Projektowanie Graficznych Interfejsów Użytkownika

Robert Szmurło

Qt

Trolltech

http://www.trolltech.com/

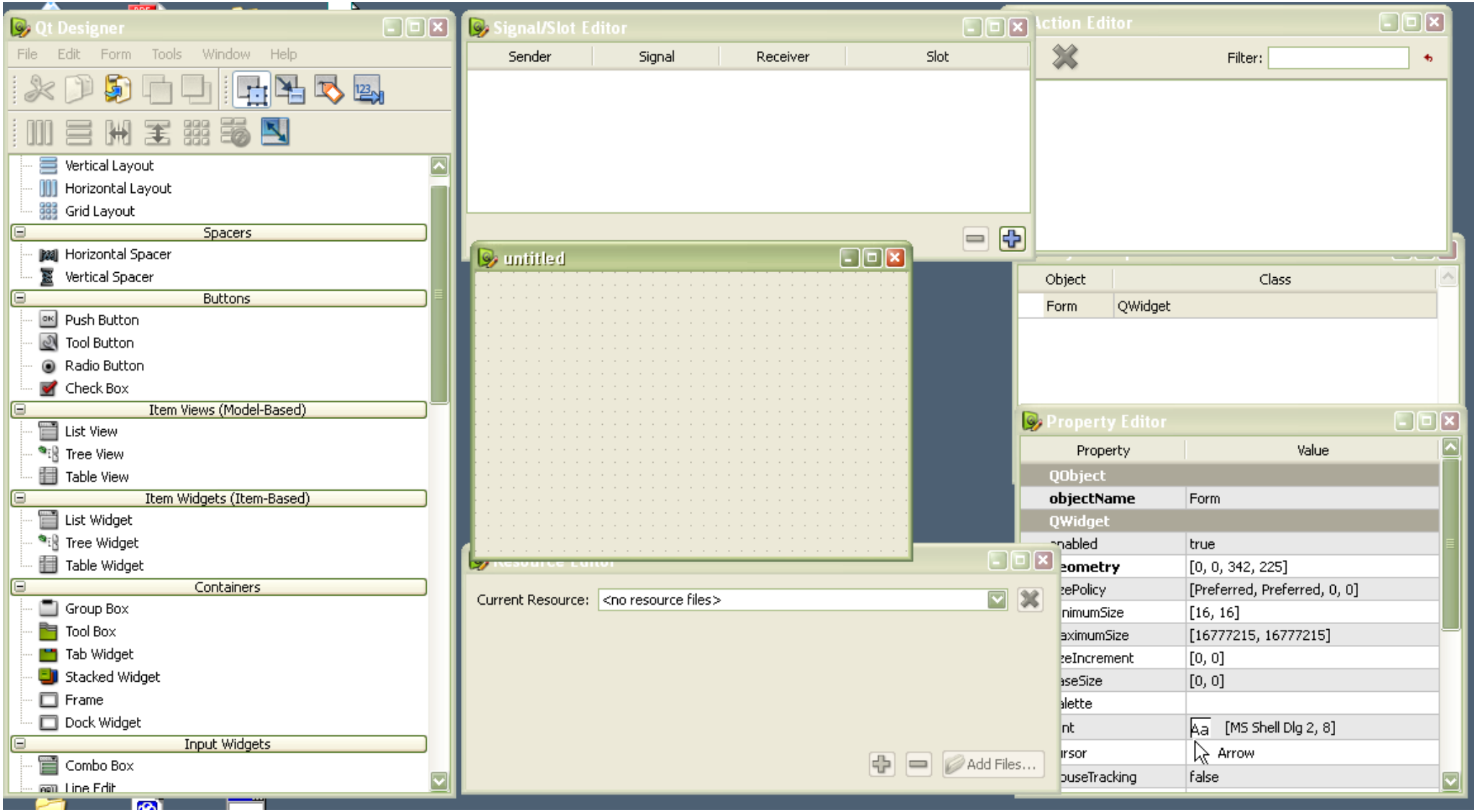
Qt Wprowadzenie

- Twórcy: norweska firma Trolltech, Qt jest ich głównym produktem.
- Podstawowy język biblioteki: **C++**
    - Pozostałe języki: Java, Python, Mono, i wiele innych.
- Qtopia – środowisko oraz biblioteka dla urządzeń przenośnych.
- Zastosowania Qt:
  - aplikacje wieloplatformowe (Mac OS X, Windows, Unix)
  - głównie aplikacje techniczne, specjalistyczne, ale
  - przede wszystkim KDE.
  - oraz Borland z Kylix.
- Kto tego używa Qt?

Co może biblioteka Qt oprócz GUI?

- Szeroki zestaw klas pozwalający na budowanie aplikacji opartych na graficznym interfejsie użytkownika.
- Podsystem sieciowy. Niezależny od platformy bardzo wygodny interfejs gniazd oparty na sygnałach/slotach.
- Podsystem SQL, możliwość dołączania własnych sterowników. Funkcjonalność podobna do JDBC. Obsługiwane: MySQL, Oracle, ODBC, PostgreSQL, Sybase, IBM DB2
- Wsparcie dla XML, parser SAX 2 (Simple API for XML), DOM Level 2.

Qt – Designer

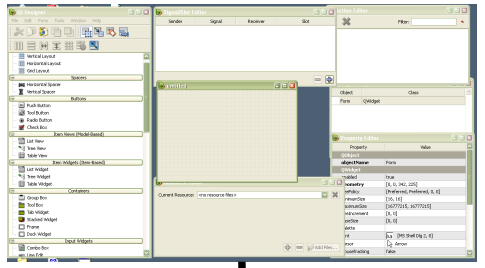

Qt Designer
File Edit Form Tools Window Help
Vertical Layout
Horizontal Layout
Grid Layout
Spacers
Horizontal Spacer
Vertical Spacer
Buttons
Push Button
Tool Button
Radio Button
Check Box
Item Views (Model-Based)
List View
Tree View
Table View
Item Widgets (Item-Based)
List Widget
Tree Widget
Table Widget
Containers
Group Box
Tool Box
Tab Widget
Stacked Widget
Frame
Dock Widget
Input Widgets
Combo Box
Line Edit
Signal/Slot Editor
Sender
Signal
Receiver
Slot
Action Editor
Filter:
untitled
Object
Class
Form
QWidget
Property Editor
Property
Value
QObject
objectName
Form
QWidget
true
[0, 0, 342, 225]
[Preferred, Preferred, 0, 0]
[16, 16]
[16777215, 16777215]
[0, 0]
[0, 0]
[MS Shell Dlg 2, 8]
Arrow
false
Current Resource:
<no resource files>
Add Files...

