Wzorce oprogramowania Gof

(Gang of Four – skrót odnoszący się do autorów książki: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*)

zastosowane w modelu obiektowym

Identyfikacja wzorców projektowych

- Dobrze zbudowany system obiektowy jest pełen wzorców obiektowych
- Wzorzec to zwyczajowo przyjęte rozwiązanie typowego problemu w danym kontekście
- Strukturę wzorca przedstawia się w postaci diagramu klas
- Zachowanie się wzorca przedstawia się za pomocą diagramu sekwencji
- Wzorce projektowe: Wzorzec reprezentuje powiązanie problemu z rozwiązaniem
  (wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Każdy wzorzec składa się z trzech części, które wyrażają związek między konkretnym kontekstem, problemem i rozwiązaniem **(Christopher Aleksander)**
- Każdy wzorzec to trzyczęściowa reguła, która wyraża związek między konkretnym kontekstem, rozkładem sił powtarzającym się w tym kontekście i konfiguracją oprogramowania pozwalająca na wzajemne zrównoważenie się tych sił w celu rozwiązania zadania. **(Richar Gabriel)**
- Wzorzec to pomysł, który okazał się użyteczny w jednym rzeczywistym kontekście i prawdopodobnie będzie użyteczny w innym. **(Martin Fowler)**

2. Przegląd wzorców projektowych

1. Wzorce kreacyjne
2. Wzorce strukturalne
3. Wzorce czynnościowe

2.1. Wzorce kreacyjne

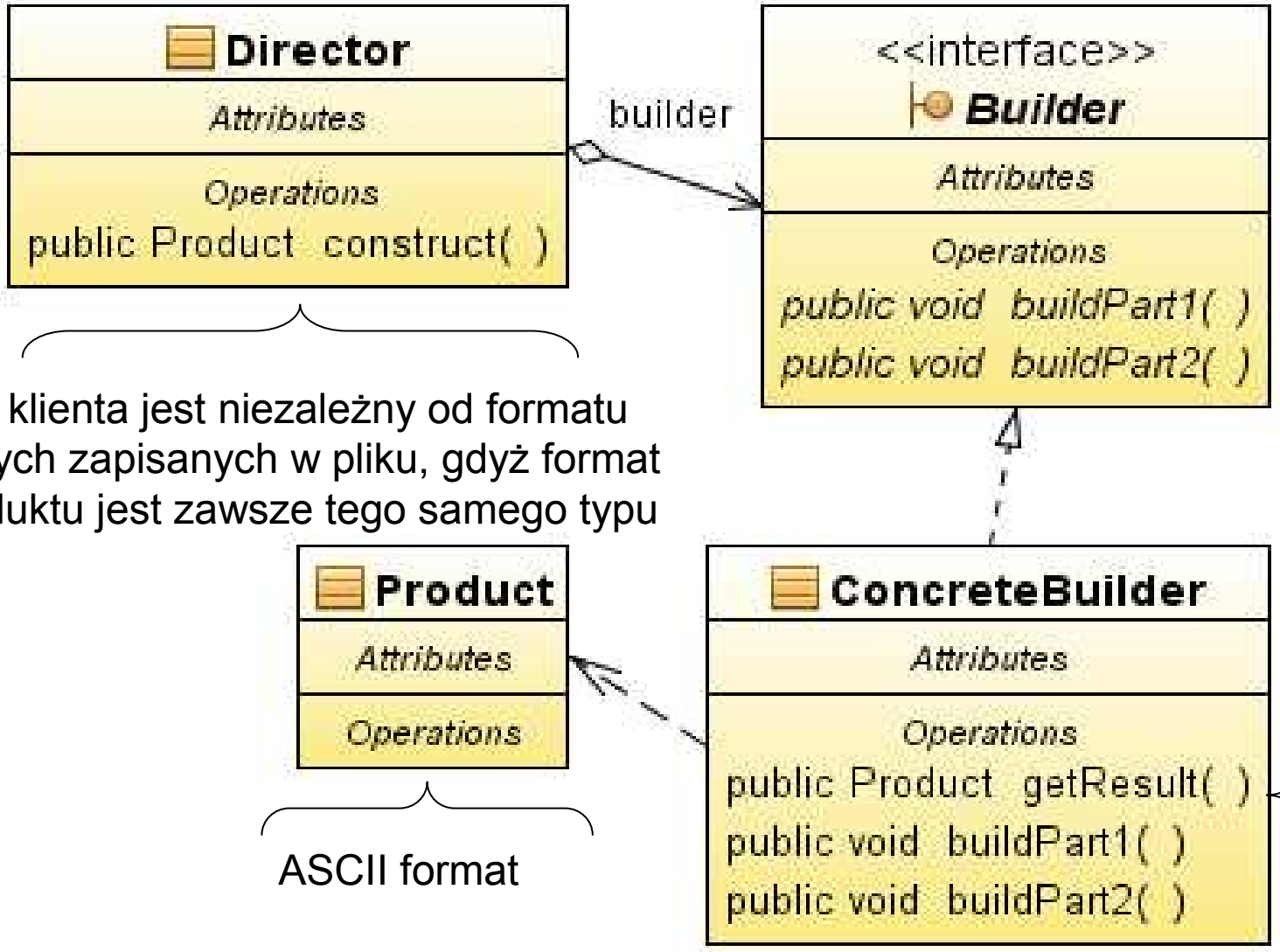
Izolacja reguł tworzenia obiektów od reguł określających sposób używania obiektów (oddzielenie w kodzie programu kodu tworzącego obiekty od kodu, który używa obiekty) – klasy typu „Control”

1) Budowniczy - *Builder*
2) Fabryka abstrakcyjna – *Abstract Factory*
3) Metoda wytwórcza – *Factory Method*
4) Prototyp - *Prototype*
5) Singleton

Wybór wzorca kreacyjnego

Wzorzec projektowy	Aspekt, który może się zmienić
1)Abstract Factory	Rodziny obiektów
2)Builder	Sposób tworzenia złożonych obiektów
3)Factory Method	Podklasa tworzonego obiektu
4)Prototype	Typ klasy tworzonego obiektu
5)Singleton	Jedna kopia obiektu

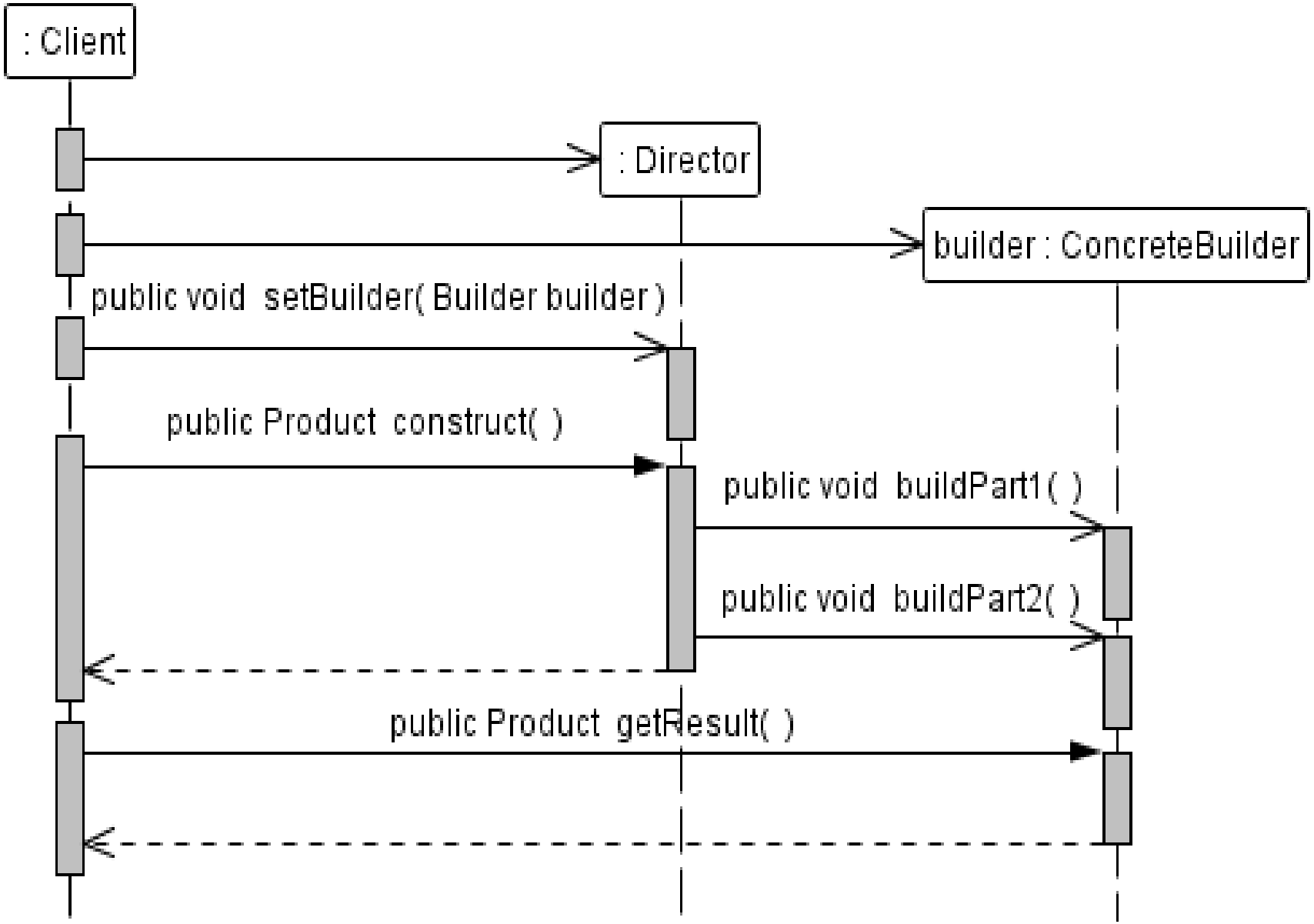
1) Budowniczy - *Builder*



Kod klienta jest niezależny od formatu danych zapisanych w pliku, gdyż format produktu jest zawsze tego samego typu

ASCII format

ASCII konwerter z formatów RTF lub XML


: Client
: Director
builder : ConcreteBuilder
public void setBuilder(Builder builder)
public Product construct( )
public void buildPart1( )
public void buildPart2( )
public Product getResult( )

Charakterystyka wzorca Budowniczy

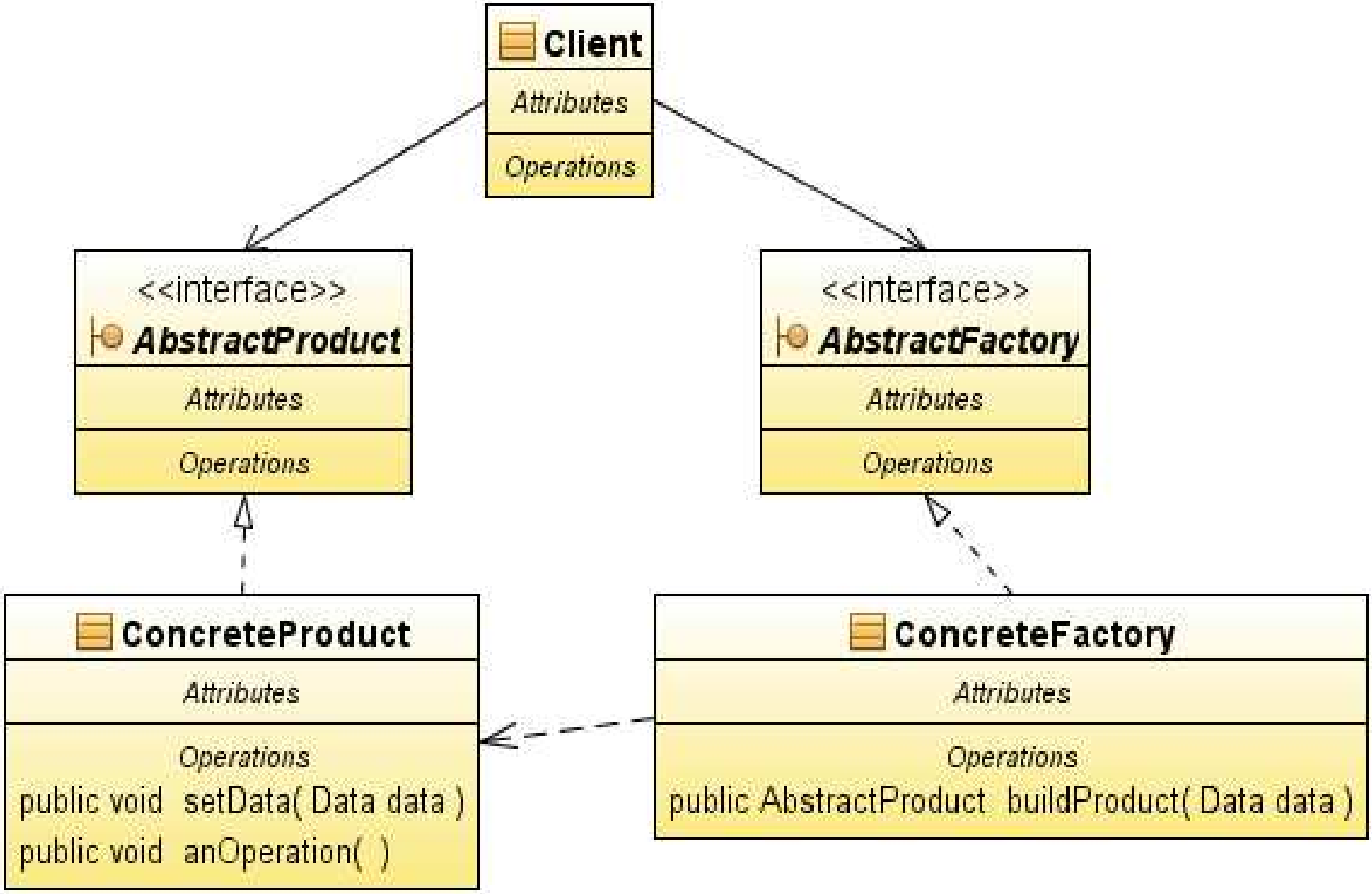
- **Problem**: Tworzenie dowolnych obiektów złożonych reprezentowanych w zróżnicowany sposób oraz oddzielenie konstrukcji obiektów złożonych od ich reprezentacji.

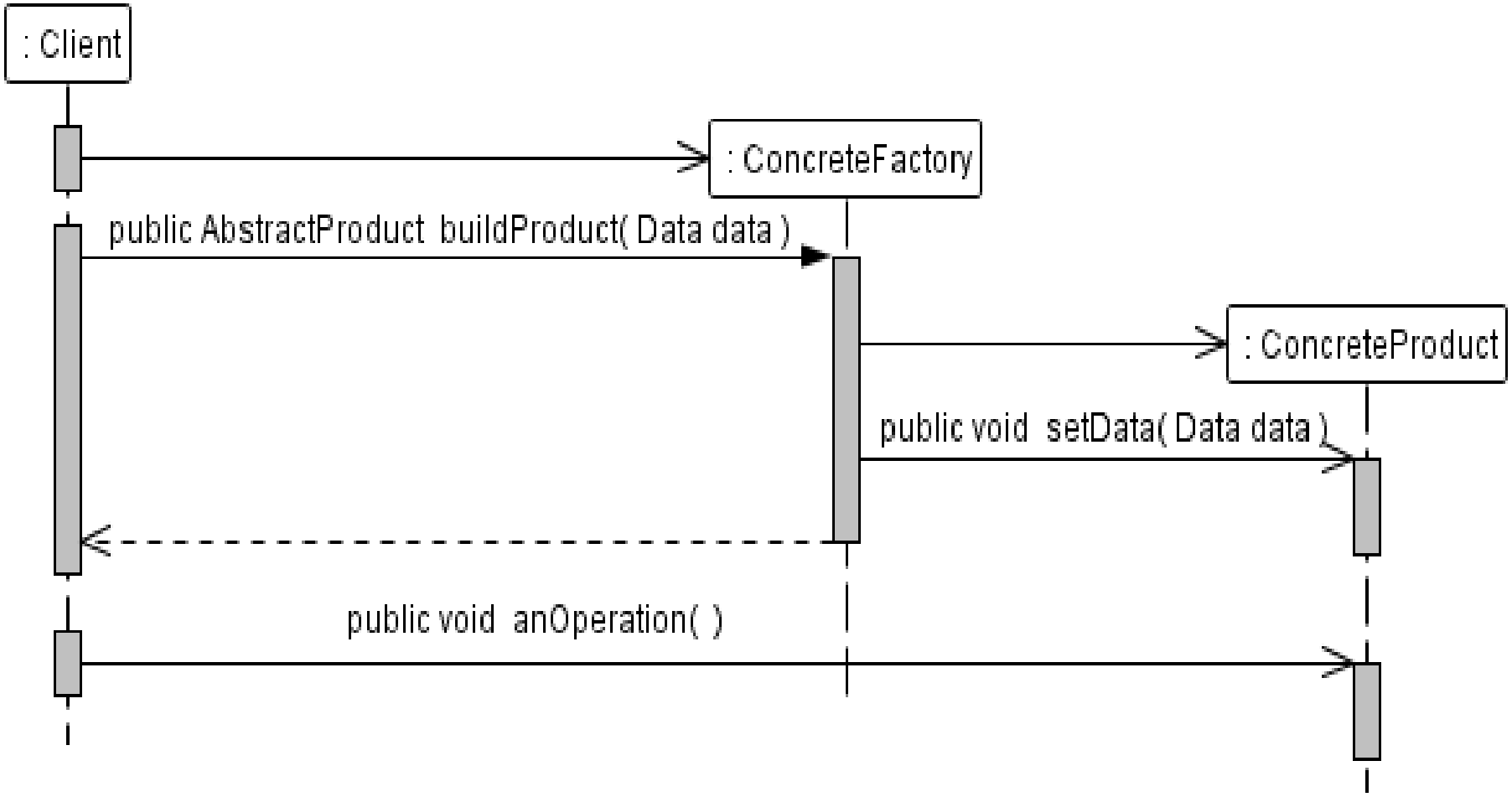
- **Rozwiązanie**: Obiekt typu ***Director*** zleca budowę obiektu typu ***Product*** obiektom typu ***ConcreteBuilder*** implementujących interfejs ***Builder***

- **Klient wzorca**: Klient zleca obiektowi typu ***Director*** wykonanie obiektu ***Product*** dostarczając mu obiekt typu ***ConcreteBuilder***

- **Rezultat**:
  - Udostępniony obiektom typu ***Director*** abstrakcyjny interfejs ***Builder*** pozwala na dowolny sposób konstruowania obiektów typu ***Product***
  - Oddzielenie kodu służącego do konstruowania obiektów typu ***Product*** od kodu służącego do używania tych obiektów np. obiekty typu ***Director*** utworzone do odczytu dokumentów w formacie RTF nie wpływają na kod klienta wzorca, który zajmuje się analizą treści dokumentów a jednocześnie może zlecić obiektowi typu ***Director*** za pomocą obiektu typu ***ConcreteBuilder*** wygenerowanie tej treści jako innego złożonego obiektu typu ***Product*** np. reprezentującego format XML
  - Algorytm tworzenia obiektu typu ***Product*** jest niezależny od tworzonych składowych, które mogą być dowolnego typu
  - Lepsza kontrola budowy obiektu typu ***Product*** przez obiekt ***ConcreteBuilder*** za pośrednictwem implementowanych operacji interfejsu ***Builder*** i sterowanych przez obiekt typu ***Director***

2) Fabryka abstrakcyjna – *Abstract Factory*




: Client
: ConcreteFactory
public AbstractProduct buildProduct(Data data)
: ConcreteProduct
public void setData(Data data)
public void anOperation( )

