EasyLanguage

****

Oliver Suchen  
Simon Thomann

CT Projekt

Herr Lempp

Gliederung

[2 Informationen zum Projekt 3](#_Toc376715996)

[2.1 Um was geht es bei unserem Projekt? 3](#_Toc376715997)

[3 Informationen zum Quellcode 3](#_Toc376715998)

[3.1 Ungarische Notation 4](#_Toc376715999)

[4 Informationen zur Dokumentation 5](#_Toc376716000)

[5 Software von Drittanbietern 6](#_Toc376716001)

[5.1 GitHub 6](#_Toc376716002)

[5.2 Qt 6](#_Toc376716003)

[5.3 Boost 7](#_Toc376716004)

[6 Genesis 7](#_Toc376716005)

[7 Die Entwicklung des Quellcodes 11](#_Toc376716006)

[8 Die Entwicklung der Datenbanken 16](#_Toc376716007)

[9 Finale Version 19](#_Toc376716008)

[10 Konflikte 23](#_Toc376716009)

[10.1 GitHub 23](#_Toc376716010)

[10.2 Qt 23](#_Toc376716011)

[10.3 Quellcode 23](#_Toc376716012)

[11 Quellcode kompilieren 24](#_Toc376716013)

[11.1 MingW 4.8 32-Bit 24](#_Toc376716014)

[12 Quellen 26](#_Toc376716015)

# Informationen zum Projekt

Wohlgemerkt lautet der Titel unseres Projekts „EasyLanguage“. Dieser Name setzt sich aus dem Englischen Wort „Easy“ für „einfach“ und „Language“ für „Sprache“ zusammen. Der Name soll also darstellen, dass damit einfach eine Sprache erlernt werden kann. Hinzufügend müssen wir sagen, dass unsere Software bis jetzt nur für die Sprache Englisch und Spanisch getestet wurde. Das Hinzufügen eigener Sprachen erfolgt auf eigene Gefahr.

## Um was geht es bei unserem Projekt?

Unser Projekt hat das Ziel einem Nutzer durch das Trainieren von Vokabeln eine Sprache beizubringen. Dabei werden Sprachdatenbanken und eine dazugehörige Konjugationstabelle eingelesen. Diese Sprachdatenbanken und Konjugationstabellen müssen jedoch von irgendeiner weiteren Person angeboten werden, beispielsweise von uns, den Software-Herstellern oder anderen freien Drittanbietern. In den Sprachdatenbanken werden Vokabeln gespeichert. Es wird zwischen einem normalen Wort und einem Verb differenziert. Sollte die Vokabel ein Verb sein, so wird getestet ob jenes regelmäßig oder unregelmäßig ist. Sollte das Verb regelmäßig sein, wird anhand der Konjugationstabelle, für die entsprechende Sprache, versucht, alle Personen für das Verb zu konjugieren. Sollte das Verb unregelmäßig sein, müssen alle Personen mit einer unregelmäßigen Konjugation angegeben werden. Personen die eine regelmäßige Konjugation besitzen, müssen nicht angegeben werden und können automatisch konjugiert werden. Dies haben wir aus Gründen der Speicherverwaltung eingebaut, denn so eine Sprachdatenbank kann durchaus riesig werden.

# Informationen zum Quellcode

Unser Projekt wurde in C++ programmiert. Genau genommen verwenden wir C++11. Dies mussten wir aufgrund eines bestimmten Datentyps namens std::tuple (#include <tuple> ). Permanent haben Simon und ich versucht zusammen zu arbeiten, da Simon jedoch geringere Kenntnisse in C++ besitzt als ich, musste des Öfteren ich ihm bei seinem Quellcode helfen. Nichtsdestotrotz haben wir es noch durch eine gute Zusammenarbeit geschafft, einen funktionstüchtigen Quellcode auf die Beine zu stellen.

## Ungarische Notation

Wir haben stark darauf geachtet unseren Variablen gute Namen zu geben. Dazu haben wir die sogenannte Ungarische Notation verwendet. Jedoch leicht erweitert.  
Der Bezeichner einer Variablen setzt sich nun wie folgt zusammen:

{Präfix}{Datentyp}{Bezeichner}

Es gelten alle Präfixe und Datentypen die hier verfügbar sind:

<http://de.wikipedia.org/wiki/Ungarische_Notation>

Die Liste der Präfixe wird durch folgende Liste erweitert:

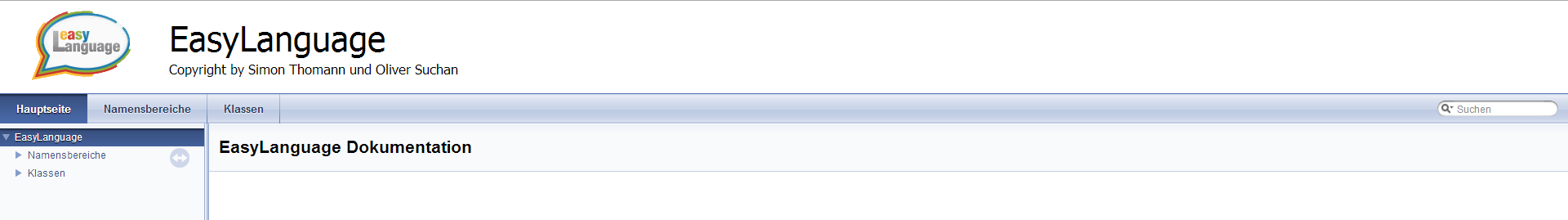
|  |  |  |
| --- | --- | --- |
| Präfix | abgeleitet von | Bedeutung |
| tpl | **t**u**pl**e | Verhält sich wie ein Array nur, dass es eine konstante Anzahl an Elementen besitzt. Alle Darauffolgenden Datentypen oder Präfixe beziehen sich auf den Inhalt des Tuples. Sollte der gleiche Datentyp mehrmals verwendet werden, reicht es ihn einmal aufzuführen |

Die Liste der Datentypen wird durch folgende List erweitert:

|  |  |  |
| --- | --- | --- |
| Datentyp | abgeleitet von | Bedeutung |
| voc | **Voc**abulary | Eigener Datentyp Voc, für die Vokabeln einer Sprache |
| verb | **Verb** | Eigener Datentyp Verb, für die Verben einer Sprache |
| lang | **Lang**uage | Eigener Datentyp, Language, für eine Sprache |
| lect | **Lect**ion | Eigener Datentyp Lection, für die Lektionen einer Sprache |
| qst | **QSt**ring | String Datentyp des Qt-Frameworks |
| st | **s**ize\_**t** | Für die Größe eines Arrays oder dem Index in einem Array |
| cn | **C**onjugatio**n** | Eigener Datentyp Conjugation, zum Speichern von Konjugationen |
| cj | **C**on**j**ugator | Eigener Datentyp Conjugator, zum Konjugieren von Verben |
| trnr | **Tr**ai**n**e**r** | Eigener Datentyp Trainer, zum Trainieren von Vokabeln |

