2021•2022
Faculteit Industriële Ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis
Codegenerator van SceneBuilder-FXML naar Haskell

PROMOTOR :
Prof. dr. Kris AERTS
PROMOTOR :
Dhr. Wouter GROENEVELD

Kai Broux, Luigi Guerriero
Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Gezamenlijke opleiding UHasselt en KU Leuven

2021•2022

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Codegenerator van SceneBuilder-FXML naar Haskell

PROMOTOR :

Prof. dr. Kris AERTS

PROMOTOR :

Dhr. Wouter GROENEVELD

Kai Broux, Luigi Guerriero

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Woord vooraf

Deze masterproef is de laatste stap om af te studeren als industriële ingenieurs in de elektronica-ict aan de UHasselt en KU Leuven. Het onderwerp van de masterproef bevindt zich meer in de softwarewereld, waar wij beide ook meer interesse in tonen om mee verder te gaan. De masterproef vormde een grote uitdaging, maar de combinatie van het interessante onderwerp en deze grote uitdaging zorgde ervoor dat er veel is bijgeleerd. Deze kennis en ervaring nemen we mee naar de toekomst toe. Het is vanzelfsprekend dat deze masterproef niet was gelukt zonder enkele personen. We maken dan ook graag gebruik van deze gelegenheid om een aantal personen te bedanken.

Om te beginnen willen we onze interne begeleider, Prof. dr. Kris Aerts, bedanken voor zijn begeleiding tijdens de gehele masterproef. We konden telkens rekenen op zijn wijze raad, opvolging, enthousiasme en kritische blik op het project. Daarnaast stond hij ook telkens spoedig klaar om ons te helpen met moeilijkheden en om ons uitgebreide feedback te geven.

Ook willen we onze externe begeleider, Wouter Groeneveld, bedanken voor het kritisch reflecteren van enkele keuzes die wij hebben gemaakt in onze masterproef. Daarnaast appreciëren wij ook zijn tijd en inspanning om deze masterproef tot een succes te helpen.

Bovendien willen wij de docenten van de opleiding van industrieel ingenieur (elektronica-ict) bedanken. Door hun kennis, werkervaring, manier van lesgeven en enthousiasme hebben wij niet alleen deze masterproef succesvol kunnen afronden, maar ook de hele vier jaar durende opleiding.

Verder willen wij nog onze familie en vrienden bedanken. Gedurende onze hele opleiding stonden zij klaar voor ons met hun steun en motivatie. Vooral in zware periodes bood dit een houvast en daarom was deze masterproef ook niet zonder hun gelukt.

Tot slot willen we elkaar nog bedanken. Deze masterproef was een genoegen om samen te maken. De samenwerking tussen elkaar verliep ten allen tijden vloeiend en probleemloos. We konden op elkaar rekenen en ondersteunden elkaar indien nodig.

Kai Broux en Luigi Guerriero
juni 2022
Diepenbeek

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Verklarende woordenlijst

Binding	= Een vertaling van een informatiemodel naar een technische implementatie.
Callback	= Code die meegegeven wordt aan een ander codeblok om uit te voeren wanneer er een gebeurtenis plaats vindt binnen de tweede codeblok.
Enum	= Een enum is een speciale klasse die een groep constanten vertegenwoordigt (onveranderlijke variabelen).
Hashcode	= Een hashcode is een (numerieke) waarde die wordt gebruikt om een object te identificeren.
Interface	= Koppeling tussen twee programma's, computersystemen of tussen een gebruiker en een programma. In die context meestal specifiek benoemd als User Interface”
(Event) Listener	= Een procedure of functie in een computerprogramma die wacht tot een event optreedt.
Node	= Een node is een entiteit waaruit een boomstructuur bestaat
Out-of-the-box	= Een term voor producten die geen tot weinig installatie behoeven.
Overhead	= Elke combinatie van extra of indirecte rekentijd, geheugen, bandbreedte, of andere middelen die nodig zijn om een specifieke taak uit te voeren.
Parser	= Een parser is een computerprogramma dat de grammaticale structuur van een invoer volgens een vastgelegde grammatica ontleedt.
Pattern-matching	= Het controleren en lokaliseren van specifieke reeksen van data van een bepaald patroon in ruwe data of een reeks tokens.
Placeholder	= Een placeholder is een teken in een String die de tijdelijke plaatst inneemt van definitieve gegevens.
Toolkit	= Hulpmiddelen die helpen programmeren met een bepaalde programmeertaal.
Widget	= Een widget is de samentrekking van **Wi**ndows en ga**dget** en is de algemene naam voor een van de elementen waaruit een grafische gebruikersomgeving kan worden opgebouwd.
Wrapper	= Softwaretool die opereert tussen een oorspronkelijk bestandsformaat en een doelformaat.

Abstract

FunTTop te Diepenbeek heeft als doel het gebruik van functioneel programmeren in het werkveld te verhogen om zo de voordelen op het gebied van efficiëntie en het aantal regels code t.o.v. imperatieve programmeertalen te kunnen valoriseren. Tegelijkertijd stelt FunTTop vast dat het programmeren van een grafische user interface (GUI) nog steeds omslachtig is in Haskell, de academisch ontwikkelde functionele taal. Om die reden beoogt deze masterproef een codegenerator te ontwikkelen die de output van SceneBuilder (FXML) kan omzetten naar Haskell-code.

De moeilijkheden om een codegenerator te ontwikkelen zijn tweedelig. Eerst moet een GUI-bibliotheek voor Haskell gekozen worden die toekomstbestendig is. Het is immers de bedoeling om een beperkte FXML-subset van widgets te selecteren die later uitbreidbaar is. Daarnaast is het vereist dat de codegenerator zelf het FXML-bestand leest, de gelijkaardige GUI-elementen herkent en de Haskell-code genereert.

Een lijst van GUI-bibliotheken werd gekozen op basis van een literatuurstudie. Hierna werd voor iedere bibliotheek een GUI-applicatie nagemaakt. Hieruit werd gi-gtk uiteindelijk geselecteerd. Voor de codegenerator zelf is vertrokken van een functie die de FXML-boomstructuur overloopt en analyseert. Daarna werden voor de overeenkomstige GUI-elementen uit FXML en Haskell zelfgemaakte datastructuren aangemaakt die in een laatste stap Haskell-code genereren. De ontwikkelde codegenerator kan op die manier met precisie een twaalftal FXML-elementen herkennen en omzetten.

Abstract in English

FunTTop in Diepenbeek aims to increase the use of functional programming in the field to be able to valorize the advantages in terms of efficiency and the number of lines of code compared to imperative programming languages. At the same time, FunTTop finds that programming a graphical user interface (GUI) is still cumbersome in Haskell, the academically developed functional language. Therefore, this master's thesis aims to develop a code generator that can convert the output of SceneBuilder (FXML) to Haskell code.

The difficulties of developing a code generator are twofold. First, a GUI library for Haskell has to be chosen that is future-proof. After all, the intent is to select a limited FXML subset of widgets that can be expanded later. In addition, the code generator itself requires a level of efficiency that allows to read the FXML file, to recognize the similar GUI elements, and to generate the Haskell code.

A list of GUI libraries was chosen based on a literature study. After this, a demo GUI application was recreated for each library. From these, gi-gtk was eventually selected. The FXML hierarchy is run through and analyzed by a function of the code generator. Afterwards, custom data structures are created for the corresponding GUI elements from FXML and Haskell, which generate Haskell code in a final step. The developed code generator can thus recognize and convert a dozen FXML elements with precision.

1 Inleiding

1.1 Situering en probleemstelling

Het tempo waarop programmeertalen evolueren, is exemplarisch voor de snelheid waaraan de informaticawereld verandert. Het concept van functioneel programmeren is een voorbeeld van een academisch concept dat de laatste jaren ingang vond in *mainstream* programmeertalen. De onderzoeksgroep FunTTop van KU Leuven, campus Diepenbeek, is hiervan een pleitbezorger, maar stelt tegelijkertijd vast dat het programmeren van een grafische user interface (GUI) nog steeds vrij omslachtig is in Haskell, de academisch ontwikkelde, maar ook commercieel gebruikte functionele programmeertaal. Bovendien wordt dit meestal op een niet zo declaratieve wijze beschreven.

Aan de andere kant van het spectrum staat de programmeertaal Java waar JavaFX recent gelabeld werd als de aangeraden user interface bibliotheek doordat JavaFX applicaties op meerdere platforms werken en eenvoudig te ontwikkelen zijn [1]. Zo is het mogelijk om met SceneBuilder op een heel eenvoudige manier een user interface te bouwen door de benodigde elementen rechtstreeks te manipuleren met de muis. In de gezamenlijke opleiding industriële wetenschappen aan UHasselt/KU Leuven op campus Diepenbeek wordt deze aanpak om GUIs te ontwikkelen al verschillende jaren met succes toegepast in het eerste bachelorjaar.

JavaFX is een module geschreven in Java bovenop de Java Development Kit (JDK). Java op zich is niet altijd de meest efficiënte taal qua systeembronnen en energieconsumptie [2], [3]. Daarom is het interessant om een alternatief platform en/of programmeertaal te zoeken die de met-gebruiksgemak gegenereerde FXML van SceneBuilder kan omzetten naar een GUI-applicatie. Een kandidaat-programmeerparadigma om dit te realiseren is het functioneel programmeren, en meer bepaald de programmeertaal Haskell. Dit is de premisse van de onderzoeksgroep FunTTop waarbinnen deze masterproef zich situeert. FunTTop heeft immers als doelstelling om het gebruik van functioneel programmeren in het werkveld te verhogen en van Haskell, de academische functionele taal, in het bijzonder.

FunTTop staat niet alleen in deze visie. Verschillende grote bedrijven maken al gebruik van Haskell omdat het toestaat om code sneller en correcter te ontwikkelen, meer programma-eigenschappen verifieerbaar zijn, de foutenlast kleiner is en er minder regels code nodig zijn, maar ook omwille van de kracht waarmee programma's in Haskell parallel uitgevoerd worden. Dit leidt ertoe dat bedrijven die hun servers efficiënter willen gebruiken, functionele talen zoals Haskell gebruiken voor efficiënte parallelle gegevensverwerking op hun servers. Facebook is zo een voorbeeld [3], [4]. Ondertussen hebben reguliere programmeertalen, zoals Java en C#, functionele programmeerfuncties zoals lambda's overgenomen omdat functionele talen talrijke voordelen bieden die later besproken zullen worden [5].

1.2 Doelstellingen

De opdracht voor deze masterproef bestaat erin om een codegenerator te ontwikkelen die Haskell-code genereert op basis van de FXML-output van SceneBuilder, een GUI-tool waarbij ontwikkelaars eenvoudig hun GUI-design kunnen beschrijven. De GUI-applicatie in Haskell die door de

codegenerator wordt gegenereerd, zal visueel zo identiek mogelijk moeten zijn aan de JavaFX implementatie.

De codegenerator vereist hierbij een niveau van uitbreidbaarheid en een efficiënte methode om het FXML-bestand te lezen, de overeenkomstige GUI-elementen te herkennen en de Haskell-code te genereren.

Buiten de codegenerator zelf, is de bibliotheekkeuze voor GUIs een belangrijk deel van de masterproef. Er bestaan immers al een redelijk aantal GUI-bibliotheken en het is niet de bedoeling om op dat punt het wiel opnieuw uit te vinden. De tool moet bovenop een bestaande bibliotheek werken die actief onderhouden wordt. De gekozen bibliotheek vereist daarom ook een niveau van uitbreidbaarheid en gebruiksgemak. Sowieso is het in het tijdsbestek van deze masterproef niet mogelijk om de volledige subset van FXML-Widgets over te nemen, maar de gekozen bibliotheek moet wel intrinsiek die mogelijkheid bieden.

1.3 Overzicht

In het hoofdstuk “2 Literatuurstudie” worden enerzijds de eigenschappen van JavaFX en de SceneBuilder onderzocht, en anderzijds die van Haskell en enkele belangrijke GUI-bibliotheken van Haskell. Na een eerste technologiescan, worden een viertal frameworks/bibliotheken geselecteerd.

In het hoofdstuk “3 Bibliotheekkeuze” wordt, op basis van de gekozen bibliotheken, éénzelfde voorbeeld van een GUI-applicatie uitgewerkt. Bovendien wordt hierbij iedere bibliotheek in detail bestudeerd en uiteindelijk wordt de keuze voor de bibliotheek voor de codegenerator toegelicht.

In het hoofdstuk “4 Codegenerator” wordt vervolgens een subset van JavaFX elementen geselecteerd die de codegenerator moet genereren. Tevens wordt de methode van het ontwikkelen van de codegenerator ook overlopen. Het doel van de codegenerator is hierbij enerzijds een beperkte subset van JavaFX-GUI-elementen apart na te bouwen. Anderzijds is het doel om een stevige basis voor de implementatie van hiërarchie tussen de GUI-elementen te realiseren om uiteindelijk de voorbeeld-GUI-applicatie na te bouwen van hoofdstuk “3 Bibliotheekkeuze”. Pas daarna wordt de JavaFX-subset uitgebreid met extra elementen.

In hoofdstuk “5 Resultaten en discussie” worden de resultaten besproken. Hierbij worden de GUI-elementen apart getest, alsook de volledige voorbeeld-GUI-applicatie van hoofdstuk “3 Bibliotheekkeuze” en een voorbeeld-applicatie waar alle GUI-elementen in voorkomen. Ook wordt er in dit hoofdstuk kritisch gereflecteerd over de huidige gebreken van de implementatie van de codegenerator en de design-keuzes die zijn genomen. Eventueel toekomstig werk wordt verder in dit hoofdstuk besproken.

Ten slotte volgt een besluit in het hoofdstuk “6 Conclusie” die een samenvatting van het werk van de masterproef overloopt.

2 Literatuurstudie

2.1 JavaFX en SceneBuilder

JavaFX is een softwareplatform om (desktop)applicaties met een GUI te ontwikkelen die ondersteund worden door verschillende besturingssystemen waaronder: Windows, Linux, MacOS, iOS en Android [6]. Sinds JavaFX 2.0 is JavaFX grootschalig beschikbaar voor programmeertalen die op de Java Virtual Machine (JVM) draaien zoals Java, Groovy en JRuby [1]. De JVM is hierbij een platformonafhankelijke omgeving voor het uitvoeren van Java Bytecode, een gedeeltelijke compilatie van Java-code [7].

Een GUI wordt in JavaFX gedefinieerd als een boomstructuur van JavaFX-*nodes*. Een *node* wordt in de informatica beschreven als een onafhankelijke eenheid van een groter geheel. In de context van JavaFX worden *nodes* gezien als visuele componenten van de GUI [8]. Deze *nodes* worden recursief in een container geplaatst, waarbij de top-level container de *scene* is. De JavaFX-*nodes* worden ofwel manueel geprogrammeerd, ofwel beschreven in een XML gemarkeerd tekstbestand van het type FXML. FXML-bestanden worden gemaakt en onderhouden met een GUI-tool, genaamd SceneBuilder waarbij ontwikkelaars eenvoudig hun GUI-design beschrijven waarbij automatisch FXML gegenereerd wordt die gebruikt wordt om een JavaFX applicatie te ontwikkelen [2].

2.2 Waarom Haskell over Imperatieve talen

Haskell is een pure functionele programmeertaal waarbij het “functionele” betekent dat de bouwblokken van programma’s functies zijn. Een functie wordt als argument aan een andere functie doorgegeven, als resultaat geretourneerd of toegewezen aan een variabele. Dit vermogen om functies als gegevens te behandelen, zorgt voor een hoger abstractieniveau en dus meer mogelijkheden voor hergebruik. Het “pure” aspect van Haskell is dat er geen bijwerkingen zijn wanneer een functie geëvalueerd wordt. De uitkomst van een functie hangt alleen af van de parameters en iedere evaluatie van die functie zal hetzelfde resultaat geven [3], [9].

Dat Haskell functioneel en puur is, is een eerste significant voordeel vergeleken met traditionele imperatieve programmeertalen. Bij imperatieve programmeertalen worden programma’s beschreven als een sequentie van instructies die data muteren. Het muteren van data tussen de sequenties en functies van een imperatieve taal zorgt er soms voor dat bepaalde functies niet altijd hetzelfde resultaat teruggeven en daardoor onverwacht gedrag tonen [3].

Haskell is ook een *lazy* programmeertaal waarbij de *laziness* refereert naar het niet strikt evalueren, ook wel *call-by-need* genoemd. Een uitdrukking wordt nooit geëvalueerd totdat het nodig is voor de evaluatie van een volgende uitdrukking. *Laziness* zorgt voor een minimale hoeveelheid berekening die wordt uitgevoerd tijdens de uitvoering van het programma wat op gebied van prestatie een pluspunt is in vergelijking met imperatieve programmeertalen [3], [9].

Recursie is de norm van programmeerstijl voor Haskell. Recursie vervangt de reguliere iteratieve lussen die toestandsmutaties van variabelen vereisen en zorgt ervoor dat complexe code aanzienlijk korter wordt beschreven [3]. Gecombineerd met het feit dat Haskell functies gebruikt zonder bijwerkingen zorgt ervoor dat een robuust programma gerealiseerd wordt met weinig regels code vergeleken met imperatieve programmeertalen. Zo concludeerde de studie van [10] dat functionele talen zoals F# en Haskell in het algemeen beduidend beknopter waren. Haskell had volgens de studie

van [10] gemiddelde twee keer minder regels nodig voor dezelfde taken van *Rosetta Code* dan andere programmeertalen en tot drie keer minder regels code dan C.

2.3 Haskell is niet perfect

Hoewel Haskell een robuuste foutongevoelige taal is, schittert het niet op ieder aspect. Ten eerste zijn Haskell en functionele talen in het algemeen minder geheugenefficiënt vergeleken met talen zoals C en C++ [10]. Bovendien is Haskell niet altijd evident om aan te leren aangezien volgens [11] en [12] het concept van recursie en mathematische vaardigheid vereist is om Haskell op een vlotte manier te begrijpen. Daarnaast zijn de *compiler-warnings* en *-errors* niet altijd even duidelijk in Haskell [10]. Ten slotte blijft Haskell achter op het gebied van GUI-bibliotheken aangezien vele bibliotheken al afgeschreven zijn en GUIs ontwikkelen voor andere programmeertalen populairder is.

2.4 De wereld van GUIs in Haskell

In de loop der jaren zijn er talloze GUI-bibliotheken ontwikkeld voor Haskell, met een breed scala aan verschillende interfaces. Zo kent Haskell *high-level* (functioneel en declaratief) en *low-level* GUI-bibliotheken [13]. *Medium-level* bibliotheken worden eerder gezien als bibliotheken die functioneel zijn, maar toch een declaratieve smaak hebben.

Low-level bibliotheken en programmeertalen worden volgens Alan Perlis beschreven als “een programmeertaal waarbij de programma’s aandacht nodig hebben voor het irrelevante” [14, p. 42]. Het irrelevante wordt volgens [13] beschreven als alle implementatiedetails die zich verschuilen achter het abstracte conceptueel model. In de realiteit van de wereld van Haskell GUIs zijn dit de bibliotheken die als *wrapper* toegepast zijn rond een bibliotheek die in imperatieve programmeertalen zoals C/C++ zijn geschreven [15]. In de context van programmeertalen en bibliotheken bevatten deze dunne *wrapper*s code die intern *application programming interface* (API) functies oproepen zonder de originele API van de onderliggende implementatie aan te passen. Enkele voorbeelden van deze bibliotheken zijn WxHaskell en GTK2Hs die toelaten om APIs van de onderliggende toolkit, resp. WxWidgets en GTK+, rechtstreeks op te roepen in Haskell op een eenvoudige, iets declaratievere manier dan het origineel.

High-level bibliotheken zorgen voor een duidelijke scheiding tussen de implementatie van de applicatie en de interface [16]. In de wereld van Haskell GUIs zijn deze type bibliotheken meestal gebaseerd op het *Functional Reactive Programming* (FRP) paradigma. Deze type bibliotheken staan eerder verder weg van het concept van typische *wrappers* die bij *low-level* bibliotheken gebruikt worden en staan meer bekend om de *reactive*, *event*-gedreven eigenschappen op een functionele manier te implementeren. Het *reactive*- en *event*-gedreven duidt in deze context aan dat het programma alleen specifieke code uitvoert die wordt bepaald door de sequentie van *events* (muisklik, venster vergroten, …) [17].

In de volgende paragraaf worden enkele populaire *low-*, *medium-* en *high-level* kandidaat-bibliotheken besproken met hun voor- en nadelen. Vervolgens worden de belangrijkste kandidaat-bibliotheken verder in detail getest en bestudeerd om uiteindelijk een keuze te maken voor de bibliotheek die gebruikt wordt voor de codegenerator.

2.5 Enkele populaire bestaande Haskell GUI bibliotheken.

2.5.1 WxHaskell

WxHaskell is een *medium-level* GUI-bibliotheek voor Haskell waarbij de bibliotheek zorgt voor de nodige *bindings* van Haskell naar WxWidgets, een cross-platform GUI-bibliotheek geschreven in C++. *Bindings* worden in deze context beschreven als een verbinding tussen een programmeertaal en softwarebibliotheek. Ongeveer 75% van WxWidgets functionaliteit is overgenomen in WxHaskell. Volgens [12] worden WxHaskell GUIs gedefinieerd met een imperatieve IO monad, die een "declaratieve smaak" heeft waardoor er toch een hoog niveau van abstractie wordt bereikt.

Een eerste voordeel van WxHaskell is dat de bibliotheek gebaseerd is op WxWidgets. WxWidgets wordt ondersteund door grote industriële spelers zoals *AVG*, *AMD*, *Lockheed* en *NASA* waardoor de bibliotheek onderhouden wordt indien er nieuwe platform-specifieke *features* worden uitgebracht [12], [18]. Een tweede voordeel van WxWidgets, en dus WxHaskell, is dat applicaties gemakkelijk omgezet worden naar verschillende besturingssystemen zonder de applicatie helemaal opnieuw te coderen aangezien het een cross-platform bibliotheek is [12]. Verder hebben WxWidgets applicaties een *native look and feel* wat betekent dat WxWidgets de widgets van het besturingssysteem gebruikt waar de applicatie op draait [12], [18]. Ook lijken de WxHaskell en C++ *callbacks* sterk op elkaar waardoor ontwikkeling vlotter verloopt. Ten slotte wordt WxHaskell met relatief weinig regels code geschreven en biedt het goede prestaties, korte compileertijden en weinig geheugenlekken [12], [19], [20].

WxHaskell kent ook nadelen. WxHaskell is, in contrast tot WxWidgets, minder onderhouden waardoor niet alle functionaliteit van WxWidgets *up-to-date* is. Zoals eerder vermeld wordt ook niet de volledige WxWidgets-functionaliteit ondersteund. Hoewel de compileertijd klein is, zorgt WxHaskell soms wel voor grote uitvoerbare bestanden na het compileren [12].

2.5.2 FLTKHS

FLTKHS is gebaseerd op FLTK, een mature cross-platform C++ bibliotheek. FLTKHS is eerder een *low-level* bibliotheek, maar laat toe om *native* GUIs in puur Haskell te maken [20].

FLTKHS kent als eerste voordeel dat er weinig *dependencies* (afhankelijkheden) zijn. Het is ook een van de gemakkelijkste bibliotheken om te installeren op Windows. Bovendien is *Fluid*, een GUI generator die oorspronkelijk bedoeld was om FLTK-code te genereren, sinds kort beschikbaar voor FLTKHS [20]. Net als bij WxHaskell lijken de Haskell en C++ *callbacks* volgens [20] ook sterk op elkaar waardoor ontwikkeling vlotter verloopt.

Echter kent FLTKHS ook nadelen. Zo zijn volgens [20] de compileer- en link-tijden lang en niet optimaal. Hoewel FLTKHS als *low-level* wordt beschreven, is volgens [20] FLTKHS niet altijd even efficiënt. Er is ook geen duidelijke documentatie over FLTKHS beschikbaar, maar het goede nieuws is dat de bestaande documentatie van FLTK gebruikt kan worden om een oplossing in Haskell om te zetten.

2.5.3 FranTK

FranTK wordt volgens [16] beschreven als een *high-level* bibliotheek om op een declaratieve manier GUIs te schrijven.

FranTK kent net als in JavaFX ook het principe van een *Listener* [16], [21]. Listeners zijn in FranTK krachtiger dan in Java omdat in FranTK Listeners geparametriseerd worden over elk type [16]. Er is dus geen dubbele code nodig om verschillende *Mouse*- en *ActionEvent-Listeners* te implementeren zoals in Java. Daarnaast wordt FranTK geïmplementeerd op een toolkit-onafhankelijke manier zodat het gemakkelijk te porten is op verschillende platformen en besturingssystemen [16].

Aangezien FranTK eerder *high-level* is, is er geen mogelijkheid voor *low-level* controle waardoor efficiëntie niet gegarandeerd wordt [16].

2.5.4 GTK2Hs/gtk3

GTK2Hs is een *medium-level* GUI-bibliotheek voor Haskell waarbij de bibliotheek zorgt voor de nodige Haskell *bindings* naar GTK+. GTK+ is een multi-platform *toolkit* om GUIs te ontwikkelen, met een eerder *medium*- tot *low-level* interface [22], [23].

GTK2Hs heeft net als WxHaskell het grote voordeel dat het platform-onafhankelijk ontwikkeld wordt. Echter heeft GTK2Hs op het gebied van actieve ontwikkeling en interfaces een aantal grote voordelen t.o.v. WxHaskell.

Ten eerste wordt nog actiever gewerkt aan GTK2Hs dan WxHaskell. Ten tweede heeft GTK2Hs volgens experimenten van [24] een meer extensieve interface wat voor rijkere interface-elementen zorgt vergeleken met WxHaskell. Bovendien heeft GTK+, en dus ook GTK2Hs een API beschikbaar met een duidelijke documentatie. Hoewel de API voor GTK+ is gemaakt, is het gebruik van GTK2Hs ook mogelijk aangezien GTK2Hs een *mapping* is van GTK+ naar Haskell [25]. Daarnaast biedt GTK2Hs het voordeel dat met Glade, een tool om visueel een GUI te maken waarbij Haskell-code wordt gegenereerd, eenvoudig een GUI kan worden gemaakt [22]. Ten slotte zijn er inmiddels ook Haskell bindings ontwikkeld voor de voorlaatste versie van GTK+ (GTK 3), genaamd gtk3. Deze bindings zijn echter niet compleet [26].

Toch is GTK2Hs niet perfect. Het afhandelen en het werken met threads is vooral beperkt. *Multithreading* is ook niet ondersteund door GTK+, en dus ook niet door GTK2Hs. Haskell *threads* zijn standaard niet parallel-uitvoerbaar naast de GTK+ *threads* [24]. Hoewel er volgens [24] de oplossing bestaat om de *threads* van de GTK+ hoofd-*thread* te pauzeren om de Haskell *threads* een kans te geven, resulteert de oplossing in *threads* die zeer traag worden uitgevoerd. Gelukkig bestaat er een handleiding van [27] die snelle oplossingen biedt voor *multithreading* in GTK2Hs. Ook adviseert het ontwikkelingsteam van GTK2Hs om de gi-gtk bibliotheek te gebruiken aangezien de gi-gtk bibliotheek *out-of-the-box* de volledige *bindings* van GTK 3 ondersteunt.

2.5.5 gi-gtk

gi-gtk is gelijkaardig aan GTK2Hs omdat het voor de *bindings* van Haskell naar GTK+ zorgt. Echter biedt gi-gtk enkele voordelen vergeleken met GTK2Hs. Ten eerste zijn de *bindings* van gi-gtk auto-

gegenereerd in plaats van met de hand geschreven in GTK2Hs. Dit betekent dat de functionaliteit van gi-gtk meer compleet is aangezien er meer *bindings* zijn voorzien vergeleken met GTK2Hs [28], [29]. Ten tweede werkt gi-gtk *out-of-the-box* met GTK+3, vergeleken met GTK2Hs die GTK+2 ondersteunt (tenzij expliciet de gtk3-*bindings* worden gebruikt). Ten slotte biedt gi-gtk ook een declaratieve versie van *bindings* aan (gi-gtk-*declarative*), die declaratief programmeren in GTK+ mogelijk maakt.

Inmiddels is ook de laatste *major release* van GTK+ (GTK 4) in december 2020 uitgekomen [30]. Hierbij zou de gi-gtk bibliotheek theoretisch ondersteund moeten zijn, maar op dit moment gebruiken de meeste handleidingen nog het mature GTK 3 voor de ontwikkeling [26].

2.5.6 Fudgets

Fudgets is een hoog-niveau declaratieve GUI-bibliotheek voor Haskell waarbij Widgets als "fudgets" worden beschreven. Een *fudget* is een "**fu**nctioneel equivalent van een wi**dget**, die in contrast tot andere GUI-bibliotheken niet object-georiënteerd is" [31, p. 46]. Een fudget is tegelijkertijd ook een proces dat kan communiceren met andere fudgets en met de buitenwereld [31].

Volgens [31] is uitbreiden van een programma evident aangezien *fudgets* op een hiërarchische manier gemakkelijk zijn uit te breiden: de bouwblokken zijn *fudgets* en het gehele programma is ook een *fudget*. De abstractie op een hoog-level-niveau van de fudgets als bouwblokken zorgt voor een apart, maar toch logisch concept over een GUI.

2.5.7 Qtah

Qtah is een *medium-level* GUI-bibliotheek voor Haskell die gebaseerd is op Qt 4/5. Op dit moment gebruikt Qtah een groot aantal klassen van QtCore, QtGui, en QtWidgets [32]. Het is een jonge bibliotheek die aan het groeien is. Daarnaast is het ook een bibliotheek dat makkelijk uit te breiden is door het toevoegen van functies of methodes in enkele regels code. Hoewel Qt een *integrated development environment* (IDE) met een ingebouwde GUI-*builder-tool* ter beschikking heeft, is het niet direct evident hoe de *tool* met Qtah werkt [33].

Qtah maakt gebruik van Hoppy om C++ code te *binden*. Hoppy is een *foreign function interface* (FFI) generator voor toegang tot C++ bibliotheken vanuit Haskell [34].

2.6 Eerste technologiescan van de bestaande bibliotheken

Tabel 1 toont een overzicht van de verschillende bestaande Haskell GUI-bibliotheken. De afwegingen van voor- en nadelen volstaat nog niet om een selectie te maken van welke bibliotheek er gebruikt wordt. Wel valt op dat vooral de medium- tot laag-niveau bibliotheken meer *up-to-date* zijn vergeleken met de hoog-niveau bibliotheken. Zo heeft Fudgets zijn laatste *major release* in 2000 gehad waarbij enkel de homepagina wordt geüpdatet voor gebruikersinstructies. FranTK wordt al helemaal niet meer onderhouden. Deze twee bibliotheken vallen dan ook al af.

Bibliotheken gebaseerd op een onderliggende *toolkit* zoals gi-gtk, Qtah en FLTKHS worden nog altijd actief onderhouden. Bovendien hebben deze bibliotheken het voordeel dat er een duidelijke documentatie beschikbaar is en dat er meestal enkele demo's en tutorials erover te vinden zijn. Kortom zijn de volgende bibliotheken op het eerste zicht gepaste kandidaten voor de codegenerator:

- FLTKHS,
- GTK2Hs,
- gi-gtk,
- Qtah.

In het hoofdstuk “3 Bibliotheekkeuze” worden deze kandidaatbibliotheken verder bestudeerd waarna op basis van de eigenschappen van de bibliotheek één enkele bibliotheek over blijft die gebruikt wordt voor de codegenerator.

Tabel 1: Vergelijking van de verschillende Haskell GUI-bibliotheken

	Stijl	**Multi-platform**	**Onder-houden**	**Voordelen**	**Nadelen**
WxHaskell	Medium-low-level	Ja	Nee	+Widgets hebben *native-look* en *feel* +prestatie +korte compileertijd +gebaseerd op WxWidgets	-Grote *binaries* na compileren -Niet 100% functionaliteit WxWidgets -moeilijk te installeren
FLTKHS	Low-level	Ja	Ja	+Haskell en C++ *callbacks* lijken op elkaar +*Custom* widgets mogelijk door C++ methodes te *overriden* +Gemakkelijk te installeren op Windows +Duidelijke documentatie	-veel baseren op FLTK -veel *low-level* implementatie -lange compileertijden -widgets hebben geen *native look*
FranTK	High-level	Alleen Linux en Windows	Nee	+*Listeners* geparametriseerd +*High-level*	-Efficiëntie en prestatie niet gegarandeerd door *high-level*
GTK2Hs/ gtk3	Medium-low-level	Ja	Nee	+Actieve development +Rijkere interface- elementen vergeleken met WxHaskell +grootste deel functionaliteit GTK+ +mogelijkheid voor gebruik GTK 3 met gtk3 +Interessante tutorials te vinden +GUI-*builder tool* ter beschikking: Glade	-Geen officiële support voor *Multithreading* (omwegen bestaan) -Niet alle bindings voor alle GTK+ Widgets -GTK 3 *bindings* niet compleet
gi-gtk	Medium-low-level	Ja	Ja	+meer compleet, meer *bindings* voorzien dan GTK2hs +mogelijkheid om declaratief te werken met gi-gtk-declarative +veel tutorials en voorbeelden te vinden +meer mogelijkheden om met *threads* te werken +GUI-*builder tool* ter beschikking: Glade	/
Fudgets	High-level	Alleen Unix-like systemen. Werkt het beste op GNU/Linux.	Nee	+Duidelijke documentatie +Gemakkelijk hiërarchisch op te bouwen met fudgets	-Weinig voorbeelden en tutorials te vinden
Qtah	Medium-low level	Ja	Ja	+Gebaseerd op Qt +Actieve *development* +Veel uitgebreide functionaliteit Qt mogelijk	-Geen duidelijke en uitgebreide tutorials/demo’s -Relatief jong -Niet de volledige Qt-*bindings*

2.7 Glade vs SceneBuilder

In de vorige hoofdstukken werd al eerder aangehaald dat Glade en SceneBuilder gelijkaardige tools zijn om de layout van GUIs op een eenvoudige manier te ontwikkelen. Aangezien de onderliggende *toolkit* van de Haskell-bibliotheek gi-gtk (GTK+) een GUI-*builder tool* ter beschikking heeft waarbij de layout ook direct in Haskell wordt ingeladen, is het belangrijk deze in detail te bespreken vooraleer eventuele verdere keuzes voor de masterproef te beargumenteren. In deze sectie wordt bestudeerd hoe gelijkaardig deze twee *tools* zijn, en welke voordelen of nadelen Glade te bieden heeft.

Om de verschillen en gelijkenissen weer te geven, wordt in figuur 1 en figuur 2 een voorbeeld-GUI-applicatie, dat ook gebruikt wordt in ”3. Bibliotheekkeuze”, nagemaakt in beide tools.

Het eerste wat opvalt is dat beide *tools* hetzelfde ingedeeld zijn en op eerste zicht ook dezelfde functionaliteit bevatten. Zo hebben beide *tools* voornamelijk drie onderdelen die aangeduid zijn in figuur 1 en figuur 2:

1. Een hiërarchisch overzicht van de GUI met de keuzeopties voor nieuwe GUI-elementen toe te voegen,
2. Een preview van de GUI-applicatie zelf,
3. Een tab waarbij de eigenschappen van een GUI-element in detail kunnen worden gewijzigd.

Beide *tools* hebben, algemeen gesproken, dezelfde workflow waarbij de GUI-elementen geplaatst kunnen worden en de eigenschappen in detail gewijzigd kunnen worden. Nadat een gewenst resultaat is bekomen, worden de layouts van de GUI-applicaties opgeslagen in hun eigen beschreven markup-bestand. Vervolgens wordt, zoals eerder aangehaald, de layout van SceneBuilder in een FXML-bestand opgeslagen. Daarnaast wordt de layout van Glade in een Glade-bestand opgeslagen. Beide bestanden zijn van een XML-achtig formaat die de informatie opslaat over de gebruikte GUI-elementen met de corresponderende eigenschappen.

De XML-gemarkeerde bestanden van beide *toolkits* worden respectievelijk in bijlage A en bijlage B weergegeven. Hierbij valt meteen op dat het aantal regels code voor het Glade-bestand beduidend hoger is vergeleken met het aantal regels code in het FXML-bestand. Dit komt omdat in het Glade-bestand zowel de GUI-elementen alsook de eigenschappen als aparte XML-elementen worden voorgesteld. Bij het FXML-bestand worden enkel de GUI-elementen als aparte XML-elementen voorgesteld waarbij de eigenschappen van het GUI-element als attributen worden gedefinieerd van het XML-element. Toch biedt het Glade-bestand een groot voordeel vergeleken met het FXML-bestand. Het Glade-bestand wordt gebruikt in meerdere programmeertalen zoals C, C++, C#, Vala, Java, Perl, Python en Haskell vergeleken met FXML dat alleen standaard in Java, JRuby en Groovy gebruikt kan worden [1], [35].

