SS 2011

Projektarbeit : Qt 4.7

Erstellt von : Sylvain David Diffo Tatsi

Matrikelnummer : 29650

Betreuer: Prof. Dr. -Ing. Holger Vogelsang

Inhaltverzeichnis

Einführung

Im 5. Semester an der Hochschule Karlsruhe, Fachrichtung Informatik muss eine Projektarbeit von Studenten erstellt werden, indem eine abgeschlossene Aufgabenstellung selbständig bearbeitet werden soll. Die verschiedenen Themen sind von den Dozenten der Hochschule am Anfang des Semesters vorgeschlagen und jeder Student soll ein Thema davon auswählen.

Das behandelte Thema in Diesem Bericht ist „Qt 4.7", das anhang eines Beispiels konkret dargestellt wird. Das Thema ist in sich sehr interessant, weil seit dem Anfang des Studiums das Erstellen von graphischen Oberflächen nicht gelehrt wurde und dieses Thema ist die Gelegenheit es zu lernen und anzuwenden.

Das gewählte Beispiel für dieses Thema ist das Damenspiel, das mithilfe von Qt und C++ implementiert werden soll.

In diesem Bericht wird zuerst kurz Qt 4.7 präsentiert und danach die Implementierung des Damenspiels dargestellt.

I. Qt 4.7

1. Einführung

Qt ist eine Klassenbibliothek, die mit C++ geschrieben wurde. Mit ihm kann man grafische Benutzeroberflächen programmieren, die Plattform- und Betriebssystem übergreifend sind. Die Entwicklung von Qt hat in 1991 mit *Haavard Nord* und *Eirik Chambe*-eng in Norwegen angefangen. In 1994 haben die 2 Entwickler das Unternehmen *Trolltech* gegründet, das die Entwicklung von Qt sehr beschleunigt hat [HIST]. *Trolltech* wurde Anfang 2008 von Nokia angekauft und seitdem hat sich der Name von *Trolltech* 2-mal geändert und heißt heute *Qt Development Frameworks* [TROLL].

Mithilfe von Qt kann man Anwendungen sowohl für mobile Betriebssysteme wie *Symbian* von Nokia als auch für Desktop-Betriebssysteme wie *MS Windows*, *Mac OS X* und *Linux* entwickeln. Es existiert auch Anbindungen von Qt zu andere Programmiersprachen außer C++. Zum Beispiel für Java existiert Qt Jambi, für Python gibt es PyQt oder PySide [ANB]. Es gibt heutzutage immer mehr neue

Anbindungen, die für die aktuellen Plattform wie Android oder Iphone noch experimentell entwickelt sind.

Neben der Entwicklung grafischer Benutzeroberflächen bietet Qt mehrere Module, die die Programmierung einer vollständigen Anwendung ermöglichen. Es gibt z.B. QtSql für die Unterstützung von SQL, QtNetwork für die Netzwerk-Entwicklung und QtXml für die Unterstützung von XML [ANB].

Mit Qt benötigt man eine Proprietäre Lizenz, sobald man eine Anwendung entwickeln will, die unter keiner freien Lizenz stehen wird. Qt steht deshalb unter einer sogenannten Mehrfachlizenzierung zur Verfügung(GPL Version 3 und Proprietär). Es ist auch zu notieren, dass ab der Version 4.5 von Qt, steht sie auch zusätzlich unter LGPL Version 2.1 [ANB]. Diese LGPL Lizenz führt dazu, dass man eine proprietäre Anwendung entwickeln kann mit einer kostenlosen Lizenz und ohne den Quellcode veröffentlichen zu müssen [LGPL].

Qt verwendet einen Präprozessor, um die Fähigkeiten von C++ zu erweitern: Im Standard-C++ existieren Z.B Konzepte wie Signale und Slots nicht, was aber in Qt eingeführt ist. Der erzeugte Code folgt damit dem C++-Standard, so dass die existierenden C++ Compiler problemlos ihn übersetzen können.

Um bequem mit Qt zu programmieren bietet Nokia einen SDK, der unter anderen die Qt-Bibliotheken, der GCC-Compiler, die Dokumentation und *Qt Creator* beinhaltet. Es kann aber auch mit anderen Entwicklungsumgebungen wie *MS Visual Studio* oder *Code::Blocks* eine Qt-Anwendung hergestellt werden.

Die aktuelle Version vom Qt ist 4.7.4(seit dem 1. September 2011).

