PEARSON Education

ADDISON-WESLEY

Jasmin Blanchette, Mark Summerfield

C++

GUI-Programmierung mit Qt 3

Alle Features wie Signals, Slots,
Events, Layouts und Graphics

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Hauptfenster einrichten

In diesem Kapitel lernen Sie, wie Sie mit Qt Hauptfenster erzeugen. Am Ende sollten Sie in der Lage sein, eine komplette Benutzerschnittstelle zu erstellen, einschließlich Menüs, Werkzeugleisten, Statuszeilen und so vielen Dialogen, wie die Anwendung erfordert.

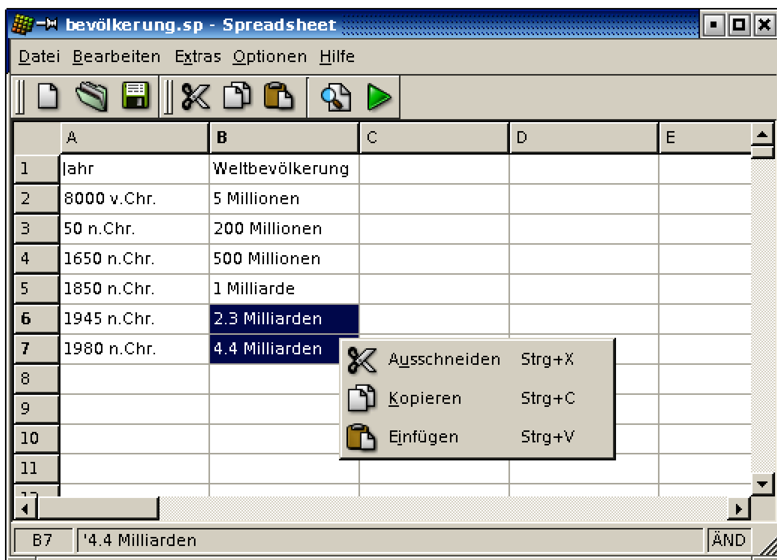


Abbildung 3.1: Spreadsheet als Beispiel für ein Tabellenkalkulationsprogramm

Das Hauptfenster einer Anwendung ist das Grundgerüst, auf dem die Benutzeroberfläche der Anwendung aufgebaut wird. Grundlage dieses Kapitels ist das Hauptfenster der Spreadsheet-Anwendung aus Abbildung 3.1. Spreadsheet ist ein Tabellenkalkulationsprogramm und verwendet die in ***Kapitel 2*** erzeugten Dialoge SUCHEN, GEHE ZU ZELLE und SORTIEREN.

Hinter den meisten GUI-Anwendungen steht zusätzlicher Quellcode, der die eigentliche Funktionalität bereitstellt – beispielsweise Code zum Lesen aus und Schreiben in Dateien oder Code zum Verarbeiten der Daten, die auf der Benutzeroberfläche ange-

zeigt werden. Wie diese Funktionalität implementiert wird, werden Sie – wiederum am Beispiel der hier entwickelten Spreadsheet-Anwendung – in ***Kapitel 4*** sehen.

3.1 Klassen von QMainWindow ableiten

Für das Hauptfenster einer Anwendung müssen Sie eine Klasse von `QMainWindow` ableiten. Viele der Techniken zur Dialogerstellung, die Sie in ***Kapitel 2*** kennen gelernt haben, lassen sich auf die Erzeugung von Hauptfenstern übertragen, da sowohl `QDialog` als auch `QMainWindow` von `QWidget` abgeleitet sind.

Sie können Hauptfenster auch mit dem *Qt Designer* erzeugen, doch zum besseren Verständnis wollen wir in diesem Kapitel den Code von Hand aufsetzen. Wer den eher visuellen Ansatz vorzieht, sollte im *Qt Designer*-Handbuch das Kapitel »Creating a Main Window Application« nachlesen.

Der Quellcode für das Hauptfenster der Spreadsheet-Anwendung ist verteilt auf die beiden Dateien *mainwindow.h* und *mainwindow.cpp*. Beginnen wir mit der Headerdatei:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <qmainwindow.h>
#include <qstringlist.h>

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

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0, const char *name = 0);

protected:
    void closeEvent(QCloseEvent *event);
    void contextMenuEvent(QContextMenuEvent *event);
```

In dem Code wird die Klasse `MainWindow` von der Klasse `QMainWindow` abgeleitet. Sie enthält das Makro `Q_OBJECT`, da sie über eigene Signale und Slots verfügt.

Die Funktion `closeEvent()` ist als virtuelle Funktion in `QWidget` deklariert und wird automatisch aufgerufen, wenn der Benutzer das Fenster schließt. Sie wird in `MainWindow` reimplementiert, damit wir dem Benutzer die Standardfrage »Sollen die Änderun-

gen gespeichert werden?« stellen und Benutzerpräferenzen auf Festplatte sichern können.

Die Funktion `contextMenuEvent()` wird aufgerufen, wenn der Benutzer ein Widget mit der rechten Maustaste anklickt oder die auf seiner Plattform geltende Kontextmenü-Taste drückt. Sie wird in `MainWindow` reimplementiert, um ein Kontextmenü einzublenden.

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

Einige Menübefehle wie DATEI/NEU oder HILFE/INFO sind in `MainWindow` als private-Slots implementiert. Die meisten Slots haben Rückgabewerte vom Typ `void`. Ausnahmen hierzu bilden `save()` und `saveAs()`, die einen booleschen Rückgabewert haben. Wird ein Slot als Antwort auf ein Signal ausgeführt, wird der Rückgabewert ignoriert, wird der Slot jedoch als Funktion aufgerufen, steht der Rückgabewert wie bei jeder normalen C++-Funktion zur Verfügung.

```
    void updateCellIndicators();
    void spreadsheetModified();
    void openRecentFile(int param);

private:
    void createActions();
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool maybeSave();
    void loadFile(const QString &fileName);
    void saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    void updateRecentFileItems();
    QString strippedName(const QString &fullFileName);
```

Zur Unterstützung der Benutzerschnittstelle benötigt das Hauptfenster noch einige weitere private-Slots und -Funktionen.

```
    Spreadsheet *spreadsheet;
    FindDialog *findDialog;
    QLabel *locationLabel;
    QLabel *formulaLabel;
    QLabel *modLabel;
    QStringList recentFiles;
    QString curFile;
    QString fileFilters;
    bool modified;

    enum { MaxRecentFiles = 5 };
    int recentFileIds[MaxRecentFiles];

    QPopupMenu *fileMenu;
    QPopupMenu *editMenu;
    QPopupMenu *selectSubMenu;
    QPopupMenu *toolsMenu;
    QPopupMenu *optionsMenu;
    QPopupMenu *helpMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QAction *newAct;
    QAction *openAct;
    QAction *saveAct;
    ...
    QAction *aboutAct;
    QAction *aboutQtAct;
};

#endif
```

Zusätzlich zu den private-Slots und -Funktionen verfügt `MainWindow` über eine Reihe von ebenfalls als `private` deklarierten Variablen – die jeweils im Zuge ihrer ersten Verwendung besprochen werden.

Gehen wir nun die Implementierung durch:

```
#include <qaction.h>
#include <qapplication.h>
#include <qcombobox.h>
#include <qfiledialog.h>
#include <qlabel.h>
#include <qlineedit.h>
#include <qmenubar.h>
#include <qmessagebox.h>
#include <qpopupmenu.h>
#include <qsettings.h>
#include <qstatusbar.h>
```

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

Die `#include`-Direktiven inkludieren die Headerdateien von allen Qt-Klassen, die in unserer abgeleiteten Klasse verwendet werden, sowie einige eigene Headerdateien, einschließlich *finddialog.h*, *gotocelldialog.h* und *sortdialog.h* aus ***Kapitel* 2**.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
    spreadsheet = new Spreadsheet(this);
    setCentralWidget(spreadsheet);

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

    readSettings();

    setCaption(tr("Spreadsheet"));
    setIcon(QPixmap::fromMimeSource("icon.png"));

    findDialog = 0;
    fileFilters = tr("Spreadsheet-Dateien (*.sp)");
    modified = false;
}
```

Der Konstruktor beginnt mit der Erzeugung eines `Spreadsheet`-Widgets, das zum »zentralen Widget« des Hauptfensters erhoben wird. Dieses Widget belegt den Raum zwischen den Werkzeugleisten und der Statuszeile. Die Klasse `Spreadsheet` ist von `QTable` abgeleitet und besitzt Funktionalität wie sie für Tabellenkalkulationsprogramme typisch ist – beispielsweise die Berechnung von Formeln mit Zellenangaben. Die Implementierung dieser Klasse erfolgt in ***Kapitel* 4**.

Danach erzeugen wir den Rest des Hauptfensters durch Aufruf der `private`-Funktionen `createActions()`, `createMenus()`, `createToolBars()` und `createStatusBar()`. Außerdem rufen wir die `private`-Funktion `readSettings()` auf, um die gespeicherten Anwendungseinstellungen einzulesen.

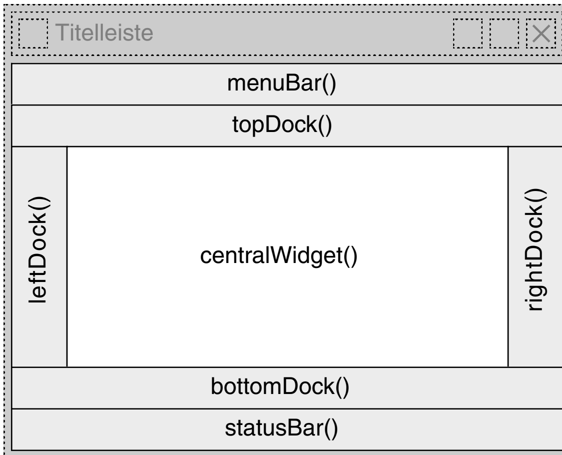


Abbildung 3.2: Die Widgets von QMainWindow

Für das Fenstersymbol verwenden wir die Bilddatei *icon.png* (eine PNG-Datei). Qt unterstützt eine Vielzahl von Bildformaten, einschließlich BMP, GIF[1], JPEG, MNG, PNG, PNM, XBM und XPM. Der Aufruf von `QWidget::setIcon()` setzt das Symbol, das oben links im Fenster angezeigt wird. Leider gibt es keine plattformunabhängige Möglichkeit, das auf dem Desktop angezeigte Anwendungssymbol zu setzen (siehe *http://doc.trolltech.com/3.2/appicon.html*).

