1

Programming GUIs using Qt

1.1 An introductory example

To get a small feel for how one programs a GUI using `qtbase`, the R package that interfaces R with the Qt libraries, we show how to produce a simple dialog to collect a date from a user.

If the underlying libraries and package are installed, the package is loaded as any other R package:

```
require(qtbase)
```

Constructors As with all other toolkits, in QtGUI components are created with constructors. For this example, we will set various properties later, rather than at construction time. For our GUI we have four basic widgets: a widget used as a container to hold the others, a label, a single-line edit area and a button.

```
w <- Qt$QWidget()
l <- Qt$QLabel()
e <- Qt$QLineEdit()
b <- Qt$QPushButton()
```

The constructors are not found in the global environment, but rather are found in the `Qt` environment provided through `qtbase`. As such, the `$` lookup operator is used. For this example, we use a `QWidget` as a top-level window, leaving for Section 2.6 to discuss the `QMainWidget` object and its task-tailored features.

Widgets in Qt have various properties that set the state of the object. For example, the widget object, `w`, has the `windowTitle` property that is adjusted as follows:

```
w$windowTitle <- "An example"
```

Qt objects are essentially environments. In the above, the named component `windowTitle` of the environment holds the value of the `windowTitle` property of the object, so the `$` use is simply that for environments.

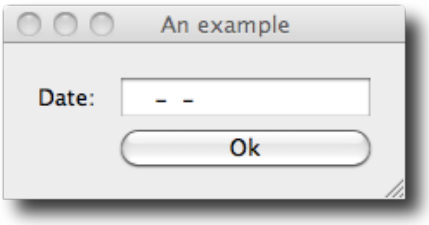

Figure 1.1: Screenshot of our sample GUI to collect a date from the user.

A more typical use is a method call. Qt overloads the `$` operator for method calls (as does `RGtk2`). For example, both the button object and label object have a `text` property. The setter `setText` can be used to assign a value. For example,

```
l$setText("Date:")
b$setText("Ok")
```

Although, the calling mechanism is more complicated than just the lookup of a function stored as the component `setText`, as the object is passed into the body of the function, the usage is similar.

Layout Managers Qt uses layout managers to organize widgets. This is similar to Java/Swing and `tcltk`, but not `RGtk2`. Layout managers will be discussed more thoroughly in Chapter 2, but in this example we will use a grid layout to organize our widgets. The placement of child widgets into the grid is done through the `addWidget` method and requires a specification, by index and span, of the cells the child will occupy.

```
lyt <- Qt$QGridLayout()
lyt$addWidget(l, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(e, 0, 1, 1, 1)
lyt$addWidget(b, 1, 1, 1, 1)
```

One can adjust properties of the layout, but we leave that discussion for later.

We need to attach our layout to the widget `w`, which is done through the `setLayout` method:

```
w$setLayout(lyt)
```

Then to view our GUI (Figure 1.1), we call the widget's `show` method.

```
w$show()
```

Callbacks As with outher GUI toolkits, we add interactivity to our GUI by binding callbacks to certain events. To add a command to the clicking or

pressing of the button is done by attaching a handler to the "pressed" signal for the button (the "clicked" signal is only for mouse clicks). Widgets have various signals they emit. Additionally, there are window-manager events that may be of interest, but using them requires more work than is shown below. The `qconnect` function is used to add a handler for a signal. The function needs, at a minimum, the object, the signal name and the handler. Herr we print the value stored in the "Date" field.

```
handler <- function(checked)  print(e$text)
id <- qconnect(b, "pressed", handler)
```

We will discuss callbacks more completely in Section 1.6.

Refinements At this point, we have a working dialog built with `qtbase`. There is much room for refinement, which due to Qt's many features are relatively easy to implement. For this example, we want to guide the user to fill out the date in the proper format. We could have used Qt's `QDateEdit` widget to allow point-and-click selection, but instead show two ways to help the user fill in the information with the keyboard

The `QLineEdit` widget has a number of ways to adjust its behavior. For example, an input mask provides a pattern for the user to fill out. For a date, we may want the value to be in the form "year-month-date." This would be specified with `"0000-00-00"`, as seen by consulting the API for `QLineEdit`. To add an input mask we have:

```
e$setInputMask("0000-00-00")
```

Further, for the line edit widget in Qt we can easily implement validation of the entered text. There are a few built-in validators, and for this purpose the regular expression validator could be used, but instead we wish to determine if we have a valid date by seeing if we can coerce the string value to a date via R's `as.Date` function with a format of `"%Y-%m-%d"`. The method `setValidator` can be used to set the validator that is in charge of the validation. However, rather than passing a function, one must pass an instance of a validator class. For our specific needs, we need to create a new class.

Object-oriented support The underlying Qt libraries are written in `C++`. The object oriented nature is preserved by `qtbase`. Not only are Qt's classes and methods implemented in R, the ability to implement new subclasses and methods is also possible. For the validation task, we need to implement a subclass of the `QValidator` class, and for this subclass implement a `validate` method. More detail on working with classes and methods in `qtbase` is provided in Section 1.8.

The `qsetClass` function is used to set a new class. To derive a subclass, we need just this:

```
qsetClass("dateValidator", Qt$QValidator)
```

The `validate` method is implemented as a virtual class in Qt, in R we implement it as a method of our subclass using the `qsetMethod` function. The signature of the `validate` method is a string containing the input and an index indicating where the cursor is in the text box. The return value of this method indicates a state of "Acceptable", "Invalid", or if neither can be determined "Intermediate." Qt uses enumerations (cf. Section 1.7) to specify these values. The actual values are integers, but are kept as components of the environments provided by `qtbase` so can be referenced by name. In this case, the enumeration is in the `Qt$QValidator` class.

```
## FIXME: should probably use an input mask to restrict the input to
## 0000-00-00, then use as.Date() here to validate the date.
qsetMethod("validate", dateValidator, function(input, pos) {
  if(!grepl("^[0-9]{4}-[0-9]{1,2}-[0-9]{1,2}$", input))
    return(Qt$QValidator$Intermediate)
  else  if(is.na(as.Date(input, format="%Y-%m-%d")))
    return(Qt$QValidator$Invalid)
  else
    return(Qt$QValidator$Acceptable)
})
```

To use this new class, we call its constructor, which has the same name as the class, and then set it as a validator for the line edit widget:

```
validator <- dateValidator()
e$setValidator(validator)
```

1.2 The Qt library

Qt is an open-sourced, cross-platform application framework that is perhaps best known for its widget toolkit. The features of Qt are divided into about a dozen modules. We highlight some of the more important and interesting ones:

Core Basic utilties, collections, threads, I/O, ...
Gui Widgets, models, etc for graphical user interfaces
OpenGL Convenience layer (e.g., 2D drawing API) over OpenGL
Webkit Embeddable HTML renderer (shared with Safari, Chrome)

Other modules include functionality for networking, XML, SQL databases, SVG, and multimedia. However, R packages already provide many of those features.

The history of Qt begins with Haavard Nord and Eirik Chambe-Eng in 1991, and follows with the Trolltech company until 2008, and now Nokia, a major cell-phone producer. While it was originally not available as open-source on every platform, as of version 4 it was released universally under the GPL. With the release of Qt 4.5, Nokia placed Qt under the LGPL, so it is available for use in proprietary software, as well. Popular software developed

with Qt include the communication application Skype and the KDE desktop for Linux.

Qt is developed in C++ with extensions that require a special preprocessor called the *Meta Object Compiler* (`MOC`). The `MOC` allows for convenient syntax in the definition of signals, slots (signal handlers), and properties, which behave very similarly to those of GTK+.

There are many languages with bindings to Qt, and R is one such language. The `qtbase` package interfaces with every module of Qt. As its name suggests, `qtbase` forms the base for a number of R packages that provide high-level special-purpose interfaces to Qt. As these are still early in development, they will not be mentioned here.

While `qtbase` is not yet as mature as `tcltk` and `RGtk2`, we include it in this book, as Qt compares favorably to GTK+ in terms of GUI features and excels in several areas, including its fast graphics canvas and integration of the WebKit web browser. [1] In addition, Qt, as a commercially supported package, has thorough documentation of its API, including many C++ examples. However, the complexity of C++ and Qt may present some challenges to the R user. In particular, the developer should have a strong grounding in object-oriented programming and have a basic understanding of memory management.

The `qtbase` package is developed as part of the *qtinterfaces* project at R Forge: `https://r-forge.r-project.org/projects/qtinterfaces/`. It depends on the Qt framework, available as a binary install from `http://qt.nokia.com/`.

1.3 Classes and objects

The `qtbase` package exports very few objects. The central one is an environment, `Qt`, that represents the Qt library in R. [2] The components of this environment are `RQtClass` objects that represent an actual C++ class or namespace. For example, the `QWidget` class is represented by `Qt$QWidget`:

```
Qt$QWidget
```

```
Class 'QWidget' with 320 public methods
```

An `RQtClass` object contains methods in the class scope (*static* methods in C++), enumerations defined by the class, and additional `RQtClass` objects representing nested classes or namespaces. Here we list some of the components of `QWidget` and access one of the enumeration values:

```
head(names(Qt$QWidget), n = 3)
```

[1] There is a GTK+ WebKit port, but it is not included with GTK+ itself.

[2] The `Qt` object is an instance of `RQtLibrary`. The `qtbase` package provides infrastructure for binding any conventional C++ library, even those independent of Qt. Third party packages can define their own `RQtLibrary` object for some other library.

```
[1] "connect"              "DrawChildren"         "DrawWindowBackground"
```

```
Qt$QWidget$DrawChildren
```

```
Enum value: DrawChildren (2)
```

Most importantly, however, an instance of `RQtClass` is in fact an R `function` object, and serves as the constructor of instances of the class. For example, we could construct an instance of `QWidget`:

```
w <- Qt$QWidget()
```

The `w` object has a class structure that reflects the class inheritance structure of Qt:

```
class(w)
```

```
[1] "QWidget"             "QObject"             "QPaintDevice"
[4] "UserDefinedDatabase" "environment"         "RQtObject"
```

The base class, `RQtObject`, is an environment containing the properties and methods of the instance. For `w`, we list the first few using `ls`:

```
head(ls(w))
```

```
[1] "setShown"                 "releaseShortcut"
[3] "actions"                  "setAccessibleDescription"
[5] "isWindowModified"         "size"
```

Properties and methods are accessed from the environment in the usual manner. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window:

```
w$windowTitle                              # initially NULL
```

```
NULL
```

```
w$windowTitle <- "a new title"
w$windowTitle
```

```
[1] "a new title"
```

Although Qt defines methods for accessing properties, the R user will normally invoke methods that perform some action. For example, we could show our widget:

```
w$show()
```

The environment structure of the object masks the fact that the properties and methods may be defined in a parent class of the object. For example, a button widget is provided by the `QPushButton` constructor, as in

```
b <- Qt$QPushButton()
```

QPushButton extends QWidget and thus inherits the properties like windowTitle:

```
is(b, "QWidget")                          # Yes
```

```
[1] TRUE
```

```
b$windowTitle
```

```
NULL
```

It is important to realize this distinction when referencing the documentation. As with GTK+, the methods are documented with the class that declares the method.

1.4 Methods and dispatch

In C++, it is possible to have multiple methods and constructors with the same name, but different signatures. This is called *overloading*. An overloaded method is roughly similar to an S4 generic, save the obvious difference that an S4 generic does not belong to any class. The selected overload is that with the signature that best matches the types of the arguments. The exact rules of overload resolution are beyond our scope.

It is particularly common to overload constructors. For example, a simple push button can be constructed in several different ways. Here again is the invocation of the QPushButton constructor with no arguments:

```
b <- Qt$QPushButton()
```

By convention, all classes derived from QObject, including QWidget, provide a constructor that accepts a parent QObject. This has important consequences that are discussed later. We demonstrate this for QPushButton:

```
w <- Qt$QWidget()
b <- Qt$QPushButton(w)
```

An alternative constructor for QPushButton accepts the text for the label on the button:

```
b <- Qt$QPushButton("Button text")
```

Buttons may also have icons, for example

```
icon <- Qt$QIcon(system.file("images/ok.gif", package="gWidgets"))
b <- Qt$QPushButton(icon, "Ok")
```

We have passed three different types of object as the first argument to Qt$QPushButton: a QWidget, a string, and finally a QIcon. The dispatch depends only on the

type of argument, unlike the constructors in `RGtk2`, which dispatch based on which arguments are specified. (In particular, dispatch is based on position of argument, but not on names given to arguments. We use names only for clarity in our examples.)

1.5 Properties

Every `QObject`, which includes every widget, may declare a set of properties that represent its state. We list some of the available properties for our button:

```
head(qproperties(b))
```

	type	readable	writable
objectName	QString	TRUE	TRUE
modal	bool	TRUE	FALSE
windowModality	Qt::WindowModality	TRUE	TRUE
enabled	bool	TRUE	TRUE
geometry	QRect	TRUE	TRUE
frameGeometry	QRect	TRUE	FALSE

As shown in the table, every property has a type and logical settings for whether the property is readable and/or writeable. Virtually every property value may be read, and it is common for properties to be read-only. For example, we can fully manipulate the `objectName` property, but our attempt to modify the `modal` property fails:

```
b$objectName <- "My button"
b$objectName
```

```
[1] "My button"
```

```
b$modal
```

```
[1] FALSE
```

```
try(b$modal <- TRUE)
```

Qt provides accessor methods for getting and setting properties. The getter methods have the same name as the property, so they are masked at the R level. Setter methods are available and are typically named with the word "set" followed by the property name:

```
b$setObjectName("My button")
```

However, it is recommended to use the replacement syntax shown in the previous example, for the sake of symmetry.

1.6 Signals

Qt in C++ uses an architecture of signals and slots to have components communicate with each other. A component emits a signal when some event happens, such as a user clicking on a button. Qt allows one to define a special type of method known as a slot in another component (or the same) that can be connected to the signal as the handler. The two components are decoupled as the emitter does not need to know about the receiver except through the signal connection. In R, any function can be treated as a slot and connected as a signal handler. This is similar to the signal handling in `RGtk2`. The function `qconnect` establishes the connection of an R function to a signal. For example

```
b <- Qt$QPushButton("click me")
qconnect(b, "clicked", function() print("ouch"))
b$show()
```

Signals are defined by a class and are inherited by subclasses. Here, we list some of the available signals for the `QPushButton` class:

```
tail(qsignals(Qt$QPushButton))
```

```
                          name                                signature
3 customContextMenuRequested customContextMenuRequested(QPoint)
4                      pressed                              pressed()
5                     released                             released()
6                      clicked                           clicked(bool)
7                      clicked                              clicked()
8                      toggled                           toggled(bool)
```

The signal definition specifies the callback signature, given in the `signature` column. Like other methods, signals can be overloaded so that there are multiple signatures for a given signal name. Signals can also have default arguments, and arguments with a default value are optional in the signal handler. We see this for the `clicked` signal, where the `bool` (logical) argument, indicating whether the button is checked, has a default value of `FALSE`. The `clicked` signal is automatically overloaded with a signature without any arguments.

The `qconnect` function attempts to pick the correct signature by considering the number of formal arguments in the callback. When two signatures have the same number of arguments, one will be chosen arbitrarily. To connect to a specific signature, the full signature, rather than only the name, should be passed to qconnect. For example, the following is equivalent to the invocation of `qconnect` above:

```
qconnect(b, "clicked(bool)", function(checked) print("ouch"))
```

Any object passed to the optional argument `user.data` is passed as the last argument to the signal handler. The user data serves to parameterize

the callback. In particular, it can be used to pass in a reference to the sender object itself.

The `qconnect` function returns a dummy `QObject` instance that provides the slot that wraps the R function. This dummy object can be used with the `disconnect` method on the sender to break the signal connection:

```
proxy <- qconnect(b, "clicked", function() print("ouch"))
b$disconnect(proxy)
```

```
[1] TRUE
```

One can block all signals from being emitted with the `blockSignals` method, which takes a logical value to toggle whether the signals should be blocked.

Unlike GTK+, Qt widgets generally do not emit hardware events, such as a mouse press, via signals. Instead, a method in the widget is invoked upon receipt of an event. The developer is expected to extend the widget and override the method to catch the event. The apparent philosophy of Qt is that hardware events are low-level and thus should be handled by the widget, not some other instance. We will discuss extending classes in Section **??**. Example 1.2 demonstrates handling widget events.

1.7 Enumerations and flags

Often, it is useful to have discrete variables with more than two states, in which case a logical value is no longer sufficient. For example, the label widget has a property for how its text is aligned. It supports the alignment styles left, right, center, top, bottom, etc. These styles are enumerated by integer values and Qt defines these by name within the relevant class or, for global enumerations, in the `Qt` namespace. Here are examples of both:

```
Qt$Qt$AlignRight
```

```
Enum value: AlignRight (2)
```

```
Qt$QSizePolicy$Exanding
```

```
NULL
```

The first is the value for right alignment from the `Alignment` enumeration in the `Qt` namespace, while the second is from the `Policy` enumeration in the `QSizePolicy` class (referenced here by `QSizePolicy::Policy`).

Although these enumerations can be specifed directly as integers, they are given the class `QtEnum` and have the overloaded operators | and & to combine values bitwise. This makes the most sense when the values correspond to bit flags, as is the case for the alignment style. For example, aligning the text in a label in the upper right can be done through

```
l <- Qt$QLabel("Our text")
l$alignment <- Qt$Qt$AlignRight | Qt$Qt$AlignTop
```

To check if the alignment is to the right, we could query by: [3]

```
as.logical(l$alignment & Qt$Qt$AlignRight)
```

```
[1] FALSE
```

1.8 Defining Classes and Methods

As Qt is implemented in an object-oriented language, C++, the designers of the API expect the developer to extend Qt classes, like `QWidget`, during the normal course of GUI development. This is a significant difference from GTK+, where it is only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbase` package allows the R user to extend C++ classes in order to enhance the features of Qt.

We will demonstrate this functionality by example. In the introductory example of Section 1.1 we saw this minimal use of `qsetClass`. Our aim is to extend the `QValidator` class to restrict the input in a text entry (`QTextEdit`) to a valid date. The first step is to declare the class, and then methods are added individually to the class definition, in an analogous manner to the `methods` package. We start by declaring the class:

```
qsetClass("DateValidator", Qt$QValidator)
```

The class name is given as `DateValidator` and it extends the `QValidator` class in Qt. Note that only single inheritance is supported.

As a side-effect of the call to `qsetClass`, a variable named `DateValidator` has been assigned into the global environment (the scoping is similar to `methods::setClass`):

```
DateValidator
```

```
Class 'R:.GlobalEnv::DateValidator' with 57 public methods
```

It is an error to redefine the class as above without first deleting the object.

To define a method on our class, we call the `qsetMethod` function:

```
qsetMethod("validate", DateValidator, validateDate)
```

```
[1] "validate"
```

The virtual method `validate` declared by `QValidator` has been overriden by the `DateValidator` class. The `validateDate` function implements the override and has been defined invisibly for brevity.

[3] As flags, combinations of enumerations, are not stored as integers they are not of class `QtEnum`, so the algebra of these operators is limited.

As `DateValidator` is an `RQtClass` object, we can create an instance by invoking `DateValidator` like any other `function`:

```
validator <- DateValidator()
```

Now that we have our validator, we can use it with a text entry:

```
e <- Qt$QLineEdit()
v <- DateValidator(e)
e$setValidator(v)
```

```
NULL
```

```
e$show()
```

```
NULL
```

As Qt is implemented in an object-oriented language, `C++`, the designers of the API expect the developer to extend Qt classes, like `QWidget`, during the normal course of GUI development. This is a significant difference from GTK+, where it is only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbase` package allows the R user to extend `C++` classes in order to enhance the features of Qt. The `qtbase` package includes functions `qsetClass` and `qsetMethod` to create subclasses and methods of subclasses.

To create a subclass of some class, say `QWidget`, the basic use is simply

```
qsetClass("SubClass", Qt$QWidget)
```

This assigns to the variable `SubClass` an object inheriting from `RQtClass`. (The `where` argument can be used to specify where this variable is defined, in a manner similar to S4's `setClass`.) Note the lack of parentheses for the parent object (`Qt$QWidget`), as the parent also inherits from `RQtClass`. These objects are constructors (functions) that have augmented environments containing the available methods and properties.