2. Qt Creator

Qt Creator ist eine integrierte Entwicklungsumgebung (IDE), die von Nokia zur verfügung gestellt ist und ermöglich die Entwicklung von Qt-Anwendungen auf verschiedene Plattformen(*MS Windows*, Linux, Mac OS X). Da in dem SDK von Nokia es ein Compiler enthalten ist, enthält Qt Creator keinen mehr. Die aktuelle Version von Qt Creator ist die 2.3.1 [CREA].

Die Hauptkomponenten von Qt Creator sind unter anderen:

- Eine C++ und JavaScript Code-Editor mit Autovervollständigung und Syntaxhervohebung
- Eine integrierte Fenster Editor(*Qt Designer*), um einfach per Maus Fenster zu Zeichnen
- Eine komplette und sehr hilreiche Dokumentation über Qt (*Qt Assistant*)
- Eine internationalisierungswerkezeug(*Qt Linguist*), das die Übersetzung und Internationalisierung von Anwendungen beschleunigt.

3. Klassenhierarchie in Qt

Qt ist wie C++ eine Objektorientierte Programmiersprache(OOP). Die Vererbung spielt deshalb eine sehr wichtige Rolle: alle Objekte in Qt nutzen diese Möglichkeit und sind eine Spezialisierung der Basisklasse QObjekt. Ein Beispiel von der Klassenhierarchie in Qt sieht so aus:

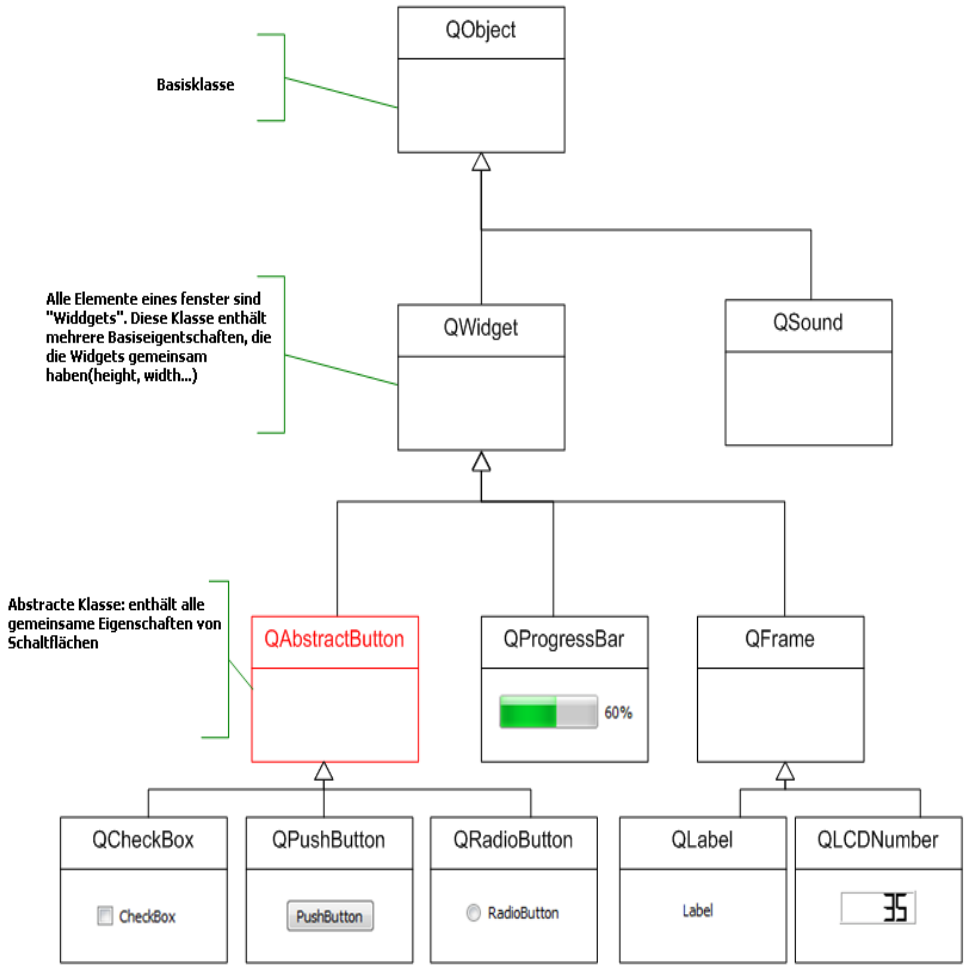

Abbildung1: Beispiel Klassenhierarchie in Qt [HIERA]

Wie es auf dem Abbildung 1 zu sehen ist, fangen alle Objektnamen mit „Q" an. Eine vollständige Dokumentation zu jeder Klasse ist in Qt Assistant zu finden oder im Internet unter der Adresse von Nokia: http://doc.qt.nokia.com/4.7/index.html .

