HÖGSKOLAN SKÖVDE

Institutionen för kommunikation och information
Examensarbete i datavetenskap 20p (kursnamn+poäng)
C-nivå
Vårterminen 2005

Dynamiska dialoger i dataspel

Benny Peczek

Dynamiska dialoger i dataspel

Examensrapport inlämnad av Benny Peczek till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Mikael Thieme.

2007-06-03

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Dynamisk dialog i dataspel

Benny Peczek

Sammanfattning

Detta är ett projekt som diskuterar och framställer en arkitektur för dynamisk dialog i dataspel. Den är skapad för att kunna ge mer spelarinteraktion i datarollspel i form av större frihet när det gäller hur en spelare vill spela sin karaktär samtidigt som den även ska ge bättre respons från NPC:er, just i ämnet dialoger. Beroende på vad och, framför allt, *hur* du som spelare säger något ska du få olika känslomässiga respons från dem. De ska kunna bli irriterade, hata, gilla och älska dig, med mera, samtidigt som de även ska kunna bli rädda för dig. Utöver detta har de även försetts med individuella minnen som tillåter dem att komma ihåg vad du har sagt och hur många gånger du har sagt det, vilket i sin tur tillåter att du får olika respons ju fler gånger du säger samma sak.

Nyckelord: Dataspel, Rollspel, Dialog, NPC, AI, Programmering, Speldesign, Tillståndmaskiner, Interaktion i spel

Innehållsförteckning

1 Introduktion

Den här rapporten hamnar någonstans i gränslandet mellan design och programmering. Den tar upp bristerna när det gäller rollspel jämfört med datarollspel och hur just ordet ”roll” ofta förbises till allt för stor grad i den senare delen (Colantonio & Avellone, 2003). I detta fall inriktar den sig mer specifikt på området dialog, som utåt sett är ett designmässigt problem för den här typen av spel men som ändå kan finna sina grunder i en brist på tillräcklig teknik. Syftet är inte att försöka inte hitta den ultimata lösningen utan snarare att ta ett steg i rätt riktning och med hjälp av nya typer av tekniktillämpningar låta spelaren få större kontroll över sin karaktär i en spelvärld och samtidigt få uppleva bättre respons. Med det i åtanke kan sägas att den här rapporten riktar sig lika mycket till programmerare som till designers.

Vad är då problemet med dialog i dag? De som någon gång i sitt liv har spelat ett riktigt rollspel bör känna till den typen av kontroll en spelare får över sin karaktär i sociala situationer med andra karaktärer och hur han eller hon själv står för sin karaktärs utveckling i form av just interaktion med andra. Det är nämligen du som spelare som *är* karaktären och det är alltid du som bestämmer hur han eller hon agerar och reagerar. I dataspel används ofta låsta dialoger som inte tillåter mycket i form av personliga sätt att uttrycka sig på och ännu mindre i form av känslomässig respons från andra karaktärer i spelvärlden. Under de senaste 20-30 åren har vi sett flera spel som försökt utveckla just den sociala biten i datarollspel men även de allra främsta har långt kvar att gå (DeSmedt & Loritz, 1999; Schwab, 2004). Det här arbetet försöker därför att föra utvecklingen vidare från den punkt där spel så som *Fallout 2* (Black Isle Studios, 1998) och *Vampire: The Masquerade – Bloodlines* (Troika Games, 2004) stannade. Spelare ska själva få välja hur de vill att sin karaktär ska agera och hur han eller hon ska bli sedd av andra och designers bör få möjligheten att realisera detta i spel. Inte till punkten av total frihet men åtminstone ett nytt steg i riktning mot målet.

Ett av de större problemen med en dialogstruktur, just för spel, är att hitta rätt balans mellan vad som känns naturligt och trovärdigt samtidigt som spelaren har kvar kontroll över sin karaktär, utan att designern som skriver historien blir lidandes. En ultimat lösning skulle troligtvis kunna vara något i stil med en kunskapsbank där NPC:er sparar olika fakta och uppfattningar de memorerar under sin livstid, från vilken de sedan kan hämta information och bygga sina egna meddelanden med hjälp av en typ av generator som kan ha automatiska diskussioner med spelaren. Med största sannolikhet kommer det nog att dröja lång tid innan vi får se det här i spel – eller i någon fungerande form över huvud taget (DeSmedt & Loritz, 1999). Ett par av de problem som i dag kvarstår är alltså att ge mänskliga NPC:er ett mänskligt beteende samtidigt som de faktiskt kan säga det som en spelutvecklare vill att de ska säga.

1.1 Kapitel 2: Bakgrund

Detta kapitel tar upp relationer och skillnader mellan rollspel och datarollspel. Det berättar lite grann om hur det hela började och även om hur det ser ut i dag. Kapitlet tar även upp exempel i form av andra spel så väl som relaterade forskningsarbeten som gjorts just på området dialog.

1.2 Kapitel 3: Problem

Problemkapitlet beskriver vilka problem det är som detta arbete vill lösa. Problemet delas upp i de två delarna utveckling, en arkitektur som tillåter det resultatet vill åstadkomma samt den implementation som krävs i samband med det, samt utvärdering, vad som bestämmer ifall projektet var lyckat eller inte. Dessa två delar beskrivs lite kortfattat och arbetets kriterier listas upp.

1.3 Kapitel 4: Metod

Det fjärde kapitlet i rapporten går in mer i detalj på vilka typer av metoder som kan tänkas användas för att lösa de två delmålen som listades upp i kapitel 3. Det vill säga hur det skulle kunna gå till för att nå det slutgiltiga målet. Utöver detta beskrivs även projektets olika kriterier – flexibilitet, utbyggbarhet och trovärdighet – mer i detalj.

1.4 Kapitel 5: Resultat

Den största biten av arbetet ligger i resultatkapitlet. Återigen beskrivs delmålen var för sig och detaljer angående vad som faktiskt har gjorts presenteras. Under implementationen ses även prov på hur, exempelvis, parsing av dialogfiler har gått till. Till sist tar det upp en exempelimplementation som använder sig av den här arkitekturen. I detta exempel får en spelare möjlighet att föra dialoger med två olika mänskliga invånare i en virtuell värld. NPC:erna i fråga består av samtidiga och parallella tillståndsmaskiner i form av templates och olika känslomässiga attribut för att bestämma hur de svarar och påverkas av vad spelaren säger.

2 Bakgrund

Det här är ett kapitel det skulle gå att skriva en bok av. Rollspel i olika former har funnits sedan lång tid innan någon först försökte datorisera dem för cirka 30 år sedan. Vissa av dessa försök av varit okej medan andra har varit rent ut sagt dåliga. Utvecklingen är dock tydlig och det märks faktiskt att industrin strävar efter att faktiskt kunna nå en punkt där digitala och icke-digitala rollspel ska kunna vara desamma – även om det kanske inte alltid är en medveten strävan.

2.1 Rollspel – från papper till data

Rollspel har sedan många år tillbaka varit en relativt stor och populär industri. De har funnits i varianter alltifrån två personer som helt enkelt talar med varandra till gigantiska samlingar av böcker och regelsystem på tusentals sidor. En del av dem är som sagt inte baserat på annat än vanlig social interaktion i form av dialog mellan två eller flera personer. Andra fokuserar mer på fiktionsstrider och äventyr. Oavsett vilken typ av rollspel det är så har de alla en sak gemensamt; du spelar en karaktär och det är du som bestämmer vad din karaktär gör och hur han eller hon agerar. Detta är ganska uppenbart genom att bara se på ordet rollspel – du spelar helt enkelt ut en roll. Rollen är formad efter vissa mallar och du agerar enligt dessa och utvecklar din karaktär både i färdigheter så väl som personlighet och åsikter.

Precis som allt annat som blir populärt inom en genre har rollspelen spridits vidare till intilliggande plattformar. Mest populärt och uppmärksammat är nog dataspelen. Det spel många anser ha varit det allra första försöket till ett rollspel i digitalt format är *Adventure* (Crowther & Woods, 1976). Det är ett enkelt spel där spelaren kastas in en så kallad dungeon där han får slåss mot monster och leta skatter. Tydliga paralleller kan här dras till de populära hack-and-slash rollspelen. Adventure har sedan dess följts upp av en hel våg med dataspel som getts titeln RPG. Många klassificerar ett spel som RPG så fort det har något som helst system baserat på erfarenhetspoäng – i de flesta fall följer de även en mer eller mindre linjär berättelse men detta är inte alltid nödvändigt.

Vad är då ett RPG (rollspel)? Jo, det är en social situation där två eller flera personer agerar ut karaktärer i en fiktionsvärld. Vanligtvis sker detta i person, det vill säga inte över Internet eller andra digitala medier, men så behöver inte vara fallet. I många fall används tärningar för att representera handlingar som kan lyckas eller misslyckas och olika system används för att reflektera tärningar och odds mot attribut och färdigheter. Dessa skrivs ofta ner på så kallade rollformulär. En annan vanlig detalj i ett rollspel är spelledaren. Hans eller hennes uppgift är att styra världen runt omkring spelarna – det är han som bestämmer vad, hur, när och varför andra karaktärer säger eller gör olika saker och det är dessutom han eller hon som mentalt eller med hjälp av verktyg modellerar världen. Rollspel kan ha olika inriktningar men det som är gemensamt för alla typer är just att du spelar ut en roll mot flera andra.

För oss som vuxit upp med rollspel som ett av våra främsta intressen är de här spelen ofta rent av förolämpande. Allt för många spel ignorerar totalt det absolut viktigaste i ett *roll*spel – rollen som du, spelaren, själv ska ha kontroll över och utveckla. Erfarenhetspoäng utvecklar självklart karaktärens färdigheter, men inte ens här har du som spelare någon större kontroll att driva karaktären i någon speciell riktning. Att kunna påverka karaktärens attityd, personlighet och åsikter är i princip omöjligt. Karaktären eller karaktärerna du får till ditt förfogande gör enbart vad en designer någon gång skrev ner att de skulle göra. Som spelare lämnas du med ytterst lite frihet

och i princip går spelen bara ut på att du leder någon annan persons roll efter en tydligt förutbestämd linje. Det har gått så pass långt att rollspelare jublar av glädje när de spelar ett spel som erbjuder ett eller flera alternativa slut – vilket fortfarande knappast medför något större mått av rollspelande.

Det allra viktigaste i ett rollspel är just att det är du själv som ska ha kontrollen över din karaktär och att denna ska utvecklas under spelandets gång. Ett viktigt moment i karaktärsutveckling – utöver färdigheter och fysiska attribut – är dess personlighet. Ett av de absolut tydligaste och även ibland roligaste sätten för rollspelare att ge utryck för och, framför allt, *spela* ut sin personlighet är genom dialog med andra karaktärer i spelvärlden (Avellone, 2005). Det är dock också ett av de mest bristfälliga områdena inom dagens datarollspel.

2.2 Datarollspel

Detta är ett försök till att föra över icke-digitala formen av rollspel till ett dataspel. Det är vanligt att regelsystem angående färdigheter och slump tas direkt från existerande rollspel och förs in i en digital miljö med digitala rollformulär och funktioner som tar hand om tärningsrullandet åt en. Somliga tycker att denna typ av systemöverföring är bra medan andra ogillar den (Colantonio, 2003). Spelledaren försvinner näst intill alltid ur leken och dess plats tas upp av speldesigners som skapar en förutbestämd värld som spelaren får ta sig igenom. Det är just detta som är den största skillnaden mellan ett egentligt rollspel och ett datarollspel – det finns ingen aktivt tänkande person som kan göra förändringar i världen enligt spelarnas handlingar och beslut.

Somliga spel har försökt att föra in spelledarrollen även i den digitala världen. Ett exempel på detta kan vi se i *Vampire: The Masquerade – Redemption* (Nihilistic Software, 2000) där multiplayerversionen spelades över LAN och en av spelarna tog rollen som spelledare. Det var sedan dennas uppgift att skapa världen för spelarna och i realtid reagera på deras handlingar. Ett positivt inlägg som tyvärr förhindrades av att det inte gick att spela över Internet.

När det gäller karaktärsutveckling är det ett av de mest skrämmande inslagen för oss som har spelat rollspel större delen av våra liv. Så fort förkortningen XP (experience points) används i ett spel eller om det använder sig av separata stridsscener, i typ med *Final Fantasy*-serien (SquareSoft/Square Enix, 1987-2007), börjar folk kalla det för ett rollspel. Visst är det en form av karaktärsutveckling men det är absolut inte något som nödvändigtvis medför att du faktiskt styr och formar din egen roll, det är snarare så att du leder fram någon annans roll längs en i förväg inspelad väg.

2.3 Linjär dialog i datarollspel

Inte långt efter att *Adventure* släppts började fler utvecklare intressera sig i att göra textbaserade rollspel. Eftersom spelen i stor del utspelades med hjälp av textkommandon gavs spelaren intrycket av en förvånansvärt fri upplevelse – som dock ofta dog bort allt eftersom det upptäcktes hur få kommandon som faktiskt gick att använda och hur sällan så var fallet. Vissa spel, exempelvis *Leisure Suit Larry in the Land of the Lounge Lizards* (Sierra, 1987), implementerade till och med möjligheten att själv skriva vad du skulle säga till andra karaktärer i spelet. Denna typ av dialogtolkning medförde att du själv fick skriva exakt vad du ville att din karaktär skulle säga. Det var dock bristfälligt i sitt utförande och karaktärerna hade märkvärt begränsade resurser för svar.

Allt eftersom åren gick började denna typ av dialogfrihet försvinna ur spelen. Spelen blev större, de tog längre tid att utveckla och de byggde allt för mycket på precis stämning och handling för att ge utrymme för sådant som kan förstöra fantasin och inlevelsen med sin bristfällighet. Nu tog i stället möjligheten att välja mellan ett par alternativa diskussionsområden och svar över.

De flesta utvecklare har dock inte gett mycket utrymme för faktiskt rollspelande i dessa alternativ. Vanligtvis är de bara alternativ som bestämmer vilken information du vill fråga om och du har egentligen ingen som helst input på hur dialogen faktiskt går till. Ett relativt nytt spel där just detta fenomen ofta syns är *World of Warcraft* (Blizzard, 2004). Detta är ändå ett MMORPG med dedikerade servrar just för in-character rollspelande. Trots detta har ändå alla NPC:er (non-player character) i världen helt fasta meddelanden där den enda personifiering de medför är att klistra in din karaktärs namn lite var stans – även om karaktären omöjligt kan veta vad du heter. Den enda frihet du har att påverka detta är, som sagt, bara valet att tala om ett ämne eller inte tala om ett ämne. Dessa ämnen är i sin tur näst intill exklusivt bara för att fråga efter vägvisningar eller uppdrag.

Ett annat vanligt fenomen vi ser i spel med alternativa svar för dialoger är sådana där vi får en falsk frihet i form av att kunna ta en ståndpunkt i en konversation. Vi får här möjligheten att hålla med NPC:n eller att säga emot. Det låter bra till en början men det upptäcker snabbt att det egentligen inte spelar någon som helst roll vilka alternativ du väljer. Antingen har de bara ingen övrig effekt på spelet än vad som kommer stå i nästföljande meddelanden eller så får du bara tala med karaktären om och om igen tills att du väljer det andra alternativet. Här finns många exempel att finna, näst intill i alla så kallade rollspel till dator eller konsol. Ett relativt nytt sådant ser vi i *The Legend of Zelda: Twilight Princess* (Nintendo, 2006) där spelaren i början av spelet stöter på en irriterande liten pojke som vägrar släppa förbi dig förrän han fått låna ditt träsvärd. Du får alternativen att låna honom svärdet eller att inte göra det. Följer du den mall som ofta är förväntad hos vuxna och säger nej blir du helt enkelt fast fram till dess att du ändrar dig. Även om det i sig är en dödssynd att ta upp *Twilight Princess* i ett dåligt sammanhang är detta ett tydligt exempel på när valfriheten bara ger oss en falsk förhoppning om att vi kan påverka och forma vår karaktär och avatar i spelvärlden.

