Tygron Bachelor Project
Eindverslag

13 Juli 2009

Sid Mijnders (1308181) Tom Lotgering (1183052)

TU Delft, EWI
Vakcode: IN3405

Commissie:
Hans Geers (TU begeleider)
Alexander Hofstede (opdrachtgever)
Bernard Sodoyer (coordinator BSc)

Versie: 2

Voorwoord

Dit document is het eindverslag van ons bachelorproject aan de TU Delft. Het bachelorproject is het eindproject van de driejarige Bacheloropleiding, en moet worden afgerond om tot de (pre-)Master te worden toegelaten. Ons bachelorproject startte op 13 april 2009 en duurde tot 10 juli 2009, in totaal ongeveer 12 weken.

Als project hebben wij een opdracht uitgevoerd voor het bedrijf Tygron, ontwikkelaar van Serious Games. Onze contactpersoon bij het bedrijf was Alexander Hofstede. Wij kregen de opdracht om een systeem te maken waarmee Tygron gemakkelijk situaties uit de echte wereld in spellen kan weergeven en aanpassen. Tijdens deze opdracht hebben wij veel ervaring opgedaan m.b.t. het structureren van een project (organisatorisch) en het ontwikkelen van een ontwikkeltool (technisch).

Wij willen graag Alexander Hofstede bedanken voor het aanbieden van deze opdracht, en het beschikbaar stellen van alle informatie waarmee wij het systeem hebben kunnen maken tot wat het is. Ook willen wij onze begeleiders bij de TU, Hans Geers en Bernard Sodoyer, bedanken voor het lezen en (strict) controleren van de documentatie en voortgang van het project. Bovendien willen we de heer Sodoyer van harte bedanken voor de hulp in het vormen van de projectgroep.

Delft, Juni 2009

Tom Lotgering
Sid Mijnders

Inhoudsopgave

Samenvatting

Tygron is een bedrijf dat serious games, gericht op stedelijke gebiedsontwikkeling, maakt. Bij het maken van zo'n spel moet een situatie uit de echte wereld in een eerder gemaakt spel geïntegreerd worden. Er moeten dus aanpassingen aan dit spel worden gemaakt. Tygron zoekt nu een oplossing waarmee objecten en relaties uit de echte wereld op een modulaire manier in een spel gezet kunnen worden, zodat deze later, eventueel tijdens runtime, makkelijk zijn aan te passen.

Het doel van onze opdracht is een product te leveren waar objecten, relaties, en andere nuttige dingen in kunnen worden gevoerd om zo een model te maken. Dit model kan gebruikt worden in een spel om het gedrag van instanties van deze objecten te regelen. Er moet dus ook een uitvoerbaar deel in onze oplossing komen. Verder is gewenst dat er een graphical user interface meegeleverd wordt om de elementen en relaties te kunnen definiëren, en dat het systeem samenwerkt met de Serious Game Engine, die is ontwikkeld en wordt gebruikt door Tygron.

Het project heeft de volgende fasen doorlopen. Een orientatiefase, waarin wij ons bij Tygron op hebben georienteerd. Een analysefase, waarin het probleem is omgezet naar concrete requirements voor het systeem. Een onderzoeksfase, waarin wij onderzoek gedaan hebben naar mogelijke technieken die wij konden gebruiken. Deze "voorbereidingsperioden" werden gevolgd door een ontwerpfase, waarin de klassen uit de analysefase zijn uitgewerkt. Tot slot is dit ontwerp geïmplementeerd.

Om met ons systeem een spel te maken moet nu het volgende gedaan worden (ervan uitgaande dat de gebruiker met de GUI werkt). Als eerst moet een nieuw project aangemaakt worden. Dan kunnen er in het view-panel objecten, acties, events, etc. toegevoegd worden, en relaties worden gelegd tussen toegevoegde elementen. Het model kan tussentijds worden opgeslagen, en later weer worden geopend. Als het model naar wens is moet dit worden geëxporteerd. Tijdens dit exporteren wordt een jarfile gemaakt met daarin een xml bestand met het model en enkele classfiles. De jarfile die hier wordt geëxporteerd kan in een spelproject worden gebruikt om het model uit te voeren.

Er is ook ruimte voor uitbreidingen aan het systeem. De belangrijkste zijn: De mogelijkheid om tijdens het uitvoeren van een spel aanpassingen aan het model te maken, een verbeterde parsing van de invoer van de precondities, zodat de syntax hiervoor netter/ makkelijker wordt, en een extra laag om het model heen maken, omdat wat met ons systeem wordt gemaakt nog relatief low-level is.

Er waren enkele zaken waar we moeite mee hebben gehad tijdens dit project. Ten eerste zijn metamodellen nieuw voor ons. Het heeft extra tijd gekost deze te begrijpen en ons in te werken in het abstractieniveau hiervan. Ook zijn deze modellen moeilijker uit te leggen in de documentatie dan "normale" (niet-meta) modellen, en volgen bij het ontwerp de klassen minder vanzelfsprekend uit de eisen.

Ten tweede wist de opdrachtgever niet concreet wat hij wilde. Dit maakte het moeilijk om de eisen goed (naar de wensen van de begeleiders) op te schrijven en een ontwerp te maken. Het systeem volgt hierdoor wel de wensen van de opdrachtgever, maar werkt te dicht bij de code om het ontwikkelproces van een spel echt makkelijker te maken. Met onze ervaring met het maken van ontwikkeltools was het voor ons niet mogelijk om dit in de analysefase te voorzien. En hierom had dit project misschien anders moeten worden aangepakt; sneller de usability van een systeem testen, bijvoorbeeld door een vroeg prototype, om ervaring op te doen. De opdrachtgever had meer baat gehad bij een systeem

waarmee een zogehete "workflow" gemodelleerd kon worden.

Tot slot zou het geholpen hebben als er meer diagrammen van eerder gemaakte spellen aanwezig zijn. Dit zou het makkelijker gemaakt hebben te onderzoeken wat er precies met ons systeem gemaakt moet kunnen worden. Er waren geen diagrammen die de de relaties tussen spel-objecten beschrijven, en ook geen diagrammen die de relaties tussen actoren (welke met behulp van het systeem gemakkelijk geimplementeerd zouden moeten kunnen worden) beschrijven.

1. Inleiding

Deze inleiding beschrijft kort de context van het project, en de inhoud van dit rapport.

1.1. Context

Enkele maanden voor aanvang van het project was er contact gelegd met de opdrachtgever in verband met een potentiële opdracht voor het bachelorproject. Het is ons echter pas in April gelukt om, met de hulp van de heer Sodoyer, een projectgroep hiervoor te vormen. De aanvankelijke opdracht bij de opdrachtgever was toen al vergeven, maar in goed overleg en na enige communicatie tussen ons, de opdrachtgever en de heer Sodoyer, zijn we het uiteindelijk eens geworden over de onderstaande opdracht.

1.1.1. Opdrachtgever

Tygron is een bedrijf dat in 2005 is opgericht in Delft. Tygron heeft zich gespecialiseerd in het maken van "serious games". Deze spellen kunnen de spelers inzicht geven in processen en relaties tussen actoren betrokken bij stedelijke gebiedsontwikkeling. Klanten van het bedrijf zijn vaak gemeentes of bedrijven die willen weten welke gevolgen hun beslissingen kunnen hebben op hun omgeving, en op inwoners en andere bedrijven. Een serious game wordt als maatwerk gemaakt door de situatie van een klant te analyseren en te verwerken in een "bestaand" spel. Om dit te kunnen doen moeten de elementen, de processen en de afhankelijkheden hiertussen uit de echte wereld goed overeenkomen met die in de spelsituatie. Om het maken van serious games te vergemakkelijken is in 2006 de Serious Game Engine (SGE) gemaakt. Deze engine bevat functionaliteit m.b.t. simulatie en multiplaying, en een abstractie van de onderliggende 3D-rendering basis.

1.1.2. Aanleiding

Voor het implementeren van een situatie van een klant in een spel bestaat momenteel een ad hoc oplossing, die geschikt is voor een enkel spel, maar die zich niet leent voor hergebruik, of voor aanpassing van het spel aan afwijkende situaties tussen klanten, zonder het spel grondig te veranderen.

Het doel van het project is om de opdrachtgever in staat te stellen op een efficiënte manier maatwerk te leveren aan haar klanten, door bestaande spellen aan te passen aan de specifieke situatie van de klant.

1.1.3. Opdrachtomschrijving

Het doel van de opdracht is de gebruikers van het beoogde product van het project, programmeurs en proces analisten, de mogelijkheid te bieden op een efficiënte manier de werking van een spel aan te passen, door de relaties tussen, en eigenschappen van, bestaande spel-objecten te veranderen. De opdracht is dus om een dynamische, modulaire structuur (bestaande uit klassen) te leveren waarmee de programmeurs makkelijk objecten en acties van een spel aan kunnen maken en hier relaties en eigenschappen aan kunnen toekennen. Deze structuur moet er toe leiden dat situaties uit de werkelijkheid beter kunnen worden nagebootst in een spel. De opdrachtgever heeft aangegeven dat het nuttig

zou kunnen zijn als deze aanpassingen ook door de administrator van het spel, tijdens het spelen daarvan, kunnen worden gemaakt. Alhoewel niet noodzakelijk voor de goedkeuring van het product, zou dit idealiter door middel van een grafische interface moeten kunnen worden gedaan. De opdracht, zoals we deze per email hebben ontvangen, wordt in de bijlagen meegeleverd.

1.2. Verslag

Het doel van dit rapport is het presenteren van de resultaten en het verloop van het bachelorproject.

In hoofdstuk 2, projectorganisatie, wordt beschreven hoe wij dit project hebben aangepakt. Hier komt aan bod wat de planning was, welke documentatie wij hebben gemaakt, en hoe de communicatie is verlopen.

In hoofdstuk 3 worden de resultaten van de analysefase besproken; wat het doel van deze opdracht is, hoe op dit moment een spel bij Tygron wordt gemaakt, en wat het probleem hierbij is. Om voor dit probleem een oplossing te maken zijn de gebruikers van ons (toekomstige) systeem beschreven en functionele eisen opgesteld.

Hoofdstuk 4 bevat een beschrijving van de verschillende onderwerpen waar we onderzoek naar hebben gedaan. Bij elk onderwerp wordt eerst de werking uitgelegd, en vervolgens gezegd of we het gebruikt hebben in de implementatie.

In hoofdstuk 5 behandelen wij het ontwerp zoals dit er in de ontwerpfase uitzag. Eerst wordt de gekozen indeling van het systeem in packages gemotiveerd, vervolgens wordt de werking van het systeem uitgelegd, en tot slot wordt elke klasse in de model- en runtime packages in detail behandeld.

Hoofdstuk 6 beschrijft alles rondom de implementatie van het systeem; welke tools er zijn gebruikt, wat de planning was en wie wat heeft geïmplementeerd. Ook de aanpassingen aan het ontwerp in hoofdstuk 5 zijn hierin opgenomen. Verder wordt er in dit hoofdstuk extra aandacht besteed aan welke requirements uiteindelijk wel en niet geïmplementeerd zijn, en wat wij eventuele toekomstige gebruikers van het systeem aanraden om met de code te doen.

In hoofdstuk 7 wordt de kwaliteit van het systeem geanalyseerd. Dit gebeurt door eerst de algemene factoren voor software quality toe te passen op ons systeem, en vervolgens te behandelen welke onderdelen van het systeem zijn getest en welke tests er hierop zijn uitgevoerd.

Hoofdstuk 8 bevat conclusies over wat wij in het algemeen van het project vonden, dus wat we er wel of niet goed of leuk aan vonden, en wat wij achteraf anders gedaan zouden hebben.

In hoofdstuk 9 geven wij aanbevelingen aan het bedrijf over wat ze het beste met hun probleem kunnen doen; of dit gemaakte systeem nuttig is of dat er beter naar een andere oplossing gekeken kan worden.

2. Projectorganisatie

In dit hoofdstuk wordt besproken hoe wij de organisatie van dit project hebben aangepakt. In paragraaf 2.1 wordt gekeken naar de planning die wij in het Plan van Aanpak hebben opgesteld, en wat wij hier nu van vinden. Paragraaf 2.2 behandelt hoe de communicatie met de begeleiders van de TU en onze contactpersoon bij Tygron is verlopen. In paragraaf 2.3 wordt opgesomd welke documenten wij voor dit project hebben gemaakt.

2.1. Planning

In deze paragraaf wordt de planning van dit project besproken. Eerst wordt voor het gemak opnieuw de planning met zijn uitleg geplaatst zoals deze in het Plan van Aanpak stond. Vervolgens kijken wij hierop terug en geven we aan wat we hier naderhand anders aan hadden gedaan. Tot slot geven we een korte beschrijving van alle fases en geven we een benadering van de uiteindelijk gevolgde planning.

2.1.1. Oorpronkelijke planning

Het project zal uit vier fasen bestaan:
- In fase 1, "de orientatie", zal (literatuur-)onderzoek worden gedaan naar bestaande oplossingen voor het probleem, en zal waar nodig het bestaande systeem worden bestudeerd om integratie mogelijk te maken.
- In fase 2, "het ontwerp", zal een ontwerp voor het op te leveren systeem worden gemaakt.
- In fase 3, "de implementatie", zal het ontwerp daadwerkelijk worden geïmplementeerd.
- In fase 4, "de conclusie", zal de acceptatie en presentatie van het product, dan wel het rapport omtrent het project centraal staan.

De planning is als volgt:

Week	Fase
1 t/m 3	Orientatie
4 t/m 6	Ontwerp
7 t/m 11	Implementatie
12	Conclusie

2.1.2. Commentaar

Bovenstaande tabel geeft echter niet goed weer welke fase er daadwerkelijk wanneer heeft plaatsgevonden. Het is misschien zelfs zo dat wij de beschrijving hierboven niet eens meer correct zouden willen noemen. Er staat namelijk dat de eerste fase, de orientatiefase, bestaat uit het zoeken naar een oplossing voor het probleem. Er staat echter nergens aangegeven wanneer wij onderzoeken wat het probleem precies inhoudt, en wat de bedoelingen van de opdrachtgever zijn, wat toch een belangrijk deel uit zou moeten maken van de orientatie. De orientatiefase zou eigenlijk opgesplitst moeten worden in drie fasen:

- orientatie, waarin wij het probleem en de wensen van het bedrijf vaststellen.

- analyse, waarin wij de informatie uit de orientatiefase analyseren en omzetten in eisen.
- onderzoek, waarin wij onderzoeken wat nodig is om de eisen te implementeren.

We hadden bovendien in het maken van de planning geen rekening gehouden met de Mei vakantie, waardoor de uiteindelijke planning niet geheel overeen kwamen met de oorspronkelijke planning.

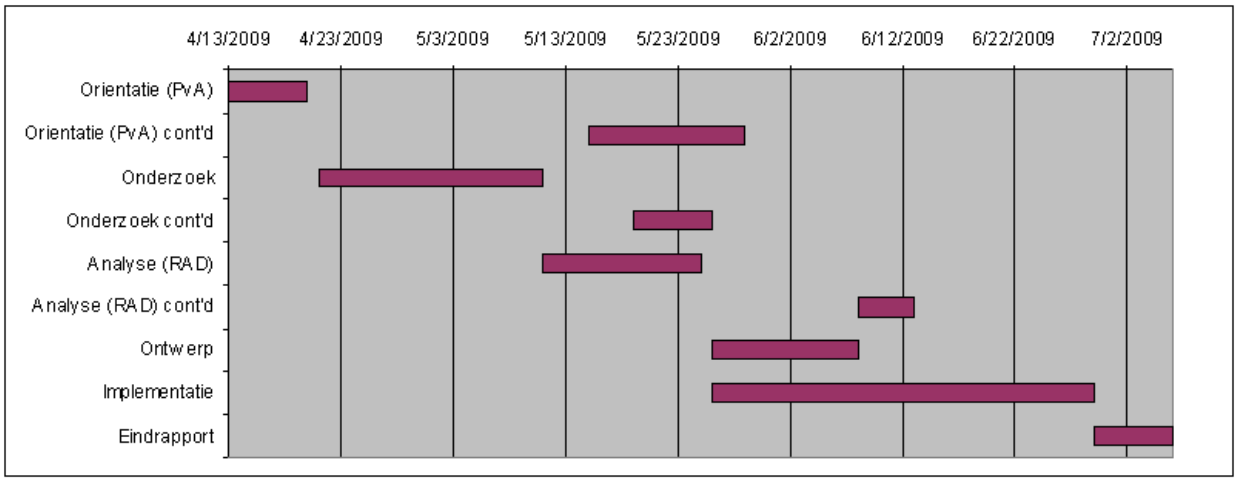
2.1.3. Uiteindelijke planning

Orientatie: Het was voor ons al snel duidelijk wat het bedrijf precies wilde. Het heeft ons echter meerdere versies van het plan van aanpak gekost om dit ook duidelijk op te schrijven, en daardoor zijn we hier uiteindelijk nog behoorlijk lang mee bezig geweest.

Onderzoek: Hier waren wij in de tweede week mee begonnen, en hadden na een paar weken gevonden wat wij nodig hadden voor de basis van de implementatie en het ontwerp van ons systeem (dit is gebaseerd op het Adaptive Object Model, hierover meer in hoofdstuk 4 over het onderzoek). Wij hadden echter nog niet genoeg onderzoek gedaan naar de werking van de SGE, wat nodig was om een koppeling te kunnen leggen met ons systeem. Om deze reden is hier later, in de weken 7 en 8, nog extra onderzoek naar gedaan. Ook zijn tijdens de implementatiefase nog enkele dingen onderzocht, zoals het tijdens runtime parsen, compileren en linken van stukjes java, maar dit vinden wij niet onder onderzoek vallen.

Analyse: deze fase liep sterk uit. Omdat wij in eerste instantie niet duidelijk genoeg hadden beschreven wat het bedrijf wilde en hoe ons systeem daarbij zou helpen, duurde het lang voor de documentatie hiervan werd goedgekeurd en moest een deel van het RAD bijna geheel herschreven worden. De nieuwe indeling ziet er als volgt uit:

Taken	Begin Datum	Duur (dagen)	Eind Datum
Orientatie (PvA)	4/13/2009	7	4/20/2009
Orientatie (PvA) vervolg	5/15/2009	14	5/29/2009
Onderzoek	4/21/2009	20	5/11/2009
Onderzoek vervolg	5/19/2009	7	5/26/2009
Analyse (RAD)	5/11/2009	14	5/25/2009
Analyse (RAD) vervolg	6/8/2009	5	6/13/2009
Ontwerp	5/26/2009	13	6/8/2009
Implementatie	5/26/2009	34	6/29/2009
Eindrapport	6/29/2009	7	7/6/2009



Figuur 1. *Gantt diagram van het verloop van het project.*

2.2. Communicatie

Nadat de opdracht per mail was opgestuurd is eerst een introductieoverleg gehouden met de projectbegeleider (Hans Geers). Dit was ook de eerste keer dat de projectleden elkaar ontmoetten. Nog diezelfde week is een plan van aanpak gemaakt, en het eerste gesprek met de contactpersoon bij het bedrijf (Alexander Hofstede) gehouden om de opdracht helder te krijgen.

Voor de rest van het project is vrijwel iedere week een overleg met Geers gehouden, waarbij de documenten die die week per mail waren ingeleverd werden besproken.

Communicatie met het bedrijf verliep onregelmatig. Wij zijn enkele malen bij het bedrijf langsgegaan, meestal wanneer we dringend vragen hadden, en in een van de laatste weken om een korte demonstratie te geven. Ook is het een paar keer voorgekomen dat we per email vragen stelden en antwoord kregen. Op beide manieren kregen we naar behoefte de informatie die we nodig hadden.

2.3. Documentatie

Er zijn voor dit project verscheidene documenten gemaakt. In de eerste week werd het Plan van Aanpak ingeleverd, waarin de opdracht van het bedrijf en de planning werden beschreven. Vervolgens is na de derde week het onderzoeksverslag ingeleverd. Het grootste gedeelte van de tijd zijn we bezig geweest met het RAD. Dit is het belangrijkste document uit de analysefase. Hierin worden de eisen aan het product vastgesteld, en een begin aan het ontwerp gemaakt. Dit ontwerp was in het RAD al redelijk uitgebreid gedaan, en kon hierna makkelijk worden uitgewerkt in het Design Document. De laatste weken is tijdens de implementatie ook gewerkt aan een testrapport, waarin de tests worden besproken die wij op onze klassen hebben uitgevoerd. Als afronding van dit project is in de laatste week bovendien dit rapport geschreven.

Alle documenten zijn nagekeken door dhr. Geers en/of dhr. Sodoyer, en de meeste documenten zijn ook besproken met dhr. Hofstede.

3. Probleemstelling en analyse

In dit hoofdstuk wordt de analyse-fase van ons project behandeld. Eerst komt het doel van de opdracht aan de orde. Vervolgens worden de huidige situatie bij het bedrijf en de probleemstelling besproken. In paragraaf 3.3 en 3.4 worden de gebruikers van ons systeem beschreven en de eisen die wij hebben opgesteld naar aanleiding van de probleemstelling. Tot slot reflecteren wij kort op deze fase.

3.1. Doelstelling

Het doel van het project is de opdrachtgever in staat te stellen op een efficiënte manier bestaande spellen aan te passen aan de specifieke situatie van een klant.

3.2. Huidige situatie

In deze sectie wordt het huidige proces bij Tygron omtrent het maken van een spel beschreven.

Tygron maakt voor klanten spellen op maat. Voor sommige klanten worden geheel nieuwe spellen gemaakt, en voor sommige klanten wordt een bestaand spel aangepast. De klant geeft over het algemeen zelf aan welk van de twee aan de orde is. Als de klant een bestaand spel kent en daarmee wil spelen wordt het aangepast, en anders wordt er een nieuw spel ontworpen.

Als het bedrijf een opdracht krijgt moet eerst de situatie van de klant geanalyseerd worden. Onder situatie vallen de actoren die een rol spelen en de problemen die deze actoren met elkaar of met hun omgeving (geografisch gezien) hebben. De analyse wordt door een proces analist gedaan. De proces analist onderzoekt de situatie van de klant en brengt hierover een rapport uit aan de opdrachtgever. Wat in een rapport staat is afhankelijk van de wensen van de klant. Als de klant bijvoorbeeld aanpassing van een bestaand spel zoals het "Watergame" wil, wordt vooral gekeken naar welke actoren (met betrekking tot het onderwerp van het spel) voor die klant van belang zijn en wat voor problemen en belangen zij hebben. Dit gebeurt bijvoorbeeld door interviews te houden en te vragen met welke actoren wordt samengewerkt. Hieruit worden ook de relaties tussen de actoren, en de processen waarbinnen die samenwerking valt, duidelijk. Voor bijzonder complexe (bedrijfs-)processen wordt een diagram gemaakt dat het proces beschrijft, bijvoorbeeld een activity diagram of EDPC diagram. Er wordt echter al snel begonnen met het ontwerp van (de aanpassingen aan) een spel dat hierop aansluit, en het uitvoerig in kaart brengen van de situatie van de klant is geen doel op zich.

Vervolgens wordt er in de ontwikkelomgeving, zowel voor het maken van een nieuw als het maken van een aangepast spel, een heel nieuw leeg project aangemaakt. Hier worden, naast het programmeren van nieuwe objecten en relaties, van eerdere projecten/spellen stukken code naartoe gekopieerd. Er kan dan bijvoorbeeld gedacht worden aan een klasse als Building of actoren die in meerdere spellen terugkomen. Op deze manier wordt een spel verkregen dat aangepast is aan de klant.

Het testen van het spel gebeurt als volgt. Eerst wordt door de programmeurs zelf getest wat ze programmeren. Vervolgens wordt intern, door iedereen die aan het spel meewerkt, getest. Dit gebeurt door voor belangrijke stukken code, bijvoorbeeld algoritmes, unit tests te maken of door scenario's te doorlopen. Tot slot worden er testsessies georganiseerd binnen het bedrijf of met hulp van ervaren testers.

3.3. Probleemstelling

Het maken van kleine aanpassingen aan een spel, voor een klant, kost relatief veel moeite. Er wordt een geheel nieuw project voor aangemaakt en het spel wordt gekopieerd. Vervolgens moeten delen van de code van het spel worden herschreven, om bijvoorbeeld actoren toe te voegen of te verwijderen, of om andere items toe te voegen.

Dit proces is arbeidsintensief; er is een programmeur voor nodig ook wanneer er geen nieuwe functionaliteit wordt toegevoegd, maar alleen bestaande functionaliteit wordt aangepast. Onderhoud is lastig, omdat er veel replicatie van code plaats vindt.

3.4. Gebruikers

- **(Game-)Developer**: De persoon die ons systeem gebruikt, door ofwel handmatig (door te programmeren) ofwel met de meegeleverde GUI klassen, acties en relaties te definiëren. De developer gebruikt alleen de functionaliteit van ons systeem die de GUI beschikbaar stelt.
- **Programmeur**: De programmeur werkt buiten de scope van ons systeem. Hij programmeert bijvoorbeeld, aan de hand van een door ons geleverde interface, de effecten van acties en maakt aanpassingen aan de engine/het spel/etc. De programmeur is integreert ons systeem met de spellen die worden gemaakt. Een deel van ons systeem kan alleen worden uitgevoerd vanuit de context van een spel. De programmeur is verantwoordelijk voor het opzetten van, en gebruik vanuit, de context. De interactie van de programmeur met ons systeem door middel van code in een spel is vereist, en zal daarom ook worden beschreven in dit document.
- **Administrator**: De administrator is een speciale actor die een spelsessie begeleidt. Hij heeft meer rechten dan een reguliere speler en kan tijdens het spelen aanpassingen maken door modules in en uit te schakelen. Dit is mogelijk met de dynamische functionaliteit van ons systeem.

3.5. Functionele eisen

Must have:

Modelleren:

M1. Met het systeem kan een nieuw type spel-object worden gespecificeerd. Een spel-object type heeft de volgende eigenschappen:

M1.1. Het heeft een naam.

M1.2. Het heeft velden. Een veld in een spel-object type moet een naam en een specifiek type hebben. Dit type kan een spel-object type zijn of een klasse.

M1.3. Het kan annotations hebben.

M1.4. Het kan een sub-type zijn van een ander type.

M2. Met het systeem kunnen de eigenschappen van een spel-object type worden gewijzigd. (zie **M1.1**, **M1.2**, **M1.3**)

M3. Met het systeem kan een spel-object type aan een spel-model worden toegevoegd (zie Glossary).

M4. Het systeem ondersteunt het verwijderen van een spel-object type uit een spel-model. De specifieke functionaliteit van dat spel-object moet uit het spel-model worden verwijderd, maar de overige spel-functionaliteit (ook datgene wat gedeeltelijk van dit spel-object afhankelijk was) moet blijven functioneren.

M4.1. De acties van het spel-object type worden verwijderd.

M4.2. Spel-object types die sub-type zijn van het verwijderde type blijven bestaan, maar zonder de functionaliteit van het verwijderde spel-object type over te erven.
M4.3. Andere spel-object types die een veld hebben met het verwijderde spel-object type, werken nu alsof ze dit veld niet meer hebben.

M5. Het spel-model bevat een hiërarchische (boom) structuur van "modules".
M5.1. Een module kan worden aangemaakt onder een bestaande module.
M5.2. De naam van een module kan worden gewijzigd.
M5.3. Een module (inclusief inhoud, zie **M5.4**, **M4**) kan worden verwijderd.
M5.4. Spel-object types worden altijd bevat door een module.
M5.5. Het is mogelijk een bestaande module uit een ander spel-model te importeren.
M5.5.1. De inhoud van de geïmporteerde module kan worden gekopieerd naar het huidige spel-model. en:
M5.5.2. Het huidige spel-model kan refereren naar de geïmporteerde module. Wijzigingen in de module hebben dan effect in het spel-model dat ernaar refereert.

M6. Het systeem ondersteunt het toevoegen van een causaal effect en een noodzakelijke voorwaarde voor dat effect aan een spelgebeurtenis.
M6.1. Een causaal effect kan als Java code in een .java document worden toegevoegd.

Uitvoeren:
M7. Het gemaakte spel-model kan worden uitgevoerd vanuit een spel.
M7.1. Wanneer een event plaats vindt wordt, mits de noodzakelijke voorwaarde waar is, het in het spel-model gedefinieerde effect uitgevoerd.
M7.2. Ondersteunt het instantiëren van spel-object types gedefinieerd in het spel-model.
M7.3. Een waarde van een veld van een spel-object kan worden gezet mits het type van de waarde overeenkomt met het type van dat veld.
M7.4. Een waarde van een veld van een spel-object type kan worden opgevraagd.

Should have:
S1. Functionele eisen **M1.1**, **M1.2**, **M1.3**, **M2**, **M3**, **M4**, **M5**, **M6** worden ondersteund door middel van een grafische gebruikersinterface.

S2. De interface kan de volgende relaties tussen spel-objecten en effecten visualiseren:
S2.1. Spel-object type is van invloed op voorwaarde voor causaal effect (zie **M6**).
S2.3. Spel-object type bevat ander spel-object type (zie **M1.2**).
S2.3. Spel-object type is sub-type van ander spel-object type (zie **M1.4**).

S3. Het systeem sluit aan op de bestaande spel-architectuur en SGE; dat wil zeggen:
S3.1. De functionaliteit voor functionele eis **M7** (inclusief **M7.1**-**M7.4**) is beschikbaar vanuit een spel op basis van SGE. (zie non-functionele eisen 3.3.6)
S3.2. Een veld van een in het spel-model gedefinieerd spel-object type kan als type een bestaand spel-object type hebben.
S3.3. Functionele eis **M6** kan gebruik maken van bestaande spel-gebeurtenissen.

Could have:
C1. (Developer, Programmeur) Het effect in functionele eis **M6** kan bestaan uit een keten van andere effecten.

Would like to have (but probably won't):
W1. (Administrator) Een module in een spel-model kan worden uitgezet tijdens het uitvoeren van een spel dat van dat spel-model gebruik maakt.
W1.1. De inhoud van de module wordt (vergelijkbaar met functionele eis **M4**) uit het spel verwijderd.

W1.2. Een module die volgens functionele eis **W1** is uitgezet, kan weer worden aangezet.

W2. (Administrator) De "voorwaarden" in functionele eis **M6** kunnen tijdens het uitvoeren van een spel gedeeltelijk worden uitgeschakeld. Effecten die hiervan afhankelijk waren worden dan uitgevoerd ook als niet aan de uitgeschakelde voorwaarde wordt voldaan.

3.6. Reflectie

Het systeem dat we wilden maken is gebaseerd op een metamodel, waar we eigenlijk nog geen ervaring mee hebben. We moesten geregeld op een hoger niveau van abstractie denken dan normaal gesproken het geval is voor programmas. Het bleek hierdoor ook een uitdaging om de werking van het systeem te beschrijven.

Het schrijven en bespreken van de documentatie verliep niet vlot, omdat het ons pas in een laat stadium is gelukt het doel van het project aan de begeleidend docent uit te leggen, waardoor deze fase uitliep en we weinig tijd over hadden voor de implementatie. Het was hierbij makkelijk geweest als er meer diagrammen beschikbaar waren geweest die de werking van een spel van Tygron beschreven.

4. Onderzoek

In dit hoofdstuk wordt het onderzoek beschreven dat wij hebben gedaan om een ontwerp te maken dat aansluit op de (functionele) eisen. Dit is nodig om sommige delen en ontwerpbeslissingen van het ontwerp in hoofdstuk 5 te begrijpen. De volgende onderwerpen komen aan de orde:

- Het Adaptive Object Model (AOM)
- Event Driven Process Chains (EDPC)
- Code Injection
- Reflectie
- De Serious Game Engine (SGE)

Bij elk onderwerp geven we een uitleg van het onderwerp, en geven we aan of het in het ontwerp en de implementatie is gebruikt, en waarom wel of niet.

4.1. Het Adaptive Object Model

4.1.1. Algemene beschrijving

Het Adaptive Object Model (AOM) is een model dat de laatste jaren steeds meer gebruikt wordt. Vaak in "business systems" die producten moeten managen en makkelijk uitgebreid moeten kunnen worden met nieuwe producten. Er worden verschillende namen uitwisselbaar voor gebruikt, waaronder:

- Dynamic Object Model (DOM)
- Reflective Architecture
- Meta Architecture

Een reflective architecture is een architectuur die tijdens runtime kan worden aangepast aan nieuwe wensen van de gebruiker. Volgens [3] is een AOM/DOM hierom bepaald type reflective architecture. Dit is waar wij ook van uitgaan.

Het AOM kan gedefinieerd worden als een systeem dat klassen, attributen, relaties en gedrag (behavior) representeert als metadata. Er worden dus, als je het vergelijkt met een "normaal" systeem, elementen uit de code naar buiten gehaald en opgeslagen als data. Hierbij kan gedacht worden aan bijvoorbeeld namen van klassen en methodes. Dit leidt ertoe dat AOMs twee kenmerken bezitten die over het algemeen als nuttig worden gezien en waar volgens sommige bronnen de laatste jaren steeds meer vraag naar is, namelijk flexibiliteit en de mogelijkheid om het model tijdens runtime aan te passen. Dit vereist wat nadere uitleg.

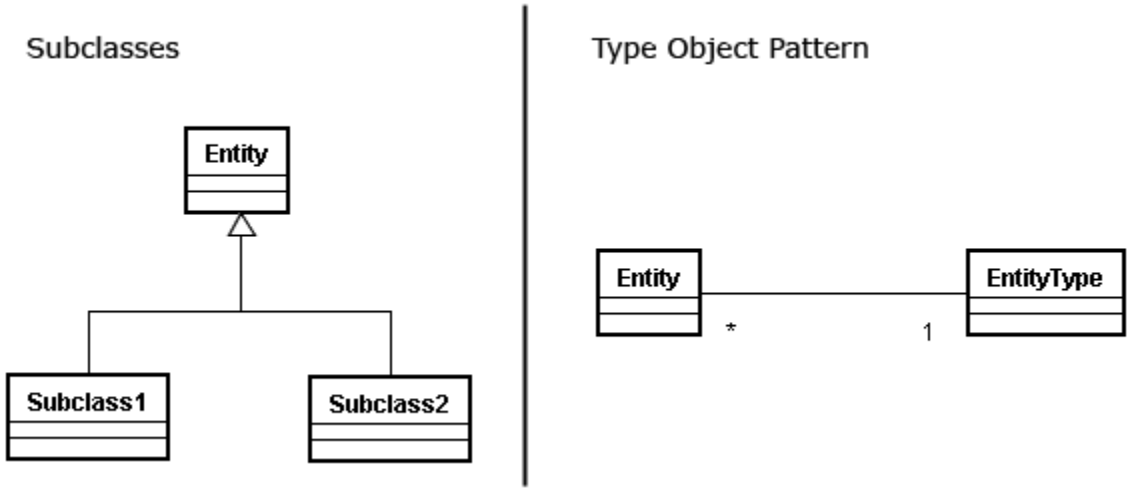
Het systeem is flexibel omdat het de mogelijkheid geeft om het systeem aan te passen "zonder te programmeren". Want zoals eerder gezegd wordt het hele systeem van het bedrijf, dus de objecten, regels en het gedrag, vastgelegd als data. Deze data wordt vervolgens door het AOM gebruikt om invulling te geven aan gegeneraliseerde klassen en zo het systeem op te bouwen. Dit wordt in het algemeen de interpretatie van een systeem genoemd [3][4]. Het enige wat dus veranderd hoeft te worden om het systeem aan te passen is data, en niet de code van het programma. En dit is ook de reden dat deze aanpassingen tijdens runtime kunnen gebeuren, want als de code niet wordt veranderd hoeft er ook niets opnieuw gecompileerd te worden.

4.1.2. Technische werking

In deze sectie worden de technische aspecten van het Adaptive Object Model besproken, dus aan welke eisen de klassen en opbouw van een systeem moeten voldoen om het een AOM te noemen. Het AOM is opgebouwd uit verscheidene kleinere patterns.

Type Object Pattern

Het Type Object Pattern kijkt of het mogelijk is om verschillende objecten (vaak Entities genoemd) samen te vatten in types. In dit deel van het AOM (het Type Square gedeelte, zie onder) worden hier de meer "tastbare" objecten mee bedoeld, in tegenstelling tot bijvoorbeeld actie-objecten, maar het kan eigenlijk overal op gebruikt worden. Waar mogelijk kan in plaats van subklassen van een Entity een EntityType-klasse worden gebruikt en zo een overvloed aan subklassen worden vermeden. Hierbij kan bijvoorbeeld gelet worden op overeenkomsten in de attributen van de subklassen. Daarnaast zorgt dit pattern ervoor dat er op een dynamische manier, zonder te programmeren, nieuwe Entities voor het systeem kunnen worden gemaakt (om code voor nieuwe subklassen te schrijven moet het model aangepast worden, om een instantie van een EntityType-klasse te maken niet).



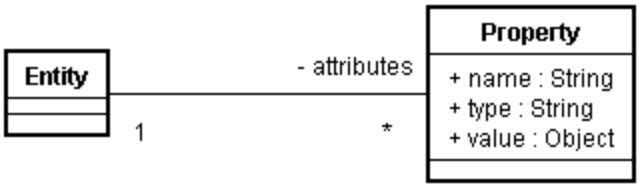
Figuur 2. *Verduidelijking van het Type Object Pattern.*

Als we als voorbeeld een systeem nemen waarin producten moeten worden beheerd vindt de volgende verandering plaats:
In plaats van
    een Product-klasse met verschillende subclasses (voor ieder product)
komt er
    een Product-klasse die een ProductType-klasse heeft.

