Projektbeskrivning

Textäventyr

2018-05-27

Projektmedlemmar:
Robert Sehlstedt <robse205@student.liu.se>
Payam Tavakoli <payta322@student.liu.se>

Handledare:
Teodor Riddarhaage <teori199@ida.liu.se>

Planering

1. Introduktion till projektet

Vi ska utveckla ett textbaserat äventyrsspel. Spelaren är del av ett narrativ där målet är okänt och berättelsen kommuniceras mellan spelaren och spelet genom textdialoger.

Spelaren kommer att kunna röra sig mellan och besöka olika platser, varje plats är en del av en övergripande karta och har ett syfte i berättelsen. På dessa platser kommer till exempel att finnas interaktiva objekt som även dessa (oftast) har något syfte för berättelsens fortsättning. När spelaren lyckas ta sig framåt i handlingen kommer denne att belönas med poäng, som kan jämföras i en lista med alla lokala poängrekord som satts.

Spelets slutmål är okänt och det är upp till spelaren att lista ut hur man gör framsteg. Under vissa delar av berättelsen finns fallgropar som kan sabotera för spelaren eller göra att den förlorar spelet.

2. Ytterligare bakgrundsinformation

Det finns ingen ytterligare nödvändig bakgrundsinformation.

3. Milstolpar

Nedan är de milstolpar vi önskar att uppnå under projektets gång.

#	Beskrivning
1	Vi får upp två rutor, en större ruta som endast visar text (textfönster) och en mindre ruta under (mindre i höjd) som tar indata från spelaren (skrivfönster). Om spelaren skickar ett meddelande så visas det längst ner i textfönstret.
2	En tolk behandlar strängarna som skrivs av användaren och kan ge respons på godtycklig sträng. Tolken skriver sitt svar längst ner i textfönstret.
3	Har en testplats som vi står på, som är grund för vidare testning av platsfunktionalitet. Exakt uppbyggnad avvägs noga.
4	Testplatsen kan tolka meddelanden från skrivfönstret (polymorfism borde kunna användas lämpligt), antingen genom en extern tolkklass eller att den tolkar själv.
5	Testplatsen kan innehålla ett item som spelaren kan plocka upp och lägga i ett inventory. (Behöver inte kunna visa inventory grafiskt än, men skriv i textfönster om man skriver I/inventory/inv.)
6	Inventory visas grafiskt i en ruta till höger om textfönstret.
7	Vi kan gå mellan flera olika testplatser genom att skriva w/e/s/n.

	Testplatserna ger ett meddelande när vi når platsen. Spelaren kan få platsbeskrivning genom kommando.
8	Testplatserna ger ett meddelande som är beroende på de items som fortfarande finns kvar. Dvs om spelaren tagit upp ett item försvinner det och ska sedan inte tas med i platsbeskrivningen.
9	Items kan användas för att interagera med spelet, till exempel en nyckel kan öppna ett (visst) lås.
10	Spelaren kan tilldelas poäng när denne gör någonting i spelet. I framtiden ges poängen när spelaren tar sig framåt i berättelsen. En ”stegräknare” räknar hur många actions spelaren har tagit i spelet.
11	Spelaren kan på något sätt förlora spelet, då sparas poängen i en poänglista. Denna lista sparas lokalt i en fil som laddas när spelet startar. Poängen som sparas beräknas av (poäng dividerat med antal steg spelaren tagit) När spelet slutar (spelaren förlorar) stängs spelet ner.
12	Ta fram en berättelse, utifrån denna kan vissa framtida implementationer ändras lite.
13	Har implementerat en klocka som kan räkna hur många sekunder vi varit på en viss plats. Kan användas för en massa roligt i storyn.
14	Alla platser i spelet implementeras utifrån den berättelse som framtagits.
15	Det är möjligt att öppna hela spelet från en fil (krävs för betyg 5).
16	När spelet startar kommer en meny med knapparna ”Play”, ”Quit”. På sidan syns top x av highscores.
17	När spelaren “dör” i spelet så kommer man till menyn, istället för att spelet stängs ner.
18	Kan nu göra ”extrasaker”. Implementera hunger och törst, så att spelaren måste hitta mat och dryck under spelets gång.
19	Det är nu möjligt att välja mellan olika berättelser i menyn, med olika svårighetsgrader. Olika highscores för alla narrativ, kan välja vilken man vill se.
20	Implementera sanity, där spelaren kan bli galen och därmed förlora spelet om mätaren går till 0.

4. Övriga implementationsförberedelser

Vi har tänkt att använda JavaFX istället för Swing för att göra de grafiska komponenterna. I nuläget har ingen av oss använt JavaFX, men ser det som ett bra sätt att lära sig då detta verkar vara mer populärt nu för tiden än Swing.

Platserna vill vi bygga upp genom ett interface som bland annat kräver en tolkningsmetod och en beskrivningsmetod, move, pickup osv, därefter skapar vi en abstrakt klass som innehåller viss funktionalitet som alla objekten har gemensamt. och sedan skapar vi en egen klass för varje plats i spelet.

Möjligtvis kan vi på något sätt skapa en generisk platsklass och sedan ha enumklasser som beskriver just vilken plats det är, men det blir nog ingen bra lösning då varje plats ska ha ganska mycket skiljande funktionalitet också. Vi är inte heller helt säkra på hur varje plats själv ska tolka meddelanden, om de ska använda en extern tolkklass som gör det åt dem eller tolka det själva.

Varje plats kommer ha pekare till grannplatser, alltså sådana platser som vi kan gå till genom att skriva w/e/s/n. Om spelaren då skriver ”w” så körs metoden *move(west)* för den platsen vi står i. Om platsen inte har en pekare västerut så säger spelet att spelaren inte kan gå dit.

