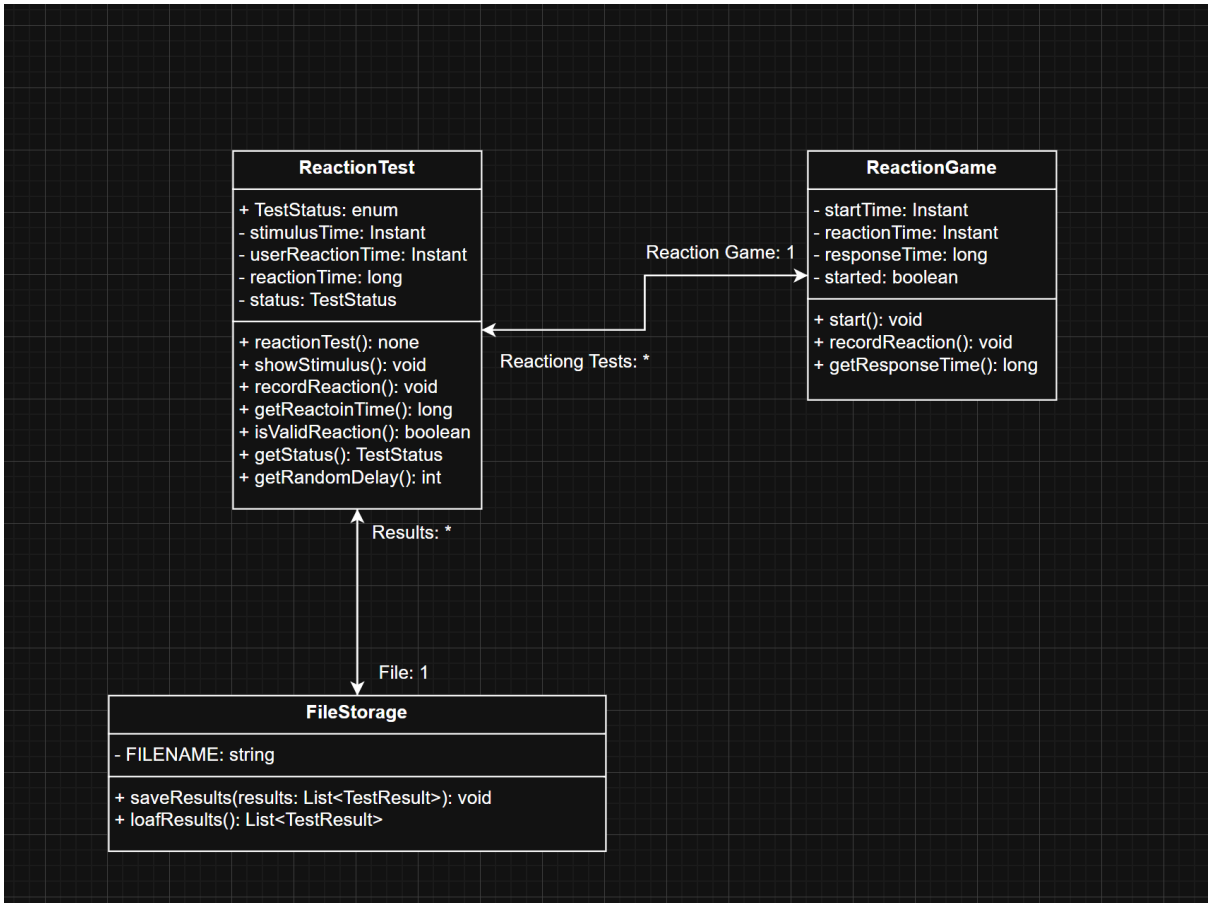
Del 5: Dokumentasjon

Beskrivelse av appen

Appen vi har valgt som sluttprosjektet er en digital reaksjonstest, noe lignende Human Benchmark. Hensikten med appen er å teste reaksjonstiden til brukeren. Fra brukerens perspektiv så vil de bli møtt med en nøytral skjerm der de starter testen og blir opplyst om hvordan testen foregår. Etterpå vil de bli bedt om å trykke på skjermen når den skifter farge. Derfra er det opp til brukeren å trykke på skjermen når bakgrunnsfargen skifter farge. Appen vil kalkulere tiden det tok fra når skjermen skiftet farge, til brukeren trykket på skjermen. Om brukeren trykker for tidlig eller dobbeltklikker for å prøve å jukse, så vil testen bli regnet som ugyldig og brukeren vil få beskjed.

Når brukeren har fullført testingen, så vil de bli informert om reaksjonstiden deres beregnet i sekunder. Informasjonen om vil bli lagret i en liste som blir vist på appen i en ledertavle. Appen vil kunne regne ut den raskeste tiden og gjennomsnittstiden til brukeren om de gjennomfører testen flere ganger.

