Projektarbeit:
"htmlLaTeX" - Konvertierungsoftware
–Ausarbeitung–

von:
Kaiser, Björn
Mühlendamm 6
24937 Flensburg
bjoern-kaiser@versanet.de
Matrikel-Nr.: 371658

und

Baß, Björn
Ritterstraße 28
24939 Flensburg
b-bass@versanet.de
Matrikel-Nr.: 341125

Betreuer: Prof. Dr. Hans Werner Lang
SoSe 2011-I
Fachbereich Technik
Fachhochschule Flensburg

Abgabetermin: 23.03.2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung - Motivation

Das Ziel dieses Projektes ist eine möglichst flexible Anwendung zu schaffen, die aus einem Markup in eine andere per XML[1]-definierbare Syntax konvertieren kann.

Dies wurde am Beispiel von der Konvertierung von JavaDoc - generiertem HTML-Code und einer anschließenden Umwandlung zu LaTeX[2] verfolgt.

Um möglichst menschenlesbar weitere Konvertierungsszenarien umsetzen zu können wurden folgende Festlegungen getroffen:

- Die Eingabesemantik wird in einer XML-Datei beschrieben und soll die Umsetzung des Ausgangscodes in eine (pseudo)-HTML Semantik beschreiben.
- Die Ausgabesemantik wird ebenso beschrieben und definiert die Konvertierung in das Zielformat.
- Diese beiden Konfigurationsdateien müssen im Rahmen ihrer Syntax in der jeweils inline verfassten DTD [3] frei definiert werden können.

2. Durchführung der Projektarbeit

2.1. Rahmenbedingungen und Tools

Betriebssystem Es wurde parallel unter *Windows 7*[4] und *Linux*[5] entwickelt. Dies war erforderlich, da die notwendige Screenreader-Software unter Linux nicht verfügbar ist und keine Alternative unseren Anforderungen genügte (Vergrößerung, Invertierung, Curser- / Mauszeigerverfolgung).

Sprache Um einerseits hohe Plattformunabhängigkeit und andererseits Performanz zu erreichen, wurde das Projekt mit Hilfe der C++ Klassenbibliothek *Qt*[6] erstellt.

IDE Es wurde die Entwicklungsumgebung[7] *Qt Creator*[8] benutzt.

Lokalisierung Es wurden Übersetzungen in den Sprachen Englisch und Deutsch erstellt, die zur Laufzeit gewechselt werden können.

[1]**X**tensible **M**arkup **L**anguage
[2]**La**mport **TeX** - umfangreiche Sammlung von TeX-Makros
[3]**D**okument**t**yp**d**efinition
[4]Microsoft® und Windows® sind eingetragene Marken der Microsoft Corporation.
[5]Linux® ist ein eingetragenes Markenzeichen von Linus Torvalds
[6]Qt® ist ein eingetragenes Markenzeichen der Nokia Corporation. http://qt.nokia.com/
[7]engl. **I**ntegrated **D**evelopment **E**nvironment
[8]http://qt.nokia.com/products/developer-tools

Interface Das Programm ist sowohl über ein grafisches Interface als auch als per Skript bedienbar und verwaltet die Programmeinstellungen in einer XML-Datei.

Dokumentation Die Entwicklerdokumentation der Implementierungsdetails wurde aus dem Quellcode mithilfe des freien Dokumentationswerkzeuges *Doxygen*[9] generiert. Da die Dokumentation des Quelltextes ausführlich ist, werden in dieser Ausarbeitung vorwiegend die Arbeitsweisen, Überlegungen und Konzepte besprochen. Für Implementierungs*details* sei auf den Anhang (Seite 57ff) oder die Entwicklerdokumentation verwiesen, die auch alle Klassendiagramme enthält und die Kommentare vom Quellcode getrennt übersichtlich darstellt und durchsuchbar ist.

Konventionen Als Dokumentationssprache wurde Englisch gewählt. Die Headerdateien (Klassendeklarationen `.h`) beginnen mit einer allgemeinen Beschreibung der Funktion der Klasse und werden durch die speziellen Kommentare zu den einzelnen Membern ergänzt. So können aus den Headerdateien alle Informationen gewonnen werden, die für die Benutzung der Klasse relevant sind.

Die Klassendefinitionsdateien (`.cpp`) enthalten an einigen Stellen zusätzliche Kommentare, die die Implementierung dieser Klasse betreffen, um die Weiterentwicklung zu ermöglichen.

Versionierung Für die Versionierung wurde das verteilte Versionskontrollsystem *Git*[10] verwandt. Als Hoster diente der spezialisierte Webhosting-Dienst *GitHub*[11].

Unit Tests Nach einigen Problemen mit der Qt-eigenen Komponente für Unit Tests (QTestLib) haben wir auf Unit Tests vorerst verzichtet. (Nach einmaligem Kompilieren einer beliebigen Klasse, verursachte das Hinzufügen des Makros `Q_OBJECT`, das für die Verwendung der QTestLib erforderlich ist, unter Linux einer Fehler beim Linker - dies ist leider erst am Ende des Projektbearbeitungszeitraumes durch eine Weiche in der `.pro`-Datei von uns gelöst worden, die den Qt spezifischen Präprozessor MOC[12] zu korrektem Verhalten veranlassen konnte.)

Webpräsenz Auf http://opus4711.github.com/htmlatex/ können der Quelltext, die Entwicklerdokumentation, das Pflichtenheft und die Ausarbeitung (dieses Dokument) in einem zip oder tar-Archiv heruntergeladen werden.

[9]http://www.stack.nl/~dimitri/doxygen/index.html
[10]http://git-scm.com/
[11]http://github.com/
[12]meta object compiler

Eingabehilfen Als Screenreader wurde *Dolphin*[13] eingesetzt.

Issues Als Issue-Tracking-System wurde die Issues-Komponente von *GitHub* eingesetzt.

2.2. Implementierungsphase

Das Projekt wurde mithilfe der Extreme Programming-Methode teils in paralleler Einzelarbeit auf Personen- und Aufgaben(Issue)-bezogenen Entwicklungszweigen, teils gemeinsam umgesetzt.

Zu Begin wurde auf dem Hauptentwicklungszweig “master” der noch funktionslose Prototyp der Anwendung erstellt und die grafische Oberfläche entworfen. Im Folgenden wurden Teilfunktionalitäten in getrennten Branches in kurzen Zyklen (zwischen 30 Min und 5 Std. Programmierzeit) implementiert. Nach gegenseitiger Absprache wurde der Branch dann auf den beiden Betriebssystemen getestet und mit dem Hauptentwicklungzweig verschmolzen (“merge”).

Dateisysteme Um Plattformunabhängig zu bleiben, mussten für die Ermittlung der Dateipfade Anpassungen vorgenommen werden.

Listing 1: Beispiel einer Pfadweiche aus `documentreader.cpp`

```
        QFileInfo myfileinfo;
        if (QFileInfo(_indexFileInfo.absolutePath() + ↘
            →new_node->getAttributes()["href"]).exists())
            // unix
            myfileinfo = QFileInfo(_indexFileInfo.absolutePath()
                                   + new_node->getAttributes()["href"]);
        else
            // windows
            myfileinfo = QFileInfo(_indexFileInfo.absolutePath()
                                   + QDir::separator() + ↘
                                       →new_node->getAttributes()["href"]);
```

Die Verwaltung der unterschiedlichen Zeilenenden im Quellcode übernahm Git.

Kodierung Die Kodierungsfunktionen von Qt lieferten entgegen der Dokumentation nicht auf beiden Systemen die angestrebten Ergebnisse.

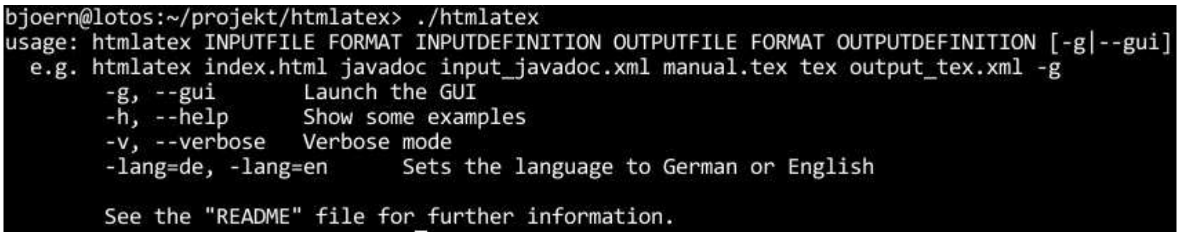
Wurde ein `QTextStream` mit `setCodec("UTF-8")` festgelegt, konnte er UTF-8 Text mit und ohne BOM[14] nicht korrekt verarbeiten. Erst die auf einen `QString` angewandte Methode `toLatin1()` konnte plattformübergreifend die von uns

[13] Dolphin® ist ein eingetragenes Markenzeichen der Dolphin Computer Access Ltd. http://www.yourdolphin.com/

[14] Byte Order Mark

genutzte UTF-8 Auswahl korrekt darstellen. Uns ist bewusst, dass die Latin1-Codierung in diesem Bereich weitestgehend deckungsgleich ist, dies aber für größtmögliche Flexibiltät wohl nur ein Workaround ist. Bei der Verarbeitung der Übersetzungsdateien für dieses Projekt mittels `QTextCodec` und der Methode `codecForName("utf8")` trat dieses seltsame Verhalten nicht auf.

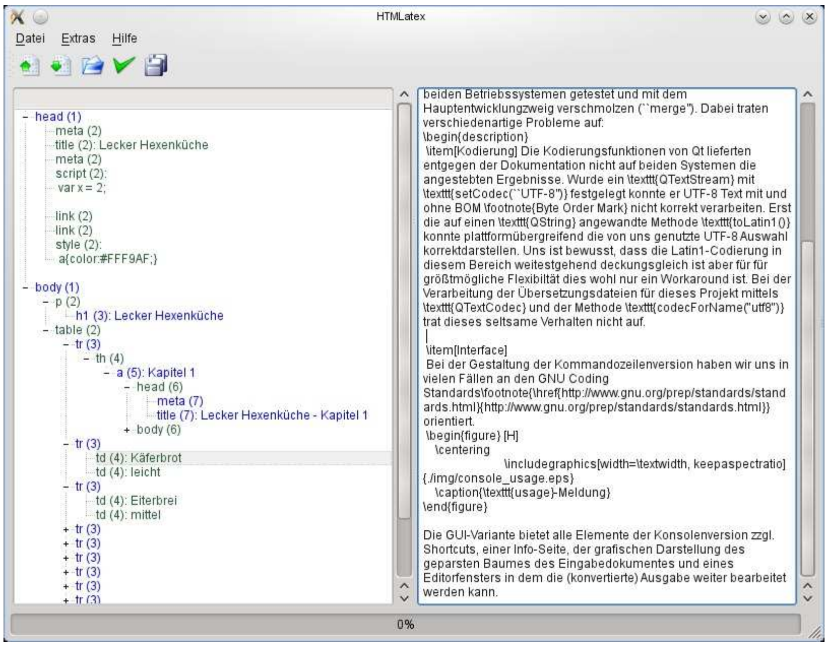
Interface Bei der Gestaltung der Kommandozeilenversion haben wir uns in vielen Fällen an den GNU Coding Standards[15] orientiert.



```
bjoern@lotos:~/projekt/htmlatex> ./htmlatex
usage: htmlatex INPUTFILE FORMAT INPUTDEFINITION OUTPUTFILE FORMAT OUTPUTDEFINITION [-g|--gui]
  e.g. htmlatex index.html javadoc input_javadoc.xml manual.tex tex output_tex.xml -g
        -g, --gui       Launch the GUI
        -h, --help      Show some examples
        -v, --verbose   Verbose mode
        -lang=de, -lang=en      Sets the language to German or English

        See the "README" file for further information.
```

Abbildung 1: GUI



```
- head (1)
    meta (2)
    title (2): Lecker Hexenküche
    meta (2)
    script (2):
     var x = 2;

    link (2)
    link (2)
    style (2):
     a{color:#FFF9AF;}

- body (1)
  - p (2)
       h1 (3): Lecker Hexenküche
  - table (2)
     - tr (3)
        - th (4)
           - a (5): Kapitel 1
              - head (6)
                   meta (7)
                   title (7): Lecker Hexenküche - Kapitel 1
              + body (6)
     - tr (3)
          td (4): Käferbrot
          td (4): leicht
     - tr (3)
          td (4): Eiterbrei
          td (4): mittel
     + tr (3)
     + tr (3)
     + tr (3)
     + tr (3)
     + tr (3)
     + tr (3)
```

```
beiden Betriebssystemen getestet und mit dem
Hauptentwicklungzweig verschmolzen (``merge"). Dabei traten
verschiedenartige Probleme auf:
\begin{description}
 \item[Kodierung] Die Kodierungsfunktionen von Qt lieferten
entgegen der Dokumentation nicht auf beiden Systemen die
angestebten Ergebnisse. Wurde ein \texttt{QTextStream} mit
\texttt{setCodec(``UTF-8")} festgelegt konnte er UTF-8 Text mit und
ohne BOM \footnote{Byte Order Mark} nicht korrekt verarbeiten. Erst
die auf einen \texttt{QString} angewandte Methode \texttt{toLatin1()}
konnte plattformübergreifend die von uns genutzte UTF-8 Auswahl
korrektdarstellen. Uns ist bewusst, dass die Latin1-Codierung in
diesem Bereich weitestgehend deckungsgleich ist aber für für
größtmögliche Flexibiltät dies wohl nur ein Workaround ist. Bei der
Verarbeitung der Übersetzungsdateien für dieses Projekt mittels
\texttt{QTextCodec} und der Methode \texttt{codecForName("utf8")}
trat dieses seltsame Verhalten nicht auf.
 |
 \item[Interface]
 Bei der Gestaltung der Kommandozeilenversion haben wir uns in
vielen Fällen an den GNU Coding
Standards\footnote{\href{http://www.gnu.org/prep/standards/stand
ards.html}{http://www.gnu.org/prep/standards/standards.html}}
orientiert.
 \begin{figure} [H]
   \centering
                \includegraphics[width=\textwidth, keepaspectratio]
{./img/console_usage.eps}
   \caption{\texttt{usage}-Meldung}
\end{figure}

Die GUI-Variante bietet alle Elemente der Konsolenversion zzgl.
Shortcuts, einer Info-Seite, der grafischen Darstellung des
geparsten Baumes des Eingabedokumentes und eines
Editorfensters in dem die (konvertierte) Ausgabe weiter bearbeitet
werden kann.
```

Abbildung 2: GUI-Variante

Die GUI-Variante bietet alle Elemente der Konsolenversion zzgl. Shortcuts, einer Info-Seite, einem Fortschrittsbalken, der grafischen Darstellung des ge-

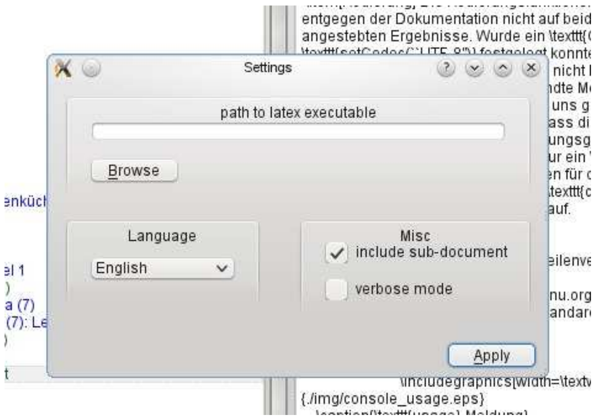
[15]http://www.gnu.org/prep/standards/standards.html

parsten Baumes des Eingabedokumentes und eines Editorfensters in dem die (konvertierte) Ausgabe weiter bearbeitet werden kann.

