

Programowanie w środowiskach RAD

Język C++ w środowiskach RAD

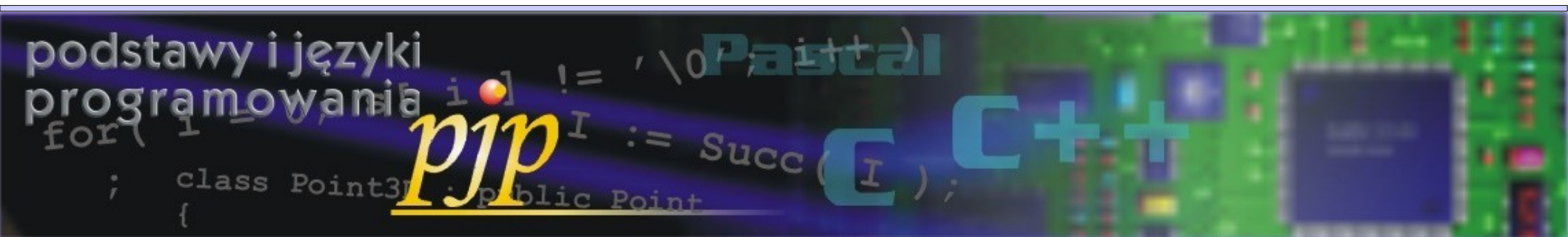
Roman Simiński

roman.siminski@us.edu.pl

www.siminskionline.pl

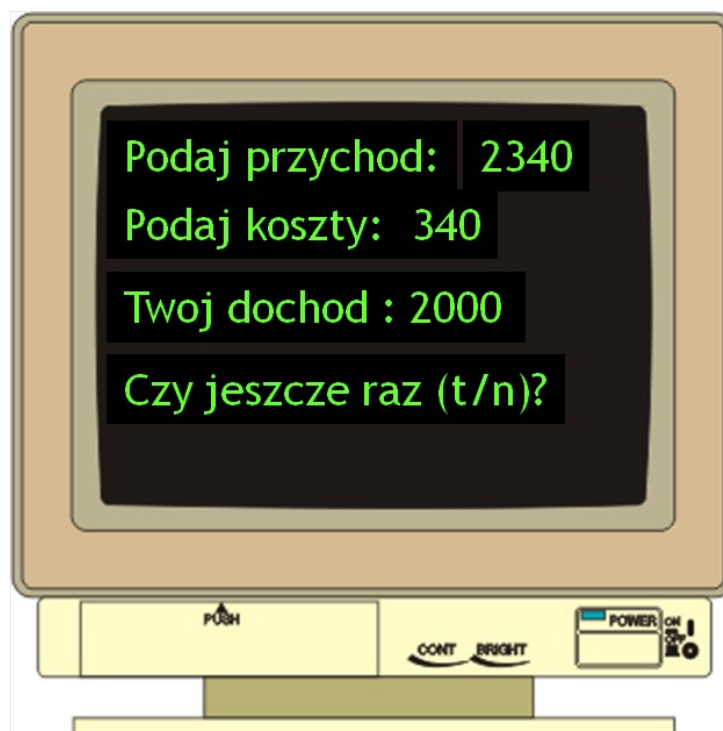
Programowanie sterowane zdarzeniami

Geneza, koncepcja, rodzaje, przykłady



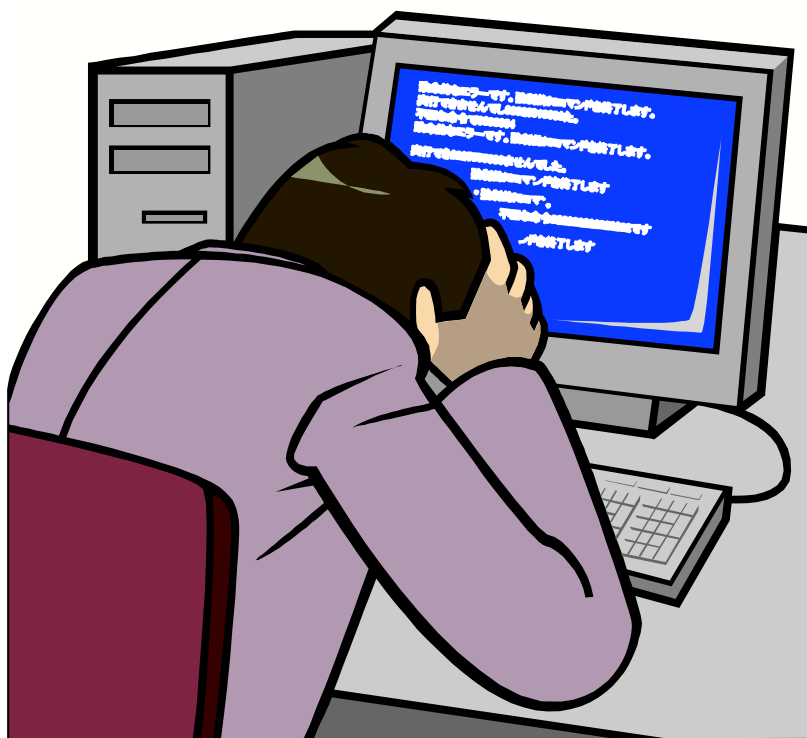
Jeszcze wcale nie tak dawno temu ...

Komputery wykorzystywały tekstowy tryb pracy monitora a użytkownicy ...



Jeszcze wcale nie tak dawno temu ...

... mieli tego dość!



Podstawą programowania były „sekwencja” i „dyktatura”

- Program ma ściśle określony początek i koniec.
- *Sekwencja* — wykonanie programu od początku do końca, pod „dyktando” kolejnych instrukcji napisanych przez programistę.
- *Dyktatura* — program dominuje, realizuje scenariusz określony kodem, użytkownik wykonuje operacje pod dyktando programu.
- *Nikła interakcja* — informacje wprowadzane przez użytkownika mają wpływ na wykonanie programu, ale zwykle jest on niewielki.
- To jest:

Programowanie kierowane przepływem sterowania
flow driven programming

```
int main()
{
    Data data;
    init();
    readData( &data );
    processData( &data );
    showData( &data );
    done();
    return 0;
}
```

Przetwarzanie
wsadowe

```
int main()
{
    char key;
    do
    {
        showMenuItems();
        switch( key = getKey() )
        {
            case '1' : doAction1();
                       break;
            case '2' : doAction2();
                       break;
        }
    }
    while( key != ESC );
    return 0;
}
```

Prosta
interakcja

Programowanie kierowane przepływem sterowania

Flow driven programming

- Jest to tradycyjna technika programowania, w ramach której program wykonywany jest zgodnie z aktualną ścieżką przepływu sterowania, zmienianą przez instrukcje warunkowe i iteracyjne, w oparciu o dane zewnętrzne i wewnętrzne, dostarczane do programu oraz wartościowane w zdeterminowanych momentach jego wykonania.
- Ta technika programowania stosowana jest powszechnie w jednozadaniowych systemach, ukierunkowanych na przetwarzanie sterowanie prostymi poleceniami klawiaturowymi, zwykle w środowiskach znakowych.
- Aktualnie stanowi postawę realizacji systemów preferujących przetwarzanie wsadowe, wykorzystujących uproszczoną interakcję z użytkownikiem.

A co z programami wykorzystującymi tryb graficzny?

Tryb graficzny wykorzystywany był tylko w wyspecjalizowanych aplikacjach:

- programach graficznych — tworzenie i obróbka,
- systemach CAD/CAM (Pierre Bézier i Paula de Casteljau),
- grach komputerowych,
- programach specjalizowanych — np. poligrafia.

Problemy:

- Problemy z pracą w trybie graficznym (karty, tryby, sterowniki).
- Zbyt słaby sprzęt, brak wsparcia ze strony systemu operacyjnego.
- Brak standaryzacji w zakresie graficznego interfejsu użytkownika.
- Utrudniona nawigacja — raczkujące urządzenia wskazujące.

W końcu jednak komputery „zmeźniały”

Powstają systemy operacyjne wykorzystujące:

GUI – *graphical user interface*

czyli

*graficzny podsystem komunikacji
z użytkownikiem*

zwany też:

graficznym interfejsem użytkownika

Prawdziwy zamęt wprowadziły jednak „gryzonie”...

Czy GUI wpływa na sposób programowania aplikacji?

Tak – aplikacje z GUI programuje się trudniej

Ale prawdziwe zamieszanie wprowadza pojawienie się w powszechnym użytku:

Komputerowej myszy!



Od Dyktatora do Sługi – upadek programisty... ?

Kiedyś to ja
dyktowałem warunki, a teraz
muszę kombinować, co
on kliknie i kiedy... !

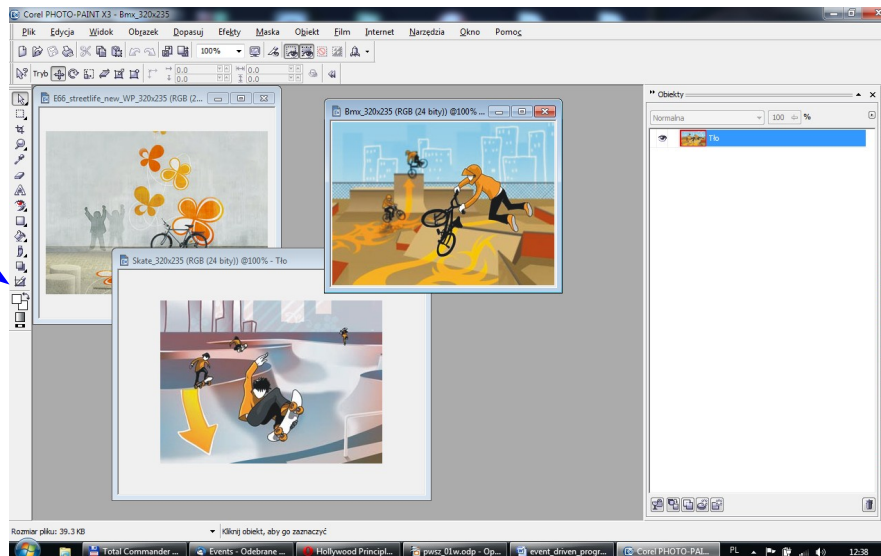


Kliknę sobie
tam, gdzie chcę i kiedy chcę!
Ten komputer jest
mój...!



