

Projektowanie obiektowe

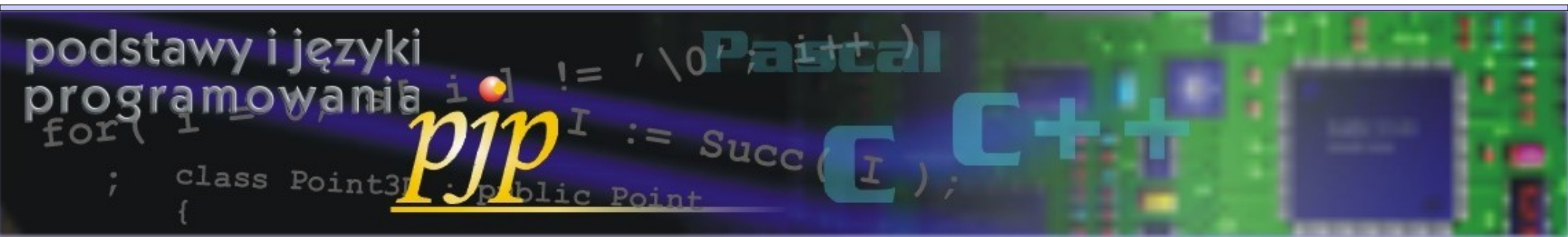
Roman Simiński

roman.siminski@us.edu.pl

www.siminskionline.pl

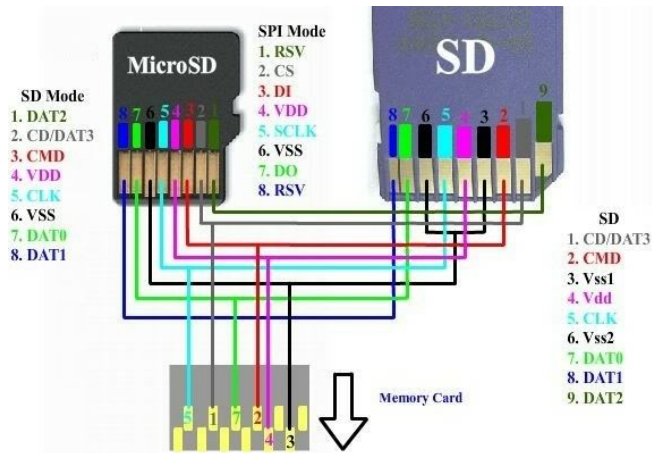
Wzorce projektowe

Wybrane wzorce strukturalne

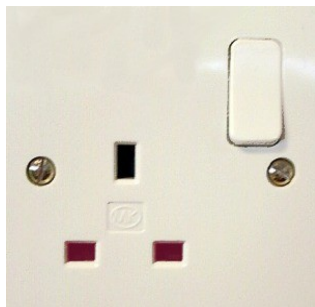


Wzorzec Adapter - Adapter Pattern - koncepcja

Problem



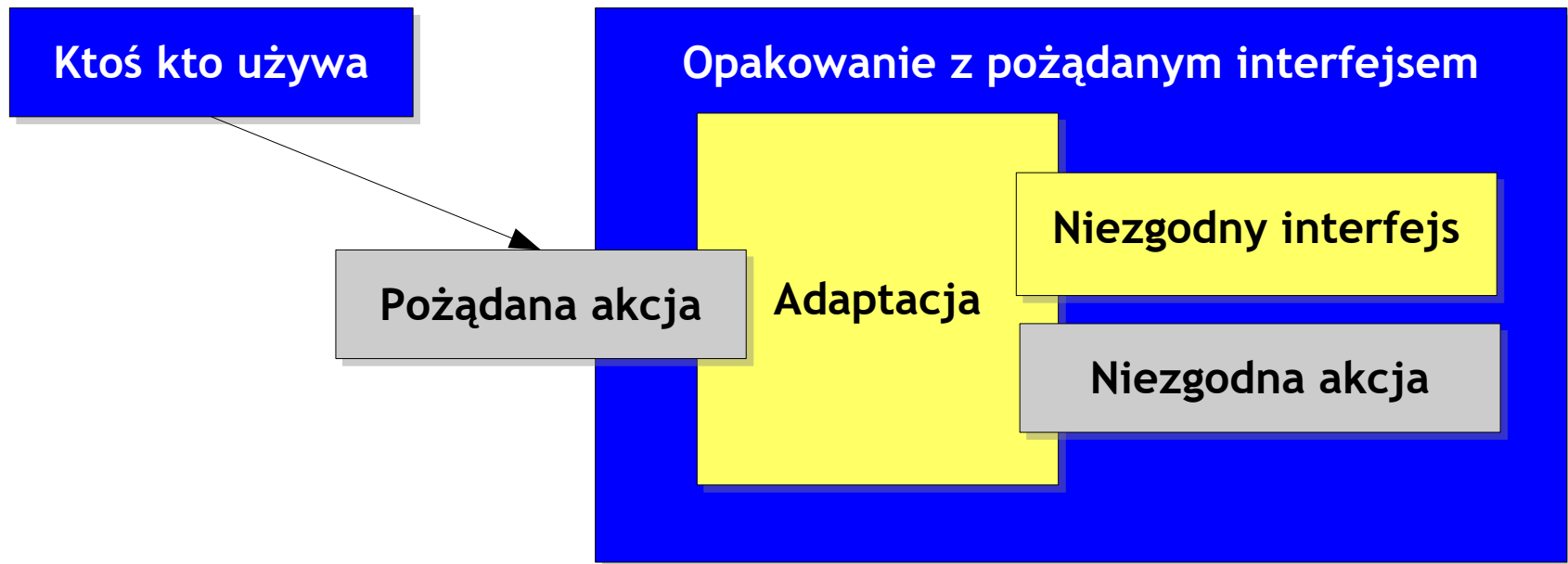
Rozwiązanie



Wzorzec Adapter

W programowaniu obiektowym wykorzystywane są:

- ▶ **Adapter** ma połączyć niezgodne ze sobą interfejsy.
- ▶ Osiąga się to poprzez *opakowanie* obiektu o niezgodnym interfejsie tak, aby opakowany obiekt realizował funkcje interfejsu pożądanego.
- ▶ *Adapter* bywa zatem zwany *opakowaniem* – *wrapper'em*, przy czym to pojęcie pojawia się też w innych kontekstach – np. przy wzorcu *fasada*.



Przykładowa implementacja adaptera w języku C++

Wzorzec Adapter

- ▶ Załóżmy, że istnieje klasa **APIWindow**, realizująca operacje na oknach GUI.
- ▶ Klasa ta pochodzi z bibliotek systemowych, nie mamy dostępu do kodu źródłowego.
- ▶ Chcemy, albo musimy wykorzystać obiekty tej klasy do wykonywania operacji okienkowych.
- ▶ Ale w naszym systemie zaplanowaliśmy zupełnie inny sposób obsługi okien, nie chcemy go zmieniać, w naszym programie służy do tego klasa **MyWindow**.

```
class MyWindow
{
    public:
        MyWindow() {}

        void show()
        {
            ...
        }
};
```

Zaplanowany interfejs obsługi okien
naszej aplikacji.

Ale on będzie musiał działać z wykorzystaniem
