

Signal-/Slot-Verbindungen in Qt-Anwendungen debuggen

Richtig verbinden

Dreizehn Regeln unterstützen Qt-Entwickler dabei, Probleme mit Signal-/Slot-Verbindungen zu vermeiden. Wo sie doch auftreten, hilft die freie Bibliothek Conan, indem sie einen Debugger in das Programm einbaut, der diese Verbindungen sichtbar macht. Alexander Nassian



© ImageTeam, Fotolia.com

Qt-Anwendungen nutzen das Signal-/Slot-Konzept, um Events zu verarbeiten. Programmierer definieren diese Signale und Slots als Methoden: Signal-Methoden repräsentieren dabei die Events, und einer oder mehrere Slots enthalten die Methoden, welche das Qt-Programm aufruft, wenn sich ein Event ereignet. Die Methode »QObject::connect()« sorgt für die richtige Zuordnung: Sie verbindet ein Signal mit einem Slot (**Abbildung 1**). Das Herstellen der Verbindung und die Aufrufe beim Auslösen des Ereignisses finden zur Laufzeit statt: Erst dadurch sind viele elegante Lösungen implementierbar, wie zum Beispiel dynamisch erzeugte GUIs. Doch wenn alles zur Laufzeit passiert, stellt sich die Frage, wie Entwickler etwa Tippfehler in den Namen der Signal- oder Slot-Methoden erkennen können. Zwar ist auch das zur Laufzeit möglich, aber hier liegt die Schwierigkeit beim Debuggen. Qt

schreibt im Debug-Modus aussagekräftige Warnungen auf die Konsole, aber häufig steht kein Konsolenfenster zur Verfügung, oder es ist bereits mit anderen Meldungen überladen.

Dreizehn Regeln zur Fehlervermeidung

Um nicht in immer wiederkehrende Fallen zu stapfen, helfen 13 einfache Regeln dabei, Fehler zu vermeiden. Eine Übersicht gibt **Tabelle 1**.

Wenn der Entwickler eine neue Verbindung einrichtet, prüft er, ob die Parametertypen des Signals zu denen des zugeordneten Slots passen. Zulässig sind vollständig übereinstimmende Typen, aber auch der Fall, in dem die Slot-Methode weniger Typen als das Signal definiert, ist gültig. Einzig mehr oder gänzlich unterschiedliche Parameter sind hier nicht erlaubt (Regel 1).

In den »SIGNAL«- und »SLOT«-Makros der Qt-»connect()«-Anweisung darf der Programmierer ausschließlich Typen, aber keine Variablen definieren und sollte darum prüfen, ob sich versehentlich ein Variablenname eingeschlichen hat (Regel 2):

```
connect(this, SIGNAL(a(int x)),
        this, SLOT(b(int y)); // FALSCH
connect(this, SIGNAL(a(int)),
        this, SLOT(b(int)); // RICHTIG
```

Slot-Methoden muss er stets in einem Block »private slots:«, »protected slots:« oder »public slots:« deklarieren. Beim Kompilieren erscheint keine Fehlermeldung, falls das »slots«-Keyword fehlt, da Slots für den Compiler normale, gültige Methoden sind. Der umsichtige Programmierer prüft daher, ob er wirklich alle als Slot verwendeten Methoden in einem solchen Bereich deklariert hat (Regel 3):

```
private slots: // RICHTIG
    void myCorrectSlot();
private: // FALSCH
    void myWrongSlot();
```

Programmierer vergessen oft, Signal-Methoden den Rückgabewert »void« zuzuweisen. Geben sie stattdessen zum Beispiel »int« an, verursacht das häufig nicht einmal einen Compiler-Fehler. Der Meta Object Compiler (MOC) erzeugt für jedes definierte Signal eine Implementierung in der entsprechenden Moc-Datei, die dann allerdings nicht mehr eindeutig ist (Regel 4).