By default, the parent's constructor is inherited by the subclass. To create a new constructor, say to initialize values, or to make a compound widget, the `constructor` argument is used. This takes a function that will be called by positional arguments, so default values are used, but not by name. The function should allow a `parent` argument, to pass along a parent widget. A typical minimal usage might be

```
qsetClass("SubClass2", Qt$QWidget, function(parent=NULL) {
  super(parent)
  ## ... more here
})
```

Within the body of a constructor, the `super` variable refers to the parent's constructor, and in the above is used to set the parent object. Within the body of

a constructor the variable `this` refers to the environment of the instance of the subclass being constructed. To set a property, we can assign to a component. This environment augments the constructor's environment, so one can refer to the methods without a prefix. For example, a simple constructor to pass in a window title and some special property might look like

```
qsetClass("SubClass3", Qt$QWidget, function(title, prop, parent=NULL) {
  super(parent)
  this$property <- prop
  setWindowTitle(title)
})
```

One may define new methods, or override methods from a base class through the `qsetMethod` function. The arguments are the method name, the class the method is defined in, the method's definition, and an optional value for `access` indicating if the method should be `"public"`, `"protected"`, or `"private"`.

Within the body of the method definition the variable `super` is a function, somewhat like `do.call`, that can be used to call a method from the parent class by name (the first argument) passing in all subsequent arguments. The functionality is similar to that provided by S4's `callNextMethod`. The variable `this` refers again to the instance's environment. Again for convenience this is placed into the search path of the function call.

For example, methods to create accessors for a property may be defined as

```
qsetClass("SubClass4", Qt$QWidget)
qsetMethod("property", SubClass4, function() property)
qsetMethod("setProperty", SubClass4, function(value) {
  this$property <- value
})
```

To override a base method by adding to its call can be done with the `super` function. For example, to store the window title in our property one might do this:

```
qsetMethod("setWindowTitle", SubClass4, function(value) {
  setProperty(value)
  super("setWindowTitle", value)
})
```

Example 1.1: A "error label"
A common practice when validation is used for text entry is to have a "buddy label." That is an accompanying label to set an error message. As Qt uses "buddy" for something else, we call this an "error label" below. We show how to implement such a widget in `qtbase` where we define a new type of widget that combines a `QTextEdit` and the `QLabel` for reporting errors.

This demonstrates the definition of a custom constructor for an R class. The R function implementing the constructor must be passed during the call to `qsetClass`:

```
qsetClass("ErrorLabel", Qt$QLineEdit,
          function(text, parent=NULL, message="") {
            super(parent)

            this$widget <- Qt$QWidget()           # for attaching
            this$error <- Qt$QLabel()             # set height=0
            this$error$setStyleSheet("* {color: red}")  # set color

            lyt <- Qt$QGridLayout()               # layout
            widget$setLayout(lyt)

            lyt$setVerticalSpacing(0)
            lyt$addWidget(this, 0, 0, 1, 1)
            lyt$addWidget(error, 1, 0, 1, 1)

            if(nchar(message) > 0)
              setMessage(message)
            else
              setErrorHeight(FALSE)
            if(!missing(text)) setText(text)
          })
```

By convention, every widget in Qt accepts its parent as an argument to its constructor. Via `super`, the parent is passed to the base class constructor and on up the hierarchy. The `super` function does not exist outside the scope of a constructor (or method). Within a constructor, `super` invokes the constructor of the parent (super) class, with the given arguments. Within a method implementation, `super` will call a method (named in the first parameter) in the parent class (see Example 1.2).

In addition to the call to `super`, we define a `QWidget` instance to contain the line edit widget and its label. These are placed within a grid layout. The use of `this` refers to the object we are creating. The new method `setErrorHeight` is used to flatten the height of the label when it is not needed and is defined below. The final line sets the initial text in the line edit widget. The R environment where `setText` is defined is extended by the environment of this constructor, so no prefix is needed in the call.

The `widget` component is needed to actually show the widget. We create an accessor method:

```
qsetMethod("widget", ErrorLabel, function() widget)
```

We extend the API of the line edit widget for this subclass to modify the message. We define three methods, one to get the message, one to set it and a convenience function to clear the message.

```
qsetMethod("message", ErrorLabel, function() error$text)
qsetMethod("setMessage", ErrorLabel, function(msg="") {
  if(nchar(msg) > 0)
    error$setText(msg)
  setErrorHeight(nchar(msg) > 0)
})
qsetMethod("clear", ErrorLabel, function()  setMessage())
```

Finally, we define the method to set the height of the label, so that when there is no message it has no height. We use a combination of setting both the minimum and maximum height.

```
qsetMethod("setErrorHeight", ErrorLabel, function(do.height=FALSE)
{
  if(do.height) {
    m <- 18; M <- 100
  } else {
    m <- 0; M <- 0
  }
  error$setMinimumHeight(m)
  error$setMaximumHeight(m)
})
```

To use this widget, we have the extra call to `widget()` to retrieve the widget to add to a GUI. In the following, we just show the widget.

```
e <- ErrorLabel()
w <- e$widget()                 # get widget to show
w$show()                        # to view widget
e$setMessage("A label")         # opens message
e$clear()                       # clear message, shrink space
```

1.9 Common methods for QWidgets and QObjects

The widgets we discuss in the sequel inherit many properties and methods from the base `QObject` and `QWidget` classes. The `QObject` class is the base class and forms the basis for the object heirarchy and the event processing system. The `QWidget` class is the base class for objects with a user interface. Defined in this class are several methods inherited by the widgets we discuss.

The function `qmethods` will show the methods defined for a class. It returns a data frame with variables indicating the name, return value, signature, and whether the method is protected and static. For example, for a simple button we have many methods, of which we can only discuss a handful.

```
out <- qmethods(Qt$QPushButton)
dim(out)                                # many methods
```

```
[1] 435   5
```

Showing or hiding a widget Widgets must have their `show` method called in order to have them drawn to the screen. This call happens through the `print` method for an object inheriting from `QWidget`, but more typically is called recursively by Qt when showing the children as a top-level window is drawn. The method `raise` will raise the window to the top of the stack of windows, in case it is covered. The method `hide` will hide the widget.

A widget can also be hidden by calling its `setVisible` method with a value of `FALSE` and reshown using a value of `TRUE`. Similary, the method `setEnabled` can be used to toggle whether a widget is sensitive to user input, including mouse events.

Only one widget can have the keyboard focus. This is changed by the user through tab-navigation or mouse clicks (unless customized, see `focusPolicy`), but can be set programatically through the `setFocus` method, and tested through the `hasFocus` method.

Qt has a number of means to notify the user about a widget when the mouse hovers over it. The `setTooltip` method is used to specify a tooltip as a string. The message can be made to appear in the status bar of a top-level window through the method `setStatusTip`.

The size of a widget A widget may be drawn with its own window, or typically embedded in a more complicated GUI. The size of the widget can be adjusted through various methods.

First, we can get the size of the widget through the methods `frameGeometry` and `frameSize`. The `frameGeometry` method returns a `QRect` instance, Qt's rectangle class for integer sizes. Rectangles are parameterized by an $x - y$ position and two dimensions (`x`, `y`, `width` and `height`). In this case, the position refers to the upper left coordinate and dimensions are in pixels. The convenience function `qrect` is provided to construct `QRect` instances. The `frameSize` method returns a `QSize` object with properties `width` and `height`. The `qsize` function is a convenience constructor for objects of this class.

The widget's size can be adjusted several ways: with the `resize` method, by modifying the rectangle and then resetting the geometry with `setGeometry`, or directly through the same method when integer values are given for the arguments.

```
w <- Qt$QWidget()
rect <- w$frameGeometry
rect$width()
rect$setWidth(2 * rect$width())
w$setGeometry(rect)
```

Although the above sets the size, it does not fix it. If that is desired, the methods `setFixedSize` or `setFixedWidth` are available.

When a widget is resized, one can constrain how it changes by specifying a minimum size or maximum size. These values work in combination with the size policy of the widget. The properties `minimumSize`, `minimumWidth`, `minimumHeight`, `maximumSize` `maximumWidth` and `maximumHeight`, and their corresponding setters, are the germane ones. How these get used is determined by the `sizePolicy`roperty. For example, buttons will only grow in the x direction – not the y direction due to their default size policy.

The method `update` is used to request a repainting of a widget and the method `updateGeometry` is used to notify any layout manager that the the size hint has changed.

Fonts

Fonts in Qt are handled through the `QFont` class. In addition to the basic constructor, one constructor allows the programmer to specify a `family`, such as `helvetica`; `pointsize`, an integer; `weight`, an enumerated value such as `Qt$QFont$Light` (or `Normal`, `DemiBold`, `Bold`, or `Black`); and whether the italic version should be used, as a logical.

For example, a typical font specification may be given as follows:

```
f <- Qt$QFont(family="helvetica", ps=12,
              weight=Qt$QFont$Bold,
              italics=TRUE)
```

For widgets, the `setFont` method can be used to adjust the font. For example, to change the font for a label we have

```
l <- Qt$QLabel("Text for the label")
l$setFont(f)
```

The `QFont` class has several methods to query the font and to adjust properties. For example, there are the methods `setFamily`, `setUnderline`, `setStrikeout` and `setBold` among others.

Styles

Qt uses styles to provide a means to customize the look and feel of an application for the underlying operating system. It is generally recommended that a GUI be consistent with the given style. For example, each style implements a palette of colors to indicate the states "active" (has focus), "inactive" (does not have focus), and "disabled" (not sensitive to user input). Although, many widgets do not have a visible distinction between active or inactive. The role an object plays determines the type of coloring it should receive. A palette has an enumeration in `QPalette::ColorRole`. Sample ones are `Qt$QPalette$Highlight`, to indicate a selected item, or `Qt$QPalette$WindowText` to indicate a foreground color.

These roles are used for setting the foreground or background role to give a widget a different look, as illustrated in Example 1.2.

Style Sheets

Cascading style sheets (CSS) are used by web deisgners to decouple the layout and look and feel of a web page from the content of the page. Qt implements the mechanism in the `QWidget` class to customize a widget through the CSS syntax. The implemented syntax is described in the overview on stylesheets provided with Qt documentation and is not summarized here, as it is quite readable.

To implement a change through a style sheet involves the `setStyleSheet` method. For example, to change the background and text color for a button we could have

```
b <- Qt$QPushButton("Style sheet example")
b$show()
b$setStyleSheet("QPushButton {color: red; background: white}")
```

One can also set a background image:

```
ssheet <- sprintf("* {background-image: url(%s)}", "logo.png")
b$setStyleSheet(ssheet)
```

1.10 Drag and drop

Qt has native support for basic drag and drop activities for some of its widgets, such as text editing widgets, but for more complicated situations such support must be programmed in. The toolkit provides a clear interface to allow this.

A drag and drop event consists of several stages: the user selects the object that initiates the drag event, the user then drags the object to a target, and finally a drop event occurs. In addition, several decisions must be made, e.g., will the object "move" or simply be copied. Or, what kind of object will be transferred (an image? text?, ...) etc. The type is specified using a standard MIME specification.

Initiating a drag and drop source When a drag and drop sequence is initiated, the widget receiving the mouse press event needs to set up a `QDrag` instance that will transfer the necessary information to the receiving widget. In addition, the programmer specifies the type of data to be passed, as an instance of the `QMimeData` class. Finally, the user must call the `exec` method with instructions indicating what happens on the drop event (the supported actions) and possibly what happens if no modifier keys are specified. These are given using the enumerations `CopyAction`, `MoveAction`, or `LinkAction`.

This is usually specified in a method for the `mousePressEvent` event, so is done in a subclass of the widget you wish to use.

Creating a drop target The application must also set up drop sources. Each source has its method `setAcceptDrops` called with a `TRUE` value. In addition, one must implement several methods so again, a subclass of the desired widget is needed. Typically one implements at a minimum a `dropEvent` method. This method has an `QDropEvent` instance passed in which has the method `mimeData` to get the data from the `QDrag` instance. This data has several methods for querying the type of data, as illustrated in the example. If everything is fine, one calls the even'ts `acceptProposedAction` method to set the drop action. One can also specify other drop actions.

Additionally, one can implement methods for `dragMoveEvent` and `dragLeaveEvent`. In the example, the move and leave event adjust proporties of the widget to indicate it is a drop target.

Example 1.2: Drag and drop

We will use subclasses of the label class to illustrate how one implements basic drag-and-drop functionality. Out treatment follows the Qt tutorial on the subject. We begin by setting up a label to be a drag target.

```
qsetClass("DragLabel", Qt$QLabel, function(text="", parent=NULL) {
  super(parent)
  setText(text)

  setAlignment(Qt$Qt$AlignCenter)
  setMinimumSize(200, 200)
})
```

The main method to implement for the subclass is `mousePressEvent`. Its argument `e` contains event information for the mouse press event, we don't need it here. We have the minimal structure here: implement mime data to pass through, set up a `QDrag` instance for the data, then call the `exec` method to initiate the drag event. The `exec` method call has optional arguments to specify what action should be done. The default here is move, copy, link, for which only copy makes sense.

```
qsetMethod("mousePressEvent", DragLabel, function(e) {
  md <- Qt$QMimeData()
  md$setText(text)

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)
```

```
  drag$exec()
})
```

```
[1] "mousePressEvent"
```

Implementing a label as a drop target is a bit more work, as we customize its appearance. Our basic constructor follows:

```
qsetClass("DropLabel", Qt$QLabel, function(text="", parent=NULL) {
  super(parent)

  setText(text)
  setAcceptDrops(TRUE)

  this$bgrole <- backgroundRole()
  setAlignment(Qt$Qt$AlignCenter)
  setMinimumSize(200, 200)
  setAutoFillBackground(TRUE)
  clear()
})
```

We wish to override the call to `setText` above, as we want to store the original text in a property of the widget. Note the use of `super` with a method definition below to call the next method.

```
qsetMethod("setText", DropLabel, function(str) {
  this$orig_text <- str
  super("setText", str)                 # next method
})
```

The `clear` method is used to restore the label to an initial state. We have saved the background role and original text as properties for this purpose.

```
qsetMethod("clear", DropLabel, function() {
  setText(this$orig_text)
  setBackgroundRole(this$bgrole)
})
```

The enter event notifies the user that a drop can occur on this target by changing the text and the background role.

```
qsetMethod("dragEnterEvent", DropLabel, function(e) {
  super("setText", "<Drop Text Here>")
  setBackgroundRole(Qt$QPalette$Highlight)

  e$acceptProposedAction()
})
```

The move and leave events are straightforward. We call `clear` when the drag leaves the target to restore the widget.

```
qsetMethod("dragMoveEvent", DropLabel, function(e) {
  e$acceptProposedAction()
})
qsetMethod("dragLeaveEvent", DropLabel, function(e) {
  clear()
#  e$acceptProposedAction()
})
```

Finally, the important drop event. The following shows how to implement this in more generality than is needed for this example, as we only have text in our mime data.

```
qsetMethod("dropEvent", DropLabel, function(e) {
  md <- e$mimeData()

  if(md$hasImage()) {
    setPixmap(md$imageData())
  } else if(md$hasHtml()) {
    setText(md$html)
    setTextFormat(Qt$Qt$RichText)
  } else if(md$hasText()) {
    setText(md$text())
    setTextFormat(Qt$Qt$PlainText)
  } else {
    setText("No match")                 # replace...
  }

  setBackgroundRole(this$bgrole)
  e$acceptProposedAction()
})
```

2

Layout managers

Qt provides a set of classes to faciliate the layout of child widgets of a component. These layout managers, derived from the `QLayout` class, are tasked with determining the geometry of child widgets, according to a specific layout algorithm. Layout managers will generally update the layout whenever a parameter is modified, a child widget is added or removed, or the size of the parent changes. Unlike GTK+, where this management is tied to a container object, Qt decouples the layout from the widget.

In this chapter we discuss the basic layouts and end with a discussion on the `QMainWindow` class for top-level windows.

We begin with an example that shows many of the different types of layouts.

Example 2.1: Using layout managers to mock up an interface

This example illustrates how to layout a somewhat complicated GUI by hand using a combination of different layout managers provided by Qt. Figure 2.1 shows a screenshot of the finished layout.

Qt provides standard layouts for box layouts and grid layouts, in addition there are notebook containers and special layouts for forms, all seen in the following.

Our GUI is layed out from the outside in. The first layout used is a grid layout which will hold three main areas: one for a table (we use a label for now), one for a notebook, and a layout to hold some buttons.

A `QWidget` instance can hold one immediate layout set by the `setLayout` method. We use a widget for a top-level window and begin by specifing a grid layout.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
gridLayout <- Qt$QGridLayout()
w$setLayout(gridLayout)
```

Here we define the two main widgets and the layout for our buttons.

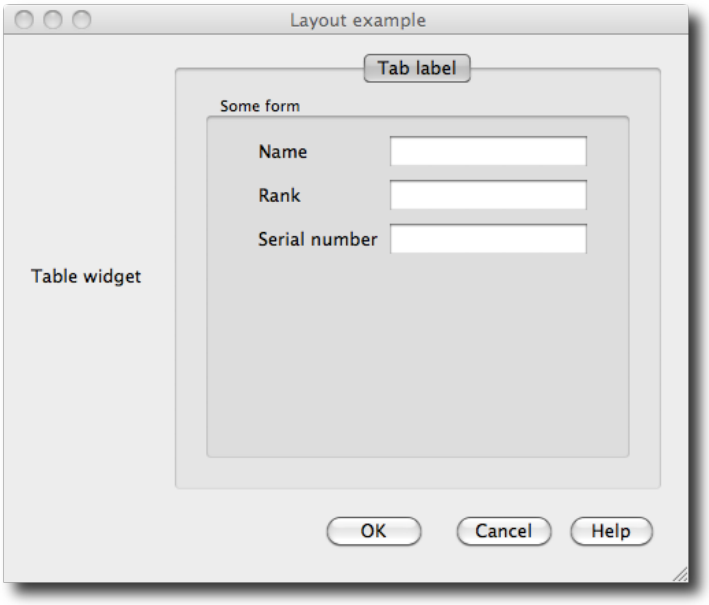

Figure 2.1: A mock GUI illustrating various layout managers provided by Qt.

```
tableWidget <- Qt$QLabel("Table widget")
nbWidget <- Qt$QTabWidget()
buttonLayout <- Qt$QHBoxLayout()
```

Grid layouts have two main methods, `addWidget` which is inherited from the base `QLayout` class, and `addLayout`. To use them, we specify what part of the grid the child widget or layout will occupy through a 0-based row and column and optionally a specification of the row and column span.

```
gridLayout$addWidget(tableWidget, row=0, column=0,
                     rowspan=1, colspan=1)
gridLayout$addWidget(nbWidget, 0, 1)
gridLayout$addLayout(buttonLayout, 1, 1)
```

When resizing, we want to give the area to the notebook widget in row 1, column 1. We do this by adding a weight to the stretch value.

```
gridLayout$setRowStretch(1, stretch=5)
```

```
NULL
```

```
gridLayout$setColumnStretch(1, stretch=5)
```

```
NULL
```

Buttons are added to our box layout through the `addWidget` method. In this case, we want to push the buttons to the right side of the GUI, so we first add a stretch. Stretches are specified by integers. Unallocated space is given first to widgets that have a non-zero stretch factor. We also set spacing of 12 pixels between the "OK" and "Cancel" buttons.

```
b <- sapply(c("OK", "Cancel", "Help"),
            function(i) Qt$QPushButton(i))
buttonLayout$addStretch(1L)
buttonLayout$addWidget(b$OK)
buttonLayout$addSpacing(12L)
buttonLayout$addWidget(b$Cancel)
buttonLayout$addWidget(b$Help)
```

For our notebook widget we add pages through the `addTab` method. We pass in the child widget and a label below. In addition, we set a tooltip for the tab label to give more feedback to the user.

```
nbPage <- Qt$QWidget()
nbWidget$addTab(nbPage, "Tab label")
nbWidget$setTabToolTip(0, "A notebook page with a form")
```

We wish to layout a form inside of the notebook tab, but first create a framed widget using a `QGroupBox` widget to hold the form layout. This widget allows us to easily specify a title. We addd this to the page using a box layout.

