Eindverslag PSOPV

Visuele Programmeer IDE AxeSki

Auteur:
Matthijs Kaminski

Auteur:
Axel Faes

Begeleider:
Jonny Daenen

Opdrachtgever:
Raf Van Ham

4 juni 2015

Inhoudsopgave

1 Executive Summary

AxeSki is een visuele programmeer omgeving. Het hoofddoel van de IDE is om event-based programma's te kunnen schrijven. Een gebruiker kan standaard programma's maken net zoals in andere programmeertalen. Een bijkomend voordeel is dat de gebruiker goed leert hoe event-based programming werkt. De IDE stelt dit concept eenvoudig voor zodat zelfs nieuwe programmeurs snel programma's kunnen schrijven. AxeSki biedt geen restricties, zoals een gelimiteerd aantal processen en laat gebruikers toe complexe programma's in elkaar te steken.

De gebruiker werkt voornamelijk met **Events**, dit zijn informatiepakketten die in een programma rondgezonden kunnen worden tussen verschillende entiteiten. Deze events worden aangeduid door een unieke naam. De informatie die erin zit wordt ook aangeduid door een unieke naam binnen het event.

Vervolgens kan de gebruiker met behulp van programmeerstructuren een **klasse** opbouwen. Een klasse is een entiteit binnen het programma die kan reageren op bepaalde Events, en er ook zelf kan uitsturen. Deze programmeerstructuren omvatten de standaard programmeer constructies zoals een while-loop, if-condities, operaties, variabelen, etc. Tijdens het aanmaken van de klasse moet de gebruiker niet weten waar de binnenkomende events vandaan komen, maar wel hoe deze events afgehandeld moeten worden. Een klasse kan ook terug events verzenden. Opnieuw moet de gebruiker zich niet bezighouden naar wie de events verzonden worden, maar indien nodig wel met welke informatie deze events verzonden worden.

De blokken waarmee de gebruiker het programma opbouwt, hebben een neutrale kleur. De verschillende vensters, het klasse-view, het frame-view en het event-view zijn eenvoudig in gebruik. Hierdoor heeft AxeSki een **professionele look**. Om bruikbaarheid van de IDE te bevorderen kan de gebruiker de taal aanpassen. Er zijn twee ondersteunde talen, namelijk Engels en Nederlands.

Deze klassen kunnen geïnstantieerd worden en kunnen verbonden worden met elkaar om de events door te sturen. Een instantie is een black box, de gebruiker moet zich, wanneer hij instanties maakt, niet bezighouden met wat deze black box doet, maar wel hoe verschillende instanties met elkaar communiceren. De soorten Events die een instantie kan opvangen en versturen zijn deze die gespecifieerd zijn in de bijhorende klasse.

Om een programma op te bouwen beschikt de gebruiker over verschillende tools. De gebruiker kan ten alle tijde een programma **opslaan** zodanig dat zijn voortgang behouden blijft. Deze opslag gebeurt in een leesbaar formaat (XML) zodanig dat de gebruiker zelf deze bestanden zou kunnen aanpassen. In AxeSki is een editor ingebouwd om het opslagbestand te bewerken. De gebruiker heeft dus de keuze om te werken met de visuele IDE, of om het ruwe bestand aan te passen.

Een andere beschikbare tool is **debugging**. De gebruiker kan stap voor stap door het programma heen lopen. Zodanig kan de gebruiker gemakkelijk kijken of het programma de gewenste uitvoer heeft. De gebruiker kan ook een stoppunt definiëren in het programma zodat de uitvoer zal pauzeren op dat punt. Hierdoor kan de gebruiker eventuele fouten in een programma gemakkelijker opsporen.

Een andere hulpvolle tool is **typechecking**. De gebruiker kan in zijn programma variabelen aanmaken die informatie bevatten. Deze informatie behoort tot een bepaald type, zoals een getal of string. AxeSki typechecking zorgt ervoor dat de gebruiker geen operaties tracht uit te voeren met een niet-compatiebele variabele. De ondersteunde types zijn numerieke waarden, booleaanse waarden en strings.

Om de IDE interactief te houden is een **canvas** geïmplementeerd. Instanties van klassen worden hierin getoond en kunnen tijdens het runnen van een programma hier events opvangen van de gebruiker, zoals toetsaanslagen of een muisklik. Een klasse kan ook kostuums bevatten. Dit laat toe om een instantie te koppelen aan een visuele afbeelding. Dit laat toe om de programma's die de gebruiker kan maken meer mogelijkheden tot feedback te geven.

Achter de IDE zitten verschillende complexe design patronen verwerkt. AxeSki simuleert parallele uitvoering waardoor alle instanties en events die opgevangen simultaan uitgevoerd worden. Dit gebeurt door threading te simuleren, de uitvoering wordt na een stap doorgegeven aan de volgende thread die uitgevoerd moeten worden. Echte systeem threads gebruiken is veel te zwaar voor AxeSki. De IDE moet vele threads kunnen runnen en zal dus veel context switches hebben, om het snel te houden hebben we zelf threads geïmplementeerd. Het bewegen van de blokken (de drag-and-drop) is zelf geïmplementeerd. Het correct nesten en verslepen gebeurt door zelf-geïmplementeerde logica. Er moet communicatie bestaan tussen de blokken om typechecking te kunnen weergeven. Zo kunnen hoge niveau blokken zoals functies zijn"kind" blokken updaten, en de kind blokken kunnen hun ouder laten weten dat ze aangepast zijn.

Alhoewel AxeSki functioneel is en meer dan de geplande features bevat, is er toch nog ruimte voor uitbreidingen. De IDE is zo gebouwd zodanig dat aanpassingen en toevoegingen eenvoudig te implementeren. Alle onderdelen, zoals de uitvoering en het visuele uitzicht zijn volledig losgekoppeld.

Dit project heeft ons veel geleerd. We kunnen beter omgaang met grotere software projecten. De belangrijkheid van documentatie is nog duidelijker geworden. Grote projecten vereisen goede documentatie zodat alle bouwstenen van het project correct in elkaar gestoken kunnen worden. We hebben regelmatig afgesproken met onze begeleider zodat we op de correcte weg bleven.

2 Beschrijving van het project

2.1 Einddoel van applicatie

Het einddoel van de applicatie is een visuele IDE met een professioneel uiterlijk en eenvoudige werking. Hierin kunnen event-based programmas op een intuïtieve en eenvoudige manier uitgewerkt worden. De uitvoering van blokken kan visueel gevolgd en onderbroken worden in de debug modus. Het doorgeven van Events tussen Instanties kan via wires in het Frame-view. Een visueel canvas kan gebruikt worden om instanties en veranderingen ervan te tonen. Ook kunnen hierin input Events worden gegenereerd op instanties.

2.2 Interpretatie van opgave

Onze interpretatie zorgt ervoor dat de gebruiker op verschillende niveau's programma's kan maken in de IDE. Eenderzijds kan de gebruiker de flow van het programma opbouwen door middel van blokken met elkaar te verbinden. Deze verbindingen worden Events genoemd die uitgelegd staan in Sectie 3.1. Deze flow wordt gemaakt door grote blokken van een bepaald type die bv. een telefoon of drukknop voorstellen, met elkaar te verbinden, dit noemen we Instanties van een type. Door deze flow te maken kan de gebruiker op intuïtieve wijze een programma opbouwen. Deze flow toont aan wat er gebeurt en wanneer iets gebeurt.

Anderzijds kan de gebruiker de types van de grote blokken (zie Sectie 3.1) opbouwen, dit noemen we een Klasse. Dit gebeurt door een programma te maken van een opeenvolging van kleine blokken. Deze kleine blokken stellen algemene programmeer structuren voor zoals een while-loop. Door deze opbouw kan de gebruiker zien hoe iets werkt.

Uiteindelijk kan de gebruiker het gemaakte programma runnen. Hierbij kan de gebruiker input events (met behylp van toetsaanslagen en de muis) sturen naar de grote blokken.

Door het opdelen van het programma naar een niveau waar de gebruiker het wat en wanneer maakt van een programma en een niveau waar hij de hoe maakt, is de IDE laagdrempelig en eenvoudig in gebruik. Er is een console geïmplementeerd. In deze console kan de gebruiker ook tekst afprinten. Hierdoor heeft de gebruiker een visueel canvas en de console waar tekst afgeprint in kan worden. Deze console toont ook systeem output, zoals runtime errors.

Dit is ons eindverslag. Eerst wordt er een algemeen beeld gegeven over de IDE waaronder de functionaliteit. Hierna wordt het ontwerp, de gebruikte algoritmes en datastructuren, bekeken. Uiteindelijk geven we mogelijke uitbreidingen voor de IDE aan. In de bijlagen vind u de geïmplementeerde kleine blokken, een handleiding, reflectie en een logboek. Ten opzichte van het analyseverslag zijn er verschillende toevoegingen gebeurt. In het analyseverslag was de GUI,

en de drag-and-drop niet uitgewerkt. De verdere analyse is hetzelfde gebleven. De algoritmes en datastructuren die bij de analyse gemaakt zijn, bleken goed en efficiënt te werken.

2.3 Noden van de opdrachtgever

Naast de algemene features van de applicatie wenst de opdrachtgever dat er aandacht wordt besteed aan volgende punten. Deze staan gerangschikt van meest prioritair naar minder prioritair.

1. De opdrachtgever wenst een professioneel uiterlijk.
2. De applicatie moet beschikbaar zijn in verschillende talen.
3. De IDE moet bruikbaar zijn door een persoon met beperkte programmeer kennis.
4. De opdrachtgever wenst dat er een debug modus aanwezig is waarin het programma vertraagd wordt afgespeeld en de flow van het programma duidelijk wordt aan de gebruiker.
5. Een door de gebruiker gecreeërd programma moet opgeslaan worden in een opslag formaat dat nog leesbaar is in tekstformaat.
6. De opdrachtgever wenst dat er geen globale variabelen aanwezig kunnen zijn in het programma.

2.4 Bestaande Software

In de volgende sectie worden enkele bestaande software producten besproken. Er wordt uitgelegd welke elementen we overgenomen hebben en welke elementen niet overgenomen zijn.

Sratch

Scratch [1] is een visuele porgrammeer IDE gemaakt door MIT. Scratch focused meer op kinderen en beschikt daardoor ook over minder complexe programmeer structuren.

voorstellen van een Sprite. Een sprite komt in onze applicatie overeen met een instantie van een Klasse. In onze applicatie heeft een Klasse ook een visuele voorstelling en kan deze ook meerdere uiterlijken hebben. Ook kan een Sprite tekstballonen tonen in het canvas, dit is niet de prioriteit in onze applicatie en dus niet geïmplementeerd. Het plaatsen en dupliceren van een sprite zal bij ons vervangen door het toevoegen van een of meer Instanties van een reeds bestaande Klasse.

Achtergrond van het canvas. Scratch geeft de mogelijkheid om de achtergrond van de canvas ook te behandelen als een sprite. Deze feature is niet van belang voor onze omgeving aangezien we focussen op het event-driven programmeren.

Programmeer Blokken in een Sprite. Scratch biedt een hoop mogelijkheden aan om acties te doen met een Sprite. Uit de motion-blokken hebben we enkel de mogelijkheid om de x- en y-positie van een instantie van een Klasse te veranderen. Uit looks hebben we de mogelijkheid overgenomen om het uiterlijk te veranderen in een eerder ingevoegde appearance. Uit de sound en pen blokken hebben we niets overgenomen.

Er is de mogelijkheid om een variable te creeëren in een Klasse, dit wordt gezien als een private member variabele van die Klasse.

Uit events hebben we de broadcastblok overgenomen. In onze applicatie heeft het echter de betekenis dat een instantie van de Klasse een uitgaande poort heeft voor dat specifieke event en niet alle instanties van Klassen die op dat event geabboneert zijn het event ontvangen. Ook het abboneren op een event gebeurt bij ons anders zoals besproken in Sectie 3.1 Events.

Uit de control blokken: we hebben de standaard controle structuren zoals een while, if-else, forever.

Sensing blokken zoals het checken op collision hebben we niet overgenomen.

Uit de operators nemen we alle functionaliteit over: logica, String, random en rekenkundige operaties.

Uit de categorie More blocks van defines nemen we de functionaliteit over, echter wordt dit voorgesteld bij ons door interne functies. Hierdoor is het ook makkelijker om de flow van het programma in een Klasse te volgen. Bij Scratch is dit onoverzichtelijk en dit willen we vermijden.

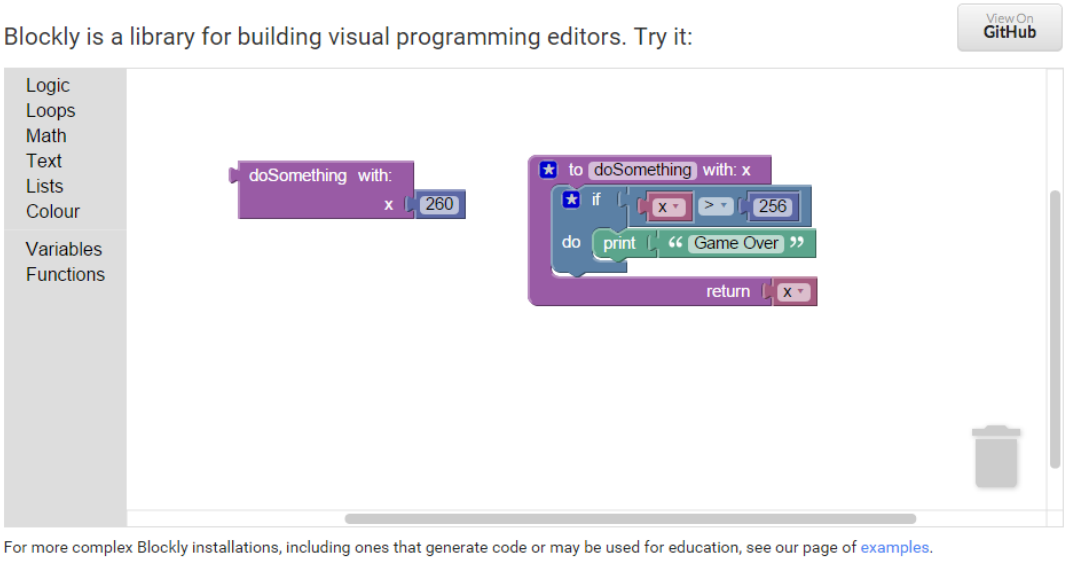
Tenslotte hebben we de blokken een professionelere look dan de blokken van Scratch gegeven. Dit gebeurt door gebruik te maken van strakkere lijnen en neutrale kleuren.

Blockly

Blockly [2] is een visuele programmeer IDE gemaakt door Google. Blockly definiëert een imperatieve programmeertaal en is niet event-based zoals onze programmeer IDE. De gemaakte code kan geconverteerd worden naar een gekozen formaat. Dit kan oa. Javascript of Python zijn.

De blokken zijn gemodelleerd naar puzzelstukjes. Dit maakt het gemakkelijk

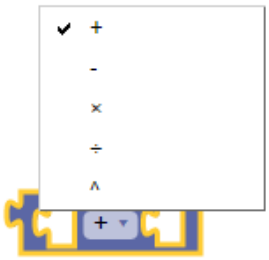
om te zien hoe de blokken in elkaar gestoken kunnen worden. Blockly is gericht op nieuwe programmeurs. Het versimpelt verschillende programmeerconcepten. Er zijn enkel globale variabelen en lijsten zijn niet nul-based maar één-based. Variabelen zijn niet case-sensitive en kunnen bestaan uit allerhande tekens (inclusief spaties).



Figuur 1: *Blockly.*

Functies Blocky laat toe om functies te creeëren. Functies kunnen parameters meekrijgen. Deze functies zijn globaal en kunnen vervolgens op elke plaats in het programma opgeroepen worden. Deze functies zijn gelijkaardig aan de functies van ons project. Echter behoren onze functies tot een bepaalde Klasse.

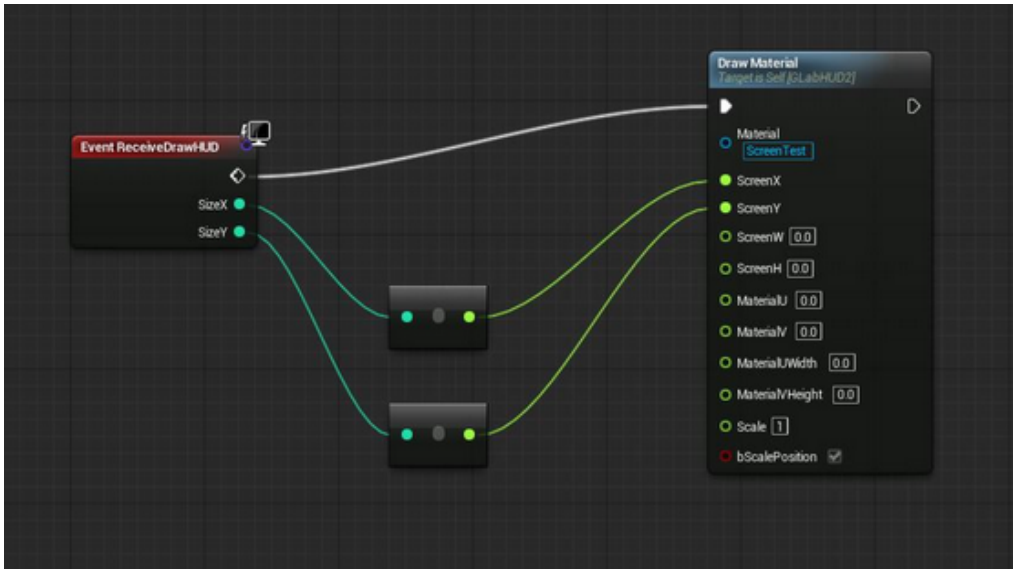
Operator Blokken Ons concept om operatoren toe te passen is gelijkaardig aan het concept dat Blockly gebruikt. Er is slechts 1 operator blok voor de binaire arithmische operatoren, alsook één operator blok voor de binaire logische vergelijkings operatoren.



Figuur 2: *Blockly operators.*

Unreal Engine 4: Blueprints

De Unreal Engine 4 [3] is een game engine die uitgebracht is door Epic Games. In de Unreal Engine 4 is een visueel scripting systeem ingebouwd. Dit wordt Blueprints genoemd. Het laat toe om volledige gameplay elementen visueel te scripten via een node-based interface. Het is mogelijk om een volledige game te implementeren in Blueprints.



Figuur 3: *Unreal Engine 4.*

De Unreal Engine gebruikt standaard enkel C++ code, ook voor de scripting. Blueprints compilen achterliggend ook naar C++ code. Hierdoor is er geen interpreting nodig van de nodes en is er ook geen snelheidsverlies.

Nodes In de Unreal Engine kunnen nodes verbonden worden met elkaar. Dit duidt op een opeenvolging, net zoals er in een imperatief programma de uitvoering van ene instructie overgaat naar de andere. Dit is te zien als de witte lijn. Vervolgens kunnen parameters doorgegeven worden, dit zijn de gekleurde lijnen. Dit kan vergeleken worden met ons systeem in het wireFrame. In ons systeem worden echter events met elkaar doorverbonden en niet de functies. [3]

Events Het beginpunt van een flow van nodes in de Unreal Engine is een event dat ontvangen word. Dit kan een eigen gemaakt event zijn, of dit kan een standaard events zijn zoals te zien in de afbeelding.

3 Diepgaande Beschrijving van het project

3.1 Diepgaande uitleg

In deze sectie wordt op hoog niveau de IDE beschreven. Dit als korte inleiding voordat de algoritmes en datastructuren vermeld worden. Hierdoor kan verschillende terminologie verduidelijkt worden.

Zoals al eerder vermeldt kan een gebruiker de programma flow kunnen opbouwen in het deel van de IDE dat we het Frame-view noemen. Hierin kunnen instanties van klassen die eerder gedefineerd werden de mogelijkheid worden geboden om informatie aan elkaar door te geven. Deze informatie noemen we een Event. Wanneer een instantie een Event verstuurd en wat hij met een ontvangen Event doet is beschreven in de klasse waartoe hij behoort.

In de volgende paragrafen zal een diepgaande uitleg worden geven over wat een Events is en hoe de gebruiker er gebruik van maakt alsook hoe hij een klasse kan definiëren en welke standaard eigenschappen een klasse in de IDE bevat.

Events

Om onderlinge communicatie tussen Instanties van Klassen voor te stellen, gebruiken we een **Event**. Een Event kan al dan niet informatie bevatten. Een Event zonder informatie kan beschouwd worden als een trigger. De informatie dat een Event kan bevatten kan uit meerdere delen bestaan. De infomatie kan dus bestaan uit meerdere primitieve types (int, string of boolean). Elk deeltje in die informatie noemen we een variable. Met elke variable wordt een naam geassocieerd. Deze kan de gebruiker dan gebruiken om de variabele uit een Event op te vragen. Ook het Event zelf moet een ID hebben dat als type geldt.

Eens de gebruiker een Event heeft gedefineerd in de daarvoor voorziene omgeving kan hij er verder in de IDE gebruik van maken. Dit doet hij dan door deze Event te selecteren in een dropdown menu van bepaalde blokken, dit maakt

een EventInstance aan. Hij kan dus vanaf dan een Event het eerder gedefineerde type aanmaken en invullen met de informatie die hij wenst mee te geven. Het zenden van een EventInstance door een klasse noemt een emit. Naar welke instanties van klassen het EventInstance wordt verzonden, kan worden bepaald in het Frame-view van de IDE.

Er is een aparte view waarin alle Instanties van Klassen als blokjes getoond worden, dit view noemen we het **Frame-view**. Deze blokjes bevatten inkomende en uitgaande poorten. Deze stellen respectievelijk de evenementen voor die een Instantie wil ontvangen en de evenementen die het uitzendt. Er kunnen verbindingen gemaakt worden tussen de uitgaande poorten van een instantie en de inkomende poorten van een andere instantie, waarbij de respectievelijke evenementen hetzelfde type hebben.

Dit aparte view is echter de begin positie van alle gewenste Instanties van de aangemaakte Klassen. De gebruiker heeft de optie om de verbindingen al dan niet te tonen. Een extra view, het **Canvas-view** het bewegen van de instanties toont. De gebruiker kan zo de flow van Events bekijken.

Nieuwe **Events kunnen aangemaakt worden** door de gebruiker in een aparte sectie van de IDE. Een Event moet een type hebben, vervolgens kan er informatie meegegeven worden aan dit Event. Deze informatie is een POD (plain old data) die opgebouwd wordt door de gebruiker. Hierin zal elke variable een unieke naam en specifiek type hebben.

Er zijn **standaard Events** beschikbaar zoals oa. KeyA, MousePress, Start, enz. Deze events zijn voorgedefineerd en dienen om interactie te hebben met het visuele canvas.

Klassen

Een Klasse kan worden vergeleken met eenderzijds een Sprite in de visuele programmeeromgeving Scratch [1] en anderzijds een klasse uit een object geörienteerde taal zoals Java. Het verschil in deze applicatie is dat de Instanties van een Klasse expliciet aangemaakt worden in de Frame-view. Een Klasse bestaat uit: input Events, Handlers voor die Events, functie definities en member variabelen. Een Klasse kan worden voorgesteld in het Frame-view. Deze appearance kan door een functie in de Klasse worden veranderd. Een Instantie kan dan in het Frame-view beslissen van welke andere Instanties het die input Events ontvangt of naar welke instaties hij Events verstuurd.

Een Klasse kan **Events ontvangen**. Dit werd besproken 3.1. Het afhandelen van een Event gebeurt door een handler die het event binnen krijgt. Het raadplegen van de inhoud van een Event, kan doormiddel van een accessblok. Deze kan een variable accessen die in het binnenkomende event zit.

Een Klasse kan **Events emitten**. Dit kan met behulp van een Emit blok. Hierin moet een Event worden geselecteerd. Als een Event informatie bevat zal deze ook hier moeten worden ingevuld.

Een Klasse kan ook een overzicht hebben met alle Events die erdoor worden geëmit, dit is getoond in het Wire-view, indien een instance bestaat van die Klasse.

Een uitbreiding van de visuele omgeving ten opzichte van andere IDE's is toe te laten om **functie aanroepen** te maken binnen een Klasse. Oorspronkelijk was het idee om dit voor te stellen met een lijn die twee functieblokken zou verbinden. Bij een Klasse met veel interne functie aanroepen wordt dit echter onoverzichtelijk.

Een functie oproep vanuit een andere functie wordt voorgesteld door blokje. Dit blokje bevat de naam van de functie die kan worden opgeroepen. Alsook zijn input parameters en return waarde. De parameters worden by-value doorgegeven aan de functie. Hierin kunnen variable gebruikt worden die als constante worden doorgegeven. Een variable is data dat een bepaald type heeft zoals een number of string. Er kan geschreven worden naar een variable en de variable kan gelezen worden. De onderkant van een functieaanroep blok bevat een leeg vakje voor de return waarde. Hier kan een variabele aan gekoppeld worden om deze waarde op te vangen.

Member Variabelen zijn variabele die gelden per Instantie van een Klasse. Deze kunnen bijvoorbeeld de toestand van van de Instantie van een Klasse die een lamp voorstelt in het canvas voorstellen.

Blokken

Een blok is een blokje dat de gebruiker kan plaatsen in het programmeer venster van de IDE. Deze blokken kunnen alles voorstellen, bv. variabelen, types, control-flow, functions, enz. . Deze zijn onderverdeeld in verschillende categorieën. De verdere uitleg met betrekking tot deze categorieën en de blokken die erbij horen staan in bijlage op Sectie E.

4 Functionaliteit

Alle features die we wilden implementeren zijn geïmplementeerd. Alle noden van de opdrachtgever (Sectie 2.3) zijn geïmplementeerd. Er zijn verschillende uitbreidingen die we toch graag geïmplementeerd hadden, echter zijn deze features geen core functionaliteit. Deze features zijn niet geïmplementeerd omwille van zowel tijdsbeperkings als de voorkeur om het programma zorgvoldig af te werken. Deze features worden besproken in Sectie 12.

Eerst worden de verschillende features opgesomd, hierna volgt een meer gedetailleerde beschrijving van de features.

4.1 Features:

- professioneel uiterlijk (neutrale kleuring)
- beschikbaar in meerdere talen
- debug modus: stappen doorheen het programma
- inladen en opslaan van programma's in een leesbaar formaat
- events aanmaken
- klassen aanmaken
- programma opbouwen via blokken
- blokken verwijderen
- instanties maken
- programma uitvoeren

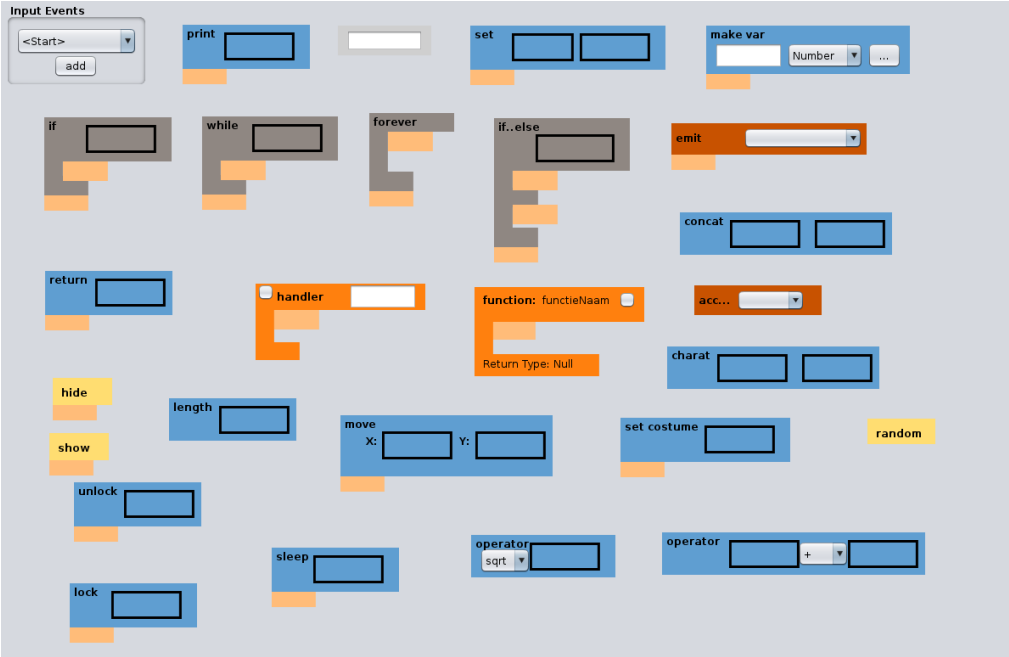
4.2 Extra Features:

- selectie plaatsing blokken
- breakpoints plaatsen
- typechecking
- sleep blok
- data editor
- kostuums opslaan
- verplaatsen aanknopingspunt wires
- naam van een klasse aanpassen
- doorzichtigheid blokken
- hulp menu
- event filter

4.3 Uitleg eatures:

Professionele look

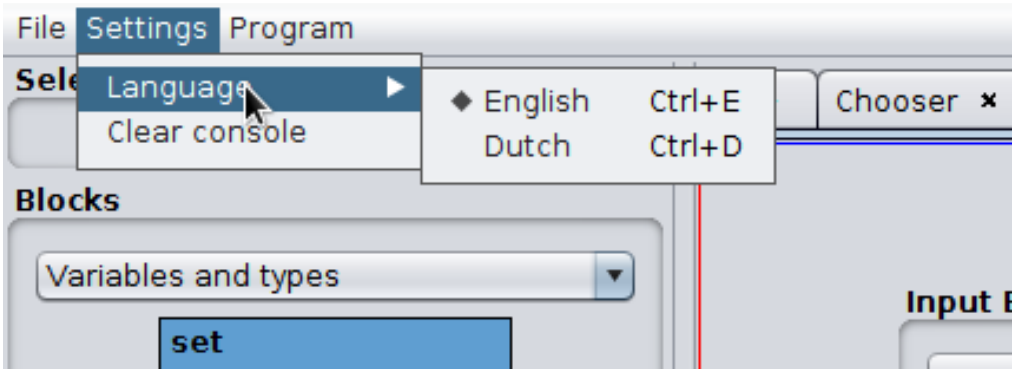
De IDE heeft een **professioneel** uiterlijk zoals te zien in Figuur 4. De verschillende blokken die de gebruiker kan plaatsen hebben neutrale kleuren.



