Test-Driven Development with Qt and KDE

Kevin Ottens

Introduction

Goals

- Discover the possibilities of the Qt and KDE frameworks
- Practice Test-Driven Development (TDD)

Method

- Putting in place the project infrastructure
- Developing the GUI part of the application
- Show the advantages of the model/view split available in Qt
- Apply TDD on the application model development

Outline

Outline

CMake and first source file

- Create a `main.cpp` file containing the `main()` function, only displaying a debug message;
- Create a file `CMakeLists.txt` for this project named *spreadsheet*;
- Create the build directory, and compile the project using CMake.

KApplication and first display

- Create a `KAboutData` and use it with `KCmdLineArgs::init()`
- Create a `KApplication` and an empty `QWidget` as main window

Toward a standard window

- Add a class `SpreadSheetMainWindow` which inherits from `KXmlGuiMainWindow`, use it as main window;
- Put in place the following actions: *N*ew, Open, Save, SaveAs, Quit, Cut, Copy, Paste
- Add one slot by action in SpreadSheetMainWindow (except for *Q*uit which will be connected to the `close()` slot of the window.

Hint : Use the `KActionCollection` and the method `setupGUI()` provided by `KXmlGuiMainWindow`.

Spreadsheet interface

- Add a `QTableWidget` as the window central widget (force and initial size of 100 by 100);
- Add the actions *C*ut, Copy and *P*aste in the contextual menu of `QTableWidget`;
- Add two `QLabel` to `SpreadSheetMainWindow` and position them in the status bar (they'll be used to indicate the coordinates of the current cell, and the formula contained).

Important in the following: In `SpreadSheetMainWindow`, the pointer to the `QTableWidget` will be of type `QTableView*`.

Outline

Implementing the base features

- Until now, we only put building blocks together
- No test written yet
- Here will develop the GUI based base features
- Our first unit test will be written nonetheless

Populate the table

Hardcode the following dataset for the table initialization:

William	Adama	1947-02-24
Saul	Tigh	1949-03-22
Lee	Adama	1973-04-03
Kara	Thrace	1980-04-08
Laura	Roslin	1952-04-28
Gaius	Baltar	1971-06-04

Location

- Write a test for a
  - `static QString locationFromIndex(int row, int column)`
  - which converts `(0, 0)` in `A1`, `(0, 1)` in `B1` and so on;
- Implement the method while increasing the amount of data for the test;
- Implement the necessary to update the status bar depending on the current cell, we'll use `locationFromIndex()`.

Manipulate data

- Implement *C*opy, Cut and *P*aste for one cell at a time;
- Extand it for contiguous cell zones (requires to force the behavior of selections in the view), pasting starts at the current cell.

Outline

MVC for our spreadsheet

- We'll now create our own data model for this spreadsheet.
- The main features will be implemented thanks to this model.
- Here, the TDD approach will be fully applied.

Model/view approach setup

- Implement a test for a new class `SpreadSheetModel` inheriting from `QAbtractTableModel`, during its creation such a model will contain only empty cells and have a size of 2^{16} columns by 2^{16} lines;
- Create a new class `SpreadSheetModel` which pass the previous test;
- Replace the `QTableWidget` of exercise 8 with a `QTableView`, give it an empty `SpreadSheetModel` during its creation.

Information: `rowCount()` and `columnCount()` must return 0 when the `QModelIndex` given as parameter is valid. For more information refers to the documentation of `QAbstractTableModel`.

Ensure lines and columns labelling

- Write the unit test for a
  - `static QString SpreadSheetModel::rowNameFromIndex(int)`
  - verify that we obtain `"1"` for 0, `"2"` for 1, `"10"` for 9, and so on;
- Write the method `QString rowNameFromIndex(int);`
- Write the unit test for a
  - `static QString SpreadSheetModel::columnNameFromIndex(int)`
  - verify that we obtain `"A"` for 0, `"B"` for 1, `"Z"` for 25, `"AA"` for 26, `"BA"` for 52, and so on;
- Write the method `QString columnNameFromIndex(int);`
- Overload `headerData()` in `SpreadSheeModel` to use the static methods freshly implemented... hey! the tests first;
- Modify the method `locationFromIndex()` of `SpreadSheetMainWindow` to eliminate the code duplication.

Information: Latest modification done without modifying any test, you made your first refactoring avoiding regressions.

Now, the questions will only be the unit tests that the model must pass.