Im Gegensatz zu einem normalen Objekt, das nur Eigenschaften und Methoden enthält, hat ein Objekt in Qt noch weitere Felder und zwar Signale und Slots, um Ereignisse zu verwalten.

4. Signale und Slots

Sie steuern die Kommunikation zwischen die Objekte des Programms. Sie sind eine besondere Art und Weise Ereignisse zu behandeln, wenn sie eintreten. Signale werden emittiert, sobald ein Ereignis eintritt. Ein Slot ist eine Methode oder Funktion, die mit einem Signal verbunden wird. Es enthält dann die Behandlung des Signals(was gemacht werden soll, wenn das Signal emittiert wird) [SIGN].

Die Objekte in Qt haben vorgegebene Signale und Slots, können aber von dem Entwickler selbst implementiert werden falls es kein geeignete gibt. Die Verknüpfung erfolgt mithilfe der statischen Methode „*connect*" von QObjekt.

Syntax:

```
QObject::connect(&sender, SIGNAL(SendSignal()), &receiver,
SLOT(doSomething()));
```

Sender ist das sendende Objekt bei dem ein Ereignis eigetroffen ist und *receiver* das Zielobject, das auf dem Ereignis reagieren soll. *Sender* und *receiver* können auch das gleiche Objekt repräsentieren.

II. Implementierung vom Damenspiel

Hier wird erklärt wie die graphische Oberfläche des Spiels entstanden ist. Zuerst wird einen Screenshot des Spiels angezeigt, dann die Architektur der Klassen(Klassendiagramm) und die Implementierung der wichtigsten Klassen des Spiels.

1. Screenshot

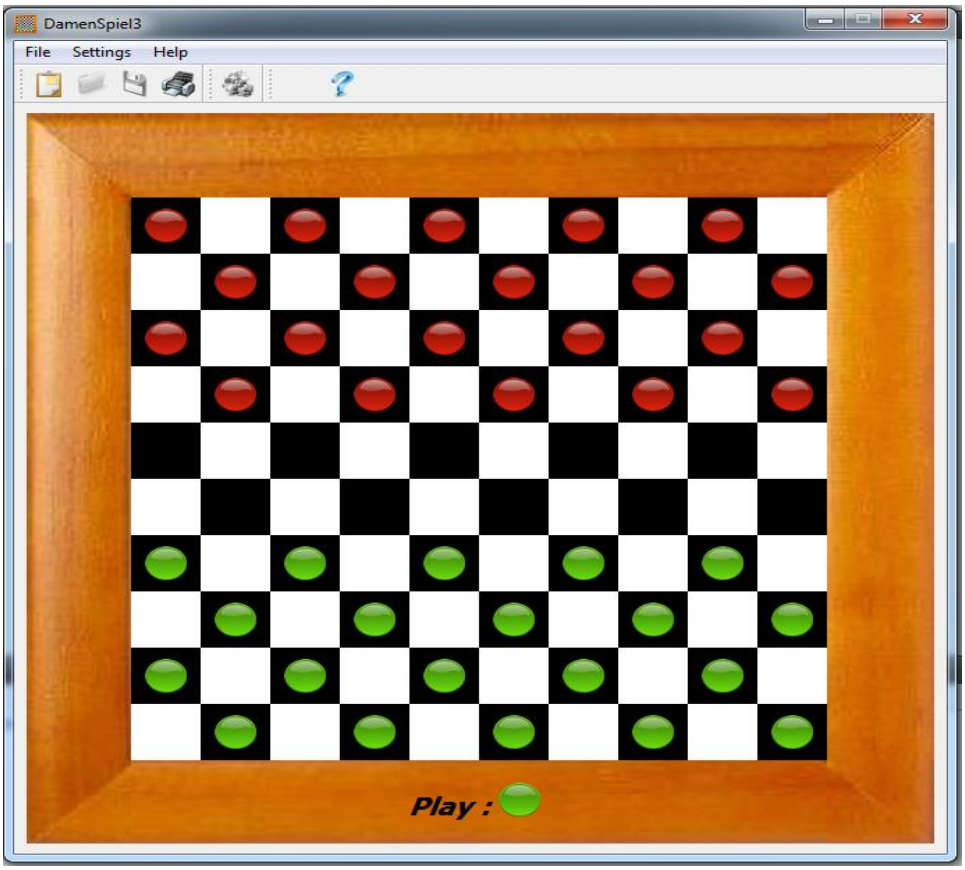

Abbildung2: Screenshot vom Damenspiel

5. Architektur des Spiels

Zu Jeder Klasse gehören eine Implementierungsdatei(.c++) und eine Headerdatei(.h). Zusätzlich gibt es für die Fenster, die mit Qt Designer gezeichnet wurde, eine UI-Datei(.ui).

