POP 11g

Ida Helene Jensen (xdr622)
Julian Philippe Pedersen (rsk975)
Kjartan Martin Johannesen (vnb683)

22. december 2020

1 Indledning

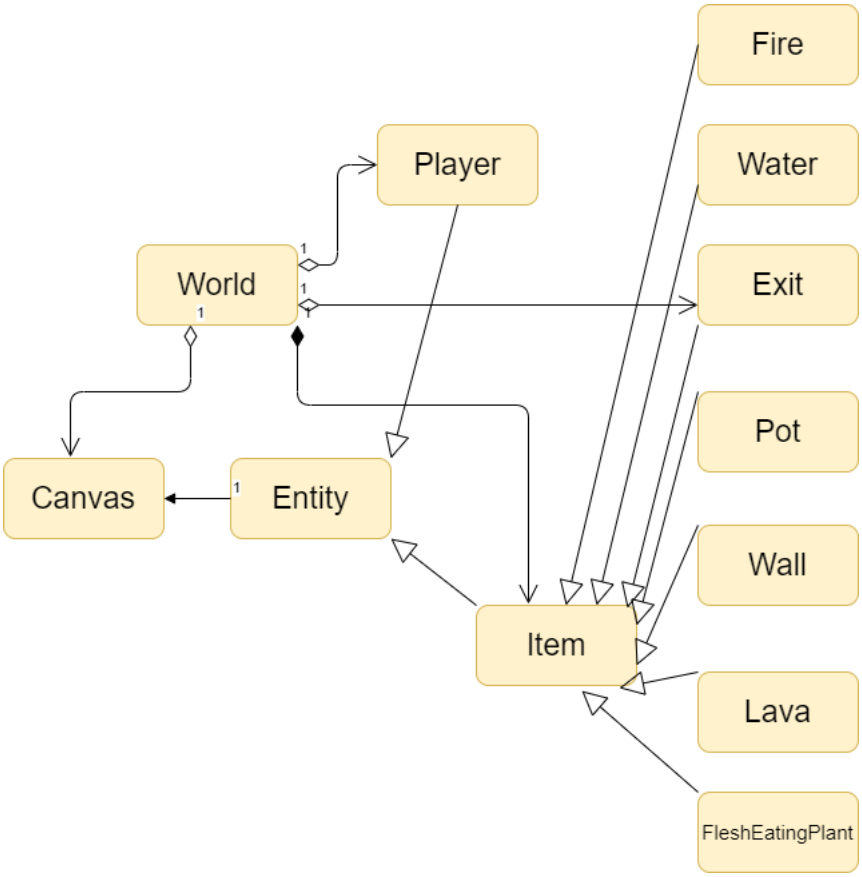
I denne opgave har vi forsøgt os med at lave et rogue-like game i F#. Rogue-like er en spilgenre som afstammer fra tekst rollespil som Dungeons and Dragons. De er lavet af tile grafik, hvilket betyder at man har et map som internt er repræsenteret ved en masse tiles. Ydermere foregå spillet tit i en form for dungeon hvor spilleren kæmper mod verdenen for at undslippe. Derudover er spilgenren også karakteriseret ved perma-death, hvilket betyder at hvis spilleren dør skal de starte helt forfra. Med disse definitioner i mente gav vi os til at prøve selv at lave et rogue-like spil. Vores spil endte med at følge den opsatte procedure, hvor vores tapre helt, @, prøver at undslippe både FleshEatingPlant og ild for i sidste ende at komme ud af den sorte undergrundsverden de er havnet i. Vi gennemgår i denne rapport vores proces fra ide til realitet, og vi vil vise både UML-diagram lavet over klasserne i spillet, samt fremhæve vigtige kodestykker.