Makros für Meta-Objekte

Das »Q_OBJECT«-Makro erlaubt es Qt-Klassen, mit dem Signal-/Slot-Mechanismus zu arbeiten. Es deklariert die für den Meta-Object-Mechanismus (vergleichbar

Tabelle 1: Merkgeln für Signal-/Slot-Verbindungen

Nr.	Thema	Regel
1	Parameter	Überprüfen, ob die Typen der Parameter bei Signalen und Slots übereinstimmen.
2	»SIGNAL«, »SLOT«	Ausschließlich Typen sind erlaubt.
3	Keywords	Signal-Methoden in einem »signals«-Block, Slot-Methoden in einem »slots«-Block deklarieren.
4	Rückgabewert	Signal-Methoden geben »void« zurück.
5	»Q_OBJECT«	Direkt nach der Klassendefinition.
6	»Q_OBJECT«	Kein Semikolon nach dem Makro.
7	Vererbung	Signal-/Slot-Klassen nur von »QObject« oder einer Ableitung ableiten.
8	Keywords	Das Keyword »emit« nutzen, auch wenn dies nicht notwendig ist.
9	Debugger	Drei Breakpoints helfen bei der Fehlersuche.
10	»disconnect«	Sicherstellen, dass das Programm zuvor kein »disconnect()« aufgerufen hat.
11	»qmake«	Wer eine Klasse Signal-/Slot-fähig macht, darf »qmake« nicht vergessen.
12	»connect«	Verbindungen sofort nach dem Erzeugen des Objekts anlegen.
13	Namen	Sinnvolle Namen für Signal- und Slot-Methoden vergeben.

mit Reflection) notwendigen Methoden, wie zum Beispiel »metaObject()« oder »qt_metacall(QMetaObject::Call, int, void*)«. Alternativ können Entwickler für Klassen, die keine Signale oder Slots definieren, aber trotzdem auf das »QMetaObject« zugreifen möchten, das Makro »Q_GADGET« verwenden. Es ist außerdem wichtig, das Makro »Q_OBJECT« direkt nach der Klassendefinition zu deklarieren (Regel 5).

```
class MyTestClass
{
    Q_OBJECT
    ...
};
```

Einen leicht zu übersehenden Fehler deckt die Suchfunktion des Editors zuverlässig auf: Ein Semikolon nach dem »Q_OBJECT«-Makro verwirrt den Compiler. Die Definition des Makros verlangt, hier kein Semikolon zu setzen. Der geübte Entwickler sucht daher bei einer nichts sagenden Fehlermeldung des Compilers oder des Linkers in allen Projektdateien nach der Zeichenkette »Q_OBJECT;« und findet hier oft auch schon die Fehlerquelle (Regel 6).

```
class MyTestClass {
    Q_OBJECT; // FALSCH
    Q_OBJECT // RICHTIG
    ...
};
```

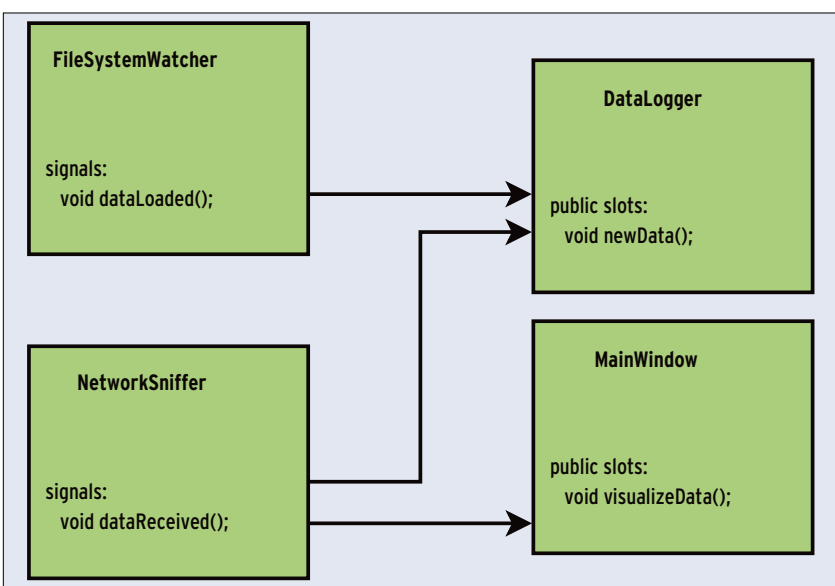


Abbildung 1: Qt-Anwendungen verwenden Signale (links) und Slots (rechts) für die Event-Verarbeitung.