Figuur 4: *Alle geïmplementeerde blokken.*

Meerdere talen

De IDE is beschikbaar in zowel het **Nederlands als het Engels**. Hoe het wisselen tussen de verschillende talen gebeurt is te zien in Figuur 5



Figuur 5: *Selecteer de taal.*

Inladen en opslaan

De gebruiker kan programma's **opslaan en inladen** in een leesbaar formaat, dit gebeurt door middel van het XML-formaat. Het inladen zal de IDE niet doen crashen op een verkeerde layout, er kan wel een eigen error message uitgeprint worden naar de console zodanig dat de gebruiker weet wat er fout gaat. Er is ook de mogelijkheid om een nieuw project te starten, dit verwijdert alle hudige niet opgeslagen progressie.

Debug modus

Er is een **debug modus** aanwezig. Deze laat toe om stap voor stap doorheen het programma te lopen. Er zijn verschillende extra features geïmplementeerd met betrekking tot de debug modus, deze staan uitgelegd in Sectie 4.4.

Events

De gebruiker kan nieuwe **events aanmaken en verwijderen**. De inhoud van deze events kan de gebruiker aanpassen. Hij kan nieuwe informatie toevoegen, informatie verwijderen of het type van de informatie aanpassen.

Klassen

Klassen kunnen aangemaakt en verwijderd worden.

Blokken plaatsen

Het Klasse-view laat toe om **blokken** te plaatsen. Deze blokken kunnen aan de linkerkant geselecteerd worden. In het dropdown menu staan verschillende categorieëen met verschillende blokken. Blokken kunnen vervolgens volgens hun regels genest worden in elkaar. Het blokje "Input Events" toont alle input events van een bepaalde klasse. Via het dropdown menu kunnen input events toegevoegd worden, via het kruisje kan dat input-event verwijderd worden. Globale

variabelen zijn niet toegelaten in de IDE. De gebruiker kan nieuwe member **variabelen** aanmaken. Een member variabele kan overal in een klasse gebruikt worden. Elke instantie heeft zijn eigen instanties van de member variabelen. Er kunnen verschillende **kostuums** toegevoegd worden aan een klasse. Deze kan als primair gezet worden, zodanig dat elke instantie van een klasse begint met dat kostuum.

Blokken verwijderen

De gebruiker kan een **blok verwijderen** door middel van een optie die tevoorschijn komt via een popup. Deze popup kan getoond worden als de gebruiker de rechtmuisknop indrukt. De gebruiker kan geen geneste blokken apart verwijderen. Het verwijderen van een blok zal altijd van toepassing zijn op de meest externe blok, deze verwijdert dan ook alle geneste blokken. De reden dat geneste blokken niet verwijderd kunnen worden is duidelijkheid voor de gebruiker. Blokken die genest zijn vormen een geheel. Een actie zoals verwijderen heeft dan ook betrekking op het geheel.

Instanties

In het Frame-view kan de gebruiker **instanties** aanmaken van een Klasse. De gebruiker kan input-events verbinden met output-events van hetzelfde type. Het Canvas-view toont alle instanties en hun geselecteerde uiterlijk. De gebruiker kan deze instanties en verbindingen ook weer verwijderen.

Uitvoering

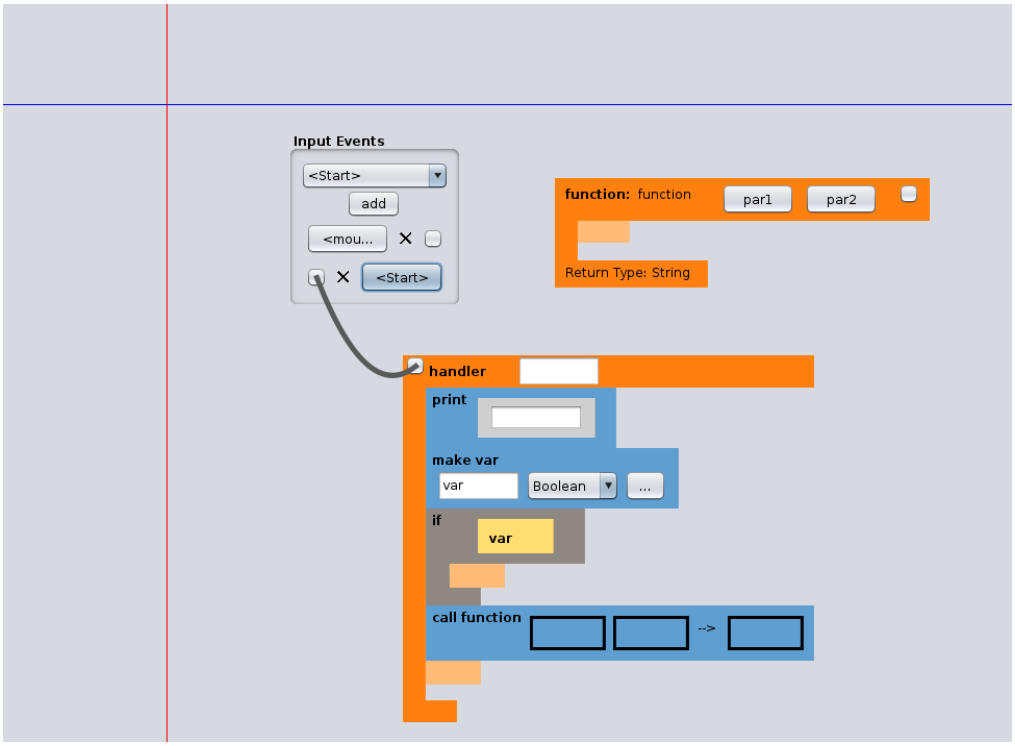
De gebruiker kan een programma **compileren, uitvoeren, stoppen of stappen** (een stap uitvoeren). Deze actie is mogelijk voor zowel de debug modus als de gewone modus.

4.4 Uitleg extra features:

We hebben verschillende extra features geïmplementeerd. Sommige features hiervan zijn slechts kleine toevoegingen, andere zijn grotere toevoegingen.

Selecteren blokken

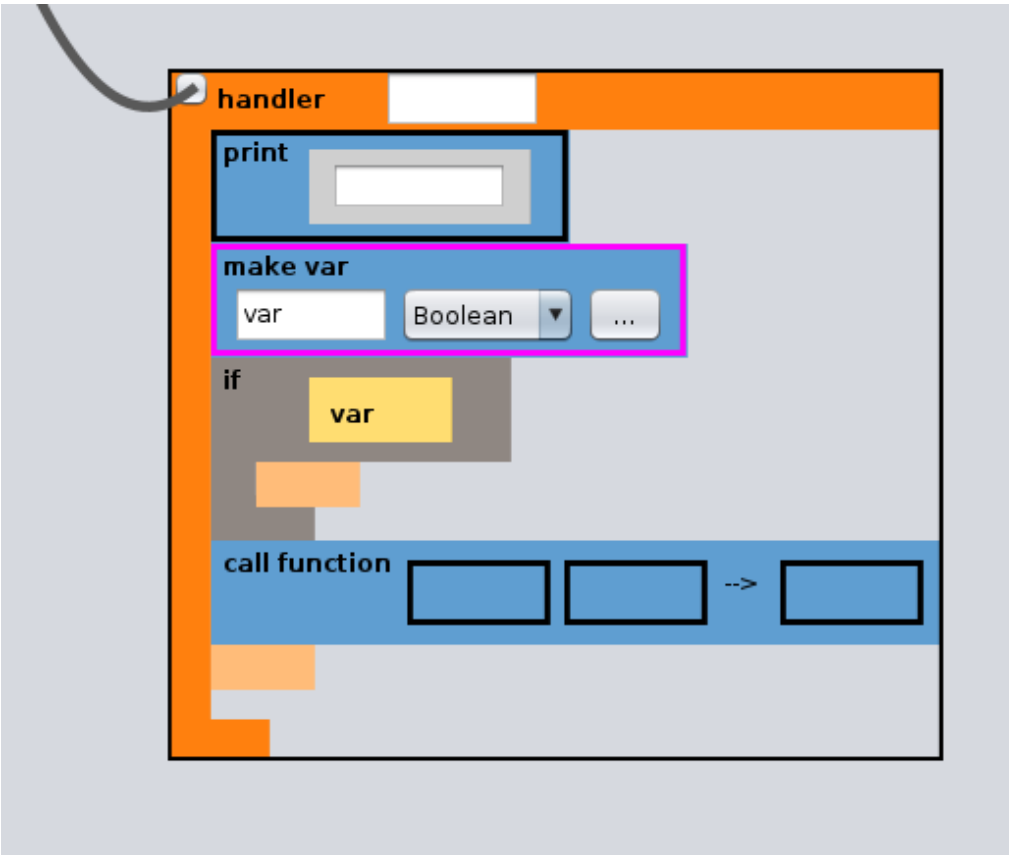
In zowel het Klasse view als het Frame view kan de gebruiker in de linkerbalk blokken selecteren. Vervolgens zal elke muisklik op het Klasse venster een nieuwe blok van het geselecteerde type maken op de muispositie. Beide views zijn ook "oneindig" groot (gelimiteerd door de grote van een int). De blauwe lijn stelt de $x - as$ voor, de rode lijn stelt de $y - as$ voor zoals te zien in Figuur 6



Figuur 6: *Infinite view*

Breakpoints plaatsen

De **debug modus** is uitgebreid met de mogelijkheid om break-points aan te duiden. Deze kunnen gezet worden op een blok. Als de gebruiker doorheen het programma stapt/loopt in debug modus, gaan zwarte omlijningen aanduiden waar de uitvoer zich momenteel bevindt zoals te zien in Figuur 7. De `break` stopt enkel het proces dat de `break` tegenkomt, de andere processen blijven doorlopen.



Figuur 7: *Debug view*

Typechecking

Er is ook **typechecking** toegepast. Dit gebeurt op drie niveau's. Eenderzijds gebeurt typechecking visueel. Dit wordt getoond door een rode border (Sectie 8.3). Anderszijds zal de compiler stellen dat compilatie faalt indien het programma niet volledig is. Als er toch fouten door deze checks komen, is er nog runtime errorchecking. Dit kan optreden wanneer een functie aanroep gebeurt naar een niet-bestaande functie. Er wordt dan ook gespecifieerd wat de fout exact is.

Sleep blok

Er is een `sleep` blok geïmplementeerd om een proces te laten slapen voor het gespecifieerde aantal milliseconden.

Data editor

We hebben een data editor geïmplementeerd zoals te zien in Figuur 8. De data editor laat het programma zien in zijn XML-vorm. De gebruiker kan deze dan vrij aanpassen. Als de gebruiker terug wisselt naar een ander view zullen de aanpassingen in de data editor ook zichtbaar zijn in de visuele views. De Data editor heeft ook syntax highlighting. Dit is geïmplementeerd via een externa library [4].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<program>
    <events/>
    <classes>
        <class name="ff">
            <events/>
            <emits/>
            <memberVariables/>
            <handlers>
                <handler event="" name="" x="226.0" y="189.0">
                    <block x="23.0" y="32.0"/>
                </handler>
            </handlers>
            <functions/>
            <floatingBlocks>
                <block x="108.0" y="284.0">
                    <return x="0.0" y="0.0"/>
                </block>
            </floatingBlocks>
            <costumes/>
        </class>
    </classes>
    <wireframe>
        <instances/>
        <wires/>
    </wireframe>
</program>
```

Figuur 8: *Data Editor*

Kostuums opslaan

De kostuums van een klasse worden ook opgeslagen in dezelfde locatie als de XML-file. Zodanig kan een project gebruikt worden op verschillende computers zonder dat de XML-file aangepast moet worden.

Aanknopingspunt wire

Het aanknopingspunt voor een wire kan wisselen van links naar rechts (en andersom) zoals te zien in Figuur 9. Dit verhoogt de leesbaarheid bij grotere programma's.

Figuur 9: *Switchen aanknopingspunt*

Naam klasse aanpassen

De naam van een klasse kan aangepast worden. Door de rechtermuisknop te gebruiken op het tab van die klasse, kan zijn naam aangepast worden.

Doorzichtigheid blokken

Als de gebruiker een blok vast neemt met zijn muis, zal de blok doorzichtig kleuren. Dit maakt het duidelijker voor de gebruiker waar de blok terecht zal komen.

Hulp menu

Er is een **hulp menu** toegevoegd. De gebruiker kan via de rechtermuisknop een hulp-dialoog openen over de blok die zich momenteel onder de muis bevindt. Deze dialoog geeft een korte uitleg over de blok zoals zijn functionaliteit.

Event filter

In het Frame view is een filter geïmplementeerd. Hierbij kan de zichtbaarheid van events geregeld worden. Verbindingen tussen instanties kunnen op deze manier zichtbaar/onzichtbaar gemaakt worden. Dit wordt geregeld per event type. Dit staat verder beschreven in Sectie 10.3.

4.5 Evaluatiecriteria

In het analyseverslag stonden enkele evaluatiecriteria waaraan de applicatie moest voldoen. Enkele van deze criteria hebben betrekking tot features die geïmplementeerd zijn. Andere criteria stellen iets over de uitvoering van de IDE.

Zo moest een verzonden event gelijktijdig opgevangen worden door de geabboneerde instanties. Na elke primitieve stap zullen alle verzonden events afgehandeld worden. Hierdoor worden die ook gelijktijdig opgevangen. Aangezien de IDE zelf threading simuleert gebeurt de uitvoer ook concurrent, en worden er verschillende processen gesimuleerd. Hierdoor zal een proces met een eeuwige loop niet de Virtual machine doen crashen.

Er worden geen limitaties (vanuit de IDE) opgelegd op het aantal klassen, events, instanties, processen dat een gebruiker kan maken.

De IDE ondersteund volledige functies, hierdoor is **parameter passing, return waardes en recursiviteit** mogelijk. Een lock kan niet gezet worden indien een ander proces al een lock gezet heeft op dezelfde instantie. Hierdoor zijn deadlocks niet mogelijk en indien de gebruiker locks gebruikt, kunnen geen racing conditions optreden.

Bij een runtime fout zal enkel het proces dat de fout genereerde afbreken.

5 Problemen

5.1 Problemen

Een probleem dat we ondervonden was het **drag-and-drop** systeem maken. We hadden de mogelijkheid om drag-and drop functionaliteit van Java te gebruiken [5]. Echter is het doel van de drag-and drop van Java anders dan hetgeen de IDE moet hebben. De Java drag-and drop wordt gebruikt om componenten te verplaatsen tussen verschillende paneels, de IDE moet componenten verplaatsen en nesten binnen eenzelfde paneel.

Hierdoor hebben we ervoor gekozen om **zelf drag-and-drop te implementeren**. De implementatie details vind u in Sectie 10.4. De IDE berekent op basis van waar de gebruiker een blok sleept, of (en hoe) deze blok genest moet worden.

Een ander probleem was het invoegen van de implementatie van het opslaan en het compileren. Het opslaan (en het compileren) van blokken zijn losgekoppeld van de blokken zelf (de logica). Hoe dit gerealiseerd werd, was toch even een probleem waar over nagedacht moest worden. Hiervoor werd gekozen om het **visitor** patroon toe te passen [6].

De applicatie plaatst geen limitaties op het aantal processen dat tegelijkertijd kan lopen. Enkele duizenden processen tegelijkertijd laten lopen is geen probleem voor onze applicatie, echter indien de gebruiker een miljoen processen tracht te laten lopen zal de applicatie beginnen haperen, of indien de gebruiker nog meer processen laat lopen, kan de applicatie vasthangen. Echter reageert onze applicatie net zoals een computer reageert. Deze kan ook vele processen tegelijkertijd laten lopen, maar teveel processen zal de computer doen haperen.

Bij het uitprinten van grote hoeveelheden tekst naar de console bleef de GUI niet responsief. Dit werd opgelost door een aparte thread in te voeren die een buffer bevat waarin de tekst wordt opgeslaan. Deze buffer stuurt de text met een lagere frequentie door naar de console. Op deze manier bijft de GUI responsief.

5.2 Feedback verwerken

Er zijn verschillende aanpassingen gemaakt op basis van feedback die we gekregen hebben van zowel de begeleider als de opdrachtgever. Er zijn verschillende implementatiedetails die gebruiksvriendelijker geworden zijn door de feedback die we gekregen hebben.

Een voorbeeld hiervan is het mechanisme om blokken te verwijderen in het Klasse-view. Eerst stond er in het Klasse-view een vuilbak waarop de gebruiker zijn te-verwijderen blokken kon slepen. Dit werd een niet-passend mechanisme gevonden. Als reactie op deze feedback hebben we deze vuilbak verwijderd en de mogelijkheid geboden om blokken te verwijderen door een optie te selecteren in het rechtermuisklik-menu.

Ook hebben we feedback verkregen over het kleurenschema dat we konden gebruiken. Door nieuwe combinaties te proberen hebben we iets gevonden wat neutraal en professioneel is.

6 Keuze programmeertaal: Java

We hebben gekozen voor Java omwille van het grote aanbod van uitgebreide API's. Beide teamleden zijn bekend met de programmeertaal door de cursus object-georiënteerd programmeren 2.

Ook is de taal cross-platform wat een eis is van het project. Andere talen zoals C++ kunnen ook cross-platform applicaties maken, echter is Java meer out-of-the-box cross-platform. Tenzij de programmeur speciale libaries zou gebruiken, is de applicatie altijd cross-platform.

Multithreaded programmeren wordt ook versimpeld door gebruik te maken van de `Thread` libary die Java aanbiedt. Hiermee kan de uitvoering op een andere thread gebeuren dan de uitvoering van de GUI.

De automatische garbage collection van Java zorgt er voor dat er snel een werkende applicatie kan gerealiseerd worden. Java biedt ook een sterke GUI libary aan nl. `Swing`.

7 Views-Models-Uitvoering

Het programma is opgebouwd uit drie grote modules. Deze keuze werd gemaakt om de ontwikkeling op te kunnen splitsen. Zo werd eerst de uitvoerings module gebouwd zodat er zekerheid was over de werking van de applicatie. Hoewel de creatie van een programma om deze manier omslachtig is, kon deze module grondig getest worden.

Na een creeëren van een betrouwbare uitvoering, kon de Model module worden uitgewerkt. Hierin werd de applicatie logica gestoken. Na het vervolledige van deze module werd de connectie gemaakt tussen de uitvoering en de model module. Dit in vorm van een compiler.

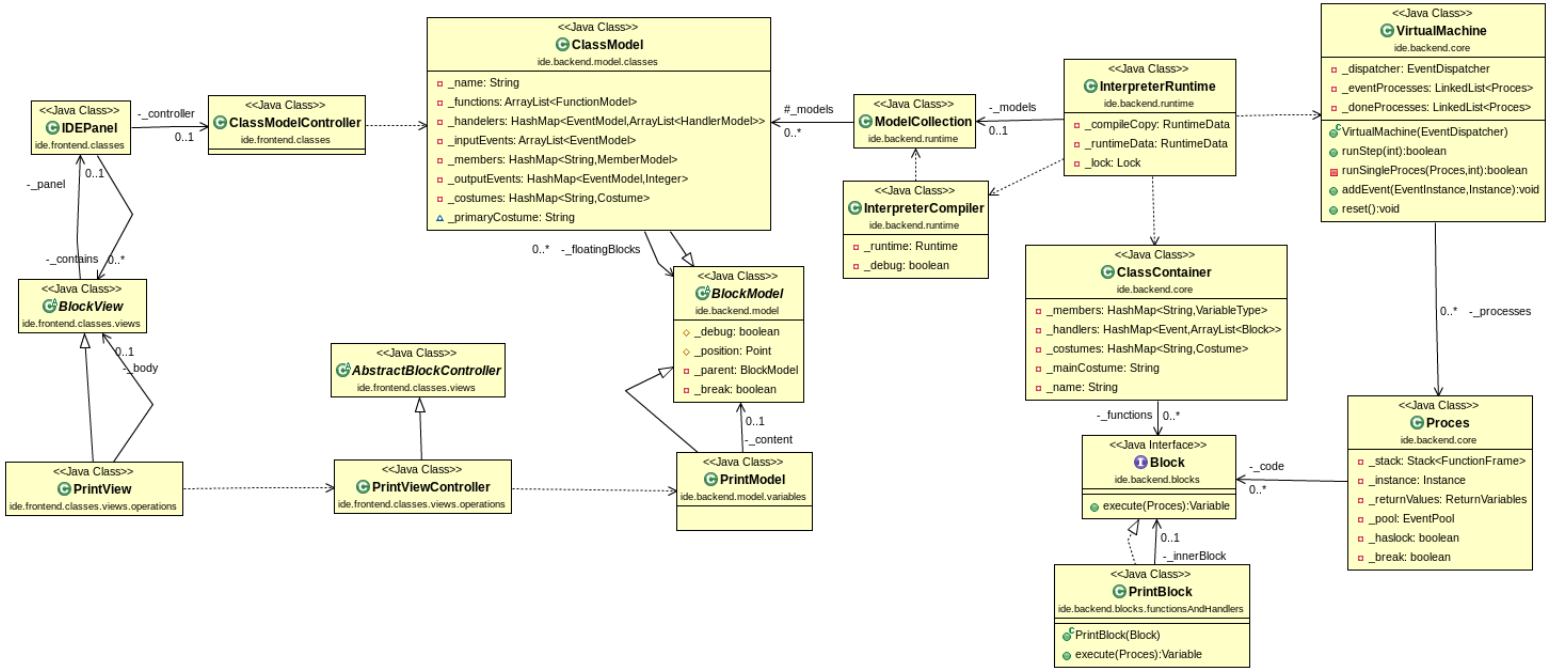
Door het inladen en opslaan van een programma mogelijk te maken via de model module kan er snel een programma worden geschreven in het gekozen bestandsformaat (XML). Het opslaan en inladen gebeurt via interfaces, het gebruikte opslagformaat kan dus ook simpel aangepast worden door andere implementerende classes te maken.

Daarna is de Views module opgebouwd. Doormiddel van de implementatie van de DataEditor kon gemakkelijk de correctheid van een programma gecontrolleerd worden. Dit door de reeds betrouwbare backend.

De opbouw van het programma om deze manier brengt een aantal extra voordelen mee. Zo staat de backend (de uitvoering), de model logica en de view apart van elkaar. Door andere executieblokken en een andere compiler te schrijven kan een programmeur zeer eenvoudig een nieuwe backend voorzien. De koppeling tussen de drie modules gebeurt door basis klassen zoals BlockModel, of de Compiler klasse. De frontend kan ook volledig aangepast worden losstaand van de executie.

7.1 Pipeline

In onderstaande figuur word de volledige pipeline getoond van een enkele blok. Er kan gezien worden dat de views enkel communiceren met de models. Deze models zitten in de ModelCollection die in verbinding staat met de runtime en de compiler. De runtime (en de compiler) vormt de verbinding tussen de models en de uitvoerblokken. Er is te zien dat er verder geen verbindingen zijn tussen de models en de uitvoer. In de volgende secties zullen de verschillende componenten van deze pipeline in detail besproken worden.



Figuur 10: *Pipeline.*

8 Modellen

8.1 Nesting van blokken

Een programma opgebouwd uit blokken kan gezien worden als een boom. Hierbij vormen de verschillende functies of handlers verschillende deelbomen en stellen blokken bladeren voor in deze deelbomen. Om logica in het nesten van blokken te steken is er nood aan onderlinge communicatie tussen de blokken. Zo moet een blok informatie aan zijn kinderen kunnen geven en een kind informatie aan zijn ouder.

Een kind moet ook zijn boom kunnen afsporen om te kijken van welke deelboom hij deel uitmaakt. Een voorbeeld hiervan in een return block. Deze zal nagaan als hij tot de deelboom van een functie behoort en wat het return type is van de functie.

Ook is er nood om de applicatie logica te scheiden van het visuele gedeelte van het programma. Hierdoor kan het visuele gedeelte makkelijk veranderd worden zonder dat de logica verloren gaat.

De volgende sectie bespreekt hoe beide deelproblemen worden opgelost in de implementatie doormiddel van het invoeren van Modellen.

8.2 Modellen, Visuele representatie en Controller

De applicatie logica voor het opbouwen van een progamma wordt gescheiden van visualisatie van een programma door door de invoering van het concept Model. Dit model beheert de applicatie logica van de blok. Een visuele block (view) kan dan communiceren met zijn model via een controller. En zal veranderingen aan het model observeren door middel van het Oberver Pattern [7]. Het patroon wat beschreven wordt doormiddel van deze drie componenten: Model, View en Controller is het MVC patroon.

8.3 Implementatie

Deze sectie zal de implementatie in het programma bespreken van bovenstaande concepten en deelproblemen. Eerst worden de twee basis modellen **BlockModel** en **ConnectedBlocks** besproken. BlockModel stelt een knoop voor in de boom van blokken. ConnectedBlocks wordt gebruikt voor een collectie knopen met dezelfde ouder.

Onderlinge Communicatie bespreekt hoe de onderlinge communicatie tussen ouder en kind werd geïmplementeerd. Hierna volgen enkel specifiëke modellen van blokken die van deze communicatie gebruik maken besproken. Dit zijn opeenvolgend **Variabelen**, gedrag van een **Access blok in een Handler**, gedrag

van een **Return blok** en het gedrag een **Emit blok**. Ook de implementatie van **TypeChecking** wordt besproken.

BlockModel

BlockModel is de basis klasse voor het model van elke block. sBlockModel leidt af van de abstracte klasse `java.util.Observable` [8]. Een BlockModel bevat de volgende basis attributen: Een Blockmodel parent, een positie, een debug status en een break status. De parent attribut verwijst naar de block waarin de block zich bevindt. Voor een top level block zoals een functie of handler zal dit `null` zijn. Ook voor blokken die nergens toe behoren zal dit zo zijn.

De positie van de block wordt opgeslaan voor het inladen van een programma mogelijk te maken. Deze zal dus geupdate worden bij elke verplaatsing.

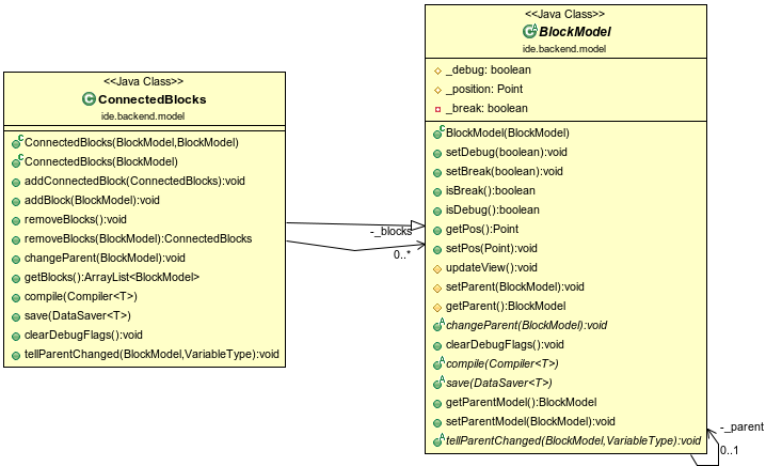
De attributen debug status en break worden in detail besproken in Sectie 9.10.

ConnectedBlocks

Naast het inelkaar steken van blokken is er ook de nood om blokken onderelkaar te plaasten of meerder blokken in eenzelfde block te steken. Zoals bijvoorbeeld de body van een if-statement. Hiervoor wordt de de klasse ConnectedBlocks gebruikt. Deze is afgeleid van een BlockModel en bevat dus dezelfde functionaliteit. Alle blokken die zich in een ConnectedBlocks bevinden zullen dezelfde parent hebben namelijk die connectedBlock. Bij het verwijderen van een blok uit de connectedBlock zal een nieuwe connectedBlock worden gecreeërd. Deze bevat de verwijderde blok plus alle blokken die zich onder deze blok bevonden.

Onderlinge Communicatie

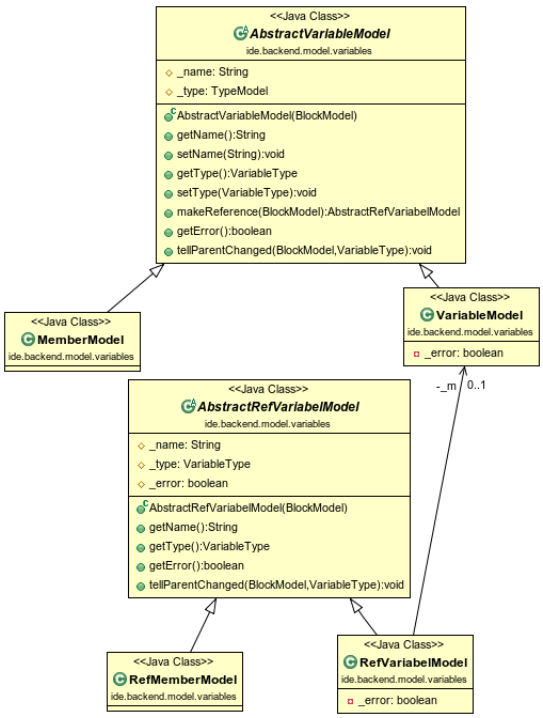
Om de applicatie logica te implementeren was er nood aan een onderlinge communicatie tussen blokken. Een parent-blok moet zijn kind kunnen verwittigen voor veranderingen maar ook communicatie in de andere richting is gewenst.
Deze communicatie wordt mogelijk gemaakt door volgende functies uit BlockModel `changeParent` en `tellParentChanged`. Doormiddel van `changeParent` laat een parent weten aan zijn kind(eren) dat deze veranderd is naar de meegeven BlockModel. `tellParentChanged` geeft de mogelijkheid om een kind te laten weten wat zijn type is en wie hij is (mogelijks heeft een parent meerdere kinderen).



Figuur 11: *BlockModel en ConnectedBlocks.*

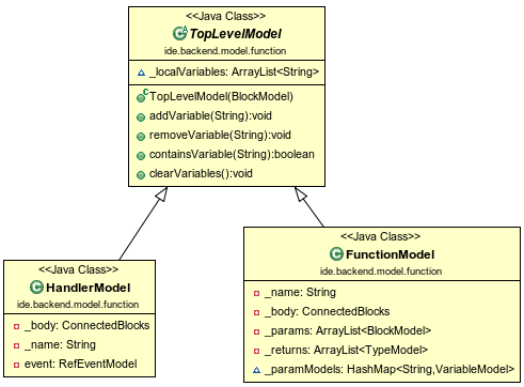
Variabelen

Bij variabelen wordt er een onderscheidt gemaakt tussen lokale variable en membervariabelen van een klasse. Voor hun gemeenschappelijke basis en eenvoud in gebruik op plaatsen waar geen onderscheid moeten worden gemaakt worden zowel voor de declaratie modellen als referentie modellen een abstract basis klasse ingevoerd. Dit zijn respectievelijk AbstractVariableModel en AbstractRefVariabelModel (Figuur 12).



Figuur 12: *VariableModels*

Variabelen zijn een voorbeeld waarbij er nood is aan onderlingen communicatie tussen parent en child. Een declaratie van een lokale variabele gebeurt door een VariableModel. Een lokale variabele bestaat enkel in de scope van een een Handeler of Functie. TopLevel blok is een abstracte basis klasse die afleid van BlockModel voor een functie of handler. Deze Klasse houdt een lijst bij van alle namen van zijn lokale variabele.



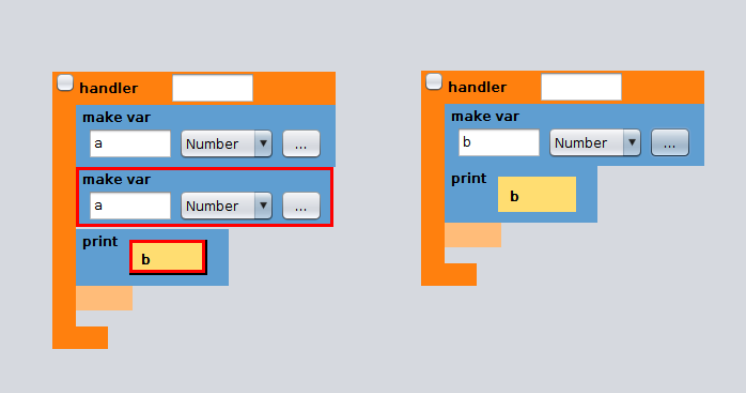
Figuur 13: *TopLevelModel*

Bij het toevoegen van een VariableModel aan een TopLevelModel zal deze al zijn kinderen laten weten dat hun parent veranderd is. Dit wordt doorgegeven tot onderste kinderen (een variable is altijd het einde van een tak).
Als op een VariableModel de `changeParent` functie wordt opgeroepen zal deze nakijken als hij zich bevindt in een ToplevelBlock, dit doormiddel van onderstaande pseudo code. Zoja, als deze veranderd is, wordt er gevraagd als de naam van de variabele al in gebruik is. Zoja, wordt een error getoond aan de gebruiker (Figuur 14). Anders, wordt deze toegevoegd. Er moet ook bij verandering van naam hiermee rekening worden gehouden. Bij het verwijderen zal de variabele uit de lijst moeten worden verwijderd.