GUI-Anwendungen verwenden in der Regel viele Bilder, wobei manche Bilder mehrfach, aber in unterschiedlichen Kontexten eingesetzt werden. In Qt gibt es eine Vielzahl von Methoden, Anwendungen mit Bildern auszustatten. Die häufigsten sind:

- Die Bilder werden in Dateien gespeichert und zur Laufzeit geladen.
- XPM-Dateien werden in den Quellcode eingebunden. (Dies funktioniert, weil XPM-Dateien auch gültige C++-Dateien sind.)
- Qts Mechanismus der »Bildersammlung«.

Hier wollen wir uns auf den Ansatz der »Bildersammlung« konzentrieren, da er leichter und effizienter ist als das Laden von Dateien zur Laufzeit und mit jedem unterstützten Dateiformat funktioniert. Die Bilder werden im Quellcodebaum in einem Unterverzeichnis namens *images* gespeichert. Durch Aufnahme eines `IMAGES`-Abschnitts

1 Wenn Sie in einem Land leben, das Software-Patente anerkennt und in dem Unisys ein Patent auf die LZW-Dekomprimierung hält, kann es sein, dass Unisys für die Nutzung der GIF-Technologie den Erwerb einer Lizenz fordert. Deshalb ist GIF standardmäßig in Qt deaktiviert. Unseres Wissens nach dürfte das Patent Ende 2004 weltweit abgelaufen sein.

```
IMAGES        = images/icon.png \
                images/new.png \
                images/open.png \
                ...
                images/find.png \
                images/gotocell.png
```

in die *.pro*-Datei der Anwendung, teilen wir *uic* mit, eine C++-Quellcodedatei zu erzeugen, die die Daten für alle angegebenen Bilder enthält. Diese Daten werden dann in die ausführbare Datei der Anwendung kompiliert und können mit `QPixmap::fromMimeSource()` ausgelesen werden. Der Vorteil liegt darin, dass Symbole und andere Bilder nicht verloren gehen können; sie befinden sich immer in der ausführbaren Datei.

Wer seine Hauptfenster und Dialoge mit *Qt Designer* erstellt, kann den Designer auch dazu verwenden, die *.pro*-Datei zu bearbeiten und Bilder in die Bildersammlung aufzunehmen.

3.2 Menüs und Werkzeugleisten einrichten

Die meisten modernen GUI-Anwendungen verfügen sowohl über Menüs als auch über Werkzeugleisten, wobei beide mehr oder weniger die gleichen Befehle aufweisen. Anhand der Menüs können die Benutzer die Anwendung kennen lernen und herausfinden, wie man mit ihr arbeitet, während die Werkzeugleisten für schnellen Zugriff auf häufig benötigte Funktionen sorgen.

Qt vereinfacht die Programmierung von Menüs und Werkzeugleisten durch sein »Aktionen«-Konzept. Eine *Aktion* ist ein Element, das in wenigen Schritten einem Menü, einer Werkzeugleiste oder beiden hinzugefügt werden kann:

- Die Aktionen erzeugen.
- Die Aktionen den Menüs hinzufügen.
- Die Aktionen den Werkzeugleisten hinzufügen.

In der Spreadsheet-Anwendung werden die Aktionen mit `createActions()` erzeugt:

```
void MainWindow::createActions()
{
    newAct = new QAction(tr("&Neu"), tr("Ctrl+N"), this);
    newAct->setIconSet(QPixmap::fromMimeSource("new.png"));
    newAct->setStatusTip(tr("Erstellt ein neues Tabellenblatt-Dokument"));
    connect(newAct, SIGNAL(activated()), this, SLOT(newFile()));
```

Die Aktion NEU verfügt über eine Zugriffstaste (N), ein Tastaturkürzel (`Strg`+`N`)[2], ein Elternelement (das Hauptfenster), ein Symbol (*new.png*) und eine Kurzbeschreibung in der Statuszeile. Das `activated()`-Signal der Aktion ist mit dem private-Slot `newFile()` des Hauptfensters verbunden, welchen wir im nächsten Abschnitt implementieren werden. Ohne diese Verbindung würde nichts passieren, wenn der Benutzer das Menüelement DATEI/NEU auswählt oder den NEU-Schalter in der Werkzeugleiste anklickt.

Die anderen Aktionen für die Menüs DATEI, BEARBEITEN und EXTRAS sind analog zur NEU-Aktion aufgebaut.

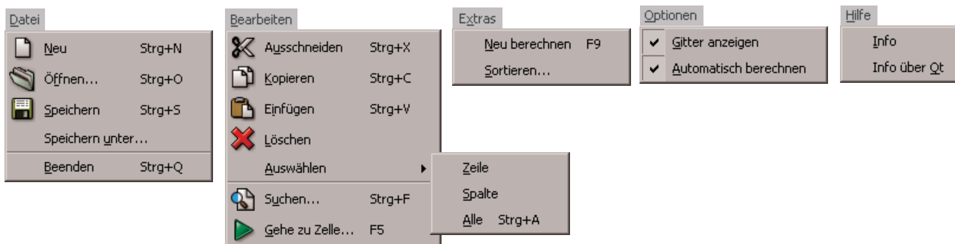


Abbildung 3.3: Die Menüs der Spreadsheet-Anwendung

Eine Ausnahme bildet die Aktion GITTER ANZEIGEN aus dem Menü OPTIONEN:

```
showGridAct = new QAction(tr("&Gitter anzeigen"), 0, this);
showGridAct->setToggleAction(true);
showGridAct->setOn(spreadsheet->showGrid());
showGridAct->setStatusTip(tr("Blendet das Gitter des Tabellenblatts ein "
                             "und aus"));
connect(showGridAct, SIGNAL(toggled(bool)),
        spreadsheet, SLOT(setShowGrid(bool)));
```

GITTER ANZEIGEN ist eine Umschaltaktion. Sie wird im Menü mit einem Häkchen vor dem Befehl angezeigt und ist in der Werkzeugleiste als Toggle-Button implementiert. Wenn die Aktion eingeschaltet wird, zeigt die `Spreadsheet`-Komponente ein Gitter an. Wir initialisieren die Aktion mit der Standardeinstellung der `Spreadsheet`-Komponente, damit Aktion und Komponente übereinstimmen. Anschließend richten wir eine Verbindung zwischen dem `toggled(bool)`-Signal der GITTER ANZEIGEN-Aktion und

2 Anm.d.Übers.: Im QAction-Konstruktor müssen Sie für die Tastaturkürzel immer die englischen, in <`qnamespace.h`> definierten Tastennamen (CTRL, ALT...) verwenden; deutsche Tastenbezeichnungen wie z.B. STRG werden nicht erkannt! Um dennoch in den Menüs die deutschen statt der englischen Tastenbezeichnungen anzuzeigen, müssen Sie die Tastaturkürzel wie im Beispiel gezeigt in `tr()`-Aufrufe einfassen, die Übersetzungsdatei *qt-de.qm* aus dem *translations*-Verzeichnis von Qt in das Verzeichnis der Anwendung kopieren und die `main()`-Funktion um Code ergänzen, der die Übersetzungsdatei dynamisch lädt (siehe Readme-Datei im *spreadsheet*-Verzeichnis auf der Buch-CD). Ausführlichere Informationen zur Lokalisierung von Anwendungen finden Sie in Kapitel 15.

dem `setShowGrid(bool)`-Slot der `Spreadsheet`-Komponente ein, die von `QTable` abgeleitet ist. Sobald diese Aktion einem Menü oder einer Werkzeugleiste hinzugefügt wird, kann der Benutzer das Gitter ein- und ausblenden lassen.

Die Aktionen GITTER ANZEIGEN und AUTOMATISCH BERECHNEN sind voneinander unabhängige Umschaltaktionen. Mithilfe von `QAction` und der davon abgeleiteten `QActionGroup`-Klasse können Sie sogar sich gegenseitig ausschließende Aktionen realisieren.

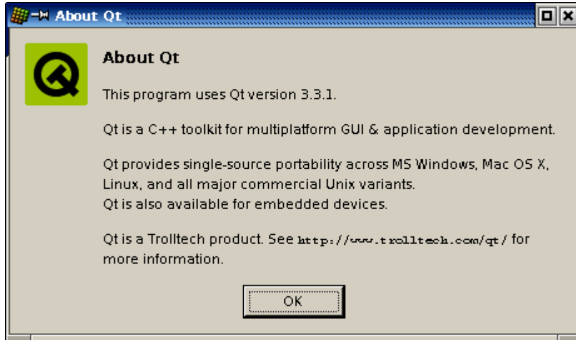


Abbildung 3.4: Infos über Qt

```
    aboutQtAct = new QAction(tr("Info über &Qt"), 0, this);
    aboutQtAct->setStatusTip(tr("Zeigt den Info-Dialog der Qt-Bibliothek "
                                "an"));
    connect(aboutQtAct, SIGNAL(activated()), qApp, SLOT(aboutQt()));
```

Für den INFO ÜBER QT-Befehl verwenden wir den `aboutQt()`-Slot des `QApplication`-Objekts, auf den wir über die globale Variable `qApp` zugreifen können.

