Wzorce projektowe
Wykład 5 – część 2

Zofia Kruczkiewicz

Wzorce projektowe

1. Identyfikacja wzorców projektowych
2. Przegląd wzorców projektowych
   2.1. Wzorce kreacyjne
   2.2. Wzorce strukturalne
   2.3. Wzorce zachowania
3. Dodatek - uzupełnienie

Wzorce projektowe

1. **Identyfikacja wzorców projektowych**

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)**

Wzorce projektowe

1. **Identyfikacja wzorców projektowych**
2. **Przegląd wzorców projektowych**

Gang of Four – skrót odnoszący się do autorów ksiązki:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*

Wzorce projektowe

1. **Identyfikacja wzorców projektowych**
2. **Przegląd wzorców projektowych**

   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*

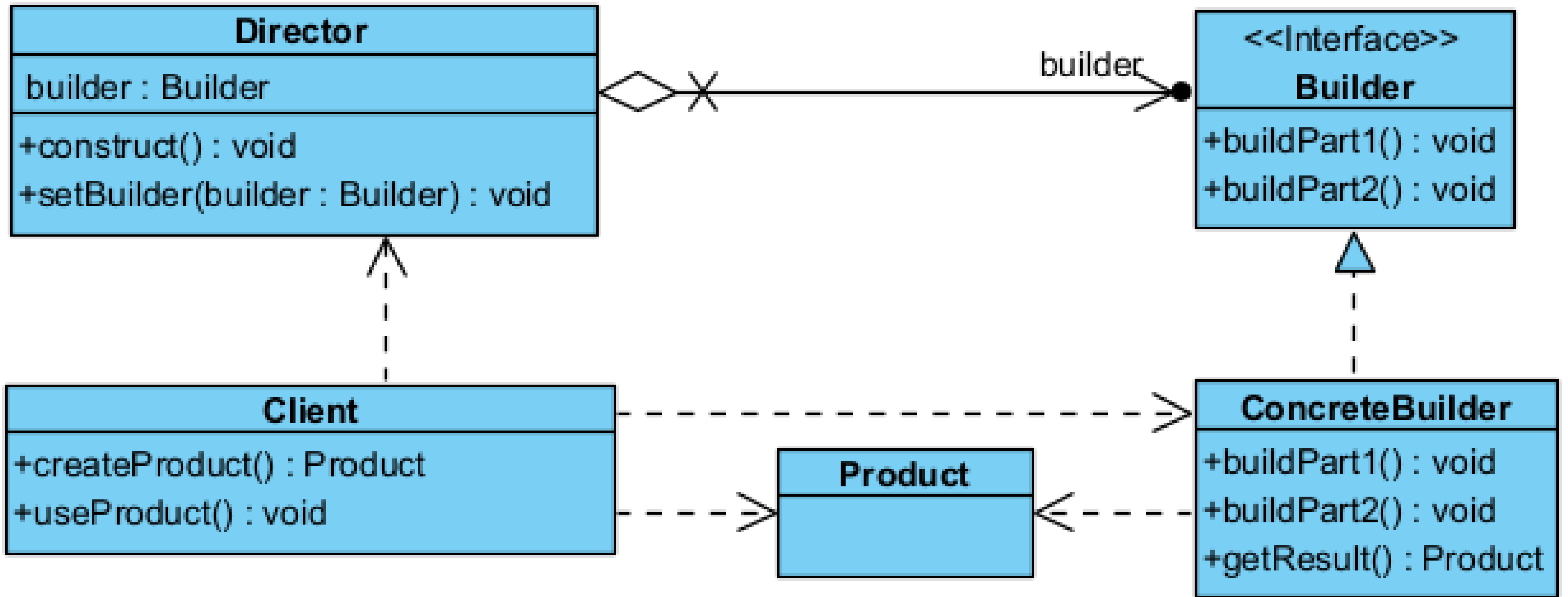
   2.1. Wzorce kreacyjne

Wzorce kreacyjne - wybór

Cel stosowania: 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"

Wzorzec projektowy	Kontekst zastosowania
1)Builder	**Sposób tworzenia złożonych obiektów**
2)Abstract Factory	**Rodziny obiektów**
3)Factory Method	Podklasa tworzonego obiektu
4)Prototype	Typ klasy tworzonego obiektu
5)Singleton	Jedna kopia obiektu

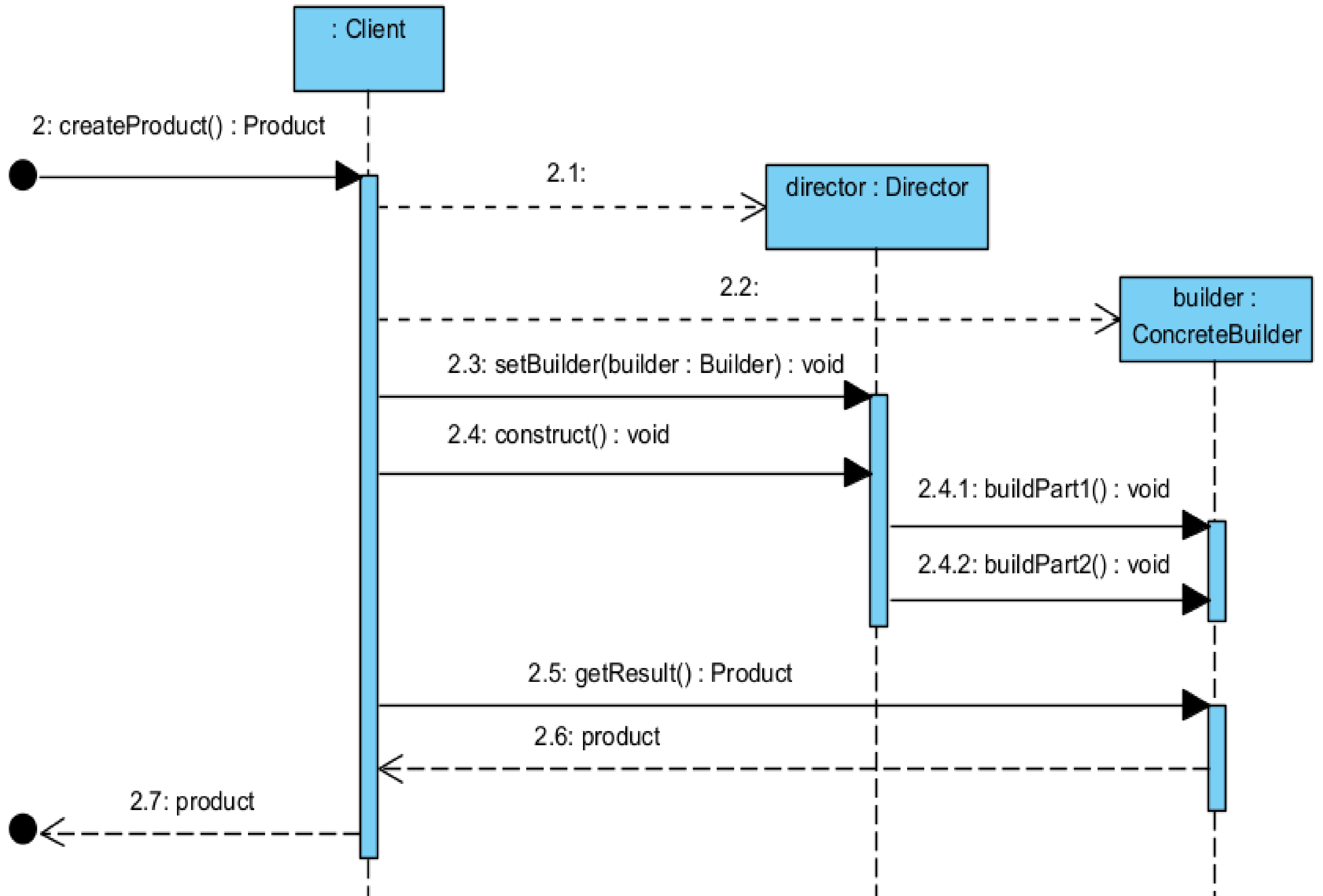
1) Budowniczy - *Builder*



Kod klasy używającej obiekt typu ***Director*** 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
2: createProduct() : Product
2.1:
director : Director
2.2:
builder : ConcreteBuilder
2.3: setBuilder(builder : Builder) : void
2.4: construct() : void
2.4.1: buildPart1() : void
2.4.2: buildPart2() : void
2.5: getResult() : Product
2.6: product
2.7: product

Charakterystyka wzorca *Builder*

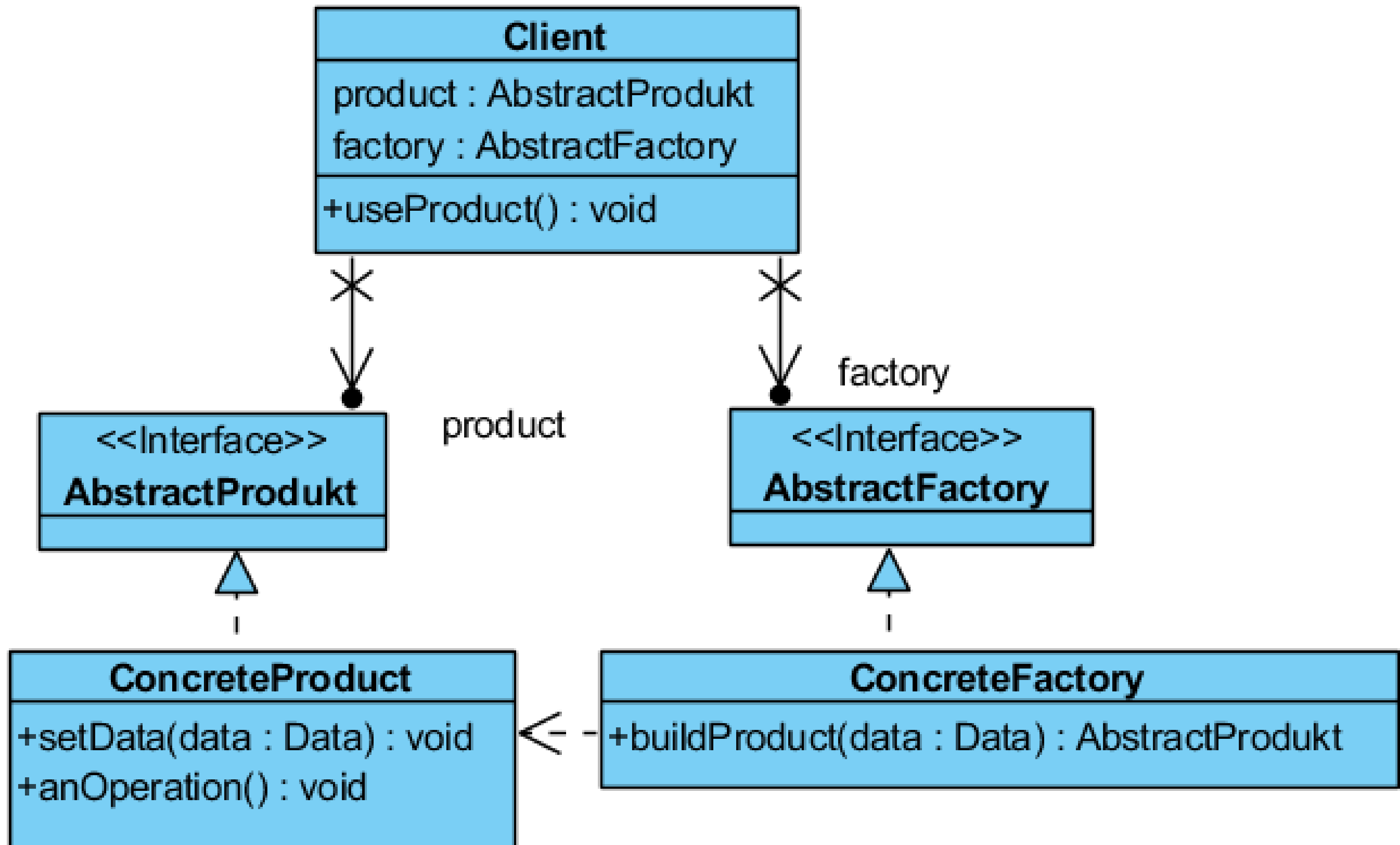
- **Problem**: Tworzenie dowolnych obiektów złożonych reprezentowanych w zróżnicowany sposób oraz oddzielenie konstrukcji obiektów złożonych od ich reprezentacji.

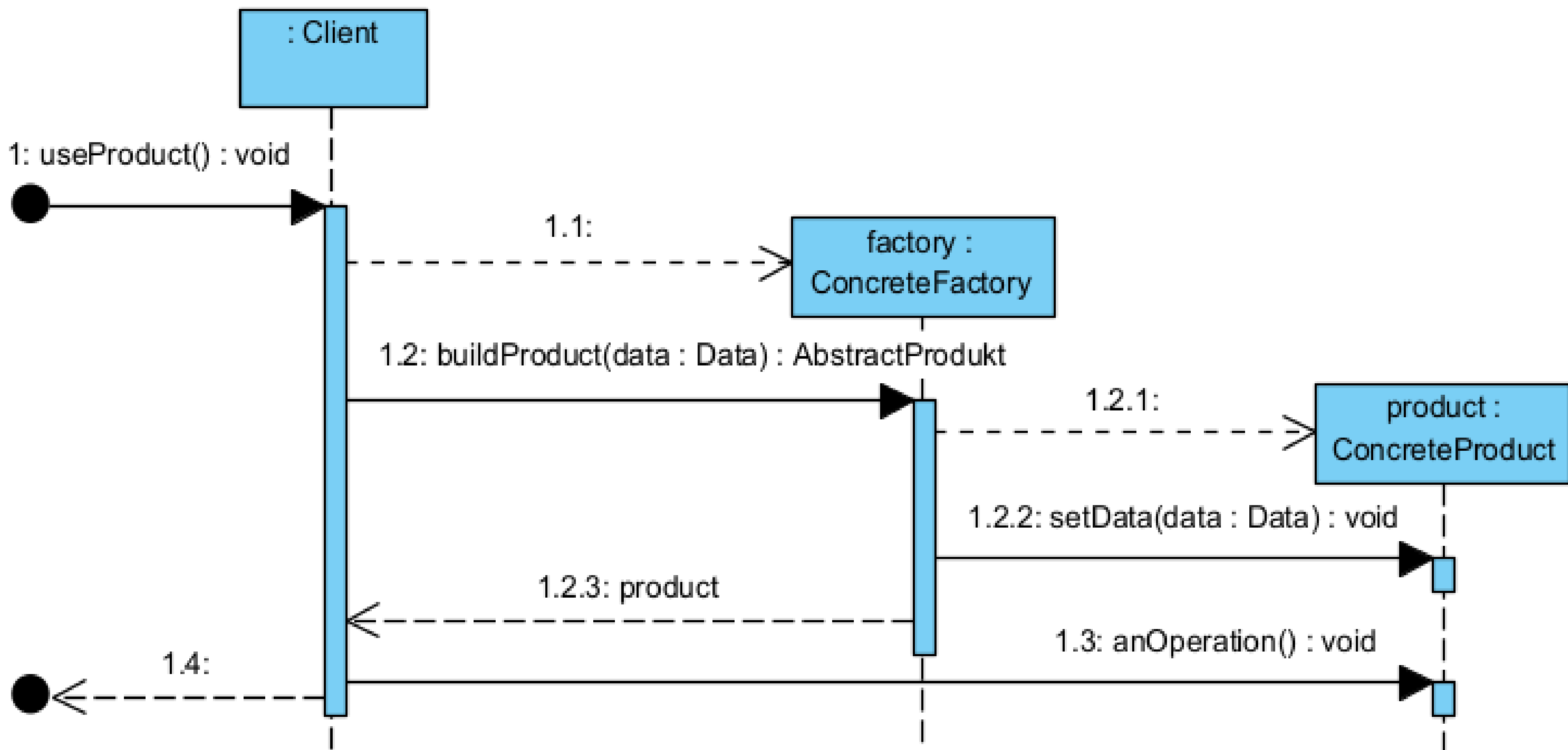
- **Rozwiązanie**: Obiekt typu ***Director*** zleca budowę obiektu typu ***Product*** obiektom typu ***ConcreteBuilder*** implementujących interfejs ***Builder***

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

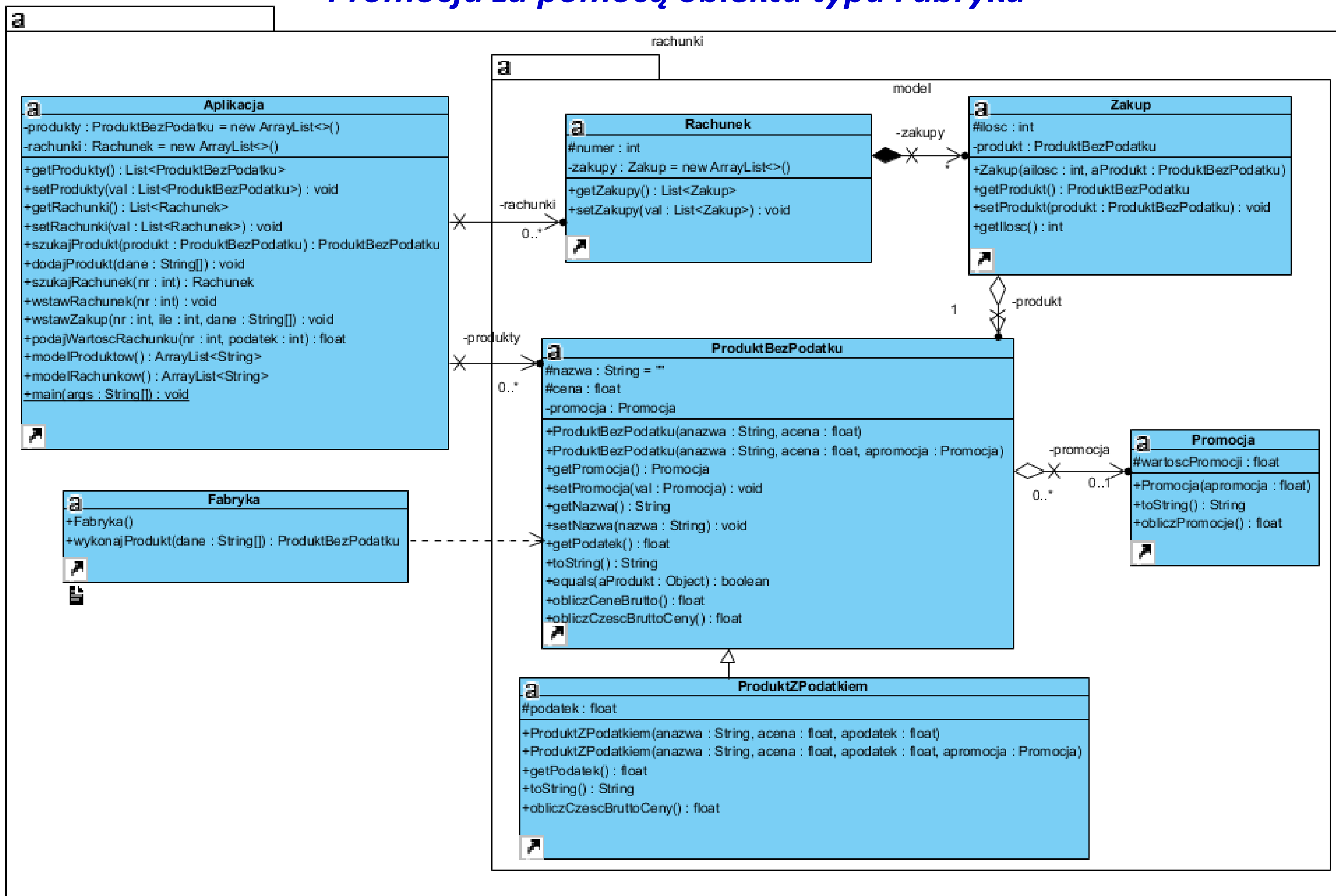
- **Rezultat**:
  - Udostępniony obiektom typu ***Director*** abstrakcyjny interfejs ***Builder*** pozwala na dowolny sposób konstruowania obiektów typu ***Product***
  - Oddzielenie kodu służącego do konstruowania obiektów typu ***Product*** od kodu służącego do używania tych obiektów np. obiekty typu ***Director*** utworzone do odczytu dokumentów w formacie RTF nie wpływają na kod klienta wzorca (***Client***), 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***
- **Pokrewne wzorce: Abstract Factory**

2) Fabryka abstrakcyjna – *Abstract Factory*

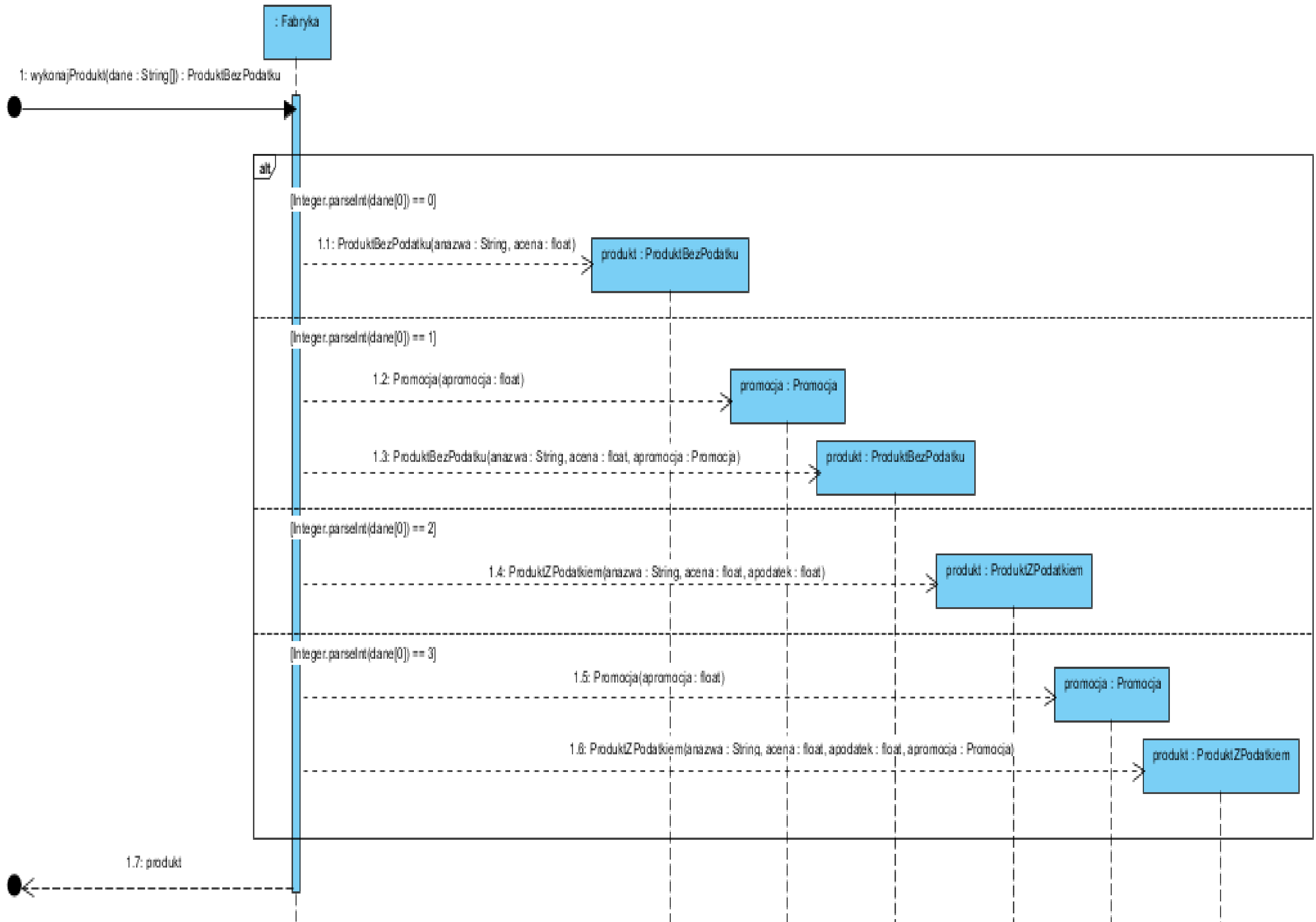



: Client
1: useProduct() : void
1.1:
factory :
ConcreteFactory
1.2: buildProduct(data : Data) : AbstractProdukt
1.2.1:
product :
ConcreteProduct
1.2.2: setData(data : Data) : void
1.2.3: product
1.3: anOperation() : void
1.4:

Przykład zastosowanie wzorca *Abstract Factory* – *tworzenie obiektów typu ProduktBezPodatku, ProduktZPodatkiem z uwzględnieniem obiektów typu Promocja za pomocą obiektu typu Fabryka*



(2) ProduktBez Podatku wykonajProdukt(String dane[])



```
public class Fabryka                    //Fabryka -decyzje na poziomie tworzenia kodu
 { public Fabryka() { }
   public ProduktBezPodatku wykonajProdukt(String dane[])
   { ProduktBezPodatku produkt = null;
     Promocja promocja;
     switch ( Integer.parseInt(dane[0]) )
    { case 0: produkt= new ProduktBezPodatku(dane[1], Float.parseFloat(dane[2]));
             break;
     case 1: promocja = new Promocja(Float.parseFloat(dane[3]));
              produkt = new ProduktBezPodatku (dane[1],
                                   Float.parseFloat(dane[2]),promocja);
             break;
     case 2: produkt = new ProduktZPodatkiem (dane[1], Float.parseFloat(dane[2]),
                                   Float.parseFloat(dane[3]));
              break;
     case 3: promocja = new Promocja(Float.parseFloat(dane[4]));
             produkt= new ProduktZPodatkiem(dane[1], Float.parseFloat(dane[2]),
                                   Float.parseFloat(dane[3]),promocja);
             break;
     }
     return produkt; }
}
```

Charakterystyka wzorca *Abstract Factory*

- **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. ***TitleBook*** i ***TitleBookRead***
  - System używa tworzone obiekty znając klasy bazowe tych obiektów i klasy bazowe fabryk
  - Utrudnione dołączanie nowych typów tworzonych obiektów.
- **Implementacja**: Zdefiniowanie klas typu „Control ”. Do wyboru obiektów można stosować pliki konfiguracyjne, tabele baz danych itp.
- **Pokrewne wzorce**: **Factory Method**, **Prototype**, **Singleton**

Wzorce projektowe

1. **Identyfikacja wzorców projektowych**
2. **Przegląd wzorców projektowych**

   Gang of Four – skrót odnoszący się do autorów ksiązki:

   Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*

   2.1. Wzorce kreacyjne

   2.2. Wzorce strukturalne

Wzorce strukturalne – tworzenie złożonych obiektowych struktur danych

- **Cel:** Składanie klas i obiektów w większe struktury

- **Wzorce klasowe:**

  zastosowanie dziedziczenia i polimorfizmu do składania struktur interfejsów lub ich implementacji