Alla fysiska objekt på en plats är en del av en objektklass(Item) som innehåller ett namn och en beskrivning. Vi vet ännu inte hur funktionaliteten för att använda items kommer se ut. Platsbeskrivningen blir nog en Stringbuilder som lägger till vissa strängar beroende på om ett item fortfarande finns på platsen eller inte. Alltså måste det finnas pekare till varje item i listan, och om de är null (spelaren har tagit upp det), så skrivs det inte ut med platsbeskrivningen.

Poängsystemet kan rimligtvis göras genom en singleton denna laddar upp en lokal fil med en lista över tidigare highscores. Innan vi fått en meny behöver vi bara printa poängen på något sätt.

5. Utveckling och samarbete

Vi båda satsar på betyget 5, vi har tillsammans upprättat ett google-docs där vi fortlöpande ska dokumentera och utvärdera vad vi har gjort under projektet. Både för att ha bra koll på vad den andre gjort och hur den tänkte med sin lösning, men också för att ha bra underlag längre fram i kursen.

Det känns som en bra idé att arbeta mycket tillsammans tidigt under projektet, då ”coren” av spelet kommer bli avgörande för hur implementationen senare ser ut, och ingen av oss har tidigare erfarenhet av JavaFX, Då vi är vänner har vi god kommunikation och har tänkt komma på många labbtillfällen, och kommer även att plugga mycket helger och kvällar.

SLUTINLÄMNING

6. Implementationsbeskrivning

I detta kapitel beskrivs hur programmet är implementerat, och diskuterar även de lösningar vi valt att använda.

6.1. Milstolpar

Nedan är de milstolpar vi lyckats uppnå under projektets gång.

Grönt betyder att milstolpen är avklarad.

Gult betyder att milstolpen är delvis avklarad.

Rött betyder att milstoplen inte är avklarad.

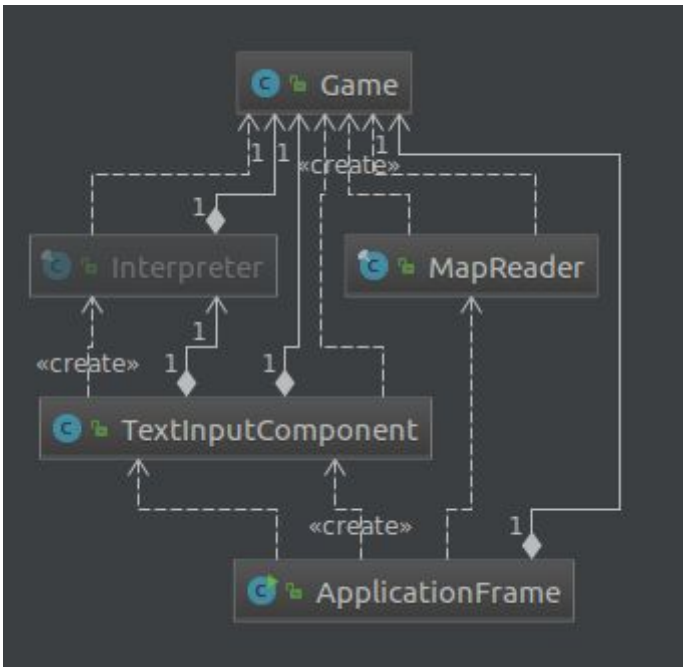
1	*Vi får upp två rutor, en större ruta som endast visar text (textfönster) och en mindre ruta under (mindre i höjd) som tar indata från spelaren (skrivfönster). Om spelaren skickar ett meddelande så visas det längst ner i textfönstret.*
2	*En tolk behandlar strängarna som skrivs av användaren och kan ge respons på godtycklig sträng. Tolken skriver sitt svar längst ner i textfönstret.*
3	Har en testplats som vi står på, som är grund för vidare testning av platsfunktionalitet. Exakt uppbyggnad avvägs noga.
4	Testplatsen kan tolka meddelanden från skrivfönstret (polymorfism borde kunna användas lämpligt), antingen genom en extern tolkklass eller att den tolkar själv.
5	Testplatsen kan innehålla ett item som spelaren kan plocka upp och lägga i ett inventory. (Behöver inte kunna visa inventory grafiskt än, men skrivs i textfönster om man skriver I/inventory/inv)
6	Inventory visas grafiskt i en ruta till höger om textfönstret.
7	Vi kan gå mellan flera olika testplatser genom att skriva w/e/s/n. Testplatserna ger ett meddelande när vi når platsen. Spelaren kan få platsbeskrivning genom kommando.
8	Testplatserna ger ett meddelande som är beroende på de items som fortfarande finns kvar. Dvs om spelaren tagit upp ett item försvinner det och ska sedan inte tas med i platsbeskrivningen.
9	Items kan användas för att interagera med spelet, till exempel en nyckel kan öppna ett (visst) lås.

10	Spelaren kan tilldelas poäng när denne gör någonting i spelet. I framtiden ges poängen när spelaren tar sig framåt I berättelsen. En ”stegräknare” räknar hur många actions spelaren har tagit i spelet.
11	Spelaren kan på något sätt förlora spelet, då sparas poängen i en poänglista. Denna lista sparas lokalt i en fil som laddas när spelet startar. Poängen som sparas beräknas av (poäng dividerat med antal steg spelaren tagit) När spelet slutar (spelaren förlorar) stängs spelet ner.
12	Ta fram en berättelse, utifrån denna kan vissa framtida implementationer ändras lite.
14	Alla platser i spelet implementeras utifrån den berättelse som framtagits.
15	Det är möjligt att öppna hela spelet från en fil (krävs för betyg 5).