Store text

```
void SpreadSheetTest::testThatTextIsStored()
{
    SpreadSheetModel m;
    QModelIndex index = m.index(21, 0);

    m.setData(index, "A string");
    QCOMPARE(m.data(index).toString(),
             QString("A string"));

    m.setData(index, "A different string");
    QCOMPARE(m.data(index).toString(),
             QString("A different string"));

    m.setData(index, "");
    QCOMPARE(m.data(index).toString(), QString(""));
}
```

Verify that many cells exist

```
void SpreadSheetTest::testThatManyCellsExist()
{
    SpreadSheetModel m;
    QModelIndex a = m.index(0, 0);
    QModelIndex b = m.index(23, 26);
    QModelIndex c = m.index(699, 900);

    m.setData(a, "One");
    m.setData(b, "Two");
    m.setData(c, "Three");

    QCOMPARE(m.data(a).toString(), QString("One"));
    QCOMPARE(m.data(b).toString(), QString("Two"));
    QCOMPARE(m.data(c).toString(), QString("Three"));
//=>
```

Verify that many cells exist

```
//=>
    m.setData(a, "Four");

    QCOMPARE(m.data(a).toString(), QString("Four"));
    QCOMPARE(m.data(b).toString(), QString("Two"));
    QCOMPARE(m.data(c).toString(), QString("Three"));
}
```

Store numeric data

```
void SpreadSheetTest::testNumericCells()
{
    SpreadSheetModel m;
    QModelIndex index = m.index(0, 20);

    m.setData(index, "X99"); // String
    QCOMPARE(m.data(index).toString(), QString("X99"));

    m.setData(index, "14"); // Number
    QCOMPARE(m.data(index).toString(), QString("14"));
    QCOMPARE(m.data(index).toInt(), 14);
//=>
```

Store numeric data

```
//=>
    // Whole string must be numeric
    m.setData(index, "99 X");
    QCOMPARE(m.data(index).toString(), QString("99 X"));
    bool ok;
    m.data(index).toInt(&ok);
    QVERIFY(!ok);

    m.setData(index, " 1234 "); // Blanks ignored
    QCOMPARE(m.data(index).toString(), QString("1234"));
    QCOMPARE(m.data(index).toInt(), 1234);

    m.setData(index, " "); // Just a blank
    QCOMPARE(m.data(index).toString(), QString(" "));
}
```

Hint : Add the asserts one by one.

Access to the original data for editing

```
void SpreadSheetTest::testAccessToLiteralForEditing()
{
    SpreadSheetModel m;
    QModelIndex index = m.index(0, 20);

    m.setData(index, "Some string");
    QCOMPARE(m.data(index, Qt::EditRole).toString(),
             QString("Some string"));

    m.setData(index, " 1234 ");
    QCOMPARE(m.data(index, Qt::EditRole).toString(),
             QString(" 1234 "));

    m.setData(index, "=7");
    QCOMPARE(m.data(index, Qt::EditRole).toString(),
             QString("=7"));
}
```

Outline

Add the support for formulas

- It is wise to add asserts of each proposed test one by one
- Or to add more tests...
- It depends on your confidence level concerning formulas processing

Formulas support basics

```
SpreadSheetTest::testFormulaBasics()
{
    SpreadSheetModel m;
    QModelIndex index = m.index(2, 10);

    m.setData(index, " =7"); // note leading space
    QCOMPARE(m.data(index).toString(), QString("=7"));
    QCOMPARE(m.data(index, Qt::EditRole).toString(),
             QString(" =7"));

    m.setData(index, "=7"); // constant formula
    QCOMPARE(m.data(index).toString(), QString("7"));
    QCOMPARE(m.data(index, Qt::EditRole).toString(),
             QString("=7"));
    // Adding intermediate tests here to guide you is fine
    // Go ahead!
//=>
```

Formulas support basics

```
//=>
    m.setData(index, "=(7)"); // parenthesis
    QCOMPARE(m.data(index).toString(), QString("7"));

    m.setData(index, "=(((10)))"); // more parenthesis
    QCOMPARE(m.data(index).toString(), QString("10"));

    m.setData(index, "=2*3*4"); // multiply
    QCOMPARE(m.data(index).toString(), QString("24"));

    m.setData(index, "=12+3+4"); // add
    QCOMPARE(m.data(index).toString(), QString("19"));

    m.setData(index, "=4+3*2"); // precedence
    QCOMPARE(m.data(index).toString(), QString("10"));
//=>
```

Formulas support basics

```
//=>

    m.setData(index, "=5*(4+3)*(((2+1)))"); // full expression
    QCOMPARE(m.data(index).toString(), QString("105"));
}
```

Error management

```
void SpreadSheetTest::testFormulaErrors()
{
    SpreadSheetModel m;
    QModelIndex index = m.index(0, 0);

    m.setData(index, "=5*");
    QCOMPARE(m.data(index).toString(), QString("#Error"));

    m.setData(index, "=((((5))");
    QCOMPARE(m.data(index).toString(), QString("#Error"));
}
```

Minus, divide, negate

The client notices that he forgot to ask for the support of the minus, divide and negate operators in formulas. You now have to add them:

- Add tests for minus and then implement
- Add tests for divide and then implement
- Add tests for negate and then implement

