Projektowanie Graficznych Interfejsów Użytkownika

Robert Szmurło

Plan Kursu

- 1 Spotkanie (Użyteczność):
  - Wstęp, Kategorie interfejsów użytkownika, metodyka, przykłady złych projektów (1 h).
  - Web Usability + zalecenia Web Design (1 h)
  - Projektowanie UI + ćwiczenie (2h)
- 2 Spotkanie (Prototypowanie):
  - Projektowanie UI + Projekty widoków
  - HTML + CSS (1h), Delphi (1h),
- 3 Spotkanie (Architektura, Wzorce Projektowe)
  - Architektura uwzględniająca GUI – MVC + Java (1.5h)
  - Prezentacja MS Visual Studio (koncentracja na WinForms+WebForms) (1.5 h).
  - Qt Designer – Jako narzędzie do projektowania wizulanego.
- 4 Spotkanie (Obsługa komunikatów, pakiety, ciekawostki)
  - Mapowanie komunikatów w narzędziach programistycznych (1h).
  - Prezentacja i implementacja przykładu w przenośnym środowisku Qt (1h).
  - Podsumowanie z podkreśleniem wagi Użyteczności (1h).
  - Innowacje: Interfejsy użytkownika 3D, Mac OSX, Microsoft Office 2007 (1h).

Prace Domowe

3 prace domowe

Praca 1
Stworzyć projekt systemu internetowej księgarni.
Stworzyć abstrakcyjny projekt aplikacji opartej na przeglądarce internetowej. (Projekt!)

Praca 2
Stworzyć prototyp zaprojektowanego systemu. (Prototyp)

Praca 3
Zaprojektować obiektową architekturę systemu aby zmaksymalizować wykorzystanie kodu. (Projekt)

Plan szczegółowy

- Spotkanie 4:
  - Mapowanie komunikatów (2h):
    - programowanie zdarzeniowe, komunikacja, pętla obsługi
    - Język C (Win API) + GLUT
    - Delphi
    - MFC C++ oraz .NET
    - Java
    - Qt – inny sposób rozwiązania problemu komunikatów
    - Struts – jako przykład aplikacji WEB
  - Biblioteka Qt, jako przykład przenośnego środowiska z uwzględnieniem wzorców projektowych(1h).
  - Podsumowanie:
    - Podsumowanie użyteczności interfejsów użytkownika.
    - Innowacyjne interfejsy użytkownika: compiz, beryl, Mac OS X
    - Przegląd nowości w interfejsie Microsoft Office 2007.

Na początek: programowanie zdarzeniowe

- Przetwarzanie wsadowe (sekwencyjne) – popularne w dla aplikacji na konsolę tekstową, zwłaszcza w środowisku Unix,
  - komendy, przetwarzanie potokowe, bogate opcje, słaba użyteczność dla użytkowników początkujących,
- Przetwarzanie zdarzeniowe – wykonywanie zadań na żądanie, udostępnienie punktów startowych zadań,
- A 'vi', a Norton Commander? Programowanie zdarzeniowe jest ogólną koncepcją i nie jest związane wyłącznie z graficznymi interfejsami użytkownika. Niemniej programowanie zdarzeniowe stanowi trzon interfejsów graifcznych.

```
szmurlor@nor:~> cdrecord --help
This is wodim, not cdrecord. Don't expect it to behave like cdrecord in any
way, don't refer to it as "cdrecord". Send problem reports to
debburn-devel@lists.alioth.debian.org, don't bother Joerg Schilling with any
problems caused by this application.
Copyright (C) 2006 cdrkit maintainers, (C) 1994-2006 Joerg Schilling

Usage: wodim [options] track1...trackn
Options:
        -version        print version information and exit
        dev=target      SCSI target to use as CD/DVD-Recorder
        gracetime=#     set the grace time before starting to write to #.
        timeout=#       set the default SCSI command timeout to #.
        debug=#,-d      Set to # or increment misc debug level
        kdebug=#,kd=#   do Kernel debugging
        -verbose,-v     increment general verbose level by one
        -Verbose,-V     increment SCSI command transport verbose level by one
        -silent,-s      do not print status of failed SCSI commands
        driver=name     user supplied driver name, use with extreme care
        driveropts=opt  a comma separated list of driver specific options
        -setdropts      set driver specific options and exit
        -checkdrive     check if a driver for the drive is present
        -prcap          print drive capabilities for MMC compliant drives
        -inq            do an inquiry for the drive and exit
        -scanbus        scan the SCSI and IDE buses and exit
        -reset          reset the SCSI bus with the cdrecorder (if possible)
        -abort          send an abort sequence to the drive (may help if hung)
        -overburn       allow to write more than the official size of a medium
        -ignsize        ignore the known size of a medium (may cause problems)
        -useinfo        use *.inf files to overwrite audio options.
        speed=#         set speed of drive
        blank=type      blank a CD-RW disc (see blank=help)
        -format         format a CD-RW/DVD-RW/DVD+RW disc
        formattype=#    select the format method for DVD+RW disc
        ts=#            set maximum transfer size for a single SCSI command
        -load           load the disk and exit (works only with tray loader)
        -lock           load and lock the disk and exit (works only with tray loader)
        -eject          eject the disk after doing the work
        -dummy          do everything with laser turned off
        -msinfo         retrieve multi-session info for mkisofs >= 1.10
        -toc            retrieve and print TOC/PMA data
        -atip           retrieve and print ATIP data
        -multi          generate a TOC that allows multi session
                        In this case default track type is CD-ROM XA mode 2 form 1 - 2048 bytes
        -fix            fixate a corrupt or unfixated disk (generate a TOC)
        -nofix          do not fixate disk after writing tracks
        -waiti          wait until input is available before opening SCSI
        -immed          Try to use the SCSI IMMED flag with certain long lasting commands
        -force          force to continue on some errors to allow blanking bad disks
        -tao            Write disk in TAO mode. This option will be replaced in the future.
        -dao            Write disk in SAO mode. This option will be replaced in the future.
        -sao            Write disk in SAO mode. This option will be replaced in the future.
        -raw            Write disk in RAW mode. This option will be replaced in the future.
        -raw96r         Write disk in RAW/RAW96R mode. This option will be replaced in the
future.
        -raw96p         Write disk in RAW/RAW96P mode. This option will be replaced in the
future.
        -raw16          Write disk in RAW/RAW16 mode. This option will be replaced in the
future.
        -clone          Write disk in clone write mode.
        tsize=#         Length of valid data in next track
        padsize=#       Amount of padding for next track
        pregap=#        Amount of pre-gap sectors before next track
        defpregap=#     Amount of pre-gap sectors for all but track #1
        mcn=text        Set the media catalog number for this CD to 'text'
        isrc=text       Set the ISRC number for the next track to 'text'
        index=list      Set the index list for the next track to 'list'
```

Sekwencyjne kontra Zdarzeniowe

- Standardowe programowanie sekwencyjne:
  - Program wykonuje pewną czynność a następnie użytkownik odpowiada,
  - **Program kontroluje użytkownika.**
- Programowanie zorientowane zdarzeniowo
  - Powszechne dla systemów okienkowych (graficznych)
  - Użytkownik wykonuje pewną czynność (np. wprowadza dane) a następnie program odpowiada,
  - Użytkownik może wydawać komendy w dowolnym momencie,
  - **Użytkownik kontroluje program,**
  - Rzeczywista kontrola systemu operacyjnego (zarządza komunikatami przesyłanymi między aplikacjami)
  - Model zdecydowanie zalecany w przypadku aplikacji wymagającej znacznej interakcji z użytkownikiem,

Komunikacja

- Komunikaty stanowią trzon działania graficznych interfejsów użytkownika.
  - Jedną z podstawowych zalet komunikatów jest możliwość ich asynchronicznej obsługi, filtracji, rozgłaszania.
  - Komunikaty są w swojej zasadzie działania podobne do działania pakietów sieci komputerowej. (Dlatego np. System graficzny X Window jest oparty na sieci komputerowej.)
  - Komunikacja między kontrolkami na ekranie,
  - Między kontrolkami i formularzem lub pewnym obiektem kontrolującym,
    - Uruchomienie: exec, click, start,
    - Uzyskanie kontroli (focus, onEnter itp)
    - Interakcja z klawiaturą i myszką
    - Wystąpienie błędu
    - Zmiana wartości (przez użytkownika)
    - Komunikaty o tworzeniu kontrolki, usuwaniu, pokazywaniu, chowaniu
    - Zmiana rozmiaru, uaktualnienie widoku
    - Stoper (Timer)
    - Pojawienie się nowego urządzenia, itp...

Obsługa komunikatów

1. Stwierdza do której aplikacji należy wysłać komunikat. (Zazwyczaj dana aplikacja posiada „focus”.

Aplikacja

Start

Pętla obsługi komunikatów

Koniec

System operacyjny

Interpreter komunikatów

Komunikaty:

WM_TEXTCHANGE: Mam imię.
WM_TEXTCHANGE: Mam ocenę.
WM_LMBUTTON: Oblicz średnią.
...
...

Komunikat (message)

Akcje użytkownika

Wprowadza imię.

Wprowadza liczbę.

