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 werden 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 anhand eines Beispiels konkret dargestellt wird. Das Thema ist an sich sehr interessant, weil seit dem Anfang des Studiums das Erstellen von graphischen Oberflächen nicht gelehrt wurde und dieses Thema die Gelegenheit bietet 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. Beschreibung

Qt ist eine Klassenbibliothek, die mit C++ geschrieben wurde[BIB]. Mit ihr kann man GUIs programmieren, die Portable sind. Das heißt mit dem Selben Code, kann die Anwendung unter verschiedene Betriebssysteme laufen. Dafür braucht man nur eine neue Kompilierung in der Zielumgebung(Zielbetriebssystem).Qt enthält auch mehrere Module außer GUI-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.

Die Entwicklung von Qt hat in 1991 mit *Haavard Nord* und *Eirik Chambe*-eng in Norwegen angefangen. Im Jahre 1994 haben die 2 Entwickler das Unternehmen *Trolltech* gegründet, das die Entwicklung von Qt sehr beschleunigt hat. *Trolltech* wurde Anfang 2008 von Nokia angekauft und seitdem hat sich der Name von *Trolltech* 2-mal geändert und heißt heute *Qt Development Frameworks*.

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. Qt bietet die interessante Möglichkeit mit anderen Programmiersprachen benutzt zu werden(sog. Anbindung). Zum Beispiel existiert für Java Qt Jambi, für Python gibt es PyQt oder PySide. Es gibt heutzutage immer mehr neue Anbindungen, die für die aktuellen Plattforme wie Android oder Iphone noch experimentell entwickelt werden.

Seit der Version 4.5 von Qt steht sie unter LGPL Lizenz, GPL Version 3 und eine proprietär Lizenz. Die LGPL Lizenz führt dazu, dass *man eine proprietäre Anwendung entwickeln kann mit einer kostenlosen Lizenz und ohne den Quellcode veröffentlichen zu müssen* [LGPL]. Es besteht aber mit dem Erwerb einer proprietären Lizenz die Möglichkeit kostenpflichtiger Anwendungen zu realisieren.

Qt hat neue Konzepte eingeführt, die im C++ normalerweise nicht enthalten sind. Ein Beispiel davon ist das Signal- und Slot Konzept, das zur Behandlung von Ereignissen im Programm dient. Ein Präprozessor wird benötigt um diese neue Konzepte umzuwandeln und einen Code zu generieren, der ein normaler C++ Compiler übersetzen kann.

Um bequem mit Qt zu programmieren bietet Nokia ein SDK, das unter anderem die Qt-Bibliotheken, den GCC-Compiler, die Dokumentation von Qt und *Qt Creator(eine IDE)* beinhaltet. Es kann aber auch mit anderen Entwicklungsumgebungen wie *Dev-C++* oder *Eclipse* 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)[CREA], die von Nokia zur verfügung gestellt wird und ermöglicht die Entwicklung von Qt-Anwendungen. Da in dem SDK von Nokia ein Compiler enthalten ist, enthält Qt Creator keinen mehr. Die aktuelle Version von Qt Creator ist 2.3.1.

Die Hauptkomponenten von Qt Creator sind unter anderem:

- Ein C++ und JavaScript Code-Editor

- Ein integrierter Fenster Editor(*Qt Designer*), um einfach per Maus Fenster zu Zeichnen
- Eine komplette und sehr hilfreiche Dokumentation über Qt (*Qt Assistant*)
- Eine internationalisierungswerkezeug(*Qt Linguist*), das die Übersetzung und Internationalisierung von Anwendungen beschleunigt.

