TFT Rolldown Simulator - TDT4100 prosjektoppgave

Tien Tran
Student
NTNU
Trondheim, Norge
tient@stud.ntnu.no

I. Beskrivelse av appen

Appen er en simulator av videospillet Teamfight Tactics (TFT). I spillet skal man bruke gull på å kjøpe xp-poeng, få nye enheter i butikken, og kjøpe enheter fra butikken.

I spillet finnes det enheter av forskjellige sjeldenhetsgrad, og de er gitt ved kostnaden til enheten. Kostnadene som finnes er 1, 2, 3, 4 og 5. Ved å øke nivået sitt øker man sannsynligheten for å finne sjeldnere eller dyrere enheter.

Man kan bruke gull på å generere en ny butikk og når man kjøper en enhet blir den lagt til i benken, med stjernenivå på 1. Kjøper man tre av den samme enheten med samme stjernenivå slår de seg sammen til en singulær sterkere versjon med stjernenivå økt med 1.

Dette er en prosess som vanligvis krever mye mental konsentrasjon ettersom i det faktiske spillet er man ofte tidsbegrenset, og man må klare å bruke så mye gull som mulig på kortest mulig tid. Ellers kan det hende at man ender opp med et svakere brett eller at motstanderene kjøper enhetene dine før deg.

Derfor har jeg utviklet en simulator hvor man kan øve på å rulle og kjøpe enheter

II. Diagram

Jeg har valgt å lage et sekvensdiagram som skal illustrere prossessen av å kjøpe xp-poeng i applikasjonen:

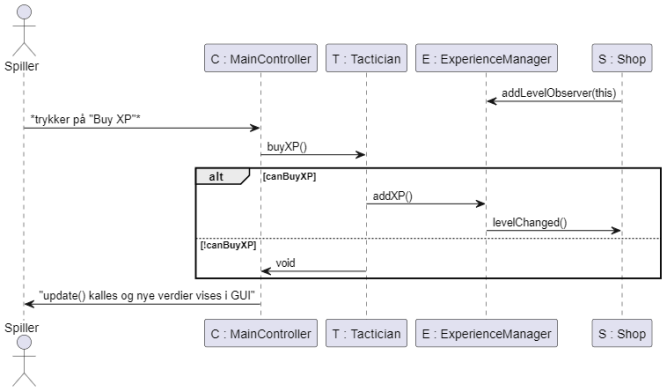

Fig. 1: Sekvensdiagram over kjøp av xp-poeng

Sekvensdiagrammet begynner med butikken som registrerer seg som lytter til xp-håndteringensklassen, slik at den skal bli varslet når nivået endres. Deretter vises et scenario hvor spilleren trykker på «Buy XP» knappen som finnes på det grafiske grensesnittet. Da kaller kontrolleren på metoden `buyXP()` på instansen av klassen `class Tactician`. Metoden inneholder logikk som sjekker om spilleren har nok gull til å gjennomføre kjøpet og kun hvis spilleren har nok gull fortsetter den å kalle på `addXP()` på instansen av klassen `class ExperienceManager`. Denne metoden legger til standardmengden xp-poeng, 4, og kaller på metoden `informObservers()` som itererer over alle registrerte observatører, i dette tilfellet en instans av klassen `class Shop implements LevelObserver`, og kaller på deres `levelChanged(int level)` metode med det riktige nye nivået.

På denne måten vil alle tall og verdier bli oppdatert når kontrolleren til slutt kaller metoden `update()` som henter verdier og setter labels til de riktige verdiene.

III. Spørsmål

A. *Hvilke deler av pensum i emnet dekkes i prosjektet, og på hvilken måte? (For eksempel bruk av arv, interface, delegering osv.)*

Jeg føler prosjektet dekker mye av ferdighetene man skal tilegne seg etter emnet. Dette i form av prosedyreorientert programmering, objektorientert programmering, Java-teknikker, objektorienterte teknikker og JavaFX.