Klika na „Oblicz średnią”

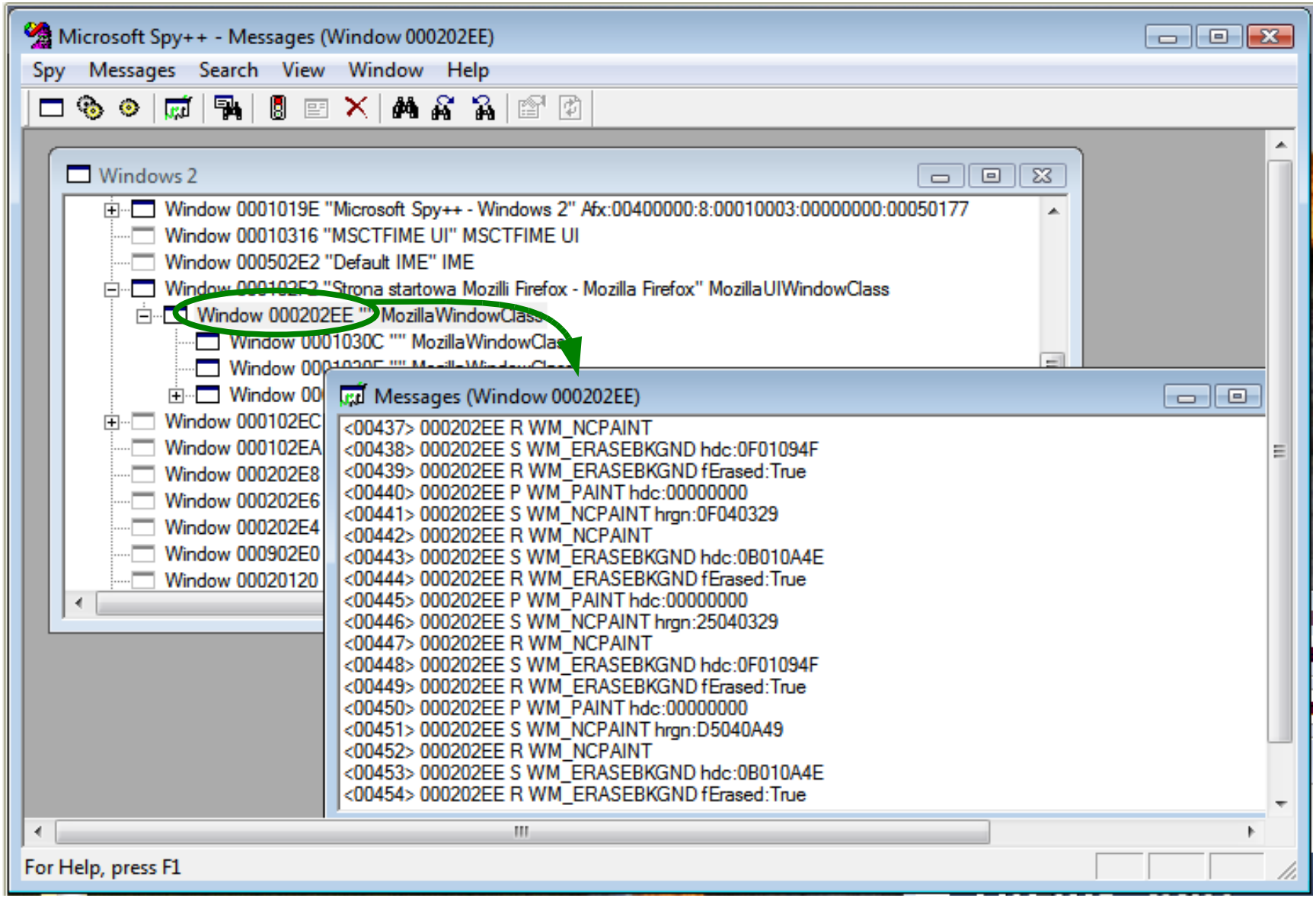
itd. kolejne czynności...

Podsumowanie obsługi komunikatów

- Samą komunikacją zajmuje się system graficzny. On jest odpowiedzialny za dotarcie komunikatu do odbiorcy.
- Programy zajmują się jednie obsługą komunikatów.
  - Zazwyczaj większość komunikatów jest obsługiwana przez biblioteki i przekierowywana do odpowiednich kontrolek graficznych. (*GUI Toolkits*)
  - Zadaniem programisty jest implementacja odpowiednich metod obsługi komunikatów. (*Message handler*)
  - Program nie musi obsługiwać wszystkich komunikatów.
- Wszystkie biblioteki GUI umożliwiają stworzenie własnych sposobów obsługi komunikatów.
- Komunikaty posiadają często argumenty, które są przekazywne przez system operacyjny do metody obsługi komunikatów razem z komunikatem.

Spy++

- Narzędzie monitorowania komunikatów.



Realizacja - Pętla obsługi komunikatów

- Podejście tradycyjne (Win32 API, GLUT):

```
GUIMessage * msg;

while (( msg = waitForNextMessage() ) != null) {
    DispatchMessage( ApplicationHandle, msg );

    if (msg == GUIQuitMessage) {
        PostQuitMessage( ApplicationHandle, msg );
        break;
    }
}
```

- Podejście obiektowe (Qt, MFC, .NET, Java):

```
Application app = new Application(argc, argv);
Window w = new MyWindow();
w.show();

return app.Exec();
```

Program „Hello World” w Win32 API

- Struktura programu wykorzystującego Win32 API w języku C.
  - Charles Petzold, autor popularnych książek o Windows API powiedział: "*The original hello-world program in the Windows 1.0 SDK was a bit of a scandal. HELLO.C was about 150 lines long, and the HELLO.RC resource script had another 20 or so more lines. (...) Veteran C programmers often curled up in horror or laughter when encountering the Windows hello-world program.*"

```c
#include <windows.h>

/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);

/*  Make the class name into a global variable  */
char szClassName[ ] = "WindowsApp";

int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nFunsterStil)

{
    HWND hwnd;               /* This is the handle for our window */
    MSG messages;            /* Here messages to the application are saved */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by windows */
    wincl.style = CS_DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;                 /* No menu */
    wincl.cbClsExtra = 0;                      /* No extra bytes after the window class */
    wincl.cbWndExtra = 0;                      /* structure or the window instance */
    /* Use Windows's default color as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
           NULL,                /* No menu */
           hThisInstance,       /* Program Instance handler */
           NULL                 /* No Window Creation data */
           );

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);

    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 - The value that PostQuitMessage() gave */
    return messages.wParam;
}


/*  This function is called by the Windows function DispatchMessage()  */

LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)                  /* handle the messages */
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            TextOut(hdc, 15, 3, "Hello, world!", 13);
            EndPaint(hwnd, &ps);
            break;
        case WM_LBUTTONDBLCLK:
            MessageBox(NULL, "Hello, world!", "", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;
        default:                       /* for messages that we don't deal with */
            return DefWindowProc(hwnd, message, wParam, lParam);
    }

    return 0;
}
```

dalej...

Kod „Hello World” - część 1

```
#include <windows.h>

/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);

/*  Make the class name into a global variable  */
char szClassName[ ] = "WindowsApp";

int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nFunsterStil)

{
    HWND hwnd;               /* This is the handle for our window */
    MSG messages;            /* Here messages to the application are saved */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by windows */
    wincl.style = CS_DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;                 /* No menu */
    wincl.cbClsExtra = 0;                      /* No extra bytes after the window class */
    wincl.cbWndExtra = 0;                      /* structure or the window instance */
    /* Use Windows's default color as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
```

1. Deklaracja procedury, która będzie obsługiwała nadsyłane komunikaty.

2. Punkt startowy programu, który w przypadku Win32 zawsze nazywał się WinMain

3. Długa i skomplikowana specyfikacja parametrów tworzonego okna.

4. Rejestracja nowego okna z wprowadzonymi parametrami.

5. Wywołujemy funkcję tworzącą instancję zarejestrowanego okna. Tworzymy nowy wątek.

Kod „Hello World” - część 2

```
            NULL,                 /* No menu */
            hThisInstance,        /* Program Instance handler */
            NULL                  /* No Window Creation data */
            );

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);

    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 - The value that PostQuitMessage() gave */
    return messages.wParam;
}


/*  This function is called by the Windows function DispatchMessage()  */

LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)                  /* handle the messages */
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            TextOut(hdc, 15, 3, "Hello, world!", 13);
            EndPaint(hwnd, &ps);
            break;
        case WM_LBUTTONDBLCLK:
            MessageBox(NULL, "Hello, world!", "", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;
        default:                      /* for messages that we don't deal with */
            return DefWindowPro                ssage, wParam, lParam);
    }

    return 0;
}
```

DefWindowProc

1. Pokazujemy stworzone okno na ekranie.

2. Klasyczna pętla komunikatów.

3. Wcześniej zadeklarowana pętla, która obsługuje komunikaty przesyłane do okienka.

WindowProcedure - szczegóły

```
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)                  /* handle the messages */
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            TextOut(hdc, 15, 3, "Hello, world!", 13);
            EndPaint(hwnd, &ps);
            break;
        case WM_LBUTTONDBLCLK:
            MessageBox(NULL, "Hello, world!", "", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;
        default:                      /* for messages that we don't deal with */
            return DefWindowPro   essage, wParam, lParam);
    }

    return 0;
}
```

DefWindowProc

1. Dodatkowe argumenty, przekazywane razem z komunikatem.

2. Komunikat przesyłany gdy okno wymaga przerysowania.

3. Podwójnie wciśnięty lewy przycisk myszki.

4. Komunikat wysyłany tuż przed zamknięciem okna. (usunięciem z pamięci)

5. Jeżeli nie jest to żaden przez nas obsługiwany komunikat, wywołaj obsługę domyślną.

Delphi

- Struktura programu opiera się na VCL (Visual Component Library)
  - „VCL (ang. Visual Component Library) - biblioteka stworzona w języku Object Pascal (obiektowej wersji języka Pascal) przez firmę Borland na potrzeby środowiska Delphi, potem zaadaptowana też do środowiska C++ Builder.
  - Biblioteka VCL należy do jednych z bardziej przejrzystych i dobrze zaprojektowanych bibliotek wspomagających programowanie w środowisku Windows, zwłaszcza tworzenie interfejsu użytkownika. Przez długie lata dystansowała w tej dziedzinie konkurencyjną bibliotekę firmy Microsoft - MFC, dołączaną do środowiska Visual Studio.
  - Obecnie biblioteka VCL integruje w sobie też możliwość korzystania z technologii .NET firmy Microsoft.
  - Podstawą biblioteki VCL jest klasa bazowa TObject. Z niej dziedziczą wszystkie pozostałe klasy biblioteki.”