Ett annat tydligt exempel på denna form av dialogalternativ ser vi i *RuneScape* (Jagex Ltd., 2001). Här tas spelaren steget längre än i *Twilight Princess* och i vissa fall får han eller hon möjligheten att rent av förolämpa den tilltalade. I ett fåtal fall när det gäller obetydliga NPC:s som annars bara går att döda kan sådant leda till att spelaren blir attackerad (se Figur 1). Efter att striden i sig är slut har det ingen som helst konsekvens på resten av världen. Du kan döda 10 000 av kungens vakter och 20 000 civila människor men du anses fortfarande vara en hjälte för att du lyckades hitta ett ägg och lite mjölk precis i tid för Kungens födelsedag. I de fall där man inte blir attackerad utan i stället får ett argt svar kan man helt enkelt bara starta en ny konversation med karaktären och välja ett nytt alternativ. Detta fortlöper fram till dess att du svarat så som du, enligt utvecklarna, borde svara och det har sedan inga negativa konsekvenser.

Figur 1. Då du talar med en Ogre Enclave Guard får du upp två svarsalternativ. Väljer du att förolämpa ogren kommer han att bli arg och attackera dig.

Dessa typer av dialoger fungerar i sig och många spelare gillar uppenbarligen ändå spelen. Det är dock ingenting som borde klassificeras som renodlade rollspel och det tillfredsställer inte oss som faktiskt vill spela vår karaktär så som vi tycker att den borde spelas (Colantonio & Avellone, 2003). Det finns dock utvecklare som håller med oss.

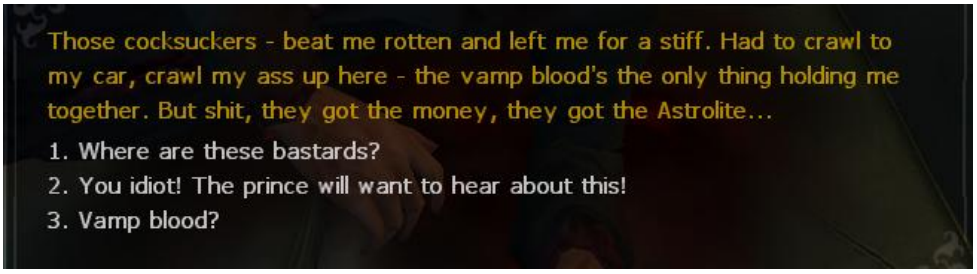
2.4 Frihet i dialog i datarollspel

Att kalla det frihet kan i vissa fall vara en överdrift men det är definitivt steg i rätt riktning. I kategorin av spel där utvecklarna tydligt försöker ge oss mer i form av dialog och konsekvenser, som i sin tur kan leda till din karaktärs personlighetsutveckling, finner vi bland andra *Vampire: The Masquerade – Bloodlines* (Troika Games, 2004), *Fallout 2* (Black Isle Studios, 1998), *Star Wars: Knights of the Old Republic* (BioWare, 2003) och *Neverwinter Nights 2* (Obsidian Entertainment, 2006). Dessa spel har försökt föra vidare dialogen till det som hela genren en gång grundades på för nu 30 år sedan. De använder sig fortfarande av fasta svarsalternativ men de arbetar mer med att faktiskt ge konsekvenser på vad det är man säger och låter till en viss del karaktärer i spelvärlden reagera känslomässigt. De når dock inte ända fram.

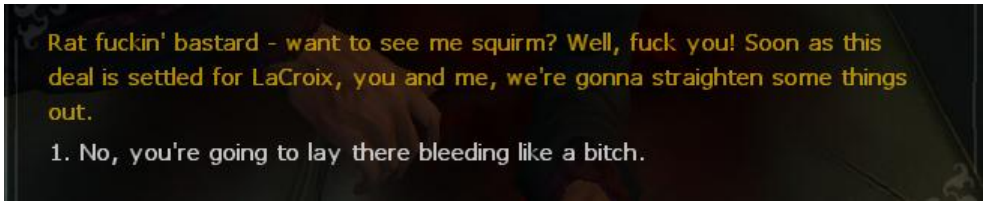
I *Vampire: The Masquerade – Bloodlines* får man som spelare välja en av sju vampyriska klaner man ska spela som, samt om man ska vara man eller kvinna. Dessutom har man sina attribut och färdigheter som i andra datarollspel – en viktig skillnad här är dock att dessa förs vidare in i dialogerna. En karaktär med hög färdighet inom Intimidation kan exempelvis få upp alternativ att skrämma den han talar med, medan en karaktär med hög färdighet inom Seduction i stället kan få möjligheten att förföra. Dessa färdigheter i blandning med de olika klanerna innebär att man får en mycket mer personifierade alternativ i dialogerna – alternativ som bättre passar den karaktär man faktiskt vill spela som.

Under spelets gång kan man sedan använda sina färdigheter i dialogerna eller helt enkelt bara välja olika alternativ för att ta sig runt problem man annars hade behövt slå sig igenom. Näst intill hela den första halvan av spelet går faktiskt att enbart tala

sig igenom om man säger rätt saker och utnyttjar sina färdigheter korrekt. I andra fall kan man få motsatt effekt och man kan rent av missa hela uppdrag eller andra bonusar på grund av att man förolämpat den NPC som annars hade gett ut det. Ett exempel på detta kan ses tidigt i spelet då man som spelare för första gången stöter på en av Prinsens andra hantlangare. Oavsett vad man säger i denna dialog kommer man att få nästa uppdrag. Exakt vad det är man säger kan dock förändra ens relation med honom (se Figur 2). Väljer man att inte bara förolämpa honom utan att även hota med att skvallra till Prinsen angående hans misslyckanden kommer det att leda till ett hat och ni kommer senare i spelet att slåss (se Figur 3). Ett par karaktärer kan till och med rent av sluta tala med en om man beter sig allt för illa.



Figur 2. Ett par av dina alternativ då du för första gången möter en ghoul till Prinsen.



Figur 3. Förolämpar du ghoulen och dessutom väljer att skvallra till Prinsen kommer det leda till att ni senare i spelet kommer att slåss.

På så vis går spelet absolut i rätt riktning men som sagt är det fortfarande en lång bit kvar. Det ultimata målet är självklart en totalt fri dialog men för att nå dit kommer vi troligtvis passera andra stadier först. En del som fattas i just *Vampire: The Masquerade – Bloodlines* är att du aldrig kan bete dig annorlunda än vad svarsalternativen tillåter. I vissa enstaka fall när konversationen är kort kan du ibland få välja mellan olika former av alternativ som alla tar upp samma ämne men i längre diskussioner får du helt enkelt nöja dig med hur alternativet i sig uttrycker sig. Du får alltså ingen möjlighet att ställa samma fråga eller säga samma kommentar på olika sätt för att få olika känslomässig respons från den du talar med.

2.5 Relaterade arbeten

Förutom de olika exemplen på dialog vi kan se i, exempelvis, de ovan nämnda spelen pågår även en hel del forskning kring området och har åtminstone gjort så sedan mitten av 80-talet (DeSmedt & Loritz, 1999). I de flesta av dessa forskningsområden ser vi att man försöker åstadkomma vad som misslyckades i vågen av PC-spel som släpptes under 80-talet och tidigt 90-tal. Nämligen att man vill låta spelaren tala helt fritt med en NPC. Det må vara ett bra mål men det är ingenting vi lär se under de närmsta 20 åren (DeSmedt & Loritz, 1999), om ens då.

För att något sådant ska lyckas krävs det att NPC:er kan representera personligheter och förändringar i dessa. Detta kräver i sin tur att de ska kunna visa och förändra sina känslor och reagera på minnen och upplevelser. Hur detta ska kunna representeras i en kodstruktur har vi också fått se exempel på (Morris, 2002). Fram till att vi faktiskt kan se resultaten av detta i form av spel behöver vi ändå föra utvecklingen framåt och

fylla ut det tomrum som skulle uppstå i fall vi bara satt och väntade på den ultimata lösningen.

2.5.1 ELIZA

Ett av de mest välkända programmen när det gäller autogenererad dialog är troligtvis ELIZA. Detta utvecklades 1966 för att simulera dialog med en psykologliknande personlighet (Weizenbaum, 1966), under en tid då forskning på detta område blommade upp kraftigt. ELIZA fungerar så att mänsklig textinput tolkas av ett program och att olika nyckelord letas upp. Dessa nyckelord används sedan för att generera ett svar utefter olika mallar. Programmet har dock flera begränsningar, bland annat kan det bara tolka en mening åt gången och inga kommatecken får användas. Meningarna får inte heller vara direkta frågor, utan snarare påståenden om ens egen situation (se Figur 4).

Eliza: Hello. I am ELIZA. How can I help you?
Input: I am drunk.
Eliza: How long have you been drunk?
Input: Approximately since I drank that bottle of vodka.
Eliza: Can you elaborate on that?
Input: I drank vodka because I wanted to get that funny feeling I get when I'm drunk.
Eliza: Can you elaborate on that?
Input: You're an idiot.
Eliza: How long have you been an idiot?

Figur 4. Exempeldialog från en diskussion med ELIZA. Uppenbarligen finns det en hel del brister.

2.5.2 SmarterChild

Mycket längre än ELIZA har vi inte kommit i dag. Forskningen går framåt men tillgängliga produkter är sällan märkvärt bättre. Ett exempel som är värt att nämna är SmarterChild (Colloquis, 2001). Det är en så kallad chatterbot utvecklad för AOL Instant Messenger (AIM), vilket tillåter vem som helst som har tillgång till AIM att tala med sin virtuella kompis. SmarterChild bygger på ett mer komplext system av igenkänning av mönster och frågeställningar och kan på så sätt tolka flera olika typer av meddelanden än vad ELIZA kan (se Figur 5). Det känns dock ändå aldrig som att det faktiskt är en riktig person man talar med, vilket i en spelmiljö vore helt oacceptabelt.

Input:	How are you doing today?
SmarterChild:	Great thx! You?
Input:	Not too bad. Although I have a strange urge to eat something.
SmarterChild:	Ha! Wow.
Input:	Yeah, kind of impressive, huh?
SmarterChild:	Very interesting question, g00w.
Input:	Actually, it was a statement only said as a question to see if you agreed with it.
SmarterChild:	It was what?!
Input:	Apparently you don't understand.
SmarterChild:	Apparently I don't understand, huh?
Input:	See, just like that! That's what I did!
SmarterChild:	Like that that s what you did? No way?! Really?
Input:	Yes, really.
SmarterChild:	Oh.

Figur 5. Ett utdrag från en konversation med SmarterChild. Uppenbarligen verkar den mer intelligent än ELIZA men långt ifrån vad som känns mänskligt.

2.5.3 Prolog

Prolog är inget dialogarbete i sig, utan det är egentligen ett programmeringsspråk designat specifikt för hantering av diverse typer av AI (Colmerauer & Roussel, 1992). Det är skapat för att kunna dra slutsatser mellan olika logiska förhållanden och relationer. Exempelvis om B är närmre A än C och C är närmre A än D så måste B vara närmre än D. Med hjälp av funktioner för att känna igen logiska mönster har det utvecklats ett tillägg till Prolog kallat DCG (definite clause grammar). DCG används för att bygga upp grammatiskt korrekta meningar genom att organisera ord efter ordklasser i olika trädstrukturer och genom att definiera en uppsättning regler efter vilka dessa kan sättas ihop (Bratko, 1990). Som sagt är det ingen egentlig arkitektur för dialogsystem men med viss omkringliggande logik skulle det troligtvis kunna bli användbart för vidare forskning just på det området.

3 Problem

Syften med arbetet är att utforma och utveckla en arkitektur för dynamiska dialoger i dataspel – då framför allt hos *mänskliga* NPC:er. Detta system bör sedan kunna användas för att interaktionen mellan spelare och NPC ska förbättras på så sätt att spelaren får mer frihet inte bara i *vad* han säger utan även *hur* han gör det. Dialogalternativen kommer att finnas kvar men de ska gå att modifiera för att bättre passa spelarens sätt att agera ut sin roll. Dessutom ska NPC:er ge en trovärdig respons på spelarens handlingar och på så sätt kännas mer levande – de ska kunna reagera enligt känslor och ha olika svar på samma tilltal.

En fråga som direkt dyker upp är om inte låsta alternativ ändå är för begränsande för att helt kunna agera ut sin roll i ett spel. Ett kort och enkelt svar är ja, de är för begränsande. Möjligheten att själv kunna modifiera *hur* det sägs ger dock en större kontroll över personligheten bakom just din karaktär. Hur pass begränsad man sedan blir i antalet alternativ beror till stor del på designern som använder systemet – möjligheten att lägga in en ovanligt stor mängd alternativ för att fylla de olika önskemål en spelare kan tänkas ha finns ju där. Självklart är det omöjligt att göra listan oändligt lång men vi måste också komma ihåg att total frihet ger upphov till en hel del out-of-character problem, nämligen det att karaktären kommer att få tillgång till samma kunskap som spelaren har. Detta ska självklart inte alltid vara fallet, speciellt inte om spelaren har spelat hela spelet tidigare och genom fri text direkt kommer att kunna hoppa över viktiga spelmoment som bryter mot in-character-regler angående rollspel. Ett annat starkt argument, som vi såg bevis för i bakgrundskapitlet, är att tekniken inte på långa vägar stödjer tillräckligt bra AI för att trovärdigt och enligt unika personligheter kunna svara på fri text.

3.1 Delmål 1: Utveckling

För att kunna realisera de ovan nämnda egenskaperna behövs en arkitektur som stödjer de olika aspekterna för denna typ av interaktion. En spelares karaktär måste kunna befinna sig i olika mentala tillstånd samt ha frihet att påverka sina meddelandealternativ för att bättre passa den attityd spelaren vill att han ska ha. Då behövs alltså en metod som stödjer sådana tillstånd och som även tillåter multipla varianter för alla alternativ som kan väljas i dialoger.

NPC:er ska, som sagt, kunna reagera på spelarens handlingar på ett trovärdigt sätt. Detta projekt fokuserar på NPC:er som ska verka mänskliga, vilket leder till att känslor kommer att spela en stor roll. De ska alltså kunna ge respons i dialogen som baseras på vad de tycker om spelaren och hur de känslomässigt sett reagerar på vad som sägs och framför allt *hur* det sägs. Dessutom behöver de kunna komma ihåg vad spelarens karaktär har sagt – sällan ser vi NPC:er i spel som reagerar när spelarens karaktär upprepar sig, vilket uppenbarligen är en detalj som skiljer dem från oss levande människor.

Förutom känslor drivs människor dessutom av en del instinkter och en kraftig sådan som detta system kommer att försöka ge möjligheter för är rädsla. Olika människor hanterar rädsla på olika sätt – somliga grips av panik medan andra agerar vaksamt och försiktigt. Grunden för deras beteende ligger dock i just det faktum att de är rädda. Är man som människa verkligen rädd för någon eller något har instinkter en tendens att ta över kontrollen, därför kommer den här arkitekturen att använda sig av ett sätt att hantera rädsla som ligger utanför det övriga spektrumet av känslor som varje mänsklig NPC kommer att ha tillgång till.

För att kunna implementera detta kommer det först att behöva skapas ett system som enkelt kan hantera arkitekturen. Alltså ett system som kan foga samman känslor, minnen och rädsla. Dessa NPC-relaterade attribut måste även ha en anknytning till spelaren och dialogen. Dialogen i sin tur behöver ett system där det går att skriva exempel som visar prov på hur det här systemet faktiskt kan komma att fungera i ett spel. För att då kunna fokusera just på de tidigare nämnda delarna och slippa underliggande tekniska bitar, så som hantering av input och grafik, behövs även ett bibliotek som enkelt kan hantera detta.

