

Wzorce projektowe w C++

Krystian Piękoś

Informacje organizacyjne

- Czas trwania szkolenia:

- 9:00 - 16:00

- Materiały szkoleniowe:

- <https://infotraining.bitbucket.io/cpp-dp>
- repozytorium GIT

- Przerwy

- Lista obecności

- Ankieta

Plan szkolenia

- SOLID OOP
- Wzorce projektowe - wprowadzenie
- Wzorce kreacyjne
- Wzorce strukturalne
- Wzorce behawioralne

Object Oriented Programming

Fundamenty OOP

Fundamenty OOP

■ Abstrakcja

Fundamenty OOP

- Abstrakcja
- Enkapsulacja

Fundamenty OOP

- Abstrakcja
- Enkapsulacja
- Polimorfizm

Fundamenty OOP

- Abstrakcja
- Enkapsulacja
- Polimorfizm
- Ponowne wykorzystanie kodu
 - kompozycja & dziedziczenie

Obiekt

- Zawiera zarówno dane składowe (*pola*) jak i implementację funkcji, które operują na tych danych (*funkcje składowe*)
- Wykonuje akcję po otrzymaniu żądania (*request*) od klienta
- Wewnętrzny stan jest zaenkapsulowany

Interfejs

- Zbiór operacji, które mogą być wykonane na obiekcie
- Nic nie mówi o implementacji
 - różne obiekty mające ten sam interfejs mogą różnie go implementować

Klasa

- Definiuje reprezentację (dane) oraz zachowanie (implementację) dla obiektu
- Obiekty są instancjami danej klasy
- Za pomocą dziedziczenia klas można definiować **nowe klasy** wykorzystując kod klas, które już istnieją

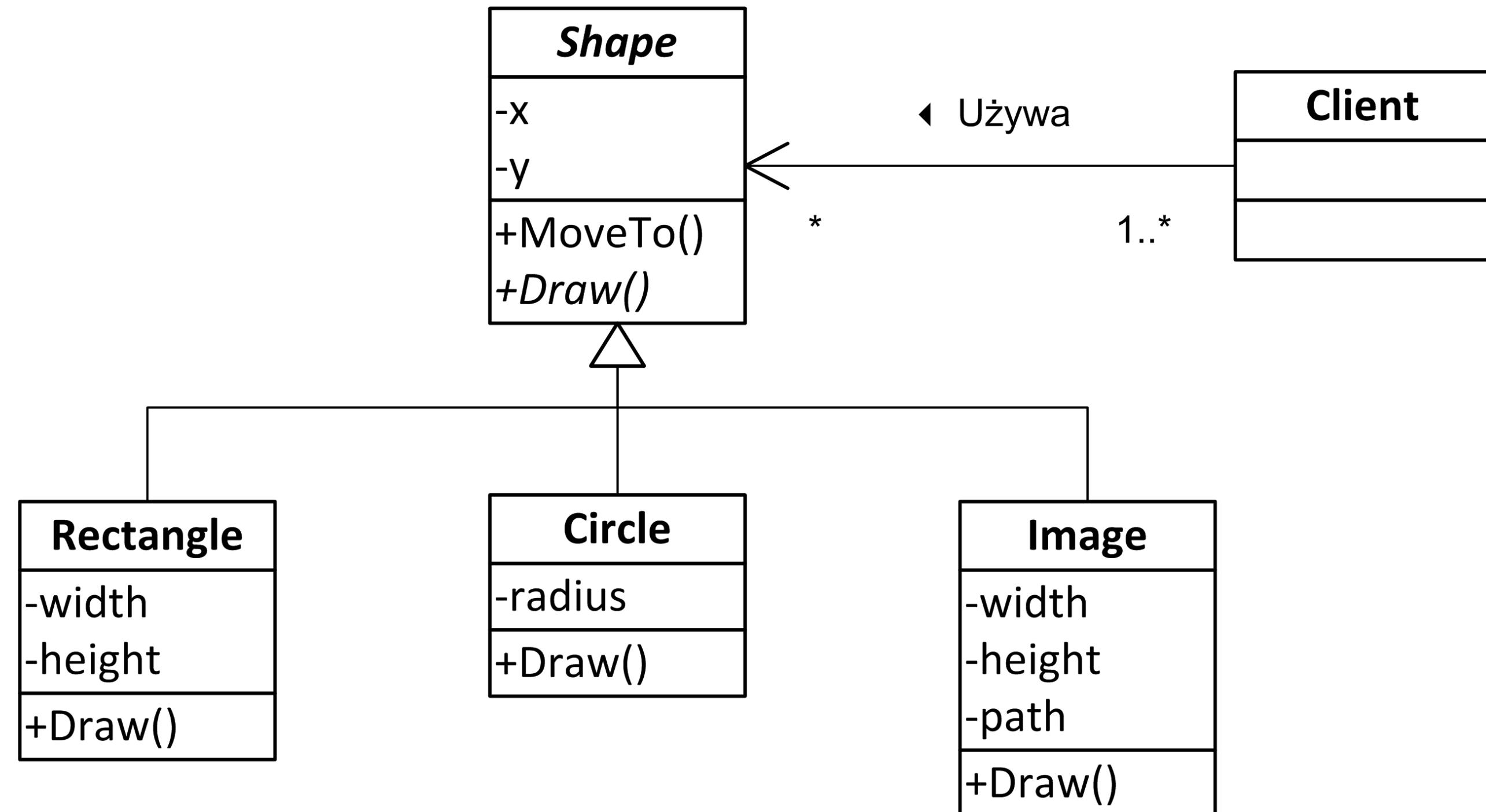
Klasa abstrakcyjna

Klasa abstrakcyjna

- Definiuje interfejs dla klientów

Klasa abstrakcyjna

- Definiuje interfejs dla klientów
- Operacje, które klasa abstrakcyjna deklaruje, ale których nie implementuje, nazywa się operacjami abstrakcyjnymi
 - w C++ są to **metody czysto wirtualne**



Klasa abstrakcyjna - kod

```
01  class Shape {
02      int x_, y_;
03  public:
04      Shape(int x = 0, int y = 0) : x_{x}, y_{y}
05      {}
06
07      virtual ~Shape() = default;
08
09      virtual void move(int dx, int dy) {
10          x_ += dx;
11          y_ += dy;
12      }
13
14      virtual void draw() const = 0;
15  };
```

Ekstrakcja interfejsu

```
01 class Shape {
02 public:
03     virtual ~Shape() = default;
04
05     virtual void move(int dx, int dy) = 0;
06     virtual void draw() const = 0;
07 };
08
09 class ShapeBase : public Shape {
10     Point coord_;
11 public:
12     void move(int dx, int dy) override
13     {
14         coord_.x += dx;
15         coord_.y += dy;
16     }
17 };
```

Ekstrakcja interfejsu

```
01 class Shape {
02 public:
03     virtual ~Shape() = default;
04
05     virtual void move(int dx, int dy) = 0;
06     virtual void draw() const = 0;
07 };
08
09 class ShapeBase : public Shape {
10     Point coord_;
11 public:
12     void move(int dx, int dy) override
13     {
14         coord_.x += dx;
15         coord_.y += dy;
16     }
17 };
```

Ekstrakcja interfejsu

```
01 class Shape {
02 public:
03     virtual ~Shape() = default;
04
05     virtual void move(int dx, int dy) = 0;
06     virtual void draw() const = 0;
07 };
08
09 class ShapeBase : public Shape {
10     Point coord_;
11 public:
12     void move(int dx, int dy) override
13     {
14         coord_.x += dx;
15         coord_.y += dy;
16     }
17 };
```

Polimorfizm

Polimorfizm

- Zapewnienie tego samego interfejsu dla wielu obiektów różnych typów

Polimorfizm

- Zapewnienie tego samego interfejsu dla wielu obiektów różnych typów
- Umożliwia zastępowanie w czasie wykonywania programu jednych obiektów drugimi, jeśli mają identyczny interfejs

Rodzaje polimorfizmu

Rodzaje polimorfizmu

- Dynamiczny

Rodzaje polimorfizmu

■ Dynamiczny

- implementacja jest wiązana z wywołaniem w trakcie działania programu - późne wiązanie
 - dziedziczenie publiczne i nadpisywanie metod z klasy bazowej
 - duck typing (np. Python)

Rodzaje polimorfizmu

■ Dynamiczny

- implementacja jest wiązana z wywołaniem w trakcie działania programu - późne wiązanie
 - dziedziczenie publiczne i nadpisywanie metod z klasy bazowej
 - duck typing (np. Python)

■ Statyczny

- działa na etapie kompilacji
 - szablony

Podstawowe techniki OOP

Podstawowe techniki OOP

■ Dziedziczenie

Podstawowe techniki OOP

- Dziedziczenie
- Kompozycja

Podstawowe techniki OOP

- Dziedziczenie
- Kompozycja
- Delegacja

Dziedziczenie

Dziedziczenie implementacji

- Definiuje implementację danego obiektu wykorzystując implementację innego obiektu
- Mechanizm współdzielenia kodu
- C++ - dziedziczenie prywatne

```
01 class Bar
02 {
03     int data_;
04 public:
05     void request();
06 };
07
08 class Foo : private Bar
09 {
10 public:
11     void another_request();
12 };
```

Dziedziczenie interfejsu

- Określa, kiedy jeden obiekt może być używany zamiast drugiego
- C++ - publiczne dziedziczenie po klasie, która ma (czysto) wirtualne funkcje składowe

Dziedziczenie - wady (?/!)

Dziedziczenie - wady (?/!)

- jest statyczne
 - zachowanie (implementacja) jest związana z typem

Dziedziczenie - wady (?/!)

- jest statyczne
 - zachowanie (implementacja) jest związana z typem
- narusza enkapsulację
 - pola **protected** - implementacja typu pochodnego może zależeć od szczegółów implementacji typu bazowego

OOP Tip #1

Program to an interface, not an implementation!

Kompozycja

Kompozycja

- jest definiowana dynamicznie (runtime)

Kompozycja

- jest definiowana dynamicznie (runtime)
- nie może naruszyć enkapsulacji

Kompozycja

- jest definiowana dynamicznie (runtime)
- nie może naruszyć enkapsulacji
- pozwala tworzyć typy zgodne z SRP

OOP Tip #2

Favor object composition over class inheritance!