Źródło: http://pl.wikipedia.org/wiki/Visual_Component_Library

Delphi – Struktura Programu

Project1.pas

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

1. Dołączenie zasobów takich jaki ikony, obrazy, itp.

2. Punkt startowy programu, który w przypadku VCL jest procedurą startową modułu Object Pascala. (W tym przypadku programu.)

3. Standardowa inicjalizacja parametrów aplikacji. (Czyli pierwsza część z poprzedniego przykładu.)

4. Inicjalizacja okna zdefiniowanego za pomocą klasy TForm1. Wyświetlenie okna w zależności od atrybutu Visible.

5. Uruchomienie pętli obsługi komunikatów.

Definicja przykładowej klasy okna: TForm1

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants,
  Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

4. Importujemy projekt wizualny czyli graficzny układ kontrolek na ekranie wraz ze zdefiniowanymi wartościami atrybutów.

1. Nasza klasa okna dziedziczy funkcjonalność klasy TForm.

2. Która z kolei dziedziczy funkcjonalność z TCustomForm.

```
TForm = class(TCustomForm)
public
  procedure ArrangeIcons;
  procedure Cascade;
  procedure Next;
  procedure Previous;
  procedure Tile;

(...)
```

```
TCustomForm = class(TScrollingWinControl)
private
  FActiveControl: TWinControl;
  FFocusedControl: TWinControl;
(...)
  procedure RefreshMDIMenu;
  procedure ClientWndProc(var Message: TMessage);
  procedure CloseModal;
(...)
```

itd.

3. I tutaj mamy znaną, tylko pod trochę inną nazwą funkcję która implementuje obsługę komunikatów.

Procedura obsługi komunikatów - Delphi

```
procedure TCustomForm.ClientWndProc(var Message: TMessage);
(...)
var
  DC: HDC;
  PS: TPaintStruct;
  R: TRect;
begin
  with Message do
    case Msg of
      WM_PAINT:
        begin
          DC := TWMPaint(Message).DC;
          if DC = 0 then
            TWMPaint(Message).DC := BeginPaint(ClientHandle, PS);
          try
            if DC = 0 then
            begin
              GetWindowRect(FClientHandle, R);
              R.TopLeft := ScreenToClient(R.TopLeft);
              MoveWindowOrg(TWMPaint(Message).DC, -R.Left, -R.Top);
            end;
            PaintHandler(TWMPaint(Message));
          finally
            if DC = 0 then
              EndPaint(ClientHandle, PS);
          end;
        end;
    else
      Default;
    end;
end;
```

1. Zwróćmy uwagę, że komunikaty zostają nam przekazane za pomocą rekordu, a nie osobnych parametrów.

2. Atrybut Msg jest bezpośrednio mapowany z WParam i stanowi kod komunikatu.

3. Wywołanie odpowiedniej procedury rysującej (zazwyczaj zaimplementowanje w obsludze komunikatu OnPaint), prpo zainicjowaniu odpowiedniego kontekstu rysowania.

4. Jeżeli do tej pory nie obsłużyliśmy komunikatu, przekażmy go do obsługi domyślnej.

Procedura obsługi w TComponent

```
procedure TControl.WndProc(var Message: TMessage);
var
  Form: TCustomForm;
  KeyState: TKeyboardState;
  WheelMsg: TCMMouseWheel;
begin
  if (csQDesigning in ComponentState) then
  begin
    Form := GetParentForm(Self);
    if (Form <> nil) and (Form.Designer <> nil) and
      Form.Designer.IsDesignMsg(Self, Message) then Exit
  end;
  if (Message.Msg >= WM_KEYFIRST) and (Message.Msg <= WM_KEYLAST) then
  begin
    Form := GetParentForm(Self);
    if (Form <> nil) and Form.WantChildKey(Self, Message) then Exit;
  end

(...)

  else if Message.Msg = CM_VISIBLECHANGED then
    with Message do
      SendDockNotification(Msg, WParam, LParam);
  Dispatch(Message);
end;
```

1. Funkcja przekazująca kontrolę do procedur obsługi komunikatów oznaczonych specjalnym słowem kluczowym **message**.

Słowo kluczowe message

```
TScrollingWinControl = class(TWinControl)
private
  FHorzScrollBar: TControlScrollBar;
(...)
  procedure UpdateScrollBars;
  procedure WMSize(var Message: TWMSize); message WM_SIZE;
  procedure WMHScroll(var Message: TWMHScroll); message WM_HSCROLL;
  procedure WMVScroll(var Message: TWMVScroll); message WM_VSCROLL;
  procedure CMBiDiModeChanged(var Message: TMessage); message CM_BIDIMODECHANGED;
protected
  procedure AdjustClientRect(var Rect: TRect); override;
```

- W Object Pascalu (języku programowania Delphi) wprowadzono specjalne słowo kluczowe message, które wykorzystywane jest przez procedury obsługi komunikatów zdefiniowane w komponentach VCL do mapowania komunikatów systemu operacyjnego (i nie tylko) na odpowiednie procedury obsługi (*ang. handling procedures*).
- Można definiować własne komunikaty. Do tego wykorzystuje się liczbę WM_USER, która definiuje minimalny kod komunikatu.

GLUT – aplikacje OpenGL

- Aplikacja OpenGL jest wyświetlana w środowisku graficznym. Jej struktura nie odbiega zasadniczo od pozostałych przykładów.

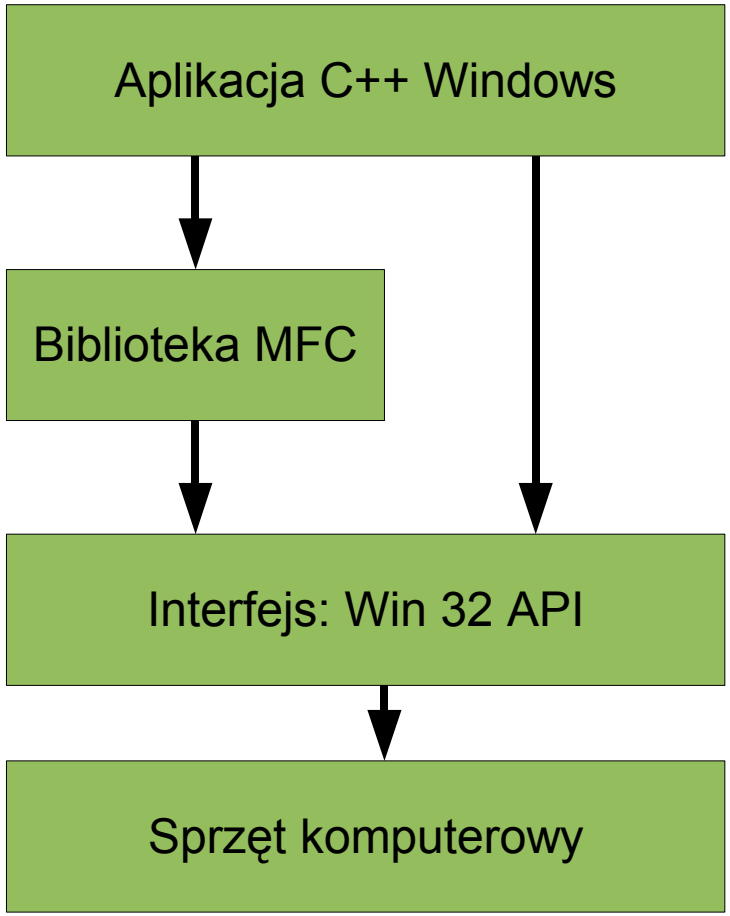
```
#include <iostream>
#include <cstdlib>
#include <GL/glut.h>
using namespace std;

// function prototypes
void disp(void);
void keyb(unsigned char key, int x, int y);

// window identifier
static int win;

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

  // initialize glut
  glutInit(&argc, argv);
  // specify the display mode to be RGB and single buffering
  // we use single buffering since this will be non animated
  glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
  // define the size
  glutInitWindowSize(500,500);
  // the position where the window will appear
  glutInitWindowPosition(100,100);
  // create the window, set the title and keep the
  // window identifier.
  win = glutCreateWindow("Yet another teapot");
  glutDisplayFunc(disp);
  glutKeyboardFunc(keyb);
  // define the color we use to clearscreen
  glClearColor(0.0,0.0,0.0,0.0);
  // enter the main loop
  glutMainLoop();
  return 0;
}
```

1. Jak widzimy struktura programu jest bardzo podobna. Zawiera on inicjalizację oraz wywołanie metody obsługującej mapowanie komunikatów na procedury obsługi. W naszym przypadku są to disp i keyb.

```
void disp(void){
  glClear(GL_COLOR_BUFFER_BIT);
  glutWireTeapot(0.5);
  // glutSolidTeapot(0.5);
  // glutWireSphere(0.5,100,100);
}
void keyb(unsigned char key, int x, int y){
  cout << "Pressed key " << key <<
     "on coordinates (" << x << "," << y << ")";
  cout << endl;
  if(key == 'q'){
    cout << "Got q,so quitting " << endl;
    glutDestroyWindow(win);
    exit(0);
  }
}
```

Biblioteka Microsoft MFC

- Microsoft Foundation Classes – było odpowiedzią Microsoftu na biblioteki takie jak VCL opracowane przez Borlanda oraz na własny produkt Visual Basic.
- Jej zadaniem jest „obudowanie” standardowego i niewygodnego Win32 API za pomocą obiektowych komponentów, ułatwiających tworzenie aplikacji.
- Należy zdawać sobie sprawę, że Win32 API jest napisane w języku C, a MFC w obiektowym języku C++.
- .NET jest w również nakładką na Win32 API. Można powiedzieć następcą MFC (i jeszcze kilku innych technologi).
- MFC zawiera około 200 logicznie zorganizowanych klas zamiast ponad 2000 funkcji Win32 API.



