Interfejsy aplikacji w środowisku Windows

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(20 lutego 2023)

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

Kontakt

Budynek C, pokój 3.26
sk@tu.kielce.pl

Materiały do pobrania, aktualności, terminy zaliczeń

http://staff.tu.kielce.pl/sk

Organizacja wykładów

- Wykłady są nieobowiązkowe (zgodnie z postanowieniami regulaminu), ale...
- Na wykłady czasem warto zajrzeć.

Warunki zaliczenia wykładu

Egzamin zaliczeniowy po zakończeniu wykładów.

Organizacja laboratoriów

- Zajęcia laboratoryjne są obowiązkowe.
- Dopuszcza się jedną nieobecność.
- Większa liczba nieobecności powoduje zmniejszenie oceny do niedostatecznej włącznie (3 lub więcej nieobecności).
- W przypadku usprawiedliwionej nieobecności zajęcia można odrobić z inną grupą (jeśli istnieje taka możliwość).

Warunki zaliczenia laboratoriów

Wykonanie ćwiczeń i zaliczenie sprawdzianów kontrolnych.

Treść wykładów

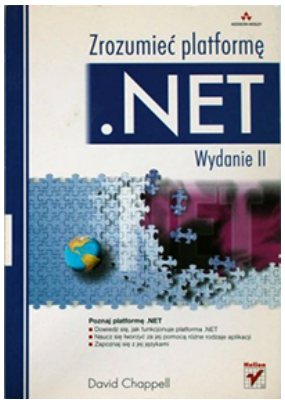
- Wprowadzenie do języka orientowanego obiektowo. Kompilatory i środowiska programistyczne dostępne dla systemu operacyjnego Windows.
- Zmienne i typy. Instrukcje sterujące, tablice i listy. Klasy, obiekty, metody.
- Operacje na tekstach, znaki specjalne, przetwarzanie łańcuchów znaków.
- Aplikacje konsolowe z parametrami.
- Biblioteki programistyczne w systemie operacyjnym Windows.
- Interfejs graficzny, budowa, zadania. Technologie budowy interfejsu graficznego w systemie operacyjnym Windows (Windows Form, WPF lub inne).
- Organizacja dostępu do bazy danych. Wybrana technologia i obsługa baz danych.

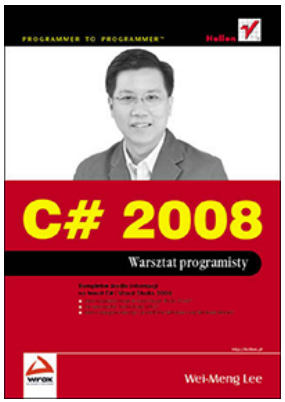
Treść laboratoriów

- Środowisko programowania. Struktura projektu.
- Budowa prostej aplikacji orientowanej obiektowo. Przetwarzanie warunkowe i iteracyjne. Aplikacje przetwarzające łańcuchy znaków.
- Budowa aplikacji konsolowej z obsługą wywołania parametrycznego. Obsługa parametrów linii poleceń systemu Windows.
- Opracowanie aplikacji z graficznym interfejsem użytkownika. Aplikacje sterowane zdarzeniami.
- Współpraca z plikami. Cechy systemu plików w systemie operacyjnym Windows.
- Tworzenie przykładowej bazy danych w wybranym systemie zarządzania relacyjnymi bazami danych. Budowa aplikacji do przeglądania tabeli bazy danych.
- Edycja danych w tabeli, obsługi powiązanych tabel w bazie danych.

Literatura

- Chappell D. *Zrozumieć platformę .NET.* Wydawnictwo Helion, Gliwice 2007.
- Lee W.M. *C# 2008. Warsztat programisty.* Wydawnictwo Helion, Gliwice 2010.
- Matulewski J. *C# 3.0 i .NET 3.5. Technologia LINQ.* Wydawnictwo Helion, Gliwice 2008.
- Michelsen K. *Język C#. Szkoła programowania.* Wydawnictwo Helion, Gliwice 2007.
- Ganczarski J., Owczarek M. *C++ Wykorzystaj potęgę aplikacji graficznych.* Wydawnictwo Helion, Gliwice 2008.





1 Informacje ogólne

2 Przetwarzanie tablic znaków i łańcuchów znaków

3 Obsługa wywołania parametrycznego

4 Graficzny interfejs użytkownika

5 Podstawy Windows API

6 Qt wprowadzenie

7 Sygnały i sloty

8 Kontrolki i szablony

9 Qt Designer

10 Wiele okien

11 Typy i kolekcje

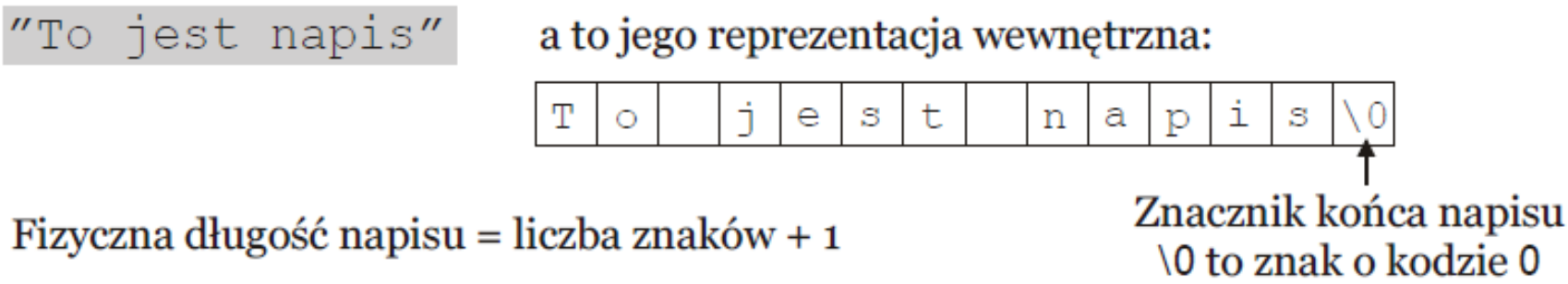
12 Obsługa plików

Łańcuchy znakowe jako tablice znaków

Typ char jest wbudowanym typem języków C/C++ służącym do reprezentacji pojedynczego znaku o wielkości 1 bajta. Do reprezentacji łańcuchów znaków wykorzystuje się zwykłe tablice znakowe.

Tablice takie nie różnią się od innych tablic w języku C, wprowadzano jedynie *kilka udogodnień*, czyniących łatwiejszym manipulowanie takimi tablicami.

W języku C przyjęto koncepcję łańcuchów ze **znacznikiem końca** (*null terminated strings*).



Tablice znakowe można inicjować w zwykły sposób, przewidziany dla tablic:

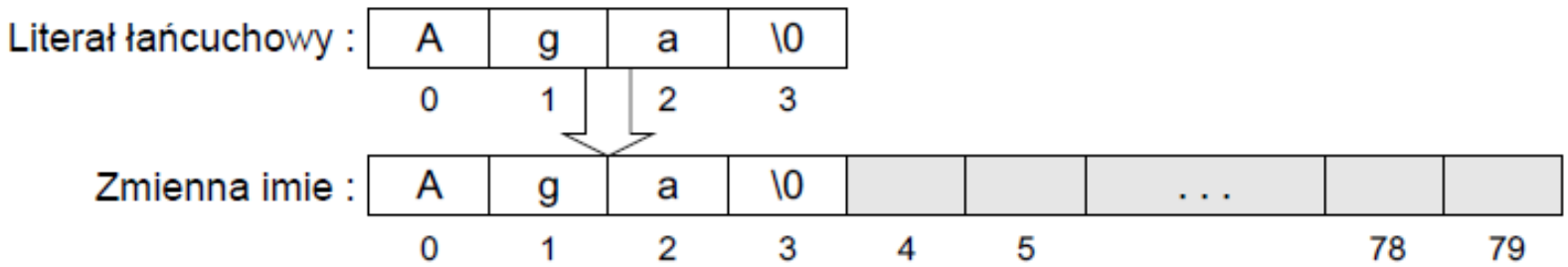
```
const int n = 80;
char name[n] = {'A', 'g', 'a'}; // czy dobrze?
char name[n] = {'A', 'g', 'a', '\0'};
```

Można wykorzystywać wygodniejszą formę:

```
const int n = 80;
char name[n] = "Aga";
```

lub

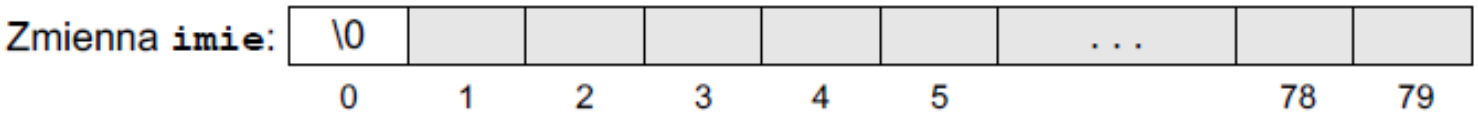
```
char imie[] = "Aga";
```



Deklaracja łańcucha zainicjowanego napisem pustym:

```
const int n = 80;
char imie[n] = {'\0'};
char imie[m] = "";
```

Reprezentacja wewnętrzna łańcucha pustego:



Ustawianie łańcucha pustego po deklaracji:

```
imie[0] = '\0';
```

Przetwarzanie tablic polega zwykle na *przemaszerowaniu* zmienna indeksową po tablicy, dopóki nie ma końca napisu, oznaczanego znakiem '\0':

```
const int n = 80;
int i;
char str[n];

for(i = 0; str[i] != '\0'; i++)
  \\ operacje na każdym znaku str[i]
```

Wyprowadzanie zawartości napisu str do strumienia wyjściowego znak po znaku:

```
for(i = 0; str[i] != '\0'; i++)
  cout << str[i];
```

lub krócej:

```
for(i = 0; str[i] != '\0'; cout << s[i++ ]);
```

Do manipulowania tablicami znakowymi opracowano szereg funkcji bibliotecznych (plik nagłówkowy `string.h`). Wybrane funkcje do pracy z tablicami znakowymi:

- `atol()` – konwertuje wartość zapisaną w łańcuchu znaków do postaci liczby typu całkowitego,
- `memchr()` – szuka wystąpienia bajtu,
- `memcmp()` – porównuje bloki pamięci,
- `memcpy()` – kopiuje zawartość jednego bloku pamięci do drugiego,
- `memmove()` – kopiuje zawartość jednego bloku pamięci do drugiego (bloki mogą na siebie zachodzić),
- `memset()` – wypełnia pamięć bajtem,
- `strcat()` – scala dwa łańcuchy znaków w jeden,
- `strchr()` – szuka pierwszego wystąpienia znaku w łańcuchu znaków,

- `strcmp()` – porównuje dwa łańcuchy znaków,
- `strcoll()` – porównuje dwa łańcuchy znaków leksykograficznie,
- `strcpy()` – kopiuje łańcuch znaków do tablicy znaków,
- `strcspn()` – szuka pierwszego wystąpienia znaku (z puli znaków) w łańcuchu znaków,
- `stricmp()` – porównuje dwa łańcuchy znaków (ignoruje wielość liter),
- `stricoll()` – porównuje dwa łańcuchy znaków leksykograficznie (ignoruje wielkość liter),
- `strlen()` – oblicza długość łańcucha znaków,
- `strncat()` – scala dwa łańcuchy znaków w jeden; uwzględnia maksymalną liczbę znaków jaka może zostać dopisana,
- `strncmp()` – porównuje określoną liczbę znaków dwóch łańcuchów znaków,

- `strncpy()` – kopiuje określoną liczbę znaków łańcucha,
- `strpbrk()` – szuka pierwszego wystąpienia znaku (z puli znaków bez `'\0'`) w łańcuchu znaków,
- `strrchr()` – szuka ostatniego wystąpienia znaku w łańcuchu znaków,
- `strspn()` – zwraca indeks pierwszego znaku, który nie należy do puli znaków,
- `strstr()` – szuka pierwszego wystąpienia łańcucha znaków w innym łańcuchu znaków,
- `strtok()` – zastępuje (w łańcuchu znaków) pierwszy znaleziony znak znakiem terminalnym,
- `strtol()` – konwertuje wartość zapisaną w łańcuchu znaków w dowolnym systemie liczbowym do liczby całkowitej,
- `strtoul()` – konwertuje wartość zapisaną w łańcuchu znaków w dowolnym systemie liczbowym do liczby całkowitej bez znaku.

Klasa string

Klasa `string` nie jest wbudowanym typem języka, a zaprojektowaną klasą. Wchodzi w skład przestrzeni nazw `std`. Służy do reprezentacji ciągu znaków.

Posiada rozbudowany zestaw metod dostosowanych do działań na łańcuchach takich jak:

- wyszukiwanie znaków,
- zamiana znaków,
- wstawianie nowych znaków,
- usuwanie znaków.

Przykłady konstruktorów klasy `string`:

- pusty ciąg znaków:

```
string empty;
```

- tablica znaków lub literał:

```
const char *dogName = "dusty";

string dog(dogName);
string cat("garfield");
```

- obiekt złożony z wielokrotnych wystąpień znku:

```
string line(20, '-');
```

Przeciążone operatory klasy `string`:

- `[]` – odwołanie się do konkretnego znaku w stringu,
- `+=`, `+` – konkatenacja dwóch łańcuchów,
- `==` – porównanie łańcuchów,
- `=` – przypisanie,

```
const char *dogName = "dusty";

string catName("garfield");
string animals;

animals = dogName + " ";
animals += catName;

char firstLetter = catName[0];
```

Do dyspozycji mamy szereg użytecznych metod operujących na ciągach znaków, oto niektóre z nich:

- `size()` – zwraca liczbę znaków w ciągu,
- `lenght()` – zwraca liczbę znaków w ciągu,
- `clear()` – czyści zawartość łańcucha,
- `empty()` – sprawdza, czy ciąg znaków jest pusty,
- `resize()` – zmienia wielkość ciągu znaków,
- `max_size()` – zwraca maksymalną liczbę znaków, jaką można przechować,

- `append()` – dodaje do łańcucha kolejne znaki lub ciąg znaków,
- `push_back()` – dodaje znak na koniec ciągu,
- `assign()` – podmienia ciąg znaków na inny,
- `insert()` – wstawia ciąg znaków w określone miejsce,
- `replace()` – podmienia ciąg znaków począwszy od podanej pozycji,
- `swap()` – zamienia ze sobą dwa łańcuchy znaków,

- `pop_back()` – usuwa ostatni znak,
- `c_str()` – rzutuje typ `string` na `char *`,
- `copy()` – kopiuje podciąg znaków,
- `find()` – wyszukuje pozycję wystąpienia danego ciągu,
- `compare()` – porównuje dwa łańcuchy ze sobą,
- `getline()` – zapisanie całej linii (z białymi znakami) do łańcucha.

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

Argumenty wywołania

Większość programów (poleceń systemowych i innych aplikacji) można wywołać z **argumentami** (przełącznikami). Sposób podawania argumentów nie jest ściśle zdefiniowany. Zależy od programisty, możliwości jakie daje system operacyjny i język programowania.

```
ls -ld /etc/
```

```
egrep ^[^#] /etc/crontab
```

```
egrep usb\ 1-[0-9] boot.msg
```

```
apt install mc
```

```
echo "Hello World!"
```

```
find /etc -name "shad*" -type f -exec grep root: {} \;
```

Obsługa argumentów wywołania

Aby korzystanie z argumentów wywołania było możliwe, funkcja `main` programu powinna być zdefiniowana z nagłówkiem:

```
int main(int argc, char **argv)
```

lub

```
int main(int argc, char *argv[])
```

Pierwszy parametr, o zwyczajowej nazwie `argc`, jest liczbą argumentów podanych podczas wywołania programu.

Pierwszym argumentem, o numerze 0, jest zawsze nazwa wywoływanego programu. Tak więc argc jest zawsze co najmniej jeden.

Zmienna argv jest tablicą napisów zawierających kolejne argumenty wywołania: argv[0], argv[1],..., argv[argc - 1].

Indeksowanie rozpoczyna się od zera, dlatego ostatnią wartością indeksu jest argc - 1.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
  cout << "Liczba argumentow: " << argc << endl << endl;

  for(int i = 0; i < argc; i++)
    cout << "Argument " << i << ": " << argv[i] << endl;