Beide *tools* lijken op eerste zicht ook grotendeels dezelfde functionaliteit en workflow te bevatten. Echter is het gebruiksgemak van SceneBuilder verder verfijnd dan met Glade. Zo werkt SceneBuilder met het *drag-and-drop* principe waarbij de GUI-elementen op een intuïtieve wijze gesleept worden naar een gewenste locatie. Dit principe is bij Glade niet aanwezig aangezien de GUI-elementen moeten worden aangeklikt om geplaatst te worden en vervolgens pas de mogelijkheid heeft om aangepast te worden.

Bovendien valt Glade ook kort op het gebruiksgemak doordat het de gebruiker forceert met primitieve widgets te werken. Om een Button te maken met een zelf-gedefinieerde grootte moet bij Glade expliciet de Button in een aparte container geplaatst worden. Dit principe gecombineerd met het feit

dat er niet met *drag-and-drop* gewerkt kan worden, zorgt ervoor dat de workflow in Glade drastisch omslachtiger is vergeleken met SceneBuilder.

Daarnaast heeft Glade ook het nadeel dat het oorspronkelijk alleen op Linux werkt. Er zijn dus omwegen nodig om Glade op Windows te laten werken door bijvoorbeeld MSYS2, een software distributie platform die het mogelijk maakt Linux applicaties native op Windows te uit te voeren, te gebruiken [36], [37].

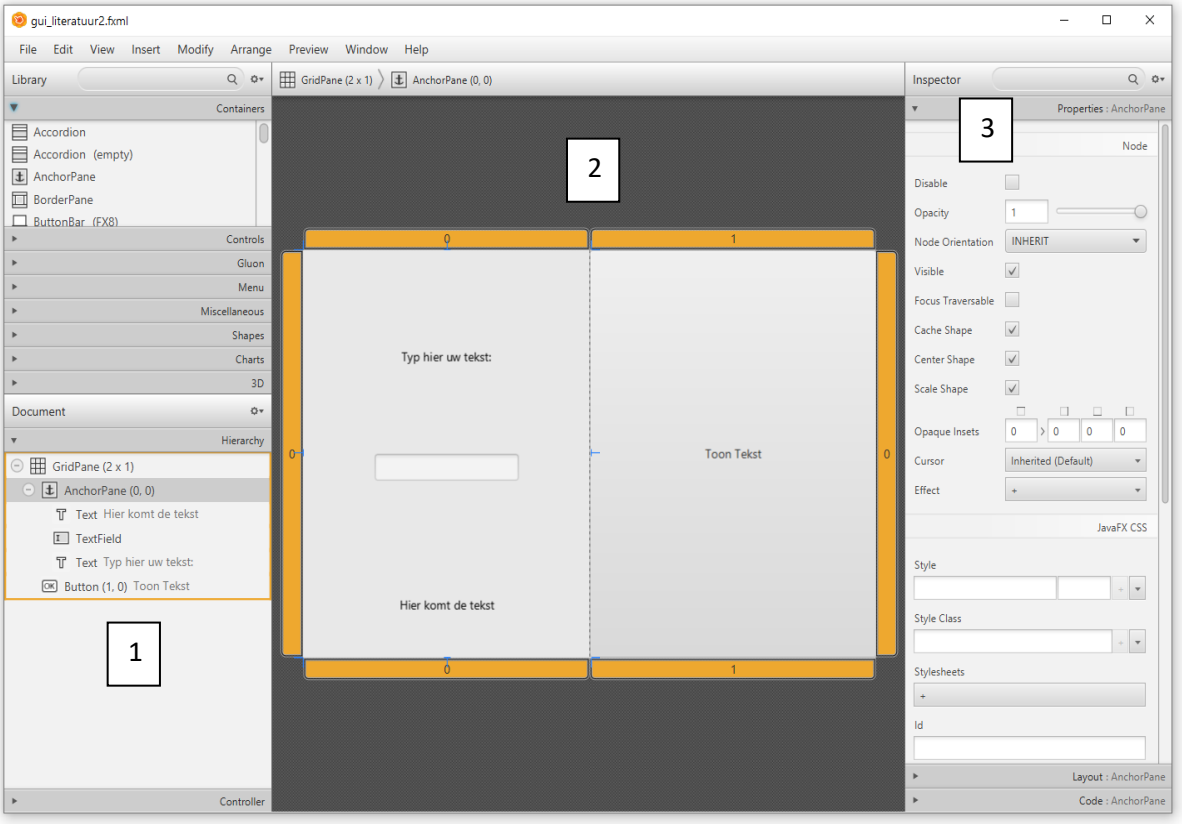
Verder wordt nog actief gewerkt aan beide *tools* om de laatste *features* van de onderliggende *toolkit* (JavaFX en GTK) te ondersteunen. Hoewel de laatste versie van Glade pas in eind november 2020 uitkwam, ondersteunt Glade nog steeds de essentiële features die in GTK+ gebruikt worden [38].

Om af te sluiten toont tabel 2 een samenvatting van de vergelijking tussen SceneBuilder en Glade.

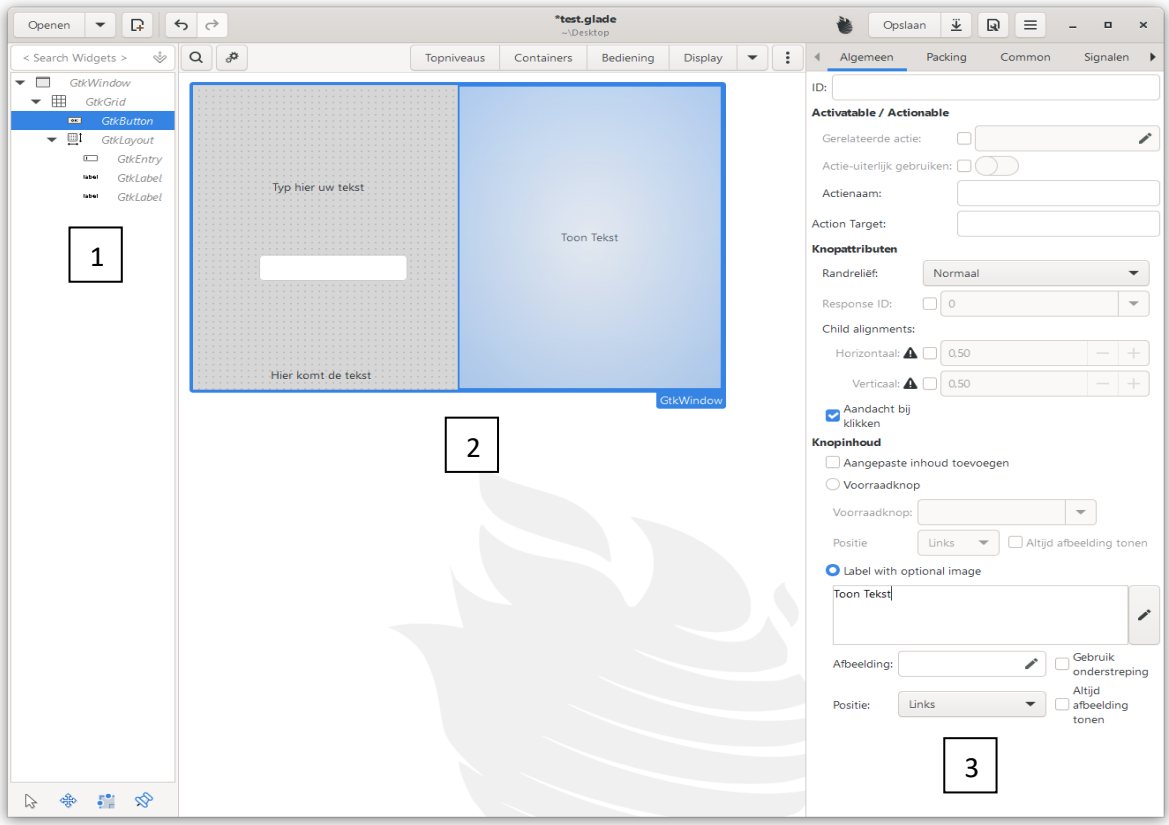
Tabel 2: Vergelijking van verschillende aspecten van SceneBuilder en Glade

	SceneBuilder	Glade
Workflow	GUI grafisch maken met de GUI-elementen van de JavaFX. Vervolgens de layout exporteren die gebruikt kan worden in Java/Groovy/JRuby.	GUI grafisch maken met de GUI-elementen van de GTK. Vervolgens de layout exporteren die gebruikt kan worden in meerdere programmeertalen zoals C, C++, C#, Vala, Java, Perl, Python en Haskell (met gi-gtk/Gtk2Hs).
Markup-bestand	• FXML: GUI-elementen als aparte XML-elementen waarbij eigenschappen als attribuut worden aangepast. • Minder regels code (≈ 37)	• Glade: zowel GUI-elementen als eigenschappen als aparte XML-elementen. • Meer regels code (≈ 81)
Gebruiksgemak	• GUI-elementen slepen met *drag and drop* • Primitieve elementen voor de gebruiker vergemakkelijkt	• GUI-elementen plaatsen door te klikken • Forceren om met primitieve elementen te werken
Multiplatform	Ja	Nee, maar mogelijkheid om op Windows te draaien met MSYS2
Onderhouden	Laatste update in maart 2022	Laatste update in november 2020

Op zich is er dus geen dwingende reden of grote meerwaarde om SceneBuilder en FXML te gebruiken in plaats van Glade behalve dat de opdrachtgever expliciet vroeg om een *tool* voor FXML te bouwen. Verder onderzoek, eventueel in verdere masterproeven, moet uitwijzen of de totaaloplossing met SceneBuilder en FXML superieur is t.o.v. Glade.



Figuur 1: Een voorbeeld GUI-applicatie in SceneBuilder met verschillende onderdelen: hiërarchische weergave van de elementen (1), preview van de GUI-applicatie (2) en deelvenster voor de eigenschappen van een element te wijzigen (3)



Figuur 2: Een voorbeeld GUI-applicatie in Glade met verschillende onderdelen: hiërarchische weergave van de elementen (1), preview van de GUI-applicatie (2) en deelvenster voor de eigenschappen van een element te wijzigen (3)

3 Bibliotheekkeuze

3.1 Vereisten bibliotheek

Om een keuze te maken voor de te-gebruiken bibliotheek voor de codegenerator is het belangrijk om te reflecteren welke eigenschappen de bibliotheek moet bezitten.

Een eerste belangrijke kwestie is dat de bibliotheek *future-proof* moet zijn aangezien het de bedoeling is om in volgende masterproeven verder te werken aan de codegenerator. Hierbij moet de bibliotheek zowel een grote subset aan widgets bevatten als actief onderhouden worden om toekomstige nieuwe *features* van de bovenliggende *toolkit* te ondersteunen. Met *future-proof* wordt bedoeld dat er voortdurend aan de bovenliggende *toolkit* wordt gewerkt en actief onderhouden wordt. De onderliggende Haskell-bibliotheek mag ook niet meer dan één versie achterlopen op de laatste hoofdversie van de bovenliggende *toolkit*. Ook is het een voordeel indien het mogelijk is om zelfgemaakte widgets te maken.

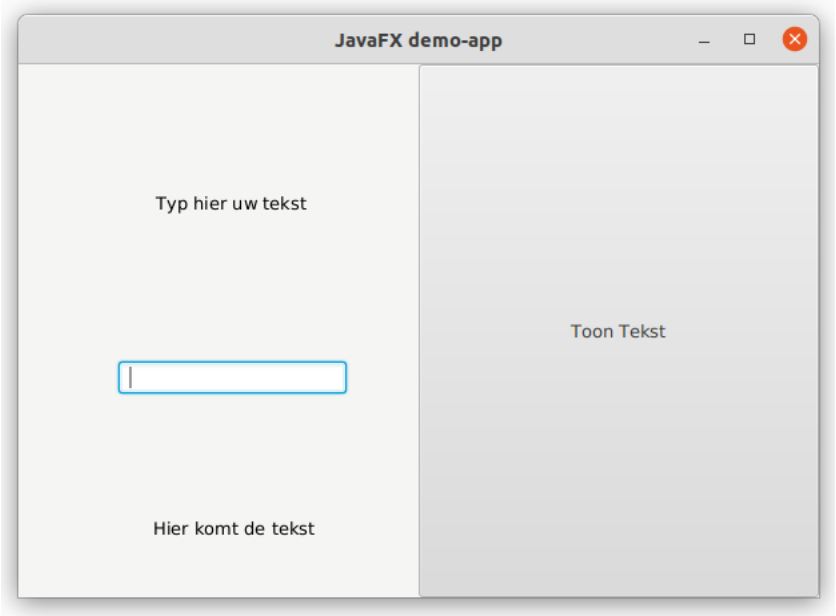
Ten tweede is het gebruiksgemak van de bibliotheek ook een belangrijke factor. Een eerste voorbeeld is als de bibliotheek op een eenvoudige manier geïnstalleerd kan worden en op verschillende platforms kan werken. Dit is belangrijk omdat eindgebruikers uiteindelijk zowel de *tool* als de onderliggende bibliotheek moeten installeren. Intern, voor de ontwikkeling van de codegenerator, is het gebruiksgemak van de bibliotheek een belangrijke factor in de mate dat die de ontwikkelingssnelheid van de codegenerator beïnvloedt. Soortgelijk is het aantal regels code voor een bepaalde implementatie een deelfactor die het ontwikkelen van de codegenerator gemakkelijker maakt.

Om de kandidaat-bibliotheken in detail te bestuderen, wordt er in elke bibliotheek een demo-GUI-applicatie met identiek uitzicht gemaakt. Vervolgens wordt de implementatie per bibliotheek besproken. Ook zal de *build*-tijd (en dus ook compileertijd) van iedere bibliotheek worden getest met het Linux *time* commando om de eventuele snelheidsverschillen in kaart te brengen. De gebruikte computerspecificaties om de bibliotheken te benchmarken zijn terug te vinden in bijlage C. Ten slotte wordt op basis van de voor- en nadelen de uiteindelijke bibliotheek gekozen die gebruikt wordt voor de codegenerator.

3.2 Demo applicatie in JavaFX

3.2.1 Uitzicht en gedrag demo-applicatie

Figuur 3 toont de applicatie die uiteindelijk gemaakt wordt. De bedoeling van de demo-applicatie is dat de gebruiker iets intypt en dat de ingetypte tekst van de gebruiker getoond wordt na het drukken van de knop. Er zijn in deze applicatie zowel statische labels, als een inputveld en labels die als output dienen. Er is ook een (grote) knop die een actie triggert. Dit zijn zowat de minimale onderdelen van een interactieve GUI-applicatie.



Figuur 3: Gewenste demo-applicatie in JavaFX

3.2.2 Implementatie SceneBuilder voor JavaFX

Figuur 4 toont de hiërarchie van de verschillende widgets en grafische elementen die gebruikt worden in het demo-programma. Het gegenereerde FXML-bestand is te vinden in bijlage D. Als top widget is er een *GridPane* van dimensie 2x1. De *GridPane* wordt gezien als een tabel van cellen waarin andere widgets worden toegevoegd. In de rechtercel bevindt zich de Button-widget die gebruikt wordt om de tekst te tonen. In de linkercel bevindt zich een *Anchorpane.* De *Anchorpane* is een grafisch element die toelaat om elementen die zich in de *Anchorpane* bevinden, vrij rond te plaatsen in het gebied van de *Anchorpane* [39]. In de *Anchorpane* bevinden zich het *TextField*-element waarin de gebruiker tekst kan typen, en twee *Labels*.

Figuur 4: Hiërarchie van widgets voor JavaFX in SceneBuilder

3.3 Implementatie van de demo-applicatie in de verschillende kandidaat GUI-bibliotheken

3.3.1 GTK2Hs

Figuur 5 toont de implementatie van de demo-applicatie in GTK2Hs, gebruikmakend van de gtk3 *bindings*. Bij de implementatie van de demo-applicatie in GTK2Hs vallen er enkele zaken op. Ten eerste heeft deze bibliotheek een *medium-level* stijl waarbij bijvoorbeeld GUI-elementen op een declaratieve manier worden aangemaakt. Het verkrijgen van de tekst uit het tekstvak wordt dan weer op een imperatieve, *low-level* manier afgehandeld m.b.v. de *callback*-hulpfunctie *getEntryText.* Ook is het mogelijk om meerdere attributen van een bepaald widget met de *set*-functie in te stellen zoals wordt gedaan voor de attributen van het venster (*window*). Ten slotte heeft een programma in GTK2Hs een vaste structuur:

1. *initGUI* functie aanroepen samen met een toplevel element (*window*) die de GUI initialiseert,
2. Widgets initialiseren en de *callbacks* definiëren,
3. Afsluiten met de *widgetShowAll* functie die voor de benodigde allocatie zorgt samen met de mainGUI loop.

Tabel 3 toont de tegenhanger van iedere JavaFX-widget in GTK2Hs. De GTK2Hs bibliotheek voorziet een directe tegenhanger voor iedere JavaFX-widget die in het voorbeeldprogramma is gedefinieerd.

Tabel 3: Tegenhanger van de JavaFX widgets in GTK2Hs

JavaFX	**GTK2Hs**
GridPane	Grid
Anchorpane	Layout
Label	Label
TextField	Entry
Button	Button

Tabel 4 toont de *build*-tijden voor de *real*, *user* en *sys*-timings voor het *builden* en compileren van het GTK2Hs-programma. Het builden van het GTK2Hs-programma gebeurt direct met de ghc-compiler in de terminal.

Tabel 4: Initiële build-tijden en build-tijden na een kleine aanpassing voor het voorbeeld-GTK2Hs-programma

	Initieel	Na enkele aanpassingen
Real	0,820s	0,817s
User	0,684s	0,649s
Sys	0,136s	0,124s

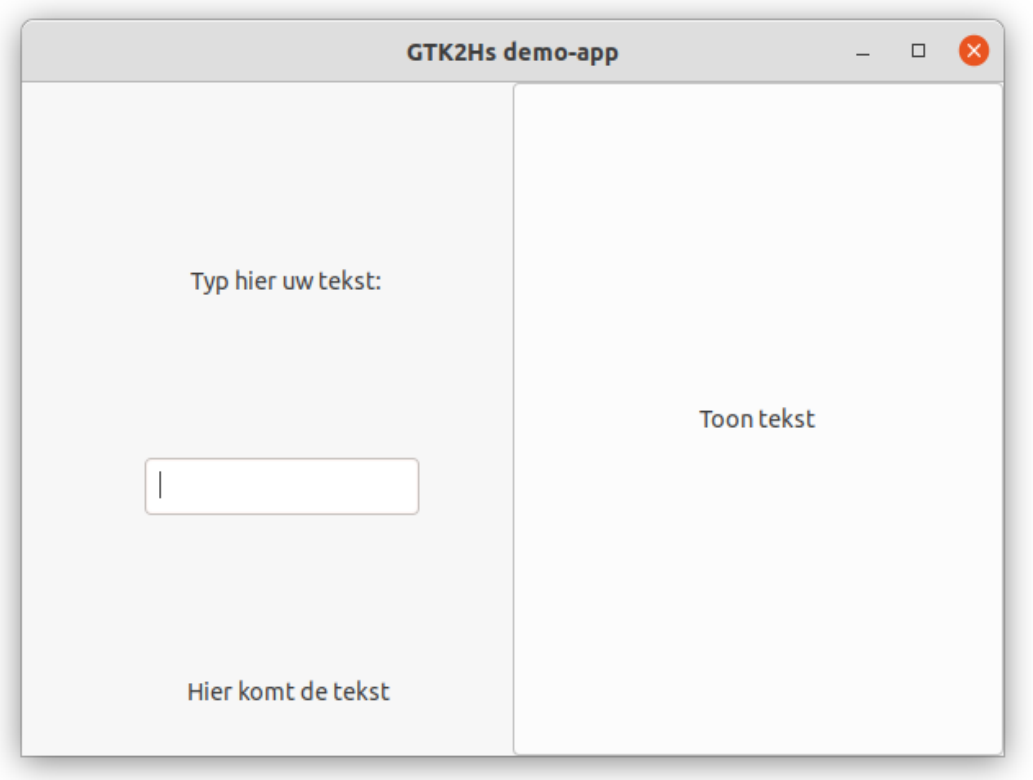
De GTK2Hs bibliotheek is op eerste zicht compleet en gemakkelijk om mee te werken. Zo is om te beginnen iedere JavaFX component dat nodig is voor de voorbeeldapplicatie na te maken in GTK2Hs. Daarnaast worden de *callback*-functies direct in het *main*-block gedefinieerd waarbij het aanroepen naar andere widgets in de *callback*-functie mogelijk is. Zo is het mogelijk om de *callback*-functie van de *Button* direct in de *main*-block te definiëren waarbij op een eenvoudige manier de *textLabel* geüpdatet wordt. Ook zorgt de *set*-functie ervoor dat de attributen van een bepaald widget in één

enkele regel geïmplementeerd wordt. Ten slotte heeft GTK2Hs het voordeel dat alle widgets een *native look* hebben, zoals te zien is in figuur 6.

```
1  import Graphics.UI.Gtk
2
3  --hulp functie voor tekst te updaten in de label
4  updateText label text = set label [labelText  := text]
5
6  --hulp functie voor de tekst van een Entry te verkrijgen
7  getEntryText entry = do
8      s <- entryGetText  entry
9      if s == ""
10     then return ""
11     else return s
12
13 main :: IO ()
14 main = do
15     initGUI
16     window <- windowNew                                                    } 1
17     set window [ windowTitle := "GTK2Hs demo-app", windowResizable := True,
18                  windowDefaultWidth  := 600, windowDefaultHeight := 400 ]
19
20     --initialiseer elementen voor GTKLayout: 1) textLabel, textInput, textLabel
21     labelTextType <- labelNew (Just "Typ hier uw tekst:")
22     labelTextUpdate <- labelNew (Just "Hier komt de tekst")
23     textField <- entryNew
24
25     --Layout = Anchorpane in GTK, initialisere + children toevoegen
26     layoutContainer <- layoutNew Nothing Nothing
27     layoutPut layoutContainer labelTextType 103 109
28     layoutPut layoutContainer textField 75 223
29     layoutPut layoutContainer labelTextUpdate 101 353
30
31     --button initialiseren
32     buttonShowText <- buttonNewWithLabel "Toon tekst"
33     buttonShowText `on` buttonActivated $ do
34         x <- getEntryText textField
35         updateText labelTextUpdate x
36
37     --grid initialiseren en children aan parents toevoegen
38     grid <- gridNew                                  -- grid (2x1) maken
39     gridSetRowHomogeneous grid True                  -- horizontaal even groot
40     gridSetColumnHomogeneous grid True               -- verticaal even groot
41     let attach x y w h item = gridAttach grid item x y w h -- hulpfunctie gridplaatsing
42     attach 0 0 1 1 layoutContainer                   -- x y w h -> layoutContainer links
43     attach 1 0 1 1 buttonShowText                    -- x y w h -> button rechts
44     containerAdd window grid                         -- grid als child aan topwindow
45
46     widgetShowAll window                                                   } 3
47     mainGUI
```

2

Figuur 5: Code van de implementatie van de demo-applicatie in GTK2Hs



Figuur 6: Uiteindelijke demo-applicatie in GTK2Hs

3.3.2 gi-gtk

Figuur 7 toont de implementatie van de demo-applicatie in gi-gtk. Ten eerste valt op dat de hulpfunctie *getEntryText,* die gebruikt wordt in de GTK2Hs implementatie, nu in één directe functie geïmplementeerd wordt. Daarnaast valt op dat alles strikt is en expliciet moet worden aangegeven zoals de widgets, *enums* en functies die uit de juiste deel-*imports* moeten worden opgeroepen. Een ander voorbeeld van uitdrukkelijkheid is de constructor van *Layout* op regel 28 waarbij expliciet moet worden aangegeven dat het *Nothing*-type ook van het *Adjustment*-type is. Dit is nodig omdat het type-inferentie-algoritme anders niet tot een type komt. Ten slotte komt de structuur van het programma grotendeels overeen met dezelfde drie stappen die te zien waren in de implementatie van GTK2Hs:

1. *Gtk.init* functie aanroepen samen met een toplevel element (*window*) die de GUI initialiseert,
2. Widgets initialiseren en de *callbacks* definiëren,
3. Afsluiten met de *widgetShowAll* functie die voor de benodigde allocatie zorgt samen met de *Gtk.main* loop.

Tabel 5 toont de tegenhanger van iedere JavaFX-widget in gi-gtk. Net als bij GTK2Hs voorziet de gi-gtk bibliotheek een directe tegenhanger voor iedere JavaFX-widget die in het voorbeeldprogramma is gedefinieerd.

Tabel 5: Tegenhanger van de JavaFX widgets in gi-gtk

JavaFX	**gi-gtk**
GridPane	Grid
Anchorpane	Layout
Text	Label
TextField	Entry
Button	Button

Tabel 6 toont de *build*-tijden voor de *real*, *user* en *sys*-timings voor het *builden* en compileren van het gi-gtk-programma. Het builden van het gi-gtk-programma gebeurt direct met de ghc-compiler in de terminal.

Tabel 6: Initiële build-tijden en build-tijden na een kleine aanpassing voor het voorbeeld-gi-gtk-programma

	Initieel	Na enkele aanpassingen
Real	2,199s	2,053s
User	1,946s	2,126s
Sys	0,208s	0,198s

De gi-gtk-bibliotheek kent grotendeels dezelfde voordelen als de GTK2Hs bibliotheek zoals de n*ative look* van widgets, vaste structuur in de code en een groot aantal directe tegenhangers van JavaFX widgets. De gi-gtk-bibliotheek heeft het voordeel boven de GTK2Hs bibliotheek, dat er meer directe *bindings* voorzien zijn. Bijna iedere functie en widget in GTK+ is terug te vinden in gi-gtk. Toch kent de bibliotheek het nadeel dat het strikt is en alles expliciet moet worden gedefinieerd uit de juiste sub-modules. Dit resulteert niet per se in meer regels code, maar wel meer code per regel. Bovendien kost het de programmeur meer moeite om alles expliciet te definiëren bij iedere widget en functie. Ten slotte zijn de *build*-tijden enigszins langer dan bij GTK2Hs. Figuur 8 toont de demo-applicatie in gi-gtk.

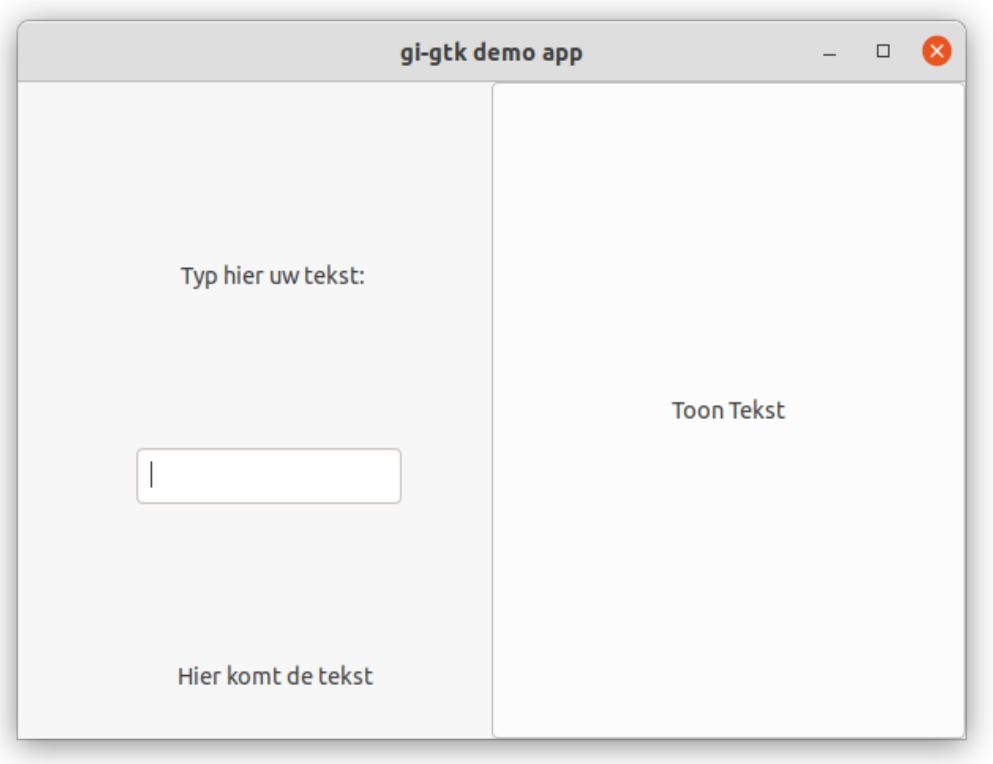
```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ExtendedDefaultRules #-}

module Main (Main.main) where
import qualified GI.Gtk as Gtk
import GI.Gtk.Enums (WindowType(..), Orientation(..))
import GI.Gtk (Adjustment(Adjustment))

main :: IO ()
main = do
  Gtk.init Nothing
  window <- Gtk.windowNew WindowTypeToplevel
  Gtk.set window [Gtk.windowTitle := "gi-gtk demo-app", Gtk.windowResizable := True,
                  Gtk.windowDefaultWidth := 600, Gtk.windowDefaultHeight := 400]

  --initialiseer elementen voor GTKLayout: 1) textLabel, textInput, textLabel
  labelTextType <- Gtk.labelNew (Just "Typ hier uw tekst:")
  labelTextUpdate <- Gtk.labelNew (Just "Hier komt de tekst")
  textField <- Gtk.entryNew

  --Layout = Anchorpane in GTK, initialisere + children toevoegen
  layoutContainer <- Gtk.layoutNew (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)
  Gtk.layoutPut layoutContainer labelTextType 103 109
  Gtk.layoutPut layoutContainer textField 75 223
  Gtk.layoutPut layoutContainer labelTextUpdate 101 353

  --button initialiseren
  buttonShowText <- Gtk.buttonNew
  Gtk.setButtonLabel buttonShowText "Toon Tekst"
  Gtk.onButtonClicked buttonShowText $ do
    x <- Gtk.getEntryText textField
    Gtk.labelSetText labelTextUpdate x

  --grid initialiseren en children aan parents toevoegen
  grid <- Gtk.gridNew                          -- grid (2x1) maken
  Gtk.gridSetColumnHomogeneous grid True       -- horizontaal even groot
  Gtk.gridSetRowHomogeneous grid True          -- verticaal even groot
  Gtk.gridAttach grid layoutContainer 0 0 1 1-- x y w h -> layoutContainer links
  Gtk.gridAttach grid buttonShowText 1 0 1 1 -- x y w h -> button rechts
  Gtk.setContainerChild window grid            -- grid als child aan topwindow

  Gtk.onWidgetDestroy window Gtk.mainQuit
  Gtk.widgetShowAll window
  Gtk.main
```

1

2

3

Figuur 7: Code van de implementatie van de demo-applicatie in gi-gtk



Figuur 8: Uiteindelijke demo-applicatie in gi-gtk

3.3.3 FLTKHS

Figuur 9 toont de implementatie van de demo-applicatie in FLTKHS. Ten eerste is de FLTKHS-bibliotheek meer *low-level* vergeleken met de andere bibliotheken. Zo bestaan de widgets uit meer primitieve types die een vorm toegekend krijgen met de bijhorende parameters zoals lengte, breedte en coördinaten. Hierdoor wordt de hiërarchie ook moeilijk te definiëren wat ertoe heeft geleid dat widgets op een manuele manier werden geplaatst met coördinaten. Bovendien zijn *callback*-functies moeilijk definieerbaar in de hoofd-block (*ui*-block). Daarnaast zijn de *callback*-functies zodanig gemaakt dat ze alleen de widget zelf aanpassen die de *callback*-functie toegewezen heeft gekregen. Daarom is het in de implementatie niet mogelijk om direct vanuit de *callback*-functie van de *Button* de tekst elders aan te passen. Ten slotte heeft een programma bij FLTKHS de volgende structuur:

1) Definities van de *callback*-functies,
2) Ui-block waar alle widgets en windows worden gedefinieerd,
3) Het ui-block samen met de *FL.flush* functie die ervoor zorgt dat het display up-to-date blijft en ervoor zorgt dat alle I/O buffers worden ververst [40],
4) Het ui-block samen met de *FL.ReplRun* functie die voor de uiteindelijke GUI zorgt [41].

Tabel 7 toont de tegenhanger van iedere JavaFX-widget in FLTKHS. Zoals eerder vermeld zijn de widgets primitief in FLTKHS. Er bestaat voor vele JavaFX widgets geen directe tegenhanger in FLTKHS. Zo wordt bijvoorbeeld een Label uit JavaFX geïmplementeerd als een primitieve Box-widget zonder randen in FLTKHS.

Tabel 7: Tegenhanger van de JavaFX widgets in FLTKHS

JavaFX	**FLTKHS**
GridPane	/
Anchorpane	/
Label	Box (zonder tekst)
TextField	Input
Button	Button

Tabel 8 toont de *build*-tijden voor de *real*, *user* en *sys*-timings voor het *builden* en compileren van het FTLKHS-programma. Het builden van het FTLKHS-programma gebeurt via Stack, een cross-platform programma om Haskell projecten te ontwikkelen.