# Informationen zur Dokumentation

Beim Quellcode haben wir uns dazu entschlossen, nur die Header-Dateien zu dokumentieren, da dort die signifikanten Informationen zu Funktionen und Klassen stehen. Nach der Dokumentation haben wir anhand einer Software namens Doxygen eine HTML-Dokumentation generiert. Diese Dokumentation ist sowohl auf Github als auch auf dem beigelegten Datenträger verfügbar. Um diese Dokumentation anzuzeigen, muss man nach der Datei „index.html“ suchen. Sobald diese gefunden wurde, einfach in ihrem Browser des Vertrauens öffnen. Dies sollte dann ungefähr so aussehen:



# Software von Drittanbietern

## GitHub

GitHub ist eine Software die es ermöglicht Quellcode einer Software mit Versionskontrolle kostenlos auf einem externen Server zu speichern. Sogenannte Repositorien. Die kostenlose Version besagt jedoch, dass es ein öffentliches Repository sein muss, sprich von allen Personen einsehbar. Sollte man ein privates Projekt anlegen wollen, muss man einen bestimmten Betrag zahlen um sich eine gewissen Anzahl an privaten Repositorien anzueignen. Additional besteht die Möglichkeit als Gruppe an einem Projekt zu arbeiten, weshalb wir uns auch für GitHub entschieden haben. Es gibt weitere zahlreiche andere Angebote die im Prinzip das gleiche Ergebnis liefern, dennoch entschieden wir uns der Einfachheit halber für GitHub. Durch die Integration von GitHub in die IDE Qt, welche wir verwenden, bot uns dies natürlich einen weiteren Vorteil. Geänderten Quellcode werden durch sogenannte Commits dokumentiert. Dem Commit entsprechend werden ebenfalls alle vorgenommenen Änderungen an den Dateien, die geprüft werden sollen, angezeigt. Hier der Link zu unserem GitHub-Repository: <https://github.com/cranktec/EasyLanguage>  
Später mehr, zu den Problemen denen wir entgegnet sind.

## Qt

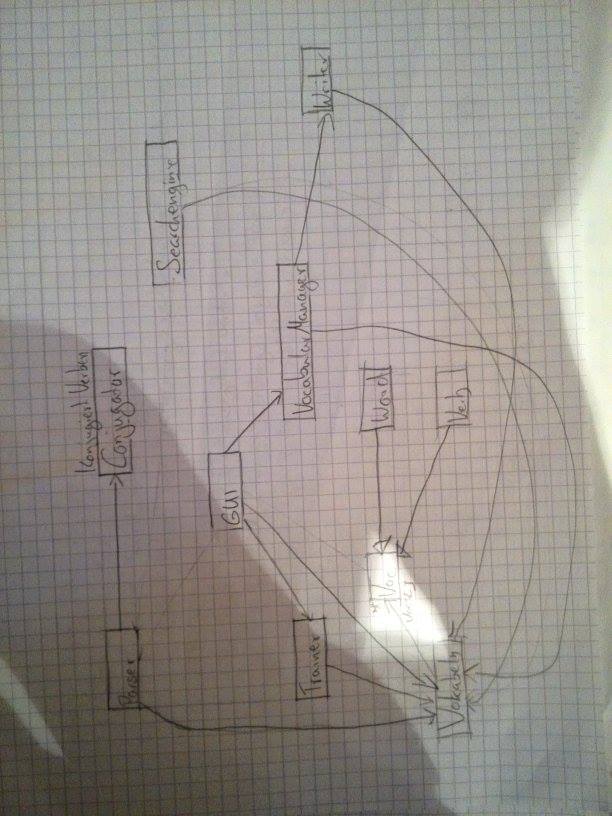
Qt ist ein kostenloses und opensource Framework mit einer cross-platform-ability. Sprich auf vielen verschiedenen Betriebssystemen verfügbar. Darunter Windows, Linux, iOS, Mac OS, Android und vielen mehr. Zusätzlich ist es für viele verschiedenen Programmiersprachen verfügbar, glücklicherweise auch für C++. Die Lizenz besagt, dass man dieses Framework umsonst verwenden darf, vorausgesetzt, dass man ein opensource Projekt plant. Da dies bei uns der Fall ist, hatten wir keinen Grund zur Sorge wenn wir das Framework verwenden würden, was wir auch taten. Des Weiteren liefert Qt eine hauseigene IDE namens QtCreator. Mit dieser ist es möglich Quellcode zu managen und zu kompilieren. Der Hauptgrund weswegen wir uns für Qt entschieden, war die Möglichkeit ohne vielen Aufwand sehr gute GUIs zu erstellen. Da wir unbedingt in C++ programmieren wollten, um uns so auch auf das kommende Robotik-Projekt vorzubereiten kam uns dies sehr gelegen, da wir ansonsten sehr viel mehr Aufwand in die GUI hätten Stecken müssen statt in die wesentlichen Dinge.