Wer Signale und Slots verwendet, muss die entsprechende Klasse von »QObject« oder einer seiner Ableitungen selbst ableiten. »QObject« stellt einige wichtige Methoden für diesen Mechanismus, wie zum Beispiel »connect()« und »metaObject()«, zur Verfügung. Achtung: Bis Qt 4 war beispielsweise »QThread« selbst nicht von »QObject« abgeleitet. Darum war es schwierig bis unmöglich, Signale und Slots in Multithreaded-Anwendungen zu verwenden (Regel 7).

Signale mit »emit«

Das Schlüsselwort »emit« ist zwar über »#define« als leerer String definiert und hat damit keine Wirkung, es ist jedoch sehr nützlich, weil es die Lesbarkeit des Quellcodes verbessern kann. So ist es nicht nötig, einer Signalauslösung »emit« voranzustellen, denn es handelt sich ja um normale Methoden, aber zur besseren Lesbarkeit sollten Entwickler auf diese vier Zeichen nicht verzichten (Regel 8).

Sollte ein Slot wider Erwarten nicht aufgerufen werden, genügt oft der Einsatz des Debuggers mit zwei oder drei Breakpoints. Den ersten setzt der Entwickler beim Aufruf von »connect()« für die betroffene Verbindung. Die zweite Unterbrechung folgt beim Beginn der geplanten Slot-Methode, und bei eigenen Signalen setzt der Programmierer noch einen dritten Breakpoint an der Stelle, an der er das Auslösen des Events erwartet. Dieser kurze Check-up beseitigt Fehler oft, bevor es nötig wird, schwerere Geschütze aufzufahren (Regel 9).

Einfach, aber ebenfalls oft die Ursache: Hat das Programm die Verbindung vor ihrer geplanten Auslösung mit »disconnect()« aufgelöst, kann ein »emit« nicht funktionieren (Regel 10).

An »qmake« denken

Wenn Entwickler eine bereits bestehende Klasse nachträglich um das Makro »Q_OBJECT« erweitern, müssen sie »qmake« erneut ausführen. Erst dieser Aufruf aktualisiert die Projektdatei und schließlich das Makefile. Für die betroffenen Klassen ist es erforderlich, den Meta Object Compiler laufen zu lassen. Er generiert unter anderem den nötigen Code für den

Signal-/Slot-Mechanismus. Dazu gehören auch die Implementierungen der definierten Signale (Regel 11).

Zur Sicherheit sollten Programme Signale und Slots immer direkt nach der Erzeugung des entsprechenden Objektes verbinden (siehe Listing 1). Ruft das Programm zuvor noch Methoden auf, könnte es passieren, dass diese bereits Signale senden, für die es dann noch keine eigenen Slot-Methoden gibt (Regel 12).

Schließlich sollten Entwickler darauf achten, für Signale und Slots sinnvolle Namen zu vergeben, die immer demselben Schema folgen (siehe Listing 2). Für Signal-Methoden bieten sich dabei Namen in der Vergangenheitsform an (zum Beispiel »buttonClicked()«, nicht »clickButton()«), da sie ein Ereignis beschreiben, das in der Vergangenheit liegt, und Slot-Methoden sollten immer ein Verb in der aktiven Gegenwartsform enthalten, das die erwartete Reaktion beschreibt (Regel 13).

Wenn alle Stricke reißen

Wer ein Projekt oder eine Komponente neu entwickelt, kann recht leicht diese Regeln befolgen und sich nach gewissen Best Practices richten. Wenn jedoch ein Projekt bereits eine gewisse Größe er-

reicht hat und nun Probleme auftreten, ist das schwieriger. Für diesen Fall gibt es jedoch auch Abhilfe in Form einer Art Debugger, der sich speziell um Signale und Slots kümmert.

