen.

Sortering. Det finnes en comparator-klasse for sortering av highscorer.

Exception-hierarkiet. Dette blir brukt ofte, for eksempel i figur 7, hvor jeg kaster unntaket videre.

IO. Reader/Writer blir brukt til filhåndtering (figur 6).

Lambda-streams. Brukes under JUnit-testing.

Den neste seksjonen heter «Objektorienterte teknikker» og har to punkter: delegering og observatør-observert. Delegering brukes for eksempel når man skal sjekke hvor mange scorer det er i highScore objektet. Da skriver jeg i koden «highScore.size()», og dette delegerer en oppgave til ArrayList-objektet. Observatør-Observert-teknikken ble brukt minst en gang i applikasjonen. Applikasjonen inneholder en timer, denne timeren tikker ned og vil sende et varsel til GameControlleren hver gang den tikker ned. På denne måten fungerer timeren som observert og GameControlleren som observatør.

De siste punktene fra pensum er sammenhengen mellom FXML, JavaFX-klasser og kontrollerklassen. Dette er implementert, og vil bli utdypt i et senere avsnitt.

2: Deler av pensum ikke dekket

Det pensum som ikke er blitt brukt i applikasjonen er funksjonelle grensesnitt. Grunnen til at jeg ikke har brukt det, er rett og slett at jeg mener jeg har hatt bedre løsninger alle steder funksjonelle grensesnitt kunne blitt brukt.

Et sted hvor et funksjonelt grensesnitt hadde vært en alternativ løsning er som en erstatning for «HighscoreComparator»-klassen. Her kunne det funksjonelle grensesnittet «comparable» blitt brukt i stedet.

3: Hvordan forholder koden seg til model-view-controller (MVC) prinsippet?

MVC-prinsippet er en måte å strukturere koden i applikasjoner. Her er utgangspunktet at «model» skal holde på data, «view» skal vise data og «controller» skal flytte data. Det er akkurat slik jeg har satt opp applikasjonen.

«Model»-delen min er alle klasser utenom «GameApp» og «GameController». Klassene inneholder logikken til spillet, og metoder for håndtering av input-data fra brukeren.

«View»-delen min er FXML-filen «Game.fxml». Her brukte jeg scenebuilder for å lage grensesnittet. Det er satt opp en del elementer som skal vise og interagere med brukeren.

«Controller»-delen min er bindeleddet mellom «View» og «Model» -delen. Klassen «GameController» er satt opp som «controller» for «fxml» filen. Dette gjør at jeg kan koble til alle elementer i «fxml»-filen. Klassen setter så de ulike «fxml»-elementene til verdier hentet fra «Model»-delen. Et eksempel på dette er når man initialiserer spillet. Da lager «controlleren» en ny instans av «Game»-klassen. For så å sette alle «FXML Rectangles» til sine respektive farger hentet fra «Game»-klassen.

4: Hvordan har applikasjonen blitt testet?

Applikasjonen har blitt testet med en rekke «JUnit» tester. Det er blitt laget en test for hver klasse som implementerer metoder som er relevant å teste. Det skal sies at ikke hver eneste metode er blitt testet implisitt. Ofte er metoder avhengig av andre metoder, og blir derfor testet samtidig.

Jeg valgte testene jeg skulle gjøre med intensjon å teste alt som kan gå galt. For eksempel ved testing av validator-metoden til «HighscoreHandler». Her gav jeg ugyldig input med forventning om et spesifikt unntak tilbake, i stedet for å sende inn en gyldig input.

Utgangspunktet mitt når jeg lagde testene var at om testen feilet, er det koden som er feil, ikke testen. Jeg avdekket en rekke «bugs» som resultat av dette. Det skal sies at ofte var koden korrekt, men testen oppførte seg ikke som forventet.

5: Utfordringer

Min største utfordring i løpet av prosjektet var å holde meg i tøylene. Jeg forestilte meg ting som databaser hvor rekorden ble lagret i skyen. Men visste at dette ville ta for lang tid, og måtte holde meg unna å bruke unødvendig mye tid på kompliserte implementasjoner.

Ellers var utviklingen av applikasjonen både lærerikt og spennende.