Eine Ressourcendatei(.qrc) wurde auch erstellt und beinhaltet alle Ressourcen(Bilder, Klänge), die die Anwendung benutzt.

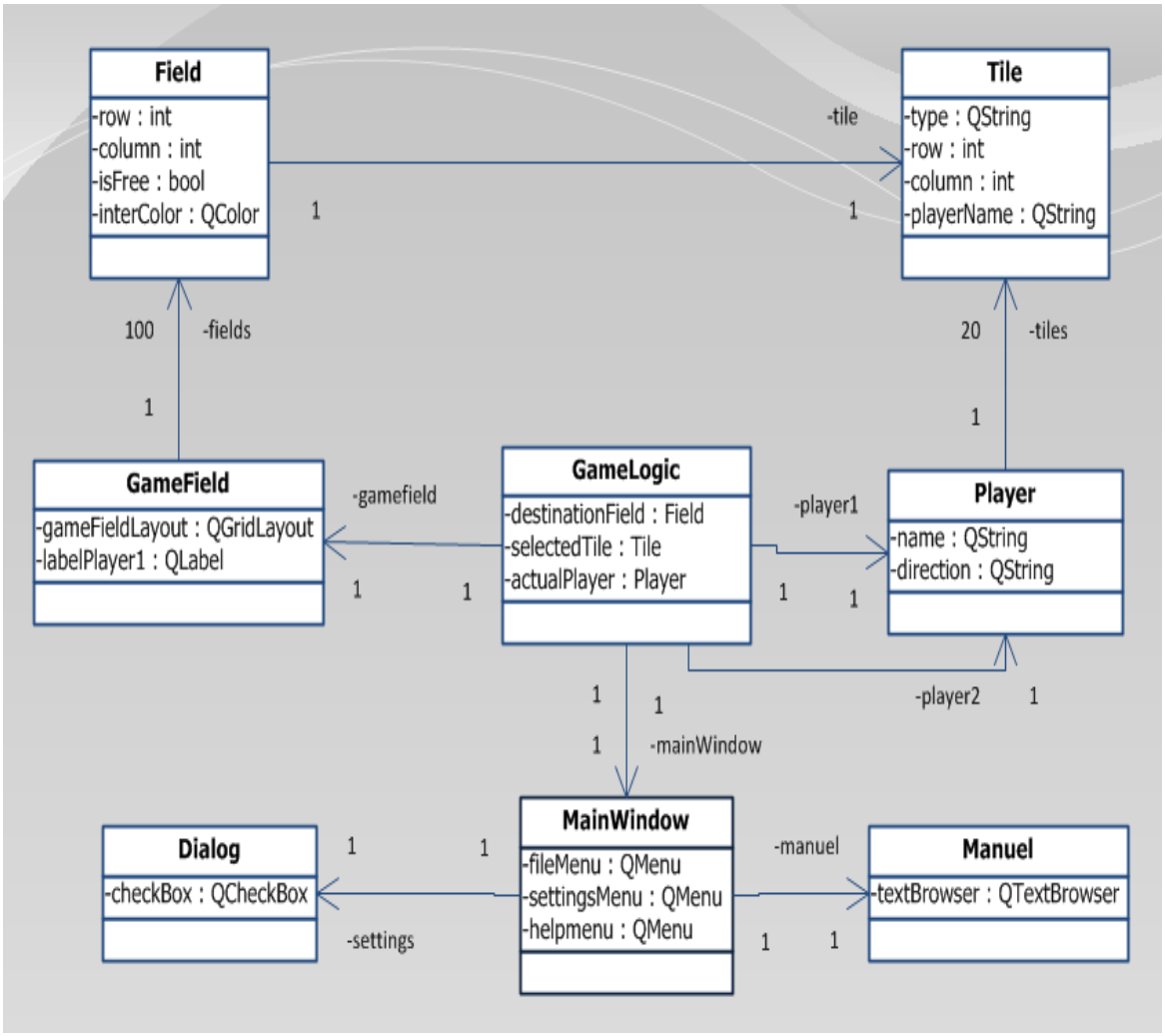

Abbildung3: Klassendiagramm

6. Implementierung der wichtigsten(sichtbaren) Klassen

- Tile

Diese Klasse repräsentiert einen Spielstein. Sie ist eine Spezielisierung von QLabel und ihre Definition sieht so aus:

```
#include<QtGui>
class Tile : public QLabel{
    Q_OBJECT
private:
    //type of the tile "normal" or "king"
    QString type;
    //the number of the row where the tile is situated
    int row;
    //the number of the column where the tile is situated
    int column;
    //the player who owns the tile
    QString playerName;
public:
    Tile(QString type, int row, int Column, QString playerName);
```

```
    QString getType();
    QString getPlayerName();
    int getRow();
    int getColumn();
    void setType(QString newType);
    void setRow(int newRow);
    void setColumn(int newColumn);
    void setPlayerName(QString newPlayerName);
    void mousePressEvent(QMouseEvent *ev);
   ~Tile();
signals:
    //emit that signal when the tile is clicked
    void clicked();
      };
```

In der Konstruktor wird nur die Grundinformationen des Objekt ermittelt. Das Objekt nimmt sein normales Aussehen(wie im Spiel auf der Abbildung2), nur wenn die Funktion *setPixmap(const QPixmap &)* aufgerufen wird. In unseren Fall z.B:

```
tile = new Tile("normal", -1, -1, "name");
QPixmap redTile(":/images/redTile2.png");
tile->setPixmap(redTile);
```

der Spielstein sieht dann so aus:

Abbildung4: ein roter Spielstein

Normalerweise schickt ein QLabel kein Signal, wenn er geklickt ist. Deshalb wurde ein eigenes Signal in der Klasse *Tile(sie ist ein Spezialisierung von QLabel)* eingefügt. Dieses Signal wird emittiert sobald das Ereignis *mousePressEvent* von *Tile* ausgelöst wird. Wir brauchen deshald die Methode *mousePressEvent(QmouseEvent *ev)* zu überladen. Nach der implementierung sieht die Methode so aus:

```