Outline

Verify that a reference works

```
void SpreadSheetTest::testThatReferenceWorks()
{
    SpreadSheetModel m;
    QModelIndex a = m.index(0, 0);
    QModelIndex b = m.index(10, 20);

    m.setData(a, "8");
    m.setData(b, "=A1");

    QCOMPARE(m.data(b).toString(), QString("8"));
}
```

Verify that changes propagate

```
void SpreadSheetTest::testThatChangesPropagate()
{
    SpreadSheetModel m;
    QModelIndex a = m.index(0, 0);
    QModelIndex b = m.index(10, 20);

    m.setData(a, "8");
    m.setData(b, "=A1");

    QCOMPARE(m.data(b).toString(), QString("8"));

    m.setData(a, "9");
    QCOMPARE(m.data(b).toString(), QString("9"));
}
```

Verify that formulas are recalculated

```
void SpreadSheetTest::testThatFormulasRecalculate()
{
    SpreadSheetModel m;
    QModelIndex a = m.index(0, 0);
    QModelIndex b = m.index(1, 0);
    QModelIndex c = m.index(0, 1);

    m.setData(a, "8");
    m.setData(b, "3");
    m.setData(c, "=A1*(A1-A2)+A2/3");

    QCOMPARE(m.data(c).toString(), QString("41"));

    m.setData(b, "6");

    QCOMPARE(m.data(c).toString(), QString("18"));
}
```

Verify that changes propagate on several levels

```
void SpreadSheetTest::testThatDeepChangesPropagate()
{
    SpreadSheetModel m;
    QModelIndex a1 = m.index(0, 0);
    QModelIndex a2 = m.index(1, 0);
    QModelIndex a3 = m.index(2, 0);
    QModelIndex a4 = m.index(3, 0);

    m.setData(a1, "8");
    m.setData(a2, "=A1");
    m.setData(a3, "=A2");
    m.setData(a4, "=A3");
    QCOMPARE(m.data(a4).toString(), QString("8"));

    m.setData(a2, "6");
    QCOMPARE(m.data(a4).toString(), QString("6"));
}
```

Verify the use of cells in several formulas

```
void SpreadSheetTest::testThatFormulasWorkWithManyCells()
{
    SpreadSheetModel m;
    QModelIndex a1 = m.index(0, 0);
    QModelIndex a2 = m.index(1, 0);
    QModelIndex a3 = m.index(2, 0);
    QModelIndex a4 = m.index(3, 0);
    QModelIndex b1 = m.index(0, 1);
    QModelIndex b2 = m.index(1, 1);
    QModelIndex b3 = m.index(2, 1);
    QModelIndex b4 = m.index(3, 1);
//=>
```

Verify the use of cells in several formulas

```
//=>
    m.setData(a1, "10");
    m.setData(a2, "=A1+B1");
    m.setData(a3, "=A2+B2");
    m.setData(a4, "=A3");
    m.setData(b1, "7");
    m.setData(b2, "=A2");
    m.setData(b3, "=A3-A2");
    m.setData(b4, "=A4+B3");

    QCOMPARE(m.data(a4).toString(), QString("34"));
    QCOMPARE(m.data(b4).toString(), QString("51"));
}
```

Verify that the circular references give an error

```
void SpreadSheetTest::testCircularReferences()
{
    SpreadSheetModel m;
    QModelIndex a1 = m.index(0, 0);
    QModelIndex a2 = m.index(1, 0);
    QModelIndex a3 = m.index(2, 0);

    m.setData(a1, "=A1");
    m.setData(a2, "=A1");
    m.setData(a3, "=A2");

    QCOMPARE(m.data(a1).toString(), QString("#Error"));

    m.setData(a1, "=A3");

    QCOMPARE(m.data(a1).toString(), QString("#Error"));
}
```

Outline

GUI cleanup

- If not done yet, remove the initialization of the table which was implemented previously
- If necessary, modify *Cut*, *Copy* and *Paste* so that they recopy formulas and not results;
- Add a test to ensure that the status bar displays the formula of the current cell, not its value;
- Add the code necessary to make the test pass.

Hint: Use the parent/child relationship of `QObject` and its name property.

To go further...

Some aspects are ignored in this training, and it would be necessary to see them to complete the application :

- saving, for this it'd be required to take a look at `QIODevice`, `KIO::NetAccess` and `KFileDialog`;
- some small display issues tied to the lack of events emitted by the model, then it'd be necessary to take care of the `QAbstractItemModel` signals like `dataChanged()`;
- function support in formulas;
- and probably more...

Outline

Conclusion

Facts

- We got a decent spreadsheet for the given effort
- The application is 433 lines of code, and only has two classes;
- Our tests are 368 lines of code, and they cover our model as well as a part of the GUI (we could have made more);
- We progressed by small modifications by adding a constraint at a time.

What have you learned?

- How to apply test-driven development and its advantages
- Now you can add complex features progressibely and avoiding regressions
- Better understand the features provided by Qt and KDE
- How to obtain a lot of results with a few lines of code

Questions?

Kevin Ottens
ervin@kde.org