

# Projektowanie i programowanie obiektowe

**Roman Simiński**

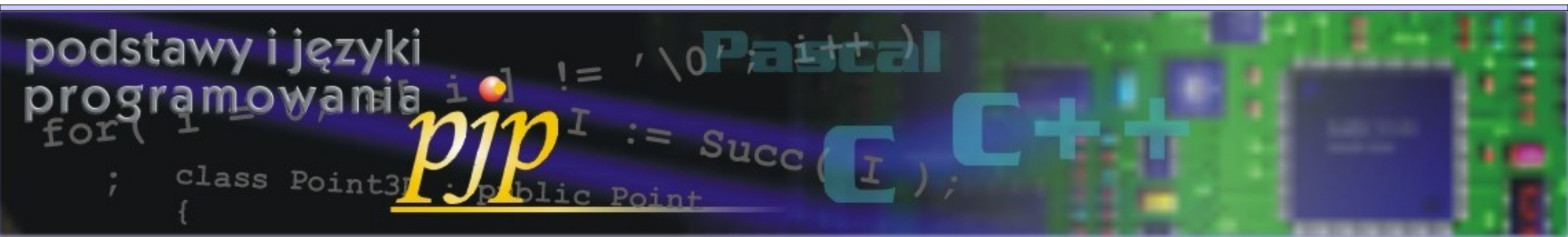
roman.siminski@us.edu.pl

roman@siminskionline.pl

programowanie.siminskionline.pl

## Wzorce projektowe

Wybrane wzorce kreacyjne: Fabryka

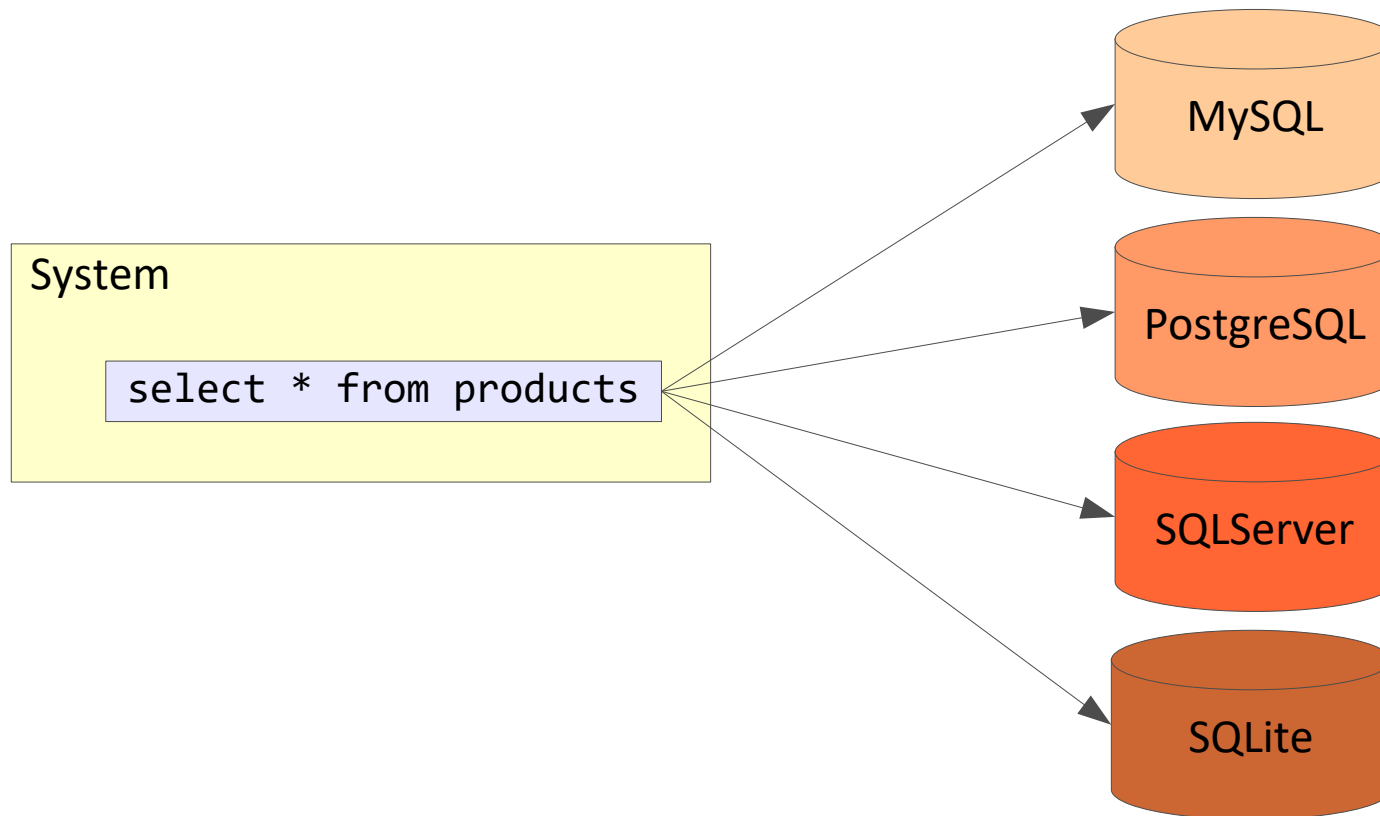


# Fabryka prosta

## Simple Factory

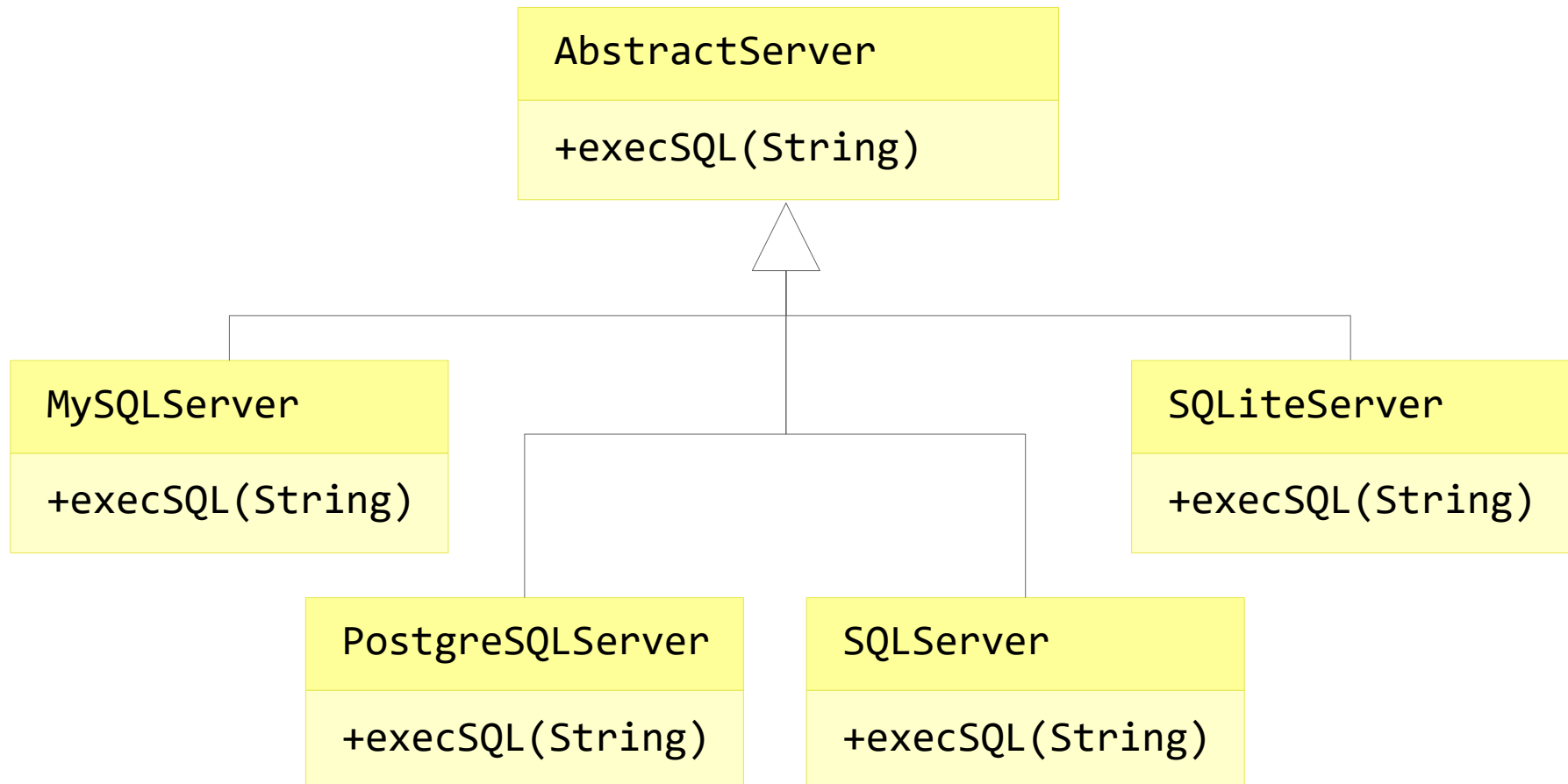
# Przykład

- ▶ Załóżmy, że piszemy program wykorzystujący relacyjną bazę danych.
- ▶ W programie posługiwać się będziemy standardowymi poleceniami SQL, obsługiwanymi przez większość serwerów baz danych.
- ▶ Chcemy aby nasz program współpracował z większością popularnych baz danych, w danym momencie z jedną.



# Przykład

- ▶ Będziemy używać obiektów zapewniających dostęp do każdej z baz. Zatem dla każdej bazy zdefiniujemy odpowiednią klasę.
- ▶ Klasy te będą dziedziczyć z klasy abstrakcyjnej.