  return EXIT_SUCCESS;
}
```

```
foo.exe -a bb "c c c" 1 2 33 3.141 2,718
```

```
Liczba argumentow: 9

Argument 0: D:\foo.exe
Argument 1: -a
Argument 2: bb
Argument 3: c c c
Argument 4: 1
Argument 5: 2
Argument 6: 33
Argument 7: 3.141
Argument 8: 2,718
```

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

Interfejs użytkownika

Interfejs użytkownika (*user interface*, *UI*) – przestrzeń, w której następuje interakcja człowieka z maszyną.

W technice interfejs użytkownika to część urządzenia lub oprogramowania odpowiedzialna za interakcję z użytkownikiem.

Człowiek nie jest zdolny do bezpośredniej komunikacji z maszynami. Aby było to możliwe, są one wyposażone w urządzenia wejścia-wyjścia tworzące interfejs użytkownika.

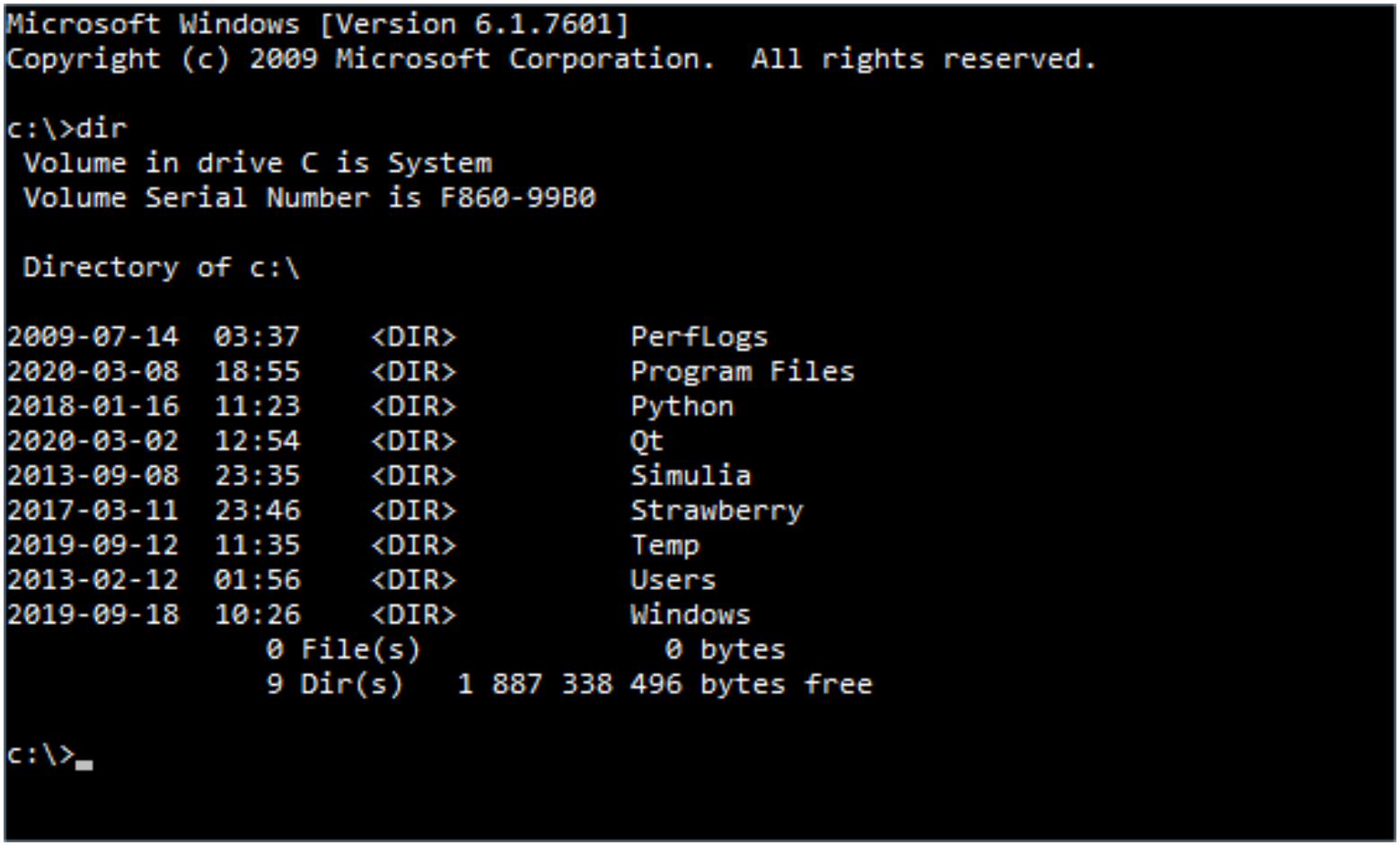
W informatyce najczęściej jako interfejs użytkownika rozpatruje się część oprogramowania zajmującą się obsługą urządzeń wejścia-wyjścia przeznaczonych dla interakcji z użytkownikiem.

W komputerach zwykle za obsługę większości funkcji interfejsu użytkownika odpowiada system operacyjny, który narzuca standaryzację wyglądu różnych aplikacji.

Użytkownicy postrzegają oprogramowanie wyłącznie przez interfejs użytkownika.

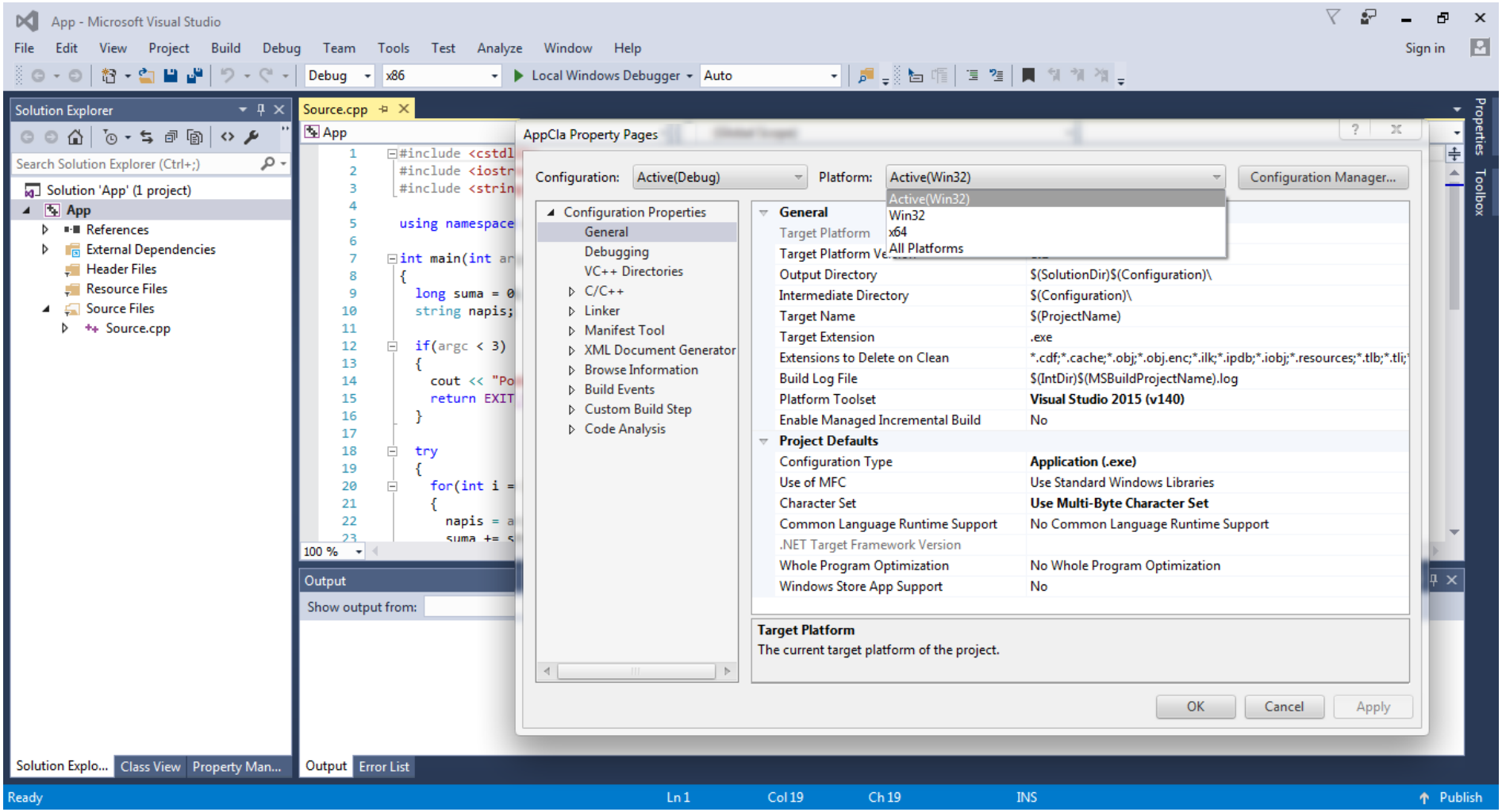
Do najczęściej stosowanych interfejsów użytkownika zalicza się:

- **wiersz poleceń** (*command-line interface*, *CLI*) – urządzenie wejściowe to klawiatura, a wyjściowe to drukarka lub wyświetlacz w trybie znakowym,

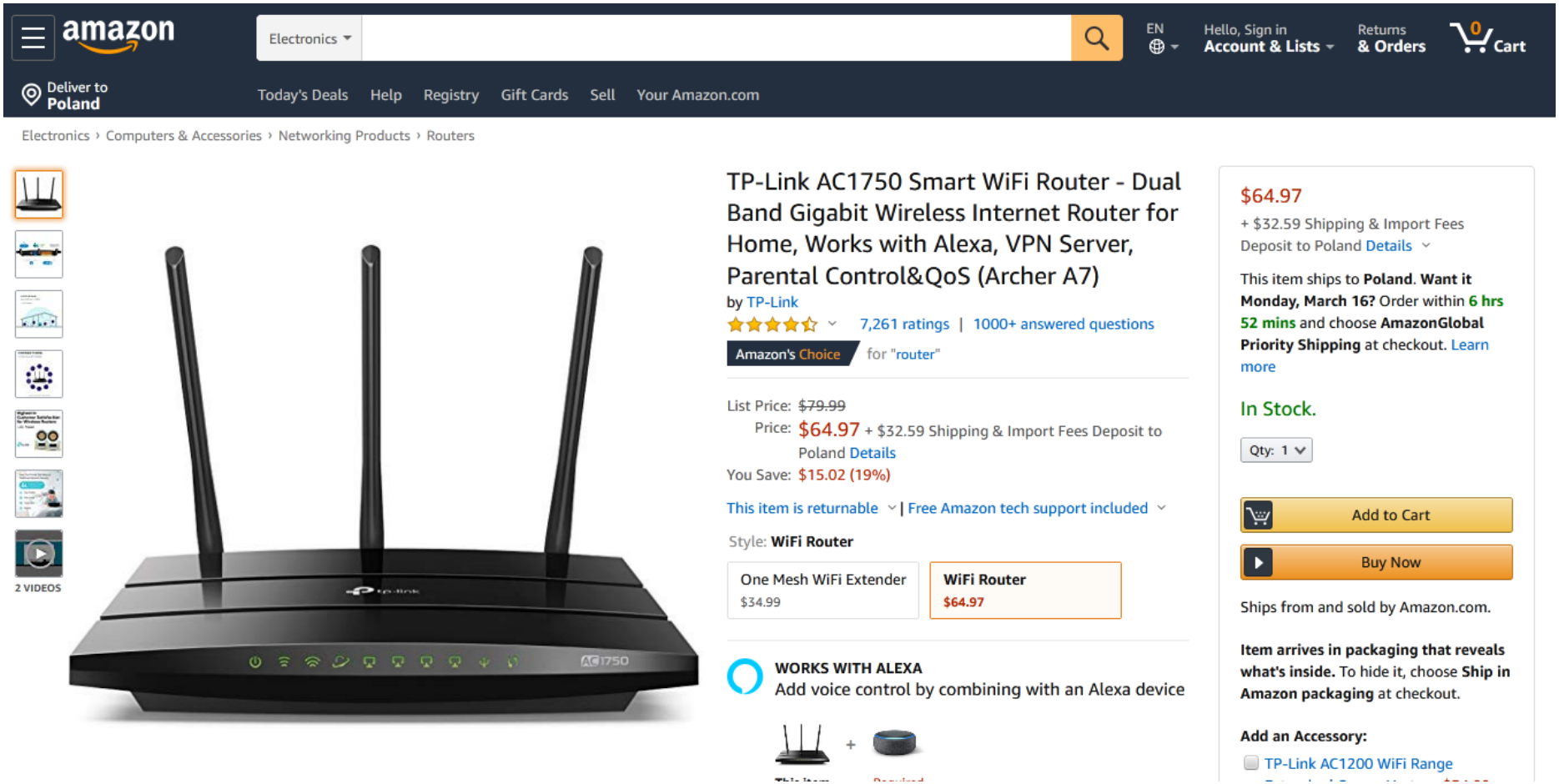


- **interfejs tekstowy** (*text user interface*, *TUI*) – urządzenie wejściowe to klawiatura lub myszka, a wyjściowe to wyświetlacz w trybie znakowym,

- **interfejs graficzny** (*graphic user interface*, *GUI*) – wejście to urządzenie wskazujące (zwykle myszka), a wyjściowe to wyświetlacz graficzny,



- **interfejs strony internetowej** (*web user interface*, *WUI*) – wejście i wyjście jest realizowane poprzez stronę internetową wyświetlaną w przeglądarce,



- **interfejs głosowy** (*voice user interface*, *VUI*) – urządzenie wejściowe to mikrofon, a wyjściowe to głośniki,
- **interfejs gestowy** – urządzenie wejściowe to ciało ludzkie lub specjalny kontroler, a wyjściowe to wyświetlacz graficzny (przykładem jest Kinect dla konsoli Xbox 360).

Graficzny interfejs użytkownika

Graficzny interfejs użytkownika, interfejs graficzny, środowisko graficzne, GUI – określenie sposobu prezentacji informacji przez komputer oraz interakcji z użytkownikiem, polegającego na rysowaniu i obsługiwaniu widżetów (*widget*).

Interfejs graficzny został wymyślony przez przedsiębiorstwo Xerox w latach 70. XX wieku w laboratorium PARC, a następnie później wykorzystywany i udoskonalany przez inne przedsiębiorstwa.

Środowisko graficzne jest grupą wzajemnie współpracujących programów, zapewniającą możliwość wykonywania podstawowych operacji na komputerze.

Do tych operacji należą: uruchamianie programów, poruszanie się po katalogach, zmiana ustawień w trybie graficznym, najczęściej okienkowym.

Zapewnia alternatywny dla wiersza poleceń (konsoli tekstowej) sposób pracy na komputerze.

Najważniejszym elementem graficznego interfejsu jest **okno programu** (lub kilka takich okien) prezentowane na **pulpicie**.

Bezpośrednio na pulpicie lub wewnątrz okna są rozmieszczone (najczęściej w formie graficznych ikon lub menu) elementy interakcyjne, zwane **widżetami** (lub **kontrolkami**, nawiązując do pulpitów sterowniczych).

Użytkownik komunikuje się z aplikacją pośrednio przez te widżety, najczęściej za pomocą **myszy** i **klawiatury**.

Mysz (czasem **trackball**) jest odpowiedzialna za przesuwanie kursora, wskazującego odpowiednią pozycję na ekranie, a naciskanie przycisków jest związane z obszarem, w którym zawiera się aktualna pozycja kursora.

Klawiatura jest związana z pojęciem **skupienia**. Jest to stan, który może posiadać w jednym momencie dokładnie jeden widżet w całym systemie okienkowym.

Jeśli użytkownik naciska klawisze, informacja o tym przekazywana jest do tego widżetu, który aktualnie **posiada skupienie**.

Składowe interfejsu użytkownika:

- **mechanizm wejściowy** (*input*) – realizuje wprowadzanie danych przez użytkownika do systemu (listy wyboru, pola tekstowe, formularze),
- **mechanizm wyjściowy** (*output*) – realizuje dostarczanie danych przez system dla użytkownika (komunikaty, okna dialogowe, raporty, strony web),
- **mechanizm nawigacji** – realizuje sterowanie systemem przez użytkownika (menu, przyciski).

Proces wytwarzania oprogramowania

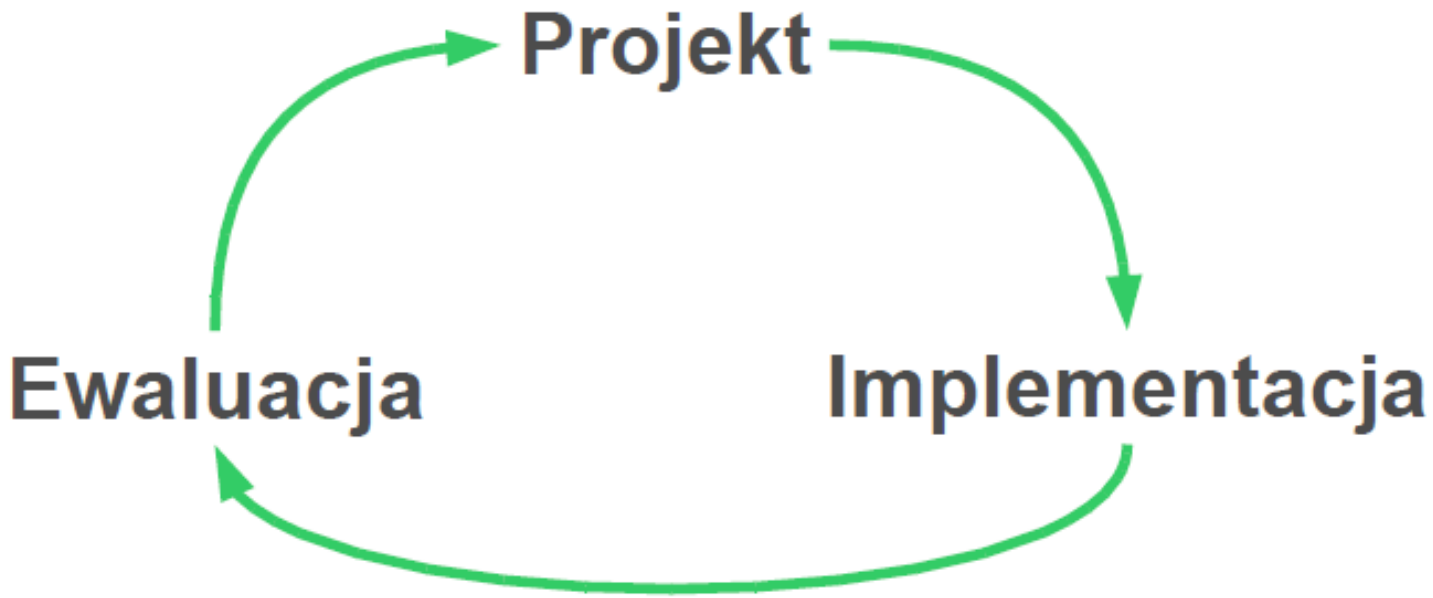
Czyli kilka haseł z inżynierii oprogramowania z perspektywy tworzenia interfejsu użytkownika.

Inżynieria oprogramowania – dziedzina inżynierii systemów zajmująca się wszelkimi aspektami produkcji oprogramowania: od analizy i określenia wymagań, przez projektowanie i wdrożenie, aż do ewalucji gotowego oprogramowania.

Podczas gdy informatyka zajmuje się teoretycznymi aspektami produkcji oprogramowania, inżynieria oprogramowania koncentruje się na stronie praktycznej.

„Tradycyjny” sposób wytwarzania oprogramowania

Proces wytwarzania oprogramowania



Projekt

- Analiza zadań:
  - poznaj użytkownika – co chce (widzieć) użytkownik?
  - unikaj rzutowania własnych upodobań na użytkowników.
- Korzystaj z zaleceń projektowych:
  - wykorzystaj przyzwyczajenia użytkownika,
  - zapobieganie popełniania błędów,
  - zalecenia często mają charakter heurystyczny i mogą się wykluczać.

Implementacja

- Stosuj prototypy:
  - proste implementacje,
  - zgrubne: papier, szkic elektroniczny,
  - dokładne: HTML, UI Builders.
- Wybierz odpowiedni model tworzenia UI:
  - model wejścia/wyjścia,
  - pakiety narzędziowe (*toolkits*),
  - programy do budowania interfejsu.

Ewaulacja

- Testowanie prototypów:
  - ocena eksperta – wiedza heurystyczna oparta na doświadczeniu,
  - ocena przewidująca – testowanie za pomocą symulowanego użytownika (kolega z zespołu),
  - ocena empiryczna – analiza jak radzi sobie z prototypem rzeczywisty użytkownik.

Podstawowe zasady projektowania interfejsu GUI

Wygląd – interfejs powinien być podzielony na różne obszary przeznaczone do różnych celów:

- każdy obszar powinien mieć jasno wytyczone granice,
- każdy obszar powinien mieć jasno określone przeznaczenie,
- każdy obszar powinien zawierać tylko te informacje, które są potrzebne do realizacji określonego przeznaczenia,
- obszary informacyjne powinny być uszeregowane w kolejności przetwarzania tej informacji przez użytkownika (z góry w dół, od lewej do prawej).

Uświadamianie zawartości – interfejs powinien uświadamiać użytkownika, w którym miejscu się znajduje i co oznaczają prezentowane informacje:

- wszystkie okna i raporty muszą mieć tytuły jednoznacznie identyfikujące ich zawartość,
- menu musi pokazywać, w którym miejscu jest użytkownik (i jak się tu dostał),
- przyciski powinny mieć napisy identyfikujące ich funkcje; jeśli napisy te nie są pokazywane cały czas na ekranie, to powinny być pokazywane przy najechaniu na przycisk wskaźnikiem myszy,
- przyciski odpowiedzi na oknach komunikatów powinny być łatwo interpretowane w kontekście treści komunikatu,

- forma informacji na sąsiadujących obszarach powinna być różna (np. tekst-grafika, różne czcionki); rozróżnianie kolorem może nie być wystarczające,
- jeśli informacje na sąsiadujących obszarach są podobne w formie, to muszą być oddzielone dodatkowym elementem (np. linią),
- każde pole edycji musi mieć etykietę jednoznacznie identyfikującą zawartość pola,
- pola edycji, których format wewnętrzny nie jest oczywisty (np. pola z datą), muszą mieć dodatkowe oznaczenie formatu wprowadzanych danych,
- raporty powinny mieć datę sporządzenia.

Estetyka – interfejs powinien zapewniać równowagę pomiędzy ilością prezentowanej informacji a jej atrakcyjnością wizualną:

- interfejs użytkownika powinien być zarówno funkcjonalny jak i przyjemny dla oka,
- ilość wolnego miejsca pomiędzy elementami interfejsu powinna być dostosowana do wymagań użytkownika (więcej dla początkujących, mniej dla zaawansowanych),
- należy unikać tworzenia formularzy lub raportów dużych, zawierających ponad wiele pól danych,
- formularz lub raport powinien zawierać tylko informacje, które mogą być jednorazowo przetworzone przez człowieka,

- tekst główny powinien być prezentowany czytelną czcionką (8-10 punktów?); na formularzach powinna być używana czcionka bezszeryfowa, na raportach czcionka szeryfowa,
- należy unikać stosowania więcej niż dwóch różnych czcionek; stanowczo trzeba unikać czcionek ozdobnych, lecz trudnych do czytania,
- stosowane kolory powinny być stonowane (kontrastowe zwracają uwagę, lecz są męczące dla oka,
- kolor nie może być jedynym wyróżnikiem informacji; interfejs użytkownika nie może ukrywać informacji przed osobami cierpiącymi na daltonizm.

Doświadczenie użytkownika – interfejs powinien uwzględniać zarówno łatwość nauki dla początkujących jak i łatwość użycia dla doświadczonych użytkowników:

- interfejs użytkownika powinien być łatwy do nauczenia się posługiwania się nim przez początkujących użytkowników,
- interfejs użytkownika powinien ułatwiać i przyspieszać wykonywanie działań przez zaawansowanych użytkowników,
- menu powinno składać się nie więcej niż z trzech poziomów w przypadku menu głównego i nie więcej niż z dwóch poziomów w przypadku menu kontekstowych,
- menu powinno prezentować wszystkie dostępne funkcje aplikacji, tzn. nie powinno być takiej funkcji, do której nie można by się było dostać z menu,

- menu na każdym poziomie powinno zawierać nie więcej niż kilka pozycji; w przypadku bardziej rozbudowanego menu wskazane jest, by pozycje menu były logicznie pogrupowane oraz by częściej używane funkcje były w pewien sposób wyróżnione,
- częściej wykorzystywane funkcje powinny być dostępne bezpośrednio poprzez przyciski narzędziowe; przyciski narzędziowe powinny mieć obrazek kojarzący się z wykonywaną funkcją oraz nazwę funkcji; jeśli nazwy funkcji na przyciskach nie mogą być pokazane, to powinny być pokazywane przy najechaniu myszą na przycisk,
- przyciski powinny być logicznie pogrupowane na paskach narzędziowych,
- w przypadku aplikacji realizującej liczne funkcje wskazane jest, aby umożliwiała ona konfigurację pasków narzędziowych, w tym umieszczanie na paskach przycisków wiodących do wszystkich funkcji aplikacji, również tych rzadziej wykorzystywanych,

- wskazane jest, aby bardziej złożona aplikacja prezentowała swoje możliwości za pomocą podpowiedzi; jeśli podpowiedzi te zajmują istotnie dużo miejsca na ekranie, to aplikacja powinna umożliwić wyłączenie podpowiedzi i włączenie ich ponownie na żądanie,
- aplikacja powinna umożliwiać szybki dostęp do funkcji za pomocą skrótów klawiszowych; w przypadku bardziej złożonej aplikacji wskazane jest zapewnienie możliwości definiowania własnych skrótów klawiszowych;
- aplikacja powinna mieć system pomocy ekranowej wyjaśniającej podstawowe mechanizmy zastosowane w aplikacji i wyjaśniającej sposób wykorzystania tych mechanizmów.

Spójność – interfejs powinien być spójny dla ułatwienia użytkownikowi przewidywania skutków podejmowanych przez niego działań:

- interfejs użytkownika powinien być spójny dla zapewnienia przewidywalności podejmowanych działań przez użytkownika,
- wszystkie formularze i raporty w aplikacji powinny być zaprojektowane w jednolity sposób, tzn. z użyciem jednolitego aparatu pojęciowego (terminologii) i z zastosowaniem jednolitej formy (takiego samego układu, czcionek i kolorów) oraz sposobu nawigacji,
- interfejs użytkownika aplikacji powinien być spójny z innymi aplikacjami z tej samej dziedziny wykorzystywanymi w systemie operacyjnym.

Minimalizacja wysiłku – interfejs powinien ułatwiać działania użytkownika, tak by ilość kroków wiodących do osiągnięcia celu była jak najmniejsza:

- interfejs użytkownika powinien ułatwiać użytkownikowi wykorzystanie funkcji aplikacji tak, aby wysiłek użytkownika był jak najmniejszy,
- zaleca się, aby ilość kliknięć myszą poprzez menu lub przyciski narzędziowe do każdej funkcji nie przekraczała trzech,
- w przypadku funkcji wielokrotnie wykorzystywanych zaleca się zastosowanie mechanizmu powtarzania funkcji lub grupowania przedmiotów działania funkcji,
- w przypadku bardziej złożonych aplikacji zaleca się zastosowanie mechanizmu umożliwiającego łączenie wielu różnych funkcji w jedną.

Poprawnie zaprojektowany interfejsu WUI

- Dobrze dobrane wygląd i zachowanie (*look and feel*).
- Przemyślany schemat układu strony (*layout*) – identyfikator, wyszukiwarka, menu główne, manu narzędziowe.
- Odpowiednio pogrupowane elementy interfejsu i punkty startowe (co na stronie można znaleźć i zrobić).
- Czytelne komunikaty.
- Krótkie teksty odnośników i opcji.
- Wyróżnione linki i elementy do klikania.
- Menu nawigacyjne (jeden typ użytkowników) i wyszukiwarka (drugi typ użytkowników).
- Informacja, gdzie jesteśmy w strukturze serwisu.
- J. Nielsen podaje, że aż 30% wszystkich kliknięć to kliknięcie przycisku Wstecz – unikanie ramek i nawigacji opartej na Flash.

Złe praktyki w projektowaniu interfejsu WUI

Na tej stronie znajduje się formularz zamawiania podróży.

Imię (obowiązkowe)

Nazwisko (obowiązkowe)

Wycieczka (obowiązkowe)

Sugerowana cena:) (obowiązkowe)

Nr. tel. stacjonarnego (obowiązkowe, pełen)

Dodatkowe życzenia

Preferowany sposób transportu Samolot Autobus Prom

Miejsca dla niepalących

Jedzenie wegetariańskie

Ilość osób

Waluta PLN USD EURO

Posiłki Śniadanie Obiad Kolacja

Podsumowanie cen posiłków 0 PLN

Preferowane warianty wycieczek Krajoznawcze (Wciśnij CTRL, by zaznaczyć więcej niż jedną opcję.)

Zgadzam się na przetwarzanie danych osobowych. (obowiązkowe)

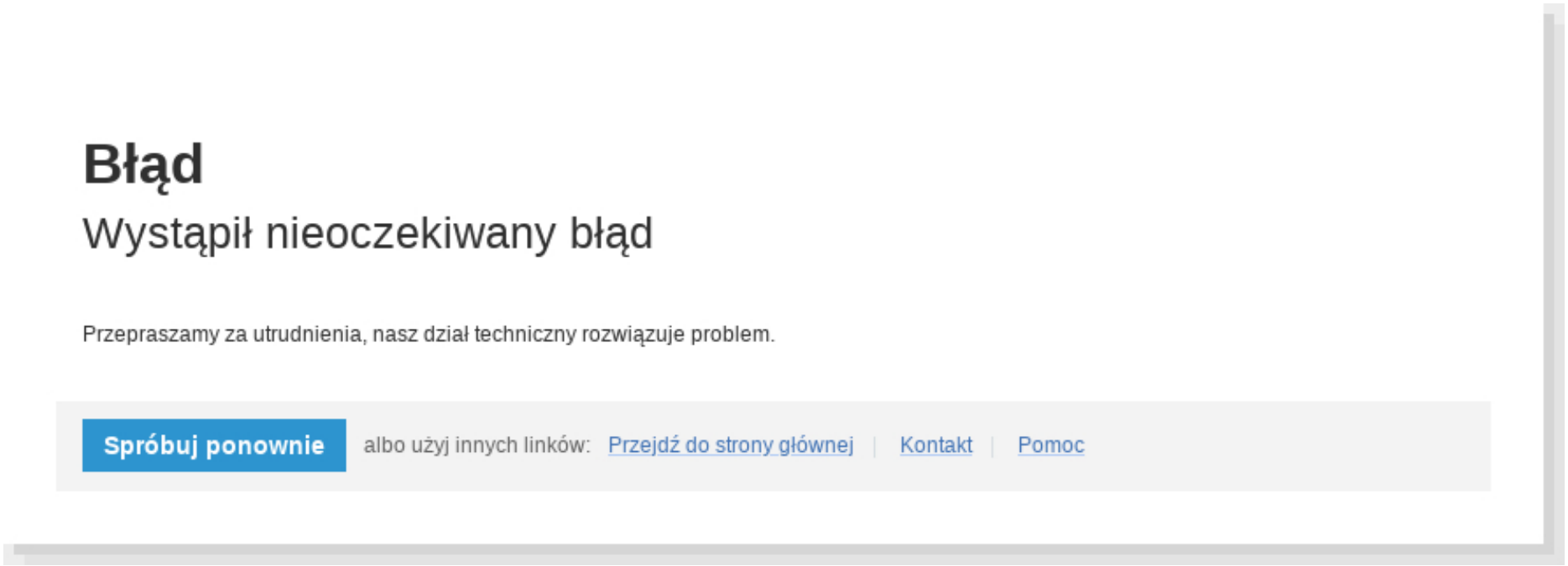
Wyślij Anuluj


Patimex
producent węgla drzewnego
O FIRMIE
OFERTA
GALERIA
DOJAZD
KONTAKT
HOLZKOHLE
GRILL
ca 2,5 kg
WĘGIEL
Z PIEKŁA RODEM
O FIRMIE
OFERTA
GALERIA
DOJAZD
KONTAKT

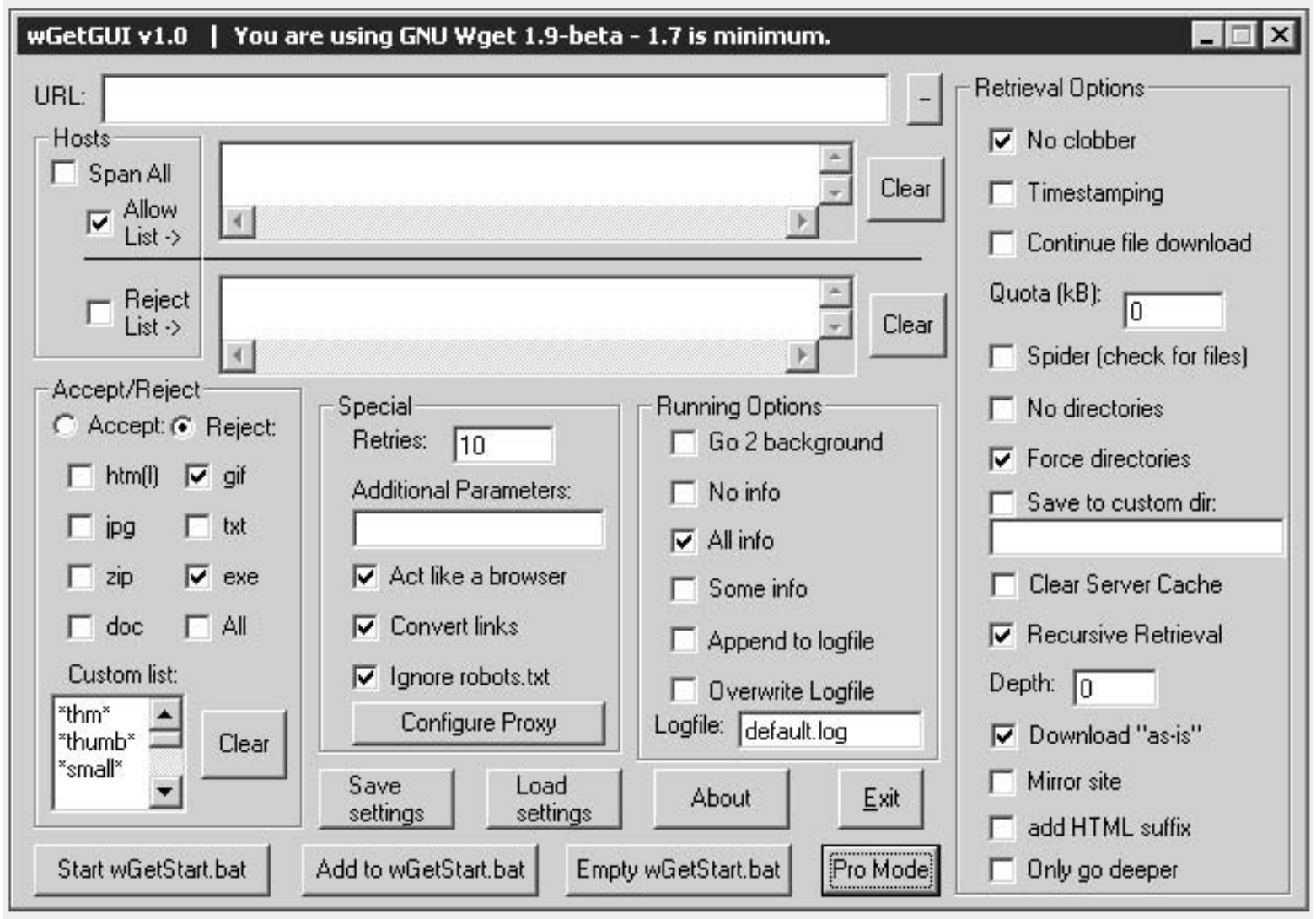

HURTOWNIA
KONTAKT
tel. 15 / 822-35-50 ul. Sandomierska 6
fax 15 / 822-39-00 39-400 Tarnobrzeg
kontakt256@wp.pl
Strona główna
O firmie
Siedziba
Oferta
Spot reklamowy
Kontakt
>> znajdź nas na FB i polub - będziesz na bieżąco <<
MATTEL
LEGO
AXEL
STARPAK
sticker BOO
STAEDTLER
Angelina Ballerina
skelanimals
BEN 10 ALIEN FORCE
hipo
SNOOPY
trodat
MERF
Trefl
AXIOM
BAKUGAN
NEW VESTROIA
TRANSFORMERS
BALLOGRAF
Discovery CHANNEL
Hasbro
Fisher-Price
artyk
SAFARI
e-EDU
My little Pony
MILITARY
REALTOY
Natalia
Pentel
PUCCA
Littlest Pet Shop
Mega Creative
MASKOTKI PLUSZOWE
DEEF
SCOOBY-DOO!
Barbie
HOT WHEELS
LittleBIG Planet
THOMAS & FRIENDS
Miami Ink
(C) 2010 - 2020 Hurtownia Kontakt | Polityka Prywatności | Polityka Cookies

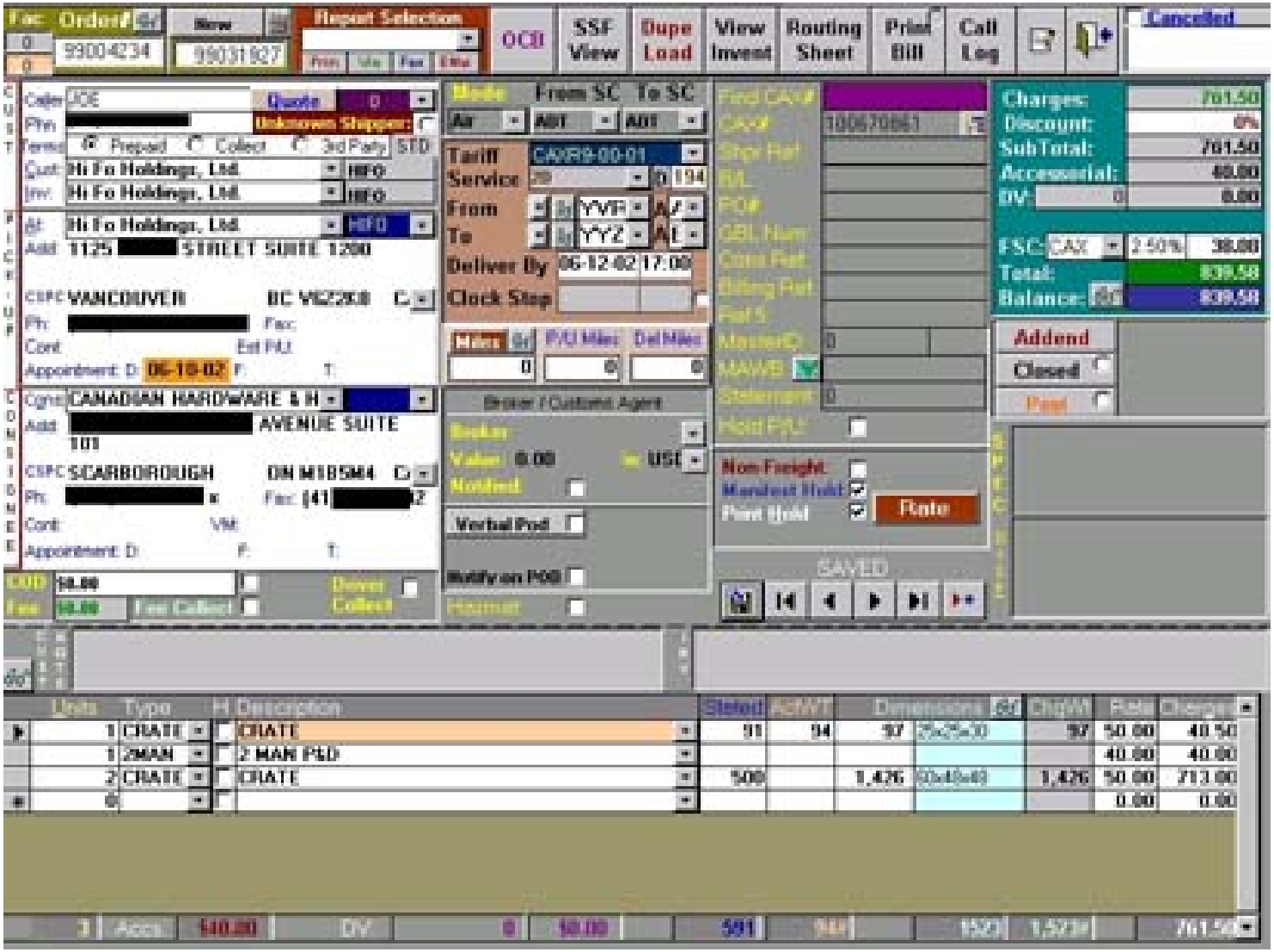

MULTAX
CZYSTE GAZY
STRONA GŁÓWNA
O FIRMIE
CZYSTE GAZY
MIESZANINY
USŁUGI
PLAN DOJAZDU
Nawigacja: Strona główna /
WITAMY NA STRONIE MULTAX S.C.
Firma nasza zajmuje się produkcją czystych gazów i ich mieszanin oraz wykonywaniem instalacji laboratoryjnych, świadczy usługi w zakresie transportu materiałów niebezpiecznych a także udziela wszelkich porad, dotyczących techniki pracy z gazami. Od początku działalności towarzyszą nam koty, których zdjęcia ilustrują stronę.
KONTAKT:
multax@wp.pl
Tel. (22) 7229291
Tel/fax. (22) 7229895
ADRES:
Zielonki-Parcela ul. Zachodnia 41
05-082 Stare Babice
Woj. Mazowieckie
Pracujemy od poniedziałku do piątku w godzinach 8-16
STRONA GŁÓWNA O FIRMIE CZYSTE GAZY MIESZANINY USŁUGI PLAN DOJAZDU KONTAKT

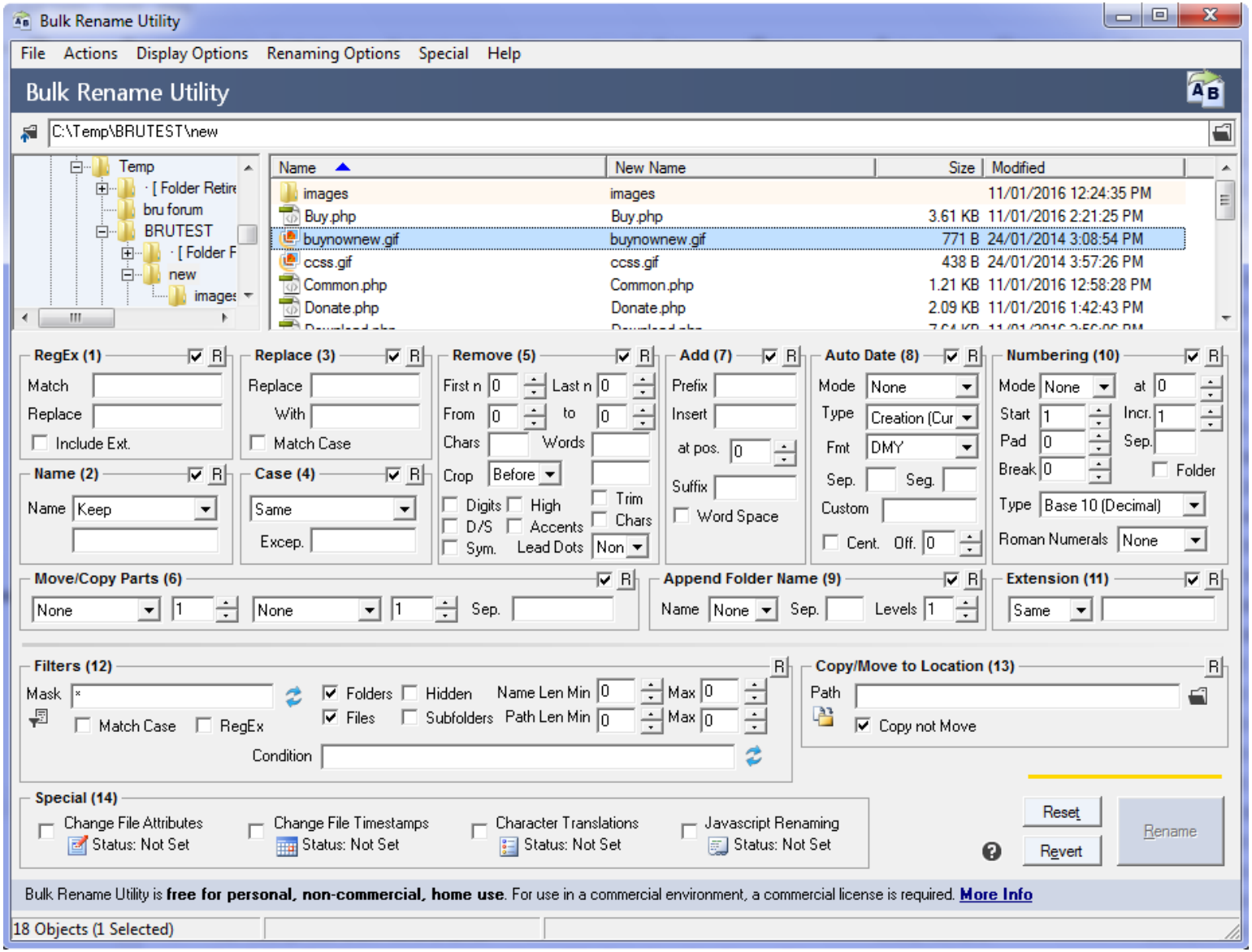

Wystąpił błąd
Coś jest zepsute. To zapewne chwilowo, więc spróbuj ponownie za kilka minut.
Jeżeli ten problem występuje już od dłuższego czasu powiadom nas o tym na adres ng@burdamedia.pl.
Postaramy się ten błąd naprawić jak najszybciej. Przepraszamy za niedogodności.

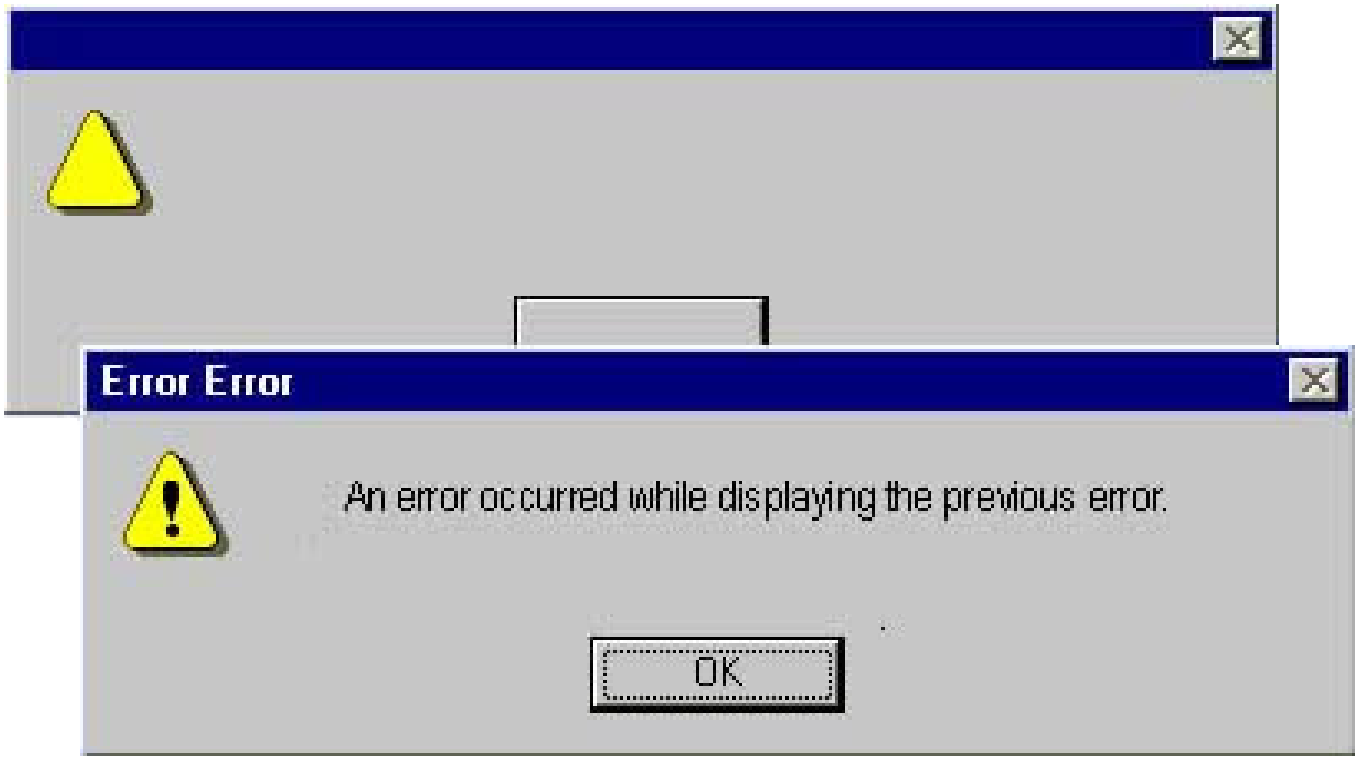

Błąd
Wystąpił nieoczekiwany błąd
Przepraszamy za utrudnienia, nasz dział techniczny rozwiązuje problem.
Spróbuj ponownie
albo użyj innych linków:
Przejdź do strony głównej
Kontakt
Pomoc

Złe praktyki w projektowaniu interfejsu GUI

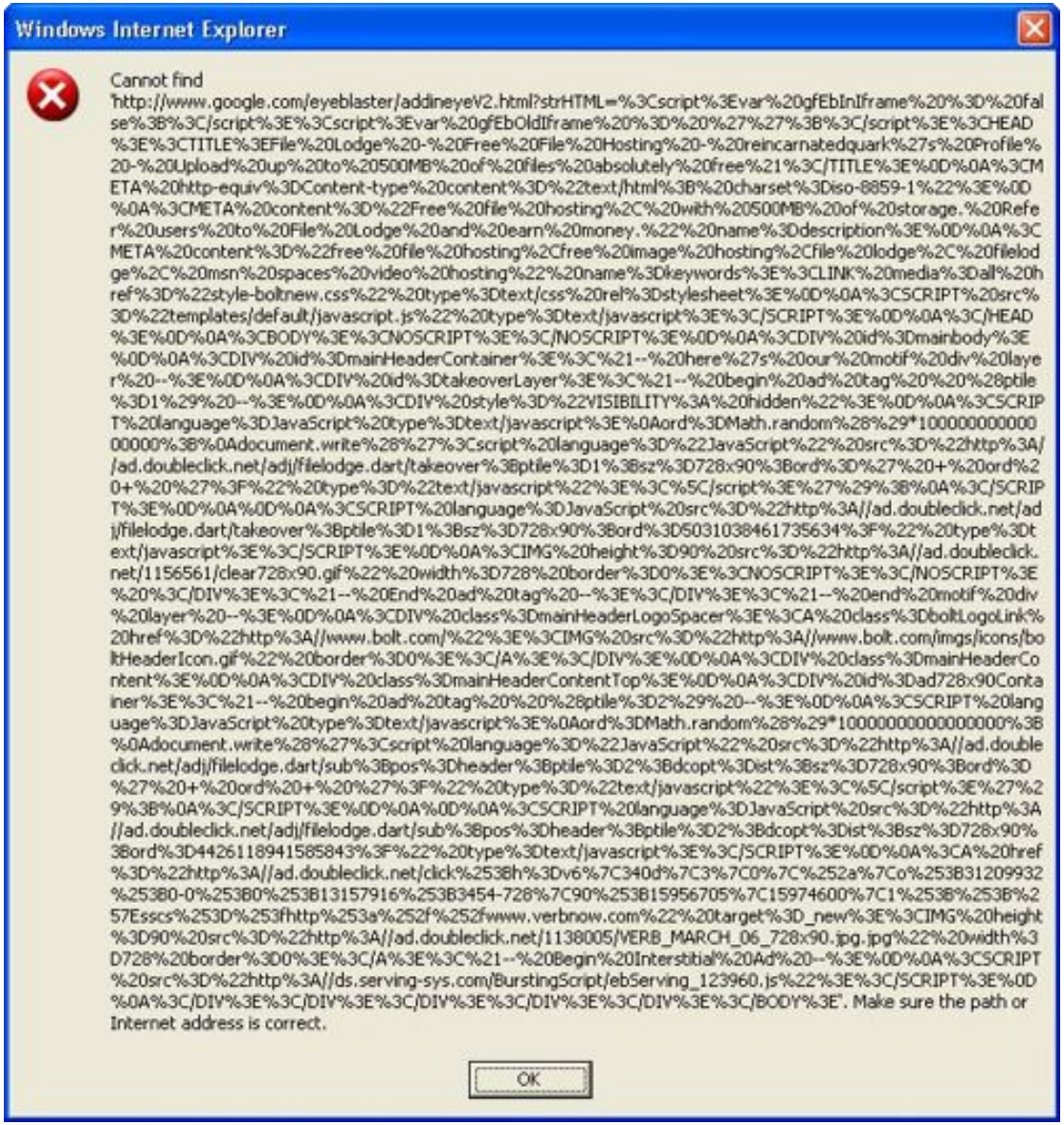



Ordered
New
Report Selection
OCD
SSF View
Dupe Load
View Invent
Routing Sheet
Print Bill
Call Log
Cancelled
Quote
Unknown Shipper
Prepaid
Collect
3rd Party
Hi Fo Holdings, Ltd.
VANCOUVER
BC
CANADIAN HARDWARE & H
SCARBOROUGH
ON
From SC
To SC
Air
Tariff
Service
From
To
Deliver By
Clock Stop
Broker / Customs Agent
Charges:
Discount:
SubTotal:
Accessorial:
DV:
FSC
Total:
Balance:
Addend
Closed
Post
Non Freight
Manifest Hold
Print Hold
Rate
SAVED
Units
Type
Description
Stated
ActWT
Dimensions
ChgWT
Rate
Charge
1 CRATE
CRATE
1 2MAN
2 MAN P&D
2 CRATE
CRATE


Bulk Rename Utility
File Actions Display Options Renaming Options Special Help
Bulk Rename Utility
C:\Temp\BRUTEST\new
Temp
bru forum
BRUTEST
new
images
Name
New Name
Size
Modified
images
images
11/01/2016 12:24:35 PM
Buy.php
Buy.php
3.61 KB
11/01/2016 2:21:25 PM
buynownew.gif
buynownew.gif
771 B
24/01/2014 3:08:54 PM
ccss.gif
ccss.gif
438 B
24/01/2014 3:57:26 PM
Common.php
Common.php
1.21 KB
11/01/2016 12:58:28 PM
Donate.php
Donate.php
2.09 KB
11/01/2016 1:42:43 PM
RegEx (1)
Match
Replace
Include Ext.
Replace (3)
Replace
With
Match Case
Remove (5)
First n
Last n
From
to
Chars
Words
Crop
Before
Digits
High
D/S
Accents
Sym.
Lead Dots
Trim
Chars
Add (7)
Prefix
Insert
at pos.
Suffix
Word Space
Auto Date (8)
Mode
None
Type
Creation (Cur
Fmt
DMY
Sep.
Seg.
Custom
Cent.
Off.
Numbering (10)
Mode
None
at
Start
Incr.
Pad
Sep.
Break
Folder
Type
Base 10 (Decimal)
Roman Numerals
None
Name (2)
Name
Keep
Case (4)
Same
Excep.
Move/Copy Parts (6)
None
Sep.
Append Folder Name (9)
Name
None
Sep.
Levels
Extension (11)
Same
Filters (12)
Mask
Match Case
RegEx
Folders
Files
Hidden
Subfolders
Name Len Min
Path Len Min
Max
Condition
Copy/Move to Location (13)
Path
Copy not Move
Special (14)
Change File Attributes
Status: Not Set
Change File Timestamps
Status: Not Set
Character Translations
Status: Not Set
Javascript Renaming
Status: Not Set
Reset
Revert
Rename
Bulk Rename Utility is free for personal, non-commercial, home use. For use in a commercial environment, a commercial license is required. More Info
18 Objects (1 Selected)


Error Error
An error occurred while displaying the previous error.
OK

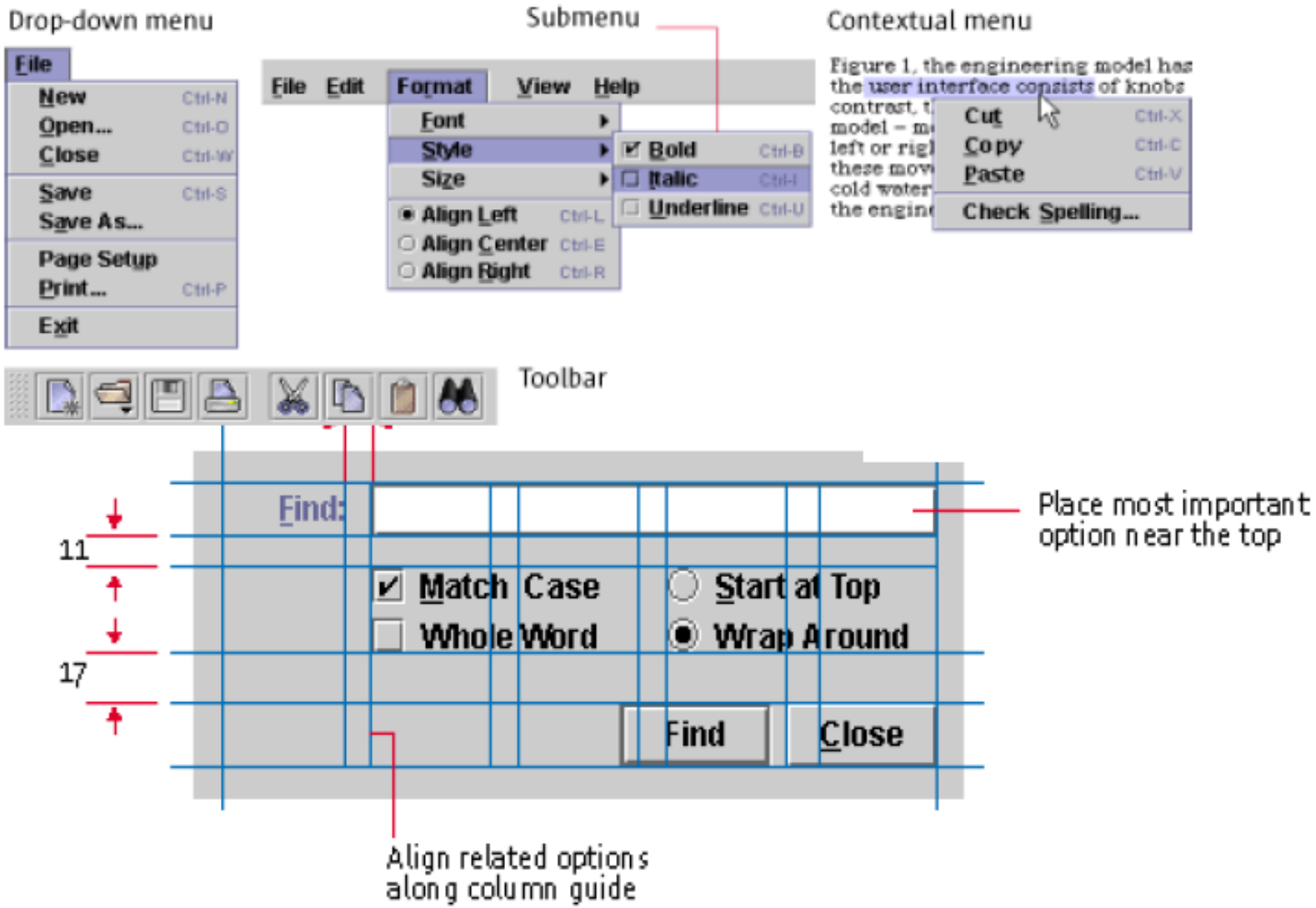

Microsoft Access
[Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257) [Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257) [Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257) [Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257) [Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257) [Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257) [Microsoft][ODBC SQL Server Driver][SQL Server]Implicit conversion from data type datetime to float is not allowed. Use the CONVERT function to run this query. (#257)
OK

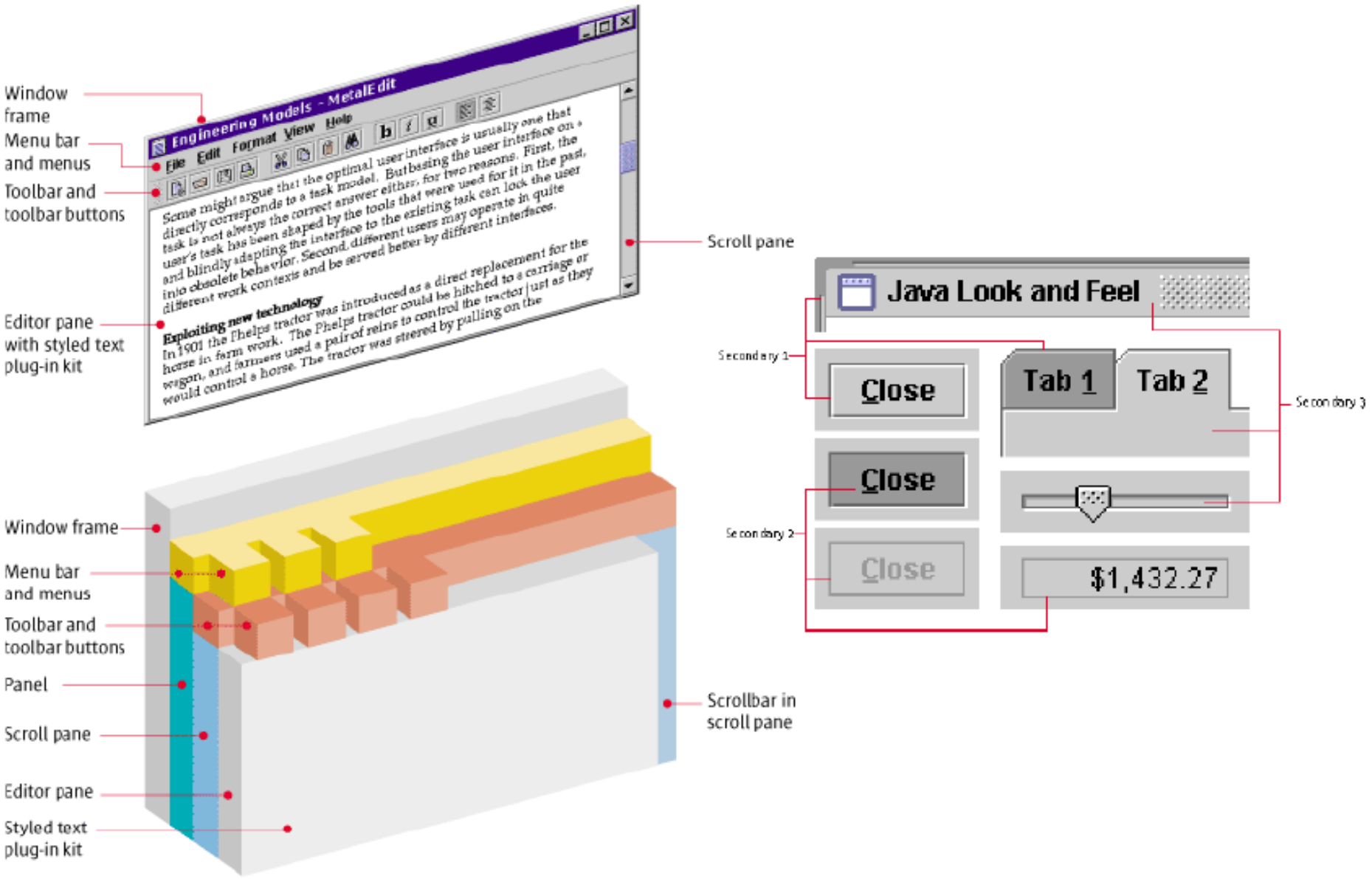

Windows Internet Explorer
Cannot find
