

# 1 Inleiding

## 1.1 Situering

MICAS (Microelectronics and Sensors) is een afdeling van het departement Elektrotechniek (ESAT) van de Katholieke Universiteit Leuven. De afdeling geniet een wereldwijde internationale erkenning als onderzoeksinstituut. MICAS houdt zich o.a. bezig met onderzoek over geïntegreerde schakelingen en *sensor design* op het hoogst mogelijke wetenschappelijke niveau. Tevens vervult MICAS een onderwijsende taak.

Eén van de lopende projecten is “IR 13: ingenieurs van 13 jaar”. Hierbij wil men leerlingen van de eerste graad middelbaar onderwijs ingenieursvaardigheden aanleren. Tijdens de lessen TO zullen de leerlingen leren omgaan met technische problemen, aangepast aan hun niveau en ervaring, zoals een ingenieur. Hiervoor werd lesmateriaal ontworpen op basis van een legotrein. De trein kan dan aan de hand van logische modules (logische poorten, flip-flops, *timers*) en sensors beïnvloed worden.

## **1.2 Globale probleemstelling**

In de lessen technologische opvoeding van de eerste graad in het secundair onderwijs leren de leerlingen elektronica. Dit gebeurt met behulp van logische panelen zoals AND-, OR- en NOT-poorten. Volgens ESAT zijn deze lessen niet voldoende om ingenieursvaardigheden aan te leren. Daarom willen ze in samenwerking met RVO-Society, LEGO en het Actieplan Wetenschapsinformatie en –Innovatie een nieuw lessenpakket ontwikkelen: IR13, ingenieur vanaf 13.

Het project wil leerlingen van 13 jaar leren omgaan met technische problemen aangepast aan hun niveau en de leerlingen warm maken voor basiselektronica en technologie. Het pakket bestaat uit een legotrein die kan worden aangestuurd met logische modules zoals EN- en OF-poorten, flip-flops, *timers* en sensoren. Zo kan de trein versnellen en vertragen, stoppen bij obstakels, toeteren bij een seinhuis en de lichten aanzetten als het donker is. Op een wagon van het treintje bevindt zich een moederbord waarmee de leerlingen modules kunnen verbinden.

De leerlingen krijgen een cursus met verschillende opdrachten die ze eerst op papier oplossen en daarna met het treintje kunnen uittesten. Zo leren de kinderen als ingenieurs ontwerpen, modelleren, testen en methodisch denken. De stap die ontbreekt om de ontwerpmethode zo realistisch mogelijk te maken, is een simulator waar de modules ingebracht kunnen worden. De leerlingen kunnen dan met behulp van een grafische gebruikersinterface de modules oproepen en verbinden. Hierdoor kunnen ze hun ontwerp simuleren en de fouten eruit halen alvorens het te testen met het treintje. Op deze manier doorlopen de leerlingen alle fases van een ontwerpproces en leren ze denken als ingenieurs.

### 1.3 Doelstellingen

Na afloop wil de opdrachtgever beschikken over een gebruiksvriendelijk eindpakket voor het simuleren van de logische trein. Aangezien we werken met een eigen ontwikkeling, kan deze volledig aangepast worden aan de wensen en eisen van de opdrachtgever.

Een eerste parameter waarmee we rekening moeten houden, is dat de eindgebruikers kinderen zijn van ongeveer dertien jaar. Dit betekent dat de bediening van het programma zeer eenvoudig moet zijn. Concreet houdt dit in dat alles zeer duidelijk en makkelijk herkenbaar weergegeven moet worden. Zo zullen we bijvoorbeeld dezelfde afbeeldingen en namen voor de modules gebruiken als deze in de cursus. Om de bediening eenvoudig te houden, zal alles grafisch weergegeven worden. Tevens zal de bediening zoveel mogelijk met de muis kunnen gerealiseerd worden.

Daarnaast dient het programma om enkele mogelijke fouten in het ontwerp te ontdekken. Het is echter onmogelijk te verzekeren dat de gemaakte schakeling volledig correct is opgebouwd. We kunnen er wel voor zorgen dat enkele vaak gemaakte fouten gedetecteerd kunnen worden. Zo mogen de leerlingen bijvoorbeeld nooit twee uitgangen met elkaar verbinden. Door het detecteren en het signaleren van deze fouten, maken we het programma in zekere mate *'foolproof'*.

Anderzijds zullen we ervoor moeten zorgen dat er achteraf op een eenvoudige wijze aanpassingen kunnen doorgevoerd worden. De opdrachtgever wil in het bijzonder snel en eenvoudig nieuwe modules kunnen toevoegen. Bovendien zal de verwerking van een ingang voor iedere module beschreven moeten zijn met simpele instructies.

Om tot een goed eindresultaat te komen, zullen we dus rekening moeten houden met zowel de eindgebruiker als de opdrachtgever. Aangezien de eindgebruikers kinderen zijn, wordt het programma voorzien van een duidelijke, grafische structuur en een eenvoudige bediening. Anderzijds zal de achterliggende logica simpel worden gehouden. Hierdoor kunnen er snel en zonder al te veel moeite aanpassingen aangebracht worden.

## 1.4 Methode

Voor een goed ontwerp is het noodzakelijk dat de ontwikkelaar een grondige analyse en studie doorvoert alvorens het eigenlijke programmeren. De ontwikkeling van softwareapplicaties verloopt dan meestal ook in verscheidene stappen. In een eerste fase wordt het probleemdomain vastgesteld. Dus voor het eigenlijke programmeren, voeren we in samenspraak met de opdrachtgever een grondige studie en analyse door. Hieruit volgt in welke omstandigheden het programma moet functioneren en welke mogelijkheden geïmplementeerd moeten zijn.

Na deze fase zijn er duidelijke afspraken zoals welke programmeertaal, programma's of bibliotheken er gebruikt zullen worden bij het programmeren. In de verdere ontwikkeling zullen we geen conflicten meer mogen aantreffen. Indien dit toch zo is, zal het probleemdomain voor een deel opnieuw geanalyseerd moeten worden.

In een volgende fase wordt de totale opdracht opgesplitst in meerdere kleinere deelproblemen. De masterproef beschrijft elk deelprobleem en zet het dan om naar code. Dit geeft ons de mogelijkheid om elk aan een apart deelprobleem te werken, wat de ontwikkelsnelheid ten goede komt. Elk van deze deelproblemen heeft een testbaar geheel als eindresultaat.

Een volgende stap bestaat dan ook uit het testen van het deelprobleem. Voldoet het aan alle regels? Is het gemakkelijk in gebruik? Loopt het niet vast? Indien het programma hieraan voldoet, kunnen we verder met de ontwikkeling van andere deelproblemen. Zoniet dienen we de nodige aanpassingen door te voeren en opnieuw de testfase te doorlopen.

Zo krijgen we een steeds groeiende applicatie die meer en meer naar het eindresultaat evolueert. Deze methode zorgt voor een zekerheid dat alle deelproblemen op zich werken. Uiteindelijk zal het systeem in zijn geheel nog uitgebreid getest moeten worden met alle functionaliteiten die gelijktijdig kunnen lopen. Op deze manier ontstaat stapsgewijs een robuust systeem dat aan de wensen van de opdrachtgever voldoet. Eenmaal een werkend programma is bekomen, kunnen eventueel nog handige functies toegevoegd worden. Dit zorgt voor een betere gebruiksvriendelijkheid of een mooiere lay-out voor het programma.

Bij de eigenlijke ontwikkeling van een softwarepakket van zulke aard moeten een aantal beslissingen genomen worden wat betreft gebruikte technieken en technologieën. We moeten deze dus nader bekijken en afwegen tegen elkaar. Omwille van het financiële aspect kunnen we echter alle commerciële pakketten al meteen elimineren. Onderstaande opsomming geeft een overzicht op welke deelaspecten van het project we een duidelijke keuze hebben gemaakt:

- Programmeertaal (Java),
- *Editor kit* (Piccolo),
- In- en uitlezen definities (Xstream),
- *Logger* bibliotheek (Sun).

## 2 Ontwerptechnieken

### 2.1 Inleiding

Bij het uitwerken van onze masterproef zijn er verscheidene bestaande technologieën aan bod gekomen. In dit hoofdstuk zullen we de werking van deze technologieën uitleggen en hoe we ze hebben gecombineerd met onze masterproef. Op deze manier is het eenvoudiger een goed beeld te krijgen van de werking van het programma.

We starten met een onderzoek naar de mogelijke programmeertalen. Hierop volgend is er een woordje uitleg over de manier waarop we het programma kenbaar maken naar de buitenwereld toe. Vervolgens wijden we uit over de gebruikte software. Om af te sluiten gaan we dieper in op de externe bibliotheken. Dankzij deze externe bibliotheken is het mogelijk om het programmeren te vereenvoudigen en bovendien te versnellen.

### 2.2 Programmeertaal

Bij de start van onze masterproef moet er al een zeer ingrijpende beslissing worden genomen. Namelijk met welke programmeertaal wordt het programma geschreven? De programmeertalen kunnen onderverdeeld worden in 3 niveaus.

Een eerste mogelijkheid bestaat erin te schrijven met een zogenaamd *low-level* programmeertaal, de machinetaal. Dit is de enige taal die de CPU begrijpt. Hierbij worden alle instructies binair geschreven. Deze manier van werken komt niet in aanmerking voor onze masterproef. Eén level hoger dan de machinetaal hebben we de assembleertaal. In deze vorm worden de machine-instructies symbolisch opgeschreven. Er is echter wel nog een *assembler* nodig om de code te vertalen voor de CPU. Hoewel het al makkelijker te schrijven is dan de machinetaal, is de assembleertaal toch niet geschikt voor het schrijven van complexe programma's.

Op het hoogste niveau hebben we de *high-level* programmeertalen. Deze zijn het meest geschikt bij het schrijven van programma's, dus ook voor onze masterproef. Voor het uitvoeren van een *high-level* programmeertaal, zal de computer de code eerst opnieuw moeten vertalen. Hiervoor maken we gebruik van een *compiler*. In deze groep kunnen we de talen nog verder opsplitsen als procedurele talen en objectgeoriënteerde talen. Bij procedurele talen zullen de programma-instructies worden uitgevoerd in de volgorde zoals ze zijn geprogrammeerd. Een meer gebruikt concept is deze van het objectgeoriënteerd programmeren. Hierbij wordt de software opgebouwd als een verzameling van samenwerkende objecten. Zulk een object bestaat uit data en methodes om deze data te veranderen. Doordat de data enkel wordt aangepast via de methodes van een object, verkrijgen we een vorm van afscherming en beveiliging van de data. Bovendien kan de

interne werking van een object eenvoudig worden aangepast, zonder dat de omliggende programmastructuur dit merkt.

Voor projecten van grotere omvang, zoals deze masterproef, verkiest men een *high-level* programmeertaal. Er zal om precies te zijn, een objectgeoriënteerde taal verkozen worden bij zulke projecten.

## 2.2.1 High-level programmeertalen

Dat we het programma zullen schrijven met een high-level programmeertaal was al meteen duidelijk. Maar in deze groep vinden we tientallen mogelijke talen terug. Een eerste opgave was dan ook om in al de verschillende mogelijkheden een geschikte programmeertaal te vinden. Om dit te verwezenlijken hebben we een tiental mogelijke programmeertalen onderzocht. Om niet te ver af te dwalen zullen we enkel de vier grootste kanshebbers verder in detail uitwerken. Andere talen die werden onderzocht, maar niet verder worden uitgelegd zijn o.a. Pascal, C, Smalltalk, Visual Basic, Ruby, enz... .

Vooraleer we verder gaan, zullen we eerst de term *garbage collection* uitleggen. Deze term heeft een belangrijke rol gespeeld bij de keuze van een programmeertaal. Objectgeoriënteerde talen maken bij het uitvoeren van een programma gebruik van objecten. Deze objecten staan in de geheugenruimte die voor het programma is voorzien. Wanneer de objecten niet meer actief zijn, dus niet meer gebruikt worden, kan het hiervoor toegekende geheugen terug vrij gegeven worden. Deze geheugenruimte is dan weer beschikbaar voor andere objecten of toepassingen van het programma. De *garbage collector* zorgt er dus voor dat onnodig toegekende ruimte wordt vrijgegeven.

De verder uitgewerkte talen zijn allemaal objectgeoriënteerd en bezitten een automatische *garbage collector*. De enige uitzondering op deze regel is C++. Hier moet de *garbage collecting* manueel gebeuren, maar kan door gebruik van de juiste bibliotheek ook automatisch gebeuren.

### 2.2.1.1 Visual Basic .NET

Visual Basic .NET, of kortweg VB.NET, is de opvolger van Visual Basic. Het verschil met Visual Basic is dat VB.NET is gebaseerd op de Microsoft .NET Framework. Dit .NET Framework is een deel van het Microsoft Windows besturingssysteem. Het voorziet voorgeprogrammeerde oplossingen voor algemene programmavereisten.

Een groot nadeel van VB.NET is een gebrek aan voorkennis. Hoewel het een vrij eenvoudige programmeertaal betreft, zullen we toch eerst de taal moeten aanleren. Met behulp van onze reeds parate kennis van C++ en Java zou dit evenwel niet voor problemen mogen zorgen. Verder zijn de kosten die verbonden zijn met het programmeren in VB.NET nadelig. In vergelijking met andere talen, brengt VB.NET een relatief dure ontwikkelomgeving met zich mee.

### 2.2.1.2 C++

C++ wordt ook wel een *mid-level* programmeertaal genoemd, omdat het zowel *high-level* als enkele *low-level* mogelijkheden combineert. C++ is een zeer krachtige, maar een vrij complexe programmeertaal. Het geheugen kan zelf beheerd worden, *real-time* programmering is mogelijk, gebruik van pointers, enz... . Dit alles zorgt ervoor dat er krachtige programma's mee geschreven kunnen worden. Deze uitgebreide mogelijkheden zullen er echter ook voor zorgen dat het schrijven relatief bemoeilijkt wordt. Aangezien deze verregaande opties niet bruikbaar zijn voor deze masterproef, kunnen we ons beperken tot een eenvoudigere programmeertaal.

Een pluspunt van deze taal is dat programma's meestal sneller zijn in uitvoering. Hoewel dit verschil tegenwoordig steeds kleiner en kleiner wordt. Langs de andere kant, zal de code veel moeilijker te debuggen zijn. Bovendien is het programma platformafhankelijk. Dit wil zeggen dat eenzelfde code niet zonder problemen kan worden gebruikt op verschillende besturingssystemen zoals Windows, Linux of Mac OS. Ten slotte kunnen we ook nog opmerken dat met C++ het moeilijker is om alles grafisch zichtbaar te maken.

### 2.2.1.3 C#

C#, uitgesproken als C-sharp, is gebaseerd op zowel C++ als Java en de .NET Framework van Microsoft. Hierdoor verkrijgen we een programmeertaal die gemakkelijk begrijpbaar is en relatief eenvoudig aan te leren. Toch blijft het krachtig genoeg om relatief complexe problemen op te lossen. Bovendien biedt C# nog enkele extra mogelijkheden die niet in C++ of Java zijn geïntegreerd. Deze taal verleent zich daarom perfect als vertrekpunt voor onze masterproef. Het enige echte nadeel is dat we nog geen enkele ervaring hebben met het programmeren in C#.

#### 2.2.1.4 Java

Bij de ontwikkeling van Java baseerde men zich op de talen C en C++. De grootste verschillen met deze programmeertalen is dat Java een eenvoudiger objectmodel heeft en minder *low-level* mogelijkheden. Bovendien wordt Java niet gecompileerd naar machinecode, specifiek voor een bepaalde computerarchitectuur. Na een compilatie zal er echter een bytecode verkregen worden. Deze bytecode kan dan draaien op een zogenaamde JVM (Java Virtual Machine) ongeacht de computerarchitectuur. Dankzij deze mogelijkheid kunnen we met eenzelfde broncode zorgen dat het programma op ieder platform kan werken. Hierdoor zijn de scholen vrij om te kiezen met welk besturingssysteem ze willen werken.

Andere voordelen die we bekomen door te werken met Java zijn o.a. de automatische *garbage collector*, de herbruikbaarheid van de code, enz. . Verder is voor onze masterproef ook de mogelijkheid van *multithreading* nuttig. Hiermee kunnen meerdere processen tegelijkertijd worden uitgevoerd. Aangezien Java een wereldwijd veelgebruikte taal is, zijn er talloze bibliotheken beschikbaar die het programmeren vergemakkelijken.

Programma's in Java hebben anderzijds de eigenschap dat ze meer geheugen verbruiken. Mede hierdoor zal het programma over het algemeen langzamer zijn dan bij de meeste andere programmeertalen. Een ander minpunt is dat bij het gebruik van Java een JRE (Java Runtime Environment) geïnstalleerd moet zijn. Al deze negatieve eigenschappen vormen echter geen bijkomende problemen voor onze masterproef.

#### 2.2.1.5 Conclusie

Wanneer we alle voor- en nadelen van de verschillende programmeertalen naast elkaar zetten, kunnen we concluderen dat er voor onze masterproef drie talen voldoen. Zowel C++, C# als Java verlenen zich goed om een simulator uit te werken. Als eerste elimineren we C++, vanwege een persoonlijke voorkeur voor de andere kandidaten. Onze uiteindelijke keuze is dan toch gevallen om verder te werken met Java. De grotere voorkennis ten opzichte van C# is de voornaamste reden die heeft gezorgd voor de doorslag.