# Przykład

```
...  
AbstractServer base;  
  
// Ustalenie bazy z którą współpracujemy, np. poprzez odczyt informacji  
// z pliku konfiguracyjnego, utworzenie obiektu odpowiedniej bazy  
...  
base.execSQL( "select * from produkty" );  
...  
...
```

- ▶ Dzięki klasom abstrakcyjnym i/lub interfejsom możemy pisać elastyczny kod, który wykorzystuje polimorfizm.
- ▶ Taki kod może działać na dowolnym obiekcie z ustalonej hierarchii klas wykorzystującej dziedziczenie.
- ▶ Jak oprogramować identyfikację konkretnej bazy i utworzenie odpowiedniego obiektu?

# Przykład

```
...
AbstractServer base;

String dbType = config.getDataBaseType();

switch( dbType )
{
    case "mysql"      : base = new MySQLServer();
                       break;
    case "postgresql": base = new PostgreSQLServer();
                       break;
    case "sqlite"     : base = new SQLiteServer();
                       break;
    case "sqlserver"  : base = new SQLServer();
                       break;
}

...
base.execSQL( "select * from produkty" );
...
```

# Przykład

```
...  
AbstractServer base;  
  
String dbType = config.getDataBaseType();  
  
switch( dbType )  
{  
    case "mysql"      : base = new MySQLServer();  
                       break;  
    case "postgresql": base = new PostgreSQLServer();  
                       break;  
    case "sqlite"     : base = new SQLiteServer();  
                       break;  
    case "sqlserver"  : base = new SQLServer();  
                       break;  
}
```

```
...  
base.execSQL( "select * from ...");  
...  
...
```

Tutaj tracimy elastyczność, każda modyfikacja hierarchii klas powoduje konieczność modyfikacji tego kodu.

Kod staje się zależny od klas podrzędnych, konkretnych, niższego poziomu.

