



Jasmin Blanchette
Jasmin Blanchette

Mark Summerfield
Mark Summerfield

C++ GUI Programmierung mit Qt4

Die offizielle Einführung

2. Auflage



ADDISON-WESLEY



3 Hauptfenster erstellen

In diesem Kapitel lernen Sie, wie Sie mithilfe von Qt Hauptfenster erstellen. Am Ende des Kapitels werden Sie in der Lage sein, die gesamte Benutzeroberfläche einer Anwendung mit Menüs, Symbolleisten, Statusleisten und allen erforderlichen Dialogfeldern zu erstellen.

Das Hauptfenster einer Anwendung stellt das Framework bereit, auf dem die Benutzeroberfläche aufgebaut wird. Grundlage dieses Kapitels ist das Hauptfenster der in Abbildung 3.1 dargestellten Tabellenkalkulation *Spreadsheet*. Diese Anwendung verwendet die Dialogfelder *FIND*, *GO TO CELL* und *SORT*, die wir in Kapitel 2 erstellt haben.

Hinter den meisten GUI-Anwendungen steht ein Code-rumpf, der die zugrunde liegende Funktionalität bereitstellt, zum Beispiel Code zum Lesen und Schreiben von Dateien oder zur Verarbeitung der in der Benutzeroberfläche präsentierten Daten. In Kapitel 4 sehen wir, wie eine solche Funktionalität implementiert wird, wobei wir wieder die Tabellenkalkulation als Beispiel verwenden.

3.1 Subklassen von `QMainWindow`

Das Hauptfenster einer Anwendung wird erstellt, indem wir Subklassen von `QMainWindow` anlegen. Viele der in Kapitel 2 erörterten Verfahren zum Erstellen von Dialogfeldern sind auch für Hauptfenster relevant, da sowohl `QDialog` als auch `QMainWindow` von `QWidget` erben.



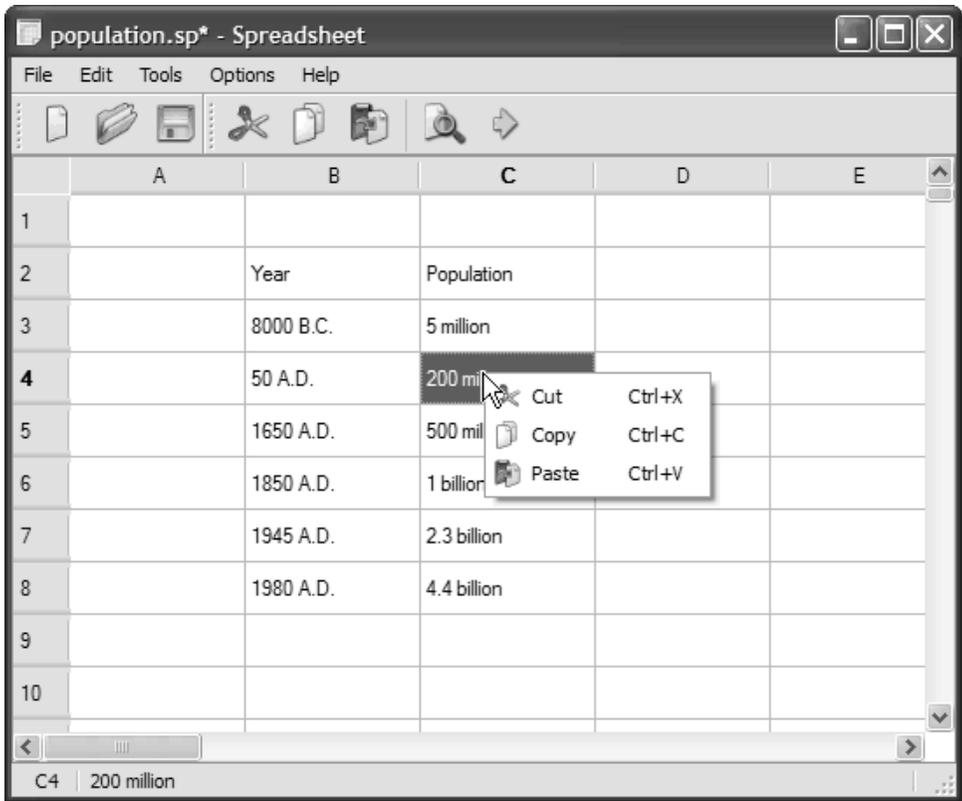


Abbildung 3.1: Die Anwendung Spreadsheet

Zwar lassen sich Hauptfenster mithilfe von *Qt Designer* erstellen, in diesem Kapitel werden wir jedoch alles per Code erledigen, um die Vorgehensweise zu demonstrieren. Wenn Sie den visuellen Ansatz bevorzugen, schauen Sie sich das Kapitel »*Creating Main Windows in Qt Designer*« in der Onlineanleitung zu *Qt Designer* an.

Der Quellcode für das Hauptfenster der Tabellenkalkulation verteilt sich auf die Dateien *mainwindow.h* und *mainwindow.cpp*. Beginnen wir mit der Headerdatei:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;

```

```

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void closeEvent(QCloseEvent *event);

```

Wir definieren die Klasse `MainWindow` als Subklasse von `QMainWindow`. Sie enthält das Makro `Q_OBJECT`, da sie ihre eigenen Signale und Slots bereitstellt.

`closeEvent()` ist eine virtuelle Funktion in `QWidget`, die automatisch aufgerufen wird, wenn der Benutzer das Fenster schließt. Sie wird in `MainWindow` reimplementiert, sodass wir dem Benutzer die Standardfrage stellen können, ob er seine Änderungen und die Benutzereinstellungen auf dem Datenträger speichern möchte.

```

private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void find();
    void goToCell();
    void sort();
    void about();

```

Einige Menüoptionen wie `FILE|NEW` und `HELP|ABOUT` werden als private Slots in `MainWindow` implementiert. Der Typ des Rückgabewerts der meisten Slots ist `void`, `save()` und `saveAs()` geben jedoch einen `bool`-Wert zurück. Der Rückgabewert wird ignoriert, wenn ein Slot als Reaktion auf ein Signal ausgeführt wird. Wenn wir jedoch einen Slot als Funktion aufrufen, steht uns der Rückgabewert genau wie bei einer normalen C++-Funktion zur Verfügung.

```

    void openRecentFile();
    void updateStatusBar();
    void spreadsheetModified();

private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool okToContinue();
    bool loadFile(const QString &fileName);

```

```
bool saveFile(const QString &fileName);
void setCurrentFile(const QString &fileName);
void updateRecentFileActions();
QString strippedName(const QString &fullName);
```

Das Hauptfenster benötigt weitere private Slots und mehrere private Funktionen zur Unterstützung der Benutzeroberfläche.

```
Spreadsheet *spreadsheet;
FindDialog *findDialog;
QLabel *locationLabel;
QLabel *formulaLabel;
QStringList recentFiles;
QString curFile;

enum { MaxRecentFiles = 5 };
QAction *recentFileActions[MaxRecentFiles];
QAction *separatorAction;

QMenu *fileMenu;
QMenu *editMenu;
...
QToolBar *fileToolBar;
QToolBar *editToolBar;
QAction *newAction;
QAction *openAction;
...
QAction *aboutQtAction;
};

#endif
```

Neben den privaten Slots und Funktionen verfügt `MainWindow` über viele private Variablen. Diese werden jeweils erläutert, wenn sie zum Einsatz kommen.

Sehen Sie sich nun die Implementierung an:

```
#include <QtGui>

#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
```

Die Headerdatei `<QtGui>` enthält die Definition aller in unserer Subklasse verwendeten Qt-Klassen. Darüber hinaus fügen wir einige benutzerdefinierte Headerdateien hinzu, unter anderem `finddialog.h`, `gotocelldialog.h` und `sortdialog.h` aus Kapitel 2.

```
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);

    createActions();
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();

    readSettings();

    findDialog = 0;

    setWindowIcon(QIcon(":/images/icon.png"));
    setCurrentFile("");
}
```

Der Konstruktor erstellt zunächst das Widget `Spreadsheet` und legt es als zentrales Widget des Hauptfensters fest. Das zentrale Widget nimmt die Mitte des Hauptfensters ein (siehe Abbildung 3.2). Die Klasse `Spreadsheet` ist eine Subklasse von `QTableWidget` mit einigen Funktionen für die Tabellenkalkulation, zum Beispiel der Unterstützung von Formeln. Wir werden sie in Kapitel 4 implementieren.

Anschließend richten die privaten Funktionen `createActions()`, `createMenus()`, `createContextMenu()`, `createToolBars()` und `createStatusBar()` den Rest des Hauptfensters ein. Außerdem rufen wir die private Funktion `readSettings()` auf, um die gespeicherten Einstellungen der Anwendung zu lesen.

Der `findDialog`-Zeiger wird als Null-Zeiger initialisiert. Beim ersten Aufruf von `MainWindow::find()` erstellen wir das Objekt `FindDialog`.

Am Ende des Konstruktors setzen wir den Wert des Fenstersymbols auf `icon.png`. Qt unterstützt viele Bildformate, einschließlich BMP, GIF, JPEG, PNG, PNM, SVG, TIFF, XBM und XPM. Der Aufruf von `QWidget::setWindowIcon()` legt das Symbol fest, das in der oberen linken Ecke des Fensters angezeigt wird. Leider gibt es keine plattformunabhängige Möglichkeit, das Anwendungssymbol festzulegen, das auf dem Desktop erscheint. Plattformspezifische Verfahren werden unter <http://doc.trolltech.com/4.3/appicon.html> erläutert.

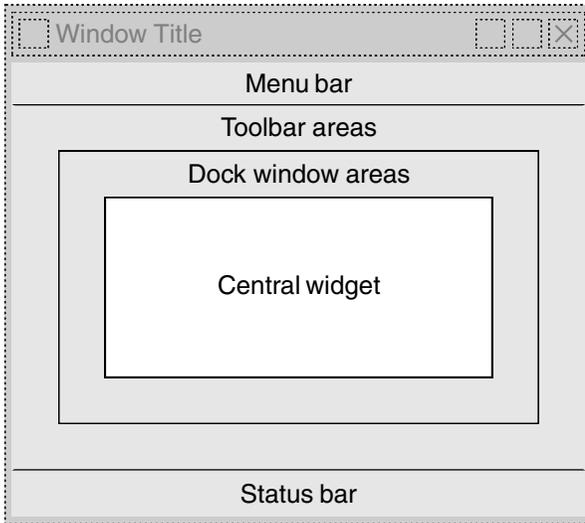


Abbildung 3.2: Die Bereiche von QMainWindow

GUI-Anwendungen verwenden im Allgemeinen viele Bilder. Es gibt mehrere Methoden, der Anwendung Bilder bereitzustellen. Folgende sind am gebräuchlichsten:

- ▶ Die Bilder in Dateien speichern und zur Laufzeit laden.
- ▶ XPM-Dateien in den Quellcode einbinden. (Das funktioniert, weil es sich bei XPM-Dateien auch um gültige C++-Dateien handelt.)
- ▶ Den Ressourcenmechanismus von Qt nutzen.

Hier verwenden wir den Ressourcenmechanismus von Qt, da er bequemer ist als das Laden der Dateien zur Laufzeit und bei allen unterstützten Bilddateiformaten funktioniert. Wir haben uns entschieden, die Bilder innerhalb der Quellstruktur in einem Unterverzeichnis mit dem Namen *images* zu speichern.

Um das Ressourcensystem von Qt verwenden zu können, müssen wir eine Ressourcendatei erstellen und in die *.pro*-Datei eine Zeile einfügen, die die Ressourcendatei kennzeichnet. In diesem Beispiel haben wir die Ressourcendatei *spreadsheet.qrc* genannt, sodass wir die folgende Zeile in die *.pro*-Datei einfügen:

```
RESOURCES    = spreadsheet.qrc
```

Die Ressourcendatei selbst verwendet ein einfaches XML-Format. Der folgende Code zeigt einen Auszug aus der von uns eingesetzten Datei:

```
<RCC>
<qresource>
  <file>images/icon.png</file>
```

```

...
<file>images/gotocell.png</file>
</qresource>
</RCC>

```

Die Ressourcendateien werden in die ausführbare Datei der Anwendung kompiliert, sodass sie nicht verloren gehen können. Wenn wir auf Ressourcen verweisen, verwenden wir das Pfadpräfix `:/` (Doppelpunkt gefolgt von einem Schrägstrich). Das Symbol wird demnach in Form von `:/images/icon.png` angegeben. Bei Ressourcen kann es sich um alle Arten von Dateien (nicht nur um Bilder) handeln und sie lassen sich fast überall dort einsetzen, wo Qt einen Dateinamen erwartet. Kapitel 12 befasst sich ausführlicher mit diesem Thema.

3.2 Menüs und Symbolleisten erstellen

Die meisten modernen GUI-Anwendungen stellen Menüs, Kontextmenüs und Symbolleisten bereit. Die Menüs ermöglichen den Benutzern, die Anwendung kennen zu lernen und herauszufinden, wie neue Aufgaben ausgeführt werden, während die Kontextmenüs und Symbolleisten einen schnellen Zugriff auf häufig benutzte Funktionen bieten. Abbildung 3.3 zeigt die Menüs der Anwendung *Spreadsheet*.

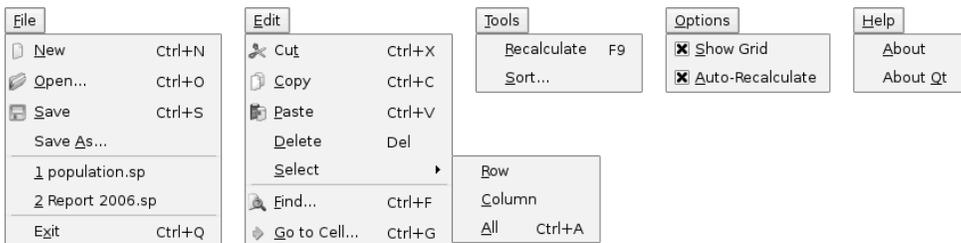


Abbildung 3.3: Die Menüs der Anwendung *Spreadsheet*

Durch sein Aktionskonzept vereinfacht Qt die Programmierung von Menüs und Symbolleisten. Bei einer *Aktion* handelt es sich um ein Element, das einer beliebigen Anzahl von Menüs und Symbolleisten hinzugefügt werden kann. Das Erstellen von Menüs und Symbolleisten umfasst in Qt die folgenden Schritte:

- ▶ Die Aktionen erstellen und einrichten
- ▶ Menüs erstellen und sie mit den Aktionen füllen
- ▶ Symbolleisten erstellen und sie mit den Aktionen füllen

In der Tabellenkalkulation werden die Aktionen in `createActions()` erstellt:

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(QKeySequence::New);
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
}
```

Die Aktion `New` verfügt über einen Menüeintrag mit Zugriffstaste (`NEW`), ein übergeordnetes Element (das Hauptfenster), ein Symbol (`new.png`), eine Tastenkombination und einen Statusleistentipp. In den meisten Systemen sind bestimmten Aktionen standardisierte Tastenkombinationen zugeordnet. Zum Beispiel ist der Aktion `NEW` unter `Windows`, `KDE` und `GNOME` die Tastenkombination `[Strg]+[N]` und auf `Mac OS X` die Tastenkombination `[MacBef]+[N]` zugeordnet. Mit dem entsprechenden Wert der Enumeration `QKeySequence::StandardKey` lässt sich gewährleisten, dass `Qt` die korrekten Tastenkombinationen für die Plattform, auf der die Anwendung läuft, bereitstellt.

Wir verbinden das Signal `triggered()` der Aktion mit dem privaten Slot `newFile()` des Hauptfensters, den wir im nächsten Abschnitt implementieren werden. Diese Verbindung gewährleistet, dass der Slot `newFile()` aufgerufen wird, wenn der Benutzer den Menüeintrag `FILE|NEW` auswählt, auf die Schaltfläche `NEW` in der Symbolleiste klickt oder die Tastenkombination `[Strg]+[N]` drückt.

Die Aktionen `OPEN`, `SAVE` und `SAVE AS` sind der Aktion `NEW` sehr ähnlich, sodass wir gleich zum Abschnitt mit der Liste der zuletzt geöffneten Dateien im Menü `FILE` übergehen:

```
...
for (int i = 0; i < MaxRecentFiles; ++i) {
    recentFileActions[i] = new QAction(this);
    recentFileActions[i]->setVisible(false);
    connect(recentFileActions[i], SIGNAL(triggered()),
            this, SLOT(openRecentFile()));
}
```

Wir füllen das Array `recentFileActions` mit Aktionen. Jede Aktion wird ausgeblendet und mit dem Slot `openRecentFile()` verbunden. Später werden wir sehen, wie die Aktionen für die zuletzt bearbeiteten Dateien sichtbar gemacht und verwendet werden.

```
exitAction = new QAction(tr("E&xit"), this);
exitAction->setShortcut(tr("Ctrl+Q"));
exitAction->setStatusTip(tr("Exit the application"));
connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```

Die Aktion `EXIT` unterscheidet sich etwas von den anderen bisher gezeigten Aktionen. Es gibt keine standardisierte Tastenkombination für das Beenden einer Anwendung,

sodass wir hier die Tastenkombination explizit festlegen. Außerdem verbinden wir die Aktion mit dem von Qt bereitgestellten Slot `close()` des Fensters.

Nun können wir zur Aktion **SELECT ALL** übergehen:

```
...
selectAllAction = new QAction(tr("&All"), this);
selectAllAction->setShortcut(QKeySequence::SelectAll);
selectAllAction->setStatusTip(tr("Select all the cells in the "
                                "spreadsheet"));
connect(selectAllAction, SIGNAL(triggered()),
        spreadsheet, SLOT(selectAll()));
```

Der Slot `selectAll()` wird von einem der Vorfahren von `QTableWidget` bereitgestellt, und zwar von `QAbstractItemView`, sodass wir ihn nicht selbst zu implementieren brauchen.

Gehen wir nun zur Aktion **SHOW GRID** im Menü **OPTIONS** weiter:

```
...
showGridAction = new QAction(tr("&Show Grid"), this);
showGridAction->setCheckable(true);
showGridAction->setChecked(spreadsheet->showGrid());
showGridAction->setStatusTip(tr("Show or hide the spreadsheet's "
                                "grid"));
connect(showGridAction, SIGNAL(toggled(bool)),
        spreadsheet, SLOT(setShowGrid(bool)));
```

Bei **SHOW GRID** handelt es sich um eine aktivierbare Aktion. Sie wird im Menü mit einem Häkchen versehen und in der Symbolleiste als Umschalter dargestellt. Wenn die Aktion aktiviert wird, zeigt die Spreadsheet-Komponente ein Raster an. Wir initialisieren die Aktion mit der Standardeinstellung für die Spreadsheet-Komponente, sodass sie beim Start synchronisiert wird. Dann verbinden wir das Signal `toggled(bool)` der Aktion **SHOW GRID** mit dem Slot `setShowGrid(bool)` der Spreadsheet-Komponente, den sie von `QTableWidget` erbt. Nachdem diese Aktion einem Menü oder einer Symbolleiste hinzugefügt wurde, kann der Benutzer das Raster ein- und ausschalten.

Die Aktionen **SHOW GRID** und **AUTO-RECALCULATE** sind unabhängig voneinander aktivierbare Aktionen. Durch die Klasse `QActionGroup` unterstützt Qt auch sich gegenseitig ausschließende Aktionen.

```
...
aboutQtAction = new QAction(tr("About &Qt"), this);
aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

Für die Aktion ABOUT QT verwenden wir den Slot `aboutQt()` des Objekts `QApplication`, der über die globale Variable `qApp` zugänglich ist. Damit wird das in Abbildung 3.4 gezeigte Dialogfeld geöffnet.