'http://www.google.com/eyeblaster/addineyeV2.html?strHTML=%3Cscript%3Evar%20gfEbInIframe%20%3D%20false%3B%3C/script%3E%3Cscript%3Evar%20gfEbOldIframe%20%3D%20%27%27%3B%3C/script%3E%3CHEAD%3E%3CTITLE%3EFile%20Lodge%20-%20Free%20File%20Hosting%20-%20reincarnatedquark%27s%20Profile%20-%20Upload%20up%20to%20500MB%20of%20files%20absolutely%20free%21%3C/TITLE%3E%0D%0A%3CMETA%20http-equiv%3DContent-type%20content%3D%22text/html%3B%20charset%3Diso-8859-1%22%3E%0D%0A%3CMETA%20content%3D%22Free%20file%20hosting%2C%20with%20500MB%20of%20storage.%20Refer%20users%20to%20File%20Lodge%20and%20earn%20money.%22%20name%3Ddescription%3E%0D%0A%3CMETA%20content%3D%22free%20file%20hosting%2Cfree%20image%20hosting%2Cfile%20lodge%2C%20filelodge%2C%20msn%20spaces%20video%20hosting%22%20name%3Dkeywords%3E%3CLINK%20media%3Dall%20href%3D%22style-boltnew.css%22%20type%3Dtext/css%20rel%3Dstylesheet%3E%0D%0A%3CSCRIPT%20src%3D%22templates/default/javascript.js%22%20type%3Dtext/javascript%3E%3C/SCRIPT%3E%0D%0A%3C/HEAD%3E%0D%0A%3CBODY%3E%3CNOSCRIPT%3E%3C/NOSCRIPT%3E%0D%0A%3CDIV%20id%3Dmainbody%3E%0D%0A%3CDIV%20id%3DmainHeaderContainer%3E%3C%21--%20here%27s%20our%20motif%20div%20layer%20--%3E%0D%0A%3CDIV%20id%3DtakeoverLayer%3E%3C%21--%20begin%20ad%20tag%20%20%28ptile%3D1%29%20--%3E%0D%0A%3CDIV%20style%3D%22VISIBILITY%3A%20hidden%22%3E%0D%0A%3CSCRIPT%20language%3DJavaScript%20type%3Dtext/javascript%3E%0Aord%3DMath.random%28%29*10000000000000000%3B%0Adocument.write%28%27%3Cscript%20language%3D%22JavaScript%22%20src%3D%22http%3A//ad.doubleclick.net/adj/filelodge.dart/takeover%3Bptile%3D1%3Bsz%3D728x90%3Bord%3D%27%20+%20ord%20+%20%27%3F%22%20type%3D%22text/javascript%22%3E%3C%5C/script%3E%27%29%3B%0A%3C/SCRIPT%3E%0D%0A%0D%0A%3CSCRIPT%20language%3DJavaScript%20src%3D%22http%3A//ad.doubleclick.net/adj/filelodge.dart/takeover%3Bptile%3D1%3Bsz%3D728x90%3Bord%3D5031038461735634%3F%22%20type%3Dtext/javascript%3E%3C/SCRIPT%3E%0D%0A%3CIMG%20height%3D90%20src%3D%22http%3A//ad.doubleclick.net/1156561/clear728x90.gif%22%20width%3D728%20border%3D0%3E%3CNOSCRIPT%3E%3C/NOSCRIPT%3E%20%3C/DIV%3E%3C%21--%20End%20ad%20tag%20--%3E%3C/DIV%3E%3C%21--%20end%20motif%20div%20layer%20--%3E%0D%0A%3CDIV%20class%3DmainHeaderLogoSpacer%3E%3CA%20class%3DboltLogoLink%20href%3D%22http%3A//www.bolt.com/%22%3E%3CIMG%20src%3D%22http%3A//www.bolt.com/imgs/icons/boltHeaderIcon.gif%22%20border%3D0%3E%3C/A%3E%3C/DIV%3E%0D%0A%3CDIV%20class%3DmainHeaderContent%3E%0D%0A%3CDIV%20class%3DmainHeaderContentTop%3E%0D%0A%3CDIV%20id%3Dad728x90Container%3E%3C%21--%20begin%20ad%20tag%20%20%28ptile%3D2%29%20--%3E%0D%0A%3CSCRIPT%20language%3DJavaScript%20type%3Dtext/javascript%3E%0Aord%3DMath.random%28%29*10000000000000000%3B%0Adocument.write%28%27%3Cscript%20language%3D%22JavaScript%22%20src%3D%22http%3A//ad.doubleclick.net/adj/filelodge.dart/sub%3Bpos%3Dheader%3Bptile%3D2%3Bdcopt%3Dist%3Bsz%3D728x90%3Bord%3D%27%20+%20ord%20+%20%27%3F%22%20type%3D%22text/javascript%22%3E%3C%5C/script%3E%27%29%3B%0A%3C/SCRIPT%3E%0D%0A%0D%0A%3CSCRIPT%20language%3DJavaScript%20src%3D%22http%3A//ad.doubleclick.net/adj/filelodge.dart/sub%3Bpos%3Dheader%3Bptile%3D2%3Bdcopt%3Dist%3Bsz%3D728x90%3Bord%3D4426118941585843%3F%22%20type%3Dtext/javascript%3E%3C/SCRIPT%3E%0D%0A%3CA%20href%3D%22http%3A//ad.doubleclick.net/click%253Bh%3Dv6%7C340d%7C3%7C0%7C%252a%7Co%253B31209932%253B0-0%253B0%253B13157916%253B3454-728%7C90%253B15956705%7C15974600%7C1%253B%253B%257Esscs%253D%253fhttp%253a%252f%252fwww.verbnow.com%22%20target%3D_new%3E%3CIMG%20height%3D90%20src%3D%22http%3A//ad.doubleclick.net/1138005/VERB_MARCH_06_728x90.jpg.jpg%22%20width%3D728%20border%3D0%3E%3C/A%3E%3C%21--%20Begin%20Interstitial%20Ad%20--%3E%0D%0A%3CSCRIPT%20src%3D%22http%3A//ds.serving-sys.com/BurstingScript/ebServing_123960.js%22%3E%3C/SCRIPT%3E%0D%0A%3C/DIV%3E%3C/DIV%3E%3C/DIV%3E%3C/DIV%3E%3C/DIV%3E%3C/BODY%3E'. Make sure the path or Internet address is correct.
OK

Wytyczne projektowe interfejsów GUI

- Zgodność z projektem abstrakcyjnym.
- Czytelność reprezentowanych obiektów.
- Zgodność ze środowiskiem – odpowiednia metafora interfejsu.
- Zgodność z wytycznymi platformy:
  - definicję kolorów tła, ikon, innych elementów,
  - rozmieszczenie i odległości w oknach dialogowych,
  - wymagania dotyczące różnych postaci menu,
  - skróty, lokalizacja elementów, postać elementów nieaktywnych.

Przykładowe wytyczne projektowe Java




Window frame
Menu bar and menus
Toolbar and toolbar buttons
Editor pane with styled text plug-in kit
Scroll pane
Engineering Models - MetalEdit
File Edit Format View Help
Exploiting new technology
Java Look and Feel
Secondary 1
Secondary 2
Secondary 3
Close
Close
Close
Tab 1
Tab 2
$1,432.27
Window frame
Menu bar and menus
Toolbar and toolbar buttons
Panel
Scroll pane
Editor pane
Styled text plug-in kit
Scrollbar in scroll pane

Wytyczne projektowe GNOME (*GNOME Human Interface Guidelines*)

Core material (podstawy):

- **Design principles** (zasady projektowania) – ogólne wytyczne projektowe.
- **Patterns** (wzór, szablon) – niezbędne i opcjonalne elementy konstrukcyjne.
- **User interface elements** (elementy interfejsu użytkownika) – wytyczne dotyczące typowych elementów, takich jak przyciski, paski postępu, itd.

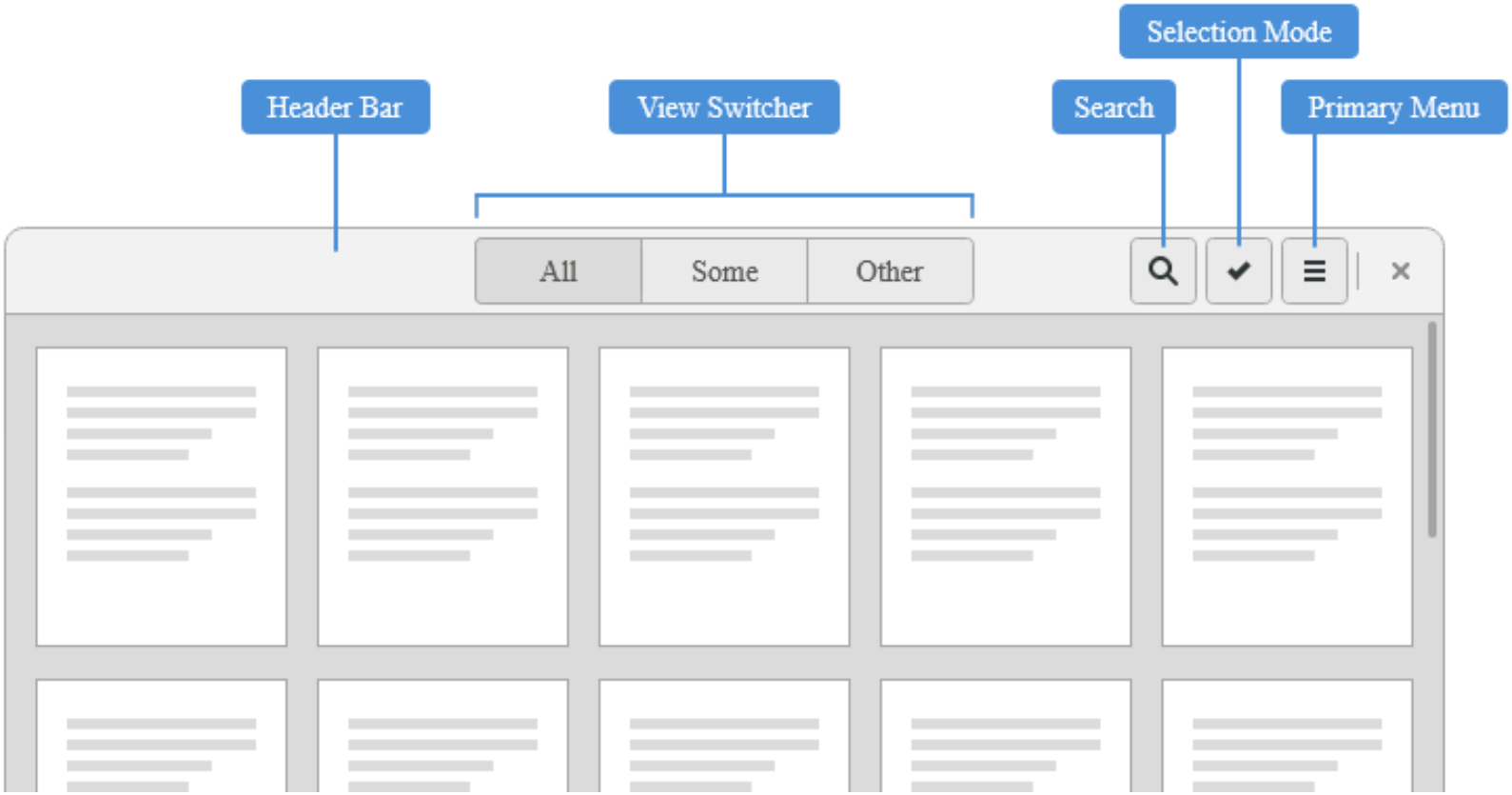
Common guidelines (wytyczne):

- **Application basics** (podstawy projektowania aplikacji) – podstawowe zachowanie i cechy aplikacji.
- **Compatibility** (zgodność) – korzystanie z wytycznych dla aplikacji wieloplatformowych lub dedykowanych GNOME.
- **Visual layout** (układ wizualny) – układ elementów w interfejsie graficznym użytkownika.
- **Writing style** (styl pisania) – sposób pisania, stosowanie skrótów i dużych liter.

- **Icons and artwork** (ikony i elementy grafiki) – wskazówki dotyczące wybierania i tworzenia ikon.
- **Typography** – (typografia) – porady dotyczące rozmiarów i stylów czcionek, a także znaków specjalnych.
- **Pointer and touch input** (wskaźnik i wprowadzanie dotykowe) – interakcja myszy, touchpada i ekranu dotykowego.
- **Keyboard input** (klawiatura) – zasady obsługi klawiatury, nawigacja, skróty klawiaturowe.
- **Display compatibility** (wyświetlacz) – jak obsługiwać różne typy wyświetlaczy.

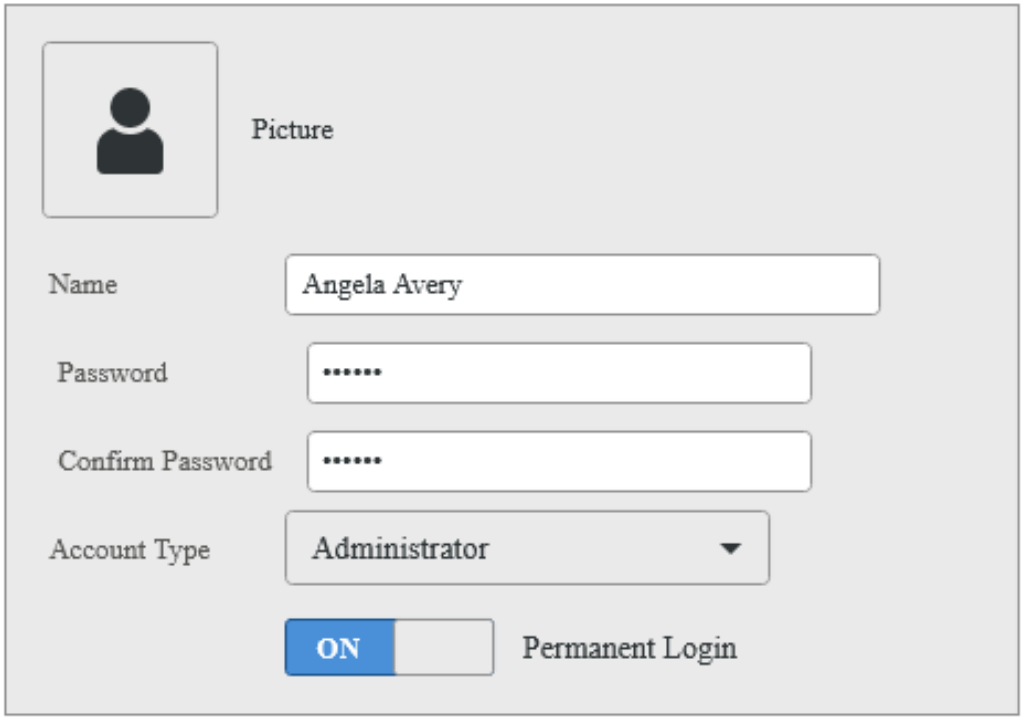
Przykłady są ułamkiem (początki poszczególnych rozdziałów) całego dokumentu.

Okno

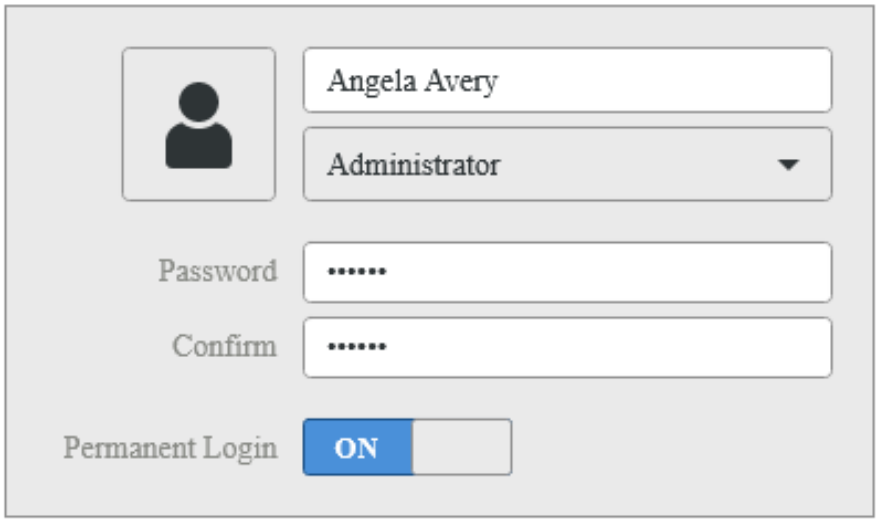


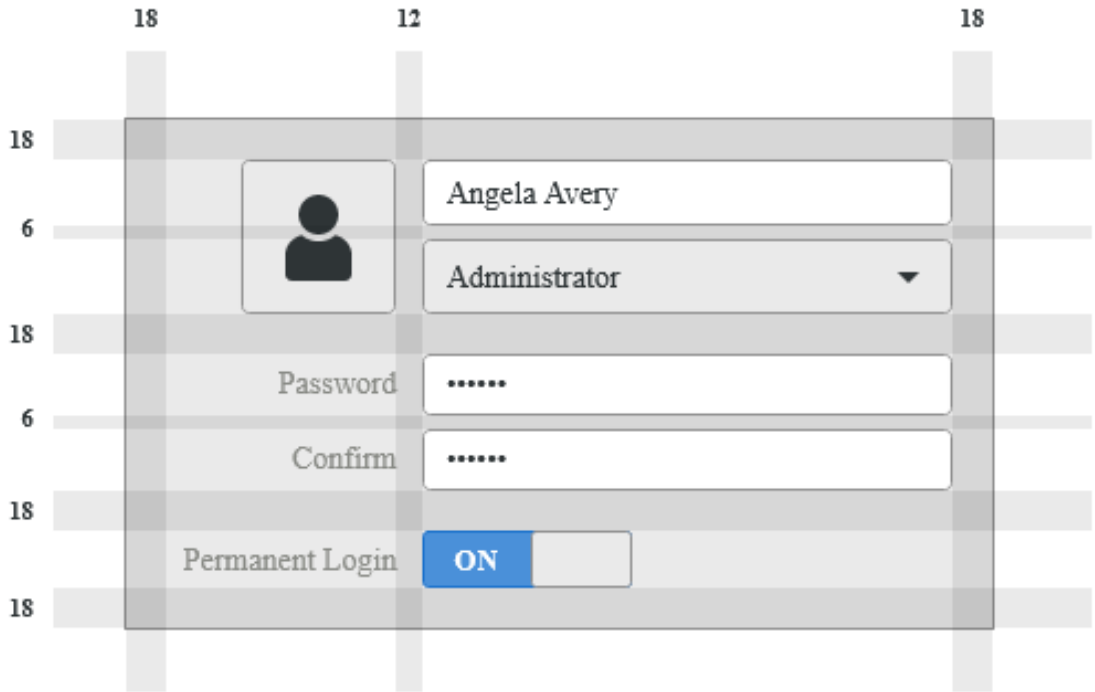
Układ i rozmieszczenie

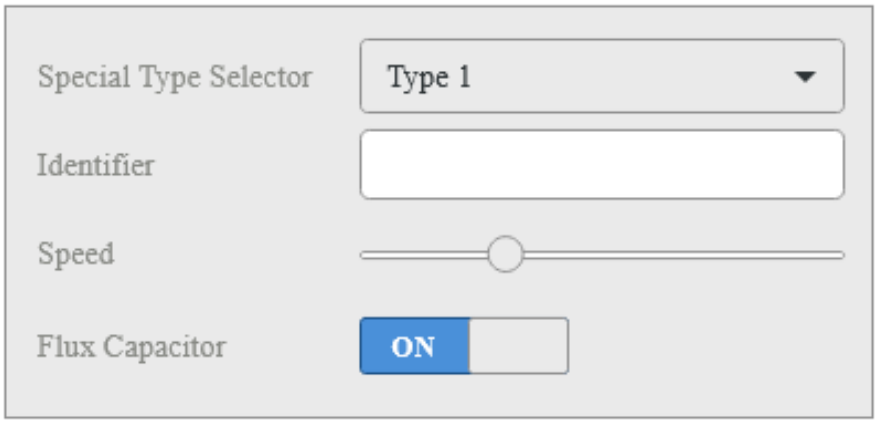
Incorrect spacing and alignment:

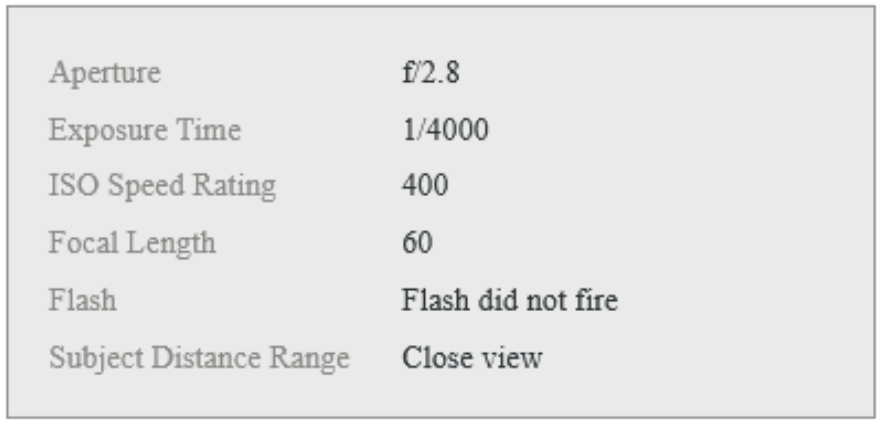


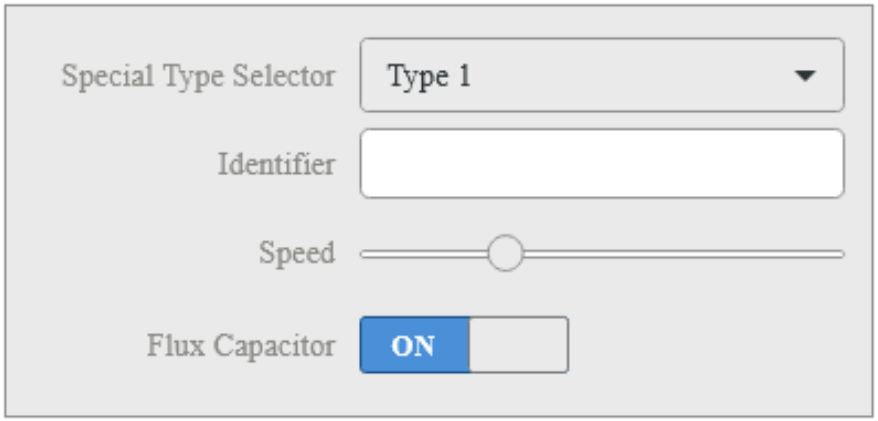
Correct spacing and alignment:

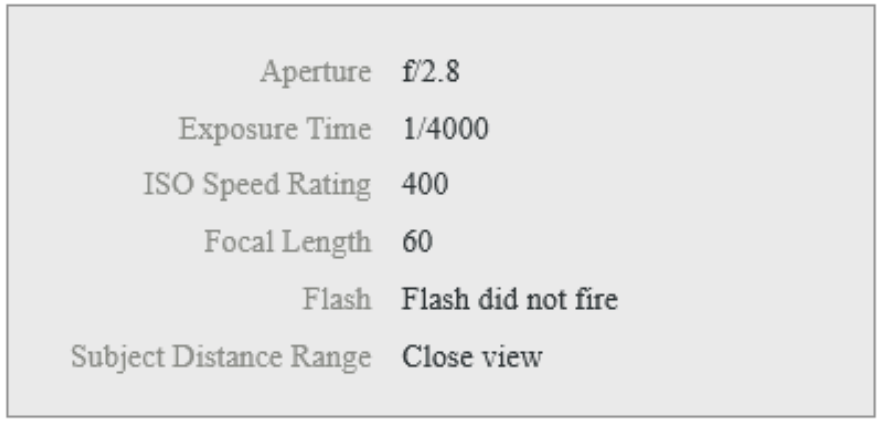



18
12
18
18
6
18
6
18
18
Angela Avery
Administrator
Password
Confirm
Permanent Login
ON


Special Type Selector
Type 1
Identifier
Speed
Flux Capacitor
ON


Aperture
f/2.8
Exposure Time
1/4000
ISO Speed Rating
400
Focal Length
60
Flash
Flash did not fire
Subject Distance Range
Close view


Special Type Selector
Type 1
Identifier
Speed
Flux Capacitor
ON


Aperture
f/2.8
Exposure Time
1/4000
ISO Speed Rating
400
Focal Length
60
Flash
Flash did not fire
Subject Distance Range
Close view

Typografia

Default fonts

Wherever possible, use the default system fonts as provided by the distribution or operating system on which your application is running. In GNOME 3, the default font is Cantarell, which was originally designed and developed by David Crossland.

Variants, sizes and weights

Different text weights and colors can and should be used to distinguish different kinds of information. At the same time, too many variants, sizes, and weights can make text harder to read and isn't an efficient or elegant way to convey information. Make an effort to minimize the range of font variants, sizes and weights.

- Use smaller and/or lighter text for less important information, and heavier/darker text to attract attention to important text.
- Avoid the use of italic or oblique faces, as these are visually more complex, and can be distracting.
- Never capitalize every letter in a word or sentence. Shouting at your users isn't nice.
- Do not use graphical backdrops or "watermarks" behind text. These interfere with the contrast between the text and its background.

Take advantage of Unicode

Unicode provides a wide variety of characters which, when used correctly, can dramatically improve the impression given by your application. The following Unicode characters are recommended:

Usage	Incorrect	Correct	Unicode to use
Quotation	"quote"	“quote”	U+201C LEFT DOUBLE QUOTATION MARK, U+201D RIGHT DOUBLE QUOTATION MARK
Time	4:20	4:20	U+2236 RATIO
Multiplication	1024x768	1024×768	U+00D7 MULTIPLICATION SIGN
Ellipsis	Introducing...	Introducing…	U+2026 HORIZONTAL ELLIPSIS
Apostrophe	The user's preferences	The user’s preferences	U+2019 RIGHT SINGLE QUOTATION MARK
Bullet list	- One\n- Two\n- Three	• One\n • Two\n • Three	U+2022 BULLET
Ranges	June-July 1967	June–July 1967	U+2013 EN DASH

Klawiatura

Keyboard navigation

Make sure that it is possible to move around and interact with every part of your user interface using the keyboard, by following these guidelines.

- Follow the standard GNOME keys for navigation. `Tab` is the standard key for moving around an interface with GTK+ and GNOME.
- Use a logical keyboard navigation order. When navigating around a window with `Tab`, keyboard focus should move between controls in a predictable order. In Western locales, this is normally left to right and top to bottom.
- In addition to navigation using `Tab`, make an effort to allow movement using the arrow keys, both within user interface elements (such as lists, icon grids or sidebars), and between them.

Function	Shortcut	Legacy Shortcut	Description
Activities Overview	Super	None	Opens and closes the Activities Overview
Applications View	Super + A	None	Opens and closes the applications view of the Activities Overview
Message Tray	Super + M	None	Toggles the visibility of the Message Tray.
Lock	Super + L	None	Locks the system by blanking the screen and requiring a password to unlock, if one has been set.

Application

Standard application keyboard shortcuts and menu items. These application shortcuts should not be reassigned to other actions, even when the corresponding action is not provided by your application.

Label	Shortcut	Description
Help	F1	Opens the default help browser on the contents page for the application.
About	None	Opens the About dialog for the application. Use the standard GNOME 3 dialog for this.
Quit	Ctrl + Q	Closes the application, including all application windows.

File

Standard file keyboard shortcuts and menu items.

Label	Shortcut	Description
New	Ctrl + N	Creates a new content item, often (but not always) in a new primary window or tab. If your application can create a number of different types of document, you can make the New item a submenu, containing a menu item for each type. Label these items New document type, make the first entry in the submenu the most commonly used document type, and give it the Ctrl + N shortcut.
Open...	Ctrl + O	Opens an existing content item, often by presenting the user with a standard Open File dialog. If the chosen file is already open in the application, raise that window instead of opening a new one.
Open Recent	None	A submenu which contains a list of no more than six recently used files, ordered according to most recently used.
Save	Ctrl + S	Saves the current content item. If the document already has a filename associated with it, save the document immediately without any further interaction from the user. If there are any additional options involved in saving a file, prompt for these first time the document is saved, but subsequently use the same values each time until the user changes them. If the document has no current filename or is read-only, selecting this item should be the same as selecting Save As.

Edit

Standard edit keyboard shortcuts and menu items.

Label	Shortcut	Description
Undo *action*	Ctrl + Z	Reverts the effect of the previous action.
Redo *action*	Shift + Ctrl + Z	Performs the next action in the undo history list, after the user has moved backwards through the list with the Undo command.
Cut	Ctrl + X	Removes the selected content and places it onto the clipboard. Visually, remove the content from the document in the same manner as Delete.
Copy	Ctrl + C	Copies the selected content onto the clipboard.
Paste	Ctrl + V	Inserts the contents of the clipboard into the content item. When editing text, if there is no current selection, use the caret as the insertion point. If there is a current selection, replace it with the clipboard contents.

Myszka i gesty

Primary and secondary buttons

Mice and touchpads often have two main buttons. One of these acts as the primary button, and the other acts as the secondary button. Typically, the left button is used as the primary button and the right button is used as the secondary button. However, this order is user-configurable and does not translate to touchscreen input. These guidelines therefore refer to primary and secondary action, rather than left and right.

Use the primary action for selecting items and activating controls. The secondary action can be used for accessing additional options, typically through a context menu.

Do not depend on input from secondary or other additional buttons. As well as being physically more difficult to click, some pointing devices and many assistive technology devices only support or emulate the primary button.

Press and hold should be used to simulate the secondary button on single button pointing devices. Therefore, do not use press and hold for other purposes.

Mouse and keyboard equivalents

Ensure that every operation in your application that can be done with the mouse can also be done with the keyboard. The only exceptions to this are actions where fine motor control is an essential part of the task. For example, controlling movement in some types of action games, or freehand painting in an image-editing application.

If your application allows items to be selected, the following equivalent actions should be in place.

Action	Mouse	Keyboard
Open an item	Primary button	Space
Add/remove item from selection	Ctrl and primary button	Ctrl + Space
Extend selection	Shift and primary button	Shift in combination with any of the following: Space Home End PageUp PageDown
Change selection	Primary button	Any of the following: ← ↑ → ↓ Home End PageUp PageDown
Select all	Primary button on first item, then primary button and Shift on the last item	Ctrl + A
Deselect all	Primary click on the container background	Shift + Ctrl + A

Application touch conventions

Using touch input consistently with other applications will allow users to easily learn how to use your application with a touch screen. The following conventions are recommended, where relevant.

Action	Description	Result
Tap		
	Tap on an item.	Primary action. Item opens — photo is shown full size, application launches, song starts playing.
Press and hold		
	Press and hold for a second or two.	Secondary action. Select the item and list actions that can be performed.
Drag		
	Slide finger touching the surface.	Scrolls area on screen.

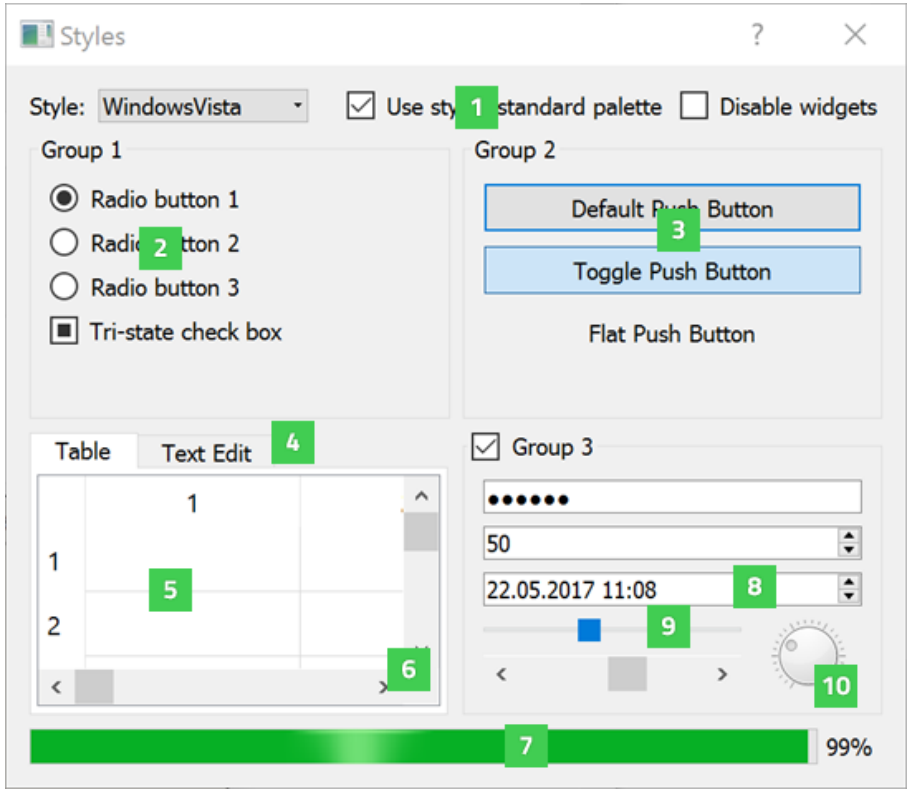
Podsumowanie

- Podstawy użyteczności:
  - projektuj dla ludzi,
  - używaj metafor,
  - utrzymuj zgodność aplikacji,
  - informuj użytkownika,
  - aplikacja powinna być prosta i ładna,
  - wybaczaj użytkownikowi,
  - udostępnij możliwość bezpośredniej manipulacji.
- Integracja z pulpitem:
  - zarejestruj skojarzenie do dokumentów,
  - umieszczaj skrót do aplikacji w menu pulpitu,
  - oprócz nazwy *marketingowej* dołącz człon związany z funkcjonalnością,
  - opracuj dokumentację użytkownika i system podpowiedzi (*tooltip*, dymki).

- Używaj odpowiednich typów okien:
  - podstawowe okno aplikacji,
  - okna pomocnicze,
  - okna postępu,
  - okna dialogowe,
  - okna kreatorów i asystentów.
- Używaj odpowiednich typów menu:
  - zgrupowany podstawowy pasek menu i pod-menu (*submenu*),
  - używaj komend, pól wyboru i przełączników (*checkbox*, *radio button*),
  - używaj menu wyskakujących związanych z elementami interfejsu(*pop-up menu*).

- Pamiętaj o odpowiedniej organizacji pasków narzędzi:
  - staraj się nie używać pionowych pasków narzędzi,
  - pozwól na dostosowani wyglądu,
  - pozwól na ukrywanie pasków,
  - stosuj podpowiedzi.

- Używaj odpowiednich kontrolek (*widget*):
  - steruj widocznością kontrolek,
  - steruj aktywnością kontrolek,
  - dobierz odpowiednią kontrolkę do zadania.



- Czasy reakcji:
  - 0,1 s – kliknięcia myszką, ruch wskaźnika, przesunięcie okna, wciśnięcie klawisza,
  - 1 s – wyświetlanie okna postępu zadań, wykonanie standardowych komend użytkownika, 10 s – wyświetlanie wykresów lub innych elementów, o których użytkownika spodziewa się, że są pracochłonne.
- Poprawienie odbioru przy długich czasach reakcji:
  - wyświetl informację najszybciej jak to możliwe,
  - pokazuj przybliżone/częściowe wyniki zanim wyznaczysz dokładne/całe,
  - przygotuj wcześniej dane przewidując typowe komendy,
  - mniej ważne czynności wykonuj w tle,
  - wyświetlaj przybliżony czas zakończenia zadania,
  - wizualizuj postęp (wskaźnik myszy, paski postępu, okna z informacją),
  - umożliwiaj przerwanie operacji.

- Interakcja użytkownika:
  - staraj się zachować niezależność od prawego przycisku myszy,
  - upewnij się, że każda operacja może być wykonana zarówno za pomocą myszki i klawiatury,
  - nie ograniczaj obszaru przemieszczania się kursora,
  - unikaj wymagania wciskania kilku klawiszy myszki jednocześnie,
  - zezwól anulować operację myszki po wciśnięciu klawisza [ESC],
  - kontroluj logiczną nawigację za pomocą klawisza [TAB].

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

Windows API, WinAPI (*Application Programming Interface*) to interfejs programistyczny systemu operacyjnego Windows. Jest to zestaw funkcji, stałych i zmiennych umożliwiających działanie programu w systemie operacyjnym.

Korzystanie z WinAPI jest możliwe w bardzo wielu wielu językach programowania (potrzebna jest tylko obsługa bibliotek DLL), m.in. C, C++, C#, Visual Basic, Object Pascal, Python a nawet asembler.

Obecnie większość środowisk programistycznych posiada zaimplementowane odpowiednie pliki nagłówkowe umożliwiające korzystanie z WinAPI.

Zbiór funkcji WinAPI jest bardzo obszerny i zawiera funkcje grupowane w kategorie:

- zarządzanie, administracja, serwisowanie i konfigurowanie systemu,
- monitoring wydajności,
- grafika 2D, 3D oraz multimedia,
- usługi sieciowe,
- bezpieczeństwo (np. kryptografia, autoryzacja, uwierzytelnianie),
- obsługa pamięci, plików, peryferiów, procesów i wątków,
- tworzenie i zarządzanie oknami.

Funkcje przychodzą wraz z plikami bibliotek DLL dostarczanymi z systemem, np. `kernel32.dll`, `user32.dll`, `gdi32.dll`, `wsock32.dll`, znajdującymi się w katalogu `%SystemRoot%\system32`.

Obecnie większość środowisk programistycznych posiada zaimplementowane odpowiednie pliki nagłówkowe umożliwiające korzystanie z WinAPI.

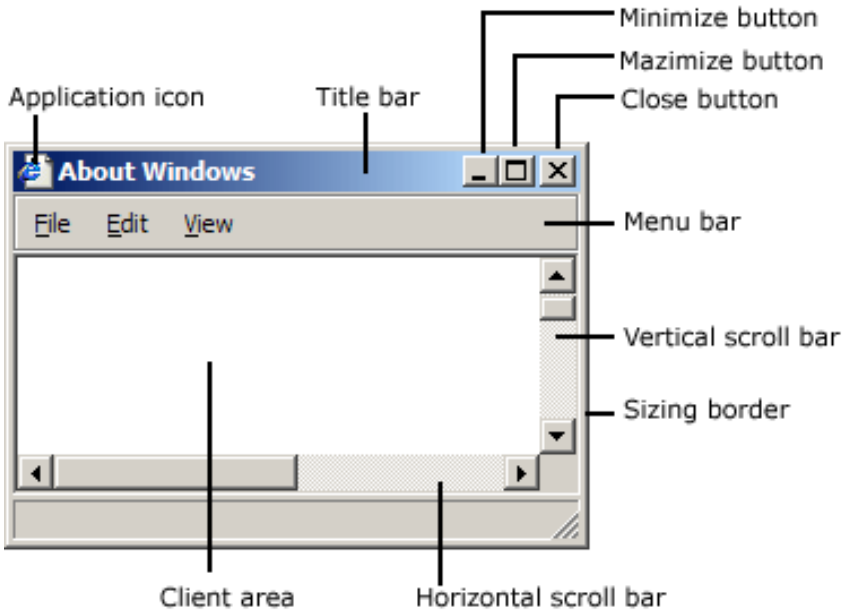
Aplikacja a okna

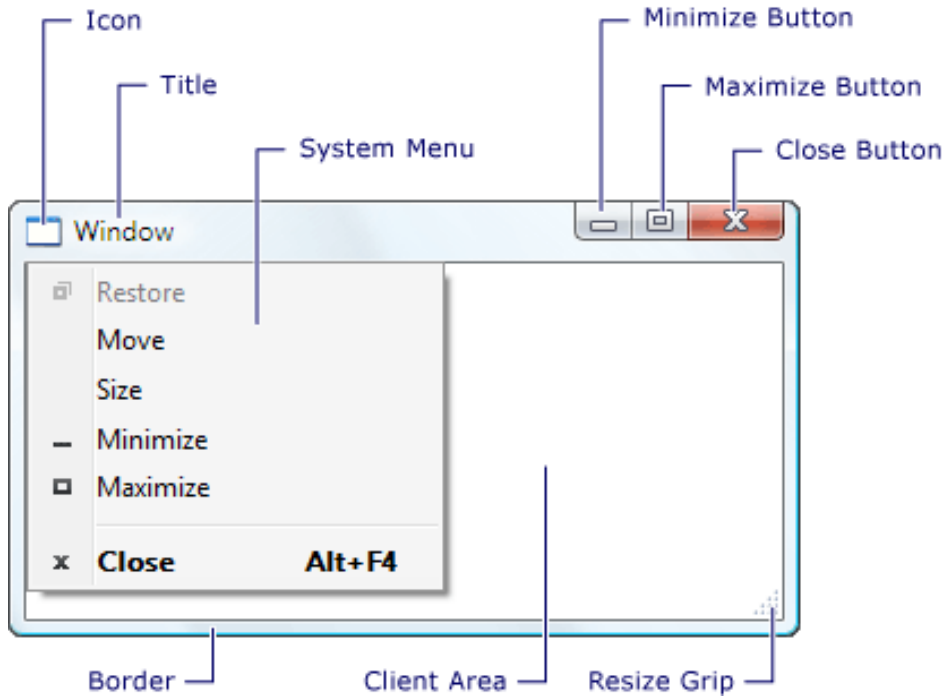
Aplikacja nie musi mieć żadnych okien, ale zwykle ma jedno lub wiele okien.

Aplikacja obsługująca pojedynczy dokument to tzw. aplikacja **SDI** (*Single Document Interface)*, np.: Paint, Notepad.

Aplikacja obsługująca i wyświetlająca wiele podobnych dokumentów to tzw. aplikacja **MDI** (*Multiple Document Interface*), np. przeglądarka WWW, zaawansowany program graficzny.

Budowa okna


Minimize button
Mazimize button
Close button
Application icon
Title bar
About Windows
File
Edit
View
Menu bar
Vertical scroll bar
Sizing border
Client area
Horizontal scroll bar


Icon
Title
System Menu
Minimize Button
Maximize Button
Close Button
Window
Restore
Move
Size
Minimize
Maximize
Close
Alt+F4
Border
Client Area
Resize Grip

Na marginesie, okno w systemie Windows składa się z dwóch części: **obszaru klienckiego** (*client area*) i **obszaru systemowego** (*nonclient area*).

Obszar kliencki okna to zwykle wewnętrzny obszar okna, którego zawartość aranżowana i obsługiwana jest przez programistę. Musi on zdefiniować kod, odpowiedzialny za *przerysowanie* zawartości obszaru klienckiego.

Obszar systemowy okna obejmuje obszar paska tytułu, menu, obramowanie okna, paski przewijania, ikony zamykania okna, minimalizacji, maksymalizacji, itd. Zarządzanie wyglądem oraz obsługa tych elementów okna realizowana jest zwykle przez system.

Komunikaty

Aplikacje w Windows działają na zasadzie **odpowiedzi na zdarzenia**, a przebieg programu zależy od zdarzeń, które zajdą w trakcie jego działania.

Zdarzenia mają swoje źródła – mogą nimi być klawiatura, myszka, timer systemowy lub inny działający program (lub element systemu operacyjnego).

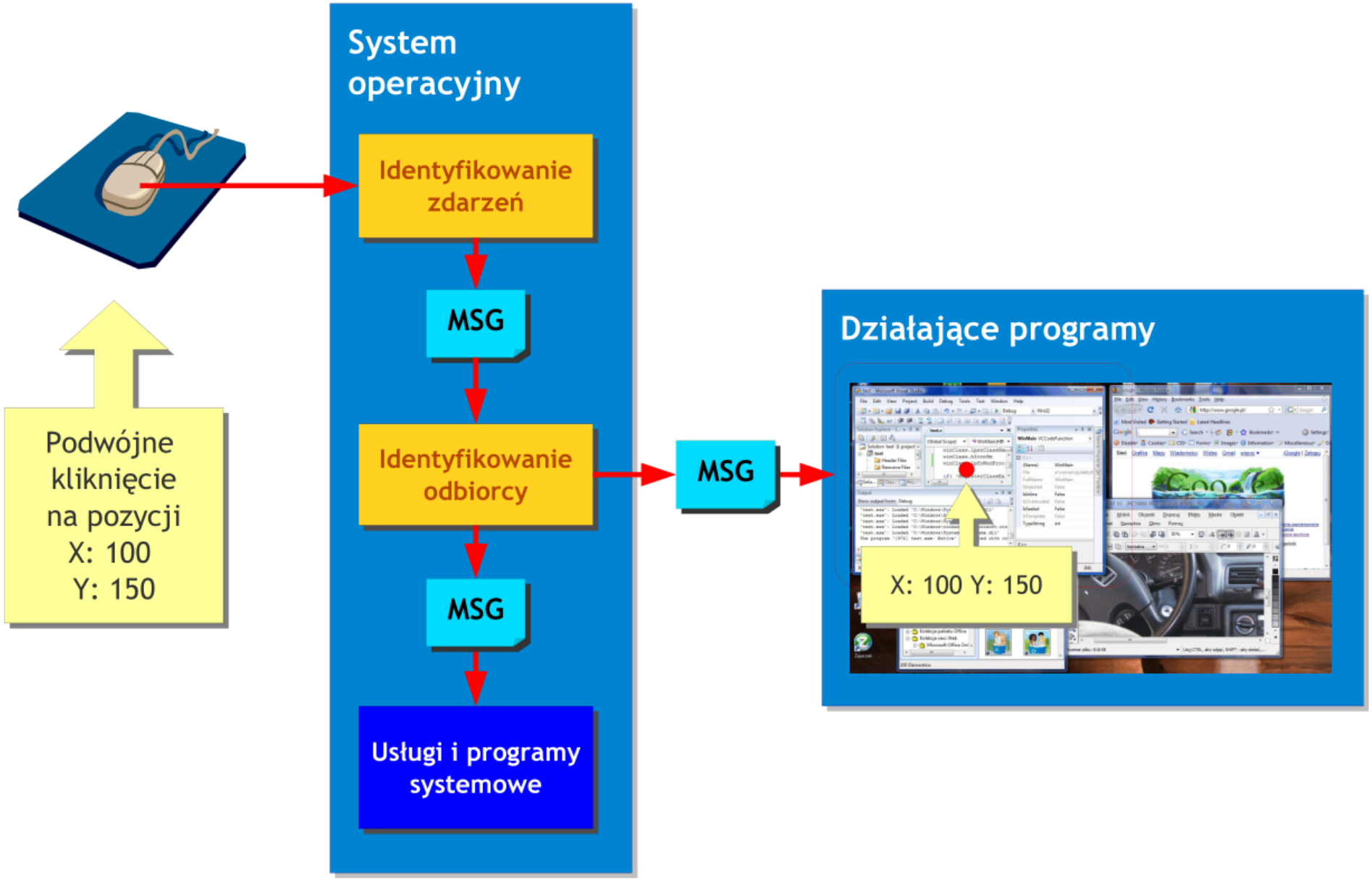
Program podejmuje akcję po otrzymaniu informacji o zaistnieniu zdarzenia dla niego dedykowanego. Polega to zwykle na uruchomieniu pewnej procedury obsługi zaistniałego zdarzenia.

Informacje o zdarzeniu zapisuje się w pewnej strukturze danych, którą przekazuje się programowi. Ta struktura danych to najczęściej rekord lub obiekt.

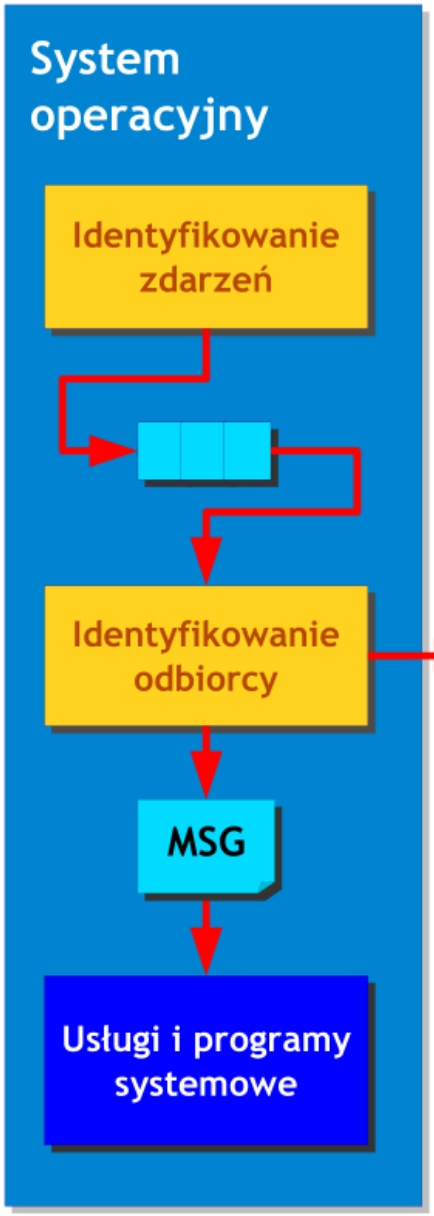
W WinAPI rekord informacji o zaistniałym zdarzeniu nazywa się **komunikatem** (*message*, *MSG*).

Zdarzenia wykrywa i rozpoznaje system operacyjny, po wypełnieniu rekordu komunikatu informacjami o zdarzeniu, przekazywany jest on do odpowiedniego odbiorcy, gdzie powinna nastąpić obsługa komunikatu.

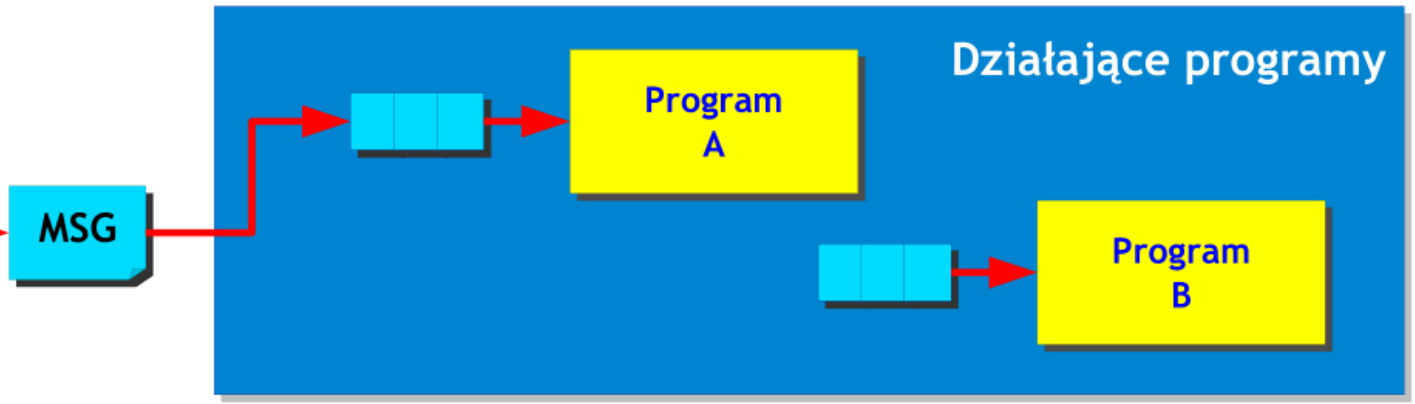
Przebieg obsługi zdarzenia



Kolejka komunikatów

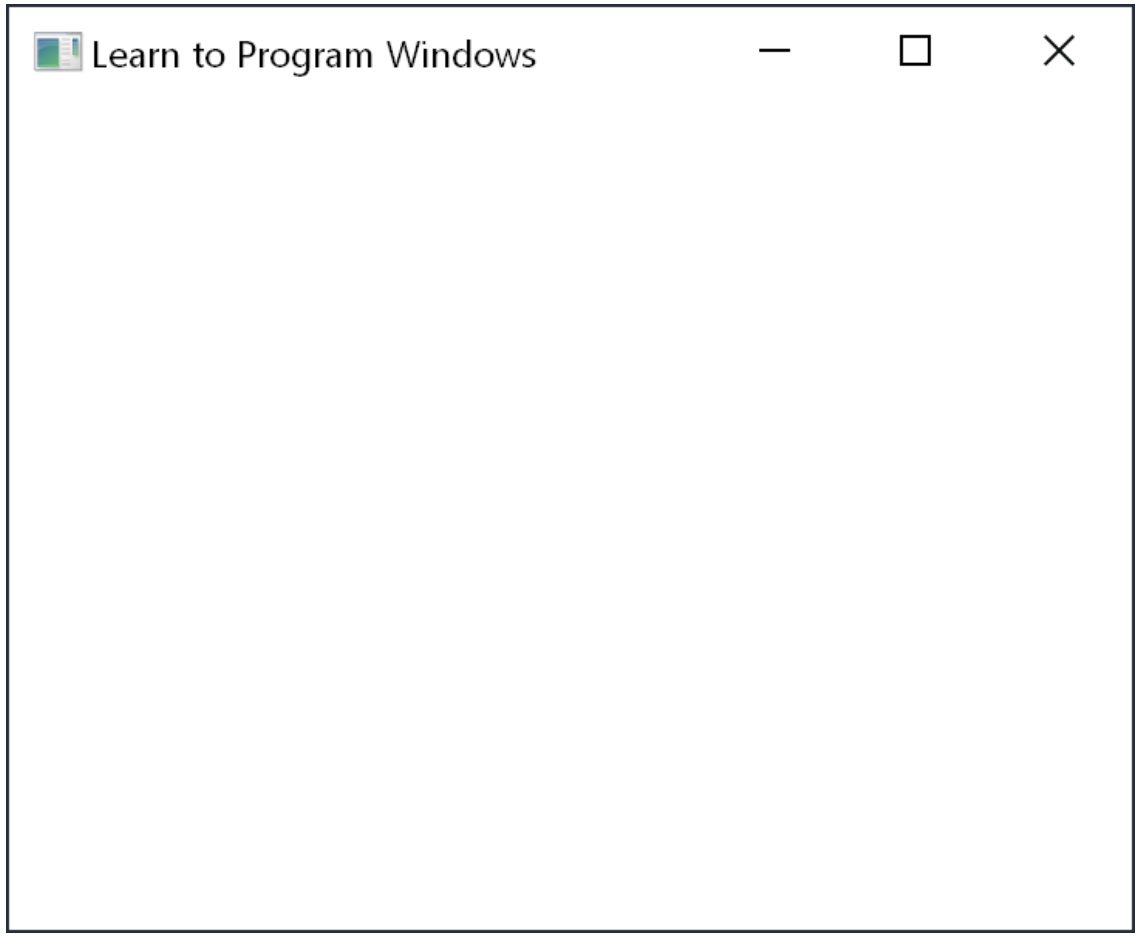


- Komunikaty opisujące zdarzenia są kolejkowane.
- System obsługuje pojedynczą systemową kolejkę komunikatów oraz kolejki przydzielane indywidualnie dla każdego programu (wątku) wykorzystującego GUI.



- Komunikaty wstawiane są do systemowej kolejki, skąd są pobierane przez system.
- Po określeniu odbiorcy, komunikat wstawiany jest do indywidualnej kolejki programu (wątku).

Przykładowy program wykorzystujący WinAPI (prosty)