3. Klassenhierarchie in Qt

Qt ist eine Objektorientierte Programmbibliothek. Die Vererbung spielt deshalb eine sehr wichtige Rolle: alle Klassen in Qt nutzen diese Möglichkeit und sind außer einigen Ausnahmen alle Spezialisierungen der Basisklasse QObjekt. Ein Beispiel von der Klassenhierarchie in Qt sieht so aus:

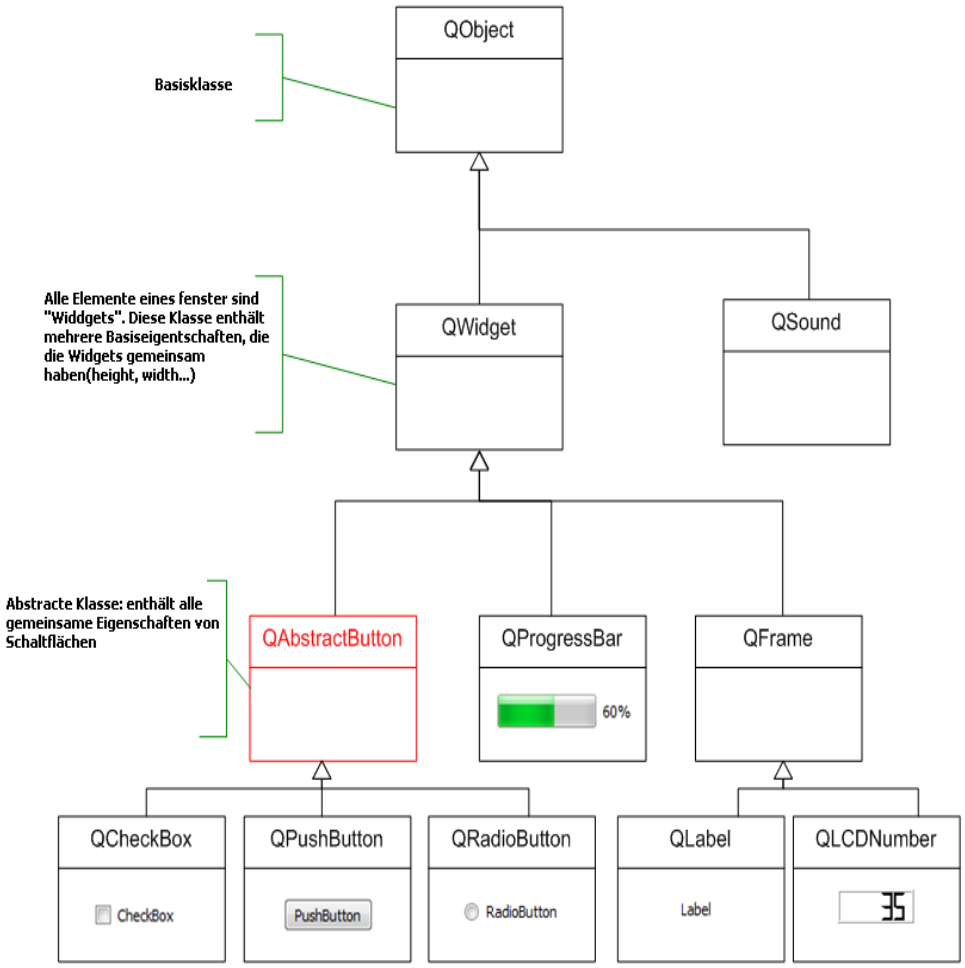

Abbildung1: Beispiel Klassenhierarchie in Qt [HIERA]

Wie es auf der Abbildung 1 zu sehen ist, fangen alle Klassennamen 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 den Objekten 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).

Die Objekte in Qt haben vorgegebene Signale und Slots, können aber von dem Entwickler selbst implementiert werden falls es kein geeigneten 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 ein Screenshot des Spiels angezeigt, dann die Architektur der Klassen(Klassendiagramm) und die Implementierung der wichtigsten Klassen des Spiels.