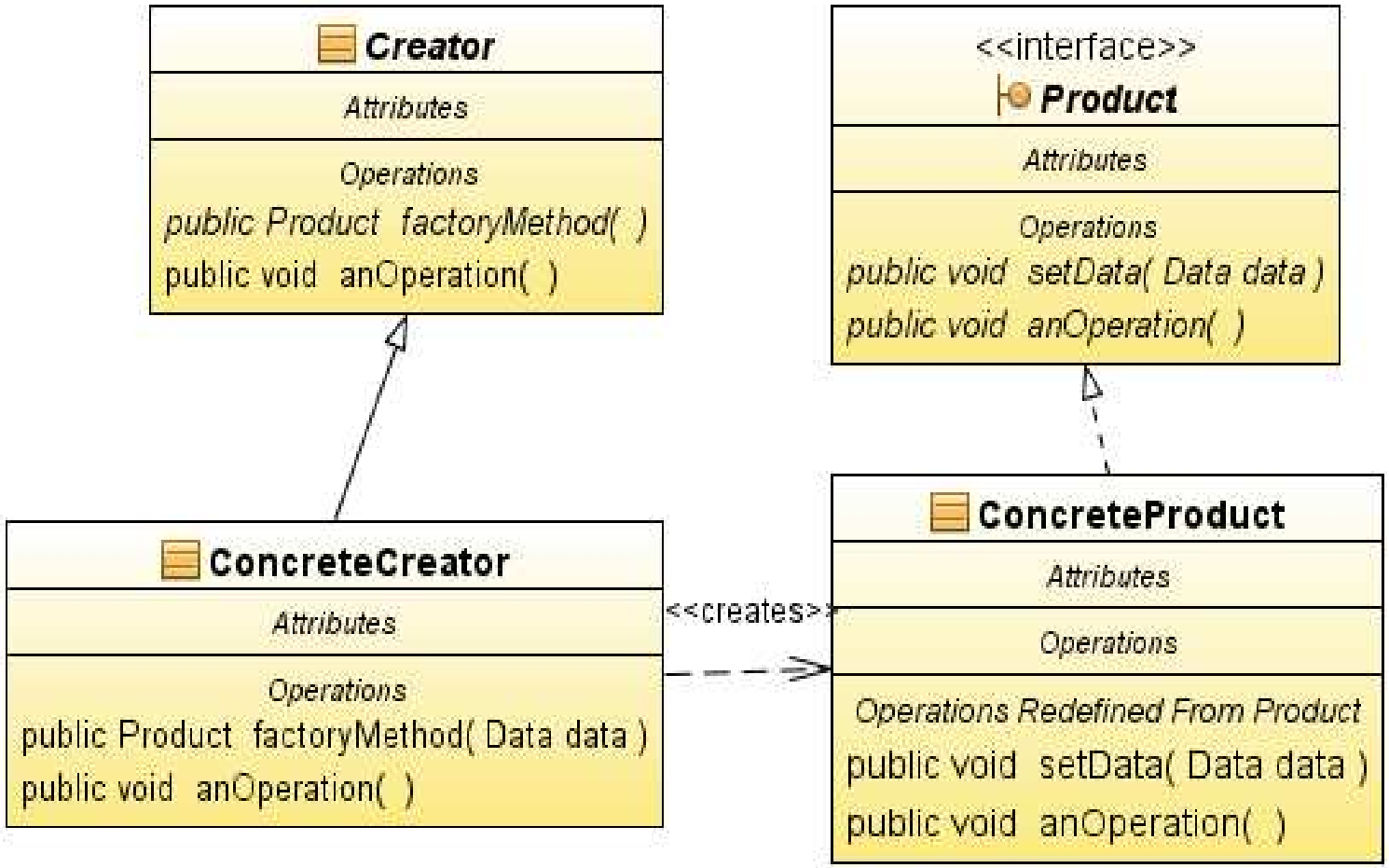
Charakterystyka wzorca abstrakcyjnej fabryki obiektów

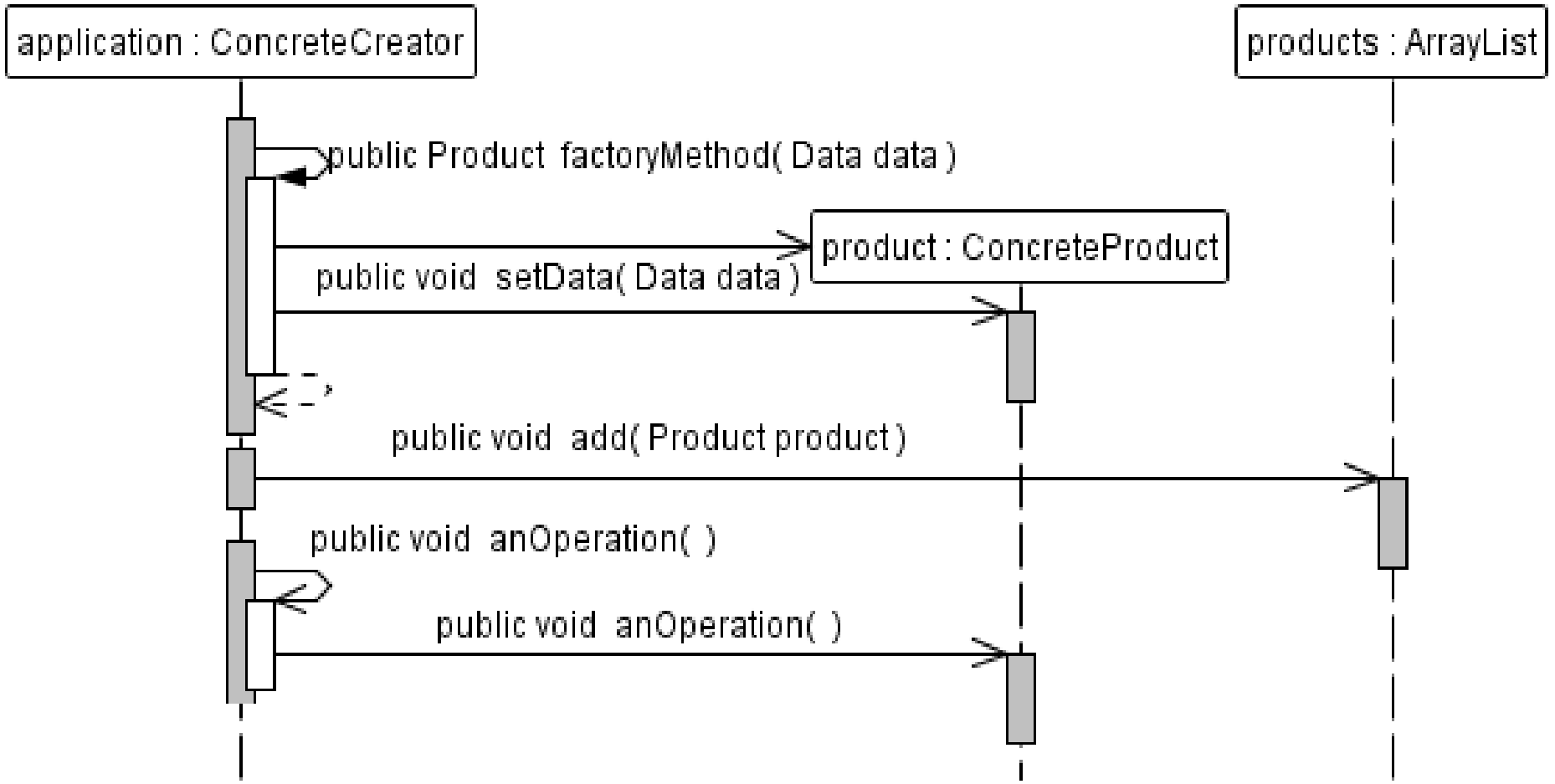
- **Problem**: Tworzenie właściwych rodzin powiązanych lub zależnych obiektów w przypadkach:
  - różne systemy operacyjne,
  - różne wymagania dotyczące efektywności, wydajności itp.
  - różne wersje aplikacji
  - różne typy aplikacji współpracujących (np. różne typy baz danych)
  - różne funkcjonalności dla różnych użytkowników
  - różne grupy elementów zależnych od ustawień związanych z lokalizacją (np. format danych)
- **Rozwiązanie**: Obiekt typu ***Client*** używa interfejsu fabryki abstrakcyjnej (***AbstractFactory***) i interfejsu klas bazowych (***AbstractProduct***). Instancjami, które implementują te interfejsy są obiektu typu ***ConcreteFactory*** i ***ConcreteProduct.*** Każdy obiekt typu ***ConcreteFactory*** potrafi tworzyć jedną z rodzin obiektów klas ***ConcreteProduct***.
- **Klient wzorca**: jako obiekt klasy ***Client*** **zarządza** obiektami tworzonymi przez fabrykę obiektów, **ale jest niezależny** od reguł tworzenia tych obiektów

- **Rezultat**:
  - Izolacja reguł tworzenia obiektów od reguł określających sposób używania obiektów
  - Określenie reguł tworzenia obiektów, które mogą najlepiej realizować cele aplikacji
  - Konfiguracja aplikacji za pomocą powiązanych rodzin obiektów np. ***TProdukt*** i ***TPromocja***
  - System używa tworzone obiekty znając klasy bazowe tych obiektów i klasy bazowe fabryk
- **Implementacja**: Zdefiniowanie klas typu „Control ”. Do wyboru obiektów można stosować pliki konfiguracyjne, tabele baz danych itp.

  Przykładem fabryki zastosowanej w implementacji wzorca **Domain Store** przez pakiet TopLink jest klasa typu ***EntityManagerFactory*** jako typ ***AbstractFactory,*** natomiast klasa typu ***EntityManager*** reprezentuje klasę typu ***AbstractProduct***. Na podstawie pliku ***persistence.xml*** jest tworzony konkretny obiekt fabryki ***EntityManagerFactory*** jako typ ***ConcreteFactory,*** która tworzy obiekt konkretny typu ***EntityManager*** jako obiekt typu ***ConcreteProduct***. Plik persistence.xml określa technologię np. ***TopLink***. Konkretny obiekt typu ***EntityManager*** jest powiązany z innymi konkretnymi obiektami, np. typu ***Transaction*** i ***Query***

3) Metoda wytwórcza – *Factory Method*




application : ConcreteCreator
products : ArrayList
public Product factoryMethod(Data data)
product : ConcreteProduct
public void setData(Data data)
public void add(Product product)
public void anOperation( )
public void anOperation( )

Charakterystyka wzorca metody wytwórczej

- **Problem**: Obiekt klasy A przetwarzającej dane innego obiektu lub obiekt klasy B pochodnej klasy A, nie wie, jaki obiekt należy przetwarzać z danej rodziny obiektów.

- **Rozwiązanie**: Obiekt typu ***Creator*** deklaruje metodę wytwórczą, która deklaruje wytwarzany obiekt typu ***Product.*** Działanie jest przekazane do obiektu typu ***ConcreteCreator***, który wytwarza obiekt typu ***ConcreteProduct*** metodą wytwórczą, odpowiednio zaimplementowaną w klasie ***ConcreteCreator***.

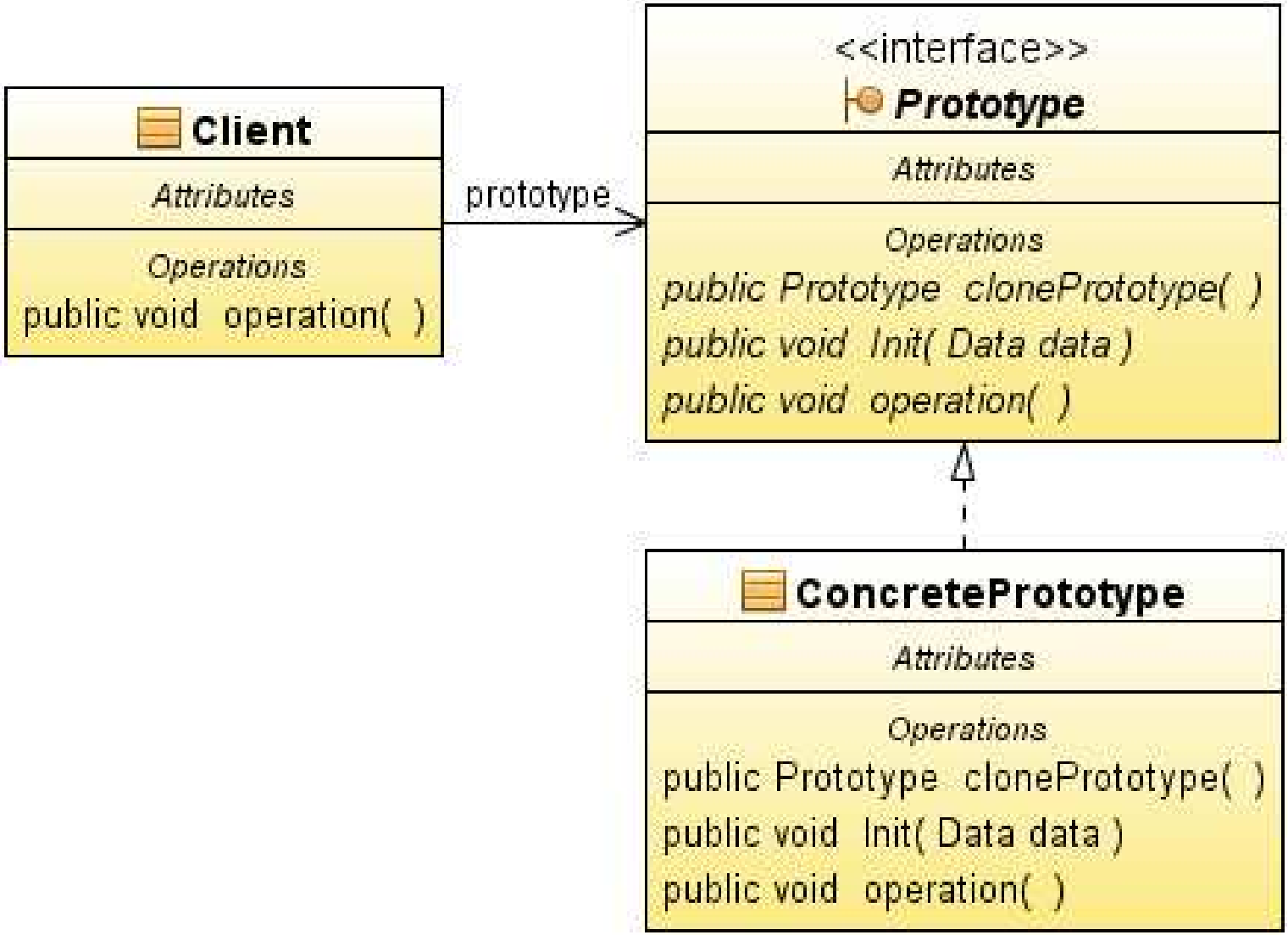
- **Klient wzorca**: klient wybiera odpowiedni obiekt klasy ***ConcreteCreator*** z metodą wytwórczą odpowiednią do wytworzenia obiektu typu ***ConcreteProduct.*** Przetwarzanie obiektu typu ***ConcreteProdukt*** polega na tym, że tylko metoda wytwórcza zna reprezentacje obiektu i sposób jego tworzenia, natomiast pozostałe metody znają interfejs abstrakcyjny klasy ***Product*** i powinny używać jedynie te obiekty.

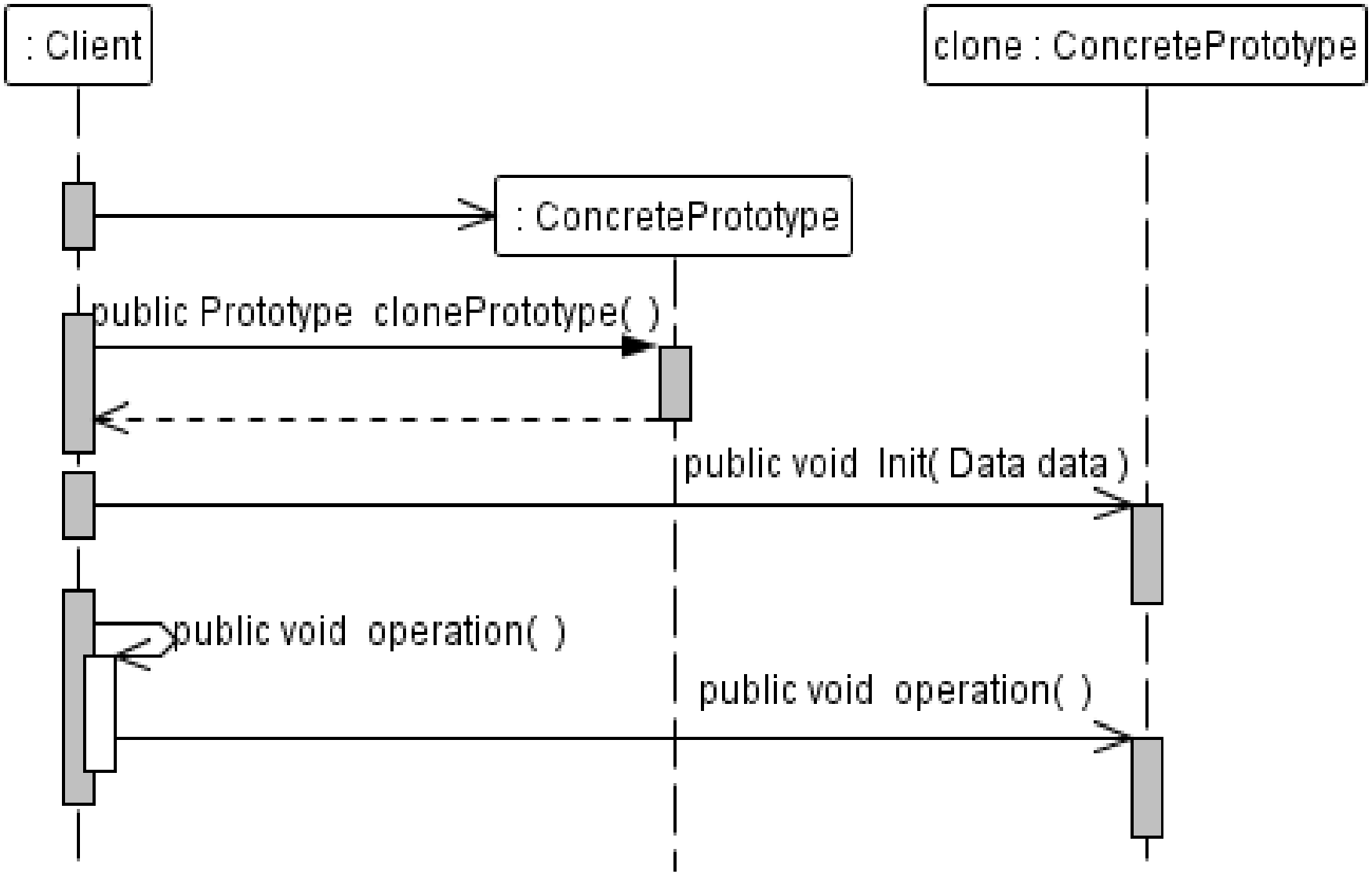
- **Rezultat**:
  - Izolacja reguł tworzenia obiektów od reguł określających sposób używania obiektów w ramach rodziny klas ***Creator***
  - Określenie reguł tworzenia obiektów, które mogą najlepiej realizować cele aplikacji
  - Obiekt typu ***ConcreteCreator*** powinien służyć również do używania obiektu typu ***ConcreteProdukt*** lub jego pochodnego, a jedynie jego metoda wytwórcza powinna znać reguły tworzenia obiektów typu ***ConcreteProdukt.***

