NTNU
Det skapende universitet

Arkitektur for lokasjonsbevisste sosiale spill

Johan Gunnar Gaustadsæthe Isaksen
Kristian Gaupseth Waagsbø

Master i informatikk
Oppgaven levert: Juli 2010
Hovedveileder: Hallvard Trætteberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Sammendrag

Modelldrevet utvikling er generelt brukt på større informasjonssystemer, men denne oppgaven omhandler bruk av modelldrevet utvikling på spill. Ved å benytte nyere teknologi som lar deg benytte grafiske verktøy til å modellere og bruke disse modellene i stedet for kode, begynner det å bli interessant å se på modellbasert utvikling innen nye bruksområder.

Oppgaven er laget for PlayTrd-prosjektet som er opprettet og ledet av Hallvard Trætteberg, introduserer en spilltype som kalles pervasive gaming. Pervasive gaming er kort og godt myntet på sosiale spill som fremhever kommunikasjon og samhandling. Vi ønsket å utvikle en arkitektur som tilrettela for et bredt utvalg av spill av denne typen, samtidig som vi skulle finne et spillkonsept som vi kunne utvikle videre. Målsetningen med dette var å evaluere om modellbasert utvikling egnet seg som metodikk for å lage spill, samtidig som vi ønsket å teste i hvor stor grad verktøyene egnet seg til å drive denne type utvikling.

Verktøyene som tas i bruk i denne oppgaven er basert på Eclipseplattformen. Det vil si at Eclipse er utviklingsplattform som ligger i bakgrunn for all teknologi som benyttes. Eclipse Modeling Framework er rammeverket som tilbyr modelleringsverktøyene. Gjennom utnyttelse av Xtext er det opprettet et domenespesifikt språk som gir en forenklet syntaks for å kjøre logikk i modellene. Denne logikken konverteres til SCXML, en standardisert syntaks som Eclipse gjennom utvidelser kan eksekvere. Kombinasjonen av modeller og tilstandsmaskiner er funksjonaliteten som i utgangspunktet trengs for å drive en enkel prototype av et spill.

Gjennom oppgaven vil det komme frem at vi gjennom en todelt modell, en modell med generell funksjonalitet og en modell med særegen funksjonalitet, konkluderer med at modellering viste seg godt egnet for vår utvikling, samtidig som at verktøyene var både nyttige i bruk og til en viss grad brukervennlig. Med en viss grad mener vi at det er fremdeles mye rom til videreutvikling av implementasjonen som eksekverer den utviklede tilstandsmaskinnotasjonen, da denne inneholder en del små feil og vi anser den som vanskelig å feilsøke.

Med det arbeidet som har blitt utført det siste året, anser vi at nye oppgaver som startes opp i prosjektet i fremtiden nå har fått på plass et godt grunnleggende arbeid som kan danne basis for en enklere og raskere oppstart. Dermed tror vi også at det vil bli lettere å ta arbeidet noen steg videre og kanskje få testet prototyper i en ordentlig setting, med mobile enheter og personer løpende rundt i godt samarbeid i Trondheim sentrum.

Forord

Denne masteroppgaven ble utført siste året av masterstudiet i Informatikk ved Norges teknisk-naturvitenskapelig universitet (NTNU) i Trondheim, med retning systemarbeid og menneske-maskin interaksjon.

Motivasjonen vår til å velge denne oppgaven var blant annet at den gav oss mulighet til å jobbe to sammen i arbeidet og at vi ønsket å utvide kunnskapen vår innenfor emnet modelldrevet utvikling. Som så mange andre ved universitetet så er vi vant til mer programmeringssentriske løsninger på oppgaver av denne typen, og derfor syntes vi det var en spennende utfordring som gav oss muligheten til å jobbe med ny og spennende teknologi.

Vi vil takke vår veileder Hallvard Trætteberg for god faglig undervisning, veiledning og gjennomlesning denne perioden. På grunn av oppgavens natur har det vært noen utfordringer, og Hallvard har alltid vært tilgjengelig for hjelp da vi har stått fast. Under hele perioden har Hallvard jobbet med liknende oppgaver parallelt med oss, og alltid entusiastisk opplyst oss på møtene våre og bidratt med veiledning når vi stod fast med arbeidsoppgavene våre. I høstsemesteret hadde vi en del fellesmøter sammen med Trætteberg, Alf Inge Wang og doktorgradsstudentene som jobber med liknende prosjekter. Disse var motiverende, og gav oss anledning til å diskutere idéer i oppstartsfasen når vi jobbet med å definere spillkonseptet vårt.

Til slutt må vi også takke medstudenter og venner på ITV-263 (Sule) for samvær, lek og diskusjoner under arbeidsperioden. Uten dem hadde prosessen vært betraktelig mindre lystig. Motivasjonen til å bli ferdig har økt med den offentlige ”word count” tabellen felles for alle masterstudentene på salen.

Innholdsfortegnelse

Figurliste

Tabelliste

1 Innledning

Denne oppgaven tar for seg elementer innen temaet modellbasert utvikling. Modellbasert utvikling er i og for seg ikke noe nytt og uprøvd, men likevel er det elementer innen denne bestemte retningen som er spennende og også relativt nytt.

Først kan man nevne settingen som dette prosjektet ønsker å drive modellbasert utvikling innenfor. Spill er ikke det første man tenker på når man snakker om resultater for modelldrevet utvikling. En mer vanlig løsning er store systemer med mye organisasjonslogikk. Det å prøve ut slik teknologi i et spill er et av de sentrale temaene i denne oppgaven.

Videre kan man nevne at modelleringen som benyttes her er veldig sentral for hele utviklingen, fordi modellene man jobber med ikke nødvendigvis bare fungerer som en basis for fremtidig programmering. Modelleringen tilsvarer programmeringen med i en standard utviklingsmetode. Med det menes det at modellering kan utføre den samme jobben som programmering generelt bruker å benyttes til. Om man ønsker det kan kode genereres direkte av verktøyene rett i fra modellene.

Modelldrevet utvikling er en form for utvikling som kanskje ikke rangerer høyt hos veldig mange, spesielt blant programmerere. Årsaken er at utviklere ser på modeller som en tungvint eller kanskje også et unødvendig forberedende steg de føler ikke gir så mye utbytte i forhold til det endelige produktet. Med nyere teknologi viskes mye av gapet mellom modeller og ferdigprodusert kode ut. Med Eclipse Modeling Frameworks modelleringsverktøy Ecore, kan man generere ferdig kode direkte fra modellene. Dette fører nå til at mange kan revurdere sine synspunkter hva modellering angår.

Studenter ved NTNU som går enten informatikk eller datateknikk opplever introduksjonen til teknologien som programmeringssentrisk. Man lærer først å programmere og deretter aspekter innenfor modellering, slik som UML. Noe forhold til modelldrevet utvikling derimot er det svært få som får, da fag som involverer dette kommer sent i studiet og er mer avhengig av hvilke spesialiseringer man velger.

1.1 Problemstilling

Utgangspunktet til å finne problemstillingen var oppgavebeskrivelsen, som var publisert på websidene til IDI (IDI, 2010):

“PLAY=TRD is a platform for games using the city of Trondheim as the arena for game play. The platform is being developed at IDI in cooperation with Trådløse Trondheim, Telenor and

StudentbyEN. The task is to design and prototype a SOA-based architecture for PLAY=TRD, with special focus on location aware and social games."

Fornorsket sier oppgaveteksten at målet for oppgaven var å designe og prototype en SOA-basert arkitektur for PLAY=TRD med spesiell fokus på lokasjonsbevisste og sosiale spill.

Hovedmålet, slik vi nå har tolket oss frem til med denne oppgaven, er å prøve ut en modellbasert arkitektur for en spillmotor laget for bruk til lokasjonsbevisste spill. Videre skal vi finne styrker og svakheter ved denne. Egner modellbasert utvikling seg i våre øyne, eller er alternative utviklingsmetoder bedre?

Også innenfor vår problemstilling, har vi som oppgave å lage et spill som kan brukes til å demonstrere at teknologien fungerer.

Vårt prosjekt eksisterer i kontekst av et større prosjekt ledet av Hallvard Trætteberg. Derfor eksisterer det en del overlapping av teknologi og arbeid, der funksjonalitet som vi benytter oss av er utviklet av Trætteberg eller andre, og innspill fra oss også har hatt innflytelse på arbeidet til Trætteberg. Blant annet så er spillet vårt som presenteres i kapittel **Error! Reference source not found.** et spesifikt spill som blir flettet med en generell modell utviklet av Trætteberg. Denne generelle modellen tilbyr all funksjonalitet som er ansett som universell, det vil si funksjonalitet som er tiltenkt flere eller kanskje også alle andre spill som utvikles i PlayTrd-prosjekter.

Vår oppgave går ut på å ta i bruk teknikker som ikke tradisjonelt er brukt til å implementere lokasjonsbevisste spill, samtidig bedømme hvor enkelt det er å legge til nye spillideer i rammeverket ved å lage nye modeller. Arbeidet skal løses iterativt og resultere i prototyper som vi kan benytte til å validere elementer av arbeidet vårt.

For å oppsummere dette mer konkret ønsker vi gjennom vår oppgave å diskutere følgende forskningsspørsmål:

- Egner modellbasert utvikling seg til utvikling av spill?
- I hvor stor grad er verktøyene vi benytter gjennom utviklingen enkle i bruk og nyttig?

1.2 Disposisjon

Dette er en kort oversikt over kapitlene som følger i denne oppgaven, og en kort beskrivelse av innholdet i hvert kapittel.

2**. Spillgenre og modellbruk**
Introduserer konteksten til prosjektet, prosjektet PlayTrd og introduserer noe sentral teori angående spill

3**. Metode**
Presenterer modelldrevet utvikling og annen sentral teori.

4**. Teknologi**
Innblikk i teknologi som blir benyttet i utviklingen av prosjektet.

5**. Zombiespill**
Endrer fokuset i oppgaven fra teori til implementasjonen av et spill som vi har utviklet som et ledd i å få svar på forskningsspørsmålene introdusert i problemstillingen.

6**. Arkitektur**
Arkitektur i systemet. Presenterer både vårt system og elementer fra den generelle modellen utviklet av Hallvard Trætteberg, som er nært knyttet opp mot vårt system.

7**. Resultater**
Presenterer resultater som kom ut av modellene presentert i kapittelet foran, og tar for seg elementer i arbeidsprosessen.

8**. Diskusjon og fremtidig arbeid**
Diskusjon vedrørende resultatene våre og samarbeidet i tillegg til våre tanker angående fremtidig arbeid.

1.3 Akronymer

I oppgaven er det benyttet en del ord, uttrykk og forkortninger som vi ønsker å forklare her i dette underkapittelet.

Begrep:	Forklaring:
PlayTrd	Play Trondheim, se kapittel 2.2
EMF	Eclipse modeling framework
SCXML	State Chart Extensible Markup Language
MDE	Model driven engineering
MMORPG	Massively multiplayer online role-playing game
Wi-Fi	Wireless Fidelity
SOA	Service Oriented Architecture
GUI	Graphical User Interface
URI	Uniform Resource Identifier
DSL	Domain Specific Language (Domenespesifikt språk)

2 Spillgenre og modellbruk

Vår oppgave omhandler spill. Før vi går gjennom teknologi og implementasjonsdetaljer er det naturlig å ta en titt på spill sett fra vår kontekst. Dette kapittelet starter med en titt på Trådløse Trondheim og PlayTrd, et pågående prosjekt som oppgaven vår faller inn under. Deretter ser vi nærmere på nye trender som utvikler seg i dataspill i dag, der pervasive gaming er en ny sjanger som enda ikke har tatt helt av, men er i vekst. Vi avslutter kapittelet med å se nærmere på denne type spill inneholder, spillets struktur og hvorfor vi mener at spill egner seg til å bli modellert.

2.1 Trådløse Trondheim

I 2005 ble selskapet TrådIøse Trondheim (Trådløse Trondheim) startet som et forsknings- og utviklingsprosjekt ved NTNU. I 2006 gikk flere offentlige og industrielle aktører sammen for å realisere prosjektet og i september samme år ble trådløse Trondheim lansert. Store deler av midtbyen, selve sentrum der folk ferdes, har nå trådløst nett tilgjengelig for alle som vil benytte seg av denne tjenesten.

Trådløse Trondheim har gitt muligheten til og utforske lokasjonsbevisste spill i dette området ved bruk av Wi-Fi. Hovedfokuset til Trådløse Trondheim ligger på utendørsdekning, den geografiske avgrensningen er stor og målet er 99,5 % oppetid. Dette sørger for ingen begrensninger når det kommer til å utforske og utvikle ideer som PlayTrd (se neste underkapittel) i Trondheim ved bruk av trådløst nett.

2.2 PlayTrd

PlayTrd er et konsept utviklet av Hallvard Trætteberg og Alf Inge Wang ved institutt for datateknikk og informasjonsvitenskap ved NTNU. Konseptet går ut på at nye studenter som kommer til Trondheim, skal lære seg byen og kjenne ved hjelp av teknologi. Det er i kontekst av denne teknologien vi arbeider i denne oppgaven. Ideen går ut på at man er sin egen spillkarakter i en pervasive verden (se kapittel 2.3) i Trondheim. Spillkarakteren starter på bar bakke, med ingen kunnskap om Trondheim som by. Spillerens nivå, attributter og anlegg for å manipulere verden rundt seg øker gradvis med erfaringen på lik linje med en karakter i et rollespill, og i løpet av det første semesteret vil man kunne ha nok data til å si hvor kjent en aktiv spiller er i byen. Hvis spilleren har spesifikke interesser som kunst, sport, historie og lignende kan man også lage dimensjoner for å vite spillerens kunnskap og erfaring i disse. Dette konseptet er blitt gjort mulig av et annet prosjekt kalt Trådløse Trondheim som har sørget for at hele Trondheim sentrum har trådløst nett. En tenkt aktivitet kan være en quiz runde, hvor spillerne deler seg i ulike grupper og forflytter seg etter hvert som spørsmålene de svarer på leder de fram til neste destinasjon. Destinasjonene de har vært på lagres, og

problemene de må løse for å komme videre dukker umiddelbart opp ved ankomst. Dette er et eksempel på et spill eller en lek som kan iverksettes av PlayTrd. For en ny, muligens ung student på et nytt sted kan dette være en måte å leke seg frem til ny kunnskap ved hjelp av teknologi.

2.3 Pervasive gaming

Med dataalderen har dataspill inntatt en mer dominerende rolle for underholdning i samfunnet. Før dataalderen var spill designet og spilt ute i den fysiske verden med bruk av egenskaper fra den virkelige verden, slikt som fysiske objekter, vår sans for rom og romlige relasjoner. (MagerKurth, Cheok, Mandryk L., & Nilsen, 2005) Tradisjonelt har utendørsspill og brettspill dekket denne type underholdning, og med dagens teknologi kan man endre de tradisjonelle utendørsspillene til å ta i bruk elementer som ikke eksisterer i den virkelige verden, men likevel virtuelt er der og kan brukes i spillet.

Fram mot internettalderen var det eneste som eksisterte en-spiller, det var du og kun du som var den store helten mot omgivelsene. Medspillerene var utelukkende datakontrollerte objekter. Forutsetningene er helt annerledes i dag. Spillbransjen er verdens største, og i størsteparten av spill som kommer ut til konsoll eller PC er en-spiller kun et aspekt av spillet. Fokuset ligger i kommunikasjon og samhandling med andre spillere over internet, enten mot omgivelsene eller mot andre spillere. Selv om det i dag kommer ut spill som har en-spiller som det eneste alternativet, kan man se på hvilke typer spill som klarer å vekke størst interesse og holde på spillerne; Spill som har kommunikasjon og samhandling med andre spillere som et nøkkelkonsept har blitt regelen og ikke unntaket. Massively multiplayer online role-playing game (Wikipedia, 2010) er en sjanger som bruker disse nøkkelkonseptene og har den største aktive spillerbase pr. i dag. MMORPG-spill har høy informasjonskompleksitet og disse spillene kan godt sammenlignes med et større informasjonssystem. En relativ ny sjanger innen dataspill er treningsspill. Målet er å bedre fysikken eller rett og slett bruke kroppen som styringsmekanisme i spillet. I dag kan man spille på ulike eksterne instrumenter og i tillegg synge og se hvor nøyaktig og korrekt trommeslagene, guitarstrengene eller tonen i stemmen er i forhold til sangen. Man kan også danse på eksterne dansematter, balansere eller hoppe på Wii Fit. (Wikipedia Wii Fit, 2010) Det kan virke som om det ikke finnes begrensninger på hva man kan gjøre om til et spill.

Litt naivt kan man kombinere fysisk bevegelse med kommunikasjon og samhandling og man har pervasive gaming. Pervasive gaming er en ung sjanger som bygger videre på dagens trend om å inkludere mer virkelighet og sosial interaksjon i dataspill. Dataspill fanger spilleren med illusjonen om å være fordypet i en virtuell verden (Amory, Naicker, & Vincent, 1999), som er designet med et optimalt nivå av informasjonskompleksitet og mer interaksjon enn tradisjonelle spill, (Malone, 1981) og tanken er at man ikke nødvendigvis trenger å befinne seg stasjonert foran en skjerm for å oppleve disse positive egenskapene.

Med dagens håndholdte mobile dataenheter i nettverk kan man ta det et steg lenger og kombinere datamaskiners regnekraft og grafikk med de fysiske og sosiale aspektene til den virkelige verden.

2.3.1 Lokasjonsbevisste spill

Det finnes opptil flere undersjangere innen pervasive gaming. En populær tilnærming er å ta i bruk hele verden, arkitekturen vi lever i, som et spillbrett. En hel bygning, en blokk, eller en hel by blir spillebrettet og menneskelige spillere blir proaktive og høyt uforutsigbare spillebrikker. (MagerKurth, Cheok, Mandryk L., & Nilsen, 2005) De fysiske begrensningene i rommet i spillverden blir vegger, tak og gulv i den virkelige verden, og sosial interaksjon mellom mennesker øye til øye og ikke gjennom avatarer på skjermen.

Lokasjonsbevisste spill må vite hvor spillerne befinner seg i rommet, spillene bruker posisjonering av spillere og spillobjekter som en essensiell del under kjøring. Det er en rekke lokaliseringsteknikker tilgjengelig, de vanligste og mest kjente er; GPS satellitt signal, WiFi, GSM signalstyrke eller nærhetslyttende teknologier som RFID. Det er fordeler og ulemper med alle, både i forhold til nøyaktighet og implementering. GPS er uten tvil den beste teknologien for å bestemme posisjon til en spiller i sanntid. Hvis applikasjonen krever det kan man på serversiden følge bevegelsene til alle spillerne. Den største ulempen med GPS er at det kreves at alle spillerne har maskinvare med GPS satellitt signal. I tillegg er teknologien vanskelig å håndtere hvis deler av spillet foregår innendørs, da signalene er svake eller ikke-eksisterende innendørs. Hvis et antall studenter som er nye i byen skal spille et sosialt spill hvor målet er å bli kjent med byen, vil det være en fordel om man kan delta selv om man ikke har en enhet med GPS-teknologi. Det er tidligere nevnt at PlayTrd er et pågående prosjekt med en fremtidsrettet visjon, og med dagens raske utvikling er trenden at satelittnavigering blir stadig mer utbredt. Det uavhengige analysefirmaet Berg Insight anslår at det i 2009 er totalt solgt 150 millioner enheter med GPS/WCDMA teknologi, mot 78 millioner i 2008. Estimert tall for 2014 er 770 millioner (Berg Insight, 2010). Nyeste generasjonen av smartphones kommer ofte med GPS-teknologi. GSM-triangulering er et mulig alternativ, men gir ikke nøyaktigheten påkrevd for å implementere for eksempel kartet i zombiespillet nevnt i kapittel 5.2. GSM-lokalisering har en beste nøyaktighet på 50 meter i urbane strøk med høy basestasjonfrekvens. (Wikipedia GSM Mobile Phone Tracking)

Oppgaven vi skal løse avhenger ikke av å velge riktig posisjoneringsteknologi. Det er ikke totalt irrelevant, men i forhold til å finne ut om vår arkitektur er brukbar til lokasjonsbaserte spill er dette noe vi ikke trenger å ta stilling til i denne oppgaven. Hendelser generert av spillere i vårt spill kan likegodt være generert av en ”Wizard of Oz” (Sharp, Rogers, & Preece, 2007) til vårt formål.

2.3.2 Kommunikasjon

Pervasive spillapplikasjoner representerer distribuerte omgivelser og kommunikasjon mellom deltakerne og sentrale komponenter er essensielt. For eksempel når en handling utføres av en spiller som endrer tilstanden til spillet, må dette bli overført til resten av spillerne. ”In urban environments WiFi hotspots are one method of transferring such data, with cell phone based services such as GPRS or 3G (UMTS) providing alternative options” (Broll, Ohlenburg, Lindt, Herbst, & Braun, 2006). I Trondheim sentrum er WiFi en metode for å sende slik data. Som tidligere nevnt i kapittel 2.1 om Trådløse Trondheim, er fordelen at omtrent hele midtbyen er trådløs.