Nachdem wir die Aktionen erzeugt haben, können wir uns dem Menüsystem zuwenden, über das die Aktionen aufgerufen werden:

```
void MainWindow::createMenus()
{
    fileMenu = new QPopupMenu(this);
    newAct->addTo(fileMenu);
    openAct->addTo(fileMenu);
    saveAct->addTo(fileMenu);
    saveAsAct->addTo(fileMenu);
    fileMenu->insertSeparator();
    exitAct->addTo(fileMenu);

    for (int i = 0; i < MaxRecentFiles; ++i)
        recentFileIds[i] = -1;
```

In Qt sind alle Menüs Instanzen von `QPopupMenu`. Wir erzeugen das DATEI-Menü und ergänzen es um die Aktionen NEU, ÖFFNEN, SPEICHERN, SPEICHERN UNTER und BEENDEN. Um die inhaltlich zueinander gehörenden Befehle optisch zu gruppieren, fügen wir eine horizontale Trennlinie ein. Die `for`-Schleife übernimmt die Initialisierung des `recentFilesIds`-Arrays, das wir im nächsten Abschnitt für die Implementierung der Slots des DATEI-Menüs benötigen.

```
    editMenu = new QPopupMenu(this);
    cutAct->addTo(editMenu);
    copyAct->addTo(editMenu);
    pasteAct->addTo(editMenu);
    deleteAct->addTo(editMenu);

    selectSubMenu = new QPopupMenu(this);
    selectRowAct->addTo(selectSubMenu);
    selectColumnAct->addTo(selectSubMenu);
    selectAllAct->addTo(selectSubMenu);
    editMenu->insertItem(tr("&Auswählen"), selectSubMenu);

    editMenu->insertSeparator();
    findAct->addTo(editMenu);
    goToCellAct->addTo(editMenu);
```

Zum BEARBEITEN-Menü gehört ein Untermenü. Dieses Untermenü wird wie sein übergeordnetes Menü von `QPopupMenu` abgeleitet. Wir erzeugen das Untermenü, indem wir einfach `this` als Elternelement angeben und es an der Position in das BEARBEITEN-Menü einfügen, an der es erscheinen soll.

```
    toolsMenu = new QPopupMenu(this);
    recalculateAct->addTo(toolsMenu);
    sortAct->addTo(toolsMenu);

    optionsMenu = new QPopupMenu(this);
    showGridAct->addTo(optionsMenu);
    autoRecalcAct->addTo(optionsMenu);

    helpMenu = new QPopupMenu(this);
    aboutAct->addTo(helpMenu);
    aboutQtAct->addTo(helpMenu);

    menuBar()->insertItem(tr("&Datei"), fileMenu);
    menuBar()->insertItem(tr("&Bearbeiten"), editMenu);
    menuBar()->insertItem(tr("E&xtras"), toolsMenu);
    menuBar()->insertItem(tr("&Optionen"), optionsMenu);
    menuBar()->insertSeparator();
    menuBar()->insertItem(tr("&Hilfe"), helpMenu);
}
```

Die Menüs EXTRAS, OPTIONEN und HILFE werden analog erstellt und dann in die Menüleiste eingefügt. Die Funktion `QMainWindow::menuBar()` liefert einen Zeiger auf `QMenuBar` zurück. (Die Menüleiste wird erzeugt, wenn `menuBar()` das erste Mal aufgerufen wird). Zwischen den Menüs OPTIONEN und HILFE fügen wir ein Trennelement ein. In Motif und ähnlichen Designs wird dadurch das HILFE-Menü rechtsbündig ausgerichtet. Andere Stile ignorieren das Trennelement.

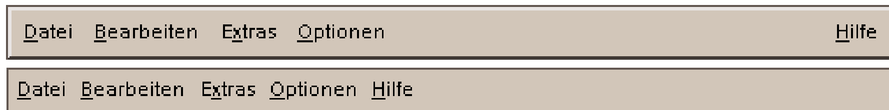


Abbildung 3.5: Menüleiste in Motif und im Windows-Stil

Werkzeugleisten werden ähnlich wie Menüs erzeugt:

```
void MainWindow::createToolBars()
{
    fileToolBar = new QToolBar(tr("Datei"), this);
    newAct->addTo(fileToolBar);
    openAct->addTo(fileToolBar);
    saveAct->addTo(fileToolBar);

    editToolBar = new QToolBar(tr("Bearbeiten"), this);
    cutAct->addTo(editToolBar);
    copyAct->addTo(editToolBar);
    pasteAct->addTo(editToolBar);
    editToolBar->addSeparator();
    findAct->addTo(editToolBar);
    goToCellAct->addTo(editToolBar);
}
```

Für die Spreadsheet-Anwendung erzeugen wir eine DATEI-Werkzeugleiste und eine BEARBEITEN-Werkzeugleiste. Werkzeugleisten können wie Popup-Menüs Trennstriche enthalten.

Abbildung 3.6: Die Werkzeugleisten der Spreadsheet-Anwendung

Zur Vervollständigung der Benutzeroberfläche erzeugen wir noch ein Kontextmenü:

```
void MainWindow::contextMenuEvent(QContextMenuEvent *event)
{
    QPopupMenu contextMenu(this);
```

```
        cutAct->addTo(&contextMenu);
        copyAct->addTo(&contextMenu);
        pasteAct->addTo(&contextMenu);
        contextMenu.exec(event->globalPos());
    }
```

Wenn der Benutzer die rechte Maustaste anklickt (oder – sofern vorhanden – die Kontextmenütaste drückt), wird ein »context menu«-Ereignis an das Widget gesendet. Durch Reimplementierung der Funktion `QWidget::contextMenuEvent()` können wir auf dieses Ereignis reagieren und ein Kontextmenü an der aktuellen Mauscursorposition einblenden lassen.

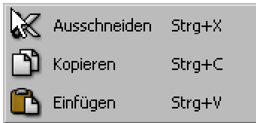


Abbildung 3.7: Das Kontextmenü der Spreadsheet-Anwendung

Neben Signalen und Slots sind *Ereignisse* ein wesentliches Konzept der Qt-Programmierung. Ereignisse werden vom Qt-Kernel als Reaktion auf Mausklicks, Tastaturbetätigungen, Größenänderungen und Ähnlichem erzeugt. Ereignisse können durch Reimplementierung von virtuellen Funktionen abgefangen und bearbeitet werden – was wir im Folgenden tun wollen.

In unserem Beispiel implementieren wir das Kontextmenü in `MainWindow`, da hier die Aktionen gespeichert sind. Es wäre jedoch auch möglich gewesen, das Kontextmenü in `Spreadsheet` zu implementieren. Wenn der Benutzer das `Spreadsheet`-Widget mit der rechten Maustaste anklickt, sendet Qt das Kontextmenü-Ereignis zuerst an dieses Widget. Wenn `Spreadsheet` die Funktion `contextMenuEvent()` reimplementiert und das Ereignis verarbeitet, ist die Ereignisbehandlung damit beendet; ansonsten wird das Ereignis an das übergeordnete Element weitergereicht (das `MainWindow`). Ereignisse werden in ***Kapitel 7*** ausführlich besprochen.

Der Ereignis-Handler für das Kontextmenü unterscheidet sich von dem bisher gezeigten Code darin, dass er ein Widget (ein `QPopupMenu`) als Variable auf dem Stack erzeugt. Wir hätten allerdings genauso gut auch `new` und `delete` verwenden können:

```
QPopupMenu *contextMenu = new QPopupMenu(this);
cutAct->addTo(contextMenu);
copyAct->addTo(contextMenu);
pasteAct->addTo(contextMenu);
contextMenu->exec(event->globalPos());
delete contextMenu;
```

Ebenfalls bemerkenswert an diesem Code ist der `exec()`-Aufruf. `QPopupMenu::exec()` blendet das Popup-Menü an einer gegebenen Bildschirmposition ein und wartet mit der Rückkehr, bis der Benutzer einen Befehl ausgewählt hat (oder das Popup-Menü

ausblenden lässt). Zu diesem Zeitpunkt hat das `QPopupMenu`-Objekt bereits seinen Zweck erfüllt und kann deshalb aufgelöst werden. Wenn das `QPopupMenu`-Objekt sich auf dem Stack befindet, wird es am Ende der Funktion automatisch aufgelöst. Im anderen Fall müssen wir `delete` aufrufen.

Damit ist die Bearbeitung der Menüs und Werkzeugleisten, soweit es die Benutzerschnittstelle angeht, abgeschlossen. Was noch fehlt ist die Implementierung etlicher Slots und der Code für die Anzeige der zuletzt geöffneten Dateien im DATEI-Menü. Das soll Thema der nächsten zwei Abschnitte sein.

3.3 Implementierung des Datei-Menüs

In diesem Abschnitt werden wir die Slots und `private`-Funktionen implementieren, die dafür sorgen, dass die Befehle im Menü DATEI die Ihnen zugedachten Aufgaben erfüllen.

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

Der Slot `newFile()` wird aufgerufen, wenn der Benutzer den Menübefehl DATEI/NEU auswählt oder den NEU-Schalter in der Werkzeugleiste anklickt. Die `private`-Funktion `maybeSave()` fragt den Benutzer, ob er seine Änderungen speichern möchte, für den Fall, dass es noch ungespeicherte Änderungen gibt. Die Funktion liefert `true` zurück, wenn der Benutzer entweder JA oder NEIN anklickt (wobei JA das Dokument speichert), und `false`, wenn der Benutzer den Vorgang abbricht. Die `private` Funktion `setCurrentFile()` aktualisiert den Titel des Fensters, um anzuzeigen, dass ein unbenanntes Dokument bearbeitet wird.

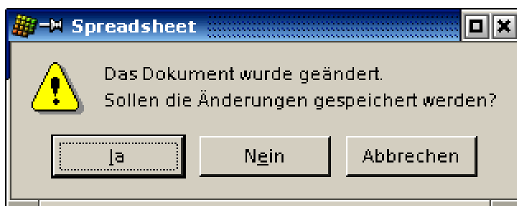


Abbildung 3.8: »Sollen die Änderungen gespeichert werden?«

```
bool MainWindow::maybeSave()
{
    if (modified) {
        int ret = QMessageBox::warning(this, tr("Spreadsheet"),
                     tr("Das Dokument wurde geändert.\n"
                        "Sollen die Änderungen gespeichert werden?"),
                     QMessageBox::Yes | QMessageBox::Default,
                     QMessageBox::No,
                     QMessageBox::Cancel | QMessageBox::Escape);
        if (ret == QMessageBox::Yes)
            return save();
        else if (ret == QMessageBox::Cancel)
            return false;
    }
    return true;
}
```

In `maybeSave()` zeigen wir das Meldungsfenster aus Abbildung 3.8 an. Das Meldungsfenster verfügt über die Schalter JA, NEIN und ABBRECHEN[3]. Der Modifizierer `QMessageBox::Default` legt fest, dass der JA-Schalter standardmäßig aktiviert ist. Der Modifizierer `QMessageBox::Escape` sorgt dafür, dass die Esc-Taste alternativ zum NEIN-Schalter verwendet werden kann.

