Wzorce oprogramowania Gof (cd)

(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 (c.d.)

2.3. Wzorce czynnościowe

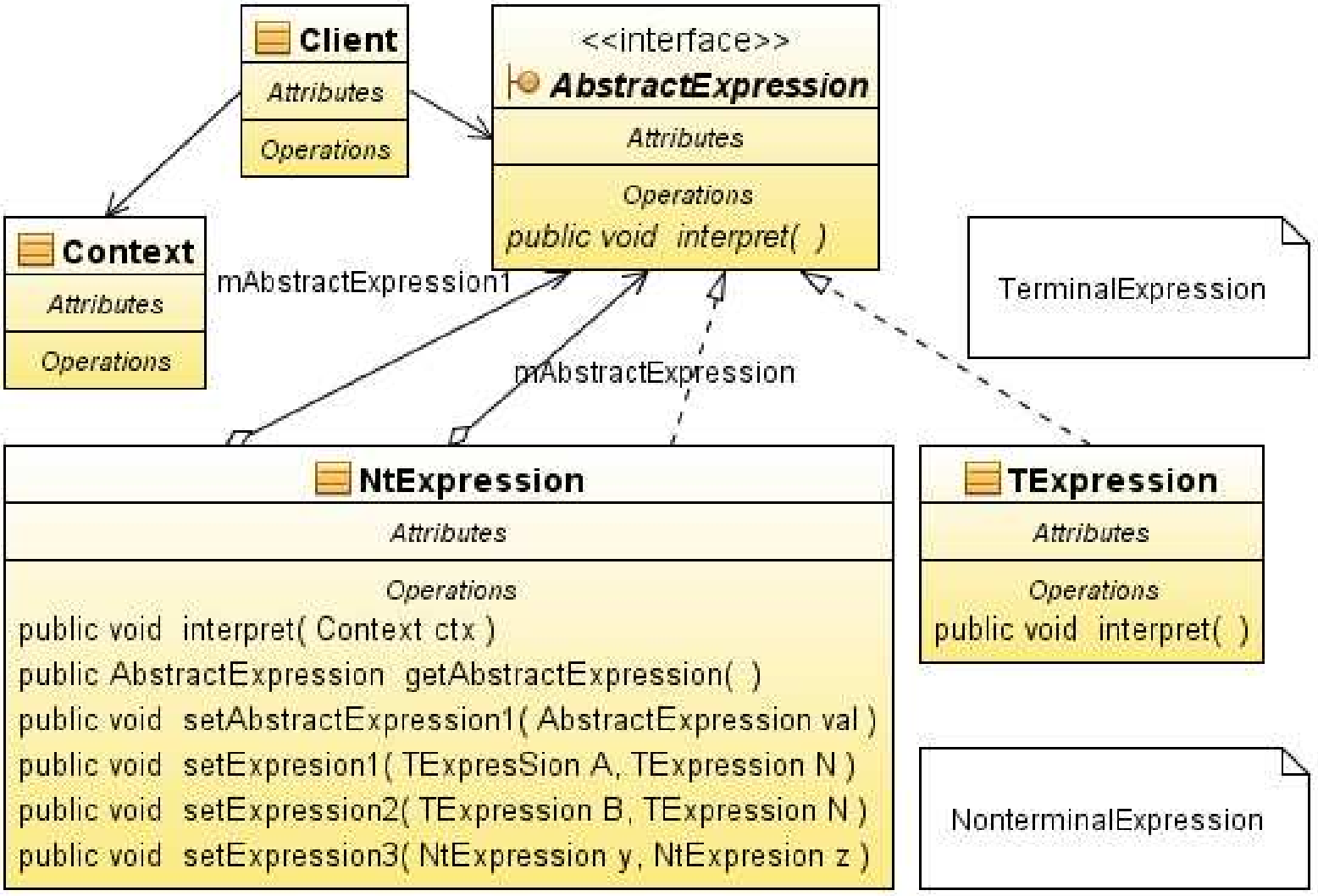
Przydzielają algorytmy i zobowiązania obiektom, obejmują wzorce obiektów, klas oraz komunikacji między obiektami

1) Interpreter
2) Iterator
3) Łańcuch zobowiązań – *Chain of Responsibility*
4) Mediator
5) Metoda szablonowa – *Template method*
6) Obserwator - *Observer*
7) Odwiedzający - Visitor
8) Pamiątka - *Memento*
9) Polecenie - *Command*
10) Stan - *State*
11) Strategia - *Strategy*

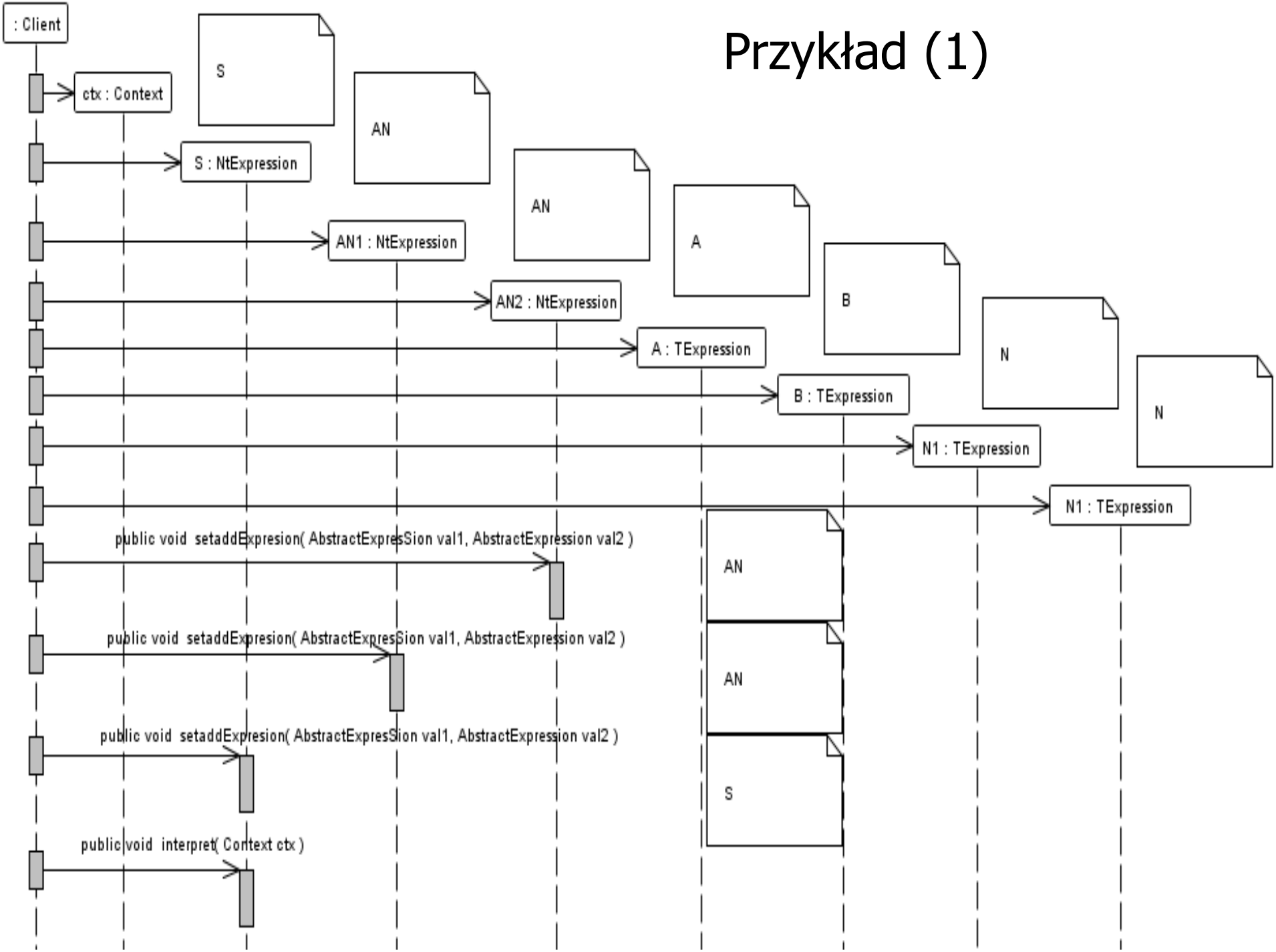
Wybór wzorca strukturalnego

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)Strategy	Algorytm
10)Visitor	Operacje, które można zastosować do obiektu (obiektów) bez zmiany jego klasy (ich klas)
11)Template Method	Kroki algorytmu

1) Interpreter - *Interpreter*



Przykład (1)


: Client
ctx : Context
S
AN
S : NtExpression
AN
AN1 : NtExpression
A
B
AN2 : NtExpression
N
A : TExpression
N
B : TExpression
N1 : TExpression
N1 : TExpression
public void setaddExpresion(AbstractExpresSion val1, AbstractExpression val2)
AN
public void setaddExpresion(AbstractExpresSion val1, AbstractExpression val2)
AN
public void setaddExpresion(AbstractExpresSion val1, AbstractExpression val2)
S
public void interpret(Context ctx)

Przykład (2)

Dane są:

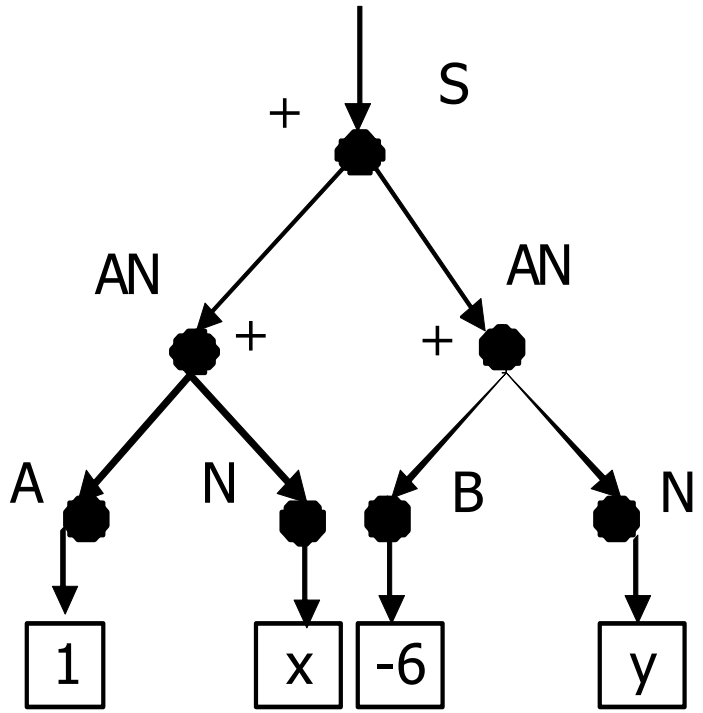
TerminalExpression = {-6, -2, 1, 5, x, y},

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))



