The Property Browser Framework

by Jarek Kobus

The most recent Qt Solutions release included the Property Browser solution, a framework providing a set of graphical editors for Qt properties similar to the one used in Qt Designer.

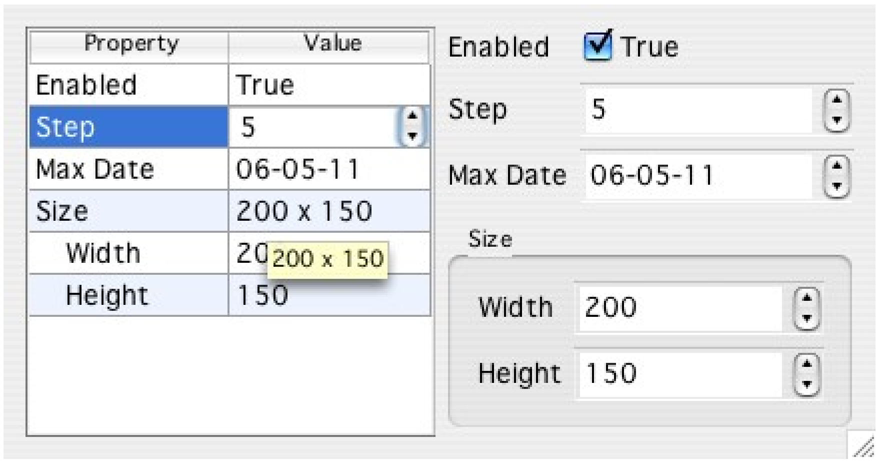
In this article we will take a closer look at the dynamics of the Property Browser classes, presenting two ways of using them in your own applications, and see how the framework can be extended with custom types and editors.

[Download source code]

The Big Picture

The Property Browser framework includes browser widgets, property managers, property factories and the properties themselves.

The browser widgets are user interfaces that enables the user to edit a given set of hierarchically structured properties: `QtTreePropertyBrowser` presents the properties using a tree structure and `QtGroupBoxPropertyBrowser` displays the properties within a group box.



In addition, it is possible to extend the framework by writing custom property browser widgets derived from the `QtAbstractPropertyBrowser` class.

Custom editor widgets for properties are created by factories derived from the QAbstractEditorFactory class.

Properties only need to be encapsulated in `QtProperty` instances before they can be used in browser widgets. These instances are created and managed by property managers derived from the `QtAbstractPropertyManager` class.

Populating the Property Browser

When using a property browser widget, managers must be created for each of the required property types if we want the user to be able to edit the properties.

The framework provides managers for the most common types. For example, to create an integer property we first instantiate the `QtIntPropertyManager` class, then we use its `addProperty()` function to create the property:

```
QtIntPropertyManager *intManager;
QtProperty *priority:

intManager = new QtIntPropertyManager;
priority = intManager->addProperty("Priority");

priority->setToolTip("Task Priority");
intManager->setRange(priority, 1, 5);
intManager->setValue(priority, 3);
```

The `QtProperty` class provides functions for setting a property's name, tooltip, status tip and "What's This?" text. To set and alter the property's value and range, each manager has its own type specific API. For example, here's how an enum property is set up and given a default value:

```
QtEnumPropertyManager *enumManager;
QtProperty *reportType;
QStringList types;
...
types << "Bug" << "Suggestion" << "To Do";
enumManager->setEnumNames(reportType, types);
enumManager->setValue(reportType, 1); // "Suggestion"
```

Properties can be categorized into groups using the `QtGroupPropertyManager` class. For example:

```
QtGroupPropertyManager *groupManager;
QtProperty *task1;

groupManager = new QtGroupPropertyManager;
task1 = groupManager->addProperty("Task 1");

task1->addSubProperty(priority);
task1->addSubProperty(reportType);
```

In addition, each property can have zero or more subproperties. These are just normal properties that are added to a property using its `addSubProperty()` function.

The properties created by a group property manager don't have any value, they are just provided as grouping elements in the property hierarchies and can be added to the browser in the same way as any other property.

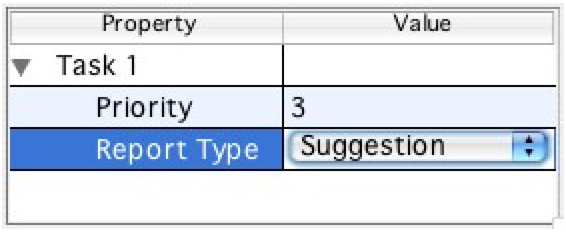
Once we have the properties, we associate each property manager with a factory for the preferred editor for that type. As with managers, the framework provides factories for common widgets like QSpinBox and QComboBox.

```
QtSpinBoxFactory *spinBoxFactory;
QtEnumEditorFactory *enumFactory;

spinBoxFactory = new QtSpinBoxFactory;
enumFactory = new QtEnumEditorFactory;

QtTreePropertyBrowser *browser;
browser = new QtTreePropertyBrowser;
browser->setFactoryForManager(intManager, spinBoxFactory);
browser->setFactoryForManager(enumManager, enumFactory);
```

By relating a factory to a manager, we ensure that whenever a property created by that manager appears in the browser, the specified factory will be used to create an editor for it.

```
browser->addProperty(task1);
browser->show();
```

Finally, it only remains to add our properties to the browser and ensure that the browser widget is shown.



