Dokumentation

Vier Gewinnt

GUI Anwendung mit dem QT Framework

Beleg im Rahmen der Veranstaltung
Entwicklung von Multimediasystemen
Angewandte Informatik
HTW-Berlin
SS2014

Emanuel Kern
s0535955@htw-berlin.de

Inhalt

1. Einleitung

Im Rahmen des Belegs wurde eine GUI Anwendung mit Hilfe des Frameworks QT erstellt, mit der sicht das Gesellschaftsspiel Vier Gewinnt an einem PC spielen lässt. Dabei besteht die Möglichkeit gegen einen zweiten menschlichen Spieler oder den Computer anzutreten.
Die Gestaltung der grafischen Oberfläche wurde mit QML umgesetzt. QML ist die QT eigene Sprache zur Beschreibung von Nutzeroberflächen. Besonderer Schwerpunkt lag auf der Einarbeitung in QML und das Mapping von QML Objekten auf QT-Klassen (C++).

2. Anmerkungen zur Abgabe

Ausführen der Anwendung: Im Ausführungsverzeichnis müssen die Dateien „tr_de.qm“, „tr_en.qm“ und „connectfourdatabase“ liegen. Die Datenbank wird von der Anwendung nicht automatisch korrekt neu erstellt, fals nicht vorhanden.

Lokalisation: Lässt sich über Optionen Einstellen, aber wird erst beim Neustart der Anwendung verändert. Ein Überladen des Eventhandlers jeder QML Klasse inklusive derer, für die gar kein C++ Backing vorgesehen war, aber lokalisierte Strings enthalten, wäre dafür nötig. Da die Implementation der Lokalisation erst am Ende erfolgte wurde das als unverhältnismäßig weggelassen.

Persistenz: Spieleinstellungen (spielspezifische, die nicht während eines laufenden Spieles verändert werden können) werden nur gespeichert, wenn ein neues Spiel gestartet wird. Weder das Laden eines Spiels noch das Verändern der Einstellungen im „NewGameScreen“ mit folgendem Abbruch verändert die Grundeinstellungen für neue Spiele.
Spiel Speichern überschreibt den alten Spielstand des gleichen Spiels, wenn dieser existiert.

Compiler Warnings: Momentan gibt es noch sehr viele Warnings, diese betreffen aber in erster Linie signed Variablen bei Zählschleifen und Warnungen bezüglich der Initialisierungsreihenfolge von Klassenmembern. Mit den richtigen Flags können diese Warnung sicherlich unterdrückt werden. Sie stellen bezüglich des Programmablaufs kein Problem dar.

Highscore Drucken: War eine Anforderung und war in meiner ursprünglichen Anwendung mit konservativer Windowsoberflächer auch enthalten, aber wurde bei der Entwicklung der QML-Oberfläche übersehen und nun weggelassen.

Timer: Eine meiner selbst gestellten Anforderungen war ein Timer. Dieser wurde leicht abgeändert. Statt das Spiel sofort zu verlieren, wenn man seinen Zug nicht innerhalb der eingestellten Zeit beendet, wird ein zufälliger Zug ausgeführt.

Gespeicherte Spiele: In der Datenbank sind einige Spiele enthalten in denen KI vs. KI Spieler antreten. Diese Spiele spielen sich „automatisch“ zu Ende, wenn man sie lädt. Sie wurden erstellt, als der „KI Spieler“ noch vom Nutzer gesteuert wurde.