3.2 Delmål 2: Utvärdering

För att visa på ett exempelscenario av vad som är tänkt att kunna åstadkommas med detta arbete kommer en liten applikation att sättas ihop där man som spelare kan välja att vara en man eller kvinna i en begränsad värld. Med hjälp av detta skulle det gå att visa prov på vilka olika sätt man kan interagera med NPC:er genom dialog, hur spelarens karaktärs eget mentala tillstånd påverkar detta samt hur NPC:er kan reagera på det. För att detta ska bli möjligt kommer denna lilla värld att innehålla två stycken NPC:er som spelaren ska kunna tala med. Argumentet för att använda två stycken i stället för bara en är att kunna visa prov på hur olika NPC:er kan agera olika jämte mot spelarens beteende. Olika människor kan ha olika standarder för vad som anses förolämpande och acceptabelt, samt vad som inbringar respekt och vad som inbringar förakt. Dessutom, som nämnts tidigare, kan olika personer agera helt olika då de sätts under press i form av rädsla. Två karaktärer att interagera med ger därför möjlighet till en bra kontrast till hur detta system kan användas inom spel.

Vad är det då som ska utvärderas? Jo, eftersom detta arbete riktar sig lika mycket till speldesigners som programmerare kommer ett brett fält av kriterier att behandlas:

- Flexibilitet – Är det ett flexibelt system som faktiskt tillåter ett individuellt agerande från NPC till NPC?
- Utbyggbarhet – Är det lätt att bygga ut systemet för ett större spektrum av känslor – eller kanske rentav begränsa det?
- Trovärdighet – Klarar arkitekturen av att representera NPC:er på ett sätt som faktiskt känns trovärdigt?

4 Metod

För att lösa problemet som togs upp i kapitel 3 krävs metoder, både för utveckling och också utvärdering. Det finns ett flertal av dessa att välja mellan och de medför alla olika fördelar och nackdelar, samtidigt som de kan vara olika svåra att använda. Målet med metoden för just det här projektet är att finna någon som är pålitlig att använda samtidigt som den inte är allt för innefattande så att den får plats inom tids- och resursramarna.

4.1 Metod för delmål 1: Utveckling

I och med kapitel 3.1 framgår vad som ska skapas, nästa steg blir då att ta reda på vilken metod som bör användas för detta. Det finns ett flertal olika metoder att välja mellan och även om de stor del är bra kan de vara svåra att genomföra i praktiken.

4.1.1 Intervjuer

Ett sätt att ta reda på hur en arkitektur för den här typen av system skulle kunna utformas vore genom intervjuer med experter på området – personer som tidigare arbetet med liknande projekt. Sådana typer av personer skulle kunna innefatta både designers och programmerare. Designers vet vad de vill kunna använda i sina produkter och vet vilka brister de stött på i tidigare projekt. Programmerare vet vilka tekniska begränsningar de haft och kan även ha en aning om varför dessa uppkom. Att intervjua båda typer av personer och dessutom från flera olika projekt borde det gå att samla en bra grund information som sedan går att bygga ut för att förhoppningsvis få systemet ännu bättre.

Ett direkt problem som dyker upp i den här typen av metod är först och främst att man måste ha kontakter för att kunna genomföra den. Dessutom måste dessa kontakter vara villiga att ställa upp i sådana studier. Ett andra problem är ett som alltid återkommer vad du än sysslar med, nämligen tid. Att hitta kontakter, intervjua dem och sammanställa informationen är en tidskrävande process och det finns ingen garanti för att just de personer du valt att intervjua kommer att tillföra sådan information som kan komma till användning.

4.1.2 Litteraturanalys

En annan metod som kan användas är att granska olika former av litteratur. Dessa kan vara böcker och artiklar som skrivits just om dialoger i dataspel men de kan också handla om mer generella implementationsarkitekturer för olika system. Genom att granska olika spel framgår ganska tydligt vad de är kapabla till. Med hjälp av detta går det sedan att använda sig av litteratur av olika slag för att finna en form av arkitektur som skulle kunna stödja vad som redan påträffats i tidigare spel som använder sig av dialogsystem samtidigt som det förhoppningsvis kan erbjuda möjligheter att, åtminstone bitvis, implementera sådant som tidigare endast påträffats i icke-digitala rollspel.

Problemet med litteraturanalys är att det kan vara svårt att hitta en konkret text som just handlar om det här specifika ämnet. Troligtvis finns det ett par stycken där ute någonstans men det gäller att hitta dem och det är inte alltid det lättaste. Baseras projektet i stället på litteratur som beskriver mer generella tekniker vilka sedan jämförs mot redan existerande produkter finns det dock en större risk att viktiga detaljer kan missas – vilket troligtvis vore ett mindre problem om man använde sig av direkt intervjuer.

4.1.3 Fallstudier

Denna metod skulle kunna användas genom en direkt studie av hur tidigare projekt implementerats och att detta sedan jämförs med vilka brister den slutliga produkten hade. Först och främst finns en tidigare arkitektur att studera och för det andra kan man i efterhand se vilka brister denna hade och försöka göra förändringar för att lösa dessa samt bygga ut den tidigare arkitekturen.

Det första problemet med denna metod är att hitta en professionellt utformad arkitektur som är helt öppen för användare att titta på. Många företag som tidigare utvecklat system som anses eller ansetts vara nytänkande är troligtvis benägna om att hålla dessa metoder för sig själv. Ett andra problem är att om man använder sig av en tidigare implementerad arkitektur med ett par identifierade brister finns chansen att det nya systemet följer samma riktlinjer och på så sätt hamnar i liknande situationer där brister påträffas. Självklart skulle det gå att förändra arkitekturen iterativt men då stöter vi återigen problemet med tidsbrist, vilket sällan verkar gå att undvika.

4.1.4 Val av metod

På grund av en relativt stor informationsbank gällande icke-digitala rollspel samt de brister som ofta framgår när dessa förs över till digital form kommer denna arkitektur utvecklas till stor del med hjälp av metoden litteraturanalys. Det viktigaste här är att finna lösningar på tidigare problem samt att finna ett system som kan tillåta att dessa byggs ut ytterligare vid senare tillfällen. Användning av en metod som helt och hållet grundar sig på problem som redan fått lösningar bör därför inte vara den ultimata. Även om den nya arkitekturen finner en del brister i områden som i andra fall fungerat finns det redan information där ute som talar om hur dessa kan lösas. Det gäller snarare att hitta ett system som kan lösa de nya problemen. Skulle denna sedan inte vara ultimat för alla typer av problem som tidigare påträffats och lösts går det alltid att arbeta vidare på ett projekt där man försöker sammanfoga de två.

Typen av litteratur kommer att vara en blandning av designerspecifik information samt generella programmeringslösningar för diverse typer av problem. Detta förhindrar att systemet faller in i samma spår som tidigare arkitekturer men att det ändå bibehåller en viss kontakt med vad folk faktiskt försökt att åstadkomma vid tidigare tillfällen när det gäller rollspelsattribut i dataspel – i detta fall just för mänskliga NPC:er.

4.1.5 Implementation

För att kunna validera och utvärdera arkitekturen kommer den att behöva implementeras. Implementeringen kommer självfallet att använda sig just av metoden implementering, troligtvis i Visual Studio (Microsoft, 2003).

4.2 Metod för delmål 2: Utvärdering

Vi vet redan vad vi ska göra för att få en bas att utvärdera systemet på – skapa en liten virtuell värld med två NPC:er av olika personligheter som spelaren kan föra dialoger med. Världen bör vara grafisk för att även ge visuell feedback på hur dialogen faktiskt påverkar NPC:erna i den, vilket lätt bör kunna lösas med hjälp av olika ansiktsuttryck. Tillsammans med dialogen i sig bör detta ge oss möjligheten att se att vi faktiskt får väntade resultat när vi för konversation med NPC:er. Som nämnt i kapitel 3.2 har vi även ett par kriterier som måste mötas:

- Flexibilitet. Det här har till viss del redan nämnts men det är värt att ta upp det igen. Eftersom det här projektet har som mål att öka den faktiska rollspelskänslan i datarollspel är det viktigt att man som spelare faktiskt känner att man befinner sig i en liten värld som befolkas av faktiska individer och inte bara informationskällor för ett linjärt äventyr. För att detta ska bli möjligt är det viktigt att de olika individerna faktiskt har sina egna personligheter i dialogen när man talar till dem och att alla inte följer samma spår bara att dialekten de talar med är annorlunda.
- Utbyggbarhet. Olika utvecklare kan begära olika djup på sina produktioner, därför är det viktigt att varje designer var för sig kan bestämma sig för olika alternativ när det gäller antalet känslolägen en NPC kan befinna sig i. Det krävs alltså att varje implementation av detta system ska kunna använda sig av olika former av känslor. Detta blir även viktigt då vi använder oss av NPC:er som inte är mänskliga, även om det ligger utanför fokus för detta arbete, och därför inte har samma skala av känslor som en människa kan ha.
- Trovärdighet. Det här är ett kriterium som ligger ganska när flexibiliteten. Visst är det bra om NPC:er i spelet faktiskt kan spegla olika personligheter men gör de det på ett trovärdigt sätt? Om en implementation av arkitekturen kan ge trovärdighet betyder det troligtvis att arkitekturen i sig är bra, åtminstone just för detta kriterium.

4.2.1 Intervjuer

Precis som för utvecklingen av arkitekturen så kan utvärderingen göras med hjälp av intervjuer med diverse experter på området – det vill säga professionella speldesigners som använder sig av dialogsystem i sitt dagliga arbete. Detta borde tala om vad som är bra med arkitekturen, det vill säga vilka nyheter den tillför samt hur dessa kan användas inom spel, samtidigt som det också kan finna brister, sådant som begränsas av arkitekturen och vilka nyheter som inte riktigt når dit man skulle kunna önska. Att intervjua designers ger dock bara en inblick i det slutliga användandet av systemet och ingen direkt utvärdering av arkitekturen på en mer generell kodnivå. För detta skulle det gå att vända sig till en andra kategori av experter att intervjua – nämligen programmerare. Här skulle det gå att använda informationen man fått ut från designers och diskutera vidare med programmerare gällande hur dessa brister eventuellt kan ligga direkt i strukturering av funktioner och klasser.

Problemet med den här typen av metod är just att den kräver kontakter som är villiga att lägga ned sin tid på testning och utfrågning. Dessutom kräver det tid från projektansvarig att faktiskt se till att skaffa dessa kontakter och schemalägga med de personer som ska komma att intervjuas. Slutligen krävs ytterligare mer tid för sammanställning och utvärdering av de faktiska åsikterna man fått fram genom intervjuerna.

4.2.2 Experiment

En metod som inte behöver kräva inblandning från utomstående personal är att själv utföra diverse experiment. I detta fall kan experiment innebära att en exempelimplementation utvecklas utifrån arkitekturen där de olika kriterierna testas i direkta spelsammanhang. En positiv aspekt med detta, förutom att den kan utföras av en ensam person, är att alla kriterierna slås samman i ett enda projekt. Förutom att de utvärderas var för sig får man även en inblick i hur de samspelar med varandra. Självfallet går det även att använda detta exempel i kombination med intervjuer, då

experter får kommentera utifrån vad som åstadkommits med hjälp av systemet och reflektera genom detta i stället för att direkt gå in på vad som skulle kunna göras.

Nackdelar med att använda den här metoden, utan att blanda den med intervjuer, är först och främst bristen på variation i åsikterna. Det kommer endast att innefatta en persons åsikt, en person som i sin tur leder till nästa problem – att vara opartisk. Experimentet kommer troligtvis att utföras av samma person som utvecklade systemet till att börja med. Denna utvecklare kommer då att, medvetet eller undermedvetet, känna till de brister systemet kan tänkas ha och arbeta sig runt dessa på olika sätt som andra användare skulle kunna se som komplicerade och, framför allt, hämmande för det slutgiltiga resultatet.

4.2.3 Val av metod

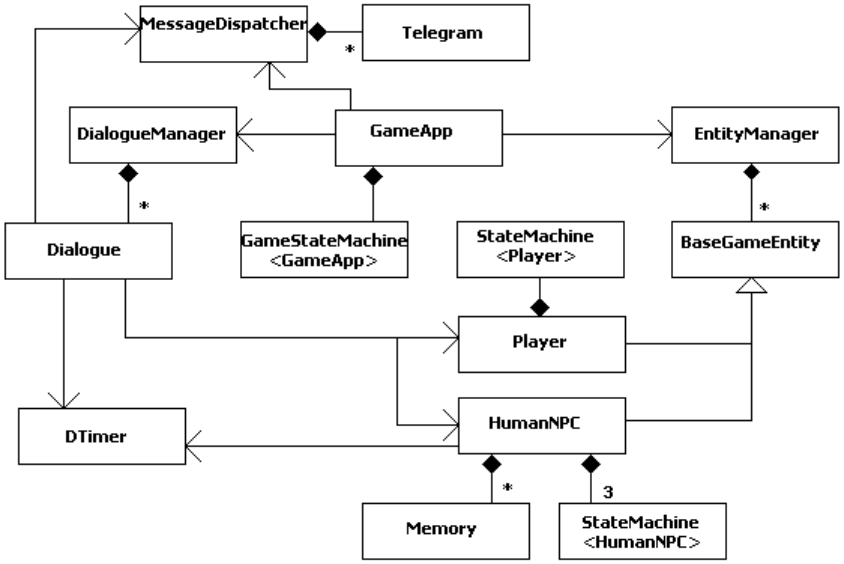
Just på grund av brist på kontakter och tid kommer detta arbete att använda sig av experimentmetoden. Det ska dock påpekas att den nödvändigtvis inte behöver vara sämre än intervjuer med experter, just på grund av att den faktiskt tillåter en användare att se hur kriterierna samspelar med varandra. Detta ger en inblick i hur det faktiskt kan användas i spel, vilket just är dess huvudsakliga uppgift. Som tidigare påpekat kan det ses som en brist att samma person som utvecklade systemet avgör hur bra det här men det ska också has i åtanke att en användare av systemet måste vara insatt i det. Skulle man använda sig av en utomstående testare skulle det kräva tid för inlärning och mental bearbetning av systemet – något som dess skapare redan besitter.

5 Resultat

Efter att metoderna för projektet valts utvecklades arkitekturen enligt dessa med mål att möta de kriterier som gicks igenom i kapitel 3.2. För att sedan försöka sig på en utvärdering av arkitekturen jämt emot dessa skapades ett exempelscenario i vilka olika kriterier kunde testas och mätas för att ge en inblick i hur de skulle kunna komma att fungera i ett faktiskt spel.

5.1 Resultat för delmål 1: Utvecklig

Den slutgiltiga arkitekturen för det här dialogsystemet är utvecklat i en objektorienterad miljö med hjälp av samtidiga och parallella tillståndsmaskiner i form av templates. Vad denna arkitektur löser är att den först och främst tillåter spelare att själva välja hur deras karaktär ska bete sig i och uttrycka sig i dialoger med NPC:er. Dessutom tillåter den att NPC:er reagerar känslomässigt inte bara på vad spelaren säger utan även *hur* han eller hon säger det. NPC:s reaktion kan ändra dess permanenta så väl som temporära känslor jämte mot spelaren likväl som den kan påverka hans permanenta och temporära rädsla. Dessutom har NPC:erna försetts med minnen som tillåter dem att under vissa tidsintervall komma ihåg vad spelaren har sagt och därmed kunna reagera olika ju fler gånger spelaren säger samma sak. Det underliggande ramverket (se Figur 6) för denna arkitektur innehåller alla de nödvändiga grunderna för att detta ska kunna fungera.