Ein Beispiel für einen solchen Debugger ist die freie Bibliothek Conan [1]. Sie nutzt die Qt-internen Headerdateien, um zur Laufzeit alle bestehenden Verbindungen zu analysieren. Die Bibliothek ermittelt, welche Qt-Objekte Signal-/Slot-Verbindungen zu anderen halten. Der Funktionsumfang ist hierbei überschaubar aber nützlich. Der Debugger stellt, während das Programm läuft, alle verwalteten Qt-Objekte und die von ihnen definierten Signale und Slots dar. Zusätzlich zeigt er auf Wunsch konkrete Verbindungen an, welche die Objekte miteinander eingehen.

Nach dem Einbinden des Conan-Headers steht Entwicklern die Klasse »ConanWidget« zur Verfügung. Dieses Widget erlaubt es nun, Qt-Objekte zu debuggen (siehe Listing 3). Es ist nicht zwingend erforderlich, mit »AddRootObject()« erst Objekte hinzuzufügen, da Conan ohnehin alle verfügbaren auslesen kann. Hat der Entwickler jedoch bereits die problematischen Objekte identifiziert, kann er die Informationsflut dadurch auf ein Minimum reduzieren.

Die Implementierung von Conan ist sehr komplex: Das »Q_OBJECT«-Makro deklariert eine Methode »metaObject()«, die Zugriffe auf »QMetaObject« erlaubt und

darüber Informationen zu vorhandenen Signalen und Slots sowie Verbindungen zwischen diesen liefert. Nicht möglich ist hingegen, die konkreten verbundenen Objekte oder gar eine Aufstellung sämtlicher von Qt verwalteten Objekte zu erhalten, da die entsprechenden Listen als »private« deklariert sind und es keine zugehörigen Zugriffsmethoden gibt. Die Liste aller Parent- und Child-Objekte einer Qt-Anwendung reicht hier nicht aus, da zum Beispiel das Hauptfenster einer GUI-Anwendung selbst meist kein Parent definiert hat.

Qt neu übersetzen

Conan nutzt eine interne API-Funktion der Qt-Bibliothek, um die Informationen zu erhalten. Diese Funktion ist jedoch nicht exportiert, so dass eine komplette Quellcode-Installation von Qt nötig ist, damit Conan arbeiten kann. Problematisch ist auch die Tatsache, dass es jederzeit zu Änderungen in dieser API kommen kann. Deswegen ist es wichtig, bei einer Aktualisierung der Qt-Bibliothek auf dem Entwicklerrechner zunächst zu prüfen, ob Conan die neue Version unterstützt. Wenn nicht, könnte es zu Abstürzen der kompletten Anwendung kommen. Die aktuelle Conan-Version arbeitet mit Qt 4.4.3 bis 4.5.2.

Die meisten Entwickler arbeiten mit einem Standard-Qt aus einem der Installationspakete von Nokia. Diese Pakete

Listing 1: Signalverlust durch zu spätes Verbinden

```
01 MyTestClass* ptr = new MyTestClass(); // FALSCH
02 ptr->doSomething();
03 connect(ptr,
04     SIGNAL(somethingHappened()),
05     this,
06     SLOT(doWork()));
07
08 MyTestClass* ptr = new MyTestClass(); // RICHTIG
09 connect(ptr,
10     SIGNAL(somethingHappened()),
11     this,
12     SLOT(doWork()));
13 ptr->doSomething();
```

Listing 2: Sinnvolle Bezeichner für Signale und Slots

```
01 signals:
02 void buttonClicked(); // RICHTIG
03 void clickButton(); // FALSCH
04
05 public slots:
06 void clear(); // RICHTIG
07 void buttonHovered(); // FALSCH
```

