

Шаблон проектирования «Наблюдатель»

Иногда бывает необходимо предоставить одному классу доступ к методам другого класса, объявленным в разделах `private` или `protected`. Подобная ситуация может возникнуть, например, когда несколько классов входят в одну библиотеку. У этих классов могут быть методы, предназначенные исключительно для взаимодействия между собой. Поскольку пользователю библиотеки не требуется вызывать эти методы непосредственно, лучше, если он не увидит их в разделе `public`. Для решения этой задачи можно воспользоваться ключевым словом `friend`, но сторонники красивого программирования не любят этот подход, так как в результате разные классы оказываются слишком тесно связаны между собой. В подобной ситуации рекомендуется использовать шаблон проектирования «Наблюдатель». В рамках этого шаблона объекты разных классов обмениваются между собой сообщениями (обычно у каждого типа сообщений есть один источник и один или более получателей) и при этом обладают минимумом информации друг о друге. В Qt оказывается неожиданно просто решить обе описанные задачи: реализовать шаблон «Наблюдатель» и при этом скрыть методы, реализующие этот шаблон, в непубличных разделах класса.

Один из способов решить обе проблемы разом – использование сигналов и слотов. Судя по вопросам, которые мне иногда задают, многие не понимают, что для механизма сигналов и слотов совершенно не важно, в каком разделе объявлен слот, открытом или закрытом. Раздел, в котором объявлен слот, влияет на то, кто может вызывать слот как обычный метод. Но наличие слота в закрытом разделе класса совершенно не влияет на то, кто может вызывать слот как слот, то есть с помощью сигнала. Таким образом, использование сигналов и слотов не только позволяет реализовать шаблон «Наблюдатель» (для реализации которого они и были задуманы), но и скрыть механизм этой реализации из публичных разделов класса.

Другой способ проникнуть в непубличные разделы класса извне – использование свойств. Рассмотрим класс:

```
class QxtObserver : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int prop READ getProp WRITE setProp)
public:
    explicit QxtSomeClass(QObject *parent = 0);
private:
    int prop;
    void setProp(int value);
    int getProp();
};
```

Вызвать методы `setProp()` и `getProp()` напрямую из внешнего объекта нельзя. Но, поскольку они являются частью объявления свойства, они могут быть вызваны с помощью методов `property()` и `setProperty()` класса `QObject`:

```
QxtObserver * observer;
int i;
...
observer->setProperty("prop", QVariant(i));
```

Как и в случае с сигналами и слотами, объект, обращающийся к объекту `observer`, может ничего не знать о классе `QxtObserver`. Все что нужно – знать имя и тип свойства.

Ромбическое наследование

Все знают, что множественным наследованием не стоит злоупотреблять. Одна из ситуаций, когда множественное наследование может быть полезным – ситуация, когда класс требуется представить как коллекцию интерфейсов. Интерфейсы представляют собой чистые абстрактные классы, то есть, такие классы, которые содержат только чистые виртуальные функции. Интерфейсы реализуются в классах, которые наследуют от двух классов: от своего неабстрактного класса предка и от класса интерфейса. При этом интерфейсы могут образовывать иерархию наследования, параллельную иерархии классов, их реализующих.

Рассмотрим, например, иерархию из двух интерфейсов:

```
class IBase
{
public:
    virtual int method1() = 0;
};

class IDescendant : public virtual IBase
{
public:
    virtual int method2() = 0;
};
```

И иерархию классов, реализующих эти интерфейсы

```
class BaseClass : public QObject, public virtual IBase
{
    Q_OBJECT
public:
    explicit BaseClass(QObject *parent = 0)
    {}
    int method1()
    {
        QFile
        return 1;
    }
    IBase * queryIBase()
    {
        return dynamic_cast<IBase *>(this);
    }
};

class DescendantClass : public BaseClass, public virtual IDescendant
{
    Q_OBJECT
public:
    explicit DescendantClass()
    {}
    int method2()
    {
        return 2;
    }
    IDescendant * queryIDescendant()
    {
        return dynamic_cast<IDescendant *>(this);
    }
};
```

В результате программист может пользоваться классом `DescendantClass`, например, так:

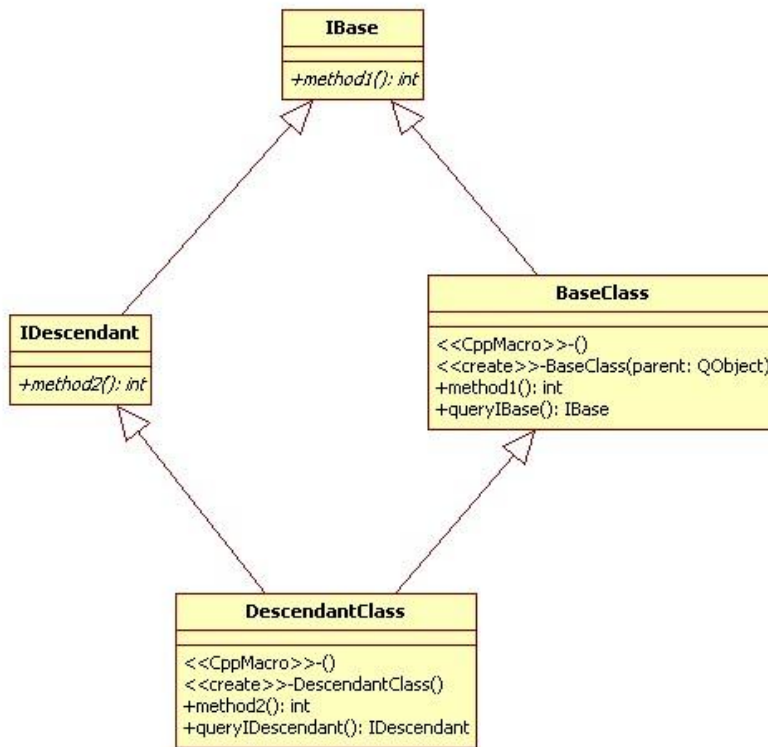
```
DescendantClass * dc = new DescendantClass();
IBase * ib = dc->queryIBase();
IDescendant * id = dc->queryIDescendant();
```

```

qDebug () << ib->method1 ();
QDebug () << id->method2 ();
QDebug () << id->method1 ();

```

Причем, благодаря виртуальному наследованию, мы можем быть уверены, что `ib->method1()` и `id->method1()` вызывают один и тот же метод. Русское и английское название этого метода (по-английски – diamond inheritance, причем слово diamond в данном случае нужно переводить как «ромб», а не как «бриллиант») отражает внешний вид диаграмм UML для иерархий таких классов (на диаграмме не отражен тот факт, что `BaseClass` наследует классу `QObject`):



Одно из ограничений множественного наследования в Qt заключается в том, что класс не может наследовать параллельно двум классам, каждый из которых является потомком `QObject` (нельзя, например, создать класс, который был бы потоком `QFile` и `QThread`). В ситуации с интерфейсами это не является проблемой. Интерфейсы все равно не должны быть потомками `QObject`, иначе у них появятся неабстрактные методы и собственные данные. Из этого следует, между прочим, что метод, объявленный в интерфейсе нельзя сделать слотом. Но ничто не мешает сделать слотом реализацию этого метода в классе, который реализует интерфейс.