Zaawansowane programowanie obiektowe - wzorce operacyjne

Opracowanie: dr Piotr Jastrzębski

Wzorce operacyjne

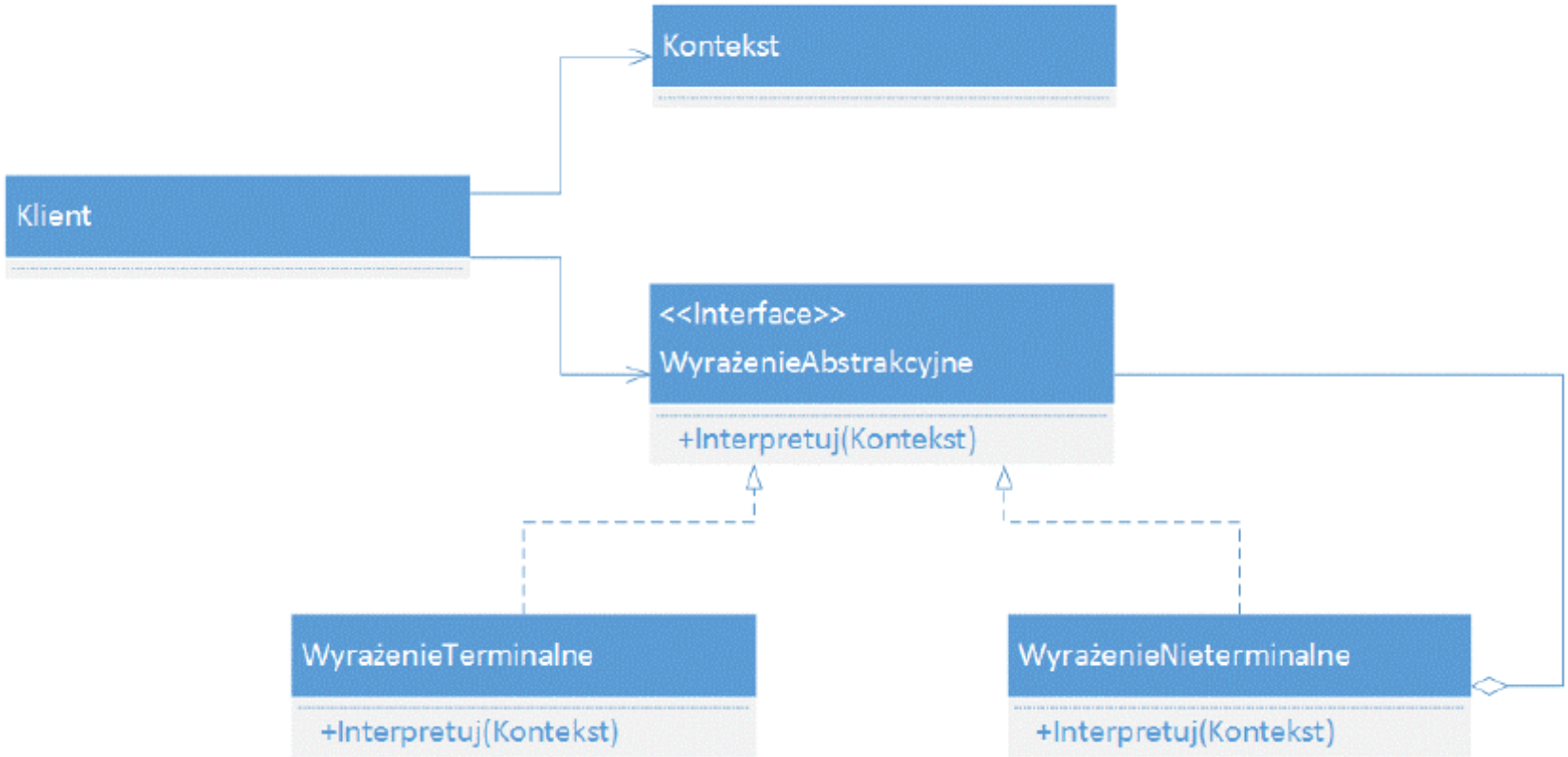
Wzorce operacyjne

Wzorce operacyjne (czynnościowe) opisują zachowanie obiektów, komunikację pomiędzy nimi i ich odpowiedzialność.

- Interpreter
- Iterator (kursor)
- Łańcuch zobowiązań
- Mediator
- Metoda szablonowa
- Obserwator
- Odwiedzający (wizytator)
- Pamiątka (znacznik)
- Polecenie
- Stan
- Strategia (polityka)

Interpreter

Interpreter – czynnościowy wzorzec projektowy, którego celem jest zdefiniowanie opisu gramatyki pewnego języka interpretowalnego, a także stworzenie dla niej interpretera, dzięki któremu będzie możliwe rozwiązanie opisanego problemu.