```
Zoek functie of handler waartoe blok behoort(){
              als heeft Parent{
                     huidig = getParent();
                     Zolang handler of functie niet gevonden en
                         niet top van boom bereikt{
                             loop naar boven in boom en houdt
                                 positie bij huidig.
                     }
              }
              Als huidig een functie of handler is
                     geef huidig terug.
              anders
                     geef null terug.

       }
```

Referenties naar een lokale variabele moeten ook in dezelfde scope zitten als hun declaratie. Dit moet duidelijk gemaakt worden aan de gebruiker. Dit wordt gerealiseerd op dezelfde manier. Een RefVariableModel zal bij oproep van `changeParent`

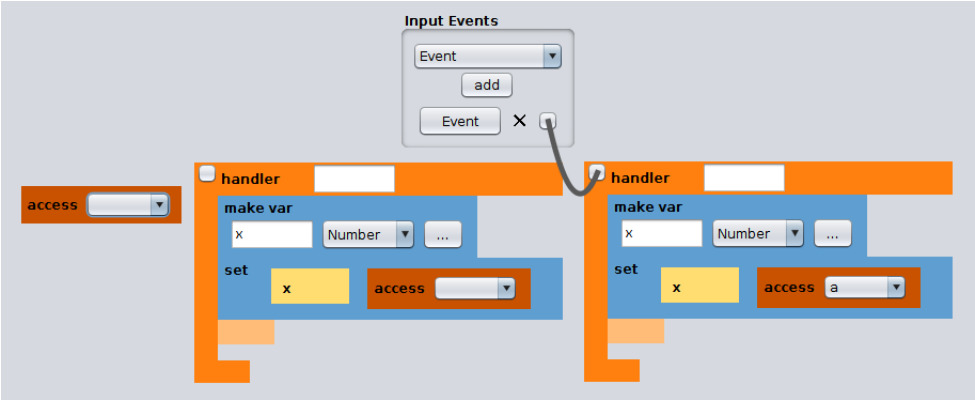
zoeken naar zijn TopLevelModel. Deze vergelijkt hij met het TopLevelModel van de VariableModel naar welk hij refereert, waarvan hij een referentie bezit. Als deze niet gelijk zijn wordt er een error getoond aan de gebruiker (Figuur 14).



Figuur 14: *Error duplicate naam en foute scope.*

AccessModel en HandlerModel

Een access blok wordt gebruikt voor informatie uit een InputEvent dat verwerkt wordt door een handler op te halen. Uiteraard moet een access blok weten als hij al dan niet in een handler steekt. Dit kan door bij het oproepen van de `changeParent` te gaan zoeken naar zijn TopLevel en na te kijken als dit een handler is. En als deze Handler reeds verbonden is met een InputEvent. Als een handler van een InputEvent wordt verwijderd of veranderd. Kan deze verandering makkelijk worden doorgegeven worden door dezelfde `changeParent` functie. Als een AccesModel zijn Event heeft gevonden kan een member geselecteerd worden. En kan het type van de member worden opgevraagd. Dit kan dan gebruikt worden voor de typechecking (Sectie 8.3).



Figuur 15: *Access blok zoekt naar Event handler.*

ReturnModel en FunctionModel

Een ReturnModel werkt op een gelijkaardige manier als een AccessModel. Bij het veranderen van zijn parent zal hij zoeken als hij zich in een FunctieModel bevindt. Zoja, vraagt hij naar het return type zodat hiermee type checking kan gebeuren (Sectie 8.3).

EmitModel

Een emit blok wordt gebruikt voor verzenden van OutputEvents. Het kan zijn dat een Event Members bevat die dan ingevuld moeten worden. Het is duidelijk dat een EmitModel moet weten welke Events er allemaal beschikbaar zijn in het programma. Deze kan hij ophalen uit de ModelCollection (Sectie 8.4). Na het selecteren van een Event kan een EmitModel net zoals een AccessModel nagaan wat de type(s) zijn van de member(s). Dit kan dan weer gebruikt worden voor type checking (Sectie 8.3). Aangezien de members van een EventModel ook VariableModels zijn, kunnen deze geobserveerd worden voor veranderingen. Bij een verandering zoals het type. Zal dit worden geupdate in het view van het model .

TypeChecking

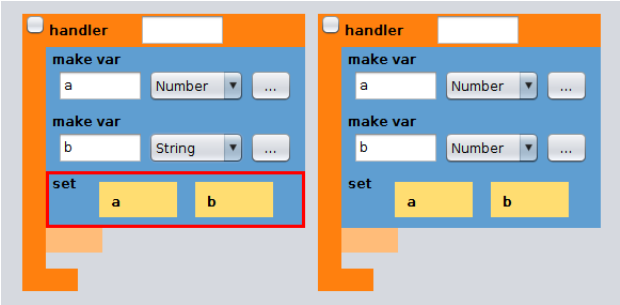
Door gebruik te maken van eerder beschreven functies `changeParent` en `tellParentChanged` wordt typechecking geïmplementeerd in het programma. Uiteraard heeft niet elke blok een type en moet niet iedere blok aan zijn parent melden wat zijn type is.

Voor sommige blokken zoals SetModel is de implementatie eenvoudig. SetModel kan gebruikt worden om de waarde van twee variable of een variable en een waarde aan elkaar gelijk te stellen. Hier moet de parent nakijken na het ontvangen van het type van beide kinderen als er een error optreed (zie onderstaande

code fragment).

```
public void tellParentChanged(kind , type van kind) {
              _fout = false;
              als kind is rechter kind
                     _rechter type is type van kind;
              Als SetBlock volledig{
                     als (types niet overeenkomen en rechter
                         type is geen letterijke waarde )
                             _fout = true
              }
              Laat view weten dat er een fout is.
       }
```

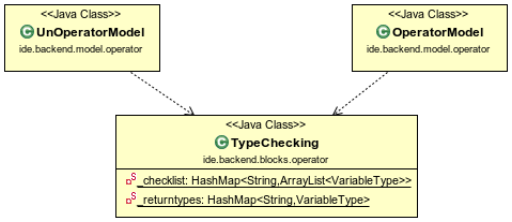
Doordat een AbstractRefVariableModel bij verandering van type van de variabele waarnaar hij verwijst opnieuw aan zijn parent laat weten dat er een verandering is. Kan bij elke verandering een error onstaan of ongedaan gemaakt worden (Figuur 16).



Figuur 16: *Typecheck error.*

Type checking voor gebeurt op een gelijkaardige manier voor de volgende blokken: EmitModel, AccessModel, ReturnModel, MoveModel, CharAtModel en LengthModel.

Voor de operator blok gebeurt dit anders. Een unaire en binaire operatie worden respectievelijk voorgesteld door een UnOperatorModel en een OperatorModel. Deze klassen maken gebruik van een TypeChecking klasse. Bij het ontvangen van de types van beide operanden zal worden nagekeken als de operatie mogelijk is en wat het return type is van de operatie. Dit return type kan dan weer verder worden doorgegeven.



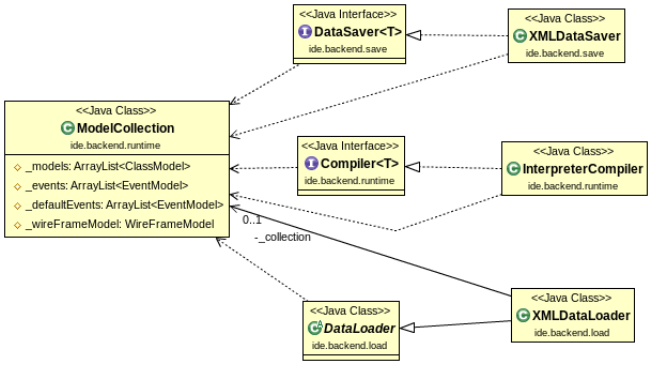
Figuur 17: *Operators.*

8.4 ModelCollection

De data van een programma wordt bewaart in drie delen. Namelijk het WireFrameModel wat alle informatie bevat over de bestaande instanties en de connecties tussen deze instanties. De EventModels die de bestaande Events beschrijven. En de ClassModels die de bestaande klassen beschrijven.

Om interactie tussen deze modellen mogelijk te maken en alle informatie centraal te verzamelen. Wordt er een data klasse ingevoerd, de ModelCollection klasse bewaart alle informatie van het programma. Hierdoor kan ervoor gezorgd worden dat aanpassingen in een module door kunnen worden gegeven aan een andere module.

Deze data collectie wordt gebruikt om het programma te compileren, op te slaan of in te laden.



Figuur 18: *ModelCollection.*

9 Executie

Zoals al eerder vermeld in dit verslag is dient onze applicatie om event-driven programma's te maken. In de volgende sectie wordt er kort uitgelegd wat het **event-driven programming** paradigma inhoudt en hoe dit geïmplementeerd

werd. De uitvoer verloopt concurrent. Er is gekozen om een **eigen interpreter** te schrijven. Hierdoor is er volledige controle over de concurrente uitvoering.

De uitvoeromgeving is opgesplitst in enkele onderdelen. De **runtime** die op hoog niveau controle heeft over het programma, zoals het laten lopen of stoppen van een programma. Events verzonden door de gebruiker moeten opgevangen worden en doorgestuurd worden naar de uitvoering. Dit gebeurt door de **EventCatcher**. Er moet een omzetting gebeuren van de bestaande modellen naar een uitvoerbaar formaat. Dit gebeurt door gebruik te maken van een **Compiler**.

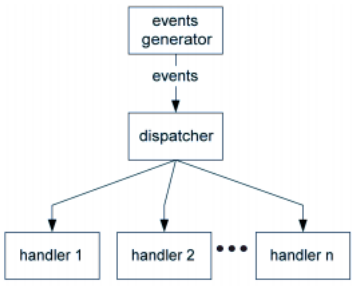
De eigenlijke uitvoering moet ook geregeld worden. Het schedulen van de verschillende gesimuleerde threads moet efficiënt en eerlijk gebeuren. Di is de taak van de **virtual machine**. Er moet ook ondersteuning geboden worden voor **debugging**. Het probleem bij de implementatie van debugging is het vermijden van het vervuilen van de gewone uitvoering.

9.1 Event-driven programming.

Event-driven programming is een programmeer paradigma waarbij de flow van het programma wordt bepaald door events gecreeërd door de gebruiker zoals input events of door events veroorzaakt door delen in het programma [9].

Extended handlers design pattern

Als design pattern voor onze applicatie hebben we ons gebasseerd we op het extended handlers design pattern dat Stephen Ferg uitlegt in zijn paper: Event-Driven Programming: Introduction, Tutorial, History [10].



Figuur 19: *Extended handler.*

In Figuur 19 stelt de eventgenerator in onze applicatie het generenen van events door gebruikers input en door instanties voor. Door dat events talrijk gegenereerd kunnen worden zal de Dispatcher een queue zijn die een stroom van events opvangt. Hij zal ervoor zorgen dat het event door de juiste handelers wordt afgewerkt.

Doordat in onze applicatie events worden doorgegeven aan specifieke andere instanties van Klassen zoals beschreven in de Sectie 3.1. Zal de dispatcher ervoor moeten zorgen dat de juiste handlers van de juiste instanties worden aangeroepen.

9.2 Concurrent computing

Concurrent computing [11] is een vorm van computing waarbij een deel berekeningen worden uitgevoerd zodat het lijkt alsof ze gelijktijdig worden uitgevoerd. We hebben ervoor gekozen om niet multi-threaded te werken om de complexiteit van het project zo laag mogelijk te houden.

Daarom is concurrent computing de oplossing voor onze applicatie. In tegenstelling tot parallel computing is dit wel mogelijk op een thread. Gelijktijdige processen zoals twee events die samen worden opgeroepen lijken hierdoor ook gelijk te worden afgehandeld. In tegenstelling tot het sequentiel uitvoeren van de twee events.

Bij concurrent computing worden processen in executie stappen opgedeeld. Er wordt gebruik gemaakt van timeslices waarin van elk proces een deel executie stappen worden uitgevoerd, wij noemen dit een primitieve stap. Na dat bepaald aantal of tijd wordt het proces gepauzeerd en verder gegaan met het volgende. Dit wordt herhaald zolang een proces niet volledig is afgerond.

Verschil met parallel computing

Parallel computing is gelijkaardig aan concurrent computing, echter gebeurt het werk op verschillende cores. Bij concurrent computing zal de uitvoering van twee identieke events die gelijktijdig worden aangeroepen niet gelijktijdig eindigen omdat de uitvoering steeds wisselt tussen de twee events.

Probleem met concurrent computing

Een eerste probleem is **racing conditions**. Hierbij proberen twee processen hetzelfde algortime uit te voeren waarbij een bepaalde sequentie van uitvoering belangrijk is. Het volgende voorbeeld gevonden op [11] toont een functie waarbij een private member variabele balance wordt geaccessed en verandered. Stel dat er twee processen runnen die respectievelijk withdraw(200) en withdraw(300) oproepen en dat balance 250 bedraagt. Bij beide processen zal de conditie 250 > withdrawal, slagen, want balance is nog niet aangepast. Echter zal hierna balance aangepast worden en uiteindelijk -250 bedragen. Dit is een verkeerde uitvoering.

```
public class Main {
       public boolean withdraw(int withdrawal) {
       if (balance >= withdrawal) {
```

```
                balance -= withdrawal;
                return true;
        }
        return false;
        }
}
```

Onze oplossing hiervoor is een **lock** op een private member variabele toelaten. Deze zorgt dan dat enkel dat proces aan die variabele kan voor zowel te lezen als te schrijving.

Hierbij komt een ander probleem tevoorschijn, nl. een deadlock [12]. Om dit probleem op te lossen stellen we dat slechts één proces gelijktijdig locks kan aanbrengen op een instantie.

9.3 Runtime

Op het hoogste niveau is een **Runtime** aanwezig. Deze regelt de algemene uitvoering van een programma.

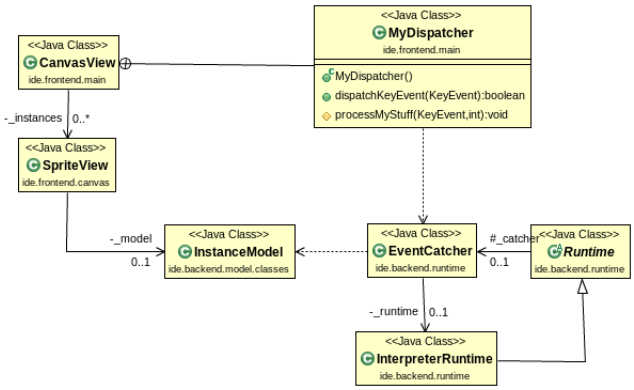
De IDE gebruikt een Abstracte klasse Runtime zodanig dat de gebruikte programmeertaal waarin de uitvoering gebeurt niet direct vasthangt aan de IDE. Er is een klasse aanwezig die de Runtime voor de geïmplementeerde programmeertaal implementeerd. De abstracte Runtime bevat de ModelCollection (Sectie 8.4). De geïmplementeerde Runtime bevat de nodige componenten voor de executie mogelijk te maken zoals een Compiler en een Virtual Machine (zie Sectie 9.6). De Runtime zorgt voor de vlotte uitvoering van het programma. Er is een functie aanwezig die continue de Virtual Machine aanroept zolang er niet gestopt moet worden. Door een aparte thread aan te maken zal hij deze functie parallel kunnen runnen met de GUI. Verdere executie is uitgelegd in Sectie 9.6.

9.4 Eventcatcher

De Runtime krijgt ook events binnen die opgevangen worden in de GUI. De GUI vangt key-presses en mouse-events op. Het CanvasView vangt de keypresses op en SpriteView vangt mousePress en mouseRelease events op. Deze worden doorgestuurd naar de EventCatcher die de binnengekregen events vertaalt naar events die de Runtime kan gebruiken. Deze stuurt vervolgens het event door te sturen naar de virtual machine.

De **EventCatcher** gebruikt een hashmap die systeem-events (events van de IDE) koppelt aan Swing events. Als de hashmap het ontvangen event niet bevat, verstuurt de EventCatcher een standaard event (`"<default>"`). Hierdoor kan de gebruiker ook programma's maken die reageren op key's die niet individueel ondersteund zijn door de IDE.

Alle systeem-events beginnen met een `"<"` en eindigen met een `">"`. Hiertussen zit de beschrijving van het event. Er is een start-event: `"<Start>"`, keypress events: `"<keyA>"` (van A tot Z) en mouse-event: `"<mousePress>"` en `"<mouseRelease>"`.

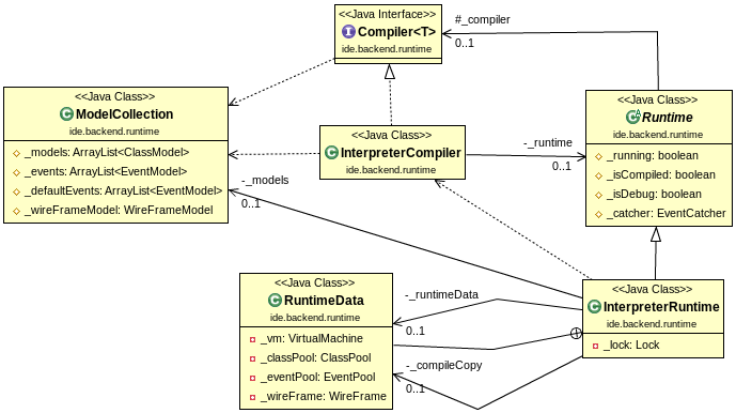


Figuur 20: *EventCatcher.*

9.5 Compileren

De modellen die bestaan in het programma moeten omgezet worden naar een uitvoerbaar formaat. Hiervoor is gekozen om een **Compiler** te implementeren.

Elke visuele view van een blok heeft een model zoals uitgelegd in Sectie 8. Dit model wordt bij het compileren meegegeven aan een Compiler Klasse door de Runtime. Deze klasse is een information expert met betrekking tot compileren. De IDE gebruikt een Interface Compiler zodanig dat de gebruikte programmeertaal van de uitvoering niet direct vasthangt aan de IDE. De interface bevat verschillende functies die elk een ander type model compileren. Omdat het wireFrame ook voorgesteld wordt als een model kan het wireFrame op een gelijkaardige manier gecompileerd worden. Doordat de modellen als een boom opgebouwd zijn, kan de Compiler door deze boom lopen om een programma te compileren. Het design van de Compiler gebruikt het het visitor design patroon. Het algoritme voor het compileren van een blok wordt hiermee gescheiden van de datastructuur van de blok.



Figuur 21: *Compiler.*

Visitor patroon

Onderstaand code fragment geeft aan hoe het visitor patroon gebruikt wordt [6]. De nood aan het visitor patroon komt door het feit dat Java een BlockModel niet zal downcasten naar bijvoorbeeld een PrintModel bij een functie aanroep. Hierdoor zal niet de juiste CompileBlock functie van de compiler worden aangeroepen. Door het model zelf de oproep te laten doen is dit wel mogelijk. Dit gebeurt door een compile functie van het model op te roepen en de compiler mee te geven. Het model zal nu zelf de compile functie van de compiler aanroepen waardoor de functie met juiste type parameter wordt gekozen. In dit geval PrintModel. Andere oplossingen zouden als gevolg hebben dat we de correcte functie moeten kiezen door middel van een soort switch, of dit nu via lambda functies, if-statements of een eigenlijk switch statement gebeurt. Deze oplossing is minder elegant dan het visitor patroon.