```
#include <windows.h>

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
    ↪ LPARAM lParam);

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    ↪ PWSTR pCmdLine, int nCmdShow)
{
  // Register the window class.
  const wchar_t CLASS_NAME[] = L"Sample Window Class";

  WNDCLASS wc = {};

  wc.lpfnWndProc   = WindowProc;
  wc.hInstance     = hInstance;
  wc.lpszClassName = CLASS_NAME;

  RegisterClass(&wc);
```

```
// Create the window.
HWND hwnd = CreateWindowEx(
  0,                              // Optional window styles.
  CLASS_NAME,                     // Window class
  L"Learn to Program Windows",    // Window text
  WS_OVERLAPPEDWINDOW,            // Window style

  // Size and position
  CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

  NULL,       // Parent window
  NULL,       // Menu
  hInstance,  // Instance handle
  NULL        // Additional application data
  );

if(hwnd == NULL)
{
  return 0;
}
ShowWindow(hwnd, nCmdShow);
```

```
  // Run the message loop.
  MSG msg = {};
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return 0;
}
```

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
    ↪ LPARAM lParam)
{
  switch(uMsg)
  {
    case WM_DESTROY:
      PostQuitMessage(0);
      return 0;

    case WM_PAINT:
    {
      PAINTSTRUCT ps;
      HDC hdc = BeginPaint(hwnd, &ps);

      FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW + 1));

      EndPaint(hwnd, &ps);
    }
    return 0;
  }
  return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

Czym jest Qt?

Qt jest zestawem przenośnych bibliotek i narzędzi programistycznych dedykowanych dla języków **C++** i **QML**.

Ich podstawowym składnikiem są klasy służące do budowy graficznego interfejsu programów komputerowych, począwszy od wersji 4.0 Qt zawiera też narzędzia do tworzenia programów konsolowych i serwerów.

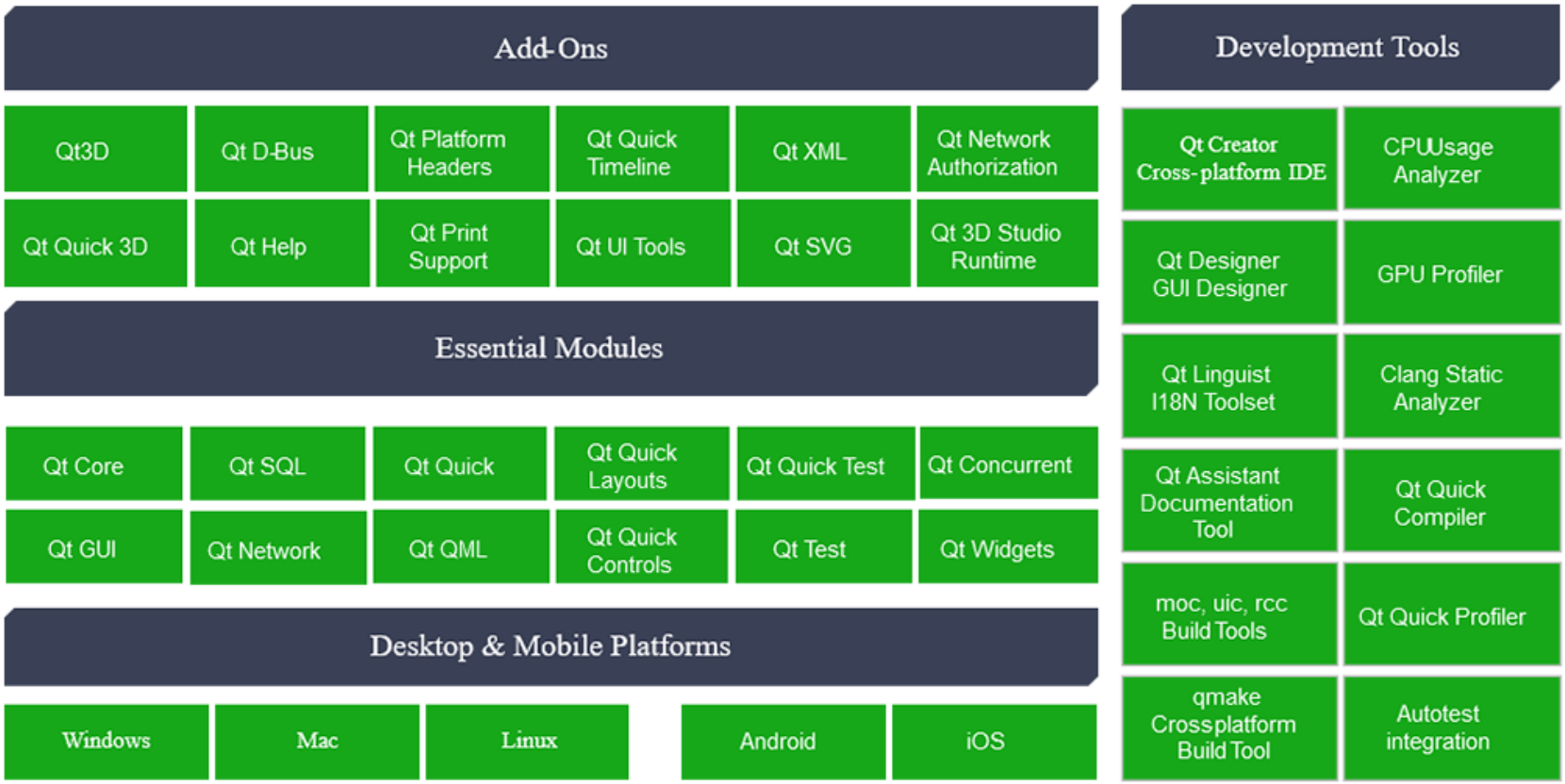
Środowisko Qt jest dostępne dla platform: **X11** (m.in. GNU/Linux, BSD, Solaris), **Windows**, **Mac OS X**, **Haiku** oraz dla urządzeń wbudowanych opartych na Linuksie, Windows CE, Symbian, Android.

Biblioteki Qt dostępne są w języku C++ i Java; mogą też być wykorzystywane w programach napisanych w innych językach, m.in. **Ada**, **C#**, **Pascal**, **Perl**, **PHP**, **Ruby** i **Python**.

Charakteryzują się w pełni obiektową architekturą. Zawierają wiele technologii programowania graficznego interfejsu użytkownika, istniejących wcześniej jedynie w **Tk**: mechanizm sygnałów i slotów, automatyczne rozmieszczanie widżetów oraz zhierarchizowany system obsługi zdarzeń.

Biblioteki Qt, oprócz obsługi interfejsu użytkownika, zawierają także niezależne od platformy systemowej moduły obsługi procesów, plików, sieci, grafiki trójwymiarowej (**OpenGL**), baz danych (**SQL**), **języka XML**, lokalizacji, wielowątkowości, zaawansowanej obsługi napisów oraz wtyczek. Qt składa się z modułów, które posiadają podobny schemat i API (*Application Programming Interface*).

Zawierają także własne, niezależne od biblioteki **STL** szablony klas kontenerów.



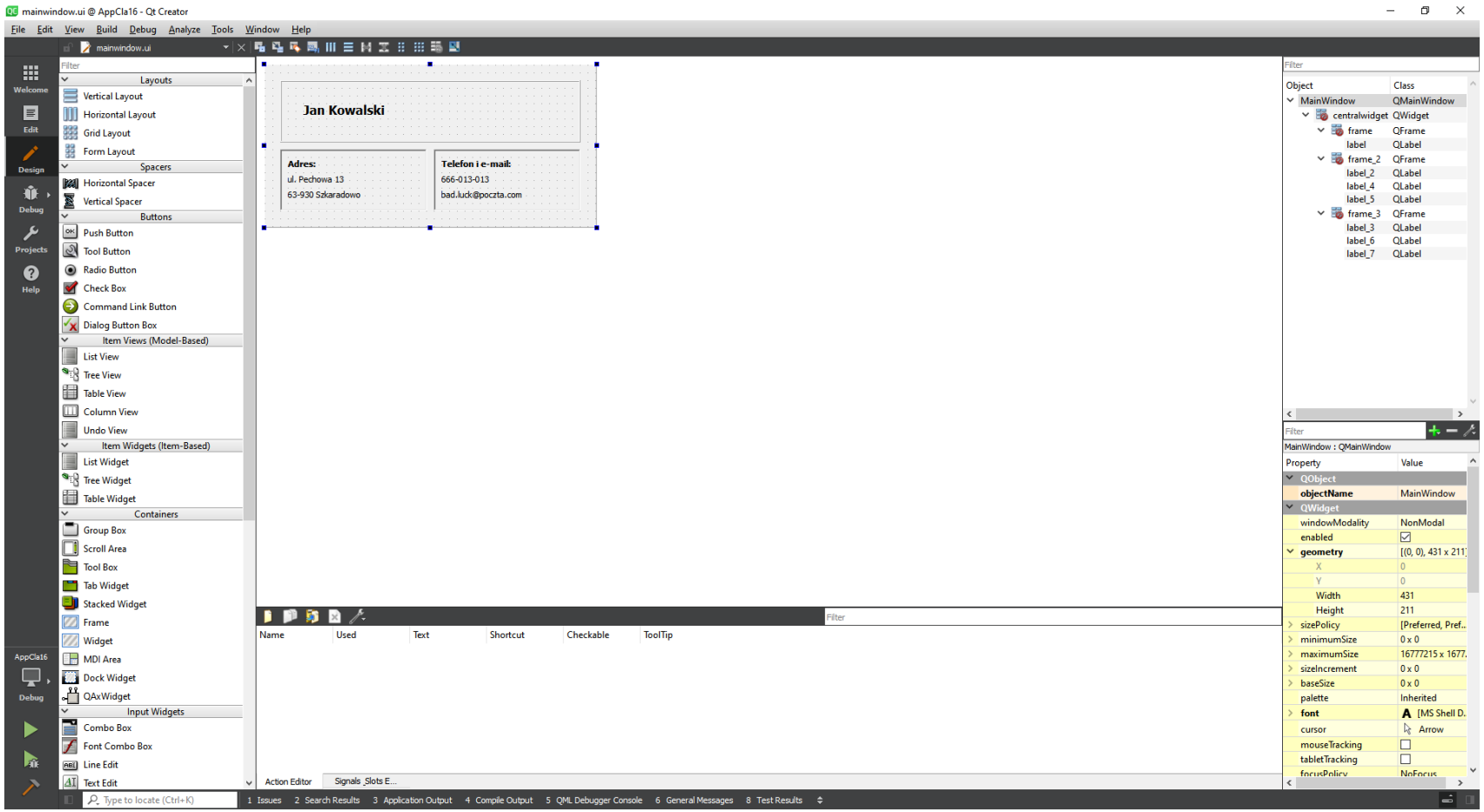
W skład Qt wchodzi wiele specjalistycznych narzędzi programistycznych. Są to m. in.:

- **moc** (*Meta Object Compiler*) – specjalny preprocesor, który na podstawie plików nagłówkowych (`*.h`) generuje dodatkowe pliki źródłowe (`*.cpp`),
- **uic** (*User Interface Compiler*) – kompilator plików `*.ui` zwykle generowanych za pośrednictwem programu **Qt Designer**,
- **qmake** – program do zarządzania procesem kompilacji, jego głównym zadaniem jest utworzenie, a później aktualizacja pliku `Makefile` na podstawie prostego opisu zawartego w definicji projektu (`*.pro`),
- **Qt Creator** – zintegrowane środowisko programistyczne (IDE).
- **Qt Designer** – aplikacja typu RAD pozwalająca na tworzenie wrstwy wizualnej aplikacji, obecnie zintegrowana z QT Creator,
- **Qt Linguist** – aplikacja wspomagająca tłumaczenie programu na różne języki,
- **Qt Assistant** – aplikacja zawierająca rozbudowany system pomocy dla programistów (podpowiedzi, dokumentacja).

Qt Creator używa kompilatora **G++** na Linuksie i OS X (obecnie macOS) oraz **MinGW** na Windowsie. Obsługiwany jest także **Clang** oraz **Intel C++ Compiler**. Bibliotek Qt można również używać z Visual Studio.

Qt Creator nie zawiera **debuggera**. Posiada jedynie plug-in, który działa jako interfejs pomiędzy środowiskiem a natywnym debuggerem C++. Obsługiwane debuggery to:

- GNU Debugger (GDB),
- Microsoft Console Debugger (CDB),
- LLVM debugger (LLDB),
- Wbudowany debugger JavaScript.


mainwindow.ui @ AppCla16 - Qt Creator
File Edit View Build Debug Analyze Tools Window Help
Jan Kowalski
Adres:
ul. Pechowa 13
63-930 Szkaradowo
Telefon i e-mail:
666-013-013
bad.luck@poczta.com
Layouts
Vertical Layout
Horizontal Layout
Grid Layout
Form Layout
Spacers
Horizontal Spacer
Vertical Spacer
Buttons
Push Button
Tool Button
Radio Button
Check Box
Command Link Button
Dialog Button Box
Item Views (Model-Based)
List View
Tree View
Table View
Column View
Undo View
Item Widgets (Item-Based)
List Widget
Tree Widget
Table Widget
Containers
Group Box
Scroll Area
Tool Box
Tab Widget
Stacked Widget
Frame
Widget
MDI Area
Dock Widget
QAxWidget
Input Widgets
Combo Box
Font Combo Box
Line Edit
Text Edit
Object
Class
MainWindow
QMainWindow
centralwidget
QWidget
frame
QFrame
label
QLabel
frame_2
QFrame
label_2
label_4
label_5
frame_3
label_3
label_6
label_7
MainWindow : QMainWindow
Property
Value
QObject
objectName
MainWindow
QWidget
windowModality
NonModal
enabled
geometry
Width
431
Height
211
sizePolicy
minimumSize
maximumSize
sizeIncrement
baseSize
palette
Inherited
font
cursor
Arrow
mouseTracking
tabletTracking
Name
Used
Text
Shortcut
Checkable
ToolTip
Action Editor
Signals &Slots E...
1 Issues
2 Search Results
3 Application Output
4 Compile Output
5 QML Debugger Console
6 General Messages
8 Test Results

QT Designer udostępnia raczej ograniczoną liczbę komponentów, nastawiając się na możliwość dodawania własnych dostosowanych do konkretnych potrzeb.

Układ elementów na ekranie oraz ich projekt leksykalny zapisywany jest w pliku z rozszerzeniem `*.ui` (format jak XML).

Plik ten jest za pomocą specjalnego narzędzia (**uic**) transformowany na kod w języku C++ (`ui_*.h`), który tworzy układ okienka.

Oczywiście w środowiskach zintegrowanych cały proces jest zautomatyzowany.

Qt rozszerza C++ między innymi dzięki wykorzystaniu **makr** i **introspekcji**.

```
foreach(int value, intList)
{
  ...
}
```

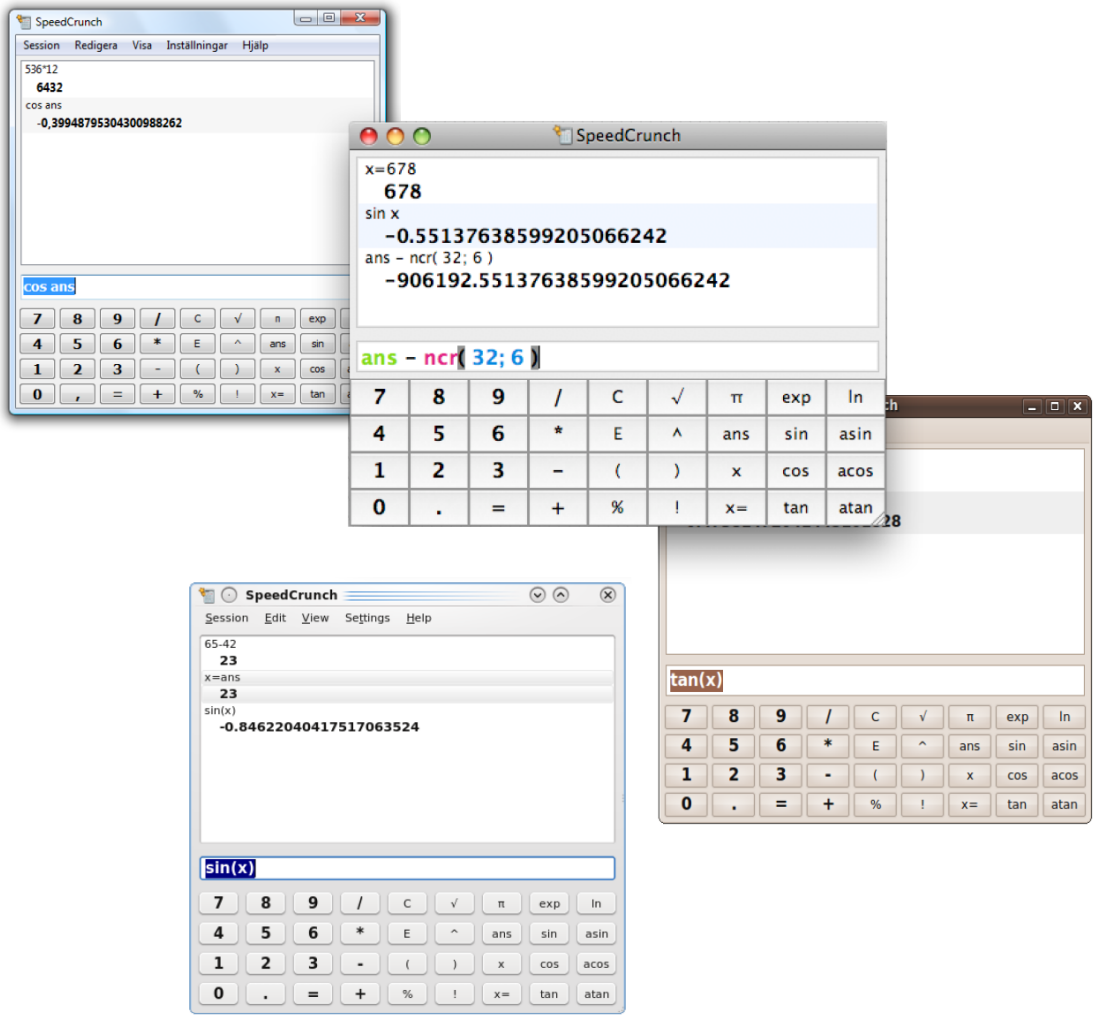
```
QObject *button = new QPushButton;
button->metaObject()->className(); // QPushButton
```

```
connect(button, SIGNAL(clicked()), QCoreApplication::instance(),
    ↪ SLOT(close()));
```

Kod to **nadal zwykły C++**.

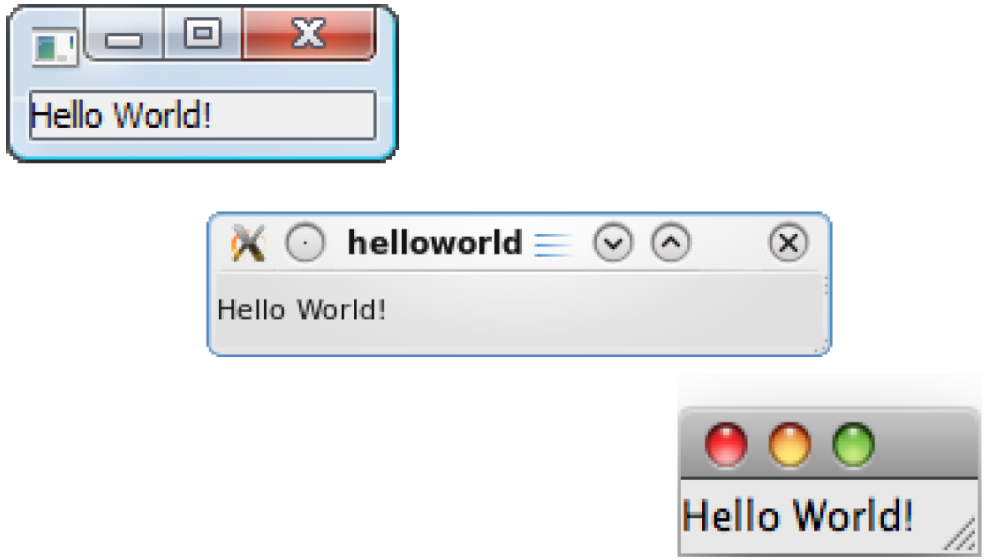
Dodatkowo Qt charakteryzuje się łatwym API, wysoką produktywnością i otwartością źródeł (w większości modułów).

Aplikacje można pisać pod szereg stacjonarnych (Windows, macOS, Linux/Unix X11) oraz mobilnych i wbudowanych ystemów operacyjnych (Android, Windows Phone, iOS, QNX).



Wieloplatformowe aplikacje budowanego na podstawie jednego kodu źródłowego.

Wygląd i sposób funkcjonowania jest natywny dla odpowiedniego środowiska, w którym są uruchamiane.



Gdzie Qt jest wykorzystywane?

Licencja Qt

LGPL (*Lesser General Public License*) – darmowa:

- aplikacja może być otwarto źródłowa lub zamknięta,
- zmiany dokonane w bibliotece Qt muszą zostać dostarczone społeczności.

GPL (*General Public License*) – darmowa:

- aplikacja musi być otwarto źródłowa,
- zmiany dokonane w bibliotece Qt muszą zostać dostarczone społeczności.

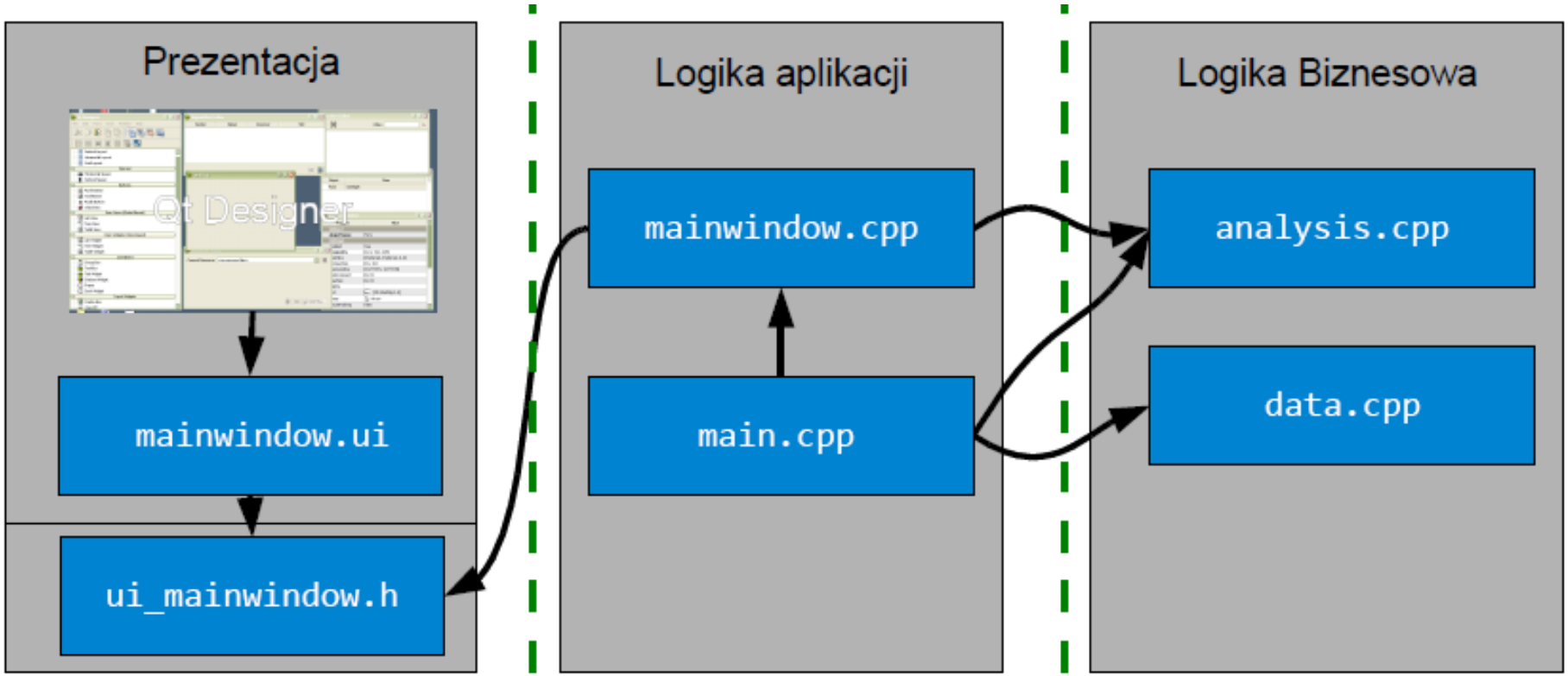
Commercial (komercyjna) – płatna:

- aplikacja może być zamknięta,
- nir trzeba przekazywać zmian dokonanych w bibliotece Qt społeczności.

Struktura aplikacji Qt (najprostszej)

```
#include <QApplication>
#include <QLabel>

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

  QLabel myLabel("Hello World!");
  myLabel.show();

  return myApp.exec();
}
```

Architektura aplikacji Qt



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.

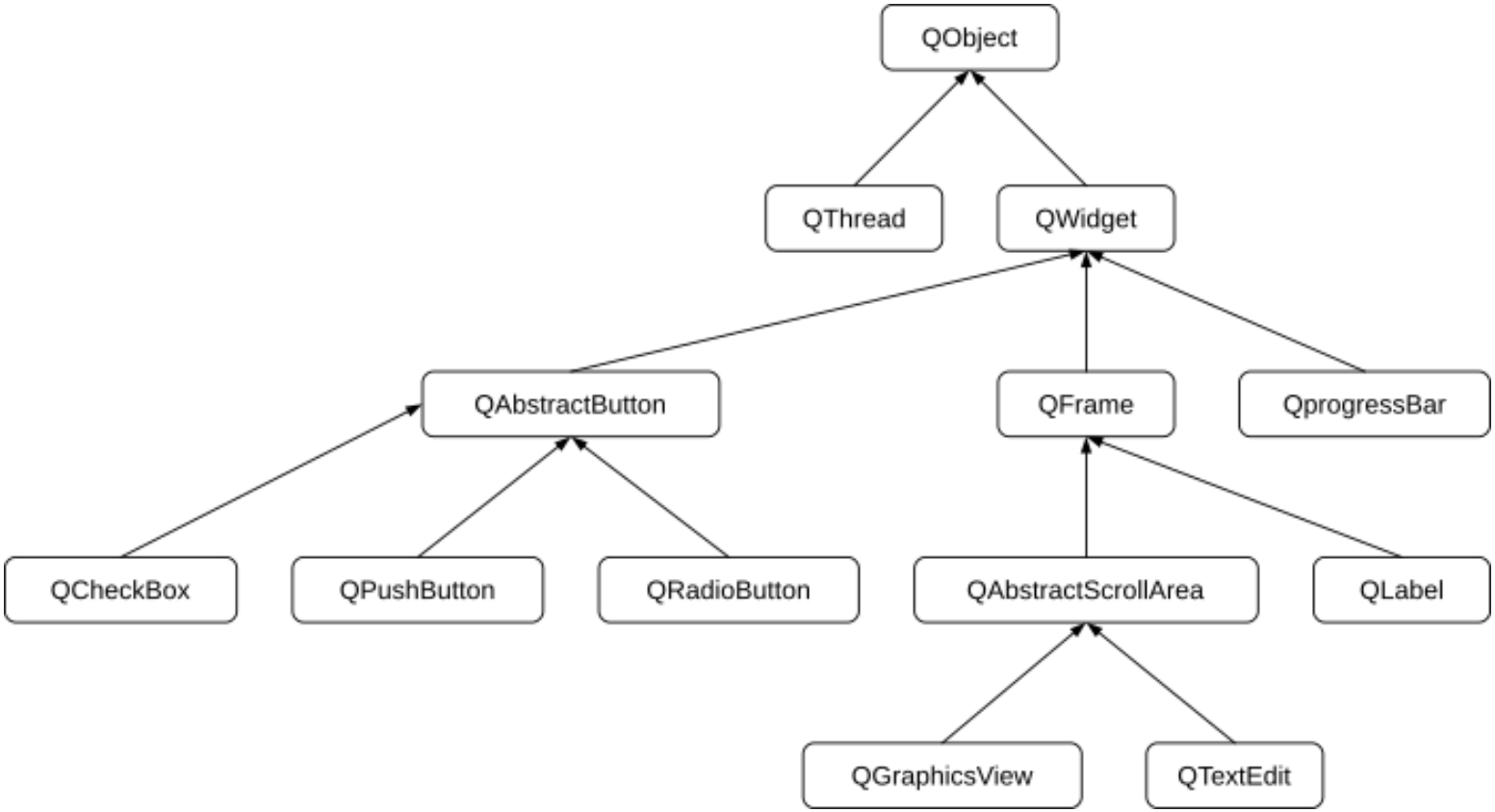
Całe szczęście! Niezależna od nikogo logika biznesowa, która musi udostępniać informacje o swoim stanie za pomocą odpowiedniego interfejsu.

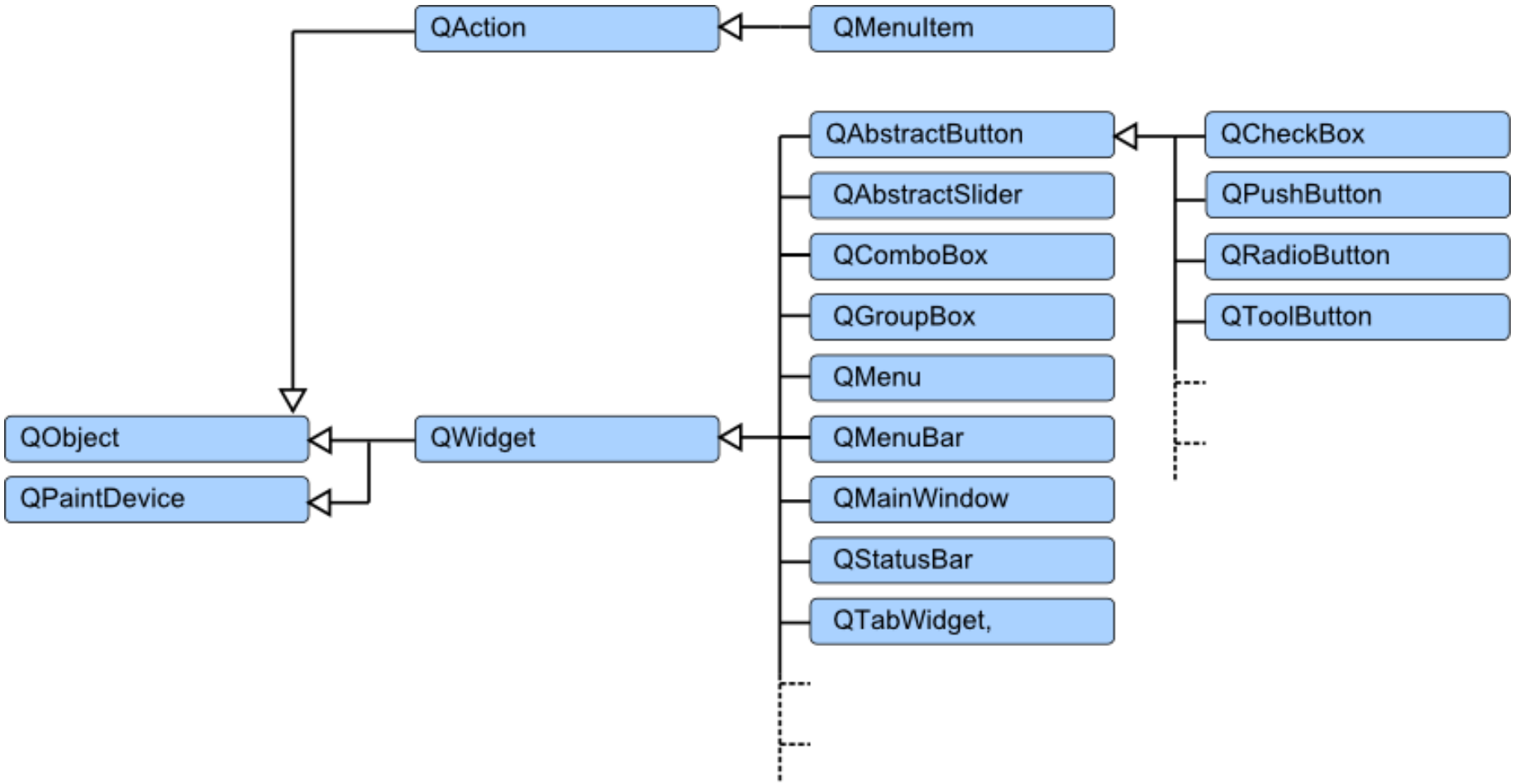
QObject

`QObject` **jest klasą bazową** dla bardzo wieli innych klas i widżetów Qt.

Klasa ta implementuje wiele mechanizmów, które stanowią podstawy biblioteki Qt:

- zdarzenia,
- sygnały,
- sloty,
- właściwości obiektów,
- mechanizmy zarządzania pamięcią.


QObject
QThread
QWidget
QAbstractButton
QFrame
QprogressBar
QCheckBox
QPushButton
QRadioButton
QAbstractScrollArea
QLabel
QGraphicsView
QTextEdit


QAction
QMenuItem
QAbstractButton
QCheckBox
QAbstractSlider
QPushButton
QComboBox
QRadioButton
QGroupBox
QToolButton
QMenu
QObject
QWidget
QMenuBar
QPaintDevice
QMainWindow
QStatusBar
QTabWidget,

`QObject` **nie jest klasą bazową** dla:

- klas, które muszą posiadać minimalny narzut obliczeniowy (np. realizujące podstawowe operacje graficzne),
- kontenerów danych (np.: `QString`, `QList`, `QChar`),
- klas, które mogą być kopiowane (ponieważ instancji klasy `QObject` nie można kopiować.

Każda instancja klasy `QObject` jest samodzielnym bytem. Zwykle posiada własną nazwę (`QObject::objectName`). Obiekt `QObject` ma określone miejsce w hierarchii innych instancji.

Meta dane

Biblioteka Qt posługuje się mechanizmem **introspekcji** (łac. *introspectio*, wglądanie do wnętrza).

Introspekcja jest to mechanizm pozwalający postrzegać funkcje i moduły znajdujące się w pamięci jako obiekty, a także pobierać o nich informacje, np. określać typ obiektu, budowę klasy obiektu.

Każdy obiekt klasy `QObject` posiada **metaobiekt** (*meta-object*) `QMetaObject` posiadający następujące informacje:

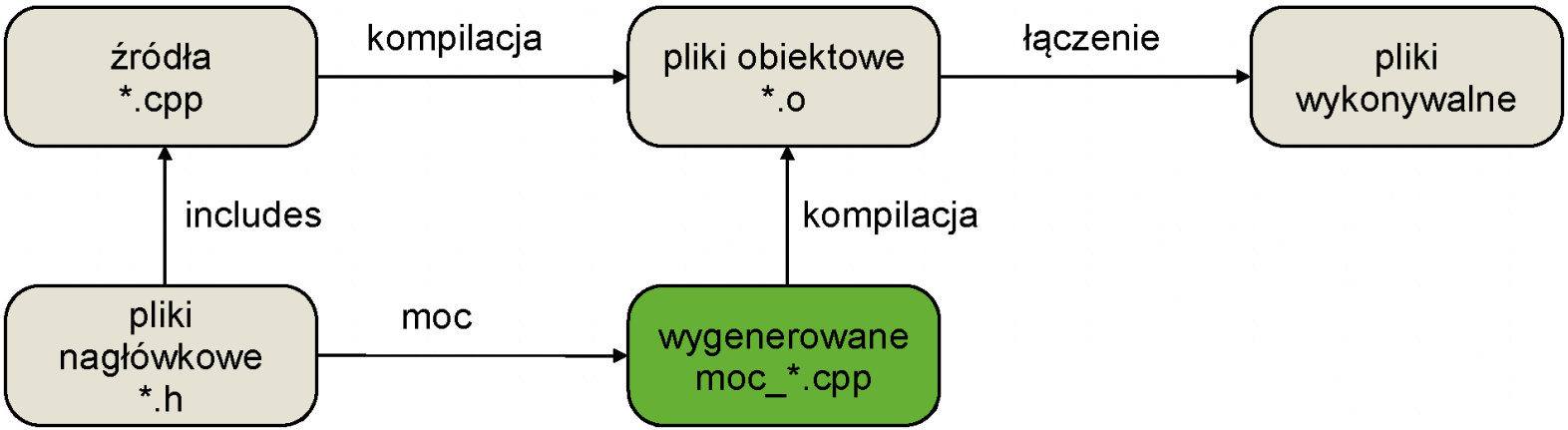
- nazwa klasy (`QObject::className`),
- dziedziczenie (`QObject::inherits`),
- właściwości, sygnały i sloty,
- informacje ogólne (`QObject::classInfo`).

Klasy mają wiedzę o sobie w czasie wykonywania programu.

Umożliwia to dynamiczne rzutowanie typów bez korzystania z mechanizmu **RTTI** (*run-time type information*).