6.2. Dokumentation för programkod, inklusive UML-diagram

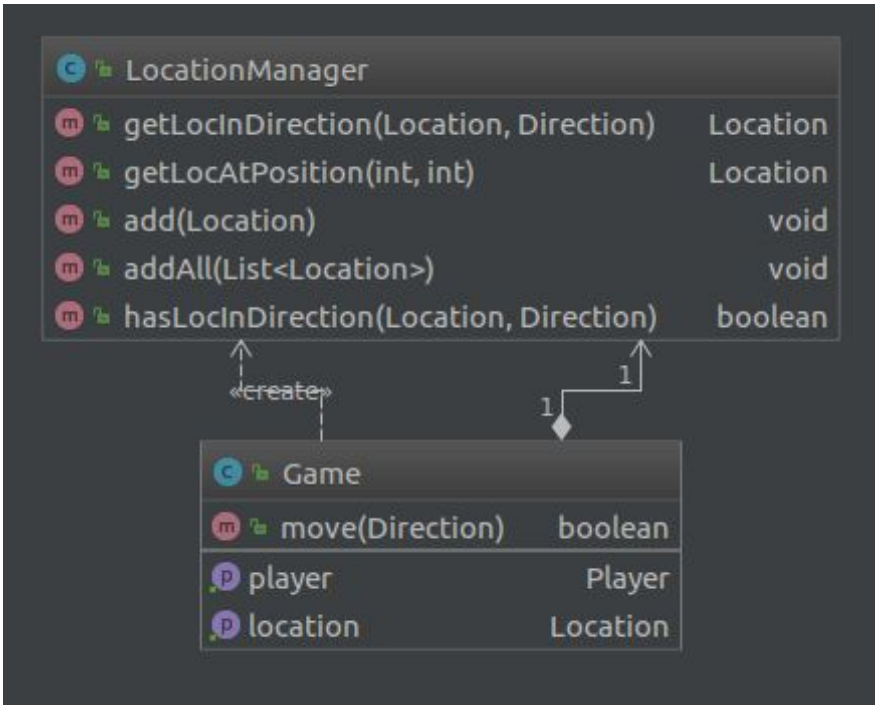
Vi påbörjade projektet med vetskapen att vi ville använda JavaFX istället för Swing. Den information vi hittade på internet angående biblioteket lät lovande. Främst ville vi hitta ett enkelt sätt att ge text som indata, tolka denna text och sedan ge ett svar med text som utdata.

Lösningen vi fann var att skapa en ApplicationFrame som är en underklass till JavaFX-bibliotekets Application-klass, denna innehåller alla spelets grafiska komponenter (se bild 1).
AppliacationFrame är även den klass som skapar ett nytt Game med hjälpklassen MapReader.



1. UML Diagram för ApplicationFrame

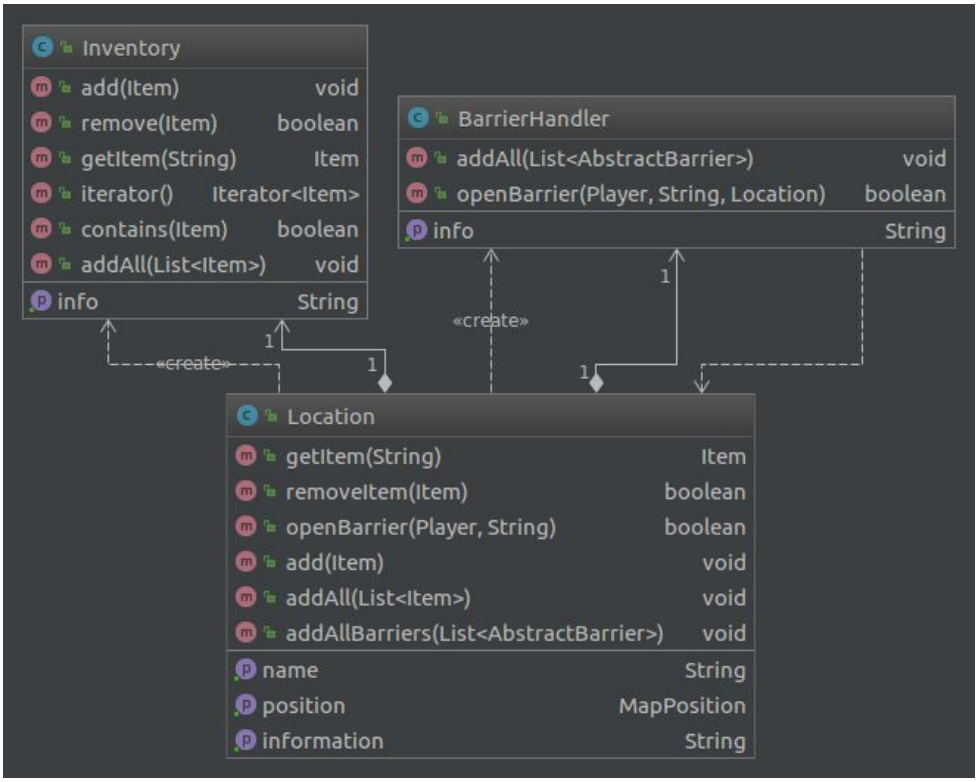
TextInputComponent är den komponent som spelaren använder för att förmedla information till spelet. Denna klass gör även viss logik, i form av att den kontrollerar om spelaren har vunnit eller ej, och agerar enligt denna information, Informationen vidarebefordras sedan till en tolkklass, Interpreter, som avgör om det som spelaren har skickat till spelet betyder någonting eller inte. Den skickar sedan vidare informationen till Game om nödvändigt, och ger ett svar till spelaren. Alternativt säger den till spelaren att informationen är ogiltig.



2. LocationManagers metoder

Ett Game innehåller en spelare (Player), nuvarande plats (Location) samt en klass som behandlar och innehåller alla platser i spelet, LocationHandler (se bild 2).