interfejsu klasy *APIWindow*.

Wzorzec Adapter

```
class APIWindow
{
public:
    APIWindow() {}

    void displayFrame()
    {
        cout << "\ndisplayFrame";
    }

    void displayInterior()
    {
        cout << "\ndisplayInterior";
    }
};
```

Istniejący, działający interfejs obsługi okien.

Ale on nam nie odpowiada.

- ▶ Ponieważ nie możemy zmodyfikować kodu klasy **APIWindow**, musimy odpowiednio zaprojektować własny kod.
- ▶ W rozważanym przypadku będzie to zaadaptowanie klasy **APIWindow** poprzez utworzenie *opakowania obiektu tej klasy*.

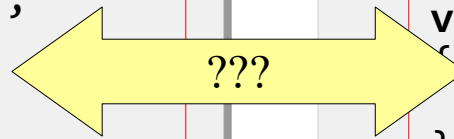
Wzorzec Adapter

- ▶ Interfejsy obu klas są niezgodne, należy zmodyfikować własny kod.

```
class APIWindow
{
public:
    APIWindow() {}

    void displayFrame()
    {
        cout << "\ndisplayFrame";
    }

    void displayInterior()
    {
        cout << "\ndisplayInterior";
    }
};
```



```
class MyWindow
{
public:
    MyWindow() {}

    void show()
    {
        ...
    }
};
```

Wzorzec Adapter, pierwszy krok adaptacji

- ▶ Ustalamy pożądany interfejs i przygotowujemy go do adaptacji.
- ▶ W C++ najlepiej klasę definiującą interfejs uczynić klasą abstrakcyjną.

```
class MyWindow
{
public:
    MyWindow() {}

    virtual ~MyWindow() {}

    virtual void show() = 0;
};
```

W języku C++ klasy abstrakcyjne realizujące rolę interfejsów powinny mieć zdefiniowany destruktork wirtualny

Określamy pożądany interfejs obsługi okien.

