Objektorientierte Programmierung mit C++ (WS 2010)

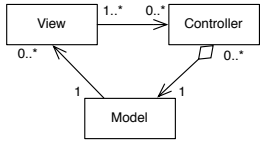
Dr. Andreas F. Borchert, Tobias Brosch
Institut für Angewandte Informationsverarbeitung
Universität Ulm

Blatt 11: Abgabetermin 19. Januar 2011 11 Uhr

Mandelbrotmenge Interaktiv (Teil 1)

Ziel dieses und der nächsten Übungsblätter ist es, die Mandelbrotmenge interaktiv darzustellen. Dabei werden wir unsere bereits kennengelernten Techniken vertiefen und nochmals üben.

Model-View-Controller-Pattern



Ein häufig verwendetes Design-Pattern, insbesondere im Zusammenhang mit GUIs (Graphical User Interfaces), ist das MVC-Pattern. Es strukturiert den Code in drei Komponenten:

1. Model: Repräsentiert die Daten.
2. View: Visualisiert das Modell.
3. Controller: Stellt Methoden zur Manipulation des Modells bereit.

Multiarray

Im Übungsblatt zur Mandelbrotmenge, hatten wir ein Verfahren gesehen, welches uns zu jedem Bildpunkt einen Wert zwischen 0 und 1 bestimmt. Das Ergebnis dieser Berechnung, wollen wir zunächst in einem zweidimensionalen Array abspeichern, welches Teil unseres Modells sein wird. Hierzu verwenden wir die `multi_array`-Klasse aus der `boost`-Bibliothek. Für die Zwecke dieses Beispiels, werden folgende Methoden ausreichend sein:

```
#include <boost/multi_array.hpp>
int main() {
```



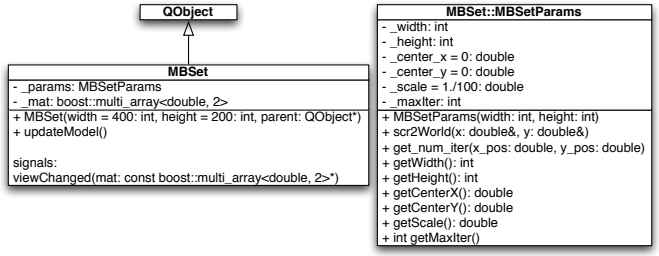
```
        // initialize 2-dimensional matrix of type double
        // and size 300 times 200
        boost::multi_array<double, 2> mat(
                boost::extents[300][200]);
        // can be resized
        mat.resize(boost::extents[100][50]);

        const std::size_t* dims = mat.shape();
                int width = dims[0];
                int height = dims[1];

        // access and assign elements
        for(int i = 0; i < width; ++i)
                for(int j = 0; j < height; ++j)
                        mat[i][j] = 0;
}
```

Model

Unser Model wird durch die Klasse `MBSet` dargestellt. Dabei kapseln wir die Parameter dieser Klasse in ihrer inneren Klasse MBSetParams.



Dabei implementiert `MBSet::MBSetParams::get_num_iter` die bekannte Mandelbrotiteration

```
double MBSet::get_num_iter(double x_pos, double y_pos) const {
  double x = 0;
  double y = 0;
  int maxIter = params.getMaxIter();
  for (int i = 0; i < maxIter; ++i) {
```

```
      double x_tmp = x * x - y * y + x_pos;
      y = 2 * x * y + y_pos;
      x = x_tmp;
      if (x * x + y * y > 32) {
        return std::pow((double) i / maxIter, 0.2);
      }
    }
    return 0;
}
```

und `MBSet::MBSetParams::scr2World` rechnet Bildschirm- bzw. Array-Koordinaten in Zahlen der komplexen Ebene um. Dabei dienen die Parameter `x` und `y` sowohl als Ein- als auch als Ausgabeparameter. Beim Aufruf beinhalten die Variablen die Arraykoordinaten, nach Beendigung, die Koordinaten in der Welt der komplexen Ebene:

```
void MBSet::MBSetParams::scr2World(double &x, double &y) const {
        x = (x - _width / 2) * _scale + _center_x;
        y = (_height / 2 - y) * _scale + _center_y;
}
```

Die Methode `updateModel` verwendet diese beiden Methoden, um unser Array `_mat` mit den entsprechenden Werten unserer Mandelbrotiteration zu füllen.

