Analyseverslag PSOPV

Visuele Programmeer IDE

Auteur:
Matthijs Kaminski

Auteur:
Axel Faes

Begeleider:
Jonny Daenen

Opdrachtgever:
Raf Van Ham

1 maart 2015

Inhoudsopgave

1 Beschrijving van het project

1.1 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. Het veroorzaken van specifieke events en het opvangen hiervan kan visueel gevolgd worden in de debug modus. Het doorgeven van Events tussen Instanties kan via wires in het Wired-view. Een visueel canvas kan gebruikt worden om instanties en veranderingen ervan te tonen. Ook kunnen hierin input Events worden gegenereerd op instanties.

1.2 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 zien welke grote blokken en welke verbindingen actief zijn.

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 zal een console geïmplementeerd worden. 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.

In dit verslag beschrijven we onze analyse. Eerst wordt bestaande software geanalyseerd (Sectie 2), daarna wordt een beschrijving gegeven van de Visuele IDE (Sectie 3). Er wordt getoond wat de evaluatie criteria voor de IDE zijn. In Sectie 6 worden de gebruikte algoritmes uitgelegd. Hierna volgen implementatie details met betrekking tot de code structuur en de structuur van datafiles (XML). Uiteindelijk geven we mockups die een algemeen beeld tonen van de IDE, alsook een taakverdeling en planning die opgesteld is. In de bijlagen vind u de geïmplementeerde kleine blokken.

1.3 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 Bestaande Software

Hierin staan enkele programma's beschreven die gelijkaardig zijn aan onze IDE. Er wordt uitgelegd welke elementen we overgenomen hebben en welke elementen niet overgenomen zijn.

2.1 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.

Visuel voorstellen van een Sprite.

Een sprite komt in onze applicatie overeen met een instantie van een Klasse. In onze applicatie zal een Klasse ook een visuele voorstelling hebben en kan deze ook meerdere uiterlijken hebben. Het aanpassen van deze zal echter beperkt blijven tot het scaleren van een visuele afbeelding. Ook kan een Sprite tekstballonen tonen in het canvas, dit is niet de prioriteit in onze applicatie. 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 nemen we enkel de mogelijkheid om de x- en y-positie en eventueel rotatie van een instantie van een Klasse te veranderen. Uit looks nemen we enkel de mogelijk over om het uiterlijk te veranderen in een eerder ingevoegde appearance. Uit de sound en pen blokken nemen we niets over.

Er zal de mogelijkheid zijn om een variable te creeëren in een Klasse, dit wordt gezien als een private member variabele van die Klasse. In tegenstelling tot Scratch gaan we eerst enkel een variabele implementeren. Een lijst beschouwen we als een extra.

Uit events nemen we het broadcastblok over. In onze applicatie heeft het echter te 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.

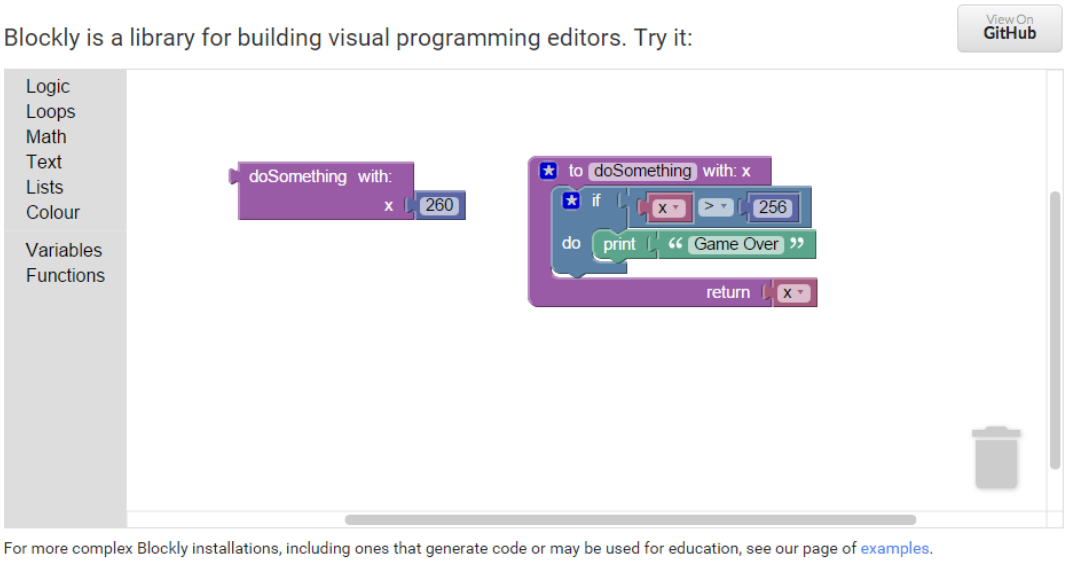
Uit de control blokken: we inplementer de standaard controle structuren zoals een while, if-else, repeat. De clone blokken zien we als extra indien er nog tijd over is.

Sensing blokken zoals het checken op collesion zien we als extra indien er tijd over is.

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

Uit de categorie More blocks van 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 geven we de blokken een professionelere look dan de blokken van Scratch. Dit gebeurt door gebruik te maken van strakkere lijnen en neutrale kleuren.



Figuur 1: Blockly.

2.2 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.

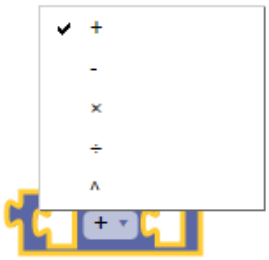
De blokken zijn gemodelleerd naar puzzelstukjes. Dit maakt het gemakkelijk om te zien hoe de blokken in elkaar gestoken kunnen worden. Blockly is gericht op nieuwe programmeurs. Het versimpelt verschillende programmeerconcepten. Er zijn enkel globale variabelen en lijsten zijn niet nul-based maar één-based.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

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 opera-



Figuur 2: Blockly operators.

toren, alsook één operator blok voor de binaire logische vergelijkings operatoren.

2.3 Unreal Engine 4: Blueprints

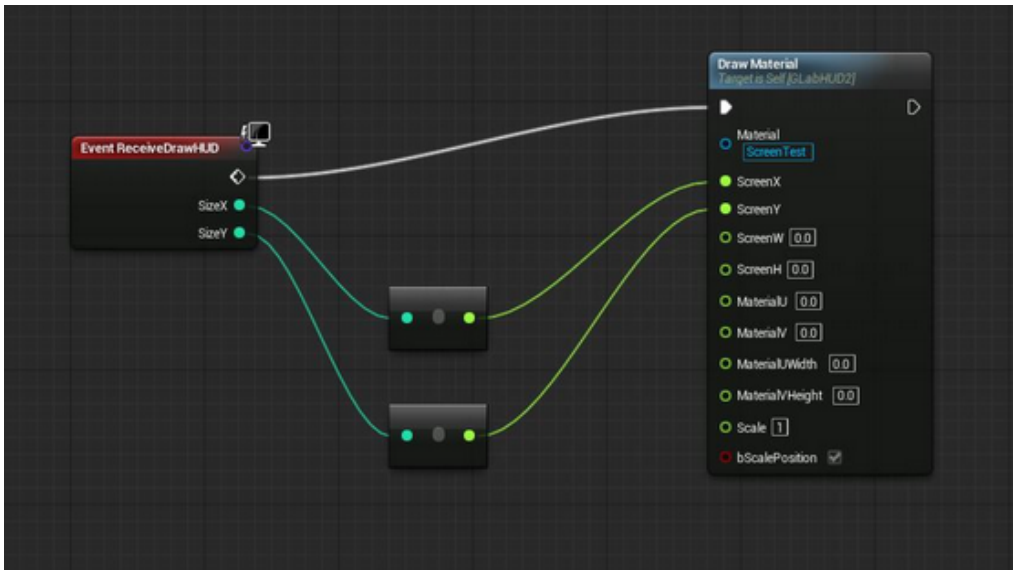
De Unreal Engine 4 [3] is een game engine die uitgebracht is door Epic Games. In de Unreal Engine 4 is een visueel scripting systeem ingebouwd. Dit wordt Blueprints genoemd. Het laat toe om volledige gameplay elementen visueel te scripten via een node-based interface. Het is mogelijk om een volledige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



Figuur 3: Unreal Engine 4.

3 Diepgaande Beschrijving van het project

3.1 Diepgaande uitleg

In deze sectie zal onze interpretatie van de opgave worden uitgelegd. Deze interpretatie is voorgelegd aan de opdrachtgever en is goedgekeurd. De volledige visuele voorstelling van de applicatie wordt getoond in de Sectie 10 Mockups van dit analyse verslag.

Zoals al eerder vermeldt zal een programma zijn flow kunnen opbouwen in het deel van de IDE dat we het Wired-view noemen. Hierin zullen 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 onderlingen 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 er een EventInstance van aan te maken. 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 Wired-view van de IDE.

Er is een aparte view waarin alle Instanties van Klassen als blokjes getoond worden, dit view noemen we het **Wired-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.

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. Voor de debug-modus zou dit view statisch zijn. Terwijl een extra view het eventueel 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. Het doorgeven van Events door/aan specifieke instanties werd beschreven in 3.1.

Er zijn **standaard Events** beschikbaar zoals oa. onKeyPress, onClicked, onStart, 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 wired-view. Een Klasse bestaat uit: input

Events, Handlers voor die Events, functie definities en member variabelen. Een Klasse kan worden voorgesteld in het Wired-view. Deze appearance kan door een functie in de Klasse worden veranderd. Een Instantie kan dan in het Wired-view beslissen van welke andere Instanties het die input Events ontvangt of naar welke instaties hij Events verstuurd.

Een Klasse kan op **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.

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

Een Klasse zal ook een overzicht hebben met alle Events die erdoor worden geemit.

Een uitbreiding van visuele omgeving door toe te laten om **functie aanroepen** te maken binnen een Klasse. Eerst 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 zal worden voorgesteld door een pijl naar een ander, kleine blokje. Dit blokje bevat de naam van de functie die zal worden opgeroepen. Alsook zijn input parameters. 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 positie van de Instantie van de Klasse 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 A.

3.2 Prioritair functies

Als hoogste prioriteit hebben we een stripped down versie van de 'programmeertaal' gekozen. Dit zodanig dat we een simpele versie hebben om programma's te kunnen testen. Deze stripped down versie zal de meeste features van de programmeertaal implementeren. Niet alle blocks zullen dan geïmplementeerd worden, maar de VM, proces etc zullen afgemaakt zijn.

Hierna bouwen we hierop de IDE en de module voor het opslaan en inladen van programma's. Er zal dan een ruw prototype aanwezig zijn van de Visuele IDE. De programmeertaal zal dan verder afgemaakt worden, alsook de professionele look van de GUI. Als er hierna nog tijd is, kunnen extra's geïmplementeerd worden.

3.3 Extra's

Er zijn enkele features die we als extra gelaten hebben.

Data types

Een eerste extra zijn lists of meer in het algemeen, extra primitieve data types. Momenteel zijn er maar enkele primitieve types gepland. Namelijk: numbers, string, booleans, events en literal values. Er is geen mogelijkheid om lijsten te kunnen aanmaken, alsook geen mogelijkheid voor integers, etc. Extra types zijn altijd welkom in een programmeertaal. In het design dat we gemaakt hebben kan op simpele wijze nieuwe types aangemaakt worden.

Static functions

Static functions zijn ook gepland als extra. Dit zijn functions die opgeroepen kunnen worden door alle Klassen en dus niet behoren tot een bepaalde Klasse. Een voorbeeld gebruik zou kunnen zijn: stel dat de gebruiker een derde-machts wortel functie wilt maken. Deze functie moet momenteel behoren tot een bepaalde Klasse. Dus deze functie moet ofwel opgeroepen worden via een soort event, of via code-duplicatie de functie in elke Klasse implementeren. Een static functie zou dit probleem oplossen. Dit kan ook simpel toegevoegd worden via ons design.

Multiple returns

De mogelijkheid voor meerdere return values bij een functie is een extra. Onze programmeertaal bíedt de mogelijkheid tot meerdere return waardes, echter zal in de GUI dit niet geïmplementeerd worden. Dit implementeren in de GUI zien we als extra.

Instanties op runtime aanmaken

Momenteel kunnen instanties enkel op creation time aangemaakt worden. Tijdens het runnen van een programma kunnen er niet dynamisch extra instanties aangemaakt worden. Dit is toch een interessante extra, aangezien dit een heel krachtig mechanisme is. In ons huidig design kan er gemakkelijk een clone functie (om instanties om runtime aan te maken) toegevoegd worden.

Collision detection

Functions om collision detection te testen tussen verschillende instanties zien hebben we als extra genomen.

4 Evaluatiecriteria

Er zijn verschillende criteria die we stellen aan onze software. Sommige van deze criteria is subjectief en niet getoond in onderstaande bulletpoints. Onder de subjectieve criteria verstaan we oa. de professionele look van de IDE.

- Multilanguage user interface (ondersteunde talen zijn engels en nederlands).
- Geen crash bij inladen van foute XML data.
- Verkeerde invoer onmogelijk maken bij het wireFrame (enkel events van hetzelfde type kunnen verbonden worden, en input kan enkel met output punten verbonden worden)
- Het verzenden van events wordt gelijktijdig opgevangen door de geabonneerde instanties. De uitvoering van de code gebeurt hierna ook op een concurrent manier.
- Mogelijkheid om projecten op te slaan.
- IDE blijft niet vasthangen bij een infinite loop.
- IDE geeft geen beperking op het aantal processes die tegelijkertijd kunnen runnen.
- IDE geeft geen beperking op het aantal Klassen die tegelijkertijd kunnen bestaan.
- IDE geeft geen beperking op het aantal Events die tegelijkertijd kunnen bestaan.
- IDE geeft geen beperking op het aantal functions die tegelijkertijd kunnen bestaan.
- Recursieve aanroepen zijn mogelijk.

- Parameter passing is geïmplementeerd.
- Bij type-error op runtime breekt enkel dat proces af.
- Bij variable-not-found error op runtime breekt enkel dat proces af.
- Deadlocks zijn niet mogelijk d.m.v. de locks die de gebruiker kan gebruiker.
- Er is geen probleem met racing conditions m.b.t. processen die dezelfde variabelen willen gebruiken indien de gebruiker locks gebruiker.
- Processen worden in dezelfde orde toegevoegd als ze worden toegevoegd.
- Geen globale variabelen.
- Een lock kan niet gezet worden indien een ander proces al een lock gezet heeft op dezelfde instantie.

5 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. Java biedt ook een sterke GUI libary aan nl. Swing.

6 Algoritme en datastructuren

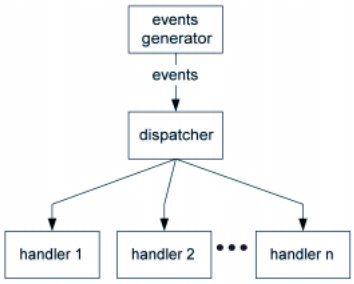
Zoals al eerder vermeld in dit verslag is dient onze applicatie om event-driven programma's te maken. In de volgende sectie halen we kort aan wat het event-driven programming paradigma inhoudt en hoe we dit gaan implementeren. Verder halen we ook aan hoe we onze uitvoer gaan controlleren. Hier verkiezen we concurrent programming. We leggen onze keuze uit met voor en nadelen.

6.1 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 [4].

Extended handlers design pattern

Als design pattern voor onze applicatie basseren we ons op het extended handlers design pattern dat Stephen Ferg uitlegt in zijn paper: Event-Driven Programming: Introduction, Tutorial, History [5].



Figuur 4: Extended handler.