2 Problemanalyse og Design

Vi startede projektet ud med at skrive alle de klasser som er givet i opgaven ned og tænke over i hvilken rækkefælge ting skal implementeres og hvilke klasser de nedarver fra hver. Derudover startede vi også med at vælge de to udvidelselser til spillet, så vi allerede fra starten kunne indtænke dem i vores design. Vi fandt frem til at vi først og fremmest skulle have en *Canvas class* som skulle holde styr på alle *Items* og *Player* selv, kort sagt at hvad der sker på skærmen. For at gøre det muligt at skabe levels i forskellige størrelser, mente vi at vores *Canvas* klasse skulle tage nogle dimensioner som

kan skabe firkanter i forskellige størrelser. Vi valgte dog kun at gøre det muligt at gøre vores *Canvas* til forksellige størrelser firkanter, istedet for at åbne for mulighederne for at lave trekantet eller sekskantet levels. Dette er gjort fordi underlige kanter ville gøre implementationen af funktioner som FullyOccupy meget mere bøvlet, da ikke alle felter ville være små kvadratter, det ville også ødelægge ideen om at lave felter, som er det der styrer hvordan *Player* flytter sig og hvordan ting spawner ind. I *Canvas* skal der være mulighed for at sætte både x og y koordinat, samt foregrunds og baggrunds farve og hvilken typeenhed en ting skal repræsenteres ved. Udover muligheden for at sætte de fem overstående variable skal *Canvas* også kunne vise dem i terminalen.
Som subclass til *Canvas* har vi brug for en *World class*. Den klasse har til formål at vise verden som spilleren spiller i mens spilleren stadig er i live, altså *World* er hvad spilleren ser og interagere med. Den skal ligesom *Canvas* kunne skabe en firkantet verden som spilleren kan spille i, derfor skal den tage en bredde og en højde. *World* skal ikke kun vise levelet men også sørge for at hente spillerens træk, finde ud af hvilke *Items* der er brugt og tjekke for spillerens død.
Derudover har vi brug for en abtrakt klasse som har til formål at vise alt hvad der skal være på *Canvas*. Denne klasse er kaldt *Entity* og er den klasse der sørger for hvad der skal renders på *Canvas*. Klassen skal derfor tage de samme fem variable som man kunne sætte i *Canvas* og skrive dem til *Canvas*, så de kan blive vist. her ville de give mening at en del af *Entity* laver et kald til *Canvas*. Da *Entity* er en abstrakt klasse betyder det at flere andre klasser skal arve dens egenskaber, disse klaser er *Player* klassen og den abstrakte klasse *item*. Item har 7 subclasses, nemlig *Wall, Fire, Water, FleshEatingPlant, Exit, Lava* og *Pot*. Hvor de to sidste er de udvidelser vi valgte fra starten.
Da vi skulle udtænke hvordan hver subclass til *Item* skulle implementeres startede vi med først at finde ud af hvad de skulle indeholde. den første klasse *Wall* skal udfylde et helt felt, men skal som sådan ikke interagere med spilleren, en vægs formål er kun at blokere for spilleren.
Klassen *Fire* har to egenskaber, den skader spilleren med 1 i damage hver gang spilleren går ind i et felt med ild, dog aktiveres ilden kun fem gange. Derfor har den en counter der tæller hvor mange gange ilden har givet spilleren skade, så når der er gået fem interaktioner så forsvinder ilden igen. Ydermere fylder ilden ikke hele feltet ud modsat væggen.
Klassen *Water* skal heller ikke fylde et felt helt ud, derudover skal vand helbrede to liv.
Klassen *FleshEatingPlant* giver spilleren 5 damage hver gang spilleren kommer i nærheden af dens felt, da *FleshEatingPlant* selv skal udfylde et helt felt. Dette gør at spilleren ikke kan gå igennem planten ligesom man ikke kan gå igennem væggen.