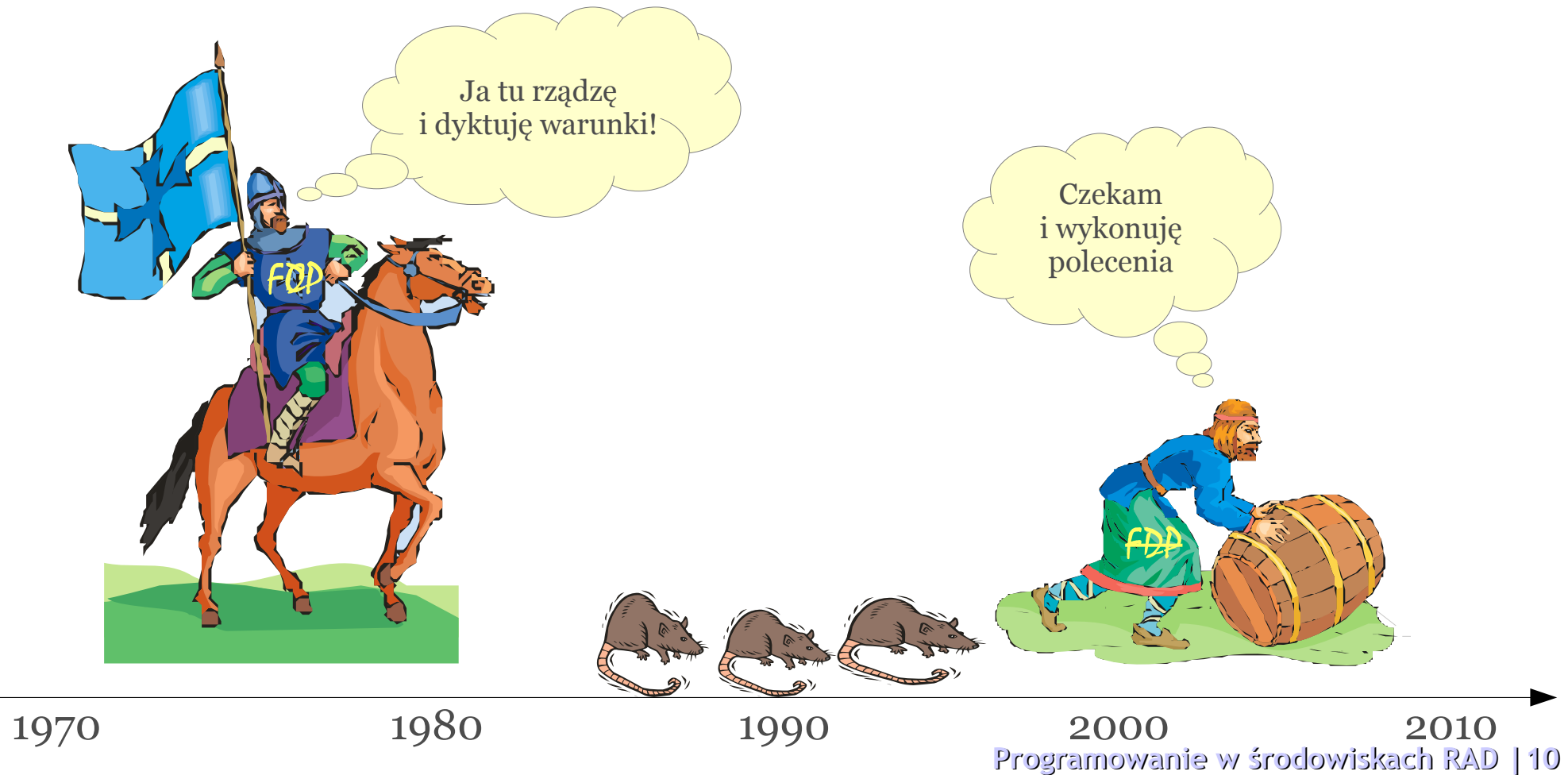
Świat
programisty

Świat
użytkownika

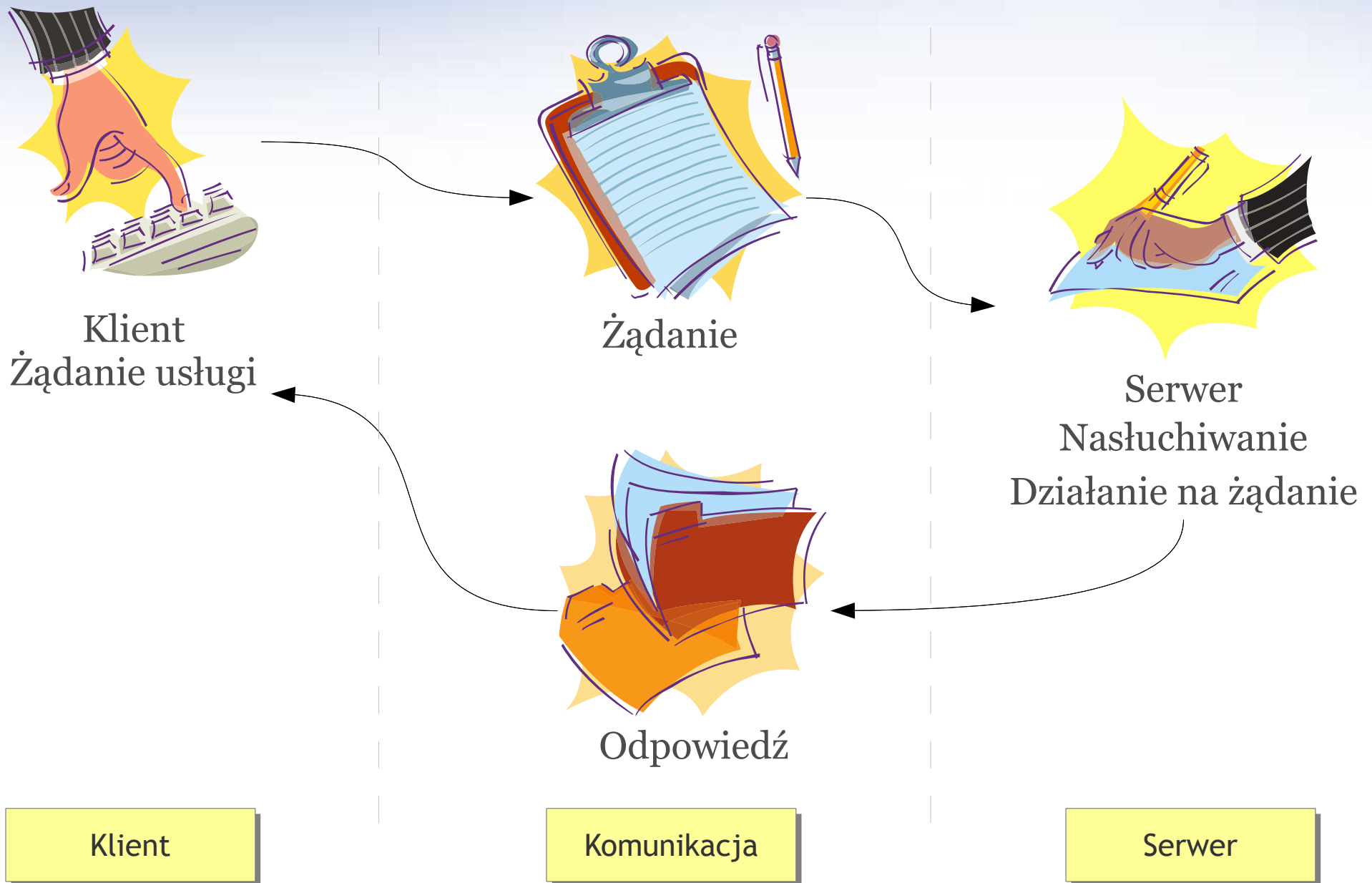


Zmierzch programowania sterowanego przytywem – FCP

- Rozwój GUI i manipulatorów myszopodobnych zmienił techniki programowania.
- Na zmiany wpłynął również rozwój architektury *klient-serwer* oraz *protokołów* wykorzystujących tę koncepcję (czyli większość protokołów internetowych).