Property	Value
Task 1	
Priority	3
Report Type	Suggestion

Note that to provide different editor widgets for the same property type, we only need to call `setFactoryForManager()` with another instance of the type-specific factory.

Once the browser widget is populated, the user can interact with it and edit the properties. We can monitor the values of properties by connecting to each property manager's signals, such as `QtIntPropertyManager::valueChanged()` and `QtEnumPropertyManager::enumChanged()`.

The Variant Approach

In the previous section we populated the property browser using specialized classes for each property type. But the framework also provides a convenient second approach that uses QVariant to hold all the properties' values and attributes.

In this approach, the developer only needs one property manager class and one editor factory class. These are selected from the following three classes:

- `QtVariantProperty` inherits `QtProperty` but has an enriched API, allowing properties values and attributes to be directly queried and altered.
- `QtVariantPropertyManager` can be used for all property types, and its additional API can be used to query for supported variant types and their attribute lists.
- `QtVariantEditorFactory` is capable of creating various editor widgets for the types supported by `QtVariantPropertyManager`.

Using this approach, the code from the previous section to manage an integer property now looks like this:

```
QtVariantPropertyManager *variantManager;
variantManager = new QtVariantPropertyManager;

QtVariantProperty *priority = variantManager->addProperty(QVariant::Int, "Priority");
priority->setAttribute("minimum", 1);
priority->setAttribute("maximum", 5);
priority->setValue(3);
```

Note the additional argument in the `QtVariantManager::addProperty()` function. It is the type of property we want to create. This can be any of values defined in the QVariant::Type enum plus additional user values obtained using the `qMetaTypeId()` function.

Subsequently, instead of calling `setRange()` on `QtIntPropertyManager`, we use the `setAttribute()` function of `QtVariantProperty` class twice, passing minimum and maximum values for the property. Finally, we call the `setValue()` function directly on the `QtVariantProperty` instance.

The ID of an enum property is not built into QVariant, and must be obtained using the static `QtVariantPropertyManager::enumTypeId()` function.

```
QtVariantProperty *reportType = variantManager->addProperty(
    QtVariantPropertyManager::enumTypeId(), "Report Type");
QStringList types;
types << "Bug" << "Suggestion" << "To Do";
reportType->setAttribute("enumNames", types);
reportType->setValue(1); // "Suggestion"
```

Then we can call `setAttribute()` for the property, passing `"enumNames"` as the attribute name, to set the possible enum values that the property can have.

Group properties are created in the same way as enum properties, but with an ID retrieved using the `QtVariantPropertyManager::groupTypeId()` function.

In the end we create an instance of the `QtVariantEditorFactory` class, and use it with our variant manager in our chosen property browser:

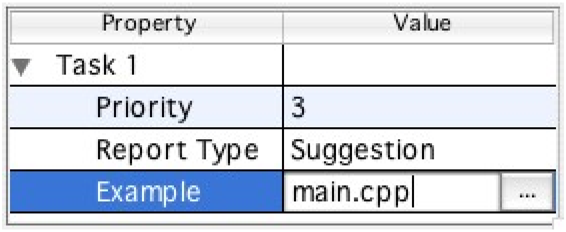
```
QtTreePropertyBrowser *browser;
QtVariantEditorFactory *factory;
...
browser->setFactoryForManager(variantManager, factory);
```

Extending the Framework

So far we have used the existing properties, managers and factories provided by the Property Browser framework. But the framework also allows the developer to create and support custom property types.

Let's say we want to extend our previous example, and support a custom file path property. The value of our custom property can be a QString, and the property can in addition possess a filtering attribute. We can also create a custom editor for our type, `FileEdit`, a simple composition of QLineEdit and QToolButton. We want to let the line edit show the current file path, and show a file browser dialog when the tool button is pressed (allowing the user to choose a file from the files that match our filtering attribute).

The editor class provides getter and setter functions for its fields, as well as a signal that can be emitted whenever the file path changes.



Property	Value
▼ Task 1	
Priority	3
Report Type	Suggestion
Example	main.cpp ...

We have described two approaches for populating a browser widget: One uses type specific property managers and editor factories; the other uses the framework's variant base classes. Both approaches support custom types.