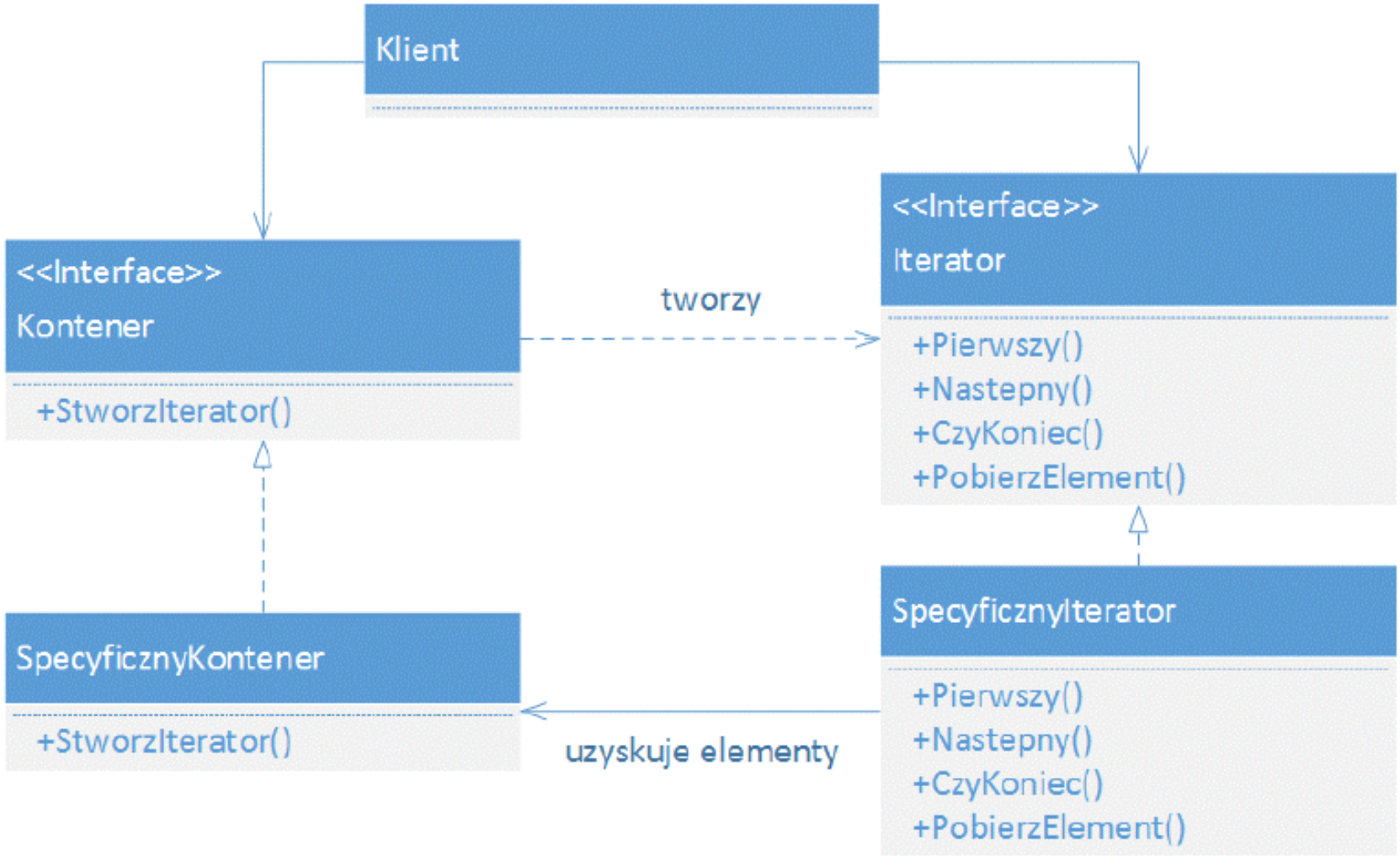

Kontekst
Klient
<<Interface>>
WyrażenieAbstrakcyjne
+Interpretuj(Kontekst)
WyrażenieTerminalne
+Interpretuj(Kontekst)
WyrażenieNieterminalne
+Interpretuj(Kontekst)

Zastosowania:

- interpretacja języka
- kompilatory
- odwrotna notacja polska,. . .

Iterator (kursor)

Iterator – czynnościowy wzorzec projektowy (obiektowy), którego celem jest zapewnienie sekwencyjnego dostępu do podobiektów zgrupowanych w większym obiekcie.


Klient
<<Interface>>
Kontener
+StworzIterator()
tworzy
<<Interface>>
Iterator
+Pierwszy()
+Nastepny()
+CzyKoniec()
+PobierzElement()
SpecyficznyKontener
+StworzIterator()
uzyskuje elementy
SpecyficznyIterator
+Pierwszy()
+Nastepny()
+CzyKoniec()
+PobierzElement()

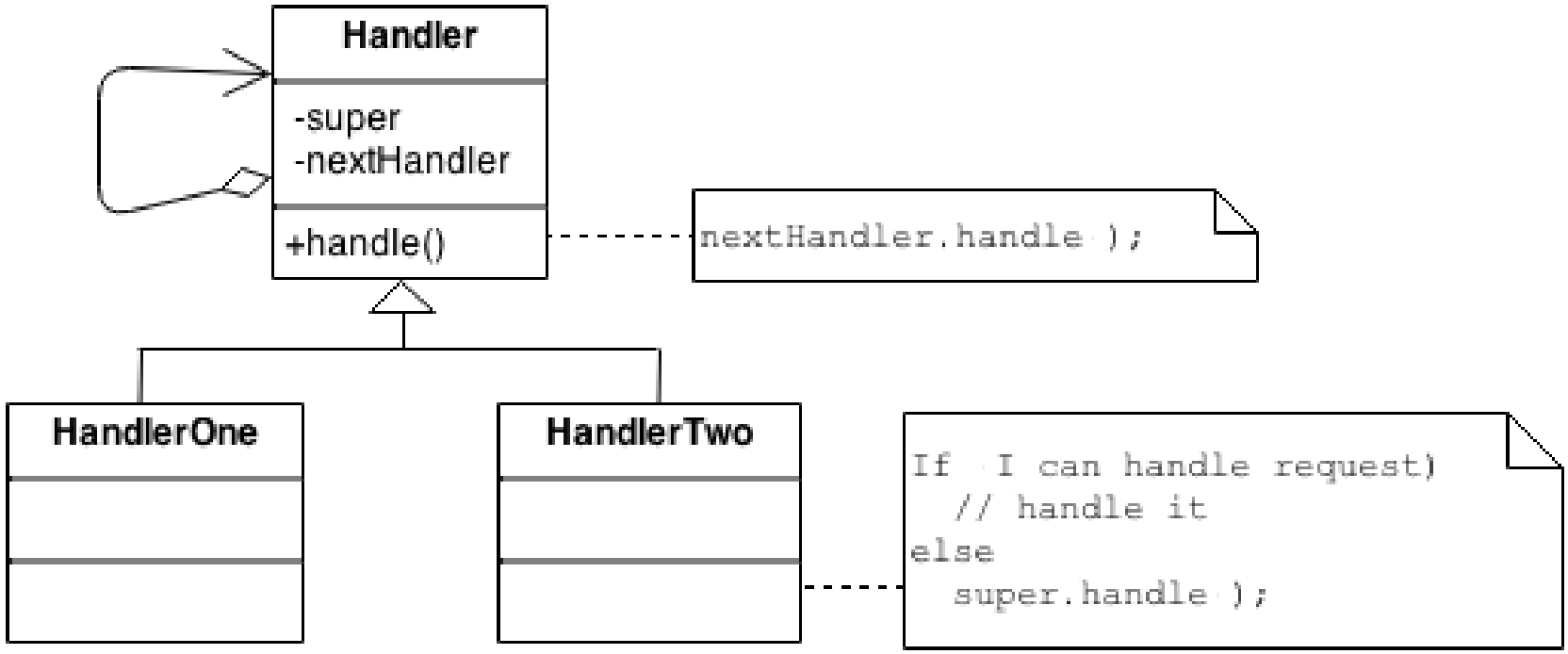
Zastosowania:

- przetwarzanie zróżnicowanych kolekcji,
- aplikacje, w których dane są przechowywane w kolekcjach różnych typów.

Łańcuch zobowiązań

Łańcuch zobowiązań - czynnościowy wzorzec projektowy, w którym żądanie może być przetwarzane przez różne obiekty, w zależności od jego typu.

```
public void operacja(żądanie: Żądanie)
{
   jeśli potrafimy obsłużyć dany typ żądania żądanie:
      obsłuż żądanie
   w przeciwnym wypadku:
      przekaż żądanie następnikowi
}
```


Handler
-super
-nextHandler
+handle()
nextHandler.handle);
HandlerOne
HandlerTwo
If I can handle request)
// handle it
else
super.handle);

Zastosowanie:

- Wzorzec znajduje zastosowanie wszędzie tam, gdzie mamy do czynienia z różnymi mechanizmami podobnych żądań, które można zaklasyfikować do różnych kategorii. Dodatkową motywacją do jego użycia są często zmieniające się wymagania.

Zalety:

- elementy łańcucha mogą być dynamicznie dodawane i usuwane w trakcie działania programu,
- zmniejszenie liczby zależności między nadawcą a odbiorcami,
- implementacja pojedynczej procedury nie musi znać struktury łańcucha oraz innych procedur.

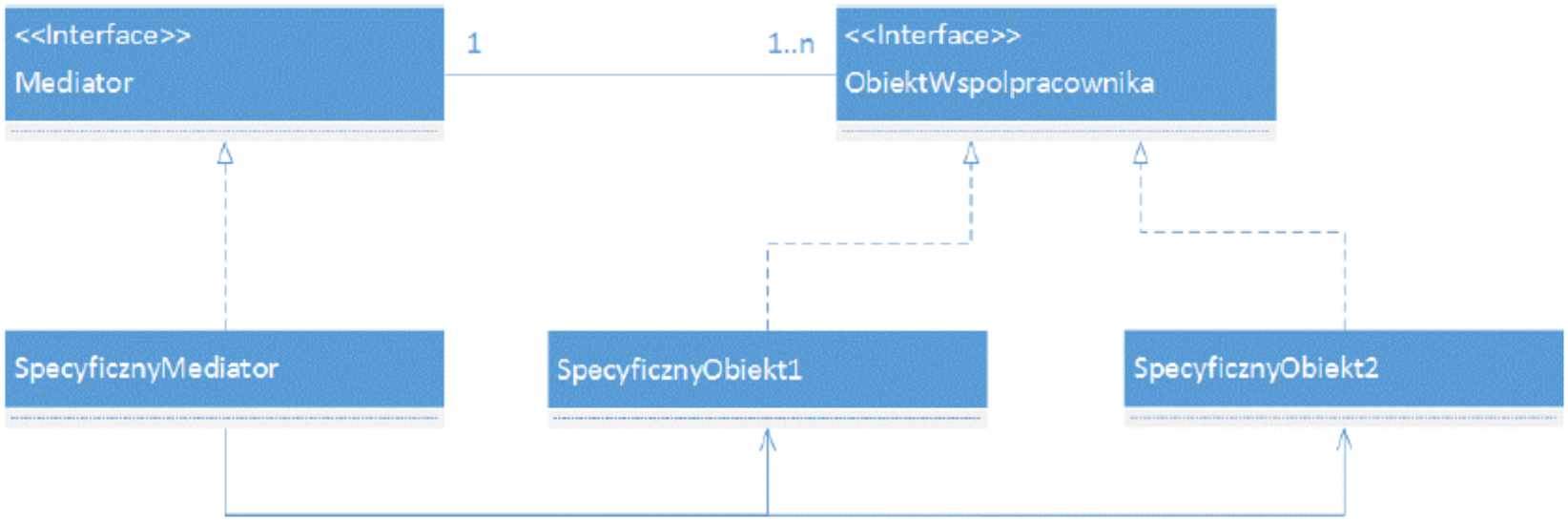
Wady:

- wzorzec nie gwarantuje, że każde żądanie zostanie obsłużone,
- śledzenie i debugowanie pracy działania łańcucha może być trudne.