Property Pattern

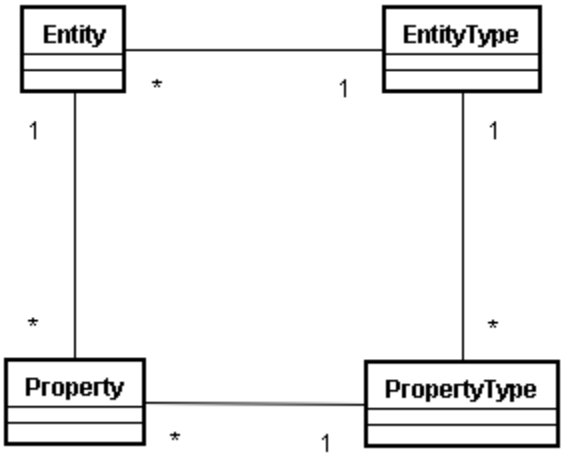
Het Type Object pattern heeft toch nog een duidelijk nadeel in vergelijking met subklassen. Subklassen van een klasse kunnen namelijk t.o.v. elkaar andere attributen hebben. Dit is een mogelijkheid die wegvalt als je de subklassen vervangt door instanties van één klasse. Om toch deze functionaliteit te behouden zonder dat het ten koste gaat van de dynamische mogelijkheden van het Type Object Pattern wordt het Property Pattern gebruikt. Dit gebeurt

door de attributen niet vast in de klasse te coderen, maar in plaats daarvan elke Entity een lijst van Properties bij te laten houden (die de attributen moeten voorstellen). Deze Properties hebben meestal naam, type en value attributen en kunnen dynamisch toegevoegd en verwijderd worden.



Figuur 3. *Het Property Pattern.*

Vervolgens wordt in veel AOMs nog een keer het Type Object Pattern gebruikt op Property. Deze wordt dus net als bij Entity opgesplitst in de klassen Property en PropertyType. De model wat daarmee wordt verkregen wordt een Type Square genoemd.



Figuur 4. *Een Type Square.*

Dit is een stap die de consistentie van het systeem ten goede komt. EntityTypes kunnen op deze manier de PropertyTypes specifieren die hun Entity kan hebben. De hierboven beschreven TypeSquare, gevormd met behulp van de twee patterns, vormt de basis van het Adaptive Object Model.

4.1.3. Conclusie

Wij vonden dat dit model er aantrekkelijk uit zag om te gebruiken voor deze opdracht. Aan ons is namelijk gevraagd om een 'modulair' systeem te ontwikkelen waarbij het eventueel mogelijk is om tijdens runtime aanpassingen te maken. Ook staat in ons PvA dat de gebruiker van ons systeem niet noodzakelijk hoeft te kunnen programmeren. En dit zijn precies de kenmerken van een AOM. Om deze reden hebben wij dit model gebruikt als basis voor ons eigen model, en zijn hieruit verder gaan uitbreiden.

4.2. Event Driven Process Chains

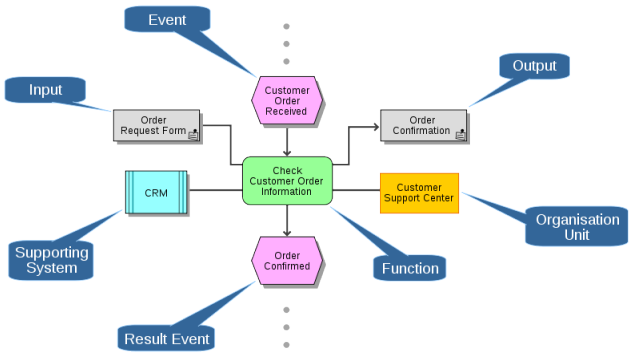
4.2.1. Omschrijving

Voor het visualiseren van objecten en relaties, en het aanpassen daarvan in een grafische interface, is het wenselijk dat de visualisatie aansluit op bestaande methoden om een dergelijk systeem te ontwerpen/visualiseren. We hebben een aantal gangbare methodes onderzocht, waarvan Event Driven Process Chains en UML-Activity Diagrams het meest aansluiten op onze behoeften.

Event Driven Process Chains worden gebruikt om bedrijfsprocessen te modelleren. Deze modelleringstechniek wordt al gebruikt door de procesanalist en sluit daardoor goed aan op de kennis van de gebruiker. Daarnaast vormt het een model voor zowel het probleem als voor een deel van de door de gebruiker te ontwerpen oplossing. Het heeft echter een hoge mate van abstractie en weinig aansluiting met object-oriented ontwerpen.

In Event Driven Process Chains worden de volgende elementen gebruikt die relevant zijn voor onze toepassing [2]:

- Event. Events geven gebeurtenissen aan waarna een functie (of bedrijfs-proces) wordt uitgevoerd. Een Event wordt weergegeven door een hexagon.
- Functies. Een functie beschrijft een transitie tussen toestanden, en eventuele invoer en uitvoer. Een functie kan een abstractie zijn van een ander EDPC. Een functie kan meerdere inkomende en/of uitgaande toestanden hebben, wat expliciet gemodelleerd kan worden door middel van logische connectoren. Een functie wordt weergegeven door een afgeronde rechthoek.
- Objecten. Een object kan een abstract object zoals informatie, of een fysiek object, zoals bijvoorbeeld de *resource* zand, zijn. Een object wordt weergegeven door een grijze rechthoek.



Figuur 5. *Een voorbeeld van een EDPC model, met beschrijving van de soorten elementen.*

Aangezien EDPC een hoge mate van abstractie heeft, is het nuttig het EDPC te combineren met bepaalde UML diagrammen, zoals het class-diagram. In [1] wordt beschreven hoe die combinatie gemaakt kan worden. Hierdoor ontstaat een diagram dat beter aansluit op een object-oriented implementatie.

4.2.2. Conclusie

Wij hebben uiteindelijk enkele, maar niet alle, elementen uit Event Driven Process Chains gebruikt. Deze hebben wij gecombineerd met UML om met de GUI de gebruiker in staat te

stellen objecten op een makkelijke manier te bewerken, en daarnaast een soort van workflow relaties aan te geven.

4.3. Code Injection

4.3.1. Omschrijving

Code injection is nuttig om functionaliteit toe te voegen of te veranderen aan klassen. Het is mogelijk nieuwe klassen te maken, of een bepaalde klasse tijdens runtime aan te passen, zolang die klasse op dat moment niet al geladen is. Code injection zou het gebruikt kunnen worden om properties aan een object toe te voegen. In combinatie met een property pattern is het echter niet zinvol dit te gebruiken voor data-klassen, want dit pattern zorgt er al voor dat er dynamisch properties aan objecten kunnen worden toegevoegd. Het zou echter wel nuttig kunnen blijken te zijn voor bepaalde controllers.

Java heeft beperkte standaard ondersteuning voor het aanmaken en injecteren van bytecode, maar er zijn libraries beschikbaar waarmee deze functionaliteit toch mogelijk wordt. Een voorbeeld van een dergelijke library is Javassist.

4.3.2. Conclusie

Wij hebben bij dit project geen gebruik gemaakt van code injection. Dit was niet nodig, omdat we voor het maken en bewerken van klassen het Adaptive Object Model een betere, en elegantere, keuze vonden. Wij hebben er dus voor gekozen om tijdens het uitvoeren van ons systeem alleen data te veranderen, en geen code.

4.4. Java-Reflectie

4.4.1. Omschrijving

We hebben de mogelijkheden, en API, van Java-reflectie onderzocht. Reflectie zou de basis kunnen vormen van een dynamisch object model, en zou gebruikt kunnen worden om bestaande klassen naar een nieuw systeem over te zetten, en om attributes te gebruiken.

Met behulp van reflectie kan, onder andere, tijdens runtime het type van een object, en de methods, fields en attributes van klassen opgevraagd worden. Het is ook mogelijk om dynamisch methodes aan te roepen, of fields te wijzigen.

4.4.2. Conclusie

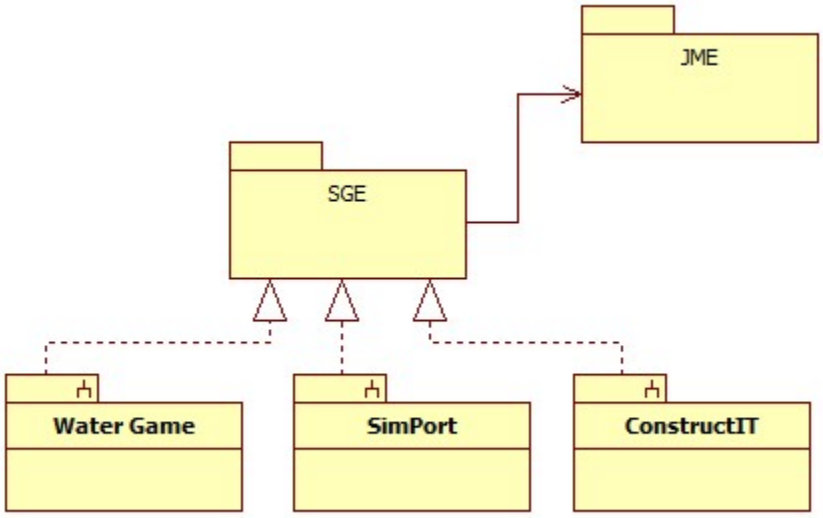
Reflectie is niet van groot belang als het bovenstaande property pattern wordt gebruikt, omdat met dat pattern de metadata die door reflectie wordt gebruikt, ook expliciet in het object-model aanwezig is. Met behulp van reflectie kan echter wel een bestaande klasse worden geanalyseerd, waardoor er vervolgens een dynamische klasse op basis van het property pattern van kan worden gemaakt. Om deze reden hebben wij niet uitgebreid gebruik gemaakt van de mogelijkheden van reflectie, maar is het enkele malen toegepast om bijvoorbeeld op klassen te checken, of om in de GUI het property-panel dynamisch op te bouwen.

4.5. Serious Game Engine (SGE)

Tygron maakt voor haar spellen gebruik van een intern ontwikkelde game engine, genaamd Serious Game Engine, ofwel SGE. We hebben de architectuur en werking hiervan onderzocht aangezien het deel uitmaakt van het huidige systeem, en omdat ons systeem hier eventueel mee moest kunnen samenwerken.

4.5.1. Globale architectuur

Het doel van de Serious Game Engine is om te zorgen dat de programmeurs van een spel zich uitsluitend op spel-specifieke functionaliteit hoeven te richten. Globaal werkt de SGE als volgt. SGE bestaat uit een Java library, met daarin een aantal klassen, die vanuit een spel gebruikt kunnen worden. Hieronder vallen klassen voor rendering van 3D objecten of het lezen van user-input regelen. Ook zijn er klassen voor bijvoorbeeld netwerk communicatie, *serialization*, en het renderen en gebruik van een 2D interface binnen een 3D renderer viewport.



Figuur 5. *Globaal overzicht van de SGE engine, en spellen die daarvan gebruik maken.*

4.5.2. JME, FengGUI, openGL

De SGE gebruikt de jMonkey Engine (JME). JME is een game-engine voor Java, gebaseerd op openGL. Met behulp van JME kunnen bijvoorbeeld 3D objecten worden gerenderd en kan input van de gebruiker worden gelezen. Hierbij wordt gebruik gemaakt van openGL, en dus niet via de standaard Java Foundation Classes (JFC) (waaronder AWT en Swing) die normaal worden gebruikt voor het tekenen van graphics, en user-interactie in Java.

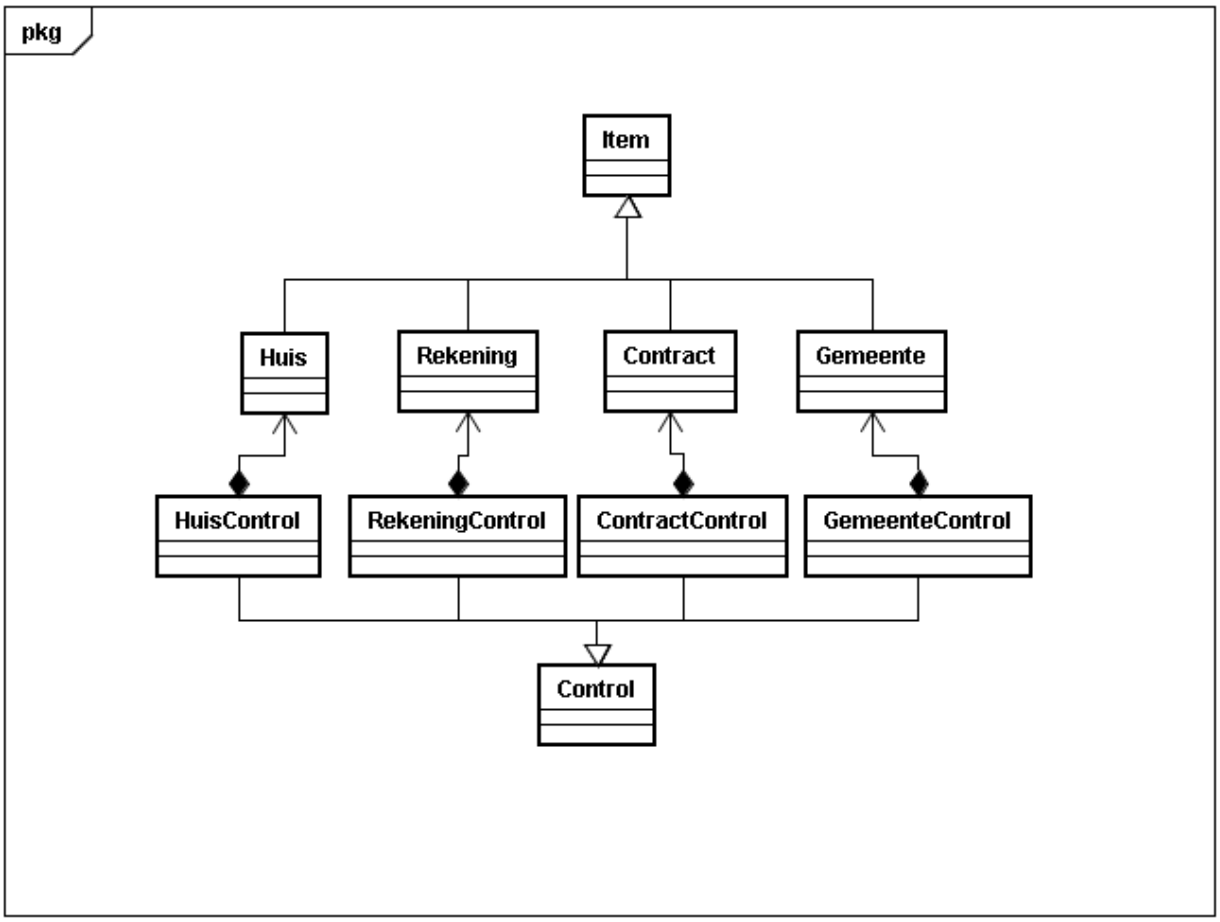
Zoals de meeste game engines werkt JME met behulp van een "game-loop": tijdens het uitvoeren van een spel wordt het beeld per "frame" geupdated. Dit gebeurt zo'n 60 maal per seconde. De game-loop herhaalt voor het tekenen van ieder frame een aantal stappen: eerst wordt een update methode aangeroepen en vervolgens een render methode. In de render methode wordt op basis van de tekenbare spel-objecten een beeld getekend, door polygon-informatie via openGL naar de GPU (Graphics Processing Unit) te sturen. In de update methode worden de spel-objecten geupdated, en wordt de toestand van het

toetsenbord en de muis uitgelezen. Binnen een spel werken events dus niet als het "*event*" system zoals dat bij Java GUIs bekend is, waarbij klikken met de muis en gebruik van toetsenbord etcetera resulteert in het aanroepen van bepaalde *ActionListeners*.

Voor het tekenen van GUIs binnen een spel (zoals knoppen in een menu en dergelijke) wordt binnen SGE FengGUI gebruikt. Deze FengGUI library biedt functionaliteit die vergelijkbaar is met de functionaliteit van AWT of Swing, zoals het plaatsen, tekenen en gebruik van *Buttons*, *TextBoxes*, etcetera. FengGUI implementeert een "*event*" systeem op basis van het *pollen* van de muis en toetsenbord van OpenGL. Dit systeem werkt vergelijkbaar met het "*event*" systeem van AWT/Swing, maar gebruikt hiervoor andere *interfaces* (in de Java-OOP zin van het woord).

4.5.3. Items

Een spel bevat verscheidene spel-objecten. Een spel-object kan van alles zijn, bijvoorbeeld een fysiek (al is het natuurlijk virtueel) object zoals een huis, een aannemer of een contract, of een meer abstract object zoals een bankrekening of een deelgemeente. Binnen SGE *extendt* ieder spel-object de Item klasse. De Item klasse bevat twee velden, namelijk het ID en het versienummer. Voor ieder Item object is het ID uniek. Het ID wordt elders in het spel gebruikt als referentie naar het item. Het versienummer wordt verhoogd elke keer dat een attribuut van het item gewijzigd wordt. Zo kan worden bepaald welke items zijn veranderd, wat van belang is voor de netwerk-communicatie (zie 4.5.5).
De subklassen van Item zijn in principe uitsluitend dataklassen, en bevatten dus geen functionaliteit (in de vorm van methoden, behalve getters en setters). Er zijn echter een paar uitzonderingen waarbij een subklasse van Item een methode getX heeft terwijl het de desbetreffende waarde niet bevat, en het die waarde via een omweg (bijvoorbeeld via een ander *veld*) opvraagt en teruggeeft.
We noemen deze spel-objecten, instanties van de subklassen van Item, in het vervolg items.



Figuur 5. *Een voorbeeld van een aantal spel-objecten, in een op SGE gebaseerd spel.*

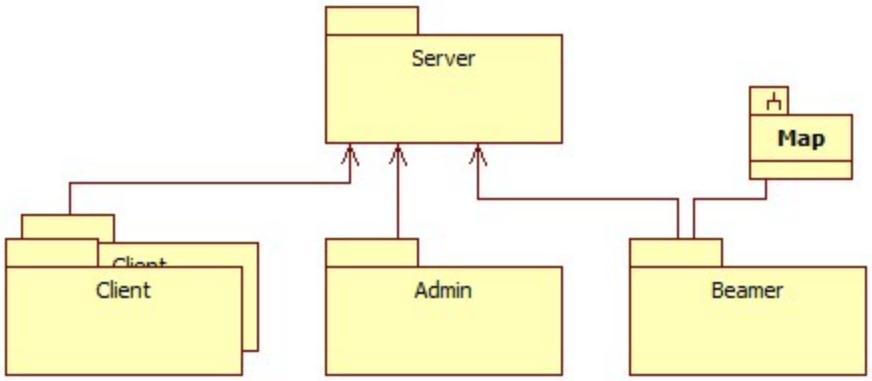
4.5.4. Controllers

Voor iedere subklasse 'X' van Item bestaat een subklasse XControl welke extendt van Control<X>. Deze worden de controllers genoemd. Alle items worden beheerd door hun bijbehorende controllers. De controllers bewerken de gegevens van de items en zijn verantwoordelijk voor het ophogen van het versienummer van de items.

4.5.5. Client/Server implementation

SGE is ontwikkeld voor *multiplayer* spellen die door meerdere gebruikers, vanaf aparte computers, gespeeld kunnen worden. Het heeft hiervoor een client-server architectuur, waarbij meerdere clients verbonden zijn met de gameserver. Clients kunnen uitsluitend met de server verbinden, dus een server kan geen connectie met een client starten, en een client kan niet met een andere client verbinden. De communicatie tussen client en server maakt gebruik van Remote Method Invocation (RMI). Deze communicatie gaat *uitsluitend* in de vorm van "events". De handelingen van een speler genereren een "client-event" dat naar de server wordt gestuurd, waar het in de Switchboard klasse wordt verwerkt tot een "server-event". Server-events worden gebruikt door de server om spelfunctionaliteit uit te voeren. Vervolgens worden door middel van een "update-event" gegevens teruggestuurd naar de clients. Er zijn verschillende soorten clients voor een spel. De meeste clients zijn de

gewone game-clients van normale spelers. Er is echter ook een Administrator die het spel beheert. Hij gebruikt een aparte Admin client. Er bestaat ook een Beamer client, die alleen het spel weer kan geven en waarmee verder geen interactie van de gebruiker met het spel mogelijk is. De server beslist op basis van het type client welke update-events van belang zijn voor de client.



Figuur 7. *Multiplayer architectuur*

4.5.6. Events

Er zijn verschillende interpretaties mogelijk van het woord "event", afhankelijk van de context. Binnen Java GUIs zijn events bekend als datgene waarop implementaties van Listeners kunnen reageren, zoals bijvoorbeeld een MouseDown, WindowResized, of ButtonPressed event. Binnen FengGUI, zoals dat in SGE gebruikt wordt, bestaan ook dit soort events die op acties van de gebruiker binnen de GUI reageren. Daarnaast worden door het spel zelf ook andere soorten events gebruikt. Zo kan er bijvoorbeeld een event worden afgevuurd als het bouwen van een huis is voltooid, een aanvraag is goedgekeurd, een gemeenteraadsverkiezing wordt gehouden, etcetera. Er zijn echter ook events binnen SGE. Zo wordt er binnen de server een event afgevuurd als een wijziging heeft plaatsgevonden waar een client weet van moet hebben, waardoor de game-clients worden geupdated (zie 4.5.5). Het is belangrijk om in het vervolg van deze tekst deze mogelijke interpretaties van "event" in acht te nemen! Deze paragraaf gaat verder met het gebruik van events van de SGE. Dit is ook het type events wat relevant voor ons is, omdat ons systeem eventueel zou moeten aansluiten op de SGE, niet op FengGUI.

Er zijn vanuit het perspectief van de engine gezien twee soorten events; clientside events en serverside events. De klasse die aan de clientkant centraal staat bij de events is de EventManager. De EventManager, een static klasse, beheert de listeners en vuurt events af. Een klasse kan alleen een event afvuren door in de EventManager fireEvent() aan te roepen en het event als argument mee te geven. In het event staan in ieder geval het type event en de source (het object dat fireEvent aanroept), en optioneel een argument. De serverkant heeft de Brain-klasse met ongeveer dezelfde functionaliteit m.b.t. events.

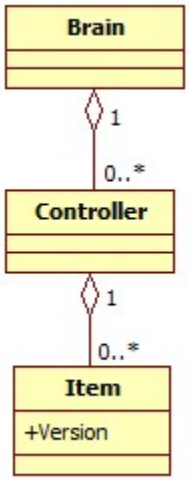
Voor het afhandelen van events worden er in het begin het van implementeren van een spel twee klassen gemaakt. LogicManager zorgt voor het afhandelen, dus ontvangen en doorsturen, van events aan de clientkant en SwitchBoard vertaalt client-events naar server-events, welke verder door het Brain kunnen worden afgehandeld. Beide klassen worden in de EventManager toegevoegd als listener voor alle events. Deze klassen worden "generic

listeners" genoemd, omdat ze alle events ontvangen. Een klasse kan ook als "specific listener" worden toegevoegd door het type event op te geven als hij zich registreert in de EventManager. Als een event wordt afgevuurd worden eerst alle generic listeners en daarna de specific listeners afgegaan.

De events zelf worden bijgehouden in enums. Elke enum staat voor een bepaald type event en implementen allemaal de EventTypeEnum klasse uit de SGE. Bij deze eventtypes/enums kan bijvoorbeeld gedacht worden aan een ClientEventType-enum of ViewEventType-enum. Om een event te maken moet deze handmatig aan een enum worden toegevoegd. Sommige events (bijvoorbeeld de view events) zijn al gedefinieerd in de SGE, omdat deze hier moeten worden afgehandeld.

4.5.7. Brain

De Brain-klasse is de belangrijkste controlklasse van het spel. Alle controllers van items zijn gelinkt aan het brain, en de enige manier om de waardes van een item te updaten is via het brain. Brain is een klasse op de server die zowel lokaal (in single player games) als op een aparte computer kan draaien. Dit heeft tot gevolg dat alle manipulatie van items op de server gebeurt. Het is dus niet mogelijk om een item op een client te updaten en vervolgens naar de server te sturen. Voor verdere informatie over de client/server implementatie, zie paragraaf 4.5.5.



Figuur 6. *Samenwerking van het Brain, de controllers en items.*

4.5.8. Annotations

Binnen Java kan code met behulp van zogeheten Annotations worden aangevuld met meta-data. Annotations zijn een soort van "comments" die voor een parser en compiler begrijpbaar zijn. Annotations kunnen worden toegevoegd aan onder andere klassen, velden en methodes. Via de *reflection* API van Java kunnen deze Annotations tijdens *runtime* worden opgevraagd. Er zijn standaard Annotations (zoals @ Override, of @ Author) maar er kunnen ook nieuwe typen Annotations door een programmeur worden gedefinieerd. Binnen SGE wordt gebruik gemaakt van deze Annotations. Items hebben voor sommige velden de volgende annotations: XMLValue om aan te geven dat het veld kan worden opgeslagen en HTML, Editable voor bepaalde tools. Controllers hebben ook (class) annotations: UpdateEvent definieert wat voor Event de Controller genereert. ItemClasses

bepaalt wat voor items de controller gebruikt. Subscriptions bepaalt of het item relevant is voor clients dan wel administrators, en dus voor welke gebruikers-groep het item zichtbaar zal zijn.

4.5.9. Voorbeeld: Watergame

Een voorbeeld van een spel dat met SGE gemaakt is, en waarvan de spel-objecten door ons te ontwerpen systeem beschreven zouden moeten kunnen worden, is het Watergame. Ter illustratie is dit de volledige lijst van spel-objecten in Watergame:

BeamSetting, Building, CivilBuilding, CivilType, ClientWord, Cost, Economy, FinanceSetting, FlightPoint, Function, FunctionalTerrain, Indicator, IndicatorGameLog, Measure, MeasureSet, PublicBuilding, PublicType, ServerWord, Strategy, Subsidy, Terrain, Type, WGActor, WGBlock, WGEffect, WGFont, WGMessage, WGModel, WaterSetting, WaterTerrain.

4.5.10. Conclusie

Aanvankelijk hadden we de hoop het bestaande systeem te kunnen converteren naar ons nieuwe systeem. Dit bleek echter na ons onderzoek niet realistisch, aangezien het niet voldoende gestandaardiseerd was. We hebben dan ook een aantal compromissen moeten sluiten met betrekking tot de integratie in huidige spellen. (zie hiervoor eerst het hoofdstuk Ontwerp) Zo konden we niet de bestaande Item klassen converteren naar ons model. In plaats daarvan hebben we ervoor gekozen om de klassen uit ons meta-model gebruik te kunnen laten maken van bestaande Item klassen.

Het is ook niet realistisch om gebleken om directe integratie in een spel mogelijk te maken zonder minimale aanpassingen aan een dergelijk spel. Een spel moet nu ons runtime package aanroepen om het model te laden, dit kan in een enkele regel code. Ook kunnen de bestaande spel-events niet direct worden gebruikt. De bestaande spel-events zouden echter gemakkelijk kunnen worden aangepast om gebruik van ons events systeem te maken. Ook zonder een dergelijke aanpassing kan ons systeem echter wel al parallel werken aan het bestaande systeem.

Er zijn ook vormen van integratie waarvoor we hebben gekozen deze achterwege te laten, omdat ze niet essentieel waren voor een "proof of concept", die gemakkelijk te implementeren zouden zijn. Bijvoorbeeld het opslaan van een spel in uitvoering, op basis van ons systeem. Dit kan gemakkelijk worden geimplementeerd, maar is niet van belang om de werking van het systeem te bewijzen.

4.6. Reflectie

De functionele eisen werden pas in week 7-8 goed uitgewerkt, maar dit bleek geen problemen op te leveren voor het onderzoek. Zelf hadden we al eerder een goed beeld van wat er in het systeem moest komen, dus wat wij in de eerste drie weken van het project hebben onderzocht bleek voldoende en er hoefde hierna dus geen extra onderzoek verricht te worden.
Wel moest er extra onderzoek verricht worden naar de werking van de Serious Game Engine. Deze was in het onderzoeksrapport te mager beschreven, en dit hebben wij uiteindelijk in het RAD uitvoeriger gedaan.

5. Ontwerp

In dit hoofdstuk wordt het ontwerp van ons systeem beschreven. Ons systeem hoefde in eerste instantie niet samen te werken met een bestaand systeem. Een proof-of-concept van het idee achter het model is voldoende. Daarom is besloten het eerst te ontwerpen als "stand-alone" systeem, wat later aangepast zou kunnen worden om samen te werken met bijvoorbeeld SGE. In paragraaf 5.1 wordt de indeling van ons systeem in model, runtime en view uitgelegd. Dit is hoe het systeem in packages is verdeeld. Vervolgens wordt verdergegaan met een andere indeling, die meer de samenwerking tussen de klassen beschrijft. Deze indeling bestaat uit de volgende stukken:

- Basis van het model (AOM)
- Importeren en managen van metatypes
- Acties
- Events

Bij elk stuk van het systeem wordt hier een klassendiagram en een uitleg van dat gedeelte gegeven. Tot slot worden, net als in het Design Document, alle klassen in detail besproken.

5.1. Packages

De belangrijkste functie van dit systeem is dat het de gebruiker in staat stelt een model te maken waarin entiteiten en verschillende soorten relaties hiertussen kunnen worden gemaakt. Een andere eis is dat dit model uitgevoerd moet kunnen worden. Dit is nodig om het gemaakte model te kunnen gebruiken in spellen en scenario's. Zonder een uitvoerbaar gedeelte kan er ook nooit een proof-of-concept in de vorm van een scenario gegeven worden. Daarnaast is er ook een GUI om het systeem heen gemaakt. De indeling van het systeem is hierdoor als volgt:

5.1.1. View

Het *view* package bevat alle klassen die uitsluitend met de GUI te maken hebben. Met deze klassen kan een model van een spel gemaakt worden.

5.1.2. Model

Het *model* package bevat klassen die door zowel de *view* en de *runtime* packages gebruikt worden. Deze klassen vormen de basis van ons systeem; het meta-model waarmee een klant-model geïmplementeerd kan worden. Dit model kan geconstrueerd worden met de klassen uit het view-package (ze kunnen ook handmatig in code gebruikt worden, maar dit raden wij af), en uitgevoerd worden met behulp van de klassen uit het runtime-package.

5.1.3. Runtime

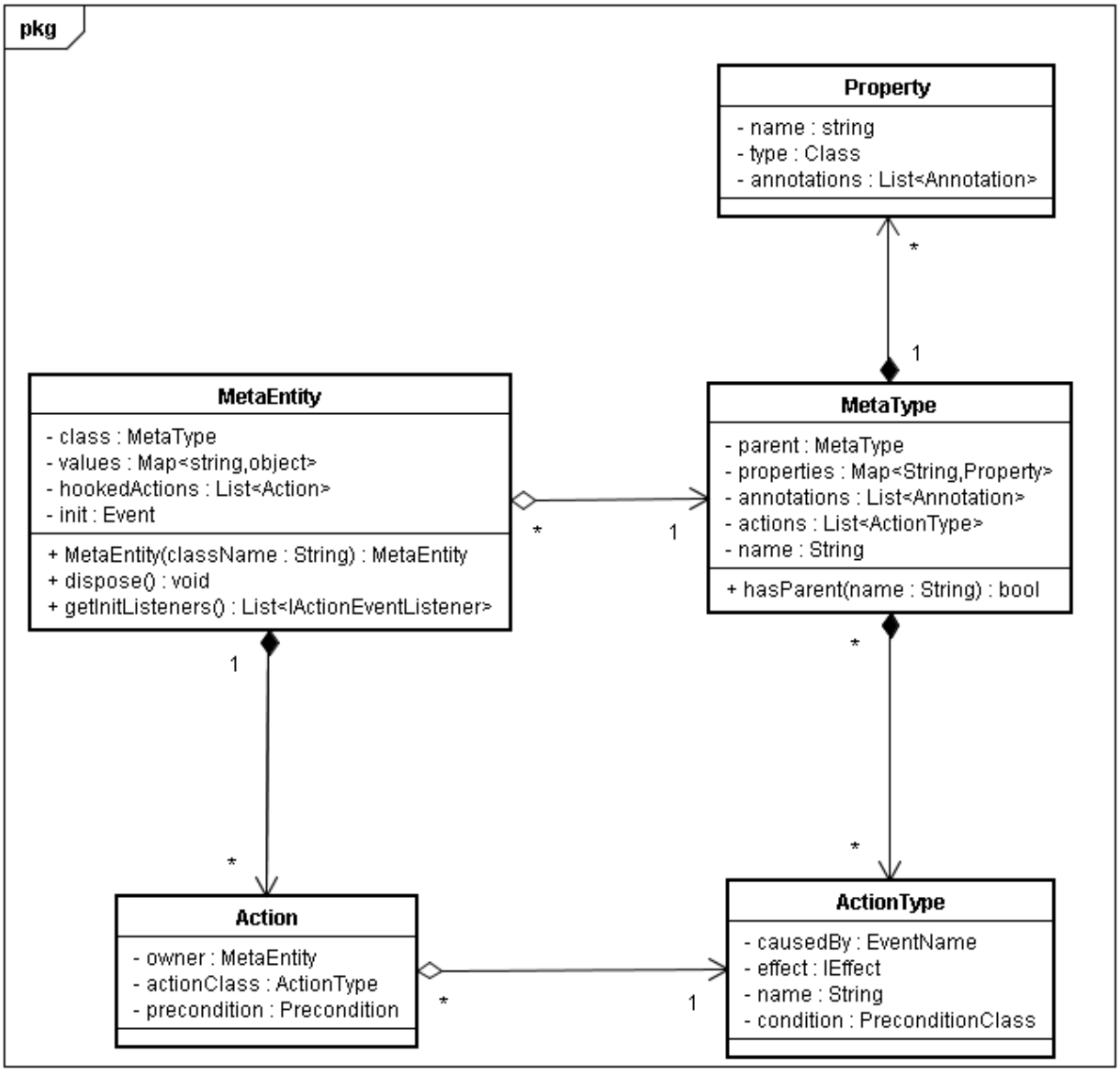
Het *runtime* package bevat alle klassen die uitsluitend met de runtime library te maken hebben, en dus niet door het *view* package worden gebruikt. Deze klassen worden tijdens het spelen van een spel gebruikt en werken samen met de Metaklassen die in het model zijn gedefiniëerd. Instanties van MetaEntity en Action dienen als de instanties van de "templates" die door instanties van MetaType en ActionType in het model worden

gedefinieerd.

5.2. Functionele indeling

In deze paragraaf stappen we over op een andere indeling, die meer inzicht biedt in hoe de verschillende klassen van het systeem qua functionaliteit samenwerken. Hier wordt dus ook de connectie tussen de model- en runtimeklassen duidelijker. Ook is het aan de hand van deze indeling makkelijker te beschrijven hoe het proces in de ontwerpfase is verlopen, omdat we deze stukjes redelijk los van elkaar hebben ontworpen.

5.2.1. AOM-basis van het model

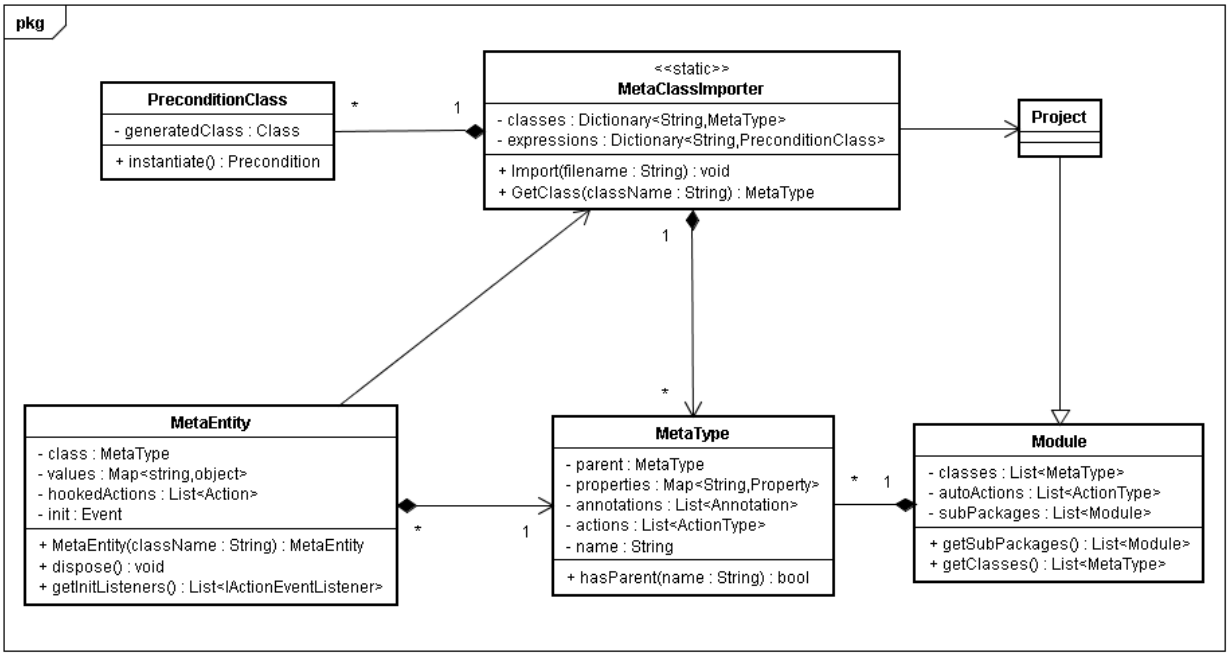


Figuur 8. *De basis van het model.*

Dit is het model waar we mee zijn begonnen, en de rest van het systeem omheen hebben ontworpen. Bij het ontwerp hebben we goed gekeken naar het Adaptive Object Model (paragraaf 4.1). We hebben de structuur hiervan op twee plaatsen toegepast; het lag voor de hand om de relaties tussen properties, types en entities zo in te delen, maar het bleek ook goed mogelijk om op dezelfde manier met acties om te gaan. Hier is voor een ontwikkeltool veel makkelijker mee om te gaan dan wanneer de acties in de metatypes worden gehouden.