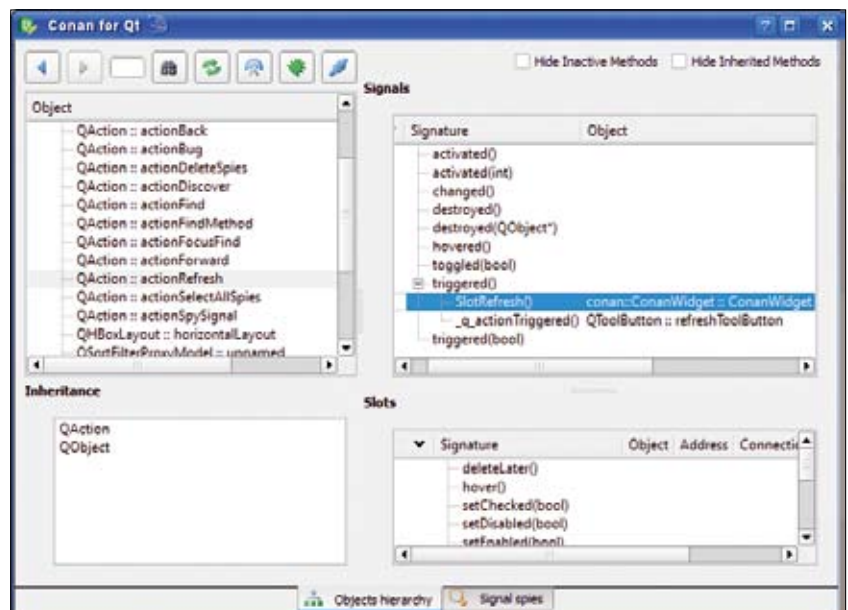


Abbildung 2: Das Conan-Widget in Aktion. Es analysiert gerade sich selbst, das »ConanWidget«-Objekt.

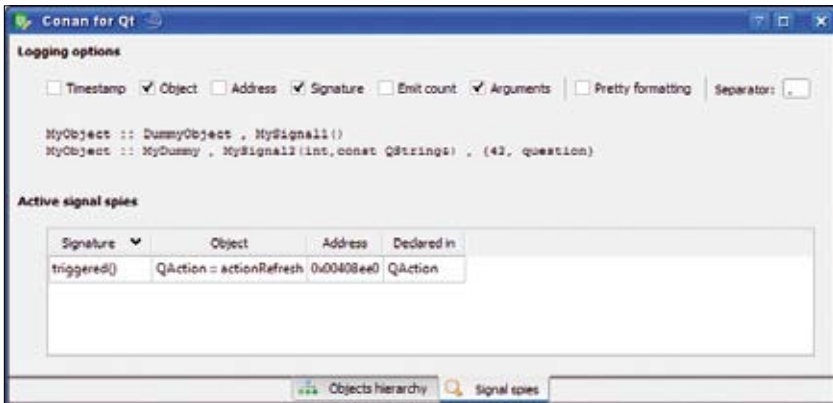


Abbildung 3: Der Signal-Spy registriert Signal-Methoden für die Beobachtung. Das Ausgabeformat ist über die »Logging options« vielseitig einstellbar und eignet sich auch für die automatische Verarbeitung.

enthalten jedoch nur die Bibliotheken und die Header der exportierten Typen. Conan benötigt aber auch die intern genutzten, privaten Header. Daher ist es notwendig, zunächst Qt aus den Sources [2] heraus zu übersetzen und zu installieren.

Die Source-Pakete enthalten eine detaillierte Installationsanleitung. Im Wesentlichen ist nur der klassische Dreischritt »./configure«, »make« und »make install« nötig. Bei der »configure«-Anweisung gibt es die Möglichkeit, verschiedene Optionen zu aktivieren, etwa für SQL-Support oder das Crosscompiling von Anwendungen. Entwickler sollten auch darauf achten, dass in den Compile- und Installationspfaden keine Leerzeichen auftauchen, die verwirrende Fehlermeldungen verursachen.