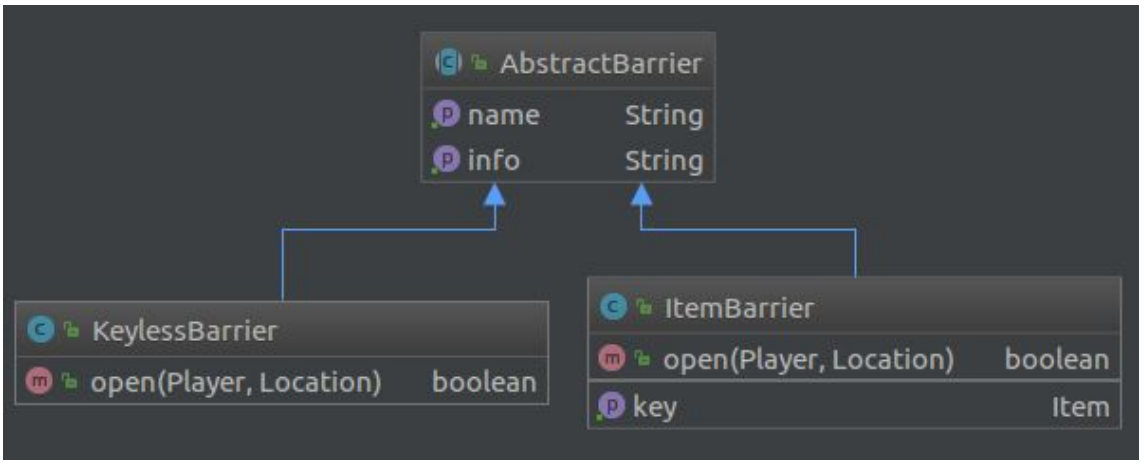
LocationManager håller reda på vilka platser som är grannar, om det finns platser på en viss position eller om man vill hitta en ny plats i en riktning från där man står just nu.



3. Översikt på samspelet mellan klasserna Location, BarrierHandler och Inventory

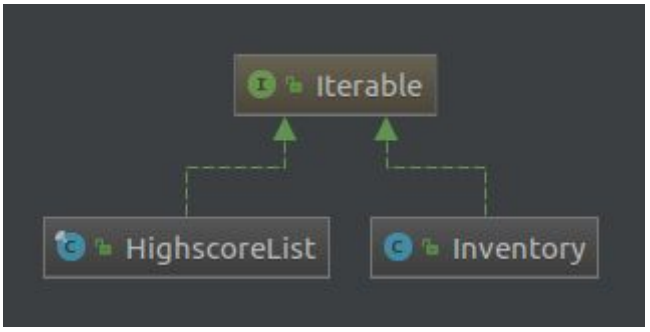
Spelets huvudingrediens är platser (Location), spelaren går mellan dessa och kan hitta olika saker på varje plats som den kan interagera med. Platsen innehåller ett Inventory, där alla tillgängliga Items finns. Platsen kan då användas för att hitta Items som finns i den, ta bort eller lägga till Items. (se bild 3)

Det finns dessutom en BarrierHandler som hanterar de barriärer som finns på platsen. Dessa barriärer är konkreta implementationer av den abstrakta barriären AbstractBarrier, och används för att gömma ett Inventory. (se bild 4)



4. Skillnaden mellan en KeylessBarrier och en ItemBarrier är key fältet som krävs för att öppna en ItemBarrier

Barriärer är någonting som blockerar andra delar av spelet, till exempel en kista eller dörr som kräver nycklar för att öppna, alternativt en låda som inte kräver något annat än en handling. Den nuvarande versionen stödjer dock inte dörrar. Barriärerna har en metod som försöker öppna barriären, och en KeylessBarrier och ItemBarrier har då olika krav för att öppna sig själva. När de öppnar sig lägger de till sitt inventory på platsen.



5. HighScoreList och Inventory är båda itererbara

Spelet har en del behållare som innehåller en lista av någonting. Det har då ibland varit smidigt att använda gränssnittet Iterable (se bild 5),som gör att vi kan iterera över ett objekt i klassen när man gör en loop som går igenom just den klassen. Detta är någonting som vi gjort för Inventory som är en behållare för Items, och HighscoreList som är en behållare för Highscores.

Highscores är en abstraktion av spelarens poäng, den finns i Player och innehåller antal poäng och steg som spelaren har tagit. Den kan dessutom beräkna en "score" med hjälp av dessa poäng och steg.

När spelet startar så skapar ApplicationFrame ett Game med hjälp av MapReader, vilket är en hjälpklass som går igenom en fil i JSON-format. Dessa filer måste ha en viss struktur och utgör en karta i spelet.

6.3. Användning av fritt material

Vi har använt JavaFX som grafisk grund för projektet, vilket ingår i Java 8 paketet.
Vi har även använt JSON för att spelet ska kunna läsa in data från filer och skapa objekt av dem. Specifikt använder vi JSON för att MapReader ska kunna läsa in kartor.

6.4. Användning av objektorientering

1. Objekt/Klasser

Klasser är ett väldigt kraftfullt verktyg för att abstrahera funktionalitet. Om vi till exempel behöver kunna utföra en grupp relaterade uppgifter kan dessa samlas i en egen klass.

När vi började arbeta med olika typer av barriärer hade vi först flera olika listor i varje Location för varje typ av barriär, och dessutom två olika metoder för att öppna dem. Vi bestämde oss för att istället skapa en BarrierHandler, som innehöll en lista med AbstractBarrier. Då behöver bara metoden open i Location kalla på metoden open i BarrierHandler, som hanterar öppningen av barriärerna.

Om man arbetar funktionellt och inte objektorienterat kan man skapa abstrakta datatyper. I detta fall hade vi då kunna skapa en datatyp som representerades med hjälp av en lista. Därefter skulle vi ha implementerat alla funktioner som behövs för att manipulera denna lista på samma sätt som vi nu gjort med objektorientering.

Den objektorienterade lösningen vinner främst på enkelheten att arbeta med. Båda lösningarna skapar i praktiken abstrakta datatyper, men att skapa en egen klass som alltid håller koll på sig själv är betydligt mer strukturerat. I alternativet hade vi möjligtvis behövt göra flera abstraktioner (precis som vi gjort), och det blir väldigt mycket samlad kod endast för att skapa och manipulera datatypen.