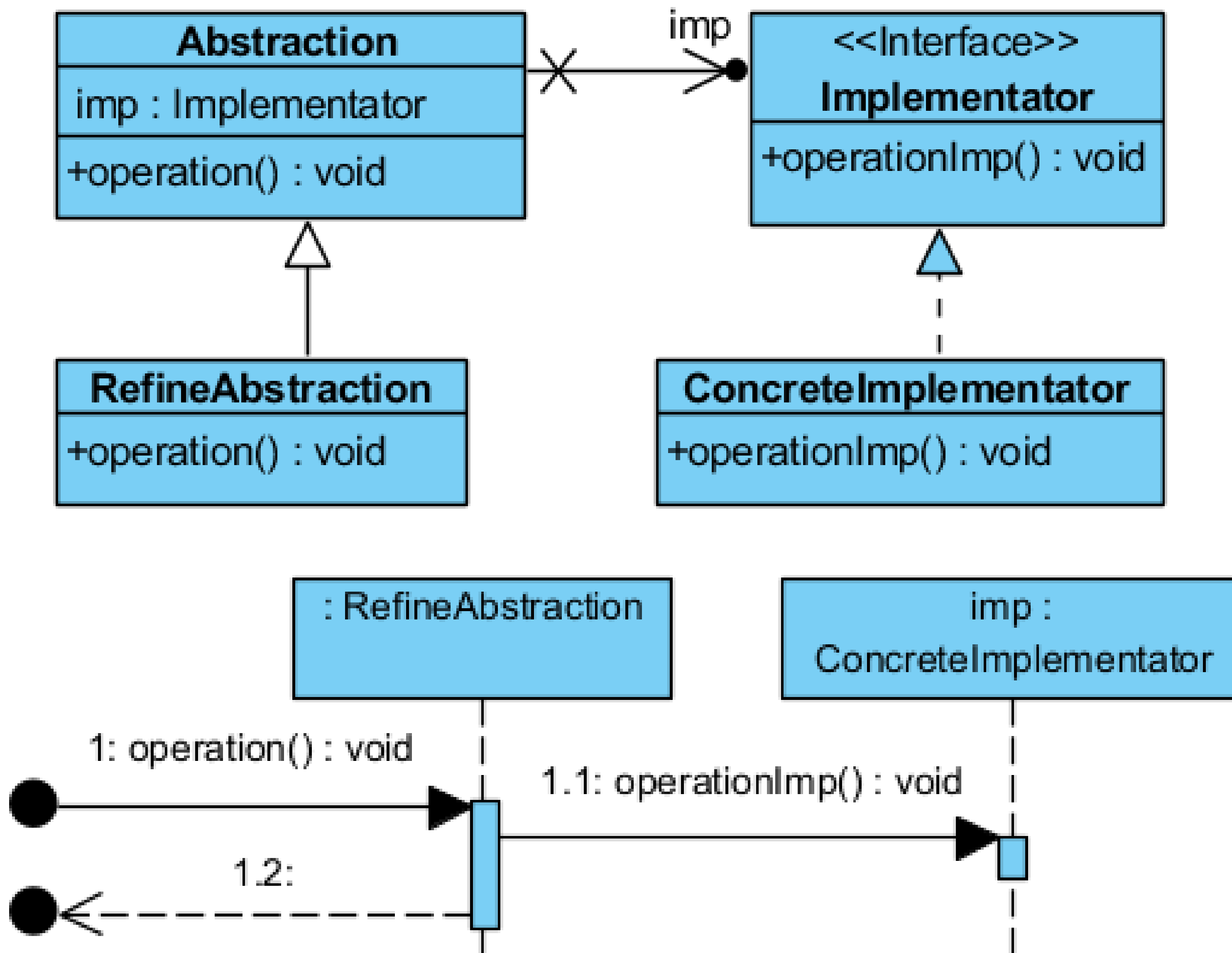
- **Wzorce obiektowe:**

  opisują sposoby składania obiektów za pomocą powiązań w celu uzyskania nowej funkcjonalności. Istnieje możliwość składania obiektów podczas działania programu

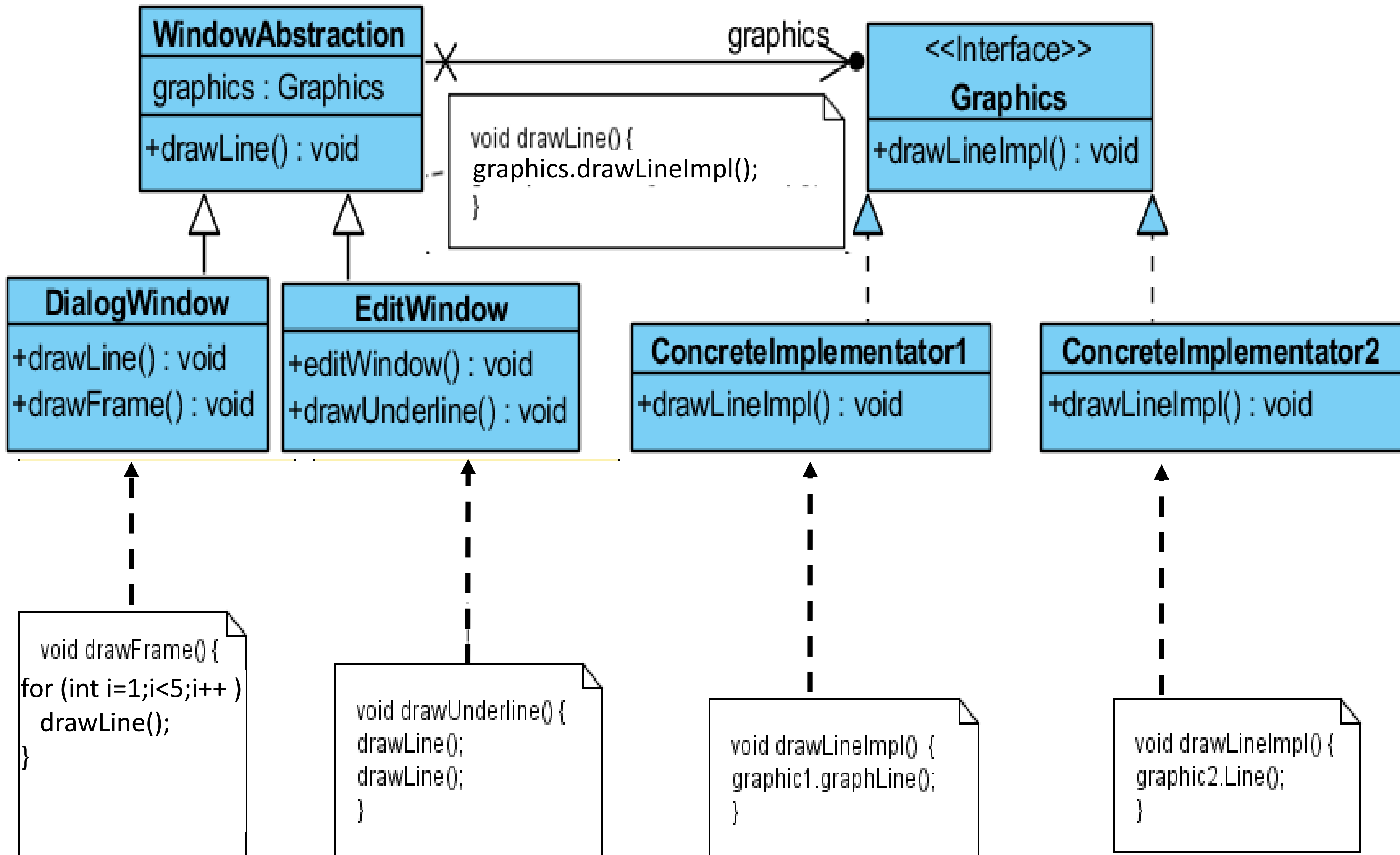
Wzorce strukturalne - wybór

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 podsystemu**
6) Flyweight - obiektowy	**Koszt przechowywania obiektów w pamięci**
7)Proxy - obiektowy	Sposób dostępu oraz położenie obiektu

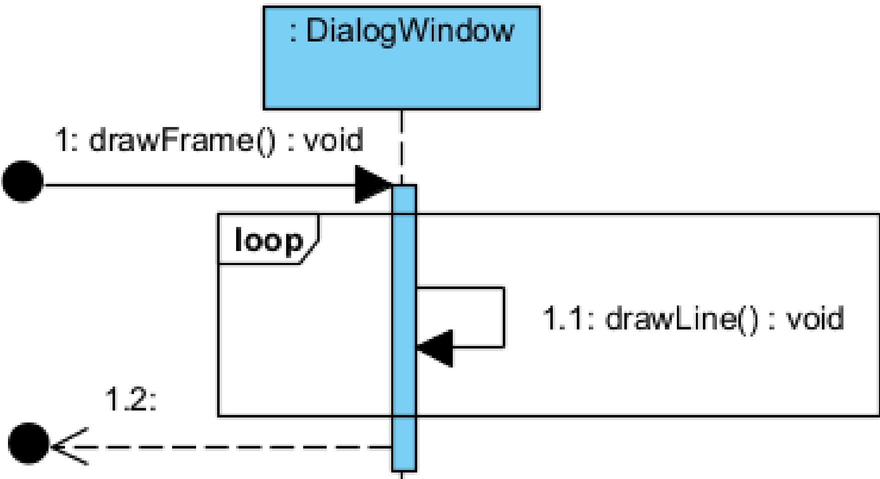
2) Most - *Bridge – wzorzec obiektowy*

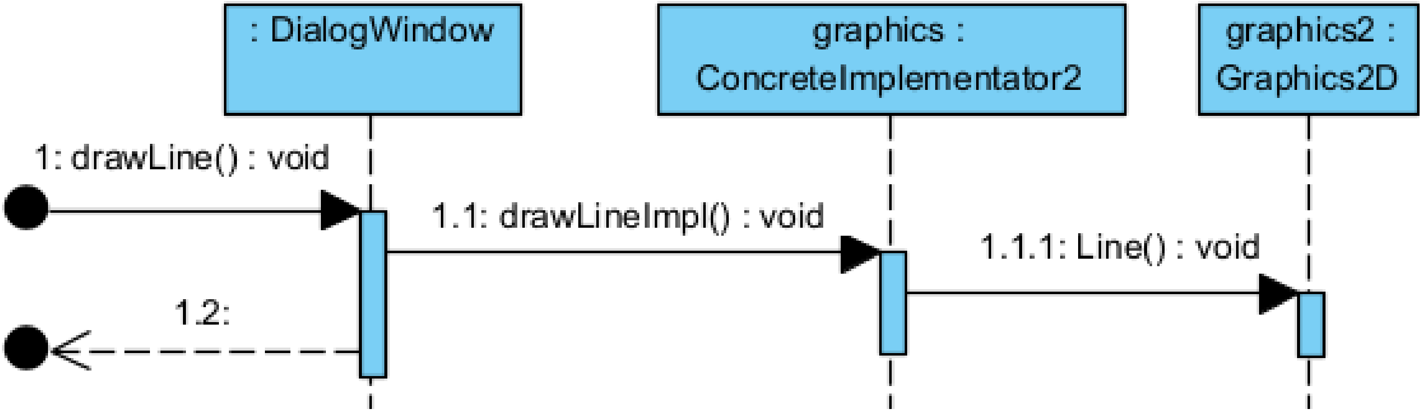


Przykład

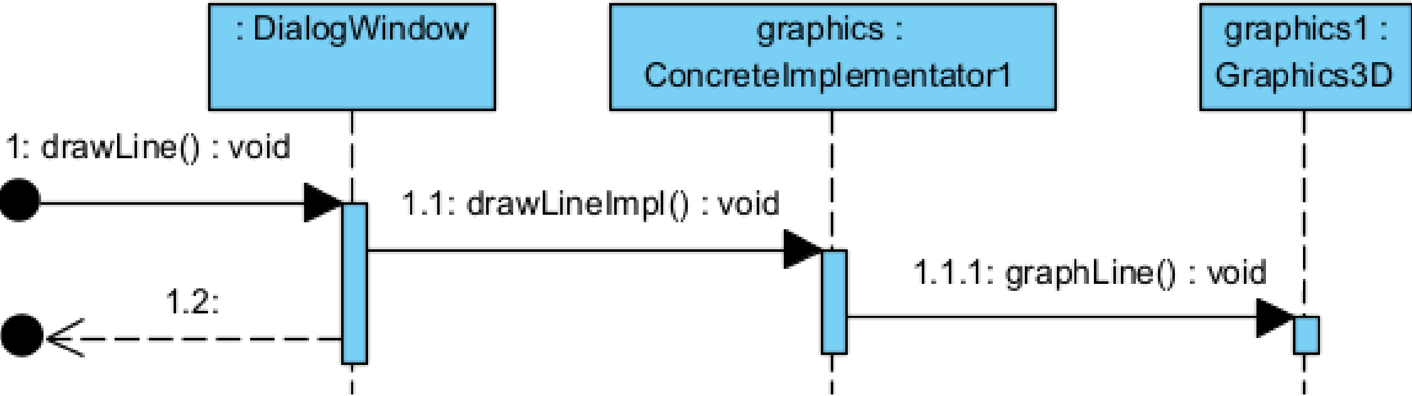


Przykład (cd)





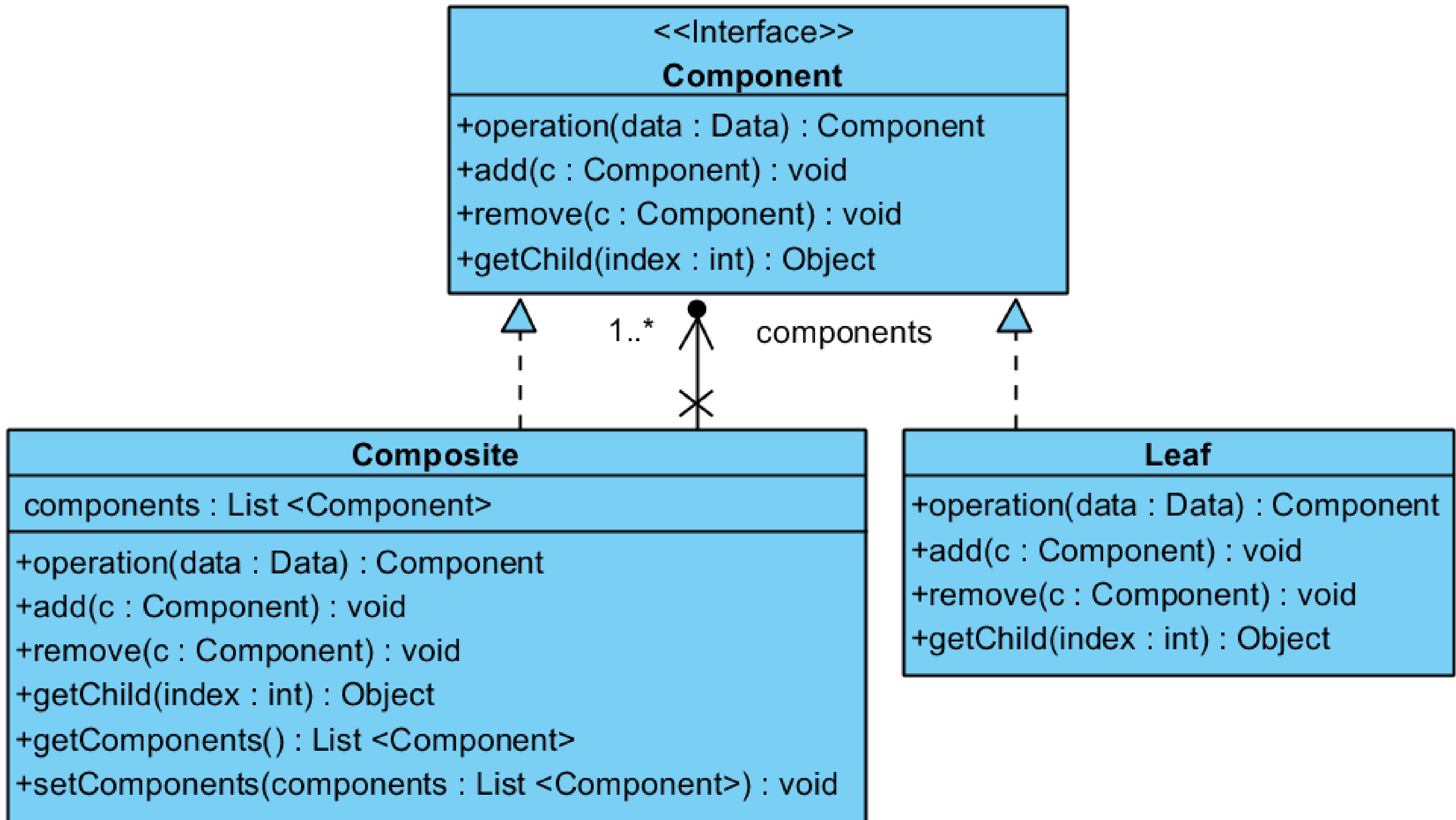
lub

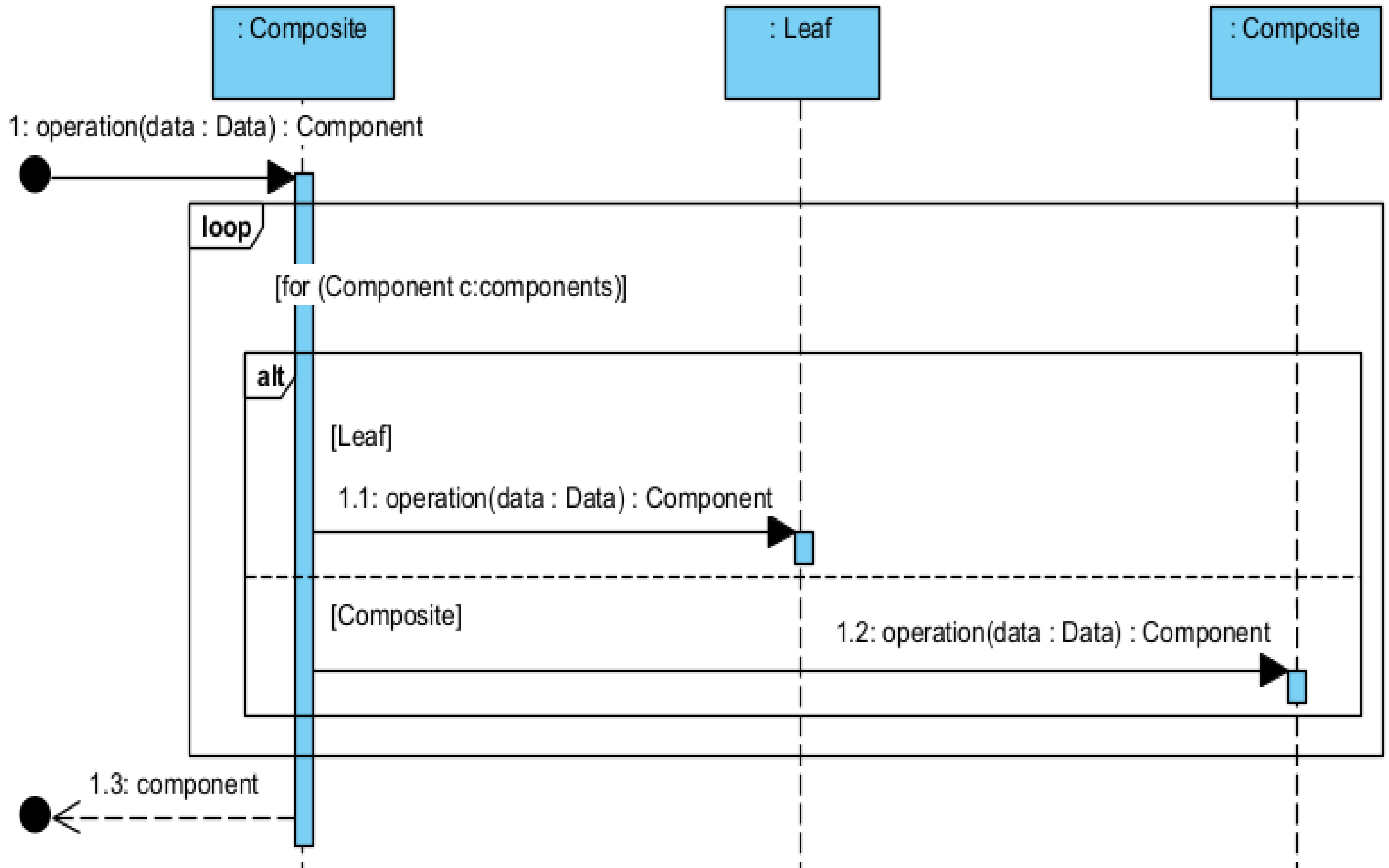


Charakterystyka wzorca *Bridge*

- **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
- **Implementacja**: nowa klasa typu „Boundry”
- **Pokrewne wzorce : Abstract Factory** tworzy i konfiguruje **Bridge**; **Adapter**

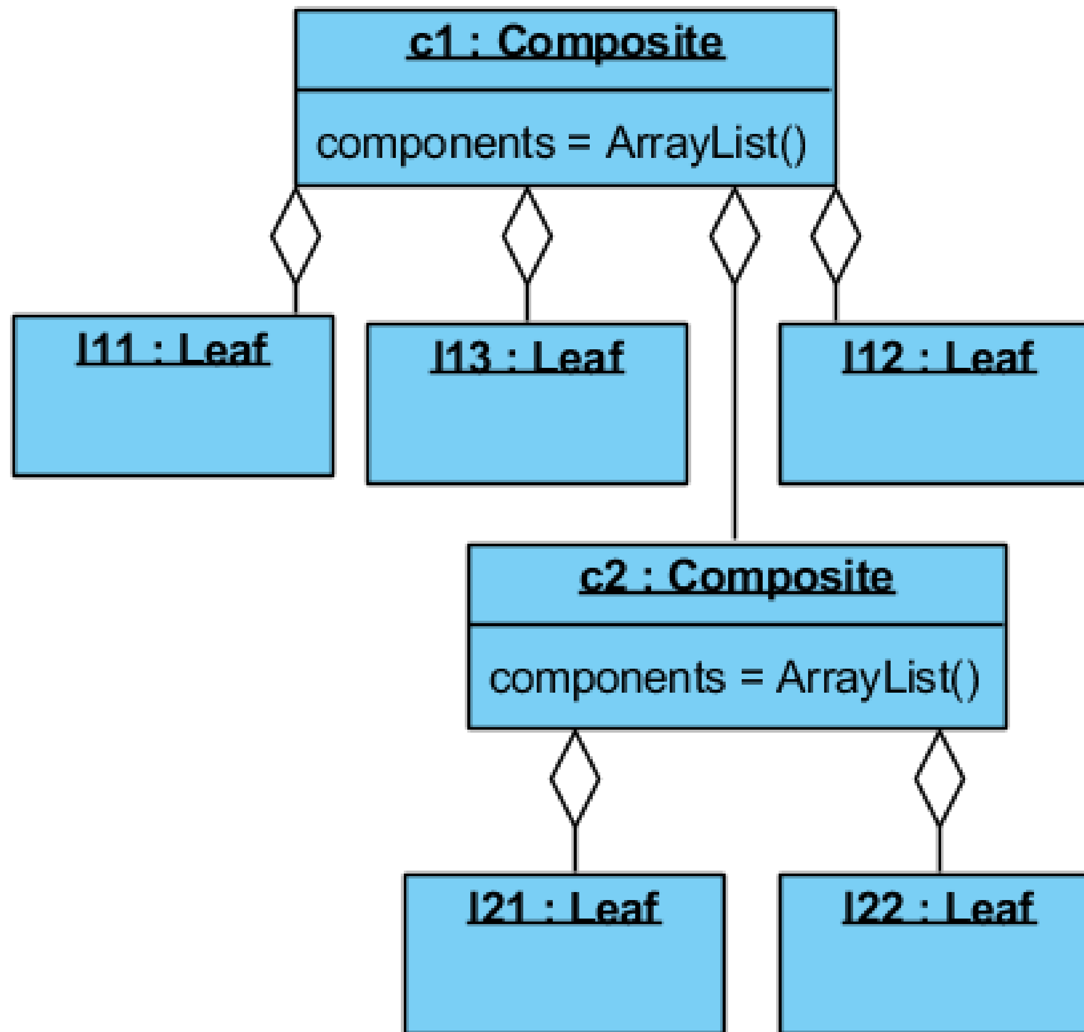
3) Kompozyt – *Composite – wzorzec obiektowy*





: Composite
: Leaf
: Composite
1: operation(data : Data) : Component
loop
[for (Component c:components)]
alt
[Leaf]
1.1: operation(data : Data) : Component
[Composite]
1.2: operation(data : Data) : Component
1.3: component

Diagram obiektów wzorca *Composite*



Przykład zastosowania wzorca *Composite*

Client

frame : JFrame

panel : JPanel

Composite

button : JButton

textfFeld : JTextField

scrollPanel : JScrollPanel

textArea : JTextArea

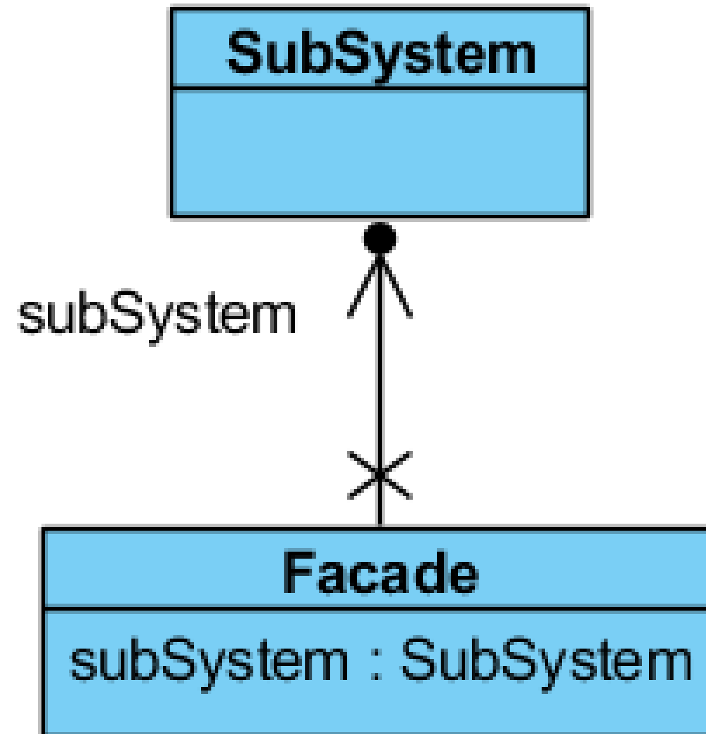
scrollBar : JScrollBar

Leaf

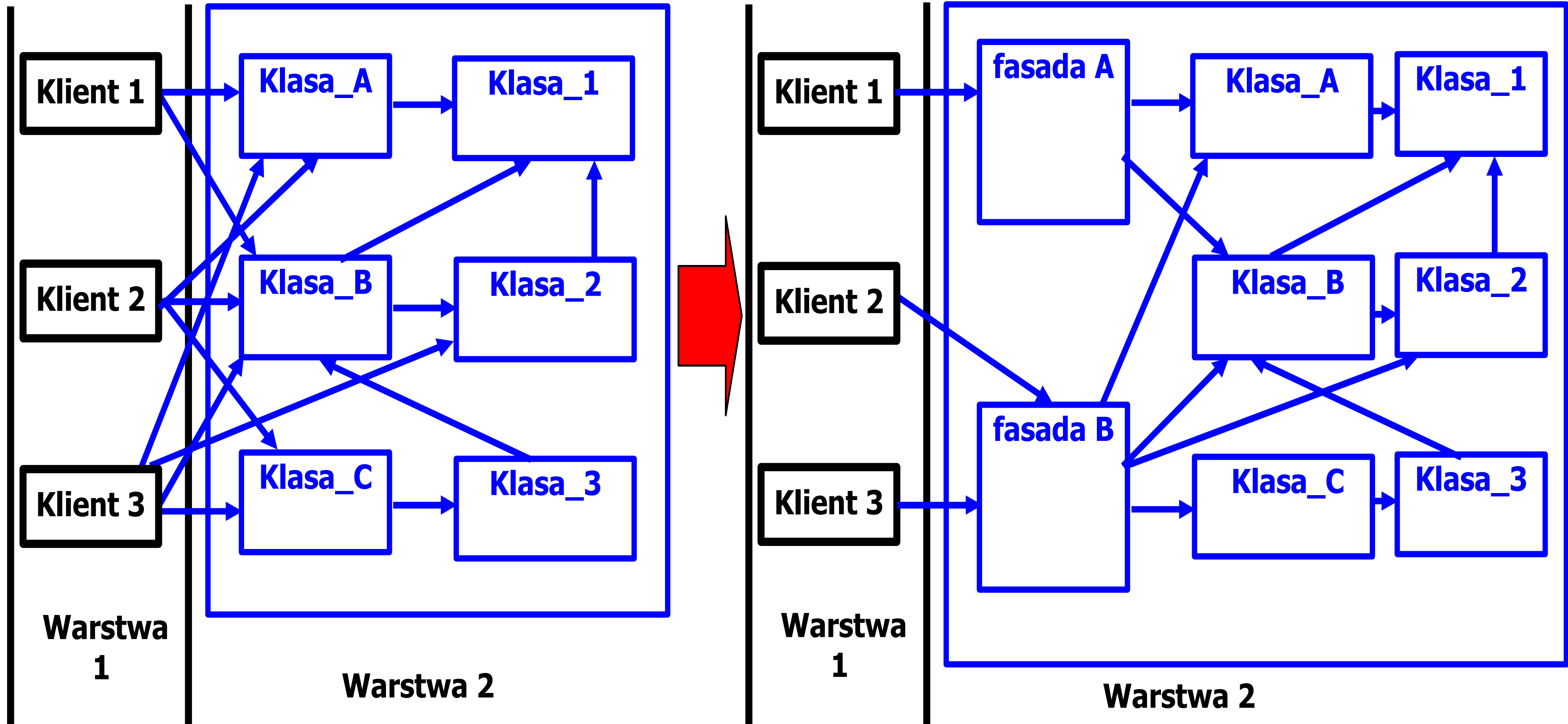
Charakterystyka wzorca *Composite*

- **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 złożonych
  - Łatwe dodawanie nowych obiektów
  - Trudność w zachowaniu ograniczeń przy budowie obiektów złożonych
- **Implementacja**: nowa klasa typu „Boundry" np. w pakiecie ***Swing***
  - klasa ***JComponent*** reprezentuje interfejs typu ***Component,***
  - natomiast klasa ***JButton*** klasę typu ***Leaf***,
  - natomiast klasa ***JPanel*** reprezentuje klasę typu ***Composite***.
- **Pokrewne wzorce: Chain of Responsibility, Decorator**, **Flight** (brak referencji do rodziców, ale współdzielenie komponentów), **Iterator**, **Visitor**