# Przykład

```
...  
AbstractServer base;  
  
String dbType = config.getDataBaseType();  
  
switch( dbType )  
{  
    case "mysgl"      : base = new MySQLServer();  
                      break;  
    case "p"          : ...  
    case "s"          : ...  
    case "s"          : ...  
}  
  
...  
base.execSQL( "select * from produkty" );  
...  
...
```

To rozwiązanie łamie zasady **SOLID**:

Zasadę **Open/Close**

Zasadę **Dependency inversion**



# Przykład

```
...  
AbstractServer base;  
  
String dbType = config.getDataBaseType();  
  
switch( dbType )  
{  
    case "mysgl"      : base = new MySQLServer();  
                      break;  
    ca  
    ca  
    ca  
}  
  
...  
base.execSQL( "select * from produkty" );  
...  

```

Jak zamknąć ten fragment kodu na modyfikację otwierając jednocześnie na rozszerzenia?

Jak uczynić ten fragment kodu niezależnym od klas podrzędnych — jak odwrócić zależność?

# Wprowadzamy klasę konstruuującą odpowiedni obiekt

```
class SQLServerFactory
{
    public AbstractServer createServer( ConfigFile config )
    {
        String dbType = config.getDataBaseType();

        switch( dbType )
        {
            case "mysgl"      : return new MySQLServer();
            case "postgresgl": return new PostgreSQLServer();
            case "sqlite"     : return new SQLiteServer();
            case "sqlserver"  : return new SQLServer();
        }
        return null;
    }
}
```

```
SQLServerFactory serverFactory = new SQLServerFactory();
```

```
AbstractServer base = serverFactory.createServer( config );
base.execSQL( "select * from produkty" );
```

# Przeniesienie konstruowania obiektu do prostej fabryki

- ▶ Prosta fabryka *przejmuje obowiązek utworzenia obiektu* odpowiedniej klasy.
- ▶ Konkretnie utworzenie obiektu realizuje wyodrębniona *funkcja kreująca*. Jej rezultatem jest utworzony obiekt.
- ▶ Aby to wszystko działało, tworzone obiekty powinny być egzemplarzami klas pochodnych pewnej *klasy abstrakcyjnej* lub powinny *implementować wspólny interfejs*.
- ▶ Fabryka otrzymuje zwykle jakąś informację o kontekście działania programu – jest to najczęściej parametr funkcji kreującej, na podstawie którego ustalana jest odpowiednia wersja tworzonego obiektu.

Sprawa wydaje się zagniatwana – aby utworzyć odpowiedni obiekt trzeba utworzyć obiekt prostej fabryki tylko po to, aby wywołać jego funkcję składową.

Uprościć sprawę może *fabryka statyczna*.

# Statyczna klasa fabryki

```
static class SQLServerFactory
{
    public static AbstractServer createServer( ConfigFile config )
    {
        String dbType = config.getDataBaseType();

        switch( dbType )
        {
            case "mysgl"      : return new MySQLServer();
            case "postgresgl": return new PostgreSQLServer();
            case "sqlite"     : return new SQLiteServer();
            case "sqlserver"  : return new SQLServer();
        }
        return null;
    }
}

. . .
AbstractServer base = SQLServerFactory.createServer( config );
base.execSQL( "select * from produkty" );
. . .
```

# Przykładowa implementacja prostej fabryki w języku Java

# Klasa pomocnicza i klasa bazowa dla kreowanych obiektów

```
class ConfigFile
{
    public String getDataBaseType()
    {
        // Funkcja „zaśleпка”, tu powinno być odczytanie z pliku
        // konfiguracyjnego informacji o aktualnym typie serwera
        return "sqlite";
    }
}

abstract class AbstractServer
{
    public abstract void execSQL( String query );
}
```

# Klasy dla obiektów konkretnych

```
class MySQLServer extends AbstractServer
{
    @Override
    public void execSQL( String query )
    {
        System.out.println( "MySQLServer: " + query );
    }
}

class PostgreSQLServer extends AbstractServer
{
    @Override
    public void execSQL( String query )
    {
        System.out.println( "PostgreSQLServer: " + query );
    }
}
```