Delegacja

Delegacja

- bardziej uniwersalny od dziedziczenia sposób rozszerzania zachowania klasy

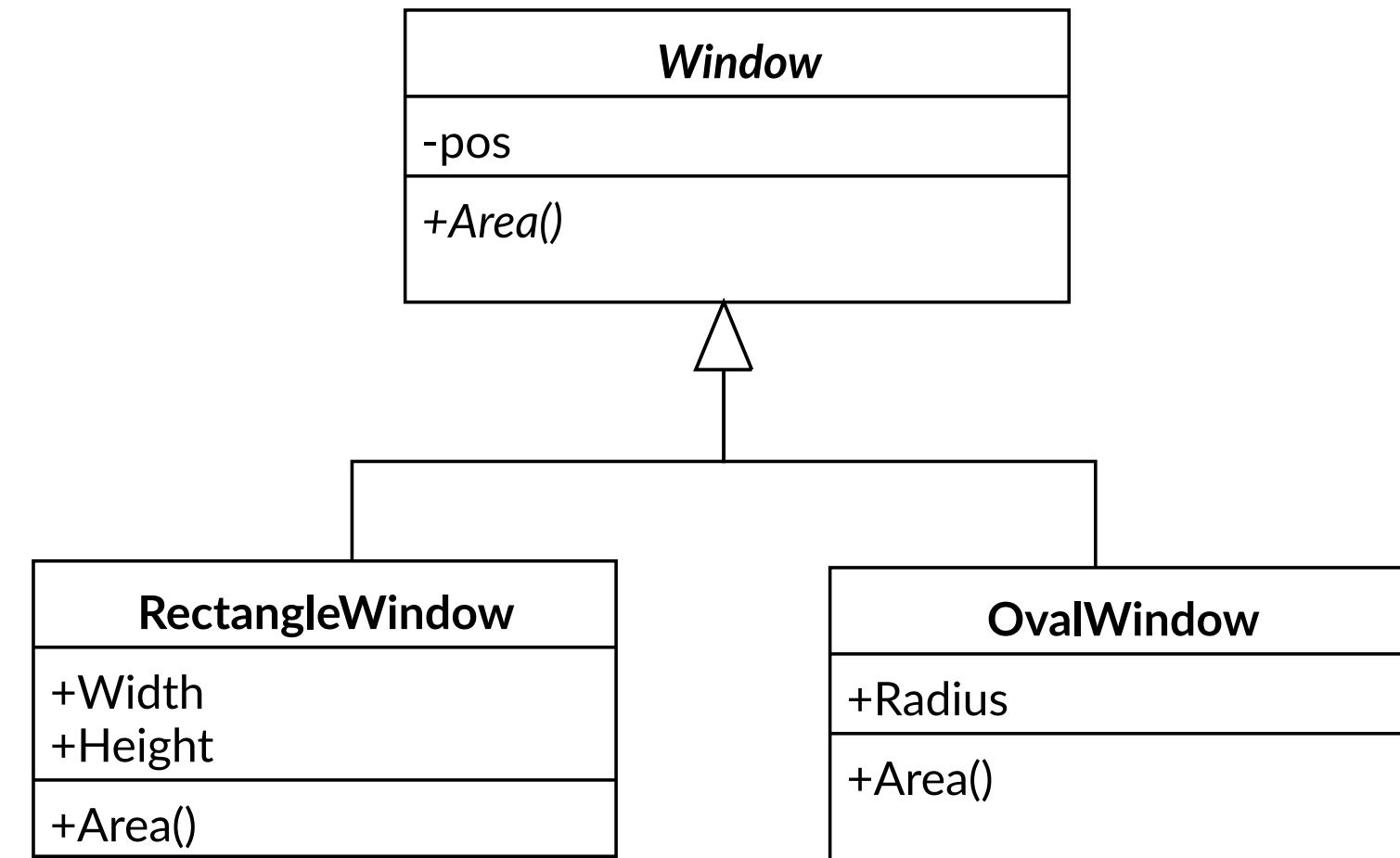
Delegowanie żądań

- Dwa obiekty są zaangażowane w obsługę żądania

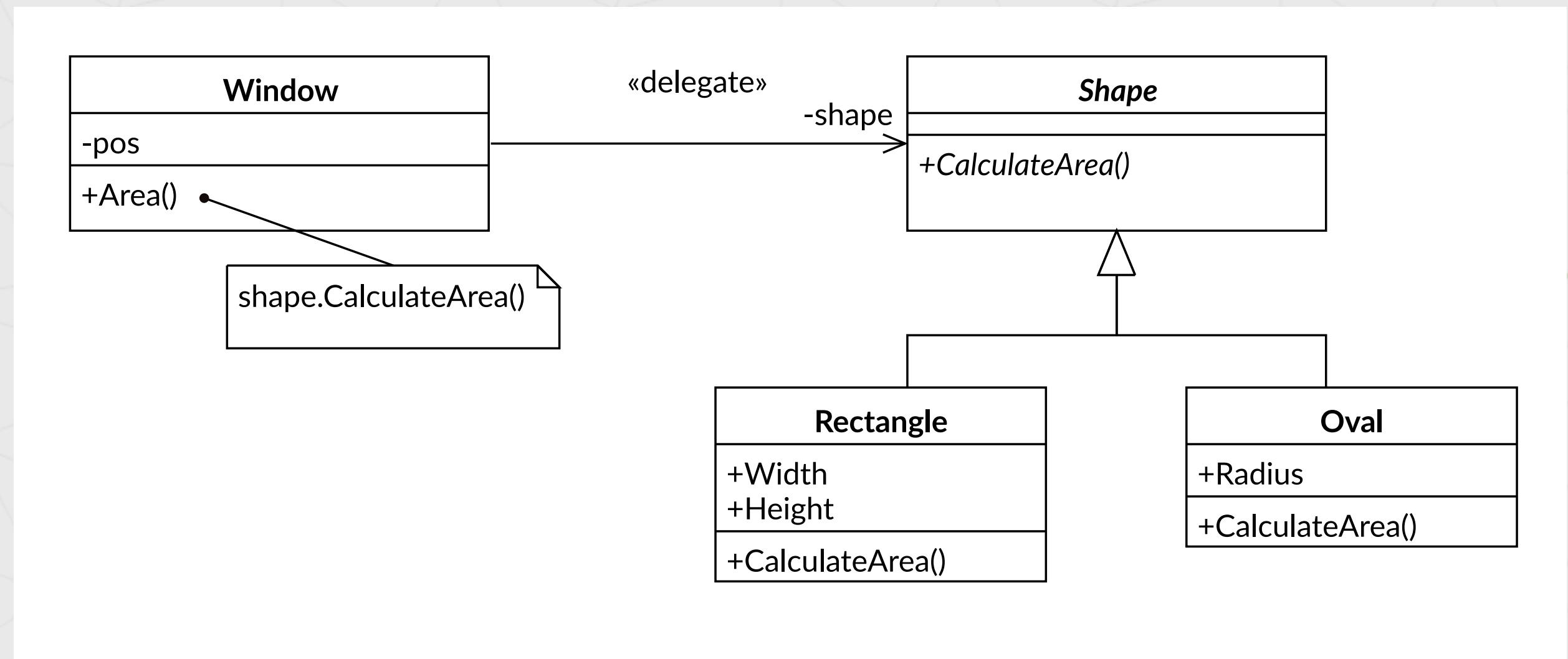
Delegowanie żądań

- Dwa obiekty są zaangażowane w obsługę żądania
 - obiekt przyjmujący żądanie przekazuje operacje swojemu **delegatowi**

Delegacja vs. Dziedziczenie



■ to samo ale z użyciem delegacji



Delegacja - zalety & wady

- Zalety
- Wady

Delegacja - zalety & wady

■ Zalety

- umożliwia składanie zachowań w czasie wykonywania programu - obiekt przyjmujący żądanie może zmieniać swoje zachowanie

■ Wady

Delegacja - zalety & wady

■ Zalety

- umożliwia składanie zachowań w czasie wykonywania programu - obiekt przyjmujący żądanie może zmieniać swoje zachowanie

■ Wady

- dynamiczne, wysoce sparametryzowane oprogramowanie jest trudniej zrozumieć niż oprogramowanie statyczne

Zasady OOP

- Dobre projekty zorientowane obiektowo:

Zasady OOP

- Dobre projekty zorientowane obiektowo:
 - Powinny nadawać się do wielokrotnego użytku

Zasady OOP

- Dobre projekty zorientowane obiektowo:
 - Powinny nadawać się do wielokrotnego użytku
 - Być proste do rozbudowy

Zasady OOP

- Dobre projekty zorientowane obiektowo:
 - Powinny nadawać się do wielokrotnego użytku
 - Być proste do rozbudowy
 - Być łatwe w serwisowaniu i modyfikacji

Zasady OOP

- Dobre projekty zorientowane obiektowo:
 - Powinny nadawać się do wielokrotnego użytku
 - Być proste do rozbudowy
 - Być łatwe w serwisowaniu i modyfikacji
 - Być łatwe w testowaniu

S.O.L.I.D. OOP

- Single Responsibility Principle
- Opened-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

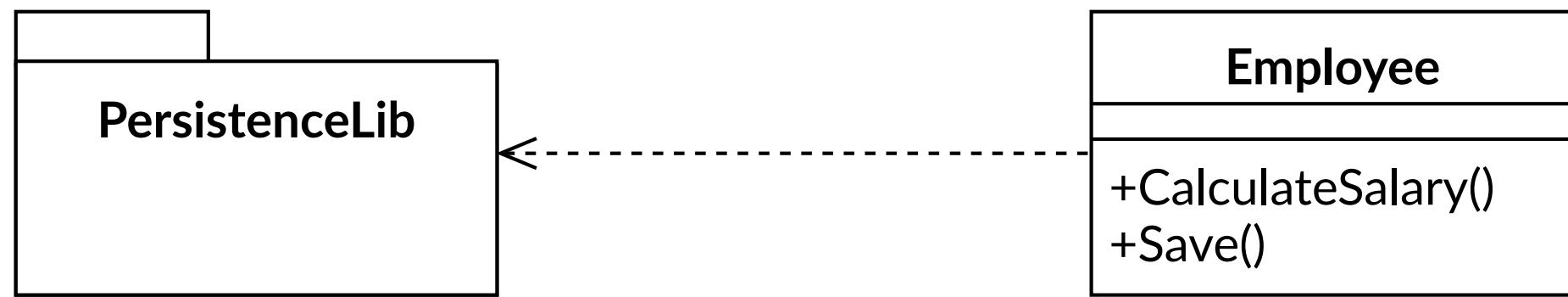
- każdy obiekt w kodzie powinien mieć tylko jedną odpowiedzialność, a wszystkie usługi tego obiektu powinny koncentrować się na jej realizacji



Każda klasa powinna mieć tylko jeden

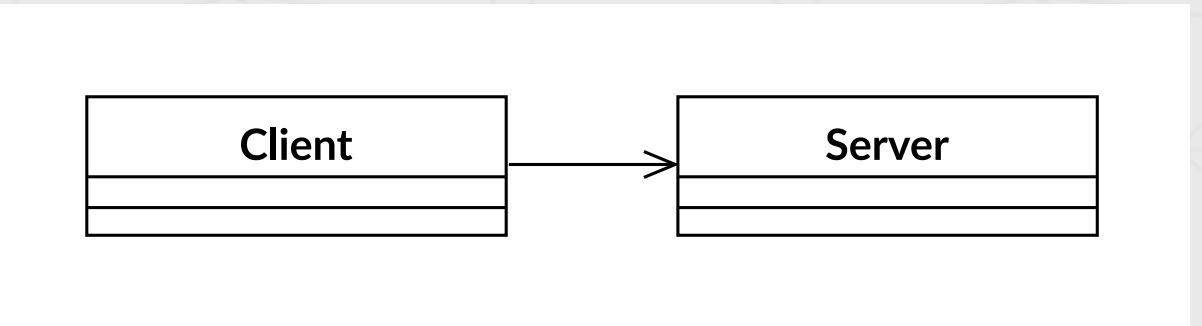


Każda klasa powinna mieć tylko jeden powód do modyfikacji!

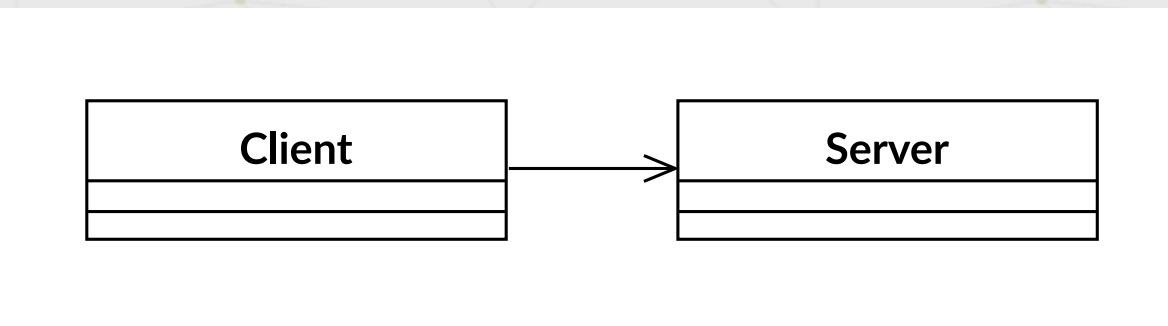


Open-Closed Principle

- Klasy powinny być otwarte na rozbudowę i zamknięte na modyfikacje.