In Figuur 4 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.

6.2 Concurrent computing

Concurrent computing [6] 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 [6] 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 [7]. Om dit probleem op te lossen stellen we dat slechts één proces gelijktijdig locks kan aanbrengen.

6.3 Drawing

Blokken worden getekend door middel van rechthoeken. Deze kunnen genest worden in elkaar.

Het WireFrame is de collectie van Instances en Wires. Een Instance wordt getekend als een blok waarop enkele punten getekend zijn die de inkomende en uitgaande Events voorstellen. De Wires worden getekend als lijnen. Deze lijn kan bestaan uit meerdere punten en wordt getekend door de gebruiker. Deze lijn zal dus niet automatisch gegenereerd worden.

6.4 Model en Controller

Elke visuele component is verbonden met een model (zie Sectie 8.6). Dit model bevat info die nodig is om aan type checking te doen in de GUI. Ook worden de blokken die genest zitten in deze blok bijgehouden in het model. De wireFrame zal ook voorgesteld worden als een Model. De controller die behoort tot een model zal nagaan of een blok genest kan worden in een bestaande blok door informatie op te vragen aan het model van de bestaande block. Er wordt ook doorgegeven wanneer er iets genest wordt.

6.5 Runtime

Op het hoogste niveau is een Runtime (zie Sectie 8.7) aanwezig. De IDE gebruikt een Abstracte klasse Runtime zodanig dat de gebruikte programmeertaal niet direct vasthangt aan de IDE. Er is een klasse aanwezig die de Runtime voor de geïmplementeerde programmeertaal implementeerd. De abstracte Runtime bevat alle modellen van de blocks alsook het wireFrame model, de geïmplementeerde Runtime bevat de nodige data voor de executie van de code zoals een Compiler en een Virtual Machine (zie Sectie 6.7). De Runtime zorgt voor de vlotte uitvoering van alle code. 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 6.7.

6.6 Compileren

Elke visuele view van een blok heeft een model zoals uitgelegd. Deze blok 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 niet direct vasthangt aan de IDE. Hij heeft verschillende functies die elk een ander type BlockModel (8.6) compileren. Omdat het wireFrame ook voorgesteld wordt als een model kan het wireFrame simpel gecompileerd worden omwille van deze functies. Elk model weet welke blokken genest zijn zullen alle geneste blokken ook worden gecompileerd. Het design van de Compiler is het visitor design pattern. Het algoritme voor het compilen van een blok wordt gescheiden van de datastructuur van de blok.

6.7 Virtual Machine

Onze virtual machine bevat een lijst van alle processen die momenteel moet uitgevoerd worden. Incrementeel zal er telkens één primitieve stap uitgevoerd worden van elk proces. Als er een proces uitgevoerd moet worden, zal dit toegevoegd worden aan het einde van de lijst. Als een bepaald proces klaar is, wordt dit verwijderd uit de lijst.

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.

Een proces

Een proces (zie Sectie 8.5) 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.

Stoppen van uitvoering

Voor het stoppen van de uitvoer zal er een event worden verzonden naar de Runtime bij het opvangen van dit event zal de uitvoering worden gestopt.

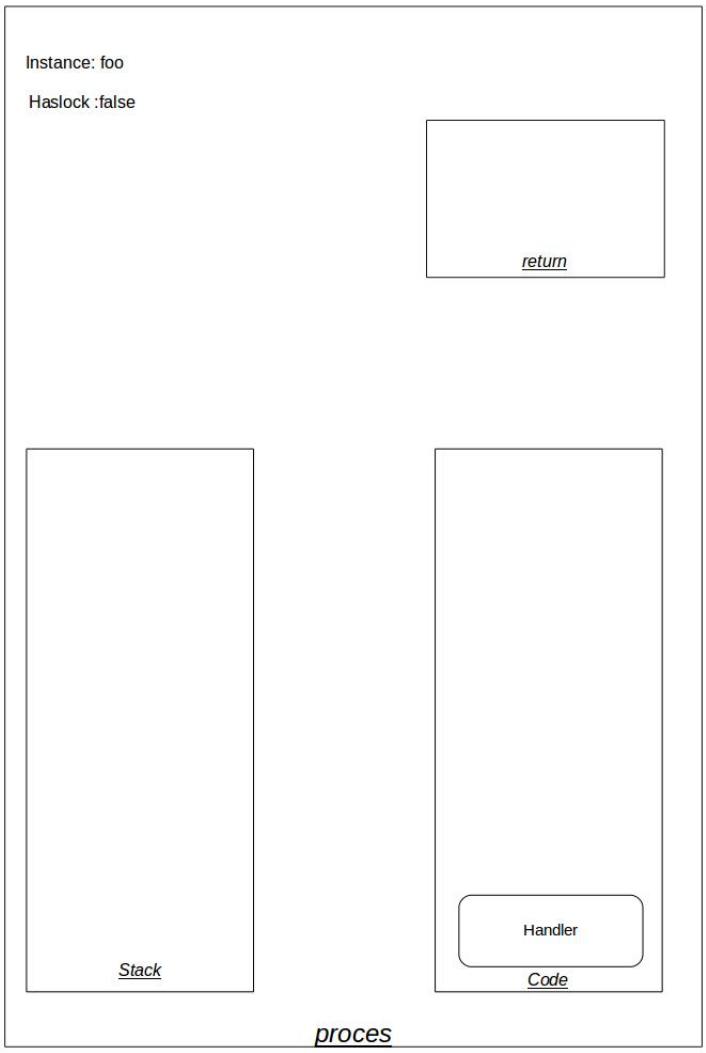
6.8 Voorbeeld implementatie

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:

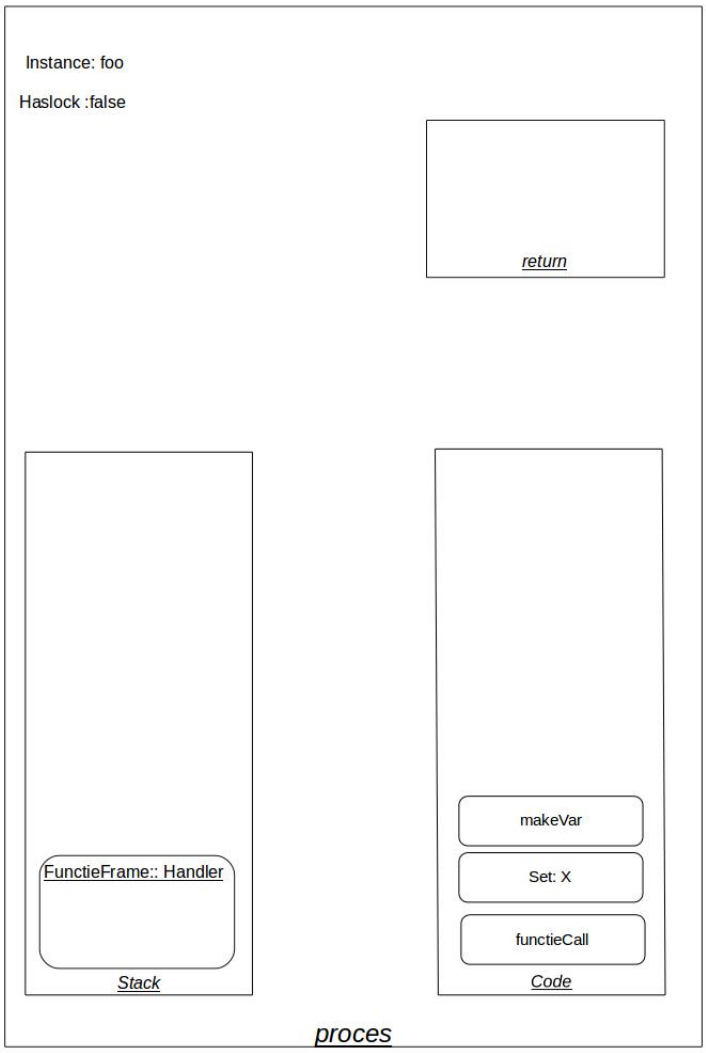
```
Event event1{
	members:
		- member1(number, value)
}
class Class1 {

	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:

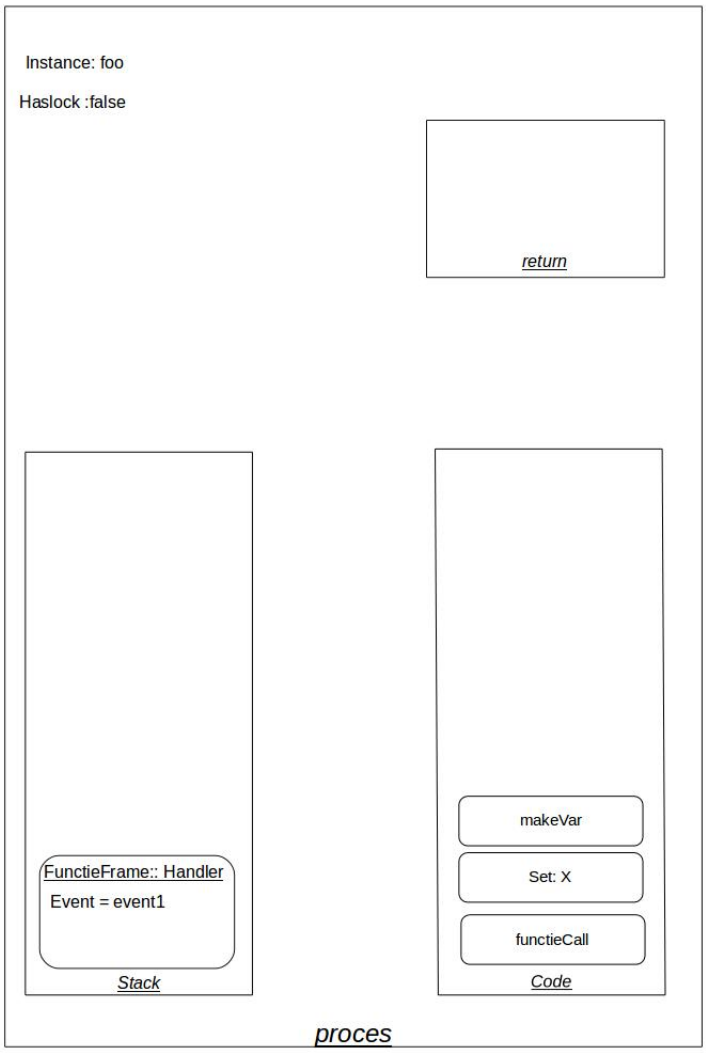
Stap 1: De handler zal zijn execute functie uivoeren. Deze zal een nieuw FunctieFrame aanmaken. Ook zal de eventInstance die mee werd gegeven bij het aanmaken van het proces op de stackframe geduwd. De blok van de handler op de Stack wordt vervangen door de inhoud.
Stap 2: De execute van makeVar zal 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 cree¨ert 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.



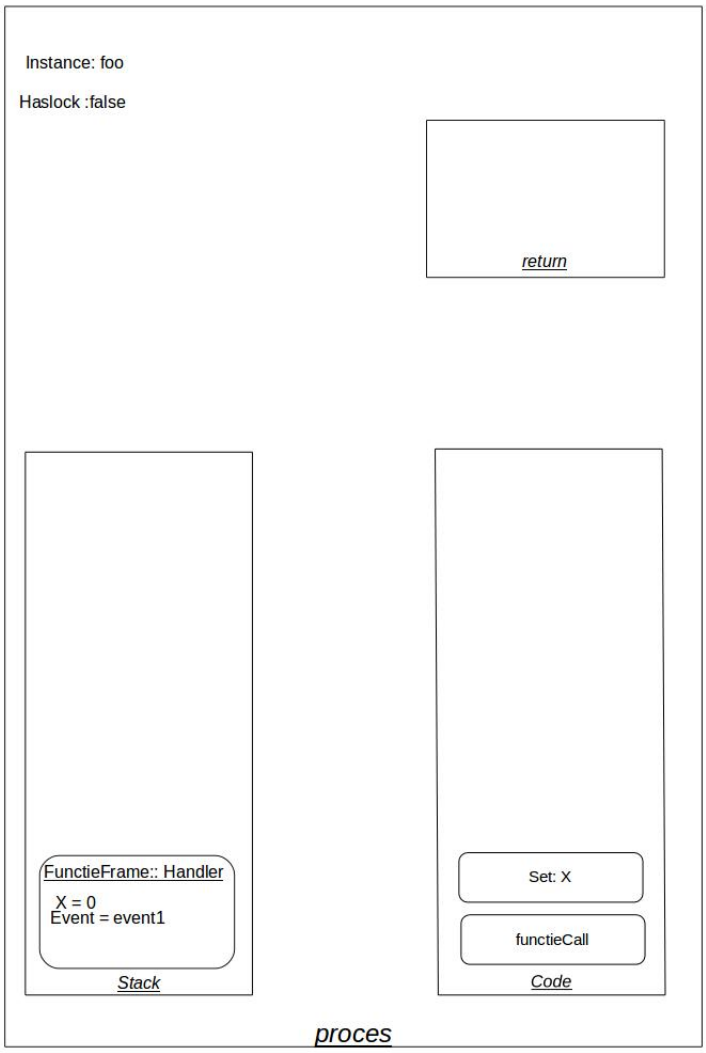
Figuur 5: Stap 1



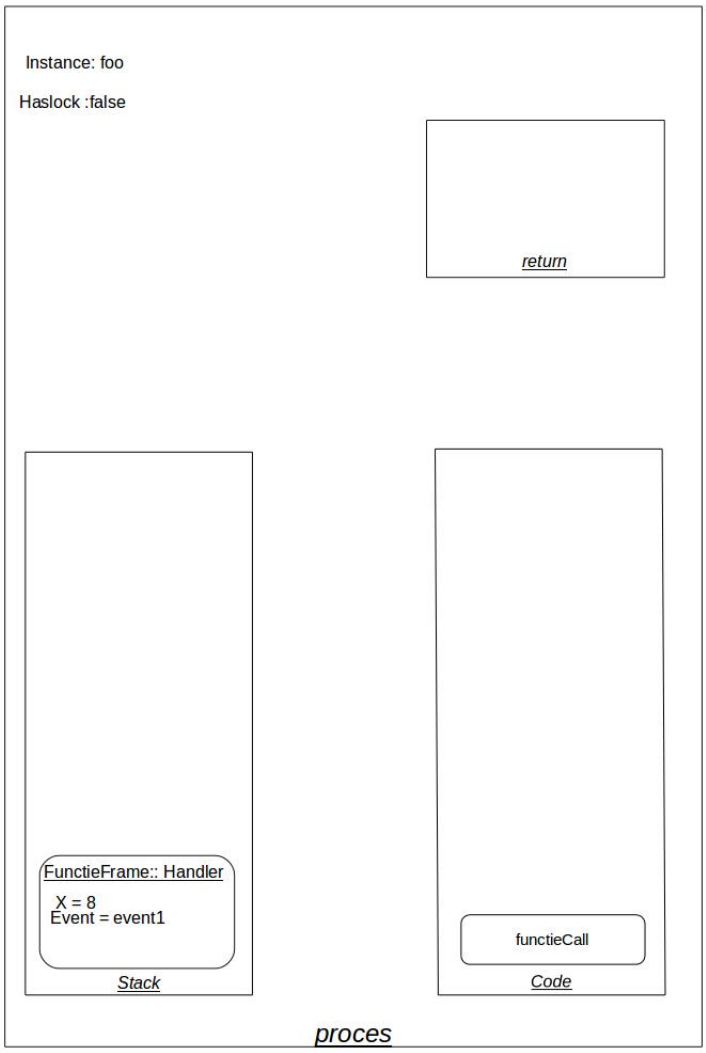
Figuur 6: Stap 2



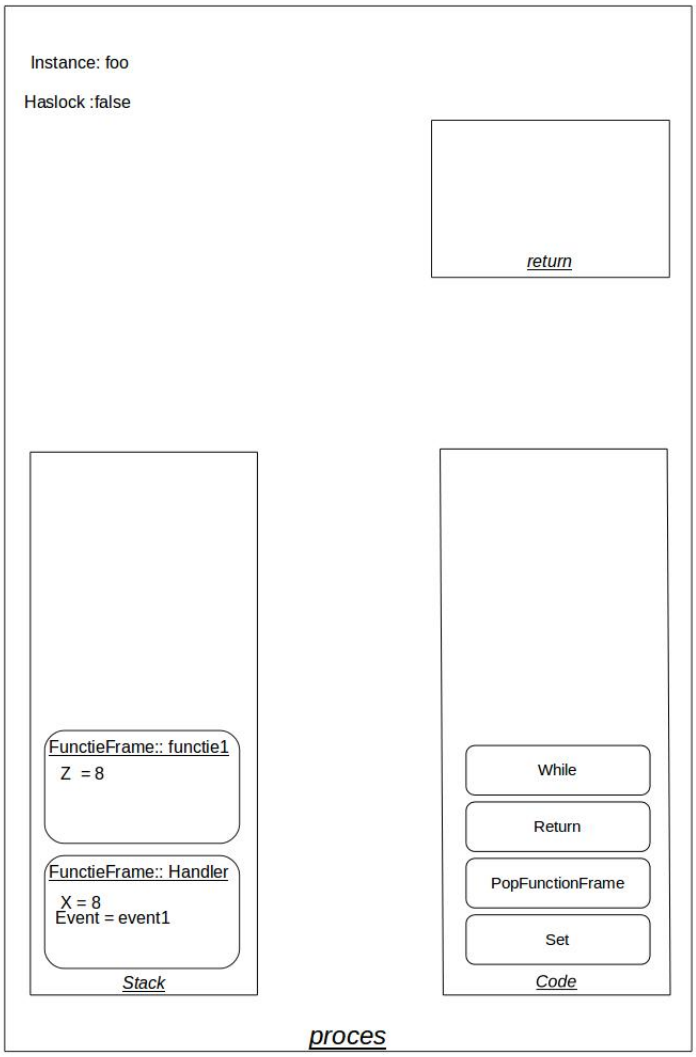
Figuur 7: Stap 3



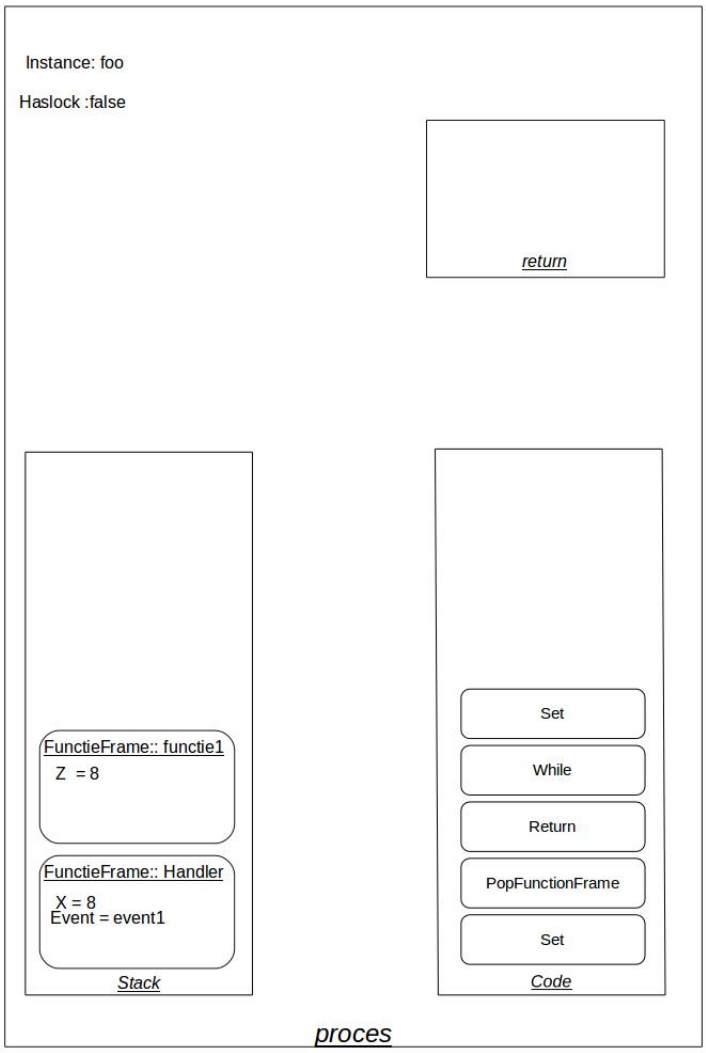
Figuur 8: Stap 4



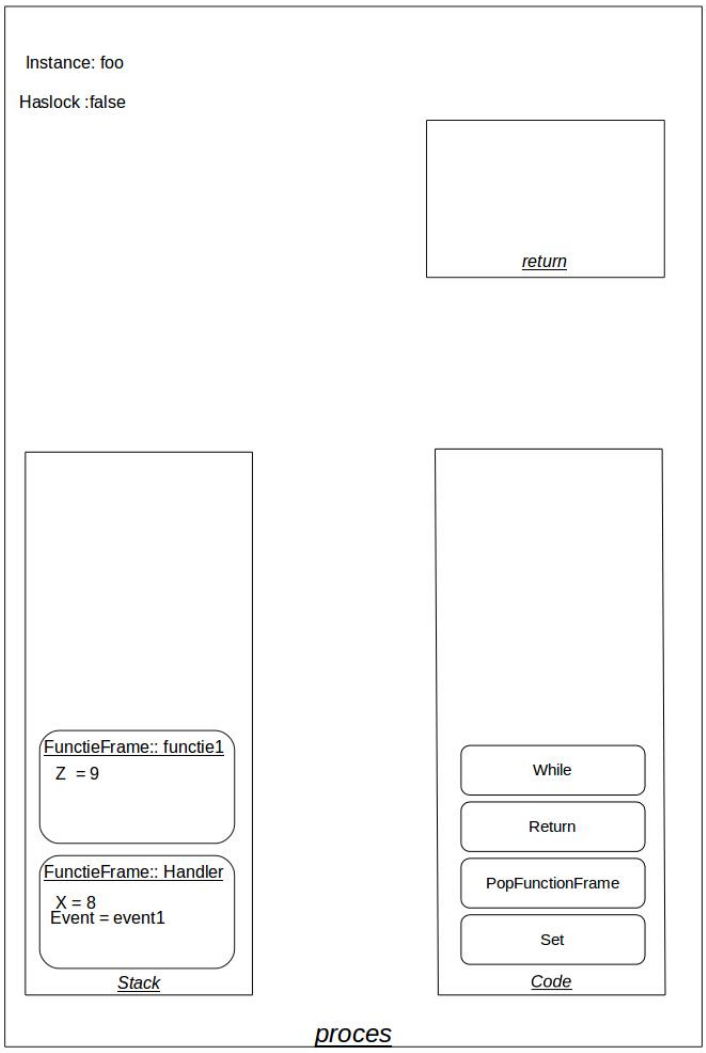
Figuur 9: Stap 5



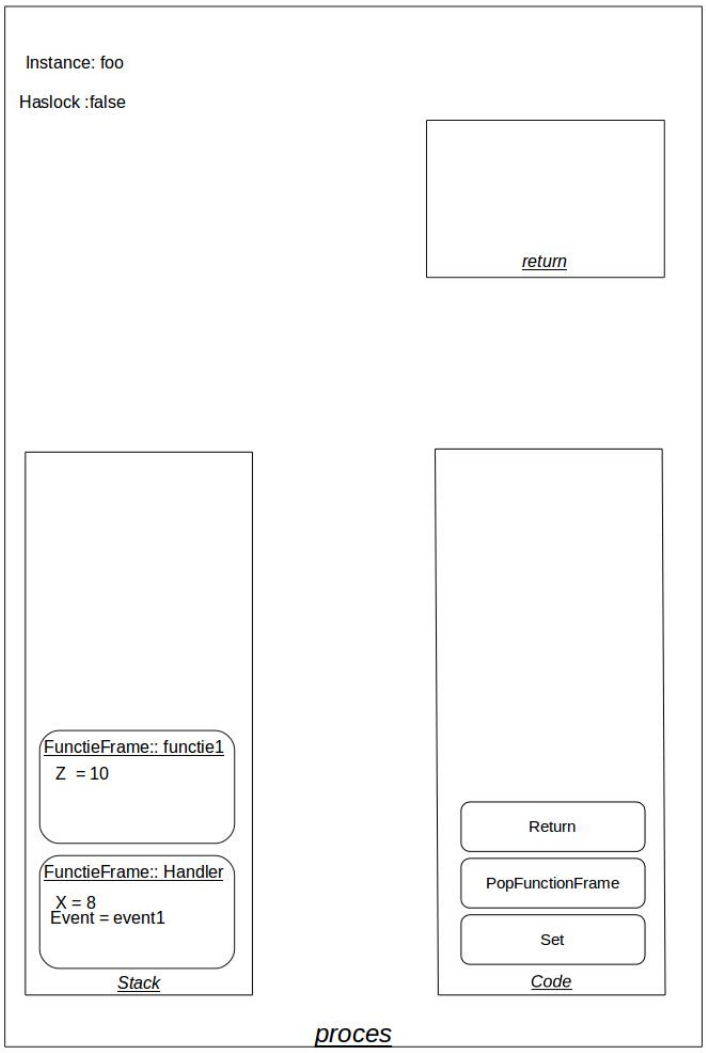
Figuur 10: Stap 6



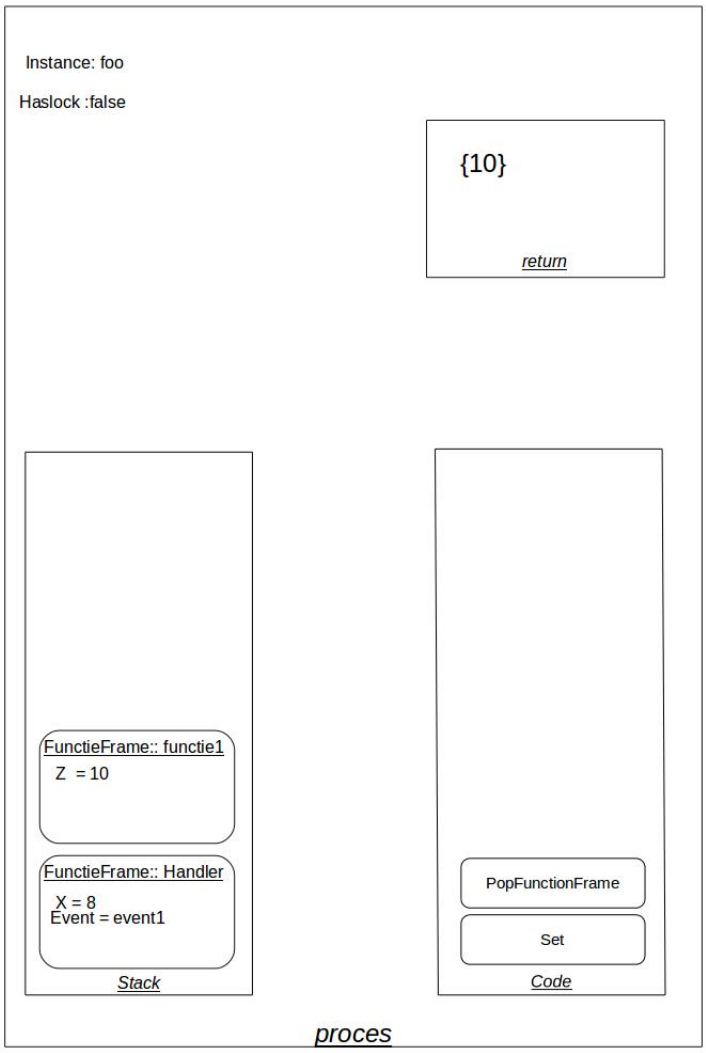
Figuur 11: Stap 9



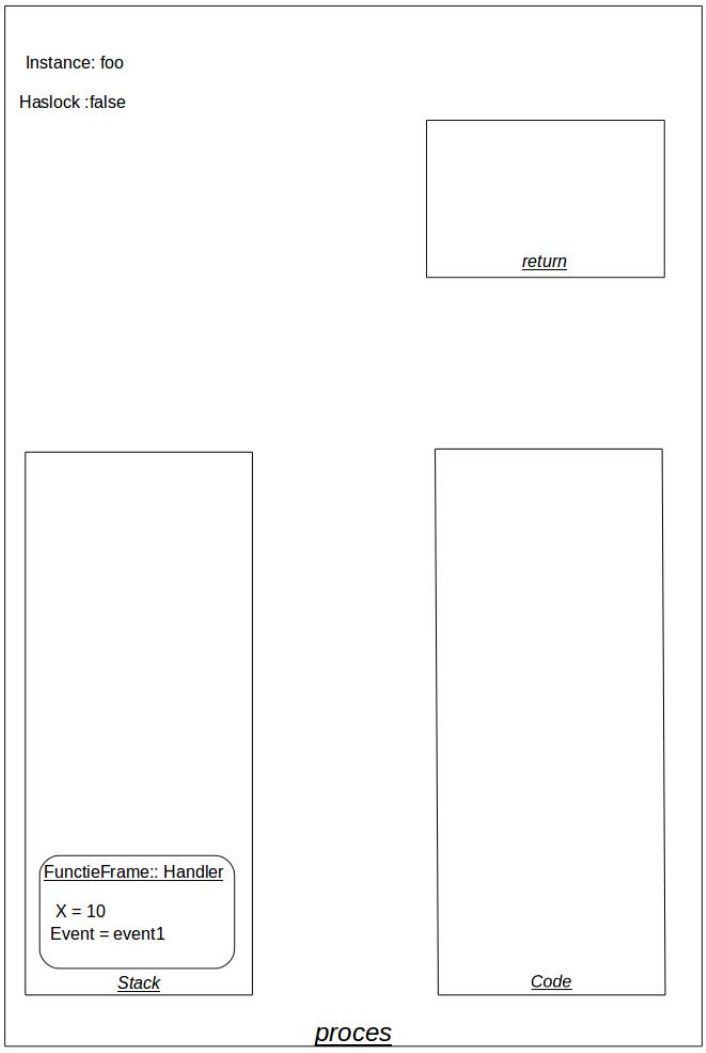
Figuur 12: Stap 10



Figuur 13: Stap 11



Figuur 14: Stap 12



Figuur 15: Stap 13

6.9 Lambda expressions

Sinds 1.8 biedt java de mogelijk aan om lambda expressions [8] te gebruiken, we gaan hier dan ook gebruik van maken. Onze operator blok is hier geschikt voor. Via lambda expressions kan men anonieme interface methodes aanma-

ken. 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);
```

7 Code structuur

Alle code zal opgedeeld worden in verschillende modules. Deze modules definiëren hoe er met andere modules gecommuniceerd word. Hierdoor word het gemakkelijk om modules intern van werking te veranderen zonder dat er iets moet veranderen aan andere modules. Een module is een cluster van klassen die instaan voor elk instaan voor een doel dat beschouwd wordt door de module.

7.1 Exceptions Module

Deze module heeft enkel als verantwoordelijkheid om alle excepties te groeperen.

7.2 Variables Module

Deze module bevat alle klassen die te maken hebben met de data types die gebruikt worden.