Diagram



Figur 1: Klassediagram av hoved klassene i reaction game

Spørsmål

1. Reaction Game tar for seg en del av TDT4100 pensumet. Disse er delt inn i kategorier under den tematiske pensumoversikten.
   Under prosedyreorientert programmering bruker appen alt av uttrykk, datatyper og variabler. Eksempler fra koden vil være bruk av uttrykk i for-løkker og programsetning, forskjellige bruk av datatyper som varierer mellom primitive data typer som int, long og boolean, til referanser som String og ArrayList. I tillegg har koden kontrollstruktur, funksjonskall og programflyt som kommer naturlig av å bruke uttrykk i metoder og funksjoner.
   Aspektet av objektorientert programmering er reflektert i prosjektet i og med at den oppfyller kriteriene. Reaction Game har to klasser som skaper instanser, metoder og konstruktør, unntakshåndtering i tillegg til bruken av this. I figur 1 vises klassediagrammet av appen, her er objekt interaksjoner og innkapsling visualisert. Prosjektet tar også for seg testing av objektoppførsel ved bruk av JUnit.
   I pensumsdelen som dekker Java-teknikker, dekker prosjektet synlighetsmodifikatorene, bruk av wrapper-klasser, Collection-rammeverk og behandling av data med **Stream**-teknikken (ikke IO, men "objekt-strømmer"). Stream ble brukt under JUnit testene for å teste forskjellige intervaller.
   Til slutt, dekker prosjektet pensum kapitlet om JavaFX og FXML.

2. Deler av pensumet som ikke ble dekket i prosjektet var under kapittelet om objektorientert programmering som omhandlet objektoppførsel, grensesnitt og implementasjon av grensesnitt og arv. Under Java-teknikker ble det ikke brukt instanceof og casting, sortering med Comparable<T> og Comparator<T>, **Exception**-hierarkiet og checked vs. unchecked exceptions, IO med byte- og tegnstrømmer, funksjonelle grensesnitt og Java 8-syntaksen for "funksjoner", standard funksjonelle grensesnitt. Under Objektorienterte teknikker ble verken delegering eller observatør-observert brukt.
   Vi ser at det er en stor mangel på grensesnitt i prosjektet, noe som dekker er porsjon av pensumet. For eksempel, så kunne vi ha definert et GameInterface som spesifiserer felles oppførselen for spillet i ReactionGame klassen. Interface GameInterface hadde hatt funksjonene: void startGame(); void endGame(); void SaveResults();.
   I tillegg er arv også en del av pensumet, her kunne vi hatt en base Game klasse med funksjonaliteter som ReactionGame klassen kunne ha arvet fra. Til sist så kunne det vært en sorteringsfunksjon etter rasket reaksjonstid ved bruk av Comparable<T> eller Predicate <T> for å filtrere spillere.

3. Mappene er godt strukturert, der mappen model/ og fxui/ er separerte, fordi de inneholder forskjellige ting. Model/ inneholder logikk og datamodeller, mens fxui/ inneholder brukergrensesnitt og kontrollere. Innad Model-laget er det ReactionGame som håndterer kjernefunksjonene, mens FileStorage og ResultManager håndterer data behandling og persistens. I View-laget brukes JavaFX og FXML for UI. Dette er lagret separert fra spill logikken. Controller-laget har ReactionController som håndterer brukerinteraksjon. Dette laget er koblet til UI-elementer i og med at den har noen FXML-annotasjoner. Det er en del svakheter i koden vår ifølge Model-View-Controller-prinsippet. De største etter vår mening er blanding av ansvar i Controller-laget. I ReactionController håndterer den både UI-logikk, spill logikk og lagring. Dette bidrar til en mindre oversiktlig struktur og gjør den vanskeligere å lese. En løsning til dette er å fordele ansvaret i flere spesialiserte kontrollere.

4. Testene vi valgte er beregning av reaksjonstid (at tiden beregnes riktig), validering av reaksjoner (at for tidlige klikk registreres korrekt), statistikkberegning (at gjennomsnitt, median, og beste tid beregnes riktig), generering av tilfeldige intervaller (at de ligger innenfor definerte grenser), fillagring og -innlesing (at data lagres og leses korrekt), tilstandshåndtering i ReactionTest (at tilstandsoverganger skjer korrekt). Grunnen til hvorfor vi valgte disse testene var fordi vi tenkte at disse dekket hovedområdene til appens funksjonalitet. Det er kritisk at disse funksjonene må funke fordi det er grunnlaget for at reaksjonstesten funker. Vi tenker at vi har testet de største delene av koden, men så klart har vi ikke dekket alle. Noe vi savner når vi ser gjennom koden er tester for feilhåndteringer. Dette kan være testing for ugyldige resultater og filhåndteringsfeil. Grunnen til hvorfor disse ble nedprioritert var fordi disse feilene har mindre sjanse for å oppstå om f.eks, fillagring og innlesning og valideringstesten funker.