Pakiet **Swing:** Klasa **JPanel (*ConcreteCreator*)** dziedzicząca po klasie abstrakcyjnej **JComponent** (***Creator***) używa **metody wytwórczej getComponentGraphics**, i tworzy obiekt typu **DebuggGraphics** (***ConcreteProduct***) o interfejsie typu Graphics (***Product***)

```
/**
   * Returns the graphics object used to paint this component.
   * If <code>DebugGraphics</code> is turned on we create a new
   * <code>DebugGraphics</code> object if necessary.
   * Otherwise we just configure the
   * specified graphics object's foreground and font.
   *
   * @param g the original <code>Graphics</code> object
   * @return a <code>Graphics</code> object configured for this component  */
  protected Graphics getComponentGraphics(Graphics g) {
      Graphics componentGraphics = g;
      if (ui != null && DEBUG_GRAPHICS_LOADED) {
          if ((DebugGraphics.debugComponentCount() != 0) &&
                  (shouldDebugGraphics() != 0) &&
                  !(g instanceof DebugGraphics)) {
              componentGraphics = new DebugGraphics(g, this);
          }
      }
      componentGraphics.setColor(getForeground());
      componentGraphics.setFont(getFont());
      return componentGraphics;
  }
```

4) Prototyp - *Prototype*




: Client
clone : ConcretePrototype
: ConcretePrototype
public Prototype cloneProtype( )
public void Init(Data data)
public void operation( )
public void operation( )

Charakterystyka wzorca Prototyp

- **Problem**: Oddzielenie w kodzie programu kodu tworzenia obiektów od ich używania bez budowania hierarchii klas fabryk w sytuacji, gdy potrzebna jest ograniczona liczba obiektów.
- **Rozwiązanie**: Obiekt typu ***Client*** otrzymuje potrzebny obiekt typu ***Prototype*** jako ***ConcretePrototype*** na drodze klonowania obiektów
- **Klient wzorca**: Obiekt używa sklonowane obiekty typu ***Prototype***.
- **Rezultat**:
  - Dodawanie i usuwanie obiektów bez pośrednictwa obiektów typu fabryki
  - Zredukowanie liczby podklas
  - Dynamiczne ładowanie klas typu ***Prototype***
- **Implementacja**:
  Metoda klasy ArrayList do utworzenia własnej kopii

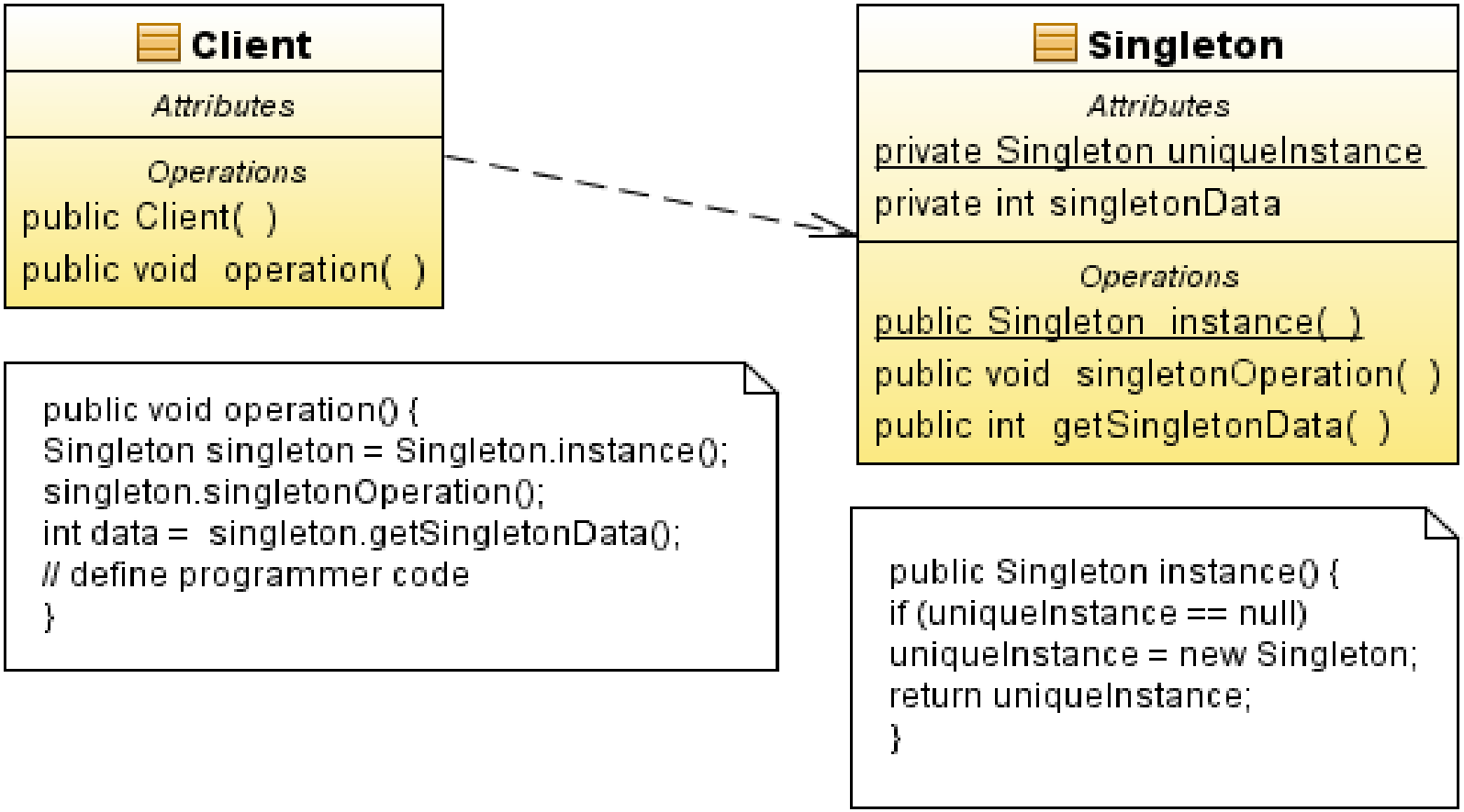
```
public Object clone() {
  try {
     ArrayList<E> v = (ArrayList<E>) super.clone();
     v.elementData = Arrays.copyOf(elementData, size);
     v.modCount = 0;
     return v;
  } catch (CloneNotSupportedException e) {
     throw new InternalError();}
  }
```

Przykład klonowania kolekcji obiektów

```
public class Main {

    public static void main(String[] args) {
        ArrayList kolekcja1, kolekcja2 = new ArrayList();
        kolekcja2.add(new Integer(1));
        kolekcja2.add("B");
        kolekcja1=(ArrayList)kolekcja2.clone();
        kolekcja1.add("C");
        kolekcja2.remove(0);
        System.out.println(kolekcja1.toString());   //[1, B, C]
        System.out.println(kolekcja2.toString());   //[B]
    }
}
```

5) Singleton


Client
Attributes
Operations
public Client( )
public void operation( )
Singleton
Attributes
private Singleton uniqueInstance
private int singletonData
Operations
public Singleton instance( )
public void singletonOperation( )
public int getSingletonData( )
public void operation() {
Singleton singleton = Singleton.instance();
singleton.singletonOperation();
int data = singleton.getSingletonData();
// define programmer code
}
public Singleton instance() {
if (uniqueInstance == null)
uniqueInstance = new Singleton;
return uniqueInstance;
}

Charakterystyka wzorca Singleton

- **Problem**: Gwarancja, że klasa ma tylko jeden egzemplarz i istnieje globalny dostęp do tego obiektu np. system plików lub system okien
- **Rozwiązanie**: Obiekt typu ***Singleton*** sam pilnuje, aby nie powstał inny obiekt tego samego typu
- **Klient wzorca**: Obiekt typu ***Singleton*** może mieć wielu klientów
- **Rezultat**:
  - Mniejsza przestrzeń nazw
  - Kontrolowany dostęp do jedynego egzemplarza
- **Implementacja**: atrybuty klas typu **static** np. W klasie System atrybut out typu PrintStream reprezentujący standardowe urządzenie wyjściowe np.
  System.out.println(”Singleton”),
  gdzie atrybut out w klasie System zdefiniowano:

  public static final PrintStream **out**

2.2. Wzorce strukturalne – tworzenie złożonych obiektowych struktur danych

Składanie klas i obiektów w większe struktury

Wzorce klasowe: **zastosowanie dziedziczenia i polimorfizmu do składania struktur interfejsów lub ich implementacji**

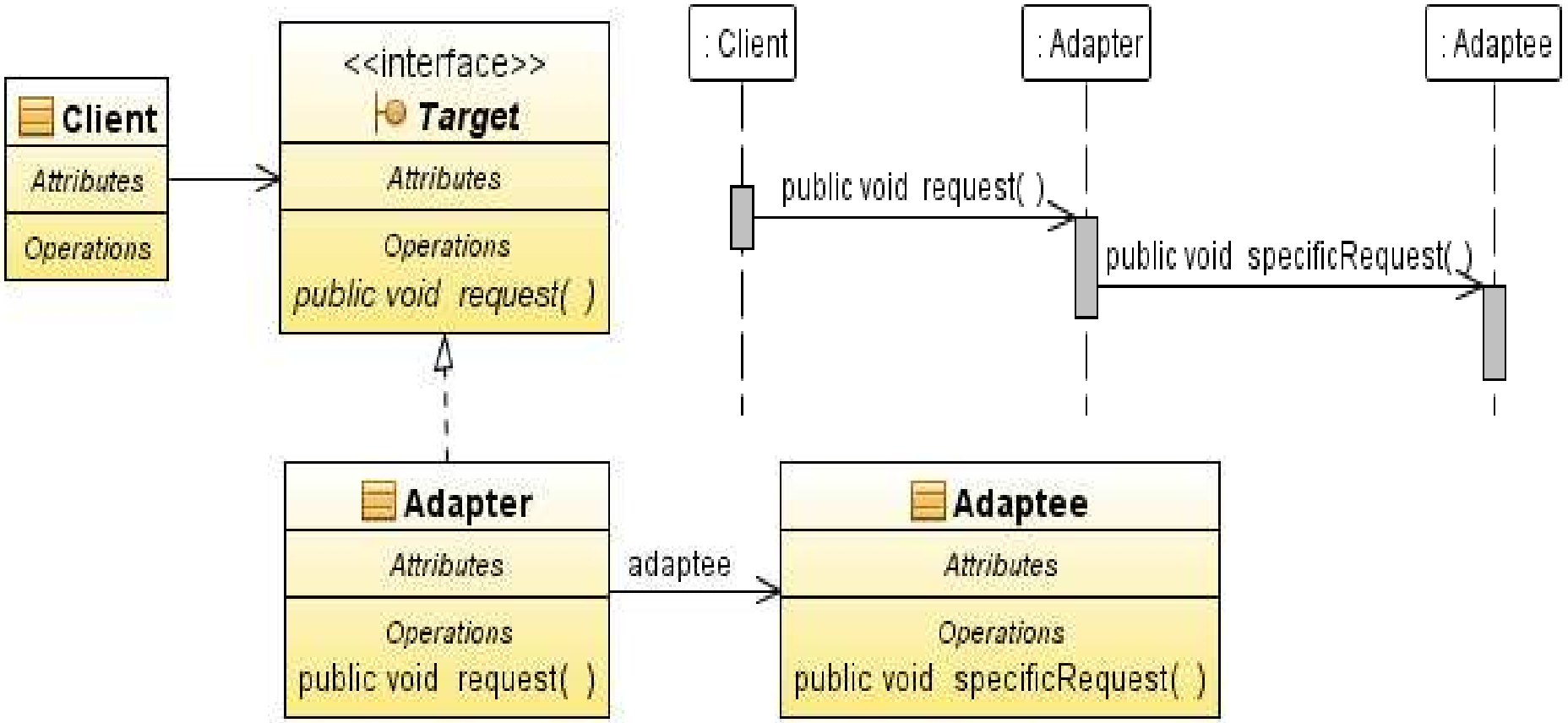
Wzorce obiektowe: opisują sposoby składania obiektów w celu uzyskania nowej funkcjonalności. Istnieje możliwość składania obiektów podczas działania programu

- **Adapter – *wzorzec klasowy i obiektowy***
- **Dekorator - *Decorator – wzorzec obiektowy***
- **Fasada - *Facade - wzorzec obiektowy***
- **Kompozyt – *Composite - wzorzec obiektowy***
- **Most – *Bridge - wzorzec obiektowy***
- **Pełnomocnik – *Proxy - wzorzec obiektowy***
- **Pyłek – *Flyweight - wzorzec obiektowy***

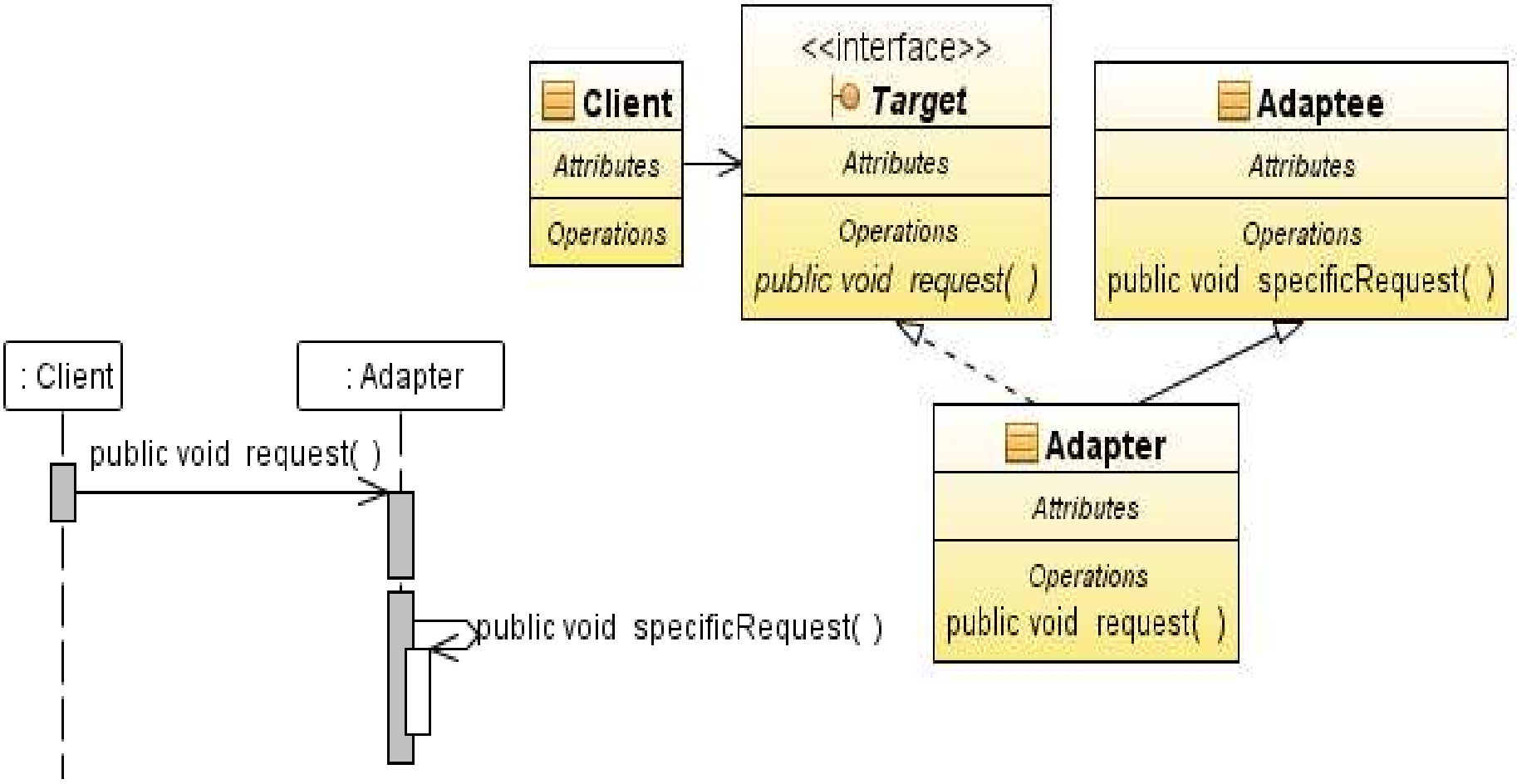
Wybór wzorca strukturalnego

Wzorce strukturalne	Aspekt, który może się zmienić
1)Adapter - klasowy i obiektowy	Interfejs obiektu
2)Bridge - obiektowy	Implementacja obiektu
3)Composite - obiektowy	Struktura i schemat obiektu
4) Decorator - obiektowy	Zadanie obiektu bez zmiany podklas
5) Facade - obiektowy	Interfejs posystemu
6) Flyweight - obiektowy	Koszt przechowywania obiektów w pamięci
7)Proxy -- obiektowy	Sposób dostepu oraz położenie obiektu

1a) Adapter obiektów (wzorzec obiektowy) – *Adapter (składanie obiektów typu Adaptee)*



1b) Adapter klas (wzorzec klas) – adapter jest podklasą klasy adaptowanej *Adaptee*