## 2.3 Programmavoorstelling

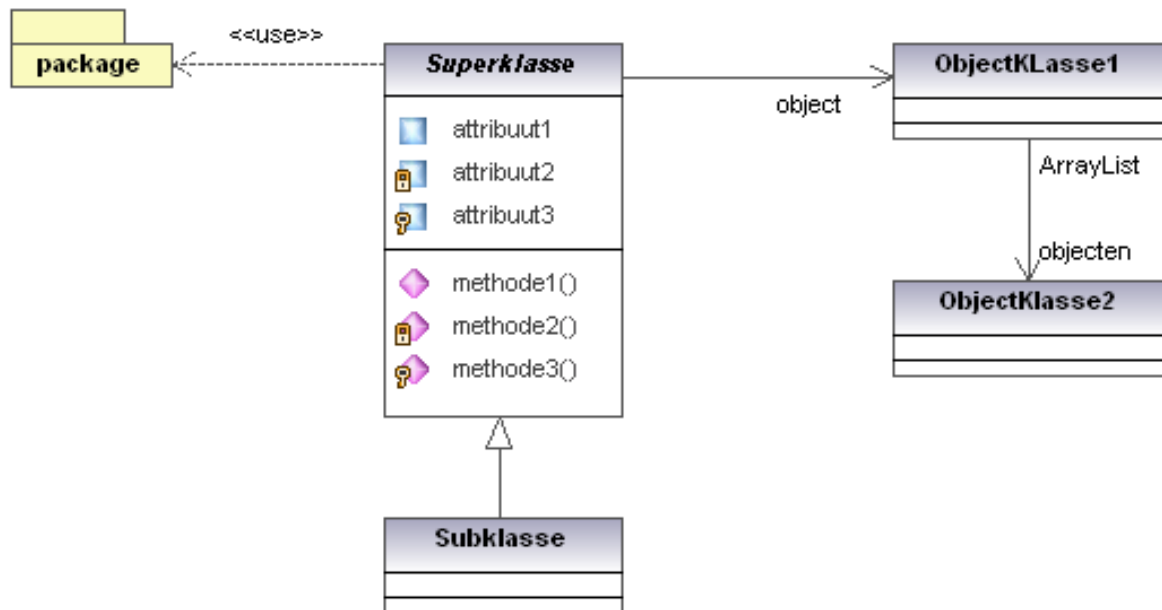
### 2.3.1 UML

UML staat voor Unified Modelling Language. Het betreft een gestandaardiseerde, visuele taal voor het weergeven van objectmodellen. Het resultaat is een grafische weergave van het systeem, of eventueel slechts een deel ervan. Hiermee wordt het geheel voorgesteld op een begrijpbare wijze. Op deze manier kan men zich een beeld vormen van de interne structuur van het programma. Het voordeel is dat er geen kennis nodig is van de interne werking van de programmacode. Zelfs mensen zonder ervaring met de programmeertaal kunnen een duidelijk beeld krijgen van de werking van het programma.

Figuur 1 is een eenvoudig voorbeeld hoe een programma grafisch kan worden voorgesteld. Elke klasse wordt weergegeven door middel van een rechthoek. Wanneer een klasse abstract is, zal de naam van de klasse cursief geschreven zijn. Dit is het geval voor Superklasse in figuur 1. Een abstracte klasse is een klasse waarvan men geen objecten kan aanmaken. De overerving van een klasse wordt aangegeven door een pijl met een driehoek als pijlpunt. In figuur 1 erft de klasse Subklasse over van Superklasse. Bij overerving worden methodes en attributen van de superklasse beschikbaar voor alle subklassen.

Elke klasse van het model kan voorzien worden van attributen en methodes. Voor elk attribuut en methode staat een symbool om zijn toegankelijkheid weer te geven. De klasse Superklasse heeft 3 attributen en 3 methodes. Attribuut1 en methode1() zijn beide public, dit wil zeggen toegankelijk vanuit elke plaats in de code. Attribuut2 en methode2() daarentegen zijn *private*, enkel de code van deze Superklasse kan hiervan gebruik maken. Ten slotte kunnen attributen en methodes ook *protected* zijn. In dit geval is het attribuut of de methode enkel bereikbaar voor de code van de klasse en voor elke subklasse die ervan overerft. In figuur 1 is dit zo voor attribuut3 en methode3().

Om aan te geven dat een klasse gebruik maakt van een andere klasse of *package* zal bij de pijl de vermelding <<use>> staan. Anderzijds is het ook mogelijk dat een bepaalde klasse een object in bezit heeft van een andere klasse. Dit wordt aangeduid met een pijl van de klasse die het object bezit naar de andere klasse. In figuur 1 is de Superklasse in bezit van een object van de ObjectKlasse1 met als naam object. Wanneer aan het begin van de pijl de vermelding ArrayList staat, wil dit zeggen dat de klasse een ArrayList van objecten bezit.



**Figuur 1: Voorbeeld van een UML**

### 2.3.2 API

API is de afkorting van *Application Programming Interface*. Dit is een verzameling definities op basis waarvan een computerprogramma kan communiceren met een ander programma of onderdeel. Een API definieert de toegang tot de functionaliteit die er achter schuil gaat. De buitenwereld kent geen details van de functionaliteit of implementatie, maar weet dankzij de API wel hoe deze kan worden aangesproken. Zo kan een programma bijvoorbeeld gebruik maken van een API voor te printen zonder te moeten weten hoe het printen werkt. Het grote voordeel van een API is dan ook dat de programmeur de details van de API niet hoeft te kennen. Hierdoor zijn er meerdere implementaties mogelijk zolang deze voldoen aan de API.

De meest gebruikte API in onze masterproef is de API van Java zelf. Deze is uitgebreid gedocumenteerd en terug te vinden op de website van Sun. Verder maken we ook gebruik van de API's van Piccolo en Xstream.

### 2.3.3 Javadoc

Javadoc is een *document generator* van Sun Microsystems. Javadoc genereert op een eenvoudige wijze documentatie in het HTML-formaat vanuit de Java code. In de Java code is het mogelijk commentaar voor een klasse, variabele of methode te voorzien met als beginteken `/**` en als eindteken `*/`. Javadoc neemt de tekst tussen deze tekens op als commentaar en zet ze als uitleg bij de bijhorende klasse, variabele of methode in de documentatie. Het is ook mogelijk parameters zoals auteur, methodeparameter of teruggegeven waarde van een methode, enz. ... in de commentaar mee te geven. Javadoc creëert zo een overzichtelijke documentatie van de verschillende klassen met hun verschillende variabelen en methodes en uitleg bij hun werking. Hierdoor is het makkelijker het programma te begrijpen zonder te veel de code te moeten onderzoeken.

## 2.4 Ontwikkelomgeving

Er zijn talrijke programma's beschikbaar die bruikbaar zijn om javaprogramma's te schrijven. Enkele voorbeelden zijn:

- Eclipse
- Netbeans
- Jcreator
- Kawa
- jEdit
- ...

Programma's zoals Eclipse en Netbeans zijn veel uitgebreider dan meer eenvoudiger editors, bijvoorbeeld Kawa. Door middel van de extra mogelijkheden van de eerstgenoemde programma's is het eenvoudiger een overzicht te houden over het project. Bovendien zijn er allerlei functies voorzien om het compileren en debuggen gemakkelijker te laten verlopen. Deze IDE's (Integrated Development Environment) zijn meestal ook voorzien van code voltooiing of *code completion*, wat het programmeren enigszins vereenvoudigt. Al deze extra mogelijkheden zorgen er wel voor dat de software veel zwaarder is.

Een andere extra mogelijkheid die Eclipse en Netbeans biedt, is het eenvoudig aanmaken van een GUI (Graphical User Interface). We kunnen hierbij de nodige componenten slepen en plaatsen in een grafische omgeving. Het programma zal dan zelf de nodige code genereren. Deze werkwijze lijkt misschien eenvoudiger en sneller dan het traditioneel zelf uitschrijven van de GUI. Toch opteren we om zelf de code te schrijven. Dit zorgt ervoor dat we zeker zijn van een zo compact mogelijke code en is het eenvoudiger om wijzigingen aan te brengen.

## **2.5 Subversion**

Met het gebruik van Subversion kan er gemakkelijker met meerdere personen en op verschillende plaatsen aan eenzelfde project worden gewerkt. Het betreft een versiebeheersysteem. Hiermee staan al de bestanden op een centrale plaats, in ons geval een server op de KHLim. Eclipse heeft dan ook nog het grote voordeel dat subversion via een plugin (nl. Subclipse) kan geïntegreerd worden.

Wanneer iemand wil verder werken aan het project, moet hij een 'checkout' doen. Dit houdt in dat de gegevens die op de server staan, gekopieerd worden naar zijn lokale harde schijf. Nu kunnen wijzigingen aan de data doorgevoerd worden. Als alle nodige aanpassingen zijn gebeurd, moet er een 'commit' gedaan worden. Met deze commit worden de veranderingen opnieuw opgeslagen op de centrale ruimte, de server.

Zowel bij een 'commit' als een 'checkout' worden enkel de gewijzigde gegevens of eventueel nieuwe bestanden doorgestuurd via Internet. Dit zorgt er dus voor dat het internetverkeer beperkt blijft. Aangezien we een internetlimiet hebben op kot, is deze optie een pluspunt. Dankzij deze mogelijkheid ontstaat ook de mogelijkheid om beide tegelijkertijd aan de masterproef te werken. Zolang we niet aan hetzelfde bestand werken, zullen er dan geen conflicten optreden als we opnieuw een 'commit' uitvoeren.

Een andere handige functie van subversion is de mogelijkheid om terug te grijpen naar een oudere versie. Wanneer achteraf blijkt dat verschillende aanpassingen niet goed zijn, wordt er dus gewoon opnieuw een oudere versie van de server afgehaald. Dit zorgt voor een aanzienlijke vereenvoudiging voor het debuggen van het programma.

## 2.6 Editorkit

Het gebruik van een *editorkit* vereenvoudigt het maken van grafische applicaties in Java. Een *editorkit* is een bibliotheek waarin verschillende klassen voorzien zijn voor grafische toepassingen aan te maken. Zo bespaart een *editorkit* heel wat programmeerwerk. Er zijn verschillende *editorkits* voorhanden zoals JHotDraw, GraphViz, Jung, Grappa, Piccolo, InfoVis, enz... In de volgende paragrafen bespreken we kort de meest geschikte *editorkits* voor onze GUI.

### 2.6.1 JHotDraw

JHotDraw is een *framework* voor grafische toepassingen aan te maken. Het ondersteunt zowel geometrische als door de gebruiker aangemaakte vormen. Hiervoor bevat JHotDraw een collectie van verschillende vormen die elk apart gewijzigd kunnen worden. JHotDraw bevat twee verschillende klassen, namelijk de Drawing klasse en de DrawingView klasse. De Drawing klasse is het model van de applicatie terwijl DrawingView het voorkomen van de tekening in de GUI bevat. Hierdoor vergroot de flexibiliteit van JHotDraw. Alle applicaties in JHotDraw bestaan bijgevolg uit een tekenmodel, een tekenzicht en een collectie van tekenvormen samen met een *tool*.

### 2.6.2 Jung

Jung is een bibliotheek voor de modellering, de analyse en de visualisatie van data. Jung ondersteunt verschillende voorstellingen van entiteiten en mechanismen. Zo zijn er onder andere mechanismen aanwezig voor optimalisatie en het aanpassen van diagrammen. Bovendien voorziet Jung ook een visualisatie *framework*. Dit maakt het makkelijk om *tools* aan te maken om netwerkdata interactief te verkennen. Hiervoor kan de gebruiker zelf een eigen *lay-out* aanmaken of gebruik maken van één van de *lay-outs* die door Jung voorzien zijn.

### 2.6.3 Grappa

Grappa is een tekenpakket voor diagrammen dat de weergave en manipulatie van diagrammen vereenvoudigt. Het is bruikbaar bij Java applicaties en Java *applets* en is een onderverzameling van GraphViz. Grappa bevat methoden om tekeningen in te lezen en weg te schrijven met behulp van het dot tekst formaat. Dit is een taal waarin een diagram volledig in tekst beschreven is die zowel computer als mens kunnen gebruiken. Hierdoor biedt Grappa het voordeel dat de gebruiker diagrammen eenvoudig kan beschrijven, maar een nadeel is dat er een bijkomende taal gekend moet zijn.

### 2.6.4 Piccolo

Piccolo is een bibliotheek voor zowel Java als C# voor het ontwikkelen van gestructureerde grafische applicaties met effecten zoals inzoomen en animaties. Piccolo bestaat uit een hiërarchische structuur van objecten en camera's, waardoor het mogelijk is om objecten te groeperen en te manipuleren. Piccolo maakt gebruik van de Java2D API voor de grafische verwerking. Hierdoor moet er minder rekening gehouden worden met de lagere niveaudetails van het programmeren. De belangrijkste klasse in Piccolo is de PNode klasse. De andere klassen zijn afgeleid van deze PNode klasse. Zo is alles wat zichtbaar en waar interactie met de gebruiker aan gekoppeld is, een PNode. Voor deze interacties zijn er in Piccolo verschillende *event listeners* voorzien.

### 2.6.5 Conclusie

Na het bestuderen van de verschillende editorkits en de bijhorende voorbeelden, is gebleken dat de beste keuze Piccolo is. Met Piccolo is het namelijk eenvoudig om PNodes aan elkaar te linken en zo een duidelijke structuur te krijgen. Verschillende PNodes kunnen samen een figuur voorstellen en kunnen elk een andere *event listener* krijgen toegewezen. Dit is handig voor het aanmaken van de modules die nodig zijn om de trein aan te sturen. Zo kan er bijvoorbeeld aan elke in- of uitgang een andere gebeurtenis worden toegewezen. Terwijl de in- of uitgang een onderdeel is van de module waartoe hij behoort. Een ander voordeel van Piccolo is dat alles gemakkelijk geïntegreerd kan worden in Swing, de standaardbibliotheek van Java voor grafische applicaties.

Voor Piccolo is er voldoende documentatie beschikbaar. Zo is de volledige API beschikbaar op de webpagina van Piccolo. Ook staat er een uitgebreide handleiding online, wat het aanleren van de bibliotheek vergemakkelijkt. Bovendien zijn er voldoende voorbeelden beschikbaar gesteld door de ontwikkelaars die de werking van Piccolo verduidelijken.

## 2.7 XML

XML is de afkorting van Extensible Markup Language. Het betreft een formaat om gegevens gestructureerd vast te leggen. XML is in eerste instantie ontworpen om het delen van de gegevens over verschillende informatiesystemen te vereenvoudigen. Het grote voordeel is dat de gestructureerde data behalve voor de machine, ook goed herkenbaar en leesbaar blijft voor de gebruiker. Met behulp van deze XML kunnen we dan verschillende richtingen uit. Het kan enerzijds dienen om de inhoud om te zetten naar een bepaald formaat, bijvoorbeeld HTML, met behulp van een zogenaamde opmaaktaal. Het kan anderzijds ook dienen om de gegevens gemakkelijk en gestructureerd te raadplegen.

Eén van de vereisten van onze opdrachtgever is dat de gegevens over alle mogelijke modules in een extern bestand worden gedefinieerd. XML biedt hiervoor een goede oplossing. Alle gegevens kunnen gestructureerd worden voorgesteld, zodat ook alles duidelijk blijft voor de gebruiker. Dit zorgt ervoor dat achteraf vrij eenvoudig wijzigingen kunnen doorgevoerd worden, zonder dat er een goede kennis nodig is van de programmeertaal. Bovendien moet het programma niet opnieuw gecompileerd worden bij wijzigingen aan de gegevens. Al deze eigenschappen zorgen voor een zeer goed gebruiksgemak. Nu zijn er geen moeilijke ingrepen meer nodig om eventueel modules aan te passen.

Vanuit de XML-bestanden kunnen we op een vrij eenvoudige wijze de gegevens omzetten naar objecten in Java. Zo bekomen we dan objecten die in het programma kunnen gebruikt worden, waarop al de nodige bewerkingen kunnen uitgevoerd worden. Het inlezen van de gegevens in de programmacode, noemt men *parsing*. Om dit uit te voeren zijn er reeds verschillende zogenaamde *XML-parsers* ontworpen. De meest bekende, en ook meest gebruikte *parsers*, zijn de DOM API en de SAX API. We zullen nu verder ingaan op de werking van deze *parsers*.

### 2.7.1 Parsers

Een eerste mogelijkheid om de gegevens vanuit XML-bestanden te *parsen* is door gebruik te maken van de DOM API. Dit is de afkorting voor Document Object Model Application Programming Interface. Met behulp van deze DOM API is het mogelijk om doorheen het volledige XML-bestand te navigeren. Het bestand wordt opgevat als een boomstructuur volgens de aanwezige elementen, waar dan de nodige gegevens worden uitgefilterd. Doordat het volledige document moet ingeladen worden in het geheugen, is deze manier van werken iets tijdrovender.



Een andere veel gebruikte manier om XML te *parsen* is met behulp van de SAX. Dit is de afkorting voor Simple API for XML. Hiermee bekomen we een *event*-gestuurde API voor het inlezen van XML-bestanden. Elk XML-bestand wordt dan serieel ingelezen, waardoor we een sequentiële gegevensstroom verkrijgen. Hierdoor hebben we een snelle en efficiënte manier om de gegevens uit te lezen. Nadelig is echter dat we hierdoor moeilijker willekeurig data uit het XML-bestand kunnen halen.

Ten slotte zijn er ook nog de minder gebruikte methodes *pull parsing* en *data binding*. Bij *pull parsing* wordt het document behandeld als reeks objecten. Het uitlezen gebeurt dan door middel van een iteratief patroon. Terwijl bij *data binding* de gegevens worden aangeboden als een zelf bepaalde gegevensstructuur van een programmeertaal .