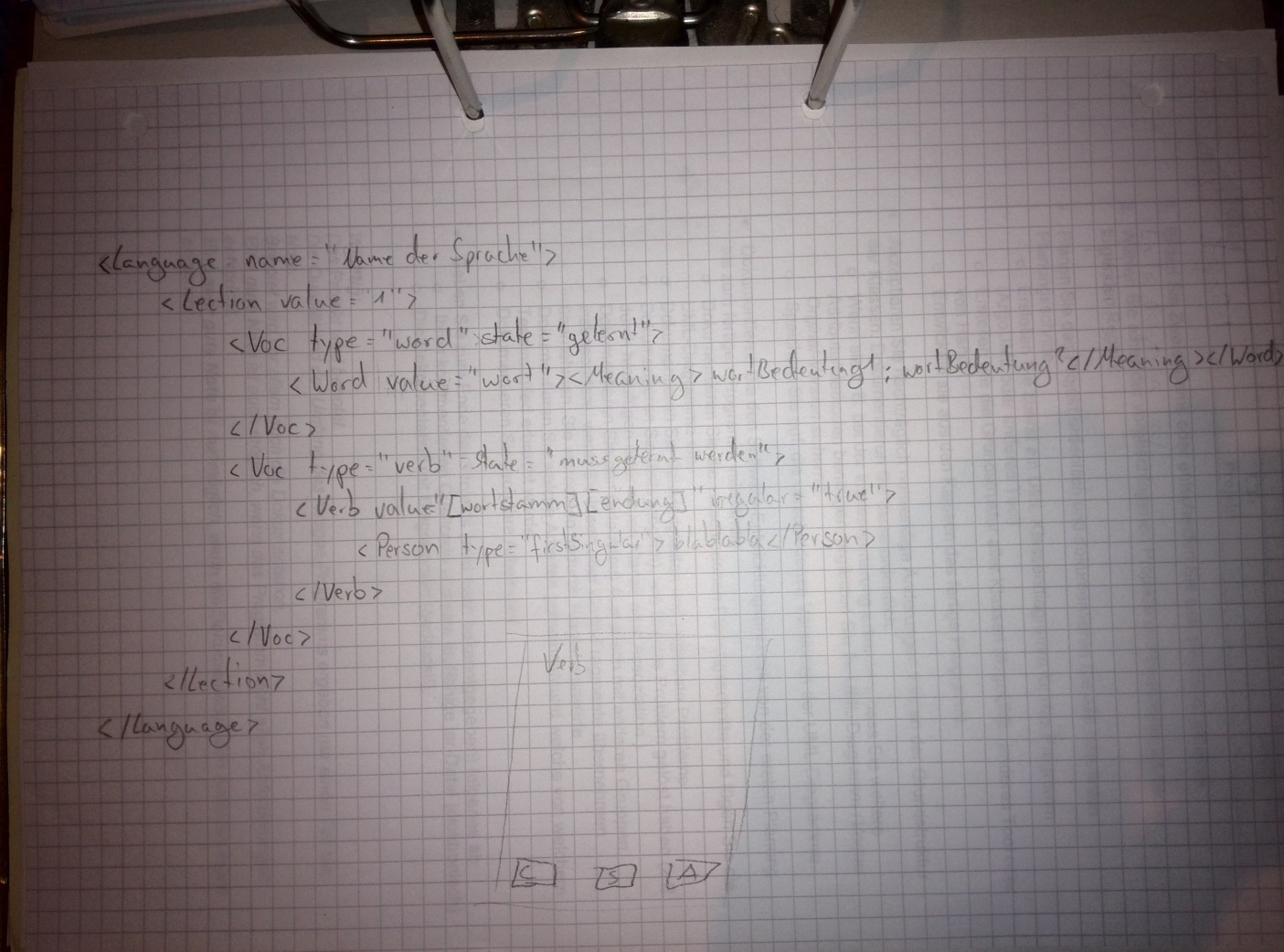
## Boost

Boost ist eine frei verfügbare opensource Bibliothek zum Einlesen bestimmter Dateistrukturen, speziell für die Programmiersprache C++. Darunter beispielsweise auch XML-Strukturen, welche wir sowohl für unsere Sprachdatenbanken als auch für unsere Konjugationstabellen verwendeten. Boost hat den Vorteil, dass es sehr effizient arbeitet und ebenfalls leicht zu bedienen ist. Ein Nachteil ist jedoch die Größe der Bibliothek, die ungefähr 90 MB beträgt. Da alle Dateien dieser Bibliothek in gewisser Weise aufeinander aufbauen, hatten wir keine andere Möglichkeit als die gesamte Bibliothek unserem Quellcode anzuhängen. Aus diesem Grund kann das Kompilat unserer Software durchaus über eine „normale Größe“ hinauswachsen.

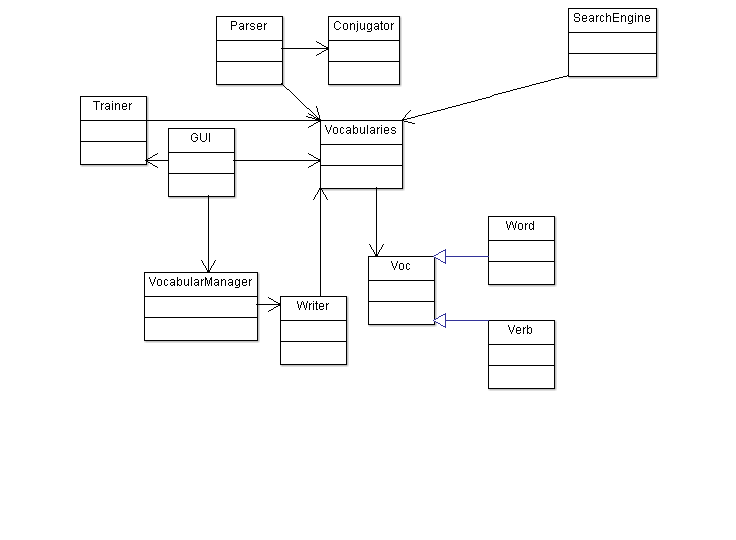
# Genesis

Es standen 2 Projekt-Möglichkeiten für unser Team zur Auswahl: Funktionsplotter oder Vokabel-Trainer. Da ich, Simon, jedoch zu diesem Zeitpunkt mangelnde Erfahrungen in der Programmiersprache C++ besaß, entschieden wir uns das vermutlich einfachere Projekt zu programmieren, den Vokabel-Trainer, nur um später festzustellen, dass die benötigte Arbeit und Kognition das selbe Niveau besitzen würden. Dennoch bereuen wir nichts. Nun als die Idee unseres Projekts feststand, entschlossen wir uns direkt dafür die XML-Struktur für unsere Datenbanken zu verwenden. Im Prinzip stand alles schon so fest, wie wir es in unserer ersten CT-Stunde geplant hatten, bis auf, dass die geplanten Klassen zu chaotisch und unüberlegt waren. Dazu jedoch später mehr. Hier einmal unser erstes UML-Diagramm und unsere erste Sprachdatenbank. Handschriftlich.