```
f <- Qt$QGroupBox()
f$setTitle("Some form")
#
lyt <- Qt$QHBoxLayout()
nbPage$setLayout(lyt)
lyt$addWidget(f)
```

The form layout allows us to create standardized forms where each row contains a label and a widget. Although, this could be done with a grid layout, using the form layout is more convenient, and allows Qt to style the page as appropriate for the underlying operating system.

```
formLayout <- Qt$QFormLayout()
f$setLayout(formLayout)
```

Our form template just uses 3 line-edit widgets. The `addRow` method makes it easy to specify the label and the widget.

```
l <- sapply(c("name", "rank", "snumber"), function(i) Qt$QLineEdit())
formLayout$addRow("Name", l$name)
formLayout$addRow("Rank", l$rank)
formLayout$addRow("Serial number", l$snumber)
```

Finally, we set the minimum size for our GUI and call `show` on the top-level widget.

```
w$setMinimumSize(width=500, height=400)
w$show()
```

2.1 Box layouts

Box layouts arrange child widgets by packing in values horizontally or vertical. The `QHBoxLayout` class implements a horizontal layout whereas `QVBoxLayout` provides a vertical one. Both of these classes subclass the `QBoxLayout` class where most of the functionality is documented. The `direction` property specifies how the layout is done. By default, this is left to right or top to bottom, but can be set (e.g., using `Qt$Qt$LeftToRight`).

Child widgets are added to a box container through the `addWidget` method. The basic call specifies just the child widget, but one can specify an integer value for `stretch` and an alignment enumeration. In addition to adding child widgets, one can nest child layouts through `addLayout`.

A count of child widgets is returned by `count`. Some methods use an index (0-based) to refer to the child components. For example, The `insertWidget` can be used to insert a widget, with arguments similar to `addWidget`. Its initial argument is an integer specifying the `index`. All child widgets with this index or higher have their index increased by 1. Internally, layouts store child components as items of class `QLayoutItem`. The item at a given numeric index is returned by `itemAt`. The actual child component is retrieved by passing the item to the layout's `widget` method.

Qt provides the methods `removeItem` and `removeWidget` to remove an widget from a layout. Once removed from one layout, these may be reparented if desired, or destroyed. This is done by setting the widget's parent to `NULL` using `setParent`.

Size and space considerations The allocation of space to child widgets depends on several factors.

The Qt documentation for layouts spells out the steps: [1]

1. All the widgets will initially be allocated an amount of space in accordance with their sizePolicy and sizeHint.
2. If any of the widgets have stretch factors set, with a value greater than zero, then they are allocated space in proportion to their stretch factor.
3. If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an Expanding size policy first.
4. Any widgets that are allocated less space than their minimum size (or minimum size hint if no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum

[1] `http://doc.qt.nokia.com/4.6/layout.html`

Table 2.1: Possible size policies

Policy	Meaning
Fixed	to require the size hint exactly
Minimum	to treat the size hint as the minimum, allowing expansion
Maximum	to treat the size hint as the maximum, allowing shrinkage
Preferred	to request the size hint, but allow for either expansion or shrinkage
Expanding	to treat like `Preferred`, except the widget desires as much space as possible
MinimumExpanding	to treat like `Minimum`, except the widget desires as much space as possible
Ignored	to ignore the size hint and request as much space as possible

size or minimum size hint in which case the strech factor is their determining factor.)

5. Any widgets that are allocated more space than their maximum size are allocated the maximum size space they require. (Widgets do not have to have a maximum size in which case the strech factor is their determining factor.)

Every widget returns a size hint to the layout from the the `sizeHint` method/property. The interpretation of the size hint depends on the `sizePolicy`roperty. The size policy is an object of class `QSizePolicy`. It contains a separate policy value, taken from the `QSizePolicy` enumeration, for the vertical and horizontal directions. The possible size policies are listed in Table **??**.

tab:qt:size-policies As an example, consider `QPushButton`. It is the convention that a button will only allow horizontal, but not vertical, expansion. It also requires that its has enough space to display its entire label. Thus a `QPushButton` instance returns a size hint depending on the label dimensions and has the policies `Fixed` and `Minimum` as its vertical and horizontal policies respectively. We could prevent a button from expanding at all:

```
b <- Qt$QPushButton("No expansion")
b$setSizePolicy(vertical=Qt$QSizePolicy$Fixed,
                horizontal=Qt$QSizePolicy$Fixed)
```

Thus, the sizing behavior is largely inherent to the widget, rather than any layout parameters. This is a major difference from GTK+, where a widget can only request a minimum size and all else depends on the parent container widget. The Qt approach seems better at encouraging consistency in the layout behavior of widgets of a particular type.

Stretch factors are used to proportionally allocate space to widgets when they expand. When more than one widget can expand, how the space gets allocated is determined by their stretch factors (for box layouts set by the second argument of the `addWidget` method).

When a widget is allocated additional space, its alignment is determined by an alignment specification. This can be done when the widget is added, or later through the `setAlignment` method of the layout. The alignment is specified as a flag using the `Qt::AlignmentFlag` enumeration.

Spacing For a box layout, the space between two children is controlled through the `setSpacing` method. This sets the common width in pixels, which can be adjusted individually through the `addSpacing` method. The margin area around all the children can be adjusted with the `setContentsMargins` method, although this is often specified through the style.

Springs and Struts A stretchable blank widget can be added through the `addStretch` method, where an integer is specified to indicate the stretch factor. If no other widgets have a stretch specified then this widget will take all the non-requested space.

A strut (`addStrut`) can be specified in pixels to restrict the dimension of the box to a minimum height (or width for vertical boxes).

Scrolling layouts

It may be desirable to constrain the size of a layout and allow the user to pan through its children with scrollbars. The `QScrollArea` class makes this very straightforward, as you simply place a container widget into the scroll area. The method `setWidget` is used to add the child widget.

```
sa <- Qt$QScrollArea()
w <- Qt$QWidget()
sa$setWidget(w)
## no add to the child widget w
w$setMinimumSize(400,400)
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
for(i in rownames(state.x77)) {
  msg <- sprintf("%s had a population of %s thousand in 1977",
                 i, state.x77[i,"Population"])
  lyt$addWidget(Qt$QLabel(msg))
}
```

To ensure that a given child widget is visible, the method `ensureWidgetVisible` is available, where the widget is passed as the argument. So, to ensure the value for New York (row 32 in the data set) is visible, we have:

```
widget <- lyt$itemAt(32 - 1)$widget()
sa$ensureWidgetVisible(widget)
```

Framed Layouts

A frame with a title is a common decoration to a container often utilized to group together widgets that are naturally related. In Qt the `QGroupBox` class provides a container widget which can then hold a layout for child items. Its method `setTitle` can be used to set the title, or it can be passed to the constructor. If the standard position of the title determined from the style is not to the likeing, it can be adjusted through the `setAlignment` method. This method takes from the alignment enumeration, for example `Qt$Qt$AlignLeft`. `Qt$Qt$AlignHCenter` or `Qt$Qt$AlignRight`. The property `flat` can be set to `TRUE` to minimize the allocated space.

Group boxes have a `checkable` property that if enabled the widget will be drawn with a checkbox to control whether the children of the group box are sensitive to user input.

Separators

The `QGroupBox` widget provides a framed area to separate off related parts of GUI. Sometimes, just a separating line is all that is desired. There is no separate separator widget in Qt, unlike GTK+. However, the `QFrame` class provides a general method for framed widgets (such as a label) that can be used for this purpose with the appropriate settings.

The frame shape can be a box or other types. For this purpose a line is desired. The enumerations `Qt$QFrame$HLine` or `Qt$QFrame$VLine` can be passed to the method `setFrameShape` to specify a horizontal or vertical line. Its appearance can be altered by `setFrameShadow`. A value of `Qt$QFrame$Sunken` or `Qt$QFrame$Raised` is suitable.

2.2 Grid Layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns.

The `addWidget` method is used to add a child widget to the layout and the `addLayout` method creates nested layouts. Both methods have similar arugments. There are two methods. One where one specifies the child, and just a row and column index and optionally an alignment and another where the row and column indices are followed by specifications for the row and column span followed by an optional alignment.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column. The item method `widget` is

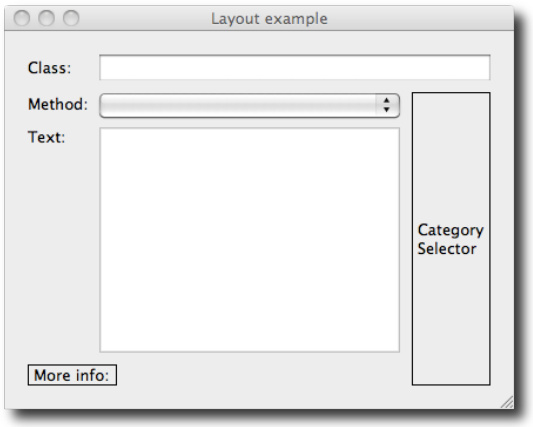

Figure 2.2: A mocked up layout using the `QGridLayout` class.

used to find the corresponding widget. Removing a widget is similar for a box layout using `removeItem` or `removeWidget`. The methods `rowCount` and `columnCount` can be used to find the current size of the grid.

Rows and columns are dual concepts and are so implemented. Consequently, both have similar methods differing only by the use of `column` or `row`. We discuss columns. A column minimum width can be set through `setColumnMinimumWidth`. If more space is available to a column than requested, then the extra space is apportioned according to the `stretch` factors. This can be set for a column through the `setColumnStretch` method. Taking an integer value 0 or larger.

The spacing between widgets can be set in both directions with `setSpacing`, or fine-tuned with `setHorizontalSpacing` or `setVerticalSpacing`. (The style may set these too wide for some tastes.)

Example 2.2: Using a grid layout

To illustrate grid layouts we mock up a GUI centered around a central text area widget (FIgure 2.2). If the window is resized, we want that widget to get the extra space allocated to.

We begin by setting a grid layout for our parent widget.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
lyt <- Qt$QGridLayout()
w$setLayout(lyt)
```

We use the default left-alignment for labels in the following. Our first row has a label in column 1 and a text-entry widget spanning two columns.

```
lyt$addWidget(Qt$QLabel("Class:"), 0, 0)
lyt$addWidget(Qt$QLineEdit(), 0, 1, rowspan=1, colspan=2)
```

Our second row starts with a label and a combobox each spanning the default of one column.

```
lyt$addWidget(Qt$QLabel("Method:"), 1, 0)
lyt$addWidget(Qt$QComboBox(), 1, 1)
```

The third column of the second row and rest of the layout is managed by a sublayout, in this case a vertical box layout. We use a label as a stub and set a frame style to have it stand out.

```
lyt$addLayout(slyt <- Qt$QVBoxLayout(), 1, 2, rowspan=3, 1)
slyt$addWidget(l <- Qt$QLabel("Category\nSelector"))
l$setFrameStyle(Qt$QFrame$Box)
```

The text edit widget is added to the third row, second column. Since this widget will expand, we set an alignment for the label. Otherwise, the default alignment will center it in the vertical direction.

```
lyt$addWidget(Qt$QLabel("Text:"), 2, 0, Qt$Qt$AlignTop)
lyt$addWidget(l <- Qt$QTextEdit(), 2, 1)
```

Finally we add a space for information on the 4th row. Again we placd this in a box. By default the box would expand to fill the space of the two columns, but we fix this below as an illustration.

```
lyt$addWidget(l <- Qt$QLabel("More info:"), 3, 0,
              rowspan=1, colspan=2)
l$setSizePolicy(Qt$QSizePolicy$Fixed, Qt$QSizePolicy$Preferred)
l$setFrameStyle(Qt$QFrame$Box)
```

As it is, since there are no stretch factors set, the space allocated to each row and column would be identical when resized. To force the space to go to the text widget, we set a positive stretch factor for the 3rd row and 2nd column. Since the others have the default stretch factor of 0, this will allow that cell to grow when the widget is resized.

```
lyt$setRowStretch(2, 1)
lyt$setColumnStretch(1,1)
```

2.3 Form layouts

Forms can easily be generated with the grid layout, but Qt provides an even more convenient form layout (`QFormLayout`) that has the added benefit of conforming to the traditional styles for the underlying operating system. This can be used in combination with the `QDialogButtonBox` (Section 3.3), which provides a container for buttons that also tries to maintain an appearance consistent with the underlying operating system.

To add a child widget with a label is done through the `addRow` method, where the label, specified first, may be given as a string and the widget is specified second. The first argument can also be a widget to replace the label, and the second a layout for nesting layouts. The `insertRow` method is similar, only one first specifies the row number using a 0-based index. The `setSpacing` method can be used to adjust the default spacing between rows.

After construction, the widget may be retrieved through the `itemAt` method. This returns a layout item, to get the widget call its `widget` method. The `setWidget` method can be used to change a widget.

Example 2.3: Simple use of `QFormLayout`
The following illustrates a basic usage where three values are gathered.

```
w <- Qt$QWidget()
w$setWindowTitle("Example of a form layout")
w$setLayout(flyt <- Qt$QFormLayout())
l <- list()
flyt$addRow("mean", l$mean <- Qt$QLineEdit())
flyt$addRow("sd", l$sd <- Qt$QLineEdit())
flyt$addRow("n", l$n <- Qt$QLineEdit())
```

```
w$show(); w$raise()
```

```
NULL
```

```
NULL
```

The form style can be overridden using the `setFormAlignment` and `setLabelAlignment` methods. The Mac OS X default is to have center aligned form with right-aligned labels, whereas the Windows default is to have left-aligned labels. One can also override the default as to how the fields should grow when the widget is resized (`setFieldGrowthPolicy`).

For example,

```
flyt$setFormAlignment(Qt$Qt$AlignLeft | Qt$Qt$AlignTop)
flyt$setLabelAlignment(Qt$Qt$AlignLeft);
```

2.4 Notebooks

A notebook container is provided by the widget `QTabWidget`. This is not a layout, rather a notebook page consists of a label and widget. Of course, you can use a layout within the widget.

Pages are added through the method `addTab`. One can specify a widget; a widget and label; or a widget, icon and label. As well, pages may be inserted by index with the `insertTab` method.

Tabs allow the user to select from among the pages, and in Qt the tabs can be customized. The text for a tab is adjusted through `setTabText` and the icon through `setTabIcon`. These use a 0-based index to refer to the tab. A tooltip can be added through `setTabToolTip`. The tabs will have close buttons if the property `tabsClosable` is `TRUE`. One connects to the `tabCloseRequested` signal to actually close the tab. The tab position is adjusted through the `setTabPosition` method using values in the enumeration `QTabWidget::TabPosition`, which uses compass headings like `Qt$QTabWidget$North`. Calling `isMovable` with `TRUE` allows the pages to be reorganized by the user.

When there are numerous tabs, the method `setUsesScrollButtons` can indicate if the widget should expand to accomodate the labels or add scroll buttons.

The current tab is adjusted through the `currentIndex` property. The actual widget of the current tab is returned by `curentWidget`. To remove a page the `removeTab` is used, where tabs are referred to by index.

Example 2.4: A tab widget
A simple example follows. First the widget is defined with several properties set.

```
nb <- Qt$QTabWidget()
nb$setTabsClosable(TRUE)
nb$setMovable(TRUE)
nb$setUsesScrollButtons(TRUE)
```

We can add pages with a label or with a label and an icon:

```
nb$addTab(Qt$QPushButton("page 1"), "page 1")
icon <- Qt$QIcon("small-R-logo.jpg")
nb$addTab(Qt$QPushButton("page 2"), icon,  "page 2")
## we add numerous tabs to see scroll buttons
for(i in 3:10) nb$addTab(Qt$QPushButton(i), sprintf("path %s", i))
```

The close buttons put out a request for the page to be closed, but do not handle directly. Something along the lines of the following is then also needed.

```
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
```

QStackedWidget

The `QStackedWidget` is provided by Qt to hold several widgets at once, with only one being visible. Similar to a notebook, only without the tab decorations to switch pages. For this widget, children are added through the `addWidget` method and can be removed with `removeWidget`. The latter

needs a widget reference. The currently displayed widget can be found from `currentWidget` (which returns `NULL` if there are no child widgets). Alternatively, one can refer to the widgets by index, with `count` returning their number, and `currentIndex` the current one and `indexOf` returns the index of the widget specified as an argument.

2.5 Paned Windows

Split windows with handles to allocate space are created by `QSplitter`. There is no limit on the number of child panes that can be created. The default orientation is horizontal, set the `orientation` property using `Qt$Qt$Vertical` to change this.

Child widgets are added through the `addWidget` method. These widgets are referred to by index and can be retrieved through the `widget` method.

The `moveSplitter` method is not implemented, so programattically moving the a splitter handle is not possible.

```
sp <- Qt$QSplitter()
sp$addWidget(Qt$QLabel("One"))
sp$addWidget(Qt$QLabel("Two"))
sp$addWidget(Qt$QLabel("Three"))
sp$setOrientation(Qt$Qt$Vertical)
```

```
sp$widget(0)$text              # text in first widget
```

```
[1] "One"
```

2.6 Main windows

In Qt the `QMainWindow` class provides a widget for use as the primary widget in an interface. Although it is a subclass of `QWidget` – which we have used in our examples so far as a top-level window – the implementation also has preset areas for the standard menu bars, tool bars and status bars. As well, there is a possible dock area for "dockable widgets."

A main window has just one central widget that is specified through the `setCentralWidget` method. The central widget can then have a layout and numerous children. The title of the window is set from the inherited method `setWindowTitle`.

Actions

The menubars and toolbars are representations of collections of actions, defined through the `QAction` class. As with other toolkits, an action encapsulates a command that may be shared among parts of a GUI, in this case

menu bars, toolbars and keyboard shortcuts. In Qt the action can hold the label (`setText`), an icon (`setIcon`), a status bar tip (`setStatusTip`), a tool tip (`setToolTip`), and a keyboard shortcut (`setShortcut`). The text and icon may be set at construction time, in addition to using the above methods.

Actions inherit the `enabled` method to toggle whether an action is sensitive to user input. Actions emit a `triggered` signal when activated (sending to the callback a boolean value for checked when applicable).

Keyboard shortcuts A keyboard shortcut use a `QKeySequence` to bind a key sequence to the command. Key sequences can be found from the standard shortcuts provided in the enumeration `Qt$QtKeySequence`, for example

```
Qt$QKeySequence$Cut
```

```
Enum value: Cut (8)
```

This value (or more simply the "Cut" string) can be passed to the constructor to create the shortcut.

Using these standard shortcuts ensures that the keyboard shortcut is the standard one for the underlying operating system. Alternatively, custom shortcuts can be used, such as

```
Qt$QKeySequence("Ctrl-X, Ctrl-C")
```

```
QKeySequence instance
```

This shows how a modifier can be specified (from `"Ctrl"`) a key (case insensitive), and how a comma can be used to create multi-key shortcuts.

For buttons and labels, a shortcut key can be specified by prefixing the text with an ampersand `&`, such as

```
button <- Qt$QPushButton("&Save")
```

Then the shortcut will by Alt-S.

The shortcut event occurs when the shortcut key combination is pressed in the appropriate context. The default context is when the widget is a child of the parent window, but this can be adjusted through the method `setShortcutContext`.

Checkable actions, and action groups Actions can be set checkable through the `setCheckable` method. When in a checked state, the `checked` property is `TRUE`. When a checkable action is checked the `toggled` signal is emitted, the argument `checked` passes in the state.

A `QActionGroup` can be used to group together checkable actions so that only one is checkable (like radio buttons). To use this, you create an instance and use the `addAction` to link the actions to the group.

Example 2.5: Creating an action
To create an action, say to "Save" an object requires a few steps. It is recommended that the main constructor is passed the parent widget the action will apply within.

```
parent <- Qt$QMainWindow()
saveAction <- Qt$QAction("Save", parent)
```

We could also pass the icon to the constructor, but instead set the icon, a shortcut, a tooltip, and a statusbar tip through the action's methods.

```
saveAction$setShortcut(Qt$QKeySequence("Save"))
iconFile <- system.file("images/save.gif", package="gWidgets")
saveAction$setIcon(Qt$QIcon(iconFile))
saveAction$setToolTip("Save the object")
saveAction$setStatusTip("Save the current object")
```

The action encapsulates a command, in this case we have a stub:

```
qconnect(saveAction, "triggered", function(checked)
         print("Save object"))
```

Menubars

Main windows may have a menubar. This may appear at the top of the window, or the menubar area on Mac OS X. Menubars are instances of `QMenuBar` which provides access to list of top-level submenus.

