GUIDED-SCRIPT-EDITOR

Studienarbeit HS2013

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

EATON

Autoren:	Slobodan Stojkovic & Stefan Leimgruber
Betreuer:	Prof. Dr. Luc Bläser
Projektpartner:	Eaton Automation GmbH, St.Gallen
Experte:	Michael Greminger
Abteilung:	Informatik

Inhaltsverzeichnis

Aufgabenstellung

Aufgabenstellung Studienarbeit für Stefan Leimgruber und Slobodan Stojkovic:

Guided Script Editor für das Galileo Programmiersystem

1. Auftraggeber und Betreuer

Diese Studienarbeit findet in Zusammenarbeit mit der *Eaton Automation GmbH* statt.

Ansprechpartner Auftraggeber:

- Matthias Schwyn, Eaton Automation GmbH, Software Engineer R&D Lean Automation

Betreuer HSR:

- Prof. Dr. Luc Bläser, Institut für Software

2. Ausgangslage

Eaton Automation hat ein grafisches Programmiersystem namens Galileo, welches in der Automationsbranche dazu eingesetzt wird, um interaktive Benutzeroberflächen zu entwickeln. Diese Benutzeroberflächen beinhalten verschiedene grafische Komponenten wie Knöpfe, Listen, Fehlerlisten, Rezepteditor und so weiter. Eine solche Benutzeroberfläche wird zunächst durch einen Übersetzer in ein Kompilat umgewandelt. Danach kann dieses durch ein Laufzeitsystem auf dem Zielsystem ausgewertet werden, um das entsprechende User Interface darzustellen.

Galileo beinhaltet eine einfache Scriptsprache, welche dazu dient, dass Benutzer gewisse Abläufe selbst programmieren können. Dazu soll nun ein Editor angeboten werden, welcher den Benutzer bei der Entwicklung des Scripts interaktiv unterstützt. Das Projekt lässt sich verschiedene Etappen gliedern:

Teil 1: Analyse

Es soll die aktuelle Skriptsprache analysiert werden und Vorschläge zur Vereinfachung und Vereinheitlichung der Funktionsaufrufe in dem Programmiermodell gemacht werden. Erste mögliche Varianten für Funktionsaufrufe sind (analog zu modularen oder objekt-orientierten Sprachen): Modul.Funktion(Objekt, Argument1, ...) und/ oder Objekt.Funktion(Argument1, ...).

Bei den Vorschlägen gilt es besonders darauf zu achten, dass sie die Benutzerführung durch Templates / Intellisense optimal unterstützen.

Es sollen die Möglichkeiten von Intellisense, Tooltips, Templates, "Drag and Drop" von Funktionen und Parametern sowie weitere Ansätze zur Unterstützung/Führung des Nutzers analysiert werden.

Daraus soll eine Benutzerführung erarbeitet werden, die einerseits dem versierten Anwender erlaubt, produktiv und schnell zu skripten, und andererseits auch den Anfänger unterstützt und möglichst verhindert, dass dieser syntaktisch falsche Scripte erstellt.

Es soll zudem evaluiert werden, ob es geeignete Winforms Texteditorcontrols gibt, die Syntax-Highlighting unterstützen.

Teil 2: Design und Implementation Editor mit Syntax Check

Es soll ein Editor für das Galileo Programmiersystem entwickelt werden, welcher das in Teil 1 definierte Programmiermodell für die Aufrufe unterstützt. In dieser Etappe soll vor allem die Syntaxprüfung mit einer einfachen Benutzerführung (z.B. Auswahl der Funktionen) im Editor realisiert werden.

Hierbei soll insbesondere auf Erweiterbarkeit geachtet werden, d.h. für den Fall, dass Eaton Automation die Sprache in Zukunft erweitern oder eine andere Sprache einbauen wird.

Teil 3 (optional): Erweiterte Editor-Funktionen (Syntax Highlighting, Einrücken, Intellisense)

Optional können fortgeschrittene Editor-Features wie Syntax Highlighting, Einrücken, erweitertes Intellisense, Block Completion, Templates etc. für eine möglichst komfortable Benutzerführung umgesetzt werden.

3. Ziele und Aufgabenstellung

Die Aufgabe dieser Arbeit ist es, dass ein Editor mit geeigneter Benutzerführung für das Gallileo Programmiersystem entworfen und implementiert wird. Als Technologie wird .NET in Absprache mit Auftraggeber eingesetzt.

Folgende spezifische Ziele werden vorgegeben:

- Aufnahme der genauen Anforderungen für den neuen Editor.
- Kleine Konzeptstudie zur Sprachsyntax und Benutzerführung im Editor. Wahl eines Ansatzes in Absprache mit dem Betreuer und dem Auftraggeber.
- Entwurf und Implementierung des Editors in .NET für die Galileo-Sprache mit der gewählten Sprachsyntax und einfacher Benutzerführung. Automatisierte Tests mit einer Kollektion von signifikanten Beispiel-Script-Code-Szenarien.
- Optional: Erweiterte Editor-Features für möglichst komfortable Benutzung.

4. Zur Durchführung

Mit dem HSR-Betreuer finden wöchentliche Besprechungen statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen. Besprechungen mit dem Auftraggeber werden nach Bedarf durchgeführt.

Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer und dem Auftraggeber per E-Mail zugestellt wird.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsresultate in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsresultate erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

5. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen (siehe https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html?&L=0). Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Die Dokumentation ist vollständig auf CD/DVD in 3 Exemplaren abzugeben. Auf Wunsch ist für den Auftraggeber eine gedruckte Version zu erstellen.

6. Termine

Siehe auch Terminplan auf https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html?&L=0

16.09.13	Beginn der Studienarbeit, Ausgabe der Aufgabenstellung durch die Betreuer.
16.12.13	Die Studierenden senden folgende Dokumente der Arbeit per Email zur Prüfung an ihre Betreuer: - Kurzfassung - A0-Poster Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen unter den allgemeinen Infos Diplom-, Bachelor- und Studienarbeiten zur Verfügung.
20.12.13	Die Studierenden senden die vom Betreuer abgenommene und freigegebene Kurzfassung als **Word-Dokument** an das Studiengangsekretariat .
20.12.13	Abgabe des Berichtes an den Betreuer bis 17.00 Uhr.

7. Beurteilung

Eine erfolgreiche Studienarbeit erhält 8 ECTS-Punkten (1 ECTS Punkt entspricht einer Arbeitsleistung von ca. 25 bis 30 Stunden). Für die Modulbeschreibung der Studienarbeit siehe https://unterricht.hsr.ch/staticWeb/allModules/19456_M_SAI.html

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/5
2. Berichte (Abstract, Mgmt Summary, techn. u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/5
3. Inhalt *)	3/5

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit festgelegt.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Studienarbeiten.

Rapperswil, den 20. September 2013

Der verantwortliche Dozent

Prof. Dr. Luc Bläser
Institut für Software
Hochschule für Technik Rapperswil

Erklärung der eigenständigen Arbeit

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Ort, Datum

Rapperswil, 20. Dezember 2013

Name, Unterschrift

Slobodan Stojkovic *Stefan Leimgruber*

Abstract

Ausgangslage

Eaton Automation ist der Hersteller eines grafischen Programmiersystems namens Galileo, welches in der Automationsbranche dazu eingesetzt wird, um interaktive Benutzeroberflächen zu entwickeln. Diese Benutzeroberflächen beinhalten verschiedene grafische Komponenten wie Knöpfe, Listen, Fehlerlisten, Rezepteditor und einiges mehr. Galileo bietet zudem eine einfache Script-Sprache, mit welcher Benutzer Automations-Abläufe selber programmieren können. Dazu soll nun ein moderner Editor im Sinne einer integrierten Entwicklungsumgebung realisiert werden, welcher den Benutzer bei der Programmierung der Automation-Scripts interaktiv unterstützt.

Aufgabenstellung

Die Aufgabe teilt sich grundsätzlich in zwei Teile: Als erster Teil gilt es, die aktuelle Script-Sprache zu analysieren und Vorschläge zur Vereinfachung und Vereinheitlichung der Funktionsaufrufe in dem Programmiermodell zu unterbreiten. Im zweiten Teil soll schliesslich der definierte Sprachumfang der Script-Sprache für den Editor implementiert werden. Hierbei soll auf Erweiterbarkeit geachtet werden, insbesondere dafür, dass Eaton die Sprache in Zukunft erweitern oder ändern kann.

Vorgehen

In diesem Projekt wurde zuerst der Sprachumfang definiert und die Grammatik der Script-Sprache formalisiert. Danach wurden verschiedene Verarbeitungsschritte eines Compilers umgesetzt: Mit dem Lexer wird der Source-Text in einen Symbol-Strom zerlegt. Diese Information dient dem Syntax-Highlighting im Editor. Danach wird mit dem Parser der Symbol-Strom in einen abstrakten Syntax-Baum abgebildet und währenddessen die syntaktische Korrektheit des Scripts überprüft. Zuletzt folgt der semantische Checker, welcher die semantischen Regeln auf dem Syntax-Baum überprüft.

Ergebnisse

Die Lösung wurde im Visual Studio 2010 mit C# und dem .NET Framework 4.5 entwickelt. Die gesamte Syntax der Sprache wurde umgesetzt, wobei die spezifischen Funktionsdefinitionen jedoch konfigurierbar sind. Das Resultat ist gelungen: Es ist ein voll funktionsfähiger Programmier-Editor, der Eaton Scripts mit dem gesamten Komfort von Auto-Completion, Syntax-Highlighting sowie automatischer Markierung der Syntax- und Semantik-Fehler unterstützt.

Management Summary

Ausgangslage

Eaton Automation ist in der Herstellung von Panels (Steuerungsbildschirme) für Maschinen tätig. Diese Panels können für jeden Kunden individuell angepasst werden. Dafür liefert Eaton Automation mit seiner Hardware auch die Software namens Galileo mit, mit welcher der Kunde seine Benutzeroberfläche selbstständig zusammenstellen kann. Auf die Benutzeroberfläche können diverse Steuerungselemente platziert werden. Gibt es nun Funktionalitäten, welche die Steuerungselemente nicht von Haus aus bedienen können, besteht die Möglichkeit eigene Abläufe und Prozeduren zu definieren mit einer sogenannten Script-Sprache.