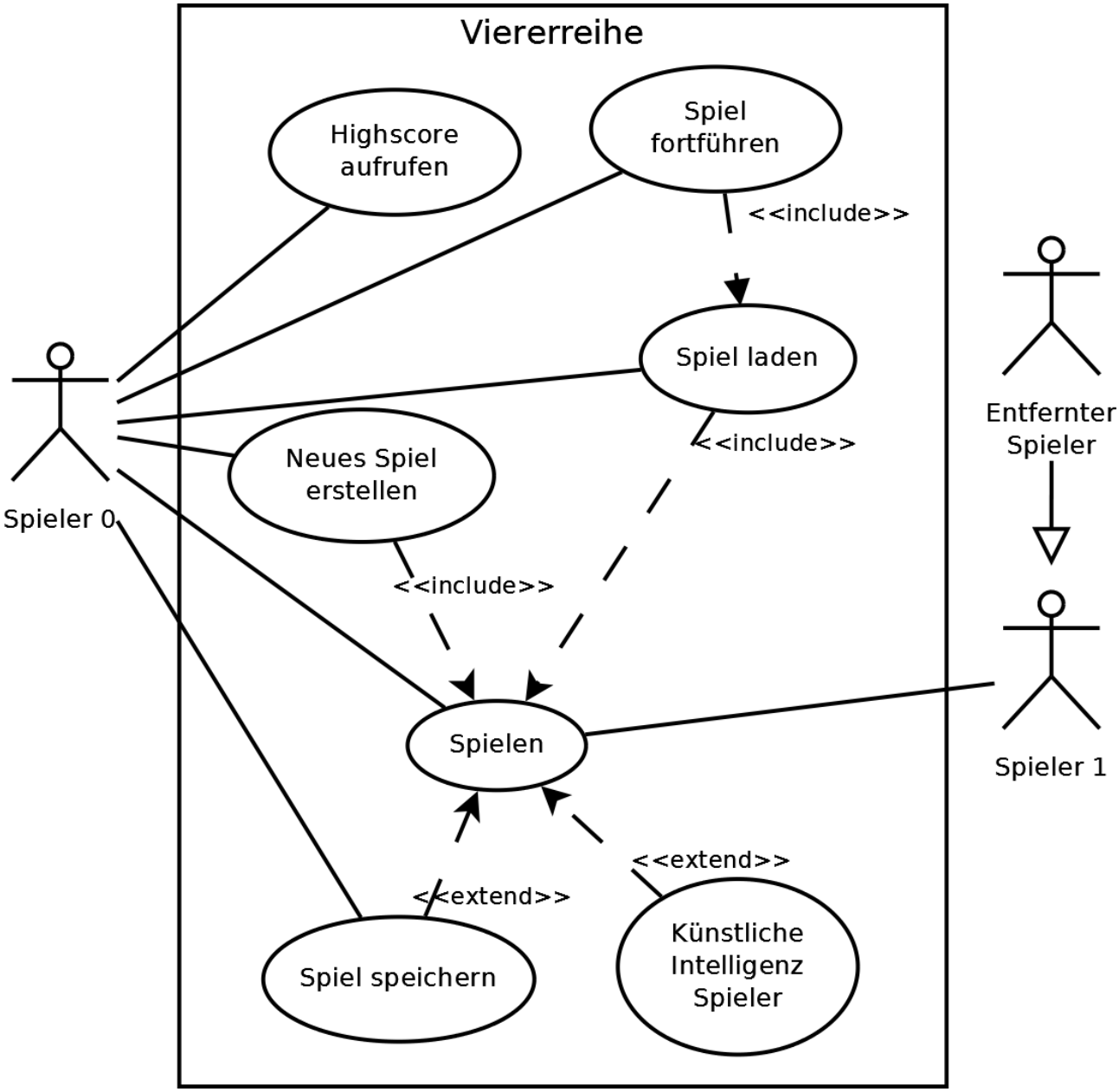
3. Werkzeuge & Entwicklungsablauf

Werkzeuge:
QtCreator - Programmentwicklung
Git (Github) – Versionsverwaltung
Inkscape, Dia – Diagramme und Symbole
Audacity – Audiofiles nachbearbeiten
LibreOffices Writer, LibreOffice Impress – Dokumentation und Präsentation

Die Entwicklung beruht auf einem angefangen Vorsemesterprojekt. Allerdings wurde die GUI komplett neu gestaltet, aber auch in den restlichen Teilen, insbesondere bei der Datenstrukturierung wurde viel verändert.
Zunächst waren die Kernelemente der Logik bereits fertig, also die Klassen mit denen sich ein Spiel ausführen lässt. Dann wurde die GUI fast vollständig in QML erstellt ohne Verbindung zur Programmlogik. Erst dann wurde die Datenhaltung und die Schnittstelle zwischen Programmlogik und Oberfläche entwickelt.
Auf Grund der geringen Erfahrung bei der Entwicklung von Benutzeroberflächen und der vollständig fehlenden Erfahrung mit QML waren immer wieder rückwirkend Anpassung an den vermeintlich fertigen Komponenten notwendig.

4. Use Cases

Im abgebildeten Use Case Diagramm ist grob die Nutzung des Programms zusammengefasst und im Folgenden sind die Uses Cases beschrieben.



Spielen: Die Spieler können abwechselnd mit der Maus einen Spielstein (Token) platzieren und durch Anklicken in eine Spalte des Spielfelds werfen. Wird ein Spieler vom Computer gesteuert, so führt dieser seine Züge automatisch aus (siehe Extension Künstliche Intelligenz)

Künstliche Intelligenz Spieler: Ein Zug wird vom Computer berechnet und ausgeführt, wenn einer der Spieler computergesteuert ist.

Spiel laden: Ein neues Spiel kann entweder durch das Laden oder „Neues Spiel erstellen“ begonnen werden. Beim Laden wird aus einer Datenbank ein Spiel ausgewählt (nicht über den Filebrowser).

Spiel fortführen: Den aktuellsten Spielstand laden (siehe Use Case Spiel laden)

Spiel speichern: Ein laufendes Spiel wird in der Datenbank gesichert. Spiel speichern wird auch beim Schließen der Anwendung während eines laufenden Spiels oder bei der Rückkehr zum Hauptmenü automatisch ausgeführt (ohne explizite Bestätigung durch den Nutzer). Jedoch nicht, wenn das Spiel ausdrücklich abgebrochen wird.

Highscore aufrufen: Eine Liste aller Spieler sortiert nach deren bisheriger Erfolgsquote bei den Spielen aufrufen.

Neues Spiel erstellen: Ein neues Spiel kann mit diversen Optionen erstellt und dann gestartet werden. Dazu gehören.

- Wahl des Spielers (Name)
- Erstellen neuer Spieler (inklusive KI Spieler)
- Einstellen eines Timers
- Einstellen der Spielfeldgröße (4-9x 4-9)

Entfernter Spieler: Der Nutzer „entfernter Spieler“ wurde nicht umgesetzt, aber die Konzeption wird in der Ausarbeitung beschrieben. Ist der Spieler 2 ein entfernter Spieler, führt er seine Züge an der gleichen Anwendung, die eine Verbindung über das lokale Netzwerk mit der laufenden aufgebaut hat, aus.

5. GUI: Gestaltung & Konzeption

Die grafische Oberfläche wurde mit QML Types erstellt, die von der C++ basierten Anwendungslogik eingebunden werden und teilweise auf explizite Qt Klassen gemapped sind. Näheres zur Schnittstelle zwischen QML und C++ findet sich in den Abschnitten Architektur und QML. Damit ist die rein grafische Repräsentation nicht nur durch voneinander getrennte Klassen, sondern auch verschiedene Programmiersprachen von der Programmlogik getrennt.

5.1 QML – Qt Modelling Language

QML ist eine CSS angelehnte deklarative Sprache. Durch den deklarativen Ansatz ist es übersichtlich und einfach möglich die grafische Oberfläche zu beschreiben (Anordnung, Farbgebung usw.). Zusätzlich ist der Code, um einen QML Type einzubinden sehr schlank im Vergleich zur Verwendung oder sogar Vererbung von entsprechenden QT Klassen in C++.
Bsp.:

```
Button {
  width: parent.width/10
  height: parent.height/10
}
```

Definiert ein Objekt vom Typ Button in der oberen linke Ecke seines Elternobjektes. Es nimmt ein Zehntel der Größe seines Elternobjekts ein.