Dank des Dokument-Generators Doxygen ist eine Einbindung der Entwicklerdokumentation dieses Projektes als “Hilfe”-Komponente kein großer Aufwand mehr, da direkt Qt Compressed Help (.qch) ausgegeben werden kann.

Einstellungen Die Einstellungen werden in der Datei `htmlatex_settings.dat` binär serialisiert.

Lokalisierung Zu übersetzende Strings wurden mittels der Qt-Funktion `tr("str")` markiert, mit dem Programm `lupdate` in ein XML-Zwischenformat (.ts) extrahiert und mithilfe des `QtLinguist` für deutsch und englisch übersetzt. Die Übersetzungen werden dann mittels des Programms `lrelease` in das binäre .qm-Format (Qt Message File Format) überführt, das auf hohe lookup-Geschwindigkeit optimiert ist und von der Programmdatei direkt genutzt wird. Fehlt die gewünschte Sprachdatei, werden die im Quelltext direkt geschriebenen Strings ausgegeben.



path to latex executable

Language: English

Misc
[✓] include sub-document
[ ] verbose mode

Abbildung 3: Einstellungen

externer Programmaufruf Der Pfad zu LaTeXProgrammdatei kann ebenfalls in den Einstellungen eingegeben werden. Für Linux könnte auch ein Aufruf des Kommandos `which` Voreinstellungen liefern.

3. Programmablauf

Es wurde die im Pflichtenheft visualisierte Basis für die Konvertierung angestrebt. Hierbei haben wir uns für einen XML-DOM[16]-Baum für die interne Verarbeitung festgelegt.

[16]Document Object Model

1. Das Eingabeformat steht – im Falle von JavaDoc, wie es z.B. für Java unter http://download.oracle.com/javase/6/docs/api/overview-summary.html zu finden ist – leider nicht in einem XML konformen Format bereit. Um es hierin zu überführen, nimmt ein dafür geschriebener Präprozessor die nötigen Umformungen vor. Er wird in einer Hook-Methode vor dem Einlesevorgang ausgeführt und ist leicht zu ersetzen. Er wird aktiv, wenn als Eingabeformat *JavaDoc* ausgewählt wird.

2. Nun beginnt der Einlesevorgang mithilfe der Eingabe-Definitionsdatei im XML-Format. Die einzelnen Elemente der Datei werden als Knoten im QDomDocument-Baum gesetzt. Und in der GUI-Version grafisch dargestellt. Per Kontextmenü sollen die Knoten, ihre Inhalte und Attribute manipuliert werden. Dies ist jedoch noch nicht implementiert.

3. Auf diesem Baum wird beim anschließenden Konvertieren gearbeitet. Es wird zuerst ein Blatt gesucht, dann dessen Inhalt mithilfe der Ausgabe-Definitionsdatei ersetzt und in den ebenso ersetzten Inhalt des Elternknoten an die Position eines Markers geschrieben und das Blatt anschließend entfernt. Wenn dieser Elternknoten keine weiteren Kinder mehr hat, wird der Marker auch entfernt.

   Die `requires`- und `location`-Knoten der Ausgabe-Definitionsdatei dienen dazu spezielle Anforderungen (z.B. eine bestimmte Position) für den Ausgabestring zu bestimmen. Im Falle von LaTeXist z.B. eine `usepackage`-Anweisung möglich, die bestimmte Pakete im Bedarfsfall nachlädt und dann in den Prolog der Datei gehört. Nun wird der Vorgang so of wiederholt bis der Baum auf einen Knoten reduziert ist und den gesamten String enthält.

   Um auch sehr große Bäume abarbeiten zu können müsste dieser Algorithmus auf kleinere Teilbäume angewendet werden, die danach einzeln die Ausgabe zusammensetzen, um keine Engpässe zu produzieren.

   Leider konnte dieser Schritt bis zu diesem Zeitpunkt noch nicht fertiggestellt werden.

4. In der GUI-Variante kann nun der übersetzte String nachträglich bearbeitet und abgespeichert werden.

5. Wird als Ausgabeformat PDF[17] gewählt wird das Programm *latex* aufgerufen, um diese Datei in DVI[18] zu übersetzen. Wird eine `'rerun to get crossreferences right'` zurückgegeben, wird dieser Vorgang wiederholt. Es würden noch zwei weitere externe Programmaufrufe folgen(`dvips` und schließlich `ps2pdf`).

[17]Portable Document Format
[18]Device independent file format

4. Reflexion

Die Wahl eine verteilten Versionskontrollsystems hat sich über ihren Hauptzweck hinaus als überaus hilfreich erwiesen, sei es um die Problematik der Zeilenenden zu lösen, das Finden von Fehlern (als Diff zur letzten funktionierenden Version) zu erleichtern oder das Arbeiten an verschiedenen Entwicklungszweigen zu ermöglichen und deren Zusammenführung sehr zu erleichtern. Allerdings ist der Lernaufwand im Einstieg nicht eben gering.

Die Wahl von Qt als C++ Klassenbibliothek hat sich bei der GUI-Entwicklung für beide Plattformen als hilfreich, ab auch manchmal als tückisch erwiesen: So haben wir mit der Fehlersuche bei scheinbar selbstverständlichen Dingen – wie der UTF-8 Kodierung – unnötig Zeit verloren. Hier wäre für so ein Projekt eine intensivere Einarbeitung im Vorfeld vonnöten gewesen.

Die Zusammenarbeit war sehr eng und neben den üblichen Wegen, wie z.B. direkte Zusammenarbeit, E-Mails, Telefon durch das Ticketsystem mithilfe von Newsfeeds auch sehr schnell und verlässlich.

5. Ausblick

Neben Verbesserungen am Interface ist es nun interessant zu überprüfen, ob das Programm in der Lage ist, in der Praxis mit mehr Formaten erfolgreich zu arbeiten und Aussenstehende zu gewinnen, die Formatdefinitonen in XML-Form anpassen.

Zur Senkung des Aufwandes wäre es wohl sehr wünschenswert, wenn dieses z.B. mittels einer Eingabemaske zum Bearbeiten und Erstellen dieser Beschreibungsdateien geschähe. Darüber hinaus würde ein Syntaxhighlighting Tippfehler weiter minimieren.

Die Anpassungen, die der Präprozessor vorzunehmen hat, damit das Eingabeformat XML-Konform wird, könnten auch anpassbar in XML definiert werden.

Der Editor für die Ausgabedateien sollte neben *Suchen&Ersetze*n und einer Schriftgrößen-Einstellung die üblichen Textbearbeitungswerkzeuge zum *Ausschneiden*, *Kopieren* und *Einfügen*, so wie für *Rückgängig* und *Wiederholen* bekommen. `undo()`, `redo()`, `cut()`, `copy()`, `paste()` und `setFontPointSize()` sind im *QTextEdit*-Widget schon enthalten und deshalb auch leicht nachzurüsten.

Die Baumansicht des Eingabedokumentes sollte per Kontextmenü einen speziellen Knoten oder Unterbaum vor dem Übersetzungsvorgang entfernen können, um in Dokumentenstrukturen mit vielen Unterdokumenten die Auswahl begrenzen zu können.

Ebenso ist es erstrebenswert, die maximal zu verfolgende Linktiefe begrenzen zu können und Linkschleifen zu vermeiden.

A. Listings

A.1. console

Listing 2: `console.h`

```cpp
#ifndef CONSOLE_H
#define CONSOLE_H

#include "translationmapper.h"
#include "documentreader.h"
#include "converter.h"
#include <QObject>
#include <QString>
#include <iostream>

/** This class provides the command line functionality of the application.
  * @author Bjoern
  */
class Console : public QObject
{
    Q_OBJECT
public:
    /** This is the only constructor.
        @param <arguments> is an array of strings which contains the startup arguments
        of the application except the executable file's name and optional paramters.
        @param <options> is an array of strings which contains just the optional
        startup arguments of the application.
        @author Bjoern
      */
    Console(QStringList arguments, QStringList options);
private:
    /** This method processes the application's startup arguments and performs the
        conversion by means of a DocumentReader and a Converter object.
        @param <arguments> is an array of strings which contains the startup arguments
        of the application except the executable file's name and optional paramters.
        @param <options> is an array of strings which contains just the optional
        startup arguments of the application.
        @author Bjoern
      */
    void _performInitialOperations(QStringList arguments, QStringList options);
};

#endif // CONSOLE_H
```

Listing 3: `console.cpp`

```cpp
#include "console.h"

Console::Console(QStringList arguments, QStringList options)
{
    _performInitialOperations(arguments, options);
};
/** This method processes the application's startup arguments and performs the
    conversion by means of a DocumentReader and Converter object.
    @author Bjoern
  */
void Console::_performInitialOperations(QStringList arguments, QStringList options)
```

```
{
    /* arguments:
       0 = source file path
       1 = source file type
       2 = input definition file path
       3 = target file path
       4 = target file type
       5 = output definition file path
    */
    if (arguments.count() == 6)
    {
        TranslationMapper* translationmapper = new TranslationMapper;
        QString sourcefilepath(arguments.at(0));
        QString filetypestring(arguments.at(1));
        QString inputdefinitionfilepath(arguments.at(2));
        translationmapper->createDocumentReaderData(inputdefinitionfilepath);
        QFile file(sourcefilepath);
        Node* root = 0;
        if (file.exists())
        {
            DocumentData::FileType filetype = DocumentData::Unknown;
            if (filetypestring.toLower() == "javadoc")
                filetype = DocumentData::JavaDocHTML;
            DocumentReader* reader = new DocumentReader(translationmapper);
            root = reader->read(sourcefilepath, filetype);
            delete reader;
            std::cout << tr("Read source file(s) --> success").toStdString() << std::endl;
        }
        else
        {
            file.close();
            std::cout << tr("I/O error - the source file doesn't exit: ").toStdString()
                    << sourcefilepath.toStdString() << std::endl;
            return;
        }
        file.close();
        QString targetfilepath(arguments.at(3));
        filetypestring = QString(arguments.at(4));
        QString outputdefinitionfilepath(arguments.at(5));
        translationmapper->createOutputElementMap(outputdefinitionfilepath);
        file.setFileName(targetfilepath);
        if (file.open(QFile::WriteOnly))
        {
            DocumentData::FileType filetype = DocumentData::Unknown;
            if (filetypestring.toLower() == "tex")
                filetype = DocumentData::Tex;
            else if (filetypestring.toLower() == "pdf")
                filetype = DocumentData::PDF;
            // root - converting...
            Converter* converter = new Converter(this, translationmapper);
            converter->convert(targetfilepath, root, filetype);
            std::cout << tr("Perform conversion --> success").toStdString() << std::endl;
            delete converter;
        }
        else
            std::cout << tr("I/O error - can't write to file: ").toStdString()
                    << targetfilepath.toStdString() << std::endl;
        delete translationmapper;
    }
    else
    {
        std::cout << tr("Error - unexpected number of arguments.").toStdString() << std::endl;
        std::cerr << "usage: htmlatex INPUTFILE FORMAT INPUTDEFINITION OUTPUTFILE FORMAT ↘
            →OUTPUTDEFINITION [-g|--gui]\n"
```

```
                << "  e.g. htmlatex index.html javadoc input_javadoc.xml manual.tex tex ↘
                    →output_tex.xml -g\n"
                << "\t-g, --gui \tLaunch the GUI\n"
                << "\t-h, --help \tShow some examples\n\n"
                << "\tSee the \"README\" file for further information." << std::endl;
    }
};
```