Biblioteka Microsoft MFC

- Redukcja kodu programu implementując wielokrotnie wykorzystywany kod inicjalizujący zasoby.
- Mniejszy rozmiar plików wykonywalnych. (Znaczna część funkcjonalności znajduje się w bibliotece dzielonej np. mfc422.dll)
- Szybsza implementacja i rozbudowa aplikacji. (Prostszy kod)
- Było problematyczne dla pierwszych programistów nieprzyzwyczajonych do programowania obiektowego.
- Programy MFC muszą być napisane w C++ i wymagają wykorzystania klas. A w szczególności technologii:
  - tworzenia i zarządzania klasami, dziedziczenia, przesłaniania metod, enkapsulacji danych, polimorfizmu.

Obiektowość? A funkcje globalne Afx...?

- Pomimo obiektowego projektu, projektantom MFC nie udało się obejść bez funkcji globalnych. (W rzeczywistości .NET też posiada specjalny typ funkcji globalnych w postaci metod statycznych.)
- Przykłady funkcji globalnych:
  - AfxAbort() - bezwarunkowe zakończenie aplikacji,
  - AfxBeginThread() - utworzenie i uruchomienie nowego wątku,
  - AfxGetApp() - zwraca wskaźnik do obiektu aplikacji,
  - AfxGetMainWnd() - zwraca wskaźnik do głównego okna aplikacji,
  - AfxGetInstanceHandle() - zwraca uchwyt aktualnej aplikacji,
  - AfxRegisterWndClass() - rejestruje własną, dodatkową klasę okna dla naszej aplikacji MFC
- *Przypis: Wnioskiem może być stwierdzenie, że obiektowość nie jest „panaceum” i stanowi rozwiązanie tylko części problemów projektowania oprogramowania.*

MFC – Hello World

```
#include <afxwin.h> // Base MFC header file

class CHelloApp : public CWinApp
{
public:
        virtual BOOL InitInstance();
};
class CHelloWnd : public CFrameWnd
{
protected:
        afx_msg void OnPaint();
        DECLARE_MESSAGE_MAP();
};
void CHelloWnd::OnPaint()
{
        CPaintDC dc(this);
        dc.TextOut(72, 72, "Hello MFC!");
}
BEGIN_MESSAGE_MAP(CHelloWnd, CWnd)
        ON_WM_PAINT()
END_MESSAGE_MAP()
CHelloApp helloApp;  // The single global CHelloApp
object
BOOL CHelloApp::InitInstance()
{
        CHelloWnd* pMainWnd = new CHelloWnd;
        if (!pMainWnd->Create("AfxFrameOrView",
        "Hello MFC App"))
                return FALSE;
        m_pMainWnd = pMainWnd;

        m_pMainWnd->ShowWindow(m_nCmdShow);
        m_pMainWnd->UpdateWindow();
        return TRUE;
}
```

- Minimalna aplikacja MFC musi implementować co najmniej dwie klasy dziedziczące z:
  - CWinApp
    - definiuje obiekt aplikacji,
    - zawiera w sobie pętlę obsługi komunikatów
  - CFrameWnd (zazwyczaj)
    - definiuje główne okno aplikacji.
  - Aby w MFC powyższe klasy były widoczne dla aplikacji musimy dołączyć plik <Afxwin.h>

MFC – Hello World

```
#include <afxwin.h> // Base MFC header file

class CHelloApp : public CWinApp
{
public:
        virtual BOOL InitInstance();
};
class CHelloWnd : public CFrameWnd
{
protected:
        afx_msg void OnPaint();
        DECLARE_MESSAGE_MAP();
};
void CHelloWnd::OnPaint()
{
        CPaintDC dc(this);
        dc.TextOut(72, 72, "Hello MFC!");
}
BEGIN_MESSAGE_MAP(CHelloWnd, CWnd)
        ON_WM_PAINT()
END_MESSAGE_MAP()
CHelloApp helloApp;  // The single global CHelloApp

BOOL CHelloApp::InitInstance()
{
        CHelloWnd* pMainWnd = new CHelloWnd;
        if (!pMainWnd->Create("AfxFrameOrView",
                              "Hello MFC App"))
                return FALSE;
        m_pMainWnd = pMainWnd;

        m_pMainWnd->ShowWindow(m_nCmdShow);
        m_pMainWnd->UpdateWindow();
        return TRUE;
}
```

- Mapa komunikatów, która jest wykorzystywana przez pętlę obsługi komunikatów zaimplementowanę w MFC w klasie CwinApp.
- Specjalne makro implementujące szereg zmiennych i funkcji potrzebnych do obsługi komunikatów.
- Inicjujemy naszą aplikację. Metoda InitInstance jest automatycznie wywoływana przez MFC po uruchomieniu aplikacji.
  - Tworzymy tutaj nowe okno.
  - Zapamiętujemy wskaźnik do niego jako do okna głównego.
  - Pokazujemy i uaktualniamy.

MFC - Obsługa komunikatów

- W Win32 API komunikaty są obsługiwane za pomocą „wielkiej” sekcji switch/case.
- W MFC zrealizowano „mapę komunikatów” w formie tabeli przechowującej:
  - numer komunikatu (np WM_PAINT, WM_RESIZE, itp.)
  - wskaźnik do odziedziczonej funkcji obsługi komunikatu
- MFC udostępnia szereg predefiniowanych metod obsługi komunikatów *(ang. message handlers)* dla wszystkich możliwych komunikatów. Naszym zadaniem zazwyczaj jest przesłonięcie oryginalnej metody. Do zrobienia tego musimy:
  - Poinformować MFC, że zamierzamy implementować własną metodę obsługi (za pomocą makra w mapie komunikatów o odpowiedniej nazwie)
  - Musimy napisać naszą metodę stosując odpowiednia konwencję nazewnictwa oraz odpowiednio zadeklarować argumenty. (Bardzo często wymaga to przeglądania dokumentacji MSDN)
- Dla przykładu, metoda OnPaint nie pobiera żadnych argumentów i reaguje na komunikat z żądaniem przerysowania ekranu. jej prototyp wygląda:

```
afx_msg void OnPaint();
```

afx_msg informuje kompilator, że metoda obsługuje komunikat. (Podobno nie miało to znaczenia, ale było zarezerwowane dla przyszłych wersji MFC.)

Nazewnictwo makr obsługi

- Jeżeli mapa zawiera:
  - ON_WM_CHAR( )
  - ON_WM_LBUTTONDOWN( )
  - ON_WM_RBUTTONDOWN( )
- Wówczas odpowiadające makrom komunikaty to:
  - WM_CHAR
  - WM_LBUTTONDOWN
  - WM_RBUTTONDOWN
- A w naszym programie musimy przesłonić domyślne funkcje naszymi własnymi:
  - CWnd::OnChar(UINT ch, UINT count, UINT flags);
  - CWnd::OnLButtonDown(UINT flags, CPoint loc);
  - CWnd::OnRButtonDown(UINT flags,CPoint loc);

Microsoft .NET – Struktura programu

- Idea obudowania Win32 API pozostała ta sama co w MFC.
- Klasy mają inne nazwy.
- Prawdziwa obiektowość za pomocą delegacji i zdarzeń.
- Minimalny program pokazujący na ekranie puste okno.

```
using System;
using System.Windows.Forms;

namespace Project1
{
    class OknoByHand : Form
    {
        public OknoByHand()
        {
            Text = "okno by hand lub Hello World!";
        }

        public static int Main()
        {
            Application.Run(new OknoByHand());
            return 0;
        }
    }
}
```

1. Dziedziczymy funkcjonalność z klasy Form, która za nas implementuje obsługę mapowania komunikatów, posiada zdefiniowane szereg zdarzeń, itp. Idea bardzo zblizona do RAD z Delphi.

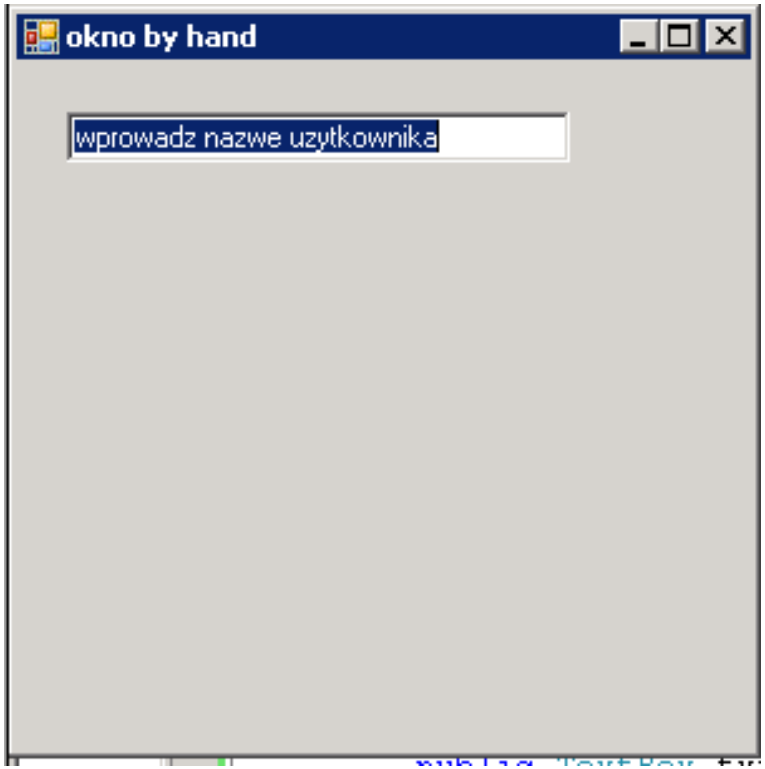
2. W konstruktorze definiujemy napis na pasku tytułu. Tutaj również powinniśmy zainicjalizować komponenty naszego okna.