Um mit C++ bzw. den in C++ geschriebenen QT-Klassen interagieren zu können ist QML um Java Script mit teilweiser Typisierung erweitert. Insbesondere lässt sich das Signal-Slot-Konzept einbinden. Dadurch ist QML um eine imperativen Ansatz erweitert, der mit dem Programmierparadigma von C++ ausreichend verknüpft werden kann.
Bsp.: („//...“ bedeutet, dass dort noch andere Deklarationen stehen können wie z.B. Größe, Icon, Beschriftung stehen könnten.)

QML:
```
Button {
   //...
   // Dieser Button ist bei der Erstellung unsichtbar
   visbile: false
   // C++ definiertes Signal auswerten
   onActivate: visible=true

   // Slot auf der C++ Klasse aufrufen
   onClicked: myButtonClicked()
   //...
}
```

C++:
```
//...
signals:
   void activate();
public slots:
   void myButtonClicked();
```

1. Der Button ist zunächst unsichtbar
2. Wird im C++ Code das Signal activate() aufgerufen, so wird im QML Code onActivate ausgewertet. Mit dem prefix on wird also automatisch ein Slot für dieses Signal erstellt.
3. Der nun sichtbare Button wird angeklickt. Im QML Code wird das Signal clicked emittiert (buttoneigenes Signal).
4. Im QML Code selbst wird das Signal ausgewertet und der Slot myButtonClicked() im C++ Code aufgerufen.

Damit ist sowohl Kommunikation von C++ nach QML als auch andersherum möglich.
Das Beispiel ist vereinfacht. Um das Beispiel lauffähig zu machen, muss die zum Button korrespondierende QT Klasse vererbt werden. Der QML Code muss im C++ Code registriert und geladen werden und die vererbte Klasse auf den QML Typ gemapped werden. Die eigentliche Kommunikation verläuft aber auf die beschriebene Art und Weise ab.

<u>Bindings:</u>
Bindings sind ein wichtiges Konzept in QML, das möglichst häufig benutz wird. Durch den deklarativen Ansatz der Sprache sind sie eigentlich logisch können aber durch imperativen Code zertört werden. Ein Binding im Sinne von QML bedeutet, dass eine Zuweisung (imperativ) dynamisch gilt.
```
height: mainWindow.height * 0.1
```

Wäre dies imperativ, dann würde height im Moment der Zuweisung ein bestimmter Wert zugeordnet. In QML bedeutet es aber, dass height immer 1/10 der Höhe von mainWindow hat, solange dieses Binding gilt. Verändert sicht mainWindow.height, so verändert sich auch height dynamisch.

Weist man height im JavaScriptCode (z.B. bei der Auswertung eines Signals allerdings Imperativ einen Wert zu, so wird das Binding gebrochen. Dies immer im Hinterkopf zu haben und wann immer möglich zu vermeiden war eines der zentralen Probleme bei der Entwicklung der QML Oberfläche.
Gleichzeitig stellen die Bindings aber das mächtigste Werkzeug dar eine dynamisch korrekt funktionierende Anwendung zu schreiben.

style.js
Um möglichst weitgehend von Bindings zu profitieren wurden Parameter, die für die gesamte Oberfläche gelten sollen in einer kleinen JavaScript Bibliothek zusammengefasst. Hier stehen zum Beispiel Farbwerte und Größenangaben, insbesondere für die Schriftgröße angelehnt an html Bezeichnungen (h1, h2, h3,..., p). Eingehen möchte ich auf die Größenangaben. Diese werden nämlich nur durch die Vorwärtsdeklaration einer Java Script Funktion in der Bibliothek deklariert. Den Funktionsvariablen wird erst von der Anwendung eine Funktionsdefinition zugeordnet. Dadurch können die so definierten Funktionen bereits auf Bindings zum Hauptfenster zurückgreifen (vergleichbar mit Callbackfunktionen). Die Funktionen können dann über die Bibliothek von allen QML Typen wie Bindings verwendet werden, da sie keinen statischen sondern dynamischen Wert liefern.
Das erspart die Weitergabe von (eigentlich global gültigen) Bindings durch die gesamte Hierarchy von verwendeten Typen.
Bsp.:
Ein Textfeld soll die Schriftgröße „p“ verwenden.
Das Textfeld ist aber Teil eines Buttons, der Teil einer Liste ist, die Teil eines Items zur Zusammenfassung der Liste ist. Dieses Item ist auf dem Hauptbildschirm angeordnet:

Textfeld → Button → Liste → Item → Hauptfenster

Wäre die Schriftgröße nur im Hauptfenster deklariert, dann müsste zwischen allen diesen Objekten ein Binding manuell erstellt werden. Passiert das für alle verwendeten Parameter, so wird der QML Code unnötig überfrachtet.

Types und Styles:
In QML hat man - wie auch im C++ Framework von QT - Zugriff auf etliche vordefinierte Typen mit korrespondieren C++ Klassen des Frameworks. Da die Oberfläche mobilen Geräten gerecht werden sollte (siehe folgender Abschnitt) konnten aber nicht die Standardtypen benutzt werden. QML ist explizit dafür ausgelegt für mobile Plattformen zu entwickeln. Daher gibt es zu vielen Standardtypen sogenannte Styletypen. Das Konzept ist vergleichbar mit abstrakten Klassen in C++. Einem Style müssen die nicht definierten Elemente zugewiesen werden vergleichbar mit der Implementation abstrakter Methoden. Das können Parameter oder wiederum neue QML Typen sein. Die eigentliche Funktionalität ist aber bereits implementiert.
Bsp.: Checkbox

Eine Checkbox ist im einfachsten Fall zusammengesetzt aus folgendem:

- klickbare Fläche
- Hintergrund
- Einer grafischen Repräsentation des aktiven Zustands

Würde man die Checkbox vollständig neu programmieren müsste die Mausinteraktion, die entsprechenden Signale und Zustände usw. implementiert werden. Stattdessen weist man dem Style einfach die notwendigen Elemente zu.