A.2. converter

Listing 4: `converter.h`

```
#ifndef CONVERTER_H
#define CONVERTER_H

#include "node.h"
#include "translationmapper.h"
#include "settings.h"
#include "documentdata.h"
#include <QFile>
#include <QString>
#include <QTextStream>
#include <QStringList>
#include <QDir>
#include <QObject>
#include <iostream>

/** This class converts the QDomDocument-Tree to a textfile
  * by performing the conversions descibed in the 'Output-Definition'-XML file.
  * @author Bjoern
  * @version 0.1
  */
class Converter : public QObject
{
    Q_OBJECT
signals:
    void updateTextEdit(QString text);
    void updateProgressBar(int percentage);
public:
    // TODO take parameter for the output-XML-file
    /** Constructor
      * @param <filepath> where to write the output
      * @param <root> the root-node of the QDomDocument to convert from
      * TODO: @param <parts> the number of parts that are needed for the conversion
      * use enum{start=0, content=1, end=2} (change the XML aliases to numbers)
      */
    Converter(QObject* parent, TranslationMapper* translationmapper);
    /** converts to QString and writes the output file
      * @param <qint32> the number of parts in the outputDocument
      */
    void convert(const QString filepath, Node* root, DocumentData::FileType filetype);
private:
    /*****FIELDS*****/
    /** outputfilestream */
    QTextStream _stream;
    /** the root node */
    Node * _root;
    TranslationMapper* _translationMapper;
    /** the active node */
```

```cpp
    static Node * _cursor;
    QString _errormessage;
    qint32 _noOfParts;
    QStringList replacementMarks;

    /** the compiler assigns values for start, content and end */
    enum DocumentPosition { start, content, end };

    // QStringList-Beispiele: _parts << "meinString"; _parts.insert(0, "ersterString");
    // _parts.append("meinString") entspricht dem "<<"-operator
    QStringList _parts;

    /*****METHODS*****/
    /** retrieves the first leaf of the tree */
    Node * getLeaf(Node* node = _cursor);
    /** Is the node a leaf? */
    bool isLeaf(Node * node = _cursor);
    /** returns the node's attributes */
    QMap<QString,QString> getAttributes(Node * node = _cursor);
    /** returns the node's content */
    QString getContent(Node * node = _cursor);
    /** returns the node's name */
    QString getName(Node * node = _cursor);
    /** Writes the information of the current (leaf) node into the parent, deletes
    * the active node and sets the cursor to the next leaf. */
    bool consume(Node * node = _cursor);
    int match(QString content, QString pattern);

    /** gets the next sibling */
    Node * _getNextSibling();
    /** gets the next child */
    Node * _getNextChild();
    /** get the next node in the tree
    * if no siblings left it gets the next child */
    Node * _getNextNode();
    /** get the Level ot the current node */
    qint64 _getTreeLevel();
    /** gets the key of the parent node <distance> levels up the tree */
    QString _peekParent(qint32 distance);
    /** true if string is empty */
    QBool _isEmpty();
    /** true if content is empty */
    QBool _isEmptyContent();
    /** removes tokens e.g. "----CONTENT----" */
    void removeToken();
    /** true if started as GUI
     * only necessary if the view(Editor part of the GUI)
     * or a progressbar has to receive extra information
     */
    //QBool isGUI();

    /** delete node and set cursor to the next node */
    void _consume();
    /** writes <content> at the end of the desired <part> */
    void _write(QString content, qint32 part);
    /** returns offset if <pattern> is found otherwise returns "-1" */
    qint64 _tryMatch(QString pattern);
    /** replaces place holder with the content of the current node */
    QString _replace(Node* node = _cursor);
};

#endif // CONVERTER_H
```

Listing 5: `converter.cpp`

```cpp
#include "converter.h"

Converter::Converter(QObject *parent, TranslationMapper* translationmapper)
    : QObject(parent), _errormessage(""), _noOfParts(0)
{
    _root = 0;
    _translationMapper = translationmapper;
    replacementMarks << "----CONTENT----" << "----TEXT----";

    /* test match()
    QString cont = "0123Justus456789";
    QString pattern = "Justus";
    int i = match(cont, pattern);
    std::cerr << "Converter: i: " << QString::number(i).toStdString() << "\n";
    end test */
};
Node* Converter::_cursor = 0;
void Converter::convert(const QString filepath, Node* tree, DocumentData::FileType filetype)
{
    if (!tree)
    {
        std::cerr << tr("error - CConverter.convert() : tree == 0").toStdString() << std::endl;
        return;
    }
    _cursor = getLeaf(tree);
    int nodecount = (int)tree->getTreeNodeCount();
    int i = 0;
    while (consume())
    {
        if (Settings::DEBUG)
        {
            //std::cerr << "CConverter.convert() - while():\n\ti: " << i << std::endl;
        }
        i++;
        emit updateProgressBar((int)((double)i / (double)nodecount * 100.0));
    }
    std::cerr << "Converter.convert(): cursor.count: " << _cursor->getCount() << std::endl;
    if (Settings::DEBUG)
        std::cerr << "Content: " << _cursor->getContent().toStdString() << std::endl;

    /*
    QString convertedtext(tree->content());
    for (int i = 0; i < _parts.count(); i++)
        convertedtext += _parts.at(i);
        */

    // ERNSTER CODE !!!
    // save conversion output to file
    if (filetype == DocumentData::Tex)
    {
        QFile file(filepath);
        if (!file.open(QFile::WriteOnly | QFile::Text))
        {
            if (Settings::DEBUG)
            {
                std::cerr << tr("MainWindow._saveAs() - can't write to file: path: ").toStdString()
                        << filepath.toStdString() << std::endl;
            }
        }
        else
        {
            QTextStream stream(&file);
            stream.setCodec("UTF-8");
            stream << _cursor->getContent().toLatin1();
            file.close();
            if (Settings::DEBUG)
                std::cerr << tr("Converter.convert() saved to TEX file").toStdString() << std::endl;
        }
    }
    else if (filetype == DocumentData::PDF)
    {
        Settings settings;
        // invoke external program and write the output to a file
        QString command = settings.getValue("latexpath") + " " + filepath;
        if (Settings::DEBUG)
        {
            std::cerr << tr("Converter.convert() save to PDF:\ncommand: ").toStdString()
                    << command.toAscii().data() << std::endl;
        }
        int errorcode = system(command.toAscii().data());
        if (errorcode == 0)
        {
            if (Settings::DEBUG)
                std::cerr << tr("Converter.convert() saved to PDF file").toStdString() << std::endl;
        }
        else
        {
            if (Settings::DEBUG)
                std::cerr << tr("Converter.convert() en error occurred saving to PDF file: error code ").toStdString()
                << QString::number(errorcode).toStdString() << std::endl;
        }
    }
    else
        std::cerr << tr("Converter.convert(): file not saved - unknown file type").toStdString() << std::endl;
    emit updateTextEdit(_cursor->getContent());
    emit updateProgressBar(0);
};
Node * Converter::getLeaf(Node* node)
{
    Node * result = node;
    while (result->firstChild() != 0)
        result = result->firstChild();
    return result;
};
bool Converter::consume(Node * node)
{
    Node *parent = node->getParent();
    if (!parent)
    {
        if (Settings::DEBUG)
        {
            std::cerr << "Converter.consume(): root node found. node.getName(): "
                    << node->getName().toStdString() << std::endl;
        }
        return false;
    }
    QString parentcontent = parent->getContent();//_replace(parent);
    //std::cerr << "parentcontent: " << parentcontent.toStdString() << std::endl;
    QString childcontent = _replace();
    for (int i = 0; i < replacementMarks.count(); i++)
    {
        int index = match(parentcontent, replacementMarks.at(i));
        //std::cerr << "index: " << index << std::endl;
        if (index > 0)
        {
            if (replacementMarks.at(i) == "----CONTENT----")
                parentcontent.insert(index, childcontent);
            else if (replacementMarks.at(i) == "----TEXT----")
                parentcontent.insert(index, childcontent);
            // else: eventuell fuer weitere marker
        }
        else
            parentcontent.append(childcontent);
    }
    parent->setContent(parentcontent);
    //std::cerr << "node: " << node->getName().toStdString() << " parentcontent: " << ↘
        →parentcontent.toStdString() << std::endl;
    parent->removeChild(node);
    _cursor = getLeaf(parent);
    return true;
};
bool Converter::isLeaf(Node * node)
{
    return (node->getCount() == 0);
};
QMap<QString,QString> Converter::getAttributes(Node * node)
{
    return node->getAttributes();
};
QString Converter::getContent(Node * node)
{
    return node->getContent();
};
QString Converter::getName(Node * node)
{
    return node->getName();
};
Node * Converter::_getNextSibling()
{
    // Demo-Code
    Node* result = 0;
    Node* parent = _cursor->getParent();
    int index = parent->indexOf(_cursor);
    // next sibling's index is (index + 1)
    if (index + 1 < parent->getCount())
        result = parent->childAt(index + 1);
    // _cursor = result; ?
    return result;
};
Node * Converter::_getNextChild()
{
    // Demo-Code
    return _cursor->nextChild();
};
Node * Converter::_getNextNode()
{
   // if cursor has sibling return _getNextSibling
   // elseif cursor has children return _getNextChild
    //else
    return 0;
};
qint64 Converter::_getTreeLevel()
{
    // Demo-Code
    // return the greatest distance of a node to the root node
    return _root->getTreeLevel();
};
```

```
QString Converter::_peekParent(qint32 distance)
{
    // if distance < current->layer()
    //   return the parent node <distance> hops up the tree
    // Demo-Code
    return QString("");
};
QBool Converter::_isEmpty()
{
    // Demo-Code
    return QBool(false);
};
QBool Converter::_isEmptyContent()
{
    // Demo-Code
    return QBool(_cursor->getContent().isEmpty());
};
void Converter::removeToken()
{
};
void Converter::_write(QString content, qint32 part)
{
};
qint64 Converter::_tryMatch(QString pattern)
{
    // Demo-Code
    return 0;
};
QString Converter::_replace(Node* node)
{
    QString result("");
    TranslationData data = _translationMapper->outputMap()[node->getName()];
    QString to = data.to();
    //std::cerr << "to: " << "?"<< to.toStdString() << "?" << std::endl;
    //std::cerr << "content: " << node->content().toStdString() << std::endl;
    QList<TranslationDataNode> datanodes = data.requires();
    TranslationDataNode datanode;
    for (int j = 0; j < datanodes.count(); j++)
    {
        datanode = datanodes.at(j);
        //datanode.name()
    }
    for (int i = 0; i < replacementMarks.count(); i++)
        result = to.replace(replacementMarks.at(i), node->getContent());
    //std::cerr << "_replace: result: " << result.toStdString() << std::endl;
    return result;
};
int Converter::match(QString content, QString pattern)
{
    if (!content.isEmpty())
        return content.indexOf(pattern, 0, Qt::CaseSensitive);
    return -1;
};
```

A.3. documentdata

Listing 6: `documentdata.h`

```
#ifndef DOCUMENTDATA_H
#define DOCUMENTDATA_H

#include "node.h"
#include "settings.h"
#include <QObject>
#include <QString>
#include <QFile>
#include <QFileInfo>
#include <QTextStream>
#include <QStringList>
#include <QMessageBox>
#include <iostream>

/** This class provides a well-formed XML representation of a single input document.
    @author Bjoern
  */
class DocumentData : public QObject
{
    Q_OBJECT
public:
    enum FileType { JavaDocHTML, HTML, Tex, PDF, Unknown };
    /** This is the only constructor. It returns an object created from a QFileInfo
        object which holds detailed information about a single input document such
        as its absolute file path. The text() method returns a well-formed XML
        representation of the input document taking the file type into account.
        The application's internal tree structure is supposed to represent a whole
        input document including its subdocuments. So a Node object is used by
        the DocumentReader to determine where to add the subdocument represented
        by this class (DocumentData).
        @param <fileinfo> holds detailed information about the input document.
        @param <node> is the corresponding parent node in the application's
        internal tree structure.
        @param <filetype> indicates the file typ of the input document (i.e.
        JavaDoc HTML).
        @author Bjoern
      */
    DocumentData(QFileInfo fileinfo, Node* node, FileType filetype);
private:
    /** This attribute contains a well-formed XML representation of the underlying
        input document.
        @author Bjoern
      */
    QString _text;
    /** This attribute holds detailed information about the input document.
        @author Bjoern
      */
    QFileInfo _fileInfo;
    /** This attribute contains the file type of the input document (i.e. JavaDoc
        HTML).
        @author Bjoern
      */
    FileType _fileType;
    /** The application's internal tree structure is supposed to represent a whole
        input document including its subdocuments. So the Node object is used by
        the DocumentReader to determine where to add the subdocument represented
        by this class (DocumentData).
        @author Bjoern
      */
    Node* _node;
    /** This attribute indicates whether or not the input document has already
        been preprocessed by the appropriate preprocessing method.
        @author Bjoern
      */
    bool _preprocessed;
    /** This hook-method calls all preprocessing methods.
        @author Bjoern
      */
    void _preprocessingHook();
    /** This method changes the specified HTML-file in order to gain well-formed
        XML (XHTML).
        @author Bjoern
      */
    void _preprocessHTML();
public:
    /** This method returns a well-formed XML representation of the underlying
        input document.
        @author Bjoern
      */
    QString text();
    /** This method returns a QFileInfo object which holds detailed information
        about the input document.
        @author Bjoern
      */
    QFileInfo fileInfo() const;
    /** The application's internal tree structure is supposed to represent a whole
        input document including its subdocuments. So the Node object is used by
        the DocumentReader to determine where to add the subdocument represented
        by this class (DocumentData).
        @author Bjoern
      */
    Node* node() const;
};

#endif // DOCUMENTDATA_H
```