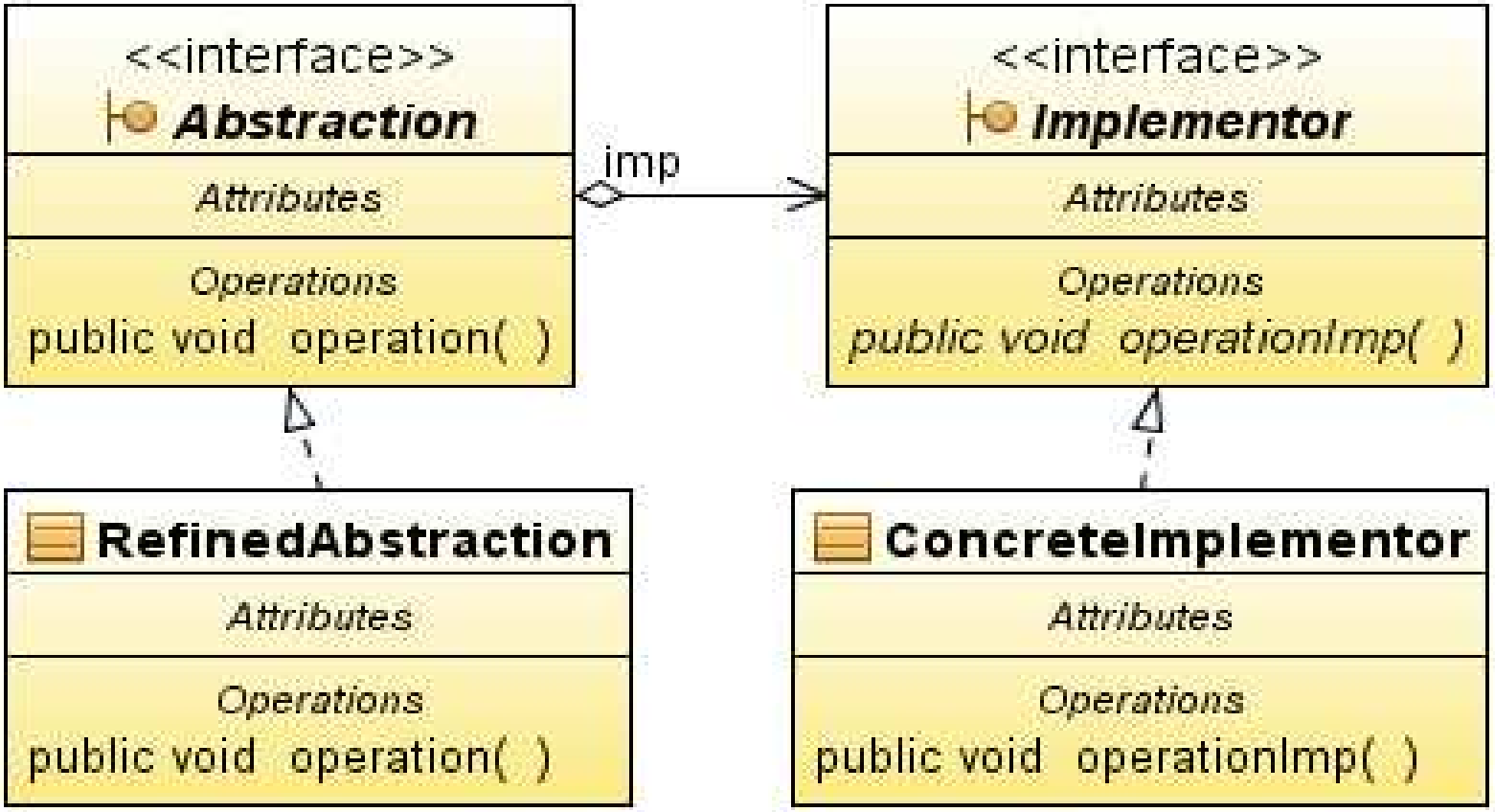


Charakterystyka wzorca Adapter

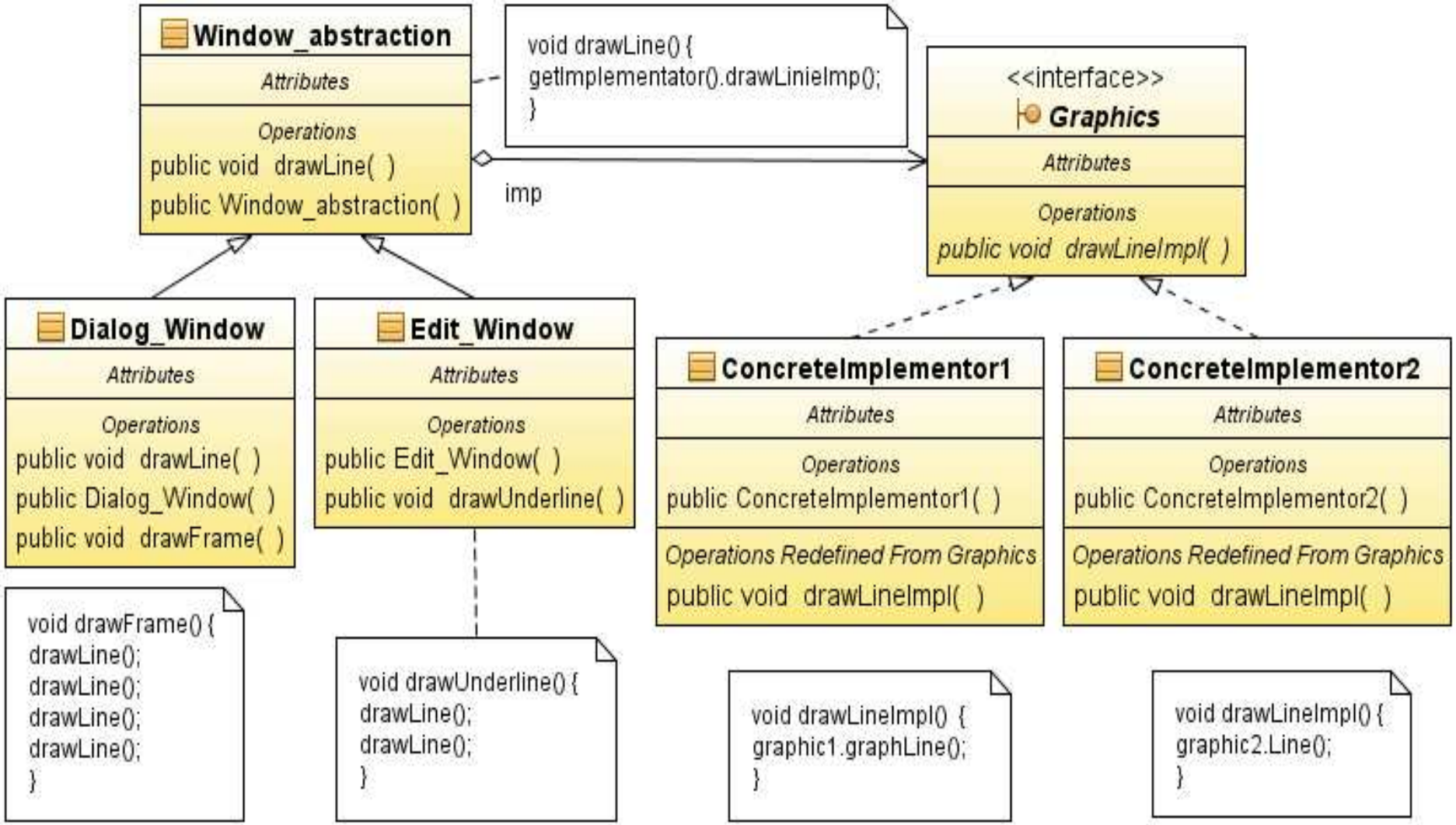
- **Problem**: Należy dostosować interfejs klasy do interfejsu oczekiwanego przez klienta wzorca np. związanego ze zmianą biblioteki klas obsługujących grafikę
- **Rozwiązanie**: Obiekt typu ***Client*** używa obiektów typu ***Adapter*** implementujących interfejs typu ***Target*** i jednocześnie pośredniczących w dostępie do metod obiektów klasy ***Adaptee***.
- **Klient wzorca**: Klient ***Client*** wzorca jest niezależny od zmian nagłówków metod lub ich zmian ich definicji w bibliotekach klas (***Adaptee***) implementujących specjalizowane operacje np. operacje graficzne, ponieważ klient używa zawsze metod pośrednika typu ***Adapter***, które nie zmieniają nagłówka.

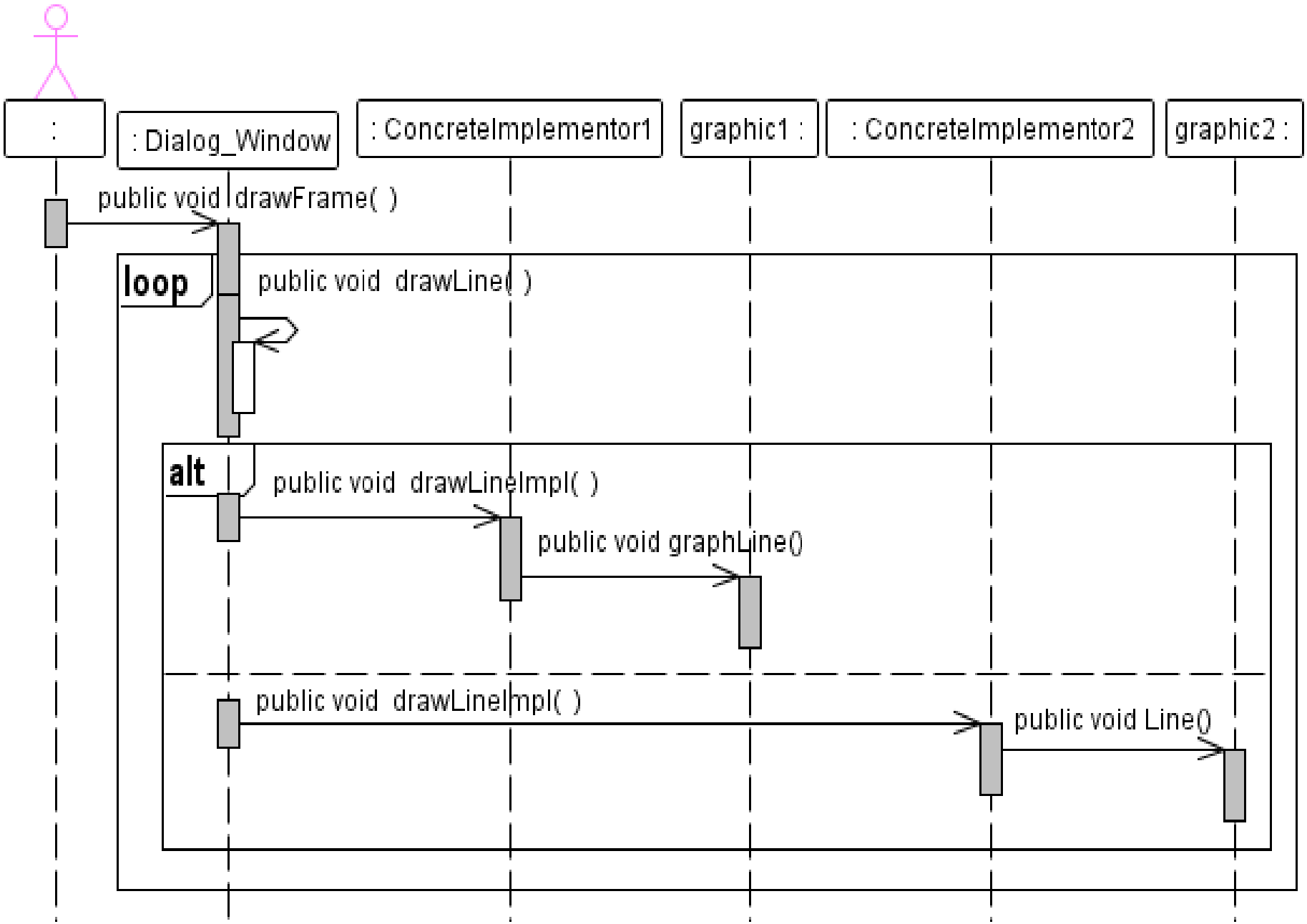
- **Rezultat**:
  - Adapter obiektów (składanie obiektów, folia 1a)):
    - Umożliwia jednemu obiektowi typu ***Adapter*** działanie z wieloma obiektami typu ***Adaptee*** i jej pochodnymi. W obiekcie typu ***Adapter*** może dodać nową funkcjonalność (zaleta)
    - W przypadku hierarchii klas typu ***Adaptee*** odzwierciedlającą zmianę zachowania tego obiektu obiekt ***Adapter*** musi odwoływać się do obiektów podklas ***typu Adaptee***, a nie do adaptowanego typu ***Adaptee***
  - Adapter klas (wielokrotne dziedziczenie):
    - Dostosowanie interfejsu klasy, używanego w programie do interfejsu klasy z nowych bibliotek, jednak nie dotyczy to jej podklas (wada)
    - Umożliwia klasie ***Adapter*** przedefiniowanie części zachowania klasy ***Adaptee***, ponieważ jest jego podklasą (zaleta)
    - Wprowadza tylko jeden obiekt typu ***Adapter*** udostępniający obiekt adaptowany typu ***Adaptee*** (wada)
- **Implementacja**: nowa klasa typu „Boundry”

2) Most - *Bridge – wzorzec obiektowy*



Przykład

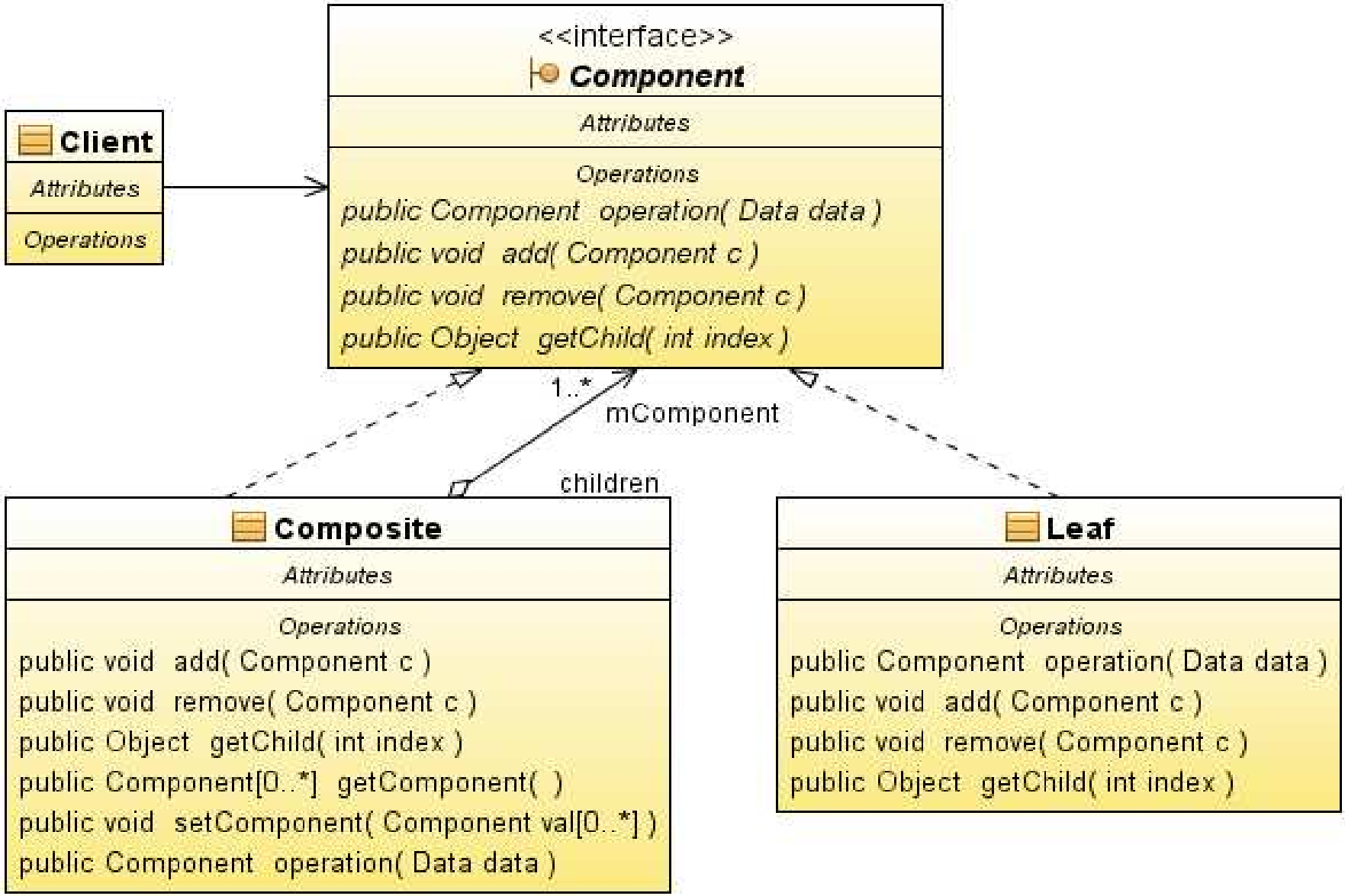

Window_abstraction
Attributes
Operations
public void drawLine( )
public Window_abstraction( )
void drawLine() {
getImplementator().drawLinieImp();
}
imp
<<interface>>
Graphics
Attributes
Operations
public void drawLineImpl( )
Dialog_Window
Attributes
Operations
public void drawLine( )
public Dialog_Window( )
public void drawFrame( )
Edit_Window
Attributes
Operations
public Edit_Window( )
public void drawUnderline( )
ConcreteImplementor1
Attributes
Operations
public ConcreteImplementor1( )
Operations Redefined From Graphics
public void drawLineImpl( )
ConcreteImplementor2
Attributes
Operations
public ConcreteImplementor2( )
Operations Redefined From Graphics
public void drawLineImpl( )
void drawFrame() {
drawLine();
drawLine();
drawLine();
drawLine();
}
void drawUnderline() {
drawLine();
drawLine();
}
void drawLineImpl() {
graphic1.graphLine();
}
void drawLineImpl() {
graphic2.Line();
}


:
: Dialog_Window
: ConcreteImplementor1
graphic1 :
: ConcreteImplementor2
graphic2 :
public void drawFrame( )
loop
public void drawLine( )
alt
public void drawLineImpl( )
public void graphLine()
public void drawLineImpl( )
public void Line()

Charakterystyka wzorca Most

- **Problem**: Należy oddzielić abstrakcję od implementacji, tak aby mogły zmieniać się jedna niezależnie od drugiej
- **Rozwiązanie**: Klasa abstrakcyjna (interfejs) typu ***Abstraction*** rozszerzana przez klasę abstrakcyjną ***RedefineAbstraction*** używa w swoich metodach metod klasy abstrakcyjnej (interfejs) ***Implementor,*** implementowanych przez konkretne klasy ***ConcreteImplementor*** współpracujące z różnymi klasami (o różnych interfejsach) pochodzących z różnych platform, bibliotek.
- **Klient wzorca**: Klient jednakowo traktuje każdy z obiektów klas ***Abstraction*** i ***RedefineAbstraction*** bez wiązania się z konkretna platformą, biblioteką
- **Rezultat**:
  - Oddzielenie abstrakcji od implementacji, eliminacja zależności podczas kompilacji lub działania programu, wprowadzenie architektury wielowarstwowej
  - Rozszerzalność hierarchii klas ***Abstraction*** i ***Implementor***
  - Łatwe dodawanie nowych obiektów
  - Ukrywanie szczegółów implementacji przed klientami szczegółów implementacji
- **Implementacja**: nowa klasa typu „Boundry”