Als resultaat is hier het bovenstaande model uitgekomen, waarmee de gebruiker klassen (de MetaTypes) kan maken en hier dynamisch zowel properties als actietypes aan kan toevoegen. Deze kunnen vervolgens tijdens de uitvoer gebruikt worden door de MetaEntity en Action klassen. De MetaType, ActionType en Property klassen zitten om deze reden in het model package, en de MetaEntity en Action klassen in het runtime package.

5.2.2. Importeren en managen van Metaklassen



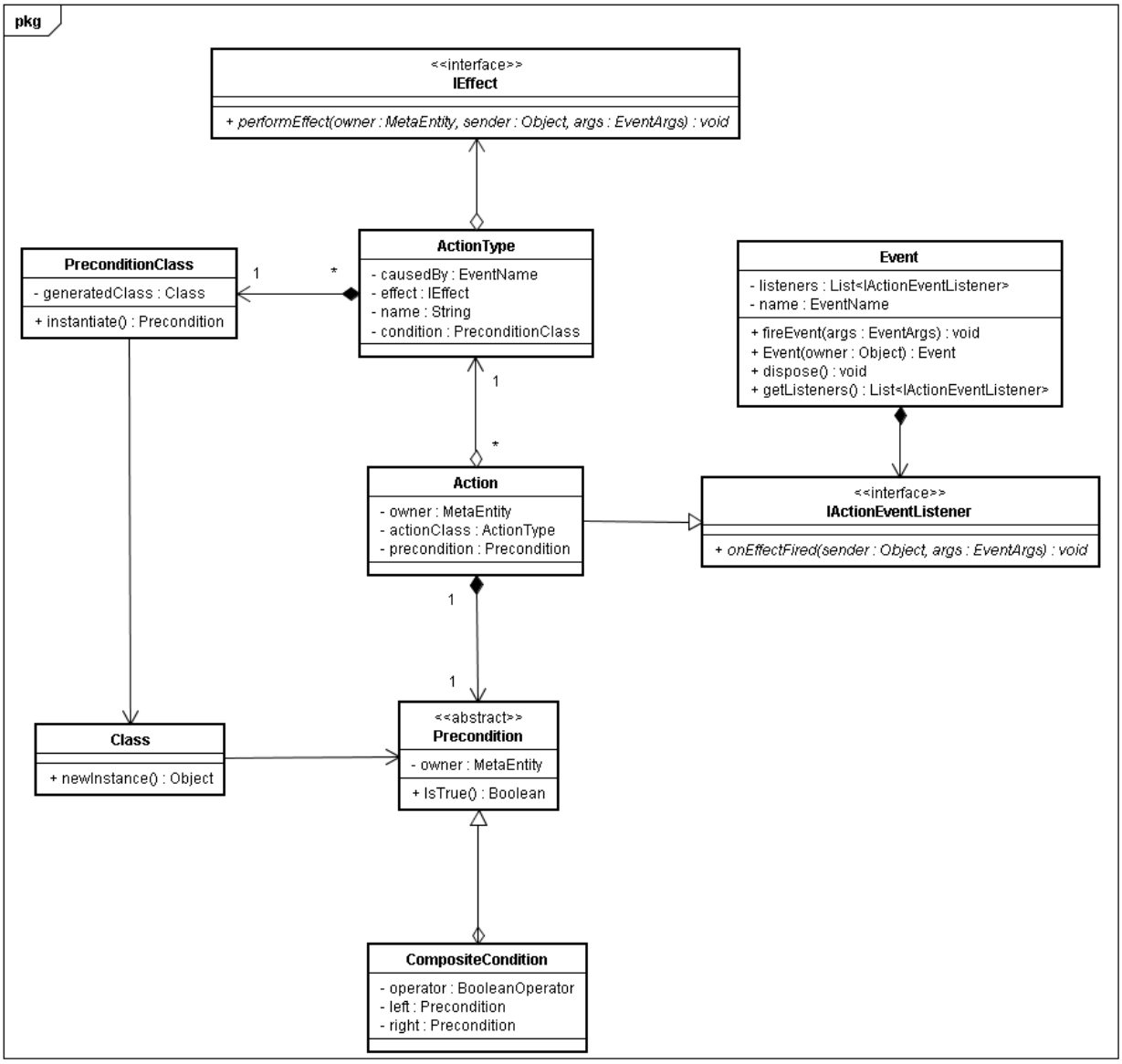
Figuur 9. *Het importeren van Metaklassen.*

Nadat een model is gemaakt, moet het ook bij het uitvoeren van een spel gebruikt kunnen worden. Het leek ons handig deze koppeling te maken door de GUI een .xml bestand te laten exporteren met daarin de gemaakte klassen en hun onderlinge relaties en structuur. Deze keuze is gemaakt omdat er binnen het bedrijf al veel wordt gewerkt met .xml bestanden, en door de structuur kunnen deze makkelijk weer ingelezen worden en omgezet worden naar klassen. Dit is het model zoals het er in de ontwerpfase uitzag om metaklassen, die met ons systeem zijn gemaakt, te gebruiken in een spel.

Nadat een project vanuit de GUI is geexporteerd naar een .xml bestand, kan deze door de MetaClassImporter ingelezen worden. De MetaClassImporter doorloopt de boom van modules in het project en bewaart de metaklassen in een HashMap, zodat ze kunnen worden opgevraagd door MetaEntities, wanneer deze worden geïnstantieerd. In dit model is de MetaClassImporter een runtime-klasse, omdat deze alleen tijdens het uitvoeren van een

spel wordt gebruikt. Project en Module zijn model-klassen, omdat deze door de gebruikers van het systeem worden gemaakt.

5.2.3. Acties

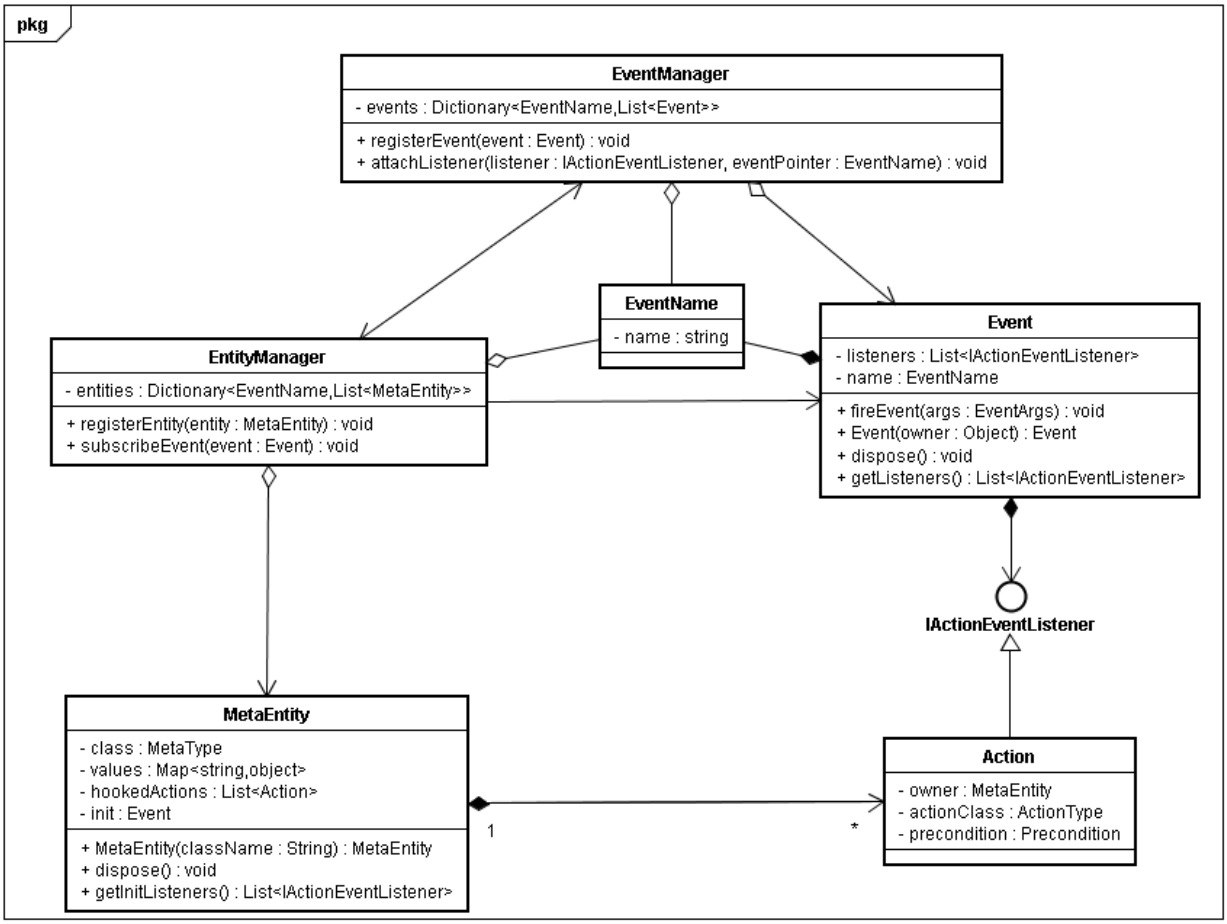


Figuur 10. *Werking van acties.*

Vanuit de GUI worden ActionTypes gedefinieerd. De instanties van deze klasse werken als AOM klassen voor de Actions. Actions luisteren naar een Event (gedefinieerd door het ActionType). Als het Event wordt afgevuurd, controleert de Action of zijn Precondition op dat moment waar is, en indien dat het geval is, roept het zijn Effect aan. Een effect is een implementatie van de IEffect interface. De precondition is een instantie van een klasse die door de GUI wordt gecompileerd, op basis van de daarin opgegeven preconditie-expressie. Om deze klasse te instantiëren wordt de getInstance methode van het Class object dat die klasse beschrijft aangeroepen, aangezien er geen code bestaat waar vanuit die instantiatie

met behulp van het 'new' keyword zou kunnen worden gedaan. Met behulp van CompositeCondition kan een boom van precondities worden opgebouwd.

5.2.4. Events



Figuur 11. *Werking van events.*

Events kunnen handmatig (door een programmeur) of vanuit de GUI (door een developer) worden gedefinieerd en afgevuurd. Events worden geïdentificeerd met behulp van de EventName (een String). Implementaties van de IActionEventListener kunnen zich voor een bepaald soort event aanmelden door zich aan de lijst van listeners van een bepaald event toe te voegen, of door zich via EventManager aan te melden op alle events met een bepaalde naam. De events registreren zich bij de eventmanager om dit mogelijk te maken. Met behulp van de EntityManager wordt het bovendien mogelijk om bij het aanmaken van een nieuw event, alle van dit event afhankelijke acties te vinden. Hierdoor is het mogelijk vanuit de GUI acties aan events te koppelen, zonder kennis over waar en wanneer die events worden gedefinieerd.

5.3. Details van de klassen

In deze paragraaf wordt elke klasse van ons systeem apart besproken. Eerst wordt er een korte omschrijving van de functionaliteit van de klasse gegeven en vervolgens worden de

attributen en methodes uitgelegd. De getters, setters, constructors en voor de hand liggende attributen en methodes worden hier niet behandeld, met uitzondering als we willen uitleggen waarom we ervoor gekozen hebben om deze te gebruiken.

5.3.1. View klassen

ExpressionCompiler

Methodes:

- compile(String expression) : PreconditionClass
  Implementeert een opgegeven expressie in een klasse en compileert deze, zodat de klasse die wordt gemaakt gebruikt kan worden door ActionType.

5.3.2. Runtime klassen

IActionEventListener

Dit is een interface die wordt gebruikt om te reageren op een Event. Deze interface wordt geimplementeert door Action.

Action

De actions zijn de objecten waarmee tijdens de uitvoer van het spel waardes aangepast kunnen worden. Actions staan geregistreerd bij de Events waar ze naar moeten "luisteren". Als deze wordt afgevuurd, en hun precondition is waar, dan voeren ze het effect van hun ActionType uit.

Attributen

- MetaEntity owner
  De beheerder van de actie. Het object waardoor de actie wordt aangeroepen. Dit kan worden doorgegeven aan de precondition.
- ActionType actionClass
  Bevat de algemene info van dit type actie.
- Precondition precondition
  De actie kan uitgevoerd worden als de preconditie true is.

Methodes

- onEventFired (van de IActionEventListener Interface)
  Voert het effect uit.

EntityManager

Een "static klasse". Houdt alle MetaEntity-instanties bij die gemaakt worden en koppelt ze aan EventNames. Als een Entity geregistreerd wordt, worden gelijk ook alle acties geïnstantieerd en als listeners aan Events gekoppeld.

Attributen

- HashMap<EventName, LinkedList<MetaEntity>> entities
  Houdt voor elke Event bij welke MetaEntities eraan gekoppeld zijn.

Methodes

- registerEntity(MetaEntity entity)
  Voegt de entity toe aan de lijsten van alle events waar deze entity op kan reageren, en maakt voor al die events een Action aan voor de entity, en voegt deze toe aan het event als listener.
- subscribeEvent(Event event)
  Als er een Event wordt gemaakt zorgt deze methode ervoor dat voor alle meta-entities die aan dat event gekoppeld horen te worden, dit gebeurt.

Event

Events spelen een belangrijke rol in het plaatsen van Actions. Elk ActionType heeft een Event waar het op reageert en door deze af te vuren kunnen dus Actions aan de GUI en aan elkaar gekoppeld worden.

Attributen
- Object owner
  De sender van het Event. Wordt meegegeven als het event wordt afgevuurd.
- List<IActionEventListener> listeners
  De listeners/actions. Moeten geïnformeerd worden als het event wordt afgevuurd.
- EventName name
  De naam van het event. Er kunnen meerdere instanties van Events zijn met dezelfde naam.

Methodes
- fireEvent(Object args)
  Meldt aan alle listeners dat het event is afgevuurd

EventManager

Een statische klasse. De EventManager houdt bij welke Events welke EventNames hebben, en heeft ook methodes om hier listeners aan te koppelen. Zo kan een Action bijvoorbeeld aan meerdere events tegelijk gekoppeld worden.

Attributen
- HashMap<EventName, List<Event>>
  Houdt bij welke events welke EventName hebben.

Methodes
- registerEvent(Event event)
  Voegt een Event/EventName koppel toe aan de hashmap. Subscribed het event ook in de EntityManager.
- attachListener(IActionEventListener listener, EventName eventPointer)
  Voegt een willekeurige implementatie van IActionEventListener als listener toe aan alle events die als EventName eventPointer hebben.
- attachListeners(MetaEntity owner, ActionType at, EventName eventPointer)
  Maakt een nieuwe Action van type 'at' met 'owner' als *owner*, en voegt deze als listener toe aan alle events die als EventName eventPointer hebben.

MetaClassImporter

Importeert alle klassen (/MetaTypes) uit de jar-bestanden van een project en zet ze in een hashmap.

Attributen
- HashMap<String, MetaType> classes
  Houdt de ingelezen klassen bij.

Methodes
- import(String filename)
  Initialisatiestap voor het importeren. Leest een jar-bestand in en zet deze om in een Module, die vervolgens wordt meegegeven aan importModule().
- importModule(Module m)
  Wordt aangeroepen door import(). Doorloopt de meegegeven Module en submodules op een recursieve manier en zet de classes in de hashmap.
- getClass(String className) : MetaType
  Doorzoekt de HashMap classes en geeft de MetaType met naam className terug.

MetaEntity

Dit zijn de "objecten" die tijdens het uitvoeren van het spel worden gebruikt. Elke MetaEntity heeft een MetaType, die de klasse voorstelt. Een instantie van MetaEntity kan worden gezien als een daadwerkelijke instantie van die MetaType-klasse, een instantie van een MetaType is uiteraard ook een instantie, maar kan beter worden gezien als een nieuwe klasse.

Attributen
- MetaType class
  Het type Entity. Wordt class genoemd omdat het voor de gebruikers (in de GUI) lijkt of ze nieuwe Java klassen aanmaken.
- Map<String, Object> values
  De waardes van de properties die in MetaType gedefiniëerd zijn.
- List<Action> HookedActions
  Dit is zodat deze listeners van de events kunnen worden verwijderd.
- Event init
  Initialisatie-event. Laat aan de rest van het programma weten dat deze MetaEntity er is, en dus gebruikt kan worden (bijvoorbeeld door andere Events).

Methodes
- MetaEntity(String className)
  Constructor

Precondition

De preconditie voor een actie. Als een event wordt afgevuurd wordt gekeken of aan deze preconditie wordt voldaan. Als dit zo is wordt het effect van de actie uitgevoerd en anders gebeurt er niks.

Attributen
- MetaEntity owner
  De owner van de actie waar dit een preconditie van is, zodat hier informatie van kan worden opgevraagd (bijvoorbeeld: als de status van de owner x is, return true).

Methodes

- isTrue() : boolean
  Geeft true als de Preconditie waar is (voert dus de eerder opgegeven en gecompileerde expressie uit).

CompositeCondition

Een subklasse van Precondition. Bevat twee Preconditions (kunnen ook weer CompositeConditions zijn) die aan elkaar gekoppeld zijn door een boolean operator.

Attributen
- BooleanOperator operator
  De boolean operator (AND, OR of NOT)
- Precondition left
- Precondition right

BooleanOperator

Dit is een enum van de operatoren AND, OR en NOT, met ingebouwde functionaliteit, zodat niet in een hogere klasse gecheckt hoeft te worden welke operator het is, maar de operator gelijk uitgevoerd kan worden.

5.3.3. Model klassen

ActionType

Het type dat tijdens runtime aan een Action wordt toegewezen (elke actie moet een type hebben). Bevat informatie over wat dit type actie moet doen (effect) en de logische oorzaak (causedBy). Een ActionType kan los van een MetaType aangemaakt worden.

Attributen
- EventName causedBy
  Het Event waar dit type actie op reageert. Dit kan bijvoorbeeld een muisklik zijn, of een Event dat gemaakt is om acties aan elkaar te linken.
- IEffect effect
  Wat er moet gebeuren bij het uitvoeren van de actie.
- String name
  De naam van dit type actie.
- PreconditionClass condition
  De preconditie moet true zijn om het effect uit te kunnen voeren. Als de condition null is wordt dit gezien als true.

Effect

Effects zorgen voor de link tussen ons systeem en het systeem van het bedrijf. Het programmeren hiervan is niet mogelijk in onze GUI en moet dus handmatig door de programmeur gedaan worden. Wij leveren een interface met de volgende methode:

- performEffect(MetaEntity owner, Object sender, Object args)
  Voert de code van het effect uit.
  Owner is de MetaEntity die de Action bevat die dit effect heeft afgevuurd. Deze kan null zijn.
  Sender is het object dat het Event bevat waar dit effect op reageert.

Args is een object dat door sender wordt meegegeven, dat aanvullende informatie geeft over het event.

EventName

Elke Event heeft een EventName. De EventName kan in zekere zin gezien worden als het "type" van een event en wordt ook in de GUI gebruikt, waar Events alleen tijdens runtime gebruikt worden.

Attributen
- String name
  De naam van het event

Methodes
- equals(Object other) : boolean
  Vergelijkt twee EventNames op het name-attribuut (en geeft true als ze gelijk zijn).
- hashcode() : int
  Geeft de hashcode van de name.

MetaType

MetaTypes zijn die types die MetaEntities binnen ons systeem hebben. Ze representeren de klassen die in de GUI door de gebruiker gemaakt worden.

Attributen
- MetaType parent
  Hiermee is het mogelijk om subklasses te maken.
- Map<String, Property> properties
  De attributen van het type (de klasse)
- Annotation[] annotations
  Annotations worden door het bedrijf gebruikt en kunnen hier opgegeven worden. Annotations zijn meta-data die als flag fungeren.
- String name
- ActionType[] actionTypes
  Lijst van acties waar dit MetaType de owner van is.

Methodes
- hasParent(String name) : boolean
  Geeft true als er ergens hoger in de hierarchie een MetaType met de opgegeven naam bestaat. Dit betekent dat dit MetaType erft van het opgegeven MetaType.

Module

Een module is een stukje van het programma wat in- en uitgeschakeld kan worden zonder dat de rest van het programma niet meer werkt.

Attributen
- List<MetaType> classes
  "Losse" klassen binnen deze module, dus die niet in onderliggende modules zitten.
- List<ActionType> autoActions
  De acties die niet aan een instantie van een MetaType gebonden zijn. Hierdoor kan het systeem ge-bootstrapped worden met behulp van externe Events.

- List<Module> subPackages
  De onderliggende modules.

Methodes
- getSubPackages() : List<Module>
- getClasses() : List<MetaType>

Project

Het Project bevat informatie over de modules en classes die gemaakt zijn.

Attributen
- ModulesXMLPath
  Wordt gebruikt om de modules te vinden.
- DynamicClassesJarPath
  De JAR waarin de classes staan die door het bedrijf en de GUI worden gemaakt (denk aan Preconditions en Effects).

Property

Properties zijn de attributen van de klassen die door MetaType-instanties gedefiniëerd worden. Deze constructie is gebruikt om met de GUI dynamisch attributen aan klassen toe te kunnen voegen.

Attributen
- String name
  De naam van de Property. Per MetaType moet deze naam uniek zijn.
- Class type
  Het type van het attribuut.
- Annotation[] annotations
  Annotations worden door het bedrijf gebruikt en kunnen hier opgegeven worden. Annotations zjin meta-data die als flag fungeren.

5.4. Reflectie

De ontwerpfase verliep vlot, en we zijn eigenlijk geen grote problemen tegengekomen. Wel stuitten we op een paar functies die we in het ontwerp wilden stoppen, waarvan we niet zeker wisten of ze mogelijk waren, en waar later extra onderzoek voor nodig was. Bijvoorbeeld het compileren van precondities en effecten. Ook was de rol van enkele klassen binnen het model, zoals MetaEntity en MetaType, moeilijk uit te leggen, omdat er met een metamodel gewerkt wordt.

6. Implementatie

Dit hoofdstuk beschrijft de implementatiefase van het project. In paragraaf 6.1 wordt begonnen met welke tools wij voor de implementatie hebben gebruikt. In 6.2 wordt de taakverdeling beschreven volgens welke wij gewerkt hebben. Paragraaf 6.3 behandelt het uiteindelijke ontwerp, dus wat er veranderd is aan het ontwerp besproken in hoofdstuk 5. Paragraaf 6.4 gaat over de implementatie van de GUI. Tot slot wordt in paragraaf 6.5 gereflecteerd op de problemen in deze fase.

6.1. Tools

Wij hebben bij het implementeren van het systeem verschillende hulpmiddelen gebruikt. In deze paragraaf worden deze op een rijtje gezet.

- Java Devolopment Kit 6.0
- Eclipse SDK 3.2.0
- Eclipse SDK 3.4.x
- Netbeans IDE 6.5.1
- TortoiseSVN 1.5.0.13316
- http://svn2.xp-dev.com, als repository voor het project.
- JUnit 4.1.0

6.2. Taakverdeling

Er is niet vooraf een planning gemaakt voor deze fase. De verdeling is gedaan door een lijst bij te houden met onderdelen die geïmplementeerd moesten worden. Hier kozen we telkens een onderdeel van waar we vervolgens, meestal los van elkaar, aan gingen werken. Het is moeilijk om precies weer te geven wie welke onderdelen heeft geïmplementeerd, omdat klassen die door de een gemaakt zijn vaak later weer door de ander gewijzigd zijn. De verdeling ziet er uiteindelijk ongeveer zo uit:

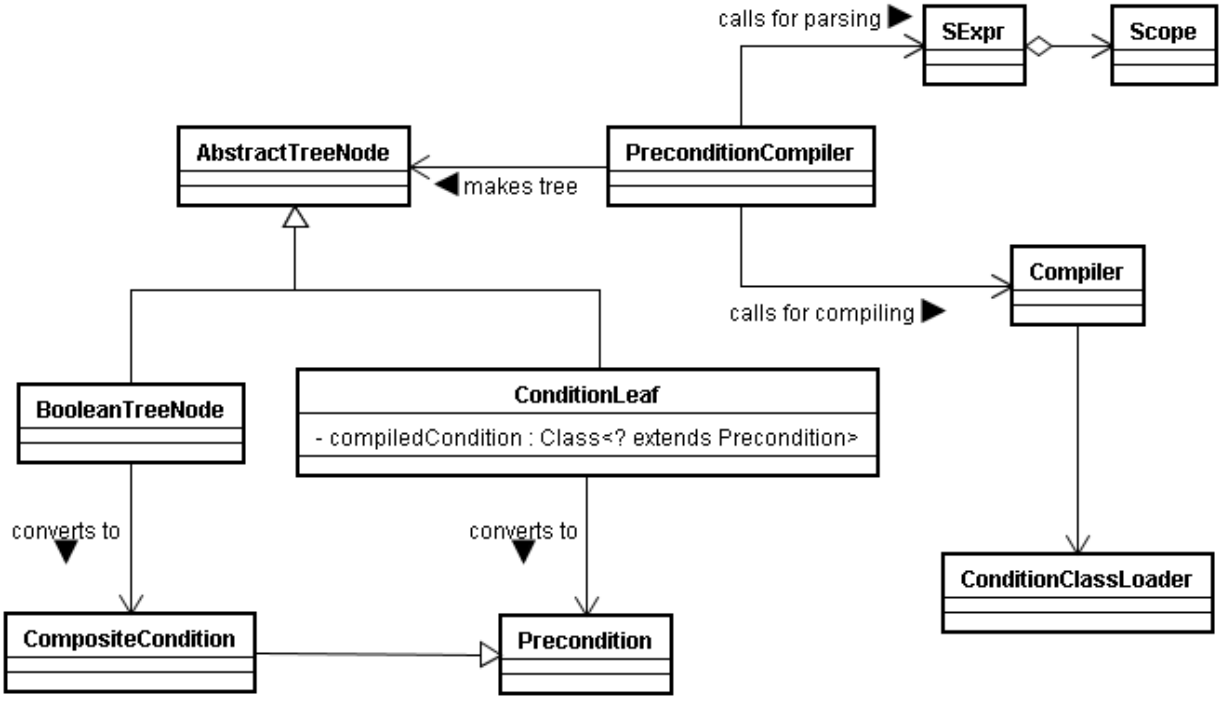
Taak	Persoon
Model package	Tom
Preconditie parsen & compileren	Sid
User interface	Tom
Runtime package	Sid

6.3. Uiteindelijke ontwerp

In hoofdstuk 5 is het ontwerp besproken zoals het er op papier, voor het implementeren, uitzag. Tijdens de implementatie zijn er enige wijzigingen aan het ontwerp aangebracht, of delen toegevoegd. Omdat het niet nuttig is het hele ontwerp hier weer te herhalen, hebben we ervoor gekozen in deze paragraaf alleen deze wijzigingen te bespreken.

6.3.1. Parsen en compileren van precondities

Aan dit gedeelte van het systeem zijn de meeste klassen toegevoegd, en zijn ook de gebruikte klassen een beetje veranderd. De volgende diagram geeft een overzicht van het parsen en compileren van precondities.



Figuur 12. *Parsen en compileren van precondities.*

Dit parsen en compileren gebeurt wanneer een project vanuit de GUI wordt geëxporteerd. Het zou slecht zijn voor het gebruikersgemak als dit eerder zou worden gedaan, bijvoorbeeld bij elke wijziging van een preconditie, want compileren duurt relatief lang. In de GUI worden precondities ingevoerd als Strings. Hier moeten klassen van worden gemaakt die tijdens de uitvoer worden aangeroepen.

Om dit te doen wordt eerst een (statische) methode van de PreconditionCompiler aangeroepen met een preconditie-String als argument.

De PreconditionCompiler roept vervolgens SExpr aan om deze string te parsen. Tijdens het parsen wordt gekeken naar boolean operatoren ('&', '|' '!') en aan de hand van deze wordt de String opgedeeld in kleinere precondities. De reden hiervoor is de modulariteit van het systeem; als een van de precondities faalt, doordat bijvoorbeeld een klasse die hij aanroept niet meer bestaat, moet het systeem gewoon deze ene preconditie negeren en verdergaan met het evalueren van de rest van de preconditie. Op deze manier kunnen er naar willekeur gedeeltes van het systeem worden uitgeschakeld zonder dat het crasht.

Wanneer tijdens het parsen een stuk String wordt gevonden wat niet meer verder kan worden opgedeeld in kleinere precondities, wordt deze String verwerkt in een .java bestand (die Preconditie extend) en gecompiled.

Van de structuur van de grote preconditie wordt zo een boom gemaakt, die de kleinere, gecompileerde, precondities, en de relaties hiertussen (de boolean operators) bevat. Deze boom kan samen met de rest van het project worden weggeschreven naar een xml bestand, en voor het uitvoeren van een spel weer worden ingelezen. Om de precondities hier te gebruiken worden de Preconditie-klassen, opgeslagen in de boom, geïnstantieerd en wordt

de boom zelf omgezet naar een Preconditie.

In dit nieuwe ontwerp wordt de PreconditionClass klasse, te zien op figuur 10, niet meer gebruikt. De functionaliteit hiervan is opgevangen door de ConditionLeaf en de gecompileerde preconditieklassen.

6.3.2. Utilities package

Naast de view, model en runtime packages hebben we er tijdens het implementeren voor gekozen een nieuw package aan te maken, het utilities package. Hier zijn de klassen ingekomen die wel door de model en runtime klassen worden gebruikt, maar toch niet echt tot het model of de runtime behoren. De inhoud van dit package bestaat nu uit:

- De klassen voor het parsen en compileren van Preconditions
- De klassen voor het compileren van Effecten.
- ClassLoaders (ConditionClassLoader en JarFileClassLoader)
- Pair.java, een klasse die in veel situaties handig is, bijvoorbeeld als return type van een methode waarbij je meerdere types wil teruggeven.

6.3.3. ClassPropertyType, MetaPropertyType

Om tijdens runtime bij het toekennen van een waarde aan een property van een metatype te controleren dat het property inderdaad het juiste type heeft, wordt het runtime type van die waarde vergeleken met het type dat voor dat property gespecificeerd was. In het oorspronkelijke ontwerp werd hiervoor in Property een Class gebruikt, maar dit werkt niet voor zowel meta- als gewone types. Daarom hebben we Property aangepast: Property bevat nu een AbstractPropertyType, dat kan worden geimplementeerd door middel van een MetaPropertyType ClassPropertyType, welke beide de verwachte functionaliteit aanbieden voor respectievelijk meta- en gewone types.

6.3.4. Annotations

Bij het ontwerp gingen we ervan uit dat Java Annotations door middel van code geïnstantieerd konden worden. Dit bleek echter niet het geval te zijn. Daarom was het niet mogelijk om MetaTypes en Properties een lijst van Annotations mee te geven. In plaats daarvan wordt dus een String meegegeven. Zodanig kan alsnog metadata worden toegevoegd, wat het doel van de annotations was.

6.4. GUI

We hadden tijdens het ontwerp nagelaten de GUI in detail uit te werken (zie paragraaf 6.5 voor meer informatie hierover), wat voor enkele creatieve oplossingen heeft gezorgd.

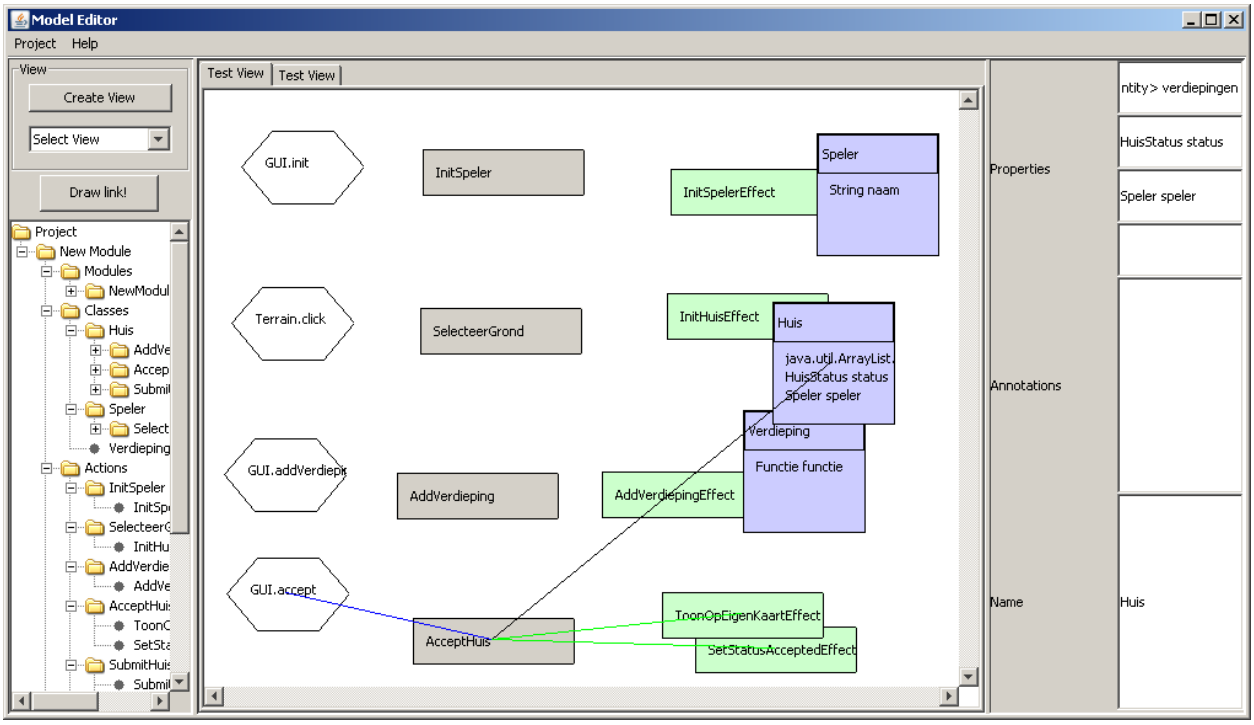
We hebben voor de GUI een facade-package gemaakt van het model-package, zodat hierin aanpassingen konden worden gemaakt ten behoeve van de GUI zonder het model-package te beinvloeden.

De lijst met door de gebruiker instelbare eigenschappen (zie voorbeeld, rechter balk) van de diverse elementen van het model wordt met behulp van reflectie opgebouwd op basis van de model-facade klasse, en is hierdoor uitermate geschikt voor het toevoegen van elementen aan de model-facade. Zo kunnen bijvoorbeeld nieuwe soorten Effect gemakkelijk worden toegevoegd aan de GUI.

Het paneel met de project-boom is, in tegenstelling tot het oorspronkelijke ontwerp, aan

de linker kant geplaatst, zodat het tegelijk met de lijst van eigenschappen zichtbaar is. Hierdoor is het makkelijker om een element uit de boom te selecteren, en daarvan de eigenschappen te veranderen.

Er zijn, in tegenstelling tot het oorspronkelijke ontwerp, geen knoppen meer om onderdelen toe te voegen. Dit gebeurt nu doormiddel van een context (rechter-muis) menu. Relaties tussen model-elementen kunnen worden gedefinieerd door middel van de "Draw Link" knop, en vervolgens op de twee relevante elementen te clicken.



Figuur 13. *Uiteindelijke ontwerp GUI.*

6.5. Reflectie

Precondities

In de requirements staat dat de gebruiker precondities moet kunnen opgeven aan de gemaakte acties. Deze eis hadden wij, zoals ook in paragraaf 6.3 staat, nog niet uitgebreid uitgewerkt in het ontwerp. In paragraaf 6.3 staat beschreven wat het uiteindelijke ontwerp hiervan is geworden. Hier wordt meer het proces en het onderzoek van dit probleem beschreven.

Parsen
Bij het parsen is het probleem om een preconditie (String) op te delen in verschillende stukjes die in een preconditie-boom van CompositeConditions en Preconditions kunnen worden gezet. Eerst is besloten om de precondities hiervoor een ietwat andere syntax te geven dan de normale Java syntax; haakjes die de structuur aangeven moeten worden ingevoerd als curly brackets. Op deze manier is het voor de parser een stuk makkelijker om de structuur van een preconditie te herkennen. Ook mogen alleen enkele boolean operators worden gebruikt, dus geen "&&" of "||". De operatoren zijn bij het uitvoeren altijd lazy.

Als voorbeeld:

"(true | false) && true" moet worden: "{true | false} & true", maar bij een functieaanroep moeten de parentheses wel blijven staan.

Om te parsen hebben wij er vervolgens voor gekozen om de String eerst om te zetten naar een abstracte syntax, een S-expressie. Hierbij wordt gelet op de curly-braces structuur van de preconditie. Deze S-expressie wordt daarna aan een andere parser gegeven, die parst op boolean operators en de S-expressie omzet in een boom.

Compileren
Het compileren heeft veel te maken met het parsen, maar kostte minder tijd om te implementeren. De reden hiervoor is dat we gewoon de Java compiler gebruiken, en hier een bestandje aan geven. Hier hoefde, naast het schrijven van een tijdelijk bestandje, verder weinig voor geïmplementeerd te worden. Problematisch was dat we dit compileren tijdens het uitvoeren van de GUI wilden doen, en vervolgens direct de klassen wilden kunnen gebruiken. Hier waren we nog onbekend mee en hebben we onderzoek naar moeten doen.

Er zijn binnen Java verschillende manieren om code tijdens runtime te compileren. De aantrekkelijkste was aanvankelijk het gebruik van de nieuwe compiler API die in Java SE 6 is geintroduceerd. Deze optie viel echter af, omdat hierbij de compiler wordt opgevraagd die op het systeem, de computer waarop ons programma wordt uitgevoerd, staat geïnstalleerd. Dit betekent dat:

- Het programma enkel op computers kan draaien waar een JDK op staat. Dit is geen groot probleem, want het programma wordt gebruikt door developers, maar legt toch een beperking op aan de gebruiker.
- Het programma ook daadwerkelijk met deze JDK moet worden uitgevoerd, omdat de compiler anders alsnog niet wordt gevonden. Hierdoor zou de gebruiker het programma niet "out of the box" kunnen gebruiken, wat niet goed is voor de portability en usability van het programma.

