Prosjektoppgave i IDATA2001

Jørgen Finsveen

Utviklingen av WarGames og applikasjonen oppbygning

Eksamen 2022 – Programmering 2

NTNU Ålesund 21.05.2022

WarGames

NTNU
Kunnskap for en bedre verden

Rapport

Utviklingen av WarGames og applikasjonen oppbygning

Eksamen 2022 – Programmering 2

VERSJON
2.0

DATO
21.05.2022

FORFATTER
Jørgen Finsveen

ANTALL SIDER OG VEDLEGG
12 sider og 7 vedlegg

SAMMENDRAG

Utviklingen av WarGames

Denne rapporten omhandler utviklingen av applikasjonen WarGames. Rapporten beskriver hvordan applikasjonen har blitt utviklet fra konsept til et ferdig produkt. Målet med denne rapporten er å framlegge arbeidsprosessen, samt å reflektere over tilnærmingen, det ferdige produktet og veien til applikasjonen slik den er levert. Applikasjonen har til hensikt å la en bruker simulere et slag mellom egendefinerte armeer av forskjellige enheter, størrelser og i forskjellig terreng.

Historikk

VERSJON	DATO
1.0	15.05.2022

VERSJONSBESKRIVELSE
Førsteutkast, ingenting vedlagt, manglende diagrammer og uferdig refleksjonsavsnitt.

VERSJON	DATO
2.0	21.05.2022

VERSJONSBESKRIVELSE
Ferdigstilt rapport, klar for levering.