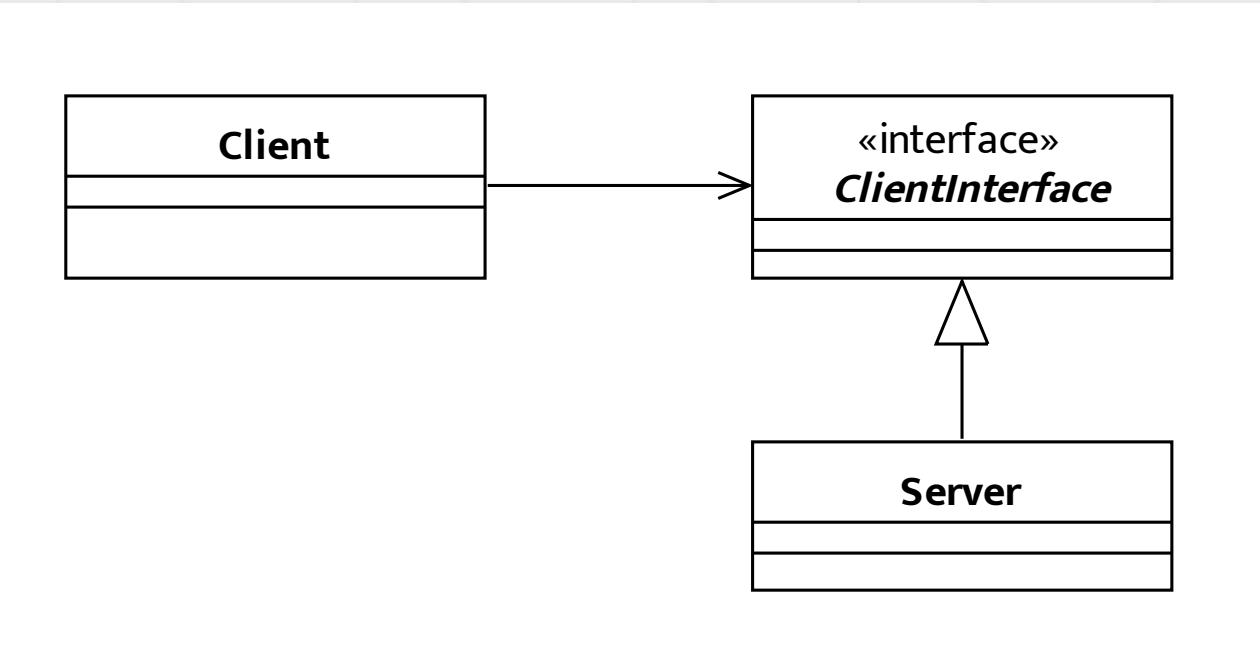


Naruszenie zasady OCP



Poprawione rozwiązanie - wprowadzenie interfejsu

Poprawione rozwiązanie - wprowadzenie interfejsu



Liskov Substitution Principle

- Musi istnieć możliwość podstawiania typów pochodnych w miejsce ich typów bazowych

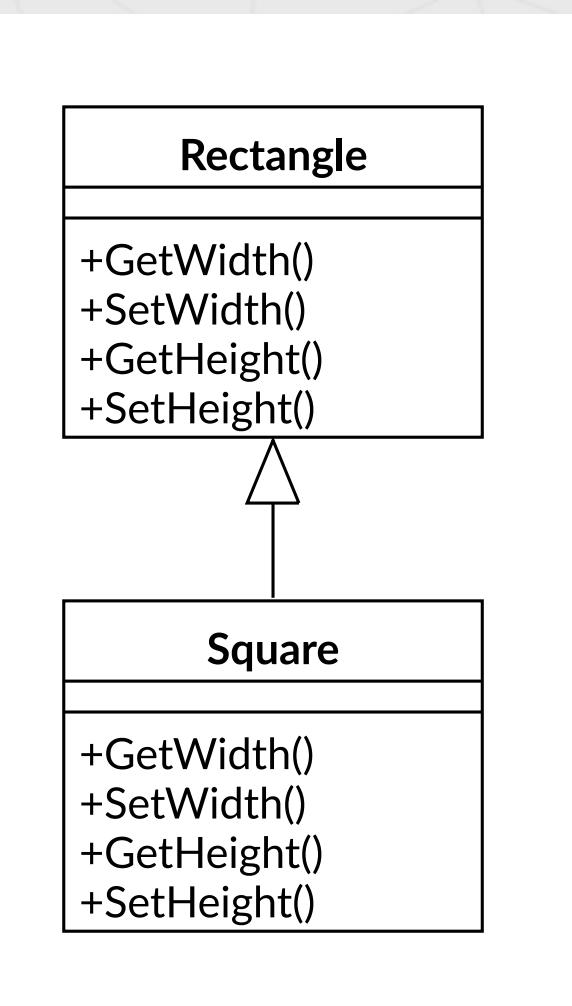
Jeżeli **S** jest podtypem **T**, wtedy obiekty typu **T** mogą zostać zastąpione
instancjami typu **S** bez naruszenia istotnych właściwości programu
(niezmienników, poprawności, itp.).

Design by contract

- Pre-conditions cannot be strengthened in a subtype
- Post-conditions cannot be weakened in a subtype
- Invariants of the supertype must be preserved in a subtype

Naruszenie zasady LSP

Naruszenie zasady LSP

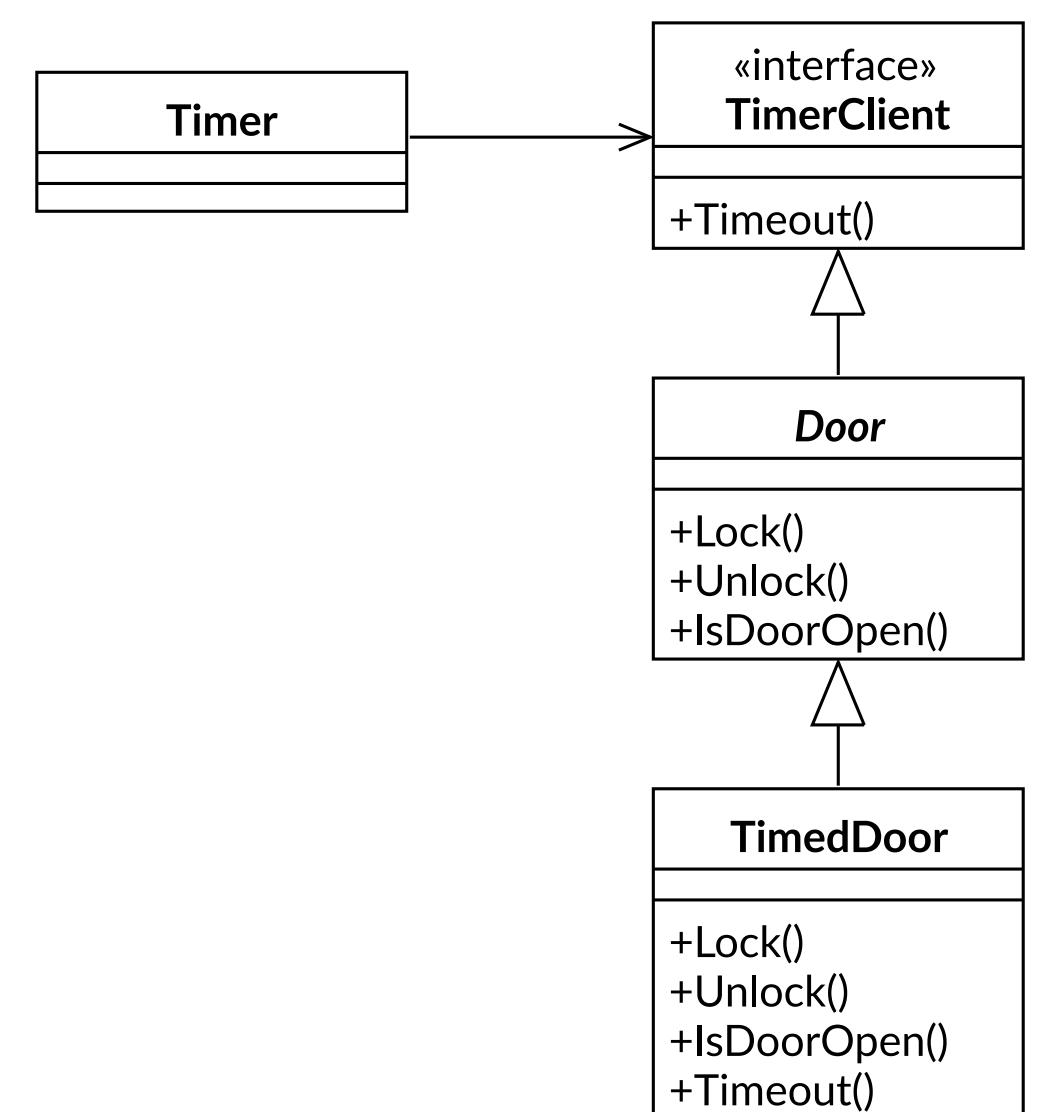


Interface Segregation Principle

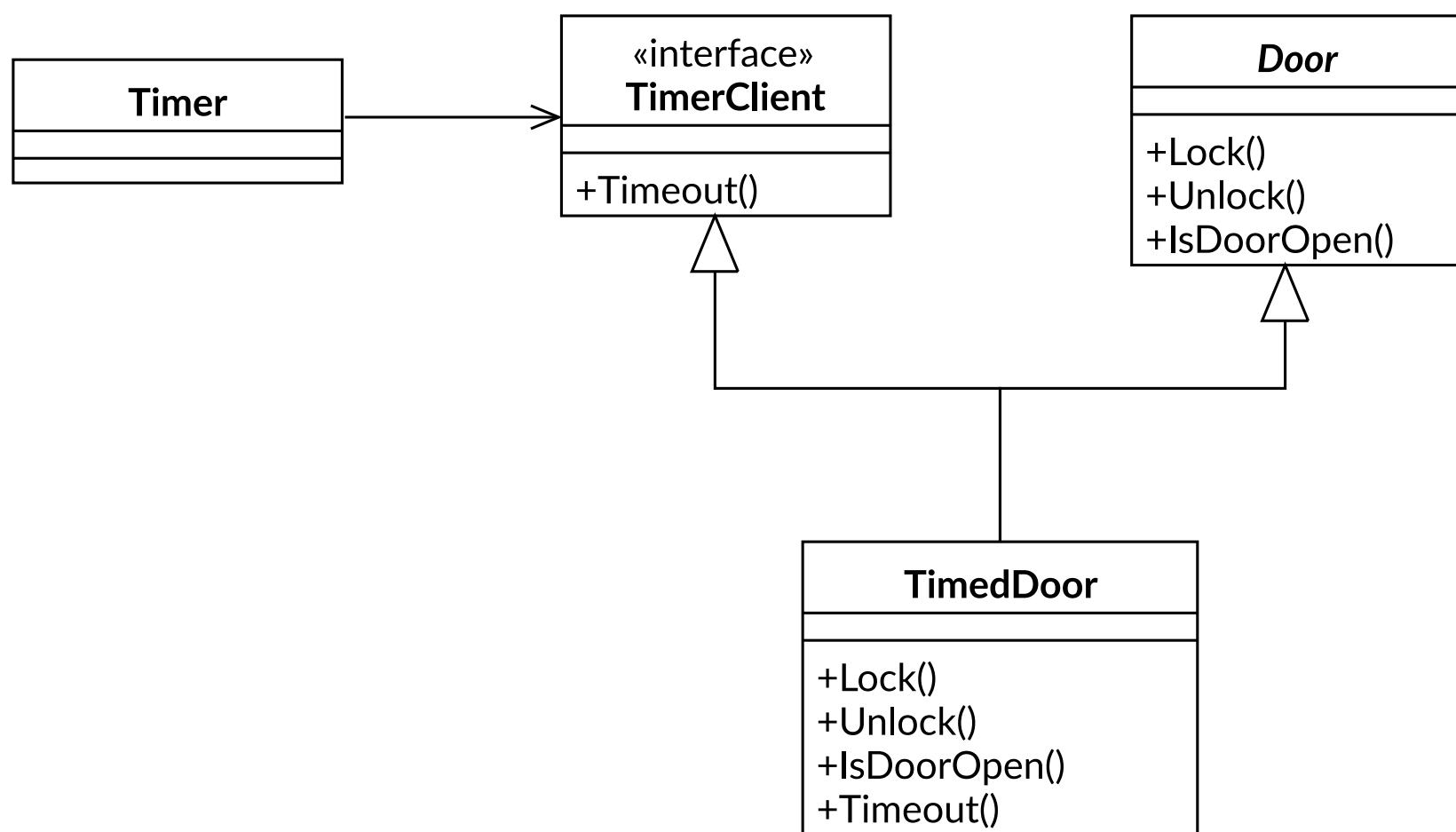
- Klient nie powinien być zmuszany do zależności od metod, których nie używa.