Zielsetzung

Ziel dieser Arbeit ist es den aktuellen Script-Editor durch einen Neuen „intelligenten" Editor zu ersetzten. Dieser Editor soll dem Kunden erlauben, mit einfachen Mitteln eigene Funktionalität einzubauen. Dies sollte unter anderem mit automatischer Vervollständigung und Überprüfung der Logik unterstützt werden. Der alte Editor schränke ausserdem den fortgeschrittenen Benutzer bei der Arbeit zu fest ein.

Vorgehen

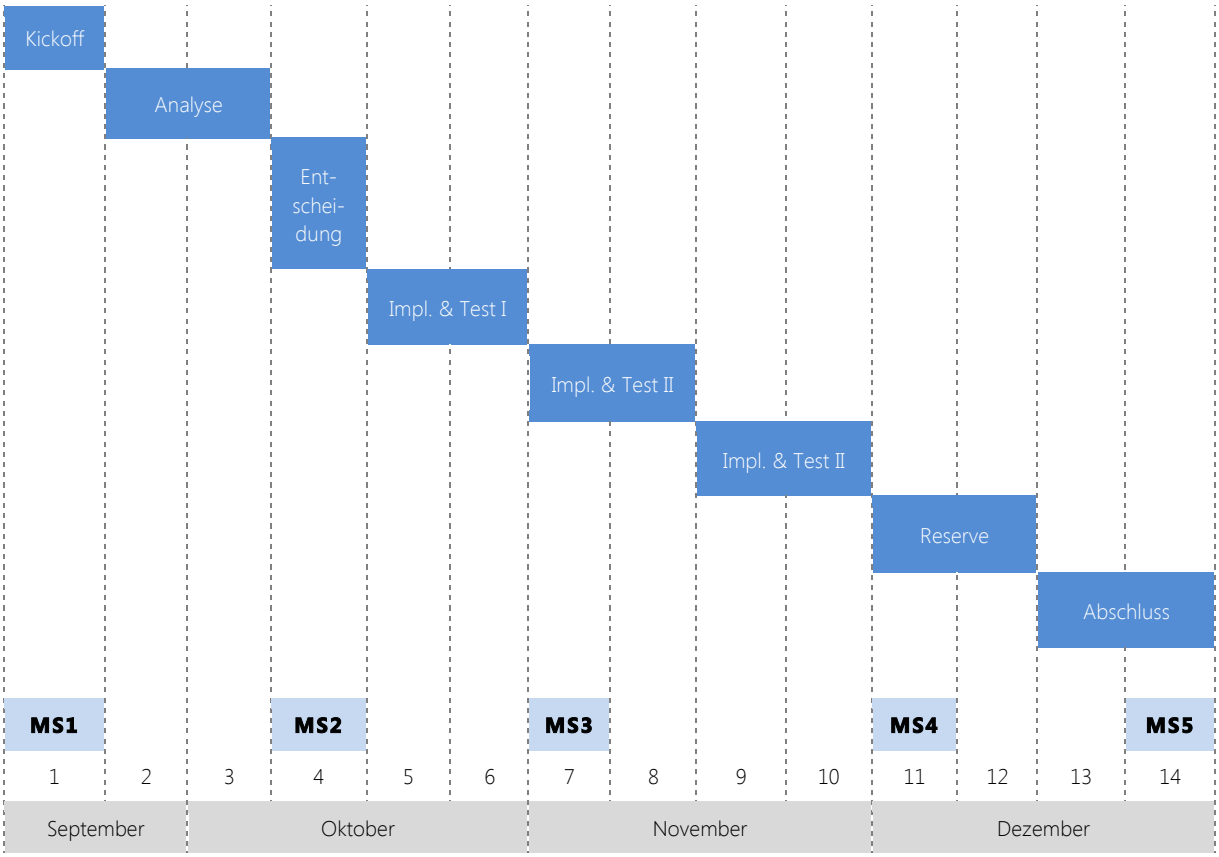
Bei einer solchen Aufgabenstellung ist es sehr wichtig analytisch vorzugehen, d.h. die Schritte müssen klar definiert und ausgeführt werden. Deshalb mussten wir uns zuerst mit dem Umfang der Script-Sprache beschäftigen und diese analysieren. Wie in jeder Sprache gibt es diverse Regeln, welche eingehalten werden müssen. Diese haben wir durchdacht und festgehalten in einer sogenannten Grammatik, die genaue Regeln für die syntaktische Korrektheit des Codes definiert. Die Grammatik wird mit unserer Software in verschiedenen Schritten analysiert. Zu Beginn gibt es eine Zeichenerkennung, aus welcher ein Symbol-Strom entsteht. Dieser Symbol-Strom beinhaltet unter anderem Wörter und Schlagwörter sogenannte Keywords der Script-Sprache und Operanden. Der letzte Schritt kümmert sich um die „Rechtschreibung" der Script-Sprache, d.h. er schaut ob das geschriebene auch im Sinne der Script-Sprache ist.

Resultate

Schlussendlich haben wir den Editor so entwickelt, dass der Kunde seine Funktionalität problemlos, auch ohne Vorkenntnisse oder Erfahrungen mit dem Editor, entwickeln kann. Durch intelligente und automatische Vervollständigung und Überprüfung der geschriebenen Scripts, direkt während der Eingabe, wird der Kunde beim Erstellen seiner gewünschten Scripts gut geführt.

Planung

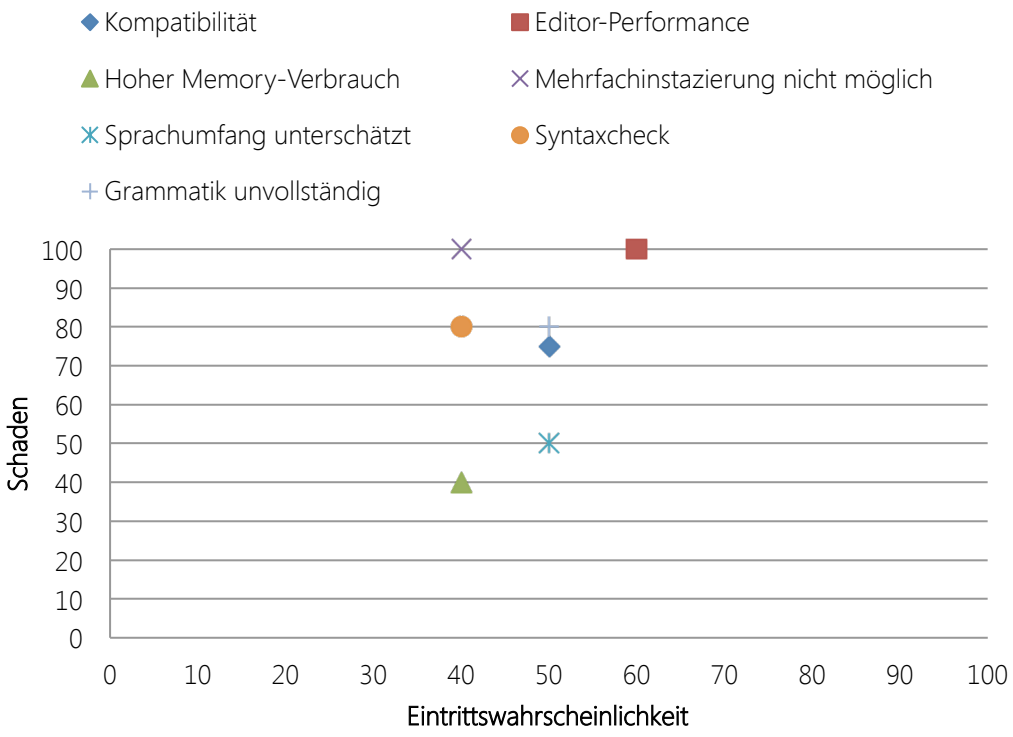
PHASE	WOCHE	DAUER
Kickoff	16.09.2013 – 23.09.2013	1 Woche
Analyse	23.09.2013 – 07.10.2013	2 Wochen
Entscheidung	07.10.2013 – 14.10.2013	1 Woche
Implementation & Test I	14.10.2013 – 28.10.2013	2 Wochen
Implementation & Test II	28.10.2013 – 11.11.2013	2 Wochen
Implementation & Test III	11.11.2013 – 25.11.2013	2 Wochen
Reserve	25.11.2013 – 09.12.2013	2 Wochen
Abschluss	09.12.2013 – 20.12.2013	2 Wochen



MEILENSTEIN	RESULTATE	TERMIN
MS1 Kickoff	Konkrete Aufgabenstellung	19.09.2013
MS2 Analyse + Entscheidung	Sprachmodell Evaluierung 3rd Party Control Wireframes Featureliste	10.10.2013
MS3 Zwischenstand Impl. & Test	Prototyp	31.10.2013
MS4 Ende Implementation	Guided-Script-Editor	28.11.2013
MS5 Abschluss Projekt	Abgabe Bericht	20.12.2013

Risiken

RISIKO	BESCHREIBUNG	VORBEUGUNG
Kompatibilität	Editor lässt sich nicht mit neuster .NET Version kompilieren oder nicht auf 64 Bit kompilieren.	Fortlaufend darauf achten, vor allem bei Verwendung von 3rd-Party-Software
Editor-Performance	Syntax-Highlighting verlangsamt Editor oder Auto-Completion kommt spürbar verzögert.	Fortlaufend darauf achten und falls nötig optimieren.
Hoher Memory-Verbrauch	Pro Editor Instanz wird zu viel Memory-Speicher benötigt. Daher problematisch, wenn mehrere Instanzen existieren.	Memory-Verbrauch im Auge behalten.
Mehrfachinstanziierung nicht möglich	Gemeinsame Ressourcen oder ähnliches verhindern, isolierte Verwendung von mehreren Editor-Instanzen.	Fortlaufend mit mehreren Instanzen testen.
Sprachumfang unterschätzt	Sprachumfang der Galileo Script-Sprache unterschätzt.	In die Tiefe anstatt Breite entwickeln (Reine Definition von Funktionen ist dann Fleissarbeit).
Syntax-Check nicht möglich	Allgemeiner Syntax-Check lässt sich aufgrund von Sprachgegebenheiten nicht umsetzen.	Frühe Analyse der Sprache.
Grammatik unvollständig	Die Grammatik welche aufgestellt wird, deckt den Sprachumfang nicht ab.	Viele verschiedene Test-Scripts erstellen, welche den Sprachumfang abdecken und mit der Grammatik überprüfen lassen.