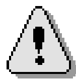
Der Aufruf von `warning()` mag auf den ersten Blick ein wenig kompliziert anmuten, doch die allgemeine Syntax ist ganz einfach:

```
QMessageBox::warning(eltern, titel, meldungstext, schalter0,  schalter1, ...)
```

`QMessageBox` verfügt weiterhin über die Funktionen `information()`, `question()` und `critical()`, die das Verhalten von `warning()` aufweisen, jedoch ein anderes Symbol anzeigen.

Information

Frage


Warnung

Kritisch

Abbildung 3.9: Symbole für das Meldungsfenster

3 Anm.d.Übers.: Die von Qt vordefinierten Dialoge und Meldungsfenster sind ausnahmslos englischsprachig. Die Schalter des hier besprochenen Meldungsfensters tragen demnach per Voreinstellung die Schaltertitel »Yes«, »No« und »Cancel«. Um die vordefinierten Qt-Dialoge vollständig zu lokalisieren, müssen Sie die Übersetzungsdatei *qt-de.qm* aus dem translations-Verzeichnis von Qt in das Verzeichnis der Anwendung kopieren und die `main()`-Funktion um Code ergänzen, der die Übersetzungsdatei dynamisch lädt (siehe Readme-Datei im *spreadsheet*-Verzeichnis auf der Buch-CD). Ausführlichere Informationen zur Lokalisierung von Anwendungen finden Sie in Kapitel 15.

```
void MainWindow::open()
{
    if (maybeSave()) {
        QString fileName =
                QFileDialog::getOpenFileName(".", fileFilters, this);
        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}
```

Der `open()`-Slot gehört zu dem Menübefehl DATEI/ÖFFNEN. Wie `newFile()` ruft er zuerst `maybeSave()` auf, um den Benutzer entscheiden zu lassen, was mit etwaigen nicht gespeicherten Änderungen geschehen soll. Anschließend ruft er die statische Hilfsfunktion `QFileDialog::getOpenFileName()` auf, um den Dateinamen abzufragen. Die Funktion öffnet einen Dialog, in dem der Benutzer eine Datei auswählen kann, und liefert den Dateinamen zurück – beziehungsweise einen leeren String, wenn der Benutzer auf ABBRECHEN klickt.

Die Funktion `getOpenFileName()` übernimmt drei Argumente. Das erste Argument teilt der Funktion mit, in welchem Verzeichnis zuerst gesucht werden soll (in unserem Fall das aktuelle Verzeichnis). Das zweite Argument `fileFilters` gibt die Dateifilter an. Ein Dateifilter besteht aus einem erläuternden Text und einem Suchmuster. Im `MainWindow`-Konstruktor wurde `fileFilters` wie folgt initialisiert:

```
    fileFilters = tr("Spreadsheet-Dateien (*.sp)");
```

Wollten wir neben dem ureigenen Dateiformat der Spreadsheet-Anwendung auch Dateien unterstützen, die von Lotus 1-2-3 stammen oder deren Daten durch Kommata getrennt sind (csv-Dateien), hätten wir die Variable wie folgt initialisiert:

```
fileFilters = tr("Spreadsheet-Dateien (*.sp)\n"
                 "CSV-Textdateien (*.csv)\n"
                 "Lotus 1-2-3-Dateien(*.wk?)");
```

Das dritte Argument zu `getOpenFileName()` gibt schließlich an, dass der eingeblendete `QFileDialog` ein Kind des Hauptfensters sein soll.

Die Eltern/Kind-Beziehung hat für Dialoge eine andere Bedeutung als für Widgets. Ein Dialog ist immer ein Toplevel-Widget (ein eigenständiges Fenster). Besitzt der Dialog ein übergeordnetes Element, wird er mittig über seinem Elternelement angezeigt und teilt sich mit diesem einen Eintrag in der Taskleiste.

```
void MainWindow::loadFile(const QString &fileName)
{
    if (spreadsheet->readFile(fileName)) {
        setCurrentFile(fileName);
```

```
            statusBar()->message(tr("Datei geladen"), 2000);
        } else {
            statusBar()->message(tr("Ladevorgang abgebrochen"), 2000);
        }
    }
```

Die `private`-Funktion `loadFile()` wurde in `open()` aufgerufen, um die Datei zu laden. Sie ist bewusst als eigene Funktion konzipiert, da wir ihre Funktionalität auch zum Laden der zuletzt geöffneten Dateien benötigen.

Mit `Spreadsheet::readFile()` wird die Datei von der Festplatte gelesen. Wenn das Laden erfolgreich war, rufen wir `setCurrentFile()` auf, um den Titel des Fensters zu aktualisieren. Andernfalls informiert `Spreadsheet::loadFile()` den Benutzer mittels eines Meldungsfenster über das Problem. Grundsätzlich empfiehlt es sich, die Fehlermeldungen von den Komponenten der unteren Ebene anzeigen zu lassen, da diese genauere Informationen über Art und Ursache des Fehlers liefern können.

In beiden Fällen geben wir für 2000 Millisekunden (2 Sekunden) eine Meldung in der Statuszeile aus, um den Benutzer darüber zu informieren, was die Anwendung gerade macht.

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

void MainWindow::saveFile(const QString &fileName)
{
    if (spreadsheet->writeFile(fileName)) {
        setCurrentFile(fileName);
        statusBar()->message(tr("Datei gespeichert"), 2000);
    } else {
        statusBar()->message(tr("Speichervorgang abgebrochen"), 2000);
    }
}
```

Der `save()`-Slot gehört zu dem Menübefehl DATEI/SPEICHERN. Wenn die Datei bereits einen Namen hat, weil sie zuvor geöffnet oder bereits gespeichert wurde, ruft `save()` die Funktion `saveFile()` mit diesem Namen auf. Andernfalls wird `saveAs()` aufgerufen.

```
bool MainWindow::saveAs()
{
    QString fileName =
            QFileDialog::getSaveFileName(".", fileFilters, this);
    if (fileName.isEmpty())
        return false;

    if (QFile::exists(fileName)) {
        int ret = QMessageBox::warning(this, tr("Spreadsheet"),
                     tr("Die Datei %1 besteht bereits.\n"
                        "Möchten Sie sie ersetzen?")
                     .arg(QDir::convertSeparators(fileName)),
                     QMessageBox::Yes | QMessageBox::Default,
                     QMessageBox::No | QMessageBox::Escape);
        if (ret == QMessageBox::No)
            return true;
    }
    if (!fileName.isEmpty())
        saveFile(fileName);
    return true;
}
```

Der `saveAs()`-Slot gehört zu dem Befehl DATEI/SPEICHERN UNTER. Wir rufen `QFileDialog::getSaveFileName()` auf, um einen Dateinamen vom Benutzer abzufragen. Wenn der Benutzer auf ABBRECHEN klickt, wird `false` zurückgeliefert und bis zu `maybeSave()` weitergereicht. Andernfalls ist der zurückgelieferte Dateiname entweder ein neuer Name oder der Name einer bereits existierenden Datei. Im letzteren Fall rufen wir `QMessageBox::warning()` auf, um das Meldungsfenster aus Abbildung 3.10 anzuzeigen.

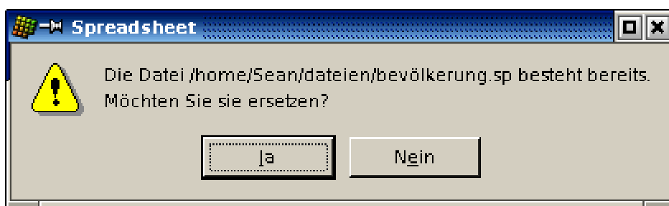


Abbildung 3.10: »Möchten Sie sie ersetzen?«

Der Text, der dem Meldungsfenster übergeben wird, lautet

```
tr("Die Datei %1 besteht bereits.\n"
    "Möchten Sie sie ersetzen?")
    .arg(QDir::convertSeparators(fileName))
```

Die Funktion QString::arg() ersetzt den »*%1*«-Platzhalter durch das für ihn übergebene Argument und liefert den resultierenden String zurück. Lautet der Dateiname beispielsweise *A:\tab04.sp*, wäre obiger Code – vorausgesetzt, die Anwendung wird nicht in andere Sprachen übersetzt – äquivalent zu:

```
"Die Datei A:\\tab04.sp besteht bereits.\n"
"Möchten Sie sie ersetzen?"
```

Der Aufruf von QDir::convertSeparators() wandelt alle Schrägstriche, die Qt als portierbares Verzeichnistrennzeichen verwendet, in das für die jeweilige Plattform typische Trennzeichen um ('/' für Unix und Mac OS X, '\' für Windows).

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

Wenn der Benutzer auf DATEI/BEENDEN oder das X in der Titelleiste des Fensters klickt, wird der Slot QWidget::close() aufgerufen. Damit wird ein »close«-Ereignis an das Widget gesendet. Durch Reimplementierung von QWidget::closeEvent() können wir Versuche, das Hauptfenster zu schließen, abfangen und entscheiden, ob das Fenster tatsächlich geschlossen werden soll oder nicht.

Gibt es Änderungen, die noch nicht gespeichert wurden, und der Benutzer klickt auf ABBRECHEN, »ignorieren« wir das Ereignis und verlassen das Fenster unverändert. Andernfalls akzeptieren wir das Ereignis, was dazu führt, dass Qt das Fenster schließt und die Anwendung beendet wird.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    modLabel->clear();
    modified = false;

    if (curFile.isEmpty()) {
        setCaption(tr("Spreadsheet"));
    } else {
        setCaption(tr("%1 - %2").arg(strippedName(curFile))
                                .arg(tr("Spreadsheet")));
        recentFiles.remove(curFile);
        recentFiles.push_front(curFile);
```

```
            updateRecentFileItems();
        }
    }
```

In `setCurrentFile()` setzen wir die private-Variable `curFile`, die den Namen der bearbeiteten Datei speichert, löschen die Statusanzeige ÄND und aktualisieren den Titel. Beachten Sie, dass `arg()` mit zwei »%*n*«-Parametern verwendet wird. Der erste `arg()`-Aufruf ersetzt »%1«, der zweite »%2«. Es wäre leichter gewesen,

```
setCaption(strippedName(curFile) + tr(" - Spreadsheet"));
```

zu schreiben, aber der Einsatz von `arg()` lässt dem Übersetzer mehr Spielraum. Mit `strippedName()` wird die Pfadangabe der besseren Lesbarkeit wegen aus dem Dateinamen entfernt.