```
public class InterpeterCompiler implements Compiler<Block>{
       public Block compileBlock(PrintModel model) throws
           CompileException {
              if (model niet compleet) throw CompileException();
              return new PrintBlock((Block)
                  model.getContent().compile(this));
       }
}

public class PrintModel extends BlockModel{
       public <T> T compile(Compiler<T> c) throws
           CompileException {
              return c.compileBlock(this);
       }
}
```

9.6 Virtual Machine

De virtual machine zorgt voor de goede uitvoering van een enkele cycle in het programma. Er zijn verschillende gesimuleerde threads aanwezig die we processen noemen en elk van deze processen moet uitgevoerd worden. Dit is de taak van de virtual machine. Verzonden events moeten ook verwerkt worden. Er moeten nieuwe processen aangemaakt worden voor elk opgevangen event. Hiervoor is de EventDispatcher geïmplementeerd.

De **virtual machine** bevat enkele lijsten van processen. Er is de lijst van processen die deze cycle uitgevoerd moeten worden. Een cycle wordt gedefinieërd als de uitvoeren van enkele stappen van deze lijst. Het aantal stappen dat van elke lijst uitgevoerd wordt, is gespecifieërd door de runtime, zie Sectie 9.7. Er is een lijst van processen die de hudige stap al uitgevoerd hebben, processen die niet afgelopen zijn worden niet in deze lijst gestoken. En uiteindelijk is er een lijst waarin alle processen zitten die opgestart zijn met behulp van een verzonden event, deze lijst is chronologisch geordend. Het eerste proces in de lijst is het proces opgestart door het eerste verzonden event. Dit houdt zowel events verzonden door de GUI, als events verzonden door andere processen in.

Na het uitvoeren van een cycle, als de lijst van huidige uit te voeren processen leeg is, wordt de lijst van uitgevoerde processen samengevoegd met de lijst van processen aangemaakt door events. Een event wordt eigenlijk pas opgevangen na een cycle en dus worden alle processen opgestart door een event achteraan bijgevoegd aan de lijst van uitgevoerde processen. Deze uiteindelijke lijst zal de volgende cycle gebruikt wordt. Processen die volledig klaar zijn met uitvoeren worden verwijderd.

Als een Event verstuurd wordt zal de Virtual Machine dit Event doorgeven aan een Event Dispatcher. Dit is een Klasse die de taak voor het verzenden van Events op zich neemt. De **Event Dispatcher** kent alle verbindingen tussen de Instanties, en hiermee kunnen nieuwe processen aangemaakt worden zodat de Virtual Machine deze kan gebruiken. Deze processen worden vervolgens in de correcte lijst in de virtual machine gestoken.

Een proces

Een **proces** is een gesimuleerde thread. Deze beheert nodige data zoals een variable-stack en code. Een proces wordt uitgevoerd door de VM. Een proces kan gerunned worden en deze zal dan één primitieve stap 3.1uitvoeren.

Het proces bevat een stack van Blocks. Initiël bevat die stack enkel de handler die opgeroepen is. Het bevat de instantie waarvan het de handler uitvoert. Het bevat ook een Stack van functionFrames (van Hashmaps waarbij Strings worden gemapt op Variables (zie Sectie 9.6).

Deze laatste stack stelt de stack van actieve functies voor. Een proces bevat een run-functie die één primitieve stap zal uitvoeren. Deze functie kan een event teruggeven als deze gecreeërd werd in de primitieve stap. Als het geen event terug geeft is er geen event gecreeërt maar is het proces nog niet afgelopen. Als het wel een afgelopen (de stack van Blocks is leeg) zal het een ProcesFinishedException gooien. Hierdoor weet de virtualmachine dat het event niet terug op de queue moet worden gezet. Het proces bevat ook nog een klasse ReturnVariables.

De klasse bevat ook nog een boolean locked die aangeeft als het proces verantwoordelijk is voor het locken van een variabele. Hierop kunnen er wel nog nieuwe locks gebeuren binnen dat proces. Een voorbeeld van de werking van een proces is uitgewerkt in Sectie 9.8.

FunctionFrame

Een functionFrame stelt het stackframe voor van een functie blok. Deze bevat een hashmap [13] waarbij de lokale Variables van die functie gemapt worden op hun naam in die functie.

De keuze van een Hashmap zorgt ervoor dat het toevoegen van nieuwe variabelen en het opzoeken van bestaande in O(1) tijd kan gebeuren aangezien een functie een beperkt aantal lokale variabele bevat. Door het gebruik van een Hashmap is er ook geen extra compile stap nodig omdat we de variable dadelijk kunnen mappen op de content en geen indices van een array moeten worden berekent. [13]

Stoppen van uitvoering

Zoals aangehaald in Sectie 9.3 bevat de Runtime klasse een functie die continue de VM aanroept. Als de gebruiker wenst te stoppen met uitvoering zal deze functie stoppen met uitvoeren.

Functies

In de GUI zijn enkel functies ondersteund die maar één enkele returnwaarde hebben. De uitvoeromgeving ondersteund echter meerdere returnwaardes. Het toevoegen van meerdere returnwaardes in de IDE vereist enkel een aanpassing aan de GUI.

9.7 Debugging

De uitvoering moet uiteraard ook **debugging** capaciteiten bezitten. Er moest echter opgelet worden dat de structuur van de gewone uitvoering niet vervuild wordt met de toevoeging van een debugging modus. De uiteindelijke implementatie is afgeleid van de implementatie van debugging in standaard programmeertalen zoals Java of C++. Wanneer er voor deze talen gecompilleerd wordt voor debugging, word er extra data, debugsymbolen, toegevoegd aan de executable.

De applicatie zal dit spiegelen en ook op bepaalde plaatsen extra debugging blokken toevoegen.

De runtime kan laten weten aan de Virtual Machine hoeveel stappen er uitgevoerd moeten worden in een cycle. Meestal zal dit aantal een stap zijn. Hierbij voert de Virtual machine een blok uit van elk proces. Voor het debuggen zal de runtime echter stellen dat de virtual machine drie blokken moet uitvoeren.

Bij de compilatie in debug-modus plaatst de compiler speciale debug blokken voor en na elke blok. Hierdoor komt het aantal op drie blokken. De eerste debug blok weet of de gecompileerde blok een breakpoint bevat. Deze zal, als er gebreaked moet worden, een break-exception gooien. Deze stopt dan de uitvoer van het programma. De uitvoer stopt voor de volledige uitvoering, elk proces zal gestopt worden.

Beide debug blokken kennen ook het model dat gecompileerd word. Als de debug blok uitgevoerd worden, zullen de blokken respectievelijk de debug-modus van het model aan of uit zetten.

Een typische uitvoering in debug-modus houdt in: zet de debug-modus van de vorige blok uit, zet de debug-modus van de huidige blok aan en ,indien er niet gebreaked is, voer de huidige blok uit. Bij het toevoegen van een proces zal de debug-modus van de handler aangezet worden. Daarna zal de eerste stap de volgende uitvoer hebben: Voer de handler uit, zet debug modus van de eerste blok aan, voer de eerste blok uit. De laatste stap van een handler zal maar een blok moeten uitvoeren, namelijk het uitzetten van de debug-modus van de handler.

Een model zal (via het observer-patroon) laten weten aan zijn views dat de debug-modus aan of uitgezet is. Deze views kunnen hier dan op reageren. Ze zullen een zwarte rand tekenen rond hun blok. Hierdoor wordt het duidelijk aan de gebruiker dat de uitvoer op deze plaats zit.

9.8 Voorbeeld implementatie

Er bestaat een Klasse die een input event: event1 accepteerd. Dit event wordt afgehandeld door handler1 van de Klasse. De gebruiker heeft deze handler geïmplementeerd in blokken op de volgende manier:

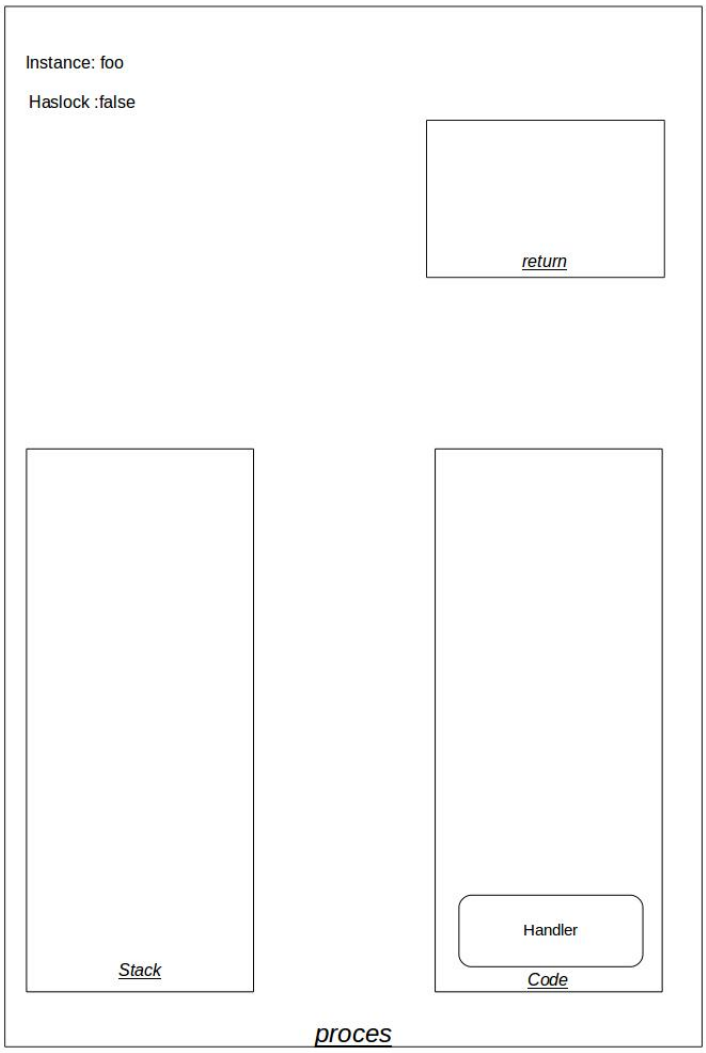
```
Event event1{
       members:
              - member1(number, value)
}
class Class1 {

       handler( event1) {
```

```
        makeVar(number, x)
        set(X, acces(event1, member1)
        functieCall(functie1, parameters: x, return:x)
    }

    functie(functie1, parameters: z){
        while(z < 10){
        set(z, z + 1)
        }
        return z;

    }


}
```

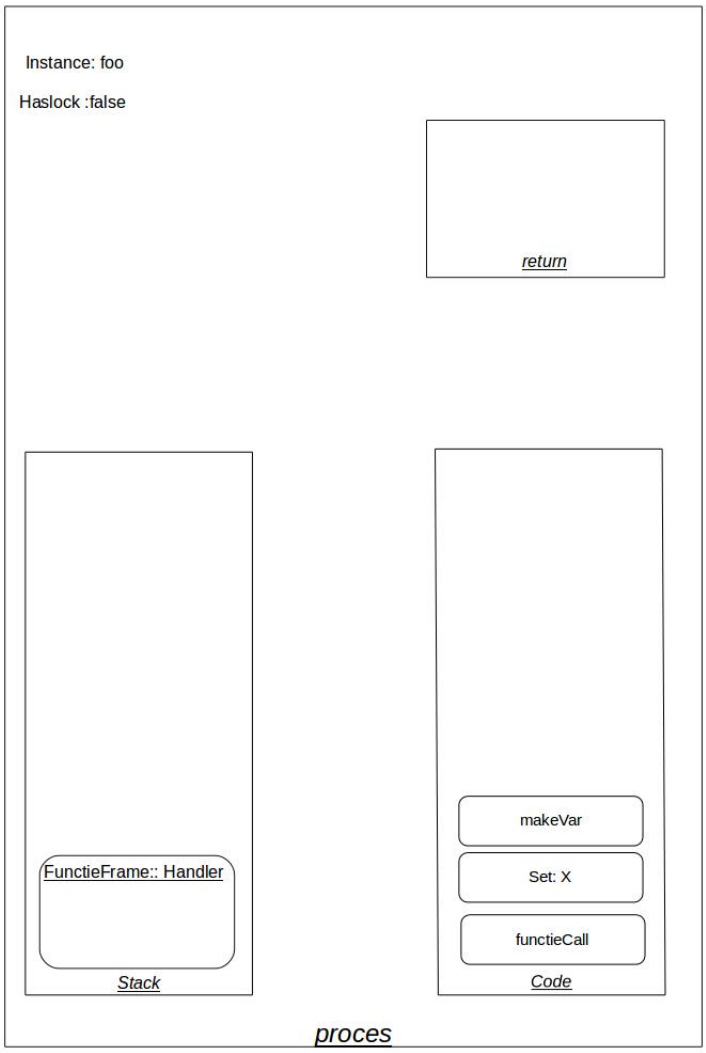
Instantie instance1 is een instantie van Class1 waarnaar een instantie van event1 wordt verzonden. De eventDispatcher zal dus een proces aanmaken met instantie1 en op de Code stack de handler pushen.
Het uitvoeren van het proces zal in de volgende primitieve stappen gebeuren:

Stap 1: De handler zal zijn execute functie uivoeren. Deze zal een nieuw FunctieFrame aanmaken. Ook zal de eventInstance die mee werd gegeven bij het aanmaken van het proces op de stackframe geduwd. De blok van de handler op de Stack wordt vervangen door de inhoud.
Stap 2: De execute van makeVar zal een nieuwe variable van het type number maken op het huidige FunctieFrame.
Stap 3: De execute van Set zal de het Event in de huidige Stackframe zoeken en hieruit de juiste member halen nl. member1. De waarde hiervan wordt in x geplaatst. We gaan er van uit dat member1 = 8.
Stap 4: De execute van FunctieCall doet de volgende stappen. Ophalen van de de parameter waarde die in de call staan. Deze slaat gij op in volgorde van de oproep. Hierna haalt hij het functieblok met de juiste functie naam op. Hiervan haalt hij de parameter namen op. Hij creëert een nieuw FunctieFrame en daarop pushed hij de parameternamen met de juiste eerder opgehaalde waarde. Hier is dit dus z met de waarde 8. Dit gebeurt achter de schermen met makeVar- en set-blokken. Hierna worden er eerst nog set-blokken voor de return waardes op de stack gepushed. Nu wordt de execute van de functie aangeroepen. Deze zal al zijn blokken op de stack pushen zonder nieuw frame te creeëren.
Stap 5: De bovenste blok is nu de While blok. De execute van deze blok zal zijn conditie controleren nl $z < 10$ deze evalueert naar true aangezien x = 8. Hierdoor zal de body van de While-blok op de code stack worden gepushed. Maar eerst wordt de While-blok er zelf ook nog op gepusht.
Stap 6: De waarde van X wordt in de set-blok verhoogt met 1.
Stap 7: herhaling stap 5.
Stap 8: herhaling stap 6.

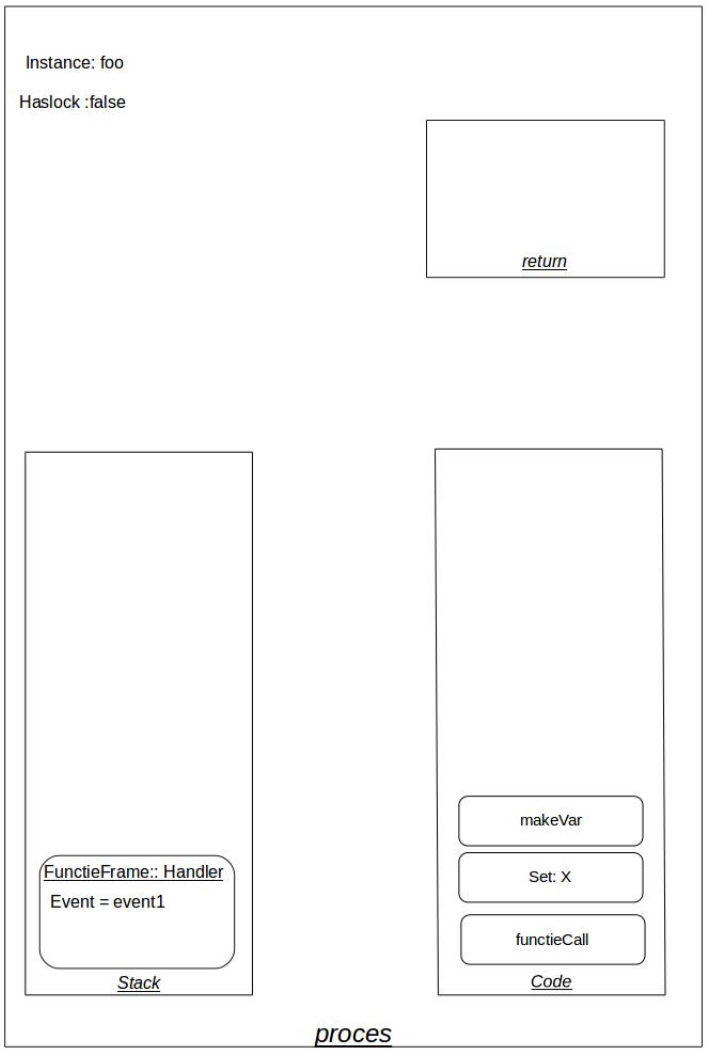
Stap 9: De conditie van de While-blok evalueert nu naar false. Hierdoor worden er geen blokken op de code stack gepusht.
Stap 10: De return blok zoekt in de huidige FunctieFrame de waarde van z op en plaats deze in de returnVariables.
Stap 11: Aangezien de functie is afgelopen mag het FunctieFrame van de stack worden gepopt.
Stap 12: De set-blok zal nu x in het huidige FunctieFrame veranderen naar de return waarde
Stap 13: Het proces bevat geen blokken meer dus zal een ProcesFinishedException gooien naar de VM deze zal het proces verwijderen uit de queue.



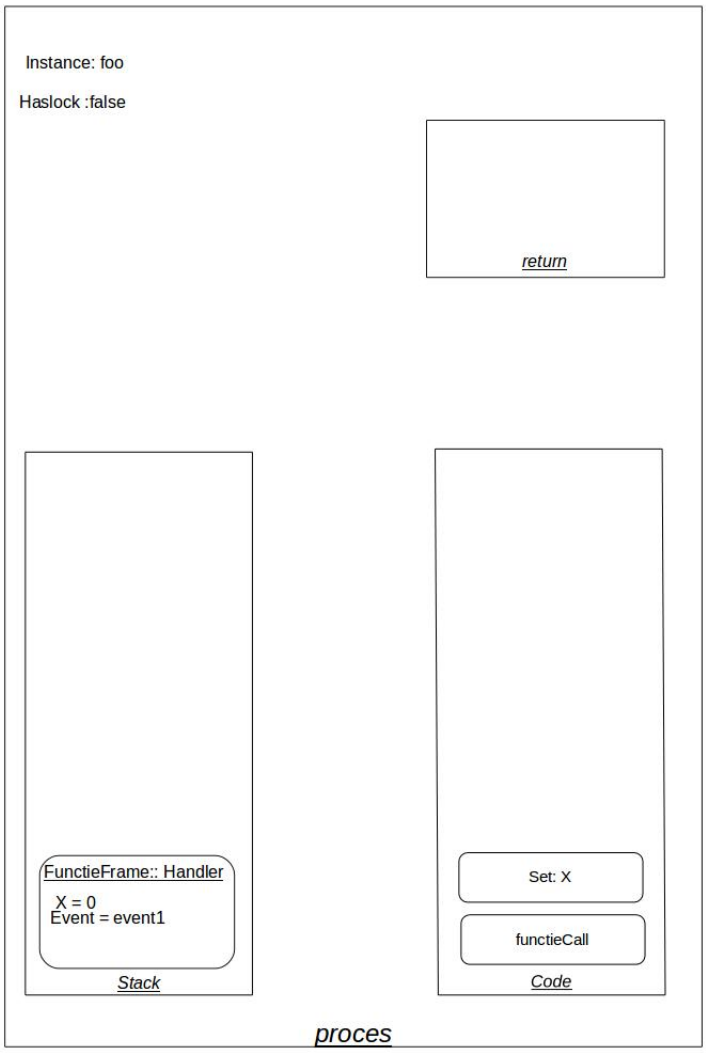
Figuur 22: *Stap 1*



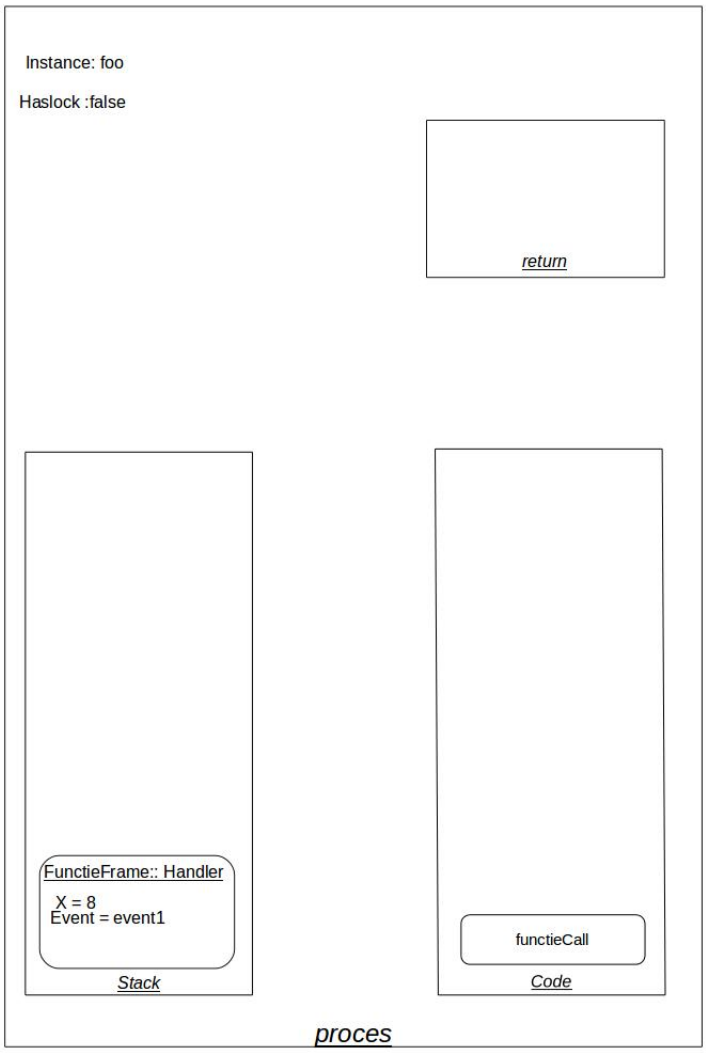
Figuur 23: *Stap 2*



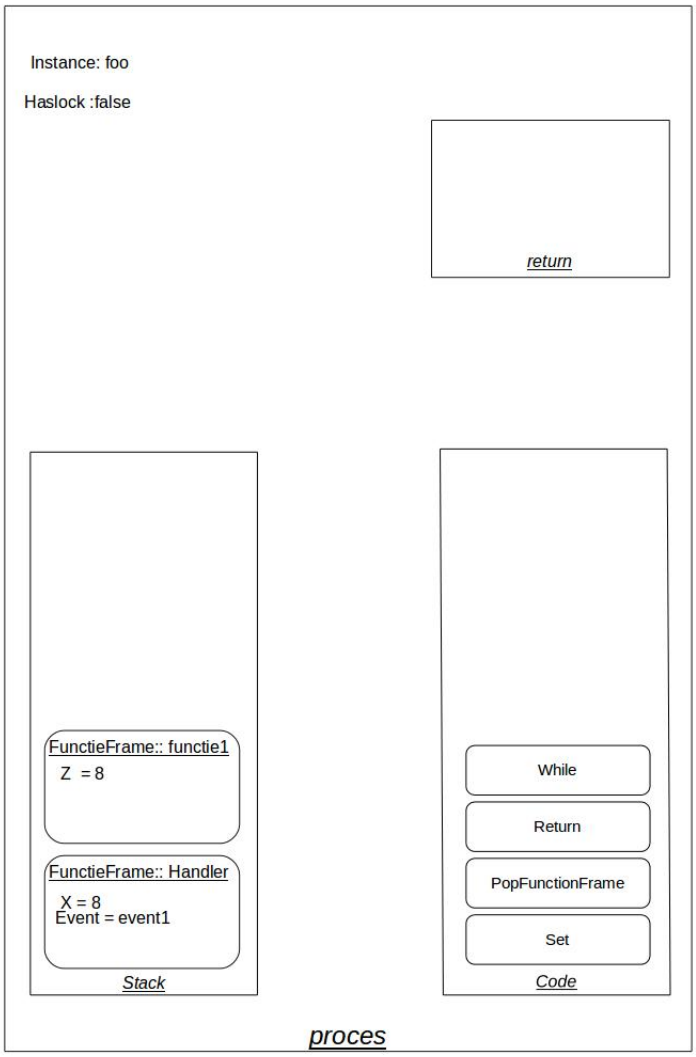
Figuur 24: *Stap 3*



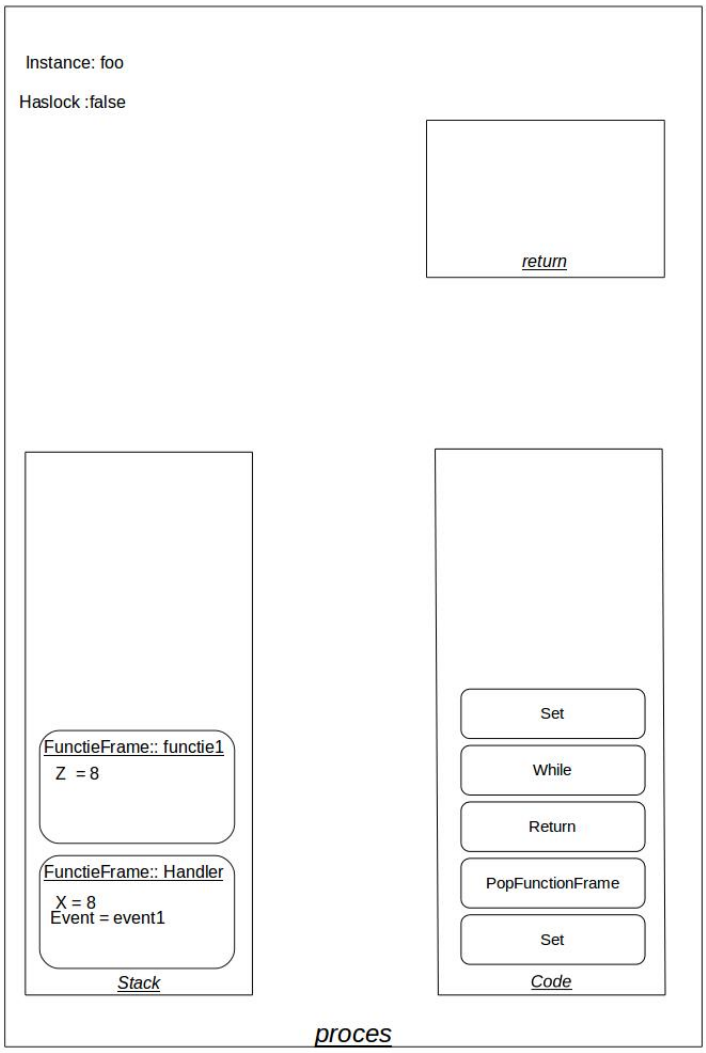
Figuur 25: *Stap 4*



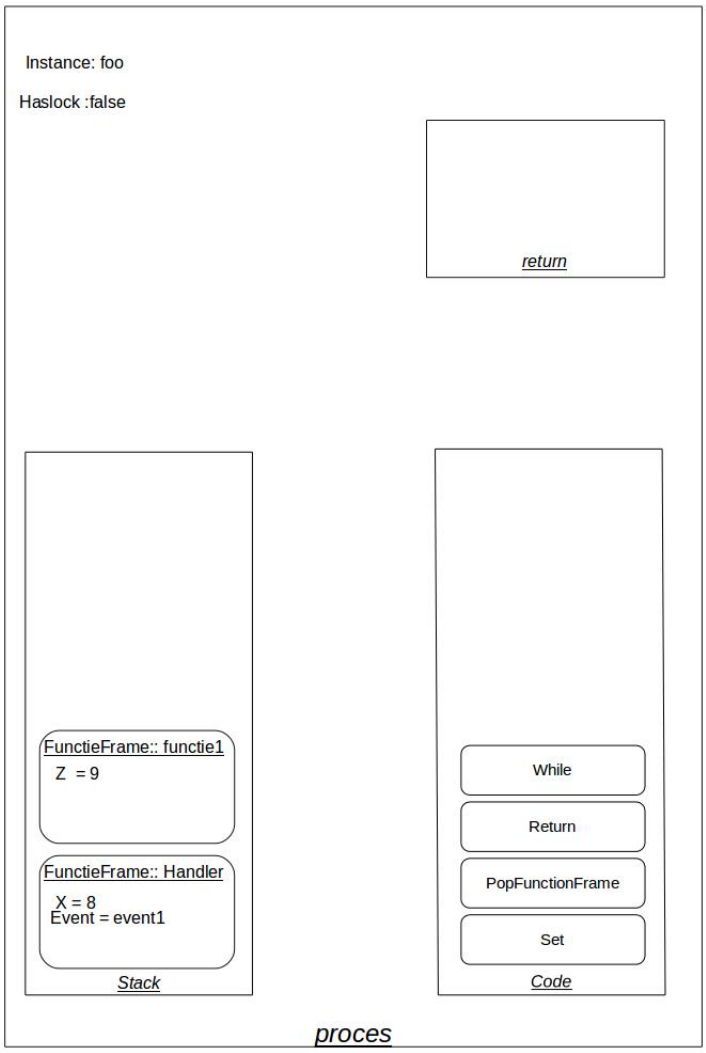
Figuur 26: *Stap 5*



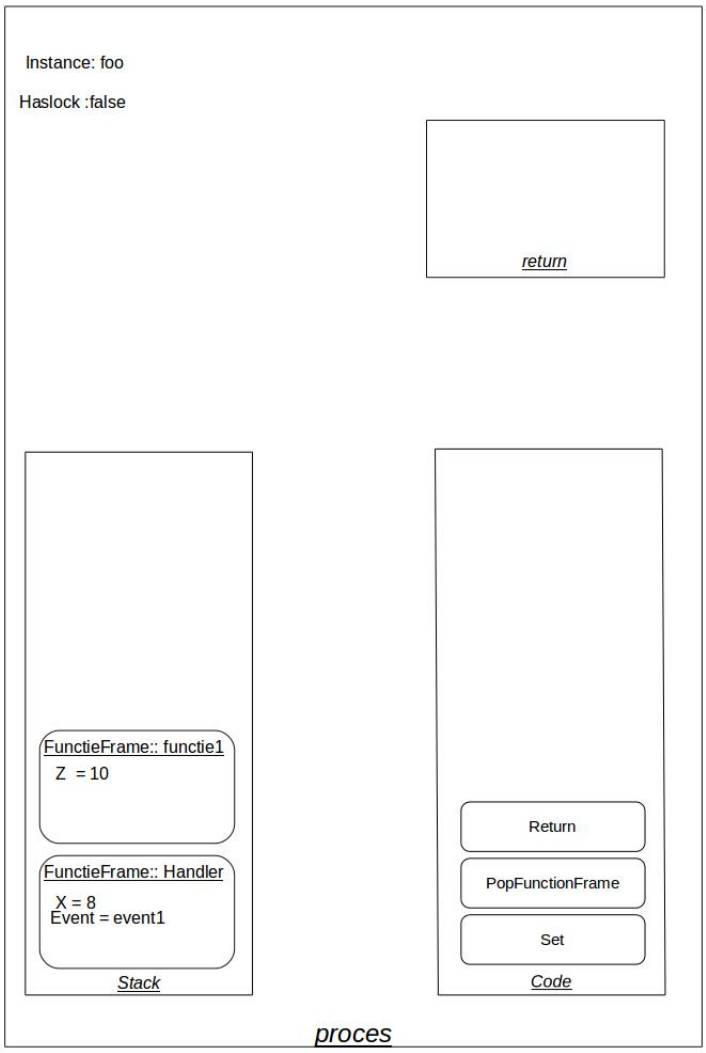
Figuur 27: *Stap 6*



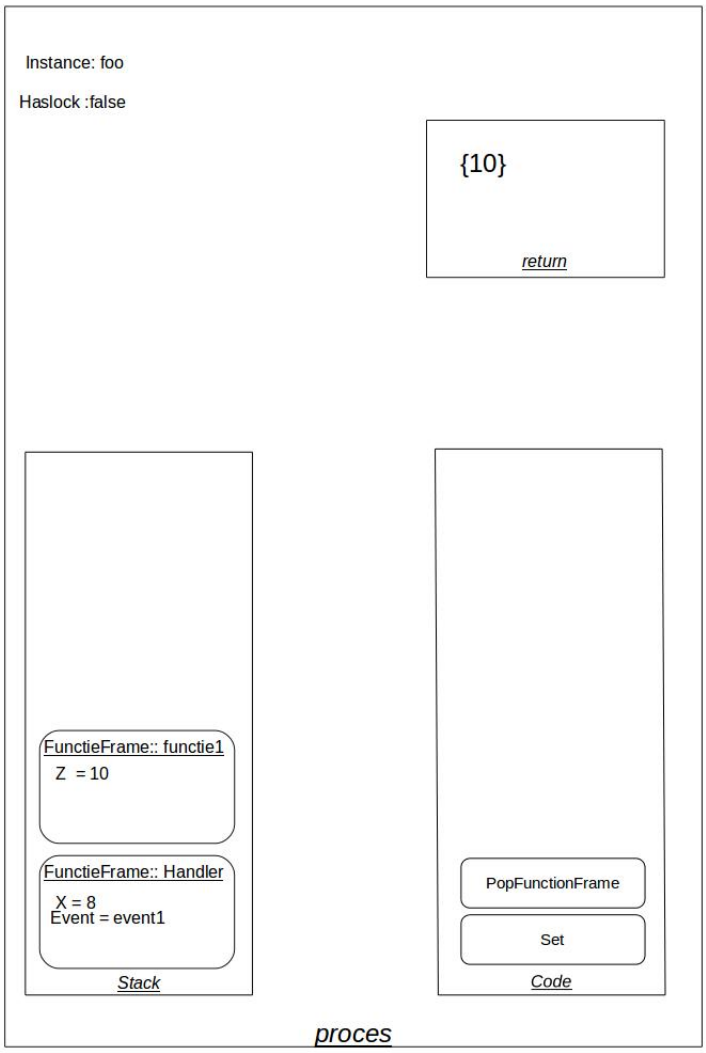
Figuur 28: *Stap 9*



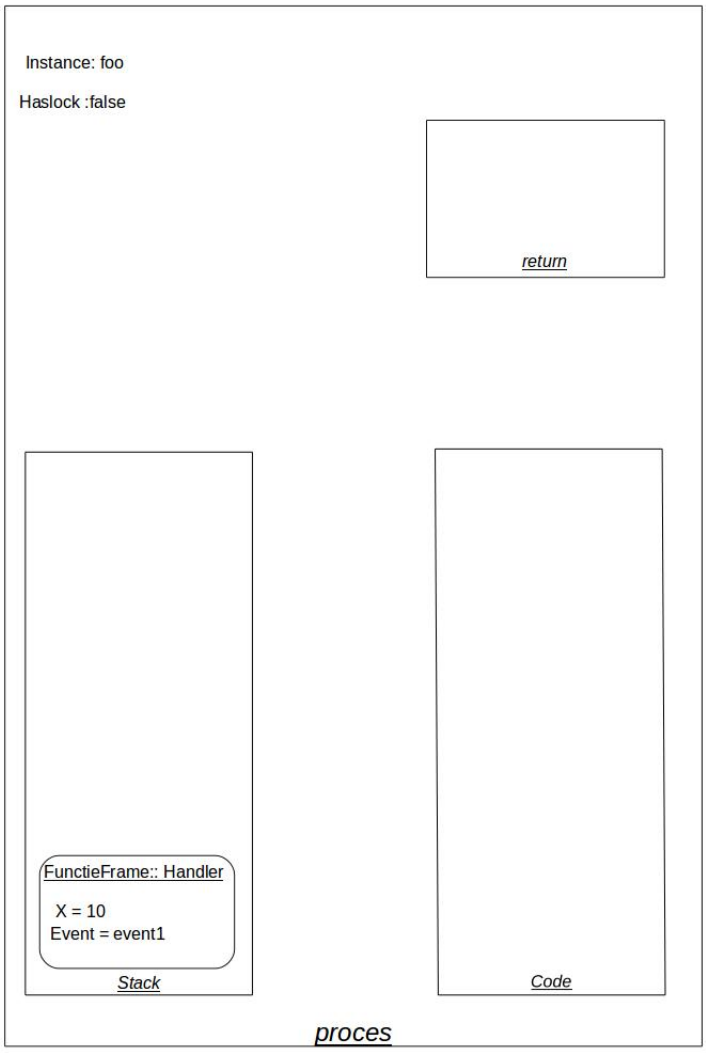
Figuur 29: *Stap 10*



Figuur 30: *Stap 11*



Figuur 31: *Stap 12*



Figuur 32: *Stap 13*

9.9 Lambda expressions

Sinds 1.8 biedt Java de mogelijk aan om lambda expressies [14] te gebruiken, we gaan hier dan ook gebruik van maken. Onze operator blok is hier geschikt voor. Via lambda expressies kan men anonieme interface methodes aanmaken.

Hieronder volgt een voorbeeld waarin duidelijk volgt hoe wij het zouden gebruiken.

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

In de operator block zal een statische hashmap bijgehouden worden. Deze word slechts één keer geïnitialiseerd. In deze hashmap worden strings gemapt op lambda functies. De string + mapt op een lambda functie enz. Deze lambda functie wordt hieronder beschreven:

```
    interface OperatorMath {
        Variable operation(Variable a, Variable b);
    }

    OperatorMath addition = (a, b) -> a.addVar(b);
    OperatorMath subtraction = (a, b) -> a.subVar(b);
```

9.10 Debugging

De runtime kan laten weten aan de Virtual machine hoeveel stappen er uitgevoerd moeten worden in een cycle. Meestal zal dit aantal een stap zijn. Hierbij voert de Virtual machine een blok uit van elk proces. Voor het debuggen zal de runtime echter stellen dat de virtual machine drie blokken moet uitvoeren.

Bij de compilatie in debug-modus plaatst de compiler speciale debug blokken voor en na elke blok. Hierdoor komt het aantal op drie blokken. De eerste debug blok weet of de gecompileerde blok een breakpoint bevat. Deze zal, als er

gebreaked moet worden, een break-exception gooien. Deze stopt dan de uitvoer van het programma. De uitvoer stopt voor de volledige uitvoering, elk proces zal gestopt worden.

Beide debug blokken kennen ook het model dat gecompileerd word. Als de debug blok uitgevoerd worden, zullen de blokken respectievelijk de debug-modus van het model aan of uit zetten.

Een typische uitvoering in debug-modus houdt in: zet de debug-modus van de vorige blok uit, zet de debug-modus van de huidige blok aan en ,indien er niet gebreaked is, voer de huidige blok uit. Bij het toevoegen van een proces zal de debug-modus van de handler aangezet worden. Daarna zal de eerste stap de volgende uitvoer hebben: Voer de handler uit, zet debug modus van de eerste blok aan, voer de eerste blok uit. De laatste stap van een handler zal maar een blok moeten uitvoeren, namelijk het uitzetten van de debug-modus van de handler.

Een model zal (via het observer-patroon) laten weten aan zijn views dat de debug-modus aan of uitgezet is. Deze views kunnen hier dan op reageren. Ze zullen een zwarte rand tekenen rond hun blok. Hierdoor wordt het duidelijk aan de gebruiker dat de uitvoer op deze plaats zit.

10 Verschillende onderdelen van GUI

Zoals eerder beschreven zal de GUI onderverdeeld worden in vier grote onderdelen. Namelijk ClassView, EventView, WirePanel en het Canvas. Deze structuur wordt ook door getrokken in het ontwerp van het programma. Sectie 8.4 bespreek de ModelCollection klasse waarin alle informatie van het programma verzameld wordt.

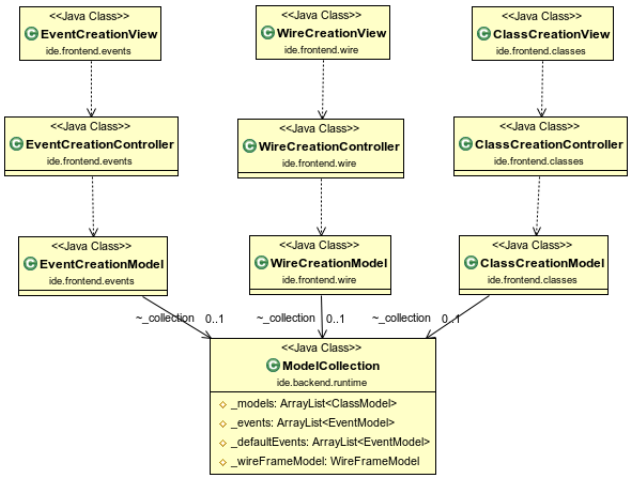
Uiteraard is er onderlinge communicatie nodig tussen deze verschillende onderdelen. Zo zal bijvoorbeeld het creeëren, aanpassen of verwijderen van een klasse of Event gevolgen hebben op meerdere plaatsen in het programma.

Voor het GUI deel van het programma wordt er sterk gebruik gemaakt van het Model-View-Controlller ontwerp patroon. Hierdoor proberen we het grafische deel van het programma niet te vervuilen met applicatie logica. Voor een goede basis te hebben, zijn we vertrokken van een MVC module aangeboden uit het vak Object-georiënteerd programmeren.

Hoewel de ModelCollection klasse alle data bevat van het programma. Hebben we ervoor gekozen om elk view een apart model tegeven. Dit omzowel de ModelCollection niet te vervuilen met View specifieke benodigdheden. Als de mogelijkheid om elke View apart te ontwikkelen. Figuur 33 toont de globale structuur van de connectie tussen de GUI en de modellen. Om de eerder ver-

melde gevolgen van aanpassingen door te geven in elk view. Zal elk model van een view de ModelCollection observeren voor veranderingen.

Het vervolg van deze sectie zal kort de structuur van de views bespreken.



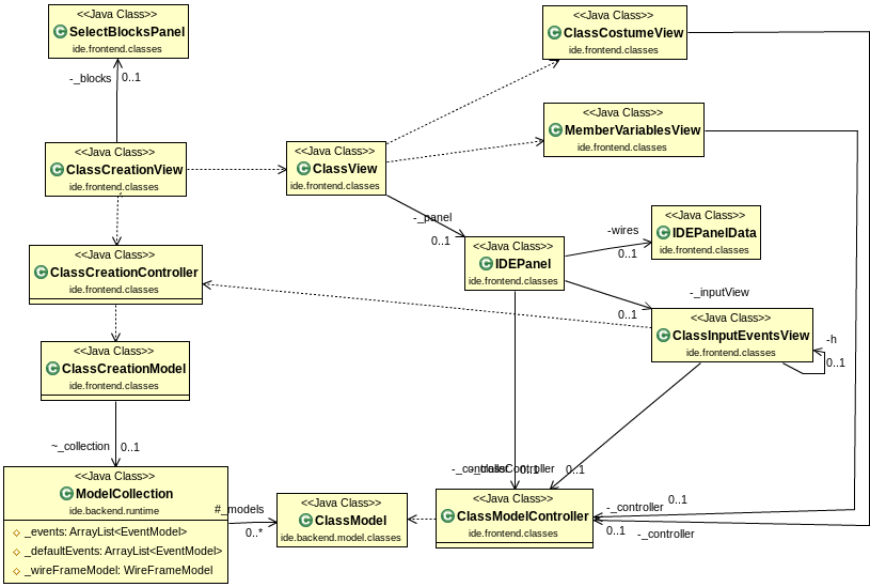
Figuur 33: *Connectie model views.*

10.1 Klasse view

Deze sectie bespreekt kort hoe de GUI van het klasse view is opgebouwd. Het ClassCreationView wordt gebruikt voor het aanmaken en tonen van de verschillende klassen die de gebruiker defineerd. Dit view bevat ook het paneel SelectBlocksPanel waarop de beschikbare blokken worden getoond. Dit view heeft ook de controllen over het aanmaken en plaatsen van blokken.

Het ClassCreationView bevat dus voor elke klasse een ClassView. Een Classview bestaat uit drie delen. Een paneel waarop blokken geplaatst worden. Een deel voor het inladen en selecteren van kostuums (ClassCostumeView). En tenslotte een deel voor het creeëren en aanpassen van member variabelen (MemberVariablesView).

Het eerste deel wordt opgedeeld in twee delen. Een paneel voor de draden tussen Handlers en inputEvents en een voor de blokken. Dit wordt besproken in Sectie 37. Het blokken deel gebeurt door de IDEPanel-klasse. Deze bevat een dateklasse (IDEPanelData) en een klasse voor het paneel met InputEvents (ClassInputEventView). Aangezien het laatste paneel toegang moet hebben tot alle bestaande Events, kan dit via een controller naar het ClassCreationModel dewelke dan weer aan alle Events kan in de ModelCollection.



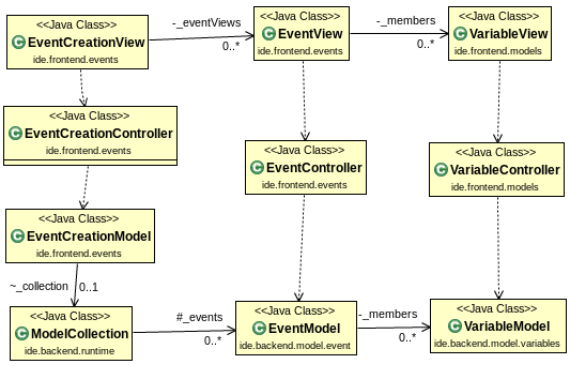
Figuur 34: *Klasse view tak van GUI.*

10.2 Event View

Deze sectie bespreekt kort hoe de GUI van het Event view is opgebouwd. Het EventCreationView wordt gebruikt voor het aanmaken en aanpassen van de verschillende Events die de gebruiker defineerd. Deze klasse bevat voor elke elk Event een EventView.

Een EventView stelt het view voor van een Event. Een EventView communiceert met zijn Model via een controller. In een EventView bevinden zich alle members van een Event. Iedere member heeft zijn EigenVariable View. Hierdoor moet bij het aanpassen van Variable enkel dit view worden aangepast.

Een variableView staat rechtstreeks in verbinding met zijn model via een controller. Hierdoor kan het type van een variable aangepast worden zonder het EventView hiermee te belasten.



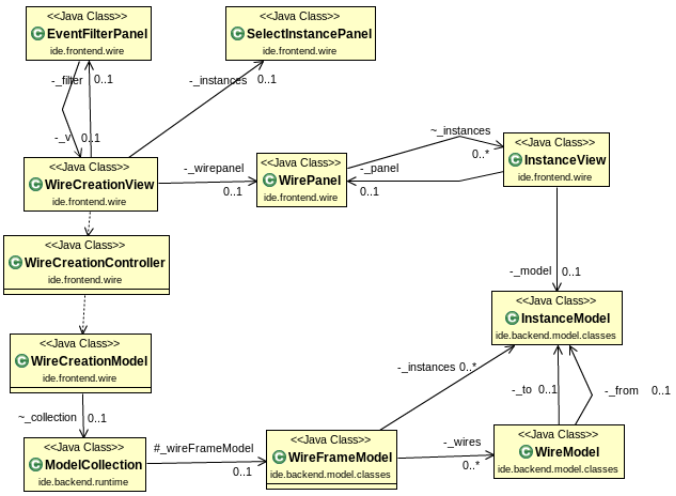
Figuur 35: *Event view tak van GUI.*

10.3 Wire View

Deze sectie bespreekt kort hoe de GUI van het Wire view is opgebouwd. Het WireCreationView bestaat uit drie delen: SelectInstancePanel, WirePanel en EventFilterPanel. SelectInstancePanel staat in voor de creatie van nieuwe instanties. WirePanel wordt gebruik voor het maken van connecties tussen instanties.

Aangezien Wires worden getekend op de manier zoals beschreven in sectie 37 is er geen nood voor een aparteView voor elke Wire. Voor het tekenen van de Wires wordt er gebuik gemaakt van de Input- en Output Event knoppen op een instanceView. Het toevoegen van een Wire gebeurt via de WireCreationController.

EventFilterPanel regelt welke Wires er moeten getekend worden op het WirePanel.



Figuur 36: *Event view tak van GUI.*

10.4 Drag and drop

Het tekenen van blokken en het drag en drop systeem hebben we zelf geïmplementeerd. De reden om een eigen drag-and drop systeem te implementeren vindt u in secite 5.1. De custom drag-and drop maakt gebruik van AWT Shapes en van Swing JComponents.

Voor de drag-and drop zijn enkel de meest externe blokken gekend. Door een simpele berekening (kijken of de muis zich binden een externe blok zit) kan de blok gevonden worden waarover de gebruiker hovert. Zodra het systeem identificeert over welke externe blok de gebruiker een blok loslaat, kan het er berekent worden waar in de hiërarchie de losgelaten blok geplaatst moet worden.

Een blok kent zijn specifieke regels. Hij kan zeggen of hij een blok kan bevatten, of een andere blok mogelijks genest kan worden, en hij kan dit opvragen aan zijn kind-blokken indien hij deze heeft. De meeste externe blok vraagt de meest geneste blok aan die de losgelaten blok kan bevatten.

Hierna wordt bepaald welke visuele veranderen er moeten optreden, zoals het verhogen van zijn hoogte. Een blok kan de hoogtes en breedtes van zijn kind-blokken opvragen.

Omdat enkel de meest externe blokken gekend zijn aan het drag-and drop systeem moeten alle JComponents gebonden zijn aan de externe blokken. Een blok kan aan zijn kinderen zijn JComponents (en hun locaties relatief aan hun eigenlijke parent-blok). Deze berekent nieuwe locaties en kan deze aan zijn parent geven als deze bestaat, of de JComponents binden aan zichzelf zodat ze zicht-

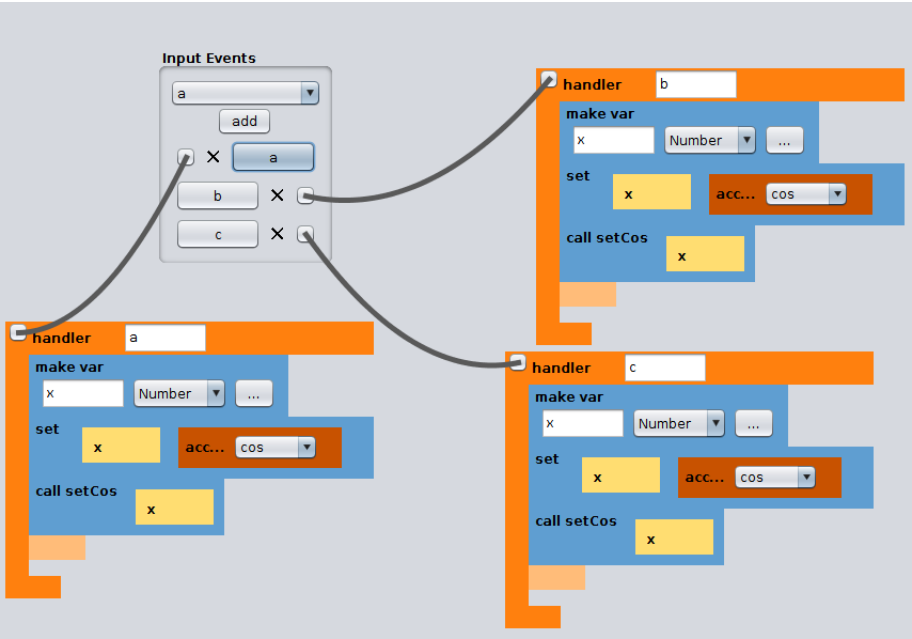
baar worden op het scherm.

10.5 Wires

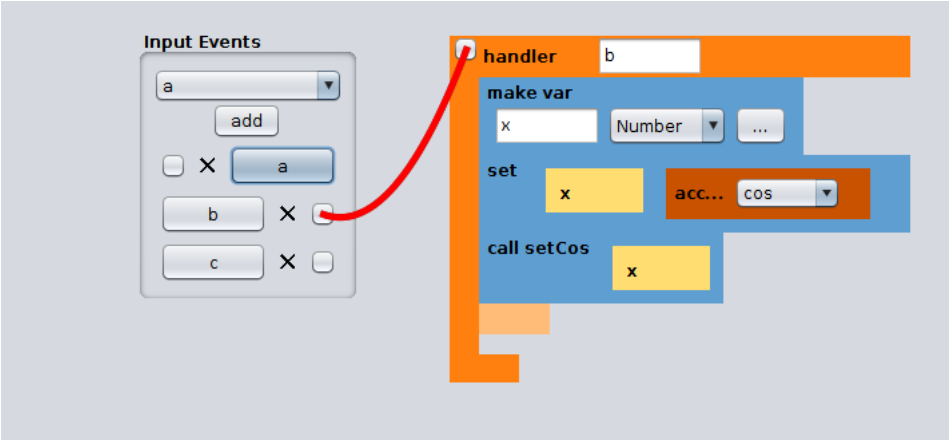
In de IDE wordt er zowel voor het verbinden van Events tussen instances in het WireFrame als voor verbinden van Input Events en Handlers in ClassView gebruik gemaakt van wires die automatische worden getekend. Voor het tekenen wordt er gebruik gemaak van `QuadCurve2D` [15]. Wires worden altijd op de voorgrond getekend door gebruik te maken van een `JLayeredPane` [16]. Deze bestaat steeds uit twee panels namelijk een panel voor de wires en een voor de eigenlijke blokken. Het eerste wordt louter gebruikt voor het tekenen van de Wires en zal geen input events van de gebruiker opvangen.

Als er een linker muis klik wordt geregistreed op het panel van de blokken zal er berekend worden als er op een wire wordt geklikt. Als dit het geval is zal de eerste Wire die de klik bevat als geselecteerde Wire worden genomen.

Bij het bewegen van het panel waarop de blokken zich bevinden of het bewegen van de blokken zelf zal ervoor zorgen dat alle wires opnieuw moeten worden berekend en opnieuw moeten worden getekend.



Figuur 37: *Wires.*



Figuur 38: *Wire selected.*

11 Opslaan en Inladen

Deze sectie bespreekt hoe een bestand wordt opgeslaan en hoe problemen bij het inladen van een semantisch incorrect programma worden afgehandeld. Daarna wordt ook besproken hoe ondersteuning voor meerdere talen mogelijk werd gemaakt in de appplicatie.

11.1 Keuze XML

Voor het opslaan van een programma moest een leesbaar formaat worden gekozen. Eerder in dit verslag werd al aangehaald dat een programma kan beschouwd worden als een boom structuur. De combinatie van deze twee aandachtspunten en het bestaan van betrouwbare verwerkings bibliotheken voor XML hebben ervoor gezorgd dat er gekozen werd voor XML.

In het analyseVerslag werd er gesteld dat data wordt opgeslagen naar meerdere files. Dit is echter een vervelende implementatie omdat de IDE dan meerdere files aanmaakt die aanpasbaar moeten zijn door de gebruiker. Voor de gebruiker is het simpelder indien er slechts èèn XML file was om aan te passen. De beschrijving van de XML voor alle geïmplementeerde blokken en XML DTD is achteraan beschreven in bijlage D.

Het volgende deel van deze sectie beschrijft de eigenlijke implementatie van het opslaan en inlezen van een programma. Als ook het afhandelen van een semantisch incorrect programma bij inladen.

11.2 Opslaan en inladen van een programma

Deze sectie beschrijft kort hoe een programma wordt opgeslaan en hoe het wordt ingeladen. Voor het inladen worden ook complexere zaken zoals het inladen van variabelen en functies besproken. Alsook de gevolgen van het inladen van een semantisch foutief programma.

Opslaan

Het opslaan van een progamma gebeurt in XML. Dit zal gebeuren zoals beschreven in Bijlage D.

Voor het opslaan wordt gebruik gemaakt van de boomstructuur gecreërd door de **modellen** (Sectie 8) deze zijn centraal verzameld in de ModelCollection (Sectie 8.4). Er is dus enkel nood aan een module die deze boom structuur afloopt en opslaat naar een bestand. Echter hebben we ervoor gekozen om het opslaan van een model los te koppelen van hun klasse. Dit om de klasse niet te vervuilen en de mogelijkheid te hebben om gemakkelijk een ander bestandsformaat te kiezen voor het opslaan van een programma. Er werd een Interface DataSaver geëimplementeerd die deze uitbreidbaarheid verzekerd.

Hierdoor duikt er in Java het probleem op waarbij er niet automatisch de functie wordt genomen met de meest specifieke klasse van een instantie. Om dit op te lossen werd er ook hier gebruik gemaakt van het Visitor patroon zoals beschreven in Sectie 9.5

Voor het opslaan van **kostuums** van een klasse hebben we gekozen om de ingeladen afbeeldigen te kopiëren naar de folder waar de gebruiker het programma wenst op te slaan. Elke afbeelding wordt hernoemt naar een combinatie van de gekozen kostuums naam en de klasse waarbij deze behoort. De paden van de afbeeldingen worden relatief opgeslaan t.o.v de gekozen folder.

Inladen

Voor het inladen van het programma wordt geëist dat de XML correct gedefinieerd is. Bijlage D beschrijft deze definitie. De volgende paragrafen beschrijven het gedrag van de applicatie bij gebruik van incorrecte semantiek in verschillende delen van het bestand.

Events Als eerste worden de Events ingeladen. Aangezien deze overal in het programma kunnen gebruikt worden. Een Event moet uniek zijn. Bij het inlezen van een duplicaat Event zal dit Event worden overgeslaan. De members van een Event moeten uniek zijn. Bij het inlezen van een duplicate member zal deze member worden overgeslaan.

Bij het inlezen van blokken die gebruik maken van Event zal het bestaan van het Event gecontrolleerd worden.

Variabelen Het programma wordt per klasse ingeladen. Klasse naam moet uniek zijn. Duplicate namen worden overgeslaan. Voor een klasse worden eerst de membervariablen ingeladen ook deze moeten uniek zijn. Membervariabelen worden gestokeerd in een lijst. Bij het inlezen van handler of functie zal deze lijst worden aangevuld als er een definitie van een lokale variable wordt ingelezen.

Bij het inlezen van een referentie naar een variable wordt nagekeken als deze variablen gedefinieerd is. Zoja, dan wordt er een referentie naar deze variable gecreeërd. Anders zal de referentie niet worden ingeladen. Maar het programma wordt wel verder ingeladen. Voor elke functie of handler wordt er vertrokken vanuit de lijst met member variabelen.

Functies Functies moeten een unieke naam hebben. Bij een duplicate functie wordt deze overgeslaan.

Functie aanroepen Bij het tegenkomen van een functie oproep wordt deze ingeladen en ingevuld met de gewenste parameters en return. Echter worden alle functie oproepen apart opgeslaan zodat deze later gekoppeld kunnen worden aan hun gewenste functie. Als na het inlezen van alle functies, de functie waarnaar de functie oproep refereert niet gevonden is, zal het programma niet worden ingeladen.

Handlers Als bij het inladen van een Handler een onbekend Event wordt gegeven zal deze handler geen Event worden toegekend. De handler en zijn body worden wel ingeladen.

Kostuums Bij ongeldig pad naar een afbeelding of het beschreven bestand is geen afbeelding zal de er geen afbeelding worden getoond in het programma maar zal het inladen gewoon verder gaan.

Instanties De naam van een instantie moet uniek zijn. Bij het inladen van een duplicate instantie zal deze worden overgeslaan.

Wires Duplicate wires worden overgeslaan. Bij het inladen van een wire tussen onbekende instanties of met een onbekend Event. Zorgt ervoor dat vanaf dat punt geen instanties of wires worden ingeladen.

11.3 Multilanguage IDE

De visuele programmeer IDE moet multilanguage zijn. De gebruiker moet kunnen wisselen tussen verschillende talen. Aangezien we Java gebruiken zijn we gaan kijken naar standaard klassen om multilanguage applicaties te maken. De oplossing is de ResourceBundle Class die Java aanbiedt [17].

Een ResourceBundle laad automatisch de nodige file gegeven een bepaalde filenaam (en eventueel een locale). Vervolgens kan via een getString("label") de

tekst in de gevraagde taal opgevraagd worden. Een locale beschrijft een bepaalde taal, zo kan er een UK engels, en een US engels gemaakt worden [18]. Hieronder staat beschreven hoe de files voor verschillende talen genoemd moeten worden. De inhoud van deze files volgt een simpel formaat nl, key = tekst [19].
Voorbeeldcode: [19]