Figur 6. Klassdiagram för arkitekturen av ramverket för dialogsystemet.

Vid sidan av själva ramverket utvecklades dessutom ett exempelspecifikt system för att, genom externa filer, faktiskt kunna läsa in dialog och alternativ som skulle kunna användas av arkitekturen i ett eventuellt spel. För detta syfte skapades ett enkelt skriptspråk, fokuserat just på arkitekturen av det här systemet, samt omkringliggande nödvändigheter så som parsing av skriptet.

5.1.1 GameApp

Detta är den egentliga spelapplikationen och bör utformas efter just det spel som ska implementeras. Kort kan dock nämnas att den innehåller en tillståndsmaskin bestående av flera GameState. Ett viktigt tillstånd applikationen kan befinna sig i är InDialogue. När spelet befinner sig i detta tillstånd skickas tangenttryckningar som är

relevanta för att välja och skifta mellan dialogalternativ vidare direkt till den pågående dialogen (via DialogueManager) som där får avgöra hur tangenttryckningen ska hanteras.

5.1.2 MessageDispatcher och Telegram

MessageDispatcher, som är en singleton, arbetar tätt tillsammans med tillståndsmaskinsmodellen för arkitekturen. De olika tillstånden arbetar tillsammans genom att skicka meddelanden, i form av Telegram, mellan sina olika ägare. MessageDispatchers uppgift är att se till att dessa Telegram når sin destination vid den utsatta tiden. För att veta vem meddelandet ska skickas till och vem det kommer ifrån markeras de med ett ID för avsändare så väl som mottagare. Dessa ID:n används sedan för att få en pekare till rätt mottagare ifrån EntityManager.

Förutom avsändare och mottagare innehåller även meddelandena två andra viktiga bitar information. Dessa är msgType och msgString. msgType är en integer som i form av ett enum talar om vilken typ av meddelande Telegrammet är av. msgString är ett värde, i form av en string, som kan skickas med meddelandet. Här är en lista på de viktigaste meddelandena i den här strukturen och vad för typ av string-värden som bör medfölja:

- ChangeState – Ett meddelande som talar om för mottagaren att byta tillstånd. Namnet på det tillstånd som ska bytas till medföljer i form av en string.
- GetMoreAngry – Talar om för NPC:s nuvarande EmotionalState (mer om detta senare) att gå över till ett argare tillstånd. Här behövs inte något string-meddelande medfölja.
- GetLessAngry – Talar om för NPC:s nuvarande EmotionalState att gå över till ett mindre argt tillstånd. Precis som med GetMoreAngry behöver man inte skicka med något string-värde.
- DecreaseTempAnger/DecreasePermAnger – Talar om för NPC:s EmotionalStateMachine (mer om detta senare) att minska mängden TempAnger/PermAnger (mer om detta senare) med ett värde som medföljer i string-form.
- IncreaseTempAnger/IncreasePermAnger – Fungerare som DecreaseTempAnger/IncreaseTempAnger men används för att i stället öka TempAnger/PermAnger.
- ModifyTempAnger – Det här meddelandet talar om för NPC:s EmotionalStateMachine att den ska modifiera NPC:s TempAnger för att glida lite närmre PermAnger.
- GetAfraid – Talar om för NPC:s nuvarande FearState att det ska bli Afraid.
- StopBeingAfraid – Talar om för NPC:s nuvarande FearState att bli NotAfraid.
- DecreaseTempFear/DecreasePermFear – Talar om för NPC:s FearStateMachine (mer om detta senare) att minska TempFear/PermFear med ett värde som medföljer i string-form.
- IncreaseTempFear/IncreasePermFear – Talar om för NPC:s FearStateMachine att öka TempFear/PermFear med ett värde som medföljer i string-form.
- ModifyTempFear – Fungerar likadant som ModifyTempAnger fast för TempFear.

- StartTalking – Talar om för NPC:s ActivityStateMachine (mer om detta senare) att den ska gå in i en dialog. Inget string-värde behöver skickas med här.
- StopTalking – Talar om för NPC:s ActivityStateMachine att den ska gå ur en dialog. Liksom med StartTalking behöver man inte skicka med något string-värde.

5.1.3 StateMachine

Arkitekturen använder sig av två olika tillståndsmaskiner. En av dem är dock bara en utbyggd variant för att hantera GameState, så den behöver vi inte gå in i detalj på här. Den andra typen av tillståndsmaskin är den som används av entities. Denna tillståndsmaskin, som är en template som bygger på dess användare (`template class<user_type>`), har funktionalitet för att hantera ett nuvarande, föregående och globalt tillstånd. Detta tillåter att varje tillståndsmaskin förutom sitt nuvarande tillstånd även känner till det föregående tillståndet samt att den har ett globalt tillstånd som uppdateras vid sidan av det nuvarande. Det globala tillståndet kan även ta hand om meddelanden som det nuvarande tillståndet inte kunde hantera. Detta förklaras nog enklast genom att helt enkelt visa koden för det:

```
bool handleMessage(const Telegram& msg) const
{
    if(m_currentState  &&  m_currentState->onMessage(m_owner,
msg))
    {
        return true;
    }
    else       if(m_globalState       &&       m_globalState-
>onMessage(m_owner, msg))
    {
        return true;
    }
    return false;
}
```

5.1.4 State

Varje tillstånd – vilka för övrigt representeras som template singletons, ungefär i samma stuk som StateMachine – som kan finnas i en tillståndsmaskin består av ett par gemensamma funktioner. De viktigaste här är enter(), execute() och exit(). Då en tillståndsmaskin går in i ett nytt nuvarande tillstånd (currentState) körs dess enter()-funktion. Sedan, under update(), körs dess execute()-funktion. Till sist, då tillståndsmaskinen går ur ett tillstånd, körs dess exit()-funktion. Förutom dessa ganska standardiserade funktioner har varje tillstånd dessutom ett namn i form av en string. Detta namn används för att kunna identifiera olika tillstånd genom de string-värden som kan skickas med i ett Telegram.

Anledningen till att tillstånden är templates är den att de ska kunna ta en specifik användare som parameter för sina funktioner. Exempelvis om klassen används av en

HumanNPC ska den kunna ta en pekare till en HumanNPC i sina funktioner för att korrekt kunna hantera just den användaren och dess interface.

5.1.5 EntityManager

EntityManagern, även den en singleton, ser ut som en EntityManager ofta brukar göra. Den har ett interface som tillåter att man kan registrera och ta bort Entities i form av BaseGameEntity. Den sparar alla Entities i en map där deras pekare listas mot en ID i form av en string. Just i detta fall används en funktion som tillåter att man själv inte behöver namnge sina Entities. Väljer man att avstå skapar EntityManager i stället ett nytt unikt namn för Entitien – ett siffervärde i stigande ordning. Självklart går det dock att själv namnge sina Entities, vilket är att rekommendera.

En ganska unik detalj som bör påpekas är att Entities går att hämta på två olika sätt. Antingen kan man hämta dem med sitt ID, vilket är unikt, eller så kan man använda deras karaktärsnamn (så länge det är en Entity som har ett personligt namn, det vill säga). I det senare fallet skulle man exempelvis kunna hämta en Entity vid namn Benny. Det finns dock ingenting som säger att flera individer i en värld kan ha samma namn, så i det här fallet returnerar EntityManager en vector av alla Entities som har ”Benny” som sitt personliga namn. I spelsammanhang tillåter detta att NPC:er kan missförstå spelare som inte är tydliga nog. Exempelvis kanske du, som spelare, talar om för en NPC att tala om för A att han måste lämna tillbaka lånade pengar innan klockan 8:00 nästa morgon. Finns det flera karaktärer i NPC:s kännedom som har namnet A kan därmed hända att han lämnar meddelandet till fel person, vilket, i min mening, skulle ge karaktärerna en mer mänsklig känsla där de kan missförstå information som inte är tydlig nog.

5.1.6 BaseGameEntity

Detta är basklassen för spelets Entities. Den innehåller ett par rent virtuella funktioner, så som handleMessage() och update(), men även ett par variabler som är viktiga för dialogsystemets funktionalitet. Dessa variabler är personalName och uniqueName. uniqueName är ett string-värde som fungerar som deras unika ID och är vanligt förekommande i de flesta spel. En mer annorlunda variabel är dock personalName. Detta är Entities personliga namn som absolut inte behöver vara unikt. Tar vi en mänsklig NPC som exempel skulle hans unika namn kunna vara ”Benny J. Peczek” medan hans personliga namn skulle kunna vara ”Benny”. Detta fungerar på ett liknande sätt för Entities som inte är mänskliga. Som exempel i detta fall kan vi ta en blomma, vars unika namn skulle kunna vara ”RedFlower63” medan dess personliga namn skulle kunna vara ”Red Flower”. Hur detta kan användas i ett spel finns nämnt i kapitel 5.1.5.

5.1.7 Player

Vid första anblick kan man undra varför det finns en klass vid namn Player i ramverket. Möjligtvis skulle ett mer passande namn vara BasePlayer, för det är det som är tanken med den här klassen – att den ska agera basklass för de spelare som skapas för just en specifik implementation. Eftersom Player innehåller några bitar som är ganska centrala för att en användare till fullo ska kunna utnyttja det här systemet har den implementerats som en basklass i arkitekturen i stället för att vara implementationsspecifik.

För det första innehåller Player en tillståndsmaskin för dess olika mentala tillstånd. Som grund innehåller den bara tillståndet Normal men kan fyllas ut med flera tillstånd efter önskemål. Denna tillståndsmaskin är den huvudsakliga anledningen till att Player har blivit en basklass men den innehåller även ett par variabler som används i kombination med de som kan finnas i HumanNPC. Dessa variabler är följande:

- Charisma – Ett värde för spelarens utstrålning. Detta modifierar (positivt eller negativt) förändringar i TempAnger som medförs i dialoger genom att helt enkelt adderas på värdet. För att dra en parallell till verkliga livet betyder detta att en person som har en starkare utstrålning har lättare att få folk att tycka om honom eller henne.
- Intimidation – Fungerar som Charisma, fast för TempAnger.

Förutom dessa två variabler finns även en string som avgör spelarens kön – förslagsvis male eller female. Vad som är viktigt att påpeka är att denna arkitektur har utformats efter tankegången att den ska användas för interaktion mellan Entities som är, åtminstone i närheten av, mänskliga och därmed har mänskliga attribut. Systemet kan dock användas för varelser som har helt andra typer av attribut och i sådant fall kan det argumenteras att dessa variabler kanske borde förbli implementationsspecifika och alltså inte finnas med i ramverket. Dock ska också nämnas att deras grundvärden är 0, vilket betyder att de inte har någon som helst inverkan på något annat i spelet så det går utmärkt att helt ignorera dem om man så vill.

Ett sista argument till varför Player ligger i ramverket är det att klassen Dialogue (mer om den senare) använder sig av en Player och därför kräver dess närvaro.

5.1.8 HumanNPC

Liksom för Player så är detta tänkt att vara en basklass för de NPC:er i implementationen som ska vara av mänsklig karaktär. Det skulle även gå bra att fylla ut registret av basklasser med, exempelvis, DemonicNPC, WerewolfNPC eller något annat som kan kräva en specifik form av hantering. Som tidigare nämnt är detta projekt dock fokuserat just på mänskliga NPC:er. Precis som för Player är HumanNPC även en av byggstenarna för klassen Dialogue så det är viktigt att, åtminstone i det här fallet, inkludera den i ramverket.

5.1.8.1 Tillståndsmaskiner

Det finns flera saker som gör HumanNPC men en som först borde pekas ut är att den innehåller tre tillståndsmaskiner, nämligen följande:

- EmotionalStateMachine – En tillståndsmaskin som består av de olika känslomässiga tillstånden en NPC kan befinna sig i. Vilka dessa ska vara kan specificeras i implementationen.
- FearStateMachine – En tillståndsmaskin som talar om ifall NPC:n är rädd eller inte. Denna går dock att utveckla vidare om man så känner för det och lägga till flera olika grader av rädsla. I ett exempel, som kommer att ses närmre på senare, används dock bara två lägen – rädd eller inte rädd.
- ActivityStateMachine – En tillståndsmaskin som avgör vad NPC:n för tillfället sysselsätter sig med. Liksom de andra två bör denna fyllas i under implementationen men ett uppenbart exempel på vad som borde finnas med i ett spel som använder den här arkitekturen är tillståndet Talking.

5.1.8.2 Startmeningar

En viktig del i det här systemet är att man som spelare ska kunna få olika svar om man säger samma sak flera gånger. Som en liten bieffekt av detta tänkande har NPC:n fått ge olika startmeningar beroende på om han eller hon har talat med spelaren tidigare – globalt så väl som i sitt nuvarande aktivitetstillstånd. För att enkelt lösa detta har NPC:n helt enkelt två booleska variabler. Den första sätts till true då NPC:n för första gången talar med spelaren. Exempelvis kanske han första gången presenterar sig för spelaren eller påpekar att han aldrig sett honom där tidigare. Den andra sätts till false varje gång NPC:n byter aktivitetstillstånd för att visa att spelaren ännu inte har talat med honom i sitt nuvarande tillstånd och det sätts inte till true förrän han gör just det. Exempelvis kanske NPC:n blivit tillsagd att vänta på spelaren i en gränd. Första gången spelaren dyker upp kanske NPC:n undrar varför han skulle vänta i gränden men detta är nog inget han kommer fråga varenda gång spelaren talar med honom.

5.1.8.3 Kontrollerade känslor

Några andra viktiga variabler i HumanNPC-klassen är de som kontrollerar skiftet mellan olika känslomässiga och skräcktillstånd. Som nämndes i kapitel **Fel! Hittar inte referenskälla.** behöver vi ett sätt att skilja på temporära och permanenta känslor. För att lösa detta har vi två integers vid namn TempAnger och PermAnger. Deras relativa värde bestäms av en tredje integer vid namn Tolerance. För varje alternativ spelaren väljer i en dialog kan TempAnger och PermAnger påverkas i positiv eller negativ riktning. Positiv riktning innebär att NPC:n blir mer arg medan negativ riktning innebär att han eller hon blir mindre arg. Det är TempAnger som bestämmer vilket känslomässigt tillstånd NPC:s EmotionalStateMachine befinner sig i för till tillfället. Om TempAnger sjunker under eller blir lika med 0 minskar ilskan med ett tillstånd. Om det å andra sidan stiger över Tolerance ökar ilskan med ett steg.

Varje gång ilskan tar ett steg i vardera riktning justeras TempAnger och PermAnger med tolerans för att bli relativt korrekta mot det nya intervallet. Om Tolerance exempelvis är 10, TempAnger är 4 och PermAnger är 7 så kan vi se vad som händer om vi minskar TempAnger med 5. TempAnger sjunker då under 0 så vi byter till ett mindre ilsket tillstånd. För att det inte ska förbli under eller lika med 0 modifierar vi det med Tolerance. Vi adderar alltså 10 till TempAnger och PermAnger. Nya TempAnger-värdet blir då 9 medan PermAnger blir 17. Här kan vi även se hur detta skulle kunna se ut i kod:

```
if(npc->getTempAnger() <= 0)
{
    npc->modifyTempAnger(npc->getTolerance());
    npc->modifyPermAnger(npc->getTolerance());
    if(npc->getTempAnger() <= 0)
    {
        npc->getEmoSM()->changeState(Loving::instance());
        return;
    }
}
else if(npc->getTempAnger() > npc->getTolerance())
{
```

```
        npc->modifyTempAnger( -(npc->getTolerance()) );
        npc->modifyPermAnger( -(npc->getTolerance()) );
        if(npc->getTempAnger() > npc->getTolerance())
        {
            npc->getEmoSM()->changeState(Neutral::instance());
            return;
        }
    }
```

Som vi ser i kodexemplet byter vi tillstånd ännu en gång om det skulle vara så att TempAnger ligger utanför ramarna även efter modifikationen. Om så är fallet blir det upp till nästa tillstånd i ordningen att göra en ny koll. De tillstånd som avslutar kedjan av känslor kommer självklart vara de som sticker ut mest i och med att de inte har något nytt tillstånd att gå över till. Då kan man välja att lösa det på flera sätt. Ett sätt att göra det på är att ha en gräns för hur långt bortom sina ramar TempAnger kan ta sig. Exempelvis kanske man vill att de avslutande känslorna på bägge sidor ska vara så pass starka att en person totalt kan drunkna i dem. I så fall kan det vara bra att sätta en relativt hög gräns eller ta till alternativ två och inte ha någon gräns alls. Vill man i stället att de ska bestämmas av Tolerance precis som alla andra kan man helt enkelt använda denna som en gräns.