Mediator

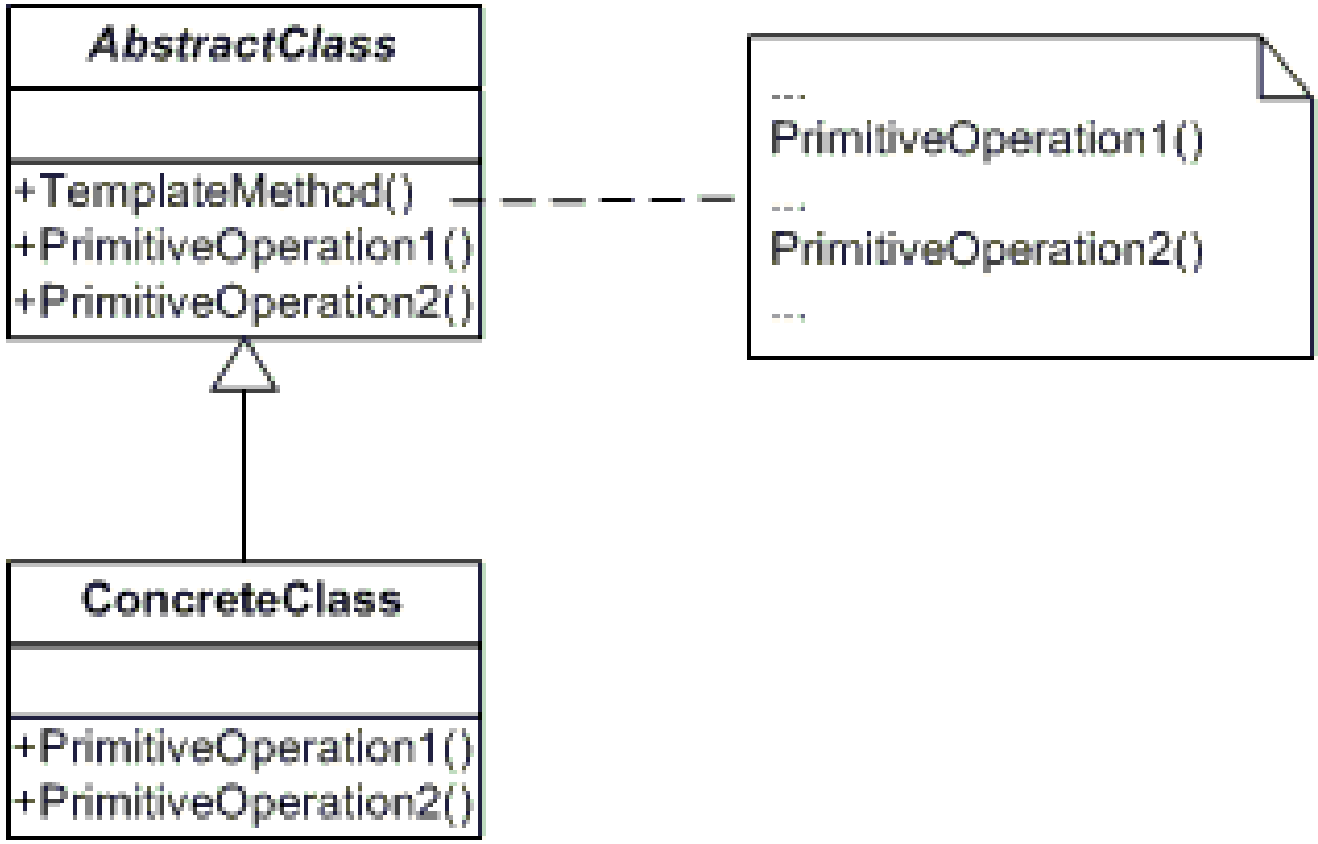
Mediator – wzorzec projektowy należący do grupy wzorców czynnościowych. Mediator zapewnia jednolity interfejs do różnych elementów danego podsystemu.

Wzorzec mediatora umożliwia zmniejszenie liczby powiązań między różnymi klasami, poprzez utworzenie mediatora będącego jedyną klasą, która dokładnie zna metody wszystkich innych klas, którymi zarządza. Nie muszą one nic o sobie wiedzieć, jedynie przekazują polecenia mediatorowi, a ten rozsyła je do odpowiednich obiektów.


<<Interface>>
Mediator
1
1..n
<<Interface>>
ObiektWspolpracownika
SpecyficznyMediator
SpecyficznyObiekt1
SpecyficznyObiekt2