2. Subtypspolymorfism

Vi vill återigen använda exemplet ovan för att förklara användningen av subtypspolymorfism. AbstractBarrier är den abstrakta klassen som innehåller grundläggande funktionalitet för alla typer av barriärer. Denna innehåller dessutom en abstrakt metod som kräver att alla implementationer av den innehåller metoden open.Open ska användas för att se om barriären kunde öppnas med den givna informationen, och i så fall öppna den. Alla de olika konkreta barriärerna måste också ha olika metoder, eftersom de inte öppnas med samma villkor.

Vi lät då vår BarrierHandler gå igenom hela listan med alla AbstractBarrier och försöka öppna dem alla med samma givna information. På detta sätt behövde vi inte skapa en egen metod för alla olika implementationer av AbstractBarrier, utan varje klass hanterar sin egen öppning.

Lösningen utan objektorientering blir att helt enkelt titta vilken typ av barriär det handlar om, och sedan köra en open-funktion för den typen, förutsatt att man skapat en abstrakt datatyp, likt den som diskuterades i egenskap 1.

Det vi vinner genom objektorienteringen i detta fall är mycket mer påtagligt. Det blir mer modulärt eftersom vi inte behöver lägga till nya kontroller där vi vill försöka öppna alla olika barriärer, vi behöver bara lägga till en ny typ som ärver från AbstractBarrier.

3. Ärvning med overriding

Overriding i den bemärkelsen att vi skrivit över våra egna metoder har inte använts i projektet, men här kommer vi att ge ett exempel på hur det skulle kunna se ut.

Hypotetiskt skulle vi kunnat göra om AbstractBarrier till en GenericBarrier, som alltså inte är abstrakt. Den skulle implementera en metod open som är identiskt med den metod som just nu används i KeylessBarrier, därefter utökar vi GenericBarrier till ItemBarrier och gör en överskridning på metoden open i den, som istället kommer att titta om ett Item finns i spelarens Inventory.

I praktiken skulle detta ge oss samma funktionalitet som redan finns, men att vi då istället använder oss av override.
Det kan på liknande sätt som nu användas för polymorfism.

På liknande sätt som vid punkten om subtypspolymorfism skulle en implementering utan objektorientering ske genom abstrakt datatyp där det inte finns någon overriding. Vilket innebär att varje konkret implementation skulle behöva skapa sin egen open-funktion.

Ärvning och objektorientering kan i detta fall spara mycket kod. Det kan användas likt subtypspolymorfism, men där man vill att det ska finnas en standardmetod för hur dessa behandlas och endast vissa specialfall behöver ändras.

Här är det väldigt uppenbart att den objektorienterade lösningen är bättre än en abstrakt datatyp implementerad med funktioner. Just för att man slipper skapa konkreta implementationer i varje datatyp. Det blir även mer överskådligt och strukturerat.

4. Inkapsling

Denna egenskap är någonting som används för att dölja implementationen av någonting. Ett exempel är vår klass BarrierHandler som ska hantera alla olika barriärer som finns på en Location. Utomstående klasser behöver inte veta exakt hur BarrierHandler fungerar internt, utan endast vilken information som klassen delar med sig av. I den nuvarande versionen innehåller den endast en lista där alla barriärerna finns, men vi skulle kunna införa en annan implementation, det spelar ingen roll för utomstående. Det är en del i idén att klasser ska ta hand om sig själv och man vill inte att andra klasser ska kunna komma och manipulera all

information som finns inom en klass.

Återigen skulle en abstrakt datatyp vara ett sätt att göra någon typ av inkapsling där man gömmer implementationen. Det känns lite som ett eko nu men eftersom objektorientering är ett sätt att abstrahera datatyper vore det naturligt att använda dessa.

Den objektorienterade versionen är såklart enklare att arbeta med då den är specialanpassad just för detta syfte. Dessutom är det enkelt att bli begränsad angående vilken typ av information som en abstrakt datatyp innehåller, och vill man verkligen gömma mycket information blir det väldigt omfattande kod.

6.5. Motiverade designbeslut med alternativ

1. Till en början diskuterade vi hur Interpreter skulle utformas för att den enkelt skulle kunna användas. Vi hade en tanke om att den skulle vara en singleton, eftersom det i ett tidigt stadie kändes som att den bara skulle tolka text och vidarebefordra information. Det som upptäcktes var att även tolken behöver komma åt spelets information, och flera av de grafiska komponenterna för att uppdatera dem när någonting skedde.

   Eftersom vi då behövde information om en viss instans av spelet var inte singleton en bra lösning längre. Just i detta projekt hade det säkert fungerat att använda en singleton, men vi ville skapa ett realistiskt program där man skulle kunna starta flera instanser av spelet.

   Istället gjorde vi en vanlig konstruktor som tar in de grafiska komponenterna och Game, då kan tolken alltid komma åt dessa på ett smidigt sätt.

2. Spelet behövde ett system för att hantera platsers positioner, och då ville vi skapa en klass som representerar just dessa positioner. Då skapade vi klassen MapPosition, vilken innehåller en position i planet (x,y). Vi valde denna representation för enkelhetens skull, även om den inte är perfekt. Det finns vissa problem med implementationen eftersom man just nu inte kan skapa en karta där två platser ligger bredvid varandra, men utan att man kan gå mellan dem. LocationManager är sedan den klass som kopplar klasserna till varandra, med hjälp av dessa MapPositions.

   En bättre lösning hade varit att en LocationManager på något sätt kan veta vilka vägar som leder ut från en Location. Till exempel om en MapPosition innehåller den informationen. Vi kan tänka oss en lista bestående av Direction som finns i MapPosition. Det känns som en rimlig lösning och hade gjort det enklare att implementera dörrar dessutom. Anledningen till att vi inte hunnit så långt är dock avsaknad av tid.