Wzorzec Adapter, drugi krok adaptacji

Tworzymy klasę Adaptera, która:

- ▶ będzie dziedziczyć po klasie definiującej pożądany interfejs,
- ▶ będzie implementować metody interfejsu, które będą kierować odpowiednie wywołania do obiektu klasy adaptowanej **APIWindow**, który z kolei będzie polem prywatnym klasy adaptera.

```
class WindowAdapter : public MyWindow
{
public:
    WindowAdapter() : MyWindow() {}

    void show()
    {
        apiWin.displayFrame();
        apiWin.displayInterior();
    }

private:
    APIWindow apiWin;
};
```

Adapter dziedziczy po klasie docelowej

Adapter implementuje interfejs klasy docelowej

Adapter posługuje się obiektem klasy adaptowanej

Wzorzec Adapter, trzeci krok adaptacji

Wykorzystujemy obiekt klasy Adaptera wszędzie tam, gdzie chcemy wykorzystać obiekt docelowej obsługi okna:

```
int main()
{
    MyWindow * w = new WindowAdapter();
    w->show();
    delete w;
    return EXIT_SUCCESS;
}
```

W sposób przeźroczysty
wykorzystujemy zaadaptowany
obiekt o niezgodnym interfejsie

Programujemy wykorzystując
požadany interfejs

```
displayFrame
displayInterior_
```

Wzorzec Adapter, *adapter obiektowy*

Przedstawiony przykład prezentuje ***adapter obiektów*** – pożądany interfejs uzyskuje się poprzez wykorzystanie instancji obiektu adaptowanej klasy/interfejsu.

```
class WindowAdapter : public MyWindow
{
public:
    WindowAdapter() : MyWindow() {}

    void show()
    {
        apiWin.displayFrame();
        apiWin.displayInterior();
    }

private:
    APIWindow apiWin;
};
```

Adapter dziedziczy po klasie docelowej

Adapter implementuje interfejs klasy docelowej wykorzystując odmienny interfejs obiektu adaptowanego

Adapter posługuje się obiektem klasy adaptowanej

Wzorzec Adapter, *adapter obiektowy*, dynamiczny

Adaptowany obiekt może być tworzony dynamicznie.

```
class WindowAdapter : public MyWindow
{
public:
    WindowAdapter() : MyWindow() { apiWin = new APIWindow(); }

    void show()
    {
        apiWin->displayFrame();
        apiWin->displayInterior();
    }

    ~WindowAdapter()
    {
        delete apiWin;
    }

private:
    APIWindow *apiWin;
};
```

The diagram illustrates the dynamic creation and management of an `APIWindow` object within the `WindowAdapter` class. A red box highlights the line `apiWin = new APIWindow();` in the constructor, with an arrow pointing to the text 'Adapter dziedziczy po klasie docelowej'. Another red box highlights the `delete apiWin;` line in the destructor, with an arrow pointing to the text 'Dlatego wirtualny destruktor w klasie bazowej był konieczny'. A third red box highlights the `APIWindow *apiWin;` line in the private section, with an arrow pointing to the text 'Adapter posługuje się obiektem klasy adaptowanej tworzonym dynamicznie'.

Adapter dziedziczy po klasie docelowej

Dlatego wirtualny destruktor w klasie bazowej był konieczny

Adapter posługuje się obiektem klasy adaptowanej tworzonym dynamicznie

Warianty dla *adaptera obiektowego*

Adapter obiektu pozwala na wykorzystanie obiektów klas potomnych.

```
class APIWindow
{
public:
    APIWindow() {}

    virtual void displayFrame() { cout << "\ndisplayFrame"; }
    virtual void displayInterior() { cout << "\ndisplayInterior"; }
};

class APIDesktopWindow : public APIWindow
{
public:
    APIDesktopWindow() : APIWindow() {}

    void displayFrame() { cout << "\nAPIDesktopWindow::displayFrame"; }
    void displayInterior() { cout << "\nAPIDesktopWindow::displayInterior"; }
};

class APIMobileWindow : public APIWindow
{
public:
    APIMobileWindow() : APIWindow() {}

    void displayFrame() { cout << "\nAPIMobileWindow::displayFrame"; }
    void displayInterior() { cout << "\nAPIMobileWindow::displayInterior"; }
};
```

Warianty dla *adaptera obiektowego*

Adapter dobiera odpowiedni obiekt do realizacji wymaganych operacji.

