

Programowanie w środowiskach RAD

Język C++ w środowiskach RAD

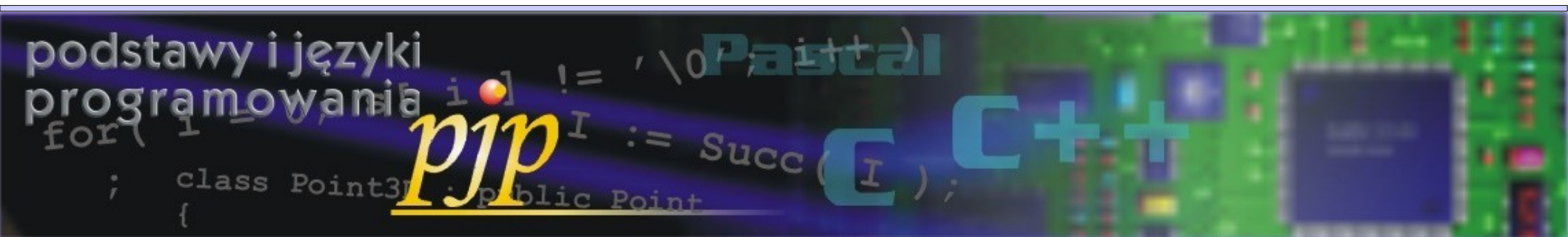
Roman Simiński

roman.siminski@us.edu.pl

www.siminskionline.pl

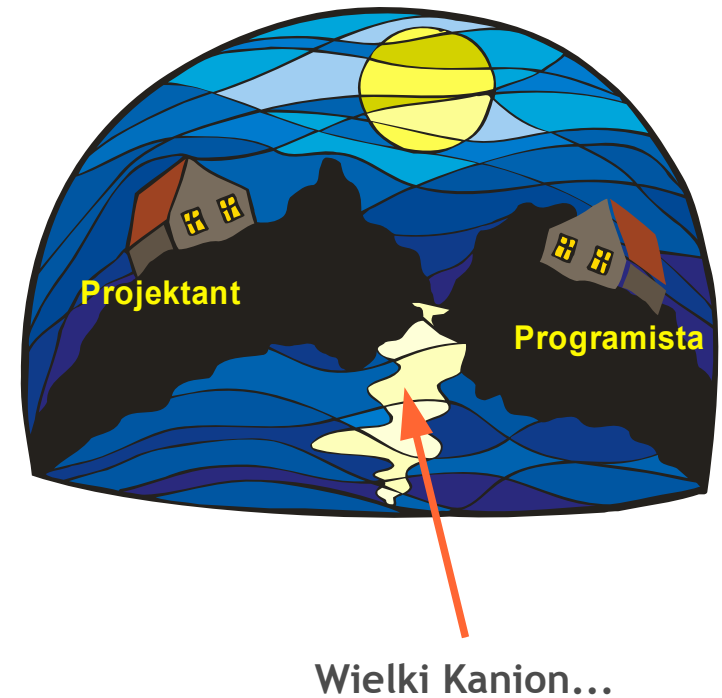
C++ Builder i biblioteka VCL

Wprowadzenie, architektura aplikacji, elementy biblioteki VCL



RAD – Rapid Application Development jako metodyka

- RAD to pragmatyczna *metodyka projektowania systemów informatycznych*, ukierunkowana na szybkie wytworzenie prototypu systemu, podatnego na dalszy rozwój w iteracyjnym cyklu projektowania.
- RAD jako pojęcie zostało zaproponowane przez Jamesa Martina in 1991. Zaproponowana przez niego metodyka pozwalała właśnie na *iteracyjny rozwój* projektu i szybkie wytworzenie *prototypu*.
- RAD jest odpowiedzią na metodyki projektowania i programowania strukturalnego, oparte na *modelu wodospadu* (np. SSADM).
- Rygorystycznie przestrzegany brak nawrotów pomiędzy kolejnymi etapami oraz długo czas oczekiwania na działający system powodował zarówno problemy *projektowe* jak i *komercyjne*.

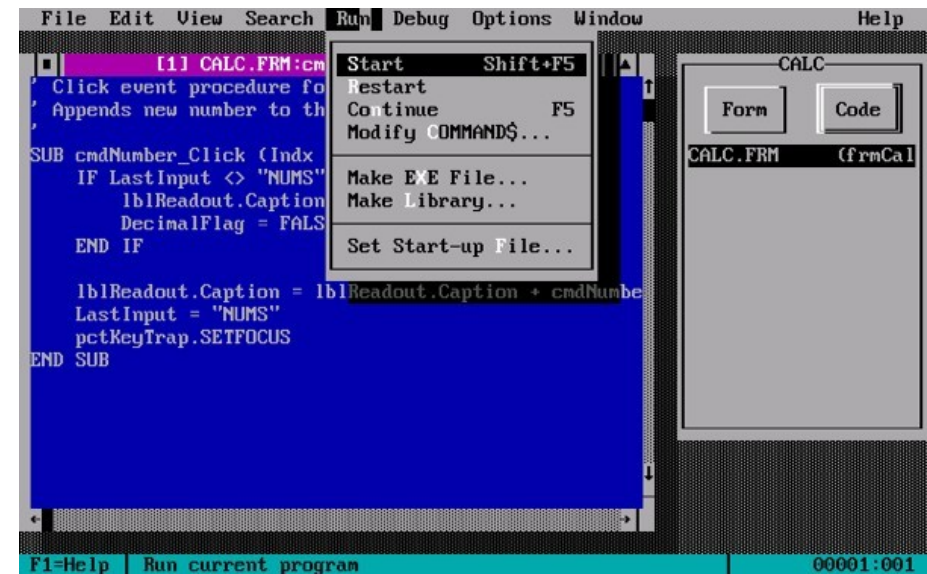
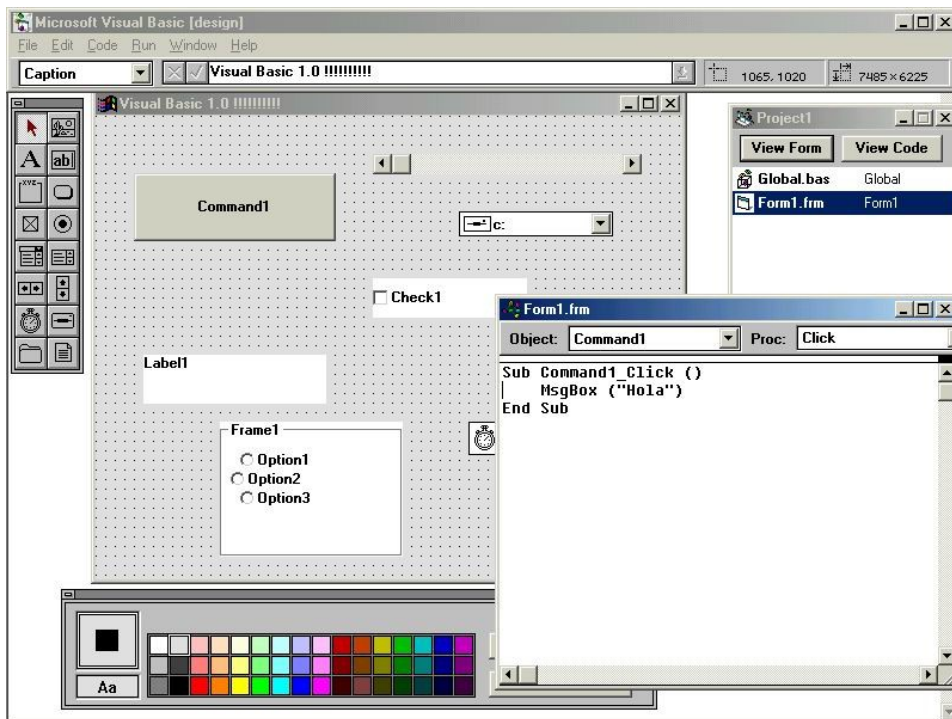


RAD — Rapid Application Development jako narzędzie

- Praktycznie wykorzystanie koncepcji RAD spowodowało rozwój środowisk narzędziowych pozwalających na szybkie prototypowanie aplikacji.
- Środowiska programistyczne typu RAD wywodzą się z klasycznych *zintegrowanych środowisk programistycznych* (IDE) oraz oparte są na *bibliotekach* pozwalających łatwo kreować i wykorzystywać *GUI* oraz dostęp do *baz danych*.
- Większość środowisk RAD bazuje na GUI wykorzystującym *programowanie sterowane zdarzeniami*.

RAD – Rapid Application Development jako narzędzie

- Pierwszym popularnym środowiskiem o cechach RAD był *VisualBasic* firmy Microsoft (koniec lat 80-tych XX w., oficjalna premiera 1991).
- VB zyskał ogromną popularność i odebrał spory rynek twórcom klasycznych kompilatorów i środowisk IDE zorientowanych na kod – m.in. firmie Borland.