Klassen *Exit* er en lukket dør som først bliver åbnet når spilleren er ved siden af den og spilleren har mindst 5 liv tilbage. Så Exit skal gå fra at være helt fyldt ud på et felt til at forsvinde.
Klassen *Pot* var en af de valgte udvidelser og er krukker som spilleren kan smadre og så sker der en effekt. Vi valgte at Pots skulle have et i health, dermed smadres den første gang spilleren interagere med den. Derudover skal krukken forsvinde når den er smadret. Når en spiller smadre en krukke får spilleren 2 health. Krukken smider dermed en form for potion.
Klassen *Lava* er ligesom vand, men er meget farligt og skader spilleren meget. Lavaen er statisk og bevæger sig ikke, men den forsvinder heller ikke igen. Den er en af de valgte udvidelser.
Fælles for alle de beskrevne klasser er at de skal kunne vises på *Canvas*, at de har en boolskværdi der afgør om de optager et helt felt og at de skal kunne interagere med *Player* klassen, for at kunne lave om på hvordan items interagere med spilleren syntes vi det gav mening at der var en InteractWith funktion i den abstrakte klasse *Item*, som senere kunne overrides i hver subklasse, så at ild fx kunne skade mens vand kunne heale. Begge ting bruger InteractWith med spilleren, dog på hver sin måde. For at vide hver på *Canvas* et item skal renders skal hver item subclass sin egne initialiseringskoordinater. Vi mente at denne metode gør det lettere at placere items i levelet og holde styr på hvor der er items.



Figur 1: UML diagram af roguelike spillet

3 Programbeskrivelse

Canvas og Entity Den første ting vi valgte at implementere var *Canvas* klassen. Den er bygget sådan at den tager nogle dimensioner som argument, højde og bredde, som bestemmer hvor stort et level skal være. *Canvas* har fire members, *Set, Show, SetOnOccupied* og *Coord*. *Set* er den member som *Entity* senere referer til, og er altså en funktion der kan går at alt der skal renders bliver sat korrekt. Set tager 5 argumenter nemlig x-koordinat, y-koordinat, typeenhed, foregrundsfarve og baggrundsfarve. Derefter bruges vores variable *field* til at finde de to koordinater og indsætte typeenheden, samt farverne på denne plads. *field* virker sådanne at der skabes en 2D array med dimensioner givet ved *Canvas* som starter med at sætte alle felter til hvide punktummer på en sort baggrund. Disse punktummer bliver senere erstattet af andre typer af enheder og farver når verden fyldes op med items.

Et andet meget vigtigt member i *Canvas* er *Show* som er den funktion der sørger for at noget bliver vist i konsollen, så brugeren kan se hvad der foregår. Funktionen er bygget op sådan at den starter med at cleare konsollen, så brugeren ikke skal se hver træk de har lavet, men istedet får en mere flydende overgang mellem hver træk. Show er bygget op af to nested for-loops som har til formål både at holde styr på rækker og koloner. I enden af loopet bliver typeenheden og farverne sat til det felt hvori de tilhøre, så altså de koordinater hvor de skal ses i konsollen. Derudover bliver typeenheden printet i konsollen med *printf*.
Klassen *Entity* tager de samme 5 argumenter som blev gennemgået i *Canvas.Set*, som initialiseringsværdier. Disse værdier bliver propeties indeni *Entity*, så de kan bruges i det abstrakte member *RenderOn*. *RenderOn* tager et canvas og vores default implementation af *RenderOn* bruger *Canvas.Set* på den givet canvas. Det er altså denne funktion der sørger for at alle klasse der nedarver fra *Entity* kan renders på banen. Klasserne der faktisk nedarver fra *Entity* er *Player* og *Items*, dog nedarver 7 andre klasser fra *Items*, så de er dermed indirekte nedarvet fra *Entity*. Alle de ovenstående klaser der nedarver fra *Entity* bruger den defaulte version af *this.RenderOn*, medmindre andet er specifikeret. *Entity* er derfor en form for bindeled mellem alt der skal vises i konsollen og *Canvas*.
Items og subclasses Som beskrevet tidligere skulle den abstrakte klasse Item kunne interagere med player, dette er gjort med den abstrakte funktion InteractWith, som i vores implementation tager en player. Denne funktion bliver overridet i hver klasse der inherit fra Item, da fx vand interagere anerledes med player end ild gør. Som eksempel kan man se på hvordan InteractWith i Item, Water og Fire klassen:

```
type Item (...) =
    inherit Entity (...)
    abstract member InteractWith : Player -> unit

type Water (...) =
    inherit Item (...)
    override this.InteractWith(player: Player) = player.Heal(2)

type Fire (...) =
    inherit Item (...)
    override this.InteractWith(p: Player) = p.Damage(1)
```

Det abstract member gør ikke noget i sig selv, men når det bruges nede i hver klasse kan man let bestemme hvordan den givne klasse skal interagere med spilleren.