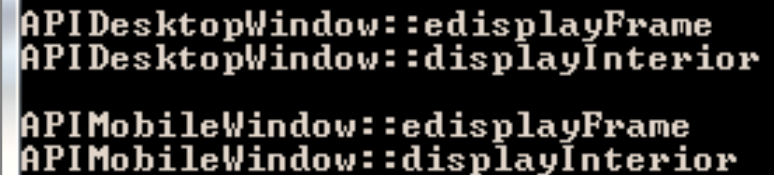
```
class WindowAdapter : public MyWindow
{
public:
    enum WinVer { DESKTOP, MOBILE };

    WindowAdapter( WinVer v ) : MyWindow(), ver( v ) {
        switch( ver )
        {
            case DESKTOP : apiWin = new APIDesktopWindow(); break;
            case MOBILE   : apiWin = new APIMobileWindow(); break;
        }
    }

    void show() {
        apiWin->displayFrame();
        apiWin->displayInterior();
    }

    ~WindowAdapter() {
        delete apiWin;
    }

private:
    APIWindow *apiWin;
    WinVer     ver;
};
```



```
APIDesktopWindow::displayFrame
APIDesktopWindow::displayInterior

APIMobileWindow::displayFrame
APIMobileWindow::displayInterior
```

```
MyWindow * w;
```

```
w = new WindowAdapter( WindowAdapter::DESKTOP );
w->show();
delete w;
...
w = new WindowAdapter( WindowAdapter::MOBILE );
w->show();
delete w;
```

Warianty dla *adaptera obiektowego*

Adapter może być pasywny – nie decyduje o typie obiektu adaptowanego.

```
class WindowAdapter : public MyWindow
{
public:
    WindowAdapter( APIWindow *win )
        : MyWindow(), apiWin( win )
        {}

    void show()
    {
        apiWin->displayFrame();
        apiWin->displayInterior();
    }

    ~WindowAdapter()
    {
        delete apiWin;
    }

private:
    APIWindow *apiWin;
};
```

```
MyWindow * w;
```

```
w = new WindowAdapter( new APIDesktopWindow() );
w->show();
delete w;
...
w = new WindowAdapter( new APIMobileWindow() );
w->show();
delete w;
```

```
APIDesktopWindow::displayFrame
APIDesktopWindow::displayInterior

APIMobileWindow::displayFrame
APIMobileWindow::displayInterior
```

Ogólny schemat wykorzystania *adaptera obiektowego*

```
class Docelowy
{
public:
    Docelowy() {}
    virtual void docelowaMetoda() = 0;
    virtual ~Docelowy() {}
};

class Adaptowany
{
public:
    Adaptowany() {}
    void adaptowanaMetoda() { cout << "adaptowanaMetoda"; }
};

class AdapterObiektowy : public Docelowy
{
public:
    AdapterObiektowy() : Docelowy() { obiekt = new Adaptowany(); }
    ~AdapterObiektowy() { delete obiekt; }

    void docelowaMetoda()
    {
        obiekt->adaptowanaMetoda();
    }
private:
    Adaptowany * obiekt;
};
```

```
Docelowy * d = new AdapterObiektowy();
d->docelowaMetoda();
delete d;
```


Ogólny schemat wykorzystania *adaptera klasowego*

```
class Docelowy
{
public:
    Docelowy() {}
    virtual void docelowaMetoda() = 0;
    virtual ~Docelowy() {}
};

class Adaptowany
{
public:
    Adaptowany() {}
    void adaptowanaMetoda() { cout << "adaptowanaMetoda"; }
};

class AdapterKlasowy : public Docelowy, private Adaptowany
{
public:
    AdapterKlasowy() : Docelowy(), Adaptowany() { }

    void docelowaMetoda()
    {
        adaptowanaMetoda();
    }
};
```

```
Docelowy * d = new AdapterKlasowy();

d->docelowaMetoda();

delete d;
```

Adapter obiektowy vs adapter klasowy

```
class AdapterObiektowy : public Docelowy
{
public:
    AdapterObiektowy() : Docelowy() { obiekt = new Adaptowany(); }

    ~AdapterObiektowy() { delete obiekt; }

    void docelowaMetoda()
    {
        obiekt->adaptowanaMetoda();
    }
private:
    Adaptowany * obiekt;
};
```

```
class AdapterKlasowy : public Docelowy, private Adaptowany
{
public:
    AdapterKlasowy() : Docelowy(), Adaptowany() { }

    void docelowaMetoda()
    {
        adaptowanaMetoda();
    }
};
```

Adapter obiektowy vs adapter klasowy, wady i zalety

Adapter obiektowy:

- ▶ używa kompozycji obiektów, która daje możliwość adaptacji klasy oraz jej podklas (wykorzystanie polimorfizmu);
- ▶ brak możliwości przeciążenia metod obiektu adaptowanego.