Som setts ovan kommer PermAnger att agera individuellt från TempAnger. Detta innebär att man kan simulera olika känslomässiga fenomen hos NPC:erna, som till exempel att de kan bli irriterade även på folk de egentligen älskar. I just det exemplet skulle man kunna låta PermAnger sjunka väldigt lågt – för att resultera i en enorm brist på ilska (exempelvis kärlek) – medan man låter TempAnger stiga för att visa på en större irritation eller hat. PermAnger, som är NPC:s egentliga känsla, visar då att NPC:n egentligen tycker om spelaren men TempAnger, som är det värde som bestämmer tillståndet, säger ändå att han för tillfället kommer att agera som om han vore irriterad. För att kunna fastställa PermAngers betydelse kommer först en ny variabel in i bilden, nämligen en som valt att kallas CoolDownTimer. Då NPC:n inte befinner sig i en dialog med spelaren kommer den att lägga på all tid som passerar till sin CoolDownTimer. Varje gång den når sitt målvärde kommer den att skicka ett ModifyTempAnger-meddelande till sin EmotionalStateMachine. Vad tillståndsmaskinen då gör är att flytta TempAnger ett steg närmre PermAnger. Om skillnaden överstiger Tolerance, eller rentav råkar ligga precis vid en gräns, kommer detta innebära att EmotionalStateMachine till slut kommer att föra in TempAnger i ett nytt känslomässigt tillstånd och därmed automatiskt justera sitt tillstånd. Slutligen, efter att tillräckligt lång tid har passerat, kommer TempAnger vara detsamma som PermAnger. Ska man dra en parallell till verkliga livet här är detta en simulation av att NPC:n lugnat ned sig eller fått tid på sig att tänka över saker och ting.

5.1.8.4 Rädsla

NPC:s FearStateMachine fungerar på liknande sätt som dess EmotionalStateMachine. I det här fallet använder den sig av TempFear och PermFear, vilkas gränser bestäms av en integer som kallas för Courage. Som vi senare kommer att se använder sig exemplet bara av två skräcktillstånd – Afraid och NotAfraid. När TempFear stiger över Courage och tillståndet för tillfället är NotAfraid övergår det i stället till Afraid. Det går dock självklart att använda flera intervaller precis som för

EmotionalStateMachine, så som SlightlyAfraid eller VeryAfraid. Detta är dock upp till användaren av systemet att bestämma.

Något som skiljer skräcktillstånd från känslomässiga tillstånd är att NPC:erna här har fått en variabel för att motverka Players Intimidation (se kapitel 5.1.7). Denna variabel kallas CourageMod och vad den helt enkelt gör är att modifiera det fear-värde som kommer som input till hur NPC:n ska reagera. Anledningen till att detsamma inte görs för att motverka Players Charisma är den att rädsla är en stark känsla som låter folk komma undan med mycket som de vanligtvis inte borde göra. Även om en person vanligtvis är ganska feg – har ett lågt Courage-värde – kanske vissa omständigheter får honom eller henne att prestera bortom sina vanliga gränser. Detta skulle då kunna representeras med hjälp av CourageMod. Detsamma gäller självklart omvänt; en person som vanligtvis är väldigt modig kan under vissa omständigheter bli väldigt feg. Då hat och kärlek är känslor man sällan kan rå för behöver inte rädsla vara något man har en större möjlighet att ta kontroll över och därför har valet gjorts att utforma systemet på det här sättet.

5.1.8.5 Minnen

En sista, men minst lika viktig, del i HumanNPC-klassen är en vector av Memory-pekare. När spelaren säger något till en NPC i en dialog kollar NPC:n om det meddelandet redan finns i hans vector. Om så är fallet kan han enkelt avgöra hur många gånger meningen har sagt och sedan reagera därefter. Om det inte finns sparat, skapar han ett nytt Memory och lägger till det i vectorn.

Detta lilla system tjänar två viktiga syften. Det första är att det tillåter NPC:n att ge olika svar när spelaren ställer samma fråga flera gånger. Exempelvis om spelaren frågar hur han ska gå till väga för att klara av ett uppdrag kanske NPC:n ger ett ganska enkelt svar på frågan. Ställer spelaren samma fråga igen kanske NPC:n i stället ger ett mer detaljerat svar, under uppfattningen av att spelaren inte förstod hans första förklaring. Skulle spelaren ställa samma fråga 20 gånger kanske NPC:n i stället ger irriterade svar, ibland rent av ilskna. Det andra syftet har just med irritation att göra. Föreställ dig att spelaren har hittat ett alternativ i dialogen som får NPC:n att gilla honom mer. Spelaren skulle då kunna utnyttja detta på så vis att han säger samma sak om och om igen för att hela tiden göra så att NPC:n gillar honom mer och mer. Minnen kan här användas för att förebygga detta – för varje gång ett alternativ har sagts ökar mängden TempAnger och PermAnger det medför. Även om det till början var något som gav din karaktär ett positivt intryck hos NPC:n kommer utnyttjande av det att i stället medföra irritation.

5.1.9 Memory

Memory är en relativt enkel klass vars uppgift är att hålla reda på data gällande ett specifikt minne. Detta problem skulle kunna ha lösts med hjälp av tillståndsmaskiner. Varje meddelande som kan sägas i en dialog skulle helt enkelt få sitt eget tillstånd hos NPC:n som talar om ifall det har sagts eller inte. Har det sagts kan detta tillstånd även innehålla en liten räknare som talar om hur många gånger. Denna metod skulle ha ett par fördelar, bland vilka följande bör nämnas:

- Lösningen hade varit fin. Det vill säga att den vore enhetlig med lösningarna för känslor och rädsla i och med att den bygger på tillståndsmaskiner.

- Tillståndsmaskiner är lätta att föreställa sig visuellt och kan lätt ritas upp grafiskt, vilket kan vara till hjälp för en designer som inte är allt för insatt i programmeringen bakom det hela.

Uppenbarligen skulle detta ha blivit en hel kod som skulle skrivas eller på något sätt genereras. Om vi exempelvis säger att ett spel innehåller 200 NPC:er vilka man ska kunna tala med, utav vilka tio har större roller. Ska detta system med verklighetstrogna dialoger verkligen utnyttjas bör man förvänta sig att varje mindre dialog kommer att innefatta omkring 100 olika meddelanden medan de viktigare NPC:erna kan ha meddelanden som räknas i tusental. Detta skulle därmed kräva någonting mellan 20 000 och 30 000 tillståndsmaskiner. Förslagsvis använder man sig då av någon typ av grafiskt verktyg som tillåter att designers själva skapar tillståndsmaskiner till meddelandena för att slippa sätta programmerare på den ofantliga mängd upprepande kod som måste skrivas.

Med ett grafiskt verktyg för designers att själva rita tillståndsmaskiner för sina dialoger skulle detta kunna vara en acceptabel lösning (och den skulle ju dessutom vara fin!). Dock används inte denna metod på grund av olika anledningar. En designer som i slutändan kommer att fylla systemet med dialog bryr sig troligtvis inte om hur enhetlig den underliggande kodstrukturen är. Det han eller hon bryr sig om är hur smidigt det är att använda. Här skulle då möjligheten att rita upp tillstånd och låta verktyget generera den kod som behövs kunna komma till användning. Detta är dock att gå lite väl långt. Måhända att en designer kanske inte kan programmera men bara för det bör man inte anta att han eller hon är dum i huvudet. Vi behöver inte ge dem färgkritor för att de ska förstå.

I stället för att spendera en enorm tid på att implementera ett system för att skapa tillståndsmaskiner för vartenda meddelande i ett spel (där 20 000 till 30 000 faktiskt kan vara en enorm underdrift om det nu ska innefatta så mycket frihet som systemet är tänkt att användas för) är det i stället fullt acceptabelt att helt enkelt låta minnena lagras i form av enkla klasser. När ett meddelande sägs till NPC:n lagras det med sitt ID och det har en liten räknare som talar om hur många gånger meddelandet har sagts. Tar vi en snabb titt på implementationen av detta innebär det en minimal klass som högst tar en halvtimme att koda om du inte vet vad du håller på med, jämfört med ett system som ska kunna skapa och hantera tiotusentals tillståndsmaskiner efter ritade grafer och ändå ha exakt samma funktionalitet som den vi åstadkom efter den spenderade halvtimmen. Skillnaden detta innebär för våra designers är att de kan slänga bort sina färgkritor och återgå till att helt enkelt använda siffror i någon form av tabell (Chandler, 2005) för att markera hur många gånger ett meddelande ska ha sagts för att ge ett specifikt resultat. Detta resonemang kräver dock att en tabell med kolumner som kan närma sig tvåsiffriga tal inte får en designer att falla i koma.

Ett sista argument som kan användas mot dem som antyder att det fortfarande är för fult att inte använda sig av tillståndsmaskiner i hela lösningen av systemet är följande: projektet går ut på att med hjälp av känslomässig interaktion mellan spelare och NPC – som kan påverkas av spelarens eget sätt att spela ut sin roll – skapa ett system som kan ge en mer realistisk och dynamisk atmosfär i ett datarollspel. Det är alltså känslorna som följer av dialogerna och spelarens egen möjlighet att påverka detta som står i centrum. Minnen är inte känslor i sig, de är endast verktyg som är nödvändiga för att ge ett tillräckligt djup i interaktionen. De är nödvändiga på samma sätt som det är extremt nödvändigt att över huvud taget ha en NPC att tala med. Bara för att känslorna i sig hanteras i tillståndsmaskiner betyder det dock inte att NPC:n själv ska implementeras i form av en tillståndsmaskin – eller spelaren, för den delen. På grund

av detta är det inte ens fult att inte implementera dem med hjälp av tillståndsmaskiner, just på grund av att de är *verktyg för det egentliga systemet* – det är ett enkelt sätt att lagra ett par variabler mappade mot ett ID-värde.

Memory innehåller den följande uppsättningen variabler för att kunna utföra sitt syfte:

- TimeStamp – Den tid då minnet skapades.
- Duration – Den tid minnet ska kommas ihåg. Då nuvarande tiden minus minnets TimeStamp överstiger Duration tas minnet bort.
- Amount – Antalet gånger meddelandet i fråga har sagts under dess nuvarande existens.
- ID – Ett ID för det meddelande som ska kommas ihåg.

Dessa data kan användas med hjälp av ett enkelt interface som tillåter användaren att sätta nya tider, modifiera dess levnadstid och antalet gånger det har sagts samt att hämta de olika variablernas värden.

5.1.10 DialogueManager

DialogueManager är en template-klass vars uppgift troligtvis är ganska uppenbar bara genom att kolla på dess namn. Det är nämligen denna klass som hanterar alla olika Dialogues som skapas under körningen av applikationen. Denna hantering går till så att den sparar en vector av Dialogues samt en pekare till den nuvarande dialogen (om det finns någon sådan). En viktig detalj som bör nämnas här är att den endast sparar en Dialogue av varje Player-NPC-par, nämligen den senaste.

Då GameApp går in i ett InDialogue-tillstånd säger den åt DialogueManager att skapa en Dialogue mellan en NPC (i det här fallet en HumanNPC) och en Player. DialogueManager gör som den blir tillsagd och lägger in en ny Dialogue i sin vector och sätter CurrentDialogue till den nyskapade. När denna Dialogue sedan tagit slut är det dags för DialogueManager att lägga in den som den enda sparade Dialogue ur detta Player-NPC-par. Den kollar då ifall det finns några fler sparade och om så är fallet tar den bort den äldsta av de två. Anledningen till detta kommer vi att se senare i rapporten.

5.1.11 Dialogue

En Dialogue är vad som representerar en konversation mellan en Player och en NPC (i vårt fall, en HumanNPC). Förutom pekare till dessa innehåller den även en hel del annan nödvändig information för att kunna fungera i systemet, nämligen följande:

- StartTime – En integer som talar om när denna dialog skapades.
- EndTime – En integer som talar om när denna dialog slutade.
- Text – En string som representerar HumanNPC:s nuvarande meddelande.
- Alternatives – En vector som innehåller ID:n till de meddelandealternativ spelaren har att välja mellan.
- AltMessages – En vector som innehåller de faktiska meddelande mappade till sina ID:n i Alternatives. Just i mitt fall används denna på grund av hastighetsproblem – det tog helt enkelt för lång tid att leta reda på meddelandetexten i en extern fil under varje uppdatering. Därför sparas de i stället undan dem på en mer lättillgänglig plats.

- AltForm – Ett enum som talar om i vilken form de olika meddelandena ska visas. I exemplet används neutral, polite och rude.

Hur man bär sig åt för att spara meddelanden och dess olika variabler (exempelvis hur de påverkar TempAnger, osv.) kan avgöras under implementationen av varje specifik applikation. Troligtvis kommer det dock göras genom en extern fil eller databas av något slag, vilket, som sagt, är anledningen till att både ID:n och meddelanden sparas undan till varje alternativ i dialogen. Om så är fallet bör man även välja att spara undan ett filnamn eller annan vägvisare för att låta Dialogue veta vart den kan hitta dessa meddelanden.

En annan detalj du kanske undrar över efter att ha läst om de olika variablerna är AltForm. Detta är ett enum som används för att skapa en gammaldags form av tillståndsmaskin – en vanlig switch. Beroende på vilken AltForm vi befinner oss i så hämtas olika meddelanden för dialogen.

Då spelaren gör ett val i dialogen, det vill säga väljer ett alternativ, hämtas all nödvändig information för just detta meddelande; dess ID, text, eventuella Telegram som ska skickas i och med valet, om meddelandet ska kommas ihåg och i så fall under hur lång tid samt de olika grundmodifikationerna på NPC:s ilska och rädsla – de sistnämnda kan sedan komma att påverkas ytterligare av Player- eller HumanNPC-specifika attribut. Hur dessa värden hämtas beror till stor del på hur de sparas och det är därmed en implementationsspecifik process som inte behövs gå in på i mer detalj för tillfället. Något som däremot borde tittas lite närmre på är hur minnen påverkas av dialoger.

När ett meddelandealternativ har valts av spelaren avgörs om det ska kommas ihåg eller inte. Troligtvis är det så att det ska göra det, för att undvika utnyttjanden som diskuterades i kapitel 5.1.8.5. Antingen så anges en specifik livslängd för minnet eller så används ett standardvärde. Minnet får då en tidsstämpel som stämmer överrens med nuvarande tid. Vad som är viktigt att notera är dock att så länge som en HumanNPC befinner sig i en dialog med spelaren kan inte ett minne upphöra eller ett tempvärde jämnas ut mot sin permanenta släkting. Anledningen till detta är så att man inte kan sitta och vänta på att säga något tills HumanNPC har blivit på ett bättre humör eller tills han eller hon glömt bort något.