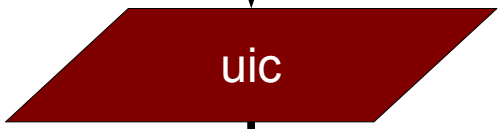
Qt Designer – Cechy charakterystyczne

- Służy tylko do tworzenia projektów wizualnych.
  - Dodatkowo potrafi łączyć część komunikatów pomiędzy elementami widoku.
- Kolejne wersje ewoluowały. Trolltech nie mogła zdecydować się na konkretny profil tego narzędzia:
    - pierwsze wersje służyły tylko do projektowania,
    - kolejne wersje umożliwiały również częściową edycję kodu
    - najnowsza wersja wraca do koncepcji tylko projektu wizualnego
- Wszystkie wersje Qt Designera były dostosowane do integracji z zintegrowanymi środowiskami programistycznymi:
    - kdevelop (Linux)
    - Visual Studio
    - i w końcu **Eclipse**

Qt Designer – Cechy charakterystyczne

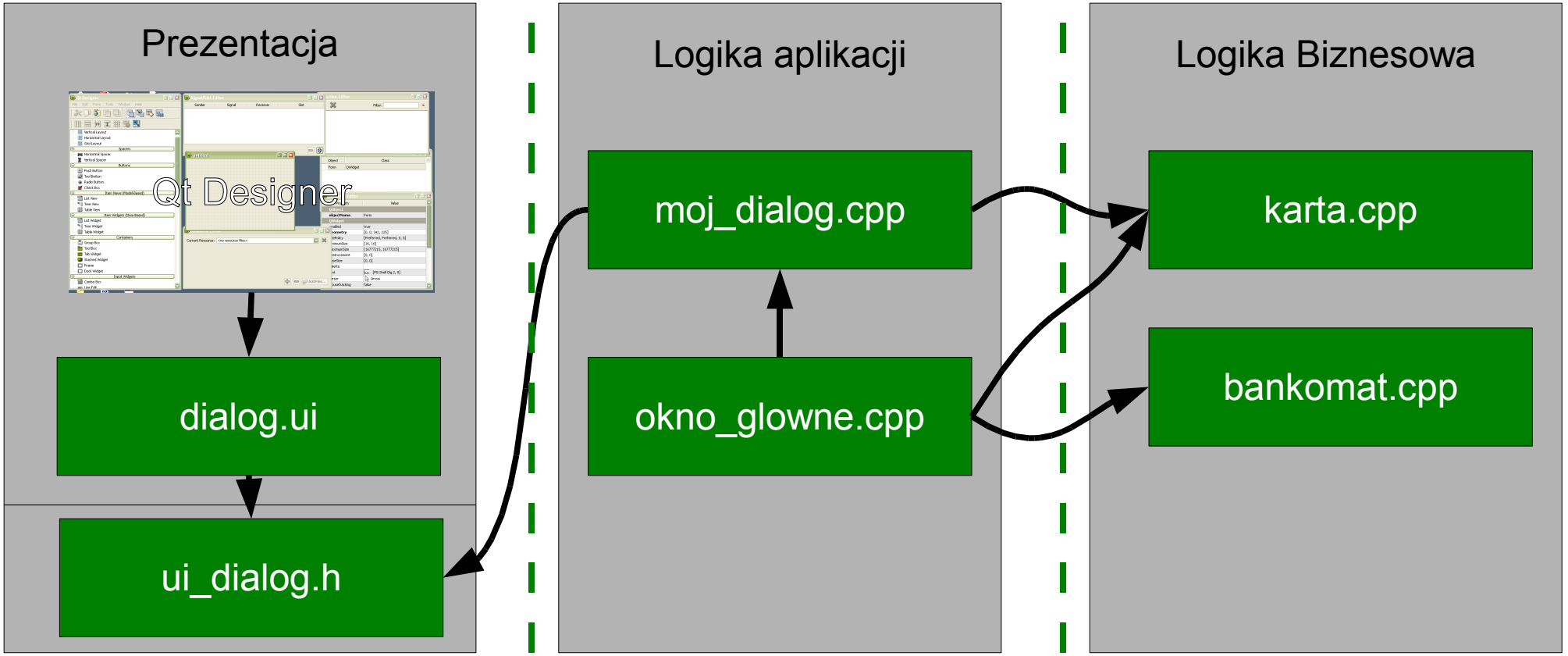
- Projekt udostępnia ograniczoną liczbę komponentów, nastawiając się na możliwość dodawania własnych dostosowanych do konkretnych potrzeb (jaki to wzorzec projektowy?)
- Qt Designer zapisuje układ elementów na ekranie oraz ich projekt leksykalny w pliku z rozszerzeniem .ui (format xml).
  - Plik ten jest za pomocą specjalnego narzędzia (uic) transformowany na kod w języku C++ (ui_dialog.h), który tworzy układ okienka.
- Oczywiście w środowiskach zintegrowanych cały proces jest zautomatyzowany (oprócz dołączania pliku nagłówkowego (ui_dialog.h))!





#include <ui_dialog.h>

Jaki to wzorzec?