Tabel 8: Initiële build-tijden en build-tijden na een kleine aanpassing voor het voorbeeld-FLTKHS-programma

	Initieel	Na enkele aanpassingen
Real	5,958s	0,336s
User	3,853s	0,293s
Sys	1,300s	0,051s

De FLTKHS-bibliotheek heeft meer nadelen dan voordelen vergeleken met de andere bibliotheken. Ten eerste zijn, door het *low-level* karakter van de bibliotheek, bestaande JavaFX widgets moeilijker te implementeren aangezien deze widgets opnieuw zelf moeten worden gemaakt. Het *low-level* karakter van FLTKHS zorgt er wel voor dat *custom*-widgets maken toegankelijker is dan bij andere bibliotheken. Echter is dit een nadeel aangezien het zelf maken van de widgets meer tijd en regels code vereist. Daarnaast zijn de *callback*-functies moeilijker te implementeren in FLTKHS en is het bewerken van andere widgets in de *callback*-functie zelf niet evident. Ook heeft FLTKHS langere *build*-tijden vergeleken met de GTK2Hs en gi-gtk bibliotheek. Ten slotte heeft FLTKS geen *native look* zoals te zien is in figuur 10.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Main where
import qualified Graphics.UI.FLTK.LowLevel.FL as FL
import Graphics.UI.FLTK.LowLevel.Fl_Types
import Graphics.UI.FLTK.LowLevel.FLTKHS
import Graphics.UI.FLTK.LowLevel.Fl_Enumerations

--functie voor het togglen als op de Button wordt gedrukt
buttonCb :: Ref Button -> IO ()
buttonCb b' = do
  l' <- getLabel b'
  if (l' == "Hello world")
    then setLabel b' "Goodbye world"
    else setLabel b' "Hello world"

ui :: IO ()
ui = do
 window <- windowNew (Size (Width 600) (Height 400)) Nothing (Just "FLTKHS demo app")
 begin window

 --initialiseer de elementen die in de Layout moeten komen: 1) textLabel, textInput, textLabel
 labelTextType <- boxNewWithBoxtype NoBox (toRectangle $ (103, 109, 100, 24)) "Typ hier uw tekst"
 labelTextUpdate <- boxNewWithBoxtype NoBox (toRectangle $ (101, 353, 100, 24)) "Hier komt de tekst"
 textField <- inputNew (toRectangle $ (75, 223, 171, 24)) (Nothing) (Just FlNormalInput)

 --button initialiseren
 b' <- buttonNew (Rectangle (Position (X 300)(Y 0)) (Size(Width 300)(Height 400))) (Just "Toon Tekst")
 setLabelsize b' (FontSize 13)
 setCallback b' buttonCb  --callback van de button initialiseren

 end window
 showWidget window

main :: IO ()
main = ui >> FL.run >> FL.flush

replMain :: IO ()
replMain = ui >> FL.replRun
```

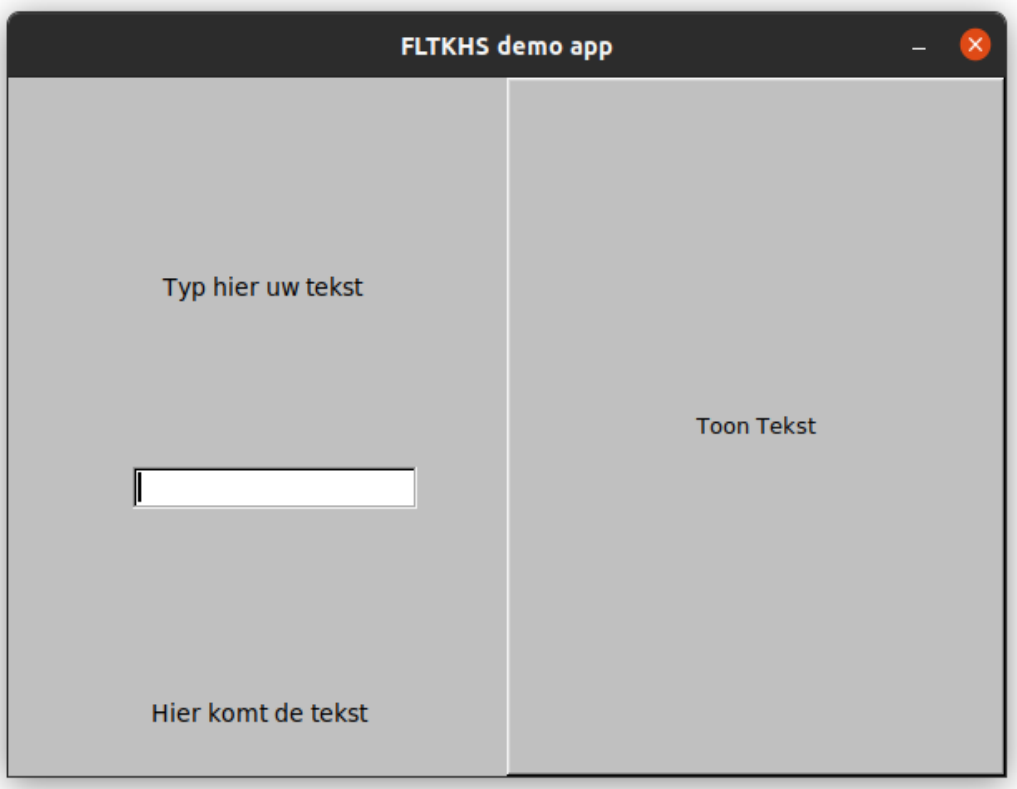
1

2

3

4

Figuur 9: Code van de implementatie van de demo-applicatie in FLTKHS



Figuur 10: Uiteindelijke demo-applicatie in FLTKHS

3.3.4 Qtah

Figuur 11 toont de code van de demo-applicatie in Qtah. Hetgeen direct opvalt, is dat er veel meer regels code zijn dan bij de vorige bibliotheken. Om te beginnen komt dit omdat elke widget die gebruikt wordt, individueel moet worden geïmporteerd (regel 5 tot en met regel 19). Daarnaast is het bij Qtah ook niet mogelijk om een widget al direct alle juiste attributen (zoals de hoogte en breedte bijvoorbeeld) mee te geven. Vaak kan enkel de tekstwaarde worden meegegeven bij het aanmaken van een widget. Regel 40 toont dit principe, waarbij een label wordt aangemaakt met als tekstwaarde "Typ hier uw tekst:". Hierdoor worden dus enkele regels code gebruikt om de attributen van de widgets nog juist in te stellen. Verder heeft Qtah een vaste structuur die er als volgt uitziet:

1) Aanmaken van de applicatie, waarin een nieuw venster (window) wordt gecreëerd,
2) Window aanmaken met alle widgets die erin horen en deze ook al op de juiste plaats neerzetten met de juiste attributen,
3) Widgets koppelen aan bepaalde signalen (Events) met een *callback*-functie.

Tabel 9 toont de tegenhanger van elke JavaFX widget in Qtah. Voor elke widget bestaat er ongeveer een soortgelijke tegenhanger, al zijn de grotere widgets (zoals de Panes) niet exact hetzelfde. Dit brengt uiteraard moeilijkheden met zich mee omdat er geen duidelijke en directe manier is om een tegenhanger van een JavaFX widget in Qtah te implementeren.

Tabel 9: Tegenhanger van de JavaFX widgets in Qtah

JavaFX	**Qtah**
GridPane	≈ QSplitter
Anchorpane	≈ Q(V)BoxLayout
Label	QLabel
TextField	QTextEdit
Button	QPushButton

Tabel 10 toont de *build*-tijden voor de *real*, *user* en *sys*-timings voor het *builden* en compileren van het qtah-programma. Het *builden* van het qtah-programma gebeurt via Cabal, een systeem om Haskell programma's te *builden*.

Tabel 10: Initiële build-tijden en build-tijden na een kleine aanpassing voor het voorbeeld-Qtah-programma

	Initieel	Na enkele aanpassingen
Real	9min 54,892s	3,355s
User	9min 24,867s	2,167s
Sys	0min 32,683s	1,309s

De Qtah-bibliotheek heeft enkele voordelen. Om te beginnen bestaat de bibliotheek uit een groot aantal widgets en GUI-elementen. Dit zorgt voor veel mogelijkheden om widgets en GUI-elementen van JavaFX om te zetten (met behulp van de codegenerator) naar Haskell-code. Een ander voordeel is dat de bibliotheek uitbreidmogelijkheden heeft waardoor er nieuwe widgets kunnen toegevoegd worden indien nodig.

Toch zijn er ook enkele nadelen. Doordat er een groot aantal widgets en GUI-elementen zijn, is het moeizaam om de correcte widget te vinden dat overeenkomt in JavaFX. Ook zijn de meeste widgets niet exact hetzelfde waardoor deze moeilijk zijn om te omvormen, zoals de *GridPane* of *AnchorPane* in het voorbeeldprogramma. Daarnaast zorgt de grote omvang van de bovenliggende *toolkit* ervoor dat er aanzienlijk veel Qt-submodules moeten worden gecompileerd (bij de eerste keer uitvoeren)

waardoor *build*- en compileertijden uiterst langer zijn dan de andere bibliotheken. Eenmaal het programma initieel gecompileerd is, zijn de successieve *build*-tijden gelijkaardig, maar toch nog langer dan de andere bibliotheken. Een ander nadeel van Qtah is dat het onpraktisch is om coördinaten mee te geven aan bepaalde widgets. Dit zorgt ervoor dat er moet gewerkt worden met verschillende vormen van layouts in combinatie met een vaste waarde voor de breedte en hoogte om de verschillende widgets op de juiste plaats te krijgen. Deze manier van werken is uiteraard niet ideaal. Ten slotte toont figuur 12 de demo-applicatie van Qtah.

```
{-# LANGUAGE ScopedTypeVariables #-}

module Main where

import Foreign.Hoppy.Runtime (withScopedPtr)
import qualified Graphics.UI.Qtah.Core.QCoreApplication as QCoreApplication
import qualified Graphics.UI.Qtah.Widgets.QAbstractButton as QAbstractButton
import qualified Graphics.UI.Qtah.Widgets.QApplication as QApplication
import qualified Graphics.UI.Qtah.Widgets.QBoxLayout as QBoxLayout
import qualified Graphics.UI.Qtah.Widgets.QLabel as QLabel
import qualified Graphics.UI.Qtah.Widgets.QPushButton as QPushButton
import qualified Graphics.UI.Qtah.Widgets.QSplitter as QSplitter
import qualified Graphics.UI.Qtah.Widgets.QTextEdit as QTextEdit
import qualified Graphics.UI.Qtah.Widgets.QVBoxLayout as QVBoxLayout
import qualified Graphics.UI.Qtah.Widgets.QWidget as QWidget
import Graphics.UI.Qtah.Signal (connect_)
import System.Environment (getArgs)

-- UI type om later terug te geven (met de juiste attributen)
data UI = UI
  { uiWindow :: QWidget.QWidget
  , uiLabel2 :: QLabel.QLabel
  , uiTextEdit :: IO String }

main :: IO ()
main = withScopedPtr (getArgs >>= QApplication.new) $ \_ -> do
  ui <- newWindow
  QWidget.show $ uiWindow ui
  QCoreApplication.exec

  -- functie om een window aan te maken
  newWindow = do
  window <- QWidget.new
  QWidget.setWindowTitle window "Qtah Demo App"
  QWidget.setFixedSizeRaw window 600 400

  -- linkerkant aanmaken
  label1 <- QLabel.newWithText "Typ hier uw tekst:"   -- eerste label (bovenaan)
  textEdit <- QTextEdit.new                           -- textedit (in het midden)
  QWidget.setFixedSizeRaw textEdit 150 30
  label2 <- QLabel.newWithText "Hier komt de tekst"   -- tweede label (onderaan)
  linkerkant <- QWidget.new
  QWidget.setFixedSizeRaw linkerkant 300 300
  linkerkantLayout <- QVBoxLayout.new        -- een Layout (anchorpane, links)
  QWidget.setLayout linkerkant linkerkantLayout
  -- alle widgets (labels/textedits) meegeven aan de layout van de linkerkant
  QBoxLayout.addWidget linkerkantLayout label1
  QBoxLayout.addWidget linkerkantLayout textEdit
  QBoxLayout.addWidget linkerkantLayout label2
```

1

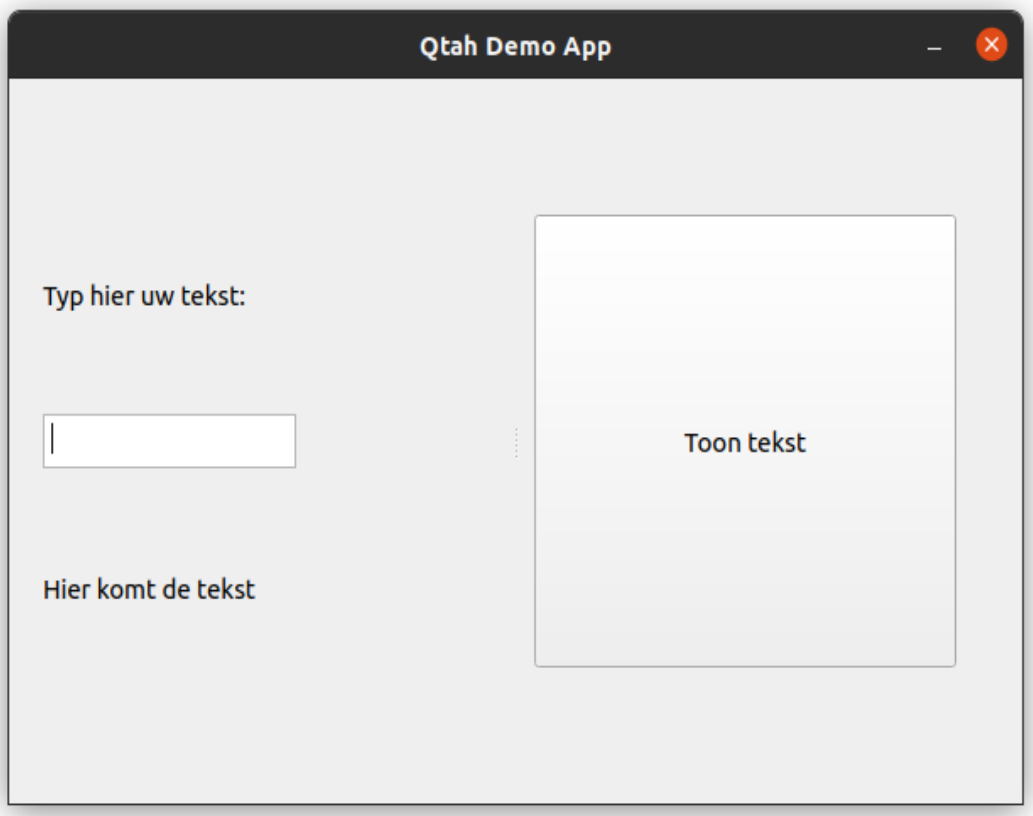
2

```
50
51  -- rechterkant aanmaken
52  button <- QPushButton.newWithText "Toon tekst"   -- button aanmaken
53  QWidget.setFixedSizeRaw button 250 250
54  rechterkant <- QWidget.new
55  QWidget.setFixedSizeRaw rechterkant 300 300
56  rechterkantLayout <- QVBoxLayout.new             -- een Layout (anchorpane, rechts)
57  QWidget.setLayout rechterkant rechterkantLayout
58  QBoxLayout.addWidget rechterkantLayout button    -- button meegeven layout rechts
59
60  -- splitter aanmaken die  widget (window) in twee delen verdeeld
61  splitter <- QSplitter.new
62  QSplitter.addWidget splitter linkerkant
63  QSplitter.addWidget splitter rechterkant
64
65  -- globale layout (anchorpane) met parent de window en waar de splitter wordt meegegeven
66  layout <- QVBoxLayout.newWithParent window
67  QBoxLayout.addWidget layout splitter
68
69  -- ui definiëren met de window, tweede label
70  --(omdat de tekst hiervan moet veranderen) en de tekst van de textedit
71  let ui = UI { uiWindow = window
72              , uiLabel2 = label2
73              , uiTextEdit = QTextEdit.toPlainText textEdit}
74
75  -- een signaal koppelen aan de button om een actie (toontekst) uit te voeren
76  connect_ button QAbstractButton.clickedSignal $ \_ -> toonTekst ui
77
78  return ui -- de ui terug geven
79
80  -- functie om de tekst in label2 te veranderen naar de tekst van de textedit
81  toonTekst :: UI -> IO ()
82  toonTekst ui = do
83  tekst <- uiTextEdit ui
84  QLabel.setText (uiLabel2 ui) tekst
```

2

3

Figuur 11: Code van de implementatie van de demo-applicatie in Qtah



Figuur 12: Uiteindelijke demo-applicatie in Qtah

3.4 Vergelijking geteste bibliotheken

Het is in de meeste bibliotheken gelukt om het voorbeeldprogramma van JavaFX na te maken. Echter waren bepaalde bibliotheken beter dan andere bibliotheken. Zo toont tabel 11 de gedetailleerde voor- en nadelen van de geteste bibliotheken.

Tabel 11: Gedetailleerd overzicht van de voor- en nadelen van de onderzochte GUI-bibliotheken

	GTK2Hs	gi-gtk	FLTKHS	Qtah
Aantal regels code (≈)	33	31	29	58
Build-tijd (real, initieel)	0,820s	2,199s	5,958s	9min 54,892s
Build-tijd (real, na aanpassingen)	0,817s	2,053s	0,336s	3,355s
Voordelen	+ Directe tegenhangers JavaFX widgets beschikbaar +*callback*-functies eenvoudig te integreren +set-functie beschikbaar +custom widgets mogelijk	+ Dezelfde voordelen als GTK2Hs + Grotere subset bindings/widgets dan GTK2Hs	+ Korte compileertijden bij successieve *builds*	+ Grote subset aan widgets beschikbaar
Nadelen	- niet de volledige GTK+ subset/bindings beschikbaar	- Exacte types bij bepaalde widgets en functies expliciet meegeven	- Te *low-level* waardoor meeste widgets handgemaakt moeten worden - *callback* - functies moeilijk te implementeren	- Geen directe aanhanger of meerdere aanhangers mogelijk voor bepaalde JavaFX widget - Niet evident om widgets met coördinaten te plaatsen - Veel imports - Lange initiële *build*-tijden

De FLTKHS-bibliotheek heeft de grootste gebreken uit alle bibliotheken. Ten eerste is deze bibliotheek veel te *low-level* waardoor veel widgets uit JavaFX zelf gemaakt moeten worden. Dit is niet evident aangezien dit veel regels code bevat en moeite vergt. Bovendien waren de *callback*-functies in FLTKHS moeilijk te implementeren en passen ze alleen widgets aan waar de *callback*-functie direct aan gekoppeld is. Daarnaast had de FLTHS wel het voordeel dat de *build*-tijden bij successieve *builds* aanzienlijk korter zijn dan deze bij de andere bibliotheken. Ook valt nog op dat hoewel deze

bibliotheek het minst aantal regels code nodig had, er rekening mee gehouden moet worden dat het niet mogelijk was om de functionaliteit van het voorbeeldprogramma volledig te implementeren.

De Qtah-bibliotheek heeft het grote voordeel dat er een grote subset aan widgets aanwezig is. Echter zijn er voor bepaalde JavaFX-widgets, zoals de AnchorPane, geen directe tegenhangers, wat ook betekent dat werken met coördinaten voor widgets niet evident is. De Qtah-bibliotheek heeft ook de meeste regels code nodig. Indien nodig, is het volgens de documentatie van Qt ook mogelijk om zelfgemaakte widgets te maken [42]. Echter zijn deze zelfgemaakte widgets gemaakt voor de GUI-*builder-tool* "Qt Designer". Zoals eerder vermeld in "2.5.7 Qtah" is het dus onduidelijk hoe de tool gebruikt wordt met Qtah. Ten slotte heeft Qtah buitengewone lange *build*-tijden bij de eerste keer uitvoeren van het programma omdat alle bovenliggende Qt-submodules moeten worden gecompileerd. Eenmaal het programma initieel gecompileerd is, zijn de successieve build-tijden aanzienlijk korter, maar toch enigszins langer dan de andere bibliotheken.

De GTK2Hs en gi-gtk-bibliotheken lijken sterk op elkaar. De initiële en successieve *build*-tijden waren nagenoeg gelijk bij beide bibliotheken, aangezien geen verder optimalisaties mogelijk zijn van externe *build*-omgevingen zoals Stack en Cabal. Bijna iedere *binding* en widget hebben bovendien dezelfde naam in beide bibliotheken en beide bibliotheken hebben veel directe tegenhangers van de JavaFX-widgets. Het werken met de *callback*-functies is in beide bibliotheken ook probleemloos toepasbaar. Beide bibliotheken maken het ook mogelijk om zelfgemaakte widgets te maken via de *GtkDrawingArea*, een blanco widget waarop kan worden getekend [43].

3.5 gi-gtk als bibliotheek voor de codegenerator

De GTK2Hs-bibliotheek heeft het voordeel dat niet alle types expliciet moeten worden meegegeven bij bepaalde widgets en functies. Dit is bij gi-gtk wel het geval. Dit kan leiden tot meer code per regel en een nadelig gebruiksgemak voor de programmeur. Ook heeft de GTK2Hs-bibliotheek een minieme snelheidswinst in *build*-tijden vergeleken met gi-gtk. Het kleine verschil in build-tijden is echter in praktijk zo goed als verwaarloosbaar.

Toch heeft de gi-gtk-bibliotheek het cruciaal voordeel dat er meer *bindings* beschikbaar zijn en dat het met een latere GTK+ versie gebruikt wordt. Het is immers de bedoeling om een codegenerator te ontwikkelen die een basis-subset van JavaFX-widgets ondersteunt en later uitgebreid wordt naar een grotere subset. Daarom wordt voor de codegenerator de gi-gtk bibliotheek gekozen omwille van de toekomstbestendigheid.

De volgende stap bestaat erin om de codegenerator zelf te ontwikkelen waarbij gi-gtk als bibliotheek gebruikt wordt voor de Haskell-code die wordt gegenereerd. Hierbij is het belangrijk dat ten eerste een keuze wordt gemaakt voor welke GUI-elementen zullen worden overgenomen uit JavaFX en GTK+ (gi-gtk). Pas daarna wordt de exacte methode uitgelegd hoe de codegenerator wordt opgebouwd.

4 Codegenerator

4.1 Overeenkomende GUI-elementen JavaFX en GTK+ (gi-gtk)

Alvorens te starten met het bouwen van de codegenerator, moet onderzocht worden in welke mate de GUI-elementen van JavaFX en GTK met elkaar overeenstemmen. Om tot deze vergelijkende tabel te komen zijn de JavaFX documentatie, GTK-documentatie, SceneBuilder en Glade langs elkaar gezet om eenvoudig de overeenkomende elementen op te merken, zoals aangetoond in bijlage E. Het resultaat, de lijst met de overeenkomende GUI-elementen van JavaFX en GTK, wordt getoond in tabel 12.

Tabel 12: Overeenkomende containers (A) en controls/widgets (B) van JavaFX en GTK

A

JavaFX	GTK
HBox	hbox
VBox	vbox
AnchorPane	Layout
GridPane	Grid
ScrollPane	ScrolledWindow
SplitPane	Paned
TabPane	Notebook
StackPane	Stack
Accordion	~TreeView
Toolbar	Actionbar
DialogPane	Dialog

B

JavaFX	GTK
Button	Button
Label	Label
TextField	Entry
CheckBox	CheckButton
ComboBox	ComboBoxText
RadioButton	RadioButton
Hyperlink	LinkButton
ImageView	Image
ListView	ListView
ColorPicker	ColorButton
MediaView	MediaFile/Video
MenuButton	MenuButton
PasswordField	PasswordEntry
ProgressBar	ProgressBar
Scrollbar	Scrollbar
Separator	Separator
Slider	Scale
Spinner	SpinButton
ToggleButton	ToggleButton
Treeview	Treeview

Het valt op dat tabel 12 opgedeeld is in twee deeltabellen. Dit is omdat er een onderscheid mogelijk is tussen de GUI-elementen. Deeltabel A geeft de containers weer. Dit zijn de GUI-elementen die andere GUI-elementen bevatten en grotendeels verantwoordelijk zijn voor de relaties tussen de GUI-elementen. Containers kunnen bijvoorbeeld meerdere widgets en deelcontainers bevatten. Deeltabel B geeft de *controls*, oftewel de widgets zelf weer. Dit zijn de GUI-elementen die visueel direct zichtbaar zijn en waar, eventueel, interactie met de gebruiker mee gedaan kan worden.

4.2 Keuze subset GUI-elementen en doel voor codegenerator

In de inleiding is reeds aangegeven dat het niet de bedoeling is om in deze masterproef de volledige subset van GUI-elementen om te zetten. Daarom is het belangrijk om een stevige subset/basis van GUI-elementen te hebben waar verder op kan worden gebouwd.

Het grootste doel voor de codegenerator is dan om het voorbeeldprogramma bij het hoofdstuk “3 Bibliotheekkeuze” na te bouwen. Dit voorbeeldprogramma bevat namelijk een variëteit aan GUI-elementen zoals de meest voorkomende widgets en bovendien ook een hiërarchische structuur.

Een eerste deel-doelstelling is om volgende GUI-elementen, die in het voorbeeldprogramma gebruikt worden, na te bouwen:

- GridPane (container),
- AnchorPane (container),
- Button (control),
- Label (control),
- TextField (control).

Vervolgens is de tweede deel-doelstelling om de hiërarchie tussen de GUI-elementen te realiseren. De implementatie van hiërarchie is immers een belangrijk aspect die vormgeeft aan een toekomstbestendige basis voor de codegenerator. Bovendien is hiërarchie ook noodzakelijk om het volledige voorbeeldprogramma van hoofdstuk “3 Bibliotheekkeuze” na te bouwen.

Indien beide deel-doelstellingen zijn behaald, is het de bedoeling om de JavaFX-subset uit te breiden. De uitbreiding zal bestaan uit enkele containers en controls waaronder:

- HBox (container),
- VBox (container),
- TabPane (container),
- CheckButton (control),
- ComboBox (control),
- LinkButton (control),
- RadioButton (control).

4.3 Java als programmeertaal codegenerator

Zoals eerder aangegeven in hoofdstuk “2.1 JavaFX en SceneBuilder”, is het FXML-bestand een XML-gemarkeerd tekstbestand die de *nodes*/widgets beschrijft met bijhorende eigenschappen. Om de codegenerator te ontwikkelen is het dan ook belangrijk dat het FXML-bestandstype eerst moet worden *geparsed* om de eigenschappen van iedere *node*/widget te extraheren. Dit kan op verschillende manieren gedaan worden door bijvoorbeeld *custom* XML-*parsers* in een programmeertaal naar keuze te bouwen of bestaande XML-*parsers* te gebruiken zoals Xerces Java die een hoge performantie garandeert [44].

Toch bestaat er een betere oplossing dan het manueel parsen van een XML-document, namelijk om Java zelf te gebruiken voor het FXML-bestand te parsen.

Het voordeel dat Java met zich meebrengt is dat het parsen van FXML al intern wordt gedaan waarbij relaties tussen de elementen eenvoudig opgevraagd kunnen worden. Dit voordeel kan aangetoond worden met een voorbeeld in figuur 13 waarbij de FXML wordt weergegeven van een GridPane met enkele *child-Nodes*. Op eerste zicht is deze XML-structuur niet eenvoudig te volgen. Alvorens de *child-Nodes* worden bereikt in de XML-boom, zijn er enkele *Column*- en *Rowconstraints* te vinden die informatie geven over de grootte van de cellen. Pas daarna komen de *child-Nodes* zoals de AnchorPane en Button aan bod. Hierdoor is het bij het manueel parsen van FXML onduidelijk waartoe iedere *Column*- en *Rowconstraint* behoort. Het wordt dan ook snel ingewikkeld en moeizaam om de

relaties tussen de Nodes uit te zoeken en de eigenschappen van de cellen te koppelen aan de *child-Nodes* indien FXML manueel wordt geparsed.

Het voordeel dat FXML al in Java wordt geparsed is dus dat de eigenschappen en relaties onderling tussen de *Nodes* eenvoudiger kunnen worden geëxtraheerd. Bij een GridPane bijvoorbeeld, kan eenvoudig in Java de methodes worden opgeroepen om de cel eigenschappen rond een bepaalde *child-Node* op te vragen. Enkele voorbeelden hiervan worden getoond in figuur 14. Een bijkomend voordeel van Java is ook dat de eigenschappen van de widgets en *Nodes* zelf op een eenvoudige manier worden geëxtraheerd door de *getters* en *setters* van de *Node* direct aan te roepen. Door de reeds opgenoemde voordelen zal Java dus gekozen worden voor verdere ontwikkeling van de codegenerator.

Toch zijn er ook enkele voordelen wanneer de codegenerator in een andere taal wordt gemaakt, zoals Haskell zelf. Ten eerste zou het principe van een *self-healing-systeem* geïntroduceerd worden waarbij de gegenereerde Haskell-code geen syntaxfouten meer bevat. Dit is omdat de syntax van Haskell niet controleerbaar is in Java, maar wel in Haskell zelf. Bovendien is het herkennen van bepaalde FXML-elementen in Haskell wel evidenter mogelijk door bijvoorbeeld *pattern matching* toe te passen.

```
<GridPane maxHeight="1080.0" maxWidth="1920.0"
                    prefHeight="400.0" prefWidth="600.0" hgap="10">
   <columnConstraints>
     <ColumnConstraints hgrow="SOMETIMES" maxWidth="540.0"
          minWidth="10.0" percentWidth="50.0" prefWidth="302.0" />
     <ColumnConstraints hgrow="SOMETIMES" maxWidth="331.0"
          minWidth="10.0" percentWidth="50.0" prefWidth="298.0" />
     </columnConstraints>
     <rowConstraints>
       <RowConstraints minHeight="10.0" vgrow="SOMETIMES" />
     </rowConstraints>
       <children>
          <AnchorPane prefHeight="400.0" prefWidth="600.0">
             <children>
                <TextField layoutX="75.0" layoutY="223.0"
                                alignment="TOP_CENTER" promptText="Vul hier iets in"  />
                <Label layoutX="90.0" layoutY="94.0" text="Typ hier uw tekst" />
                <Label layoutX="88.0" layoutY="341.0" text="Hier komt de tekst " />
              </children>
           </AnchorPane>
                <Button maxHeight="1.79" maxWidth="1.79" mnemonicParsing="false"
                                prefHeight="477.0" prefWidth="343.0" text="Toon tekst"
                                GridPane.columnIndex="1" />
       </children>
 </GridPane>
```

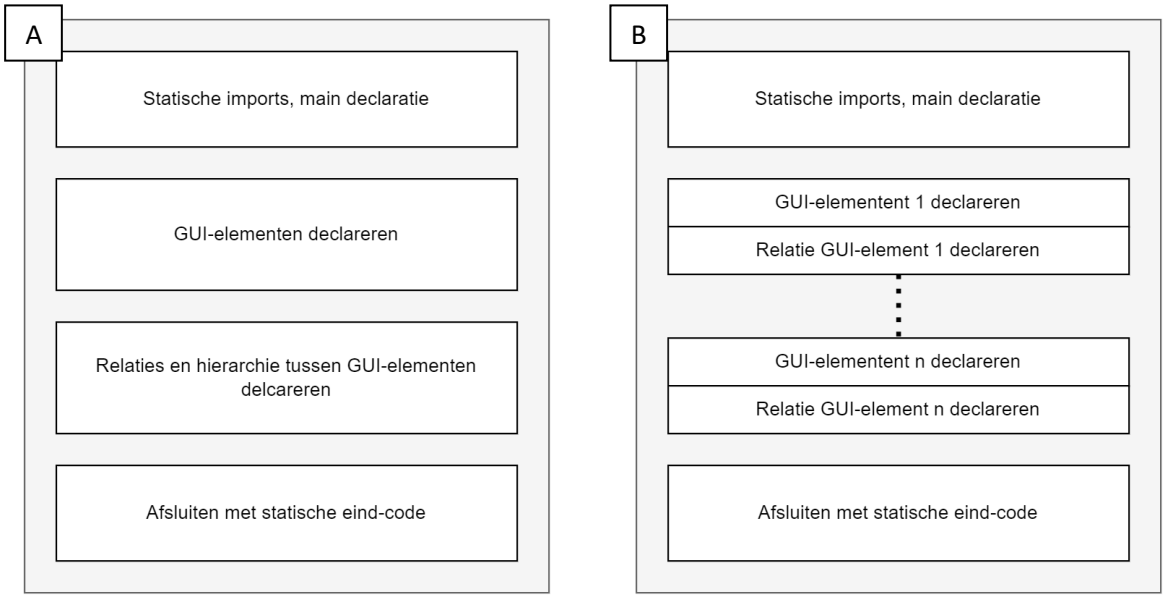
Figuur 13: FXML-code-snippet van een GridPane met enkele child-nodes

```
GridPane.getColumnIndex(childNode); //colomn of child in GridPane
GridPane.getRowIndex(childNode);    //row of child in GridPane
((GridPane) node).getHgap();        //spacing of specefic node in Tree
```

Figuur 14: Eenvoudige methodes in Java om de ingewikkelde relatie-eigenschappen in een GridPane te extraheren

4.4 Overzicht structuur van de te-genereren code

Het is belangrijk om de structuur van een typisch gi-gtk-programma te overlopen vooraleer de codegenerator gebouwd wordt. Zoals eerder aangegeven in het hoofdstuk “3.3.2 gi-gtk” bestaat het programma uit voornamelijk drie delen waaronder: statische begin-code, declaratie van widgets en relaties en statische eind-code. Hierbij is er een variatie bij het gedeelte waar de widgets en relaties worden gedeclareerd, zoals aangetoond in figuur 15. Een uitgewerkt voorbeeld van echte code wordt in bijlage F gevonden.

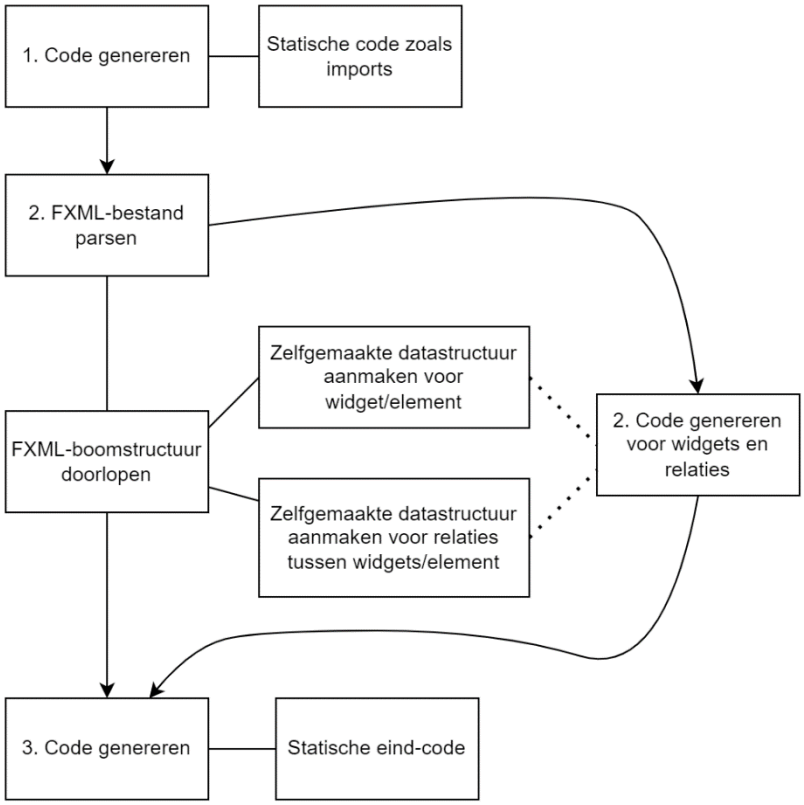


Figuur 15: Deelvariaties A en B van de codestructuur gi-gtk

Figuur 15 toont de beide deelvariaties van de codestructuur. Ten eerste is er deelvariatie A waarbij het declareren van de GUI-elementen en het declareren van de relaties tussen de GUI-elementen gescheiden zijn. Hierbij worden dus eerst de widgets en containers gedefinieerd om vervolgens de relaties tussen een bepaald widget en container te definiëren. Vervolgens is er deelvariatie B waarbij een GUI-element zoals een button gedeclareerd kan worden, en vervolgens direct erna ook de relatie tussen de button en de bovenliggende container gedeclareerd kan worden. Deze twee mogelijkheden zijn gelijkaardig aan de *Salami* versus *Russian Puppet-style* bij de declaratie van een *XML schema definition* (XSD), en bieden beide een aantal voor- en nadelen. De implementatie van de codegenerator gebruikt deelvariatie A. De voor- en nadelen van beide deelvariaties worden besproken in het hoofdstuk “5 resultaten en discussie”, waar ook wordt toegelicht waarom de keuze is gemaakt voor deelvariatie A voor de codegenerator.

4.5 Overzicht te nemen stappen

De codegenerator bestaat uit drie hoofdstappen. Deze drie delen helpen ook het stappenplan voor de codegenerator vormgeven. Het stappenplan is grafisch weergegeven in figuur 16.



Figuur 16: Schematische weergave van het stappenplan voor een werkende codegenerator

Een eerste stap is om statische code te genereren voordat het FXML-bestand wordt geparsed. Deze code bevat zowel de vaste GTK-imports alsook de begindeclaratie van de *main*-functie en de *Gtk.init*-functie die altijd moet worden gedeclareerd in het begin van het programma.

De tweede stap is om het FXML-bestand te *parsen*. Zo moet tijdens of na het *parsen custom* datastructuren worden aangemaakt van de widgets, van de grafische elementen en van de relaties uit het FXML-bestand. Dit is omdat de eigenschappen van de JavaFX-elementen niet blindelings over te nemen zijn aangezien de GTK-varianten niet exact dezelfde eigenschappen bevatten. Ook kunnen met *custom* datastructuren bepaalde uitzonderingen van widgets eenvoudig aangepast worden na het parsen. Verdere reflectie over de zelfgemaakte datastructuren zullen in “5.6 Zelfgemaakte klassen voor de GTK-Widgets ” besproken worden.

Tijdens de *parse*-stap kan tegelijkertijd al Haskell-code gegenereerd worden o.b.v. de *custom* datastructuren en relaties die zijn aangemaakt. De Haskell-code die wordt gegenereerd wordt aangepakt met templates in combinatie met de Java StringBuilder en een *tool* van [45] om *placeholders* in een String-template te vervangen. De code voor deze tool is te vinden in bijlage G.

De laatste stap van de codegenerator bestaat uit Haskell-code genereren voor de statische eind-code. Deze code bevat onder andere het declareren van de *Gtk.widgetShowAll*-functie en de *Gtk.main*-functie die de *main*-loop afsluiten.

De reden waarom er ook datastructuren voor relaties aangemaakt moet worden zal in de discussie in het hoofdstuk “5.4 Waarom relaties en widgets gescheiden houden” verduidelijkt worden.

4.6 FXML parsen

De bedoeling van het FXML-bestand te *parsen*, is om de FXML-boomstructuur te overlopen en daaruit datastructuren aan te maken voor de GUI-elementen/relaties. Er is hierbij vertrokken van een functie uit [46] die alle Nodes van een FXML-bestand recursief doorloopt.

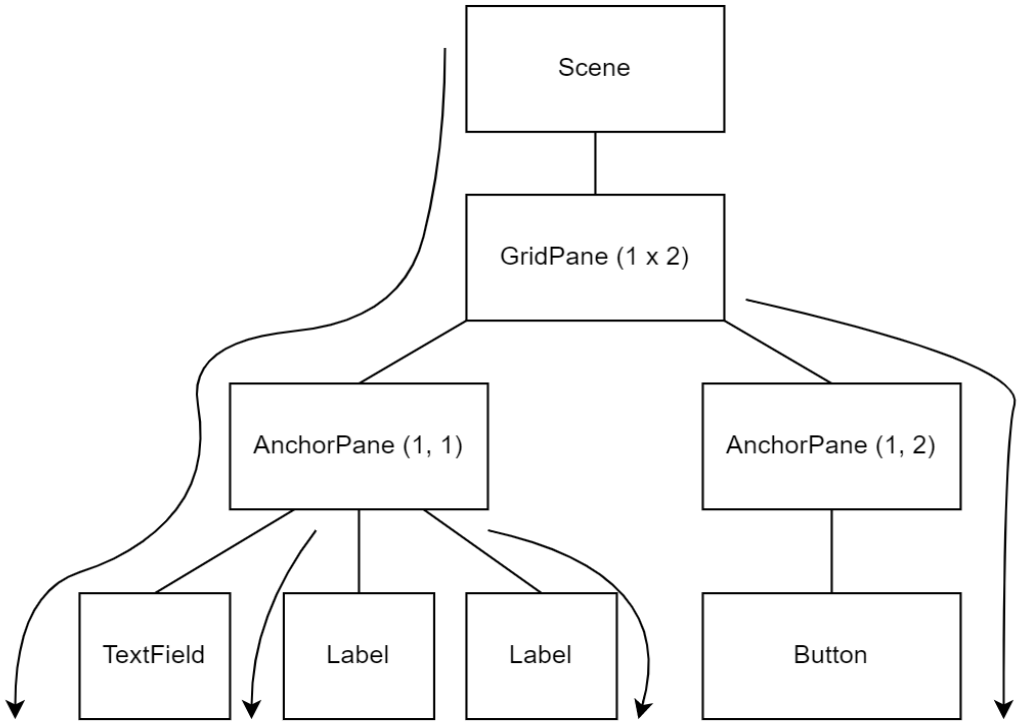
Om de FXML-boomstructuur te overlopen is het mogelijk om recursie te gebruiken zoals getoond in de *snippet* van figuur 17. In deze code-*snippet* wordt iedere JavaFX-node oftewel GUI-element op een hiërarchische manier uitgeprint in de console. Het resultaat in de console van het voorbeeldprogramma dat gebruikt is bij hoofdstuk "3 Bibliotheekkeuze" is te zien in figuur 18 en een visuele weergave van het overlopen van de boomstructuur is te zien in figuur 19.