- **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

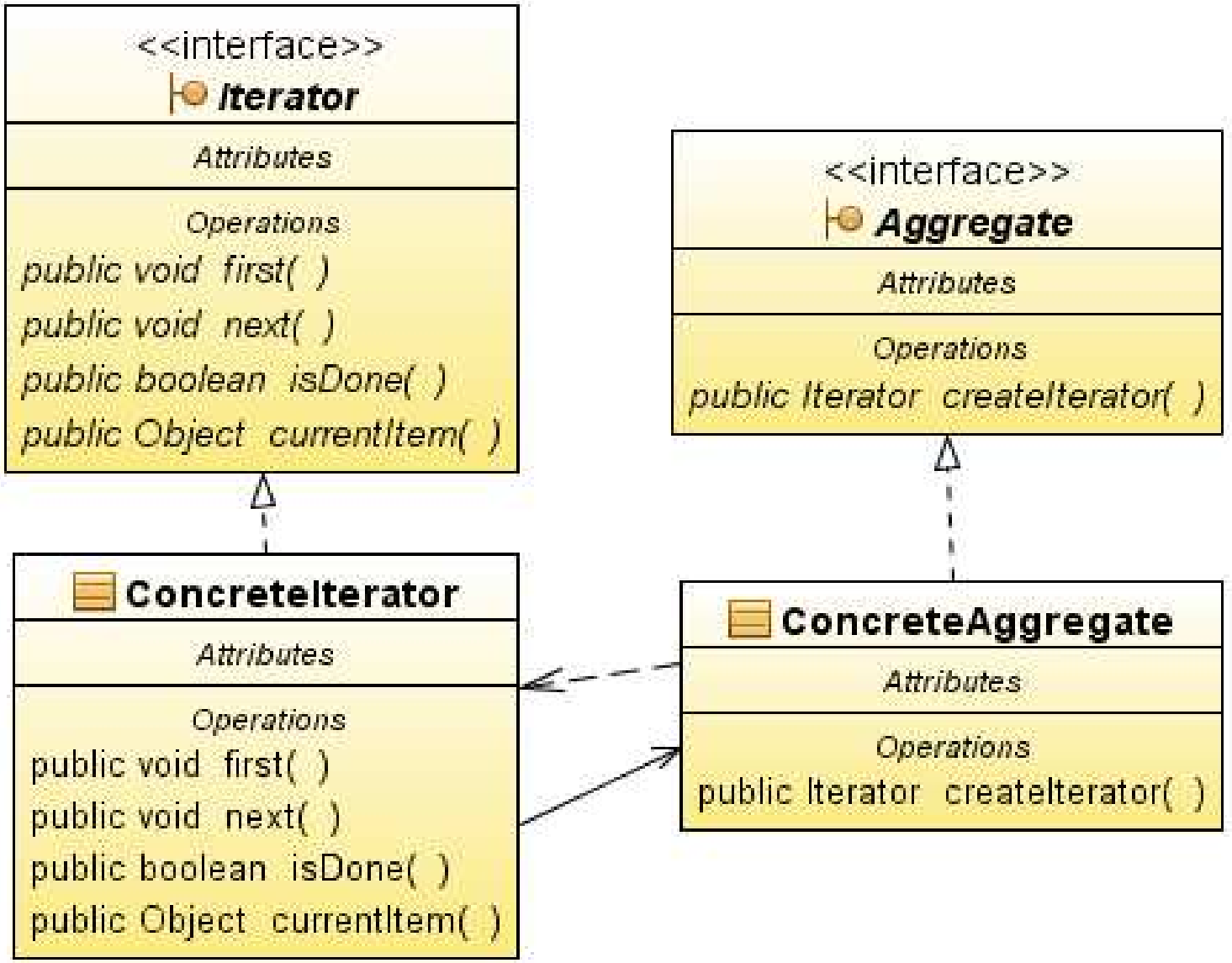
•**Rezultat**:

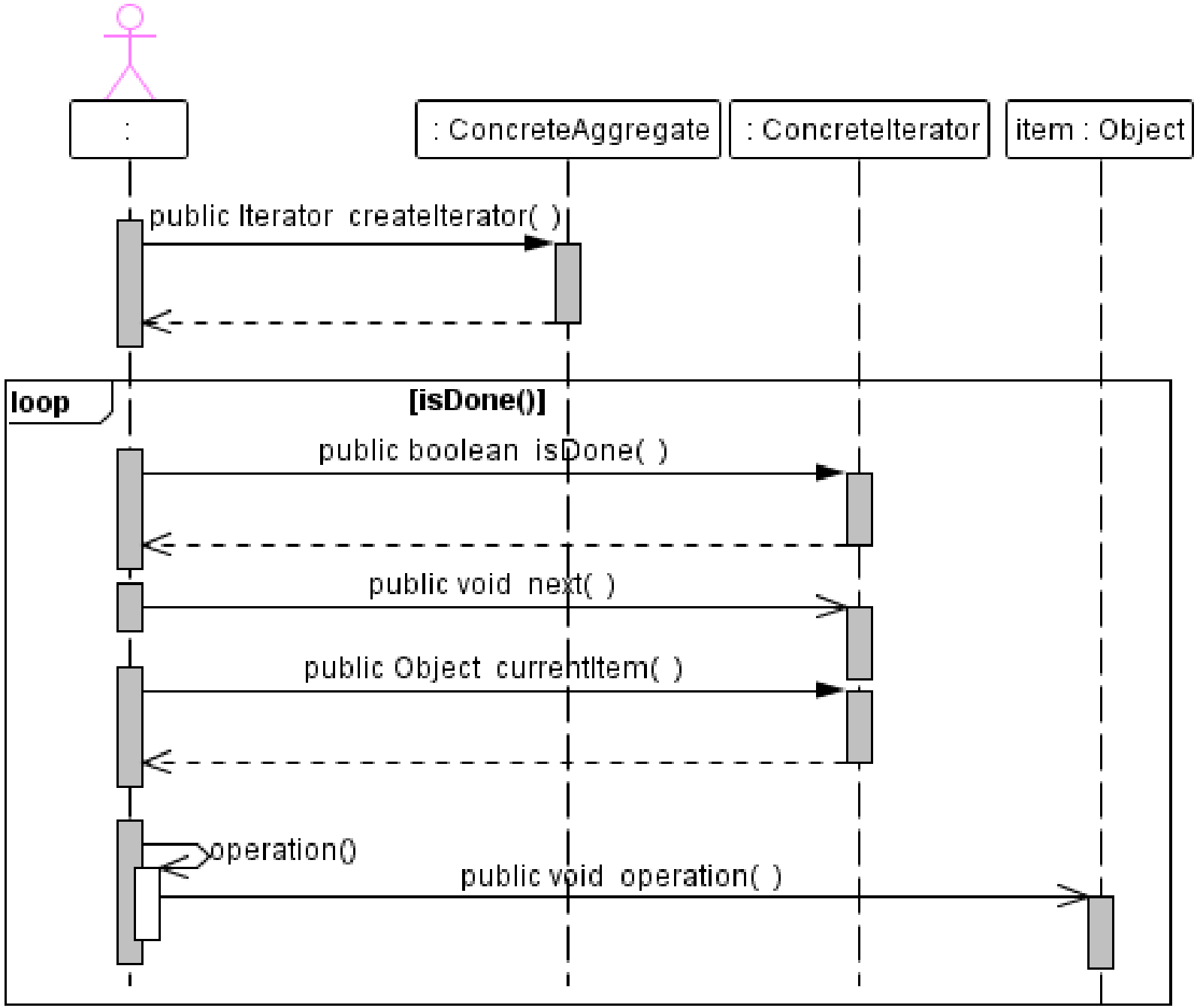
- Łatwa modyfikacja gramatyki
- Łatwa implementacja gramatyki
- Trudna obsługa złożonej gramatyki – każda klasa reprezentuje co najmniej jedną regułą produkcji
- Dodawanie nowych sposobów interpretowania wyrażeń przez modyfikacje klas

•**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**

2) Iterator - *Iterator*




:
: ConcreteAggregate
: ConcreteIterator
item : Object
public Iterator createIterator( )
loop
[isDone()]
public boolean isDone( )
public void next( )
public Object currentItem( )
operation()
public void operation( )

- **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, przez który przechodzi **Iterator**. Istnieje powiązanie jedynie między obiektami typu **ConcreteAggregate** i **ConcreteIterator.**

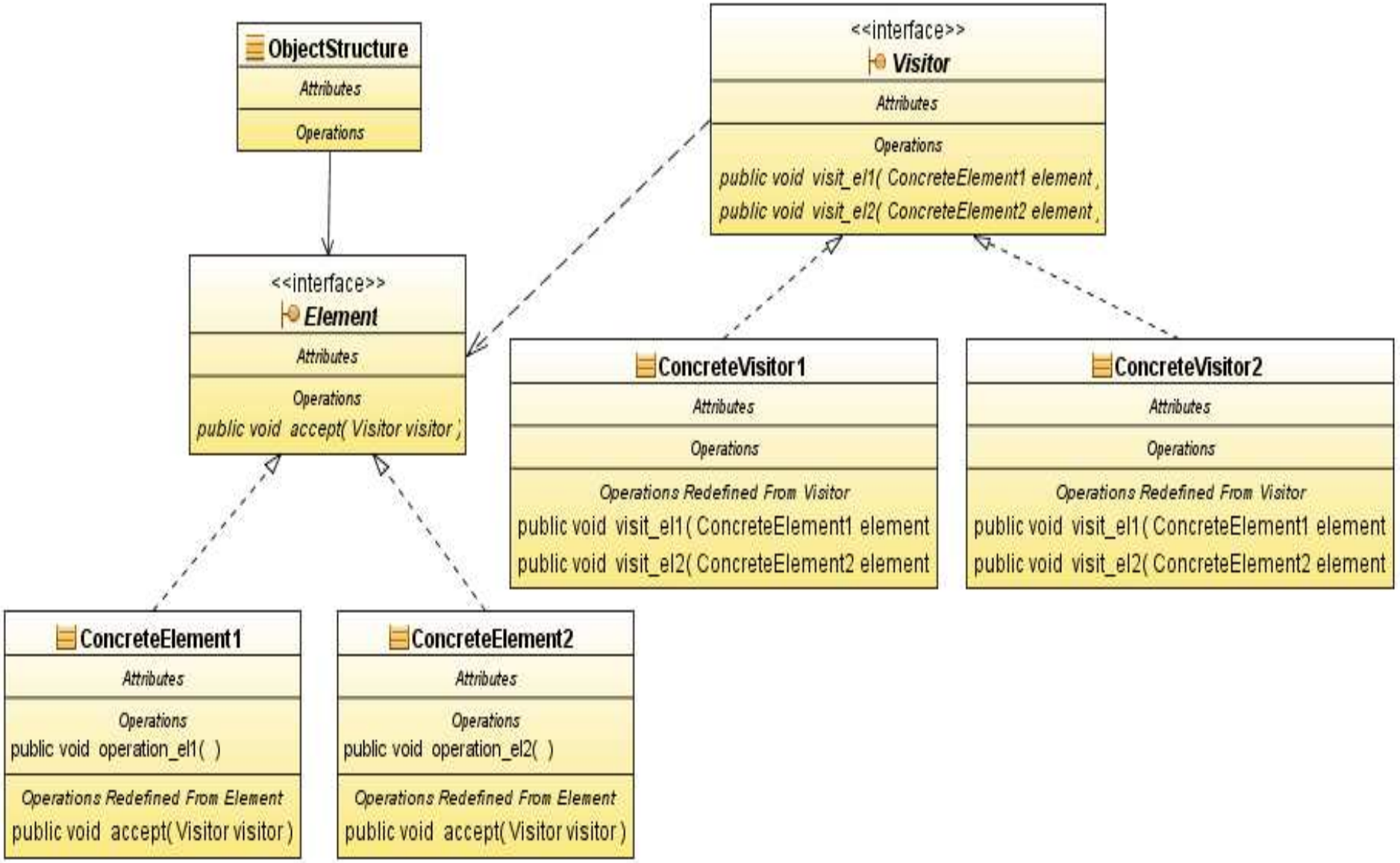
- **Klient wzorca**: klient wzorca może śledzić, który obiekt w agregacie jest bieżący i potrafi wskazać następny lub poprzedni obiekt w tym agregacie