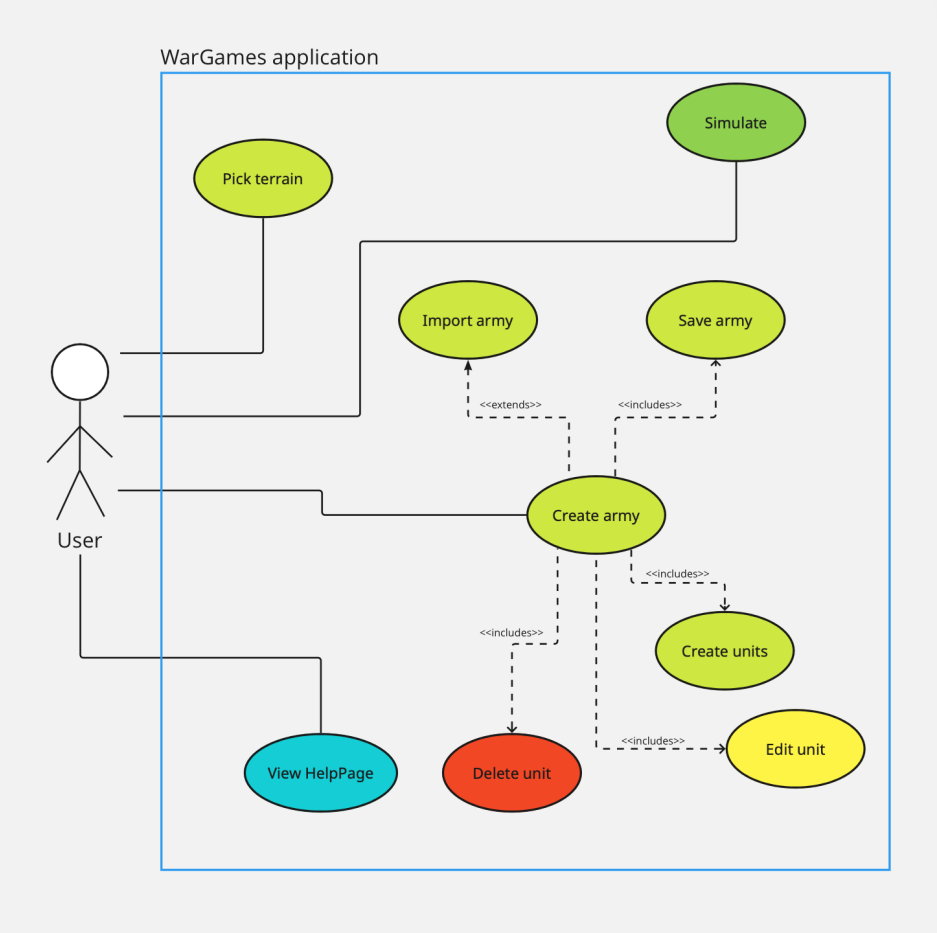
Innholdsfortegnelse

Innledning

Dette prosjektet er laget i anledningen mappeprøve i Programmering 2 i 2022. Prosjektet består i å lage en applikasjon kalt WarGames. Dette skulle være et spill som skulle gå ut på at en bruker kunne lage to armeer bestående av diverse kriger-enheter som hadde sine unike egenskaper og ferdigheter. Disse armeene skulle brukes til å simulere et slag, hvor brukeren skulle kunne se utfallet av slaget og få en vinner arme.

Kravspesifikasjon

Den ferdige applikasjonen tilbyr et stort utvalg med praktisk funksjonalitet for å gjøre simuleringen til brukeren både spennende og unik. Brukeren kan bygge opp hæren sin bestående av enheter av 7 forskjellige varianter. Hver enhet kan opprettes med enten standardverdier for beskyttelse og angrepsstyrke, eller brukerdefinerte. Brukeren kan velge å lagre armeen til fil for senere bruk, eller å hente opp en eksisterende en, for så å endre og slette enheter etter ønske. Videre kan brukeren velge blant 4 terreng som slaget skal stå ved, og så se hvert angrep som armeene gjør på hverandre, og hvilke arme som vinner.



Use-case diagram som viser hva brukeren kan gjøre.

Applikasjonen består av flere scener som tilbyr hvert sitt formål. En egen hjelpeside er med for å gi brukeren veiledning til hvordan programmet kan brukes.

