

Kandidatnummer(e)/Navn:

Ramtin Forouzandehjoo Samavat & Tobias Skipevåg Oftedal

Dato:

25.02.2023

Fagkode:

IDATT
2001

Studium:

Bachelor i ingeniørfag, data, systemutvikling.

Ant sider:

20

Faglærere:

Majid Rouhani

Atle Olsø

Tittel :

Utviklingen av spillmotoren “Paths”.

Sammendrag:

Gjennom utviklingen av spillmotoren “Paths” gitt i mappevurderingen fra IDATT2001 ved NTNU, har det blitt utviklet en robust og velfungerende applikasjon. Gjennom kravspesifikasjonen til oppgaven stilles det krav til hvilke verktøy som brukes og hvordan enkelte aspekter av applikasjonen skal utvikles. Produktet består av en applikasjon som kan kjøre forskjellige lagrede spill. Prosessen har gjennom bruk av smidige metoder og bruk av viktige kode-prinsipper, ført til at alle krav til funksjonalitet gitt i kravspesifikasjonen er oppnådd. Teamet har også implementert ytterligere funksjoner, dette inkluderer blant annet: lagring av spillobjekter, tekst til tale og mulighet for å lage historier gjennom brukergrensesnittet. Gruppemedlemmene konkluderte med at prosjektet står til det forventede nivået, men at flere funksjoner for spillet hadde vært ønskelig.

Denne oppgaven er en besvarelse utført av studenter ved NTNU.

INNHold

1 Sammendrag	1
2 Begreper og forkortelser	1
3 Introduksjon	1
3.1 Bakgrunn	1
3.2 Avgrensninger	2
3.3 Begreper/Ordliste	2
4 Teori	2
5 Kravspesifikasjon	5
6 Teknisk Design	6
7 Utviklingsprosess	7
8 Implementasjon	8
9 Testing	12
10 Utrulling til sluttbruker (deployment)	12
11 Drøfting	12
12 Konklusjon - Erfaring	13
13 Referanser	15
14 Vedlegg	16

Figurliste

Figur 1: UML Use case diagram

Figur 2: UML Klassediagram av applikasjonen

Figur 3: UML Klassediagram av "Model"

Figur 4: UML Aktivitetsdiagram

Figur 5: UML sekvensdiagram for go() metode i "Game"-klassen.

Figur 6: UML sekvensdiagram for createGame() metode i "GameManager"-klassen

Tabelliste

Tabell 1: Ordliste

1 SAMMENDRAG

Gjennom utviklingen av spillmotoren “Paths” gitt i mappevurderingen fra IDATT2001 ved NTNU, har det blitt utviklet en robust og velfungerende applikasjon. Gjennom kravspesifikasjonen til oppgaven stilles det krav til hvilke verktøy som brukes og hvordan enkelte aspekter av applikasjonen skal utvikles. Produktet består av en applikasjon som kan kjøre forskjellige lagrede spill. Prosessen har gjennom bruk av smidige metoder og bruk av viktige kode-prinsipper, ført til at alle krav til funksjonalitet gitt i kravspesifikasjonen er oppnådd. Teamet har også implementert ytterligere funksjoner, dette inkluderer: lagring av spillobjekter, tekst til tale og mulighet for å lage historier gjennom brukergrensesnittet. Gruppemedlemmene konkluderte med at prosjektet står til det forventede nivået, men at flere funksjoner for spillet hadde vært ønskelig.

2 BEGREPER OG FORKORTELSER

UML	Unified Modeling Language
GUI	Graphical User Interface
JSON	JavaScript Object Notation
MVC	Model-View-Controller pattern
API	Application Programming Interface

3 INTRODUKSJON

3.1 Bakgrunn

Denne rapporten er skrevet på bakgrunn av en mappevurdering i faget Programmering 2, ved institutt for datateknologi og informatikk, ved Norges tekniske naturvitenskapelige universitet. Problemstillingen tar for seg utviklingen av en spillmotor for valgbasert og interaktiv historiefortelling. Applikasjonens spill består av interaktive historier som er delt opp i passasjer, der spilleren må ta ulike valg. Løsningen inkluderer et grafisk brukergrensesnitt og filhåndtering for å lagre spillinformasjon. Videre stilles det også krav til

tilstrekkelig bruk av enhetstesting og unntakshåndtering. Prosjektet skal også legges under versjonskontroll gjennom GitLab for å muliggjøre samarbeid.