7.3 Blocks Module

Deze module implementeerd alle executie bloks die aanwezig zijn in de IDE.

7.4 Collections Module

Deze module bevat klasses die higher level zijn dan blocks. Het zijn collecties van Events, Classes, Instances.

7.5 Core module

Deze module staat in voor het uitvoeren van de code. De output van de compile module 7.7 zal door de VM module uitgevoert worden. Deze module kan werken in een release mode waarbij code normaal uitgevoert word, of in een debug modus waarbij er stap voor stap doorgeen het programma gelopen kan worden.

7.6 Model module

Deze module is de representatie van alle code die gemaakt kan worden in de visuele IDE. Deze module zal ook instaan voor het nakijken van de code op syntax fouten (zie Sectie 8.6). Deze module heeft verschillende klassen om events, Klassen, instances, variabelen en primitieve code blokken voor te stellen.

7.7 Runtime module

Deze module zal instaan voor het compilen en runnen van de code. De module zal een bepaalde hoeveelheid ruwe code van Klassen, Instances en Events omvormen naar uitvoerbare code. Deze module bevat ook een klasse die de runtime regelt van de programmeertaal van de IDE.

7.8 File module

Deze module behandeld het laden en saven van de IDE data. Deze data omvat de XML data. Er wordt ook gezorgd voor het multilanguage maken van de applicatie (zie Sectie 9.1).

7.9 GUI

De GUI wordt momenteel bekeken als één grote module maar zal nog opgedeeld worden in kleinere modules zoals een menu module, etc.

8 Klassen per module

Hierin worden alle grote klassen beschreven per module die gebruikt worden in de IDE. Elke klasse wordt kort toegelicht over hun functie, welke ADT's gebruikt hebben alsook hoe alle klassen samenwerken.

8.1 Exceptions

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.1.

TypeException

Deze wordt opgeroepen als er op runtime een type error gebeurt. De uitvoer van het uitvoerend proces wordt dan gestopt. De andere blijven doorgaan.

ProcesFinishedException

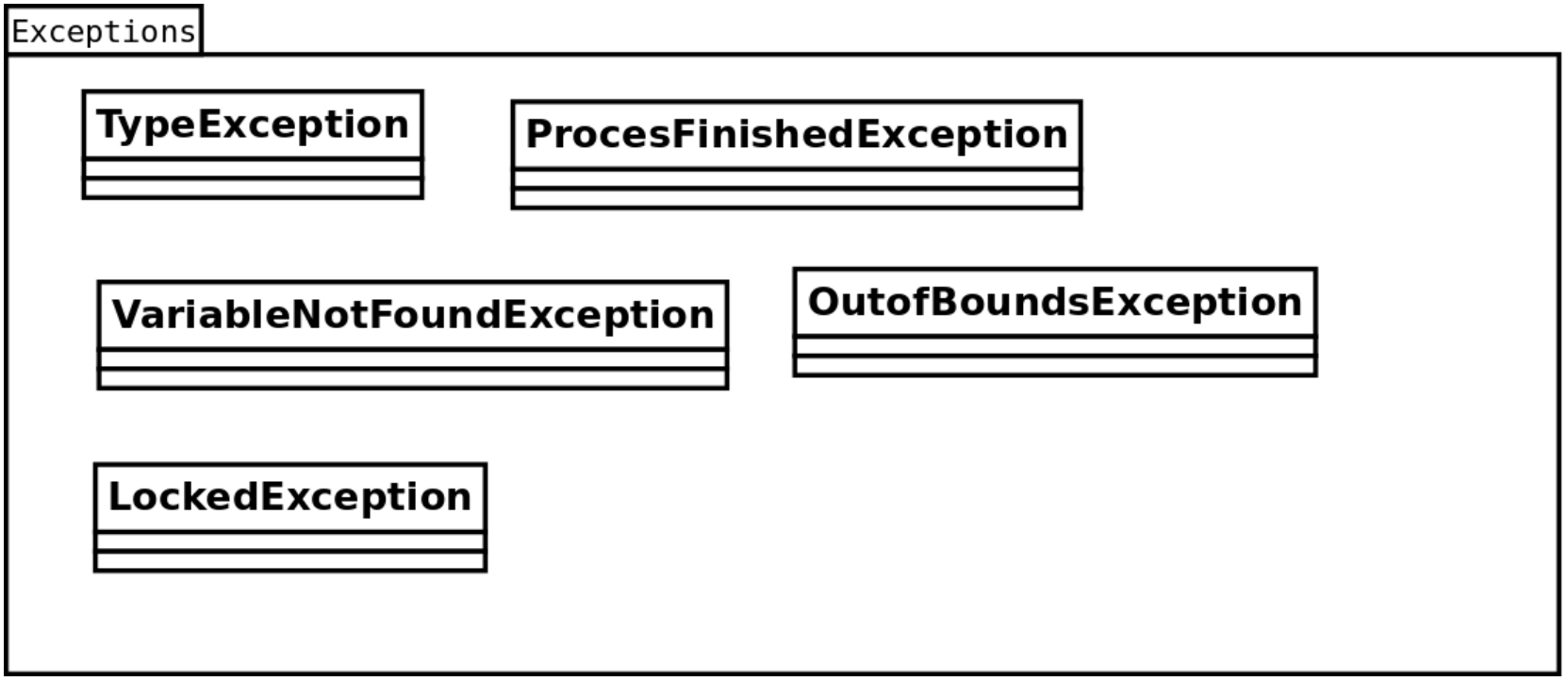
Deze exception wordt opgeroepen als een proces geen blocks meer bevat om uit te voeren.

LockedException

Deze Exception wordt opgeroepen als er een member variable van een Instance wordt aangesproken die gelocked is.

OutOfBoundsException

Deze exception wordt opgeroepen in CharAtBlock als in foute index wordt meegegeven.



Figuur 16: Exceptions Module.

8.2 Variables

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.2.

EventInstance

Een event bevat een Event voor zijn type Event waarvan het een instance is bij te houden en een Hashmap [9] waarbij het de naam van een variable (String) mapt op een variabele.

ReturnVariables

De ReturnVariables is klasse die gebruikt om return values in op te slaan en terug aan de functioncall. Deze bevat een stack [10] waarop de return waarde worden geduwt. Op deze manier kunnen we meerdere return values mogelijk maken. Echter voorzien we eerst maar een return value.

Type

Dit is Enumeratie die een van de volgende kan zijn: Boolean, String, Number, Event of Value. Een value is een letterlijke inputwaarde door de gebruiker. Deze is nog niet geconverteerd naar een feitelijk type en kan dus eenderwelk type voorstellen.

Abstact Class Variable

Deze klasse stelt een variabele voor. Deze heeft een Type als member.

BooleanVariable

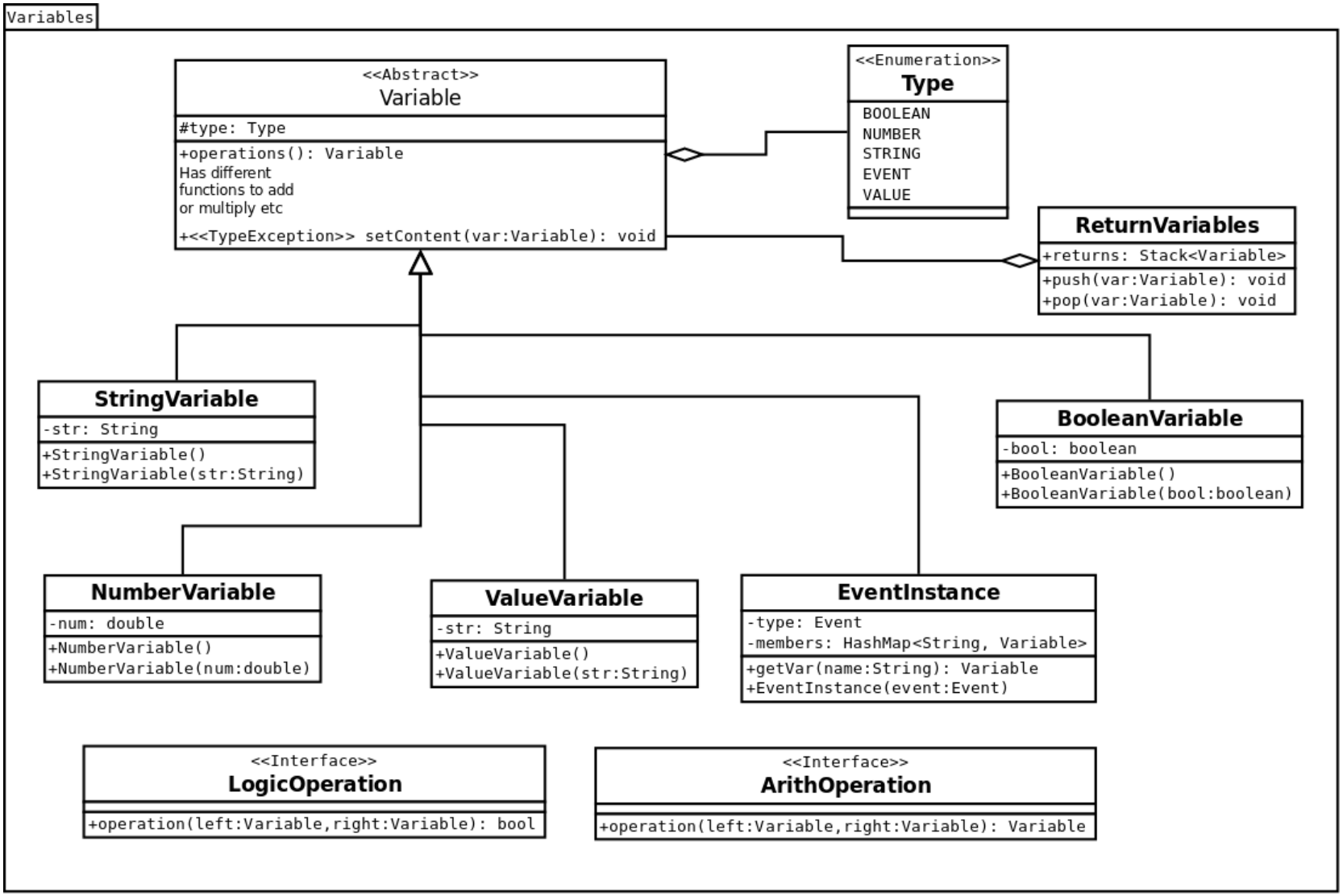
Deze wordt afgeleid van de Abstact Class Variable en bevat dus een boolean als variable.

NumberVariable

Deze wordt afgeleid van de Abstact Class Variable en bevat dus een number als variable.

StringVariable

Deze wordt afgeleid van de Abstact Class Variable en bevat dus een String als variable.



Figuur 17: Variables Module.

8.3 Blocks

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.3.

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.

HandlerBlock

Deze implementeert de interface Block. Deze bevat een `ArrayList<Block>` [11] die we de body noemen. De heeft een EventInstance die hij mee kreeg op oproep, via zijn constructor. 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.

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.

BroadCastBlock

Bevat een naam van het Event. En een Hashmap [9] 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.

FunctionBlock

Deze implementeert de interface Block. Deze bevat een `ArrayList<Block>` [11] 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>` [11] voor respectievelijke parameters en return parameters.

FunctionCallBlock

Deze bevat een String voor de functie naam. En twee `ArrayList<String>` [11] 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` [11] 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.

VariableBlock

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

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.

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 6.9 om de juiste operatie uit te kunnen voeren. De uitvoering zal dus in een primitieve stap gebeuren.

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.

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 die een string op een gegeven index. 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.

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 6.9 om de juiste operatie uit te kunnen voeren.

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.

ForeverBlock

Deze block bevat een `ArrayList<Block>` [11] die we body noemen. De execute van deze block zet al zijn blokken op de stack met onderaan de body ook nog eens zichzelf geplakt.

WhileBlock

Deze bevat een Block conditie en een `ArrayList<Block>` [11] 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>` [11] 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.

HideBlock

Voert een functie van de instance uit om te hiden.

ShowBlock

Voert een functie van de instance uit om te showen.

ChangeAppereanceBlock

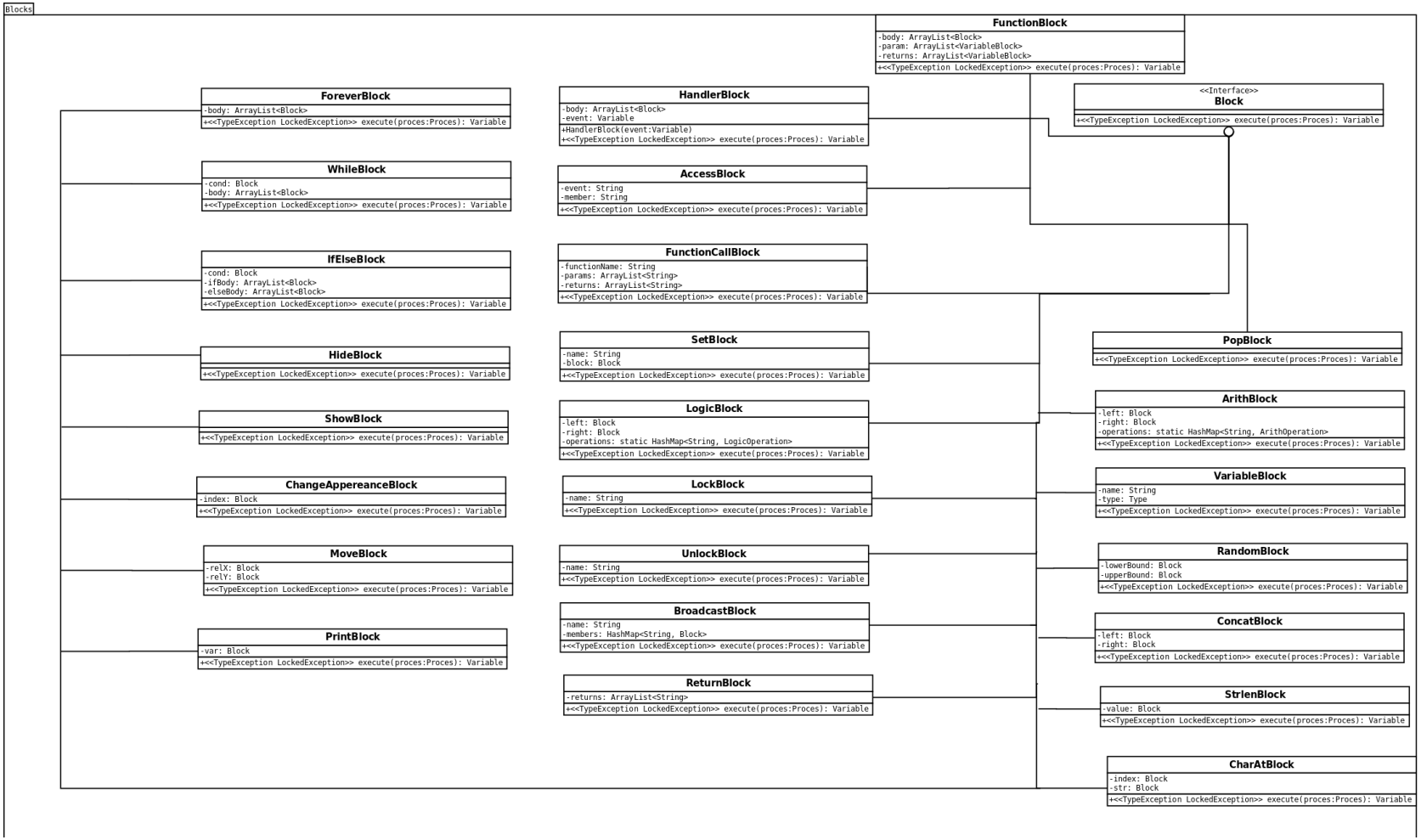
De bevat een Block index, deze kan een arith expression zijn. Dus we laten hem execute zodat we de index krijgen in een variable. Dit zal de execute van de blok doen plus het oproepen van de functie bij een instance die de appereance veranderd.