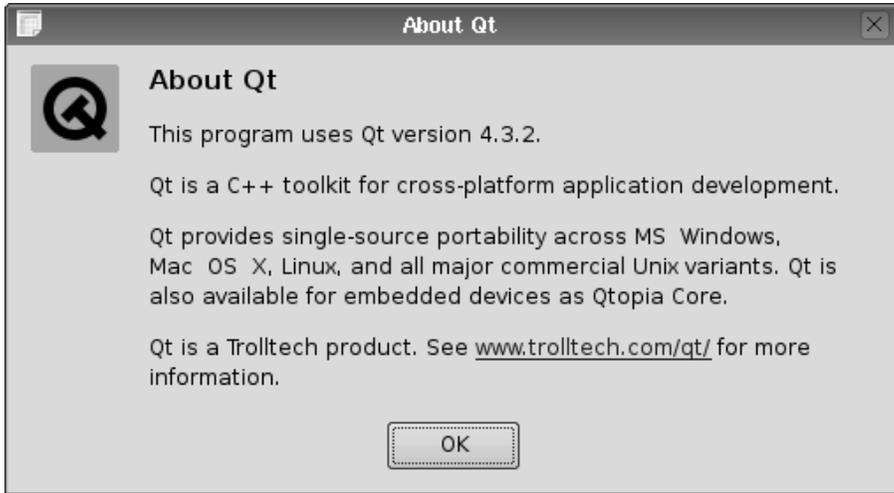


Abbildung 3.4: Das Dialogfeld ABOUT QT

Nachdem wir die Aktionen erstellt haben, können wir zum Aufbau eines Menüsystems übergehen, das diese Aktionen enthält:

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
}
```

In Qt handelt es sich bei Menüs um Instanzen von `QMenu`. Die Funktion `addMenu()` erstellt ein `QMenu-Widget` mit festgelegtem Text und fügt es in die Menüleiste ein. Die Funktion `QMainWindow::menuBar()` gibt einen Zeiger auf `QMenuBar` zurück. Die Menüleiste wird beim ersten Aufruf von `menuBar()` erstellt.

Wir beginnen mit der Erstellung des Menüs FILE und fügen diesem die Aktionen NEW, OPEN, SAVE und SAVE AS hinzu. Um eng miteinander verbundene Elemente visuell zu

gruppieren, fügen wir ein Trennzeichen ein. Wir verwenden eine `for`-Schleife, um die (anfängs verborgenen) Aktionen aus dem Array `recentFileActions` einzubauen, und fügen am Ende die Aktion `exitAction` hinzu.

Auf eines der Trennzeichen wird ein Zeiger gespeichert. Damit ist es möglich, das Trennzeichen zu verbergen (wenn es keine zuletzt verwendeten Dateien gibt) oder einzublenden, da wir nicht zwei Trennzeichen anzeigen wollen, zwischen denen sich keine Einträge befinden.

```
editMenu = menuBar()->addMenu(tr("&Edit"));
editMenu->addAction(cutAction);
editMenu->addAction(copyAction);
editMenu->addAction(pasteAction);
editMenu->addAction(deleteAction);

selectSubMenu = editMenu->addMenu(tr("&Select"));
selectSubMenu->addAction(selectRowAction);
selectSubMenu->addAction(selectColumnAction);
selectSubMenu->addAction(selectAllAction);

editMenu->addSeparator();
editMenu->addAction(findAction);
editMenu->addAction(goToCellAction);
```

Nun erstellen wir das Menü `EDIT`, indem wir mit `QMenu::addAction()` Aktionen hinzufügen, wie wir es bereits beim Menü `FILE` getan haben, und mit `QMenu::addMenu()` das Untermenü an der Stelle einfügen, an der es erscheinen soll. Wie beim Menü, zu dem es gehört, handelt es sich auch beim Untermenü um ein Element der Klasse `QMenu`.

```
toolsMenu = menuBar()->addMenu(tr("&Tools"));
toolsMenu->addAction(recalculateAction);
toolsMenu->addAction(sortAction);

optionsMenu = menuBar()->addMenu(tr("&Options"));
optionsMenu->addAction(showGridAction);
optionsMenu->addAction(autoRecalcAction);

menuBar()->addSeparator();

helpMenu = menuBar()->addMenu(tr("&Help"));
helpMenu->addAction(aboutAction);
helpMenu->addAction(aboutQtAction);
}
```

In ähnlicher Weise erstellen wir die Menüs `TOOLS`, `OPTIONS` und `HELP`. Zwischen den Menüs `OPTIONS` und `HELP` fügen wir ein Trennzeichen ein. Im Motif- und im CDE-Stil schiebt das Trennzeichen das Menü `HELP` nach rechts, in anderen Stilen wird es ignoriert. Abbildung 3.5 zeigt beide Fälle.

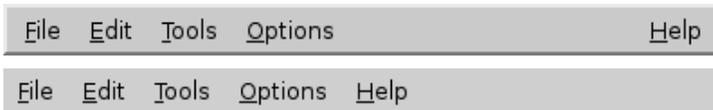


Abbildung 3.5: Die Menüleiste im Motif- und im Windows-Stil

```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

Jedes Qt-Widget kann über eine Liste der mit ihm verbundenen `QAction`-Elemente verfügen. Um ein Kontextmenü für die Anwendung bereitzustellen, fügen wir dem Spreadsheet-Widget die gewünschten Aktionen hinzu und legen die Kontextmenü-richtlinie des Widgets so fest, dass ein Kontextmenü mit diesen Aktionen angezeigt wird. Kontextmenüs werden durch Anklicken eines Widgets mit der rechten Maustaste oder durch Drücken einer plattformspezifischen Taste aufgerufen. Abbildung 3.6 zeigt das Kontextmenü der Anwendung *Spreadsheet*.

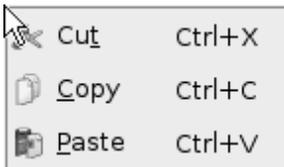


Abbildung 3.6: Das Kontextmenü der Anwendung Spreadsheet

Eine kompliziertere Möglichkeit, Kontextmenüs bereitzustellen, besteht darin, die Funktion `QWidget::contextMenuEvent()` zu reimplementieren, ein `QMenu`-Widget zu erstellen, es mit den gewünschten Aktionen zu füllen und die Funktion `exec()` dafür aufzurufen.

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);

    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
}
```

```

editToolBar->addSeparator();
editToolBar->addAction(findAction);
editToolBar->addAction(goToCellAction);
}

```

Das Erstellen von Symbolleisten weist große Ähnlichkeiten mit dem Erstellen von Menüs auf. Wir legen eine FILE- und eine EDIT-Symbolleiste an. Wie Abbildung 3.7 zeigt, kann eine Symbolleiste genau wie ein Menü über Trennzeichen verfügen.



Abbildung 3.7: Die Symbolleisten der Anwendung Spreadsheet

3.3 Die Statusleiste einrichten

Nachdem Sie die Menüs und Symbolleisten fertig gestellt haben, können Sie nun die Statusleiste der Tabellenkalkulation in Angriff nehmen. Im Normalzustand enthält die Statusleiste zwei Indikatoren: die Position und die Formel der aktuellen Zelle. Die Statusleiste wird auch zur Anzeige von Statusleistentipps und anderen temporären Meldungen verwendet. Abbildung 3.8 zeigt die Statusleiste in jedem Zustand.

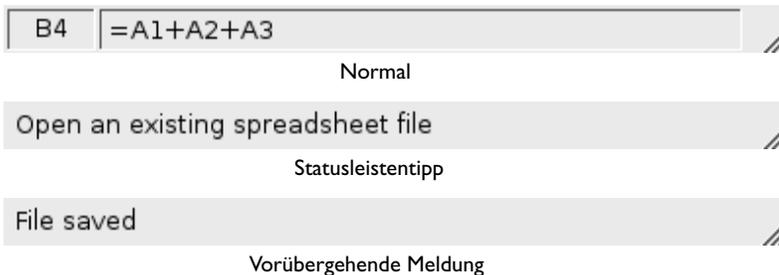


Abbildung 3.8: Die Statusleiste der Anwendung Spreadsheet

Der Konstruktor von `MainWindow` ruft `createStatusBar()` auf, um die Statusleiste einzurichten:

```

void MainWindow::createStatusBar()
{
    locationLabel = new QLabel(" W999 ");
    locationLabel->setAlignment(Qt::AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());

    formulaLabel = new QLabel;
    formulaLabel->setIndent(3);
}

```

```
statusBar()->addWidget(locationLabel);
statusBar()->addWidget(formulaLabel, 1);

connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
        this, SLOT(updateStatusBar()));
connect(spreadsheet, SIGNAL(modified()),
        this, SLOT(spreadsheetModified()));

updateStatusBar();
}
```

Die Funktion `QMainWindow::statusBar()` gibt einen Zeiger auf die Statusleiste zurück. (Die Statusleiste wird beim ersten Aufruf von `statusBar()` erstellt.) Die Statusindikatoren sind einfache `QLabel`-Elemente, deren Text wir nach Bedarf ändern. In `formulaLabel` haben wir einen Einzug eingefügt, sodass der darin angezeigte Text etwas vom linken Rand versetzt dargestellt wird. Beim Hinzufügen der `QLabel`-Elemente zur Statusleiste wird der Verwandtschaftsgrad neu festgelegt, sodass sie zu untergeordneten Elementen der Statusleiste werden.

Abbildung 3.8 zeigt, dass die beiden Label unterschiedliche Platzanforderungen haben. Der Zellenpositionsindikator braucht sehr wenig Platz, und wenn die Größe des Fensters neu festgelegt wird, sollte jeglicher zusätzliche Raum dem Zellenformelindikator auf der rechten Seite zugewiesen werden. Dies wird durch die Angabe des Dehnungsfaktors 1 im `QStatusBar::addWidget()`-Aufruf des Formel-Labels erreicht. Der Standarddehnungsfaktor 0 des Positionsindikators bedeutet, dass der Indikator vorzugsweise nicht gedehnt werden soll.

Wenn die Klasse `QStatusBar` Indikator-Widgets anordnet, versucht sie, jeweils die von `QWidget::sizeHint()` vorgegebene Idealgröße zu beachten, und dehnt dann alle dehnbaren Widgets, um den verfügbaren Platz auszufüllen. Die Idealgröße eines Widgets selbst hängt von seinem Inhalt ab und variiert, wenn wir diesen ändern. Um eine ständige Größenanpassung des Positionsindikators zu vermeiden, legen wir seine Mindestgröße so fest, dass er breit genug ist, um den längstmöglichen Text (»W999«) aufzunehmen, und fügen noch etwas zusätzlichen Platz hinzu. Außerdem setzen wir die Ausrichtung auf `Qt::AlignHCenter`, damit der Text horizontal zentriert wird.

Kurz vor dem Ende der Funktion verbinden wir zwei Signale von `Spreadsheet` mit den beiden `MainWindow`-Slots `updateStatusBar()` und `spreadsheetModified()`.