Anforderungsanalyse

Evaluierung Einsatz 3rd-Party-Control

Da man das Rad nicht immer neu erfinden sollte, haben wir uns bestehende Code-Editoren angeschaut, um sie allenfalls in unsere Studienarbeit zu integrieren.

	SHARPDEVELOP	DEVEXPRESS	SCINTILLANET	MONODEVELOP
				
Beschreibung	SharpDevelop ist eine freie Entwicklungsumgebung für .NET und enthält auch ein Editor-Control	Die Control-Library von DevExpress bietet ein RichTextEditor-Control	ScintillaNET ist ein managed Wrapper für die Scintilla Komponente (http://scintilla.org)	Ist eine Entwicklungsumgebung für den .NET Port „Mono“
Komplexität/ Umfang/ Features	Riesig (12000 Files). Unterstützt so ziemlich alles wie Code-Completion, Snippets, Zeilennummerierung, Auto-Completion etc.	Basic Features wie Syntax-Highlighting Zeilennummierung möglich. Jedoch keine Code-Completion	Unterstützt so ziemlich alles wie Code-Completion, Snippets, Zeilennummerierung, Auto-Completion etc.	Ähnlich zu SharpDevelop
Einarbeitung	Sehr gross	Mässig	Gross	Sehr gross
Lizenz	GNU LGPL	Commercial	Custom	GNU LGPLv2
Aktualität	Sehr gut	Sehr gut	August 2012	Sehr gut
Kompatibilität	Ab .NET 2.0	-	Eingeschränkt da Verwendung von System.Design.dll	-
Schlussfolgerung	Durch den sehr grossen Umfang eher ungeeignet für eine Verwendung. Zuviel „Ballast“ der mitgenommen werden würde. Ausserdem Einarbeitungszeit zu gross für 14 Wochen.	Da dieses Control noch keine Code-Completion erlaubt, müssten wir dieses zusätzlich implementieren.	Hier müssten Scintilla und der Wrapper Scintilla-NET in das Galileo übernommen werden. Ausserdem ist es mit der Lizenz kritisch, da Galileo nicht „free software“ ist.	Hier auch wieder der Umfang und Ballast und die Anpassung sowie Einarbeitungszeit an unsere Gegebenheiten zu gross.

Wir waren nach der Evaluierung nicht sehr begeistert von der Verwendung eines 3rd Party Controls. Die Einarbeitungszeit und der Ballast bei der Verwendung erscheint für die 14 Wochen zu gross. Wir wollten daher einen „Lightweight-Editor“ auf Basis der Windows Forms RichTextBox erstellen. Der Kunde konnte uns jedoch überzeugen zumindest das RichEdit-Control von DevExpress zu verwenden, da im neuen Galileo bereits DevExpress verwendet wird. Damit würde sich der Editor bereits optisch besser integrieren. Als positiver Nebeneffekt unterstützt DevExpress bereits performantes Syntax-Highlighting. Dies wäre mit dem Windows Forms Control nur bei Einsatz einer unmanaged DLL möglich.

Anforderungen

Basisfunktionalität

Die Basisfunktionalität gemäss Aufgabenstellung

FEATURE	BESCHREIBUNG
Zeilennummern	Anzeigen der Zeilennummer.
Copy&Paste	Es muss möglich sein Code aus anderen Anwendungen in den Editor einzufügen. Der Text muss als Plain-Text interpretiert werden.
Drag&Drop	Es muss möglich sein Code aus anderen Anwendungen mittels Drag&Drop in den Editor einzufügen (Plain-Text). Der Text muss als Plain-Text interpretiert werden.
Undo/Redo	Die letzten Aktionen rückgängig machen bzw. wiederholen.
Suchfunktion	Es muss eine einfache Suchfunktion (Suchen und Ersetzen) vorhanden sein.
Funktionsauswahl	Die definierten Funktionen können angezeigt und im Editor eingefügt werden.
Script-Check	Der Script-Code im Editor muss anhand der Grammatik überprüft werden können.
Kommentare	Es muss möglich sein den Code mittels Kommentaren, welche nicht interpretiert werden, zu dokumentieren.
Script-Performance	Der Script-Editor muss ein Script mit 1000 Zeilen ohne messbare Verzögerung (< 200ms) überprüfen können. Ausserdem darf das Tippen im Editor durch geladenes Script (bis 1000 Zeilen) nicht spürbar (< 200ms) langsamer werden.

Erweiterte Funktionalität

Die erweiterten Features gemäss optionalem Teil der Aufgabenstellung und Wünsche des Kunden.

FEATURE	BESCHREIBUNG
Syntax-Highlighting	Die einzelnen Textteile werden je nach ihrer Bedeutung entsprechend eingefärbt.
Auto-Completion	Dropdown zur Auto-Vervollständigung, Vorschläge, Templates (Snippets), direkt während dem Tippen.
Auto-Format	Der Code kann bei Bedarf automatisch eingerückt werden. If/else/endif jeweils auf gleicher Zeile, die Statements innerhalb eingerückt (1 Tab).
Live-Script-Check	Das Script automatisch (z.B. nach x Sekunden oder nach jedem Buchstaben) überprüfen und das Resultat anzeigen.
Fehlermarkierung im Text	Fehlerhafte Textteile direkt im Text markieren (z.B. unterstreichen).
Gute Fehlermeldungen	Gute Fehlermeldungen für den Benutzer, damit er genau weiss was das Problem ist. (z.B. Semikolon auf Zeile 4 vergessen). Zudem Möglichkeit direkt an die Fehlerposition zu springen.

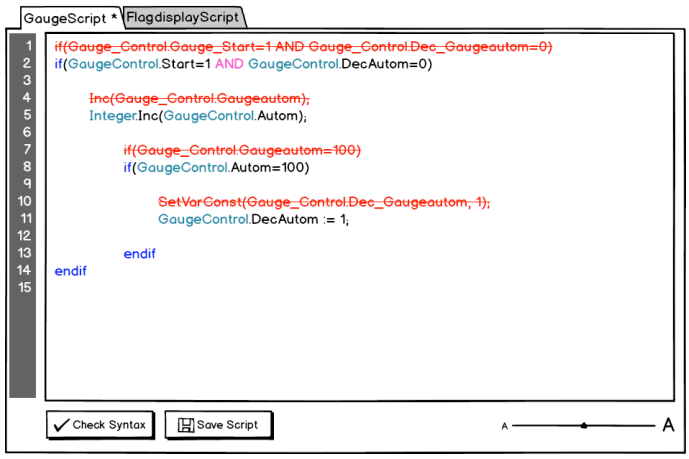
GUI-Entwurf

BESCHREIBUNG

WIREFRAME

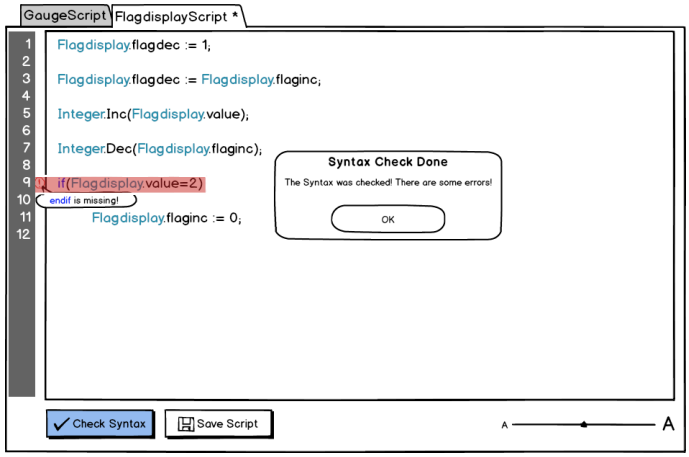
Editor-Control

Allgemeiner Entwurf des Editor-Controls mit Code-Textbox, Zeilennummern und Syntax-Highlighting. Wir haben gleich von Anfang an berücksichtigt, dass der Editor parallel in mehreren Tabs geöffnet werden kann. Rot durchgestrichen ist die alte Syntax, die wir zusätzlich für unsere Studienarbeit optimiert haben. In diesem Wireframe haben wir auch gezeigt, dass der Code eingerückt werden kann. Dies kann entweder automatisch oder manuell erfolgen.



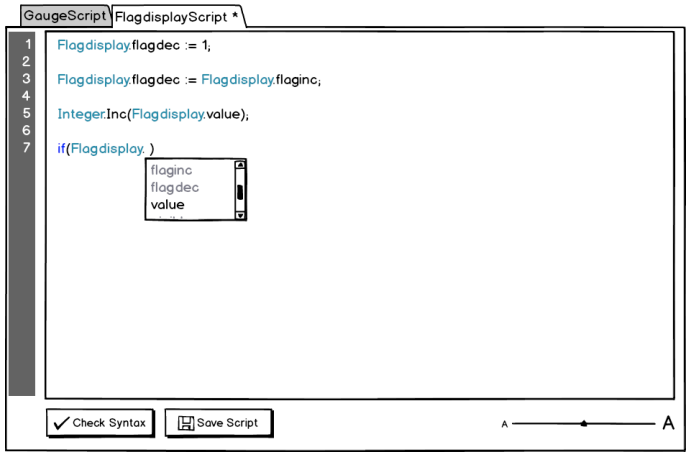
Syntax-Check

Entwurf der Umsetzung des Script-Checks. Unser Ziel ist es, Syntax-Fehler direkt im Text anzuzeigen bzw. den fehlerhaften Teil zu markieren. Dadurch wird die Fehlerbehebung erleichtert. Der erste Prototyp soll den Syntax-Check auf Knopfdruck erlauben. Später wäre eine automatische Prüfung während der Texteingabe gewünscht, sofern die Performance dies erlaubt.