# Klasy dla obiektów konkretnych

```
class SQLiteServer extends AbstractServer
{
    @Override
    public void execSQL( String query )
    {
        System.out.println( "SQLiteServer: " + query );
    }
}

class SQLServer extends AbstractServer
{
    @Override
    public void execSQL( String query )
    {
        System.out.println( "SQLServer: " + query );
    }
}
```



# Klasa prostej fabryki

```
class SQLServerFactory
{
    public AbstractServer createServer( ConfigFile config )
    {
        String dbType = config.getDataBaseType();

        switch( dbType )
        {
            case "mysql"       : return new MySQLServer();
            case "postgresql"  : return new PostgreSQLServer();
            case "sqlite"      : return new SQLiteServer();
            case "sqlserver"   : return new SQLServer();
        }
        return null;
    }
}
```

# Klasa klienta omawianego wzorca i jej wykorzystanie

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer()
    {
        SQLServerFactory factory = new SQLServerFactory();

        AbstractServer base = factory.createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

```
public class FabrykaDemo
{
    public static void main(String[] args)
    {
        SQLFactoryClient client = new SQLFactoryClient();
        client.createAndUseServer();
    }
}
```

# Fabryka statyczna raz jeszcze

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer()
    {
        SQLServerFactory factory = new SQLServerFactory();
        AbstractServer base = factory.createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

W tym przypadku można zastosować fabrykę statyczną

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer()
    {
        AbstractServer base = StaticSQLServerFactory.createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

# Wady prezentowanego podejścia

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer()
    {
        SQLServerFactory factory = new SQLServerFactory();
        AbstractServer base = factory.createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

W obu przypadkach kod nie jest zgodny z zasadą Open/Close

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer()
    {
        AbstractServer base = StaticSQLServerFactory.createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

## Fabryka może być przekazywana na zasadzie DI

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer( SQLServerFactory factory )
    {
        AbstractServer base = factory.createServer( cnfFile );

        base.execSQL( "select * from produkty" );
    }
}
```

Utworzenie obiektu klienta

```
public class FabrykaDemo01
{
    public static void main(String[] args)
    {
        SQLFactoryClient client = new SQLFactoryClient();
        client.createAndUseServer( new SQLServerFactory() );
    }
}
```

## Fabryka może być przekazywana na zasadzie DI

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void createAndUseServer( SQLServerFactory factory )
    {
        AbstractServer base = factory.createServer( cnfFile );

        base.execSQL( "select * from produkty" );
    }
}
```

Utworzenie fabryki i przekazanie do metody obiektu klienta

```
public class FabrykaDemo01
{
    public static void main(String[] args)
    {
        SQLFactoryClient client = new SQLFactoryClient();
        client.createAndUseServer( new SQLServerFactory() );
    }
}
```

# Fabryka może być przekazywana na zasadzie DI

```
class SQLFactoryClient
{
    ConfigFile cnfFile = new ConfigFile();

    public void factoryExample( SQLServerFactory factory )
    {
        AbstractServer base = factory.createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

Metod klienta wykorzystuje  
wstrzyknięty obiekt fabryki

```
public class FabrykaDemo
{
    public static void main(String[] args)
    {
        SQLFactoryClient c = new SQLFactoryClient();
        c.factoryExample( new SQLServerFactory() );
    }
}
```

Wstrzyknięcie obiektu fabryki

# Wstrzykiwać można specjalizowane fabryki

```
class LocalSQLServerFactory extends SQLServerFactory {
    public AbstractServer createServer( ConfigFile config ) {
        // Utworzenie serwera w konfiguracji lokalnej
    }
}

class RemoteSQLServerFactory extends SQLServerFactory {
    public AbstractServer createServer( ConfigFile config ) {
        // Utworzenie serwera w konfiguracji zdalnej
    }
}
```