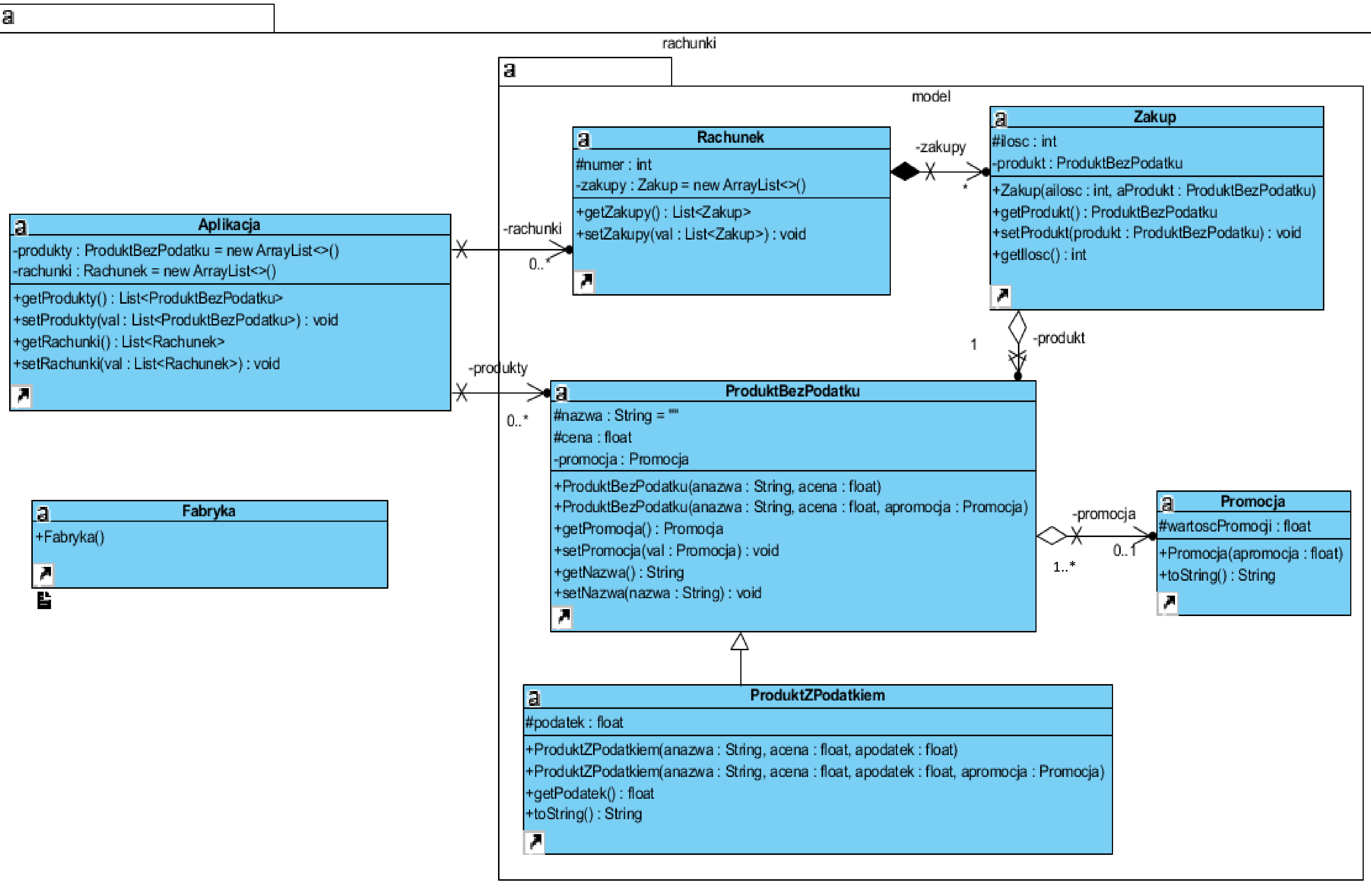
5) Fasada - *Facade – wzorzec obiektowy*



Wzorzec *Facade*



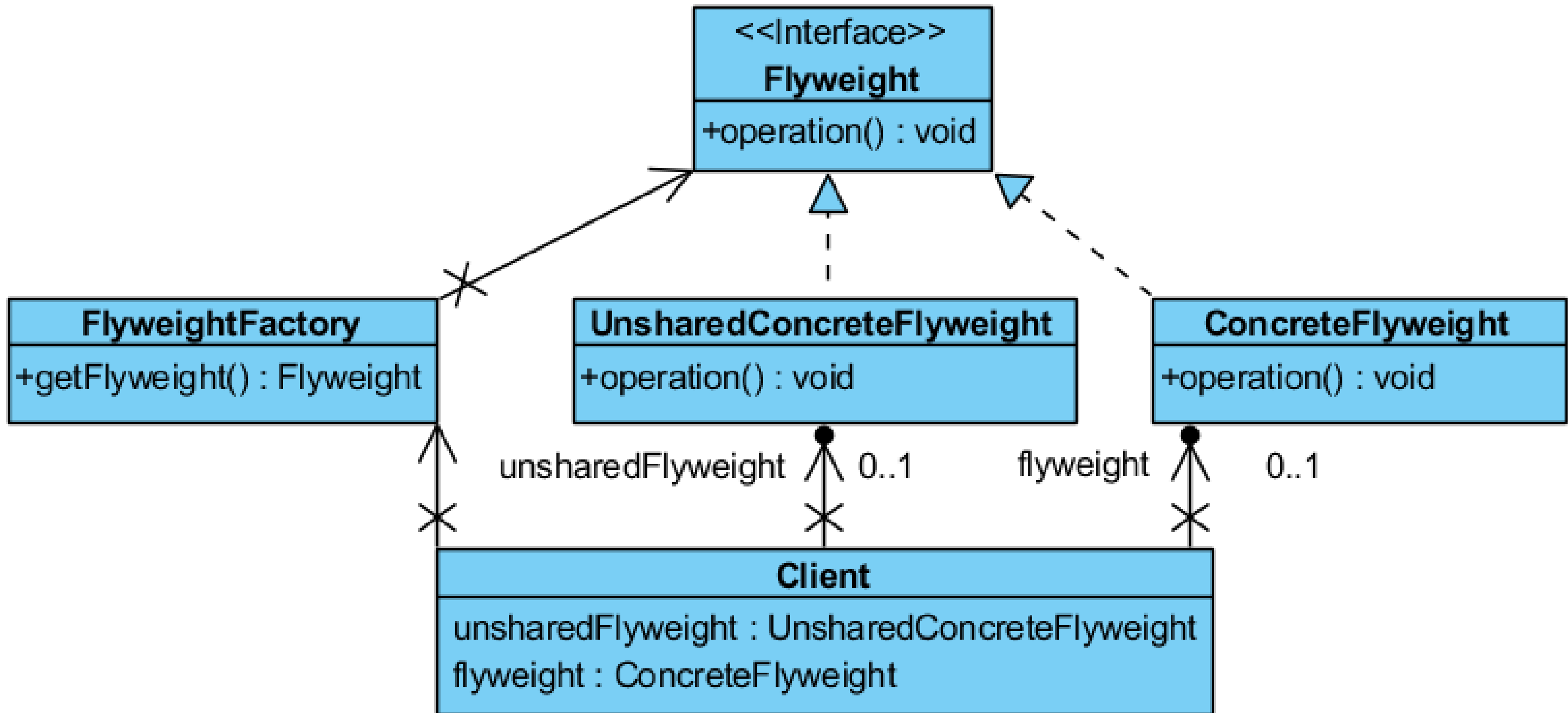
Przykład zatosowania wzorca *Facade za pomocą obiektu typu Aplikacja*



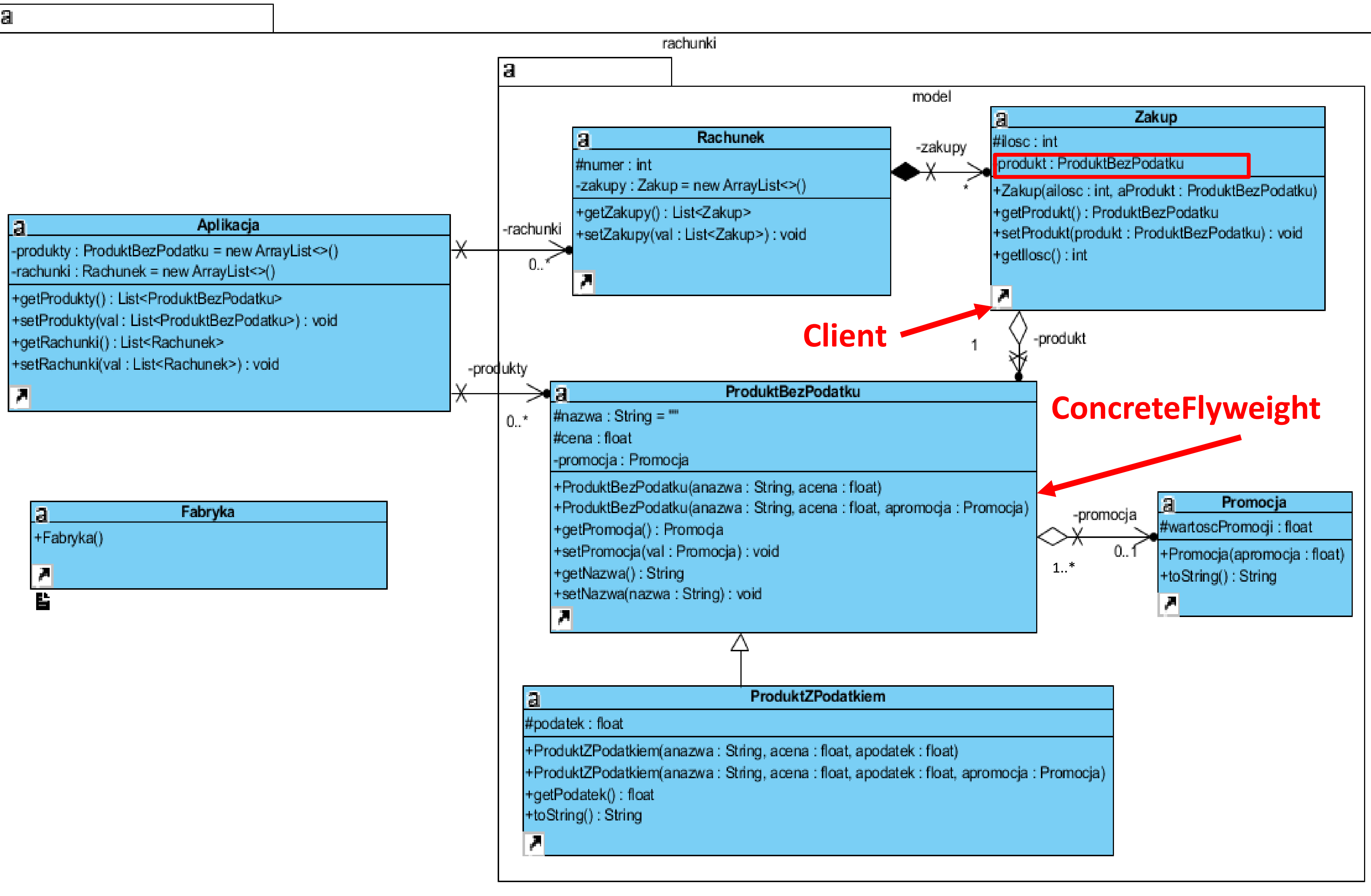
Charakterystyka wzorca *Facade*

- **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***
- **Pokrewne wzorce**: **Mediator** (który dodatkowo wprowadza nową funkcjonalność), współpraca z **Abstract Factory**, **Singleton**

6) Pyłek - *Flyweight – wzorzec obiektowy*



Przykład zastosowania wzorca *Flyweight* – *tworzenie obiektów typu Zakup zawierających informację o zakupionym produkcie*



Charakterystyka wzorca *Flyweight*

- **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 typu ***Client*** (pyłek) może być przechowywana w wielu obiektach typu ***Reservation*** (klient);
  - Referencja obiektu ***Book*** (pyłek) może być przechowywana w wielu obiektach typu ***Reservation*** (klient), a tylko w jednym typu ***Rental*** (klient);
  - **Pokrewne wzorce**: **Composite** (współdzielenie węzłow typu Leaf), **State**, **Strategy**

Wzorce projektowe

1. **Identyfikacja wzorców projektowych**
2. **Przegląd wzorców projektowych**

   Gang of Four – skrót odnoszący się do autorów ksiązki:

   Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*

   2.1. Wzorce kreacyjne

   2.2. Wzorce strukturalne

   2.3. Wzorce zachowania

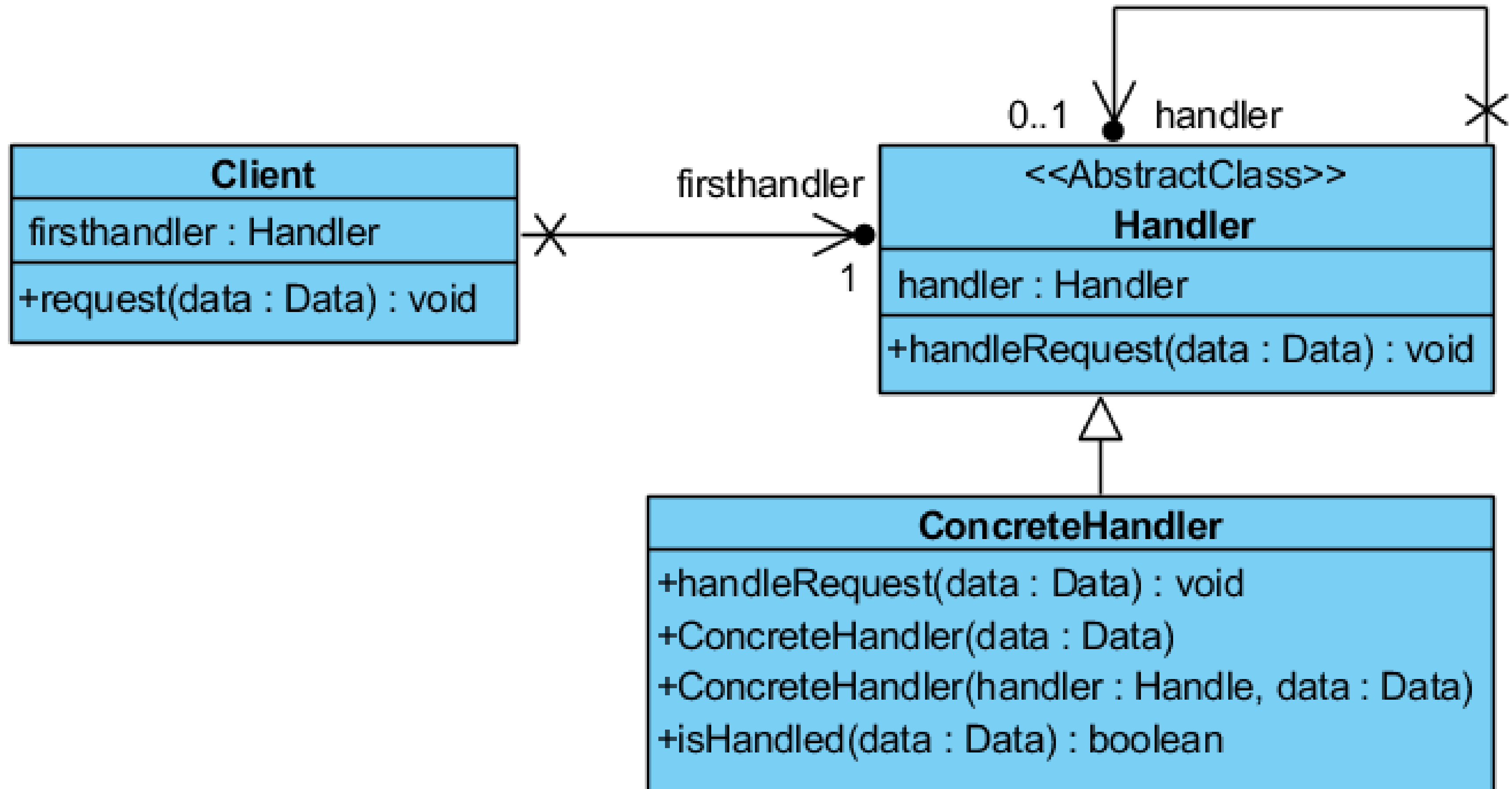
Wzorce zachowania

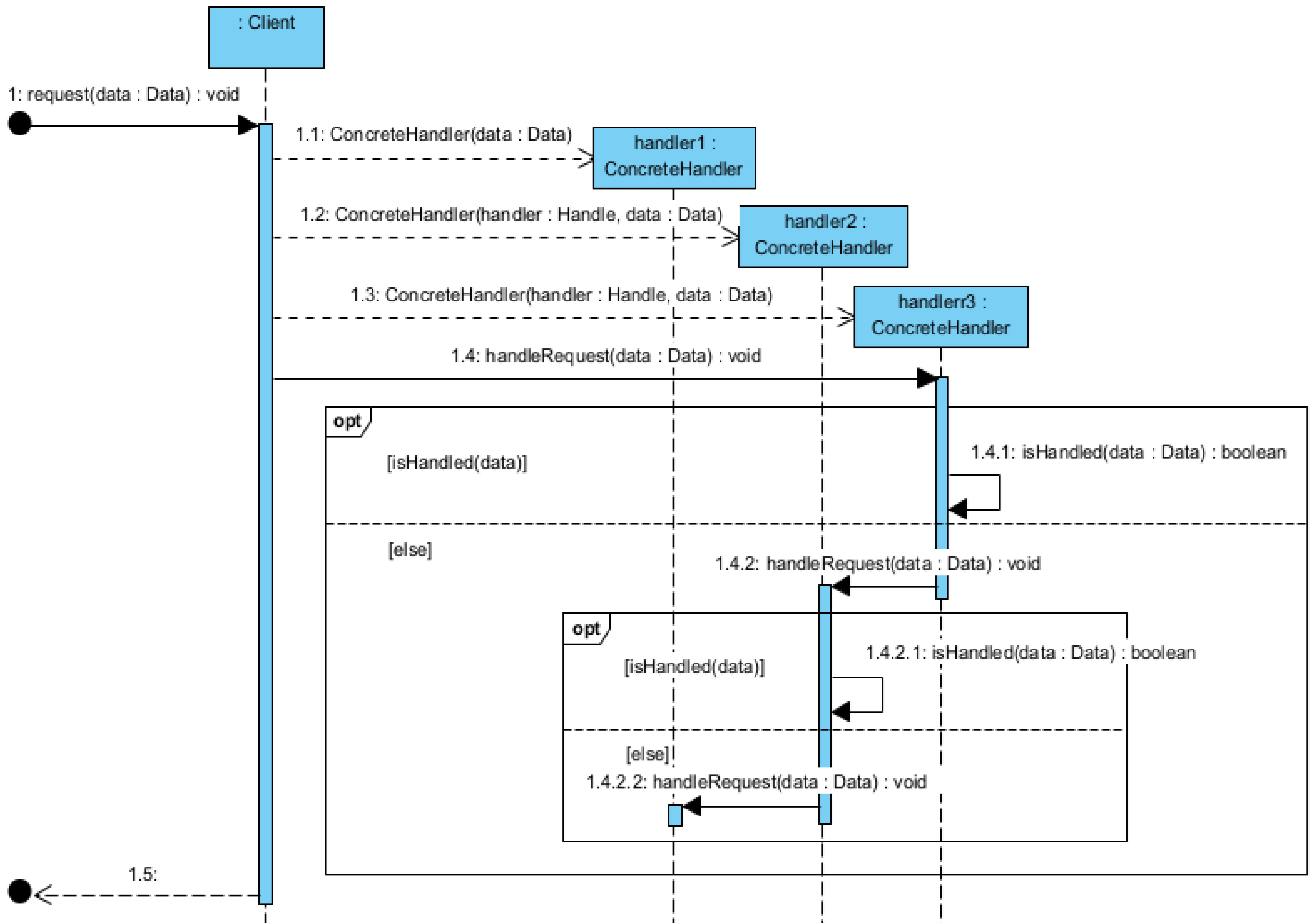
- **Cel: Przydzielają algorytmy i zobowiązania obiektom,**
- **Obejmują wzorce obiektów, klas oraz komunikacji między obiektami**

Wzorce zachowania - wybór

Wzorce zachowania	Aspekt, który może się zmienić
1) Chain of Responsibility	**Obiekt, który może zrealizować żądanie**
2) Command	Warunki i sposób realizacji żądania
3) Interpreter	Gramatyka i reprezentacja języka
4) Iterator	Sposób dostępu i przechodzenia elementów kolekcji
5) Mediator	Jak i które obiekty oddziałują na siebie?
6)Memento	Jakie prywatne informacje są przechowywane poza obiektem i kiedy?
7) Observer	**Liczba obiektów zależących od innego obiektu;jak zależne obiekty utrzymują aktualny stan**
8) State	Stany obiektów
9)Visitor	**Operacje, które można zastosować do obiektu (obiektów) bez zmiany jego klasy (ich klas)**
10) Strategy	**Algorytm**
11) Template Method	Kroki algorytmu

1) Łańcuch zobowiązań - *Chain of Responsibility*




: Client
1: request(data : Data) : void
1.1: ConcreteHandler(data : Data)
handler1 : ConcreteHandler
1.2: ConcreteHandler(handler : Handle, data : Data)
handler2 : ConcreteHandler
1.3: ConcreteHandler(handler : Handle, data : Data)
handlerr3 : ConcreteHandler
1.4: handleRequest(data : Data) : void
opt
[isHandled(data)]
1.4.1: isHandled(data : Data) : boolean
[else]
1.4.2: handleRequest(data : Data) : void
opt
[isHandled(data)]
1.4.2.1: isHandled(data : Data) : boolean
[else]
1.4.2.2: handleRequest(data : Data) : void
1.5:

Charakterystyka wzorca *Chain of Responsibility*

- **Problem:** Tworzy łańcuch odbierających obiektów i przekazuje wzdłuż niego żądanie, aż jakiś obiekt je obsłuży. Umożliwia kontakt nadającemu żądanie więcej niż z jednym obiektem i przekazania im obsłużenie tego żądania

- **Rozwiązanie**: interfejs ***Handler*** deklaruje interfejs obsługi żądań i ewentualnie implementuje odwołanie do nastepnika. Obiekt typu ***ConcreteHandler*** odpowiedzialny za wykrycie i obsługę swojego żądania; przekazuje żądanie do swojego następnika, jeśli nie może go obsłużyć.