Player og io Vi valgte at dele spilleren op i to, den ene er *Player* klassen, som holder styr på health og hvor spilleren skal gå hen, mens den anden er i er en del af *World* klassen i *member this.Play*, der sørger for at tage imod piletasteklik fra spilleren. io-delen er lavet eksternt for at holde brugerens input og alt kodens indmad adskilt. io-funktionen virker sådan at vi matcher *System.Console.ReadKey().Key* på følgende måde:

```
match System.Console.ReadKey().Key with
| System.ConsoleKey.UpArrow -> p.MoveTo(-1,0)
| System.ConsoleKey.DownArrow -> p.MoveTo(1,0)
| System.ConsoleKey.LeftArrow -> p.MoveTo(0,-1)
| System.ConsoleKey.RightArrow -> p.MoveTo(0,1)
| System.ConsoleKey.Q -> p.Damage(10)
| _ -> p.MoveTo(0,0)
```

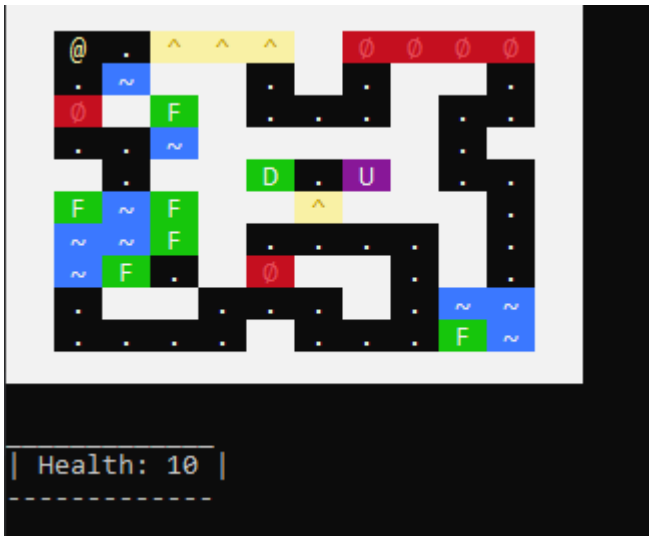
Vi matcher altså brugerens input på seks følgende måder, nemlig de fire piletaster, Q som er en måde at dræbe sin karakter på og restarte og til sidst wildcardet, som bare gør at hvis brugeren trykker på enhver anden knap sker der intet med karakteren. Koordinaterne som ses ved de fire piletaster sendes til *p.MoveTo* som er inde i *Player* klassen. Player klassen tager sig derimod af at kalkulere hvad hver piletasttryk betyder, samt spillerens helbred.
World *World* klassen har 3 bemærkelsesværdige members nemlig *this.BufferCanvas(canvas:Canvas), this.DoInteractWith()* og *this.Play()*. *this.BufferCanvas* tager et canvas, og bruger en andet member kaldet *this.AddItem (item:Item)* som appender et givent item fra *Item* subklasserne til en liste. Dette gøres for lettere at kunne holde styr på hvilke items der er i brug. Det som *this.BufferCanvas* gør er at gå igennem hver item i itemlisten ved brug at et for-loop og render det til canvas, ved brug af funktionen *RenderOn*. *this.DoInteractWith* tager *dpos* som er en tuple ints. Den vigtige del af *DoInteractWith* er at sørge for at lave en *collideList*, der tjekker om et felt har et item med *FullyOccupy* eller ej. Hvis et felt er fyldt af en item med *FullyOccupy = true* betyder det at spilleren ikke kan gå derover, hvis *FullyOccupy = false* så kan spilleren lave sit træk og gå over i feltet. Derudover sørger *DoInteractWith* også at hver items i listen der skal interagere med spilleren faktisk går det. Det betyder at hvis spilleren er tæt nok på vil member *InteractWith* fra *Item* klassen blive brugt.
Som beskrevet ovenfor tager member this.Play sig af brugeren input til Player. Udover input er det også dette member der opretter alle items i spillet og holder styr på hvad der sker renders, buffers og sendes ud i konsollen når spilleren ikke er død. Nedenstående kode demostrere hvordan this.Play ser ud:

```
member this.Play() =
    while not player.IsDead do
        _level.ResetField()
        this.BufferCanvas (_level)
        player.RenderOn(_level)
        _level.Show()
        _level.DisplayMessage()
        _level.SetMessage("")
        this.IO()
    System.Console.ReadKey()
    System.Console.ResetColor()
```