Klient-serwer, sieć, przetwarzanie rozproszone



Programowanie sterowane zdarzeniami

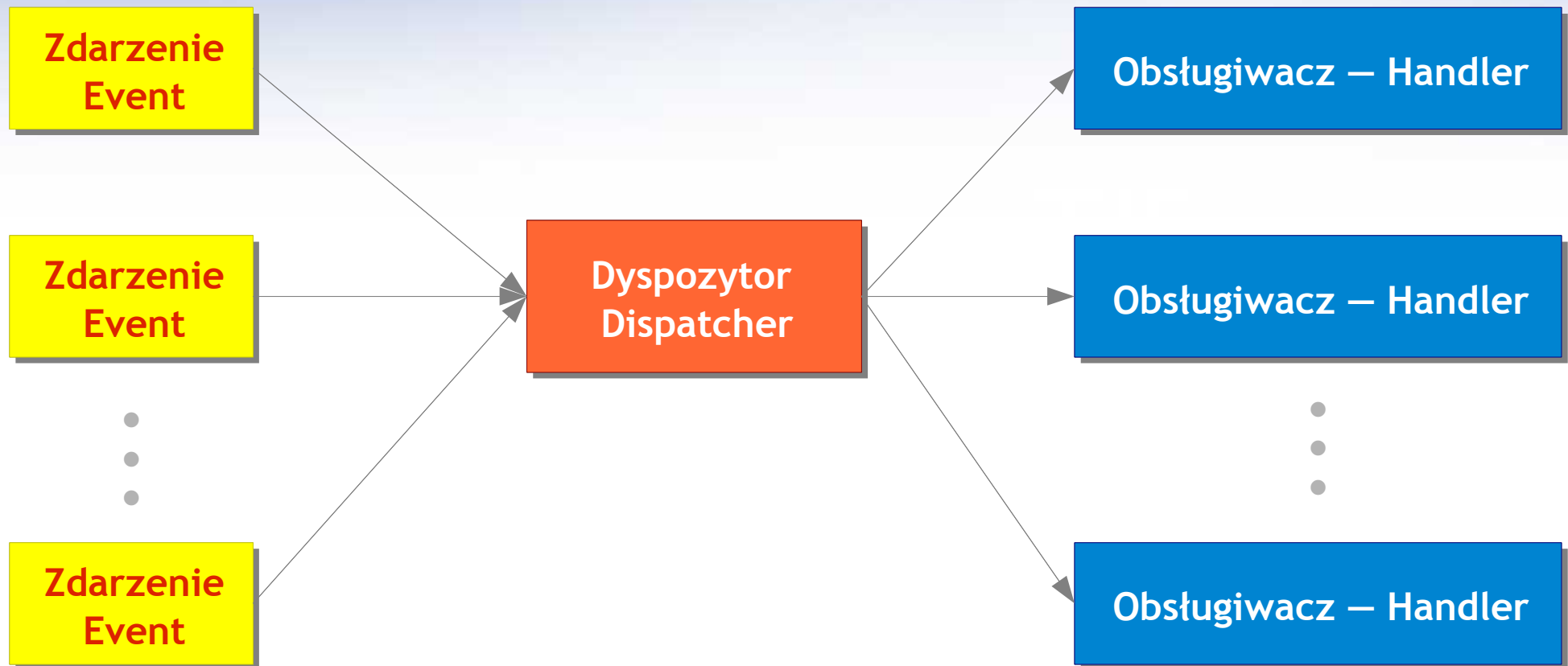
Event driven programming — EDP

- Nowa (relatywnie) technika programowania, zakładająca że działanie programu polega na wykonywaniu akcji będących odpowiedzią na *zdarzenia* dotyczące programu.
- Zdarzenia mogą pochodzić z *otoczenia programu* (użytkownik, system, sieć) lub z jego *wnętrza*.
- Zdarzenia powstają zwykle asynchronicznie w stosunku do działania programu, ten nie kontroluje momentu powstania zdarzeń oraz ich źródła (za wyjątkiem zdarzeń wewnętrznych).

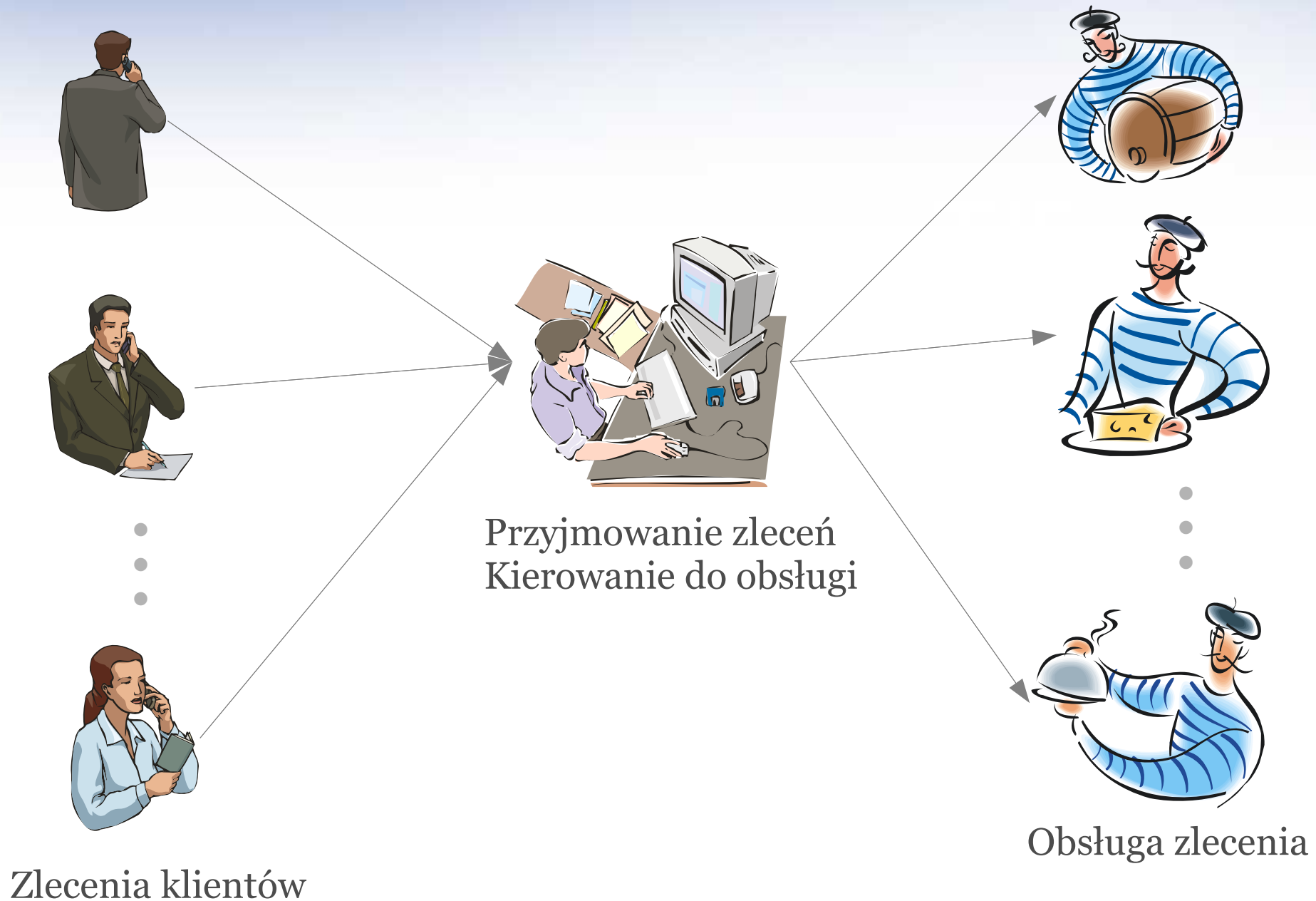
Zasada Hollywood – Hollywood Principle



Event driven programming – koncepcja organizacji



Koncepcja stara jak świat i powszechnie znana

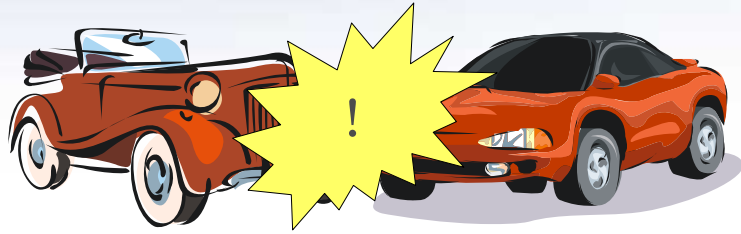


Zdarzenia – events

Zdarzenie (ang. *event*) – wynik asynchronicznej w stosunku do działania programu akcji *użytkownika*, *urządzenia* lub *programu*. Zdarzenie jest *rejestrowane*, oraz zapamiętywany jest jego *kontekst*.

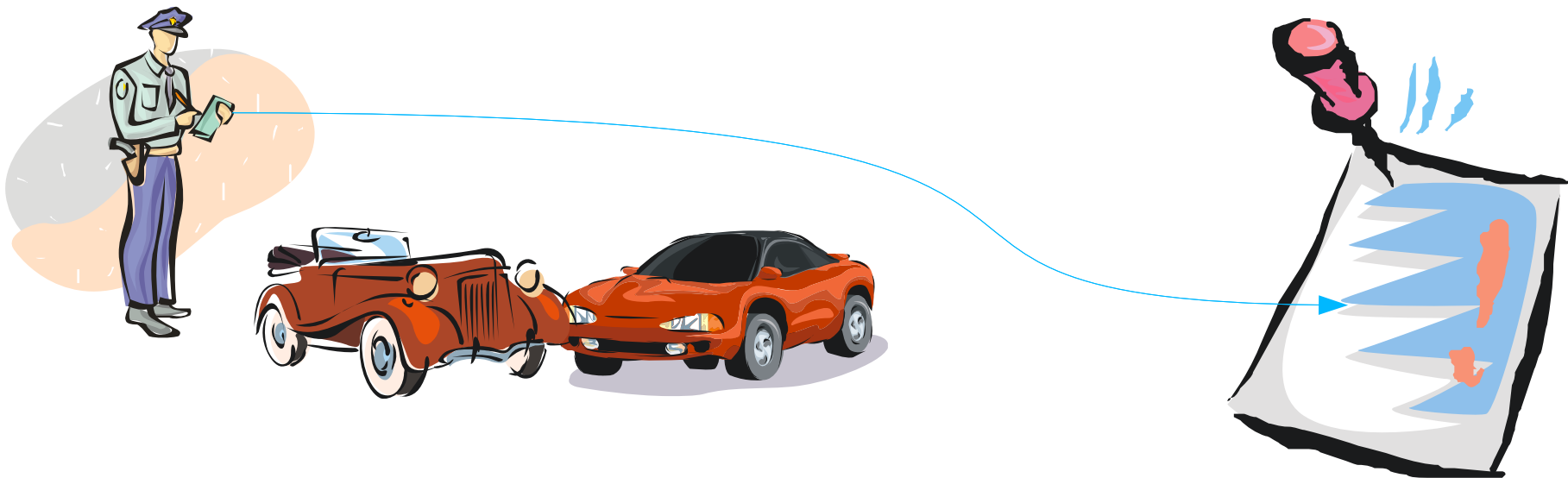
- Zdarzenie w sensie *programistycznym*, to informacja pewnym zdarzeniu *rzeczywistym*, które jest istotne dla systemu informatycznego i powinno zostać w pewien sposób obsłużone.
- Zdarzenia generowane są zwykle przez *otoczenie systemu* – użytkowników, inne systemy, urządzenia sprzętowe (sensory, czujniki), system operacyjny i jego składowe.
- Zdarzenia generowane są również przez *sam system* – różne składowe systemu mogą generować być źródłem zdarzeń kierowanych do *innych składowych* tego samego systemu jak i do jego *otoczenia*.

Zdarzenia – events



Zdarzenie jako element rzeczywistości

Zdarzenie jako informacja dla systemu



Źródła zdarzeń



Przykład rekordu opisu informacji o zdarzeniu

Przykład opisu informacji o zdarzeniu — rekord zawierający pola to typie zdarzenia, umownym kodzie w ramach danego typu, informacje o klawiaturze i myszy:

```
struct EventInfo
{
    int what;      // Rodzaj zdarzenia
    int code;      // Kod zdarzenia
    int key;       // Informacja o zdarzeniu klawiaturowym
    int x, y;      // Informacja o pozycji kursora myszy
    int buttons;   // Informacja o stanie przycisków myszy
};
```

W rzeczywistości rekordy opisu zdarzenia są zwykle bardziej skomplikowane i zawierają więcej informacji. Powyższy przykład ma charakter poglądowy.