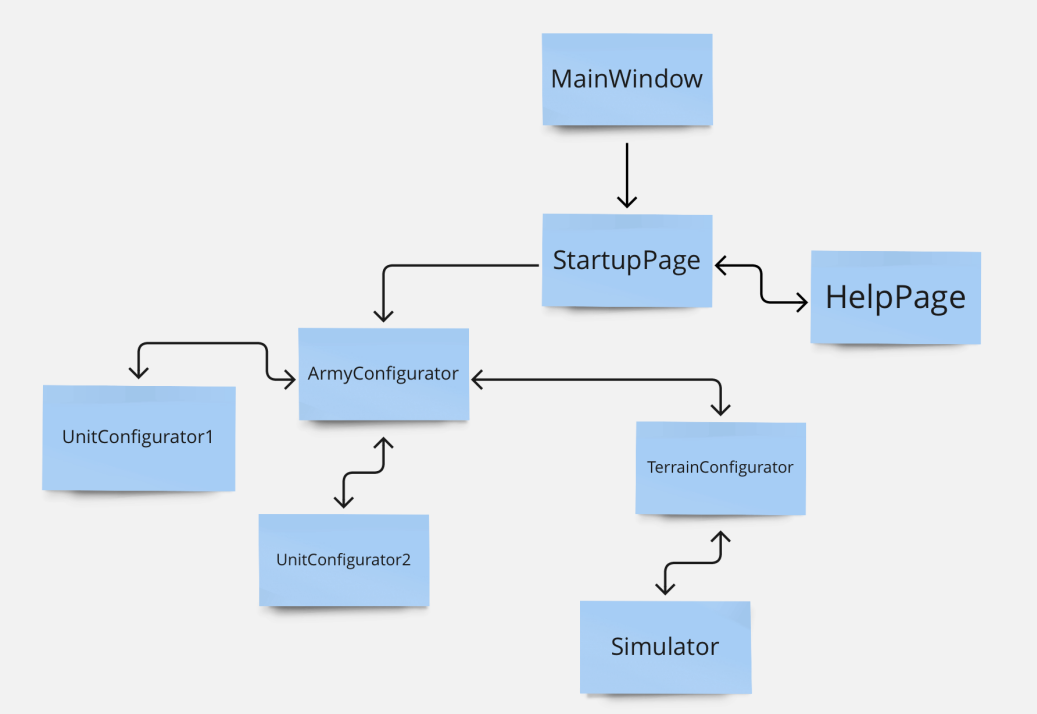
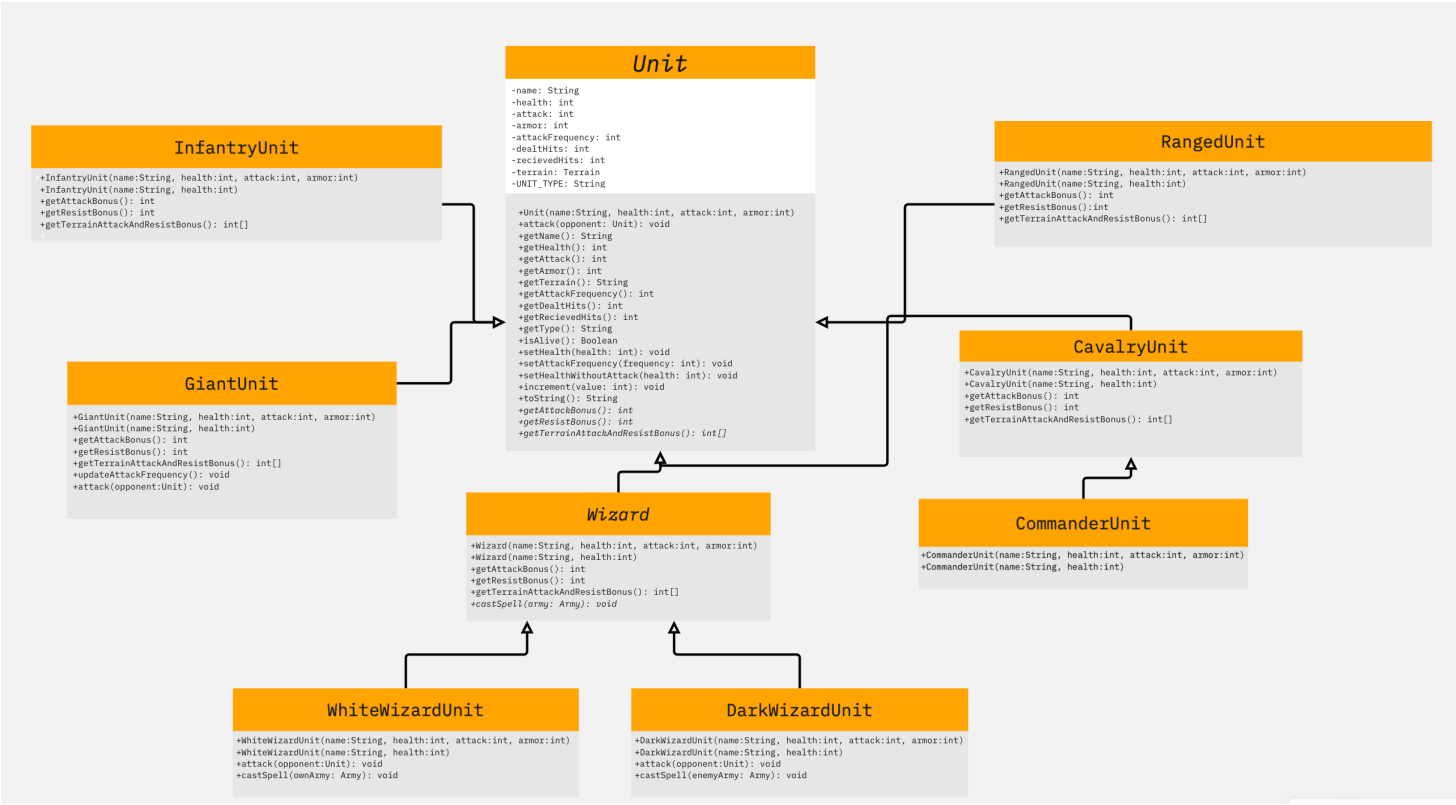


Diagram som viser hvordan brukeren kan manøvrere mellom de forskjellige scenene.

Design

Forretningsmodellen består av flere klasser som representerer forskjellige enheter, hvor alle er sub-klasser til superklassen Unit. Jeg valgte å lage flere enheter enn de definert i oppgavebeskrivelsen, og endte da opp med blant annet DarkWizardUnit og WhiteWizardUnit, som begge arver fra superklassen Wizard som igjen arver fra superklassen Unit.