Pervasive gaming og kommunikasjon har noen problemområder man må være kjent med. I artikkelen ”Meeting Technology Challenges of Pervasive Augmented Reality Games” tas flere viktige problemer opp; Kommunikasjonssignal som feiler eller ikke er tilgjengelig i visse områder, kompabilitet og deltakere som slutter seg til eller går under spill. Om signalbrudd og utilgjengelighet er det: ”…very important that all game components can handle such disconnections and do not rely on continuous connections with all the participants.” (Broll, Ohlenburg, Lindt, Herbst, & Braun, 2006) Dette og problemet med deltakere som går inn og ut av spillverdenen er problemer vi mener vi å ha en indirekte løsning på i den tilstandsbaserte arkitekturen i PlayTrd som finnes i kapittel 6 ved å tilrettelegge for lagring av tilstander direkte i datamodellene. Rammeverket lagrer tilstanden til hver deltaker i en tilstandsmaskin, og i maskinen er det et begrenset antall transisjoner til andre tilstander, altså begrenset antall valg deltakeren kan gjøre gitt sin nåværende tilstand. Problemet med at deltakere kan avslutte å spille spontant eksisterer i alle flerspillerspill, men dette er ekstra sårbart i en pervasive verden. ”Such spontaneous disconnections are much more common in pervasive games than in traditional games and a game server must handle this issue.” (Broll, Ohlenburg, Lindt, Herbst, & Braun, 2006)

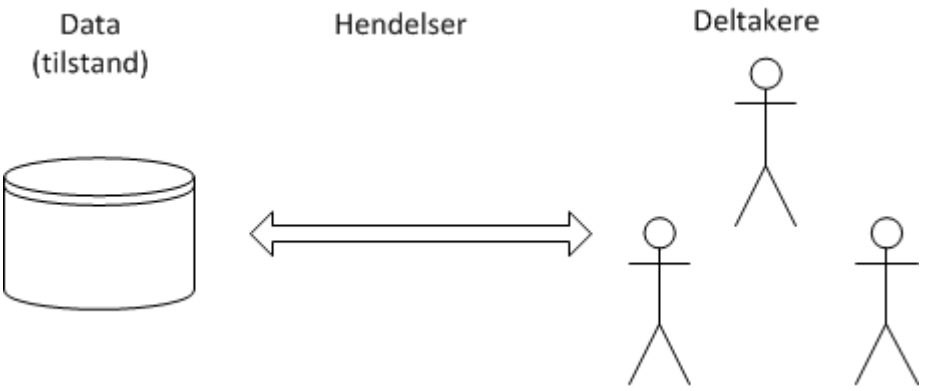
2.4 Argumentasjon for modellbruk

Med vår modellerfaring er dette et prosjekt som krever at vi blir kjent med en ny metode, samt at mye av arbeidet ikke direkte vil føre til konkrete resultater, men gi oss ny innsikt i konseptene vi arbeider med. Dette er med andre ord ingen oppgave for fossefallsmetoden (Hawryszkiewycz, 2001), men heller en slags blanding av modelldrevet utvikling og prototyping. Gode rammeverk for modellbasert utvikling av spillteknologi eksisterer omtrent ikke. Sammenlignet med andre utviklingsmetoder med fokus på fremgangsmåter som involverer programmering, har disse fra starten av rammeverk å forholde seg til som gjør jobben lettere. Vårt utgangspunkt er at vi enda ikke har identifisert alle grunnleggende konsepter, samtidig som disse må abstraheres ned i hierarkiet. Dermed må omfattende domenemodellering til og det kan sies at vi bidrar til å lage et rammeverk for modellbasert utvikling av spill.

2.4.1 Spillets tilstand

PlayTrd som et rammeverk for pervasive spill inneholder en rekke elementer som kan minne om et datarollespill. På et lavt nivå kan vi si at den store forskjellen imellom de er at pervasive gaming utspiller seg i den virkelige verden med sin egen fysiske kropp som karakter, mens i et MMORPG bruker en spesiallaget avatar i en fiktiv verden. Flere spillelementer brukes i begge verdener; Karakterer og evner, lokasjoner, oppdrag og ting. (penger, verktøy, våpen, osv.) Disse elementene kjennetegnes i alle spill hvor deltakeren trer inn i en rolle i en annen verden, enten det være seg et dataspill som World of Warcraft (Wikipedia WOW), et pervasive spill som zombies vs survivors (se kapittel 5.2), eller en terningbasert imaginær spillomgivelse som GURPS (Wikipedia GURPS, 2010). GURPS-rammeverket har et rigid oppsett av regler og lover som omfatter alt fra hvilke perks, quirks, evner og utstyr en spiller kan ha, til hva som er lovlig og mulig å utføre. Skal man utvikle spill som har en slik omfattende verden hvor valgene og mulighetene er mange, må man gjøre grundig forarbeid for å spesifisere domenet. Med et omfattende rammeverk som PlayTrd er ikke denne spesifiseringen noe man gjør en gang for så å kalle seg ferdig.

Hvis man tenker seg en kjøring av et pervasive spill eller hvilket som helst rollespill med mange deltakere, inkludert attributter til spillere og omgivelser, er dette til mye informasjon som må lagres. I tillegg kommer hendelsene som endrer på dataene. Hvis vi ser på systemet som en svart boks er spillets tilstand i en instans er definert av dataene som gir informasjon om karakterene og verden. Hendelser som endrer spillets tilstand er aksjoner fra deltakerne og dette endrer seg over tid. Figuren under illustrerer dette.



Figur 1 Spillets tilstand

Med tanke på definisjonen av spillets tilstand over, hvor tilstanden til en hver tid er gitt av dataen, må systemet reagere på og oppdatere data kontinuerlig under kjøring. Den raske tilstandsendringen fører til krav om et reaktivt system.

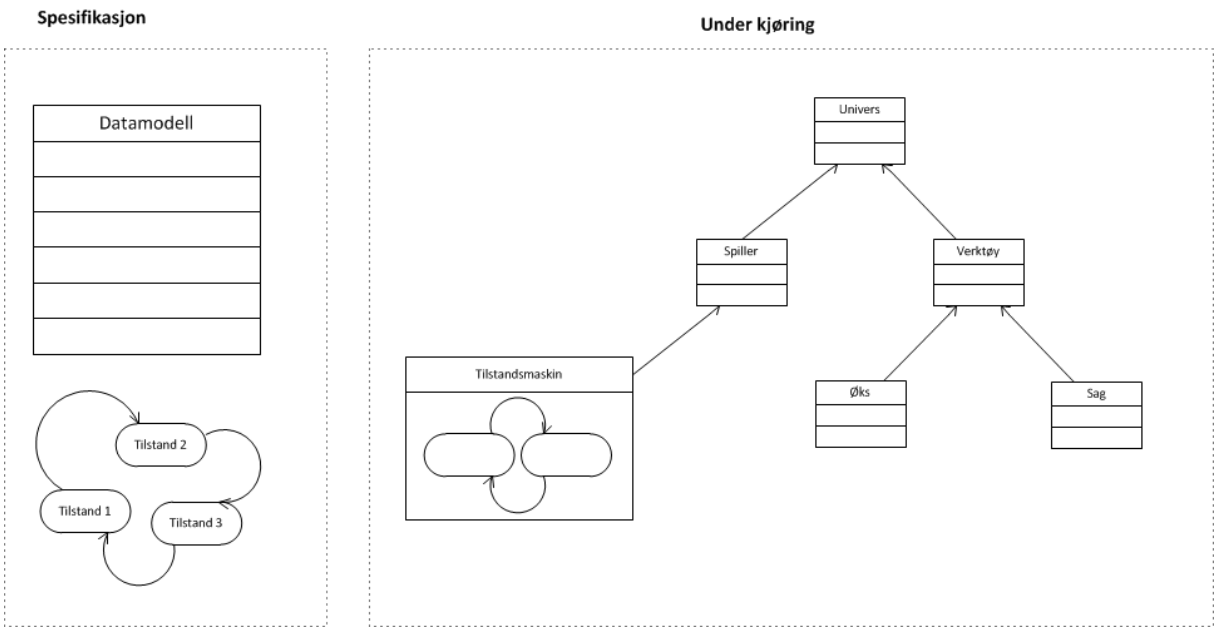
2.4.2 Tilstandsmaskiner

Modeller er godt egnet for å abstrahere konsepter og klassifisere de hierarkisk. Gitt domenet for PlayTrd er dette godt egnet å spesifisere i modeller. I kapittel 3.1 diskuteres modellbasert utvikling og de største fordelene er gjenbruk og automatisk kodegenerering. Hvis man kunne bruke domenemodeller aktivt i implementasjonen vil man spare mye tid som ellers ville blitt brukt på programmering. Videre kan spilldefinisjonen i kapittel 2.4.1 fortelle om mulige løsninger på en slik fremgangsmåte. Hvis vi summerer tilstander, data og krav om et reaktivt system, ligger det i kortene at en løsning vil være å bruke tilstandsmaskiner som motor for å endre dataverdier. Endelige tilstandsmaskiner er reaktive av natur og det finnes mange variasjoner på definisjoner av tilstandsmaskiner. Felles for dem alle er tilstander, hendelser og transisjoner til andre tilstander. En kombinasjon av tilstandsmaskiner og modeller som datarepresentasjon, hvor tilstandsmaskinene er koblet opp mot og endrer datamodeller er oppsettet vi i denne oppgaven skal prøve ut på en pervasive spillimplementasjon.
Problemene med pervasive spill nevnt i kapittel 2.3.2 er de største utfordringene spillere som blir frakoblet eller forlater spillet underveis. Dette oppsettet er robust og enkelt nok til å håndtere gjenopptaking av spill ved en eventuell frakobling. Man trenger bare datamodellen til spilleren og vite i hvilken tilstand spilleren var i da han sluttet å spille.

2.4.3 Oppførselen til systemet

Når vi nå har definert en mulig løsning på hvordan dette kan løses i praksis, har vi laget en illustrasjon som viser tankegangen. Figuren under viser en mer spesifisert utgave av Figur 1 hvor tilstandsmaskiner er koblet til og oppdaterer datamodellen. Spesifikasjonen til venstre viser datamodellen og en tilstandsmaskin under design-tid, mens den høyre delen av figuren viser modellen instansiert med tilstandsmaskin som kommuniserer med objektene.

Tilstandsmaskinen har muligheten til å aksessere alle verdier i domenet gjennom hierarkiet av objekter.



Figur 2 Data og tilstandsmaskiner

Denne type modellering setter krav til presisjon og riktig formalisering. Siden modellene blir delt med tilstandsmaskiner må de være formalisert slik at de klarer å kommunisere sammen og vi har muligheten til å trekke konklusjoner rundt denne tilstandsbaserte arkitekturen. For å trekke våre konklusjoner må vi ha noe å modellere. I kapittel 5.2 har vi definert et spill som kan realiseres i PlayTrd, og mye av arbeidet vårt har gått med til presisere modellen og validere dens oppsett og innhold. Det er ikke selve spillet som legges vekt på i denne oppgaven, men vi måtte ha noe å forholde oss til og bruke som et verktøy for å kunne svare på om denne metoden egner seg til spillutvikling.

3 Metode

Etter å ha sett nærmere på spillteori i forrige kapittel, er neste steg å introdusere teori angående metodikk. Dette kapittelet har som hensikt å drøfte teorien som ligger bak utviklingsmetodikkene som vi bruker i gjennomføringen av prosjektet. Kapittelet starter med en innføring om modellbasert utvikling og drøfter også prototyping, før det avsluttes med tanker rundt hvordan prosjektet ville blitt løst hvis man skulle brukt litt mer tradisjonelle utviklingsmetodikker.

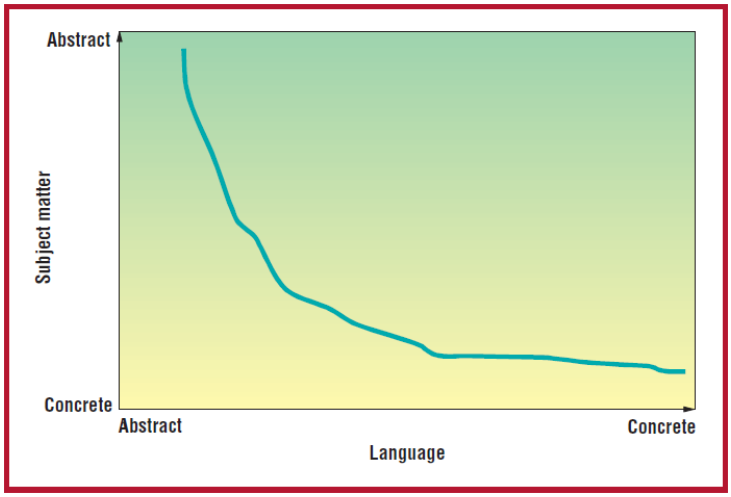
3.1 Modellbasert utvikling

Noe av temaet i denne oppgaven er å se på verdien av å basere seg på modeller i utviklingssammenheng, og da spesielt til vårt formål. Modeller og modelleringsteknikker kan ha flere formål og betydninger. Den tradisjonelle måten er å bruke modeller i designfasen av et prosjekt og senere som dokumentasjon. Model-driven engineering (MDE) er en relativt ny disiplin innen programvareutvikling og naiv definisjon på dette kan være ”..simply the notion that we can construct a model of a system that we then can transform into the real thing.” (Mellor, Clark, & Futagami, 2003) Nøkkelordet er transform, MDE er ikke utbredt men har likevel i teorien mange store fordeler hvis man velger å gå for en MDE-tilnærming. I dette subkapittelet går vi inn på hvordan modeller har blitt brukt historisk, hva det vil si å bruke MDE og noen viktige karakteristikker for å oppnå suksess med modeller.

Modeller har historisk hatt en sekundær rolle innen denne ingeniørvitenskapen. Motstanden mot modeller baserer seg på at de er mer til hinder enn hjelp. Modeller er et grensesnitt mellom utviklere, og blir raskt utdatert i forhold til hvordan systemet utvikler seg. Det blir vanskelig å se formålet med å endre modellen i takt med koden for å dokumentere systemet. Analytikeren har én forretningshensikt, programmereren en annen, og modellene fungerer som rådgivende skjemaer i startfasen på prosjektet.

Med en modell-drevet tilnærming må man endre fokus til modellene og kvaliteten på disse. I Pragmatics of Model-driven Development (Selic, 2003) hevdes det at:”…software developments primary focus and products are models rather than computer programs”. Den samme artikkelen mener at det er på tide at vi begynner å utnytte modellers fulle potensiale, og sammenligner et eventuelt paradigmeskifte, fra kode-sentrisk til modell-drevet, med historiske fremskritt som damp og elektrisk energi. ”In an industry that prides itself on its rapid advances, this apparent reluctance to move forward despite an obvious need might seem surprising.” (Selic, 2003) For at en modell skal være brukbar og effektiv må den møte visse kriterier, de viktigste er abstraksjon, forståbarhet og nøyaktighet. For å unngå forvirring og få ut mest relevant informasjon i en modell, må man gjemme eller fjerne urelevante detaljer, spesielt hvis modellen skal transformeres og være en del av systemet. Model-driven development (Mellor, Clark, & Futagami, 2003) relaterer brukbarheten av

modeller i en slik kontekst i relasjon med abstraksjonsnivået til modelleringsspråket: “Two dimensions: The language's abstraction level and the degree of abstraction for the subject matter under study. The subject matter axis has an inverted scale which leads to a neat curve with analysis models at the top and design models lower down”. Se Figur 3.



Figur 3: Modelleringspråkets abstraksjonsnivå

Start med et abstrakt problem i et abstrakt språk, og end opp med en konkret fremstilling av løsningen i et konkret lav-nivå språk. Egen erfaring med modeller i utviklingsprosjekter tilsier at abstraksjon og nøyaktighet blir nedprioritert til fordel for å ”komme i gang”. De fungerer som en startplattform hovedsakelig for å gi en felles forståelse av de konsepter man skal lage. Etter kort tid inn i implementeringsfasen har det allerede blitt produsert komponenter som ikke eksisterer i et diagram, men som man likevel ”måtte ha med”. Det er denne arbeidsmåten og tankegangen MDE distanserer seg i fra, modellene er ikke til mye hjelp når de ikke er formelt koblet til designavgjørelsene som gjøres i implementeringen. Fordelene med MDE består ikke bare av det faktum at koding blir betraktelig enklere, men som Mellor, Clark og Futugami hevder i (Mellor, Clark, & Futagami, 2003): ”…enables reuse at the domain level, increases quality as models are successively improved, reduces costs by using an automated process, and increases software solutions’ longevity.” For å oppnå disse fordelene kan man ikke bare se på abstraksjon, men modellen må også ha høy grad av forståbarhet og presentere informasjonen som ligger der på en måte som direkte appellerer til vår intuisjon. ”One reason why programs are not particularly expressive, even when based on languages that support sophisticated abstractions, is that they require too much detailed parsing of text to be properly understood” (Selic, 2003). En siste avgjørende karakteristikk er nøyaktighet, modellen må sørge for en virkelighetsnær representasjon av det modellerte systemets interesseegenskaper. Med MDE er nøyaktighet garantert, for modellen blir omsider systemet det modellerer. Det er mange grunner til at MDE bør tas seriøst,

utviklingen er konstant og mange verktøy tillater å bruke MDE i flere applikasjonsområder for å generere applikasjonskode automatisk direkte fra modeller.

3.2 Modeller i vår oppgave

Vår oppgave tar utgangspunkt i modelltransformasjon, den opprinnelige modellen er en del i den endelige løsningen. EMF-rammeverket gir muligheten til å lage en Ecore-modell hurtig gjennom en drag-and-drop diagram-editor, og transformere den til en datamodell med grafisk grensesnitt for å sette inn verdier. Datamodellen blir videre transformert til et tilstandsbasert kjørbart spill.

I PlayTrd-rammeverket er det en generell modell som inneholder hva vi ser på som universelle konsepter og abstraksjoner, denne blir også kalt domenemodell. Se 6.2 Den generelle modellen for bedre beskrivelse. Modellen har gjennomgått flere endringer og utvidelser mens vi har arbeidet med den. Den er nå utvidet til å inneholde konsepter for de fleste tenkte spill i en pervasive verden i trondheim sentrum, men det er ikke dette som er det essensielle; Skulle det dukke opp et behov for å utvide modellen ved for at man for eksempel mangler et konsept for et spill i domenet, kan dette gjøres hurtig uten å ha innvirkning på tidligere spill eller ideer. Alternativt kan man subklasse det man trenger i en spesifikk modell. En annen fordel er at med en generell gjenbrukbar modell får man automatisk kontinuerlig kvalitetsøkning etter hvert som man arbeider med den. I fremtiden vil modellen gjennomgå flere tilnærmet konsekvensfrie utbedringer. Når vi har arbeidet med å integrere domenemodellen med en abstrakt spillmodell har vi oppdaget at konsepter som er nødvendige for spillet kan flyttes inn i domenemodellen. Oppdagelser som dette gjør at domenemodellen blir mer informasjonsrik og nye spillmodeller blir enklere å lage med tanke på at konsepter allerede eksisterer.

3.3 Prototyping

Meningen med en prototype er å kunne utføre såkalte billige tester av arbeid i progress. Med billig mener vi her at prototypen kan være et forsøk på å skape en representasjon av et ferdig produkt på en enklest mulig måte, med den hensikten å teste funksjonalitet på tenkte sluttbrukere. Hensikten er ikke å vise frem et fullført produkt, men å få innspill og bekreftelser på at produktet har utviklet seg i riktig retning.

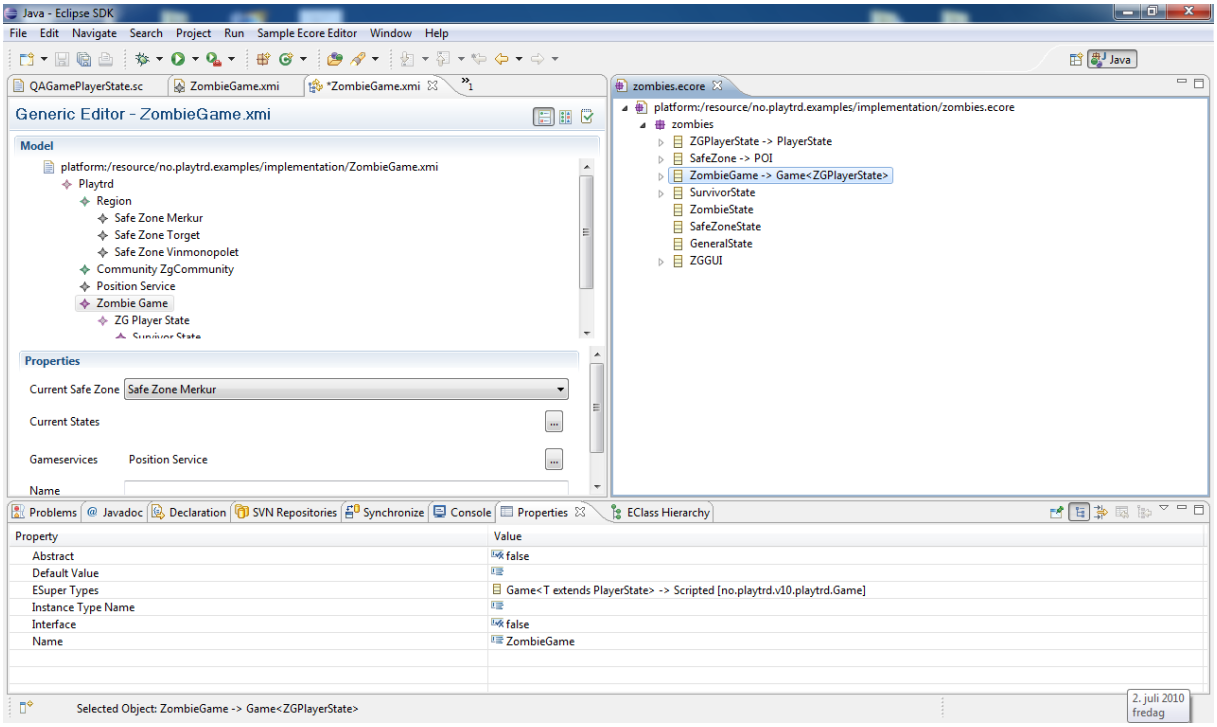
Prototyping kan også gjerne brukes i flere omganger, hvor resultatet av en tids arbeid blir en prototype som man bruker som grunnlag i å planlegge videre arbeid. I dette tilfellet er prototypene tettere knyttet mot resultatene av utviklingen, og vil derfor også bety noe mindre arbeid spesifikt knyttet til å lage prototypen, samt at prototypen blir noe mer realistisk ut ifra hva man kan forvente av det endelige produktet.

4 Teknologi

Etter å ha introdusert både spillteori og utviklingsmetodikk, er det nå på tide å se nærmere på teknologien som kan drive et prosjekt av denne typen. Gjennom implementasjon og testing av dette prosjektet blir det brukt en del teknologi. Utviklingsmiljø, standarder for modellering og tilstandsmaskiner samt en del annet. Dette kapittelet presenterer Eclipseplattformen som er essensiell fordi den er utviklingsmiljøet som ligger i bakgrunn for alt som blir utviklet i dette prosjektet. Det er Eclipse som tilbyr modelleringsverktøy, som kjører spillinstanser i utviklingsfasen og som tilbyr funksjonalitet for å utvikle og eksekvere tilstandsmaskiner. I dette kapittelet ser vi på Eclipse og teknologien som er sentral for prosjektet, deriblant Xtext og SCXML. Til slutt i kapittelet ser vi markedet som teknologien vår er aktuell for og hva slags teknologi som er sentral på disse plattformene i dag.