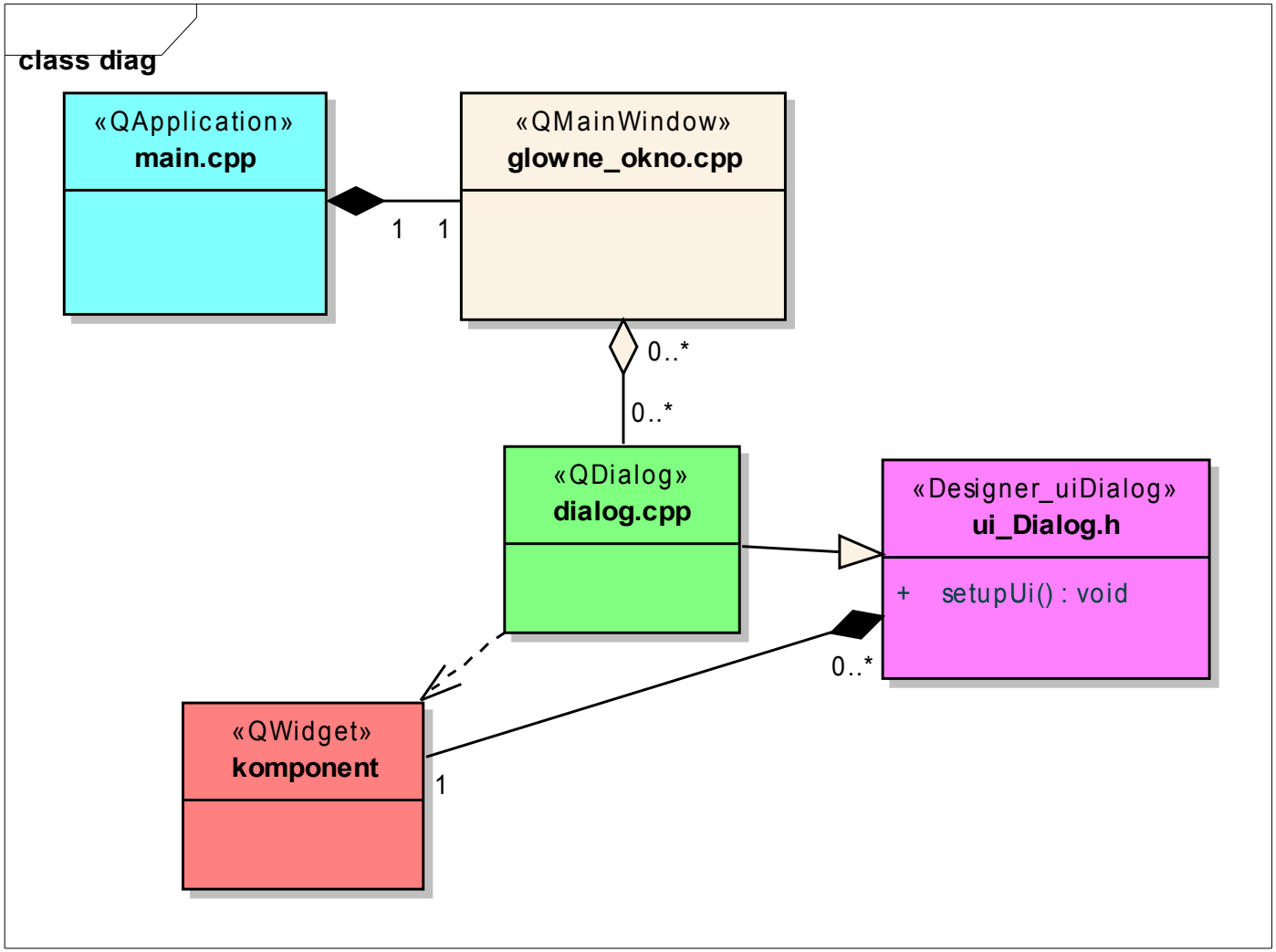


Pasywny widok/prezentacja

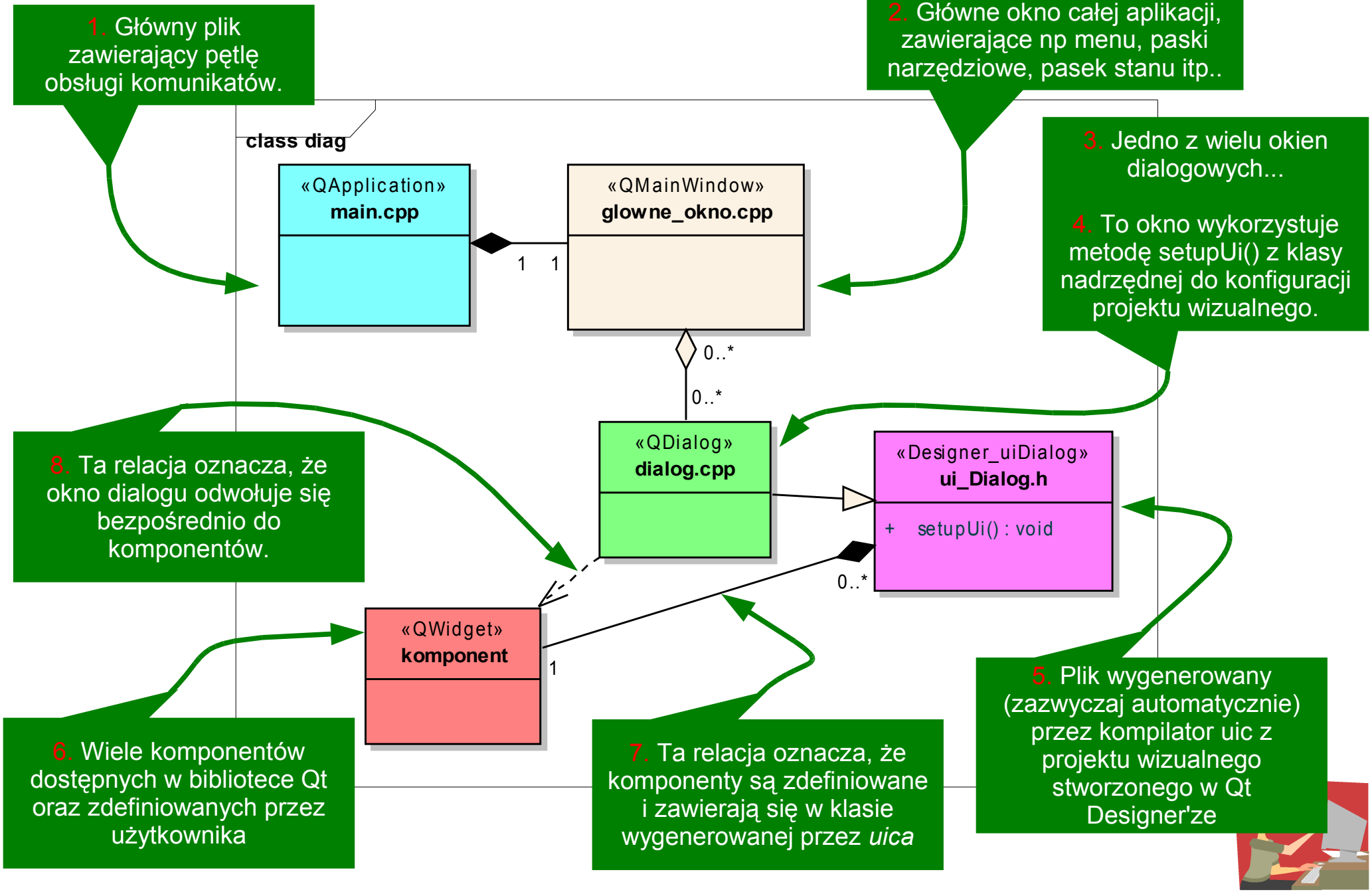
Aktywna logika aplikacji, która zajmuje się nie tylko reakcją na komunikaty od użytkownika, ale także w odpowiedni sposób wyświetla dane na ekranie.

Całe szczęście! Niezależna od nikogo logika biznesowa, która musi udostępniać informacje o swoim stanie za pomocą odpowiedniego interfejsu.

Struktura programu GUI w Qt

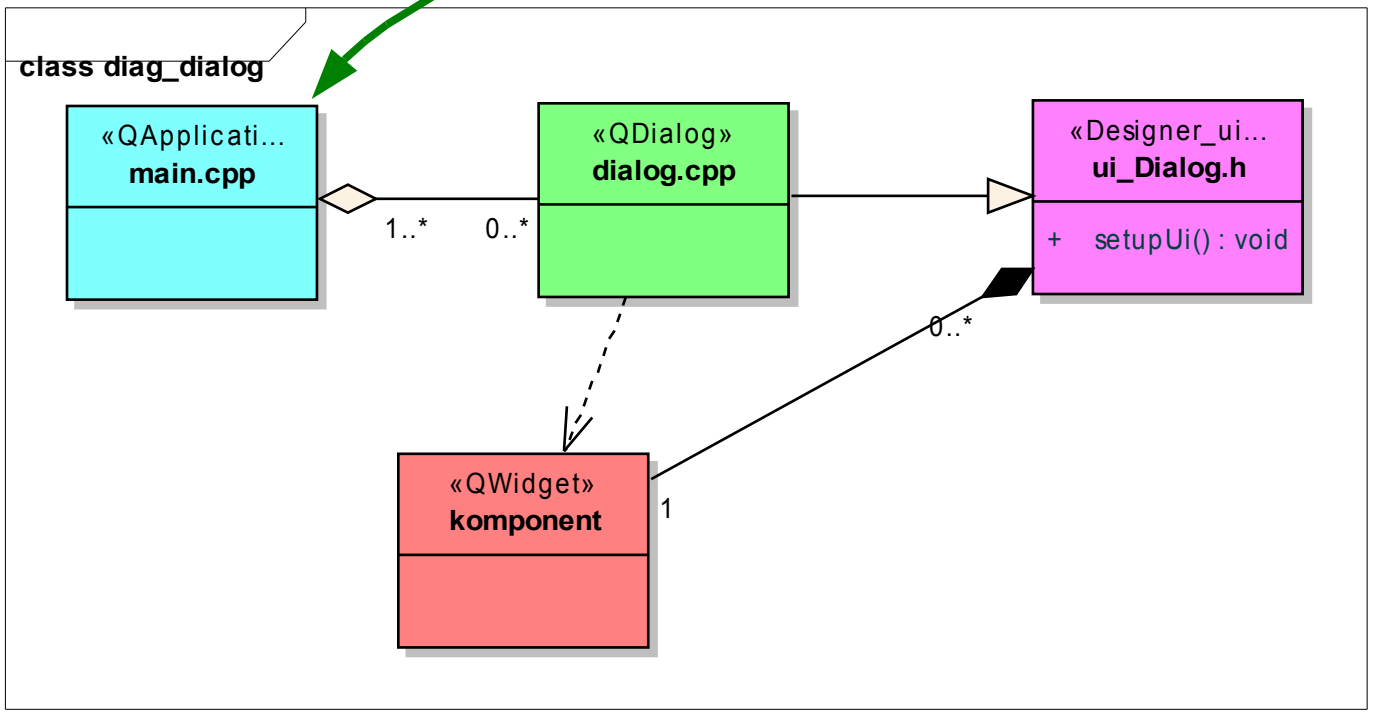


Struktura programu GUI w Qt - wyjaśnienie



Przykład bez głównego okna

- W prostszym przykładzie zakładamy, że główny program tworzy bezpośrednio okno dialogowe.