```
void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}
```

Der Slot `updateStatusBar()` aktualisiert den Zellenpositions- und den Zellenformelindikator. Er wird jedes Mal aufgerufen, wenn der Benutzer den Zellencursor in eine neue Zelle bewegt. Der Slot wird am Ende von `createStatusBar()` auch als normale Funktion verwendet, um die Indikatoren zu initialisieren. Das ist notwendig, da Spreadsheet beim Startvorgang kein `currentCellChanged()`-Signal ausgibt.

```
void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

Der Slot `spreadsheetModified()` setzt den Wert der Eigenschaft `windowModified` auf `true` und aktualisiert damit die Titelleiste. Außerdem aktualisiert die Funktion die Positions- und Formelindikatoren, sodass sie den aktuellen Stand der Dinge widerspiegeln.

3.4 Das Datei-Menü implementieren

In diesem Abschnitt implementieren wir die erforderlichen Slots und privaten Funktionen, um die Optionen des Menüs `FILE` funktionsfähig zu machen und die Liste der zuletzt geöffneten Dateien zu verwalten.

```
void MainWindow::newFile()
{
    if (okToContinue()) {
        spreadsheet->clear();
        setCurrentFile("");
    }
}
```

Der Slot `newFile()` wird aufgerufen, wenn der Benutzer auf die Menüoption `FILE | NEW` oder die Schaltfläche `NEW` in der Symbolleiste klickt. Die private Funktion `okToContinue()` fragt den Benutzer mit dem in Abbildung 3.9 gezeigten Dialogfeld, ob er seine Änderungen speichern möchte (»DO YOU WANT TO SAVE YOUR CHANGES?«), wenn ungespeicherte Änderungen vorliegen. Die Funktion gibt `true` zurück, wenn der Benutzer entweder `YES` oder `NO` ausgewählt hat (wobei das Dokument bei `YES` gespeichert wird), und `false`, wenn der Benutzer `CANCEL` gewählt hat. Die Funktion `Spreadsheet::clear()` löscht alle Zellen und Formeln der Tabellenkalkulation. Zusätzlich zur Festlegung der privaten Variablen `curFile` und zur Aktualisierung der Liste der zuletzt geöffneten Dateien aktualisiert die private Funktion `setCurrentFile()` den Titel des Fensters, um anzuzeigen, dass ein unbenanntes Dokument bearbeitet wird.

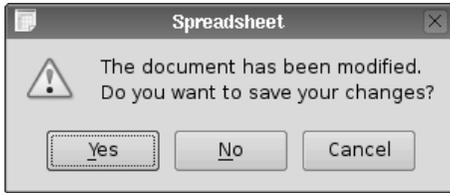


Abbildung 3.9: Abfrage zum Speichern bei geänderten Daten

```
bool MainWindow::okToContinue()
{
    if (isWindowModified()) {
        int r = QMessageBox::warning(this, tr("Spreadsheet"),
            tr("The document has been modified.\n"
                "Do you want to save your changes?"),
            QMessageBox::Yes | QMessageBox::Default,
            QMessageBox::No,
            QMessageBox::Cancel | QMessageBox::Escape);
        if (r == QMessageBox::Yes) {
            return save();
        } else if (r == QMessageBox::Cancel) {
            return false;
        }
    }
    return true;
}
```

In `okToContinue()` überprüfen wir den Status der Eigenschaft `windowModified`. Lautet der Wert `true`, zeigen wir das in Abbildung 3.9 dargestellte Meldungsfeld an. Es enthält die Schaltflächen YES, NO und CANCEL.

Die Klasse `QMessageBox` stellt viele Standardschaltflächen bereit und versucht automatisch, eine Schaltfläche als Standardschaltfläche (die aktiviert wird, wenn der Benutzer `Enter` drückt) und eine Schaltfläche zum Abbrechen (die aktiviert wird, wenn der Benutzer `Esc` drückt) einzurichten. Es ist auch möglich, bestimmte Schaltflächen als Standard- und Escape-Schaltflächen auszuwählen sowie die Texte auf den Schaltflächen festzulegen.

Der Aufruf von `warning()` mag auf den ersten Blick etwas abschreckend erscheinen, die allgemeine Syntax ist aber eigentlich unkompliziert:

```
QMessageBox::warning(übergeordnetes_Element, Titel, Meldung, Schaltfläche0,
    Schaltfläche1, ...);
```

Außer der Funktion `warning()` stellt `QMessageBox` auch die Funktionen `information()`, `question()` und `critical()` bereit, die jeweils über ein eigenes spezielles Symbol verfügen, wie sie in Abbildung 3.10 zu sehen sind.



Information



Frage



Warnung



Fehler

Abbildung 3.10: Symbole für Meldungsfelder im Windows-Stil

```
void MainWindow::open()
{
    if (okToContinue()) {
        QString fileName = QFileDialog::getOpenFileName(this,
            tr("Open Spreadsheet"), ".",
            tr("Spreadsheet files (*.sp)"));

        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}
```

Der Slot `open()` entspricht `FILE|OPEN`. Wie `newFile()` ruft er zuerst `okToContinue()` auf, um eventuelle ungespeicherte Änderungen zu verarbeiten. Dann verwendet er die statische Hilfsfunktion `QFileDialog::getOpenFileName()`, um vom Benutzer einen neuen Dateinamen anzufordern. Die Funktion öffnet ein Dateidialogfeld, lässt den Benutzer eine Datei auswählen und gibt den Dateinamen zurück – oder einen leeren String, falls der Benutzer auf `CANCEL` geklickt hat.

Das erste Argument von `QFileDialog::getOpenFileName()` ist das übergeordnete Widget. Die hierarchische Beziehung hat bei Dialogfeldern nicht die gleiche Bedeutung wie bei anderen Widgets. Ein Dialogfeld ist stets ein eigenständiges Fenster, wenn es jedoch über ein übergeordnetes Element verfügt, wird es standardmäßig über diesem Element zentriert. Ein untergeordnetes Dialogfeld nutzt auch den Taskleisteneintrag gemeinsam mit seinem übergeordneten Element.

Das zweite Argument ist der Titel des Dialogfelds. Das dritte Argument gibt das Startverzeichnis an – in unserem Fall das aktuelle Verzeichnis.

Das vierte Argument legt die Dateifilter fest. Ein Dateifilter besteht aus einem Beschreibungstext und einem Platzhaltermuster. Würden wir Dateien mit durch Kommas getrennten Werten und Lotus 1-2-3-Dateien zusätzlich zu dem eigenen Dateiformat der Tabellenkalkulation unterstützen, so hätten wir den folgenden Filter verwendet:

```
tr("Spreadsheet files (*.sp)\n"
    "Comma-separated values files (*.csv)\n"
    "Lotus 1-2-3 files (*.wk1 *.wks)")
```

Die private Funktion `loadFile()` wurde in `open()` aufgerufen, um die Datei zu laden. Wir machen daraus eine unabhängige Funktion, da wir dieselbe Funktionalität brauchen, um die zuletzt geöffneten Dateien zu laden:

```
bool MainWindow::loadFile(const QString &fileName)
{
    if (!spreadsheet->readFile(fileName)) {
        statusBar()->showMessage(tr("Loading canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
    return true;
}
```

Wir verwenden `Spreadsheet::readFile()`, um die Datei vom Datenträger zu lesen. Ist der Ladevorgang erfolgreich, rufen wir `setCurrentFile()` auf, um den Titel des Fensters zu aktualisieren; andernfalls hat `Spreadsheet::readFile()` den Benutzer mit einem Meldungsfeld bereits über das Problem informiert. Im Allgemeinen ist es sinnvoll, Fehlermeldungen von Komponenten der unteren Ebene ausgeben zu lassen, da sie die genauen Fehlerdetails angeben können.

In beiden Fällen zeigen wir zwei Sekunden (2.000 Millisekunden) lang eine Meldung in der Statusleiste an, um den Benutzer über die Arbeit der Anwendung auf dem Laufenden zu halten.

```
bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveFile(const QString &fileName)
{
    if (!spreadsheet->writeFile(fileName)) {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}
```

Der Slot `save()` entspricht `FILE|SAVE`. Wenn die Datei bereits einen Namen hat, da sie zuvor schon einmal geöffnet oder gespeichert wurde, wird `saveFile()` von `save()` mit diesem Namen aufgerufen; andernfalls wird einfach `saveAs()` aufgerufen.

```
bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                                                    tr("Save Spreadsheet"), ".",
                                                    tr("Spreadsheet files (*.sp)"));
    if (fileName.isEmpty())
        return false;

    return saveFile(fileName);
}
```

Der Slot `saveAs()` entspricht `FILE|SAVE AS`. Wir rufen `QFileDialog::getSaveFileName()` auf, um den Dateinamen vom Benutzer anzufordern. Wenn der Benutzer auf `CANCEL` klickt, geben wir den Wert `false` zurück, der bis zum Aufrufer (`save()` oder `okToContinue()`) weitergegeben wird.

Existiert die Datei bereits, verlangt die Funktion `getSaveFileName()` vom Benutzer eine Bestätigung, dass die Datei überschrieben werden soll. Dieses Verhalten lässt sich ändern, indem `QFileDialog::DontConfirmOverwrite` als zusätzliches Argument an `getSaveFileName()` übergeben wird.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

Wenn der Benutzer auf `FILE|EXIT` oder auf die Schaltfläche zum Schließen in der Titelleiste des Fensters klickt, wird der Slot `QWidget::close()` aufgerufen. Dieser sendet ein `close`-Ereignis an das Widget. Durch eine Reimplementierung von `QWidget::closeEvent()` können wir Versuche, das Hauptfenster zu schließen, abfangen und entscheiden, ob das Fenster tatsächlich geschlossen werden soll oder nicht.