Naruszenie ISP

Naruszenie ISP



ISP - lepsze rozwiązanie



Dependency Inversion Principle

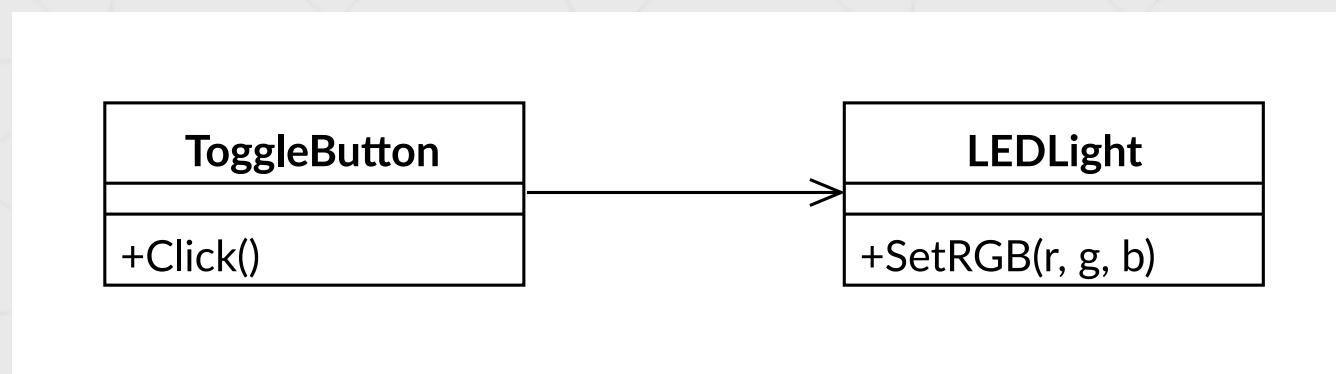
Dependency Inversion Principle

- Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji

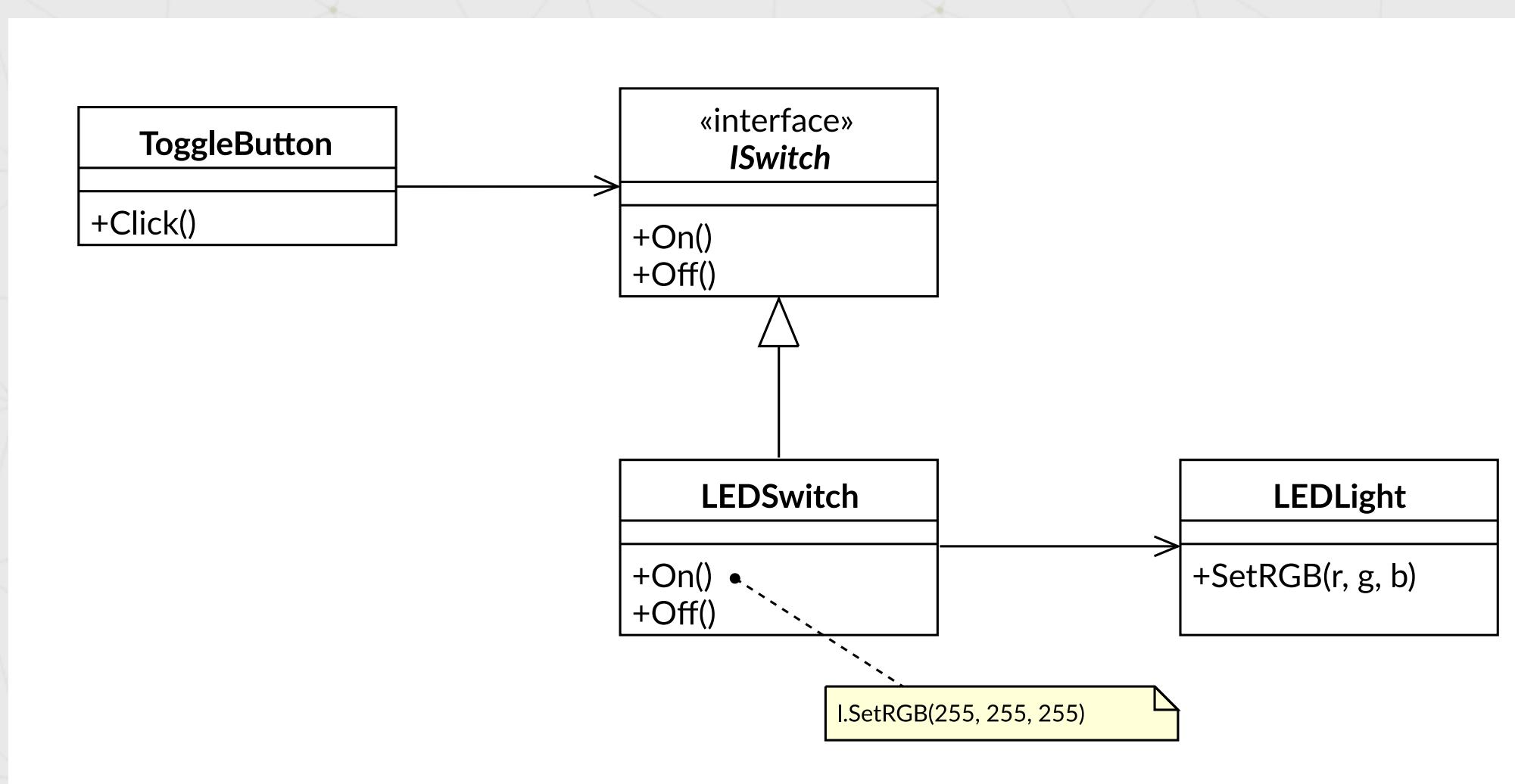
Dependency Inversion Principle

- Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji
- Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji

Naruszenie DIP



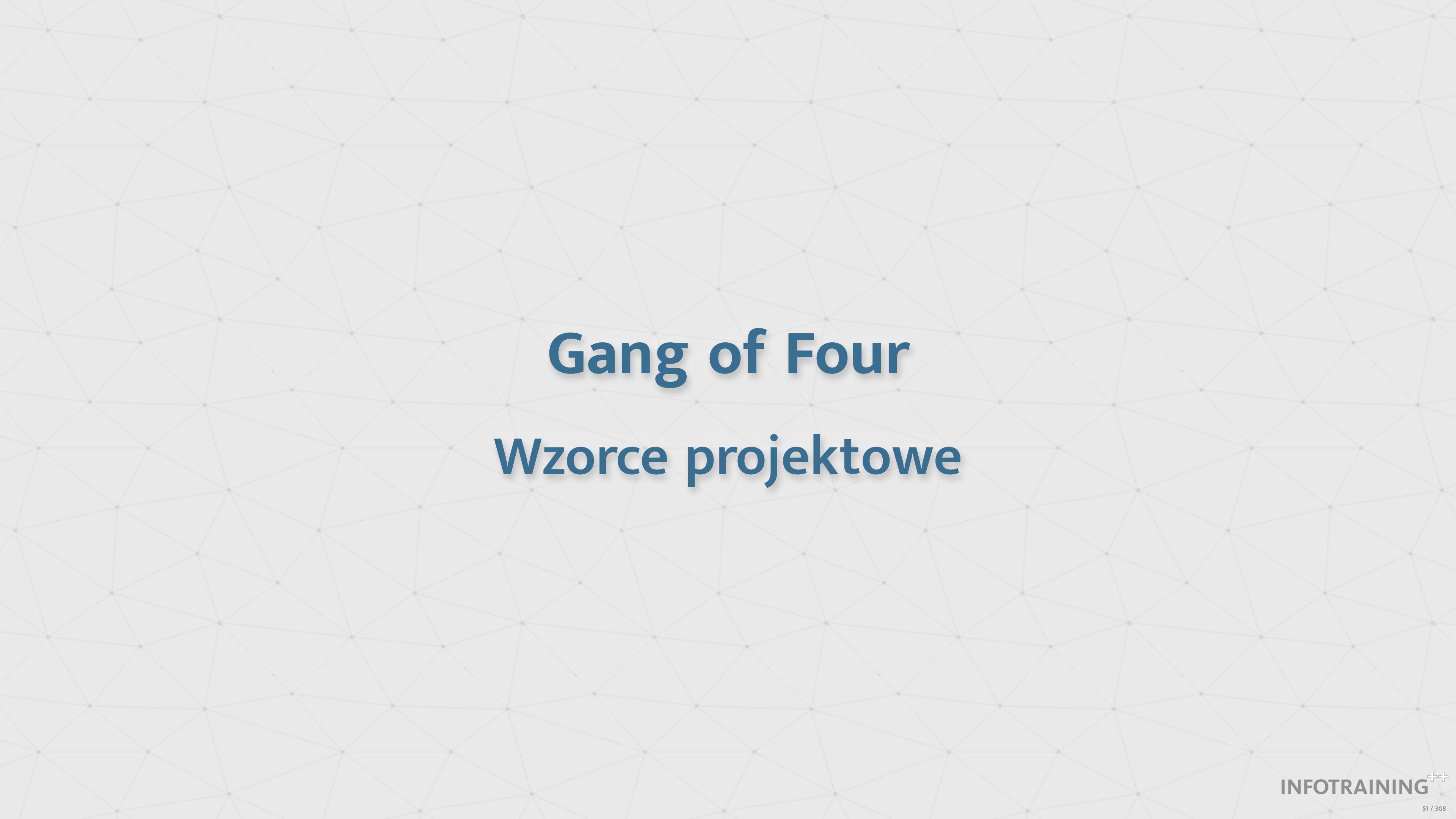
DIP



Wzorce projektowe

“Wzorzec opisuje **problem występujący wielokrotnie w danym środowisku, pokazując **podstawowe rozwiązanie** tego problemu dane w taki sposób, aby można wielokrotnie użyć tego **rozwiązania** do wszystkich wystąpień danego **problemu**, bez konieczności ponownego wykonywania tych samych czynności projektowych”**

- Christopher Alexander - "A Pattern Language"



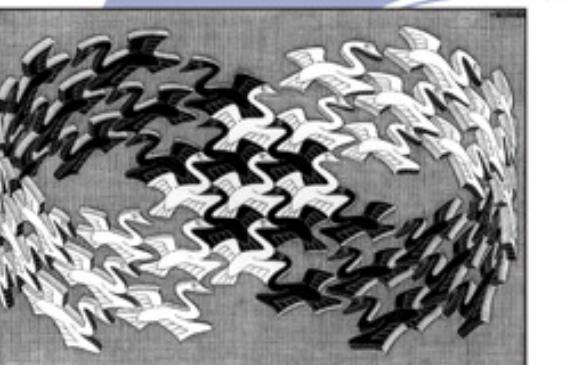
Gang of Four

Wzorce projektowe

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

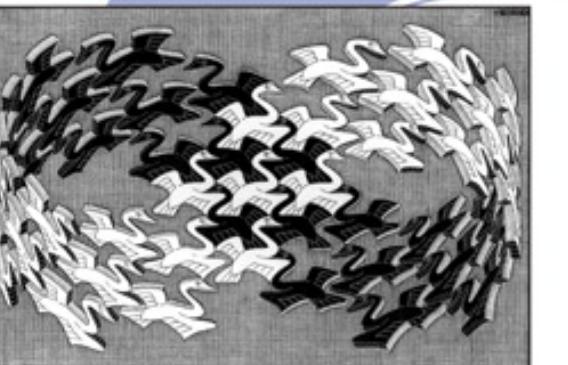


ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

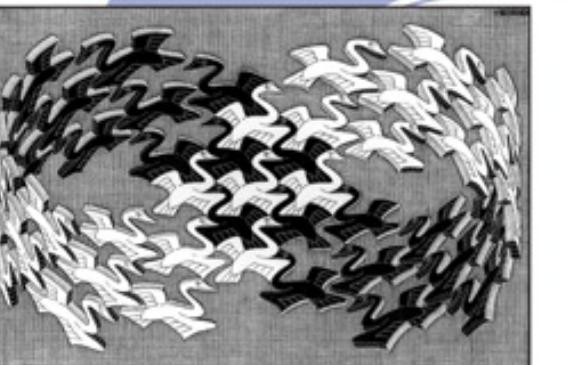


■ Erich Gamma

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

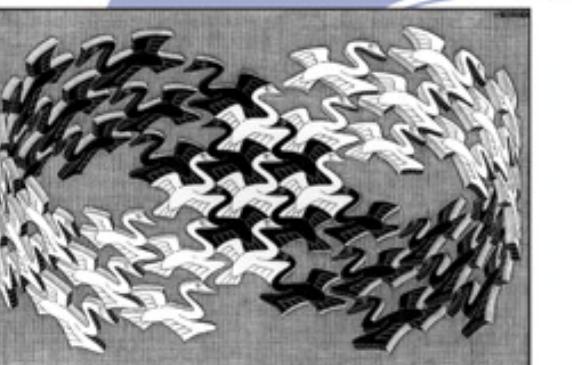


- Erich Gamma
- Richard Helm

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

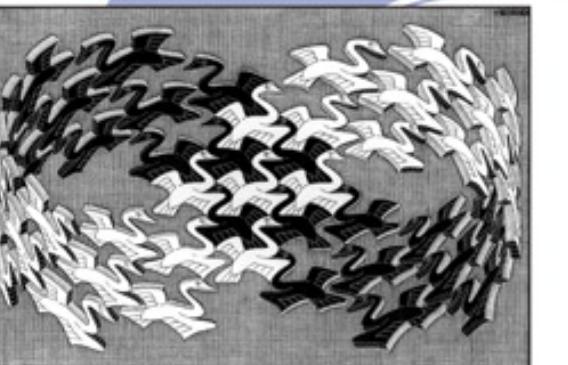


- Erich Gamma
- Richard Helm
- Ralph Johnson

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Wzorce projektowe - zalety

Wzorce projektowe - zalety

- Znakomicie sprawdziły się w wielu rzeczywistych aplikacjach systemów zorientowanych obiektowo

