Knihovny pro tvorbu GUI

Filip Vaverka, **Petr Veigend**

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 00 Brno

veigend@fit.vut.cz

Požadavky na grafická uživatelská rozhraní

- Z pohledu uživatele
  - Rychlá odezva
  - Příjemný a uniformní vzhled, dobrá integrace s OS
- Z pohledu vývojáře
  - Snadná a rychlá implementace, možnost prototypování
  - Odděleni logiky aplikace od definice jejího vzhledu
- Z pohledu systému
  - Nízká náročnost na výkon HW (především v nečinnosti)
  - Efektivní využití HW prostředků daného systému (např. vykreslování pomocí GPU)

Základní principy implementace

- Obvykle založena na **nekonečné smyčce událostí** tzv. „event-loop“
  - Běží po celou dobu běhu aplikace
  - Periodicky se dotazuje operačního systému na příchozí události, na které reaguje
  - Nebo čeká na příchod další události (pomocí systémového volání)

```
while forever do
    události ← ZískatUdálosti()
    for U ∈ události do
        ZpracujUdálost(U)
    end
    Čekat(0.1s)
end
```

```
while forever do
    události ← ČekatNaUdálosti()
    for U ∈ události do
        ZpracujUdálost(U)
    end
end
```

Základní principy implementace

- Obvykle založena na **nekonečné smyčce událostí** tzv. „event-loop“
  - Běží po celou dobu běhu aplikace
  - Periodicky se dotazuje operačního systému na příchozí události, na které reaguje
  - Nebo čeká na příchod další události (pomocí systémového volání)

```
while forever do
    události ← ZískatUdálosti()
    for U ∈ události do
        ZpracujUdálost(U)
    end
    Čekat(0.1s)
end
```

```
while forever do
    události ← ČekatNaUdálosti()
    for U ∈ události do
        ZpracujUdálost(U)
    end
end
```

- Po dobu zpracování událostí (v obou případech) program nereaguje!

Událostmi řízené programování

- Aplikace nemusí testovat, zda vznikly nové události (případně stav vstupních zařízení)
- Události vznikají jednak na straně operačního systému
  - Uživatelský vstup, požadavky na vykreslení okna atd.
- Mohou také vznikat uvnitř aplikace samotné
  - Akce se zpožděním (časovače), komunikace mezi prvky uživatelského rozhraní apod.
- Usnadňuje komunikaci mezi paralelně běžícími částmi aplikace

- Multiplatformní
  - Qt (C++), GTK (C), SWING (Java), ...
- Microsoft Windows
  - WinForms, WPF (.NET), UWP, ...

- Jeden z nejpoužívanějších frameworků pro tvorbu GUI (1,5 miliónu vývojářů po celém světě)
- Kompletní vývojová platforma pro desktop, mobilní a vestavěné aplikace
  - Vývojové prostředí (QtCreator), meta systém pro překlad a sestavení (QMake), sada knihoven
- Multiplatformní
  - Linux, Windows, Android, iOS, ...
- Rozsáhlá sada knihoven/modulů pro usnadnění různých oblastí vývoje aplikací
  - Qt Core, Qt Widgets, Qt GUI, Qt Multimedia, Qt Network, Qt QML, Qt Quick, Qt SQL, Qt Test, ...
- **Rozšíření jazyka C++ o systém událostí**
  - Zajištěno kombinací generování kódu při překladu (QMake/MOC – *Meta-Object Compiler*) a knihoven

- Základní modul Qt obsahující hlavní rozšíření C++
- Meta-objektový systém
  - Umožňuje komunikaci mezi objekty pomocí signálů a slotů (viz dále)
  - Oproti C++ udržuje meta informace o objektu a jeho vlastnostech („property“)
  - Hierarchie objektů umožňuje automatické uvolňování objektů
  - Pomocí `QObject`, `Q_OBJECT`, `MOC`
- Vytvoření vlastního Qt objektu