Auto-Completion

Entwurf der Auto-Completion. Die Auto-Completion soll aus einer ListBox bestehen, die während der Texteingabe an der aktuellen Textposition angezeigt wird. Die Vorschläge können mittels Pfeiltasten selektiert werden. Mittels der Tab- und der Enter-Taste wird ein Vorschlag übernommen und in der aktuellen Textposition eingefügt. Da die Entwicklung einer guten Auto-Completion sehr aufwendig werden kann, muss hier während der Entwicklungsphase ein guter Mittelweg zwischen guten Vorschlägen und Aufwand gefunden werden.



Sprachumfang und Sprachdefinition

Ausgangslage

Der Sprachumfang der Script-Sprache ist historisch gewachsen und somit auch die Übersicht und Komplexität. In einem ersten Schritt ging es darum die unterschiedlichen Funktionen aus dem alten Script-Editor zu analysieren und für den neuen Script-Editor zusammenzufassen und zu gruppieren. Grundsätzlich gab es folgende Gruppen, welche unterschiedlich viele Funktionen beinhalteten.

Alte Sprachdefinition

GRUPPE	BEZEICHNUNG
Standard	Standardfunktionen
Intern	Interne Funktionen
Recipe	Rezeptfunktionen
Tag	Variablenfunktionen
SpecFunc	Spezialfunktionen
Time	Zeitfunktionen
PlcText	Textfunktionen
Password	Benutzerveraltungsfunktionen

Neue Sprachdefinition

Für den neuen Script-Editor sollte es nur noch drei Gruppen geben und zwar folgende: *Math, Runtime und String.*
Wir konzentrierten uns primär auf die Funktionen, welche für Scripte, Tags, Rezepte oder Masken definiert sind und teilten diese jeweils in die neuen Gruppen ein. Die Funktionen sollten modulbasiert aufgerufen werden, d.h. die Gruppen beinhalten die Funktionen. So sieht ein Aufruf modulbasiert aus:
Gruppe.Funktion([Parameter]);
String.Append(myName, yourName);
Weiter sollten auch objektbasierte Aufrufe ermöglicht werden, d.h. Funktionen sind nicht Module der Gruppe, sondern sind vom Objekt abhängig. So sieht ein Aufruf objektbasiert aus:
Tag.Funktion([Parameter]);
myRecipe.SaveActual();

Ein Nachteil, welchen wir beheben konnten, war, dass in der alten Sprachdefinition teilweise sehr umständliche Funktionsaufrufe verwendet werden um z.B. einem Tag einen Wert zuzuweisen. Oder auch einem Tag einen anderen Tag zuzuweisen. Statt einer Zuweisung mit einem Zuweisungsoperator (z.B. „=“) bekannt aus anderen Script-Sprachen, mussten jeweils unterschiedliche Funktionen aufgerufen werden z.B. SetVar(Tag1,Tag2); anstatt Tag1 = Tag2; oder auch SetVarConst(Tag1,Val);

Vergleich Neue und Alte Sprachdefinition

Wie erwähnt ist der Sprachumfang relativ gross, deshalb zeigen wir hier nur ein paar Beispiele, wie gewisse Funktionen in der alten Sprachdefinition definiert waren und wie sie nun in der neuen definiert sind. Im Anhang befindet sich die gesamte Übersicht der Sprachdefinition, welche mit dem Kunden gereviewt wurde und auch von ihm akzeptiert wurde.
Bemerkung zur Tabelle: Die Bezeichnungen „M / O" bedeuten M = Modulbasiert; O = Objektbasiert. Die Zuweisung ist weder noch.

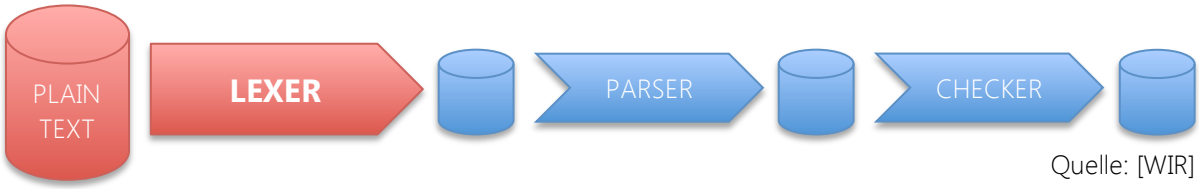
	BESCHREIBUNG	ALT	NEU
-	Einem Tag einen konstanten Wert zuweisen.	Group.Tag.SetVarConst(tag1, 5);	tag1 = 5;
-	Einem Tag einen anderen Tag zuweisen.	Group.Tag.SetVar(tag1, tag2); Group.PlcText.Assign(string1, string2);	tag1 = tag2; string1 = string2;
M	In-/Dekrementieren eines Tags	Group.Tag.Inc(tag1); Group.Tag.Dec(tag1);	Math.Inc(tag1); Math.Dec(tag1);
M	Zwei Strings anhängen	Group.PlcText.Append(stri1, stri2);	String.Append(stri1, stri2);
M	Zwei Strings vergleichen	Group.PlcText.Compare(stri1, stri2);	String.Compare(stri1, stri2);
M	Script ausführen	Group.Standard.Script(scriptA);	Runtime.RunScript(scriptA);
M	Maske wechseln	Group.Standard.MaskChange(mask.xmsk);	Runtime.ChangeMask(maskA);
O	Funktion ist beim Aufruf eines neuen Rezepteintrages erfüllt	Group.Recipe.OnRecipeLoad(recipe1);	recipe1.OnLoad();
O	Rezepteintrag laden	Group.Recipe.LoadRecipeEntry(recipe2, 3);	recipe2.LoadEntry(3);

Architektur

Grammatik der neuen Script-Sprache

Programm	=	StatementSequence
StatementSequence	=	(Statement)*
Statement	=	Assignment | FunctionCallStatement | If-Statement
FunctionCallStatement	=	FunctionCall ";"
Assignment	=	Designator "=" Expression ";"
Designator	=	Identifier (["."Identifier])*
Identifier	=	Character (Character | Number)*
Expression	=	AndExpression ("or" AndExpression)*
AndExpression	=	ComparisonExpression ("and" ComparisonExpression)*
ComparisonExpression	=	SimpleExpression (Comparison SimpleExpression)*
SimpleExpression	=	Term (("+" | "-") Term)*
Term	=	Factor (("*" | "/") Factor)*
Factor	=	Literal | Designator | [("not" | "-") Factor] | "(" Expression ")" | FunctionCall
Expressionlist	=	Expression ("," Expression)*
FunctionCall	=	Designator "(" Expressionlist ")"
If-Statement	=	"if" "(" Expression ")" StatementSequence (Else-Statement) "endif"
Else-Statement	=	"else" StatementSequence
Comparison	=	"==" | ">" | "<" | ">=" | "<=" | "<>"
Literal	=	IntegerLiteral | FloatLiteral | BoolLiteral
IntegerLiteral	=	Number
FloatLiteral	=	Number "." Number
BoolLiteral	=	"true" | "false"
Number	=	Digit (Digit)*
Digit	=	0 ... 9
Character	=	"a" ... "z" | "A" ... "Z"

Lexer



Quelle: [WIR]

> Der Übersetzungsprozess vom Plain-Text bis hin zum Compilat, beginnt grundsätzlich bei der lexikalischen Analyse oder auch Lexer genannt. Dieser erkennt die, in der Grammatik definierten, Symbole wie z.B. Identifier, Tokens (Operatoren), Zahlen und Kommentare. [WIR]

Beim Lexer erhalten wir den Plain-Text und gehen dann Zeichen für Zeichen durch. Wir haben unseren Lexer so aufgebaut, dass er die Blanks auslässt und diese nicht weiter verarbeitet. Weiter merken wir uns jeweils die Position der Zeichen, bzw. der zusammengesetzten Symbole. Somit können wir diese dafür verwenden, um die Position anzuzeigen, wenn es zu einem Fehler kommt. Ebenfalls wird die Position für das Syntax-Highlighting verwendet.
Der Lexer erkennt auch bereits schon die von uns definierten Key-Words, so z.B. IF, ELSE, ENDIF, AND, OR, etc. Das Ergebnis, welches aus dem Lexer folgt ist ein Symbol-Strom. Dieser wird nun vom Parser weiterverarbeitet.

Parser

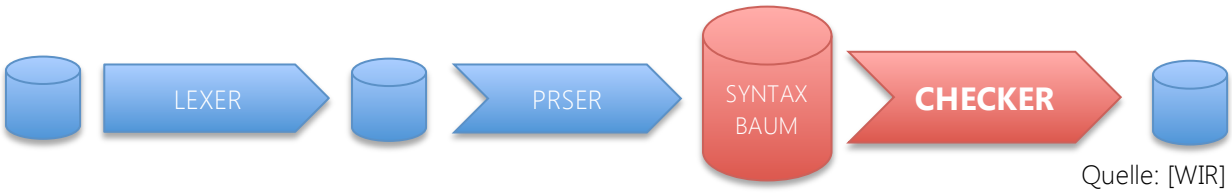
Quelle: [WIR]

> ..der eigentliche Parser [...] kümmert sich um die Grammatik der Eingabe, führt eine syntaktische Überprüfung der Eingangsdaten durch und erstellt in der Regel aus den Daten einen Ableitungsbaum ([...] Parse-Tree). Dieser wird danach zur Weiterverarbeitung der Daten verwendet; typische Anwendungen sind die semantische Analyse [...]. [WPAR]

Der Parser erhält als Input den Symbol-Strom, welcher Symbol um Symbol abgearbeitet wird (für definierte Sprache reicht: „one symbol look ahead“). Der Parser wurde als Top-Down-Parser implementiert (sogenannter LL1 Parser, d.h. linksrekursiv absteigend mit „one symbol look ahead“).
Er übersetzt die erhaltenen Symbole mit den in der Grammatik definierten Regeln, zu einem abstrakten Syntax-Baum. So wird z.B. aus zwei Identifiers, die durch einen Punkt getrennt sind wie tag1.tag2 ein Designator.
Der Parser geht immer soweit, bis er auf einen Fehler stösst. Wird ein Fehler detektiert so bricht das Parsen ab und es wird dem Benutzer ein Fehler angezeigt, mit Position und der Fehlermeldung. Beim Parser werden die Positionen kumuliert, dadurch können wir z.B. bei

einem fehlerhaften Designator den gesamten Designator kennzeichnen, da wir die Positionen von den einzelnen Symbolen aus dem Lexer kennen.