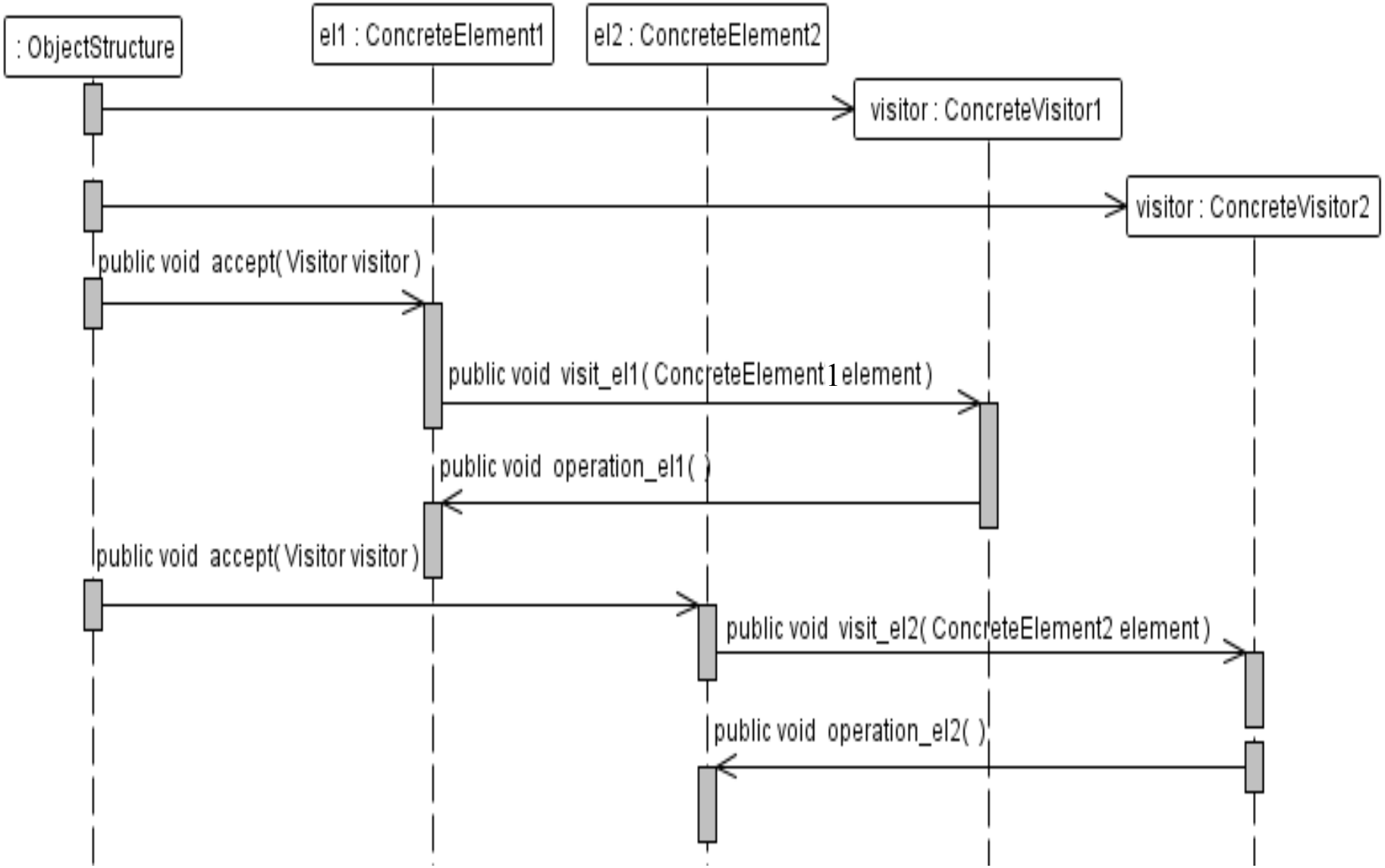
- **Rezultat**:
  - Możliwość dowolnego przejścia przez agregat
  - Interfejs iteratora upraszcza interfejs agregatu
  - W danej chwili może odbywać się wiele przejść przez agregat za pomocą iteratora

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

- **Pokrewne wzorce**
  - Kompozyt (**Composite**)
  - Metody wytwórcze (**Factory Method**)
  - Pamiątka (**Memento**) do przechowania stanu Iteratora

3) Odwiedzający - *Visitor*




: ObjectStructure
el1 : ConcreteElement1
el2 : ConcreteElement2
visitor : ConcreteVisitor1
visitor : ConcreteVisitor2
public void accept(Visitor visitor)
public void visit_el1(ConcreteElement1 element)
public void operation_el1( )
public void accept(Visitor visitor)
public void visit_el2(ConcreteElement2 element)
public void operation_el2( )

- **Problem:** Należy pozwolić na zdefiniowanie nowej operacji bez zmiany definicji klasy obiektów, w których ona działa.

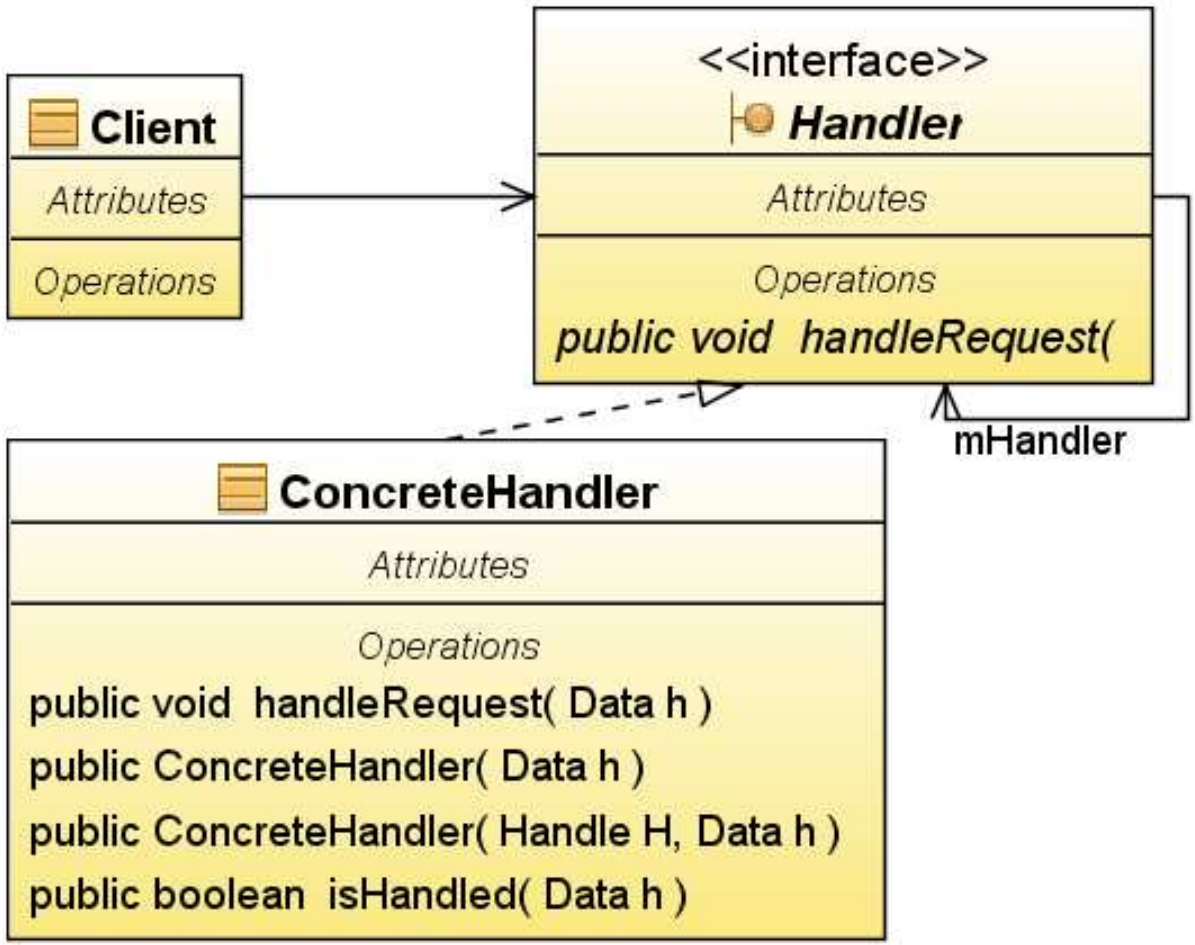
- **Rozwiązanie:** obiekt typu **ObjectStructure** może wprowadzić swoje obiekty i umożliwia obiektom, które implementują interfejs **Element** (i mogą być złożone) odwiedzanie przez obiekty implementujące interfejs **Visitor**. Interfejs **Visitor** deklaruje metody wizytujące (np. visit_el1), które 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**.