I dette prosjektet har det blitt brukt:

- Arv med `class BoardUnit extends Unit`
- Grensesnitt og implementering av grensesnitt, for eksempel `interface Persistence` og `class Keybinds implements Persistence`
- Observerbar-observatør teknikken for å få `class Shop` til å abonnere gjennom `levelChanged()` til nivået som `class ExperienceManager` holder styr på
- Diverse synlighetsmodifikatorer som `private`, `public`, `static`, `final`
- Bruk av wrapperklasser, for eksempel gjennom de statiske metodene `Integer.valueOf()` eller `String.format()`
- Casting av JavaFX klasser, spesifikt for å hente elementer fra FXML-filen, spesielt ved traversering av node-treet
- Collection-rammeverket i stor grad for diverse datastrukturer, hovedsakelig `Map<K, V>` og `List<T>`

- Type-generics i grensesnittet `interface Persistence<T>`
- Unntakshåndtering ved hjelp av `try catch` blokker samt `throw new` for å avfyre unntak
- IO-filhåndtering, ved å lagre innstillinger slik at innstillingene bevares selv når applikasjonen er lukket
- Lambda-funksjoner, brukt i event handlere og ved funksjonell programmering med `.forEach()`
- Streams, brukt til å formattere en streng, spesifikt
```
Arrays.stream(this.units)
.map(Unit::toString)
.collect(
  Collectors.joining("\t|\t")
 );
```
- MVC-mønsteret, med separasjon mellom FXML-filer (View), modellklassene under mappen models og kontrolleren `class MainController`.

En viktig del har også vært å kunne implementere logikk som replikerer logikken som faktisk skal foregå. For eksempel skal det implementeres «tilfeldig trekk uten tilbakelegging» når en enhet dukker opp i butikken og kjøpes, men med tilbakelegging når enheten selges. Det å kunne implementere riktig uten logiske feil føler jeg også er en viktig ferdighet innen programmering generelt, selv om det ikke spesifikt står i pensum. Slike logiske feil kunne gått gjennom enhetstester siden de er mer «skjulte» og vanskelig å teste.

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

Delegering kunne blitt brukt hvis det for eksempel finnes spesielle enheter som oppfører seg annerledes, da det finnes metoder som kunne blitt overskrevet, for eksempel `levelUp()`

`instanceof` kunne da også blitt relevant siden man vil noen ganger måtte trenge å sjekke hvilken enhetstype spesifikt man håndterer.

Sortering med `Comparable<T>` eller `Comparator<T>` ville kunne blitt implementert hvis spillet videreutvikles til å for eksempel vise frem hvilke aktive traits man har (En trait har flere enheter, og hvis man kjøper flere ulike enheter av samme trait får man bonus effekter, ofte i form av styrke), og dette skal bli vist i sortert rekkefølge, for eksempel: 8 Scrap, 3 Ambusher, 2 Sentinel, …

C. 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. Det er mulig (og bra) å reflektere rundt svakheter i egen kode)

Koden forholder seg greit i henhold til Model-View-Controller-prinsippet, siden jeg har egne filer for stukturen til GUI-en gjennom FXML filer. Denne FXML filen er fullstendig adskilt fra modellene og har ikke noe med implementasjonen av programmet å gjøre. Jeg syns det ikke er riktig at modellene skal oppdatere GUI-en direkte, dette er noe kontrolleren burde gjøre, siden jeg anser kontrolleren som bindeleddet mellom modell og GUI. Så jeg har gjort det slik at når en knapp blir trykket på så kaller kontrolleren på metoder fra modellene, og det er også kontrolleren som sørger for at verdiene i GUI-en er riktig i henhold til verdiene som modellklassene holder på. Dette gjør jeg gjennom en `update()` metode.

D. 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)

For det første har jeg fulgt med mens jeg programmerte og implementert feilhåndtering «on the spot» når jeg skriver kode som muligens kan forårsake unntak. På denne måten terminerer ikke programmet dersom noe galt skulle skje, som gjorde det litt enklere å debugge.

Jeg har valgt å lage enhetstester for hovedsakelig klassene under mappen «core», siden disse klassene er selve kjernen av applikasjonen; Det er disse klassene som bygger spillet. Da har jeg valgt to sentrale klasser: `class ExperienceManagerTest` og `class TacticianTest`. Den første er ansvarlig for håndtering av xp, en sentral rolle i spillet. Den andre er ansvarlig for gullet og handlingene som spilleren kan gjøre, også veldig sentralt i spillet.

Jeg har ikke laget enhetstester for alle klasser, mest fordi at dette er et stort prosjekt og tidsrammen gjorde at prosjektet ikke kunne bli perfekt. Som sagt tidligere har jeg prioritert de viktigste klassene i applikasjonen først.

I tillegg lagde jeg enhetstest for fillagring, mest fordi det var et krav, men også fordi lagring er viktig og man vil sørge for at spillet ikke «gjør seg selv korrupt». Ved å bekrefte at lagringen og innlastingen av data fungerer som det skal, vil kvaliteten på brukeropplevelsen øke.