Listing 7: documentdata.cpp

```
#include "documentdata.h"

DocumentData::DocumentData(QFileInfo fileinfo, Node* node, FileType filetype)
    : _text(""), _fileInfo(fileinfo), _preprocessed(false)
{
    this->_fileType = filetype;
    this->_node = node;
};
void DocumentData::_preprocessingHook()
{
    _preprocessHTML();
};
void DocumentData::_preprocessHTML()
{
    if (_fileType != DocumentData::JavaDocHTML)
        return;
    QString path = _fileInfo.filePath();
    QFile file(path);
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        if (Settings::DEBUG)
        {
            std::cerr << tr("DocumentData._preprocessHTML() : can't preprocess file path: ↘
                ↪").toStdString()
                      << path.toStdString() << std::endl;
        }
        return;
    }
    if (Settings::DEBUG)
    {
        std::cerr << tr("DocumentData._preprocessHTML() : preprocessing file path: ").toStdString()
                << path.toStdString() << std::endl;
    }
    _text = QString(file.readAll()).toLatin1();
    file.close();
    // processing document
    // add missing finalizing slashes to empty elements
    // the following two lines create an array of QString objects
    QStringList elements;
    elements << "br" << "img" << "hr" << "meta" << "link";
    QRegExp regex;
    foreach (QString element, elements)
    {
        regex = QRegExp("<" + element + // opening tag and tag name
                        "([^\\/>])*" // consume all characters except '/' and '>'
                        "(?!\\/)>"); // the character '/' is missing before '>'
        // the empty element stored in the iteration variable 'element' is found
        // without the character '/'
        while(regex.indexIn(_text, 0) >= 0)
        {
            // insert missing '/'
            // regex.indexIn() returns the found position and
            // regex.cap(0).count() is the number of characters of mathed string
            _text.insert(regex.indexIn(_text, 0) + regex.cap(0).count() - 1, "/");
        }
    }
    _preprocessed = true;
};
/**
  * @author Bjoern
  */
QString DocumentData::text()
{
    // the preprocessed document is stored in the attribute "_text"
    if (!_preprocessed)
        _preprocessingHook();
    return _text;
};
QFileInfo DocumentData::fileInfo() const
{
    return this->_fileInfo;
};
Node* DocumentData::node() const
{
    return this->_node;
};
```

A.4. documentreader

Listing 8: documentreader.h

```
#ifndef DOCUMENTREADER_H
#define DOCUMENTREADER_H

#include "node.h"
#include "documentdata.h"
```

```cpp
#include "translationmapper.h"
#include "settings.h"
#include <QObject>
#include <QString>
#include <QDomDocument>
#include <QDomNamedNodeMap>
#include <QMapIterator>
#include <QStack>
#include <QDir>
#include <iostream>

/** The DocumentReader class creates the application's internal tree structure
    from a specified input document and is able to include referenced subdocuments.
    @author Bjoern
  */
class DocumentReader : public QObject
{
    Q_OBJECT
public:
    /** This is the only constructor.
        @param <translationmapper> is used to obtain a DocumentReaderData object
        which mainly describes how a reference to a another document is defined in
        the input document.
        @author Bjoern
      */
    DocumentReader(TranslationMapper* translationmapper);
    /** This method creates a Node tree structure which corresponds to the whole
        input document including its subdocuments if desired
        (_includeSubDocuments is set). The nodes of each document are processed
        recursively by the _readElement() method. Whereas the documents
        (subdocuments) are processed in an iteration loop by means of the
        _documentStack.
        @param <indexfilepath> contains the absolute file path to the index document.
        @param <filetype> indicates of which type the input document is.
        @author Bjoern
      */
    Node* read(QString indexfilepath, DocumentData::FileType filetype);
private:
    /** The _translationMapper provides two data objects: firstly a
        DocumentReaderData object which is used by the DocumentReader and
        secondly a key-value-pair structure which is used by the Converter.
        @author Bjoern
      */
    TranslationMapper* _translationMapper;
    /** This attribute provides detailed information about the input index file
        such as absolute and relative path etc..
        @author Bjoern
      */
    QFileInfo _indexFileInfo;
    /** This attribute indicates of which type the input document is.
        @author Bjoern
      */
    DocumentData::FileType _fileType;
    /** This stack is populated with DocumentData objects when processing the
        input document. The usage of a stack allows to process the input document
        and its subdocuments in an iteration loop instead of a recursion because
        a recursion may raise a stack overflow issue.
        @author Bjoern
      */
    QStack<DocumentData*> _documentStack;
    /** This attribute indicates whether or not subdocuments are included by the
        input document processing.
        @author Bjoern
      */
    bool _includeSubDocuments;
    /** This method processes a single document recursively. It adds for each node
        of the input document a corresponding Node object to the application's
        internal tree structure. If a reference to a subdocument is found a new
        DocumentData object is generated and pushed on the _documentStack.
        @param <element> is the current parent QDomElement object. Notice that the
        _readElement() method performs a recursion to process a single document.
        @param <node> is a pointer pointing to the very Node object of the
        application's internal tree structure to which currently generated Node
        objects will be added. Notice that the _readElement() method performs a
        recursion to process a single document.
        @author Bjoern
      */
    void _readElement(QDomElement element, Node* node);
};

#endif // DOCUMENTREADER_H
```

Listing 9: `documentreader.cpp`

```
#include "documentreader.h"

DocumentReader::DocumentReader(TranslationMapper* translationmapper)
{
    _fileType = DocumentData::Unknown;
    _translationMapper = translationmapper;
    Settings settings;
    _includeSubDocuments = (bool)settings.getValue("includesubdocuments").toInt();
};
/** This method starts by pushing a DocumentData object created from the index
    document to the _documentStack. Then the iteration loop is started which ends
    when all DocumentData objects are popped from the _documentStack. Each
    DocumentData object contains a well-formed XML representaion of the underlying
    input document. Additionally it provides a pointer to the node of the
    application's internal tree structure to which the document will be added to.
    This XML representation is passed to a QDomDocument object which validates it
    and retrieves the root element. Then the _readElement() method processes the
    document with the given root element.
    @author Bjoern
  */
Node* DocumentReader::read(QString indexfilepath,
                           DocumentData::FileType filetype)
{
    if (_translationMapper == 0)
    {
        if (Settings::DEBUG)
            std::cerr << tr("DocumentReader.read() : _translationMapper is 0").toStdString() << ↘
                 →std::endl;
        return 0;
    }
    Settings settings;
    _includeSubDocuments = (bool)settings.getValue("includesubdocuments").toInt();
    _fileType = filetype;
    _indexFileInfo = QFileInfo(indexfilepath);
    // start reading the whole document tree
    Node* root = new Node(0, "html", 0);
    // add the index document to the stack of documents
    _documentStack.push(new DocumentData(_indexFileInfo, root, _fileType));
    // begin processing the documents stored on the document stack
    while(!_documentStack.isEmpty())
    {
        DocumentData* documentdata = _documentStack.pop();
        QDomDocument doc;
        QString errorStr = "";
        int errorLine = -1;
        int errorColumn = -1;
        if (doc.setContent(documentdata->text().toLatin1(),
                           false,
                           &errorStr,
                           &errorLine,
                           &errorColumn))
        {
            if (doc.documentElement().tagName().toLower() == "html")
                _readElement(doc.documentElement(), documentdata->node());
            else
            {
                std::cerr << tr("Error in DocumentReader::read()").toStdString()
                          << std::endl << tr("\tat \"if (doc.setContent())\" returned ↘
                             →false;").toStdString()
                          << std::endl << tr("\tFile name: ").toStdString()
                          << documentdata->fileInfo().filePath().toStdString()
                          << std::endl << tr("\tError message: <html>-tag not found").toStdString()
                          << std::endl;
            }
        }
        else
        {
            std::cerr << "Error in CDocumentReader::read()"
                      << std::endl << "\tat doc.setContent() returned false;"
                      << std::endl << "\tFile name: "
                      << documentdata->fileInfo().filePath().toStdString()
                      << std::endl << "\tError message: "
                      << std::endl << errorStr.toStdString() << " line="
                      << std::endl << QString::number(errorLine).toStdString()
                      << std::endl << "\tColumn=" << QString::number(errorColumn).toStdString()
                      << std::endl;
        }
        delete documentdata;
    }
    return root;
};
void DocumentReader::_readElement(QDomElement element, Node* node)
{
    for (int i = 0; i < element.childNodes().count(); i++)
    {
        // The QDomDocument class creates a node named "#text" which contains
        // the node value (content).
        if (element.childNodes().at(i).nodeName().toLower() == "#text")
            node->setContent(element.childNodes().at(i).nodeValue());
        else
        {
            Node* new_node = new Node(node,
                                      element.childNodes().at(i).nodeName().toLower(),
                                      node->getLayer() + 1);
            node->addChild(new_node);
            QDomNamedNodeMap attributes = element.childNodes().at(i).attributes();
            DocumentReaderData documentreference = _translationMapper->getDocumentReference();
            if (element.childNodes().at(i).nodeName().toLower() == documentreference.getTagName())
            {
                if (Settings::DEBUG)
                {
                    std::cerr << tr("#\tReadElement - '").toStdString()
                              << documentreference.getTagName().toStdString()
                              << tr("': \n#\t\tindexFileInfo.absPath: ").toStdString()
                              << _indexFileInfo.absolutePath().toStdString()
```

```
                                  << std::endl << tr("#\t\thref: ").toStdString()
                                  << new_node->getAttributes()["href"].toStdString() << std::endl;
                    }
                    QString urlattribute = documentreference.getUrlContainingAttributeName();
                    new_node->addAttribute(urlattribute, attributes.namedItem(urlattribute).nodeValue());
                    // compose absolute file path
                    QFileInfo myfileinfo;
                    if (QFileInfo(_indexFileInfo.absolutePath() + ↘
                        →new_node->getAttributes()["href"]).exists())
                        // unix
                        myfileinfo = QFileInfo(_indexFileInfo.absolutePath()
                                          + new_node->getAttributes()["href"]);
                    else
                        // windows
                        myfileinfo = QFileInfo(_indexFileInfo.absolutePath()
                                          + QDir::separator() + new_node->getAttributes()["href"]);
                    if (_includeSubDocuments)
                        _documentStack.push(new DocumentData(myfileinfo, new_node, _fileType));
                    if(Settings::DEBUG)
                    {
                        std::cerr << tr("# CDocumentReader::readElement()").toStdString()
                               << std::endl << tr("#\tat \"if (element.childNodes()").toStdString()
                               << tr(".at(i).nodeName().toLower() == \"a\")\" returned true").toStdString()
                               << std::endl << tr("#\tfound subdocument href=").toStdString()
                               << new_node->getAttributes()["href"].toStdString() << std::endl;
                    }
                }
                else
                {
                    QDomNode attribute;
                    for (int i = 0; i < attributes.count(); i++)
                    {
                        attribute = attributes.item(i);
                        new_node->addAttribute(attribute.nodeName(), attribute.nodeValue());
                    }
                }
                QDomElement new_element = element.childNodes().at(i).toElement();
                _readElement(new_element, new_node);
            }
        }
};
```

A.5. documentreaderdata

Listing 10: `documentreaderdata.h`

```
#ifndef DOCUMENTREADERDATA_H
#define DOCUMENTREADERDATA_H

#include <QString>

/** This class describes how a reference to a subdocument of an input document
    is defined. Example: In (X)HTML a subdocumet is referenced using the <a> tag
    with its attribute "href". The TranslationMapper creates an instance of
    the DocumentReaderData class using the document definition file. Finally the
    DocumentReader obtains the DocumentReaderData object from the
    TranslationMapper.
    @author Bjoern
  */
```

```
class DocumentReaderData
{
public:
    /** This is an empty constructor which initializes the attributes _tagName and
        _urlContainingAttributeName with empty strings.
        @author Bjoern
      */
    DocumentReaderData();
    /** This is the copy constructor.
        @author Bjoern
      */
    DocumentReaderData(const DocumentReaderData& documentreaderdata);
    /** This constructor initializes the attributes _tagName and
        _urlContainingAttributeName with the given string values.
        @param <tagname> contains a XML tag name.
        @param <urlcontainingattributename> contains a XML attribute name.
        @author Bjoern
      */
    DocumentReaderData(QString tagname, QString urlcontainingattributename);
private:
    /** This attribute contains the name of that XML tag which decribes a reference
        to a subdocument in the XML represented input document.
        @author Bjoern
      */
    QString _tagName;
    /** This attribute contains the name of that XML attribute belonging to the
        XML tag specified by _tagName and which holds the file path to a
        subdocument in the XML represented input document.
        @author Bjoern
      */
    QString _urlContainingAttributeName;
public:
    /** This method returns the name of that XML tag which decribes a reference
        to a subdocument in the XML represented input document.
        @author Bjoern
      */
    QString getTagName() const;
    /** This method returns the name of that XML attribute belonging to the
        XML tag specified by _tagName and which holds the file path to a
        subdocument in the XML represented input document.
        @author Bjoern
      */
    QString getUrlContainingAttributeName() const;
};

#endif // DOCUMENTREADERDATA_H
```

Listing 11: `documentreaderdata.cpp`

```
#include "documentreaderdata.h"

DocumentReaderData::DocumentReaderData() : _tagName(""),
                                           _urlContainingAttributeName("")
{
};
DocumentReaderData::DocumentReaderData(const DocumentReaderData& documentreaderdata)
{
    this->_tagName = documentreaderdata.getTagName();
    this->_urlContainingAttributeName = documentreaderdata.getUrlContainingAttributeName();
};
DocumentReaderData::DocumentReaderData(QString tagname, QString urlcontainingattributename)
```

```
{
    this->_tagName = tagname;
    this->_urlContainingAttributeName = urlcontainingattributename;
};
QString DocumentReaderData::getTagName() const
{
    return this->_tagName;
};
QString DocumentReaderData::getUrlContainingAttributeName() const
{
    return this->_urlContainingAttributeName;
};
```