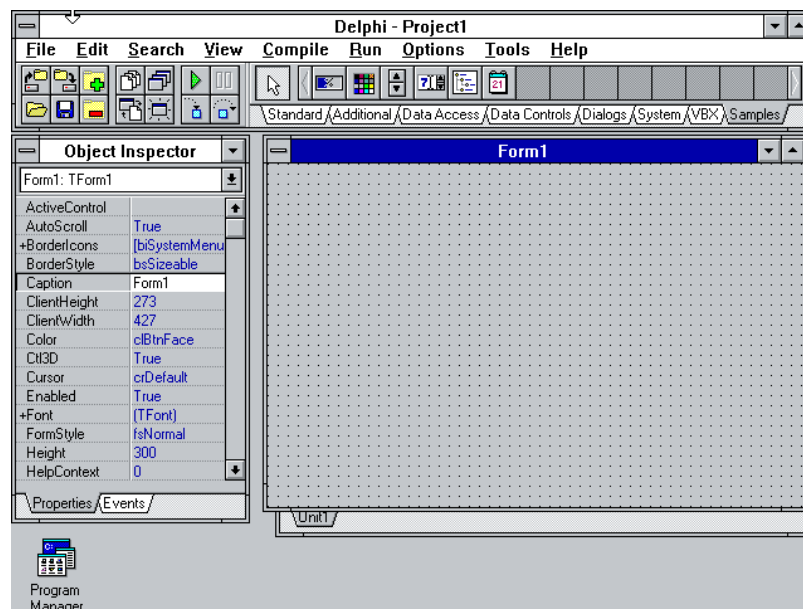


Wojna środowisk RAD

- Początek lat 90-tych XX w. to dominacja firmy *Borland* w zakresie narzędzi do programowania dla komputerów PC.
- *Borland* dominuje na rynku kompilatorów dla języka Pascal, na rynku języka C i C++ konkuruje z firmami *Microsoft*, *Watcom*, *Zortech*.
- Ówczesne IDE są ukierunkowane na edycję kodu, biblioteki obsługi GUI i dostępu do baz danych wymagają trudnego i uciążliwego kodowania.
- *VisualBasic* niejako bocznymi drzwiami wchodzi na rynek i zabiera jego pokaźną część.
- Firma *Microsoft* mocno ten produkt promuje — dzięki niemu lawinowo powstają aplikacje dla środowiska *Windows*, w tym sporo prostych ale grywalnych gier.
- Firma *Borland* chce odzyskać utracony rynek, na początku lat 90-tych XX w. powstaje projekt *VBK* — *Visual Basic Killer*.

VBK i Delphi

- Efekt projektu VBK miał nazywać się *AppBuilder*, ale nazwę tę wcześniej wykorzystuje Novell (*Visual AppBuilder*).
- Nazwa Delphi wywodzi się ponoć od stwierdzenia „*If you want to talk to [the] Oracle, go to Delphi*” i dwuznacznie nawiązuje do firmy Oracle — czołowego producenta serwerów baz danych.
- *Delphi 1* pojawia się na rynku w walentynki 1995 r. Od tamtego momentu sukcesywnie pojawiają się nowe, coraz lepsze wersje pakietu.

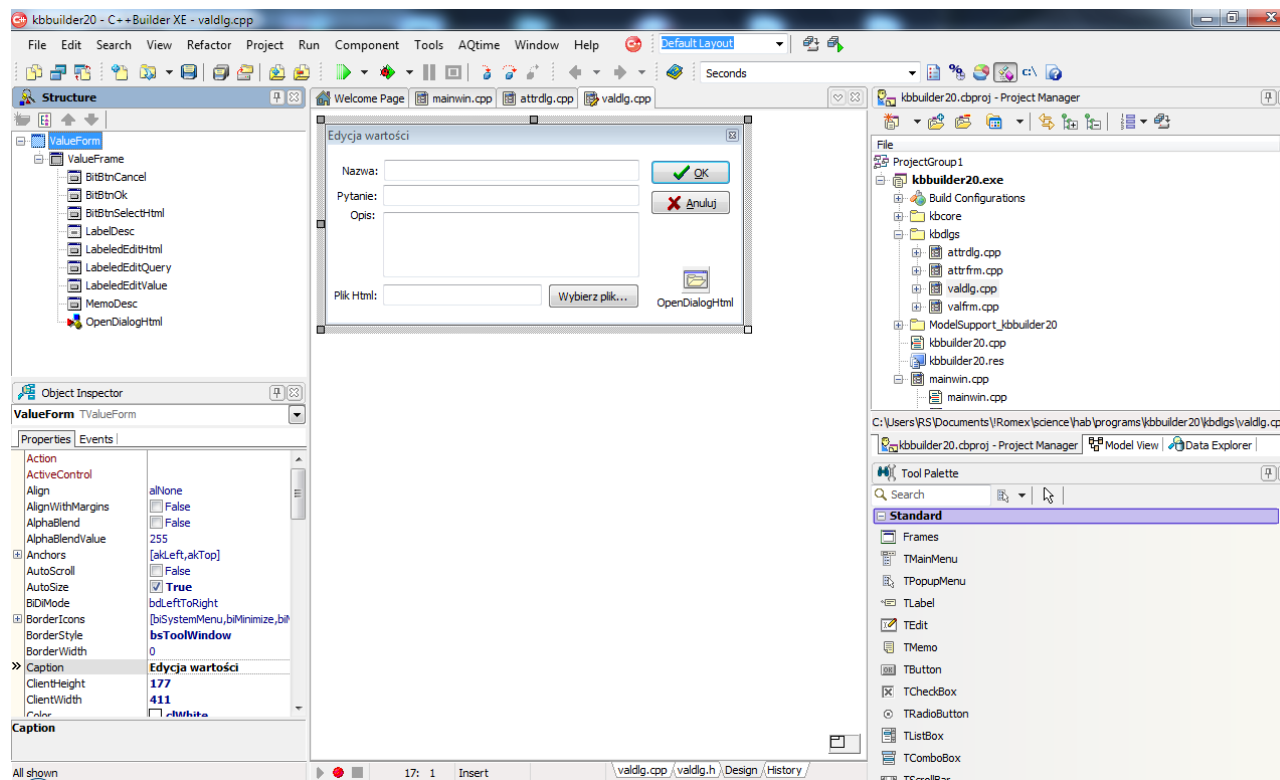


Od VBK Delphi i C++ Buildera

- Delphi bazuje na języku Pascal, wprowadzając jego obiektowy dialekt *ObjectPascal*, który jest aktualnie standardem *de facto* w zakresie języka *Pascal*. Co zapewne mocno nie podoba się jego twórcy — Niklausowi Wirthowi.
- Podstawą *Delphi* jest wizualnie zorientowane środowisko IDE, ściśle zintegrowane z biblioteką *VCL* — *Visual Component Library*.
- Z pewnym opóźnieniem do *Delphi* powstaje pakiet *C++ Builder*, który wykorzystuje te same elementy IDE oraz bibliotekę *VCL*.
- Przez długie lata twórcy *Delphi* najpierw publikują nowe wersje tego pakietu a z pewnym opóźnieniem równoważne wersje *C++ Buildera*.
- W 2006 pojawia się *Borland Developer Studio 2006*, łączące w sobie *Delphi*, *C++ Buildera* i *C# Buildera*.

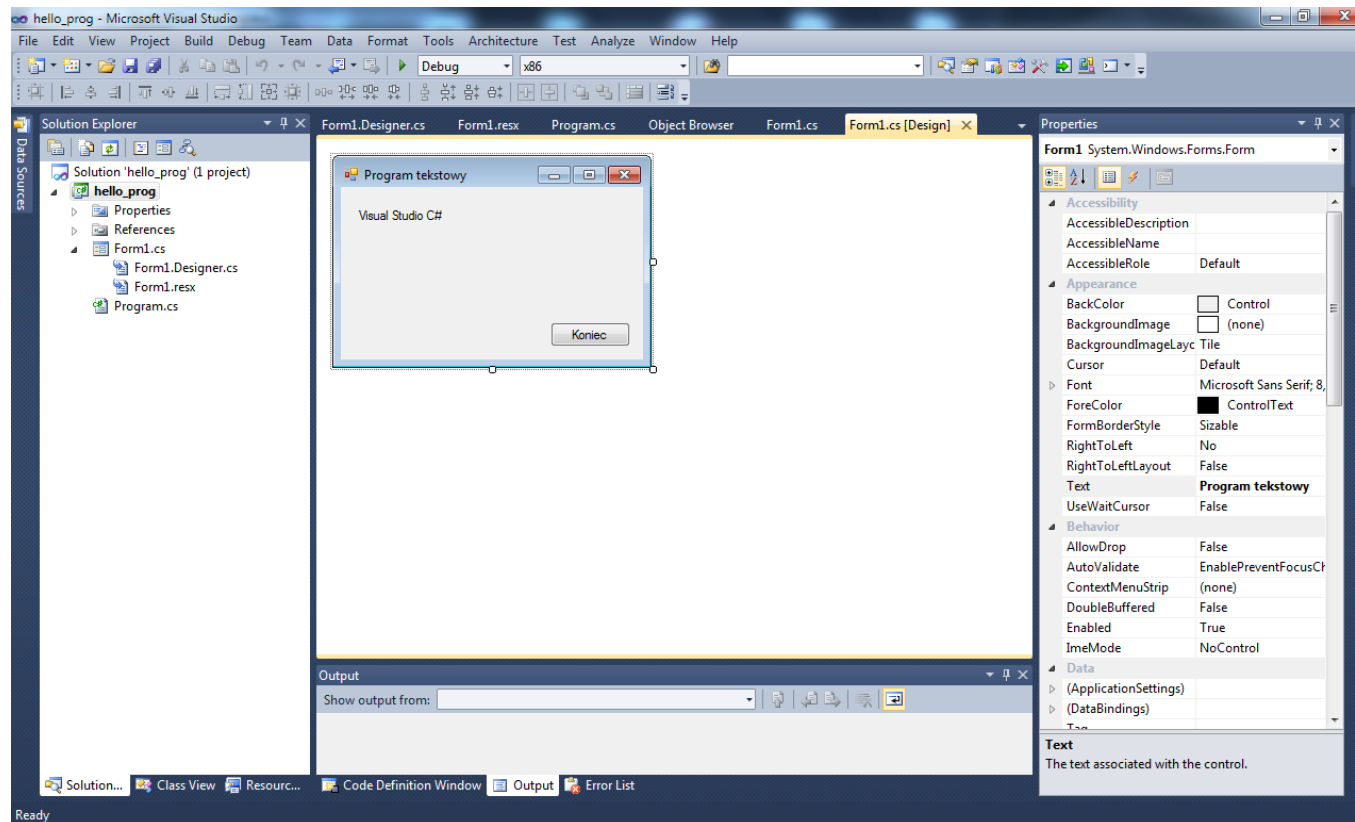
RAD Studio

- W 2010 dostępne jest *RAD Studio*, integrujące *Delphi* i *C++ Buildera XE*.
- Na początku 2011 pojawia się *Delphi* i *C++ Builder Starter* — uproszczona wersja *C++ Buildera XE*.