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.

PrintBlock

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



Figuur 18: Block Module.

8.4 Collections

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.4.

Classpool

Deze bevat alle Classes die aangemaakt zijn in het programma. Deze slaat hij op in een HashMap [9] waarbij een String (de naam) wordt gemapt op een Class. De naam van een Class moet dus uniek zijn.

Keuze ADT's: De keuze van een HashMaps [9] zorgt ervoor dat het toevoegen van nieuwe items en het opzoeken van bestaande in O(1) tijd kan gebeuren.

WiredInstance

Een wiredInstance bevat een HashMap [9] waarbij een event wordt gemapt op een lijst van instances. We gebruiken hiervoor dus `java.util.Hashmap<Event, ArrayList<Instance>>` [9] [11]. Dit stelt alle instances voor die verbonden zijn met ëën specifiek uitgaande event van ëën specieke instance.

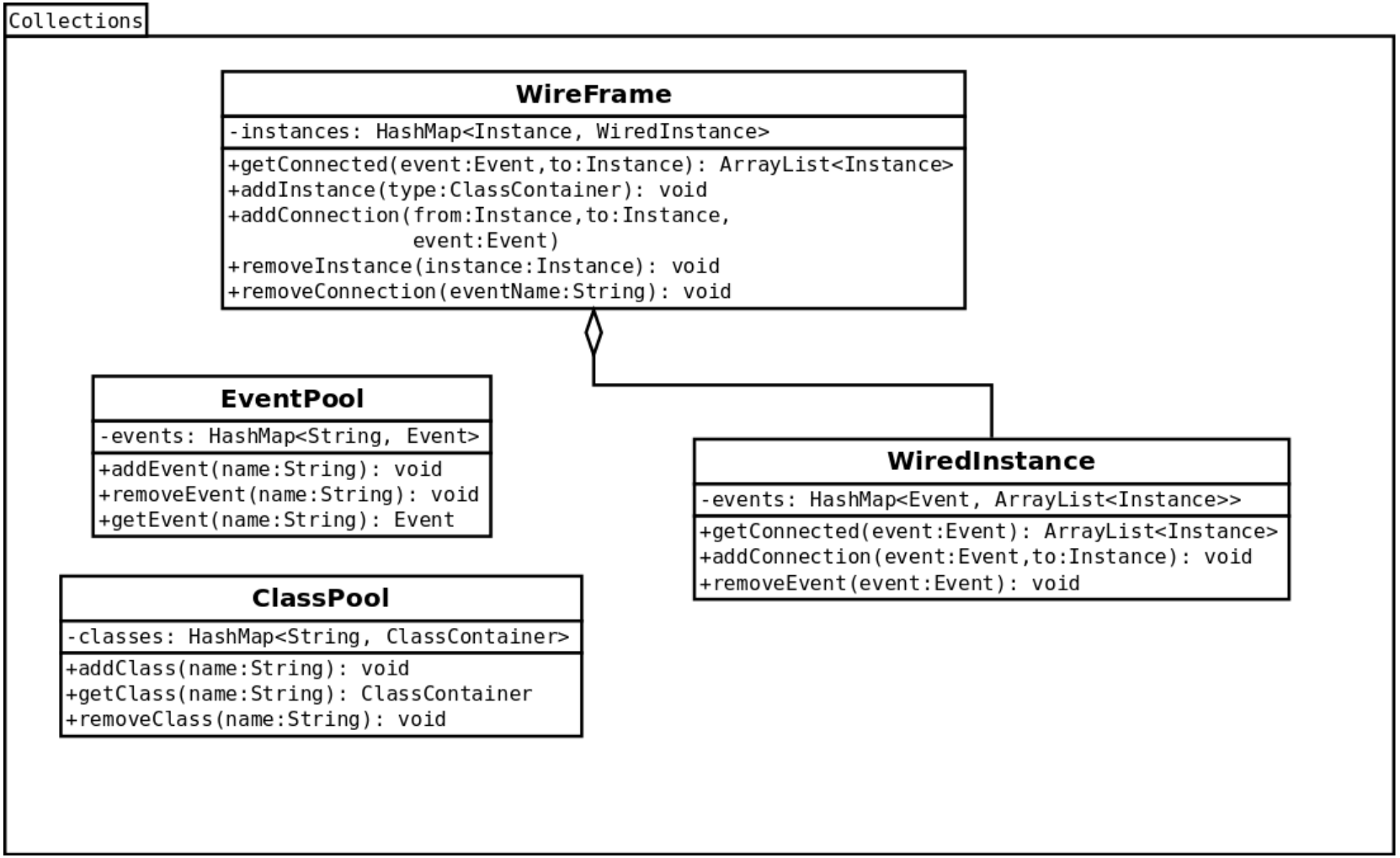
WireFrame

Een WireFrame stelt de plaats voorwaarbij alles instanties met elkaar worden verbonden met wires. Deze bevat een hashmap die Instances mapt op WiredInstances [9].

Keuze ADT's: Door gebruik te maken van deze structuur van Hashmap [9] en WiredInstance kunnen we snel toevoegen en opzoeken. Bij het toevoegen van een wire zal er voor de instance zijn wireFrame worden opgevraagt en voor het event dat verbonden wordt zal de instance waar de wire aankomt worden toegevoegd aan de bijhorende Arraylist [11]. Deze operatie duurt dan O(1) + O(1) + O(1).
Het opvragen van alle van instances die verbonden zijn met één event van één instance kan dus op dezelfde manier in O(1) + O(1).
Het is ons bekend dat het verwijderen van een wire op deze manier niet snel kan verlopen. Het verwijderen gebeurt echter niet op runtime. Daar is het zoeken van belang. We hebben dit in afweging genomen en hebben besloten te gaan voor het snel werken van de VM en dus gekozen voor snel opzoeken.

EventPool

Bevat alle Events die in het programma aangemaakt zijn. Deze bewaart hij in een HashMap [9] waarbij een String (het type) wordt gemapt op een Event. Het type van een Event moet dus uniek zijn (voor de hashmap).



Figuur 19: Collections Module.

8.5 Core

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.5.

Instance

Deze klasse stelt een instantie van een Class voor. Deze bevat een HashMap [9] van variables deze zijn zijn member variables maar ook zijn positie. Een instance bevat ook zijn Class zodat men weet welke events en functies deze bevat. Een instance bevat ook een HashSet van zijn locked members.

EventDispatcher

De EventDispatcher krijgt een eventInstance en een afzender van de virtual machine. De EventDispatcher kent alle wires en instances. Hij zal voor het gegeven eventInstance alle handlers van de ontvangers opvragen die voor dat event met de afzender zijn verbonden. Voor elke handler creeërt hij een nieuw proces dat hij aan de virtual machine geeft.

Proces

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 8.2)(zie Sectie 8.5).
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 6.8.

Keuze ADT's: De Stack [10] van Blocks zorgt ervoor dat telkens de bovenste blokken geexecute kunnen worden. De functie wordt er dus telkens volledig bovenop gezet. Omdat we enkel bovenaan moeten toevoegen en uitvoeren gebeurt deze operatie in O(1). We gebruiken hiervoor `java.util.Stack<Block>` heb Voor informatie over het toevoegen van de blokken van een functie op deze stack verwijzen we naar de Class FunctieBlock.

Het gebruiken van een Stack [10] zorgt ervoor dat we weten in welk blok er gezocht moet worden naar de lokale variabele van de huidige functie. Hierdoor

kan de totale operatie van zoeken in O(1). Als een variable niet gevonden wordt, zoekt men in de member variables van de instance.

VirtualMachine

De virtual machine bevat een lijst (queue) van processen (zie Sectie 8.5)en de eventDispatcher. Hij doorloopt de lijst van processen door telkens de voorste eraf te halen. Hij zal elk proces één primitieve stap laten uitvoeren. Als het proces in die primitieve stap een event creeërt zal de VM (virtual machine) dit event en zijn zender doorgeven aan de eventDispatcher. Zoals eerder vermeldt zal de eventDispatcher een hoeveelheid processen teruggeven (`ArrayList<Proces>` [11]). Deze worden achteraan de lijst toegevoegd. Als het huidige proces nog niet is afgerond wordt dit ook achteraan terug toegevoegd (na de eventuele nieuwe processen).

Keuze ADT's: Voor de queue gebruiken we `java.util.LinkedList<Proces>` [12] hierdoor kan het afnemen van het eerste element en toevoegen van een element aan de queue in O(1). Natuurlijk is de orde van belang in de queue wat hier FIFO is.

FunctionFrame

Een functionFrame stelt het stackframe voor van een functie blok. Deze bevat een hashmap [9] waarbij de lokale Variables van die functie gemapt worden op hun naam in die functie.

Keuze ADT's: Een Hashmap waarbij Strings worden gemapt op variables. 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 compile stap nodig omdat we de variable dadelijk kunnen mappen op de content en geen indices van een array moeten worden berekent. [9]

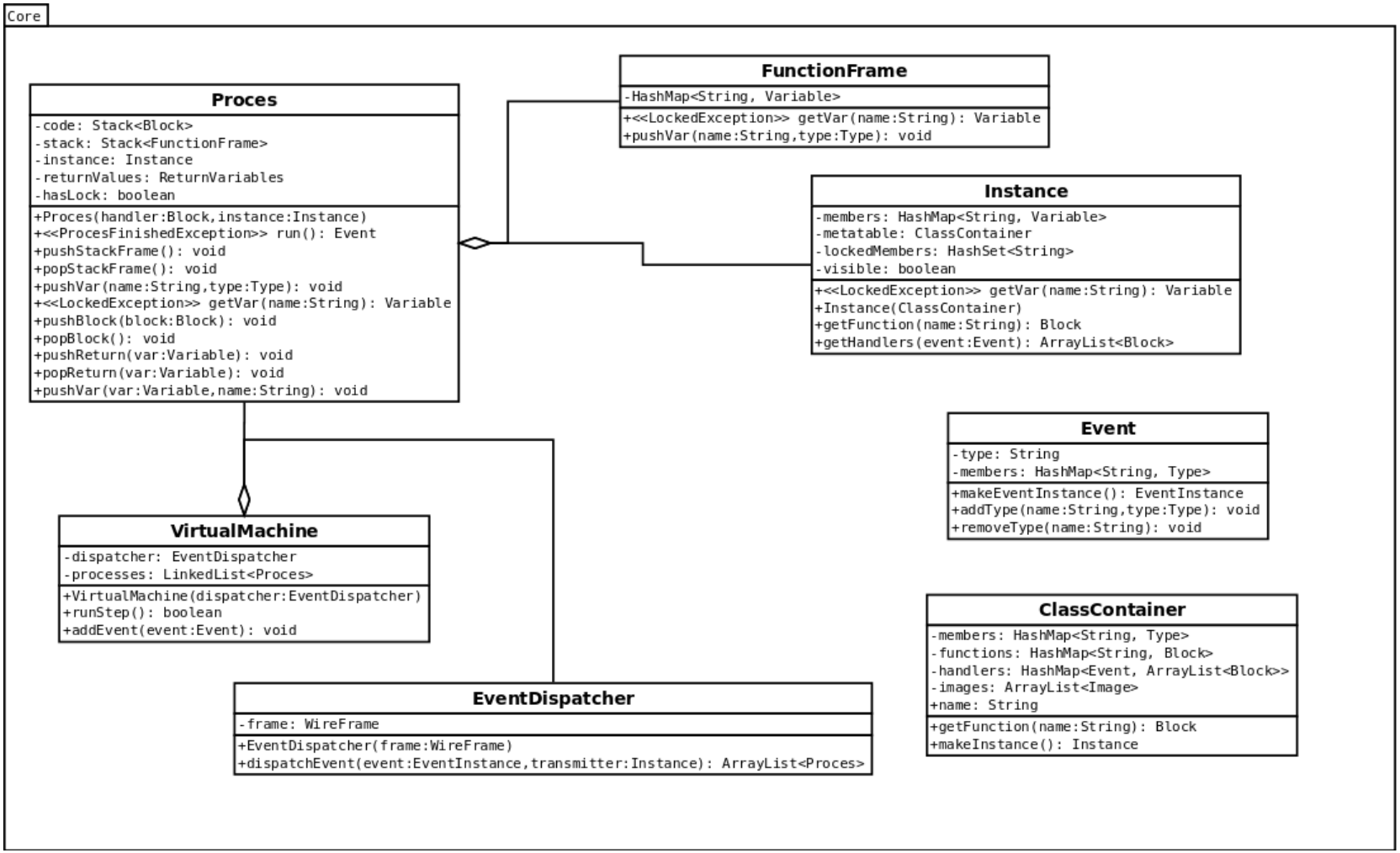
Event

Een event bevat een String voor zijn type aan te duiden en een HashMap waarbij het de naam van een variable (String) mapt op een type van variabele. Deze klasse kan een EventInstance van dit event aanmaken [9].

Class

Een Class bevat twee lijsten inputEvents en outputEvents die respecitievelijk alle inkomende en uitgaande events bevatten [11]. Een Hashmap van Strings gemapt op Types die zijn membervariabele voorstellen. De klasse bevat ook twee Hashmaps [9] functions en handlers die respecitievelijk alle functies en alle handlers bevatten. Bij functions worden functies gemapt op hun naam en bij

handlers worden gemapt op het event waarbij ze worden opgeroepen. Een Class is een factory voor zijn instances.



Figuur 20: Core Module.

8.6 Model

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.6.

Abstracte Klasse BlockModel

Deze abstracte klasse wordt als basis voor elk model van bruikbare blokken in de IDE. Deze bevat alle gemeenschappelijke attributen van alle modelen. Deze bestaan uit de positie van het view van die block in de klasse. Alsook een Boolean die aangeeft als de blok moet oplichten bij het debuggen.

Hieronder geven we enkele afgeleide blokken, een volledige beschrijving zo echter te verleiden voor dit verslag.

ClassModel

Deze klasse wordt gebruikt om een klasse in de IDE voor te stellen. Deze bevat de volgende attributen: Alle functieblokken, handlers, naam van de Klasse, outputEvents, member variabelen, images en floatingBlocks. Het gebruik van alle vorige attributen volgt logische uit de beschrijving van een Klasse. Behalve floatingBlocks deze wordt gebruikt voor het groeperen van alle blokken die de gebruiker op het veld van de Klasse heeft geplaats maar nog niet in een functie of handler heeft geplaatst.

Keuze ADT's: Voor het bijhouden van welk inputEvent wordt afgehandeld door welke handlers gebruiken we een HashMap die het EventModel mapt op een lijst van handlers. Hiervoor gebruiken we `java.util.Hashmap<EventModel,java.util.ArrayList<HandlerModel>>`. Voor de het bewaren van de member variabele en hun type gebruiken we een Hashmap die de naam van de variabele mapt op een Type 8.2. Hiervoor gebruiken we `java.util.Hashmap<String, Type>`.

WhileModel

Dit is een voorbeeld uitwerking van een BlockModel voor een programmeerblokje. De klasse van een programmeerblok model die andere blokken kan bevatten heeft voor alle blokken van die klasse een statische lijst van alle BlockModel's die de blok mag bevatten. Aangezien een While-blok twee input velden heeft, scheiden we deze hier door een conditieModel die hier verder wordt uitgewerkt en een lijst van blokken die geaccepteerd zijn in de body.