Play() sørger dermed for at genstarte felterne så fx spilleren position bliver opdateret og den tidligere postion bliver slettet. Derudover viser den hvad spilleren kan se og sender brugeren nogle beskeder i konsollen når noget bemærkelsesværdigt sker. Herunder hvis man fx bliver ramt af ilden.

4 Afprøvning

Repræsentationen af hver item i spillet er vidst på nedenstående billede

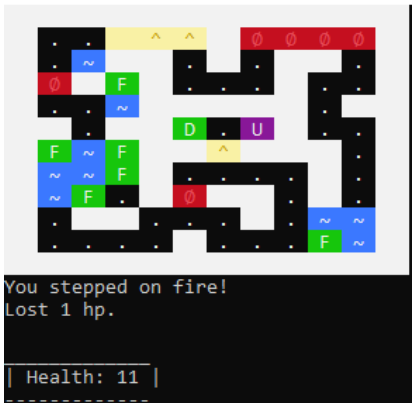


Figur 2: Repræsentation

væg er repræsenteret ved et fyldt hvidt felt, ild er en gul baggrund med et mørkegult ∧, vand er en blå baggrund med et hvidt ∼, lava er rødt Ø med mørkerød baggrund, FleshEatingPlant har en grøn baggrund med et hvidt

F, krukke er et hvidt U med mørk lilla baggrund, udgang er mørkegrøn baggrund med hvidt D og spiller er et gult @ med sort baggrund.
Den nedenstående figur giver et eksempel på hvordan displayMessage virker, her er spilleren enten gået ind i noget vand eller ild

(a) DisplayMessage heal



(b) DisplayMessage damage

Figur 3: Showing how healing and damage works

Når spilleren går ind på samme felt som ilden kommer der en besked i bunden der fortæller brugeren at de har trådt i ilden og hvor meget liv de har mistet. Det samme sker hvis man har trådt i vand udover at man her får en besked om at man har healet.
Der er to måder spillet kan ende på enten ved et win som set i første subfigur eller et gameover som set i den anden subfigur

(a) Win-screen

(b) Gameover

Figur 4: Showing win and loss

5 Konklusion

Vi har lavet et rogue-like spil i F# til det har vi brugt object-orienteret programmering. Vi har lavet klasser der håndtere hver element i spillet, som bliv gennemgået i vores afsnit om programbeskrivelse og design. Her samlet vi afsnittet op med et UML-diagram. Her kunne man se hvordan klasser relateret sig til hinanden og hvordan de var relateret, fx var der mange klasser der nedarvet fra *Items*. Vi gennemgik derefter hvordan vi konkret havde implementeret de forskellige dele konkret i programmet. Vi gennemgik fx hvordan vores player virkede og hvordan man styre spilleren gennem *World* klassen. Til sidst viste vi billeder fra spillet i konsollen og demonstreret hvordan forskellige scenarier udspiller sig.