Ein Beispiel im Projekt ist OptionsSwitch.qml, das eine Checkbox mit großer Interaktionsfläche, einem Label und einem Bild implementiert.
Wann immer möglich wurde versucht bereits vorhandene Typen mit einem eigenem Typ-Style zu implementieren.

5.2 Mittel der Gestaltung für hybride Nutzung von Desktopsystem und mobilen Geräten

Die Anwendung ist nur auf einem Windows7-Desktopsystem getestet. Die Gestaltung der Oberfläche ist aber so gehalten, dass die Anwendung auf mobilen Geräten sowie Desktopsystemen benutzt werden kann. Folgende Aspekte wurden beachtet, um dies zu gewährleisten:

Maussteuerung:
Die Steuerung der Anwendung geschieht ausschließlich mit der Maus, auf Maushovereffekte wurde verzichtet. Damit wäre die Anwendung gleichermaßen mit einer Touchsteuerung nutzbar.

Scrollflächen:
Die High Score und das Laden von Spielen benötigt scrollfähige Elemente, da eine dynamische Anzahl von Einträgen angezeigt werden muss. Statt eine Scrollbar zu benutzen, lassen sich die Flächen ziehen. Beim Ziehen wird dann eine kleine Leiste rechts als Indikator angezeigt, wie es bei mobilen Geräten üblich ist.

Große interaktive Elemente:
Auf Standardelemente, wie bei Desktopsystemen üblich (Buttons, Checkboxes, Comboboxes mit Standard Look&Feel) wurde verzichtet. Stattdessen sind die Interaktionselemente möglichst groß gehalten, was eine Touchsteuerung, aber auch die Maussteuerung erleichtert.

Elemente passen sich Fenstergröße und Seitenverhältnis an:
Dadurch wird insbesondere ermöglicht, dass auf einem mobilen Gerät sowohl Hochkant- als auch Breitbildformat benutzbar sind. Auf einem Desktopsystem kann die Fenstergröße angepasst werden. Extreme Hoch oder Breitformate werden allerdings nicht voll ausgenutzt.
Insbesondere sind alle Größenangaben in der Oberflächendefinition relativ und viele Angaben stehen in Relation zu anderen. Pixelgenaue Angaben wurden nie verwendet. Dadurch ist die GUI für verschiedene Auflösungen gleichermaßen gültig.

Verzicht auf modale Dialoge
Auf mobilen Geräten ist es üblich (nur Erfahrungswert), dass Daten automatisch persistiert werden oder auch automatisch verworfen. Gerade wenn man über die Zurück Taste zur zentralen Benutzeroberfläche zurückkehrt, geht man davon aus, dass man die Anwendung ohne Daten verloren zu haben wieder aufrufen kann. Gespeichert wird deswegen nur dann nicht, wenn ein Spiel ausdrücklich abgebrochen wird. Ansonsten wird automatisch gespeichert bzw. beim expliziten Speichern durch Save auch nicht nach einer Bestätigung gefragt.

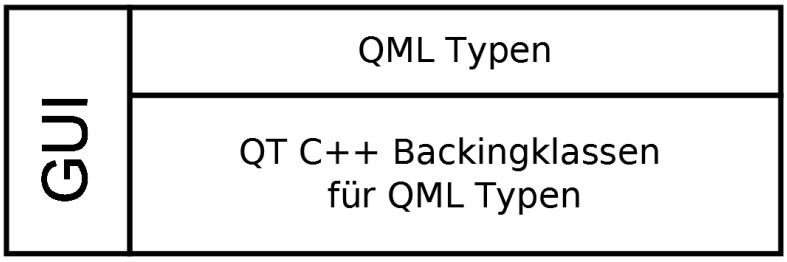
6.Architektur

Grober Ansatz:
Die Gesamtarchitektur der Anwendung folgt dem Model-View-Controller Design Pattern (MVC). Die Anwendung wurde in drei große Bereiche aufgeteilt GUI, Logik und Daten. Die Abbildung zeigt die gröbste Unterteilung.

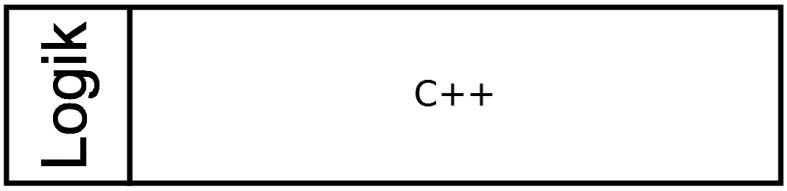
6.1 Schichten

Wichtig ist, dass die Bereiche als Schichten betrachtet werden, die aufeinander aufbauen. Das heißt, dass Klassen der Datenschicht keine Kenntnis von Klassen der Logikschicht haben und Klassen der Logikschicht die GUI nicht kennen. Dadurch ist die GUI leicht austauschbar. Es wäre zum Beispiel möglich ein anderes Framework zu verwenden.

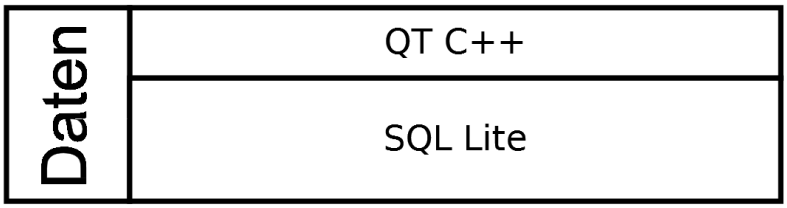
GUI:
Die GUI-Schicht setzt sich zusammen aus den QML Typen und den auf die QML Typen gamappedten Klassen. Da diese Klassen das QT Framework einbinden müssen dürfen hier auch andere QT Komfortfunktionen verwendet werden. Die C++ Klassen der GUI Schicht sind als Subschicht unter den QML Typen zu sehen, da sie die Schnittstelle zur nächsten großen Schicht, der Logik bildet.