## 2.7.2 Xstream

Xstream is een gratis bibliotheek voor Java. Hiermee kunnen we dan op een eenvoudige wijze objecten aanmaken door het inlezen vanuit XML-bestanden. Een andere mogelijkheid van Xstream is het omzetten van een object in Java naar XML. Dit is echter bij onze masterproef niet van toepassing. Om deze functionaliteiten te realiseren maakt Xstream gebruik van XPP3 (XML Pull Parser 3rd edition). Bij het converteren tussen objecten en XML wordt dus gebruik gemaakt van een *parser* gebaseerd op de *pull parsing* technologie.

Dankzij Xstream kunnen we op een snelle en eenvoudige manier al de nodige gegevens van een module uitlezen vanuit een XML-bestand. Bovendien kan met deze bibliotheek eenvoudig worden overgeschakeld naar één van de andere mogelijke *XML-parser*.

## 2.8 *Logger bibliotheek*

Een *logger* is een manier om de status van een programma of bepaalde gebeurtenissen in een bestand bij te houden. Zo kan een *logger* een hulpmiddel zijn om het *debuggen* van een applicatie te vereenvoudigen. Tevens kan de juiste werking van een applicatie zo nagegaan worden. Hierdoor is het eenvoudiger om de juiste werking van de simulatie in ons programma na te gaan.

### 2.8.1 Java Logging

In de standaardbibliotheek van Java zijn klassen voorzien om te *loggen*. Deze *logger* schrijft de gebeurtenissen naar een LogRecord object. Een filter haalt dan de berichten uit dit object aan de hand van een vooropgesteld filterniveau. De filter stuurt deze gefilterde berichten dan door naar een Handler-object. Met dit Handler-object is het mogelijk de berichten via een Formatter naar het gewenste formaat om te zetten. De *logberichten* kunnen zo naar een bestand weggeschreven worden als tekst of als XML-formaat. Het is eveneens mogelijk om zelf een Handler en Formatter te schrijven indien de standaardklassen van Java niet voldoen aan de vereisten.

### 2.8.2 Log4j

Log4j is een *logger* van de Apache Software Foundation. Log4j kan de uitvoer van de applicatie bijhouden in verschillende formaten zoals onder andere HTML, XML en gewone tekst. Log4j bestaat uit 3 hoofdcomponenten: de *logger* zelf, de *appender* en de *lay-out*. De *logger* houdt de *logberichten* bij en schrijft ze dan weg naar een *appender interface*. Net zoals bij de *logger* van Java is het bij Log4j mogelijk om een niveau in te stellen bij ieder bericht. De *lay-out* bepaalt dan het formaat van het bestand waarin deze berichten weergegeven worden. Ook hier zijn verschillende formaten beschikbaar of kan de gebruiker zelf een *lay-out* maken.

### 2.8.3 Conclusie

Java Logging en Log4j tonen veel gelijkenissen in zowel gebruik als structuur. Log4j bestaat al langer dan Java Logging en is open source. Dit betekent dat iedere gebruiker de code kan bekijken en bewerken. Java Logging is echter reeds voldoende uitgewerkt en wordt door Sun zelf ondersteund. Omdat de functionaliteit van Java Logging voldoende is voor deze masterproef, hebben we besloten deze logger te gebruiken. Op deze manier hoeft er geen extra bibliotheek toegevoegd worden.

## 3 Systeemontwerp

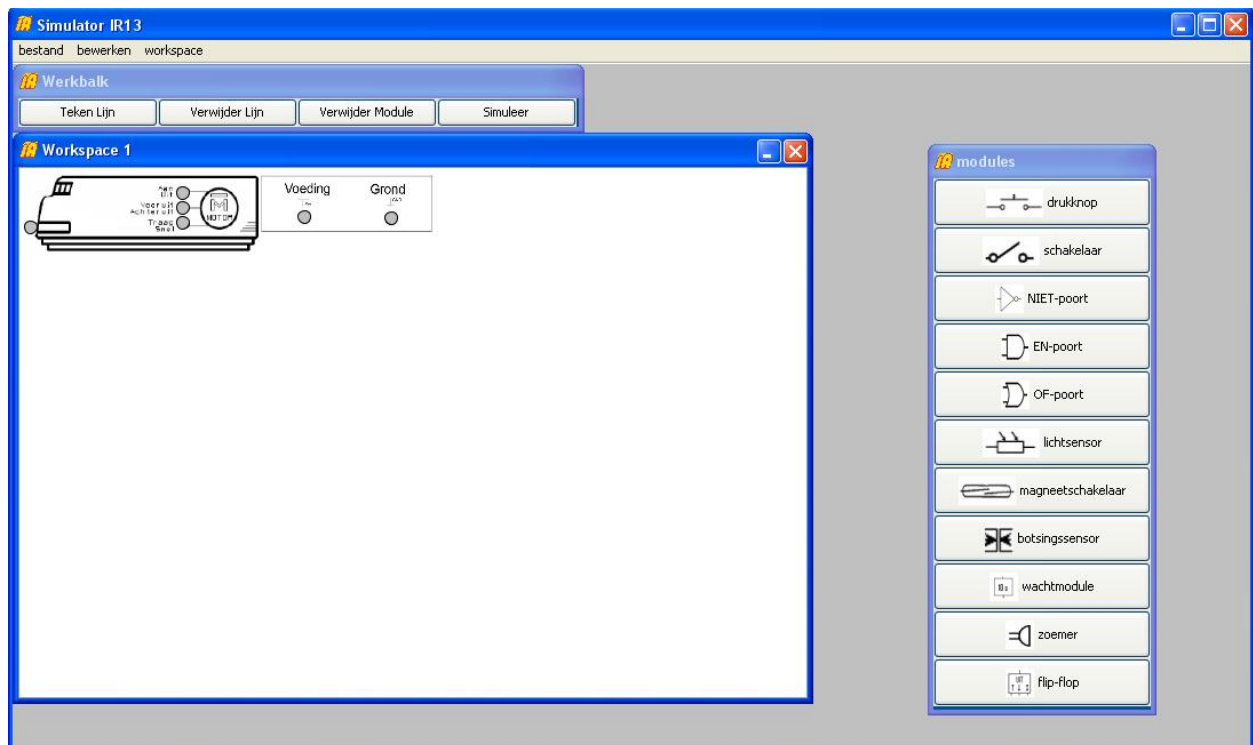
### 3.1 Inleiding

In dit hoofdstuk zullen we het eindresultaat behandelen. Zowel de werking als de gebruikte structuur van het programma zal aan bod komen. Om de tekst niet onnodig ingewikkeld te maken, zal de tekst niet te ver in detail treden. Bij de bespreking van de werking zullen dus niet alle methodes van het programma worden uitgediept. Er zal enkel worden uitgelegd wat het programma kan. De gebruikte algoritmes en strategieën worden echter wel verduidelijkt. Verder schetsen we een algemeen beeld van de structuur voor een zicht te krijgen op het programma.

Bij de start van onze masterproef is het programma opgedeeld in twee aparte gedeelten. Enerzijds worden de beschikbare modules opgebouwd aan de hand van de XML-bestanden. En anderzijds hebben we het venster waar de schakeling kan getekend worden. Deze delen van het programma zijn vervolgens samengevoegd. Als derde groot deel van het programma is er de simulatie zelf. In volgende paragrafen worden deze delen verder in detail besproken.

### 3.2 Hoofdvenster

Figuur geeft een voorbeeld van het hoofdvenster van het programma. In het hoofdvenster zijn er 3 interne vensters aanwezig. Deze zijn de workspace, de werkbalk en het selectievenster voor de modules. Bij de start van het programma opent geen nieuwe workspace. De gebruiker kan echter een nieuwe workspace openen via het menu Bestand of door de snelkoppeling Ctrl+N. De werkbalk en het selectievenster staan altijd op de voorgrond. De knoppen van de werkbalk zijn ook allemaal in de menubalk voorzien. Voor deze knoppen kan de gebruiker ook een snelkoppeling gebruiken.

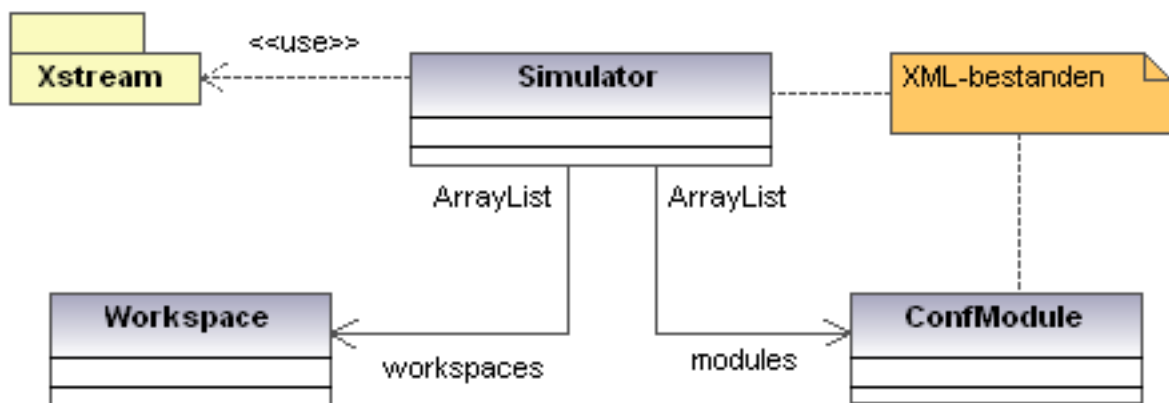


**Figuur 2: Het hoofdvenster**

### 3.3 Modules inlezen

#### 3.3.1 Programma-architectuur

Figuur 3 geeft een UML-schema over de manier waarop de configuratiemodules worden aangemaakt. De Simulatorklasse leest de informatie uit de XML-bestanden door gebruik te maken van de bibliotheek Xstream. Aan de hand van de gegevens uit de XML-bestanden worden configuratieobjecten aangemaakt. De zo verkregen objecten komen in de Simulatorklasse terecht in een ArrayList. Vervolgens wordt in de code van de Simulatorklasse een lijst opgesteld en getoond op het scherm, zodat een selectie van de modules mogelijk is. Op deze manier bekomen we een basis voor de verdere werking van het programma.



Figuur 3: UML voor het inlezen vanuit XML

#### 3.3.2 Inlezen

Bij het opstarten van de simulator dient er een controle plaats te vinden. Hierbij zal het programma controleren welke modules ter beschikking zijn voor het maken van een schakeling. De bestaande modules worden beschreven in een extern XML-bestand. Voor een goede werking te verzekeren, dienen enkele eigenschappen van de modules gekend te zijn in de programmacode. Deze nodige data is gestructureerd terug te vinden in de XML-code van de betreffende module. Deze architectuur zorgt voor een geheel waarin eenvoudig aanpassingen kunnen doorgevoerd worden.

Vooraleer de beschikbare modules ter beschikking zijn in de code, moeten de gegevens uit de XML gehaald worden. Om dit te verwezenlijken maken we gebruik van de gratis bibliotheek Xstream. Door middel van *pull parsing* zal de nodige data voor elke module worden ingelezen. Voor elke module verkrijgen we zo een object. Dankzij deze objecten worden alle gegevens van de module beschikbaar in de code.

De aangemaakte objecten van de modules worden getoond in een lijst op het scherm. In de lijst is de module voorgesteld door zowel zijn naam als een icoon weer te geven, zoals aangegeven in figuur 4. Hierdoor kan de gebruiker een module selecteren die nodig is voor de te maken schakeling. De enige ontbrekende modules in de lijst zijn de trein en de module voor voeding en grond. Deze weglating wordt doorgevoerd omdat beide modules altijd ter beschikking zijn en slechts éénmaal kunnen voorkomen in de schakeling.



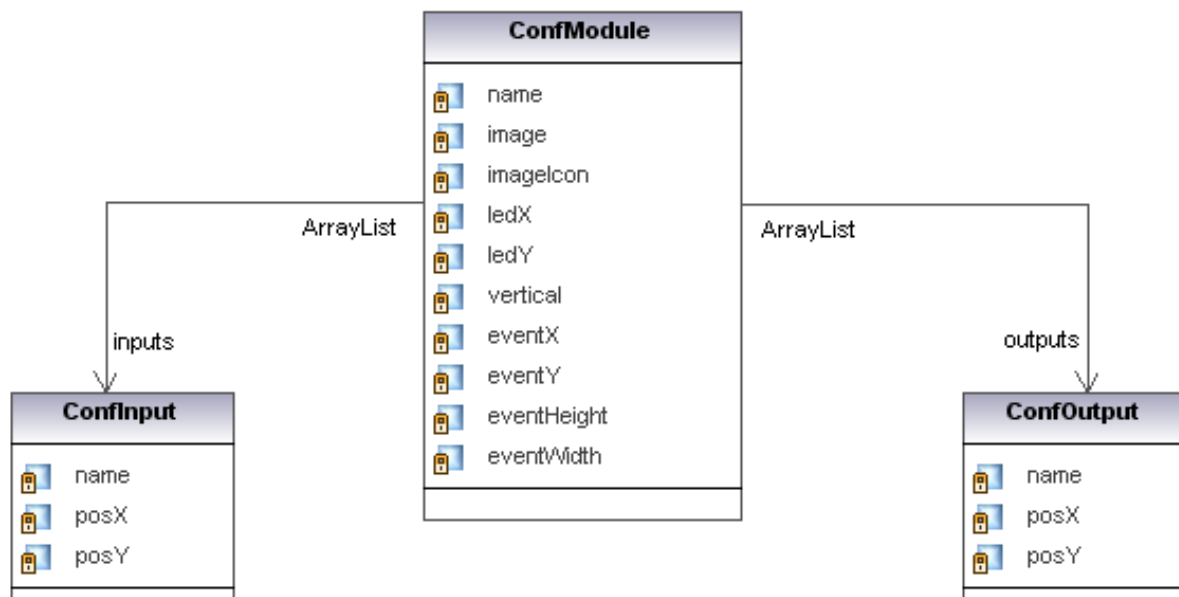
**Figuur 4: Lijst van modules**

De twee eerder vernoemde programmadelen, nl. het inlezen van de XML en het tekenen van een schakeling, moesten worden samengevoegd. Dit proces heeft echter enkele conflicten veroorzaakt. De twee gebruikte externe bibliotheken, Xstream en Piccolo, kunnen niet zonder problemen gecombineerd worden. Objecten, aangemaakt vanuit XML door Xstream, zijn niet bruikbaar bij het tekenen van een schakeling met behulp van Piccolo. Een oplossing hiervoor is het splitsen van de objecten in twee objecten. Eén van de objecten dient om de gegevens vanuit XML in de programmacode te zetten via Xstream. De bekomen objecten krijgen de benaming configuratieobjecten. Voor het tekenen verkrijgen we dan een ander object van de module. Deze objecten kunnen echter wel de data halen uit de eerder aangemaakte configuratieobjecten.

### 3.3.3 Configuratiemodules

Zoals eerder al vermeld, dienen de configuratiebestanden voor het vermijden van conflicten. Bij het opstarten van het programma, wordt voor elke beschikbare module een object gemaakt. Deze zogenaamde configuratieobjecten dienen enkel als tussenstap tussen het inlezen van de gegevens vanuit XML en het eigenlijke tekenen van een schakeling. Voor elke module zal de data vanuit de XML-bestanden geplaatst worden in deze configuratieobjecten. Op deze manier is de nodige data eenvoudig beschikbaar in de programmacode. Bovendien verkrijgen we door deze manier van werken een hiërarchie waarbij eenvoudig de modules kunnen worden aangepast, zonder hercompilatie van de code. De enige vereiste ingreep is dan een herstart van het programma om de gewijzigde data in te lezen.

De gebruikte structuur voor de configuratiemodules in het programma is getoond in figuur 5. Elke module wordt opgevat als een configuratiemodule. Elk object van de klasse ConfModule bevat de nodige gegevens voor de verwerking in het programma. De objecten die zijn aangemaakt bij het uitlezen van de XML-bestanden dienen enkel om de gegevens beschikbaar te maken in de code. Enkel bij een selectie wordt een eigenlijk object aangemaakt van de module.