Pętla: VB, VBK, Borland Developer Studio, C# i Visual Studio

- Na marginesie: Twórcą języka C# oraz biblioteki komponentów dla tego języka jest *Anders Hejlsberga*, który wcześniej pracował zespołach rozwijających *Turbo Pascal*, *Delphi* i *Visual J++*.
- C# początkowo nazywał się *Cool* — „*C-like Object Oriented Language*”.



Jądro RAD Studio: VCL — Visual Component Library

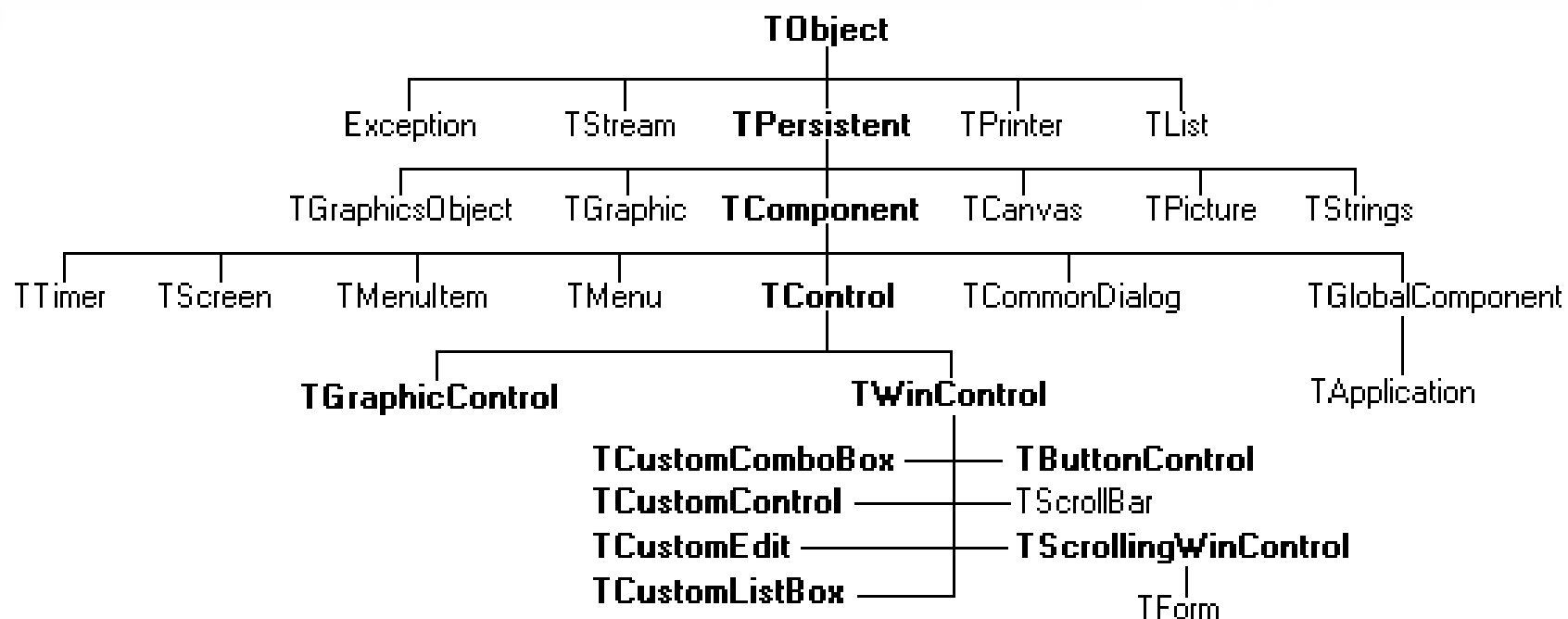
Visual Component Library — wersja podstawowa

- Oferuje dużą liczbę wizualnych i niewizualnych komponentów pozwalających na budowanie natywnych interfejsów użytkownika dla środowiska Windows.
- Zawiera zbiór standardowych elementów sterujących interfejsu (przyciski, menu, listy itd.) jak również zestaw komponentów rozszerzonych nie występujących bezpośrednio w zestawie elementów sterujących.
- Oferuje obsługę *akcji*, pozwalających centralizować przetwarzanie w aplikacji.
- Elementy sterujące wrażliwe na dane — przeznaczone do realizacji aplikacji wykorzystujących bazy danych.
- W ramach biblioteki VCL dostępna jest duża liczba klas niewizualnych, służących m.in. do zarządzania kolekcjami obiektów.

Elementy VCL – obiekty


- **Klasy** — definicje klas stanowiących szablon dla obiektów.
- **Obiekty** — egzemplarze klas zdefiniowanych w VCL, składające się z *metod* (funkcji składowych), *właściwości* oraz *zdarzeń*. Właściwości reprezentują dane obiektu, metody operacje realizowane przez obiekt, zdarzenia warunki na które obiekt odpowiada wykonaniem akcji. Wszystkie obiekty dziedziczą właściwości klasy *TObject*.
- **Komponenty** — to obiekty posiadające wizualną reprezentację, którymi można manipulować w czasie projektowania aplikacji. Wszystkie komponenty dziedziczą właściwości z klasy *TComponent*.
- **Podprogramy** — globalne podprogramy (procedury i funkcje) pochodzące z biblioteki VCL i RTL. Nie są częściami klas, są jednak intensywnie wykorzystywane z metod obiektów.
- **Stałe i zmienne globalne** — predefiniowane stałe lub zmienne, z których można korzystać w programie bez ich definiowania (np. *Application*).

Przykładowy fragment hierarchii klas w VCL



Klasa TForm

- Klasa *TForm* reprezentuje standardowe okno aplikacji.
- Dla każdego okna tworzonego w ramach aplikacji definiowana jest jego własna klasa, dziedzicząca właściwości z klasy *TForm*. Klasy domyślnie otrzymują nazwy *TForm1*, *TForma2*, itd.
- Formy (okna) pochodne od *TForm* reprezentują zarówno główne okno aplikacji jak i okna dialogowe oraz okna MDI.
- W typowych sytuacjach każde okno tworzone z wykorzystaniem VCL jest obiektem klasy pochodnej od *TForm*.

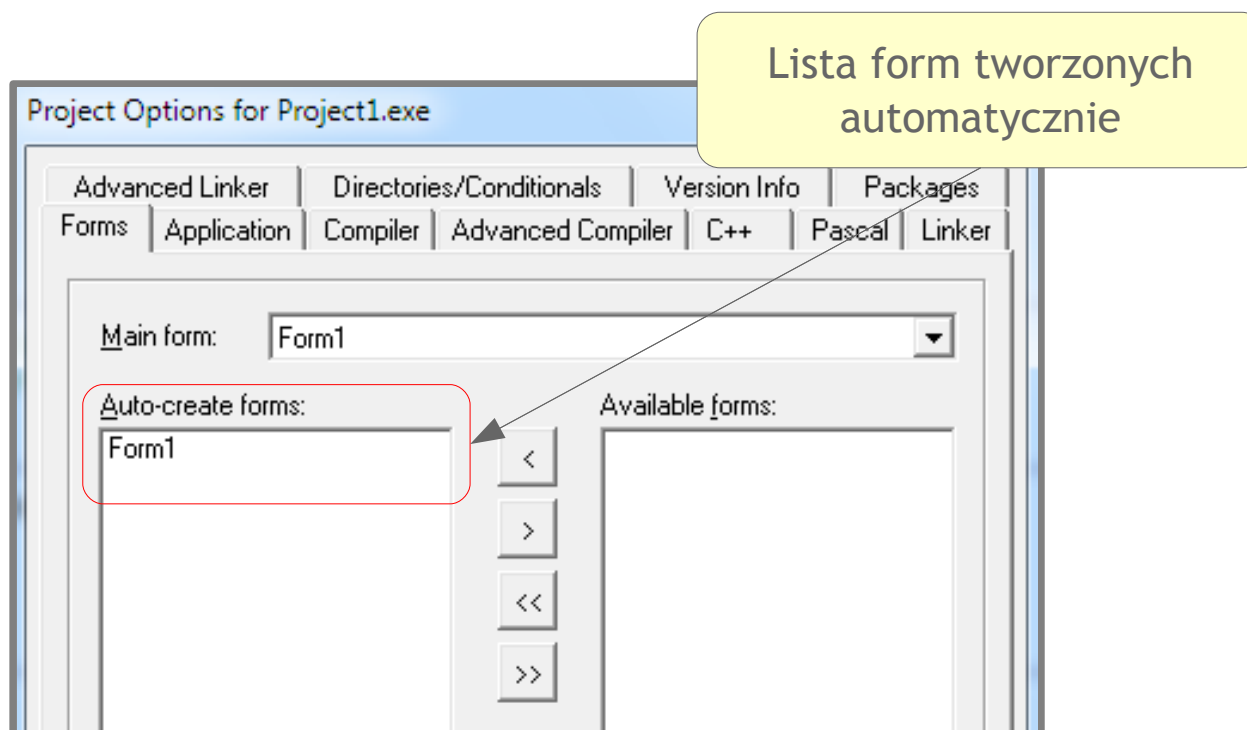


A red arrow points from a box labeled "Dziedziczenie" (Inheritance) to the `TForm1` class name in the code snippet below.

```
class TForm1 : public TForm
{
    __published:    // IDE-managed Components
    private:        // User declarations
    public:          // User declarations
    __fastcall TForm1(TComponent* Owner);
};
```

Klasa TForm

- Forma zwykle jest *właścicielem* obiektów reprezentujących elementy występujące w oknie — przycisków, linii edycyjnych, list itp.
- Obiekty reprezentujące formy są domyślnie tworzone *automatycznie* i żyją przez cały czas wykonania programu. Programista może jednak powoływać obiekty życia dynamicznie, kontrolując czas ich życia.