Metoda szablonowa

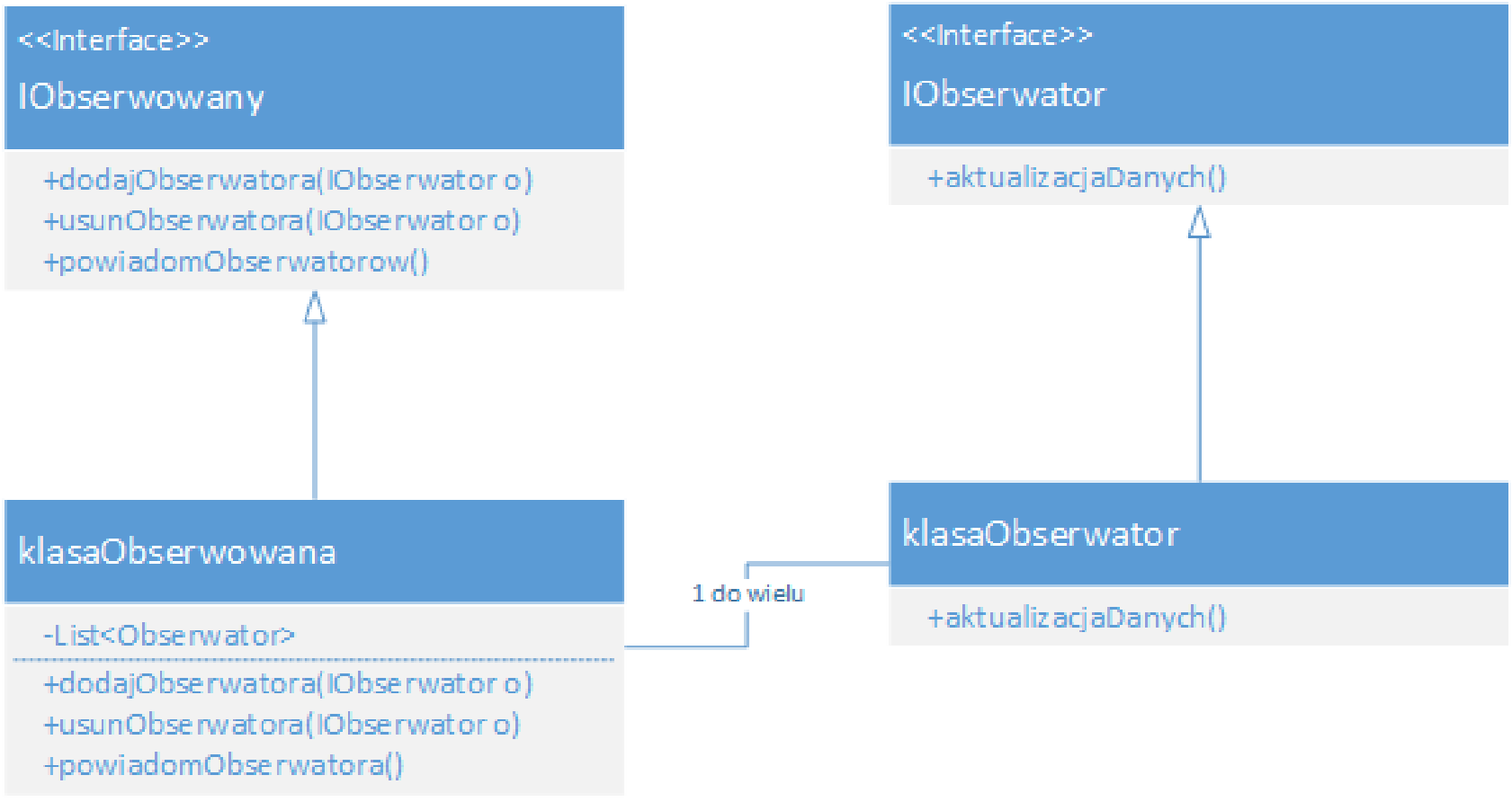
Metoda szablonowa – czynnościowy wzorzec projektowy. Jego zadaniem jest zdefiniowanie metody, będącej szkieletem algorytmu. Algorytm ten może być następnie dokładnie definiowany w klasach pochodnych. Niezmienna część algorytmu zostaje opisana w metodzie szablonowej, której klient nie może nadpisać. W metodzie szablonowej wywoływane są inne metody, reprezentujące zmienne kroki algorytmu. Mogą one być abstrakcyjne lub definiować domyślne zachowania. Klient, który chce skorzystać z algorytmu, może wykorzystać domyślną implementację bądź może utworzyć klasę pochodną i nadpisać metody opisujące zmienne fragmenty algorytmu.


AbstractClass
+TemplateMethod()
+PrimitiveOperation1()
+PrimitiveOperation2()
...
PrimitiveOperation1()
...
PrimitiveOperation2()
...
ConcreteClass
+PrimitiveOperation1()
+PrimitiveOperation2()

Obserwator

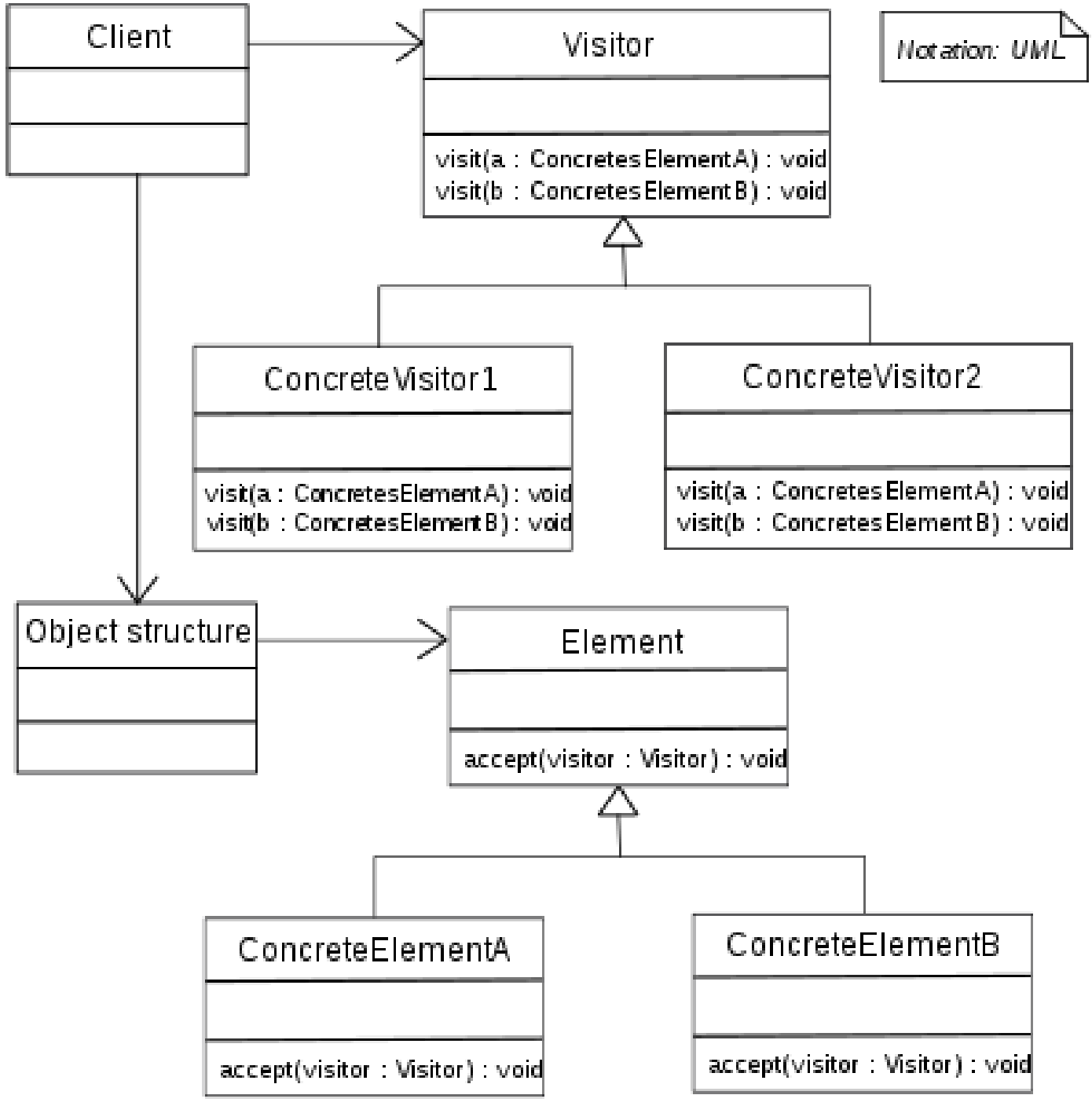
Obserwator – wzorzec projektowy należący do grupy wzorców czynnościowych. Używany jest do powiadamiania zainteresowanych obiektów o zmianie stanu pewnego innego obiektu.

```
public void powiadomObserwatorow() {
   dla każdego obserwatora obserwator z listy obserwatorzy
      wywołaj obserwator.aktualizacja(obserwator);
}
```