These submenus are added through `addMenu`, where a string with a possible shortcut are specied to label the menu. A `QMenu` instance is returned. To submenus one can add

1. nested submenus through the `addMenu` method,
2. an action through the `addAction` method, or
3. a separator through `addSeparator`.

Actions may be removed from a window through `removeAction`, but usually menu items are just disabled if their command is not applicable.

Example 2.6: Menu items
In a data editor application, one might imagine a menu item for coercion of a chosen column from one type to the next. In the following, we assume we have a function `colType` that returns the column type of the selected column or `NA` if no column is selected. We begin by making a menu bar, and a "Data" menu item. To this we add a few actions, and then a "Coerce" submenu. In the submenu, we use an action group so that only one type can be checked at a time. Actions must be added to both the action group and the submenu.

```
mb <- Qt$QMenuBar()                    # or parent$menuBar()
menu <- mb$addMenu("Data")             # a submenu
#
```

```
menu$addAction(a <- Qt$QAction("Apply Function...", parent))
qconnect(a, "triggered", function() cat("apply ..."))
menu$addAction(a <- Qt$QAction("Relevel Factors...", parent))
qconnect(a, "triggered", function() cat("relevel ..."))
#
menu$addSeparator()
#
cmenu <- menu$addMenu("Coerce")
aList <- sapply(c("character", "factor", "numeric"),
         function(i) {
           a <- Qt$QAction(i, parent)
           a$setCheckable(TRUE)
           qconnect(a, "toggled", function(checked) print(i)) ## stub
           a
         })
actionGroup <- Qt$QActionGroup(w)
sapply(aList, function(i) actionGroup$addAction(i))
sapply(aList, function(i) cmenu$addAction(i))
```

In the application, we might include logic to update the menu items along the lines of the following. If no column type is available (no column is selected) we disable the submenu, otherwise we set the check accordingly. Of course, in the application we would ensure that checking the menu item updates the state in the data model through the `triggered` handler.

```
updateMenus <- function() {
  val <- colType()
  cmenu$setEnabled(!is.na(val))
  if(!is.na(val)) {
    aList[[val]]$setChecked(TRUE)
  }
}
```

Context menus

Context menus can be added to widgets using the same `QMenu` widget (not `QMenuBar`). The `popup` method will cause the menu to popup, but it needs to be told where. (The `exec` method will also popup a menu, but blocks other input.) The location of the popup is specified in terms of global screen coordinates, but typically the location known is in terms of the widgets coordinates. (For example, the point $(0,0)$ being the upper-left corner of the widget.) The method `mapToGlobal` will convert for you. Position is in terms of a `QPoint` instance, which can be constructed or may be returned by an event handler. We illustrate both in the example.

Initiating the popup menu can be done in different ways. In the example below, we first show how to do it when a button is pressed. More natural ways are to respond to right mouse clicks, say. These events may be found

within event handlers, say the `mousePressEvent` event. (The `QMouseEvent` object passed in has a `button` method that can be checked.) However, the operating system may provide other means to initiate a popup. Rather than program these, Qt provides the `contextMenuEvent`. We can override that in a subclass, as illustrated in the example.

Example 2.7: Popup menus
We imagine a desire to popup possible function names that complete a string. Such suggestions are computed from a function in the `utils` package. We first show how to offer these in a popup menu we do this for a button press (not the most natural case):

```
b <- Qt$QPushButton("Completion example")
qconnect(b, "pressed", function(...) {
  ## compute popup
  popup <- Qt$QMenu()
  comps <- utils:::matchAvailableTopics("mean")
  l <- sapply(comps, function(i) {
    a <- Qt$QAction(i, b)
    popup$addAction(a)
  })
  popup$popup(b$mapToGlobal(Qt$QPoint(0,0)))
})
```

More naturally, we might want this menu to popup on a right mouse click in a line edit widget. To implement that, we define a subclass and reimplement the `contextMenuEvent` method. We use the `globalPos` method of the passed through event to get the appropriate position.

```
qsetClass("popupmenuexample", Qt$QLineEdit)
#
qsetMethod("contextMenuEvent", popupmenuexample, function(e) {
  popup <- Qt$QMenu()
  comps <- utils:::matchAvailableTopics(this$text)
  if(length(comps) > 10)
    comps <- comps[1:10]                # trim if large
  sapply(comps, function(i) {
    a <- Qt$QAction(i, this)
    qconnect(a, "triggered", function(...) this$setText(i))
    popup$addAction(a)
  })
  popup$popup(e$globalPos())
})
e <- popupmenuexample()
```

Toolbars

Toolbars, giving easier access to a collection of actions or even widgets, can be readily added to a main window. The basic toolbar is an instance of the `QToolBar` class. Toolbars are added to the main window through the `addToolbar` method.

To add items to a toolbar we have:

1. `addAction` to add an action,
2. `addWidget` to embed a widget into the toolbar,
3. `addSeparator` to place a sepearator between items.

Actions can be removed through the `removeAction` method, although typically they are simply disabled as appropriate. The method `actions` will return a list of actions.

Toolbars can have a few styles. The orientation can be horizontal (the default) or vertical. The `setOrientation` method adjusts this with values specified by `Qt$Qt$Horizontal` or `Qt$Qt$Vertical`. The toolbuttons can show a combination of text and/or icons. This is specified through the method `setToolButtonStyle` with values taken from the `Qt::ToolButtonStyle` enumeration. The default is icon only, but one could use, say, `Qt$Qt$ToolButtonTextUnderIcon`.

Example 2.8: A simple toolbar
The following illustrates how to put in toolbar items to open and save a file. We suppose we have a function `getIcon` that returns a `QIcon` instance from a string.

We define a top-level window to hold our toolbar and be the parent for our actions that will be placed in the toolbar. We store them in a list for ease in manipulation at a later time in the program.

```
w <- Qt$QMainWindow()
a <- list()
a$open <- Qt$QAction("Open", w)
a$open$setIcon(getIcon("open"))
a$save <- Qt$QAction("Save", w)
a$save$setIcon(getIcon("save"))
```

We define our toolbar, set its button style and then add to top-level window in the next few commands.

```
tb <- Qt$QToolBar()
tb$setToolButtonStyle(Qt$Qt$ToolButtonTextUnderIcon)
w$addToolBar(tb)
```

Finally, we add the actions to the toolbar.

```
sapply(a, function(i) tb$addAction(i))
```

Statusbars

Main windows have room for a statusbar at the bottom of the window. The status bar is used to show programmed messages as well as any status tips assigned to actions.

A statusbar is an instance of the `QStatusBar` class. One may be added to a main window through the `setStatusBar` method. For some operating systems, a size grip is optional and its presence can be adjusted through the `sizeGripEnabled` property.

Messages may be placed in the status bar of three types: *temporary* where the message stays briefly, such as for status tips; *normal* where the message stays, but may be hidden by temporary messages, and *permanent* where the message is never hidden. In addition to messages, one can embed widgets into the statusbar.

The `showMessage` method places a temporary message. The duration can be set by specifying a time in milliseconds for a second argument. Otherwise, the message can be removed through `clearMessage`.

Use `addWidget` with a label to make a normal message, use `addPermanentWidget` to make a permanent message.

Dockable widgets

In Qt main windows have dockable areas where one can anchor widgets that can be easily detached from the main window to float if the user desires. An example use might be a toolbar or in a large GUI, a place to dock a workspace browser. The main methods are `addDockWidet` and `removeDockWidget`. Adding a dock widget requires first creating an instance of `QDockWidget` and then setting the desired widget through the dock widget's `setWidget` method. Widgets may go on any side of the central widget. The position is specified through the `DockWidgetArea` enumeration, with values such as `Qt$Qt$LeftDockWidgetArea`.

Dock widgets can be stacked or arranged in a notebook like manner. The latter is done by the `tabifyDockWidget`, which moves the second argument (a dock widget) on top of the first with tabs, like a notebook, for the user to select the widget.

Floating a dock widget is intitiated by the user through clicking an icon in the widget's title bar or programatically through the `floating` property.

Example 2.9: Using a main widget for the layout of an IDE

This example shows how to mock up a main window (Figure **??**) similar to the one presented by the web application, R-Studio. (`rstudio.org`).

We begin by setting a minimum size and a title for the main window.

```
w <- Qt$QMainWindow()
w$setMinimumSize(800, 500)
```

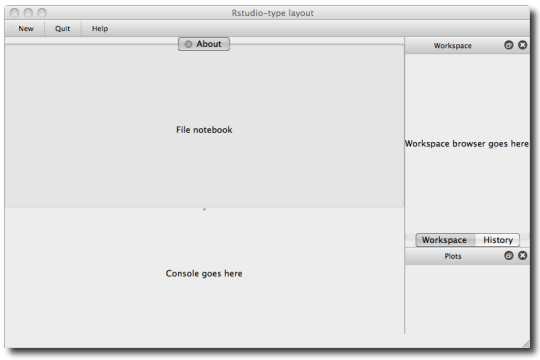

Figure 2.3: A mock of for a possible GUI using dockable widgets and a `QMainWindow` instance

```
w$setWindowTitle("Rstudio-type layout")
```

We add in a menu bar and toolbar. The actions are minimal, not including icons, commands etc. We show the file menu definitions.

```
l <- list()
mb <- Qt$QMenuBar()
w$setMenuBar(mb)
fmenu <- mb$addMenu("File")
fmenu$addAction(l$new <- Qt$QAction("New", w))
fmenu$addSeparator()
fmenu$addAction(l$open <- Qt$QAction("Open", w))
fmenu$addAction(l$save <- Qt$QAction("Save", w))
fmenu$addSeparator()
fmenu$addAction(l$quit <- Qt$QAction("Quit", w))
```

The toolbar has just a few actions added.

```
tb <- Qt$QToolBar()
w$addToolBar(tb)
tb$addAction(l$new)
tb$addSeparator()
tb$addAction(l$quit)
tb$addSeparator()
tb$addAction(l$help <- Qt$QAction("Help", w))
```

Our central widget holds two main areas: one for editing files and one for a console. As we may want to edit multiple files, we use a tab widget for that. A `QSplitter` is used to divide the space between the two main widgets.

```
centralWidget <- Qt$QSplitter()
centralWidget$setOrientation(Qt$Qt$Vertical)
w$setCentralWidget(centralWidget)
```

```
fileNotebook <- Qt$QTabWidget()
l <- Qt$QLabel("File notebook")
l$setAlignment(Qt$Qt$AlignCenter)
fileNotebook$addTab(l, "About")
fileNotebook$setTabsClosable(TRUE)
qconnect(fileNotebook, "tabCloseRequested", function(ind, nb) {
  nb$removeTab(ind)
}, user.data=fileNotebook)
centralWidget$addWidget(fileNotebook)
```

Our console widget is just a stub.

```
consoleWidget <- Qt$QLabel("Console goes here")
consoleWidget$setAlignment(Qt$Qt$AlignCenter)
centralWidget$addWidget(consoleWidget)
```

On the right side of the layout we will put in various tools for interacting with the R session. We place these into dock widgets, in case the user would like to place them elsewhere on the screen. Defining dock widgets is straightforward. We show a stub for a workspace browser.

```
workspaceBrowser <- Qt$QLabel("Workspace browser goes here")
wbDockWidget <- Qt$QDockWidget("Workspace")
wbDockWidget$setWidget(workspaceBrowser)
```

The workspace and history browser are placed in a notebook to conserve space. We add the workspace browser on the right side, then tabify the history browser (whose construction is not shown).

```
w$addDockWidget(Qt$Qt$RightDockWidgetArea, wbDockWidget)
w$tabifyDockWidget(wbDockWidget, hbDockWidget)
```

We next place a notebook to hold any graphics produced in a dock widget. This one occupies its own space.

```
plotNotebook <- Qt$QTabWidget()
pnDockWidget <- Qt$QDockWidget("Plots")
w$addDockWidget(Qt$Qt$RightDockWidgetArea, pnDockWidget)
```

Finally, we make status bar and add a transient message.

```
sb <- Qt$QStatusBar()
w$setStatusBar(sb)
sb$showMessage("Mock-up layout for an IDE", 2000)
```

3

Widgets

This chapter covers some of the basic dialogs and widgets provided by Qt, saving for later a discussion on widgets that have a model backend. Together with layouts, these form the basis for most user interfaces.

3.1 Dialogs

Qt implements the conventional high-level dialogs, including those for printing, selecting files, selecting colors, and, most usefully, sending simple messages and input requests to the user. We first introduce message and input dialogs. This is followed by a discussion of the infrastructure in Qt for implementing custom dialogs. Finally, we briefly introduce some of the remaining high-level dialogs, such as the file selector.

Message Dialogs

All dialogs in Qt are derived from `QDialog`. The message dialog, `QMessageBox`, communicates a textual message to the user. At the bottom of the dialog are a set of buttons, each representing a possible response. Normally, the type of message is indicated by an icon. If extra details are available, the dialog provides the option for the user to view them.

Qt provide two ways to create a message box. The simplest approach is to call a static convenience method for issuing common types of messages, including warnings and simple questions. The alternative, described later, involves several steps and offers more control at a cost of convenience. Here we take the simple path for presenting a warning dialog:

```
response <- Qt$QMessageBox$warning(parent = NULL, title = "Warning!",
                                   text = "Warning message...")
```

This blocks the flow of the program until the user responds and returns the standard identifier for the button that was clicked. Each type of button corresponds to a fixed type of response. The standard button/response codes are listed in the `QMessageBox::StandardButton` enumeration. In this case, there

is only a single button, "QMessageBox$Ok". The dialog is *modal*, meaning that the user cannot interact with the "parent" window until responding. If the "parent" is "NULL", as in this case, input to all windows is blocked. The dialog is automatically positioned near its parent, and if the parent is destroyed, the dialog is destroyed, as well. Additional arguments specify the available buttons/responses and the default response. We have relied on the default values for these.

For more control over the appearance and behavior of the dialog, we will take a more gradual path. First, we construct an instance of QMessageBox. It is possible to specify several properties at construction. Here is how one might construct a warning dialog:

```
dlg <- Qt$QMessageBox(icon = Qt$QMessageBox$Warning,
                      title = "Warning!",
                      text = "Warning text...",
                      buttons = Qt$QMessageBox$Ok,
                      parent = NULL)
```

This call introduces the icon property, which is a code from the QMessageBox::Icon enumeration and identifies a standard, themeable icon. The icon also implies the message type, just as a button implies a response type. We also need to specify the possible responses with the "buttons" argument.

Our dialog is already sufficiently complete to be displayed. However, we have the opportunity to specify further properties. Two of the most useful are informativeText and detailedText:

```
dlg$informativeText <- "Less important warning information"
dlg$detailedText <- "Extra details most do not care to see"
```

Both provide additional textual information at an increasing level of detail. The informativeTextQMessageBox will be rendered as secondary to the actual message text. To display the detailedText, the user will need to interact with a control in the dialog. An example is a stack trace for the warning.

After specifying the desired properties, the dialog is shown. The approach to showing the dialog depends on whether the dialog should be modal. A modal dialog is displayed with the exec method.

```
dlg$exec()
```

```
[1] 1024
```

As its name implies, exec executes a loop that will block until the user responds. As with the static convenience methods, the return value indicates the button/response.

To present a non-modal dialog, we first need to register a response listener, as the response will arrive asynchronously. Then we show, raise and activate the dialog:

```
qconnect(dlg, "finished", function(response) {
  ## handle response
  ## dlg$close() necessary?
})
dlg$show()
dlg$raise()
dlg$activateWindow()
```

There are several signals that indicate user response, including `"finished"`, `"accepted"`, and `"rejected"`. The most general is `"finished"`, which passes the button/response code as its only argument.

Modal dialogs may be window modal (`Qt$Qt$WindowModal`), where the dialog blocks all access to its ancestor windows, or application modal (`Qt$Qt$ApplicationModal`) (the default) where all windows are blocked. To specify the type of modality, call `setWindowModality`.

To summarize, we present a general message box, supporting multiple responses:

```
dlg <- Qt$QMessageBox()
dlg$windowTitle <- "[This space for rent]"
dlg$text <- "This is the main text"
dlg$informativeText <- "This should give extra info"
dlg$detailedText <- "And this provides\neven more detail"
dlg$icon <- Qt$QMessageBox$Critical
dlg$standardButtons <- Qt$QMessageBox$Cancel | Qt$QMessageBox$Ok
## 'Cancel' instead of 'Ok' is the default
dlg$setDefaultButton(Qt$QMessageBox$Cancel)
if(dlg$exec() == Qt$QMessageBox$Ok)
  print("A Ok")
```

Input dialogs

The `QInputDialog` class provides a convenient means to gather information from the user and is in a sense the inverse of `QMessageBox`. Possible input modes include selecting a value from a list or entering text or numbers. By default, input dialogs consist of an input control, an icon, and two buttons: “Ok” and “Cancel”.

Like `QMessageBox`, one can display a `QInputDialog` either by calling a static convenience method or by constructing an instance and configuring it before showing it. We demonstrate the former approach for a dialog that requests textual input:

```
text <- Qt$QInputDialog$getText(parent = NULL,
                        title = "Gather text",
                        label = "Enter some text")
```

The return value is the entered string, or "NULL" if the user cancelled the dialog. Additional parameters allow one to specify the initial text and to override the input mode, e.g., for password-style input.

We can also display a dialog for integer input. Here, we ask the user for an even integer between 1 and 10:

```
num <- Qt$QInputDialog$getInt(parent = NULL, title = "Gather integer",
                       label = "Enter an integer from 1 to 10",
                       value = 0, min = 2, max = 10, step = 2)
```

The number is chosen using a bounded spin box. To request a real value, call Qt$QInputDialog$getDouble instead.

The final type of input is selecting an option from a list of choices:

```
item <- Qt$QInputDialog$getItem(parent = NULL, title = "Select item",
                        label = "Select a letter",
                        items = LETTERS, current = 17)
```

The dialog contains a combo box filled with the capital letters. The initial choice is 0-based index 17, or the letter "R". The chosen string is returned.

QInputDialog has a number of options that cannot be specified via one of the static convenience methods. These option flags are listed in the QInputDialog$InputDialogOption enumeration and include hiding the "Ok" and "Cancel" buttons and selecting an item with a list widget instead of a combo box. If such control is necessary, we must explicitly construct a dialog instance, configure it, execute it and retrieve the selected item.

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Select item")
dlg$setLabelText("Select a letter")
dlg$setComboBoxItems(LETTERS)
dlg$setOptions(Qt$QInputDialog$UseListViewForComboBoxItems)
```

```
if (dlg$exec())
  print(dlg$textValue())
```

```
[1] "A"
```

QDialog

Every dialog in Qt inherits from QDialog, and we can leverage it for our own custom dialogs.

What makes a dialog different is when it is modal. That is the exec call is used. This state may be broken by quitting the window through the window manager, or by calling the done method with an integer specifying the return value (say Qt$QDialog$Accepted or Qt$QDialog$Rejected). This closes the window and returns control. If the QtQtWA_DeleteOnClose attribute is set, the dialog will be deleted, othewise it may be reused.

A simple example follows. We assume some parent window is defined as `parent`

Our dialog is defined and we adjust our window modality accordingly. Under Mac OS X, this will appear as a drop down sheet of the parent, not in a separate window.

```
dlg <- Qt$QDialog(parent)
dlg$setWindowModality(Qt$Qt$WindowModal)
dlg$setWindowTitle("A simple dialog")
```

Our dialog here is just a basic mock up. We use a horizontal box for the buttons, but in an real application would use the `QDialogButtonBox`

```
dlg$setLayout(lyt <- Qt$QVBoxLayout())
lyt$addWidget(Qt$QLabel("Layout dialog components here"))
blyt <- Qt$QHBoxLayout()                # for buttons
lyt$addLayout(blyt)
```

Our buttons have callbacks that call the `done` method with a return value. We use `user.data` to pass in the dialog reference.