3) Kompozyt – *Composite – wzorzec obiektowy*



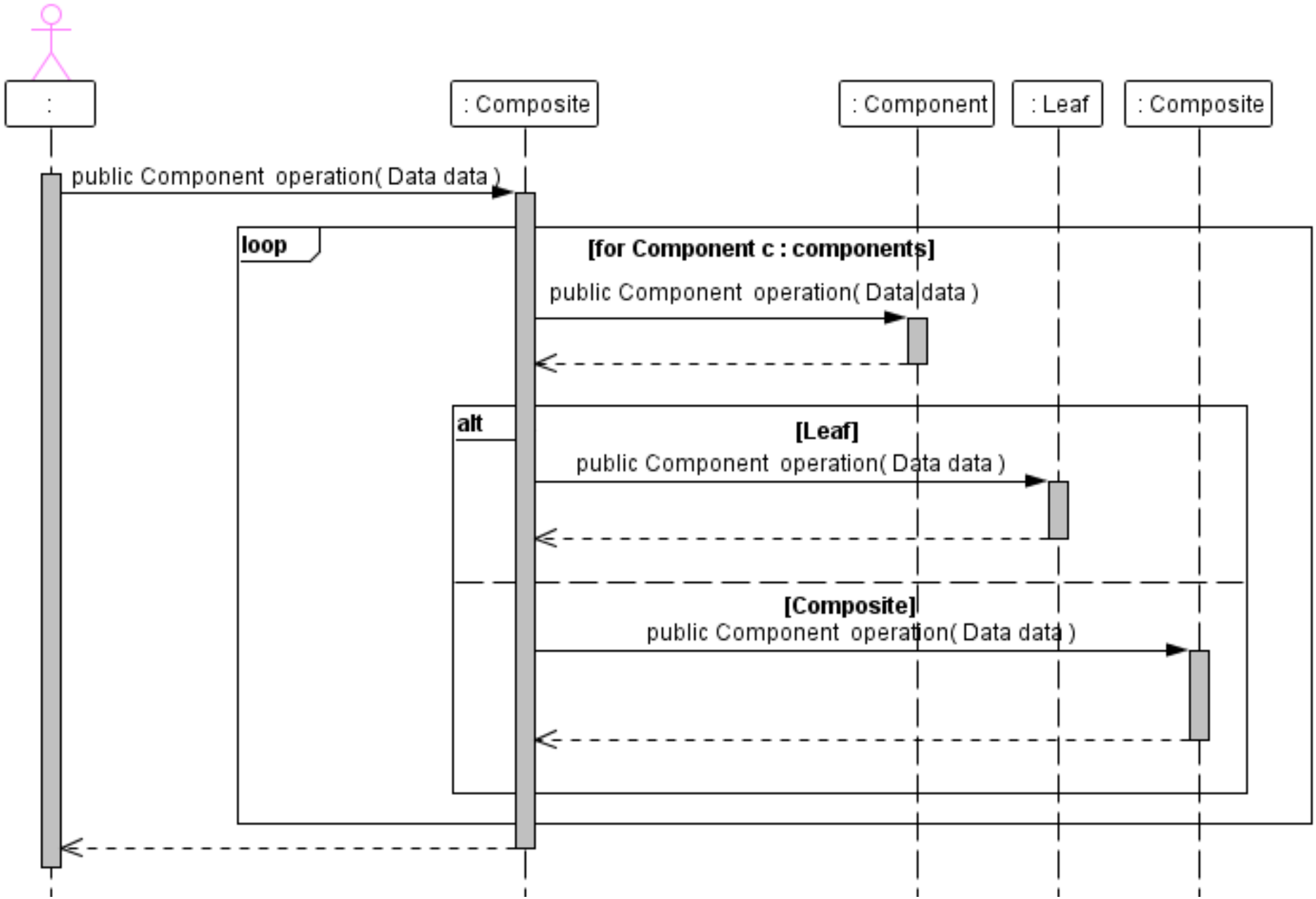
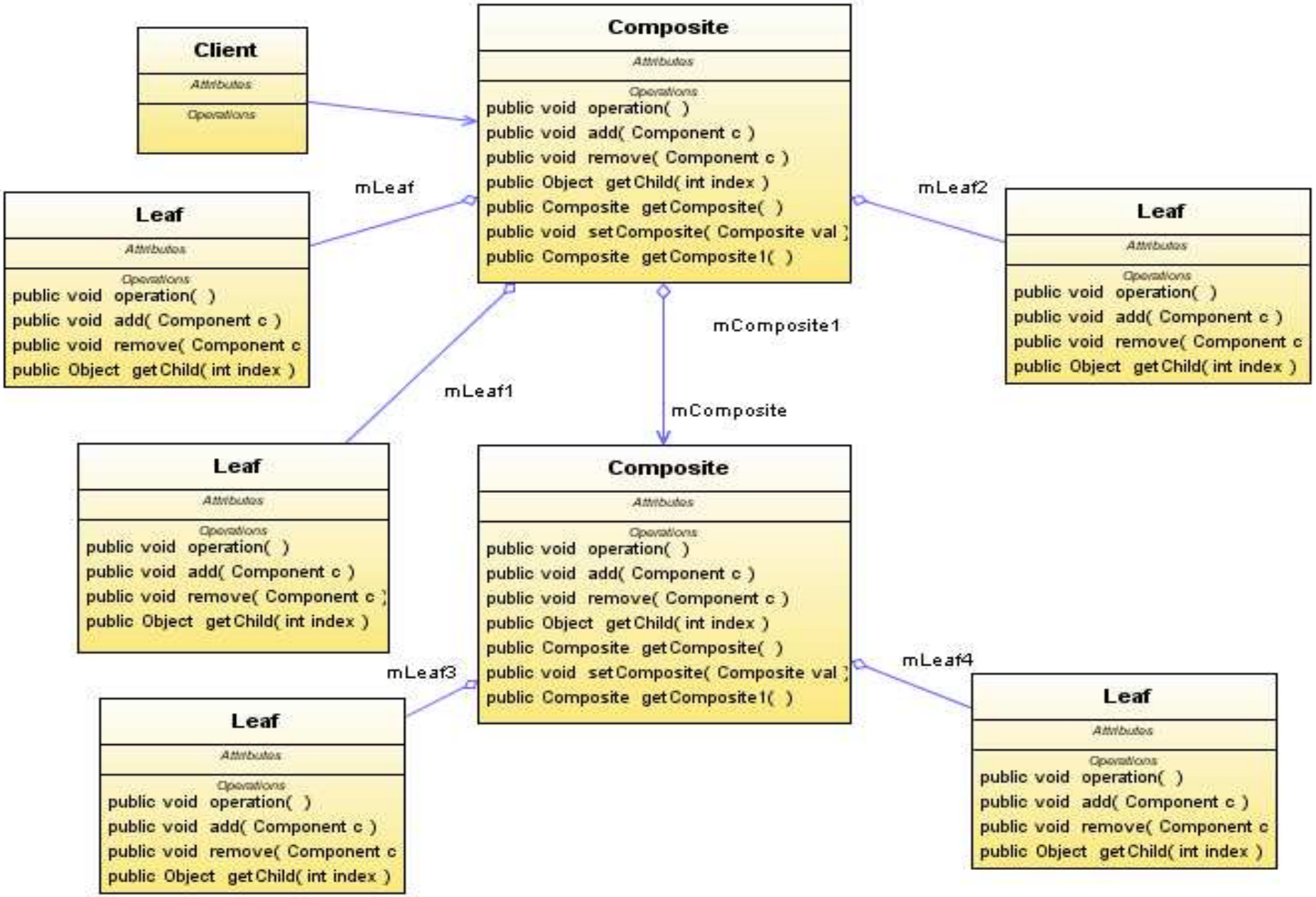
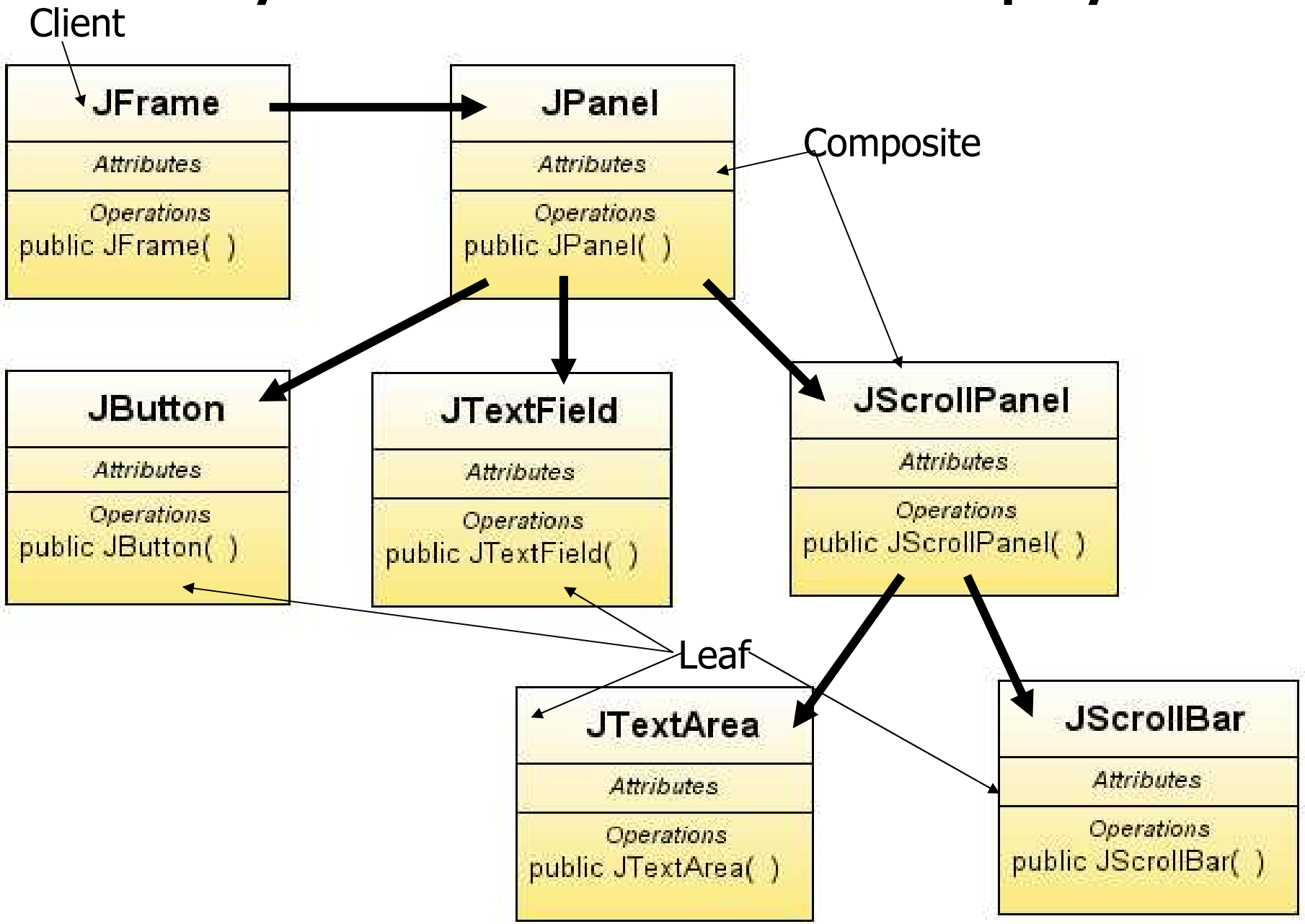

:
: Composite
: Component
: Leaf
: Composite
public Component operation(Data data)
loop
[for Component c : components]
public Component operation(Data data)
alt
[Leaf]
public Component operation(Data data)
[Composite]
public Component operation(Data data)

Diagram obiektów wzorca Kompozyt



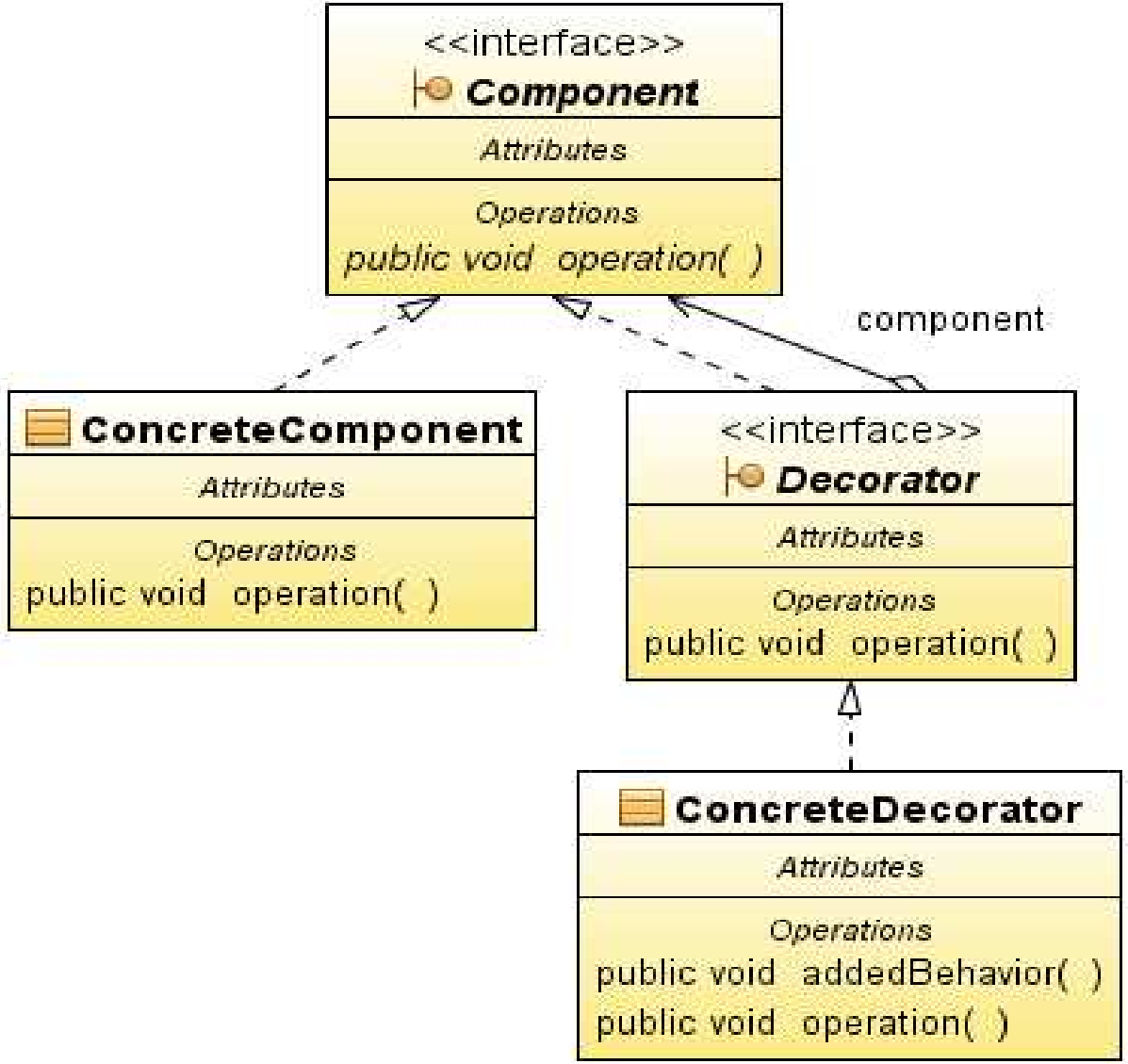
Przykład zastosowania wzorca kompozyt

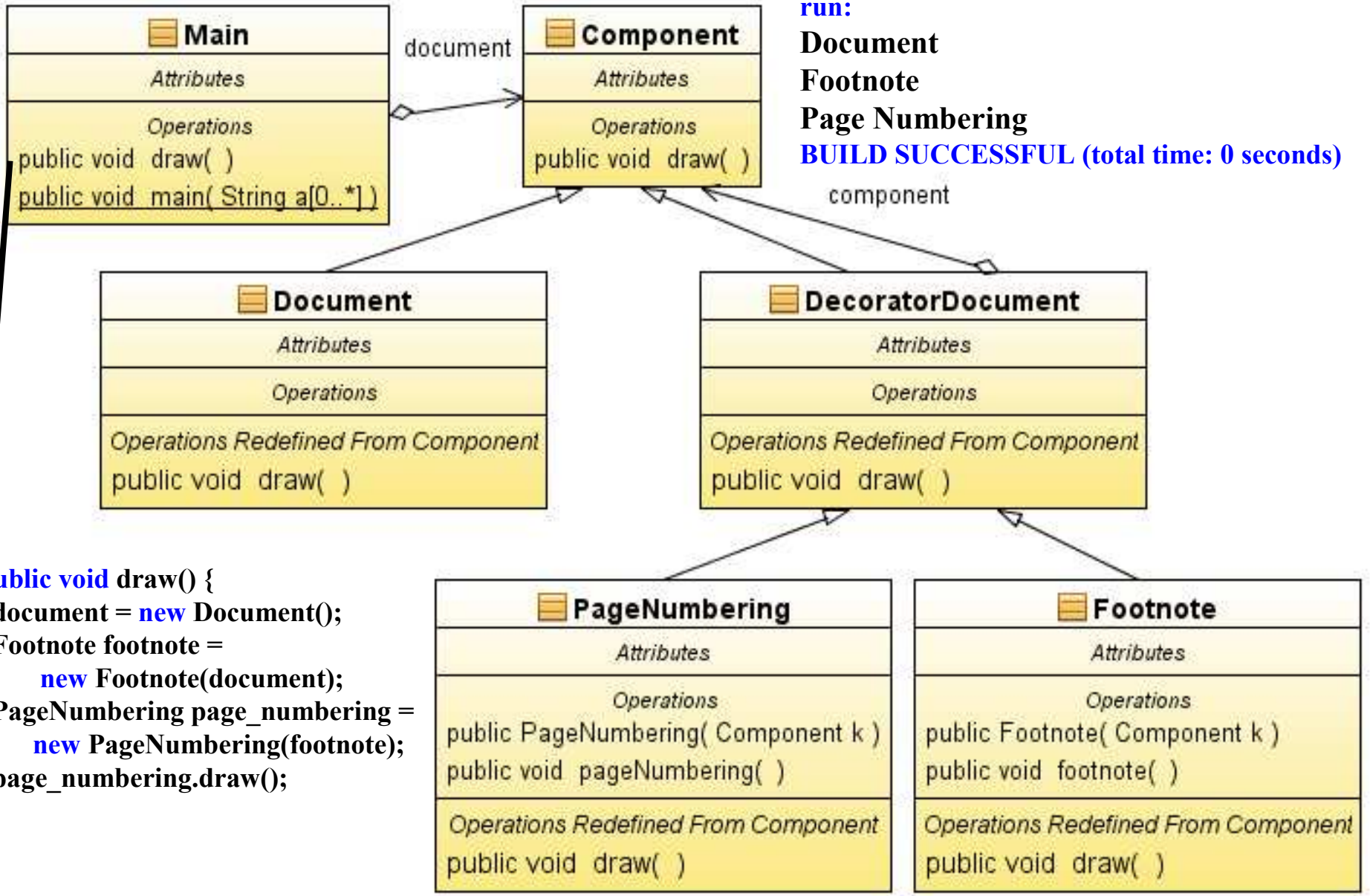


Charakterystyka wzorca Kompozyt

- **Problem**: Składa obiekty w obiektowe struktury danych (drzewiaste) typu część-całość
- **Rozwiązanie**: Klasa abstrakcyjna (interfejs) typu ***Component*** definiuje podstawowe operacje graficzne dla obiektów typu ***Leaf*** i obiektów-rodziców typu ***Composite***
- **Klient wzorca**: Klient jednakowo traktuje każdy z obiektów struktury – jako obiekty typu ***Component***
- **Rezultat**:
  - Rekurencyjne grupowanie obiektów pierwotnych (typu ***Leaf***) i obiektów złożonych (typu Composite)
  - Prosta budowa klienta, który nie musi rozróżniać obiektów pierwotnych i złozonych
  - Łatwe dodawanie nowych obiektów
  - Trudność w zachowaniu ograniczeń przy budowie obiektów złożonych
- **Implementacja**: nowa klasa typu „Boundry" np. w pakiecie ***Swing***
  - klasa ***JComponent*** reprezentuje klasę typu ***Component,***
  - natomiast klasa ***JButton*** klasę typu ***Leaf***,
  - natomiast klasa ***JPanel*** reprezentuje klasę typu ***Composite***.