<<Interface>>
IObserwowany
+dodajObserwatora(IObserwator o)
+usunObserwatora(IObserwator o)
+powiadomObserwatorow()
<<Interface>>
IObserwator
+aktualizacjaDanych()
klasaObserwowana
-List<Obserwator>
+dodajObserwatora(IObserwator o)
+usunObserwatora(IObserwator o)
+powiadomObserwatora()
1 do wielu
klasaObserwator
+aktualizacjaDanych()

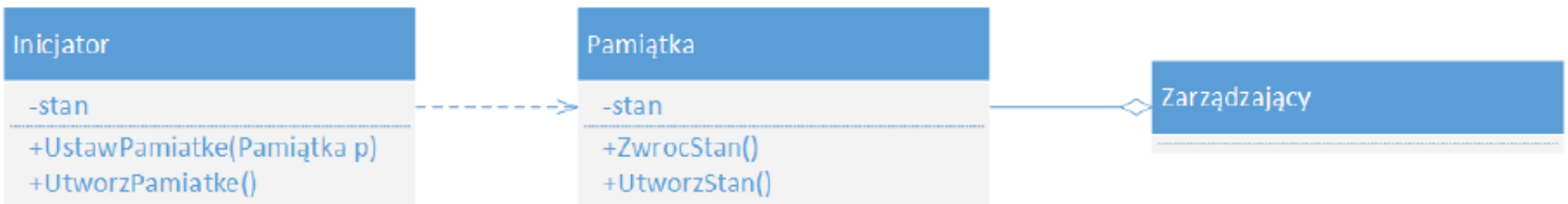
Odwiedzający (wizytator)

Odwiedzający (wizytator) – wzorzec projektowy, którego zadaniem jest odseparowanie algorytmu od struktury obiektowej na której operuje. Praktycznym rezultatem tego odseparowania jest możliwość dodawania nowych operacji do aktualnych struktur obiektów bez konieczności ich modyfikacji.


Client
Visitor
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Notation: UML
ConcreteVisitor1
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
ConcreteVisitor2
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Object structure
Element
accept(visitor : Visitor) : void
ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void