Wzorce projektowe - zalety

- Znakomicie sprawdziły się w wielu rzeczywistych aplikacjach systemów zorientowanych obiektowo
- Wskazują sposoby tworzenia całych systemów posiadających cechy charakterystyczne dobrych projektów zorientowanych obiektowo

Wzorce projektowe - zalety

- Znakomicie sprawdziły się w wielu rzeczywistych aplikacjach systemów zorientowanych obiektowo
- Wskazują sposoby tworzenia całych systemów posiadających cechy charakterystyczne dobrych projektów zorientowanych obiektowo
- Nie udostępniają gotowego kodu, a jedynie ogólne sposoby rozwiązywania problemów pojawiających się w fazie projektowania

Wzorce projektowe - zalety

Wzorce projektowe - zalety

- Wzorce zapewniają rodzaj wspólnego, jednolitego języka, który może maksymalizować efektywność komunikacji pomiędzy poszczególnymi członkami zespołu

Wzorce projektowe - zalety

- Wzorce zapewniają rodzaj wspólnego, jednolitego języka, który może maksymalizować efektywność komunikacji pomiędzy poszczególnymi członkami zespołu
- Większość wzorców umożliwia modyfikowanie pewnych fragmentów systemu całkowicie niezależnie od pozostałych elementów systemu

Wzorce projektowe - zalety

- Wzorce zapewniają rodzaj wspólnego, jednolitego języka, który może maksymalizować efektywność komunikacji pomiędzy poszczególnymi członkami zespołu
- Większość wzorców umożliwia modyfikowanie pewnych fragmentów systemu całkowicie niezależnie od pozostałych elementów systemu
- Wzorce zwykle odnoszą się do sytuacji, w których w danym oprogramowaniu muszą zostać dokonane określone zmiany - umożliwiają hermetyzację elementów wykazujących się częstą zmiennością

Wzorzec projektowy

- Wzorzec dostarcza abstrakcyjnego opisu problemu projektowego i tego jak ogólny układ elementów (klas i obiektów) rozwiązuje ten problem.

Wzorzec projektowy

ma cztery zasadnicze elementy:

Wzorzec projektowy

ma cztery zasadnicze elementy:

- **Nazwa wzorca** – skrót, którego można użyć do zwięzłego określenia problemu projektowego. Umożliwia projektowanie na wyższym poziomie abstrakcji

Wzorzec projektowy

ma cztery zasadnicze elementy:

- **Nazwa wzorca** – skrót, którego można użyć do zwięzłego określenia problemu projektowego. Umożliwia projektowanie na wyższym poziomie abstrakcji
- **Problem** – określa, kiedy stosować dany wzorzec

Wzorzec projektowy

ma cztery zasadnicze elementy:

- **Nazwa wzorca** – skrót, którego można użyć do zwięzłego określenia problemu projektowego. Umożliwia projektowanie na wyższym poziomie abstrakcji
- **Problem** – określa, kiedy stosować dany wzorzec
- **Rozwiązanie** – ogólny opis elementów składających się na rozwiązanie zdefiniowanego problemu

Wzorzec projektowy

ma cztery zasadnicze elementy:

- **Nazwa wzorca** – skrót, którego można użyć do zwięzłego określenia problemu projektowego. Umożliwia projektowanie na wyższym poziomie abstrakcji
- **Problem** – określa, kiedy stosować dany wzorzec
- **Rozwiązanie** – ogólny opis elementów składających się na rozwiązanie zdefiniowanego problemu
- **Konsekwencje** – zalety oraz wady zastosowania wzorca

Klasyfikacja wzorców wg GOF

- Kreacyjne (Creational)
- Strukturalne (Structural)
- Czynnościowe (Behavioral)

Katalog wzorców projektowych

Kreacyjne	Strukturalne	Behawioralne
Factory Method	Adapter	Interpreter
Abstract Factory	Decorator	Template Method
Prototype	Composite	Strategy
Builder	Proxy	State
Singleton	Facade	Chain Of Responsibility
	Bridge	Observer
	Flyweight	Command
		Mediator
		Iterator
		Visitor

Wzorce kreacyjne

Wzorce kreacyjne

- Factory Method

Wzorce kreacyjne

- Factory Method
- Abstract Factory

Wzorce kreacyjne

- Factory Method
- Abstract Factory
- Prototype

Wzorce kreacyjne

- Factory Method
- Abstract Factory
- Prototype
- Builder

Wzorce kreacyjne

- Factory Method
- Abstract Factory
- Prototype
- Builder
- Singleton

Wzorce kreacyjne - wprowadzenie

Wzorce kreacyjne - wprowadzenie

- Wzorce kreacyjne prowadzą do utworzenia obiektu

Wzorce kreacyjne - wprowadzenie

- Wzorce kreacyjne prowadzą do utworzenia obiektu
- Prowadzą do uabstrakcyjnienia procesu tworzenia obiektów

Wzorce kreacyjne - wprowadzenie

- Wzorce kreacyjne prowadzą do utworzenia obiektu
- Prowadzą do uabstrakcyjnienia procesu tworzenia obiektów
- Ułatwiają budowę systemu niezależnego od sposobu tworzenia, składania i reprezentowania stosowanych w nim obiektów

Wzorce kreacyjne - wprowadzenie

- Wzorce kreacyjne prowadzą do utworzenia obiektu
- Prowadzą do uabstrakcyjnienia procesu tworzenia obiektów
- Ułatwiają budowę systemu niezależnego od sposobu tworzenia, składania i reprezentowania stosowanych w nim obiektów
- Hermetyzują wiedzę o tym, jakich klas konkretnych używa system i w jaki sposób egzemplarze tych klas są tworzone i łączone ze sobą

Wzorce kreacyjne - wprowadzenie

Wzorce kreacyjne - wprowadzenie

- Umożliwiają konfigurowanie systemu z obiektami-produktami, które w znacznym stopniu różnią się strukturą i funkcjonalnością

Wzorce kreacyjne - wprowadzenie

- Umożliwiają konfigurowanie systemu z obiektami-produktami, które w znacznym stopniu różnią się strukturą i funkcjonalnością
 - Konfiguracja statyczna (podczas komplikacji)

Wzorce kreacyjne - wprowadzenie

- Umożliwiają konfigurowanie systemu z obiektami-produktami, które w znacznym stopniu różnią się strukturą i funkcjonalnością
 - Konfiguracja statyczna (podczas komplikacji)
 - Konfiguracja dynamiczna (w trakcie wykonywania programu)

Fabryki

“ Preferuj luźne powiązania między klasami

- Zasada dobrego projektowania OOP

Kod łamiący tą zasadę:

```
01 class Service
02 {
03     //...
04 public:
05     void run()
06     {
07         FileLogger logger("log.dat");
08         logger.log("Service::run() called");
09     }
10 };
```

Kod łamiący tą zasadę:

```
01 class Service
02 {
03     //...
04 public:
05     void run()
06     {
07         FileLogger logger("log.dat");
08         logger.log("Service::run() called");
09     }
10 };
```

- Klasy są silnie ze sobą związane (*strong coupling*)

Kod łamiący tą zasadę:

```
01 class Service
02 {
03     //...
04 public:
05     void run()
06     {
07         FileLogger logger("log.dat");
08         logger.log("Service::run() called");
09     }
10 };
```

- Klasy są silnie ze sobą związane (*strong coupling*)
- Trudno zmienić implementację logowania w serwisie

Kod łamiący tą zasadę:

```
01 class Service
02 {
03     //...
04 public:
05     void run()
06     {
07         FileLogger logger("log.dat");
08         logger.log("Service::run() called");
09     }
10 };
```

- Klasy są silnie ze sobą związane (*strong coupling*)
- Trudno zmienić implementację logowania w serwisie
- Klasa Service jest trudna do przetestowania jednostkowego

Fabryki

- Chcąc utworzyć obiekt, musimy dokładnie wiedzieć jaki jest jego typ

Fabryki

- Chcąc utworzyć obiekt, musimy dokładnie wiedzieć jaki jest jego typ
- Jednak czasami:

Fabryki

- Chcąc utworzyć obiekt, musimy dokładnie wiedzieć jaki jest jego typ
- Jednak czasami:
 - Chcemy tę dokładną wiedzę pozostawić w gestii kogoś innego

Fabryki

- Chcąc utworzyć obiekt, musimy dokładnie wiedzieć jaki jest jego typ
- Jednak czasami:
 - Chcemy tę dokładną wiedzę pozostawić w gestii kogoś innego
 - Dysponujemy informacją o typie obiektu, ale np. w postaci identyfikatora typu `std::string`

Factory Method

Factory Method

- Przeznaczenie
 - definiuje interfejs pozwalający na tworzenie obiektów, ale odpowiedzialność za tworzenie obiektów jest delegowana do klas pochodnych
 - Wykorzystuje mechanizm dziedziczenia pozwalając klasom pochodnym decydować, jakiej klasy obiekt zostanie utworzony

Factory Method - Scenariusz

Jak poprawić poniższy kod:

```
01 class Service
02 {
03     //...
04 public:
05     void run()
06     {
07         FileLogger logger("log.dat");
08         logger.log("Service::run() called");
09     }
10 };
```

■ Ekstrakcja interfejsu dla loggera:

```
01 class Logger
02 {
03 public:
04     virtual ~Logger() = default;
05     virtual void log(const std::string& msg) = 0;
06 };
```

■ Uabstrakcyjnienie procesu tworzenia instancji klasy Logger:

```
01 class LoggerCreator
02 {
03 public:
04     virtual ~LoggerCreator() = default;
05     std::unique_ptr<Logger> create_logger() = 0;
06 };
```

■ Wstrzyknięcie zależności od fabryki do klasy Service:

```
01 class Service
02 {
03     LoggerCreator& logger_creator_;
04 public:
05     Service(LoggerCreator& logger_creator)
06         : logger_creator_(logger_creator)
07     {}
08 };
```

- Oddelegowanie procesu kreacji loggera do obiektu fabryki:

```
01 void run()
02 {
03     auto logger = logger_creator_->create_logger();
04     logger->log("Service::run() called");
05 }
```

Factory Method - Kontekst

- Chcemy wprowadzić nową funkcjonalność poprzez napisanie nowej klasy i utworzenie instancji tej klasy