Logik:
In der Logikschicht sind die eigentlichen Spielalgorithmen abgebildet. Diese greifen ausschließlich auf die C++ Standardbibliotheken zurück. Dadurch ist es möglich mit der Logik als Kern der Anwendung eine neue Implementation zu beginnen.



Daten:
Die Datenschicht besteht aus einer C++ Klasse, die die Schnittstelle zu einer SQL Lite Datenbank ermöglicht und vor den restlichen Klassen verbirgt und aus der SQL Lite Datenbank selbst als unterste Subschicht. Um den Zugriff auf die Datenbank zu ermöglichen muss eine entsprechende Bibliothek verwendet werden. Der Einfachheit halber wurden die entsprechenden QT Funktionen verwendet. Die Klasse kann natürlich mit einer anderen Bibliothek implementiert werden. Die definierte Schnittstelle zur Logik verzichtet auf QT- bzw. datenbankspezifische Datenstrukturen, wie die Weitergabe von ResultSets oder Queryobjekten. Die gesamte Datenschicht könnte auch über andere Ansätze (XML Files z.B.) implementiert werden, das Programm wäre genauso lauffähig, solange die Daten über das definierte Interface bezogen werden können (GameDataBase.h)



Zusätzlich zu den Klassen, die einer Schicht konkret zugeordnet sind, gibt es auch sehr schlanke Datenstrukturen, die zur Kommunikation zwischen den Schichten gedacht sind. Zum Beispiel gibt es eine Player, GameSetup und AppSetup Klasse, die jeweils die relevanten Daten zu einem Spiel, der Anwendung als ganzes oder einem Spieler enthalten. Diese finden in allen Schichten Verwendung. Sie werden der Datenschicht als Referenz übergeben um sie mit Daten aus der Datenbank zu füllen oder andersherum Änderungen in der Datenbank zu persistieren und sie werden auch von der GUI Schicht verwendet um die Daten anzuzeigen.

6.2 Klassen

Ich gehe hier nur exemplarisch und auf bestimmte Aspekte von wichtigen Klassen ein. Die einzelnen Klassen sind im DoxyGen Format im Quelltext kommentiert.

Singleton Design Pattern:

Einige Klassen, die das Grundgerüst der Anwendung, sind als Singleton Klassen ausgeführt.
Es wurden nur Klassen damit implementiert, von denen nur eine Instanz gleichzeitig existiert und diese Instanz lebt auch die gesamte Laufzeit (wird nie ausgetauscht). Diese Klassen sind also an sich statischer Natur.
Zwar widerspricht das Design Pattern ein wenig dem objektorientierten Ansatz, ist aber aus mehreren Gründen hier gut geeignet und wurde auch nur für wenige Klassen verwendet.

- Sehr viele andere Klassen greifen auf die Singleton Klassen zu. Sie fungieren als zentrale Zugriffsstelle für Daten, Ausführung von Aktionen und Datenmanipulation. Es wäre notwendig, dass diese Klassen Referenzen auf die Singleton Klassen halten. Das würde einen großen Initialisierungsoverhead erzeugen und macht es schwierig die GUI unabhängig vom Rest zu halten. Da C++ nicht über einen zwei Wege-Compiler verfügt, wären wahrscheinlich Vorwärtsdeklarationen notwendig, die den Code zusätzlich unübersichtlich machen, da die Klassen sich gegenseitig referenzieren müssten.
- Die QML-Typ-Gemappten C++ Objekte werden im Hintergrund automatisch von der QT Engine erzeugt. Um diesen die notwendigen Klassen bekannt zu machen, müsste man also mit LookUps arbeiten. Meiner Meinung nach noch deutlich unsauberer als eine geringe Abweichung vom OOKonzept.

FourInARowApplication – Singleton (Logik)

Als einzige Klasse der Logik ist diese nicht streng unabhängig von der GUI, da hier die QML Typen initialisiert werden. Wenn gerade ein Spiel läuft hält diese Klasse eine Referenz auf das laufende Spiel (FourInARowNetworkGame).
Vor der Initialisierung der GUI wird hier auch die Datenbank initialisiert.

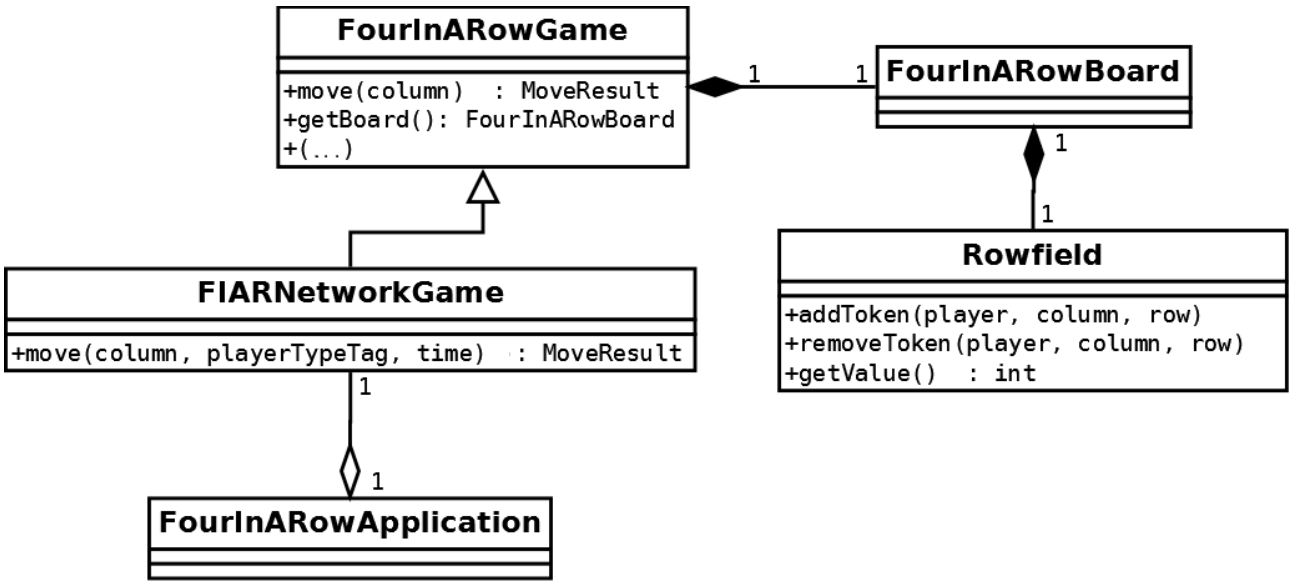
GameDataModel – Singleton (Logik)