void Tile::mousePressEvent(QMouseEvent *ev){
    if (ev->button() == Qt::LeftButton){
        emit clicked();
    }
}
```

Das emittierte Signal wird mit der Logik-Klasse verknüpft, um zum Beispiel ein bestimmter Stein zu selektieren und die mögliche Zielfelder anzuzeigen.

Jede Klasse, die ein Signal oder Slot implementiert muss das Schlüsselwort *QObjekt* als erstes Element definieren sonst wird das Programm nicht kompilieren können.

- Field

Die Klasse *Field* hat fast dieselbe logische Implementierung von *Tile, da sie eine* Spezialisierung von *QLabel* ist und dient zur Darstellung von den Feldern, wo die Spielsteine stehen.

Sie sieht auf dem Spiel so aus:

oder

Abbildung5: schwarzes und weißes Spielfeld

- Gamefield

Diese Klasse ist eine Spezialisierung von *Qwidget* und repräsentiert das gesamte Spielfeld: die einzelne und alternierende Spielfelder und die Rahmen, die als Holz aussehen.

Die Definition der Klasse sieht so aus:

```
class GameField : public Qwidget
{
    Q_OBJECT
private:
    //Array which contains the different fields of the game area
    Field *fields[10][10];
    //the Layout which contains the game area
    QgridLayout *gameFieldLayout;
    //the Layout which contains the game area and the wood's border of the
game area
    QgridLayout *borderLayout;
    //the label in the mittle of the bottom border. It shows who has to
move
    Qlabel *labelWhoPlays;
public:
    GameField();
    void mousePressEvent(QmouseEvent *ev);
    Field* getField(int I, int j);
    QgridLayout* getGameFieldLayout();
    void paintEvent(QpaintEvent *pe);
    Qlabel *getLabelWhoPlays();
    void setLabelWhoPlays
    (Qstring text);
    ~GameField();
public slots:
    //void selectPossibleNeighbors();
signals:
    void clicked();
};
```

Mithilfe von Layouts können Elemente eines Widgets an bestimmte Stellen positioniert werden. In unserem Fall wurden 2 Layouts benutzt. Das erste für die Schwarze und weisse Spielfelder und ein anderes, das das erste Layout enthält und die Rahmen aus Holz addiert.