Klassediagram for alle enhetene i WarGames. Abstrakte klasser Unit og Wizard samt abstrakte metoder står i kursiv.

Enhetene som blir laget kan samles i en instans av Army-klassen, og to armeer kan kjempe mot hverandre når de blir lagt inn i en instans av Battle-klassen.

Klassediagram som viser Army og Battle-klassen, og hvordan de henger sammen.

Det øvrige representerer forretningslogikken bak applikasjonen. Koden er kategorisert i mappene model, controllers, tools og other.

For kreasjon av units, har jeg brukt designmønsteret factory. Koden for dette finnes under mappen tools, i filen UnitFactory.java. I UnitFactory er det singleton-metoder for å skape en enkelt instans av en enhet, samt multiton-metoder for å skape flere instanser samtidig. I denne mappen ligger også designmønsteret Facade, som ikke er i bruk i applikasjonen per nå, men som kan være aktuell å bruke ved en framtidig utvidelse av applikasjonen. Videre i denne mappen ligger klassene Table og TableEntry. TableEntry representerer en «bestilling» av x antall enheter fra brukeren gjennom brukergrensesnittet. TableEntry-objektene vises som oppføringer i tabellen i armyConfigurationPage. Table-objekter holder på en samling av TableEntry-objekter, og representerer da hele tabellen. Enhetsbestillingene blir lagret i denne formen fram til de skal leveres som armeer i simulatorsiden.

Controllerne til de forskjellige scenene i GUI-et er å finne i mappen controllers. Når applikasjonen starter opp, blir start-metoden i MainWindow.java kalt. MainWindow står for opprettelse av alle de andre scenene med tilhørende controllere, og den distribuerer disse og andre nødvendige variabler rundt om til de andre scenene. Hver scene har en tilhørende controller, og controllerene holder styr på metoder knyttet til knappene til gjeldende scene, samt andre relevante metoder.

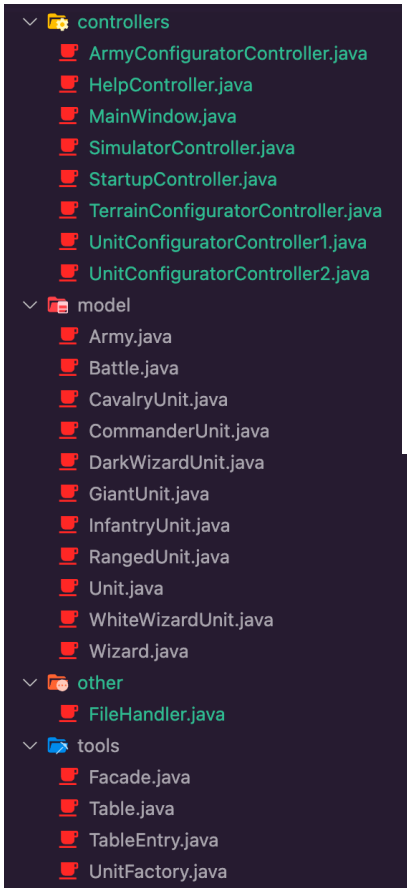
I mappen other finnes java-klassen FileHandler. Denne klassen brukes til å skrive til- og lese fra csv-filer. Disse filene inneholder ferdiglagde armeer. Alle metodene her er statiske, som er optimaliserer bruken av minne til Javas virtuelle maskin, og gjør at man ikke trenger å opprette en instans av klassen for å bruke den.

Implementasjon

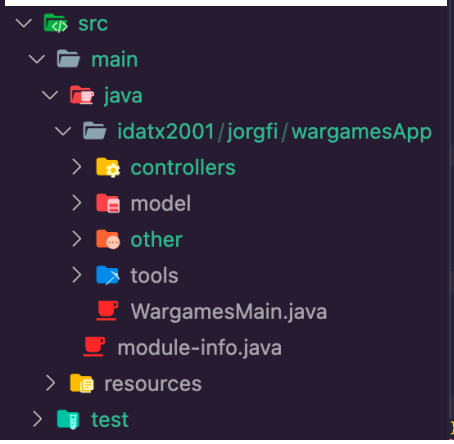
Koden for de forskjellige enhetene samt koden for Army og Battle var naturligvis det første jeg skrev. De er skrevet ganske rett fram slik som de vises i klasse-diagrammene vist ovenfor og etter spesifikasjonene fra oppgavebeskrivelsen. Testklasser har blitt skrevet parallelt med dette. Klassene GiantUnit, Wizard og tilhørende sub-klasser har blitt lagt til senere.