```
1 public static void dump(Node n) {
2     dump(n, 0);
3 }
4
5 private static void dump(Node n, int depth) {
6     for (int i = 0; i < depth; i++) System.out.print("   ");
7     System.out.println(n);
8     if (n instanceof Parent) {
9         for (Node c : ((Parent) n).getChildrenUnmodifiable()) {
10            System.out.println(n); //print node
11            dump(c, depth + 1);    //ga verder in de tree
12        }
13    }
14 }
```

Figuur 17: Code snippet om de FXML-boomstructuur recursief te doorlopen [45]

```
1 Grid hgap=0.0, vgap=0.0, alignment=TOP_LEFT
2   AnchorPane@586533c4
3     TextField@1dc29df[styleClass=text-input text-field]
4       Pane@242a3d7
5         Path[elements=[], fill=0xcccccccff, fillRule=NON_ZERO]
6         Text[text="", x=0.0, y=13.22, font=Font[name= …, size=12.21]]
7         Group@5282ab39
8           Path[elements=[MoveTo[x=0.0, y=0.22],…, stroke=0x333333ff]
9     Label@730d3e5f[styleClass=label]'Typ hier uw tekst'
10      Text[text="Typ hier uw tekst", x=0.0, y=0.0, font=Font[name= …, size=12.21]]
11    Label@4b309c26[styleClass=label]'Hier komt de tekst '
12      Text[text="Hier komt de tekst", x=0.0, y=0.0, font=Font[name= …, size=12.21]]
13  Button@13c8efb1[styleClass=button]'Toon Tekst'
14    Text[text="Toon de tekst", x=0.0, y=0.0, font=Font[name= …, size=12.21]]
```

Figuur 18: Console output van de dump-functie die de nodes op een hiërarchische wijze uitprint



Figuur 19: Visuele weergave van de FXML-boomstructuur uit het demo-programma

In figuur 20 zijn de JavaFX-*Nodes* met de belangrijkste attributen op een hiërarchische wijze te zien. Hierbij vallen enkele zaken op. Ten eerste lijkt het alsof er telkens een aparte Text-*Node* onder iedere Button en Label zichtbaar is. Deze Text-*Node* is feitelijk het tekst-attribuut van de Button of Label zelf, maar kan zelf als zelfstandige *Node* apart ook bestaan. De zelfstandige Text-*Node* komt echter bijna nooit voor, en zal meestal als attribuut toegepast zijn voor een andere *Node*. Ten tweede heeft ieder JavaFX element een unieke hashcode aangegeven na de "@". Deze hashcode maakt het mogelijk om in de boomstructuur bepaalde widgets terug te vinden. Het gebruik ervan wordt later in de *parsing*-stap verduidelijkt. Ten slotte is te zien dat de JavaFX-*Nodes* enkele eigenschappen bevatten, zoals de x- en y-coördinaten, die in SceneBuilder werden gedefinieerd.

```
8    ...
9    Label@730d3e5f[styleClass=label]'Typ hier uw tekst'
10     Text[text="Typ hier uw tekst", x=0.0, y=0.0, font=Font[name= …, size=12.21]]
11   Label@4b309c26[styleClass=label]'Hier komt de tekst '
12     Text[text="Hier komt de tekst", x=0.0, y=0.0, font=Font[name= …, size=12.21]]
13   Button@13c8efb1[styleClass=button]'Toon Tekst'
14     Text[text="Toon de tekst", x=0.0, y=0.0, font=Font[name= …, size=12.21]]
```

Figuur 20: Visuele weergave van hoe de widgets beschouwd zullen worden met de primitieve Text-node

Alvorens de volgende *parsing*-stappen worden uitgelegd, is het belangrijk te vermelden dat de Text-*Node* niet als aparte widget/element zal worden beschouwd voor de codegenerator. Hoewel in JavaFX de Text-*Node* als widget op zich gebruikt wordt om tekst voor te stellen, wordt alleen de Label-*Node* als *Node* beschouwd om tekst voor te stellen. Dit wordt gedaan omdat er maar één directe tegenhanger is in GTK voor tekst voor te stellen: de *Gtk.Label*.

4.7 Widgets parsen

Het parsen van de JavaFX GUI-elementen wordt in de dump-functie gedaan die eerder als basis werd uitgelegd in figuur 17. Figuur 21 geeft een meer gedetailleerde versie van de dump-functie waar wordt aangegeven hoe de custom-datastructuren van de widgets/controls worden aangemaakt.

```
1  private static void dump(Node n, int depth) {
2          for (int i = 0; i < depth; i++) System.out.print("   ");
3
4          //PARSE AND CREATE WIDGETS-CONTROLS
5          if(n instanceof javafx.scene.control.Button){
6              //FETCH eigenschappen widget
7              var width = n.getLayoutBounds().getWidth();
8              var height = n.getLayoutBounds().getHeight();
9              ...
10             //Widget aanmaken
11             var button = new Button(...);
12
13             //Widget bijhouden
14             guiWidgets.add(button);
15
16             //Haskell declaratie naar bestand schrijven
17             appendTextToFile("--Button \n  " + button.gtkHsCode());
18         }else if (n instanceof AnchorPane){
19             ...
20         }else if (n instanceof javafx.scene.control.Label){
21             ...
22         }else if (n instanceof TextField){
23             ...
24         }else if (n instanceof GridPane){
25             ...
26         }
27
28         ...
29
30     }
```

Figuur 21: Code-snippet voor datastructuren aan te maken voor de widgets

Ten eerste worden enkele voordelen van Java als taal in de codegenerator duidelijk. De eigenschappen van de JavaFX-*Node* worden eenvoudig opgevraagd door de ingebouwde *getters* en *setters* van de JavaFX-*Node* aan te roepen. Ook kan bijvoorbeeld voor het geval van de GridPane, de celeigenschappen eenvoudig worden opgevraagd. Toch is er ook een nadeel bij deze manier waarbij er Java wordt gebruikt. Hoewel de huidige specifieke JavaFX-*Node* op een eenvoudige manier geïdentificeerd wordt met *instanceof,* is het niet evident om telkens met *if-else-statements* te werken. Een evidentere manier om hierbij de *Nodes* te herkennen is bijvoorbeeld, zoals eerder al aangehaald, mogelijk met *pattern matching* in Haskell.

Indien de JavaFX-*Node* geïdentificeerd is, wordt vervolgens de zelfgemaakte datastructuur aangemaakt. Hierbij worden de nodige eigenschappen van de JavaFX-*Node* verkregen, de *custom*-widget klasse zelf aangemaakt en de Haskell-code van de widget-declaratie naar het bestand geschreven.

4.8 GTKWidget datastructuur

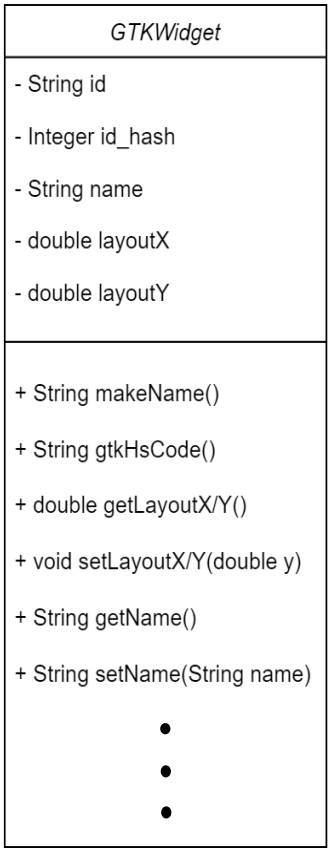
JavaFX-widgets en GTK-widgets bevatten tal van verschillende eigenschappen. Zo kan bijvoorbeeld een Button een ingebouwde tekstlabel-eigenschap bevatten en een Entry of TextField een *placeholder*-eigenschap. Toch bestaan er ook eigenschappen die voor iedere widget, zowel in JavaFX als in GTK nodig zijn. Om een overzichtelijke structuur van zelfgemaakte GUI-elementen te realiseren en dubbelzinnigheid te vermijden is het belangrijk om een basiselement oftewel hoofdklasse te voorzien met de gedeelde eigenschappen, waarvan de andere elementen erven

Figuur 22 geeft de basis klasse voor een GUI-element aan, genaamd *GTKWidget*. De belangrijkste eigenschappen hierbij zijn:

- id: dit is een unieke id-String die door de gebruiker kan worden gedefinieerd in SceneBuilder,
- id_hash: een unieke hashcode die een JavaFX-Node uniek identificeert,
- name: de naam die gebruikt zal worden voor het declareren van de widget,
- layoutX: x-coördinaat van de widget,
- layoutY: y-coördinaat van de widget.

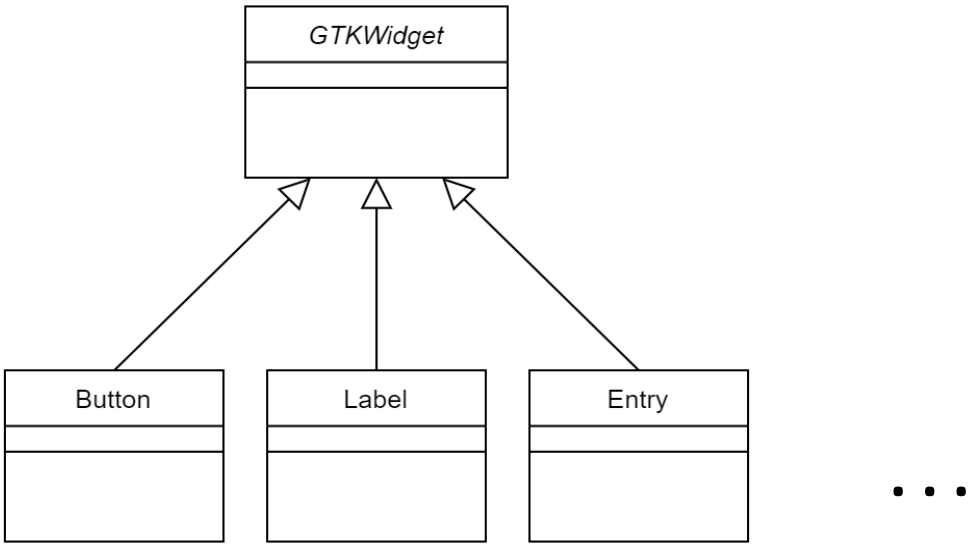
Naast de *getter* en *setter*-methodes bevat de GTKWidget ook enkele belangrijke methodes:

- makeName(): functie die het *name*-attribuut genereert op basis van de id en id_hash eigenschap,
- gtkHsCode(): functie die de Haskell-code voor de widget-declaratie genereert.



Figuur 22: Zelfgemaakte datastructuur voor de klasse van een Widget

De *gtkHsCode()*-functie is een belangrijke functie die bij iedere widget anders is en bij iedere widget wordt opgeroepen. Daarom is de *GTKWidget* klasse abstract gemaakt waarbij de specifieke widgets zelf puur subklassen zijn van deze abstracte klasse, zoals weergegeven in figuur 23 [47].

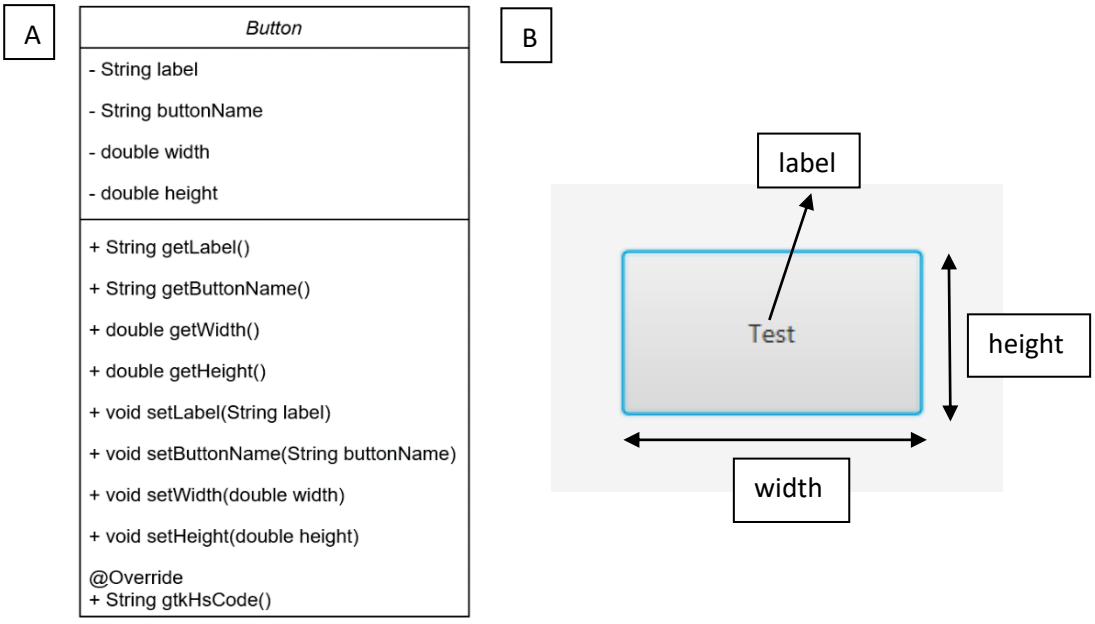


Figuur 23: Schematische weergave van de abstracte-hoofdklasse en de specifieke subklassen van de widgets

4.8.1 Button

Figuur 24 (a) geeft de zelfgemaakt klasse van de Button-widget weer. De belangrijkste eigenschappen die overeenkomen zijn:

- width: breedte van de Button,
- height: hoogte van de Button,
- label: de tekst die in de Button komt.



Figuur 24: Custom klasse Button widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

Ook valt het op dat er een aparte *buttonName*-attribuut is, naast de *name*-attribuut uit de bovenliggende GTKWidget klasse. Dit is omdat een Button met breedte en hoogte in GTK bestaat uit een aparte container en de primitieve Button-widget zelf. Om een Button een bepaalde breedte en hoogte te geven in GTK, moet de Button in een Container geplaatst worden met de gewenste afmetingen. Deze container krijgt de originele *name*-eigenschap van de bovenliggende GTKWidget klasse aangezien dit het *top-level* element is dat moet worden aangesproken als de volledige Button moet worden aangesproken. Concreet is de Haskell-code voor het genereren van een Button met bepaalde hoogte en breedte weergegeven in de code-snippet van figuur 25 (a). De codegeneratie zelf in Java wordt weergegeven in figuur 25 (b) en gebeurt, zoals eerder aangehaald met templates en de StringBuilder.

A

```
1  firstButton  <- Gtk.buttonNewWithLabel "Test"          --Primitieve Button declaratie
2  firstButtonContainer  <- Gtk.boxNew OrientationHorizontal 1   --Container voor Button
3  Gtk.set firstButtonContainer  [Gtk.widgetWidthRequest := 141, --Breedte/hoogte voor
4                                 Gtk.widgetHeightRequest := 78] --container instellen
5  Gtk.boxPackStart firstButtonContainer  firstButton True True 0 --Button in container doen
```

B

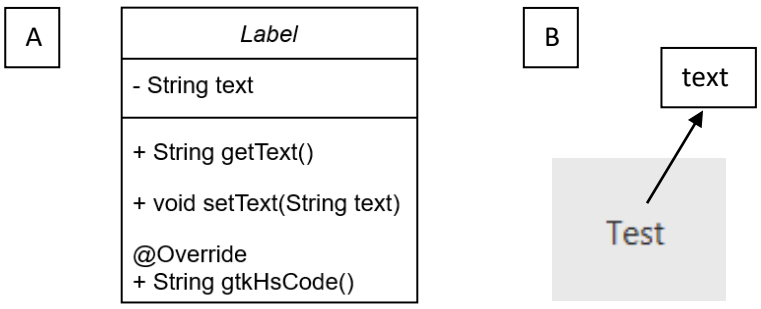
```
1 @Override
2 public String gtkHsCode(){
3     StringBuilder template = new StringBuilder();
4     if (!Objects.equals(label, ""))
5     template.append("${BUTTONNAME}
6                      <- Gtk.buttonNewWithLabel \"${LABEL}\"\n  ");
7     else template.append("${BUTTONNAME} <- Gtk.buttonNew \n  ");
8     template.append("${BUTTONCONTAINERBOXNAME}
9                      <- Gtk.boxNew OrientationHorizontal 1\n  ");
10    template.append("Gtk.set ${BUTTONCONTAINERBOXNAME}
11                     [Gtk.widgetWidthRequest :=${WIDTH},
12                     Gtk.widgetHeightRequest :=${HEIGHT}]\n  ");
13    template.append("Gtk.boxPackStart ${BUTTONCONTAINERBOXNAME}
14                     ${BUTTONNAME} True True 0\n   \n  ");
15
16    Map<String, Object> toInsert = new HashMap<String, Object>();
17    toInsert.put("BUTTONCONTAINERBOXNAME", super.getName());
18    toInsert.put("WIDTH", (int)width);
19    toInsert.put("HEIGHT", (int)height);
20    toInsert.put("BUTTONNAME", buttonName);
21    toInsert.put("LABEL", label);
22
23    return StringFormat.format(template.toString(),toInsert);
24 }
```

Figuur 25: Button-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.2 Label

Figuur 26 (a) geeft de zelfgemaakt klasse van het Label-widget weer. Er is slechts één eigenschap dat deze widget heeft (buiten de eigenschappen van GTKWidget) en dat is:

- Tekst: de tekstwaarde wat de Label zal krijgen.



Figuur 26: Custom klasse Label widget (A) met de enige eigenschap, text, gevisualiseerd (B)

Er zal dus enkel rekening gehouden moeten worden met de tekst om een Label-widget aan te maken in GTK. Hierdoor is een Label een eenvoudig widget dat met één regel code in Haskell wordt aangemaakt, zoals te zien is in figuur 27 (a). De codegeneratie zelf in Java is weergegeven in figuur 27 (b).

A

```haskell
firstLabel <- Gtk.labelNew (Just "Test") -- Label declaratie
```

B

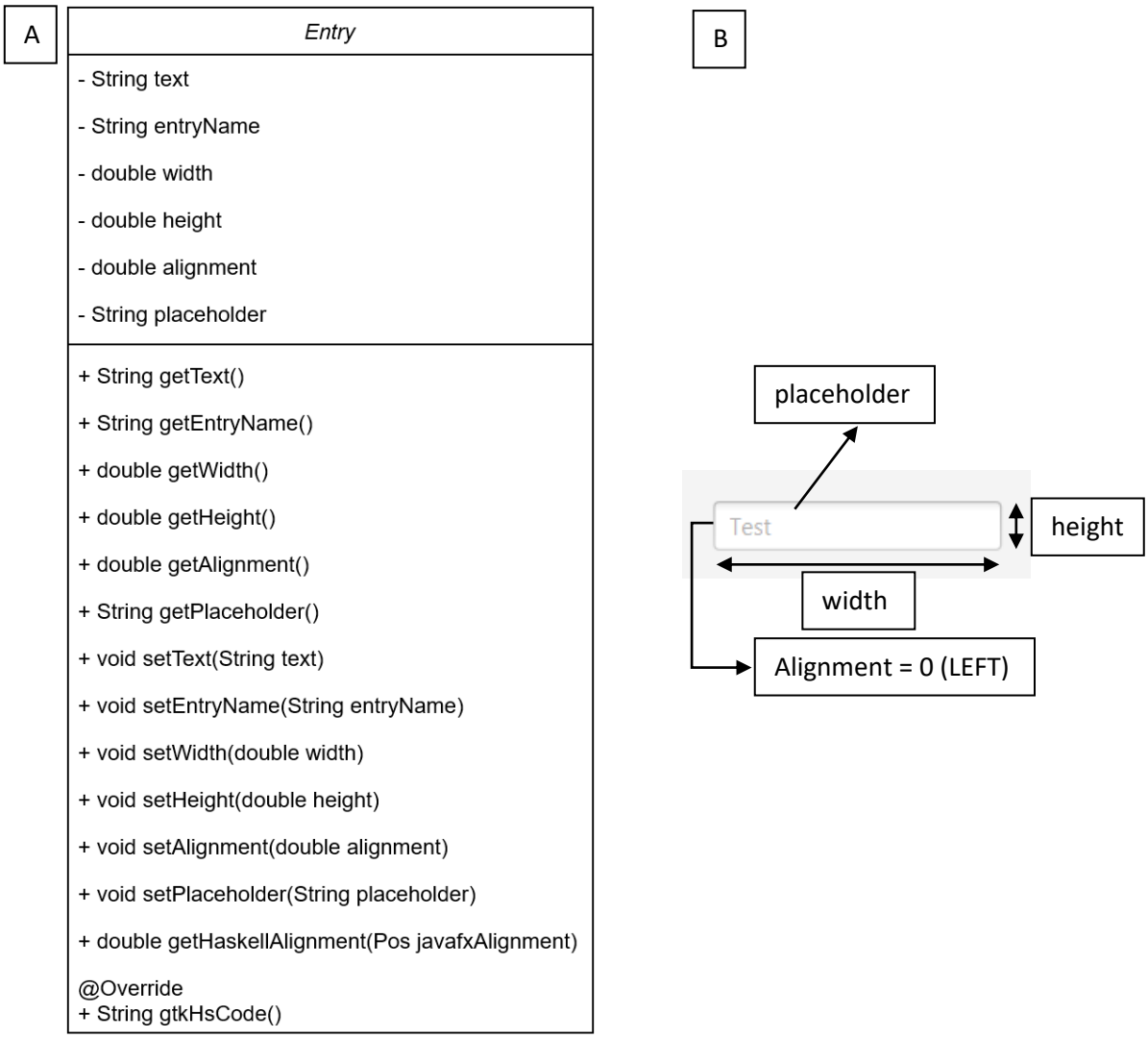
```java
@Override
public String gtkHsCode(){
    String template = "${LAYOUTNAME}  <- Gtk.layoutNew
    (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)\n  \n  ";
    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("LAYOUTNAME", super.getName());
    return StringFormat.format(template, toInsert);
}
```

Figuur 27: Label-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.3 Entry

Figuur 28 (a) geeft de zelfgemaakt klasse van het Entry-widget weer. De belangrijkste eigenschappen die overeenkomen zijn:

- width: breedte van de Entry,
- height: hoogte van de Entry,
- placeholder: de tekst in de Entry die dient als hint,
- alignment: de plaats waar de tekst in de Entry komt te staan (links, midden, rechts).



Figuur 28: Custom klasse Entry widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

De Entry-widget heeft net zoals de Button-widget een *entryName*-attribuut (*buttonName*). De reden hiervoor is exact dezelfde als die voor de Button, dus om te gebruiken in een container die voor de breedte en hoogte zorgt. Verder is er ook nog een *text*-attribuut. Met dit attribuut kan er al een tekst worden geïnstantieerd in de Entry. Deze tekst kan worden aangepast en is dus niet exact hetzelfde als het *placeholder*-attribuut. Samen met de attributen *placeholder* en *alignment* is *text* optioneel. Dit wil zeggen dat het *text*-attribuut niet altijd wordt aangemaakt. In figuur 29 (a) is er bijvoorbeeld te zien dat *placeholder* en *alignment* wel zijn aangemaakt, maar *text* niet. De codegeneratie zelf in Java is weergegeven in figuur 29 (b).

A

```
firstEntry <- Gtk.entryNew                                  -- Primitieve Entry declaratie
Gtk.entrySetPlaceholderText firstEntry (Just "Test")        -- Placeholder van Entry
Gtk.entrySetAlignment firstEntry 0.5                        -- Alignment van tekst van Entry
firstEntryContainer <- Gtk.boxNew OrientationHorizontal 1   -- Container voor Entry
Gtk.set firstEntryContainer [Gtk.widgetWidthRequest := 171, -- Breedte/hoogte voor
                             Gtk.widgetHeightRequest := 24] -- container instellen
Gtk.boxPackStart firstEntryContainer firstEntry True True 0 -- Entry in container doen
```

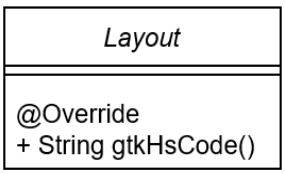
B

```
@Override
public String gtkHsCode() {
    StringBuilder template = new StringBuilder();
    template.append("${ENTRYNAME} <- Gtk.entryNew\n  ");
    if (!Objects.equals(text, ""))
    template.append("Gtk.entrySetText ${ENTRYNAME} \"${TEXT}\" \n  ");
    if (!Objects.equals(placeholder, ""))
    template.append("Gtk.entrySetPlaceholderText
                    ${ENTRYNAME} (Just \"${PLACEHOLDER}\")\n  ");
    if (!Objects.equals(alignment, 0))
    template.append("Gtk.entrySetAlignment
                    ${ENTRYNAME} ${ALLIGNMENT} \n  ");
    template.append("${ENTRYCONTAINERBOXNAME}
                    <- Gtk.boxNew OrientationHorizontal 1\n  ");
    template.append("Gtk.set ${ENTRYCONTAINERBOXNAME}
                    [Gtk.widgetWidthRequest :=${WIDTH},
                    Gtk.widgetHeightRequest :=${HEIGHT}]\n  ");
    template.append("Gtk.boxPackStart
                    ${ENTRYCONTAINERBOXNAME} ${ENTRYNAME}
                    True True 0\n   \n  ");

    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("ENTRYCONTAINERBOXNAME", super.getName());
    toInsert.put("ENTRYNAME", entryName);
    toInsert.put("WIDTH", (int)width);
    toInsert.put("HEIGHT", (int)height);
    toInsert.put("ALLIGNMENT", alignment);
    toInsert.put("PLACEHOLDER", placeholder);
    toInsert.put("TEXT", text);

    return StringFormat.format(template.toString(), toInsert);
}
```

Figuur 29: Entry-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.4 Layout

Figuur 30 toont de klasse van de Layout-widget. Deze klasse heeft geen speciale eigenschappen of attributen. De klasse bestaat uit enkel de methode om de Haskell-code te genereren. Een gevolg hiervan is dat de Haskell-code zelf ook minimaal is, zoals te zien is op figuur 31 (a). De codegeneratie zelf in Java is weergegeven in figuur 31 (b).



Figuur 30: Custom klasse Layout widget

A

```haskell
firstLayout <- Gtk.layoutNew (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)
```

B

```
@Override
public String gtkHsCode(){
    String template = "${LAYOUTNAME}  <- Gtk.layoutNew
                      (Nothing::Maybe Adjustment)
                      (Nothing::Maybe Adjustment)\n  \n  ";
    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("LAYOUTNAME", super.getName());
    return StringFormat.format(template, toInsert);
}
```

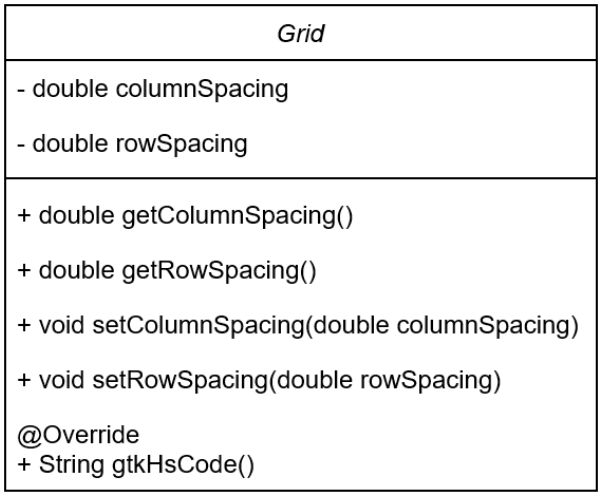
Figuur 31: Layout-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.5 Grid

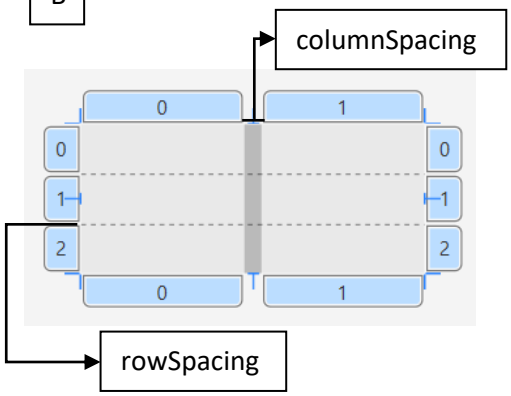
Figuur 32 (a) geeft de zelfgemaakt klasse van de Grid-widget weer. De belangrijkste eigenschappen die overeenkomen zijn:

- columnSpacing: ruimte tussen de kolommen,
- rowSpacing: ruimte tussen de rijen.

A



B



Figuur 32: Custom klasse Grid Widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

Zowel *columnSpacing* als *rowSpacing* zijn optionele eigenschappen. Dit betekent dat beide eigenschappen niet altijd een waarde moeten hebben. Figuur 33 (a) toont dat er wel een *columnSpacing*-eigenschap is (regel 4), maar geen *rowSpacing*-eigenschap. Ook is het belangrijk om zowel de kolommen als de rijen homogeen te maken. Dit wil zeggen dat elk vakje in de Grid dezelfde grootte heeft. De codegeneratie zelf in Java is ten slotte weergegeven in figuur 33 (b).

A

```
1   firstGrid <- Gtk.gridNew                        -- Grid declaratie
2   Gtk.gridSetColumnHomogeneous firstGrid True     -- Kolommen zelfde breedte geven
3   Gtk.gridSetRowHomogeneous firstGrid True        -- Rijen zelfde hoogte geven
4   Gtk.gridSetColumnSpacing firstGrid 10           -- Ruimte tussen kolommen geven
```

B

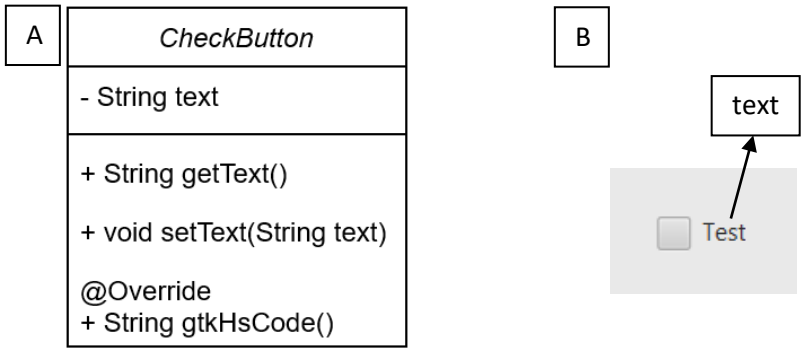
```
1 @Override
2 public String gtkHsCode(){
3     StringBuilder template = new StringBuilder();
4     template.append("${GRIDNAME} <- Gtk.gridNew\n  ");
5     template.append("Gtk.gridSetColumnHomogeneous ${GRIDNAME} True\n  ");
6     template.append("Gtk.gridSetRowHomogeneous ${GRIDNAME} True\n  ");
7     if ((int)columnSpacing != 0)
8     template.append("Gtk.gridSetColumnSpacing ${GRIDNAME} ${COLUMNSPACING}\n  ");
9     if ((int)rowSpacing != 0)
10    template.append("Gtk.gridSetRowSpacing ${GRIDNAME} ${ROWSPACING}\n  ");
11    template.append("\n  ");
12
13    Map<String, Object> toInsert = new HashMap<String, Object>();
14    toInsert.put("GRIDNAME", super.getName());
15    toInsert.put("COLUMNSPACING", (int)columnSpacing);
16    toInsert.put("ROWSPACING", (int)rowSpacing);
17
18    return StringFormat.format(template.toString(), toInsert);
19 }
```

Figuur 33: Grid-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.6 CheckButton

Figuur 34 (a) geeft de zelfgemaakt klasse van de CheckButton-widget weer. De enige eigenschap die overeenkomt is:

- text: de tekstwaarde die langs de check box verschijnt.



Figuur 34: Custom klasse CheckButton widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

Er zal dus enkel rekening gehouden moeten worden met de tekst om een CheckButton-widget aan te maken in GTK. Hierdoor is een CheckButton een eenvoudig widget dat met één regel code wordt aangemaakt, zoals te zien is in figuur 35 (a). De codegeneratie zelf in Java is weergegeven in figuur 35 (b).

A

```haskell
firstCheckButton <- Gtk.checkButtonNewWithLabel "Test" -- CheckButton declaratie
```

B

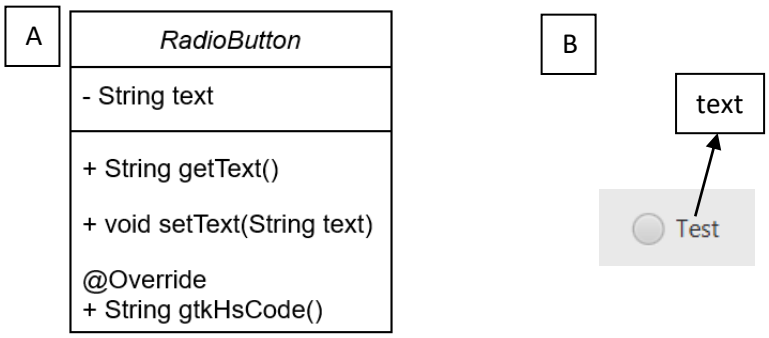
```java
@Override
public String gtkHsCode(){
    String template;
    if (!Objects.equals(text, "")){
        template = "${CHECKBUTTONNAME}
                    <- Gtk.checkButtonNewWithLabel
                    \"${TEXT}\"\n  \n  ";
    }else{
        template = "${CHECKBUTTONNAME}
                    <- Gtk.checkButtonNew\n  \n  ";
    }

    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("CHECKBUTTONNAME", super.getName());
    toInsert.put("TEXT", text);

    return StringFormat.format(template,toInsert);
}
```

Figuur 35: CheckButton-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.7 *RadioButton*

Figuur 36 (a) geeft de zelfgemaakt klasse van de RadioButton-widget weer. De enige eigenschap die overeenkomt is alweer:

- text: de tekstwaarde die langs de check box verschijnt.



Figuur 36: Custom klasse RadioButton widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

Een RadioButton lijkt bovenmatig veel op de CheckButton. Het enige verschil is dat een RadioButton tot bij een ToggleGroup kan behoren. Een ToggleGroup is een verzameling van RadioButtons waarin maar één RadioButton kan geselecteerd zijn. Figuur 37 (a) toont de Haskell-code om 2 RadioButtons aan te maken en vervolgens om deze in een ToggleGroup te steken. De codegeneratie zelf in Java wordt in figuur 37 (b) weergegeven.

A