3.2 Avgrensninger

Oppgaven inneholder avgrensninger for visse klasser og metoder som skal implementeres som en del av startfasen. Disse avgrensningene setter et klart rammeverk for hvordan grunnstrukturen til applikasjonen skal være, og sørger for at applikasjonen inneholder nødvendige logiske komponenter. I tillegg er det gitt avgrensninger om hvilke utviklingsverktøy som skal benyttes, blant annet skal JavaFX brukes for å utvikle GUI. Videre er det bestemt at GitLab skal brukes for versjonskontroll.

3.3 Begreper/Ordliste

Begrep (Norsk)	Begrep (Engelsk)	Betyding/beskrivelse
Spillmotor	Game engine	Rammeverk for å kunne lage et spill.
Valgbasert historiefortelling	Choice-based storytelling	En historie som endrer utfall basert på de valgene man tar.
Filhåndtering	File handling	En prosess der programmet jobber med å lese, skrive og manipulere data i filer.
Grafisk brukergrensesnitt	Graphical user interface	En visuell samhandling mellom bruker og program.

Tabell 1: Ordliste

4 TEORI

I dette kapittelet skal vi se nærmere på teorier og beste praksiser som er relevante for utviklingen av programvare. Dette inkluderer konsepter og prinsipper som modularisering, single responsibility, open-closed, kobling, cohesion, dry code og refaktorering. I tillegg skal vi se på designmønstre som singleton, factory og builder.

Et viktig prinsipp innenfor utviklingen av programvare er modularisering, som innebærer å dele programmet opp i mindre, uavhengige komponenter med spesifikke ansvarsområder og oppgaver (Jee, 2021). Ved å dele opp programmet på denne måten blir det mer strukturert og fleksibel, samtidig som at komponentene kan utvikles og gjenbrukes separat.

Et prinsipp som henger tett sammen med modularisering, er single responsibility. Dette prinsippet handler om at klasser og moduler skal kun ha et spesifikt ansvarsområde (Erinc, 2020). Slik gjør man koden mer lesbar og lettere å teste. Klare, definerte ansvarsområder fører også til redusert uønsket oppførsel i andre deler av systemet.

Ved å følge prinsippet om single responsibility, blir det lettere å oppnå prinsippet om open-closed. Dette prinsippet handler om at programmet skal være åpen for utvidelse, men lukket for endring (Erinc, 2020). Det betyr at man skal kunne implementere ny funksjonalitet i programmet, men ikke på bekostning av å endre den eksisterende koden. Dette kan oppnås ved å bruke grensesnitt, abstrakte klasser og polymorfi.

Ved å anvende modularisering og prinsippet om single responsibility oppnår man også høy cohesion i programmet. Dette er et viktig prinsipp i programvareutvikling som handler om at klasser kun skal ha et ansvar og utføre en spesifikk oppgave (Barnes og Kölling, 2016, s. 278). Hvor stor grad av cohesion man har i et program avhenger av hvor funksjonelt relaterte elementene i hver modul er. Med høy cohesion jobber elementene i en klasse mot et felles mål, mens lav cohesion betyr at elementene er lite sammenhengende og jobber mot ulike mål. Klasser med høy cohesion er lettere å vedlikeholde og mer gjenbrukbare.

Et annet viktig prinsipp omhandler koblingen mellom komponentene. Som skrevet av Cay S. Horstmann i hans bok Core Java (Horstmann, 2016, s. 134), er løse koblinger mellom komponenter ønskelig, slik at de er mest mulig uavhengige. Dersom det er sterk kobling mellom komponenter, kan endringer i en komponent føre til uønskede endringer i en annen. Med løse koblinger forhindres dette og gjør koden mer fleksibel og lettere å videreutvikle og vedlikeholde.

En annen faktor verdt å nevne er "DRY" (Don't Repeat Yourself) code. Dette prinsippet handler om å redusere duplisering av kode (Long, 2017). Når man dupliserer kode, øker sannsynligheten for feil og gjør det vanskeligere å vedlikeholde koden. Ved å følge dette prinsippet gjør man koden lettere å endre og mer gjenbrukbar.