A.6. itemdelegate

Listing 12: `itemdelegate.h`

```
#ifndef ITEMDELEGATE_H
#define ITEMDELEGATE_H

#include "model.h"
#include <QItemDelegate>
#include <QPainter>
#include <QPixmap>
#include <QBrush>

/** This class is derived from the QItemDelegate class. In this application its
    purpose is to control how the mainWindow's treeView displays data. The paint()
    method influences the drawing of the underlying widget directly. For further
    information look up QItemDelegate and QTreeView in the Qt documentation.
    @author Bjoern
  */
class ItemDelegate : public QItemDelegate
{
    Q_OBJECT
public:
    /** This is the only constructor.
        @param <model> is the data structure which serves the treeView of the
        mainWindow as data source. Because this class controls the way how data is
        actually drawn on the widget it needs access to the tree structure as well.
        @param <parent> may be specified to take advantage of Qt's destructor
        chaining mechanism which prevents memory leaks effectivly.
        @author Bjoern
      */
    ItemDelegate(Model* model = 0, QObject* parent = 0);
private:
    /** The model is the data source of the mainWindow's treeView. The Model class
        is derived from the abstract class QAbstractItemModel and provides the
        functionality that a QTreeView object demands in order to display data.
        @author Bjoern
      */
    Model* model;
    /** This is the outline color of the selected node.
        @author Bjoern
      */
    QColor colorFocusLine;
    /** This is the background color of the selected node.
        @author Bjoern
```

```
     */
    QColor colorFocusBackground;
    /** This is the color of a marked node.
        @author Bjoern
     */
    QColor colorMarked;
    /** This is the text color.
        @author Bjoern
     */
    QColor color;
    /** This is the text color of nodes whose _layer number is even.
        @author Bjoern
     */
    QColor colorLayer0;
    /** This is the text color of nodes whose _layer number is uneven.
        @author Bjoern
     */
    QColor colorLayer1;
public:
    /** This method sets the outline color of the selected node.
        @author Bjoern
     */
    void setColorFocusLine(QColor color);
    /** This method returns the outline color of the selected node.
        @author Bjoern
     */
    QColor getColorFocusLine() const;
    /** This method sets the background color of the selected node.
        @author Bjoern
     */
    void setColorFocusBackground(QColor color);
    /** This method returns the background color of the selected node.
        @author Bjoern
     */
    QColor getColorFocusBackground() const;
    /** This method sets the color of a marked node.
        @author Bjoern
     */
    void setColorMarked(QColor color);
    /** This method returns the color of a marked node.
        @author Bjoern
     */
    QColor getColorMarked() const;
    /** This method sets the text color.
        @author Bjoern
     */
    void setColor(QColor color);
    /** This method returns the text color.
        @author Bjoern
     */
    QColor getColor() const;
    /** This method sets the text color of nodes whose _layer number is even.
        @author Bjoern
     */
    void setColorLayer0(QColor color);
    /** This method returns the text color of nodes whose _layer number is even.
        @author Bjoern
     */
    QColor getColorLayer0() const;
    /** This method sets the text color of nodes whose _layer number is uneven.
        @author Bjoern
     */
    void setColorLayer1(QColor color);
    /** This method returns the text color of nodes whose _layer number is uneven.
```

```cpp
        @author Bjoern
      */
    QColor getColorLayer1() const;
    /** This method draws the data on the underlying mainWindow’s treeView.
        @param <painter> is a pointer which allows drawing on the mainWindow’s
        treeView.
        @param <option> holds various kinds of information such as the coordinates
        and size of the rectangle on which the node data is supposed to be drawn.
        It also provides information whether or not the currently drawn node is
        selected. Notice that this method is called for each node of the
        mainWindow’s treeView every time the treeView is repainted what can be
        caused be various events such as resizing the widget.
        @param <index> is the index of the currently drawn node. Using the
        underlying model data structure the node can be identified and accessed
        by means of the given index.
        @author Bjoern
      */
    void paint(QPainter* painter, const QStyleOptionViewItem &option,
               const QModelIndex &index) const;
};

#endif // ITEMDELEGATE_H
```

A.7. main

Listing 13: `main.cpp`

```cpp
#include <QtGui/QApplication>
#include <QtCore/QCoreApplication>

#include "mainwindow.h"
#include "console.h"
#include "settings.h"
#include <QString>
#include <QTranslator>
#include <QTextCodec>
#include <QPointer>

int main(int argc, char* argv[])
{
    /** 0 = executable’s name
     *  1 = source file path
     *  2 = source file type
     *  3 = target file path
     *  4 = target file type
     *  x = "-g" or "--gui"
     *  y = "-v" or "--verbose"
     */

    QStringList arguments;
    QStringList options;
    for (int i = 1; i < argc; i++)
    {
        QString argument(argv[i]);
        argument = argument.toLower();
        if (argument.startsWith('-'))
            options << argument;
        else
```

```
            arguments << argument;
    }
    /** open the GUI if argument "-g" or "--gui" is given */
    if ((bool)options.contains("-g")
        | (bool)options.contains("--gui"))
    {
        QApplication a(argc, argv);
        MainWindow w(arguments, options, 0);
        w.show();
        return a.exec();

    }
    else if (argc == 1)
        std::cerr << "usage: htmlatex INPUTFILE FORMAT INPUTDEFINITION OUTPUTFILE FORMAT OUTPUTDEFINITION [-g|--gui]\n"
                << "  e.g. htmlatex index.html javadoc input_javadoc.xml manual.tex tex output_tex.xml -g\n"
                << "\t-g, --gui \tLaunch the GUI\n"
                << "\t-h, --help \tShow some examples\n"
                << "\t-v, --verbose \tVerbose mode\n"
                << "\t-lang=de, -lang=en \tSets the language to German or English\n\n"
                << "\tSee the \"README\" file for further information." << std::endl;
    else if (argc == 2)
    {
        if ((bool)options.contains("-h")
            | (bool)options.contains("--help"))
        {
            QString helpstring("Examples with GUI:\n\nopen file initially:\n\nhtmlatex index.html javadoc -g\nhtmlatex index.html --gui javadoc");
            helpstring += "\n\nconvert file initially:\n\nhtmlatex -g index.html javadoc mytexoutput.tex tex\nhtmlatex index.html javadoc --gui mytexoutput.tex tex";
            helpstring += "\n\nExample with console:\n\nconvert file:\n\nhtmlatex index.html javadoc mytexoutput.tex tex";
            std::cerr << helpstring.toStdString() << std::endl;
        }
    }
    else
    {
        Settings settings;
        if ((bool)options.contains("-v")
            | (bool)options.contains("--verbose"))
        {
            settings.setValue("verbose", "1");
            std::cerr << "verbose mode" << std::endl;
        }
        else
            settings.setValue("verbose", "0");
        if ((bool)options.contains("-lang=de")
            | (bool)options.contains("-lang=en"))
        {
            if ((bool)options.contains("-lang=de"))
                settings.setValue("language", QString::number((int)QLocale::Germany));
            else
                settings.setValue("language", QString::number((int)QLocale::C));
        }
        QCoreApplication a(argc, argv);
        // Set translation environment for the application texts
        QLocale::Country language = (QLocale::Country)settings.getValue("language").toInt();
        QTranslator translator;
        if (language == QLocale::Germany)
        {
            if(translator.load(QString("htmlatex_de.qm")))
                std::cerr << "language set to German" << std::endl;
        }
        else
        {
            if (translator.load(QString("htmlatex_en.qm")))
                std::cerr << "language set to English" << std::endl;
        }
        a.installTranslator(&translator);
        QTextCodec::setCodecForTr(QTextCodec::codecForName("utf8"));
        Console console(arguments, options);
        exit(0);
        return a.exec();
    }
};
```

A.8. mainwindow

Listing 14: `mainwindow.h`

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "translationmapper.h"
#include "documentreader.h"
#include "model.h"
#include "itemdelegate.h"
#include "converter.h"
#include "settingsdialog.h"
#include <QMainWindow>
#include <QFileDialog>
#include <QTranslator>
#include <QTextCodec>

/** This namespace is introduced to separate GUI related source code
    @author Bjoern
  */
namespace Ui
{
    class MainWindow;
};
/** This is the central class of the GUI application. The treeView on the left
    hand side displays the opened document as a tree structure and allows
    modification via context menu. The textEdit on the right hand side displays
    the conversion output and can be edited directly.
    @author Bjoern
  */
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    /** This is the only constructor.
        @param <arguments> is an array of strings which contains the startup argument
        of the application except the executable file's name and optional paramters.
        @param <options> is an array of strings which contains just the optional
        startup arguments of the application.
        @param <parent> is a pointer pointing to the parent widget (QDialog and
        QMainWindow are derived from QWidget).
        @author Bjoern
      */
    MainWindow(QStringList arguments, QStringList options, QWidget* parent);
    /** This is the destructor.
       @author Bjoern
      */
    ~MainWindow();
private:
    /** This declaration allows access to the separated GUI source code via the
       short label ui.
       @author Bjoern
      */
    Ui::MainWindow* ui;
    /** The model is the data source of the treeView. The Model class is derived
       from the abstract class QAbstractItemModel and provides the functionality
       that a QTreeView object demands in order to display data.
       @author Bjoern
      */
    Model* _model;
    /** The _itemDelegate controls how the treeView displays data. The
       ItemDelegate class is derived from the QItemDelegate class. Its paint()
       method influences the drawing of the widget directly.
       @author Bjoern
      */
    ItemDelegate* _itemDelegate;
    /** The _settingsDialog is a modal dialog whose settings are saved to a
       binary file.
       @author Bjoern
      */
    SettingsDialog* _settingsDialog;
    /** The _translator translates the application's texts.
       @author Bjoern
      */
    QTranslator _translator;
    /** The _translationMapper provides two data objects: firstly a
       DocumentReaderData object which is used by the DocumentReader and
       secondly a key-value-pair structure which is used by the Converter.
       @author Bjoern
      */
    TranslationMapper* _translationMapper;
    /** The _converter converts the internal tree structure.
       @author Bjoern
      */
    Converter* _converter;
    /** This method performs opening and converting operations depending on the
       application's startup arguments.
       @param <arguments> is an array of strings which contains the startup argument
       of the application except the executable file's name and optional paramters.
       @param <options> is an array of strings which contains just the optional
       startup arguments of the application.
       @author Bjoern
      */
    void _performInitialOperations(QStringList arguments, QStringList options);
private slots:
    /** This slot is connected to _settingsDialog and is called when the user
       applies settings. It triggers the retranslation of the application's texts.
       @param <language> contains the information to which language the
       _translator is supposed to translate.
       @author Bjoern
      */
    void _languageChanged(QLocale::Country language);
    /** This slot is called when the user clicks the menu item or tool bar button
       for "open".
       @author Bjoern
      */
    void _open();
    /** This slot is called when the user clicks the menu item or tool bar button
```

```
            for "convert".
            @author Bjoern
          */
        void _convert();
        /** This slot is called when the user clicks the menu item or the push button
            for "save as". It allows to save the conversion output as .tex or .pdf file.
            @author Bjoern
          */
        void _saveAs();
        /** This slot is called when the user clicks the menu item or tool bar button
            for "set input definition".
            @author Bjoern
          */
        void _setInputDefinition();
        /** This slot is called when the user clicks the menu item or tool bar button
            for "set output definition".
            @author Bjoern
          */
        void _setOutputDefinition();
        /** This slot is called when the user clicks the menu item or tool bar button
            for "settings".
            @author Bjoern
          */
        void _showSettings();
        /** This slot is called when the user clicks the menu item or tool bar button
            for "about".
            @author Bjoern
          */
        void _about();
        /** This slot is called when the user clicks the right mouse button on the
            _treeView to show the context menu.
            @author Bjoern
          */
        void _showTreeViewContextMenu(QPoint point);
        /** This slot is called when the user chooses to remove a node from the treeView.
            @author Bjoern
          */
        void _treeViewRemoveNode();
        /** This slot is connected to the _converter.
            @author Bjoern
          */
        void _updateProgressBar(int percentage);
    };

    #endif // MAINWINDOW_H
```

Listing 15: `mainwindow.h`

```
    #ifndef MAINWINDOW_H
    #define MAINWINDOW_H

    #include "translationmapper.h"
    #include "documentreader.h"
    #include "model.h"
    #include "itemdelegate.h"
    #include "converter.h"
    #include "settingsdialog.h"
    #include <QMainWindow>
    #include <QFileDialog>
    #include <QTranslator>
    #include <QTextCodec>
```

