# Zdarzenia
# (*events*, *connection points*)

# Zdarzenia

**Serwer komunikuje się z klientem za pomocą zgłoszenia zdarzenia. Pozwala to na asynchroniczną pracę serwera, który zgłasza zaistnienie określonego stanu za pomocą serii zdarzeń. Odbywa się to poprzez wywołanie metod interfejsu, który implementowany jest przez klienta. Klient w takim przypadku posiada również cechy serwera, gdyż jest w stanie odbierać wywołania komponentu.**

# Zdarzenia

**Komponent musi mieć wstępnie zdefiniowany zestaw zdarzeń (strukturę interfejsu), przez które będzie się odwoływał do klienta. Dlatego interfejs, który implementowany jest przez klienta, musi być zdefiniowany wraz z interfejsami komponentu i wyróżniony jako odbiorca zdarzeń. W ten sposób komponent stawia wymagania w stosunku do klienta.**

# Zdarzenia

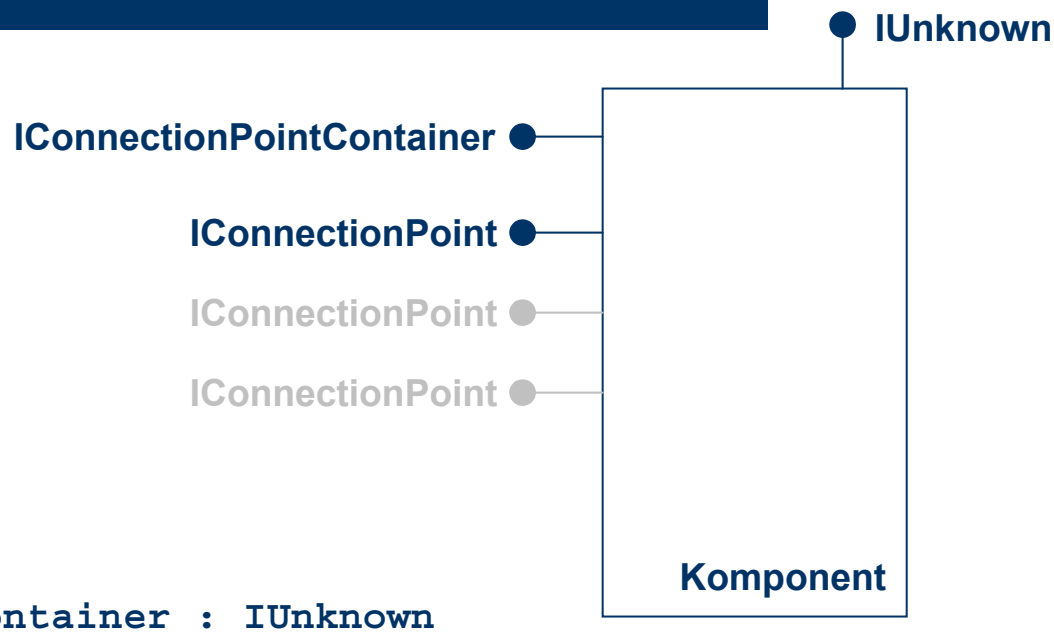
```
[ object, uuid(6764B148-93BE-40C1-B2BA-81650C57E33E), oleautomation ]
interface ICountdown : IUnknown
{
  ...
};

[ object, uuid(95C0B4CE-A39B-463A-BE4B-06137333BD8E), oleautomation ]
interface ICountdownEvents :IUnknown
{
  HRESULT EverySecondTick( [in] int ElapsedTime );
};

[
  uuid(D7EBA784-2BF1-11D3-A48B-0000861C844E), version(VERSION),
  helpstring("Timers 3.0 Type Library")
]
library CounterLib
{
  coclas Counter
  {
    interface ICountdown;
    [source] interface ICountdownEvents;

  }
};
```