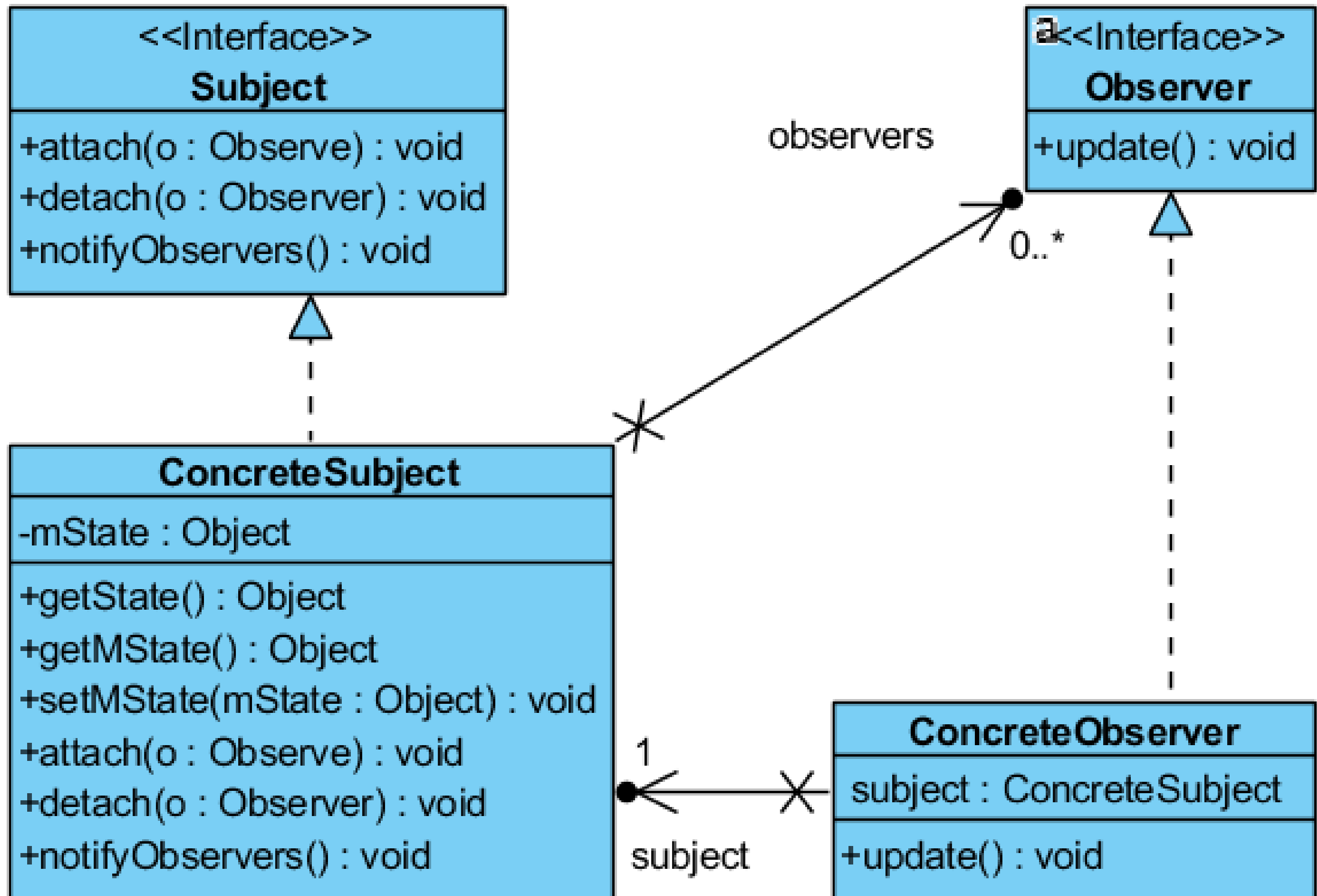
- **Klient wzorca**: generuje i kieruje żądania do listy obiektów ***ConcreteHandler***

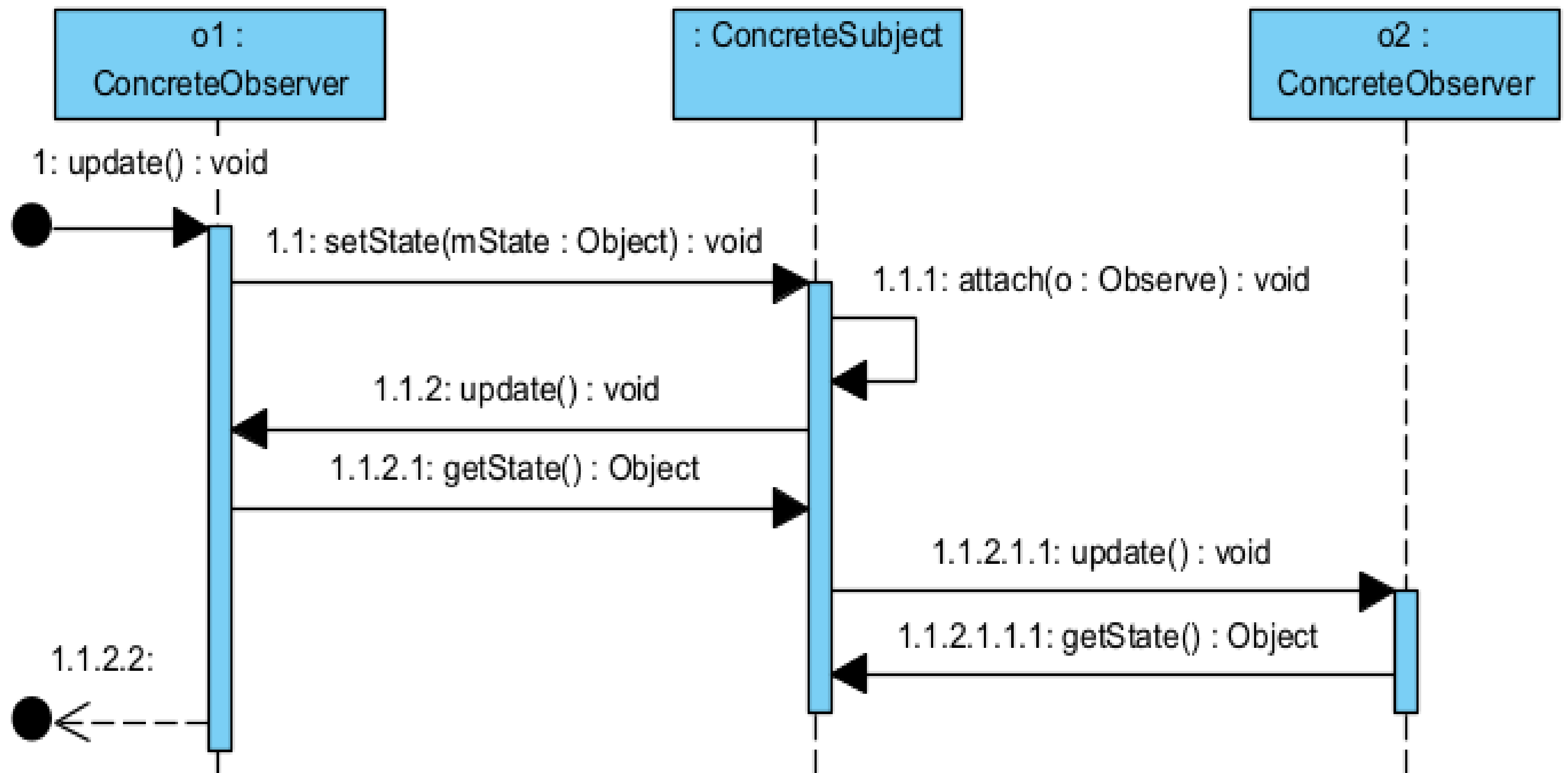
- **Rezutat**: Obiekt do obsługi żadań typu ***ConcreteHandler*** nie ma wyraźnej wiedzy o innych obiektach z łańcucha i nie musi znać struktury łańcucha. Łańcuch zobowiązań zwiększa elastyczność w przyznawaniu zgłoszeń serwisowych poprzez zmianę podklasy obiektów i struktur łańcucha obiektów - ale bez gwarancji otrzymania żądania.

- **Implementacja**: Służy do obsługi zdarzeń

- **Pokrewne wzorce:**
  - Stosowany w połączeniu z wzorcem Kompozyt (**Composite**)

7) Obserwator - *Observer*

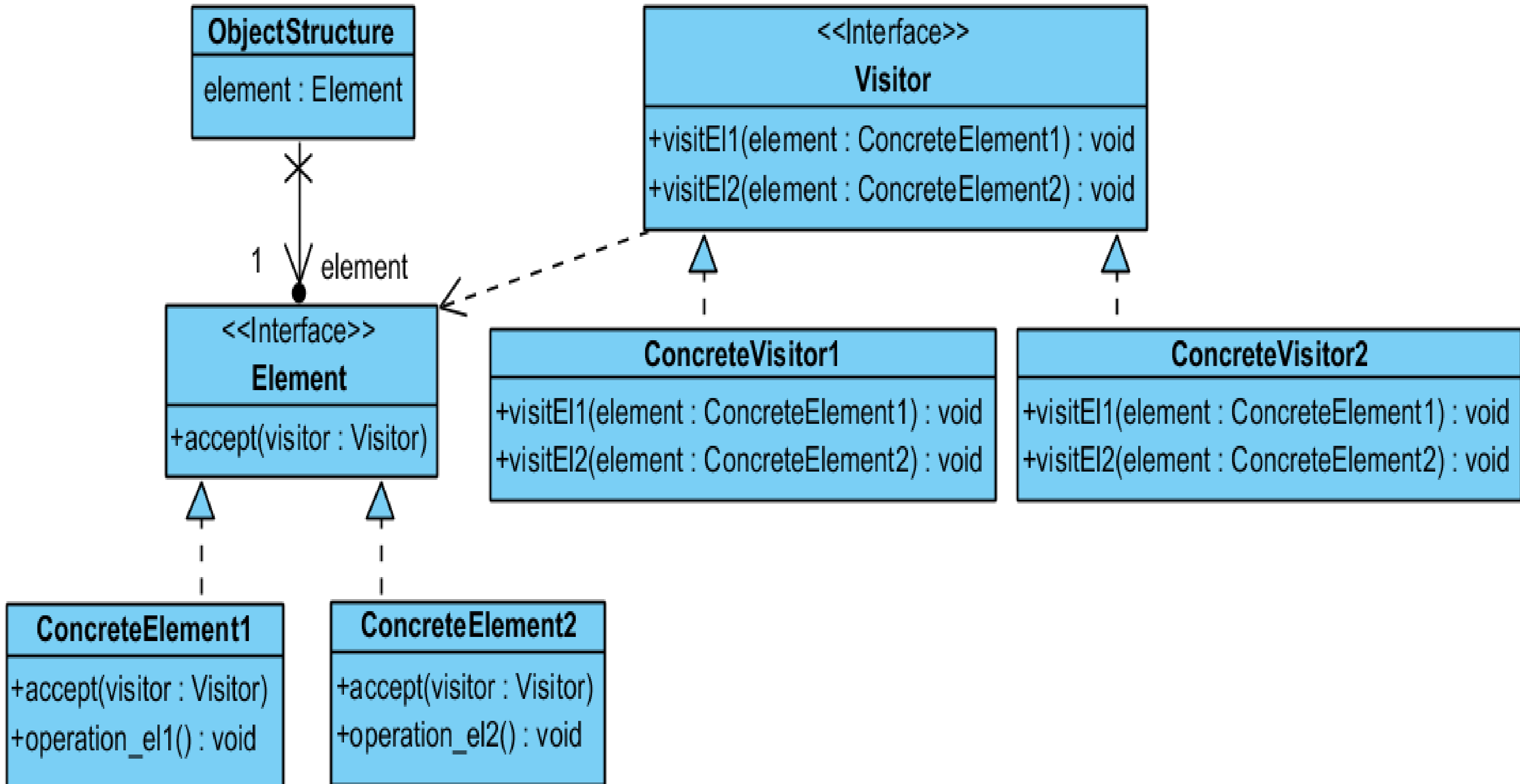


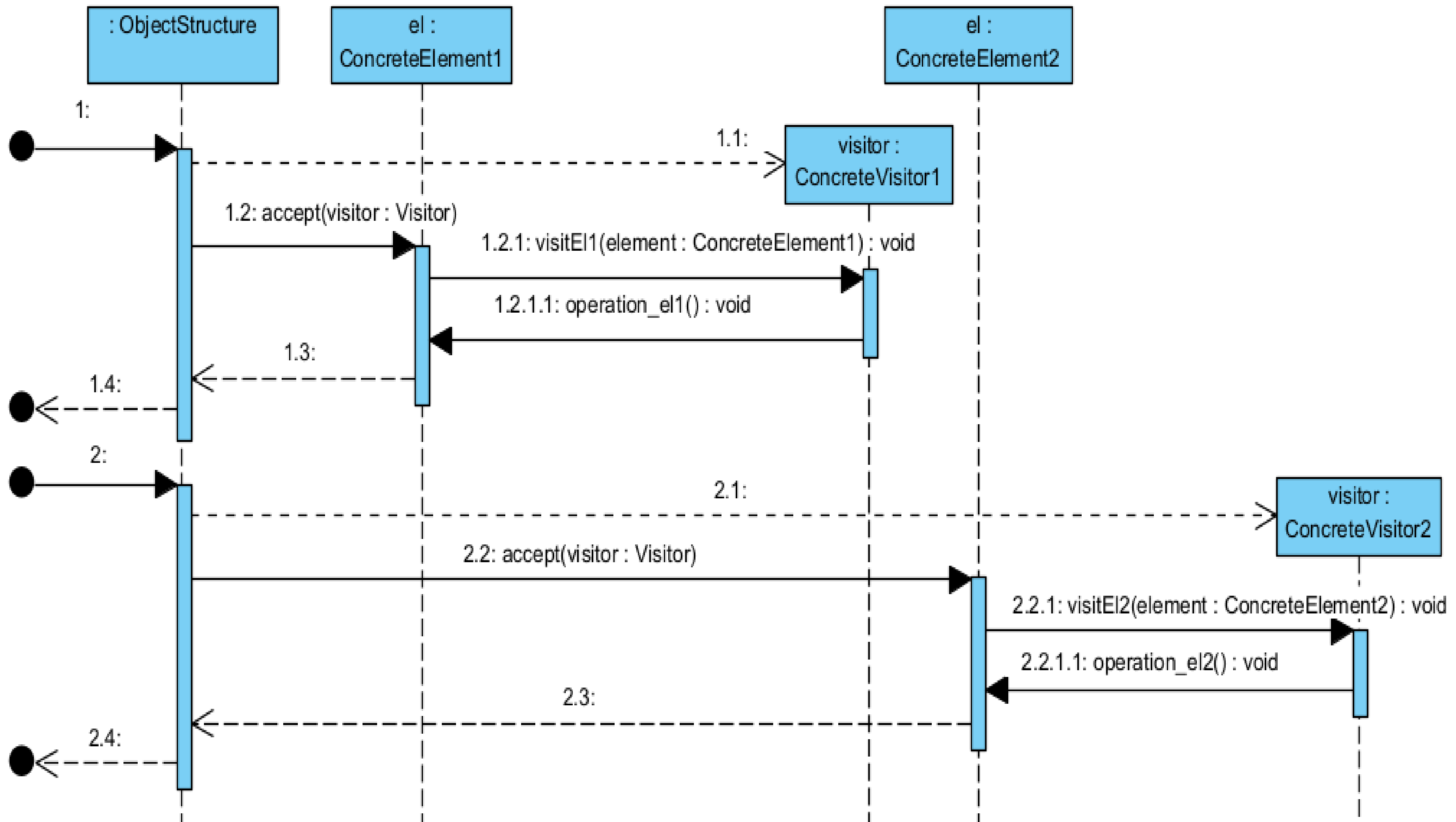

o1 :
ConcreteObserver
: ConcreteSubject
o2 :
ConcreteObserver
1: update() : void
1.1: setState(mState : Object) : void
1.1.1: attach(o : Observe) : void
1.1.2: update() : void
1.1.2.1: getState() : Object
1.1.2.1.1: update() : void
1.1.2.1.1.1: getState() : Object
1.1.2.2:

Charakterystyka wzorca *Observer*

- **Problem:** Konieczność okreslenia zależności typu jeden-do-wielu między obiektami - gdy obiektu zmienia stan, wszystkie obiekty zależne są automatycznie powiadamiane i aktualizowane
- **Rozwiązanie:** Obiekty typu ***ConcreteSubject*** zna swoich obserwatorów i ma dostęp do wielu obiektów typu ***ConcreteObserver*** - kiedy zmienia swój status, powiadamia obserwatorów (interfejs ***Observer***) ich metodą aktualizacji (**update**). Obiekt typu ***ConcreteObserver*** ma odwołanie do obiektu typu ***ConcreteSubject*** i posiada jego stan, który musi być zgodny ze stanem tego obserwowanego obiektu typu ***ConcreteSubject***
- **Rezultat:**
  - Abstrakcyjny związek pomiędzy obiektami typu ***ConcreteSubject*** i obiektów typu ***ConcreteObserver***
  - Wsparcie dla wysyłania wiadomości - obiekty typu ***ConcreteSubject*** przesyłają zgłoszenie, nie znając odbiorcy
  - Nieoczekiwane modyfikacje, nie zawsze pożądane - ze względu na fakt, że obserwatorzy nie wiedzą o istnieniu innych obserwatorów
- **Pokrewne wzorce:**
  - Zastosowanie wzorców **Mediator** lub **Singleton** do komunikacji między obserwowanymi obiektami i obserwatorami

9) Odwiedzający - *Visitor*





: ObjectStructure
el : ConcreteElement1
el : ConcreteElement2
1:
1.1:
visitor : ConcreteVisitor1
1.2: accept(visitor : Visitor)
1.2.1: visitEl1(element : ConcreteElement1) : void
1.2.1.1: operation_el1() : void
1.3:
1.4:
2:
2.1:
visitor : ConcreteVisitor2
2.2: accept(visitor : Visitor)
2.2.1: visitEl2(element : ConcreteElement2) : void
2.2.1.1: operation_el2() : void
2.3:
2.4:

: ObjectStructure
el : ConcreteElement1
visitor : ConcreteVisitor1
1:
1.1:
1.2: accept(visitor : Visitor)
1.2.1: visitEl1(element : ConcreteElement1) : void
1.2.1.1: operation_el1() : void
1.3:

: ObjectStructure
el : ConcreteElement2
visitor : ConcreteVisitor2
1:
1.1:
1.2: accept(visitor : Visitor)
1.2.1: visitEl2(element : ConcreteElement2) : void
1.2.1.1: operation_el2() : void
1.3:

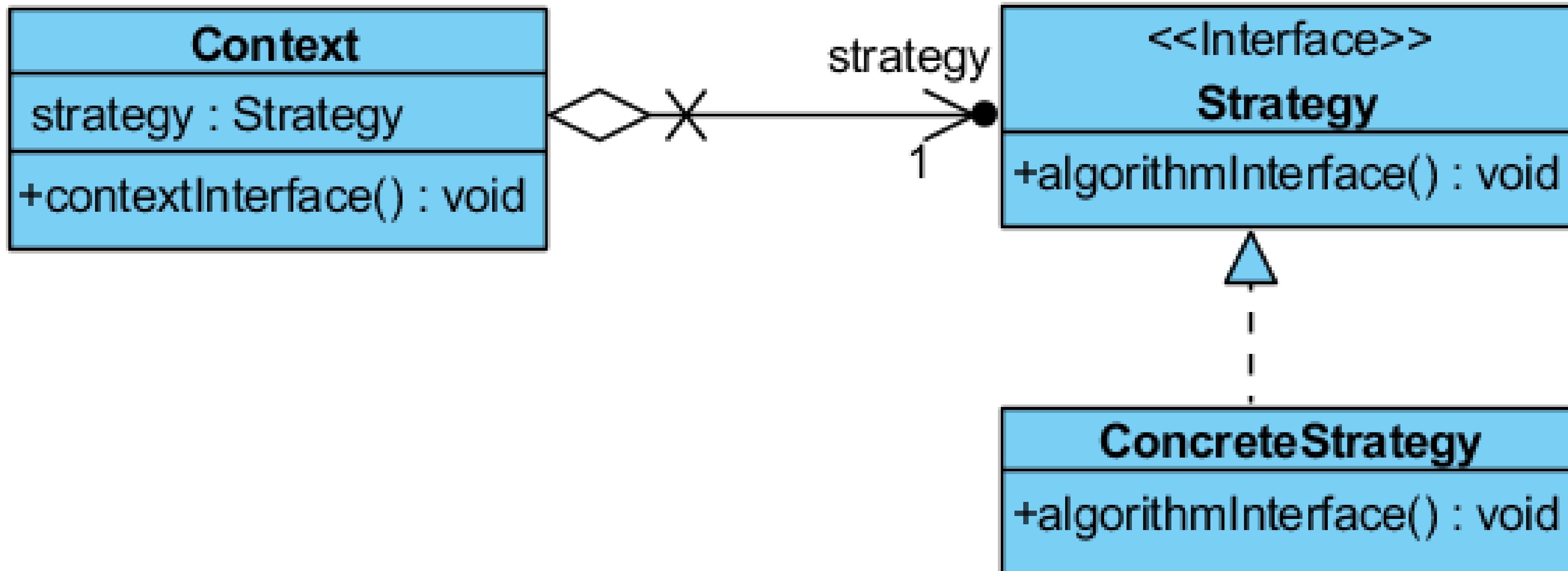
Charakterystyka wzorca *Visitor*

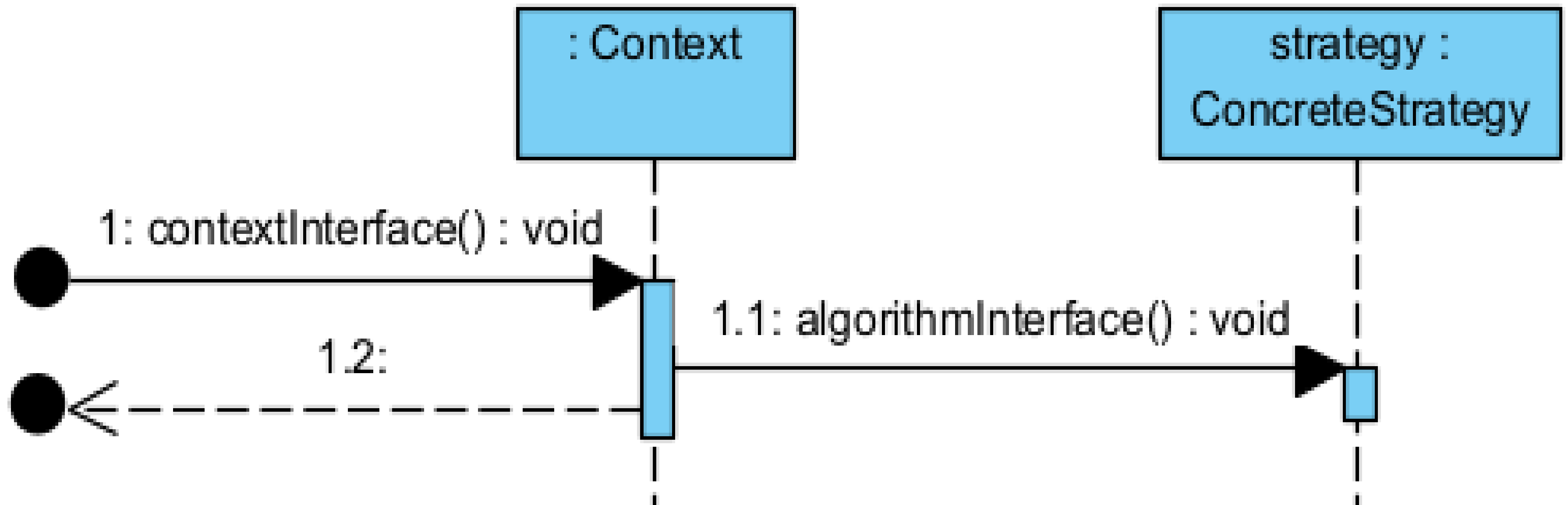
- **Problem:** Należy pozwolić na zdefiniowanie nowej operacji bez zmiany definicji klasy obiektów, w których ona działa.
- **Rozwiązanie:** obiekt typu ***ObjectStructure*** może wprowadzić swoje obiekty i umożliwia obiektom, które implementują interfejs ***Element*** (i mogą być złożone) odwiedzanie przez obiekty implementujące interfejs ***Visitor***. Interfejs ***Visitor*** deklaruje metody wizytujące (np. **visit_el1**), które otrzymują, jako parametr, obiekt do odwiedzenia, implementujący interfejs ***Element***. Obiekt typu ***ConcreteVisitor*** implementuje metody wizytujące, która pozwalają na przechowywanie informacji o stanie poszczególnych obiektów typu ***ConcreteElement*****.** Te odwiedzane obiekty posiadają metody umożliwiajace odwiedzenie ich stanu (np. **operation_el1**), które są wywoływane przez metody wizytujące obiektów typu ***ConcreteVisitor***.
- **Klient wzorca:** Klient reprezentowany przez obiekt typu ***ObjectStructure*** musi utworzyć obiekty typu ***ConcreteVisitor*** i przejść przez całą strukturę obiektów typu ***Element***, odwiedzając każdy element za pomocą obiektu typu ***ConcreteVisitor***. Każdy obiekt typu ***Element*** wywołuje metodę wizytującą obiektu ***ConcreteVisitor***, dając dostęp do siebie, i pozwala jej wywołać odpowiednią metodę swojej klasy, umożliwiającą zbadanie stanu.

- **Rezultat:**
  - Łatwe dodawanie nowych działań, które są zależne od złożonych elementów. Nowa operacja wymaga dodania nowego obiektu typu ***ConcreteVisitor***
  - Połączenie związanych ze sobą działań w całej strukturze obiektów typu ***Element*** w klasie, która implementuje interfejs ***Visitor*** oraz separacji niepowiązanych w podklasach obiektów implementujących interfejs ***Visitor***
  - Trudne dodawanie nowych klas ***ConcreteElement***, bo trzeba zadeklarować nową metodę wizytującą w interfejsie ***Visitor*** i nowe implementacje metod wizytujących w klasach ***ConcreteVisitor***

- **Pokrewne wzorce:**
  - Wzorzec **Visitor** może służyć do odwiedzania obiektów wzorca Kompozyt (**Composite**)
  - Odwiedzający może służyć do interpretowania we wzorcu **Interpreter**

10) Strategia - *Strategy*




: Context
strategy :
ConcreteStrategy
1: contextInterface() : void
1.1: algorithmInterface() : void
1.2:

Charakterystyka wzorca *Strategy*

- **Problem:** Wybór różnych reguł biznesowych lub różnych wersji algorytmów w zależności od kontekstu
- **Rozwiązanie**: Separuje wybór wersji algorytmu od jego implementacji
- **Rezultat**:
  - Zdefiniowanie rodziny algorytmów
  - Zdefiniowanie interfejsu klasy typu ***Strategy*** zawierającej metodę dostarczającą algorytm i metody w klasie typu ***Context***, która korzysta z algorytmu
  - Eliminacja instrukcji wyboru lub warunkowej do wyboru algorytmu strategii - wprowadzenie mechanizmu polimorfizmu, szczególnie, gdy wybór algorytmu nie ma charakteru przejściowego

- **Klient**: Decyzję o **implementacji** obiektu strategii i kontekstu podejmuje właściciel tych obiektów, który dostarcza informacji o utworzeniu właściwego obiektu strategii oraz obiektu kontekstu np. za pomocą fabryki obiektów. Jest nim obiekt typu **klient wzorca** np. dowolny obiekt z warstwy prezentacji.
- **Implementacja**:
  - Obiekt kontekstowy klasy bazowej typu ***Context*** **(„Entity”)** i jej pochodnych, który używa algorytmu, posiada obiekt klasy bazowej typu ***Strategy*** **(„Entity”)** lub jej pochodnych**,** dostarczający algorytm**.** Obiekt kontekstu posiada metodę wirtualną wywołującą wirtualną metodę algorytmu obiektu strategii. Każda klasa dziedzicząca od klasy typu ***Strategy*** implementuje taką metodę algorytmu w inny sposób.
  - Decyzja o sposobie użycia strategii, czyli użycia właściwego obiektu typu ***Strategy*** zależy od klasy typu ***Context***
- **Pokrewne wzorce:**
  - Kandydaci na obiekty wzorca Pyłek (**Flyweight**)