Procedury obsługi zdarzeń – event handlers

Procedura obsługi zdarzenia (ang. *event handler*) – wydzielony fragment oprogramowania obsługujący zdarzenie lub zdarzenia odpowiedniego rodzaju.

- W obrębie aplikacji to najczęściej podprogram otrzymujący informacje o zaistniałym zdarzeniu, obsługujący to zdarzenie zgodnie z logiką aplikacji.
- Procedury obsługi są zazwyczaj dedykowane dla konkretnych typów zdarzeń.
- Po pomyślnej obsłudze zdarzenia informacje o nim są zerowane lub w opisie zdarzenia umieszcza się informację o jego obsłużeniu.

Procedury obsługi zdarzeń – event handlers

Przykładowa, hipotetyczna procedura obsługi zdarzenia pochodzącego z myszki (zakładamy, że *Control* to typ elementów okna dialogowego):

```
void handleMouseDown( EventInfo event )
{
    Control control = findControlOnXY( event.x, event.y );
    if( control == NULL )
        return;
    switch( control.type )
    {
        case NORMAL_BUTTON : pressButton( control );
                             event.what = EV_NONE;
                             break;
        case RADIO_BUTTON   : markRadioButton( control );
                             event.what = EV_NONE;
                             break;
        case CHECK_BUTTON   : markCheckBox( control );
                             event.what = EV_NONE;
                             break;
    }
}
```

Dyspozytor – dispatcher

Dyspozytor (ang. *dispatcher*) – zadaniem dyspozytora jest pobieranie napływających zdarzeń, identyfikowanie ich i kierowanie do odpowiednich procedur obsługi (żargonowo: *handlerów*).

- Dyspozytor pracuje zazwyczaj iteracyjnie — wyjście z iteracji następuje po zidentyfikowaniu zdarzenia końca.
- Dyspozytor powinien radzić sobie w sytuacji, gdy zdarzenie nie zostało obsłużone w żadnej z procedur obsługi.
- Dyspozytor może dokonywać konwersji zdarzeń — dostosowując je do specyfiki działania aplikacji.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;
    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Definicja rekordu opisu zdarzenia

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;
    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Pętla, czyli iteracja bez określonego warunku zakończenia (czasem instr. *loop*)

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;

    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Sprawdzenie, czy dostępne jest następne zdarzenie. Jeżeli tak, to jest pobierane, jeżeli nie to koniec działania.

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;

    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Czy wykryte zdarzenie nie jest czasem sygnałem zakończenia programu?

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;

    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Wstępne przetworzenie informacji o zdarzeniu,
konwersja, zamiana, czasem sztuczki i triki

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;

    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Czy jest rzeczywiście coś do zrobienia?

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;

    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Przekazanie zdarzeń do konkretnych procedur obsługi

Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Pseudokod głównej iteracji programu — pierwsza faza pracy dyspozytora, podejście proceduralne.

```
int main()
{
    EventInfo event;

    for( ; ; )
    {
        event = getNextEvent();
        if( event.what == EV_QUIT )
            break;
        preprocessEvent( event );
        if( event.what != EV_NONE )
            forwardEvent( event );
        else
            idleAction();
    }
    return EXIT_SUCCESS;
}
```

Gdy nie wykryto oczekującego na obsługę zdarzenia — obsługa procesu realizowanego w tle

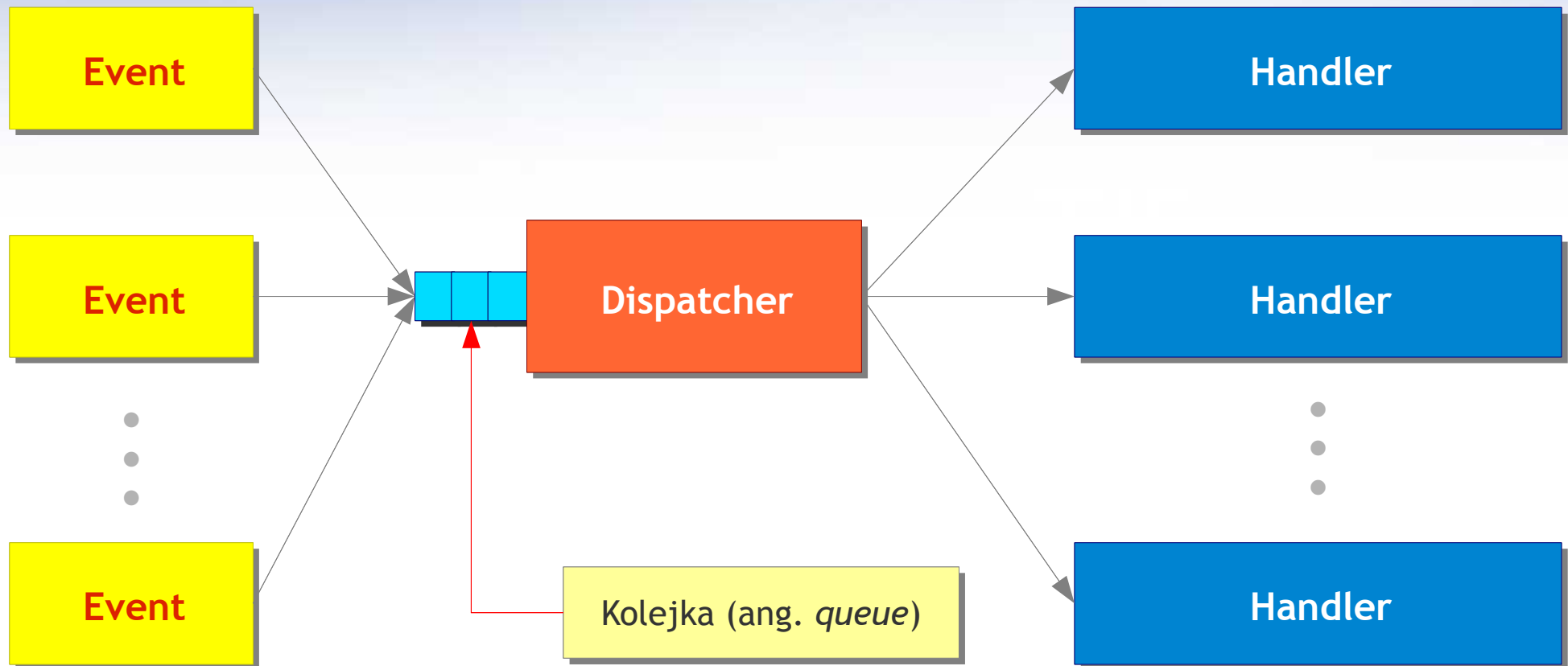
Uwaga — to tylko przykład jednego z możliwych sposobów organizacji pracy dyspozytora.

Dyspozytor – dispatcher

Szczegółowa identyfikacja zdarzenia i przekazanie do procedur obsługi zgodnie z typem zdarzenia.

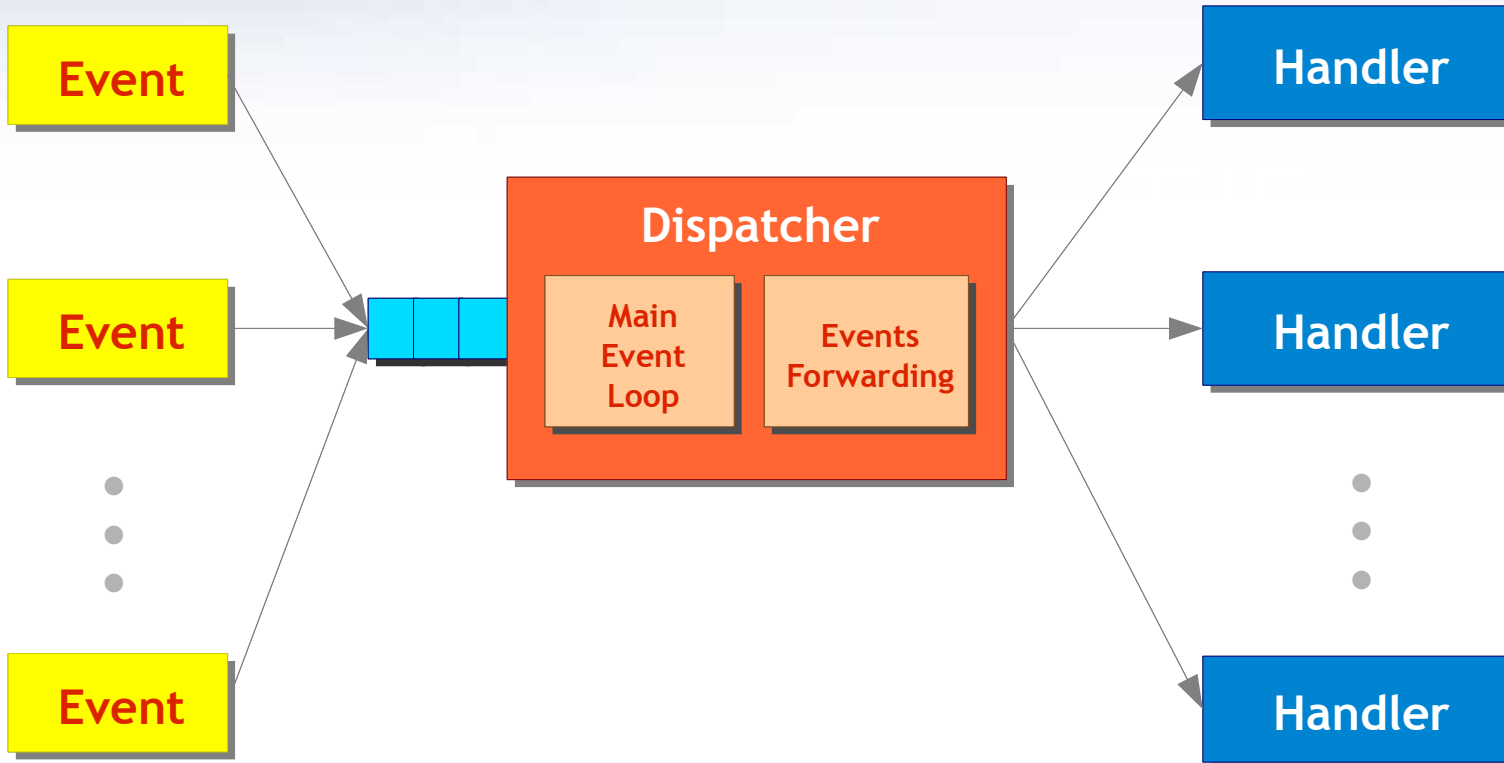
```
void forwardEvent( EventInfo event )
{
    switch( event.what )
    {
        case EV_MOUSE_DOWN : handleMouseDown( event );
                             break;
        case EV_KEY_DOWN    : handleKeyPress( event );
                             break;
        case EV_APP_EVENT   : handleAppEvent( event );
                             break;
        default              : handleUnknown( event );
                             break;
    }
    if( event.what != EV_NONE )
        processUnhandledEvent( event );
}
```

A gdy zdarzenia pojawiają się zbyt szybko...