Kompletny kod

dialog.h

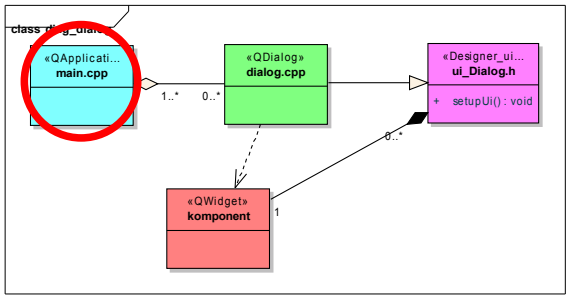
```
#include <QDialog>
#include <ui_Dialog.h>

class MojDialog : public QDialog,
                  private Ui::ui_Dialog{

    Q_OBJECT  // to jest potrzebne ze względu
              // na sygnały i sloty w Qt!
public:
    MojDialog(QWidget * parent);
};
```

ui_Dialog.h

main.cpp

```
#include <QApplication>
#include <CFormLogin.h>

MojDialog::MojDialog(QWidget * parent = 0)
{
   setupUi(this);
   QObject::connect( cmdCancel, SIGNAL(clicked()),
                     this, SLOT( accept() ) );
   setWindowTitle( tr("Pojedyncze dziedziczenie.") );
}

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

         MojDialog * dialog = new MojDialog;
         dialog->show();

         return app.exec();
}
```

main.cpp - (logika aplikacji)



main.cpp

```
#include <QApplication>
#include <QPushButton>

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

    MojDialog * dialog = new MojDialog;
    dialog->show();

    return app.exec();
}
```

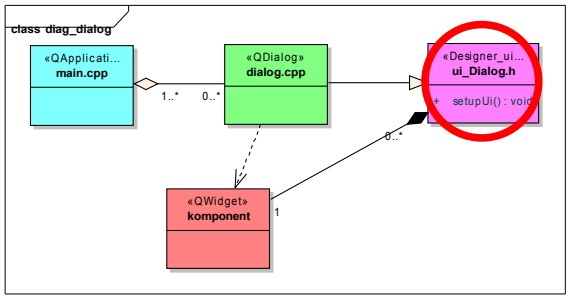
Pliki nagłówkowe, czyli importujemy biblioteki.

Główny obiekt aplikacji.

Przykładowe okno do pokazania.

Wystartowanie pętli obsługi komunikatów.

ui_Dialog.h - (prezentacja)



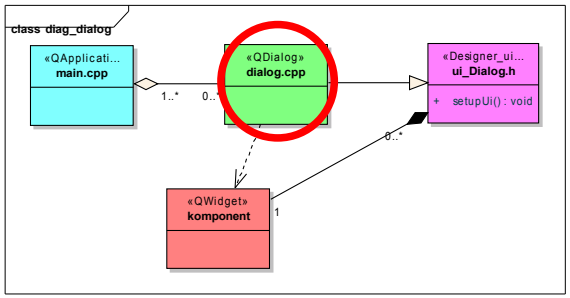

```
#include <QtCore/QVariant>
(...)
#include <QtGui/QWidget>

class auto_ui_Dialog
{
public:
    QGridLayout *gridLayout;
    QFrame *frame_2;
(....)
    QSpacerItem *spacerItem1;
    QPushButton *cmdCancel;
    QPushButton *cmdOK;

    void setupUi(QWidget *FormLogin)
    {

    FormLogin->setObjectName(QString::fromUtf8("FormLogin"));
    FormLogin->resize(QSize(362, 177).expandedTo(FormLogin->minimumSizeHint()));
    gridLayout = new QGridLayout(FormLogin);
    gridLayout->setObjectName(QString::fromUtf8("gridLayout"));
    QSizePolicy sizePolicy1(static_cast<QSizePolicy::Policy>(0), static_cast<QSizePolicy::Policy>(0));
    cmdOK = new QPushButton(frame);
    cmdOK->setObjectName(QString::fromUtf8("cmdOK"));
    cmdOK->setDefault(true);
    cmdOK->setFlat(false);
(...)
    QWidget::setTabOrder(txtPassword, cmdOK);
    QWidget::setTabOrder(cmdOK, cmdCancel);
   }

};

namespace Ui {
    class ui_Dialog: public auto_ui_Dialog {};
} // namespace Ui
```

Plik jest bardzo długi i tak naprawdę nie powinniśmy musieć nigdy go analizować. Na pewno nie można go ręcznie modyfikować, ponieważ jest automatycznie generowany przez kompilator uic.

Z naszego kodu odwołujemy się do kontrolek za pomocą nazw które zdefiniowaliśmy w Qt Designer.

Ten obiekt będziemy dziedziczyć w naszej klasie.

dialog.cpp - (logika aplikacji)



Stosujemy dziedziczenie. Dziedziczymy obiekt ui_Dialog, który jest automatczynie generowany przez *uic*

Wywołujemy metodę setupUi() do utworzenia układu wizualnego.