For å kunne forbedre og optimalisere programmet, er refaktorering et nyttig verktøy. Hensikten med refaktorering er å forbedre koden uten å endre hovedfunksjonaliteten, samtidig som man tilpasser den til nye endringer og krav (Barnes og Kölling, 2016, s. 282). Ved å regelmessig utføre refaktorering forbedres kodekvaliteten. I tillegg øker lesbarheten, koden blir mer vedlikeholdbar og mindre kompleks.

I tillegg til de nevnte prinsippene finnes en rekke designmønstre som kan være hensiktsmessige å anvende i programvareutvikling. Den første av dem er singleton. Ved å implementere singleton klasser hindrer man at det kan opprettes flere instanser av samme klasse (Gupta, 2022). I tillegg gir man den ene instansen global tilgang og gjør den lett tilgjengelig gjennom hele applikasjonen.

Factory-mønsteret er et annet nyttig designmønster i programvareutvikling som brukes til å abstrahere opprettelsen av ulike typer av samme objekt (Gupta, 2022). Ved å anvende factory kan man flytte opprettelsen av objekter til en egen klasse eller metode og kalle den der man ønsker å opprette objektene. Fordelen med å bruke factory-mønsteret er at det gjør programmet mer fleksibelt og gjenbrukbart, og gir en løsere kobling mellom komponenter. I tillegg blir det lettere å introdusere nye elementer uten å endre eksisterende kode, som oppfyller prinsippet om open-closed.

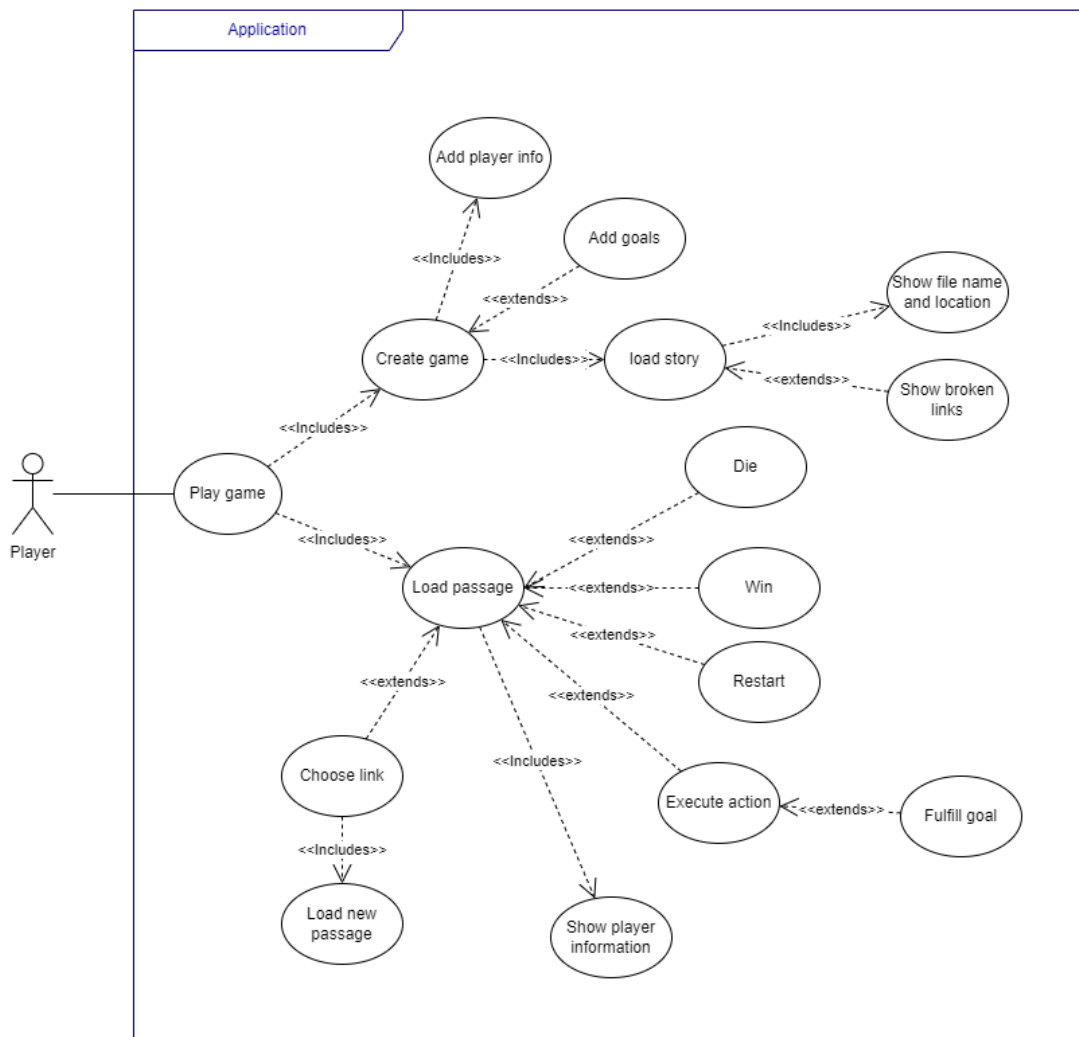
Builder-mønsteret er et designmønster som gjør opprettelsen av komplekse objekter mer fleksibel (Kiwy, 2021). Dette mønsteret gjør det mulig å konstruere objekter med valgfrie parametere. På denne måten kan man velge de parameterne som er hensiktsmessige å bruke på det aktuelle tidspunktet og samtidig ha muligheten til å endre og legge til flere av parameterne et senere.