```haskell
radioButton1 <- Gtk.radioButtonNewWithLabelFromWidget
                (Nothing::Maybe Gtk.RadioButton) "Test" -- RadioButton 1 declaratie
radioButton2 <- Gtk.radioButtonNewWithLabelFromWidget
                (Nothing::Maybe Gtk.RadioButton) "Test2" -- RadioButton 2 declaratie
Gtk.radioButtonSetGroup radioButton1 [radioButton2] -- Toggle Group voor de RadioButtons
```

B

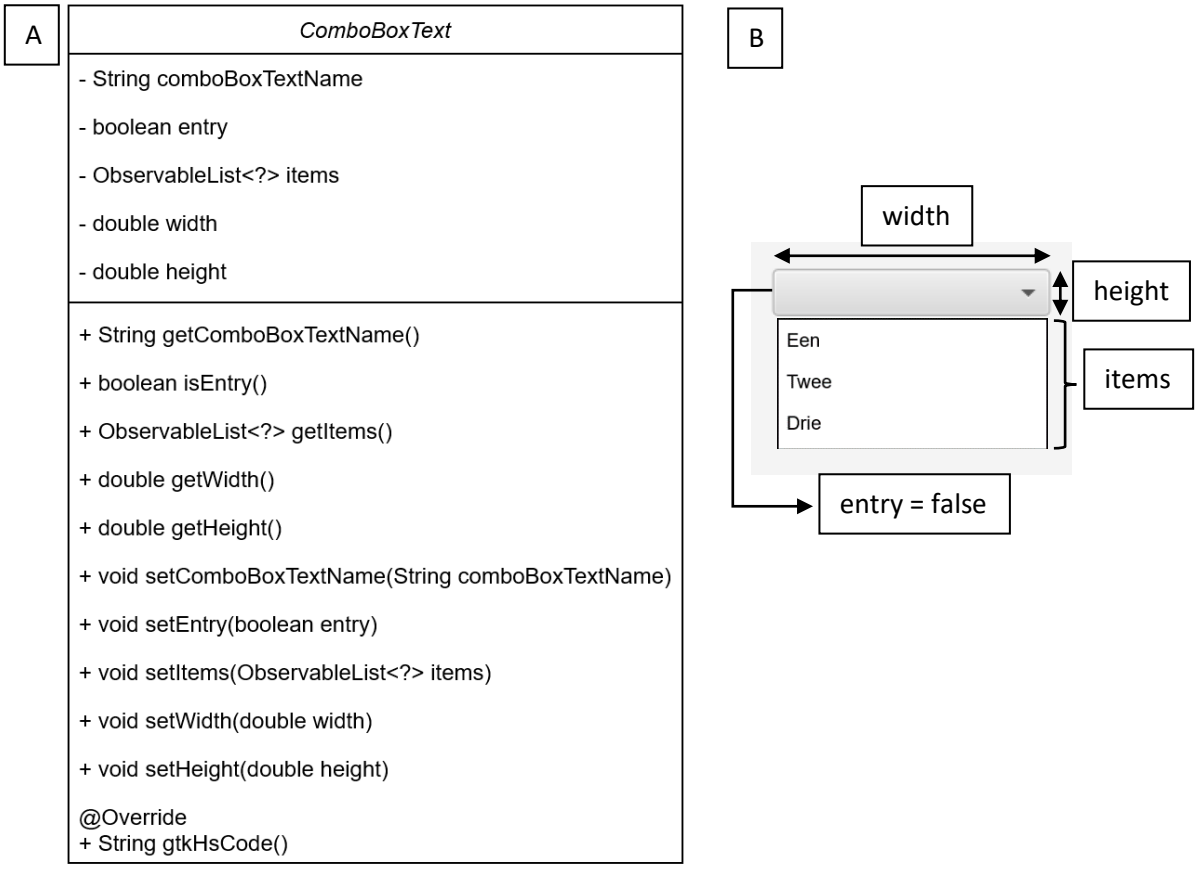
```java
@Override
public String gtkHsCode(){
    StringBuilder template = new StringBuilder();
    if (!Objects.equals(text, ""))
    template.append("${RADIOBUTTONNAME}
                    <- Gtk.radioButtonNewWithLabelFromWidget
                    (Nothing::Maybe Gtk.RadioButton) \"${TEXT}\"\n  ");
    else template.append("${RADIOBUTTONNAME} <-
                         Gtk.radioButtonNewFromWidget
                         (Nothing::Maybe Gtk.RadioButton)\n  ");
    template.append("\n  ");

    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("RADIOBUTTONNAME", super.getName());
    toInsert.put("TEXT", text);

    return StringFormat.format(template.toString(), toInsert);
}
```

Figuur 37: RadioButton-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.8 ComboBoxText

Figuur 38 (a) geeft de zelfgemaakt klasse van de ComboBoxText-widget weer. De belangrijkste eigenschappen die overeenkomen zijn:

- entry: Boolean die bepaald als er kan getypt worden in de ComboBoxText zelf,
- items: lijst van waardes waaruit gekozen kan worden,
- width: breedte van de ComboBoxText,
- height: hoogte van de ComboBoxText.



Figuur 38: Custom klasse ComboBoxText widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

De ComboBoxText-widget heeft zoals de Button en de Entry ook een *comboBoxTextName*-attribuut (*buttonName* en *entryName*). De reden hiervoor is, net zoals bij de Button, om te gebruiken in een container die voor de breedte en hoogte zorgt. In Haskell moet elk item in de ComboBoxText met een aparte regel worden aangemaakt. Dit in combinatie met de container die moet worden aangemaakt, zorgt ervoor dat de ComboBoxText relatief veel regels code heeft om aangemaakt te worden, zoals te zien is in figuur 39 (a). De codegeneratie zelf in Java wordt in figuur 39 (b) weergegeven.

A

```
comboBoxText <- Gtk.comboBoxTextNew -- primitieve ComboBoxText declaratie
Gtk.comboBoxTextAppendText comboBoxText "Een" -- Item toevoegen
Gtk.comboBoxTextAppendText comboBoxText "Twee" -- Item toevoegen
Gtk.comboBoxTextAppendText comboBoxText "Drie" -- Item toevoegen
comboBoxTextContainer <- Gtk.boxNew OrientationHorizontal 1 -- Container voor ComboBoxText
Gtk.set comboBoxTextContainer [Gtk.widgetWidthRequest := 150, -- Breedte/hoogte voor
                               Gtk.widgetHeightRequest := 24] -- container instellen
Gtk.boxPackStart comboBoxTextContainer comboBoxText
                                    True True 0 -- ComboBoxText in container doen
```

B

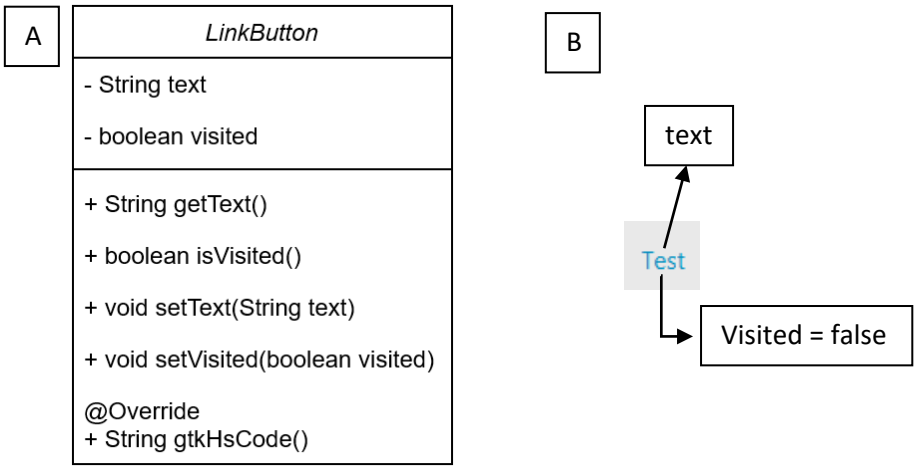
```
@Override
public String gtkHsCode(){
    StringBuilder comboBoxTextItems = new StringBuilder("");
    for (Object obj : items){
        String item = obj.toString();
        comboBoxTextItems.append("Gtk.comboBoxTextAppendText ").
        append(comboBoxTextName).append(" ").append("\"").
        append(item).append("\"").append("\n  ");
    }

    StringBuilder template = new StringBuilder();
    if(entry) template.append("${COMBOBOXTEXTNAME}
                               <- Gtk.comboBoxTextNew \n  ");
    else template.append("${COMBOBOXTEXTNAME}
                          <- Gtk.comboBoxTextNewWithEntry \n  ");
    template.append(comboBoxTextItems);
    template.append("${COMBOBOXCONTAINERNAME}
                     <- Gtk.boxNew OrientationHorizontal 1\n  ");
    template.append("Gtk.set ${COMBOBOXCONTAINERNAME}
                     [Gtk.widgetWidthRequest :=${WIDTH},
                     Gtk.widgetHeightRequest :=${HEIGHT}]\n  ");
    template.append("Gtk.boxPackStart ${COMBOBOXCONTAINERNAME}
                     ${COMBOBOXTEXTNAME} True True 0\n   ");

    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("COMBOBOXCONTAINERNAME", super.getName());
    toInsert.put("WIDTH", (int)width);
    toInsert.put("HEIGHT", (int)height);
    toInsert.put("COMBOBOXTEXTNAME", comboBoxTextName);

    return StringFormat.format(template.toString(),toInsert);
}
```

Figuur 39: ComboBoxText-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

4.8.9 LinkButton

Figuur 40 (a) geeft de zelfgemaakt klasse van de LinkButton-widget weer. De belangrijkste eigenschappen die overeenkomen zijn:

- text: tekstwaarde van de LinkButton,
- visited: Boolean die bepaald als de LinkButton al is bezoekt of niet (visueel).



Figuur 40: Custom klasse LinkButton widget (A) met de belangrijkste eigenschappen gevisualiseerd (B)

De LinkButton-widget is bijna identiek aan de Label-widget en bestaat dus ook uit relatief weinig regels Haskell-code, zoals te zien is in figuur 41 (a). Hier is er echter nog een extra attribuut, namelijk *visited*. Dit attribuut is optioneel en moet dus niet altijd worden aangemaakt. Als dit attribuut toch wordt aangemaakt, komt er een extra regel Haskell-code bij. De codegeneratie zelf in Java wordt in figuur 41 (b) weergegeven.

A

```haskell
firstLinkButton <- Gtk.linkButtonNewWithLabel "" (Just "Test")
```

B

```java
@Override
public String gtkHsCode(){
    StringBuilder template = new StringBuilder();
    if(!Objects.equals(text, ""))
    template.append("${LINKBUTTONNAME}
                    <- Gtk.linkButtonNewWithLabel
                    \"\" (Just \"${TEXT}\")\n  ");
    else template.append("${LINKBUTTONNAME}
                         <- Gtk.linkButtonNew \"\"\n  ");
    if (visited) template.append("Gtk.linkButtonSetVisited
                                  ${LINKBUTTONNAME} True\n  ");
    template.append("\n  ");

    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("LINKBUTTONNAME", super.getName());
    toInsert.put("TEXT", text);

    return StringFormat.format(template.toString(),toInsert);
}
```

Figuur 41: LinkButton-declaratie in Haskell (A) en de gtkHsCode() functie van de codegenerator (B)

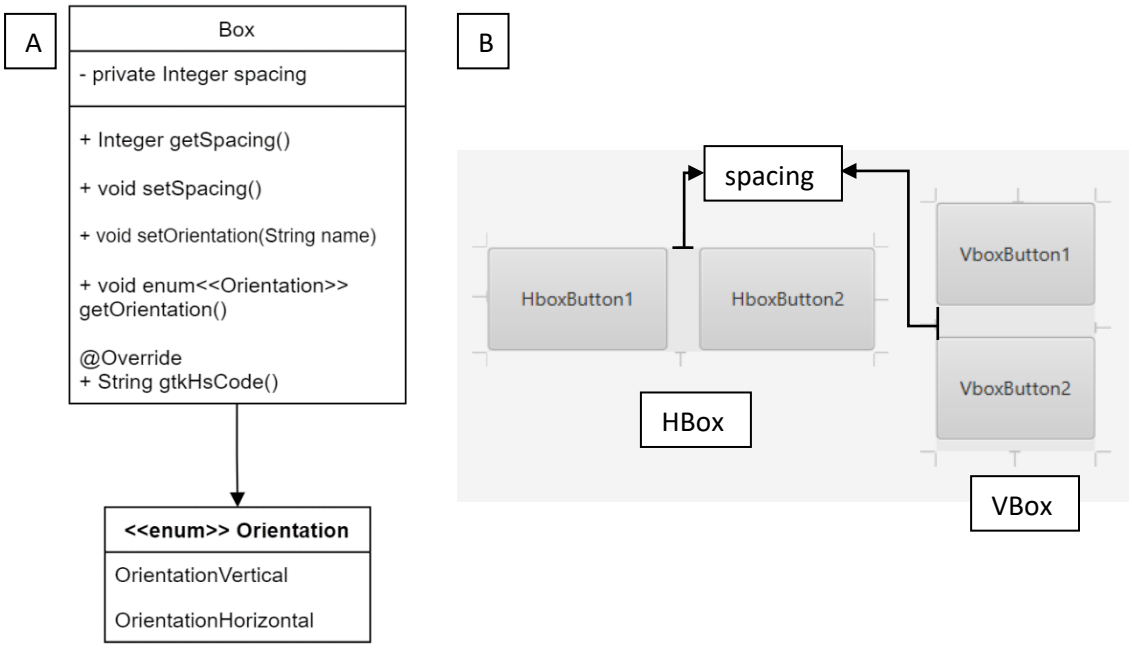
4.8.10 HBox en VBox

De HBox en VBox zijn beide containers die widgets respectievelijk horizontaal en verticaal kunnen stapelen. De HBox en VBox zijn daarnaast vergelijkbaar met een GridPane met één rij of één kolom die alleen horizontaal of alleen verticaal kan groeien.

Aangezien de HBox en VBox soortgelijk zijn en dezelfde eigenschappen delen, is besloten een generieke zelfgemaakte klasse Box te gebruiken waarbij de oriëntatie (verticaal of horizontaal) als *enum* ingesteld kan worden. Dit principe sluit mooi aan bij het advies van de documentatie van gi-gtk [48] die aanhaalt om een generieke Box-widget te gebruiken en de oriëntatie mee te geven zoals te zien is in figuur 42. De documentatie haalt immers ook aan dat de HBox en VBox gedeprecieerd zijn en er daarom met de GTK-Box-widget moet worden gewerkt.

Figuur 42 (a) geeft de zelfgemaakt klasse van de Box-widget met de Orientation-*enum* weer. De belangrijkste eigenschap die overeenkomt uit JavaFX en gi-gtk zijn:

- spacing: ruimte tussen de widgets in de HBox of VBox.



Figuur 42: Custom Box-klasse (A) met de belangrijkste eigenschappen gevisualiseerd (B)

Figuur 43 (b) geeft de gtkHsCode-functie van een Box weer. Hierbij wordt op een eenvoudige manier de Orientation-*enum* direct in een String-vorm geïnjecteerd om zo de Haskell-code te genereren dat te zien is in figuur 43 (a).

A

```
1 -- VBox                    Orientation             spacing
2 vBox_container <- Gtk.boxNew Gtk.OrientationVertical 30
3
4 -- HBox                    Orientation             spacing
5 hBox_container <- Gtk.boxNew Gtk.OrientationHorizontal 30
```

B

```
@Override
public String gtkHsCode(){
    String template = "${NAME}
                       <- Gtk.boxNew
                       Gtk.${ORIENTATION}
                       ${SPACING} \n  \n";

    Map<String, Object> toInsert = new HashMap<String, Object>();
    toInsert.put("NAME", super.getName());
    toInsert.put("ORIENTATION", orientation);
    toInsert.put("SPACING", spacing);

    return StringFormat.format(template,toInsert);
}
```

Figuur 43: VBox/HBox declaratie in Haskell (A) en de gtkHsCode() functie van de Box in de codegenerator (B)

4.8.11 Notebook

De Notebook-container uit GTK is vergelijkbaar met de TabPane uit JavaFX. Met deze container is het mogelijk om meerdere tabbladen te implementeren in één bepaald gebied met een voor-gedefinieerde grootte.

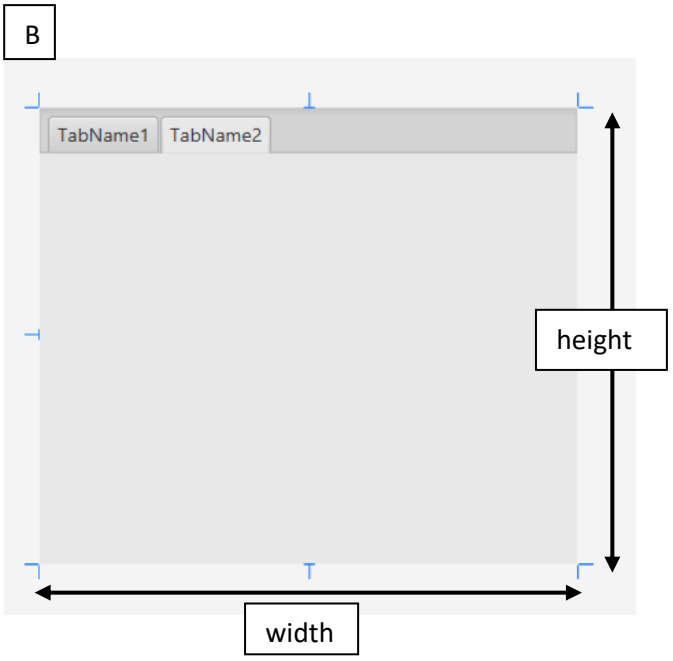
Figuur 44 (a) geeft de zelfgemaakt klasse van de Notebook-widget weer. De belangrijkste eigenschappen die overeenkomen zijn:

- width: breedte van de Notebook,
- height: hoogte van de Notebook.

A

Notebook
- private Integer width - private Integer height - private String notebookName
+ Integer getWidth() + Integer getHeight() + String getNotebookName() + void setNotebookName(String name) + void setWidth(int width) + void setHeight(int height) @Override + String gtkHsCode()

B



Figuur 44: Custom Notebook-klasse (A) met de belangrijkste eigenschappen gevisualiseerd (B)

Net als bij de Button-Widget in het deel “4.8.1 Button” is hier ook een aparte *name*-attribuut geïmplementeerd, naast het *name*-attribuut uit de bovenliggende GTKWidget-klasse. Dit is omdat een Notebook met breedte en hoogte in GTK bestaat uit een aparte container en de primitieve Notebook-widget zelf zoals ook te zien is in de Haskell-declaratie van de Notebook in figuur 45(a). Bovendien valt ook in figuur 44 (b) op dat de tab-namen (“TabName1” en “TabName2”) niet als eigenschap worden aangeduid en ook niet als *data-member* in de zelfgemaakte klasse worden gebruikt. Dit is omdat de tab-namen in JavaFX en GTK als een Label worden voorgesteld. Daarom zal de link tussen de tab-namen en de Notebook gemaakt worden als de relaties worden gemaakt. Het parsen van de Notebook en de relatie ervan wordt in hoofdstuk “4.10.3 NotebookRelation” verduidelijkt. Ten slotte toont figuur 45 (b) de codegeneratie in Java.

```
1 noteBook <- Gtk.notebookNew --notebook aanmaken
2 noteBook <- Gtk.boxNew OrientationHorizontal 1 --omliggende container maken
3 Gtk.set noteBook [Gtk.widgetWidthRequest := 570, Gtk.widgetHeightRequest := 369]
4 Gtk.boxPackStart noteBook noteBook_2092241187 True True 0 --omringen in container
```

A

```
 1 @Override
 2 public String gtkHsCode(){
 3     StringBuilder template = new StringBuilder();
 4     template.append("${NOTEBOOKNAME} <- Gtk.notebookNew \n  ");
 5     template.append("${NOTEBOOKCONTAINERNAME}
 6                     <- Gtk.boxNew OrientationHorizontal 1\n  ");
 7     template.append("Gtk.set ${NOTEBOOKCONTAINERNAME}
 8                      [Gtk.widgetWidthRequest :=${WIDTH},
 9                      Gtk.widgetHeightRequest :=${HEIGHT}]\n  ");
10     template.append("Gtk.boxPackStart ${NOTEBOOKCONTAINERNAME}
11                      ${NOTEBOOKNAME} True True 0\n   \n  ");
12
13     Map<String, Object> toInsert = new HashMap<String, Object>();
14     toInsert.put("NOTEBOOKCONTAINERNAME", super.getName());
15     toInsert.put("WIDTH", (int)width);
16     toInsert.put("HEIGHT", (int)height);
17     toInsert.put("NOTEBOOKNAME", notebookName);
18
19     return StringFormat.format(template.toString(), toInsert);
20 }
```

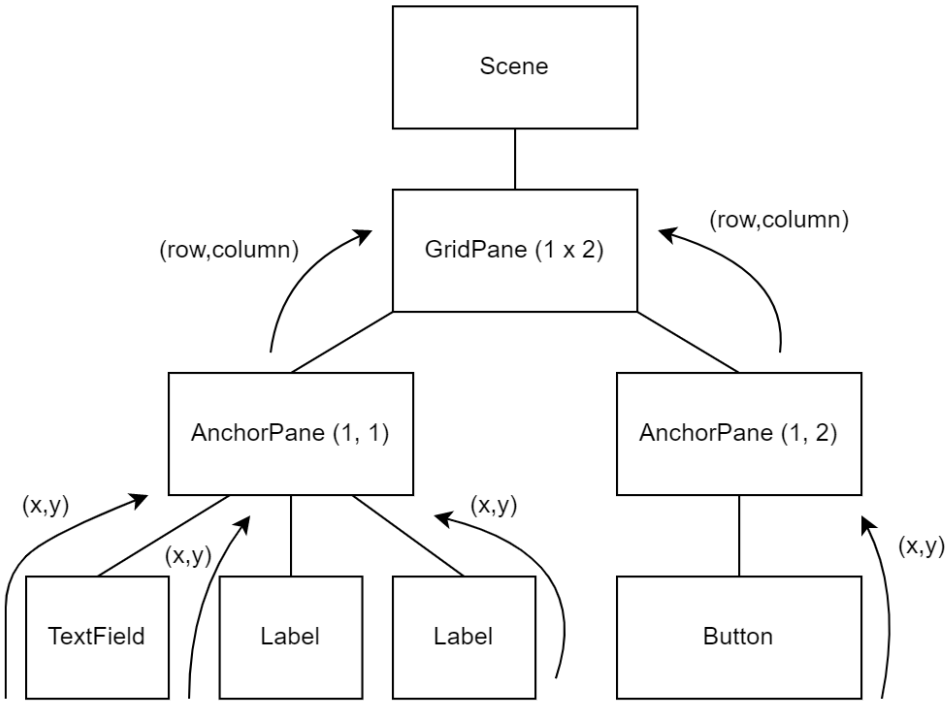
B

Figuur 45: Notebook declaratie in Haskell (A) en de gtkHsCode()-functie van de Notebook in de codegenerator (B)

4.9 Relaties parsen

Een relatie geeft het verband of verhouding tussen twee elementen weer. In de context van GUI-programmeren geeft een relatie het verband tussen twee of meerdere *Nodes* of GUI-elementen weer. Relaties geven meer precies weer waar een bepaalde GUI-element zich bevindt in een groter overliggende container-element.

Relaties kunnen *geparsed* worden door telkens naar het bovenliggend element, oftewel *Parent-Node*, te kijken tijdens het doorlopen van de FXML-boom. Dit is schematisch weergegeven in figuur 46. Hierbij wordt er naar de *Parent-Node* gekeken om te kijken hoe de huidige *child-Node* zich verhoudt t.o.v. de *Parent-Node*. In het geval van de *Nodes* die in een AnchorPane zitten, is het belangrijk om te kijken naar de x- en y-coördinaten van de *child-Nodes*. In het geval waarbij de *parent-Node* een GridPane is, moet worden gekeken in welke rij en kolom de huidige *child-Node* zich bevindt.



Figuur 46: Schematische weergave voor de relaties te parsen in de FXML-boomstructuur

Figuur 47 geeft de code-*snippet* weer van hoe een relatie wordt *geparsed* en wordt aangemaakt tijdens de dump-functie. Om een relatie te identificeren, is het ten eerste belangrijk dat er bij de huidige JavaFX-*Node* naar de bovenliggende *Node* wordt gekeken m.b.v. de *getParent()*-methode. Vervolgens kan de *Parent-Node* geïdentificeerd worden door met *instanceof* te werken, net als hoe de widgets worden geïdentificeerd bij hoofdstuk “4.7 Widgets Parsen”. Daarna worden de eigenschappen verkregen die bij de desbetreffende relatie horen, om vervolgens de relatie zelf aan te maken. In dit geval gaat het over een AnchorPane waarbij de relatie van een *child-Node* in de AnchorPane-container afhankelijk is van de x- en y-coördinaten van die *child-Node*.

```
1 private static void dump(Node n, int depth) {
2     for (int i = 0; i < depth; i++) System.out.print("   ");
3
4     //PARSE EN CREEER WIDGETS-CONTROLS
5     if(n instanceof javafx.scene.control.Button){
6         ...
7     }else if (n instanceof AnchorPane){
8         ...
9     }
10
11    //RELATIES IDENTIFICEREN EN AANMAKEN
12    if(n.getParent() instanceof AnchorPane){
13        //Parent-hashcode eigenschap fetchen
14        String APParentName = getParentName(n.getParent().hashCode());
15
16        //eigenschappen specefieke relatie fetchen
17        int x = (int)currentNode.getLayoutX();
18        int y = (int)currentNode.getLayoutY();
19
20        //Relatie voor een Layout (AnchorPane) aanmaken
21        LayoutRelation layoutRel =
22        new LayoutRelation(APParentName, currentNode.getName(), x, y);
23        relations.add(layoutRel);
24    }else if(n.getParent() instanceof GridPane){
25        ...
26    }
27 }
```

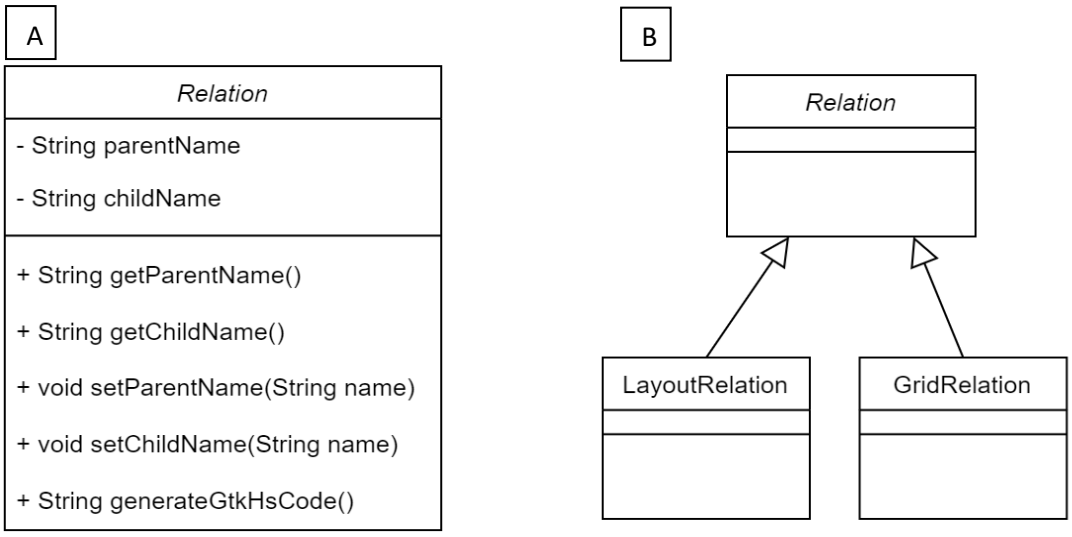
Figuur 47: Code-snippet hoe de relatie wordt geparsed in de dump-functie

4.10 Relation datastructuur

Figuur 48 (a) geeft de zelfgemaakt klasse van Relation weer. De belangrijkste eigenschappen van deze klasse zijn:

- parentName: de declaratie-naam van de Parent-Node of Parent-widget,
- childName: de declaratie-naam van de Child-Node of Child-widget.

Naast de *getter*- en *setter*-methodes bevat de Relation-klasse ook de belangrijke *generateGtkHsCode()*-methode die de Haskell-code van de relatie kan declareren. De *generateGtkHsCode()*-methode is hierbij een belangrijke functie die bij iedere relatie anders is en bij iedere relatie wordt opgeroepen. Daarom is de *Relation*-klasse abstract gemaakt waarbij de specifieke relaties zelf puur subklassen zijn van deze abstracte klasse, zoals weergegeven in figuur 48 (b).

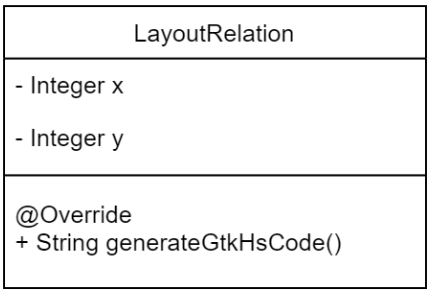


Figuur 48: Custom Relation-klasse (A) en schematische weergave van de abstracte-hoofdklasse en de specifieke subklassen van de relaties (B)

4.10.1 LayoutRelation

Figuur 49 geeft de zelfgemaakt LayoutRelation-klasse weer. De belangrijkste eigenschappen van deze klasse zijn:

- x: de x-coördinaat waar de *child*-widget zich bevindt in de Layout,
- y: de y-coördinaat waar de *child*-widget zich bevindt in de Layout.



Figuur 49: Custom LayoutRelation-klasse gebruikt voor een Layout

Daarnaast geeft figuur 50 een voorbeeld van de Haskell-code weer voor een relatie tussen een *child*-widget en een Layout te declareren.

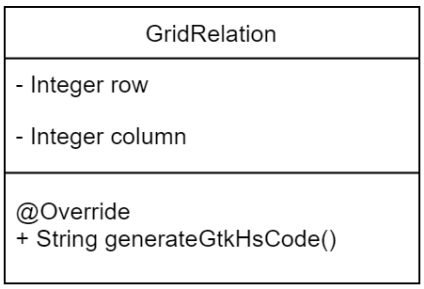
```
1  --  layoutPut parentLayout childWidget x   y
2  Gtk.layoutPut firstLayout  firstButton 236 162
```

Figuur 50: Voorbeeld van Haskell-code uit de generateGtkHsCode()-methode voor de relatie te declareren voor een Button in een Layout met coördinaten x=236 en y=162

4.10.2 GridRelation

Figuur 51 geeft de zelfgemaakt GridRelation-klasse weer. De belangrijkste eigenschappen van deze klasse zijn:

- row: de rij waar de *child*-widget zich bevindt in de Grid,
- column: de kolom waar de *child*-widget zich bevindt in de Grid.



Figuur 51: Custom GridRelation-klasse gebruikt voor een Grid

Daarnaast geeft figuur 52 een voorbeeld van de Haskell-code weer voor een relatie tussen een *child*-widget en een Grid te declareren. Hierbij wordt de *width*- en *height*-eigenschap op 1 gezet aangezien dit een standaardwaarde is.

```
1 --  gridAttach parentGrid    childWidget col row width height
2 Gtk.gridAttach firstGridPane childButton 0   0    1     1
```

Figuur 52: Voorbeeld van Haskell-code uit de generateGtkHsCode()-methode voor de relatie te declareren een Button in een Grid met de rij en kolom op 0.

4.10.3 NotebookRelation

De NotebookRelation *parsen* is op eerste zicht niet vanzelfsprekend en er zijn verschillende redenen die het *parsen* ervan bemoeilijken. Om het probleem te begrijpen is het eerste belangrijk om te kijken hoe de Haskell-code van een relatie opgebouwd is.

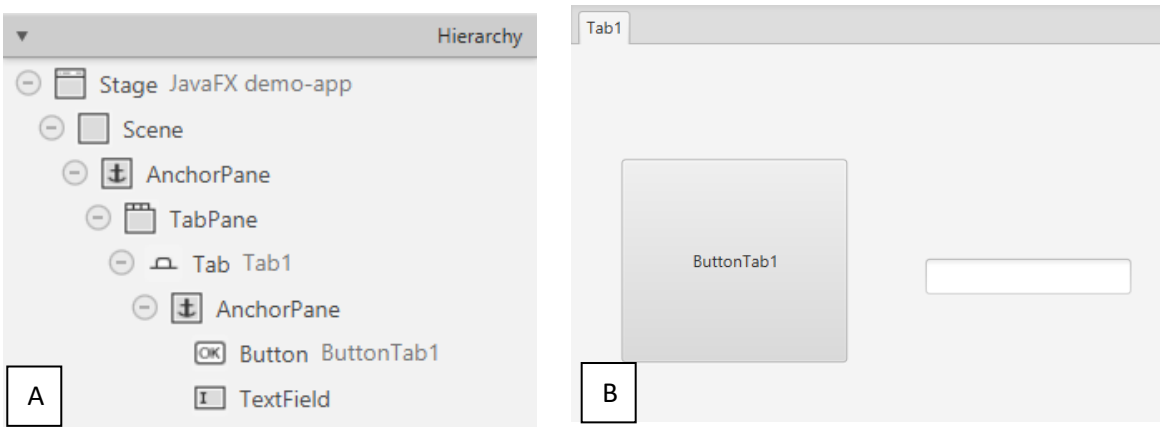
Figuur 53 toont de Haskell-code om een relatie aan te maken voor een Notebook en een Tab (met onderliggende *children*). Hiervoor wordt de *notebookAppendPage*-functie opgeroepen die de volgende parameters moet hebben:

- noteBook: de Notebook in kwestie waar de Tab moet worden gekoppeld,
- childContainer: de inhoud van een Tab, meestal omliggende container,
- label: de Tab-titel van de Tab.
-