```
ok <- Qt$QPushButton("Ok")
cancel <- Qt$QPushButton("Cancel")
blyt$addWidget(ok)
blyt$addWidget(cancel)
qconnect(ok, "pressed", function(dlg) dlg$done(1), user.data=dlg)
qconnect(cancel, "pressed", function(dlg) dlg$done(0), user.data=dlg)
```

```
if(dlg$exec())
  print("Yes")
```

File and Directory chooing dialogs

`QFileDialog` allows the user to select files and directories, by default using the platform native file dialog. As with other dialogs there are static methods to create dialogs with standard options. These are `"getOpenFileName"`, `"getOpenFileNames"`, `"getExistingDirectory"`, and `"getSaveFileName"`. To select a file name to open we would have:

```
fname <- Qt$QFileDialog$getOpenFileName(NULL, "Open a file...", getwd())
```

All take as intial arguments a parent, a caption and a directory. Other arguments allow one to set a filter, say. For basic use, these are nearly as easy to use as R's `file.choose`. If a file is selected, `fname` will contain the full path to the file, otherwise it will be `NULL`.

To allow multiple selection, call the plural form of the method:

```
fnames <- Qt$QFileDialog$getOpenFileNames(NULL, "Open file(s)...", getwd())
```

To select a file name for saving, we have

```
fname <- Qt$QFileDialog$getSaveFileName(NULL, "Save as...", getwd())
```

And to choose a directory,

```
dname <- Qt$QFileDialog$getExistingDirectory(NULL, "Select directory", getwd())
```

To specify a filter by file extension, we use a name filter. A name filter is of the form `Description (*.ext *.ext2)` (no comma) where this would match files with extensions `ext` or `ext2`. Multiple filters can be used by separating them with two semicolons. For example, this would be a natural filter for R users:

```
rfilter <- paste("R files (*.R .RData)",
                 "Sweave files (*.Rnw)",
                 "All files (*.*)", sep=";;")
fnames <- Qt$QFileDialog$getOpenFileNames(NULL,
                          "Open file(s)...", getwd(),
                          rfilter)
```

Explicitly constructing a dialog Although the static functions provide most of the functionality, to fully configure a dialog, it may be necessary to explicitly construct and manipulate a dialog instance. Examples of options not available from the static methods are history (previously selected file names), sidebar shortcut URLs, and filters based on low-level file attributes like permissions.

Example 3.1: File dialogs
We construct a dialog for opening an R-related file, using the file names selected above as the history:

```
dlg <- Qt$QFileDialog(NULL, "Choose an R file", getwd(), rfilter)
dlg$fileMode <- Qt$QFileDialog$ExistingFiles
dlg$setHistory(fnames)
```

The dialog is executed like any other. To get the specified files, call `selectedFiles`:

```
if(dlg$exec())
  print(dlg$selectedFiles())
```

3.2 Labels

As seen in previous example, basic labels in Qt are instances of the `QLabel` class. Labels in Qt are the primary means for displaying static text and images. Textual labels are the most common, and the constructor accepts a string for the text, which can be plain text or, for rich text, HTML. Here we use HTML to display red text:

```
l <- Qt$QLabel("<font color='red'>Red</font>")
```

The class, by default, guesse if the string is rich text or not and in the above identifies the HTML. One can explicitly set the text format (`setTextFormat`) if need be.

The label text is stored in the `text` property. Properties relevant to text layout include: `alignment`, `indent` (in pixels), `margin`, and `wordWrap`.

3.3 Buttons

Buttons are created by `QPushButton` which inherits most of its functionality from `QAbstractButton`. A button has a `text` property for storing a label and an `icon` property to show an accompanying image.

Buttons are associated with commands. One may bind to the inherited signal `clicked`, which is called when the button is activated by the mouse, a short cut key, or a call to `clicked`. The callback receives a logical value `checked` if the button is "checkable." Otherwise, `pressed` and `released` signals are emitted.

Button boxes

Dialogs often have a standard button placement that varies among operating systems. Qt provides the `QDialogButtonBox` class to store buttons. This class accepts the standard buttons listed in the `QDialogButtonBox::StandardButton` enumeration. Each standard button has a default role from a list of roles specified in `QDialogButtonBox::ButtonRole`, if a non-standard button is desired, then a role must be specified. The method `addButton` method is used to add a button, as specified by either a standard button or by a label and role. This method returns a `QPushButton` instance.

In a dialog, a button may be designated as the default button for a dialog. To specify a default, the button's `setDefault` is called with a value of `TRUE`.

To get feedback, one can connect each button to the desired signal. More conveniently, the button box has signals `accepted`, which is emitted when a button with the `AcceptRole` or `YesRole` is clicked; `rejected`, which is emitted when a button with the `RejectRole` or `NoRole` is clicked; `helpRequested`; or `clicked` when any button is clicked. For this last signal, the callback is passed in the button object.

Example 3.2: A yes-no-help set of buttons
We use a dialog button box to hold a standard set of buttons. Figure 3.1 shows the difference in their display for two different operating systems. Below, we just illustrate how to specify callbacks based on the button's role, but only put in stubs for their commands.

```
db <- Qt$QDialogButtonBox()
```

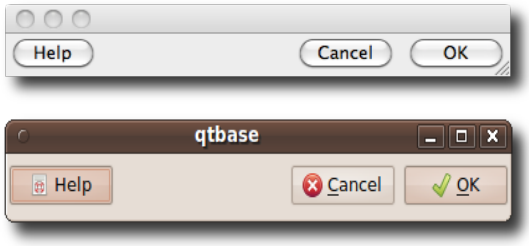

Figure 3.1: Dialog button boxes and their implementation under Mac OS X and Linux.

```
ok <- db$addButton(Qt$QDialogButtonBox$Ok)
db$addButton(Qt$QDialogButtonBox$Cancel)
## Or: db$addButton("Cancel", Qt$QDialogButtonBox$RejectRole)
db$addButton(Qt$QDialogButtonBox$Help)
#
ok$setDefault(TRUE)
#
qconnect(db, "accepted", function() print("accepted"))
qconnect(db, "rejected", function() print("rejected"))
qconnect(db, "helpRequested", function() print("help"))
```

Icons and pixmaps

In Figure 3.1 we see that in some toolkits buttons may carry a standard icon. A user can also pass in an icon through the `setIcon` method. Icons are instances of the `QIcon` class. Icons may be read in from files or from `QPixmap` instances. Although one may, it is not necessary to define icons for different sizes and display modes (normal, disabled, active, selected), as Qt can do so automatically.

For example, we define an icon and add it to a button as follows. The call to `setEnabled` shows how Qt can draw the icon in a disabled state.

```
ok <- system.file("images/ok.gif", package="gWidgets")
icon <- Qt$QIcon(ok)
b <- Qt$QPushButton("Ok")
b$setIcon(icon)
b$setEnabled(FALSE)
```

The `QPixmap` class creates a means to represent an image on a paint device. One can create icons from pixmaps, and the `pixmap` method will return a pixmap from an icon. (The size may be specified with integers, and the mode can be set). Pixmaps can be directly made through the constructor. For example, the following creates a button icon using code from the `ggplot` package.

```
png(f <- tempfile())
grid.newpage()
grid.draw(GeomHistogram$icon())
dev.off()
pix <- Qt$QPixmap(f)
b <- Qt$QPushButton("Histogram")
b$setIcon(Qt$QIcon(pix))
```

The `QPixmap` class has several methods for manipulating the image not discussed here.

3.4 Checkboxes

The `QCheckBox` class implements a checkbox. Like the `QPushButton` class, class inherits methods and properties from the `QAbstractButton` class. For example, the checkbox label is associated with the `text` property.

Qt has three states for a checkbox: the obvious `Qt$Qt$Checked`, `Qt$Qt$Unchecked` states, and a third indeterminate state specified by `Qt$Qt$PartiallyChecked`. These are stored in the `checkState` for setting and getting.

In addition to the inherited signals `clicked`, `pressed`, and `released`, The signal `stateChanged` is emitted. Callbacks are passed the state as an integer.

Groups of checkboxes

The `QButtonGroup` can be used to group together buttons, such as checkboxes, into logical units. The layout of the buttons is not managed by this widget. By default, the buttons are exclusive, like a radio button group, but this can be adjusted by passing `FALSE` to the method `setExclusive`. Buttons are added to the button group one-by-one through the method `addButton`. An optional ID can be passed in to identify the buttons, but it may be more convenient to work with the list of buttons returned by the `buttons` method.

When a button is clicked, pressed or released a signal is emitted, for example `buttonClicked` is when a button is clicked. The callback receives the button object.

Example 3.3: Using checkboxes to provide a filter
Filtering a data set by the levels of a factor is a familiar interface. This is commonly implemented by using a checkbox group with each level assigned to a toggle. Below we show how to do so for a cylinders variable, such is in the `Cars93` data set of the `MASS` package.

We begin by defining the levels and a widget to store our buttons.

```
data(Cars93, package="MASS")
cyls = levels(Cars93$Cylinders)
w <- Qt$QGroupBox("Cylinders:")
lyt <- Qt$QVBoxLayout()
```

```
w$setLayout(lyt)
```

We want to be able to select more than one button, so set the `exclusive` property accordingly.

```
bg <- Qt$QButtonGroup()
bg$setExclusive(FALSE)
```

Next we prepare a button for each value of `cyls`, add it to the layout and then the button group. Finally we initialize the buttons to all be checked.

```
sapply(seq_along(cyls), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cyls[i]))
  lyt$addWidget(button)
  bg$addButton(button, i)
})
sapply(bg$buttons(), function(i) i$setChecked(TRUE))
```

Our simple callback to the `buttonClicked` signal shows how to see which buttons were checked after the button was pressed. (The `buttonPressed` is called before the widget state reflects the button press.)

```
qconnect(bg, "buttonClicked", function(button) {
  checked <- sapply(bg$buttons(), function(i) i$checked)
  if(any(checked)) {
    ind <- Cars93$Cylinders %in% cyls[checked]
    print(sprintf("You've selected %d cases", sum(ind)))
  }
})
```

3.5 Radio groups

Radio buttons are created by the `QRadioButton` constructor, which inherits from `QAbstractButton`. Radio buttons are typically exclusive. To group buttons together, Qt links all buttons that share the same parent widget. Radio buttons, like other buttons, have a `text` and `icon` property. In addition, radio buttons have a checked or unchecked state which can be queried with `isChecked` and set with `setChecked`.

Example 3.4: Using a radio group to filter

Instead of a group of checkboxes, we might also filter through the exclusive selection offered by radio buttons.

First, we place the radio buttons into a list for easy manipulation. A `QButtonGroup` could also be used here, but we use an R-based approach for variety.

```
w <- Qt$QGroupBox("Weight:")
```

```
l <- list(Qt$QRadioButton("Weight < 3000", w),
          Qt$QRadioButton("3000 <= Weight < 4000", w),
          Qt$QRadioButton("4000 <= Weight", w)
          )
```

Next we define a layout for our radio button widgets. The buttons are children of `w` below, so are exclusive within that.

```
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

To add the widgets to the layout and set the first one to be checked we have the following:

```
sapply(l, function(i) lyt$addWidget(i))
l[[1]]$setChecked(TRUE)
```

The `toggled` signal is emitted twice when a button is clicked: once for when the check button is clicked, and once for the toggle of the previously checked button. Below, we condition on the value of `checked` to restrict to one call.

```
sapply(l, function(i) {
  qconnect(i, "toggled", function(checked) {
    if(checked) {
      ind <- which(sapply(l, function(i) i$isChecked()))
      print(sprintf("You checked %s.", l[[ind]]$text))
    }
  })
})
```

3.6 Sliders and spin boxes

Sliders and spin boxes are similar widgets used for selecting from a range of values. Sliders give the illusion of selecting from a continuum, whereas spinboxes offer a discrete choice, but underlying each is a selection from an arithmetic sequence. In Qt the two have similar method names.

Sliders

Sliders are implemented by `QSlider` a subclass of `QAbstractSlider`, which also provides functionality for scrollbars. Sliders in Qt are for selecting from integer valuves. To specify the range of possible values to select from the methods `setMinimum` and `setMaximum` are used (assuming integer values). Movement between the possible values is adjusted by `setSinglePageStep` with broader motion by `setPageStep`. (If we think of the arguments `from`, `to`, `by` of `seq` then these are the minimum, maximum and the single page step.) The actual value stored in the widget is found in the property `value`.

Sliders can be horizontal or vertical, the orientation can be changed by passing a `Qt::Orientation` value to the `setOrientation` method. To adjust the appearance to ticks for a slider, the method `setTickPosition` is used with values drawn from the `QSlider::TickPosition` enumeration (for example, `Qt$QSlider$TicksBelow`, `"TicksLeft"` or the default `"NoTicks"`). The method `setTickInterval` is used to specify an interval between the ticks.

The signal `valueChanged` is emitted when the slider is moved. It passes back the current value to a callback. The `sliderMoved` signal is similar, only the slider must be down, as when being dragged by a mouse.

Spin boxes

Spin boxes are derived from `QAbstractSpinBox` which provides the base class for `QSpinBox` (for integers), `QDoubleSpinBox` and `QDateTimeEdit` (not pursued here).

The methods have similar names as for sliders: `setMinimum`, `setMaximum`, and `setValue` have similar usages. The step size is provided by `setSingleStep`. The property `wrapping` can be set to `TRUE` to have the values wrap at the ends.

In Qt spinbuttons can have a prefix or suffix with the numbers. These are set by `setPrefix` or `setSuffix`.

As with sliders, the signal `valueChanged` is emitted when the spin button is changed,

Example 3.5: A range selector
We combine a slider and spinbox to make a range selection widget, offering the speedier movement of the slider, with the finer adjustments of the spin box.

Our slider construction sets values one-by-one.

```
sl <- Qt$QSlider()
sl$setMinimum(0)
sl$setMaximum(100)
sl$setSingleStep(1)
sl$setPageStep(5)
```

To style our slider we make it horizontal, set the tick position and interval through the following:

```
sl$setOrientation(Qt$Qt$Horizontal)
sl$setTickPosition(Qt$QSlider$TicksBelow)
sl$setTickInterval(10)
```

The basic spin box construction is relatively straightforward. We add a "%" suffix to make it seem like we are selecting a percent.

```
sp <- Qt$QSpinBox()
sp$setMinimum(0)
sp$setMaximum(100)
```

```
sp$setSingleStep(1)
sp$setSuffix("%")
```

To link the two widgets, we define callbacks for their `valueChanged` signal updating the other widget when there is a change. This could possibly cause an infinite loop, so we check if the suggested value is not equal to the current one before updating.

```
f <- function(value, obj) {
  if(! isTRUE(all.equal(value, obj$value)))
    obj$setValue(value)
}
qconnect(sp, "valueChanged", f, user.data=sp)
qconnect(sl, "valueChanged", f, user.data=sl)
```

3.7 Single-line text

As seen in previous examples, a widget for entering or displaying a single line of text is provided by the `QLineEdit` class. The `text`roperty holds the current value. The text may be set as the first argument to the constructor, or through the method `setText` (provided `readOnly` is `FALSE`). Text may be inserted through the `insert` method, replacing the currently selected text or inserting at the cursor. One can programatically position the cursor by index through the `setCursorPosition` method. As is typical, the index is 0 for the left most postion, 1 for between the first and second character, etc. The rightmost index can be found from `nchar(widget$text)`, say. The `setSelection` method takes two indices to indicate the left and right bounds of the selection.

When there is a selection, the methods `hasSelectedText` and `selectedText` are applicable. If `dragEnabled` is `TRUE` the selected text may be dragged and dropped on the appropriate targets.

If desired, it is possible to mask the displayed text with asterisks (common with passwords) by setting the `echoMode` property. Value are taken from the`QLineEdit::EchoMode` enumeration, e.g., `Qt$QLineEdit$Password`. If desired, the property `displayText` holds the displayed text.

Undo, redo The widget keeps an undo/redo stack. The methods `modified`, `isRedoAvailable`, `isUndoAvailable` are helpful in tracking if the text has changed and `undo` and `redo` can go through the changes.

The widget emits several different signals that are of use. The `cursorPositionChanged` signal is emitted as the cursor is moved. The old and new positions are passed along. The `selectionChanged` signal is emitted as the selection is updated. The `textChanged` signal is emitted when the text is changed. Any callback is passed the new text. Similarly for `textEdited`, the difference beting that this signal is not emitted when text is set by `setText`. The distinction

between the signals `editingFinished` and `returnPressed` is due to the former being called only if a valid entry is given.

Completion

Using Qt's `QCompleter` framework, a list of possible words can be presented for completion. The word list is generally sepecified by a model, but may also be specified as a character vector to the constructor. A simple usage is presented by example.

Example 3.6: Using completion on the `Qt` object

The `Qt` environment has many components. This example shows how completion can assist in exploring them by name. We use a form layout to arrange our two line edit widgets – one to gather a class name and one for method and property names.

```
w <- Qt$QWidget()
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Class name", c_name <- Qt$QLineEdit())
lyt$addRow("Method name", m_name <- Qt$QLineEdit())
```

The completer for the class is constructed just one. We use `ls` to list the components of the environment. We see that completions are set for a line edit widget through the `setCompleter`.

```
c_comp <- Qt$QCompleter(ls(Qt))
c_name$setCompleter(c_comp)
```

The completion for the methods depends on the class. As such, we update the completion when editing is finished for the class name.

```
qconnect(c_name, "editingFinished", function() {
  cl <- c_name$text
  val <- get(cl, envir=Qt)
  if(!is.null(val)) {
    m_comp <- Qt$QCompleter(ls(val()))
    m_name$setCompleter(m_comp)
  }
})
```

Masks and Validation

`QLineEdit` has various means to restrict and validate user input. The `maxLength` property can be set to restrict the number of allowed characters. To set a pattern for the possible answer, an input mask can be set through `setInputMask`. Input masks are specified through a string indicating a pattern. For example,

"999-99-9999" is for a U.S. Social Security number. The API for QLineEdit contains a full description.

A illustrated in Example 1.1, Qt also implements a validation framework where the value in the widget is validated before being committed. When a validator is set, using setValidator the method is called before the value is transferred from the GUI to the widget. The function can return one of three different states of validity "Acceptable" (i.e, Qt$QValidator$Acceptable), "Invalid", or the indeterminate "Intermediate". The function is passed the current value and the index of the cursor.

The validator must be an instance of a subclass of QValidator with a validate method. This requires constructing a subclass.

3.8 Multi-line text

Multi-line text is displayed and edited through the QTextEdit class. This widget allows for more than plain text. It may show rich text through HTML syntax including images, lists and tables.

As a basic text editor, the widget is a view for an underlying QTextDocument instance. The document can be used to retrieve that document, and setDocument to replace it. The method toPlainText is used to retrieve the text as plain text with a corresponding setPlainText for replacing the text. Text can also be added. The append method will append the text to the end of the document, insertPlainText will insert the text at the current cursor position replacing any selection, and paste will paste the current clipboard contents into the current cursor position. (Use copy and cut to replace the clipboards contents.) More complicated insertions can be handled through the QTextCursor class.

To illustrate

```
te <- Qt$QTextEdit()
te$setPlainText("The quick brown fox")
te$append("jumped over the lazy dog")
```

```
te$toPlainText()
```

```
[1] "The quick brown fox\njumped over the lazy dog"
```

Formatting By default, the widget will wrap text as entered. For use as a code editor, this is not desirable. The setLineWrapMode takes values from the enumeration QTextEdit::LineWrapMode, to control this. A value Qt$QTextEdit$NoWrap will turn off wrapping. When wrapping is enabled, one can control how with the enumeration QTextOption::WrapMode and the method setWordWrap.

The setAlignment method aligns the current paragraph with values from Qt::Alignment.

Formatting is managed by the QTextFormat class with subclasses such as QTextCharFormat, For formatting at the character level and QTextBlockFormat to format blocks of text. (A dcoument is comprised of blocks containing paragraphs etc.) There are methods to adjust the alignment, font properties, margins etc.

The text cursor The cursor position is returned from the position method of the text cursor, which is found through the textCursor method. The position is an integer indicating the index of the cursor if the buffer is thought of as a single string. To set the cursor position one first gets the text cursor, sets its position with setPosition, then sets the textedit's text cursor through setTextCursor.