Hoewel deze methode de meeste mogelijkheden biedt voor het compileren, hebben deze twee beperkingen er toch voor gezorgd dat we de wat oudere methode voor het compileren hebben gebruikt. Deze methode maakt gebruik van de com.sun.tools.javac.Main klasse in tools.jar (standaard met Java meegeleverd). Met deze methode kun je een .java bestand opgeven en wordt deze gecompileerd, zonder dat er een JDK op de computer geïnstalleerd hoeft te zijn. Het enige nadeel hiervan is dat tools.jar, ongeveer 11 MB groot, meegeleverd moet worden met ons systeem, maar dit zien wij niet als een probleem.

GUI
De GUI was nog niet goed uitgewerkt in het ontwerp. Het enige wat aan de GUI ontworpen was was hoe deze eruit ging zien, maar de achterliggende functionaliteit was nog niet uitgedacht. Dat we hier toen minder op gelet hebben komt doordat dit niets met de functionele basis van het systeem (de must-haves in de model en runtime packages) te maken had, waar we ons tijdens de ontwerpfase vooral op hebben geconcentreerd. Dit heeft als gevolg gehad dat we niet tevreden waren met de code voor de samenwerking tussen het model en de GUI. Deze is twee weken voor het einde van het project nog voor een gedeelte herschreven.

6.6. Conclusie

Onze implementatie ondersteunt de volgende functionele requirements. Groen (+) wordt ondersteund, geel (~) gedeeltelijk, en grijs (-) wordt niet ondersteund.

Eis	Beschrijving	Commentaar
M1 (+)	Met het systeem kan een nieuw type spel-object worden gespecificeerd.	
M2 (+)	Met het systeem kunnen de eigenschappen van een spel-object type worden gewijzigd.	
M3 (+)	Met het systeem kan een spel-object type aan een spel-model worden toegevoegd.	
M4 (+)	Het systeem ondersteunt het verwijderen van een spel-object type uit een spel-model.	
M5 (+)	Het spel-model bevat een hiërarchische (boom) structuur van "modules".	
M6 (+)	Het systeem ondersteunt het toevoegen van een causaal effect en een noodzakelijke voorwaarde voor dat effect aan een spelgebeurtenis.	
M7 (+)	Het systeem ondersteunt het uitvoeren van een spel-model vanuit een spel.	
S1 (+)	Functionele eisen **M1-M6** worden ondersteund door middel van een grafische gebruikersinterface.	
S2 (+)	De interface kan relaties tussen spel-objecten en effecten visualiseren.	
S3 (~)	Het systeem sluit aan op de bestaande spel-architectuur en SGE.	Er wordt geen gebruik gemaakt van bestaande Events. Hiervoor zou een kleine aanpassing aan de SGE EventManager nodig zijn.
C1 (~)	Het effect in functionele eis **M6** kan bestaan uit een keten van andere effecten.	Meerdere effecten worden als keten uitgevoerd, maar het is niet mogelijk hiervan de volgorde te bepalen.
W1 (-)	Een module in een spel-model kan worden uitgezet tijdens het uitvoeren van een spel dat van dat spel-model gebruik maakt.	Deze functionaliteit is niet geimplementeerd, al is het ontwerp er wel voor geschikt.

6.7. Aanbevelingen

Alhoewel het product in principe werkt, zouden de volgende aanpassingen gemaakt moeten worden om het echt bruikbaar te maken. Het gaat hier om uitbreidingen: hier zijn wij niet aan toegekomen, maar zouden later (door andere mensen) geimplementeerd kunnen worden. Dit zijn dingen waar bij het ontwerp van het systeem wel rekening mee is gehouden.

Het maken van aanpassingen aan het model tijdens runtime. De opdrachtgever had aanvankelijk aangegeven hierin geinteresseerd te zijn, en we hebben hier in ons ontwerp dan ook rekening mee gehouden. Later bleek echter dat dit niet daadwerkelijk van belang

was, en daarom is dit niet geimplementeerd. Op het moment moet een model eerst geexporteerd worden voor de klassen kunnen worden ingelezen, maar met het gebruikte Adaptive Object Model is het zeker mogelijk om dit dynamischer te doen.

Het volledig parsen van MetaType-aanroepen. Momenteel worden deze slechts deels geparsed, en worden onderdelen direct doorgegeven aan een Java compiler. Hierdoor is de syntax voor het schrijven van precondities onnodig complex. Door deze expressies volledig te parsen is het mogelijk om bijvoorbeeld het gebruik van MetaTypes te maskeren als echte klassen, wat voor de programmeur meer intuitief is.

Op het moment wordt het in de GUI al snel erg druk met alle events, acties en objecten. Dit hebben we geprobeerd te verlichten door de mogelijkheid in te bouwen meerdere "views" te kunnen maken en tegelijk open te hebben, waarin delen van het programma apart kunnen worden bewerkt. Maar met het achterliggende systeem zou het ook mogelijk moeten zijn een extra laag over het model te maken, zodat er op een hoger niveau gewerkt kan worden. Hierbij kan bijvoorbeeld gedacht worden aan het direct kunnen linken van acties, zodat de events uit het zicht worden geschoven. Dit sluit misschien ook beter aan bij de reeksen van gebeurtenissen zoals Tygron deze opstelt in scenario's. Zie ook hoofdstuk 8.

7. Software kwaliteit en testen

In dit hoofdstuk wordt aandacht besteed aan de kwaliteit van de software. Eerst wordt in paragraaf 7.1 de kwaliteit van ons systeem beschreven aan de hand van algemeen bekende factoren die de kwaliteit van software bepalen. In paragraaf 7.2 wordt alles rondom het testen van het systeem besproken; wat er is getest, waarom, en hoe.

7.1. Factoren voor software kwaliteit

Understandability

De code in de klassen die wij hebben gemaakt kunnen via de GUI worden aangeroepen, maar het is ook mogelijk om zonder deze GUI een model te maken, al raden wij dit niet aan. Daarnaast zijn de gebruikers developers en programmeurs, die het systeem eventueel nog zouden willen uitbreiden. Understandability van ons model en de code is dus belangrijk bij dit project. Bij alle klassen, attributen en methoden in de model, runtime en utility packages is JavaDoc gemaakt. Daarnaast is bij sommige stukken die moeilijker in elkaar zitten (bijvoorbeeld de parser) wat extra pseudocode toegevoegd. Ook wordt in de documentatie (soms meerdere malen) uitgelegd wat de rol van een klasse in het systeem is.

Completeness

Het systeem is compleet in de zin dat alle noodzakelijke onderdelen om met het systeem te kunnen werken aanwezig zijn. Wat niet compleet aan het systeem is is dat niet alle requirements zijn gehaald. Ook wordt de error handling op dit moment gedaan door gewoon te throwen, die later eventueel opgevangen kunnen worden. Dit zien wij niet als een groot probleem -onze gebruikers zien misschien zelfs liever een gedetailleerde stacktrace dan een minder informatief bericht- maar hierdoor is het duidelijk minder compleet.

Conciseness

Omdat deze kwaliteit vooral belangrijk is bij systemen waarbij de hoeveelheid geheugen een bottleneck kan zijn, zien wij dit niet als een kwaliteit die hoge prioriteit heeft. Als er tijd over zou zijn zou er meer aan optimalisatie gewerkt kunnen worden. Wel is het zo dat in Eclipse, in makkelijke gevallen, automatisch wordt herkend of code bereikbaar is.

Portability

De gebruikte programmeertaal is Java, wat van zichzelf al bekend staat om zijn hoge portability. Er worden voor het gebruik van het systeem (model, runtime en GUI) geen extra eisen aan het systeem gesteld. Hier is rekening mee gehouden bij bijvoorbeeld de keuze van de compiler. Er is, als van Effecten gebruik wordt gemaakt, enkel een tekstverwerker op het systeem nodig om Java bestanden te schrijven.

Consistency

Er zijn enkele begrippen, die wij veel gebruiken in ons systeem en de documentatie, die op verschillende manieren kunnen worden aangeduid. Bijvoorbeeld de termen MetaType en class worden uitwisselbaar gebruikt. Net zoals MetaEntity en object, Property en attribuut. Ook het woord "model" wordt vaak gebruikt, zowel voor het model dat wij tijdens de

ontwerpfase voor ons systeem gemaakt hebben als een model dat in de GUI van ons systeem kan worden gemaakt. Welk van de twee wij bedoelen is dan uit de context op te maken.

Maintainability

Het model dat in ons systeem gebruikt wordt, het Adaptive Object Model, hoeft waarschijnlijk niet snel aangepast te worden. De enige aanpassingen die hier echt nuttig voor zouden zijn, zouden meer uitbreidingen zijn, die om het systeem heen worden gemaakt. En dit is met Java goed mogelijk. Enkele mogelijke aanpassingen worden in de paragraaf aanbevelingen in implementatie beschreven, en de code is goed gedocumenteerd, om het uitbreiden te vergemakkelijken.

Er is geen rekening gehouden met het reserveren van bijvoorbeeld geheugen, opslag of processorgebruik voor toekomstige aanpassingen. Dit vonden wij niet nuttig.

Testability

Dankzij het (gebrek aan) ontwerp van de GUI viel dat onderdeel slecht te testen. Het model en runtime gedeelte van ons systeem kon wel goed getest worden. Dit is deels gebeurd.

Usability

Er is een graphical user interface bij ons model gemaakt. Hier zijn helaas geen uitgebreide usability tests op gedaan om te kijken wat de gebruikers ervan vinden. De meeste functionaliteit zit onder de rechter muisknop, wat voor ervaren computergebruikers redelijk intuitief zou moeten zijn, en bij de demonstratie zijn nog enkele tips meegenomen. Verder komt er een kleine beschrijving van wat alle symbolen in de user interface betekenen. Error messages weergeven wordt, zoals eerder gezegd, niet gedaan.

Reliability

Wij verwachten dat het product, als er geen aanpassingen aan worden gemaakt, blijft functioneren op de manier zoals het dat nu doet. Het zal zich dus niet onverwachts anders gaan gedragen of niet meer werken wanneer het op een andere computer wordt gebruikt. De reliability is echter niet getest.

Structuredness

Het systeem is in Java geschreven. Dit zorgt vrijwel automatisch al voor een structuur bestaande uit objecten. Verder is gebruik gemaakt van het Adaptive Object Model, wat een paar patterns met zich meebrengt.

Efficiency

In ons systeem draait het voornamelijk om de flexibiliteit. Dit heeft tot gevolg dat er niet met gespecialiseerde klassen wordt gewerkt, maar met zogenaamde metaklassen, waardoor het over het algemeen tijdens de uitvoer trager is. We proberen hier rekening mee te houden door zo veel mogelijk klassen te precompilen of in te lezen voordat begonnen wordt met het spelen van een spel en verwachten -met het oog op de snelheid van computers vandaag de dag- niet dat dit voor problemen zal zorgen.

Security

Security binnen in het model is niet op gelet, en ook niet echt belangrijk voor de spellen die door Tygron gemaakt worden. Klanten van Tygron spelen de spellen om zelf kennis en ervaring op te doen. Security met betrekking tot het opslaan van modellen, zodat de klanten niet zelf, buiten Tygron om, wijzigingen kunnen aanbrengen, is in kleine mate aanwezig. Het xml-bestand met het model wordt bij het exporteren in een jarfile geplaatst.

7.2. Het testen

Deze paragraaf beschrijft alles wat wij rondom het testen van onze applicatie hebben uitgevoerd. Er is voor gekozen om het testen niet heel uitgebreid, in verschillende documenten, te beschrijven. Reden hiervoor is dat op het moment van schrijven het afkrijgen van het eindrapport en de functionaliteit prioriteit hebben.

Als eerste zal in paragraaf 7.2.1 het doel van dit document worden beschreven, wie het zou moeten lezen en waarom. Vervolgens wordt in paragraaf 7.2.2 beschreven welke onderdelen wij belangrijk vonden om te testen, welke onderdelen we minder uitgebreid hebben getest, en de motivatie hiervoor. Tot slot wordt in paragraaf 7.2.3 een overzicht van de uitgevoerde tests gegeven. Deze tests zijn met behulp van JUnit geïmplementeerd en worden met het systeem meegeleverd.

7.2.1. Wat wordt er getest

Na een klassendiagram van het systeem gemaakt te hebben is dit voor een gedeelte direct naar java-code getransformeerd. Het gaat hier om de klassen uit de packages model en runtime waarmee een MetaType kan worden gemaakt, hier een actie (en effect) aan kan worden gekoppeld, en het geheel met behulp van events kan worden uitgevoerd. De code die hierbij is gemaakt is dus niet test-driven tot stand gekomen, maar bestaat voor het grootste gedeelte uit getters en setters en andere "simpele" functionaliteit. Na een paar testscenario's met deze basis te hebben uitgeprobeerd, waar al deze klassen in werden gebruikt, gaan wij er van uit dat deze code goed werkt en niet meer uitvoerig getest hoeft te worden.

Wat noodzakelijker is om te testen zijn ingewikkeldere procedures zoals het parsen en compileren van precondities, het maken van een boom hiervan, en testen of Events en MetaEntities zich bij het instantieren goed aanmelden aan het systeem (zodat ze door elkaar gebruikt kunnen worden).

Wij gebruiken geen tools om de GUI geautomatiseerd te testen. Het testen hiervan is handmatig gebeurd door het uitvoeren van de geïmplementeerde handelingen. Bijvoorbeeld het maken van een nieuwe klasse/actie en het wijzigen van eigenschappen.

Verder zijn er enkele performance-tests uitgevoerd om vast te stellen in hoeverre de flexibiliteit ten koste is gegaan van de efficiency.

7.2.2. Tests

MetaType en MetaEntity

We zijn begonnen met enkele tests voor de klassen MetaType en MetaEntity, omdat deze klassen in het midden van het systeem staan, en alle andere klassen hiermee samenwerken. Deze tests zijn gewoon om te kijken of alles goed werkt, en niet in detail uitgewerkt. Hieronder staan enkele tests die hierop zijn uitgevoerd.

MetaType
- Aanmaken van een MetaType, en kijken of alles de goede waarden heeft gekregen.
- Het toevoegen van een property.
- het toevoegen van een property met bestaande naam.
- Het toevoegen van een property met een MetaType als type.
- Het toevoegen van een property met een Java klasse als type.

MetaEntity
- Maken van een MetaEntity met een correcte, bestaande klasse als type.
- Maken van een MetaEntity met een niet-bestaande klasse als type.
- Kijken of values van properties van het goede type zijn (zoals in het model gespecificeerd)
  - als de property een MetaType klasse als type heeft.
  - als de property een bestaande Java klasse als type heeft.

Samengestelde precondities

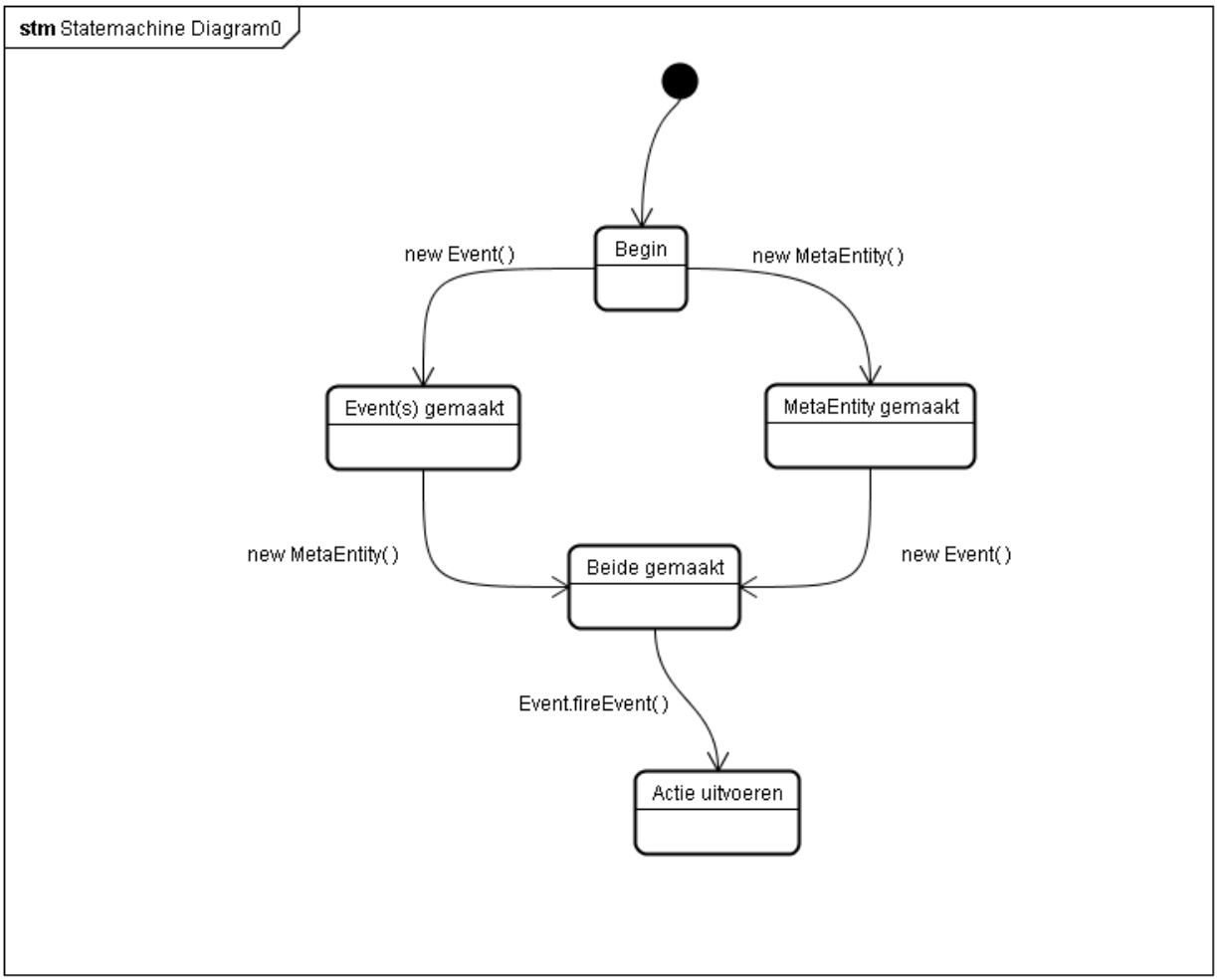
Voor samengestelde precondities hebben we MC/DC als adequaatheidscriterion gekozen. Daarom hebben we het volgende test-model afgeleid en hiervoor tests geimplementeerd.

Test Case	IsNegated	Left: Value	Left: Valid	Right: Value	Right: Valid	Operator	Decision
1	False	True	True	True	True	And	True
2	True	False	True	False	True	And	True
3	False	-	False	False	True	And	False
4	False	False	True	-	False	And	False
5	False	False	True	True	True	Or	True

Tabel 1. *MC/DC Coverage voor CompoundPrecondition.isTrue*

Maken van Events en MetaEntities

Voor het afvuren en aanmelden voor Events hebben we path-coverage als adequaatheidscriterion gekozen. De volgende state-machine beschrijft het aanmelden voor events, globaal. Op basis hiervan zijn ook tests geimplementeerd.



Figuur 14. *Path Coverage voor afvuren van event.*

Parsen en compileren

Het parsen en compileren van precondities werkt in ons systeem nauw samen. Het compileren gebeurt tijdens het parsen, en het is dus niet mogelijk het parsen en compileren apart te testen. In de PreconditionCompiler (de belangrijkste klasse bij het parsen en compileren) worden twee methodes getet: parseAndCompile(String), de "main" methode voor het parsen en compileren, en findOperator(String), een belangrijke methode tijdens het parsen.

Eerst findOperator(). Deze methode krijgt een String, met daarin mogelijk een of meerdere boolean operators. FindOperator moet de String splitten op de eerste operator die hij vindt, en dit samen met de operator zelf en zijn positie teruggeven. De operator kan zijn: '&' of '|'
De positie (een integer) kan zijn:
0: aan het begin van de String (met tekst rechts ervan)
1: aan het eind van de String (met tekst links ervan)
2: ergens middenin de String
3: zowel aan het eind als het begin (de enige character in de String dus)

Verder is er nog de mogelijkheid dat er een andere operator verderop in de String staat, welke hij moet negeren. Hier zijn de volgende testcases voor opgesteld en

geïmplementeerd:

operator	positie	andere operator aanwezig
'&'	0	nee
'|'	1	niet mogelijk
'&'	2	ja
'|'	3	niet mogelijk

Vervolgens is het parseAndCompile() getest. De parser werkt, zoals eerder uitgelegd, door eerst op structuur en negatie te parsen, en vervolgens op operatoren. Wanneer op structuur wordt geparst wordt een s-expressie teruggegeven, die zowel een value als een list (van s-expressies) kan zijn. Een value is gewoon een String. Als de parser een value zonder boolean tegenkomt, wordt deze string gecompileerd.

Wat volgt is een lijst met gevallen waar de parser mee om moet kunnen gaan, bij elk element waarop getest wordt en de test zelf:

value met boolean: "`1+1==2 & 2+2==2`"

list, met value links en list rechts: "` true & {1+1==2 & 2+2==4}`"
(De value is hier " true &" en de list bevat de value "1+1==2 & 2+2==4")

list, met list links en value rechts: "`{true | false} & true`"

aanroep van een methode, geneste structuur, meerdere operatoren op hetzelfde "niveau":
"`{{false | java.lang.Math.random()==3} & true | false} & true`"

overbodige haakjes, list-value-list structuur: "`{true | false} & true & {{{false}}}`"

zelfde met een extra operator in het midden: "`{true | false} & true & true & {false | false}`"

structuren verbonden door 1 enkele operator: "`{{true | false} & true} & {true & {false | false}}`"

En tot slot wordt in de volgende tests op verscheidene manieren negatie getest:
"`{!{true}}`"
"`{!{!true & !false}}`"
"`!{!true==!(true)|!(1==1)}`"
"`!{{!{true | false} & true} & !{true & !{false | false}}}`"

Performance

We hebben de performance van het veranderen van een waarde van een Property van een MetaType, het opvragen van de waarde van zo'n property, en het afvuren van een Event getest.

Om get en set met een normaal programma te vergelijken, is een kleine test-klasse gemaakt waar een waarde met behulp van een directe get en set methode kon worden

opgevraagd en verandert. Het afvuren van events werd vergeleken met het direct aanroepen van de preconditie en het effect.

	Tijd normaal (ms)	Tijd bij ons (ms)	Verhouding
Set (x 10^7)	156	1812	11.61
Get (x 10^7)	31	328	10.58
Event (x10^6)	32	171	5.34

Ons systeem werkt zo'n factor 10 langzamer dan een niet-dynamisch aanpasbare implementatie. Dit is snel genoeg om in een spel met real-time performance te werken, maar betekent wel dat het niet geschikt is voor spellen waarbij het uiterste uit de computer gehaalt moet worden. De vertraging komt vooral door het gebruik van HashMaps. Alhoewel hierdoor de algoritmische complexiteit wel goed is, werkt het toch erg langzaam.

8. Conclusies

Met het in dit project gemaakte systeem kunnen, vanuit een grafische interface, spel-objecten en diverse relaties daartussen worden gedefinieerd. Deze kunnen vervolgens worden geexporteerd, en gebruikt vanuit een spel. Deze spel-objecten kunnen worden gegroepeerd in modules, en een nieuw spel-model kan gemakkelijk worden opgebouwd door modules te combineren.

Hiermee is bewezen dat een dergelijk systeem mogelijk is, en wordt de opdrachtgever de mogelijkheid geboden met een dergelijk systeem te experimenteren, om te beslissen of en zoja hoe het nuttig is.
Alhoewel aan de meeste functionele eisen is voldaan, en uit tests is gebleken dat het eindproduct werkt, hebben we niet veel vertrouwen in het nut van de applicatie. Sommige essentiele onderdelen van programmeren, zoals het definieren van een constructor, die in Java triviaal zijn, zijn met dit systeem omslachtig. De programmeur krijgt minder correctie en hulp vanuit het systeem dan hij vanuit Java gewend is; zoals het tijdens het programmeren (en dus voor het uitvoeren) controlleren of types overeenkomen.
Het definieren van nieuwe spel-objecten gaat met het systeem iets sneller dan vanuit Java, maar heeft een vergelijkbaar niveau van programmeer ervaring nodig. Voor het definieren van effecten voor conditionele acties is nog steeds een Java programmeur nodig. Het systeem kan hierdoor slechts een beperkte bijdrage leveren aan de efficientie van het uitvoeren van deze twee stappen.
Als echter spel-objecten en effecten eenmaal zijn gedefinieerd, kan met behulp van het systeem wel heel gemakkelijk relaties hiertussen worden gelegd.

Reflectie

Het Bachelorproject is het grootste project binnen de studie Technische Informatica. We hebben hierin veel geleerd. Zo hebben we veel geleerd op het gebied van projectvaardigheden doordat we 3 maanden intensief hebben samengewerkt. Doordat we voor een echte opdrachtgever hebben gewerkt, hebben we geleerd dat de wensen en behoeften van een opdrachtgever niet altijd identiek zijn. Bovendien hebben we het een en ander geleerd over het toepassen van software engineering in de praktijk, zoals het belang van duidelijke communicatie, rapportage, naleven van deadlines, nadrukkelijk analyserenen van de behoeftes van de klant, en het belang van ontwerpen voor implementatie.
We hebben ook enkele nieuwe dingen geleerd over de opbouw van een project, bijvoorbeeld een Plan van Aanpak of onderzoeksverslag waar we nog niet eerder mee te maken hebben gehad.

De samenwerking verliep gedurende het hele project goed. In het begin van het project, in de orientatiefase tot ergens halverwege het RAD, is vaak thuis gewerkt, met enkele afspraken met de begeleider en het bedrijf. Hier besproken we gelijk hoe we verder zouden werken. Deze manier van werken werd vergemakkelijkt door het gebruik van moderne 'collaboration tools' zoals Google docs, Google sites, en een TU weblog. Vooral Google docs is erg belangrijk gebleken: dit maakte het mogelijk om een gedeeld document op te zetten, waar we tegelijkertijd in konden werken. Na het onderzoek is dagelijks en op vaste tijden op de drebbelweg aan het project gewerkt.

In retrospect zouden we het een en ander anders hebben aangepakt. Het was beter geweest om in de analysefase een duidelijker beeld te krijgen van wat de opdrachtgever precies voor ogen had voor we begonnen met het ontwerp. Het is zeker zo dat we iets

hebben gemaakt waar de opdrachtgever hetgeen mee kan maken wat hij wil maken, maar niet precies op de manier waarop handig zou zijn. Het beter zijn geweest om het systeem nog meer "high-level" te maken, al zou dat niet een opdracht op het niveau van bachelorstudenten zijn geweest. In de aanbevelingen (hoofdstuk 9) staan een paar voorstellen die misschien beter gewerkt zouden hebben.

We zouden liever bovendien sneller een soort van prototype van het systeem voor handen hebben gehad dat we konden demonstreren, en waarop we van de gebruikers feedback hadden kunnen krijgen. We waren op zich al wel snel, in de zevende week, aan de implementatie bezig, maar dit werd een beetje tegengewerkt doordat we vervolgens nog veel documentatie moesten verbeteren. Nu was zo'n demonstreerbaar systeem er uiteindelijk wel, maar pas een week voor het eindverslag ingeleverd moest worden. In de tijd die we toen over hadden hebben we weinig meer aan de implementatie kunnen veranderen.

9. Aanbevelingen

Het niveau van abstractie waar we voor gekozen hebben sluit beter aan op de wensen van de opdrachtgever, dan de behoeften daarvan. Doordat het systeem zowel het schrijven en integreren van code toestaat en hiervan afhankelijk is, is het toevoegen van nieuwe functionaliteit niet gemakkelijker geworden met behulp van ons systeem. We zien twee mogelijk betere alternatieven:

Ten eerste, een systeem dat op basis van een model uit het probleem domein, een oplossing in het implementatie domein kan genereren. Dit heet domain-specific modeling (DSM), en is een actief en veelbelovend onderzoeksgebied. Hiervoor zou voor een specifiek spel een systeem moeten worden ontworpen dat een specifiek onderdeel door middel van een model kan beschrijven/implementeren. Bijvoorbeeld de relaties tussen actoren in Watergame, als ware een Workflow. Zo'n oplossing is specifieker dan ons systeem, maar hierdoor wordt het definieren van nieuwe functionaliteit wel aanzienlijk makkelijker. Dit loont echter alleen als het vooraf duidelijk is dat een bepaald spel meerdere malen op een bepaalde manier zal moeten worden aangepast. Er zijn wel standaard tools beschikbaar om een (grafische) domain-specific language te definieren en gebruiken.

Het tweede alternatief is een uitbreiding aan de IDE (Eclipse) met behulp van bijvoorbeeld een plug-in, dat in staat is voor de spel-object klassen in het project, die objecten en de relaties daartussen te visualiseren, en het verwijderen en rangschikken te faciliteren. Het maken van klassen en dergelijke zou handmatig moeten gebeuren, eventueel op basis van een code-template en met enkele beperkingen of aanpassingen om de werking van de plugin mogelijk te maken. Het wijzigen door middel van de plug-in zou in principe als refactorings kunnen worden geimplementeerd. Dit zou vergelijkbare functionaliteit bieden als ons systeem, zonder dat de programmeur sommige delen in een GUI, en andere delen vanuit code hoeft te implementeren.

Literatuurlijst

1. Loos, P. & Allweyer, T. (1998) *Process Orientation and Object-Orientation - An Approach for Integrating UML and Event-Driven Process Chains (EPC)* http://isym.bwl.uni-mainz.de/downloads/Publikationen/Loos_Allweyer_1998_Process_Orientation_OO_Integrating_UML_and_EPC.pdf
2. Wikipedia.org, *Event-driven Process Chain* http://en.wikipedia.org/wiki/Event-driven_process_chain
3. Yoder, J.W. & Johnson, R. *The Adaptive Object-Model Architectural Style* http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.pdf
4. Adaptiveobjectmodel.com (site by Yoder, J.W.) http://www.adaptiveobjectmodel.com/

Bijlagen:

Opdracht

TYGRON
Serious Gaming

Bachelorproject "Relatie- & logicamodule serious game engine"

Wat is Tygron Serious Gaming?

Tygron is een jong bedrijf dat zich specialiseert in managementgames met een ruimtelijk aspect en met veel spelelementen die in traditionele managementgames ontbreken. Denk hierbij aan 3D werelden, (online) multiplayer mogelijkheden, etc.
De spellen van Tygron geven spelers inzicht in de, vaak complexe, processen en relaties tussen actoren bij verschillende soorten gebiedsontwikkeling.
Tygron is gevestigd in de voormalige Caballero Fabriek in Den Haag.
Kijk voor meer informatie op http://www.tygron.nl .

Watergame

Het vlaggenschipspel van Tygron is de serious game "Watergame" (http://www.watergame.nl). Een eerste versie van dit spel is inmiddels klaar voor de markt en is al uitvoerig gebruikt bij de gemeente Tiel. De interesse naar Watergame is dusdanig groot, dat Tygron op zoek is naar mensen die kunnen helpen de beleving van Watergame nog verder door te tillen.
De Watergame is een "Multiplayer Serious Game" over de invloed van water op de ruimtelijke ordening. Doormiddel van het spel kan men:

- Leren over gebiedsprocessen,
- Draagvlak creëren bij gebiedspartners,
- Bewustwording en verandering teweegbrengen.

De opdracht

Eén van de belangrijkste diensten van Tygron is het leveren van maatwerk aan de hand van de bestaande spellen. In zo'n situatie analyseert Tygron, in samenwerking met zijn partners, de situatie van de klant en verwerkt de resultaten in een bestaand spel.
Om dit mogelijk te maken beschikt Tygron over een "Serious Game Engine" waarmee alle spellen op een modulaire wijze geprogrammeerd kunnen worden.
De kans is echter groot dat, bij een maatwerk product, de processen en afhankelijkheden uit de realiteit wél sterk lijken op die uit het bestaande spel maar nét een andere volgorde of relatie hebben, in zo'n situatie zal een en ander, soms grondig, aangepast moeten worden. Om dit ontwikkelproces te ondersteunen zijn er reeds wat ideeën voor een modulair systeem waarmee acties en "objecten" uit het spel op eenvoudig wijze aan elkaar gekoppeld kunnen worden zodat de invulling, betekenis en het gedrag hiervan slechts eenmalig geprogrammeerd hoeven te worden en de relaties ertussen vervolgens

op relatief eenvoudige wijze her te schikken zijn. Door hier vervolgens een GUI aan te koppelen ontstaat een grafische modelleer tool die goed gebruikt kan worden door een proces analist / spelontwikkelaar en tevens, als bijkomstigheid, als een visualisatie van het onderliggende proces of de onderliggende relaties voor de spelers zou kunnen fungeren.

Werkzaamheden

Voor de bovenstaande ontwikkelingen zijn een goed literatuuronderzoek, ontwerp en implementatie van groot belang.

Hoewel verwant, gaat het bij deze opdracht niet om een *full-fledged* grafisch programmeren systeem. De functionaliteit zal specifiek toepasbaar moeten zijn op een afgebakend gebied van spel logica.
Elke "module" kan een bepaald spelelement of een (gebruikers- of systeem-)actie beschrijven, waartussen relaties en afhankelijkheden gedefinieerd kunnen worden. In de module moet op een flexibele manier, maar mogelijk wel met enig handwerk, een set van eigenschappen en / of een procedure ingebouwd kunnen worden. Door middel van een input-output systeem, *dependencies*, een *gate*/check, etc. zouden de modules vervolgens gekoppeld kunnen worden. De beste invulling hiervan zal onderzocht moeten worden.

Voor de implementatie zullen de studenten gebruik kunnen maken van een door ons ontwikkeld platform (SGE), gemaakt in Java en de JMonkey Engine (http://www.jmonkeyengine.com), die gebruik maakt van Java en OpenGL.
Gezien de beperkte tijd zou een op zichzelf staand "*proof of concept"* ook volstaan.

Indien gewenst kunnen de studenten komen te werken op ons kantoor aan de Saturnusstraat 60 in Den Haag.

Contactpersoon Tygron: Alexander Hofstede jobs@tygron.nl, Tel. 015 - 711 27 38

Plan van Aanpak

Tygron Bachelor Project

Plan van Aanpak

25 Mei 2009

Sid Mijnders **Tom Lotgering**

Begeleider: Hans Geers, Bernard Sodoyer
Versie: 5e

Inhoudsopgave

Voorwoord
Dit is het Plan van Aanpak voor het bachelor's project van Sid Mijnders en Tom Lotgering voor Tygron Serious Gaming.

Samenvatting
Het te ontwerpen systeem moet in ieder geval een verzameling klassen/modules bevatten waarmee de beoogde relaties en eigenschappen van objecten efficiënt kunnen worden gedefinieerd. Voor het project zal de nadruk op het ontwerp en testen liggen. Er zijn 12 weken voor het project geplanned, waarvan 3 weken onderzoek, 3 weken ontwerp, en 5 weken voor implementatie.

1. Introductie

1.1. Aanleiding
Dit Plan van Aanpak is opgesteld als begin van het bachelor's project, als eerste 'deliverable'. Het geeft een nauwkeurige beschrijving van de opdracht, geeft een overzicht van de aanvankelijke kennis omtrent de opdracht, en hoe de opdracht uitgewerkt zal worden. Het is tot stand gekomen op basis van de opdracht definitie zoals aangeleverd door de opdrachtgever, en enkele gesprekken met de opdrachtgever en begeleidend docent.

1.2. Accordering en bijstelling

1.2.1. Accordering
Er zal met het Plan van Aanpak akkoord worden gegaan als het opdrachtgevende bedrijf,

dhr. Sodoyer en dhr. Geers het met de inhoud en opzet eens zijn.

1.2.2. Bijstelling
Het Plan van Aanpak zal in de eerste weken door ons worden aangepast en bijgesteld wanneer er nieuwe informatie beschikbaar is gekomen (bijvoorbeeld na gesprekken). Na een paar weken zal het plan goedgekeurd worden. Hierna zal elke bijstelling met de opdrachtgever besproken moeten worden.

1.3. Toelichting op de opbouw van het plan
Dit Plan van Aanpak is als volgt opgebouwd;

In hoofdstuk 2 wordt de projectopdracht beschreven. Hier wordt gespecificeerd wat de huidige situatie is, wat er precies veranderd of gemaakt moet worden, hoe dit gedaan moet worden en wat het nut hiervan voor de opdrachtgever is. Dit kan gezien worden als het contract waaraan beide partijen zich moeten houden.