Checker



Quelle: [WIR]

> Die semantische Analyse überprüft die [...] Semantik, also über die syntaktische Analyse hinausgehende Bedingungen an das Programm. Zum Beispiel muss eine Variable in der Regel deklariert worden sein, bevor sie verwendet wird, und Zuweisungen müssen mit kompatiblen (verträglichen) Datentypen erfolgen. [...] Dabei werden die Knoten des vom Parser generierten Syntaxbaums mit Attributen versehen, die Informationen enthalten. [..] Die Ausgabe der semantischen Analyse nennt man dann [...] Syntaxbaum (Syntax-Tree). [WCHE]

Für den Checker haben wir folgende Semantik-Regeln aufgestellt, welche wir auch so umgesetzt haben:

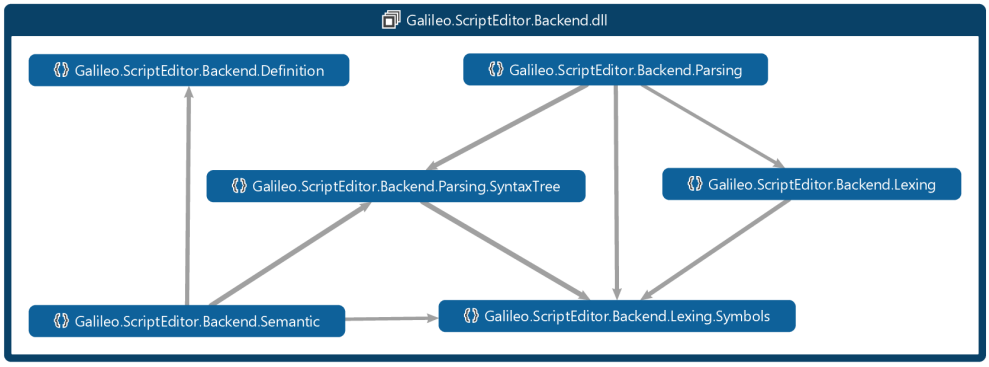
Designator	
	Die Identifier müssen definiert sein bevor sie verwendet werden. Beispiele eines Designators: Gruppe.Funktion (Modulbasierte Funktionen) Verschatelte Tags (z.B. currentRoom.cooler.temperature..) Tag.Funktion (Objektbasierte Funktionen)
Assignment	
	Der Designator und die Expression müssen beim Assignment denselben Typ haben. Designator.Type = Expression.Type
Expression	
	BinaryExpression
	Operatoren können nur auf Operanden vom gleichen Typ angewendet werden.
	Die Operatoren „OR“ und „AND“ dürfen nur mit booleschen Expressions verwendet werden.
	UnaryExpression
	Expressions mit dem Vorzeichen „NOT“ müssen vom Typ ‚boolean‘ sein.
	Expressions mit dem Vorzeichen „-“ müssen vom Typ ‚word‘ ‚dword‘ oder ‚float‘ sein.
If-Statement	
	Die Expression im If-Statement muss einen booleschen Wert zurückliefern.
FunctionCallStatement (ProcedureCall)	
	Ein ProcedureCall darf keinen Rückgabewert haben, sondern muss “void” sein.
FunctionCall	
	Die Expressions in der Expressionlist müssen in der korrekten Reihenfolge und mit korrektem Typ angegeben werden.

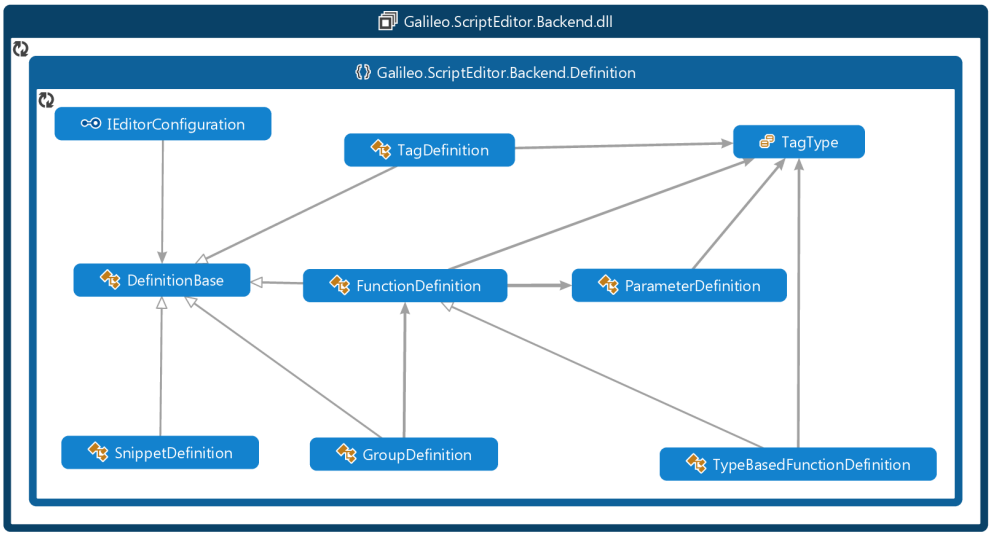
Wir haben für die Datentypen-Überprüfung ein ResultType definiert, welcher jeweils aus der Definition (siehe Kapitel Klassen-Struktur, Backend) gelesen wird und gegen die o.g. Regeln geprüft wird. Der Checker wird ebenfalls beim Auftreten eines Fehlers unterbrochen und dem Benutzer wird eine Fehlermeldung angezeigt.

Klassen-Struktur

Backend

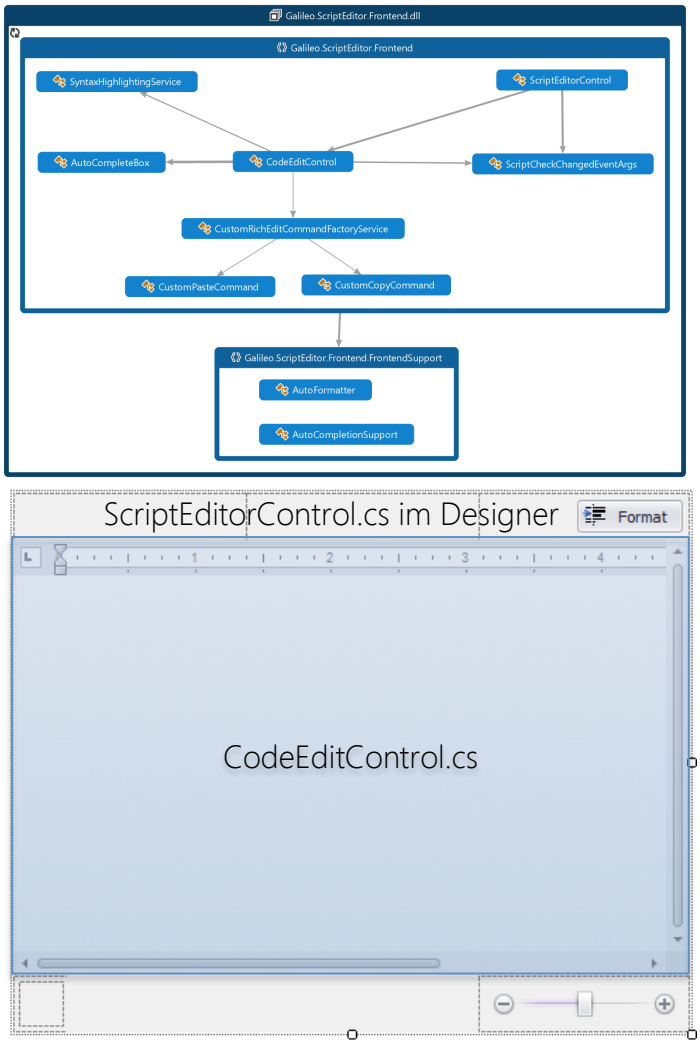
Das Backend wird in eine eigene Assembly ausgelagert. Dies stellt sicher, dass keine Referenzen auf den Frontend-Teil vorhanden sind, was zu einer zyklischen Abhängigkeit zwischen Frontend und Backend führen würde.
In der Backend-Assembly befinden sich die Compiler-Komponenten: Lexer, Parser, Checker. Zusätzlich gibt es ein Package für die definierten Lexer-Symbole und die Syntax-Tree Elemente für das Parsing. Ausserdem gibt es Klassen für die Definierung von Tags, Funktionen, Gruppen welche für die Editor-Konfiguration benötigt werden. Dem Script-Editor wird beim Start eine Liste von Definitionen (Klasse DefinitionBase) mitgegeben. Diese benötigt der Script-Editor um Vorschläge liefern zu können.