Der Code für die Herstellung der Spielfelder sieht so aus:

```
gameFieldLayout = new QGridLayout;
    for (int i = 0; i < Data::NUMBER_ROWS; i++){
        for(int j = 0; j < Data::NUMBER_COLUMNS; j++){
            if ((i % 2 == 0 && j % 2 == 0) ||(i % 2 == 1 && j % 2 == 1)){
                fields[i][j] = new Field(Qt::black, i, j);
            } else {
                fields[i][j] = new Field(Qt::white, i, j);
            }
           gameFieldLayout->addWidget(fields[i][j], i, j);
       }
    }
 gameFieldLayout->setHorizontalSpacing(0);
 gameFieldLayout->setVerticalSpacing(0);
```

Am Anfang ist ein *QGridLayout*-Objekt erstellt, dann sind alternierend die schwarzen und weißen Felder instanziiert, ihre Zeiger in einem Array gespeichert und auf den entsprechenden Stellen des Spielfelds platziert(mithilfe der Funktion *addWidget*).

Die Funktion *addWidget* addiert ein gegebenes *Widget* zu einem Zell der *QGridLayout*. Die Koordinaten des Anfangs des Zells müssen als Parameter der Funktion mit dem Widget übergeben werden. Wenn das *Widget* mehr als ein Zell des *QGridLayouts* nehmen soll, müssen auch zusätzlich r*owSpan* und *columnSpan* übergeben werden. Sie bestimmen wie viele Zeile und Spalte das *Widget* nehmen wird.

setHorizontalSpacing und *setVerticalSpacing* sind nötig, um den Abstand zwischen den Zellen des Layouts zu definieren. In unserem Fall ist dieser gleich 0.

Diese *QGridLayout* sieht dann so aus:

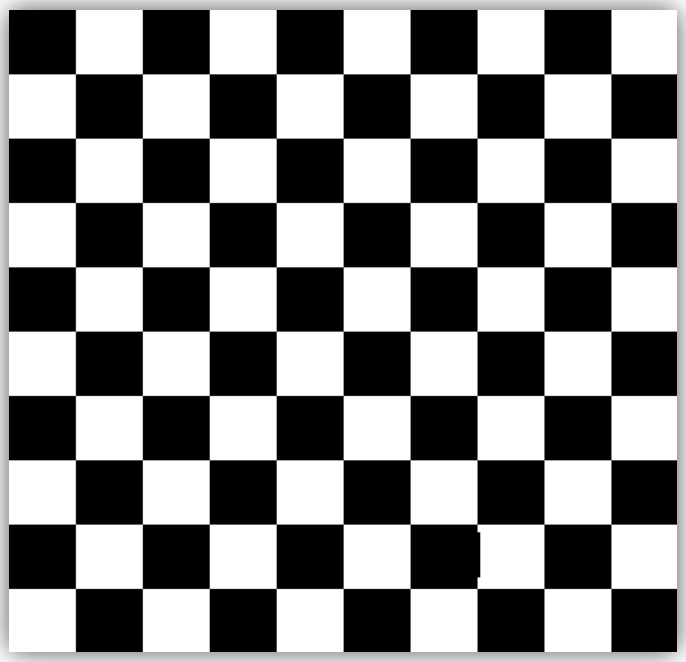

Abbildung6: *QGridLayout* mit den Spielfeldern

Um das gesamte Spielfeld zu bekommen, muss noch den Rahmen addiert werden.

Der Code dafür sieht so aus:

```
    borderLayout = new QGridLayout;

    borderLayout->addWidget(borderTopLeftLabel, 0, 0, 1, 1);              (A)
    borderLayout->addWidget(borderTopLabel, 0, 1, 1, 10);                 (B)
    borderLayout->addWidget(borderTopRightLabel, 0, 11, 1, 1);            (C)
    borderLayout->addWidget(borderLeftLabel, 1, 0, 10, 1);                (D)
    //die Spielfelder in der Mitte addieren
    borderLayout->addLayout(gameFieldLayout, 1, 1, 10, 10,
Qt::AlignHCenter);                                                        (I)
    borderLayout->addWidget(borderRightLabel, 1, 11, 10, 1);              (H)
    borderLayout->addWidget(borderBottomLeftLabel, 11, 0, 1, 1);          (E)
    borderLayout->addWidget(borderBottomLabel, 11, 1, 1, 10);             (F)
    borderLayout->addWidget(labelPlayer1, 11, 1, 1, 10, Qt::AlignCenter);
    borderLayout->addWidget(borderBottomRightLabel, 11, 11, 1, 1);        (G)
    borderLayout->setHorizontalSpacing(0);
    borderLayout->setVerticalSpacing(0);
```

Es wird auch ein *QGridLayout* mit 12x12 Zellen benutzt.