Da Beides eventuell schlecht lesbar sein kann, hier noch einmal Beides in einer etwas „digitaleren“ Form, sprich UML-Diagramm mit ArgoUML und die Datenbank als Text:



<Language name=**"Name der Sprache"**>

<Lection value=**"1"**>

<Voc type=**"word"** state=**"gelernt"**>

<Word value=**"wort"**>  
<Meaning>**wortBedeutung1;wortBedeutung2**</Meaning>  
</Word>

</Voc>

<Voc type=**"verb"** state=**"muss gelernt werden"**>

<Verb value=**"[wortstamm][wortendung]"** irregular=**"true"**>

<Person type=**"firstSingular"**>**blablabla**</Person>

</Verb>

</Voc>

</Lection>

</Language>

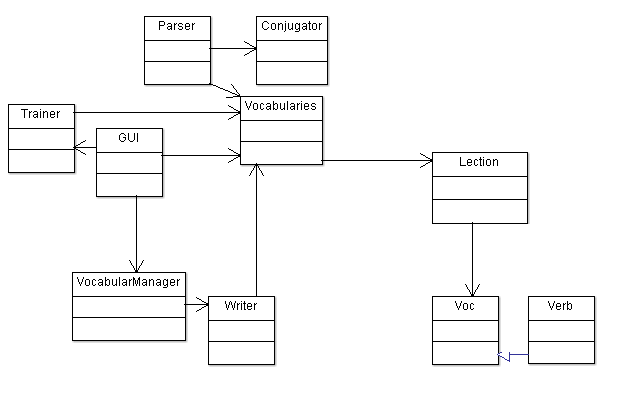
Dazu müssen wir sagen, dass unser finales UML-Diagramm stark von obigem abweicht. Zusätzlich wurden natürlich alle Member hinzugefügt und dokumentiert. Die Datenbank hingegen unterscheidet sich nicht allzu stark davon. Die erste Konjugationstabelle die wir entwarfen, entsprach hingegen direkt unseren Vorstellungen und musste nicht bearbeitet werden.

# Die Entwicklung des Quellcodes

Legende:

* Normale Klasse
* Singleton Klasse
* Interface

Wir entschlossen uns als Erstes die Klasse *Searchengine* rauszunehmen und dies die GUI übernehmen zu lassen. Später jedoch, entschlossen wir uns es aus zeitlichen Gründen ganz raus zu lassen. Ebenfalls stellten wir fest, dass alle benötigten Attribute der Klasse *Word* bereits der Basisklasse *Voc* zugeordnet worden sind, und es sinnlos gewesen wäre, extra eine Klasse anzufertigen, welche keine spezifischen Member besitzt, nur einen anderen Klassennamen. Additional bemerkten wir, dass es gut wäre, wenn wir Vokabeln in Lektionen unterteilen, so wie in den uns bekannten Sprach-Schulbüchern es auch der Fall ist. Aus diesem Grunde fügten wir eine weitere Klasse mit dem Namen *Lection* hinzu. Nach diesen Änderungen sah unser UML-Diagramm wie folgt aus:



Funktionen der Klassen:

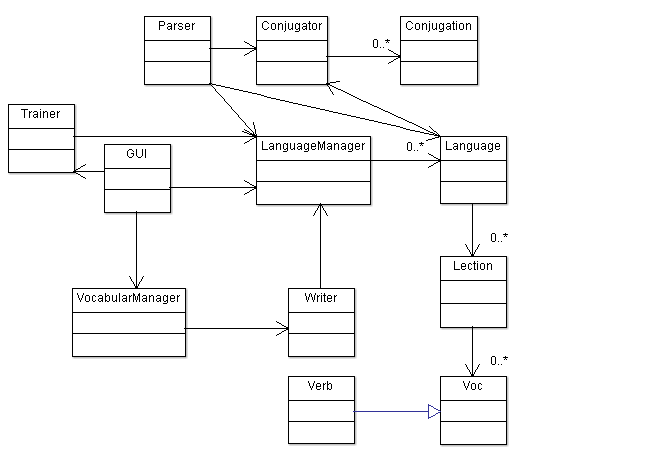
* Voc  
  Speichert Informationen zu Vokabeln, wie beispielsweise Bedeutungen, Synonyme und das Wort selbst. Außerdem auch den aktuellen Lernstatus.
* Verb  
  Speichert Informationen zu einem Verb, beispielsweise wie es konjugiert werden muss wenn es unregelmäßig ist, den Wortstamm und die Wortendung.
* Trainer  
  Klasse um Vokabeln zu trainieren, es werden immer automatisch 10 zu trainierende Vokabeln für eine bestimmte Sprache oder Lektion generiert. Anzahl ist anpassbar.
* Lection  
  Speichert alle Vokabeln einer bestimmten Lektion.
* Vocabularies  
  Speichert alle Lektionen aller Sprachem. Wird zu einem späteren Zeitpunkt durch die Klasse Language ersetzt.
* Conjugator  
  Konjugiert Verben. Wird vom Parser erzeugt und der entsprechenden Sprache hinzugefügt.
* VocabularManager  
  Zu diesem Zeitpunkt war der Zweck davon, dass Vokabeln/ Lektionen hinzugefügt und bearbeitet werden können. Später wurde diese Klasse jedoch wieder gelöscht und direkt in die GUI integriert. Keine Zwischenklasse. Außerdem besitzt sie nun nur noch die Funktionen Vokabeln und Lektionen hinzuzufügen.
* GUI  
  GUI die anhand der Qt IDE generiert wurde.
* Parser  
  Liest alle Sprachdatenbanken und die dazugehörigen Konjugationstabellen aus dem Ordner *languages* im aktuellen Verzeichnes ein und generiert daraus eine Instanz der Klasse *Languages*.
* Writer  
  Speichert alle bearbeiteten Sprachen neu ab, sodass beispielsweise durch den Trainer gewonnene oder verlorene „Punkte“ nicht verloren gehen.

Nun setzten wir uns an die Planung der Klasse *Conjugator*, schnell viel uns auf, dass es sehr kompliziert ist, alles nur mit einer Klasse zu regeln. Wir hätten mehrere *Tuples* und *Vektoren* verwenden müssen, was den gesamten Code komplizierter und unleserlicher gemacht hätte. Glücklicherweise ist C++ eine objektorientierte Sprache und hat den Sinn, komplizierte Dinge, einfach zu machen, durch beispielsweise das Aufteilen in mehrere verschiedene, eigene Objekte. Daraus schlussfolgerten wir, dass wir eine Klasse *Conjugation* benötigten. In dieser Klasse sollten sich Wortendungen befinden und deren entsprechenden regelmäßigen Konjugationen. Beispielsweise im Spanischen gibt es die Endungen *er*, *ir* und *ar.* Nehmen wir nun das Verb *aprender* (lernen) als Beispiel. Dieses Verb wird wie folgt konjugiert:

* Yo aprendo
* Tu aprendes
* El/Ella/Usted aprende
* Nosotros aprendemos
* Vosotros aprendéis
* Ellos/Ellas/Ustedes aprenden

All diese Informationen werden in der Klasse *Conjugation* gespeichert. Sprich die Wortendung und die Konjugationen, inklusive Pronomen.