```
public class FabrykaDemo
{
    public static void main(String[] args)
    {
        SQLFactoryClient client = new SQLFactoryClient();
        if( localServerRequired() )
            client.createAndUseServer( new LocalSQLServerFactory() );
        else
            client.createAndUseServer( new RemoteSQLServerFactory() );
    }
}
```

Wstrzyknięcie różnych fabryk



# Metod fabrykująca Factory method

## „Bezobektowe” podejście do tworzenia obiektów

- ▶ Zastosowanie prostej fabryki wymaga utworzenie osobnej klasy dla fabryki i wykorzystania zawartej w niej metody tworzącej konkretne obiekty.
- ▶ Wykorzystanie fabryki statycznej pozwala na uproszczenie operacji (nie trzeba tworzyć obiektu) kosztem utraty elastyczności.
- ▶ Wzorzec *metoda fabrykująca* pozwala na rezygnację z klasy fabryki.
- ▶ Odbywa się to poprzez umieszczenie metody tworzącej obiekty w klasie klienta wzorca.

# Metoda fabrykująca, przykład

```
class SQLServerClient
{
    ConfigFile cnfFile = new ConfigFile();

    public AbstractServer createServer( ConfigFile config )
    {
        String dbType = config.getDataBaseType();
        switch( dbType )
        {
            case "mysql"       : return new MySQLServer();
            case "postgresql"  : return new PostgreSQLServer();
            case "sqlite"      : return new SQLiteServer();
            case "sqlserver"   : return new SQLServer();
        }
        return null;
    }

    public void createAndUseServer()
    {
        // Utworzenie obiektu serwera z wykorzystaniem metody fabrykującej.
        AbstractServer base = createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

# Metoda fabrykująca, przykład

```
class SQLServerClient
{
    ConfigFile cnfFile = new ConfigFile();

    public AbstractServer createServer( ConfigFile config )
    {
        String dbType = config.getDataBaseType();
        switch( dbType )
        {
            case "mysql"      : return new MySQLServer();
            case "postgresql" : return new PostgreSQLServer();
            case "sqlite"     : return new SQLiteServer();
            case "sqlserver"  : return new SQLServer();
        }
        return null;
    }

    public void createAndUseServer()
    {
        // Utworzenie obiektu serwera z wykorzystaniem metody fabrykującej.
        AbstractServer base = createServer( cnfFile );
        base.execSQL( "select * from produktv" );
    }
}
```

Ponownie kod nie jest zgodny  
z zasadą Open/Close

# Abstrakcyjna metoda fabrykująca, przykład

```
abstract class AbstractSQLServerClient
{
    ConfigFile cnfFile = new ConfigFile();

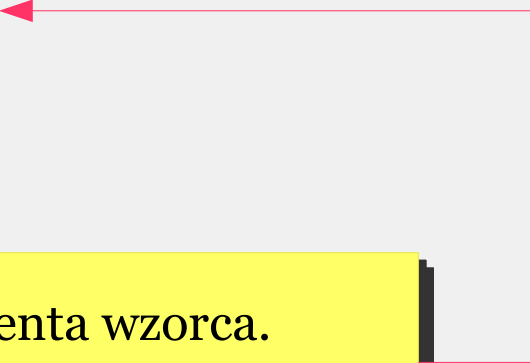
    public abstract AbstractServer createServer( ConfigFile config );

    public void createAndUseServer()
    {
        // Utworzenie obiektu serwera z wykorzystaniem metody fabrykującej.
        AbstractServer base = createServer( cnfFile );
        base.execSQL( "select * from produkty" );
    }
}
```

# Abstrakcyjna metoda fabrykująca, przykład

```
class SQLServerClient extends AbstractSQLServerClient
{
    // Konkretna implementacja metody fabrykującej
    @Override
    public AbstractServer createServer( ConfigFile config )
    {
        String dbType = config.getDataBaseType();
        switch( dbType )
        {
            case "mysql"      : return new MySQLServer();
            case "postgresql" : return new PostgreSQLServer();
            case "sqlite"     : return new SQLiteServer();
            case "sqlserver"  : return new SQLServer();
        }
        return null;
    }
}
```

```
AbstractSQLServerClient client = new SQLServerClient();
client.createAndUseServer();
```



Utworzenie i wykorzystanie obiektu klienta wzorca.  
Ważne: pracujemy na referencji do klasy abstrakcyjnej.