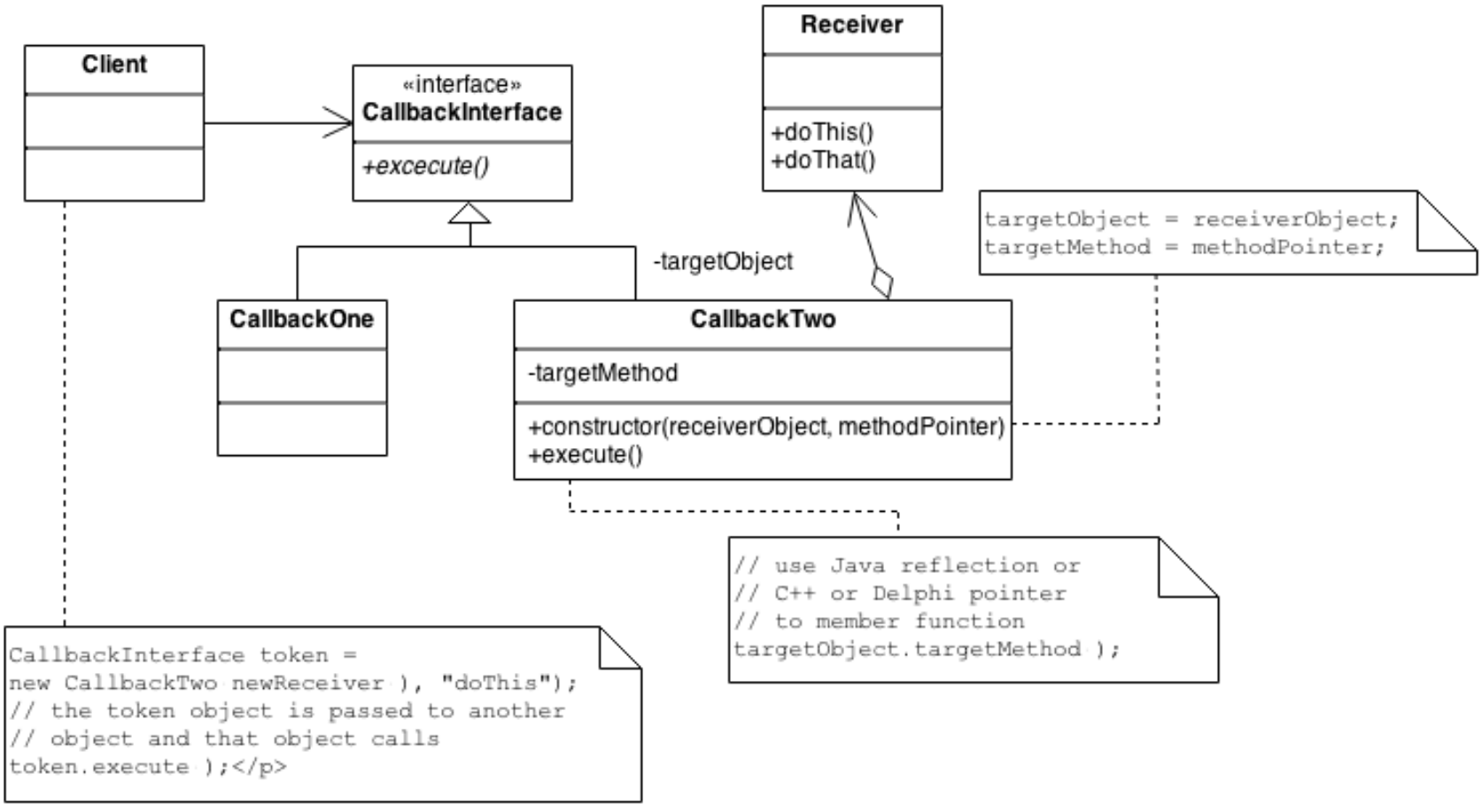
Pamiątka (znacznik)

Pamiątka – czynnościowy wzorzec projektowy. Jego zadaniem jest zapamiętanie i udostępnienie na zewnątrz wewnętrznego stanu obiektu bez naruszania hermetyzacji. Umożliwia to przywracanie zapamiętanego stanu obiektu.


Inicjator
-stan
+UstawPamiatke(Pamiątka p)
+UtworzPamiatke()
Pamiątka
-stan
+ZwrocStan()
+UtworzStan()
Zarządzający