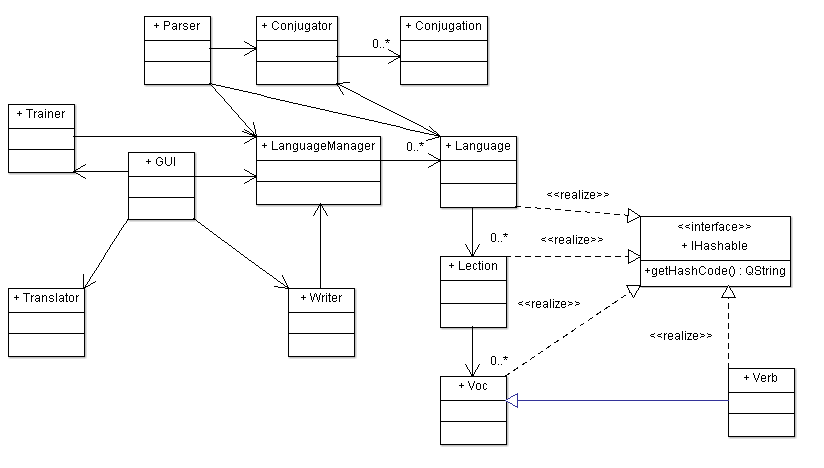
Wir nannten *Vocabularies* zu *Language* um und es viel uns auf, dass wir eine Zwischenklasse brauchen, die unsere Sprachen, bzw. *Language*-Instanzen, managt. Eine Instanz der Klasse *Language* enthielt jeweils eine Instanz der Klasse. Darum entworfen wir eine weitere Singleton Klasse, namens *LanguageManager*. Diese sollte alle Sprachen speichern und abrufbereitmachen. Das UML-Diagramm entwickelte sich wie folgt:

Erweiterung zu *Funktionen der Klassen*:

* Language  
  Speichert alle Lektionen einer Sprache und einen dazugehörigen Konjugierer der durch die Klasse *Parser* erzeugt wurde.
* Conjugation  
  Speichert Informationen wie ein *Conjugator* Verben zu konjugieren hat
* LanguageManager  
  Speichert *Language*-Instanzen und überprüft ob Sprachdatenbanken zur Laufzeit verändert wurden. Siehe kommendes Interface *IHashable.*

Nun kamen wir an den Punkt, an dem wir uns überlegten, wie wir am besten überprüfen können, ob eine *Language*-Instanz zur Laufzeit bearbeitet wurde. Wir nahmen uns ein Beispiel an Java, wo jede Klasse eine Funktion mit dem Namen *getHashCode()* besitzt. Diese liefert einen Hashcode für die aktuelle Instanz zurück um diese identifizieren zu können. Aus diesem Grund schrieben wir ein Interface mit dem Namen IHashable. In C++ gibt es kein direktes Schlüsselwort wie in Java für Interfaces sondern dies wird aufgrund der Methoden einer Klasse entschieden. Das heißt wir mussten eine normale Klasse mit einer abstrakten Methode (virtual QString *getHashCode*() = 0;) erstellen um unser gewünschtes Interface zu erhalten. Nun ließen wir die Klassen *Verb*, *Voc*, *Lection* und *Language* dieses Interface implementieren. Jede Klasse besaß nun eine eigene Definition für die Funktion *getHashCode()*. Damit war es uns ermöglicht die Sprachdatenbanken zu identifizieren. Wir generieren einen Hashcode sobald eine Sprachdatenbank eingelesen wurde und einen sobald das Programm geschlossen wird. Sollten diese Beiden Hashcodes nicht äquivalent sein, werden jene Sprachdatenbanken neu geschrieben.

Des Weiteren hatten wir einen groben Aufbau unserer GUI, den wir besser jetzt nicht hier zeigen, weswegen wir entschlossen die Klasse *VocabularManager* zu löschen und die GUI direkt auf die Singleton Klasse *Writer* zugreifen zu lassen. Additional fügten wir eine weitere Klasse mit dem Namen *Translator* hinzu, die Vokabeln einer Sprache ins Deutsche und die Vokabeln einer Sprache in eine andere eingelesene Sprache übersetzen konnte. Diese Funktion ist jedoch mit Vorsicht zu bedenken, da es vorkommen könnte, dass bei dem Übersetzen Fehler auftreten, auch wenn wir uns schworen nicht wie der Google-Übersetzer zu enden. Wörter aus dem Deutschen können – bis jetzt - nicht in eine andere Sprache übersetzt werden. Aus obigem Entwicklungsprozess ergibt sich folgendes UML-Diagramm:



Erweiterung zu *Funktionen der Klassen*:

* Translator  
  Übersetzt Wörter einer Sprache in eine andere geparste Sprache und ins Deutsche. Wörter aus dem Deutschen können jedoch nicht in eine andere Sprache übersetzt werden.
* IHashable  
  Wird von Klassen implementiert, die die Möglichkeit besitzen sollen Hashcodes ihrer aktuellen Instanz zu erzeugen. Dies dient um jene Instanz zu identifizieren zu können.

Das Konzept unserer Klassen stand soweit und es hat sich bis jetzt auch nichts mehr geändert. Nun wagten wir uns an die Member der jeweiligen Klassen, was nicht so schwer war, da wir auf Anhieb all unsere benötigten Funktionen und Attribute fanden. Siehe *Finale Version*.

# Die Entwicklung der Datenbanken

Wie bereits gesagt, entsprach die Struktur der Konjugationstabellen direkt unseren Vorstellungen und musste nicht nachträglich bearbeitet werden. Darum fangen wir auch mit dieser an. Hier als Beispiel die Englische-Konjugationstabelle:

<Conjugation>

Die Wortendung, für die diese Konjugation gilt [x] -> Wörter ohne bestimmte Endung

<Postfix value=**"[x]"**>

Personen die Singular sind

<Singular>

value = das Pronomen der Person; conjugation = Die Konjugation für die Person; [rw] -> rootword -> Der Wortstamm des Verbs

<First value=**"I"** conjugation=**"[rw]"** />

<Second value=**"You"** conjugation=**"[rw]"** />

<Third value=**"He;She;It;"** conjugation=**"[rw]s"**/>

</Singular>

Personen die Plural sind

<Plural>

es gilt das Gleiche wie für Singular

<First value=**"We"** conjugation=**"[rw]"**/>

<Second value=**"You"** conjugation=**"[rw]"**/>

<Third value=**"They"** conjugation=**"[rw]"** />

</Plural>

</Postfix>

</Conjugation>

Wie bereits gesagt, war dies unsere erste Version für eine Sprachdatenbank:

name = Name der Sprache

<Language name=**"Name der Sprache"**>

Value = die Lektionsnummer

<Lection value=**"1"**>

type = der Typ der Vokabel word/verb;

state = der Lernstatus der Vokabel;

<Voc type=**"word"** state=**"gelernt"**>

value = das Wort

<Word value=**"wort"**>

Die Bedeutungen des Worts, getrennt durch Semikolone  
<Meaning>**wortBedeutung1;wortBedeutung2**</Meaning>

</Word>

</Voc>

<Voc type=**"verb"** state=**"muss gelernt werden"**>

irregular = Regel- Unregelmäßigkeit des Verbs

<Verb value=**"[wortstamm][wortendung]"** irregular=**"true"**>

type = Stelle der Person

blablabla = Die Konjugation der Person

<Person type=**"firstSingular"**>**blablabla**</Person>

</Verb>

</Voc>

</Lection>

</Language>

Zu diesem Zeitpunkt traf sie noch recht genau unsere Vorstellungen. Als wir später jedoch die Klasse *Word* löschten und nur noch *Voc* und *Verb* besaßen, viel uns auf, dass wir das gleiche Prinzip auf unsere Datenbanken anwenden konnten.