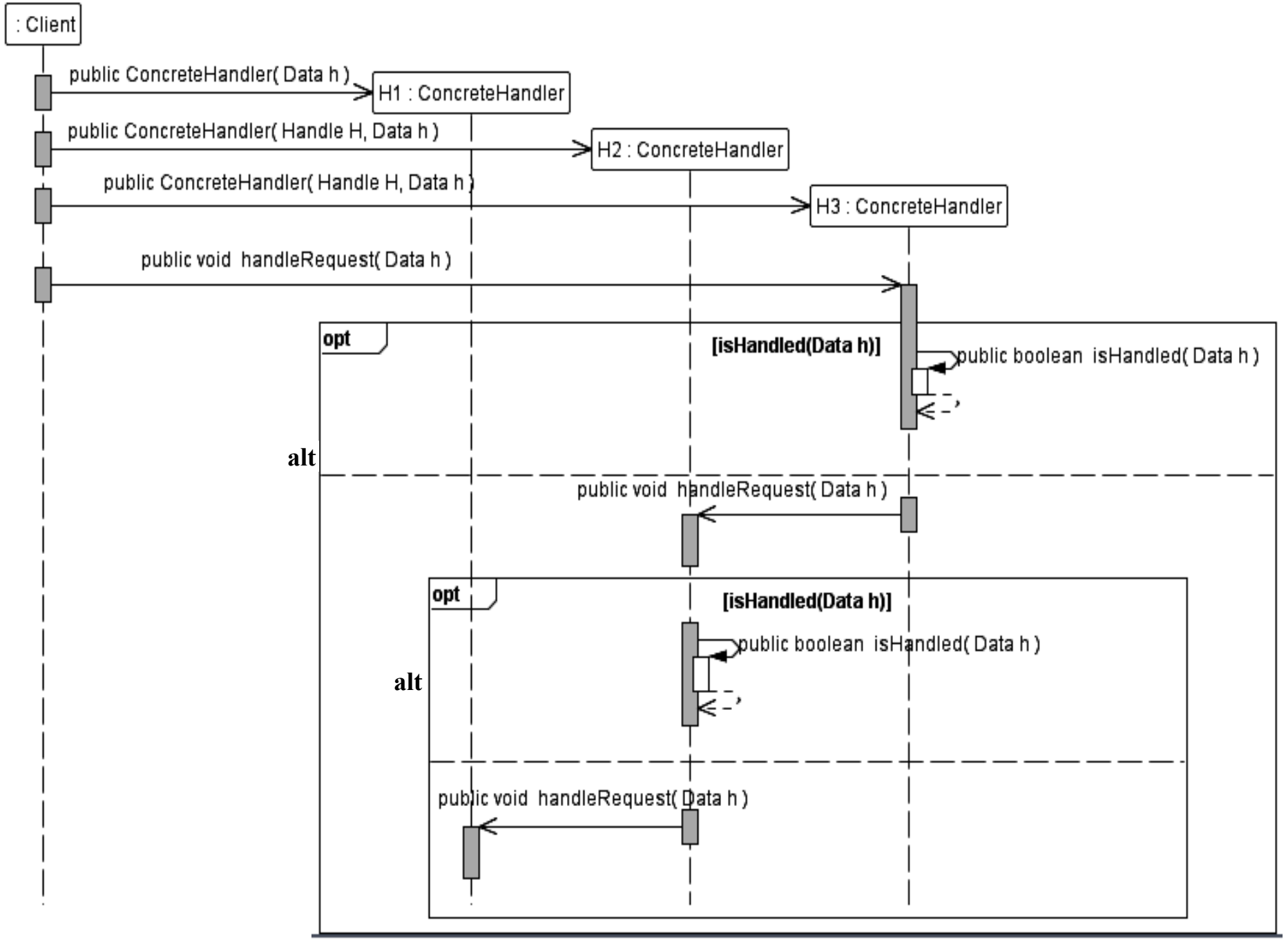
- **Klient:** Klient reprezentowana przez obiekt typu **ObjectStructure** musi utworzyć obiekty typu **ConcreteVisitor** i przejść przez całą strukturę obiektów typu **Element**, odwiedzając każdy element, za pomocą obiektu typu **ConcreteVisitor**. Każdy obiekt typu **Element** wywołuje metodę wizytującą obiektu **ConcreteVisitor**, dając dostęp do siebie, i pozwala jej wywołać odpowiednią metodę swojej klasy, umożliwiajacą zbadanie stanu.

- **Wynik:**
  - 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 odwiedziania obiektów wzorca Kompozyt (**Composite**)
  - Odwiedzający może służyć do interpretowania we wzorcu **Interpreter**

4) Łańcuch zobowiązań - *Chain of Responsibility*




: Client
public ConcreteHandler(Data h)
H1 : ConcreteHandler
public ConcreteHandler(Handle H, Data h)
H2 : ConcreteHandler
public ConcreteHandler(Handle H, Data h)
H3 : ConcreteHandler
public void handleRequest(Data h)
opt
[isHandled(Data h)]
public boolean isHandled(Data h)
alt
public void handleRequest(Data h)
opt
[isHandled(Data h)]
public boolean isHandled(Data h)
alt
public void handleRequest(Data h)

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.

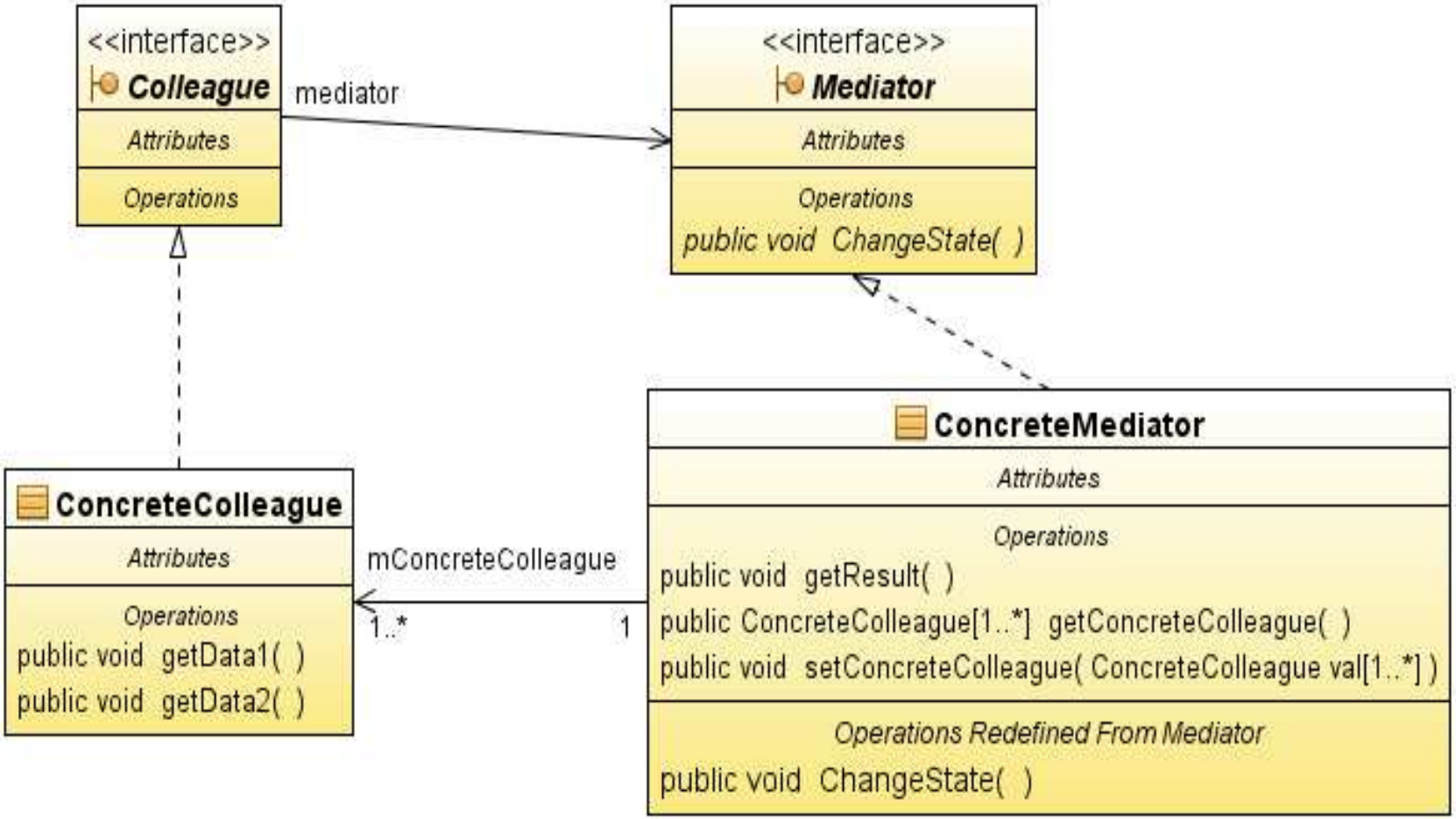
- **Klient**: generuje i kieruje żądania do listy obiektów **ConcreteHandler**

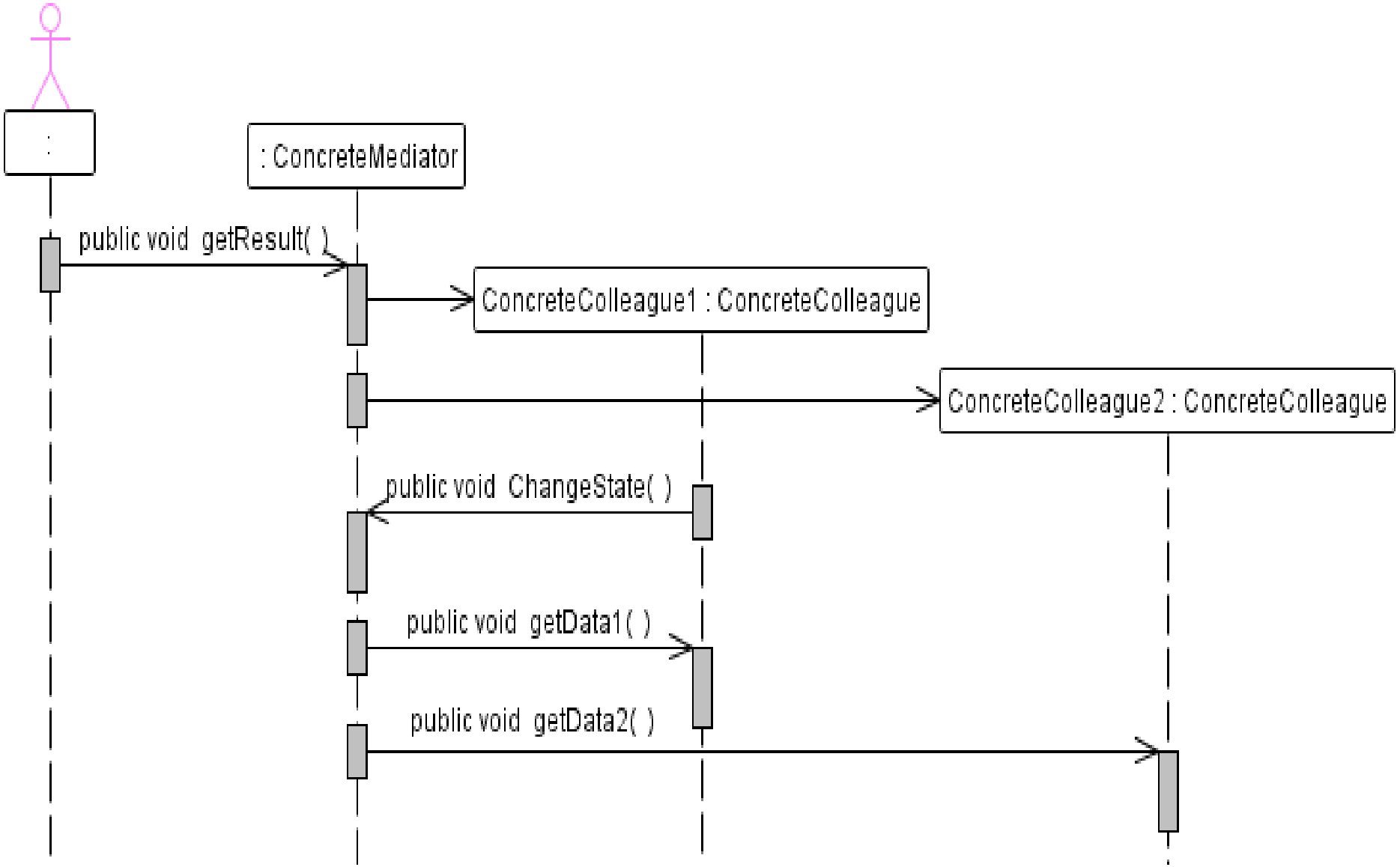
- **Wynik**: Obiekt do obsługi żadań typu **ConcreteHandler** nie ma wyraźnej wiedzy o innych obiektach z łańcucha i nie muszą znać struktury łańcucha. Łańcuch zobowiązań zwiększa elastyczność w przyznawaniu zgłoszeń serwisowych poprzez zmianę podklasy obiektów i struktur łańcucha obiektów - ale bez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

- **Realizacja**: Służy do obsługi zdarzeń

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

5) Mediator - *Mediator*




:
: ConcreteMediator
public void getResult( )
ConcreteColleague1 : ConcreteColleague
ConcreteColleague2 : ConcreteColleague
public void ChangeState( )
public void getData1( )
public void getData2( )