Factory Method - Problem

Factory Method - Problem

- Chcemy tworzyć instancje konkretnych klas za pomocą interfejsu

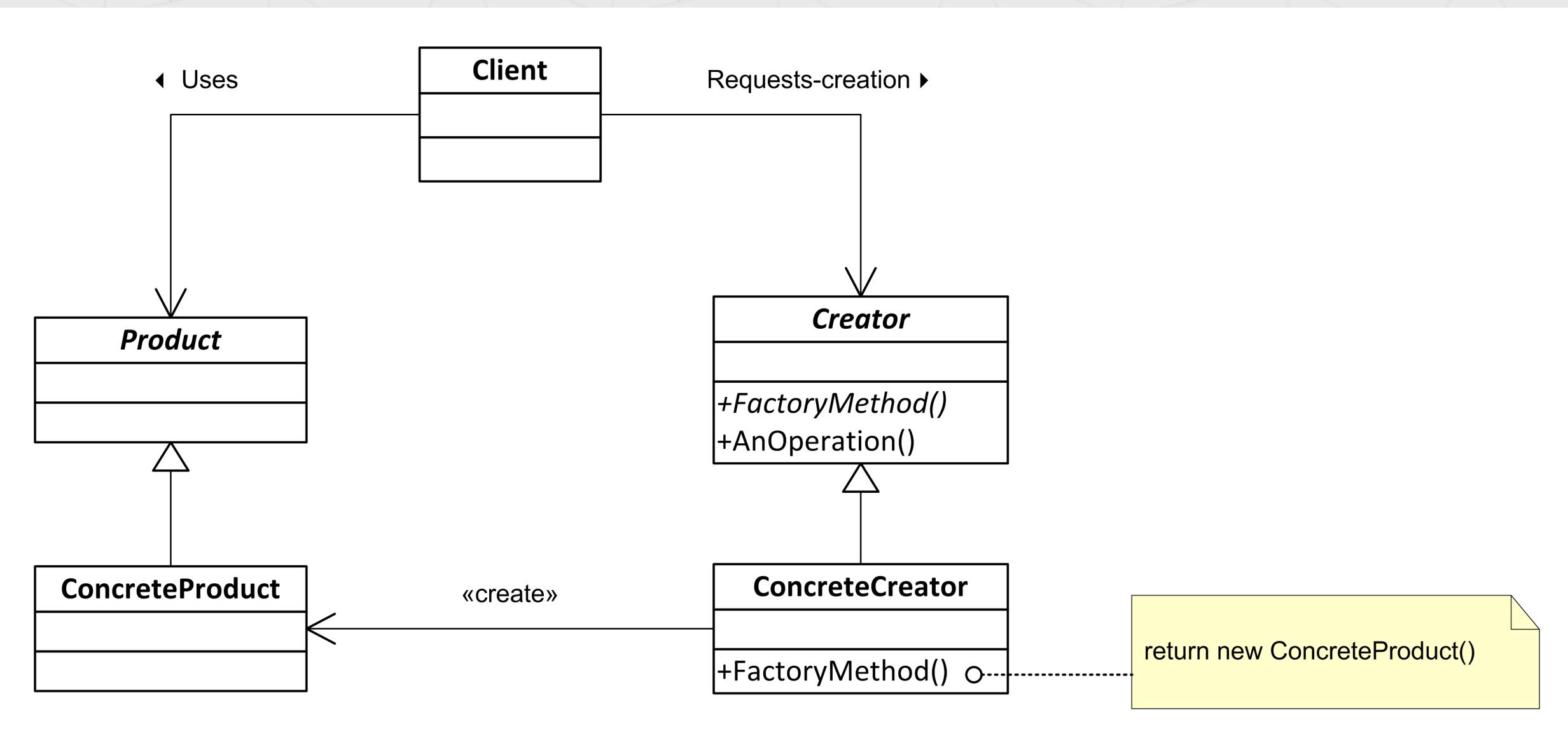
Factory Method - Problem

- Chcemy tworzyć instancje konkretnych klas za pomocą interfejsu
- Dana klasa nie może przewidzieć typu obiektu, który ma być utworzony

Factory Method - Problem

- Chcemy tworzyć instancje konkretnych klas za pomocą interfejsu
- Dana klasa nie może przewidzieć typu obiektu, który ma być utworzony
- Informacja o typie obiektu, który ma być utworzony jest dostępna dopiero w run-time'ie

Factory Method - Struktura



Factory Method - Współpraca

- Obiekt klasy **Creator** deleguje do swoich klas pochodnych odpowiedzialność za takie zdefiniowanie metody wytwórczej, by tworzyła egzemplarz odpowiedniej klasy **ConcreteProduct** (podklasy klasy abstrakcyjnej **Product**)

Factory Method - Konsekwencje

Factory Method - Konsekwencje

- Eliminuje potrzebę wstawiania klas konkretnych w kod aplikacji.

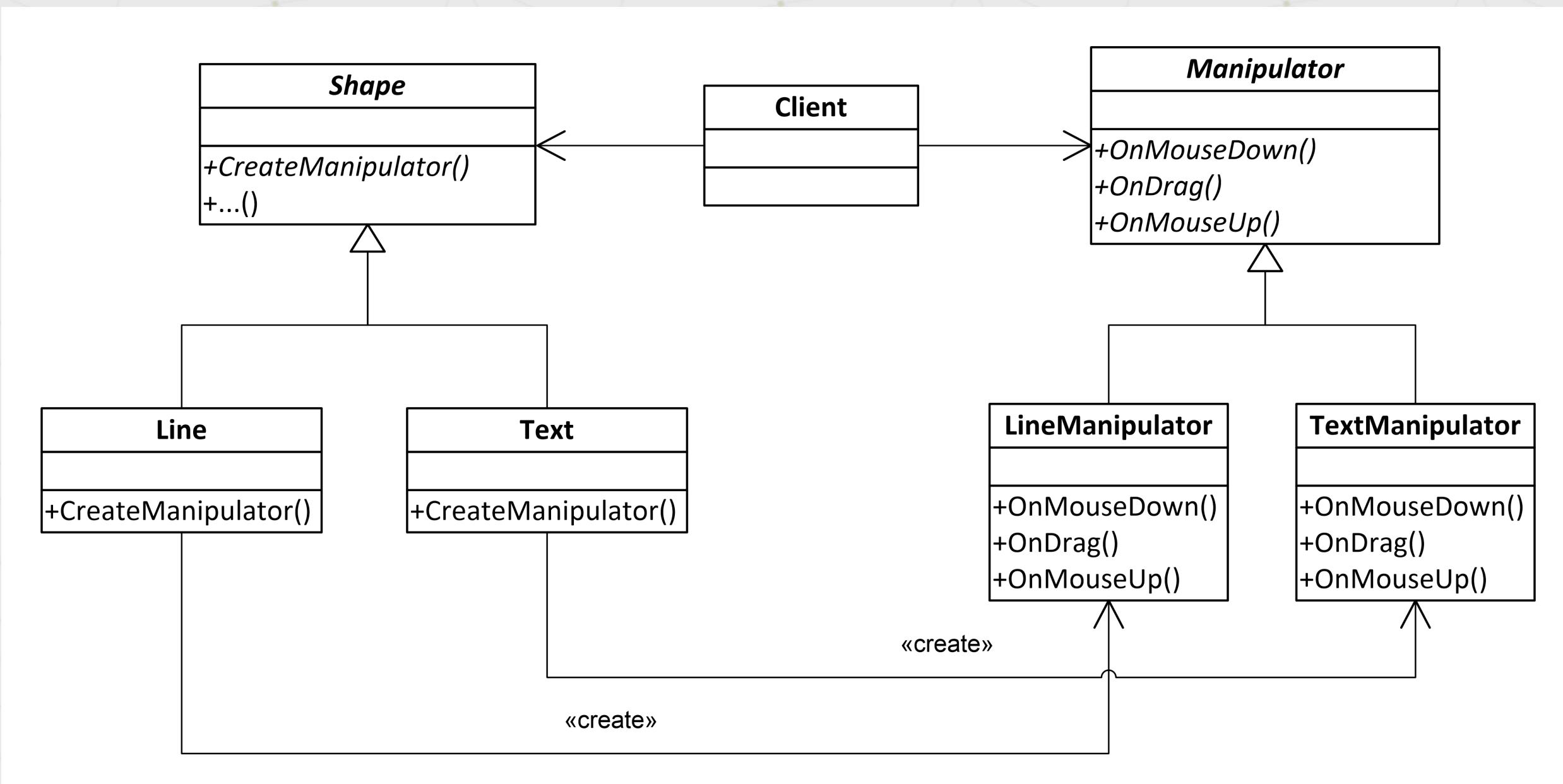
Factory Method - Konsekwencje

- Eliminuje potrzebę wstawiania klas konkretnych w kod aplikacji.
- Tworzenie obiektów wewnątrz klasy za pomocą metody wytwórczej jest bardziej elastyczne niż tworzenie ich bezpośrednio.

Factory Method - Konsekwencje

- Eliminuje potrzebę wstawiania klas konkretnych w kod aplikacji.
- Tworzenie obiektów wewnątrz klasy za pomocą metody wytwórczej jest bardziej elastyczne niż tworzenie ich bezpośrednio.
- Promuje luźne powiązania między obiektami, ponieważ redukuje zależność kodu aplikacji od konkretnych klas.

Factory Method - równoległe hierarchie klas



Factory Method - równoległe hierarchie klas

Factory Method - równoległe hierarchie klas

- Równoległe hierarchie klas powstają wtedy, gdy klasa przekazuje niektóre ze swych zobowiązań odrębnej klasie

Factory Method - równoległe hierarchie klas

- Równoległe hierarchie klas powstają wtedy, gdy klasa przekazuje niektóre ze swych zobowiązań odrębnej klasie
- Wzorzec Factory Method umożliwia łączenie równoległych hierarchii klas

Factory Method - Implementacja

Factory Method - Implementacja

- Implementacja klasy fabryki jako:
 - klasa abstrakcyjna (interfejs), która nie dostarcza implementacji dla metod, które deklaruje
 - klasa konkretna, która dostarcza domyślną implementację metody wytwarzającej

Factory Method - Implementacja

- Implementacja klasy fabryki jako:
 - klasa abstrakcyjna (interfejs), która nie dostarcza implementacji dla metod, które deklaruje
 - klasa konkretna, która dostarcza domyślną implementację metody wytwarzającej
- W prostym przypadku interfejs fabryki można zastąpić typem

```
01 using LoggerCreator = std::function<std::unique_ptr<Logger>()>;
```

Sparametryzowane metody wytwarzanie

- Metoda wytwarzająca przyjmuje parametr, który identyfikuje typ tworzonego obiektu
- W rezultacie pojedyncza instancja fabryki może tworzyć obiekty różnych typów

Sparametryzowane metody wytwórcze

```
01 using LoggerCreator = function<unique_ptr<Logger>()>;
02
03 class LoggerFactory
04 {
05     unordered_map<string, LoggerCreator> creators_;
06 public:
07     unique_ptr<Logger> create(const string& id) const
08     {
09         auto& creator = creators_.at(id);
10         return creator();
11     }
12
13     void register_creator(const string& id, const LoggerCreator& creator);
14 };
```

Sparametryzowane metody wytwórcze

```
01 using LoggerCreator = function<unique_ptr<Logger>()>;
02
03 class LoggerFactory
04 {
05     unordered_map<string, LoggerCreator> creators_;
06 public:
07     unique_ptr<Logger> create(const string& id) const
08     {
09         auto& creator = creators_.at(id);
10         return creator();
11     }
12
13     void register_creator(const string& id, const LoggerCreator& creator);
14 };
```

Sparametryzowane metody wytwórcze

```
01 using LoggerCreator = function<unique_ptr<Logger>()>;
02
03 class LoggerFactory
04 {
05     unordered_map<string, LoggerCreator> creators_;
06 public:
07     unique_ptr<Logger> create(const string& id) const
08     {
09         auto& creator = creators_.at(id);
10         return creator();
11     }
12
13     void register_creator(const string& id, const LoggerCreator& creator);
14 };
```

Sparametryzowane metody wytwórcze

```
01 using LoggerCreator = function<unique_ptr<Logger>()>;
02
03 class LoggerFactory
04 {
05     unordered_map<string, LoggerCreator> creators_;
06 public:
07     unique_ptr<Logger> create(const string& id) const
08     {
09         auto& creator = creators_.at(id);
10         return creator();
11     }
12
13     void register_creator(const string& id, const LoggerCreator& creator);
14 };
```

Sparametryzowane metody wytwórcze

```
01 using LoggerCreator = function<unique_ptr<Logger>()>;
02
03 class LoggerFactory
04 {
05     unordered_map<string, LoggerCreator> creators_;
06 public:
07     unique_ptr<Logger> create(const string& id) const
08     {
09         auto& creator = creators_.at(id);
10         return creator();
11     }
12
13     void register_creator(const string& id, const LoggerCreator& creator);
14 };
```

■ Użycie fabryki:

```
01 LoggerFactory logger_factory;
02
03 logger_factory.register_creator("FileLogger",
04     []{ return make_unique<FileLogger>("log.dat"); });
05
06 //...
07
08 auto logger = logger_factory.create("FileLogger");
09 logger.log("Message");
```

Factory Method - Pokrewne wzorce

- Iterator
- Abstract Factory – fabrykę abstrakcyjną często implementuje się za pomocą metod wytwórczych

Factory Method - Podsumowanie

- Umożliwia inicjowanie przez jeden obiekt procesu tworzenia innego obiektu w sytuacji, gdy nie jest znana klasa tworzonego obiektu
- Kod klienta jest nakierowany na interfejsy
- Umożliwia łączenie równoległych hierarchii klas