```
if(object->inherits("QAbstractItemView"))
{
  QAbstractItemView *view = static_cast<QAbstractItemView*>(widget);
  view->...
}
```

Metadane są generowane podczas procesu kompilacji przez tzw. **kompilator metaobiektów** (*meta-object compiler*, *moc*).



Czego szuka moc?

```
class Counter : public QObject
{
  Q_OBJECT
  Q_CLASSINFO("author", "Wiktor Wektor")
  Q_PROPERTY(int value READ value WRITE setValue)

  public:
    Counter(QObject *parent = 0);
    int value() const;

  public slots:
    void setValue(const int);

  signals:
    void valueChanged(int);

  private:
    int m_value;
};
```

Klasa musi dziedziczyć z `QObject` i powinna zawierać makro `Q_OBJECT`.

Wartością makra `slots` jest **pusty ciąg znaków**, jest ono zatem obojętne dla kompilatora C++. Jednak jego wystąpienie jest niezbędne do prawidłowego działania kompilatora moc.

Wartością makra `signals` jest **słowo zarezerwowane języka** C++ `protected`, co oznacza, że każdy sygnał jest chronioną metodą klasy.

Stąd sygnały objęte są wszystkimi ograniczeniami związanymi z chronionymi metodami klas i mogą być wywoływane tylko z poziomu danej klasy, jej potomków oraz klas zaprzyjaźnionych.

Kompilator metaobiektów analizuje treść deklaracji klasy i generuje dodatkowy plik C++ zawierający metody obsługujące metaobiekty. Domyślnie plik generowany przez moc kierowany jest na standardowe wyjście i ma nazwę z przedrostkiem `moc_*`.

Z kompilatora moc można korzystać także bezpośrednio z wiersza poleceń. Typowe wywołanie wygląda następująco:

```
moc foo.h -o foo.cpp
```

Właściwości

Klasy dziedziczące po `QObject` powinny realizować właściwości przez metody akcesora i modyfikatora. Modyfikator często jest slotem.

```
class Counter : public QObject
{
  Q_OBJECT

  public:
    int value() const;

  public slots:
      void setValue(const int);
    ...
};
```

Zwykle modyfikator zwraca typ `void` i posiada jeden argument. Akcesor zwraca wartość i nie ma argumentów.

Konwencja nazewnictwa: `value` (akcesor), `setValue` (modyfikator). Dla zmiennych logicznych odpowiednio: `isEnabled`, `setEnabled`.

Dostęp do właściwości powinien być hermetyzowany.

```
class Counter : public QObject
{
  ...
  public slots:
    void setValue(const int);

  signals:
    void valueChanged(int);

  private:
    int m_value;
};
```

Można sprawdzić poprawność wartości.

```
void Counter::setValue(const int newValue)
{
  if(!checkValue(newValue))
  {
    ...
    return;
  }
  if(m_value != newValue)
  {
    m_value = newValue;
  }
}
```

Można reagować na zmiany (aktualizując elementy zależne od wartości danego atrybutu).

```
void Counter::setValue(const int newValue)
{
  ...
  if(m_value != newValue)
  {
    m_value = newValue;
    updateValue();
  }
}
```

Dlaczego używać akcesorów?

```
int Counter::value() const
{
  return m_value;
}
```

Akcesory pozwalają na odpowiednią separację użytych struktur danych.

```
QSize size() const
{
  return m_size;
}
```

```
int width() const
{
  return m_size.width();
}
```

Właściwości można eksportować przez warstwę metadanych. Służy do tego makro Q_PROPERTY.

```
Q_PROPERTY(type name
  (READ getFunction [WRITE setFunction] |
  MEMBER memberName [(READ getFunction | WRITE setFunction)])
  [RESET resetFunction]
  [NOTIFY notifySignal]
  [REVISION int]
  [DESIGNABLE bool]
  [SCRIPTABLE bool]
  [STORED bool]
  [USER bool]
  [CONSTANT]
  [FINAL]
  [REQUIRED])
```

Tworzenie właściwości.

```
class Counter : public QObject
{
  Q_OBJECT
  Q_PROPERTY(int value READ value WRITE setValue)

  public:
    Counter(QObject *parent = 0);
    int value() const;
    void setValue(const int);

  private:
    int m_value;
};
```

Konstruktor, metody akcesora i modyfikatora.

```
Counter::Counter(QObject *parent)
: QObject(parent)
{
  m_value = 0;
}
```

```
int Counter::value() const
{
  return m_value;
}
```

```
void Counter::setValue(const int newValue)
{
  m_value = newValue;
}
```

Wykorzystanie właściwości

Za pomocą metod obiektu (bezpośrednio).

```
int myCounterValue = myCounter->value();
```

```
myCounter->setValue(69);
```

Dostęp przez warstwę metadanych.

```
int myCounterValue = myCounter->property("value").toInt();
```

```
myCounter->setProperty("value", 69);
```

Odpytywanie o właściwości w czasie wykonania.

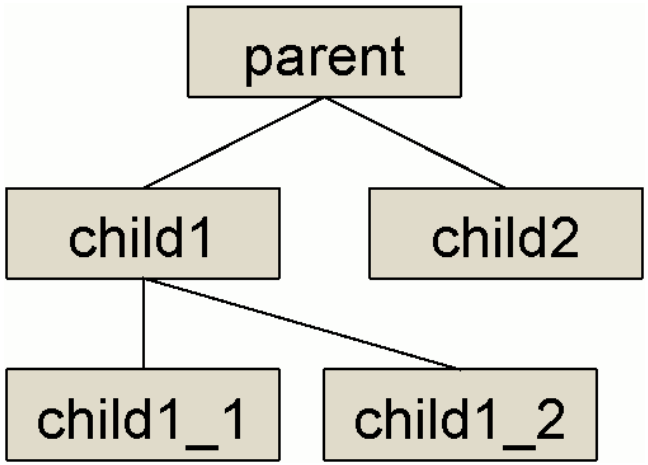
```
QObject *object = ...

const QMetaObject *metaobject = object->metaObject();
int count = metaobject->propertyCount();

for(int i = 0; i < count; ++i)
{
  QMetaProperty metaproperty = metaobject->property(i);
  const char *name = metaproperty.name();
  ...
}
```

Zarządzanie pamięcią

Obiekty `QObject` mogą posiadać klasy bazowe (rodziców) i klasy potomne (dzieci).

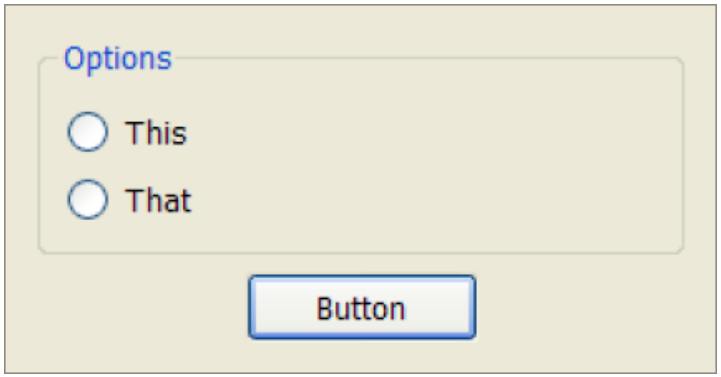
Usuwany **obiekt bazowy** usuwa wszystkie **obiekty potomne**.

```
QObject *parent = new QObject();
QObject *child1 = new QObject(parent);
QObject *child2 = new QObject(parent);
QObject *child1_1 = new QObject(child1);
QObject *child1_2 = new QObject(child1);

delete parent;
```



Te same zasady dotyczą obiektów dziedziczących z `QObject`.

```
QDialog *parent = new QDialog();
QGroupBox *box = new QGroupBox(parent);
QPushButton *button = new QPushButton(parent);
QRadioButton *option1 = new QRadioButton(box);
QRadioButton *option2 = new QRadioButton(box);

delete parent;
```



Wskaźnika `this` można użyć jako odwołanie do rodzica.

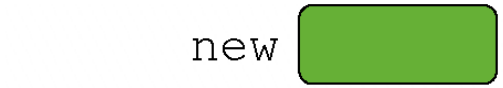
W C++ `this` odnosi się w niestatycznych metodach klasy do obiektu na rzecz którego dana metoda została wywołana (poniżej przykładowy konstruktor klasy `Dialog`).

```
Dialog::Dialog(QWidget *parent)
: QDialog(parent)
{
  QGroupBox *box = new QGroupBox(this);
  QPushButton *button = new QPushButton(this);
  QRadioButton *option1 = new QRadioButton(box);
  QRadioButton *option2 = new QRadioButton(box);
}
```

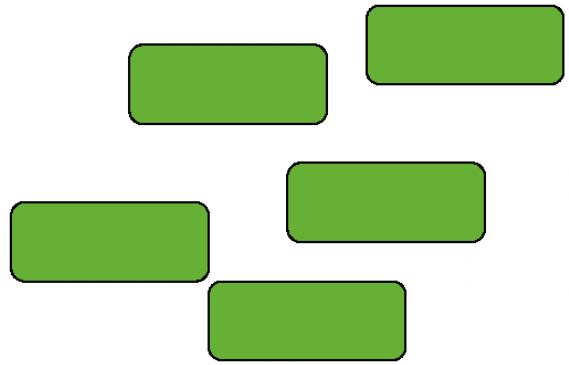
Jeśli zaalokujemy rodzica na **stosie**, to zostanie on automatycznie usunięty (zmienna automatyczna) wraz zw swoimi potomkami.

```
void showDialog()
{
  Dialog myDialog;

  if(myDialog.exec() == QDialog::Accepted)
  {
    ...
  }
} // myDialog jest w tym momencie usuwany
```

- Używając słowa kluczowego new pamięć alokowana jest na **stercie**.
- Obiekty alokowane na stercie żyją tak długo jak są potrzebne.
- Odpowiedni obszar sterty musi być zwolniony za pomocą `delete` żeby zapobiec wyciekom pamięci.

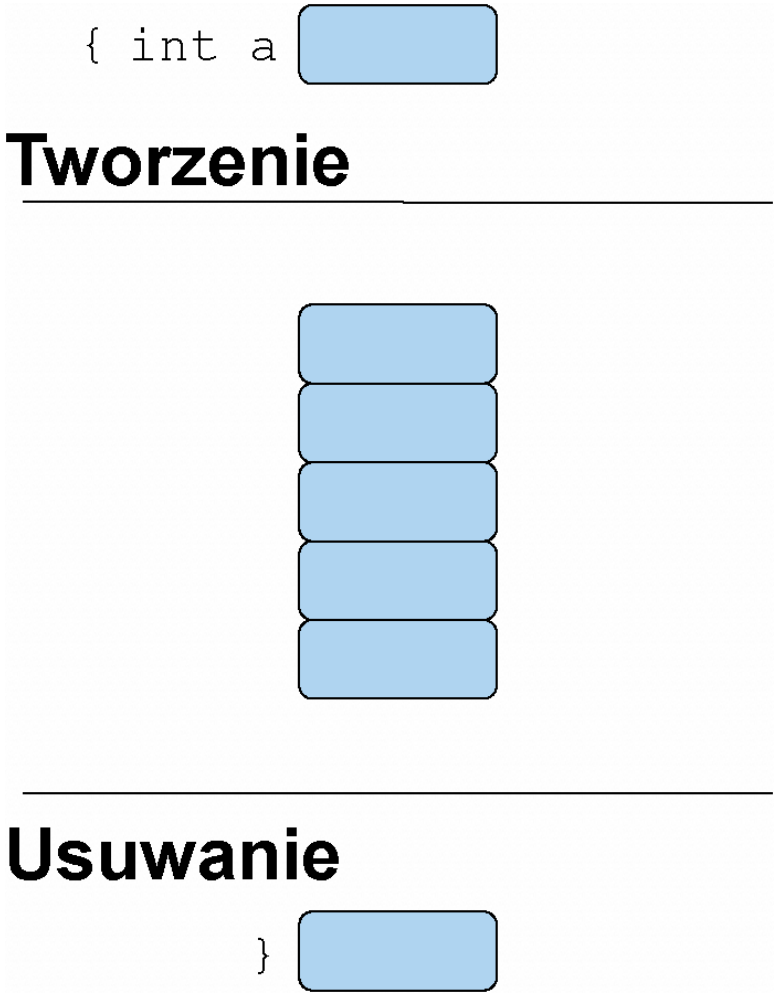


Tworzenie

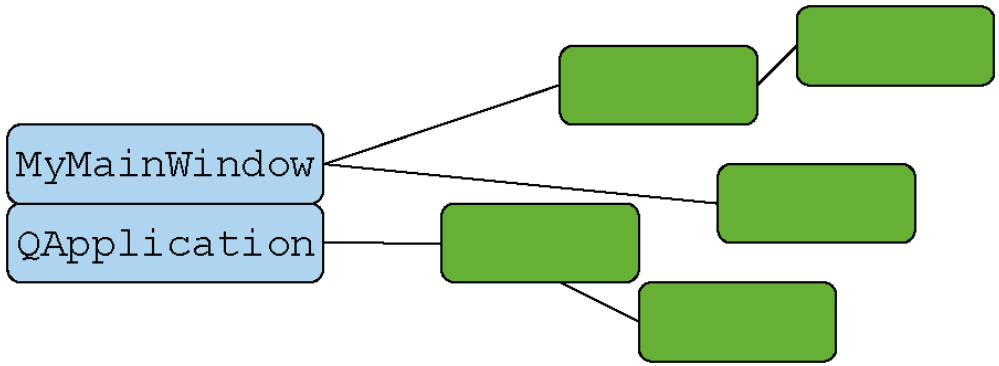


Usuwanie

- Zmienne lokalne alokowane są na **stosie**.
- Zmienne alokowane na stosie są automatycznie usuwane gdy utracą *widoczność*.
- Podobna sytuacja ma miejsce w przypadku obiektów.



By zrealizować automatyczne zarządzanie pamięcią, wystarczy by węzły **rodziców** były alokowane na **stosie**.




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

  MainWindow w;
  w.show();

  return a.exec();
}
```

```
MainWindow::MainWindow(QWidget
    ↪ *parent)
: QMainWindow(parent)
{
  QGroupBox *box = new
      ↪ QGroupBox(this);
  QPushButton *button = new
      ↪ QPushButton(this);
  QRadioButton *option1 = new
      ↪ QRadioButton(box);
  ...
}
```

Konstruktory

Większość klas pochodnych od `QObject` pozwala na ustawienie rodzica z domyślną wartością `0` (`null`).

```
Counter(QObject *parent = 0);
```

```
Counter::Counter(QObject *parent): QObject(parent)
```

Rodzicem `QWidget` jest inny obiekt takiego typu.

W przypadku obiektów większości kontrolek zwykle dostępnych jest wiele różnych konstruktorów (w tym jeden pozwalający zdefiniować wyłącznie rodzica).

```
QPushButton(QWidget *parent=0);
QPushButton(const QString &text, QWidget *parent=0);
QPushButton(const QIcon &icon, const QString &text, QWidget
    ↪ *parent=0);
```

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

Sygnały i sloty

Mechanizm **sygnałów** i **slotów** pełni kluczową rolę w bibliotece Qt, a także odróżnia ją od innych podobnych frameworków, które najczęściej wykorzystują funkcje wywoływane zwrotnie (*callback function*).

Ceną prostoty stosowania sygnałów i slotów jest niewielkie rozszerzenie składni oraz konieczność użycia niestandardowych narzędzi.

Sygnałem (*signal*) w bibliotece Qt nazywamy taką metodę klasy, która uruchamiana jest w wyniku wystąpienia określonego zdarzenia w trakcie działania programu.

Przykładem takiego zdarzenia jest przyciśnięcie przycisku (`QPushButton`), które powoduje wyemitowanie sygnału `clicked()`.

Slotem (*slot*) jest także metodą klasy, która jest wywoływana wtedy, gdy zostanie wyemitowany połączony z tym slotem sygnał.

Pojedynczy sygnał może być podłączony do dowolnej ilości slotów. Analogicznie wiele sygnałów może być podłączonych do jednego slotu.

Połączenie sygnału z gniazdem tworzy statyczna metoda `QObject::connect()`.

Sygnały i sloty mogą być definiowane wyłącznie w klasach dziedziczących pośrednio lub bezpośrednio po klasie `QObject`. Każdy obiekt dziedziczący po klasie `QObject` posiada metaobiekt klasy `QMetaObject`.

Metaobiekty przechowują informacje na tematy klasy, które są niezbędne do prawidłowego funkcjonowania mechanizmu sygnałów i slotów. Są to między innymi nazwa klasy oraz lista sygnałów i slotów zdefiniowanych w tej klasie.

Metaobiekt zwracany jest przez metodę `QObject::metaObject()`, przy czym wszystkie instancje danej klasy zwracają ten sam obiekt, więc ilość zużywanej pamięci zależy tylko od liczby klas.

Definicje slotów standardowo umieszcza się w publicznej części klasy, a po słowie zarezerwowanym `public` musi wystąpić makro `slots`.

Metodę reprezentującą slot implementujemy w tradycyjny sposób. Nie ma także żadnych przeszkód, aby tak zdefiniowane metody wywoływać bez pośrednictwa mechanizmu sygnałów i slotów.

Wartością makra `slots` jest pusty ciąg znaków, jest więc ono obojętne dla kompilatora C++.

Definicje sygnałów umieszcza się w sekcji `signals` klasy.

Dodatkowym ograniczeniem jest to, że metoda będąca sygnałem nie może zwracać żadnej wartości. Metody reprezentującej sygnał, w przeciwieństwie do gniazd, nie implementujemy. Implementacje sygnałów zostaną wygenerowane przez kompilator **moc**.

Wartością makra `signals` jest słowo zarezerwowane `protected`.

Z sygnałami związane jest jeszcze jedno makro, czyli `emit`. Jest ono używane przy wywołaniu metody-sygnału.

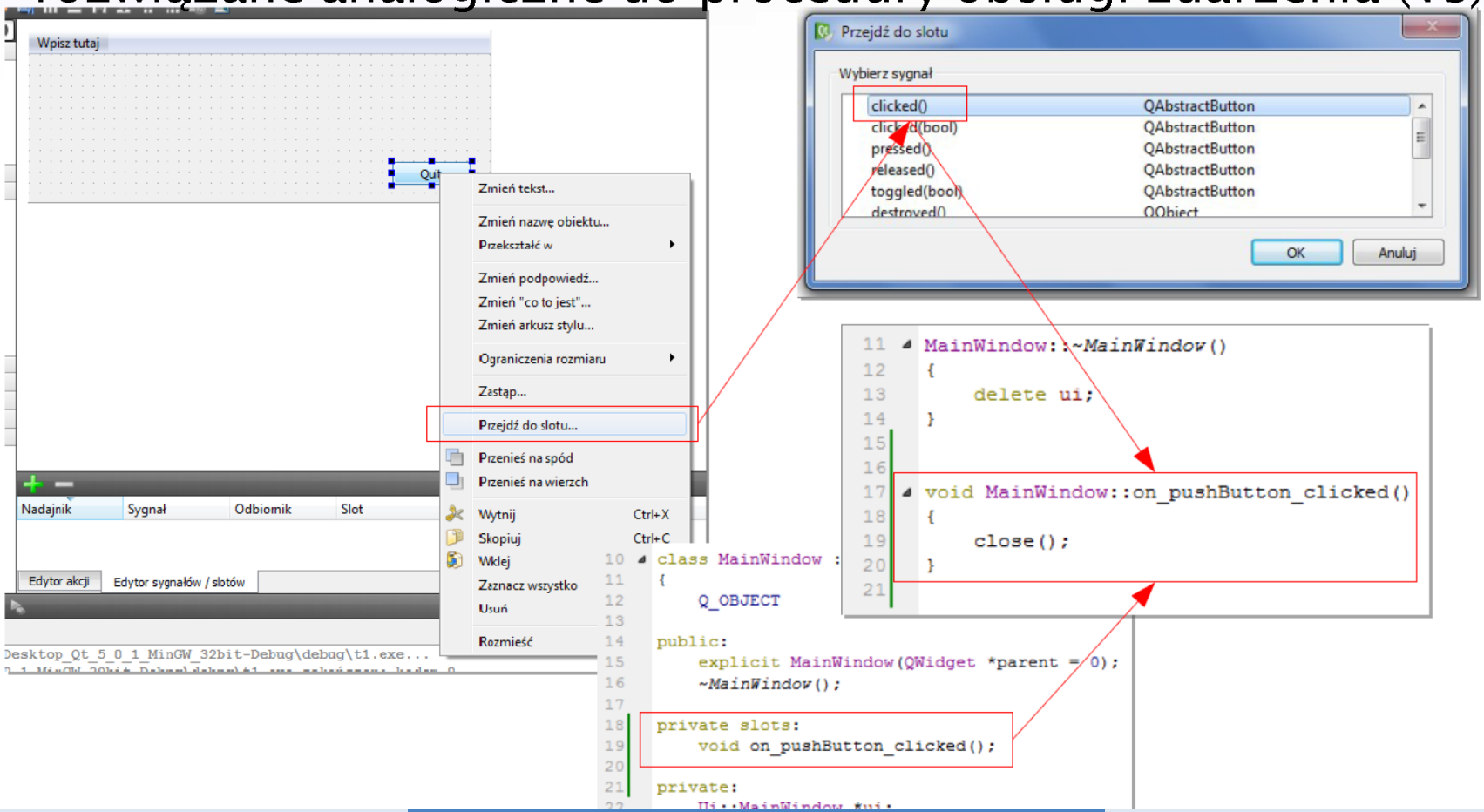
Oznacza to, że mamy do czynienia z wywołaniem sygnału, a implementacja metody zostanie wygenerowana przez kompilator **moc**.

Wartością makra `emit` jest pusty ciąg znaków.

Okno z przyciskiem Primo Secondo Terzo

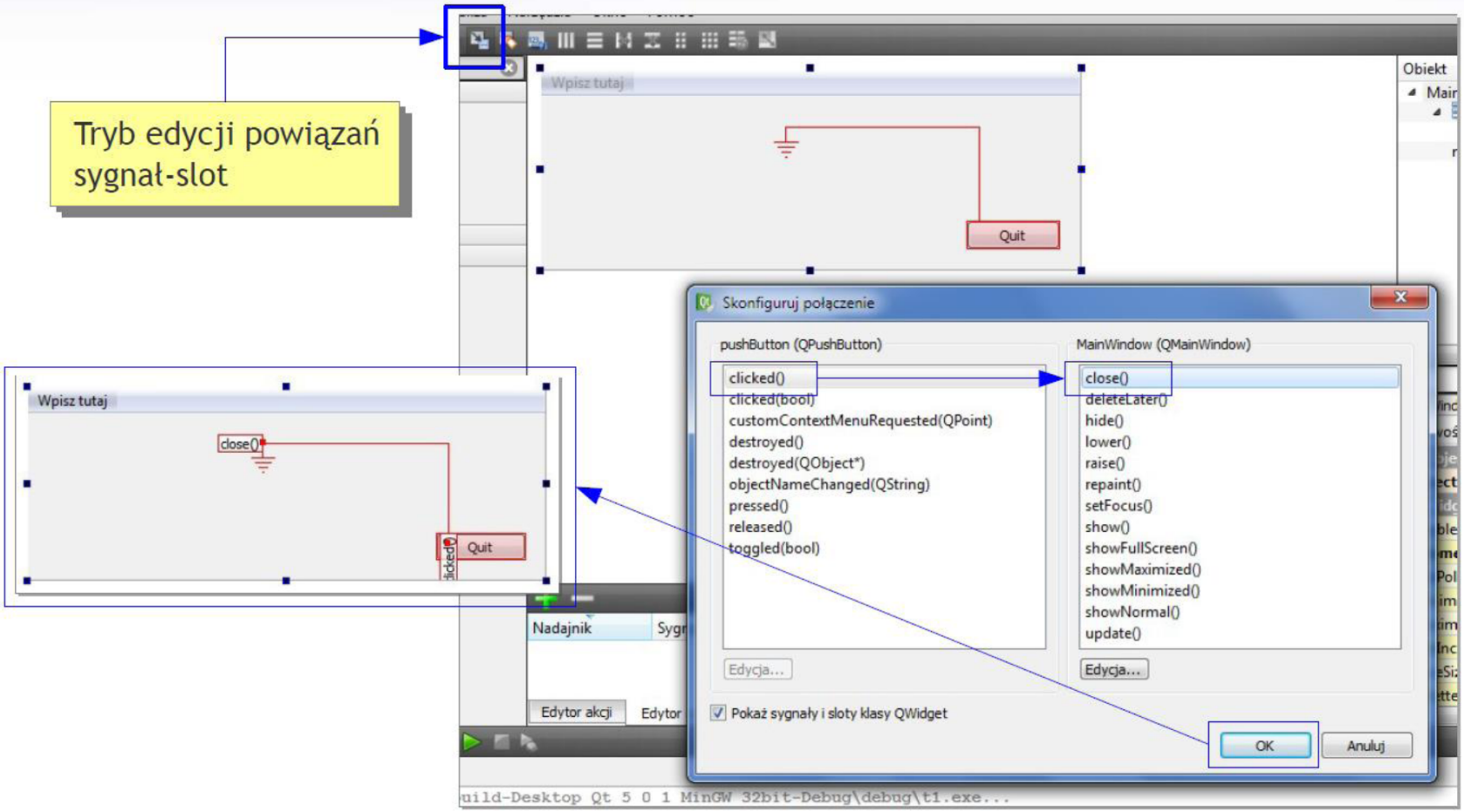
Sygnały i sloty – okno z przyciskiem zamykania wer. 1

- Wykorzystanie *slotu* jako *funkcji przypisanej do sygnału* — rozwiązane analogiczne do procedury obsługi zdarzenia (VS)



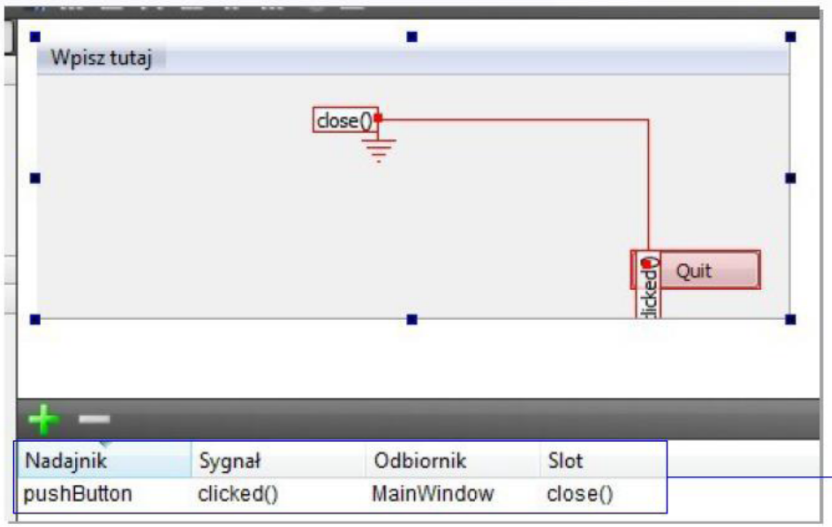
Sygnały i sloty – okno z przyciskiem zamykania wer. 2

- Wykorzystanie powiązania *sygnał–slot* — rozwiązane z wykorzystaniem QtDesigner'a.



Sygnały i sloty – okno z przyciskiem zamykania wer. 2

- Informacje o powiązaniu *sygnał-slot* zapisywane są do pliku definicji okna głównego *mainwindow.ui*.



mainwindow.ui*

```
    </rect>
   </property>
  </widget>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections>
  <connection>
   <sender>pushButton</sender>
   <signal>clicked()</signal>
   <receiver>MainWindow</receiver>
   <slot>close()</slot>
   <hints>
    <hint type="sourcelabel">
     <x>350</x>
     <y>130</y>
    </hint>
    <hint type="destinationlabel">
     <x>196</x>
     <y>46</y>
    </hint>
   </hints>
  </connection>
```

Sygnały i sloty – okno z przyciskiem zamykania wer. 3

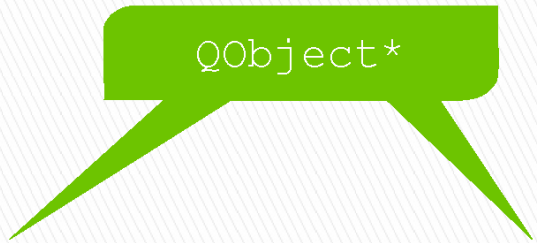
- Wykorzystanie powiązania *sygnał-slot* — wykorzystaniem makra *connect* w kodzie programu.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT( close()));
}

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

Sygnały i sloty a wywołania zwrotne (callbacks)

- Wywołanie zwrotne to wskaźnik do funkcji, która jest wywoływana kiedy nastąpi dane zdarzenie. Każda funkcja może być przyporządkowania do wywołania zwrotnego.
  - Brak bezpieczeństwa typów (type-safety)
  - Zawsze działa jako bezpośrednie wywołanie.
- Mechanizm slotów i sygnałów jest bardziej dynamiczny:
  - Bardziej ogólny mechanizm
  - Dokonywane jest sprawdzenie typów
  - Łatwiejsza implementacja wielowątkowości

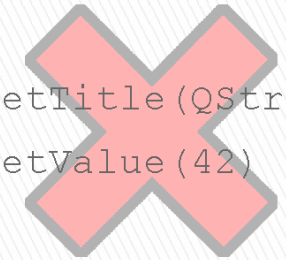
Tworzenie połączenia




```
QObject::connect( src, SIGNAL( signature ), dest, SLOT( signature ) );
```

```
<function name> ( <arg type>... )
```

Sygnatura składa się z nazwy funkcji i typów argumentów. Nazwy zmiennych bądź ich wartości nie są dozwolone.



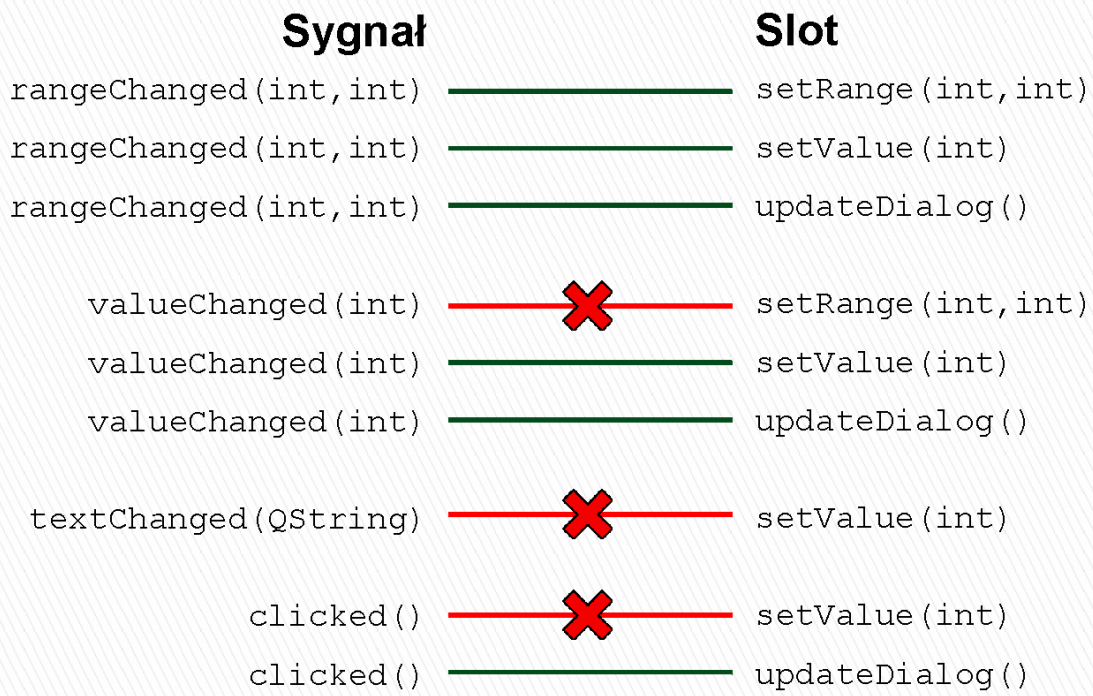
```
setTitle(QString text)
setValue(42)
```

```
setItem(ItemClass)
```

```
clicked()
toggled(bool)
setText(QString)
textChanged(QString)
rangeChanged(int,int)
```

Tworzenie połączenia

- Qt może ignorować argumenty, ale nie może ich tworzyć z niczego



Synchronizowanie wartości

- Połączenie dwustronne

```
connect(dial1, SIGNAL(valueChanged(int)), dial2, SLOT(setValue(int)));
```

```
connect(dial2, SIGNAL(valueChanged(int)), dial1, SLOT(setValue(int)));
```

- Stop pętli nieskończonej – sygnał nie jest emitowany, jeżeli nie nastąpiła zmiana wartości

```
void QDial::setValue(int v)
{
    if(v==m_value)
        return;
    ...
```

Nie należy o tym zapominać!

Własne sloty i sygnały

```
class AngleObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(qreal angle READ angle WRITE setAngle NOTIFY angleChanged)

public:
    AngleObject(qreal angle, QObject *parent = 0);
    qreal angle() const;

public slots:
    void setAngle(qreal);

signals:
    void angleChanged(qreal);

private:
    qreal m_angle;
};
```

Jaki sygnał wysłać jak się zmieni?

Modyfikatory to często sloty.

Sygnały to często akcesory

Implementacja modyfikatora

```
void AngleObject::setAngle(qreal angle)
{
    if(m_angle == angle)
        return;

    m_angle = angle;
    emit angleChanged(m_angle);
}
```

Zabezpieczenie przez pętlą nieskończoną. **Nie należy zapomnieć!**

Uaktualnij stan i wyślij odpowiedni sygnał

Sygnały najczęściej definiowane są w sekcji chronionej (protected) dlatego można je wysyłać z klas potomnych

Dodatek: Przesyłanie wartości

- Często istnieje potrzeba przekazania wartości w instrukcji connect

```
connect(key, SIGNAL(clicked()), this, SLOT(keyPressed(1)));
```

- Przykładowo implementując klawiaturę:

- Nie jest to prawidłowy zapis

Dodatek: Przesyłanie wartości

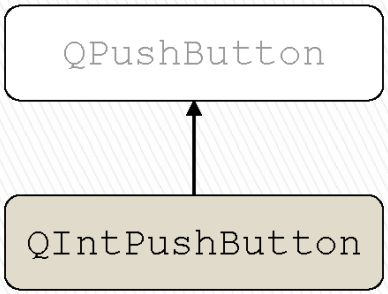
- Rozwiązanie #1: wiele slotów

connections

```
{
    ...

public slots:
    void key1Pressed();
    void key2Pressed();
    void key3Pressed();
    void key4Pressed();
    void key5Pressed();
    void key6Pressed();
    void key7Pressed();
    void key8Pressed();
    void key9Pressed();
    void key0Pressed();

    ...
}
```

Dodatek: Przesyłanie wartości

- Rozwiązanie #2: własna implementacja przycisku




```
{
    ...

signals:
    void clicked(int);

    ...
}
```

```
{
    QIntPushButton *b;

    b=new QIntPushButton(1);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    b=new QIntPushButton(2);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    b=new QIntPushButton(3);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    ...
}
```

Ocena rozwiązań

- #1: wiele slotów
  - Wiele slotów zawierających niemalże ten sam kod
  - Kod trudny w utrzymaniu
  - Trudne rozszerzenie funkcjonalności (ciągle nowy slot)
- #2: własna implementacja przycisku
  - Nowa wyspecjalizowana klasa
  - Trudne rozszerzenie funkcjonalności

Mapowanie sygnałów

- Klasa `QSignalMapper` rozwiązuje ten problem
  - Przyporządkuje wartość do każdego emitera
  - Jest warstwą pośredniczącą

```
{
    QSignalMapper *m = new QSignalMapper(this);
    QPushButton *b;

    b=new QPushButton("1");
    connect(b, SIGNAL(clicked()),
            m, SLOT(map()));
    m->setMapping(b, 1);

    ...

    connect(m, SIGNAL(mapped(int)), this, SLOT(keyPressed(int)));
}
```

Stwórz QSignalMapper

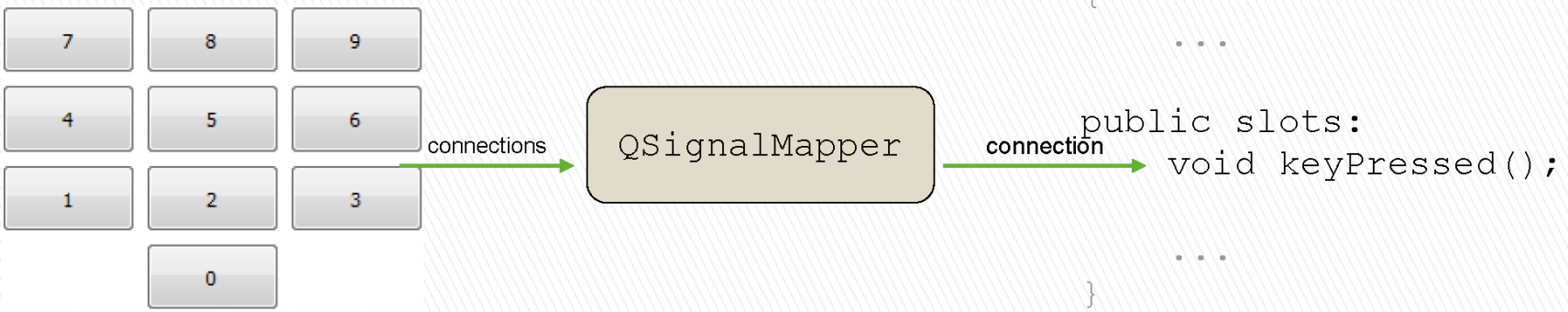
Połącz przyciski do warstwy pośredniczącej

Przyporządkuj emiterowi wartość

Połącz pośrednika do slotu

Mapowanie sygnałów

- Każdy przycisk ma przyporządkowaną wartość liczbową



- Warstwa pośrednicząca (QSignalMaper) emituje sygnał `mapped(int)`, który przekazuje przyporządkowaną danemu przyciskowi wartość.