# Zdarzenia
## `IConnectionPointContainer`

**Zanim klient uruchomi proces komponentu, który może zgłosić zdarzenie, musi przekazać mu swój interfejs. Przez przekazaniem interfejsu sprawdza czy posiadany interfejs odpowiada wymaganiom komponentu, a następnie przekazuje mu wskaźnik do swojego interfejsu `IUnknown`.**

**Klient weryfikuje swój interfejs za pomocą odwołań do interfejsu `IConnectionPointContainer`. Interfejs ten musi być zaimplementowany przez komponent.**

# Zdarzenia
## IConnectionPointContainer




```
interface IConnectionPointContainer : IUnknown
{
  HRESULT EnumConnectionPoints (
    [out] IEnumConnectionPoints ** ppEnum
  );
  HRESULT FindConnectionPoint (
    [in] REFIID riid,
    [out] IConnectionPoint ** ppCP
  );
}
```

# Zdarzenia
## IConnectionPointContainer

```
HRESULT FindConnectionPoint (
    [in] REFIID riid,
    [out] IConnectionPoint ** ppCP
  );
```

**Metoda `FindConnectionPoint` zwraca klientowi interfejs `IConnectionPoint`, który odpowiada posiadanemu interfejsowi. Gdy podany GUID nie odpowiada żadnemu z interfejsów wymaganych przez komponent, zwrócony zostaje błąd `E_NOINTERFACE`.**

```
HRESULT CCountdown::FindConnectionPoint(REFIID riid,
                                        IConnectionPoint ** ppCP)
{
  if (riid==IID_ICountdownEvents)
    return QueryInterface(IID_IConnectionPoint, (void**)ppCP);
  return E_NOINTERFACE;
}
```

# Zdarzenia
## IConnectionPointContainer

```
HRESULT EnumerateConnectionPoints (
    [out] IEnumConnectionPoints ** ppEnum
  );
```

**Metoda `EnumConnectionPoints` zwraca obiekt typu `IEnumConnectionPoints`, który pozwala na dostęp do wszystkich obiektów `IConnectionPoint` wspieranych przez komponent. Pełna implementacja tej metody jest istotna, gdy komponent wspiera różne rodzaje interfejsów zdarzeń. Może ona zwracać `E_NOIMPL`.**

# Zdarzenia
## IConnectionPoint

**Komponent dla każdego typu interfejsu źródłowego (odbierającego zdarzenia) powinien mieć zaimplementowany obiekt `IConnectionPoint`. W najprostszym przypadku, gdy jest tylko jeden rodzaj interfejsu źródłowego, komponent może bezpośrednio dziedziczyć `IConnectionPoint`. Metody tego interfejsu służą klientowi do zgłaszania (rejestracji) wskaźnika swojego `IUnknown`.**

```
interface IConnectionPoint : IUnknown
{
  HRESULT GetConnectionInterface( [out] IID * pIID );
  HRESULT GetConnectionPointContainer(
    [out] IConnectionPointContainer **ppCPC);
  HRESULT Advise( [in] IUnknown * pUnkSink, [out] DWORD * pdwCookie );
  HRESULT Unadvise( [in] DWORD dwCookie );
  HRESULT EnumConnections( [out] IEnumConnections ** ppEnum );
}
```

# Zdarzenia
## IConnectionPoint

```
HRESULT GetConnectionInterface( [out] IID * pIID );
```

**Metoda `GetConnectionPointInterface()` pozwala na identyfikację z jakim interfejsem źródłowym skojarzony jest obiekt `IConnectionPoint`. Metoda ta jest szczególnie użyteczna podczas wyliczania obiektów `IConnectionPoint` za pomocą obiektu `IEnumConnectionPoints`.**

# Zdarzenia
## IConnectionPoint

```
HRESULT GetConnectionPointContainer(
    [out] IConnectionPointContainer **ppCPC);
```

**Metoda `GetConnectionPointContainer()` zwraca rodzicielski obiekt `IConnectionPointContainer`.**

```
HRESULT Advise(
    [in] IUnknown * pUnkSink, [out] DWORD * pdwCookie );
```