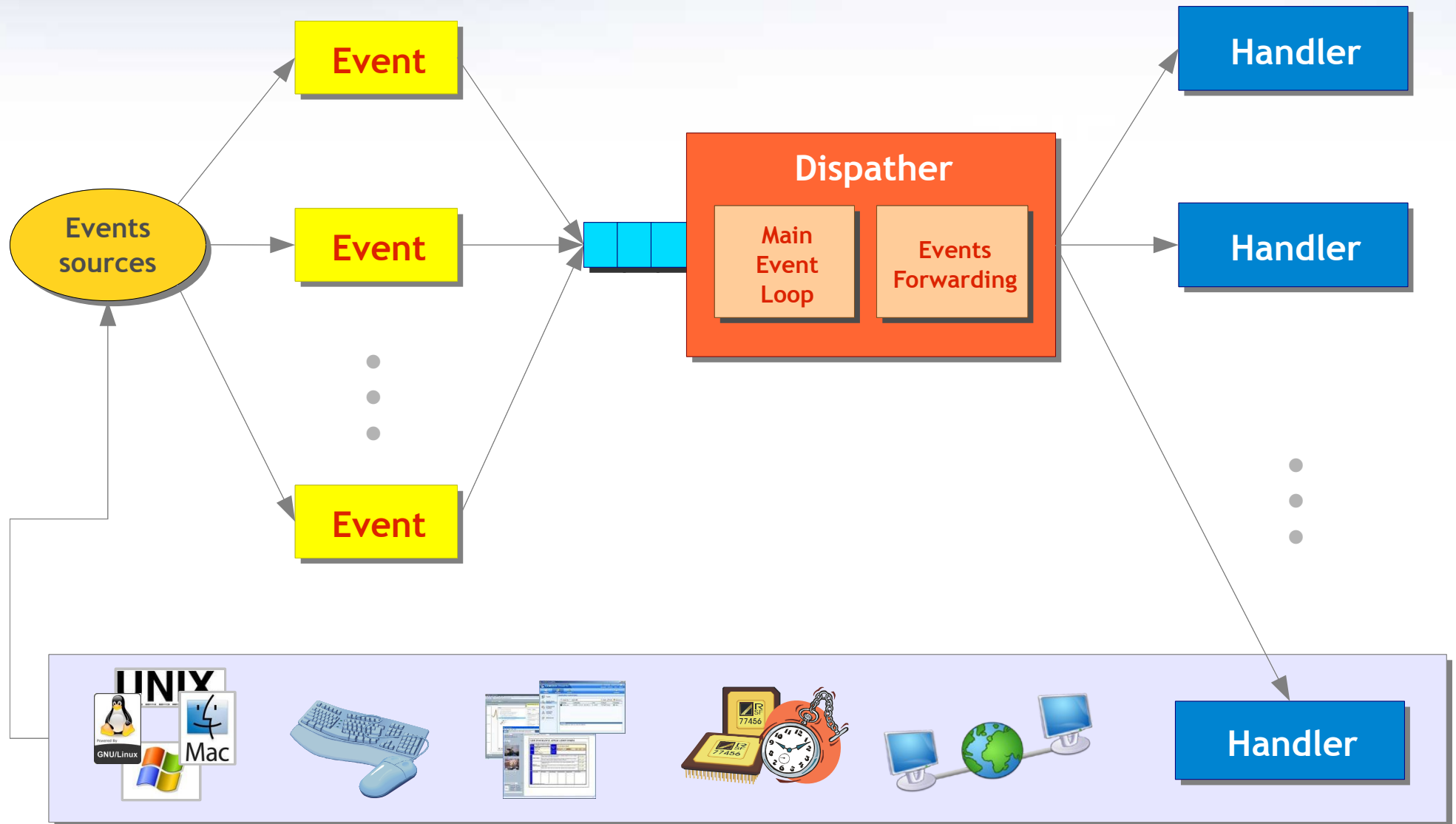


Kolejka zdarzeń(ang. *event queue*) – struktura danych typu FIFO, przechowująca informacje o zdarzeniach. Kolejka może być *priorytetowa*, istnieje możliwość reorganizowania kolejności kolejkowanych zdarzeń.

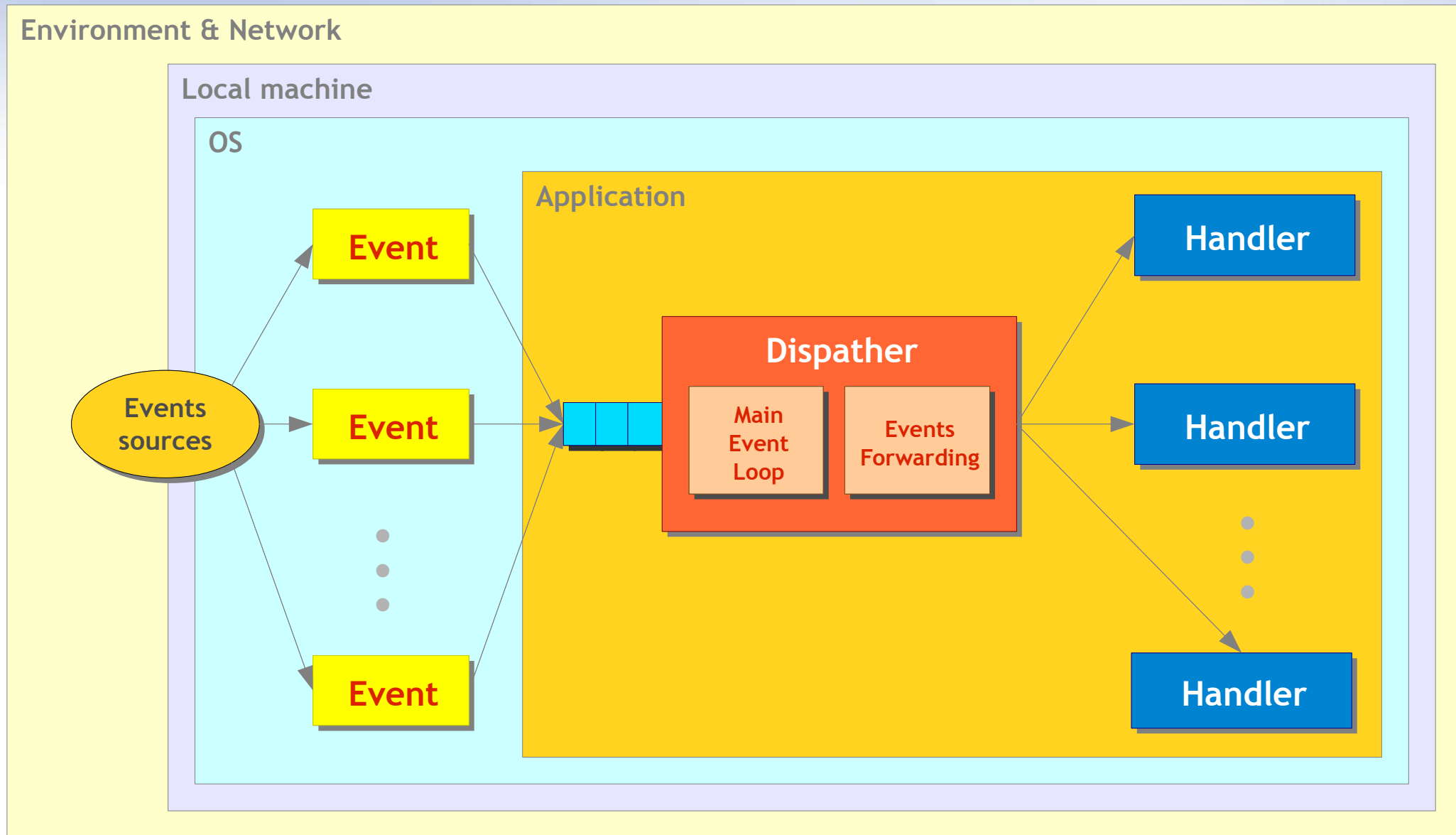
Model EDP bardziej szczegółowo...