5 KRAVSPESIFIKASJON

Applikasjonen har funksjonelle og ikke funksjonelle krav. Spillet skal være et interaktivt, ikke-lineært narrativ med flere passasjer. Programmet skal ha et grafisk brukergrensesnitt som viser spillets gang. Før spillet starter skal brukeren ha mulighet til å sette startverdier for karakterens helse og mål.

Fasaden til spillet vil bestå av en “Game” klasse med “Player” instanse, et “Story” objekt og en liste med “Goal” objekter. Det finnes fire typer mål: “ScoreGoal”, “HealthGoal”, “GoldGoal” og “InventoryGoal”, som alle skal arver fra et “Goal” grensesnitt. Gjennom spillets gang skal visse handlinger utføres på spilleren, dette skal skje gjennom klassene: “ScoreAction”, “HealthAction”, “GoldAction” og “InventoryAction”, som alle arver fra et “Action” grensesnitt. Forflytning mellom passasjer skjer “Link” objekter som dannet et tre av passasjer. Passasjene vil være en del av et “Story” objektet, hvor en av passasjene vil være spesifisert som den første passasjen i historien.

For å kunne lagre historiene skal programmet ha metoder for å lese til og fra tekstfiler på med filendelsen: “.paths”. Applikasjonen skal være robust, og må derfor ha nødvendige unntakshåndtering samt enhetstesting for å avdekke eventuelle feil. Prosjektet skal fra starten av legges under versjonskontroll for å distribuere koden mellom utviklerne.



Figur 1: UML Use case diagram

6 TEKNISK DESIGN

I dette kapittelet skal vi beskrive det overordnede bildet av den valgte løsningen. Produktet av prosjektet er en desktop-applikasjon, som bruker Model-View-Controller (MVC) arkitekturmønsteret som en sentral del av systemutviklingen. Dette arkitekturmønsteret deler systemet inn i tre hovedkomponenter med separate ansvarsområder. Den første komponenten er modellen, som inneholder applikasjonsdata og kjernefunksjonaliteten til systemet. Den andre komponenten er visningen, som samhandler med brukeren og henter inn og viser data. Den tredje komponenten er kontrolleren, som håndterer input-data fra brukeren og fungerer som et mellomledd mellom modellen og visningen.

Vi valgte dette arkitekturmønsteret for å oppnå en tydelig separasjon av ansvar, som gjør programmet modulært og fleksibelt. I tillegg er komponentene uavhengige av hverandre og kan derfor utvikles og gjenbrukes separat. Dette gjorde det mulig å utvikle kjernefunksjonaliteten parallelt med brukergrensesnittet uten å skape konflikt.