```
1 --                    notebook      child             tab-title
2 Gtk.notebookAppendPage firstNotebook gridPaneContainer (Just labelTab)
```

Figuur 53: Voorbeeld van Haskell-code uit de generateGtkHsCode()-methode voor de relatie te declareren tussen een notebook en een tab met titel.

Met de Haskell-code in gedachte kan het probleem van het grotere geheel worden aangekaart. Zo toont figuur 54 (b) een voorbeeld van een simpele TabPane met één enkele Tab die een AnchorPane met een Button en een TextField bevat. Volgens de hiërarchie (van SceneBuilder) in figuur 54 (a) bestaat de TabPane in JavaFX uit een Tab met daarin de *child-Nodes*.



Figuur 54: Hiërarchie in SceneBuilder (A) van het voorbeeld TabPane GUI-applicatie (B)

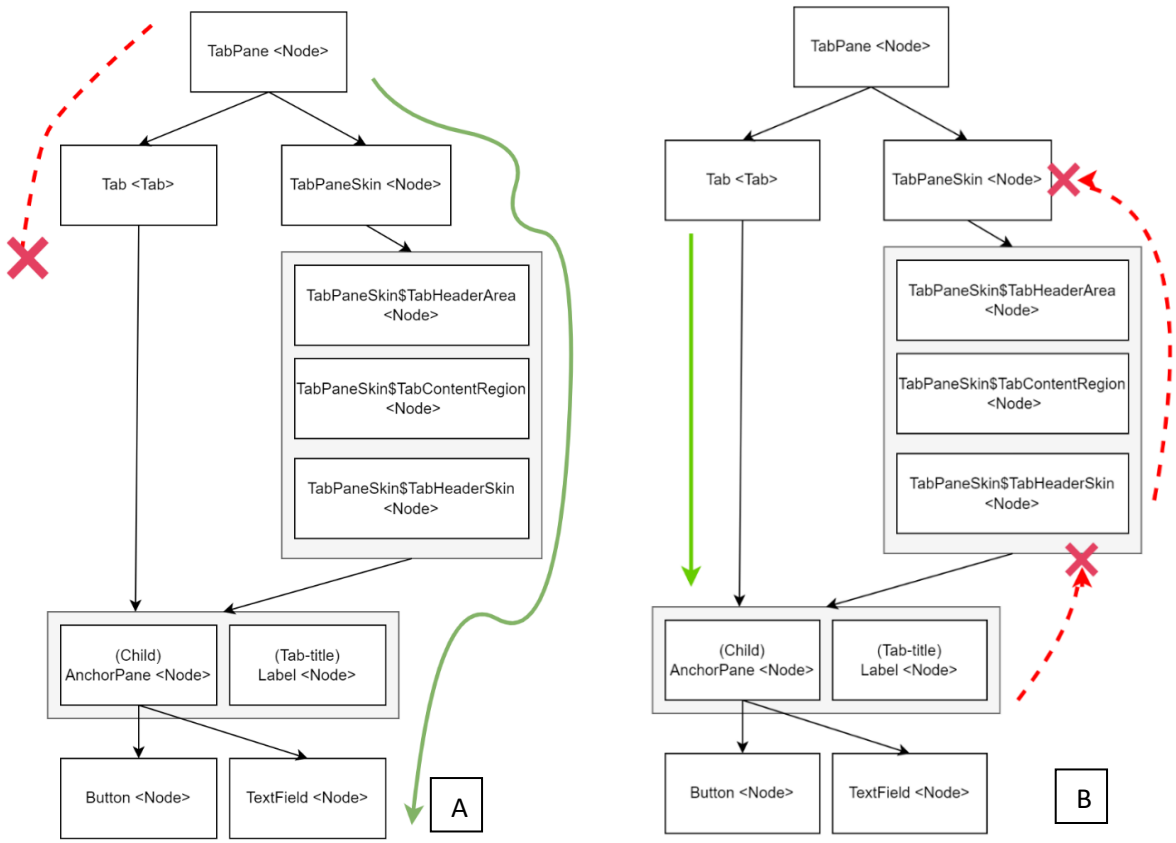
Het herkennen en aanmaken van de *child-Nodes* zoals de Button en TextField in de codegenerator gaat op eerste zicht zonder problemen volgens de dump-functie. De *child-Nodes* zoals de Button en TextField worden hierbij verkregen omdat deze *children* zijn van de zogenaamde TabPaneSkin-*Node* die op zijn beurt een *child-Node* is van de TabPane-*Node* zoals aangetoond in figuur 54 (a) [39]. Hierbij is het opvallend dat er met de Tab niet verder gewerkt wordt in de dump-functie om de inhoud (de *child-Nodes* zoals de Button en TextField) van de TabPane te verkrijgen.

Dit is omdat de Tab zelf niet van het *Node*-Type is [49]. De TabPaneSkin die de *child-Nodes* en Tab-Labels bevat is zelf ook een *child* van de TabPane en van het *Node*-type. Dit betekent dat de TabPaneSkin het *parsen* van de onderliggende *child-Nodes* toch mogelijk maakt.

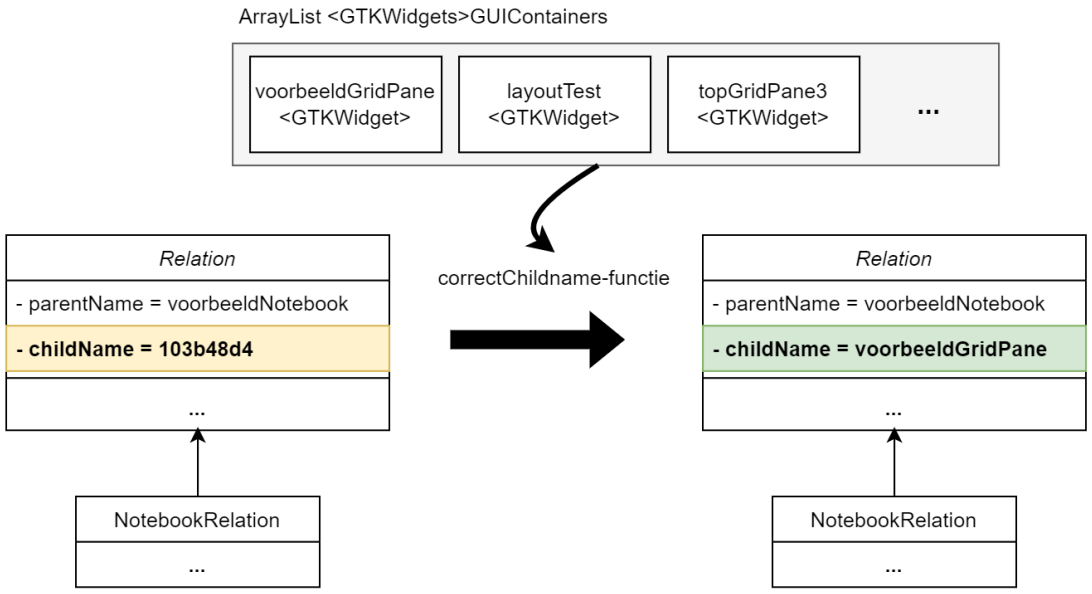
Dat de Tab niet van het *Node*-Type is, is in principe geen probleem voor het aanmaken van de *child-Nodes*. Toch zorgt dit voor problemen met het aanmaken van de relatie. Dit is omdat de hiërarchie tussen de TabPane en de *child*-widgets moeilijk te achterhalen is aangezien de *child-Nodes* zich in de verschillende lagen van de TabPaneSkin bevinden. Dit probleem is schematisch weergegeven in figuur 55 (b).

Een oplossing hiervoor is om voor de relaties puur naar de Tab te focussen aangezien de *child-Nodes* en Tab-Label direct vindbaar zijn. Het probleem hierbij is dat de namen van de zelfgemaakte *child*-GTK-widgets nodig zijn om een relatie te maken.

Een schematische weergave voor de oplossing is weergegeven in figuur 56. Om dit op te lossen wordt tijdens het *parsen* en aanmaken van een NotebookRelation enkel de hashcode van de *child-Node* opgeslagen aangezien de zelfgemaakte GTKWidget-versie van die *child-Node* nog niet is aangemaakt. Na het *parsen* wordt vervolgens in een laatste stap voor iedere NotebookRelation die de hashcode van de *child-Node* bevat, de overeenkomstige GTKWidget-*child* naam ingesteld voordat de Haskell-code wordt gegenereerd. Dit wordt gedaan m.b.v. de *correctChildname*-functie.



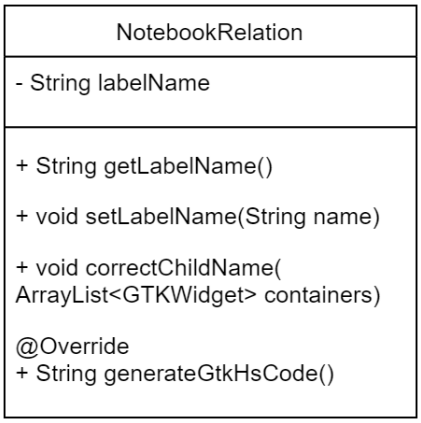
Figuur 55: De JavaFX TabPane van het voorbeeld vanuit een parse-standpunt voor het aanmaken van de GTKWidgets (A) en vanuit een relatie-standpunt waar de relatie tussen de GTKWidgets worden gemaakt(B)



Figuur 56: Schematische weergave hoe de NotebookRelations worden aangepakt na het parsen vooraleer de Haskell-code ervan wordt gegenereerd.

Ten slotte wordt uiteindelijk de NotebookRelation-klasse zelf besproken. Figuur 57 geeft hierbij de zelfgemaakt NotebookRelation-klasse weer. De belangrijkste eigenschappen van deze klasse zijn:

- labelName: de declaratie-naam van de zelfgemaakte GTKWidget-Label die bij de Notebook hoort,
- correctChildName-functie: de functie die op het einde wordt opgeroepen om de correcte *childName*-attributen in te stellen o.b.v. de reeds ingestelde *child*-hashcodes.



Figuur 57: Custom NotebookRelation-klasse gebruikt voor een Notebook

4.11 Alles samengevat

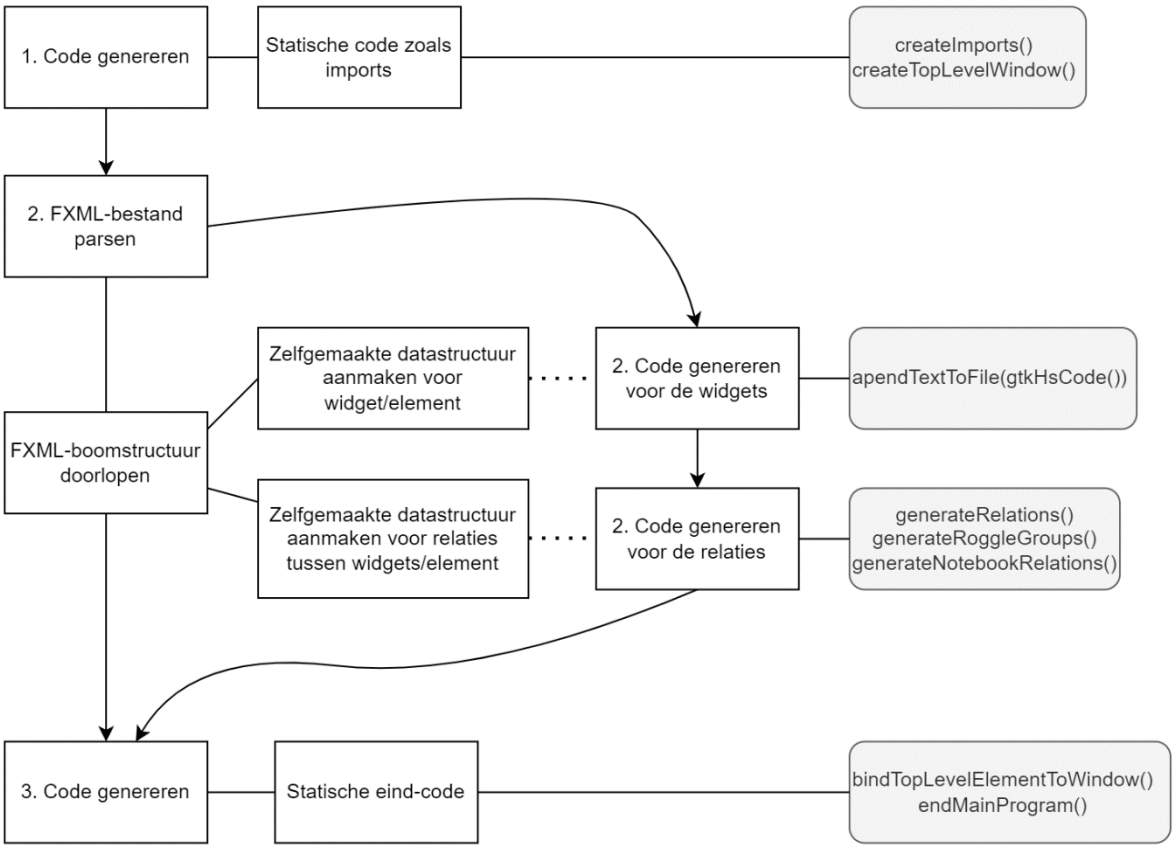
Tot slot wordt het stappenplan van de codegenerator nog eens samengevat. Figuur 58 geeft het stappenplan van de werkende codegenerator visueel weer.

Om te beginnen genereert de codegenerator een Haskell-bestand waarin alle code komt te staan.

Vervolgens genereert de codegenerator ook statische code. Onder statische code vallen imports, het begin van de *main*-functie en het creëren van een venster die de applicatie bevat.

Daarna volgt de *parse*-stap. Deze stap doorloopt het FXML-bestand en op basis daarvan worden twee zaken uitgevoerd. Eerst maakt de dump-functie voor elke widget in JavaFX een gelijkaardig datastructuur aan in Java die de widget in Haskell voorstelt. Bij het aanmaken van de widget, vult de dump-functie ook al direct het bestaande Haskell-bestand aan met de declaratie-code van de widget. Daarnaast maakt de dump-functie ook al relaties tussen de widgets aan. Het Haskell-bestand wordt in deze stap nog niet aangevuld met de code voor de relaties omdat eerst alle widgets aangemaakt moeten worden. Enkel de relaties zelf worden dus intern, in de dump-functie, in de codegenerator zelf aangemaakt.

Tot slot volgt nog de afsluitcode. Deze code bestaat uit drie delen. Het eerste deel is de code voor de relaties die zijn aangemaakt in de dump-functie. Hierbij behoren ordinaire relaties tussen widgets onderling, maar ook bijvoorbeeld het binden van RadioButtons aan een ToggleGroup. Het tweede deel is het binden van het hoogste element aan de *window* dat is aangemaakt in de eerste stap. Het derde en laatste deel zorgt ervoor dat de *main*-functie wordt afgesloten en dat de applicatie opgestart kan worden.



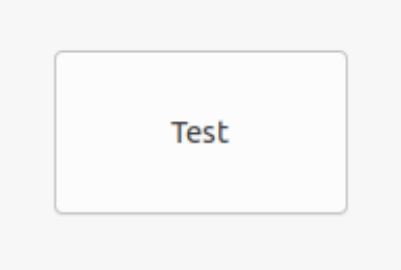
Figuur 58: Schematische weergave van het stappenplan van de werkende codegenerator

5 Resultaten en discussie

5.1 Deelresultaten Widgets

5.1.1 Button

Figuur 59 toont een Button in Haskell (gi-gtk) die de codegenerator heeft gemaakt.



Figuur 59: Button in Haskell met de gi-gtk bibliotheek

Een Button-widget in gi-gtk is nagenoeg exact hetzelfde als een Button in JavaFX. *Label*, *breedte* en *hoogte* zijn attributen die overeenkomen in beide talen terwijl er geen attributen zijn die de ene taal wel heeft maar de andere taal niet. In figuur 59 heeft *label* de waarde “Test”.

5.1.2 Label

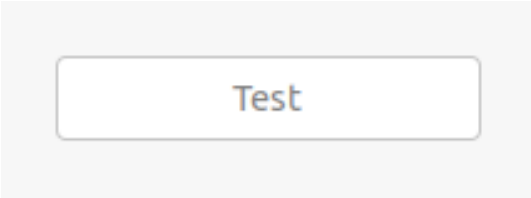
Figuur 60 toont een Label in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

Figuur 60: Label in Haskell met de gi-gtk bibliotheek

Een Label-widget in gi-gtk is, net zoals de Button, quasi exact hetzelfde als een Label in JavaFX. *Text* is het enig attribuut dat Label heeft en deze komt ook overeen in zowel gi-gtk als in JavaFX. In figuur 60 heeft *text* de waarde “Test”. Een Label heeft in JavaFX echter wel een eenvoudige manier voor tekstmodificaties zoals een *font*, *size* of *textfill* (kleur). Dit is in gi-gtk moeizaam om te implementeren.

5.1.3 Entry

Figuur 61 toont een Entry in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

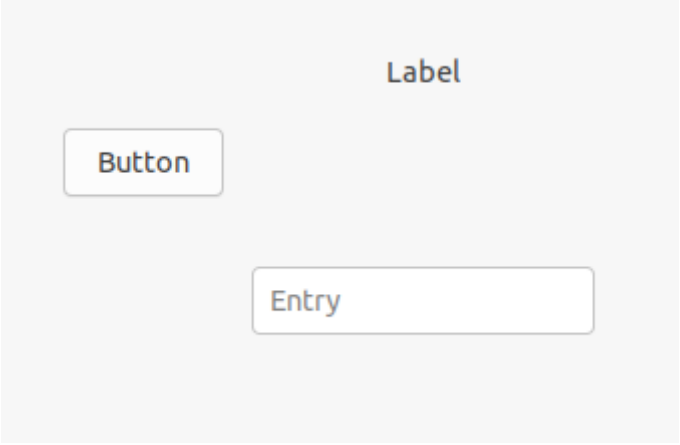


Figuur 61: Entry in Haskell met de gi-gtk bibliotheek

Een Entry-widget in gi-gtk komt vrijwel precies overeen met een TextField in JavaFX. *Text*, *breedte*, *hoogte*, *alignment* en *placeholder* (*Prompt Tex*t in JavaFX) zijn attributen die in beide talen voorkomen. Figuur 61 heeft geen waarde voor *text*, maar wel voor *placeholder* (= "Test"). *Alignment* heeft waarde 0.5 in gi-gtk wat overeenkomt met CENTER in JavaFX.

5.1.4 Layout

Figuur 62 toont een Layout met een Label, Button en Entry in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

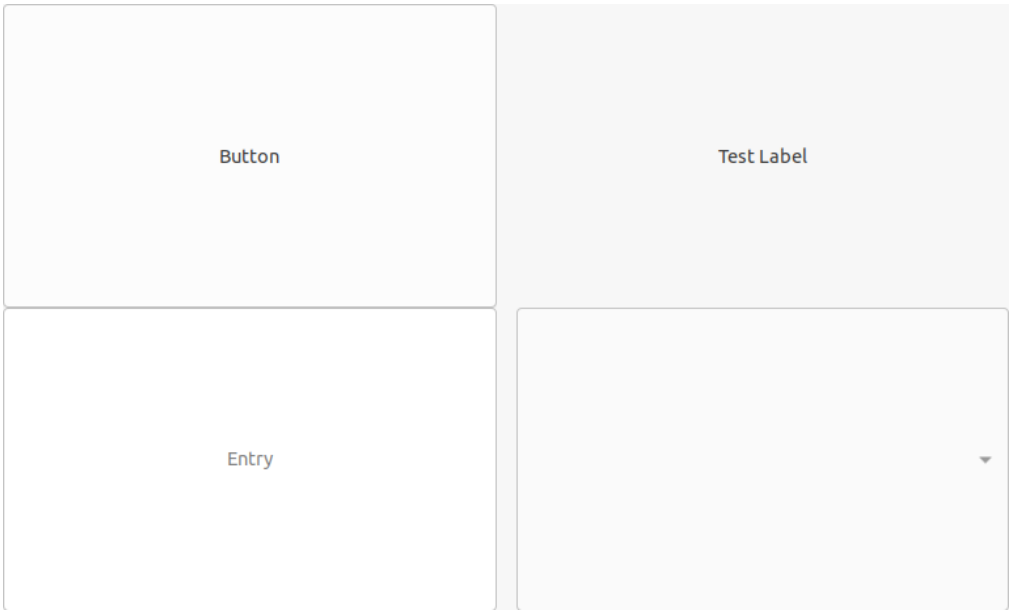


Figuur 62: Layout met een Label, Button en Entry in Haskell met de gi-gtk bibliotheek

Een Layout-widget in gi-gtk komt overeen met een AnchorPane in JavaFX. Layout zelf heeft geen enkel attribuut en dat is hetzelfde voor AnchorPane. Een Layout dient voornamelijk als container waar het mogelijk is om andere widgets dan vrij te plaatsten met een X- en Y-coördinaat. Figuur 62 toont dit principe, waar drie widgets (Label, Button en Entry) vrij zijn geplaatst.

5.1.5 Grid

Figuur 63 toont een Grid met een Button (linksboven), Label (rechtsboven), Entry (links beneden) en ComboBoxText (rechts beneden) in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

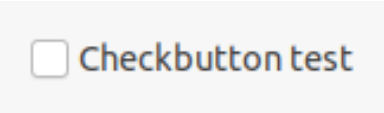


Figuur 63: Grid met een Button, Label, Entry en ComboBoxText in Haskell met de gi-gtk bibliotheek

Een Grid-widget in gi-gtk komt overeen met een GridPane in JavaFX. *ColumnSpacing* en *rowSpacing*, respectievelijk *Hgap* en *Vgap* in JavaFX, zijn attributen die in zowel gi-gtk als in JavaFX voorkomen. Figuur 63 heeft een columnSpacing van 15 pixels en een rowSpacing van 0 pixels. Verder heeft JavaFX nog attributen die de *alignment* van het Child kunnen regelen (*halignment*, *valignment*) terwijl gi-gtk dit niet heeft. Daarnaast worden in gi-gtk de kolommen en rijen homogeen gemaakt. Dit betekent dat elke cel in de Grid dezelfde breedte en hoogte heeft.

5.1.6 CheckButton

Figuur 64 toont een CheckButton in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

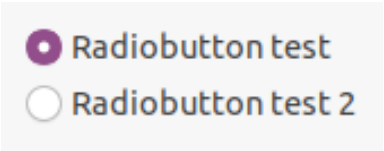


Figuur 64: CheckButton in Haskell met de gi-gtk bibliotheek

Een CheckButton-widget in gi-gtk komt overeen met een CheckBox in JavaFX. *Text* en *active* (*selected* in JavaFX) zijn attributen die overeenkomen in beide talen. Echter is het *active*-attribuut niet geïmplementeerd door de codegenerator. Dit attribuut bepaalt de status van de CheckButton en is dus vooral handig als er gewerkt wordt met *callback*-functies waarin deze status moet achterhaald worden. In figuur 64 staat *active* op false en heeft *text* de waarde “Checkbutton test”.

5.1.7 RadioButton

Figuur 65 toont twee RadioButtons die samen één ToggleGroup vormen in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

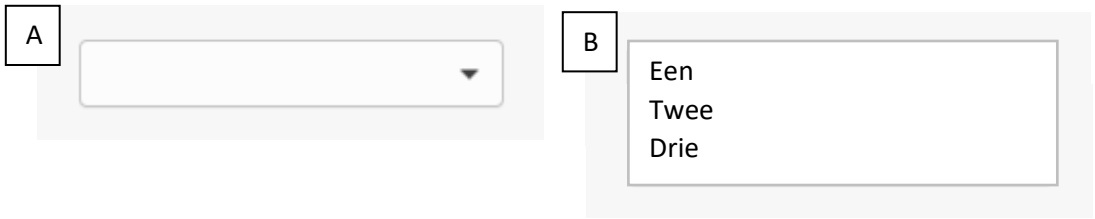


Figuur 65: Twee RadioButtons die samen één ToggleGroup vormen in Haskell met de gi-gtk bibliotheek

Een RadioButton-widget in gi-gtk komt overeen met een RadioButton in JavaFX. Een RadioButton is quasi identiek als een CheckButton, maar de vierkantige *box* die kan worden aangevinkt bij een CheckButton is nu vervangen door een cirkelvormige *box*. Daarnaast kan een RadioButton ook bij een ToggleGroup behoren. Een ToggleGroup is een groep van RadioButtons waar maximum één RadioButton kan aangevinkt zijn. Op figuur 65 betekent dit dat ofwel "Radiobutton test" ofwel "Radiobutton test 2" aangevinkt kan zijn.

5.1.8 ComboBoxText

Figuur 66 toont een ComboBoxText in Haskell (gi-gtk) die de codegenerator heeft gemaakt.



Figuur 66: ComboBoxText voor (A) en na (B) er een keuze kan gemaakt worden in Haskell met de gi-gtk bibliotheek

Een ComboBoxText-widget in gi-gtk komt overeen met een ComboBox in JavaFX. *Entry* (*editable* in JavaFX), *items*, *width* en *height* zijn attributen die beide talen beschikken. In gi-gtk is er ook een methode *getActiveText* die de aangeduide waarde van de ComboBoxText teruggeeft. Deze methode is vooral handig als er *callback*-functies worden opgeroepen. Verder heeft ComboBoxText geen attribuut dat tekst toont dat dient als een hint (zoals *placeholder* bij Entry bijvoorbeeld). Dit principe bestaat in JavaFX wel onder het attribuut *prompt text*. Figuur 66 heeft *entry* op *false* staan (typen in de ComboBoxText is niet mogelijk). Daarnaast is het *items*-attribuut een lijst van Strings met als waarde: Een, Twee en Drie.

5.1.9 LinkButton

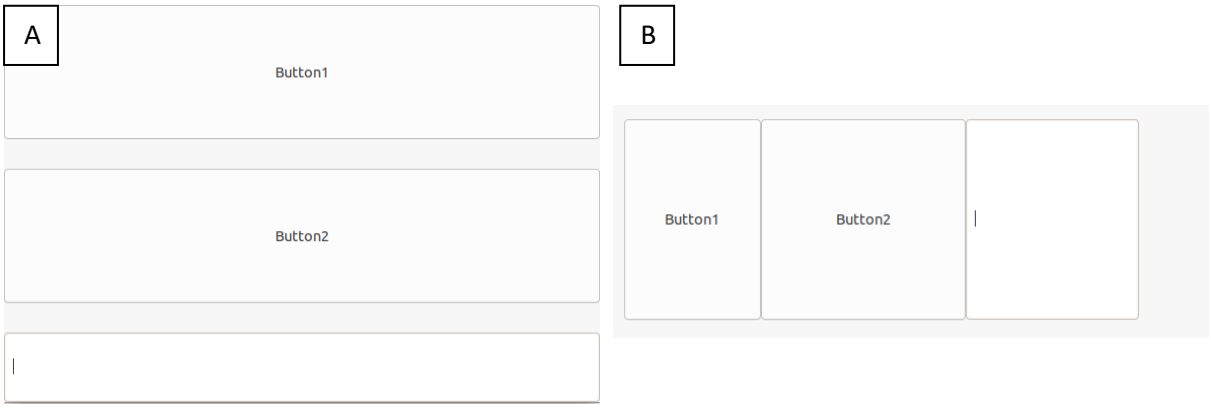
Figuur 67 toont een LinkButton in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

Figuur 67: LinkButton in Haskell met de gi-gtk bibliotheek

Een LinkButton-widget in gi-gtk komt overeen met een Hyperlink in JavaFX. *Text* en *visited* zijn attributen die voorkomen in beide talen. Een LinkButton is eigenlijk een uitbreiding van een Label door het attribuut *visited*. Dit attribuut bepaalt of de tekst wordt weergegeven alsof de link al is bezocht of niet. Verder is het mogelijk om in gi-gtk eenvoudig een link mee te geven aan de LinkButton. Dit wordt gedaan bij het aanmaken van de LinkButton of door de functie *setUri*. In JavaFX is het echter moeilijker om een link te koppelen aan een Hyperlink. De Hyperlink zelf heeft geen attribuut die dit regelt en daarom moet dit gebeuren aan de hand van een *setOnAction* methode. Dit werkt alleen met *callback*-functies en aangezien de codegenerator dit (nog) niet afhandelt, is er dus (nog) geen manier om een link te geven aan een LinkButton.

5.1.10 HBox en VBox

Figuur 68 toont een VBox (a) en een HBox (b) met twee Buttons en één Entry in Haskell (gi-gtk) die de codegenerator heeft gemaakt.

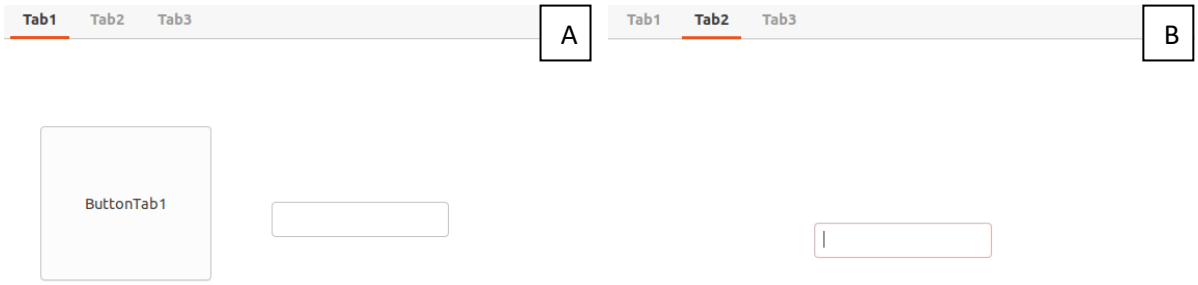


Figuur 68: VBox (A) en HBox (B) met twee Buttons en één Entry in Haskell met de gi-gtk bibliotheek

HBox en VBox-widgets in gi-gtk komen overeen met HBox en VBox in JavaFX. HBox en VBox worden in gi-gtk aangemaakt door een Box-widget. Box heeft *orientation* en *spacing* als attributen. *Orientation* bepaalt als het een HBox (OrientationHorizontal) of VBox (OrientationVertical) is. Verder hebben HBox en VBox, net zoals GridPane, een *alignment*-eigenschap in JavaFX dat er in gi-gtk niet is. Figuur 68 toont dat de VBox (a) wel *spacing* heeft, maar de HBox niet (b).

5.1.11 Notebook

Figuur 69 toont een Notebook met drie Tabs in Haskell (gi-gtk) die de codegenerator heeft gemaakt. Tab 1 bestaat uit een Layout met een Button en een Entry en Tab 2 bestaat uit een Layout met enkel een Entry.

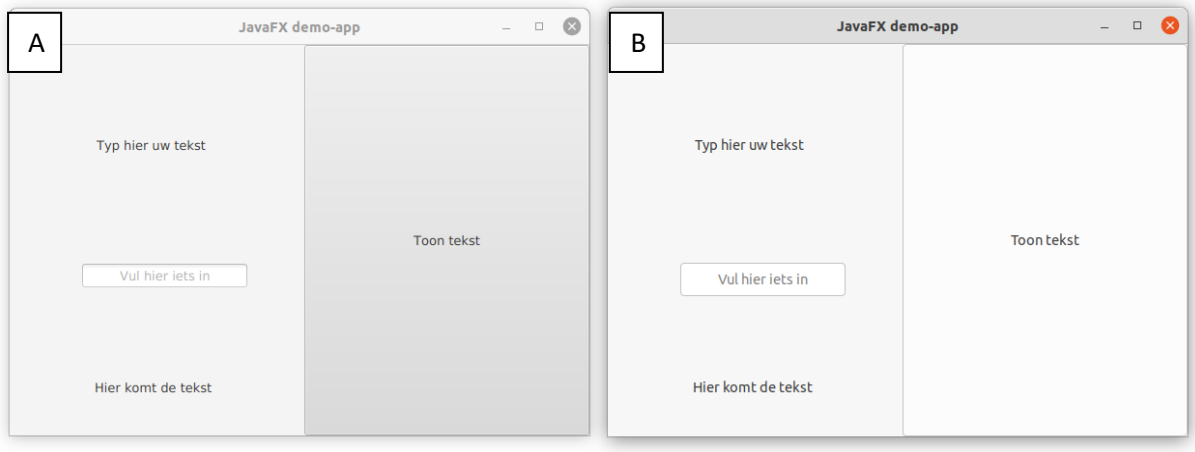


Figuur 69: Notebook met drie Tabs waarin Tab 1 bestaat uit een Layout met een Button en een Entry (A) en Tab 2 uit een Layout met enkel een Entry (B) in Haskell met de gi-gtk bibliotheek

Een Notebook-widget in gi-gtk komt overeen met TabPane in JavaFX. *NoteBookLabels*, *width* en *height* zijn attributen dat voorkomen in beide talen. *NoteBookLabels* is een lijst van Labels dat zorgt voor de naam van de Tabs ("Tab1", "Tab2" en "Tab3" in figuur 69). In elke Tab van de Notebook hoort dan een container (Layout, Grid, …) met daarin widgets naar keuze.

5.2 Resultaat volledige demo GUI-applicatie

Figuur 70 toont de demo GUI-applicatie in JavaFX (a) en in Haskell met behulp van de gi-gtk bibliotheek (b).



Figuur 70: Demo GUI-applicatie in JavaFX (A) en in Haskell met gi-gtk (B)

De demo-GUI-applicatie bestaat uit een Grid (GridPane) van 2x1 (2 kolommen, 1 rij). De linker kolom van de Grid bestaat uit een Layout (AnchorPane) met daarin twee Labels ("Typ hier uw tekst" en "Hier komt de tekst") en één Entry (TextField met "Vul hier iets in"). De rechter kolom bestaat uit een Layout met daarin enkel een Button ("Toon tekst").

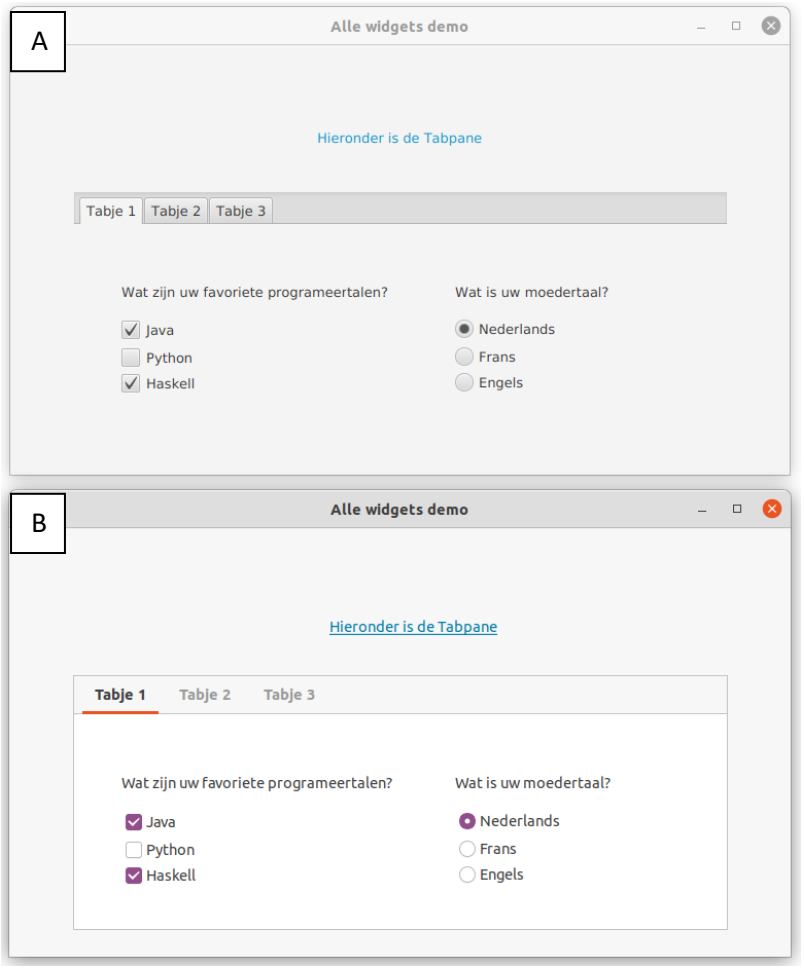
In het algemeen ziet de demo-GUI-applicatie in Haskell er hetzelfde uit als deze in JavaFX. De breedte en hoogte van de applicatie zijn identiek en de positie van de widgets komen ook exact overeen. De

widgets zelf komen correct overeen, al is de hoogte van de Entry groter in gi-gtk dan de hoogte van de TextField in JavaFX. Dit valt te verklaren doordat de Entry een minimumhoogte heeft waaraan voldaan moet worden. De hoogte van het TextField is kleiner dan deze minimumhoogte waardoor het verschil ontstaat.

De Haskell-code van de demo-GUI-applicatie is terug te vinden in bijlage H.

5.3 Uitgebreide demo met alle mogelijke widgets in één GUI-applicatie

Om te controleren als alle widgets goed omgezet worden door de codegenerator is er een GUI-applicatie gemaakt die alle widgets bevat. Figuur 71 toont de eerste Tab van deze applicatie in JavaFX (a) en in Haskell met behulp van de gi-gtk bibliotheek (b). Figuur 72 toont de tweede Tab van deze applicatie in JavaFX (a) en in Haskell met behulp van de gi-gtk bibliotheek (b). Figuur 73 toont de derde Tab van deze applicatie in JavaFX (a) en in Haskell met behulp van de gi-gtk bibliotheek (b). De GUI-applicatie zelf bestaat uit een Layout met daarin een LinkButton (“Hieronder is de Tabpane”) en een Notebook met drie Tabs (“Tabje 1”, “Tabje 2” en “Tabje 3”).



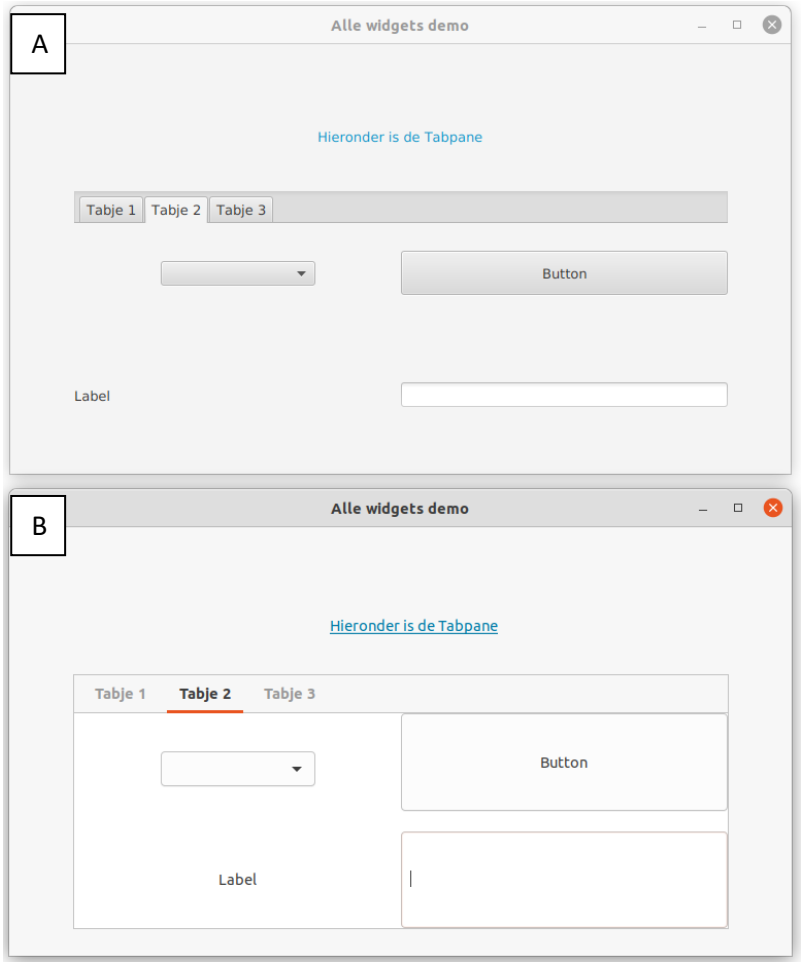
Figuur 71: Tabje 1 van de GUI-applicatie met alle widgets in JavaFX (A) en in Haskell met gi-gtk (B)

Figuur 71 toont Tabje 1 van de GUI-applicatie met alle widgets. Deze bestaat uit een Layout (AnchorPane) met daarin twee Labels (“Wat zijn uw favoriete programmeertalen?” En “Wat is uw moedertaal?”), drie CheckButtons (“Java”, “Python” en “Haskell”) en drie RadioButtons

("Nederlands", "Frans" en "Engels"). De drie RadioButtons behoren ook nog samen tot een ToggleGroup.

Visueel ziet Tabje 1 er ietwat anders uit in gi-gtk dan in JavaFX, maar de positie van de widgets is exact hetzelfde. De widgets zelf functioneren ook hetzelfde in gi-gtk als in JavaFX en daarnaast zijn de breedte en hoogte ook identiek.

Figuur 72 toont Tabje 2 van de GUI-applicatie met alle widgets. Deze bestaat uit een Grid (GridPane) van 2x2. Op positie 1x1 is er een Layout met daarin een ComboBoxText met drie opties waarbij de items niet zichtbaar zijn op de figuur. Ook bevindt er zich op positie 2x1 een Button en op positie 1x2 een Label. Tot slot staat er op positie 2x2 een Entry (TextField).

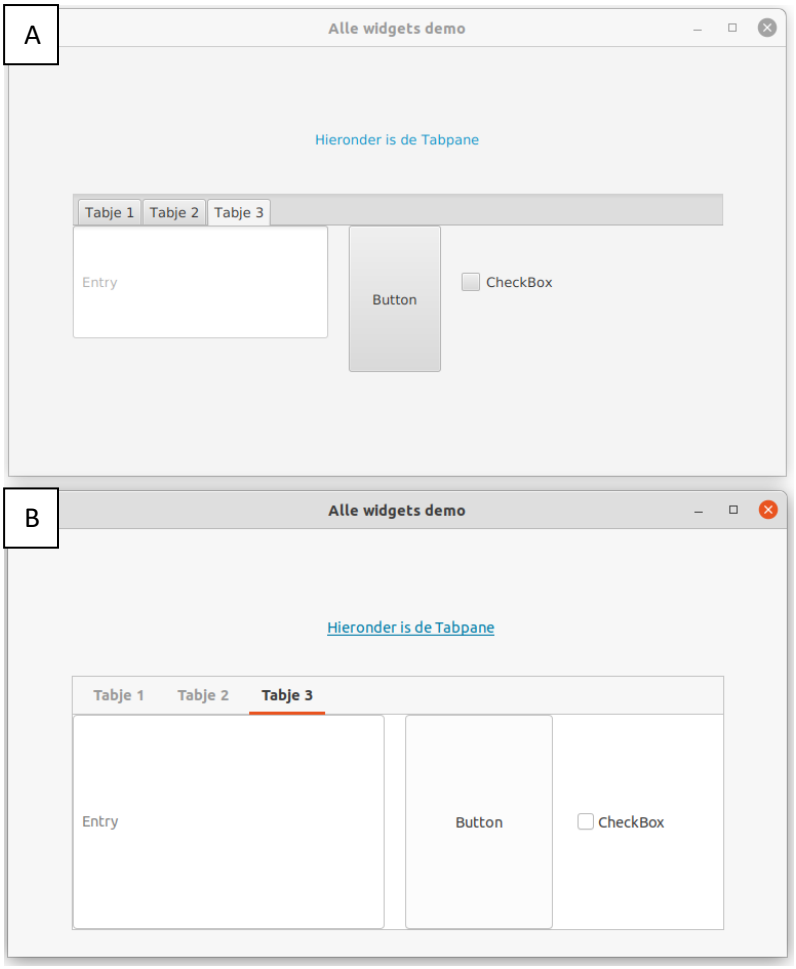


Figuur 72: Tabje 2 van de GUI-applicatie met alle widgets in JavaFX (A) en in Haskell met gi-gtk (B)

Visueel zijn er in deze Tab opnieuw enkele verschillen. Echter is het inhoudelijk ook enigszins anders dan Tabje 1. De positie van de widgets in gi-gtk komen bijna exact overeen met deze in JavaFX. Enkel de Label heeft een andere positie. Dit valt te verklaren omdat een Grid in gi-gtk zijn *child* in het centrum van zijn cel positioneert. Daarnaast komt de hoogte van de Button en van de Entry ook niet exact overeen. In gi-gtk is de hoogte (en ook de breedte) gelijk aan de volledig cel van de Grid. Een *child* van een Grid wordt dus niet alleen in het midden van zijn cel geplaatst, maar neemt ook de volledige cel in. De reden waarom de ComboBoxText wel exact overeenkomt in gi-gtk is doordat deze nog eerst in een Layout is geplaatst. De Layout neemt het volledige vakje van de Grid in, maar in de Layout zelf is alles vrij te plaatsen. De widgets zelf functioneren wel exact hetzelfde in gi-gtk als in

JavaFX. Tot slot is er een kleine *gap* te zien tussen de rijen van de Grid. Dit valt te verklaren doordat het attribuut *rowSpacing* een waarde heeft gekregen.

Figuur 73 toont Tabje 3 van de GUI-applicatie met alle widgets. Dit Tabje bestaat uit een HBox met daarin een Entry (TextField), een Button en een CheckButton (CheckBox).



Figuur 73: Tabje 3 van de GUI-applicatie met alle widgets in JavaFX (A) en in Haskell met gi-gtk (B)

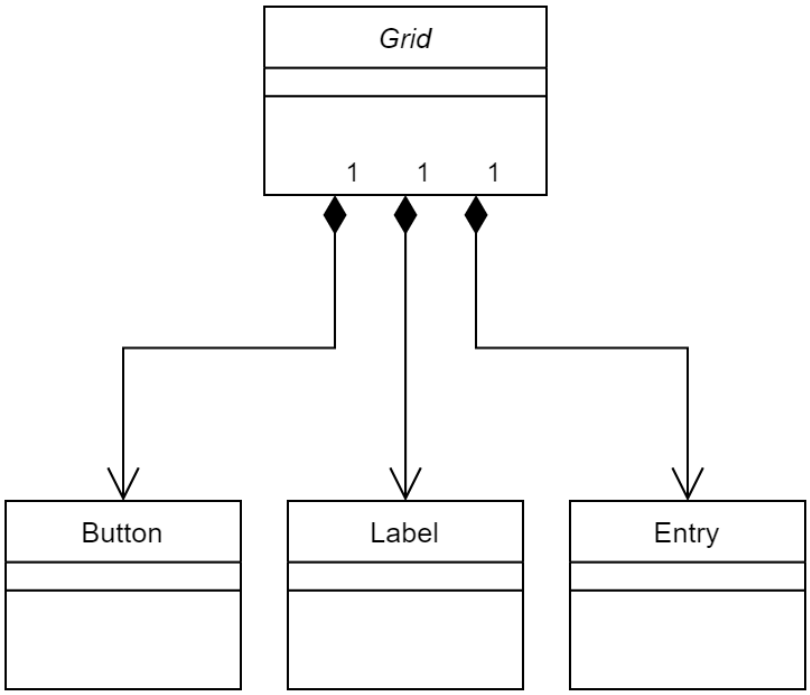
Visueel ziet deze Tab er in gi-gtk ook weer anders uit dan in JavaFX. De positie van de widgets is niet overal identiek. De Button en de CheckButton zijn enigszins naar de rechterkant geplaats in gi-gtk. Dit komt vooral doordat de breedte van de Entry te groot is. Dit valt te verklaren omdat de HBox (en ook de VBox) zijn widgets verdeelt over de hele breedte (of hoogte in het geval van een VBox). De verhoudingen van de breedtes in gi-gtk lijken wel hetzelfde als deze in JavaFX. De hoogte van de widgets zijn echter ook niet hetzelfde (in een VBox gebeurt dit met de breedte). In gi-gtk nemen de widgets de volledige hoogte in van de HBox, terwijl er in JavaFX gespeeld kan worden met de hoogtes. Een HBox (en ook een VBox) in gi-gtk zorgt er dus altijd voor dat zijn widgets de volledige Box in neemt. Het verhinderen is mogelijk door een Layout te gebruiken in de Box zelf. De widgets zelf functioneren opnieuw exact hetzelfde in gi-gtk als in JavaFX. Tot slot is er nog een kleine opening tussen de widgets. Deze opening komt door het *spacing* attribuut van de Box.

De Haskell-code van de GUI-applicatie met alle widgets is terug te vinden in bijlage I.

5.4 Waarom relaties en widgets gescheiden houden

Doorheen hoofdstuk 4 werd telkens vermeld dat de relaties en *Nodes*/widgets aparte datastructuren kregen. Een kritische blik die hierop geworpen kan worden is of het ook mogelijk is om de relaties hiërarchisch in de widgets te verwerken.

Figuur 74 toont het principe van de relaties en GUI-elementen samenvoegen op een schematische manier. Hierbij worden de *child-Nodes* direct in het container-element geplaatst om het vervolgens mogelijk te maken de nodige relaties te genereren. Hoewel dit principe op eerste zicht een overzichtelijker en compactere oplossing geeft, bevat het enkele gebreken vergeleken met de bestaande oplossing in de codegenerator.

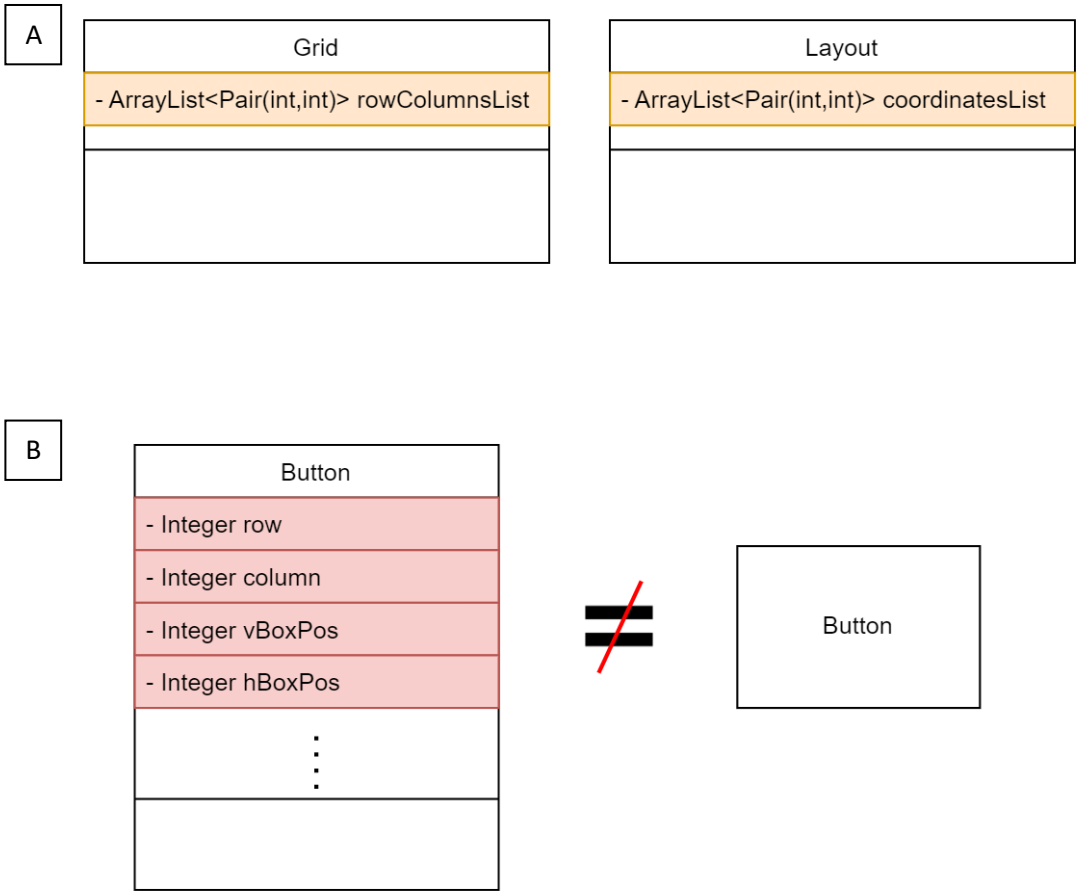


Figuur 74: Voorstelling van verschillende zelfgemaakte Child-Nodes direct in de zelfgemaakte Grid-Node

Ten eerste zouden de datastructuren van de widgets, en vooral de containers niet meer puur zijn. Figuur 75 toont dit schematisch weer met een voorbeeld. De rij- en kolom-eigenschappen, x- en y-coördinaten en diverse eigenschappen, die tot een relatie behoren, zouden ergens moeten bijgehouden worden door bijvoorbeeld de *child-Node*. Dit zorgt ervoor dat de datastructuren van de widgets/*Nodes* niet meer puur zijn aangezien ze eigenschappen bevatten die ze niet direct bezitten. Indien de eigenschappen van een relatie in de bovenliggende container-widget wordt opgeslagen, moet hierbij ook rekening worden gehouden met een complexere datastructuur.

Ten tweede zou het parsen van de widgets onduidelijker zijn en meer werk vereisen. Figuur 76 (b) geeft dit probleem schematisch weer. Dit is omdat naast de eigenschappen van de *Node* zelf ook de eigenschappen van de relaties, zoals de rij en kolom, verkregen moeten worden. Voor het verkrijgen van deze eigenschappen voor een relatie moet soms gekeken worden naar de *Parent-Node* en soms naar de (*child-*)*Node* zelf. Dit principe naar code vertaald zoals in figuur 76 (a) betekent dat de code onoverzichtelijk is en moeilijker optimaliseerbaar vergeleken met de relatie-eigenschappen en widget-eigenschappen apart te verkrijgen. Een eventuele leesbare, maar inefficiënte, oplossing voor dit probleem zou zijn om twee aparte dump-functies te maken waarbij één functie wordt gebruikt voor

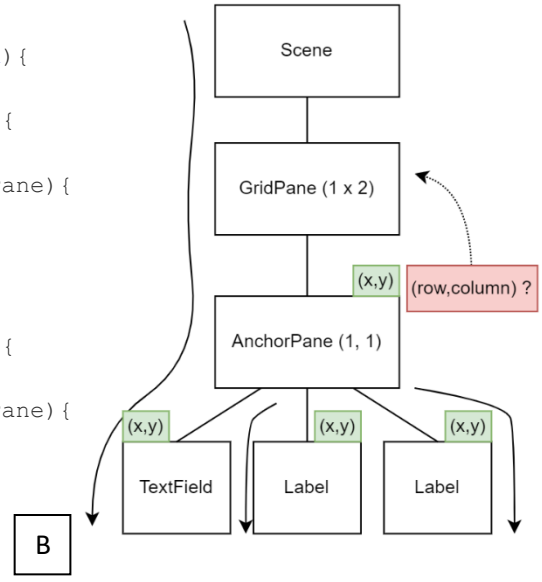
de *Nodes* te *parsen* en de andere functie voor de relatie-eigenschappen te parsen en die in de reeds zelfgemaakte datastructuren toe te voegen.



Figuur 76: Meer complexere en uitgebreide datastructuur indien relatie-eigenschappen in de container-Widget worden bijgehouden (A) en het probleem waarbij de Widget niet meer puur is indien de relatie-eigenschappen worden opgeslagen in de child-Widgets zelf (B)

A