Die neue Funktion ist *addLayout* (I) mit dem man ein Layout auf einem anderen platzieren kann. Sie funktioniert genau wie die *addLayout*-Funktion.

Das folgende Bild zeigt wo welches Widget auf dem neuen *QGridLayout* zu finden ist.

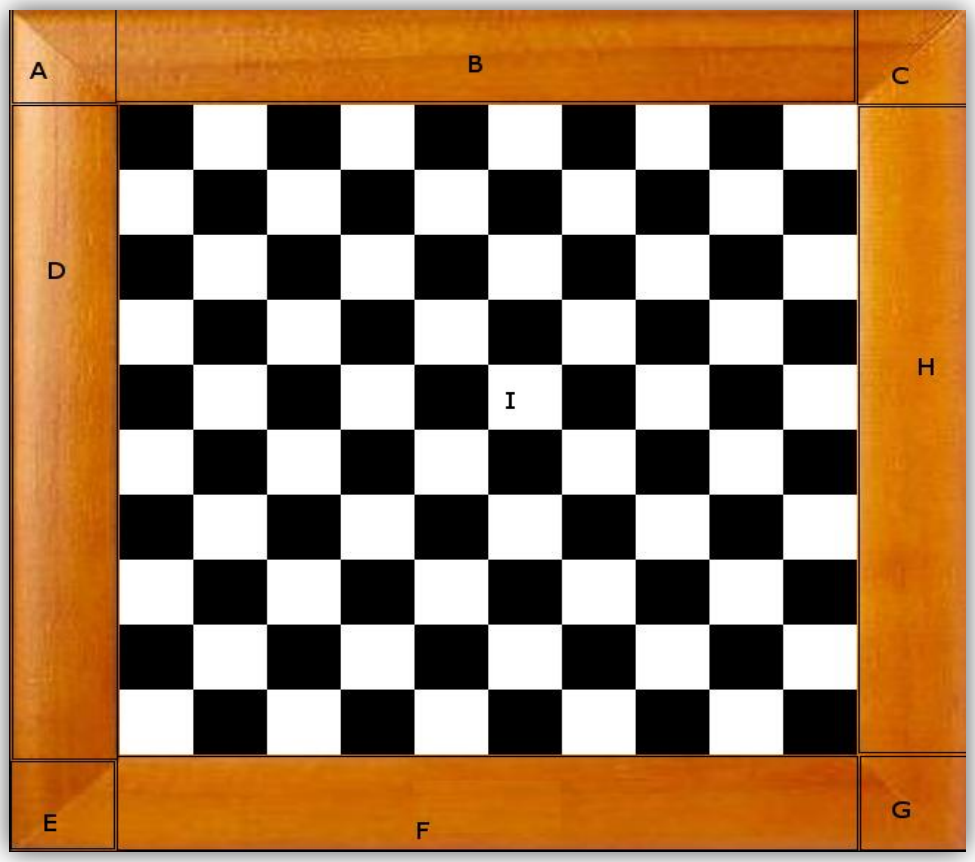

Abbildung7: Das gesamte Spielfeld

Die Rahmen sind Bilder aus der Ressourcendatei, die als *QPixmap* in einem *QLabel* addiert wird, und das *QLabel* dann mithilfe von der Funktion *addWidget* auf dem *QGridLayout* an der richtigen Stelle platziert ist.

Am Ende des Konstruktors der Klasse Gamefield wird unser *borderLayout* als Layout verwendet. Das ist mithilfe der Anweisung *setLayout(borderLayout).*

- MainWindow

Die Klasse Mainwindow ist die Klasse, die das Hauptfenster darstellt. Sie ist eine Spezialisierung der Klasse QMainwindow, die die Konzeption eines Hauptfensters für eine Anwendung erleichtert.

Dort ist jeder Menu von der Klasse QMenu und jeder Untermenu von der Klasse QAction.

