Table of content

English Русский

API Design Principles

One of Qt's most reputed merits is its consistent, easy-to-learn, powerful API. This document tries to summarize the know-how we've accumulated on designing Qt-style APIs. Many of the guidelines are universal; others are more conventional, and we follow them primarily for consistency with existing APIs.

Although these guidelines are aimed primarily at public APIs, you are encouraged to use the same techniques when designing internal APIs, as a courtesy to your fellow developers.

You may also be interested to read Jasmin Blanchette's Little Manual of API Design [chaos.troll.no]

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.

In his Qt Quarterly 13 article about API design [doc.qt.nokia.com], Matthias tells us he believes that 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.

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 QProgressBar with a QSlider, or a QString with a QByteArray, 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.

In general, in Qt, we prefer to rely on static polymorphism than on actual inheritance when there's no compelling reason to do otherwise. This keeps the number of public classes in Qt down and makes it easier for new Qt users to find their way around in the documentation.

Good: QDialogButtonBox and QMessageBox have similar APIs for dealing with buttons (addButton (), setStandardButtons(), etc.), without publicly inheriting from some "QAbstractButtonBox" class.

Bad: QAbstractSocket is inherited both by QTcpSocket and QUdpSocket, two classes with very different modes of interaction. Nobody seems to have ever used (or been able to use) a QAbstractSocket pointer in a generic and useful way.

Dubious: QBoxLayout is the base class of QHBoxLayout and QVBoxLayout. Advantage: Can use a QBoxLayout and call setOrientation() in a toolbar to make it horizontal/vertical. Disadvantages: One extra class, and possibility for users to write ((QBoxLayout *)hbox)->setOrientation(Qt::Vertical), which makes little sense.

Property-Based APIs

Newer Qt classes tend to have a "property-based API". E.g.:

```
    QTimer timer;
    timer.setInterval(1000);
    timer.setSingleShot(true);
    timer.start();
```

By property, we mean any conceptual attribute that's part of the object's state — whether or not it's an actual Q_PROPERTY. When practicable, users should be allowed to set the properties in any order; i.e., the properties should be orthogonal. For example, the preceding code could be written

```
    QTimer timer;
    timer.setSingleShot(true);
    timer.setInterval(1000);
    timer.start();
```

For convenience, we can also write

```
timer.start(1000).
```

Similarly, for QRegExp, we have

```
    QRegExp regExp;
    regExp.setCaseSensitive(Qt::CaseInsensitive);
    regExp.setPattern("***.*");
    regExp.setPatternSyntax(Qt::WildcardSyntax);
```
To implement this type of API, it pays off to construct the underlying object lazily. E.g. in QRegExp's case, it would be premature to compile the "***.*" pattern in setPattern() without knowing what the pattern syntax will be.

Properties often cascade; in that case, we must proceed carefully. Consider the "default icon size" provided by the current style vs. the "iconSize" property of QToolButton:

```
    toolButton->iconSize();    // returns the default for the current style
    toolButton->setStyle(otherStyle);
    toolButton->iconSize();    // returns the default for otherStyle
    toolButton->setIconSize(QSize(52, 52));
    toolButton->iconSize();    // returns (52, 52)
    toolButton->setStyle(yetAnotherStyle);
    toolButton->iconSize();    // returns (52, 52)
```
Notice that once we set iconSize, it stays set; changing the current style doesn't change a thing. This is good. Sometimes, it's useful to be able to reset a property. Then there are two approaches:

* pass a special value (such as QSize(), -1, or Qt::Alignment(0)) to mean "reset"
* have an explicit resetFoo() or unsetFoo() function

For iconSize, it would be enough to make QSize() (i.e., QSize(-1, -1)) mean "reset".

In some cases, getters return something different than what was set. E.g. if you call widget->setEnabled(true), you might still get widget->isEnabled() return false, if the parent is disabled. This is OK, because that's usually what we want to check (a widget whose parent is disabled should be grayed out too and behave as if it were disabled itself, at the same time as it remembers that deep inside, it really is "enabled" and waiting for its parent to become enabled again), but must be documented properly.

C++ Specifics

Value vs. Object

Pointers vs. 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, we at Qt Software 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.