Den teknologiske plattformen vi benytter oss av og som blir presentert i dette kapittelet var definert som et krav når vi påtok oss oppgaven. Det vil si at vi ikke stod fritt til å velge Eclipseplattformen selv. Dette er fordi Hallvard Trætteberg allerede hadde startet prosjektet PlayTrd når vi kom inn, og det ville vært unaturlig for oss ikke å benytte oss av samme teknologi som prosjektet ellers nyttiggjør seg av. Vi kjenner til at det har foregått en prosess hva seleksjon av teknologi angår, og at for eksempel Business Process Model Notation (OMG, 2010), oftest forkortet til BPMN, har vært testet i stedet for Ecore diagrammer for å modell og logikk, med det resultatet at BPMN ble ansett som ikke egnet for dette prosjektet.

4.1 Eclipse



Figur 4: Utvikling av modeller i Eclipse

Eclipseplattformen er for de aller fleste et utviklingsmiljø kjent for utvikling av Java-applikasjoner. Verktøyet er basert på åpen kildekode og tilbyr en åpen og utvidbar plattform for å kjøre og utvikle prosjekter, ikke bare i Java. Gjennom utvidelser kan man utvide Eclipse sin funksjonalitet langt utover de rammene som kommer gjennom den utgaven som blir offisielt utgitt. Eclipse er aktivt utviklet, gjennom årlige nye utgaver av plattformen.

En komponent i Eclipse blir kalt en plugin eller en utvidelse på norsk. Plattformen i seg selv og verktøy som utvider funksjonaliteten er sammensatt at utvidelser. Enkle verktøy kan bestå av en utvidelse, mens mer avanserte verktøy kan bestå av mange utvidelser. Den sentrale tankegangen med slike utvidelser er at funksjonalitet i dem kan benyttes av brukere eller gjenbrukt og videre utvidet av andre utvidelser (Steinberg, Budinsky, Paternostro, & Merks, 2009). Eksempler på utvidelser som er sentrale for dette prosjektet er: Xtext, EMF og SCXML. Xtext blir benyttet for å konvertere et scriptespråk med elementer fra Javascript til et XML basert scriptespråk som SCXML håndterer og kjører. EMF sees nærmere på i neste seksjon.

4.2 Eclipse Modeling Framework (EMF)

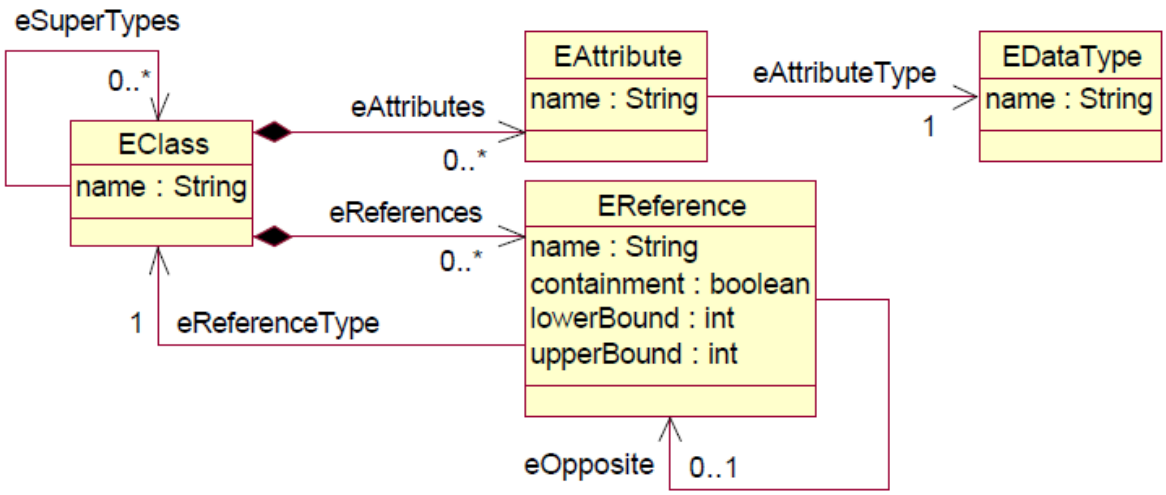
EMF prosjektet er et modelleringsrammeverk for å bygge applikasjoner basert på en strukturert datamodell. EMF kan hjelpe brukere til enkelt å konvertere modeller til javakode som er både korrekt og enkel å endre. Samtidig kan det påstås at EMF hjelper med å selge

modellering som konsept til de som er mer skeptiske til anvendeligheten til modeller ved å tilrettelegge for enkel konvertering mellom modeller og kode.

Hvis man sammenlikner EMF med Unified Modeling Language (UML), som er en ofte anvendt standard i vanlig modellering, tilbyr EMF bare et mindre subsett av funksjonaliteten som UML tilbyr. EMF inneholder en metamodell, kalt Ecore, for å beskrive modeller og kunne eksekvere dem. Deler av denne modellen kan sees i Figur 5. Den sentrale funksjonaliteten som trengs er å kunne beskrive klasser og relasjoner mellom disse. Dette er kjernefunksjonaliteten i Ecore og i programvare generelt. Relasjoner involverer igjen sentrale begreper som hierarki, underklassing og tilhørighet (containment). Hierarki benytter vi for å kunne gjenbruke elementer av modellen uten å måtte bringe inn alle elementer fra alle modeller samtidig, ved å bruke en generell modell som inneholder kjernefunksjonalitet i forhold til hva man ser for seg spill som utvikles ønsker å tilby. Underklassing bruker vi for å enklere nyansere små forskjeller i klasser som til en viss grad likner litt på hverandre. Containment styrer klassenes levetid og tilhørighet. En instans av en klasse er eid av en annen klasse, som da styrer hvordan vi kan "kommunisere" med denne instansen og når instansen skal forsvinne. En instans vil da gjerne forsvinne når eierklassen forsvinner.

En annen styrke er at modellene kan genereres på flere måter. Modellene lagres i et format kalt XMI (XML Metadata Interchange), som man kan generere fra både eksterne verktøy, direkte gjennom å skrive XML kode eller gjennom interne verktøy som kan generere XMI direkte fra for eksempel javakode. Med andre ord finnes det måter å generere modeller for flere forskjellige typer utviklere.

I Figur 5 ser man et utsnitt av funksjonalitet som er sentral i EMF. Med hjelp av de konseptene klasser, attributter og relasjoner kan man raskt og enkelt lage modeller som er lettforståelige men som samtidig også gir det utgangspunktet som er nødvendig for Eclipse til å jobbe videre med modellen.



Figur 5: Sentrale elementer i EMF

4.3 Xtext

Xtext er et rammeverk som kan brukes til å lage såkalte domenespesifikke språk (DSL). DSL er språk designet for å løse spesifikke problemer, for eksempel kan man designe et språk for å enkelt utvikle en tilstandsmaskin gjennom programmeringssentrisk syntaks, se kapittel 4.5, med hensikt i å konvertere innholdet til en standard støttet av Eclipse gjennom andre utvidelser for å kjøre tilstandsmaskiner. Xtext gir tilgang til å utvikle DSL i Eclipse og målsetningen for prosjektet er at det skal være en svært rask og effektiv prosess å lage nye eller utvide eksisterende språk (The Eclipse Foundation, 2010).

4.4 SCXML

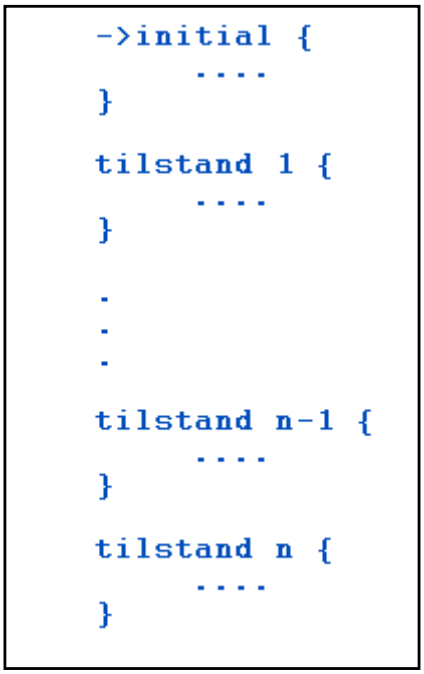
SCXML står for State Chart XML. Standarden er publisert gjennom World Wide Web Consortium (W3C). Standarden angir et XML-basert språk for å definere tilstandsmaskiner. Med Commons SCXML underprosjektet finnes det en implementasjon med mål å kunne utvikle og eksekvere tilstandsmaskiner definert med SCXML-språket i et javabasert miljø (Apache Commons, 2010).

4.5 Tilstandsmaskiner

Xtext er et rammeverk for å lage programmeringsspråk og domenespesifikke språk. Hallvard Trætteberg har laget et tilstandsmaskinspråk som er tett koblet opp mot dynamiske instanser i XMI-form. For å gi en innføring i språket skal vi først kort forklare grunnleggende oppsett med overganger, betingelser og syntaks, senere i kapittel 4.6 går vi igjennom et enkelt spørsmål og svar-spill for å demonstrere språket i praksis.

4.5.1 Overganger

En tilstandsmaskin må alltid ha en starttilstand, en såkalt *initial state.* Fra starttilstanden og alle andre tilstander kan det være overganger til andre tilstander basert på valg og betingelser inne i tilstanden.



Figur 6: Struktur tilstandsmaskin

Overganger beskrives med tegnene -> (pil), og starttilstanden må alltid ha en pil foran navnet. Figur 6 viser strukturen til en tilstandsmaskin med n-tilstander, overganger trenger ikke skje sekvensielt som man kan få inntrykk av; I tilstand n kan det for eksempel være overgang til tilstand 1 basert på hva som står i de doble klammeparentesene, se eksempelet under.

```
->tilstand 1 on [[...]];
```

4.5.2 Statements

Språket er identisk Javascript for å bruke sammenligningsoperatorer og behandle matriser. Sekvensiell utførelse starter med ”do” etterfulgt av hva skal gjøre. Betingede valg eksisterer i form av if-setninger. Eksempelet under viser en tilstand med flere sammenligningsoperatorer og kontrollstrukturer. ”On enter” etterfulgt av kode er det som blir utført straks man entrer denne tilstanden.

```
tilstand-1 {

     on enter if [[ a > b && c != a]] do {{

          a = d[0];
          d[0] = b;

     }};

     ->tilstand-2 on [[d.size() == 5]];

}
```

Figur 7: Eksempeltilstand i scxmlxt

Uttrykket ”->tilstand-2 on ” sier at man skal bevege seg til tilstanden med navnet ”tilstand-2” om etterfølgende uttrykk valideres som sant.

4.6 Question-Answer Game

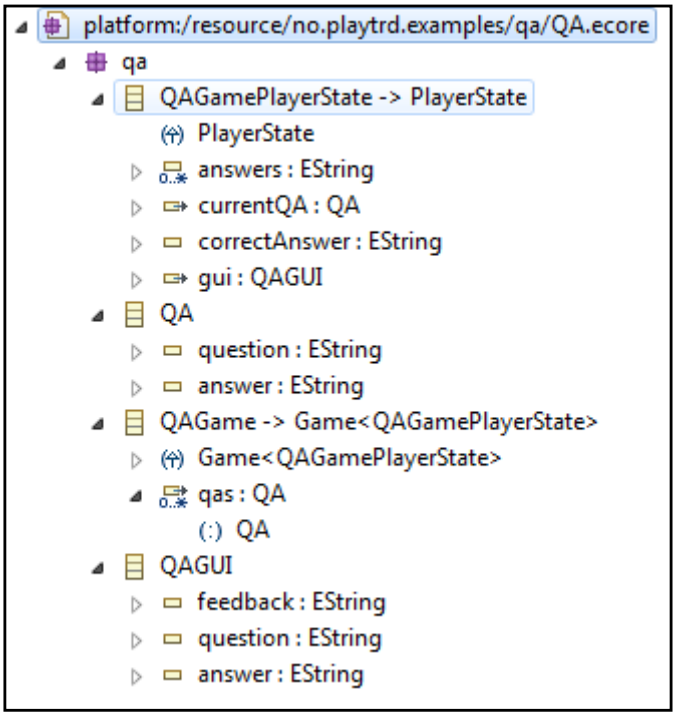
Question Answer Game er et elementært spørrespill hvor hver spiller alltid har et spørsmål de må svare på, og man får ikke et nytt før man har korrekt svar på gjeldende spørsmål. Man er ferdig når alle spørsmålene er besvart riktig.

Spørrespillet består av to tilstandsmaskiner; Selve spillmaskinen består av 3 tilstander og styrer hele spillet fra det blir initialisert til det er ferdig. Disse tre tilstandene er Init, Active og Finished, denne tilstandsmaskinen vil bli kalt hovedmaskinen. Den andre tilstandsmaskinen som vi kaller spørremaskinen, er tilstandene for hvert spørsmål og består av Idle, Ask og Correct. Scriptingen eller koden som trenges for å kjøre dette spillet er lite nok til å kunne presenteres i denne rapporten. For å få et overblikk over hva som skjer i koden må vi først se på hvordan Ecore-modellen til spillet ser ut.

4.6.1 Modell

Modellen inneholder all informasjon man trenger i et spørrespill. Dataen som er beskrevet i modellen under er alt som trengs for å representere enhver tilstand under alle tenkelige kjøringer av spillet. I denne underseksjonen går vi detaljert igjennom modellen til QA-spillet.

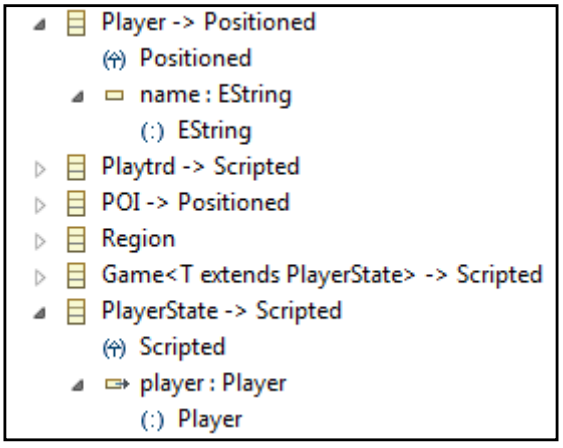
Hovedklassen her er QAGame som inneholder to objekter; Game arver fra klassen QAGamePlayerState, denne klassen ses øverst i modellen. Arrayet qas innholder objekter av typen QA, den andre klassen i treet. QA har to attributter av typen EString, et spørsmål og et korrekt svar for dette spørsmålet.



Figur 8: QA-Spill Ecore-modell

QAGUI har 3 attributter av typen EString, denne klassen eksisterer for å presentere spørsmålene til spilleren og gi muligheten til å svare. Dette blir demonstrert under når vi ser på spillet under kjøring.

QAGamePlayerState er den viktigste klassen her, klassen i seg selv er en subklasse av PlayerState. PlayerState eksisterer i den generelle modellen og det er her fordelene med klassehierarkiet og ressursoppbygningen kommer til syne. Bildet under viser et utsnitt av den generelle modellen, trestrukturen til klassene Player og Playerstate er på laveste nivå for å vise arv.



Figur 9: Klasser fra Ecore-modell

I PlayerState finner vi en referanse til Player som igjen arver fra Positioned. Positioned er en annen klasse i den generelle modellen og alle objekter som trenger å være posisjonert i et pervasive spill, mennesker, steder eller andre elementer, arver fra denne klassen.

QAGamePlayerState arver også indirekte fra denne ressursen og man slipper således å tenke på posisjoneringsteknologi for alle nye spill som modelleres. POI (Point Of Interest) fungerer på samme måte som man kan se i ecore-modellen over. Man kan velge hvilke tjenester og objekter man vil benytte seg av fra PlayTrd-rammeverket, med elementære roller og funksjonalitet allerede tilstede er tankegangen at arbeidet for å legge til nye ideer/spill redusert.

4.6.2 Tilstandsmaskin script

Tilstandsmaskinen er delen av arkitekturen som driver spillet fremover, den reagerer på endringer og oppdaterer data, alt etter forhåndspesifiserte regler etter tilstandsmaskinprinsipper. En mer detaljert forklaring finnes i 6.1. Her gjennomgåes hva som skjer i scriptkoden under en gjennomføring av spillet med en spiller. Vi begynner med hovedmaskinen vi nevnte først og ser på den første tilstanden i denne.

```
->initial {
    on enter if [[thisQAGamePlayerState.currentQA == null]] do {{
        thisQAGamePlayerState.currentQA = thisQAGame.qas[0];
    }};
    ->active on [[thisQAGamePlayerState.currentQA]];
}
```

Figur 10: Scriptkode fra QAGame

Ved entring av denne tilstanden sjekker man først om currentQA (gjeldende spørsmål) er null, altså ikke satt. Hvis dette stemmer tilordner man det første spørsmålet i arrayet ”qas” fra QAGame til currentQA. Til slutt går man til Active hvis currentQA settes til en ny verdi og ikke er null. Før vi går igjennom Active må vi se på spørremaskinen. Spørremaskinen står i Idle og venter på at Init skal kjøre før den kan utføre noe som helst.

```
->idle {
    ->ask on [[thisQAGamePlayerState.currentQA != null && thisQAGamePlayerState.correctAnswer == null]];
}
ask {
    on enter do {{
        thisQAGamePlayerState.gui.question = thisQAGamePlayerState.currentQA.question;
        thisQAGamePlayerState.gui.feedback = null;
    }};
    on [[thisQAGamePlayerState.gui.answer]]
    if [[thisQAGamePlayerState.gui.answer != thisQAGamePlayerState.currentQA.answer]] do {{
        thisQAGamePlayerState.gui.feedback = "Wrong answer, try again";
    }};
    ->correct on [[thisQAGamePlayerState.gui.answer == thisQAGamePlayerState.currentQA.answer]];
}
```

Figur 11: Scriptkode fra QAGame

Figuren viser Idle og Ask i spørremaskinen. I Idle er den første instruksjonen en transisjon basert på to utsagn som må være sanne. Den kan ikke gå til Ask-tilstanden før currentQA er satt og currentAnswer enda ikke er avgitt. CurrentQA ble satt i Init. Ved enter til Ask settes gui’et sin question attributt til currentQA som vi allerede har hentet ut fra arrayet ”qas”. Scriptsnutten ”on [[thisQAGamePlayerState.gui.answer]]” venter på et svar fra spilleren og

scriptet kjører ikke videre før et svar er avgitt, if-setningen som kommer rett etterpå sjekker om avgitt svar er ulikt riktig svar. Feedback i gui'et vil alltid være "Wrong answer, try again" helt til man får det riktig. Den siste instruksjonen er transisjonen til Correct når man avgir korrekt svar.

```
correct {
    on enter do {{
        thisQAGamePlayerState.gui.feedback = "Correct!";
        thisQAGamePlayerState.correctAnswer = thisQAGamePlayerState.gui.answer;
        thisQAGamePlayerState.gui.answer = null;
    }};
    ->idle on [[thisQAGamePlayerState.currentQA == null && thisQAGamePlayerState.correctAnswer == null]];
}
```

Figur 12: Scriptkode fra QAGame

I Correct oppdaterer man gui-feedback til å vise "Correct!", setter correctAnswer til å være det riktige svaret og fjerner svaret i guiet. Deretter går man tilbake til Idle.

```
active {
    on [[thisQAGamePlayerState.currentQA != null && thisQAGamePlayerState.correctAnswer != null]]
    do {{
        thisQAGamePlayerState.answers.add(thisQAGamePlayerState.correctAnswer);
        thisQAGamePlayerState.currentQA = null;
        thisQAGamePlayerState.correctAnswer = null;
        if (thisQAGamePlayerState.answers.size() < thisQAGame.qas.size()) {
            thisQAGamePlayerState.currentQA = thisQAGame.qas[thisQAGamePlayerState.answers.size()];
        }
    }};
    ->finished on [[thisQAGamePlayerState.answers.size() == thisQAGame.qas.size()]];
}
```

Figur 13: Scriptkode fra QAGame

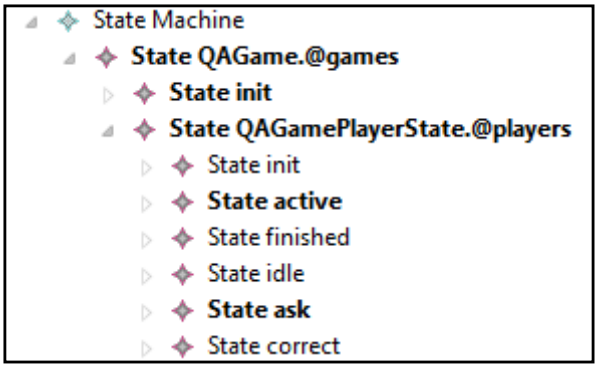
Active er en litt mer komplisert tilstand, denne tilstanden er aktiv så lenge man har spørsmål igjen og fortsatt svarer på spørsmål. Først er det en sjekk på om currentQA og currentAnswer er satt. Ved dette tidspunktet har spørsmålet blitt stilt i det enkle grensesnittet i Eclipse og maskinen venter på svar. Når riktig svar kommer blir en sekvens med instruksjoner utført, disse starter med "do" og blir utført innenfor klammeparentesene som kan sees på figuren ovenfor. Gjeldende spørsmål og riktig svar blir fjernet, og neste spørsmål fra arrayet "qas" blir satt i GUI's tekstfeltet. Dette skjer så lenge det er spørsmål igjen. Transisjonen til Finished er når antall korrekte avgitte svar er lik antall elementer i "qas". I Finished er spillet ferdig, og dette blir vist med å oppdatere tekstfeltet i GUI til "Finished!".

```
finished {
    on enter do {{
        thisQAGamePlayerState.gui.question = null;
        thisQAGamePlayerState.gui.feedback = "Finished!";
    }};
}
```

Figur 14: Scriptkode fra QAGame

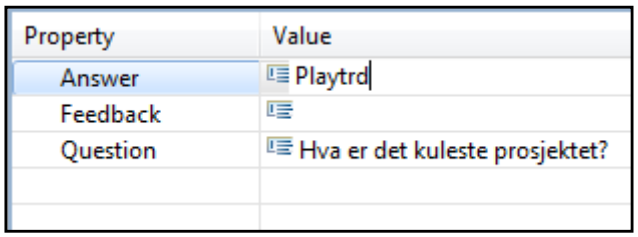
4.6.3 Kjøring

Når man kjører denne scriptkoden vil spillmaskinen gå til Active og spørremaskinen til Idle umiddelbart. Dette kan ses på figuren under som er et utklipp av datamodellen under kjøring, de aktive tilstandene er fet skrift.