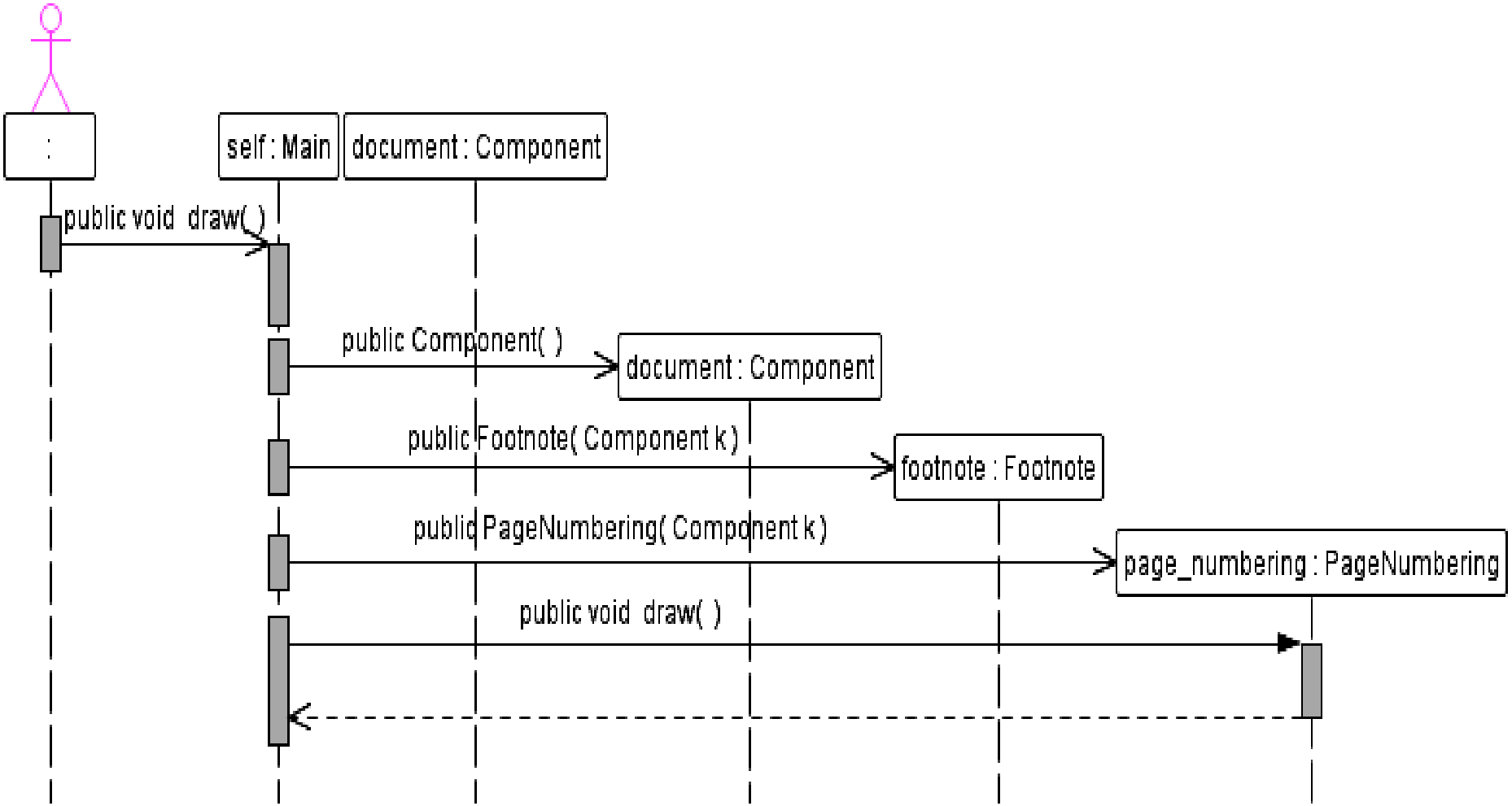
4) Dekorator – *Decorator – wzorzec obiektowy*

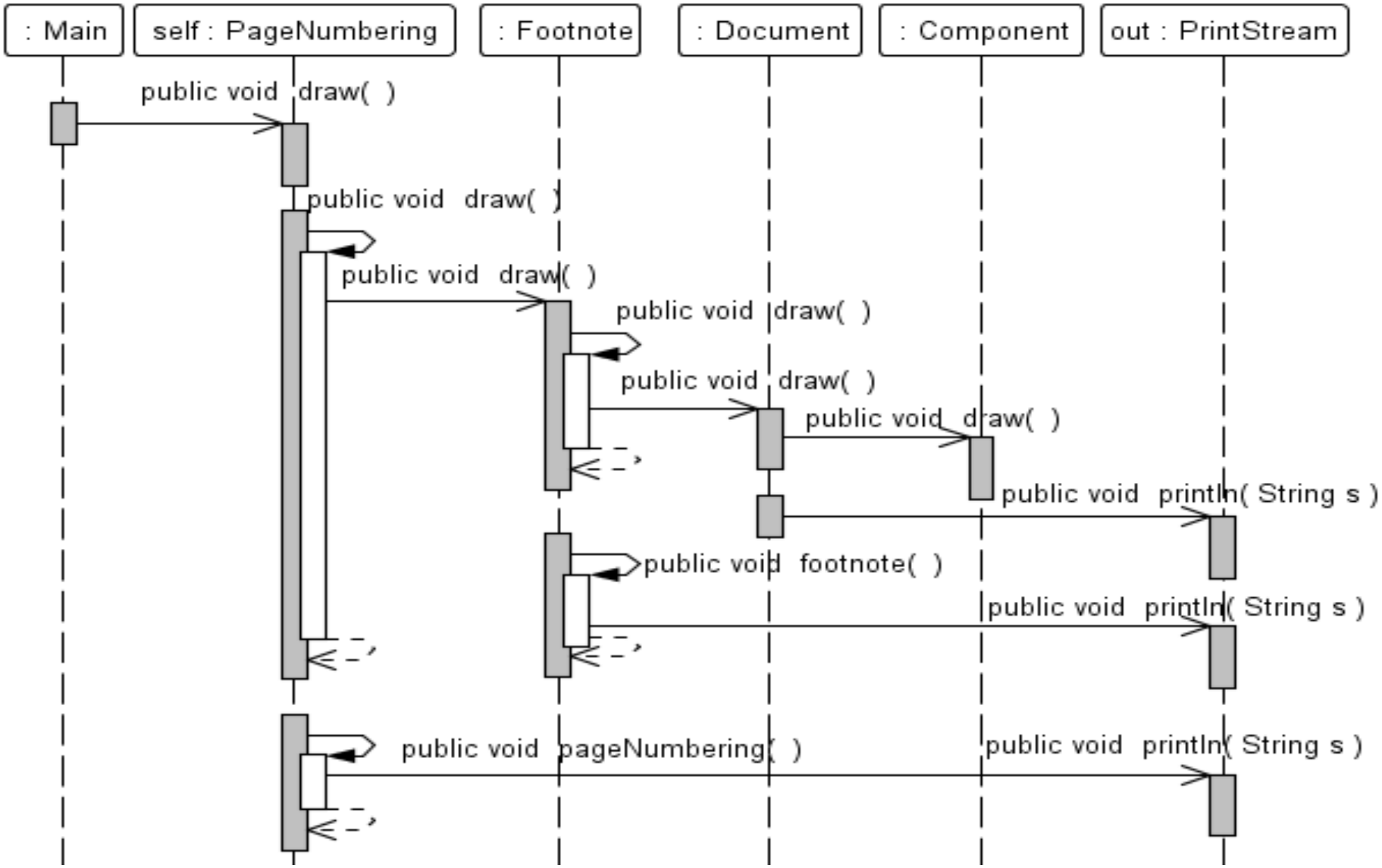





```
public void draw() {
  document = new Document();
  Footnote footnote =
        new Footnote(document);
  PageNumbering page_numbering =
       new PageNumbering(footnote);
  page_numbering.draw();
}
```

```
run:
Document
Footnote
Page Numbering
BUILD SUCCESSFUL (total time: 0 seconds)
```


:
self : Main
document : Component
public void draw( )
public Component( )
document : Component
public Footnote(Component k)
footnote : Footnote
public PageNumbering(Component k)
page_numbering : PageNumbering
public void draw( )


: Main
self : PageNumbering
: Footnote
: Document
: Component
out : PrintStream
public void draw( )
public void draw( )
public void draw( )
public void draw( )
public void draw( )
public void draw( )
public void println(String s)
public void footnote( )
public void println(String s)
public void pageNumbering( )
public void println(String s)

```
class Component {
    public void draw() {    }
}
//-------------------------------------------------------------------------
class Document extends Component {
    public void draw() {
        System.out.println("Document");   }
}
//-------------------------------------------------------------------------
class DecoratorDocument
                        extends Component
{  Component component;
    public void draw() {
        component.draw();    }
}
//-------------------------------------------------------------------------
class FootNote
                extends DecoratorDocument
{  public FootNote(Component k)
      {      component = k;    }

    public void draw() {
       super.draw();
      System.out.println(" FootNote ");    }
}
```

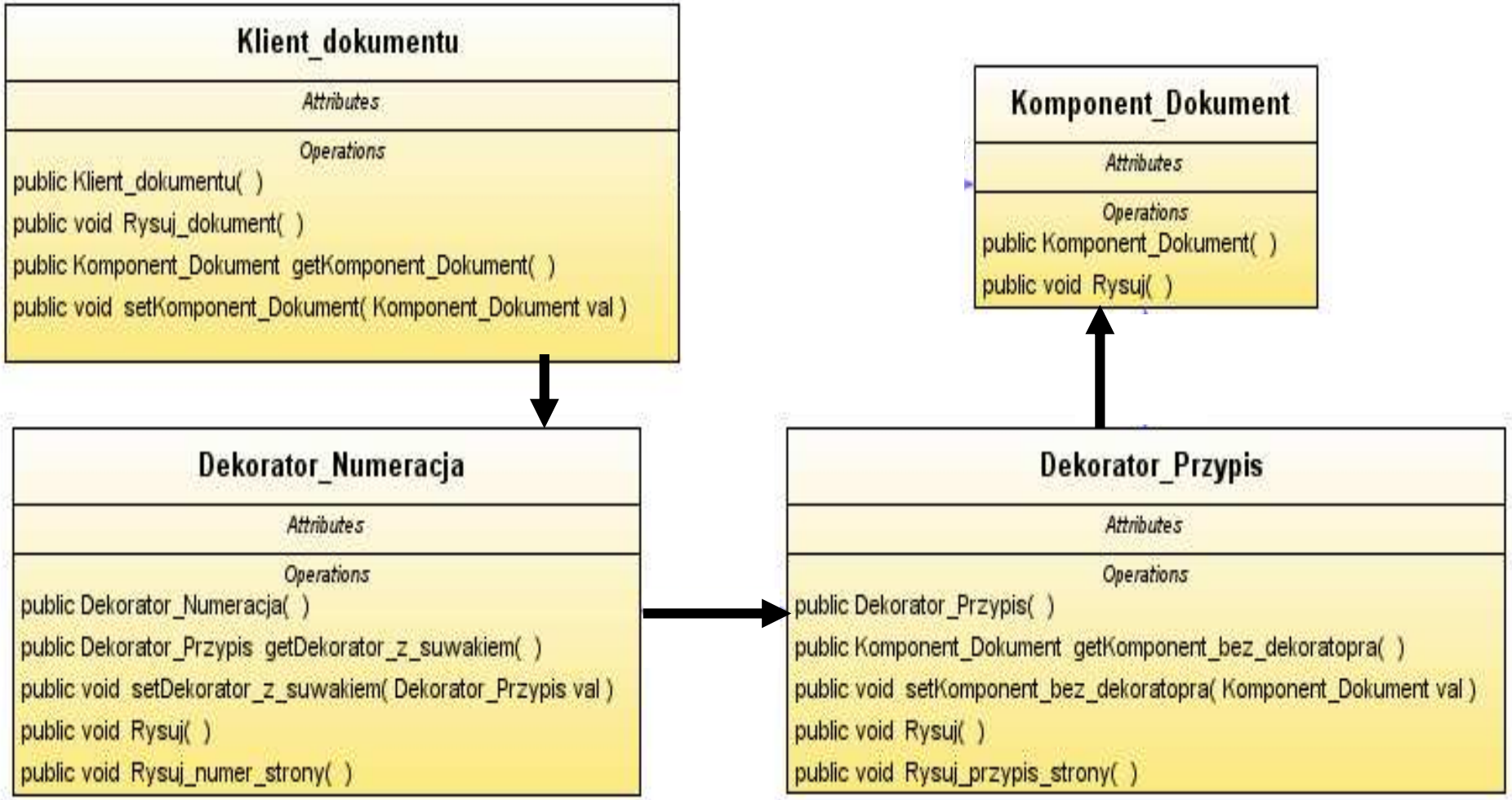
```
class PageNumbering extends  DecoratorDocument
{    public PageNumbering(Component k)
        {      component = k;    }

    public void draw() {
       super.draw();
        System.out.println("PageNumbering ");    }
}
//-------------------------------------------------------------------------
public class Main {
 Component calydokument;

 public void draw() {
   Component dokument = new Document();
   FootNote footNote = new FootNote(dokument);
   calydokument = new PageNumbering(footNote);
   calydokument.draw();
}

public static void main(String a[]) {
        Main main = new Main();
        main.draw();    }
}
```

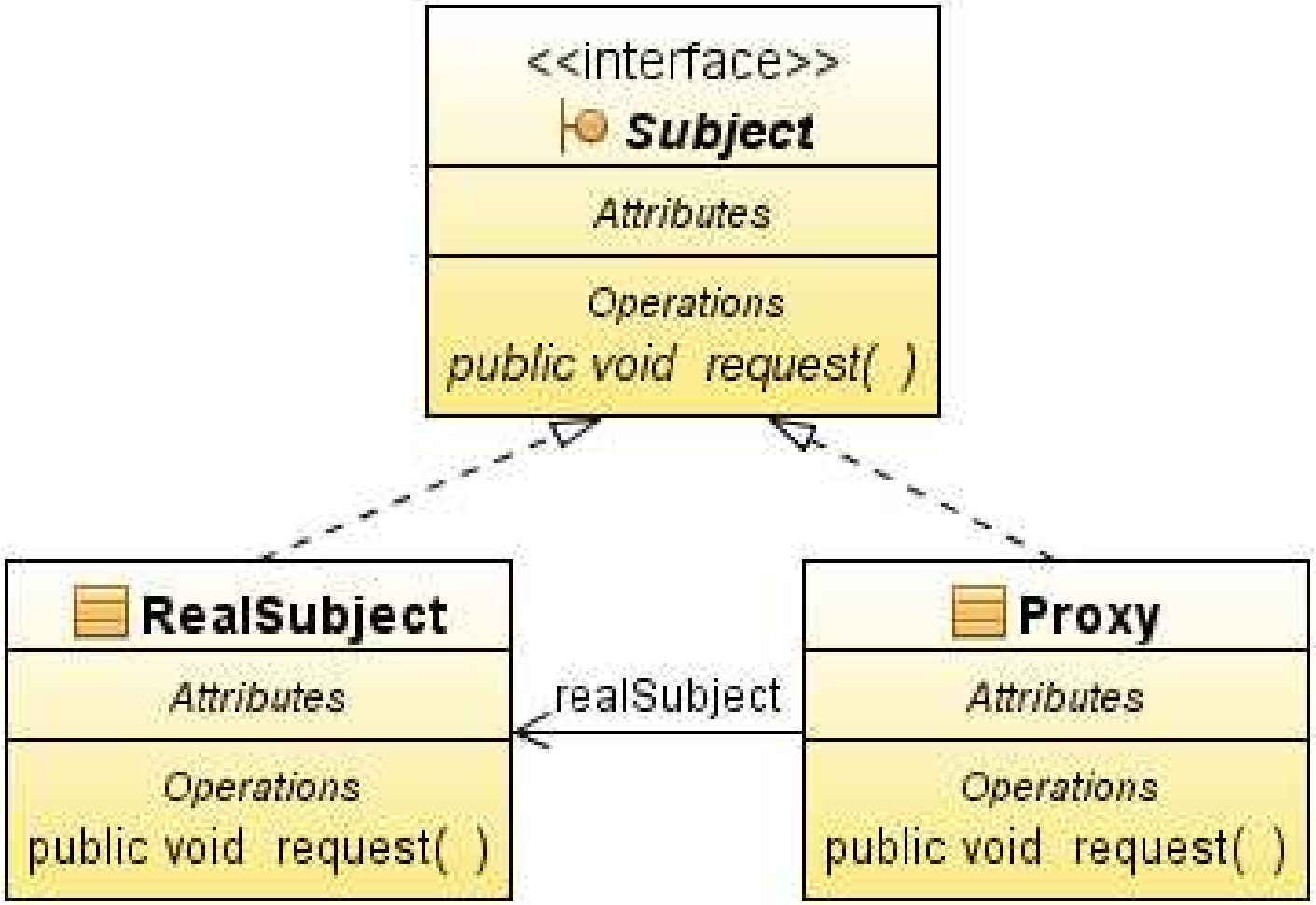
Zastosowanie dekoratora do budowy widoku dokumentu