Virtual Functions

When a member function of a class is declared virtual in C++, it's primarily to allow customizing the behavior of the function through overloading it in a custom subclass. The purpose of making the function virtual is so existing calls to that function will visit your code path instead. If nobody outside of the class calls this function, you should be very careful before you declare it as virtual.

```
    // QTextEdit in Qt 3: member functions that have no reason for being virtual
    virtual void resetFormat();
    virtual void setUndoDepth( int d );
    virtual void setFormat( QTextFormat *f, int flags );
    virtual void ensureCursorVisible();
    virtual void placeCursor( const QPoint &pos;, QTextCursor **c = 0 );
    virtual void moveCursor( CursorAction action, bool select );
    virtual void doKeyboardAction( KeyboardAction action );
    virtual void removeSelectedText( int selNum = 0 );
    virtual void removeSelection( int selNum = 0 );
    virtual void setCurrentFont( const QFont &f );
    virtual void setOverwriteMode( bool b ) { overWrite = b; }
```
When QTextEdit was ported from Qt 3 to Qt 4, almost all virtual functions were removed. Interestingly (but not unexpected), there were no big complaints Why? Because Qt 3 didn't make use of polymorphism for QTextEdit; Qt 3 doesn't call these functions – you do. In short, there was no reason to subclass QTextEdit and reimplement these functions unless you called these functions yourself. If you needed polymorphism in your application outside of Qt, you would add polymorphism yourself.

Avoiding virtual functions

In Qt, we try to minimize the number of virtual functions for a number of reasons. Each virtual call complicates bugfixing through inserting an uncontrolled node in the call graph (making the outcome somewhat unpredictable). People do crazy things from inside a reimplementation of a virtual function, such as:

* sending events
* emitting signals
* reentering the event loop (e.g., by opening a modal file dialog)
* deleting the object (i.e., somehow causing "delete this")

There are many other reasons to avoid excessive use of virtual functions:

* you cannot add, move or remove virtual functions without breaking BC
* you cannot easily overload a virtual function
* compilers can almost never optimize or inline calls to virtual functions
* calling the function requires a v-table lookup, making it 2-3 times slower than a normal function
* virtual functions make the class hard to copy by value (possible, but very messy and discouraged)

Experience has taught us that a class with no virtual functions tends to have fewer bugs and generally causes less maintenance.

A general rule of thumb is that unless we as a toolkit and primary users of this class call that function, it should probably not be virtual.

Virtualness vs. copyability

Polymorphic objects and value-type classes are not good friends.

Classes with virtual functions must declare a virtual destructor to avoid memory leaks as the base class is destroyed without cleaning up data in the subclass.

If you want to be able to copy and assign to a class, or compare by value, you probably need a copy constructor, an assignment operator and an equals-operator.

```
    class CopyClass {
    public:
        CopyClass();
        CopyClass(const CopyClass &other);
        ~CopyClass();
        CopyClass &operator=(const CopyClass &other);
        bool operator==(const CopyClass &other) const;
        bool operator!=(const CopyClass &other) const;

        virtual void setValue(int v);
    };
```

If you create subclasses of this class, unexpected things can start happening in your code. Normally, if there are no virtual functions and no virtual destructor, people cannot not create a subclass and rely on polymorphism. However if you add virtual functions, or a virtual destructor, there suddenly becomes a reason to create the subclass, and now things get complicated. At first glance it’s easy to think you can simply declare virtual operators. But wandering down this path can and will lead to chaos and destruction (read: unreadable code). Studying the following example:

```
    class OtherClass {
    public:
         const CopyClass &instance() const; // what does it return? What should I
assign it to?
    };
```

(this section is under construction)

Constness

C++ provides the keyword “const” to signify that something will not change or have side effects. This applies to simple values, to pointers and what’s pointed to, and as a special attribute to functions that don’t change the state of the object.

Note however that const does not provide much value in itself – many languages don’t even provide any “const” keyword, but that doesn’t automatically render them deficient for that reason. In fact, if you remove function overloads and use search and replace to remove all occurrances of the keyword “const” from your C++ source code, it’s very likely to compile and work just fine. It’s important to keep a pragmatic approach to the use of “const”.

Let’s walk through some areas that use “const” that are relevant to API design in Qt:

Input arguments: const pointers

Const functions that take input pointer arguments almost always take const pointer arguments.

If the function is really declared const, it means it will neither have side effects, nor alter the visible state of its object. So why should it require a non-const input argument? Remember that const functions are often called from within other const functions, and from there, non-const pointers are hard to come by (without a const_cast, and we really like to avoid const_cast where we can).