```
public class Main {
       public static void main(String[] args) {
              // De constructor heeft 2 parameters:
              // een taal en een land
              Locale locale = new Locale("en", "UK");
              // language.properties is een file
              ResourceBundle language =
                  ResourceBundle.getBundle("language", locale);
              System.out.println(language.getString("label"));
       }
}
```

De verschillende files:

```
language.properties
language_en.properties
```

Een mogelijke inhoud van `language.properties`:

```
label1 = een bepaalde tekst
label2 = een andere tekst
```

Een mogelijke inhoud van `language_en.properties`:

```
label1 = a certain text
label2 = another text
```

11.4 Omzetting van Modellen naar Views

Bij het gebruik van GUI zal elk view eerst bij reset de bestaande modellen opvragen en hiervoor views creeëren. Voor het inladen van een klasse zullen de blokken ingeladen worden en gekoppeld worden aan views. Dit is mogelijk aangezien een View kan werken door views aan elkaar toe te voegen maar ook door enkel modellen te koppelen.

Het aanmaken van de Views wordt gedaan door de LoadClassViewFromModel klasse. Deze zal de blokken van een ClassModel koppelen aan hun views en op het IDEPanel plaatsen. Hier werdt er geen gebruik gemaakt van het Visitor patroon zodat een model niet vervuild wordt met de functionaliteit van view creatie.

12 Mogelijke Uitbreidingen

Er zijn verschillende uitbreidingen mogelijk. Sommige van deze uitbreidingen zijn de toevoeging van kleine features, andere uitbreidingen brengen grotere veranderen aan de IDE. Alle functionaliteit die we beloofd hadden in het analyseverslag hebben we kunnen implementeren.

12.1 Extra blocks invoegen

Er kunnen extra blokken bijgemaakt worden. Om een blok te maken moeten er enkele zaken aangepast/aangemaakt worden. Het vereist veel stappen om een nieuwe blok toe te voegen, maar hierdoor blijven de verschillende verantwoordelijkheden goed gescheiden van elkaar.

- Een executie block, afgeleid van de basis klasse Block. Deze bevat de uitvoeringslogica.
- Een model, afgeleid van de basis klasse BlockModel De bevat de model logica, zoals error en type checking.
- De view, afgeleid van de basis klasse BlockView. Deze bevat het uiterlijk van de view
- De controller, afgeleid van AbstractBlockController. Deze vormt een brug tussen het model en de view.
- Een compile functie in de Compiler.
- Een save functie in de DataSaver.
- Een load functie in de DataLoader.
- Een entry in SelectBlocksPanel, zodanig dat de gebruiker de blok kan gebruiken.
- Een entry in de LoadClassViewFromModel, zodanig dat de omzetting van de model naar het view gebeurt..

12.2 Selectie via muis

De gebruiker zou hierbij een veld kunnen selecteren met zijn cursor. Alle blokken binnen dit veld zullen dan gelijktijd geselecteerd zijn. Ze kunnen dan samen verplaatst of verwijderd worden.

12.3 Meerdere return waardes

Zoals vermeld in Sectie 9.6 ondersteund de uitvoeromgeving meerdere returnwaardes voor functies. De GUI ondersteund dit niet. Een simpele uitbreiding zou zijn om de GUI dit wel te laten implementeren. Er is modulair gewerkt om parameters en returnwaardes visueel te plaatsen. Het overstappen van één returnwaarde naar meerdere werkt slechts een kleine aanpassing aan het visuele view.

12.4 Clone blocks

Een handige feature zou het clonen van blokken zijn in het klasse-view. Code duplicatie is uiteraard een teken van slechte software. Echter is het kopiëren van enkele blokken tijdbesparend. Als de gebruiker drie blokken apart moet plaatsen, zal hij meer tijd verliezen dan wanneer hij deze zou kunnen kopiëren.

Deze feature zou gemakkelijk gerealiseerd kunenn door een clone functie te implementeren op de modellen, en de views opnieuw te genereren. Een andere mogelijk zou zijn om de te-clonen blok om te zetten naar zijn xml representatie en deze terug in de laden. Het inladen genereert al modellen en views. Hiervoor is geen verandering nodig aan de model klasse.

12.5 Undo/Redo

Een undo of redo actie is ook een nuttige feature voor gebruikers. Een mogelijke implementatie zou zijn om een stack bij te houden van de veranderen die plaats vinden in de IDE. De undo knop zou deze veranderen vervolgens kunnen reversen.

12.6 Meerdere projecten

De mogelijkheid om meerdere projecten open te hebben staan in de IDE zou ook een handige feature kunnen zijn. Dit vereist slechts een kleine aanpassing aan de IDE. Achterliggend kan de IDE een lijst bijhouden van XML-data. Als de gebruiker een ander project opent, word het huidge project opgeslagen in de lijst, en het andere project aangemaakt of ingeladen. Aan de gebruikers kant kan dit getoond worden als verschillende tabbladen voor de verschillende projecten.

12.7 Java bindings

Een meer geavanceerde uitbreiding zou de mogelijkheid tot Java-bindings zijn. De gebruiker zou dan zelf blokken kunnen aanmaken. Dit vereist een grotere aanpassing. Het aanmaken van een blok zou versimpelt moeten worden zodanig dat de gebruiker zich niet moet bezighouden met interne werkingen van inladen, opslaan, compileren, etc. Om dit te realiseren zou reflectie gebruikt kunnen worden in combinatie met een goed systeem wat alle interne werking zoals het opslaan, inladen en compileren afhandelt.

12.8 Extra data structuren

Het toevoegen van extra data-structuren zoals lijsten zou ook handig zijn voor complexere programma's. In de IDE wordt een base class variable gebruikt, hiermee kunnen simpel nieuwe data types aangemaakt kunnen worden. Voor het manipuleren van deze data zouden extra blokken aangemaakt moeten worden (zoals een getVariableOnIndex blok voor een lijst). Het toevoegen van extra data structuren vereist niet veel meer werk dan het toevoegen van nieuwe blokken.

12.9 Meer debug opties

De toevoeging vane extra debug opties zoals het tonen van statestieken zodat de gebruiker kan zien hoeveel processen er zijn, enz. De terugkoppeling van zulke statestieken is mogelijk via het observer patroon dat gebruikt wordt voor de debugging. Deze data moet enkel gegenereerd worden in de virtual machine, doorgegeven worden via het observer patroon en dan kan het view deze informatie tonen in een dialoog.

12.10 Static functions

Momenteel behoren functies tot een bepaalde klasse. Een functie van klasse A kan niet gebruikt worden door klasse B. Om code herbruikbaarder te maken en code duplicatie te vermijden zouden statische functies, functies die door elke klasse gebruikt kunnen worden, een goede toevoeging zijn.

Aan de backend moet weinig veranderen voor deze uitbreiding. De ModelCollection zou een lijst van functies bevatten die dan aan elke gecompileerde klasse meegeven wordt zodanig dat de klasse deze functies kan gebruiken. Aan de front-end moet wel veel veranderen. Er moet een extra view ingevoerd worden waarin de gebruiker deze functies kan maken. Dit view zal gelijkaardig zijn aan het klasse-view, echter kunnen hierin geen member variabelen gebruikt worden. Het nieuwe view kan bijvoorbeeld afleiden van dezelfde basis klasse als het klasse-view.

Referenties

[1] MIT, "Scratch." `https://scratch.mit.edu/`, 2015. [Online; accessed 15-Februari-2015].

[2] Google, "Blockly." `https://developers.google.com/blockly/`, 2015. [Online; accessed 15-Februari-2015].

[3] E. Games, "Unreal Engine 4." `https://www.unrealengine.com`, 2015. [Online; accessed 15-Februari-2015].

[4] bobbylight, "RSyntaxTextArea." `https://github.com/bobbylight/RSyntaxTextArea`, 2015. [Online; accessed 28-may-2015].

[5] oracle.com, "Introduction to DnD." http://docs.oracle.com/javase/tutorial/uiswing/dnd/intro.html, 2015. [Online; accessed 28-may-2015].

[6] wikipedia.com, "Visitor Pattern." http://en.wikipedia.org/wiki/Visitor_pattern, 2015. [Online; accessed 27-May-2015].

[7] wikipedia.com, "Observer pattern." http://en.wikipedia.org/wiki/Observer_pattern, 2015. [Online; accessed 27-May-2015].

[8] oracle.com, "Observable." http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html, 2015. [Online; accessed 27-May-2015].

[9] wikipedia.com, "Event-driven-programming." http://en.wikipedia.org/wiki/Event-driven_programming, 2015. [Online; accessed 15-Februari-2015].

[10] S. Ferg, "Event-Driven Programming: Introduction, Tutorial, History." http://Tutorial_EventDrivenProgramming.sourceforge.net, 2006. [Online; accessed 15-Februari-2015].

[11] wikipedia.com, "Concurrent Computing." http://en.wikipedia.org/wiki/Concurrent_computing, 2015. [Online; accessed 15-Februari-2015].

[12] wikipedia.com, "Concurrent Computing." http://en.wikipedia.org/wiki/Deadlocks, 2015. [Online; accessed 15-Februari-2015].

[13] oracle.com, "Class HashMap<K,V>." http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html, 2015. [Online; accessed 19-Februari-2015].

[14] oracle.com, "Lambda Expressions." http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html, 2015. [Online; accessed 19-Februari-2015].

[15] oracle.com, "QuadCurve2D." http://docs.oracle.com/javase/7/docs/api/java/awt/geom/QuadCurve2D.html, 2015. [Online; accessed 27-May-2015].

[16] oracle.com, "JLayerdPane." http://docs.oracle.com/javase/7/docs/api/javax/swing/JLayeredPane.html, 2015. [Online; accessed 27-May-2015].

[17] oracle.com, "Class ResourceBundle." http://docs.oracle.com/javase/7/docs/api/java/util/ResourceBundle.html, 2015. [Online; accessed 15-Februari-2015].

[18] oracle.com, "Class Locale." http://docs.oracle.com/javase/7/docs/api/java/util/Locale.html/, 2015. [Online; accessed 15-Februari-2015].

[19] jenkov.com, "Java ResourceBundle." http://tutorials.jenkov.com/java-internationalization/resourcebundle.html/, 2015. [Online; accessed 15-Februari-2015].

[20] oracle.com, "Class ArrayList<E>." http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html, 2015. [Online; accessed 19-Februari-2015].

A Bijlage Gebruikersdocumentatie

Deze sectie zal een nieuwe gebruiker wegwijs maken in de visuele programmeer omgeving. Deze documentatie is onderverdeeld in enkele delen: programma creatie, menu en functionalieit hierin, debugging en de data editor.

A.1 Installatievereisten

AxeSki vereist Java versie 8 of hoger.

A.2 Algemene uitleg programmeer taal

AxeSki is een visuele programmeer omgeving. Het hoofddoel van de IDE is om event-based programma's te kunnen schrijven. Een gebruiker kan standaard programma's maken net zoals in andere programmeertalen. Een bijkomend voordeel is dat de gebruiker goed leert hoe event-based programming werkt.

De gebruiker werkt voornamelijk met **Events**, dit zijn informatiepakketten die in een programma rondgezonden kunnen worden tussen verschillende entiteiten. Deze events worden aangeduid door een unieke naam. De informatie die erin zit wordt ook aangeduid door een unieke naam binnen het event.

Vervolgens kan de gebruiker met behulp van programmeerstructuren een **klasse** opbouwen. Een klasse is een entiteit binnen het programma die kan reageren op bepaalde Events, en er ook zelf kan uitsturen. Deze programmeerstructuren omvatten de standaard programmeer constructies zoals een while-loop, if-condities, operaties, variabelen, etc. In een klasse bepaald de gebruiker hoe hij inkomende events afgehandeld. Een klasse kan ook terug events verzenden, hierin bepaald de gebruiker welke informatie in het te versturen event plaatst.

A.3 Datatypes

In de programmeer omgeving zijn drie verschillende datatypes beschikbaar. Namelijk Booleaanse waarde, Numerieke waarde en Strings. Ook kan er gebruik worden gemaakt van letterlijke waarden.

A.4 Programma creatie

Een programma wordt opgebouwd in drie delen. Namelijk het aanmaken van Events, definiëren van klasse en aanmaken van instanties en verbindingen maken tussen deze instanties. Alle drie de delen kunnen parallel worden opgebouwd. Elk deel gebeurt in een ander component van de GUI. Deze sectie helpt de gebruiker te werken met elk van deze componenten.

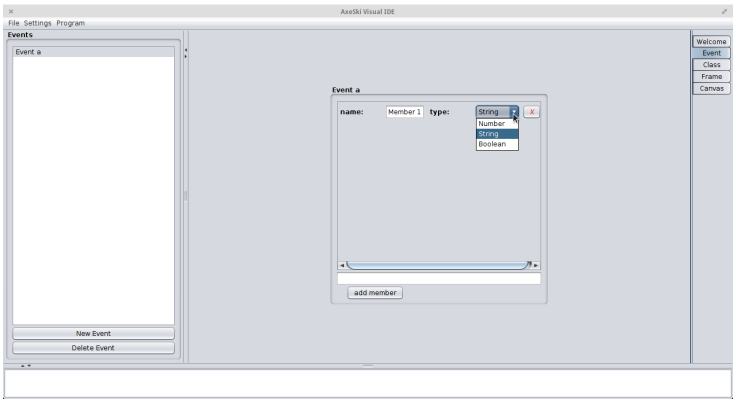
Event creatie

Zoals eerder vermeld kunnen instanties met elkaar communiceren via het doorsturen van Events. Naast de standaard beschikbare InputEvents zoals MousePressed kan de gebruiker zelf Events defineren. Deze Events zullen dan beschikbaar zijn voor de definitie van klassen.

In het Event creatie venster heeft de gebruiker aan de linkerkant een overzicht van alle Events die hij reeds gedefineerd heeft. Een nieuw Event toevoegen kan via de new Event knop. Het verwijderen kan via de "Delete Event" knop.

Bij het aanmaken van een nieuw Event zal een unieke naam moeten geven worden. Na het succesvol aanmaken van een Event kan het Event gedefineerd worden. Een Event kan gezien worden als een pakketje data dat verstuurd wordt. Om de data van een Event in te vullen of op te vragen moet de data verdeeld worden in verschillende variabelen. Deze variabelen worden Members genoemd.

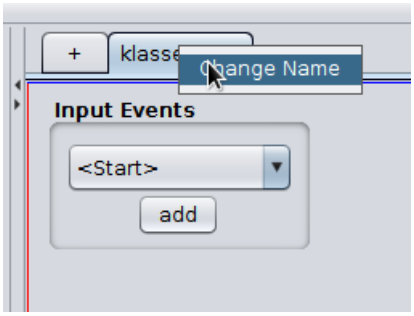
De gebruiker kan Members toevoegen aan zijn Events door een unieke naam in te geven in het input veld onder de reeds bestaande members. Na het aanmaken kan het type van een member op elk tijdstip veranderd worden. Een Member kan verwijderd worden door op het rode kruisje te drukken.



Figuur 39: *EventView.*

Klasse creatie

In het klasse view kan er een nieuwe klasse worden toegevoegd door op de "+" tab te drukken. Hierna wordt gevraagd om een unieke naam te geven voor de nieuwe klasse. De naam van een klasse kan veranderd worden door rechts te klikken op de tab van de klasse waarvan men de naam wenst te veranderen. Het venster voor een klasse te creeëren bestaan uit vier delen.

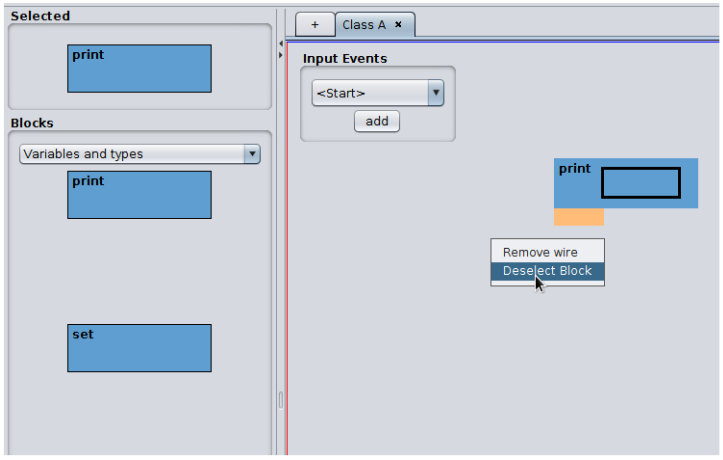


Figuur 40: *Veranderen klasse naam.*

Blokken selectie

Aan de linkerkant bevinden zich alle verschillende blokken die gebruikt kunnen worden. Deze blokken zijn onderverdeeld in verschillende categorieen. Deze staan kort vermeld in Sectie A.8. Het selecteren van een blok kan door links op de gewenste blok te drukken. De huidige geselecteerde blok is zichtbaar in de linkerboven hoek van dit venster.

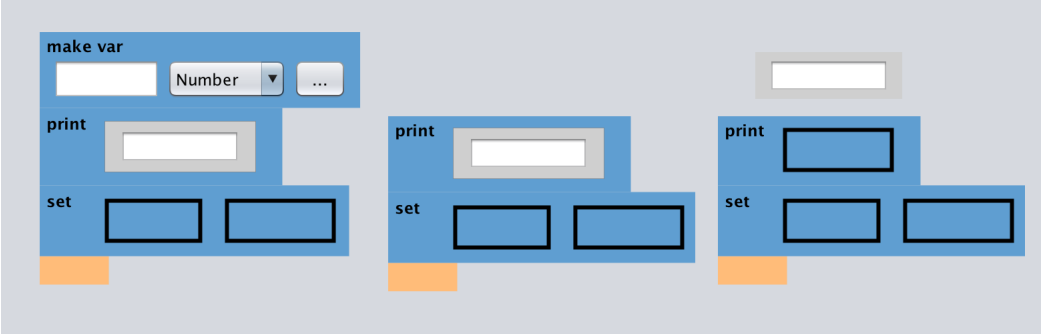
Als een blok geselecteerd is kan deze op het paneel worden geplaatst van een klasse. De selectie blijft actief tot de gebruiker een blok versleept of tot de gebruiker de selectie ongedaan maakt. Dit kan door rechts op het paneel te drukken de "deselect block" optie te kiezen. Rechts klikken op een blok zal de optie "remove block" tonen. Deze optie verwijderd de volledige groep blokken waarvan de aangeklikte blok deel van uitmaakte als ook de blokken die deze blok zelf bevat.



Figuur 41: *Selected block.*

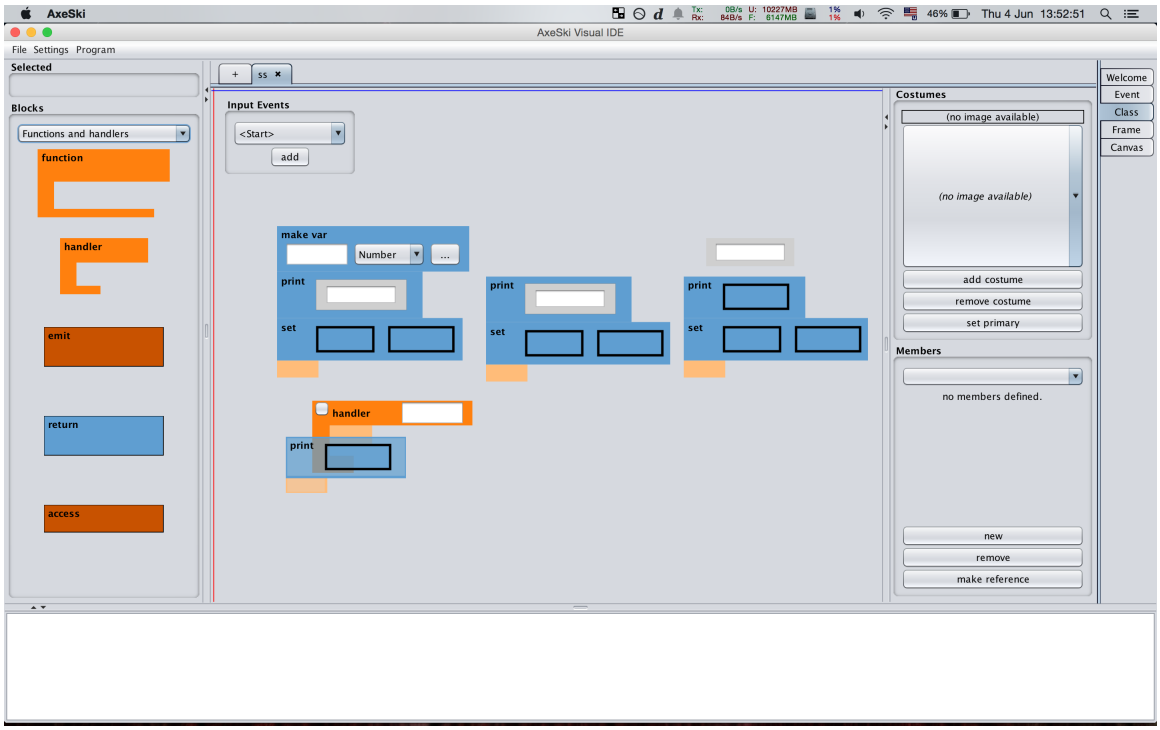
Verslepen blokken

Een blok kan versleept worden door deze vast te nemen met de muis cursor. Als de blok een body vormt zoals te zien in Figuur 42, moet de bovenste blok vastgenomen worden. Als een andere blok in de body vastgenomen wordt, zal deze losklikken. Als een geneste blok vastgenomen wordt, zal deze ook losklikken uit zijn ouder blok. Dit is te zien in Figuur 42. Hierbij toont de linkerkant een volledige body. Het midden toont hetgene losgeklikt wordt wanneer de print blok versleept zou worden. Het rechtergedeelte toont wat er gebeurt als de value blok versleept wordt.



Figuur 42: *Losklikken blokken.*

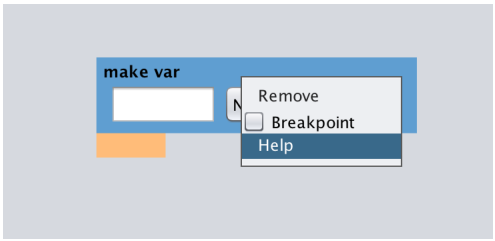
Als een blok versleept wordt, zal deze doozichtig worden zoals te zien in Figuur 43.



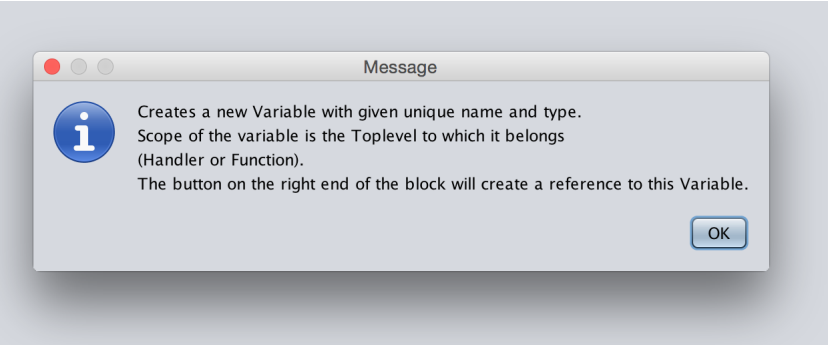
Figuur 43: *Losklikken blokken.*

Hulp menu

De gebruiker kan een hulp menu openen voor een bepaalde blok. Deze actie kan uitgevoerd worden door de rechtermuisknop in te drukken en op de hulp optie te drukken zoals te zien in Figuur 44. Dit hulp menu geeft een korte samenvatting van de functie van die bepaalde blok, te zien in Figuur 45



Figuur 44: *Hulp menu.*



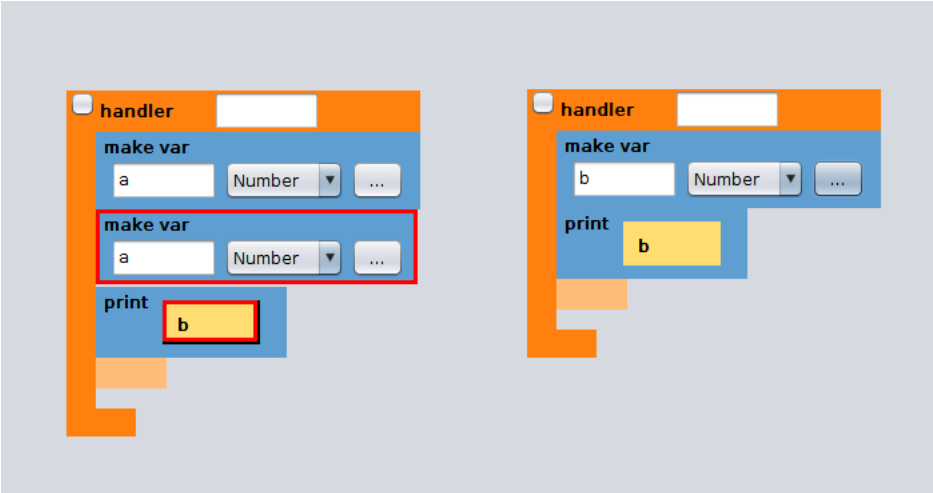
Figuur 45: *Hulp dialog.*

Klasse paneel

Het middelste deel van het scherm is gereserveerd voor het plaatsen van blokken. Hierin kan de gebruiker de werking van een klasse definiëren. Dit paneel is van oneindige grootte. De gebruiker kan doorheen het paneel verplaatsen door het paneel vast te nemen en het weg te trekken.

Lokale variabele Binnen een functie of handler kan er gebruik worden gemaakt van variabelen. Deze kunnen aangemaakt worden doormiddel van een MakeVar-blok en deze toe te voegen aan een functie of handler. De variabele bestaat dan enkel in de scope van de functie of handler. De naam van die variabele moet dus ook uniek zijn in die scope. Als dit niet het geval is zal de de MakeVar-blok die de nieuwe variabele definiërt een rode rand krijgen zoals te zien in Figuur 46.

Om gebruik te maken van een variabele kan er een referentie gemaakt worden naar de variabele. Dit gebeurt via de knop op het einde van de MakeVar-blok. Zoals al eerder vermeld bestaat een variabele enkel in de scope of definitie waarin zich zijn definitie (MakeVar-blok) bevindt. Als een referentie zich in een andere scope bevindt dan zijn definitie zal de referentie een rode rand tonen zoals in Figuur 46.



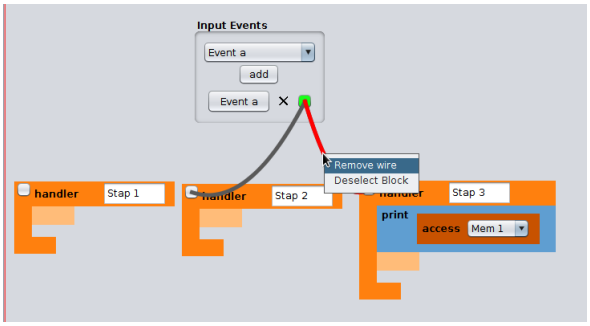
Figuur 46: *Variabele*

InputEvents en Handlers Standaard bevindt zich in de linkerboven hoek van het klasse paneel,het InputEvents paneel. Dit paneel geeft aan op welke inputEvents de klasse reageert. Bovenaan dit paneel bevindt zich een dropdown menu waarin alle standaard Events en door de gebruiker gedefineerde Events ter beschikking zijn. De gebruiker kan een inputEvent aan een klasse toevoegen door het gewenste Event te selecteren en op "add" te drukken. Het verwijderen van een inputEvent kan door op het kruisje naast de naam van het InputEvent te drukken.

Een Handler is een blok die de gebruiker kan gebruiken om een InputEvent op te vangen. Hij kan een Handler toekennen aan een InputEvent door op het vierkantje in de linkerboven hoek te drukken en daarna op het vierkantje naast de naam van het InputEvent te drukken.

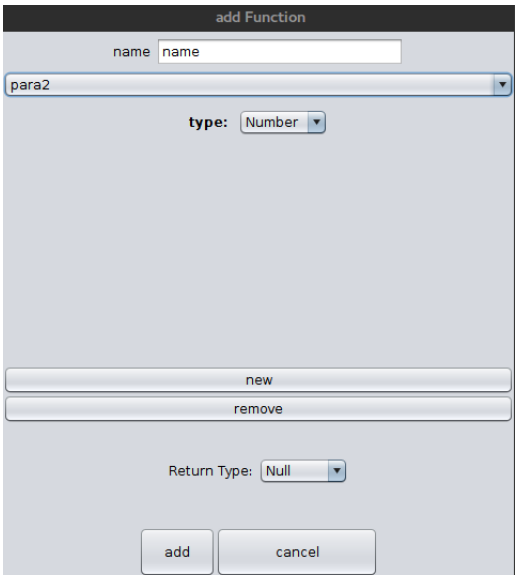
Een verbinding kan verwijderd worden door het kabeltje aan te klikken en vervolgens op rechts te klikken op het paneel en de optie "remove wire" te selecteren. Voor een beter overzicht te hebben kan de gebruiker op de naam van het InputEvent drukken zodat de connectie aan de linker of rechter kant van het paneeltje staat.

Als een InputEvent members heeft kan deze opgevraagd worden doormiddel van een AccessBlock. Deze zal de members tonen van het InputEvent waarmee de Handler waarin hij zich bevindt mee verbonden is.



Figuur 47: *Werking InputEvents en Handler.*

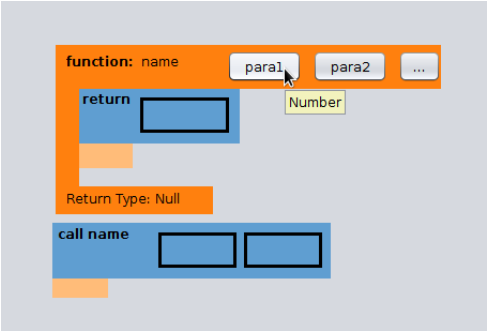
Functies Bij het selecteren van een Functie blok zal er een dialoog worden getoond waarin de gebruiker de functie header kan beschrijven. Hierin zal het het elke parameter van de functie moeten definiëren. Alsook het returntype van de functie. Na het voltooien van deze dialoog kan de gebruiker de functie verder invullen.



Figuur 48: *Aanmaken van een functie.*

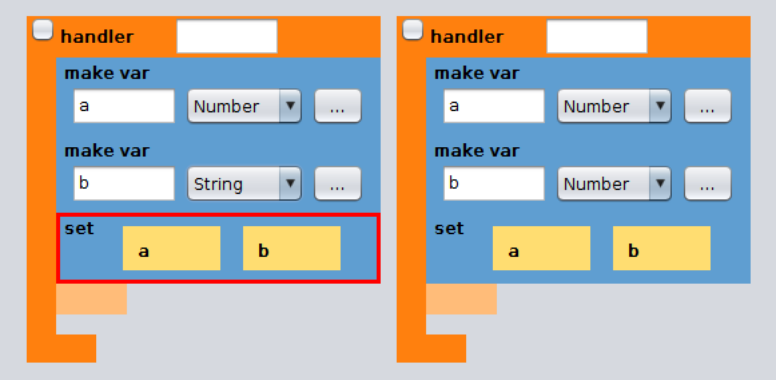
Het type van een parameter kan worden verkregen door te hoveren over de naam van de parameter. Een referentie naar een parameter kan worden gecreeërd door op de naam van de parameter te drukken.

Een functie oproep naar een functie kan worden aangemaakt door op het vierkantje te drukken dat zich na de laatste parameter van de functie blok bevindt.



Figuur 49: *Functie blok en functie aanroep blok.*

Typechecking Bij het gebruik van blokken waarbij enkel bepaalde types toegelaten zijn, geeft het programma feedback over de types. Dit gebeurt doormiddel van een rode rand. Deze error verhinderen echter het uitvoeren van het programma niet. Bij de uitvoering zal in de console een gepaste error weergeven worden.



Figuur 50: *Typechecking*

Voor een blok specifieke toegelaten blokken kan er naar de help functie van een blok worden gekeken. Voor de mogelijke combinaties van het gebruik van een binaire of unaire operatie kan de tabel in Figuur 51 geraadpleegd worden.

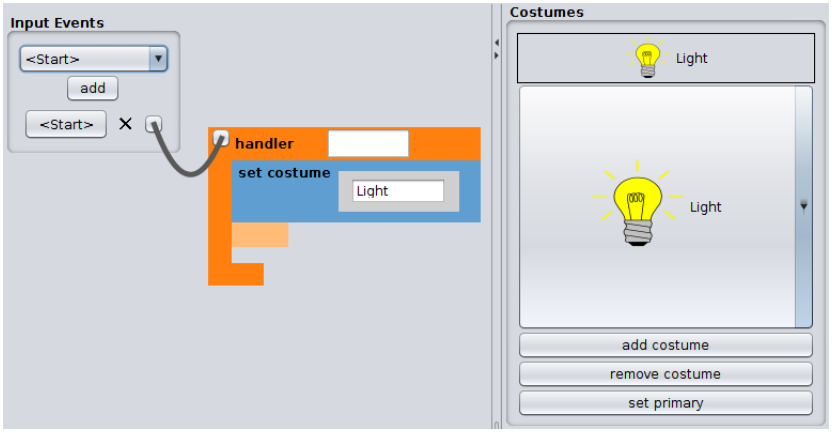
Links	Rechts	Operator	Geeft
Booleaans	Booleaans	== , || , &&	Booleaans
Booleaans	Waarde	== , || , &&	Booleaans
String	String	==, < , >	Booleaans
String	String	+	String
String	Waarde	==, < , >	Booleaans
String	Waarde	+	String
Numeriek	Numeriek	==. <,>	Booleaans
Numeriek	Waarde	%,*, -, ^,/,+	Numeriek
Numeriek	//	sqrt	Numeriek
Booleaans	//	!	Booleaans

Figuur 51: *Operator combinaties.*

Kostuums In de rechterboven hoek van het klasse creatie venster bevindt zich een paneel waarin er kostuums aan een klasse kunnen worden toegevoegd. Het dropdown menu geeft een overzicht van alle reeds ingeladen kostuums. Deze kostuums kunnen dan in de klasse definitie gebruikt worden. Zo kan de gebruiker het uitzicht van een instantie van een klasse doen veranderen doormiddel van een set costume blok. Hierin kan een variabele of letterlijke waarde worden geplaatst. De waarde daarvan wordt dan bekeken als String. Als er een kostuum bestaat met die gegeven String als naam zal het gezet worden als uiterlijk voor die instantie.

Voor het selecteren van een standaard afbeelding van instanties van een klasse kan gebruik worden gemaakt van de "set primary knop" in het kostuum paneel. Het primaire kostuum wordt verkleint weergegeven bovenaan het paneel.

Bij het toevoegen van een nieuw kostuum zal een unieke naam moeten worden gekozen. Bij het mislukken van inladen of selecteren van een niet ondersteund bestandstype zal er geen afbeelding worden getoond. Het verwijderen van een kostuum kan door de "remove" knop te gebruiken. Dit zal het huidige geselecteerde kostuum in het dropdown menu verwijderen. Als dit kostuum het primaire kostuum was, zal het eerste kostuum als primair worden geplaatst.

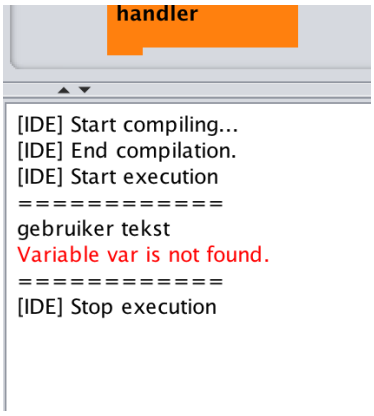


Figuur 52: *Kostuum paneel.*

Member variabelen Onder het Costume View bevindt zich het paneel voor member variabelen. Deze variabelen zijn beschikbaar dooreen de hele klasse. Om een referentie naar een member variable aan te maken, kan de gebruik de gewenste member selecteren en op "make reference" drukken. De geselecteerd blok (linksboven) toont nu een variable blok. Deze kan de gebruiker plaatsen op het panel zoals andere blokken. Een member variable zijn type kan aangepast worden via het dropdown menu.

Console

Er is een console aanwezig waar de gebruiker output naar kan schrijven. Error output zoals variabelen die niet gevonden worden, worden in rode tekst uitgeprint. Standaard output wordt geprefixed door `[IDE]`.

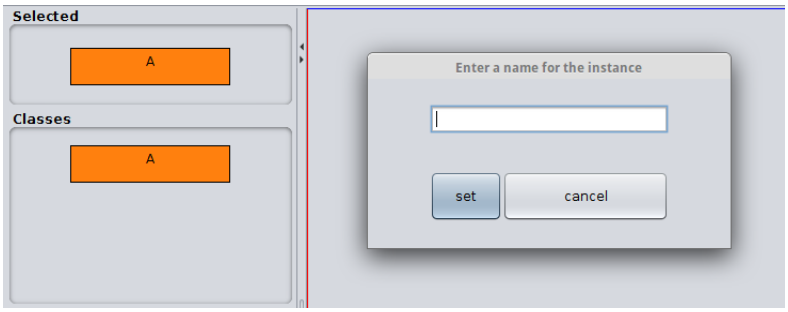


Figuur 53: *Console output.*

Instanties en verbindingen

Deze sectie beschrijft hoe een gebruiker instanties van klassen kan aanmaken. En hoe hij de communicatie hier tussen kan defiëren doormiddel van wires.

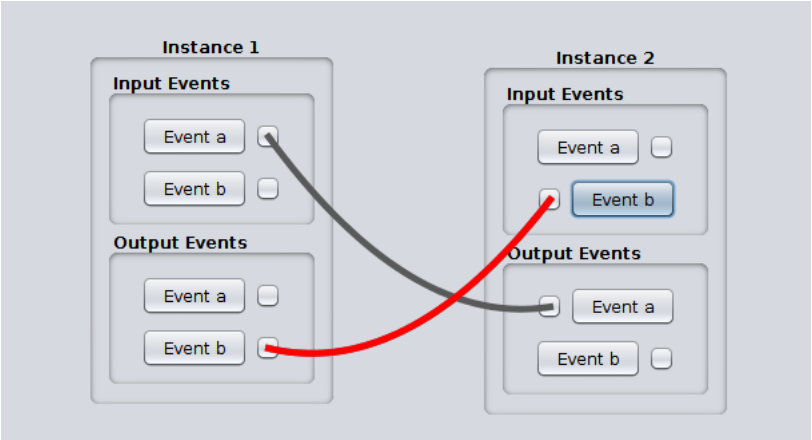
Aanmaken en verwijderen van een instantie Aan de linkerzijde bevind zich een lijst met alle bestaande klasse. Bij het aanklikken van een klasse zal deze geselecteerd worden. Hierna kan er op het paneel gedrukt worden om een instantie van de geselecteerde klasse aan te maken. Er zal gevraagd worden aan de gebruik om een unieke naam te geven voor een instantie. Een instantie kan verwijderd worden door rechts te klikken op een instantie en de optie "remove" te selecteren.



Figuur 54: *Aanmaken van instantie.*

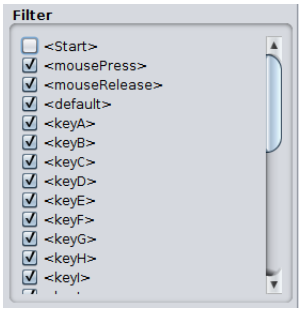
Verbindingen tussen instanties Instanties kunnen Events naar elkaar sturen via wires. Wires kunnen enkel verbonden worden tussen dezelde Input- en OutputEvents van hetzelfde type. De gebruiker kan dit door het vierkantje naast een eventnaam van een InputEvent aan te klikken en vervolgens hetzelfde te doen voor een OutputEvent van hetzelfde type.

Verbindingen kunnen verwijderd worden door een wire aan te klikken en vervolgens bij het rechtskikken de optie "remove wire" te kiezen.



Figuur 55: *Events tussen instanties.*

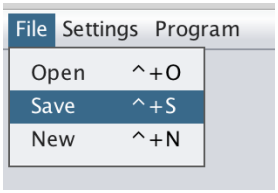
Filter Aan de linkerzijde van het Frame bevindt er zich een filter die alle bestaande Events in het programma bevat. Standaard is een Event aangevinkt. Dit betekend dat wires die dit type Event versturen getoond worden op het paneel. Als een Event is uitgevinkt zullen deze wires niet getekend worden.



Figuur 56: *Filter.*

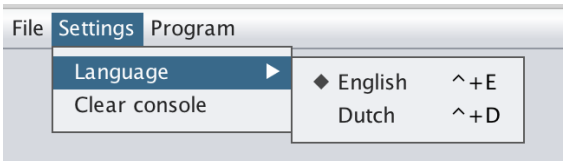
A.5 Menu

De user kan verschillende acties voltooien via het menu. Het filemenu is het menu waarmee de gebruiker interactie kan hebben met het opslaan en inladen van programma's. Zoals te zien in Figuur 59 kan de gebruiker hier files openen en opslaan. Ook kan een nieuw project gestart worden. Dit zal alle huidige progressie verwijderen. Alle opties zijn beschikbaar via sneltoetsen.



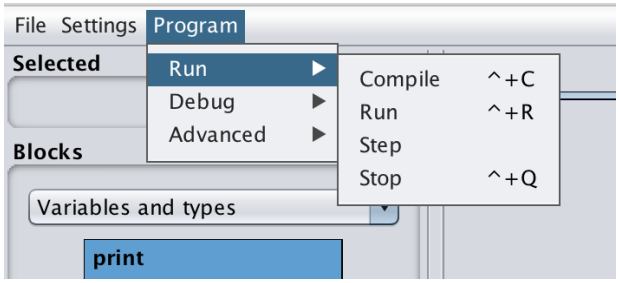
Figuur 57: *File menu.*

In het settings menu kan de gebruiker de console leegmaken, ook kan hier de taal veranderd worden. Er is de keuze tussen Nederlands en Engels. Het wisselen tussen de talen is zoals te zien in Figuur 58 ook beschikbaar via sneltoetsen.



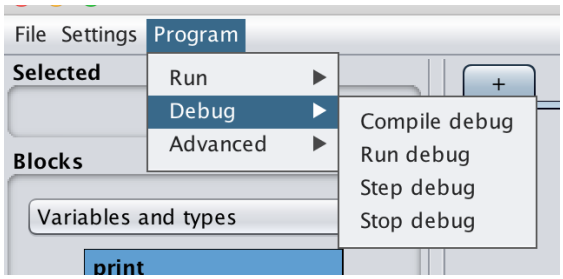
Figuur 58: *Settings menu.*

De gebruiker kan het programma uitvoeren via dit menu. Er staan enkele opties geschikbaar. Een programma kan gecompileerd worden, kan uitgevoerd worden, kan gestopt worden en er kan doorheen het programma gestapt worden.



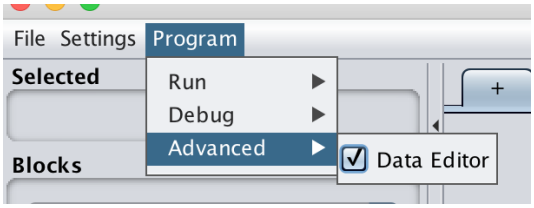
Figuur 59: *Run menu.*

Het debugmenu heeft dezelfde functionaliteit als het runmenu. De gebruiker gebruikt hiermee het programma in debug modus. De functionaliteit van deze modus is uitgelegd in Sectie A.6



Figuur 60: *Debug menu.*

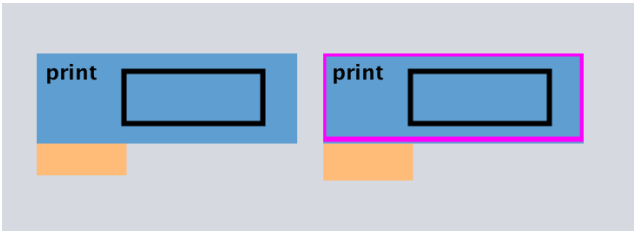
Het data menu heeft een optie om de Data Editor aan te zetten of uit te zetten.



Figuur 61: *Data menu.*

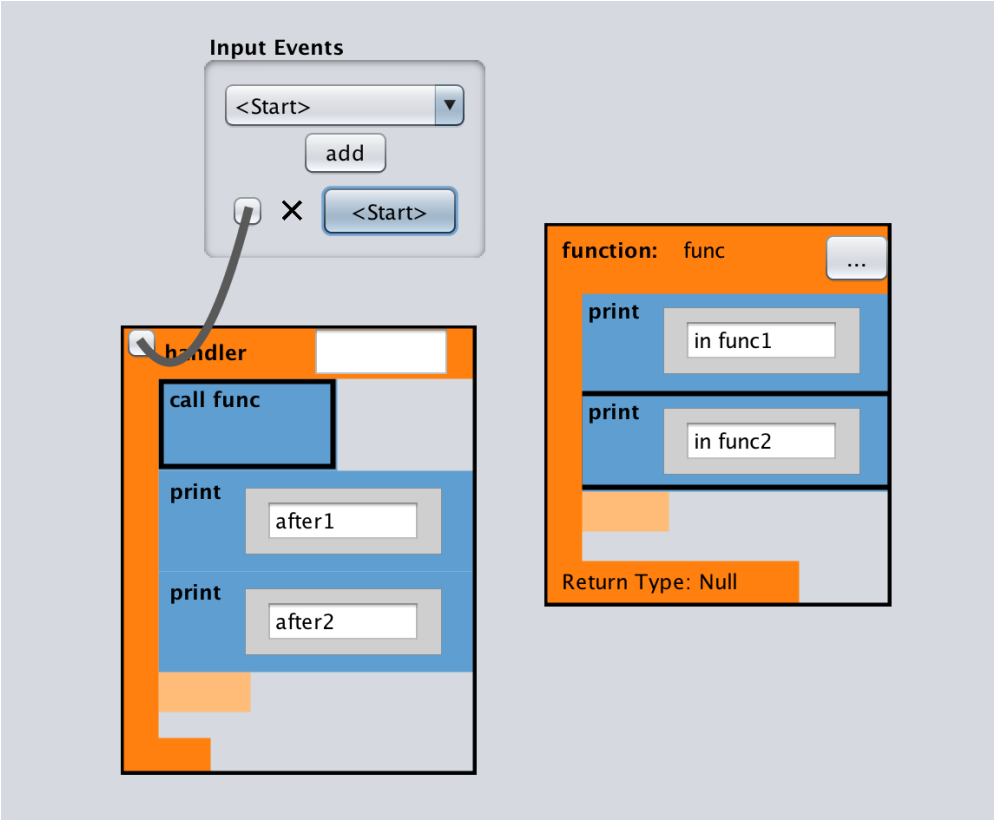
A.6 Debugging

In de IDE kan de gerbuiker debuggen. Blokken kunnen breakpoints bevatten. Dit kan getoggled worden door een optie in een popup menu. Dit menu wordt geopend door de rechtermuisklik in te drukken op een blok. Die blok krijgt vervolgens een paarse rand zoals te zien in Figuur 62.



Figuur 62: *Console output.*

De gebruiker kan doorheen code stappen in debug modus. Dit toont een zwarte rand rond de momenteel uitvoerende blokken.



Figuur 63: *Console output.*

A.7 Data editor

De Data Editor is een ingebouwde tekst editor om een programma te kunnen maken of aanpassen in zijn XML-vorm. Deze editor heeft syntax hightlighting zoals te zien in Figuur 64. De data editor heeft zal aanpassingen doorvoeren naar de visuele views wanneer het view veranderd. Bij een foutieve XML zal een nieuw programma aangemaakt worden.

File Settings Program

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<program>
    <events>
        <event type="event a">
            <member name="member 1" type="NUMBER"/>
            <member name="member 2" type="STRING"/>
        </event>
        <event type="event b"/>
    </events>
    <classes>
        <class name="voorbeeld">
            <events>
                <inputEvent type="&lt;Start&gt;"/>
            </events>
            <emits/>
            <memberVariables/>
            <handlers>
                <handler event="&lt;Start&gt;" name="" x="147.0" y="418.0">
                    <block x="23.0" y="32.0"/>
                </handler>
            </handlers>
            <functions>
                <function name="functie" x="583.0" y="40.0">
                    <params/>
                    <block x="20.0" y="40.0"/>
                </function>
            </functions>
            <floatingBlocks>
                <block x="237.0" y="149.0">
                    <setVar x="0.0" y="0.0"/>
                    <print x="0.0" y="53.0"/>
                </block>
            </floatingBlocks>
            <costumes/>
        </class>
    </classes>
    <wireframe>
        <instances/>
        <wires/>
    </wireframe>
</program>
```