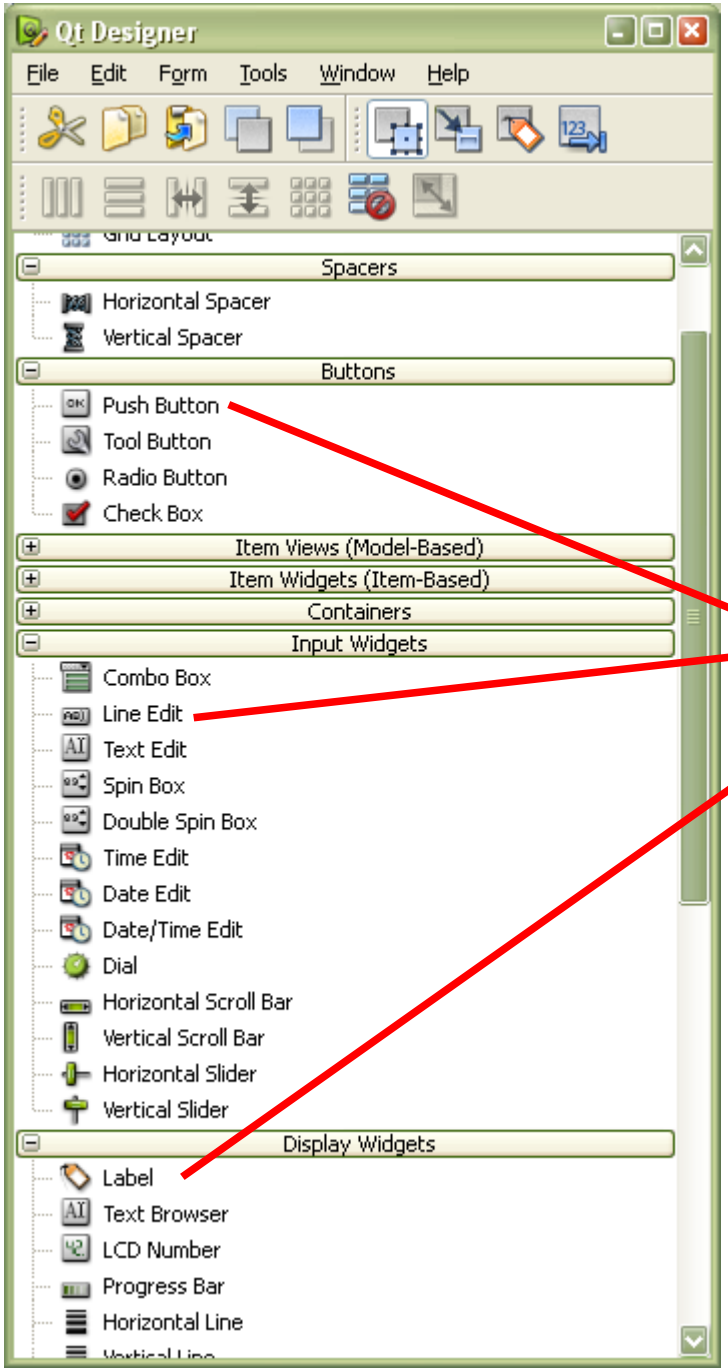
dialog.h

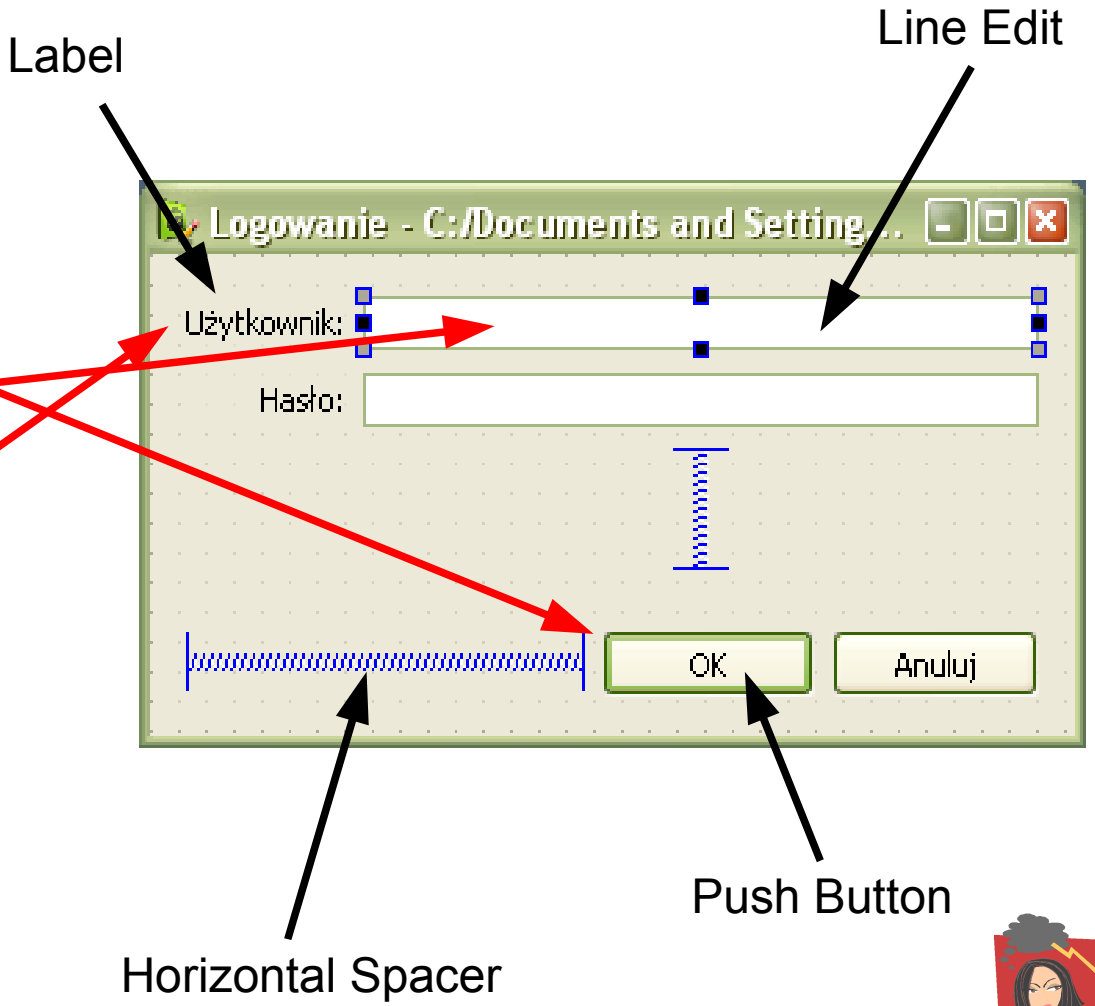
```
#include <QDialog>
#include <ui_Dialog.h>

class MojDialog : public QDialog,
                  private Ui::ui_Dialog{

    Q_OBJECT  // to jest potrzebne ze względu
              // na sygnały i sloty w Qt!
public:
    MojDialog(QWidget * parent);
};
```

dialog.cpp

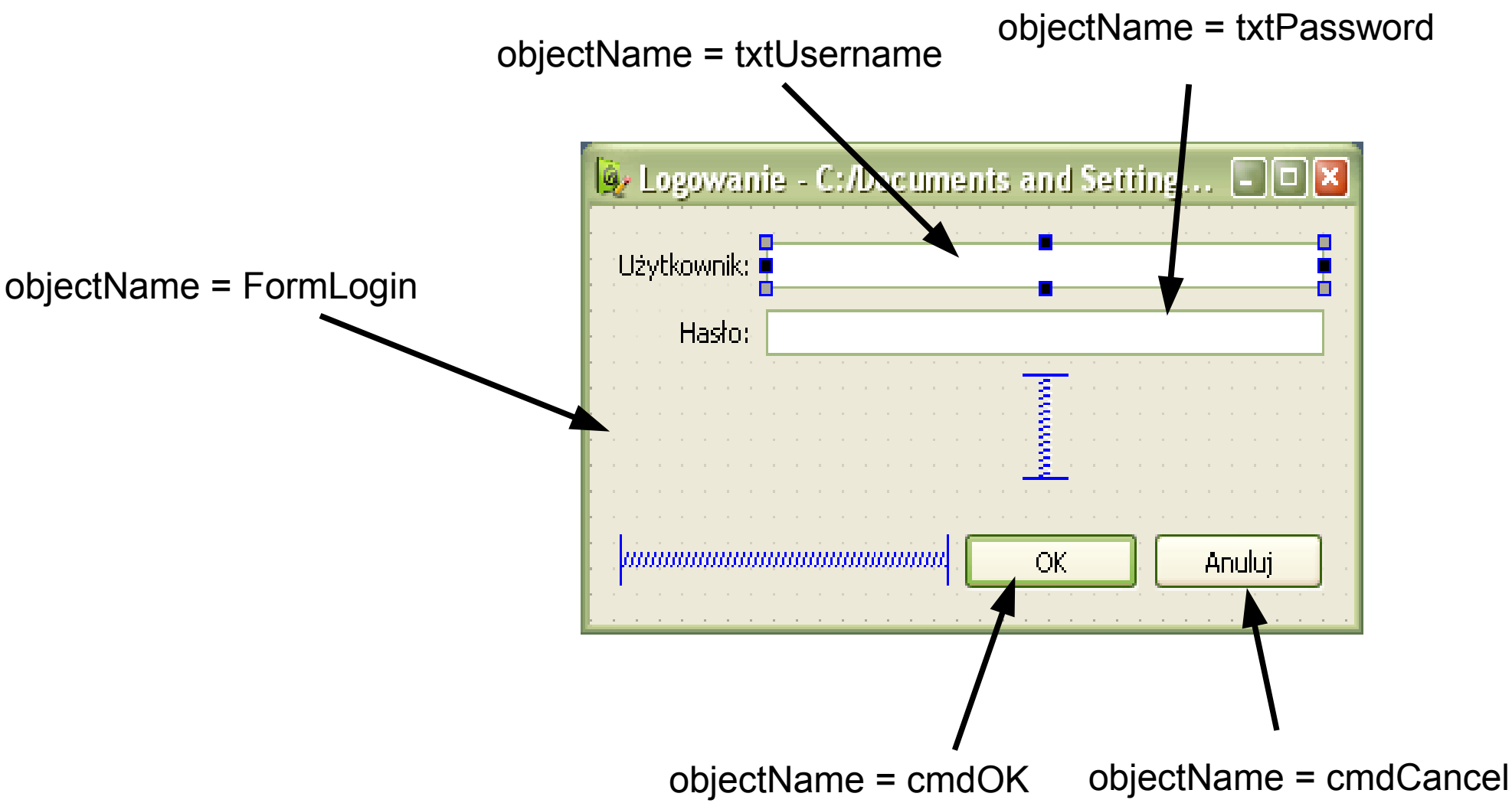
```
#include <QApplication>
#include <CFormLogin.h>

MojDialog::MojDialog(QWidget * parent = 0)
{
    setupUi(this);

    QObject::connect( cmdCancel, SIGNAL(clicked()),
                      this, SLOT( accept() ) );
    setWindowTitle( tr("Wielokrotne dziedziczenie.") );
}
```