Frontend

Das Frontend ist eine CustomControl Library und enthält Referenzen auf die DevExpress Libraries. Das CodeEditControl (Abbildung unten, blau markiert) ist ein Control abgeleitet von DevExpress RichEditControl. Es ist die eigentliche Editor-TextBox inklusive Zeilennummer und Syntax-Highlighting. Auch wird die Auto-Completion-Box direkt von diesem Control angezeigt und an die aktuelle Textposition positioniert. Um zu verhindern, dass Text mit Formatierung in den Editor eingefügt werden kann müssen zusätzlich eigene Commands (CustomPasteCommand, CustomCopyCommand) erstellt werden und mittels CustomCommandFactory dem Control injektiert werden. Das Package Frontend-Support beinhaltet Hilfsklassen die keine direkten Referenzen auf DevExpress oder GUI-Klassen besitzen, jedoch nur vom Frontend benutzt werden. Einerseits ist dies der AutoFormatter, welcher den Code einrückt und die Klasse AutoCompletionSupport, welche die Vorschläge für die Auto-Completion liefert. Zu guter Letzt das eigentliche Script-Editor-Control ist die Klasse ScriptEditorControl, welches ein WinForms CustomControl ist. Dies kann dann vom Kunden im Galileo benutzt werden. Über das Property „ScriptCode" kann der Script-Code gesetzt werden (Plain-Text) und mit dem boolschen Property „ValidScriptCode" kann die Gültigkeit des Scripts abgefragt werden. In diesem Control werden auch allfällige Parse oder Checker Fehler angezeigt.



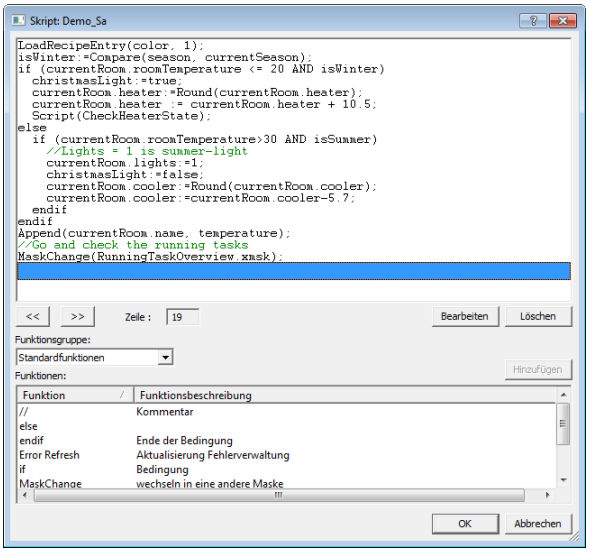
Lösung / Resultat

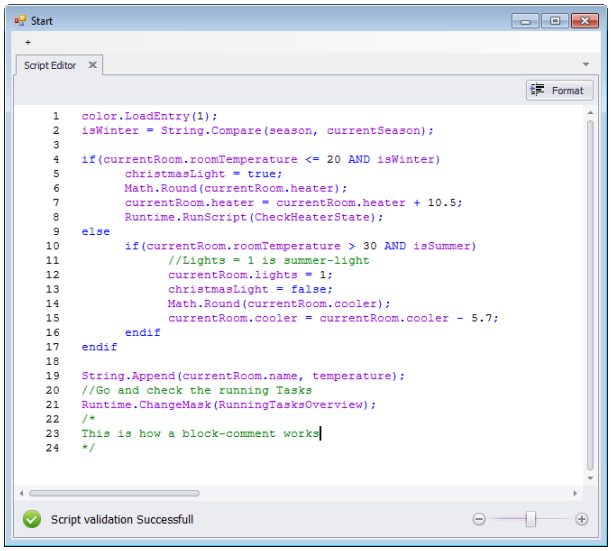
Vergleich Ausgangslage / Resultat

	ALT	NEU
Geschwindigkeit Effizienz	War sehr langsam, da alle Funktionen mit der Maus angeklickt werden mussten und teilweise sehr versteckt waren.	Durch die Auto-Completion sind die Funktionen einfach zu finden und die Arbeit dadurch sehr effizient. Auto-Completion erscheint automatisch oder wird erzwungen mit CTRL + SPACE.
Lernfördernd	Sehr gewöhnungsbedürftig, da die Funktionen verschachtelt in den Gruppen stecken. Die Gruppen sind historisch gewachsen und deshalb teilweise sehr unübersichtlich. Gruppe mit Spezialfunktionen unnötig.	Auf drei Gruppen beschränkt und somit auch einfacher die gewünschten Funktionen zu finden. Selbsterklärend.
Enabling / Disabling von Aktionen	Nicht konsequent durchgezogen, z.B. können leere Zeilen gelöscht werden.	Es gibt nur einen Button (Auto-Format), der kann immer betätigt werden.
Copy&Paste Drag&Drop	Wurde vom alten Editor nicht unterstützt.	Problemlos beim neuen Editor. Das Script wird auch gleich geprüft (Lexer, Parser, Checker) und das Syntax-Highlighting wird auch gleich gemacht.
Suchen	Wurde vom alten Editor nicht unterstützt.	Wird unterstützt. CTRL + F
Einfügen von Zeilen	Zeilen können nur ans Ende des Scripts hinzugefügt werden.	Beliebiges Einfügen von neuen Zeilen.
Auto-Formatierung	Mühsam Zeile um Zeile.	Per Knopf-Druck und sogar bei unvollständigen Scripts.
Fehlerbehandlung	Allgemeine Fehlermeldung nur beim Einfügen von neuen Zeilen. Auf den ersten Blick teilweise nicht verständlich.	Während der Eingabe wird fortlaufend das Script geprüft und Fehlermeldungen mit Positionen angezeigt. Der Text wird je nach Fehler blau (Semantic-Error) oder rot (Parsing-Error) wellenförmig unterstrichen.
Syntax-Highlighting	Das Script hat kein Syntax- Highlighting sondern ist nurschwarz/weiss	Keywords, Tags, Funktionen, etc. werden markiert.
Mehrfachinstanziierung	Jeweils nur eine Editor-Instanz möglich.	Es können problemlos mehrere Instanzen des Editors gemacht werden.
Undo / Redo	Wird nicht unterstützt.	Wird unterstützt Undo: CTRL + Z Redo: CTRL + Y

	ALT	NEU
(Block-)Kommentar	Kommentare können, wie die Funktionen auch, als Zeile eingefügt werden. Block-Kommentare, d.h. Kommentare über mehrere Zeilen sind nicht möglich.	Unterstützt Zeilenweise- und Block-Kommentare. //Comment /* Block- Comment */
Zoom	Keine Zoomfunktion enthalten, d.h. der Text kann nicht vergrössert oder verkleinert werden.	Der Slider ermöglicht das vergrössern oder verkleinern der Texteingabe pro Editor-Instanz.
Zeilennummern	Zeilennummern sind in einer Textbox ersichtlich, jedoch muss eine Zeile markiert sein, um ihre Zeilennummer zu sehen.	Zeilennummern pro Zeile auf der linken Seite ersichtlich.
Snippets	Ein If-Snippet gibt es nicht. Es muss ein If eingefügt werden, dann alle Funktionen oder bzw. die Sequenz eigegeben werden und dann noch ein endif hinzugefügt werden.	Mit der Auto-Completion besteht die Möglichkeit ein If-Snippet oder If-else-Snippe einzufügen, welches bereits die korrekte Klammerung macht und ein endif (und else) einfügt. Weiter wird der Cursor in die Bedingung gesetzt, damit gleich weiter geschrieben werden kann.

Auf den folgenden Screenshots ist der Quervergleich des alten und neuen Script-Editors. Bis auf den Block-Kommentar ist das Script 1:1 dasselbe.
Im unteren Bereich des alten Script-Editors ist die Gruppen-, bzw. Funktionsauswahl ersichtlich, welche beim neuen durch die Auto-Completion wegfällt und dem Benutzer die Bedienung vereinfacht.





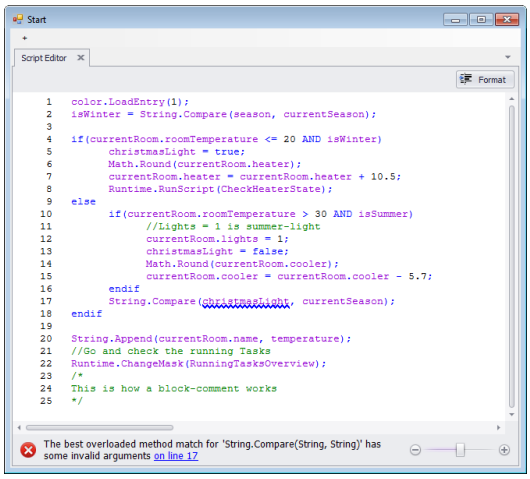
Umgesetzte Features

FEATURE

SCREENSHOT

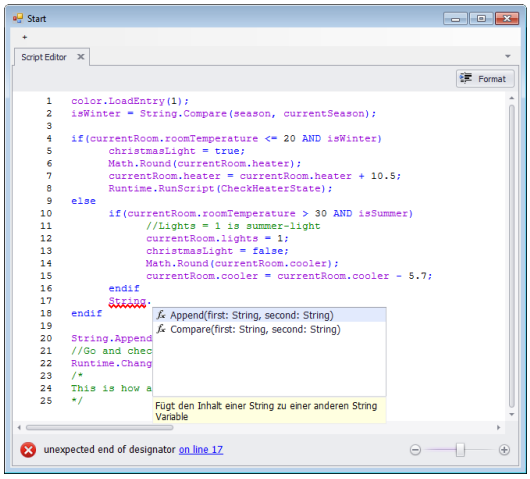
Syntax-Highlighting

Klassisches Syntax-Highlighting direkt während der Eingabe. Dies erlaubt die schnelle optische Unterscheidung von Tags, Keywords (z.B. if, else, and, or, not), Kommentaren oder ungültigen Zeichen wie z.B. ‚öääüç?§'.