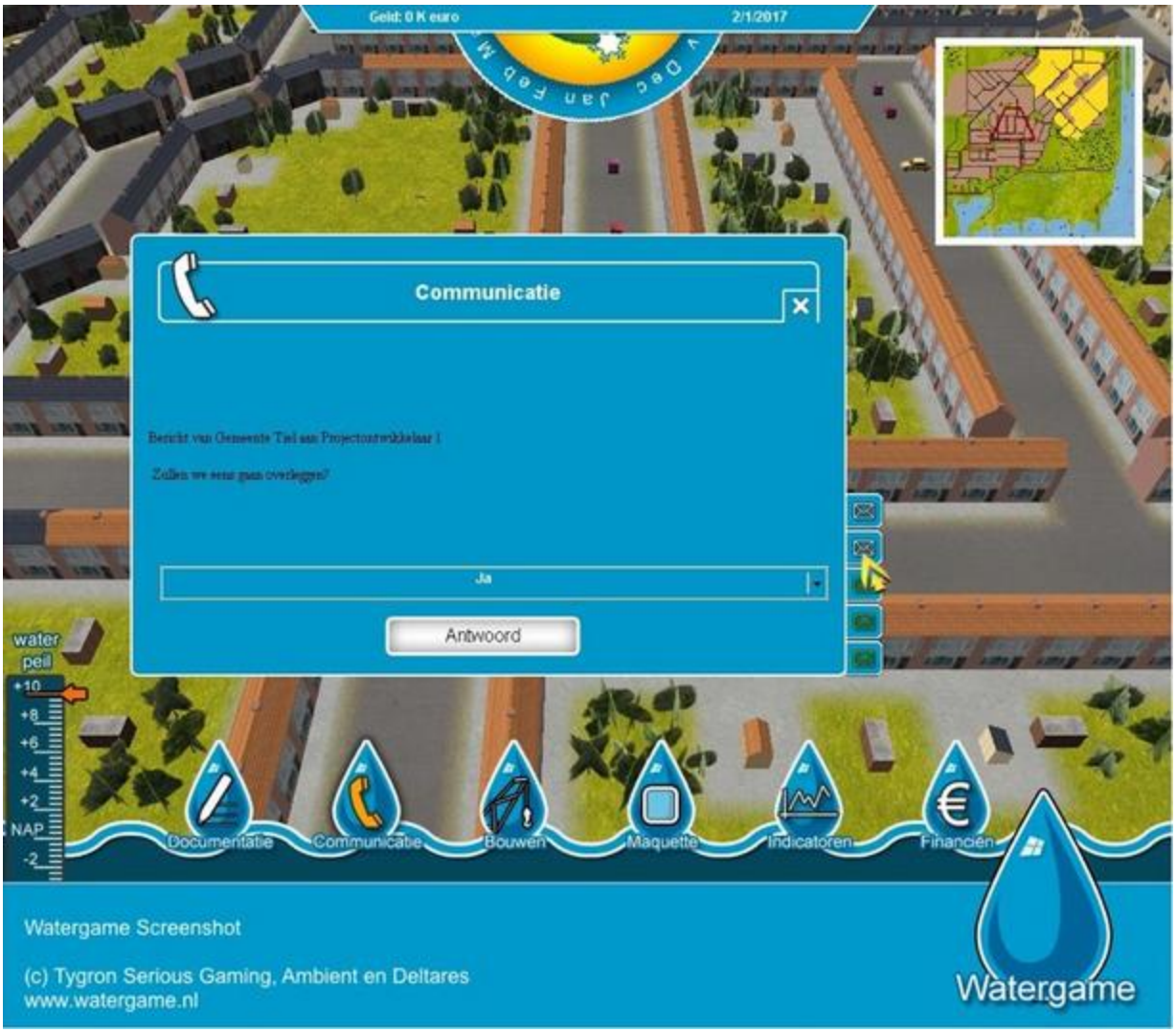
In hoofdstuk 3 wordt een planning opgesteld voor het project. Deze bestaat uit een lijst met activiteiten (compleet met de tijd en resources die naar schatting nodig zullen zijn) en een onderverdeling in fases.

Hoofdstuk 4 gaat over de projectinrichting. Het doel hiervan is weer te geven hoe de in hoofdstuk 3 gegeven aanpak uitgevoerd gaat worden. Er wordt o.a. vastgesteld wat er van de projectleden verwacht mag worden op het gebied van kennis en vaardigheden en welke verantwoordelijkheden zij op zich nemen. Maar ook zaken rondom het project, die niet direct met de inhoud te maken hebben, worden hier beschreven, zoals het monitoren van het project, de financiering, rapportering en over welke resources de projectleden beschikken.

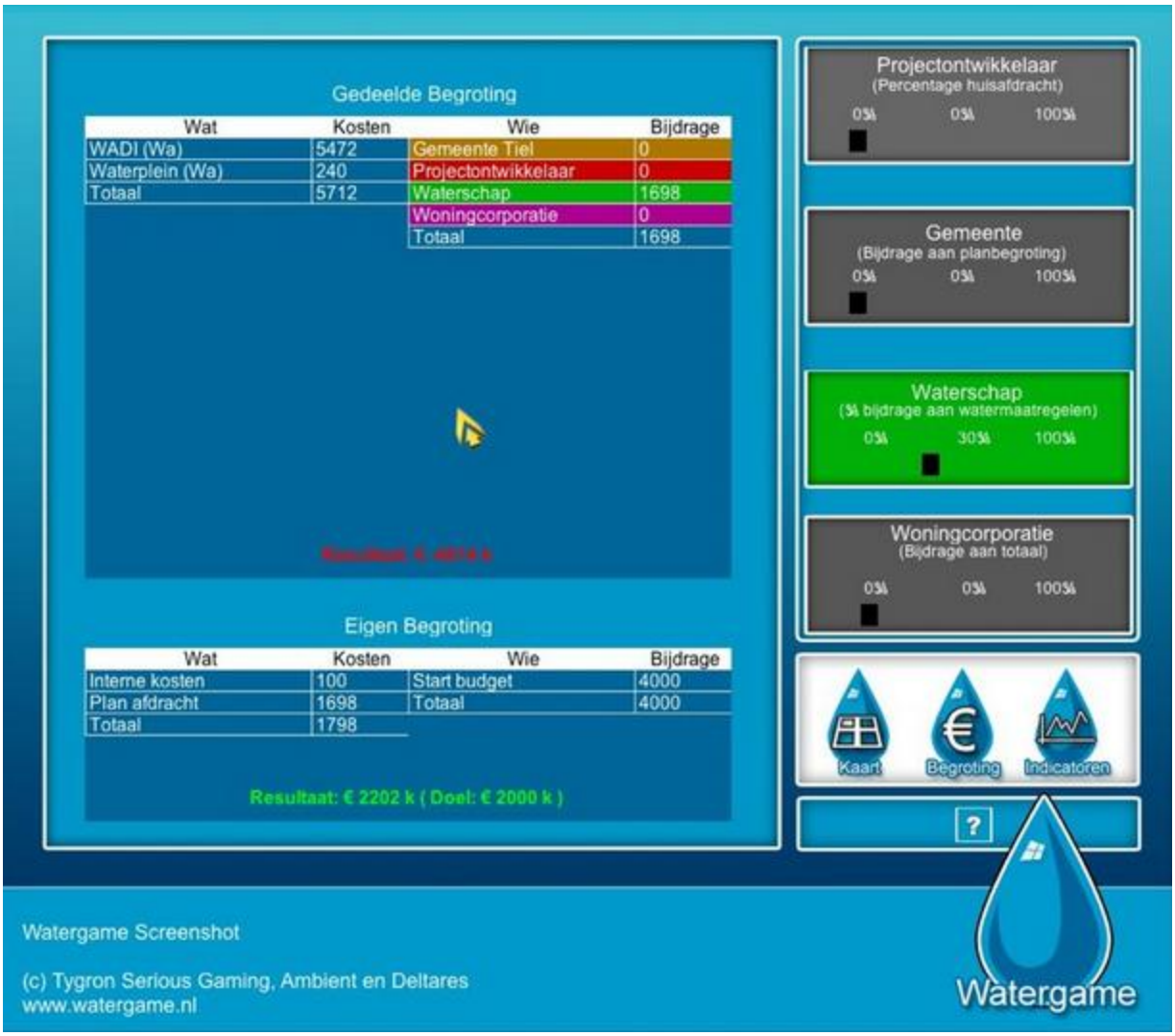
In hoofdstuk 5, kwaliteitsborging, wordt tot slot verder uitgewerkt wat de projectleden en met name de opdrachtgever kunnen doen om de kwaliteit van het project zo hoog mogelijk te houden. Ook worden hier risico's besproken die bij projecten als deze kunnen voorkomen en welke maatregelen er worden genomen om deze risico's te voorkomen.

2. Projectopdracht

2.1. Projectomgeving
Tygron is een bedrijf dat zich specialiseert in het maken van "serious games". Deze spellen kunnen de spelers inzicht geven in processen en relaties tussen actoren bij verschillende soorten stedelijke ontwikkeling. De serious game waar deze opdracht om draait is de "Watergame" (figuur 1, 2), een spel dat de gebruiker meer inzicht kan geven in de inrichting van het watersysteem in Nederland. Tygron heeft aangegeven dat de interesse voor dit spel hoog genoeg is om het spel beter te maken en de "beleving van de gebruikers verder door te tillen".



Figuur 1 *Watergame*



Figuur 2 *Watergame*

Een serious game wordt gemaakt door situaties van klanten te analyseren en te verwerken in een "bestaand" spel. Dit is een algemene opzet die eerder door Tygron is gemaakt en waarin specifieke situaties geprogrammeerd kunnen worden. Om dit goed te kunnen doen moeten de elementen, de processen en de afhankelijkheden hiertussen uit de echte wereld goed overeenkomen met die in de spelsituatie. Hiervoor bestaat momenteel een ad hoc oplossing, die geschikt is voor een enkel spel, maar die zich niet leent voor hergebruik, of voor aanpassing van het spel aan afwijkende situaties tussen klanten, zonder het spel grondig aan te passen.

2.2. Doelstelling project

Het doel van het project is om de opdrachtgever in staat te stellen op een efficiënte manier maatwerk te leveren aan haar klanten, door bestaande spellen aan te passen aan de specifieke situatie van de klant.

2.3. Opdrachtformulering

De doelstelling moet gerealiseerd worden door de gebruikers van het beoogde product van het project, namelijk programmeurs en proces analisten, de mogelijkheid te bieden op een efficiente manier de werking van het spel aan te passen, door de relaties tussen, en

eigenschappen van, bestaande spel-objecten te veranderen. De opdracht is dus om een dynamische, modulaire structuur (bestaande uit klassen) te leveren waarmee de programmeurs makkelijk alle objecten en acties van het spel aan kunnen maken en hier relaties en eigenschappen aan kunnen toekennen. Deze structuur moet er toe leiden dat situaties uit de echte wereld beter na te bootsen zijn in het spel. De opdrachtgever heeft aangegeven dat het nuttig zou kunnen zijn als deze aanpassingen ook door de administrator van het spel, tijdens het spelen daarvan, moeten kunnen worden gemaakt. Alhoewel niet noodzakelijk voor de goedkeuring van het product, zou dit idealiter door middel van een grafische interface moeten kunnen worden gedaan.

2.4. Op te leveren producten en diensten

Must have:
Een verzameling klassen/modules waarmee de geanalyseerde relaties en eigenschappen van objecten efficient kunnen worden gedefinieerd. De gebruiker moet met behulp van het systeem in staat zijn op basis van *properties* van spel-objecten *preconditions* te kunnen modelleren op basis waarvan voorwaardelijke *acties* kunnen worden uitgevoerd. Hierbij volstaat een *proof of concept.*
Should have:
Een grafische gebruikers-interface waarmee de relaties en eigenschappen van bestaande spel-objecten kunnen worden aangepast, en nieuwe spel-objecten op basis van bestaande objecten kunnen worden gedefinieerd.
Het ontwerp van de modules zou moeten aansluiten op de bestaande spel-architectuur, en in het bijzonder de engine. De bestaande spel-objecten ("items") zouden moeten worden hergebruikt, en het systeem zou parallel of in het verlengde van de bestaande "controllers" moeten werken, d.m.v. events.
Could have:
Door verschillende acties te combineren, in bijvoorbeeld een keten, zouden nieuwe acties gedefinieerd moeten kunnen worden.
De grafische gebruikers-interface zou door gebruikers die beperkte ervaring hebben met programmeren gebruikt moeten kunnen worden. De gebruiker kan worden geacht te *scripten*, niet te *coden*. Als het voor sommige stukken van het spel, bijvoorbeeld de effecten van acties, nodig zou blijken te zijn om toch echt te programmeren, zou dit buiten ons systeem vallen.
Would like to have:
De aanpassingen aan het spel zouden tijdens het uitvoeren van het spel moeten kunnen worden gemaakt.
De voorwaarden ("preconditions") voor sommige acties zijn niet altijd hard, het systeem zou in staat moeten zijn voorwaarden te negeren, of tijdelijk uit te zetten. Dus als een preconditie er voor de spelers niet meer toe doet, omdat ze bijvoorbeeld de situatie willen uitproberen zonder de preconditie, of als het object/de actor waar de preconditie mee te maken heeft (tijdelijk) niet in het spel aanwezig is, moet het systeem hier flexibel mee om kunnen gaan.

2.5. Eisen en beperkingen

Het project is geslaagd als, met behulp van het op te leveren systeem, gemaakte spel-code door programmeurs goed te hergebruiken is. Het moet gemakkelijk zijn om relaties tussen objecten te veranderen. Spel-objecten moeten gemakkelijk kunnen worden toegevoegd aan-, en verwijderd uit een spel.

3. Aanpak en Tijdsplanning

Het project zal uit vier fasen bestaan:

- In fase 1, "de orientatie", zal (literatuur-)onderzoek worden gedaan naar bestaande oplossingen voor het probleem, en zal waar nodig het bestaande systeem worden bestudeerd om integratie mogelijk te maken.
- In fase 2, "het ontwerp", zal een ontwerp voor het op te leveren systeem worden gemaakt.
- In fase 3, "de implementatie", zal het ontwerp daadwerkelijk worden geïmplementeerd.
- In fase 4, "de conclusie", zal de acceptatie en presentatie van het product, dan wel het rapport omtrent het project centraal staan.

De planning is als volgt:

Week	Fase
1 t/m 3	Orientatie
4 t/m 6	Ontwerp
7 t/m 11	Implementatie
12	Conclusie

4. Projectinrichting

4.1. Organisatie
Er is (nog) geen verdeling gemaakt in taken of verantwoordelijkheden tussen de beide projectleden. De leden hebben dus dezelfde verantwoordelijkheden. Hierdoor wordt van de projectleden verwacht dat zij beide zo veel mogelijk en het liefst volledige kennis hebben van alle aspecten van het project. Daarnaast wordt ook verwacht dat zij zich inzetten voor het project door bijvoorbeeld initiatief te tonen, toegewezen taken op tijd af hebben en goed met elkaar te communiceren.

4.2. Personeel
Voor het project staan 15 studiepunten. Dit staat officieel gelijk aan 15 * 28 = 420 uur werk per persoon, wat verdeeld over twaalf weken weer gelijk staat aan 7 uren tijdsinvestering per werkdag. Nu is het bij projecten zoals deze moeilijk te voorspellen hoeveel tijd er in bijvoorbeeld de implementatie of het literatuuronderzoek zal gaan zitten, maar deze 7 uren zullen wel als richtlijn genomen worden.

De groepsleden worden geacht kennis te hebben van o.a.:
Java
UML (en eventueel andere manieren om objecten en relaties weer te geven)
De verschillende manieren van testen behandeld bij het vak "Software Quality and Testing"

Ook wordt er verwacht dat de groepsleden een duidelijke presentatie kunnen maken (slides) en geven.

Uitzondering op de beschikbaarheid: van 25 april t/m 5 mei zal Sid Mijnders op vakantie zijn, maar er waarschijnlijk voor kunnen zorgen dat hij beschikking heeft over internet, en dus mails kan sturen en ontvangen en onderzoek kan doen.

4.3. Administratieve procedures
...

4.4. Financiering
Er zal in geen enkele fase van het project een uitwisseling van geld plaatsvinden die betrekking heeft op de uitvoer van het project en/of het resultaat. De projectleden worden niet verwacht buitengewone kosten te maken tijdens de uitvoering van het project, en zullen voor het uitvoeren van het project geen vergoeding krijgen van de opdrachtgever.

4.5. Rapportering
Allereerst wordt het (dit) Plan van Aanpak geschreven, en aan alle betrokken partijen ter evaluatie aangeboden, en zal vervolgens doorlopend impliciet dan wel expliciet worden aangepast. D.w.z. op een gegeven moment zullen beslissing die van invloed zijn op de gang van zake van het project, die tegenstrijdig/aanvullend zijn t.o.v. dit document, niet in dit document worden doorgevoerd, maar slechts bestaan als begrip tussen de betrokken partijen. In week 6 zal het ontwerp van het software systeem worden gepresenteerd. Vervolgens zal in week 6 een uitgebreide tussentijdse evaluatie worden georganiseerd om de stand van zaken, en eventuele bijstellingen van de verwachtingen duidelijk te maken. Tijdens de conclusie-fase zal een rapport worden aangeboden en een presentatie worden gehouden.
Tussendoor zal de opdrachtgever geinformeerd worden over significante ontwerp beslissingen.

4.6. Resources
Door de opdrachtgever zullen twee werkplekken inclusief laptop ter beschikking worden gesteld. Tevens kunnen de projectleden gebruik maken van een source-control server (SVN) van de opdrachtgever. De projectleden kunnen ook gebruik maken van werkplekken op de TU wanneer zij daar behoefte aan hebben. Op het moment wordt ook al gebruik gemaakt van een wiki op Google Sites i.c.m. de beta service van Google Docs om een overzicht van de voortgang te behouden en documenten te delen. Eventueel zal er gebruik worden gemaakt van een weblog om de voortgang van het project te documenteren.

5. Kwaliteitsborging

Om de kwaliteit van het eindproduct te waarborgen zal software testing gedurende de ontwerp en implementatie fasen een belangrijke rol spelen. Er zal een teststrategie worden opgesteld en gebruikt om de essentiële onderdelen te testen. Tevens zullen er, voor zover er interfaces gemaakt worden, usability-tests worden uitgevoerd met de gebruikers om de bruikbaarheid van de interface te garanderen.

Tot slot worden enkele bekende risico's bij projecten besproken.

1) Het onderzoek of de implementatie van de projectleden wijkt te veel af van wat de opdrachtgever voor ogen had. Dit kan komen doordat het niet duidelijk genoeg was wat de bedoeling van de opdrachtgever was, en wordt voorkomen door goed contact te houden met de opdrachtgever. Er wordt dus van de opdrachtgever verwacht dat deze altijd open staat voor vragen, en bij tussentijdse reviews duidelijk is over wat er veranderd moet worden.

2) Er zijn weinig harde deadlines. Van de projectleden wordt dus verwacht dat zij zelf de tijd die in het project wordt geïnvesteerd zo structureren dat alles op tijd klaar is en niet een groot deel van het werk aan het eind nog afgeraffeld hoeft te worden. Dit wordt gedaan door bijvoorbeeld een plan zoals in hoofdstuk 3 op te stellen.

Orientatieverslag

Tygron Bachelor Project

Onderzoeksverslag

25 Mei 2009

Tom Lotgering Sid Mijnders

Begeleiders: Hans Geers, Bernard Sodoyer

Versie: 4e

1. Inleiding

In dit document worden enkele onderwerpen beschreven die voor het ontwerp en/of de implementatie van het te maken systeem een belangrijke rol spelen.

Allereerst beschrijven we in sectie 2 een modelleringstechniek voor bedrijfsprocessen, die is onderzocht als kandidaat om enkele grafische elementen van te gebruiken als onderdeel voor een grafische interface voor ons systeem. In sectie 3 beschrijven we ons onderzoek met betrekking tot de *game-engine* van Tygron. In sectie 4 wordt een overzicht gegeven van gangbare methodes om de eigenschappen van objecten te veranderen of objecten toe te voegen. In sectie 5 wordt ons onderzoek besproken met betrekking tot *dynamic controllers*.

2. Visualisering / Modellering

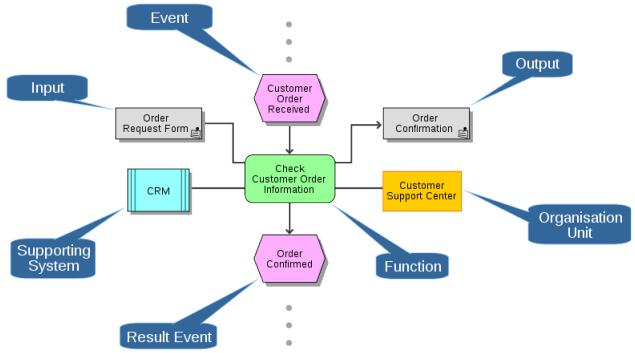
Voor het visualiseren van objecten en relaties, en het aanpassen daarvan in een grafische interface, is het wenselijk dat de visualisatie aansluit op bestaande methoden om een dergelijk systeem te ontwerpen/visualiseren. We hebben een aantal gangbare methodes onderzocht, waarvan Event Driven Process Chains en UML-Activity Diagrams het meest aansluiten op onze behoeften.

Event Driven Process Chains worden gebruikt om bedrijfsprocessen te modelleren. Deze modelleringstechniek wordt al gebruikt door de procesanalist en sluit daardoor goed aan op de kennis van de gebruiker. Daarnaast vormt het een model voor zowel het probleem als voor een deel van de door de gebruiker te ontwerpen oplossing. Het heeft echter een hoge mate van abstractie en weinig aansluiting met object-oriented ontwerpen.

In Event Driven Process Chains worden de volgende elementen gebruikt die relevant zijn voor onze toepassing [2]:

- Event. Events geven gebeurtenissen aan waarna een functie (of bedrijfs-proces) wordt uitgevoerd, en de toestand waarin een proces eindigt. Een Event wordt weergegeven door een hexagon.
- Functies. Een functie beschrijft een transitie tussen toestanden, en eventuele invoer en uitvoer/effecten. Een functie kan een abstractie zijn van een ander EDPC. Een functie kan meerdere inkomende en/of uitgaande toestanden hebben, wat expliciet gemodelleerd kan worden door middel van logische connectoren. Een functie wordt weergegeven door een afgeronde rechthoek.

- Objecten. Een object kan een abstract object zoals informatie, of een fysiek object, zoals bijvoorbeeld de *resource* zand, zijn. Een object wordt weergegeven door een grijze rechthoek.



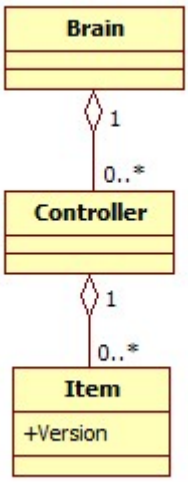
Figuur 1. Een voorbeeld van een EDPC model, met beschrijving van de soorten elementen.

Aangezien EDPC een hoge mate van abstractie heeft, is het nuttig het EDPC te combineren met bepaalde UML diagrammen, zoals het class-diagram. In [1] wordt beschreven hoe die combinatie gemaakt kan worden. Hierdoor ontstaat een diagram dat beter aansluit op een object-oriented implementatie.

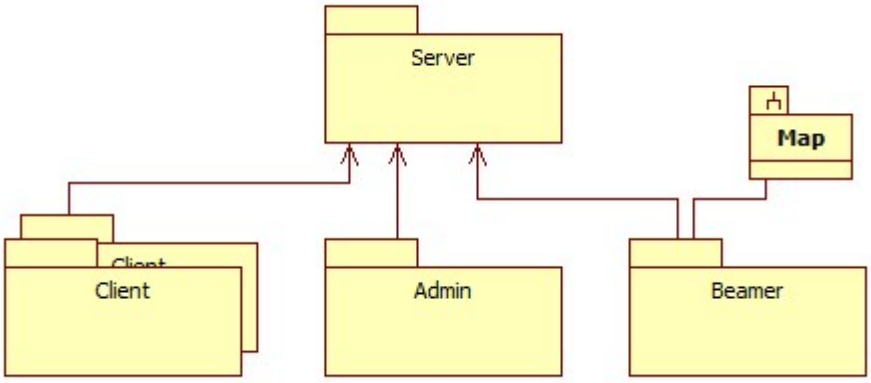
3. Serious Game Engine

We hebben onderzocht of het mogelijk is het te ontwerpen systeem te laten aansluiten op het bestaande systeem. Daarvoor hebben we ons in de huidige architectuur (de Serious Game Engine, SGE) moeten verdiepen. Hiervoor hebben we de documentatie bekeken, maar dit bleek niet voldoende te zijn. We zullen binnenkort ook de code van de engine nader bestuderen om definitieve conclusies over de samenwerking te kunnen trekken.

Globaal werkt de Serious Game Engine als volgt. SGE bestaat uit een library, met daarin een aantal klassen, die vanuit een spel gebruikt kunnen worden. Hieronder vallen klassen die, via de *jMonkey engine*, zaken zoals rendering van 3D objecten, en het lezen van user-input regelen. Ook zijn er klassen voor bijvoorbeeld netwerk communicatie, *serialisation*, en het renderen en gebruik van een 2D interface binnen een 3D renderer viewport.
In multiplayer SGE spellen wordt de input van de gebruiker via het Switchboard in het Brain verwerkt. Het brain is een gecentraliseerde server-klasse die op basis van de gebruikerinput spelacties initieert, en hiervan het resultaat naar de desbetreffende gebruikers stuurt. Er zijn verschillende soorten gebruikers en respectievelijke *client-applications*. Er zijn *administrators*, die in staat zijn alle waarden in een spel te veranderen; spelers, die hierin enigszins beperkt zijn; en er zijn de *beamer* gebruikers/ toeschouwers, waarvan de applicatie *read-only* werkt. De functionaliteit van het brain is *hard coded*.



Figuur 2. Globale structuur SGE objecten, controllers.



Figuur 3. Globale structuur multiplayer spel.

In SGE erft ieder abstract/reëel spel-object van de Item klasse. Een Item fungeert in principe als data-container met *public properties*, maar bevat tevens "properties" met geinfereerde waardes. De *Fields* kunnen annotations hebben: XMLValue om aan te geven dat het veld kan worden opgeslagen en HTML, Editable voor bepaalde tools. Iedere Item instantie heeft een uniek ID. Alle referenties binnen het spel gaan door middel van het ID, er wordt alleen een (echte) referentie naar het object bijgehouden in de Controller die het object beheert.

Voor iedere soort Item bestaat een Controller-klasse die dat type Item beheert. De controller bevat onder meer een lijst van de instanties van die klasse. De controllers luisteren naar events vanuit het "Brain" door middel van de Switchboard klasse. Controllers hebben ook (class) annotations: UpdateEvent definieert wat voor Event de Controller genereert. ItemClasses bepaalt wat voor items de controller gebruikt. Subscriptions bepaalt of het item relevant is voor clients dan wel administrators, en dus voor welke gebruikers-groep het item zichtbaar zal zijn.

4. Dynamic Items

4.1. Adaptive Object Model

4.1.1. Algemene beschrijving

Het Adaptive Object Model (AOM) is een model dat de laatste jaren steeds meer gebruikt wordt. Vaak in "business systems" die producten moeten managen en makkelijk uitgebreid moeten kunnen worden met nieuwe producten. Er worden verschillende namen uitwisselbaar voor gebruikt, waaronder:

- Dynamic Object Model (DOM)
- Reflective Architecture
- Meta Architecture

Volgens [4] is dit niet helemaal juist en is een AOM/DOM een bepaald type reflective architecture. Dit is waar wij ook van uitgaan.

Het AOM kan gedefinieerd worden als een systeem dat klassen, attributen, relaties en gedrag (behavior) representeert als metadata. Er worden dus, als je het vergelijkt met een "normaal" systeem, elementen uit de code naar buiten gehaald en opgeslagen als data. Hierbij kan gedacht worden aan bijvoorbeeld namen van klassen en methodes. Dit leidt ertoe dat AOMs twee kenmerken bezitten die over het algemeen als nuttig worden gezien en waar volgens sommige bronnen de laatste jaren steeds meer vraag naar is, namelijk flexibiliteit en de mogelijkheid om het model tijdens runtime aan te passen. Dit vereist wat nadere uitleg.

Het systeem is flexibel omdat het de mogelijkheid geeft om het systeem aan te passen "zonder te programmeren". Want zoals eerder gezegd wordt het hele systeem van het bedrijf, dus de objecten, regels en het gedrag, vastgelegd als data. Deze data wordt vervolgens door het AOM gebruikt om invulling te geven aan gegeneraliseerde klassen en zo het systeem op te bouwen. Dit wordt in het algemeen de interpretatie van een systeem genoemd [4][7]. Het enige wat dus veranderd hoeft te worden om het systeem aan te passen is data, en niet de code van het programma. En dit is ook de reden dat deze aanpassingen tijdens runtime kunnen gebeuren, want als de code niet wordt veranderd hoeft er ook niets opnieuw gecompileerd te worden.

Ook voor ons ziet een dergelijk model er aantrekkelijk uit om te gebruiken voor deze opdracht. Aan ons is namelijk gevraagd om een 'modulair' systeem te ontwikkelen waarbij het eventueel mogelijk is om tijdens runtime aanpassingen te maken (bijvoorbeeld door een speler in de administrator rol). Ook staat in ons PvA dat de gebruiker van ons systeem niet noodzakelijk hoeft te kunnen programmeren. En dit zijn precies de kenmerken van een AOM.

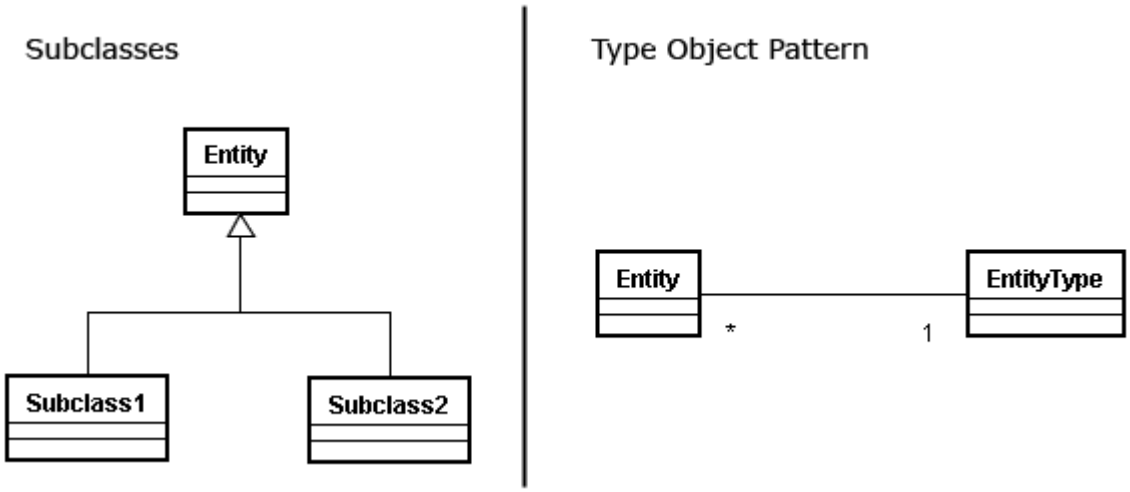
4.1.2. Technologische werking

In deze sectie worden de technische aspecten van het Adaptive Object Model besproken, dus aan welke eisen de klassen en opbouw van het systeem moeten voldoen om het een AOM te noemen. Het AOM is opgebouwd uit verscheidene kleinere patterns.

Type Object Pattern

Het Type Object Pattern kijkt of het mogelijk is om verschillende objecten (in artikelen en verder vaak Entities genoemd) samen te vatten in types. In dit deel van het AOM (het Type Square gedeelte, zie onder) worden hier de meer "tastbare" objecten mee bedoeld, in tegenstelling tot bijvoorbeeld actie-objecten, maar het kan eigenlijk overal op gebruikt worden. Waar mogelijk kan in plaats van subklassen van een Entity een EntityType-klasse worden gebruikt en zo een overvloed aan subklassen worden vermeden. Hierbij kan bijvoorbeeld gelet worden op overeenkomsten in de attributen van de subklassen. Daarnaast zorgt dit pattern ervoor dat er op een dynamische manier, zonder te

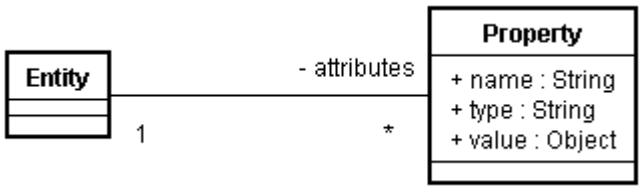
programmeren, nieuwe Entities voor het systeem kunnen worden gemaakt (om code voor nieuwe subklassen te schrijven moet het model aangepast worden, om een instantie van een EntityType-klasse te maken niet).



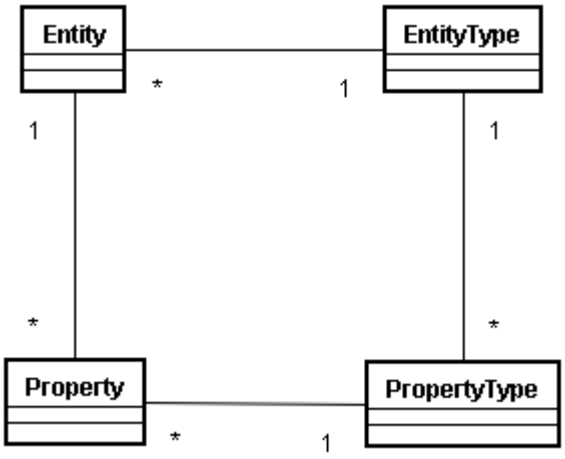
Als we als voorbeeld een systeem nemen waarin producten moeten worden beheerd vindt de volgende verandering plaats:
In plaats van
een Product-klasse met verschillende subclasses (voor ieder product)
komt er
een Product-klasse die een ProductType-klasse heeft.

Property Pattern

Het Type Object pattern heeft toch nog een duidelijk nadeel in vergelijking met subklassen. Subklassen van een klasse kunnen namelijk t.o.v. elkaar andere attributen hebben. Dit is een mogelijkheid die wegvalt als je de subklassen vervangt door instanties van één klasse. Om toch deze functionaliteit te behouden zonder dat het ten koste gaat van de dynamische mogelijkheden van het Type Object Pattern wordt het Property Pattern gebruikt. Dit gebeurt door de attributen niet vast in de klasse te coderen, maar in plaats daarvan elke Entity een lijst van Properties bij te laten houden (die de attributen moeten voorstellen). Deze Properties hebben meestal naam, type en value attributen en kunnen dynamisch toegevoegd en verwijderd worden.



Vervolgens wordt in veel AOMs nog een keer het Type Object Pattern gebruikt op Property. Deze wordt dus net als bij Entity opgesplitst in de klassen Property en PropertyType. De model wat daarmee wordt verkregen wordt een Type Square genoemd.



Dit is een stap die de consistentie van het systeem ten goede komt. EntityTypes kunnen op deze manier de PropertyTypes specifieren die hun Entity kan hebben.

De hierboven beschreven TypeSquare, gevormd met behulp van de twee patterns, vormt de basis van het Adaptive Object Model. Een ander pattern dat veel gebruikt wordt om het model uit te breiden is het Strategy pattern. Dit is een manier om dynamisch algoritmes toe te passen en zo de "behavior" van het systeem te kunnen veranderen. In het Strategy pattern wordt gebruik gemaakt van Rule objecten met subklassen PrimitiveRule en CompositeRule (waarvan de namen voor zich spreken). Vaak worden deze Rules gelinkt aan de EntityType klasse, waar ze de methodes implementeren. Dit zorgt er bijvoorbeeld voor dat er tijdens de uitvoer van een spel binnen de klasse niet telkens op states gecheckt hoeft te worden om daarop zijn gedrag aan te passen, want het gedrag is al dynamisch aan de goede situatie aangepast. In ons systeem kan dit bijvoorbeeld gebruikt worden om acties te representeren die door een bepaald soort object kunnen worden uitgevoerd.

4.2. Reflection

We hebben de mogelijkheden, en API, van Java-reflectie onderzocht. Reflectie zou de basis kunnen vormen van een dynamisch object model, en zou gebruikt kunnen worden om bestaande klassen naar een nieuw systeem over te zetten, en om attributes te gebruiken.

Met behulp van reflectie kan, onder andere, tijdens runtime het type van een object, en de methodes, fields en attributes van klassen opgevraagd worden. Het is ook mogelijk om dynamisch methodes aan te roepen, of fields te wijzigen.

Reflectie is niet van groot belang als het bovenstaande property pattern wordt gebruikt, omdat met dat pattern de metadata die door reflectie wordt gebruikt, ook expliciet in het object-model aanwezig is. Met behulp van reflectie kan echter wel een bestaande klasse worden geanalyseerd, waardoor er vervolgens een dynamische klasse op basis van het property pattern van kan worden gemaakt.

4.3. Code Injection

Code injection is nuttig om functionaliteit toe te voegen of te veranderen aan klassen. Het is mogelijk nieuwe klasses te maken, of een bepaalde klasse tijdens runtime aan te passen, zolang die klasse op dat moment niet al geladen is. Code injection zou het gebruikt kunnen worden om properties aan een object toe te voegen. In combinatie met

een property pattern is het echter niet zinvol dit te gebruiken voor data-klassen, maar het zou echter wel nuttig kunnen blijken te zijn voor bepaalde controllers.

Java heeft beperkte standaard ondersteuning voor het aanmaken en injecteren van bytecode, maar er zijn libraries beschikbaar waarmee deze functionaliteit toch mogelijk wordt. Een voorbeeld van een dergelijke library is Javassist.

5. Dynamic Controllers

We hebben gezocht naar bestaande methoden om dynamisch de functionaliteit van de Controllers te implementeren, maar hebben geen generieke oplossingen gevonden. Hiervoor zal verder onderzoek, of een ad hoc oplossing noodzakelijk zijn.