Figuur 64: *Data Editor.*

A.8 Overzicht van bestaande blokken

Deze sectie geeft een overzicht van alle blokken per categorie.

Variabelen

Bevat alle operaties om variabele aan te maken. Gelijk te stellen of uit te printen.

- print
- value
- make var
- set

Condities

Bevat alle conditie blokken.

- while
- if
- if-else
- forever

Functies en handlers

Bevat handler en functie blokken. En operaties voor het opvangen van informatie uit een Event of het versturen van een Event.

- emit
- access
- function
- handler
- return

Physics

De blokken in deze categorie kunnnen gebruik worden voor de positie of uiterlijk van een instantie te veranderen.

- move
- show
- hide
- set costume

Locks

Bevat blokken die gebruikt kunnen worden voor het locken en unlocken van variabelen.

- lock
- unlock

Trivia

Bevat blokken die in geen connectie hebben met andere blokken.

- sleep

Math

Bevat blokken voor wiskundige bewerkingen alsook het aanmaken van een willekeurig getal.

- unaire operator
- binaire operator
- random

String

Bevat blokken voor operaties op variabele van het type String.

- length
- concat
- charAt

B Bijlage Reflectie

Het project is van significante grootte en was hierdoor een veeleisend project. Dit droeg echter wel toe tot de voldoening bij het in onze ogen succesvolle afronding van het project. Desondanks dat het project al zwaar was, hebben we toch nog extra functionaliteit kunnen toevoegen. Het project heeft bijgedragen tot de kennis van beide teamleden over specifieke implementatie details alsook de verkregen ervaring met het omgaan met grote software projecten.

In het begin wisten zowel de opdrachtgever als wij niet meteen in welke richting het project moest opgaan. Door ideeën uit te wisselen zijn we tot een mooi resultaat bekomen waar we zelf trots op zijn. De feedback van de opdrachtgever bij zowel de presentatie als de voorstelling van het prototype heeft bijgedragen tot het eindproduct. We hebben verschillende feedbackmomenten gevraagd aan onze begeleider zodanig dat we altijd op de correcte weg bleven.

Beiden waren we zeer gedreven om zorgvuldig het project af te maken. Axel kan snel een goed overzicht krijgen met betrekking tot het een globaal efficiënt ontwerp is. Matthijs kan dit ontwerp goed in detail brengen en zorgt dat niets uit het oog verloren wordt zodat het eenvoudig te implementeren is.

Tijdens de analyse hadden we veel aandacht voor de uitwerking van de uitvoeromgeving. Echter hadden we weinig tijd gestoken in de uitwerking van de GUI. Hierdoor hadden we wel een bijna foutloze uitvoeromgeving maar moesten we nog veel nadenken over de implementatie van de GUI. In onze planning hadden we ook de benodigde tijdsduur voor het maken van de GUI onderschat. Dit leverde weinig problemen op aangezien we vroegtijdig begonnen waren aan de implementatie van de uitvoeromgeving. Door het programma te ontwikkelen in modules en bij de uitwerking van de modules het goed opsliten in kleinere deelproblemen konden we onze tijd efficiënt besteden en zaten we eenzelfde component.

C Bijlage Log

C.1 Taakverdeling

Het aanmaken van de variabelen, excepties en uitvoeringsblokken zijn door beide teamleden geïmplementeerd. De basis voor de modellen is gemaakt door Matthijs, terwijl Axel de virtual machine uitbouwde. Matthijs heeft de Proces klasse uitgewerkt. Axel heeft de Compiler en Runtime in elkaar gestoken. Axel heeft de DataLoader gemaakt terwijl Matthijs de DataSaver maakte. De modellen werden hierna verder uitgebreid door beide teamleden. Matthijs maakte de event creatie terwijl Axel het menu en de Data editor maakte. Beide teamleden hebben hierna gewerkt aan het implementeren van de drag-and-drop. Eenmaal de basis voor drag-and-drop gelegd was heeft Axel dit verder uitgewerkt terwijl Matthijs de class en instance creatie aanmaakte.

Matthijs implementeerde de communicatie tussen modellen en de views zodanig dat Axel het debuggen kon uitwerken. Axel heeft verschillende bugs verwijderd en details afgewerkt, Matthijs werkte aan het view voor de member variabelen en de costumes. Matthijs implementeerde typechecking en de event-filter terwijl Axel verschillende nieuwe blokken implementeerde. Functies zijn gemaakt door beide teamleden. Uiteindelijk hebben beide teamleden de IDE nog getest en gedebugged.

C.2 Analyse

- 7 januari 2015: Keuze top 3 onderwerpen en motivatie.
- 20 januari 2015: Voorbereiding mockups voor opdrachtgever.
- 26 januari 2015: Meeting met opdrachtgever.
- 27 januari 2015: Mockups en verslag van de eerste meeting met de opdrachtgever.
- 29 januari 2015: Afspraak met begeleider Jonny Deanen voor bespreking interpretatie.
- 4 februari 2015: Uitwerking voorstel voor opdrachtgever.
- 6 februari 2015: Afspraak met begeleider Jonny Daenen m.b.t uitwerking van voorstel.
- 8 februari 2015: Inzending voorstel voor de opdrachtgever.
- 12 februari 2015: Beschrijving van opslag formaat (XML) en mogelijke blokken.
- 14 februari 2015: Begin van beschrijving, interpretatie, multilanguage en begin evaluatiecriteria.

- 15 februari 2015: Bestaande software (Scratch, Blockly en Unreal). Begin beschrijving Event-driven programming en concurrent computing. Keuze programmeertaal en begin modules.
- 16 februari 2015: Bespreking werking klasse en UML.
- 17 februari 2015: Toevoeging klassen: Class, Instance, Proces, EventDispatcher, VM, Event, EventPool, EventInstance, WiredInstance, WireFrame, ClassPool. En UML ervan.
- 18 februari 2015: Verder werken aan klassen van Blokken en bijhorende UML.
- 19 februari 2015: Beschrijving extra's en prioritaire functies. Uitleg gebruik van Lambda functies en bronnen. Toevoeging aan evaluatiecriteria.
- 20 februari 2015: Bespreking huidige status van verslag met begeleider Jonny Daenen.
- 22 februari 2015: Toevoeging voorbeeld proces. Herschrijving van diepgaande beschrijving met een nieuwe inleiding. Verplaatsen van secties naar bijlagen.
- 23 februari 2015: begin UML en beschrijving GUI models.
- 24 februari 2015: Mockups begonnnen en afwerking beschrijving GUI models.
- 25 februari 2015: Beschrijving en verdeling modules
- 26 februari 2015: Mockups en de beschrijving ervan. Herordening van modules. Toevoeging nieuwe inleiding. Gesprek van huidige toestand met begeleider Jonny Daenen.
- 27 februari 2015: Toevoeging uitleg design patroon compiler. Invullen van Log en taakverdeling.