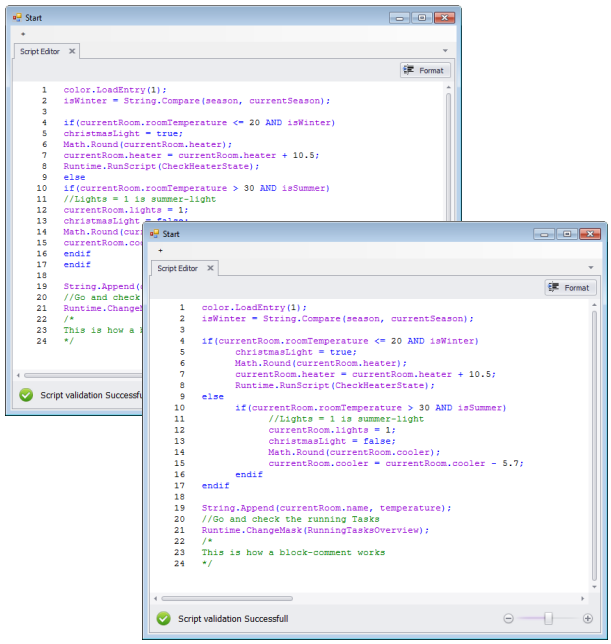
Live-Script-Check

Der Script-Check wird fortlaufend, d.h. direkt während der Eingabe ausgeführt. Der Script-Check beinhaltet Parse (rote Wellenlinie) und Checker (blaue Wellenlinie) Fehler. Zudem wird die Fehlermeldung angezeigt mit Möglichkeit direkt zur fehlerhaften Position zu springen. Auch beim schnellen Tippen oder bei einer Zeilenanzahl von 1000 Zeilen merkt man keine Verzögerung durch den „Live-Script-Check" .



Auto-Completion

Das Auto-Completion Feature trägt sehr zur Produktivität bei der Erstellung eines Scriptes bei. Einerseits kann sich der Benutzer Vorschläge anzeigen lassen (Welche Tags bzw. ChildrenTags, welche Funktionen gibt es?), andererseits vervollständigt es die Eingaben, damit die Funktionsnamen oder Tagnamen nicht immer ausgeschrieben werden müssen. Bei der Vervollständigung kann der Benutzer auch den hinteren Teil des Namens eingeben. Bei Tags werden auch gleich die Datentypen angezeigt und bei Funktionsparametern werden nur vom Datentyp her passende Tags vorgeschlagen. Desweiteren gibt es die Möglichkeit vorgefertigte Code-Teile (sogenannte Snippets) einzufügen. Dies erleichtert repetitive Aufgaben wie z.B. das Schreiben eines if/else/endif-Blockes.



Code einrücken

Um den Code lesbarer zu machen besteht die Möglichkeit eine automatische Code-Formatierung vorzunehmen. Dies führt vor allem bei verschachtelten „If's" zu massiv besserer Lesbarkeit. Die Formatierung kann auch bei unvollständigen Scripts gemacht werden, so wird gleich ein fehlendes „endif" erkannt.

Testing

Beim Testing haben wir uns ganz auf den Backend-Teil konzentriert und diesen mit Unit-Tests getestet. Wir haben die drei Teile Lexer/Parser/Checker jeweils wie folgt getestet. Beim Lexer haben wir alle einzelnen Symbole getestet und ebenfalls einen „Massentest" gemacht, in welchem wir in beliebiger Reihenfolge die einzelnen Symbole auflisteten. Dort haben wir hauptsächlich geschaut, dass die Positionen und Typen stimmen und ungültige Symbole auch als ungültige Symbole erkannt werden. Beim Parser sind wir ebenfalls alle Grammatik-Regeln durchgegangen und haben dort jeweils geschaut, ob der Output gemäss der vordefinierten Referenz übereinstimmt. Beim semantischen Test haben wir jeweils gegen die Regeln getestet, welche wir aufgestellt haben (siehe Kapitel Architektur-Checker). Weiterführend haben wir auch versucht Kombinationen zu testen, indem wir auch längere Scripts erstellt haben.
Trotz hoher Test-Abdeckung konnte beim Lexer/Parser/Checker keine 100% Testabdeckung erreichen, weil beispielsweise bewusst nicht erreichbare Fälle im defensiven Programmierstil mit Exceptions geschützt sind. Die Testabdeckung macht zudem nur eine Aussage über die Abdeckung der statischen Programmpfade und nicht über die dynamischen Wertekombinationen von Variablen, die das Verhalten steuern. Wir prüfen vor allem den vordefinierten Output von Beispiel-Scripts mit der definierten Referenz. Insgesamt haben wir 87 Testfälle mit einer Test-Abdeckung von 98.89%.

Hierarchy	Not Covered (Blocks)	Covered (Blocks)	Covered (% Blocks)	Not Covered (% Blocks)
SA@SA-PC 2013-12-13 16:02:36	22	1344	98.39%	1.61%
Galileo.ScriptEditor.Backend.dll	22	1344	98.39%	1.61%
{} Galileo.ScriptEditor.Backend.Definition	4	129	96.99%	3.01%
{} Galileo.ScriptEditor.Backend.Lexing	4	344	98.85%	1.15%
{} Galileo.ScriptEditor.Backend.Lexing.Symbols	1	86	98.85%	1.15%
{} Galileo.ScriptEditor.Backend.Parsing	6	307	98.08%	1.92%
{} Galileo.ScriptEditor.Backend.Parsing.SyntaxTree	0	184	100.00%	0.00%
{} Galileo.ScriptEditor.Backend.Semantic	7	294	97.67%	2.33%

Weitere Ideen / Open-Issues

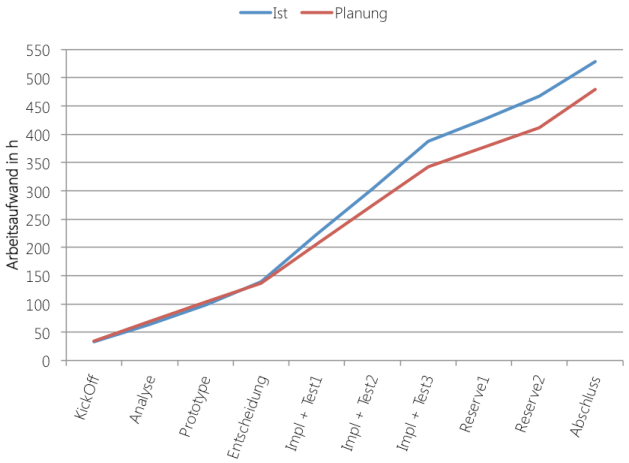
IDEE	BESCHREIBUNG
Undo Ausbau	Anstatt den letzten Schritt ruckgängig machen, zusätzlich zur letzten korrekten Version des Skripts zurückspringen.
Suchfeld	Separates Suchfeld im ScriptEditorControl, welches das integrierte Such-Control (CTRL+F) aufrufen könnte und direkt den Suchstring einfügen. Weitere Infos: http://documentation.devexpress.com/#windowsforms/CustomDocument6877

ISSUE	BESCHREIBUNG
Array-Typen	Arrays werden vom Lexer noch nicht erkannt. Dazu könnte man z.B. das Identifier-Symbol anpassen, dass dieses noch einen Index speichern kann und im Lexer den Index innerhalb der [ ] Klammern auslesen.
Automatischer Zeilenumbruch	Das RichEditControl von DevExpress macht automatisch, wenn man am Ende der Zeile angelangt ist, einen Zeilenumbruch. Dies ist für einen Code-Editor jedoch nicht gewünscht. Momentan ist die Zeilenbreite jedoch hoch genug gewählt, dadurch sollte dieser Fall nur sehr selten vorkommen.
Hyperlinks	Im RichEditControl können trotz Deaktivierung teilweise Hyperlinks z.B. durch Eingabe von „www.hsr.ch“ oder ähnlichem erstellt werden. In der DevExpress Sample Applikation wird nach dem deaktivieren von Hyperlinks das Dokument neu in den Editor geladen, deshalb geht es dort. Für uns sieht das eher nach einem Workaround aus.
Fehlermeldungen auslagern	Die Fehlermeldungen sind momentan noch im Parser/Checker „hardcoded“. Dies sollte man zwecks besserer Konfigurierbarkeit und Übersetzung noch in die Properties auslagern.

Auswertung Arbeitsaufwand

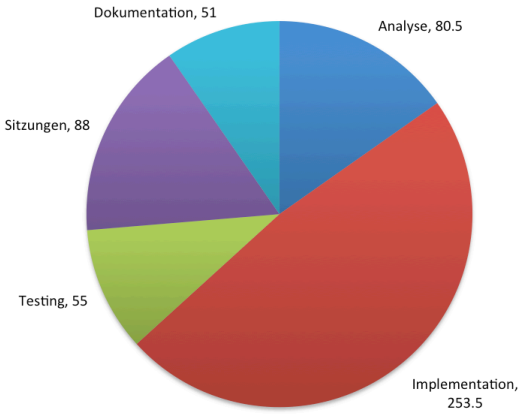
Arbeitsaufwand Total

Für die Studienarbeit hatten wir einen Arbeitsaufwand von insgesamt 480 Stunden zu leisten (8 ECTS = 240 Stunden * 2 Personen). Insgesamt haben wir 528, also rund 48 Stunden mehr als gefordert aufgewandt. Dieser Mehraufwand wurde vor allem in der Implementation geleistet, da wir auch noch alle optionalen Features aus der Ausgabenstellung umgesetzt haben.



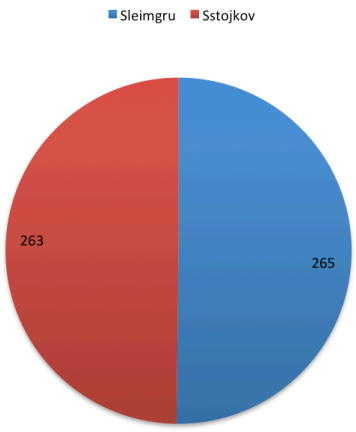
Arbeitsaufwand pro Aktivität

Die geleisteten Stunden, aufgeteilt nach Aktivität. Der grösste Posten ist Implementation + Test mit insgesamt 308.5 Stunden. Für die Analyse welche zugleich Teil 1 aus der Aufgabenstellung ist, haben wir 80.5 Stunden aufgewendet. Der dritte grosse Posten sind die Sitzungen. Dies beinhaltet die wöchentlichen Meetings mit Prof. Bläser und zudem die Meetings mit Eaton sowie die Präsentation in St. Gallen. Vor allem die Vorbereitungen für diese Treffen haben jeweils relativ viel Zeit benötigt.