3. Nie musimy deklarować własnego obiektu Application. >NET robi to za nas udostępniając obiekt aplikacji związany z głównym wątkiem. przekazując do Run nasze okno wskazujemy, że aplikacja ma działać dopuki to okno nie zostanie zamknięte.

Microsoft .NET – struktura programu

```
using System;
using System.Windows.Forms;

namespace Project1
{
    class OknoByHand : Form
    {
        public TextBox txtUserName;

        public OknoByHand()
        {
            Text = "okno by hand";

            txtUserName = new TextBox();
            txtUserName.Left = 20;
            txtUserName.Top = 20;
            txtUserName.Width = 200;
            txtUserName.Text = "wprowadz nazwe uzytkownika";

            Controls.Add(txtUserName);
        }

        public static int Main()
        {
            Application.Run(new OknoByHand());
            return 0;
        }
    }
}
```

- Dodaliśmy kontrolkę do naszego projektu w „stylu .NET Framework 1.1”.
- 1. Zadeklarowaliśmy zmienną lokalną, która przechowuje referencję do kontrolki TextBox.
- 2. „Manualnie” zdefiniowaliśmy parametry kontrolki.
- 3. Dodaliśmy kontrolkę do naszego okna.

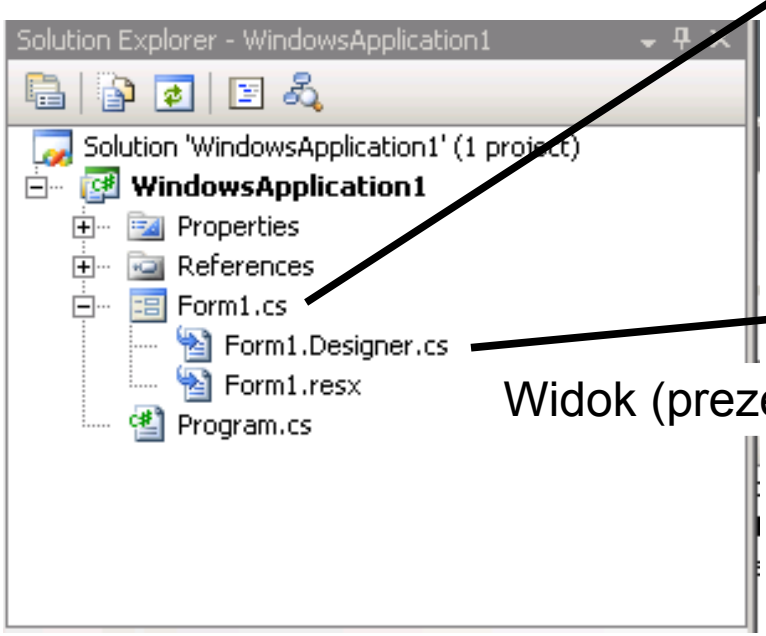


Microsoft .NET 2.0

- Microsoft w .NET 2.0 dodał do języka nowe słowo kluczowe **partial**.
- Słowo to pozwala dzielić definicję jednej klasy na wiele plików. Dzięki temu możemy elegancko oddzielić kod odpowiadający za konfigurację interfejsu graficznego od kodu kontrolera.

```
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void listBox1_KeyPress(object sender, KeyPressEven
        {

        }
    }
}
```

Kontroler (logika aplikacji)



Widok (prezentacja)

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.listBox1 = new System.Windows.Forms.ListBox();
    this.linkLabel1 = new System.Windows.Forms.LinkLabel();
    this.SuspendLayout();
    //
    // listBox1
    //
    this.listBox1.FormattingEnabled = true;
    this.listBox1.Location = new System.Drawing.Point(47, 30);
    this.listBox1.Name = "listBox1";
    this.listBox1.Size = new System.Drawing.Size(111, 17);
    this.listBox1.TabIndex = 0;
    this.listBox1.KeyPress += new System.Windows.Forms.KeyPressEventH
    //
    // linkLabel1
    //
    this.linkLabel1.AutoSize = true;
    this.linkLabel1.Location = new System.Drawing.Point(181, 30);
```

Microsoft .NET - Delegacje

- Specjalna „struktura”, typ, w rzeczywistości w CIL klasa, która potrafi przechowywać wskaźnik (referencję) do metody statycznej lub obiektu, która ma zostać uruchomiona w momencie uruchomienia delegacji. W rzeczywistości delegacja w naturalny sposób potrafi przechowywać listę metod, które mają zostać uruchomione.
- Delegacja w ukryty sposób przechowuje informacje o:
  - nazwie wywoływanej metody
  - typie wywoływanego obiektu
  - argumentach wywoływanej metody
  - argumentach zwracanych przez wywoływaną metodę
- Delegacja przechowuje powyższe informacje w sposób ukryty, co oznacza, że nie jest to widoczne bezpośrednio dla programisty. Co więcej, programista zazwyczaj nie używa właściwości delegacji związanych z tym, że jest ona klasą. Jedną z ważniejszych cech delegacji jest to, że aby można jej było użyć trzeba ją utworzyć za pomocą operatora new, podobnie jak inne normalne klasy.

Przykład delegacji - deklaracja

- Poniższy przykład bazuje na założeniu, że mamy do stworzenia pewien system monitoringu stanu rzek. Każda rzeka ma swoje własne parametry określające maksymalny dopuszczalny poziom oraz informacje o aktualnym poziomie. Uaktualniając poziom rzeki chcemy aby w razie przekroczenia wartości krytycznej lub nawet niebezpiecznego zbliżenia się do tej wartości rzeka wywoływała specjalne metody które jej dostarczymy. Metody będą w stanie odpowiednio obsłużyć daną sytuację wyjątkową.

```
 1 public class River
 2     {
 3         (...)
 4
 5         public delegate void AboutToFlood(string msg);
 6         public delegate void Flooded(string msg);
 7
 8         private AboutToFlood aboutToFloodListener;
 9         private Flooded floodedListener;
10
11         (...)
12     }
```

Rejestracja delegacji

- Rozwińmy kod i dodajmy do niego dwie metody, które będą rejestrowały metody do naszych delegacji. ' Rejestrowały ' czyli ustawią atrybuty prywatne tak aby wskazywały na podane metody.
- Rejestracja jest niezbędna ponieważ atrybuty deklarujące delegacje są prywatne.

```
 1 public class River
 2     {
 3         (...)
 4
 5         public delegate void AboutToFlood(string msg);
 6         public delegate void Flooded(string msg);
 7
 8         private AboutToFlood aboutToFloodListener;
 9         private Flooded floodedListener;
10
11         public void OnAboutToFlood(AboutToFlood aboutToFlood)
12         {
13             aboutToFloodListener = aboutToFlood;
14         }
15
16         public void OnFlooded(Flooded flooded)
17         {
18             floodedListener = flooded;
19         }
20
21         (...)
22     }
```

Uruchomienie delegacji

- Dodajmy jeszcze wywołanie metod zapamiętanych w naszych delegacjach w momencie wystąpienia sytuacji wyjątkowych, czyli zbliżenia i przekroczenia wartości maksymalnej. - Wyzwolenie delegacji.

1. Deklarujemy atrybut w C#.