Keuze ADT's: Voor het bijhouden van welke blokken een blok mag bevatten gebruiken we een `java.util.HashSet<Class>`. Hiermee kan dan worden nagegaan als een blok in een andere kan worden geplaatst. We kiezen voor een HashSet zodat het controlleren een constante tijd kan gebeuren.

WireFrameModel

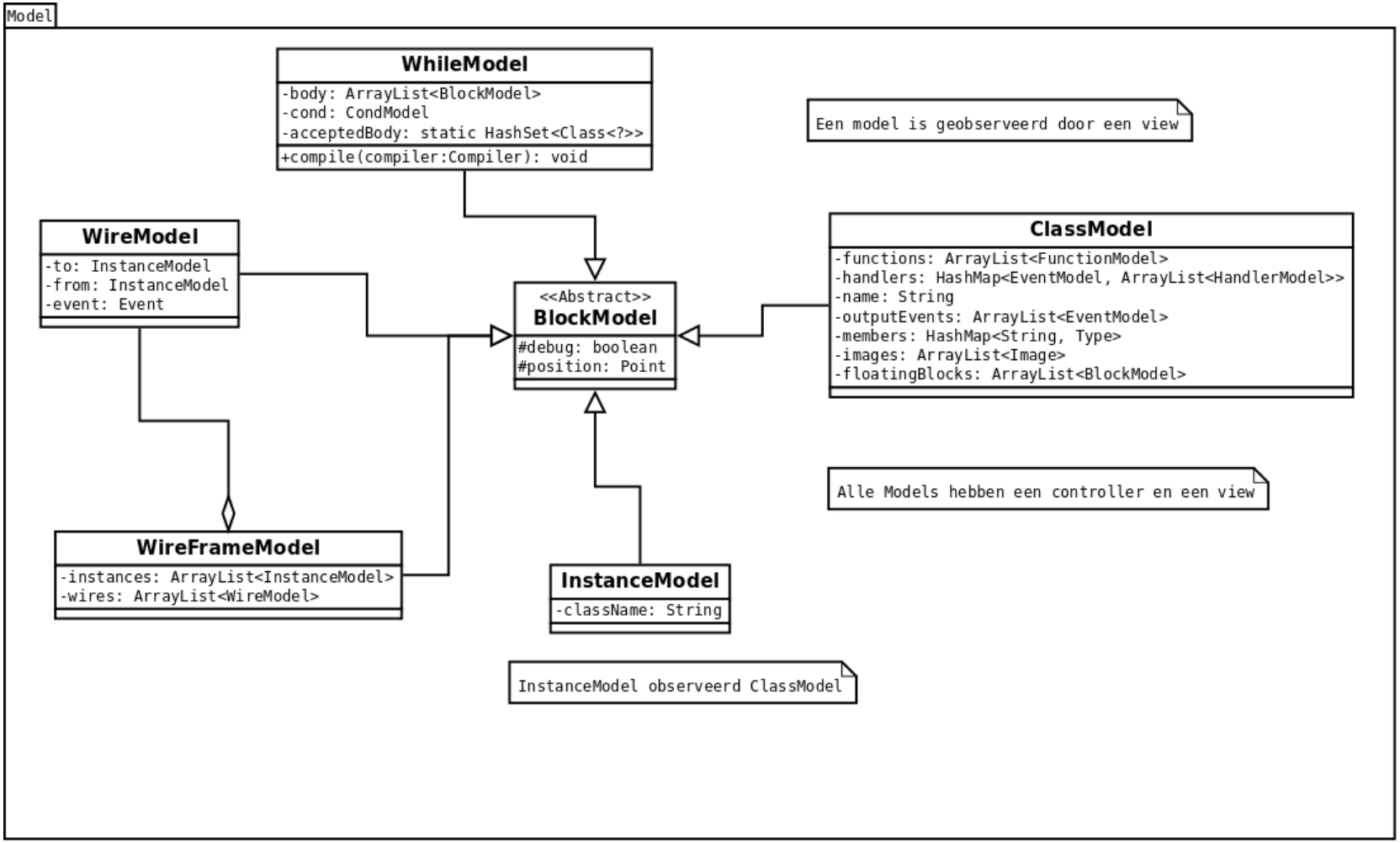
Door dat de compiler de juiste compile functie zoekt op het type, is het WireFrame ook een BlockModel. Deze klasse bevat alle instanties die in het frame staan als ook de wires die ze verbinden. Beide worden bijgehouden in een lijst.

WireModel

Dit is een blockModel om een wire voor te stellen. Deze bevat een InstanceModel waarvan het vertrekt en een waar het aankomt. Ook weet de wire welk event het verstuurd.

InstanceModel

Dit is een model voor een instance in het WireFrame. Een Instance hoort bij een Klasse. Daarom zal hij de Klasse waarvan hij een instantie is observeren voor veranderingen.



Figuur 21: Model Module.

8.7 RunTime

De verantwoordelijkheden van deze module worden beschreven in Sectie 7.7.

Abstracte Klasse Runtime

Deze klasse wordt gebruikt voor het uitvoeren van het gecreeërde programma. Alle Klassen en hun blokken zullen eerste door een Compiler die deze klasse bevat gecompileerd worden. Na het succesvol compileren zal de Runtime verantwoordelijk zijn voor het starten van de virtual machine, het stoppen ervan en het doorgeven van inputEvents veroorzaakt door de gebruiker aan deze virtual machine.

InterpeterRuntime

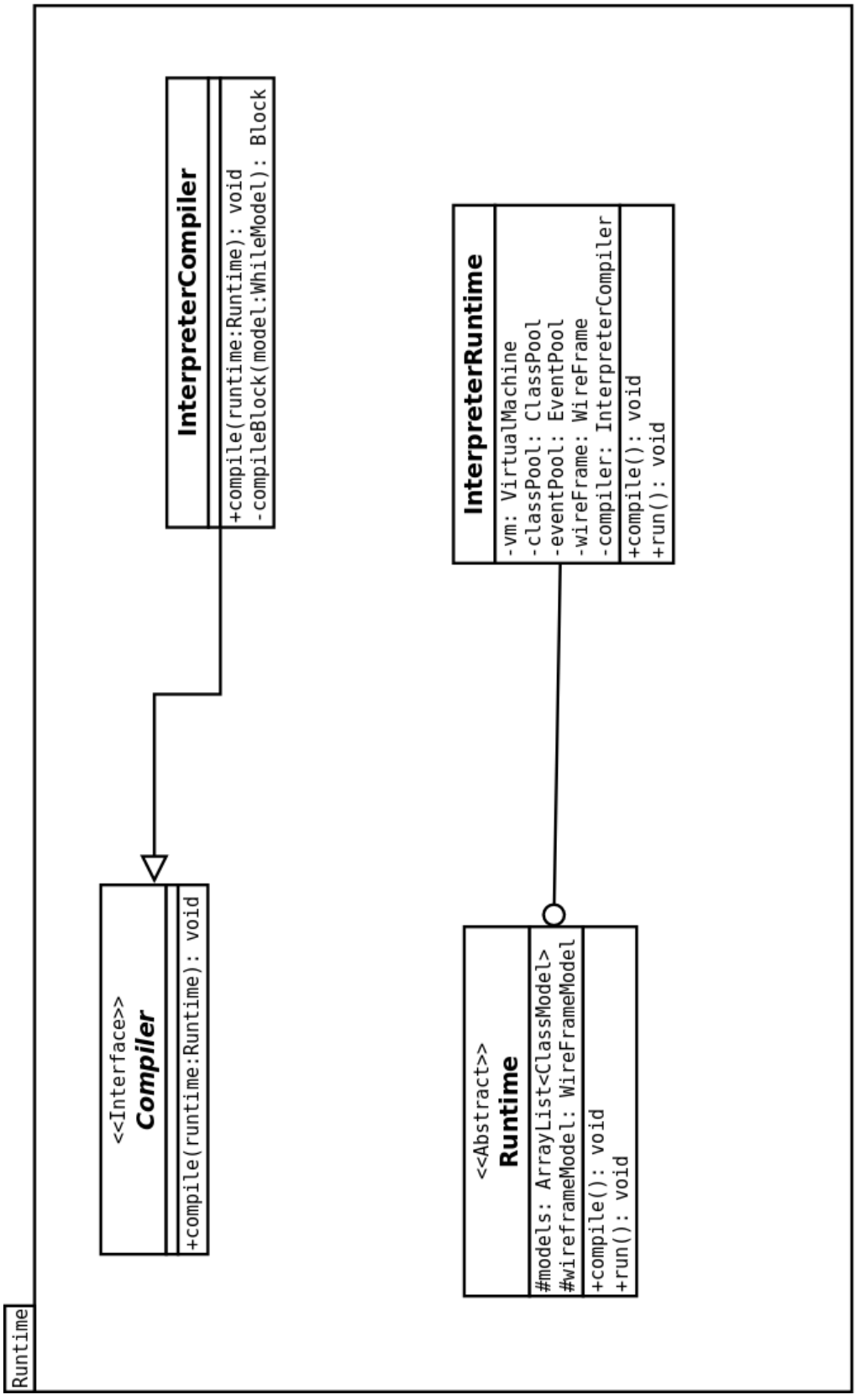
Dit is de runtime die geïmplementeerd zal worden voor dit project. Deze bevat een virtual machine die beschreven wordt in Sectie 8.5. De compiler die wordt gebruikt in dit project wordt beschreven in Sectie 8.7. Deze compiler vult de attributen Classpool, EventPool en WireFrame in deze worden respectievelijk beschreven in Sectie 8.4, Sectie 8.4 en Sectie 8.4.

Interface Compiler

We voorzien een Interface voor de compiler zodat er makkelijk een andere compiler kan worden geïmplementeerd. Deze bevat een functie voor het compileren van een Runtime.

InterpeterCompiler

Deze klasse bevat de compiler die geïmplementeerd zal worden voor dit project. De bevat functies voor het compileren van een model. Een gedetailleerd beschrijving is te vinden in Sectie 6.6.



Figuur 22: Runtime Module.

8.8 File

LanguageModule

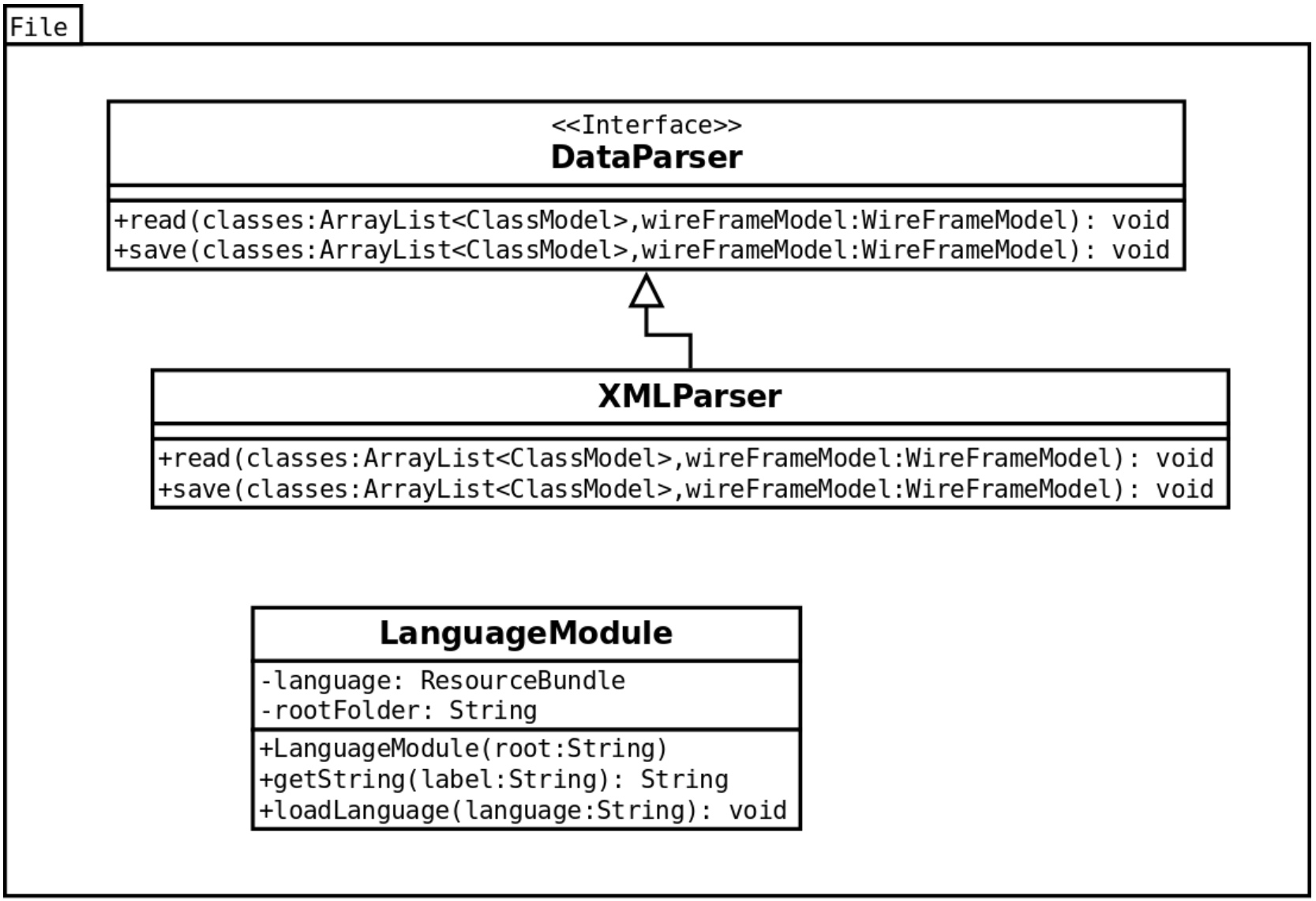
De language class zorgt voor het multilanguage maken van de IDE. Dit gebeurt door een `ResourceBundle` zoals uitgelegd in Sectie 9.1

DataParser

Een interface klasse die de IDE gebruikt om data te schrijven naar een file. Deze data zal bestaan uit de informatie zijn die nodig is voor projecten die bestaan uit Events, Instanties, Klassen en gerelateerde data.

XMLParser

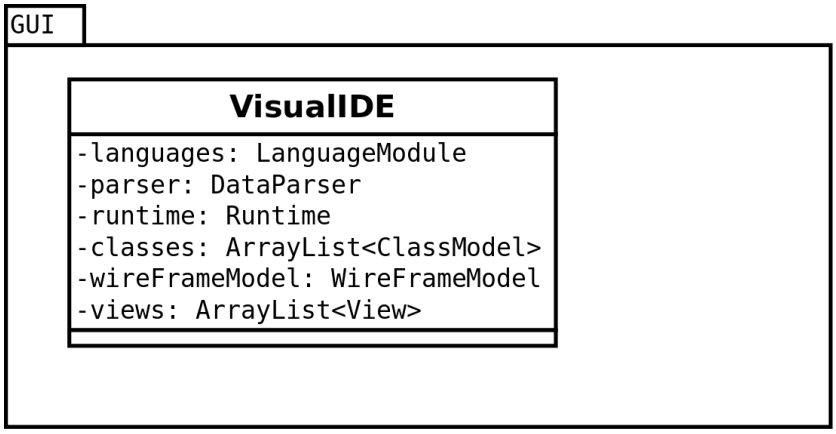
Dit is een implementerende klasse van een DataParser. Deze klasse wordt gebruikt om XML data te lezen en te schrijven naar een bestand. De XML parser is een information expert met betrekking tot XML data, hij bevat functies waarmee de Modellen van een Klasse, Instantie, enz. geschreven en gelezen kunnen worden. De XML data zelf wordt geparsed door een `DOMParser` van de Java Standard Library.