Arbeitsteilung

Die Arbeitsaufteilung hat sehr gut geklappt. Slobodan Stojkovic hat 263 Stunden und Stefan Leimgruber 265 Stunden geleistet. Es ist nicht verwunderlich, dass die Stunden so nahe beieinander liegen, denn wir haben exakt den gleichen Stundenplan und konnten darum immer gemeinsam am Projekt arbeiten.



Persönlicher Bericht

Slobodan Stojkovic

Bereits bevor das 5. Semester begonnen hatte, waren wir auf der Suche nach einer spannenden Arbeit. Wir hatten bezüglich der Technologie unsere Vorstellungen und Wünsche, jedoch keine weiteren Vorstellungen über ein konkretes Thema, bzw. Gebiet.
Die Technologie sollte Microsoft sein und wenn möglich etwas mit C#, WPF, WinForms, im Usability-Bereich, oder ähnliches. Wir hatten verschiedene Dozenten angefragt und darunter auch Prof. Dr. L. Bläser. Als er uns mitteilte, dass er ein Projekt mit einer Firma aus St.Gallen hatte, waren wir sehr gespannt darauf. Nach einem Meeting mit dem Industriepartner Eaton Automation GmbH war für uns schnell klar, dass wir dieses Projekt realisieren wollten und so begann auch unsere Studienarbeit im 5. Semester.
Mir war anfänglich nicht ganz klar in welche Richtung sich das Projekt, bzw. der Script-Editor entwickeln wird. Da ich keine Erfahrungen im Themengebiet Compilerbau hatte, war mir nicht ganz klar wie wir dies realisieren sollten.
Wenn ich heute zurückschaue und nachdenke, wie wir das Projekt mit Herr Bläser angegangen sind, dann bin ich sehr froh, hatten wir eine so kompetente Betreuung. Denn wir wurden nicht einfach ins kalte Wasser geschmissen, sondern erhielten Privatunterricht im Compilerbau. Zum Glück mussten wir uns nicht durch die Compilerbau-Literatur durchkämpfen und uns alles selber beizubringen, sondern wir gingen immer Schritt für Schritt das nächste Problem, bzw. den nächsten Teilschritt des Compilers an. Denn hätten wir uns alles selber beibringen müssen, wären wir bestimmt nicht so weit gekommen und hätten bestimmt auf einige Features verzichten müssen, welche wir schlussendlich implementiert haben.
Ich habe bei dieser Studienarbeit sehr viel im Themengebiet des Compilerbaus gelernt, deshalb bin ich sehr froh, dass wir keine 0815-CRUD-Lösung implementiert haben, sondern wirklich in ein komplexes Thema eingestiegen sind. Ich konnte mein Wissen aus dem Fach Mathematik für Informatiker 2 auffrischen, den Sinn des Faches verstehen und sehr viel Neues dazulernen. Weiterführend konnte ich meine Kenntnisse in den .NET Technologien (vor allem C#) vertiefen.
Ebenfalls war die Zusammenarbeit mit dem Industriepartner Eaton Automation GmbH sehr interessant und motivierend, denn sie standen uns jeder Zeit zur Verfügung und schätzen unsere Lösung immer sehr. Dies entnahmen wir den Feedbacks unserer Präsentationen und auch dem E-Mail-Kontakt.
Über meinen Teamkollegen möchte ich auch gerne ein paar Worte verlieren. Da Herr S. Leimgruber und ich ein eingespieltes Team sind, konnten wir uns immer sehr gut organisieren und es tauchten nie gravierende Probleme auf, welche die Zusammenarbeit beeinträchtigt hätten.
Zusammenfassend kann ich sagen, dass ich SEHR stolz auf unsere Lösung bin, welche oft vom Umfang her von anderen Parteien als mögliche Bachelorarbeit eingestuft wurde und wir sie trotzdem als Studienarbeit gemeistert haben. Wir haben nicht nur die „obligatorischen" Features implementiert, sondern auch die optionalen Features, ganz zur Zufriedenheit unseres Industriepartners.

Stefan Leimgruber

Nach dem erfolgreichen Software-Engineering 2 Projekt, war ich sehr erfreut über eine weitere Zusammenarbeit mit Prof. Dr. Luc Bläser. Mein Projektpartner ist Herr Stojkovic. Wir kennen uns sprichwörtlich „since day one " an der HSR. Da wir jegliche Projekte, Testate und Miniprojekte zusammen gemacht haben, sind wir ein sehr eingespieltes Team. Diesmal sind wir jedoch nicht unser eigener Auftraggeber wie im SE2 Projekt, sondern es ist die Firma Eaton Autiomation GmbH in St. Gallen. Den Auftraggeber Herr Michael Greminger, Head of R&D Lean Automation, lernten wir bereits im Sommer bei einem ersten Gespräch kennen. Es wurde auch gleich eine erste Version der Aufgabenstellung angeschaut. Da uns die Aufgabe sehr zusagte und auch Herr Greminger einen angenehmen Eindruck machte, gaben wir schnell unsere Zusage für dieses Projekt. Herrn Mathias Schwyn unsere Ansprechsperson für die Dauer der Studienarbeit lernten wir beim Kick-Off Meeting kennen.
Die ursprüngliche Aufgabe sah vor, einen hochgradig geführten Script-Editor für das neue Galileo zu entwickeln. Später einigte man sich darauf, dass es jedoch eher Sinn machen würde, einen Editor für den fortgeschrittenen Benutzer zu erstellen und gleichzeitig den Anfänger zu begleiten. Der Fokus verschiebte sich dadurch eher etwas auf die Analyse der Sprache und die Validierung des Script-Codes. Da wir keine Erfahrungen im Compilerbau haben schien uns diese Aufgabe für eine Studienarbeit durchaus als sehr anspruchsvoll. Wir haben infolgedessen von Prof. Bläser einen wirklich hervorragenden Crash Kurs in das Thema Compilerbau erhalten. Ich persönlich empfand die Entwicklung am Lexer/Parser/Checker eine der kniffligsten Programmier-Aufgaben im Informatik-Studium an der HSR. Durch die wirklich hervorragende Betreuung hat es jedoch immer Spass gemacht und der Lernerfolg war enorm. Ich erinnere mich noch als unser Betreuer in einer Sitzung direkt Beispielcode zum Parser (direkt bezogen auf die Galileo-Script-Sprache) auf einen Block geschrieben hatte. Solch eine unkomplizierte Vorgehensweise hat mir sehr gut gefallen und war für das Verständnis sehr wertvoll.
Trotz der reichlichen Arbeit am Script-Check konnten wir uns auch im GUI Bereich etwas austoben. Da wir ein TextEdit-Control von DevExpress verwendeten, konnten wir nach der Implementation des Lexers auch das Syntax-Highlighting ziemlich schnell im Control umsetzen. Ein grosses Erfolgserlebnis hatten wir nach der Implementierung des „Live-Parsing", d.h. die Script-Überprüfung bei jedem Tastenanschlag. Auch wenn unsere Arbeit insgesamt weniger auf GUI und Usability ausgelegt war, ist es jedoch schon schön, etwas zu entwickeln was Usability-Technisch sehr fortschrittlich ist. Eine weitere spannende Aufgabe war die Entwicklung einer sogenannten Auto-Completion. Hier wird man sich bewusst, wie komplex die Evaluierung der Vorschläge sein kann, da man während der Eingabe vielfach keine syntaktisch korrekten Scripts vor sich hat. Trotzdem haben wir auch in diesem Bereich etwas geschaffen worauf man stolz sein kann und es macht wirklich Spass Code mit Hilfe seiner selbst entwickelten Auto-Completion zu erstellen.
Ich möchte mich auch noch bei der Firma Eaton und auch bei Prof. Bläser für das Lob während der Arbeit bedanken. Dies motiviert einen wirklich zu Spitzenleistungen. Nun freue ich mich auf die Bachelorarbeit die wir auch wieder zusammen mit Prof. Luc Bläser machen dürfen.

Glossar

BEGRIFF	ERLÄUTERUNG
Tag/Tags	Attribut das von einer Maschine oder lokal (z.B. pi) zur Verfügung gestellt wird
Rezept	Rezept
Funktionsgruppen	Zugehörigkeit der Modulbasierten Funktionen zu einer Gruppe z.B. Math, String, Runtime
IntelliSense / Auto-Completion	Hilfsmittel für automatische Code Vervollständigung
DevExpress	Firma welche Control-Libraries verkauft. Siehe www.devexpress.com
Lexer	Siehe Kapitel Architektur – Lexer
Parser	Siehe Kapitel Architektur – Parser
Checker	Siehe Kapitel Architektur - Checker

Literaturverzeichnis

ABK.	LITERATUR
[WIR]	Wirth, Prof. Dr. Niklaus. Grundlagen und Techniken des Compilerbaus. München: Oldenbourg Wissenschaftsverlag GmbH, 2011.
[WPAR]	Wikipedia: Parser. (04. November 2013). Abgerufen Dezember 2013 von http://de.wikipedia.org/wiki/Parser
[WCHE]	Wikipedia: Semantische Analyse. (24. September 2013). Abgerufen Dezember 2013 von http://de.wikipedia.org/wiki/Compiler#Semantische_Analyse

Abbildungsverzeichnis

ORT	ABBILDUNG
Titelblatt	http://zoomwalls.com/apple-white-keyboard-macro-hd-wallpaper/
Protokolle	http://www.du-bist-witzenhausen.de/wp-content/uploads/2012/04/Notizblock.jpg
Sprachumfang	http://thumbs.dreamstime.com/z/source-code-programming-16826823.jpg
Theorieunterlagen	http://img.talkandroid.com/uploads/2013/11/stack_of_books.jpg

Dankesbrief vom Industriepartner

Mit dem Skripteditor haben sie einen wichtigen Baustein des neuen Galileo Designtools geschaffen. Mit seiner neuen Funktionsweise wird er das Programmieren für unzählige Galileo Anwender erleichtern und ihnen eine Menge Zeit ersparen. Wir sind sehr zufrieden mit dem Ergebnis und möchten uns herzlich bedanken.
Wir wünschen ihnen viel Erfolg im weiteren Verlauf ihres Studiums.