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

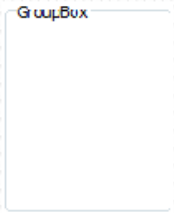
Komponenty interfejsu użytkownika

- Interfejs użytkownika składa się z pojedynczych widgetów

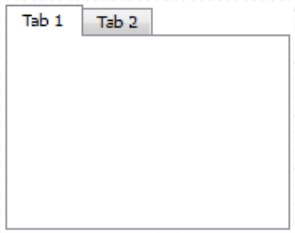
- 46 widgetów dostępnych jest z poziomu QtDesigner.
- 59+ bezpośrednio dziedziczy z `QWidget`

Widget w widgecie

- Widgety tworzą hierarchie



QGroupBox



QTabWidget

- Kontenery pozwalają uporządkować warstwę wizualną...
- ...ale również mogą dostarczać pewne funkcjonalności (np.. `QRadioButton`)

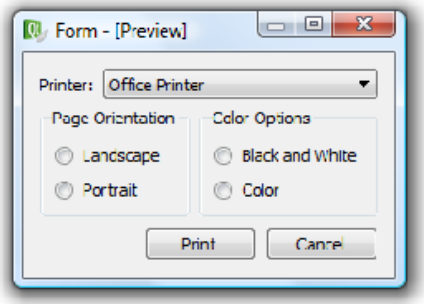
Czym jest widget?

- Zajmuje prostokątny obszar ekranu
- Odbiera zdarzenia z urządzeń wejściowych
- Emituje sygnały dla "znaczących" zmian

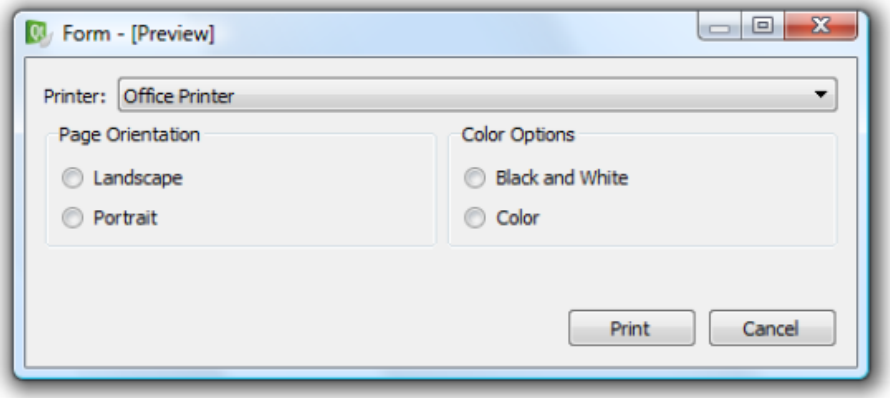
Wszystkie widgety:

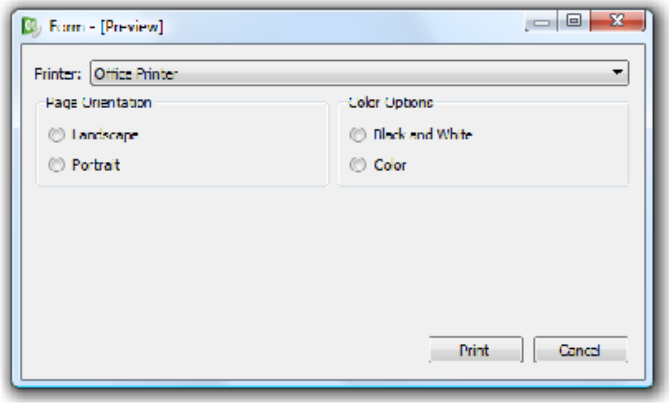
- Są zorganizowane w określoną hierarchię.
- Mogą zawierać inne widgety w sobie

Przykładowe okno dialogowe



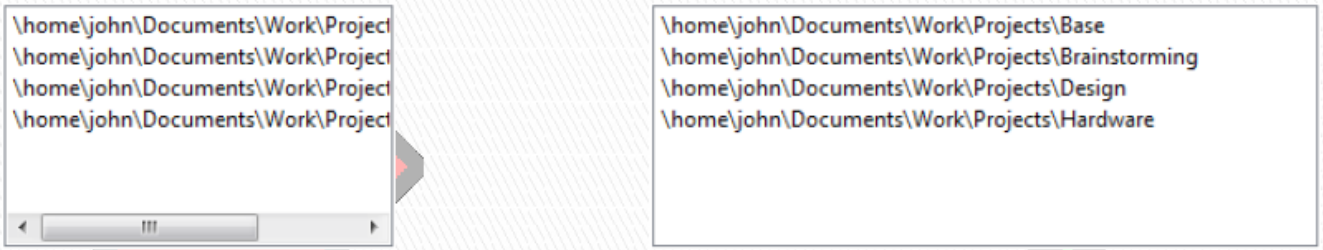
- Rozmieszczenie widgetów kontrolują szablony – by interfejs użytkownika był elastyczny





Dlaczego elastyczność jest dobra?

- Umożliwia widgetom dostosowanie się do treści

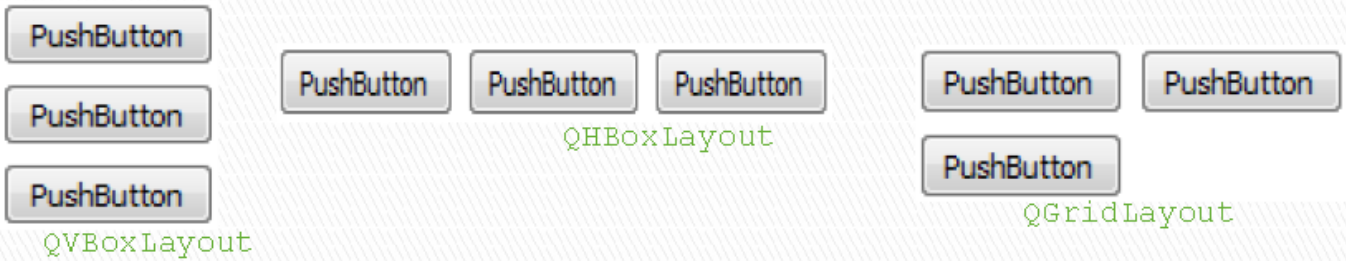


- Umożliwia widgetom dostosowanie się do tłumaczenia

- Umożliwia dostosowanie się do preferencji użytkownika

Szablony

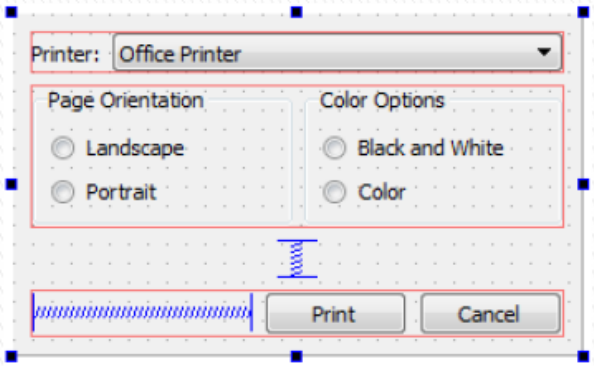
- Istnieje wiele różnych szablonów



- Widgety walczą ze sobą o pozycję i rozmiar
- Spacer może zostać zastosowany do wypełnienia pustki i rozciągania

Przykładowe okno dialogowe

- Okna dialogowe zbudowane są z wielu warstw szablonów i widgetów

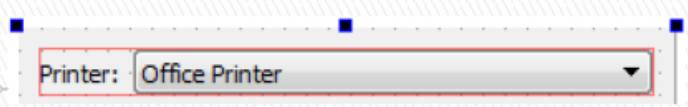


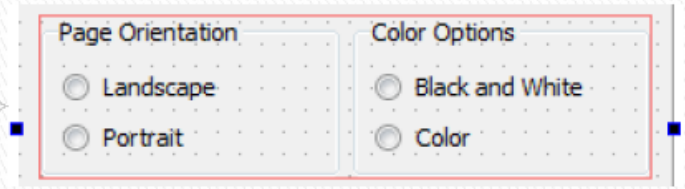
Object	Class
Form	QWidget
horizontalLayout	QHBoxLayout
label	QLabel
printerBox	QComboBox
horizontalLayout_2	QHBoxLayout
cancelButton	QPushButton
horizontalSpacer	Spacer
printButton	QPushButton
horizontalLayout_3	QHBoxLayout
groupBox	QGroupBox
landscapeButton	QRadioButton
portraitButton	QRadioButton
groupBox_2	QGroupBox
bwButton	QRadioButton
colorButton	QRadioButton
verticalSpacer	Spacer

Szablony nie są rodzicami dla widgetów którymi zarządzają.

Przykładowe okno dialogowe

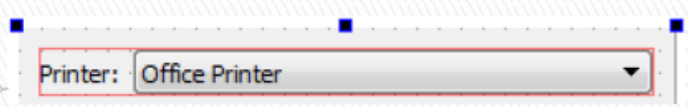
```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);

QHBoxLayout *groupLayout = new QHBoxLayout();
...
outerLayout->addLayout(groupLayout);

outerLayout->addSpacerItem(new QSpacerItem(...));

QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new
QPushButton("Cancel"));  outerLayout-
>addLayout(buttonLayout);
```



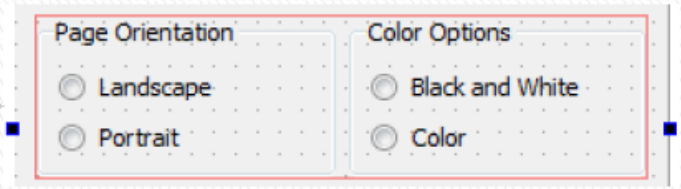


Przykładowe okno dialogowe

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);
```

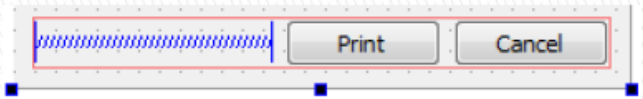



```
QHBoxLayout *groupLayout = new QHBoxLayout();
...
outerLayout->addLayout(groupLayout);
```



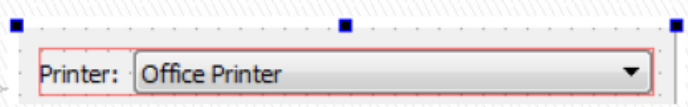

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new
QPushButton("Cancel"));  outerLayout-
>addLayout(buttonLayout);
```

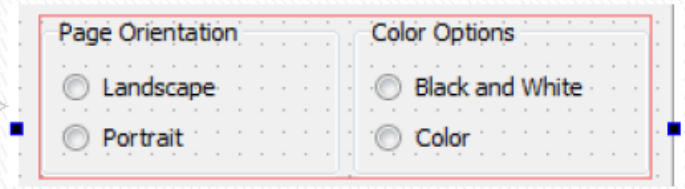


Przykładowe okno dialogowe

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);
```

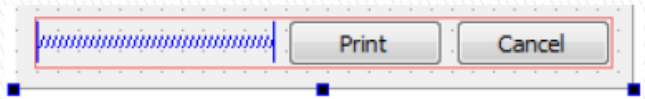



```
QHBoxLayout *groupLayout = new QHBoxLayout();

...

outerLayout->addLayout(groupLayout);
```



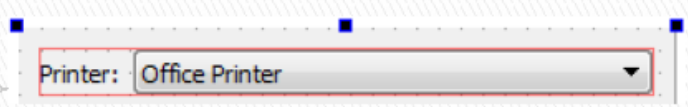

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

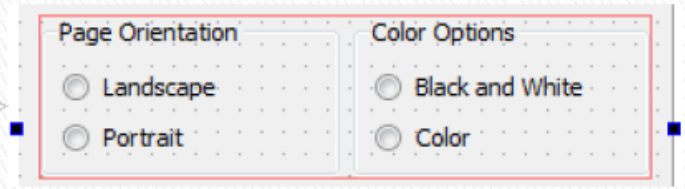


Przykładowe okno dialogowe

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);
```




```
QHBoxLayout *groupLayout = new QHBoxLayout();

...

outerLayout->addLayout(groupLayout);
```

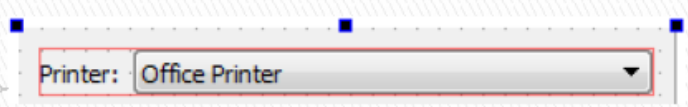



```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

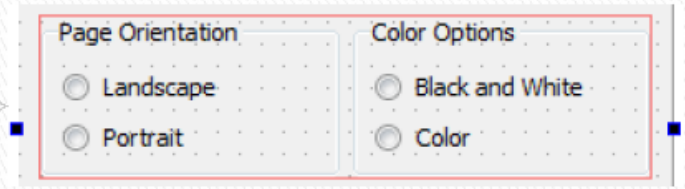
```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

Przykładowe okno dialogowe

```
QVBoxLayout *outerLayout = new QVBoxLayout(this);

QHBoxLayout *topLayout = new QHBoxLayout();
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
outerLayout->addLayout(topLayout);
```

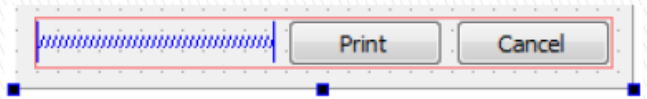



```
QHBoxLayout *groupLayout = new QHBoxLayout();

...

outerLayout->addLayout(groupLayout);
```



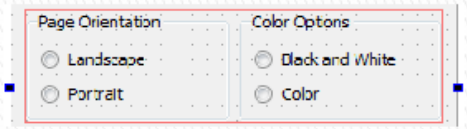

```
outerLayout->addSpacerItem(new QSpacerItem(...));
```

```
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);
```

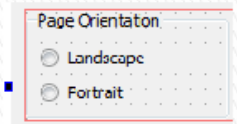


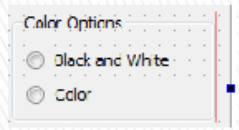
Przykładowe okno dialogowe

- Układ horyzontalny, zawiera dwa panele, mające układ wertykalny, zawierające pola jednokrotnego wyboru



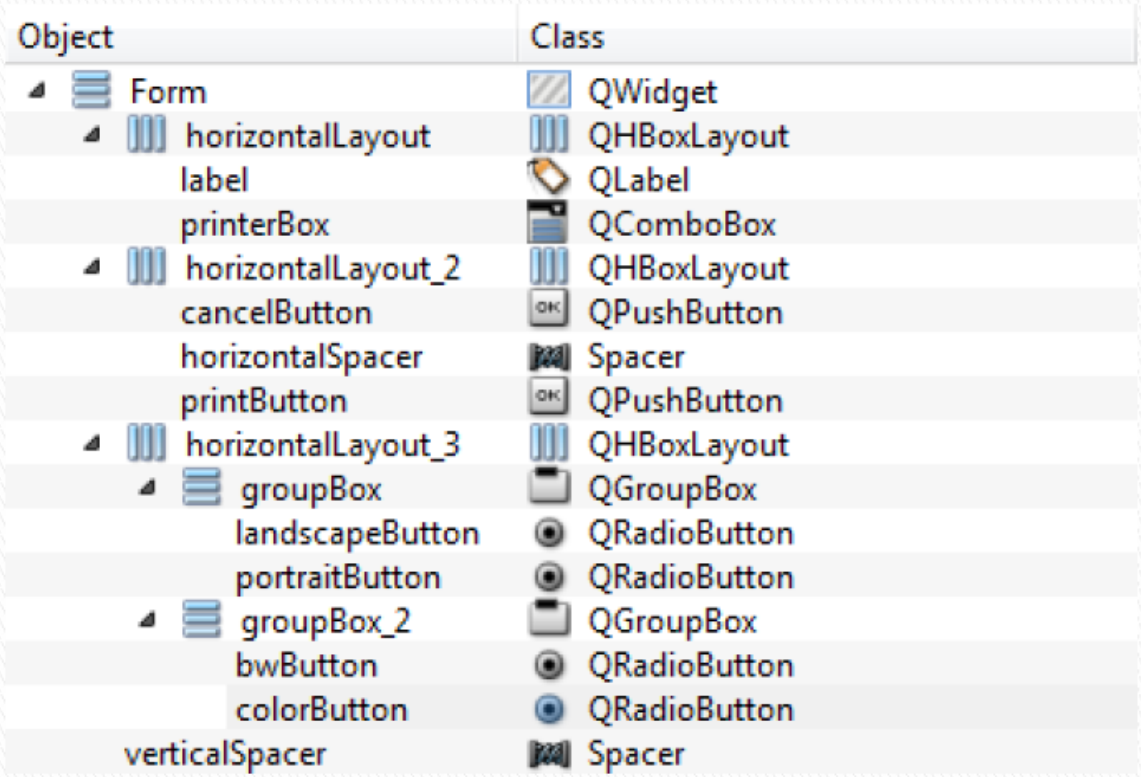

```
QHBoxLayout *groupLayout = new QHBoxLayout();

QGroupBox *orientationGroup = new QGroupBox();
QVBoxLayout *orientationLayout = new QVBoxLayout(orientationGroup);
orientationLayout->addWidget(new QRadioButton("Landscape"));
orientationLayout->addWidget(new QRadioButton("Portrait"));
groupLayout->addWidget(orientationGroup);

QGroupBox *colorGroup = new QGroupBox();
QVBoxLayout *colorLayout = new QVBoxLayout(colorGroup);
colorLayout->addWidget(new QRadioButton("Black and White"));
colorLayout->addWidget(new QRadioButton("Color"));
groupLayout->addWidget(colorGroup);
```





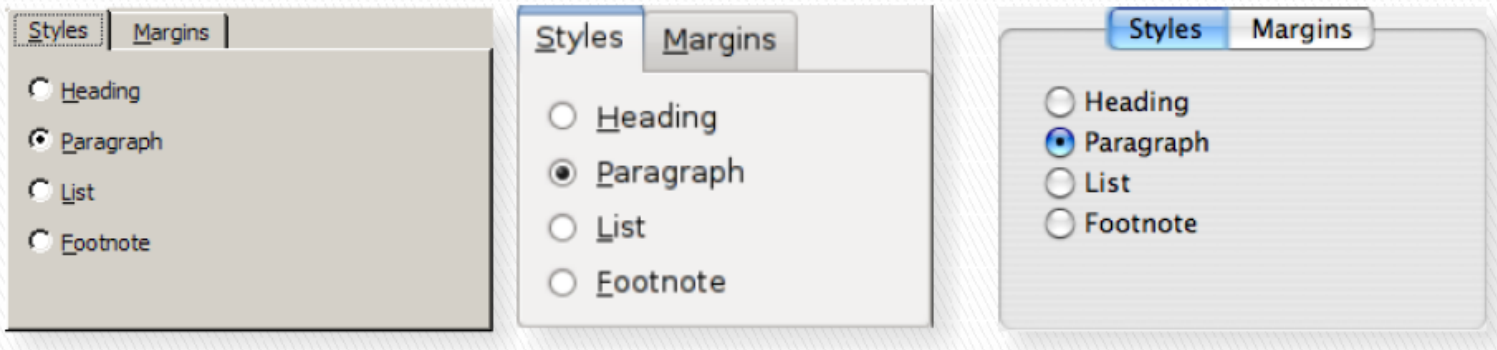
Przykładowe okno dialogowe

- Taki sam układ można uzyskać korzystając z QtDesigner



Style dla różnych platform

- Widgety są renderowane przy użyciu określonego stylu (specyficznego dla danej platformy) by zapewnić natywny wygląd.

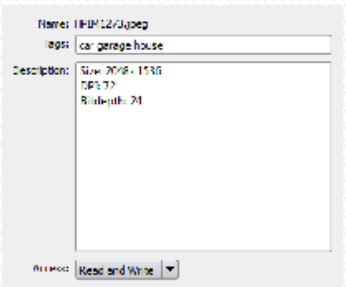


Problemy

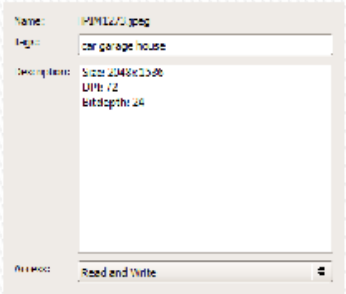
- Należy wziąć pod uwagę również inne rzeczy, projektując aplikację wieloplatformową, niż tylko zmiana stylu:
  - Układ okna
  - Umiejscowienie przycisków okna dialogowego
  - Wygląd typowych okien dialogowych

Problemy

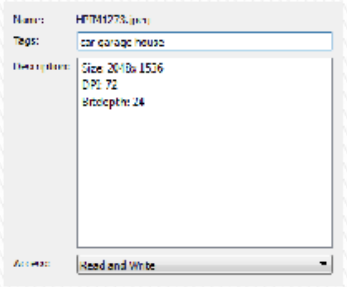
- Należy wziąć pod uwagę również inne rzeczy, projektując aplikację wieloplatformową, niż tylko zmiana stylu:
  - Układ okna
  - Umiejscowienie przycisków okna dialogowego
  - Wygląd typowych okien dialogowych



Plastique



ClearLooks

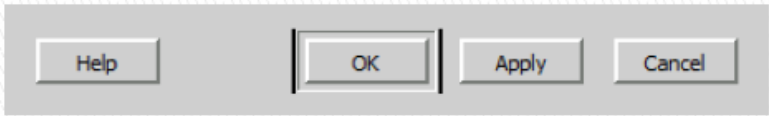


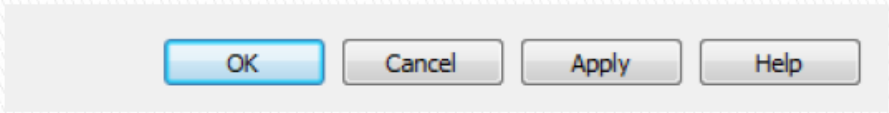
Windows

MacOS X

Problemy

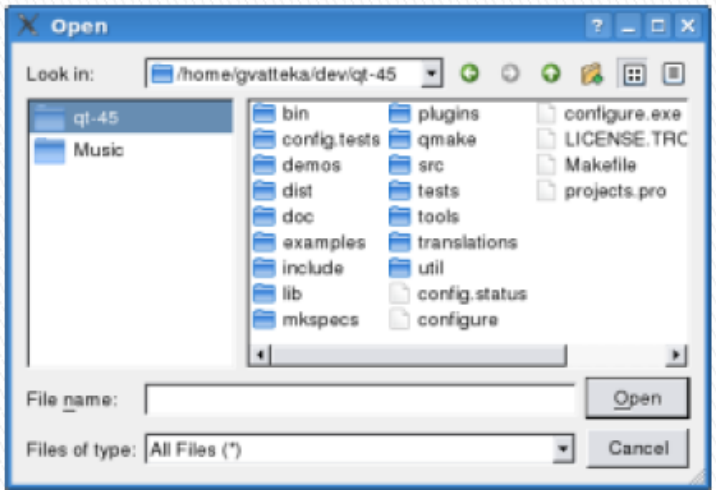
- Należy wziąć pod uwagę również inne rzeczy, projektując aplikację wieloplatformową, niż tylko zmiana stylu:
  - Układ okna
  - Umiejscowienie przycisków okna dialogowego
  - Wygląd typowych okien dialogowych

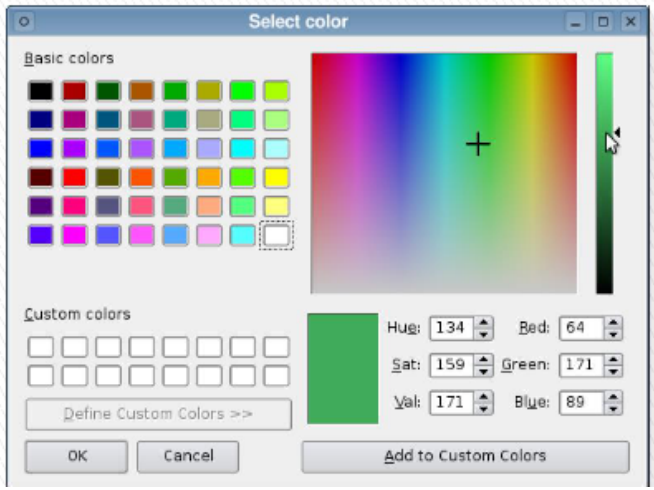




Problemy

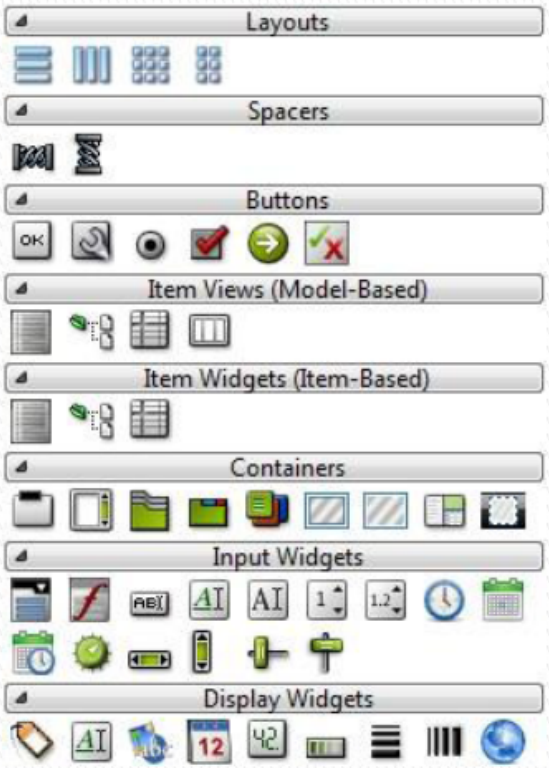
- Należy wziąć pod uwagę również inne rzeczy, projektując aplikację wieloplatformową, niż tylko zmiana stylu:
  - Układ okna
  - Umiejscowienie przycisków okna dialogowego
  - Wygląd typowych okien dialogowych





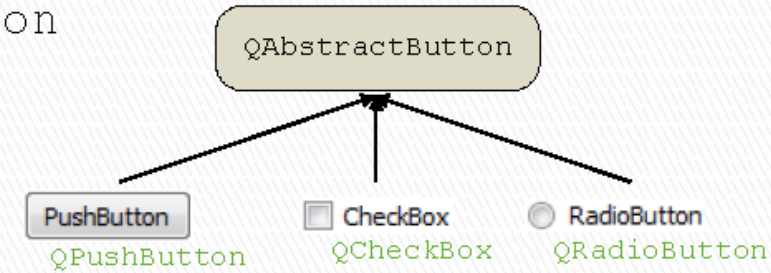
Standardowe widgety

- Qt zawiera liczne widgety, przydatne w typowych zastosowaniach programistycznych.
- Są one podzielone tematycznie w QtDesignerze.

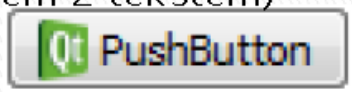


Przyciski

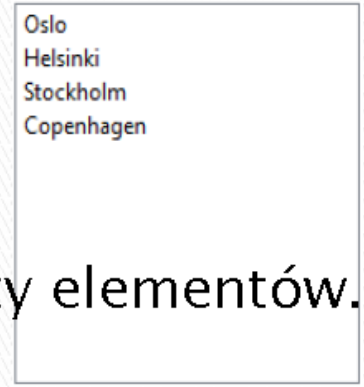
- **Wszystkie przyciski dziedziczą po klasie bazowej** `QAbstractButton`



- Sygnały
  - `clicked()` – emitowany podczas kliknięcia.
  - `toggled(bool)` – emitowany kiedy zmienia się zaznaczenie.
- Właściwości
  - `checkable` – prawda, jeżeli przycisk można zaznaczyć. Można zmienić zachowanie standardowego przycisku PushButton.
  - `checked` – prawda, jeżeli przycisk jest zaznaczony.
  - text – napis na przycisku.
  - icon – ikona przycisku (może być wyświetlana razem z tekstem)



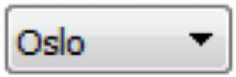
Listy



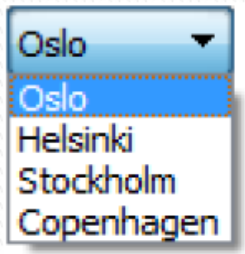
QListWidget

- `QListWidget` używany do wyświetlenia listy elementów.
- Dodawanie elementów
  - `addItem(QString)` – dodaje element na koniec listy
  - `insertItem(int row, QString)` – wstawia element pod zadanym indeksem
- Zaznaczanie
  - `selectedItems` – zwraca listę elementów `QListWidgetItem`
- Sygnały
  - `itemSelectionChanged` – emitted when the selection is changed

- `QComboBox` shows a list with a single selection in a more compact format.



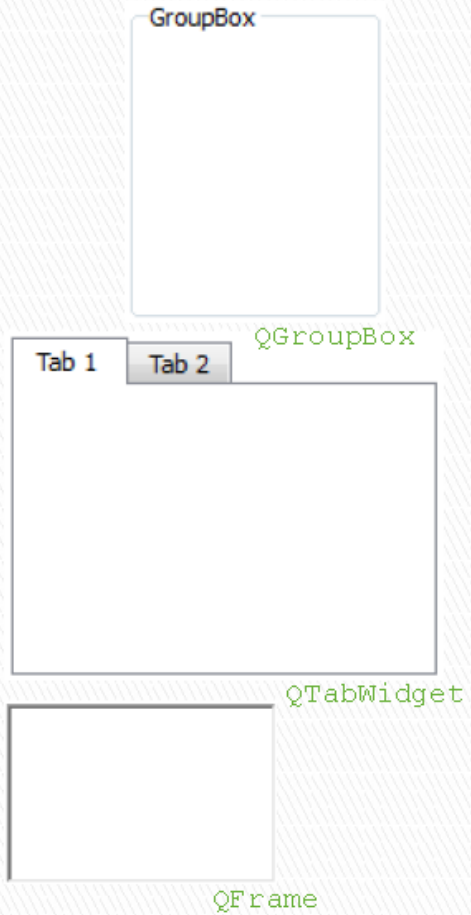
QComboBox



Kontenery

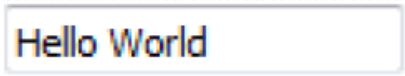
- Kontenery porządkują wizualną stronę interfejsu użytkownika
  - Są to w większości elementy pasywne
  - Zwykły `QWidget` może zostać użyty jako kontener
  - QtDesigner: Umieść widgety w kontenerze i zastosuj szablon.
  - Kod: Utwórz szablon dla kontenera i dodaj do niego komponenty

```
QGroupBox *box = new QGroupBox();
QVBoxLayout *layout = new QVBoxLayout(box);
layout->addWidget(...);
...
```



Komponenty do wprowadzania danych

- `QLineEdit` – jednowierszowe pole tekstowe
- Sygnały:
  - `textChanged(QString)` – emitowany gdy zmienia się tekst
  - `editingFinished()` – emitowany, gdy „opuszczamy” widget
  - `returnPressed()` – emitowany, gdy naciśnięto klawisz enter
- Właściwości
  - `text` – tekst widgetu
  - `maxLength` – maksymalna długość tekstu (w znakach)
  - `readOnly` – pole tylko do odczytu (kopiowanie nadal możliwe)



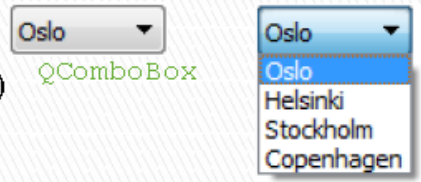
`QLineEdit`

Komponenty do wprowadzania danych

- Używaj `QTextEdit` lub `QPlainTextEdit` dla dużej ilości tekstu (wiele wierszy)
- Sygnały
  - `textChanged()` – emitowany, gdy zmienia się tekst
- Właściwości
  - `plainText` – tekst niesformatowany
  - `html` – tekst sformatowany (ze znacznikami html)
  - `readOnly` – pole tylko do odczytu

`QTextEdit`

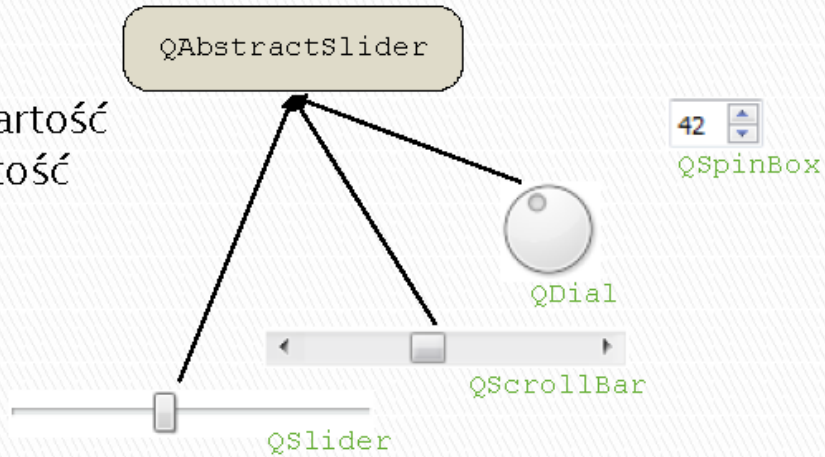
- `QComboBox` – **można edytować zawartość (właściwość** `editable`**)**
- Sygnały
  - `editTextChanged(QString)` – emitowany podczas zmiany tekstu
- Właściwości
  - `currentText` – aktualnie wybrany tekst
  - `currentIndex` – indeks wybranego elementu (od zera)



`QComboBox`

Komponenty do wprowadzania danych

- Spory wybór komponentów do wprowadzania danych całkowitych
- Podobny wybór jest dla danych rzeczywistych, czasu i daty
- Sygnały:
  - `valueChanged(int)` – emitowane, gdy zmienia się wartość
- Właściwości:
  - `value` – aktualna wartość
  - `maximum` – maksymalna wartość
  - `minimum` – minimalna wartość



Komponenty do wyświetlania danych

- `QLabel` wyświetla tekst lub obrazek
- Właściwości:
  - `text` – tekst dla komponentu
  - `pixmap` – obraz do wyświetlenia

`QLabel`

`QLabel`

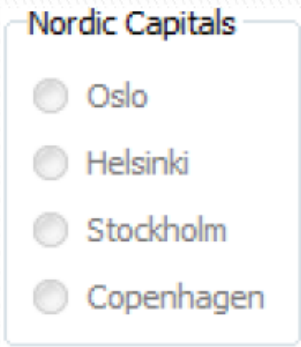
- `QLCDNumber` dedykowany do wyświetlenie danych całkowich
- Właściwości:
  - `intValue` – prezentowana wartość (ustawiana za pomocą `display(int)`)


`QLCDNumber`

Wspólne właściwości widgetów

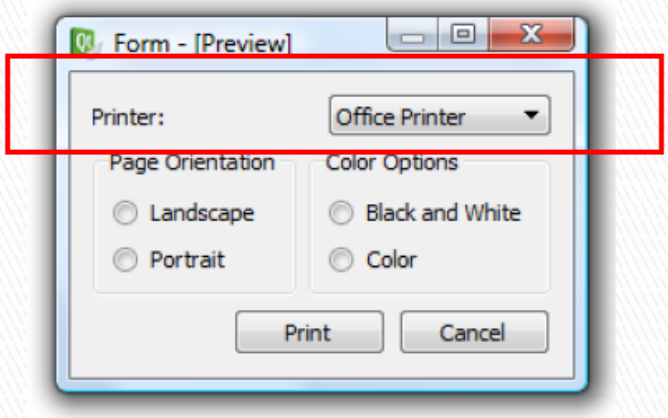
- Wszystkie widgety posiadają zestaw wspólnych właściwości, ponieważ dziedziczą z klasy bazowej `QWidget`
- `enabled` – czy komponent jest aktywny

- `visible` – czy jest widoczny (ustawiane za pomocą metod `show()` oraz `hide()`)

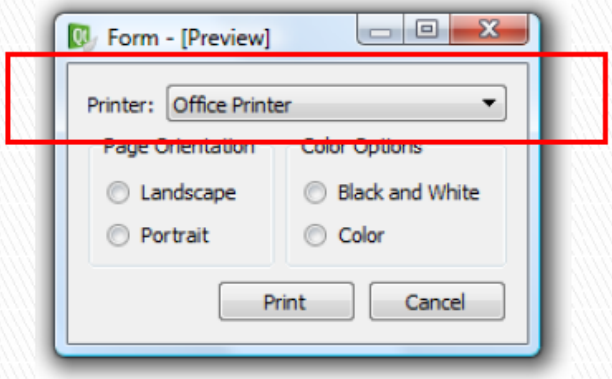


- Oczywiście, te właściwości mają wpływ również na komponenty potomne.

Polityka rozmiaru




```
printerList->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed)
```



Polityka rozmiaru

- Ustawienia wielkości widgetu są łączone z polityką rozmiaru w następujący sposób:
  - `Fixed` – zajmij tyle miejsca ile określono (w polu sizeHint)
  - `Minimum` – określa najmniejszy możliwy rozmiar
  - `Maximum` – określa maksymalny możliwy rozmiar
  - `Preferred` – dany rozmiar widgetu jest preferowany, ale nie jest to wymagane
  - `Expanding` – podobnie jak `preferred`, ale z nastawieniem na rozszerzanie
  - `MinimumExpanding` – jak dla `minimum`, ale z nastawieniem na rozszerzanie
  - `Ignored` – widget dostaje tyle przestrzeni, ile to tylko możliwe

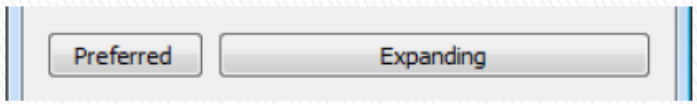
Polityka rozmiaru

- *Co w przypadku zmiany obszaru roboczego?*
  - `Fixed` – rozmiar komponentu nie zmienia się
  - `Minimum` – może być większy (ale nie mniejszy niż ustalona wartość)
  - `Maximum` – może się zmniejszyć (ale nie może być większy niż ustalona wartość)
  - `Preferred` – może się zarówno **zmniejszyć** *jak i* **zwiększyć** *(ale najlepiej, jak utrzyma zadany rozmiar)*
  - `Expanding` – może się zarówno **zmniejszyć** *jak i* **zwiększyć**, chce się **zwiększyć**
  - `MinimumExpanding` – może się **zwiększyć**, *chce się* ***rozszerzać***
  - `Ignored` – może się **zwiększyć** *lub* **zmniejszyć**

Co się stanie jeśli?

- Dwa komponenty w polityce preffered ustawimy obok siebie?

- Jeden preferred, jeden expanding



- Dwa expanding obok siebie

- Gdy nie ma wystarczająco dużo miejsca (fixed)

Jeszcze trochę o rozmiarach

- Rozmiary mogą być ponadto kontrolowane, ustalając najmniejszy i największy możliwy rozmiar.