Before:

```
    bool QWidget::isVisibleTo(QWidget *ancestor) const;
    bool QWidget::isEnabledTo(QWidget *ancestor) const;
    QPoint QWidget::mapFrom(QWidget *ancestor, const QPoint &pos) const;
```

QWidget declares many const functions that take non-const pointer input arguments. Note that the function is allowed to modify the widget, but not itself. Functions like these are often accompanied by const_casts. It would have been nice if these functions took const pointer arguments:

After:

```
    bool QWidget::isVisibleTo(const QWidget *ancestor) const;
    bool QWidget::isEnabledTo(const QWidget *ancestor) const;
    QPoint QWidget::mapFrom(const QWidget *ancestor, const QPoint &pos) const;
```

Note that we fixed this in QGraphicsItem, but QWidget must wait until Qt 5:

```
    bool isVisibleTo(const QGraphicsItem *parent) const;
    QPointF mapFromItem (const QGraphicsItem *item, const QPointF &point) const;
```

Return values: const values

The result of calling a function that does not return a reference is an R-value.

Non-class R-values always have cv-unqualified type. So even if it is syntactically possible to add a “const” on them it does not make much sense as it won’t change anything regarding access rights.
Most modern compilers will print a warning when compiling such code.

When adding a “const” to a class type R-values access to non-const member functions is prohibited as well as direct manipulation of it members.

Not adding a “const” allows such access, but is rarely needed as the changes end with the life time of the R-value object, which will usually happen at the end of the full-[removed]loosely spoken “at the next semicolon”).

Example:

```
    struct Foo
    {
        void setValue(int v) { value = v; }
        int value;
    };

    Foo foo()
    {
        return Foo();
    }

    const Foo cfoo()
    {
        return Foo();
    }

    int main()
    {
        // The following does compile, foo() is non-const R-value which
        // can't be assigned to (this generally requires an L-value)
        // but member access leads to a L-value:
        foo().value = 1;  // Ok, but temporary will be thrown away at the end of the
full-expression.

        // The following does compile, foo() is non-const R-value which
        // can't be assigned to, but calling (even non-const) member
        // function is fine:
       foo().setValue(1); // Ok, but temporary will be thrown away at the end of the
full-expression.

        // The following does _not_compile, foo() is _const_ R-value
        // with const member  which member access can't be assigned to:
        cfoo().value = 1;  // Not ok.

        // The following does _not_compile, foo() is _const_ R-value,
        // one cannot call non-const member functions:
        cfoo().setValue(1); // Not ok
    }
```

Return values: pointers vs. const pointers

On the subject of whether const functions should return pointers or const pointers, this is where most people find that the concept of "const correctness" falls apart in C++. The problem starts when const functions, which do not modify their own state, return a non-const pointer to a member. The simple act of returning this pointer does not affect the object's visible state, nor does it change the state of its responsibilities. But it does give the programmer indirect access to modify the object's data.

This example shows one of the many ways to circumvent constness using const functions that return non-const pointers:

```
    QVariant CustomWidget::inputMethodQuery(Qt::InputMethodQuery query) const
    {
        moveBy(10, 10); // doesn't compile!
    window()->childAt(mapTo(window(), rect().center()))->moveBy(10, 10); // compiles!
    }
```

Functions that return const pointers do protect against this (perhaps unwanted / unexpected) side-effect, at least to a certain degree. But which functions would you prefer to return a const pointer, or a list thereof? If we take the const-correct approach, every const function that returns a pointer to one of its members (or a list-of-pointers-to-members), must return a const pointer. In practise this unfortunately leads to unusable APIs:

```
    QGraphicsScene scene;
    // ... populate scene

    foreach (const QGraphicsItem *item, scene.items()) {
         item->setPos(qrand() % 500, qrand() % 500); // doesn't compile! item is a
const pointer
    }
```

`QGraphicsScene::items()` is a const function, and this might lead you to think it should only return const pointers.

In Qt we use the non-const pattern almost exclusively. We've chosen a pragmatic approach: Returning const pointers is more likely to result in excessive use of const_cast than what problems arise from abusing non-const pointer return types.

Return values: by value or const reference?

If we hold a copy of the object to return, returning a const reference is the fastest approach; however, this restrains us later on if we want to refactor the class. (Using the d-pointer idiom, we can change the memory representation of Qt classes at any time; but we cannot change a function's signature from "const QFoo &" to "QFoo" without breaking binary compatibility.) For this reason, we generally return "QFoo" rather than "const QFoo &", except in a few cases where speed is critical and refactoring isn't an issue (e.g. QList::at()).