```

/** This namespace is introduced to separate GUI related source code
    @author Bjoern
  */
namespace Ui
{
    class MainWindow;
};
/** This is the central class of the GUI application. The treeView on the left
    hand side displays the opened document as a tree structure and allows
    modification via context menu. The textEdit on the right hand side displays
    the conversion output and can be edited directly.
    @author Bjoern
  */
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    /** This is the only constructor.
        @param <arguments> is an array of strings which contains the startup argument
        of the application except the executable file's name and optional paramters.
        @param <options> is an array of strings which contains just the optional
        startup arguments of the application.
        @param <parent> is a pointer pointing to the parent widget (QDialog and
        QMainWindow are derived from QWidget).
        @author Bjoern
      */
    MainWindow(QStringList arguments, QStringList options, QWidget* parent);
    /** This is the destructor.
        @author Bjoern
      */
    ~MainWindow();
private:
    /** This declaration allows access to the separated GUI source code via the
        short label ui.
        @author Bjoern
      */
    Ui::MainWindow* ui;
    /** The model is the data source of the treeView. The Model class is derived
        from the abstract class QAbstractItemModel and provides the functionality
        that a QTreeView object demands in order to display data.
        @author Bjoern
      */
    Model* _model;
    /** The _itemDelegate controls how the treeView displays data. The
        ItemDelegate class is derived from the QItemDelegate class. Its paint()
        method influences the drawing of the widget directly.
        @author Bjoern
      */
    ItemDelegate* _itemDelegate;
    /** The _settingsDialog is a modal dialog whose settings are saved to a
        binary file.
        @author Bjoern
      */
    SettingsDialog* _settingsDialog;
    /** The _translator translates the application's texts.
        @author Bjoern
      */
    QTranslator _translator;
    /** The _translationMapper provides two data objects: firstly a
        DocumentReaderData object which is used by the DocumentReader and
        secondly a key-value-pair structure which is used by the Converter.
        @author Bjoern
      */
    TranslationMapper* _translationMapper;
    /** The _converter converts the internal tree structure.
        @author Bjoern
      */
    Converter* _converter;
    /** This method performs opening and converting operations depending on the
        application's startup arguments.
        @param <arguments> is an array of strings which contains the startup argument
        of the application except the executable file's name and optional paramters.
        @param <options> is an array of strings which contains just the optional
        startup arguments of the application.
        @author Bjoern
      */
    void _performInitialOperations(QStringList arguments, QStringList options);
private slots:
    /** This slot is connected to _settingsDialog and is called when the user
        applies settings. It triggers the retranslation of the application's texts.
        @param <language> contains the information to which language the
        _translator is supposed to translate.
        @author Bjoern
      */
    void _languageChanged(QLocale::Country language);
    /** This slot is called when the user clicks the menu item or tool bar button
        for "open".
        @author Bjoern
      */
    void _open();
    /** This slot is called when the user clicks the menu item or tool bar button
        for "convert".
        @author Bjoern
      */
    void _convert();
    /** This slot is called when the user clicks the menu item or the push button
        for "save as". It allows to save the conversion output as .tex or .pdf file.
        @author Bjoern
      */
    void _saveAs();
    /** This slot is called when the user clicks the menu item or tool bar button
        for "set input definition".
        @author Bjoern
      */
    void _setInputDefinition();
    /** This slot is called when the user clicks the menu item or tool bar button
        for "set output definition".
        @author Bjoern
      */
    void _setOutputDefinition();
    /** This slot is called when the user clicks the menu item or tool bar button
        for "settings".
        @author Bjoern
      */
    void _showSettings();
    /** This slot is called when the user clicks the menu item or tool bar button
        for "about".
        @author Bjoern
      */
    void _about();
    /** This slot is called when the user clicks the right mouse button on the
        _treeView to show the context menu.
        @author Bjoern
      */
    void _showTreeViewContextMenu(QPoint point);
    /** This slot is called when the user chooses to remove a node from the treeView.
        @author Bjoern
```

```
       */
    void _treeViewRemoveNode();
    /** This slot is connected to the _converter.
        @author Bjoern
      */
    void _updateProgressBar(int percentage);
};

#endif // MAINWINDOW_H
```

A.9. model

Listing 16: `model.h`

```
#ifndef MODEL_H
#define MODEL_H

#include "node.h"
#include <QAbstractItemModel>

/** This class is derived from the QAbstractItemModel. It serves the mainWindow's
    treeVoew as data source.
    @author Bjoern
  */
class Model : public QAbstractItemModel
{
public:
    /** This is the only constructor.
        @author Bjoern
      */
    Model(QObject* parent = 0);
    /** This is the only destructor. It deletes the application's internal tree
        structure by deleting the _root node which starts a destructor chain.
        @author Bjoern
      */
    ~Model();
private:
    /** This is the root node of the application's internal tree structure.
        @author Bjoern
      */
    Node* _root;
public:
    /** When a node of the mainWindow's treeView is selected the treeView's method
        "currentIndex()" returns a QModelIndex object. The nodeFromIndex() method
        of this class returns the Node object of the application's internal tree
        structure which corresponds to that given QModelIndex.
        @author Bjoern
      */
    Node* nodeFromIndex(const QModelIndex &index) const;
    /** This method sets the root node of the application's internal tree
        structure.
        @param <node> is a Node object.
        @author Bjoern
      */
    void setRootNode(Node* node);
    /** This method returns the root node of the application's internal tree
        structure.
        @author Bjoern
      */
    Node* root() const;
    QModelIndex index(int row, int column, const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role) const;
    /** This method is called by the mainWindow when a node is removed from the
        treeView. The treeView repaints the visible items.
        @author Bjoern
      */
    void refresh();
};

#endif // MODEL_H
```

Listing 17: `model.cpp`

```
#include "model.h"

Model::Model(QObject* parent) : QAbstractItemModel(parent), _root(0)
{
};
Model::~Model()
{
    delete _root;
};
void Model::setRootNode(Node* node)
{
    if (_root != 0)
        delete _root;
    _root = node;
    // reset() notifies the treeView to refetch data for visible items
    reset();
};
QModelIndex Model::index(int row, int column, const QModelIndex &parent) const
{
    if (!_root
        || (row < 0)
        || (column < 0))
        return QModelIndex();
    Node* parentNode = nodeFromIndex(parent);
    Node* childNode = parentNode->childAt(row);
    if (!childNode)
        return QModelIndex();
    return createIndex(row, column, childNode);
};
Node* Model::nodeFromIndex(const QModelIndex &index) const
{
    // cast the index's void* to a Node*
    if (index.isValid())
        return static_cast<Node*>(index.internalPointer());
    // In a model the root node is represented by an invalid index.
    return _root;
};
int Model::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode)
```

```
        return 0;
    return parentNode->getCount();
};
int Model::columnCount(const QModelIndex &parent) const
{
    // just one column meets our needs
    return 1;
};
QModelIndex Model::parent(const QModelIndex &child) const
{
    Node* node = nodeFromIndex(child);
    if (!node)
        return QModelIndex();
    Node* parentNode = node->getParent();
    if (!parentNode)
        return QModelIndex();
    Node* grandparentNode = parentNode->getParent();
    if (!grandparentNode)
        return QModelIndex();
    int row = grandparentNode->indexOf(parentNode);
    return createIndex(row, 0, parentNode);
};
QVariant Model::data(const QModelIndex &index, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    Node* node = nodeFromIndex(index);
    if (!node)
        return QVariant();
    return node->getName() + ": " + node->getContent();
};
QVariant Model::headerData(int section, Qt::Orientation orientation, int role) const
{
    return QVariant();
};
Node* Model::root() const
{
    return this->_root;
};
void Model::refresh()
{
    reset();
};
```

A.10. node

Listing 18: `node.h`

```
#ifndef NODE_H
#define NODE_H

#include <QString>
#include <QList>
#include <QMap>

/** The Node class provides rudimental functionality to build tree structures and
    access its nodes. Each node is aware of its parent and children which can be
    accessed by methods such as getParent() or childAt(int index). Additionally
    each node holds an array of attributes.
```

```
  */
class Node
{
private:
    /** The _instCount class member counts the created instances of the Node class.
        @author Bjoern
      */
    static qint64 _instCount;
    /** The _treeLevel attribute holds the greatest distance of a leaf to the root
        of the whole tree.
        @author Bjoern
      */
    static qint64 _treeLevel;
    /** The _treeNodeCount attribute holds the number of nodes of the whole tree.
        @author Bjoern
      */
    static qint64 _treeNodeCount;
    /** The _id is a unique number which allows identification of the node.
        @author Bjoern
      */
    qint64 _id;
    /** The _layer number allows to determine to which layer / generation a node
        belongs. The root node has the _layer number 0, its children 1 and its
        grand children 2 and so on.
        @author Bjoern
      */
    qint64 _layer;
    /** This is the name of the node.
        @author Bjoern
      */
    QString _name;
    /** This is the content of the node.
        @author Bjoern
      */
    QString _content;
    /** This pointer points to the parent node. In case of the root node it points to null.
        @author Bjoern
      */
    Node* _parent;
    /** This is generic array of pointers pointing to the child nodes.
        @author Bjoern
      */
    QList<Node*> _children;
    /** This is a key-value-structure which provides access to the value of type
        QString via keys of the type QString.
        @author Bjoern
      */
    QMap<QString, QString> _attributes;
    /** The _cursor indicates the current index position within the array of child
        nodes (_children). The _cursor moves when the methods firstChild(),
        nextChild() and lastChild() are called.
        @author Bjoern
      */
    int _cursor;
public:
    /** This is the copy-construcotr of the class.
        @author Bjoern
      */
    Node(const Node& node);
    /** This constructor creates an instance initiallzing its _parent, _name and
        and _layer number.
        @author Bjoern
      */
    Node(Node* parent, QString name, qint64 layer);
    /** This destructor ensures that all child nodes are deleted as well.
        @author Bjoern
      */
    ~Node();
    /** This method returns the _treeLevel number (class member).
        @author Bjoern
      */
    qint64 getTreeLevel() const;
    /** This method returns the number of nodes of the whole tree (class member).
        @author Bjoern
      */
    qint64 getTreeNodeCount() const;
    /** This method returns the ID of the node.
        @author Bjoern
      */
    qint64 getID() const;
    /** This method returns the _layer number of the node.
        @author Bjoern
      */
    qint64 getLayer() const;
    /** This method returns the name of the node.
        @author Bjoern
      */
    QString getName() const;
    /** This method sets the name of the node.
        @author Bjoern
      */
    void setName(QString name);
    /** This method returns the content of the node.
        @author Bjoern
      */
    QString getContent() const;
    /** This method sets the content of the node.
        @author Bjoern
      */
    void setContent(QString content);
    /** This method returns a pointer pointing to the parent node object.
        @author Bjoern
      */
    Node* getParent() const;
    /** This method sets the pointer pointing to the parent node object.
        @author Bjoern
      */
    void setParent(Node* parent);
    /** This method returns the number of child nodes.
        @author Bjoern
      */
    int getCount() const;
    /** This method returns true, if the array of child nodes (_children) contains
        the specified pointer to a Node object.
        @author Bjoern
      */
    bool containsChild(Node* node) const;
    /** This method returns a pointer to the child node specified by its array index.
        @author Bjoern
      */
    Node* childAt(int index) const;
    /** This method returns a pointer to the next child node by moving the _cursor
        one step forward (increment of 1). If the node has less than two children
        null is returned.
        @author Bjoern
      */
    Node* nextChild();
    /** This method returns a pointer to the first child node. If the node has no
```

```
           children null is returned.
           @author Bjoern
         */
        Node* firstChild();
        /** This method returns a pointer to the last child node. If the node has no
           children null is returned.
           @author Bjoern
         */
        Node* lastChild();
        /** This method appends a generic array of pointers pointing to Node objects
           to the array of child nodes (_children).
           @author Bjoern
         */
        void addChildren(QList<Node*> nodes);
        /** This method appends a pointer pointing to a Node object to the array of
           child nodes (_children).
           @author Bjoern
         */
        void addChild(Node* node);
        /** This method removes the specified pointer pointing to a Node object from
           the array of child nodes (_chidren).
           @author Bjoern
         */
        void removeChild(Node* node);
        /** This method returns the array index of the specified pointer pointing to a
           Node object. If it isn't contained in the array of child dnodes -1 is
           returned.
           @author Bjoern
         */
        int indexOf(Node* node) const;
        /** This method returns the attributes of the node.
           @author Bjoern
         */
        QMap<QString, QString> getAttributes() const;
        /** This method adds an attribute by means of a key-value-pair.
           @author Bjoern
         */
        void addAttribute(QString key, QString value);
    };

    #endif // Node_H
```

Listing 19: `node.cpp`

```
#include "node.h"

Node::Node(const Node& node)
{
    _instCount++;
    this->_id = _instCount;
    this->_name = node.getName();
    this->_content = node.getContent();
    this->_layer = node.getLayer();
    this->_parent = node.getParent();
    this->_treeLevel = node.getTreeLevel();
    // add child nodes
    for (int i = 0; i < node.getCount(); i++)
        this->addChild(new Node(*node.childAt(i)));
    // add attributes
    QList<QString> keys = node.getAttributes().keys();
    for (int i = 0; i < keys.count(); i++)
```

```
        this->addAttribute(keys.at(i), node.getAttributes()[keys.at(i)]);
};
Node::Node(Node* parent, QString name, qint64 layer) : _layer(layer),
    _name(name), _content(""), _parent(parent), _cursor(0)
{
    if (_parent == 0)
        _treeNodeCount = 0;
    else
        _treeNodeCount++;
    _instCount++;
    if (layer > _treeLevel)
        _treeLevel = layer;
    this->_id = _instCount;
    this->_children = QList<Node*>();
    this->_attributes = QMap<QString, QString>();
};
Node::~Node()
{
    qDeleteAll(_children);
};
qint64 Node::_instCount = 0;
qint64 Node::_treeLevel = 0;
qint64 Node::_treeNodeCount;
qint64 Node::getTreeLevel() const
{
    return this->_treeLevel;
};
qint64 Node::getTreeNodeCount() const
{
    return this->_treeNodeCount;
};
qint64 Node::getID() const
{
    return this->_id;
};
qint64 Node::getLayer() const
{
    return this->_layer;
};
QString Node::getName() const
{
    return this->_name;
};
void Node::setName(QString name)
{
    this->_name = name;
};
QString Node::getContent() const
{
    return this->_content;
};
void Node::setContent(QString content)
{
    this->_content = content;
};
Node* Node::getParent() const
{
    if (_parent != 0)
        return this->_parent;
    return 0;
};
void Node::setParent(Node* parent)
{
    this->_parent = parent;
```

```
};
int Node::getCount() const
{
    return _children.count();
};
bool Node::containsChild(Node* node) const
{
    bool result = false;
    for (int i = 0; i < _children.count(); i++)
    {
        if (_children.at(i)->_id == node->_id)
        {
            result = true;
            break;
        }
    }
    return result;
};
Node* Node::childAt(int index) const
{
    return _children.value(index);
};
Node* Node::nextChild()
{
    if (_cursor < _children.count())
    {
        _cursor++;
        return _children.at(_cursor - 1);
    }
    return 0;
};
Node* Node::firstChild()
{
    _cursor = 0;
    if (_children.count() > 0)
        return _children.at(_cursor);
    return 0;
};
Node* Node::lastChild()
{
    if (_children.count() > 0)
    {
        _cursor = _children.count() - 1;
        return _children.at(_cursor);
    }
    return 0;
};
void Node::addChildren(QList<Node*> nodes)
{
    for (int i = 0; i < nodes.count(); i++)
    {
        if (!containsChild(nodes.at(i)))
        {
            nodes.at(i)->setParent(this);
            _children.append(nodes.at(i));
        }
    }
};
void Node::addChild(Node* node)
{
    if (node != 0)
    {
        if (!containsChild(node))
        {
            node->setParent(this);
            _children.append(node);
        }
    }
};
void Node::removeChild(Node* node)
{
    if (_children.removeOne(node))
    {
        _cursor = -1;
        delete node;
    }
};
int Node::indexOf(Node* node) const
{
    return _children.indexOf(node);
};
QMap<QString, QString> Node::getAttributes() const
{
    return this->_attributes;
};
void Node::addAttribute(QString key, QString value)
{
    this->_attributes[key] = value;
};
```

A.11. settings

Listing 20: `settings.h`