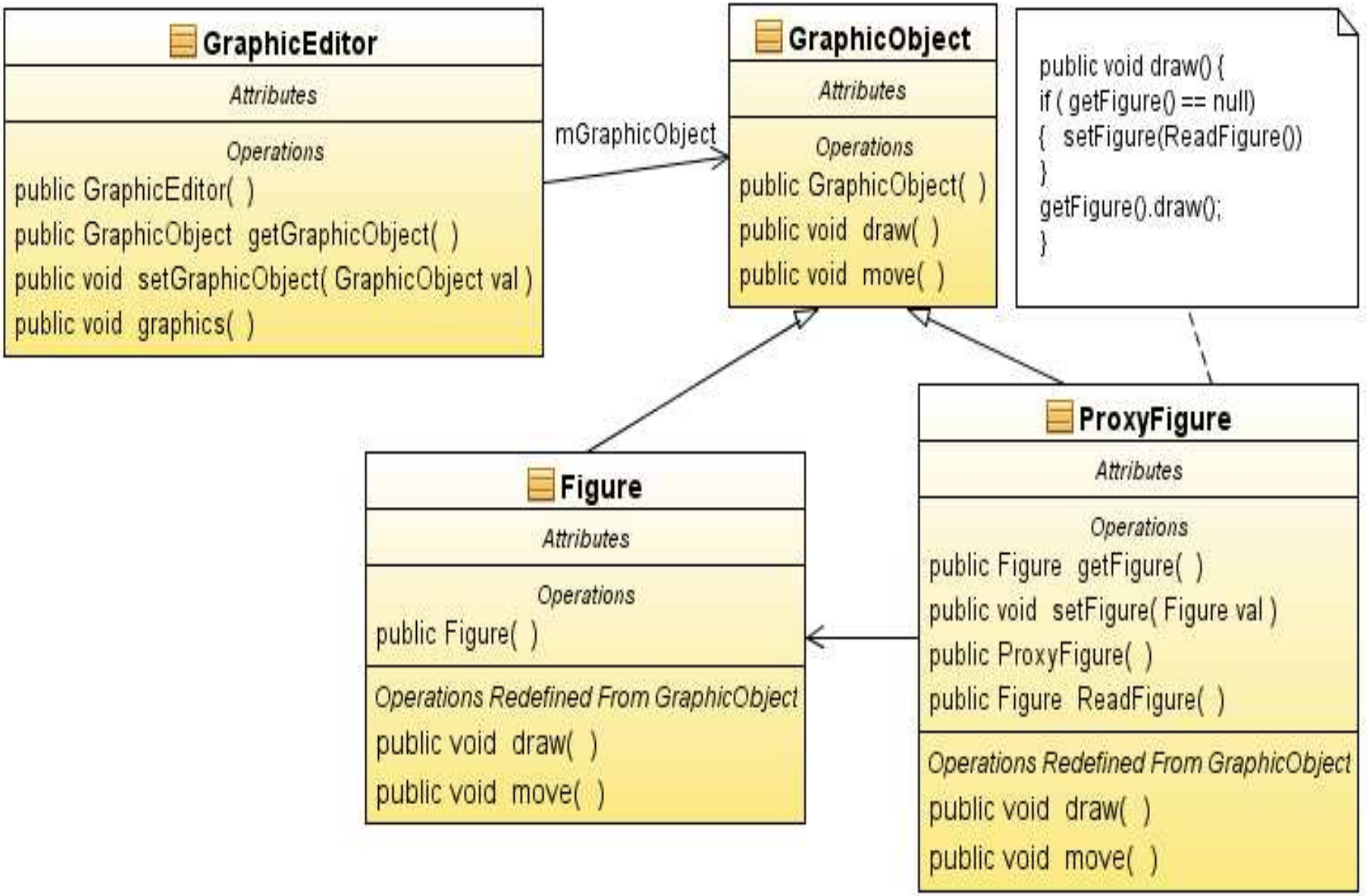
Charakterystyka wzorca Dekorator

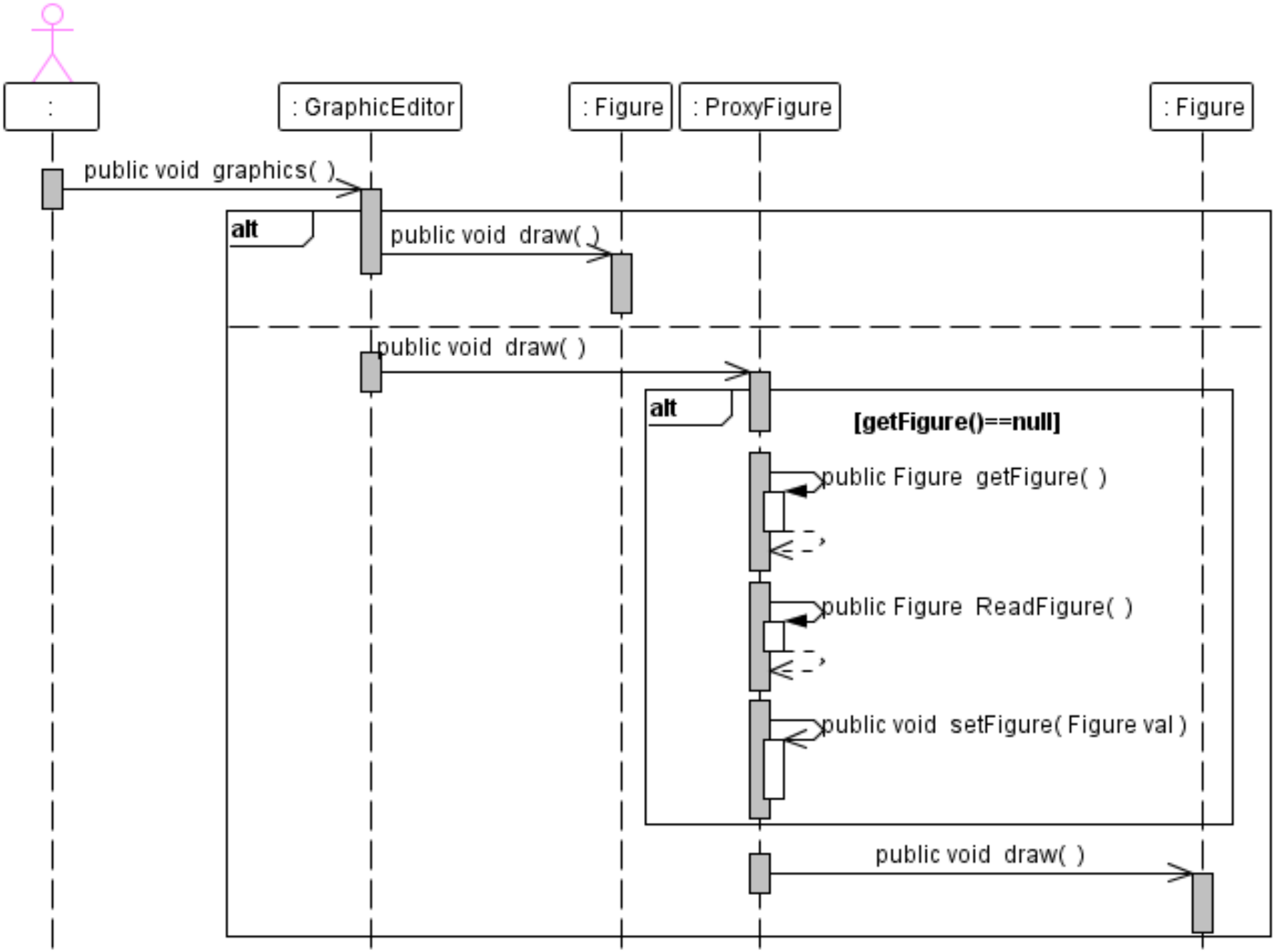
- **Problem**: Należy dynamicznie rozwijać funkcjonalność obiektu jako alternatywa dla tworzenia hierarchii klas.
- **Rozwiązanie**: Obiekt typu ***Component*** jest klasą abstrakcyjną (interfejsem) dla obiektów wizualnych. Jej interfejs definiuje operacje rysowania i obsługi zdarzeń implementowane przez klasę ***ConcreteComponent.*** Abstrakcyjna klasa (interfejs) typu ***Dekorator*** dziedziczy operacje wizualne od interfejsu ***Component*** i definiuje dodatkowe operacje graficzne realizowane przez klasę ***ConcreteDekorator.***
- **Klient wzorca**: Dokument zrealizowany z komponentów wizualnych bez dekoratorów i z dekoratorami
- **Rezultat**:
  - dynamiczne i przezroczyste dodawanie dodatkowych komponentów wizualnych do podstawowych komponentów wizualnych
  - Łatwe usuwanie dodatkowych funkcjonalności
  - Zastąpienie dekoratorami dodawanymi dynamicznie do klas rozbudowanej hierarchii klas zawierających na stałe równoważne funkcjonalności
- **Implementacja**: nowa klasa typu „ Boundry”, biblioteka komponentów typu Swing, biblioteka komponentów typu Java Server Faces

5) Pełnomocnik - *Proxy – wzorzec obiektowy*



Przykład


GraphicEditor
Attributes
Operations
public GraphicEditor( )
public GraphicObject getGraphicObject( )
public void setGraphicObject(GraphicObject val)
public void graphics( )
mGraphicObject
GraphicObject
Attributes
Operations
public GraphicObject( )
public void draw( )
public void move( )
public void draw() {
if (getFigure() == null)
{ setFigure(ReadFigure())
}
getFigure().draw();
}
Figure
Attributes
Operations
public Figure( )
Operations Redefined From GraphicObject
public void draw( )
public void move( )
ProxyFigure
Attributes
Operations
public Figure getFigure( )
public void setFigure(Figure val)
public ProxyFigure( )
public Figure ReadFigure( )
Operations Redefined From GraphicObject
public void draw( )
public void move( )

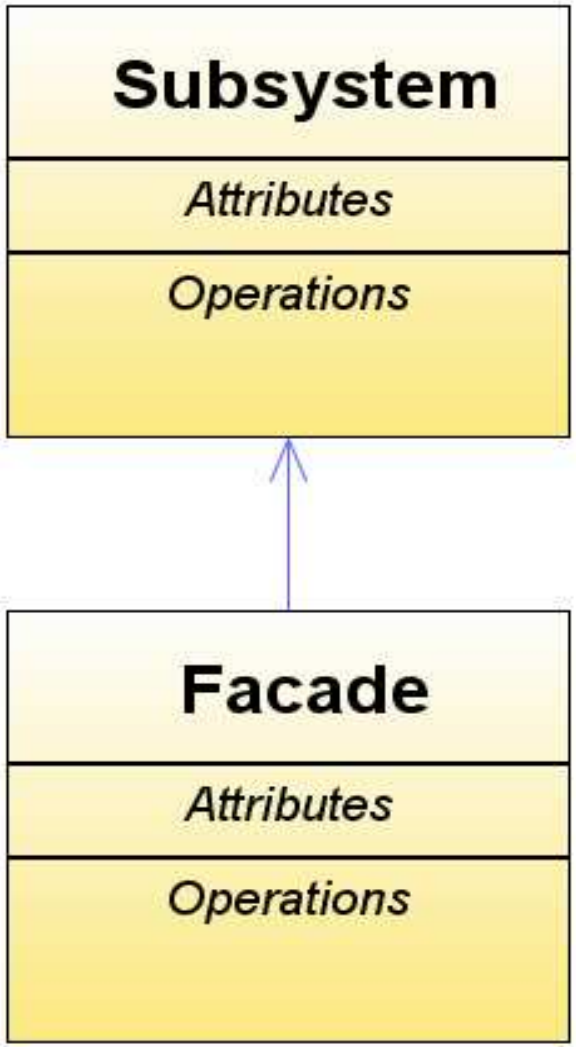

:
: GraphicEditor
: Figure
: ProxyFigure
: Figure
public void graphics( )
alt
public void draw( )
public void draw( )
alt
[getFigure()==null]
public Figure getFigure( )
public Figure ReadFigure( )
public void setFigure(Figure val)
public void draw( )

Charakterystyka wzorca Pełnomocnik

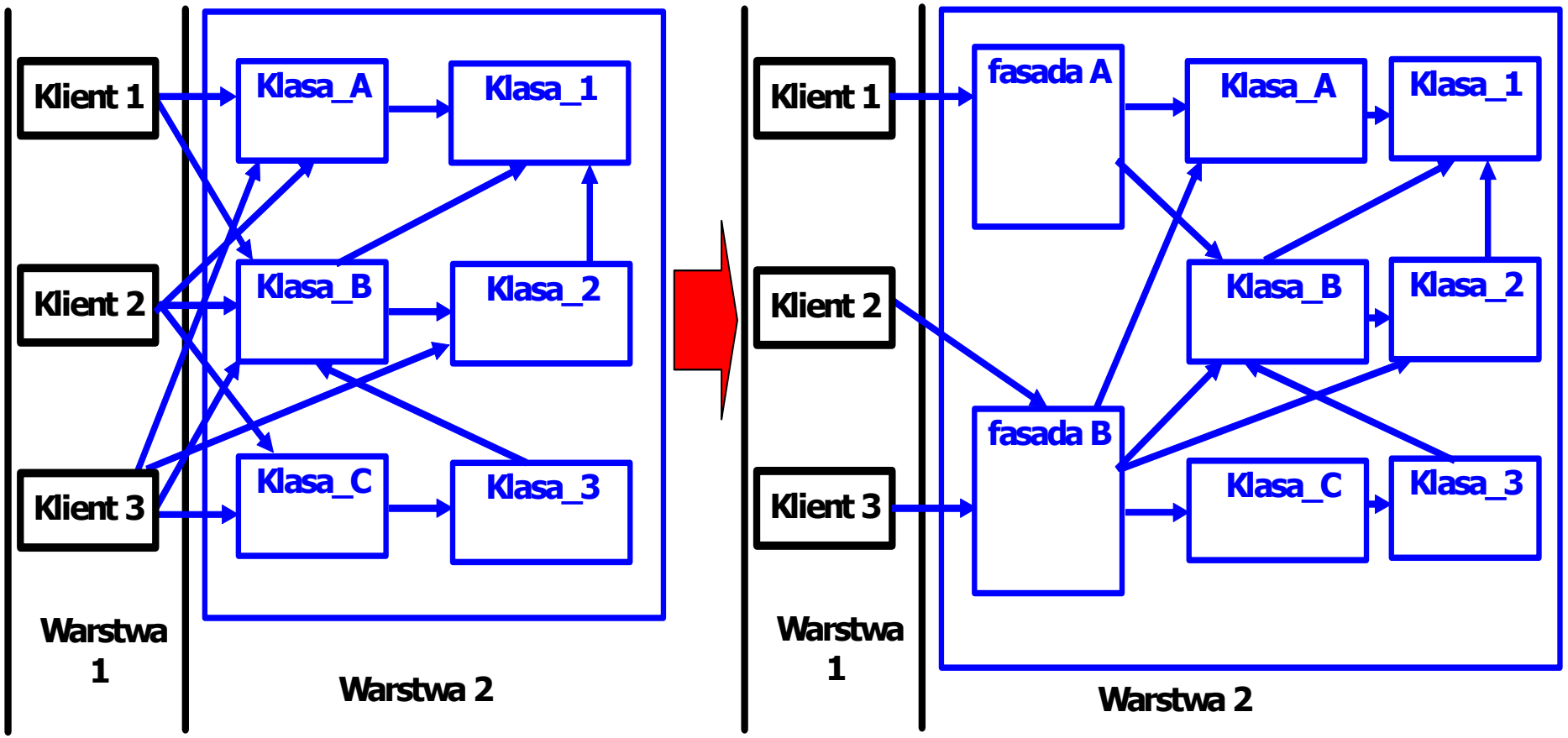
- **Problem**: reprezentuje inny obiekt w celu sterowania dostępem do niego
- **Rozwiązanie**: Obiekt typu ***Proxy*** przechowuje referencję do prawdziwego obiektu typu ***RealObject*** i może być zastąpiony przez obiekt typu ***RealObject***, ponieważ mają taki sam interfejs (dziedziczą od klasy typu ***Subject***) oraz może kontrolować dostęp do obiektu typu ***RealObject***.

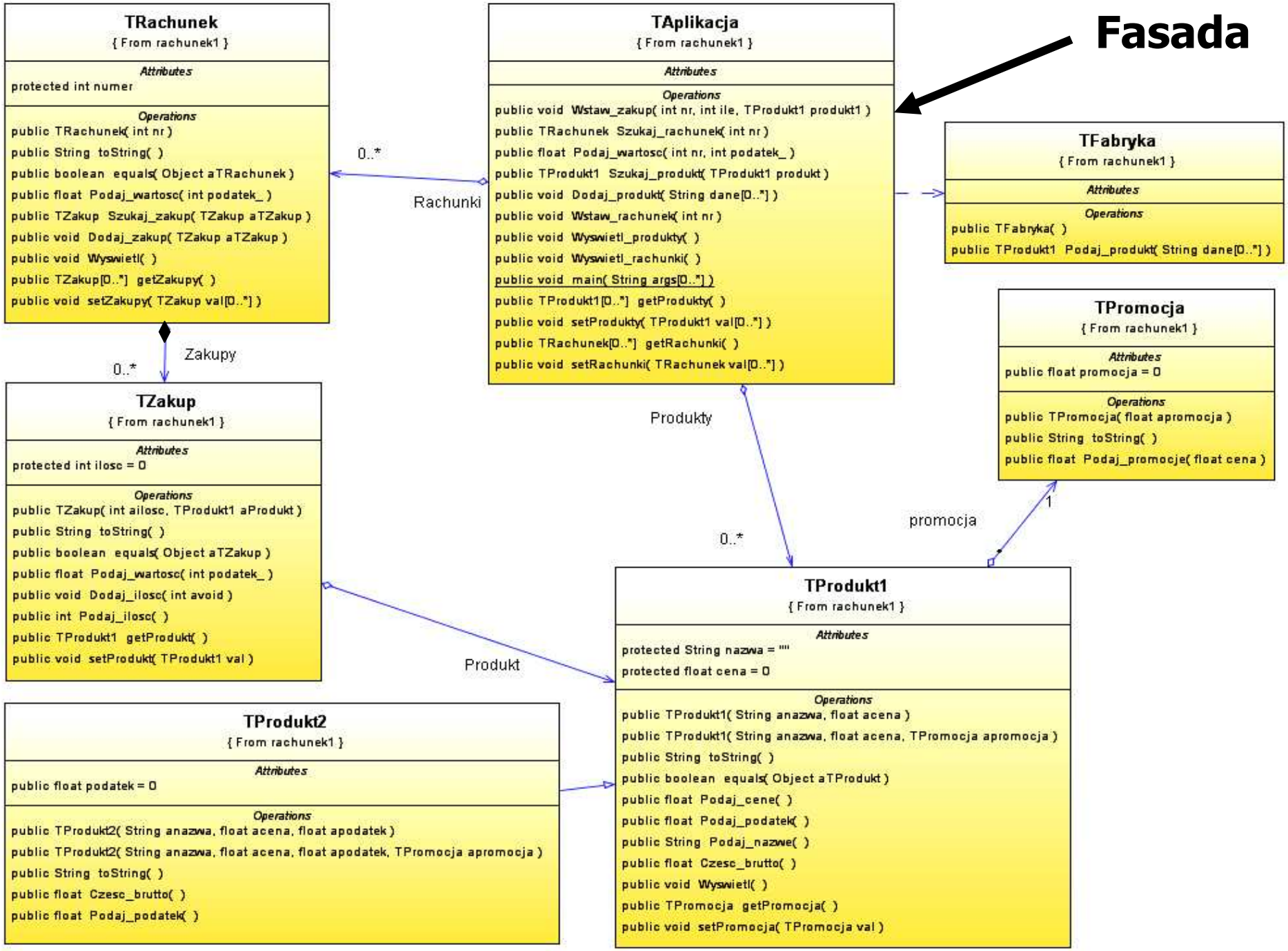
  Obiekt typu ***Proxy*** może być zdalnym obiektem odwołującym się do obiektu typu ***RealObject***, wirtualnym obiektem buforującym dostęp do obiektu typu ***RealObject***, obiektem zabezpieczającym przed dostępem nie powołanym do obiektu typu ***RealObject***
- **Klient wzorca**: Dowolny obiekt z dowolnej warstwy wielowarstwowego programu
- **Rezultat**:
  - **Zdalny *Proxy*** może ukrywać obiekty typu ***RealObject*** w dowolnej przestrzeni adresowej
  - **Wirtualny *Proxy*** poprawia wydajność za pomocą buforowania danych obiektu typu ***RealObject*** i ogranicza niepotrzebne operacje na tym obiekcie np. modyfikacje, zapisy do pliku
  - **Zabezpieczający** Proxy autoryzuje dostęp do obiektów typu ***RealObject***