```
1  //PARSE NODES/WIDGETS
2  if(n instanceof javafx.scene.control.Button){
3      //PARSE RELATIE EIGENSCHAPPEN
4      if(n.getParent() instanceof AnchorPane){
5      ...
6      }else if(n.getParent() instanceof GridPane){
7      ...
8      }
9  }else if (n instanceof AnchorPane){
10     //PARSE RELATIE EIGENSCHAPPEN
11     if(n.getParent() instanceof AnchorPane){
12     ...
13     }else if(n.getParent() instanceof GridPane){
14     ...
15     }
16 }
```



Figuur 75: Extra overhead bij het parsen in een code-snippet (A) en schematische visualisatie waarbij bepaalde eigenschappen zoals de rij en kolom bij de Parent-Node moet geraadpleegd worden (B)

5.5 Waarom toch doorgaan met de codegenerator en SceneBuilder

Eén van de aanleidingen voor het creëren van een codegenerator voor deze masterproef is omdat Haskell op eerste zicht geen kant en klare GUI-oplossingen aanbood. Hoewel Glade deze rol voor een groot deel vervult, zijn er toch enkele redenen waarom de codegenerator en het gebruik van SceneBuilder hun nut kunnen behouden voor toekomstig werk.

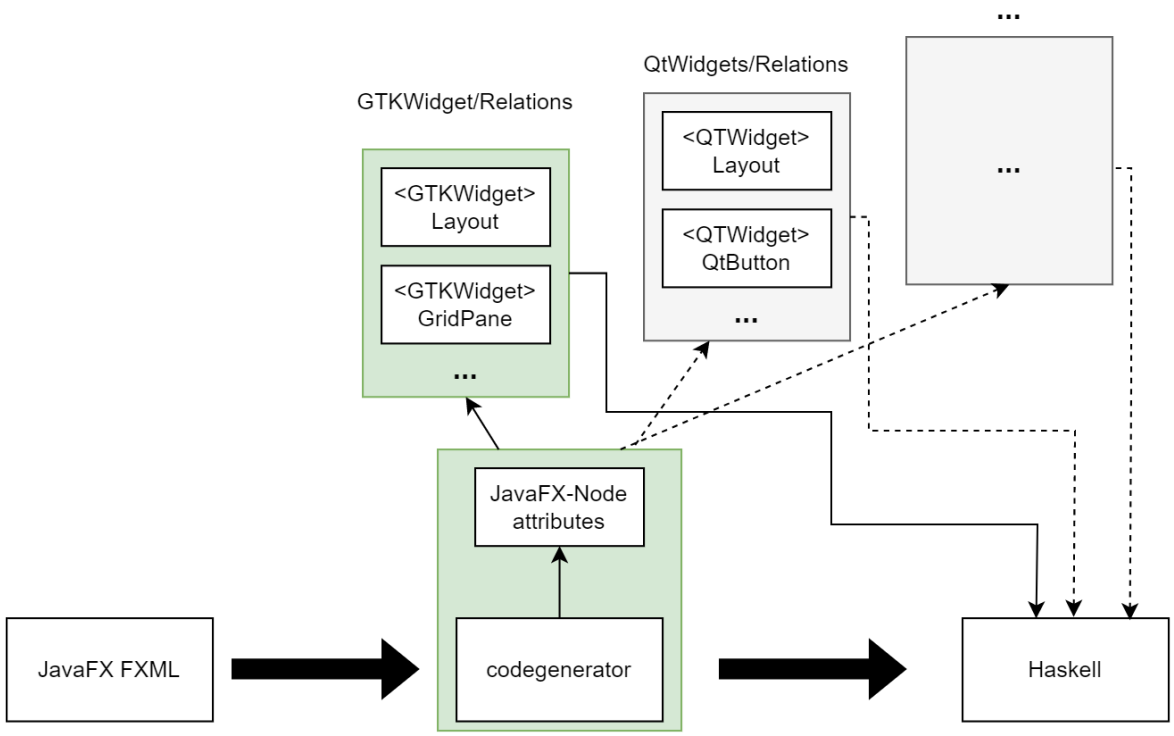
Ten eerste is SceneBuilder nog steeds eenvoudiger te gebruiken dan Glade. Het *drag-and-drop*-principe om bepaalde elementen te slepen, is voor een overgroot deel van personen en vooral leerlingen intuïtiever dan het klikken en daarna pas aanpassen van een element. Ook is de exacte workflow, de GUI-elementen zelf uit JavaFX en de interface van SceneBuilder al gekend bij de meeste studenten aangezien het al enkele jaren met succes wordt gebruikt. De combinatie van SceneBuilder en de codegenerator maakt het dus mogelijk voor een vlotte introductie met GUIs in Haskell.

Ten tweede heeft SceneBuilder het voordeel dat het voor een groter publiek ter beschikking gesteld wordt. Dit is omdat SceneBuilder multi-platform is en direct op de meeste besturingssystemen uitgevoerd kan worden.

Daarnaast is het principe van GUIs in Java door meer mensen gekend doordat Java een *cross-platform* is waarbij de GUI-ontwikkeling al verder is ontwikkeld. De populariteit van JavaFX (en dus SceneBuilder) is door de grote featureset de laatste jaren dermate toegenomen dat veel projecten de standaard Java GUI-API Swing vervangen door JavaFX [50]. De gelijkenissen om met een GUI-tool in Java zoals SceneBuilder te werken, zorgen er dus voor dat toekomstige ontwikkelaars zich sneller thuis voelen met de ontwikkeling van Haskell-GUIs in combinatie met de codegenerator en SceneBuilder.

Bovendien heeft de codegenerator een robuuste parsing-methode die zonder grote problemen de JavaFX-*Nodes* en relaties ervan herkent. Dit zorgt ervoor dat de codegenerator later *geport* kan worden naar andere Haskell GUI-bibliotheken of zelfs GUI-bibliotheken in andere programmeertalen. De grootste aanpassingen die dan moeten worden gemaakt zijn de zelfgemaakte datastructuren aanmaken in de bibliotheek naar keuze en het selecteren van de gepaste eigenschappen die overeenkomen met de corresponderende JavaFX-elementen. Dit principe is visueel weergegeven in figuur 77.

Ten slotte is ook besloten verder te gaan met de codegenerator aangezien voorafgaand de masterproef, het concept van Glade voor Haskell-GUIs te ontwikkelen nog niet bekend was. Het was immers ook het doel om kennis te verwerven over de wereld van GUIs in Haskell. Pas halverwege de masterproef werd door onderzoek bekend dat Glade een manier was om Haskell GUIs te ontwikkelen. Tegen deze tijd waren de eerste stappen in de ontwikkeling van de codegenerator al genomen.

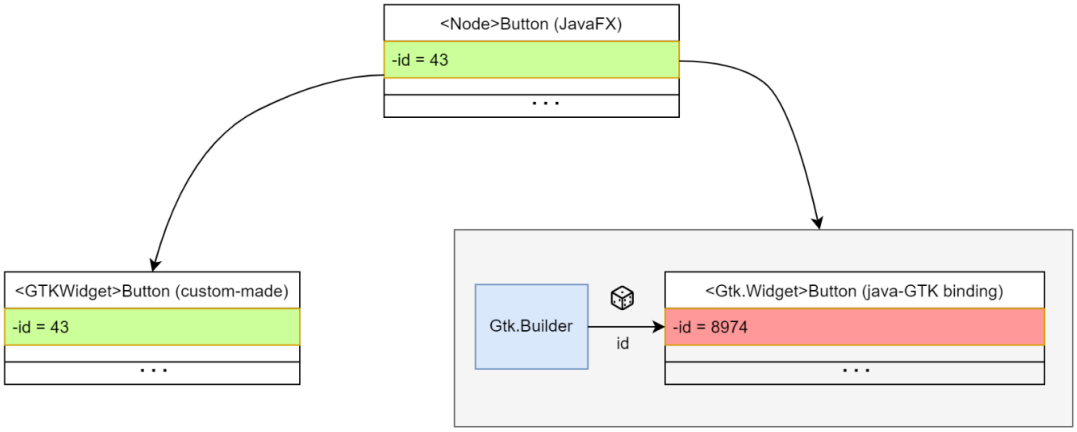


Figuur 77: Schematische weergave van een eventuele latere uitbreiding van de codegenerator naar een andere bibliotheek naar keuze

5.6 Zelfgemaakte klassen voor de GTK-Widgets

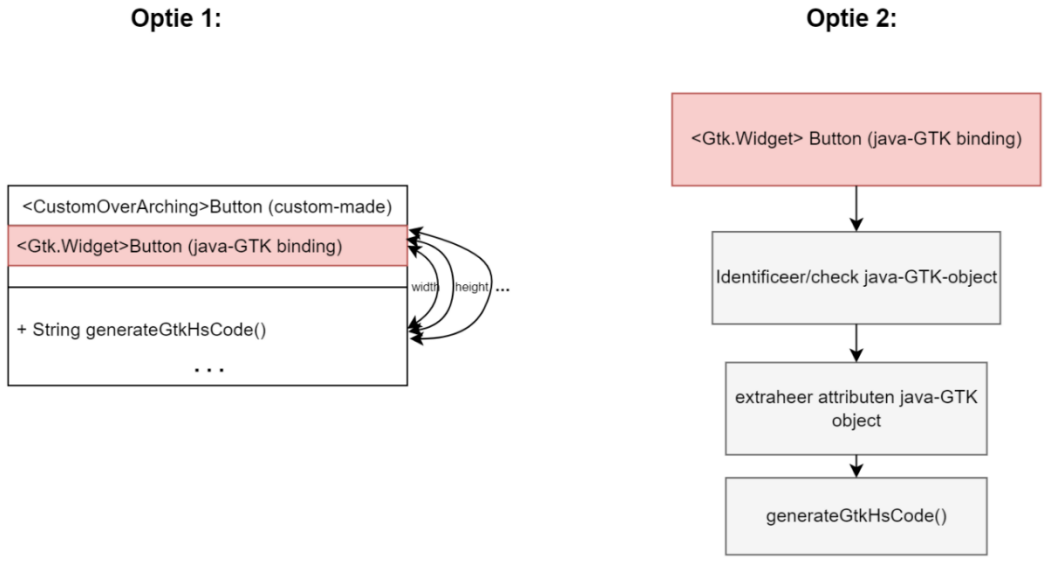
Zelfgemaakte GTKWidget-instanties zijn telkens aangemaakt bij het parsen om de GTK-Widget in Java voor te stellen. Een kritische blik die hierop geworpen kan worden is waarom er geen gebruik is gemaakt van GTK-Java bindings om de *geparsede* widgets voor te stellen en die vervolgens 1 op 1 te mappen op de widgets van gi-gtk. Dit resulteert in minder zelfgemaakte code, minder onderhoudswerk bij nieuwe objecten en kan semiautomatisch worden gedaan via reflectie.

De reden waarom er toch is gewerkt met zelfgemaakte datastructuren zijn tweedelig. Ten eerste zorgen de zelfgemaakte GTKWidget-klassen voor flexibiliteit. Hoewel het niet de bedoeling is om meer attributen/methodes toe te voegen dan nodig, is er bij zelfgemaakte klassen altijd de mogelijkheid om bepaalde attributen eenvoudiger te manipuleren. Een voorbeeld hiervan is de identificatie van een JavaFX-widget (met hashcode/id/naam) dat nu ook eenvoudig instelbaar en manipuleerbaar is in de zelfgemaakte GTKWidget-klasse. Op die manier is het nu mogelijk om de exacte JavaFX en GTKWidget-instanties te herkennen en de GTKWidget-instanties onderling te differentiëren in een latere stage (bijvoorbeeld waar de relaties worden gemaakt). Indien gebruik wordt gemaakt van de GTK-Java *bindings* (bijvoorbeeld de java-gnome *bindings*), is het mogelijk om bijvoorbeeld geen id mee te geven in de *constructor* ervan en zijn de gegenereerde id's niet hetzelfde als de id's van overeenkomende JavaFX-widgets [51]. De GTK+ documentatie zegt immers dat de id van een widget intern in een Builder wordt opgeslagen en wordt doorgespeeld naar de widget bij het aanmaken [52]. Als programmeur is er dus minder vrijheid om deze parameters te manipuleren die toch wel belangrijk zijn in de toepassing van de codegenerator. Het principe hiervan wordt nog eens visueel weergegeven in figuur 78.



Figuur 78: Schematische vergelijking van hoe compromisloos bepaalde attributen te veranderen zijn bij bestaande Java-GTK bindings vergeleken met de flexibelere zelfgemaakte implementatie van de GTKWidget-instantie

Ten tweede is het mogelijk om met de zelfgemaakte GTKWidget-klasse direct de methode op te roepen om de String te genereren voor de Haskell-code (*gtkHsCode()*). Aangezien deze methode niet is ingebouwd in een standaard Java-GTK binding, wordt de methode voor de Haskell-code in String-vorm te genereren elders afgehandeld. Een mogelijke oplossing hiervoor is om de klasse van de Java-GTK-binding in een overkoepelende klasse met de gtkHsCode()-methode te implementeren. Dit verslaat echter het beginargument en principe om geen zelfgemaakte klasse te gebruiken. Een andere oplossing is om de Haskell-code in String-vorm te genereren in elders af te handelen, na of tijdens het *parsen*. Het nadeel hierbij is dat er extra complexiteit en *overhead* moet worden toegevoegd om de Java-GTK-objecten te herkennen. Tevens is er in beide gevallen ook de extra *overhead* dat de attributen van het Java-GTK-object één per één moeten worden geëxtraheerd om het door te geven in een gtkHsCode()-methode. Dit probleem wordt nog eens visueel weergegeven in figuur 79.



Figuur 79: Schematische vergelijking van twee verschillende opties hoe de gtkHsCode()-methode afgehandeld kan worden gebruikmakend van een java-GTK binding.

5.7 Huidige gebreken codegenerator

Hoewel de codegenerator op eerste zicht robuust is en functioneert naar behoren, schiet het in bepaalde aspecten nog te kort.

Zo werd ten eerste in “4.4 Overzicht structuur van de te-genereren code” al aangehaald dat omwille van de designkeuze, de codegenerator code genereert met codevariatie A waardoor de declaratie van elementen en de relaties gescheiden zullen zijn. Dit zorgt ervoor dat de code moeilijk leesbaar is voor de eindgebruiker aangezien een bepaalde logische relatie zich niet direct bij de declaratie van het element zelf bevindt.

Buiten dat het resultaat niet meteen leesbaar is, heeft de codegenerator intern ook nog gebreken. Indien er in JavaFX/SceneBuilder expliciet met Text-Nodes gewerkt zal worden, zullen deze moeilijk geparsed worden door de codegenerator aangezien de Text-Node als attribuut wordt gezien van een grotere Node. De gebruiker zal dus op dit moment verplicht gebruik moeten maken van de Label-Node in SceneBuilder om tekst voor te stellen.

Daarnaast is de huidige oplossing voor de TabPane niet de meest efficiënte. Dit is omdat de Tab niet de enige uitzondering is, dat niet van het *Node*-type is. Er zijn immers nog enkele belangrijke containerachtige widgets die niet direct van het *Node*-type zijn of waarvan de *children* niet direct verkrijgbaar zijn zoals:

- Accordion,
- SplitPane,
- ToolBar,
- ButtonBar,
- TitledPane,
- ScrollPane.

Softwareingenieur Christoph Keimel suggereert om voor dit probleem een adapter te gebruiken met een eigen implementatie van de functie *getChildrenUnmodifiable()* om de *child-Nodes* te extraheren [49]. Hoewel dit een mogelijke oplossing is, geeft hij zelf toe dat dit misschien niet de beste is aangezien hij dit principe niet voor alle mogelijke aspecten heeft afgetoetst.

Bovendien heeft de codegenerator nog maar een beperkte subset van widgets die het kan genereren. Dit betekent dat de gebruiker nog rekening moet houden met de beperking dat niet alle elementen van SceneBuilder gebruikt kunnen worden voor een succesvol resultaat in Haskell door de codegenerator.

Daarnaast heeft het gebruik van de zelfgemaakte GTKWidget-klasse zoals eerder aangegeven het nadeel dat er meer onderhoudswerk is indien de subset van widgets wordt uitgebreid. Dit gebrek wordt, zoals eerder al aangegeven, wel gecompenseerd met de flexibiliteit om bepaalde attributen te manipuleren.

Ten slotte kent de codegenerator nog een tekortkoming op het vlak van gebruik. Het resultaat van de Haskell-code met gi-gtk werkt op dit moment alleen op Linux en niet direct *out-of-the-box* in Windows. Er zijn wel mogelijkheden om de gegeneerde code op Windows te laten werken met enkele tussenstappen en omwegen [53], [54]. Deze oplossingen vergen echter aanzienlijke moeite om het werkend te krijgen. Om een kant en klare oplossing te bekomen die in Windows werkt, dat vooral door de studenten in de opleiding gebruikt wordt, is een Docker-implementatie een mogelijke oplossing. De Docker-implementatie moet hierbij in staat zijn de Haskell-GUI-applicatie uit te voeren

in een XServer zoals X11. Op die manier wordt het mogelijk om Linux/UNIX GUIs weer te geven in Windows [55].

5.8 Toekomstig werk

Hoewel de codegenerator op dit moment de gekozen FXML-subset met succes omzet, is er nog plaats voor verdere verbetering en toekomstig werk. Zo is het ten eerste mogelijk de FXML-subset verder aan te vullen en is er de mogelijkheid de bestaande subset verder uit te breiden met meer overeenkomende attributen. Een extra stap bij het aanvullen van de subset van widgets, is om zelfgemaakte widgets te implementeren met de *GtkDrawingArea*.

Ook is het mogelijk in een vervolgstap een meer efficiënte *all-in-one* oplossing te zoeken om niet traditionele Nodes zoals de TabPane in het parsen af te handelen.

Verder is het belangrijk om *callback*-functies te implementeren in de codegenerator. Op dit moment maakt de codegenerator een GUI met diverse widgets aan, maar er is nog geen interactie tussen de verschillende widgets. Deze interactie vindt plaats door *callback*-functies. Het is dus vanzelfsprekend dat deze *callbacks* moeten worden gerealiseerd in de toekomst.

Daarnaast is het nodig om Windows beter te ondersteunen om een zo breed mogelijk publiek aan te spreken. Een mogelijke oplossing om de Haskell-GUIs werkend op Windows te krijgen is, zoals eerder aangegeven in “5.7 gebreken codegenerator”, om te werken met Docker om de Haskell-GUIs uit te voeren en een X-Server om de GUI-applicatie te weergeven in Windows.

Er is ook nog de mogelijkheid om een Haskell-*validator* te verwezenlijken in de codegenerator. Momenteel genereert de codegenerator enkel een Haskell-bestand en zet daar vervolgens de geschikte code, in tekstvorm, in. Er is dus de mogelijkheid dat er syntaxfouten voorkomen in het Haskell-bestand aangezien de tekst niet wordt gevalideerd. De Haskell-*validator* in de codegenerator valideert, onmiddellijk in Java, de Haskell-code die wordt aangemaakt om zo te voorkomen dat er syntaxfouten zijn in het Haskell-bestand.

Ten slotte is er de mogelijkheid om het project voor deze masterproef verder uit te breiden. Zo kan de codegenerator eventueel uitgebreid worden voor andere Haskell-bibliotheken, doordat de *parsing*-stap robuust is. Dit is realiseerbaar door de *parsing*-stap verder te generaliseren en hierbij zelfgemaakte datastructuren aan te maken voor een andere bibliotheek of eventueel bestaande Java-*bindings* van de bibliotheek naar keuze te gebruiken.

6 Conclusie

Het programmeren van GUIs in Haskell is nog steeds vrij omslachtig. Daarom is er nood voor een methode om de ontwikkeling van Haskell-GUIs minder omslachtig te maken. Een (begin van) antwoord hierop is deze masterproef waarin een codegenerator ontwikkeld werd die Haskell-code voor GUIs genereert. De codegenerator vertrekt hierbij vanuit de output van SceneBuilder-FXML waarbij SceneBuilder zelf al jarenlang succesvol gebruikt wordt door de studenten van de gezamenlijke opleiding.

Een eerste stap alvorens de codegenerator te ontwikkelen, was de bibliotheekkeuze. Hierbij staan toekomstbestendigheid en gebruiksgemak van de bibliotheek centraal, aangezien het de bedoeling is om in toekomstig werk de codegenerator uit te breiden. Uit een eerste technologiescan bleken vooral de medium-level bibliotheken zoals GTK2Hs, gi-gtk, FLTKHS en qtah een geschikte keuze. Na iedere kandidaat-bibliotheek verder te hebben uitgetest, bood gi-gtk de meeste voordelen op het vlak van toekomstbestendigheid en gebruiksgemak.

In de tweede stap volgde de ontwikkeling van de codegenerator zelf. Het algemeen stappenplan dat werd genomen was om eerst statische Haskell-begincode te genereren. Dan volgt het FXML-bestand *parsen* waarbij de elementen en relaties afzonderlijk worden aangemaakt. Uiteindelijk wordt er nog statische Haskell-eindcode gegenereerd.

Voor het parsen van FXML en de verdere ontwikkeling van de codegenerator zelf, werd Java als programmeertaal gekozen, aangezien JavaFX het al standaard mogelijk maakt om FXML in te lezen. Het gebruik van Java zorgde voor een eenvoudige methode om de relaties tussen de verschillende elementen in FXML te herkennen.

Het *parsen* zelf vertrok met een functie die de FXML-boom recursief doorloopt. Voor de overeenkomende widgets uit JavaFX en GTK (gi-gtk) is vervolgens besloten om zelfgemaakte datastructuren aan te maken voor een twaalftal prominente widgets. Deze zelfgemaakte datastructuren maken het mogelijk om attributen van de widgets eenvoudig te manipuleren.

Uiteindelijk is de codegenerator erin geslaagd om een twaalftal overeenkomende GUI-elementen uit JavaFX met precisie te kunnen omzetten naar Haskell. Dit is deels te danken aan de robuuste *parsing* waarbij zorgvuldig is omgegaan om de overeenkomende attributen van de GUI-elementen te selecteren en de relaties tussen de widgets nauwgezet te herkennen en af te handelen. Dit zorgt er ook voor dat toekomstige uitbreiding van widgets en relaties op een relatief eenvoudige manier gebeurt.

De codegenerator is echter niet perfect. Zo is één van de bezwaren dat er bij de zelfgemaakte GTKWidget-klasse meer onderhoudswerk is indien nieuwe widget-types worden aangemaakt, vergeleken met het gebruik van Java-GTK bindings. Dit gebrek wordt wel gecompenseerd met de flexibiliteit om bepaalde attributen eenvoudiger te manipuleren.

Ook zijn er nog enkele interne gebreken in de codegenerator waarbij het afhandelen van niet-traditionele *Node*-types zoals de TabPane niet op de meest efficiënte manier gebeurt.

Een ander bezwaar is dat op dit moment de codegenerator nog niet voor iedereen toegankelijk is. Dit is omdat Haskell-GUIs van gi-gtk momenteel alleen eenvoudig op Linux werken. Studenten die Windows gebruiken, en die op dit moment gebruik willen maken van de codegenerator om Haskell-GUIs te ontwikkelen, zullen op dit moment enkele tussenstappen moeten nemen om alles werkend te krijgen.

Een volgende stap om de codegenerator uit te breiden is om de FXML-subset verder aan te vullen. Ook is het mogelijk door de robuuste *parsing* om andere Haskell-bibliotheken te gebruiken, indien de *parsing*-stap verder wordt gegeneraliseerd. Daarnaast kan in een volgende stap de codegenerator verder toegankelijker worden gemaakt voor de studenten van het eerste bachelorjaar in de gezamenlijke opleiding.

Ondanks de huidige gebreken, vormt de ontwikkeling van de codegenerator een sterke basis om Haskell-GUIs minder omslachtig te maken. Studenten en mensen die in het algemeen geïnteresseerd zijn in functioneel programmeren hebben het voordeel dat de codegenerator een middel is om functioneel programmeren op het vlak van GUIs meer aanspreekbaar te maken. Het resultaat van de masterproef is dus een stap dichter gekomen bij het doel om functioneel programmeren gangbaar te maken waarbij de schoonheid en voordelen ervan worden blootgelegd aan een groter publiek.

Bibliografie

[1] J. Weaver, W. Gao, S. Chin, D. Iverson en J. Vos, Pro JavaFX 8: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients, Apress, 2014.

[2] L. Prechelt, „An Empirical Comparison of Seven Programming Languages," *Computing Practices,* pp. 23-29, 2000.

[3] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe en F. J. Paulo, „On Haskell and energy efficiency," *The Journal of Systems and Software,* vol. CXLIX, pp. 554-580, 2019.

[4] S. Marlow, L. Brandy, J. Coens en J. Purdy, „There is no Fork: an Abstraction for Efficient, Concurrent, and Concise Data Access," *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming,* pp. 325-337, 2014.

[5] A. Hejlsberg en M. Torgersen, „Overview of C# 3.0," Microsoft, March 2007. [Online]. Beschikbaar: https://docs.microsoft.com/en-us/previous-versions/dotnet/articles/bb308966(v=msdn.10)?redirectedfrom=MSDN. [Geopend 11 December 2021].

[6] R. G. Flatscher en G. Müller, „Employing Portable JavaFX GUIs with Scripting Languages," *Proceedings of the Central European Conference on Information and Intelligent Systems,* vol. XXXII, pp. 333-341, 2021.

[7] M. Tyson, „What is the JVM? Introducing the Java Virtual Machine," InfoWorld, 17 januari 2020. [Online]. Beschikbaar: https://www.infoworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html. [Geopend 1 juni 2022].

[8] J. Jenkov, „JavaFX Overview," Jenkov.com, 9 maart 2021. [Online]. Beschikbaar: http://tutorials.jenkov.com/javafx/overview.html. [Geopend 26 februari 2022].

[9] A. S. Mena, „A Real World Guide to Programming," *Practical Haskell,* vol. II, pp. 1-595, 2019.

[10] S. Nanz en C. A. Furia, „A Comparative Study of Programming Languages in Rosetta Code," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* pp. 778-788, 2015.

[11] V. Tirronen, S. Uusi-Mäkelä en V. Issomöttönen, „Understanding beginners' mistakes with Haskell," *Journal of Functional Programming,* pp. 1-30, 2015.

[12] D. Leijsen, „wxHaskell - A Portable and Concise GUI Library for Haskel," *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '04,* pp. 57-68, 2004.

[13] C. Antony en E. Conal, „Genuinely Functional User Interfaces," in *ACM SIGPLAN Haskell Workshop*, Firenze, 2001.

[14] A. J. Perlis, „EPIGRAMS IN PROGRAMMING," *SIGPLAN Notices,* vol. 17, nr. 9, pp. 7-13, 1982.

[15] E. Conal en J. Paul, „Haskell UI framework,” Stack Overflow, 26 augustus 2012. [Online]. Beschikbaar: https://stackoverflow.com/questions/2860988/haskell-ui-framework?noredirect=1&lq=1. [Geopend 27 februari 2022].

[16] M. Sage, „FranTk – A Declarative GUI Language for Haskell,” *ACM SIGPLAN Notices,* pp. 106-117, 2000.

[17] H. Perkins en M. Hotan, „GUI Event-Driven Programming,” 25 februari 2022. [Online]. Beschikbaar: https://courses.cs.washington.edu/courses/cse331/22wi/. [Geopend 27 februari 2022].

[18] J. Smart, K. Hock en S. Csomor, Cross-platform GUI Programming with wxWidgets, United States of America, 2006, pp. 1-700.

[19] M. M. Schrage, A. van IJzendoorn en L. C. van der Gaag, „Haskell Ready to Dazzle the Real World,” *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell - Haskell '05,* pp. 17-26, 2005.

[20] A. Siram, „Proceedings of the 2005 ACM SIGPLAN workshop on Haskell - Haskell '05,” Hask Anything!, 2016. [Online]. Beschikbaar: https://haskanything.com/content/presentation/presentation--fltkhs-easy-native-guis-in-haskell-today.html. [Geopend 11 December 2021].

[21] K. Aerts, „VISTO: A DECLARATIVE METHODOLOGY FOR GRAPHICAL USER INTERFACES, BASED ON HASKELL,” KULeuven, Leuven, 2001.

[22] K. Hoste, „An Introduction to Gtk2Hs, a Haskell GUI Library,” pp. 1-15.

[23] W. Swierstra, „The Real World,” 2007.

[24] J. Bouwmans, „Managing consistency between dependent objects,” *Center for Software Technology Department of Information and Computing Sciences,* pp. 1-46, 2010.

[25] S. E. Panitz en H. Rheinmain, „A Client for the Z21 Model Railway Control,” pp. 1-71.

[26] BrownHairedGirl, „List of language bindings for GTK,” Wikipedia, 8 mei 2022. [Online]. Beschikbaar: https://en.wikipedia.org/wiki/List_of_language_bindings_for_GTK. [Geopend 1 juni 2022].

[27] D. Wagner, „Threading and Gtk2Hs,” [Online]. Beschikbaar: http://dmwit.com/gtk2hs/. [Geopend 11 December 2021].

[28] V. Zavialov, „Introduction to GUI programming in Haskell,” Serokell, 2 oktober 2020. [Online]. Beschikbaar: https://www.youtube.com/watch?v=k1aq8ikO-8Q&t=1105s. [Geopend 26 februari 2022].

[29] „haskell-gi,” haskell-gi, januari 2022. [Online]. Beschikbaar: https://github.com/haskell-gi/haskell-gi. [Geopend 26 februari 2020].

[30] D. Shaswata, „GTK 4.0 has Officially Released with Major Improvements!," It's FOSS News, 18 december 2020. [Online]. Beschikbaar: https://news.itsfoss.com/gtk-4-release/. [Geopend 1 juni 2022].

[31] T. Hallgren en M. Carlsson, „Programming with Fudgets," *Computing Science,* pp. 3-49, 1995.

[32] B. Gardiner, „Qtah - Qt Bindings for Haskell," khumba.net, 2021. [Online]. Beschikbaar: http://khumba.net/projects/qtah. [Geopend 12 Maart 2022].

[33] Qt, „Qt Creator - A Cross-platform IDE for software development," Qt, 2022. [Online]. Beschikbaar: https://www.qt.io/product/development-tools. [Geopend 7 mei 2022].

[34] B. Gardiner, „Hoppy - C++ FFI Generator for Haskell," khumba.net, 2021. [Online]. Beschikbaar: http://khumba.net/projects/hoppy. [Geopend 12 Maart 2022].

[35] J. P. Ugarte, „Glade - A User Interface Designer," Glade, 20 november 2020. [Online]. Beschikbaar: https://glade.gnome.org/. [Geopend 3 mei 2020].

[36] liberforce, „How to install Glade 3.22 on Windows 10?," Stackoverflow, 19 september 2019. [Online]. Beschikbaar: https://stackoverflow.com/questions/57981763/how-to-install-glade-3-22-on-windows-10. [Geopend 4 mei 2022].

[37] MSYS2, „MSYS2 Software Distribution and Building Platform for Windows," MSYS2, 6 april 2022. [Online]. Beschikbaar: https://www.msys2.org/. [Geopend 4 mei 2022].

[38] Gluon, „Scene Builder," Gluon, 2022. [Online]. Beschikbaar: https://gluonhq.com/products/scene-builder/. [Geopend 3 mei 2022].

[39] Oracle, „Java® Platform, Standard Edition & Java Development Kit," Oracle, 2017. [Online]. Beschikbaar: https://docs.oracle.com/javase/9/docs/api/overview-summary.html. [Geopend 9 maart 2022].

[40] M. Sweet, C. P. Earls en B. Spitzak, „FLTK 1.0.10 Programming Manual," The FLTK Team, 7 mei 2001. [Online]. Beschikbaar: https://www.fltk.org/doc-1.0/. [Geopend 11 maart 2022].

[41] A. Siram, „fltkhs: FLTK bindings," Hackage, 21 februari 2020. [Online]. Beschikbaar: https://hackage.haskell.org/package/fltkhs. [Geopend 10 maart 2022].

[42] Qt, „Creating Custom Widgets for Qt Designer," Qt, 2022. [Online]. Beschikbaar: https://doc.qt.io/qt-5/designer-creating-custom-widgets.html. [Geopend 7 mei 2022].

[43] Lettier, „How To Build A Custom GTK Widget With Haskell," codeburst, 18 juni 2018. [Online]. Beschikbaar: https://codeburst.io/how-to-build-a-custom-gtk-widget-with-haskell-eaff04a6262. [Geopend 7 mei 2022].

[44] H. S. Cheng en G. S. V. R. K. Rao, „International Conference on Advanced Communication Technology (ICACT)," in *A Comparative Study and Benchmarking on XML Parsers*, Gangwon, Korea, 2007.

[45] kayz1, „Named placeholders in string formatting,” StackOverflow, 7 januari 2015. [Online]. Beschikbaar: https://stackoverflow.com/questions/2286648/named-placeholders-in-string-formatting. [Geopend 4 juni 2022].

[46] jewelsea, „Dump Utility,” 17 januari 2014. [Online]. Beschikbaar: https://stackoverflow.com/questions/21175767/how-to-traverse-the-entire-scene-graph-hierarchy. [Geopend 16 februari 2022].

[47] baeldung, „Using an Interface vs. Abstract Class in Java,” Baeldung, 23 juni 2021. [Online]. Beschikbaar: https://www.baeldung.com/java-interface-vs-abstract-class#:~:text=When%20to%20Use%20an%20Abstract%20Class,-Now%2C%20let%27s%20see&text=If%20we%20have%20specified%20requirements,the%20states%20of%20an%20object. [Geopend 18 april 2021].

[48] W. Thompson, I. G. Etxebarria en J. Platte, „gi-gtk-3.0.11: Gtk bindings,” Hackage, 27 december 2016. [Online]. Beschikbaar: https://hackage.haskell.org/package/gi-gtk-3.0.11/docs/. [Geopend 28 april 2022].

[49] C. Keimel, „Walking the JavaFX Scene Graph,” Software Development - Christoph Keimel, 9 maart 2015. [Online]. Beschikbaar: http://www.kware.net/?p=8. [Geopend 5 mei 2022].

[50] M. P. Robillard en K. Kutschera, „Lessons Learned While Migrating From Swing to JavaFX,” *IEEE Software,* vol. 3, nr. 37, pp. 78-85, 2020.

[51] A. Cowie, S. Pendyala en V. F. Lopes, „java-gnome 4.1,” Gnome, 4 mei 2013. [Online]. Beschikbaar: http://java-gnome.sourceforge.net/doc/api/4.1/org/gnome/gtk/Button.html. [Geopend 2 juni 2022].

[52] ptomato, „How can I read out the "glade ID" of a gtk3 object,” StackOverflow, 7 augustus 2016. [Online]. Beschikbaar: https://stackoverflow.com/questions/38812379/how-can-i-read-out-the-glade-id-of-a-gtk3-object. [Geopend 2 juni 2022].

[53] bradrn, „Using haskell gi in Windows,” Github, 21 september 2020. [Online]. Beschikbaar: https://github.com/haskell-gi/haskell-gi/wiki/Using-haskell-gi-in-Windows. [Geopend 5 mei 2022].

[54] D. Wong, „GTK Development Environment of Haskell under Windows,” Develop PAPER, 9 september 2019. [Online]. Beschikbaar: https://developpaper.com/gtk-development-environment-of-haskell-under-windows/. [Geopend 6 mei 2022].

[55] potatowagon, „How to use GUI apps in linux docker container from Windows Host,” Medium, 12 april 2020. [Online]. Beschikbaar: https://medium.com/@potatowagon/how-to-use-gui-apps-in-linux-docker-container-from-windows-host-485d3e1c64a3. [Geopend 3 juni 2022].

Bijlagen

Bijlage A: *Inhoud FXML-bestand o.b.v. het voorbeeld GUI-applicatie uit "3 Bibliotheekkeuze"*

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.Scene?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>
<?import javafx.stage.Stage?>

<Stage height="400.0" title="JavaFX demo-app" width="600.0"
 xmlns="http://javafx.com/javafx/17" xmlns:fx="http://javafx.com/fxml/1">
    <scene>
        <Scene>
         <GridPane maxHeight="1080.0" maxWidth="1920.0" prefHeight="400.0"
          prefWidth="600.0" hgap="10">
           <columnConstraints>
              <ColumnConstraints hgrow="SOMETIMES"
             maxWidth="540.0" minWidth="10.0" percentWidth="50.0" prefWidth="302.0" />
              <ColumnConstraints hgrow="SOMETIMES" maxWidth="331.0" minWidth="10.0"
             percentWidth="50.0" prefWidth="298.0" />
           </columnConstraints>
           <rowConstraints>
              <RowConstraints minHeight="10.0" vgrow="SOMETIMES" />
           </rowConstraints>
           <children>
                 <AnchorPane prefHeight="400.0" prefWidth="600.0">
                 <children>
                    <TextField layoutX="75.0" layoutY="223.0"
                      alignment="TOP_CENTER" promptText="Vul hier iets in"  />
                    <Label layoutX="90.0" layoutY="94.0" text="Typ hier uw tekst" />
                    <Label layoutX="88.0" layoutY="341.0" text="Hier komt de tekst " />
                 </children>
              </AnchorPane>
              <Button maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
             mnemonicParsing="false" prefHeight="477.0" prefWidth="343.0"
             text="Toon tekst" GridPane.columnIndex="1" />
           </children>
         </GridPane>
        </Scene>
    </scene>
</Stage>
```