Gibt es einen Dateinamen, aktualisieren wir `recentFiles`, die Liste der zuletzt geöffneten Dateien der Anwendung. Mit dem Aufruf von `remove()` entfernen wir alle Vorkommen des Dateinamens aus der Liste. Anschließend rufen wir `push_front()` auf, um den Dateinamen als erstes Element in die Liste einzufügen. Wir müssen `remove()` vorab aufrufen, um Doppeleintragungen zu vermeiden. Nach der Aktualisierung der Liste rufen wir die `private`-Funktion `updateRecentFileItems()` auf, um die Einträge im DATEI-Menü zu aktualisieren.

Die Variable `recentFiles` ist vom Typ `QStringList` (Liste von `QString`-Objekten). ***Kapitel 11*** beschäftigt sich näher mit Container-Klassen wie `QStringList` und erläutert die Beziehung zur C++-STL-Bibliothek.

Damit ist die Implementierung des DATEI-Menüs fast abgeschlossen. Es gibt nur noch eine Funktion und einen unterstützenden Slot, die wir noch nicht implementiert haben. Diesen beiden obliegt die Verwaltung der Liste der zuletzt geöffneten Dateien.

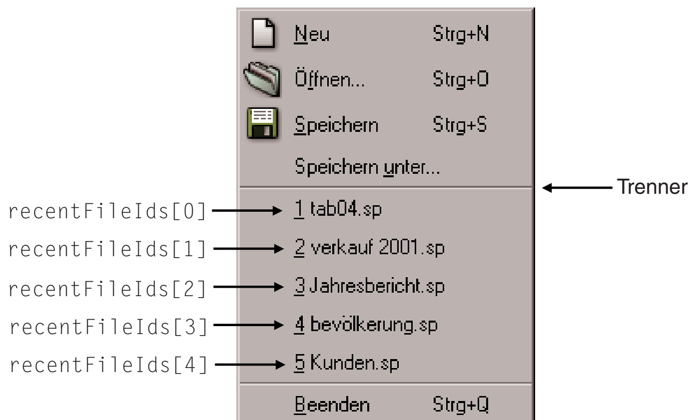


Abbildung 3.11: Datei-Menü mit der Liste der zuletzt geöffneten Dateien

```
void MainWindow::updateRecentFileItems()
{
    while ((int)recentFiles.size() > MaxRecentFiles)
        recentFiles.pop_back();

    for (int i = 0; i < (int)recentFiles.size(); ++i) {
        QString text = tr("&%1 %2")
                       .arg(i + 1)
                       .arg(strippedName(recentFiles[i]));
        if (recentFileIds[i] == -1) {
            if (i == 0)
                fileMenu->insertSeparator(fileMenu->count() - 2);
            recentFileIds[i] =
                    fileMenu->insertItem(text, this,
                                         SLOT(openRecentFile(int)),
                                         0, -1,
                                         fileMenu->count() - 2);
            fileMenu->setItemParameter(recentFileIds[i], i);
        } else {
            fileMenu->changeItem(recentFileIds[i], text);
        }
    }
}
```

Die `private`-Funktion `updateRecentFileItems()` wird aufgerufen, um die Menüeinträge für die zuletzt geöffneten Dateien zu aktualisieren. Zuerst wird sichergestellt, dass nicht mehr Elemente als zulässig in der `recentFiles`-Liste stehen (`MaxRecentFiles`, in *mainwindow.h* als 5 definiert), überzählige Elemente werden vom Ende der Liste entfernt.

Danach erzeugen wir für jeden Eintrag entweder einen neuen Menüeintrag oder verwenden, falls vorhanden, einen bereits vorhandenen Eintrag. Das erste Mal, wenn wir einen Menüeintrag erzeugen, fügen wir zudem eine Trennlinie ein. Wir tun dies hier und nicht in `createMenus()`, um sicherzustellen, dass niemals zwei Trennlinien aufeinander folgen. Auf `setItemParameter()` gehen wir gleich noch näher ein.

Es mag etwas seltsam anmuten, dass wir in `updateRecentFileItems()` immer nur Elemente erzeugen, aber nie löschen. Dies liegt daran, dass wir davon ausgehen können, dass die Liste der zuletzt geöffneten Dateien während einer Sitzung niemals kleiner wird.

Die von uns aufgerufene Funktion `QPopupMenu::insertItem()` hat die folgende Syntax:

```
fileMenu->insertItem(text, empfänger, slot, tastaturkürzel, id, index);
```

Hinter dem Parameter *text* verbirgt sich der im Menü angezeigte Text. Mit `strippedName()` entfernen wir die Pfadangaben aus dem Dateinamen. Man könnte auch die vol-

len Dateinamen beibehalten, doch würde dies das DATEI-Menü sehr breit machen. Wer volle Dateinamen vorzieht, sollte die zuletzt geöffneten Dateien am besten in einem Untermenü anzeigen lassen.

Die Parameter *`empfänger`* und *`slot`* geben den Slot an, der aufgerufen werden soll, wenn der Benutzer das Element auswählt. In unserem Beispiel stellen wir eine Verbindung zu dem `openRecentFile(int)`-Slot von `MainWindow` her.

Für *`tastaturkürzel`* und *`id`* übergeben wir Standardwerte, die bewirken, dass der Menüeintrag kein Tastaturkürzel hat und seine ID automatisch erzeugt wird. Die solchermaßen erzeugten IDs werden in dem `recentFileIds`-Array gespeichert, sodass wir später auf die Elemente zugreifen können.

Der *`index`*-Parameter bestimmt die Position, an der der Eintrag eingefügt werden soll. Durch Übergabe von `fileMenu->count()-2` fügen wir den Eintrag über der Trennlinie des BEENDEN-Befehls ein.

```
void MainWindow::openRecentFile(int param)
{
    if (maybeSave())
        loadFile(recentFiles[param]);
}
```

Der `openRecentFile()`-Slot wird aufgerufen, wenn eine zuletzt geöffnete Datei aus dem DATEI-Menü ausgewählt wird. Der `int`-Parameter ist der von uns zuvor mit `setItemParameter()` gesetzte Wert. Wir haben die Werte klugerweise so gewählt, dass sie direkt als Indizes in die `recentFiles`-Liste verwendet werden können.

Menüeinträge

ID	Text	Param
-32	1 tab04.sp	0
-33	2 verkauf2001.sp	1
-34	3 Jahresbericht.sp	2
-35	4 Bevölkerung.sp	3
-36	5 Kunden.sp	4

Zuletzt geöffnete Dateien

Index	Wert
0	A:\tab04.sp
1	C:\Verkauf2001.sp
2	D:\Jahresbericht.sp
3	C:\Bevölkerung.sp
4	C:\Customers.sp

Abbildung 3.12: Verwaltung der zuletzt geöffneten Dateien

Eine etwas weniger elegante Lösung wäre gewesen, fünf Aktionen zu erzeugen und diese mit fünf separaten Slots zu verbinden.

3.4 Statuszeile einrichten

Nachdem die Menüs und Werkzeugleisten fertig sind, können wir dazu übergehen, die Statuszeile der Spreadsheet-Anwendung einzurichten. Im Normalzustand ist die Statuszeile in drei Anzeigefelder (Indikatoren) unterteilt: die aktuelle Zellposition, die Formel der aktuellen Zelle und die ÄND-Anzeige. Daneben wird die Statuszeile aber auch dazu verwendet, Statuszeilenhinweise und andere vorübergehende Nachrichten anzuzeigen.

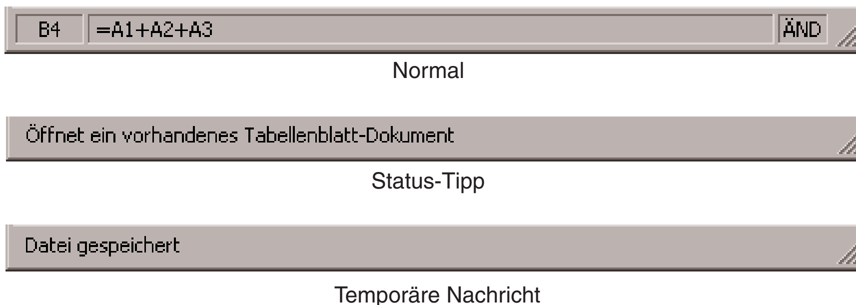


Abbildung 3.13: Die Statuszeile der Spreadsheet-Anwendung

Der `MainWindow`-Konstruktor richtet die Statuszeile mit dem Aufruf von `createStatusBar()` ein:

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

    formulaLabel = new QLabel(this);

    modLabel = new QLabel(tr(" ÄND "), this);
    modLabel->setAlignment(AlignHCenter);
    modLabel->setMinimumSize(modLabel->sizeHint());
    modLabel->clear();

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

    connect(spreadsheet, SIGNAL(currentChanged(int, int)),
            this, SLOT(updateCellIndicators()));
    connect(spreadsheet, SIGNAL(modified()),
```