Wzorce projektowe

1. **Identyfikacja wzorców projektowych**
2. **Przegląd wzorców projektowych**

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

   2.1. Wzorce kreacyjne

   2.2. Wzorce strukturalne

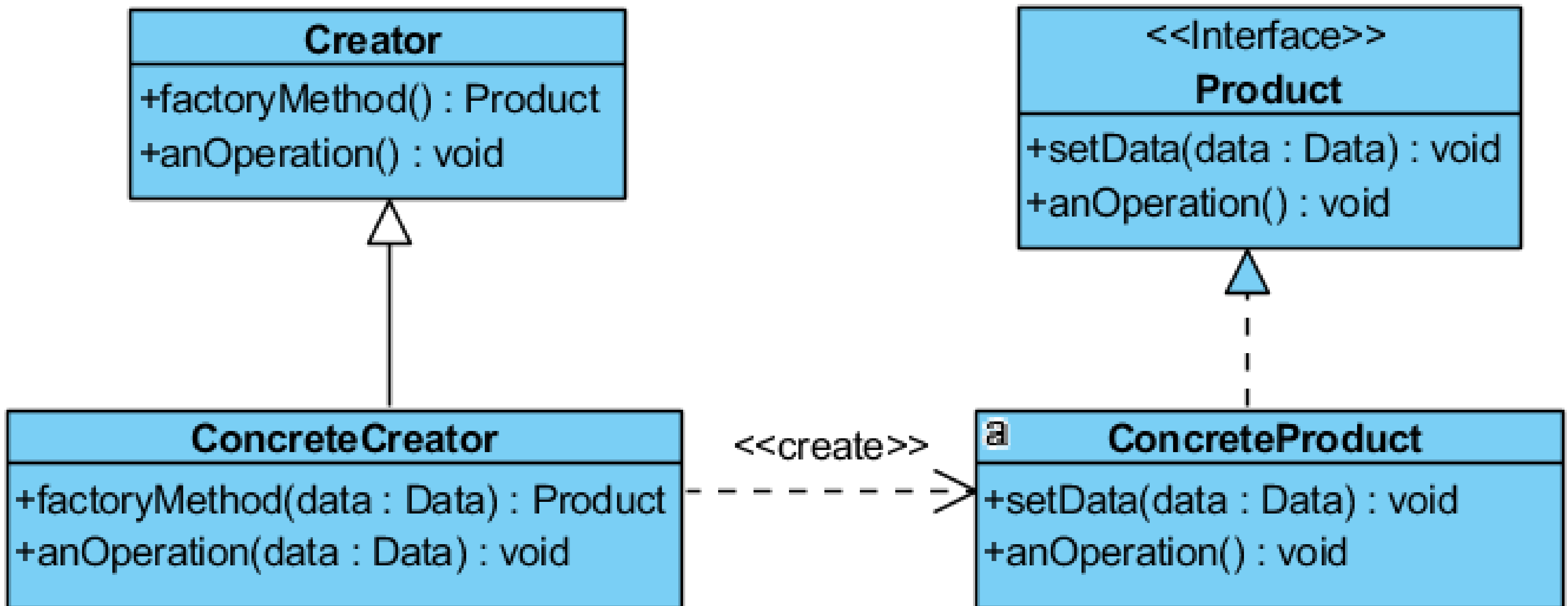
   2.3. Wzorce zachowania

3. Dodatek - uzupełnienie

Dodatkowy materiał dotyczący wzorców kreacyjnych

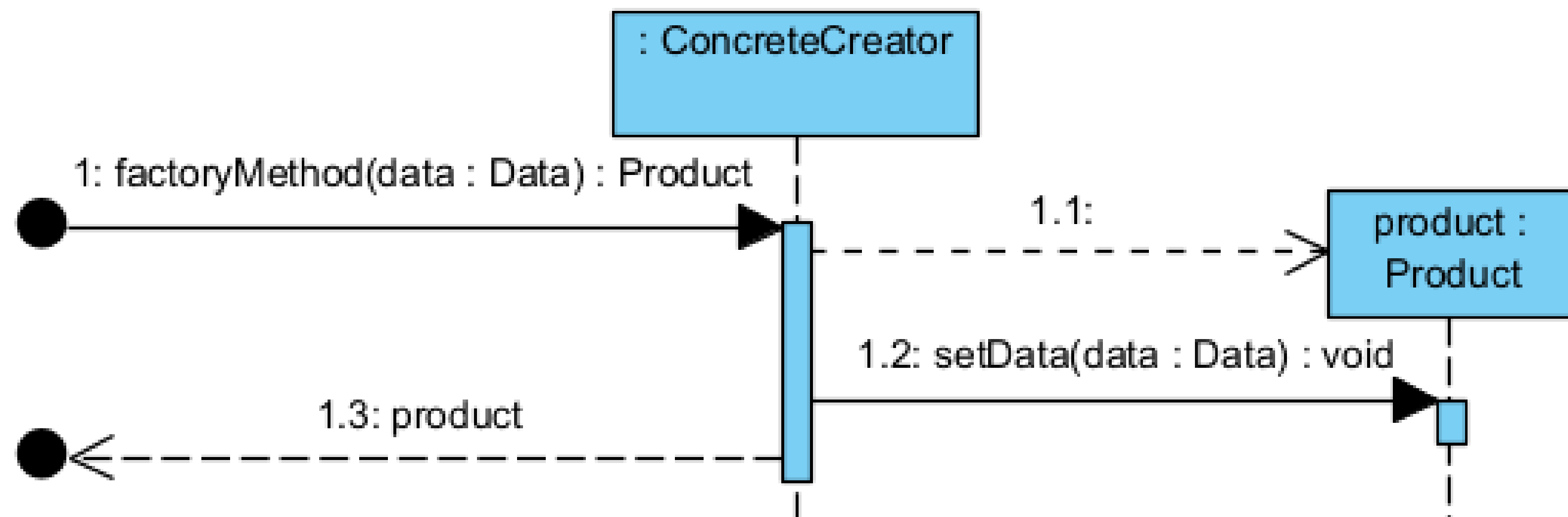
Wzorzec projektowy	Aspekt, który może się zmienić
1)Builder	Sposób tworzenia złożonych obiektów
2)Abstract Factory	Rodziny obiektów
3)Factory Method	Podklasa tworzonego obiektu
4)Prototype	Typ klasy tworzonego obiektu
5)Singleton	Jedna kopia obiektu

3) Metoda wytwórcza – *Factory Method*



: ConcreteCreator
product : Product
1: anOperation(data : Data) : void
1.1: factoryMethod(data : Data) : Product
1.2: product
1.3: anOperation() : void
1.4:

Metoda wytwórcza:



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;
  }
```

Charakterystyka wzorca *Factory Method*

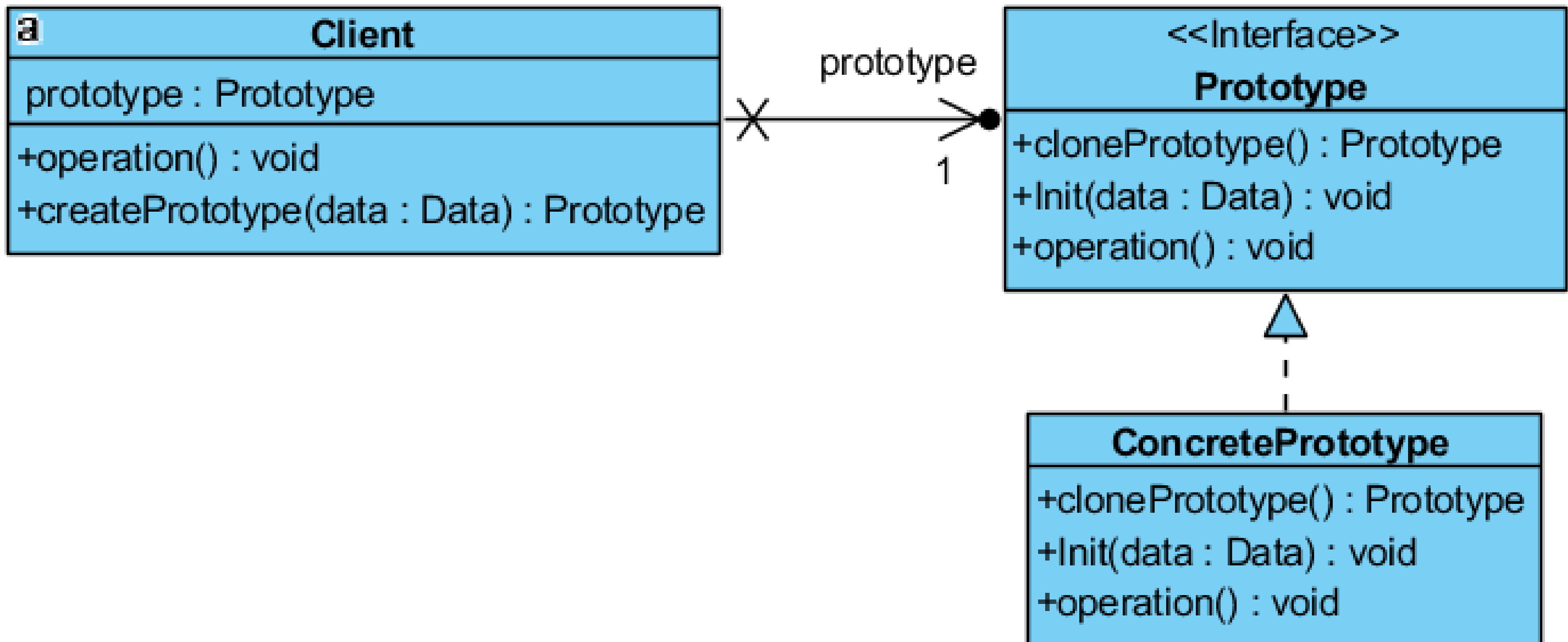
- **Problem**: Obiekt klasy A przetwarzającej dane innego obiektu lub obiekt klasy B pochodnej klasy A, nie wie, jaki obiekt należy przetwarzać z danej rodziny obiektów.

- **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***.

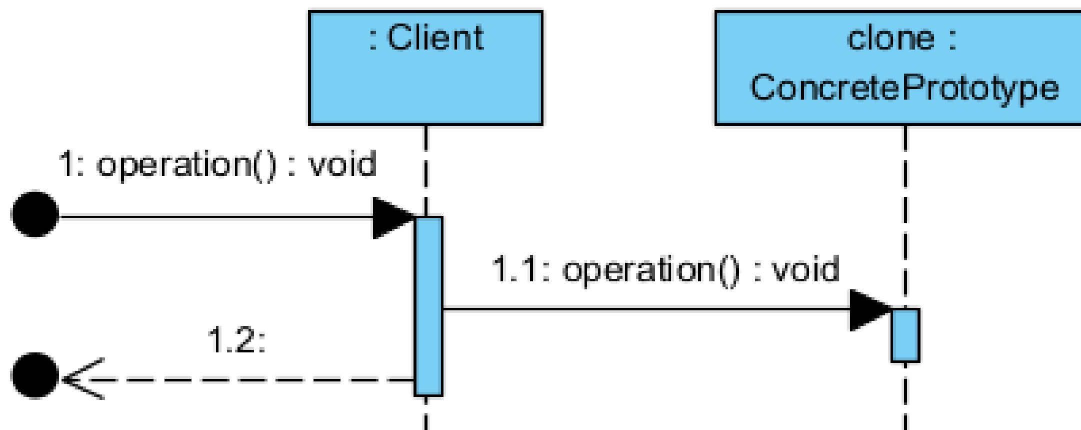
- **Klient wzorca**: klient wybiera odpowiedni obiekt klasy ***ConcreteCreator*** z metodą wytwórczą odpowiednią do wytworzenia obiektu typu ***ConcreteProduct.*** Przetwarzanie obiektu typu ***ConcreteProdukt*** polega na tym, że tylko metoda wytwórcza zna reprezentacje obiektu i sposób jego tworzenia, 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.***
- **Pokrewne wzorce: Abstract Factory**

4) Prototyp - *Prototype*



: Client
clone : ConcretePrototype
1: createPrototype(data : Data): Prototype
1.1:
prototype : ConcretePrototype
1.2: cloneProtoype() : Prototype
1.3: clone
1.4: Init(data : Data) : void
1.5: operation():void
1.6: clone



Przykład klonowania kolekcji obiektów

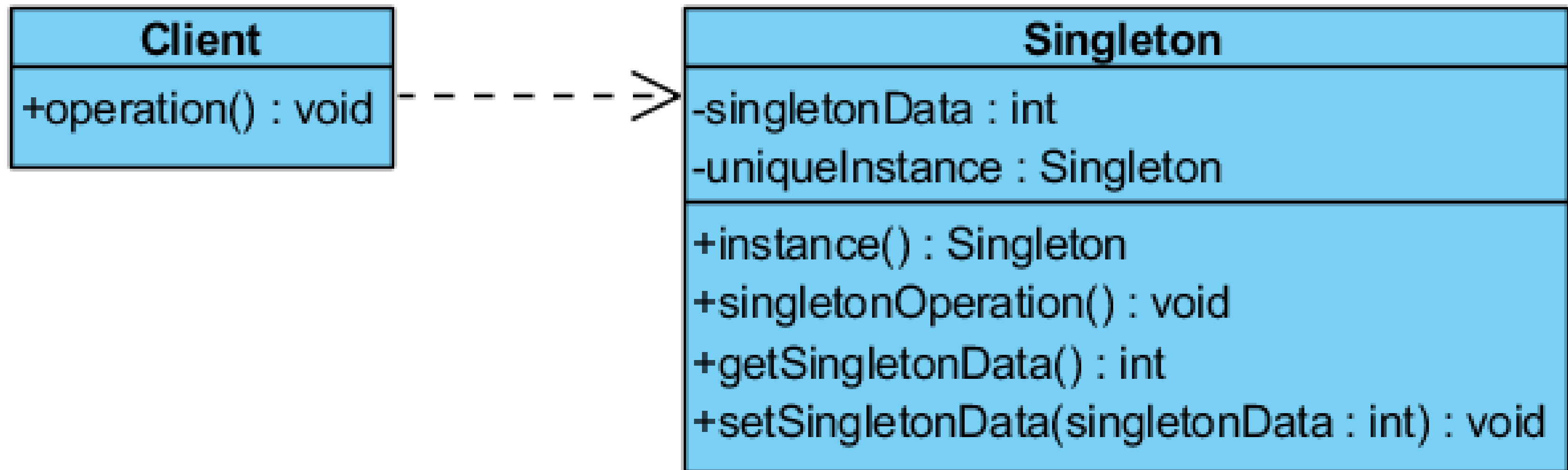
```
public class Main {

   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]
   }
}
```

Charakterystyka wzorca *Prototype*

- **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
  - Specyfikowanie nowych obiektów przez zmienianie wartości lub/i struktury
  - 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();}
}
```

5) Pojedynczy obiekt - *Singleton*




```
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
  - Możliwe udoskonalanie operacji i reprezentacji,
  - Zmienna liczba egzemplarzy
  - Większa elastyczność niż metod klasowych
- **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**
- **Pokrewne wzorce**: **Abstract Factory**, **Builder**, **Proxy**

Dodatkowy materiał dotyczący wzorców strukturalnych

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)*

Client
adapter : Target

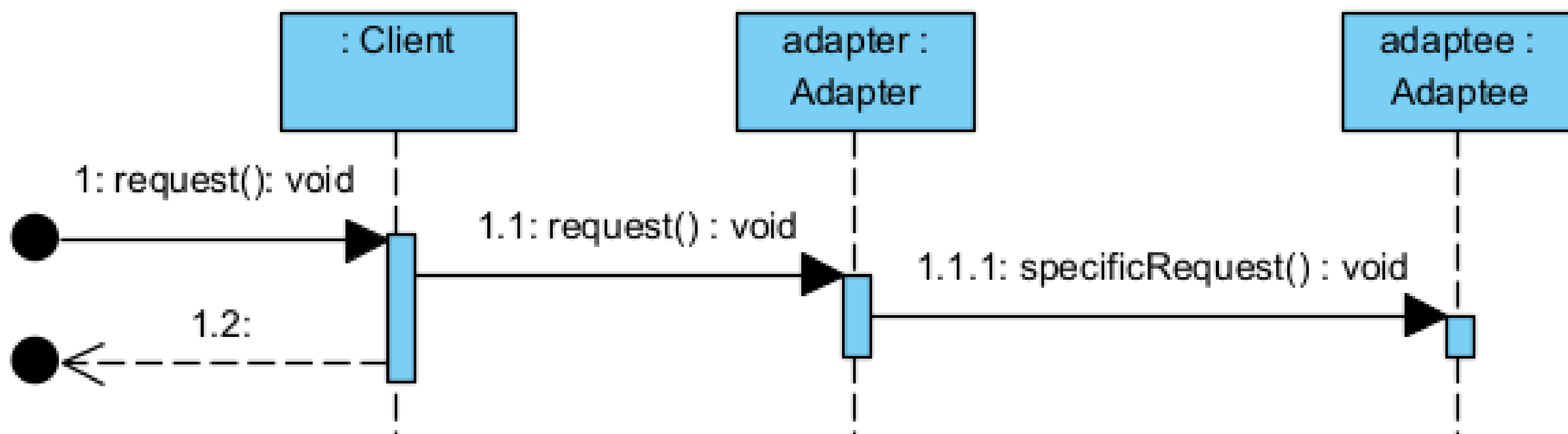
adapter
1

<<Interface>>
Target
+request() : void

Adapter
adaptee : Adaptee
+request() : void

adaptee
1

Adaptee
+specificRequest() : void



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

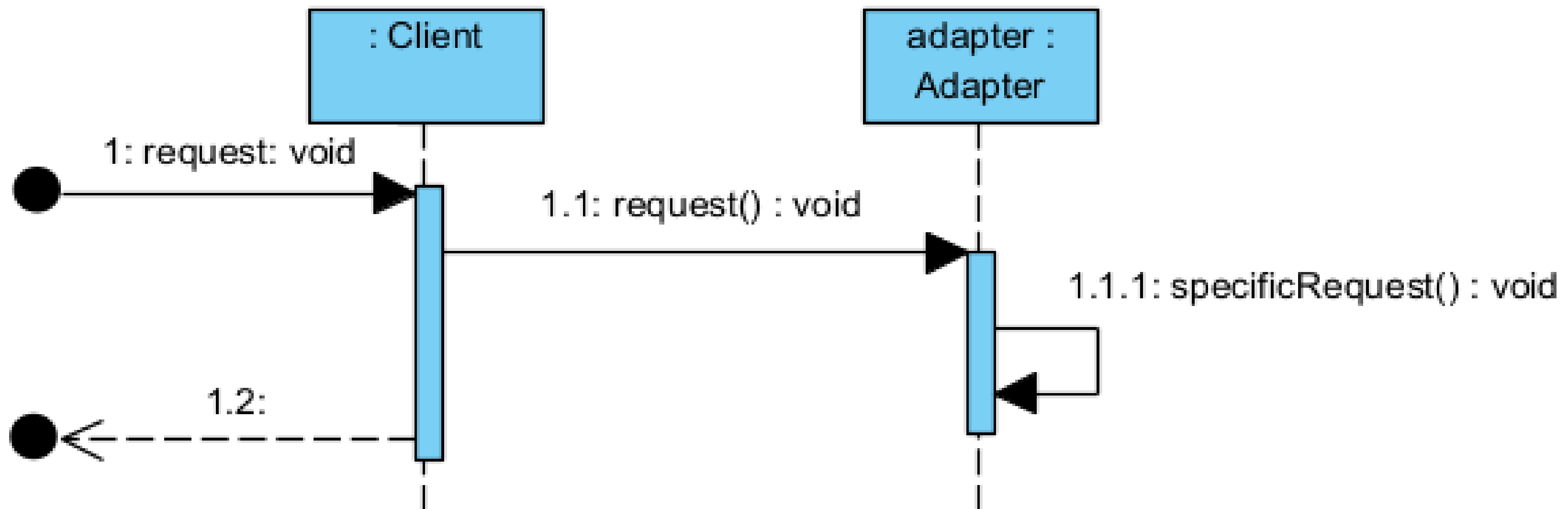
Client
adapter : Target
+request() : void

adapter
1

<<Interface>>
Target
+request() : void

Adaptee
+specificRequest() : void

Adapter
+request() : void

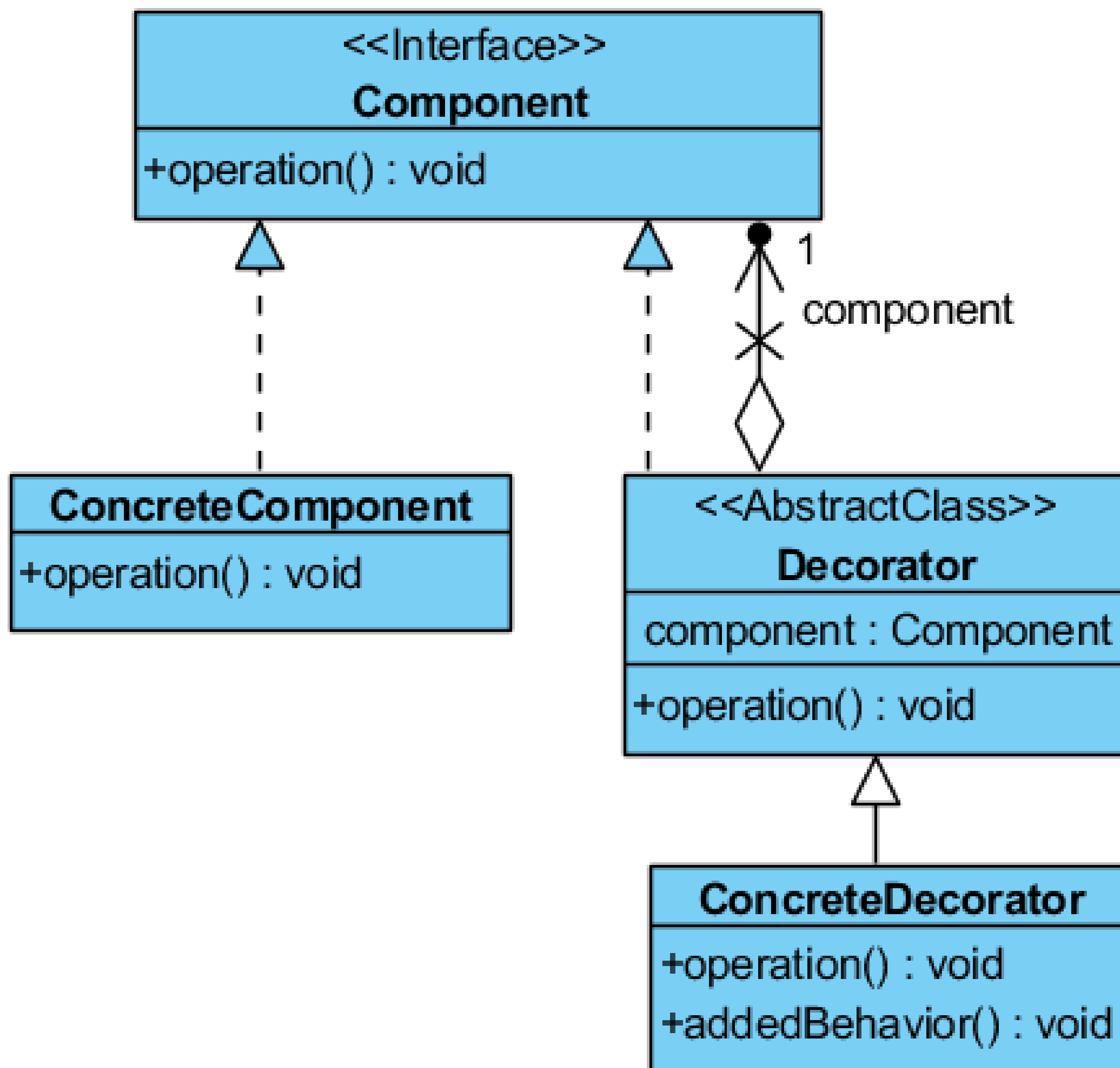


Charakterystyka wzorca *Adapter*

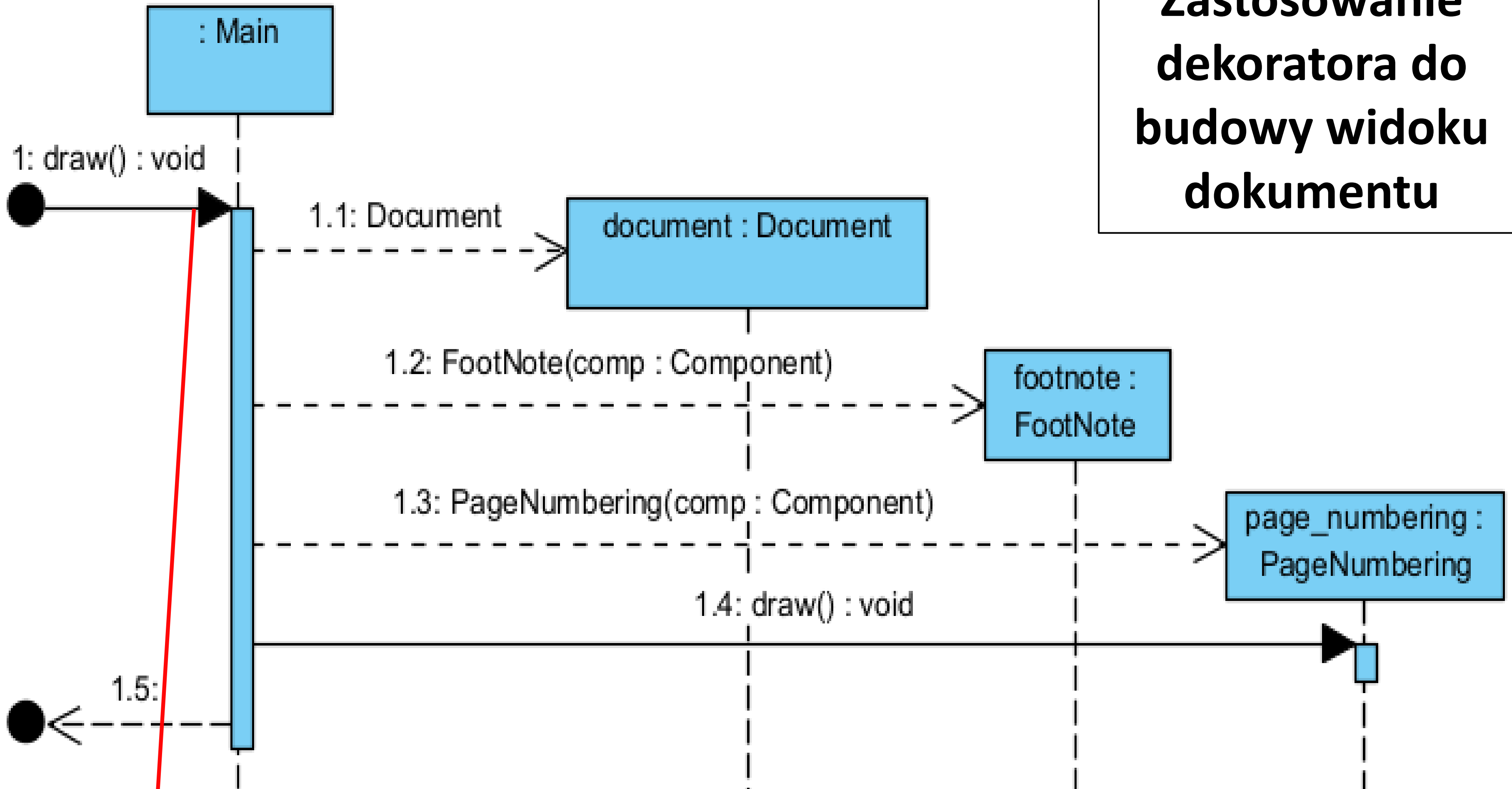
- **Problem**: Należy dostosować interfejs klasy do interfejsu oczekiwanego przez klienta wzorca np. związanego ze zmianą biblioteki klas obsługujących grafikę
- **Rozwiązanie**: Obiekt typu ***Client*** używa obiektów typu ***Adapter*** implementujących interfejs typu ***Target*** i jednocześnie pośredniczących 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, schemat 1a):
    - Umożliwia jednemu obiektowi typu ***Adapte***r 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, schemat 1b):
    - 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”
- **Pokrewne wzorce**: **Bridge**, **Decorator**, **Proxy**

4) Dekorator – *Decorator* – *wzorzec obiektowy*

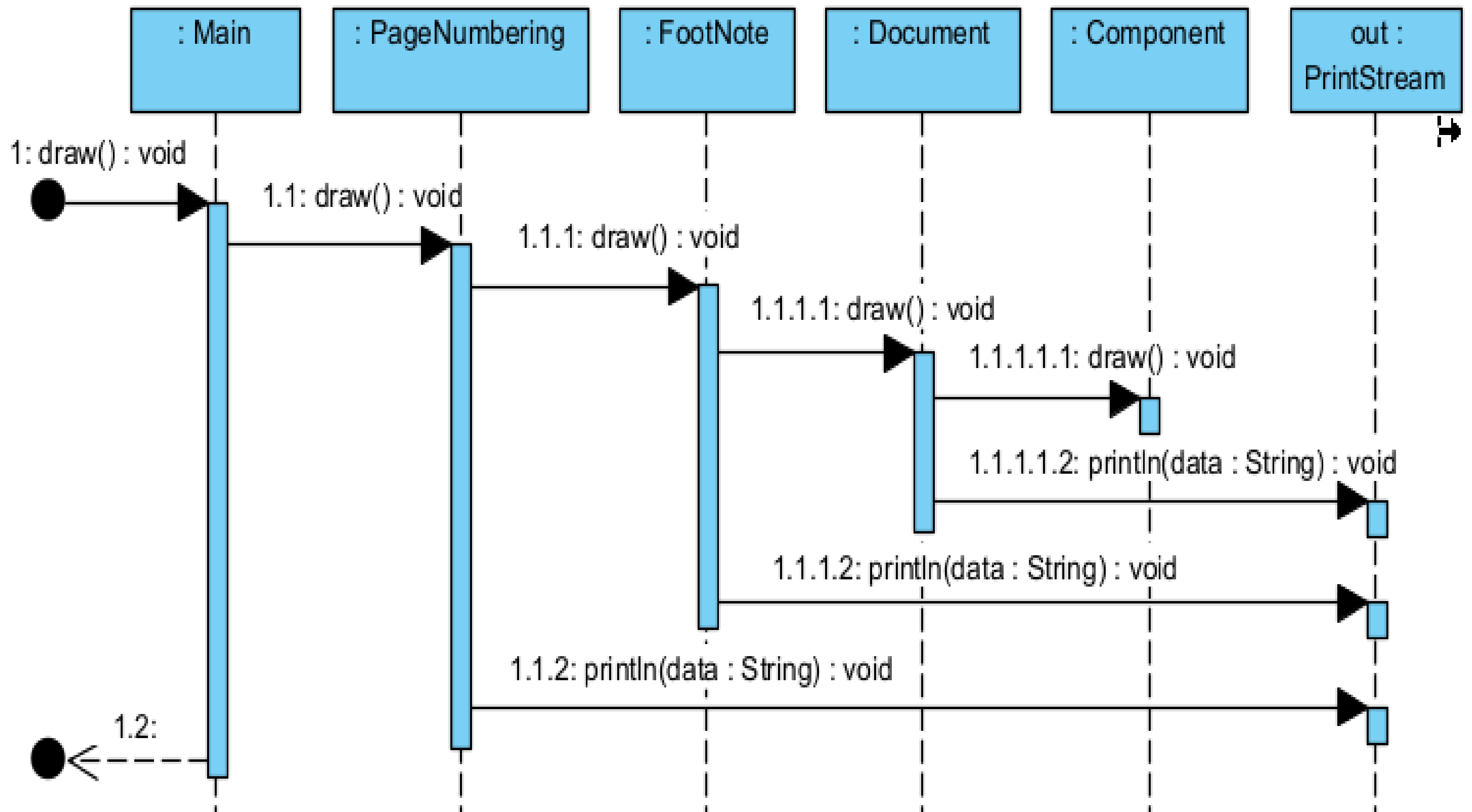


Zastosowanie dekoratora do budowy widoku dokumentu




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

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

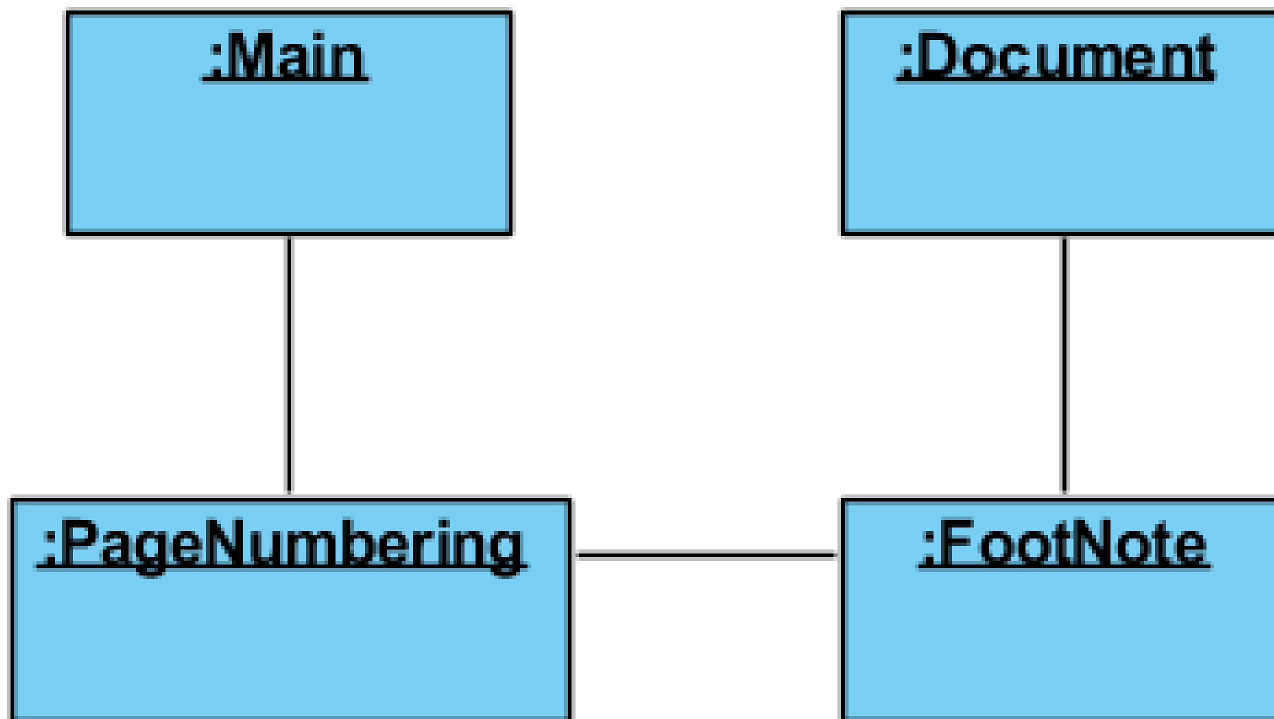


: Main
: PageNumbering
: FootNote
: Document
: Component
out : PrintStream
1: draw() : void
1.1: draw() : void
1.1.1: draw() : void
1.1.1.1: draw() : void
1.1.1.1.1: draw() : void
1.1.1.1.2: println(data : String) : void
1.1.1.2: println(data : String) : void
1.1.2: println(data : String) : void
1.2:

```java
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 ");    }
}
```

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

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

 public void draw() {
   Component document = new Document();
   FootNote footnote = new FootNote(document);
   this.document = new PageNumbering(footnote);
   this.document.draw();
}

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

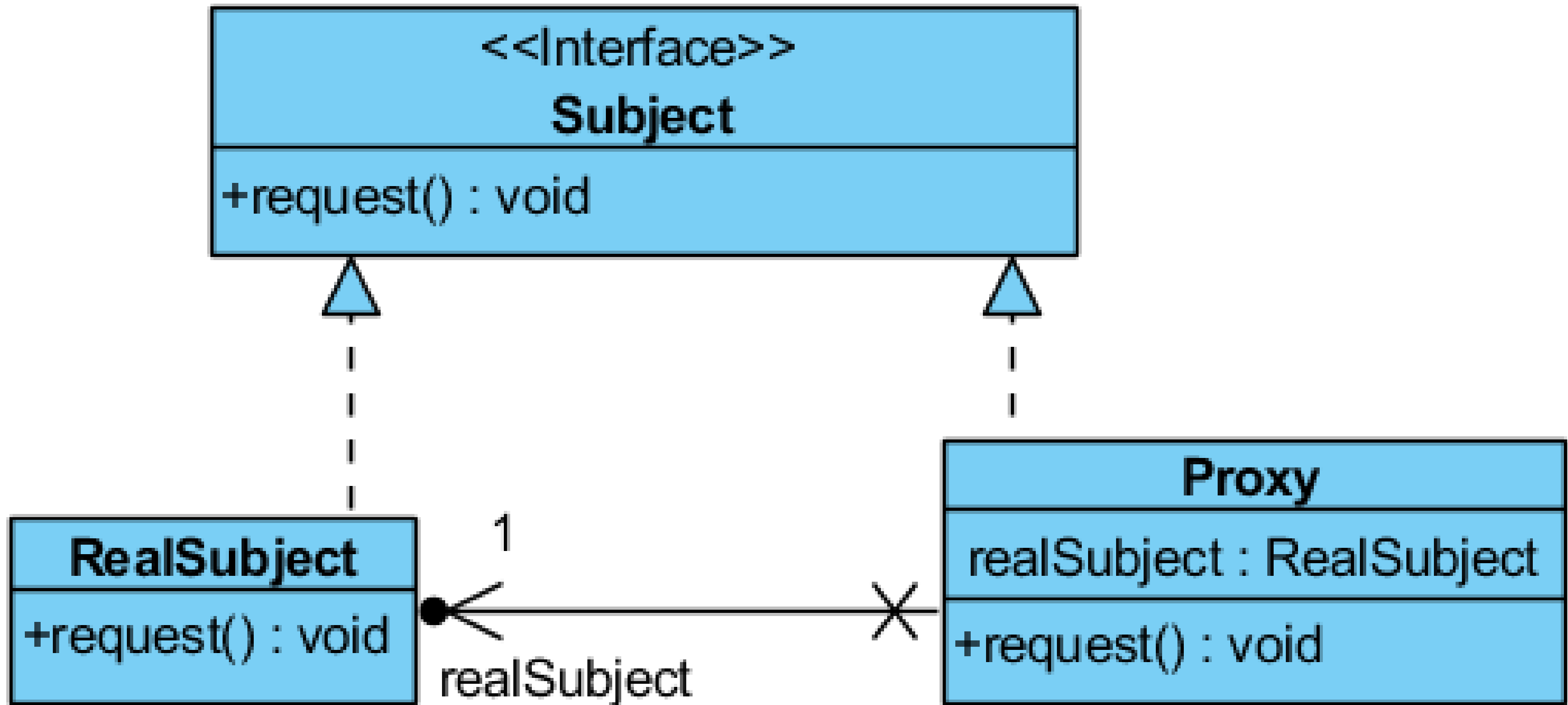
Zastosowanie dekoratora do budowy widoku dokumentu (diagram obiektów)



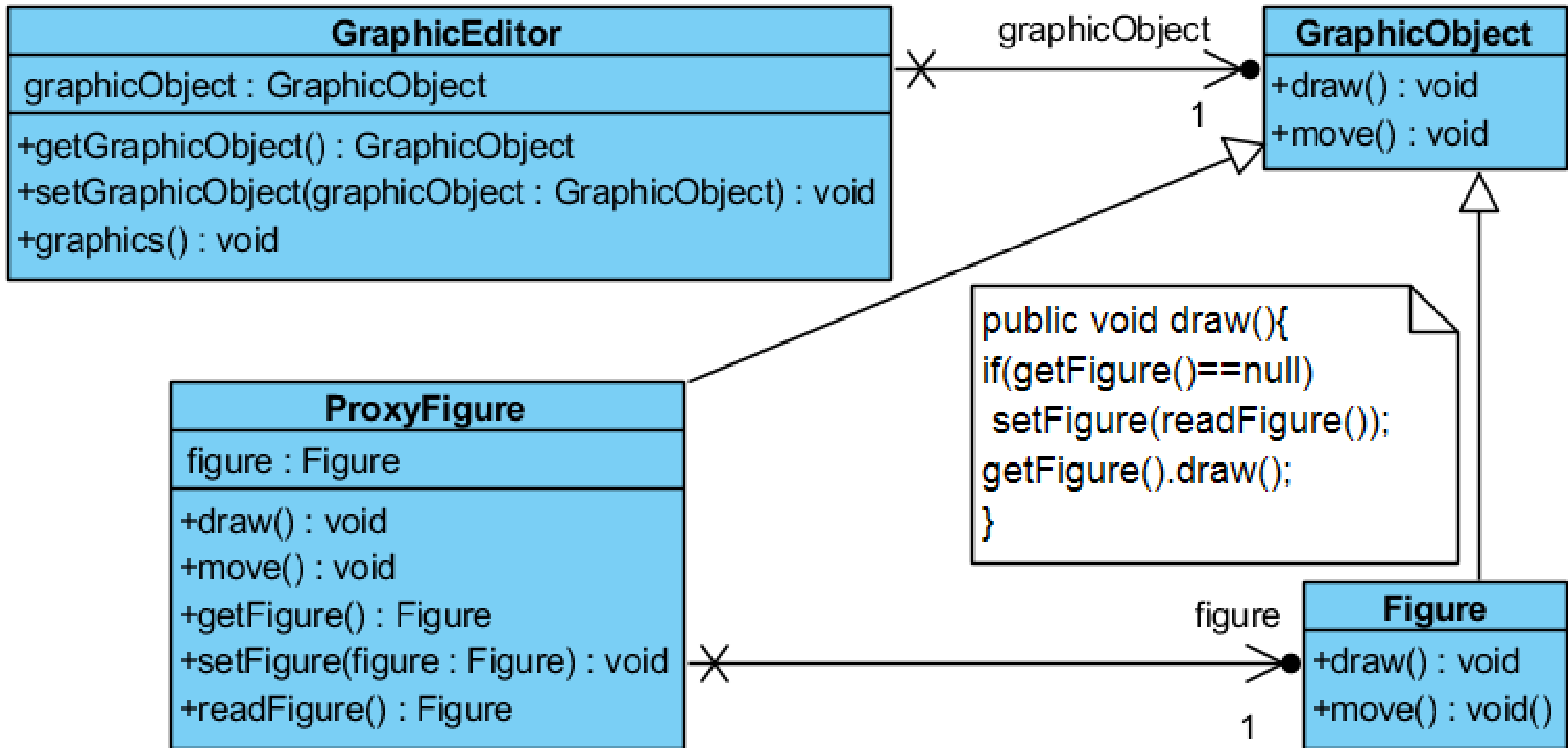
Charakterystyka wzorca *Decorator*

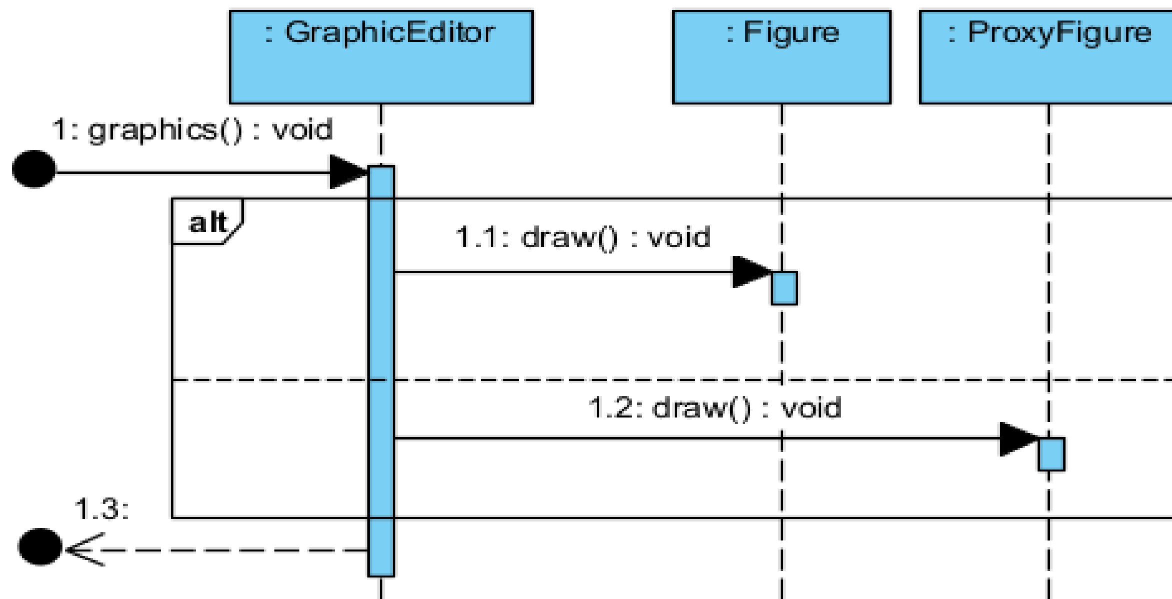
- **Problem**: Należy dynamicznie rozwijać funkcjonalność obiektu jako alternatywa dla tworzenia hierarchii klas.
- **Rozwiązanie**: Obiekt typu ***Component*** jest klasą abstrakcyjną (interfejsem) dla obiektów wizualnych. Jej interfejs definiuje operacje rysowania i obsługi zdarzeń implementowane przez klasę ***ConcreteComponent.*** Abstrakcyjna klasa (interfejs) typu ***Decorator*** dziedziczy operacje wizualne od interfejsu ***Component*** i definiuje dodatkowe operacje graficzne realizowane przez klasę ***ConcreteDecorator.***
- **Klient wzorca**: Obiekt typu ***Document*** 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 (bardziej elastyczne niż przy dziedziczeniu)
  - 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**
- **Pokrewne wzorce**: **Adapter** (jednak zmiana zobowiązań obiektu), **Composite**, **Strategy** (jednak zmiana struktury obiektu)

7) Pełnomocnik - *Proxy – wzorzec obiektowy*

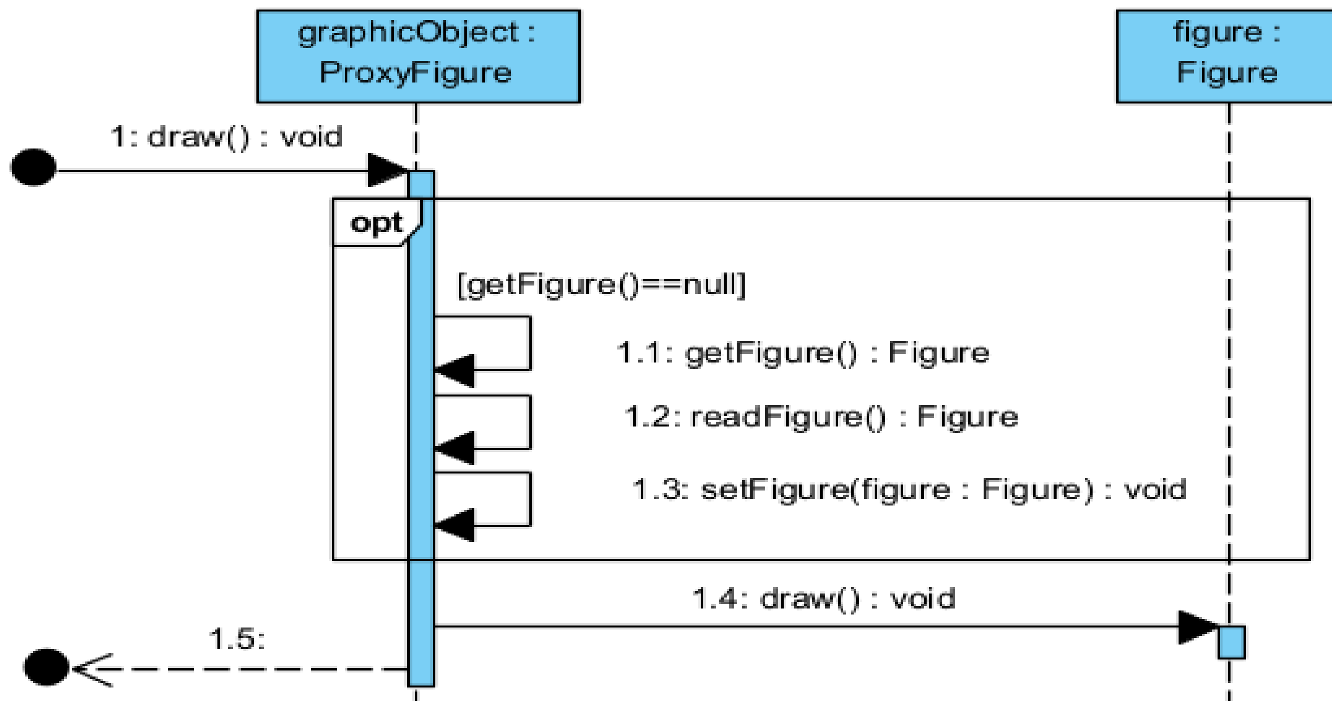


Przykład zastosowania wzorca *Proxy*





: GraphicEditor
: Figure
: ProxyFigure
1: graphics() : void
alt
1.1: draw() : void
1.2: draw() : void
1.3:



graphicObject : ProxyFigure
figure : Figure
1: draw() : void
opt
[getFigure()==null]
1.1: getFigure() : Figure
1.2: readFigure() : Figure
1.3: setFigure(figure : Figure) : void
1.4: draw() : void
1.5:

Charakterystyka wzorca *Proxy*

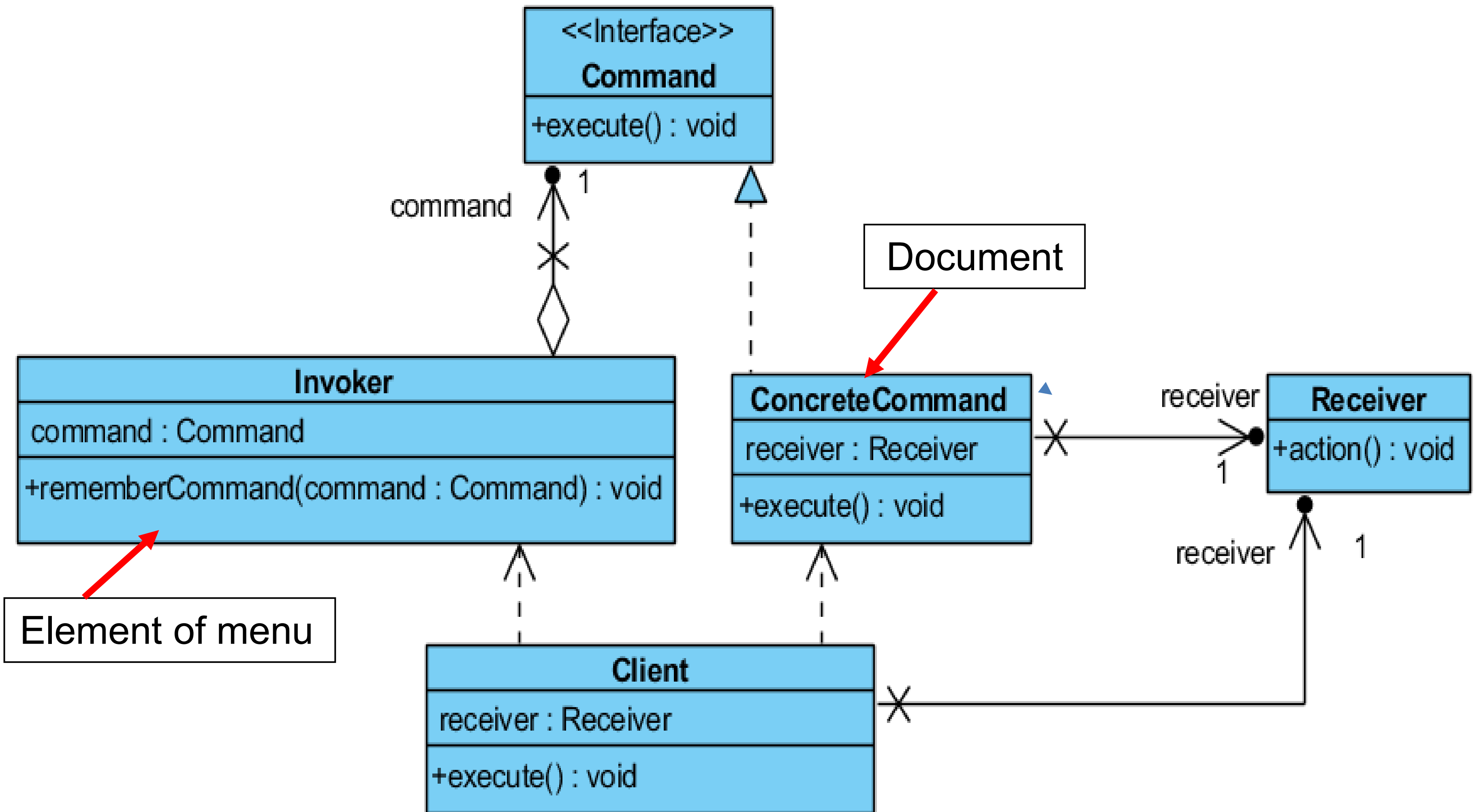
- **Problem**: reprezentuje inny obiekt w celu sterowania dostępem do niego
- **Rozwiązanie**: Obiekt typu ***Proxy*** przechowuje referencję do prawdziwego obiektu typu ***RealObject*** i może być zastąpiony przez obiekt typu ***RealObject***, ponieważ mają taki sam interfejs (dziedziczą od klasy typu ***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***
- **Pokrewne wzorce: Adapter** (jedna podzbiór interfejsu przedmiotu), **Decorator** (jednak sterowanie dostępem do obiektu)

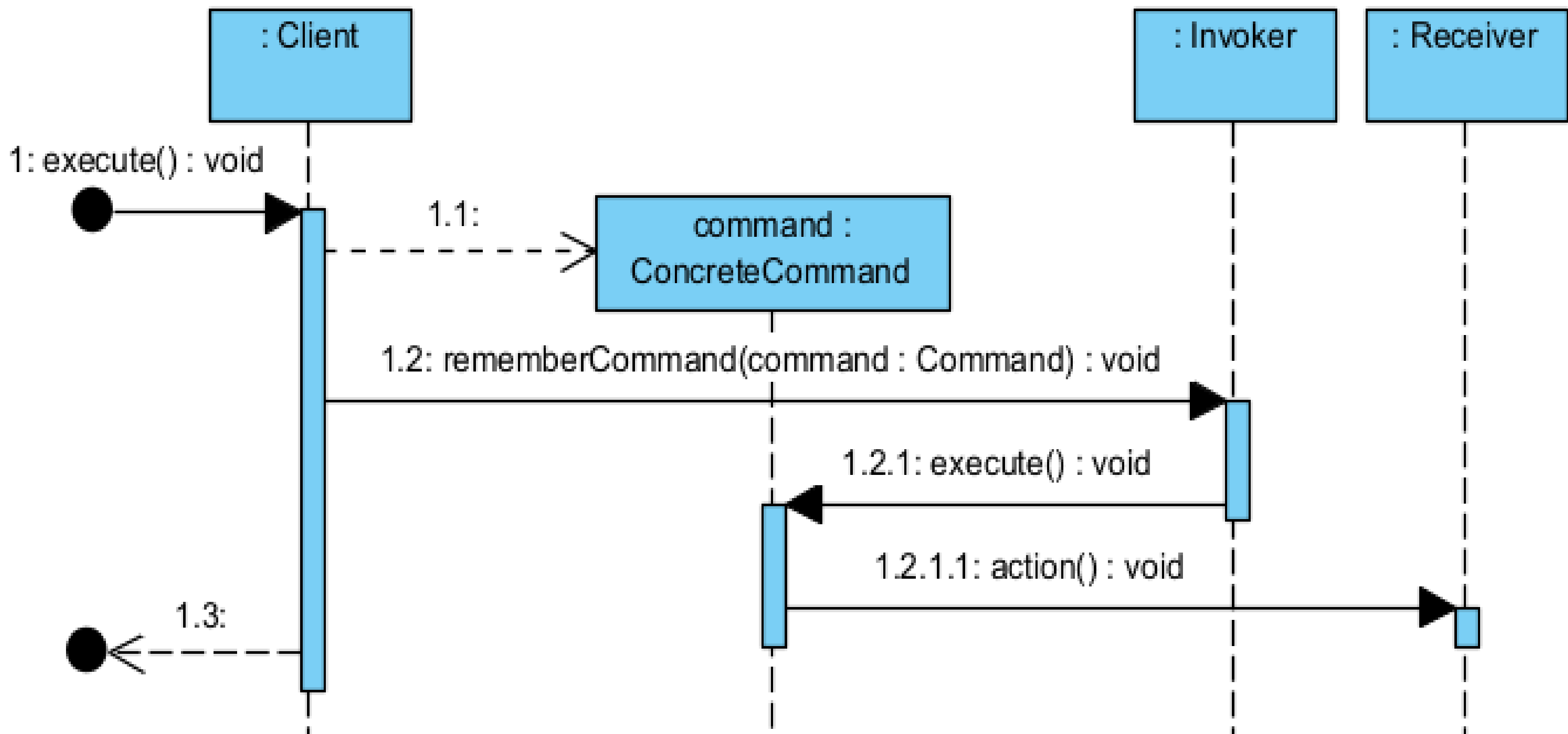
83

Dodatkowy materiał dotyczący wzorców zachowania

Wzorce czynnościowe	Aspekt, który może się zmienić
1)Chain of Responsibility	Obiekt, który może zrealizować żądanie
2)Command	**Warunki i sposób realizacji żądania**
3)Interpreter	**Gramatyka i reprezentacja języka**
4)Iterator	**Sposób dostępu i przechodzenia elementów kolekcji**
5)Mediator	**Jak i które obiekty oddziałują na siebie?**
6)Memento	**Jakie prywatne informacje są przechowywane poza obiektem i kiedy?**
7)Observer	Liczba obiektów zależących od innego obiektu; jak zależne obiekty utrzymują aktualny stan
8)State	**Stany obiektów**
9)Visitor	Operacje, które można zastosować do obiektu (obiektów) bez zmiany jego klasy (ich klas)
10)Strategy	Algorytm
11)Template Method	**Kroki algorytmu**

2) Polecenie - *Command*

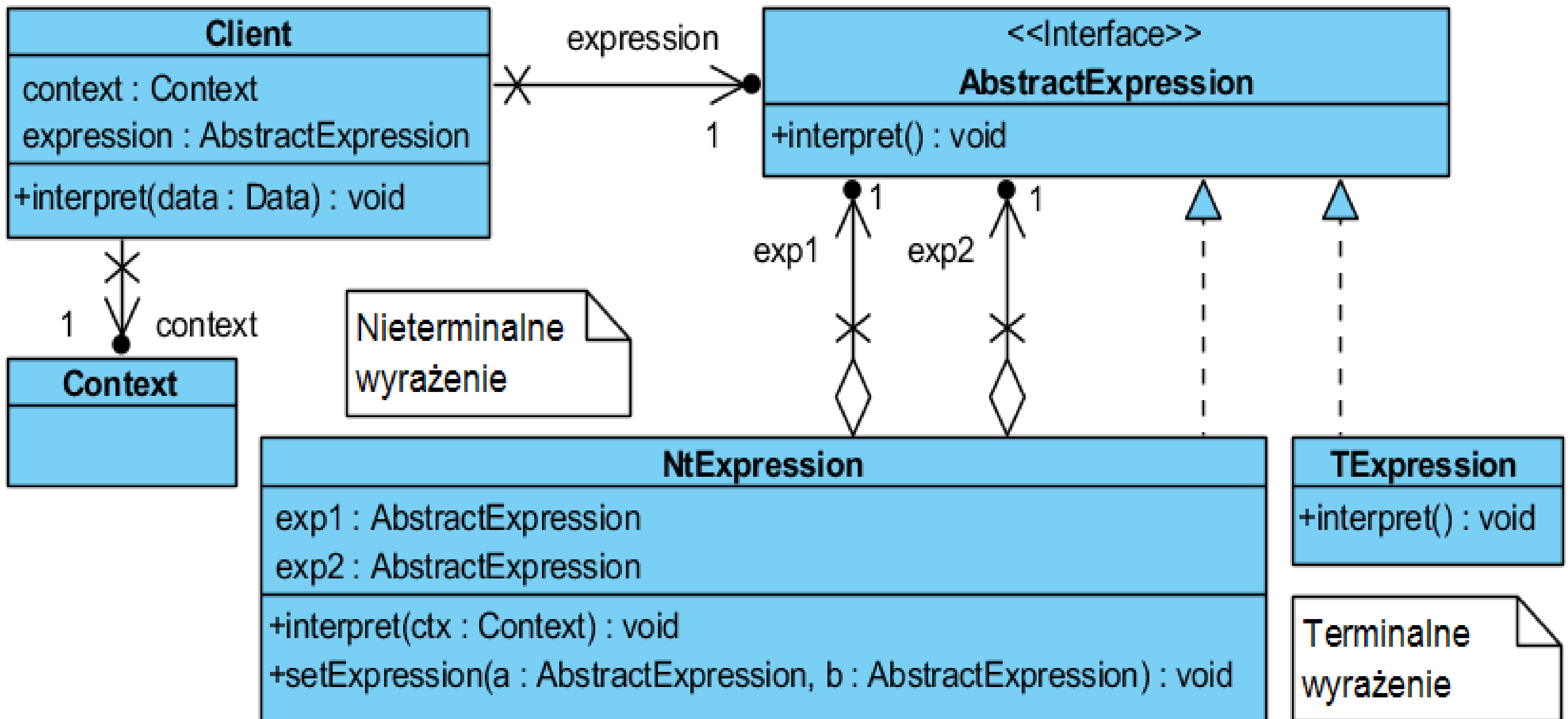



: Client
: Invoker
: Receiver
1: execute() : void
1.1:
command :
ConcreteCommand
1.2: rememberCommand(command : Command) : void
1.2.1: execute() : void
1.2.1.1: action() : void
1.3:

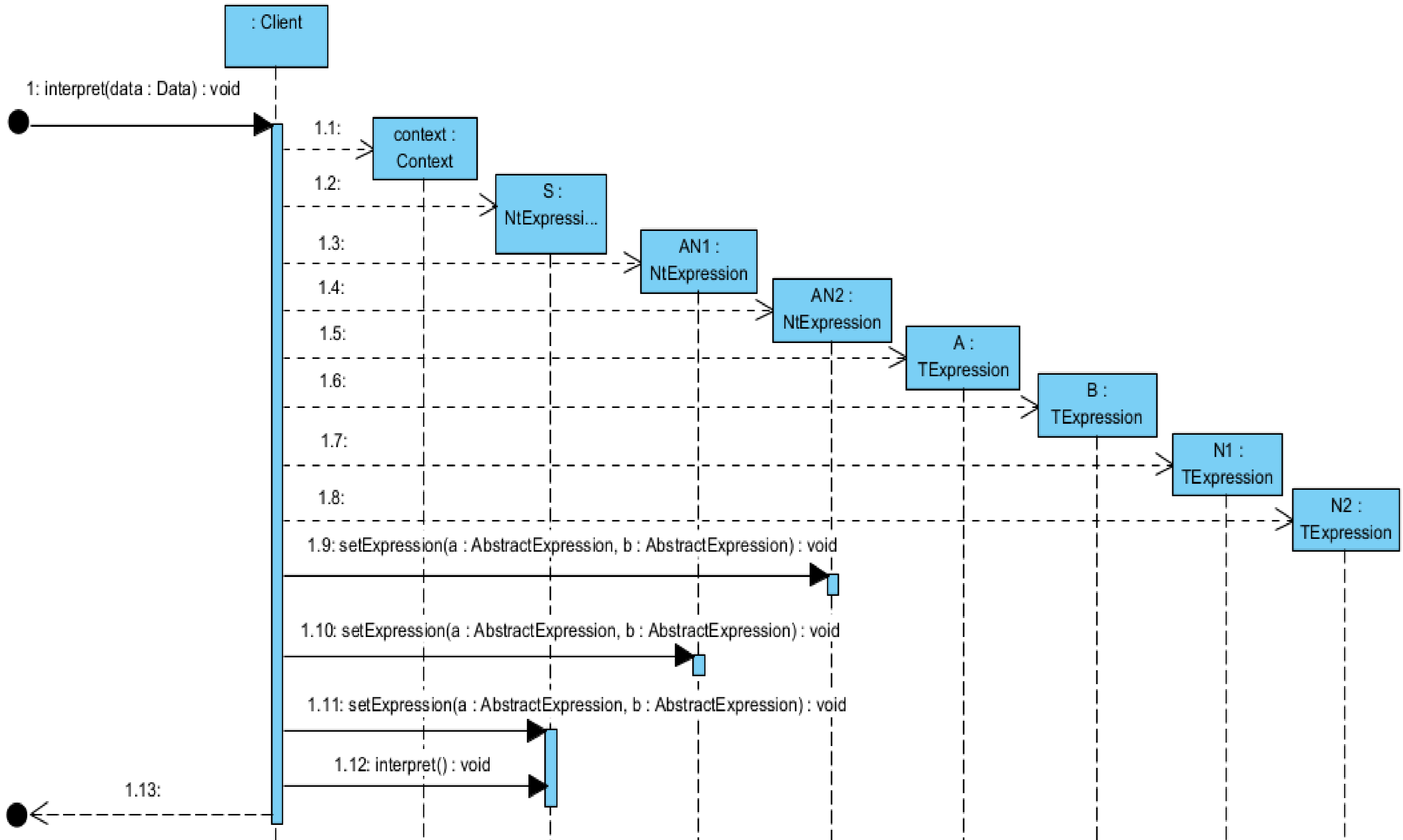
Charakterystyka wzorca *Command*

- **Problem:** Należy polecenia tworzyć w formie obiektu, co pozwala sparametryzować różne wymagania klientów i ewidencji poleceń
- **Rozwiązanie:** interfejs ***Command*** deklaruje wykonywane operacje. Obiekt typu ***ConcreteCommand*** określa związek między akcją (obiekt typu ***Invoker***) oraz obiektem odbiorcą typu ***Receiver***, i wdraża metody wykonywane przez wywołanie metod obiektu ***Receiver***. Obiekt typu ***Invoker*** za pośrednictwem **obiektu** typu ***ConcreteCommand*** wywołuje metodę obiektu typu ***Receiver*,** który wie, jak wykonać akcję.
- **Klient wzorca :** Klient tworzy obiekt typu ***ConcreteCommand*** i ustala jego obiekt typu ***Receiver***, które wykonują akcje oraz ustala obiekt zainteresowany akcją - taki jak obiekt ***Invoker***.
- **Rezultat:**
  - Separacja obiektów, które wywołuja akcję od tych, które realizują akcje.
  - Możliwość tworzenia złożonych obiektów typu ***ConcreteCommand***
  - Łatwość wstawiania nowych klas pochodnych typu ***Command***
- **Pokrewne wzorce:**
  - Kompozyt (**Composite**) – do utworzenia złożonych poleceń
  - **Memento** – do anulowania skutków polecenia