6. Referenties

1. Loos, P. & Allweyer, T. (1998) *Process Orientation and Object-Orientation - An Approach for Integrating UML and Event-Driven Process Chains (EPC)* http://isym.bwl.uni-mainz.de/downloads/Publikationen/Loos_Allweyer_1998_Process_Orientation_OO_Integrating_UML_and_EPC.pdf
2. Wikipedia.org, *Event-driven Process Chain* http://en.wikipedia.org/wiki/Event-driven_process_chain
3. Wikipedia.org, *Activity Diagram* http://en.wikipedia.org/wiki/Activity_diagram
4. Yoder, J.W. & Johnson, R. *The Adaptive Object-Model Architectural Style* http://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.pdf
5. Balaguer, F. & Yoder, J.W. (2001) *Adaptive Object-Model Architecture "How to Build Systems that can Dynamically Adapt to new Business Requirements"* http://www.metamodelling.com/OOPSLA2001/AOMoopsla2001Tutorial.pdf
6. Johnson, R.E. *Dynamic Object Model* http://st-www.cs.uiuc.edu/users/johnson/papers/dom/DynamicObjectModel.pdf
7. Adaptiveobjectmodel.com (site by Yoder, J.W.) http://www.adaptiveobjectmodel.com/
8. Sun.com, *Description of Package java.lang.reflect* http://java.sun.com/javase/6/docs/api/java/lang/reflect/package-summary.html
9. http://www.csg.is.titech.ac.jp/~chiba/javassist/

RAD-document met UML

Tygron Bachelor Project
Requirements Analysis Document

30 Juni 2009

Sid Mijnders Tom Lotgering

Begeleiders: Hans Geers, Bernard Sodoyer
Versie: 5

Inhoudsopgave

1. Inleiding

Tygron is een bedrijf dat zich specialiseert in het maken van "serious games". Deze spellen kunnen de spelers inzicht geven in processen en relaties tussen actoren die zijn gericht op stedelijke gebiedsontwikkeling. Een serious game wordt als maatwerk gemaakt door de situatie van een klant te analyseren en te verwerken in een bestaand spel. Om dit te kunnen doen moeten de elementen, de processen en de afhankelijkheden hiertussen uit de echte wereld goed overeenkomen met die in de spelsituatie. Hiervoor is momenteel een ad hoc oplossing, die geschikt is voor een enkel spel, maar die zich niet leent voor hergebruik, of voor aanpassing van een spel (zie beschrijving van de huidige situatie) aan afwijkende situaties tussen klanten, zonder het spel grondig te veranderen. Om dit op te lossen moet een systeem gebouwd worden dat naast het bestaande systeem kan worden gebruikt. Dit document beschrijft eerst het huidige systeem en vervolgens aan welke eisen het nieuwe systeem moet voldoen.

2. Huidige Situatie

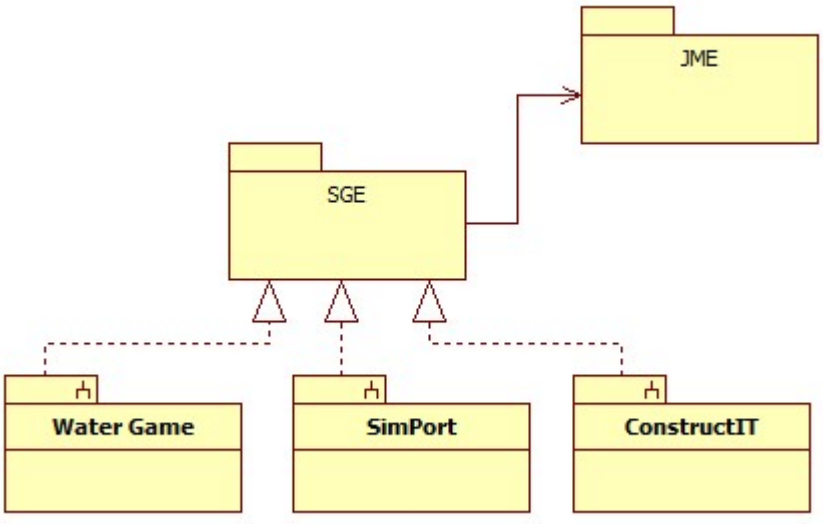
In dit hoofdstuk wordt de huidige situatie binnen het bedrijf besproken. Eerst wordt een beschrijving gegeven van de game engine die door Tygron gebruikt wordt, de Serious Game Engine (SGE), en op welke manier spellen van deze engine gebruik kunnen maken. Vervolgens wordt in paragraaf 2.19 aan de hand van een voorbeeldspel de relatie met de engine verder uitgewerkt. In paragraaf 2.2 staat kort de gang van zaken binnen het bedrijf beschreven omtrent het maken van een spel, van de analyse tot het testen, en in 2.3 staat wat hier volgens het bedrijf problematisch aan is of wat er beter/ makkelijker kan.

2.1. Beschrijving Serious Game Engine

De opdrachtgever Tygron maakt voor haar spellen gebruik van een intern ontwikkelde game engine, genaamd Serious Game Engine, ofwel SGE. Hieronder volgt een (uitputtende) beschrijving van deze engine, aangezien dit deel uitmaakt van het huidige systeem en derhalve ter volledigheid moet worden beschreven, én omdat ons te ontwerpen systeem hier *eventueel* mee moet kunnen samenwerken.

2.1.1. Globale architectuur

Het doel van de Serious Game Engine is om te zorgen dat de programmeurs van een spel zich uitsluitend op spel-specifieke functionaliteit hoeven te richten. Globaal werkt de SGE als volgt. SGE bestaat uit een Java library, met daarin een aantal klassen, die vanuit een spel gebruikt kunnen worden. Hieronder vallen klassen voor rendering van 3D objecten of het lezen van user-input regelen. Ook zijn er klassen voor bijvoorbeeld netwerk communicatie, *serialization*, en het renderen en gebruik van een 2D interface binnen een 3D renderer viewport.



Figuur 1. *Globale architectuur van de SGE engine, en spellen die daarvan gebruik maken.*

2.1.2. JME, FengGUI, openGL

De SGE gebruikt de jMonkey Engine (JME). JME is een game-engine voor Java, gebaseerd op openGL. Met behulp van JME kunnen bijvoorbeeld 3D objecten worden gerenderd en kan input van de gebruiker worden gelezen. Hierbij wordt gebruik gemaakt van openGL, en dus niet via de standaard Java Foundation Classes (JFC) (waaronder AWT en Swing) die normaal worden gebruikt voor het tekenen van graphics, en user-interactie in Java.

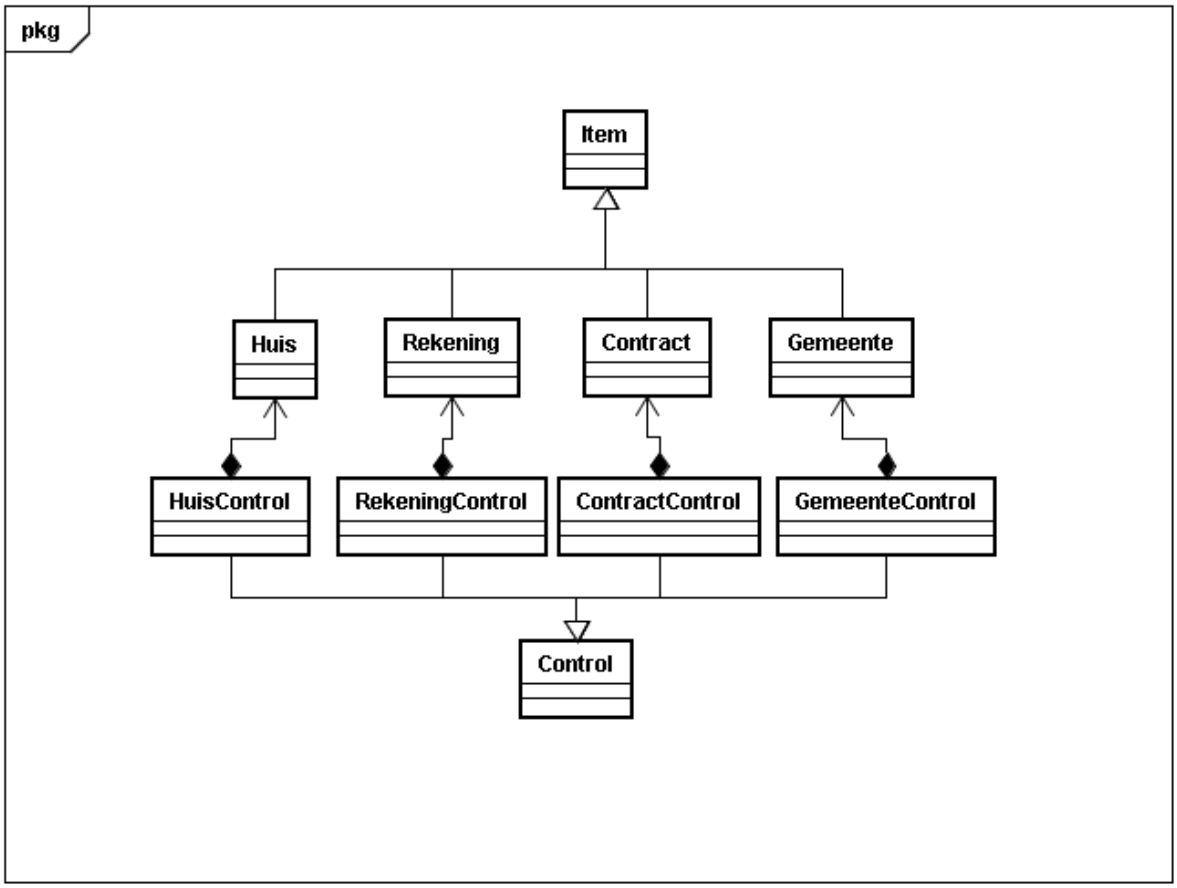
Zoals de meeste game engines werkt JME met behulp van een "game-loop": tijdens het uitvoeren van een spel wordt het beeld per "frame" geupdated. Dit gebeurt zo'n 60 maal per seconde. De game-loop herhaalt voor het tekenen van ieder frame een aantal stappen: eerst wordt een update methode aangeroepen en vervolgens een render methode. In de render methode wordt op basis van de tekenbare spel-objecten een beeld getekend, door polygon-informatie via openGL naar de GPU (Graphics Processing Unit) te sturen. In de update methode worden de spel-objecten geupdated, en wordt de toestand van het toetsenbord en de muis uitgelezen. Binnen een spel werken events dus niet als het "*event*" system zoals dat bij Java GUIs bekend is, waarbij klikken met de muis en gebruik van toetsenbord etcetera resulteert in het aanroepen van bepaalde *ActionListeners*.

Voor het tekenen van GUIs binnen een spel (zoals knoppen in een menu en dergelijke) wordt binnen SGE FengGUI gebruikt. Deze FengGUI library bied functionaliteit die vergelijkbaar is met de functionaliteit van AWT of Swing, zoals het plaatsen, tekenen en gebruik van *Buttons*, *TextBoxes*, etcetera. FengGUI implementeert een "*event*" systeem op basis van het *pollen* van de muis en toetsenbord van OpenGL. Dit systeem werkt vergelijkbaar met het "*event*" systeem van AWT/Swing, maar gebruikt hiervoor andere *interfaces* (in de Java-OOP zin van het woord).

2.1.3. Items

Een spel bevat verscheidene spel-objecten. Een spel-object kan van alles zijn, bijvoorbeeld een fysiek (al is het natuurlijk virtueel) object zoals een huis, een aannemer of een contract, of een meer abstract object zoals een bankrekening of een deelgemeente. Binnen SGE *extend* ieder spel-object de Item klasse. De Item klasse bevat twee velden, namelijk het ID en het versienummer. Voor ieder Item is het ID uniek. Het ID wordt elders in het spel gebruikt als referentie naar het item. Het

versienummer wordt verhoogd elke keer dat een attribuut van het item gewijzigd wordt. Zo kan worden bepaald welke items zijn veranderd, wat van belang is voor de netwerk-communicatie (zie 2.1.7).
De subklassen van Item zijn in principe uitsluitend dataklassen, en bevatten dus geen functionaliteit (in de vorm van methoden, behalve getters en setters). Er zijn echter een paar uitzonderingen waarbij een subklasse van Item een methode getX heeft terwijl het de desbetreffende waarde niet bevat, en het die waarde via een omweg (bijvoorbeeld via een ander *veld*) opvraagt en teruggeeft.
We noemen deze spel-objecten, instanties van de subklassen van Item, in het vervolg items.



Figuur 2. *Een voorbeeld van een aantal spel-objecten, in een op SGE gebaseerd spel.*

2.1.4. Controllers

Voor iedere subklasse 'X' van Item bestaat een subklasse XControl welke extendt van Control<X>. Deze worden de controllers genoemd. Alle items worden beheerd door hun bijbehorende controllers. De controllers bewerken de gegevens van de items en zijn verantwoordelijk voor het ophogen van het versienummer van de items.

2.1.5. Events

Er zijn verschillende interpretaties mogelijk van het woord "event", afhankelijk van de context. Binnen Java GUIs zijn events bekend als datgene waarop implementaties van Listeners kunnen reageren, zoals bijvoorbeeld een MouseDown, WindowResized, of ButtonPressed event. Binnen FengGUI, zoals dat in SGE gebruikt wordt, bestaan ook dit soort events die op acties van de gebruiker binnen de GUI reageren.
Daarnaast worden door het spel zelf ook andere soorten events gebruikt. Zo kan er

bijvoorbeeld een event worden afgevuurd als het bouwen van een huis is voltooid, een aanvraag is goedgekeurd, een gemeenteraadsverkiezing wordt gehouden, etcetera. Er zijn echter ook events binnen SGE. Zo wordt er binnen de server een event afgevuurd als een wijziging heeft plaatsgevonden waar een client weet van moet hebben, waardoor de game-clients worden geupdated (zie 2.1.7). Het is belangrijk om in het vervolg van deze tekst deze mogelijke interpretaties van "event" in acht te nemen! Deze paragraaf gaat verder met het gebruik van events van de SGE. Dit is ook het type events wat relevant voor ons is, omdat ons systeem eventueel zou moeten aansluiten op de SGE, niet op FengGUI.

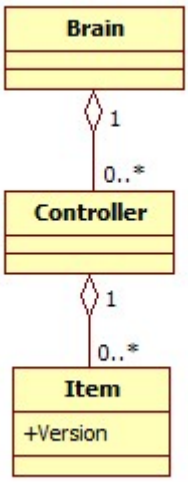
Er zijn vanuit het perspectief van de engine gezien twee soorten events; clientside events en serverside events. De klasse die aan de clientkant centraal staat bij de events is de EventManager. De EventManager, een static klasse, beheert de listeners en vuurt events af. Een klasse kan alleen een event afvuren door in de EventManager fireEvent() aan te roepen en het event als argument mee te geven. In het event staan in ieder geval het type event en de source (het object dat fireEvent aanroept), en optioneel een argument. De serverkant heeft de Brain-klasse met ongeveer dezelfde functionaliteit m.b.t. events.

Voor het afhandelen van events worden er in het begin het van implementeren van een spel twee klassen gemaakt. LogicManager zorgt voor het afhandelen, dus ontvangen en doorsturen, van events aan de clientkant en SwitchBoard vertaalt client-events naar server-events, welke verder door het Brain kunnen worden afgehandeld. Beide klassen worden in de EventManager toegevoegd als listener voor alle events. Deze klassen worden "generic listeners" genoemd, omdat ze alle events ontvangen. Een klasse kan ook als "specific listener" worden toegevoegd door het type event op te geven als hij zich registreert in de EventManager. Als een event wordt afgevuurd worden eerst alle generic listeners en daarna de specific listeners afgegaan.

De events zelf worden bijgehouden in enums. Elke enum staat voor een bepaald type event en implementen allemaal de EventTypeEnum klasse uit de SGE. Bij deze eventtypes/enums kan bijvoorbeeld gedacht worden aan een ClientEventType-enum of ViewEventType-enum. Om een event te maken moet deze handmatig aan een enum worden toegevoegd. Sommige events (bijvoorbeeld de view events) zijn al gedefinieerd in de SGE, omdat deze hier moeten worden afgehandeld.

2.1.6. Brain

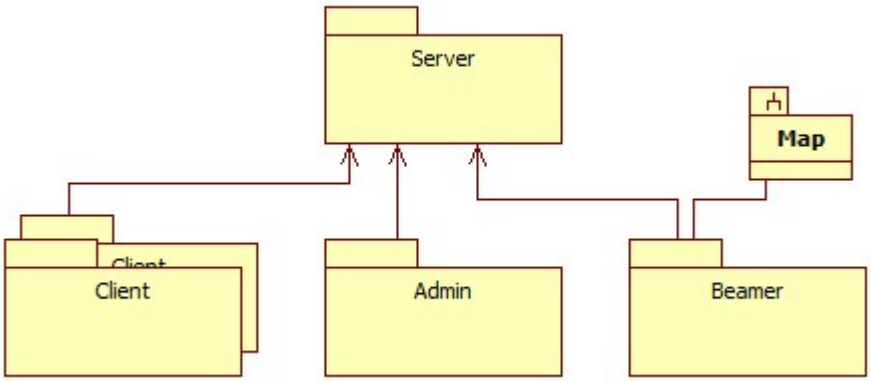
De Brain-klasse is de belangrijkste controlklasse van het spel. Alle controllers van items zijn gelinkt aan het brain, en de enige manier om de waardes van een item te updaten is via het brain. Brain is een klasse op de server die zowel lokaal (in single player games) als op een aparte computer kan draaien. Dit heeft tot gevolg dat alle manipulatie van items op de server gebeurt. Het is dus niet mogelijk om een item op een client te updaten en vervolgens naar de server te sturen. Voor verdere informatie over de client/ server implementatie, zie paragraaf 2.1.7.



Figuur 3. *Samenwerking van het Brain, de controllers en items*

2.1.7. Client/Server implementation

SGE is ontwikkeld voor *multiplayer* spellen die door meerdere gebruikers, vanaf aparte computers, gespeeld kunnen worden. Het heeft hiervoor een client-server architectuur, waarbij meerdere clients verbonden zijn met de gameserver. Clients kunnen uitsluitend met de server verbinden, dus een server kan geen connectie met een client starten, en een client kan niet met een andere client verbinden. De communicatie tussen client en server maakt gebruik van Remote Method Invocation (RMI). Deze communicatie gaat *uitsluitend* in de vorm van "events". De handelingen van een speler genereren een "client-event" dat naar de server wordt gestuurd, waar het in de Switchboard klasse wordt verwerkt tot een "server-event". Server-events worden gebruikt door de server om spelfunctionaliteit uit te voeren. Vervolgens worden door middel van een "update-event" gegevens teruggestuurd naar de clients. Er zijn verschillende soorten clients voor een spel. De meeste clients zijn de gewone game-clients van normale spelers. Er is echter ook een Administrator die het spel beheert. Hij gebruikt een aparte Admin client. Er bestaat ook een Beamer client, die alleen het spel weer kan geven en waarmee verder geen interactie van de gebruiker met het spel mogelijk is. De server beslist op basis van het type client welke update-events van belang zijn voor de client.



Figuur 4. *Multiplayer architectuur.*

2.1.8. Annotations

Binnen Java kan code met behulp van zogeheten Annotations worden aangevuld met meta-data. Annotations zijn een soort van "commentaar" dat door een parser & compiler begrijpbaar is. Annotations kunnen worden toegevoegd aan onder andere klassen, velden en methodes. Via de *reflection* API van Java kunnen deze Annotations tijdens *runtime* worden opgevraagd. Er zijn standaard Annotations (zoals @ Override, of @ Author) maar er kunnen ook nieuwe typen Annotations door een programmeur worden gedefinieerd.
Binnen SGE wordt gebruik gemaakt van deze Annotations. Items hebben voor sommige velden de volgende annotations: XMLValue om aan te geven dat het veld kan worden opgeslagen en HTML, Editable voor bepaalde tools. Controllers hebben ook (class) annotations: UpdateEvent definieert wat voor Event de Controller genereert. ItemClasses bepaalt wat voor items de controller gebruikt. Subscriptions bepaalt of het item relevant is voor clients dan wel administrators, en dus voor welke gebruikers-groep het item zichtbaar zal zijn.

2.1.9 Voorbeeld: Watergame

Een voorbeeld van een spel dat met SGE gemaakt is, en waarvan de spel-objecten door ons te ontwerpen systeem beschreven zouden moeten kunnen worden, is het Watergame. In bijlage A staat een voorbeeld van de analyse van Watergame voor Oost-Tiel. In bijlage B staat een voorbeeld van een event uit Watergame, en handelingen van spelers en acties die daaruit volgen.

Ter illustratie van de voorgaande behandeling van items en Watergame, is dit de volledige lijst van spel-objecten in Watergame:

BeamSetting, Building, CivilBuilding, CivilType, ClientWord, Cost, Economy, FinanceSetting, FlightPoint, Function, FunctionalTerrain, Indicator, IndicatorGameLog, Measure, MeasureSet, PublicBuilding, PublicType, ServerWord, Strategy, Subsidy, Terrain, Type, WGActor, WGBlock, WGEffect, WGFont, WGMessage, WGModel, WaterSetting, WaterTerrain.

2.2. Beschrijving aanmaken spel

In deze sectie wordt het huidige proces bij Tygron omtrent het maken van een spel beschreven.

Tygron maakt voor klanten spellen op maat. Voor sommige klanten worden geheel nieuwe spellen gemaakt, en voor sommige klanten wordt een bestaand spel aangepast. De klant geeft over het algemeen zelf aan welk van de twee aan de orde is. Als de klant een bestaand spel kent en "mee wil spelen" wordt het aangepast, en anders wordt er een nieuw spel ontworpen.
Als het bedrijf een opdracht krijgt moet eerst de situatie van de klant geanalyseerd worden. Onder situatie vallen bijvoorbeeld de actoren die een rol spelen en de problemen die deze actoren met elkaar of met hun omgeving (geografisch gezien) hebben. Dit wordt door een proces analist gedaan. De proces analist onderzoekt de situatie van de klant en brengt hierover vervolgens rapport uit aan de opdrachtgever. Wat in een dergelijk rapport staat is erg afhankelijk van de wensen van de klant. Als de klant bijvoorbeeld aanpassing van een bestaand spel zoals het "Watergame" wil, wordt vooral gekeken naar welke actoren (met betrekking tot het onderwerp van het spel) voor die klant van belang zijn en wat voor problemen en belangen zij hebben. Dit gebeurt bijvoorbeeld door interviews te houden en te vragen met welke actoren wordt samengewerkt. Hieruit worden ook de relaties tussen die actoren, en de "organisatorische processen" waarbinnen die samenwerking valt, duidelijk. Voor

bijzonder complexe (bedrijfs-)processen kan een diagram worden gemaakt dat het proces beschrijft, bijvoorbeeld als state-machine of als EDPC diagram. Er wordt echter al snel begonnen met het ontwerp van (de aanpassingen aan) een spel dat hierop aansluit, en het uitvoerig in kaart brengen van de situatie van de klant is geen doel op zich.

Vervolgens wordt er in de ontwikkelomgeving, ook voor het maken van een aangepast spel, een heel nieuw leeg project aangemaakt. Hier worden, naast het programmeren van nieuwe objecten en relaties, van eerdere projecten/spellen veel stukken code naartoe gekopieerd. Er kan dan bijvoorbeeld gedacht worden aan een klasse als Building of actoren die in meerdere spellen terugkomen. Op deze manier wordt een spel verkregen dat aangepast is aan de klant.

Het testen van het spel gebeurt als volgt. Eerst wordt door de programmeurs zelf getest wat ze programmeren. Vervolgens wordt intern, door iedereen die aan het spel meewerkt, getest. Dit gebeurt door voor belangrijke stukken code, bijvoorbeeld algoritmes, unit tests te maken of door scenario's te doorlopen. Tot slot worden er testsessies georganiseerd binnen het bedrijf of met hulp van ervaren testers als een soort closed beta.

2.3. Waarom is de huidige situatie problematisch?

Het maken van kleine aanpassingen aan een spel, voor een andere klant, kost relatief veel moeite. Er wordt een geheel nieuw spel voor aangemaakt en het spel wordt gekopieerd. Vervolgens moeten delen van de code van het spel worden herschreven, om bijvoorbeeld actoren toe te voegen of te verwijderen, of om andere items toe te voegen.

Dit proces is arbeidsintensief; er is een programmeur voor nodig ook wanneer er geen nieuwe functionaliteit wordt toegevoegd, maar alleen bestaande functionaliteit wordt aangepast. Onderhoud is lastig, omdat er veel replicatie van code plaats vindt.

3. Voorgesteld systeem

3.1. Overzicht

3.1.1. Functioneel overzicht (toegevoegde waarde van het systeem)

Tygron heeft de opdracht gegeven om een systeem te maken waarmee objecten en relaties uit situaties in de echte wereld op een modulaire, overzichtelijke manier in een spel geïmplementeerd kunnen worden. Modulair in de zin dat het mogelijk moet zijn om gemakkelijk (kleine) aanpassingen te maken aan deze objecten, en vooral aan de relaties, en dat er makkelijk nieuwe objecten en relaties toegevoegd en verwijderd kunnen worden. De situaties waar Tygron mee te maken krijgt hebben voornamelijk te maken met stedelijke gebiedsontwikkeling, maar ons systeem zou in principe niet beperkt moeten zijn tot het implementeren van een bepaald type situatie. Daarbij komt dat het, naast misschien een paar standaardklassen, niet te voorzien is wat voor soort objecten de opdrachtgever tegenkomt bij de analyse van een klant. Om deze reden stellen wij een systeem voor dat zo generiek is dat eigenlijk alle mogelijke situaties die bedacht kunnen worden ermee gemodelleerd moeten kunnen worden. Om een dergelijk systeem te maken is logischerwijs onderzoek gedaan naar modellen voor systemen die de gebruiker deze mogelijkheid bieden (dus het maken van objecten en relaties hiertussen). Op basis van dit onderzoek hebben wij besloten het systeem te baseren op het Adaptive Object Model (AOM), dat in het onderzoeksrapport beschreven staat. Hoe dit systeem werkt wordt verder beschreven in het Technisch overzicht (3.1.2), de Systeemmodellen (3.5), en het Design Document.

Bij deze functionaliteit kunnen wij ons enkele vragen stellen:

1. Kunnen wij het waarmaken dat álle denkbare situaties met ons systeem gemaakt kunnen worden?
2. Is het systeem dat wij ontwerpen echt nuttig voor Tygron?

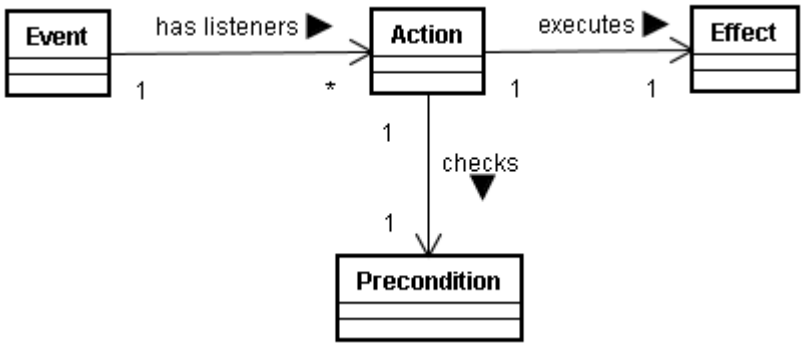
Het antwoord op vraag 1 is kort; Nee, dit kunnen wij niet waarmaken, om de simpele reden dat wij als mensen geen rekening kunnen houden met alle mogelijke situaties. Wij proberen hier wel rekening mee te houden door het systeem zo generiek mogelijk te houden.

Ook het nut van het nieuwe systeem voor het bedrijf moet nog blijken. Natuurlijk biedt ons systeem enkele mogelijkheden die er eerder niet waren. Zo biedt het AOM inderdaad de mogelijkheid om makkelijk en dynamisch (evt. tijdens het spelen) aanpassingen te maken, en zorgt de GUI voor overzicht over het model. Maar het blijft de vraag of de gebruikers, de developers, dit inderdaad makkelijker vinden werken dan het huidige systeem. Het is dan ook de bedoeling dat wij voor een zogenaamd "proof-of-concept" voor ons systeem zorgen, bijvoorbeeld door een (of enkele) scenario('s) te demonstreren, en het bedrijf er vervolgens mee gaat experimenteren en zelf tot een conclusie komt.

3.1.2. Technisch overzicht

Een spel dat met ons systeem wordt gemaakt bestaat uit een *project*, die de algemene informatie van het spel bevat (bijvoorbeeld de locatie van klassen/modules). Binnen een project kunnen *modules* gemaakt worden. Deze modules kunnen gebruikt worden om structuur binnen een project aan te geven, maar krijgen eventueel ook de functionaliteit dat ze tijdens het spelen van een spel in- en uitgeschakeld kunnen worden. Het is de bedoeling dat de gebruiker de modules gebruikt om elementen in te zetten die volgens hem/haar een afgebakend deel van het spel implementeren. Welke elementen er precies bij elkaar in een module horen te komen laten wij aan de gebruiker over, maar er kan hierbij bijvoorbeeld gedacht worden aan een klasse, compleet met bijbehorende acties. Ook is het mogelijk modules van andere projecten in een nieuw project te importeren.

Het systeem maakt qua objecten onderscheid tussen klassen en acties. Klassen en acties kunnen tijdens het spelen van het spel "geïnstantieerd" worden tot objecten en actie-instanties, waarbij waarden aan de attributen worden toegekend. Aan de klassen kunnen eigenschappen, verder *properties* genoemd, worden toegevoegd, die de attributen van de klasse voorstellen. Properties hebben allemaal een type, wat een bestaande Java klasse kan zijn, of een klasse die eerder met ons systeem is gemaakt. Op deze manier kunnen meerdere klassen aan elkaar gelinkt worden. Een klasse heeft ook een standaard attribuut *parent*, waarmee aangegeven kan worden dat het een subklasse van een andere klasse is. Deze parent moet een klasse zijn die met ons systeem is gemaakt. De acties bestaan uit drie elementen die belangrijk zijn voor de functionaliteit. Elke actie is als listener gekoppeld aan een *event*. Als dit event wordt afgevuurd wordt de actie uitgevoerd. Ten tweede werken acties met *precondities*. Een actie kan pas worden uitgevoerd als aan zijn preconditie voldaan is (hij heeft er dus maar één). Deze preconditie kan weer bestaan uit meerdere precondities die door boolean operators aan elkaar gekoppeld zijn. Tot slot is er de "body" van de actie, het *effect*. Deze bevat de code die moet worden uitgevoerd als een event wordt afgevuurd en aan de preconditie is voldaan.



Figuur 5 *Overzicht actions*

Het geheel wordt overzien door een manager, die ervoor zorgt dat de voortgang (van het modelleren, dus niet van het spelen van een spel) wordt weggeschreven naar XML-bestanden en hier ook weer uit kan worden ingelezen.

3.2. Functionele eisen

In deze paragraaf worden eerst de gebruikers van ons systeem en de meegeleverde klassen beschreven. Daarna volgen de functionele eisen in de vorm van een MoSCoW document, met bij elke eis voor welke gebruiker deze van belang is.

Gebruikers:

- **(Game-)Developer**: De persoon die ons systeem gebruikt, ofwel handmatig (door te programmeren) ofwel met de meegeleverde GUI de klassen, acties en relaties hiertussen zal definiëren. De developer gebruikt alleen de functionaliteit van ons systeem die de GUI beschikbaar stelt.
- **Programmeur**: De programmeur werkt buiten de scope van ons systeem. Hij programmeert bijvoorbeeld, aan de hand van een door ons geleverde interface, de effecten van actions en maakt aanpassingen aan de engine/het spel/etc. De programmeur is integreert ons systeem met de spellen die worden gemaakt. Een deel van ons systeem kan alleen worden uitgevoerd vanuit de context van een spel. De programmeur is verantwoordelijk voor het opzetten van, en gebruik vanuit, die context. De interactie van de programmeur met ons systeem door middel van code in een spel is vereist, en zal daarom ook worden beschreven in dit document.
- **Administrator**: De administrator is een speciale actor die een spelsessie begeleidt. Hij heeft meer rechten dan een reguliere speler en kan tijdens het spelen aanpassingen maken door modules in en uit te schakelen. Dit is mogelijk met de dynamische functionaliteit van ons systeem.

Must have:

Modelleren:

M1. Met het systeem kan een nieuw type spel-object worden gespecificeerd. Een spel-object type heeft de volgende eigenschappen:

M1.1. Het heeft een naam.

M1.2. Het heeft velden. Een veld in een spel-object type moet een specifiek type hebben. Dit type kan een spel-object type zijn of een Java klasse.

M1.3. Het kan annotations hebben.

M1.4. Het kan een sub-type zijn van een ander type.

M2. Met het systeem kunnen de eigenschappen van een spel-object type worden gewijzigd. (zie **M1.1**, **M1.2**, **M1.3**)

M3. Met het systeem kan een spel-object type aan een spel-model worden toegevoegd

(zie Glossary).

M4. Het systeem ondersteunt het verwijderen van een spel-object type uit een spel-model. De specifieke functionaliteit van dat spel-object moet uit het spel-model worden verwijderd, maar de overige spel-functionaliteit (ook datgene wat gedeeltelijk van dit spel-object afhankelijk was) moet blijven functioneren.
M4.1. De acties van het spel-object type worden verwijderd.
M4.2. Spel-object types die sub-type zijn van het verwijderde type blijven bestaan, maar zonder de functionaliteit van het verwijderde spel-object type over te erven.
M4.3. Andere spel-object types die een veld hebben met het verwijderde spel-object type, werken nu alsof ze dit veld niet meer hebben.

M5. Het spel-model bevat een hiërarchische (boom) structuur van "modules".
M5.1. Een module kan worden aangemaakt onder een bestaande module.
M5.2. De naam van een module kan worden gewijzigd.
M5.3. Een module (inclusief inhoud, zie **M5.4**, **M4**) kan worden verwijderd.
M5.4. Spel-object types worden altijd bevat door een module.
M5.5. Het is mogelijk een bestaande module uit een ander spel-model te importeren.
M5.5.1. De inhoud van de geïmporteerde module kan worden gekopieerd naar het huidige spel-model. en:
M5.5.2. Het huidige spel-model kan refereren naar de geïmporteerde module. Wijzigingen in de module hebben dan effect in het spel-model dat ernaar refereert.

M6. Het systeem ondersteunt het toevoegen van een causaal effect en een noodzakelijke voorwaarde voor dat effect aan een spelgebeurtenis.
M6.1. Een causaal effect kan als Java code in een .java document worden toegevoegd.

Uitvoeren:
M7. Het systeem ondersteunt het uitvoeren van een spel-model vanuit een spel.
M7.1. Wanneer een event plaats vindt wordt, mits de noodzakelijke voorwaarde waar is, het in het spel-model gedefinieerde effect uitgevoerd.
M7.2. Ondersteunt het instantiëren van spel-object types gedefinieerd in het spel-model.
M7.3. Een waarde van een veld van een spel-object kan worden gezet mits het type van de waarde overeenkomt met het type van dat veld.
M7.4. Een waarde van een veld van een spel-object type kan worden opgevraagd.

Should have:
S1. Functionele eisen **M1.1**, **M1.2**, **M1.3**, **M2**, **M3**, **M4**, **M5**, **M6** worden ondersteund door middel van een grafische gebruikersinterface.

S2. De interface kan de volgende relaties tussen spel-objecten en effecten visualiseren:
S2.1. Spel-object type is van invloed op voorwaarde voor causaal effect (zie **M6**).
S2.3. Spel-object type bevat ander spel-object type (zie **M1.2**).
S2.3. Spel-object type is sub-type van ander spel-object type (zie **M1.4**).

S3. Het systeem sluit aan op de bestaande spel-architectuur en SGE; dat wil zeggen:
S3.1. De functionaliteit voor functionele eis **M7** (inclusief **M7.1**-**M7.4**) is beschikbaar vanuit een spel op basis van SGE. (zie non-functionele eisen 3.3.6)
S3.2. Een veld van een in het spel-model gedefinieerd spel-object type kan als type een bestaand spel-object type hebben.
S3.3. Functionele eis **M6** kan gebruik maken van bestaande spel-gebeurtenissen.

Could have:
C1. (Developer, Programmeur) Het effect in functionele eis **M6** kan bestaan uit een keten van andere effecten.

Would like to have (but probably won't):
W1. (Administrator) Een module in een spel-model kan worden uitgezet tijdens het uitvoeren van een spel dat van dat spel-model gebruik maakt.
W1.1. De inhoud van de module wordt (vergelijkbaar met functionele eis **M4**) effectief uit het spel verwijderd.
W1.2. Een module die volgens functionele eis **W1** is uitgezet, kan weer worden aangezet.

W2. (Administrator) De "voorwaarden" in functionele eis **M6** kunnen tijdens het uitvoeren van een spel gedeeltelijk worden uitgeschakeld. Effecten die hiervan afhankelijk waren worden dan uitgevoerd ook als niet aan de uitgeschakelde voorwaarde wordt voldaan.

3.3. Non-functionele eisen

3.3.1. User interface and human factors
(n.a.v. **S2**) De grafische gebruikersinterface moet begrijpbaar zijn voor een gebruiker met kennis van UML, Event Driven Process Chains.

3.3.2. Documentation
Er hoeft geen gebruikershandleiding te worden geleverd. Documenten, zoals dit en het eindrapport, waarin de functionaliteit van het systeem wordt uitgelegd en wordt beschreven wat het toevoegt aan het bestaande systeem, en op welke manier is echter wel van belang.

3.3.3. Hardware considerations
Geen bijzondere eisen.

3.3.4. Performance characteristics
Een op basis van het systeem ontwikkeld spel moet geschikt zijn voor real-time interactie en kunnen draaien op ten minste 30 FPS. Het deel van het systeem dat door de developers gebruikt zal worden, moet zodanig snel functioneren dat het geen significante belemmering van de productiviteit kan vormen, wachttijden mogen niet hoger zijn dan enkele seconden.

3.3.5. Error handling and extreme conditions
Het systeem bevat geen hele kritieke of belangrijke informatie. De modellen die ermee gemaakt worden, moeten m.b.v. een autosave functie direct worden opgeslagen in een database of XML files, om verlies van werk te voorkomen.