Wenn nicht gespeicherte Änderungen vorhanden sind und der Benutzer `CANCEL` wählt, »ignorieren« wir das Ereignis und lassen das Fenster davon unberührt. Im Normalfall akzeptieren wir das Ereignis, woraufhin Qt das Fenster ausblendet. Außerdem rufen wir die private Funktion `writeSettings()` auf, um die aktuellen Einstellungen der Anwendung zu speichern.

Wenn das letzte Fenster geschlossen ist, wird die Anwendung beendet. Gegebenenfalls können wir dieses Verhalten deaktivieren, indem wir die Eigenschaft `quitOnLastWindowClosed` von `QApplication` auf den Wert `false` setzen, sodass die Anwendung weiterläuft, bis wir `QApplication::quit()` aufrufen.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);

    QString shownName = "Untitled";
    if (!curFile.isEmpty()) {
        shownName = strippedName(curFile);
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }

    setWindowTitle(tr("%1[*] - %2").arg(shownName)
                  .arg(tr("Spreadsheet")));
}

QString MainWindow::strippedName(const QString &fullName)
{
    return QFileInfo(fullFileName).fileName();
}
```

In `setCurrentFile()` legen wir die private Variable `curFile` fest, die den Namen der bearbeiteten Datei speichert. Bevor wir den Dateinamen in der Titelleiste anzeigen, entfernen wir den Dateipfad mithilfe von `strippedName()`, um das Ganze benutzerfreundlicher zu gestalten.

Jedes `QWidget`-Element verfügt über die Eigenschaft `windowModified`, deren Wert auf `true` gesetzt werden sollte, wenn das Dokument des Fensters nicht gespeicherte Änderungen aufweist; andernfalls sollte der Wert `false` lauten. Unter Mac OS X werden nicht gespeicherte Dokumente durch einen Punkt in der SCHLIEßEN-Schaltfläche der Titelleiste des Fensters gekennzeichnet, auf anderen Plattformen geschieht dies durch ein Sternchen hinter dem Dateinamen. Qt achtet automatisch auf dieses Verhalten, solange wir die Eigenschaft `windowModified` auf dem aktuellen Stand halten und die Markierung `[*]` an der Stelle im Titel des Fensters platzieren, an der bei Bedarf das Sternchen erscheinen soll.

An die Funktion `setWindowTitle()` haben wir folgenden Text übergeben:

```
(tr("%1[*] - %2").arg(shownName)
 .arg(tr("Spreadsheet")));
```

Die Funktion `QString::arg()` ersetzt den Parameter `%n` mit der niedrigsten Nummer durch ihr Argument und gibt den sich daraus ergebenden String zurück. In diesem Fall wird `arg()` mit zwei `%n`-Parametern verwendet. Der erste Aufruf von `arg()` ersetzt `%1` und der zweite `%2`. Wenn der Dateiname `budget.sp` lautet und keine Übersetzungsdatei geladen wird, lautet der resultierende String `budget.sp[*] - Spreadsheet`. Die Formulierung

```
setWindowTitle(shownName + tr("[*] - spreadsheet"));
```

wäre zwar einfacher gewesen, `arg()` bietet jedoch eine größere Flexibilität für Übersetzer.

Ist ein Dateiname vorhanden, aktualisieren wir `recentFiles`, d.h. die Liste der von der Anwendung zuletzt geöffneten Dateien. Wir rufen `removeAll()` auf, um alle Stellen, an denen der Dateiname vorkommt, aus der Liste zu entfernen und Duplikate zu verhindern. Dann rufen wir `prepend()` auf, um den Dateinamen als erstes Element einzufügen. Nach der Aktualisierung der Liste rufen wir die private Funktion `updateRecentFileActions()` auf, um die Einträge im Menü FILE zu aktualisieren.

```
void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);
    while (i.hasNext()) {
        if (!QFile::exists(i.next()))
            i.remove();
    }

    for (int j = 0; j < MaxRecentFiles; ++j) {
        if (j < recentFiles.count()) {
            QString text = tr("%&1 %2")
                .arg(j + 1)
                .arg(strippedName(recentFiles[j]));
            recentFileActions[j]->setText(text);
            recentFileActions[j]->setData(recentFiles[j]);
            recentFileActions[j]->setVisible(true);
        } else {
            recentFileActions[j]->setVisible(false);
        }
    }
    separatorAction->setVisible(!recentFiles.isEmpty());
}
```

Wir beginnen damit, mithilfe eines Iterators im Java-Stil alle nicht mehr vorhandenen Dateien zu entfernen. Einige Dateien wurden vielleicht in einer früheren Sitzung verwendet, sind aber danach gelöscht worden. Der Typ der Variablen `recentFiles` ist `QStringList` (Liste von `QString`-Elementen). Kapitel 11 erläutert Containerklassen wie `QStringList` ausführlich, wobei gezeigt wird, wie sie zur Standard-Template-Bibliothek von C++ (Standard Template Library, STL) in Beziehung stehen, und erklärt die Verwendung der Iteratorklassen von Qt im Java-Stil.

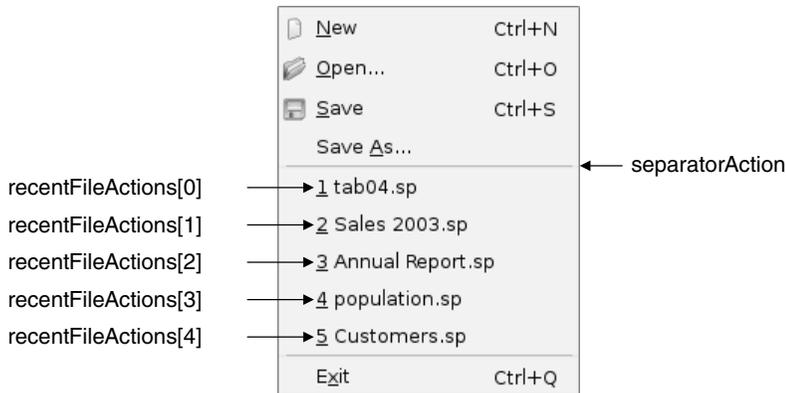


Abbildung 3.11: Das Menü FILE mit den zuletzt geöffneten Dateien

Dann gehen wir die Liste der Dateien erneut durch und verwenden diesmal eine arrayartige Indizierung. Für jede Datei erstellen wir einen String, der aus einem kaufmännischen Und-Zeichen, einer Ziffer ($j + 1$), einem Leerzeichen und dem Dateinamen (ohne Pfadangabe) besteht. Wir legen fest, dass die entsprechende Aktion diesen Text verwenden soll. Wäre die erste Datei beispielsweise *C:\Eigene Dateien\tab04.sp*, würde der Text der ersten Aktion `&1 tab04.sp` lauten. Abbildung 3.11 zeigt die Entsprechungen zwischen dem Array `recentFileActions` und dem resultierenden Menü.

Jede Aktion kann mit einem Datenelement vom Typ `QVariant` verbunden sein. Dieser Typ kann Werte vieler C++- und Qt-Typen speichern. Kapitel 11 geht näher darauf ein. Hier speichern wir den vollständigen Namen der Datei im `data`-Element der Aktion, sodass wir ihn später leicht abrufen können. Außerdem legen wir fest, dass die Aktion sichtbar sein soll.

Sind mehr Dateiaktionen als zuletzt geöffnete Dateien vorhanden, blenden wir diese zusätzlichen Aktionen einfach aus. Ist mindestens eine kürzlich geöffnete Datei vorhanden, legen wir abschließend fest, dass das Trennzeichen sichtbar sein soll.

```
void MainWindow::openRecentFile()
{
    if (okToContinue()) {
        QAction *action = qobject_cast<QAction *>(sender());
        if (action)
            loadFile(action->data().toString());
    }
}
```

Wenn der Benutzer eine kürzlich geöffnete Datei auswählt, wird der Slot `openRecentFile()` aufgerufen. Die Funktion `okToContinue()` wird verwendet, falls nicht gespeicherte Änderungen vorliegen. Sofern der Benutzer die Aktion nicht abbricht, finden wir mithilfe von `QObject::sender()` heraus, welche Aktion den Slot aufgerufen hat.

Anhand der von `moc`, dem Metaobjekt-Compiler von Qt, generierten Metainformationen führt die Funktion `qobject_cast<T>()` eine dynamische Typumwandlung durch. Sie gibt einen Zeiger auf die angeforderte `QObject`-Subklasse oder `0` zurück, wenn das Objekt nicht in diesen Typ umgewandelt werden kann. Im Gegensatz zu `dynamic_cast<T>()` in Standard-C++ funktioniert `qobject_cast<T>()` von Qt über die Grenzen dynamischer Bibliotheken hinweg. In unserem Beispiel verwenden wir `qobject_cast<T>()`, um einen `QObject`-Zeiger in einen `QObject`-Zeiger umzuwandeln. Verläuft die Typumwandlung erfolgreich (was der Fall sein sollte), rufen wir `loadFile()` mit dem vollständigen Dateinamen auf, den wir aus den Aktionsdaten extrahieren.

Da wir wissen, dass der Sender eine `QObject` ist, würde das Programm auch dann noch funktionieren, wenn wir stattdessen `static_cast<T>()` oder eine traditionelle Typumwandlung im C-Stil verwendet hätten. Eine Übersicht über die verschiedenen C++-Typumwandlungen finden Sie im Abschnitt »Typumwandlungen« in Anhang D.

3.5 Dialogfelder verwenden

In diesem Abschnitt erläutern wir, wie Sie Dialogfelder in Qt verwenden – wie Sie sie erstellen und initialisieren, ausführen und auf die Auswahl reagieren, die von dem Benutzer getroffen wird, der mit den Dialogfeldern interagiert. Wir nutzen die Dialogfelder `FIND`, `GO TO CELL` und `SORT`, die wir in Kapitel 2 erstellt haben. Außerdem erstellen wir ein einfaches `ABOUT`-Feld.

Wir beginnen mit dem Dialogfeld `FIND`, das in Abbildung 3.12 zu sehen ist. Da wir den Benutzer in die Lage versetzen wollen, nach Belieben zwischen dem Hauptfenster von `Spreadsheet` und dem Dialogfeld `FIND` zu wechseln, darf das Dialogfeld nicht modal sein. Ein Fenster ist *nicht modal*, wenn es sich unabhängig von allen anderen Fenstern der Anwendung ausführen lässt.

Beim Erstellen von nicht modalen Dialogfeldern werden deren Signale normalerweise mit Slots verbunden, die auf die Benutzerinteraktionen reagieren.