<Language name=**"Name der Sprache"**>

<Lection value=**"1"**>

<Voc type=**"word"** state=**"gelernt"**>

<Meaning>**wortBedeutung1;wortBedeutung2**</Meaning>

</Voc>

<Voc type=**"verb"** state=**"muss gelernt werden"**>

<Meaning>**verbBedeutung1;verbBedeutung2**</Meaning>

<Person type=**"firstSingular"**>**blablabla**</Person>

</Voc>

</Lection>

</Language>

Dadurch sah das alles schon kompakter aus, jedoch konnten wir uns damit nicht zufrieden stellen, da es teilweise sehr umständlich war, Dateien mit solch einer Struktur zu parsen. Also wagten wir einen letzten Versuch, mit welchem wir uns zufrieden stellen konnten und es dabei beließen. Hier als Beispiel, Ausschnitte unserer aktuellen Sprachdatenbank für die Sprache Spanisch:

<Language name=**"Español"**>

<Lection number=**"1"**>

type -> keine Wörter mehr zur Unterscheidung sondern ein Boolean

state -> ebenfalls keine Wörter mehr sondern ints

value = direkt das Wort

<Voc type=**"0"** state=**"3"** value=**"palabra"**>

Um das Parsen und schreiben von Datenbanken zu erleichtern werden die Definitionen als Attribut gespeichert

<Definition value=**"Wort;"**/>

Verhält sich wie Definition

<Synonym value=**"verbo;"**/>

</Voc>

Wenn es ein Verb ist, wird automatisch das Attribut regular hinzugefügt, was einen Boolean darstellt

value = Wortstamm und Wortendung werden durch ein Semikolon getrennt, keine eckigen Klammern mehr

<Voc type=**"1"** state=**"3"** value=**"sal;ir"** regular=**"0"**>

<Definition value=**"ausgehen;"**/>

<Synonym value=**""**/>

Die unregelmäßigen Personen;   
Aufbau verhält sich wie bei den Konjugationstabellen, nur statt First, Second und Third einfach Person.

Personen die regelmäßig konjugiert werden können, werden weggelassen

<Persons>

<Singular>

<Person value=**"Yo"** conjugation=**"salgo"**/>

</Singular>

</Persons>

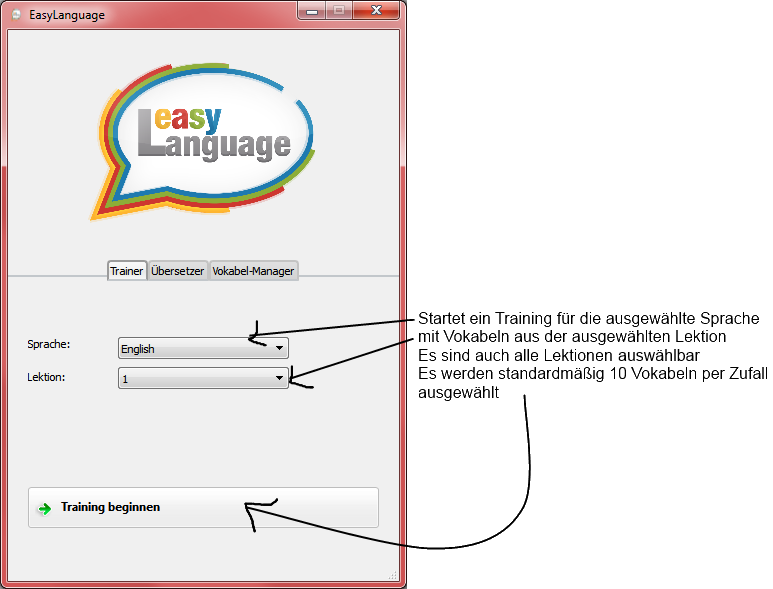
</Voc>

</Lection>

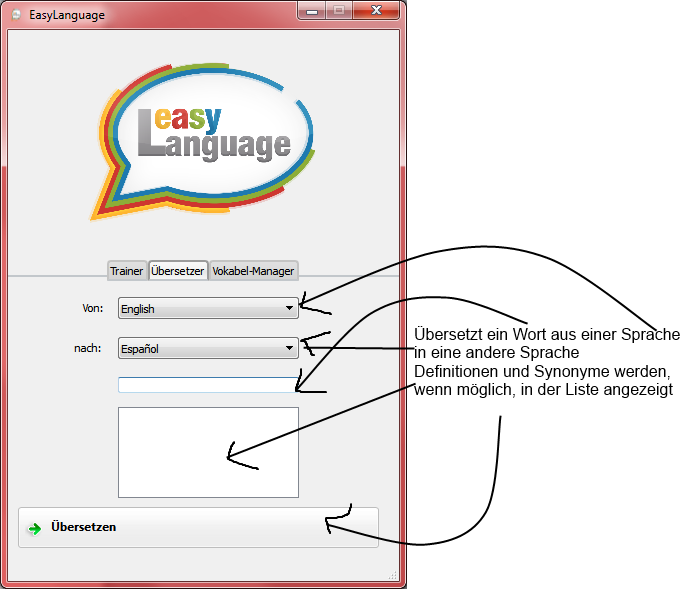
</Language>

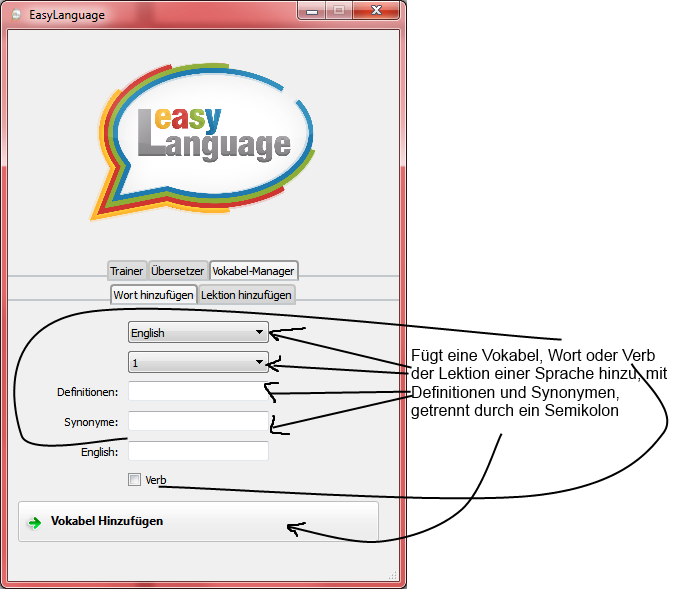
# Finale Version

Die fertige Version unseres Programms sieht so aus:

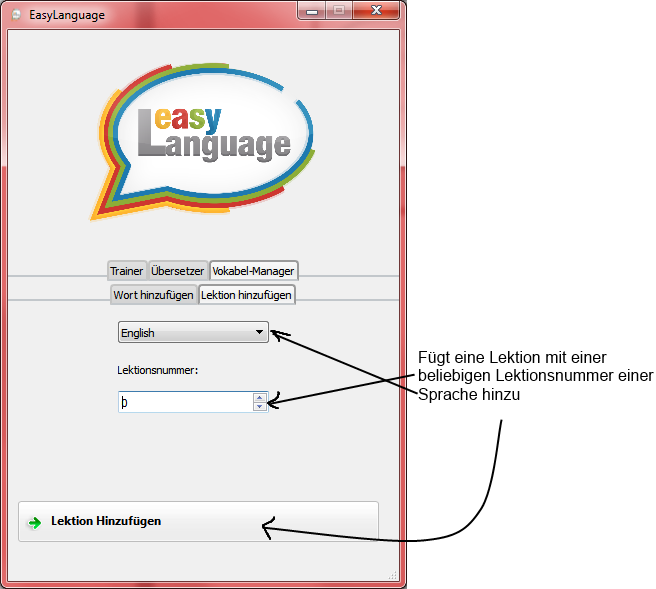


Es öffnet sich nun ein neuer Tab mit dem Namen Training. In diesem Tab werden Definitionen von Vokabeln gezeigt, welche man nun richtig eingeben muss. Das Lernstadium der entsprechenden Vokabel wird automatisch, nach Tätigung der Eingabe, neu festgelegt.



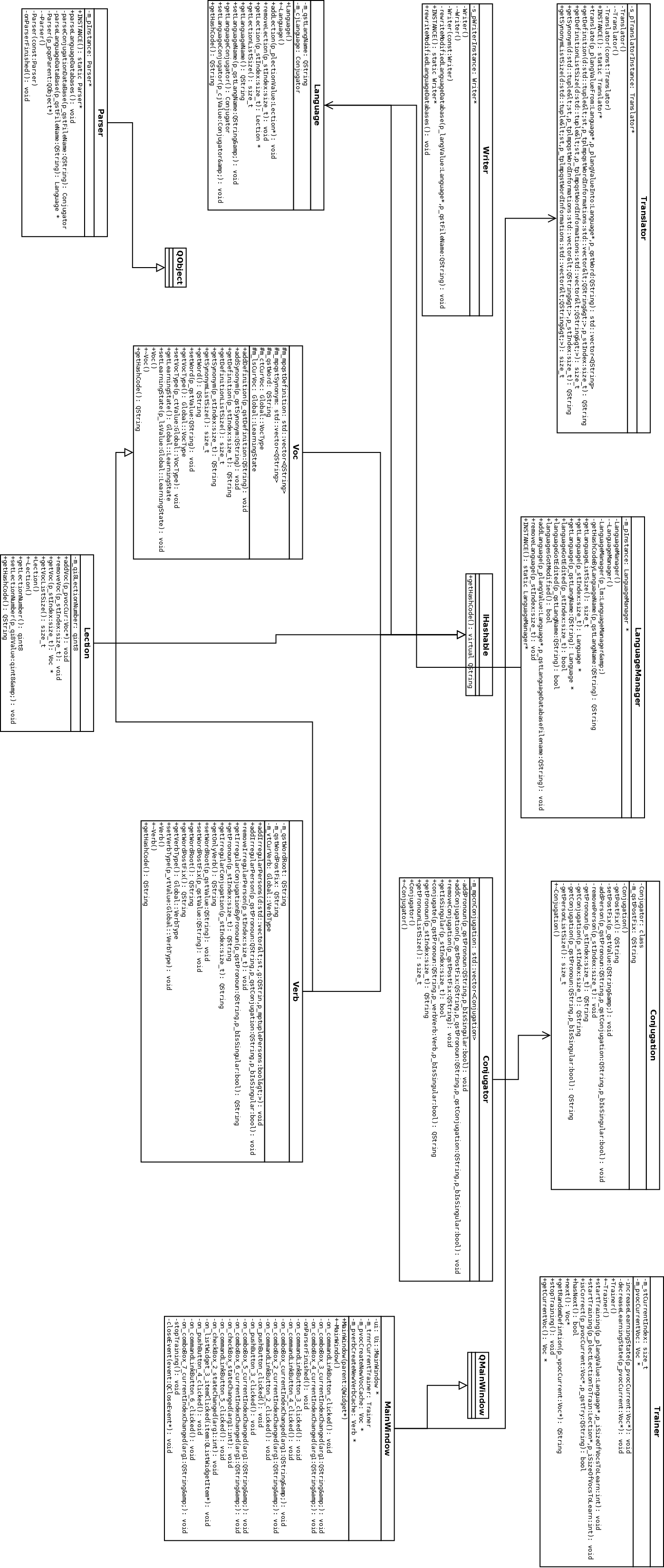


Sollte die Checkbox Verb geklickt werden, wird automatisch ein neuer Tab mit dem Namen Verb-Einstellungen geöffnet. In diesem besitzt man nun die Möglichkeit Verben zu konfigurieren. Die Checkbox Regelmäßig sollte man nur nach dem Füllen der beiden Felder, Wortstamm und Wortendung, getätigt werden. Nach dem Klicken wird die Liste mit den für die Sprache verfügbaren Personen gefüllt. Wenn man ein unregelmäßiges Verb hinzufügen möchte, sollte man die Checkbox wieder deaktivieren. Die Liste der Personen bleibt jedoch erhalten. Sobald man nun eine Person auswählt, kann man dort die unregelmäßige Konjugation hinzufügen, in dem man die Konjugation eingibt, die Checkbox Unregelmäßig klickt und danach auf den Button Fertig klickt.