7 UTVIKLINGSPROSESS

Utviklingsprosessen har fulgt en smidig tilnærming, hvor det har vært fokus på fleksibilitet, samarbeid og kontinuerlige forbedringer. Utviklingsteamet har jobbet gjennom tre iterasjoner med fokus på å levere fungerende funksjonalitet i hver iterasjon, samtidig som de har prioritert forbedring av eksisterende kode basert på tilbakemeldinger fra læringsassistenter etter hver iterasjon.

I oppstartsfasen analyserte utviklingsteamet problemstillingen med konseptene og prinsippene fra “Teori”-kapittelet som grunnlag. Etter å ha fått et overordnet bilde av klassene som skulle representere applikasjonsstrukturen, startet selve utviklingen av programmet.

Den første iterasjonen bestod av utviklingen av applikasjonens kjernefunksjonalitet. Dette innebærer utvikling av entitetsklasser som “Link”, “Passage”, “Story”, “Player” og “Game”. I tillegg til entitetsklassene ble det også utviklet to grensesnitt, “Action” og “Goal”, hver med sine egne entitetsklasser som implementerer grensesnittet.

Etter å ha utviklet kjernefunksjonalitet implementerte teamet funksjonell programmering, filhåndtering, og lagde en lavnivå-prototype i form av en wireframe. Funksjonell programmering involverte bruk av Streams for å lage nye funksjoner i “Story” klassen. Som tidligere nevnt i kravspesifikasjonene, må programmet støtte filhåndtering av historier i tekstfiler. Dette ble løst ved å lage en “FileStoryHandler” klasse som bruker klassene `BufferedWriter` og `BufferedReader` for lesing og skrivning en historie til tekstfil i et spesifikt

format. For å oppnå bedre forståelse av brukergrensesnittet som skulle utvikles, ble det laget en wireframe, en enkel skisse som illustrerer utseende og layouten til brukergrensesnittet.

I den tredje og siste iterasjonen bestod utviklingen av å anvende designmønstre og implementere et grafisk brukergrensesnitt med JavaFX basert på prototypen. I tillegg utvidet utviklingsteamet spillet med egne funksjonaliteter. Det ble benyttet tre designmønstre i applikasjonen, builder, factory og singleton. Builder-mønsteret ble implementert i “Player” klassen for å muliggjøre opprettelsen av objekter av klassen med valgfrie parametere. Factory-mønsteret abstraherte opprettelsen av ulike typer “Action” objekter. Singleton ble implementert i “GameManager” klassen for å forhindre opprettelsen av flere instanser og gi alle klasser tilgang til objektet. Videre implementerte teamet funksjonalitet for å lagre spill ved å bruke JSON for filhåndtering av "Game"-objekter. Deretter ble det implementert mulighet for å legge til “stories” fra GUI. Til slutt ble det implementert tekst-til-tale-funksjonalitet for å gjøre applikasjonen mer universelt utformet.

8 IMPLEMENTASJON

Prosjektet omfattet en rekke forskjellige verktøy og biblioteker for å sikre et robust og solid produkt med ønskede funksjoner.

For å utvikle front-enden av programmet i henhold til kravspesifikasjonene, har JavaFX vært den eneste plattformen som var tatt i bruk. For å utforme elementene i JavaFX har det blitt brukt CSS-stilark. Resten av kildekoden er også skrevet utelukkende i Java for å oppfylle kravspesifikasjoner.