Projekt wizualny w Qt Designer



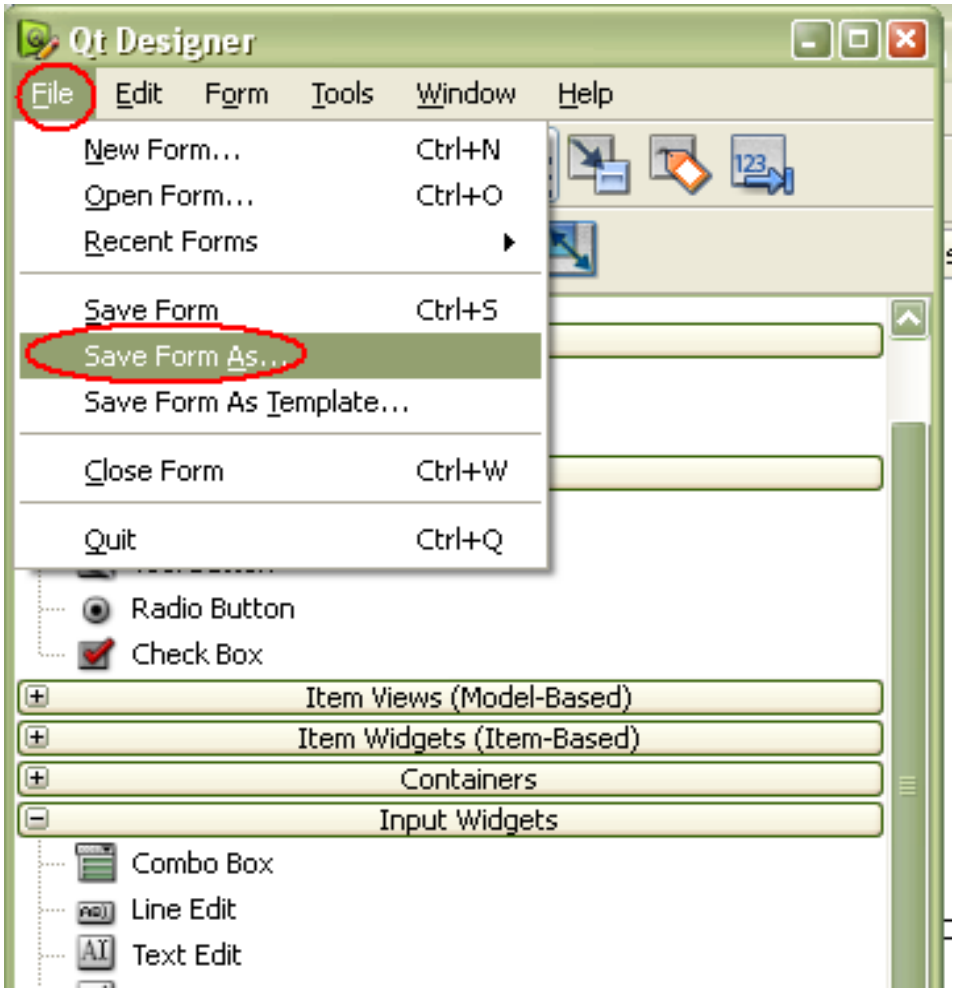


Definiujemy nazwy komponentów



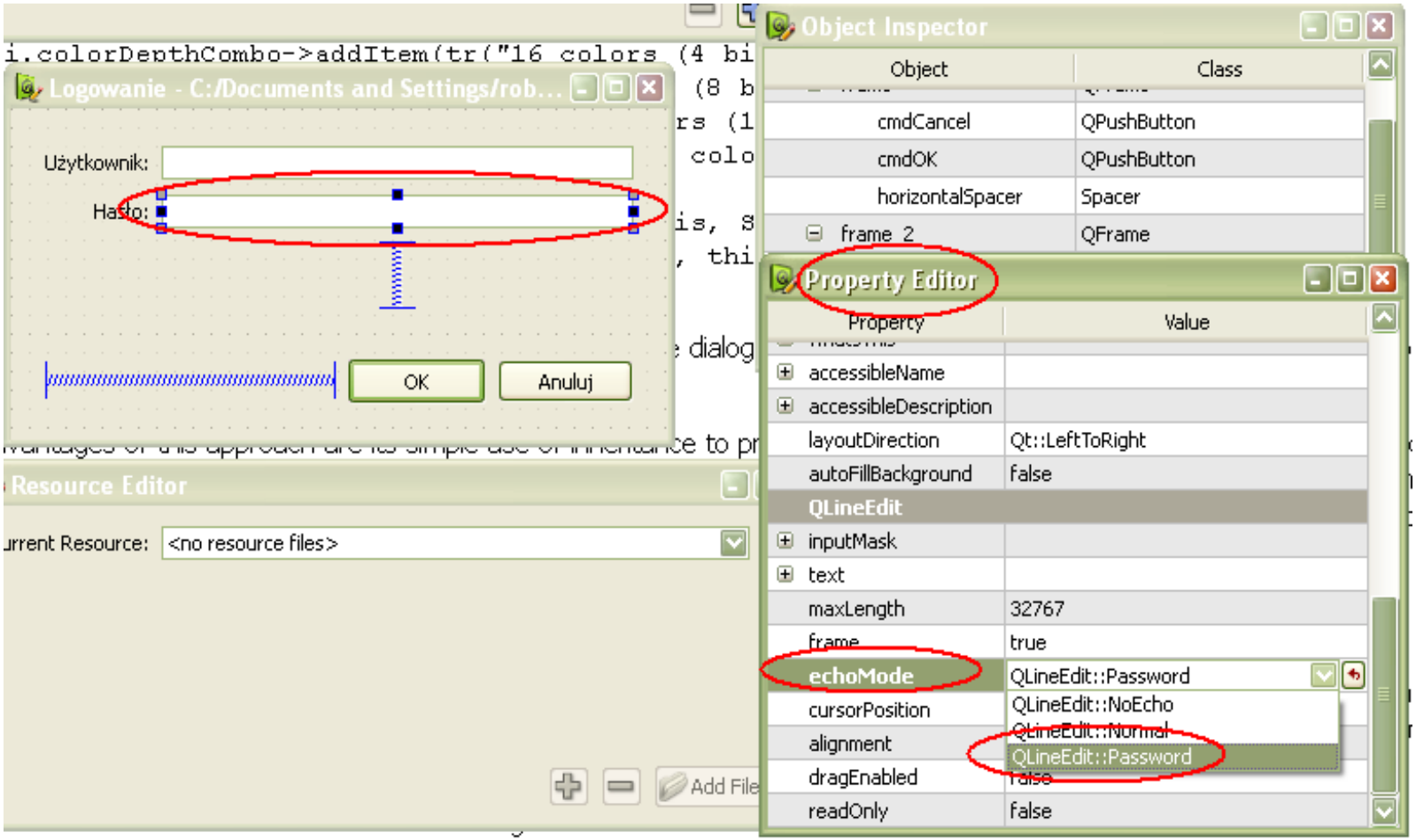
Zapiszmy nasz formularz

- W tym samy katalogu co przed chwilą utworzyliśmy projekt zapisujemy po nazwą: `FormDialog.ui`