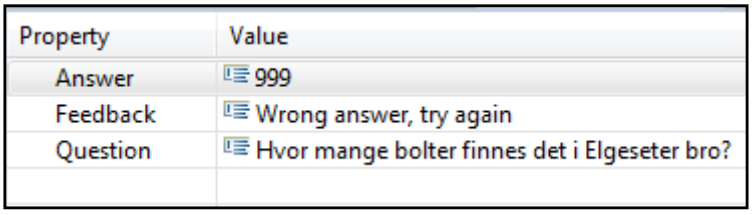


Figur 15: Tilstandsmaskin kjørende i en dynamisk instans av QAGame

Figuerene under viser det enkle GUI'et i ulike scenariorer.



Figur 16: Endring av verdier i dynamisk instans av QAGame



Figur 17: Tilbakemelding fra tilstandsmaskin i GUI

Når man skriver inn ett riktig svar går spørremaskinen fra Correct til Idle for å hente et nytt spørsmål og tilbake til Ask. Etter at spørsmålene er gjennomløpt vil tilstandsmaskinen gå til tilstanden Finished.

4.7 Tilgjengelige plattformer

I denne oppgaven var det hele tiden gitt at løsningene skulle realiseres med hjelp av utviklingsplattformen til Eclipse. Når produktene en dag i fremtiden skal tas med videre på mobile plattformer ser vi også for oss en løsning basert på en av de tilgjengelige Java-teknologiene. I denne seksjonen ønsker vi å utdype litt hva alternativene er, se litt på hvordan markedet er i ferd med å utvikle seg og hvordan mulighetene til å drive modelldrevet utvikling er på disse plattformene. Vårt sluttprodukt er ikke et produkt som er klart for å drives på slike mobile plattformer, men vi synes allikevel at problemstillingen er interessant, og at det er relevant å se litt på hvordan mulighetene er for å drive modellbasert utvikling på mobile plattformer for Playtrd prosjektet.

Det norske mobile landskapet er i dag preget av en håndfull forskjellige plattformer. Felles for de fleste plattformer er at de i stor grad er inkompitable med hverandre. Programvare skrevet for en plattform krever en betydelig mengde arbeid for å bli kompitabel med neste plattform. Det finnes ikke en enkelt teknologi eller programvare som gir kompabilitet med hele markedet.

På plattformer som ikke regnes som smarte må man oftest ty til Java Micro Edition (Java ME). Det er varierende hvor funksjonsrike mobilspill kan være dersom de skal tilpasses disse plattformene. Java ME er kanskje den teknologien som når flest plattformer, men allikevel så er den ikke nødvendigvis den løsningen som anbefales fordi den er noe begrenset hva teknologi angår og lider av et veldig fragmentert miljø, der man gjerne ser at programvare som virker på en telefon ikke nødvendigvis virker på en annen modell. Men allikevel så er det til en viss grad kompabilitet mellom Java og Java Me, noe som bidrar til at arbeid med å tilpasse modelldrevet utvikling til disse plattformene ikke nødvendigvis er håpløst selv om vi ikke har lokalisert noen kilder som kan eksemplifisere at slik konversjon faktisk eksisterer i dag.

Plattform	Tilgjengelige språk for utvikling	Anbefalt løsning for utvikling
Apple iPhone	Objective C	Objective C
Windows Mobile 6.x smarttelefoner	.NET Compact Framework, Java ME	.NET Compact Framework
Sony Ericsson "standardtelefoner"	Java ME	Java ME
Nokia Series 40 "standardtelefoner"	Java ME	Java ME
Nokia Symbian S60 telefoner	Open C/C++, Symbian C++, Java ME	Symbian C++
Androidtelefoner	Java, C/C++	Java

Tabell 1: Mobile plattformer

Det er også alltid slik at en del teknologier og plattformer til en hver tid anses som mer "in" å utvikle opp mot. Man ser nå et betydelig større utviklingsmiljø på de smarte plattformene, og spesielt de som tilbyr innebygde programvarebutikker på plattformen. De siste par årene har Apple gjort kjempesprang inn i markedet og tilegnet seg et stort marked både i antall solgte telefoner og gjennom salg og nedlasting av applikasjoner til plattformen (Digi.no, 2010). Det er tvilsomt om det lønner seg å unnlate å utvikle til iPhone dersom man skal realisere et kommersielt produkt på mobil plattform i Norge i dag. Dog er det relevant å nevne at dersom man skal utvikle opp mot iPhone så er man avhengig av å bruke maskinvare fra Apple med Mac OS X som operativsystem. Det finnes et verktøy kalt iPhonical som gir anledning til å lage applikasjoner til iPhone OS ved å utvikle tekstlige modeller som blir konvertert til kode for iPhone (iPhonical, 2010). Men det er per i dag en usikkerhet tilknyttet lovligheten til dette verktøyet fordi Apple er svært beskyttende ovenfor utviklerverktøyene sine og tillater ikke nå lengre å bruke tredjeparts programvare til å utvikle applikasjoner for sin mobilplattform (Apple, 2010).

En annen plattform som er i ferd med å innta Norge er Android. Android tilbyr et system som er åpnere, da det bygger på Linux. Linux er fri og åpen programvare, og gir Android det mange ser på som en fordel med et åpnere økosystem. Androidplattformen blir heller ikke begrenset til bare en produsent, og dermed vil vi se flere produsenter som kommer til å satse stort på Androidtelefoner i det norske markedet fremover. Blant annet ser vi nå at HTC, Samsung og Sony Ericsson lanserer nye modeller i Norge med Android. Google har på kort tid gjort store fremskritt for å tilby en konkurransedyktig plattform i markedet, og den vokser i utbredelse bak iPhone, som igjen gjør den til en mer attraktiv plattform å drive utvikling på. Android tilbyr utviklere funksjonalitet på linje med Java, og er derfor en høyaktuell plattform til å jobbe med kombinasjonen EMF og modelldrevet utvikling. Det finnes også et eksempel hvor folk har jobbet med å få hele EMF kjernen til å kjøre på Android (Song, 2009).

I tillegg finnes også Microsofts plattform Windows Mobile på markedet for smarttelefoner. Denne plattformen lider av at den er på vei mot et veiskille som kommer til å bryte kompabilitet med all eksisterende programvare i et forsøk på å modernisere og gjenvinne markedsandeler. Dette veiskillet er nært forestående, så det er nå ikke lenger noe mening i å diskutere den gamle plattformens egenskaper som plattform for modelldrevet utvikling. I tillegg så er den kommende plattformen så fersk at det igjen er vanskelig å diskutere dens egenskaper hva modelldrevet utvikling angår. Det som i alle fall er sikkert er at Microsoft vil ønske at utvikling skal foregå på sine vilkår, som igjen vil si på Windows-basert operativsystem og med Microsoft Visual Studio som utviklingsprogramvare. Derfor kan det rådes å se litt an hvordan denne plattformen blir mottatt før man investerer i utvikling på denne.

Et alternativ som kan være med å gjøre begrensninger med mobile plattformer og deres kompabilitetsproblemer mindre er å legge funksjonaliteten ute på web som

webapplikasjoner. Dette gjør at arbeidet med plattformversjoner av produkter minker ettersom logikken drives eksternt. Plattformversjonene blir da såkalte front-end applikasjoner, det vil si hovedsakelig brukergrensesnitt.

5 Zombiespill

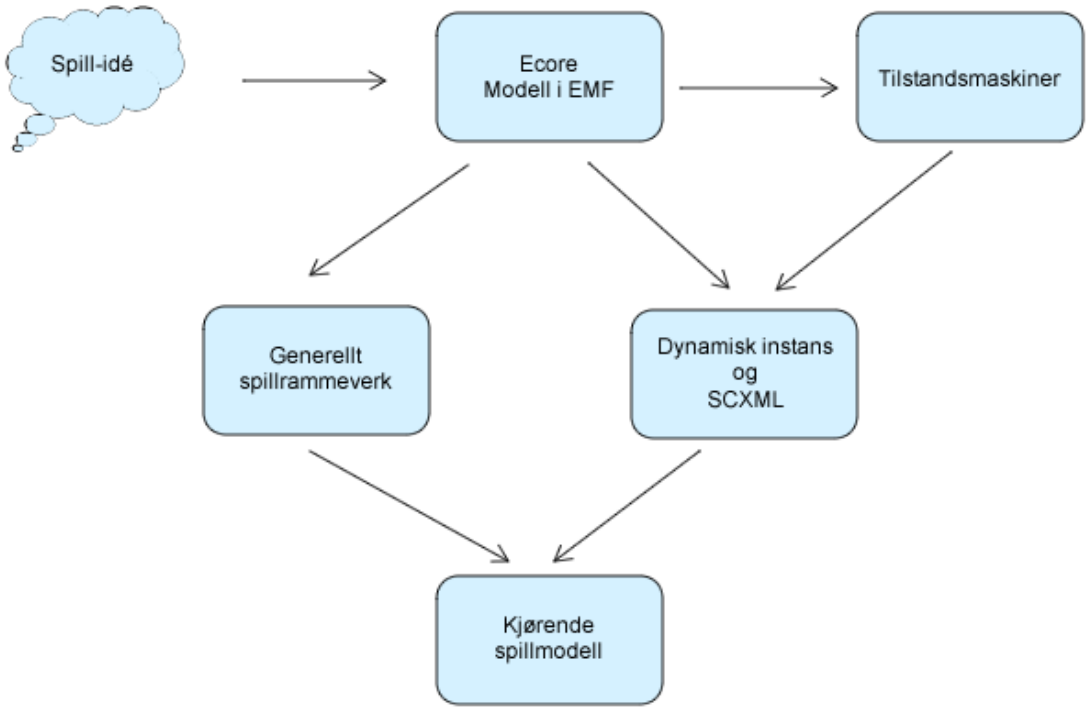
Etter at vi nå har sett nærmere på generell spillteori, introdusert økosystemet som prosjektet lever i og sett på teknologi, ønsker vi å bringe fokuset videre på et egenutviklet spillkonsept. Vårt konsept er til for å prøve utvikling av spill innenfor rammene som er satt i de forestående kapitlene. Det vil si at vi ønsker å utvikle et spill innenfor interesseområdet for PlayTrd-prosjektet, gjennom å drive modellbasert utvikling med teknologi presentert i forrige kapittel.

Dette kapittelet oppsummerer progressen i implementasjonsfasen som de neste kapitlene vil presentere resultatene av nærmere og går igjennom konseptet for å vise hvordan dette spillet er tenkt gjennomført i praksis, med mange spillere samme i en pervasive verden. Vårt fokus er å prøve ut en tilstandsbasert motor for pervasive spill, og derfor har vi ikke lagt for stor vekt på selve implementeringen av hele spillkonseptet. Som et eksempel på hvordan man kan gjøre spillet mer interessant, har vi også presentert flere funksjoner som kan være med å berike spillet men som ikke er implementert. Disse funksjonene la vi i fra oss på grunn av at hovedfokuset var heller å teste utviklingen av et spillkonsept ut ifra en teknologisk vinkling, framfor å gjøre spillet mer komplisert.

5.1 Gjennomføring

I kapittel 1.1 introduserer vi forskningsspørsmålene som er sentrale for oppgaven. Vi har allerede nevnt at vi ønsker å se hvordan modellbasert utvikling egner seg for å implementere spill, og hvordan teknologien egner seg til implementasjonsarbeidet. Senere i dette kapittelet blir spillkonseptet blitt presentert, og i de kommende kapitlene vil resultatene bli presentert gjennom et kapittel som omhandler arkitektur og et kapittel som omhandler resultater. Før vi presenterer spillkonseptet og resultatene innen arkitekturen er det relevant å si noe om arbeidsmetoden.

Som allerede presentert i forestående kapitler skulle vi drive modelldrevet utvikling med verktøy basert på Eclipseplattformen. Arbeidet skulle også resultere i varierende prototyper som skulle være med på å validere arbeidet. Figur 18 nedenfor viser komponenter som inngår i systemet, samtidig som det viser sammenhengen når det gjelder både rekkefølge arbeid ble utført i og også hvordan utvikling innenfor et element drev arbeidet videre til neste element. Det som ikke kommer frem direkte i figuren er at selv om arbeidet ble drevet videre, så hadde det også tilbakevirkende effekter. Arbeid med nye elementer hadde en validerende effekt på det andre arbeidet vi hadde gjort tidligere. Mange endringer i systemet oppstod som et resultat av denne valideringsprosessen. En annen direkte årsak til at behovet for denne valideringen oppstod var at vi ikke hadde nevneverdige erfaringer med teknologien fra tidligere oppgaver. I kapittel 7.2 går vi nærmere inn på påvirkninger som valideringsprosessene hadde gjennom utviklingen.

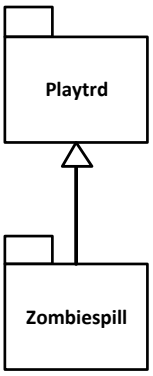


Figur 18: Progresjon i arbeidet

Oppsummert startet arbeidet med å utvikle et spillkonsept for å ha noe å implementere som kunne bygge opp under våre forskningsspørsmål. Deretter jobbet vi med å utvikle modeller i Ecore. Gjennom dialog og samarbeid med Hallvard Trætteberg ble det bestemt at spillkonseptet skulle splittes i to modeller av hensyn til enkelhet i modeller og at fremtidige spillkonsepter kunne nyttiggjøre seg av tidligere utført arbeid:

- En modell skulle være av generell karakter, som vil si at tanken var at all funksjonalitet skulle være overordnet funksjonalitet som vi så for oss var aktuell for mer enn vårt spillkonsept.
- En modell med funksjonalitet som var spesielt ment for vårt spillkonsept, funksjonalitet av såpass særegen karakter at den ikke var å anse som aktuell for særlig flere spillkonsepter enn vår egen.

Ansvarsområdene ble delt her. Trætteberg påtok seg hovedansvaret for den generelle modellen, siden han er mer involvert i fremtiden i prosjektet PlayTrd og hadde bredere erfaring med teknologien. Vi jobbet med å konkretisere den generelle modellen til å inneholde den ekstra funksjonaliteten som vårt spillkonsept behøvde. Man kan gjerne se for seg modellene som hierarkiske, slik som figuren under. Vår modell bygger ut funksjonaliteten fra den generelle.



Figur 19: Hierarkiske modeller

Selv om vi ikke har spesifisert og lagd den generelle modellen finnes det eksempler på at funksjonalitet ble ekstrahert fra vår modell og satt inn i den generelle modellen fordi funksjonalitet som vi hadde modellert ble ansett som viktig nok til å inngå som en del av den generelle modellen. Et eksempel på dette er funksjonalitet som omhandler transaksjoner av objekter mellom spillere, eller mellom "butikker" og spillere. Vi hadde modellert dette fordi vi tenkte at spillere kunne handle "Buffs" (se kapittel 5.2.3) i innlagte butikker for å modifisere spillopplevelsen. Funksjonaliteten endte opp med å bli ansett som såpass viktig at den kunne flyttes opp i den generelle modellen, der vi fortsatt kunne bruke den, men det kan da også alle andre spill som utvikles på samme måte. Den generelle modellen og vår spillmodell blir nærmere presentert i neste kapittel.

Som et ledd i å validere modellene, og som neste steg i utviklinga, drev vi med papirbaserte tilstandsmaskiner basert på standardnotasjon for tilstandsmaskiner. Det fantes flere årsaker til at vi drev med papirbaserte tilstandsmaskiner. Det var tidlig i prosessen og vi ønsket først og fremst å finne de enkle feilene og identifisere mangler i modellene. Samtidig ventet vi på at Trætteberg skulle ferdigstille implementasjonen som ville gi og en DSL basert syntaks for tilstandsmaskin-implementasjon, slik at vi måtte bruke andre hjelpemidler i mellomtiden. I kapittel 7.3 finnes eksempler fra arbeidet med papirbaserte tilstandsmaskiner.

Det siste leddet av arbeidet vårt med spillet involverte dynamiske instanser og til en viss grad utvikling av tilstandsmaskiner. Dynamiske instanser kan sees på som "modellen tatt i bruk". Gjennom å lage en dynamisk instans kan man ta i bruk innholdet i modellen og eksemplifisere innhold slik at man se hvordan modellen opererer under kjøretid. Disse dynamiske instansene er veldig enkle å opprette og er springbrettet for sammenkobling mellom modeller og tilstandsmaskiner.

Implementasjon av tilstandsmaskiner ble en prøvelse for oss. For å kunne utvikle tilstandsmaskiner var vi avhengig av en del nye komponenter som ble utviklet av Trætteberg. Når vi mottok disse komponentene begynte vi å nærme oss en fase hvor vi måtte legge mye vekt på å skrive oppgaven, og vi var plaget med en del feil som fantes i implementasjonen. Derfor inneholder ikke vårt arbeid en ferdigutviklet tilstandsmaskin basert på vårt spill. Dette

blir nærmere diskutert i kapittel 7.4 og 7.5, samt under diskusjonen i kapittel 8.1 hvor vi diskuterer kvaliteten på arbeid og resultater fra vårt ståsted.

De kommende kapitlene skal besvare at modelldrevet utvikling kan brukes til å utvikle spill. Vi skal også vise at teknologien er implementert, om enn ikke helt fininnstilt og fri for småfeil, slik at det går an å utvikle spill gjennom rammeverket som er i ferd med å komme på plass i prosjektet PlayTrd. Med videre arbeid skal det også være godt mulig å snart komme til en fase hvor man kan begynne å teste spillene i den reelle verden. Kapittel 8 omhandler diskusjon av resultater og tanker rundt fremtidig arbeid.

Men før vi går videre på å diskutere arkitekturen så er det viktig å presentere spillkonseptet som er basisen for produktet som blir presentert i de kommende kapitlene.

5.2 Spillkonseptet

Vi måtte komme opp med en spillide som kunne realiseres og brukes mot den generelle spillmodellen. Poenget er å konstruere en basis vi kan jobbe videre med som vist i Figur 18, ved at spillidéen driver i gang prosjektet. Til det kreves et spill som i teorien er praktisk gjennomførbart med tanke på PlayTrd, men også rikt nok til at vi kan modellere tilstandsmaskiner som går litt inn i dybden på spillmekanismen. Det første vi tenkte på var å modellere en egendefinert variant av enkel barnelek, men kom fort frem til at en barnelek ikke var interessant nok for oss i denne omgang og ikke et godt eksempel på et funksjonsrikt pervasive spill. Vi tenkte bredere og bestemte oss for å bruke litt mer tid på å komme opp med en spillide som tar utgangspunkt i et pc-spill vi akkurat hadde blitt kjent med på den tiden med navn Left 4 dead (Wikipedia Left 4 Dead, 2010). Left 4 Dead er et klassisk førstepersons skytespill hvor en liten gruppe mennesker forsøker å overleve mot endeløse horder av zombier. I spillet går man sammen med sine venner fra et ”safe house” til det neste med det mål om å komme til det siste “safe house” for å klare spillet. I “safe house” ligger det gjerne diverse hjelpemidler man kan bruke for å overleve mot de infiserte zombiene.

5.2.1 Oversikt

Mange spillelementer har gjennomgått flere revideringer etter hvert som selve konseptet har blitt ferdig spikret. I denne seksjonen vil hele konseptet presenteres for å gi et overblikk over mulighetene ved et slikt spill i pervasive sammenheng. Som i Left 4 Dead er det to spillertyper, mennesker(survivors) og zombier. Hvor mange det skal være av hver type avhenger av antallet spillere, men en generell regel er at det bør være flere mennesker enn zombier ved start, ettersom noen av menneskene vil bli tatt og konvertert. Det er litt forskjell på de to typene, mennesker har flere valg enn zombiene. Hovedpoenget er at

zombier skal komme nær menneskene og ta dem, slik at også de blir zombier og kan jage mennesker.

Menneskene har bare et sted til en hver tid hvor zombiene ikke kan ta dem, en såkalt trygg sone. Denne sonen forandrer lokasjon under spillets gang. Når alle overlevende har kommet til den trygge sonen vil en timer starte. Etter at timeren har rent ut er ikke denne sonen trygg lenger og en ny sone vil oppstå et stykke unna. Hvis man vil hvile mest mulig i den trygge sonen før man skal videre må, man komme seg dit tidligst mulig. Målet til de overlevende er å komme seg til den nye sonen så raskt som mulig. Samarbeid er nøkkelen for suksess, både når det gjelder å oppdage zombies og legge strategier for hvilken rute man skal ta til neste sone. Samtidig er det "overlevelse av den mest tilpassede" som råder, klarer man ikke løpe fra eller overliste zombiene blir man tatt og er det fare for at man må bruke resten av tide på å jage mennesker.

Zombier har bare ett mål for øyet og det er å ta flest mulig mennesker. Zombier vil alltid ligge et sted bak menneskene, da de ikke får vite om lokasjonen til trygge soner. Menneskene trenger denne fordelen for at ikke zombiene skal ha muligheten til å vente ved en lokasjon de vet menneskene skal til.

5.2.2 Motivasjon for å spille

For at det skal være givende å spille må det være et poengsystem. Uten et poengsystem kan det virke håpløst å starte som zombie med det ensidige målet om å jage mennesker.