- **Problem:** Należy ograniczyć znajomość złożonych powiązań pomiędzy obiektami, które oddziałują w złożony sposób i pozwolić na zmienianę 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.**

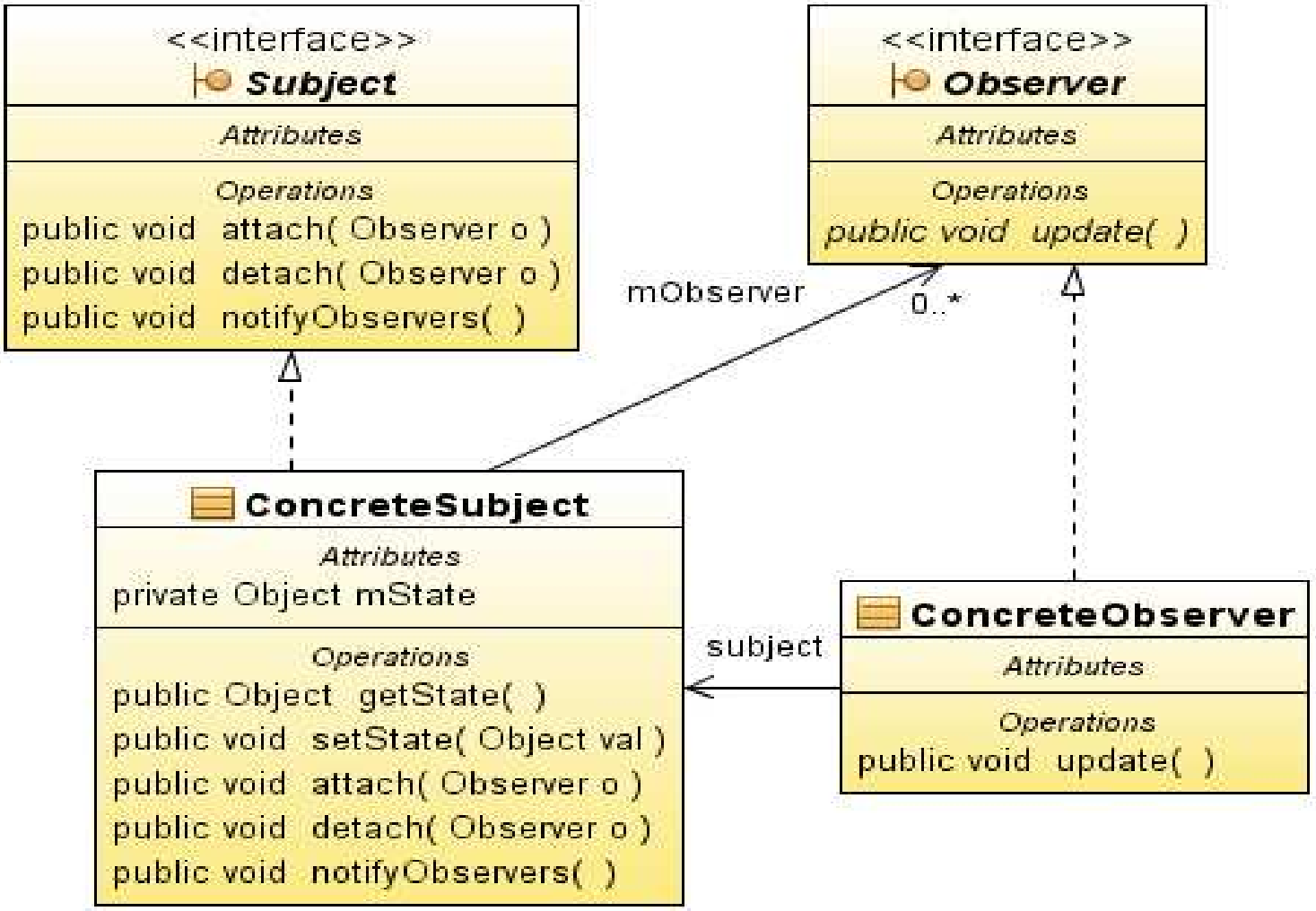
  - Obiekty typu **ConcreteColleague** wysyłają żądania do obiektu typu **ConcreteMediator**, a obiekt typu **ConcreteMediator** podejmuje decyzję i wysyła te wnioski do odpowiednich obiektów typu **ConcreteColleague**.

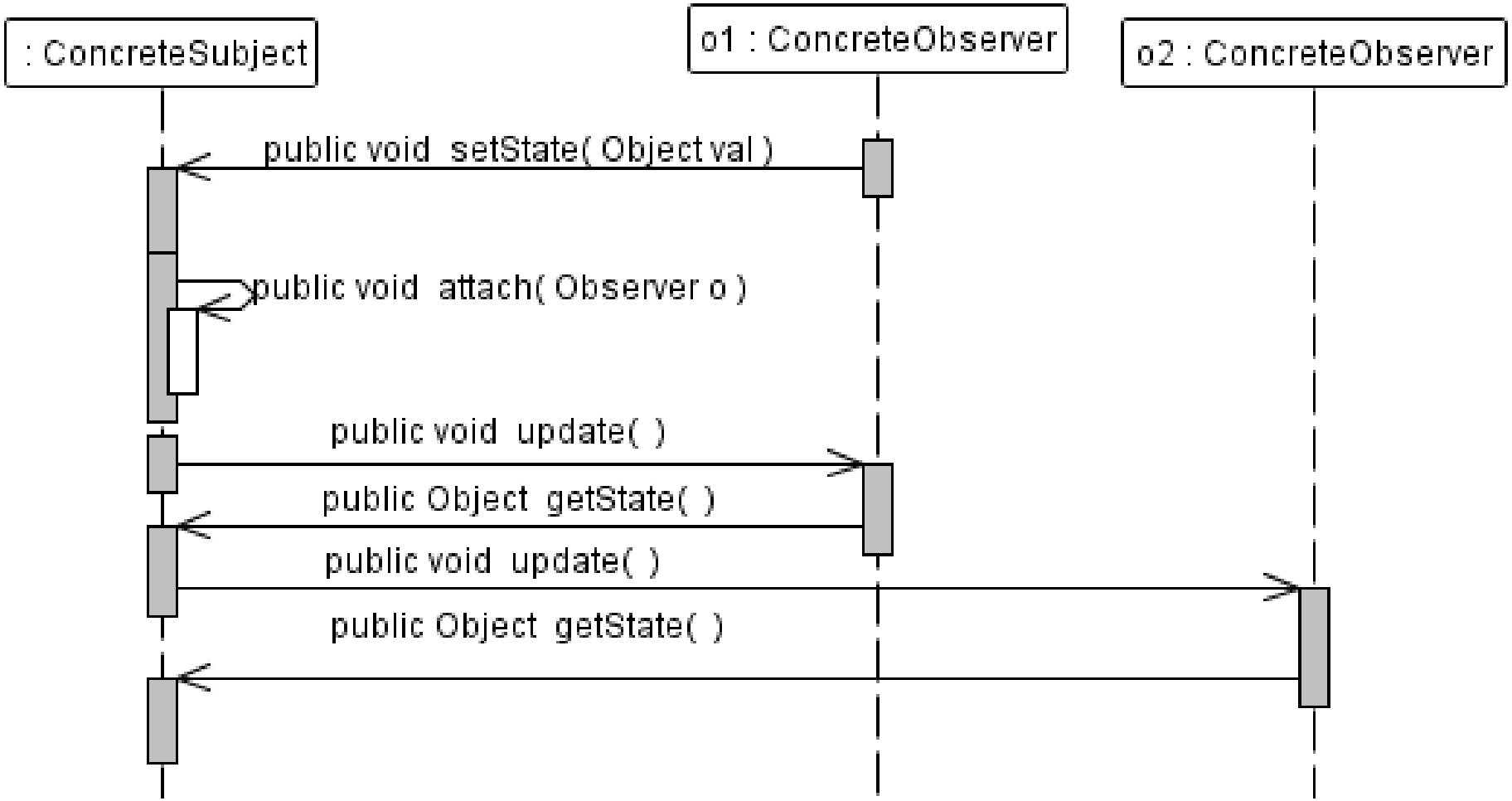
- **Wynik:**

  - Obiekt **ConcreteMediator** skupia zachowania, które byłyby umieszczone w wielu obiektach typu **ConcreteColleague**
  - Można łączyć różne rodzaje obiektów typów **ConcreteColleague** i **ConcreteMediator**
  - Uproszczenie protokołów komunikacji za pomocą związków jeden-do-wielu między obiektami typów **ConcreteMediator** i **ConcreteColleague**, zastępując wiele obiektów typu **ConcreteMediator** 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) Obserwator - *Observer*



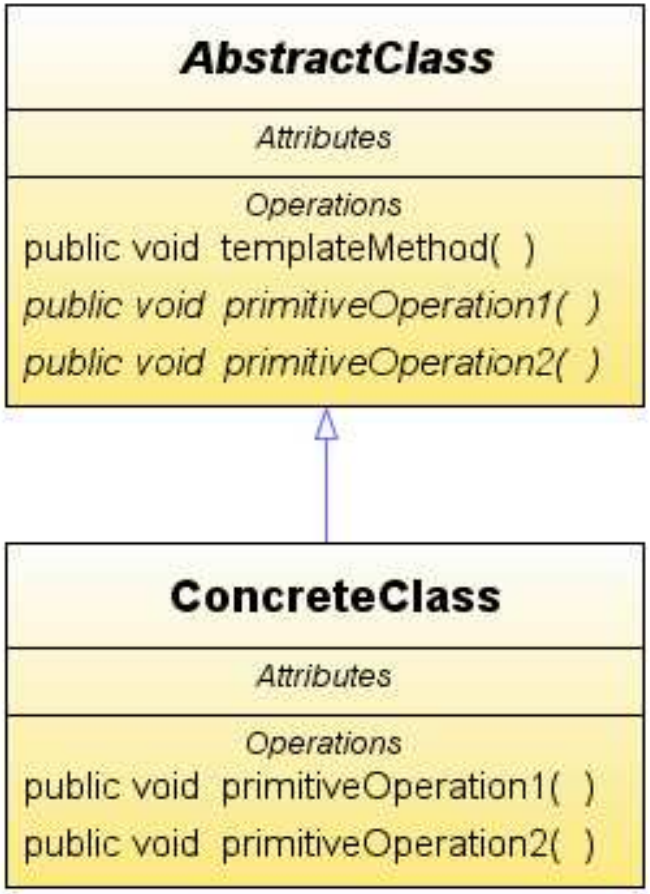

: ConcreteSubject
o1 : ConcreteObserver
o2 : ConcreteObserver
public void setState(Object val)
public void attach(Observer o)
public void update( )
public Object getState( )
public void update( )
public Object getState( )