```
            this, SLOT(spreadsheetModified()));

    updateCellIndicators();
}
```

Die Funktion `QMainWindow::statusBar()` liefert einen Zeiger auf die Statuszeile zurück. (Die Statuszeile wird erzeugt, wenn `statusBar()` das erste Mal aufgerufen wird). Die Statusanzeigen sind einfach `QLabel`-Objekte, deren Text wir im Bedarfsfall ändern. Bei der Erzeugung der `QLabel`-Objekte übergeben wir `this` als Elternelement, obwohl dies eigentlich keinen Unterschied macht, da `QStatusBar::addWidget()` automatisch dafür Sorge trägt, dass die `QLabel`-Objekte Kinder der Statuszeile werden.

Abbildung 3.13 zeigt, dass die drei Anzeigen unterschiedlichen Platzbedarf haben. Die Anzeige für die Zellposition und die ÄND-Anzeige benötigen im Gegensatz zur mittleren Anzeige nur wenig Platz. Zusätzlicher Platz, der entsteht, wenn das Fenster vergrößert wird, sollte daher der mittleren Anzeige mit der Zellenformel zukommen. Wir erreichen dies, indem wir im Aufruf von `QStatusBar::addWidget()` einen Streckungsfaktor von 1 angeben. Die anderen beiden Anzeigen haben den voreingestellten Streckungsfaktor von 0, was bedeutet, dass möglichst keine Streckung erfolgen soll.

Wenn `QStatusBar` die Anzeige-Widgets anordnet, versucht es, die für jedes Widget von `QWidget::sizeHint()` vorgegebene ideale Größe zu berücksichtigen und mit den zu streckenden Widgets den restlichen verfügbaren Raum zu füllen. Die ideale Größe eines Widgets hängt vom Inhalt des Widgets ab und ändert sich, wenn wir den Inhalt ändern. Um ständige Größenänderungen der beiden Anzeigen für Zellposition und ÄND zu unterbinden, setzen wir deren Minimalgröße so, dass jede Anzeige den für sie größtmöglichen Text aufnehmen kann (»W999« und »ÄND«), plus ein wenig Zugabe. Außerdem zentrieren wir die beiden Texte mit `AlignHCenter` horizontal.

Gegen Ende der Funktion verbinden wir zwei der Signale von `Spreadsheet` mit zwei der Slots von `MainWindow`: `updateCellIndicators()` und `spreadsheetModified()`.

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

Der `updateCellIndicators()`-Slot aktualisiert die Anzeigen für Zellposition und Zellenformel. Er wird immer dann aufgerufen, wenn der Benutzer den Zellencursor auf eine neue Zelle setzt. Der Slot wird am Ende von `createStatusBar()` auch als normale Funktion verwendet, um die Anzeigen zu initialisieren. Dies ist erforderlich, da `Spreadsheet` beim Start kein `currentChanged()`-Signal aussendet.

```
void MainWindow::spreadsheetModified()
{
    modLabel->setText(tr("ÄND"));
    modified = true;
    updateCellIndicators();
}
```

Der `spreadsheetModified()`-Slot aktualisiert alle drei Anzeigen, sodass sie den aktuellen Zustand widerspiegeln, und setzt die Variable `modified` auf `true`. (Wir haben die `modified`-Variable bei der Implementierung des DATEI-Menüs verwendet, um festzustellen, ob es ungespeicherte Änderungen gibt).

3.5 Dialoge

Dieser Abschnitt beschreibt, wie in Qt Dialoge eingesetzt werden – wie man sie erzeugt und initialisiert, sie ausführt und auf die vom Benutzer vorgenommenen Einstellungen und Eingaben reagiert. Dabei greifen wir auf die Dialoge SUCHEN, GEHE ZU ZELLE und SORTIEREN zurück, die wir bereits in ***Kapitel 2*** erstellt haben. Zusätzlich wird ein einfacher INFO-Dialog erstellt.

Beginnen wir mit dem SUCHEN-Dialog. Da der Benutzer die Freiheit haben soll, beliebig zwischen dem Spreadsheet-Hauptfenster und dem SUCHEN-Dialog hin und her zu wechseln, muss der SUCHEN-Dialog als nicht-modaler Dialog ausgelegt sein. *Nicht-modale* Fenster werden unabhängig von allen anderen Fenstern in der Anwendung ausgeführt.

Wenn nicht-modale Dialoge erzeugt werden, werden ihre Signale üblicherweise mit Slots verbunden, die auf die Interaktionen der Benutzer reagieren.

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

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

Der SUCHEN-Dialog ist ein Fenster, mit dem der Benutzer nach einer Textstelle im aktuellen Tabellenblatt suchen kann. Der `find()`-Slot wird ausgeführt, wenn der Benutzer den Befehl BEARBEITEN/SUCHEN zum Öffnen des SUCHEN-Dialogs aufruft. An diesem Punkt sind mehrere Szenarien möglich:

- Der Benutzer ruft den SUCHEN-Dialog das erste Mal auf.
- Der SUCHEN-Dialog wurde schon einmal geöffnet, ist danach aber wieder geschlossen worden.
- Der SUCHEN-Dialog wurde bereits geöffnet und ist noch sichtbar.

Wenn der SUCHEN-Dialog noch nicht existiert, erzeugen wir ihn und verbinden seine `findNext()`- und `findPrev()`-Signale mit den entsprechenden Slots von `Spreadsheet`. Wir hätten den Dialog auch im `MainWindow`-Konstruktor erzeugen können, hätten dadurch aber den Start der Anwendung unnötig verzögert. Die nachträgliche Erzeugung belastet den Start nicht und kann zudem Zeit und Speicherplatz sparen, da der Dialog, wenn er nicht benötigt wird, auch nicht erzeugt wird.

Anschließend rufen wir `show()`, `raise()` und `setActiveWindow()` auf, um sicherzustellen, dass das Fenster zu sehen ist, über den anderen Fenster liegt und aktiviert wird. An sich genügt ein `show()`-Aufruf, um ein verborgenes Fenster sichtbar zu machen. Der SUCHEN-Dialog kann aber auch aufgerufen werden, wenn sein Fenster bereits sichtbar ist. In diesem Fall würde der `show()`-Aufruf allein nichts bewirken. Um das Dialogfenster unabhängig von seinem aktuellen Status anzuzeigen, zu aktivieren und in den Vordergrund zu heben, müssen wir zusätzlich `raise()` und `setActiveWindow()` aufrufen. Die Alternative wäre, den Code wie folgt aufzubauen:

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

Genauso gut könnten Sie aber auch 90 km/h in einer Tempo 100-Zone fahren.

Kommen wir jetzt zum Dialog GEHE ZU ZELLE. Dieser Dialog soll vom Benutzer geöffnet, bearbeitet und wieder geschlossen werden, ohne dass der Benutzer die Möglichkeit hat, zwischen dem GEHE ZU ZELLE-Dialog und einem anderen Fenster der Anwendung hin und her zu wechseln, d.h. der GEHE ZU ZELLE-Dialog soll modal sein. Ein *modales* Fenster ist ein Fenster, das, wenn aufgerufen, die Anwendung blockiert und solange jegliche Interaktionen mit anderen Teilen der Anwendung unterbindet, bis es wieder geschlossen wird. Mit Ausnahme des SUCHEN-Dialogs waren alle bisher verwendeten Dialoge modal.

Ein Dialog, der mit `show()` aufgerufen wird, ist automatisch nicht-modal (es sei denn, wir rufen vorher `setModal()` auf, um ihn zu einem modalen Dialog zu machen). Ein Dialog, der mit `exec()` aufgerufen wird, ist modal. Für modale Dialoge, die mit `exec()` aufgerufen werden, müssen in der Regeln keine Signal/Slot-Verbindungen eingerichtet werden.

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

Die Funktion `QDialog::exec()` liefert `true` zurück, wenn der Dialog mit OK beendet wird, andernfalls lautet der Rückgabewert `false`. (Zur Erinnerung: Als wir den GEHE ZU ZELLE-Dialog in ***Kapitel 2*** mit dem *Qt Designer* erzeugt haben, wurde OK mit `accept()` und ABBRECHEN mit `reject()` verbunden.) Wenn also der Benutzer auf OK drückt, machen wir die im LineEdit-Eingabefeld spezifizierte Zelle zur aktuellen Zelle. Wenn der Benutzer auf ABBRECHEN drückt, liefert `exec()` den Wert `false` zurück und wir machen nichts.

Die Funktion `QTable::setCurrentCell()` übernimmt zwei Argumente: einen Zeilen- und einen Spaltenindex. In der Spreadsheet-Anwendung ist die Zelle A1 die Zelle (0, 0) und die Zelle B27 die Zelle (26, 1). Um den Zeilenindex aus dem von `QLabel::text()` zurückgelieferten `QString`-Objekt zu ermitteln, extrahieren wir die Zeilennummer mit `QString::mid()` (liefert den Teilstring von der angegebenen Startposition bis zum Ende des Strings), konvertieren das Ergebnis mit `QString::toInt()` in einen Integer und subtrahieren 1, um einen Wert auf Null-Basis zu erhalten. Für den Spaltenwert subtrahieren wir den numerischen Wert von 'A' von dem numerischen Wert des ersten groß geschriebenen Zeichens im String.

Im Gegensatz zum SUCHEN-Dialog wird der GEHE ZU ZELLE-Dialog auf dem Stack erzeugt. Für modale Dialoge ist dies, ebenso wie für Kontextmenüs, gängige Programmierpraxis, da die Dialoge nach dem Schließen nicht weiter benötigt werden.

Wenden wir uns jetzt dem SORTIEREN-Dialog zu. Der SORTIEREN-Dialog ist ein modaler Dialog, mit dem der Benutzer einen Bereich nach verschiedenen Spalten sortieren kann. Abbildung 3.14 zeigt ein Beispiel für eine Sortierung, bei der als primärer Sortierschlüssel die Spalte B und als sekundärer Sortierschlüssel die Spalte A (beide aufsteigend) verwendet wurden.

	A	B	C	D
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	D
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: Einen ausgewählten Bereich der Tabelle sortieren

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableSelection sel = spreadsheet->selection();
    dialog.setColumnRange('A' + sel.leftCol(), 'A' + sel.rightCol());

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

Der Code in `sort()` folgt einem ähnlichen Muster wie der Code in `goToCell()`:

- Der Dialog wird auf dem Stack erzeugt und initialisiert.
- Der Dialog wird mit `exec()` angezeigt.
- Wenn der Benutzer auf OK klickt, werden die vom Benutzer eingegebenen Werte aus den Widgets extrahiert und ausgewertet.

Das `compare`-Objekt speichert die primären, sekundären und tertiären Sortierschlüssel samt Sortierreihenfolgen. (Die Definition der Klasse `SpreadsheetCompare` wird im nächsten Kapitel vorgestellt). Das Objekt wird von `Spreadsheet::sort()` verwendet, um je zwei Zeilen zu vergleichen. Das `keys`-Array speichert die Spaltennummern der

Schlüssel. Wenn sich zum Beispiel der ausgewählte Bereich von C2 bis E5 erstreckt, hat Spalte C die Position 0. Das `ascending`-Array speichert die mit jedem Schlüssel verbundene Sortierreihenfolge als booleschen Wert. `QComboBox::currentItem()` liefert den Index des aktuell ausgewählten Elements, beginnend bei 0, zurück. Für den sekundären und tertiären Schlüssel vermindern wir den Index jeweils um 1, um dem »Keine«-Element Rechnung zu tragen.

Die `sort()`-Funktion erfüllt ihren Zweck, steht aber auf wackeligen Füßen. Es wird vorausgesetzt, dass der SORTIEREN-Dialog in einer bestimmten Weise implementiert ist, d.h. mit Comboboxen und »Keine«-Elementen. Dies wiederum hat zur Folge, dass wir bei Änderungen am Sortieren-DIALOG auch unseren Code überarbeiten müssen. Dieser Ansatz mag zwar für Dialoge zweckmäßig sein, die nur von einer Stelle aus aufgerufen werden, wenn aber die Dialoge von mehreren Stellen aus verwendet werden, kann dies für alle, die mit der Wartung des Codes betraut sind, der Anfang eines Albtraums sein.

Robuster wäre der Ansatz, die `SortDialog`-Klasse intelligenter zu machen, indem man sie das `SpreadsheetCompare`-Objekt selbst erzeugen lässt, auf das dann vom aufrufenden Code zugegriffen werden kann. Das würde `MainWindow::sort()` erheblich vereinfachen:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableSelection sel = spreadsheet->selection();
    dialog.setColumnRange('A' + sel.leftCol(), 'A' + sel.rightCol());
    if (dialog.exec())
       spreadsheet->performSort(dialog.comparisonObject());
}
```