```
 1public class River
 2{
 3  (...)
 4
 5  public int Level
 6  {
 7   set
 8   {
 9     riverLevel = value;
10     if (riverLevel > riverLevelMax)
11     {
12       // na poczatku sprawdzamy, czy nasza delegacja jest zainicjalizowana
13       if (floodedListener != null)
14          // jeżeli tak, to wywolujemy metode, podajac nazwe delegacji i przekazujac argument
15          floodedListener("The river " + name + " has flooded... ");
16       } else {
19         if (riverLevel > (riverLevelMax - (riverLevelMax / 10)))
20             // podobnie jak poprzednio sprawdzamy, czy delegacja jest zainicjalizowana
21                if (aboutToFloodListener != null)
22                  // jezeli jest, to ja wywolujemy
23                     aboutToFloodListener("The river " + name + " is about to flood... ");
24      }
25   }
26   get { return riverLevel; }
27 }
28
29  (...)
31}
```

Program testujacy...

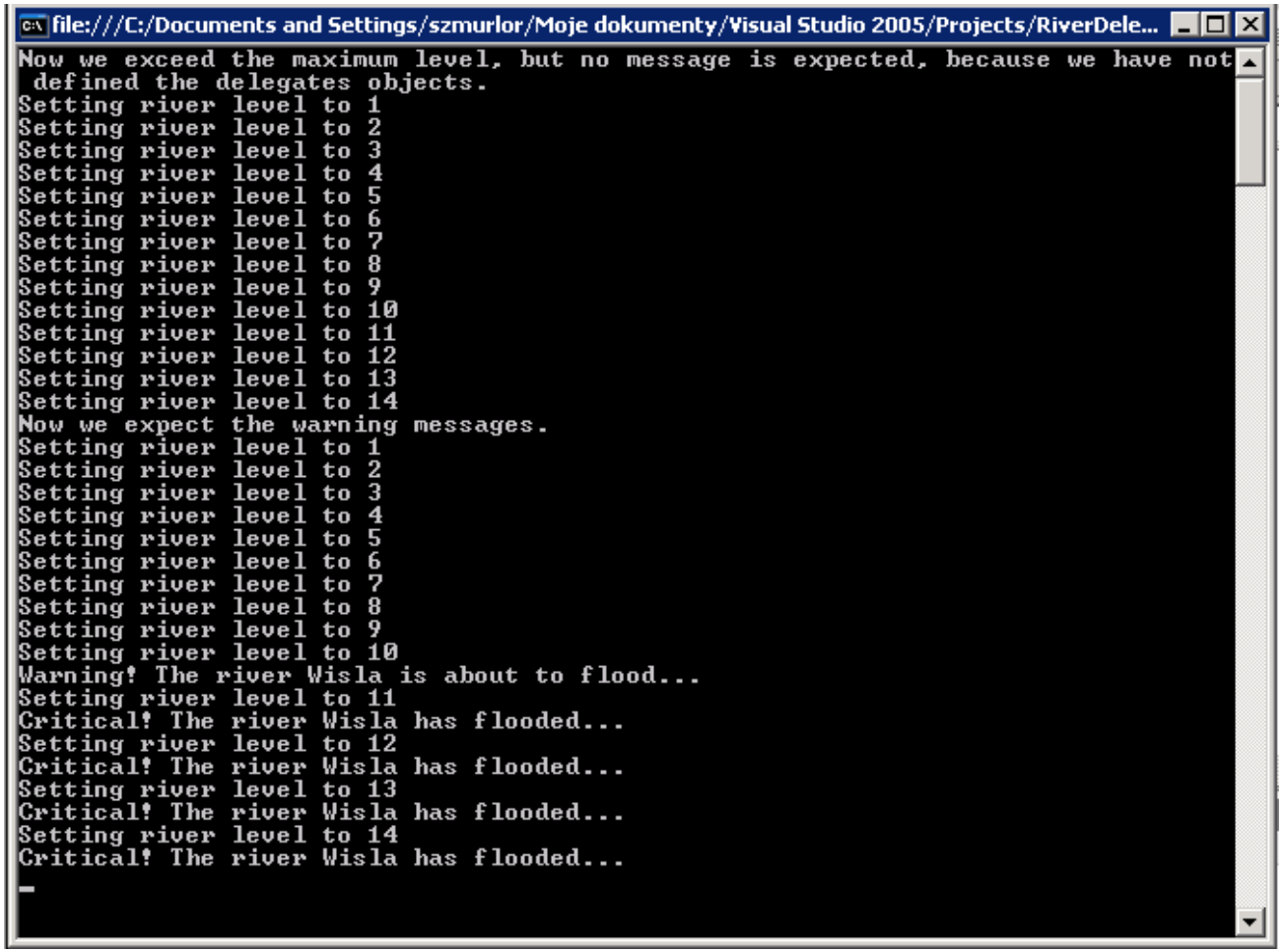
```
61      class Program
62      {
63          static void AboutToFlood_Handler(string msg)
64          {
65              Console.WriteLine("Warning! {0}", msg);
66          }
67
68          static void Flooded_Handler(string msg)
69          {
70              Console.WriteLine("Critical! {0}", msg);
71          }
72
73          static void Main(string[] args)
74          {
75              River river = new River("Wisla");
76
77              Console.WriteLine("Now we exceed the maximum level, but no „
                        „message is expected, because we have not defined the delegates objects.");
78              for (int i = 1; i < 15; i++)
79              {
80                  Console.WriteLine("Setting river level to {0}", i);
81                  river.Level = i;
82              }
83
84              // w naszym programie musimy jeszcze zainicjalizowac odpowiednie delegacje,
85              // tworzac dla kazdej metody nowy obiekt delegacji
86              river.OnAboutToFlood( new River.AboutToFlood( AboutToFlood_Handler ) );
87              river.OnFlooded(new River.Flooded(Flooded_Handler));
88
89              Console.WriteLine("Now we expect the warning messages.");
90              for (int i = 1; i < 15; i++)
91              {
92                  Console.WriteLine("Setting river level to {0}", i);
93                  river.Level = i;
94              }
95
96              Console.ReadLine();
97          }
98      }
```

1. Definiujemy metody obsługi zdarzeń, które przekażemy do tworzonych delegacji.

2. Rejestrujemy nowo tworzone obiekty delegacji w klasie River.

Więcej informacji o przykładzie

- Dokładniejszy opis przedstawionego przykładu delegacji można znaleźć:
  - http://www/wikidyd/AiSD_E/Lab/Dodatkowe_Przyk%C5%82ady/Delegate
- Pełny kod do pobrania ze strony, w pliku delegateRiver.rar
- Wynik na ekranie:



Microsoft .NET - Events

- Słowo kluczowe event zostało wprowadzone do .NET aby zmniejszyć ilość tworzonego kodu i zautomatyzować procedury rejestracji i wyrejestrowania delegacji z danych klas. Kompilator, gdy napotka słowo event automatycznie dodaje odpowiednie metody, które są ukryte przed programistą.
- Deklaracja zdarzeń składa się z dwóch etapów:
  - deklaracji odpowiedniej delegacji (przez co zdefiniujemy typ metody, która będzie wywoływana w momencie pojawienia się zdarzenia),
  - deklaracji zdarzenia przy wykorzystaniu wcześniejszej delegacji

```
public class SenderOfEvents
{
    public delegate retval AssociatedDelegate(args);
    public event AssociatedDelegate NameOfEvent;
...
}
```

Przykład deklaracji i uruchamiania zdarzenia

```
public class Car
{
    // Ta delegacja jest uzywana w zdarzeniu
    public delegate void CarEventHandler(string msg);
    // Potrafimy wygenerowac dwa zdarzenia.
    public event CarEventHandler Exploded;
    public event CarEventHandler AboutToBlow;

    public void Accelerate(int delta)
    {
        // Czy samochod jest zepsuty?
        if (carIsDead)
        {
            if (Exploded != null)
                Exploded("Sorry, this car is dead...");
        } else {
            currSpeed += delta;
            // Prawie zepsuty?
            if (10 == maxSpeed - currSpeed
                && AboutToBlow != null)
            {
                AboutToBlow("Careful! I am going to blow!");
            }
            // Nadal w porzadku
            if (currSpeed >= maxSpeed)
                carIsDead = true;
            else
                Console.WriteLine("->CurrSpeed = {0}", currSpeed);
        }
    }
}
```

1. Procedury obsługi zdarzenia przypisane do delegacji wywołujemy wpisując nazwę zdarzenia wraz z parametrami.

2. pamiętamy o zadeklarowaniu delegacji, która definiuje sposób wywołania metody obsługującej dane zdarzenie.

Rejestracja zdarzeń

```
Car.EngineHandler d = new Car.EngineHandler(CarExplodedEventHandler)
myCar.Exploded += d;
```

- W powyższym przykładzie tworzymy obiekt typu delegacja: Car.EngineHandler i przypisujemy mu metodę obsługi o nazwie: CarExplodedEventHandler.
- Następnie za pomocą operatora „**+=**” dodajemy naszą delegację do kolejki zdarzenia.
- W C# wprowadzono dodatkowe ułatwienie nie wymagające od nas jawnego tworzenia obiektu delegacji. W nowym podejściu kompilator sam automatycznie stworzy dla nas 'ukryty' obiekt delegacji o odpowiednim typie (sam go rozpozna). Naszym zadaniem jest podac jedynie nazwę metody. Wówczas kod skróci się do:

```
myCar.Exploded += CarExplodedEventHandler;
```

Program testujący

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Events *****");
        Car c1 = new Car("SlugBug", 100, 10);
        // Register event handlers.
        c1.AboutToBlow += new Car.CarEventHandler(CarIsAlmostDoomed);
        c1.AboutToBlow += new Car.CarEventHandler(CarAboutToBlow);
        Car.CarEventHandler d = new Car.CarEventHandler(CarExploded);
        c1.Exploded += d;
        Console.WriteLine("\n***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        // Remove CarExploded method
        // from invocation list.
        c1.Exploded -= d;
        Console.WriteLine("\n***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }

    public static void CarAboutToBlow(string msg)
    { Console.WriteLine(msg); }

    public static void CarIsAlmostDoomed(string msg)
    { Console.WriteLine("Critical Message from Car: {0}", msg); }

    public static void CarExploded(string msg)
    { Console.WriteLine(msg); }
}
```

1
2
3
4
5

Komunikaty niestandardowe

- Aby wykonać niektóre zdania musimy obsłużyć niewspierane przez .NET komunikaty.
- Przykładem takiego komunikatu nie obsługiwane przez platformę .NET jest informacja o zmianie woluminu (np. włożenie, usunięcie płyty CD lub pamięci usb):
  - WM_CHANGEDEVICE
- W .NET są co najmniej dwie metody własnej obsługi komunikatów:
  - oficjalnie promowana przez .NET: IMessageFilter
  - przesłonięcie wirtualnej metody WndProc

1. Wykorzystanie IFilterMessage

- Wykorzystamy stnadardowy interfjes .NET: IFilterMessage, w którym przesłonimy metodę PreFilterMessage.

```
using System;
using System.Windows.Forms;
public class MyFilter: IMessageFilter
{
  public bool PreFilterMessage(ref Message aMessage)
  {
    if (aMessage.Msg==WM_AMESSAGE)
    {
      //WM_AMESSAGE Dispatched
      //Let's do something here
      //...
    }
    // This can be either true of false
    // false enables the message to propagate to all other
    // listeners
    return false;
  }
}
```

1. Rejestracja filtra komunikatów

- Pozostaje nam tylko stworzyć obiekt filtra
- I zarejestrować go obiekcie aplikacji.

```
MyFilter fFilter = new MyFilter();
(…)
Application.AddMessageFilter(fFilter);
(…)
Application.RemoveMessageFilter(fFilter);
```

- Lub usunąć go...