Handling	Poeng
Gå inn i ny trygg sone	X / 2
Ta et menneske	X
Overleve spillet som menneske	X * 2
Plukke opp en pengesekk	X / 5

Tabell 2: Forslag til poengfordeling i Zombies vs. Survivors

Ved start bestemmes antall trygge soner menneskene må igjennom før runden er over. Spillet kan også ende ved at alle menneskene er konvertert til zombier. Ved rundens slutt er det spilleren med høyest poengsum som vinner. Spillerne får vite poengsummen underveis i spillet slik at de kan gjøre taktiske valg for å nå toppen ved spillets slutt. Tabellen over viser fordeling av poeng basert på tenkt vanskelighetsgrad på de ulike handlingene.

5.2.3 Spillelementer

Konseptene som presenteres her er elementer for å gjøre ideen rikere, og gi en rollespillaktig følelse av å være delaktig i en fiktiv verden.

Kart

For å gjøre selve kjøringen av spillet mer underholdene for deltakerne skal alle brukerne ha et minikart over området hvor spillet foregår. Dette kartet er en del av grensesnittet på enheten spillerne bruker. På dette kartet kan man se seg selv, og andre spilleres posisjoner i sanntid under kjøring. Det er på denne måten de tiltenkte ”buffene” kommer til sin rett. En buff er et midlertidig hjelpemiddel man kan skaffe seg for kredit. En buff kan bare brukes en gang, og gir fordeler av ulik art under spillets gang. I tillegg kan survivors alltid se den trygge sonen de er på vei til. Poenget med kartet forklares best med kredit og de ulike buffene i seksjonene under.

Figur 20: Tidlig skisse av kart.

Figur 20 viser kartet fra en zombie sitt perspektiv. Den store svarte markøren viser din egen posisjon, de mindre markørene er andre spillere, fargekode angir type spiller. Merk at zombier ikke kan se den trygge sonen og pengesekksymbolet for kreditt. Sitasjonen på bildet kan være at menneskene beveger enten mot eller bort fra en trygg sone nederst til venstre på kartet.

Kredit

I det avgrensede området spillerne beveger seg i, ligger det symboler av pengesekker menneskene kan bevege seg i nærheten av for å motta kreditt. Størrelsen på pengesekken avhenger av hvor langt unna en spiller må bevege seg den neste trygge sonen for å nå pengesekken. Kredit er kun for menneskene og kan kun brukes til å kjøpe buffs.

Buffs

En buff er et hjelpemiddel menneskene kan bruke for å forvirre zombies. Disse fås kjøpt i trygge soner og kan brukes når menneskene beveger seg fra en sone til en annen. Vi har laget ideer for 4 ulike typer buffs, men det er selvsagt ikke umulig at i det i fremtiden blir laget andre og mer interessante typer som kan bli inkorporert i spillkonseptet.

Usynlig

Spilleren som bruker denne buffen blir midlertidig usynlig på zombienes kart. Dette kan med fordel brukes for komme seg til en trygg sone uten at zombiene jakter på deg, eller ta en avstikker for å plukke opp en pengesekk man vet er stor på kreditt.

Se zombie (Spot the zombie)

Denne buffen kan komme i flere varianter. Den billigste varianten er et hjelpemiddel for en spiller, og gir den muligheten til å se nærmeste zombie i en begrenset periode (15 sek) på sitt kart. Den andre varianten gjør alle zombier synlig på kartet (5 sek). En variant som koster mye gir alle overlevende mennesker muligheten til å se alle zombiene en gitt periode. Dette hjelpemiddelet er typisk dyrt og gagner bare de som er ikke enda har kommet seg inn i trygg sone og føler veien dit kan bli farlig. Her kommer spørsmålet om lagspill og samlet ressursbruk inn i bildet.

Falskt menneske

Denne buffen gjør at menneskemarkøren til spilleren som bruker buffen flytter seg en gitt avstand i forhold til hvor mennesket egentlig er. For å ta et spesifikt eksempel; mennesket befinner seg helt inntil Olavsstatuen, men markøren til spilleren på zombienes kart er på bussholdeplassen Torget. Avstanden mellom mennesket og markøren er konstant i ett minutt, hvis mennesker flytter seg en gitt lengde, vil også markøren flytte seg den gitte lengden slik at avstanden forblir konstant ut tiden. Ulempen her er at markøren kan plutselig være midt på en bygning, eller en annen lokasjon hvor menneskene ikke har lov til å være, det blir enkelt å skjønne at noe ikke er riktig.

Bli zombie

Denne buffen gjør at markøren til spilleren blir lik markøren til en zombie i ett minutt. Det kan være meget verdifullt om man er heldig med timingen og zombiene ikke ser på kartet i det øyeblikket markørfargen skifter. Buffen har ingen hensikt mot zombier som ser markørskiftet på kartet.

Det er ikke mulig å si hvor mye buffene skal koste på dette tidspunktet. Tanken er at de skal bli jevnlig brukt som hjelpemiddel, men hvor effektive de ulike typene er kan man bare finne ut ved å kjøre igjennom spillet i midtbyen med mennesker som gjør hva de kan for å få flest mulig poeng i løpet av en runde. Pengesekker må være tilgjengelige for alle, og funksjonen for å beregne verdien av en pengesekk må tilfredsstille hovedmålet om at i en vanlig gjennomføring er det noen mennesker som klarer å overleve til slutten.

Zombie til menneske-konvertering

Hvis man først har blitt tatt av en zombie, forblir man zombie til hele runden er over. Det kan virke litt demotiverende å bli tatt tidlig i spillet, uten muligheter for å gå tilbake hvis man foretrekker å være menneske. Spillet føles nok litt rikere for de som er mennesker. Derfor er det en mulighet for å bli konvertert tilbake til menneske hvis man klarer å være en effektiv zombie. Det avhenger av antall spillere, men klarer man å ta 3 mennesker i en spillinstans med 15-20 spillere bør det kvalifisere til å få en ny sjanse på å unngå zombiene.

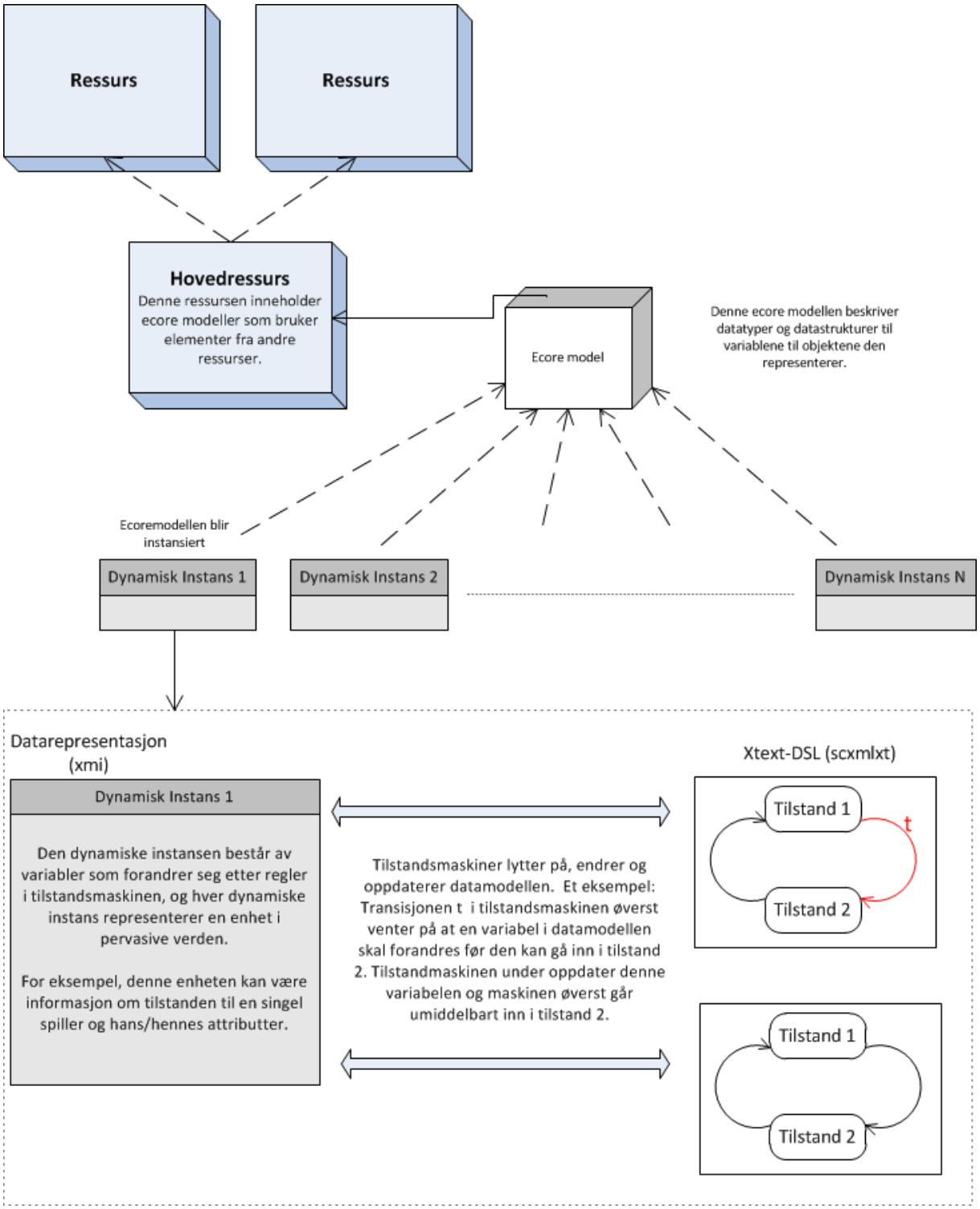
6 Arkitektur

Nå som spillkonseptet er definert, og etter at teknologi, verktøy og teori er presentert, er det på tide å bringe fokus over på implementasjonen av spillkonseptet. I dette første av to kapitler som tar for seg praktiske resultater er det arkitektur som står i fokus. Og med arkitektur mener vi her modeller. Dette kapittelet gir først en kort oversikt over oppførselen til et system av denne typen. Kapittelet fortsetter med å gi en innføring i Hallvard Trættebergs generelle modell som er en samling av hovedfunksjoner tiltenkt å ha en overordnet rolle for alle spill som utvikles innenfor PlayTrd. Deretter går fokuset videre på vår egen modell. Modellen blir presentert og utdypet i det siste underkapittelet.

Det er tidligere hevdet at i modell-drevet ingeniørvitenskap ikke eksisterer en blueprint som blir slavisk fulgt under implementasjon. Vårt fokus har vært på domeneabstraksjon og kontinuerlig oppdatering. Vi har ingen kravspesifikasjon med formelle etterprøvbare krav, men modellene har rollen som systemspesifikasjon. Modellene har en hierarkisk oppdeling og i dette kapittelet vil vi presentere fordelene dette gir oss i relasjon til tilstander, tjenester og kommunikasjon.

6.1 Datamodeller og tilstandsmaskiner i praksis

I kapittel 2.4.3 demonstrerte vi tankegangen rundt å benytte seg av datamodeller og tilstandsmaskiner som reagerer og oppdaterer dataen. Figuren under viser den faktiske oppførselen til systemet implementert med teknologien omtalt i kapittel 4.



Figur 21 Dynamiske instanser under kjøring

Selv om det i figuren ligger forklarende kommentarer, må det utdypes noe. Øverst har vi illustrert ressursoppdelingen til EMF, en eller flere sammenkoblede Ecore-modeller kan identifiseres med en URI. Dette gjør det enkelt å arbeide med mye informasjon i flere modeller samtidig uten å måtte ha alt i minnet hele tiden, siden man i EMF kan benytte seg av såkalt ”lazy loading”. Dette vil si at man laster ikke spesifikt innhold i ressursen før man faktisk trenger å bruke det, og er en fordel med stort ressursinnhold. Videre blir Ecore-modellen instansiert og koblet sammen med tilstandsmaskiner som driver mekanismen videre.

6.2 Den generelle modellen

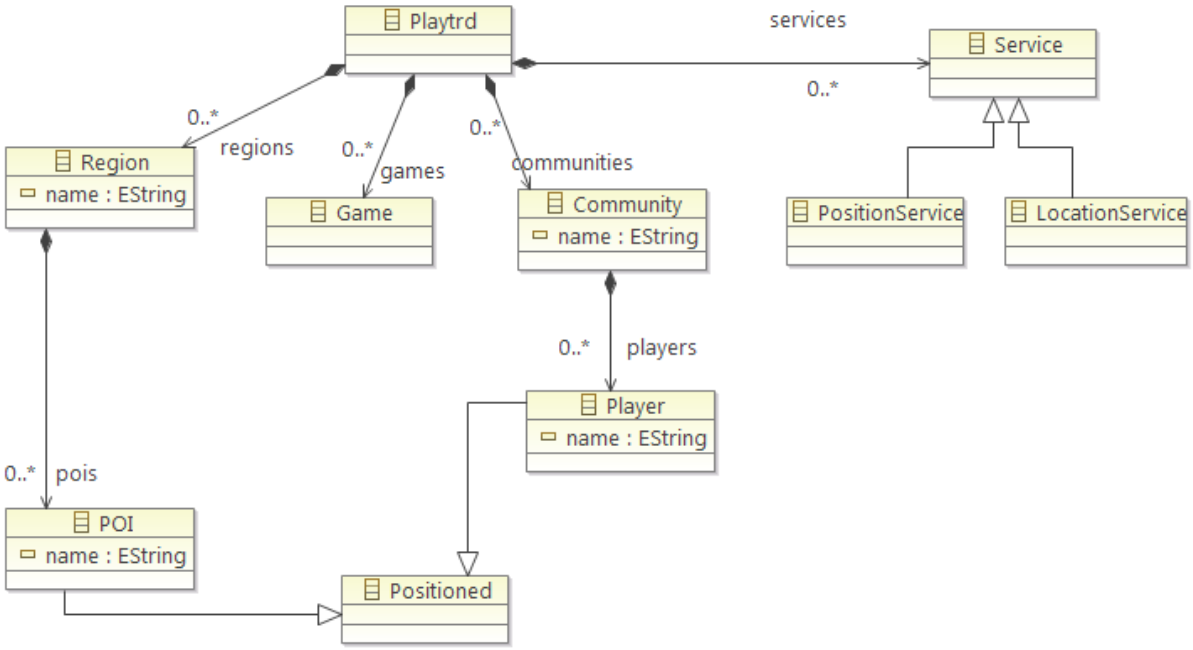
Denne generelle modellen er utviklet hovedsakelig av Hallvard Trætteberg, med innspill fra oss og andre som jobbet med underprosjekter i PlayTrd gjennom skoleåret. Modellen består av konsepter som vi anser som universell, altså funksjonalitet som vil gå igjen i mange av spillimplementasjonene i PlayTrd. Hovedprinsippet er å forenkle arbeidet for alle, samtidig som man forhindrer en mengde redundans både i arbeid og innhold i modellene. Modeller kan samarbeide ved å referere til hverandre, slik at vi oppnår et hierarki av modeller. Eksemplifisert så har vi den generelle modellen liggende på topp med all universell funksjonalitet, og vår implementasjon av zombiespillet liggende under, med noen egne klasser, noen subklasser og noen referanser til eksisterende klasser fra den generelle modellen. De følgende avsnittene presenterer sentral funksjonalitet som er implementert i den generelle modellen.

Toppnivåklassen (PlayTrd), er et sentralt element både rent generelt i Ecore og for spillene. Poenget med toppnivåklasser er at det må finnes en klasse som er beregnet til å ligge på toppnivå, som i en instans kan ta ansvar for å opprette og referere til andre instanser av klasser. Riktignok kan disse instansene av underklassene igjen opprette andre instanser av klasser lavere ned i abstraksjonshierarkiet. Tankegangen er at vi fra en instans av toppnivåklassen får muligheten til å opprette instanser av alle typer klasser i modellen, enten direkte fra toppnivå eller gjennom en eller flere nivåer av klasseinstanser som stammer fra toppnivået. Dette er gjort fordi man skal kunne enklest mulig navigere seg gjennom modellen uten at funksjonalitet blir liggende utenfor. I den generelle modellen benyttes toppnivåklassen til å definere rammene til et spill. Vi har PlayTrd på toppen, og under står vi fritt til å inkludere alle underklasser som den generelle modellen eller vårt spill tilbyr i eventuelle instanser av spill.

Fra PlayTrd-klassen har vi direkte tilgang til en del underklasser. Vi kan opprette regioner som igjen inneholder grafiske lokasjoner på såkalte interesseområder. Vi kan også opprette communities, som er begrepet på en samling av spillere som deltar i spillverdenen. Spill (games) opprettes også herifra, og derfor har PlayTrd en referanse til alle spill i spillverdenen. Vi har også tjenester (services) som vi ser nærmere på i neste avsnitt.

En annen sentral funksjonalitet som støttes i den generelle modellen er konseptet om et inventarsystem. I mange spill opererer man med inventar av objekter av forskjellige slag. I vårt spill har man såkalte "buffs" som nevnt i kapittel 5.2.3, som gir spilleren anledning til å påvirke sin egen sjanse til å lykkes under spillets gang. I den generelle modellen er det innbakt funksjonalitet for at en spiller kan eie et inventar, og å interagere med en annen eier for å bytte, eller drive kjøp og salg av inventarenheter.

Figur 22 viser et utsnitt av funksjonaliteten som er beskrevet i denne seksjonen. Merk at denne figuren kun er en gjenskapning av den originale og derfor vises ikke en del av funksjonaliteten som ikke anses som nødvendig å vise frem her.



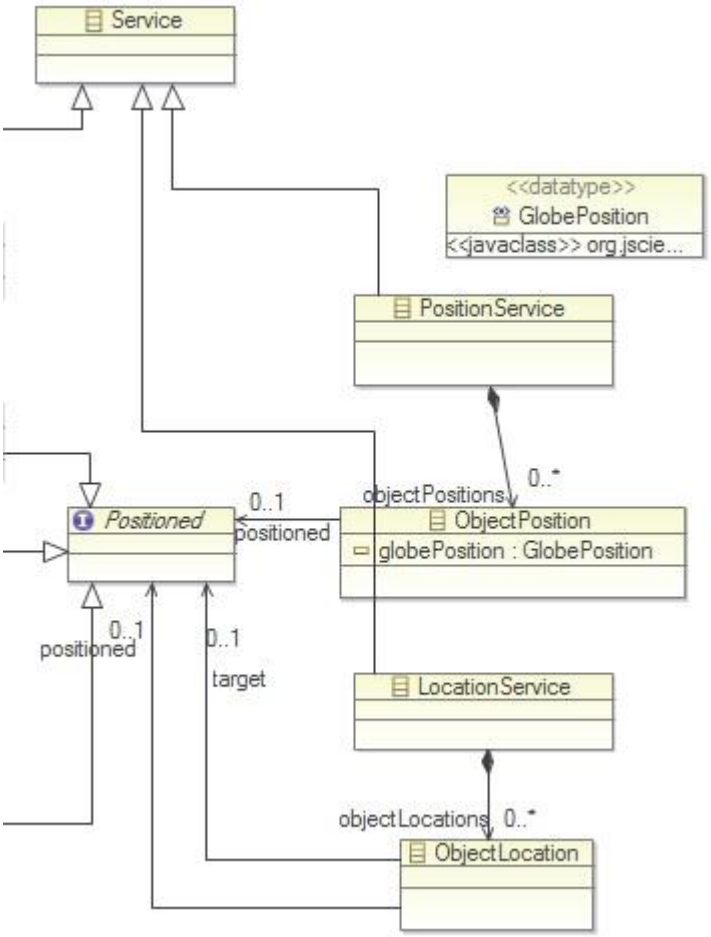
Figur 22: Utsnitt av elementer fra generell modell

6.3 Tjenester

Vår arkitektur bruker prinsipper fra et konsept forkortet SOA, fra det engelske begrepet *Service-Oriented Architecture*. Tjenestene er uavhengige, løst koblede enheter med funksjonalitet uten referanser til hverandre. "A deployed SOA-based architecture will provide a loosely-integrated suite of services that can be used within multiple business domains." (Wikipedia SOA, 2010) Vi opererer ikke innen ulike forretningsdomener, men ulike domener for spill og disse tjenestene er tilgjengelige for alle fremtidige ideer i PlayTrd rammeverket. Modellerte tjenester i PlayTrd-rammverket og deres rolle blir forklart i kapittel 6.3.1. En ny modellimplementasjon trenger ikke bruke alle tjenester, men kan velge å benytte seg av de tjenestene som tilbyr ønsket funksjonalitet.

Den generelle spillmodellen som tilhører PlayTrd-konseptet har også tjenestene i den hierarkiske klasseoppdelingen. Disse er meldingsveksling, posisjonering og samlokalisering

og kan sies å delvis utgjøre en tjenesteorientert arkitektur. Det er ikke utenkelig at behovet for en ny tjeneste kan oppstå, og måten rammeverket er strukturert på legger ingen hindring på å modellere frem en ny tjeneste uten å måtte endre hele arkitekturen.



Figur 23: Utsnitt av lokasjon-og posisjoneringstjenestene i ecore diagram editor

Figur 23 viser to tjenester, PositionService og LocationService med referanser til superklassen Service i Ecore diagrameditoren. Tjenestene er lett tilgjengelige fra toppnivå, og kan derfor enkelt brukes av implementasjoner under PlayTrd.