Figuur 23: File Module.

8.9 GUI

In dit verslag is enkel de Main klasse uitgewerkt. Dit is de klasse die het JFrame zal bevatten alsook andere noodzakelijke data. Onder andere bevat deze klasse een LanguageModule, een DataParser, een Runtime en Modellen van aangemaakte Klassen/WireFrame



Figuur 24: GUI Module.

9 Bestand

In deze sectie staat uitgelegd hoe externe files opgeslagen worden. In een eerste sectie staat hoe de IDE multilanguage gemaakt wordt en hoe de externe bestanden eruit zien. In de tweede sectie staat hoe projecten kunnen worden opgeslagen in een leesbaar formaat.

9.1 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 [13].

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 [14]. Hieronder staat beschreven hoe de files voor verschillende talen genoemd moeten worden. De inhoud van deze files volgt een simpel formaat nl, key = tekst [15].
Voorbeeldcode: [15]

```
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
```

9.2 XML Blokken

De blokken waarmee de gebruiker kan werken, wordt opgeslagen in verschillende XML files [16]. De IDE zal bij het inlezen van opgeslagen projecten, of bij het opslaan van een project deze XML file structuur gebruiken. XML is een leesbaar formaat en kan gemakkelijk met de hand aangepast worden.

De data die moet worden opgeslagen, wordt geschreven naar verschillende files. Alle instanties en de wires ertussen worden opgeslagen in een aparte file. Dit kan gezien worden als de main file. Events worden elk in een andere file opgeslagen. De Klassen worden ook elk in een andere file opgeslagen. Dit promoot herbruikbaarheid. De gebruiker van de IDE (en de IDE zelf) kan op deze manier gemakkelijk files importeren van een ander project. Elke Klasse heeft ook een lijst van nodige events zodat deze mee geïmporteerd kunnen worden. De beschrijving van de XML voor alle geïmplementeerde blokken en XML DTD is achteraan beschreven in bijlage B.

10 Mockups

10.1 Menubalk

In de menubalk van de applicatie vindt de gebruiker onder Bestand de mogelijkheid terug om een programma op te slaan, in te laden of om een nieuw programma te maken. Bij instellingen kan hij de console aan en uitzetten en de taal van de applicatie veranderen.
In het midden van de menubalk vinden we een start knop waarmee de uitvoering van het programma wordt gestart mits het correct geconstrueerd is. Hiernaast bevindt zich ook de knop om de uitvoering te stoppen.
Aan de rechterzijde bevindt zich een switch bestaande uit drie knoppen: Klasse, Kabels en Events. Deze stellen de verschillende delen van de IDE voor die gebruikt worden om een programma te creeëren. De beschrijving van deze delen op hoog niveau staat in Sectie 1.2.

10.2 Klasse creatie

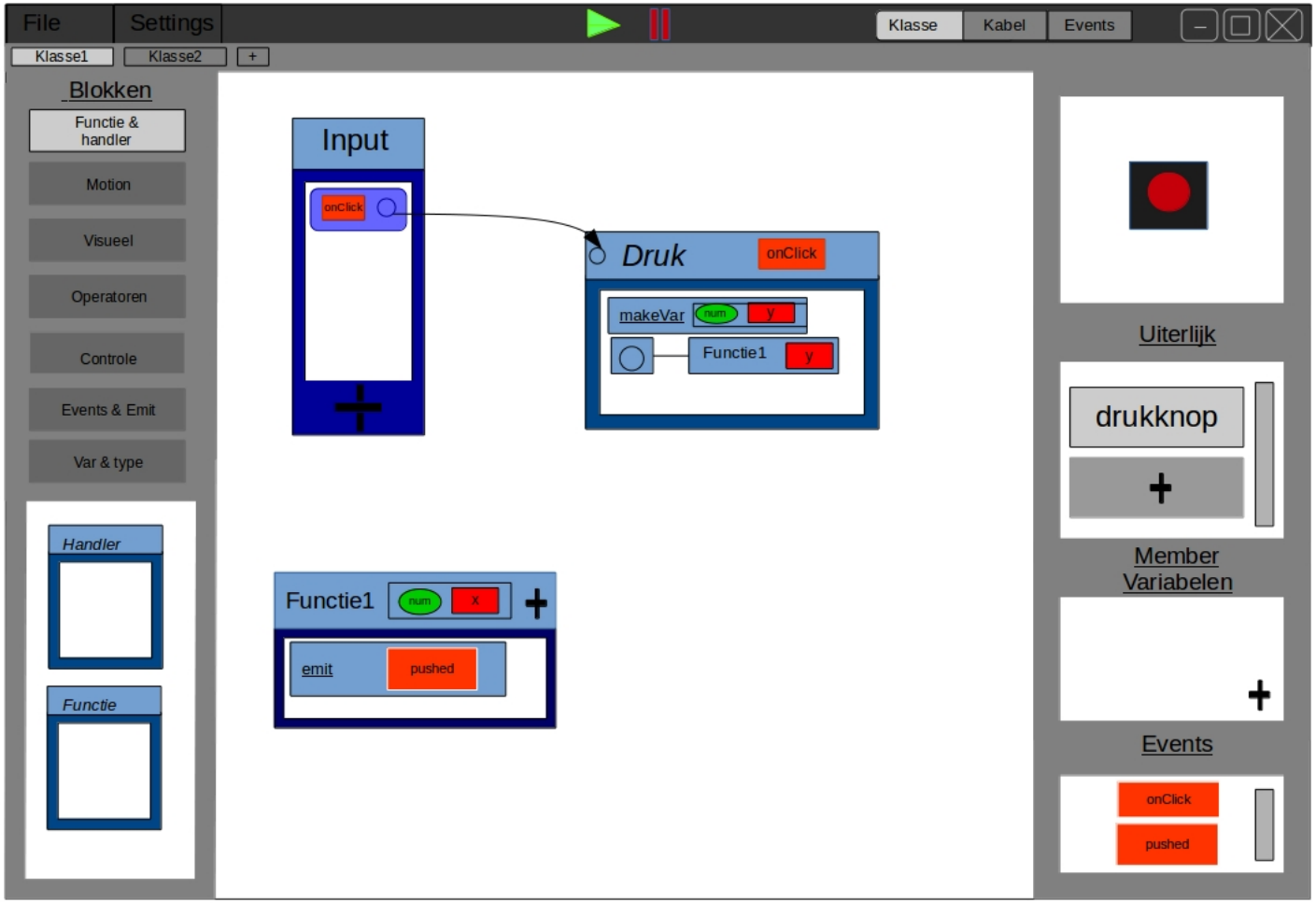
Bij het selecteren van Klasse in de switch zal aan de gebruiker het klasse creatie view getoond worden. De mockup van dit view is te zien op figuur 25. Hierin vindt hij onder de menubalk een balk waarin de reeds gecreeërde klasse zich als tabbladen bevindt. Op het einde van deze balk bevindt er zich een plus toets waarmee een nieuwe klasse kan worden toegevoegd.

Aan de linkerzijde vindt de gebruiker alle categorieën van de programmeerblokken. Bij het selecteren van een categorie zullen alle blokken die tot die categorie behoren verschijnen onder de categorieën. Bij het aanklikken van een blok zal deze op het creatie veld verschijnen.

De kleuren en vorm van blokken, types en events moet nog aangepast worden op het eenvoudige gebruik van de applicatie en het proffesionele uitzicht ervan. Onderstaande mockups dienen voor het aantonen van de werking van de applicatie.
Het inelkaar klikken van blokken zal op een intuïtieve manier moeten duidelijk worden gemaakt. Als een blok de andere niet accepteerd zal deze er boven op blijven zweven maar zal het voor de gebruiker duidelijk zijn dat de blokken niet verbonden zijn.

Het verbinden van input events met de handler die de gebruiker wenst gebeurt met pijlen. Hierbij zal een pijl enkel kunnen verbonden worden als de handler ook het juiste event als input parameter heeft.
Het definieren van een functie gebeurt doormiddel van een functieblok. Deze kan de gebruiker een naam geven. Voor het toevoegen van een parameter moet de gebruiker op plus tekenen duwen en een variable en een type invullen. Het oproepen van een functie kan met een functiecallblok. Deze bestaat uit een blokje waaraan een pijl zit met een andere blok waarin de gebruiker een functie kan selecteren uit een lijst van functies uit die klasse. Onderaan dit blok kan een blokje staan waarin de return waarde van de functie kan worden opgevangen.

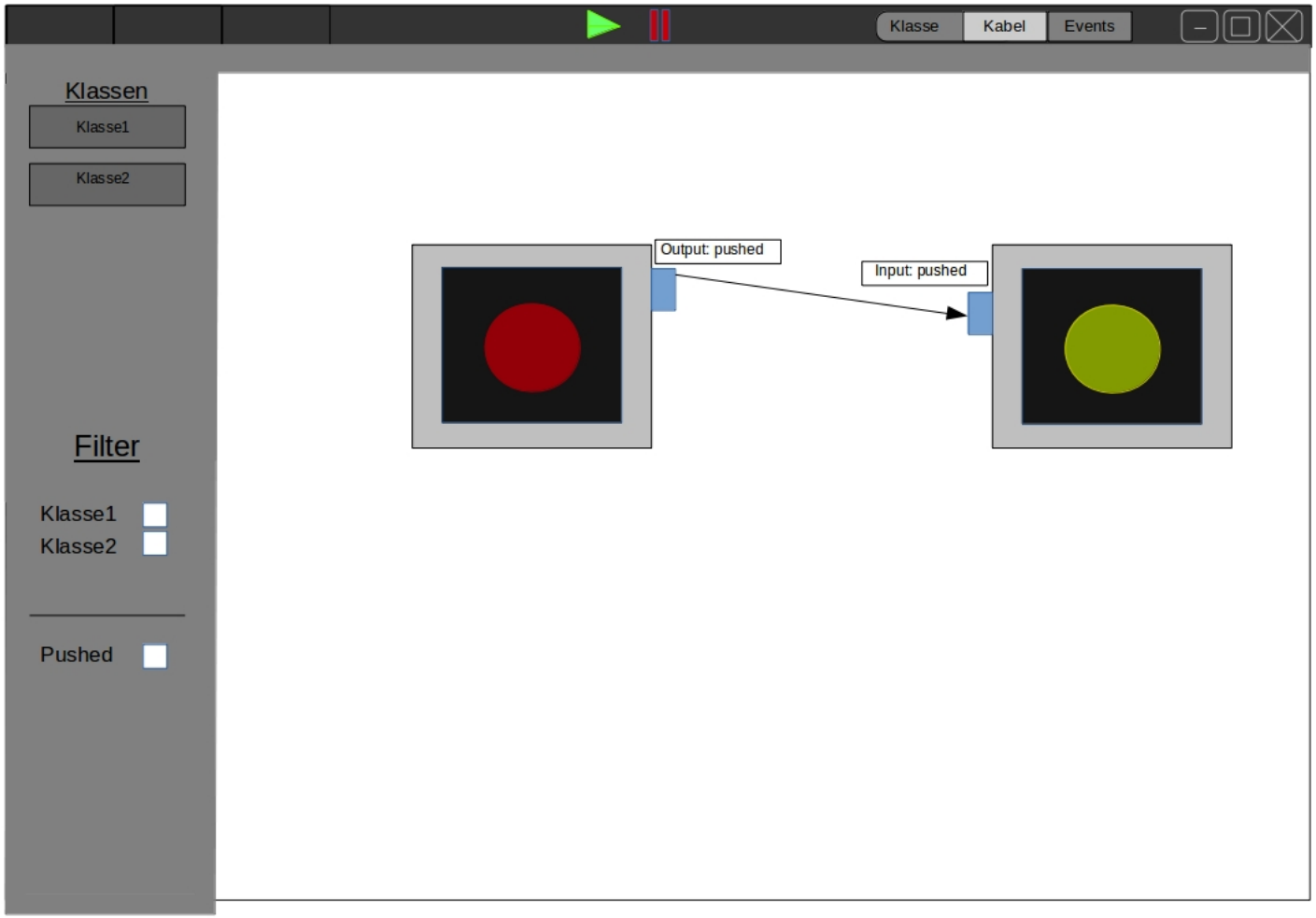
Aan de rechterzijde van de klasse creatie bevinden zich alle uitlijken van die tot een klasse behoren. Zo kan een klasse lamp bijvoorbeeld twee uitlijken bezitten namelijk een uit en aan afbeelding. Deze kan dan worden veranderd door een verander van uiterlijk blok.
Hieronder bevinden zich de member variable van de klasse. Deze kan de gebruiker aanmaken door op het plus teken te drukken en een variabele van het gewenste type aan te maken met een unieke naam. Hiervan kunnen dan blokken op het creatie veld worden geplaats om te gebruiken in functies of handlers. Tenslotte vind de gebruiker hieronder alle events die hij reeds gedefineerd heeft in de event sectie van de IDE.



Figuur 25: Mockup creatie klasse.

10.3 Creatie instanties en kabels

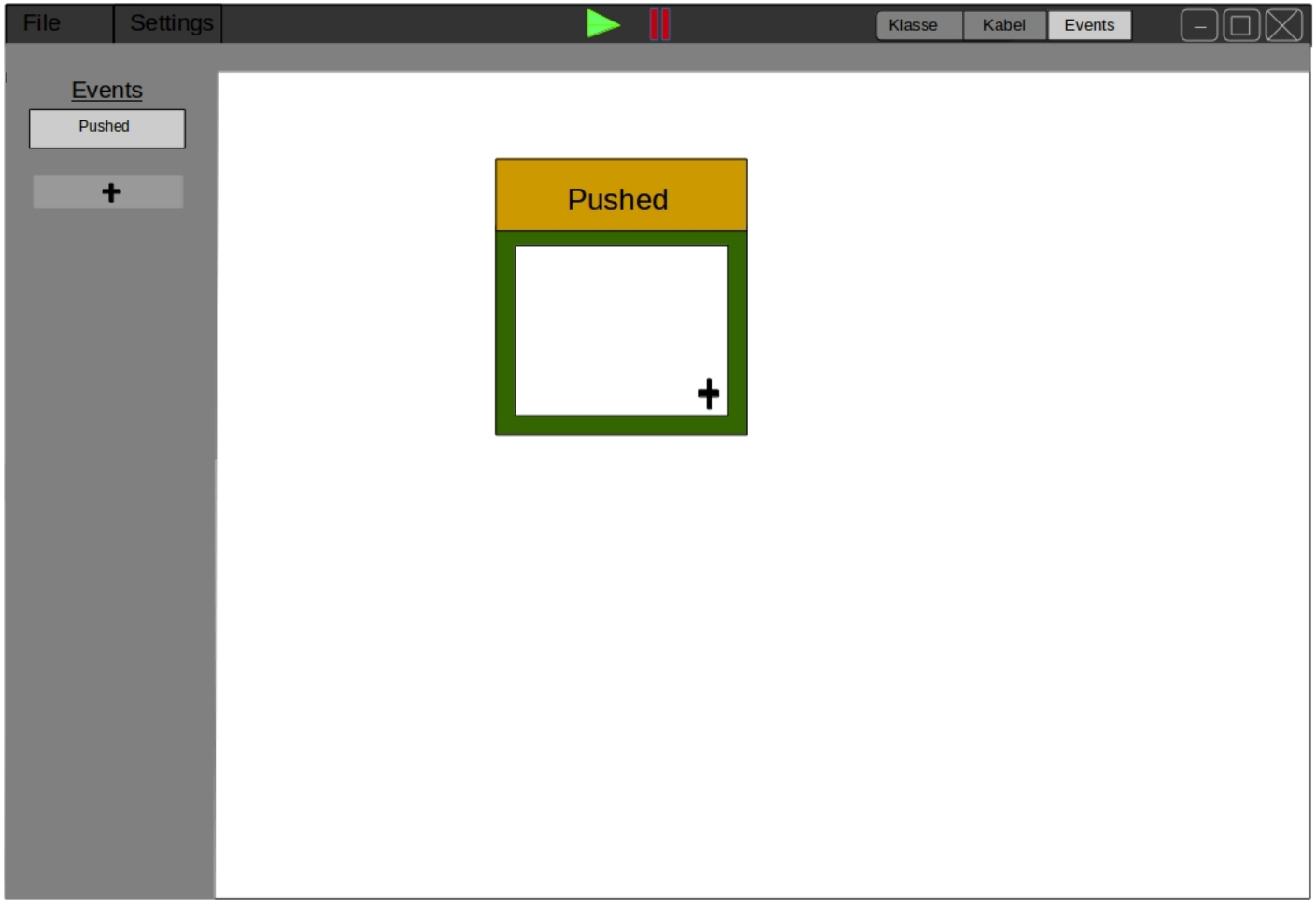
In dit view kan de gebruiker het hogere niveau van de werking van zijn programma definieren. De mockup van dit view is te zien op figuur 26. Dit doet hij door instanties aan te maken van zijn eerder gedefineerde klassen. Hiervan heeft hij dan de keuze om de uitgaande events en inkomende event met elkaar te verbinden met behulp van pijlen. Hiermee kan hij beslissen aan welke instantie hij een event doorgeeft.
Bij een programma met veel kabels kan dit al snel voor een hele cluster zorgen. Daarom wordt er een filter voorzien waarmee de gebruiker kan bepalen van welke klasses hij geen kabels toont of welke event kabels hij niet toont.