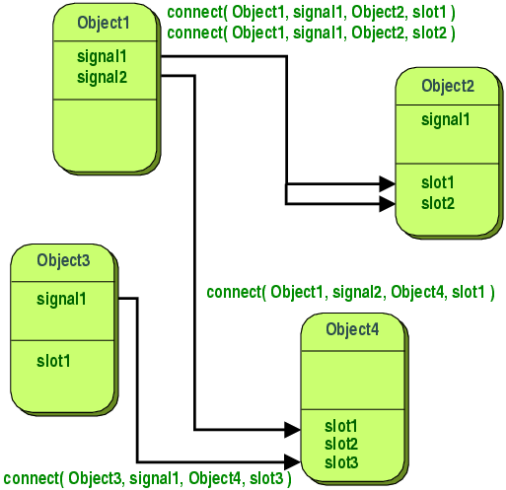
```
#include <QObject>
class MyObject : public QObject
{
    Q_OBJECT // makro, zpracuje se pomocí MOC
public:
    MyObject(QObject *parent = nullptr)
        : QObject(parent) { /* ... */ }
    // ...
};
```

- Vlastnosti („property“) objektu – proměnné, které objekt vystavuje vnějšímu světu
  - `Q_PROPERTY(<typ> <jméno> [READ <get-metoda>] [WRITE <set-metoda>] [NOTIFY <signál>])`
  - Zapouzdřuje koncept metod pro nastavení/získání hodnoty z objektu („getter-setter“)
  - Umožňuje propojení se systémem událostí (signálů a slotů)
  - Nezbytné pro využití systému Qt Quick viz dále
- `Q_PROPERTY(int x READ getX WRITE setX NOTIFY xChanged)`
  - Definuje vlastnost objektu s názvem „`x`“ typu „`int`“
  - Hodnota je získána pomocí metody „`getX`“ a změněna „`setX`“ (metody je nutné implementovat)
  - Okolí je o změně její hodnoty informováno signálem „`xChanged`“
- `Q_PROPERTY(int x MEMBER m_x)`
  - Zpřístupňuje členskou proměnnou „`int m_x;`“

```
#include <QObject>
class MyObject : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString x READ getX WRITE setX
        NOTIFY xChanged)
public:
    MyObject(QObject *parent = nullptr)
        : QObject(parent) { /* ... */ }

    QString getX(); // obdobné std::string
    void setX(const QString &x);

signals:
    void xChanged();
};
```

- Systém událostí – signály a sloty
  - Umožňuje komunikaci mezi jinak nezávislými objekty
  - Implementace návrhového vzoru pozorovatele „Observer“
  - Může (vlákna), ale obvykle nevyužívá frontu událostí

```
class Obj1 : public QObject
{ // ...
signals:
    void signal1();
};

class Obj2 : public QObject
{ // ...
signals:
    void signal1();
public slots:
    void slot1() { /* ... */ }
    void slot2() { /* ... */ }
};
```



```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    Obj1 obj1;
    Obj2 obj2;

    connect(
        obj1, &Sender::valueChanged,
        obj2, &Receiver::updateValue
    );

    MainWindow w;
    w.show();

    return a.exec(); // Event-loop
}
```

pozor na nutnost přetypování

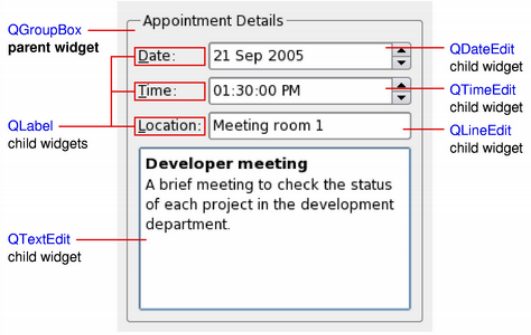
- tzv. funktorový zápis
- kontrola při překladu (starší zápis se kontroloval za běhu)
- podporuje implicitní přetypování
- nepodporuje převod funkcí z C++ do QML

...

```
// starý zápis
connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(PrintHello()));
// nový zápis
connect(ui->pushButton, &QPushButton::clicked ,this,&MainWindow::PrintHello);
...
```