```
#ifndef SETTINGS_H
#define SETTINGS_H

#include <QObject>
#include <QDataStream>
#include <QFile>
#include <QMap>
#include <iostream>

/** This class stores settings variables for the application in a binary file.
    The settings are stored in a key-value-pair data structure (an associative
    array). The class member DEBUG is used all over the application to provide
    additional information about the steps of the processings if desired (verbose).
    Saving to and loading from the binary file is performed "automatically". When
    a variable is requested or set the loading or saving is performed implicitly.
  */
class Settings : public QObject
{
    Q_OBJECT
public:
    /** This constructor tries to load the settings from the binary file. If it
        doesn't exist it is created with default settings.
        @author Bjoern
      */
    Settings();
    /** This class attribute is used to enable or disable "verbose-mode".
        @author Bjoern
      */
    static bool DEBUG;
private:
    /** This is the binary file's file path. This file contains the settings. If
        it doesn't exist it is created "automatically" with default settings.
        @author Bjoern
      */
    const QString _SETTINGSFILEPATH;
    /** This is the associative array which stores the settings as key-value-pairs.
        @author Bjoern
      */
    QMap<QString,QString> _settingsMap;
    /** This method tries to load the settings from the binary file.
        @author Bjoern
      */
    bool _load();
    /** This method tries to save the settings to the binary file.
        @author Bjoern
      */
    bool _save();
public:
    /** This method sets the specified variable to the given value.
        @param <key> is the variable's name.
        @param <value> is the new variable's value.
        @author Bjoern
      */
    void setValue(QString key, QString value);
    /** This method returns the value of the specified variable.
        @param <key> is the variable's name.
        @author Bjoern
      */
    QString getValue(QString key);
};

#endif // SETTINGS_H
```

Listing 21: `settings.cpp`

```
#include "settings.h"

Settings::Settings() : _SETTINGSFILEPATH("htmlatex_settings.dat")
{
    if (!_load())
    {
        // an integer value taken from the enumerator QLocale::Country
        _settingsMap["language"] = "82";
        _settingsMap["latexpath"] = "";
        // boolean value
        _settingsMap["verbose"] = "0";
        // boolean value
        _settingsMap["includesubdocuments"] = "1";
        _save();
    }
};
bool Settings::DEBUG = false;
bool Settings::_load()
{
    QFile file(_SETTINGSFILEPATH);
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        std::cerr << tr("Settings.load() - file can't be opened for reading").toStdString() << ↘
             →std::endl;
        return false;
    }
    QDataStream stream(&file);
    stream.setVersion(QDataStream::Qt_4_6);
    stream >> _settingsMap;
    file.close();
    DEBUG = (bool)_settingsMap["verbose"].toInt();
    return true;
};
bool Settings::_save()
{
    QFile file(_SETTINGSFILEPATH);
    if (!file.open(QFile::WriteOnly | QFile::Text))
    {
        std::cerr << tr("Settings.save() - file can't be opened for writing").toStdString() << ↘
            →std::endl;
        return false;
    }
    QDataStream stream(&file);
    stream.setVersion(QDataStream::Qt_4_6);
    stream << _settingsMap;
    file.close();
    return true;
};
QString Settings::getValue(QString key)
{
    if (!_load())
    {
        if (DEBUG)
        {
            std::cerr << tr("Settings.getValue() returned default value for key: ").toStdString() ↘
                →<< key.toStdString()
                << tr(", value: ").toStdString() << _settingsMap[key].toStdString() << std::endl;
        }
    }
    return _settingsMap[key];
};
void Settings::setValue(QString key, QString value)
{
    _settingsMap[key] = value;
    if (key == "verbose")
        this->DEBUG = (bool)value.toInt();
    _save();
};
```

A.12. settingsdialog

Listing 22: `settingsdialog.h`

```
#ifndef SETTINGSDIALOG_H
#define SETTINGSDIALOG_H

#include "settings.h"
#include <QDialog>
#include <QFileInfo>
#include <QProcess>
#include <QVariant>
#include <QMessageBox>
```

```
#include <QLocale>
#include <QFileDialog>

/** This namespace is introduced to separate GUI related source code
    @author Bjoern
  */
namespace Ui {
    class SettingsDialog;
}

/** The SettingsDialog provides a convenient way to change the application's
    settings such as the language or the inclusion of subdocuments. It makes use of
    the Settings class to laod the settings from and save them to a binary file.
    @author Bjoern
  */
class SettingsDialog : public QDialog
{
    Q_OBJECT
public:
    /** This is the only constructor which initializes the GUI.
        @author Bjoern
      */
    SettingsDialog(QWidget *parent = 0);
    /** This is the destructor.
        @author Bjoern
      */
    ~SettingsDialog();
private:
    /** This declaration allows access to the separated GUI source code via the
        short label ui.
        @author Bjoern
      */
    Ui::SettingsDialog *ui;
private slots:
    /** This slot is called when the user clicks the push button "apply".
        @author Bjoern
      */
    void _apply();
    /** This slot is called when the user clicks the push button "browse".
        @author Bjoern
      */
    void _open();
signals:
    /** This signal is connected to mainWindow and is raised when the user applies
        settings. It triggers the retranslation of the application's texts.
        @param <language> contains the information to which language mainWindow's
        _translator is supposed to translate.
        @author Bjoern
      */
    void languageChanged(QLocale::Country language);
public:
    /** This method is called by the mainWindow's _languageChanged slot after the
        mainWindow's _translator has been switched to another language. It triggers
        the retranslation of the SettingsDialog's texts.
        @author Bjoern
      */
    void retranslateUi();
};

#endif // SETTINGSDIALOG_H
```

Listing 23: `settingsdialog.cpp`

```
#include "settingsdialog.h"
#include "ui_settingsdialog.h"

SettingsDialog::SettingsDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::SettingsDialog)
{
    ui->setupUi(this);
    ui->comboBox_language->addItem(tr("English"), QVariant(QLocale::C));
    ui->comboBox_language->addItem(tr("German"), QVariant(QLocale::Germany));
    connect(ui->pushButton_apply, SIGNAL(clicked()),
            this, SLOT(_apply()));
    connect(ui->pushButton_latexpath, SIGNAL(clicked()),
            this, SLOT(_open()));
    Settings settings;
    ui->lineEdit_latexpath->setText(settings.getValue("latexpath"));
    // select language specified by settings file
    QLocale::Country initially_selected_language = ↘
         →(QLocale::Country)settings.getValue("language").toInt();
    // retrieve corresponding comboBox item index
    int itemindex = ui->comboBox_language->findData(initially_selected_language);
    ui->comboBox_language->setCurrentIndex(itemindex);
    ui->checkBox_verbose->setChecked((bool)settings.getValue("verbose").toInt());
    ui->checkBox_includesubdocuments->setChecked((bool)settings.getValue("includesubdocuments").toInt());
};
SettingsDialog::~SettingsDialog()
{
    delete ui;
};
void SettingsDialog::_apply()
{
    bool apply = true;
    QFileInfo fileinfo(ui->lineEdit_latexpath->text());
    if (!fileinfo.exists())
    {
        QMessageBox msg(QMessageBox::Warning, tr("Error"),
                        tr("The file path doesn't exist.\n\nDo you still want to proceed?"),
                        QMessageBox::Yes|QMessageBox::No);
        if (msg.exec() == QMessageBox::No)
            apply = false;
    }
    else if (!fileinfo.isFile())
    {
        QMessageBox msg(QMessageBox::Warning, tr("Error"),
                        tr("No file path specified.\n\nDo you still want to proceed?"),
                        QMessageBox::Yes|QMessageBox::No);
        if (msg.exec() == QMessageBox::No)
            apply = false;
    }
    if (apply)
    {
        Settings settings;
        settings.setValue("language", ↘
             →ui->comboBox_language->itemData(ui->comboBox_language->currentIndex()).toString());
        settings.setValue("latexpath", ui->lineEdit_latexpath->text());
        settings.setValue("includesubdocuments", ↘
             →QString::number((int)ui->checkBox_includesubdocuments->isChecked()));
        settings.setValue("verbose", QString::number((int)ui->checkBox_verbose->isChecked()));
        QLocale::Country language = ↘
             →(QLocale::Country)ui->comboBox_language->itemData(ui->comboBox_language->currentIndex()).toInt();
        emit languageChanged(language);
        ui->retranslateUi(this);
        accept();
    }
};
void SettingsDialog::_open()
{
    QFileDialog* dialog = new QFileDialog(this,
                                          tr("Set latex executable"),
                                          "",
                                          tr("executable (*.exe);;any file (*.*)"));
    // retrieve source file path and type
    if (dialog->exec() == QFileDialog::Accepted)
        ui->lineEdit_latexpath->setText(dialog->selectedFiles().at(0));
    delete dialog;
};
void SettingsDialog::retranslateUi()
{
    ui->retranslateUi(this);
};
```

A.13. translationdata

Listing 24: `translationdata.h`

```
#ifndef TRANSLATIONDATA_H
#define TRANSLATIONDATA_H

#include "translationdatanode.h"
#include <QString>
#include <QMap>

/** The Converter performs the conversion on the application's internal tree
    structure. In order to be able to translate each node of the internal tree the
    Converter retrieves a TranslationData object via the TranslationMapper. The
    Converter passes the name of that node which is currently to be translated as
    key to the TranslationMapper which returns the respective TranslationData
    object from its _outputMap (an associative array). This TranslationData object
    contains information which allow textual replacements on that node's content
    which is currently to be translated.
  */
class TranslationData
{
public:
    /** This is the only constructor which initializes the attributes.
        @author Bjoern
      */
    TranslationData();
private:
    /** The _key is used by the TranslationMapper to store and access the
        TranslationData object in its _outputMap (an associative array).
        @author Bjoern
      */
    QString _key;
    /** This attribute is unused yet.
        @author Bjoern
      */
    QString _from;
    /** This attribute is the target string and can contain a place holder for
```

```
        further additions in the translation process.
        @author Bjoern
      */
    QString _to;
    /** This array of TranslationDataNode objects allows inclusion of special
        target document formatting instructions (i.e. "requires" in TEX).
      */
    QList<TranslationDataNode> _requires;
public:
    /** This method returns the value of the _key attribute.
        @author Bjoern
      */
    QString key() const;
    /** This method sets the value of the _key attribute.
        @author Bjoern
      */
    void setKey(QString key);
    /** This method returns the value of the _from attribute.
        @author Bjoern
      */
    QString from() const;
    /** This method sets the value of the _from attribute.
        @author Bjoern
      */
    void setFrom(QString from);
    /** This method returns the value of the _to attribute.
        @author Bjoern
      */
    QString to() const;
    /** This method sets the value of the _to attribute.
        @author Bjoern
      */
    void setTo(QString to);
    /** This method returns the array of TranslationDataNode objects stored in the
        _requires attributes.
        @author Bjoern
      */
    QList<TranslationDataNode> requires() const;
    /** This method adds a TranslationDataNode object to the array of
        TranslationDataNode objects stored in the _requires attribute.
        @author Bjoern
      */
    void addRequiresNode(TranslationDataNode requiresnode);
};

#endif // TRANSLATIONDATA_H
```

Listing 25: `translationdata.cpp`

```
#include "translationdata.h"

TranslationData::TranslationData() : _key(""), _from(""), _to("")
{
    this->_requires = QList<TranslationDataNode>();
};
QString TranslationData::from() const
{
    return this->_from;
};
void TranslationData::setFrom(QString from)
{
    this->_from = from;
};
QString TranslationData::key() const
{
    return this->_key;
};
void TranslationData::setKey(QString key)
{
    this->_key = key;
};
QString TranslationData::to() const
{
    return this->_to;
};
void TranslationData::setTo(QString to)
{
    this->_to = to;
};
QList<TranslationDataNode> TranslationData::requires() const
{
    return this->_requires;
};
void TranslationData::addRequiresNode(TranslationDataNode requiresnode)
{
    this->_requires.append(requiresnode);
};
```

A.14. translationdatanode

Listing 26: `translationdatanode.h`

```
#ifndef TRANSLATIONDATANODE_H
#define TRANSLATIONDATANODE_H

#include <QString>
#include <QMap>

/** The TranslationMapper processes the input and output definition files and
    creates TranslationData objects which hold an array of TranslationDataNode
    objects. This class describes a XML node of the output definition file which
    allows inclusion of special target document formatting instructions (i.e.
    "requires" in TEX).
    @author Bjoern
  */
class TranslationDataNode
{
public:
    /** This is the only constructor which initializes the attributes.
        @author Bjoern
      */
    TranslationDataNode();
private:
    /** This is the name of the XML node describing a special target document
        formatting instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    QString _name;
    /** This is the content of the XML node describing a special target document
```