If we choose the type specific approach, we must provide a manager that is able to produce properties of our custom type, and that can store the state of the properties it creates; i.e., their values and attributes. Such a manager looks like this:

```
class FilePathManager : public QtAbstractPropertyManager
{
    ...
    QString value(const QtProperty *property) const;
    QString filter(const QtProperty *property) const;
public slots:
    void setValue(QtProperty *, const QString &);
    void setFilter(QtProperty *, const QString &);
signals:
    void valueChanged(QtProperty *, const QString &);
    void filterChanged(QtProperty *, const QString &);
protected:
    QString valueText(const QtProperty *property) const
        { return value(property); }
    void initializeProperty(QtProperty *property)
        { theValues[property] = Data(); }
    void uninitializeProperty(QtProperty *property)
        { theValues.remove(property); }
private:
    struct Data {
        QString value;
        QString filter;
    };
    QMap<const QtProperty *, Data> theValues;
};
```

Our custom `FilePathManager` is derived from the `QtAbstractPropertyManager`, enriching the interface with its attributes, and providing the required getter and setter functions for our custom property. The `valueChanged()` and `filterChanged()` signals are used to communicate with custom editor factories (shown later), and can also be generally used to monitor properties. We define a simple data structure to store the state of the value and filter for each file path property, and record each file path property in a private map.

The `FilePathManager` must also reimplement several protected functions provided by its base class that are needed by the framework: `valueText()` returns a string representation of the property's value, `initializeProperty()` initializes a new property, and `uninitializeProperty()` is used to clean up when a property is deleted.

We return empty data from getter functions if an unknown property is supplied; otherwise we return the data from the property map:

```
QString FilePathManager::value(const QtProperty *property)
        const
{
    if (!theValues.contains(property)) return "";
    return theValues[property].value;
}
```

In the setter functions, we ignore attempts to set the values of unknown properties, and only commit new data to the property map. We emit the base class's `propertyChanged()` signal to notify other components of any changes. However, note that the `setFilter()` does not need to do this because it does not change the property's value.

```
void FilePathManager::setValue(QtProperty *property, const QString &val)
{
    if (!theValues.contains(property)) return;
    Data data = theValues[property];
    if (data.value == val) return;
    data.value = val;
    theValues[property] = data;
    emit propertyChanged(property);
    emit valueChanged(property, data.value);
}
```

Before we can add instances of our custom property to the browser widget, we must also provide a factory which is able to produce `FileEdit` editors for our property manager:

```
class FileEditFactory
    : public QtAbstractEditorFactory<FilePathManager>
{
    ...
private slots:
    void slotPropertyChanged(QtProperty *property,
                             const QString &value);
    void slotFilterChanged(QtProperty *property,
                           const QString &filter);
    void slotSetValue(const QString &value);
    void slotEditorDestroyed(QObject *object);
private:
    QMap<QtProperty *, QList<FileEdit *> > createdEditors;
    QMap<FileEdit *, QtProperty *> editorToProperty;
};
```

The `FileEditFactory` is derived from the `QtAbstractEditorFactory` template class, using our `FilePathManager` class as the template argument. This means that our factory is only able to create editors for our custom manager while allowing it to access the manager's enriched interface.

As with the manager, the custom factory must reimplement some protected functions. For example, the `connectPropertyManager()` function is called when the factory is set up for use with a `FilePathManager`, and connects the manager's `valueChanged()` and `filterChanged()` signals to the corresponding `slotPropertyChanged()` and `slotFilterChanged()` slots. These connections keep the state of the editors up to date. Similarly, `disconnectPropertyManager()` is used to remove these connections.

The `createEditor()` function creates and initializes a `FileEdit` widget, adds the editor to the internal maps:

```
    FileEdit *editor = new FileEdit(parent);
    editor->setFilePath(manager->value(property));
    editor->setFilter(manager->filter(property));
    createdEditors[property].append(editor);
    editorToProperty[editor] = property;

    connect(editor, SIGNAL(filePathChanged(const QString &)),
            this, SLOT(slotSetValue(const QString &)));
    connect(editor, SIGNAL(destroyed(QObject *)),
            this, SLOT(slotEditorDestroyed(QObject *)));
    return editor;
```

Before returning the new editor widget, we connect its `filePathChanged()` signal to the factory's `slotSetValue()` slot. We also connect its `destroyed()` signal to the factory's `slotEditorDestroyed()` slot to ensure that we are notified if it is destroyed.

Internally, the task of each factory is to synchronize the state of the editors with the state of the properties. Whenever an editor changes its value we need to tell the framework about it, and vice versa. In our editor factory's `editorToProperty` map, we relate every editor created by the factory to the property it was created to edit.