När dialogen sedan tar slut sker en del intressanta saker. För det första talar Dialogue om för DialogueManager att den har avslutats så att DialogueManager kan ta bort eventuella kopior av just detta Player-NPC-par den kan ha sparat i sin vector av Dialogue-pekare. Sedan sätter Dialogue sin sluttid till den nuvarande tiden i applikationen. Vad den sedan gör är att anropa funktionen updateMemoryTimeStamps() hos HumanNPC. Vad denna funktion gör är att den går igenom sin lista av Memory-pekare och letar reda på alla dem som har en starttid som är större än starttiden för just denna Dialogue. Om den hittar något eller några sådana minnen kan vi ta för givet att de skapades (eller uppdaterades) just under denna dialog. För att undvika att spelaren kan utnyttja tiden på det sätt som tidigare angavs – att han eller hon sitter och väntar med att välja alternativ för att få HumanNPC att glömma av sina minnen – sätter vi sedan alla funna minnens sluttid till densamma som för dialogen. Först när detta är klart säger vi till HumanNPC att sluta prata genom att skicka StopTalking i ett Telegram.

5.1.12 DTimer

Detta är bara en timer som applikationen använder sig av för att sätta ut, exempelvis, starttider och sluttider för olika dialoger. Den innehåller även funktionalitet för att översättas till en 24-timmars digitalklocka som kan användas för att ge NPC:er och spelare kunskap om dygnsrytmen i spelet. Det sistnämnda är dock ingenting som är nödvändigt just för denna arkitektur så det är inget vi behöver gå in djupare på.

5.1.13 Dialogskriptet

För att på ett smidigt sätt kunna skriva och ändra i dialoger utan att behöva implementera ett system för detta som bygger på existerande skript eller XML valdes att skapa ett nytt, fokuserat, skriptspråk. Syftet med detta skript var att kunna skriva dialog för arkitekturen i en extern fil av något slag men att även kunna hantera saker så som variabler och IF-satser.

Skriptfilerna skapas enkelt genom att döpa en vanlig textfil till UniqueName.dia, där UniqueName alltså är en NPC:s unika namn. Efter det delas filen in i olika stycken efter meddelande-ID:n, både för NPC:s meddelanden så väl som spelarens. En NPC:s meddelande markeras med hjälp av ett #-tecken medan spelarens meddelanden markeras med ett &-tecken. Efter detta tecken följer meddelandets ID och sedan ett värde som skiljer sig från NPC:s och spelarens. Efter NPC:s ID följer två siffervärden som talar om i vilket intervall, baserat på minnet av föregående meddelandeval, meddelandet ligger. Det första värdet, x, säger att föregående meddelande måste ha sagts x eller fler gånger för att det här meddelandet ska väljas ut. Det andra värdet, y, säger att föregående meddelande ska ha sagts färre än y gånger. Alltså måste spelarens föregående meddelande tidigare ha sagts ett antal gånger som ligger inom intervallet [x, y) för att detta meddelande ska vara korrekt. Om så inte är fallet letar parsern upp nästa meddelande-ID som stämmer överrens och ser på detta intervall. Ett exempel på ett meddelandeintervall som alltid kommer att vara sant (i = infinity) är följande:

```
#Start1IDLE 0 i
```

Exemplet visar även hur ett startmeddelande kan se ut. Det alternativ som väljs ut då en spelare först startar en dialog med en NPC ser ut så här: StartXCURRENTSTATE, där X är 1 om spelaren inte tidigare har talat med NPC:n i detta tillstånd, eller 2 om spelaren har gjort det.

Värdet som följer spelarens meddelande-ID är ett värde som talar om ifall meddelandet ska läggas i NPC:s minne eller inte. Detta avgörs av antingen T (true) eller F (false). Om fallet är T så följs detta av ett siffervärde som anger antal sekunder minnet ska existera i, där 0 är ett standardvärde. Exempel:

```
&IntroduceNoHello T6120
```

Vad som följer meddelande-ID:t är en trädstruktur över alla tillgängliga tillstånd. Dessa markeras med ett $-tecken, följt av tillståndets namn i stora bokstäver. Här finns även ett DEFAULT-värde tillgängligt som väljs ut ifall det tillstånd man söker efter inte hittas. Detta kan vara bekvämt då man exempelvis vill att en NPC ska svara likadant i alla tillstånd men inte vill skriva ut meddelandet flera gånger. Efter tillståndsnamnet följer, för NPC:er, själva textmeddelandet inom citattecken enligt följande exempel:

```
#Start1IDLE 0 i
$HAPPY ”Hello! I am happy.”
```

```
$ANGRY "Hello. I am angry."
```

För spelarens meddelanden ser det lite annorlunda ut här. Förutom sitt mentala tillstånd kan ju en spelares svarsalternativ även bero på vilken attityd han valt. Dessa olika alternativ listas efter tillståndet och markeras med hjälp av en *. Efter detta tecken följer alltså namnet på attityden i stora bokstäver, efterföljt av meddelandet inom citattecken, precis som för NPC:n. Sedan skiljer det sig lite från hur NPC:s meddelanden ser ut. Efter själva meddelandet följer nämligen fyra stycken variabler i integer-form. Dessa fyra värden talar om hur attributen ska påverkas hos NPC:n (sedan ytterligare modifierat av spelar- och NPC-specifika attribut så som Charisma, Intimidation och CourageMod) i följande ordning: TempAnger, PermAnger, TempFear, PermFear. Alternativt kan meddelandet även följas av ordet MENU (eller bara bokstaven M). Om sådant är fallet betyder det att meddelandet inte alls ska påverka några känslor, utan att det snarare är till för att hämta en lista över nya alternativ för spelaren att välja mellan. Detta kan vara användbart då det finns en stor mängd samtalsämnen och platsen på skärmen inte räcker till, eller rentav att antal siffror på tangentbordet inte räcker till. Ett spelarmeddelande kan då se ut enligt följande:

```
&IntroduceNoHello T6120

$NORMAL

*POLITE "It's a pleasure to meet you. My name is Benny." -2
-1 -1 -1

*NEUTRAL "My name is Benny." 0 0 -1 -1

*RUDE "My name is Benny, you better remember that." 2 1 1 0
```

Efter meddelandena, vare sig de är hos NPC:n eller spelaren, följer ett meddelande-ID för svaret, vilka markeras med ett +-tecken. NPC:n har uppenbarligen bara ett sådant svar medan spelaren kan ha en hel lista, vilka numreras i stigande ordning i spelet och mappas mot respektive siffertangent. Detta kan se ut på följande vis:

```
#Start1IDLE 0 i

$DEFAULT "Hello."

+Hello

+AskForMoney

+Leave
```

För att avsluta en dialog anges inget svar alls, eller så används det fördefinierade ID:t EXIT.

Nu har vi alltså en grund för skriptet som tillåter oss att skriva dialoger som stöds av arkitekturen. Ett par nya problem utöver dessa dök dock upp. Först och främst att man ska kunna hämta programspecifika variabler och skriva ut dem i texten. Detta löstes genom att använda en Defines-klass som fick ägas av Dialogue. Denna klass fick innehålla en lista på string-defines som returnerade olika variabler och värden som kunde ersätta define-värden i skriptet. Dessa värden markerades genom att omslutas av %-tecken, enligt följande exempel:

```
#Start2IDLE

$DEFAULT "Hello, %PLAYERNAME%. The time right now is
%TIME%."

+Leave
```

Dessa defines kan även användas i IF-satser. För att skapa en IF-sats använder man bara syntaxen IF, ELSE, ENDIF och ENDELSE (eller bara END i stället för ENDIF eller ENDELSE). Ifall en IF-sats innehåller ett ELSE behöver den inte avslutas med ENDIF, utan ENDELSE räcker. Efter IF och ELSE följer namnet på ett define, en operator (=, < eller >) och sedan värdet det ska stämma överrens med. Exempel:

```
#Start1IDLE
IF PLAYERGENDER = FEMALE
IF PLAYERCOINS > 5000
$DEFAULT ”Hello, rich girl.”
ELSE //Playercoins <= 5000
$DEFAULT “Hello, poor girl.”
+NotPoor
ENDELSE //Ends else for Playercoins > 5000
ELSE //gender = male
$DEFAULT “Hello, boy.”
ENDELSE //Ends else for Playergender = female
+Hello
+Leave
```

En sista funktionalitet som krävdes var möjligheten att skicka Telegram genom skriptet till entities i den virtuella världen. För att göra detta markerade man raden med ett !-tecken. Efter detta tecken följer namnet på mottagaren av meddelandet, där NPC eller PLAYER kan användas för att syfta på just den NPC eller spelare som är delaktig i just denna dialog. Efter mottagarens namn följer själva meddelandetypen och sedan ett eventuellt värde som ska följa med meddelandet. Exempel:

```
#IntroduceNoHello T6120
$NEUTRAL
*POLITE ”It’s a pleasure to meet you. My name is
%PLAYERNAME%.” -2 -1 -1 -1
!NPC msg_learnPlayerName %PLAYERNAME%
+NiceToMeetYouFormal
*NEUTRAL “My name is %PLAYERNAME%.” 0 0 -1 -1
!NPC msg_learnPlayerName %PLAYERNAME%
+NiceToMeetYou
*RUDE ”My name is %PLAYERNAME%, you better remember that.”
2 1 1 0
!NPC msg_learnPlayerName %PLAYERNAME%
+Insulted
```

5.1.14 FileReader

För att kunna läsa av skriptet skapades sedan en parser som helt enkelt fick heta FileReader. Denna parser använde sig sedan Dialogue av för att hämta dialogmeddelanden, alternativ och de olika värden som skulle användas för känslomodifikationer, meddelanden, etc. Det mesta sköttes genom en mängd olika

funktioner som i princip bara klippte ut och klistrade ihop olika strängar för att skapa textmeddelanden och variabler som översattes till integer-format.

För att lösa IF-satserna användes helt enkelt bara av en stack där resultatet av en jämförelse lades till på slutet i form av en bool. Denna bool användes för att tala om ifall vi befann oss i en IF-sats som var sann eller falsk. När man sedan nådde ett END poppade man bara ett element från stacken.

5.1.15 Defines

Som tidigare nämnt, i kapitel 5.1.13, fick Dialogue även ta del av en klass som helt enkelt bara fick heta Defines. Denna klass har bara ett enkelt interface som låter Dialogue skicka in ett definierat värde i string-form för att få det översatt till ett värde – möjligtvis då genom en eller flera funktioner som knutits till just detta fördefinierade värde. Exempelvis det definierade värdet TIME:

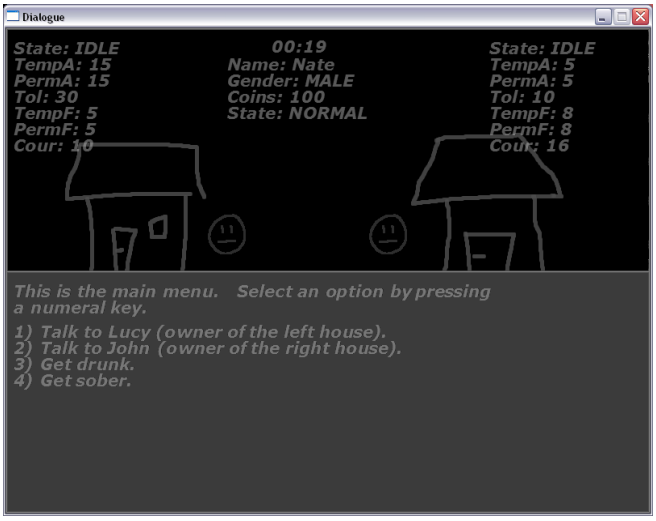
```
if(id == ”TIME”)
    return DTime->getTimeInHours();
```

5.2 Resultat för delmål 2: Utvärdering

För att se om arkitekturen mötte de kriterier som tidigare sattes upp skapades en testimplementation som använder sig av ramverket och fyller ut det med scenariospecifika detaljer. Nedan följer en presentation av detta exempelscenario samt en undersökning av de tidigare nämnda kriterierna.

5.2.1 Presentation av Dialogue-exemplet

Som tidigare planerat skapades en simpel virtuell värld för att ge exempel på hur dialogsystemet skulle kunna tänkas att användas. Du som spelare får välja mellan en kvinnlig och manlig karaktär och blir given ett slumpmässigt namn beroende på vad du valde. Interfacet du möts av är ganska simpelt (se Figur 7). Vi har en meny där olika alternativ listas upp, mappade mot varsin siffra. På den övre halvan av skärmen har vi grafik som representerar världen. Här kan vi se dess två andra invånare, vilka heter John och Lucy. John är något av en wanna-be gangster som ogillar snobbigt beteende och föredrar självsäkerhet och kaxig attityd. För att representera det här används ett lite grövre och mer slangfyllt språk för honom – vilket är anledningarna till en del fula ord i skärmdumparna. Lucy, däremot, är lite fin i kanten och ogillar ett ovårdat språk och föredrar att man är artig och tydlig.



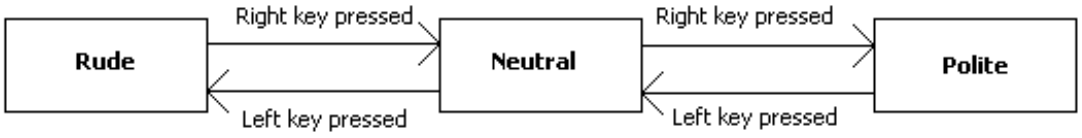
Figur 7. Här ser vi en bild på hur exempelimplementationen ser ut. I de övre hörnen finns även de två NPC:s data representerad för att lättare kunna se hur de påverkas av dialogalternativen.

Det finns även en del data utskriven i övre delen av skärmen. Upp till vänster ser vi Lucys attribut. Jämför vi dessa med Johns (uppe till höger) ser vi att Lucy är mer tolerant av sig – hon anser de flesta står lägre än henne på samhällets skala och förväntar sig inte så mycket från dem – och det kräver därför mer för att påverka hennes känslor. John, däremot, må gilla en attityd hos sina vänner men han har dessutom ett ganska hett temperament, vilket representeras av hans lägre toleransnivå. Vi ser dessutom att även om Lucy är mer tolerant är hon också mer lättskrämd medan John, som är van vid ett lite tuffare liv, är svårare att rå på. En annan detalj som bör nämnas är att deras startvärden i temporära och permanenta attribut har satts till mitten i det intervall de befinner sig i. Detta innebär att de, exempelvis, kan bli ännu mindre rädda för dig än vad de till en början är.

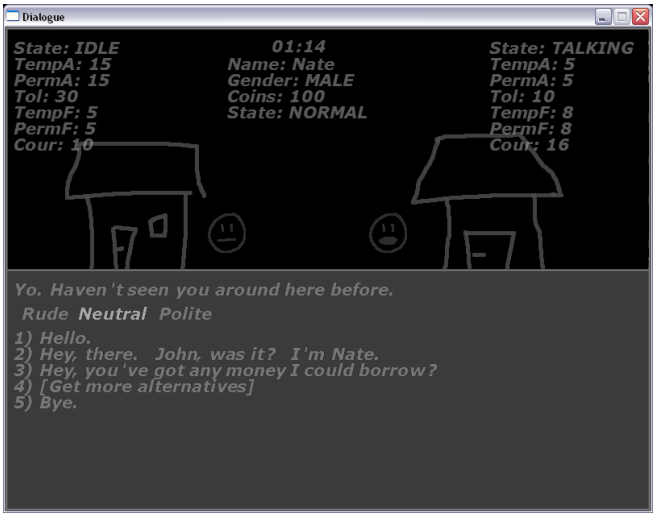
I mitten av skärmens övre halva har spelaren lite information om sig själv, så väl som en klocka. Denna klocka räknar den tid programmet har körts och är alltså inte det aktuella klockslaget i den lilla världen – vilket börjar räkna från 12:00. Då man befinner sig i en dialog saktar tiden ner till en tredjedel av den vanliga hastigheten. Detta för att det kan ta lite tid för spelaren att läsa igenom alla olika alternativ de har att välja mellan, vilket inte skulle representeras naturligt om klockan gick i normal hastighet. Förutom tiden kan vi även se vad vi heter, vilket kön vi är, hur mycket pengar vi har och vilket tillstånd vi befinner oss i. Det bör även påpekas att även NPC:erna har begränsat med pengar. Lyckas man övertala dem att låna dig pengar kommer de alltså inte ha några oändliga summor att dela ut, utan kan endast ge vad de faktiskt har – och absolut inte alltid allt av vad de har.