```
        formatting instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    QString _content;
    /** This is an associative array holding the attributes of the XML node
        describing a special target document formatting instruction (i.e.
        "requires" in TEX).
        @author Bjoern
      */
    QMap<QString,QString> _attributes;
public:
    /** This method returns the name of the XML node describing a special target
        document formatting instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    QString name() const;
    /** This method sets the name of the XML node describing a special target
        document formatting instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    void setName(QString name);
    /** This method returns the content of the XML node describing a special target
        document formatting instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    QString content() const;
    /** This method sets the content of the XML node describing a special target
        document formatting instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    void setContent(QString content);
    /** This method returns an associative array holding the attributes of the XML
        node describing a special target document formatting instruction (i.e.
        "requires" in TEX).
        @author Bjoern
      */
    QMap<QString,QString> attributes() const;
    /** This method adds an attribute to the associative array holding the
        attributes of the XML node describing a special target document formatting
        instruction (i.e. "requires" in TEX).
        @author Bjoern
      */
    void addAttribute(QString name, QString value);
};

#endif // TRANSLATIONDATANODE_H
```

Listing 27: `translationdatanode.cpp`

```
#include "translationdatanode.h"

TranslationDataNode::TranslationDataNode() : _name(""),
    _content("")
{
    this->_attributes = QMap<QString,QString>();
};
QString TranslationDataNode::name() const
{
    return this->_name;
};
void TranslationDataNode::setName(QString name)
```

```
{
    this->_name = name;
};
QString TranslationDataNode::content() const
{
    return this->_content;
};
void TranslationDataNode::setContent(QString content)
{
    this->_content = content;
};
QMap<QString,QString> TranslationDataNode::attributes() const
{
    return this->_attributes;
};
void TranslationDataNode::addAttribute(QString name, QString value)
{
    this->_attributes[name] = value;
};
```

A.15. translationmapper

Listing 28: `translationmapper.h`

```
#ifndef TRANSLATIONMAPPER_H
#define TRANSLATIONMAPPER_H

#include "translationdata.h"
#include "settings.h"
#include "documentreaderdata.h"
#include <QObject>
#include <QFile>
#include <QDomDocument>
#include <QString>
#include <QMap>
#include <iostream>

/** This class processes the input and output definition files to procure data
    structures which meet the needs of the DocumentReader and Converter classes.
    Hence it creates a DocumentReaderData object stored in the _documentReference
    attribute which describes how a reference to a subdocument of an input
    document is defined. In addition it creates a key-value-pair data structure
    (an associative array) which holds information about how the relevant nodes
    of the input document are to be translated.
    @author Bjoern
  */
class TranslationMapper : public QObject
{
    Q_OBJECT
public:
    /** This is the only constructor.
        @author Bjoern
      */
    TranslationMapper(QObject *parent = 0);
private:
    /** This attribute contains a DocumentReaderData object which describes how a
        reference to a subdocument of an input document is defined.
        @author Bjoern
        */
    DocumentReaderData _documentReference;
    /** This attribute holds a key-value-pair data structure (an associative array).
        The items of this array can be accessed via a key of the type QString
        (i.e. _outputMap["head"]). The node names of the input document serve as
        keys for this array. The received value is a TranslationData object
        providing information for the Converter which performs textual replacements.
        @author Bjoern
      */
    QMap<QString,TranslationData> _outputMap;
public:
    /** This method processes the input definition file to store a DocumentReaderData
        object in the attribute _documentReference.
        @param <inputfilepath> is the file path to the input definition file.
        @author Bjoern
      */
    void createDocumentReaderData(QString inputfilepath);
    /** This method processes the output definition file to populate the attribute
        _outputMap with a TranslationData object for each different node of the
        input document.
        @param <outputfilepath> is the file path to the output definition file.
        @author Bjoern
      */
    void createOutputElementMap(QString outputfilepath);
    /** This method returns the object held by the attribute _documentReference.
        @author Bjoern
      */
    DocumentReaderData getDocumentReference() const;
    /** This method returns the object held by the attribute _outputMap.
        @author Bjoern
      */
    QMap<QString,TranslationData> outputMap() const;
};

#endif // TRANSLATIONMAPPER_H
```

Listing 29: `translationmapper.cpp`

```
#include "translationmapper.h"

TranslationMapper::TranslationMapper(QObject *parent) : QObject(parent)
{
};
void TranslationMapper::createDocumentReaderData(QString inputfilepath)
{
    QFile file(inputfilepath);
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        if (Settings::DEBUG)
        {
            std::cerr << tr("TranslationMapper.createDocumentReaderData(): ").toStdString()
                      << tr("file.open() returned false\n\tpath: ").toStdString()
                      << inputfilepath.toStdString() << std::endl;
        }
        return;
    }
    // load content of the XML file into "filecontent"
    QString filecontent = QString(file.readAll()).toLatin1();
    file.close();
    QDomDocument doc;
    QString errorMsg = "";
```

```cpp
    int errorLine = -1;
    int errorColumn = -1;
    if (doc.setContent(filecontent, &errorMsg, &errorLine, &errorColumn))
    {
        QDomElement root = doc.documentElement();
        QDomNode node = root.elementsByTagName("documentreference").at(0);
        if (!node.isNull())
        {
            QDomNamedNodeMap attributes = node.attributes();
            QString tagname = attributes.namedItem("tagname").nodeValue();
            QString urlcontainingattributename = attributes.namedItem("urlcontainingattributename").nodeValue();
            _documentReference = DocumentReaderData(tagname, urlcontainingattributename);
        }
    }
    else if (Settings::DEBUG)
    {
        std::cerr << tr("TranslationMapper.createDocumentReaderData(): doc.setContent returned false").toStdString()
                << tr("\n\tfilecontent: ").toStdString() << filecontent.toStdString() << std::endl;
    }
};
void TranslationMapper::createOutputElementMap(QString outputfilepath)
{
    _outputMap = QMap<QString,TranslationData>();
    QFile file(outputfilepath);
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        if (Settings::DEBUG)
        {
            std::cerr << tr("TranslationMapper.createOutputElementMap(): ").toStdString()
                    << tr("file.open() returned false\n\tPath: ").toStdString()
                    << outputfilepath.toStdString() << std::endl;
        }
        return;
    }
    // load content of the XML file into "filecontent"
    QString filecontent = QString(file.readAll()).toLatin1();
    file.close();
    QDomDocument doc;
    QString errorMsg = "";
    int errorLine = -1;
    int errorColumn = -1;
    if (doc.setContent(filecontent, &errorMsg, &errorLine, &errorColumn))
    {
        QDomElement root = doc.documentElement();
        // create a node list of "node"-elements
        QDomNodeList nodes = root.elementsByTagName("node");
        for (int i = 0; i < nodes.count(); i++)
        {
            // create a node list from "node"-element's child nodes
            QDomNodeList subnodes = nodes.at(i).childNodes();
            TranslationData translationdata;
            for (int j = 0; j < subnodes.count(); j++)
            {
                if (subnodes.at(j).nodeName().toLower() == "from")
                    translationdata.setFrom(subnodes.at(j).firstChild().nodeValue());
                else if (subnodes.at(j).nodeName().toLower() == "key")
                    translationdata.setKey(subnodes.at(j).firstChild().nodeValue());
                else if (subnodes.at(j).nodeName().toLower() == "to")
                    translationdata.setTo(subnodes.at(j).firstChild().nodeValue());
                else if (subnodes.at(j).nodeName().toLower() == "requires")
                {
                    QDomNodeList requiresnodes = subnodes.at(j).childNodes();
                for (int k = 0; k < requiresnodes.count(); k++)
                {
                    TranslationDataNode datanode;
                    datanode.setName(requiresnodes.at(k).nodeName());
                    datanode.setContent(requiresnodes.at(k).firstChild().nodeValue());
                    QDomNamedNodeMap attributes = requiresnodes.at(k).attributes();
                    for (int l = 0; l < attributes.count(); l++)
                    {
                        datanode.addAttribute(attributes.item(l).nodeName(),
                                              attributes.item(l).nodeValue());
                    }
                    translationdata.addRequiresNode(datanode);
                }
            }
        }
        _outputMap[translationdata.key()] = translationdata;
    }
}
else if (Settings::DEBUG)
{
    std::cerr << tr("TranslationMapper.createOutputElementMap(): doc.setContent returned ↘
        →false").toStdString()
        << tr("\n\tfilecontent: ").toStdString() << filecontent.toStdString() << std::endl;
}
};
DocumentReaderData TranslationMapper::getDocumentReference() const
{
    return this->_documentReference;
};
QMap<QString,TranslationData> TranslationMapper::outputMap() const
{
    return this->_outputMap;
};
```

A.16. input_javadoc.xml

Listing 30: input_javadoc.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE javadoc [
<!ELEMENT javadoc (documentreference)>
<!ELEMENT documentreference EMPTY>
<!ATTLIST documentreference tagname CDATA #REQUIRED
    urlcontainingattributename CDATA #REQUIRED>
]>
<javadoc>
    <documentreference tagname="a" urlcontainingattributename="href" />
</javadoc>
```

A.17. output_tex.xml

Listing 31: output_tex.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE tex [
<!ELEMENT tex (node*)>
<!ELEMENT node ((key, from, to, requires?)*)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT requires (location*)>
<!ELEMENT location (#PCDATA)>

<!ATTLIST location position (start|content|end) "content">
]>

<!-- Tags: html, head, body, title, author, h1, h2, h3, hr, br, table, th, tr,
td, b, a, dl, dt, dd, font, noscript, script, ul, li, link,!doctype -->

<tex>

<!-- html -->
<node>
  <key>html</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[]]></to>
</node>

<!-- head -->
<node>
  <key>head</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[]]></to>
  <requires>
    <location position="start"><![CDATA[\documentclass[a4paper,% DIN A4
fontsize=12pt     %  Schriftgre   12 Punkt
]{scrartcl}          % 'article' aus dem 'Koma'-Script
\usepackage[pdfborder={0 0 0},   % Rahmen um Links auf 0 festlegen
           ]{hyperref}       % anklickbare Links im Inhaltsverzeichnis]]>
    </location>
  </requires>
</node>


<!-- title -->
<node>
  <key>title</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[]]></to>
  <requires>
    <location position="start"><![CDATA[\title{----CONTENT----}]]></location>
  </requires>
</node>



<!-- body -->
<node>
  <key>body</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[\begin{document}----CONTENT----\end{document}]]></to>
</node>



<!-- author -->
<node>
```

```
    <key>author</key>
    <from><![CDATA[<----TEXT----]]></from>
    <to><![CDATA[]]></to>
    <requires>
      <location position="start"><![CDATA[\author{----CONTENT----}]]></location>
    </requires>
  </node>



  <!-- h1 -->
  <node>
  <key>h1</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[\section{----CONTENT----}]]></to>
  </node>


  <!-- h2 -->
  <node>
  <key>h2</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[\subsection{----CONTENT----]]>}</to>
  </node>


  <!-- h3 -->
  <node>
  <key>h1</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[\subsubsection{----CONTENT----}]]></to>
  </node>


  <!-- br -->
  <node>
    <key>br</key>
    <from></from>
    <to><![CDATA[\newline]]></to>
  </node>


  <!-- p -->
  <node>
    <key>p</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[----CONTENT----]]></to>
  </node>


  <!-- table -->
  <node>
    <key>table</key>
    <from><![CDATA[----CONTENT----]]></from>
    <to><![CDATA[begin{longtable}{|p{.2textwidth}|p{.8textwidth}|}
  hline----CONTENT----end{longtable}]]></to>
    <requires>
      <location position="start"><![CDATA[\usepackage{longtable}]]></location>
    </requires>
  </node>

  <!-- th -->
  <node>
    <key>th</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[----CONTENT----&----CONTENT---- \\ \hline \endhead]]></to>
  </node>


  <!-- tr
  hier wirds kompliziert:
     &  \\
     &  \\
     &  \\
     &
  -->
  <node>
    <key>tr</key>
    <from><![CDATA[<<td>----CONTENT----</td>&<td>----CONTENT----</td>]]></from>
    <to><![CDATA[----CONTENT----&----CONTENT---- \\ \hline]]></to>
  </node>


  <!-- b -->
  <node>
    <key>b</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[\textbf{----CONTENT----}]]></to>
  </node>


  <!-- a
  wenn alles in eine datei geschrieben werden soll, dann wird es nicht gebraucht,
  sonst eine weitere .tex-Datei erstellen und einbinden
  -->
  <node>
    <key>a</key>
    <from><![CDATA[<----HREF----]]></from>
    <to><![CDATA[\input{----HREF----.tex}]]></to>
  </node>


  <!-- dl -->
  <node>
    <key>dl</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[\begin{description}----CONTENT----\textbf{}\end{description}]]></to>
  </node>


  <!-- dt -->
  <node>
    <key>dt</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[\item[----CONTENT----]]]></to>
  </node>


  <!-- dd
  fehlte in der ursprnglichen Liste der Tags... -->
  <node>
    <key>dd</key>
    <from><![CDATA[<----CONTENT----]]></from>
    <to><![CDATA[----CONTENT----]]></to>
  </node>


  <!-- ul -->
```

```
<node>
  <key>ul</key>
  <from><![CDATA[<]]></from>
  <to><![CDATA[\begin{itemize}----CONTENT----\end{itemize}]]></to>
</node>


<!-- li -->
<node>
  <key>li</key>
  <from><![CDATA[<]]></from>
  <to><![CDATA[\item ----CONTENT----]]></to>
</node>


<!-- font -->
<node>
  <key>font</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[----CONTENT----]]></to>
</node>


<!-- noscript -->
<node>
  <key>noscript</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[]]></to>
</node>


<!-- script -->
<node>
  <key>script</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[]]></to>
</node>


<!-- link -->
<node>
  <key>link</key>
  <from><![CDATA[<----CONTENT----]]></from>
  <to><![CDATA[]]></to>
</node>

</tex>
```

B. Erklärung

Hiermit erklären wir, dass das Projekt htmlLaTeXvon uns selbständig und ohne Hilfe Dritter erarbeitet und realisiert wurde.

Björn Kaiser

Björn Baß

Abbildungsverzeichnis

Tabellenverzeichnis

Listings