The opposite relation is kept in the `createdEditors` map, where properties are related to editors. Since it is possible that two or more property browsers are displaying the same instance of a property, we may need to create many editors for the same property instance, so we keep a list of editors for each property.

Combined with the private slots, these data structures allow us to inform the framework that an editor value has been changed, and let us update all relevant editors whenever the framework changes any of the properties.

When the factory is destroyed, the state of the properties is no longer synchronized, and the data shown by the editors is no longer valid. For that reason, all the editors created by the factory are deleted in the factory's destructor. Therefore, when an editor is destroyed, we must ensure that it is removed from the internal maps.

The process of adding a custom property is equivalent to adding any other property:

```
    FilePathManager *filePathManager;
    FileEditFactory *fileEditFactory
    QtProperty *example;

    filePathManager = new FilePathManager;
    example = filePathManager->addProperty("Example");

    filePathManager->setValue(example, "main.cpp");
    filePathManager->setFilter(example,
                               "Source files (*.cpp *.c)");

    fileEditFactory = new FileEditFactory;
    browser->setFactoryForManager(filePathManager, fileEditFactory);
    task1->addSubProperty(example);
```

Extending with Variants

The variant-based approach can also be used for custom types. Using this approach, we derive our manager from `QtVariantPropertyManager` and our factory from `QtVariantPropertyFactory`.

In the previous example, the `FilePathManager` class handled the data internally for our custom property type. In this case, we define the property type as a path value with an associated filter attribute. Both the path and the filter are stored as strings, and we define a new property type for the file path as a whole:

```
class FilePathPropertyType
{
};
Q_DECLARE_METATYPE(FilePathPropertyType)
```

The custom manager must reimplement several inherited functions and provide a static function to return the new property type:

```
int VariantManager::filePathTypeId()
{
    return qMetaTypeId<FilePathPropertyType>();
}
```

This function returns a unique identifier for our file path property type that can be passed to `QtVariantPropertyManager::addProperty()` to create new file path properties.

Unlike the `FilePathManager` class, all the functions inherited from the base class already have an implementation; all we have to do is to intercept the process whenever our custom property type occurs. For example, when reimplementing the `isPropertyTypeSupported()` function we should return `true` for our custom type, and call the base class implementation for all other types:

```
bool VariantManager::isPropertyTypeSupported(
     int propertyType) const
{
    if (propertyType == filePathTypeId())
        return true;
    return QtVariantPropertyManager::
        isPropertyTypeSupported(propertyType);
}
```

The `VariantFactory` class, like every other editor factory, does not have to provide any new members. Basically, the implementation is very similar to `FileEditFactory` in the previous section. The only difference in the `connectPropertyManager()` function is that we must connect the manager's `attributeChanged()` signal to a general `slotPropertyAttributeChanged()` slot instead of connecting a specific signal for the filter to a specific slot to handle changes.

We also need to check that the property can handle our custom type in the `createEditor()` function:

```
if (manager->propertyType(property) == VariantManager::filePathTypeId()) {
  ...
  connect(editor, SIGNAL(filePathChanged(const QString&)),
          this, SLOT(slotSetValue(const QString &)));
  connect(editor, SIGNAL(destroyed(QObject *)),
          this, SLOT(slotEditorDestroyed(QObject *)));
  return editor;
}
...
```

We also monitor the editor via its `destroyed()` and custom `filePathChanged()` signals.

Using custom variant managers and factories is no different than using those delivered by the framework:

```
    QtVariantPropertyManager *variantManager;
    variantManager = new VariantManager;
    QtVariantEditorFactory *variantFactory;
    variantFactory = new VariantFactory;

    QtVariantProperty *example;
    example = variantManager->addProperty(
              VariantManager::filePathTypeId(), "Example");
    example->setValue("main.cpp");
    example->setAttribute("filter", "Source files (*.cpp *.c)");

    QtVariantProperty *task1;
    task1 = variantManager->addProperty(
        QtVariantPropertyManager::groupTypeId(), "Task 1");
    task1->addSubProperty(example);
```

All that's left to do is to set up the browser to use the property manager and factory in the same way as before.

Summary

The Property Browser Solution provides a framework for creating property editors that can be used and extended with either a type-specific or a variant-based API.

The type-specific approach provides a specialized API for each property that makes compile-time checking possible. In this approach, the managers and factories can easily be subclassed and used in other projects. In addition, the developer can customize the property browser with new editor factories without subclassing.

The variant-based approach provides immediate convenience for the developer, allowing simple, extensible code to be written. However, it can be difficult to integrate into a more complex code base, and `QtVariantEditorFactory` must be subclassed to change the default editors.

The first release of the Property Browser framework contained a comprehensive set of examples and documentation. A new version, incorporating several new improvements is expected in the next Qt Solutions release.