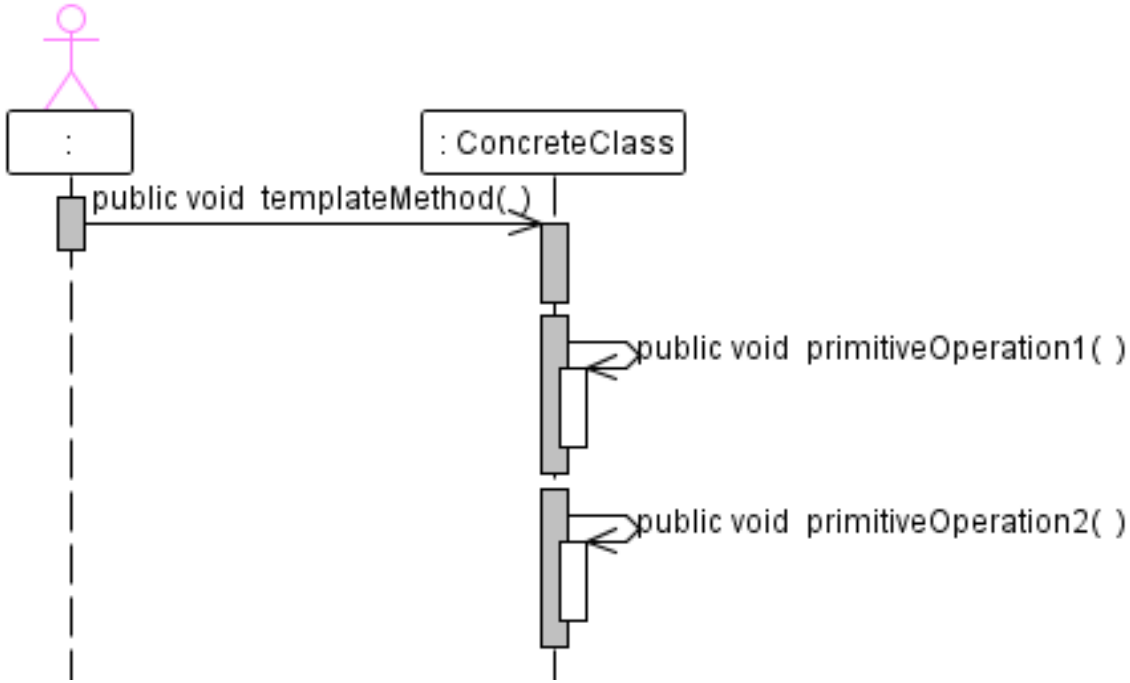
- **Problem:** Konieczność okreslenia zależności typu jeden-do-wielu między obiektami - gdy obiektu zmienia stan, wszystkie obiekty zależne są automatycznie powiadamiane i aktualizowane

- **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**
- **Wynik:**
  - Abstrakcyjny związek pomiędzy obiektami typu **ConcreteSubject** i obiektów typu **ConcreteObserver**
  - Wsparcie dla wysyłania wiadomości - obiekty typu **ConcreteSubject** przesyłają zgłoszenie, nie znając odbiorcy
  - Nieoczekiwany 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

7) Metoda szablonowa – *Template Method*





- **Problem:** Należy określić szkielet algorytmu, a szczegóły algorytmu powierzyć klasom pochodnym

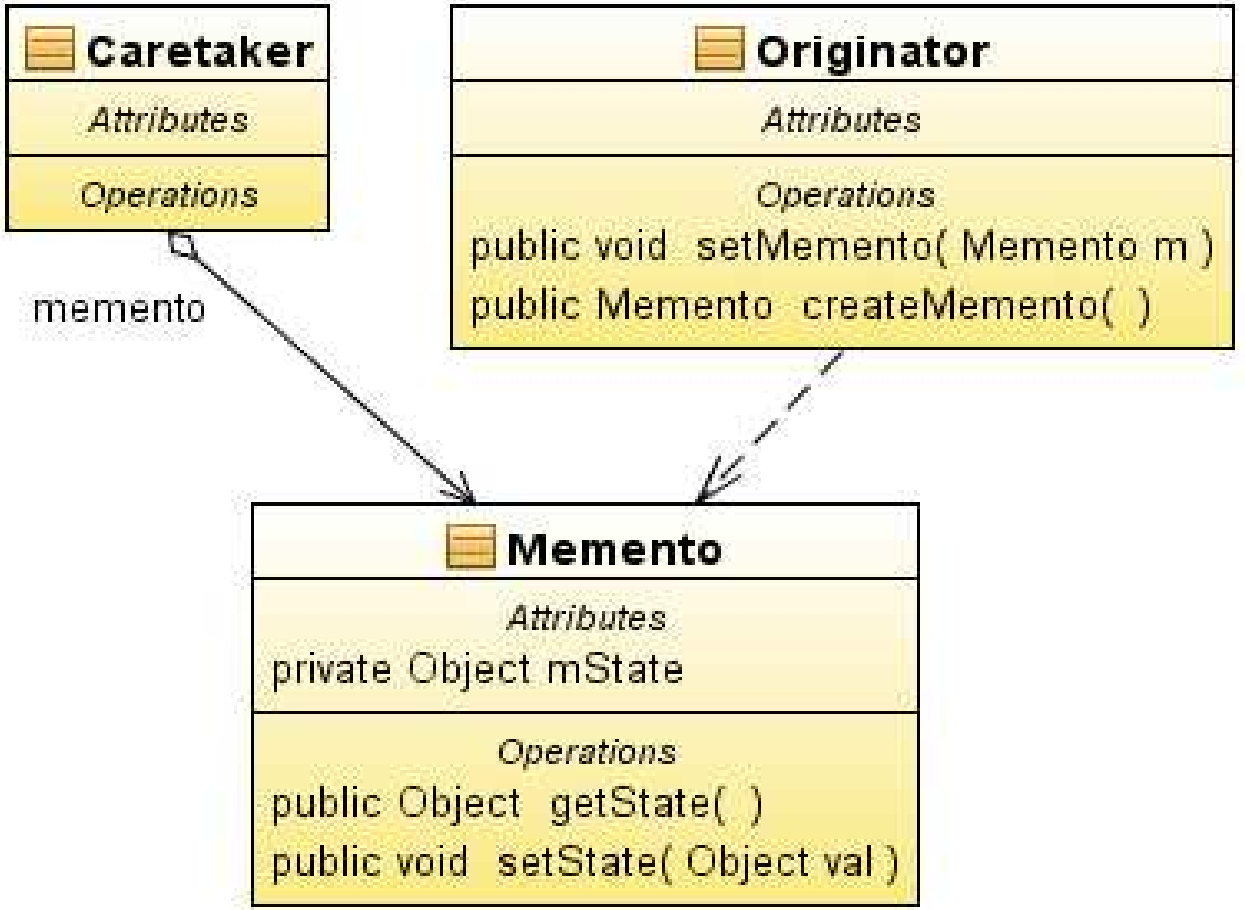
- **Rozwiązanie:** Klasa **AbstractClass** definiuje abstrakcyjny algorytm, ale pewne części abstrakcyjnego algorytmu są uzupełniane przez różne definicje, realizowane przez metody klasy **ConcreteClass**

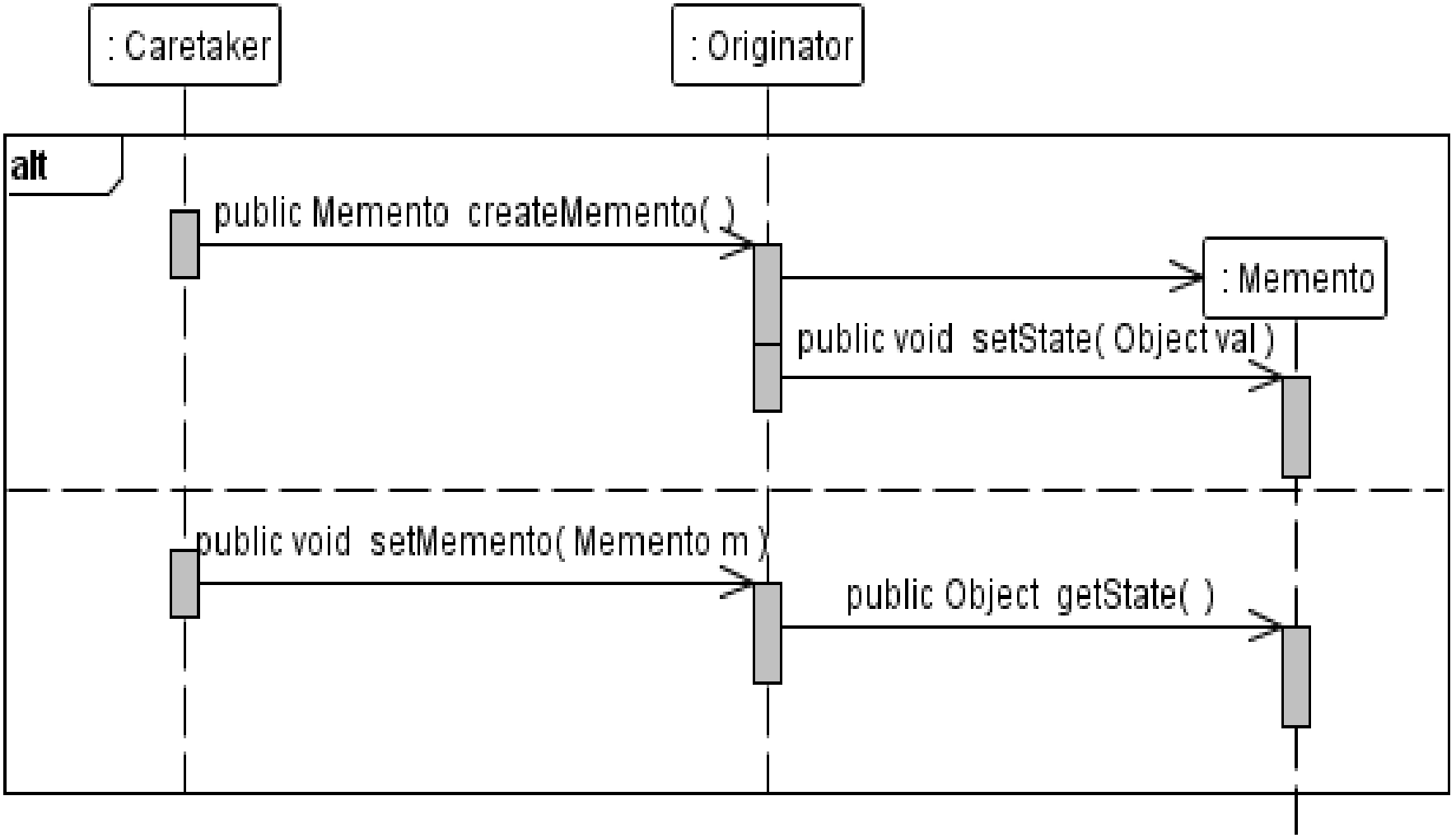
- **Wynik:** "zasada Hollywood " (nie zadzwoń do nas, my zadzwonimy do Ciebie), gdzie metoda klasy bazowej wywołuje metody z klas pochodnych

- **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**)