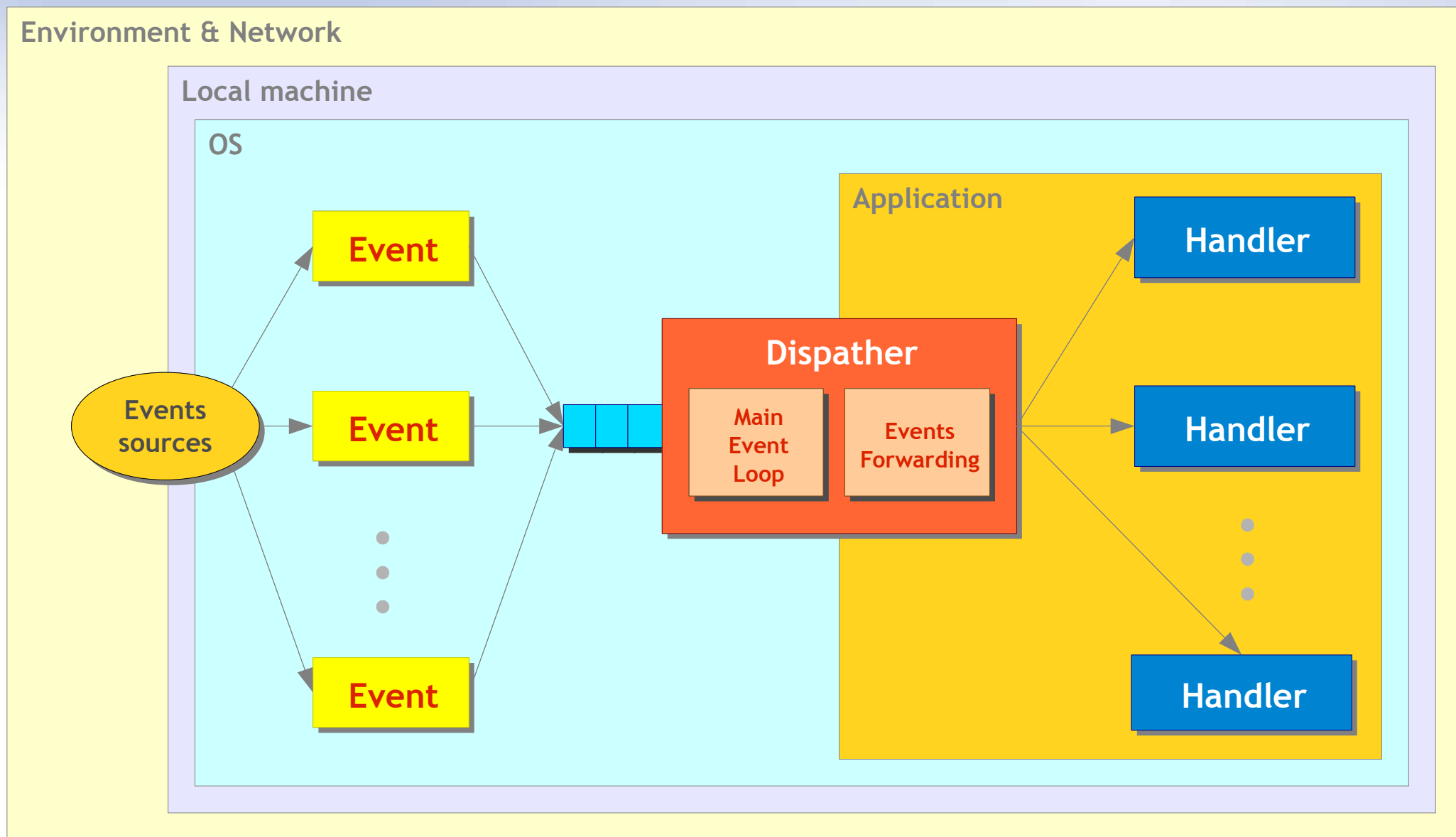
Model EDP jeszcze bardziej szczegółowo, cd. ...



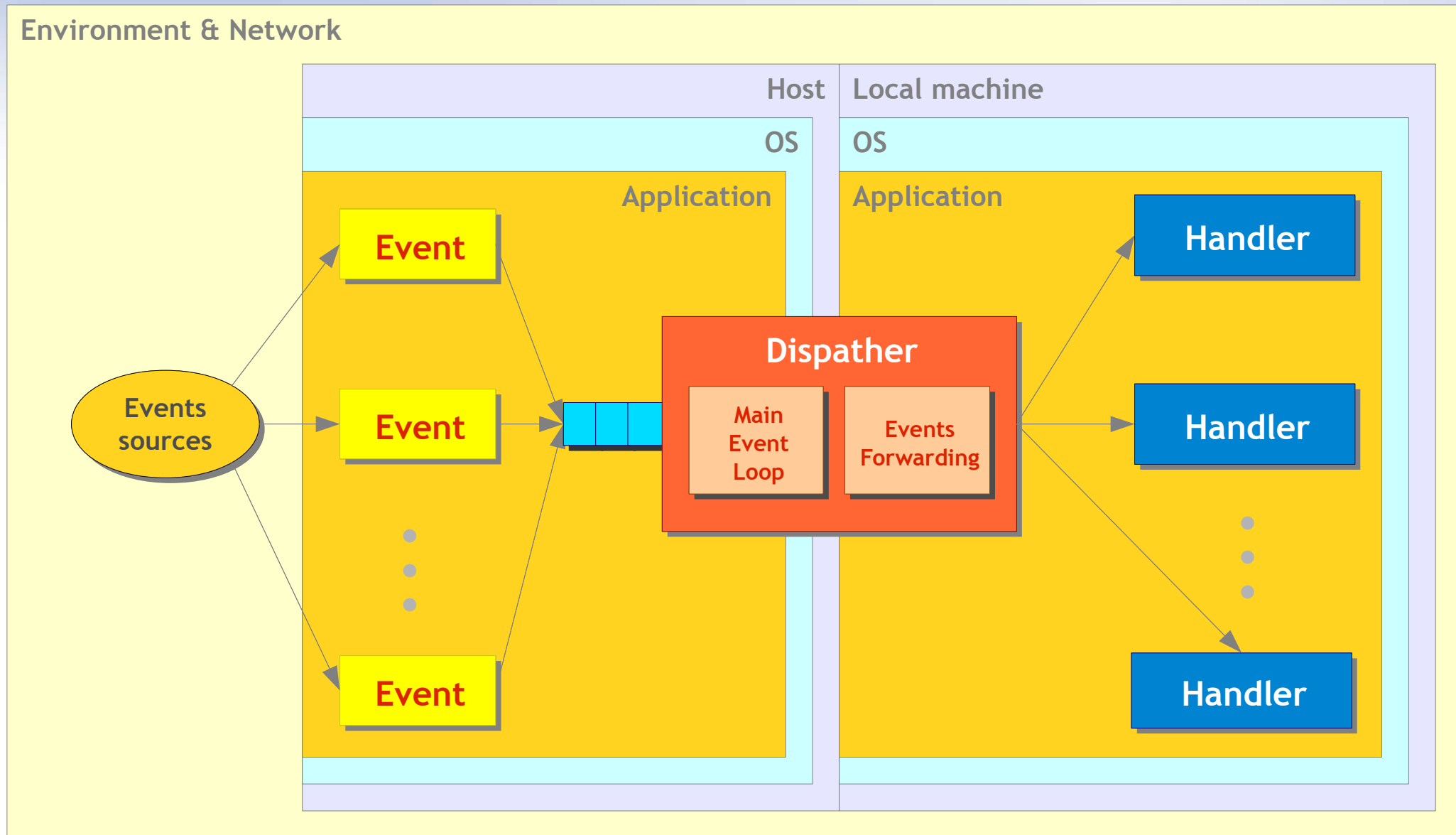
EDP, ale gdzie?



EDP, ale gdzie?

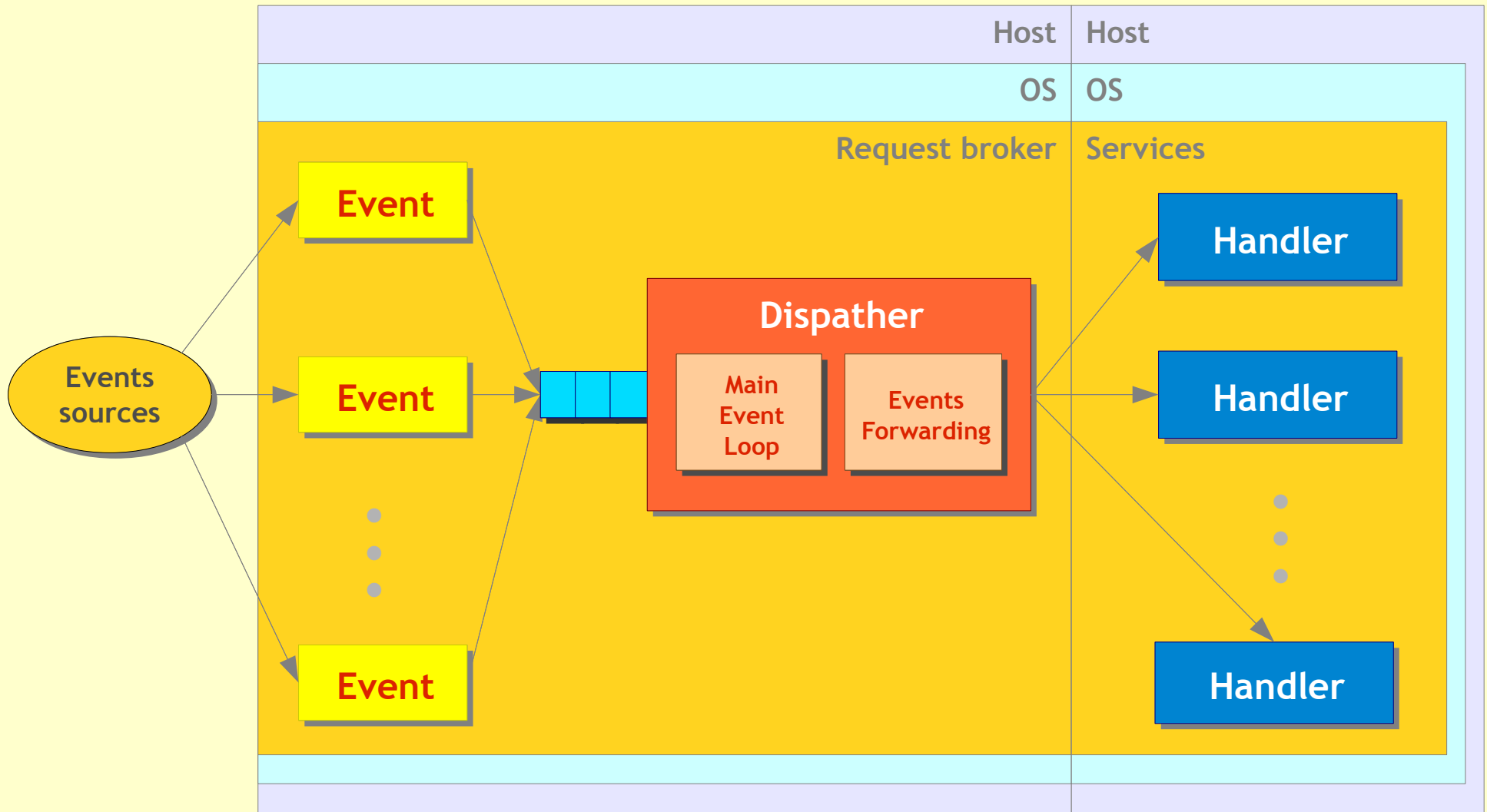


EDP, ale gdzie?