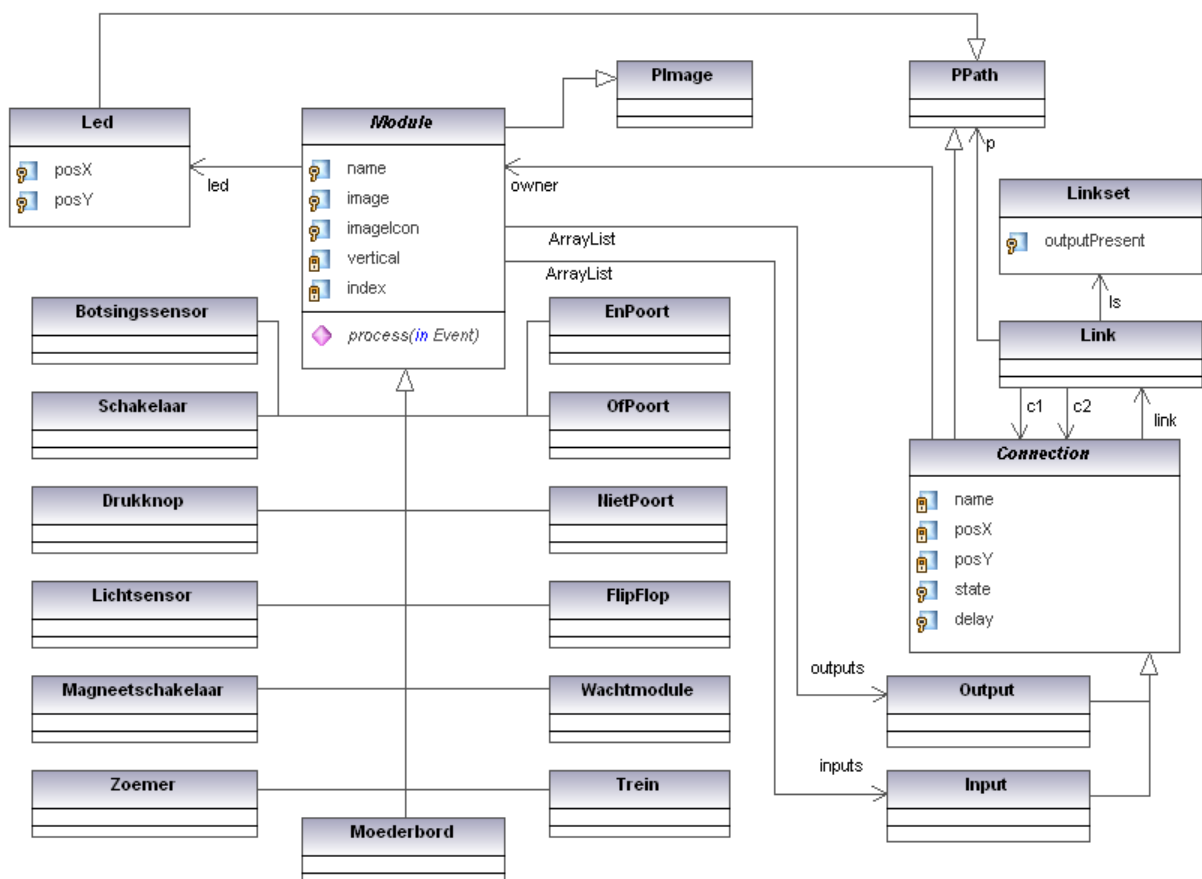


Figuur 5: UML voor de configuratiemodules

### 3.3.4 Modules

Bij iedere selectie van een module, wordt een nieuw object aangemaakt. Dit object zal dan de nodige informatie halen uit de reeds aangemaakte configuratieobjecten. Zo zullen er bijvoorbeeld kopieën worden aangemaakt van de ArrayLists van alle in- en uitgangen. De gegevens van de configuratiebestanden zullen nu verder aangevuld worden, zodat de modules kunnen bewerkt worden. De zo ontstane objecten worden gebruikt om zowel het tekenen als het simuleren van de schakeling te realiseren.

Voor elke beschikbare module is er een aparte klasse voorzien, zoals getoond in figuur 6. Alle klassen voor de modules hebben echter een gemeenschappelijke klasse waarvan ze overerven. Deze gemeenschappelijke klasse is de abstracte klasse *Module*. Het merendeel van de attributen en methoden zijn algemeen voor elke module en kunnen dus geplaatst worden in de klasse *Module*. *Module* voorziet tevens een abstracte methode *process*. Dit verplicht elke module klasse een eigen verwerking aan te maken voor deze methode. In *process* worden de binnenkomende signalen verwerkt en de uitgangen aangepast.



Figuur 6: UML voor de modules

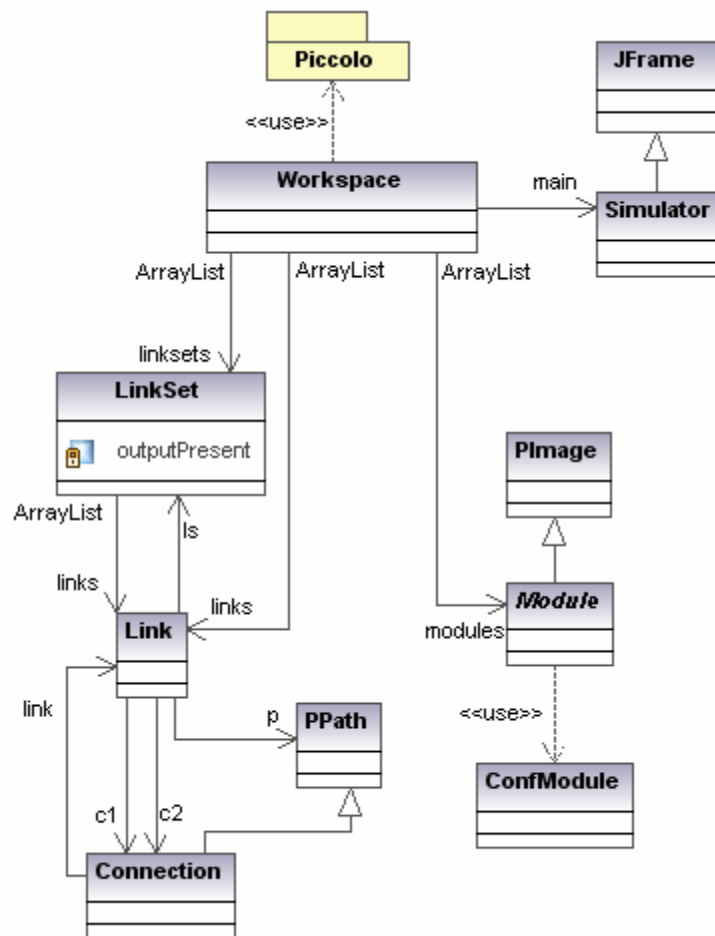


Een opvallende uitbreiding t.o.v. de configuratieobjecten, is dat de Input en Output klassen overerven van de klasse Connection. Hierdoor kan de code beperkter blijven, aangezien de Input en Output klassen hier aanvullende functies bezitten die gedeeltelijk identiek zijn. Een verdere toevoeging zijn de klassen Link en Linkset. Deze zorgen voor een goede verwerking tijdens de simulatie. Bovendien wordt het tekenen van een schakeling en de controle op mogelijke fouten in de schakeling eenvoudiger.

## 3.4 Workspace

### 3.4.1 Programma-architectuur

De workspace is een venster waarin de gebruiker een schakeling kan aanmaken. Een workspace maakt gebruik van de grafische bibliotheek Piccolo om de modules met zijn connecties en verbindingen weer te geven. Hiervoor houdt hij drie ArrayLists bij. Namelijk één voor de modules, één voor de links en één voor de linksets. Ook houdt een workspace de referentie naar de hoofdklasse Simulator bij. Figuur 7 geeft een overzicht van de structuur van een workspace.



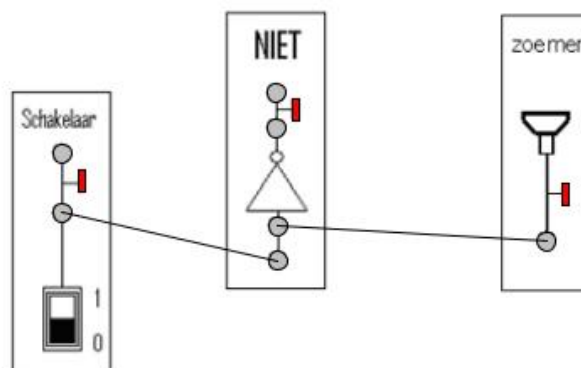
Figuur 7: UML voor het tekenen van een schakeling

Bij de selectie van een module uit het venster met de beschikbare modules, zal het programma een configuratiemodule doorgeven aan de workspace. Aan de hand van deze configuratiemodule, maakt de workspace een module aan. Hierbij voegt hij de juiste *listeners*, die nodig zijn voor het tekenen, toe aan de connecties. De ArrayList modules houdt alle modules bij die in de workspace getekend zijn. Zo is het achteraf eenvoudig om alle modules in een schakeling op te vragen.

De klasse Connection stelt een connectie op een module voor. Deze erft over van PPath zodat het gemakkelijk is om deze te tekenen op de module. Een verbinding tussen 2 connecties wordt geplaatst in een Link-object. Dit object houdt de 2 connecties bij en een PPath. Dit PPath is het pad tussen de 2 verbonden connecties en wordt gebruikt om een lijn te tekenen. Tevens houdt iedere connectie een verwijzing naar de Link waartoe hij behoort bij. De workspace houdt al de Links bij in een ArrayList links. Door deze structuur is het gemakkelijk om voor alle connecties na te gaan of ze met een andere connectie verbonden zijn.

Bij een module zijn er meestal meerdere connecties per in- of uitgang mogelijk. Indien dit het geval is, hebben deze connecties dezelfde signaalnaam. Wanneer een Link wordt aangemaakt, zal de workspace deze altijd in een LinkSet plaatsen. De workspace controleert of de signaalnaam van een van de connecties van een Link al voorkomt in een LinkSet. Indien dit het geval is, voegt hij de Link toe aan deze LinkSet. Anders maakt hij een nieuwe LinkSet aan.

Figuur 8 verduidelijkt het gebruik van een LinkSet. De 2 Links vormen samen een LinkSet omdat ze beiden de signaalnaam van de ingang van de NIET-poort bevatten. De 2 connecties aan deze ingang zijn doorverbonden. Hierdoor is de uitgang van de schakelaar onrechtstreeks verbonden met de ingang van de zoemer. Dit verduidelijkt ook meteen het nut van een LinkSet. Indien de onrechtstreekse verbinding niet zou worden bijgehouden, zou het ook mogelijk zijn 2 uitgangen onrechtstreeks te verbinden. Daarom houdt iedere LinkSet een variabele bij om aan te geven als er al een uitgang in de LinkSet aanwezig is. Op deze manier kan een onrechtstreekse verbinding van 2 uitgangen vermeden worden.

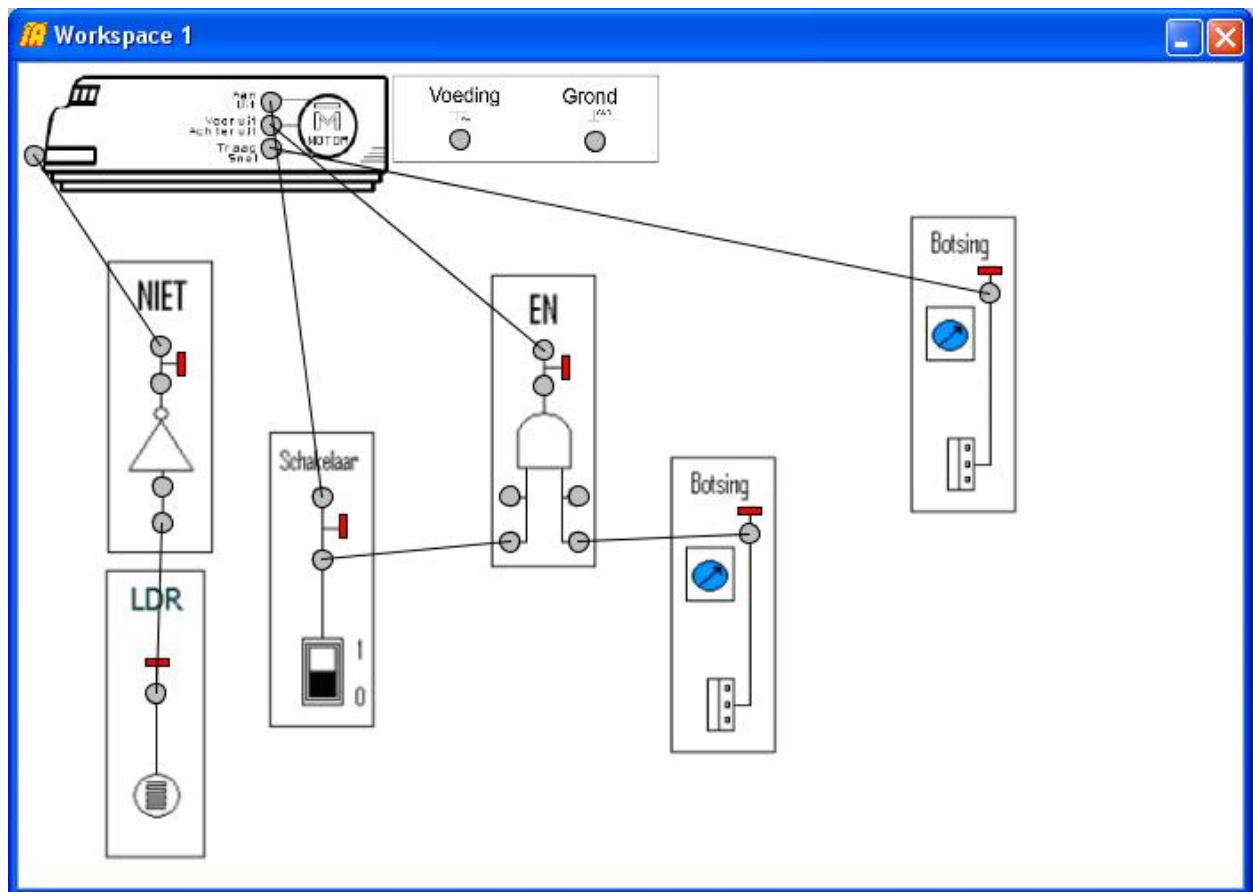


**Figuur 8: Voorbeeld van een linkset**

### 3.4.2 Teken van een schakeling

In de workspace kan de gebruiker de schakeling aanmaken die hij wil simuleren. Het is mogelijk om in meerdere workspaces tegelijk te werken. Om dit mogelijk te maken, onthoudt het programma welke workspace het laatst actief is. Ook voorziet het programma een werkbalk met knoppen om lijnen te tekenen en te verwijderen en om modules te verwijderen. Om alle acties te stoppen, volstaat het om eender waar in de workspace met de rechtermuisknop te klikken.

In iedere workspace is een trein getekend die niet verwijderd kan worden. Dit omdat er maar één trein in de schakeling aanwezig kan zijn. In het geval van de voeding en de grond gelden dezelfde regels. Ook deze kunnen daarom niet worden verwijderd van de workspace. Figuur 9 geeft een voorbeeld van hoe een workspace er uitziet.



Figuur 9: Voorbeeld van een workspace

De gebruiker kan een module selecteren in een apart venster, waarin de configuriemodules ingelezen zijn. Een tijdelijke afbeelding van de module zal bij de cursor verschijnen tot de module wordt toegevoegd aan de workspace door een muisklik. Hierbij maakt het programma de module aan en plaatst de in- en uitgangen en voegt de nodige *listeners* toe. De mogelijkheid om modules te verwijderen is ook voorzien. Hierbij zal de module transparant worden indien de cursor over de module beweegt. Dit om duidelijk aan te geven dat er interactie mogelijk is met

de module. Bij het verwijderen van een module, verwijdert het programma ook alle lijnen die met deze module verbonden zijn.

De gebruiker kan modules verbinden door een lijn te tekenen tussen de in- of uitgangen van twee modules. Hierbij licht de connectie op in een bepaalde kleur, afhankelijk of er al dan niet een lijn getekend mag worden met de connectie. Indien het toegelaten is een lijn te tekenen, zal de connectie groen oplichten wanneer de cursor erover beweegt. Als de gebruiker een foute verbinding wil maken, zoals 2 uitgangen met elkaar verbinden, zal de connectie rood oplichten. Tekent hij de lijn toch, dan geeft het programma een foutmelding. Dit is om zoveel mogelijk fouten bij de simulatie eruit te halen. Bijlage B geeft een overzicht van de mogelijke fouten die het programma kan detecteren.

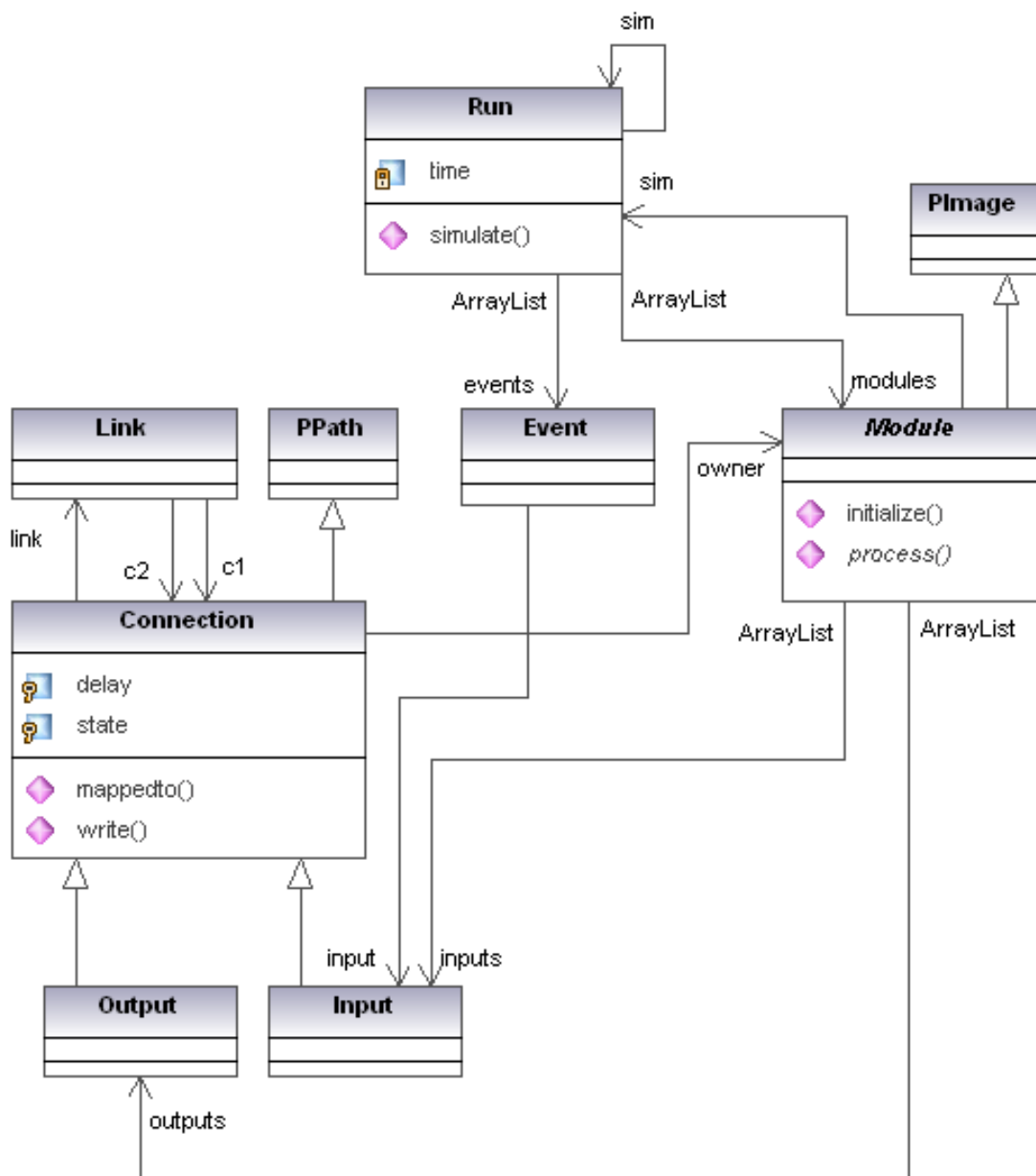
Het is ook mogelijk lijnen te verwijderen. Net zoals bij het verwijderen van een module, wordt een lijn transparant indien de cursor erover beweegt en de verbonden connecties lichten op. Indien de cursor echter over een connectie gaat waar een lijn aan verbonden is, zal de connectie oplichten. Ook hier wordt de lijn transparant en licht de verbonden connectie op. Na een muisklik op de lijn of één van de verbonden connecties, verwijdert het programma de lijn.