**Metoda `Advise()` pozwala przekazać klientowi wskaźnik swojego interfejsu źródłowego (poprzez `IUnknown`), do którego przekazywane będą zdarzenia. Metoda ta zwraca identyfikator, który później pozwala rozróżnić klientów, w przypadku gdy więcej niż jeden korzysta z tego samego komponentu.**

# Zdarzenia
## IConnectionPoint

```
HRESULT Unadvise( [in] DWORD dwCookie );
```

**Metoda `Unadvise()` informuje komponent, aby nie próbował już wywoływać metod wskazanego klienta.**

```
HRESULT EnumConnections([out] IEnumConnections ** ppEnum);
```

**Metoda `EnumConnections()` zwraca obiekt `IEnumConnections` pozwalający na dostęp do wszystkich aktywnych interfejsów zgłoszonych przez klientów metodą `Advise()`.**

# Zdarzenia
## IConnectionPoint

```
HRESULT Advise(IUnknown * pUnkSink, DWORD * pdwCookie){
  *pdwCookie = 1;
  return pUnkSink->QueryInterface(IID_ICountdownEvents,
    (void**)&g_pOutGoing);
}

HRESULT Unadvise(DWORD dwCookie){
  g_pOutGoing->Release();
  g_pOutGoing=NULL;
  return S_OK;
}

HRESULT GetConnectionInterface(IID * pIID){
  *pIID=IID_ICountdownEvents;
  return S_OK;
}

HRESULT GetConnectionPointContainer(IConnectionPointContainer **ppCPC){
 return QueryInterface(IID_IConnectionPointContainer, (void**)ppCPC);
}
```

# Zdarzenia
# Klient

```
main () {
 try {
  CCountdownPtr pCountdown(CLSID_Counter);

  IConectionPointContainer* pCPContainer;
  IConectionPoint* pCP;
  DWORD dwCookie;

  pCountdown->QueryInterface(IID_IConectionPointContainer,
     (void**)&pCPContainer);
  hr=pCPContainer->FindConnectionPoint(IID_ICountdownEvents,&pCP);
  if (FAILED(hr)) ...;

  CCountdownEvents* pCountdownEvents=new CCountdownEvents;
  hr=pCP->Advise((IUnknown*)pCountdownEvents,&dwCookie);
  if (FAILED(hr)) ...;

  pCountdown->StartCounter(100);
  ... //do some job
  hr=pCP->Unadvise(dwCookie);
  pCP->Release();
  pCPContainer->Release();
  pCountdown=NULL;
 }
 catch(){
   ...
 }
}
```

# Zdarzenia
# Klient

```
HRESULT CCountdownEvents::EverySecondTick(int ElapsedTime)
{
  PlaySound("Tick",NULL,SND_RESOURCE|SND_ASYNC);
  cout << "Time to go: " << ElapsedTime << endl;
  return S_OK;
}
```

# Zdarzenia
## *ATL wizard* - uwagi

***ATL wizard*** **pozwala na automatyczną generację kodu dla interfejsów** `IConnectionPointContainer` **i** `IConnectionPoint`**. Należy jednak zwrócić uwagę na następujące problemy występujące z automatycznie generowanymi implementacjami:**

- **Wywoływanie metod klienta odbywa się poprzez funkcje** `Fire_NazwaMetody()`**. Wywołanie to odbywa się poprzez interfejs** `IDispatch`**, tak więc klient musi zaimplementować interfejs typu** `[source]` **wraz z interfejsem** `IDispatch`**.**
- **Generowany kod dla metod typu** `Fire_*()` **w przypadku parametrów typu** `[out]` **i** `[in,out]` **zawiera błąd. Wymaga to ręcznego poprawienia wygenerowanego kodu, lub modyfikacji w klasie CComVariant.**
- **Nie ma odpowiedniego kreatora do tworzenia klienta implementującego interfejs źródłowy.**