  - Działa jak **Prototype**, jeśli musi być skopiowany przed umieszczeniem na liście poleceń

3) Interpreter - *Interpreter*



Przykład zastosowania wzorca Interpreter (1)



Przykład zastosowania wzorca Interpreter (2)

Dane są:

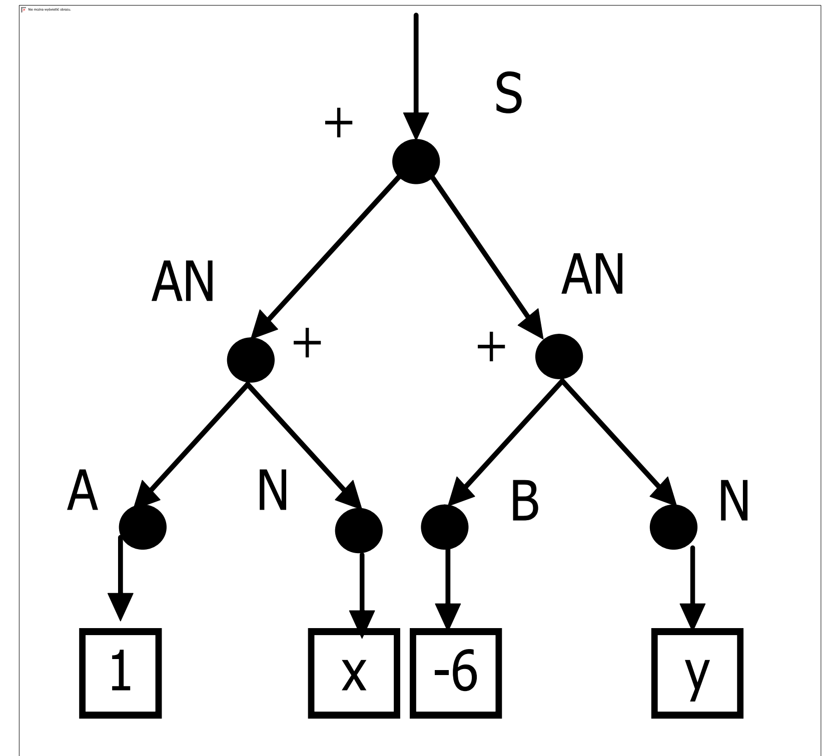
TerminalExpression = {-6, -2, 1, 5, x, y},

NonterminalExpression = {S, A, AN, B, N},

oraz produkcje w notacji **BNF**:

- <S> ::= <AN><AN>
- <AN> ::= <A><N>|<B><N>
- <N> ::= x | y
- <A> ::= 1 | -2
- <B> ::= 5 | -6

- ((1+x)+(-6+y))



Charakterystyka wzorca *Interpreter*

- **Problem**: Definicja **reprezentacji** dla gramatyki zadanego języka oraz **interpretera** zdań napisanych w danym języku definiowanym przez gramatykę

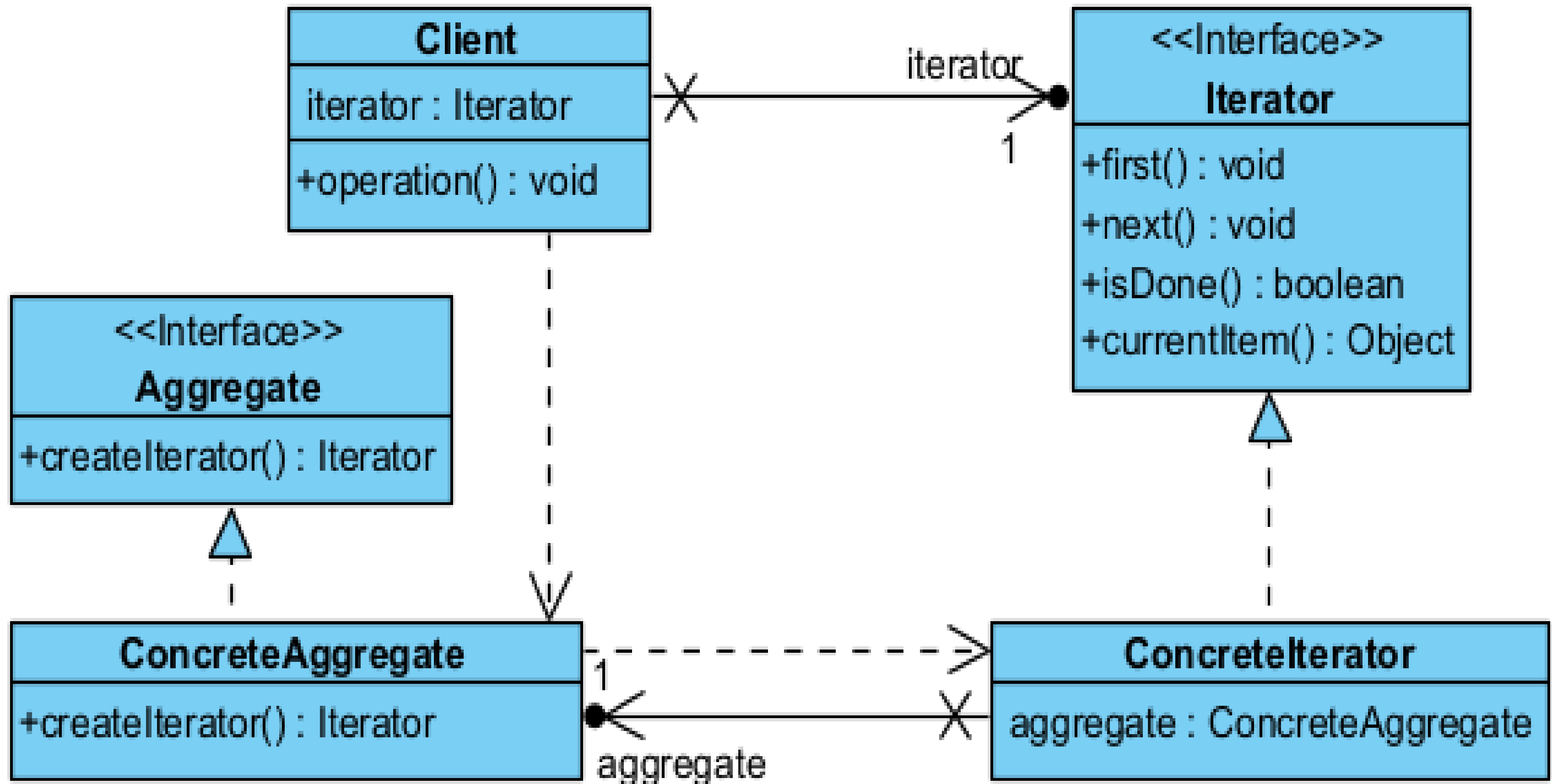
- **Rozwiązanie**: Obiekt typu ***Context*** zawiera globalne informacje dla interpretera. Obiekt typu ***Client*** buduje lub dostaje drzewo składni abstrakcyjnej reprezentujące zdanie danego języka oraz wywołuje operację **interpret** – drzewo składa się z obiektów klas ***TerminalExpression*** i ***NonterminalExpression*** implementujących interfejs ***AbstractExpression*.**

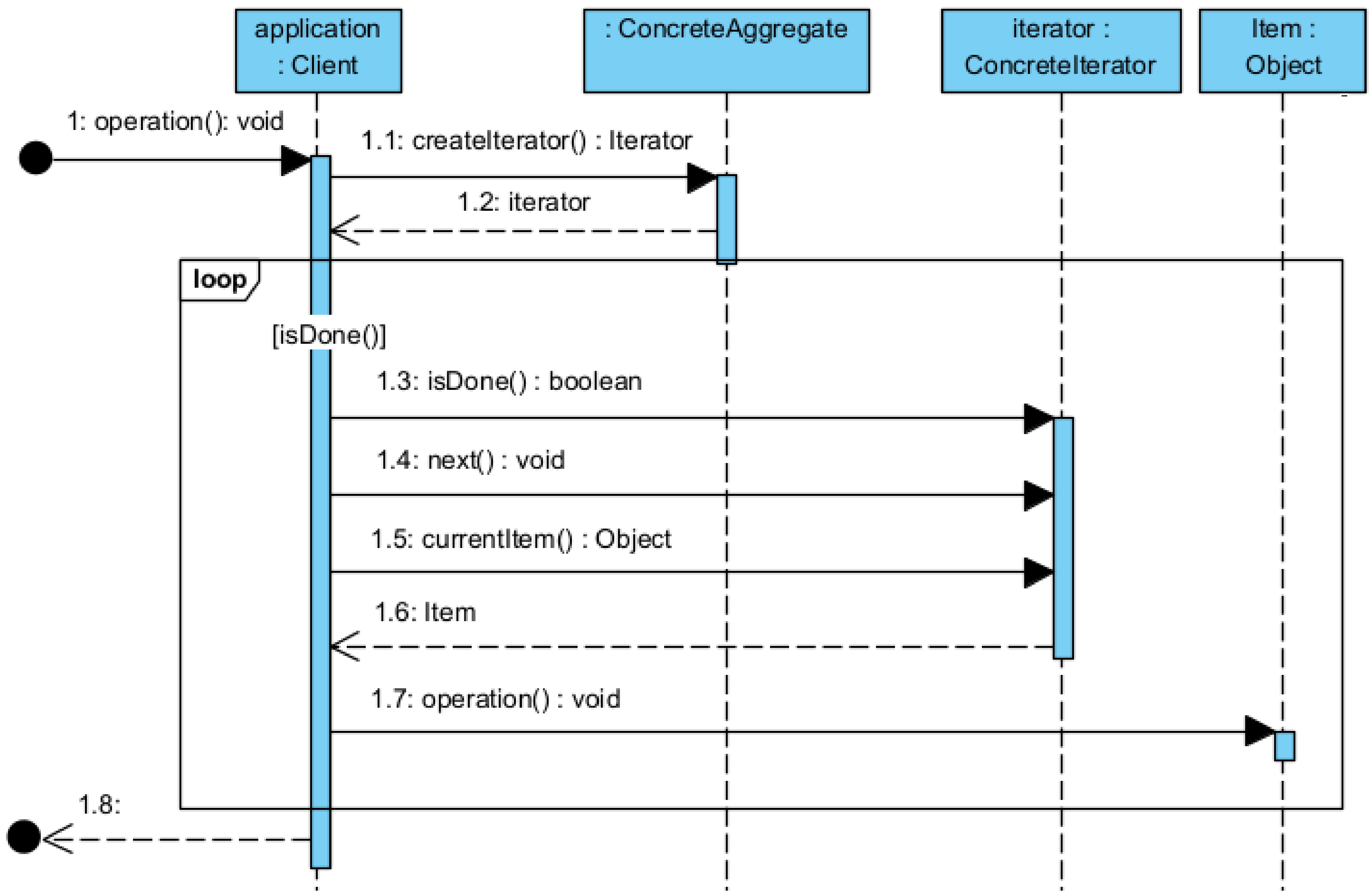
- **Klient wzorca**: Buduje lub dostaje drzewo składni oraz uruchamia proces interpretacji zdania reprezentowanego przez drzewo

- **Rezultat**:
  - Łatwa modyfikacja gramatyki
  - Łatwa implementacja gramatyki
  - Trudna obsługa złożonej gramatyki – każda klasa reprezentuje co najmniej jedną regułą produkcji
  - Dodawanie nowych sposobów interpretowania wyrażeń przez modyfikacje klas

- **Pokrewne wzorce**:
  - **Interpreter** jest przykładem zastosowania Kompozytu (**Composite**),
  - Zastosowanie wzorca Pyłek (**Flyweight**) jako symboli końcowych
  - **Iterato**r zastosowany do przechodzenia struktury
  - **Visitor** w każdym węźle drzewa może realizować działania wzorca **Interpreter**

4) Iterator - *Iterator*




application
: Client
: ConcreteAggregate
iterator :
ConcreteIterator
Item :
Object
1: operation(): void
1.1: createIterator() : Iterator
1.2: iterator
loop
[isDone()]
1.3: isDone() : boolean
1.4: next() : void
1.5: currentItem() : Object
1.6: Item
1.7: operation() : void
1.8:

Charakterystyka wzorca *Iterator*

- **Problem**: **Sekwencyjny, wielokrotny i jednakowy** dostęp do elementów obiektu zagregowanego bez podawania struktury wewnętrznej tego obiektu

- **Rozwiązanie**: Interfejs ***Iterator*** definiuje dostęp do elementów agregatów i sposób przejścia przez agregat, obiekt typu ***ConcreteIterator*** implementuje ***Iterator***. Interfejs ***Aggregate*** reprezentuje obiekt (agregat), przez który przechodzi ***Iterator***. Istnieje powiązanie jedynie między obiektami typu ***ConcreteAggregate*** i ***ConcreteIterator*.**

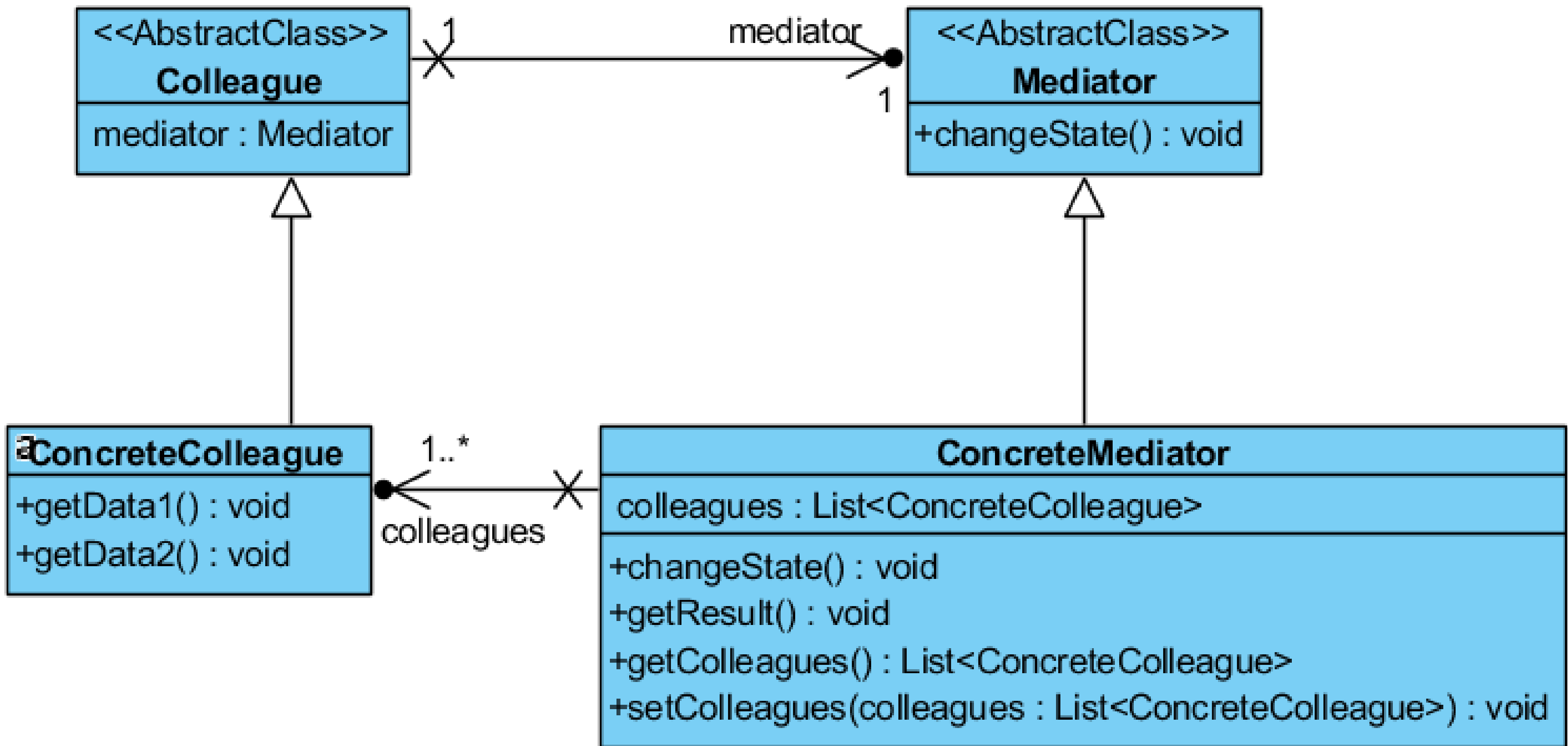
- **Klient wzorca**: klient wzorca może śledzić, który obiekt w agregacie jest bieżący i potrafi wskazać następny lub poprzedni obiekt w tym agregacie

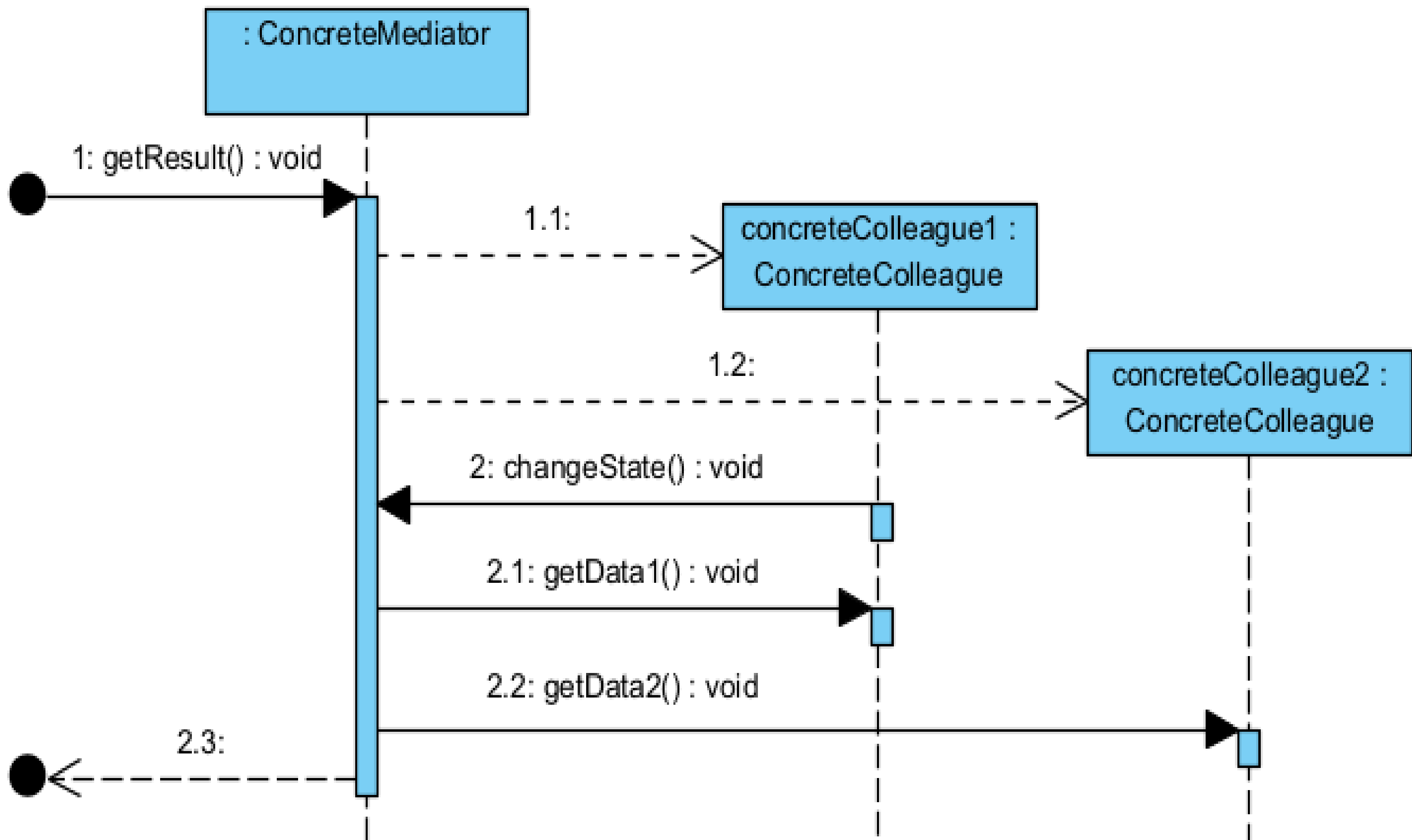
- **Rezultat**:
  - Możliwość dowolnego przejścia przez agregat
  - Interfejs iteratora upraszcza interfejs agregatu
  - W danej chwili może odbywać się wiele przejść przez agregat za pomocą iteratora

- **Implementacja**: nowa klasy typu „Control” np. klasy ***Iterator*** oraz ***ListIterator*** w pakiecie **java.util**

- **Pokrewne wzorce**
  - Kompozyt (**Composite**)
  - Metody wytwórcze (**Factory Method**) – tworzą egzemplarze podklas klasy **Iterator**
  - Pamiątka (**Memento**) do przechowania stanu obiektów typu **Iterator**

5) Mediator - *Mediator*




: ConcreteMediator
1: getResult() : void
1.1:
concreteColleague1 : ConcreteColleague
1.2:
concreteColleague2 : ConcreteColleague
2: changeState() : void
2.1: getData1() : void
2.2: getData2() : void
2.3:

Charakterystyka wzorca *Mediator*

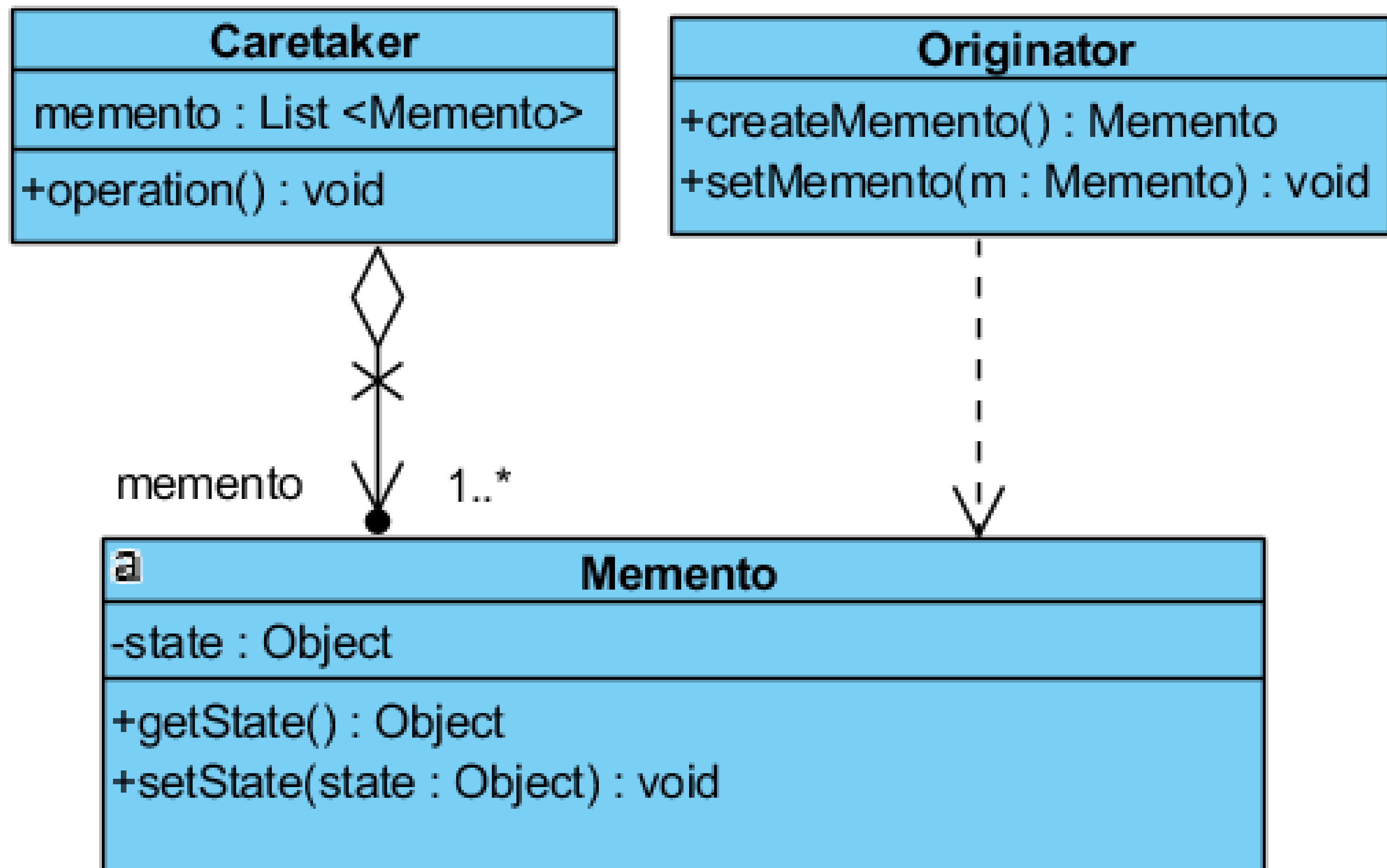
- **Problem:** Należy ograniczyć znajomość złożonych powiązań pomiędzy obiektami, które oddziałują w złożony sposób i pozwolić na zmianę sposobu komunikowania się bez konieczności definiowania nowych podklas.
  Rozwiązanie:
  - Obiekt typu ***Mediator*** definiuje interfejs porozumiewania się z obiektami typu ***Collegue***, więc każdy obiekt ***ConcreteColleague*** zna operacje obiektu typu ***ConcreteMediator*** - każdy obiekt typu ***ConcreteColleague*** nie musi komunikować się z innym obiektem ***ConcreteColleaque***, ale z obiektem typu ***ConcreteMediator***; obiekt typu ***ConcreteMediator*** koordynuje współpracę wielu obiektów typu ***ConcreteColleague*.**
  - Obiekty typu **Co*ncreteColleague*** wysyłają żądania do obiektu typu ***ConcreteMediator***, a obiekt typu ***ConcreteMediator*** podejmuje decyzję i wysyła te wnioski do odpowiednich obiektów typu ***ConcreteColleague***.

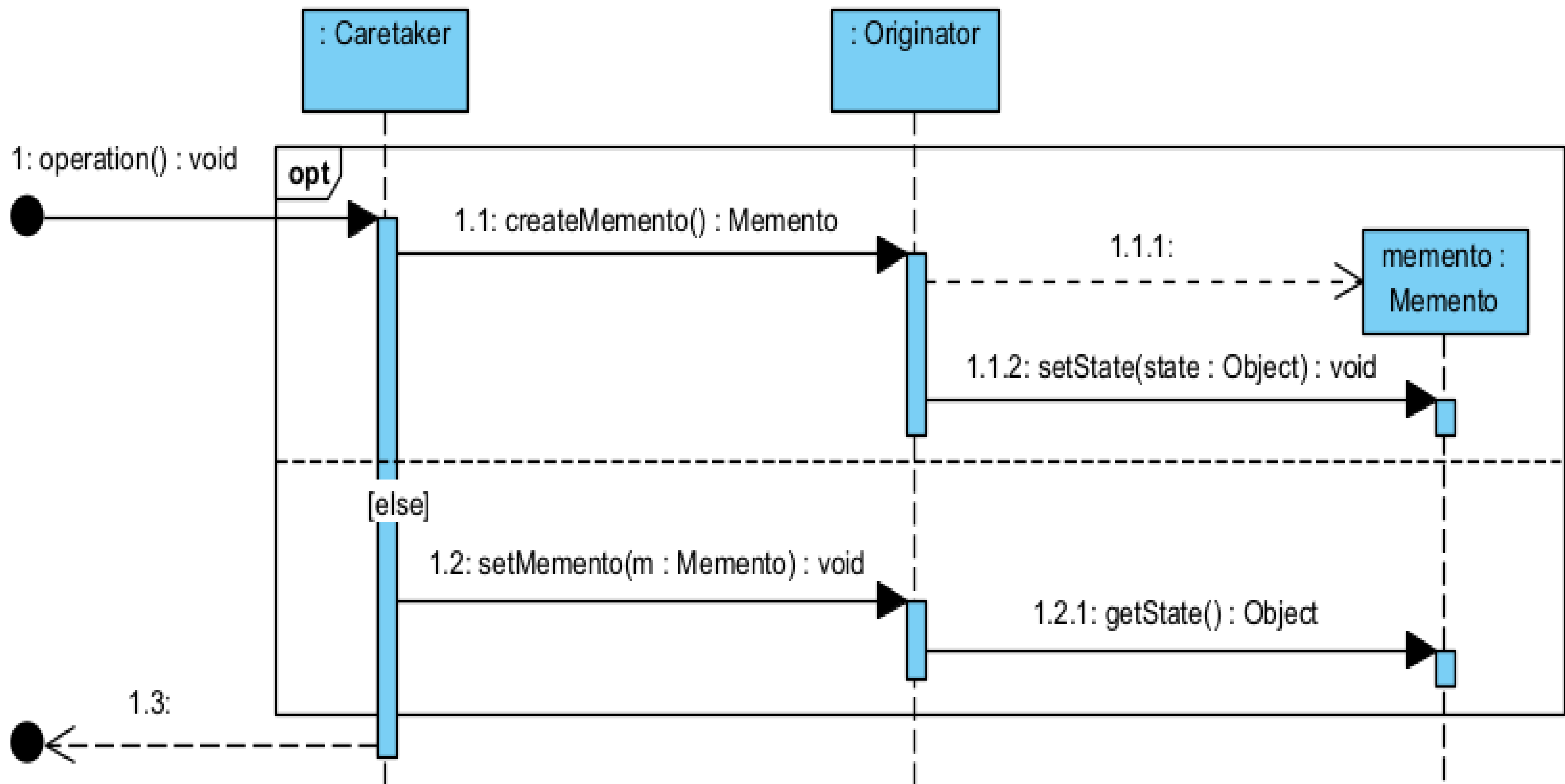
- **Rezultat:**