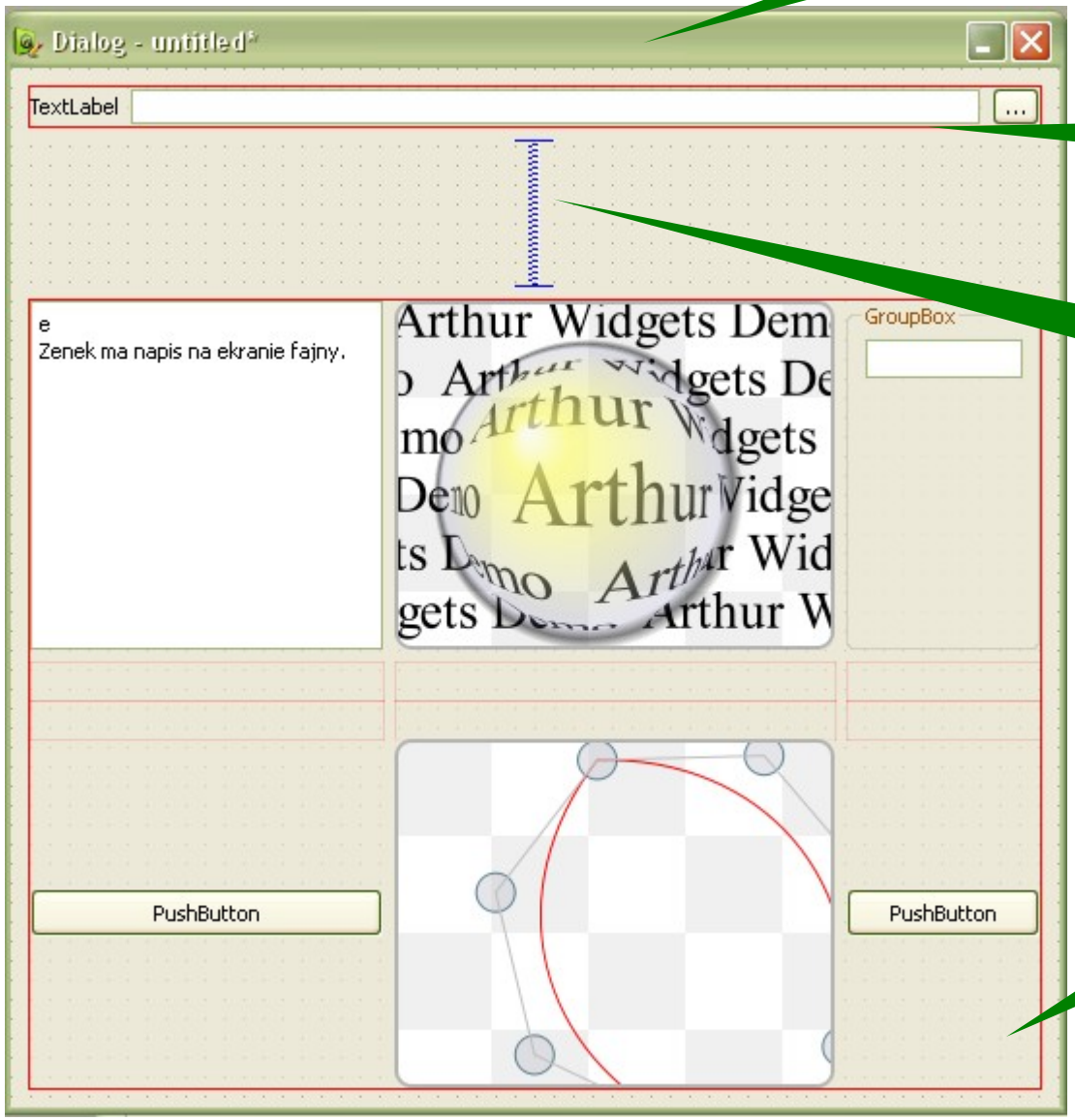
Użyteczność - Zakryj hasło

- Wracamy do Designera, modyfikujemy atrybut echoMode, zapisujemy.