6.3.1 Tjenester i PlayTrd

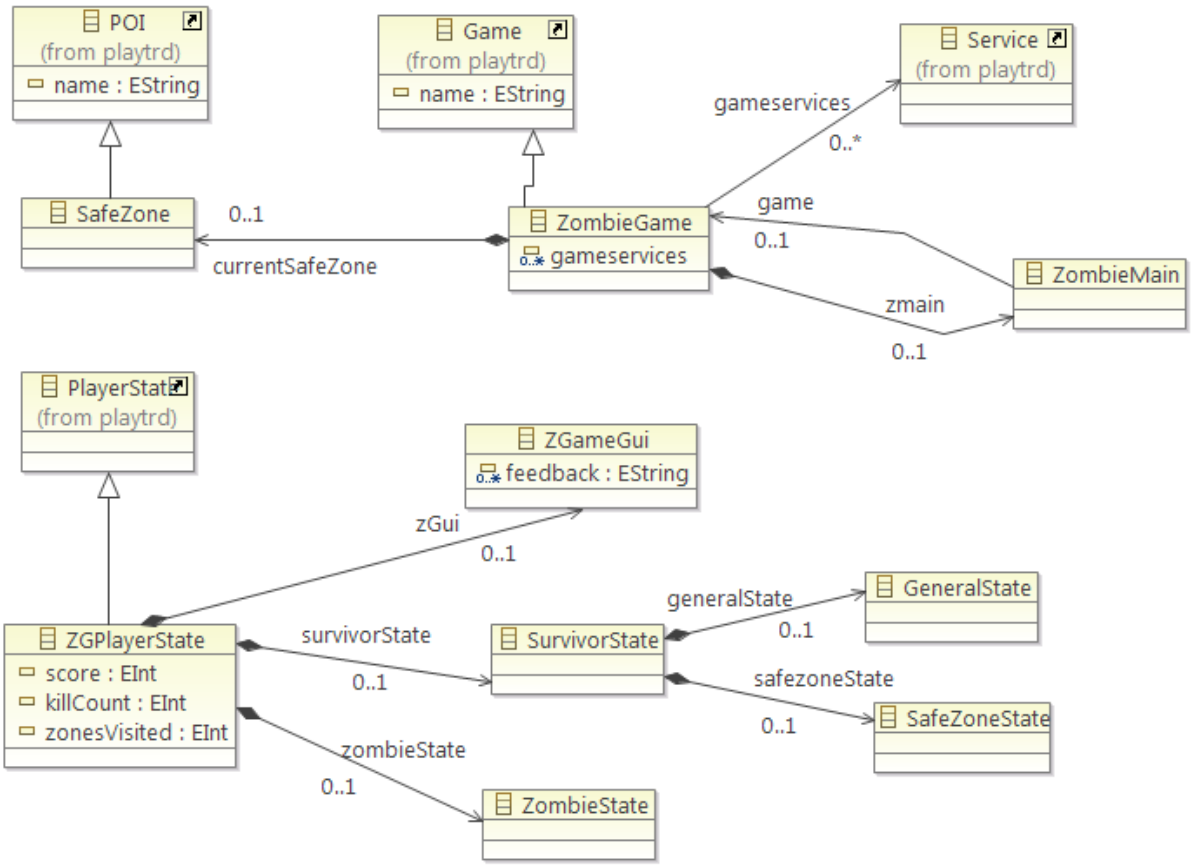
Locationservice er en tjeneste for å gi informasjon om hvilke posisjonerte objekter som er samlokaliserte. I det fleste spill og leker i en pervasive verden vil man få brukt for å vite når objekter er i nærheten av hverandre, og denne tjenesten kan ved hjelp av abonnement-tjener forhold gi melding om når og med hvilke objekter dette skjer. Hovedoppdraget til lokasjonsservice blir å gi beskjed til lyttere når to objekter som følges blir samlokalisert. En slik lytter er da typisk instansen av et Game, som kan reagere på beskjedene fra lokasjonsservice og utføre relevant spillogikk basert på hva som faktisk har hendt. Positionservice er en tjeneste for å angi posisjoner til alle objekter som kan være posisjonerte, dette inkluderer spillere og andre objekter som trenger posisjonsreferanse i

spillverdenen. Meldingsutveksling er en essensiell del av alle moderne flerspillerrammeverk og PlayTrd har messageservice som er en enkel tjeneste for meldingsutveksling mellom spillere, eller spillere og systemet.

6.4 Spillmodell

Vår spillmodell bygger videre på funksjonalitet som eksisterer i den generelle modellen. Hensikten med vår modell er å konkretisere den generelle modellen med funksjonalitet nok til å danne et enkelt spill. Vårt spillkonsept ble presentert i kapittel 5.2 og modellert deretter. Av først og fremst tidshensyn ble konseptet forenklet noe. Fra spillkonseptet er derfor for eksempel elementet “Buffs” fjernet. Årsaken ligger i at slike modifikatorer medfører logikkendringer for enkeltspillere, og dette ville krevd mye tid i utvikling og testing, selv under modelleringsfasene. Prosjektet vårt handlet ikke om å teste en kompleks spillidé, men heller om å få testet utviklingsmetodikk og teknologi.

Figur 24 nedenfor viser en gjenskapning av modellen vår. Grunnen til at vi må gjenskape figuren er at den opprinnelige figuren ikke lengre støtter å bli vist frem grafisk på grunn av en feil som oppstår i Eclipse, som gjør at man manuelt må koble sammen flere modeller, og da vil ikke lengre den grafiske visningen virke. I tillegg inkluderer modellen et par elementer som ble introdusert i forbindelse med implementasjon av tilstandsmaskiner. Klassen ZombieMain eksisterer kun fordi implementasjonen ikke støttet å kjøre tilstandsmaskiner på det hierarkiske nivået til ZombieGame. Derfor inneholder ZombieMain kun en referanse til ZombieGame.



Figur 24: Spillmodell

ZombieGame er en subklasse av Game. Subklassingen eksisterer fordi vi trenger litt ekstra funksjonalitet som ikke eksisterer i Game:

- Vi må ha referanse til gjeldende trygge sone (SafeZone). Det kan også være naturlig å anta at vi skal ha en liste av alle trygge soner i ZombieGame, men dette er ikke modellert inn.
- Vi må ha en referanse til alle tjenester som vi benytter oss av (Service), for enkel tilgang gjennom tilstandsmaskiner. Funksjonaliteten tilbys allerede gjennom toppnivåklassen i den generelle modellen. Derfor blir instanser opprettet og eid av PlayTrd, og ikke ZombieGame. ZombieGame refererer kun til de opprettede tjenestene for enklere tilgang.
- Som tidligere nevnt må vi ha referanse til en ny underklasse som bare skal referere tilbake på spillklassen (ZombieMain), på grunn av begrensninger i implementasjonen av tilstandsmaskiner.

Selv om det ikke kommer frem av modellen, så er spillklassen også ansvarlig for spillerstatus objektene (ZGPlayerState). Dette ansvaret arves fra superklassen, som opprinnelig inneholder en samling av spillerstatusobjekter definert i den generelle modellen. Men vår spillerstatusklasse (ZGPlayerState) arver fra denne klassen og kan derfor benyttes i stedet for klassen som allerede er definert i den generelle modellen. ZGPlayerState er altså en

underklasse av PlayerState. Dette er gjort fordi spillerstatus-klassen vår trenger ekstra funksjonalitet utover det superklassen er bygd for. Først og fremst må vår spillerstatus-klasse støtte to forskjellige typer spillere, siden spillere i vårt spill har to forskjellige spilleropplevelser basert på om de er såkalte overlevende (SurvivorState) eller zombie (ZombieState).

Som nevnt i forrige avsnitt trenger spillerstatusen å kunne skille mellom to forskjellige spillertyper. Vi har modellert dette så enkelt som mulig, ved å opprette to underklasser som spillerstatus-klassen referer til. Tankegangen er at en av disse referansene til en hver tid skal inneholde en null-verdi, og en skal peke på gjeldende type karakter som er aktuell for spilleren på gitt tidspunkt. Hvis spilleren fungerer som en overlevende skal da referansen zombieState være null, og referansen survivorState skal peke på et objekt av typen SurvivorState. Det kan se tungvint ut å modellere dette på denne måten, spesielt siden begge klassene ikke inneholder noen variabler. Grunnen til at dette er modellert på denne måten er at vi i tidligere så for oss mer innhold i disse klassene, for eksempel så skulle zombier kunne konverteres tilbake til overlevende ved å konvertere nok overlevende. Dette kunne vært modellert med å ha en teller i ZombieState, ikke ulik den som finnes i ZGPlayerState nå, men bare at den nullstilles hver gang en spiller blir konvertert til zombie. Telleren blir inkrement mens han konverterer overlevende, og forsvinner som en del av objektet når referansen til ZombieState forsvinner i det spilleren har konvertert nok spillere til å få en ny sjanse som overlevende.

I tillegg til disse to nevnte spillertypene trenger de overlevende å diffensieres i to nye undertyper. En overlevende kan bli konvertert til en zombie i hele spillverdenen, bortsett fra når han befinner seg i en trygg sone (SafeZone). Derfor må vi i modellen skille mellom en overlevende som befinner seg i en trygg sone og overlevende som vandrer sårbar rundt i verden. Dette har vi valgt å løse på samme måte som med differensieringen mellom overlevende og zombies. Klassen SurvivorState har to referanser, generalState til en generell status i spillverdenen (GeneralState) og safeZoneState til å indikere at spilleren er i en trygg sone. Grunnlaget til at vi valgte denne måten også på dette problemet er at vi ønsket en viss likhet i løsningene, og ikke å introdusere en ny måte å løse et liknende problem fra tidligere, selv om vi ikke har sett for oss at det er nødvendig med noe særlig ekstrafunksjonalitet i disse to klassene.

De klassene som er nevnt til nå definerer funksjonaliteten i implementasjonen vår av spillkonseptet. I tillegg til klassene som er nevnt er det relevant å også nevne at enhver spillerstatus også har et eget GUI-objekt koblet til seg. Spillere må ha individuelle GUI, basert på blant annet sin egen lokasjon og om de er overlevende eller zombier. Klassen ZGameGui som refereres til fra ZGPlayerState skal gi denne funksjonaliteten, men den er ikke ferdigdefinert ved innlevering. Dette er en GUI-testklasse og har blitt mest brukt under testing av tilstandsmaskiner. Vi har ikke fokusert nevneverdig på presentasjonen av spillet for sluttbrukere under utviklingen. ,,

7 Resultater

Å benytte modellering som utviklingsmetode i stedet for programmering kan føles nytt og utfordrende. Mange er vant til å utvikle UML-diagrammer før man starter å programmere, eller kanskje til og med etterpå, men det å faktisk benytte modelleringen som grunnlag er noe som ingen av undertegnede som har gjort før. Muligheten til å utvikle med denne metodikken åpner seg ved å benytte Eclipse som utviklingsplattform. Eclipse gir rikelig med funksjonalitet, og det finnes spesialtilpassede Eclipse-utgaver som utvider grunnfunksjonaliteten i plattformen. Den spesielle versjonen som lå til grunn for vår utvikling kalles Eclipse Modeling Tools.

Vår motivasjon til å velge denne oppgaven var av flere grunner. Vi ønsket først og fremst muligheten til å jobbe med reell teknologi som vi anså som nyttig og fremtidsrettet, men også lære oss en ny metodikk og arbeide sammen. Vi skal begge ut i arbeidslivet etter dette skoleåret og anså masteroppgaven som en mer interessant og reell utfordring når man ikke jobber alene. Det var også en fin anledning til å jobbe med noe som var markant forskjellig med det som vi var vant til fra før av. Begge heller mer mot programmering som utviklingsmetode.

Da vi startet på oppgaven merket vi ganske snart at det var en viss forskjell på oppgavebeskrivelsen og hva vi egentlig jobbet med. Vi hadde aldri diskutert og klargjort en endelig problemstilling med veilederen vår, og opplevde at oppgaven heller ble definert med tiden til hjelp gjennom samtaler med veileder, og at vi opparbeidet oss erfaring med teknologier som prosjektet benyttet seg av. For eksempel fokuserte vi ganske lite på den delen av oppgavebeskrivelsen som nevner SOA-basert arkitektur. Selv om SOA-basert arkitektur spiller inn i oppgaveløsningen, så er det ikke blitt fokusert nevneverdig på i løpet av skoleåret. Vi opplevde allikevel dette som veldig positivt, for vi stod begge fritt til å vinkle inn de elementene vi anså som mest interessant i større grad i det praktiske arbeidet. Prosjektet var samtidig å anse som i utvikling på alle fronter, både blant mastergradsstudenter, doktorgradsstudenter og professorer, så det føltes naturlig for oss at det ble som det ble.

I dette kapittelet fortsetter vi presentasjon av vårt praktiske arbeid. Vi har allerede presentert modellene i forrige kapittel, og nå vil vi gå videre med å fortelle litt hvordan vi jobbet, se nærmere på viktigheten av validering og gå gjennom eksempler fra arbeidsprosessen som involverte papirbasert utvikling av tilstandsmaskiner. Videre benytter vi SEQUAL til å kvalitetsvurdere modellen vår, før resultatene våre blir oppsummert mot slutten av kapittelet.

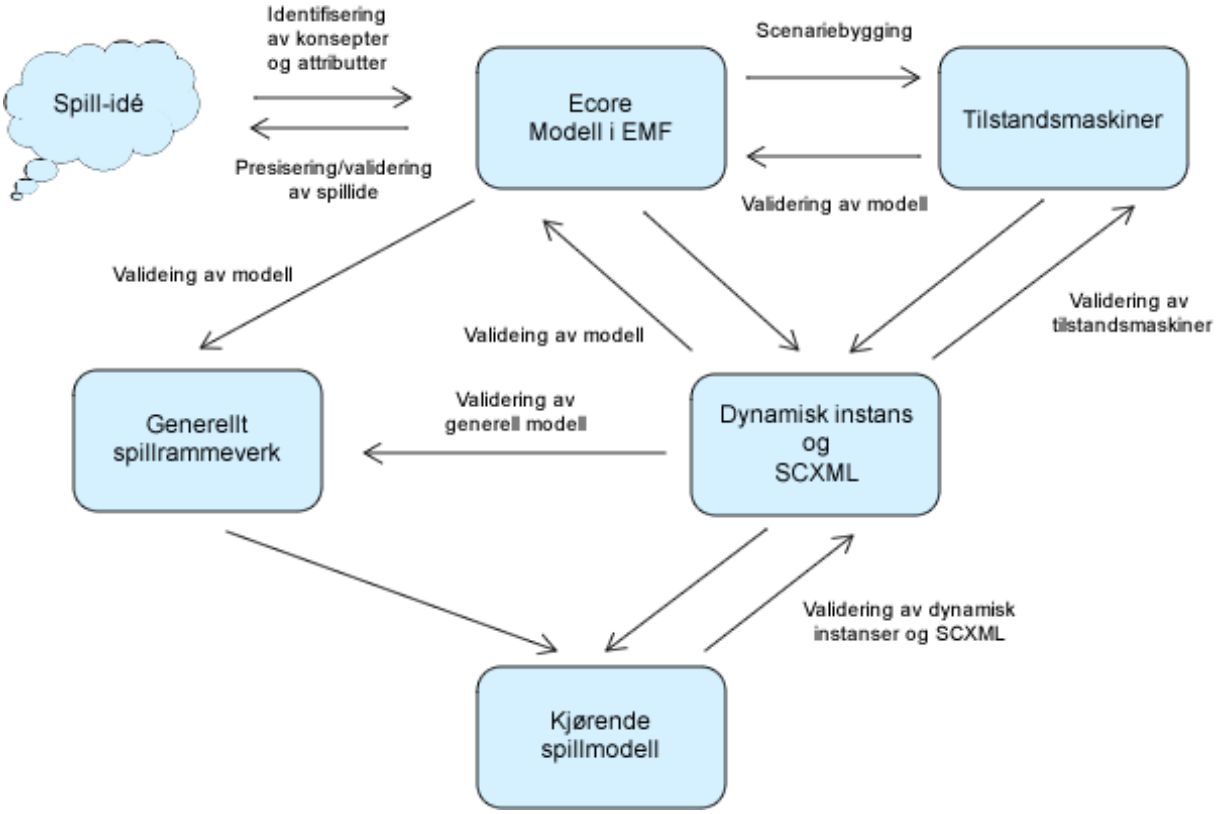
7.1 Vår utviklingsvei

Ettersom mye av teorien og teknologien ikke var kjent for oss før vi begynte med oppgaven, har vi måttet lære oss dette gradvis underveis gjennom skoleåret. Etter vi ble introdusert for teknologien av vår veileder satte vi i gang prosessen som skulle føre til sluttproduktet. Denne prosessen kan sees på som ganske todelt. Det første semesteret gikk med stort sett til å sette oss inn i teori, prøve ut teknologien og til slutt definere spillet som vi skulle utvikle i forbindelse med oppgaven.

Det andre semesteret måtte vi dele tiden mellom å drive implementasjon og å skrive oppgave. Dette arbeidet foregikk på en tilnærmet iterativ måte, spesielt arbeid med implementasjonen av spillet. Vi hadde jevnlige møter med veileder for å presentere arbeidet og få verdifull tilbakemelding og tips til videre arbeid. Implementasjonsfasen bestod for det meste av utvikling av modeller, dynamiske instanser og tilstandsmaskiner.

Figur 25 viser grovt hvordan vi arbeidet med modellene og kom frem til den endelige løsningen, figuren tar ikke hensyn til tilegning av teorikunnskap, kun hvordan selve modelleringsprosessen har foregått. Neste seksjon vil gå nærmere inn på betydningen av arbeidsprosessen som omfatter validering har hatt for det endelige produktet.

Slik det kommer frem i figuren, var det en relativ omfattende mengde arbeid som involverte validering. Mye av arbeidet vi har gjort i år har vært nytt for oss. I tillegg har for eksempel utviklingen av funksjonaliteten for eksekvering av tilstandsmaskiner foregått parallelt av Hallvard Trætteberg. Dette har vært med på å føre til at valideringssykluser har oppstått, som har ledet til at vi har gått tilbake og gjort endringer fordi vi har funnet feil og mangler i tidligere utført arbeid. Samtidig har det ved flere anledninger ført til at tilbakemeldinger fra oss har endret Trættebergs arbeid også.



Figur 25: Arbeidsprosessen

7.2 Validering

En av utfordringene ved å utvikle ved modellering har vært å endre tankemønster fra hva vi er vant til. Måten vi har fått tilbakemeldinger på har vært gjennom ulike valideringssykluser. For hvert nye steg, modell eller skisse av tilstandsmaskin, har vi gått tilbake og sett på modellen det er derivert fra er korrekt eller abstrakt nok. Ofte oppdaget vi at løsninger ikke fungerte i praksis slik vi tenkte at de skulle gjøre. Denne måten å arbeide på har og vært et viktig ledd i prosessen for å få oss til å endre tankemønster. Det er først når man bruker det man tidligere har gjort i en ny kontekst og beveger seg mot noe mer konkret at man forstår hvilken informasjon som ligger der, hvordan man kan hente det frem og bruke det videre. I dette underkapittelet forklares de ulike valideringssyklusene. Fra Figur 25 ser vi at det er identifisert noen valideringssykluser som nå blir presentert nærmere. Dette underkapittelet forsøker å få frem at valideringsprosessene som har pågått, både ubevisst og bevisst har vært en viktig del av utviklingen, og det produktet vi endte opp med til slutt.

7.2.1 Mellom modell i Ecore og tilstandsmaskin

De første tilstandsmaskinene vi lagde var papirbasert. Vi hadde ikke særlig erfaring med tilstandsmaskiner, og heller ikke jobbet noe spesielt med tilstandsmaskiner siden første året på universitetet. Resultatet var at vi lagde en slags mikstur av pseudokode kombinert med vanlig tilstandsmaskinnotasjon (en figur med eksempel). Dette resultatet er en konsekvens

av at vi tenker mer programmeringssentrisk. Vi hadde enklere for å se løsninger på problemer hvis vi kunne kode litt logikk samtidig.

Utgangspunktet for tilstandsstandsmaskinen var arbeid utført i to modeller, vår zombiemodell og Trættebergs generelle modell. Tilstandsmaskinen skulle beskrive hendelser fra vår originale spillidé. Vi oppdaget mange feil i modellen, enten ting som vi hadde oversett eller elementer som var feilmodellert. Disse feiloppdagelsene førte til at vi gikk tilbake til modellen og gjorde endringer, for så å prøve ut tilstandsmaskiner på nytt. Denne valideringssyklusen fjernet mange av de tidlige feilene i modellen og ble gjentatt mange ganger i en periode på et par måneder i oppstartsfasen på oppgaven.

7.2.2 Mellom modell og dynamisk instans/SCXML

Med tiden og tidvis også parallelt med de første tilstandsmaskinene flyttet vi noe av fokuset videre på dynamiske instanser. Dynamiske instanser gir oss enkel mulighet til å teste klasser og referanser som vi har modellert, da det er først med dynamiske instanser at vi begynner å nyttiggjøre oss av relasjoner mellom klassene.

Gjengangere som vi oppdaget her var blant annet duplisert funksjonalitet, kardinalitetsfeil og tilgangsfeil. Duplisert funksjonalitet var gjerne modellert både i den generelle modellen og vår spillmodell. Kardinalitet gjelder mengder. Ofte trenger vi mer enn en klasseinstans, og noen ganger hadde ikke vår modellering tatt hensyn til det. Tilgangsfeil omhandler at vi ikke hadde tilgang til å opprette instanser av klasser der vi innså i ettertid at vi burde opprette klasseinstansene, eller at vi hadde glemt å angi riktig eierskap av klasseinstanser.

Den dynamiske instansen er utgangspunktet for arbeidet med den andre utgaven av tilstandsmaskiner, SCXML. SCXML er en XML-basert notasjon for å beskrive tilstandsmaskiner, og Eclipse kan utvides med funksjonalitet for å kjøre slik SCXML-kode. Vi skrev ikke tilstandsmaskinene våre direkte som SCXML-kode men benyttet en notasjon utviklet av Trætteberg, som med hjelp av Xtext blir konvertert til SCXML. Tilstandsmaskinene arbeider i konteksten til dynamiske instanser og er derfor avhengig av en dynamisk instans for å virke. Tilstandsmaskinen vil da operere gjennom endringer som skjer på data i den dynamiske instansen, for eksempel kan en hendelse trigges ved at en variabel blir endret.

Arbeid på dette nivået førte til at vi oppdaget fler elementer med den opprinnelige modellen som måtte endres. Siden det er stadig behov for å mellomlagre data i variabler i tilstandsmaskiner, for å trigge hendelser, måtte vi innføre noen nye variabler i modellen. Som en teknisk begrensning ble det også klargjort for oss at vi ikke kunne lage en tilstandsmaskin for selve zombiespillet (klassen ZombieGame, lag figur). Fordi vi ikke kunne lage tilstandsmaskin her måtte vi innføre en ny klasse opprettet fra denne klassen som pekte tilbake på spillklassen, fordi vi innså at vi måtte kunne manipulere variabler i denne klassen for å kunne lage en omfattende nok tilstandsmaskin for hele spillet vårt.