Die Methode `viewChanged(...)`, definiert ein Qt-Signal. Diese muss nicht implementiert werden und wird analog zu `private` oder `public`-Methoden mit dem Schlüsselwort `signals` gekennzeichnet. Damit der Meta-Object-Kompiler von Qt entsprechenden Code für den Signal-Slot-Mechanismus generiert, muss zusätzlich das Makro `Q_OBJECT` in die Klassendefinition mit aufgenommen werden:

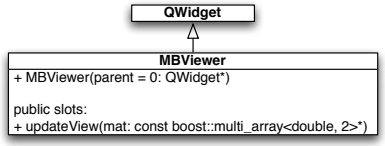
```
#include <QtGui>
class MBSet: public QObject {
Q_OBJECT
public:
  class MBSetParams {
    ...
  };
  public:
  MBSet(int width = 400, int height = 200, QObject* parent = 0);
  void updateModel();

signals:
  void viewChanged(const boost::multi_array<double, 2>* mat);
private:
  ...
};
```

Am Ende der Methode `updateModel` schickt Ihr das Signal mit `emit viewChanged(&_mat);` an alle verbundenen Slots.

Aufgabe: Implementiert die Klasse `MBSet` und ihre innere Klasse `MBSetParams` wie im UML-Diagramm angegeben in den Dateien `MBSet.h` und `MBSet.cpp`.

View



Um den Signal-Slot-Mechanismus zu ermöglichen, muss hier ebenfalls das Makro `Q_OBJECT` mit in die Klassendefinition mit aufgenommen werden:

```
#include <QtGui>
#include <boost/multi_array.hpp>

class MBViewer: public QLabel {
Q_OBJECT
public:
  MBViewer(QWidget *parent = 0);

public slots:
  void updateView(const boost::multi_array<double, 2>* mat);
};
```

Die Methode `updateView`, wird hierbei als `public slot` gekennzeichnet (kann aber wie jede `public` Methode auch „normal“ aufgerufen werden).

Der Hintergrund der Klasse `MBViewer` ist, dass ein Label in der Lage ist, eine `QPixmap` darzustellen. Da für den direkten Pixelzugriff die Klasse `QImage` jedoch besser geeignet ist, erstellen wir zunächst ein `QImage`, weisen die Pixelwerte zu, und setzen die `Pixmap` unserer von `QLabel` abgeleiteten Klasse auf eine `Pixmap` von unserem `QImage`:

```
// main part of updateView():

QImage image(width, height, QImage::Format_RGB32);
// translate double values to colors
for(int i = 0; i < width; ++i) {
  for(int j = 0; j < height; ++j) {
    image.setPixel(i, j, qRgb( 0, (*mat)[i][j] * 255, 0));
  }
}
setPixmap(QPixmap::fromImage(image));
adjustSize(); // makes sure, that out QLabel fits to its contents
```

Aufgabe: Implementiert die Klasse `MBViewer` wie im UML-Diagramm angegeben in den Dateien `MBViewer.h` und `MBViewer.cpp`.

Controller

Die Interaktion mit Unserem Modell, werden wir im nächsten Blatt betrachten. Jetzt wollen wir ersteinmal etwas auf unserem Bildschirm sehen.

Main

Nach so viel Text, bekommt Ihr zur Belohnung unser Hauptprogramm mit dazu. Wie Ihr seht, ist das „Verdrahten“ unserer Komponenten nun sehr einfach:

```
// main.cpp:
#include <QtGui>
#include "MBViewer.h" // View
#include "MBSet.h" // Model
int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  MBViewer mBViewer;
  MBSet mBSet(400, 200);
  // make view update on model change
  QObject::connect(
    &mBSet,
    SIGNAL(viewChanged(const boost::multi_array<double, 2>*)),
    &mBViewer,
    SLOT(updateView(const boost::multi_array<double, 2>*)));

  // generate initial content, which will emit the signal
  // viewChanged, so that our Slot updateView is called
  mBSet.updateModel();
  mBViewer.show();

  return a.exec();
}
```

Submission

Einreichen der Lösung mit:

```
submit cpp 11 main.cpp MBSet.h MBSet.cpp MBViewer.h MBViewer.cpp
```

Viel Spaß!