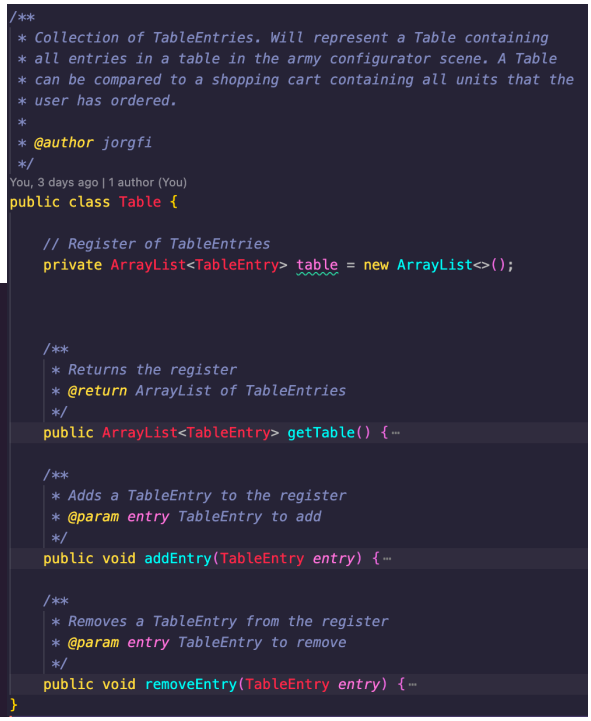
FileReader og Controller-klassene har blitt skrevet med utgangspunkt i eksempel-kode fra faglærer i tidligere forelesninger. Controller-klassene er private, og andre kontrollere kan kun motta dem etter å få tilgang tilhørende metoder fra MainWindow-klassen. Dette sørger for løs kobling.

Klassene som er å finne i tools-mappen ble skrevet etter hvert som jeg fikk bruk for dem. UnitFactory er skrevet etter spesifikasjoner fra oppgavebeskrivelsen, samt etter hvordan design mønsteret skal se ut. Klassene Table og TableEntry består aksessor- og mutator-metoder for informasjonen de holder på. Sistnevnte klasser skulle representere en bestilling av enheter, og jeg vurderte derfor en metode i Table som konverterer alle bestillinger til en arme, men jeg besluttet at det ikke var av responsibility driven design, og valgte derfor å ha en egen metode for dette i simulatorController.



Skjermdumper av pakkestrukturen i den ferdige applikasjonen, samt en komprimert versjon av koden i Table-klassen.





```
/**
 * Collection of TableEntries. Will represent a Table containing
 * all entries in a table in the army configurator scene. A Table
 * can be compared to a shopping cart containing all units that the
 * user has ordered.
 *
 * @author jorgfi
 */
You, 3 days ago | 1 author (You)
public class Table {

    // Register of TableEntries
    private ArrayList<TableEntry> table = new ArrayList<>();



    /**
     * Returns the register
     * @return ArrayList of TableEntries
     */
    public ArrayList<TableEntry> getTable() {…

    /**
     * Adds a TableEntry to the register
     * @param entry TableEntry to add
     */
    public void addEntry(TableEntry entry) {…

    /**
     * Removes a TableEntry from the register
     * @param entry TableEntry to remove
     */
    public void removeEntry(TableEntry entry) {…
}
```

Prosess

Frekvens
Jeg har hatt som prinsipp at for både prosjektet og lærdommens skyld, så skulle jeg programmere litt hver dag. Dette har blitt godt fulgt opp. Ellers har jeg hatt lengere økter på datalab i skoletiden, og i ferier.

Forberedelse
Før klassene har blitt satt sammen, så har jeg som regel prøvd å visualisere hvordan løsningen skulle være. Her kom UML-klassediagrammer og use-case diagrammer godt med. Etter å ha tegnet opp klassene og bestemt hva som trengs, så har jeg startet å skrive selve koden.