Abstract Factory

Abstract Factory

■ Przeznaczenie

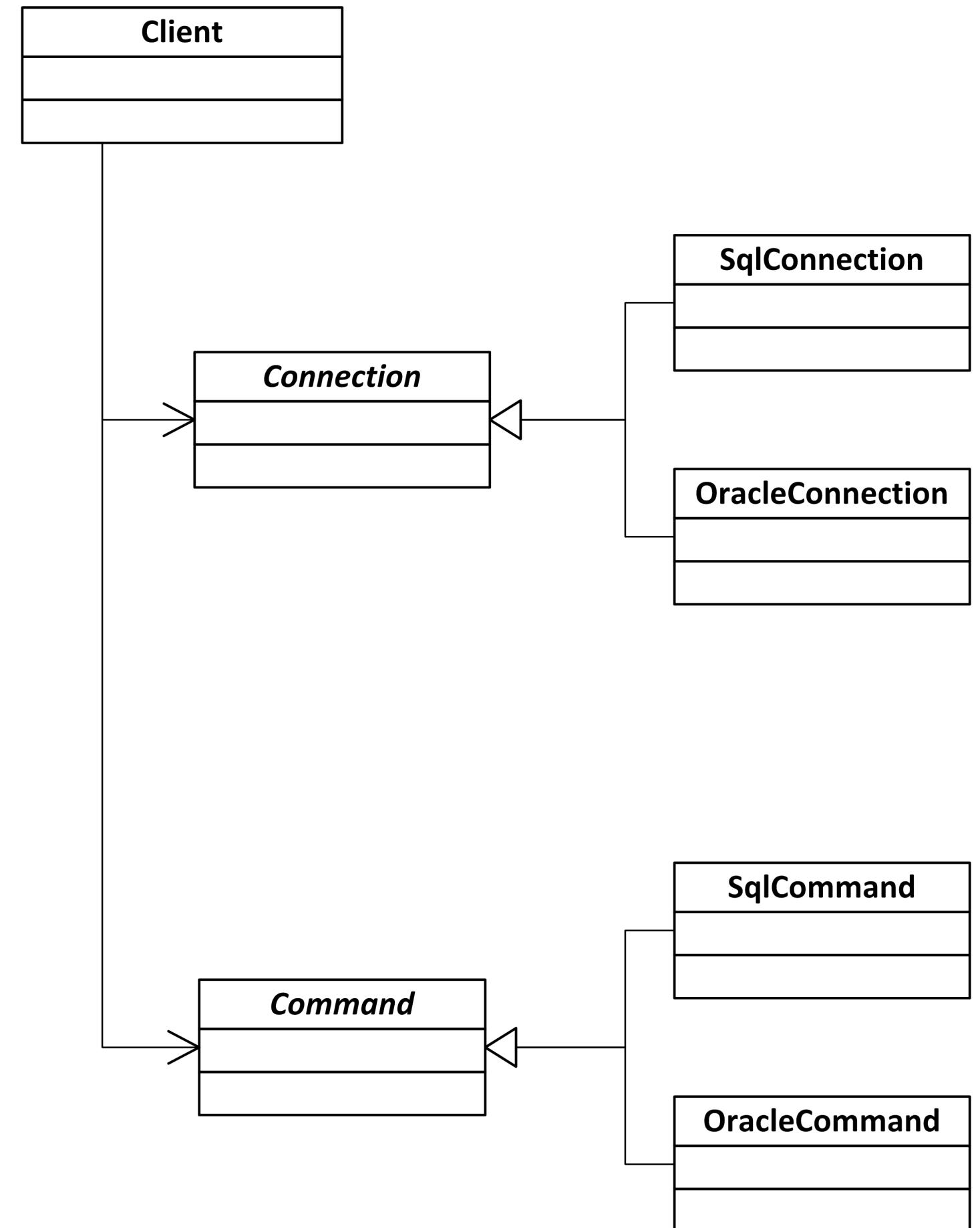
- dostarcza interfejs do tworzenia całych rodzin spokrewnionych lub zależnych od siebie obiektów bez konieczności określania ich klas rzeczywistych

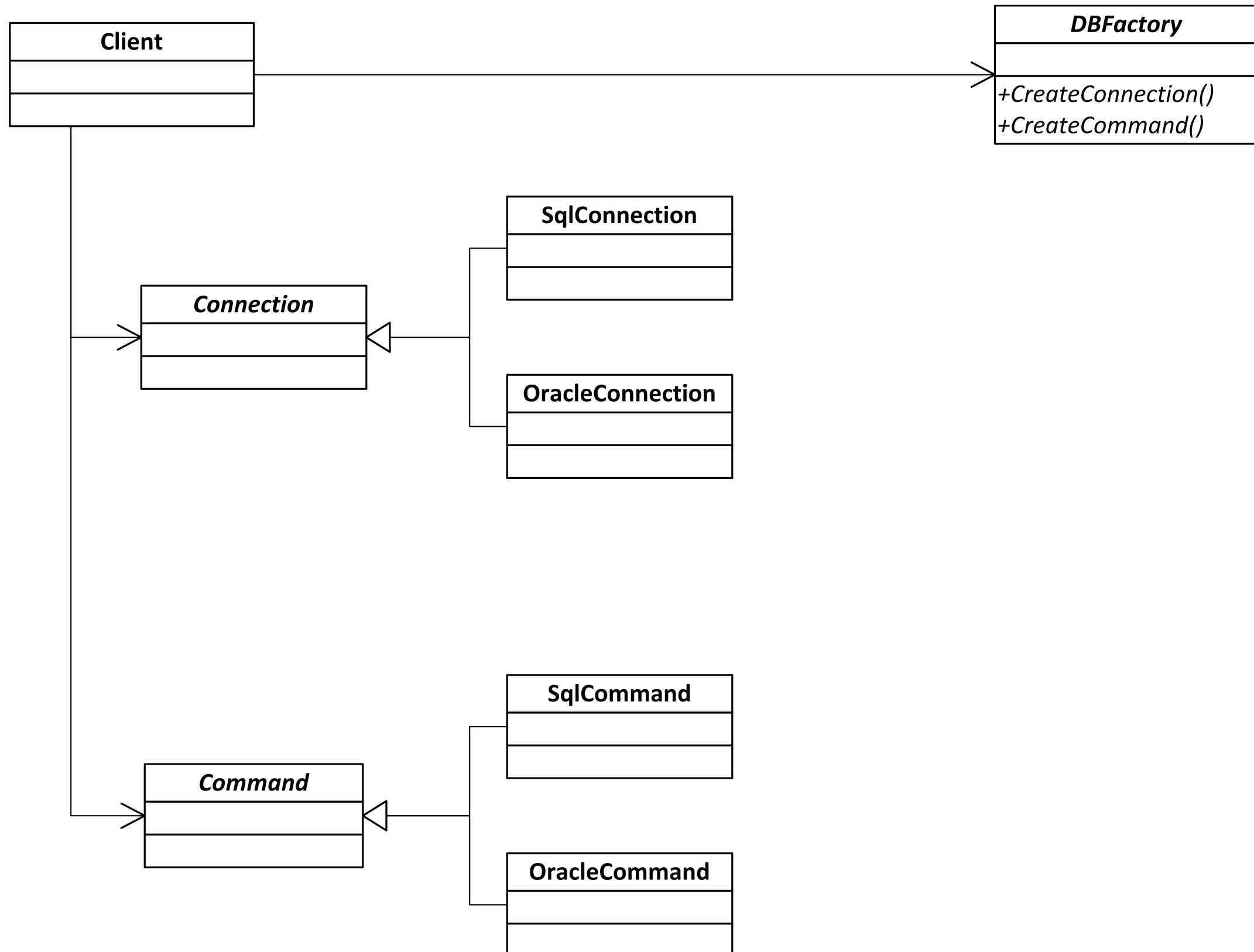
Abstract Factory - Scenariusz

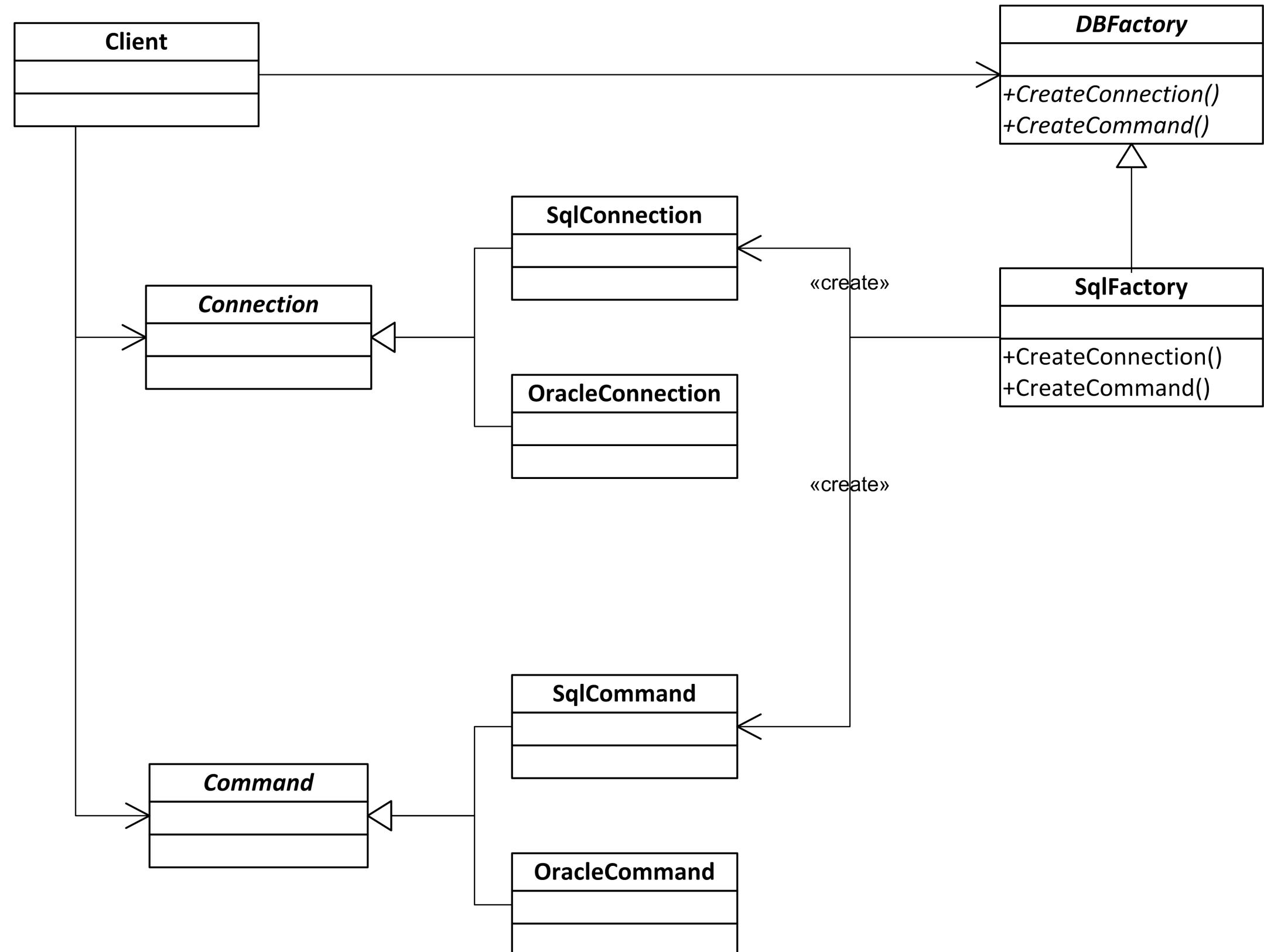
- Chcemy napisać aplikację współpracującą z wieloma RDBMSami (np. Oracle, SQL Server, itp.)
- Definiujemy podstawowe klasy abstrakcyjne, które współpracują ze sobą:
 - Connection – obiekt kontrolujący połączenie z bazą danych
 - Command – obiekt reprezentujący polecenie SQL
 - Transaction – obiekt reprezentujący transakcję