Figuur 26: Mockup WireFrame.

10.4 Event creatie

In dit view kan de gebruiker een nieuw event definiëren of een eerder gedefineerd event aanpassen. De mockup van dit view is te zien op figuur 27. Aan linkerkant van het venster bevindt er zich een overzicht van alle reeds gedefineerde events. Om een member aan het event toe te voegen gebruikt hij de plus knop en dan kan hij een member toevoegen van een gewenst type met een unieke naam. Een eerder toegevoegde member kan makkelijk verwijderd worden.



Figuur 27: Mockup creatie event.

11 Taakverdeling en Planning

11.1 Planning

We werken iteratief aan het software project. Er wordt gezorgd dat het eindverslag wekelijks bijgewerkt wordt. Voor de verschillende modules wordt eerst de kern geïmplementeerd zodanig dat er een basis versie van het project bestaat. Deze versie word in verdere weken uitgewerkt tot een finale versie. In deze lijst staan telkens de belangrijke elementen die die week geïmplementeerd worden.

- Week 1: Exceptions en Variables module maken. Aanmaak van enkele blokken uit de Blocks Module om te kunnen testen. Beginnen aan het aanmaken van de Core module
- Week 2: Aanmaken van de Core module zodanig dat de Virtual machine operationeel is. Aanmaak XML inlezing zodat programma's al gemaakt kunnen worden in XML formaat. Hierdoor kan de Core module al volledig getest worden.
- Week 3: Aanmaken Collections en Runtime module. Hierdoor ligt een basis voor de GUI klaar
- Week 4: Aanmaken van een eerste GUI prototype (en de multilanguage klasse) en de blok modellen.
- 30 April 2015: Tussentijdse rapportering met opdrachtgevers.
- Week 5: Verdere implementatie van alle blokken.
- Week 6: Professioneel look geven aan de GUI.
- Week 7: GUI verder afwerken en de blokken en hun modellen volledig afwerken.
- Week 8: Extra tijd om extra's te implementeren of eventuele achterstand in te halen.
- Week 9: Extra tijd om extra's te implementeren of eventuele achterstand in te halen.
- 4 Juni 2015: Inleveren eindverslag en project.
- 9 Juni 2015: Eindpresentatie finale software project.

11.2 Taakverdeling

Hierin staat beschreven wie welke onderdelen van een bepaalde module implementeerd.

- Exceptions module: Matthijs implementeerd alle exceptions

- Variables module: Axel implementeerd deze module volledig.
- File module: Language klasse wordt gemaakt door Axel
- File module: Aangezien de dataparser klasse zeer groot is (zo'n 40 functies) zal ieder van ons de helft van deze functies implementeren. Deze functies zijn gericht op implementatie (een functie kan een while loop inladen of opslaan). Hierdoor is er nog geen exacte opsplitsen voor wie welke functie implementeert.
- Core module: Axel implementeerd de Instance, Class, Proces en Function-Frame. Matthijs maakt de Virtual Machine, Event en de Event Dispatcher.
- Collections module: ClassPool en EventPool klassen worden gemaakt door Axel. De WireInstance en het WireFrame worden gemaakt door Matthijs.
- Runtime module: Matthijs maakt de Runtime. Aangezien de compiler klasse zeer groot is (zo'n 40 functies) zal ieder van ons de helft van deze functies implementeren. Deze functies zijn gericht op implementatie (een functie kan een while loop compiler bv). Hierdoor is er nog geen exacte opsplitsen voor wie welke functie implementeert.
- Block module: Aangezien deze module zeer groot is (zo'n 40 klassen) zal ieder van ons de helft van deze klassen implementeren. Deze klassen zijn gericht op implementatie (een klasse kan een while loop voorstellen). Hierdoor is er nog geen exacte opsplitsen voor wie welke klasse implementeert.
- Model module: Aangezien deze module zeer groot is (zo'n 40 klassen) zal ieder van ons de helft van deze klassen implementeren. Deze klassen zijn gericht op implementatie (een klasse kan een while loop voorstellen). Hierdoor is er nog geen exacte opsplitsen voor wie welke klasse implementeert.
- Drag en drop: Dit wordt eerst samen bekeken. Zodat we goed weten hoe dit geïmplementeerd moet worden.
- WireFrame view: Matthijs implementeerd de WireFrame view (en andere benodigde views hierbij).
- Event view + menu balk: Axel implementeerd de event view (en andere benodigde views hierbij) alsook de menu balk.
- Programmeer view: Dit gaat opnieuw een grote hoeveelheid klassen zijn (voor de verschillende views van de verschillende blokken). Aangezien dit veel werk is, worden deze klassen opgesplitst.

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

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

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

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

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

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

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

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

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

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

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

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

[16] W3Schools, "XML Tutorial." http://www.w3schools.com/xml/default.asp/, 2015. [Online; accessed 15-Februari-2015].

12 Log

- 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.

A Bijlage Executing Blokken

A.1 Variables en Type

Deze categorie bevat alle blokken met betrekking tot variabelen en hun types.

Type blok

Een type blok toont een bepaalde type aan voor een variabele of parameter. De ondersteunde types zijn: strings, getallen (floating point getallen) en booleans. Lijsten zouden toegevoegd kunnen worden als extra.

Variabele blok

De variabele blok creeërt een nieuwe lokale variabele van het gegeven type met een gegeven unieke identifier naam.

Lock

Het lock blok bevat een variabele die gelocked moet worden.

Unlock

Het unlock blok bevat een variabele die geunlocked moet worden.

Value

Het value blok bevat een string dat de waarde voorstelt die de user ingetyped heeft.

Set blok

De set blok is een assignment blok. Deze assigned een waarde of de waarde van een variabele aan een andere variabele.

Var blok

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

A.2 Motion

Deze categorie bevat alle blokken die een Instantie kunnen laten bewegen in het visuele canvas.

Move blok

De move blok stelt een translatie voor van de instance. De blok laat de instance op de x-as bewegen met [x] stappen en op de y-as bewegen met [y] stappen. Beide waardes zijn floating point getallen.

A.3 Visual

Deze categorie bevat alle blokken waarmee een Instantie visueel kan veranderen op het canvas.

Show

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

Hide

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

Change appearance

Het changeAppearance blok maakt het mogelijk voor een instance om zijn uiterlijk te veranderen.

A.4 String operators

Deze categorie bevat alle blokken die te maken hebben met string manipulatie.

Concat

Het concat blok laat toe om 2 strings samen te voegen.

Length

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

CharAt

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

A.5 Operator

Het operator blok stelt een operator voor zoals +,-,/,*,<,>,... .

A.6 Logic operators

Deze categorie bevat alle blokken die te maken hebben met logische operaties.

Unaire logic operator

Het unLogicOpp"blok stelt een logische operatie voor die unair is.

Binaire logic operator

Het binLogicOpp blok stelt een logische operatie voor die binair is.

A.7 Arithmetic blokken

Deze categorie bevat alle blokken die te maken hebben met arithmische operaties.

Random

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

Arith blok

Een rekenkundige expressie blok berekent een expressie en geeft een number value terug.

A.8 Functions en Handlers

Deze categorie bevat alle blokken die gebruikt worden wanneer een gebruiker een functie of handler wilt maken of aanroepen.

Handler

Het handler blok stelt een speciale functie voor die een event opvangt.

param

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

Function

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

Return blok

Een blok die gebruikt wordt om één of meerdere variabelen te returnen van een functie.

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.

A.9 Class

Deze sectie bevat alle blokken die te maken heeft met een Klasse.

Class

Het Class blok stelt een volledige class voor. Deze list al zijn input events, alle verschillende emits en alle member variabelen.

A.10 Control Blokken

Deze categorie bevat alle blokken die te maken hebben met controle blokken.

Forever blok

De forever blok herhaald een stuk code vanaf oproep van de blok tot einde van programma.

If blok

De if blok bevat een conditie en een code blok dat wordt uitgevoerd als de conditie waar is.

If-else blok

De if-else blok bevat een conditie en twee code blokken. Bij het waar zijn van de conditie wordt de eerste blok code uitgevoerd anders de tweede blok.

While blok

De while blok bevat een conditie. Zolang die conditie naar true wordt gevalueerd wordt de code herhaald.

A.11 Events en Emits

Deze categorie bevat blokken waarmee de gebruiker events kan gebruiken.

Emit blok

De emit blok verstuurt een event van een bepaald type dat er wordt ingevuld.

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.

Member blok

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

Access blok

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

A.12 Instances en Wires

Deze categorie bevat alle blokken die nodig zijn om een programma flow te maken.

Instance

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

Wire

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

WireFrame

Een wireFrame bevat instances en de wires tussen die instances.

B Bijlage XML Blokken

B.1 Variables en Type

Type blok

De `type` blok specifieert een bepaald type bv. een string of getal.

```
<type name="name of type" />
```

De DOCTYPE declaration:

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

Variabele blok

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

```
<makeVar name="name of variable">
       <type name="number" />
</makeVar>
```

De DOCTYPE declaration:

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

Lock

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

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

De DOCTYPE declaration:

```
<!ELEMENT lock (var)>
```

Unlock

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

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

De DOCTYPE declaration:

```
<!ELEMENT unlock (var)>
```

Value

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

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

De DOCTYPE declaration:

```
<!ELEMENT value (#PCDATA)>
```

Set blok

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

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

```
<setVar>
       <var name="name variable" />
       <var name="name variable 2" />
</setVar>
```

De DOCTYPE declaration:

```
<!ELEMENT setVar (var, value|var|strlen|concat|
                    logicOpp|unOpp|binOpp|random|charAt|arith)>
```

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>
```

B.2 Motion

Move blok

De `move` blok stelt een translatie voor.

```
<move xChange="3" yChange="5" />
```

De DOCTYPE declaration:

```
<!ELEMENT move EMPTY>
<!ATTLIST move xChange CDATA #IMPLIED yChange CDATA #IMPLIED>
```

B.3 Visual

Show

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

```
<show />
```

De DOCTYPE declaration:

```
<!ELEMENT show EMPTY>
```

Hide

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

```
<hide />
```

De DOCTYPE declaration:

```
<!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 changeAppearance EMPTY>
<!ATTLIST changeAppearance id CDATA #REQUIRED>
```

B.4 String operators

Concat

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

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

De DOCTYPE declaration:

```
<!ELEMENT concat (value|var|concat, value|var|concat)>
```

Length

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

```
<strlen>
       <var name="string">
<strlen>
```

De DOCTYPE declaration:

```
<!ELEMENT strlen (value|var|concat)>
```

CharAt

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

```
<charAt>
       <value> index </value>
       <var name="string"/>
<strlen>
```

De DOCTYPE declaration:

```
<!ELEMENT charAt (var|value, value|var|concat)>
```

B.5 Operator

De `operator` blok stelt een operator voor zoals +,-,/,*,<,>,... .

```
<operator name="+" />
```

De DOCTYPE declaration:

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

B.6 Logic operators

Unaire logic operator

De `unLogicOpp` blok stelt een logische operatie voor die unair is.

```
<unLogicOpp>
       <var name="varName"/>
       <operator name="not"/>
</unLogicOpp>
```

De DOCTYPE declaration:

```
<!ELEMENT unLogicOpp (var|value|unLogicOpp|binLogicOpp|arith,
    operator)>
```

Binaire logic operator

De `binLogicOpp` blok stelt een logische operatie voor die binair is.

```
<binLogicOpp>
       <var name="varName"/>
       <operator name="and"/>
       <var name="varName"/>
</binLogicOpp>
```

De DOCTYPE declaration:

```
<!ELEMENT binLogicOpp (var|value|unLogicOpp|binLogicOpp|arith,
    operator,
                      var|value|unLogicOpp|binLogicOpp|arith)>
```

B.7 Arithmetic blokken

Random

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

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

De DOCTYPE declaration:

```
<!ELEMENT random (var|value|arith,
                      var|value|arith)>
```

Arith blok

Een rekenkundige expressie blok berekent een expressie en geeft een number value terug.

```
<arith>
  <var name="varName"/>
  <operator name="+" />
  <var name="varName"/>
</arith>
```

De DOCTYPE declaration:

```
<!ELEMENT arith (var|value|arith,operator,var|value|arith)>
```

B.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>
```

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>
```

Function

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

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

De DOCTYPE declaration:

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

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:

```
<!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">
  <var name="varName1"/>
  <var name="varName2"/>
</functionCall>
```

De DOCTYPE declaration:

```
<!ELEMENT functionCall (var)*>
<!ATTLIST functionCall name CDATA #REQUIRED>
```

B.9 Block

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

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

De DOCTYPE declaration:

```
<!ELEMENT block
    (makeVar|setVar|move|show|hide|changeAppearance|if|if-else|wait|
                      repeat|forever|emit|while|functionCall)*>
<!ATTLIST block name CDATA #REQUIRED>
```

B.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>
```

Events

De `events` blok is een collectie voor alle functions 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)*>
```

Class

De `class` blok stelt volledige class voor. Deze list al zijn input events, alle verschillende emits, alle member variabelen, alle handler functions 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>
</class>
```

De DOCTYPE declaration:

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

B.11 Control Blokken

Forever block

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

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

De DOCTYPE declaration:

```
<!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:

```
<!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:

```
<!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|unLogicOpp|binLogicOpp)>
```

While blok

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

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

De DOCTYPE declaration:

```
<!ELEMENT while (cond,code)>
```

B.12 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 en de volgorde komt overeen met de volgorde van de members van het event.

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

De DOCTYPE declaration:

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

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 een event en de naam van de member die men wilt aanspreken. Deze geeft deze variable zijn value terug.

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

De DOCTYPE declaration:

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

B.13 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>
       <instance name="instance1" class="className" x="1" y="1"/>
       <instance name="instance2" class="className2" x="1"
           y="1"/>
       <wire from="instance1" to="intance2" event="eventName"/>
</wireFrame>
```

De DOCTYPE declaration:

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