Versjonskontroll
Hele prosjektet har vært under lokal versjonskontroll. Det ble raskt knyttet opp til et eksternt repository i GitHub. Dette var da mitt repository i GitHub og ikke det utdelt av faglærer, da jeg hadde problemer med å koble meg til det i en periode. Da jeg omsider fikk koblet til GitHub Classroom repoet, så koblet jeg meg til det. Når det gjelder comitting, så har jeg tenkt at ofte er bedre enn sjelden, så jeg har comittet oftere, og da gjerne også rett til eksternt repository med en gang, da jeg har forsikret meg om at koden er robust. Jeg har ikke tatt i bruk forskjellige greiner når jeg har brukt versjonskontroll.

Issues
Når jeg har fått ideer for hva jeg kan utvide, eller dersom jeg merker meg et problem, så har jeg markert disse på to forskjellige måter. Den ene var å bruke // TODO: i selve koden, og den andre var å bruke issues-fanen på GitHub. Dessverre ble ikke issues overført fra det gamle repoet til det nye i GitHub Classroom, så de er ikke tilgjengelige. Issues har da blitt en form for prosess styring, hvor noen issues representerer milepæler. Når kravene for milepælene var oppfylte kunne jeg avslutte tilhørende issue.

Refleksjon

Prosjektet har vært svært lærerikt. Jeg har fått en helt ny grad av innsikt av applikasjonsutvikling, og jeg ser nå klarere enn noen gang hvilken stor fordel det er å ha god mappestruktur, dokumentasjon og god kodedesign.

Forretningslogikk
Jeg mener selv at forretningslogikken bak applikasjonen er både godt dokumentert og godt skrevet. Jeg merket betydningen av objektorientering og arv, da dette gir rom for mye abstraksjon og effektivisering av koden. Jeg dro dessuten nytte av funksjonell programmering gjennom lambda og streams, som egnet seg godt til ting som filtrering.

Kodedesign

Etter hvert som applikasjonen ble mer kompleks, og feil oppsto, så merket jeg også hvor viktig graden av innkapsling, cohesion, coupling og responsibility driven design hadde å si for programmets fleksibilitet og lesbarhet.

Øvrige klasser

Jeg har en egen mappe innerst i prosjektet som heter tools, hvor jeg har koden for et designpattern for masseprodusering av enheter kalt UnitFactory, samt noen klasser som representerer oppføring i en tabell over enheter laget av brukeren, Table og TableEntry. De sistnevnte var min egen ide, og jeg syntes de hadde en god hensikt. Jeg opplevde at å håndtere oppføringer av enheter i en tabell ble langt enklere med bruk av disse klassene, og at det dessuten etter min mening gjorde programmet mer oversiktlig. Å ha tabell-oppføringer lagret som instanser av TableEntry-klassen, gjorde det enklere å konvertere disse til faktiske enheter ved hjelp av UnitFactory-klassen først når det var strengt tatt nødvendig, altså i den delen av applikasjonen som sto for simulering. Etter min mening tilfredsstiller dette kravet for responsibility driven design.

Grafisk brukergrensesnitt

Controller oppsettet er jeg noenlunde fornøyd med. Jeg har inkludert en klasse, MainWindow, som funker som en fasade, hvor all opprettingen av kontrollere og scener skulle foregå. Det er hensikten at denne klassen skulle være en bro mellom GUI og forretningslogikk. Jeg mener derimot at selve klassen kunne vært designet litt bedre, og at de andre kontrollerne kunne hatt godt av ytterligere abstrahering. Her sikter jeg blant annet til scenen for enhetskonfigurasjon, UnitConfiguration. Det var en utfordring at det var behov for to versjoner av den scenen, som skulle legge til en enhet i hver sin arme. Jeg endte opp med å lage to versjoner som var nesten helt identiske, og dobbelt opp med metoder i ArmyConfigurator scenen for å samhandle med dem, addUnit1(), addUnit2() osv. som etter min mening ikke var den beste løsningen, og skulle jeg gjort noe annerledes så hadde jeg satt søkelys på å slå sammen de to scenene og heller finne en måte å lagre enhetene i riktig arme fra en enkelt scene. Alt i alt synes jeg klassene tjener sin hensikt, men at prinsippene for god kodestil kunne vært bedre tilfredstillt.