Alle modules en verbindingen worden bijgehouden per workspace. Zo is er een duidelijke structuur aanwezig van modules en verbindingen. Dit is nodig zodat de simulatie eenvoudig de nodige modules en verbindingen kan ophalen.

## 3.5 De simulatie

### 3.5.1 Programma-architectuur

Figuur 10 verduidelijkt de gebruikte structuur bij het simuleren van een schakeling. Gedurende de simulatie zal de volledige coördinatie vanuit de klasse Run plaatsvinden. Om een overzicht te verkrijgen van de te simuleren schakeling, wordt de ArrayList van modules bijgehouden. De manier waarop de modules dan met elkaar zijn verbonden is terug te vinden via de klasse Connection. Bovendien bezit de klasse Run een variabele om de tijd bij te houden: het attribuut time.



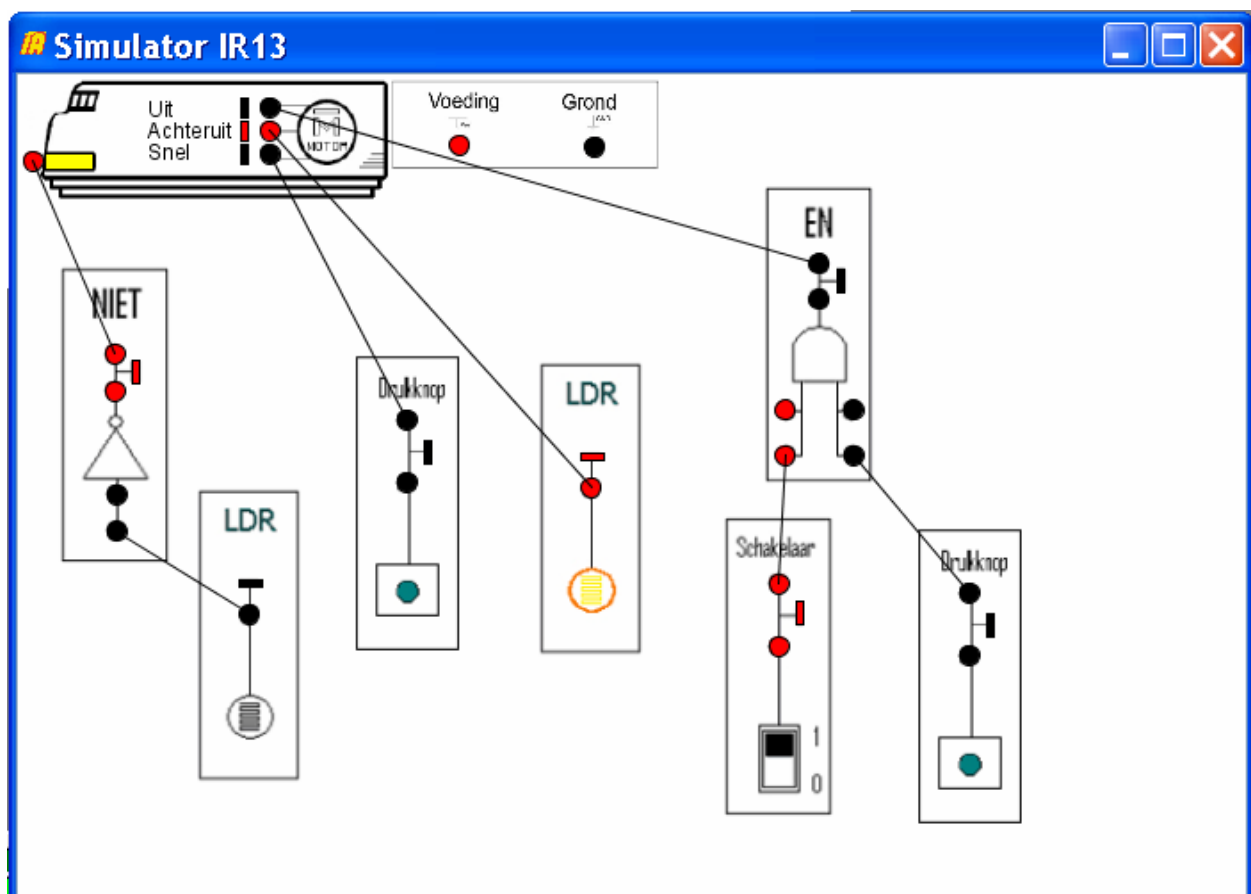
Figuur 10: UML voor de simulatie

Aan iedere module wordt een vertraging toegewezen. Dit zorgt voor een meer realistischere benadering van de werkelijkheid. Om dit fenomeen in praktijk uit te voeren wordt de *event* met een vertraging doorgestuurd naar de volgende module. De enige uitzonderingen op deze regel zijn de *events* afkomstig van een ingang. Deze genereert een event met als vertraging 0. In werkelijk heeft het doorgeven van een signaal via een verbinding eveneens slechts een verwaarloosbare tijd nodig.

### **3.5.2 Bediening**

Bij aanvang van de simulatie zullen reeds de nodige aanpassingen worden doorgevoerd. Deze initialisatie procedure verloopt eveneens door middel van de generatie en verwerking van *events*. Tijdens de simulatie is het echter niet mogelijk om te werken met realistische sensoren. Vandaar dat de gebruiker deze modules manueel kan aanpassen door middel van de muis. Ook de schakelaar en de drukknop kunnen op deze manier bediend worden. Hierdoor blijft de bediening eenvoudig. Zo verkrijgen we een simulatie die de realiteit nabootst, waarbij de gebruiker de werking van de schakeling kan beïnvloeden. Bovendien blijft de simulator gebruiksvriendelijk, waardoor kinderen van 13 jaar geen moeite ondervinden bij het gebruik van het programma.

Wanneer de simulatie wordt gestart, zal een nieuw venster verschijnen. Het venster toont de gemaakte schakeling opnieuw. Net zoals in de realiteit zal de led van een module oplichten wanneer zijn uitgang hoog is. Een led heeft een zwarte kleur voor de uittoestand en kleurt rood om het oplichten na te bootsen. Als extra toevoeging bij het simuleren zullen ook de in- en uitgangen op deze manier kleuren. Hierdoor is het eenvoudiger om het signaal te volgen. Bovendien hebben de modules die met de muis kunnen bediend worden een afbeelding afhankelijk van hun toestand. Eveneens de lamp op de locomotief van de trein kan oplichten, en zal dan een gele kleur krijgen. Figuur 11 verduidelijkt dit principe.



**Figuur 11: Weergave bij de simulatie**



### 3.5.4 Events

De procedure van het simuleren is gebaseerd op het genereren en verwerken van *events*. Wanneer een wijziging optreedt bij één van de modules, zal er een *event* gegenereerd worden. Door middel van dit *event* weten alle modules die verbonden zijn dat er een wijziging heeft plaatsgevonden. Deze wijziging zal dan worden verwerkt. Indien dit opnieuw zorgt voor een wijziging van de uitgang van deze module, zal een nieuw *event* gegenereerd worden. Dit proces zal zich blijven herhalen tot alle modules, waarop de wijziging effect heeft, zijn aangepast. Enige uitzondering op dit principe is de wachtmodule. Meer uitleg over de werking van deze wachtmodule is terug te vinden in bijlage F.

### 3.5.5 Logboek

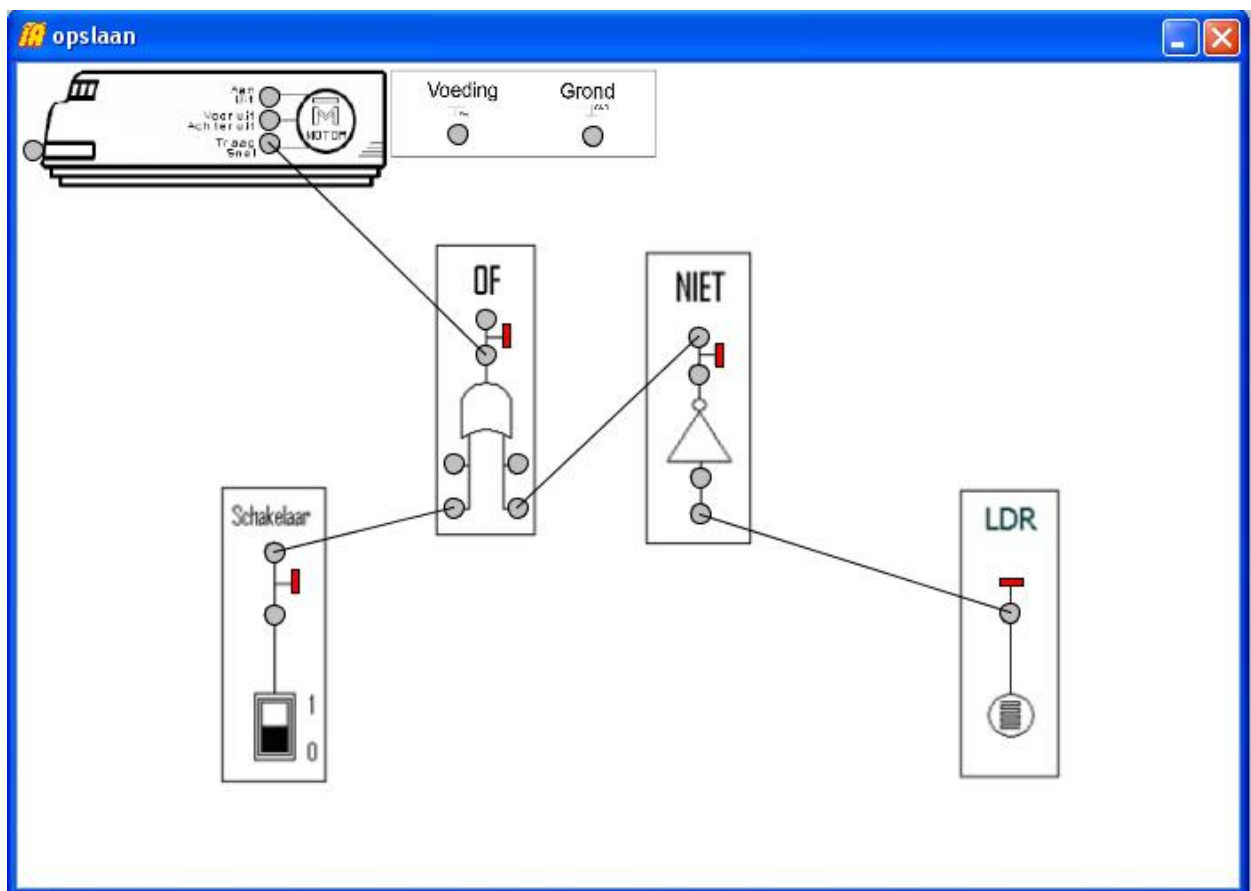
Om een goed zicht te krijgen op de simulatie van een schakeling is de grafische weergave onvoldoende. Er wordt niet getoond wanneer er al of niet *events* worden gegenereerd. Om toch een beeld te krijgen van de verwerking is het mogelijk om gebruik te maken van een logbestand. Hiermee bekomen we een overzichtelijk tekstbestand waarmee we kunnen volgen wanneer en waar een event wordt aangemaakt. Dit principe is in onze masterproef aangewend tijdens het *debuggen* van de code. *Debuggen* betekent het zoeken van mogelijke fouten in de gebruikte code of algoritmes van het programma. Op deze manier is bijvoorbeeld ontdekt dat aanvankelijk geen event op een ingang gestuurd werd indien hij was verbonden met een andere ingang. Na het oplossen van alle fouten worden de methodes voor de logger er opnieuw uit verwijderd, aangezien deze enkel nodig waren tijdens het debuggen. Zo blijft de snelheid van het programma optimaal.

## 3.6 Extra functionaliteiten

### 3.6.1 Opslaan/opslaan als

Als een gebruiker een schakeling aan het tekenen is, kan het dat hij door bijvoorbeeld tijdgebrek niet meer kan verder werken. Hiervoor is een functie voorzien om een schakeling op te slaan. Het programma schrijft dan de noodzakelijke gegevens weg in een tekstbestand. Indien de schakeling voor de eerste keer wordt opgeslagen, roept het programma de functie “opslaan als” op. Hierbij opent de verkenner en kan de gebruiker de locatie en de naam van het bestand kiezen. Het programma houdt de naam en locatie bij zodat de gebruiker ze niet steeds opnieuw moet ingeven. Dit is echter wel mogelijk door de functie “opslaan als” te kiezen.

Het programma slaat de namen van de modules en zijn index op. Ook de positie van de module in de schakeling wordt opgeslagen. Daarna volgen de 2 namen van de connecties die verbonden zijn met elkaar en de index van de module waartoe ze behoren. De woorden ‘links’ en ‘end’ geven respectievelijk het begin van de connecties en het einde van het bestand aan. Figuur 12 geeft een voorbeeld van een schakeling



Figuur 12: Opgeslagen voorbeeld van een workspace

Het opgeslagen tekstbestand horend bij figuur 12 geeft figuur 13.

```
trein 0 2.5 3.0
moederbord 1 210.0 3.0
schakelaar 2 112.0 236.0
OF-poort 3 232.0 99.0
NIET-poort 4 351.0 102.0
lichtsensor 5 526.0 236.0
links
uitgangA 5 ingangA2 4
ingangB2 3 uitgangB1 4
uitgangA1 2 ingangA2 3
ingangC_traag_snel 0 uitgangC2 3
end
```

**Figuur 13: Opgeslagen tekstbestand**

### 3.6.2 Laden

Een schakeling opslaan is natuurlijk nutteloos als hij niet opnieuw ingeladen kan worden. Daarom voorziet het programma ook een functie om schakelingen in te laden. Net zoals bij “opslaan als” opent de verkenner bij het laden, zodat de gebruiker het juiste bestand kan kiezen.

Het programma leest het tekstbestand regel per regel in. Eerst leest het de namen van de modules in met de bijhorende index en zijn posities. Dit volstaat om de juiste module aan te maken, ze te indexeren en in een nieuwe workspace te plaatsen. Vervolgens leest het programma de namen van de connecties met de index in. Aan de hand van de naam en de index kan het programma de juiste connectie op de juiste module zoeken. Hierna worden de 2 connecties op één regel van het tekstbestand met elkaar verbonden.

Door dit systeem van opslaan en laden bekomen we een eenvoudige voorstelling van een schakeling. De gegevens in het opgeslagen tekstbestand zijn tot een minimum beperkt, maar toch voldoende om de schakeling te kunnen inladen. Zo neemt een opgeslagen schakeling niet veel ruimte in van de harde schijf of een ander geheugen.

## 4 Modules toevoegen

Een vereiste van onze masterproef was dat er achteraf op een eenvoudige manier modules toegevoegd kunnen worden. Daarom hebben we de aanpassingen die nodig zijn tot een minimum proberen te houden. Om een module toe te voegen zijn er enkele stappen nodig. Als eerste zijn 2 nieuwe bestanden nodig voor een module, namelijk een XML-bestand en een javabestand. Nadat deze aangemaakt zijn, dient men enkele regels javacode toe te voegen in de hoofdklasse. Volgende paragrafen verduidelijken deze stappen.

### 4.1 Het XML-bestand

In het XML-bestand staat de configuratie van de module. De in- en uitgangen staan hierin, samen met hun posities. Indien er geen in- of uitgangen aanwezig zijn op de module, mogen deze worden weggelaten uit het XML-bestand. Ook de positie van de led staat in dit bestand, met een parameter die aangeeft of de led horizontaal of verticaal gericht is. Verder staat in dit XML-bestand de naam van het module en het pad naar de afbeeldingen van de module. Het icoon wijst op de afbeelding die getoond wordt in de lijst waaruit de modules geselecteerd kunnen worden. De gewone afbeelding duidt op de afbeelding die gebruikt wordt bij het tekenen.

Het is mogelijk dat de gebruiker tijdens de simulatie in staat is om bijvoorbeeld een knop op de module te verzetten. In dit geval moeten er nog enkele extra parameters toegevoegd worden in het XML-bestand. Deze bepalen het gebied waarop de gebruiker kan klikken. De parameters mogen weggelaten worden indien er geen interactie mogelijk is bij de simulatie.