7.2.3 Mellom vår spillverden og generell modell

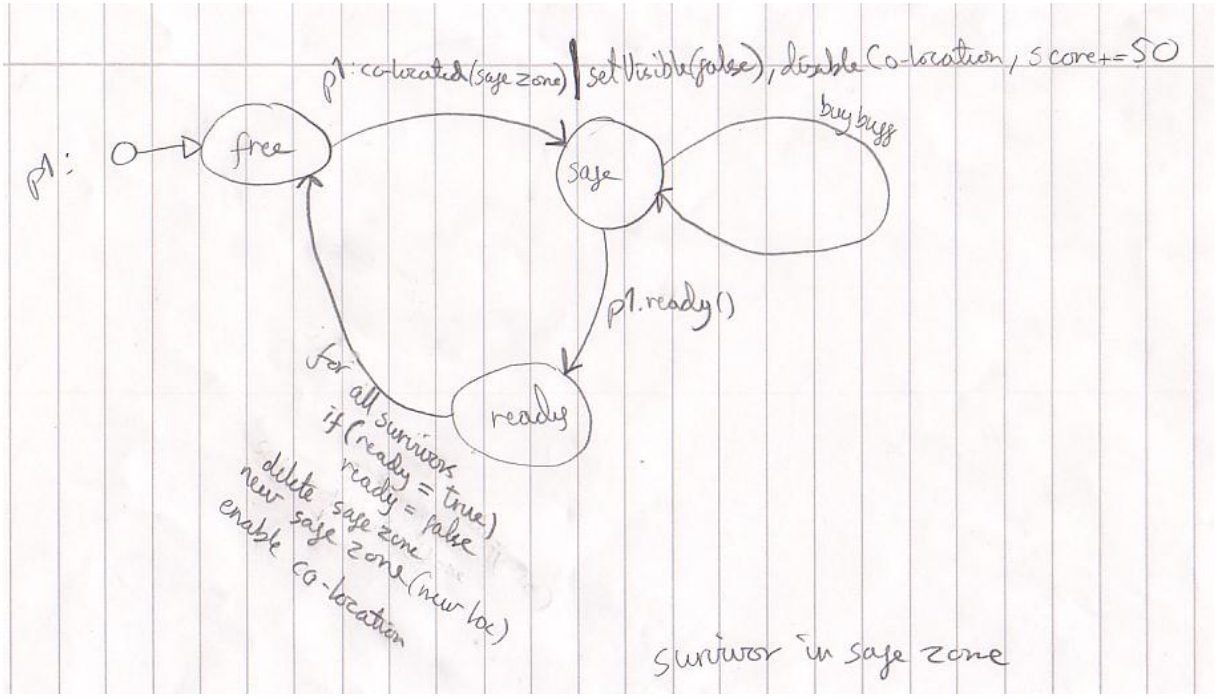
Vårt arbeid med å flette modellenes funksjonalitet til en spillinstans hjalp til med å validere innholdet i den generelle modellen ved at vi tok i bruk funksjonalitet derfra og var derfor i en god posisjon til å oppdage og identifisere problemer, men også ting som virket som det skulle, i den generelle modellen. Den generelle modellen utviklet seg stadig gjennom dialog mellom oss og Trætteberg, samt andre som jobbet med prosjekter i PlayTrd.

Ved å samle den generelle modellen og vår zombiemodell får vi den funksjonaliteten som trengs for å kjøre spillet vårt som instans, eller dette er det ideelle utfallet man så for seg ville komme etter å ha samlet modellene. I realiteten var det mye arbeid og tid som gikk med når vi først valgte å skille ut funksjonaliteten i to modeller, hvor vi selv ikke hadde ansvaret for den ene av de to, for så senere å samle funksjonaliteten. Siden de to modellene ble utviklet parallelt i starten hendte det også at en del av funksjonaliteten overlappet hverandre i starten. Det var også slik at elementer fra vår modell ble flyttet inn i den generelle modellen fordi konseptet ble ansett som abstrakt nok til at det var alminnelig å anta at flere fremtidige spill ville benytte seg av samme funksjonalitet.

7.3 Eksempler fra papirbaserte tilstandsmaskiner

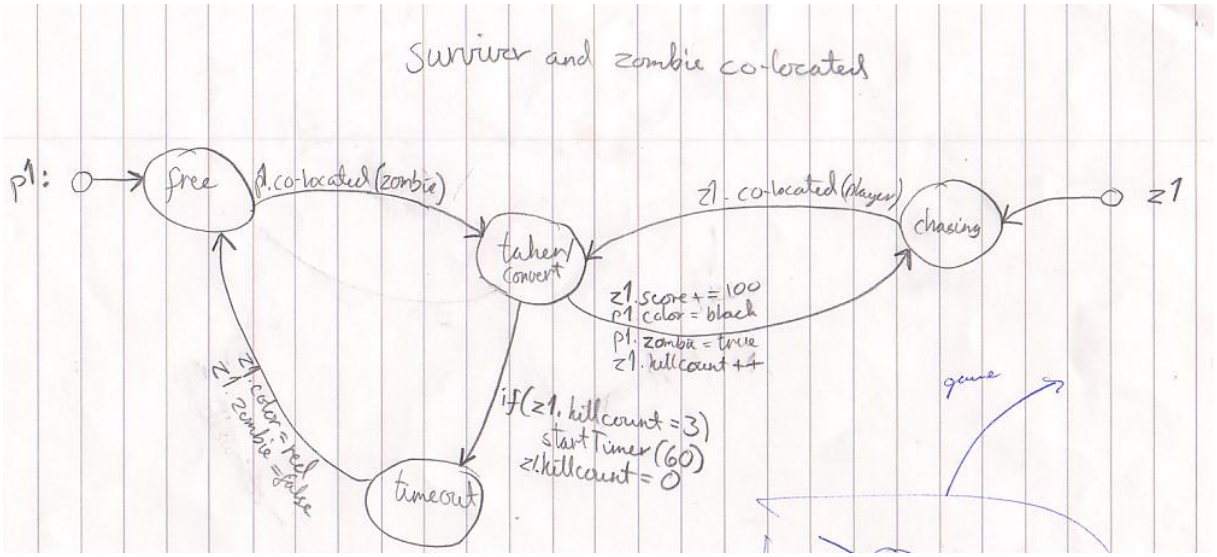
Dette underkapittelet skal inneholde noen eksempler fra papirbaserte tilstandsmaskiner og vise arbeidet med utprøving og validering av modeller.

Validering av Ecore-modell til zombiespillet ble en gjenganger. Som tidligere forklart tok vi utgangspunkt i ideen om zombiespillet og identifiserte hendelser med betingelser og beskrev tilstandsdiagrammer på papir. Denne prosessen kan man kalle valideringsprototyping da det skiller seg fra papirprototypingsbegrepet brukt i brukersentrert design for å måle brukergrensesnitt. Denne prosessen måtte vi gjøre opptil flere ganger for flere tilstandsmaskiner før vi endelig ble fornøyd og følte vi kunne ta steget videre. I dette underkapittelet ønsker vi å sette litt lys på dette papirbaserte arbeidet som ellers ville forsvunnet i søppelkurven ved innlevering. Vi fokuserer ikke på å vise tilstandsmaskiner som er riktig modellert, men prøver heller å vise i kombinasjon med underkapittelet som omhandler sluttresultatet, at dette har bidratt i prosessen. Vi har prøvd og feilet like mye som vi har gjort ting riktig.



Figur 26: Eksempel 1

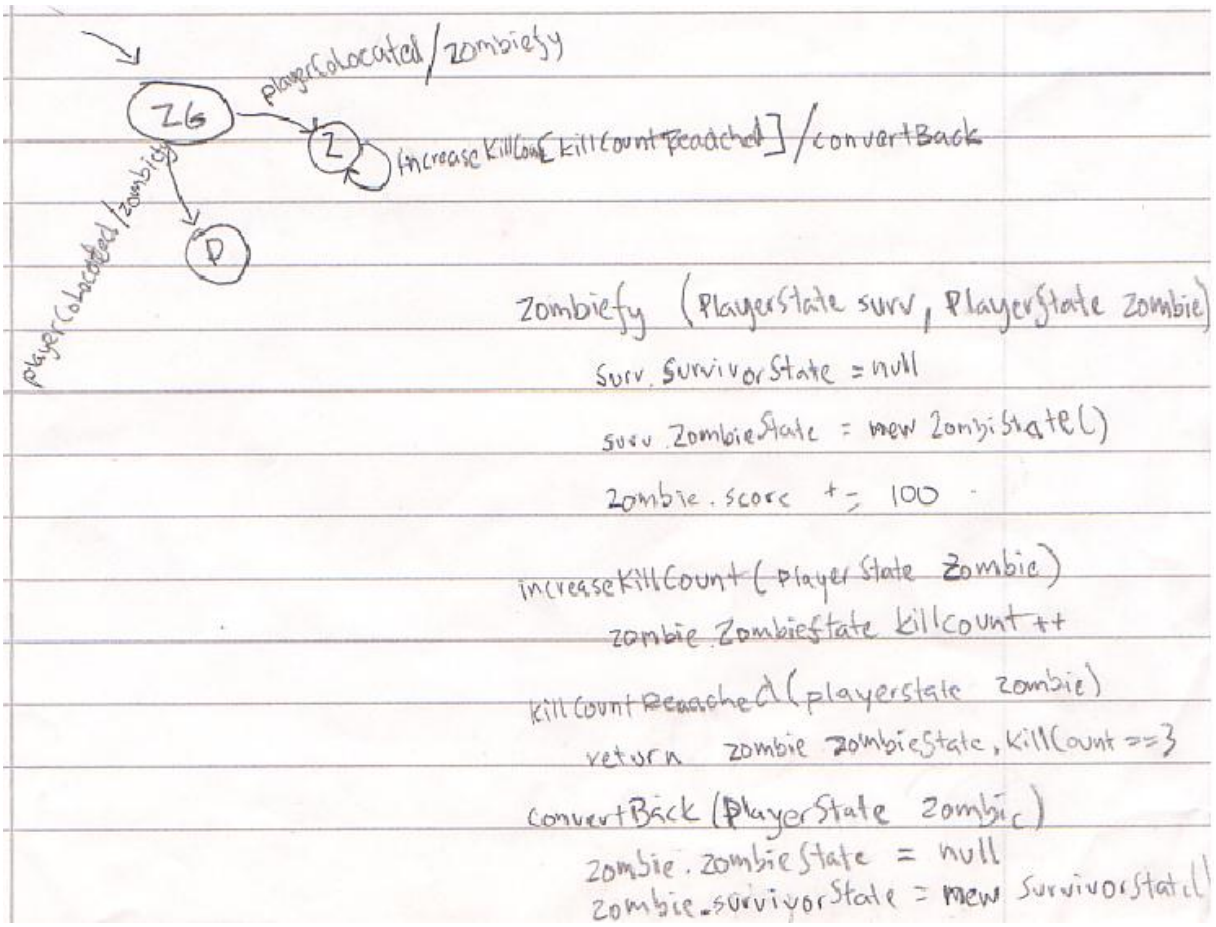
Figuren ovenfor viser tidlig arbeid med å definere atferden til en overlevende når han når en trygg sone og benytter anledningen til å kjøpe ting mens han er i den trygge sonen. Til slutt er det logikk som prøver å si at når alle overlevende er klar for å forlate den trygge sonen så opphører sonen å eksistere og blir erstattet med en ny sone et annet sted.



Figur 27: Eksempel 2

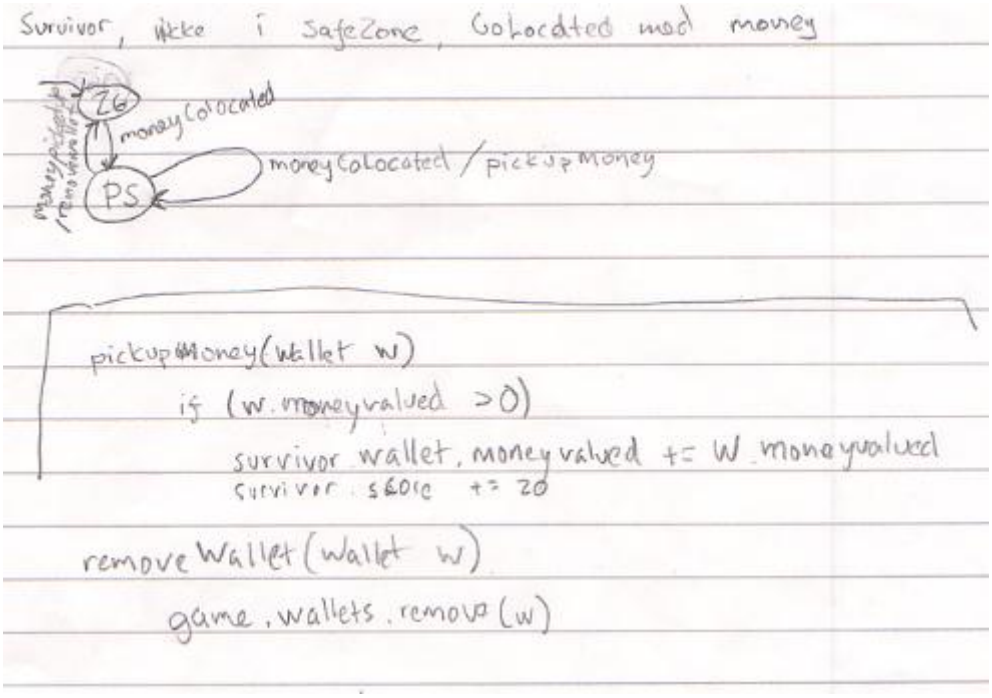
Denne figuren prøver å modellere at to spillerobjekter blir samlokalisert. En overlevende og en zombie. Vi kan innse nå i ettertid at notasjonen er grunnleggende feil, spesielt siden vi her faktisk har prøvd å modellere noe som ligner på to tilstandsmaskiner i ett. Tilstandsmaskinen prøver å samtidig modellere at en zombie blir samlokalisert med en overlevende, og at en overlevende blir samlokalisert med en zombie. Resultatet blir at den

overlevende blir konvertert til zombie og at zombien enten har oppnådd nok ”drap” til å gå tilbake som overlevende eller at kun drapsstelleren øker og zombien må fortsette sin jakt etter nye overlevende.



Figur 28: Eksempel 3

I maskinen ovenfor har vi fokusert mer på logikken som nødvendigvis må inntreffe når hendelser skjer i systemet. Hensikten er igjen å modellere hva som skjer når tilstandsmaskinen får beskjed at to objekter P og Z blir samlokalisert. Spilleren blir konvertert til zombie og zombien får ekstra poeng, og basert på antall konverteringer tidligere utført så blir han kanskje også konvertert tilbake til en overlevende. Dette er forsøkt konkretisert ovenfor gjennom tekst som mest av alt ligner på ren javakode.



Figur 29: Eksempel 4

I figuren ovenfor forsøkes å definere logikken som inntreffer når en spiller blir samlokalisert med en pengesekk. Når spillet mottar beskjed fra samlokaliseringstjenesten om at en overlevende har blitt samlokalisert med en pengesekk, blir verdien av pengesekken lagt til i spillerens lommebok, og pengesekken fjernet fra spillet. Som det fremkommer her så har vi altså gjort en del arbeid med deler av spillkonseptet som ikke endte opp med å være med i vår endelige modell.

7.4 Kvalitetsvurdering

The SEQUAL framework (Krogstie, 2008) er en referansemodell for systemmodellering til å evaluere kvaliteten på modeller. SEQUAL står for semiotic quality framework, og er laget av professor John Krogstie ved NTNU og andre. Å gjennomføre en full SEQUAL-evaluering i denne rapporten ville tatt for mye plass, det er heller ikke relevant, men siden SEQUAL-rammeverket er så omfattende, inneholder det kvalitetsvurderinger på alle områder av denne type modellering. Ettersom denne type utvikling er helt ny for oss, vil det være naturlig å se på pragmatisk kvalitet som fokuserer på effekten modellen har på deltakerne og verden, og tekniske aspekter relatert til verktøyene brukt. Fire aspekter innen pragmatisk kvalitet blir her gjennomgått relatert til vårt arbeid.

Spillmodellen (se kapittel 6.4) har gjennomgått flere redigeringer enn vi kan dokumentere. En av aspektene i pragmatisk kvalitet sier at den menneskelige tolkningen av modellen skal være korrekt i forhold til hva modellen faktisk prøver å uttrykke. Våre første forsøk på spillmodellen var vi rimelig fornøyde med der og da, men i ettertid har vi sett at modellen var langt i fra å være korrekt. I begynnelsen trodde vi det fantes informasjon i modellen som

ikke var der. Selv om dette var en prosess hvor vi på forhånd visste at modellen ville måtte gjennomgå flere redigeringer, trodde vi at innholdet i modellen var mer korrekt enn det viste seg å være. Denne feiltolkningen henger litt sammen med at vi heller ikke hadde blitt helt komfortable med EMF-rammeverket, og denne måten og tenke på. Evalueringen av de tidlige modellene gav oss uansett økt forståelse og større innsikt i verktøyene og mulighetene det gir, selv om ikke modellene var korrekte. Denne kunnskapen kom igjennom tilbakemelding fra Hallvard og valideringsmekanismene nevnt i kapittel 7.2. Det er vanskelig å kvantifisere den pragmatiske kvaliteten til modellen slik den er nå, men både for oss og andre som skal tolke modellen er innholdet i modellene rikere samtidig samt at den er mer korrekt. SEQUAL nevner menneskelig forståelse som et pragmatisk mål, at modellen blir forstått. Sitat: ”Not even the most brilliant solution to a problem would be of any use if nobody is able to understand it”. Vi forstår vår modell godt siden vi har laget den til vårt formål, spørsmålet er vel heller om en ekstern leser skjønner den like godt. For å forstå meningen bak modellen og måten den er strukturert på, må man også skjønne konteksten den er laget i, et eksempel er alle klassene som ender med ”state” og hvorfor disse tilsynelatende tomme klassene ikke er boolske variabler.

Dette leder inn i ett annet aspekt; Måten man tolker verktøyene på skal være korrekt i forhold til hva som er ment å bli uttrykt i modellen. Vi har aldri hatt trøbbel med å forstå metamodellen til ecore figurert i kapittel 4.1, som nevnt har det heller vært spørsmål om hvordan modellen skal være definert for at den skal passe med resten av rammeverket. Vi gav oss ikke før både vi og Hallvard ble fornøyd, dette var når modellen var formalisert og flere generelle konsepter hadde blitt dyttet opp i den generelle domenemodellen. Parallelt jobbet vi med tilstandsmaskinprototyping på papir for hånd, en metode for å avdekke mangler i modellen og validere dens innhold, nærmere beskrevet i kapittel 7.3. Sitat SEQUAL; ”A conceptual model can be difficult to comprehend due to the formality or unfamiliarity of the modeling language used, the complexity or size of the model, or the effort needed to deduce important properties of it.” Modellen ble mer korrekt samtidig som vi fikk økt kunnskap om domenemodellering. Et annet meget relevant aspekt er at deltakerne lærer basert på modellen. Vi kan i dag se på en gammel versjon av modellen og peke på ting som kan utelates, endres eller legges til, altså forbedringer. Dette skyldes først og fremst at vår kunnskap om temaet har økt i takt med modellforbedringene.

Det siste pragmatiske kvalitetsaspektet sier at hvis modellen har til hensikt å føre med seg forandring, blir domenet forandret, helst i en positiv retning relatert til modelleringsmålene. Vi har ved å arbeide zombiespillet identifisert konsepter som naturlig hører til domenet og ikke spesifikt til vårt spill, dermed har vårt arbeid ført til domeneforandringer, så dette kvalitetsaspektet kjenner vi oss godt igjen. Totalt sett er det flere ting som peker på at modellene våre har høy pragmatiske kvalitet, modelleringen har hatt en positiv effekt tatt i betraktning tiden prosessen har tatt.

7.5 Sluttresultat

Dette underkapittelet skal oppsummere resultatene vi oppnådde gjennom skoleårene. Som forklart i de siste kapitlene kan arbeidet vi har gjort innenfor utvikling oppsummeres innenfor følgende områder:

- Modellering
- Papirbaserte tilstandsmaskiner
- Dynamiske instanser
- Tilstandsmaskiner

Vi vil nå bruke de siste avsnittene i dette kapittelet til å oppsummere ståstedet vårt innenfor disse områdene ved innlevering. I neste kapittelet vil vi komme tilbake på våre synspunkter når vi starter med vår diskusjon av resultatene våre.

I arbeidet med modellering har vi hatt påvirkende innflytelse på spesifikasjonen av den generelle modellen utviklet med hensikt å drive flere fremtidige spill i PlayTrd prosjektet. Samtidig har vi også utviklet en egen modell med den hensikt å prøve ut teknologi og verktøy som er presentert i forbindelse med oppgaven. Modellen vår er basert på et spillkonsept som vi utviklet tidligere i semesteret, men forenklet med det hensynet at vår oppgave var ikke å teste ut et spill med stor kompleksitet hva logikk angår.

De papirbaserte tilstandsmaskinene innehar ikke noen produksjonsbar kvalitet, men ble heller et verktøy for å teste modellering og dynamiske instanser. Arbeidet som vi utførte med papirbaserte tilstandsmaskiner tvang oss til å tenke videre i prosessen, og fokusere mer på sammenkobling og andre egenskaper slik som eierskap av objekter i et litt annerledes lys en tidligere.

Dynamiske instanser er den naturlige veien videre når man utvikler modeller. Dynamiske instanser hjelper til å validere modellene ved å gi instansierte objekter ut i fra klassene i modellene. Dessuten er dynamiske instanser utgangspunktet for tilstandsmaskiner, som også krever instansierte objekter for eksekvering. Dynamiske instanser lagres i et XML basert format og kan dermed enkelt vedlikeholdes både gjennom de introduserte verktøyene og enkelt tekstlig i en editor dersom det skulle være nødvendig. En eksempelinstans trenger ikke involvere mange linjer med XML-kode.