2. Przesłonięcie procedury WndProc

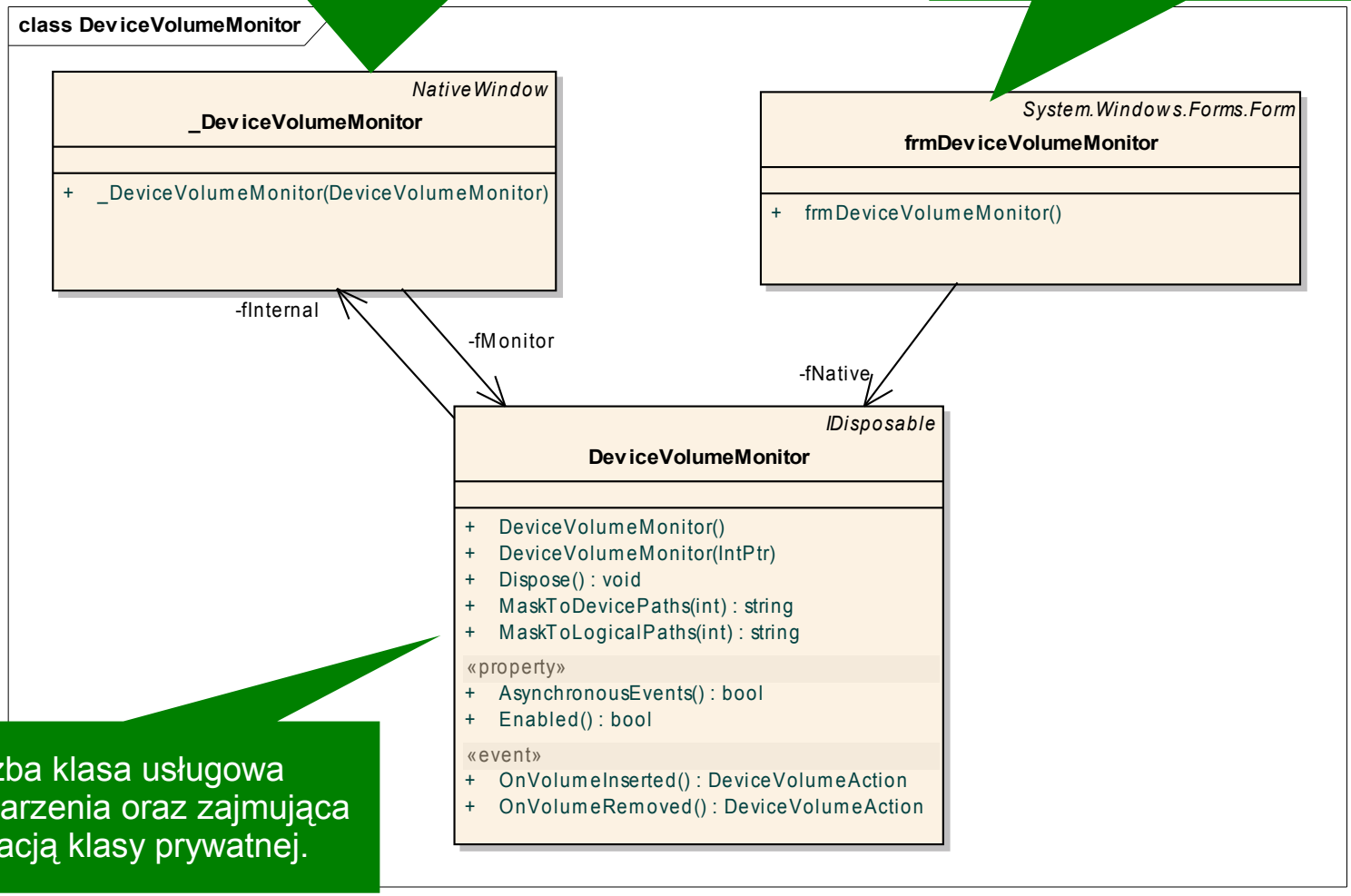
- Procedurę WndProc możemy przesłonić w klasie potomnej dziedziczącej z klasy Control lub NativeWindow. Z klasy Control dziedziczymy, gdy chcemy rozszerzyć funkcjonalność jakiejś specyficznej kontrolki.
- W naszym przykładzie dziedziczymy z NativeWindow.

```
protected override void WndProc(ref Message aMessage)
{
    if (aMessage.Msg==WM_AMESSAGE)
    {
        //WM_AMESSAGE Dispatched
        //Let's do something here
        //...
    }
}
```

Omówienie przykładu

2. Prywatna klasa dla naszej przestrzeni nazw przechowująca informacje o kodach Win32 API, kodach komunikatów oraz naszą główną pętlę obsługi komunikatów (dziedziczy z NativeWindow)

1. Ono z graficznym interfejsem użytkownika, definiuje metody obsługi zdarzeń

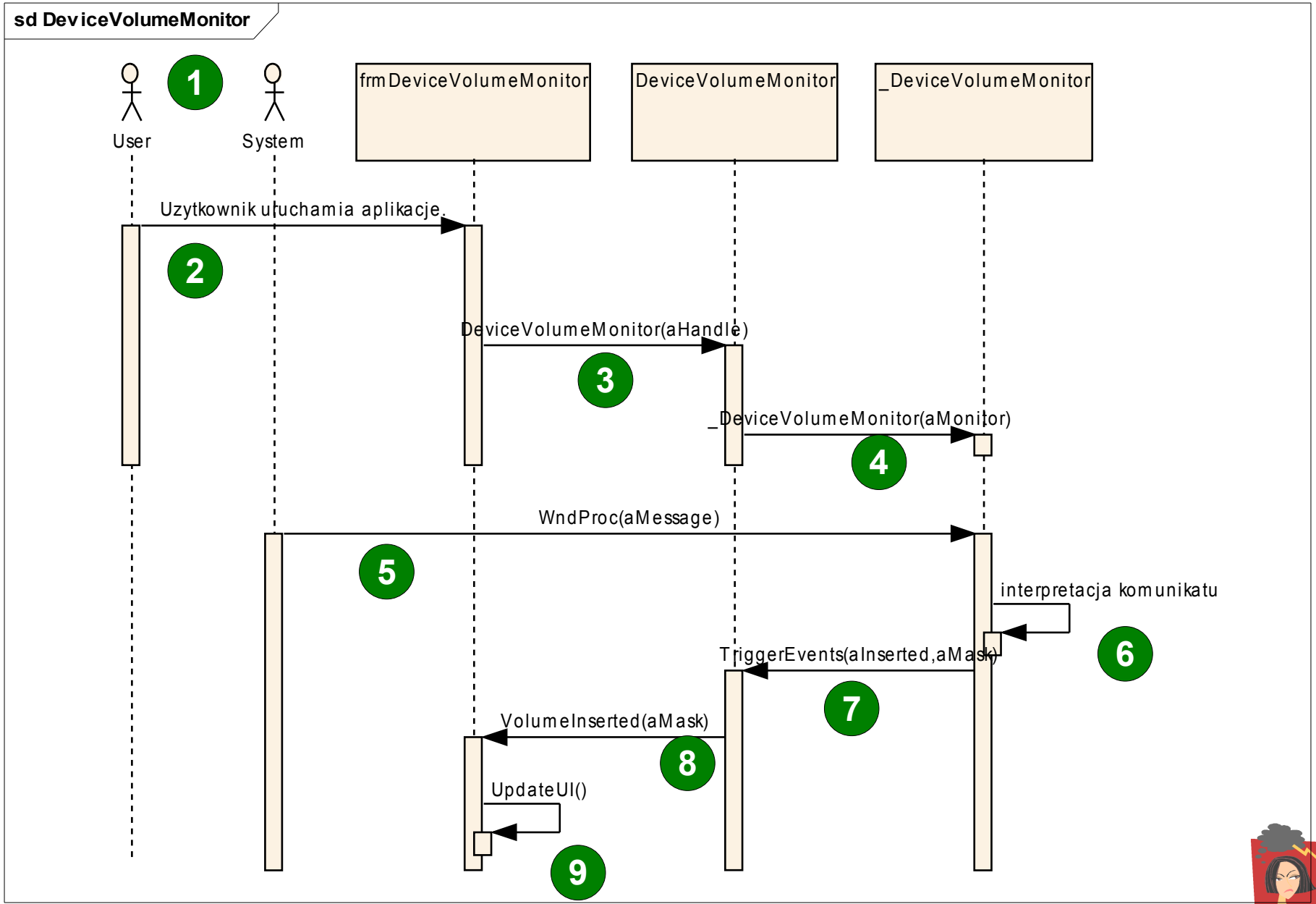


3. Publiczba klasa usługowa definiująca zdarzenia oraz zajmująca się inicjalizacją klasy prywatnej.

Omówienie przykładu

- Oryginalny przykład znajduje się na stronie:
  http://www.codeproject.com/dotnet/devicevolumemonitor.asp
- Prezentowany przykład składa się z trzech zasadniczych klas:
  - public class frmDeviceVolumeMonitor : System.Windows.Forms.Form - okno z graficznym interfejsem użytkownika, definiuje metody obsługi zdarzeń,
  - internal class _DeviceVolumeMonitor: NativeWindow - prywatna klasa dla naszej przestrzeni nazw przechowująca informacje o kodach Win32 API, kodach komunikatów oraz naszą główną pętlę obsługi komunikatów (dziedziczy z NativeWindow)
  - public class DeviceVolumeMonitor: IDisposable - publiczba klasa usługowa definiująca zdarzenia oraz zajmująca się inicjalizacją klasy prywatnej.
- W załączonym przykładzie implementujemy również asynchroniczny sposób wyzwalania zdarzeń aby nie blokować obsługi natywnych komunikatów. (pomijamy w naszym opisie)