- `maximumSize` – największy rozmiar
- `minimumSize` – najmniejszy rozmiar

```
ui->pushButton->setMinimumSize(100, 150);
ui->pushButton->setMaximumHeight(250);
```

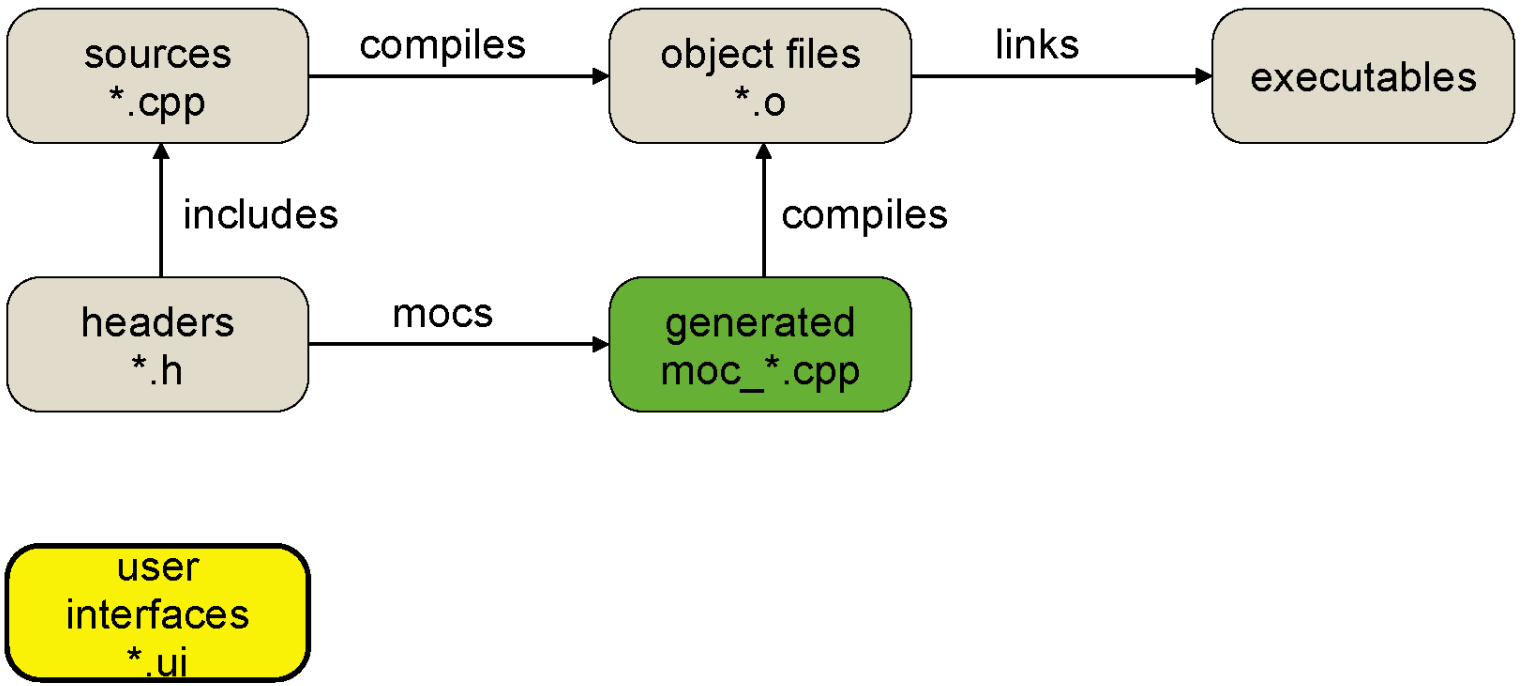
1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

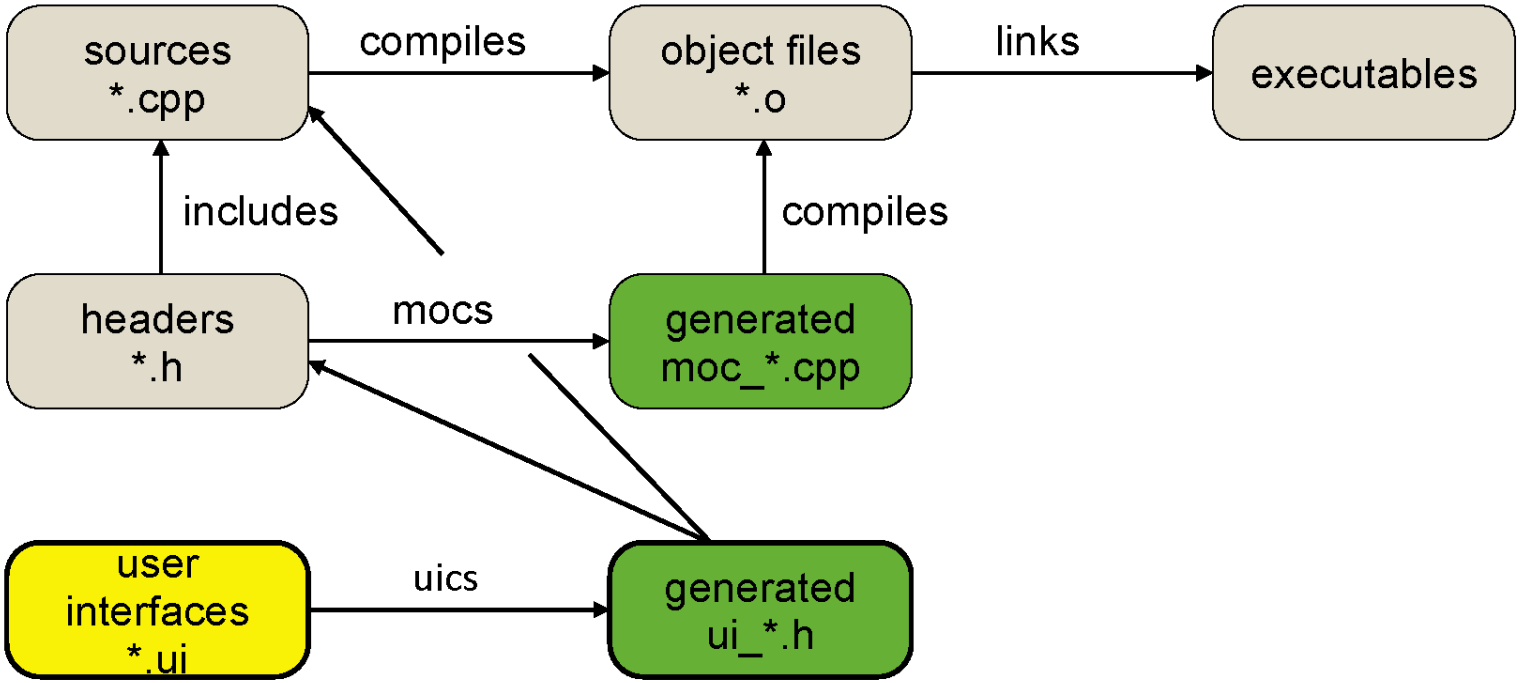
Qt Designer

Narzędzie **Qt Designer** pierwotnie było niezależnym narzędziem, natomiast obecnie jest ono częścią środowiska **Qt Creator**.

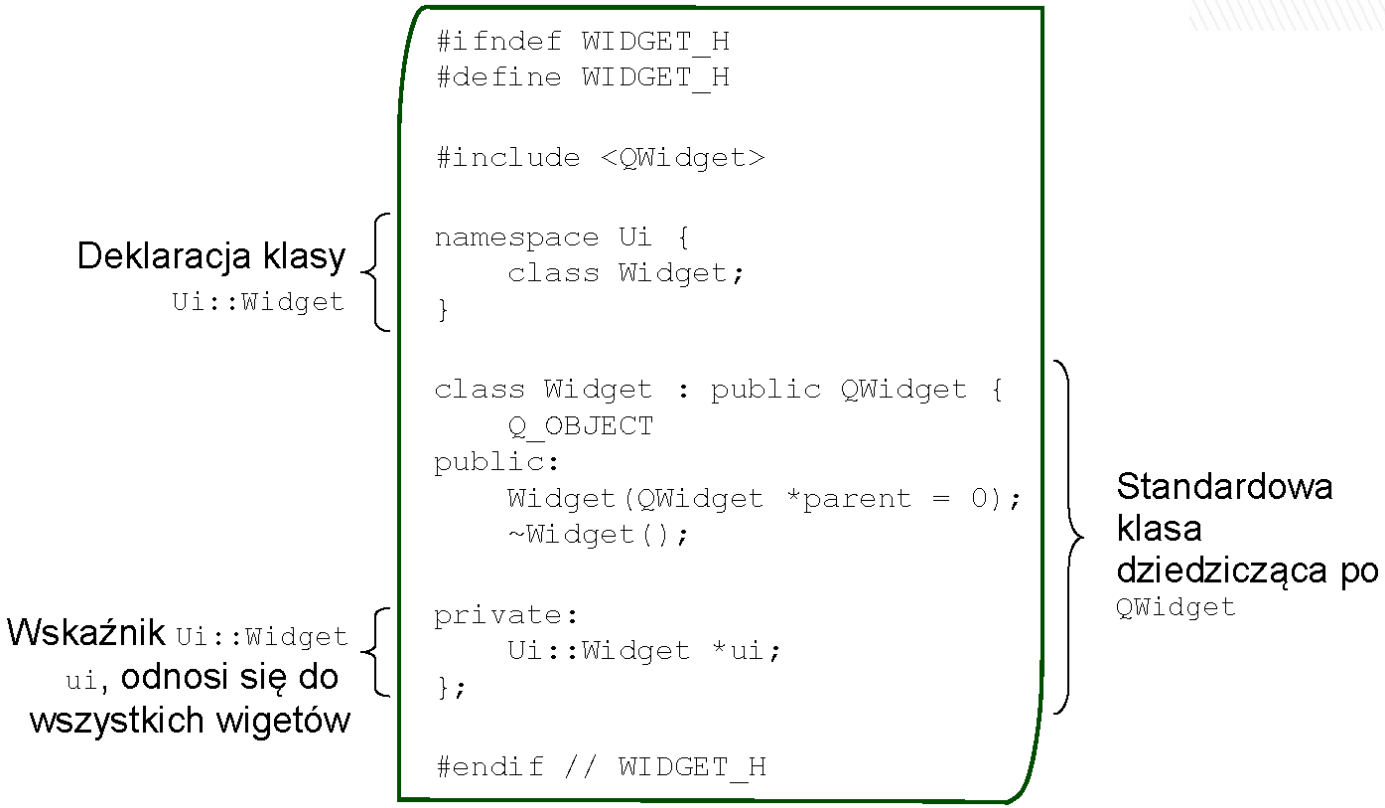
Qt Designer posiada graficzny edytor okien (formularzy). Umożliwia dodawanie komponentów metodą przeciągnij i upuść.

Podczas pracy pozwala na korzystanie z szablonów i tworzenie powiązań.

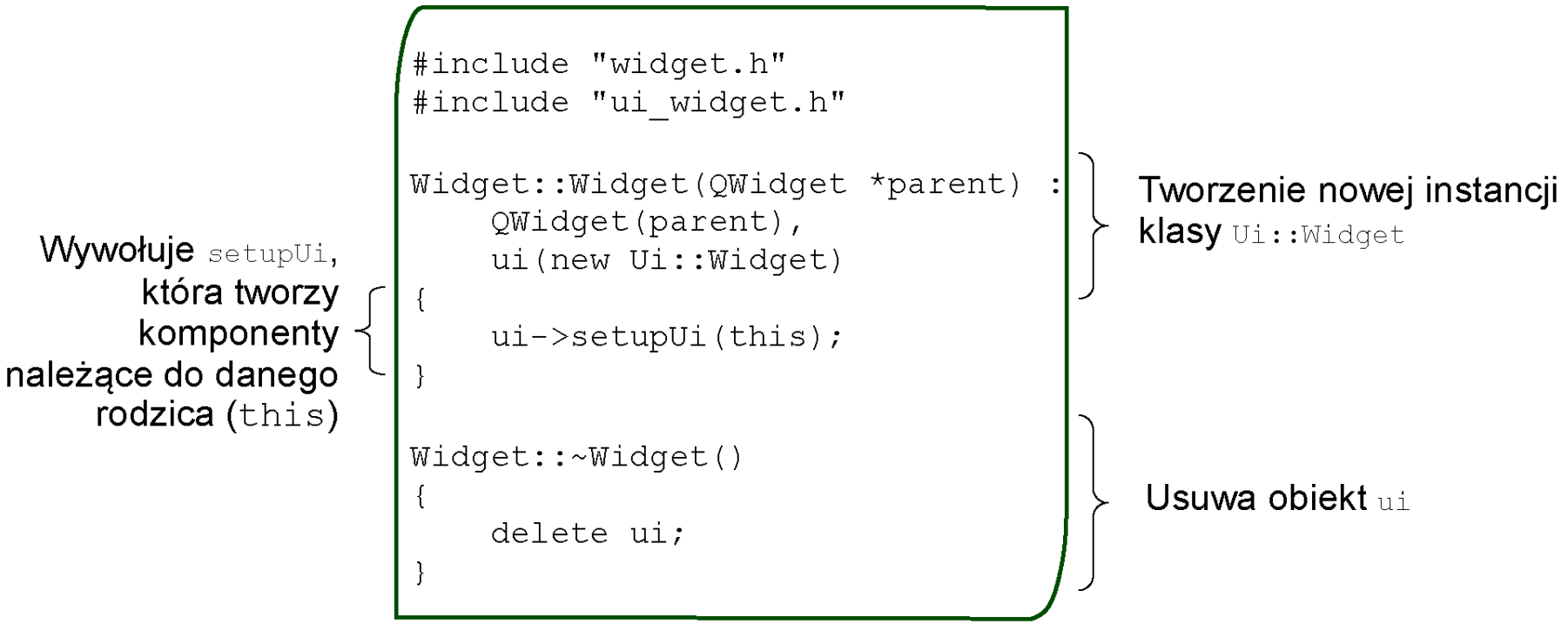

sources
*.cpp
compiles
object files
*.o
links
executables
includes
compiles
headers
*.h
mocs
generated
moc_*.cpp
user
interfaces
*.ui


sources
*.cpp
compiles
object files
*.o
links
executables
includes
compiles
headers
*.h
mocs
generated
moc_*.cpp
user
interfaces
*.ui
uics
generated
ui_*.h

Kod programu

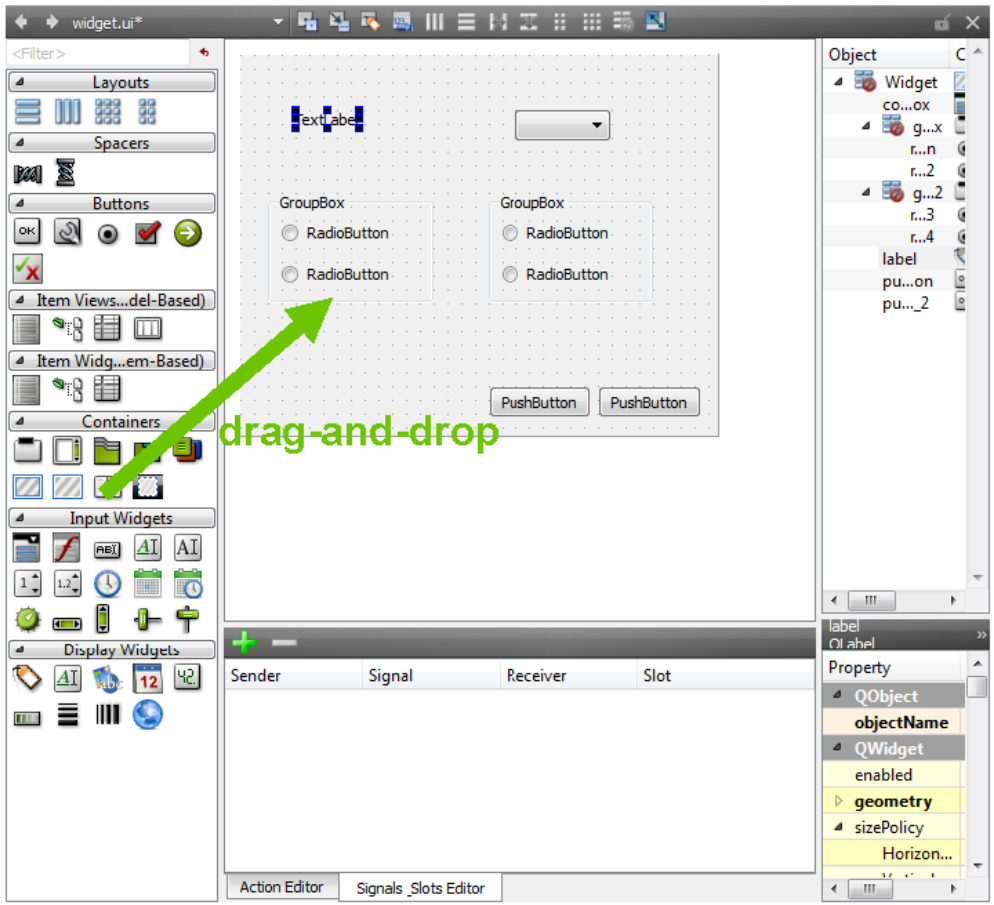



```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

namespace Ui {
    class Widget;
}

class Widget : public QWidget {
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
};

#endif // WIDGET_H
```

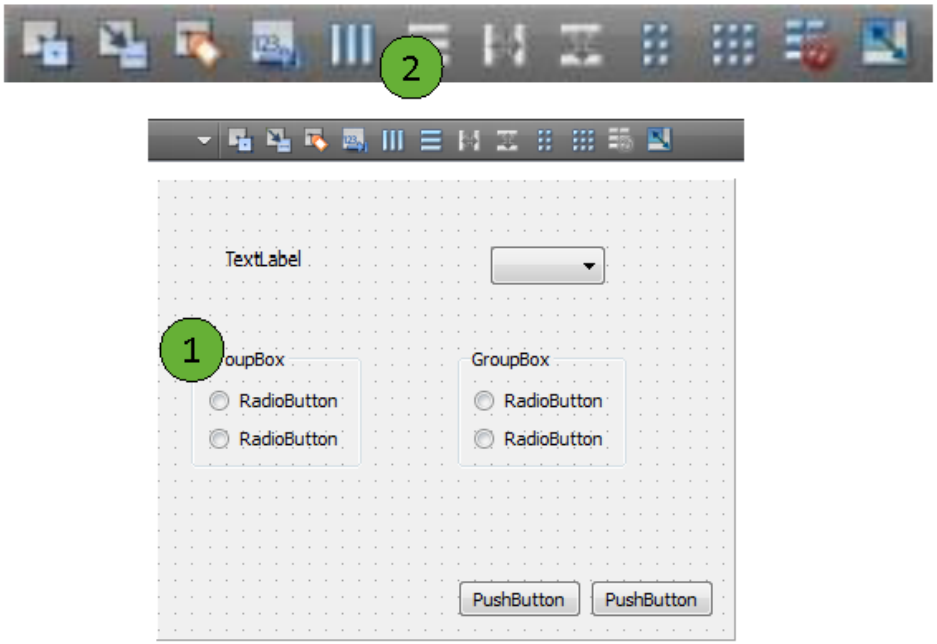

#include "widget.h"
#include "ui_widget.h"
Widget::Widget(QWidget *parent) :
QWidget(parent),
ui(new Ui::Widget)
{
ui->setupUi(this);
}
Widget::~Widget()
{
delete ui;
}
Wywołuje setupUi, która tworzy komponenty należące do danego rodzica (this)
Tworzenie nowej instancji klasy Ui::Widget
Usuwa obiekt ui

Praca z Qt Designer

1. Umieść komponenty na formularzu

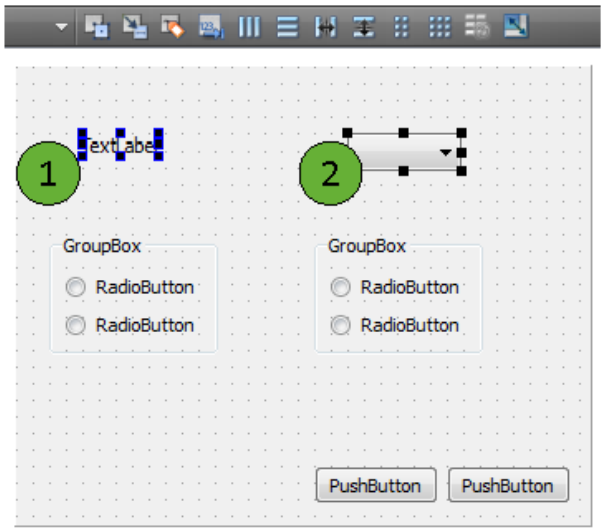


- Zastosuj odpowiednie szablony



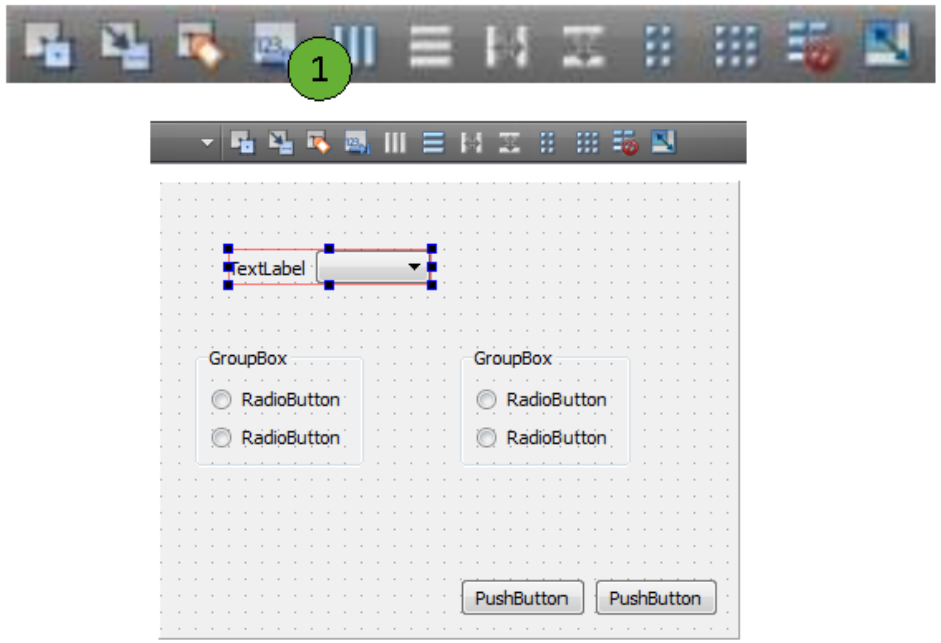
1. Zaznacz każdy komponent group box, **2**. Zastosuj szablon

- Zastosuj odpowiednie szablony



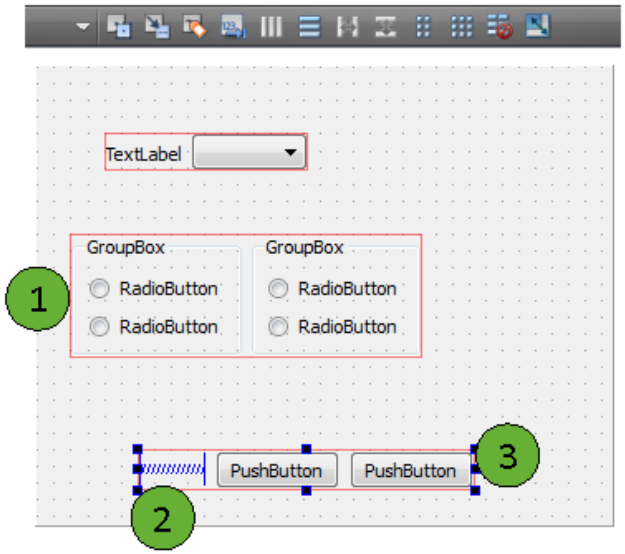
1. Zaznacz etykietę (poprzez kliknięcie), **2**. Zaznacz combobox (Ctrl+kliknięcie)

- Zastosuj odpowiednie szablony



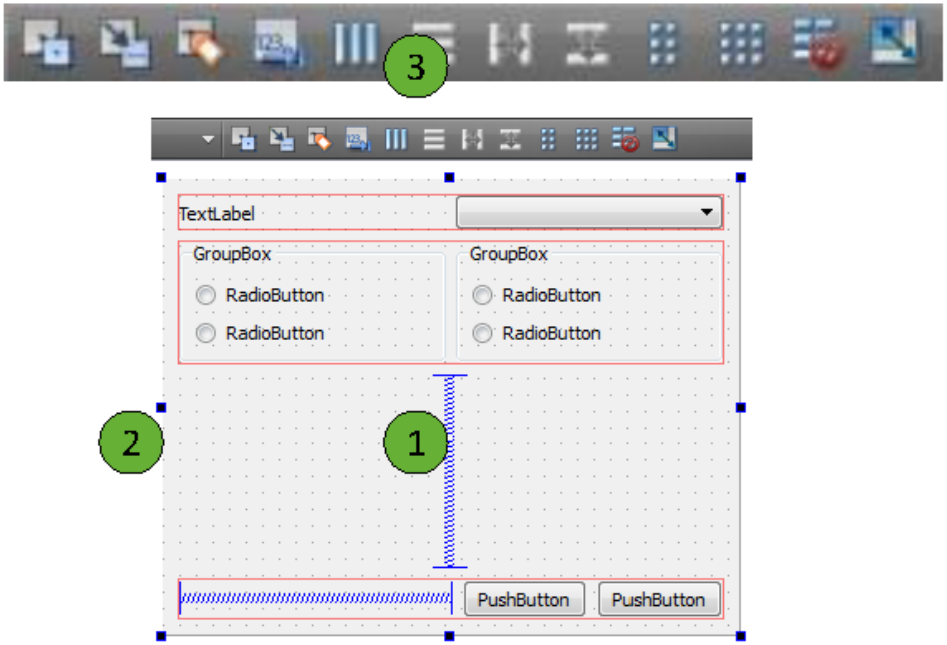
1. Zastosuj układ horyzontalny

- Zastosuj odpowiednie szablony



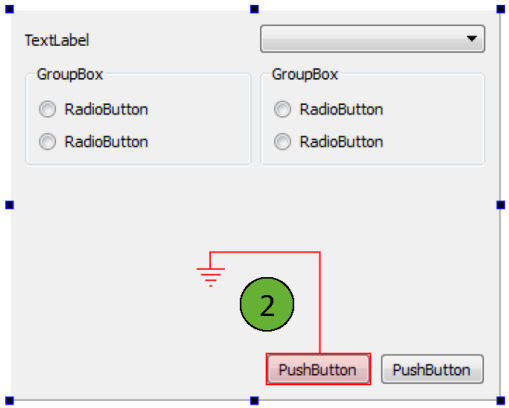
1. Zaznacz oba komponenty typu groupbox i je ustaw, **2**. dodaj spacer,
3. ponów procedurę dla przycisków

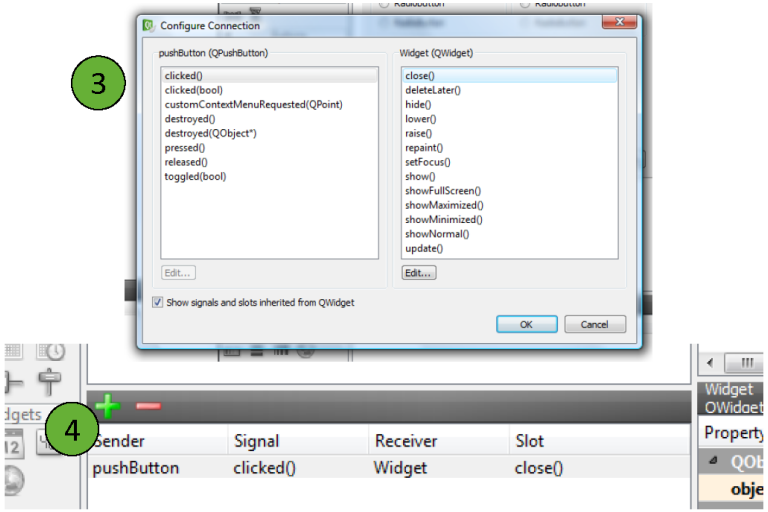
- Zastosuj odpowiednie szablony



1. Dodaj spacer, **2**. Zaznacz formularz, **3**. Zastosuj układ wertykalny

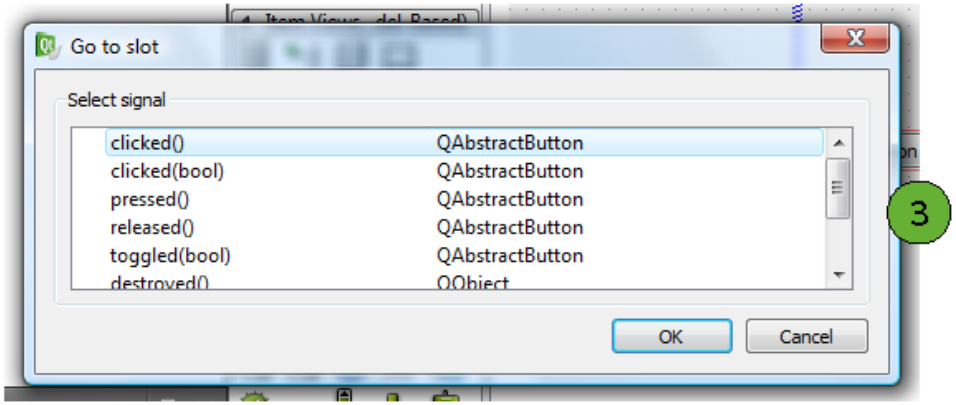
- Stwórz połączenia między elementami formularza

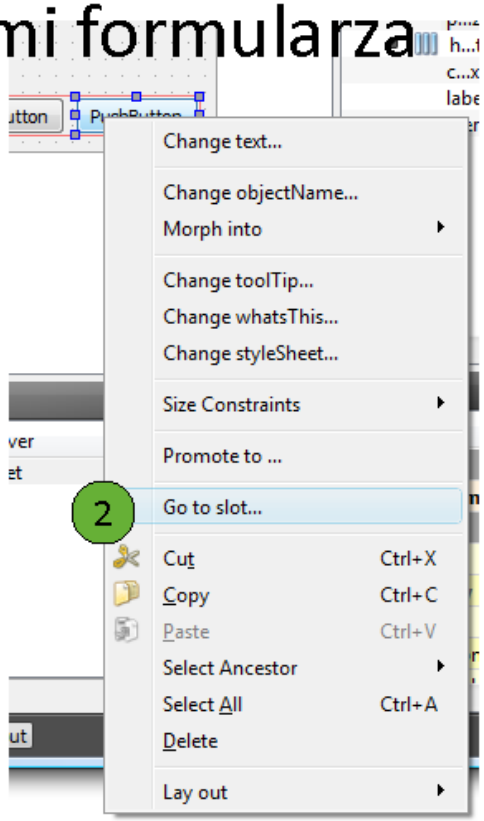




1. Wejdź w tryb edycji sygnałów i slotów, **2**. przeciągnij z jednego komponentu do innego, **3**. wybierz sygnał i slot, **4**. sprawdź poprawność połączenia

- Stwórz połączenia między elementami formularza





1. Przejdź do trybu edycji komponentów, **2**. Kliknij prawym przyciskiem myszy i wybierz **opcję przejdź do slotu 3**. wybierz odpowiedni sygnał

- Stwórz pozostały kod
- Odwołuj się do komponentów poprzez wskaźnik `ui`

```
class Widget : public QWidget {
    ...
private:
    Ui::Widget *ui;
};
```

```
void Widget::memberFunction()
{
    ui->pushButton->setText(...);
}
```

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

ąśżźć

Nowe okna

- Widgety bez ustawionego rodzica, stając się automatycznie oknami
  - `QWidget` – zwyczajne okno
  - `QDialog` – okno dialogowe, najczęściej z przyciskami typu OK, Anuluj, itp.
  - `QMainWindow` – okno aplikacji z menu głównym, paskiem narzędzi i statustu.
- `QDialog` oraz `QMainWindow` dziedziczą po `QWidget`

Używanie QWidget jako okna

- Każdy widget może być oknem
- Gdy widget posiada rodzica, przekazuje flagę `Qt::Window` do konstruktura klasy bazowej `QWidget`
- Funkcja `setWindowModality` ustawia okno jako modalne
  - `NonModal` – możliwa jest interakcja ze wszystkimi oknami
  - `WindowModal` – tylko okno rodzica jest blokowane
  - `ApplicationModal` – wszystkie inne okna są blokowane

Właściwości okna

- Metoda `setWindowTitle` ustawia tytuł okna
- Konstruktor klasy `QWidget` i flagi okna

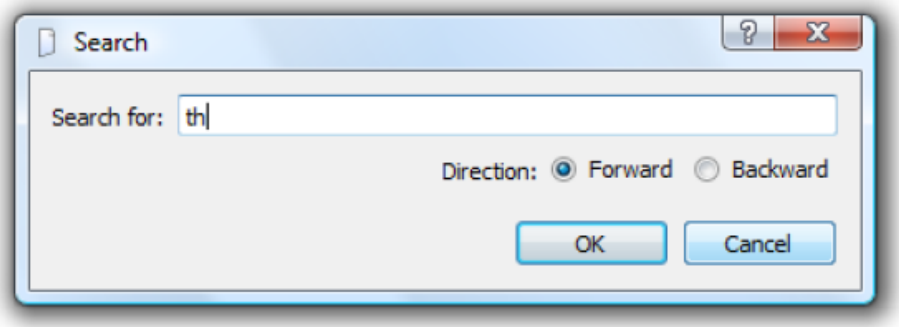
```
QWidget::QWidget(QWidget *parent, Qt::WindowFlags f=0)
```

  - `Qt::Window` – tworzy nowe okno
  - `Qt::CustomizeWindowHint` – usuń wartości domyślne
    - `Qt::WindowMinimizeButtonHint`
    - `Qt::WindowMaximizeButtonHint`
    - `Qt::WindowCloseButtonHint`
    - itp

Słowo **hint** jest tutaj kluczowe.

Wykorzystanie QDialog

- Najpopularniejsze okno dialogowe, to okno wyszukiwania



- Dziedziczy z klasy `QDialog`

Jak stworzyć własne okno dialogowe do wyszukiwania?

```
class SearchDialog : public QDialog
{
    Q_OBJECT
public:
    explicit SearchDialog(const QString &initialText,
                          bool isBackward, QWidget *parent = 0);

    bool isBackward() const;
    const QString &searchText() const;

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

Nowy konstruktor

Akcesory i modyfikatory

Jak stworzyć własne okno dialogowe do wyszukiwania?

```
SearchDialog::SearchDialog(const QString &initialText,
                           bool isBackward, QWidget *parent) :
    QDialog(parent), ui(new Ui::SearchDialog)
{
    ui->setupUi(this);

    ui->searchText->setText(initialText);
    if(isBackward)
        ui->directionBackward->setChecked(true);
    else
        ui->directionForward->setChecked(true);
}

bool SearchDialog::isBackward() const
{
    return ui->directionBackward->isChecked();
}

const QString &SearchDialog::searchText() const
{
    return ui->searchText->text();
}
```

Inicjalizacja pól zgodnie z argumentami konstruktora

Akcesory

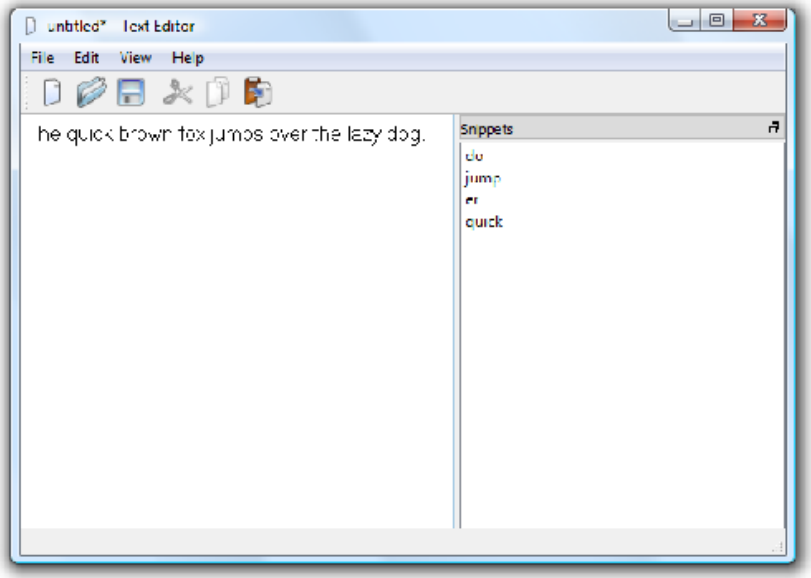
Jak stworzyć własne okno dialogowe do wyszukiwania?

```
void MyWindow::myFunction()
{
    SearchDialog dlg(settings.value("searchText","").toString(),
                     settings.value("searchBackward", false).toBool(),
                     this);

    if(dlg.exec() == QDialog::Accepted)
    {
        QString text = dlg.searchText());
        bool backwards = dlg.isBackward());
        ... //logika do wyszukiwania
    }
}
```

QDialog::exec wywołuje modalne okno dialogowe i zwraca rezultat jako Accepted lub Rejected

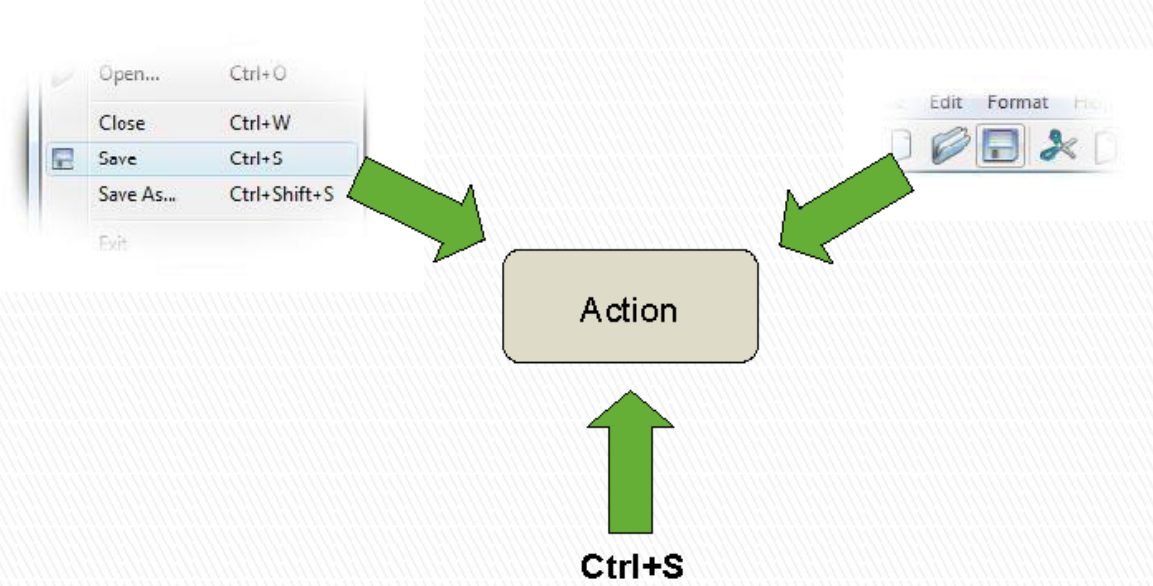
Kiedy korzystać z QMainWindow?

- `QMainWindow` to typowe okno aplikacji.
- Posiada ono zwykle:
  - Menu główne
  - Pasek narzędzi
  - Pasek stanu
  - Widget centralny



Prawidłowa implementacja menu – QAction

- Wiele elementów GUI odnosi się do tej samej funkcjonalności



- Akcje można przyporządkować do różnych komponentów, wraz z ustaleniem ikon, podpowiedzi itp.

Prawidłowa implementacja menu – QAction

- A `QAction` encapsulates all settings needed for menus, tool bars and keyboard shortcuts
- Commonly used properties are
  - `text` – the text used everywhere
  - `icon` – icon to be used everywhere
  - `shortcut` – shortcut
  - `checkable`/`checked` – whether the action is checkable and the current check status
  - `toolTip`/`statusTip` – tips text for tool tips (hover and wait) and status bar tips (hover, no wait)

Prawidłowa implementacja menu – QAction

```
QAction *action = new QAction(parent);

action->setText("text");

action->setIcon(QIcon(":/icons/icon.png"));

action->setShortcut(QKeySequence("Ctrl+G"));



action->setData(myDataQVariant);
```

Tworzenie nowej akcji

Ustawianie właściwości: tekst, ikona, skrót klawiszowy

QVariant może przechowywać dane powiązane z konkretną akcją

- Lub za pomocą QtDesigner

Dodawanie akcji

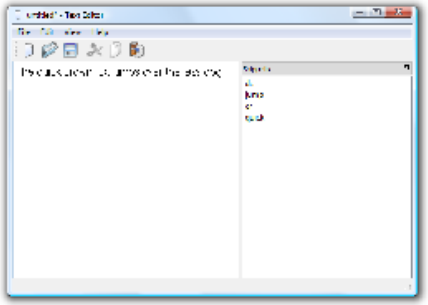
- Z poziomu kodu:

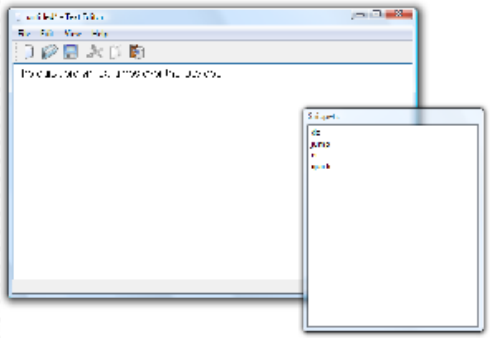
```
myMenu->addAction(action);
myToolBar->addAction(action);
```

- Z poziomu QtDesigner:

Dodatek: Dock widgets

- Dock widgets – komponenty, które można odczepić od głównego okna
- Dodaj widgety do `QDockWidget`
- `QMainWindow::addDockWidget` dodane dock do głównego okna





Dodatek: Dock widgets

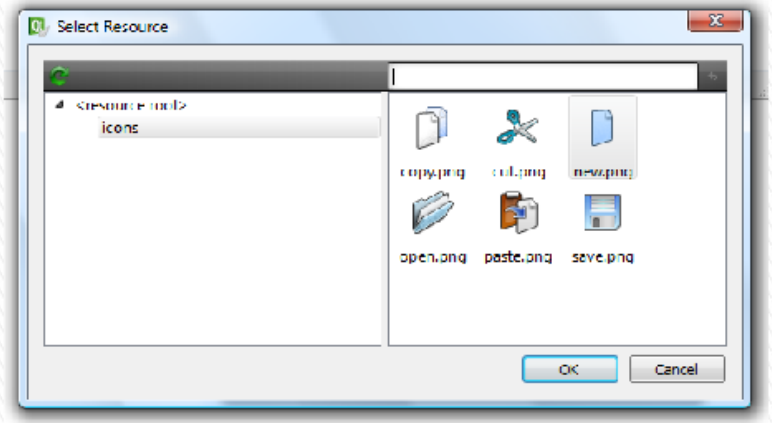
```
void MainWindow::createDock()
{
    QDockWidget *dock = new QDockWidget("Dock", this);

    dock->setFeatures(QDockWidget::DockWidgetMovable |
                      QDockWidget::DockWidgetFloatable);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea |
                          Qt::RightDockWidgetArea);
    dock->setWidget(actualWidget);

    ...

    addDockWidget(Qt::RightDockWidgetArea, dock);
}
```

Dodatek: Zasoby aplikacji – ikony

- Stosuj prefiks dla ścieżki i nazwy pliku :

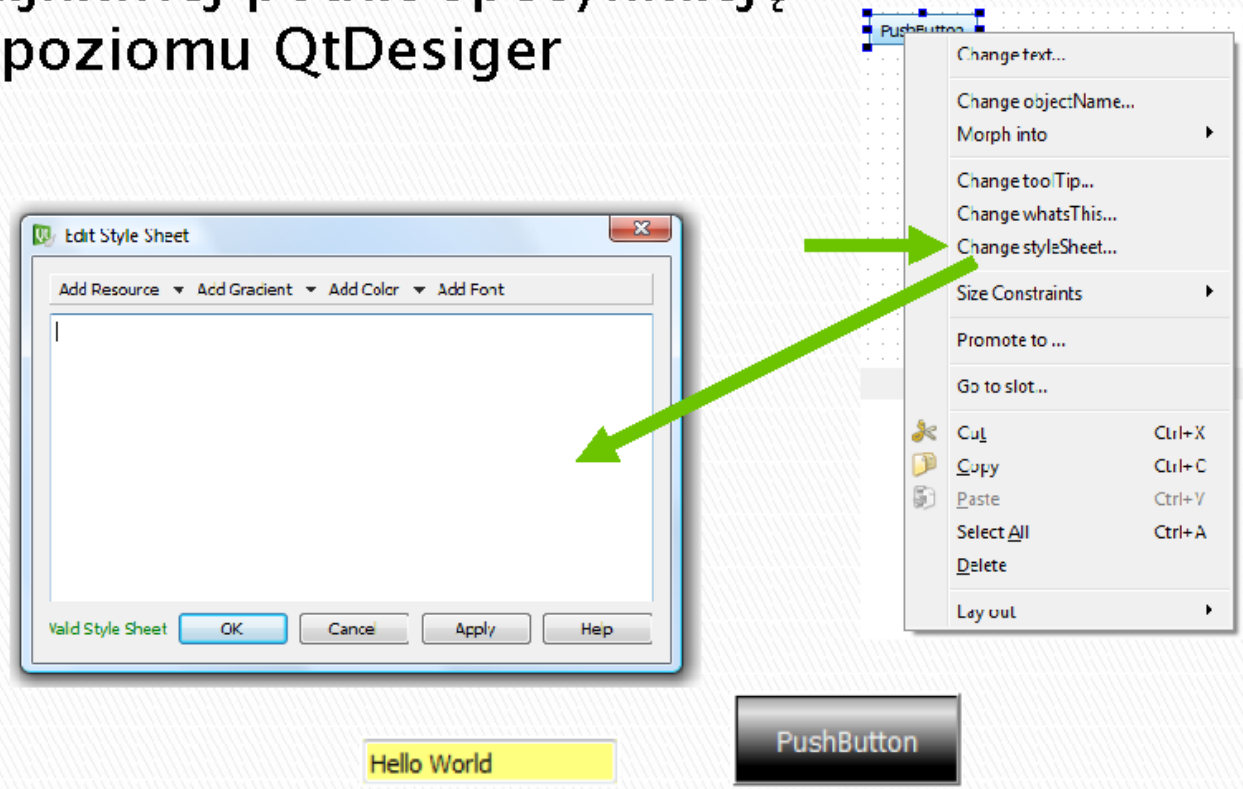
```
QPixmap pm(":/images/logo.png");
```

- Stwórz plik zasobów z poziomu QtDesigner:
- `Plik -> Nowy projekt lub plik -> Qt -> Plik z zasobami Qt`



Dodatek: Arkusze stylów

- Najłatwiej podać specyfikację z poziomu QtDesiger



Dodatek: Arkusze stylów

- Żeby zastosować styl do całej aplikacji użyj `QApplication::setStyleSheet`

Klasa jako selektor

Obiekt identyfikowany przez nazwę

```
QLineEdit { background-color: yellow }
QLineEdit#nameEdit { background-color: yellow }

QTextEdit, QListView {
    background-color: white;
    background-image: url(draft.png);
    background-attachment: scroll;
}

QGroupBox {
    background-color: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
                                      stop: 0 #E0E0E0, stop: 1 #FFFFFF);
    border: 2px solid gray;
    border-radius: 5px;
    margin-top: 1ex;
}
```

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

ąśżźć

Obsługa ciągów znaków

- W języku C tablica znaków wykorzystuje wyłącznie aktualną (lokalną) stronę kodową.