  – Obiekt typu ***ConcreteMediator*** skupia zachowania, które byłyby umieszczone w wielu obiektach typu ***ConcreteColleague***
  – Można łączyć różne rodzaje obiektów typów ***Concrete*****Colleague** i ***ConcreteMediator***
  – Uproszczenie protokołów komunikacji za pomocą związków jeden-do-wielu między obiektami typów ***ConcreteMediator*** i ***ConcreteColleague***, zastępując wiele obiektów typu ***Concrete*****Mediator** na jeden.
  – Uogólnienie przez interfejs ***Mediator*** współpracy między obiektami, które implementują interfejs ***Collegue***
  – Funkcjonalność obiektów, które implementują interfejs ***Mediator*** może prowadzić do bardzo złożonych rozwiązań, które będą trudne do utrzymania

- **Pokrewne wzorce:**
  - Obserwator (**Observer**) może służyć do komunikacji obiektów implementujących **Mediator** z obiektami typu **ConcreteColleague**

6) Pamiątka - *Memento*

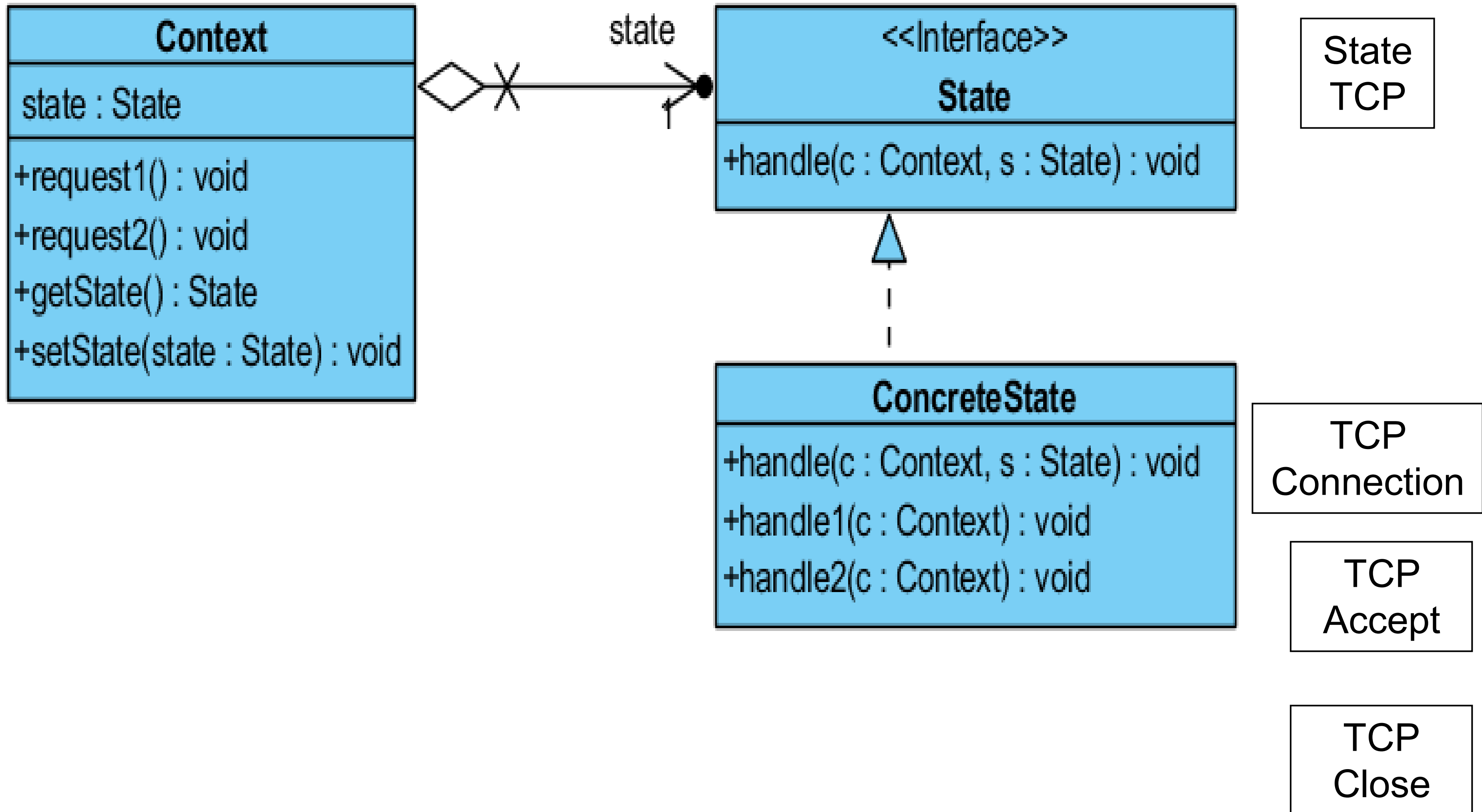


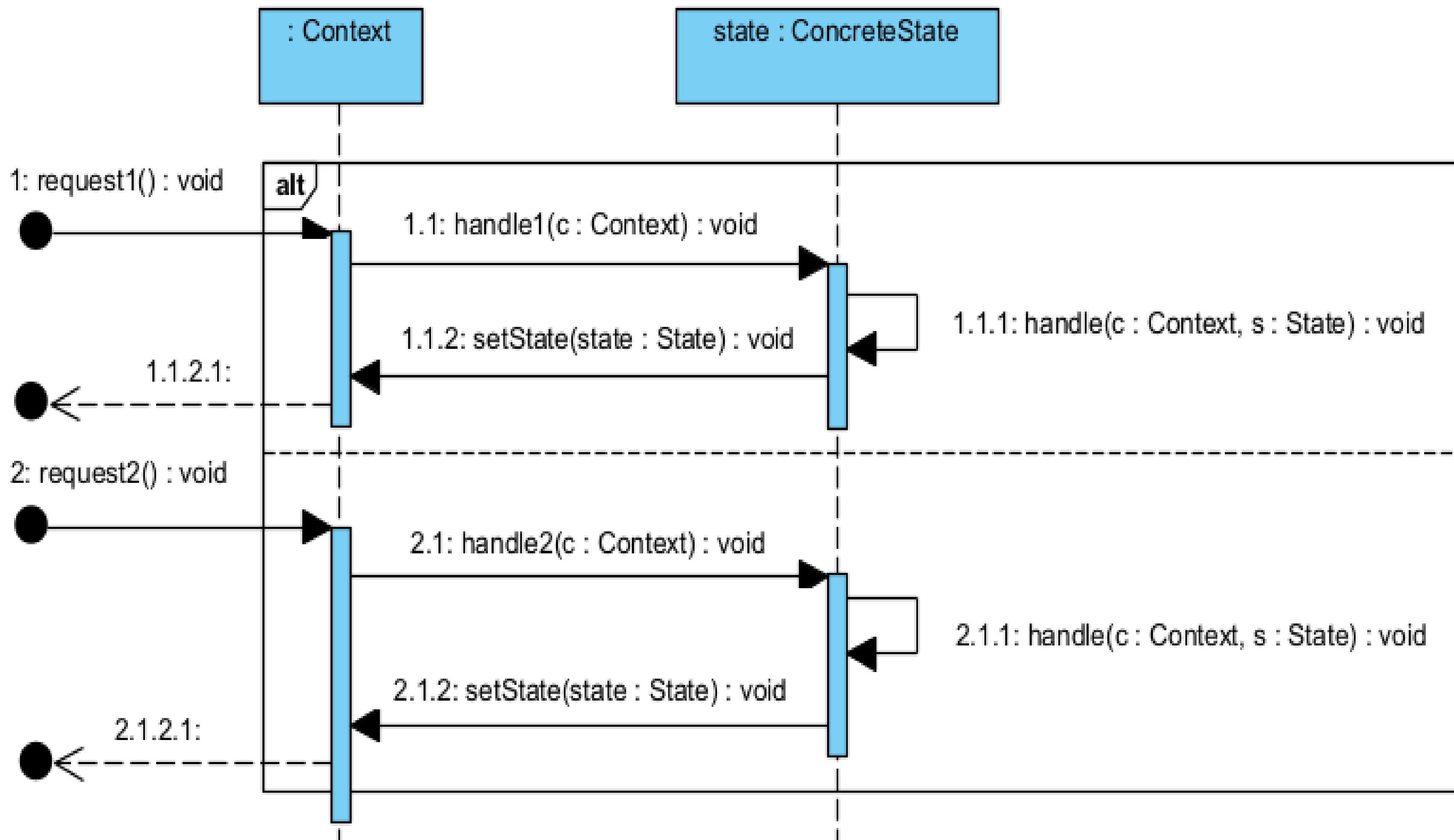

: Caretaker
: Originator
1: operation() : void
opt
1.1: createMemento() : Memento
1.1.1:
memento : Memento
1.1.2: setState(state : Object) : void
[else]
1.2: setMemento(m : Memento) : void
1.2.1: getState() : Object
1.3:

Charakterystyka wzorca *Memento*

- **Problem:** Bez naruszania hermetyzacji należy zapamiętać i udostępnić stan obiektu, aby przywrócić jego stan w przyszłości.
- **Rozwiązanie:** Obiekt typu ***Caretaker*** jest odpowiedzialny za kierowanie obiektami typu ***Originator*** - może dać polecenie obiektom typu ***Originator***, któych stan powinien być zachowany, aby utworzyły swój obiekt typu ***Memento*** i go przekazały. Obiekt typu ***Memento*** posiada stan obiektu wytwórcy (typu ***Originator***). Obiekt typu ***Caretaker*** przechowuje wszystkie obiekty typu ***Memento*** i może dać dać polecenie obiektom typu ***Originator***, aby przywróciły swój stan przekazując im ich obiekt typu ***Memento***.
- **Rezultat:**
  - Utrzymanie hermetyzacji obiektów typu ***Originator***, mimo przechowywania stanu tego obiektu poza nim
  - Uproszczenie obiektu typu ***Caretaker***
  - Zmniejszenie wydajności
  - Trudności w realizacji
  - Trudności w utrzymaniu obiektów Memento
- **Pokrewne wzorce:**
  - Wzorce Polecenie (**Command**) i **Iterator** mogą używać wzorca **Memento** do zapamiętania stanu

8) Stan - *State*

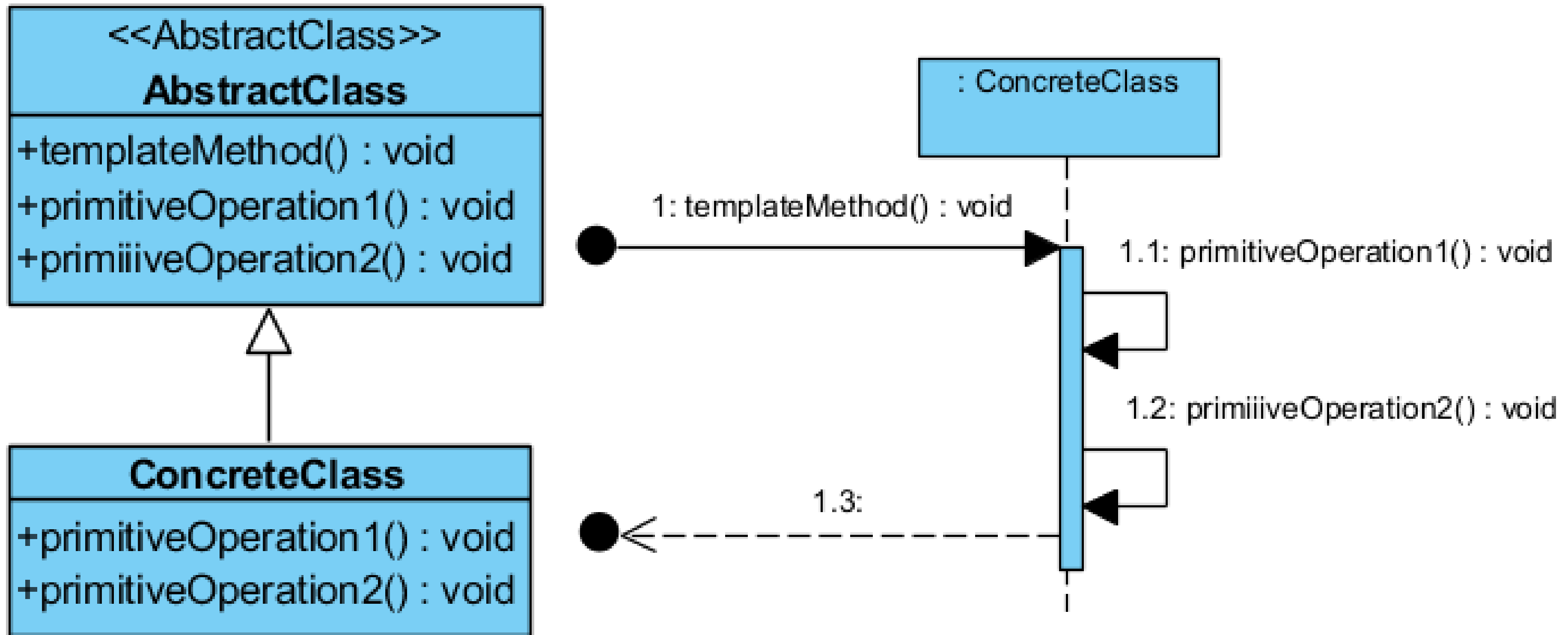



: Context
state : ConcreteState
alt
1: request1() : void
1.1: handle1(c : Context) : void
1.1.1: handle(c : Context, s : State) : void
1.1.2: setState(state : State) : void
1.1.2.1:
2: request2() : void
2.1: handle2(c : Context) : void
2.1.1: handle(c : Context, s : State) : void
2.1.2: setState(state : State) : void
2.1.2.1:

Charakterystyka wzorca *State*

- **Problem:** Można zmienić zachowanie obiektu podczas zmiany wewnętrznego stanu obiektu, tworząc jego pochodny obiekt.
- **Rozwiązanie:** Obiekt typu ***Context*** definiuje interfejs dla klientów i utrzymuje obiekt typu ***ConcreteState*,** definiujący bieżący stan. Definicja interfejsu ***State*** służy do zachowania stanu obiektu typu ***Context***. Obiekty typu ***ConcreteState*** stanowią zachowanie jednego z możliwych stanów obiektu typu ***Context***.
- **Klient wzorca:** obiekt, który konfiguruje obiekty ***ConcreteState*** za pomocą obiektu ***Context***
- **Rezultat:**
  - Lokalizacja informacji zwiazanej z każdym ze stanów obiektów typu ***Context*** w jednym obiekcie typu ***ConcreteState***
  - Każdy obiekt pochodny typu ***ConcreteState*** wprowadza nową funkcjonalność niezaleznie, co poprawia widoczność przejścia pomiędzy stanami
  - Możliwość współdzielenia takich obiektów jak typu ***ConcreteState***, ponieważ ich stany są reprezentowane przez ich rodzaje (podobnie jak wzorzec ***Flyweight***).
- **Pokrewne wzorce:**
  - Wzorzec Pyłek (**Flyweight**) - współdzielenie obiektów typu **State**
  - Obiekty typu **State** są często reprezentowane przez wzorzec **Singleton**

11) Metoda szablonowa – *Template Method*



- **Problem:** Należy określić szkielet algorytmu, a szczegóły algorytmu powierzyć klasom pochodnym
- **Rozwiązanie:** Klasa ***AbstractClass*** definiuje abstrakcyjny algorytm, ale pewne części abstrakcyjnego algorytmu są uzupełniane przez różne definicje, realizowane przez metody klasy ***ConcreteClass***
- **Wynik:** "zasada Hollywood " (nie zadzwoń do nas, my zadzwonimy do Ciebie), gdzie metoda klasy bazowej wywołuje metody z klas pochodnych
- **Realizacja:** tworzenie bibliotek, co daje podstawę do wspólnych zachowań w klasach biblioteki
- **Pokrewne wzorce:**
  - Metody szablonowe (**Template Method**) wywołują metody wytwórcze (**Factory Method**)
  - Metody szablonowe (**Template Method**) służą do zmiany części algorytmu we wzorcach Strategia (**Strategy**)

Dziękuję za uwagę