5.2.1.1 Spelaralternativ

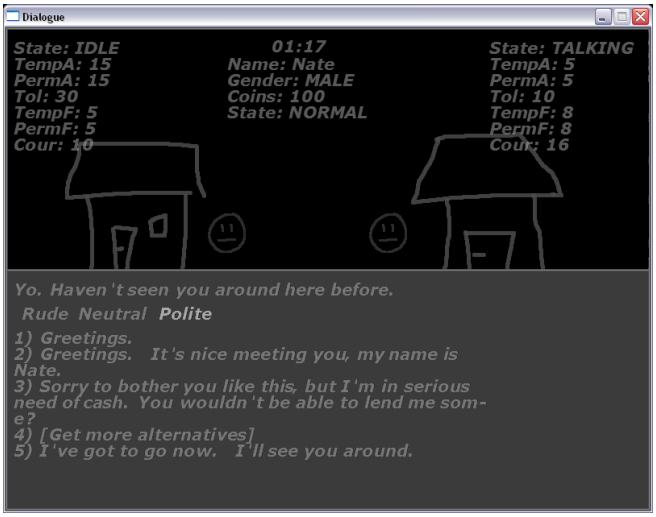
Väljer du att starta en dialog med någon NPC:erna går GameApp över till tillståndet InDialogue och NPC:n i fråga går in i aktivitetstillståndet Talking. NPC:s startmeddelande hämtas från dess .dia-fil (en vanlig textfil innehållandes skriptkoden som fått ändelsen .dia i stället för .txt) och spelarens olika alternativ listas upp. Här har man nu chansen att modifiera sina meddelanden mellan tre olika lägen (se Figur 8); rude, neutral och polite. Rude är en oartig och ofta förolämpade alternativform. Neutral är ett balanserat läge som varken indikerar förakt eller omtanke (rent tonmässigt, det vill säga – självklart kan meddelandet i sig fortfarande vara stötande för vissa NPC:er). Polite är en artig alternativform där spelaren faktiskt försöker visa att han respekterar NPC:n och inte vill framstå som stötande. Innebörden i de olika meningarna är desamma – vad som skiljer dem åt är alltså bara hur de sägs (se Figur 9 och Figur 10) och det är detta som slutligen bestämmer NPC:s reaktion samt svar på tilltalet.



Figur 8. Spelarens tre tillstånd för val av alternativform.



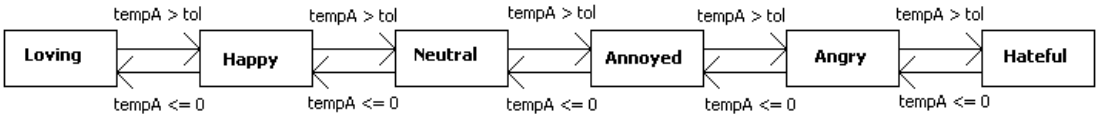
Figur 9. Här ser vi spelarens första alternativ i en neutral form.



Figur 10. Här ser vi spelarens första alternativ i en artig form. Innebörden är dock densamma som för neutral och rude.

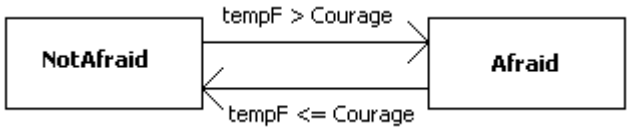
5.2.1.2 HumanNPC:s känslor

Som sagt påverkar spelarens alternativ NPC:s känslor på olika vis. Utvecklaren kan själv fylla i NPC:s känslo- och skräcktillståndsmaskiner med känslor. Som känslomässiga tillstånd används loving, happy, neutral, annoyed, angry och hateful (se Figur 11), där loving alltså är den mest positiva känslan och hateful är den mest negativa. Dessa gör sedan sina tillståndsbyten som tidigare planerat – då det temporära värdet blir lika med eller mindre än noll så sjunker ilskan, medan om värdet överstiger toleransen ökar den. De bägge ändvärdena fungerar självklart lite annorlunda eftersom de inte kan sjunka eller stiga mer. Här är det i stället gjort så att det permanenta värdet stannar vid den avslutande gränsen medan det temporära värdet kan fortsätta till den dubbla gränsen. Anledningen till detta är att exemplet ska kunna simulera hat och kärlek som kan gå bortom rationellt tänkande och alltså bli mycket svårare att överkomma.



Figur 11. HumanNPC:s olika känslomässiga tillstånd.

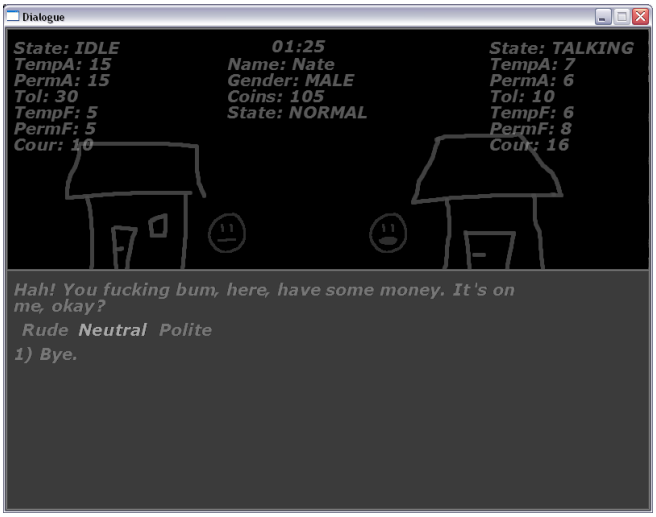
Förutom sina grundläggande känslor har ju NPC:erna dessutom möjligheten att bli rädda. Precis som för den känslomässiga tillståndsmaskinen får man själv bestämma tillstånd som ska ingå i skräcktillståndsmaskinen. Här används en ganska enkel booleansk lösning – antingen är man rädd eller så är man det inte (se Figur 12). Det går självklart att dela upp i en större intervall men det som syftas på med rädsla, i det här fallet, är en sådan rädsla som totalt tar kontroll över en och får en att sluta ta sina egentliga känslor i akt. En lättare form av rädsla kommer ju fortfarande att vara influerad av känslor så därför kan de lika gärna bli hanterade direkt i dialogen för sådana meningar som kan anses skrämma NPC:er till en viss del men inte bortom kontroll. När NPC:n väl går i det faktiska tillståndet Afraid överskrider detta som sagt de andra känslorna han eller hon har för spelaren.



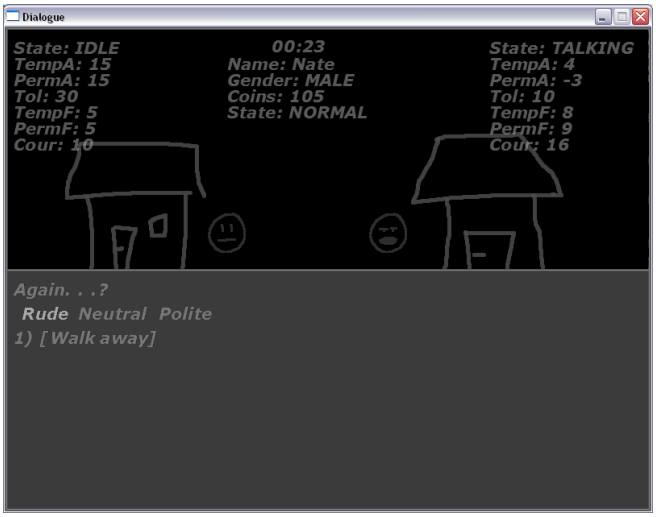
Figur 12. HumanNPC:s olika skräcktillstånd.

5.2.1.3 HumanNPC:s minnen

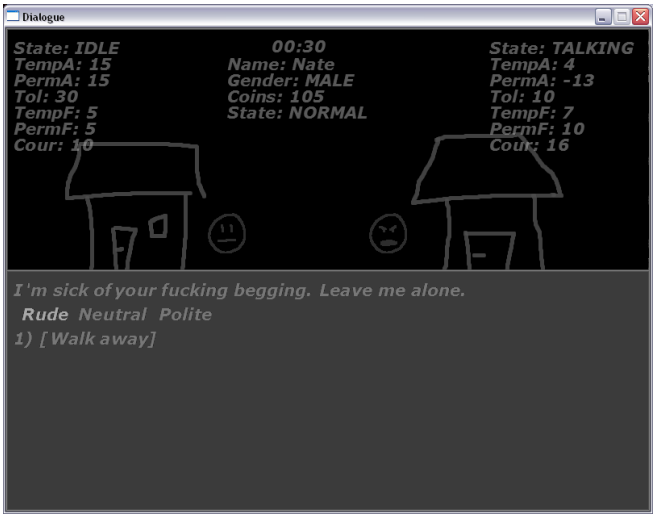
Hur minnena för en HumanNPC fungerar har egentligen redan gåtts igenom. Det som bör göras här är snarare att presentera hur de kan fungera i praktiken. För att tydligt illustrera detta ska vi ta en titt på ett exempel där en spelare frågar John om pengar. Spelaren börjar med att fråga honom artigt om hon (även om namnet i bilderna säger annorlunda antar vi för tydlighetens skull att spelaren är en kvinna) kan få låna lite pengar. Han befinner sig i sitt neutrala läge och slänger ur sig en liten kommentar men tar det mer som ett skämt och ger spelaren ett par mynt (se Figur 13). Sedan väljer hon att fråga honom igen men den här gången i ett neutralt läge. Han har kvar förfrågan i sitt minne så meddelandet som väljs ut är ett som sägs om hon redan ställt frågan en gång tidigare. Dessutom blir han nu lite irriterad, vilket visas i både grafik och ordval (se Figur 14). Han ger henne inte några pengar den här gången så hon väljer att fråga honom ännu en gång, den här gången med ett litet hot tillagt – det vill säga i en oartig form. Enligt minnet har hon ställt frågan två gånger tidigare så svaret som väljs ut är det som sägs just för ett sådant fall. Utöver allt tjat har John nu blivit arg på henne, vilket återigen syns både i grafik så väl som hans fula ordval.



Figur 13. Spelaren frågar John om hon få lite pengar. Hon gör det på ett artigt sätt och mest som ett skämt ger han henne fem mynt.

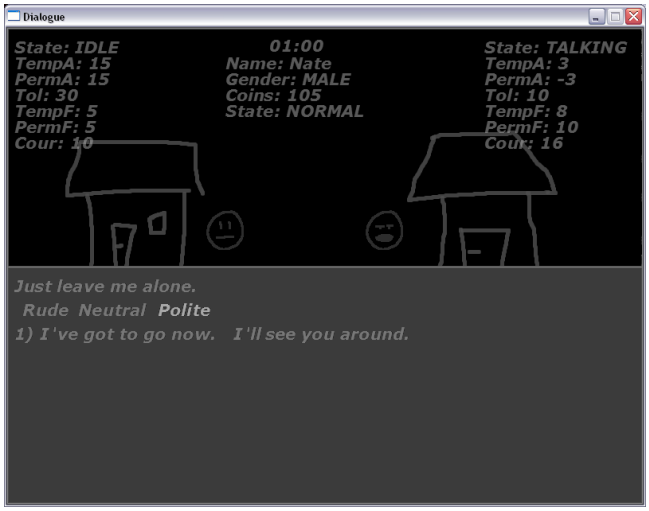


Figur 14. Spelaren frågar John en andra gång om hon kan få låna lite pengar, vilket han uppenbarligen kommer ihåg och blir irriterad.



Figur 15. En tredje förfrågan om pengar och John blir nu arg på spelaren.

I det här stadiet av dialogen har Johns temporära ilska långt överstigit hans permanenta. Spelaren väljer därför att vänta en liten stund med att fråga igen och låter honom lugna ner sig lite. När han fått lite tid frågar hon ännu en gång. Han kommer dock fortfarande ihåg att hon har frågat honom tidigare så meddelandet som väljs ut är det sista i raden av minnesbaserade pengasvar – i designen valdes alltså att sätta svaren i fyra olika steg, beroende på hur många gånger man frågat. Han må ha varit i ett neutralt läge efter att ha lugnat ned sig, men på grund av att han faktiskt kommer ihåg att hon frågat tre gånger tidigare blir han med en gång irriterad och vill fortfarande inte ge henne några mer pengar även fast hon nu frågade artigt.



Figur 16. Efter att John lugnat ned sig lite frågar spelaren återigen efter pengar. Det går dock inte för sig.

Som vi ser kan John, med hjälp av meddelande-ID:n, känna igen olika frågar oavsett vilken form de ställs i. Detta minne och igenkänning tillåter honom att ge olika svar ju fler gånger man frågar. Den gräns som sattes i detta fall var efter tre tidigare förfrågningar. Det vill säga, även om du frågar fyra eller tolv gånger kommer svaret väljas ut ur den sista listan. Detta antal går dock självklart att ställa in efter eget tycke med hjälp av skriptspråket. Det behöver inte heller vara i en intervall av ett svar per steg, utan han kan byta svarslista efter två eller tre förfrågningar. Förutom att svaren på detta vis går att göra oändliga har han ju dessutom sex olika känslolägen, plus ett extra i och med Afraid, vilket ger svaren en ännu större mångfald.

5.2.2 Har kriterierna mötts?

I kapitel 4.2 definierades ett par kriterier som skulle försöka uppnås i det här projektet. Dessutom beskrevs metoder för att testa kriterierna. Nu är alltså frågan, har de uppnåtts?

5.2.2.1 Flexibilitet

Det här kriteriet gäller frågan om man kan få ut olika typer av NPC-personligheter ur det här systemet och om de faktiskt kan representeras i dialogen. I exempelimplementation finns ju John och Lucy, som är två motpoler i åsikter och beteende. Rent variabelsmässigt har de designats så att John faktiskt ogillar en person som är överdrivet artig mot honom medan detta får Lucy att respektera och gilla dig. Tvärtom kan John faktiskt föredra en person som agerar lite mer kaxigt och överlägset medan Lucy tycker att det är avskyvärt, vilket medför et stor påverkan på världen. I exemplet där man frågar John om pengar ser vi direkt att man märker skillnad på när han är arg och när han är neutral – skillnaderna mellan Loving och Hateful är ännu större.

Det testades dessutom att implementera ett litet uppdrag man kunde få utav John om han befann sig i rätt känsloläge vid rätt tillfälle. Just detta krävde att han var på ett irriterat humör när du presenterade ditt namn för honom på ett kaxigt sätt. Genom olika alternativ kunde detta leda till ett litet ordkrig mellan dig och John vilket hade möjligheten att sluta i att John faktiskt såg upp till dig och de kontakter då påstod dig ha. Detta ledde till att han i stället föreslog att ni kunde göra lite affärer ihop. Detta är bara ett av ett oändligt antal möjliga exempel på hur olika känslor hos NPC:erna kan leda till olika möjligheter för spelaren.