For example, to move the cursor to the end can be done with

```
n <- nchar(te$toPlainText())
cursor <- te$textCursor()
cursor$setPosition(n)
te$setTextCursor(cursor)
```

Selections The text cursor hasSelection method indicates if a selection exists. If it does, the method selection returns a QTextDocumentFragment which also has a method toPlainText. A selection is determined by the position of the cursor and an anchor. The latter is found through anchor. However, to move the anchor is a bit more difficult. Basically, as the name suggests, you keep an anchor in place and move the position. The movePosition method's second argument is one of Qt$QTextCursor$KeepAnchor or Qt$QTextCursor$MoveAnchor. When the anchor is kept in place and the cursor moved a selection is defined. The movement of the position can be specified through the enumeration QTextCursor::MoveOperation with values like "Start", "End", "NextWord", etc.

To set the selection to include the first three words of the text, we have:

```
cursor <- te$textCursor()
cursor$movePosition(Qt$QTextCursor$Start) ## default is move anchor
cursor$movePosition(Qt$QTextCursor$WordRight, Qt$QTextCursor$KeepAnchor, 3)
te$setTextCursor(cursor)
```

Note the extra space at the end here:

```
te$textCursor()$selection()$toPlainText()
```

```
[1] "The quick brown "
```

Navigation The widget has a find method to move the selection to the next instance of a string. The enumeration QTextDocument::FindFlag can modify the search with values "FindBackward", "FindCaseSensitively" and "FindWholeWords".

For example, we can search through a standard typesetting string starting at the cursor point for the common word "qui" as follows:

```
te <- Qt$QTextEdit(LoremIpsum)          # some text
te$find("qui", Qt$QTextDocument$FindWholeWords)
```

```
[1] TRUE
```

```
te$textCursor()$selection()$toPlainText()
```

```
[1] "qui"
```

Signals There are several useful signals that are emitted by the widget. Some deal with changes: `cursorPositionChanged`, `selectionChanged`, `textChanged`, and `currentCharFormatChanged`. Others allow one to easily update any actions to reflect the state of the widget. These include `copyAvailable`, which passes in a boolean indicating "yes"; `redoAvailable`, which passes a boolean indicating availability; and similarly `undoAvailable`.

To get keystroke information, one can create a subclass and implement the `keyPressEvent` method, say.

Example 3.7: A tabbed text editor

This example shows how to combine the text edit with a notebook widget to create a widget for editing more than one file at a time.

We begin by defining a sub-class of `QTextEdit` so that we can define a few useful properties and method. In this example, we limit ourselves to just a few customizations of the edit widget.

```
qsetClass("TextEditSheet", Qt$QTextEdit, function(parent=NULL) {
  super(parent)

  this$nb <- NULL                       # text edit notebook
  this$filename <- NULL                 # name of file, tab

  setLineWrapMode(Qt$QTextEdit$NoWrap)
  setFontFamily("Courier")
})
```

The `nb` property stores the parent notebook widget allowing us to look this up when when we have the sheet object.

```
qsetMethod("setNb", TextEditSheet, function(nb) this$nb <- nb)
```

The file name is used for saving the sheet and for labeling the notebook tab. We define a getter and setter. The setter is also tasked with updating the tab label and illustrates how the `nb` property is employed for this.

```
qsetMethod("filename", TextEditSheet, function() filename)
qsetMethod("setFilename", TextEditSheet, function(fname) {
  this$filename <- fname
  ## update tab label
  notebook <- this$nb$notebook
  ind <- notebook$indexOf(this)
  notebook$setTabText(ind, basename(fname))
})
```

Next, we define a few methods for the sheet. First, one to save the file. We use the filename property for the suggested name to save as.

```
qsetMethod("saveSheet", TextEditSheet, function() {
  fname <- Qt$QFileDialog$getSaveFileName(this,
                         "Save file as...", filename())
  if(!is.null(fname)) {
    txt <- this$toPlainText()
    writeLines(strsplit(txt, "\n")[[1]], con=fname)
  }
})
```

We will see soon, that a method to test if a given sheet is the currently visible sheet is useful. This method returns a logical by comparing the current index of the notebook with the index of the sheet.

```
qsetMethod("isCurrentSheet", TextEditSheet, function() {
  notebook <- this$nb$notebook
  notebook$currentIndex == notebook$indexOf(this)
})
```

FInally we want to initialize a new sheet. This involves defining callbacks that update the actions in the parent notebook as appropriate. Later we define a minimal set of actions for this example and store them in the `tbactions` property of the parent notebook.

```
qsetMethod("initSheet", TextEditSheet, function() {

  qconnect(this, "redoAvailable", function(yes) {
    if(isCurrentSheet())
      this$nb$tbactions$redo$setEnabled(yes)
  })
  qconnect(this, "undoAvailable", function(yes) {
    if(isCurrentSheet())
      this$nb$tbactions$undo$setEnabled(yes)
  })
  qconnect(this, "selectionChanged", function() {
    hasSelection <- this$textCursor()$hasSelection()
    if(isCurrentSheet())
      sapply(c("cut","copy"), function(i)
             this$nb$tbactions[[i]]$setEnabled(hasSelection))
```

```
  })
  qconnect(this, "textChanged", function() {
    if(isCurrentSheet()) {
      mod <- this$document()$isModified()
      this$nb$tbactions$save$setEnabled(mod)
      }
  })
})
```

Next, we turn to the task of defining a notebook container. In fact we subclass `QMainWindow` so that we can add a toolbar. The notebook is then a property, as are the actions. Our constructor cusotmizes the notebook, sets up the actions and toolbar, then opens with a blank sheet.

```
qsetClass("TextEditNotebook", Qt$QMainWindow, function(parent=NULL) {
  super(parent)

  ## properties
  this$tbactions <- list()
  this$notebook <- Qt$QTabWidget()

  notebook$setTabsClosable(TRUE)
  notebook$setUsesScrollButtons(TRUE)
  qconnect(notebook, "tabCloseRequested",
           function(ind) notebook$removeTab(ind))
  qconnect(this$notebook, "currentChanged", function(ind) {
    if(ind > 0)
      updateActions()
  })
  setCentralWidget(notebook)

  initActions()
  makeToolbar()
  newSheet()
})
```

For our example, we implement a basic set of methods. This one maps down from the parent notebook widget to a sheet.

```
qsetMethod("currentSheet", TextEditNotebook, function() {
  this$notebook$currentWidget()
})
```

This method is used to open a new sheet, either a blank one or some initial text.

```
qsetMethod("newSheet", TextEditNotebook,
           function(title="..Scratch..", str=NULL) {

             a <- TextEditSheet()          # a new sheet
```

```
            a$setNb(this)                # set parent notebook
            a$initSheet()                # initialize the sheet

            this$notebook$addTab(a, "")  # add to the notebook
            ind <- this$notebook$indexOf(a)
            this$notebook$setCurrentIndex(ind)

            if(!is.null(str))            # set text if present
              a$setPlainText(str)
            a$setFilename(title)         # also updates tab
})
```

To open a file in a new sheet, we use a standard dialog to get the filename to open, then call `newSheet`.

```
qsetMethod("openSheet", TextEditNotebook, function() {
  dlg <- Qt$QFileDialog(this, "Select a file...", getwd())
  dlg$setFileMode(Qt$QFileDialog$ExistingFile)
  if(dlg$exec()) {
    fname <- dlg$selectedFiles()
    txt <- paste(readLines(fname), collapse="\n")
    newSheet(fname, txt)
  }
})
```

We have several actions possible in our GUI, such as the standard cut, copy and paste. We define them for the parent notebook, but the actions primarily work at the sheet level. The `initActions` is called to set up the actions. We don't show the entire method, as it is repetitive, but is primarily of this type:

```
actions <- list()                        # hold the actions
## make an action for open. This is TextEditNotebook instance
actions$open <- Qt$QAction("open", this)
actions$open$setShortcut(Qt$QKeySequence("Open"))
qconnect(actions$open, "triggered", function(obj) obj$openSheet(), user.data=this)
## ... etc. ...
this$tbactions <- actions
```

Whenever a new sheet is shown, the state of the actions should reflect that sheet. The `updateActions` method does this task. The constructor has a callback for this method whenever the sheet changes.

```
qsetMethod("updateActions", TextEditNotebook, function() {
  cur <- currentSheet()
  if(is.null(cur))
    return()

  a <- this$tbactions
  a$redo$setEnabled(FALSE)
```

```
  a$undo$setEnabled(FALSE)
  a$cut$setEnabled(cur$textCursor()$hasSelection())
  a$copy$setEnabled(cur$textCursor()$hasSelection())
  a$paste$setEnabled(cur$canPaste())

})
```

Finally, the `makeToolbar` method, which is not shown, simply maps the actions to a toolbar. Our GUI might also benefit from a menu bar, an exercise left for the reader.

Context menus Context menus may be added to the text edit widget, provided you create a subclass. The following example shows how to add to the standard context menu, which is available by the `createStandardContextMenu` method. The key is to redefine the `contextMenuEvent` method.

```
qsetClass("QTextEditWithMenu", Qt$QTextEdit)
#
qsetMethod("contextMenuEvent", QTextEditWithMenu, function(e) {
  m <- this$createStandardContextMenu()
  if(this$textCursor()$hasSelection()) {
    curVal <- this$textCursor()$selection()$toPlainText()
    comps <- utils:::matchAvailableTopics(curVal)
    comps <- setdiff(comps, curVal)
    if(length(comps) > 0 && length(comps) < 10) {
      m$addSeparator()                  # add actions
      sapply(comps, function(i) {
        a <- Qt$QAction(i, this)
        qconnect(a, "triggered", function(checked) {
          this$insertPlainText(i)
        })
        m$addAction(a)
      })
    }
  }
  m$exec(e$globalPos())
})
te <- QTextEditWithMenu()
```

4

Widgets using the MVC framework

4.1 Model View Controller implementation in Qt

The model-view-controller architecture adds a complexity to widgets that is paid in order to create more flexible and efficient use of resources. Keeping the model separate from the view allows multiple views for the same data. It also allows views to be more responsive when there are large data sets involved. The basic MVC architecture has a controller to act as a go-between for a model and its views. In Qt, this is different, an item in a model has a delegate that allows the user to see and perhaps edit the item's data within a view. Although custom delagates can be defined, we are content here to use those provided by Qt.

We discuss in the following various views: comboboxes, list views, table views and tree views. These are all familiar. The primary difficulty with using backend models is the specification and interaction with the model. For the simplest usages, Qt provides a set of convenience classes where the items and views are coupled.

Models in Qt are derived from the `QAbstractItemModel` class, although there are standard sub-classes for list, table and heirarchical models (`QStringListModel`, `QAbstractListModel`, `QAbstractTableModel`, and `QStandardItemModel`). In addition, the `qtbase` package provides the `qdataFrameModel` constructor to provide an mapping from a data frame to an item model.

A model is comprised of items, each having an associated index. Items have associated data which may vary based on the role (Table **??**) the item is playing. For example, one role describes the data for display whereas another role describes the data for editing. This allows, for example, a numeric item, to be displayed with just a few digits, but edited with all of them. Both descriptions, and others, are stored in the item's data and set through the `setData` method. For the `qdataFrameModel` constructor these roles are encoded as additional, specially named, columns in the data frame.

An item has several properties that are described by the `Qt::ItemFlag` enumeration. These indicate whether an item is selectable (`ItemIsSelectable`),

Table 4.1: Partial list of roles that an item can hold data for and the class of the data.

Constant	Description
`DisplayRole`	How data is displayed (`QString`)
`EditRole`	Data for editing (`QString`)
`ToolTipRole`	Displayed in tooltip (`QString`)
`StatusTipRole`	Displayed in status bar (`QString`)
`SizeHintRole`	Size hint for views (`QSize`)
`DecorationRole`	(`QColor`, `QIcon`, `QPixmap`)
`FontRole`	Font for default delegate (`QFont`)
`TextAlignmentRole`	Alignment for default delegate (`Qt::AlignmentFlag`)
`BackgroundRole`	Background for default delegate (`QBrush`)
`ForegroundRole`	Foreground for default delegate (`QBrush`)
`CheckStateRole`	Indicates checked state of item (`Qt::CheckState`)

editable (`ItemIsEditable`), enabled (`ItemIsEnabled`), checkable (`ItemIsCheckable`) etc. The `flags` returns the flag recording combinations of these.

4.2 The item-view classes

The use of a model separate from a view can be relatively complicated. As such, Qt provides classes that couple a standard model and view. Defining the model then is much easier. In the case of a combobox and a list view the model can be specified simply through a character vector.

Comboboxes

The `QComboBox` class implements the combobox widget. The `QListWidget` class implements a basic list view. Both do similar things – allow selection from a list of possible vlaues, although differently. As such, their underlying methods are mostly similar although their implementations have a different class inheritance. We discuss here the basic usage for comboboxes. Both widgets are views for an underlying model and we restrict ourselves, for now, to the simplest choice of these models.

By basic usage we mean setting a set of values for the user to choose from, and for comboboxes optionally allowing them to type in a new value. In the simplest use, we can define the items in their model through a character vector through the method `addItems`. (The items must be character, so numeric vectors must be coerced. This works through a convesion into a `QStringList` object.) One can remove items by index with `removeItem`. To allow a user to enter a value not in this list, the property `editable` can be set to `TRUE`.

Once values are entered the property `currentIndex` holds the 0-based index of the selected value. This will be -1 when no selection is made, and may be meaningless if the user can edit the values. Use `setCurrentIndex` to set the popup value by index (-1 to leave unselected). The `count` method returns the number of items available.

The property `currentText` returns the current text. This will be a blank string if there is no current index. There is no corresponding `setCurrentText`, rather one can use the `findText` method to get the index of the specified string. There is an option to adjust how finding occurs through the enumeration `Qt::MatchFlags`.

The signal `activated` is emitted when the user chooses an item. When activated, the item index is passed to the callback. This signal is emitted when the user finishes editing of the text by the return key. The `highlighted` signal is emitted when the popup is engaged and the user mouses over an entry. For editable comboboxes, the signal `editTextChanged` is emitted after each change to the text.

Example 4.1: An example of one combobox updating another
This example shows how one combobox, to select a region in the U.S., is used to update another, which lists states in that region. We will use the following data frame for our data., which we split into a list.

```
df <- data.frame(name=state.name, region=state.region,
                 highlight=state.x77[,1], stringsAsFactors=FALSE)
l <- split(df, df$region)
```

Our `region` combobox is loaded with the state regions. When one is selected, the callback for `activated` will first remove any items in the `state` combobox, then add in the appropriate states. The `ind` index is used to determine which region.

```
region <- Qt$QComboBox()
state <- Qt$QComboBox()
region$addItems(names(l))
region$setCurrentIndex(-1)              # no selection
qconnect(region, "activated", function(ind) {
  state$clear()
  state$addItems(l[[ind+1]]$name)       # add
})
```

The state combobox shows how the `highlighted` signal can be employed. In this case, information about the highlighted state is placed in the window's title. Not really a great choice, but sufficient for this example.

```
qconnect(state, "highlighted", function(ind) {
  pop <- l[[region$currentText]][ind + 1, "highlight"]
  w$setWindowTitle(sprintf("Popuplation is %s", pop))
})
```

Finally, we use a form layout to organize the widgets.

```
w <- Qt$QGroupBox("Two comboboxes")
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Region:", region)
lyt$addRow("State:", state)
```

A list widget

The `QListWidget` provides an easy-to-use widget for displaying a set of items for selection. It uses an item-based model for its data. The `QListView` widget provides a more general framework with different backend models. As with comboboxes, we can populate the items directly from a character vector through the `addItems` method. However, here we mostly focus on interacting with the widget through the item model.

The items in a `QListWidget` instance are of the `QListWidgetItem` class. New items can be constructed directly through the constructor. The first argument is the text and the optional second argument a parent `QListWidget`. If no parent is specified, the the item may be added through the methods `addItem`, or `insertItem` where the row to insert is specified by index.

`QListWidgetItems` can have their text specified at construction or through the method `setText` and optionally have an icon set through the `setIcons`. There are also methods to set a status bar tip or a tooltip.

The method `takeItem` is used to remove items specified by their index, `clear` will remove all of them.

Once an item is added to list widget it can be referenced several ways. The currently selected item is returned by `currentItem`, whereas `currentRow` returns the current row by index, and `currentIndex` returns a `QModelIndex` instance (with a method `row` to get the index). As well, any item may be referenced by row index through `item` or position (say within an event handler) by `itemAt`. One can search for the items with the `findItems` method, which returns a list of items. An optional second argument uses the `Qt::MatchFlags` enumeration to adjust how matches are made, for example `Qt$Qt$MatchRegExp` to match by regular expression.

Selection By default, single selection mode is enabled. This can be adjusted through the `setSelectionMode` argument by specifying a value in `QAbstractItemView::SelectionMode`, such as `SingleSelection` or `ExtendedSelection`. Extended selection allows the user to extend the current selection by simultaneously pressing the control key or the shift key (selecting all items between the current selection and the newly selected item).

To retrieve the selected values, the method `selectedItems` will return the items in a list.

Setting an item to be selected is done through `setCurrentItem` The first argument is the item, the optional second argument one of the `QItemSelectionModel::SelectionFlag` enumeration. If specified as `Qt$QItemSelectionModel$Select` (the default) the item will be selected, but other choices are possible such as `"Deselect"` or `"Toggle"`.

Checkable items The underlying items may be checkable. This is initiated by setting an initial check state (`setCheckState`) with a value from `"Checked"` (`Qt$Qt$Checked`), `"Unchecked"` or `"PartiallyChecked"`. For example, we can populate a list widget and set the values unchecked with.

```
w <- Qt$QListWidget()
w$addItems(state.name)
sapply(1:w$count, function(i)
       w$item(i-1)$setCheckState(Qt$Qt$Unchecked))
```

Then, after checking a few we can get the state along the lines of:

```
sapply(1:8, function(i) as.logical(w$item(i-1)$checkState()))
```

```
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
```

This uses the fact that the enumeration for `"Unchecked"` is 0 and `"Checked"` is 2.

Signals There are several signals that are emitted by the widget. Chief among them are `itemActivated`, which is emitted when a user clicks on an item or presses the activation key. The latter is what distinguishes it from the `itemClicked` signal. For capturing double clicks there is `itemDoubleClicked`. For these three, the undelying item is passed to the callback. The `itemSelectionChanged` signal is emitted when the underlying selection is changed.