Een meer gedetailleerde uitleg over het XML-bestand en voorbeelden zijn gegeven in bijlage C. Het programma leest de configuratie uit het XML-bestand in met behulp van Xstream en plaatst ze in een configuratiemodule. Deze configuratiemodule is gemeenschappelijk voor alle modules en dient niet aangemaakt te worden. Meer informatie over het configuratiebestand is terug te vinden in bijlage D.

## 4.2 De module

Een tweede bestand dat nodig is om een module aan te maken, is een javabestand voor de module zelf. Hiermee zal het programma de module aanmaken en plaatst tevens zijn in- en uitgangen. Eén methode moet altijd aanwezig zijn in dit bestand, namelijk de methode `process(Event evt)`. Deze methode zorgt voor de verwerking van de in- en uitgangen. Hier stelt de parameter `evt` het binnenkomende event voor. De overige variabelen en methodes erft de module over van een algemeen moduleobject.

Bij het XML-bestand kunnen extra parameters meegegeven zijn voor de interactie bij de simulatie. Indien dit het geval is, moet er een extra methode voorzien worden, namelijk `addListeners()`. Deze methode maakt het gebied aan waarop de gebruiker met zijn muis kan klikken. Ook wordt hier toegevoegd welke handelingen dienen te gebeuren als er met een muis op dit gebied geklikt wordt.

Verder kan het zijn dat ook de methode `colorNodes()` nodig is. Normaal kleurt de module bij de simulatie zijn connecties aan de hand van hun status. Ook de led zal kleuren aan de hand van de status van de uitgang. Indien dit anders moet gebeuren bij een bepaalde module, zal deze een eigen methode `colorNodes()` moeten hebben die de connecties en led kleurt. Meer uitleg hierover is terug te vinden in bijlage E.

Indien de module zijn in- of uitgangen al moet wijzigen als de simulatie start, moet ook de methode `initialize()` aanwezig zijn. Dit is bijvoorbeeld bij de NIET-poort het geval. Hier is immers bij de start van de simulatie de uitgang al 1 wanneer de ingang 0 is. Voor deze module zal de methode `initialize()` de uitgangen dus al gaan aanpassen.

Voor een module moeten dus enkele belangrijke modules voorzien worden. De methode `process(Event e)` moet altijd aanwezig zijn. De methodes `addListeners()`, `colorNodes()` en `initialize()` zijn optioneel. Indien een van deze laatste drie methodes niet in de module aanwezig zijn, zal de methode uit de algemene moduleklasse genomen worden. Meer uitleg en voorbeelden over de module is te vinden in bijlage E.

### 4.3 De code

Als de voorgaande 2 bestanden aangemaakt zijn, moeten deze nog in de code aangebracht worden. Hiervoor moet er in 2 bestanden een kleine wijziging gebeuren, namelijk Simulator.java en Workspace.java. Deze bestanden zijn terug te vinden in de *package* “be.khlim.trein.gui”.

Het eerste bestand dat gewijzigd moet worden, is Simulator.java. Hierin moet in de methode makeModules( ) nog de link naar het XML-bestand staan. Dit gebeurt door in deze methode de methode readConfModule(String location, Xstream x) te plaatsen. Deze methode zorgt voor de aanmaak van een configuratiemodule. De parameter location is het pad van het XML-bestand en x is het xstreamobject, in de klasse Simulator xstream genaamd. Voor een module met een XML-bestand met de naam ‘modulenaam.xml’ wordt dit dan:

```
readConfModule("XML/modulenaam.xml", xstream);
```

Vervolgens moet enkel nog Workspace.java gewijzigd worden. Het programma vergelijkt in de methode readModule(ConfigModule cmod, Point2D p, int ind) op de namen van de doorgegeven configuratiemodule aan de hand van if-else-lussen. Om een nieuwe module toe te voegen, volstaat het om een nieuwe if-lus toe te voegen die controleert op de naam van deze module. De naam van de module is degene die in het XML-bestand is meegegeven. In deze if-lus maakt het programma dan de juiste module aan. Voor een module met naam ‘modulenaam’ en een modulebestand met naam ‘ModuleNaam.java’ wordt dit dan.

```
else if (name.matches("modulenaam")){  
    mod = new ModuleNaam(cmod);  
}
```

Zo maakt het programma de juiste modules aan. Verder zijn er geen wijzigingen nodig.

### 4.4 Conclusie

Om te besluiten vatten we even kort de nodige stappen nog eens samen.

1. XML-bestand aanmaken
2. Module aanmaken
3. Het configuratiebestand toevoegen in de code van Simulator.java
4. Zorgen dat de juiste module wordt aangemaakt in Workspace.java

## 5 Besluit

Het programma voldoet aan de vooropgestelde eisen. Het is mogelijk een schakeling te tekenen via een eenvoudige bediening met de muis. Ook detecteert het programma zoveel mogelijk eventuele fouten bij het aanmaken van een schakeling. Hierdoor bekomen we een gebruiksvriendelijk programma dat in zekere mate *'foolproof'* is. Tevens is het uitzicht van de modules zo dicht mogelijk bij de realiteit gehouden. Zo blijft het programma duidelijk voor leerlingen van 13 jaar.

Het is eveneens mogelijk een schakeling volledig te simuleren. Ook hierbij is er rekening mee gehouden dat de eindgebruikers 13-jarige kinderen zijn. De gebruiker kan de schakelaar, drukknop en de sensors bedienen met een eenvoudige muisklik. Bij deze simulatie lichten de leds van de modules op zoals dit in werkelijkheid gebeurt. Tevens worden de connecties zwart of rood gekleurd, afhankelijk van hun toestand. Zo kan de gebruiker op een eenvoudige manier de simulatie opvolgen.

Een vereiste van de opdrachtgever was dat er eenvoudig nieuwe modules kunnen toegevoegd worden aan het programma. De nodige aanpassingen hiervoor hebben we dan ook tot een minimum beperkt. Hiervoor dienen enkel 2 bestanden aangemaakt te worden, namelijk een XML-bestand en een javabestand. Verder is het vereist om enkele regels javacode toe te voegen in het hoofdprogramma. Hierdoor kan op een relatief snelle wijze een nieuwe module worden toegevoegd. Om dit te realiseren zal echter wel enige voorkennis van Java vereist zijn.

# Literatuurlijst

Apache, “Apache log4j.” On line, <http://logging.apache.org/log4j/>, 22-04-2008

M. Chauhan, “Logging with log4j: an efficient way to log java applications.” On line, <http://www.developer.com/open/article.php/3097221>, 22-04-2008

J. Douglas, “JHotDraw pattern language.” On line, <http://softarch.cis.strath.ac.uk/PLJHD/Patterns/JHDDomainOverview.html>, 24-09-2007

J. Gosling, J. Bill, G. Steele en G. Bracha, The Java Language Specification. Stoughton: Addison Wesley Professional, 2005.

Grappa, “A Java Graph Package.” On line, <http://www.research.att.com/~john/Grappa/>, 25-09-2007

C. S. Horstmann en G. Cornell, Core Java 2 Volume I – Fundamentals. Santa Clara: Sun Microsystems Press, 2005.

JHotDraw, “JHotDraw as an open-source project.” On line, <http://www.jhotdraw.org/>, 24-09-2007

Jung, “Java Universal Network/Graph Framework.” On line, <http://jung.sourceforge.net/>, 24-09-2007

D. Mabbutt, “VB versus Java.” On line, <http://visualbasic.about.com/b/2007/04/10/vb-versus-java.htm>, 11-09-2007

J. O'Madadhain, D. Fisher, S. White en Y. Boey, “The Jung framework.” On line, [http://www.datalab.uci.edu/papers/JUNG\\_tech\\_report.html](http://www.datalab.uci.edu/papers/JUNG_tech_report.html), 24-09-2007

G. Palmer, Java Programmer's Reference. Birmingham: Wrox press Ltd, 2000.

Piccolo, “A Structured 2D Graphics Framework.” On line, <http://www.cs.umd.edu/hcil/jazz/>, 25-09-2007

L. Rutten, Cursus Inf8: Gedistribueerde Informatiesystemen. Diepenbeek: KHLim, 2007.

M. Sahaf, “Java advantages and disadvantages.” On line, <http://www.webdotdev.com/nvd/articles-reviews/java/java-advantages-and-disadvantages-1042-66.html>, 11-09-2007



Sun, “Java Logging APIs.” On line, <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/>, 22-04-2008

J. Surveyer, “VB.NET vs Java comparision.” On line, <http://theopensourcery.com/vbjava2.htm>, 11-09-2007

C. Utley, “Why should you move to C#?” On line, [http://articles.techrepublic.com.com/5100-10878\\_11-1050356.html](http://articles.techrepublic.com.com/5100-10878_11-1050356.html), 11-09-2007

D. Wearn, “Why do they program in C++?” On line, <http://lambda-the-ultimate.org/node/663>, 11-09-2007

Wikipedia. On line, <http://www.wikipedia.org>

XStream. On line, <http://xstream.codehaus.org/>, 3-10-2007

# Bijlagen

## *Lijst met bijlagen*


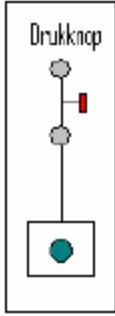

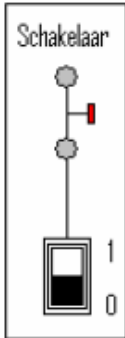
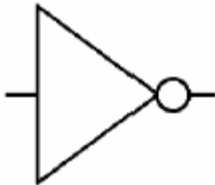
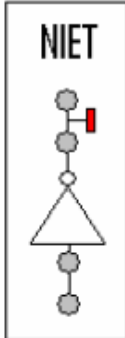

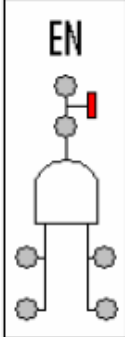
Bijlage A: Overzicht van de modules.....	50
Bijlage B: Foutmeldingen.....	55
Bijlage C: XML-sjabloon.....	59
Bijlage D: Configuratiemodulesjabloon.....	63
Bijlage E: Modulesjabloon.....	66
Bijlage F: Tijdsafhankelijke modules.....	71


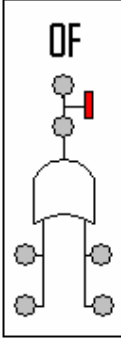
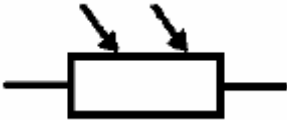
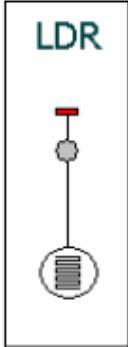

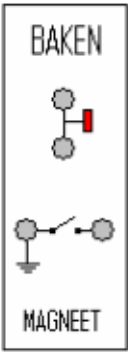

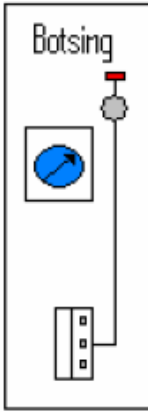
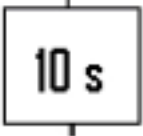
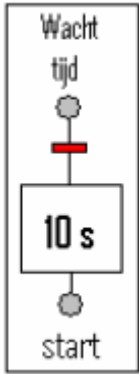
## Lijst van illustraties


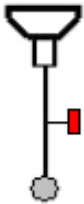
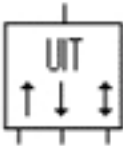
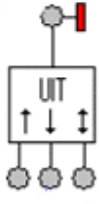
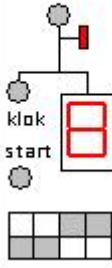
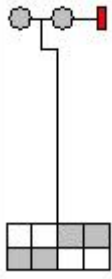
Figuur 14: Overzicht van alle beschikbare modules .....	54
Figuur 15; Foutmelding voor het verbinden van een connectie met zichzelf .....	55
Figuur 16: Foutmelding voor het tekenen van meerdere lijnen op 1 connectio.....	56
Figuur 17: Foutmelding voor het verbinden van 2 uitgangen.....	57
Figuur 18: Foutmelding voor het onrechtstreeks verbinden van 2 uitgangen.....	58
Figuur 19: benaming connecties .....	60
Figuur 20: XML voor een en-poort.....	61
Figuur 21: Afbeelding van een schakelaar .....	62
Figuur 22: XML voor een schakelaar .....	62
Figuur 23: Javacode van de klasse ConfModule.....	65
Figuur 24: Javacode van de klasse EnPoort .....	67
Figuur 25: Constructor van de klasse Lichtsensor .....	68
Figuur 26: Javacode voor de methode addListerner van een lichtsensor.....	69
Figuur 27: Javacode voor de methode initialize van een lichtsensor.....	69
Figuur 28: Javacode van de methode colorNodes van een zoemer.....	70
Figuur 29: Javacode van de klasse Wait .....	71
Figuur 30: Javacode voor de methode process van een wachtmodule.....	72


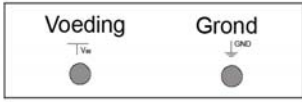
## Bijlage A: Overzicht van de modules

Figuur 14 geeft een overzicht van alle beschikbare modules.

	Naam	Symbol	Schema	Werking
	drukknop			Bij het indrukken van de drukknop verschijnt een 1 aan de uitgang. Wanneer de drukknop niet is ingedrukt is de uitgang 0.
	schakelaar			De schakelaar heeft aan de uitgang een 1 of een 0, afhankelijk van zijn stand, en houdt deze uitgang tot we de schakelaar omschakelen.
Logische poorten	NIET-poort			De NIET-poort het signaal aan zijn ingang omdraaien (inverteren).
	EN-poort			De EN-poort heeft enkel een 1 aan zijn uitgang wanneer beide ingangen 1 zijn.

	OF-poort			Aan de uitgang van de OF-poort verschijnt een 1 als minstens één van de ingangen 1 is.
sensoren	lichtsensor (LDR)			De uitgang wordt 1 wanneer er voldoende licht invalt op de sensor.
	magneetschakelaar (reed-switch)			Wanneer de sensor in de buurt komt van de magneet, zal er een 1 aan de uitgang komen.
	botsingssensor			De uitgang zal 1 worden wanneer de trein een vooraf ingestelde afstand tot een voorwerp bereikt.
	wachtmodule			Vertragingscomponent: de ingang zal met een vertraging aan de uitgang verschijnen.

uitvoersignalen	zoemer		<div data-bbox="927 159 1070 555"> zoemer   </div>	<p>Als er een 1 wordt aangelegd aan de zoemer zal er een geluid te horen zijn. Bij de simulator zal enkel zijn LED gaan branden.</p>
	flip-flop		<div data-bbox="927 636 1058 981"> FUP FLOP   </div>	<p>De uitgang wordt:</p> <ul style="list-style-type: none"> <li>➤ 1 als we de eerste ingang aansturen</li> <li>➤ 0 als we de tweede ingang aansturen</li> <li>➤ verandert van waarde als we de derde ingang aansturen</li> </ul>
	teller		<div data-bbox="927 1064 1061 1422"> Teller   </div>	<p>De teller begint af te tellen van een vooropgestelde waarde na een startsignaal. Deze waarde gaat van 0 tot F. De teller vermindert zijn waarde steeds bij een inkomend signaal op de klok. De uitgang wordt 1 als de teller op nul staat.</p>
	klok		<div data-bbox="927 1467 1058 1825"> klok   </div>	<p>De uitgang wordt om de seconde afwisselend 0 en 1</p>

	trein		<p>De trein heeft 3 ingangen die zijn gedrag beïnvloeden:</p> <ul style="list-style-type: none"> <li>➤ de bovenste bepaalt of de trein aan (1) of uit (0) is</li> <li>➤ met de middelste kunnen we de trein voor- (1) of achteruit (0) laten gaan</li> <li>➤ de onderste ingang zorgt ervoor dat de trein snel (1) of traag (0) rijdt.</li> </ul> <p>Een vierde ingang, aan de voorkant van de trein, laat een lampje branden.</p>
	moederbord		<p>De uitgang van de voeding geeft steeds een 1. Terwijl de uigang van de grond steeds 0 is.</p>

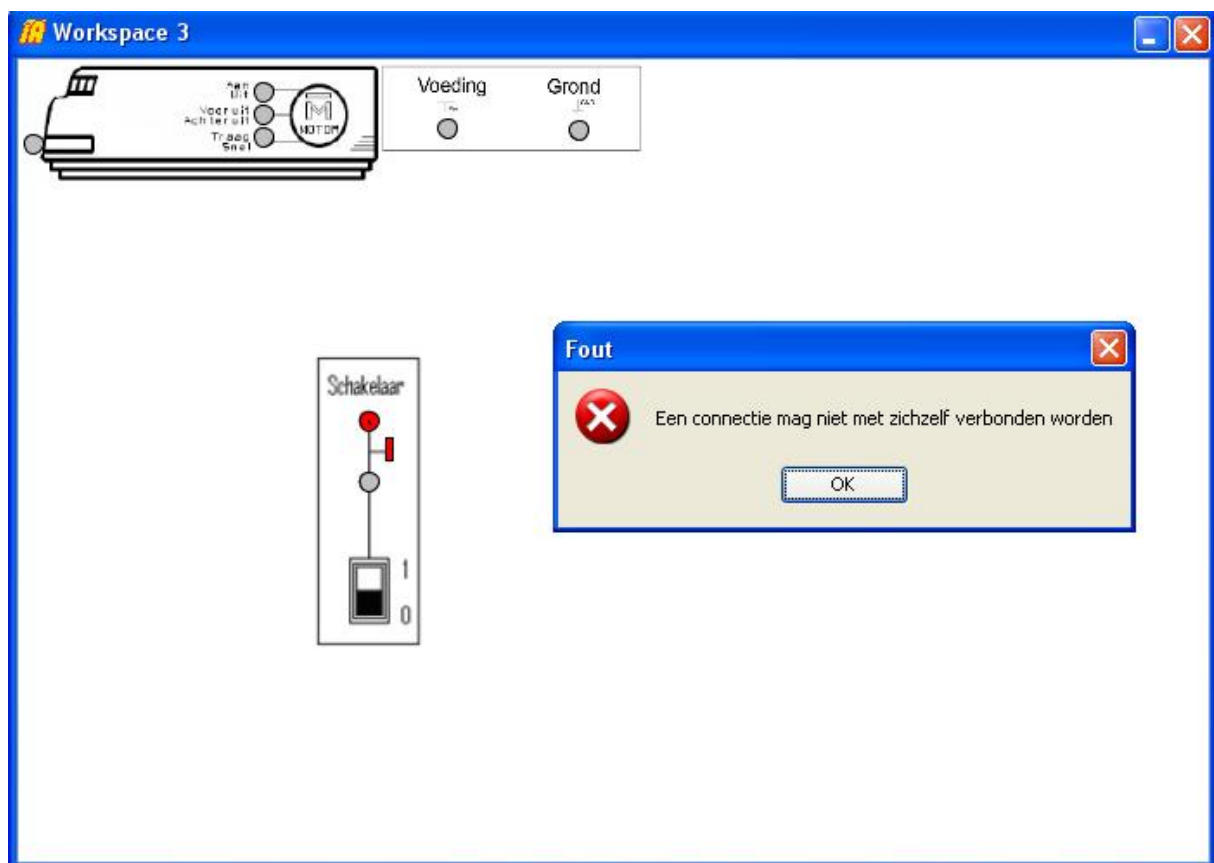
**Figuur 14: Overzicht van alle beschikbare modules**

## Bijlage B: Foutmeldingen

Het programma is in staat om enkele fouten te detecteren en te signaleren. Er zal een controle plaatsvinden op enkele veelgemaakte fouten bij het maken van schakelingen. Zo zal het bijvoorbeeld niet mogelijk zijn om 2 uitgangen met elkaar te verbinden. Hieronder volgt een overzicht van alle mogelijke fouten die het programma zal signaleren.