6) Fasada – *Facade – wzorzec obiektowy*



Wzorzec fasady

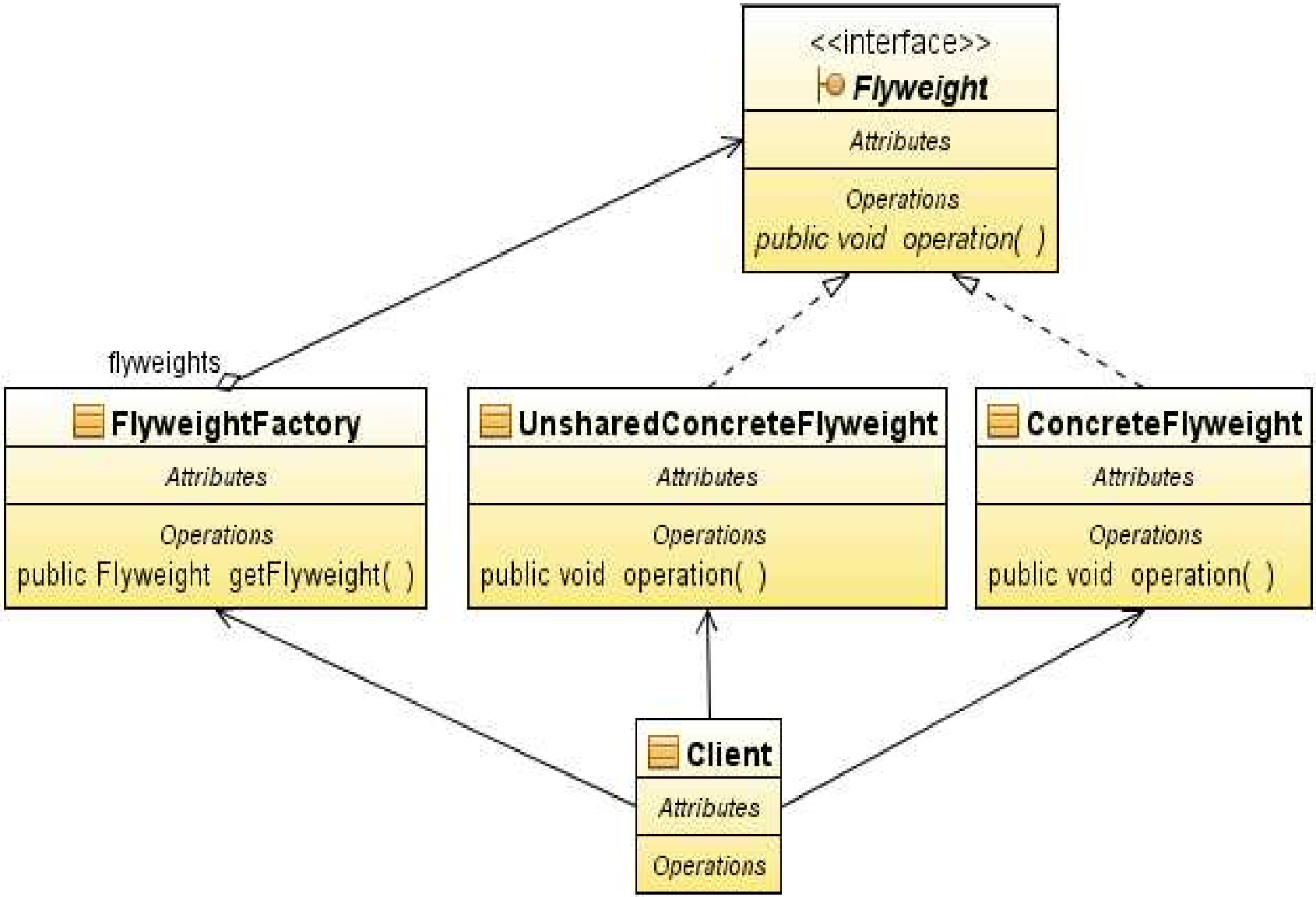


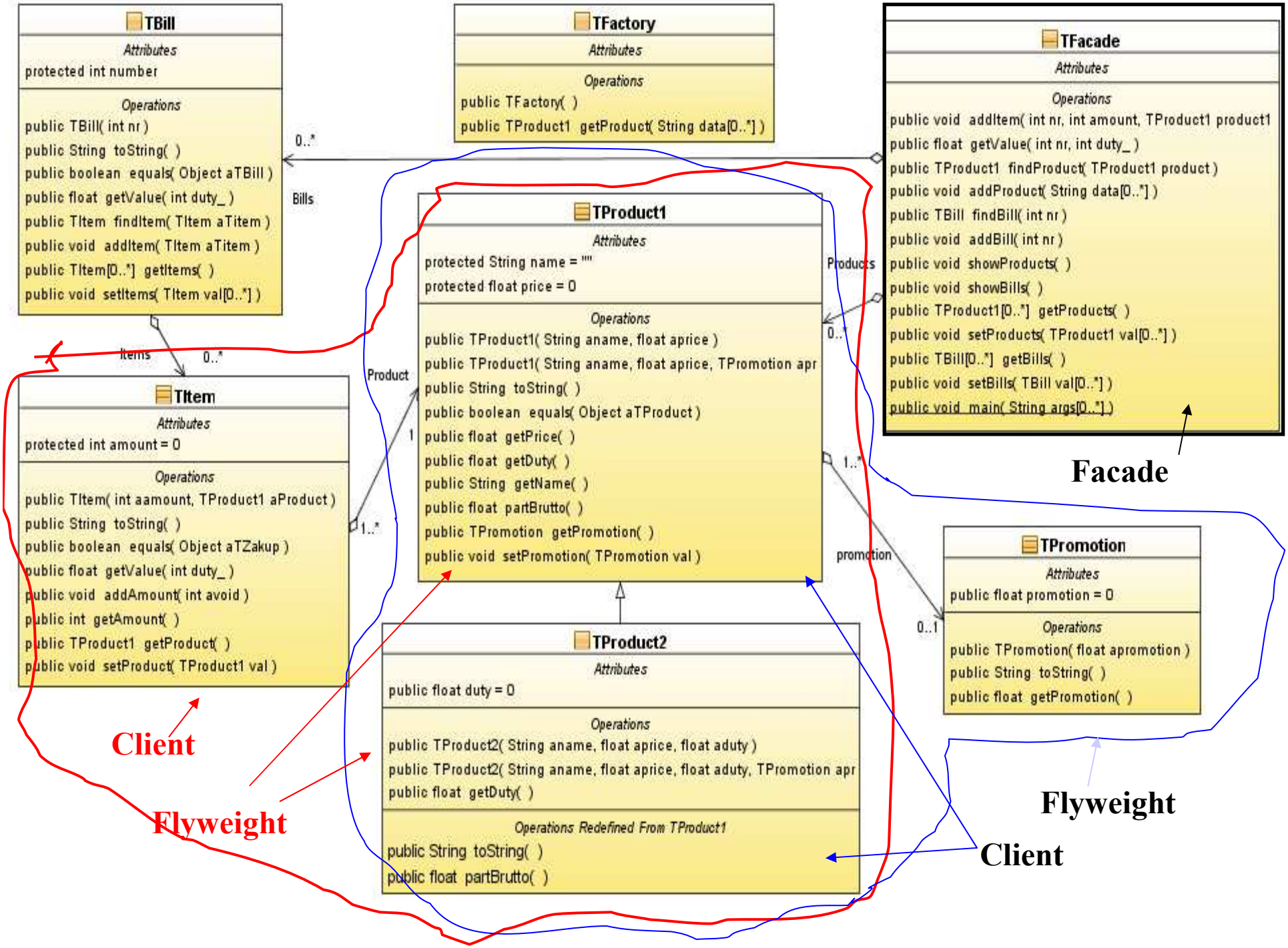

TRachunek
{ From rachunek1 }
Attributes
protected int numer
Operations
public TRachunek(int nr)
public String toString( )
public boolean equals(Object aTRachunek)
public float Podaj_wartosc(int podatek_)
public TZakup Szukaj_zakup(TZakup aTZakup)
public void Dodaj_zakup(TZakup aTZakup)
public void Wyswietl( )
public TZakup[0..*] getZakupy( )
public void setZakupy(TZakup val[0..*])
0..*
Rachunki
TAplikacja
{ From rachunek1 }
Attributes
Operations
public void Wstaw_zakup(int nr, int ile, TProdukt1 produkt1)
public TRachunek Szukaj_rachunek(int nr)
public float Podaj_wartosc(int nr, int podatek_)
public TProdukt1 Szukaj_produkt(TProdukt1 produkt)
public void Dodaj_produkt(String dane[0..*])
public void Wstaw_rachunek(int nr)
public void Wyswietl_produkty( )
public void Wyswietl_rachunki( )
public void main(String args[0..*])
public TProdukt1[0..*] getProdukty( )
public void setProdukty(TProdukt1 val[0..*])
public TRachunek[0..*] getRachunki( )
public void setRachunki(TRachunek val[0..*])
Fasada
TFabryka
{ From rachunek1 }
Attributes
Operations
public TFabryka( )
public TProdukt1 Podaj_produkt(String dane[0..*])
TPromocja
{ From rachunek1 }
Attributes
public float promocja = 0
Operations
public TPromocja(float apromocja)
public String toString( )
public float Podaj_promocje(float cena)
Zakupy
0..*
TZakup
{ From rachunek1 }
Attributes
protected int ilosc = 0
Operations
public TZakup(int ailosc, TProdukt1 aProdukt)
public String toString( )
public boolean equals(Object aTZakup)
public float Podaj_wartosc(int podatek_)
public void Dodaj_ilosc(int avoid)
public int Podaj_ilosc( )
public TProdukt1 getProdukt( )
public void setProdukt(TProdukt1 val)
Produkty
0..*
promocja
1
Produkt
TProdukt1
{ From rachunek1 }
Attributes
protected String nazwa = ""
protected float cena = 0
Operations
public TProdukt1(String anazwa, float acena)
public TProdukt1(String anazwa, float acena, TPromocja apromocja)
public String toString( )
public boolean equals(Object aTProdukt)
public float Podaj_cene( )
public float Podaj_podatek( )
public String Podaj_nazwe( )
public float Czesc_brutto( )
public void Wyswietl( )
public TPromocja getPromocja( )
public void setPromocja(TPromocja val)
TProdukt2
{ From rachunek1 }
Attributes
public float podatek = 0
Operations
public TProdukt2(String anazwa, float acena, float apodatek)
public TProdukt2(String anazwa, float acena, float apodatek, TPromocja apromocja)
public String toString( )
public float Czesc_brutto( )
public float Podaj_podatek( )

Charakterystyka wzorca fasady

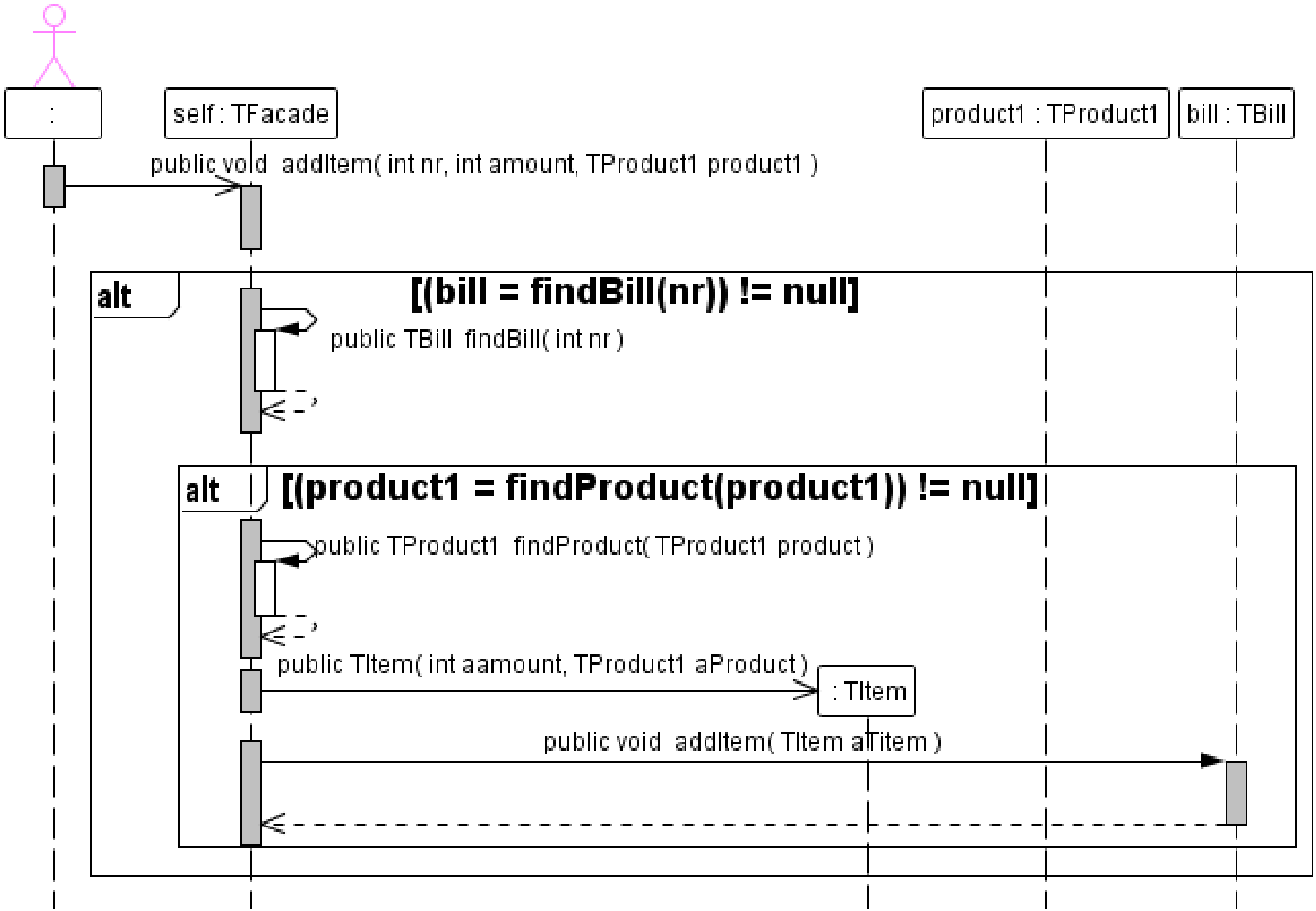
- **Problem**: udostępnienie tylko wybranych funkcji warstwy systemu
- **Rozwiązanie**: Stanowi interfejs lub interfejsy warstwy systemu- kilka fasad grupuje metody dla wybranych podsystemów
- **Klient wzorca**: otrzymuje jedynie potrzebne metody
- **Rezultat**:
  - Udostępnienie istotnych metod warstwy systemu np. reprezentujących przypadki użycia – hermetyzacja klas warstwy systemu
  - Fasada może uniemożliwiać dostęp do wszystkich metod hermetyzowanych klas
- **Implementacja**: nowa klasa typu „Control” np. SessionBean1, ApplicationBean1

7) Pyłek - *Flyweight – wzorzec obiektowy*

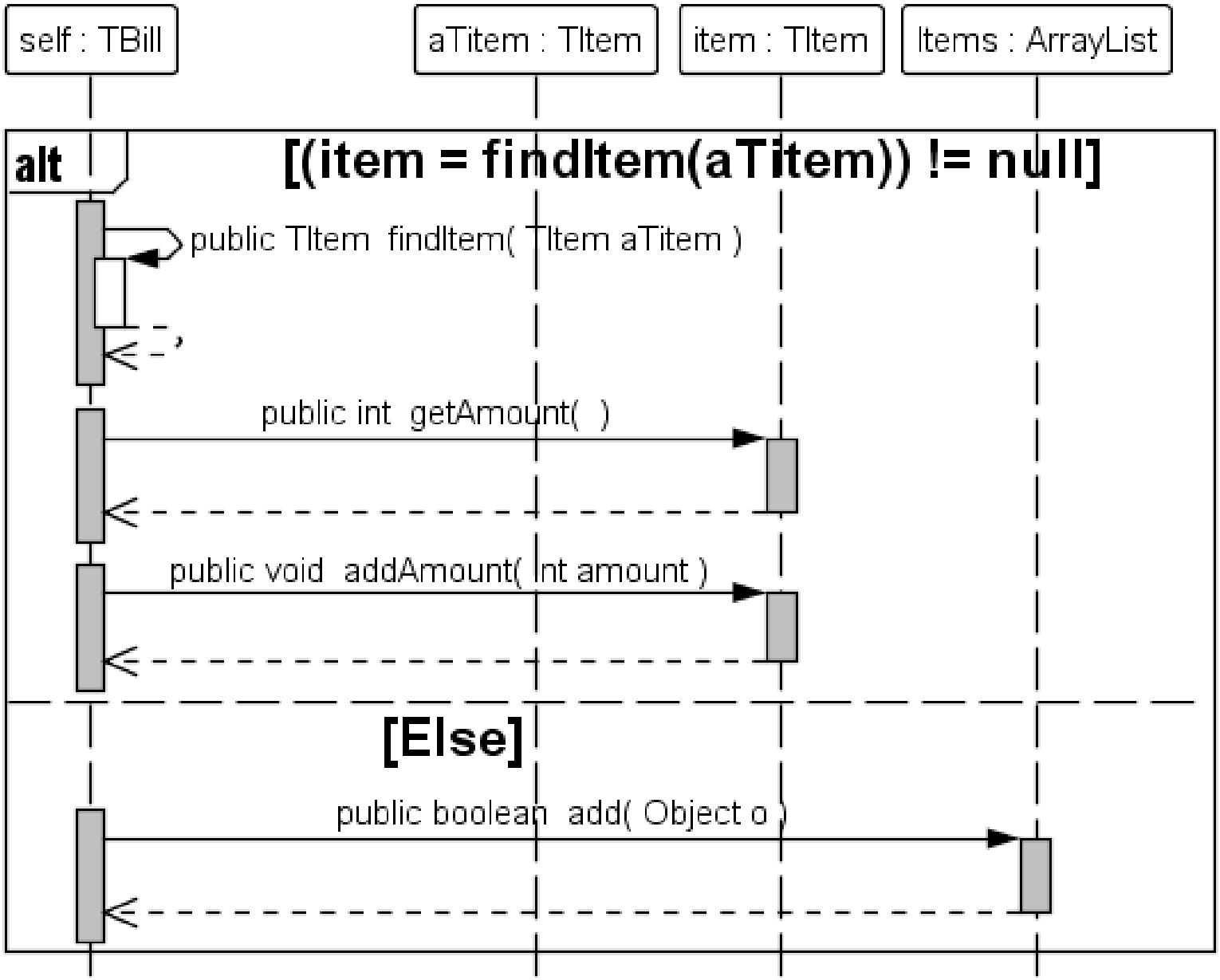



TBill
Attributes
protected int number
Operations
public TBill(int nr)
public String toString( )
public boolean equals(Object aTBill)
public float getValue(int duty_)
public TItem findItem(TItem aTitem)
public void addItem(TItem aTitem)
public TItem[0..*] getItems( )
public void setItems(TItem val[0..*])
TFactory
Attributes
Operations
public TFactory( )
public TProduct1 getProduct(String data[0..*])
TFacade
Attributes
Operations
public void addItem(int nr, int amount, TProduct1 product1
public float getValue(int nr, int duty_)
public TProduct1 findProduct(TProduct1 product)
public void addProduct(String data[0..*])
public TBill findBill(int nr)
public void addBill(int nr)
public void showProducts( )
public void showBills( )
public TProduct1[0..*] getProducts( )
public void setProducts(TProduct1 val[0..*])
public TBill[0..*] getBills( )
public void setBills(TBill val[0..*])
public void main(String args[0..*])
0..*
Bills
Items
0..*
TProduct1
Attributes
protected String name = ""
protected float price = 0
Operations
public TProduct1(String aname, float aprice)
public TProduct1(String aname, float aprice, TPromotion apr
public String toString( )
public boolean equals(Object aTProduct)
public float getPrice( )
public float getDuty( )
public String getName( )
public float partBrutto( )
public TPromotion getPromotion( )
public void setPromotion(TPromotion val)
Products
0..*
Product
1
TItem
Attributes
protected int amount = 0
Operations
public TItem(int aamount, TProduct1 aProduct)
public String toString( )
public boolean equals(Object aTZakup)
public float getValue(int duty_)
public void addAmount(int avoid)
public int getAmount( )
public TProduct1 getProduct( )
public void setProduct(TProduct1 val)
1..*
1..*
promotion
0..1
TPromotion
Attributes
public float promotion = 0
Operations
public TPromotion(float apromotion)
public String toString( )
public float getPromotion( )
TProduct2
Attributes
public float duty = 0
Operations
public TProduct2(String aname, float aprice, float aduty)
public TProduct2(String aname, float aprice, float aduty, TPromotion apr
public float getDuty( )
Operations Redefined From TProduct1
public String toString( )
public float partBrutto( )
Facade
Client
Flyweight
Flyweight
Client

Flyweight, Facade



Flyweight


self : TBill
aTitem : TItem
item : TItem
Items : ArrayList
alt
[(item = findItem(aTitem)) != null]
public TItem findItem(TItem aTitem)
public int getAmount( )
public void addAmount(int amount)
[Else]
public boolean add(Object o)

Charakterystyka wzorca Pyłek

- **Problem**: Wielokrotne wykorzystanie tego samego obiektu – współdzielenie obiektów
- **Rozwiązanie**: Obiekt typu ***Flyweight*** definiuje interfejs obiektów typu ***ConcreteFlyweight*** (współdzielone użycie) oraz typu ***UnsharedConcreteFlyweight*** (użyty jednorazowo) używane przez klientów aplikacji. Obiekty- pyłki są tworzone i zarządzane przez obiekt typu ***FlyweightFactory***
- **Klient wzorca**: przechowuje odwołania do obiektów-pyłków
- **Rezultat**:
  - Oszczędność pamięci przez współdzielenie obiektów- pyłków
- **Implementacja**: nowe klasy typu „Boundry” lub „Entity” np.
  - referencja tego samego obiektu z rodziny TProdukt1 (pyłek) może być przechowywana w wielu obiektach typu TZakup (klient);
  - Referencja obiektu TPromocja (pyłek) może być przechowywana przez jeden z obiektów z rodziny TProdukt1 (klient)