Testing

Når det gjelder testing av applikasjonen er jeg fornøyd. Jeg føler selv jeg har god testbredde med både positive og negative tester for hele forretningsmodellen. Testene var dessuten veldig praktiske for å kontrollere at programmet fungerte før brukergrensesnittet var implementert. Når det gjelder testing av brukergrensesnittet så mente jeg selv at det var langt enklere å teste det gjennom å kjøre applikasjonen, framfor å «simulere» en bruker som starter programmet. Brukergrensesnittets intuitivitet har dessuten blitt testet gjennom at jeg har latt venner kjøre programmet, for å se om det var oversiktlig og at alt fungerte slik det skulle.

Utvidelser

Jeg valgte å utvide programmet med noen ekstra enheter med litt mer unike egenskaper. Jeg sikter da til Wizard-enhetene og Giant-enheten. Wizard-enhetene skulle ha den funksjonen at de ikke skulle gjøre direkte skade på en annen enhet, men at de enten skulle kunne «forbanne» andre enheter slik at de gradvis mistet mer og mer helse (DarkWizard), eller at de skulle kunne gjenopplive en tapt enhet og la den få en ny sjanse i slaget (WhiteWizard). Giant-klassen skulle ha den fordelen at den tålte ekstra mye og gjorde mye større skade enn andre enheter, men den

skulle lide av at den var treig, og at den derfor kunne angripe en av tre ganger. Selv så mener jeg at disse enhetene ga programmet en helt ny dimensjon, og gjorde det morsommere å spille.

Problemer

Veien fram til en ferdig applikasjon var ikke problemfri. Et problem jeg støtte på var at jeg ofte fikk feilmeldingen java.lang.OutOfMemoryError, som betyr at programmet gikk tom for minne. Dette inntraff når jeg prøvde simulering gjennom brukergrensesnittet med mange enheter. Dette var ikke veldig enkelt å løse. De tiltakene jeg iverksatte var å prøve å øke minnetilgangen gjennom konfigurering av maven-kompilator pluginen. Dette er å finne på linje 30 i pom-filen, og videre utdypet i README.md filen. Videre satte jeg i gang med en refactoring der jeg prøvde å konvertere store deler av koden til statisk, deriblant FileReader og UnitFactory. Til slutt valgte jeg å sette opp en advarsel om dette i både README og i selve GUI-et. Problemet ble deriblant minsket noe, men det er fremdeles ikke fullstendig løst.

Konklusjon

Alt i alt har prosjektet vært både morsomt og lærerikt. Det har gitt på flere utfordringer som jeg håper og tror jeg har løst på en tilstrekkelig god måte. Jeg har dratt stor nytte av designmønstere, UML-diagrammer, testklasser og mye mer, og jeg har fått en bedre forståelse av applikasjonsutvikling og devops arbeid. Applikasjonen skulle ha som hensikt å la en bruker bygge opp armeer og utspille slag mellom disse, og jeg mener at min løsning dekker disse kravene og mer til. I fremtidige prosjekter vil erfaringene fra dette prosjektet være til god nytte. Se gjerne README-filen i applikasjonen der det står litt om hvordan man bruker applikasjonen, versjon, begrensninger og ressurser.

Litteratur

Ingen av oppførte kilder er direkte sitert til rapporten, men jeg vedlegger dem for å vise hva slags litteratur jeg har brukt til utforming av applikasjonen.

Barnes, D.J & Kölling, M. (2017) *Objects First with Java* Pearson, Edinburgh

Hegna, K. & Maus, A. (2017) *Javaprogrammering Kort og Godt* Universitetsforlaget, Oslo

Rumbaugh et al. (1991) *Object-Oriented Modeling and Design* Prentice-Hall International Editions, New Jersey

Seppälä, I. (2022). *Design patterns implemented in Java*. GitHub.
https://github.com/iluwatar/java-design-patterns

TutorialsPoint. (2022). *Design Pattern – Facade Pattern*. tutorialspoint.
https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

TutorialsPoint. (2022). *Design Pattern – Factory Pattern*. tutorialspoint.
https://www.tutorialspoint.com/design_pattern/factory_pattern.htm