Triada opisu okna

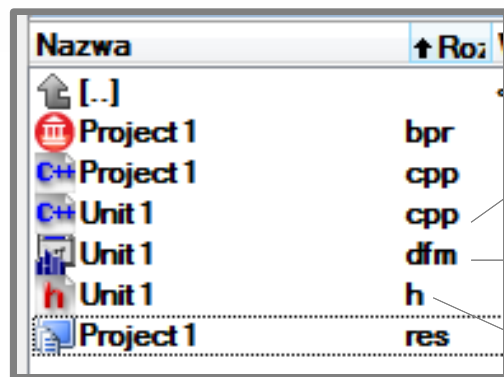
- Każde okno opisane jest własną klasą — jej definicja zapisywana jest automatycznie w *pliku nagłówkowym* (.h).
- Definicje metod klasy okna zapisywane są w *pliku implementacyjnym* (.cpp)
- Definicja informacji o oknie ustalonych na etapie jego projektowania zapisywana jest w *pliku definicji formy* (.dfm).

Po utworzeniu nowej aplikacji tworzone są:

- plik projektu,
- plik programu głównego,
- pliki definicji głównego okna aplikacji,
- inne pliki robocze lub tymczasowe oraz pliki zasobów.

Nazwy i rozszerzenia plików zmieniają się pomiędzy kolejnymi wcieleniami pakiety Builder.
W tym opracowaniu zakłada się wykorzystanie pakietu C++ Builder 6.0

Zawartość folderu projektu tuż po pierwszym zapisie

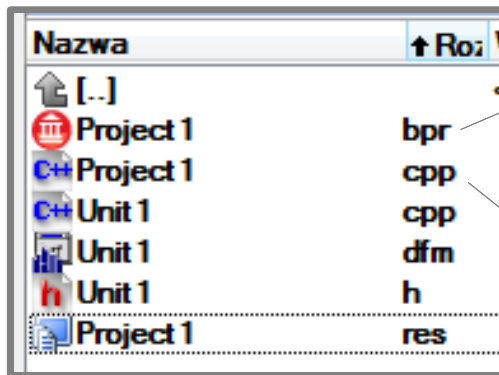


```
TForm1 *Form1;  
  
__fastcall TForm1::TForm1(TComponent* Owner): TForm(Owner)  
{  
}  
}
```

```
class TForm1 : public TForm  
{  
    __published:    // IDE-managed Components  
    private:        // User declarations  
    public:          // User declarations  
    __fastcall TForm1(TComponent* Owner);  
};  
  
extern PACKAGE TForm1 *Form1;
```

```
object Form1: TForm1  
    Left = 192  
    Top = 124  
    Width = 870  
    Height = 500  
    Caption = 'Form1'  
    Color = clBtnFace  
    Font.Charset = DEFAULT_CHARSET  
    Font.Color = clWindowText  
    Font.Height = -11  
    Font.Name = 'MS Sans Serif'  
    Font.Style = []  
    OldCreateOrder = False  
    PixelsPerInch = 96  
    TextHeight = 13  
end
```

Zawartość folderu projektu tuż po pierwszym zapisie



Utworzenie obiektu klasy *TForm1* (sterta), ustawienie wskaźnika *Form1* na utworzony obiekt

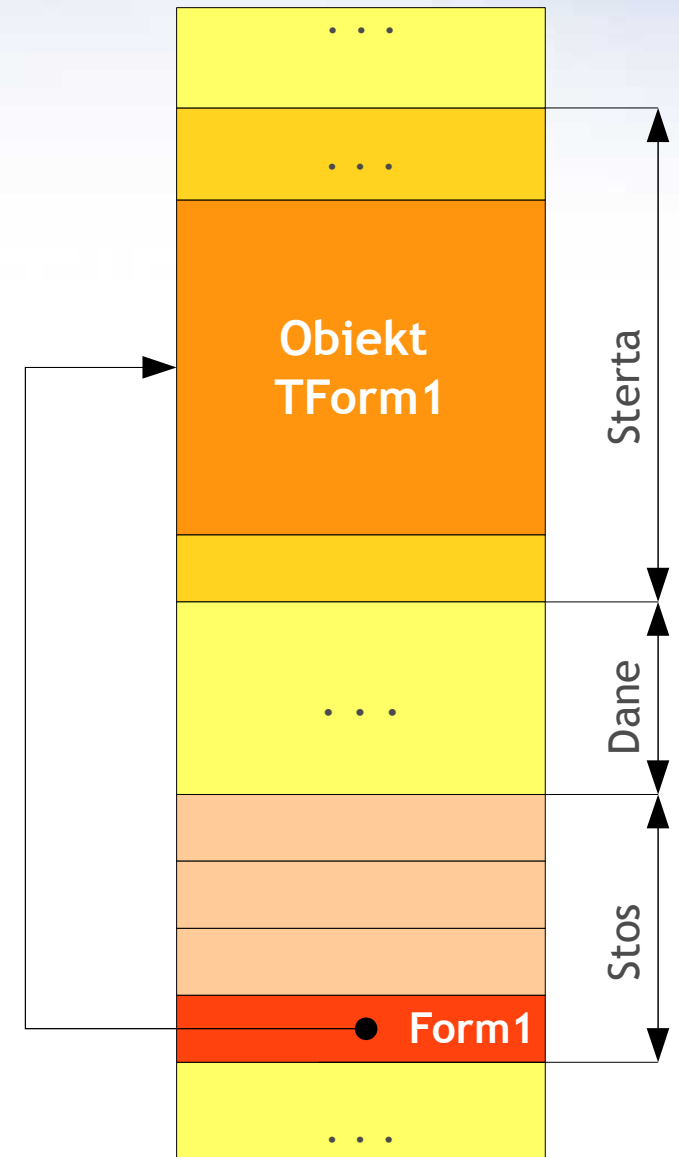
```
<?xml version='1.0' encoding='utf-8' ?>
<!-- C++Builder XML Project -->
<PROJECT>
  <MACROS>
    <VERSION value="BCB.06.00"/>
    <PROJECT value="Project1.exe"/>
    <OBJFILES value="Project1.obj Unit1.obj"/>
    <RESFILES value="Project1.res"/>
    . . .
  </MACROS>
</PROJECT>
```

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
    Application->Initialize();
    Application->CreateForm(__classid(TForm1), &Form1);
    Application->Run();
  }
  catch (Exception &exception)
  {
    Application->ShowException(&exception);
  }
  . . .
  return 0;
}
```

Odwołania za pośrednictwem zmiennych wskaźnikowych

- W systemie C++ Builder komponenty „obsługiwane” przez środowisko przyjmują postać obiektów dynamicznych, lokowanych na sterce programu i dostępnych za pośrednictwem wskaźników.
- Zwykle to kod generowany przez środowisko jest odpowiedzialny za tworzenie i likwidowanie obiektów takich, programista w typowych sytuacjach nie musi się tym zajmować.