Dieser Ansatz führt zu lose miteinander verbundenen Komponenten und ist fast immer die richtige Wahl für Dialoge, die von mehr als einer Stelle aus aufgerufen werden.

Ein noch radikalerer Ansatz wäre es, bei der Initialisierung des `SortDialog`-Objekts einen Zeiger auf das `Spreadsheet`-Objekt zu übergeben und dem Dialog die Möglichkeit einzuräumen, direkt auf dem `Spreadsheet`-Objekt zu operieren. Der `SortDialog` wäre dann zwar weniger allgemein, da er nur mit einem bestimmten Widget-Typ zusammenarbeitet, doch dafür würde der Code durch Wegfall des `SortDialog::setColumnRange()`-Aufrufs noch kürzer. Die Funktion `MainWindow::sort()` lautete dann:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
```

```
        dialog.setSpreadsheet(spreadsheet);
        dialog.exec();
    }
```

Dieser Ansatz ist das Spiegelbild des ersten: Nicht mehr der aufrufende Code muss genaue Kenntnis des Dialogs haben, sondern der Dialog muss genaue Kenntnis der vom aufrufenden Code bereitgestellten Datenstrukturen haben. Dieser Ansatz dürfte vor allem für Fälle geeignet sein, wo der Dialog Änderungen direkt vornehmen muss. Doch genauso anfällig, wie der aufrufende Code im ersten Ansatz ist, so unsicher ist der dritte Ansatz, wenn sich die Datenstrukturen ändern.

Manche Entwickler entscheiden sich irgendwann für einen Ansatz und verwenden diesen dann für alle ihre Dialoge. Vertrautheit und Einfachheit sind die Pluspunkte dieser Vorgehensweise, doch verzichtet man dadurch auf die Vorteile der anderen Ansätze. Die Entscheidung für einen bestimmten Ansatz sollte immer von dem jeweiligen Dialog abhängig gemacht werden.

Dieses Kapitel soll mit einem einfachen INFO-Dialog abgerundet werden. Hierzu könnten wir wie beim SUCHEN- oder GEHE ZU ZELLE-Dialog einen eigenen Dialog erzeugen, der die Informationen über unsere Anwendung anzeigt, doch da die meisten INFO-Dialoge vom Stil her sehr aufwändig sind, bietet Qt eine einfachere Lösung an.

```
    void MainWindow::about()
    {
        QMessageBox::about(this, tr("Info über Spreadsheet"),
                tr("<h2>Spreadsheet 1.0</h2>"
                   "<p>Copyright &copy; 2003 Software Inc."
                   "<p>Spreadsheet ist eine kleine Anwendung, die "
                   "den Einsatz von <b>QAction</b>, <b>QMainWindow</b>, "
                   "<b>QMenuBar</b>, <b>QStatusBar</b>, "
                   "<b>QToolBar</b> und vielen weiteren Qt-Klassen demonstriert."));
    }
```

Sie erzeugen den INFO-Dialog, indem Sie `QMessageBox::about()` – eine statische Hilfsfunktion – aufrufen. Diese Funktion ist der Funktion `QMessageBox::warning()` sehr ähnlich, nur dass hier das Symbol des Elternfensters statt des »Achtung«-Symbols verwendet wird.

Für `QMessageBox` und `QFileDialog` haben wir bisher immer die statischen Hilfsfunktionen aufgerufen, die einen Dialog erzeugen, initialisieren und ausführen. Es ist allerdings auch möglich – wenn auch wesentlich unbequemer – ein `QMessageBox`- oder ein `QFileDialog`-Widget wie ein normales Widget zu erzeugen und dann explizit `exec()` oder sogar `show()` darauf aufzurufen.

Abbildung 3.15: Der Info-Dialog der Spreadsheet-Anwendung

3.6 Einstellungen speichern

Im `MainWindow`-Konstruktor haben wir `readSettings()` aufgerufen, um die gespeicherten Anwendungseinstellungen zu laden. Als Pendant dazu haben wir in `closeEvent()` die Funktion `writeSettings()` aufgerufen, um die Einstellungen zu speichern. Diese beiden Funktionen sind die letzten `MainWindow`-Member-Funktionen, die implementiert werden müssen.

Der Ansatz, für den wir uns in `MainWindow` entschieden haben, mit all seinem `QSettings`-bezogenen Code in `readSettings()` und `writeSettings()`, ist nur einer von vielen. Mithilfe eines `QSettings`-Objekts kann man zu jedem Zeitpunkt und von überall im Code Einstellungen abfragen oder ändern.

```
void MainWindow::writeSettings()
{
    QSettings settings;
    settings.setPath("software-inc.com", "Spreadsheet");
    settings.beginGroup("/Spreadsheet");
    settings.writeEntry("/geometry/x", x());
    settings.writeEntry("/geometry/y", y());
    settings.writeEntry("/geometry/width", width());
    settings.writeEntry("/geometry/height", height());
    settings.writeEntry("/recentFiles", recentFiles);
    settings.writeEntry("/showGrid", showGridAct->isOn());
    settings.writeEntry("/autoRecalc", showGridAct->isOn());
    settings.endGroup();
}
```

Die Funktion `writeSettings()` speichert die geometrischen Daten des Hauptfensters (Position und Größe), die Liste der zuletzt geöffneten Dateien und die Optionen GITTER ANZEIGEN und AUTOMATISCH BERECHNEN.

`QSettings` speichert die Anwendungseinstellungen an einem plattformspezifischen Ort. Unter Windows ist dies die Systemregistrierung; unter Unix werden die Daten in Textdateien gespeichert und unter Mac OS X greift `QSettings` auf die Carbon Preferences API zurück. Der Aufruf von `setPath()` versorgt `QSettings` mit dem Namen der Organisation (als Internet-Domänenname) und dem Produktnamen. Diese Informationen werden plattformspezifisch verarbeitet, um einen Speicherort für die Einstellungen zu finden.

`QSettings` speichert die Einstellungen als *Schlüssel/Wert*-Paare. Der *Schlüssel* gleicht einem Dateisystempfad und sollte immer mit dem Namen der Anwendung beginnen. So sind zum Beispiel `/Spreadsheet/geometry/x` und `/Spreadsheet/showGrid` gültige Schlüssel. (Der Aufruf von `beginGroup()` erspart uns, den Anwendungsnamen vor jeden Schlüssel zu setzen.) Der *Wert* kann vom Typ `int`, `bool`, `double`, `QString` oder `QStringList` sein.

```
void MainWindow::readSettings()
{
    QSettings settings;
    settings.setPath("software-inc.com", "Spreadsheet");
    settings.beginGroup("/Spreadsheet");

    int x = settings.readNumEntry("/geometry/x", 200);
    int y = settings.readNumEntry("/geometry/y", 200);
    int w = settings.readNumEntry("/geometry/width", 400);
    int h = settings.readNumEntry("/geometry/height", 400);
    move(x, y);
    resize(w, h);

    recentFiles = settings.readListEntry("/recentFiles");
    updateRecentFileItems();

    showGridAct->setOn(
            settings.readBoolEntry("/showGrid", true));
    autoRecalcAct->setOn(
            settings.readBoolEntry("/autoRecalc", true));

    settings.endGroup();
}
```

Die Funktion `readSettings()` lädt die Einstellungen, die mit `writeSettings()` gespeichert wurden. Das zweite Argument an die Lesefunktionen gibt einen Standardwert vor, für den Fall, dass keine Einstellungen verfügbar sind. Die Standardwerte werden bei der ersten Ausführung der Anwendung verwendet.

Damit wäre die Implementierung von `MainWindow` abgeschlossen. Die folgenden Abschnitte beschreiben, wie man die Spreadsheet-Anwendung modifizieren könnte, um die Bearbeitung mehrerer Dokumente zu unterstützen oder einen Eröffnungsbildschirm (Splash-Screen) anzuzeigen. Die Anwendung selbst wird dann in *Kapitel 4* fertig gestellt.

3.7 Mehrere Dokumente

Wir sind inzwischen soweit, den Code für die `main()`-Funktion der Spreadsheet-Anwendung aufzusetzen:

```
#include <qapplication.h>

#include "mainwindow.h"

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

Diese `main()`-Funktion unterscheidet sich ein wenig von denen, die Sie bisher gesehen haben: Anstatt `new` zu verwenden, wird hier die `MainWindow`-Instanz als Variable auf dem Stack erzeugt. Folglich wird die `MainWindow`-Instanz automatisch aufgelöst, wenn die Funktion endet.

Mit der oben gezeigten `main()`-Funktion verfügt die Spreadsheet-Anwendung über nur ein Hauptfenster und kann immer nur ein Dokument bearbeiten. Wenn der Benutzer mehrere Dokumente gleichzeitig bearbeiten möchte, muss er mehrere Instanzen der Spreadsheet-Anwendung starten. Schöner wäre es, eine Instanz der Anwendung zu haben, die mehrere Hauptfenster bereitstellt – so wie eine Instanz eines Webbrowsers mehrere Browserfenster gleichzeitig verwalten kann.