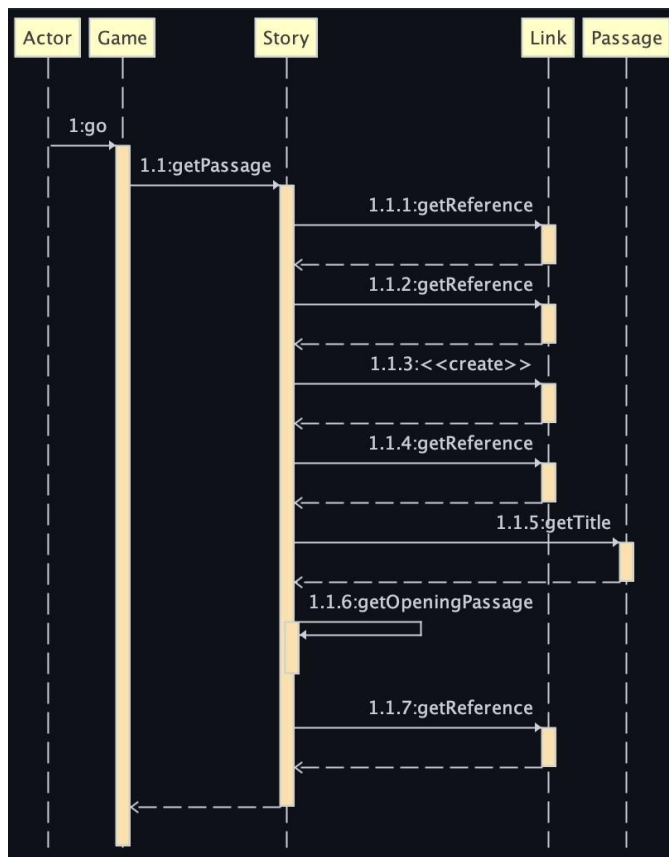
For å forbedre brukervennligheten for synshemmede, ble et API ved navn FreeTTS tatt i bruk for å generere en syntetisk stemme. I applikasjonen blir dette brukt for å lese passasje tekst, samt link referanser ut høyt. Det har i sammenheng med implementasjonen av dette også blitt lagt til dependencies fra samme utgiver, som vil gjøre det lettere for videre utvikling om man ønsker å legge til støtte for flere språk eller andre stemmer.

Gjennom prosjektet har teamet brukt IntelliJ IDEA for å utvikle programvaren. Dette ga en enkel integrasjon med utviklingsverktøyet Checkstyle for å sikre at koden følger Google sin stil-guide, og at koden opprettholder en konsekvent kodelstil for å øke lesbarheten. For prosjektets slutfase ble SonarLint brukt som et tillegg til IntelliJ sin innebygde lint for å forhindre dårlig kodelstil og kritiske feil.

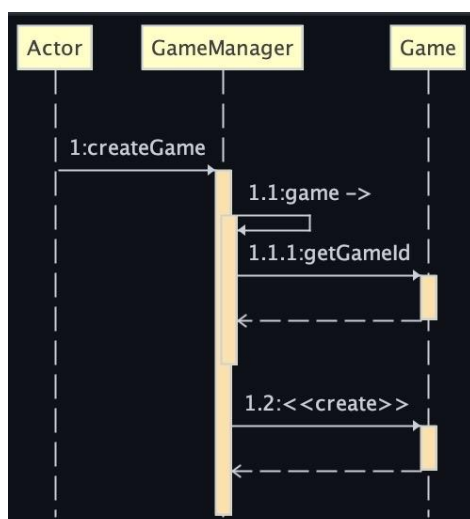
I tillegg til kjernemodulene som implementerer applikasjonens kjernefunksjonalitet og oppfyller kravspesifikasjonen, er det inkludert klasser for filhåndtering av JSON-filer og en "GameManager"-klasse. For å lagre spillobjekter falt valget på å bruke JSON filer. Lagringen oppnås ved å bruke Java-biblioteket Gson for å konvertere informasjonen om spillene til JSON filer, som deretter blir lagret som en ressurs. "Game Manager"-klassen er ansvarlig for å administrere opprettelse, sletting og lagring av spillobjekter.



Figur 2: UML Klassediagram av applikasjonen



Figur 5: UML sekvensdiagram for go() metode i “Game”-klassen



Figur 6: UML sekvensdiagram for createGame() metode i “GameManager”-klassen

9 TESTING

Testing var avgjørende for å kvalitetssikre programvaren under utviklingsprosessen. Enhetstestene kvalitetssikret de implementerte metodene i klassene, men fungerte også som et sikkerhetsnett for fremtidig utvikling og refaktoring. Etter hvert som applikasjonen ble utviklet, ble testene brukt for å sikre at endringene ikke brøt eksisterende funksjonalitet. Testmetodene er utformet slik at de er isolerte og uavhengige, og tester kun en spesifikk funksjonalitet per test. Dermed var det lettere å oppdage og behandle feil i systemet, samtidig som det gjorde det lettere å vedlikeholde testmetodene dersom koden som testes blir endret. For å sikre god testdekning, var både positive og negative tester implementert for å sjekke om metodene fungerer korrekt under forskjellige forhold.