```
. . .  
Application->CreateForm(__classid(TForm1), &Form1);  
. . .
```



Warsztat programisty – Builder 6.0

Drzewo obiektów

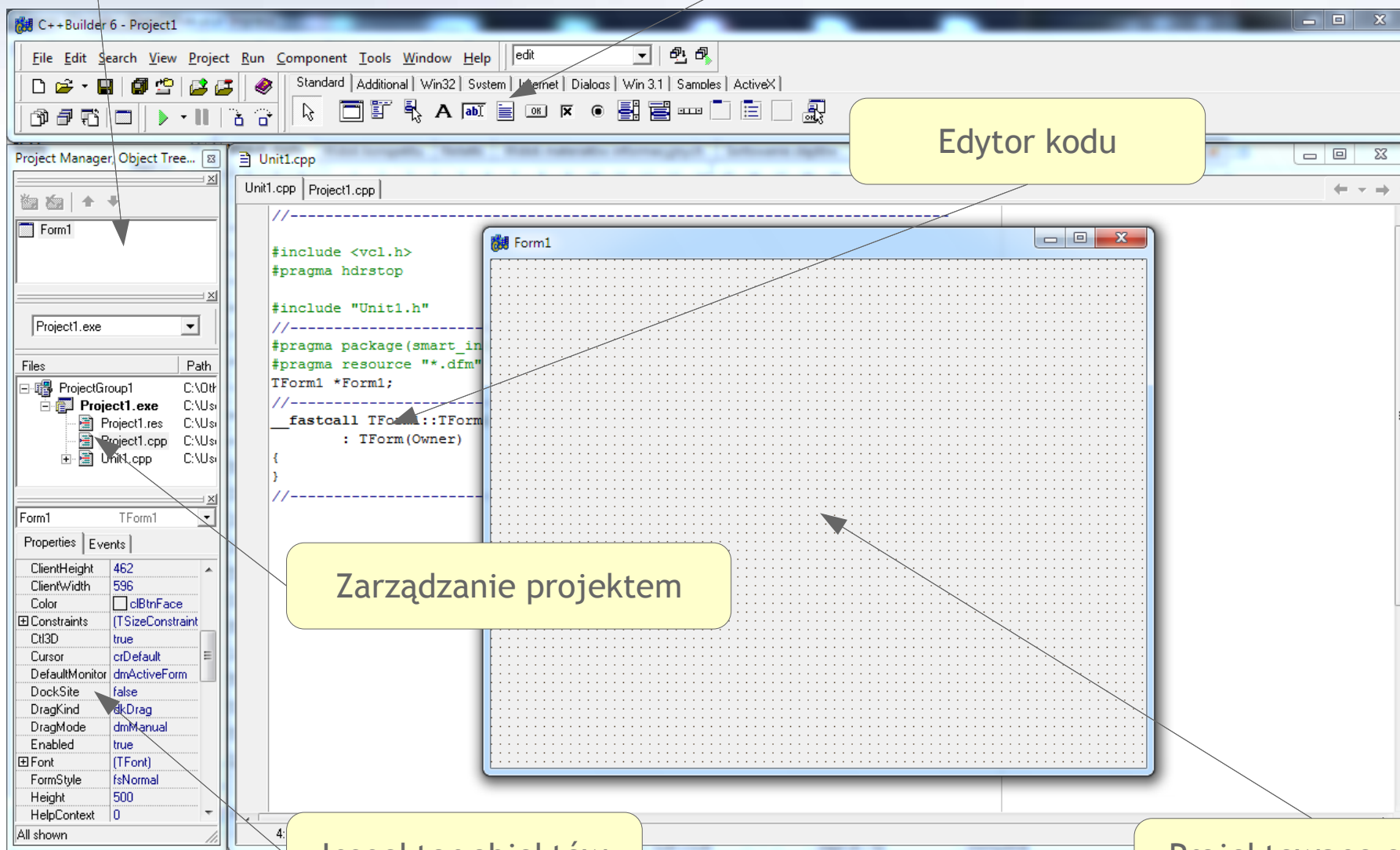
Paleta komponentów

Edytor kodu

Zarządzanie projektem

Inspektor obiektów

Projektowane okno



Warsztat programisty – Builder XE

Drzewo obiektów

Paleta komponentów

The screenshot displays the Builder XE IDE interface with several key components highlighted by yellow callouts:

- Drzewo obiektów (Object Tree):** Located on the top left, it shows a hierarchical view of the project components, including 'Form1' and 'Button1'.
- Object Inspector:** Located on the bottom left, it displays the properties of the selected component (Button1). The 'Caption' property is highlighted with a callout.
- Projektowane okno (Design Window):** The central area shows a visual representation of the form being designed, with a button labeled 'Button1' placed on a grid.
- Edytor kodu (Code Editor):** Located at the bottom, it shows the source code files for the project, including 'Unit1.cpp' and 'Unit1.h'.
- Zarządzanie projektem (Project Manager):** Located on the right side, it displays the project structure, including 'Project1.cbproj', 'Project1.exe', 'Project1.cpp', 'Project1.res', and 'Unit1.cpp'.

Additional visible elements include the main menu bar (File, Edit, Search, View, Window, Help), the toolbar, and the 'Tool Palette' on the right side, which lists various components like 'Standard', 'Additional', 'Win32', 'System', etc.

Inspektor obiektów – ważny element (skrót: F11)

The screenshot illustrates the Object Inspector in a RAD environment. It shows the source code for a form, a visual representation of the form, and the Object Inspector window. A yellow callout box points to the 'Caption' property in the Object Inspector, which is linked to the form's title bar.

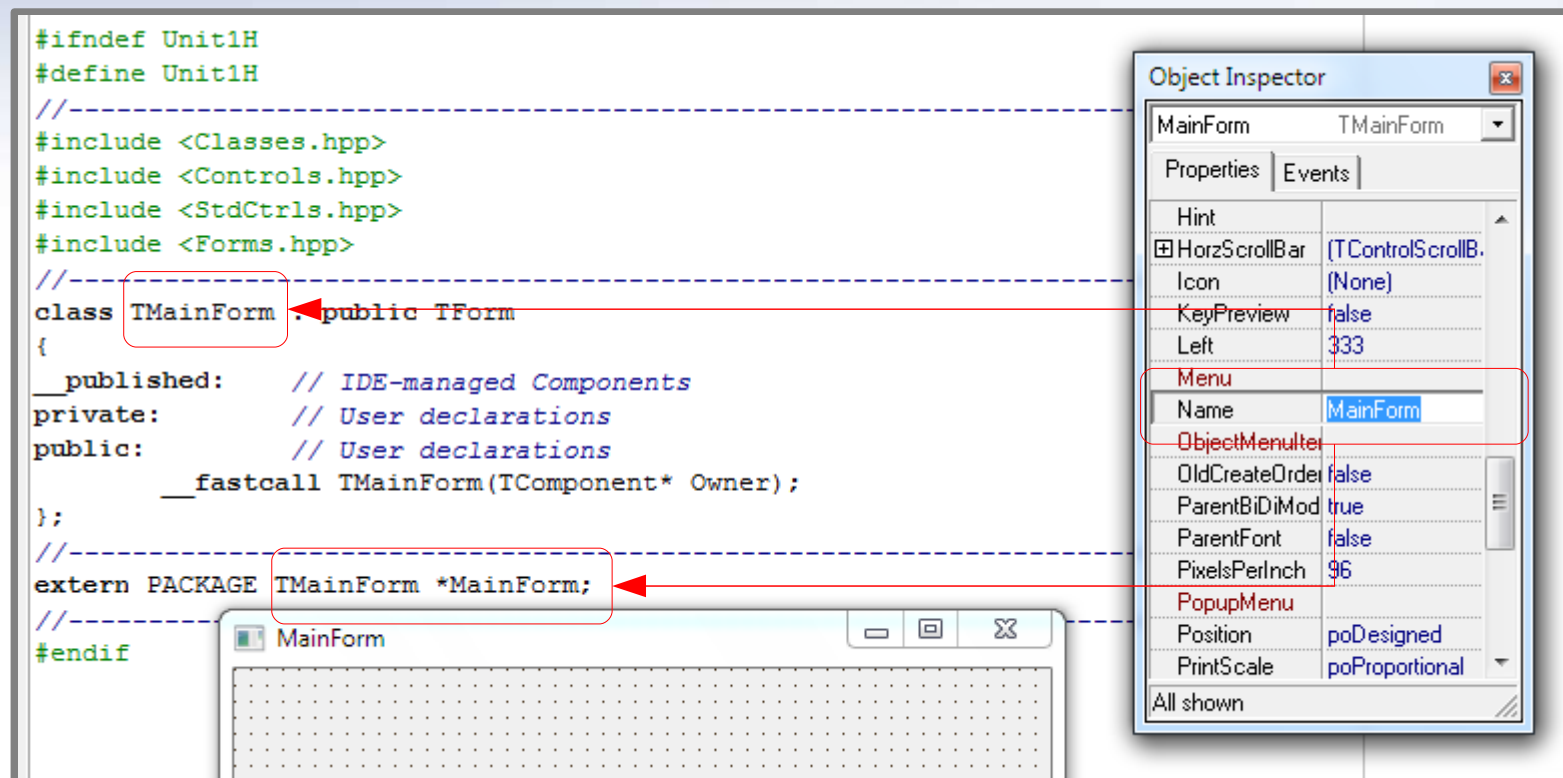
Właściwości okna wskazywanego przez *Form1*

```
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//-----
class TForm1 : public TForm
{
__published:      // IDE-managed Components
private:          // User declarations
public:            // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//-----
extern PACKAGE TForm1 *Form1;
//-----
#endif
```

Object Inspector	
Form1 TForm1	
Properties Events	
Action	
ActiveControl	
Align	alNone
AlphaBlend	false
AlphaBlendValue	255
<input checked="" type="checkbox"/> Anchors	[akLeft,akTop]
AutoScroll	false
AutoSize	false
BiDiMode	bdLeftToRight
<input checked="" type="checkbox"/> BorderIcons	[biSystemMenu,biMaximize,biMinimize]
BorderStyle	bsDialog
BorderWidth	0
Caption	Form1
ClientHeight	171
ClientWidth	408
All shown	

Inspektor obiektów to podstawowe narzędzie pozwalające na sterowanie właściwościami oraz zdarzeniami komponentów. Tego typu element występuje we większości środowisk typu RAD.

Inspektor obiektów – zmiana nazwy obiektu



Uwaga! Nazwy komponentu powinna być zmieniana wyłącznie za pośrednictwem właściwości *Name*. Określa ona nazwę pod jaką komponent będzie identyfikowany w kodzie programu. „Ręczna” modyfikacja nazwy w kodzie prowadzi zwykle do problemów. Nowsze wersje pakietu umożliwiają zmianę nazw za pośrednictwem *refaktoringu*.

Komponenty wstawiane do okna a ich reprezentacja w kodzie

The screenshot illustrates the relationship between code, a visual form, and an object inspector in a RAD environment. The code on the left defines a `TMainForm` class with a published component `Button1` of type `TButton`. The form in the center shows a button labeled `Button1` on a grid. The object inspector on the right shows the properties of `Button1`, such as `Default`, `DragCursor`, `Height`, and `Name`. Red arrows connect the `Button1` declaration in the code to the button on the form and the `Button1` entry in the object inspector.