Example 4.2: Filtering example
We illustrate the widget with a typical filtering example, where a user types in values to narrow down the available choices. We begin by setting up our widgets.

```
w <- Qt$QGroupBox("Filtering example")
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Filter:", f <- Qt$QLineEdit())      # define f
lyt$addRow("State:", lw <- Qt$QListWidget())    # define lw
lyt$addRow("", b <- Qt$QPushButton("Click me")) # define b
```

For convenience, we use a built-in data set for our choices. We populate the list widget and add a tooltip indicating the area.

```
for(i in state.name) {
```

```
  item <- Qt$QListWidgetItem(i, lw)             # populate
  txt <- sprintf("%s has %s square miles", i, state.x77[i, "Area"])
  item$setToolTip(txt)
}
```

For this example, we allow multiple selection.

```
lw$setSelectionMode(Qt$QAbstractItemView$ExtendedSelection)
```

The following callback updates the displayed items so that only ones matching the typed in string are displayed. Rather than compare each item to the matched items, we simply hide them all then unhide those that match.

```
qconnect(f, "textChanged", function(str) {
  matching <- lw$findItems(str, Qt$Qt$MatchStartsWith)
  sapply(seq_len(lw$count), function(i) lw$item(i-1)$setHidden(TRUE))
  sapply(matching, function(i) i$setHidden(FALSE))
})
```

The following shows how we can grab the selected values.

```
qconnect(b, "pressed", function() {
  vals <- sapply(lw$selectedItems(), function(i) i$text())
  print(vals)
})
```

Example 4.3: Combining a combobox and list widget to select a variable name
This example shows how we can combine a combobox and a list widget to select a variable name from a data frame. Here we select a value by dragging it. A such we need to define a sub-class of `QListWidget` to implement the `mousePressEvent`.

```
qsetClass("DraggableListWidget", Qt$QListWidget,
          function(parent=NULL) {
            super(parent)
            this$df <- NULL
          })
```

The property `df` holds the name of the dataframe that will be selected through a combobox. Here is a method to set the value.

```
qsetMethod("setDf", DraggableListWidget,
           function(df) this$df <- df)
```

For drag and drop we show how to serialize an arbitrary R object to pass through to the drop target. We pass in a list of the data frame name and the selected variable name. The method `setData` takes a MIME type (which we arbitrarily define) and a value. This value will be retrieved by the `data` method and we can then call `unserialize`.

```
qsetMethod("mousePressEvent", DraggableListWidget, function(e) {
  item <- itemAt(e$pos())
  val <- list(df=this$df, var=item$text())

  md <- Qt$QMimeData()
  md$setData("R/serialized-data", serialize(val, NULL))

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)

  drag$exec()
})
```

With this, we know create a widget to hold the combobox and the list box. The constructor creates the widgets, lays them out, initializes the data sets then sets a handler to update the variable list when the dataframe selector does.

```
qsetClass("VariableSelector", Qt$QWidget, function(parent=NULL) {
  super(parent)

  this$dfcb <- Qt$QComboBox()
  this$varList <-  DraggableListWidget()

  lyt <- Qt$QVBoxLayout()
  lyt$addWidget(dfcb)
  lyt$addWidget(varList)
  varList$setSizePolicy(Qt$QSizePolicy$Expanding,
                        Qt$QSizePolicy$Expanding)
  setLayout(lyt)

  updateDataSets()
  qconnect(dfcb, "activated", function(ind) {
    updateVarList(dfcb$currentText)
  })
})
```

Our method to update the data frame choice is a bit convoluted as we try to keep the currently selected data frame, if possible.

```
qsetMethod("updateDataSets", VariableSelector, function() {
  curVal <- this$dfcb$currentText
  this$dfcb$clear()
  x <- ls(envir=.GlobalEnv)
  dfs <- x[sapply(x, function(i)
                  is.data.frame(get(i, inherits=TRUE)))]
  if(length(dfs)) {
    this$dfcb$addItems(dfs)
    if(is.null(curVal) || !curVal %in% dfs) {
```

```
      this$dfcb$setCurrentIndex(-1)
      updateVarList(NULL)
    } else {
      this$dfcb$setCurrentIndex(which(curVal == dfs))
      updateVarList(curVal)               # curVal NULL, or a name
    }
  }
})
```

Finally, we need to update the list of variables to reflect the state of the combo box selection. Here we define a helper method to display an appropriate icon based on the class of the variable.

```
getIconFile <- function(x) UseMethod("getIconFile")
getIconFile.default <- function(x)
  Qt$QIcon(system.file("images/numeric.gif", package="gWidgets"))
getIconFile.factor <- function(x)
  Qt$QIcon(system.file("images/factor.gif", package="gWidgets"))
getIconFile.character <- function(x)
  Qt$QIcon(system.file("images/character.gif", package="gWidgets"))
```

This method populates the variable list to reflect the indicated data frame. As items are automatically drag enabled, we do not need to add anything more here, as we've implemented the `mousePressEvent`.

```
qsetMethod("updateVarList", VariableSelector, function(df=NULL) {
  this$varList$setDf(df)
  this$varList$clear()
  if(!is.null(df)) {
    d <- get(df)
    sapply(names(d), function(i) {
      item <- Qt$QListWidgetItem(i, this$varList)
      item$setIcon(getIconFile(d[,i]))
    })
  }
})
```

A table widget

The `QTableWidget` class provides a widget for displaying tabular data in an item-based approach, similar to `QListWidget`. The `QTableView` widget is more flexible, but also more demanding, as it has the ability to have different data models (which can be much faster with large tables). As such, only if your needs are not too complicated will this widget will be a good choice.

The dimensions of the table must be set prior to adding items. The methods `setRowCount` and `setColumnCount` are used.

The `QTableWidget` class has a built in model that is populated item by item. Items are of class `QTableWidgetItem` and are created first, then in-

serted into the widget, by row and column, through the method `setItem` (These operations are not vectorized and can be slow for large tables.) Items can be removed by row-and-column index with `takeItem`. The item can be reinserted. The `clear` method will remove all items, even headers items, whereas, `clearContents` will leave the headers. Both keep the dimensions.

As with `QListWidget`, items have various properties that can be adjusted. The text can be specified to the constructor, or set through `setText`. Text alignment is specified through `setTextAlignment`. The font may be configured through `setFont`. The methods `setBackground` and `setForeground` are used to adjust the colors.

Items may also have icons (`setIcon`), tooltips (`setToolTip`), and statusbar tips (`setStatusTip`).

Similar to `QListWidget`, `QTableWidgetItem` instances are returned by the method `item`, with a specification of row and column; and by `itemAt`, only with a specification of a position. The `findItems` method will return a list of items matching a string. Also, there is the method `currentItem`, to return the currently selected item. From an item, its column and row can be found through its methods `column` and `row`

Item flags As mentioned, items may have several different properties: are they editable, draggable, ...? To specify, one sets an items flags with values taken from the `Qt::ItemFlag` enumeration. The possible values are: `"NoItemFlags"`, `"ItemIsSelectable"`, `"ItemIsEditable"`, `"ItemIsDragEnabled"`, `"ItemIsDropEnabled"`, `"ItemIsUserCheckable"`, `"ItemIsEnabled"`, and `"ItemIsTriState"` (has three check states). To make an item checkable, one must first set the check state. By default, the widget is selectable, editable, drag and droppable, checkable and enabled. To remove a flag, one can specify all the ones they want, or use integer arithmetic and subtract. E.g., to remove the editable attribute one has this possibility:

```
item <- Qt$QTableWidgetItem("Set not editable")
if(item$flags() & Qt$Qt$ItemIsEditable)
  item$setFlags(item$flags() - Qt$Qt$ItemIsEditable)
```

Headers Columns may have headers (horizontal ones, rows have vertical headers). These are set all at once by specifying a character vector to `setHorizontalHeaderLabels`, or can be set with an item by `setHorizontalHeaderItem`. The header itself, of class `QHeaderView`, is returned by `horizontalHeader`. Headers have the method `setVisible` to toggle their visibility. To make the last column stretch to fill the available space is specified through the header vier method `setStretchLastSection` with a value of `TRUE`.

Otherwise, to specify the width of a column programmatically the method `setColumnWidth` is available. One specifies the column, then the width in pixels.

Sorting and Filtering This widget can have its rows sorted by the values of a column through the method `sortItems`. One specifies the column by index, and an order. The default is `"AscendingOrder"`, the alternative is `Qt$Qt$DescendingOrder`. Sorting should be done after the table is populated with items.

Rows and columns can be hidden through the table widget's methods `setRowHidden` and `setColumnHidden`. This can be used for filtering purposes without redrawing the table.

Selection For the table widget one can easily select rows, columns, blocks and even combinations thereof. An underlying selection model implements selection, but the `QTableWidget` class provides an easier interface. The currently selected items are returned as a list through the method `selectedItems`.

A given cell may be selected by index (row then column) through the method `setCurrentCell`, or if the item instance is known by `setCurrentItem`. In addition, an optional selection flag from the enumeration `QItemSelectionModel::SelectionFlag` can be specified with values among `"Select"`, to add the item to the selection; `"Clear"`, to clear all selection; `"Toggle"`, to toggle the specified item; `"Rows"`, to extend the selection to the enclosing row; and similarly for `"Columns"`. For an item itself, the method `setSelection` takes a logical value to indicate the selection state.

The current selection can be cleared by selecting an item with the `"Clear"` attribute, or by grabbing the underlying selection model and calling its `clearSelection` method. E.g, something like:

```
tbl$selectionModel()$clearSelection()
```

Signals The `QTableWidget` class has a number of signals it emits. They mostly come in pairs: "cell" ones passing in the row and column index and "item" ones passing in an item reference. For example, `cellClicked` and `itemClicked`, both called when a cell is clicked. Also there are `cellDoubleClicked`, `cellEntered`, `cellPressed`, `currentCellChanged` and `cellChanged` (with similar "item" ones). Also of interest is the `itemSelectionChanged` which is called when the selection changes.

Example 4.4: Selection of variables

This example shows how to use the table widget to select variable names in a data set. The `QListWidget` can also be used for this, but with the table widget we can add more detail in other columns We begin by placing the table widget into a box layout.

```
tbl <- Qt$QTableWidget()
tbl$setSizePolicy(Qt$QSizePolicy$Expanding, Qt$QSizePolicy$Expanding)
```

```
lyt <- Qt$QVBoxLayout()
lyt$addWidget(tbl)
```

We use a QDialogButtonBox to hold our two buttons.

```
db <- Qt$QDialogButtonBox()
lyt$addWidget(db)
db$addButton(Qt$QDialogButtonBox$Ok)
db$addButton(Qt$QDialogButtonBox$Cancel)
qconnect(db, "accepted", function() runCommand())
qconnect(db, "rejected", function() w$hide())
```

We assume we have a data frame, df, with two columns – one for the variable name and one for detail, in this case the class of the variable.

We populate our table first by setting its dimensions and then setting the headers. We stretch the last column and set the column width of the first to accomodate the longest variable name in our example.

```
tbl$clear()
tbl$setRowCount(nrow(df))
tbl$setColumnCount(ncol(df))
tbl$setHorizontalHeaderLabels(c("Variable", "Class"))
tbl$horizontalHeader()$setStretchLastSection(TRUE)
tbl$setColumnWidth(0, 200)
tbl$verticalHeader()$setVisible(FALSE)
```

Each row in this example has two items, one for the checkable item holding the variable name and one for detail. Here we add row by row.

```
for(i in 1:nrow(df)) {
  item <- Qt$QTableWidgetItem(df[i,1])
  item$setCheckState(Qt$Qt$Unchecked)
  tbl$setItem(i-1, 0, item)

  item <- Qt$QTableWidgetItem(df[i,2])
  item$setTextAlignment(Qt$Qt$AlignLeft)
  tbl$setItem(i-1, 1, item)
}
```

Finally, this command is called when the "Ok" button is pressed. For simplicity we do two passes through the data to grab the text and the check state (which we must coerce to get a logical value).

```
runCommand <- function() {
  n <- tbl$rowCount
  x <- sapply(seq_len(n), function(i) tbl$item(i-1, 0)$text())
  ind <- sapply(seq_len(n), function(i) tbl$item(i-1, 0)$checkState())
  print(x[as.logical(ind)])
}
```

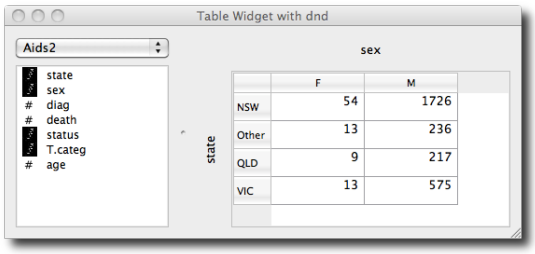

Figure 4.1: A table widget to display contingency tables and a means to specify the variables through drag and drop.

Example 4.5: A drag and drop interface to `xtabs`

This examples uses a table widget to display the output from `xtabs`. To specify the variable names, we use the `VariableSelector` class defined in Example 4.3. That provides a means to drag the variable. We next define a drop label, similar to that defined in Example **??**. We show the major differences only. Our constructor is similar, but adds three properties: a function to call for the drop event, a value beyond the text, and an angle for text rotation. We define accessors for these properties, but do not show them here.

```
qsetClass("DropLabelRotation", Qt$QLabel, function(parent=NULL) {
  super(parent)

  this$dropFunction <- function(obj, data) {}
  this$value <- list(df=NULL, var=NULL)
  this$angle <- 0L

  setAcceptDrops(TRUE)
  setAlignment(Qt$Qt$AlignHCenter | Qt$Qt$AlignVCenter)
})
```

To handle the drag events we override the methods `dragEnterEvent`, `dragLeaveEvent`, and `dropEvent`. Only the latter is shown. For that we check for data with MIME type `R/serialized-data`. In that case, we call `unserialize` on the data and then the function in the `dropFunction` property.

```
qsetMethod("dropEvent", DropLabelRotation, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  md <- e$mimeData()
  if(md$hasFormat("R/serialized-data")) {
    data <- unserialize(md$data("R/serialized-data"))
    dropFunction(this, data)
```

```
    setBackgroundRole(Qt$QPalette$Window)
    e$acceptProposedAction()
  }
})
```

For this application we want to be able to show text in the drop label in a rotated state. The following allows this. We override the `paintEvent` event if the `angle` property is non-zero, otherwise we call the parent class's paint event. The chapter on `qtpaint` will cover the techniques used in `drawRotatedText`. In short we translate the origin to the center of the label's rectangle, rotate coordinates by the angle, then place the text within this new rectangle/

```
qsetMethod("paintEvent", DropLabelRotation, function(e) {
  if(!this$angle) {
    super("paintEvent", e)
  } else {
    p <- Qt$QPainter()
    p$begin(this)
    this$drawRotatedText(p)
    p$end()
    update()
  }
})
qsetMethod("drawRotatedText", DropLabelRotation, function(p) {
  w <- this$width; h <- this$height
  p$save()
  p$translate(w/2, h/2)
  p$rotate(this$angle)
  p$drawText(qrect(-h, -w, h, w), Qt$Qt$AlignCenter, this$text)
  p$restore()
})
```

Our main widget consists of three widgets: two drop labels for the contingency table and a table widget to show the output. This could be extend to include a third variable for three-way tables, but we leave that exercise for the interested reader. The constructor simply calls two methods to be defined next.

```
qsetClass("XtabsWidget", Qt$QWidget, function(parent=NULL) {
  super(parent)

  initWidgets()
  initLayout()
})
```

```
Class 'R:.GlobalEnv::XtabsWidget' with 318 public methods
```

We have three widgets to initialize. The `xlabel` is pretty basic: we define a widget and set a drop function. For the `ylabel` we also adjust the

rotation and constrain the width based on the font size. The latter is necessary, as otherwise the label width reflects the length of the dropped text. The `clearLabels` just sets the text and initializes the values for the labels. It is not shown.

```
qsetMethod("initWidgets", XtabsWidget, function() {
  ## make Widgets
  this$xlabel <- DropLabelRotation()
  this$ylabel <- DropLabelRotation()
  pt <- this$ylabel$font$pointSize()
  this$ylabel$setMinimumWidth(2*pt); this$ylabel$setMaximumWidth(2*pt)
  this$ylabel$setRotation(-90L)

  this$tw <- Qt$QTableWidget()

  clearLabels()

  f <- function(obj, data) {
    obj$setValue(data)
    obj$setText(data$var)
    this$makeTable()
  }
  this$xlabel$setDropFunction(f)
  this$ylabel$setDropFunction(f)
})
```

We don't show the layout code for this widget, it is a simple application of `QGridLayout`, but do show how we create the call to xtabs to create the data after the drop events.

```
qsetMethod("makeTable", XtabsWidget, function() {
  df <- this$xlabel$getValue()$df
  if(is.null(xVar <- this$xlabel$getValue()$var)) {
    out <- NULL
  } else if(is.null(yVar <- this$ylabel$getValue()$var)) {
    f <- formula(sprintf("~ %s", xVar))
    out <- xtabs(f, data=get(df))
  } else {
    f <- formula(sprintf("~ %s + %s", yVar, xVar))
    out <- xtabs(f, data=get(df))
  }
  if(!is.null(out))
    updateTableWidget(out)
})
```

Finally, for the `XtabsWidget` class we define a method to update the table widget. We have two cases and the code repeats itself a bit. The basic tasks are to set the dimensions of the table, update the header labels, and then populate the table.

```
qsetMethod("updateTableWidget", XtabsWidget, function(out) {
  tw <- this$tw
  tw$clear()
  ndims <- length(dim(out))
  if(ndims == 1) {
    tw$setRowCount(1)
    tw$setColumnCount(dim(out))

    tw$setHorizontalHeaderLabels(names(out))
    tw$horizontalHeader()$setVisible(TRUE)
    tw$verticalHeader()$setVisible(FALSE)
    sapply(seq_along(out), function(i) {
      item <- Qt$QTableWidgetItem(as.character(out[i]))
      item$setTextAlignment(Qt$Qt$AlignRight)
      tw$setItem(0, i-1, item)
    })

  } else if(ndims == 2) {
    tw$setRowCount(dim(out)[1])
    tw$setColumnCount(dim(out)[2])

    tw$setHorizontalHeaderLabels(colnames(out))
    tw$setVerticalHeaderLabels(rownames(out))
    tw$horizontalHeader()$setVisible(TRUE)
    tw$verticalHeader()$setVisible(TRUE)

    for(i in 1:dim(out)[1])
      for(j in 1:dim(out)[2]) {
        item <- Qt$QTableWidgetItem(as.character(out[i,j]))
        item$setTextAlignment(Qt$Qt$AlignRight)
        tw$setItem(i-1, j-1, item)
      }
  }
})
```

Figure 4.1 shows the widget placed within a splitter window after a few variables have been dragged.

A Tree Widget

The `QTreeWidget` provides a tree view based on a coupled tree model. It is used in a manner similar to the table widget, however, one must adjust for the heiarchical nature of the data it can display. The widget shows a specific number of columns, which is specified via `setColumnCount`. The column headers are set through `setHeaderLabels` by specifying a vector of column names.

Tree Widget Items The widget organizes itself through items, with each item having 1 or more columns. The items in the tree widget are instances of the `QTreeWidgetItem` class. An item displays a value in each column. The method `setText` takes two arguments, an integer specifying a column, and a value to display. Similarly, other such item methods require one to specify the column. These methods include: `setTextAlignment`, `setFont`, `setToolTip`, `setStatusTip`, and `setIcon`. To retrieve these values, the method `data` takes a column number and a role, such as `Qt$Qt$DisplayRole` to get the text.

Heirarchy A tree is used to represent a heirarchy. The items have several methods related to this. Each item has a `parent` method pointing to the parent item, or `NULL` if a top-level item. (In the latter case, the tree widget's `invisibleRootItem` method returns the parent item.) The method `child` returns the child item, specified by index. The index of a given item in the parent is returned by `indexOfChild`. The total number of children for an item is returned by `childCount`. There is not a `nextSibling` method, but one could do something like:

```
nextSibling <- function(tr, i) {
  parent <- i$parent()
  if(is.null(parent))
    parent <- tr$invisibleRootItem()
  ind <- parent$indexOfChild(item)
  n <- parent$childCount()
  if(ind + 1 < n)
    parent$child(ind + 1)                    # 1 already added
  else
    NULL
}
```

New child items are added with either with `addChild`, adding at the end; or `insertChild` adding the child item at the specified index. Child items are removed by the `removeChild` method or the `takeChild`, if specifying by index. The latter returns the item to be reparented if desired.

The view can be made to show if a child is present by calling `setChildIndicatorPolicy` with a value of `Qt$QTreeWidgetItem$ShowIndicator` (or one of `DontShowIndicator` or `DontShowIndicatorWhenChildless`).

When an item has children and an indicator is shown, the view may be expanded by clicking on this indicator. The item method `isExpanded` returns `TRUE` when an item is expanded. This state can be set via `setExpanded`. The methods `setDisabled` and `setHidden` are also available to disable or hide an item.

Adding items to a tree The children of the invisible root item are the "top-level" items. These are added through `addTopLevelItem` or the `insertTopLevelItem`

methods, the latter requiring an index to specify where. The basic idea is that each top-level item is prepared, along with its heirarchy, then added through these methods.

Selection The currently selected item is returned by `currentItem` and may be specified by item through `setCurrentItem`. For multiple selection, see the discussion on selection models 4.3.

Signals The widget emits several signals, most passing the item and the column to the handler. Some useful signals are `itemClicked`, `itemDoubleClicked`, `itemExpanded`, and `itemActivated` (for clicking or the Enter key).

Example 4.6: A workspace browser

This example shows how to use the tree widget item to display a snapshot of the current workspace. Each object in the workspace maps to an item, where recursive objects with names will have their components represented in a heiarchical manner.

When representing objects in a workspace, we need to decide if an object has been changed. To do this, we use the `digest` function from the `digest` package to store a simple means to compare a current object with a past one. To store this information, we create a subclass of `QTreeWidgetItem` with a `digest` property.

```
require(digest)
qsetClass("OurTreeWidgetItem", Qt$QTreeWidgetItem, function(parent=NULL) {
  super(parent)
  this$digest <- NULL
})
qsetMethod("digest", OurTreeWidgetItem, function() digest)
qsetMethod("setDigest", OurTreeWidgetItem, function(value) {
  this$digest <- value
})
```

This subclass will be used for top-level items in our tree. We define the following to create an item from a variable name. Since we call this function recursively, we have an argument indicating of the item is a top-level item.