```
void MainWindow::find()
{
    if (!findDialog) {
        findDialog = new FindDialog(this);
        connect(findDialog, SIGNAL(findNext(const QString &,
                                           Qt::CaseSensitivity)),
                spreadsheet, SLOT(findNext(const QString &,
                                           Qt::CaseSensitivity)));
        connect(findDialog, SIGNAL(findPrevious(const QString &,
                                               Qt::CaseSensitivity)),
                spreadsheet, SLOT(findPrevious(const QString &,
                                               Qt::CaseSensitivity)));
    }
}
```

```
findDialog->show();
findDialog->raise();
findDialog->activateWindow();
}
```

Das Dialogfeld FIND ist ein Fenster, das dem Benutzer ermöglicht, die Tabellenkalkulation nach Text zu durchsuchen. Der Slot `find()` wird aufgerufen, wenn der Benutzer auf `EDIT|FIND` klickt, um das Dialogfeld FIND aufzurufen. An dieser Stelle sind mehrere Szenarios möglich:

- ▶ Der Benutzer hat das Dialogfeld FIND zum ersten Mal aufgerufen.
- ▶ Das Dialogfeld FIND wurde bereits zuvor aufgerufen, aber der Benutzer hat es geschlossen.
- ▶ Das Dialogfeld FIND wurde bereits zuvor aufgerufen und ist immer noch sichtbar.

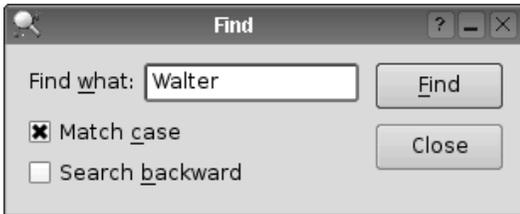


Abbildung 3.12: Das Dialogfeld FIND der Tabellenkalkulationsanwendung

Wenn das Dialogfeld FIND noch nicht existiert, erstellen wir es und verbinden seine Signale `findNext()` und `findPrevious()` mit den entsprechenden Spreadsheet-Slots. Das Dialogfeld ließe sich auch im `MainWindow`-Konstruktor erstellen. Bei verzögerter Erstellung startet die Anwendung aber schneller. Und wenn der Benutzer das Dialogfeld niemals verwendet, wird es auch nie erstellt, was sowohl Zeit als auch Speicher spart.

Dann rufen wir `show()`, `raise()` und `activateWindow()` auf, um sicherzustellen, dass das Fenster sichtbar ist, über den anderen Fenstern liegt und aktiv ist. Ein Aufruf von `show()` allein reicht aus, um ein ausgeblendetes Fenster als oberstes Fenster anzuzeigen und zu aktivieren, aber das Dialogfeld FIND kann aufgerufen werden, wenn sein Fenster bereits sichtbar ist. In diesem Fall hat `show()` keine Auswirkungen und wir müssen `raise()` und `activateWindow()` aufrufen, um das Fenster zum obersten zu machen und zu aktivieren. Alternativ hätten wir Folgendes schreiben können:

```
if (findDialog->isHidden()) {
    findDialog->show();
} else {
    findDialog->raise();
    findDialog->activateWindow();
}
```

Das ist aber in etwa so, als würde jemand vorsichtshalber in beide Richtungen schauen, bevor er eine Einbahnstraße überquert.

Wenden wir uns nun dem Dialogfeld GO TO CELL zu, das in Abbildung 3.13 zu sehen ist. Der Benutzer soll es öffnen, verwenden und schließen können, ohne in der Lage zu sein, in ein anderes Anwendungsfenster zu wechseln. Deshalb muss das Dialogfeld GO TO CELL modal sein. Ein *modales* Fenster wird bei seinem Aufruf geöffnet und blockiert dann die Anwendung, wodurch es jede andere Verarbeitung und Interaktionen verhindert, bis der Benutzer das Fenster schließt. Die von uns zuvor verwendeten Dateialog- und Meldungsfelder waren modal.

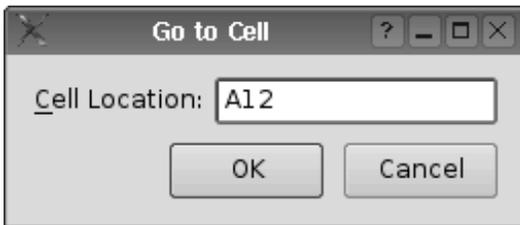


Abbildung 3.13: Das Dialogfeld GO TO CELL der Anwendung Spreadsheet

Ein Dialogfeld ist nicht modal, wenn es mithilfe von `show()` aufgerufen wird (sofern wir nicht vorher `setModal()` aufrufen, um es zu einem modalen Dialogfeld zu machen); es ist modal, wenn es mithilfe von `exec()` aufgerufen wird.

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                   str[0].unicode() - 'A');
    }
}
```

Die Funktion `QDialog::exec()` liefert den Wert »wahr« (`QDialog::Accepted`) zurück wenn der Dialog mit OK geschlossen wurde, andernfalls den Wert »falsch« (`QDialog::Rejected`). Denken Sie daran, dass wir OK mit `accept()` und CANCEL mit `reject()` verbunden haben, als wir das Dialogfeld GO TO CELL mithilfe von *Qt Designer* in Kapitel 2 erstellt haben. Wenn der Benutzer OK wählt, setzen wir die aktuelle Zelle auf den im Zeileneditor eingegebenen Wert.

Die Funktion `QTableWidget::setCurrentCell()` erwartet zwei Argumente: einen Zeilen- und einen Spaltenindex. In der Tabellenkalkulation ist Zelle A1 die Zelle (0, 0) und Zelle B27 die Zelle (26, 1). Um den Zeilenindex aus dem von `QLineEdit::text()` zurückgegebenen `QString` zu erhalten, extrahieren wir die Zeilennummer mit der Funktion

QString::mid() (die einen Teilstring von der Anfangsposition bis zum Ende des Strings zurückgibt), konvertieren diesen Wert mithilfe von QString::toInt() in den Typ int und subtrahieren 1. Für die Spaltennummer subtrahieren wir den numerischen Wert für 'A' vom numerischen Wert des ersten Großbuchstaben im String. Wir wissen, dass der String das richtige Format aufweist, da der Validierer QRegExpValidator, den wir für das Dialogfeld erstellt haben, die Aktivierung der Schaltfläche OK nur gestattet, wenn der String aus einem Buchstaben mit drei darauf folgenden Ziffern besteht.

Die Funktion goToCell() unterscheidet sich insofern von dem gesamten bisher betrachteten Code, als dass sie ein Widget (GoToCellDialog) als Variable auf dem Stack erstellt. Mit einer Zeile mehr hätten wir genauso einfach new und delete verwenden können:

```
void MainWindow::goToCell()
{
    GoToCellDialog *dialog = new GoToCellDialog(this);
    if (dialog->exec()) {
        QString str = dialog->lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                   str[0].unicode() - 'A');
    }
    delete dialog;
}
```

Modale Dialogfelder (und Kontextmenüs) auf dem Stack zu erstellen, ist ein gebräuchliches Programmiermuster, da wir das Dialogfeld (oder Menü) nach der Benutzung normalerweise nicht mehr brauchen und es am Ende des umschließenden Gültigkeitsbereichs automatisch gelöscht wird.

Kommen wir nun zum Dialogfeld SORT. Dabei handelt es sich um ein modales Dialogfeld, das es dem Benutzer gestattet, den zurzeit ausgewählten Bereich nach den angegebenen Spalten zu sortieren. Abbildung 3.14 zeigt ein Beispiel für die Sortierung, wobei Spalte B den primären und Spalte A den sekundären Sortierschlüssel (beide in aufsteigender Reihenfolge) bildet.

	A	B	C	
1	George	Washington	1789-1797	
2	John	Adams	1797-1801	
3	Thomas	Jefferson	1801-1809	
4	James	Madison	1809-1817	
5	James	Monroe	1817-1825	
6	John Quincy	Adams	1825-1829	
7	Andrew	Jackson	1829-1837	
8				

(a) Vor dem Sortieren

	A	B	C	
1	John	Adams	1797-1801	
2	John Quincy	Adams	1825-1829	
3	Andrew	Jackson	1829-1837	
4	Thomas	Jefferson	1801-1809	
5	James	Madison	1809-1817	
6	James	Monroe	1817-1825	
7	George	Washington	1789-1797	
8				

(b) Nach dem Sortieren

Abbildung 3.14: Sortieren des ausgewählten Bereichs in der Tabellenkalkulation

```

void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetItemSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());

    if (dialog.exec()) {
        SpreadsheetCompare compare;
        compare.keys[0] =
            dialog.primaryColumnCombo->currentIndex();
        compare.keys[1] =
            dialog.secondaryColumnCombo->currentIndex() - 1;
        compare.keys[2] =
            dialog.tertiaryColumnCombo->currentIndex() - 1;
        compare.ascending[0] =
            (dialog.primaryOrderCombo->currentIndex() == 0);
        compare.ascending[1] =
            (dialog.secondaryOrderCombo->currentIndex() == 0);
        compare.ascending[2] =
            (dialog.tertiaryOrderCombo->currentIndex() == 0);
        spreadsheet->sort(compare);
    }
}

```

Der Code in `sort()` folgt einem ähnlichen Muster, wie es für `gotoCell()` verwendet wurde:

- ▶ Wir erstellen das Dialogfeld auf dem Stack und initialisieren es.
- ▶ Wir öffnen das Dialogfeld mithilfe von `exec()`.
- ▶ Wenn der Benutzer auf OK klickt, extrahieren wir die von ihm eingegebenen Werte aus den Widgets des Dialogfelds und verarbeiten sie.