```
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//-----
class TMainForm : public TForm
{
__published:      // IDE-managed Components
    TButton *Button1;
private:          // User declarations
public:           // User declarations
    __fastcall TMainForm(TComponent* Owner);
};
//-----
extern PACKAGE TMainForm *MainForm;
//-----
#endif
```

Object Inspector

Button1 TButton	
Properties	Events
Default	false
DragCursor	crDrag
DragKind	dkDrag
DragMode	dmManual
Enabled	true
Font	(TFont)
Height	25
HelpContext	0
HelpKeyword	
HelpType	htContext
Hint	
Left	304
ModalResult	mrNone
Name	Button1
ParentBiDiMod	true
All shown	

Button1: TButton
Origin: 304, 24; Size: 75 x 25
Tab Stop: True; Order: 0

Komponenty wstawiane na formę reprezentowane są w kodzie za pośrednictwem wskaźników, tworzeniem i usuwaniem obiektów zajmuje się kod biblioteki.

__published – sekcja komponentów zarządzanych via RAD

```
class TForm1 : public TForm
{
    __published:    // IDE-managed Components
```

```
    TButton *Button1;
```

Tę sekcję zarządza środowisko RAD. Od tego łąpy trzymamy z daleka

```
private:           // User declarations
```

Miejsce na prywatne składowe definiowane przez programistę

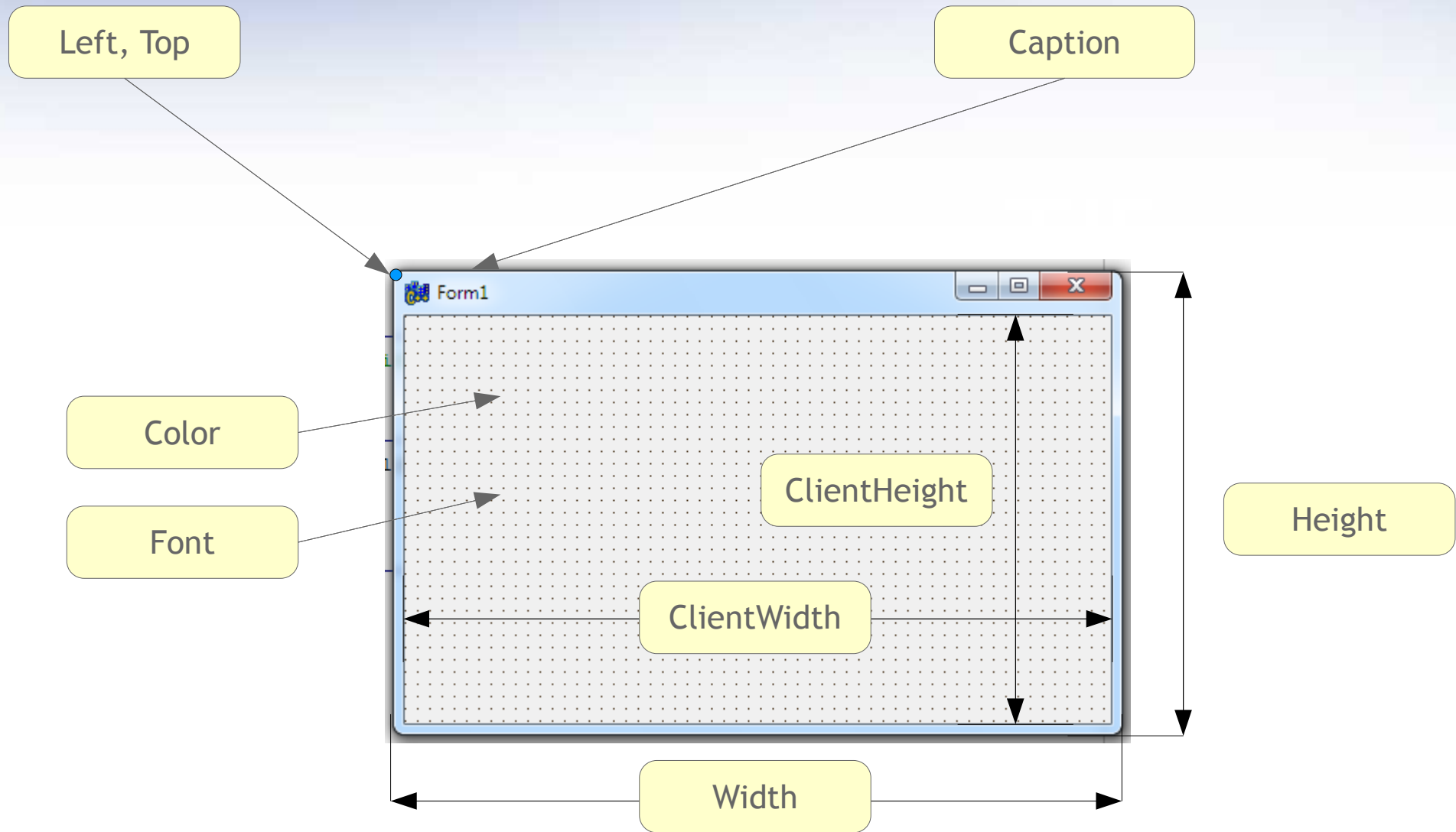
```
public:            // User declarations
    __fastcall TForm1(TComponent* Owner);
```

Miejsce na publiczne składowe definiowane przez programistę

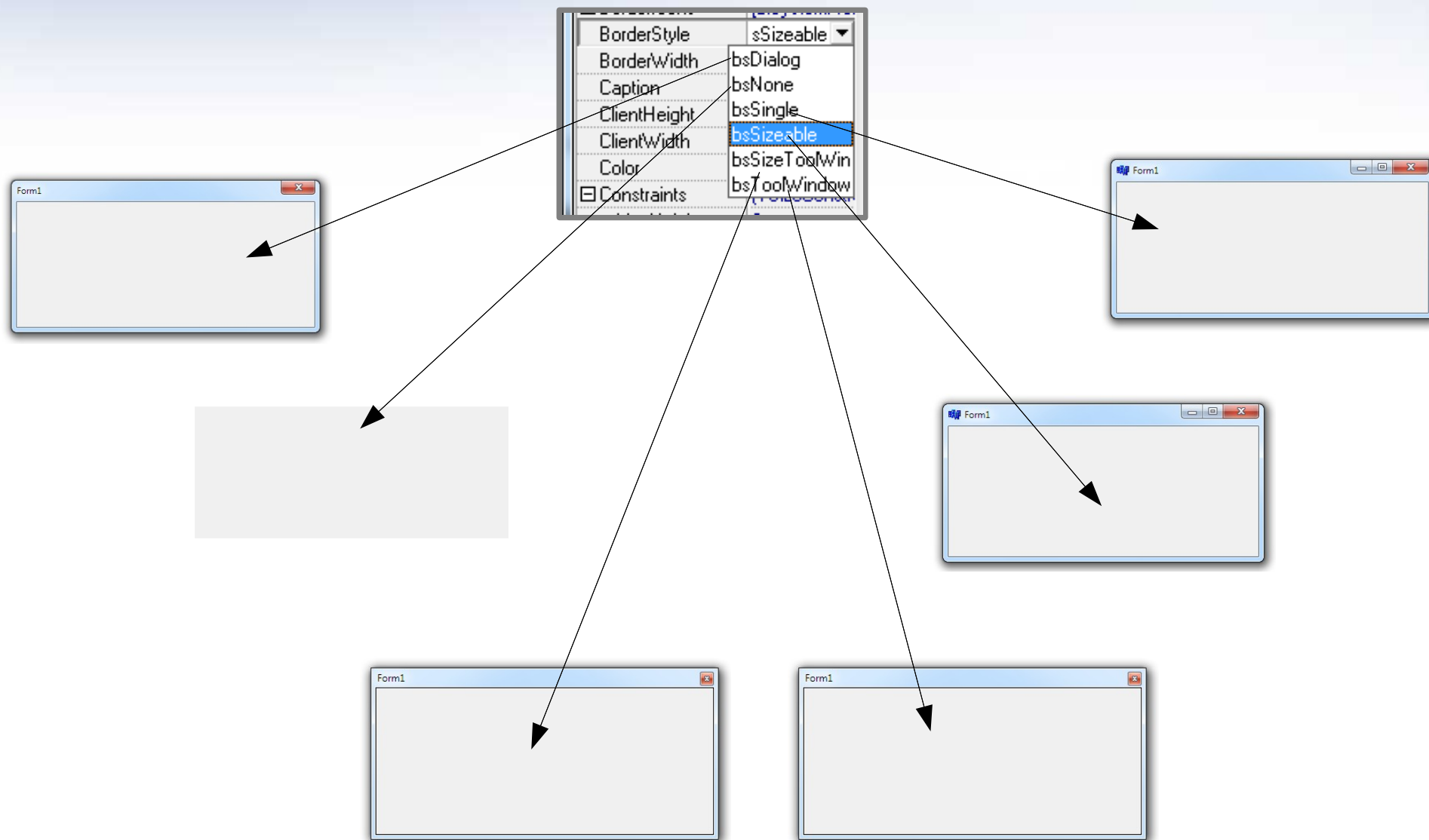
```
};
```

We większości pakietów RAD ich twórcy wprowadzają własne rozszerzenia i modyfikacje, często są to nowe słowa kluczowe (RAD Studio) bądź modyfikacje języka (Visual Studio)