Das GameDataModel ist die zentrale Instanz zum Austausch aller Daten. Es stellt Methoden zur Verfügung um von der Datenbank abstrahiert Zugriff zu erlangen. Die Daten werden in C++ struct Klassen aufbereitet zur Verfügung gestellt. Und über diese Klassen auch manipuliert.
Ich sehe diese Klasse noch als unterste Schicht der Logikschicht, weil hier nur eine Aufbereitung stattfindet.

GameDataBase – Singleton (Daten)

Führt die Lese- und Schreibzugriffe auf der Datenbank aus.
Während des Programmablaufs besteht nur eine Schnittstelle zum GameDataModel. Initialisiert und zerstört wird die Instanz von der FourInARowApplication Instanz.

FIARNetworkGame – FourInARowGame – FourInARowBoard – Rowfield (Logik)
Diese Klassen Implementieren die Algorithmen um ein Viereireihe Spiel durchzuführen

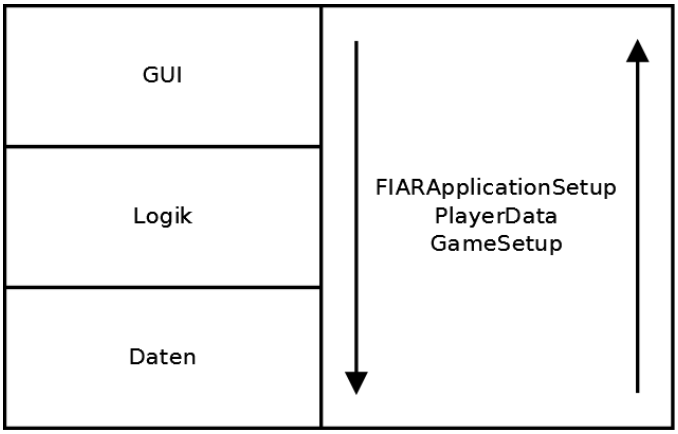


- FourInARowGame
  - ergmöglicht das Durchführen eines Spiels
  - enthält Methoden zur Konfiguration des Spiels
  - gibt nach jedem Zug den Status des Spiels zurück
- FourInARowBoard ist eine einfache Datenstruktur zur Repräsentation des Spielfeldes
  - Datenstruktur die den Zustand des Spielbretts vollständig repräsentiert
  - FourInARowGame arbeitet direkt mit den Membervariablen
  - repräsentiert Spielfeld durch ein unsigned short pro Spalte und Spieler
  - jedes gesetzte Bit bedeutet einen Spielstein
- RowField
  - zusätzliche nicht vollständige Repräsentation des Spielfelds:
  - RowField stellt einen 3 dimensionalen Raum von möglichen 4er Reihen innerhalb des Spielfelds dar. Die Dimensionen ergeben sich aus den möglichen Startpunkten (Spalte, Reihe) und der Richtung. Wobei es 4 Richtungen gibt um alle Reihen eindeutig zu beschreiben.
  - Willkürlich gewählt sind das: senkrecht nach oben, nach oben rechts, nach rechts und nach unten rechts vom ersten Spielstein der Reihe ausgehend.
  - An jedem Punkt dieses Raumes ist gespeichert, wie viele Spielsteine welches Spielers bereits in einer potentliellen Viererreihe liegen. Offensichtlich kann jeder gesetzte Spielstein mehreren potentiellen Reihen zugeordnet werden. Deswegen wird bei jedem Zug der erfolgreich ausgeführt wird, jeder Reihe, die dieser Stein angehören kann im RowField ein Spielstein hinzugefügt und direkt geprüft, ob die Anzahl 4 erreicht hat. Auf diese Weise ist nach jedem Zug sehr schnell zu ermitteln, ob das Spiel beendet wurde.

  - Jede Position im RowField nimmt nur ein Byte in Anspruch. Die niedrigsten 2 bit repräsentieren die Anzahl von Spieler 0, das 3. und 4. bit die Anzahl von Spieler 1. Die restlichen 4 bit werden nicht gebraucht. Damit sind die Zahlen 0-3 darstellbar. Versucht man eine 3er Reihe zu erhöhen, wird dies festgestellt und als Sieg des jeweiligen Spielers ausgewertet.
  - Basierend auf den Reihen wird mit jedem Zug ein Gesamtwert des Spielfelds aktualisiert. Dieser wird für die künstliche Intelligenz verwendet.
- FIARNetworkGame: Der Netzwerkmodus wurde nicht integriert, daher ist der Name etwas irreführend. Die Klasse erweitert das grundlegende Spiel und ist in der Lage zusätzliche Features der Anwendung, die über die Grundregeln des Spiels hinaus gehen zu verarbeiten. Hierzu gehören CPU Player und die Nutzung eines Timers.