3. Användning av Items var en svårighet som uppkom, och hur vi skulle implementera dem på ett rimligt sätt. Först skapade vi en enum-klass (ItemType) med till exempel STATIC och CONSUMABLE. Vi upptäckte dock att detta inte fungerade särskilt väl med hur koden såg ut i övrigt, och vi valde istället att ett item endast beskrivs med hjälp av sitt namn. Alltså är allt vi kollar på när vi matchar ett Item till en användning, om det

heter vad det borde heta.

En bättre och mer objektorienterad lösning hade varit att låta olika typer av items ha sina egna use-metoder. Denna lösning diskuterades men vi kunde inte se hur det skulle passa in i koden och fick välja det mindre attraktiva alternativet.

4. Det behövdes ett sätt att röra sig mellan olika platser och Game är den klass som håller reda på vilken plats vi står i just nu. Vi valde då att skapa en enum-klass Direction som innehåller de fyra väderstrecken syd, väst, nord och öst. Dessa riktningar innehåller också information om deras riktning i x-led och y-led, vilket gör att de kan användas för att flytta oss från en MapPosition till en annan.

   En LocationManager kan då använda sig av metoden getLocInDirection, som använder denna ändring i planet för att hitta en ny plats i den riktning som eftersöks.
   En annan lösning hade varit att representera olika riktningar med heltalsvärden. Då kan vi skriva metoder som tolkar dessa heltal och rör sig i efterfrågad riktning.

   Problemet blir att det inte är lika lättläst, och använder sig inte av polymorfism. Vår lösning använder sig av ett Direction-objekt och tittar på dess förändring i planet, men den alternativa lösningen hade behövt tolka siffror och kan enkelt ge upphov till flera olika metoder i värsta fall. Det blir också väldigt mycket tydligare att skriva NORTH än till exempel “1” för att påvisa en förflyttning.

5. Varje Location innehåller en del barriers, och dessa kan vara av olika typer. Vi valde då att skapa en BarrierHandler som innehåller alla olika implementationer av AbstractBarrier. Man kan då lägga in barriärer i dessa objekt, vilket gör dem enklare att hantera. En BarrierHandler har då en metod openBarrier som går igenom alla barriärer i den och försöker öppna dem.

   Ett alternativ hade varit att skapa listor för alla olika typer av barriärer som kan finnas på en plats. Men problem kan uppstå om man ska ha flera typer av barriärer, inte bara behöver vi då skapa fler listor med barriärer, utan också fler metoder som ska försöka öppna dessa barriärer i listorna. Det blir snabbt väldigt omständigt och är inte särskilt modulärt.

6. Under planeringen av projektet visste vi att det enda som skulle användas i form av grafik är olika textkomponenter. Vi hade hört mycket om JavaFX och hur det skulle vara modernare och enklare att använda än Swing, vilket lät väldigt lockande eftersom projektets fokus inte låg på grafik utan objektorientering.

   Alla klasser i vårt gui-paket är därför underklasser till JavaFX-biblioteket, men är våra egna implementationer av dessa klasser.
   Att använda Swing var såklart ett alternativ, men det är gammalt och att bara rita någonting på skärmen är lite svårare än i JavaFX. Dessutom vet vi att FX är mycket mer populärt i modern programmering och ville därför lära oss grunderna.

7. En lokal poänglista var något vi ville ha för att alla spelare skulle kunna tävla mot varandra. Vi tog inspiration från tetrisprojektet och valde att göra en HighscoreList i form av en singleton. När den skapas hittar den en lokal fil som innehåller alla historiska Highscore och lägger till dem i listan med hjälp av hjälpklassen IOUtilities metod loadHighscoreList. Objektet sparas sedan ner till filen igen när spelet är slut.

   En annan lösning skulle kunna ha varit en klass med endast statiska metoder, och att man använder dessa för att hämta och spara en HighscoreList till en lokal fil, utan att initiera ett objekt.

   Praktiskt taget är det nog inte en jättestor skillnad på dessa val, kanske är till och med den statiska versionen mer minneseffektiv eftersom den inte behöver spara ett objekt, men skillnaden bör vara försumbar.
   Vi valde dock den mer objektorienterade versionen, eftersom det kändes lämpligare i den här kursen.

8. Någonstans behöver vi ta reda på om vi ska fortsätta köra programmet eller om spelaren redan har vunnit. Detta löste vi genom att i TextInputComponent alltid titta om spelaren har vunnit efter att denne skickat ett meddelande. I så fall sparar vi ner Highscore till HighscoreList och stänger programmet.

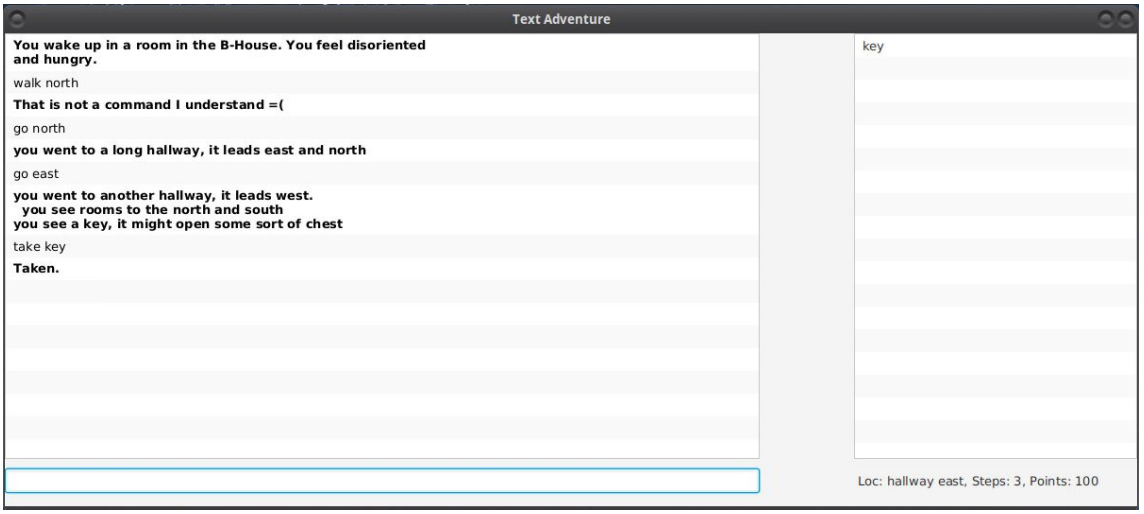
   En annan lösning hade varit att skicka vidare allting till Game som då kan avgöra om spelet är klart eller inte.
   Rent objektorienteringsmässigt är det inte snyggt att ha en grafisk komponent som utför spelets logik, och förmodligen hade den bästa lösningen varit att låta Game utföra denna logik istället.