### 1 Connectie met zichzelf verbinden

Het programma zal niet toelaten dat een connectie met zichzelf verbonden is. Aangezien dit in werkelijkheid niet mogelijk is moet het vermeden worden om realistisch te blijven. Er kan namelijk maar één draad per connectie verbonden worden. Figuur 15 toont de foutmelding die zal verschijnen. De connectie in het rood is de connectie waarvan wordt getracht met zichzelf te verbinden.

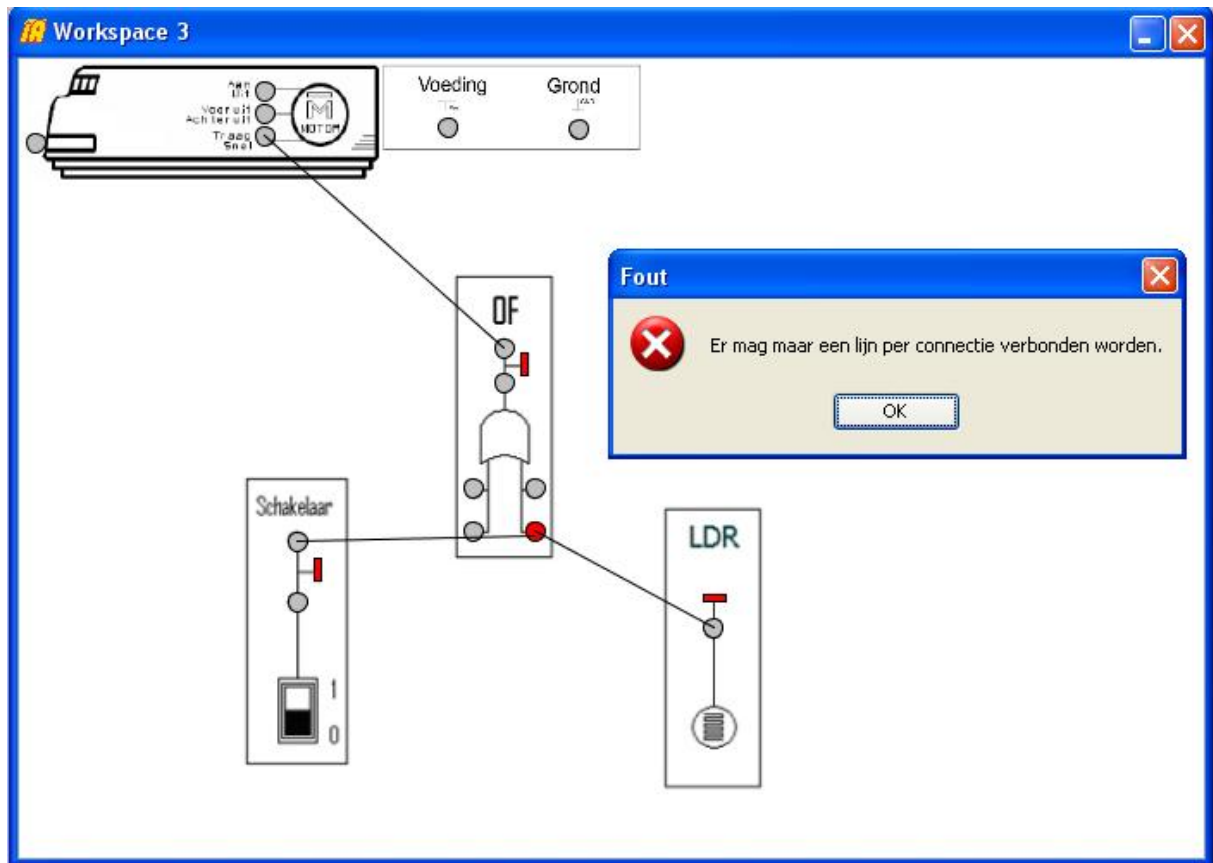


Figuur 15; Foutmelding voor het verbinden van een connectie met zichzelf



## 2 Meerdere lijnen per connectie tekenen

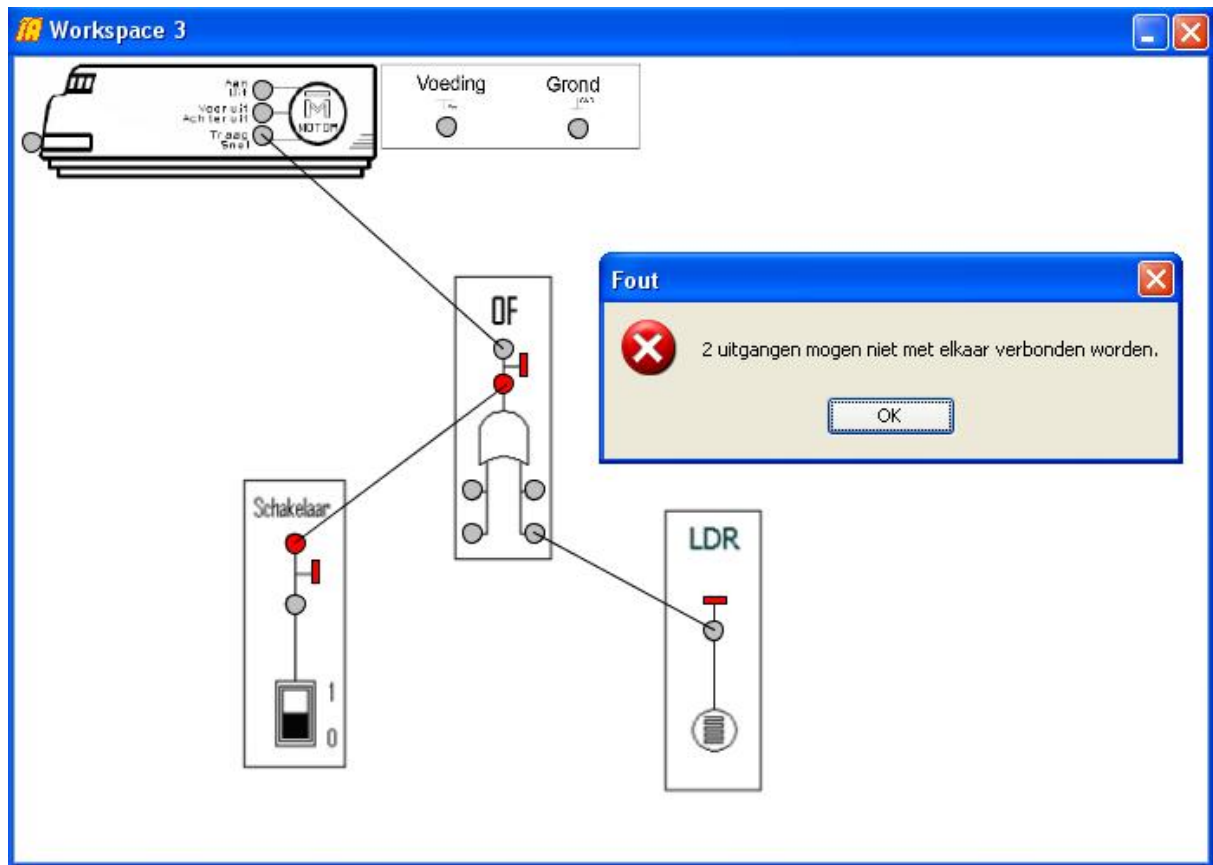
Het is onmogelijk om meerdere lijnen te tekenen in één connectie. Indien dit toch gebeurt, zal het programma een foutmelding geven. Ook hier licht de connectie waar de fout optreedt op. Figuur 16 geeft hier een voorbeeld van.



Figuur 16: Foutmelding voor het tekenen van meerdere lijnen op 1 connectio

### 3 Twee uitgangen met elkaar verbinden

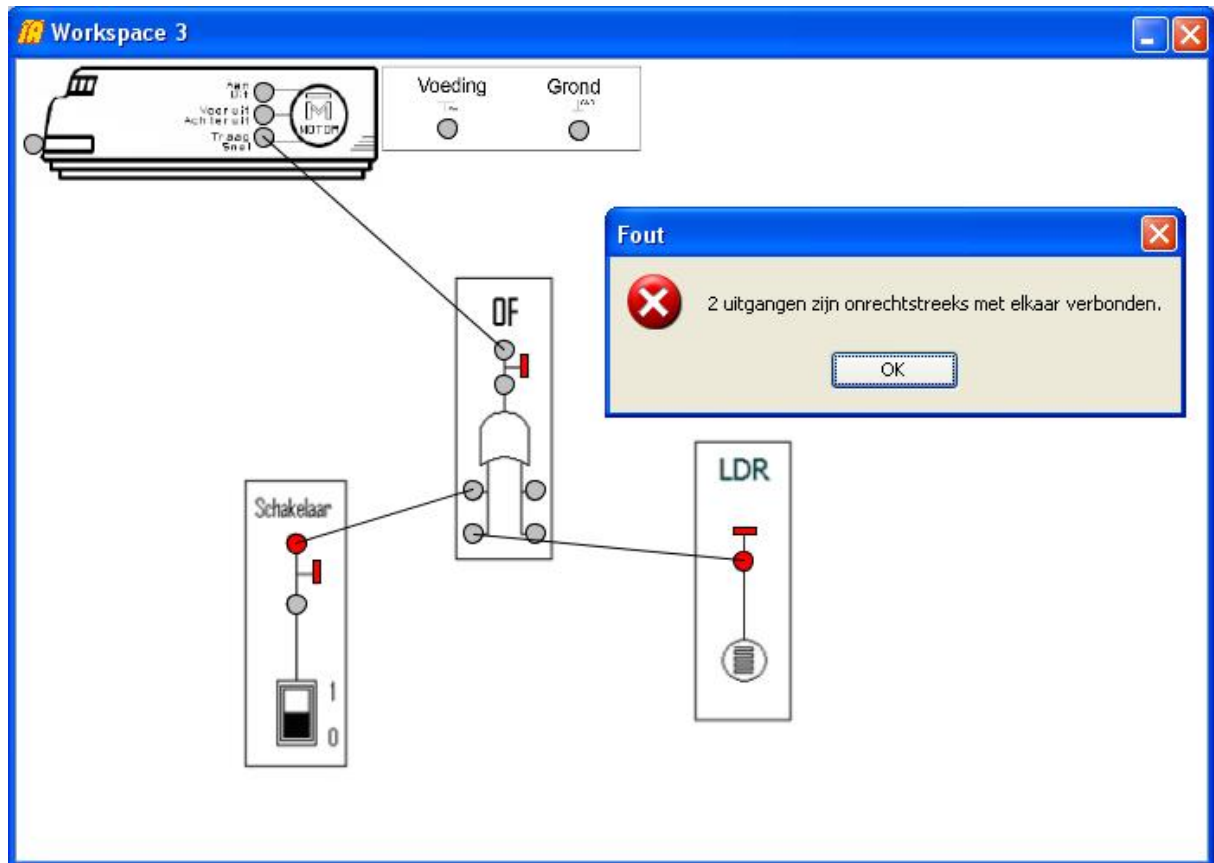
Twee uitgangen mogen nooit met elkaar verbonden worden. Indien de modules hier niet tegen beveiligd zijn, kan dit in werkelijkheid leiden tot beschadiging van de modules. Daarom is het belangrijk deze fout te vermijden bij het tekenen van een schakeling. Figuur 17 geeft een voorbeeld van deze foutmelding. De uitgangen die met elkaar verbonden zijn, worden in het rood aangegeven.



Figuur 17: Foutmelding voor het verbinden van 2 uitgangen

#### 4 Twee uitgangen onrechtstreeks met elkaar verbinden

Twee uitgangen kunnen eveneens via andere connecties onrechtstreeks met elkaar verbonden zijn. Het programma zal dit detecteren en verbieden. Figuur 18 verduidelijkt deze fout. De uitgang van de schakelaar is hier verbonden met ingang A van de OF-poort. Deze ingang is via zijn andere connectie op zijn beurt verbonden met de uitgang van de lichtsensor. Dit betekent echter dat de uitgang van de lichtsensor en de uitgang van de OF-poort onrechtstreeks zijn verbonden. Ook hier lichten de verbonden uitgangen op.



Figuur 18: Foutmelding voor het onrechtstreeks verbinden van 2 uitgangen

## Bijlage C: XML-sjabloon

Voor elk van de mogelijke modules is er een XML-bestand gemaakt. Hierin kunnen alle gegevens worden opgeslagen op een overzichtelijke en makkelijk leesbare wijze. Op deze manier verkrijgen we eveneens een structuur waarin eenvoudig aanpassingen kunnen doorgevoerd worden. Bovendien moet het programma niet hercompileren nadat er wijzigingen zijn aangebracht.

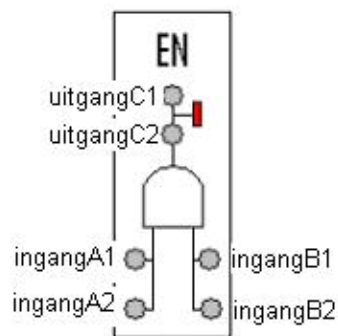
In onderstaand XML-sjabloon zien we een duidelijke hiërarchie die we nader zullen uitleggen. Alle delen tussen de symbolen `<` en `>` zijn zogenaamde *tags*. Het openen van een object gebeurt aan de hand van deze *tags*. In deze *tag* wordt tevens een naam gegeven aan een object. Het einde van elk object wordt aangegeven met `</` gevolgd door de objectnaam en `>`. Zoals zichtbaar in het sjabloon is een module hiërarchisch opgebouwd in verscheidene niveaus.

Op het hoogste niveau staat een *tag* voor de module, deze is altijd ‘module’ genaamd. Vervolgens komen dan de aansluitingen van de uitgangen, verzameld in een groter geheel genaamd outputs. Elke uitgang, aangegeven met de *tag* output, bevat verder nog enkele kenmerken. Zo zal een uitgang een naam krijgen en wordt zijn positie aangegeven, relatief ten opzichte van de linkerbovenhoek. Daarna moeten op dezelfde manier alle ingangen worden opgesomd, met als verzamelnaam inputs.

De volgende 3 kenmerken van de module zijn nodig voor het plaatsen van de led. Ook hier dienen de posities aangegeven te worden. Tevens wordt er vermeld als de led al dan niet horizontaal of verticaal staat. Dit wordt bereikt door de *tag* vertical, met true indien het verticaal staat en false voor een horizontaal geplaatste led. Hierna volgt dan de plaatsen waar de afbeeldingen zijn opgeslagen. De *tag* imageIcon duidt op de afbeelding die in de lijst zal getoond worden. Terwijl de *tag* image aangeeft waar de afbeelding van de module staat die gebruikt wordt bij het tekenen van de schakeling. En als laatste kenmerk is er de naam van de module. Indien sommige parameters niet nodig zijn voor een module, mogen deze altijd weggelaten worden.

Het is belangrijk een juiste naam te geven aan de in- en uitgangen. Dit is vereist voor het correct afhandelen van de foutdetectie bij het tekenen van een schakeling. Een ingang moet altijd met 'ingang' beginnen, gevolgd door een letter toegekend aan de ingang. Deze letter is de signaalnaam en geeft aan om welke ingang van de module het gaat. Indien er meerdere connecties per ingang mogelijk zijn, moet er achter de signaalnaam een cijfer gezet worden. Dit om onderscheid te kunnen maken tussen de connecties met dezelfde signaalnaam.

