Designing Qt-Style C++ APIs

by Matthias Ettrich

We have done substantial research at Trolltech into improving the Qt development experience. In this article, I want to share some of our findings and present the principles we've been using when designing Qt 4, and show you how to apply them to your code.

Designing application programmer interfaces, APIs, is hard. It is an art as difficult as designing programming languages. There are many different principles to choose from, many of which tend to contradict each other.

Computer science education today puts a lot of emphasis on algorithms and data structures, with less focus on the principles behind designing programming languages and frameworks. This leaves application programmers unprepared for an increasingly important task: the creation of reusable components.

Before the rise of object-oriented languages, reusable generic code was mostly written by library vendors rather than by application developers. In the Qt world, this situation has changed significantly. Programming with Qt is writing new components all the time. A typical Qt application has at least some customized components that are reused throughout the application. Often the same components are deployed as part of other applications. KDE, the K Desktop Environment, goes even further and extends Qt with many add-on libraries that implement hundreds of additional classes.

But what constitutes a good, efficient C++ API? What is good or bad depends on many factors -- for example, the task at hand and the specific target group. A good API has a number of features, some of which are generally desirable, and some of which are more specific to certain problem domains.

Six Characteristics of Good APIs

An API is to the programmer what a GUI is to the end-user. The 'P' in API stands

for "Programmer", not "Program", to highlight the fact that APIs are used by programmers, who are humans.

We believe APIs should be minimal and complete, have clear and simple semantics, be intuitive, be easy to memorize, and lead to readable code.

- Be minimal: A minimal API is one that has as few public members per class and as few classes as possible. This makes it easier to understand, remember, debug, and change the API.
- Be complete: A complete API means the expected functionality should be there. This can conflict with keeping it minimal. Also, if a member function is in the wrong class, many potential users of the function won't find it.
- Have clear and simple semantics: As with other design work, you should apply the principle of least surprise. Make common tasks easy. Rare tasks should be possible but not the focus. Solve the specific problem; don't make the solution overly general when this is not needed. (For example, QMimeSourceFactory in Qt 3 could have been called QImageLoader and have a different API.)
- Be intuitive: As with anything else on a computer, an API should be intuitive. Different experience and background leads to different perceptions on what is intuitive and what isn't. An API is intuitive if a semi-experienced user gets away without reading the documentation, and if a programmer who doesn't know the API can understand code written using it.
- Be easy to memorize: To make the API easy to remember, choose a consistent and precise naming convention. Use recognizable patterns and concepts, and avoid abbreviations.
- Lead to readable code: Code is written once, but read (and debugged and changed) many times. Readable code may sometimes take longer to write, but saves time throughout the product's life cycle.

Finally, keep in mind that different kinds of users will use different parts of the API. While simply using an instance of a Qt class should be intuitive, it's reasonable to expect the user to read the documentation before attempting to subclass it.

The Convenience Trap

It is a common misconception that the less code you need to achieve something, the better the API. Keep in mind that code is written more than once but has to be understood over and over again. For example,

```
QSlider *slider = new QSlider(12, 18, 3, 13, Qt::Vertical,
                              0, "volume");
```

is much harder to read (and even to write) than

```
QSlider *slider = new QSlider(Qt::Vertical);
slider->setRange(12, 18);
slider->setPageStep(3);
```

```
slider->setValue(13);
slider->setObjectName("volume");
```

The Boolean Parameter Trap

Boolean parameters often lead to unreadable code. In particular, it's almost invariably a mistake to add a bool parameter to an existing function. In Qt, the traditional example is repaint(), which takes an optional bool parameter specifying whether the background should be erased (the default) or not. This leads to code such as

```
widget->repaint(false);
```

which beginners might read as meaning, "Don't repaint!"

The thinking is apparently that the bool parameter saves one function, thus helping reducing the bloat. In truth, it adds bloat; how many Qt users know by heart what each of the next three lines does?

```
widget->repaint();
widget->repaint(true);
widget->repaint(false);
```

A somewhat better API might have been

```
widget->repaint();
widget->repaintWithoutErasing();
```

In Qt 4, we solved the problem by simply removing the possibility of repainting without erasing the widget. Qt 4's native support for double buffering made this feature obsolete.

Here come a few more examples:

```
widget->setSizePolicy(QSizePolicy::Fixed,
                      QSizePolicy::Expanding, true);
textEdit->insert("Where's Waldo?", true, true, false);
QRegExp rx("moc_*.c??", false, true);
```

An obvious solution is to replace the bool parameters with enum types. This is what we've done in Qt 4 with case sensitivity in QString. Compare:

```
str.replace("%USER%", user, false);          // Qt 3
str.replace("%USER%", user, Qt::CaseInsensitive); // Qt 4
```

Static Polymorphism

Similar classes should have a similar API. This can be done using inheritance where it makes sense -- that is, when run-time polymorphism is used. But polymorphism also happens at design time. For example, if you exchange a QListBox with a QComboBox, or a QSlider with a QSpinBox, you'll find that the similarity of APIs makes this replacement very easy. This is what we call "static polymorphism".