Hier die finale Version unseres UML-Diagramms. Diese wurde mit dem Programm Dia erstellt. Aufgrund der Größe kann es sein, dass es nur schlecht leserlich ist. Aus diesem Grund, gibt es das Bild unseres finalen UML-Diagramms nochmal auf dem beigelieferten Datenträger im Verzeichnis *documenation* unter dem Namen *Finales UML-Diagramm.png*. Dort wird auch unser finales Kompilat zur Verfügung stehen. Sollten jedoch Probleme beim Ausführen des Kompilats entstehen, durch fehlende benötigte Bibliotheken des Qt-Frameworks, aufgrund der sogenannten shared compilation, empfehlen wir,

unser Projekt noch einmal selbst zu kompilieren. Siehe dazu den Punkt: *Quellcode kompilieren*.



# Konflikte

## GitHub

Aufgrund eines kleinen Versionsunterschiedes auf unseren Computern, wurde aus Versehen, beim Hochladen einer neuen Version, das komplette Repository sozusagen geleert. Frühere Versionen waren nicht mehr direkt über unser Projekt verfügbar. Glücklicherweise sind die getätigten Commits trotzdem verfügbar. Um diese einsehen zu können, muss lediglich folgende Link geöffnet werden:

<https://github.com/cranktec?tab=activity>

Dies hatte jedoch den Nachteil, dass die Übersichtlichkeit unserer geänderten Dateien exorbitant dekrementiert wurde.

## Qt

Da wir die Hauseigene IDE des Qt-Frameworks benutzen wollten, entstanden Probleme beim Installieren der IDE auf den Schulrechnern. Sollten wir es jedoch installiert bekommen haben, waren die benötigten Compiler nicht verfügbar. Dies hatte den Nachteil, dass der Code, welchen wir in der Schule angefertigt hatten, nur zuhause auf unseren Rechnern ohne Probleme getestet werden konnte. Sprich, unsere Fehler konnten wir teilweise erst zu einem viel späteren Zeitpunkt herausfinden, was uns teilweise aus dem Konzept gebracht hat.

## Quellcode

Wir besaßen einmal ein Problem mit unserem *Conjugator*. Dieser lieferte aus irgendeinem Grund seltsame Zeichen. Da wir jedoch keine Zeit hatten, dem Problem auf den Grund zu gehen, entschlossen wir uns, unser Projekt komplett neu zu schreiben. Aus dieser Tat folgte der oben aufgelistete Konflikt mit GitHub. Bis jetzt wissen wir nicht, aus welchem Grund sich dieses Problem entwickelte. Vermutlich hing es aber mit der internen Zeichenkodierung des *QString*-Datentyps zusammen. Genau können wir es aber nicht sagen.

# Quellcode kompilieren

Da wir persönlich unser Projekt nur mit MingW 4.8 32-Bit kompiliert und getestet haben, können wir nur empfehlen, dass Sie uns dies abschauen, da sonst durch andere Compiler, eventuell ungewollte Compiler-Fehler auftreten könnten.

## MingW 4.8 32-Bit

Am Einfachsten ist der Kompilier-Prozess durch die Qt-Framework IDE realisierbar, welche unter folgendem Link kostenlos verfügbar ist:

<http://qt-project.org/downloads>

Wie bereits gesagt, empfehlen wir unser Projekt mit MingW 4.8 32-Bit zu kompilieren. Darum wäre es praktisch sich folgende Distribution herunterzuladen:

<http://download.qt-project.org/official_releases/online_installers/1.5/qt-windows-opensource-1.5.0-x86-online.exe>

Dies ist ein Installer, welcher die gewünschten Pakete herunterlädt. Hierbei besteht der Vorteil, das Paket OpenGL außen vor zu lassen, welches für unseres Projekt nicht von Nöten ist. Wählen Sie also einfach nur „MingW48 32-Bit“ aus.

Nachdem Sie diesen Schritt vollzogen haben, können Sie die IDE öffnen, welche unter dem Namen QtCreator verfügbar ist. Nun benötigen Sie noch den Quellcode unseres Projektes. Sie besitzen einmal die Möglichkeit, sich den Quellcode von unserem GitHub-Repository herunterzuladen, oder den Quellcode auf dem beigelegten Datenträger zu verwenden. Um die Auswahl mit dem GitHub-Repository zu testen, müssen Sie einfach zu folgenden Web-Adresse navigieren:

<https://github.com/cranktec/EasyLanguage/archive/master.zip>

Nachdem Sie sie sich für eine der obigen Optionen entschieden haben, müssen die gleichen Schritte ausgeführt werden.

1. Entpacken Sie das zip-Archiv
2. Suchen Sie nach einer Datei mit dem Namen EasyLang.pro
3. Öffnen Sie jene Datei mit der Qt-IDE
4. Nachdem öffnen werden Sie aufgefordert das Projekt zu konfigurieren. Wählen Sie den MingW-Compiler
5. Betätigen Sie in der IDE folgenden Button  
      
   Wählen Sie wie auf dem Bild den Punkt *Release* aus.
6. Nun können Sie auf den grünen Pfeil direkt unten drunter drücken, um das Projekt zu kompilieren.
7. Es sollte nun ein neuer Ordner, beginnend mit „build“ erscheinen. Navigieren Sie dort dann in den Ordner *build/release*. Kopieren Sie anschließend das Kompilat, welche die Dateiendung .*exe* besitzt.
8. Nun wird es komplizierter. Es werden Qt-Bibliotheken benötigt. Öffnen Sie hierzu ein neues Windows-Explorer-Fenster und navigieren Sie in das Qt-Installationsverzeichnis. Navigieren Sie dort dann in das Verzeichnis *{Qt-Version (vermutlich 5.2.0)}/ 5.1.1/mingw48\_32/bin*. Kopieren Sie nun das erwähnte Kompilat in einen neuen Ordner. In diesen Ordner kopieren Sie den Ordner *languages* hinein, welcher sich im Verzeichnis *examples* befindet. Versuchen Sie nun das Kompilat zu starten und kopieren Sie alle Bibliotheken in den Ordner des Kompilats aus dem obigen Verzeichnis, die durch eine Fehlermeldung genannt werden.
9. Wenn Sie diese Schritte genau so befolgt haben, sollten Sie unsere Software nun ohne Probleme verwenden können.

# Quellen

**EasyLanuage-Logo**

Dieses Logo wurde von einem Freund von Oliver erstellt und uns kostenlos zur Verfügung gestellt. Diese Person beschäftigt sich sehr viel mit der Thematik Design. Sein Name ist René Haas.

**GitHub**

<https://github.com/>

**Qt**

<http://qt-project.org>

**Boost**

<http://www.boost.org/>

**Ungarische Notation**

<http://de.wikipedia.org/wiki/Ungarische_Notation>