De naam van de uitgang is analoog aan deze van de ingang. In tegenstelling tot de ingang moet de naam nu natuurlijk met 'uitgang' beginnen, gevolgd door de signaalnaam. Ook hier moeten connecties met dezelfde signaalnaam gevolgd worden door een nummer. Figuur 19 verduidelijkt dit voor een EN-poort. Figuur 20 is de XML voor deze EN-poort.



**Figuur 19: benaming connecties**

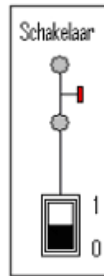
```

<module>
  <outputs>
    <output>
      <name>uitgangC1</name>
      <posX>26.5</posX>
      <posY>39.5</posY>
    </output>
    <output>
      <name>uitgangC2</name>
      <posX>26.5</posX>
      <posY>59.5</posY>
    </output>
  </outputs>
  <inputs>
    <input>
      <name>ingangA1</name>
      <posX>7.5</posX>
      <posY>121.5</posY>
    </input>
    <input>
      <name>ingangA2</name>
      <posX>7.5</posX>
      <posY>147.0</posY>
    </input>
    <input>
      <name>ingangB1</name>
      <posX>46.0</posX>
      <posY>121.5</posY>
    </input>
    <input>
      <name>ingangB2</name>
      <posX>46.0</posX>
      <posY>147.0</posY>
    </input>
  </inputs>
  <ledx>42</ledx>
  <ledy>48</ledy>
  <vertical>true</vertical>
  <imageIcon>figures/lijs/EN-poort.jpg</imageIcon>
  <image>figures/workspace/EN-poort.jpg</image>
  <name>EN-poort</name>
</module>

```

**Figuur 20: XML voor een en-poort**

Bij de simulatie is het mogelijk dat de gebruiker op de module moet klikken om een event te starten. Zo zal de gebruiker bijvoorbeeld de schakelaar aan of uit kunnen zetten. Indien dit het geval is, zijn er nog een aantal extra parameters nodig in het XML-bestand. Deze zijn eventX, eventY, eventHeight en eventWidth. EventX en eventY zijn de x- en y-coördinaten van het gebied dat men kan aanklikken. Terwijl eventWidth en eventHeight de breedte en hoogte voorstellen van het betreffende gebied. Figuur 21 geeft als voorbeeld een schakelaar waar tijdens de simulatie een muisklik de schakelaar kan verzetten.



**Figuur 21: Afbeelding van een schakelaar**

Het XML-bestand voor deze schakelaar is weergegeven in figuur 22. Hierin is te zien dat het aanklikbare gebied gelegen is op de coördinaten (22,119). Deze coördinaten zijn relatief ten opzichte van de linkerbovenhoek van de module. De breedte bedraagt 23 en de hoogte 37.

```
<module>
  <outputs>
    <output>
      <name>uitgangA1</name>
      <posX>27.0</posX>
      <posY>33.0</posY>
    </output>
    <output>
      <name>uitgangA2</name>
      <posX>27.0</posX>
      <posY>68.0</posY>
    </output>
  </outputs>
  <inputs>
  </inputs>
  <ledx>42</ledx>
  <ledy>48</ledy>
  <vertical>true</vertical>
  <eventX>22</eventX>
  <eventY>119</eventY>
  <eventWidth>23</eventWidth>
  <eventHeight>37</eventHeight>
  <imageIcon>figures/lijst/schakelaar.jpg</imageIcon>
  <image>figures/workspace/schakelaar.jpg</image>
  <name>schakelaar</name>
</module>
```

**Figuur 22: XML voor een schakelaar**

## Bijlage D: Configuratiemodulesjabloon

De gegevens die het programma inleest van de XML-bestanden, plaatst het in een configuratieobject. Dit object is algemeen voor alle modules en is maar eenmaal nodig. Figuur 23 is de javacode voor dit object. Het is duidelijk dat alle parameters die in het XML-bestand staan ook hier zijn terug te vinden. Xstream plaatst ze namelijk in dit object bij het inlezen. Ook alle methodes om deze variabelen op te vragen, zijn in dit object voorzien.

In Figuur 23 is eveneens te zien dat de variabelen eventX, eventY, eventWidth en eventHeight in deze ConfModule aanwezig zijn. Deze variabelen worden echter enkel ingelezen als ze ook daadwerkelijk in het XML-bestand aanwezig zijn. Dus ze krijgen enkel een waarde toegewezen indien er interactie met de gebruiker mogelijk is bij de simulatie.



```

package be.khlim.trein.modules.conf;

import java.util.ArrayList;

public class ConfModule {

    private ArrayList<ConfOutput> outputs; // all outputs
    private ArrayList<ConfInput> inputs; // all inputs
    private float ledx, ledy;
    private String image, imageIcon, name;
    private boolean vertical;
    private float eventX, eventY, eventHeight, eventWidth;

    public ConfModule(){

    }

    public void addName(String nm){
        name = nm;
    }

    public void addImagePath(String im){
        image = im;
    }

    public void addImageIcon(String ic){
        imageIcon = ic;
    }

    public void addVertical(boolean vert){
        vertical = vert;
    }

    public ArrayList<ConfOutput> getOutputs(){
        return outputs;
    }

    public ArrayList<ConfInput> getInputs(){
        return inputs;
    }

    public String getImage(){
        return image;
    }

    public String getImageIcon(){
        return imageIcon;
    }

    public String getName(){
        return name;
    }
}

```

```

    public float getLedX(){
        return ledx;
    }

    public float getLedY(){
        return ledy;
    }

    public boolean getVertical(){
        return vertical;
    }

    public void setEventX(float x){
        eventX = x;
    }

    public void setEventY(float y){
        eventY = y;
    }

    public void setEventWidth(float w){
        eventWidth = w;
    }

    public void setEventHeight(float h){
        eventHeight = h;
    }

    public float getEventX(){
        return eventX;
    }

    public float getEventY(){
        return eventY;
    }

    public float getEventWidth(){
        return eventWidth;
    }

    public float getEventHeight(){
        return eventHeight;
    }
}

```

**Figuur 23: Javacode van de klasse ConfModule**

## Bijlage E: Modulesjabloon

De module is een javaklasse waarin het gedrag en de eigenschappen van de modules beschreven zijn. Deze klasse erft over van een algemene klasse `Module`. Deze klasse zorgt voor het aanmaken van de in- en uitgangen en de juiste afbeeldingen. Ook de naam en de led wordt hierin aangemaakt. Hiervoor leest de module de juiste configuratie uit de configuratiebestanden. Indien we dus een nieuwe module willen toevoegen, moet deze overerven van `Module.java`. Hierbij kan ook de constructor van deze klasse overgenomen worden. In het geval van tijdsafhankelijke modules zoals de wachtmodule, is er een ander principe van toepassing. Meer uitleg hierover staat in bijlage F.

Afhankelijk van het gedrag van de module in de simulatie, is het nodig enkele methodes die in `Module.java` voorzien zijn te *overwriten*. Een methode die altijd geïmplementeerd moet worden, is `process(Event evt)`. Deze methode zorgt immers voor de juiste verwerking van de module en past indien nodig de in- en uitgangen aan. Andere mogelijke methodes om toe te voegen zijn `addListeners()`, `colorNodes()` en `initialize()`. In de volgende paragrafen zullen deze methodes besproken worden.

### 1 De methode `process(Event e)`

De methode `process(Event evt)` is de voornaamste methode van een module. Het programma roept deze methode op als het bij de simulatie een event is gestuurd naar één van de ingangen van de module. Dit event wordt dan als parameter meegegeven. Figuur 24 geeft de code voor een EN-poort. Bij deze module is het enkel nodig de methode `process(Event evt)` toe te voegen. We leggen hieronder kort enkele methodes uit die veel gebruikt worden in de verwerking van de meeste modules. Tevens tonen we hoe ze in de EN-poort gebruikt worden.

In de methode `process(Event evt)` is te zien dat er eerst gecontroleerd wordt als er door het event wel een wijziging van de ingang plaatsvindt. Indien dit niet het geval is, hoeft de verwerking niet te gebeuren. Deze controle gebeurt met de volgende if-lus.

```
if(! evt.getInput().getState() == evt.getInput().mappedto().getState())
```

Hierin is te zien dat het mogelijk is om uit het event de ingang waarop het event binnenkomt, op te halen met `getInput()`. Vanuit deze ingang is het dan mogelijk om met `mappedto()` de connectie te verkrijgen waarmee hij verbonden is. Met `getState()` wordt dan opgevraagd wat de huidige status van de ingang is. Deze status is altijd `true` of `false`, overeenstemmend met 1 of 0.

Indien eenzelfde ingang meerdere connecties heeft, hebben deze connecties dezelfde signaalnaam. Dit is een letter die toegekend wordt aan elke in- of uitgang. Zo is het eenvoudig om de connecties horend bij een bepaalde ingang van de module op te vragen. Een event binnenkomend op een connectie van een ingang kan de waarde van deze ingang wijzigen. Vervolgens moeten alle connecties van deze ingang op de nieuwe waarde gezet worden. Dit gebeurt met de methode `setInputs(Input in)`. Bij de EN-poort gebeurt dit op de volgende manier:

```
setInputs(evt.getInput());
```

De methode `writeOutputs(boolean st, char sig)` schrijft de meegegeven waarde `st` naar alle connecties van de uitgang met als signaalnaam `sig`. Indien één van deze uitgangen verbonden is met een ingang, zal deze methode een nieuw event genereren op deze ingang. Zo is de verwerking van deze module afgelopen. Als we opnieuw het voorbeeld van de EN-poort bekijken, geeft dit de volgende regel.

```
writeOutputs(inA && inB, outputs.get(0).getSignal());
```

Met `getSignal()` verkrijgt men de signaalnaam van een in- of uitgang. Een EN-poort heeft maar één uitgang met twee connecties. Dus het is hierbij voldoende om de signaalnaam van een van de twee connecties van de uitgang mee te geven. Figuur 24 geeft dan de volledige klasse voor de EN-poort.

```
package be.khlim.trein.modules;
import be.khlim.trein.gui.Event;
import be.khlim.trein.modules.conf.*;

public class EnPoort extends Module {

    public EnPoort(ConfigModule cmod){
        super(cmod);
    }
    public void process(Event evt){
        if(! evt.getInput().getState() ==
            evt.getInput().mappedto().getState()){
            setInputs(evt.getInput());
            boolean inA = false, inB = false;
            inA = evt.getInput().getState();
            for(Input in: inputs){
                if( !(evt.getInput().getSignal() ==
                    in.getSignal())){
                    inB = in.getState();
                }
            }
            writeOutputs(inA && inB, outputs.get(0).getSignal());
        }
    }
}
```

**Figuur 24: Javacode van de klasse EnPoort**

## 2 De methode *addListeners()*

Indien er bij de simulatie interactie mogelijk is met de gebruiker, moet de methode *addListeners()* worden toegevoegd aan de moduleklasse. Ook de constructor van de module verandert in dit geval. Nu moeten namelijk de extra variabelen *eventX*, *eventY*, *eventHeight*, *eventWidth* ingelezen worden uit de configuratiemodules. Dit gebeurt met de methode *setEventFrame(ConfigModule cmod)*. Figuur 25 geeft de constructor voor een lichtsensord.

```
public Lichtsensor(ConfigModule cmod){  
    super(cmod);  
    for(Output out : getOutputs()){  
        out.setDelay(0L);  
    }  
    setEventFrame(cmod);  
}
```

Figuur 25: Constructor van de klasse *Lichtsensor*

Hierin is ook te zien dat de methode *setDelay(Long l)* de vertraging van de ingangen naar de uitgangen op nul zet. Dit is nodig omdat de lichtsensord geen ingangen heeft en ze dus geen vertraging veroorzaakt. Deze methode wordt meestal toegevoegd in de constructor indien de methode *addListeners* aanwezig is in de module. Aangezien er meestal geen ingangen aanwezig zijn wanneer er gebruikersinteractie mogelijk is.

De methode *addListeners()* maakt het aanklikbaar gebied aan en voegt de nodige *listeners* toe. Bij de lichtsensord is dit een *listener* voor als de muisknop wordt ingedrukt. De javacode hiervoor is weergegeven in figuur 26. Aan de hand van de waarde van de uitgang past deze *listener* de afbeelding aan en voert de methode *process(null)* uit. Ook de in- en uitgangen moeten opnieuw gekleurd worden. vervolgens verwerkt de methode *simulate()* de gegenereerde events.

```

public void addListeners(){
    PPath p = new PPath();
    p.setPathToEllipse(eventX, eventY, eventWidth, eventHeight);
    p.setPaint(Color.white);
    p.setTransparency(0);
    setImage(image.replaceAll(".j", "_on.j"));
    setBounds(getX(),getY(),65,170);
    p.addInputEventListener(new PBasicInputEventHandler() {

        public void mousePressed(PInputEvent e){
            if(!outputs.get(0).getState()){
                setImage(image.replaceAll(".j", "_on.j"));
                setBounds(getX(),getY(),65,170);
            }
            else{
                setImage(image);
                setBounds(getX(),getY(),65,170);
            }
            process(null);
            colorNodes();
            simulate();
            e.setHandled(true);
        }
    });
    addChild(p);
}

```

**Figuur 26: Javacode voor de methode addListerner van een lichtsensor**

### 3 De methode initialize()

De methode initialize past de waarden aan van de uitgangen bij de start van de simulatie. Onder andere bij de lichtsensor is dit van toepassing. De uitgangen worden hier immers op 1 gezet omdat er wordt verondersteld dat er lichtinval is. De methode initialize( ) staat in figuur 27.

```

public void initialize(){
    writeOutputs(true, outputs.get(0).getSignal());
}

```

**Figuur 27: Javacode voor de methode initialize van een lichtsensor**

De methode writeOutputs() zet de uitgangen op 1 en schrijft een event naar de ingangen die met deze uitgangen verbonden zijn. Deze events worden dan verder afgehandeld zodat de schakeling volledig geïnitieerd is.

#### 4 De methode *colorNodes()*

Deze methode kleurt de in- en uitgangen aan de hand van hun waarde. De led wordt standaard gekleurd aan de hand van de uitgang. Indien dit niet het geval is, is het nodig om ook deze methode te voorzien in de module. De zoemer bijvoorbeeld kleurt zijn led aan de hand van de waarde van zijn ingang en wijkt dus af van de standaard. Voor de zoemer wordt deze methode dan zoals getoond in figuur 28.

```
public void colorNodes(){
    for(Input in: inputs){
        if(in.getState()){
            in.setPaint(Color.red);
            led.setPaint(Color.red);
        }
        else{
            in.setPaint(Color.black);
            led.setPaint(Color.black);
        }
    }
}
```

Figuur 28: Javacode van de methode *colorNodes* van een zoemer

## Bijlage F: Tijdsafhankelijke modules

Voor de modules waar de tijd een belangrijke rol speelt, gelden afwijkende regels. Het is niet mogelijk om deze modules op te bouwen op dezelfde wijze als de andere modules. Voor de correcte verwerking bij de simulatie is dan een nieuwe klasse aangemaakt. De klasse `Wait` zal een vooraf bepaalde tijd wachten. Vervolgens ontvangt de klasse van de module een signaal dat de tijd is verstreken. De uitgangen van de module zullen zich dan aanpassen en indien nodig *events* versturen naar de verbonden modules. Zowel de wachtmodule als de klok maken gebruik van dit principe.

Figuur 29 bevat de code van de `Wait` klasse. Het eerste attribuut `wait` dient om de tijd aan te geven hoe lang er gewacht moet worden. Vervolgens hebben we een verwijzing nodig van de module die de klasse `Wait` oproept. Het laatste attribuut dat wordt gebruikt, is de status waarin de uitgangen gewijzigd moet worden. Wanneer de klasse wordt gestart, zal er voor de aangegeven tijd niets gebeuren. Daarna zorgt de klasse voor het aanpassen van de uitgangen en het genereren van de *events*. Bovendien moet het scherm opnieuw moeten hertekend om de veranderingen zichtbaar te maken.

```
package be.khlim.trein.modules;

import java.io.Serializable;

public class Wait extends Thread implements Serializable {

    long wait;
    Module module;
    boolean state;

    public Wait(long w, boolean st, Module mod){
        wait = w;
        module = mod;
        state = st;
    }

    public void run(){
        try{
            sleep(wait);
            module.writeOutputs(state,
                                module.getOutputs().get(0).getSignal());
            module.colorNodes();
            module.simulate();
            module.sim().getLayer().repaint();
        } catch (InterruptedException e) { }
    }
}
```

Figuur 29: Javacode van de klasse `Wait`



Om gebruik te maken van deze klasse `Wait`, zijn er slechts 2 regels code nodig in de `process` methode. Eerst moet een object van de klasse `Wait` worden aangemaakt. Om dit te realiseren dien je de tijd die gewacht moet worden reeds te bepalen. Als nu de klasse wordt gestart, zal er een nieuwe *thread* starten. Op deze manier is het wachten onafhankelijk van het hoofdprogramma. De *thread* zal dan de ingestelde tijd stilgelegd worden. Hierna worden de uitgangen veranderd en de *events* gegenereerd. In het geval van de klok zal hierop volgend opnieuw de klasse `Wait` worden aangesproken. Figuur 30 geeft de code weer voor de `process` methode in het geval van de wachtmodule.

```
public void process(Event evt){
    if(! evt.getInput().getState() ==
        evt.getInput().mappedto().getState()){
        setInputs(evt.getInput());

        Wait wait = new Wait(outputs.get(0).getDelay(),
                               evt.getInput().getState(), this);
        wait.start();
    }
}
```

**Figuur 30: Javacode voor de methode `process` van een wachtmodule**