- Obsahuje základní prvky pro tvorbu uživatelských rozhraní

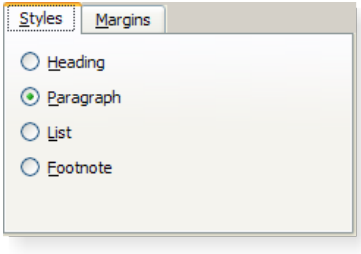
- Obsahuje základní prvky pro tvorbu uživatelských rozhraní
- QWidget – základní třída pro prvky grafického rozhraní

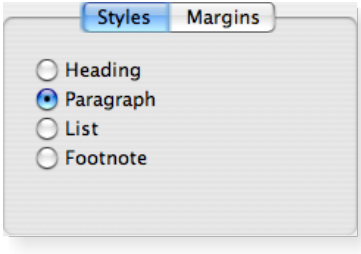


Obrázek: Základní prvky UI[1]

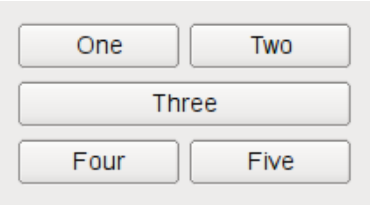
[1] https://doc.qt.io/qt-6/qtwidgets-index.html

- Obsahuje základní prvky pro tvorbu uživatelských rozhraní
- QWidget – základní třída pro prvky grafického rozhraní
- QStyle – zajišťuje vykreslování prvků rozhraní

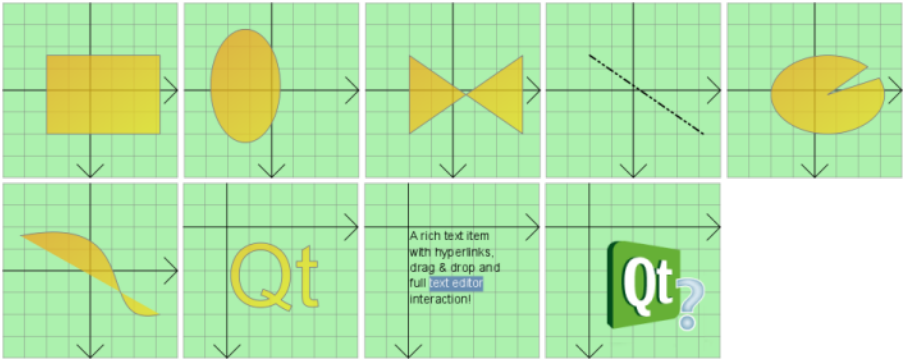




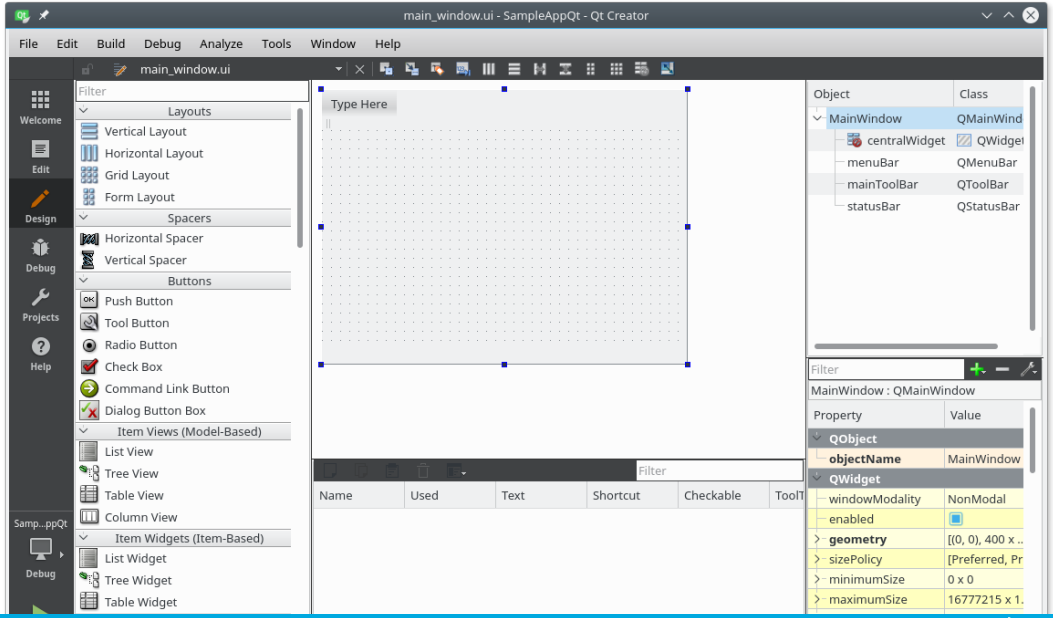
- Obsahuje základní prvky pro tvorbu uživatelských rozhraní
- QWidget – základní třída pro prvky grafického rozhraní
- QStyle – zajišťuje vykreslování prvků rozhraní
- QLayout – manažery rozložení prvků rozhraní