Const vs. the state of an object

Const correctness is a vi-emacs discussion in C++, because the topic is broken in several areas (such as pointer-based functions).

But the general rule is that a const function does not alter the visible state of a class. State means "me and my responsibilities". That's doesn't mean that non-const functions change their own private data members, nor that const functions cannot. But that the function is active, and has visible side effects. const functions in general do not have any visible side effects. Like:

```
    QSize size = widget->sizeHint(); // const
    widget->move(10, 10); // not const
```

A delegate is responsible for drawing onto something else. Its state includes its responsibilities, and therefore includes the state of what it draws upon. Asking it to draw does have side effects; it changes the appearance (and with that, the state) of the device it's painting on. Because of that, it does not make sense that paint() is const. Neither does it make sense that any of Interview's paint()s or QIcon's paint() are const. Nobody would call QIcon::paint() from inside a const function unless they explicily want to void the constness of that function. And in that case, an explicit const_cast is better.

```
    // QAbstractItemDelegate::paint is const
    void QAbstractItemDelegate::paint(QPainter **painter, const QStyleOptionViewItem
&option, const QModelIndex &index) const

    // QGraphicsItem::paint is not const
    void QGraphicsItem::paint(QPainter * painter, const QStyleOptionGraphicsItem **
option, QWidget *widget = 0)
```

The const keyword does no "work" for you. Consider removing them rather than having overloaded const/non-const versions of a function.

API Semantics and Documentation

What should you do when you pass -1 to a function? etc…

Warnings/fatals/etc

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).

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.

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.

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.

Avoiding Common Traps

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 are 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
```

The Copy Cat Trap

Case Studies

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);
        ...
    };
```

QAbstractPrintDialog & QAbstractPageSizeDialog

Qt 4.0 saw the apparition of two classes QAbstractPrintDialog and QAbstractPageSizeDialog that served as base classes for QPrintDialog and QPageSizeDialog. This served no purpose at all, since none of Qt's APIs take a QAbstractPrint- or -PageSizeDialog pointer as an argument and perform some operation on it. Using qdoc trickery, we've hidden them, but they're the prototypical examples of needless abstract classes.

This is not to say good abstraction is wrong, and indeed QPrintDialog probably should have a factory or some other mechanism for changing it – as evidenced by the #ifdef QTOPIA_PRINTDIALOG in its declaration.

QAbstractItemModel

The details of the problems with model/view in Qt 4 are documented well elsewhere, but an important generalization is that "QAbstractFoo" should not just be the union of all possible subclasses you can think of at the time of writing. Such "union of all things" base classes are almost never a good solution. QAbstractItemModel commits this error – it is really just QTreeOfTablesModel, with the consequently complicated API that causes… and which is then inherited by all the nicer subclasses,

Just adding abstraction does not make an API better automatically.

QLayoutIterator & QGLayoutIterator

In Qt 3, creating a custom layout involved subclassing both QLayout and QGLayoutIterator ("G" stands for generic). A QGLayoutIterator subclass instance pointer was wrapped in a QLayoutIterator, which users could use like any other iterator class. QLayoutIterator made it possible to write code like this:

```
    QLayoutIterator it = layout()->iterator();
    QLayoutItem **child;
    while ((child = it.current()) != 0) {
        if (child->widget() == myWidget) {
            it.takeCurrent();
            return;
        }
        ++it;
    }
```

In Qt 4, we killed QGLayoutIterator classes (and their internal subclasses for box and grid layouts) and instead asked the QLayout subclasses to reimplement itemAt(), takeAt(), and count().

QImageSink

Qt 3 had a whole set of classes that allowed images to be incrementally read and passed to an animation – the QImageSource/Sink/QASyncIO/QASyncImageIO classes. Since all these were ever used for was animated QLabels, it was total overkill.

The lesson is not to add abstraction to aide some very vague future possibility. Keep it simple. When those future things come, it will be a lot easier to factor them into a simple system than into a complex one.

other Qt3 vs. Qt4?

QWidget::setWindowModified(bool)

Q3Url vs. QUrl

Q3TextEdit vs. QTextEdit

How all those virtual functions went a-goner…

Qt's Clipping Story (naming of clipping fns)

When you set the clip rect, you actually set a region (should be setClipRegion(QRect) instead of setClipRect()).

(on the right, how it should have been…)