Arbeidet med tilstandsmaskiner utviklet gjennom DSL'et spesifisert av Hallvard Trætteberg ble som tidligere nevnt hemmet av at vi ikke fikk kommet i gang med å utvikle tidlig nok. Dette resulterte i at vi ikke fikk arbeidet oss frem til et resultat med denne biten som vi så oss fornøyd nok med til å spesifisere veldig inngående her. Derimot så har vi et eksempel i kapittel 4.6 som viser at teknologien langt på vei fungerer, selv om vi også har nevnt at vi kom ut for en del småfeil i implementasjonen som var med på å forsinke arbeidet vårt på denne fronten ytterligere.

8 Diskusjon og fremtidig arbeid

Teori, teknologi og resultater er nå presentert. Ved å bruke konsepter fra teorien har vi vist at det er mulig å benytte de valgte verktøyene til å drive modelldrevet utvikling. Det betyr at vi har gjort oss opp noen tanker rundt ulike faser av arbeidet som har foregått gjennom skoleåret. Dette kapittelet vil ta for seg nettopp våre tanker og meninger rundt sentrale emner slik som metode, verktøy, våre resultater og samarbeid gjennom skoleåret.

Som avslutning presenterer vi våre tanker rundt det fremtidige arbeidet som gjenstår for de som fortsetter med PlayTrd-prosjektet og hvilke erfaringer de kan dra nytte av fra vårt arbeid.

8.1 Metode og verktøy

Metode som vi skal diskutere her kan sees på som to relativt forskjellige ting. Først kan det være de teoretiske prinsippene rundt utviklingsmetodikk, som for oss vil si modellbasert utvikling (MDE) og prototyping. I tillegg har metode en betydning for oss som omhandler arbeidsprosessen mer slik den faktisk foregikk, som blir presentert i kapittel 5.1. Det er ingen tvil om at vi bedrev modelldrevet utvikling siden vi faktisk primært jobbet med modellering gjennom skoleåret. Fokuset på prototyping har vært mer uklart. Vi assosierer gjerne prototyping med å utføre tester med personer som ikke er direkte involvert i utviklingen av produktet, og dette har vi ikke gjort. Det at vi ikke har utført noe testing kan begrunnes på flere måter. Først og fremst inngår ikke produkttesting i våre forskningsspørsmål, og for det andre så ville tid brukt på dette kunne gått utover kvaliteten på en del av det andre arbeidet. Uansett kunne vi rimelig enkelt laget en papirprototype som hadde demonstrert den funksjonaliteten vi ønsket. Utfordringen ville fremdeles vært å omsette denne papirprototypen til et realisert produkt gjennom modellbasert utvikling. Om utgangspunktet var et spillkonsept eller en papirprototype, eller begge, anser vi ikke nå lengre som så veldig viktig.

Forskningsspørsmålene som ble stilt i begynnelsen av oppgaven er tett knyttet til metode og verktøy. Vi spurte om modellbasert utvikling egner seg som utviklingsmetode til å lage spill og vi spurte om verktøyene vi bruker egner seg, og i så fall i hvor stor grad. Det er selvsagt vanskelig å kvantifisere graden av brukbarhet, men gjennom skoleåret har vi dannet våre subjektive meninger som uttrykke i denne rapporten. Før vi sier om de er negative eller positive, vil vi gjerne komme med noen refleksjoner og eksempler fra skoleåret som bygger opp under våre meninger.

Vårt første møte med teknologi i prosessen var med en spesialutgave av Eclipse, Eclipse Modeling Tools, som inneholdt funksjonalitet for å grafisk opprette Ecore-modeller. Med grunnleggende kjennskap til Eclipse var det en enkel sak å sette opp og komme i gang med de første famlende forsøkene på å modellere spill. Vi oppdaget raskt at å modellere med den

grafiske editoren føltes veldig kjent. Vi hadde begge erfaring med UML, og funksjonaliteten i den grafiske editoren i Ecore kunne vært plukket rett ut i fra klassediagram i UML. Dette vil vi si er en fordel, for det gir en myk og fin oppstart, i stedet for å skremme bort potensielle brukere med noe som fremstår som utfordrende å mestre. Etter hvert som man dykker litt dypere i Ecore og forlater den grafiske editoren lærer man mer om funksjonaliteten som skiller UML fra Ecore. Med en spesifisert modell i Ecore er det en enkel sak å generere ferdig javakode basert på modellen. En fin funksjonalitet egnet til å vise nye brukere at man kan spare tid ved å modellere aspekter av logikken i stedet for å starte rett på programmeringen. For vår del derimot er ikke denne funksjonaliteten spesielt nyttig, for vi trenger ikke å se noe kode i denne delen av vår oppgave. Vi hadde derimot stor nytte av en annen funksjon som lager dynamiske instanser av klasser. Disse dynamiske instansene utfører funksjonalitet på linje med kjørende kode, det vil si at det gir oss anledning til å teste klassene vi har modellert i praksis. En enkel og grafisk funksjonalitet som gjør det enkelt å drive validerende arbeid av modellen.

Dynamiske instanser er også springbrettet til videre funksjonalitet. Vi fikk levert en spesialtilpasset ny pakke av Eclipse Modeling Tools samlet med noen eksisterende ekstrapakker og noen nyutviklede pakker. Hensikten med denne samlepakken var å koble sammen dynamiske instanser med tilstandsmaskiner utviklet i et DSL som oversatte en nyutviklet syntaks til XML basert SCXML syntaks som Eclipse kan eksekvere med ekstra funksjonalitet lagt inn ved hjelp av tilleggspakker. Her begynner problemene våre også å oppstå. Denne samlepakken var utviklet ganske raskt og inneholdt flere feil som forsinket arbeidet vårt litt. Men det største problemet var vel kanskje at som DSL'er flest var språksyntaksen en utfordring. Mangel på syntaks-fremheving i kombinasjon med at det ikke var vi som hadde definert språket gjorde arbeidsforholdene for oss dårligere, spesielt innenfor tidsrammene vi hadde tilgjengelig når vi først fikk satt i gang å jobbe med tilstandsmaskinene. Feil befant seg ikke nødvendigvis i vår syntaks, men ofte i implementasjonen av språket, da vi stedvis hadde tenkt og gjort litt annerledes en Trætteberg da han gjorde forarbeidet med implementasjonen. DSL-syntaksen føltes også litt mangelfull for oss som er litt mer vant til programmeringsspråk slik som Java. Man må tenke at man har flere parallelle tilstandsmaskiner, og den eneste måten å trigge endringer i parallelle tilstandsmaskiner er å forandre variabelverdier som den andre maskinen lytter etter endringer på. En annen måte å tenkt på i forhold til Java hvor man kaller opp andre metoder når man vil bringe logikken videre.

Ved å splitte funksjonalitet i to modeller som vi har vært med på å gjøre får man følelsen av at vi har vært med i oppstarten på å lage noe som er klart for å bli tatt med videre til nye høyder gjennom fremtidig arbeid. Det ligger nå en generell modell som har mye grunnleggende funksjonalitet og implementasjonen av DSL-syntaks med eksekvering av tilstandsmaskiner er laget, om enn ikke helt feilfri enda. Dette vil være med på å minke arbeidsmengden til de som eventuelt vil jobbe videre med liknende oppgaver.

Spillet er ikke enda i nærheten av å kunne kjøres i en ”ekte” pervasive verden med testbrukere. Veien dit er enda et stykke frem i tid. Zombiespillet var i utgangspunktet et verktøy for å nå dette målet, men underveis så vi at spillet ble mer enn et verktøy. Som nevnt brukte vi litt tid på å omstille tankesettet fra programmeringssentrisk til modelldrevet uten fokus på programmering, og prosessen fra ide til spillbeskrivelse har vært en god måte for å få erfaring av praksis. For å få et spill i dette rammeverket til å fungere kjørende må modellen og scriptingen være meget presis, det er blir noe helt annet å lage et ER-diagram eller UML-klassediagram. Modellen i EMF skal faktisk brukes som en del av implementasjonen, derfor har validering og presisering tatt opp mye tid. Den stadige valideringen og oppdateringen av modellen har ikke bare vært til fordel for å få ting til å virke, men har også bidratt til separeringen av funksjonalitet mellom den generelle modell og vår spillmodell.

For å komme tilbake til forskningsspørsmålene og etter hvert oppsummere standpunktene våre gjengir vi først de to spørsmålene:

- Egner modellbasert utvikling seg til utvikling av spill?
- I hvor stor grad er verktøyene vi benytter gjennom utviklingen enkle i bruk og nyttig?

Vi mener vi har vist gjennom vår eksempelimplementasjon av spillkonseptet Zombies vs. Survivors at modellbasert utvikling kan brukes til utvikling av spill. Men det er ikke essensen av spørsmålet. Vi spurte om det egner seg til utvikling av spill. Vi mener at med alminnelig kunnskap med UML enkelt kan komme i gang med å modellere spill på en måte som fritar deg for mye ekstraarbeid som programmering kan medføre gjennom overkomplisering av logikk. Innenfor rammene til PlayTrd med lokasjonsbevisste, sosiale spill mener vi at modellbasert utvikling egner seg godt.

Når det gjelder verktøyene så synes vi Eclipse tilbyr en spennende plattform som er rik på funksjonalitet, på godt og vondt. Tidvis er det vanskelig å finne riktig funksjonalitet og det har vært mange anledninger hvor vi måtte rådspørre Hallvard Trætteberg for å finne veien videre da vi ikke klarte å finne relevant hjelp på internett heller. Spesielt har strevd mye under implementasjonen av tilstandsmaskiner, da de tilbakemeldingene vi har fått i systemet når vi har prøvekjørt tilstandsmaskiner har vært svært varierte og vanskelig å tolke. Alt fra feil i bakomliggende kode til frysninger av programvaren har oppstått.

Vi evnet som tidligere nevnt, av flere årsaker, ikke å ferdigstille en fungerende tilstandsmaskin av vårt spillkonsept for å validere spørsmålet vårt om verktøyene fungerer og er nyttig. Derimot har vi vist gjennom kapittel 4.6 at det finnes en annen implementasjon som, til den grad at også det er et spill som benytter den samme teknologi, viser at tilstandsmaskinimplementasjonen virker. Denne implementasjonen mener vi egner seg til å beskrive en begrenset mengde logikk, da det fort blir uoversiktlig og tungvint når man må skrive mye logikk i en DSL-syntaks som er særegen for denne implementasjonen og ikke gir spesielt god feilinformasjon.

Å jobbe med Ecore-modeller bygger videre på det som for de fleste er velkjent modelleringsterminologi, og man har mulighet til å spesifisere modellen både grafisk, delvis grafisk og tekstlig. Det er liten tvil om at dette verktøyet er både velegnet og brukervennlig. Med gode modelleringskunnskaper så kan man raskt bygge gode datamodeller for spill, men dette er som kjent ikke hele jobben. For å implementere logikken i spill må man bygge tilhørende tilstandsmaskiner til modellene. Det er som tidligere nevnt liten tvil om at implementasjon virker, og vi har også vist at den egner seg til løsninger av denne typen. Men brukervennligheten er lite tilfredsstillende i den nåværende versjonen av implementasjonen. Vi har vært helt avhengige av den tekniske kunnskapen til Trætteberg når vi har stått fast i arbeidet med tilstandsmaskiner, og vi føler ikke at vi har fått like mye kunnskap i denne biten av arbeidet som i arbeidet med modelleringen, og som kjent måtte vi legge fra oss dette arbeidet til slutt da det viste seg at vi ikke kunne ofre mer tid på det. Samtidig tviler vi ikke på at vi har spart voldsomme mengder med tid vi egentlig måtte ha brukt til å jobbe med direkte SCXML-tekst dersom ikke denne implementasjonen hadde eksistert, og det teller definitivt som en positiv side av brukervennligheten til implementasjonen for tilstandsmaskiner.

Vår konklusjon blir derfor at vi mener at modellbasert utvikling egner seg til utvikling av spill innenfor rammeverket til prosjektet, og vi mener også at verktøyene som brukes er nyttige, men at spesielt implementasjonen laget for å kjøre tilstandsmaskiner har en del å gå på før vi kan kalle verktøyet for brukervennlig.

8.2 Samarbeidsprosessen

Det er ikke mange studenter som velger å arbeide sammen på en masteroppgave. Oppgaven er det største arbeidet vi kommer til å gjøre i skolesammenheng og tanken på å legge ansvar på andre skremmer mange. Likevel valgte vi å arbeide sammen, og i denne seksjonen redegjør vi vårt valg og diskuterer hvilke konsekvenser dette har hatt i både positiv og negativ retning.

Hovedbegrunnelsen til at vi har arbeidet sammen er pga. at vi begge fant oppgaven nyskapende og utfordrende, og at det i oppgavebeskrivelsen sto at dette er en type oppgave som kan passe for to stykk. I tillegg kjente vi hverandre fra før og har erfaring med å jobbe sammen fra prosjekter tidligere i studiet. Alle har hørt skrekkeksempler fra studenter som har skrevet master sammen, blir man uenig er fallhøyden stor om man ikke klarer å løse uenigheten. Vi var uansett sikre på at dette kom til å gå bra og det gjorde det.

Arbeidsfordelingen med praktisk arbeid som ikke innebefatter skriving har fungert godt. Oppgavens natur sørger for at mye av det praktiske arbeidet kan med fordel gjøres sammen. De positive egenskapene ved par – programmering kan i stor grad overføres til modellering, det er ikke arbeidet med å tegne bokser og piler på skjermen man bruker tid på, men hvordan disse skal være strukturert. Selv om vi ikke har hatt problemer med å være to, kan

vi ikke si om det resultatmessig har vært en fordel. Det er vel ikke uten grunn at det i industrien ikke er standard praksis å arbeide to foran en skjerm. Når det kommer til generell forståelse har vi diskutert mye og dette har hjulpet begge hvis vi ikke har vært på samme nivå når vi har satt oss inn i noe nytt, men vi har våre tanker om at en person med modelleringserfaring godt kunne tatt fatt på denne oppgaven alene. Dette må selvsagt sees i lys av den pragmatiske kvalitet (se kapittel 7.4) til modellene som har bidratt til å både gjøre domenet mer komplett og gitt oss mye erfaring. Det er ikke sikkert den positive effekten hadde vært like stor om kun en person hadde arbeidet med det samme prosjektet.

Arbeidsfordelingen med rapportskrivingen har også fungert, men noen få utfordringer har det vært. Selvsagt er det dager hvor en av oss har gjort mer enn den andre, men det jevner seg ut over tid. Når vi for eksempel har hatt flere mulige temaer å skrive om, har vi automatisk byttet slik at vi begge får velge det man har mest lyst til å jobbe med. Samtidig kan det gi en slags falsk følelse av ”trygghet” når man er to og ting går fremover, det er lett å ta for gitt at man kommer i mål fordi man er to som kan arbeide parallelt. Dette stemmer selvsagt ikke fordi begge må kvalitetssikre teksten som blir skrevet og være enige i alle utsagnene. Vi har lest hverandres tekster hvor det har hendt at den ene er uenig i innholdet, og teksten må omarbeides eller skrives på nytt. Konstruktiv kritikk til hverandre har ikke vært ukjent for oss, og nesten uten unntak fører kritikken til forbedring.

8.3 Fremtidig arbeid

Selv om vi nå har lagt ned vår arbeidsinnsats med å implementere deler av vårt spillkonsept, så lever prosjektet PlayTrd videre. Vi vet ikke mye om hvilke oppgaver som blir tilbudt til nye studenter som tar fatt på masteroppgaven, men vi føler oss trygg på at vi har vært med på å gjøre det lettere å starte nye oppgaver i prosjektet. For det første eksisterer nå den generelle modellen som ble gjennomgått i kapittel 6.2. Den inneholder mange ferdigdefinerte elementer for et spill og når den kan kobles sammen modeller og arve egenskaper fra den generelle modellen, bør det være en grei oppgave å komme i gang med å spesifisere nye særegne elementer for nye spill. For det andre finnes det også en implementasjon for å eksekvere spillenes logikk gjennom en DSL-syntaks, som konverteres til SCXML-kode og kan eksekveres i Eclipse. Selv om denne implementasjonen ikke føles helt perfekt enda på grunn av en brukervennlighet vi ikke opplever som helt optimal, er det grunnlag til å tro at denne også kan forbedres og fikses for å gjøre det lettere å arbeide med spillenes logikk også. Begge de nevnte elementene har blitt utviklet i løpet av skoleåret, og selv om det ikke er vi som har implementert noen av dem så har vårt arbeid bidratt til å validere arbeidet gjennom oppdaging av feil og at vi brukte både den generelle modellen og tilstandsmaskinimplementasjonen i vårt arbeid.

Naturligvis må målsetningen til de som fortsetter arbeidet være å ta arbeidet vårt noen steg videre. Dersom det hadde vært vi som skulle gjort et nytt prosjekt, ville vi prøvd å ta med et

spillkonsept videre til en fase hvor vi kunne stått i Trondheim sentrum med mobiltelefoner i vår hand og testet en prototype av spillet. Der skulle vi, enten med å lese av strekkoder med kameraet på telefonen som angir lokasjoner, eller ved å bruke for eksempel GPS kunne testet ut spillet vårt med en testgruppe av brukere. Uansett synes vi å ha vist med denne oppgaven at en del av det grunnleggende arbeidet allerede er på god vei til å komme på plass, og at fremtidig arbeid derfor har et bedre utgangspunkt enn oss. Vi ble kanskje også noe hemmet av vår begrensede erfaring med modelleringsdrevet utvikling, så det kan være smart å rekruttere blant de som har mer erfaring. Samtidig er det ikke bare modellering som skaper utfordringer for å lage prototyper som kan testes på mobiltelefoner. Erfaring med mobile plattformer er også en fordel. Å finne måter å koble modelldrevne spill og mobilplattformer er en oppgave som ligger i fremtiden. Vi føler det er en naturlig målsetning for det videre arbeidet å få testet spill på en mobil plattform i en mer realistisk setting.

Vi tror ikke nødvendigvis at spillkonseptet vårt er noe som kommer til å bli arbeidet videre med, men det føles ikke som noen nødvendighet heller, for det finnes ikke noen grenser for hvilke spillkonsepter som kan utvikles ut i fra konseptene som den generelle modell gir og med fokus på å skape lokasjonsbevisste sosiale spill.

Referanser

iPhonical. (2010). Retrieved Mai 07, 2010, from http://code.google.com/p/iphonical/

What is Eclipse and the Eclipse Foundation. (2010). Retrieved Mars 10, 2010, from http://www.eclipse.org/org

Wikipedia GURPS. (2010). Retrieved Mai 31, 2010, from http://en.wikipedia.org/wiki/GURPS

Wikipedia Left 4 Dead. (2010). Retrieved April 20, 2010, from http://en.wikipedia.org/wiki/Left_4_dead

Wikipedia SOA. (2010). Retrieved Juni 13, 2010, from http://en.wikipedia.org/wiki/Service-oriented_architecture

Wikipedia Wii Fit. (2010). Retrieved April 20, 2010, from http://en.wikipedia.org/wiki/Wii_fit

Amory, A., Naicker, K., & Vincent, J. A. (1999). The use of computer games as an educational tool: Identification of appropriate game types and elements. *Britis J. of Educational Technology, 4*(30), pp. 311-321.

Apache Commons. (2010). *Commons SCXML*. Retrieved Mai 09, 2010, from http://commons.apache.org/scxml/index.html

Apple. (2010). *iPhone Developer Program License Agreement*. Retrieved Mai 07, 2010, from http://developer.apple.com/iphone/terms/program/iphone_standard_agreement_20100408.pdf

Berg Insight. (2010). *Gps and mobile handsets.* http://www.berginsight.com/ReportPDF/Summary/bi-gps4-sum.pdf.

Broll, W., Ohlenburg, J., Lindt, I., Herbst, I., & Braun, A.-K. (2006, Oktober 30-31). Meeting technology challenges of pervasive augmented reality games. *Netgames*.

Digi.no. (2010). *Hver tiende nordmann har iPhone*. Retrieved April 20, 2010, from http://www.digi.no/838026/hver-tiende-nordmann-har-iphone

Hawryszkiewycz, I. (2001). *Introduction to systems analysis and design.* Pearson.

IDI. (2010). *Masteroppgaver i datateknikk*. Retrieved April 20, 2010, from http://www.idi.ntnu.no/education/prosjektoppgaver.php?seksjon=&gruppe=&fagl_id=&emnekombinasjon=&dato_reg=2&utvalg=diplom&submit=Vis+oppgaver

Krogstie, J. (2008). *Model-driven development and evolution of information systems: A quality approach (draft).*

MagerKurth, C., Cheok, A. D., Mandryk L., R., & Nilsen, T. (2005, July). Pervasive Games: Bringing Computer Entertainment Back To The Real World. *ACM Computers in Entertainment, 3*(3).

Malone, T. W. (1981). Toward a theory of intrically motivating instruction. *Cognitive Science, 4*(13), pp. 333-369.

Mellor, S. J., Clark, A. N., & Futagami, T. (2003). Model-Driven Development. *IEEE Software*, pp. 14 - 18.

OMG. (2010). *BPMN Specification*. Retrieved Juni 13, 2010, from http://www.bpmn.org/

Scneiderman, B., & Plaisant, C. (2005). *Designing the user interface.* Pearson.

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software*, pp. 19-25.

Sharp, Rogers, & Preece. (2007). *Interaction Design, 2nd Edition.* Wiley.

Song, H. (2009). *Eclipse Bugs*. Retrieved Mai 07, 2010, from Ported EMF core to Android Platform: https://bugs.eclipse.org/bugs/show_bug.cgi?id=296770

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2009). *Eclipse Modeling Framework, Second Edition.* Addison Wesley.

The Eclipse Foundation. (2010). *Xtext*. Retrieved Mai 29, 2010, from http://www.eclipse.org/Xtext/

Trådløse Trondheim. (n.d.). Retrieved April 20, 2010, from http://tradlosetrondheim.no/

Wikipedia. (2010). *Massively multiplayer online role-playing game*. Retrieved April 20, 2010, from http://en.wikipedia.org/wiki/Mmorpg

Wikipedia GSM Mobile Phone Tracking. (n.d.). Retrieved Mai 23, 2010, from http://en.wikipedia.org/wiki/Mobile_phone_tracking

Wikipedia WOW. (n.d.). Retrieved Mai 31, 2010, from http://en.wikipedia.org/wiki/World_of_warcraft