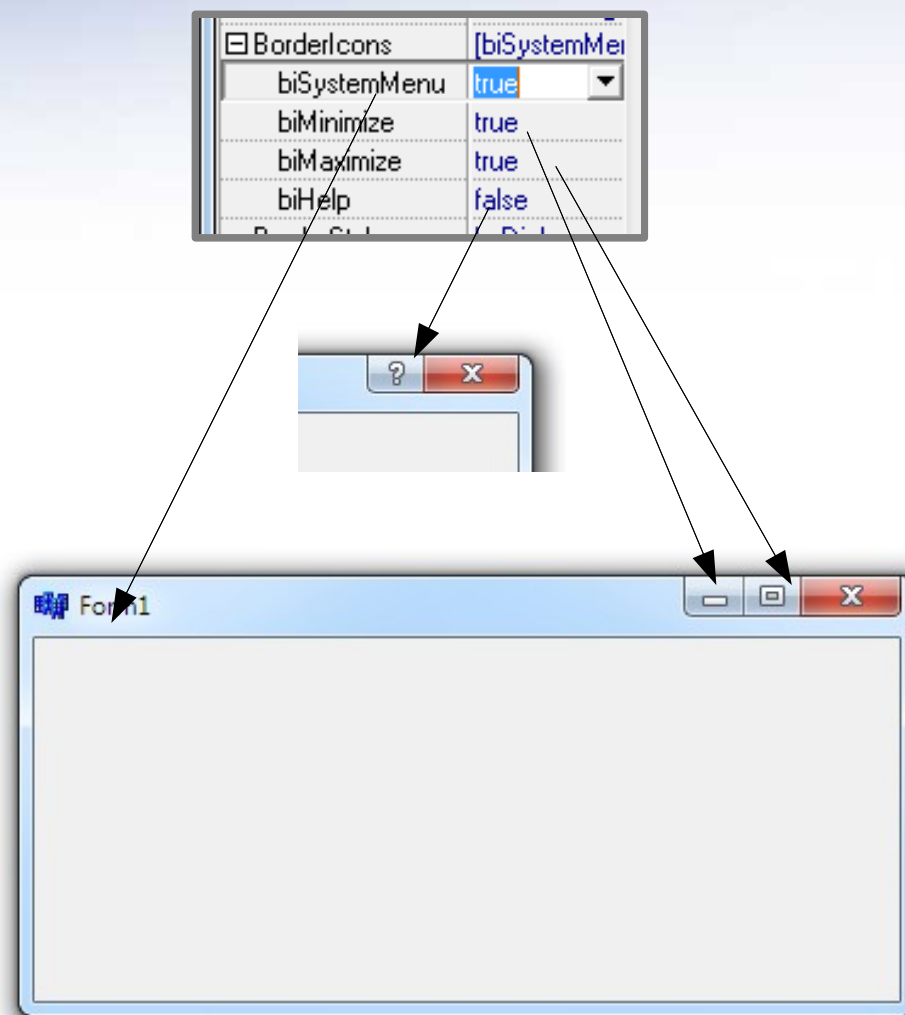
Podstawowe właściwości okna – pochodne klasy TForm



Właściwość BorderStyle

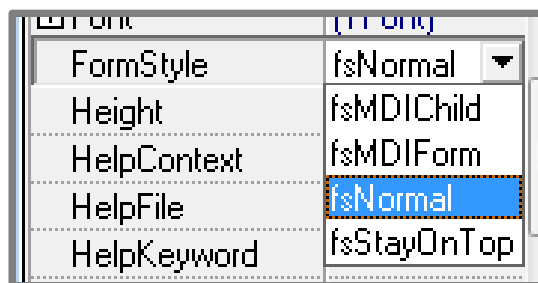
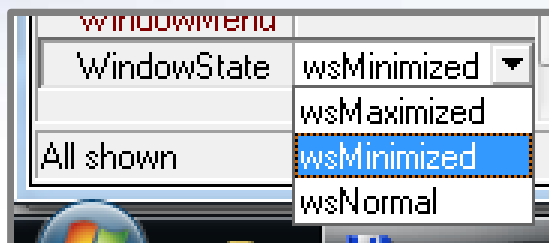
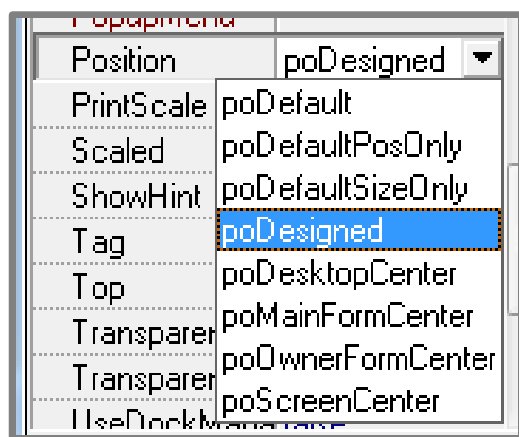


Właściwość BorderIcons



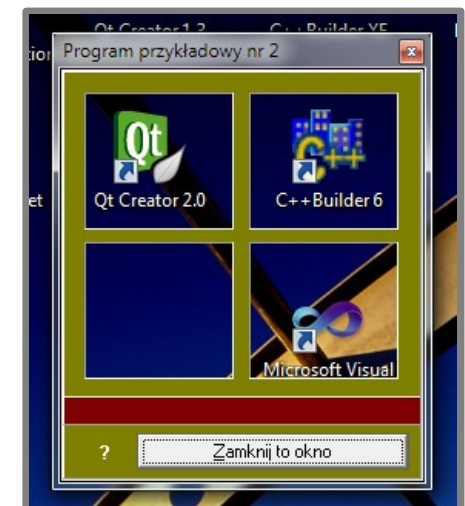
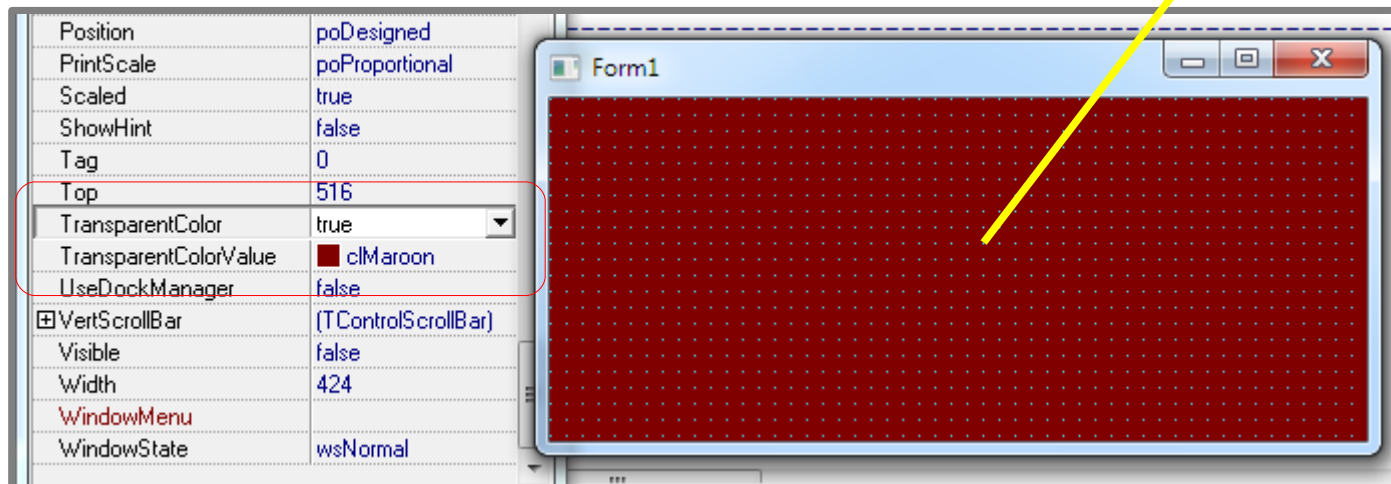
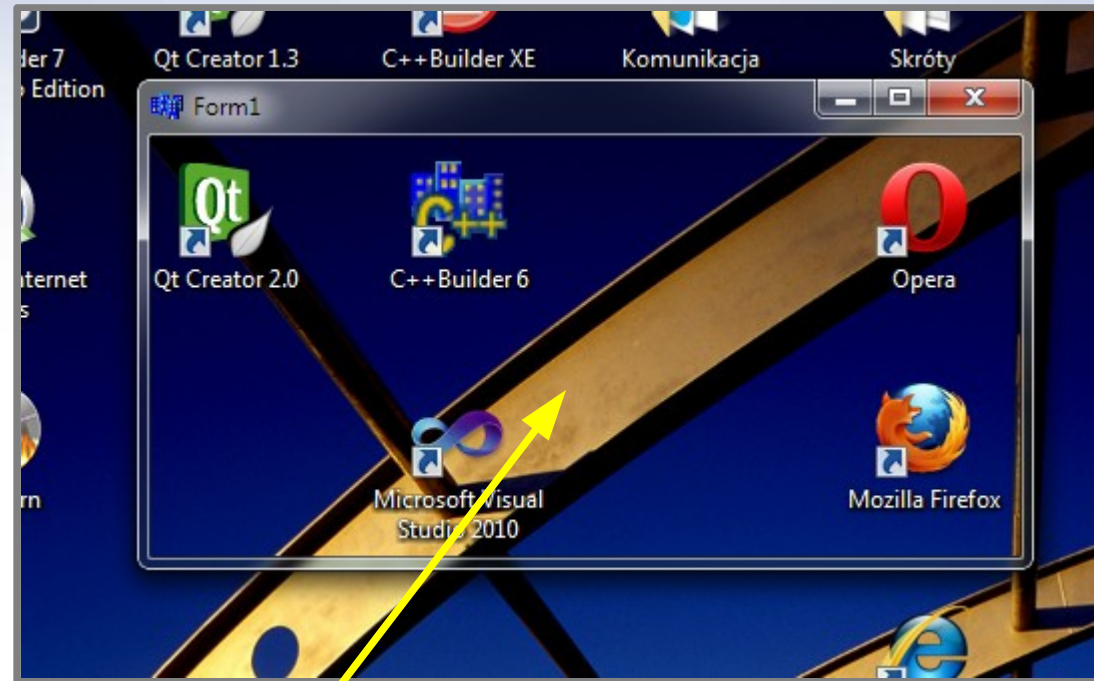
Zestaw dostępnych ikon zależy od właściwości `BorderStyle`

Właściwości: Position, WindowState, Color, FormStyle



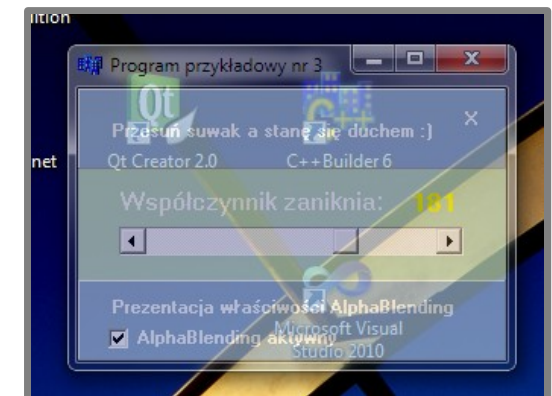
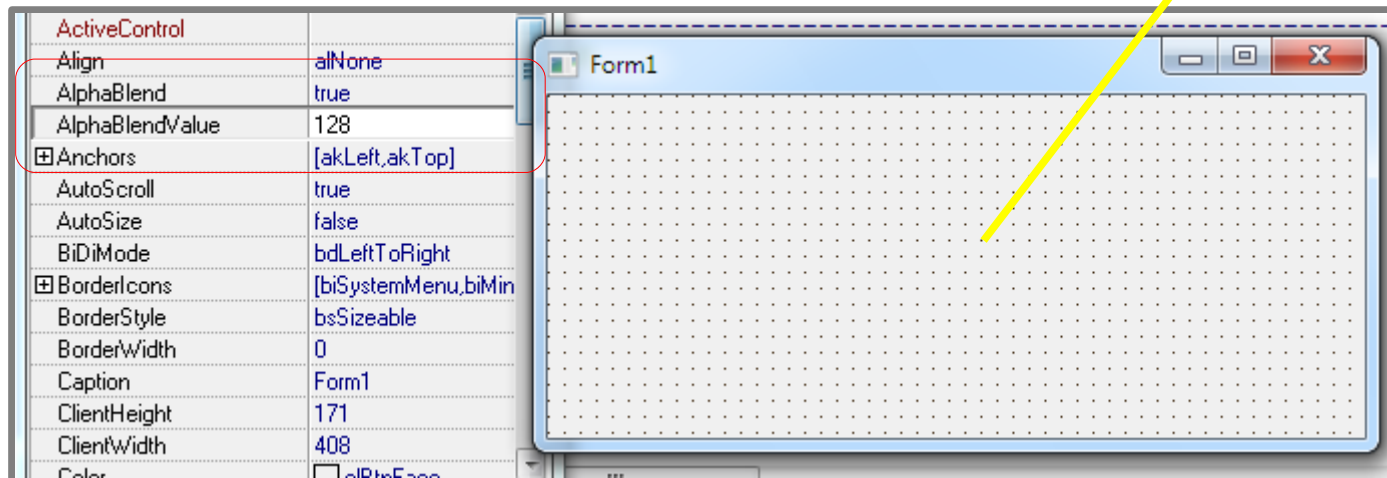
Właściwości: TransparentColor i TransparentColorValue

- Właściwości te pozwalają na określenie koloru, który w ramach okna zostanie uznany za przezroczysty.
- Przezroczystość może dotyczyć całego okna lub wybranych jego elementów.
- Uwaga, przezroczystość nie oznacza *klikalności* przez „dziurę”.



Właściwości: AlphaBlend i AlphaBlendValue

- Właściwość *AlphaBlend* pozwala na włączenie/wyłączenie ustawiania stopnia przezroczystości całego okna.
- Właściwość *AlphaBlendValue* pozwala na ustawianie stopnia przezroczystości całego okna:
 - 0 — pełna przezroczystość,
 - 255 — brak przezroczystości.



Właściwości komponentów a kod programu

Właściwości (ang. *properties*) komponentów

- Można zmieniać za pośrednictwem *Inspektora Obiektów* w trakcie projektowania.
- Właściwości komponentów można zmieniać programowo w trakcie działania programu:

```
Form1->Height = 100;  
Form1->Width = 300;  
Form1->Caption = "Okienko";  
Form1->BorderStyle = bsDialog;
```

Właściwości komponentów wyglądają pozornie jak *pola* obiektów. Właściwości nie są jednak zwykłymi polami – odczyt jak i zmiana wartości właściwości realizowane są przez niejawnie wywoływane funkcje. Dzięki temu, np. zmiana właściwości *Height* powoduje natychmiastową zmianę wysokości okna wyświetlanego na ekranie.

Właściwości komponentów a kod programu

Właściwości (ang. *properties*) komponentów

- Można zmieniać zmienną *Obiektów* w trakcie projektowania.
- Właściwości komponentów można zmieniać programowo w trakcie działania programu:

Tylko gdzie tu wpisywać kod programu... ?

```
Form1->Height = 300;  
Form1->Width = 300;  
Form1->Caption = "Okno";  
Form1->BorderStyle = bsDialog;
```

Właściwości komponentów wyglądają pozornie jak *pola* obiektów. Właściwości jednak nie są zwykłymi polami – odczyt jak i zmiana wartości właściwości realizowane są przez niejawnie wywoływane funkcje. Dzięki temu, np. zmiana właściwości `Height` powoduje natychmiastową zmianę wysokości okna wyświetlanego na ekranie.

VCL a programowanie sterowane zdarzeniami

Wstawienie przycisku klasy *TButton* na formę:

The screenshot illustrates the Delphi IDE environment during the development of a VCL application. The main components visible are:

- Object TreeView, Project Manag...:** Shows the project structure with 'Form1' and 'Button1'.
- Properties Window:** Displays the properties of the selected 'Button1' component. The 'Caption' property is set to 'Koniec'.
- Code Editor:** Shows the source code for 'Unit1.h' and 'Unit1.cpp'. The 'Unit1.h' file contains the declaration of the 'TForm1' class, which inherits from 'TForm'. The 'TForm1' class has a public property 'published: // IDE-managed' and a private member 'TButton *Button1;'. The 'Unit1.cpp' file contains the implementation of the 'TForm1' class, including the 'fastcall TForm1(TForm1)' constructor.
- Form Preview:** A small window titled 'Form1' showing the visual representation of the form. A button labeled 'Koniec' is visible on the form.

Red arrows indicate the flow of information and the relationship between the components:

- An arrow points from the 'Button1' component in the Object TreeView to the 'TButton *Button1;' declaration in the code editor.
- An arrow points from the 'Caption' property in the Properties window to the 'Koniec' text in the code editor.
- An arrow points from the 'Koniec' text in the Properties window to the 'Koniec' text in the code editor.
- An arrow points from the 'Koniec' text in the Properties window to the 'Koniec' text in the form preview.

VCL a programowanie sterowane zdarzeniami

Dla każdego komponentu wizualnego określono zdarzenia (ang. *event*), na jakie może on reagować. Reakcja na zdarzenie polega na przypisaniu do konkretnego zdarzenia procedury jego obsługi (ang. *event handler*).

The screenshot displays the VCL IDE interface. On the left, the Object Inspector shows the 'Events' tab for a 'TButton' component. A list of events is shown, including 'OnClick', 'OnContextPopup', 'OnDragDrop', 'OnDragOver', 'OnEndDock', 'OnEndDrag', 'OnEnter', 'OnExit', 'OnKeyDown', 'OnKeyPress', 'OnKeyUp', 'OnMouseDown', 'OnMouseMove', 'OnMouseUp', 'OnStartDock', 'OnStartDrag', and 'PopupMenu'. A red box highlights this list, and a red arrow points from a text box to it. In the center, the Source Code window shows the C++ code for 'TForm1', including headers and class declarations. On the right, the Form Designer shows a form with a button labeled 'Koniec'. A red box highlights the button, and a red arrow points from it to the text box. A yellow text box at the bottom contains the following text:

Inspektor obiektów prezentuje zdarzenia na które może zareagować obiekt. Można określić, jaka funkcja będzie stanowiła reakcję na zdarzenie.

VCL a programowanie sterowane zdarzeniami

Procedura obsługi zdarzenia przyjmuje postać *funkcji składowej* klasy okna, jej nazwa zawiera *nazwę obiektu* reagującego na zdarzenie oraz *typ zdarzenia*.

The screenshot illustrates the process of connecting a UI event to a code function in the Delphi IDE. On the left, the 'Events' tab for 'Button1' (TButton) is shown, with 'OnClick' selected and 'Button1Click' assigned in the 'Action' column. In the center, the code editor shows the implementation of this function: `void __fastcall TForm1::Button1Click(TObject *Sender) { Close(); }`. To the right, the class declaration for `TForm1` is shown, including the `Button1Click` method signature. Red arrows trace the flow from the IDE's event assignment to the function definition and then to the class declaration. A yellow callout box explains the `Sender` parameter, and a 'Koniec' (End) button is visible in the bottom right corner of the IDE window.

```
class TForm1 : public TForm
{
__published:      // IDE-managed Components
    TButton *Button1;
    void __fastcall Button1Click(TObject *Sender);
private:          // User declarations
public:           // User declarations
    __fastcall TForm1(TComponent* Owner);
};

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Close();
}
```

Parametr *Sender* identyfikuje obiekt reagujący na zdarzenie (odbierający zdarzenie i uaktywniający procedurę obsługi) i czasem bywa bardzo użyteczny.

Koniec

Dziękuję za uwagę

Pytania? Polemiki?
Teraz, albo:
roman.siminski@us.edu.pl