```
char *text = "Hello world!";
```

- Qstring jest klasą uwzględniającą szereg współczesnych wymagań dotyczących łańuchów
  - Obsługuje Unicode oraz mechanizmy transkrypcji
  - Niejawne współdzielenie (implicit sharing) poprawiające wydajność

```
QString s("Hello world!");
QString s2 = s //ten sam obszar pamięci
```

QString

- Dane reprezentowane są jako Unicode co umożliwia przechowanie napisu praktycznie w każdym języku
- Umożliwia konwersję między różnymi standardami (stronami kodowymi)

```
QString::toLatin1 - QString::toLocal8Bit
```

- Udostępnia dogodne API do przetwarzania i modyfikacji łańcuchów tekstowych

Tworzenie napisów

- Istnieją 3 metody tworzenia napisów:
- Metoda wykorzystująca `operator+`

```
QString res = "Hello " + name +
    ", the value is " + QString::number(42);
```

- Metody klasy `QStringBuilder`

```
QString res = "Hello " % name %
    ", the value is " % QString::number(42);
```

- Wykorzystanie `arg`

```
QString res = QString("Hello %1, the value is %2")
    .arg(name)
    .arg(42);
```

QStringBuilder

- Użycie operatora + do łączenia ciągów powoduje konieczność wielokrotnej alokacji pamięci
- Lepszym rozwiązaniem jest zastosowanie `QStringBuilder` i wykorzystanie operatora `%`
- `QStringBuilder` najpierw określa całkowitą długość tekstu wynikowego, a następnie dokonuje pojedynczej alokacji pamięci

```
QString res = "Hello " % name %
    ", the value is %" % QString::number(42);
```

```
QString temp = "Hello ";
temp = temp % name;
temp = temp % ", the value is %"
temp = temp % QString::number(42);
```

Łączenie w taki sposób obniża wydajność aplikacji

QString::arg

- Metoda arg zastępuje `%1-99` odpowiednimi wartościami

```
"%1 + %2 = %3, the sum is %3"
```

Wszystkie wyrażenia %n są zastępowane

- Konwersja działa dla napisów, znaków, liczb

```
...).arg(qulonglong a)      ...).arg(QString, ... QString)
...).arg(short a)           ...).arg(int a)
...).arg(ushort a)          ...).arg(uint a)
...).arg(QChar a)           ...).arg(long a)
...).arg(char a)            ...).arg(ulong a)
...).arg(double a)          ...).arg(qlonglong a)
```

Do 9 argumentów dla każdego arg

- Można konwertować pomiędzy systemami liczbowymi

```
...).arg(value, width, base, fillChar);
...).arg(42, 3, 16, QChar('0')); // Results in 02a
```

Podciągi

- Podciąg można wyłuskać metodami `left`, `right`

```
QString s = "Hello world!";
r = s.left(5); // "Hello"
r = s.right(1); // "!"
r = s.mid(6,5); // "world"
```

- Gdy nie podamy długości dla `mid`, zwracany jest ciąg począwszy od wskazanego znaku do końca

```
r = s.mid(6); // "world!"
```

- Metoda `replace` znajduje i zastępuje podciąg

```
r = s.replace("world", "universe"); // "Hello universe!"
```

Wyświetlanie w konsoli

- Qt jest stworzony przede wszystkim dla aplik--- -- okienkowych, stąd nie ma w nim typowej linii komend
- Do wypisania danych na konsole (debugowanie) wykorzystywana jest funkcja `qDebug`
  - Jest zawsze dostępna, nie będzie jednak wyświetlać danych prze kompilowaniu w trybie Release
  - Działa jak funkcja `printf` (ale automatycznie dodaje `\n`)

```
qDebug("Integer value: %d", 42);
qDebug("String value: %s", qPrintable(myQString));
```

  - Może być wykorzystywana z operatorem strumieniowym `<<`

```
#include <QtDebug>

qDebug() << "Integer value:" << 42;
qDebug() << "String value:" << myQString;
qDebug() << "Complex value:" << myQColor;
```

Konwersja z i na liczby

- Konwersja liczby do napisu

```
QString::number(int value, int base=10); QString
twelve = QString::number(12); // "12" QString
oneTwo = QString::number(0x12, 16); // "12"

QString::number(double value, char format='g', int precision=6);
QString piAuto = QString::number(M_PI); // "3.14159"
QString piScientific = QString::number(M_PI,'e'); // "3.141593e+00"
QString piFixedDecimal = QString::number(M_PI,'f',2); // "3.14"
```

- Konwersja napisu do liczby

```
bool ok;
QString i = "12";
int value = i.toInt(&ok);
if(ok) {
    // Converted ok
}
```

```
bool ok;
QString d = "12.36e-2";
double value = d.toDouble(&ok);
if(ok) {
    // Converted ok
}
```

Praca z std::(w)string

- Konwersja z oraz na ciągi znaków wykorzystywane w STL jest pomocna podczas wykorzystywania zawartych tam algorytmów
- Konwersja z ciągu STL

```
std::string ss = "Hello world!";
std::wstring sws = "Hello world!";

QString qss = QString::fromStdString(ss);
QString qsws = QString::fromStdWString(sws);
```

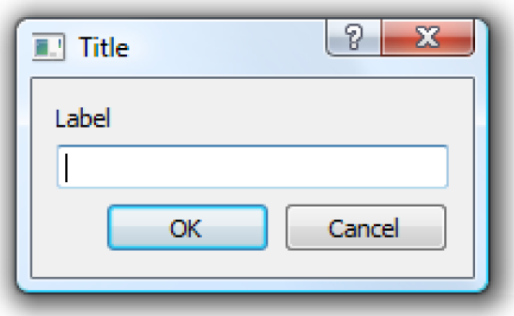
- Konwersja na ciąg STL

```
QString qs = "Hello world!";
std::string ss = qs.toStdString();
std::wstring sws = qs.toStdWString();
```

Puste ciągi czy null?

- `QString` może być null (nic nie zawierać)

```
QString n = QString();

n.isNull();  // true
n.isEmpty(); // true
```



- Może też być pusty, tj. Zawierać pusty ciąg

```
QString e =  "";

e.isNull();  // false
e.isEmpty(); // true
```

Rozdzielanie i łączenie

- `QString` może być podzielony na elementy

```
QString whole = "Stockholm - Copenhagen - Oslo - Helsinki";
QStringList parts = whole.split(" - ");
```

- W wyniku otrzymujemy obiekt `QStringList`, którego elementy można połączyć tworząc `QString`

```
QString wholeAgain = parts.join(", ");
    // Results in "Stockholm, Copenhagen, Oslo, Helsinki"
```

QStringList

- The `QStringList` jest specjalizowaną listą
- Zaprojektowana do przechowywania napisów

  Dostarcza dogodnego API do pracy z listami napisów

- Klasa wykorzystuje niejawne współdzielenie danych (implicit sharing)
  - Kopiowanie realizowane jest dopiero przy modyfikacji zawartości
  - Szybkie przy przekazywaniu wartości

Operacje na listach napisów

- Operator `<<` służy do dodawania elementów

```
QStringList verbs;
verbs = "running" << "walking" << "compiling" << "linking";
```

- Metoda `replaceInStrings` pozwala na przeszukiwanie i zastępowanie elementów całej `QStringList`.

```
qDebug() << verbs; // ("running", "walking", "compiling", "linking")
verbs.replaceInStrings("ing", "er");
qDebug() << verbs; // ("runner", "walker", "compiler", "linker")
```

Sortowanie i filtrowanie

- `QStringList` można posortować...

```
qDebug() << capitals; // ("Stockholm", "Oslo", "Helsinki", "Copenhagen")
capitals.sort();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

- ...filtrować...

Zwraca uwagę na wielkość znaków

```
QStringList capitalsWithO = capitals.filter("o");
qDebug() << capitalsWithO; // ("Copenhagen", "Oslo", "Stockholm")
```

- ...usunąć duplikaty

```
capitals << capitalsWithO;
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm",
                      //  "Copenhagen", "Oslo", "Stockholm")
capitals.removeDuplicates();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

Iteracyjne odwołanie do elementów listy

- Operator `operator[]` i metoda `length` pozwalają na odwołanie się do elementów `QStringList` jak dla klasycznej tablicy

```
QStringList capitals;
for(int i=0; i<capitals.length(); ++i)
    qDebug() << capitals[i];
```

- Innym sposobem jest wykorzystanie metody `at()` umożliwiającej odczyt danego elementu
- Metoda `at()` uniemożliwia kopiowanie elementu, co może `operator[]`
- Dostępne jest również makro `foreach`

```
QStringList capitals;
foreach(const QString &city, capitals)
    qDebug() << city;
```

Kolekcje w Qt

- `QList` jest jednym z wielu wzorców (kolekcji):
  - `QLinkedList` – lista dynamiczna
  - `QVector` – lista dynamiczna (ciągły obszar pamięci)
  - `QStack` – LIFO, last in – first out
  - `QQueue` – FIFO, first in – first out
  - `QSet` – zbiór niepowtarzalnych elementów
  - `QMap` – tablica asocjacyjna, sortowana po kluczach
  - `QHash` – tablica asocjacyjna z haszowaniem (szybsza niż `QMap`)
  - `QMultiMap` – tablica asocjacyjna z wieloma wartościami dla klucza
  - `QMultiHash` – tablica asocjacyjna z wieloma wartościami dla klucza (wymaga funkcji haszującej)

Dodawanie do listy

- Za pomocą operatora <<

```
QList<int> fibonacci;
fibonacci << 0 << 1 << 1 << 2 << 3 << 5 << 8;
```

- Za pomocą metod `prepend`, `insert` oraz `append`

```
QList<int> list;
list.append(2);
```

index 0: **2**

```
list.append(4);
```

index 0: **2**
index 1: **4**

```
list.insert(1,3);
```

index 0: **2**
index 1: **3**
index 2: **4**

```
list.prepend(1);
```

index 0: **1**
index 1: **2**
index 2: **3**
index 3: **4**

Usuwanie

- Usuwanie elementów z `QList` metodami `removeFirst`, `removeAt`, `removeLast`

```
while(!list.isEmpty())
    list.removeFirst();
```

- Za pomocą `takeFirst`, `takeAt`, `takeLast`

```
QList<QWidget*> widgets;
widgets << new QWidget << new QWidget;
while(!widgets.isEmpty())
    delete widgets.takeFirst();
```

- Usuwanie elementów o konkretnej wartości metodą `removeAll` lub `removeOne`

```
QList<int> list;
list << 1 << 2 << 3 << 1 << 2 << 3;
list.removeAll(2); // Leaves 1, 3, 1, 3
```

Dostęp do elementów

- Elementy `QList` indeksowane są od `0` do `length-1`
- Do określonych elementów listy można uzyskać dostęp funkcją `at()` lub operatorem `[]`

```
for(int i=0; i<list.length(); ++i)
    qDebug("At: %d, []: %d", list.at(i), list[i]);

for(int i=0; i<100; ++i)
    qDebug("Value: %d", list.value(i));
```

- Operator`[]` zwraca referencję (do odczytu i zapisu)

```
for(int i=0; i<list.length(); ++i)
    list[i]++;
```

Iteratory – wersja Java

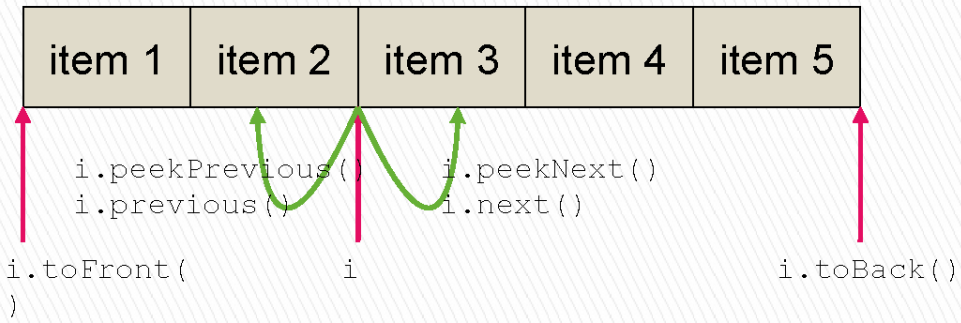
- Qt wspiera iteratory w konwencji Java:

```
QListIterator<int> iter(list);
        while(iter.hasNext())
            qDebug("Item: %d", iter.next());
```

Zwróć wartość i przejdź do następnego elementu

Użyj `QMutableListIterator` jeżeli modyfikujesz elementy listy

- Wykorzystanie iteratorów:
  - `toFront` przenosi iterator przed pierwszy element
  - `toBack` przenosi iterator za ostatni element
  - Do elementów odwołuj się przez `peekNext` oraz `peekPrevious`
  - Przesuwaj się używając `next` lub `previous`



Iteratory – wersja STL

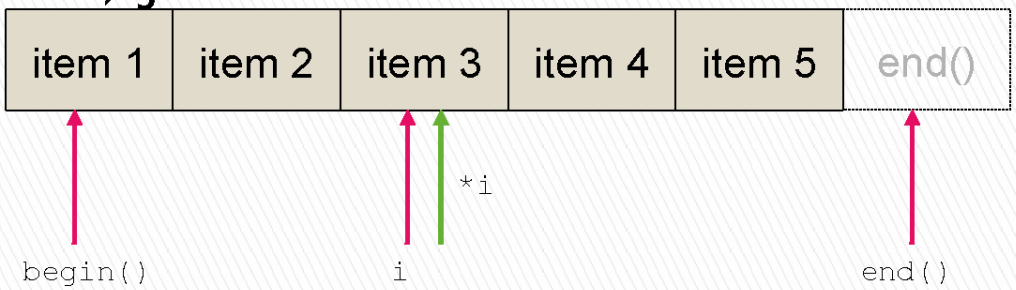
- Qt wspiera iteratory w konwencji STL:

```
for(QList<int>::const_iterator iter=list.begin();
    iter!=list.end(); ++iter)
    qDebug("Item: %d", *iter);
```

Użyj klasę `Iterator` jeżeli modyfikujesz elementy

Zarówno `Iterator|iterator` jak i `ConstIterator|const_iterator` mogą być użyte

- Iteratory STL wskazują dany element, wykorzystując nieistniejące elementy (null) jako wskaźnik końca
  - Pierwszy element dostępny jest przez `begin`
  - Wskaźnik końca zwracany jest przez `end`
  - Operator `*` odwołuje się do wartości elementu



  - Przesuwając się do tyłu najpierw należy zmienić pozycję iteratora a dopiero później odwołać się do wartości elementu

Iteracyjne przeglądanie listy

- Do iteracyjnego przeglądania elementów kolekcji można wykorzystać pętlę `foreach`

```
QStringList texts;
foreach(QString text, texts)
    doSomething(text);
```

Użycie referencji do stałej zwiększa wydajność, równocześnie uniemożliwia jednak zmianę wartości elementu

```
QStringList texts;
foreach(const QString &text, texts)
    doSomething(text);
```

Interakcja z STL

- Obiekt klasy `QList` może zostać przekonwertowany do `std::list`

```
QList<int> list;
list << 0 << 1 << 1 << 2 << 3 << 5 << 8 << 13;

std::list<int> stlList = list.toStdList();

QList<int> otherList = QList<int>::fromStdList(stlList);
```

Z listy Qt do STL

Z listy STL do Qt

- Konwersja z i do listy STL oznacza tworzenie pełnej kopii listy – nie zostanie wykorzystany mechanizm współdzielenie (ang. *implicit sharing*)

Inne kolekcje

- Jakie są alternatywy dla `Qlist`?
- `QLinkedList`
  - Wolna podczas swobodnego dostępu (po indeksach)
  - Szybka, podczas używania iteratorów
  - Szybki (stały) czas dodania do środka listy
- `QVector`
  - Używa ciągłego obszaru pamięci
  - Wolne wstawianie na początek i do środka

Inne kolekcje

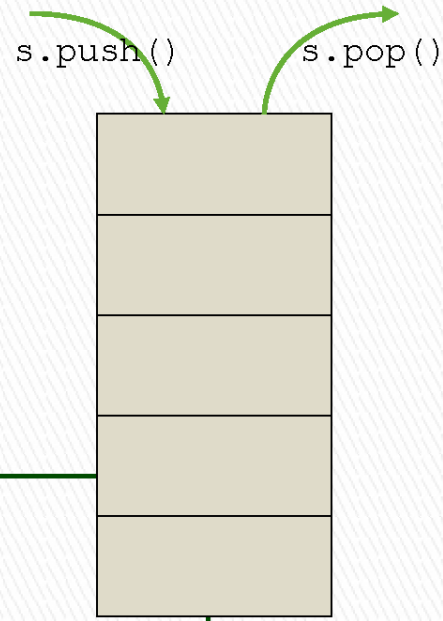
Collection	Index access	Insert	Prepend	Append
`QList`	O(1)	O(n)	Amort. O(1)	Amort. O(1)
`QLinkedList`	O(n)	O(1)	O(1)	O(1)
`QVector`	O(1)	O(n)	O(n)	Amort. O(1)

- Amort oznacza brak gwarancji do takiej złożoności
- Inne kolekcje
  - `QStringList` – bazuje na `QList`
  - `QStack` – bazuje na `QVector`
  - `QQueue` – bazuje na `QList`
  - `QSet` – bazuje na `QHash`

Stos- QStack

- Kontener typu LIFO
  last in, first out

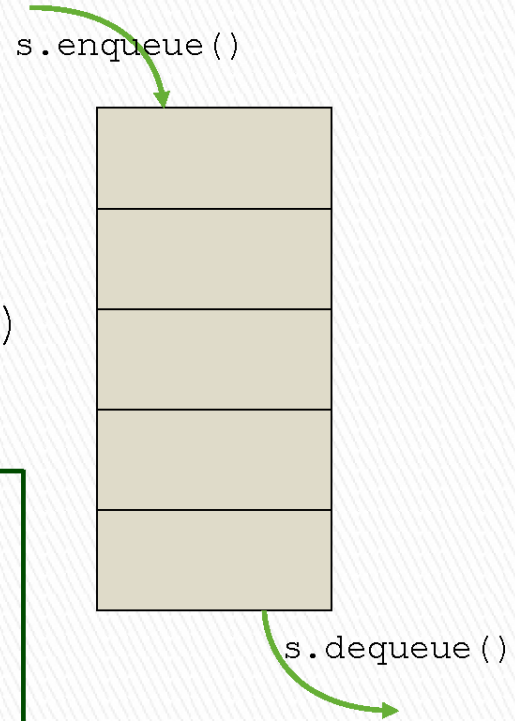
- Dodawanie: `push()`
- Usuwanie: `pop()`
- Podgląd elementu na górze: `top()`




```
QStack<int> stack;
stack.push(1);
stack.push(2);
stack.push(3);
qDebug("Top: %d", stack.top()); // 3
qDebug("Pop: %d", stack.pop()); // 3
qDebug("Pop: %d", stack.pop()); // 2
qDebug("Pop: %d", stack.pop()); // 1
qDebug("isEmpty? %s", stack.isEmpty()?"yes":"no");
```

Kolejka – QQueue

- Kontener typu FIFO
  first in, first out
- Dodawanie: `enqueue()`
- Usuwanie: `dequeue()`
- Pobranie pierwszego elementu: `head()`




```
QQueue<int> queue;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
qDebug("Head: %d", queue.head());    // 1
qDebug("Pop: %d", queue.dequeue()); // 1
qDebug("Pop: %d", queue.dequeue()); // 2
qDebug("Pop: %d", queue.dequeue()); // 3
qDebug("isEmpty? %s", queue.isEmpty()?"yes":"no");
```

Zbiór – QSet

- Wartości w ramach zbioru muszą być unikalne.
- Łatwe sprawdzenie czy wartość jest w zbiorze:

```
QSet<int> primes;
primes << 2 << 3 << 5 << 7 << 11 << 13;
for(int i=1; i<=10; ++i)
    qDebug("%d is %sprime", i, primes.contains(i)?"":"not ");
```

- Iterowanie po wszystkich elementach:

```
foreach(int prime, primes)
    qDebug("Prime: %d", prime);
```

- Można przekonwertować `QList` do `QSet`

```
QList<int> list;
list << 1 << 1 << 2 << 2 << 2 << 3 << 3 << 5;
QSet<int> set = list.toSet();
qDebug() << list; // (1, 1, 2, 2, 2, 3, 3, 5)
qDebug() << set;  // (1, 2, 3, 5)
```

Kolekcje do przechowywania danych typu klucz-wartość

- Klasy `QMap` i `QHash` umożliwiają tworzenie tablic asocjacyjnych

```
QMap<QString, int> map;

map["Helsinki"] = 1310755;
map["Oslo"] = 1403268;
map["Copenhagen"] = 1892233;
map["Stockholm"] = 2011047;

foreach(const QString &key,
map.keys())
    qDebug("%s", qPrintable(key));

if(map.contains("Oslo"))
{
    qDebug("Oslo: %d",
                map.value("Oslo"));
}
```

```
QHash<QString, int> hash;

hash["Helsinki"] = 1310755;
hash["Oslo"] = 1403268;
hash["Copenhagen"] = 1892233;
hash["Stockholm"] = 2011047;

foreach(const QString &key, hash.keys())
    qDebug("%s", qPrintable(key));
```

Wykorzystanie QMap

- Klasa `QMap` wymaga zdefiniowania operatora `operator<` dla typu klucza
  Operator ten odpowiada za kolejność kluczy
- Dodawanie elementów zrealizowano przez `operator[]` lub `insert`

```
map["Stockholm"] = 2011047;

map.insert("London", 13945000);
```

- Do odczytu należy wykorzystać `value` w połączeniu z `contains`

```
if(map.contains("Oslo"))
    qDebug("Oslo: %d",
           map.value("Oslo"));

qDebug("Berlin: %d", map.value("Berlin",42));
```

Użyj value zamiast [] by przypadkiem nie dodać elementu

Opcjonalna wartość domyśla

Jednokierunkowe funkcje skrótu

- `QMap` typ klucza zależy od definicji wykorzystanej podczas tworzenia QMapy
- `QHash` wykorzystuje wewnętrzne wartości typu `uint`
- Klucz za pomocą haszowania zmieniany jest w wartość typu `uint`
- Wykorzystanie wartości `uint` zwykle poprawia wydajność kodu
- Wykorzystanie haszowania oznacza, że klucze pamiętane są w dowolnej kolejności
- Funkcja haszująca musi być odporna na występowanie kolizji i efektywna obliczeniowo

Wykorzystanie QHash

- Należy stworzyć funkcję `qHash` dla klucza i przeciążyć operator `operator==`

```
uint qHash(const Person &p)
{
    return p.age() ^ qHash(p.name());
}

bool operator==(const Person &first, const Person &second)
{
    return ((first.name() == second.name()) &&
            (first.age() == second.age()));
}
```

Funkcja powinna mieć niską złożoność obliczeniową

- Dodawanie elementów odbywa się identycznie jak w przypadku `QMap`

Wiele wartości przypisanych do jednego klucza

- `QMultiMap` i `QMultiHash` dostarczają implementację tablic asocjacyjnych, w których wiele wartości może być przypisanych do tego samego klucza

Brak `[]`, trzeba użyć `insert`

`QMap` i `QHash` też to obsługują funkcją `insertMulti()`

```
QMultiMap<QString,int> multiMap;

multiMap.insert("primes", 2);
multiMap.insert("primes", 3);
multiMap.insert("primes", 5);
    ...
multiMap.insert("fibonacci", 8);
multiMap.insert("fibonacci", 13);

foreach(const QString &key, multiMap.uniqueKeys())
{
    QList<int> values = multiMap.values(key);
    QStringList temp;
    foreach(int value, values)
        temp << QString::number(value);

    qDebug("%s: %s", qPrintable(key), qPrintable(temp.join(",")));
}
```

Typy w Qt

- C++ nie definiuje wielkości typów dla typów wbudowanych

sizeof(void*) = ?

ARM = 4 bytes
x86 = 4 bytes
IA64 = 8 bytes
...

Zależne od architektury CPU, systemu operacyjnego, kompilatora

- W przenośności kodu istotne jest zachowanie tego samego rozmiaru typu na każdej platformie

Przenośne typy

Type	Size	Minimum value	Maximum value
uint8	1 byte	0	255
uint16	2 bytes	0	65 535
uint32	4 bytes	0	4 294 967 295
uint64	8 bytes	0	18 446 744 073 709 551 615
int8	1 byte	-128	127
int16	2 bytes	-32 768	32 767
int32	4 bytes	-2 147 483 648	2 147 483 647
int64	8 bytes	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
quintptr	“pointer sized”	n/a	n/a
qptrdiff	“pointer sized”	n/a	n/a
qreal	fast real values	n/a	n/a

Wszystkie typy zdefiniowane są w pliku nagłówkowym `<QtGlobal>`

Typy złożone w Qt

- Qt posiada wiele typów złożonych

QFont

QList

QColor

QString QRect QPen

QBrush

QSize

QPoint QImage

QPixmap

QByteArray

QVariant

- Typy wbudowane obsługiwane są przez konstruktor i metody `toTyp`

```
QVariant v;
int i = 42;
qDebug() << "Before:" << i; // Before: 42
v = i;
i = v.toInt();
qDebug() << "After:" << i;  // After: 42
```

- Typy złożone obsługiwane są za pomocą metody `setValue` oraz szablonowej `value<type>`

```
QVariant v;
QColor c(Qt::red);
qDebug() << "Before:" << c; // Before: QColor(ARGB 1, 1, 0, 0)
v.setValue(c);
c = v.value<QColor>();      // After: QColor(ARGB 1, 1, 0, 0)
qDebug() << "After:" << c;
```

Własny typ złożony

- Implementacja prostej klasy opisującej osobę

```
class Person
{
public:
    Person();
    Person(const Person &);
    Person(const QString &, int);

    const QString &name() const;
    int age() const;

    void setName(const QString &);
    void setAge(int);

    bool isValid() const;

private:
    QString m_name;
    int m_age;
};
```

Nie musi być QObject.

```
Person::Person() : m_age(-1) {}

...

void Person::setAge(int a)
{
    m_age = a;
}

bool Person::isValid() const
{
    return (m_age >= 0);
}
```

QVariant przechowujący obiekty klasy Person

- Wykorzystanie klasy `Person` w połączeniu z typem `QVariant` nie zadziała:

```
qmetatype.h:200: error: 'qt_metatype_id' is not a member of 'QMetaTypeId<Person>'
```

- Deklaracja takiego typu w warstwie metadanych rozwiązuje ten problem:

```
class Person
{
    ...
};

Q_DECLARE_METATYPE(Person)

#endif // PERSON_H
```

QVariant przechowujący obiekty klasy Person

- Jeżeli typ został zarejestrowany w systemie metadanych, Qt może go przechowywać jako `QVariant`

```
QVariant var;
var.setValue(Person("Ole", 42));
Person p = var.value<Person>();
qDebug("%s, %d", qPrintable(p.name()), p.age());
```

- Wymagania dla własnych obiektów:
  - Publiczny konstruktor domyślny
  - Publiczny konstruktor kopiujący
  - Publiczny destruktor

Ale jest jeszcze jeden problem...

- Większość połączeń z wykorzystaniem mechanizmu slotów i sygnałów jest połączeniami bezpośrednimi
  - Przy połączenia bezpośrednich wszystko działa
  - Ale istnieją połączenia kolejkowane, tj. asynchroniczne (nie-blokujące) - wówczas takie podejście nie zadziała (np. między wątkami)

```
connect(src, SIGNAL(), dest, SLOT(), Qt::QueuedConnection);

...

QObject::connect: Cannot queue arguments of type 'Person'
(Make sure 'Person' is registered using qRegisterMetaType().)
```

Błąd podczas wykonywania programu

Registering the type

- Typ błędu jednoznacznie sugeruje co trzeba zrobić
- Funkcja `qRegisterMetaType` musi zostać wywołana przed stworzeniem połączenai (zwykle w funkcji `main`)

```
int main(int argc, char **argv)
{
    qRegisterMetaType<Person>();
    ...
```

1 Informacje ogólne
2 Przetwarzanie tablic znaków i łańcuchów znaków
3 Obsługa wywołania parametrycznego
4 Graficzny interfejs użytkownika
5 Podstawy Windows API
6 Qt wprowadzenie
7 Sygnały i sloty
8 Kontrolki i szablony
9 Qt Designer
10 Wiele okien
11 Typy i kolekcje
12 Obsługa plików

ąśżźć

Pliki i systemy plików

- Sposób dostępu do systemu plików zależy od systemu operacyjnego, co rodzi następujące problemy:
  - System może mieć napędy (dyski) lub katalog root
  - Systemy mogą mieć różne separatory ścieżki dostępu “/” lub “\”
  - Systemy mogą mieć różną lokalizację katalogu do przechowywania danych tymczasowych
  - Różną lokalizację dla dokumentów użytkownika
  - Różną lokalizację dla folderu aplikacji

Ścieżki

- Klasa `QDir` dedykowana jest do obsługi ścieżek

```
QDir d = QDir("C:/");
```

- Posiada składowe statyczne umożliwiające określenie punktu początkowego

```
QDir d = QDir::root(); // C:/ on windows

QDir::current()  // Current directory
QDir::home()     // Home directory
QDir::temp()     // Temporary directory

// Executable directory path
QDir(QApplication::applicationDirPath())
```

Przeszukiwanie zawartości katalogu

- Metoda `entryInfoList` zwraca listę informacji o zawartości katalogu

```
QFileInfoList infos = QDir::root().entryInfoList();
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

Kolejność losowa

- W celu ograniczenia liczby elementów można zastosować filtr

```
QDir::Dirs
QDir::Files
QDir::NoSymLinks
```

Katalogi? Pliki? Dowiązania?

```
QDir::Readable
QDir::Writable
QDir::Executable
```

Jakie pliki?

```
QDir::Hidden
QDir::System
```

Jakie pliki?

Przeszukiwanie zawartości katalogu

- Można również posortować listę

```
QDir::Name
QDir::Time
QDir::Size
QDir::Type
```

Sortuj po...

```
QDir::DirsFirst
QDir::DirsLast
```

Katalogi przed czy po plikach

```
QDir::Reversed
```

Odwróć kolejność

- Lista wszystkich katalogów katalogu głównego sortowana po nazwie

Filtr

Kolejność

```
QFileInfoList infos =
    QDir::root().entryInfoList(QDir::Dirs, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

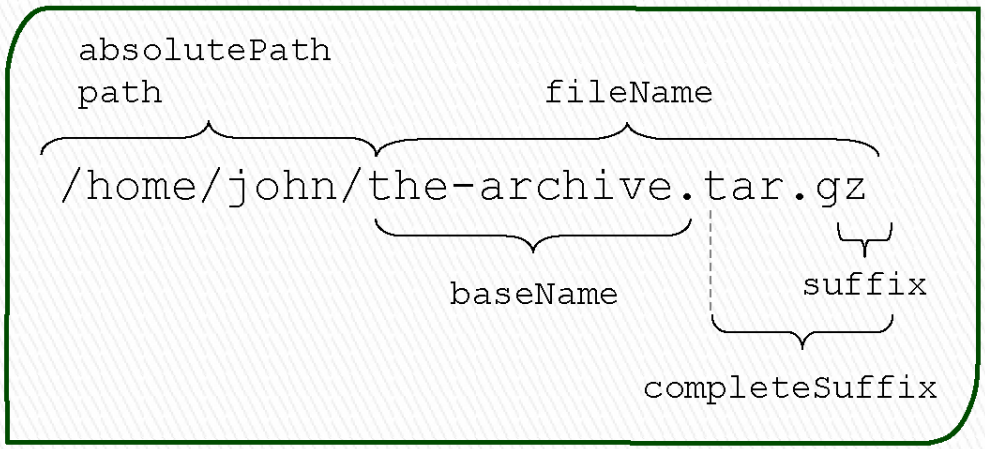
Przeszukiwanie zawartości katalogu

- Można również filtrować po rozszerzeniu / nazwie pliku

```
QFileInfoList infos =
    dir.entryInfoList(QStringList() << "*.cpp" << "*.h",
                      QDir::Files, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

QFileInfo

- Każdy obiekt `QFileInfo` ma pewna liczbę metod
  - `absoluteFilePath` – pełna ścieżka elementu
  - `isDir` / `isFile` / `isRoot` – typ elementu
  - `isWriteable` / `isReadable` / `isExecutable` – uprawnienia



Otwieranie i odczyt plików

- Dostęp do plików realizowany jest za pomocą klasy `QFile`

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QByteArray data = f.readAll();
processData(data);

f.close();
```

Wczytuje cały plik

```
while(!f.atEnd())
{
    QByteArray data = f.read(160);
    processData(data);
}
```

Odczytuje do 160 bajtów za jednym wykonaniem

Zapis do plików

- By zapisywać dane do pliku należy go otworzyć w trybie `WriteOnly` i wykorzystać metodę write

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QByteArray data = createData();
f.write(data);

f.close();
```

- Pliki mogą również zostać otwarte w trybie `ReadWrite`
- Flagi `Append` lub `Truncate` mogą być zastosowane w celu ustawienia trybu dopisywania danych bądź nadpisywania (plik zostanie wyczyszczony zaraz po otwarciu

```
if (!f.open(QIODevice::WriteOnly|QIODevice::Append))
```

Klasa QIODevice

- `QFile` dziedziczy po `QIODevice`
- Konstruktory dla `QTextStream` oraz `QDataStream` jako argument przyjmują również wskaźnik na `QIODevice`
- Istnieje szereg implementacji klasy (abstrakcyjnej) `QIODevice`
  - `QBuffer` – do odczytu i zapisu w buforach
  - `QextSerialPort` – do komunikacji przez RS232
  - `QAbstractSocket` – baza dla klas dotyczących połączeń z wykorzystaniem TCP, SSL i UDP
  - `QProcess` – do odczytu i zapisu standardowego wejścia/wyjścia danego procesu

Dowiązanie strumieni do pliku

- Metody read i write są niewygodne w wielu sytuacjach, szczególnie przy obsłudze bardziej złożonych typów
- Współczesne podejście zakłada używania operatorów strumieniowych
- Qt udostępnia dwa rodzaje operatorów strumieniowych
  - Do obsługi plików tekstowych
  - Do obsługi plików binarnych

QTextStream

- Klasa `QTextStream` obsługuje zapis i odczyt danych z plików tekstowych
- Klasa umożliwia
  - Określenie strony kodowej zapisu pliku
  - Określenie ilości linii i słów
  - Rozpoznawanie i przetwarzanie liczb

Zapis do strumienia tekstowego

- Do zapisu wykorzystuje się operator << (tak jak w STL)

```
QFile f(...);
if(!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QTextStream out(&f);
out << "Primes: " << qSetFieldWidth(3) << 2 << 3 << 5 << 7 << endl;
```

Results in:

```
Primes:   2  3  5  7
```

Odczyt za pomocą strumieni tekstowych

- Możliwy jest odczyt pliku linia po linii

```
QTextStream in(&f);
while(!f.atEnd())
    qDebug("line: '%s'", qPrintable(in.readLine()));
```

- Można również odczytywać ciągi i liczby

```
QTextStream in(&f);
QString s;
int i;
in >> s >> i;
```

- Metoda `atEnd` umożliwia sprawdzenie, czy osiągnięto koniec pliku

Obsługa plików binarnych

- Do obsługi plików binarnych używa się klasy `QDataStream`
  - Umożliwia określenie kolejności bajtów (domyślnie big endian)
  - Wspiera typu proste (wbudowane)
  - Wspiera typy złożone Qt
  - Umożliwia dodanie złożonych typów użytkownika

```
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream ds(&f);
ds << QString("Unicode string data");
```

Strumień danych jako format pliku

- Opieranie formatu pliku na klasie `QDataStream` wymaga pamiętania o następujących sprawach
  - Wersja formatu – jako że są zapisywane obiekty Qt konieczne jest przechowywanie informacji o wersji w której są zapisywane. Używając `QDataStream::setVersion` można wymusić konkretną wersję serializacji.
  - Informacja o typach – Qt nie dodaje informacji o typie danych, stąd należy zapisywać i odczytywać dane w określonej kolejności.
  - Kolejność bajtów – Dane zapisywane są w systemie big endian domyślnie, ale można to zmienić używając `QDataStream::setByteOrder`.

Strumień danych jako format pliku

```
QFile f("file.fmt");
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream out(&f);
out.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

out << value;
out << text;
out << color;
```

Od wersji Qt 1.0+

Należy dopasować kolejność i typy danych

```
QFile f("file.fmt");
if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QDataStream in(&f);
in.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

in >> value;
in >> text;
in >> color;
```

Można wykorzystać typ QVariant by nie pamiętać kolejności i typów danych.

Strumienie i własne typy danych

- Dla własnych typów danych należy zaimplementować operatory << oraz >>.

```
QDataStream &operator<<(QDataStream &out, const Person &person)
{
    out << person.name();
    out << person.age();
    return out;
}

QDataStream &operator>>(QDataStream &in, Person &person)
{
    QString name;
    int age;
    in >> name;
    in >> age;
    person = Person(name, age);
    return in;
}
```

Odczyt bezpośredni

Strumienie i własne typy danych

- Jeżeli wykorzystywany jest obiekt typu `QVariant` do przechowywania obiektu użytkownika trzeba zarejestrować zaimplementowane operatory:

```
qRegisterMetaTypeStreamOperators<Person>("Person");
```

- QVariant dodaje nazwę typu w swojej wewnętrznej reprezentacji

```
00: 0x00 0x00 0x00 0x7f 0x00 0x00 0x00 0x00    _________
08: 0x07 0x50 0x65 0x72 0x73 0x6f 0x6e 0x00    _Person_
16: 0x00 0x00 0x00 0x06 0x00 0x4f 0x00 0x6c    _____O_l
24: 0x00 0x65 0x00 0x00 0x00 0x2a              e___*
```

Do sprawdzenia:

- Stwórz
  - `Type::Type()` – Publiczny konstruktor domyślny
  - `Type::Type(const Type &other)` – Publiczny konstruktor kopiujący
  - `Type::~Type()` – Publiczny destruktor
  - `QDebug operator<<` – Ułatwia późniejsze odpluskwianie
  - `QDataStream operator<<` i `>>` – Strumieniowanie
- Zarejestruj
  - `Q_DECLARE_METATYPE` – W nagłówku
  - `qRegisterMetaType` – W funkcji main
  - `qRegisterMetaTypeStreamOperators` – W funkcji main