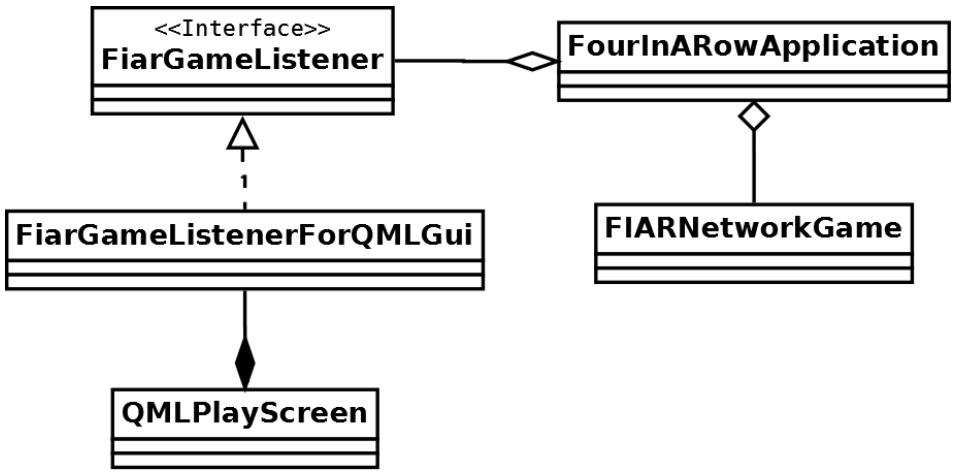
Datenaustauschklassen: FIARApplicationSetup, GameSetup, PlayerData

Diese Klassen sind eigentlich nur Datensätze, die jeweils alle Daten zur Einstellung der Anwendung, eines Spiels oder die Daten eines Spielers enthalten. Sie dienen dem Datenaustausch zwischen den Schichten. Auf Getter/Setter wurde verzichtet, die Manipulation passiert direkt.



FiarGameListener (Spiellogik) – FiarGameListenerForQMLGui (GUI)

Die Spiellogik muss in Reaktion auf Spielaktionen, die von der GUI ausgelöst werden Methoden auf der GUI aufrufen. Um weiterhin Unabhängigkeit zu gewährleisten wird eine Listener verwendet, den die GUI bei der FourInARowApplication Instanz registriert. Der FiarGameListener ist eine vollständig abstrakte Klasse. Eine Implementation wird von der GUI zur Verfügung gestellt.

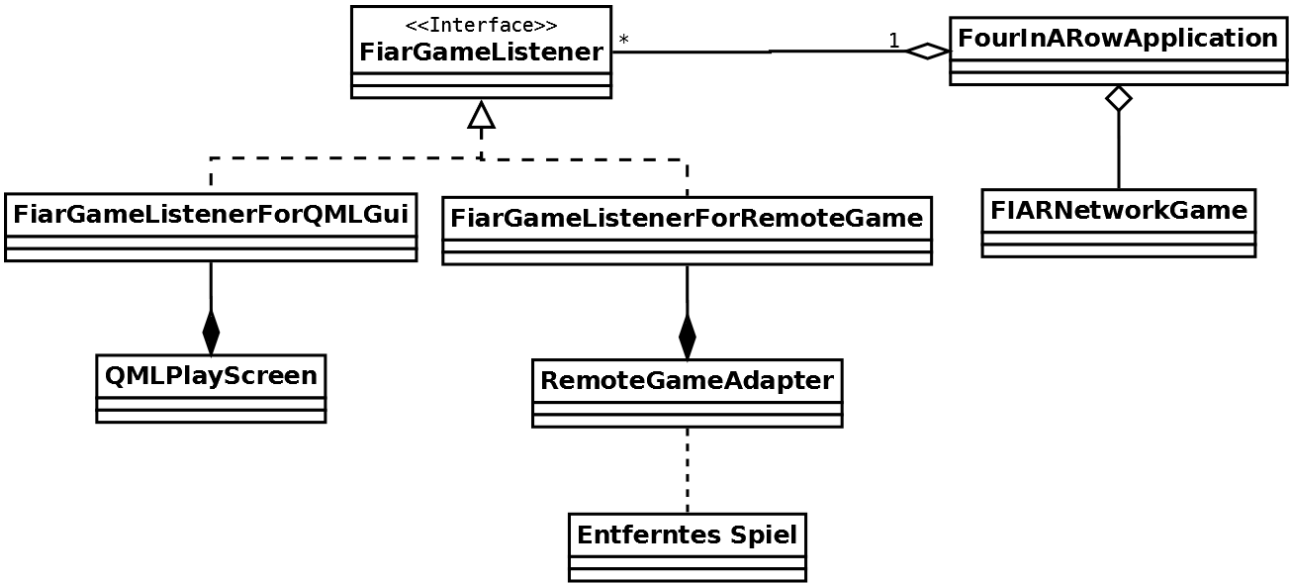


Das FiarNetworkgame kann also den bei der FourInARowApplication registrierten Listener aufrufen. Dieser ruft die entsprechenden Methoden auf dem QMLPlayScreen auf, der eine C++ Klasse der GUI ist.

GUI-Klassen
Alle Klassen die mit QML beginnen sind GUI-Klassen, die direkt auf ein QML Typ gemapped sind. Die QML Typen heißen genauso nur ohne den QML Präfix.
Das Mapping geschieht bei der Initialisierung des Programms (siehe FourInARowApplicaion).

6.3 Netzwerk Spiel

Das Netzwerksspiel ist nicht implementiert. Der Ansatz einer Schnittstelle ist über die Klasse FIARNetworkGame gegeben.
Der move(...) Methode wird ein PlayerTypeTag übergeben. Anhand dessen kann die Spiellogik unterscheiden, von wem der derzeitige Zug kommt und zum Beispiel Züge ignorieren, die von einem Spieler versucht werden, der nicht an der Reihe ist. Ein weiterer Ansatz, der weiterverfolgt werden kann ist der abstrakte FiraGameListener. Im Moment lässt sich nur ein Listener registrieren. Würde man zum Beispiel einen weiteren Listener für eine entfernte Anwendung registrieren, dann können die Methoden auch auf diesem aufgerufen werden und das Vorgehen bliebe transparent gegenüber der bereits bestehenden Anwendung. Die Abbildung skizziert dies:



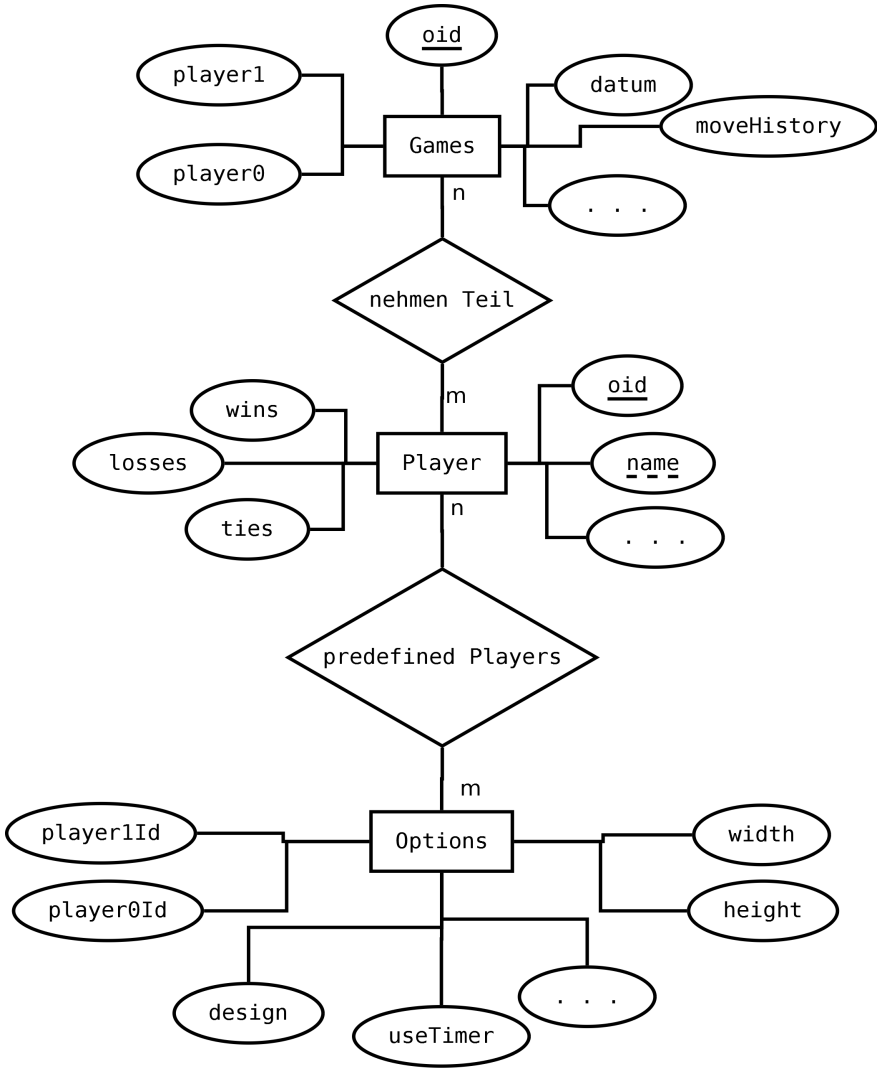
7. Künstliche Intelligenz (KI)

Die künstliche Intelligenz ist leider nur sehr rudimentär. Der verfolgte Ansatz hat nicht wirlich zufriedenstellende Ergebnisse geliefert. Daher nur kurz das Konzept.
Grundlegend ist in die Klasse Rowfield eine Bewertung des Spielfelds eingebaut. Das heißt, dass jedem Zustand des Spielfeldes ein Wert zugeordnet ist. Dieser basiert auf dem Rowfield, dass jeder Reihe, die nur von Steinen eines Spielers belegt ist einen positiven (Spieler1) oder negativen (Spieler 2) Wert zuordnet. Dabei steigt der Wert quadratisch zur Länge der Reihe. Beim suchen eines Zuges wird nun das Feld nach jedem möglichen Zug ausgewertet und der beste Wert gewählt. Die Werte der Reihen sind vorzeichenbehaftet, so dass je nach Spieler negative oder positive Werte besser sind. Dies entspricht dem MinMax Ansatz.
Um nun weiter „in die Zukunft“ zu schauen wird nicht nur der Wert der möglichen Züge ausgewertet, sondern es werden rekursiv die Werte weiterer Züge aufaddiert, wobei je Rekursionsschritt eine Gewichtung geschieht (in der Klasse treeDepthPenalty) genannt, ansonsten wären Züge, die mehrere Schritte in der Zukunft lägen genauso relevant, wie der direkt gewählte. Eventuell würde der Ansatz gut funktionieren, wenn bestimmte Grenzwerte und Erkennung eindeutiger Positionen mit implementiert würden. So kann es passieren, dass direkte Gewinnzüge nicht ausgeführt, bzw. direkte Gewinnzüge des Gegners nicht verhindert werden. Der Schwierigkeitsgrad wird über die Rekursionstiefe definiert. Leider ist subjektiv nicht festzustellen, inwiefern hier die Schwierigkeit erhöht wird. Die KI scheint zwar etwas vorausschauender zu agieren, aber ignoriert offensichtliche Situationen umsomehr.

8. Datenbank

Die Datenbank sichert die Spieler-, Spiel und Anwendungsdaten in einer SQL Lite Datenbank im Ausführungsverzeichnis (Datei: connectFourDatabase).

Zur Sicherung der Daten werden drei Tabellen benötigt. Auf zwei getrennte Tabellen für Spieler und Highscore wurde verzichtet. Stattdessen werden den Spielern die Attribute wins, losses und ties zugeordnet, wodurch eine SQL Abfrage für die Highscore ermöglicht wird. Die Anwendungseinstellungen werden in einer Tabelle mit nur einem Eintrag festgehalten (Options).



Die Beziehungen „predefine Players“ und „nehmen Teil“ sind zwar formal n zu m Beziehungen, können aber dadurch, dass immer nur zwei Spieler gesucht sind über temporäre Tabellen abgefragt werden.

Bsp.: Abfrage aller Spielstände um den Loadscreen mit Einträgen zu füllen

```
select gameOid, name0, name1, datum from
   (select oid, name as name0 from players) as players0,
   (select oid, name as name1 from players) as players1,
   (select oid as gameOid, player0, player1, datum from games) as gamesTmp
   where players0.oid=player0 and players1.oid=player1
   order by datum desc
```