EDP, ale gdzie?

Environment & Network



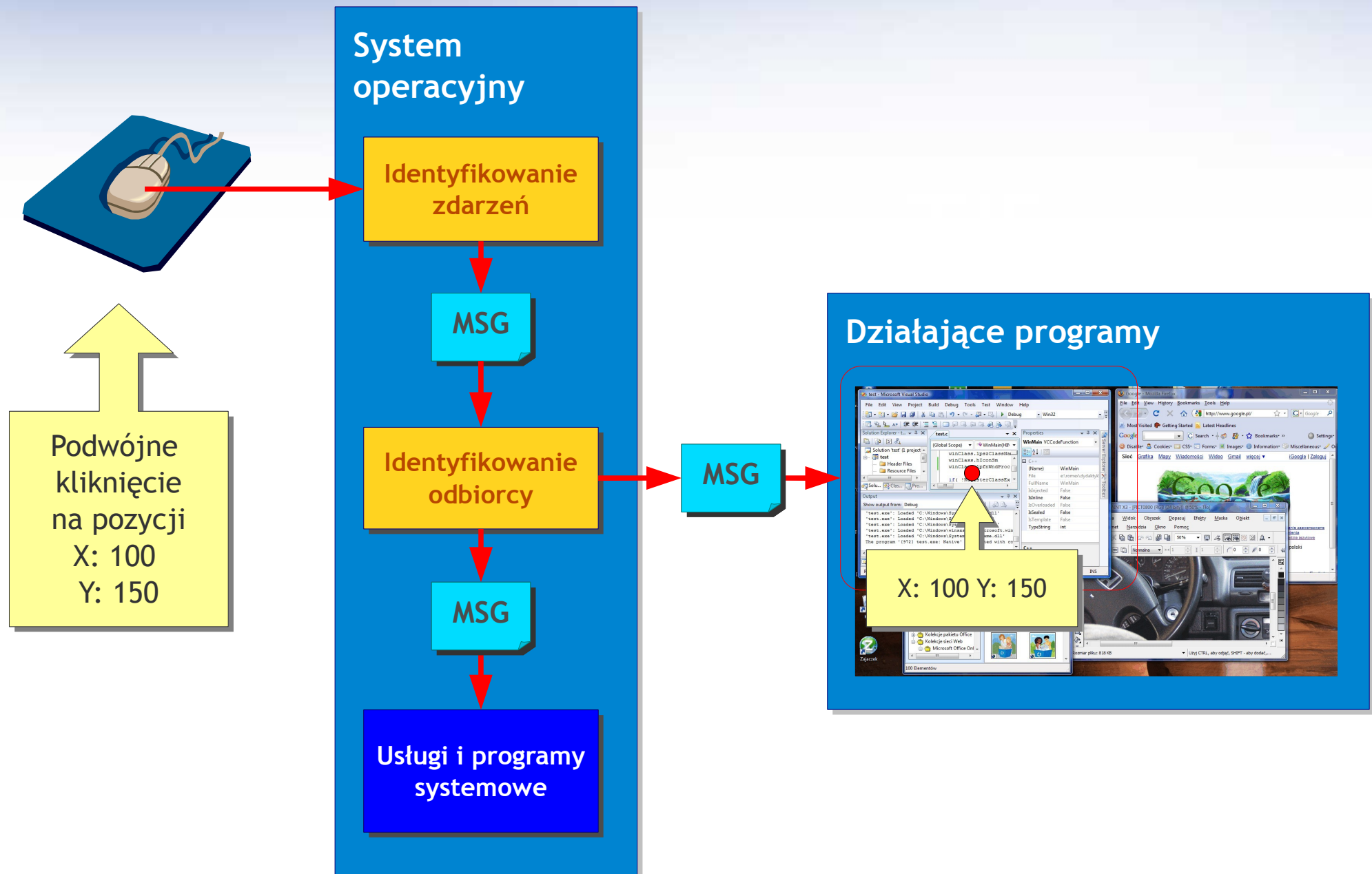
Event driven programming a OOP i GUI

- Koncepcja programowania sterowanego zdarzeniami jest niezależna od przyjętej metody programowania i stosowanego języka.
- Podejście obiektowe jest jednak metodą programowania idealnie pasującą do koncepcji programowania sterowanego zdarzeniami.
- Programowanie sterowane zdarzeniami kojarzy się głównie z systemami wykorzystującymi GUI — jednak ta koncepcja programowania może być skutecznie wykorzystywana w systemach bez GUI (np. procesor i mechanizm przerwań jest *event driven*!).
- Wiele zagadnień jest podatnych na rozwiązanie *event driven* — np. parsery, w tym parsery XML.
- Większość bibliotek obsługi GUI zintegrowanych albo bliskich systemowi operacyjnemu (WinAPI, XWindows, GTK) zaimplementowano w językach nieobiektowych (zwykle C), choć w sensie koncepcji są one przynajmniej *obiektoowo zorientowane*.

Event driven programming a OOP i GUI, cd. ...

- Programowanie na poziomie bibliotek GUI bliskich systemowi operacyjnemu jest żmudne. A co żmudne, jest trudne, a przynajmniej uciążliwe.
- Twórcy narzędzi dla programistów od lat pracują nad bibliotekami GUI umożliwiającymi łatwiejsze i efektywniejsze programowanie.
- Te biblioteki ze większości przypadków są *obiektove*.
- Historycznie najwcześniejszymi bibliotekami były *MFC* (Microsoft) i *OWL* (Borland). Borland firmował również świetną! bibliotekę *TurboVision* dla DOS.
- Wraz z pojawieniem się pakietu Delphi firma Borland udostępnia bibliotekę *VCL*, dostępną aktualnie we wszystkich narzędziach wywodzących się z firmy Borland.
- W świecie społeczności OpenSource powstają biblioteki *wxWidgets* oraz *Qt*, ta ostatnia staje się podstawą programu *QTDesigner* oraz pakietu *QTCreator*, aktualnie firmowanego przez Nokię.

Event driven programming a WinAPI



Event driven programming a WinAPI

```
typedef struct {  
    HWND hwnd;  
    UINT message;  
    WPARAM wParam;  
    LPARAM lParam;  
    DWORD time;  
    POINT pt;  
} MSG, *PMSG;
```

- *hwnd* — identyfikator okna, które otrzymuje komunikat.
- *message* — identyfikator komunikatu, starszy bajt zarezerwowany dla systemu, młodszy do wykorzystania dla programów.
- *wParam*, *lParam* — dodatkowe informacje o komunikacie, zależne od jego rodzaju.
- *time* — Czas „zaistnienia” komunikatu.
- *pt* — pozycja kursora myszy w momencie „zaistnienia” komunikatu.

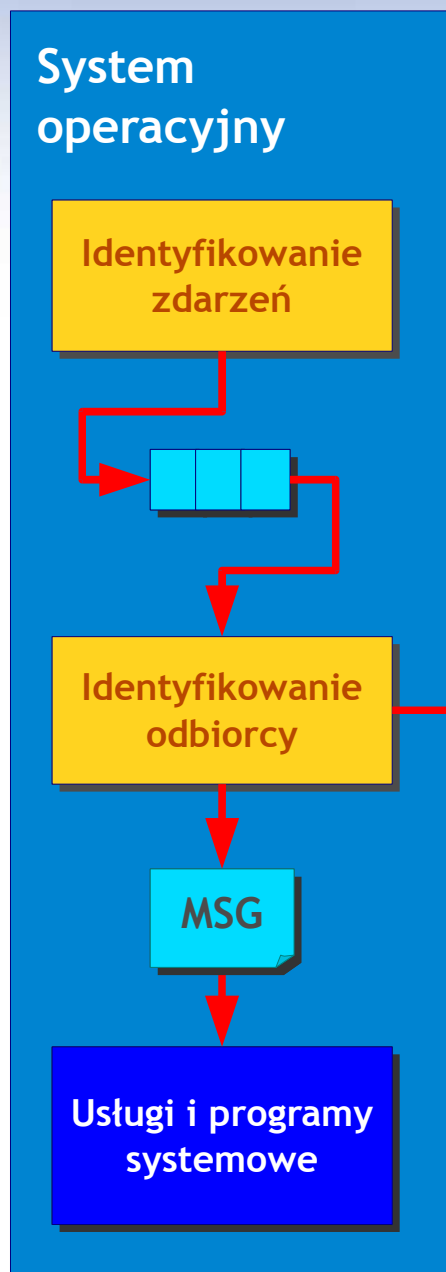
Event driven programming a WinAPI

```
typedef struct {  
    HWND hwnd;  
    UINT message;  
    WPARAM wParam;  
    LPARAM lParam;  
    DWORD time;  
    POINT pt;  
} MSG, *PMSG;
```

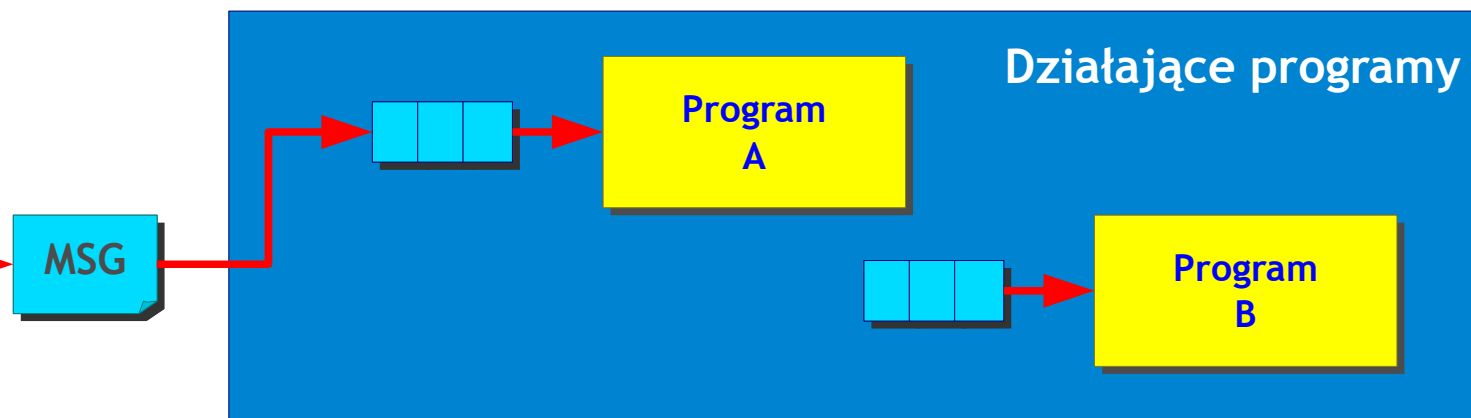
To pole ma znaczenie podstawowe i określa charakter komunikatu

- Identyfikator komunikatu *message* zawiera najczęściej nazwaną stałą określającą zaistniałe zdarzenie.
- Procedura obsługi zdarzenia sprawdza pole *message* i podejmuje stosowną obsługę, być może posiłkując się dodatkowymi polami rekordu *MSG*.
- Przykład — identyfikator *WM_PAINT* oznacza, że zaistniało zdarzenie powodujące konieczność przerysowania zawartości obszaru klienckiego okna, co powinna zrealizować procedura obsługi komunikatów.