- Obsahuje základní prvky pro tvorbu uživatelských rozhraní
- QWidget – základní třída pro prvky grafického rozhraní
- QStyle – zajišťuje vykreslování prvků rozhraní
- QLayout – manažery rozložení prvků rozhraní
- Graphics View – zobrazování komplexní grafiky



- Grafický editor souborů s definicí GUI (XML)



- XML soubory s definicí GUI jsou následně zpracovány
  - Obvykle za překladu pomocí `QMake` a `UIC` na kód v C++
  - Za běhu aplikace pomocí `QUiLoader`
- Zpracováním za překladu vznikne hlavičkový soubor

`main_window.h`

```
#include <QMainWindow>

namespace Ui { class MainWindow; }

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
};
```

`main_window.cpp`

```
#include "main_window.h"
#include "ui_main_window.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Příklad

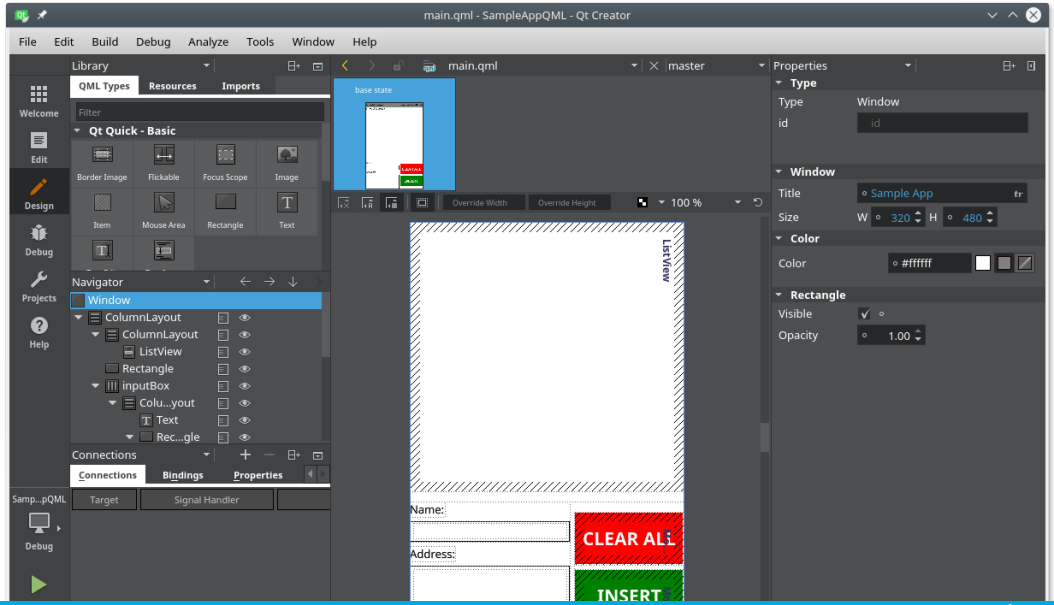
- nadstavba pro Python, která umožňuje rychlou tvorbu uživatelských rozhraní v Qt pro aplikace v Pythonu
- GUI se vygeneruje z `.ui` souboru pomocí příkazu `pyuic6` (Windows – `pyuic6.bat`)
- vygenerovaný soubor neobsahuje nic, co je implementováno ve zdrojových nebo hlavičkových souborech (např. obslužné funkce stisku tlačítka)
- příklad

Qt Quick (Qt a QML)

- Modernější alternativní systém pro tvorbu GUI v Qt
  - Základem je deklarativní jazyk `QML`, který je kombinací jazyků `JSON` a `JavaScript`
  - Umožňuje efektivní popis uživatelských rozhraní jako vizuálních komponent
  - Na základě tohoto popisu je za běhu vytvořen graf scény, jehož prvky jsou vykreslovány za použití HW akcelerace (např. OpenGL) asynchronně ve více vláknech