Tworzenie układu - layoutu

Okno posiada główny układ *(ang. layout)*

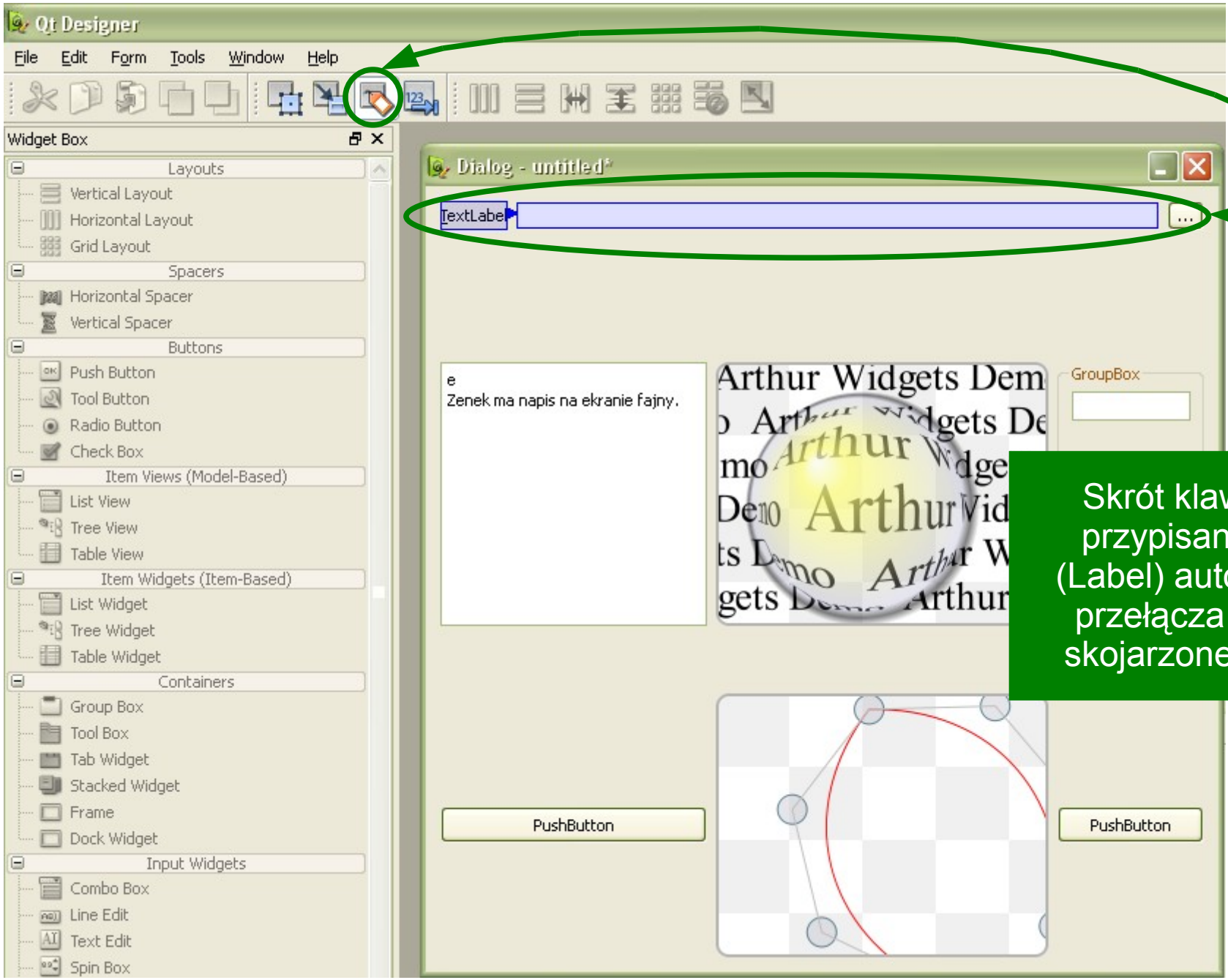


Elementy można grupować w mniejsze układy

Układy mogą być rozmieszczane na ekranie wykorzystując puste wypełnienie

Duży układ w formie siatki

Buddy elements – Elementy skojarzone

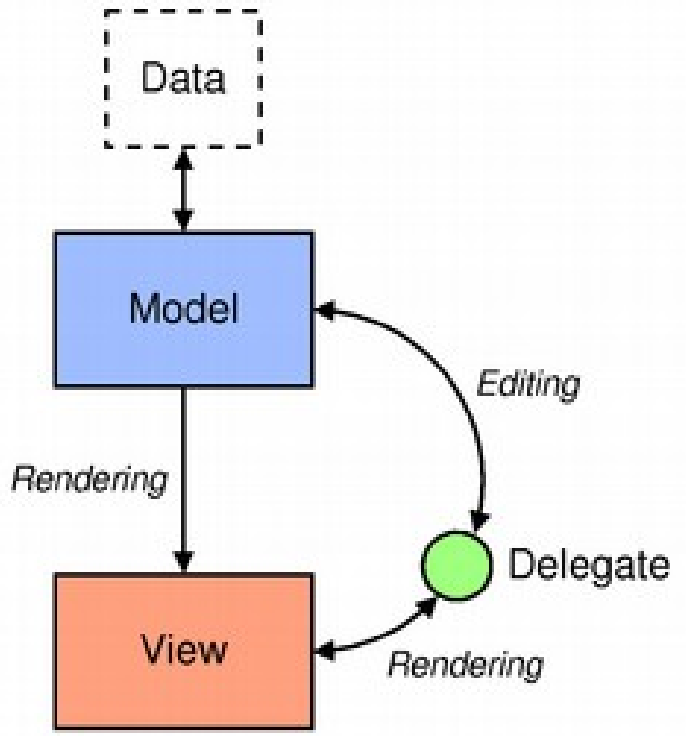


Skrót klawiaturowy przypisany opisowi (Label) automatycznie przełącza kursor do skojarzonej kontrolki.

Qt ..., a wzorzec MVC?

Programowanie Model-View

- Architektura model/view
  - The model communicates with a source of data, providing an interface for the other components in the architecture. The nature of the communication depends on the type of data source, and the way the model is implemented.
  - The view obtains model indexes from the model; these are references to items of data. By supplying model indexes to the model, the view can retrieve items of data from the data source.
  - In standard views, a delegate renders the items of data. When an item is edited, the delegate communicates with the model directly using model indexes.



- Models, views, and delegates communicate with each other using signals and slots:
  - Signals from the model inform the view about changes to the data held by the data source.
  - Signals from the view provide information about the user's interaction with the items being displayed.
  - Signals from the delegate are used during editing to tell the model and view about the state of the editor.
-

Tworzenie nowego modelu

- Model musi spełniać pewne wymagania.
- Różne elementy interfejsu muszą udostępniać różne metody dostępu.
- Dla przykładu model udostępniający dane dla kontrolek z listami napisów.

Plik nagłówkowy:

```
class StringListModel : public QAbstractListModel
{
    Q_OBJECT
public:
    StringListModel(const QStringList &strings, QObject *parent = 0)
        : QAbstractListModel(parent), stringList(strings) {}
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role = Qt::DisplayRole) const;
private:
    QStringList stringList;
};
```

Tworzenie nowego modelu

- Przykładowa implementacja metody zwracającej różne dane o wskazanym elemencie dla różnych kontekstów:

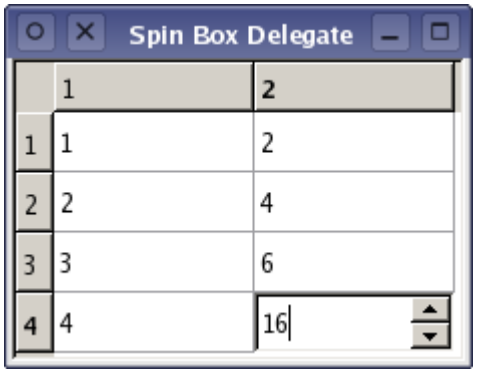
```
QVariant StringListModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();
    if (index.row() >= stringList.size())
        return QVariant();
    if (role == Qt::DisplayRole)
        return stringList.at(index.row());
    else
        return QVariant();
}
```

- W przypadku kontrolek indeksujących dane dwuwymiarowo (tabele) przykładowa implementacja może wyglądać:

```
QVariant StringListModel::headerData(int section, Qt::Orientation orientation,
                                     int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    if (orientation == Qt::Horizontal)
        return QString("Column %1").arg(section);
    else
        return QString("Row %1").arg(section);
}
```

Delegacje dla model/view w Qt

- Jeżeli chcemy dla danego pola utworzyć własną obsługę edycji danych musimy utworzyć własna klasę delegacji, a następnie zarejestrować ją w komponencie.
- Nowa delegacja musi implementować abstrakcyjna klasę QitemDelegate.
- Przykładowa deklaracja delegacji może wyglądać:




```
class SpinBoxDelegate : public QItemDelegate
{
    Q_OBJECT
public:
    SpinBoxDelegate(QObject *parent = 0);
    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option,
                          const QModelIndex &index) const;
    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
                      const QModelIndex &index) const;
    void updateEditorGeometry(QWidget *editor,
        const QStyleOptionViewItem &option, const QModelIndex &index) const;
};
```

Delegacje dla model/view w Qt

- Przykładowa implementacja metod tworzącej edytor na żądanie:

```
QWidget *SpinBoxDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &/* option */,
    const QModelIndex &/* index */) const
{
    QSpinBox *editor = new QSpinBox(parent);
    editor->setMinimum(0);
    editor->setMaximum(100);
    return editor;
}
```

- Uaktualniająca dane w edytorze.

```
void SpinBoxDelegate::setEditorData(QWidget *editor,
                                    const QModelIndex &index) const
 {
     int value = index.model()->data(index, Qt::DisplayRole).toInt();

     QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
     spinBox->setValue(value);
 }
```

- Uaktualniająca model:

```
void SpinBoxDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
                                   const QModelIndex &index) const
{
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->interpretText();
    int value = spinBox->value();
    model->setData(index, value);
}
```

- Dziękuję za uwagę.

- Chcemy być coraz lepsi!
- Jeżeli coś cię zainteresowało napisz e-maila:
  - robert@iem.pw.edu.pl
- Jeżeli coś cię bardzo znudziło napisz e-maila:
  - robert@iem.pw.edu.pl
- Jeżeli zauważyłeś błąd napisz e-maila:
  - robert@iem.pw.edu.pl