C.3 Implementatie

- 30 maart 2015: Start implementatie, exceptions, variabelen en operators zijn aangemaakt
- 31 maart 2015: Aanmaak van basis blokken zoals class, instance, event, blocks en het aanmaken van een proces. Aanmaak van enkele blokken zoals de acces blok, print blok. Aanmaak van de basis voor Models. Creatie van de virtual machine en event dispatcher.
- 1 april 2015: Uitbreiding modellen, aanmaak compiler en runtime.
- 2 april 2015: Aanmaken van verschillende modellen. Uitbreiding van de compiler. Aanmaak van functie modellen.

- 5 april 2015: Compile functies toegevoegd, accesmodel en refEventmodel.
- 6 april 2015: DataSaver interface en DataLoader interface.
- 7 april 2015: DataSaver en DataLoader implementatie. Functiecall model, emitblock en Instancemodel. Aanmaak Language module.
- 8 april 2015: Wireframe model, operatormodel. DataSaver opslaan naar zowel file als string.
- 8 april 2015: Backend is vervolledigd en getest.
- 11 april 2015: Creatie menu bar, modellen, Data editor en start main GUI frame.
- 12-13 april 2015: Start aan eventcreatie, aanmaken aparte thread voor de console. Aanmaken tabbladen systeem
- 14 april 2015: MVC event en variable.
- 14 april 2015: Afspraak met begeleider.
- 15 april 2015: Afspraak begeleider voor probleem met output naar console.
- 16 april 2015: Afmaken data edtior en main frame, werk aan event view. Eerste prototype drag-and-drop. Inladen event van file.
- 17-18-21-22 april 2015: Drag-and-drop
- 21 april 2015: Start class view
- 22 april 2015: Block views
- 23 april 2015: Block view work and class creation. Handler view implementatie start. SelectBlocksPanel en representative view is gemaakt. Toevoegen blokken aan panel.
- 23 april 2015: Afspraak met begeleider.
- 25 april 2015: Wireframe, input events van classes en prototype van wires.
- 26 april 2015: Refactoring Drag-and-drop.
- 27 april 2015: Wires tekenen is geïmplementeerd.
- 28 april 2015: Color scheme gekozen voor de blokken, verdere implementatie handler view. Werk aan de layouting van de IDE. Connecties gemaakt tussen models en views. GUI navigatie (tabs en welcome scherm). Testing van volledige pipeline van de IDE. Aanmaken if en while blokken voor de volledige IDE.
- 29 april 2015: Correcte implementatie Drag-and-drop. Switch button voor input events, instanceview layouting. Algemeen werk aan views.

- 30 april 2015: Aanmaken van Jar voor prototype
- 1 mei 2015: Demonstratie project.
- 6 mei 2015: Aanmaak variabele en referenties. Communicatie emit view en emit model. Globaal werk aan views en modellen.
- 7 mei 2015: Verbetering van file menu. Setblock en inladen variable blocks.
- 8 mei 2015: Toevoeging oneindig veld voor IDE. Correct inladen van wires. Start aan rechterpaneel klassen.
- 9-10 mei 2015: Debugging en Breakpoints toevoegen. Member variabelen maken.
- 11 mei 2015: Image selector. Costume view, verwijderen wires in wireframe. Verwijderen van warnings zodat project warning-free is.
- 12 mei 2015: Delete input events in wireframe en aanmaken canvas en input events. Opslaan images.
- 12 mei 2015: Afspraak met begeleider.
- 14-15 mei 2015: Opslaan costumes in xml, globaal werk costume view. Change Appearance blok. Implementatie van locksen unlocks. String operators, sleep, show, hide forever en move blokken zijn geïmplementeerd.
- 15 mei 2015: Hotkeys implementeren, if-else model.
- 16 mei 2015: Schrijven van enkele missende comments. Fixen bugs in verschillende views.
- 17 mei 2015: Event-filter gemaakt.
- 18 mei 2015: Bug fixes
- 19 mei 2015: Typechecking, emit en acces views.
- 22-24 mei 2015: Unaire blok, crash fixes, welcome screen glitch, emit en if-loading. Blocks kunnen in een if snappen. Typechecking condities.
- 25 mei 2015: Functie en functiecall implementatie. Layouting van de GUI.
- 26 mei 2015: Afspraak met begeleider.
- 26 mei 2015: Kleurenschema, emit typechecking, bug fixes. Verwijderen debug statements.
- 1 juni 2015: Afspraak met begeleider.
- 27 mei - 4 juni 2015: Verslag

D Bijlage XML Blokken

D.1 Variables en Type

Variabele blok

De `makeVar` blok heeft een bepaalde unieke naam. Dit element bevat een type element.

```
<makeVar name="name of variable" type="type" x="0" y="0" />
```

De DOCTYPE declaration:

```
<!ELEMENT makeVar (type)>
<!ATTLIST makeVar name CDATA #REQUIRED type CDATA #REQUIRED x
    CDATA #IMPLIED y CDATA #IMPLIED >
```

Value

De `value` blok bevat een string die de data voorstelt.

```
<value> value </value>
```

De DOCTYPE declaration:

```
<!ATTLIST value x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT value (#PCDATA)>
```

Set blok

De `setVar` blok bevat een variabele en een value die geassigned zal worden aan deze variabele.

```
<setVar name="name">
       <value> value </value>
</setVar>
```

De DOCTYPE declaration:

```
<!ATTLIST setVar name CDATA #IMPLIED x CDATA #IMPLIED y CDATA
    #IMPLIED>
<!ELEMENT setVar
    (var|value|concat|length|charat|random|operator)>
```

Print blok

De `print` blok bevat een block die uitgeprint zal worden naar de console.

```
<print>
       <value> value </value>
</print>
```

De DOCTYPE declaration:

```
<!ATTLIST print x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT print (var|value|concat|length|charat|random|operator)>
```

var blok

De `var` blok heeft een naam waarmee een variable mee kan worden aangesproken of gemanipuleerd.

```
<var name="varName"/>
```

De DOCTYPE declaration:

```
<!ELEMENT var EMPTY>
<!ATTLIST var name CDATA #REQUIRED>
```

null blok

Wordt gebruikt om in bepaalde situaties "geen waardeäan te duiden.

```
<null/>
```

De DOCTYPE declaration:

```
<!ELEMENT null EMPTY>
```

D.2 Locks

Lock

De `lock` blok bevat een variabele die gelocked moet worden.

```
<lock name="" />
```

De DOCTYPE declaration:

```
<!ATTLIST lock name CDATA #IMPLIED x CDATA #IMPLIED y CDATA
    #IMPLIED>
```

Unlock

De `unlock` blok bevat een variabele die geunlocked moet worden.

```
<unlock name="name" />
```

De DOCTYPE declaration:

```
<!ATTLIST lock name CDATA #IMPLIED x CDATA #IMPLIED y CDATA
    #IMPLIED>
```

D.3 Control Blokken

Forever block

De `forever` block bevat een block code dat wordt uitgevoerd.

```
<forever>
  <block> code </block>
</forever>
```

De DOCTYPE declaration:

```
<!ATTLIST forever x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT forever (block)>
```

If blok

De `if` blok bevat een conditie en een code blok.

```
<if>
  <cond> condition </cond>
  <block> code </block>
</if>
```

De DOCTYPE declaration:

```
<!ATTLIST if x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT if (cond,block)>
```

If-else blok

De `if-else` blok bevat een conditie en twee code blokken.

```
<if-else>
  <cond> condition </cond>
  <block> if-code </block>
  <block> else-code </block>
</if-else>
```

De DOCTYPE declaration:

```
<!ATTLIST if-else x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT if-else (cond,block,block)>
```

Conditie blok

De `conditie` bevat een variable of een logische expressie.

```
<cond>
  <var name="varName"/>
</cond>
```

De DOCTYPE declaration:

```
<!ELEMENT cond (var|value|concat|length|charat|random|operator)>
```

While blok

De `while` bevat een conditie en een code blok.

```
<while>
  <cond> condition </cond>
  <block> code </block>
</while>
```

De DOCTYPE declaration:

```
<!ATTLIST while x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT while (cond,code)>
```

D.4 Physics

Move blok

De `move` blok stelt een translatie voor.

```
<move x="0.0" y="0.0">
    <x/>
    <y/>
</move>
```

De DOCTYPE declaration:

```
<!ELEMENT move (x,y)>
<!ATTLIST move x CDATA #IMPLIED y CDATA #IMPLIED>
```

Show

De `show` blok maakt een instance zichtbaar of doet niets indien de instance al zichtbaar was.

```
<show />
```

De DOCTYPE declaration:

```
<!ATTLIST show x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT show EMPTY>
```

Hide

De `hide` blok maakt een instance onzichtbaar of doet niets indien de instance al onzichtbaar was.

```
<hide />
```

De DOCTYPE declaration:

```
<!ATTLIST hide x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT hide EMPTY>
```

Change appearance

De `changeAppearance` blok maakt het mogelijk voor een instance om zijn uiterlijk te veranderen. De id is de ID van zijn nieuw uiterlijk.

```
<changeAppearance id="0"/>
```

De DOCTYPE declaration:

```
<!ELEMENT appear EMPTY>
<!ATTLIST hide x CDATA #IMPLIED y CDATA #IMPLIED>
```

D.5 String operators

Concat

De `concat` blok laat toe om 2 strings samen te voegen.

```
<concat>
       <null>
       <var name="right var to concat">
<concat>
```

De DOCTYPE declaration:

```
<!ATTLIST concat x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT concat
    (var|value|concat|length|charat|random|null|operator,
    var|value|concat|length|charat|random|operator)>
```

Length

De `strlen` blok laat toe om te lengte van een string op te vragen.

```
<length>
       <var name="string">
</length>
```

De DOCTYPE declaration:

```
<!ATTLIST length x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT length
    (var|value|concat|length|charat|random|operator)>
```

CharAt

De `charAt` blok laat toe om character op een bepaalde index op te vragen.

```
<charat>
       <value> index </value>
       <var name="string"/>
<charat>
```

De DOCTYPE declaration:

```
<!ATTLIST charat x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT charat
    (var|value|concat|length|charat|random|null|operator,
    var|value|concat|length|charat|random|operator)>
```

D.6 Trivia

Sleep blok

De `print` blok bevat een block die bepaald hoeveel milliseconden er geslapen moet worden.

```
<sleep>
       <value> value </value>
</sleep>
```

De DOCTYPE declaration:

```
<!ATTLIST sleep x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT sleep (var|value|concat|length|charat|random|operator)>
```

D.7 Operators

Random

De `random` blok stelt een functie voor die een random gekozen getal teruggeeft tussen de meegegeven bounds.

```
<random/>
```

De DOCTYPE declaration:

```
<!ATTLIST random x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT random EMPTY>
```

Operator

De `operator` blok stelt een operator voor zoals +,-,/,*,<,>,... . Dit gebeurt met twee operatoren. Als de tweede kind-element `dummy` is, dan is de operator een unaire operator.

```
<operator op="+">
    <NULL/>
    <dummy/>
</operator>
```

De DOCTYPE declaration:

```
<!ELEMENT operator
    (NULL,|var|value|concat|length|charat|random|operator ,
    NULL|dummy|var|value|concat|length|charat|random|operator)>
<!ATTLIST operator op CDATA #REQUIRED x CDATA #IMPLIED y CDATA
    #IMPLIED>
```

D.8 Functions en Handlers

Handler

De `handler` blok stelt een speciale functie voor die een event opvangt.

```
<handler name="name" event="type of event">
       <block>
              code
       </block>
</handler>
```

De DOCTYPE declaration:

```
<!ELEMENT handler (block)>
<!ATTLIST handler name CDATA #required event CDATA #IMPLIED x
    CDATA #IMPLIED y CDATA #IMPLIED>
```

Param

De `param` blok stelt een parameter voor van een functie, deze heeft een type en een identifier.

```
<param type="string" name="name1"/>
```

De DOCTYPE declaration:

```
<!ELEMENT param EMPTY>
<!ATTLIST param type CDATA #REQUIRED name CDATA #REQUIRED>
```

Return function

Het `return` element stelt een return voor van een functie, deze heeft een type.

```
<return type="string"/>
```

De DOCTYPE declaration:

```
<!ELEMENT return EMPTY>
<!ATTLIST return type CDATA #REQUIRED>
```

Params

De `params` blok is een collectie voor alle parameters van een class.

```
<params>
       <param type="string" name="name1"/>
</params>
```

De DOCTYPE declaration:

```
<!ELEMENT params (param)*>
```

Function

De `function` blok stelt een functie voor die opgeroepen kan worden in een ander stuk code van deze class.

```
<function name="name">
    <return type="NULL"/>
     <params>
           <param type="string" name="name1"/>
           <param type="string" name="name2"/>
    </params>
        <block>
               code
        </block>
</function>
```

De DOCTYPE declaration:

```
<!ELEMENT function (param*, block)>
<!ATTLIST function name CDATA #REQUIRED x CDATA #IMPLIED y CDATA
    #IMPLIED>
```

Return Blok

Een `return` blok bevat variabelen die hij returned. Volgorde is hier van belang.

```
\lstset{language=XML}
<return>
       <var name="varName" />
</return>
```

De DOCTYPE declaration:

```
<!ATTLIST return x CDATA #IMPLIED y CDATA #IMPLIED>
<!ELEMENT return (var)*>
```

FunctionCall blok

Een `FunctionCall` blok stelt een functie oproep voor en bevat variabelen voor de oproep. De laaste variable is de return waarde als de functie iets returned.

```
<functionCall name="functioName">
  <params>
      <var name="varName1"/>
      <var name="varName2"/>
  </params>
  <returns/>
</functionCall>
```

De DOCTYPE declaration:

```
<!ELEMENT functionCall (params, returns)>
<!ATTLIST functionCall name CDATA #REQUIRED x CDATA #IMPLIED y
    CDATA #IMPLIED>


<!ELEMENT returns (var)*>
<!ELEMENT params (var)*>
```

D.9 Block

De `block` blok stelt een groepering van code voor.

```
<block>
       code
</block>
```

De DOCTYPE declaration:

```
<!ELEMENT block
    (makeVar|setVar|move|show|hide|appear|if|if-else|sleep|
                    while|forever|emit|functionCall|print|lock|unlock|return)*>
<!ATTLIST block name CDATA #REQUIRED>
```

D.10 Class

InputEvent

De `inputEvent` blok is een binnenkomende event van een class.

```
<inputEvent type="ev1"/>
```

De DOCTYPE declaration:

```
<!ELEMENT inputEvent EMPTY>
<!ATTLIST inputEvent type CDATA #REQUIRED>
```

OutputEvent

De `outputEvent` blok is een uitgaande event van een class.

```
<outputEvent type="ev2"/>
```

De DOCTYPE declaration:

```
<!ELEMENT outputEvent EMPTY>
<!ATTLIST outputEvent type CDATA #REQUIRED>
```

Costume

De `costumes` blok is een collectie voor alle costumes.

```
<costume name="s" path="path"/>
```

De DOCTYPE declaration:

```
<!ELEMENT outputEvent EMPTY>
<!ATTLIST outputEvent name CDATA #REQUIRED path CDATA #REQUIRED>
```

Events

De `events` blok is een collectie voor alle input events van een class.

```
<events>
        <inputEvent type="ev1"/>
</events>
```

De DOCTYPE declaration:

```
<!ELEMENT events (inputEvent)*>
```

Emits

De `emits` blok is een collectie voor alle emits die een class kan doen.

```
<emits>
	<outputEvent type="ev2"/>
</emits>
```

De DOCTYPE declaration:

```
<!ELEMENT emits (outputEvent)*>
```

Handlers

De `handlers` blok is een collectie voor alle handlers van een class.

```
<handlers>
	<handler name="hand" event="ev1">
		code
	<\handler>
</handlers>
```

De DOCTYPE declaration:

```
<!ELEMENT handlers (handler)*>
```

Functions

De `functions` blok is een collectie voor alle functions van een class.

```
<functions>
	<function name="func">
		code
	<\function>
</functions>
```

De DOCTYPE declaration:

```
<!ELEMENT functions (function)*>
```

MemberVariables

De `memberVariables` blok is een collectie voor alle member variables van een class.

```
<memberVariables>
	<member type="number" name="var1" />
</memberVariables>
```

De DOCTYPE declaration:

```
<!ELEMENT memberVariables (member)*>
```

Floating blocks

De `floatingBlocks` blok is een collectie voor alle losstaande blokken.

```
<floatingBlocks>
	<blocka />
</floatingBlocks>
```

De DOCTYPE declaration:

```
<!ELEMENT floatingBlocks (block)*>
```

Costumes

De `costumes` blok is een collectie voor alle costumes.

```
<costumes>
	<costume name="s" path="path"/>
</costumes>
```

De DOCTYPE declaration:

```
<!ELEMENT costumes (costume)*>
```

Class

De `class` blok stelt volledige class voor. Deze list al zijn input events, alle verschillende emits, alle member variabelen, alle handler functions, alle floating blocks en alle costumes en alle gewone functions.

```
<class name="name">
	<events>
		<inputEvent type="ev1/">
	</events>
	<emits>
		<outputEvent type="ev2"/>
	</emits>
	<handlers>
		<handler name="hand" event="ev1">
			code
		<\handler>
	</handlers>
	<functions>
		<function name="func">
			code
		<\function>
	</functions>
	<memberVariables>
		<member type="number" name="var1" />
```

```
        </memberVariables>
        <costumes/>
        <floatingBlocks>
            <block/>
        </floatingBlocks>
</class>
```

De DOCTYPE declaration:

```
<!ELEMENT class (events, emits, handlers, functions,
    memberVariables, costumes, floatingBlocks)>
<!ATTLIST class name CDATA #REQUIRED>
```

D.11 Events en Emits

Emit blok

Het `emit` block bevat de naam event en de members van van de message van dit event. De members zijn variabelen, ze hebben een extra attribuut. Namelijk om de members te matchen.

```
<emit eventName="event">
  <var member="a" name="var1">
  <var member="b" name"var2">
</emit>
```

De DOCTYPE declaration:

```
<!ELEMENT emit (var)*>
<!ATTLIST emit eventName CDATA #REQUIRED x CDATA #IMPLIED y
    CDATA #IMPLIED>
```

Event blok

Een `event` blok voor het tonen en creeëren van een event. Deze bevat een uniek type en members van een specifiek type met een unieke naam in het event.

```
<event type="eventName">
  <member type="memberType" name"memberName"/>
  <member type="memberType2" name"memberName2"/>
</event>
```

De DOCTYPE declaration:

```
<!ELEMENT event (member)*>
<!ATTLIST event type CDATA #REQUIRED>
```

Member blok

De `member` blok kan een een event zitten. Dit is een variabele dat een type heeft en een naam.

```
<member type="memberType" name="memberName"/>
```

De DOCTYPE declaration:

```
<!ELEMENT member EMPTY>
<!ATTLIST member type CDATA #REQUIRED name CDATA #REQUIRED>
```

Access blok

De `access` blok bevat de naam van de member die men wilt aanspreken. Deze geeft deze variable zijn value terug.

```
<access name="memberName"/>
```

De DOCTYPE declaration:

```
<!ELEMENT access EMPTY>
<!ATTLIST access name CDATA #REQUIRED >
```

D.12 Instances en Wires

Instance

Dit stelt de XML voor een instance op te slaan voor. Een instance heeft een class waarbij het behoort een positie en een unieke naam.

```
<instance name="instanceName" class="className" x="X" y="Y" />
```

De DOCTYPE declaration:

```
<!ELEMENT instance EMPTY>
<!ATTLIST instance event CDATA #REQUIRED name CDATA #REQUIRED
    sprite   CDATA #REQUIRED x CDATA #REQUIRED y CDATA #REQUIRED>
```

Wire

Een wire heeft twee instances en het event dat er tussen verstuurd wordt.

```
<instance from="instanceName" to="instanceName2"
    event="eventName" />
```

De DOCTYPE declaration:

```
<!ELEMENT wire EMPTY>
<!ATTLIST wire from CDATA #REQUIRED to CDATA #REQUIRED event
    CDATA #REQUIRED >
```

WireFrame

Een wireFrame bevat instances en de wires tussen die instances.

```
\lstset{language=XML}
<wireFrame>
    <instances>
          <instance name="instance1" class="className" x="1"
              y="1"/>
          <instance name="instance2" class="className2" x="1"
              y="1"/>
      </instances>
      <wires>
          <wire from="instance1" to="intance2"
              event="eventName"/>
      </wires>
</wireFrame>
```

De DOCTYPE declaration:

```
<!ELEMENT wireFrame (instance|wire)*>
```

E Blocks

Deze bijlage bespreekt de verschillende geïmplementeerde blokken. De implementatiedetails komen kort aan bod.

De Interface Block

Een Block stelt een klasse voor die een bepaald stuk code voorstelt. Deze bevat een execute functie die een proces meekrijgt zodat hij zijn code kan uitvoeren of plaaten op het proces.

E.1 Functies en Handlers

HandlerBlock

Deze implementeert de interface Block. Deze bevat een `ArrayList<Block>` [20] die we de body noemen. De heeft een EventInstance die hij mee kreeg op oproep, via een setter. Deze wordt mee op zijn variabele stack gepushed bij het aanmaken van zijn FunctionFrame. De execute van deze block pushed de body als individuele blokken op de stack van het proces dat hij meekrijgt. Achteraan de body plakt hij nog een PopBlock.

PopBlock

Deze implementeert de interface Block. De execute van deze Block popt het bovenste FunctionFrame van de stack van het proces.

FunctionBlock

Deze implementeert de interface Block. Deze bevat een `ArrayList<Block>` [20] die we de body noemen. De excute functie van deze Block zet de body op Stack van Blocks van het proces dat deze meekrijgt. Onder deze body plakt hij nog PopBlock. De Block bevat twee `ArrayList<VariableBlock>` [20] voor respectievelijke parameters en return parameters.

FunctionCallBlock

Deze bevat een String voor de functie naam. En twee `ArrayList<String>` [20] voor respectievelijk de parameters en return waardes van de Call.
De execute van deze Block zal de variabele van de waarde parameters ophalen uit het huidige bovenste FunctieFrame van de Stack van het proces. Deze slaat hij tijdelijk lokaal op. Hierna haalt hij de namen van de parameters van de functie op. Hij maakt een nieuwe FunctieFrame hierop pushed hij alle parametersnamen met de eerder opgehaalde variabelen. Hierna pushed hij op Stack de SetBlocken voor de return waardes. Uiteindelijk roept hij de execute van de functie aan zodat deze bovenaan de stack staat. Dit telt als een primitieve stap.

ReturnBlock

Deze block bevat een `ArrayList` [20] van Strings die de namen van de variables voorstellen die gereturned moeten worden. Deze zal hij ophalen in het huidige FunctieFrame en opslaan in de ReturnVariables van het proces. Een ReturnBlock popt alle blokken van de Stack tot hij een PopBock tegenkomt.

PrintBlock

Deze bevat een Block die geexecute wordt en de waarde van de variable wordt uitgeprint.

E.2 Events en emits

AccessBlock

Deze bevat twee Strings namelijk de naam van het EventInstance waarvan het een member wil opvragen. En de naam van die member. De execute functie zal dus het EventInstance van de FunctionFrame opvragen. Hierin vraagt hij de variable op van de member en deze geeft hij terug.

EmitBlock

Bevat een naam van het Event. En een Hashmap [13] van Strings die de members van het event voorstellen deze worden gemapt op Blocks. De execute van dit Block zal deze Blocks uitvoeren en de bekomen variabele kopieren en mappen op de juiste string. Het zal dan een aangemaakte event terug geven aan het proces. Dat dit op zijn beurt doorgeeft aan de VM.

E.3 String blocks

ConcatBlock

De execute van deze block geeft een variable terug die de string concatinatie van de een left- en rightBlock is. Deze Blocken kunnen dieper genest zijn maar hun execute geeft een variable terug.

StrlenBlock

Deze block geeft een variable terug die de lengte van de String bevat. Het bevat een Block die op zijn execute een Stringvariable terug geeft.

CharAtBlock

Deze block geeft een variable terug waarvan de inhoud het character op een gegeven index van een gegeven string is. De String wordt meegeven als een block en deze geeft dus een variabele terug. De block die de string bevat kan dus genest

zijn. Als die index niet gevonden is, dan wordt er een OutOfBoundsException gegooid.

E.4 Operation blocks

ArithBlock

Deze block bevat een left block en een right block. Als deze twee worden geexecute op de teruggeven variable wordt de juiste operatie uitgevoerd de bekomen variabele wordt terug gegeven. De block bevat ook een statische hashmap zoals beschreven in sectie 9.9 om de juiste operatie uit te kunnen voeren. De uitvoering zal dus in een primitieve stap gebeuren.

LogicBlock

De execute van deze block geeft een variable terug. Deze block bevat een left block en een right block. Als deze twee worden geexecute op de teruggeven variable wordt de juiste operatie uitgevoerd de bekomen variabele wordt terug gegeven. De uitvoering zal dus in een primitieve stap gebeuren. De block bevat ook een statische hashmap zoals beschreven in sectie 9.9 om de juiste operatie uit te kunnen voeren.

Random

De execute van deze block geeft een random value tussen een lower- en upperBound terug in een Variabele. De lower- en upperBound kunnen weer blokken zijn die worden execute en een variabele teruggeven.

E.5 Locks

LockBlock

Deze block lockt een bepaalde membervariabele van de instance waarbij het huidige proces hoort.

UnlockBlock

Deze block unlockt een bepaalde membervariabele van de instance waarbij het huidige proces hoort.

E.6 Conditionele blocks

IfBlock

Deze block bevat een Block conditie en een `ArrayList<Block>` [20] die de body van de If voorstelt. De execute van de block kijkt als de conditie naar true evalueert door deze te laten execute en de waarde van de variable na te kijken.

Zo ja dan wordt de body van de If op de stack van het proces gepushed anders zal de block aflopen..

WhileBlock

Deze bevat een Block conditie en een `ArrayList<Block>` [20] die we body noemen. De execute van de block kijkt als de conditie naar true evalueert door deze te laten execute en de waarde van de variable na te kijken. Zo ja dan wordt de body plus zichzelf op de stack van het proces gepushed.

IfElseBlock

Deze block bevat een Block conditie en twee `ArrayList<Block>` [20] die respecitievelijk de body van de If en else voorstellen. De execute van de block kijkt als de conditie naar true evalueert door deze te laten execute en de waarde van de variable na te kijken. Zo ja dan wordt de body van de If op de stack van het proces gepushed anders de body van de Else.

E.7 Physics

ShowBlock

Voert een functie van de instance uit om te tonen of te hiden.

ChangeAppereanceBlock

De bevat een Block index, deze kan een arith expression zijn. Dus we laten hem execute zodat we de index krijgen in een variable. Dit zal de execute van de blok doen plus het oproepen van de functie bij een instance die de appereance veranderd.

MoveBlock

Deze bevat een x- en een y-Block. De execute van de MoveBlock zal de waarde van x en y bekomen door deze Blocks te execute. De waardes geeft hij mee aan een functie van de instance die zijn x en y verhoogt met die waardes.

E.8 Variables

VariableBlock

Deze Block bevat een String en een Type. De execute van deze block zal deze variable aanmaken en op het huidige FunctionFrame zetten.

E.9 ValueBlock

ValueBlock maakt een nieuwe literal variabele aan en geeft deze terug.

GetVarBlock

De GetVarBlock kent de naam van een variabele en vraagt deze variabele op aan het proces. Deze variabele kan lokaal zijn of een member variabele. De membervariabele kan geshadowed worden door een lokale variabele.

SetBlock

Deze bevat een String die de naam is van een variabele op de huidige FunctionFrame. Deze block bevat nog een andere block, dat ofwel een variable, value of operatie aanduid. De execute van eender van deze blocken geeft steeds de inhoud (variable) terug aan de SetBlock. Deze blok telt als een primitieve stap.

SetReturnBlock

Deze blok zal de return waardes van een functie in het functieframe zetten waarin de functie aanroep gebeurt is.

E.10 Debug

DebugBlock

De DebugBlock kent het model waarvan hij gecompileerd is. Indien er gedebugged moet worden zal de blok het model laten weten dat de debug modus aan of uit moet. De blok kan ook een breakException gooien die de uitvoer stil legt na het aflopen van een cycle.

E.11 Other

SleepBlock

De sleep block kent het aantal cycles die geslaapt moeten worden. Deze block zal zichzelf plaatsen op de uit te voeren stack van blocks. Deze nieuwe block zal een cycle minder lang moeten slapen. Uiteindelijk als de counter nul bereikt, wordt de sleep block verwijderd.