# Zdarzenia
## ATL wizard - błąd w metodzie Fire*()

```
HRESULT Fire_GetValue(FLOAT * Value) {
  CComVariant varResult;
  T* pT = static_cast<T*>(this);
  int nConIndex;
  CComVariant* pvars = new CComVariant[1];
  int nConnections = m_vec.GetSize();

  for (nConIdx = 0; nConIdx < nConnections; nConIdx++) {
    pT->Lock();
    CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
    pT->Unlock();
    IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);
    if (pDispatch != NULL) {
      VariantClear(&varResult);
      pvars[0] = Value; // Brak opratora dla wskaźników do typu TYPE*
      DISPPARAMS disp = { pvars, NULL, 1, 0 };
      pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD,
                        &disp, &varResult, NULL, NULL);
    }
  }
  delete[] pvars;
  return varResult.scode;
}
```

# Zdarzenia
## ATL wizard - błąd w metodzie Fire*()

**Rozwiązanie 1:**
**Modyfikacja kodu generowanej metody:**

```
HRESULT Fire_GetValue(FLOAT * Value) {
  ...
    if (pDispatch != NULL) {
      VariantClear(&varResult);
      pvars[0].vt=VT_BYREF|VT_R4; // Dla innych typów należy właściwie
                                  // zidentyfikować typ przez VT_???
      pvars[0].byref = Value;
      DISPPARAMS disp = { pvars, NULL, 1, 0 };
      pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT, DISPATCH_METHOD,
                        &disp, &varResult, NULL, NULL);
    }
  ...
}
```

# Zdarzenia
## ATL wizard - błąd w metodzie Fire*()

**Rozwiązanie 2:**
**Uzupełnienie funkcjonalności klasy CComVariant:**

```
class CComVariant : public tagVARIANT
{
        ...
        CComVariant(float* pfltSrc)
        {
                vt = VT_R4 | VT_BYREF;
                byref = pfltSrc;
        }

        CComVariant& operator=(float* pfltSrc)
        {
                if (vt != VT_R4 | VT_BYREF)
                {
                        InternalClear();
                        vt = VT_R4 | VT_BYREF;
                }
                byref = pfltSrc;
                return *this;
        }
        ...
};
```

# Zdarzenia
## ATL wizard - implementacja klienta

**Do zaimplementowania interfejsu typu `[source]` w klasie klienta można posłużyć się gotowymi wzorcami.**

1. **Za pomocą *ATL wizard* należy stworzyć nowy projekt typu *ATL COM.***
2. **Należy stworzyć nową klasę, która będzie odbierać wywołania od komponentu (np. `CClass`).**
3. **Należy odziedziczyć tę klasę od:**
   `public IDispEventImpl<1, CClass,&DIID__IEvents, &LIBID_COMPONENTLib,1,0>`**, gdzie:**
   `CClass` **- realizowana klasa klienta**
   `DIID__IEvents` **- IID implementowanego interfejsu** `[source]`
   `LIBID_COMPONENTLib` **- LIBID biblioteki komponentu**
   `1, 0` **- wersja i podwersja biblioteki komponentu**

# Zdarzenia
## ATL wizard - implementacja klienta

4. W części `public` klasy klienta należy umieścić blok deklarujący metody obsługujące wywołania od komponentu:

```
HRESULT __stdcall GetValue(float* value){
  *value=a;
  return S_OK;
};
BEGIN_SINK_MAP( CClass )
   SINK_ENTRY_EX(1, DIID__IEvents, 1, GetValue)
END_SINK_MAP()
```

Trzeci parametr deklaracji `SINK_ENTRY_EX` odpowiada zdefiniowanemu identyfikatorowi metody (atrybut `id()` metody).

# Zdarzenia
## ATL wizard - implementacja klienta

5. Aby klient mógł odbierać wywołania komponentu należy uttworzyć obiekt `CClass` i wywołać na jego rzecz metodę `DispEventAdvise()`:

```
IComponentPtr ComponentObj(CLSID_Component);
CClass ClientObj;

ClientObj.DispEventAdvise(ComponentObj);
float res=0.0;
res=a->MethodGeneratingGetValue();
ClientObj.DispEventUnadvise(ComponentObj);

ComponentObj=NULL;
```