Wir werden die Spreadsheet-Anwendung nun dahin gehend ändern, dass sie mehrere Dokumente verwalten kann. Zuerst benötigen wir ein leicht verändertes DATEI-Menü:

Abbildung 3.16: Das neue Datei-Menü

- DATEI/NEU erzeugt ein neues Hauptfenster mit einem leeren Dokument (anstatt das aktuelle Hauptfenster wiederzuverwenden).

- DATEI/SCHLIESSEN schließt das aktuelle Hauptfenster.
- DATEI/BEENDEN schließt alle Fenster.

Im alten DATEI-Menü gab es den Menübefehl SCHLIESSEN nicht, da er identisch mit BEENDEN gewesen wäre.

Und so sieht die neue `main()`-Funktion aus:

```
#include <qapplication.h>
#include "mainwindow.h"

int main(int argc,  char *argv[])
{
    QApplication app(argc,  argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    QObject::connect(&app, SIGNAL(lastWindowClosed()),
                     &app, SLOT(quit()));
    return app.exec();
}
```

Wir verbinden den `lastWindowClosed()`-Slot von `QApplication` mit dem `quit()`-Slot von `QApplication`, was die Anwendung beendet.

Werden mehrere Fenster verwaltet, empfiehlt es sich, `MainWindow` mit `new` zu erzeugen. Das Hauptfenster kann dann, wenn es nicht mehr benötigt wird, mit `delete` gelöscht werden – was Speicher spart. Für Anwendungen mit nur einem Hauptfenster sind solche Betrachtungen natürlich überflüssig.

Der neue `MainWindow::newFile()`-Slot lautet:

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

Wir erzeugen einfach eine neue `MainWindow`-Instanz. Ein wenig verwunderlich an diesem Code mag sein, dass wir den Zeiger auf das neue Fenster nicht dauerhaft sichern, um jederzeit auf das Hauptfenster zugreifen zu können. Dies ist aber nicht nötig, da Qt bereits alle Fenster für uns verwaltet.

Die Aktionen für SCHLIESSEN und BEENDEN lauten:

```
closeAct = new QAction(tr("S&chließen"), tr("Ctrl+W"), this);
connect(closeAct, SIGNAL(activated()), this, SLOT(close()));
exitAct = new QAction(tr("&Beenden"), tr("Ctrl+Q"), this);
connect(exitAct, SIGNAL(activated()), qApp, SLOT(closeAllWindows()));
```

Der `closeAllWindows()`-Slot von `QApplication` schließt alle Fenster der Anwendung, es sei denn eines davon lehnt das Schließen-Ereignis ab. Dies ist genau das von uns benötigte Verhalten. Wir müssen uns also keine weiteren Gedanken über ungespeicherte Änderungen machen, da sich bereits `MainWindow::closeEvent()` um alles kümmert, wann immer ein Fenster geschlossen wird.

Es scheint, als ob unsere Anwendung damit jetzt in der Lage wäre, mehrere Fenster zu unterstützen. Es gibt allerdings noch ein Problem, das nicht so direkt offensichtlich ist und im Hintergrund lauert: Wenn der Benutzer immerzu Fenster erzeugt und schließt, kann es passieren, dass dem Rechner der Speicherplatz ausgeht! Das liegt daran, dass wir in `newFile()` immerzu `MainWindow`-Widgets erzeugen, aber nie löschen. Wenn der Benutzer ein Hauptfenster schließt, wird dieses Fenster zwar verborgen, verbleibt aber im Speicher. Werden viele Hauptfenster erzeugt, kann dies zu einem Problem werden.

Die Lösung besteht darin, dem Konstruktor das `WDestructiveClose`-Flag hinzuzufügen:

```
Mainwindow::Mainwindow(QWidget *parent, const char *name)
      : QMainWindow(parent, name, WDestructiveClose) {
     ...
}
```

Damit wird Qt angewiesen, das Fenster beim Schließen zu löschen. Das `WDestructiveClose`-Flag ist eines von vielen, das dem `QWidget`-Konstruktor übergeben werden kann, um das Verhalten des Widgets zu beeinflussen. Die meisten dieser Flags werden allerdings nur selten in Qt-Anwendungen benötigt.

Speicherlecks sind nicht das einzigen Problem, mit dem wir zu tun haben. Der ursprüngliche Entwurf für unsere Anwendung ging von der Annahme aus, dass es nur ein Hauptfenster gibt. Gibt es mehrere Fenster, verfügt jedes Hauptfenster über seine eigene Liste der zuletzt geöffneten Dateien und über seine eigenen Optionen. Klar ist, dass die Liste der zuletzt geöffneten Dateien für die ganze Anwendung global sein sollte. Dies lässt sich ganz einfach erreichen, indem wir die Variable `recentFiles` als `static` deklarieren, sodass nur eine Instanz davon für die ganze Anwendung existiert. Wir müssen dann allerdings auch sicherstellen, dass an allen Stellen, wo wir `updateRecentFileItems()` aufgerufen haben, um das DATEI-Menü zu aktualisieren, die Funktion nunmehr für alle Hauptfenster aufgerufen wird. Das erreichen wir mit folgendem Code:

```
QWidgetList *list = QApplication::topLevelWidgets();
QWidgetListIt it(*list);
QWidget *widget;
while ((widget = it.current())) {
    if (widget->inherits("MainWindow"))
       ((MainWindow *)widget)->updateRecentFileItems();
    ++it;
}
delete list;
```

Der Code durchläuft alle Toplevel-Widgets der Anwendung und ruft `updateRecentFileItems()` auf jedem Widget vom Typ `MainWindow` auf. Ein ähnlicher Code kann für die Synchronisierung der Optionen GITTER ANZEIGEN und AUTOMATISCH BERECHNEN verwendet werden oder um sicherzustellen, dass dieselbe Datei nicht zwei Mal geladen wird. Der Typ `QWidgetList` ist ein `typedef`-Synonym für `QPtrList<Widget>`, siehe ***Kapitel 11*** zu den Container-Klassen.

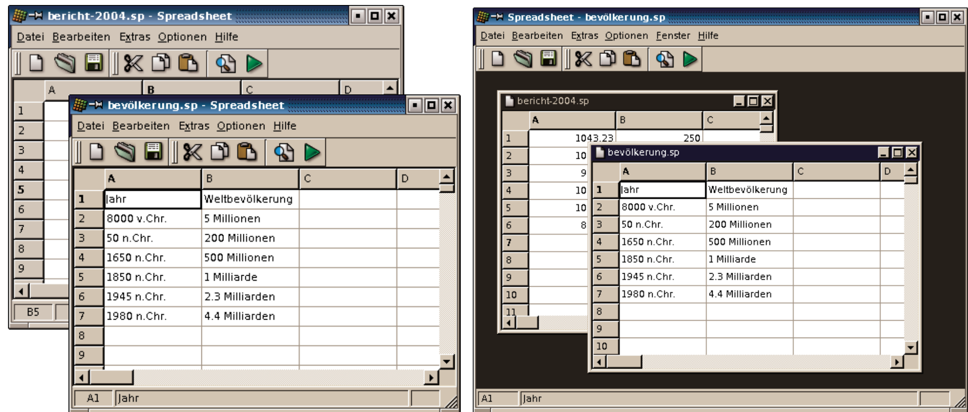


Abbildung 3.17: SDI versus MDI

Anwendungen, die nur ein Dokument pro Hauptfenster anzeigen, werden auch SDI-Anwendungen genannt (Single Document Interface). Eine weit verbreitete Alternative ist MDI (Multiple Document Interface), bei der ein einziges Hauptfenster innerhalb seines zentralen Bereichs mehrere Dokumentfenster verwaltet. Mit Qt können Sie für alle unterstützten Plattformen sowohl SDI- als auch MDI-Anwendungen erstellen. Abbildung 3.17 zeigt die Spreadsheet-Anwendung in beiden Varianten. MDI wird in ***Kapitel 6*** (Layout-Management) noch ausführlicher besprochen.

3.8 Startbildschirme (Splash-Screens)

Viele Anwendungen zeigen eingangs einen Startbildschirm (Splash-Screen) an. Manche Entwickler nutzen Startbildschirme, um davon abzulenken, dass das Starten des Programms übermäßig lang dauert, andere versuchen damit ihre Marketing-Abteilungen zufrieden zu stellen. Mit der `QSplashScreen`-Klasse ist das Hinzufügen von Startbildschirmen zu Qt-Anwendungen ein Kinderspiel.

Mit der Klasse `QSplashScreen` können Sie ein Bild einblenden, das so lange angezeigt wird, bis die Anwendung endgültig gestartet wird. `QSplashScreen` kann auch eine Meldung in das Bild zeichnen, um den Benutzer über den Fortgang des Anwendungsstart zu informieren. Üblicherweise wird der Code für den Splash-Screen in `main()` untergebracht, vor dem Aufruf von `QApplication::exec()`.

Nachfolgend sehen Sie ein Beispiel für eine `main()`-Funktion, die `QSplashScreen` verwendet, um einen Startbildschirm für eine Anwendung anzuzeigen, die beim Start verschiedene Module lädt und eine Netzwerkverbindung herstellt.

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

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

    splash->message(QObject::tr("Hauptfenster wird eingerichtet..."),
                    Qt::AlignRight | Qt::AlignTop, Qt::white);
    MainWindow mainWin;
    app.setMainWidget(&mainWin);

    splash->message(QObject::tr("Module werden geladen..."),
                    Qt::AlignRight | Qt::AlignTop, Qt::white);
    loadModules();

    splash->message(QObject::tr("Verbindungen werden hergestellt..."),
                    Qt::AlignRight | Qt::AlignTop, Qt::white);
    establishConnections();

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

    return app.exec();
}
```

Abbildung 3.18: Ein QSplashScreen-Widget

Damit ist die Benutzerschnittstelle unserer Spreadsheet-Anwendung vollständig. Im nächsten Kapitel werden wir die Kernfunktionalität der Spreadsheet-Anwendung implementieren.