Bijlage B: *Inhoud Glade-bestand uit het voorbeeld GUI-applicatie uit "3 Bibliotheekkeuze"*

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.38.2 -->
<interface>
  <requires lib="gtk+" version="3.24"/>
  <object class="GtkWindow">
    <property name="width-request">600</property>
    <property name="height-request">400</property>
    <property name="can-focus">False</property>
    <child>
      <!-- n-columns=2 n-rows=1 -->
      <object class="GtkGrid">
        <property name="visible">True</property>
        <property name="can-focus">False</property>
        <child>
          <object class="GtkButton">
            <property name="label" translatable="yes">Toon Tekst</property>
            <property name="width-request">300</property>
            <property name="height-request">400</property>
            <property name="visible">True</property>
            <property name="can-focus">True</property>
            <property name="receives-default">True</property>
          </object>
          <packing>
            <property name="left-attach">1</property>
            <property name="top-attach">0</property>
          </packing>
        </child>
        <child>
          <object class="GtkLayout">
            <property name="width-request">300</property>
            <property name="height-request">400</property>
            <property name="visible">True</property>
            <property name="can-focus">False</property>
            <child>
              <object class="GtkEntry">
                <property name="width-request">150</property>
                <property name="height-request">26</property>
                <property name="visible">True</property>
                <property name="can-focus">True</property>
              </object>
              <packing>
                <property name="x">75</property>
                <property name="y">223</property>
              </packing>
            </child>
            <child>
              <object class="GtkLabel">
                <property name="width-request">100</property>
                <property name="height-request">80</property>
                <property name="visible">True</property>
                <property name="can-focus">False</property>
                <property name="label" translatable="yes">Typ hier uw tekst</property>
              </object>
              <packing>
                <property name="x">90</property>
                <property name="y">94</property>
              </packing>
```

Bijlage C: *Specificaties van de computer om de compileertijden te testen*

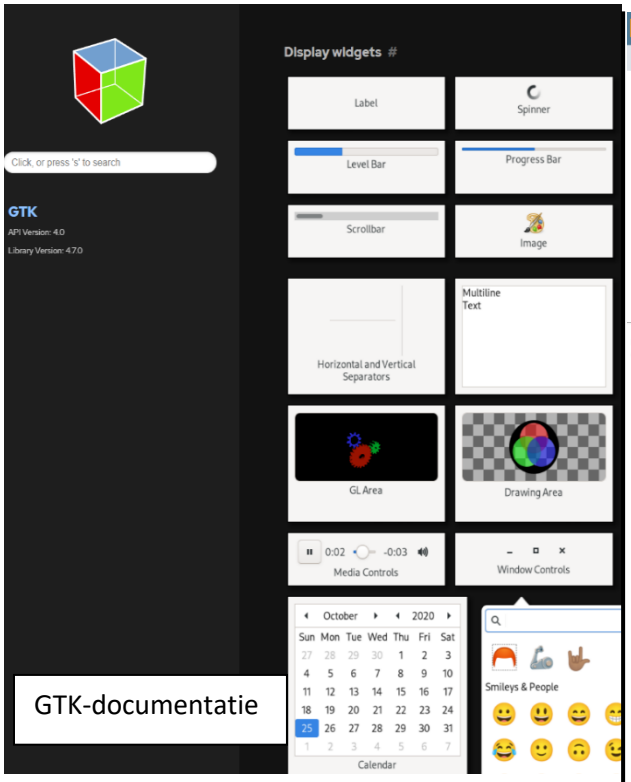
Acer Swift X (SFX14-41G) laptop:

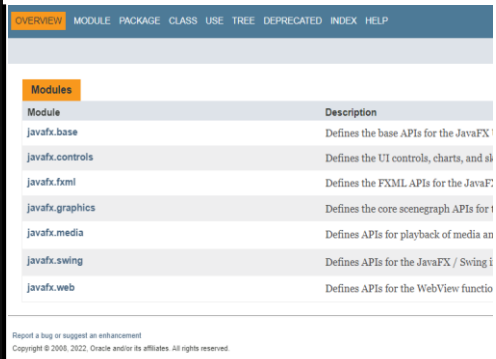
Processor	AMD Ryzen 7 5800U
Videokaart	NVIDIA GeForce RTX 3050 Ti (Laptop, 40W)
Geheugen	16GB LPDDR4x
Opslag	512GB SSD
Besturingssysteem	Ubuntu 20.04

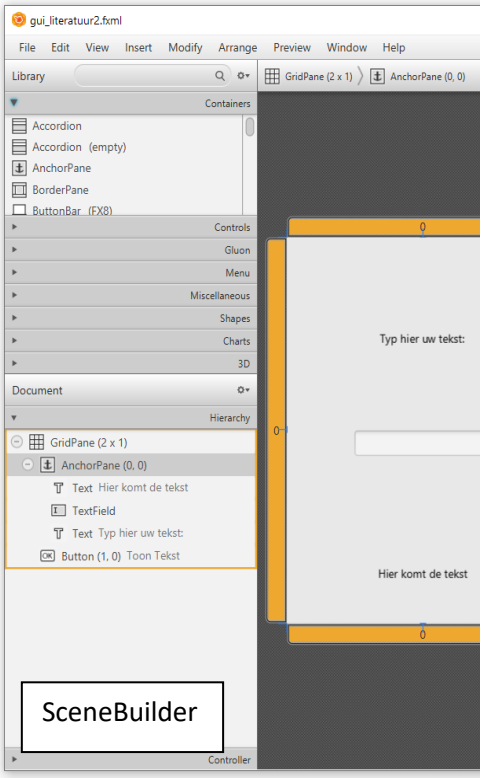
Bijlage D: *FXML van het voorbeeld GUI-applicatie*

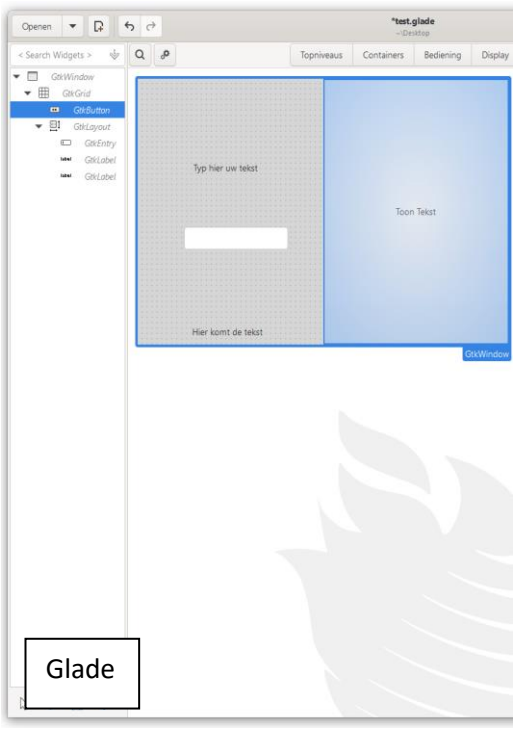
```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.Scene?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>
<?import javafx.stage.Stage?>

<Stage height="400.0" title="JavaFX demo-app" width="600.0"
         xmlns="http://javafx.com/javafx/17" xmlns:fx="http://javafx.com/fxml/1">
    <scene>
        <Scene>
         <GridPane maxHeight="1080.0" maxWidth="1920.0"
          prefHeight="400.0" prefWidth="600.0" hgap="10">
            <columnConstraints>
               <ColumnConstraints hgrow="SOMETIMES" maxWidth="540.0"
             minWidth="10.0" percentWidth="50.0" prefWidth="302.0" />
               <ColumnConstraints hgrow="SOMETIMES" maxWidth="331.0"
             minWidth="10.0" percentWidth="50.0" prefWidth="298.0" />
            </columnConstraints>
            <rowConstraints>
               <RowConstraints minHeight="10.0" vgrow="SOMETIMES" />
            </rowConstraints>
            <children>
                  <AnchorPane prefHeight="400.0" prefWidth="600.0">
                  <children>
                     <TextField layoutX="75.0" layoutY="223.0"
                   alignment="TOP_CENTER" promptText="Vul hier iets in"  />
                     <Label layoutX="90.0" layoutY="94.0" text="Typ hier uw tekst" />
                     <Label layoutX="88.0" layoutY="341.0" text="Hier komt de tekst " />
                  </children>
               </AnchorPane>
               <Button maxHeight="1.79" maxWidth="1.79" mnemonicParsing="false"
             prefHeight="477.0" prefWidth="343.0" text="Toon tekst"
             GridPane.columnIndex="1" />
            </children>
         </GridPane>
        </Scene>
    </scene>
</Stage>
```

Bijlage E: *Vergelijking van de verschillende websites en tools die gebruikt zijn om de overeenkomstige GUI-elementen te vinden tussen JavaFX en GTK*









Bijlage F: *Voorbeeld Haskell-code van Buttons in een Layout op verschillende mogelijke structuren*

```
1  --Deelvariatie A: declaraties en relaties gescheiden
2
3  --imports
4  module Main (Main.main) where
5  import qualified GI.Gtk as Gtk
6  ...
7
8  main :: IO ()
9  main = do
10   Gtk.init Nothing
11   window <- Gtk.windowNew WindowTypeToplevel
12   Gtk.set window [Gtk.windowTitle := "JavaFX demo app",
13                   Gtk.windowResizable := True,
14                   Gtk.windowDefaultWidth := 600,
15                   Gtk.windowDefaultHeight := 400]
16
17   -- DECLARATIONS
18   layout_container <- Gtk.layoutNew (Nothing::Maybe Adjustment) --Layout
19                                     (Nothing::Maybe Adjustment)
20   button_first <-  Gtk.buttonNewWithLabel "Test"  --Button1
21   button_second <- Gtk.buttonNewWithLabel "Test2" --Button2
22
23
24   -- RELATIONS
25   Gtk.layoutPut layout_container button_first 236 162  --Button1 in Layout
26   Gtk.layoutPut layout_container button_second 236 162 --Button2 in Layout
27   Gtk.setContainerChild window layout_container        --Layout in window
28
29   Gtk.onWidgetDestroy window Gtk.mainQuit
30   Gtk.widgetShowAll window
31   Gtk.main
```

```haskell
--Deelvariatie B: declaraties en relaties gecombineerd

--imports
module Main (Main.main) where
import qualified GI.Gtk as Gtk
...

main :: IO ()
main = do
  Gtk.init Nothing
  window <- Gtk.windowNew WindowTypeToplevel
  Gtk.set window [Gtk.windowTitle := "JavaFX demo app",
                  Gtk.windowResizable := True,
                  Gtk.windowDefaultWidth := 600,
                  Gtk.windowDefaultHeight := 400]

  -- DECLARATION + RELATIONS
  layout_container <- Gtk.layoutNew (Nothing::Maybe Adjustment) --Layout
                                    (Nothing::Maybe Adjustment)
  Gtk.setContainerChild window layout_container        --Layout in window

  button_first <-  Gtk.buttonNewWithLabel "Test"  --Button1
  Gtk.layoutPut layout_container button_first 236 162  --Button1 in Layout

  button_second <- Gtk.buttonNewWithLabel "Test2" --Button2
  Gtk.layoutPut layout_container button_second 236 162 --Button2 in Layout

  Gtk.onWidgetDestroy window Gtk.mainQuit
  Gtk.widgetShowAll window
  Gtk.main
```

Bijlage G: *Tool om placeholders te vervangen in een String*

```
1 public static String format(String format, Map<String, Object> values) {
2     StringBuilder formatter = new StringBuilder(format);
3     List<Object> valueList = new ArrayList<Object>();
4 
5     Matcher matcher = Pattern.compile("\\$\\{(\\w+)}").matcher(format);
6 
7     while (matcher.find()) {
8         String key = matcher.group(1);
9 
10        String formatKey = String.format("${%s}", key);
11        int index = formatter.indexOf(formatKey);
12 
13        if (index != -1) {
14            formatter.replace(index, index + formatKey.length(), "%s");
15            valueList.add(values.get(key));
16        }
17    }
18    return String.format(formatter.toString(), valueList.toArray());
19 }
```

Bijlage H: *Resultaat codegenerator van demo-applicatie uit "3 Bibliotheekkeuze"*

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE OverloadedLabels  #-}
{-# LANGUAGE ExtendedDefaultRules #-}

module Main (Main.main) where
import qualified GI.Gtk as Gtk
import GI.Gtk.Enums (WindowType(..), Orientation(..))
import GI.Gtk (Adjustment(Adjustment))
import Data.GI.Base ( AttrOp((:=)) )

main :: IO ()
main = do
  Gtk.init Nothing
  window <- Gtk.windowNew WindowTypeToplevel
  Gtk.set window [Gtk.windowTitle := "JavaFX demo-app", Gtk.windowResizable := True,
                  Gtk.windowDefaultWidth := 600, Gtk.windowDefaultHeight := 400]

  -- Grid
  grid_867946641 <- Gtk.gridNew
  Gtk.gridSetColumnHomogeneous grid_867946641 True
  Gtk.gridSetRowHomogeneous grid_867946641 True
  Gtk.gridSetColumnSpacing grid_867946641 10

  -- Layout
  layout_1143131353Container <- Gtk.layoutNew (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)

  -- Entry
  entry_1006469344 <- Gtk.entryNew
  Gtk.entrySetPlaceholderText entry_1006469344 (Just "Vul hier iets in")
  Gtk.entrySetAlignment entry_1006469344 0.5
  entry_1006469344Container <- Gtk.boxNew OrientationHorizontal 1
  Gtk.set entry_1006469344Container [Gtk.widgetWidthRequest := 171, Gtk.widgetHeightRequest := 24]
  Gtk.boxPackStart entry_1006469344Container entry_1006469344 True True 0

  --Label
  label_511781588 <- Gtk.labelNew (Just "Typ hier uw tekst")

  --Label
  label_1107663810 <- Gtk.labelNew (Just "Hier komt de tekst ")

  -- Button
  button_322866115 <- Gtk.buttonNewWithLabel "Toon tekst"
  button_322866115Container <- Gtk.boxNew OrientationHorizontal 1
  Gtk.set button_322866115Container [Gtk.widgetWidthRequest := 295, Gtk.widgetHeightRequest := 400]
  Gtk.boxPackStart button_322866115Container button_322866115 True True 0

  -- Relations
  Gtk.gridAttach grid_867946641 layout_1143131353Container 0 0 1 1
  Gtk.layoutPut layout_1143131353Container entry_1006469344Container 75 223
  Gtk.layoutPut layout_1143131353Container label_511781588 90 94
  Gtk.layoutPut layout_1143131353Container label_1107663810 88 341
  Gtk.gridAttach grid_867946641 button_322866115Container 1 0 1 1

  Gtk.setContainerChild window grid_867946641

  Gtk.onWidgetDestroy window Gtk.mainQuit
  Gtk.widgetShowAll window
  Gtk.main
```

***Bijlage I**: Resultaat codegenerator van demo-applicatie met alle GUI-elementen uitgetest*

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE OverloadedLabels  #-}
{-# LANGUAGE ExtendedDefaultRules #-}

module Main (Main.main) where
import qualified GI.Gtk as Gtk
import GI.Gtk.Enums (WindowType(..), Orientation(..))
import GI.Gtk (Adjustment(Adjustment))
import Data.GI.Base ( AttrOp((:=)) )

main :: IO ()
main = do
  Gtk.init Nothing
  window <- Gtk.windowNew WindowTypeToplevel
  Gtk.set window [Gtk.windowTitle := "Alle widgets demo", Gtk.windowResizable := True,
                                                  Gtk.windowDefaultWidth := 759, Gtk.windowDefaultHeight := 473]

  -- Layout
  layout_1899923384Container <- Gtk.layoutNew (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)

  --Label
  tabLabel_111686529 <- Gtk.labelNew (Just "Tabje 1")

  --Label
  tabLabel_111686530 <- Gtk.labelNew (Just "Tabje 2")

  --Label
  tabLabel_111686531 <- Gtk.labelNew (Just "Tabje 3")

  --Notebook
  noteBook_656612402 <- Gtk.notebookNew
  noteBook_656612402Container <- Gtk.boxNew OrientationHorizontal 1
  Gtk.set noteBook_656612402Container [Gtk.widgetWidthRequest := 634, Gtk.widgetHeightRequest := 248]
  Gtk.boxPackStart noteBook_656612402Container noteBook_656612402 True True 0

  -- hBox
  hBox_399533187Container <- Gtk.boxNew Gtk.OrientationHorizontal 20

  -- Entry
  entry_835165604 <- Gtk.entryNew
  Gtk.entrySetPlaceholderText entry_835165604 (Just "Entry")
  Gtk.entrySetAlignment entry_835165604 0.0
  entry_835165604Container <- Gtk.boxNew OrientationHorizontal 1
  Gtk.set entry_835165604Container [Gtk.widgetWidthRequest := 249, Gtk.widgetHeightRequest := 110]
  Gtk.boxPackStart entry_835165604Container entry_835165604 True True 0

  -- Button
  button_150346463 <- Gtk.buttonNewWithLabel "Button"
  button_150346463Container <- Gtk.boxNew OrientationHorizontal 1
  Gtk.set button_150346463Container [Gtk.widgetWidthRequest := 90, Gtk.widgetHeightRequest := 143]
  Gtk.boxPackStart button_150346463Container button_150346463 True True 0

  -- CheckButton
  checkButton_605203105 <- Gtk.checkButtonNewWithLabel "CheckBox"

  -- Grid
  grid_957020879 <- Gtk.gridNew
  Gtk.gridSetColumnHomogeneous grid_957020879 True
  Gtk.gridSetRowHomogeneous grid_957020879 True
  Gtk.gridSetRowSpacing grid_957020879 20

  -- Layout
  layout_1765184761Container <- Gtk.layoutNew (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)

  -- ComboBoxText
  comboBoxText_566163909 <- Gtk.comboBoxTextNew
  Gtk.comboBoxTextAppendText comboBoxText_566163909 "Een"
  Gtk.comboBoxTextAppendText comboBoxText_566163909 "Twee"
  Gtk.comboBoxTextAppendText comboBoxText_566163909 "Drie"
  comboBoxTextContainer_566163909 <- Gtk.boxNew OrientationHorizontal 1
  Gtk.set comboBoxTextContainer_566163909 [Gtk.widgetWidthRequest := 150, Gtk.widgetHeightRequest := 24]
  Gtk.boxPackStart comboBoxTextContainer_566163909 comboBoxText_566163909 True True 0
```

```
73
74  -- Button
75  button_1614992399 <- Gtk.buttonNewWithLabel "Button"
76  button_1614992399Container <- Gtk.boxNew OrientationHorizontal 1
77  Gtk.set button_1614992399Container [Gtk.widgetWidthRequest := 317, Gtk.widgetHeightRequest := 43]
78  Gtk.boxPackStart button_1614992399Container button_1614992399 True True 0
79
80  --Label
81  label_920938771 <- Gtk.labelNew (Just "Label")
82
83  -- Entry
84  entry_2039697289 <- Gtk.entryNew
85  Gtk.entrySetAlignment entry_2039697289 0.0
86  entry_2039697289Container <- Gtk.boxNew OrientationHorizontal 1
87  Gtk.set entry_2039697289Container [Gtk.widgetWidthRequest := 317, Gtk.widgetHeightRequest := 24]
88  Gtk.boxPackStart entry_2039697289Container entry_2039697289 True True 0
89
90  -- Layout
91  layout_1002893863Container <- Gtk.layoutNew (Nothing::Maybe Adjustment) (Nothing::Maybe Adjustment)
92
93  -- CheckButton
94  checkButton_1349888808 <- Gtk.checkButtonNewWithLabel "Java"
95
96  -- RadioButton
97  radioButton_1332312227 <- Gtk.radioButtonNewWithLabelFromWidget (Nothing::Maybe Gtk.RadioButton) "Nederlands"
98
99  -- RadioButton
100 radioButton_1193599565 <- Gtk.radioButtonNewWithLabelFromWidget (Nothing::Maybe Gtk.RadioButton) "Frans"
101
102 -- RadioButton
103 radioButton_511781588 <- Gtk.radioButtonNewWithLabelFromWidget (Nothing::Maybe Gtk.RadioButton) "Engels"
104
105 -- CheckButton
106 checkButton_1107663810 <- Gtk.checkButtonNewWithLabel "Python"
107
108 -- CheckButton
109 checkButton_575881029 <- Gtk.checkButtonNewWithLabel "Haskell"
110
111 --Label
112 label_1037550390 <- Gtk.labelNew (Just "Wat zijn uw favoriete programeertalen?")
113
114 --Label
115 label_105212638 <- Gtk.labelNew (Just "Wat is uw moedertaal?")
116
117 -- LinkButton
118 linkButton_859823420 <- Gtk.linkButtonNewWithLabel "" (Just "Hieronder is de Tabpane")
119
120 --Toggle Groups
121 Gtk.radioButtonSetGroup radioButton_1332312227 [radioButton_1193599565, radioButton_511781588]
122
123 -- Relations
124 Gtk.layoutPut layout_1899923384Container noteBook_656612402Container 62 143
125 Gtk.boxPackStart hBox_399533187Container entry_835165604Container True True 0
126 Gtk.boxPackStart hBox_399533187Container button_150346463Container True True 0
127 Gtk.boxPackStart hBox_399533187Container checkButton_605203105 True True 0
128 Gtk.gridAttach grid_957020879 layout_1765184761Container 0 0 1 1
129 Gtk.layoutPut layout_1765184761Container comboBoxTextContainer_566163909 84 37
130 Gtk.gridAttach grid_957020879 button_1614992399Container 1 0 1 1
131 Gtk.gridAttach grid_957020879 label_920938771 0 1 1 1
132 Gtk.gridAttach grid_957020879 entry_2039697289Container 1 1 1 1
133 Gtk.layoutPut layout_1002893863Container checkButton_1349888808 46 94
134 Gtk.layoutPut layout_1002893863Container radioButton_1332312227 370 93
135 Gtk.layoutPut layout_1002893863Container radioButton_1193599565 370 120
136 Gtk.layoutPut layout_1002893863Container radioButton_511781588 370 145
137 Gtk.layoutPut layout_1002893863Container checkButton_1107663810 46 121
138 Gtk.layoutPut layout_1002893863Container checkButton_575881029 46 146
139 Gtk.layoutPut layout_1002893863Container label_1037550390 46 58
140 Gtk.layoutPut layout_1002893863Container label_105212638 370 58
141 Gtk.layoutPut layout_1899923384Container linkButton_859823420 294 79
142
143 --Notebook relations
144 Gtk.notebookAppendPage noteBook_656612402 layout_1002893863Container (Just tabLabel_111686529)
145 Gtk.notebookAppendPage noteBook_656612402 grid_957020879 (Just tabLabel_111686530)
146 Gtk.notebookAppendPage noteBook_656612402 hBox_399533187Container (Just tabLabel_111686531)
147
148 Gtk.setContainerChild window layout_1899923384Container
149
150 Gtk.onWidgetDestroy window Gtk.mainQuit
151 Gtk.widgetShowAll window
152 Gtk.main
```