Diagram sekwencji



frmDeviceMonitor

```
public class frmDeviceVolumeMonitor : System.Windows.Forms.Form
{
    DeviceVolumeMonitor fNative;
(...)
    public frmDeviceVolumeMonitor()
    {
        InitializeComponent();
        // DeviceVolumeMonitor create instance
        fNative = new DeviceVolumeMonitor(this.Handle);
        fNative.OnVolumeInserted += new DeviceVolumeAction(VolumeInserted);
        fNative.OnVolumeRemoved += new DeviceVolumeAction(VolumeRemoved);
        UpdateUI();
    }

    [STAThread]
    static void Main()
    {
        Application.Run(new frmDeviceVolumeMonitor());
    }
    private void VolumeInserted(int aMask)
    {
        lbEvents.Items.Add("Volume inserted in "+fNative.MaskToLogicalPaths(aMask));
    }
    private void VolumeRemoved(int aMask)
    {
        lbEvents.Items.Add("Volume removed from "+fNative.MaskToLogicalPaths(aMask));
    }
}
```

1. Konstruktor inicjujący obiekt monitorujący zdarzenia, oraz rejestrujący w nim metody obsługi zdarzeń.

2. Funkcja Main uruchamiana w momencie wystartowania programu.

3. Metody obsługi komunikatów.

DeviceVolumeMonitor

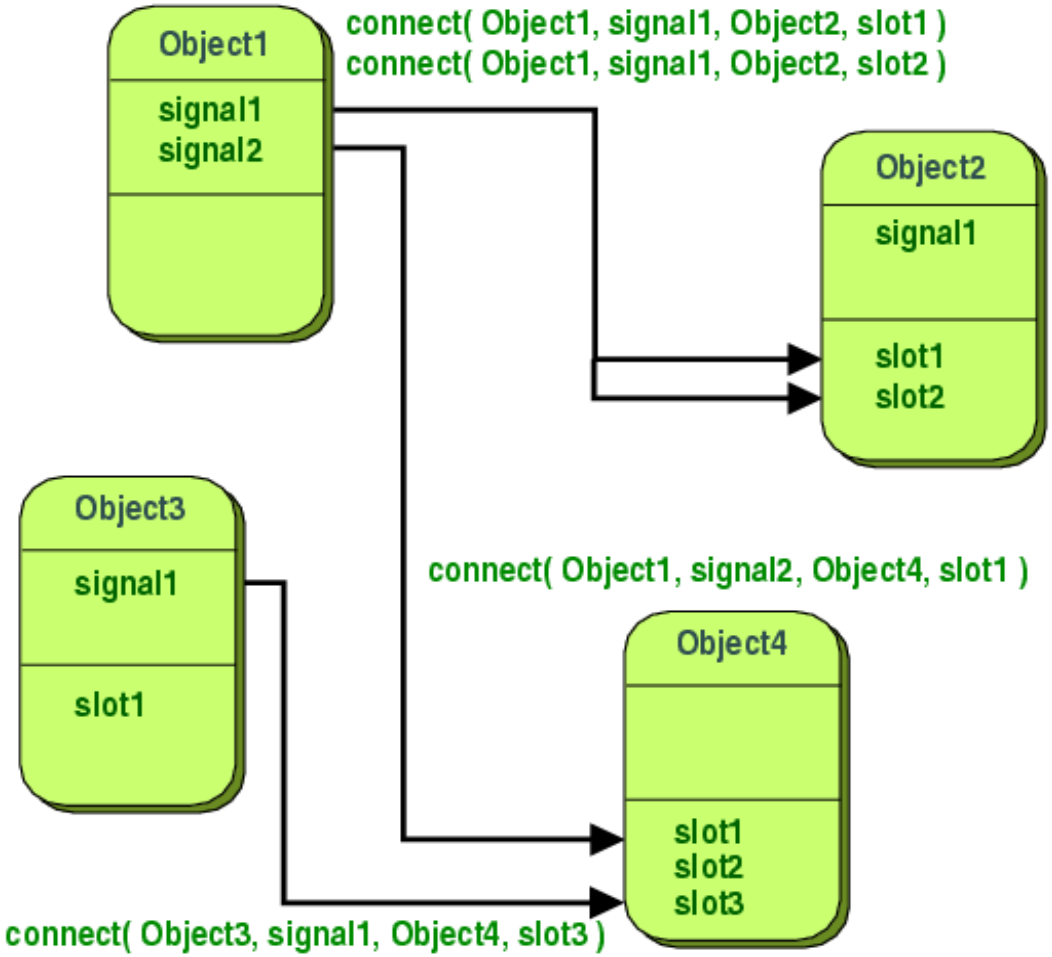
```
public class DeviceVolumeMonitor: IDisposable
{
    (...)
    public event DeviceVolumeAction OnVolumeInserted;
    public event DeviceVolumeAction OnVolumeRemoved;

    public DeviceVolumeMonitor(IntPtr aHandle)
    {
        if (aHandle!=IntPtr.Zero) { fHandle = aHandle; }
        else { throw new DeviceVolumeMonitorException("Invalid handle!"); }
        Initialize();
    }
    private void Initialize()
    {
        fInternal = new _DeviceVolumeMonitor(this);
        fDisposed = false;
     (...)
    }
    internal void TriggerEvents(bool aInserted, int aMask)
    {
        if (AsynchronousEvents) {
            if(aInserted) { OnVolumeInserted.BeginInvoke(aMask,null,null); }
            else { OnVolumeRemoved.BeginInvoke(aMask,null,null); }
        } else  {
            if(aInserted) { OnVolumeInserted(aMask); }
            else { OnVolumeRemoved(aMask); }
        }
    }
    ~DeviceVolumeMonitor()
    {
        Dispose(false);
    }
}
```

_DeviceVolumeMonitor

```
internal class _DeviceVolumeMonitor: NativeWindow {
    DeviceVolumeMonitor fMonitor;

    #region API constants and structures

    const int WM_DEVICECHANGE = 0x0219;

    public enum DeviceEvent:int {
        Arrival = 0x8000,          //DBT_DEVICEARRIVAL
        QueryRemove = 0x8001,      //DBT_DEVICEQUERYREMOVE
        QueryRemoveFailed = 0x8002, //DBT_DEVICEQUERYREMOVEFAILED
        RemovePending = 0x8003,    //DBT_DEVICEREMOVEPENDING
        RemoveComplete = 0x8004,   //DBT_DEVICEREMOVECOMPLETE
        Specific = 0x8005,         //DBT_DEVICEREMOVECOMPLETE
        Custom = 0x8006            //DBT_CUSTOMEVENT
    }
    public enum DeviceType:int {
        OEM = 0x00000000,          //DBT_DEVTYP_OEM
        DeviceNode = 0x00000001,   //DBT_DEVTYP_DEVNODE
        Volume = 0x00000002,       //DBT_DEVTYP_VOLUME
        Port = 0x00000003,         //DBT_DEVTYP_PORT
        Net = 0x00000004           //DBT_DEVTYP_NET
    }
(...)
    protected override void WndProc(ref Message aMessage)
    {
  (...)
        base.WndProc(ref aMessage);
        if(aMessage.Msg==WM_DEVICECHANGE && fMonitor.Enabled) {
            lEvent = (DeviceEvent)aMessage.WParam.ToInt32();
            if (lEvent==DeviceEvent.Arrival || lEvent==DeviceEvent.RemoveComplete) {
                lBroadcastHeader = (BroadcastHeader)Marshal.PtrToStructure(aMessage.LParam,typeof(BroadcastHeader));
                if(lBroadcastHeader.Type==DeviceType.Volume) {
                    lVolume = (Volume)Marshal.PtrToStructure(aMessage.LParam,typeof(Volume));
                    if((lVolume.Flags & (int)VolumeFlags.Media)!=0) {
                        fMonitor.TriggerEvents(lEvent==DeviceEvent.Arrival,lVolume.Mask);
                    }
                }
            }
        }
    }
}
```

Qt – Sygnały i Sloty

- Rozszerzenie języka poprzez stworzenie własnego preprocesora.
- Idea bardzo podobna do MFC. (Tylko że biblioteka jest wieloplatformowa i faktycznie obiektowa.)



Kod klasy generującej zdarzenia (Sygnały)

- Załóżmy klasę licznik, która posłuży jako przykład generatora zdarzeń oraz „słuchacza”.
- Na początku fragmenty odpowiadające za „emisję” zdarzeń.

przyklad.h

```
#include <QObject>
class Counter : public QObject
{
    Q_OBJECT
public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }
public slots:
    void setValue(int value);
signals:
    void valueChanged(int newValue);
private:
    int m_value;
};
```

przyklad.cpp

```
 (...)

void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}

 (...)
```

Kod klasy rejestrującej zdarzenia (SLOTY)

1. Podpinamy sygnał emitowany przez klasę „wysyłacz”, do naszej metody obsługi zdarzenia.
Uwaga! Qt kontroluje typy!

przyklad.h

```
#include <QObject>
class Counter : public QObject
{
    Q_OBJECT
public:
    Counter();
    int value() const { return m_value; }
public slots:
    void setValue(int value);
signals:
    void valueChanged(int newValue);
private:
    int m_value;
};
```

przyklad.cpp

```
 (...)
Counter::Counter()
{
  m_value = 0;
  connect( wysylacz,
           SIGNAL(valueChanged(int)),
           this,
           SLOT(setValue(int)
         );
}


void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
    }
}

 (...)
```

Obsługa komunikatów aplikacje WEB

- Java – **Struts** – krótki opis ponieważ mają państwo zajęcia z zaawansowanego programowania internetowego.
- Koncentracja na zagadnieniu przekazywania i obsługi komunikatów.
- Czyli:
  - Komunikaty przekazujemy:
    - jako atrybuty formularza
    - jako część urla na podstawie którego określana jest akcja, która ma zostać wywołana
  - Akcje obsługujące komunikaty potrafią interpretować dane przesyłane w formularzach.

- Dziękuję za uwagę.

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