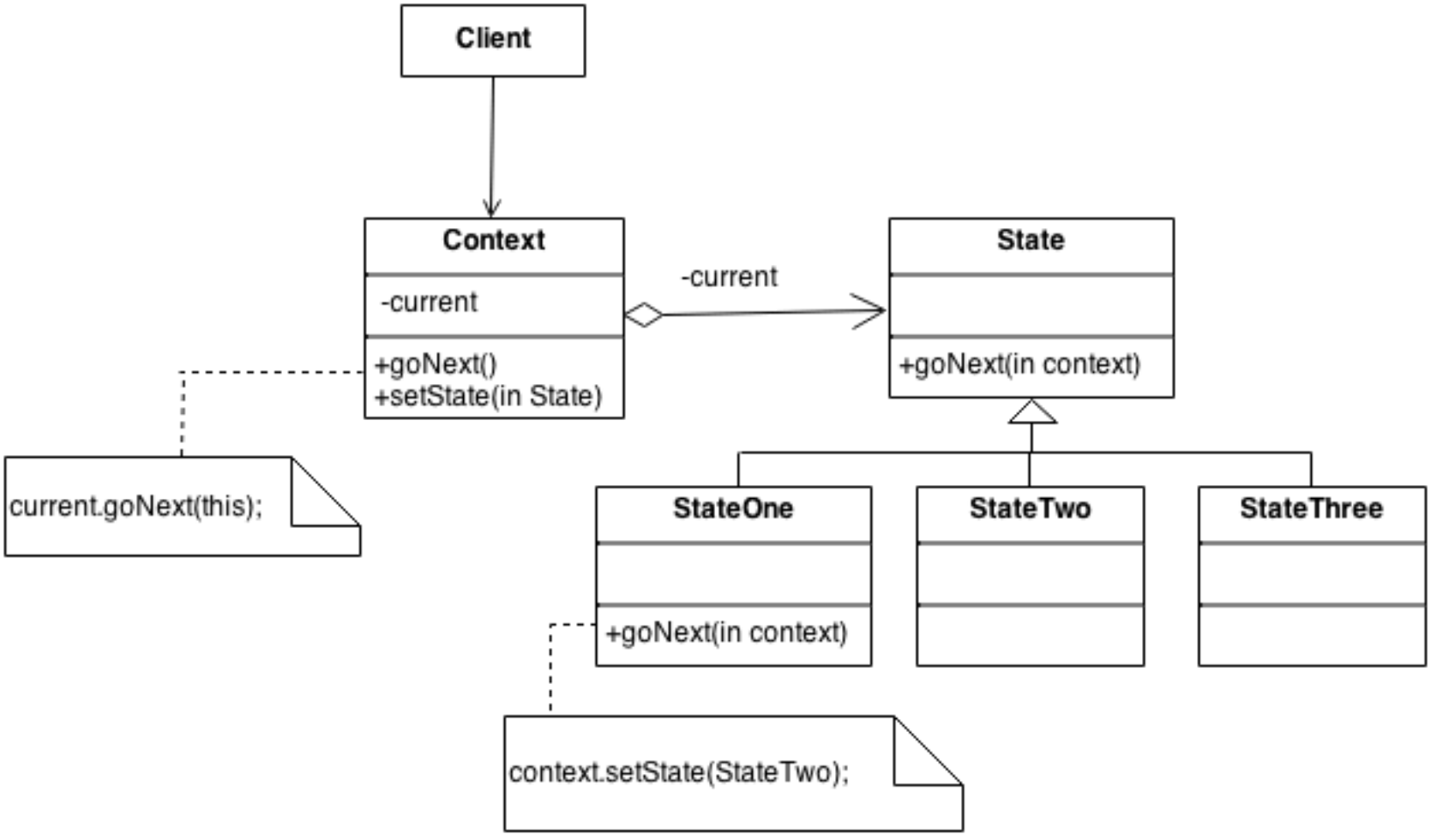
Polecenie

Polecenie – czynnościowy wzorzec projektowy, traktujący żądanie wykonania określonej czynności jako obiekt, dzięki czemu mogą być one parametryzowane w zależności od rodzaju odbiorcy, a także umieszczane w kolejkach i dziennikach.


Client
«interface»
CallbackInterface
+excecute()
Receiver
+doThis()
+doThat()
targetObject = receiverObject;
targetMethod = methodPointer;
-targetObject
CallbackOne
CallbackTwo
-targetMethod
+constructor(receiverObject, methodPointer)
+execute()
// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod);
CallbackInterface token =
new CallbackTwo newReceiver), "doThis");
// the token object is passed to another
// object and that object calls
token.execute);</p>

Stan

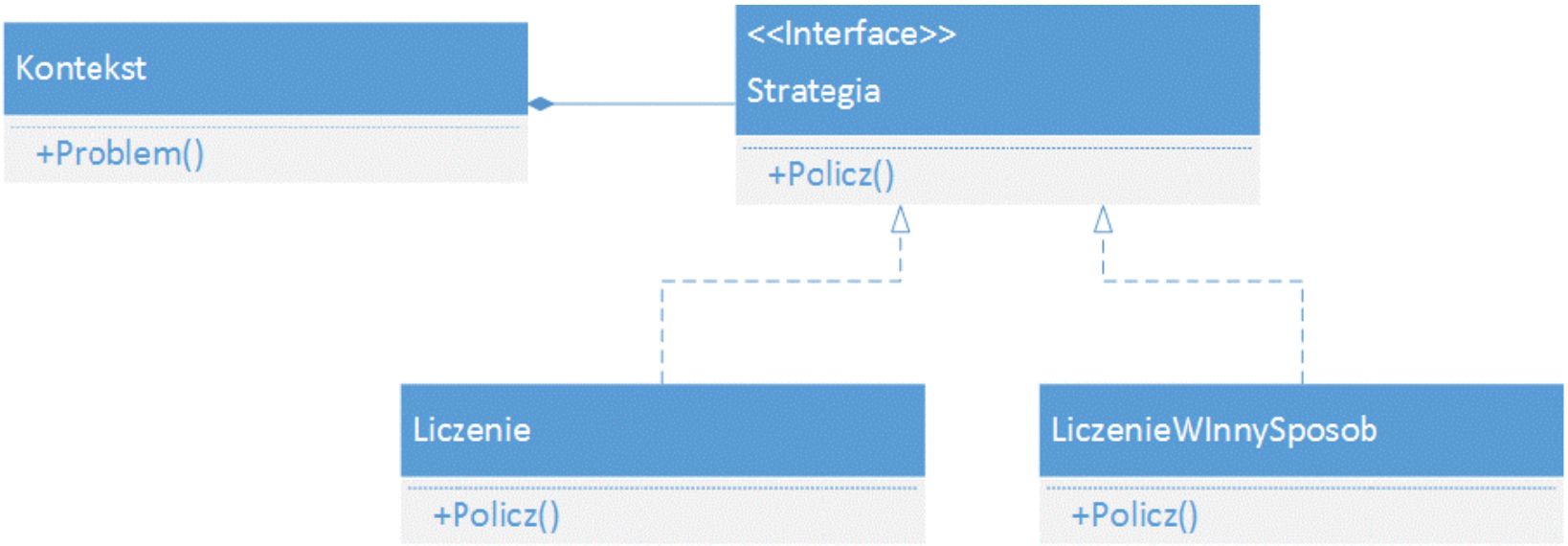
Stan – czynnościowy wzorzec projektowy, który umożliwia zmianę zachowania obiektu poprzez zmianę jego stanu wewnętrznego. Innymi słowy – uzależnia sposób działania obiektu od stanu w jakim się aktualnie znajduje.


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

Strategia (polityka)

Strategia – czynnościowy wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas. Umożliwia wymienne stosowanie każdego z nich w trakcie działania aplikacji niezależnie od korzystających z nich użytkowników.

http://www.altcontroldelete.pl/artykuly/wzorzec-strategia-przykladowa-implementacja-w-c-/


Kontekst
+Problem()
<<Interface>>
Strategia
+Policz()
Liczenie
+Policz()
LiczenieWInnySposob
+Policz()

Bibliografia

- Daniel Krasnokucki, Wzorce projektowe. Leksykon kieszonkowy, Wyd. Helion 2017.
- Wikipedia.
- http://devman.pl, dostęp online 15.03.2019.
- https://sourcemaking.com/, dostęp online 15.03.2019.
- https://www.oodesign.com/visitor-pattern.html, dostęp online 15.03.2019.
- https://medium.com/@sawomirkowalski/design-patterns-state-6e4ad27df0d7, dostęp online 15.03.2019.