Static polymorphism also makes it easier to memorize APIs and programming patterns. As a consequence, a similar API for a set of related classes is sometimes better than perfect individual APIs for each class.

The Art of Naming

Naming is probably the single most important issue when designing an API. What should the classes be called? What should the member functions be called?

General Naming Rules

A few rules apply equally well to all kinds of names. First, as I mentioned earlier, do not abbreviate. Even obvious abbreviations such as "prev" for "previous" don't pay off in the long run, because the user must remember which words are abbreviated.

Things naturally get worse if the API itself is inconsistent; for example, Qt 3 has activatePreviousWindow() and fetchPrev(). Sticking to the "no abbreviation" rule makes it simpler to create consistent APIs.

Another important but more subtle rule when designing classes is that you should try to keep the namespace for subclasses clean. In Qt 3, this principle wasn't always followed. To illustrate this, we will take the example of a QToolButton. If you call name(), caption(), text(), or textLabel() on a QToolButton in Qt 3, what do you expect? Just try playing around with a QToolButton in *Qt Designer*:

- The name property is inherited from QObject and refers to an internal object name that can be used for debugging and testing.
- The caption property is inherited from QWidget and refers to the window title, which has virtually no meaning for QToolButtons, since they usually are created with a parent.
- The text property is inherited from QButton and is normally used on the button, unless useTextLabel is true.
- The textLabel property is declared in QToolButton and is shown on the button if useTextLabel is true.

In the interest of readability, name is called objectName in Qt 4, caption has become windowTitle, and there is no longer any textLabel property distinct from text in QToolButton.

Naming Classes

Identify groups of classes instead of finding the perfect name for each individual class. For example, All the Qt 4 model-aware item view classes are suffixed with `View` (QListView, QTableView, and QTreeView), and the corresponding item-based classes are suffixed with `Widget` instead (QListWidget, QTableWidget, and QTreeWidget).

Naming Enum Types and Values

When declaring enums, we must keep in mind that in C++ (unlike in Java or C#), the enum values are used *without* the type. The following example shows illustrates the dangers of giving too general names to the enum values:

```
namespace Qt
{
    enum Corner { TopLeft, BottomRight, ... };
    enum CaseSensitivity { Insensitive, Sensitive };
    ...
};

tabWidget->setCornerWidget(widget, Qt::TopLeft);
str.indexOf("$(QTDIR)", Qt::Insensitive);
```

In the last line, what does `Insensitive` mean? One guideline for naming enum types is to repeat at least one element of the enum type name in each of the enum values:

```
namespace Qt
{
    enum Corner { TopLeftCorner, BottomRightCorner, ... };
    enum CaseSensitivity { CaseInsensitive,
                           CaseSensitive };
    ...
};

tabWidget->setCornerWidget(widget, Qt::TopLeftCorner);
str.indexOf("$(QTDIR)", Qt::CaseInsensitive);
```

When enumerator values can be OR'd together and be used as flags, the traditional solution is to store the result of the OR in an `int`, which isn't type-safe. Qt 4 offers a template class QFlags<T>, where `T` is the enum type. For convenience, Qt provides typedefs for the flag type names, so you can type `Qt::Alignment` instead of QFlags<Qt::AlignmentFlag>.

By convention, we give the enum type a singular name (since it can only hold one flag at a time) and the "flags" type a plural name. For example:

```
enum RectangleEdge { LeftEdge, RightEdge, ... };
```

```
typedef QFlags<RectangleEdge> RectangleEdges;
```

In some cases, the "flags" type has a singular name. In that case, the enum type is suffixed with Flag:

```
enum AlignmentFlag { AlignLeft, AlignTop, ... };
typedef QFlags<AlignmentFlag> Alignment;
```

Naming Functions and Parameters

The number one rule of function naming is that it should be clear from the name whether the function has side-effects or not. In Qt 3, the const function QString::simplifyWhiteSpace() violated this rule, since it returned a QString instead of modifying the string on which it is called, as the name suggests. In Qt 4, the function has been renamed QString::simplified().

Parameter names are an important source of information to the programmer, even though they don't show up in the code that uses the API. Since modern IDEs show them while the programmer is writing code, it's worthwhile to give decent names to parameters in the header files and to use the same names in the documentation.

Naming Boolean Getters, Setters, and Properties

Finding good names for the getter and setter of a bool property is always a special pain. Should the getter be called checked() or isChecked()? scrollBarsEnabled() or areScrollBarEnabled()?