10 UTRULLING TIL SLUTTBRUKER (DEPLOYMENT)

Appikajsoen er tilgjengelig som en zip-fil fra <https://gitlab.stud.idi.ntnu.no/gruppe44/mappevurdering-idatt2001>. Denne kan pakkes ut, og etter å ha fulgt instruksjonene i README, kan den kjøres fra kommandolinjen ved hjelp av "mvn javafx:run".

11 DRØFTING

Under utviklingen av applikasjonen har det vært fokus på å anvende flere designprinsipper for å oppnå god kodekvalitet. Vi har først og fremst anvendt prinsippet om modularisering ved å bruke MVC-mønsteret og prinsippet om single responsibility når vi designet klassene. Vi har oppnådd høy kohesjon ved å sikre at klassene har en klare og spesifikke hensikt, samtidig som metodene og feltene er funksjonelt relaterte til hverandre. Metodene følger prinsippet om single responsibility og gjenbrukes for å ikke repetere kode, og dermed oppfyller konseptet om DRY code. I tillegg inneholder applikasjonen grensesnitt for å lettere kunne implementere ny funksjonalitet uten å måtte endre eksisterende kode, og dermed oppnår prinsippet om å være åpen for utvidelse og lukket for endring (open-closed). Som tidligere nevnt i teorikapittelet, ønsker vi løs kobling mellom komponentene i applikasjonen. For å oppnå dette

benytter vi innkapsling av klassevariabler, som kun kan nås ved å opprette et objekt av klassen og kalle på offentlige metoder. I tillegg har vi jevnlig refaktoreert koden for å forbedre kodekvaliteten og oppnå løsere kobling.

Vi begynte tidlig å bruke listeners til å endre alle dimensjoner til programmet når bruker endret størrelse, men fant fort ut at dette ikke var ideelt. Blant annet fikk vi bugs som gjorde at hele siden ble større når man klikket på enkelte knapper, selv om knappene ikke skulle endre størrelsen, og maks størrelse allerede var satt. Etter mye utforskning av forskjellige løsninger, måtte vi ty til å bruke for eksempel BorderLayout. Den nye løsningen har mange fordeler, men gjør det vanskelig å endre alle størrelser og posisjoner 100% nøyaktig når brukeren endrer størrelsen på vinduet, slik vi opprinnelig ønsket.

Når man ikke bruker FXML mister man også bruken av metodene som åpner nye scener lettere. Dette løste vi å lage instanser av alle de forskjellige menyene i app-klassen, menyene sender deretter informasjonen de henter ut fra bruker tilbake til App klassen gjennom listeners, som kontrollerer programmet videre. Dette løste utfordringen med å åpne scener, men resulterte i at App klassen kan virke uoversiktlig ved første inntrykk.

Kildene teamet har brukt i prosjektet er nøye utvalgte og skrevet av fagfolk, og dermed gir en betydelig grad av troverdighet og sikkerhet.

12 KONKLUSJON - ERFARING

Etter at applikasjonen var ferdig utviklet, var vi fornøyde med det endelige resultatet av applikasjonen. Når det gjelder kravspesifikasjonene, føler vi at vi har oppfylt alle kravene til applikasjonen på en tilfredsstillende måte. I tillegg har vi anvendt viktige design prinsipper gjennom utviklingen, noe som har ført til en mer optimal og vedlikeholdbar kode.

Til tross for at vi er fornøyde med det nåværende produktet, er det flere funksjoner vi ønsket å implementere, men som ikke ble oppnådd. Et eksempel på dette var å innføre flere språk til tekst-til-tale-funksjonen, samt å legge til muligheten til å oversette språk i spillet. Vi er også

fornøyde med måten man kan lage historier til spillet fra GUI-en vår, noe som setter et unikt preg på vårt spill, men her finnes det dessverre enkelte funksjoner som ikke er helt optimaliserte, noe som setter begrensninger for hva brukeren kan gjøre. Ved eventuelt videre arbeid, ønsker vi å forbedre disse funksjonene for å gi en enda bedre brukeropplevelse.

Hvis vi skulle gjennomføre prosjektet en gang til, ville vi ha valgt å droppe vår visjon om at programmet skal endre størrelse 100% perfekt. Ettersom det var altfor tidkrevende å implementere det i koden.