Der Aufruf von `setColumnRange()` setzt die für die Sortierung verfügbaren Spalten auf die ausgewählten Spalten. Mit der in Abbildung 3.14 gezeigten Auswahl würde `range.leftColumn()` beispielsweise 0 liefern, was `'A' + 0 = 'A'` ergibt, während `range.rightColumn()` den Wert 2 liefert, also `'A' + 2 = 'C'`.

Das Objekt `compare` speichert den primären, den sekundären und den tertiären Sortierungsschlüssel und ihre Sortierreihenfolge. (Wir werden die Definition der Klasse `SpreadsheetCompare` im nächsten Kapitel erörtern.) Das Objekt wird von `Spreadsheet::sort()` verwendet, um zwei Zeilen zu vergleichen. Das Array `keys` speichert die Spaltennummern der Schlüssel. Wenn die Auswahl beispielsweise von C2 bis E5 reicht, hat die Spalte C die Position 0. Das Array `ascending` speichert die mit den einzelnen Schlüsseln verbundene Reihenfolge als `bool`-Wert. `QComboBox::currentIndex()` gibt

den nullbasierten Index des zurzeit ausgewählten Elements zurück. Für den sekundären und tertiären Schlüssel subtrahieren wir 1 vom aktuellen Element, um das None-Element zu berücksichtigen.

Die Funktion `sort()` eignet sich zwar für diese Aufgabe, ist aber mit etwas Vorsicht zu genießen. Sie setzt voraus, dass das Dialogfeld SORT in einer bestimmten Weise implementiert ist, und zwar mit Kombinationsfeldern und None-Elementen. Das bedeutet, dass wir beim Neuedesign des Dialogfelds SORT möglicherweise auch diesen Code neu schreiben müssen. Dieser Ansatz ist durchaus brauchbar, wenn man das Dialogfeld nur von einer Stelle aus aufruft. Wird das Dialogfeld jedoch an mehreren Stellen verwendet, kann die Wartung des Codes leicht zum Albtraum werden.

Solider und raffinierter ist es, die Klasse `SortDialog` ein `SpreadsheetCompare`-Objekt erstellen zu lassen, auf das dann der Aufrufer zugreifen kann. Die Funktion `MainWindow::sort()` vereinfacht sich dadurch erheblich:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetItemSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());
    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

Dieser Ansatz führt zu lose gekoppelten Komponenten und stellt fast immer die richtige Wahl für Dialogfelder dar, die von mehr als einer Stelle aus aufgerufen werden.

Ein radikalerer Ansatz besteht darin, bei der Initialisierung des Objekts `SortDialog` einen Zeiger auf das Objekt `Spreadsheet` zu übergeben und dem Dialogfeld zu gestatten, direkt mit `Spreadsheet` zu arbeiten. Zwar büßt `SortDialog` dadurch an Allgemeingültigkeit ein, da das Objekt nur bei einem bestimmten `Widget`-Typ funktioniert, der Code vereinfacht sich aber noch weiter, indem die Funktion `SortDialog::setColumnRange()` entfernt wird. Die Funktion `MainWindow::sort()` sieht dann folgendermaßen aus:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

Dieser Ansatz verhält sich spiegelbildlich zum ersten: Anstelle des Aufrufers benötigt nun das Dialogfeld eingehende Kenntnisse über die vom Aufrufer gelieferten Datenstrukturen. Dieser Ansatz kann nützlich sein, wenn das Dialogfeld Änderungen in

Echtzeit anwenden muss. Doch ebenso wie der aufrufende Code beim ersten Ansatz heikel ist, versagt dieser dritte Ansatz, wenn sich die Datenstrukturen ändern.

Manche Entwickler entscheiden sich einfach für einen Ansatz, wie Dialogfelder zu verwenden sind, und halten dann daran fest. Der Code wirkt vertrauter und einfacher, da sie sämtliche Dialogfelder stets nach demselben Muster verwenden. Allerdings verschonen sie auch die Vorteile der anderen Ansätze. Im Idealfall sollten Sie über Vorgehensweise für jedes Dialogfeld einzeln entscheiden.

Wir runden diesen Abschnitt mit dem Feld ABOUT ab. Wie im Fall von FIND oder GO TO CELL könnten wir ein benutzerdefiniertes Dialogfeld erstellen, um die Informationen über die Anwendung zu präsentieren, aber da die meisten Informationsfelder dieser Art recht stilisiert sind, bietet Qt eine einfachere Lösung.

```
void MainWindow::about()
{
    QMessageBox::about(this, tr("About Spreadsheet"),
        tr("<h2>Spreadsheet 1.1</h2>"
            "<p>Copyright &copy; 2008 Software Inc."
            "<p>Spreadsheet is a small application that "
            "demonstrates QAction, QMainWindow, QMenuBar, "
            "QStatusBar, QTableWidgetItem, QToolBar, and many other "
            "Qt classes."));
}
```

Das Feld ABOUT erhalten Sie durch den Aufruf von `QMessageBox::about()`, einer statischen Hilfsfunktion. Sie weist große Ähnlichkeiten mit `QMessageBox::warning()` auf, außer dass sie das Symbol des übergeordneten Fensters und nicht das Standardsymbol für Warnungen verwendet. Das resultierende Dialogfeld ist in Abbildung 3.15 zu sehen.

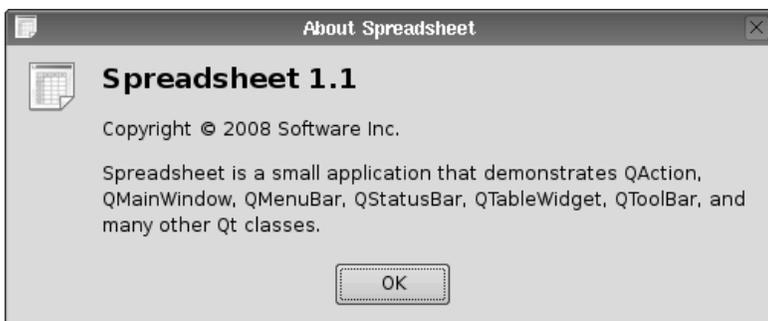


Abbildung 3.15: Das Info-Dialogfeld ABOUT SPREADSHEET

Bisher haben wir mehrere statische Hilfsfunktionen sowohl von `QMessageBox` als auch von `QFileDialog` verwendet. Diese Funktionen erstellen ein Dialogfeld, initialisieren es und rufen `exec()` auf. Darüber hinaus ist es möglich, wenn auch nicht so bequem, ein `QMessageBox`- oder `QFileDialog`-Widget wie alle anderen Widgets zu erstellen und `exec()` oder sogar `show()` explizit dafür aufzurufen.

3.6 Einstellungen speichern

Im Konstruktor `MainWindow` haben wir `readSettings()` aufgerufen, um die gespeicherten Einstellungen der Anwendung aufzurufen. In ähnlicher Weise haben wir `writeSettings()` in `closeEvent()` aufgerufen, um die Einstellungen zu speichern. Diese beiden Funktionen sind die letzten Memberfunktionen von `MainWindow`, die implementiert werden müssen.

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    settings.setValue("geometry", saveGeometry());
    settings.setValue("recentFiles", recentFiles);
    settings.setValue("showGrid", showGridAction->isChecked());
    settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}
```

Die Funktion `writeSettings()` speichert die Geometrie (Position und Größe) des Hauptfensters, die Liste der zuletzt geöffneten Dateien und die Optionen `SHOW GRID` und `AUTO-RECALCULATE`.

Standardmäßig speichert `QSettings` die Einstellungen der Anwendung an plattform-spezifischen Orten. Unter Windows wird die Systemregistrierung verwendet, unter Unix werden die Daten in Textdateien gespeichert und unter Mac OS X wird das Core Foundation Preferences-API verwendet.

Die Konstruktorargumente geben den Namen der Organisation und der Anwendung an. Diese Informationen werden in plattform-spezifischer Weise verwendet, um einen Speicherort für die Einstellungen zu finden.

`QSettings` speichert Einstellungen als *Schlüssel/Wert*-Paare. Der *Schlüssel* ähnelt dem Pfad eines Dateisystems. Unterschlüssel können mithilfe einer pfadähnlichen Syntax (z.B. `findDialog/matchCase`) oder unter Verwendung von `beginGroup()` und `endGroup()` angegeben werden:

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

Bei Wert kann es sich um `int`, `bool`, `double`, `QString`, `QStringList` oder jeden anderen von `QVariant` unterstützten Typ handeln, wozu auch registrierte benutzerdefinierte Typen gehören.

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    restoreGeometry(settings.value("geometry").toByteArray());

    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();

    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);

    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

Die Funktion `readSettings()` lädt die von `writeSettings()` gespeicherten Einstellungen. Das zweite Argument der Funktion `value()` gibt einen Standardwert an, falls keine Einstellungen zur Verfügung stehen. Die Standardwerte werden bei der ersten Ausführung der Anwendung verwendet. Da kein zweites Argument für die Geometrie oder für die Liste der zuletzt geöffneten Dateien vorgegeben ist, erhält das Fenster willkürliche, aber vernünftige Werte für Größe und Position und die Liste der zuletzt geöffneten Dateien ist bei der ersten Ausführung leer.

Die von uns in `MainWindow` gewählte Anordnung mit all dem auf `QSettings` bezogenen Code in `readSettings()` und `writeSettings()` ist nur einer von vielen Ansätzen. Ein `QSettings`-Objekt kann jederzeit während der Ausführung der Anwendung von einer beliebigen Stelle im Code aus erstellt werden, um bestimmte Einstellungen abzufragen oder zu ändern.

Damit ist die Implementierung von `MainWindow` für die Tabellenkalkulation abgeschlossen. Die folgenden Abschnitte zeigen, wie Sie die Anwendung modifizieren, sodass sie mehrere Dokumente verarbeitet, und wie Sie einen Startbildschirm implementieren. Die Funktionalität, einschließlich der Formelbehandlung und Sortierung, wird im nächsten Kapitel vervollständigt.

3.7 Mehrere Dokumente

Jetzt können Sie den Code für die Funktion `main()` der Anwendung Spreadsheet schreiben:

```
#include <QApplication>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

Die Funktion `main()` unterscheidet sich ein wenig von den bisher beschriebenen Funktionen. Anstatt `new` zu verwenden, haben wir die Instanz von `MainWindow` als Variable auf dem Stack erstellt. Die Instanz wird dann automatisch gelöscht, wenn die Funktion beendet ist.