In Qt 4, we used the following guidelines for naming the getter function:

- Adjectives are prefixed with is-. Examples:
  - isChecked()
  - isDown()
  - isEmpty()
  - isMovingEnabled()

  However, adjectives applying to a plural noun have no prefix:
  - scrollBarsEnabled(), not areScrollBarsEnabled()
- Verbs have no prefix and don't use the third person (-s):
  - acceptDrops(), not acceptsDrops()
  - allColumnsShowFocus()
- Nouns generally have no prefix:
  - autoCompletion(), not isAutoCompletion()
  - boundaryChecking()

  Sometimes, having no prefix is misleading, in which case we prefix with is-:
  - isOpenGLAvailable(), not openGL()
  - isDialog(), not dialog()

  (From a function called dialog(), we would normally expect that it returns a QDialog *.)

The name of the setter is derived from that of the getter by removing any `is` prefix and putting a `set` at the front of the name; for example, `setDown()` and `setScrollBarsEnabled()`. The name of the property is the same as the getter, but without the `is` prefix.

Pointers or References?

Which is best for out-parameters, pointers or references?

```
void getHsv(int *h, int *s, int *v) const
void getHsv(int &h, int &s, int &v) const
```

Most C++ books recommend references whenever possible, according to the general perception that references are "safer and nicer" than pointers. In contrast, at Trolltech, we tend to prefer pointers because they make the user code more readable. Compare:

```
color.getHsv(&h, &s, &v);
color.getHsv(h, s, v);
```

Only the first line makes it clear that there's a high probability that `h`, `s`, and `v` will be modified by the function call.

Case Study: QProgressBar

To show some of these concepts in practice, we'll study the QProgressBar API of Qt 3 and compare it to the Qt 4 API. In Qt 3:

```
class QProgressBar : public QWidget
{
    ...
public:
    int totalSteps() const;
    int progress() const;

    const QString &progressString() const;
    bool percentageVisible() const;
    void setPercentageVisible(bool);

    void setCenterIndicator(bool on);
    bool centerIndicator() const;

    void setIndicatorFollowsStyle(bool);
    bool indicatorFollowsStyle() const;

public slots:
    void reset();
```

```
    virtual void setTotalSteps(int totalSteps);
    virtual void setProgress(int progress);
    void setProgress(int progress, int totalSteps);

protected:
    virtual bool setIndicator(QString &progressStr,
                              int progress,
                              int totalSteps);
    ...
};
```

The API is quite complex and inconsistent; for example, it's not clear from the naming that reset(), setTotalSteps(), and setProgress() are tightly related.

The key to improve the API is to notice that QProgressBar is similar to Qt 4's QAbstractSpinBox class and its subclasses, QSpinBox, QSlider and QDial. The solution? Replace progress and totalSteps with minimum, maximum and value. Add a valueChanged() signal. Add a setRange() convenience function.

The next observation is that progressString, percentage and indicator really refer to one thing: the text that is shown on the progress bar. Usually the text is a percentage, but it can be set to anything using the setIndicator() function. Here's the new API:

```
virtual QString text() const;
void setTextVisible(bool visible);
bool isTextVisible() const;
```

By default, the text is a percentage indicator. This can be changed by reimplementing text().

The setCenterIndicator() and setIndicatorFollowsStyle() functions in the Qt 3 API are two functions that influence alignment. They can advantageously be replaced by one function, setAlignment():

```
void setAlignment(Qt::Alignment alignment);
```

If the programmer doesn't call setAlignment(), the alignment is chosen based on the style. For Motif-based styles, the text is shown centered; for other styles, it is shown on the right hand side.

Here's the improved QProgressBar API:

```
class QProgressBar : public QWidget
{
    ...
public:
    void setMinimum(int minimum);
```

```
    int minimum() const;
    void setMaximum(int maximum);
    int maximum() const;
    void setRange(int minimum, int maximum);
    int value() const;

    virtual QString text() const;
    void setTextVisible(bool visible);
    bool isTextVisible() const;
    Qt::Alignment alignment() const;
    void setAlignment(Qt::Alignment alignment);

public slots:
    void reset();
    void setValue(int value);

signals:
    void valueChanged(int value);
    ...
};
```

How to Get APIs Right

APIs need quality assurance. The first revision is never right; you must test it. Make use cases by looking at code which uses this API and verify that the code is readable.

Other tricks include having somebody else use the API with or without documentation and documenting the class (both the class overview and the individual functions).

Documenting is also a good way of finding good names when you get stuck: just try to document the item (class, function, enum value, etc.) and use your first sentence as inspiration. If you cannot find a precise name, this is often a sign that the item shouldn't exist. If everything else fails *and* you are convinced that the concept makes sense, invent a new name. This is, after all, how "widget", "event", "focus", and "buddy" came to be.