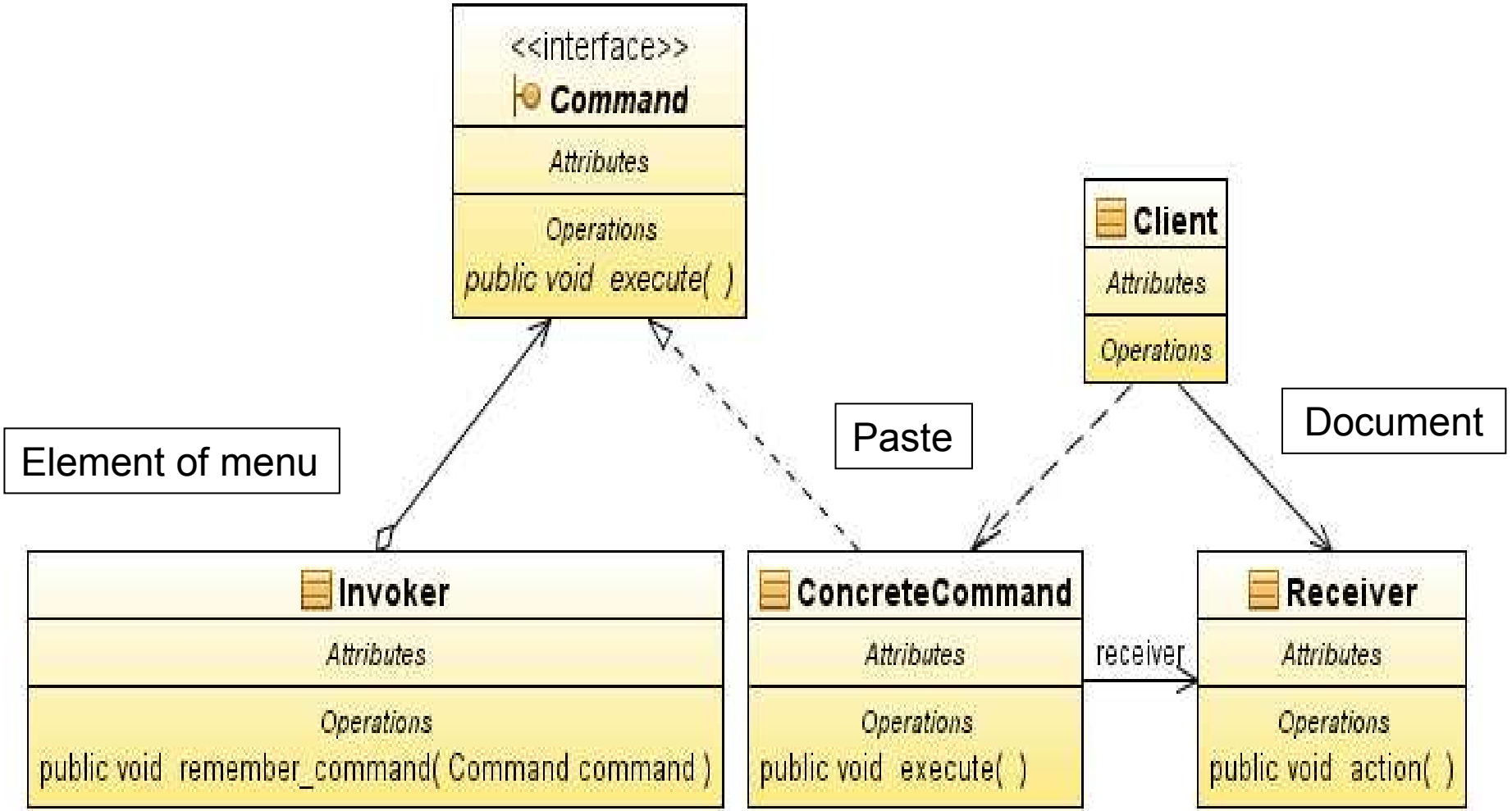
8) Pamiątka - *Memento*

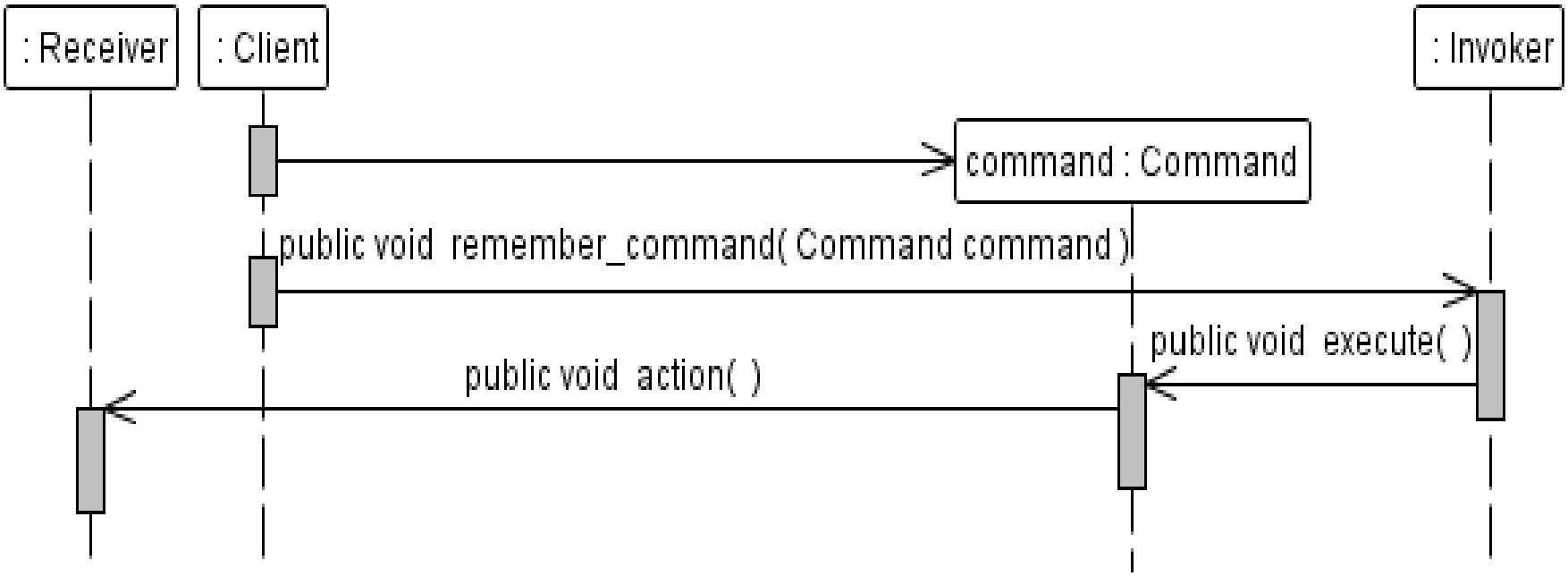



: Caretaker
: Originator
alt
public Memento createMemento( )
: Memento
public void setState(Object val)
public void setMemento(Memento m)
public Object getState( )

- **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 **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**.
- **Wynik:**
  - 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