7. Användarmanual

7.1 - Introduktion

Välkommen till detta textäventyr! Storyn som ligger just nu är att du är i B-Huset och ska hitta till en kista och ta upp en lapp. För att göra detta måste du ta dig omkring och hitta saker som kan hjälpa dig på din väg.

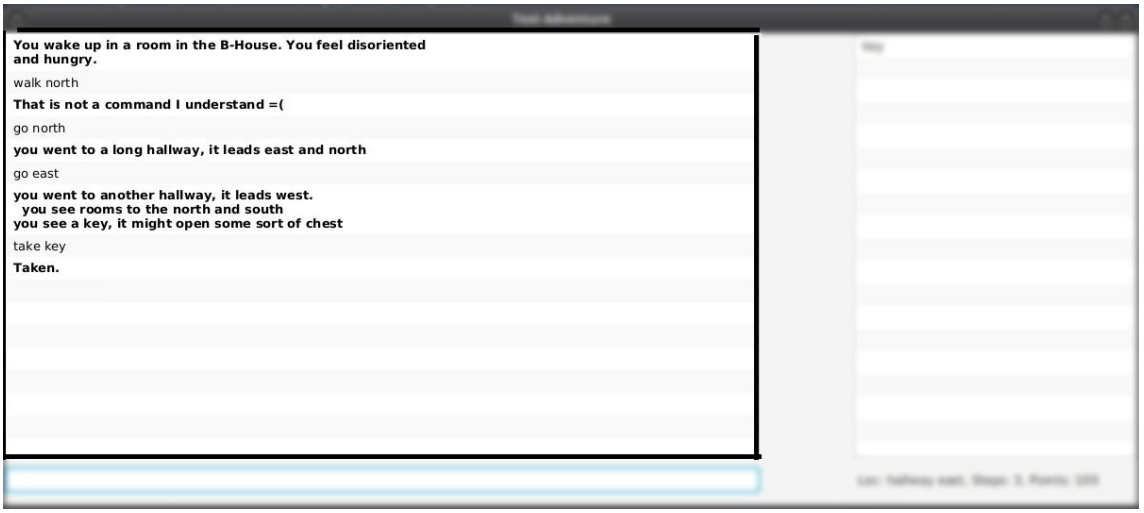
7.2 - Spelarfönstret



6. Spelarfönstret med alla dess komponenter

Ovan (bild 6) kan ni se spelfönstret, det består av mindre komponenter som vi går igenom här nedan.

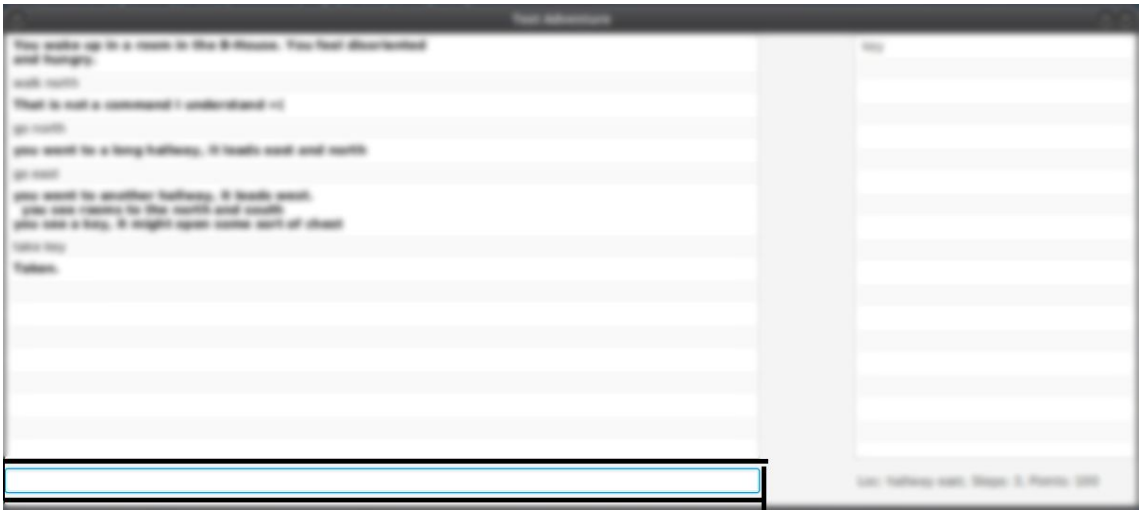
7.3 - Dialogfönster



7. Dialogfönstret

Dialogfönstret är var spelets och dina meddelanden dyker upp (se bild 7). Spelets meddelanden är dom som är fetstilta. När du skriver kommandon måste du tänka på att spelet klarar av två ord i taget. Mer om detta i Help Command delen.

7.4 - Inmatningsfönster



8. Inmatningsfönster

Kort och gott! I inmatningsfönstret (se bild 8) skriver du in texten som du vill att spelet ska bearbeta. Påminnelse att spelet endast kommandon i form av två ord. Det som skrivs kommer hamna uppe i dialogfönstret.