Adapter klasowy:

- ▶ Używa dziedziczenia, nadpisując metody klasy docelowej, dokonując „translacji” zachowania klasy adaptowanej.
- ▶ Utrudnione adaptowanie podklas klasy adaptowanej.
- ▶ Mogą pojawić się typowe problemy związane z dziedziczeniem wielobazowym.

Wykorzystanie adaptera – podsumowanie

- ▶ Adapter jest użyteczny gdy istniejące, potrzebne nam biblioteki *nie mogą* być używane z powodu *niezgodności* z interfejsem wymaganym przez aplikację.
- ▶ Opracowujemy adapter gdy *nie możemy* lub *nie chcemy* zmienić interfejsu biblioteki, nie posiadamy jej kodu źródłowego.
- ▶ Adapter wspiera wielokrotne użycie kodu i przenaszalność kodu, pozwalając na wykorzystanie klas zupełnie niepowiązanych z realizowaną aplikacją a potencjalnie powiązanych z daną platformą systemową czy sprzętową.

Przykładowa implementacja adaptera w języku Java

Adapter obiektowy z podklasami

Docelowy interfejs wg wymagań aplikacji:

```
interface MyWindow {  
    void show();  
}
```

Istniejąca klasa do zaadaptowania:

```
class APIWindow  
{  
    public APIWindow() {}  
  
    public void displayFrame() {  
        System.out.println( "APIWindow::displayFrame" );  
    }  
  
    public void displayInterior() {  
        System.out.println( "APIWindow::displayInterior" );  
    }  
}
```

Adapter obiektowy z podklasami - podklasy adaptowane

```
class APIDesktopWindow extends APIWindow
{
    public APIDesktopWindow() { super(); }
    @Override
    public void displayFrame() {
        System.out.println( "APIDesktopWindow::displayFrame" );
    }
    @Override
    public void displayInterior() {
        System.out.println( "APIDesktopWindow::displayInterior" );
    }
}
```

```
class APIMobileWindow extends APIWindow
{
    public APIMobileWindow() { super(); }
    @Override
    public void displayFrame() {
        System.out.println( "APIMobileWindow::displayFrame" );
    }
    @Override
    public void displayInterior() {
        System.out.println( "APIMobileWindow::displayInterior" );
    }
}
```

Adapter obiektowy z podklasami

```
class WindowAdapter implements MyWindow
{
    public enum Version { DESKTOP, MOBILE }

    WindowAdapter( Version v ) {
        super();
        ver = v;

        switch( ver )
        {
            case DESKTOP : apiWin = new APIDesktopWindow();
                           break;
            case MOBILE : apiWin = new APIMobileWindow();
                           break;
        }
    }

    @Override
    public void show() {
        apiWin.displayFrame();
        apiWin.displayInterior();
    }

    private APIWindow apiWin;
    private Version ver;
}
```

```
MyWindow w;

w = new WindowAdapter( WindowAdapter.Version.DESKTOP );
w.show();

w = new WindowAdapter( WindowAdapter.Version.MOBILE );
w.show();
```


Adapter obiektowy - ogólny schemat

```
interface Docelowy {  
    void docelowaMetoda();  
}  
  
class Adaptowany  
{  
    public Adaptowany() {}  
  
    public void adaptowanaMetoda() {  
        System.out.println( "adaptowanaMetoda" );  
    }  
}  
  
class AdapterObiektowy implements Docelowy  
{  
    public AdapterObiektowy() {  
        super();  
        obiekt = new Adaptowany();  
    }  
  
    public void docelowaMetoda() {  
        obiekt.adaptowanaMetoda();  
    }  
  
    private Adaptowany obiekt;  
}
```

```
Docelowy d = new AdapterObiektowy();  
d.docelowaMetoda();
```

Adapter klasowy - ogólny schemat

```
interface Docelowy {  
    void docelowaMetoda();  
}  
  
class Adaptowany  
{  
    public Adaptowany() {}  
  
    public void adaptowanaMetoda() {  
        System.out.println( "adaptowanaMetoda" );  
    }  
}  
  
class AdapterKlasowy extends Adaptowany implements Docelowy  
{  
    public AdapterKlasowy() { super(); }  
  
    @Override  
    public void docelowaMetoda() {  
        adaptowanaMetoda();  
    }  
}
```

```
Docelowy d = new AdapterKlasowy();  
d.docelowaMetoda();
```