1. Screenshot

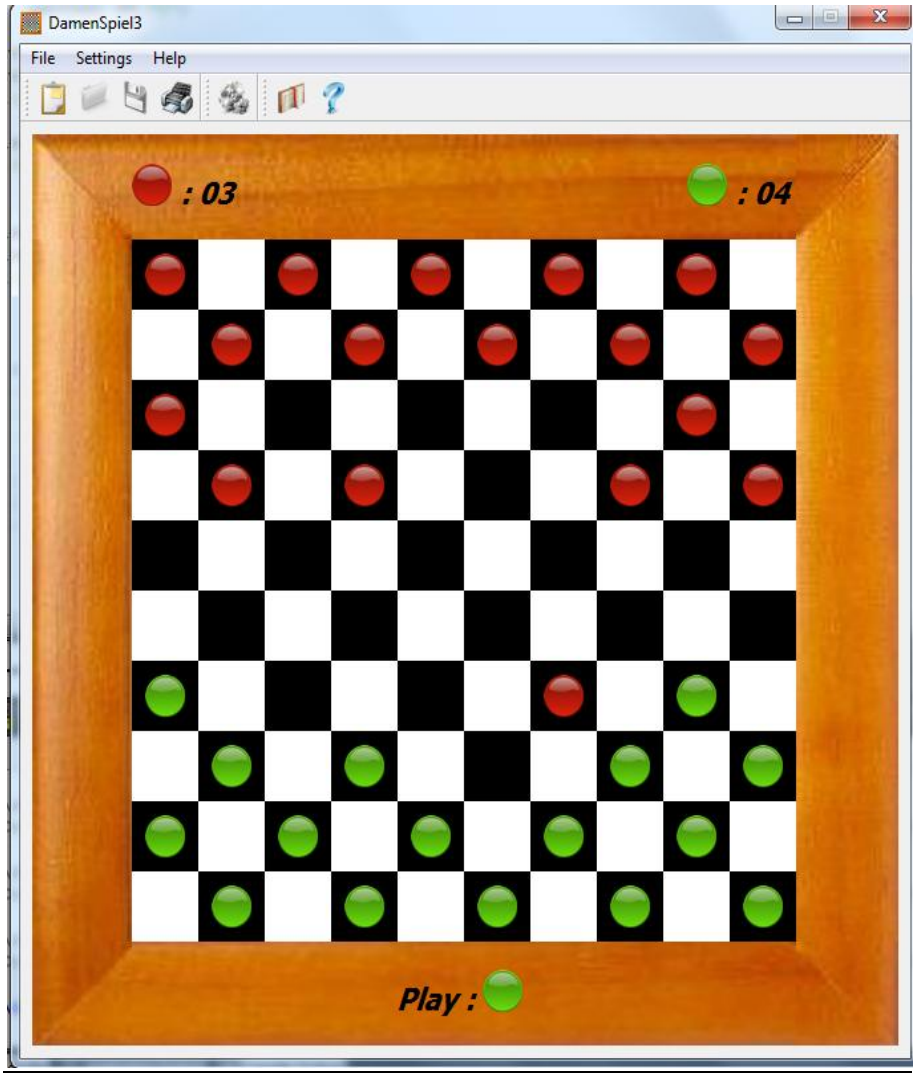

Abbildung2: Screenshot vom Damenspiel

5. Architektur des Spiels

Zu Jeder Klasse gehören eine Implementierungsdatei(.cpp) 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.

Das Klassendiagram der Anwendung sieht wie folgt aus:

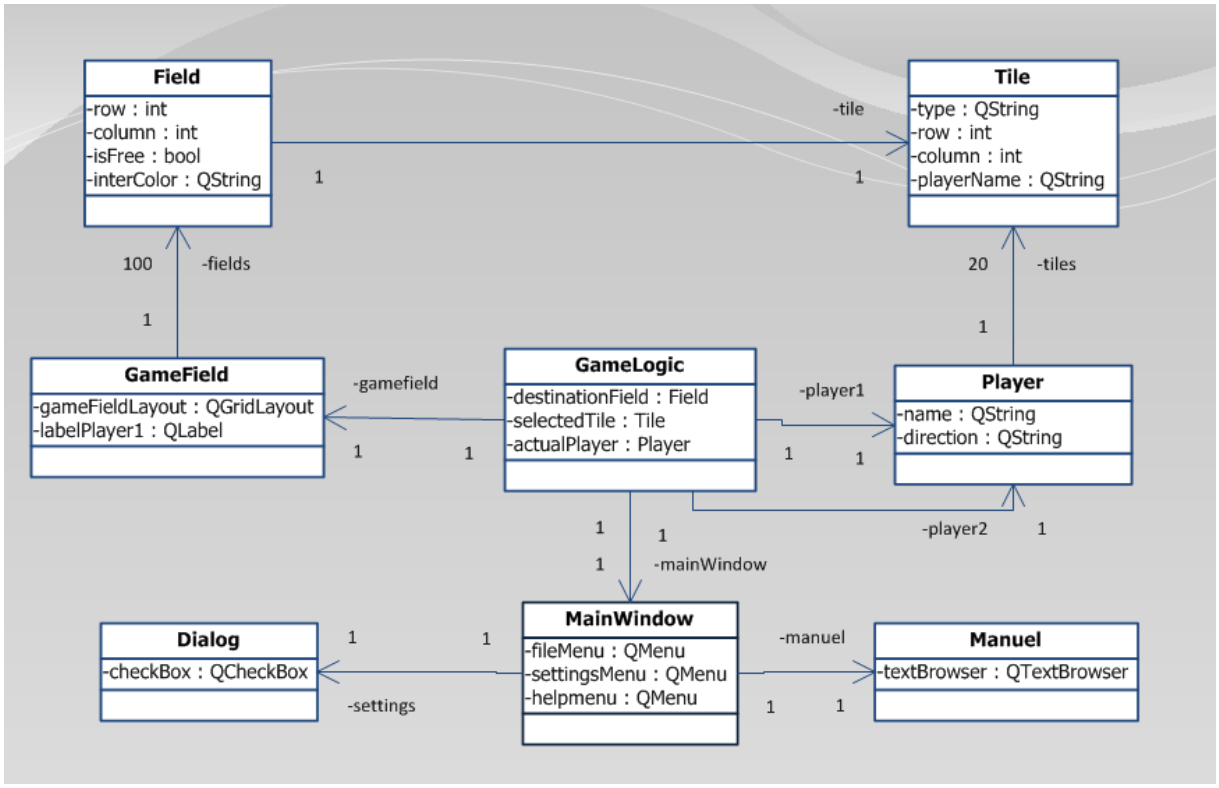

Abbildung3: Klassendiagramm

6. Implementierung der wichtigsten(sichtbaren) Klassen

- Tile

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

```
#include<QtGui>
/**
  * class which represents a tile in the game
  */
class Tile : public QLabel{
    Q_OBJECT
private:
    //type of the tile(normal or king)
    QString type;
    // row where the tile is located
    int row;
    //column where the tile ist located
    int column;
    //name of the player who ows the tile
    QString playerName;
public:
    //create a new tile
    Tile(QString type, int row, int Column, QString playerName);
    //returns the type of the tile
    QString getType() const;
    //returns the name of the player who ows the tile
    QString getPlayerName() const;
```

```
    //returns the row where the tile is located
    int getRow() const;
    //returns the column where the tile is located
    int getColumn() const;
    // change the type of a tile with the give type as parameter
    void setType(QString newType);
    //change the row of the tile with the given value in parameter
    void setRow(int newRow);
    //change the column of the tile with the given value in parameter
    void setColumn(int newColumn);
    //change the name of the player who ows the tile
    void setPlayerName(QString newPlayerName);
    //this function is called when the mouse is pressed on the tile
    void mousePressEvent(QMouseEvent *ev);
    //compare two tile and returns true when their attributes contains the
same value and false if not
    bool operator ==(Tile const& tile1);
    //destroy the tile
    virtual ~Tile();
signals:
    //this signal is emitted when the the mouse is pressed on the tile
    void clicked();
public slots:
    //void selectMe();
};
#endif // TILE_H
```

In dem Konstruktor wird nur die Grundinformationen des Objektes ermittelt. Das Objekt nimmt sein normales Aussehen(wie im Spiel auf der Abbildung2) nur an, wenn die Funktion *setPixmap(const QPixmap &)* aufgerufen wird. In unserem 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 darauf geklickt wird. 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 deshalb 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 einen bestimmten Stein zu selektieren und die möglichen Zielfelder anzuzeigen.

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

- 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, auf denen 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 einzelnen und alternierenden Spielfelder und den Rahmen, der aussieht wie Holz.