```
Rectangle {
    id: myButton
    width: 10
    height: 10
    color: "green"

    MouseArea {
        anchors.fill: parent
        onClicked: { /* JS ... */ }
    }
}
```

- Lze využít grafický editor v Qt Creatoru



- Propojení jádra aplikace (C++) se systémem QML
  - Pomocí signálů/slotů, vlastností (`Q_PROPERTY`) a metod objektů označených jako `Q_INVOKABLE`
  - Vyžaduje typové konverze (typ musí znát jak C++ tak QML)
  - Uživatelské typy v C++ musí být registrované:

```
// Hlavičkový soubor
class Typ : public QObject { Q_OBJECT /* ... */ };
// Kód
qmlRegisterType<Typ>("com.x.y", 1, 0, "Typ");
```

  - Instanci objektu je možné zpřístupnit nastavením proměnné kontextu:

```
Typ x; // Registrovaný uživatelský typ
QQmlApplicationEngine engine;
engine.rootContext()->setContextProperty("x", &x);
// ...
engine.load(/* QML Soubor */);
```

- Práce se signály v QML
- Přijetí signálu pomocí „Signal Handler“
  - Pojmenovány jako `on<Signál>` např. `onClicked` pro signál `clicked`

    ```
    MouseArea { /* ... */ onClicked: { /* JS */ } }
    ```
  - Pokud má signál parametry, jsou dostupné jako lokální proměnné pod jejich názvy

    ```
    onClicked: { console.log("Klik na", mouse.x); }
    ```
- Signály oznamující změnu hodnoty „property“ mají název ve tvaru `on<Property>Changed`
- Objekt propojení „Connection Type“ umožňuje připojení na signály libovolného objektu

```
Connections {
    target: idObjektu
    onClicked: { /* ... */ }
}
```

- Přídavné signály – rozšíření objektu o „nové“ signály

```
Rectangle {
    Component.onCompleted: { /* ... */ }
}
```

- Signály uživatelsky definovaných typů

```
// MyButton.qml
Rectangle {
    id: root
    signal activated()

    MouseArea {
        anchors.fill: parent
        onPressed: root.activated()
    }
}

// MyApp.qml
MyButton {
    onActivated: console.log("clicked")
}
```

- Práce s vlastnostmi v QML
- Hodnota může být jednorázově přiřazena

```
Component.onCompleted: { height = x.value; }
```

- Nebo může být trvale spojena s jinou – „binding“
  - Jednoduché propojení

```
Rectangle {
    height: x.value
}
```

  - Propojení výrazem nebo funkcí

```
Rectangle {
    height: 5*x.value
    width: { if(x.value < 5) return 5;
             else return x.value; }
}
```

  - Dynamické vytvoření propojení

```
Keys.onSpacePressed: {
    height = Qt.binding(function() { return 5*width })
}
```

```
int main(int argc, char *argv[])
{
    ObjX::registerType();
    ObjX x;

    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine; // QML engine
    engine.rootContext()->setContextProperty("x", &x);

    // Načtení *.qml z resource souboru
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec(); // Event-loop
}
```

- Součást překladového systému `RESOURCES += qml.qrc` v `*.pro` souboru
- Umožňují „přibalit“ soubory do výsledného spustitelného programu
- Obsah takto „přibalených“ souborů lze pak číst pomocí speciální cesty „`qrc:/`“. Např.:
  ```
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
  ```
- Vhodné pro data, jako jsou skripty v QML, obrázky, soubory jazykových mutací atd.

Další toolkity

- GTK+ (obdoba Qt, pro jazyk C, také existuje nadstavba pro Python)
- Tkinter (framework pro tvorbu GUI pro Python)
- ...

Děkuji za pozornost.