Mit dem Conan-Widget Signale und Slots debuggen

Nach erfolgreicher Installation von Qt ist es ratsam, als Test ein bestehendes Projekt mit der selbst erstellten Qt-Version komplett zu kompilieren. Danach steht dem Einsatz der Conan-Bibliothek nichts mehr im Weg. Conan einzubin-

den, ist leicht: Das zu untersuchende Projekt muss lediglich die Conan-Header »ConanWidget.h« ergänzen und die Bibliothek »Conan.so« hinzulinken. Eine Anwendung kann zum Beispiel direkt nach dem Start das Conan-Widget erzeugen und anzeigen. Alles Weitere übernimmt das Widget selbst (siehe Listing 4).

Wenn die Anwendung startet, erscheint neben dem Hauptfenster das Conan-Widget, das alle von Qt verwalteten Objekte, deren Signale und Slots sowie deren bestehende Verbindungen anzeigt (Abbildung 2). Bereits hier fällt es auf, wenn der Programmierer ein »connect()« vergessen oder ein verfrühtes »disconnect()« eingebaut hat. Enthält der Eintrag des Signals keine Unterpunkte, die bestehende Slot-Verbindungen anzeigen, löst der Entwickler das Problem, indem er die betreffenden Stellen im Sourcecode sucht und das eine oder andere »connect()« ergänzt.

Signal Spy

Ist das problematische Signal bereits bekannt, hat der Entwickler die Möglichkeit, einen so genannten Signal Spy zu definieren (Abbildung 3). Dazu klickt er einfach mit der rechten Maustaste auf

ein bestehendes Signal eines bestimmten Objekts und wählt den Punkt »Spy signal« aus. Fortan protokolliert Conan alle Signalauslösungen auf der parallel geöffneten Konsole (siehe Abbildung 4) mit. Das Format dieser Log-Einträge ist im Karteireiter »Signal Spies« frei wählbar.

Fazit

Wer 13 Regeln beachtet, vermeidet schon viele Probleme rund um Qt-Signale und Slots – um die verbleibenden Fehlerquellen kümmert sich Conan. (mg/hge) ■

Infos

- [1] Signal-/Slot-Debugger für Qt, Conan: <http://sourceforge.net/projects/conanforqt/>
- [2] Qt-Software FTP Server (Qt-Sourcen): <ftp://ftp.qtsoftware.com/qt/source>

Der Autor

Alexander Nassian programmiert in C/C++, Qt und C# für Mono. Er arbeitet als Softwareentwickler und Trainer für die Hilf! GmbH.

Listing 3: »ConanWidget« hinzufügen und anzeigen.

```

01 #include <Conan.h>
02
03 ConanWidget widget;
04 widget.AddRootObject(myMainWindow); // optional
05 widget.AddRootObject(someOtherObject); // optional
06 widget.show();
  
```

Listing 4: Eine Beispiel-Anwendung mit Conan

```

01 #include <QtGui>
02 #include <Conan.h>
03
04 int main(int argc, char **argv)
05 {
06     QApplication app(argc, argv);
07
08     QWidget *w = new QWidget();
09     w->show();
10
11     QPushButton *button = new QPushButton(w);
12     button->setGeometry(10, 10, 100, 25);
13     button->show();
14
15     ConanWidget *c = new ConanWidget();
16     c->show();
17
18     return QApplication::exec();
19 }
  
```

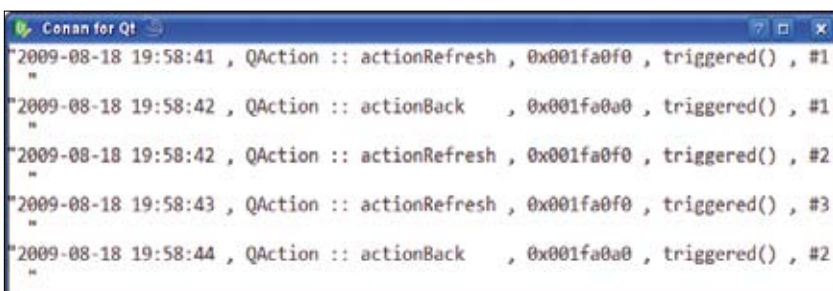


Abbildung 4: Alle gefeuerten Signale protokolliert dieses Konsolenfenster.