3.3.6. System interfacing
Er zijn geen system-interfacing eisen voor het modelleer-gedeelte van het te ontwerpen systeem. De functionaliteit voor functionele eisen **M7** (inclusief **M7.1**-**M7.4**) en **S3.1** zijn beschikbaar door middel van een Java-package.

3.3.7. Quality issues
Het systeem moet kunnen omgaan met foute invoer van de gebruikers. Fouten in een Effect of Precondition mogen het spel niet doen crashen: ze moeten 'zacht' falen, en de error loggen.

3.3.8. System modifications
Het systeem moet heel gemakkelijk uit te breiden zijn met functionaliteit die analoog werkt aan bestaande functionaliteit. Verder moet het systeem zodanig zijn ontworpen dat het bruikbaar blijft wanneer de context van het systeem (d.w.z.; de engine en een spel dat het systeem gebruikt) verandert.

3.3.9. Physical environment
Geen bijzondere eisen.

3.3.10. Security issues
Er staan in het spel geen belangrijke financiële of privé gegevens opgeslagen, dus beveiliging tegen aanvallen van buitenaf is geen belangrijke eis. Het is van belang dat een klant van de opdrachtgever niet heel gemakkelijk een spel kan aanpassen. Dit zoude opdrachtgever buiten spel zetten en deze zou hierdoor geld kunnen mislopen. Het model van de situatie moet dus zo gecodeerd en opgeslagen worden dat het alleen voor de opdrachtgever mogelijk is om aanpassingen te maken.

3.3.11. Resources and management issues
Het systeem is 'software only' en heeft zodanig geen fysiek onderhoud nodig. Het back-uppen van producten ontwikkeld met behulp van het systeem valt buiten de scope van het systeem.

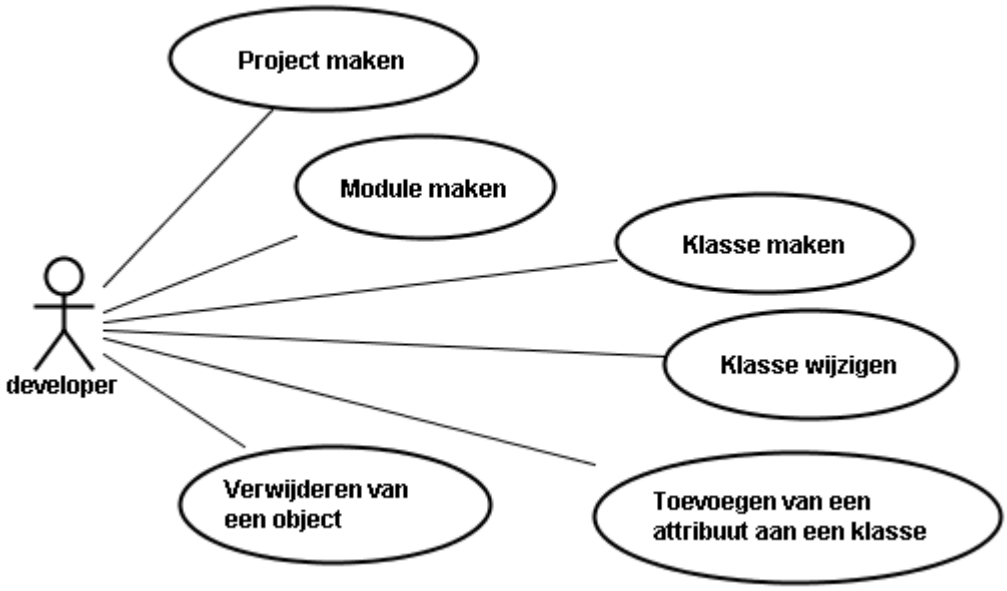
3.4. Constraints

Het project wordt gedaan in Java.

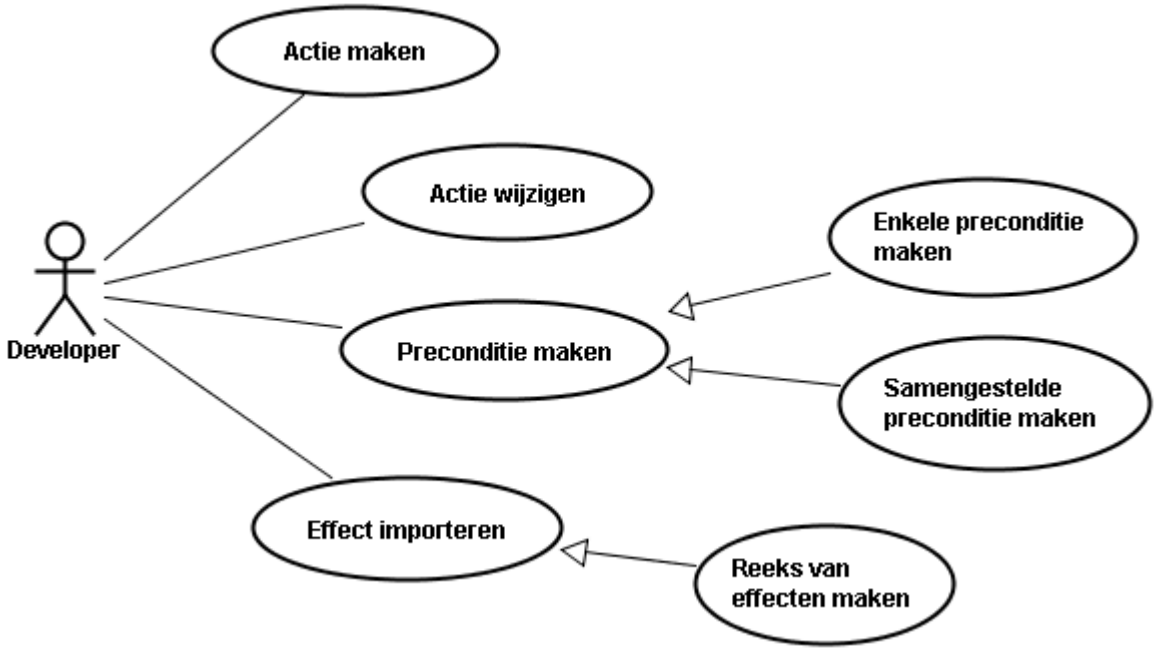
3.5. Systeemmodellen

3.5.1. Actors

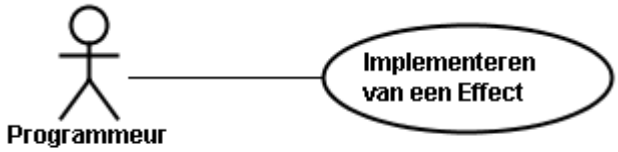
In deze sectie worden per gebruiker de acties weergegeven die deze gebruiker met ons systeem kan uitvoeren. De acties voor de developer zijn over twee diagrammen verdeeld om het overzichtelijk te houden.



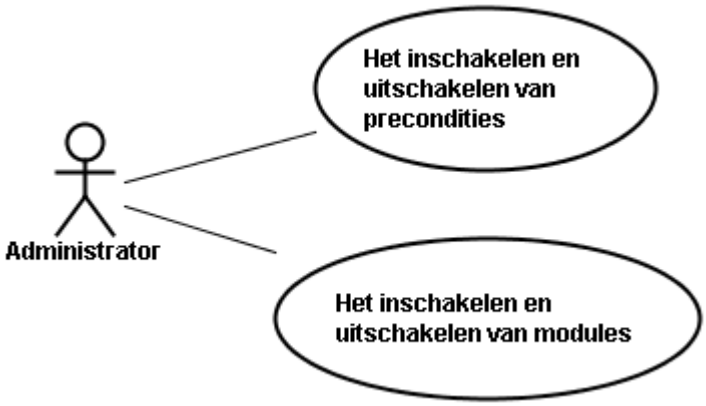
Figuur 1 *UseCase Developer (EntityTypes)*



Figuur 2 *UseCase Developer (acties)*



Figuur 3 *UseCase Programmeur*



Figuur 4 *UseCase Administrator*

3.5.2. Use case model

Use Case Het maken van een project
Actor: developer

Functionele eis(en): **M5**, **M7**, **S1**
Entry condition: De GUI is geopend

Flow of events:
1. De developer drukt in het menu op File > Nieuw project.
2. De developer kiest waar hij het project op wil slaan en geeft het een naam.
3. De developer drukt op "Opslaan".

Exit condition: Het project is aangemaakt, en in de GUI worden een leeg vlak en een lege project-tree weergegeven.

Use Case Het maken van een klasse
Actor: developer
Functionele eis(en): **M1**, **M3**, **S1**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer voert in dat hij een nieuwe klasse wil maken (bijvoorbeeld door op een knop te drukken).
2. De developer geeft de naam op van de klasse, en eventueel al andere informatie, zoals de parent van de klasse.
3. De developer bevestigt het maken van de nieuwe klasse.

Exit condition: De klasse is aangemaakt en wordt weergegeven in de GUI. Nu kan hij verder worden bewerkt.

Use Case Het toevoegen van een property/attribuut
Actoren: developer
Functionele eis(en): **M1**, **M2**, **S1**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer selecteert de klasse om het attribuut aan toe te voegen.
2. De developer voegt onder aan de lijst van attributen een nieuw attribuut toe.
3. De developer voert het type voor dit attribuut in.
4. De developer voert de naam in van het attribuut.
5. (optioneel) De developer selecteert het nieuwe attribuut, en voegt hieraan enkele Annotations toe.

Exit condition: Het attribuut is toegevoegd aan de klasse.

Use Case Het maken van een nieuwe actie
Actoren: developer
Functionele eis(en): **M6**, **S1**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer selecteert een klasse om een actie voor te definieren.
2. De developer selecteert vervolgens een Event om de actie aan te koppelen.
3. De developer schrijft een preconditie-expressie.
4. De developer selecteert een Effect.
5. De developer geeft een naam op voor de Actie en bevestigt.

Exit condition: De actie is toegevoegd.

Use Case Het maken van een preconditie
Actoren: developer

Functionele eis(en): **M6**, **S1**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer selecteert een actie en geeft aan dat hij een preconditie wil toevoegen.
2. Op het scherm dat verschijnt typt de developer een statement in (de gebruikte taal is Java).
3. De developer bevestigt en de preconditie wordt opgeslagen.
4. De preconditie kan ook aan een andere preconditie gekoppeld worden;
5. De developer selecteert de precondities en geeft de relatie hiertussen aan.

Exit condition: Er is nu een samengestelde preconditie gemaakt en opgeslagen. Deze kan aan andere precondities worden gekoppeld waardoor een hierarchie gemaakt kan worden.

Use Case Het verwijderen van een object
Actoren: developer
Functionele eis(en): **M4**, **S1**
Entry condition: Er is een project in de GUI geopend, en er is minstens één object gemaakt (dit kan zowel een klasse, actie of preconditie zijn)

Flow of events:
1. De developer selecteert een object in de GUI.
2. De developer geeft aan dat hij het wil verwijderen.
3. De developer bevestigt de verwijdering.

Exit condition: Het object is uit het model verwijderd, en de objecten waarmee het mogelijk een relatie had zijn geupdate.

Use Case Het definiëren van een effect door een reeks effecten te koppelen
Actoren: developer
Functionele eis(en): **C1**, **S1**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer opent de lijst van mogelijke effecten.
2. De developer geeft aan een nieuwe keten van effecten te willen maken.
3. De developer voegt een effect toe aan de keten.
4. De developer herhaalt 4 tot alle gewenste effecten aan de reeks zijn toegevoegd.
5. De developer geeft de keten een naam, slaat de keten op, en sluit de GUI.

Exit condition: Er is een nieuwe keten van effecten gemaakt.

Use Case Het wijzigen van een klasse
Actoren: developer
Functionele eis(en): **M1**, **S1**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer selecteert de klasse in de GUI. Nu zijn er verscheidene mogelijkheden, hij kan o.a.:
    - de naam van de klasse wijzigen
    - properties/attributen toevoegen, verwijderen, (tijdelijk) uitzetten
    - een andere parent aan de klasse geven
    - acties toevoegen, verwijderen en (tijdelijk) uitzetten
2. Nadat de developer heeft gewijzigd wat hij wilde veranderen, bevestigt hij de wijziging.

Exit condition: De klasse is gewijzigd.

Use Case Het aanmaken van een nieuwe module
Actoren: developer
Functionele eis(en): **M5**
Entry condition: Er is een project in de GUI geopend

Flow of events:
1. De developer wil een nieuwe module voor spelfunctionaliteit maken.
2. De developer geeft aan dat hij een nieuwe module wil aanmaken.
3. De developer geeft een naam en eventueel een bestandsnaam op.
4. De developer bevestigt.

Exit condition: Er is een nieuwe module aangemaakt.

Use Case Het uitschakelen van een (deel van een) module tijdens het de uitvoer van een spel
Actoren: administrator
Functionele eis(en): **W1**
Entry condition: Er wordt een spel gespeeld

Flow of events:
1. De administrator opent (via een GUI, van het spel of van ons) de lijst met modules.
2. De administrator selecteert een module en geeft aan dat hij deze wil uitschakelen.

Exit condition: De module wordt niet meer door het spel gebruikt.

Use Case Het maken van een effect
Actoren: programmeur, developer
Functionele eis(en): **M6**
Entry condition: Er is een project

Flow of events:
1. De programmeur opent zijn favoriete java IDE en maakt een nieuwe klasse aan, welke de Effect interface implementeert.
2. De programmeur implementeert de functionaliteit.
3. De programmeur slaat het bestand op.
4. De developer importeert het effect in de GUI van ons systeem.

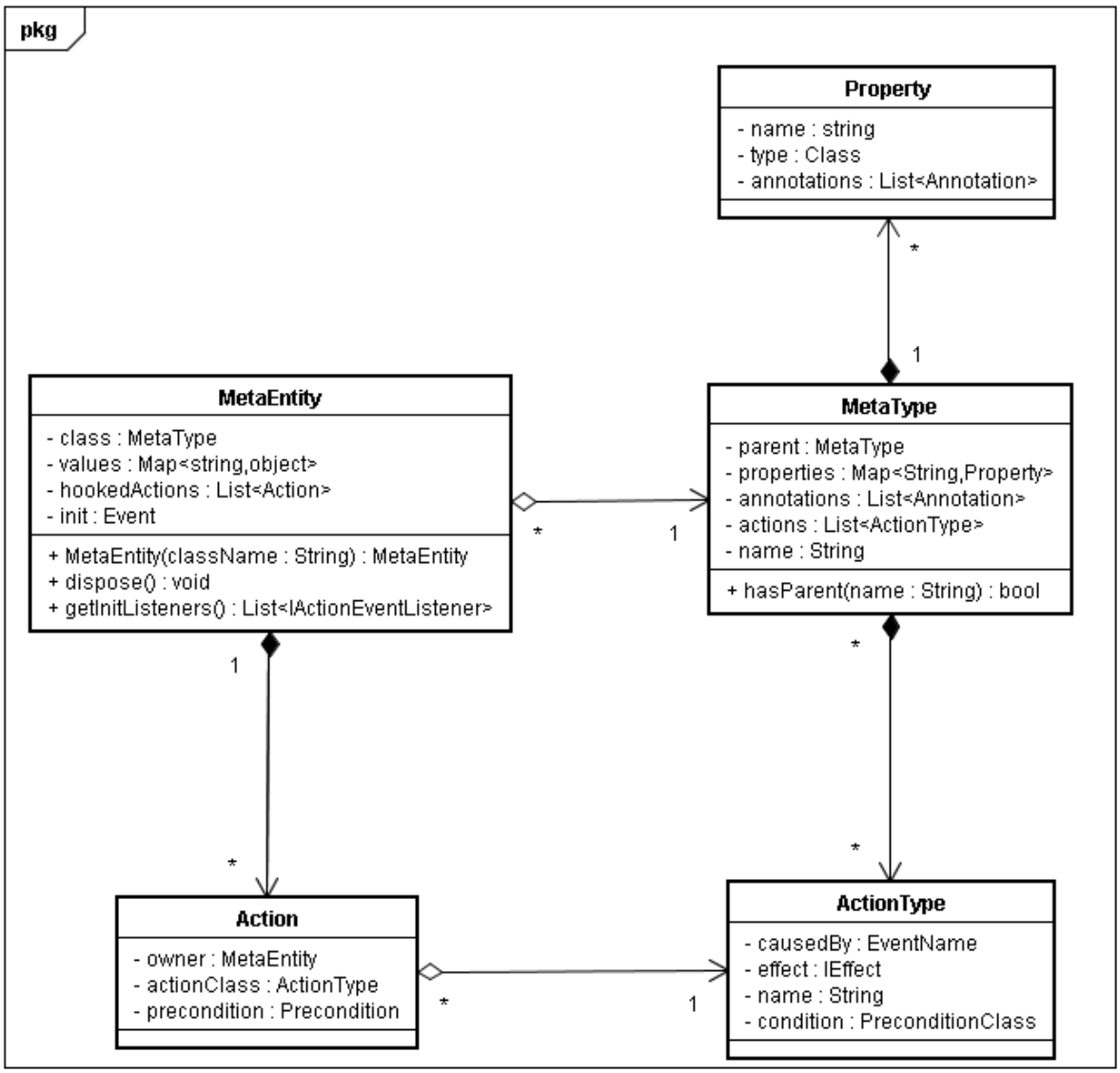
Exit condition: Er is een effect gemaakt dat door ons systeem kan worden gebruikt.

3.5.3. Object model

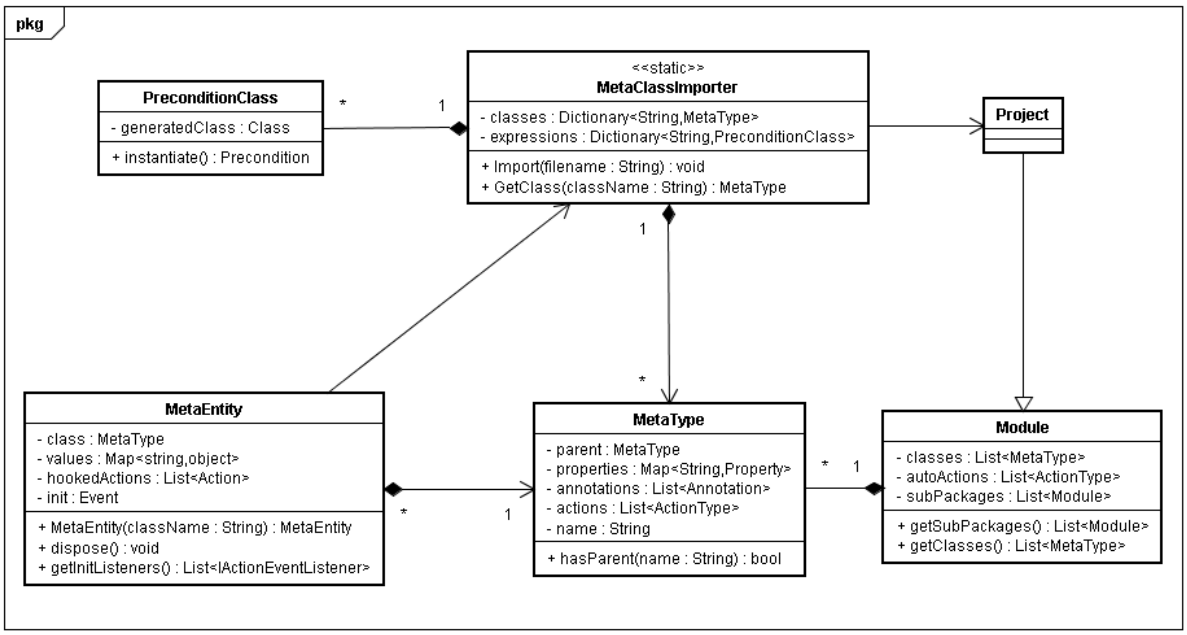
3.5.3.1. Data Dictionary

Annotations: Meta-data die in java-code aan onder andere klassen, velden en methodes kunnen worden toegevoegd.
Effect: Een stuk code dat de interface IEffect implementeert, welke door het systeem kan worden aangeroepen, en spel-functionaliteit bevat.
Module: Een afgebakend stuk code dat (bijv. door een administrator) tijdens het spelen vrij in- en uitgeschakeld kan worden.
Property: Een eigenschap van een type/klasse. Wat normaal in Java wordt gezien als een attribuut.

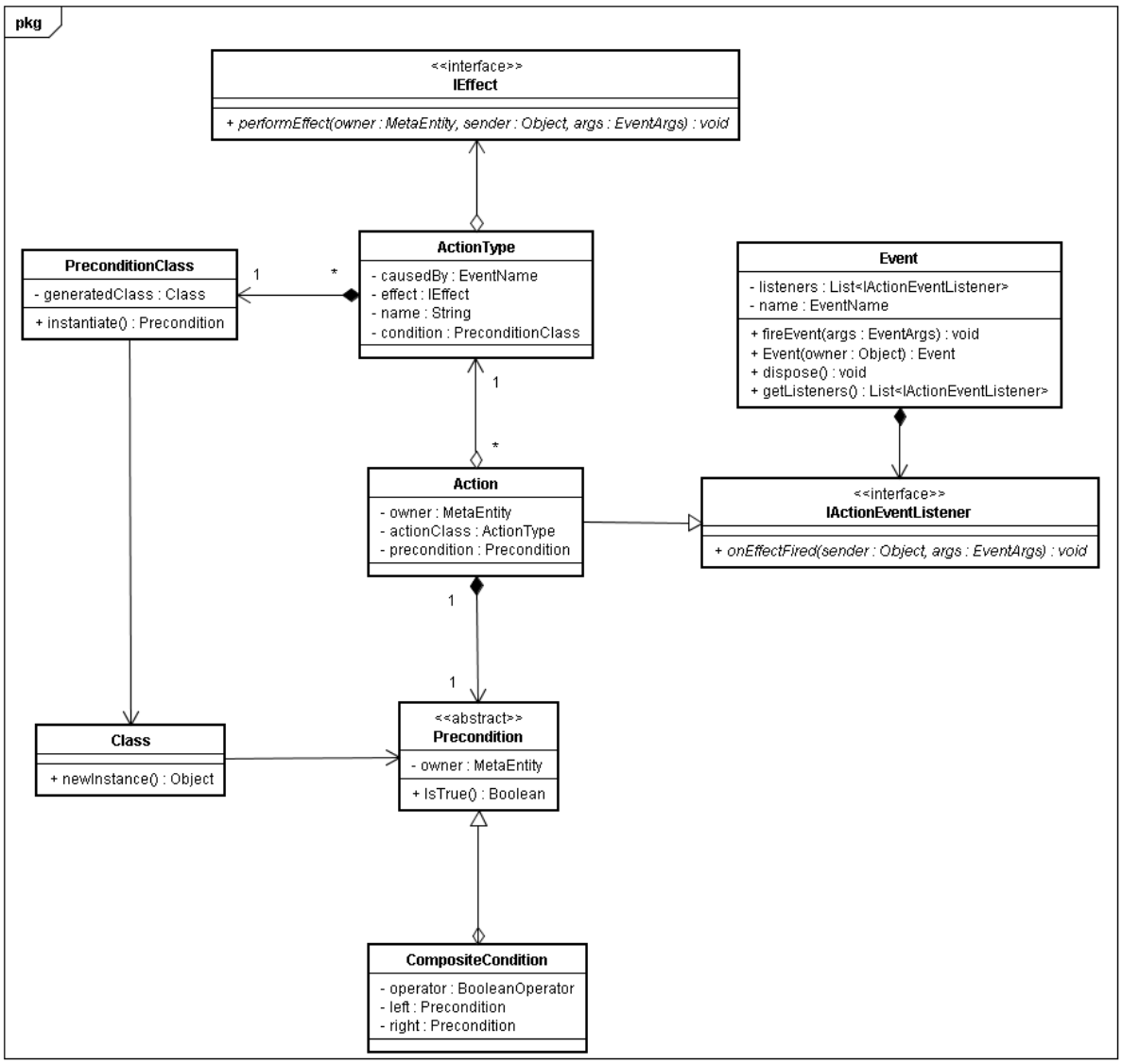
3.5.3.2. Class Diagrams



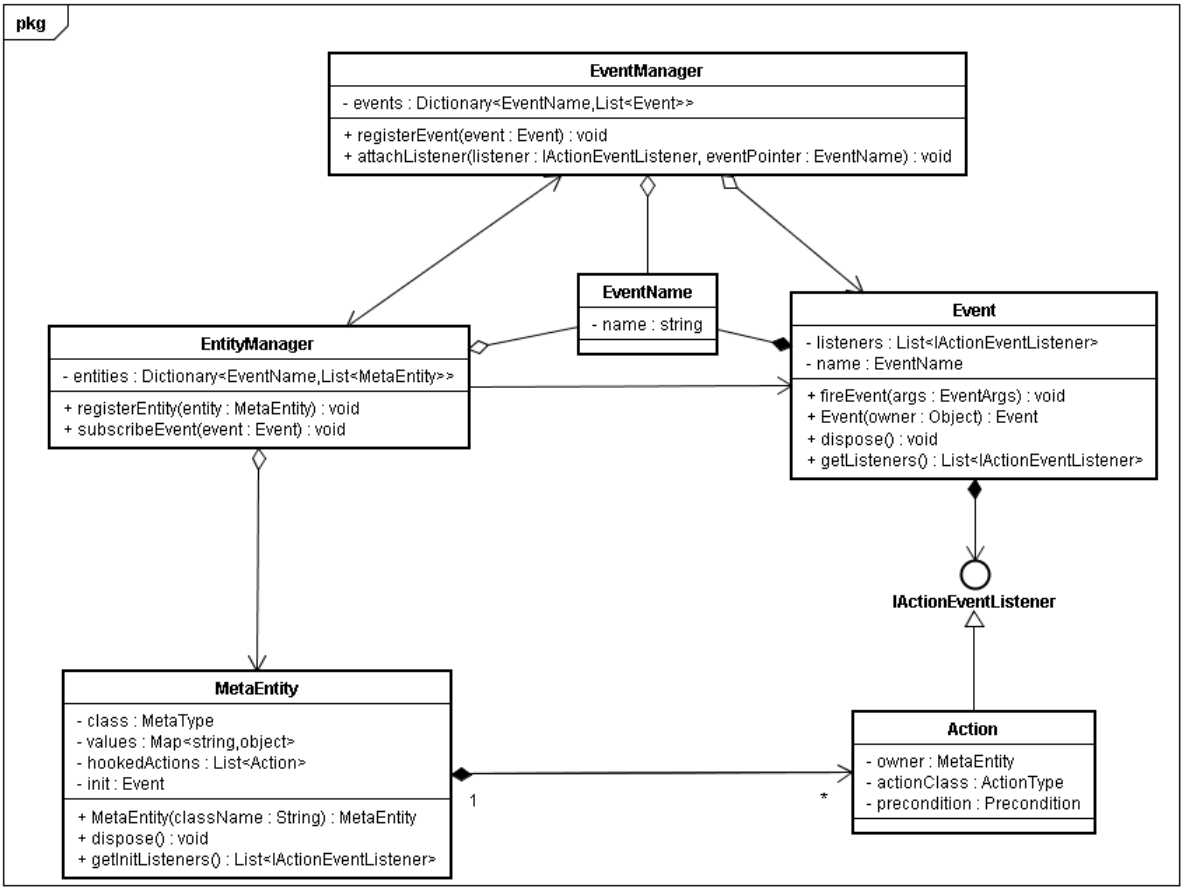
Class Diagram 1: overzicht van globale run-time structuur.



Class Diagram 2: overzicht import classes.

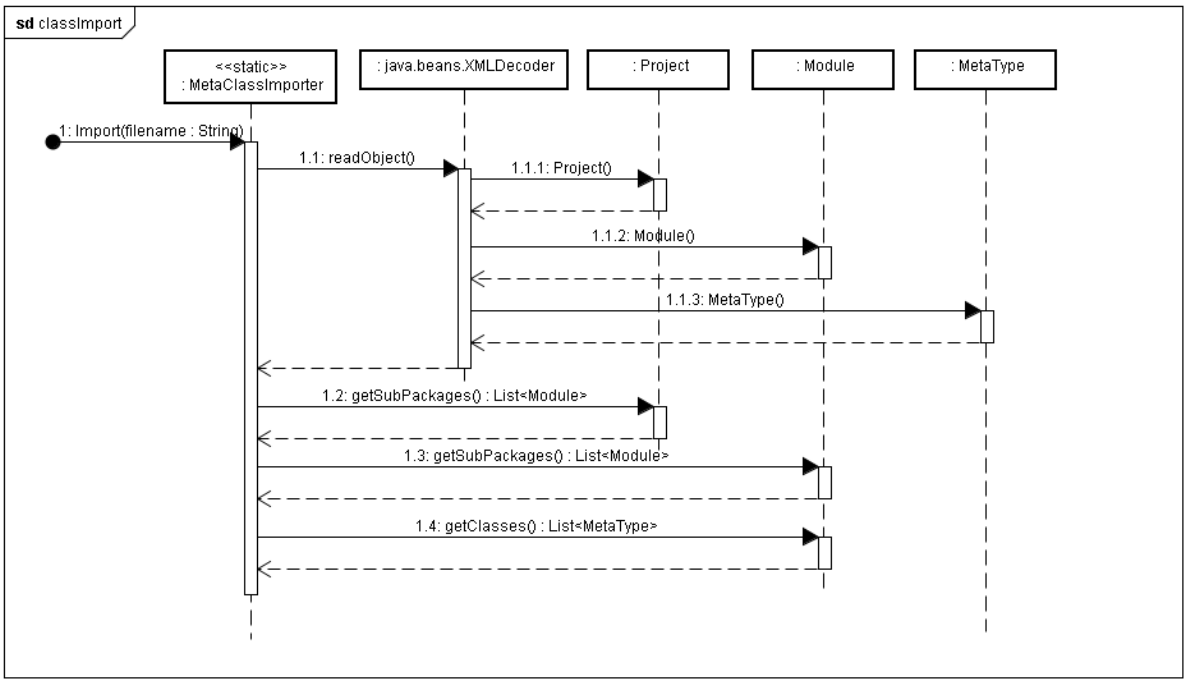


Class Diagram 3: Actions

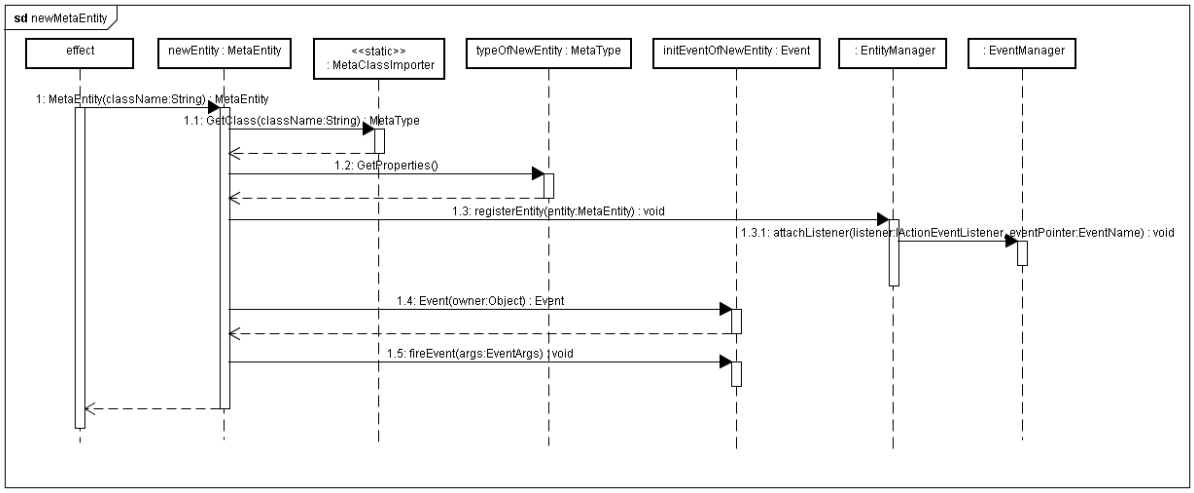


Class Diagram 4: Events

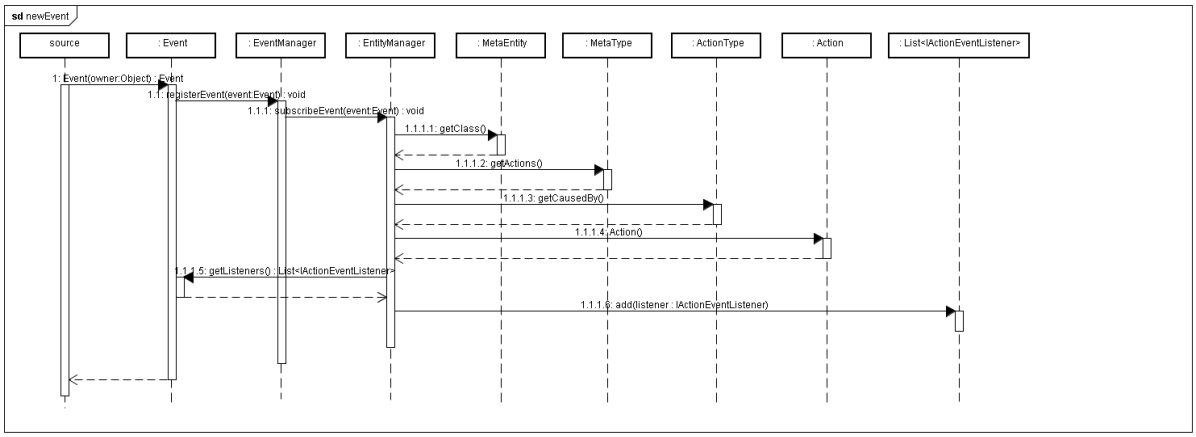
3.5.4. Dynamische modellen



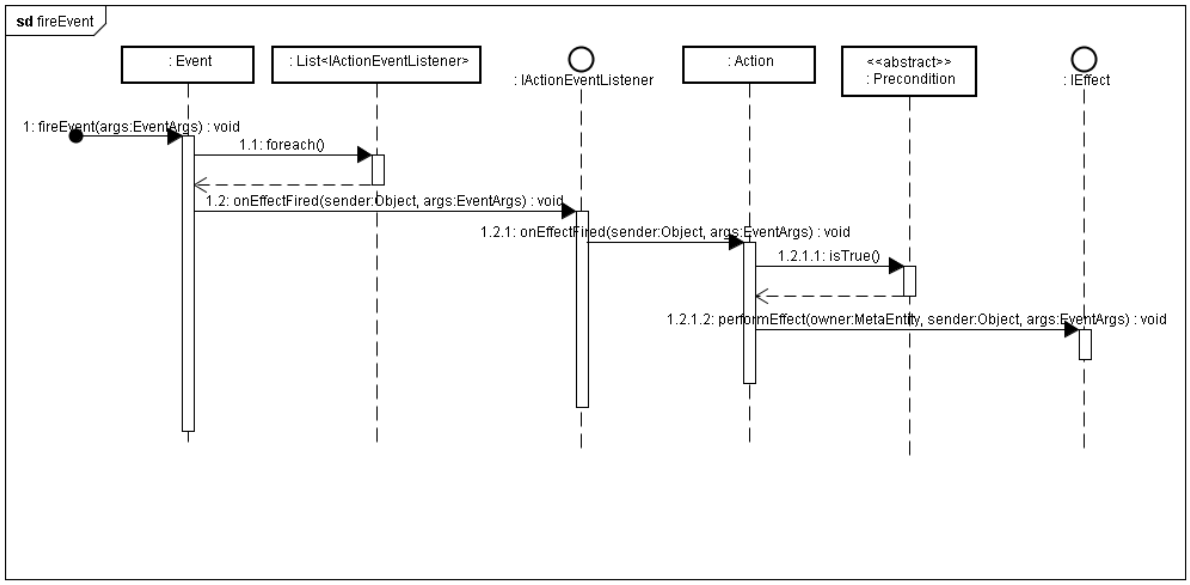
Sequence Diagram 1: Importen/Laden van een project.



Sequence Diagram 2: Het aanmaken van een nieuwe instantie.

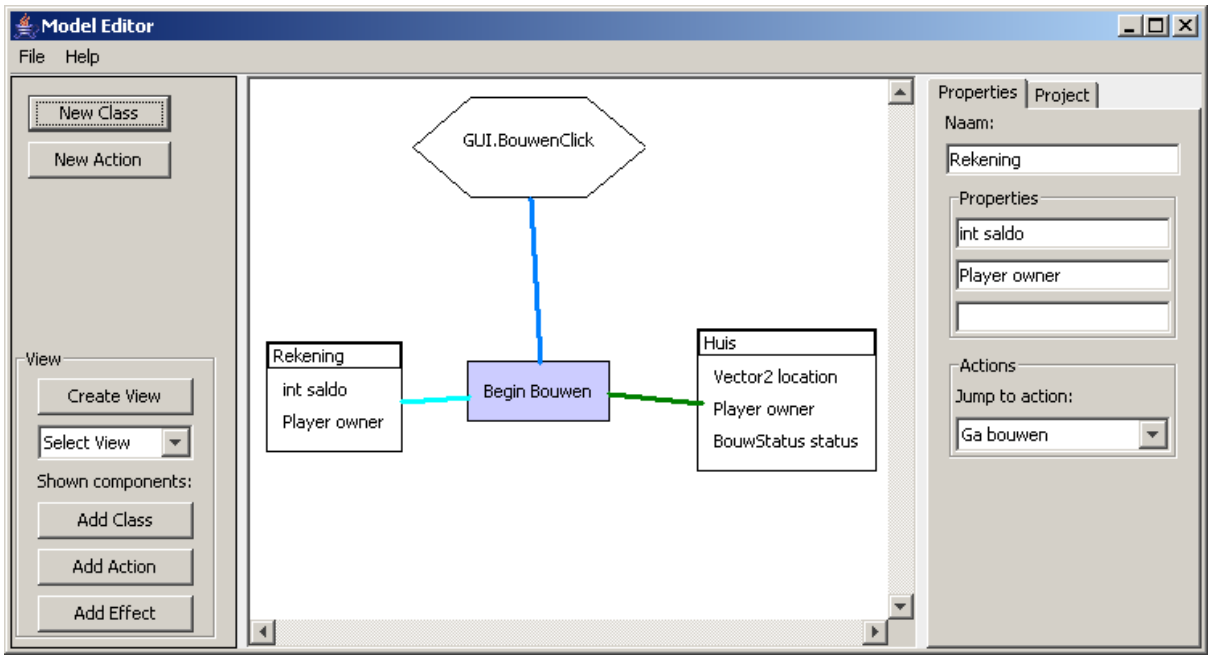


Sequence Diagram 3: Het aanmaken van een nieuwe Event-source instantie.