7.5 - Informationsfönster

9. Informationsfönstret

Informationsfönstret visar tre saker (Se bild 9).

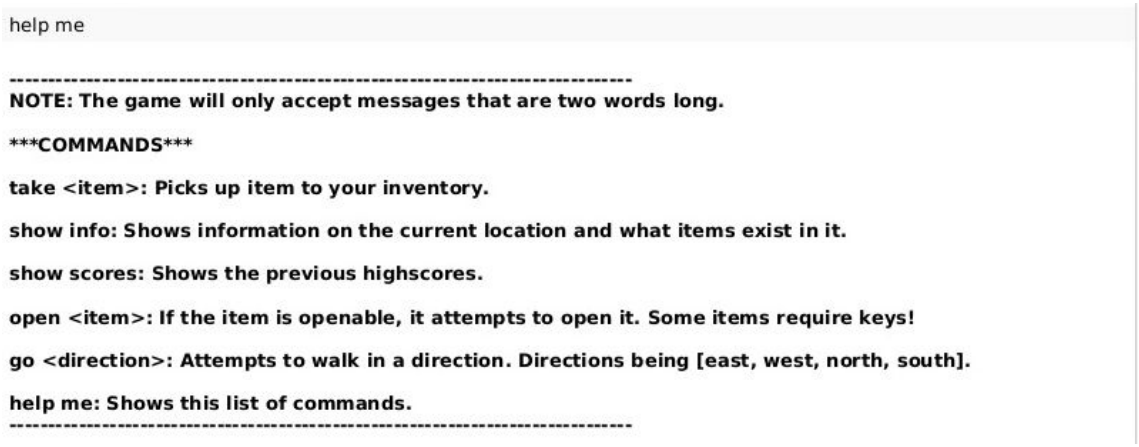
- Location - Var i spelet du befinner dig just nu
- Steps - Hur många steg du har tagit
- Points - Hur många poäng du har samlat på dig under spelets gång

7.6 - Inventoryfönster

10. Inventoryfönstret

Inventoryfönstret säger vilka items som befinner sig i din inventory för tillfället (se bild 10).

7.7 - Hjälpkommando



11. Demonstration av hjälpkommandot som finns tillgängligt i spelet

Hjälpkommandot bör användas under spelets gång om man är fast, man når det genom att skriva "help me" (se bild 11).

8. Slutgiltiga betygsambitioner

Vi siktar på betyg 5

9. Utvärdering och erfarenheter

- ***Vad gick bra? Mindre bra?***
    - När vi väl sågs och diskuterade designval och kodade så gick det väldigt bra och arbetet flöt på bra, men när det kom till att arbeta hemma så sattes inte lika mycket arbete ner då vi inte var lika självgående. Det var inte samma mängd arbete som gick ner i projektet, Robert har satt mer tid i början av projektet. Det var först i slutet som Payam hade möjlighet ta tag i det hela och satte ner all sin tid.

- ***Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken? Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har "hjälpt" bäst? Vi vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!***
    - Vi har gått på föreläsningar, gått på handledda labbar, nästan konstant stått på hjälplistan, "Vanliga problem"-sidan ,och såklart letat mycket online. Om det är något som kan bli bättre är att det ska finnas fler labbassistenter eftersom vad vi förstått av att prata med några av dom är att det är för många folk som behöver hjälp och ofta hinner dom inte med.

- ***Har ni lagt ned för mycket/lite tid?***
    - Det skulle vara kul att sätta ner mer tid, vi har många fler roliga idéer som skulle kunna förgylla spelet som vi har angett i milstolparna. Med det sagt är vi båda ganska nöjda med hur långt vi har kommit, men som sagt kan det alltid bli bättre.

- ***Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?***
    - Arbetsfördelningen var inte jämn, Robert jobbade flitigt genom hela projektet både på plats och hemma, medan Payam hade mycket att göra eftersom DÖMD hamnade under projektets gång. Väl efter DÖMD lade Payam ner all sin sin tid på projektet. Det som kunde ha varit bättre är om vi jobbade mer kvällar innan DÖMD så Payam kunde vara med mer, då det var svårare att jobba hemifrån. Payam anser också att han borde ha haft disciplin och jobbat mer på projektet innan DÖMD så att han inte behövde lägga all sin tid i slutet.

- ***Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?***
    - Projektbeskrivningen har varit nyttig för oss då vi hela tiden kollade på den för att se nästa steg. Det var väldigt användbart att skriva ner milstolparna då man hela tiden visste vad nästa steg var, även om man inte kunde följa den steg för steg, ibland eftersom saker behövdes implementeras i olika ordning.

- ***Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?***
    - Vi kom överens om att sitta och jobba kvällar tillsammans, men det blev så att man jobbade hemifrån, och det var inte jag så bra på //Payam

- ***Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.***
    - För Payam har det varit att hitta tid då det var mycket annat som gjorde att projektet bortprioriterades i början.

- ***Vilka tips skulle ni vilja ge till studenter i nästa års kurs?***
    - Att inte underskatta tiden det tar och sitta kvällar i skolan tillsammans, dom vanliga prokrastineringstipsen egentligen. Se också till att sitta tillsammans i början så ni får en bra grund i det hela.
    - Ha fokus på objektorienteringen och inte låta annat komma i vägen, gör de andra delarna enkla så man får med så mycket objektorientering som möjligt

    - Läsa igenom alla rubriker på slutrapporten för att se vad som man kan ta med i spelet, så som Inkapsling, Subtypspolymorfism och annat som förväntas ska vara med.

- ***Har ni saknat något i kursen som hade underlättat projektet?***
    - Kanske gå vidare från swing till FX?

- ***Har ni saknat något i kursen som hade underlättat er egen inlärning?***
    - Vissa av föreläsningarna kan bli väldigt utdragna och komplexa, många som jag pratade med kände att dom inte fick så himla mycket från dom.