Die Definition der Klasse sieht so aus:

```
/**
  * class which represents the whole game area
  */
class GameField : public QWidget
{
    Q_OBJECT
private:
    //array which contains all the field of the game field
    Field *fields[10][10];
    //Layout on which the diferrents fields are classified and shown
    QGridLayout *gameFieldLayout;
    //Layout wich contains the borders of the game field and the game field
itself
    QGridLayout *borderLayout;
    //Label which schows who has to play
    QLabel *labelActuelPlayer;
    //Label which shows how many adverse tiles the first player has removed
    QLabel *removedTilePlayer1;
   //Label which shows how many adverse tiles the second player has removed
```

```
    QLabel *removedTilePlayer2;
public:
    //create the game area
    GameField();
    //this function is called when the mouse is
    //void mousePressEvent(QMouseEvent *ev);#
    //return the reference of the field at the row i and the column j.
    Field* getField(int i, int j);
    //return the reference on the central layout which contains the
diffrents fields
    QGridLayout* getGameFieldLayout();
    //return the reference on the Label which shows the current player
    QLabel *getLabelActuelPlayer();
    //return the reference on the Label which shows the number of tiles
which the player1 has removed
    QLabel *getLabelRemovedTilePlayer1();
    //return the reference on the Label which shows the number of tiles
which the player2 has removed
    QLabel *getLabelRemovedTilePlayer2();
    //change the text on the label which show the current player. The new
value is the one which is given as parameter
    void setLabelActuelPlayer(QString text);
    //destroy the game area
    virtual ~GameField();
signals:
    //void clicked();
};
#endif // GAMEFIELD_H
```

Mithilfe von Layouts können Elemente eines Widgets an bestimmten Stellen positioniert werden. In unserem Fall wurden 2 Layouts benutzt. Das erste für die Schwarzen und Weißen Spielfelder und ein anderes, das das erste Layout mit dem Holzrahmen enthält.

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 werden 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 einer Zelle des *QGridLayouts*. Die Koordinaten des Anfangs der Zelle müssen als Parameter der Funktion mit dem Widget übergeben werden. Wenn das *Widget* mehr als eine Zelle des *QGridLayouts* einnehmen soll, müssen auch zusätzlich r*owSpan* und *columnSpan* übergeben werden. Sie bestimmen wie viele Zeilen und Spalten 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:

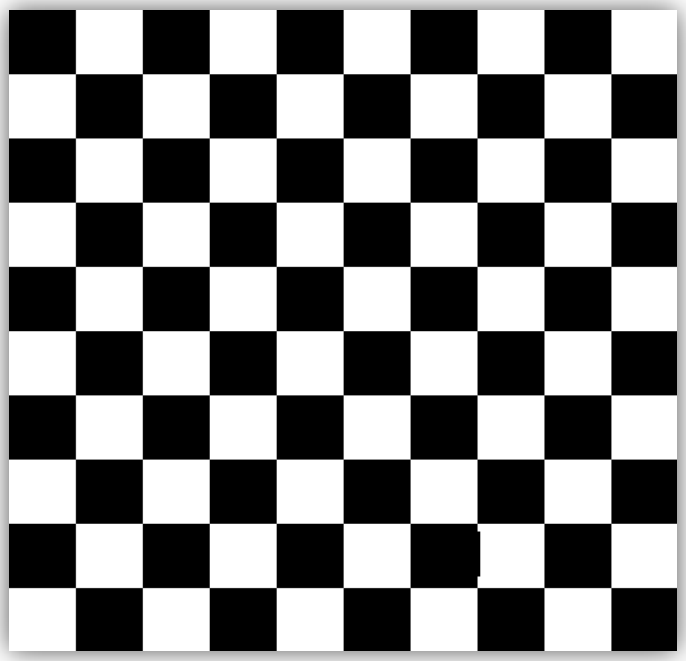

Abbildung6: *QGridLayout* mit den Spielfeldern

Um das gesamte Spielfeld zu bekommen, muss noch der Rahmen hinzugefügt 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(borderBottomRightLabel, 11, 11, 1, 1);      (G)
borderLayout->setHorizontalSpacing(0);
borderLayout->setVerticalSpacing(0);
```

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

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

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

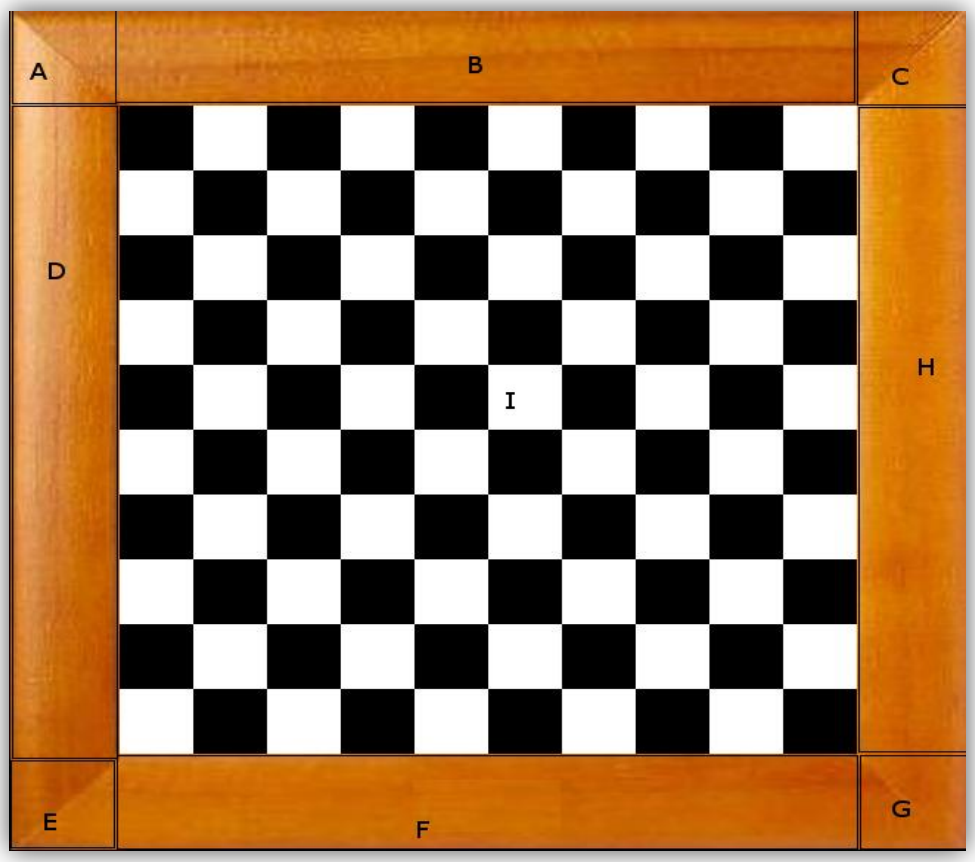

Abbildung7: Das gesamte Spielfeld

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

Am Ende des Konstruktors der Klasse Gamefield wird unser *borderLayout* als Layout verwendet. Das funktioniert 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 jedes Menu von der Klasse QMenu und jedes 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 einem Menu und einer Tastenkombination definiert werden und mit *setIcon* wird ein Icon definiert.

Mit der Funktion connect wird definiert welcher Slot aufgerufen wird, wenn auf das Menu geklickt wird(triggered()-Signal).

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

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

Die Funktion *addToolbar* fügt eine neue Toolbar im Fenster 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. Der Parameter ist das Widget, das dargestellt wird und in unserem Fall: das GameField-Objekt.

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

- Manuel und Dialog

Diese 2 Klassen wurden mit Qt Designer erstellt und nur die nötigen Signale und Slots wurden darin hinzugefügt.

- GameLogic

Das ist die Logik des Spiels. Dort sind alle Interaktionen zwischen den Klassen des Spiels verwaltet. Sie Stellt kein graphisches Element dar, aber besitzt auch einige Slots. Deshalb muss sie zumindest eine Spezialisierung von QObjekt sein, um Slots definieren zu können. Die Logik kennt alle anderen Klassen 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 erstellt wird, was sehr vorteilhaft ist. Da mit Qt 4.7 ein einziger Code implementiert werden kann, der auf verschiedenen Plattformen(Desktop und mobile) laufen kann. Mit dieser Projektarbeit wurden auch die verschiedenen Schwierigkeiten eines Softwareentwicklungsprojekts von Anfang bis zum Ende erlebt und meistens 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.

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

[BIB] 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

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

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

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