We could modify the display of each item to suit more sophisticated tastes, e.g., icons, but for this example stop by adding a column for the class of the object.

As we use the name of an object to display it, when checking for recursive structures, we also check that they have a `names` attribute.

```
makeItem <- function(varname, obj=NULL, toplevel=FALSE) {
  if(is.null(obj))
    obj <- get(varname, envir=.GlobalEnv)
```

```
  if(toplevel) {
    item <- OurTreeWidgetItem()
    item$setDigest(digest(obj))
  } else {
    item <- Qt$QTreeWidgetItem()
  }

  item$setText(0, varname)
  item$setText(1, paste(class(obj), collapse=", "))
  ## icons, fonts children...

  if(is.recursive(obj) && !is.null(attr(obj, "names"))) {
    item$setChildIndicatorPolicy(Qt$QTreeWidgetItem$ShowIndicator)
    for(i in names(obj)) {
      newItem <- makeItem(i, obj[[i]])
      item$addChild(newItem)
    }
  }
  item
}
```

Next we define a few simple functions to add, remove and replace a top-level item.

```
addItem <- function(tr, varname) {
  newItem <- makeItem(varname, toplevel=TRUE)
  tr$addTopLevelItem(newItem)
}
```

We can remove an item by index with the `takeTopLevelItem` method, but here it is more convenient to specify the item to remove.

```
removeItem <- function(tr, item) {
  root <- tr$invisibleRootItem()
  root$removeChild(item)
}
```

```
replaceItem <- function(tr, item, varname) {
  removeItem(tr, item)
  addItem(tr, varname)
}
```

Our main function is one that checks the current workspace and updates the values in the tree widget accordingly. This could be set on a timer to be called periodically, or called in response to user input.

As can be seen, we consider three cases: items no longer in the workspace to remove, new items to add, and finally items that may be new. Here is where we check the digest value to see if they need to be replaced. Rather

than fuss with inserting new items, we simply add them to the end and then sort the values.

```
updateTopLevelItems <- function(tr) {
  objs <- ls(envir=.GlobalEnv)

  curObjs <- lapply(seq_len(tr$topLevelItemCount), function(i) {
    item <- tr$topLevelItem(i-1)
    list(item=item, value=item$data(0, role=Qt$Qt$DisplayRole))
  })
  cur <- sapply(curObjs, function(i) i$value)
  names(curObjs) <- cur

  ## we have three types here:
  removeThese <- setdiff(cur, objs)
  maybeSame <- intersect(cur, objs)
  addThese <- setdiff(objs, cur)

  tr$setUpdatesEnabled(FALSE)
  for(i in removeThese) {
    item <- curObjs[[i]]$item
    removeItem(tr, item)
  }


  for(i in maybeSame) {
    obj <- get(i, envir=.GlobalEnv)
    if(digest(obj) != curObjs[[i]]$item$digest()) {
      replaceItem(tr, curObjs[[i]]$item, i)
    }
  }

  for(i in addThese)
    addItem(tr, i)

  tr$sortItems(0, Qt$Qt$AscendingOrder)
  tr$setUpdatesEnabled(TRUE)
}
```

Now we define some functions for traversing the heiarchical tree structure. First we define a function to find the index of an item, be it a top-level item or an ancestor of one.

```
indexFromItem <- function(tr, item) {
  parent <- item$parent()
  if(is.null(parent))
    parent <- tr$invisibleRootItem()
  parent$indexOfChild(item) + 1
}
```

We find it convenient to think in terms of a path rather than an item, either a vector of indices or names. The `index` argument below decides what to return. the basic approach here is to walk backwards by calling the `parent` method until it is `null`.

```
pathFromItem <- function(tr, item, index=TRUE) {
  getVal <- function(item)
    ifelse(index, indexFromItem(tr, item),
           item$data(0, role=0))

  path <- getVal(item)
  while(!is.null(item <- item$parent())) {
    path <- c(getVal(item), path)
  }
  path
}
```

This function reverses the last. It takes a path, specified by indices, and finds the item if possible.

```
itemFromPath <- function(tr, path) {
  item <- tr$invisibleRootItem()
  for(i in path) {
    item <- item$child(i)
    if(is.null(item))
      return(NULL)
  }
  item
}
```

As an illustration, we show how one can use `findItems` to return a list of items matching a query. In this case, we pass in a regular expression to match against the `class` column (column 1.)

```
matchClass <- function(tr, classname) {
  allItems <- sapply(seq_len(tr$topLevelItemCount), function(i) {
    tr$topLevelItem(i-1)
  })
  matched <- tr$findItems(classname, Qt$Qt$MatchRegExp, 1) # column 1
  sapply(allItems, function(i) i$setHidden(TRUE))
  sapply(matched, function(i) i$setHidden(FALSE))
}
```

Finally we show how this can be used. First, we set the column count and the header labels to match our `makeItem` function above.

```
tr <- Qt$QTreeWidget()
tr$setColumnCount(2)                    # name, class
tr$setHeaderLabels(c("Name", "Class"))
```

This call initializes the display.

```
updateTopLevelItems(tr)
```

An typical signal handler, to be called when an item is clicked, is illustrated next. Here we print both the path and a summary of the object in the workspace the path points to.

```
qconnect(tr, "itemClicked", function(item, column) {
  path <- pathFromItem(tr, item, index=FALSE)
  obj <- get(path[1], envir=.GlobalEnv)
  if(length(path) > 1)
    obj <- obj[[path[-1]]]              # get object
  str(obj)
  print(path)
  updateTopLevelItems(tr)
})
```

4.3 Item models and their views

In this section, we consider models and their views decoupled from each other. We focus our discussion to tabular data, although we note that Qt provides various means to provide models for heiarchical data.

Using a data frame for a model

For tabular data a model can be made quite easily with the constructor `qdataFrameModel` provided by `qtbase`. This function maps a data frame to the model, returning an instance of class `DataFrameModel` a subclass of `QAbstractTableModel`.

The default role is for display of the data. Other roles can be defined by adding additional columns with a specific naming convention. A column with name `a` can have its role information specified with another column with name `.a.ROLE` where `ROLE` is one of available roles, such as `"decoration"`, `"edit"`, `"toolTip"`, `"statusTip"`, or `"textAlignment"`.

The following illustrates a basic usage. We use a `QTableView` instance for a view. The model is connected to the view through its `setModel` method.

Example 4.7: Using `qdataFrameModel` to list variables in a data frame

This example will show how to use `qdataFrameModel` to create a means to select a variable from a data frame. We will use a built-in data set for this.

```
nms <- names(Cars93)
d <- data.frame("Variables"=nms, stringsAsFactors=FALSE)
```

We will add an icon for decoration. To do so, we create a list of icons for each of the classes represented in the variables we have. Here we use some images from the `gWidgets` package.

```
icons <- sapply(c("factor", "numeric"), function(i) {
  Qt$QPixmap(system.file(sprintf("images/%s.gif",i),
                         package="gWidgets"))
})
icons$integer <- icons$numeric
```

To add the icons as a decoration, we simply assign to the appropriately named component of our data frame. Data frames can store such data, although the default print method does not work.

```
d$.Variables.decoration <- sapply(nms, function(i)
                                  icons[class(Cars93[[i]])])
```

Similarly, we set an informative tooltip for each variable:

```
d$.Variables.toolTip <- sapply(nms, function(i) capture.output(str(Cars93[[i]])))
```

We now define a model and associate it with a view. In this example we use a `QListView` instance, although in most uses of this model we would use the `QTableView` class. We discuss views later in Section 4.3.

```
model <- qdataFrameModel(d)
view <- Qt$QListView()
view$setModel(model)
```

For the list view, we have a few properties to adjust, in this case we highlight alternating rows as follows:

```
view$setAlternatingRowColors(TRUE)
```

Table Models

The `qdataFrameModel` creates a subclass of `QAbstractTableModel`, itself a subclass of `QAbstractItemModel`. One can subclass `QAbstractTableModel` to provide a custom model for a view. This may be useful in different instances, for example, if the cell columns need to be computed dynamically or if editing of cells is desired.

Before discussing methods that must be implemented, we mention some useful inherited methods. For a table model, items are organized by row and column. This allows one to easily refer to an item by index. The inherited `index` method uses 0-based row and column arguments to return a `QModelIndex` instance. In turn, this index instance has methods `row` and `column` for getting back the coordinate information.

The `match` method can be used to return items in a column that match a specific string for a given role. The column is specified by a model index and the role by one of the `Qt::ItemDataRole` enumeration. The optional third argument is one of the `Qt:::MatchFlag` enumeration, specifying how the matching is to occur. Matches are returned as a list of indices.

The `sort` method can be used to sort the data by the values in a column. The column is specified by its index, and the order is one of `Qt$Qt$AscendingOrder` or `Qt$Qt$DescendingOrder`.

Required methods The basic implementation of a subclass must provide the methods `rowCount`, `columnCount`, and `data`. The first two describe the size of the table for any views, the third describes to the view how to display data for a given role. The signature for the `data` method is `(index, role)`. The `index` value, an instance of `QModelIndex`, has methods `row` and `column`, whereas `role` is one of the roles defined in the `Qt::ItemDataRole` enumeration. This method should return the appropriate value for the given role. For example, if one is displaying numeric data, the `display` role might format the numeric values (showing a fixed number of digits say), yet the `edit` role might display all the digits so accuracy is not lost. If a role is not implemented, a value of `NULL` should be returned, as this is mapped to a null instance of `QVariant`.

One may also implement the `headerData` method to return the appropriate data to display in the header for a given role. The main one being `Qt$Qt$DisplayRole`.

Editable Models For editable models, one must implement the `flags` method to return a flag containing `ItemIsEditable` and the `setData` method. This has signature `(index, value, role)` where `value` is a character string containing the edited value. When a value is updated, one should call the `dataChanged` method to notify the views that a portion of the model is changed. This method takes two indices, together specifying a rectangle in the table.

To provide for resizable tables, `Qt` requires one to call some (of several such) functions so that any connected views can be notified. For example, an implemented `insertColumns` should call `beginInsertColumns` before adding the column to the model and then `endInsertColumns` just after.

Table views

A table model is typically displayed through the `QTableView` widget, although one can use the model – first column only – with `QListView` or even `QComboBox`. A custom view is also possible, as illustrated later. The table view widget inherits from `QAbstractItemView` which provides the method `setModel` to link the underlying model with a view. This link passes along information about the model being changed back to the view to process and connects the delegates that allow one to edit values in the view and have the changes propogate back to the model.

Once, connected, the table view can have its properties adjusted to control its appearance. For example, columns (similarly rows) may be hidden or

shown through the `setColumnHidden` method. Their widths can be adjusted by the mouse, or through `setColumnWidth`. The grid style can be adjusted through `setGridStyle`.

The view may allow sorting of the underlying model. This is enabled through `setSortingEnabled`. The method `sortByColumn` can be called specifying the column and a sort order (e.g. `Qt$Qt$AscendingOrder`).

Headers The table view has headers, horizontal ones for the columns and vertical ones for the rows. The methods `verticalHeader` and `horizontalHeader` return instances of the `QHeaderView` class. This class has many methods. We mention `setHidden`, to suppress the header display; `showSortIndicator`, to display a sort arrow; and `setStretchLastSection` to strech the last section to fill the space. For tree views, this is `TRUE` for horizontal headers but not for table views.

Selection Models The type of selection possible is determined by the `selectionMode` of the view, the options for which are enumerated in `QAbstractItemView::SelectionMode`. Typical values are `SingleSelection` or `ExtendedSelection`, which allows one to extend their selection by pressing the Control or Shift keys while selecting.

For abstract item views, the underlying selection is handled by a selection model, which is found in the `selectionModel` property. A `QItemSelectionModel` instance can return the current selection by rows through `selectedRows` (with an argument to specify the column to check), columns (`selectedColumns`) or by index with `selectedIndexes`. In each case, a list of model indices is returned. The methods `row` and `column` are useful then. The selection model allows one to specify if the selection will apply to the entire row or column of if selection is cell by cell. This is done through the selection flags (`QItemSelectionModel::SelectionFlag`)

The selection can be updated programatically through the `setCurrentIndex` method. The index and a selection flag must be given. The latter enumerated in `QItemSelectionModel::SelectionFlag`, with values such as `Select`, to select the item; `Deselect`, `Clear`; `Toggle`; and `Rows` and `Columns` to select the entire range.

The selection model emits a few signals, notably `selectionChanged` is emitted when a new selection is made.

Example 4.8: Using a custom model to edit a data frame
This example shows how to create a custom model to edit a data frame. The performance is much less responsive than that provided by `qdataFrameModel`, as the bulk of the operations are done at the R level. We speed things up a bit, by placing column headers into the first row of the table, and not in the expected place. The numerous calls to a `headerData` method implemented

in R proved to be quite slow.

Our basic constructor simply creates a `dataframe` property and sets the data frame.

```
qsetClass("DfModel", Qt$QAbstractTableModel, function(df=data.frame(V1=character(0)), parent=
  super(parent)
  ## properties
  this$dataframe <- NULL

  setDf(df)
})
```

We need accessors for the `dataframe` property. When setting a new one, we call the `dataChanged` method to notify any views of a change.

```
qsetMethod("Df", DfModel, function() dataframe)
qsetMethod("setDf", DfModel, function(df) {
  this$dataframe <- df
  dataChanged(index(0, 0), index(nrow(df), ncol(df)))
})
```

All subclasses of `QAbstractTableModel` need to implement a few methods, here we do `rowCount` and `columnCount` using the dimensions of the current data frame.

```
qsetMethod("rowCount", DfModel, function(index) nrow(this$Df()) + 1)
qsetMethod("columnCount", DfModel, function(index) ncol(this$Df()))
```

The `data` method is the main method to implement. We wish to customize the data display based on the class of the variable represented in a column. We implement this with S3 methods. We define several. For example, to change the display role we have the following.

```
displayRole <- function(x, row, context) UseMethod("displayRole")
displayRole.default <- function(x, row, context)
  sprintf("%s", x[row])
displayRole.numeric <- function(x, row, context)
  sprintf("%.2f", x[row])
displayRole.integer <- function(x, row, context)
  sprintf("%d", x[row])
```

We see that numeric values are formatted to have 2 decimal points. By setting the display role, we can store numeric data with all its digits, so that we can edit the entire value, but have values formatted along the decimal point. The `context` argument is shown to indicate how to make the number of digits context sensitive.

Our `data` method has this basic structure (we avoid showing the cases for all the different roles.

```
qsetMethod("data", DfModel, function(index, role) {
```

```
  d <- this$Df()
  row <- index$row()
  col <- index$column() + 1

  if(role == Qt$Qt$DisplayRole) {
    if(row > 0)
      displayRole(d[,col], row)
    else
      names(d)[col]
  } else if(role == Qt$Qt$EditRole) {
    if(row > 0)
      as.character(d[row, col])
    else
      names(d)[col]
  } else {
    NULL
  }
})
```

To allow the user to edit the values we need to override the `flags` method to return `ItemIsEditable` in the flag, so that any views are aware of this ability.

```
qsetMethod("flags", DfModel, function(index) {
  if(!index$isValid()) {
    return(Qt$Qt$ItemIsEnabled)
  } else {
    curFlags <- super("flags", index)
    return(curFlags | Qt$Qt$ItemIsEditable)
  }
})
```

To edit cells we need to implement a method to set data once edited The trick is that value is a character, so we coerce to the right type for the column it is in. We do this with an S3 method. For example we have this (among others):

```
fitIn <- function(x, value) UseMethod("fitIn")
fitIn.default <- function(x, value) value
fitIn.numeric <- function(x, value) as.numeric(value)
```

The `setData` method is responsible for taking the value from the delegate and assigning it into them model, we use the `fitIn` function unless it is a column header.

```
qsetMethod("setData", DfModel, function(index, value, role) {

  if(index$isValid() && role == Qt$Qt$EditRole) {
    d <- this$Df()
    row <- index$row()
```

```
    col <- index$column() + 1

    if(row > 0) {
      x <- d[, col]
      d[row, col] <- fitIn(x, value)
    } else {
      names(d)[col] <- value
    }
    this$dataframe <- d
    dataChanged(index, index)

    return(TRUE)
  } else {
     super("setData", index, value, role)
  }
})
```

For a data frame editor, we may wish to extend the API for our table of items to be R specific. For example, this method allows one to replace a column of values.

```
qsetMethod("setColumn", DfModel, function(col, value) {
  ## pad with NA if needed
  n <- nrow(this$Df())
  if(length(value) < n)
    value <- c(value, rep(NA, n - length(value)))
  value <- value[1:n]
  d <- this$Df()
  d[,col] <- value
  this$dataframe <- d          # only notify about this column
  dataChanged(index(0, col-1), index(rowCount()-1, col-1))
  return(TRUE)
})
```

We implement a method similar to the `insertColumn` method, but specific to our task. Since we may add a new column, we call the "begin" and "end" methods to notify any views.

```
qsetMethod("addColumn", DfModel, function(name, value) {
  d <- this$Df()
  if(name %in% names(d)) {
    return(setColumn(min(which(name == names(d))), value))
  }

  beginInsertColumns(Qt$QModelIndex(), columnCount(), columnCount())
  d <- this$Df()
  d[[name]] <- value
  this$dataframe <- d
  endInsertColumns()
```

```
  return(TRUE)
})
```

To test this out, we define a model and set it in a view.

```
model <- DfModel(mtcars)
view <- Qt$QTableView()
view$setModel(model)
```

Finally, we customize the view by defining what triggers the delegate for editing and hide the row and column headers.

```
triggerFlag <- Qt$QAbstractItemView$DoubleClicked |
               Qt$QAbstractItemView$SelectedClicked |
               Qt$QAbstractItemView$EditKeyPressed
view$setEditTriggers(triggerFlag)
view$verticalHeader()$setHidden(TRUE)
view$horizontalHeader()$setHidden(TRUE)
```

Custom Views

One can write a custom view for a model. In the example, we show how to connect a label's text to the values in a column of a model. The base class for a view of an item model is `QAbstractItemView`. This class has many methods – that can be overridden – to cover keyboard, mouse, scrolling, selection etc. Here we focus on just enough for our example.

Example 4.9: A label that updates as a model is updated
This example shows how to create a simple custom view for a table model. We use the `DfModel` class developed in Example 4.8 to provide the model.

Our custom view class will be a sub-class of `QAbstractItemView`, although we don't consider the individual items below. Our goal is a simple illustration, where we provide a label with text summarizing the mean of the values in the first column of the model.

In the constructor we define a label property and call our `setModel` method.

```
qsetClass("CustomView", Qt$QAbstractItemView, function(parent=NULL) {
  super(parent)
  this$label <- Qt$QLabel()
  label$setMinimumSize(400,400)
  setModel(NULL)
})
```

The label has an accessor function which we use in our private `setLabelValue` method defined below. Here we get the model from the custom view.

```
qsetMethod("label", CustomView, function() label)
qsetMethod("setLabelValue", CustomView, function() {
  m <- this$model()
```

```
  if(is.null(m)) {
    txt <- "No model"
  } else {
    df <- m$Df()
    xbar <- mean(df[,1])
    txt <- sprintf("The mean is %s", xbar)
  }
  label$setText(txt)
},
           access="private")
```

The main method call is when the data is changed. Here we update the label text when so notified.

```
qsetMethod("dataChanged", CustomView, function(upperIndex,lowerIndex) {
  setLabelValue()
})
```

To initialize the label text, we add a call to `setLabelValue` when the model is set.

```
qsetMethod("setModel", CustomView, function(model) {
  super("setModel", model)
  setLabelValue()
})
```

Finally, to test this out, we put the view in a simple GUI along with an instance of an editable data frame. When we modify the model through that, the label's text updates accordingly.

```
model <- DfModel()
model$setDf(mtcars)
view <- Qt$QTableView()
view$setModel(model)
view$setEditTriggers(Qt$QAbstractItemView$DoubleClicked)
cv <- CustomView()
cv$setModel(model)
w <- Qt$QWidget()
lyt <- Qt$QHBoxLayout()
w$setLayout(lyt)
lyt$addWidget(view)
lyt$addWidget(cv$label())
```

5

Qt paint