I och med att det faktiskt går att se hur NPC:erna reagerar både i deras ordval och också grafiken så kan man själv välja att skräddarsy en dialog med olika svarsalternativ som passar just deras humör och personlighet. Dessutom går det ju, bland annat, att ställa in deras mod och tolerans. Därför är det till viss mån ganska flexibelt. Det går säkerligen att hitta brister om man tittar tillräckligt noga men målet som ville uppnås med just det här projektet är nått.

5.2.2.2 Utbyggbarhet

Det här kriteriet grundar sig i hur svårt (eller lätt) det är att lägga till nya känslomässiga tillstånd för HumanNPC:er men också i hur svårt det är att anpassa dem efter ickemänskliga varelser. I och med att det faktiskt sattes ihop en exempelimplementation på ganska kort tid utifrån ramverket som tidigare skapats blir den förstå frågan besvarad med en gång. Att använda sig av just de tillstånd som finns i detta exempel är absolut inget krav. Det enda som krävs är att deras namn anges som desamma i de externa dialogfilerna och inne i motorn. Att bygga ut känslospektrumet för HumanNPC:er är alltså hur lätt som helst så länge man vet hur tillstånd fungerar. Det är ju dock ingen automatisk dialoggenerator som skapar sina egna meningar utifrån olika känslor så ju mer man bygger ut känslorna desto mer arbete blir det också för dem som skriver dialogen. Detta skulle möjligtvis kunna ses som en begränsning i utbyggbarheten.

Hur blir det då om man vill skapa varelser som inte är mänskliga? Kanske en varelse som inte kan känna rädsla men har kvar alla andra känslor? Först bör dock nämnas att detta *inte* var vad ramverket skapades för men det kan ändå vara intressant att undersöka. För att lösa detta bör man först skapa en ny basklass för denna typ av varelse som får ärva från BaseGameEntity. Det är i sig absolut inget problem – det ända man behöver göra är att sätta upp de variabler och tillståndsmaskiner man vill använda. Problemet ligger i stället i Dialogue och dialogskriptet.

Dialogue använder sig specifikt av HumanNPC:s känslomässiga tillståndsmaskin och dess skräcktillståndsmaskin för att använda sig av FileReader för att hämta meddelanden från externa dialogfiler. Just dessa funktioner skulle behöva kodas om för att passa den nya uppsättningen tillståndsmaskiner. Dessutom kanske variablerna inte längre är desamma, i vilket fall man får skapa nya funktioner i FileReader för att kunna hämta flera typer av variabler än just de som redan har definierats. FileReader i sig är dock inget som tillhör den egentliga arkitekturen, utan snarare bara en teknik som användes för att kunna använda dialogsystemet i en exempelimplementation. Problemet blir alltså kvar i Dialogue. Har man tillgång till koden för Dialogue, och möjligtvis DialogueManager, är det inte så fruktansvärt svårt att lägga till funktioner för att köra dialoger mellan en Player och den nya varelsen. Dock är det inget som kan ske automatiskt.

Med denna diskussion i åtanke går det att påstå att kriteriet för utbyggbarhet till viss mån är nått när det gäller mänskliga NPC:er. Vill man däremot använda sig av något annat, som inte alls går att mappa till mänskliga känslor, kan det dock bli komplicerat. I många fall går det självklart att fuska. Vill man exempelvis inte att en varelse ska kunna bli rädd men ha kvar de andra känslorna kan man helt enkelt bara låta skräcktillståndsmaskinen innehålla ett enda tillstånd, nämligen NotAfraid.

5.2.2.3 Trovärdighet

Det här kriteriet ställer frågan om tillägget av känslor och minnen faktiskt får NPC:erna i ett spel som använder sig av denna arkitektur att kännas mer mänskliga.

”Mer” är nyckelordet i detta fall. Nej, de känns egentligen inte mänskliga – det märks fortfarande att det inte är en människa man talar med eftersom man har fasta alternativ och att vissa meddelanden i vissa fall kan upprepas ordagrant. De känns dock *mindre omänskliga* än de spel som togs upp i kapitel 2.4 på så sätt att man som spelare faktiskt får en bredare respons på sina påståenden och ordval känns det troligtvis *mer* mänskligt. Den största skillnaden är att i de ovan nämnda spelen kan man i förväg lära sig vilken mening som resulterar i vilket svar. Efter att ha talat med en NPC bara ett fåtal gånger blir det mer av en databas av information än en egentlig dialog. På detta vis kan man alltså säga att den föregående typen av dialoger känns ganska stela jämfört med vad man faktiskt kan åstadkomma med den här typen av system så länge dess bredd av känslomässiga alternativ faktiskt används.

I det här fallet påverkas NPC:er dock inte bara av vad du säger, utan även vilket känslomässigt tillstånd han eller hon befinner sig i, hur du säger något, hur många gånger du har sagt det och när du sist sa det. Det finns en hel del faktorer som påverkar och de gör att man får det mycket svårare att förvänta sig exakt vad som kommer att sägas och man blir därför mer vaksam över dialogen även om man talat med karaktären tio gånger tidigare. Ett exempel är det uppdrag som tidigare nämndes där det gällde att säga rätt sak till John vid rätt tillfälle. Även för skaparen av applikationen krävde det dussintals tester för att nå rätt alternativ. Måhända att det var ganska precisa värden som bestämde utgången men på grund av minne och känslor kan enstaka sekunder avgöra hur NPC:n i slutändan reagerar och det gör att man får den där typen av tveksamhet som man även kan få med människor i verkliga livet. Det är ett bra tecken för att det faktiskt känns mer verkligt.

6 Slutsats

För att utvärdera arkitekturen användes, som sagt, olika kriterier; flexibilitet, utbyggbarhet och trovärdighet. I och med att det går att skapa olika typer av personligheter för mänskliga NPC:er med hjälp av det här systemet kan man säga att det, åtminstone till en viss grad, är flexibelt. Vad som faktiskt var tänkt att göras just med det här projektet blev nått. Utbyggbarheten är till viss mån nådd när det gäller mänskliga NPC:er så länge man inte behöver bygga ut antalet *lager* av känslor utan endast olika känslor i sig. Det bör dock påpekas att den här arkitekturen har utvecklats just för mänskliga NPC:er så vill man använda sig av varelser med andra typer av känslouppbyggnader som inte bara går att byta namn på kan man få problem. När det gäller det sista kriteriet, nämligen trovärdigheten, bör det först och främst nämnas att arkitekturen inte på något sätt strävar efter en total trovärdighet till den grad då en spelare faktiskt kan tro att han eller hon pratar med en riktig människa. Så länge som dialogen är välskriven känns det däremot som en faktiskt dialog snarare än en informationsbank för spelaren att gräva i.

En positiv aspekt, ur användarsynvinkel, med just den här arkitekturen är att den tillåter utvecklarna själva att bestämma djupet av känslor en mänsklig NPC kan ha och det kan därmed anpassas för att justera tiden med arbete som kommer att behöva läggas ned på manusskrivande. Även på relativt låga nivåer kommer spelaren mest största sannolikhet dock att märka hur hans egna val påverkar NPC:er och att han då förhoppningsvis känner sig mer som en del av spelvärlden än han hade gjort om dialogerna var av förbestämd känslomässig respons. Arkitekturen ger alltså spelaren större möjlighet att påverka, manipulera och ställa sig in hos NPC:er – alltså mer interaktion. Dessutom ökar behovet av fokus som krävs från spelaren i och med att NPC:er har minnen vilket gör att dialoger även kan bli olika beroende på *när* spelaren talar med dem och vad de tidigare har sagt.

En annan positiv detalj som bör tas upp ligger i det kodmässiga användandet av det här systemet. Att lägga till eller ta bort känslotillstånd för NPC:er bör inte vara några problem för ens en medelmåttig programmerare. Dessutom går det lätt att modifiera ramverket för att lägga in fler variabler som påverkar känslomässig respons från NPC:er just på grund av att dialogerna bearbetas direkt av en helt egen klass, nämligen Dialogue. All typ av modifikation på respons finns alltså centrerad till en plats i källkoden.

Självklart finns det dock också nackdelar med det här systemet. En av de svagheter som lyste igenom som starkast under utvecklingen av exempelimplementationen var rädsla. Tanken med denna typ av känsla var att den skulle representera en total rädsla som helt överskrider en NPC:s andra känslor, exempelvis kärlek eller irritation. Det kan möjligtvis ha berott på dåligt skriven dialog, men övergångarna från ett vanligt känsloläge till rädsla blev ibland alldeles för tydlig. Det fanns liksom ingen bra indikation på att en NPC faktiskt började bli mer och mer rädd förrän han eller hon var så pass rädd att inget annat betydde något. I teorin antogs det att detta skulle gå att lösa genom att dialogen var skriven så att meddelanden som faktiskt skrämmer NPC:n ger någon typ av indikation på detta i deras svar. Visst skulle denna typ av lösning kunna fungera men den var absolut inte så bekväm som man skulle kunna önska.

En annan brist är en som troligtvis borde dykt upp i tankegångarna vid ett tidigare tillfälle. En spelares karaktär har möjligheten att gå in i olika mentala tillstånd som påverkar hur han eller hon säger saker, som exempelvis fylla. Detta är dock ingenting

som NPC:erna har stöd för. De kan ha olika känslor, de kan vara rädda och de kan befinna sig i olika aktivitetstillstånd men det finns alltså ingen tillståndsmaskin som kan förändra deras typer av respons beroende på mentala tillstånd. Ett exempel som skulle kunna inträffa i ett spel är att spelarens karaktär diskuterar något med en NPC på en bar och han eller hon får möjligheten att försöka få NPC:n berusad för att lättare kunna övertala han eller henne om något. Det skulle gå att ta omvägar kring detta problem genom att skapa aktivitetstillstånd så som SoberAtBar och DrunkAtBar. En smidigare metod vore dock om man slapp ge konkreta exempel och i stället bara kunde ha ett mentalt tillstånd för att just vara berusad. Det är dock en åsikt som kan skilja från person till person.

6.1 Framtida arbeten

Som nämnt under introduktionen är detta arbete mer tänkt som ett steg i en positiv riktning när det gäller dynamiska dialoger i dataspel snarare än ett försök att hitta den ultimata lösningen. Hur skulle man då kunna fortsätta utvecklingen på detta projekt med andra arbeten?

Ett projekt som skulle kunna göras för att förbättrad en här arkitekturen vore att sammanfoga rädsla och känslor med varandra. Det vill säga att bygga om arkitekturen för mänskliga NPC:s känslor så att rädsla kan bli mer gradvis och endast påverka meddelanden baserade på känslor i stället för att helt skriva över dem. Det skulle kräva ännu mer skrivande för designers som arbetar med dialogen eftersom varje rädslotillstånd skulle behöva infogas i varje känslotillstånd. Antalet olika svarsmöjligheter per fulländat meddelande – det vill säga ett meddelande som inte användas sig av default-värden – skulle alltså bli antalet känslor multiplicerat med antalet rädslotillstånd. Det skulle troligtvis dock fungera på en tolererbar nivå så länge som antalet känslotillstånd reducerades en bit från vad som användes i exempelscenariot för just det här projektet.

Ett annat möjligt projekt för att fortsätta det här arbetet skulle kunna vara att låta ramverket hantera andra typer av NPC:er än bara människor. Det skulle gå att använda en NPC-basklass för alla typer av NPC:er som kan tala och sedan låta de olika typerna ha olika former av tillståndsmaskiner för att representera hur deras beteenden fungerar. Exempelvis skulle ett spel i en futuristisk värld kanske vilja använda sig av robotar med ett mer binärt beteendemönster, vilket då alltså skulle kräva en annan uppsättning av tillstånd än de man kan tänkas ha för en människa.

När den här arkitekturen ändå tillåter så pass mycket respons både i form av känslor och minnen hos mänskliga NPC:er skulle det kanske vara intressant att låta NPC:er påverka varandra och inte bara bli påverkade utav spelaren. Man skulle alltså kunna tillämpa den här typen av system enbart på NPC:er i en spelvärld och tillåta dem att föra dialoger med varandra och påverka varandras humör och åsikter. Detta skulle troligtvis medföra att spelaren såg världen som mer levande vilket i sin tur skulle ge ännu bättre möjlighet för inlevelse. Han skulle inte längre stå i centrum för hela världen utan i stället bli ännu ett kugghjul i den större maskinen.

I arkitekturen som skapats i det här projektet kan en spelare växla mellan olika artighetsgrader helt fritt och dess påverkan har varit baserad enbart på vad som sagts, hur det sagts och hur många gånger det sagts. Här skulle det gå att fylla på med ytterligare en dimension, nämligen hur spelaren tidigare har sagt saker. NPC:er skulle kunna få möjlighet att, till olika grader, se mönster i spelarens beteende och val av hur han säger olika saker och därigenom få möjlighet att se igenom falska personligheter. Skulle en spelare exempelvis skifta väldigt mycket mellan att vara artig och oartig

mot en NPC borde denna upptäcka detta efter en stund. Detta skulle i sin tur kunna leda till att NPC:n får en brist på tillit hos spelaren eller helt enkelt bara ser honom som en opålitlig individ med grova humörsvängningar.

Arkitekturen skulle självklart också kunna anpassas för andra applikationer än just datarollspel. Först och främst i andra typer av spel där dialog förekommer men det skulle möjligtvis gå att göra ett bredare forskningsarbete där det kan förekommer som en form av instruktionsprogram för arbeten eller roller som kräver känslig interaktion med andra människor. De främsta områdena skulle i då fall kunna vara sjukvården. För att detta skulle bli passande bör det dock ligga en djupare forskning bakom användandet och att systemet i sig endast används som en hjälpreda för att realisera projektet.

7 Referenser

Adventure (1979) [Dataspel] Atari.

Avellone, C. (2005) *Dungeons and dialogue trees.* Tillgänglig på Internet: http://www.rpgdot.com/index.php?hsaction=10053&ID=1197 [Hämtad 07.05.15].

Avellone, C. & Colantonion, R. (2003) *RPG roundtable #2, part 1*. Tillgänglig på Internet: http://rpgvault.ign.com/articles/436/436852p1.html [Hämtad 07.05.15].

Bratko, I. (1990) *PROLOG programming for artificial intelligence.* Singapore: Addison-Wesley Publishers Ltd.

Buckland, M. (2005) *Programming game AI by example.* Texas: Wordware Publishing, Inc.

Chandler, R. (2005) *Organizing and formatting game dialogue*. Tillgänglig på Internet: http://www.gamasutra.com/features/20051118/chandler_01.shtml [Hämtad 07.05.15].

Colmerauer, A. & Roussel P. (1992) The birth of Prolog. *The second ACM SIGPLAN conference on History of Programming Languages*, 37-52.

DeSmedt, B. & Loritz, D. (1999) Can we talk? *AAAI 1999 Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 28-31.

Fallout 2 (1998) [Dataspel] Black Isle Studios.

Legend of Zelda: Twilight Princess, The (2006) [Dataspel] Nintendo.

Morris, T.W. (2002) Conversational agents for game-like virtual environments. *AAAI 2002 Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 82-86.

Neverwinter Nights 2 (2006) [Dataspel] Obsidian Entertainment.

RuneScape (2007) [Dataspel] Jagex Ltd.

Schwab, B. (2004) *AI game engine programming.* Boston: Charles River Media.

SmarterChild (2001) [Datorprogram] Colloquis. Tillgängligt på Internet: http://www.aim.com/ [Hämtat 07.05.15].

Star Wars: Knights of the Old Republic (2003) [Dataspel] BioWare.

Vampire: The Masquerade – Bloodlines (2004) [Dataspel] Troika Games.

Vampire: The Masquerade – Redemption (2000) [Dataspel] Nihilistic Software.

Weizenbaum, J. (1966) ELIZA – a computer program for the study of natural language. *Communications of the ACM 9, no. 1*, 36-45.

World of Warcraft (2004) [Dataspel] Blizzard.