Bei der oben gezeigten Funktion `main()` stellt die Anwendung Spreadsheet ein einzelnes Hauptfenster bereit und kann jeweils nur ein Dokument verarbeiten. Wenn wir mehrere Dokumente gleichzeitig bearbeiten wollen, können wir mehrere Instanzen der Tabellenkalkulationsanwendung starten. Dies ist aber für die Benutzer nicht so komfortabel wie eine einzelne Instanz der Anwendung, die mehrere Hauptfenster bereitstellt, wie Sie es beispielsweise von einem Webbrowser kennen, der gleichzeitig mehrere Browserfenster bereitstellen kann.

Wir werden die Tabellenkalkulationsanwendung so verändern, dass sie mehrere Dokumente verarbeiten kann. Zuerst brauchen wir ein etwas anderes FILE-Menü:

- ▶ `FILE|NEW` erstellt ein neues Hauptfenster mit einem leeren Dokument, anstatt das vorhandene Hauptfenster wieder zu verwenden.
- ▶ `FILE|CLOSE` schließt das aktuelle Hauptfenster.
- ▶ `FILE|EXIT` schließt alle Fenster.

Die ursprüngliche Version des Datei-Menüs enthielt noch nicht die Option `CLOSE`, da dies derselbe Vorgang gewesen wäre wie `EXIT`. Das neue FILE-Menü ist in Abbildung 3.16 zu sehen.

Die neue Funktion `main()` sieht wie folgt aus:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
```

```

    mainWin->show();
    return app.exec();
}

```

Bei mehreren Fenstern ist es nun sinnvoll, `MainWindow` mit `new` zu erstellen, weil wir `delete` für ein Hauptfenster aufrufen und somit Platz sparen können, wenn wir ein Fenster nicht mehr brauchen.

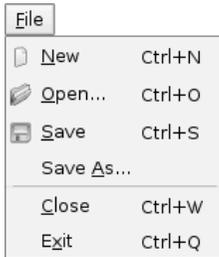


Abbildung 3.16: Das neue Datei-Menü

Der neue Slot `MainWindow::newFile()` sieht wie folgt aus:

```

void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
}

```

Wir erstellen einfach eine neue Instanz von `MainWindow`. Es mag merkwürdig erscheinen, dass wir keinen Zeiger auf das neue Fenster speichern, aber das ist kein Problem, da Qt alle Fenster für uns im Auge behält.

Die Aktionen für `CLOSE` und `EXIT` sehen folgendermaßen aus:

```

void MainWindow::createActions()
{
    ...
    closeAction = new QAction(tr("&Close"), this);
    closeAction->setShortcut(QKeySequence::Close);
    closeAction->setStatusTip(tr("Close this window"));
    connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));

    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()),
            qApp, SLOT(closeAllWindows()));
    ...
}

```

Der Slot `QApplication::closeAllWindows()` schließt alle Fenster der Anwendung, sofern keines dieses Ereignis verweigert. Dies ist genau das Verhalten, das wir hier brauchen. Wir müssen uns keine Gedanken über nicht gespeicherte Änderungen machen, da dieser Vorgang jedes Mal von `MainWindow::closeEvent()` verarbeitet wird, wenn ein Fenster geschlossen wird.

Es sieht so aus, als wäre die Anwendung jetzt in der Lage, mehrere Fenster zu verarbeiten. Leider lauert da noch ein verstecktes Problem: Wenn der Benutzer fortwährend Hauptfenster erstellt und schließt, geht dem Rechner möglicherweise der Speicher aus. Das liegt daran, dass wir ständig `MainWindow`-Widgets in `newFile()` erstellen, sie aber nie löschen. Wenn der Benutzer ein Hauptfenster schließt, wird es standardmäßig ausgeblendet und bleibt deshalb weiterhin im Speicher. Bei einer großen Anzahl von Hauptfenstern kann das problematisch werden.

Die Lösung besteht darin, im Konstruktor das Attribut `Qt::WA_DeleteOnClose` festzulegen:

```
MainWindow::MainWindow()
{
    ...
    setAttribute(Qt::WA_DeleteOnClose);
    ...
}
```

Damit wird Qt angewiesen, das Fenster nach dem Schließen zu löschen. Das Attribut `Qt::WA_DeleteOnClose` ist eines von vielen Flags, die für die Klasse `QWidget` gesetzt werden können, um deren Verhalten zu beeinflussen.

Speicherlecks sind nicht das einzige Problem, um das wir uns kümmern müssen. Unser ursprüngliches Anwendungsdesign hat implizit angenommen, dass es nur ein Hauptfenster gibt. Bei mehreren Fenstern verfügt jedes Hauptfenster über eine eigene Liste der zuletzt geöffneten Dateien und über seine eigenen Optionen. Zweifellos sollte die Liste der zuletzt geöffneten Dateien global für die gesamte Anwendung gelten. Dies lässt sich recht einfach erreichen, indem wir `recentFiles` als statische Variable deklarieren, sodass nur eine einzige Instanz der Variablen für die gesamte Anwendung existiert. Dann müssen wir aber sicherstellen, dass wir die Funktion `updateRecentFileActions()` an allen Stellen, wo wir sie bisher aufgerufen haben, um das Datei-Menü zu aktualisieren, nun für alle Hauptfenster aufrufen. Dies lässt sich mit dem folgenden Code erreichen:

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))
        mainWin->updateRecentFileActions();
}
```

Der Code verwendet das Qt-Konstrukt `foreach` (das in Kapitel 11 erläutert wird), um eine Iteration über alle Anwendungsfenster durchzuführen, und ruft `updateRecentFileActions()` für alle Widgets vom Typ `MainWindow` auf. Mit ähnlichem Code können Sie die Optionen `SHOW GRID` und `AUTO-RECALCULATE` synchronisieren oder sicherstellen, dass dieselbe Datei nicht zweimal geladen wird.

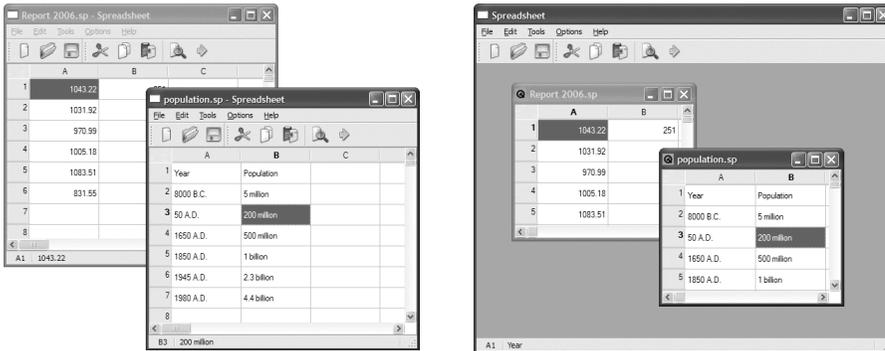


Abbildung 3.17: SDI im Vergleich zu MDI

Anwendungen, die ein Dokument pro Hauptfenster bereitstellen, werden als SDI-Anwendungen (Single Document Interface) bezeichnet. Eine gebräuchliche Alternative unter Windows ist MDI (Multiple Document Interface), wobei die Anwendung über ein Hauptfenster verfügt, das sowohl zum Erstellen von SDI- als auch MDI-Anwendungen auf allen unterstützten Plattformen verwendet werden kann. Abbildung 3.17 zeigt die Anwendung Spreadsheet unter Verwendung beider Ansätze. MDI wird in Kapitel 6 erläutert.

3.8 Startbildschirme

Viele Anwendungen präsentieren beim Startvorgang einen Startbildschirm. Einige Entwickler verwenden einen Startbildschirm, um einen langsamen Start zu verschleiern, während andere ihn benutzen, um ihre Marketingabteilungen zufriedenzustellen. Qt-Anwendungen einen Startbildschirm hinzuzufügen, ist mithilfe der Klasse `QSplashScreen` sehr einfach.

Die Klasse `QSplashScreen` zeigt vor dem Erscheinen des Hauptfensters ein Bild an. Sie kann auch Meldungen in das Bild schreiben, um den Benutzer über den Fortschritt des Initialisierungsprozesses der Anwendung zu informieren. Normalerweise befindet sich der Code für den Startbildschirm in der Funktion `main()` vor dem Aufruf von `QApplication::exec()`.



Abbildung 3.18: Ein Startbildschirm

Im folgenden Beispiel präsentiert die Funktion `main()` mit `QSplashScreen` einen Startbildschirm für eine Anwendung, die beim Startvorgang Module lädt und Netzwerkverbindungen herstellt.

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

    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();

    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;

    splash->showMessage(QObject::tr("Setting up the main window..."),
                      topRight, Qt::white);
    MainWindow mainWin;

    splash->showMessage(QObject::tr("Loading modules..."),
                      topRight, Qt::white);
    loadModules();

    splash->showMessage(QObject::tr("Establishing connections..."),
                      topRight, Qt::white);
    establishConnections();
}
```

```
mainWin.show();
splash->finish(&mainWin);
delete splash;

return app.exec();
}
```

Damit haben wir die Benutzeroberfläche der Anwendung Spreadsheet fertig gestellt. Im nächsten Kapitel werden wir die Anwendung vervollständigen, indem wir die Kernfunktionalität der Tabellenkalkulation implementieren.