Event driven programming a WinAPI



- Komunikaty opisujące zdarzenia są kolejgowane.
- 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).

Dyspozytor w WinAPI

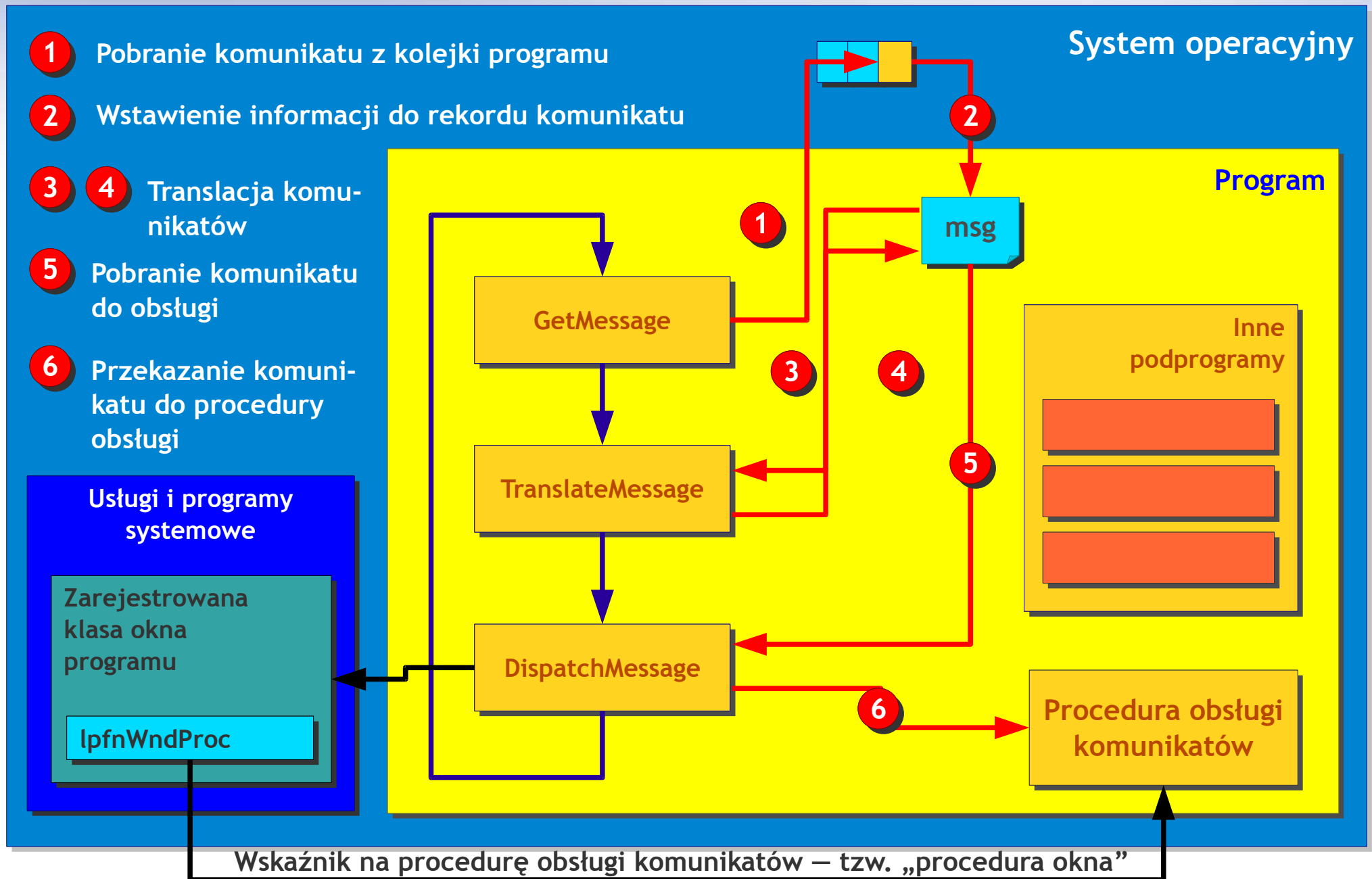
```
int WINAPI WinMain( HINSTANCE hInst, HINSTANCE hPrev,  
                    LPSTR lpszCmdLine, int nCmdShow )  
{  
    WNDCLASSEX winClass;  
    HWND hWndMain;  
    MSG msg;  
    . . .  
    ShowWindow( hWndMain, nCmdShow );  
    UpdateWindow( hWndMain );  
  
    while( GetMessage( &msg, NULL, 0, 0 ) > 0 )  
    {  
        TranslateMessage( &msg );  
  
        DispatchMessage( &msg );  
    }  
    return msg.wParam;  
}
```

Pobranie komunikatu z kolejki

Przekształcenie komunikatów
związanych z klawiaturą.

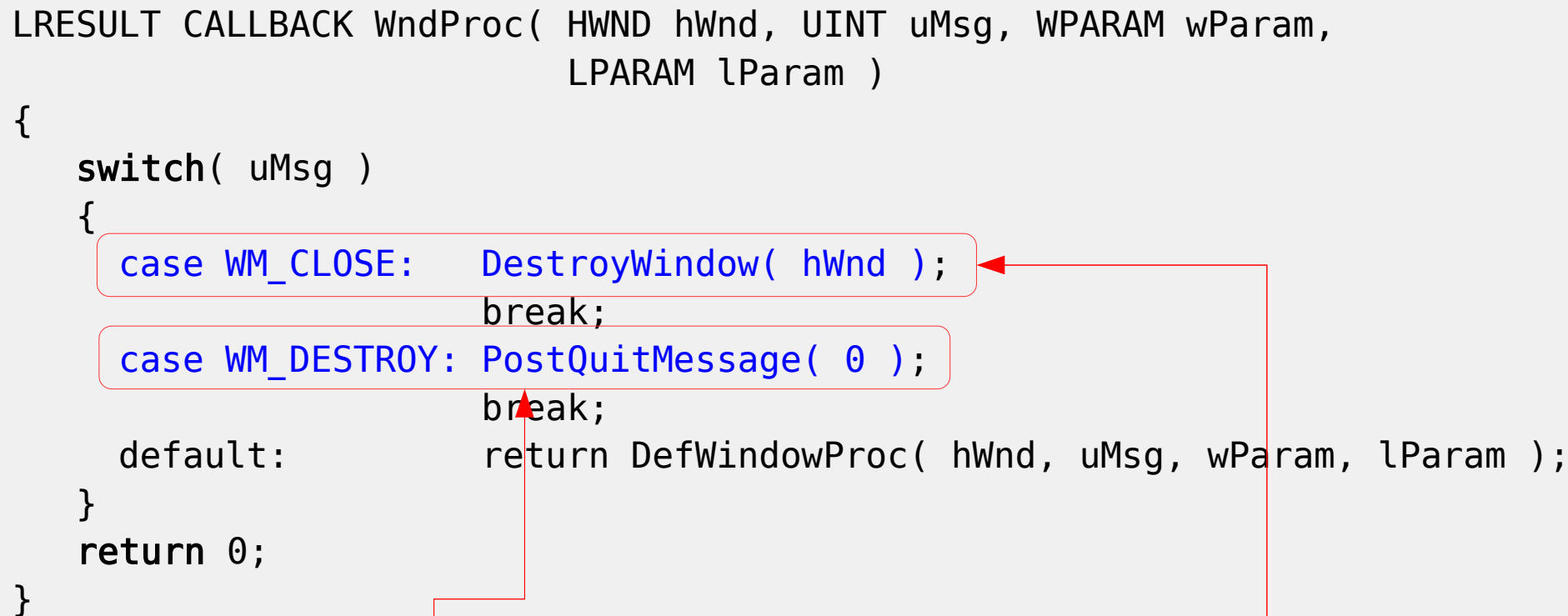
Odnalezienie i uruchomienie przez
system operacyjny
procedury obsługi komunikatu

Schemat obiegu komunikatów na poziomie WinAPI



Procedura okna – procedura obsługi komunikatów

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                          LPARAM lParam )
{
    switch( uMsg )
    {
        case WM_CLOSE: DestroyWindow( hWnd );
                       break;
        case WM_DESTROY: PostQuitMessage( 0 );
                       break;
        default:        return DefWindowProc( hWnd, uMsg, wParam, lParam );
    }
    return 0;
}
```



Program się kończy, wstaw komunikat WM_QUIT, który zakończy działanie iteracji sterowanej funkcją *GetMessage*

Komunikat oznaczający zdarzenie zamknięcia okna. Usuń okno główne i potomne, wstaw komunikat WM_DESTROY

Event driven programing... ale w czym?

- Programowanie na poziomie bibliotek bliskich systemowi operacyjnemu jest żmudne — to już było.
- Jest idealne dla pisania sprytnych programów blisko zintegrowanych z systemem operacyjnym, lub oprogramowania o wysokich wymaganiach wydajnościowych.
- Uważam że programowanie na tym poziomie ogranicza programistę tak samo, jak kiedyś ograniczało programowanie złożonych aplikacji w językach symbolicznych.
- Tematem dalszych wykładów będzie prezentacja koncepcji, technik i wybranych przykładów programowania z wykorzystaniem pakietów *C++ Builder* i biblioteki *VCL* oraz *QtCreator* i biblioteki *Qt*.

Dziękuję za uwagę

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