13 REFERANSER

Barnes, D. J og Kölling, M. (2016) *Objects First with Java: A Practical Introduction Using BlueJ*. 6. utg. England: Pearson.

Erinc, Y. K. (2020). *The SOLID Principles of Object-Oriented Programming Explained in Plain English*. Tilgjengelig fra:
<https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/> (Hentet: 19. mai 2023).

Gupta, L. (2022) *Java Factory Pattern Explained*. Tilgjengelig fra:
<https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/> (Hentet: 19. mai 2023).

Gupta, L. (2022) *Java Singleton Pattern Explained*. Tilgjengelig fra:
<https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/>
(Hentet: 19. mai 2023).

Horstmann, C.S. (2016) *Core Java*. 10. utg. New York: Prentice Hall.

Jee, C. (2021) Modularization in Software Engineering, *Medium*, 10. oktober. Tilgjengelig fra:
<https://medium.com/@caitlinjeespn/modularization-in-software-engineering-1af52807ceed>
(Hentet: 18. mai 2023).

Kiwy, F. (2021) *Exploring Joshua Bloch's Builder design pattern in Java*. Tilgjengelig fra:
<https://blogs.oracle.com/javamagazine/post/exploring-joshua-blochs-builder-design-pattern-in-java> (Hentet: 19. mai 2023).

Long, J. (2017) DRY vs WET Code, *Medium*, 12. november. Tilgjengelig fra:
<https://medium.com/jl-codes/dry-vs-wet-code-589c564aa5aa> (Hentet: 19. mai 2023).

14 VEDLEGG

Vedlegg 1 - Link til GitLab Wiki

Wiki for prosjektet er lokalisert her:

<https://gitlab.stud.idi.ntnu.no/gruppe44/mappevurdering-idatt2001/-/wikis/Main-Page>

Vedlegg 2 - Wireframes



[← Go Back](#)

Load game

Number	Name	
1	Geir's adventures.	Load
2	Olav's quest.	Load
3	Empty save	

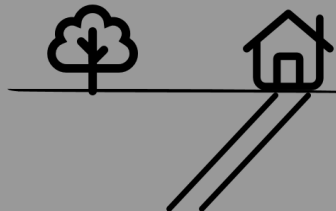
[← Go Back](#)

Create game

Game name:	<input type="text" value="enter name"/>
Player name:	<input type="text" value="enter name"/>
Health	<input type="text" value="10"/> ▼
HealthGoal:	<input type="text" value="enter a number"/>
GoldGoal:	<input type="text" value="enter a number"/>
InventoryGoal:	<input type="text" value="Iron Axe"/> ▼
ScoreGoal	: <input type="text" value="enter a number"/>

[+ Create](#)[Exit Game](#)[Reset game](#) 57 34

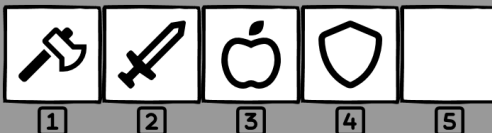
You arrive at the end of the path, there is nothing there except an abandoned house and a large tree.



What will you do?


[Enter the house](#)


[Try to climb the tree](#)




Exit Game

Reset game


 57


 34


You scare the crazy person living in the house by breaking in without knocking. He uses his second amendment rights, making you lose 3 lives, resulting in your inevitable death.





YOU DIED

 1

 2

 3


 4


 5

New Game


Exit Game

Reset game

 57

 34


You arrive by the tree, and something shiny deep between the roots of the large pine.





What will you do?


1 Use your axe


3 Eat an apple

 1

 2

 3


 4


 5


Exit Game

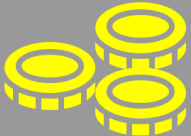
Reset game

By breaking away the roots, you find an old pouch full of three gold coins


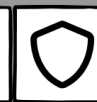
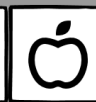




 60

 34



You have reached your Gold Goal of 60 coins!



1

2

3


4


5


Exit Game


Reset game

You eat the apple and gain 2 health. Suddenly you feel tired, and decide to go to sleep by the fence surrounding the house...








 57

 34



What will you do?

Continue



1

2

3

4

5