9) Polecenie - *Command*

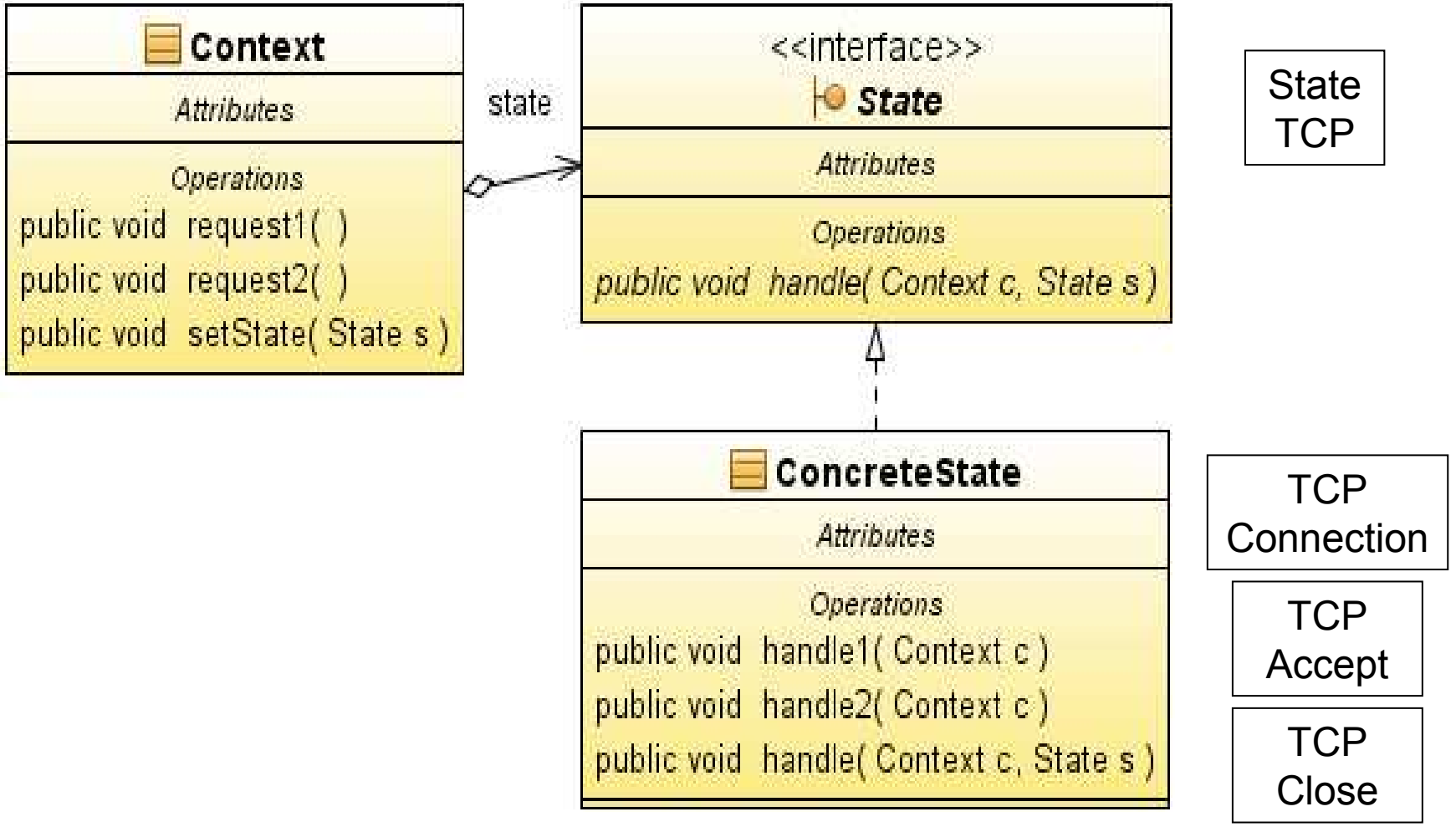


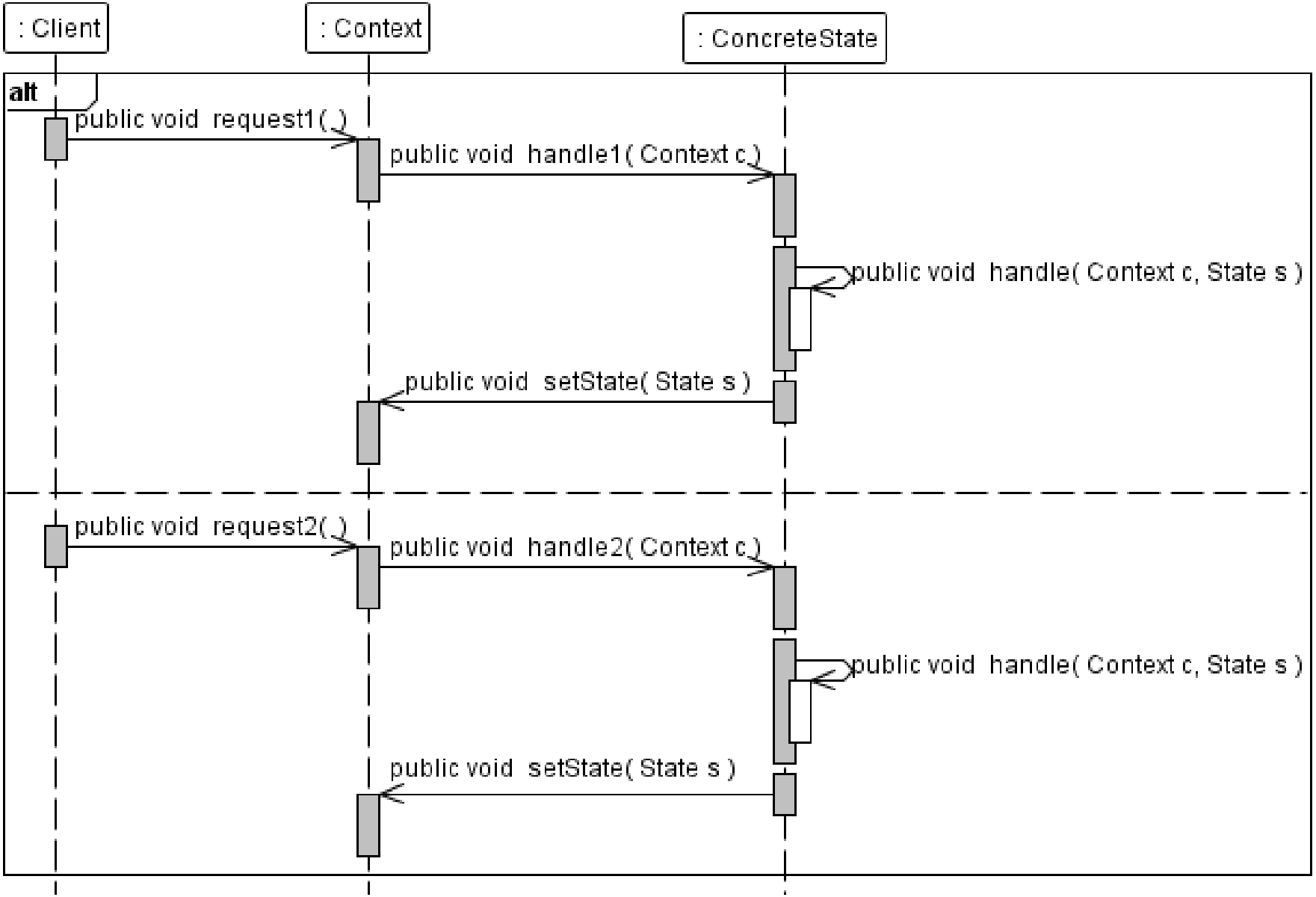

: Receiver
: Client
: Invoker
command : Command
public void remember_command(Command command)
public void execute( )
public void action( )

- **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:** Klient tworzy obiekt typu **ConcreteCommand** i ustala jego obiekt typu **Receiver**, które wykonują akcje oraz ustala obiekt zainteresowany akcją, taki jak obiekt **Invoker**.
- **Wynik:**
  - 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

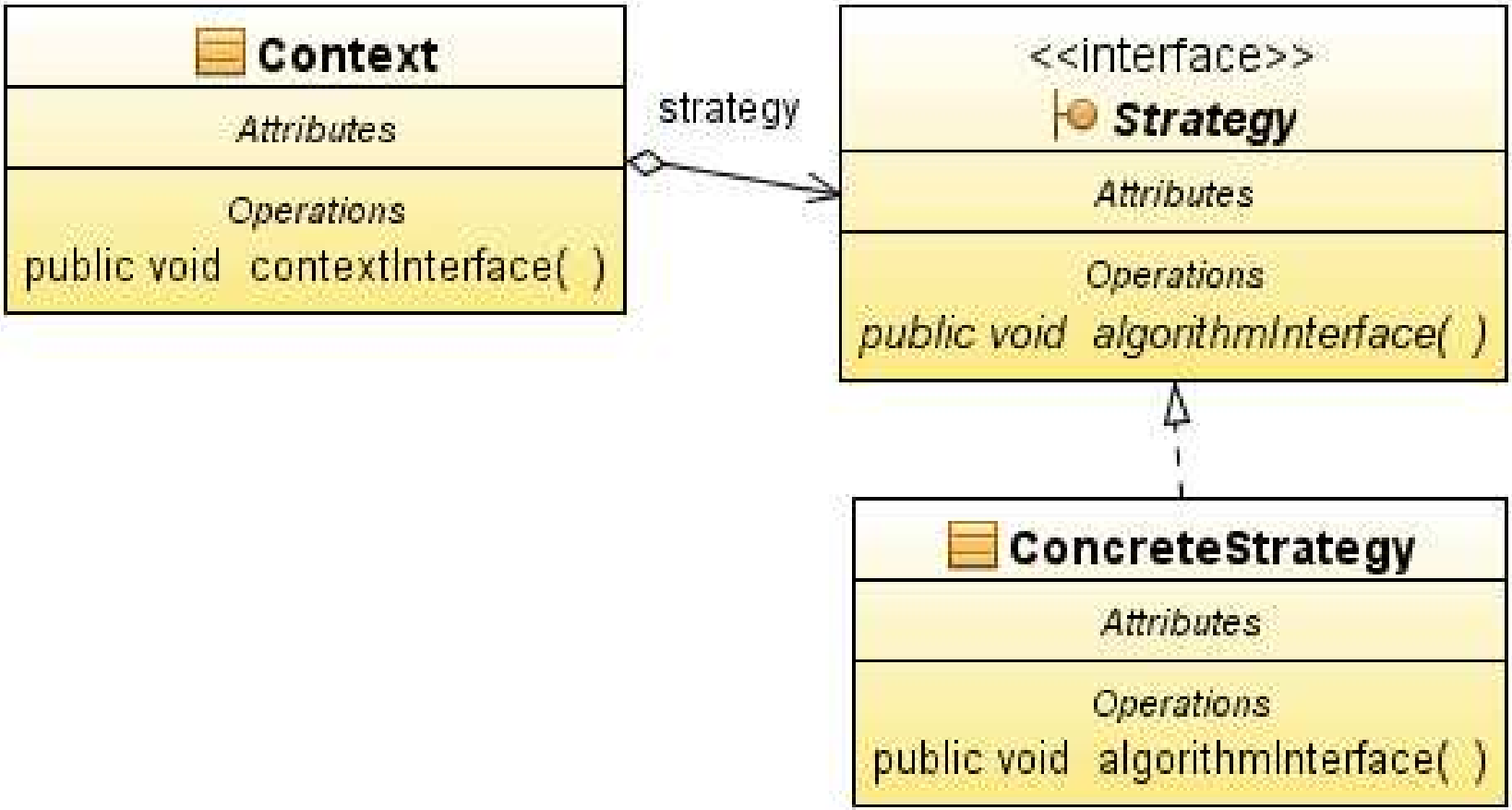
10) Stan - *State*

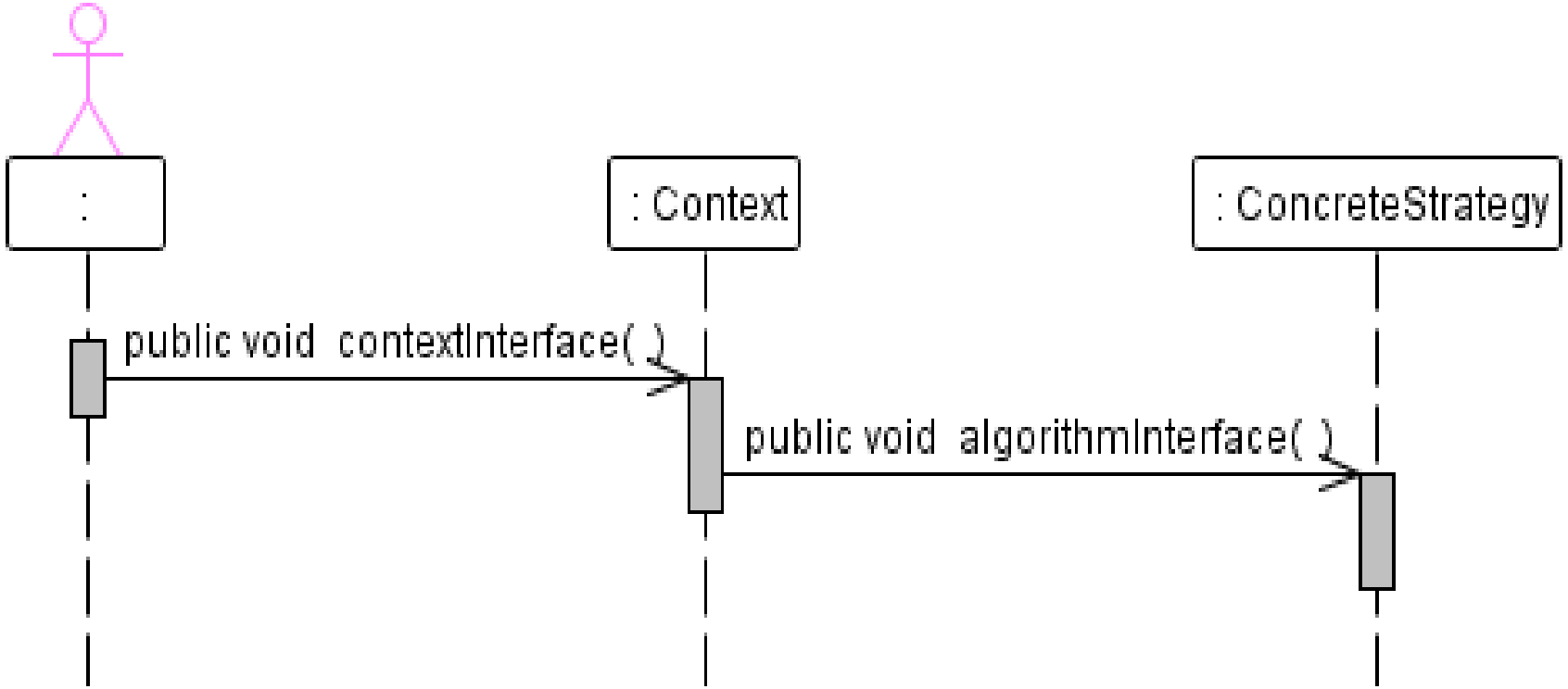



: Client
: Context
: ConcreteState
alt
public void request1( )
public void handle1(Context c)
public void handle(Context c, State s)
public void setState(State s)
public void request2( )
public void handle2(Context c)
public void handle(Context c, State s)
public void setState(State s)

- **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:** obiekt, który konfiguruje obiekty **ConcreteState** za pomocą obiektu **Context**
- **Wynik:**
  - Lokalizacja informacji zwiazanej z każdym ze stanów obiektów tyou **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 **ConcreteState**, ponieważ ich stany są reprezentowane przez ich rodzaje (podobnie jak wzorzec **Flyweight**).
- **Pokrewne wzorce:**
  - Wzorzec Pyłek (**Flyweight**) wyjaśnia jak i kiedy należy współdzielić obiekty Stan (**State**)
  - Obiekty Stan są często wzorcem **Singleton**

11) Strategia - *Strategy*




:
: Context
: ConcreteStrategy
public void contextInterface( )
public void algorithmInterface( )

- **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**)