Für unsere Settingsmenu war der folgende Code nötig:

```
this->settingsMenu = menuBar()->addMenu(tr("&Settings"));
```

Für den Untermenu von Settings haben wir den Code:

```
this->preferenceAction = new QAction(tr("&Preferences", this));
```

```
this->preferenceAction->setShortcut(Qt::CTRL + Qt::Key_P);
this->settingsMenu->addAction(preferenceAction);
connect(preferenceAction, SIGNAL(triggered()), this, SLOT(showSettings()));
 preferenceAction->setIcon(QIcon(":/images/Gears.png"));
```

Mit der Funktion *setShortcut* kann eine Verknüpfung zwischen einen Menu und eine Tastenkombination definiert werden und mit *setIcon* wird ein Icon definiert.

Mit der Funktion connect wird definiert welches Slot aufgerufen wird, wenn es auf dem Menu geklickt wird(triggered()-Signal).

Eine Toolbar kann auch einfach mit dem folgenden Code definiert werden.

```
toolBarSettings = addToolBar("Settings");
toolBarSettings->addAction(preferenceAction);
```

Die Funktion *addToolbar* fügt eine neue Toolbar im fester ein und *addAction* stellt die Verknüpfung zur entsprechenden Aktion dar.

Jedes *QMainWindow*-Objekt hat eine zentrale Zone, wo der Inhalt des Fensters dargestellt werden soll. Mit der Funktion *setCentralWidget* erfolgt dies. Das Parameter ist das Widget, das dargestellt wird und in unserem Fall: das GameField-Objekt.

```
mainWindow->setCentralWidget(gameField);
```

- Manuel und Dialog

Diese 2 Klasse wurden mit Qt Designer erstellt und nur die nötigen Signale und Slots wurden drin addiert.

- GameLogic

Das ist die Logik des Spiels. Dort sind alle Interaktionen zwischen die Klassen des Spiels verwaltet. Sie Stellt kein graphisches Element dar aber besitzt auch zumindest ein paar Slots. Deshalb muss sie zumindest eine Spezialisierung von QObjekt sein, um Slots definieren zu können. Die Logik kennt alle die anderen Klasse und die anderen Klassen kennen sie nicht. Die Signatur des Konstruktor sieht so aus:

```
GameLogic(Player *p1, Player *p2, GameField *gameField, MainWindow
*mainWindow);
```

- Die Main-Klasse:

Sie startet das Spiel.

```
int main(int argc, char *argv[])
{
```

```
    QApplication app(argc, argv);

    GameField *gameField = new GameField;

    Player *player1 = new Player("player1", Qt::green, "toTheTop");

    Player *player2 = new Player("player2", Qt::gray, "toTheBottom");

     GameLogic *gameLogic = new GameLogic(player1, player2, gameField,
fenetre);

    gameLogic->start();

    fenetre->show();

    return app.exec();

}
```

Fazit

Diese Projektarbeit war sehr interessant, weil man damit lernt wie eine graphische Oberfläche mit Qt 4.7 gebaut werden soll, was sehr vorteilhaft ist. Da mit Qt 4.7 ein einziger Code implementiert werden kann, der auf verschiedene Plattformen(Desktop und mobile) und Betriebssysteme laufen kann. Mit dieser Projektarbeit wurden auch die verschiedenen Schwierigkeiten eines Softwareentwicklungsprojekts von Anfang bis zum Ende erlebt und am meisten selbständig gelöst.

Anhang:

A. Abbilddungsverzeichnis

B. Quellenverzeichnis

[LGPL] Jürgen Wolf: Qt 4.6 - GUI-Entwicklung mit C++: Das umfassende Handbuch. Galileo Press, Bonn 2010. Seite 16.

[HIST] A Brief History of Qt.
URL: http://www.civilnet.cn/book/embedded/GUI/Qt4/pref04.html.
Stand: 05.07.2011

[HIERA] Nebra, Schaller : Personnaliser les widgets. 2007.
URL: http://www.siteduzero.com/tutoriel-3-11260-personnaliser-les-widgets.html. Stand: 07.07.2011

[ANB] Qt (Bibliothek)
URL: http://de.wikipedia.org/wiki/Qt_%28Bibliothek%29.
Stand: 09.08.2011

[CREA] Qt Creator Releases
URL: http://developer.qt.nokia.com/wiki/Qt_Creator_Releases.
Stand:01.10.2011

[TROLL] Qt Development Frameworks
URL: http://de.wikipedia.org/wiki/Trolltech Stand: 05.07.2011

[SIGN] Signal-Slot-Konzept
URL: http://de.wikipedia.org/wiki/Signal-Slot-Konzept. Stand: 09.09.2011