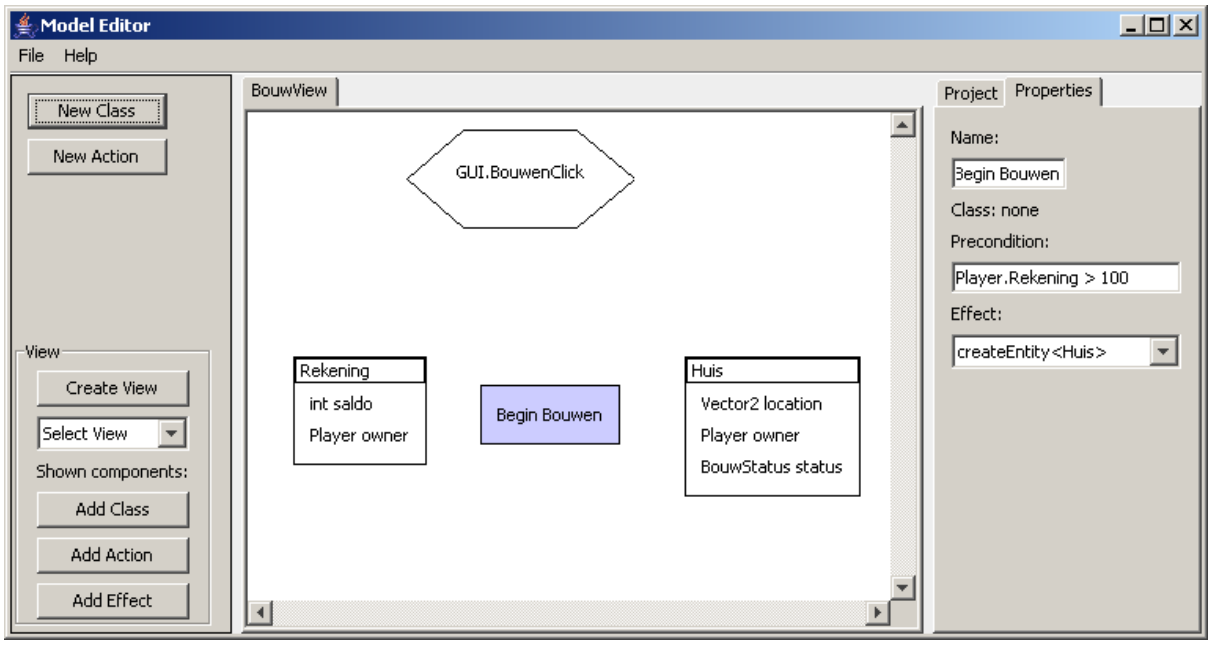


Sequence Diagram 4: Het afvuren en verwerken van een event.

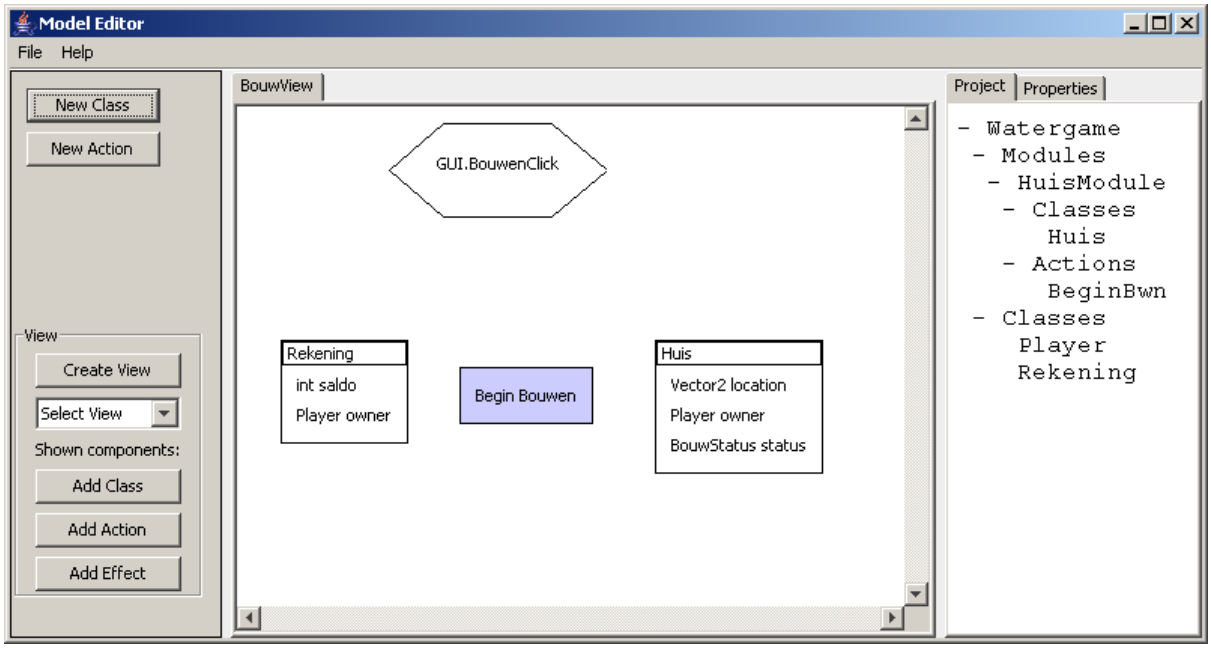
3.5.5. Gebruikersinterface



GUI Mockup 1: Properties panel van een Class.



GUI Mockup 2: Properties panel van een Action.



GUI Mockup 3: Project tree

4. Glossary

- Terminologie -

Modelleren: Tijdens de analyse van een situatie van een klant worden gegevens verzameld over welke objecten (ook wel entities) en acties er een rol spelen in de situatie. Uit deze informatie wordt vervolgens gekeken hoe dit in een spel geïmplementeerd kan worden en worden er modellen gemaakt die deze objecten, acties en de relaties ertussen aangeven (bijvoorbeeld klassen- en EDPC-diagrammen). Dit proces noemen wij modelleren.

(Spel-)Object/Entity: zie 2.1.3, Items.

Relatie: Er zijn verschillende soorten relaties die in een spel kunnen voorkomen. Het eerste soort bevat de relaties die in een UML klassendiagram voor kunnen komen, in ons geval zijn dit de aggregatie relatie (x bevat y) en overerfing (x is subtype van y).
Een ander soort relatie, dat voor een spel minstens even belangrijk is, beschrijft de manier waarop processen, in ons systeem acties genoemd, met hun omgeving en met andere acties samenwerken. Dit gaat er dus bijvoorbeeld over hoe het vervullen van een preconditie kan leiden tot een reeks acties, die er weer voor zorgen dat de waardes van andere properties gewijzigd worden. Dit soort relatie proberen wij in ons systeem weer te geven door een deel van de functionaliteit van EDPC diagrammen mee te nemen in onze GUI.

Speler: Eindgebruiker van het product van Tygron, degene die het daadwerkelijke spel speelt.

Spel-model: De spel-objecten en relaties daartussen zoals die door de developer gedefinieerd zijn voor een spel.

Bijlage A. Ontwerp Watergame

Bijlage B. Event concept Watergame

Object Design Document

Tygron Bachelor Project

Design Document

8 Juni 2009

Sid Mijnders **Tom Lotgering**

Begeleiders: Hans Geers, Bernard Sodoyer
Versie: 1

Inhoudsopgave

1. Inleiding

1.1. Voordelen

Met behulp van ons systeem kan spel functionaliteit aan een spel worden toegevoegd zonder de code van dat spel te wijzigen. Het spel hoeft zelfs niet opnieuw gecompileerd te worden.

Spel-functionaliteit kan ook uit het spel worden verwijderd, of aangepast, zonder de code van het spel te hoeven wijzigen.

Het aanmaken, wijzigen en verwijderen van spelfunctionaliteit kan worden gedaan door iemand met zeer beperkte programmeer ervaring.

De spel-functionaliteit wordt op een grafische manier weergegeven en bewerkt; dit geeft een beter inzicht in de werking van die functionaliteit.

Doordat spel-functionaliteit op een enkele plaats beheerd wordt, en niet voor meerdere spellen gekopieerd hoeft te worden, wordt het gemakkelijker onderhoud aan deze functionaliteit te plegen, omdat dit niet voor iedere kopie apart gedaan hoeft te worden.

1.2. Risico's en trade-off's

Performance vs. flexibiliteit
Met ons systeem moet het mogelijk zijn om (vrijwel) alle te bedenken situaties te implementeren in een spel, of hier in ieder geval een basis voor te maken. Om deze reden draait het bij ons systeem voornamelijk om de flexibiliteit. Dit heeft tot gevolg dat er niet met gespecialiseerde klassen wordt gewerkt, maar met zogenaamde metaklassen, waardoor het over het algemeen tijdens de uitvoer trager is. We proberen hier rekening mee te houden door zo veel mogelijk klassen te precompilen of in te lezen voordat begonnen wordt met het spelen van een spel en verwachten -met het oog op de snelheid van computers vandaag de dag- niet dat dit voor problemen zal zorgen.

Een nadeel van deze meta-architectuur is ook dat de ruimte voor een programmeur om zich uit te drukken beperkt wordt. Zo is het bijvoorbeeld niet mogelijk om functies te definieren in de klassen die met behulp van on systeem gemaakt zijn.

Sommige programmeer fouten, die normaal gesproken door de java-compiler zouden worden opgemerkt, kunnen binnen ons systeem pas in een later stadium herkent worden. In code die handmatig voor ons systeem geschreven wordt, is bijvoorbeeld geen type-checking mogelijk 'at compile-time', voor types die binnen ons systeem zijn gedefinieerd. Dit kan echter wel 'at runtime'.

1.3. Aannames

We nemen aan dat de volgende stellingen waar zijn, om ons ontwerp te rechtvaardigen:

Het is met het Adaptive Object Model (wat wij gebruiken) mogelijk om alle nodige situaties

te implementeren.

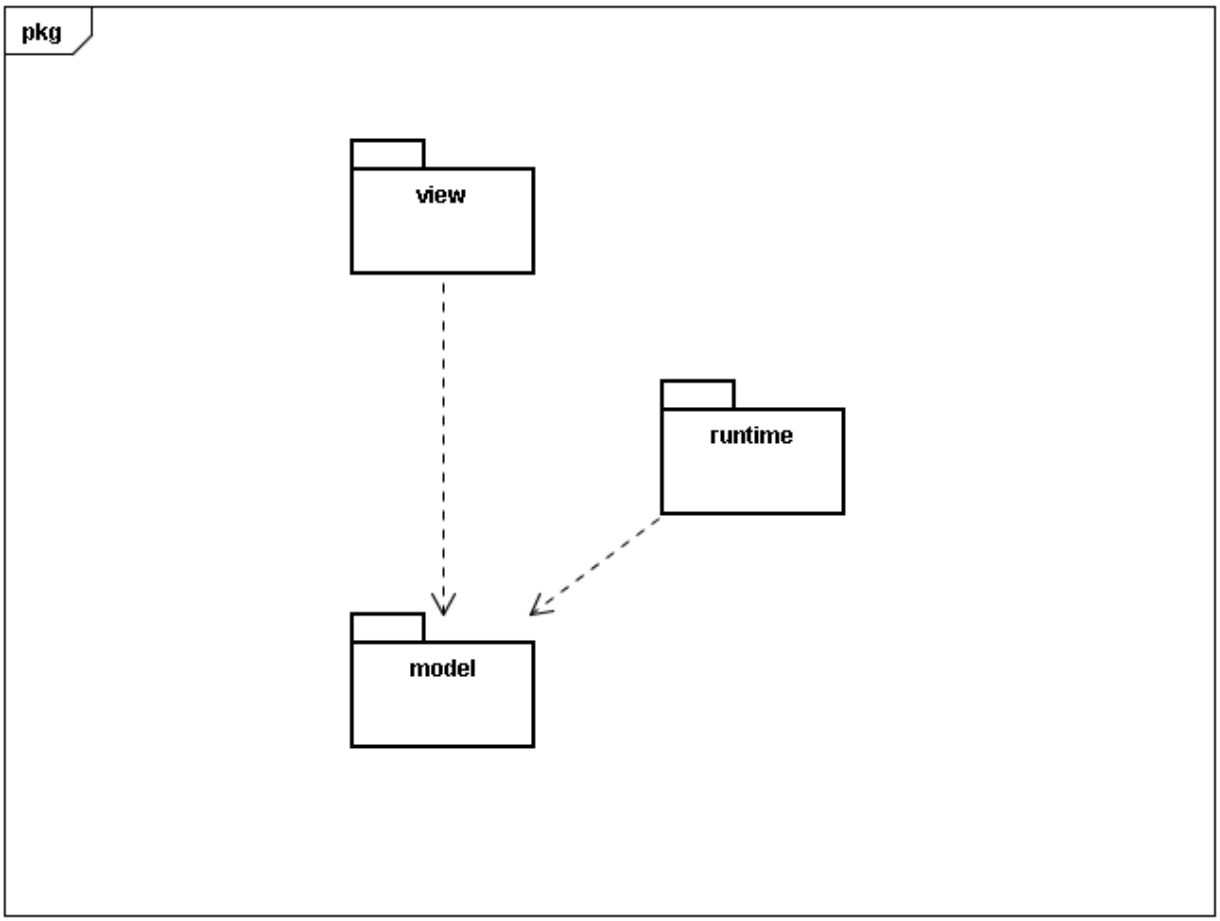
Het bedrijf is bereid aan het eigen systeem kleine aanpassingen te maken (bijvoorbeeld m.b.t. het gebruik van events) om met ons systeem samen te kunnen werken.

Programmeurs maken liever een project met onze interface dan code te copy-pasten.

2. High-level definition

2.1. Packages

Het systeem bevat globaal 3 packages, het view package, het runtime package, en het model package. Deze packages zijn als volgt van elkaar afhankelijk:



Figuur 1. System overview.

2.1.1. View

Het *view* package bevat alle klassen die uitsluitend met de GUI te maken hebben, en dus niet door het *runtime* package nodig zijn.

2.1.2. Runtime

Het *runtime* package bevat alle klassen die uitsluitend met de runtime library te maken hebben, en dus niet door het *view* package nodig zijn. Deze klassen worden tijdens het spelen van een spel gebruikt en werken samen met de Metaklassen die in het model zijn gedefiniëerd. Het gedrag het spel wordt eerder (in het model) geïmplementeerd en door deze klassen gebruikt.

2.1.3. Model

Het *model* package bevat klassen die door zowel de *view* en de *runtime* packages nodig zjin, met name die klassen van het meta-model waarmee een klant-model beschreven kunnen worden.

3. Low-level definition

3.1. Classes

In deze sectie wordt elke klasse van ons systeem apart besproken. Eerst wordt er een korte omschrijving van de functionaliteit van de klasse gegeven en vervolgens worden de attributen en methodes uitgelegd. De getters, setters, constructors en voor de hand liggende attributen en methodes worden hier niet behandeld, met uitzondering als we willen uitleggen waarom we ervoor gekozen hebben om deze te gebruiken.

3.1.1. View classes

ExpressionCompiler

Methodes:
compile(String expression) : PreconditionClass
Implementeert een opgegeven expressie in een klasse en compileert deze, zodat de klasse die wordt gemaakt gebruikt kan worden door ActionType.

3.1.2. Runtime classes

IActionEventListener

Dit is een interface die wordt gebruikt om te reageren op een Event. Deze interface wordt geimplementeert door Action.

Action

De actions zijn de objecten waarmee tijdens de uitvoer van het spel waardes aangepast kunnen worden. Actions staan geregistreerd bij de Events waar ze naar moeten "luisteren". Als deze wordt afgevuurd, en hun precondition is waar, dan voeren ze het effect van hun ActionType uit.

Attributen

- MetaEntity owner
  De beheerder van de actie. Het object waardoor de actie wordt aangeroepen. Dit kan worden doorgegeven aan de precondition.
- ActionType actionClass
  Bevat de algemene info van dit type actie.
- Precondition precondition
  De actie kan uitgevoerd worden als de preconditie true is.

Methodes
- onEventFired (van de IActionEventListener Interface)
  Voert het effect uit.

EntityManager

Een "static klasse". Houdt alle MetaEntity-instanties bij die gemaakt worden en koppelt ze aan EventNames. Als een Entity geregistreerd wordt, worden gelijk ook alle acties geïnstantieerd en als listeners aan Events gekoppeld.

Attributen
- HashMap<EventName, LinkedList<MetaEntity>> entities
  Houdt voor elke Event bij welke MetaEntities eraan gekoppeld zijn.

Methodes
- registerEntity(MetaEntity entity)
  Voegt de entity toe aan de lijsten van alle events waar deze entity op kan reageren, en maakt voor al die events een Action aan voor de entity, en voegt deze toe aan het event als listener.
- subscribeEvent(Event event)
  Als er een Event wordt gemaakt zorgt deze methode ervoor dat voor alle meta-entities die aan dat event gekoppeld horen te worden, dit ook daadwerkelijk gebeurt.

Event

Events spelen een belangrijke rol in het plaatsen van Actions. Elk ActionType heeft een Event waar het op reageert en door deze af te vuren kunnen dus Actions aan de GUI en aan elkaar gekoppeld worden.

Attributen
- Object owner
  De sender van het Event. Wordt meegegeven als het event wordt afgevuurd.
- List<IActionEventListener> listeners
  De listeners/actions. Moeten geïnformeerd worden als het event wordt afgevuurd.
- EventName name
  De naam van het event. Er kunnen meerdere instanties van Events zijn met dezelfde naam.

Methodes
- fireEvent(Object args)
  Meldt aan alle listeners dat het event is afgevuurd

EventManager

Een statische klasse. De EventManager houdt bij welke Events welke EventNames hebben, en heeft ook methodes om hier listeners aan te koppelen. Zo kan een Action bijvoorbeeld aan meerdere events tegelijk gekoppeld worden.

Attributen
- HashMap<EventName, List<Event>>
  Houdt bij welke events welke EventName hebben.

Methodes
- registerEvent(Event event)
  Voegt een Event/EventName koppel toe aan de hashmap. Subscribed het event ook in de EntityManager.
- attachListener(IActionEventListener listener, EventName eventPointer)
  Voegt een willekeurige implementatie van IActionEventListener als listener toe aan alle events die als EventName eventPointer hebben.
- attachListeners(MetaEntity owner, ActionType at, EventName eventPointer)
  Maakt een nieuwe Action van type 'at' met 'owner' als *owner*, en voegt deze als listener toe aan alle events die als EventName eventPointer hebben.

MetaClassImporter

Importeert alle klassen (/MetaTypes) uit de xml-bestanden van een project en zet ze in een hashmap.

Attributen
- HashMap<String, MetaType> classes
  Houdt de ingelezen klassen bij.

Methodes
- import(String filename)
  Initialisatiestap voor het importeren. Leest een XML-bestand in en zet deze om in een Module, die vervolgens wordt meegegeven aan importModule().
- importModule(Module m)
  Wordt aangeroepen door import(). Doorloopt de meegegeven Module en submodules op een recursieve manier en zet de classes in de hashmap.
- getClass(String className) : MetaType
  Doorzoekt de HashMap classes en geeft de MetaType met naam className terug.

MetaEntity

Dit zijn de "objecten" die tijdens het uitvoeren van het spel worden gebruikt. Elke MetaEntity heeft een MetaType, die de klasse voorstelt. Een instantie van MetaEntity kan worden gezien als een daadwerkelijke instantie van die MetaType-klasse, een instantie van een MetaType is uiteraard ook een instantie, maar kan beter worden gezien als een nieuwe klasse. Er is voor deze MetaType-MetaEntity constructie gekozen om het mogelijk te maken dynamisch met een GUI nieuwe types/klassen aan te maken.

Attributen
- MetaType class
  Het type Entity. Wordt class genoemd omdat het voor de gebruikers (in de GUI) lijkt of ze nieuwe Java klassen aanmaken.
- Map<String, Object> values
  De waardes van de properties die in MetaType gedefiniëerd zijn.

- List<Action> HookedActions
  Dit is zodat deze listeners van de events kunnen worden verwijderd.
- Event init
  Initialisatie-event. Laat aan de rest van het programma weten dat deze MetaEntity er is, en dus gebruikt kan worden (bijvoorbeeld door andere Events).

Methodes
- MetaEntity(String className)
  Constructor

Precondition

De preconditie voor een actie. Als een event wordt afgevuurd wordt gekeken of aan deze preconditie wordt voldaan. Als dit zo is wordt het effect van de actie uitgevoerd en anders gebeurt er niks.

Attributen
- MetaEntity owner
  De owner van de actie waar dit een preconditie van is, zodat hier informatie van kan worden opgevraagd (bijvoorbeeld: als de status van de owner x is, return true).

Methodes
- isTrue() : boolean
  Geeft true als de Preconditie waar is (voert dus de eerder opgegeven en gecompileerde expressie uit).

CompositeCondition

Een subklasse van Precondition. Bevat twee Preconditions (kunnen ook weer CompositeConditions zijn) die aan elkaar gekoppeld zijn door een boolean operator.

Attributen
- *BooleanOperator operator*
  De boolean operator (AND, OR of NOT)
- *Precondition left*
- *Precondition right*

PreconditionClass

De klasse die automagisch uit het ingevoerde statement wordt gecompileerd door ons programma. Deze wordt geïnstantieerd en vervolgens aan een Action gekoppeld.

Attributen
- Class generatedClass
  De java.util.Class die een dynamisch gegereerde klasse van type Precondition beschrijft.

Methode
- instantiate() : Precondition
  Gebruikt de generatedClass om een instantie te creeren van het type dat door die klasse beschreven wordt, en geeft dit terug als Precondition instantie.

BooleanOperator

Dit is een enum van de operatoren AND, OR en NOT, met ingebouwde functionaliteit, zodat niet in een hogere klasse gecheckt hoeft te worden welke operator het is, maar de operator gelijk uitgevoerd kan worden.

3.1.3. Model classes

ActionType

Het type dat tijdens runtime aan een Action wordt toegewezen (elke actie moet een type hebben). Bevat informatie over wat dit type actie moet doen (effect) en wanneer (causedBy). Een ActionType kan los van een MetaType aangemaakt worden.

Attributen
- EventName causedBy
  Het Event waar dit type actie op reageert. Dit kan bijvoorbeeld een muisklik zijn, of een Event dat gemaakt is om acties aan elkaar te linken.
- IEffect effect
  Wat er moet gebeuren bij het uitvoeren van de actie (dus wanneer aan alle precondities is voldaan).
- String name
  De naam van dit type actie.
- PreconditionClass condition
  De preconditie moet true zijn om het effect uit te kunnen voeren. Als de condition null is wordt dit gezien als true.

Effect

Effects zorgen voor de link tussen ons systeem en het systeem van het bedrijf. Het programmeren hiervan is niet mogelijk in onze GUI en moet dus handmatig door de programmeur gedaan worden. Wij leveren een interface met de volgende methode:

- performEffect(MetaEntity owner, Object sender, Object args)
  Voert de code van het effect uit.
  Owner is de MetaEntity die de Action bevat die dit effect heeft afgevuurd. Deze kan null zijn.
  Sender is het object dat het Event bevat waar dit effect op reageert.
  Args is een object dat door sender wordt meegegeven, dat aanvullende informatie geeft over het event.

EventName

Elke Event heeft een EventName. De EventName kan in zekere zin gezien worden als het "type" van een event en wordt ook in de GUI gebruikt, waar Events alleen tijdens runtime gebruikt worden.

Attributen
- String name
  De naam van het event

Methodes
- equals(Object other) : boolean
  Vergelijkt twee EventNames op het name-attribuut (en geeft true als ze gelijk zijn).

- hashcode() : int
  Geeft de hashcode van de name.

MetaType

MetaTypes zijn die types die MetaEntities binnen ons systeem kunnen hebben. Ze representeren de klassen die in de GUI door de gebruiker gemaakt worden.

Attributen
- MetaType parent
  Hiermee is het mogelijk om subklasses te maken.
- Map<String, Property> properties
  De attributen van het type (de klasse)
- Annotation[] annotations
  Annotations worden door het bedrijf gebruikt en kunnen hier opgegeven worden. Annotations zjin meta-data die als flag fungeren.
- String name
- ActionType[] actionTypes
  Lijst van acties waar dit MetaType de owner van is.

Methodes
- hasParent(String name) : boolean
  Geeft true als er ergens hoger in de hierarchie een MetaType met de opgegeven naam bestaat. Dit betekent dat dit MetaType erft van het opgegeven MetaType.

Module

Een module is een stukje van het programma wat in- en uitgeschakeld kan worden zonder dat de rest van het programma niet meer werkt.

Attributen
- List<MetaType> classes
  "Losse" klassen binnen deze module, dus die niet in onderliggende modules zitten.
- List<ActionType> autoActions
  De acties die niet aan een instantie van een MetaType gebonden zijn. Hierdoor kan het systeem ge-bootstrapped worden met behulp van externe Events.
- List<Module> subPackages
  De onderliggende modules.

Methodes
- getSubPackages() : List<Module>
- getClasses() : List<MetaType>

Project

Het Project bevat informatie over de modules en classes die gemaakt zijn.

Attributen
- ModulesXMLPath
  Wordt gebruikt om de modules te vinden.
- DynamicClassesJarPath
  De JAR waarin de classes staan die door het bedrijf en de GUI worden gemaakt (denk aan Preconditions en Effects).

Property

Properties zijn de attributen van de klassen die door MetaType-instanties gedefiniëerd worden. Deze constructie is gebruikt om met de GUI dynamisch attributen aan klassen toe te kunnen voegen.

Attributen

- String name
  De naam van de Property. Per MetaType moet deze naam uniek zijn.
- Class type
  Het type van het attribuut.
- Annotation[] annotations
  Annotations worden door het bedrijf gebruikt en kunnen hier opgegeven worden. Annotations zjin meta-data die als flag fungeren.

User Manual

Stripes User Manual

Authors: Tom Lotgering, Sid Mijnders
Version: 1
Date: July, 10. 2009

This is the user manual of the "Stripes" (Tygron Bachelors-) project. This document is intended to help get you started quickly. For more information on how Stripes works and how to use it, we refer you to the javadoc, and the object design and "Eindrapport" documents.

Goal

The goal of this program is to enable the user to make a game model by defining elements (from the real world) and drawing relations between them. The possible types of elements and their relations can be viewed below. The game model can then be exported and used in a seperate game project. This way it should be possible to implement any kind of real-life situation one can think of.

GUI

Project menu

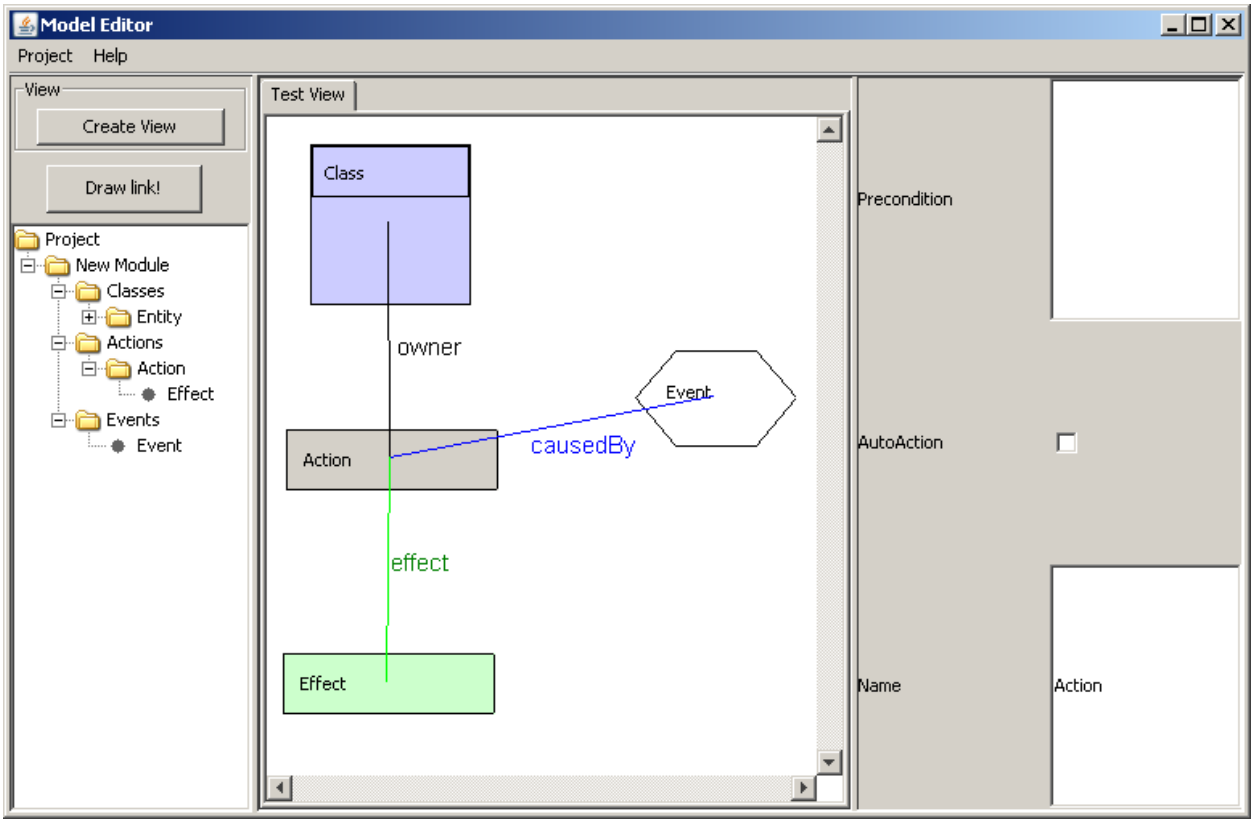
New Project A new project has to be made before elements can be added. After this has been clicked you are asked to give a name and location for the project. This will make an xml file at the specified location.
Open Project Asks to select a project xml file, then opens the selected project.
Open recent Project Opens the most recently saved project xml file.
Build Project! Exports the currently opened project to a jar file. The location and name of the exported jar file can be changed by selecting the project folder in the navigation tree, then changing the exportPath property.
Save Project Saves the project to the project xml file.
Close Project Closes the project. No changes can be made after.

Elements

There are four types of elements that can be defined within our GUI:

- Class
- Action
- Event
- Effect

The relations that are possible between them are depicted in the following picture (with exception of the "parent" link, which is a link between two Entities).



Also depicted are the property panel (on the right), the buttons and the navigation (on the left, containing a tree of elements).

Keywords:

[Class].Init

Where [Class] is the name of any Class in your game-model. This is a standard event that exists for every class defined in the model, and is automatically fired as part of the constructor of an instance of that type.

Preconditions:

All preconditions are evaluated in a context with the following objects, and as such, they can be used in your precondition definitions:
MetaEntity **owner**. The object that owns the action that contains the precondition. Will either be an instance of the class the action is linked to (thus a MetaEntity) or null in case of an AutoAction.
Object **sender**. The object that contains the event that triggered the action.
Object **args**. The argument supplied to the event that triggered the action.

Precondition syntax:

Use the following boolean operators to seperate statements in your preconditions:

& or |
(Do *not* use && or ||)

Curly braces ("{", "}") are used by the precondition parser to identify scope, instead of the usual parentheses.
Example: {true | false} & true

Due to a peculiarity in java's use of generics, take note of the following when writing preconditions:
Valid: owner.getM("field"). ...
Valid: ((MetaEntity)owner.<MetaEntity>get("field"). ...
Invalid: owner.<MetaEntity>get("field"). ... (this will generate an exception)

Changing properties:

1. Select an object in the GUI.
2. On the right side, change the (text) value of the property you want to change.
3. *Press Enter to commit the change!* (Checkbox-properties are automatically committed when changed.)

Changing relations:

Creating a relation:

1. Click the [Draw Link!] button.
2. Click on object A (in the view).
3. Click on object B (in the view). If it is a valid relationship, the relation is now created.

Removing a relation:

1. Select (click on) the object (in the view) that has the relation you want to remove. The relation should 2. now be visible (if both objects involved are present in the view).
3. Click on or near the relation. The relation should now be selected (marked by a small square in it's middle).
4. Press [Delete].

Code

All classes, methods and variables needed for using our system are documented using JavaDoc. In this section we list a couple of often-used Java statements needed to get started with using our system:

Importing a module:
MetaClassImporter.importProject("ExampleJAR.jar");

Instantiating a model-defined class:
MetaEntity entity = new MetaEntity("MetaTypeName");

Getting and setting MetaEntity values:

MetaEntity metaProperty = entity.getM("propertyName");
or:
Integer intProperty = entity.<Integer>get("propertyName");

entity.set("propertyName", someObject);

Defining an Event:
```
private Event init = new Event(this, "GUI.init");
public List<IActionEventListener> getInitListeners()
{
   return init.getListeners();
}
```

Firing an Event:
init.fireEvent(someArgument);

Example

A demonstration scenario is included in the original distribution, called ConstructItDemo. It is based on the following ConstructIt scenario:

In Construct.it heb je 7 project-ontwikkelaars en 1 gemeente in het spel zitten. Ieder heeft zijn eigen / individuele kaart en er is een collectieve kaart die voor iedereen hetzelfde is.
De spelers bouwen een nieuw project door wat grond te selecteren en vervolgends van onder af verdiepingen te bouwen, met ieder een aparte functie. Als ze klaar zijn drukken ze op 'Accept' en wordt het project op hun kaart getoond. Het is dan nog niet beschikbaar op de collectieve kaart. Als ze blij zijn met het project kunnen ze deze "Submitten", waarna het voor de gemeente zichtbaar wordt in een lijst. De gemeente krijgt dus in zijn lijst allerlei projecten ter goedkeuring van alle spelers. Als de gemeente ook vindt dat het goed is klikt deze voor dat project op "Permit". Hierna verschijnt het project bij iedereen op de collectieve kaart.

The demo game model models the sequence of steps required to build a structure, using a simple GUI mockup to facilitate interaction.

To run this demo: open the ConstructItDemo java project, and the StripesProject java project. *You will probably receive a warning that three of StripesProject's libraries could not be referenced. Correct this by pointing to the libraries found under \lib.* Compile Stripes, then compile ConstructItDemo. Now run the StripesProject GUI from your java IDE in Debug mode; your debug output will receive information necessary to debug any errors in the game model, when you try to build it.

To open the game model using the Stripes GUI, select Project > Open project > and select StripesProject\Modules\Demo.xml.

Build it using Project > Build Project. It has been tested to work; if it doesn't, it will most likely be caused by the relative paths used in the game model. These assume you are running the GUI from the IDE, using the original distribution's file structure. You should also verify the existance of the (freshly built) ConstructItDemo\ConstructItDemo\dist\ConstructItDemo.jar and dev\code\StripesProject\dist\StripesProject.jar.

Once the project is built, run the ConstructItDemo project, and click on the map and buttons in the appropriate order.

It is suggested to inspect the ConstructItDemo.MainForm class to see what scenario functionality is contained within the GUI, and what is contained in the game model. To this end, you should also familiarize yourself with the code found under ConstructItDemo\Effects.

Extending the Example

After you've concluded the previous example, now is the time to make some simple modifications to it. In this example extension, we will delay the placement of buildings on the collective map; after the permit has been granted, the user will need to press a button "Use Permits" to use any available permits and have those structures placed.
In order to accomplish this, follow the following steps *carefully*.

New Project: ConstructItDemo\Modules: **DemoExt**

Add an import to the root module:
..\..\..\ConstructItDemo\ConstructItDemo\dist\ConstructItDemo.jar

Add action: **VoegKnopToe**
Check the AutoAction checkbox

Add an existing code effect to VoegKnopToe: **AddUsePermitsButton**
Path = **..\..\..\ConstructItDemo\Effects\AddUsePermitsButton.java**

Add event "**GUI Init**" and link it to **VoegKnopToe**

Add event "**Use Permits**"

Add action **GebruikPermits**
precondition: **((MetaEntity)owner).<Status>get("status") == Status.permitted**
link to **Use Permits** event

Copy the existing ConstructItDemo module (root module > right click > add > module > copy >
ConstructItDemo\Modules**DemoModule.xml** > **root**)

Add to view: **Project**; **PermitOntwerp**; **ToonOpCollectieveKaart**; **BuildProject**

Drag **ToonOpCollectieveKaart**, **BuildProject** and link those to **GebruikPermits**
Remove the links from these two actions with **PermitOntwerp** (Select GebruikPermits, click on the link, press [Delete])

Add an existing code effect to PermitOntwerp:
..\..\..\ConstructItDemo\Effects\PermitProject.java

Build the project.

In the java ConstructItDemo project, open the file MainForm.java and find the line

stripes.runtime.MetaClassImporter.importProject("..\\Modules\\DemoJar.jar");

and replace **DemoJar** with **DemoExtJar**.

Run the project. Click the map, and buttons in appropriate order.

If you've made it this far, I salute you.