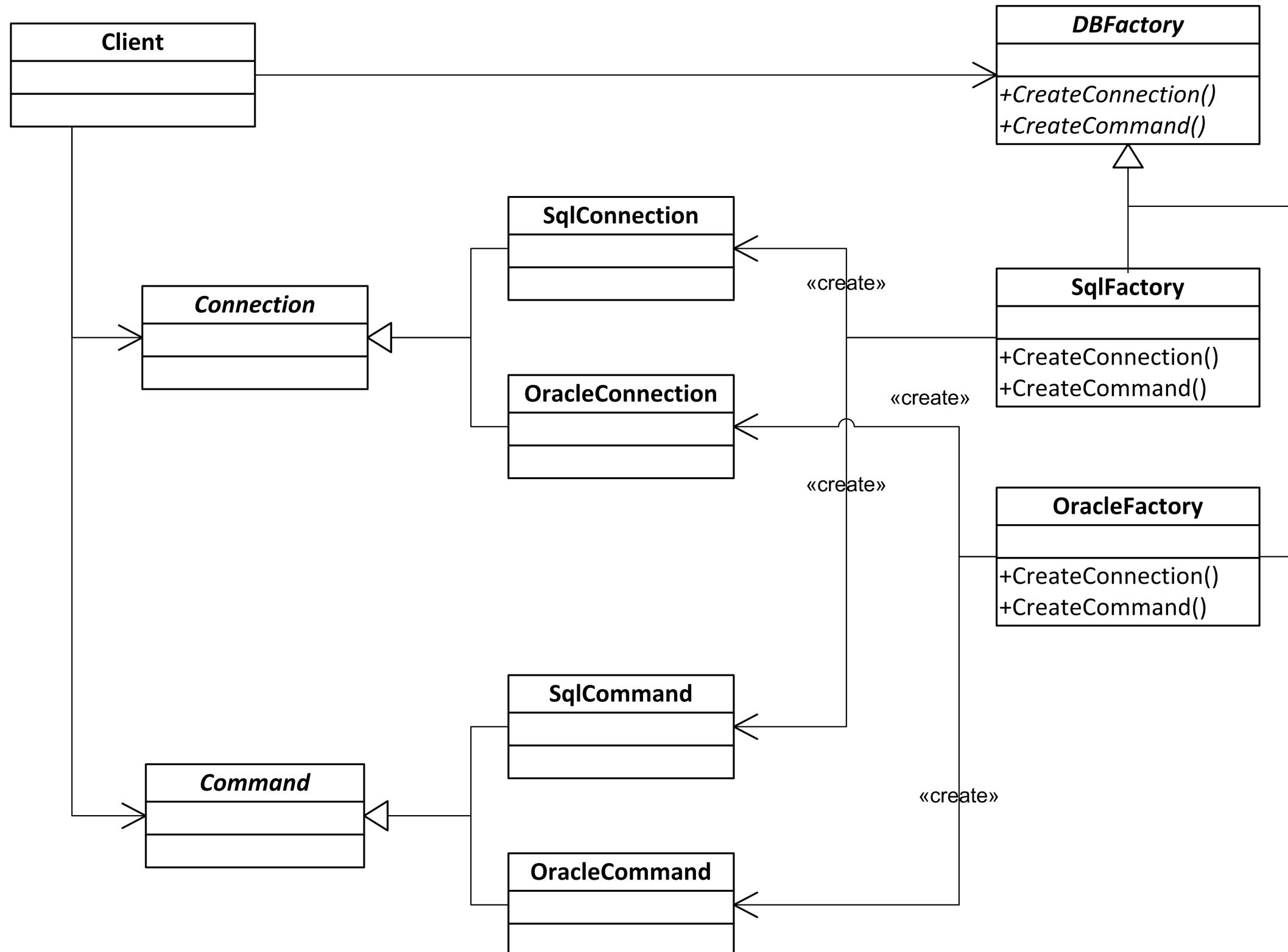
Abstract Factory - Scenariusz

- Chcemy uniknąć sztywnego tworzenia konkretnych klas w aplikacji i jednocześnie chcemy zachować spójność w ramach rodziny obiektów









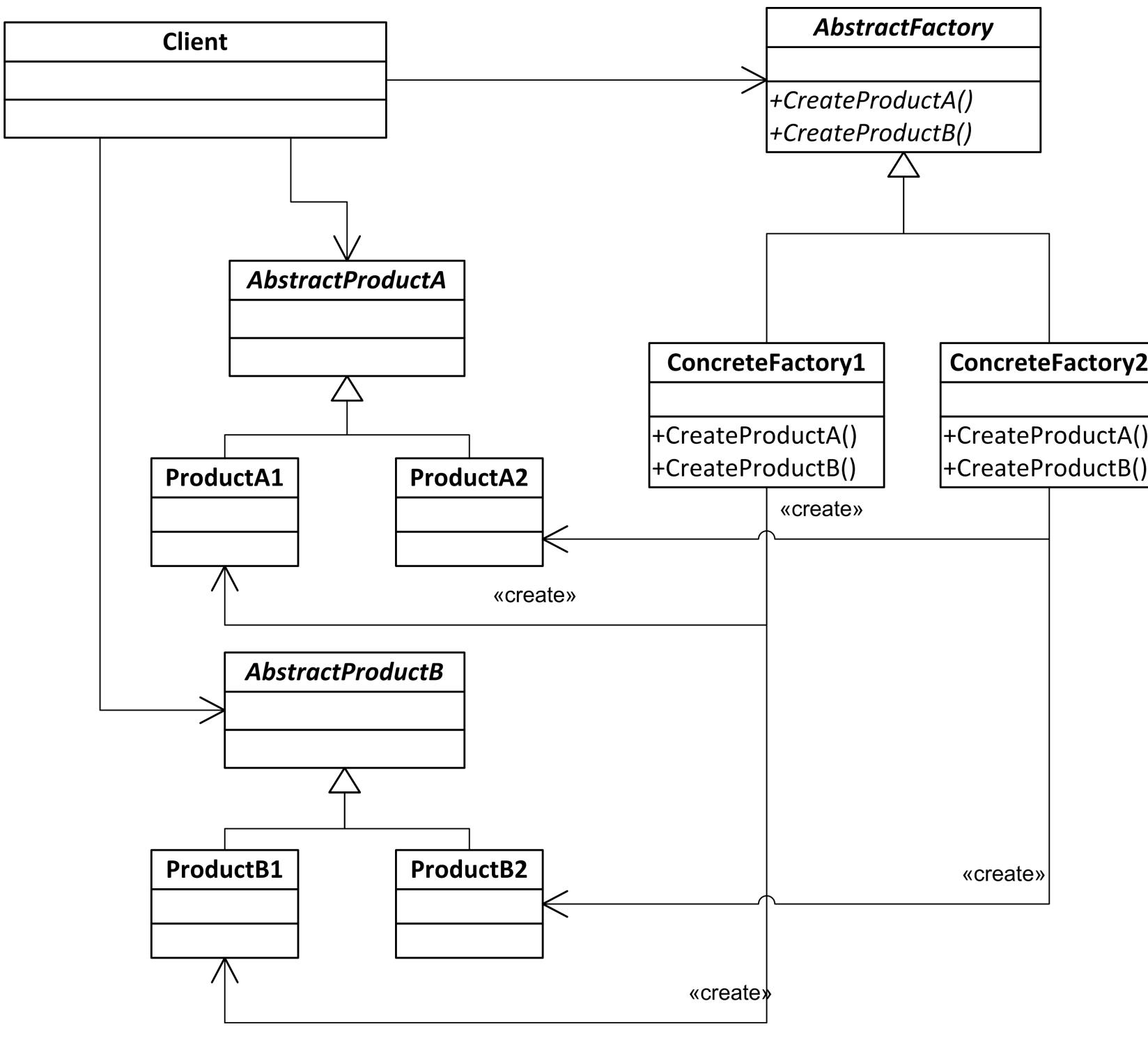
Abstract Factory - Kontekst

- W systemie istnieją rodziny powiązanych ze sobą obiektów-produktów, zaprojektowane tak, by obiekty były używane razem i ograniczenie to powinno być zachowane
- System powinien być niezależny od tego, jak jego produkty są tworzone, składowane i reprezentowane

Abstract Factory - Problem

- Chcemy umożliwić łatwą konfigurację systemu przy użyciu jednej z wielu rodzin produktów
- Kod powinien być zależny od interfejsów lub klas abstrakcyjnych
- Chcemy dostarczyć bibliotekę klas produktów, ujawniając tylko ich interfejsy, a nie implementacje

Abstract Factory - Struktura



Abstract Factory - Implementacja

- Najczęstsza implementacja

```
01 class AbstractFactory
02 {
03 public:
04     virtual ~AbstractFactory() = default;
05     virtual std::unique_ptr<ProductA> create_product_a() = 0;
06     virtual std::unique_ptr<ProductB> create_product_b() = 0;
07     virtual std::unique_ptr<ProductC> create_product_c() = 0;
08 };
```

Abstract Factory - Implementacja

- Najczęstsza implementacja

```
01 class AbstractFactory
02 {
03 public:
04     virtual ~AbstractFactory() = default;
05     virtual std::unique_ptr<ProductA> create_product_a() = 0;
06     virtual std::unique_ptr<ProductB> create_product_b() = 0;
07     virtual std::unique_ptr<ProductC> create_product_c() = 0;
08 };
```

Abstract Factory - Konsekwencje

Abstract Factory - Konsekwencje

- Odseparowanie klas konkretnych

Abstract Factory - Konsekwencje

- Odseparowanie klas konkretnych
- Łatwiejsza wymiana rodzin produktów
 - najczęściej klasa konkretnej fabryki pojawia się w aplikacji tylko raz
 - umożliwia to łatwą zmianę fabryki konkretnej używanej przez aplikację, a tym samym wymianę rodziny produktów

Abstract Factory - Konsekwencje

- Odseparowanie klas konkretnych
- Łatwiejsza wymiana rodzin produktów
 - najczęściej klasa konkretnej fabryki pojawia się w aplikacji tylko raz
 - umożliwia to łatwą zmianę fabryki konkretnej używanej przez aplikację, a tym samym wymianę rodziny produktów
- Spójność produktów
 - współpraca produktów wymaga, by aplikacja używała za jednym razem obiektów tylko z danej rodziny

Abstract Factory - Konsekwencje

- Odseparowanie klas konkretnych
- Łatwiejsza wymiana rodzin produktów
 - najczęściej klasa konkretnej fabryki pojawia się w aplikacji tylko raz
 - umożliwia to łatwą zmianę fabryki konkretnej używanej przez aplikację, a tym samym wymianę rodziny produktów
- Spójność produktów
 - współpraca produktów wymaga, by aplikacja używała za jednym razem obiektów tylko z danej rodziny
- Utrudnione dołączenie nowych produktów

Abstract Factory - Pokrewne wzorce

- Factory Method
 - klasy AbstractFactory są definiowane z wykorzystaniem metod wytwarzających
- Singleton
 - konkretna instancja fabryki może być implementowana jako singleton

Abstract Factory - Podsumowanie

Abstract Factory - Podsumowanie

- Zapewnia interfejs umożliwiający tworzenie rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych

Abstract Factory - Podsumowanie

- Zapewnia interfejs umożliwiający tworzenie rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych
- Rodziny produktów mogą być łatwo wymieniane bez ingerencji w kod klienta

Abstract Factory - Podsumowanie

- Zapewnia interfejs umożliwiający tworzenie rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych
- Rodziny produktów mogą być łatwo wymieniane bez ingerencji w kod klienta
- Spełnia regułę odwracania zależności (DIP – Dependency Inversion Principle)

Abstract Factory - Podsumowanie

- Zapewnia interfejs umożliwiający tworzenie rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych
- Rodziny produktów mogą być łatwo wymieniane bez ingerencji w kod klienta
- Spełnia regułę odwracania zależności (DIP – Dependency Inversion Principle)
- Najczęstsza implementacja wzorca: interfejs fabryki abstrakcyjnej zdefiniowany jako kolekcja metod wytwórczych

Fabryki - podsumowanie

Fabryki - podsumowanie

- Uabstrakcyjnają proces tworzenia obiektów

Fabryki - podsumowanie

- Uabstrakcyjnają proces tworzenia obiektów
- Potężne techniki programistyczne pozwalające na tworzenie kodu, który będzie uzależniony od abstrakcji

Fabryki - podsumowanie

- Uabstrakcyjnają proces tworzenia obiektów
- Potężne techniki programistyczne pozwalające na tworzenie kodu, który będzie uzależniony od abstrakcji
- Promują tworzenie luźnych zależności poprzez redukcję zależności kodu aplikacji od implementacji klas rzeczywistych

Singleton

Singleton

■ Przeznaczenie

- Gwarantuje, że dana klasa będzie miała tylko i wyłącznie jedną instancję obiektu i zapewnia globalny punkt dostępu do tej instancji
- Wszystkie obiekty korzystające z danej klasy używają tego samego egzemplarza

Singleton - Kontekst / Problem

Singleton - Kontekst / Problem

■ Kontekst

- w systemie istnieją obiekty, które z różnych powodów powinny występować tylko w jednym egzemplarzu

Singleton - Kontekst / Problem

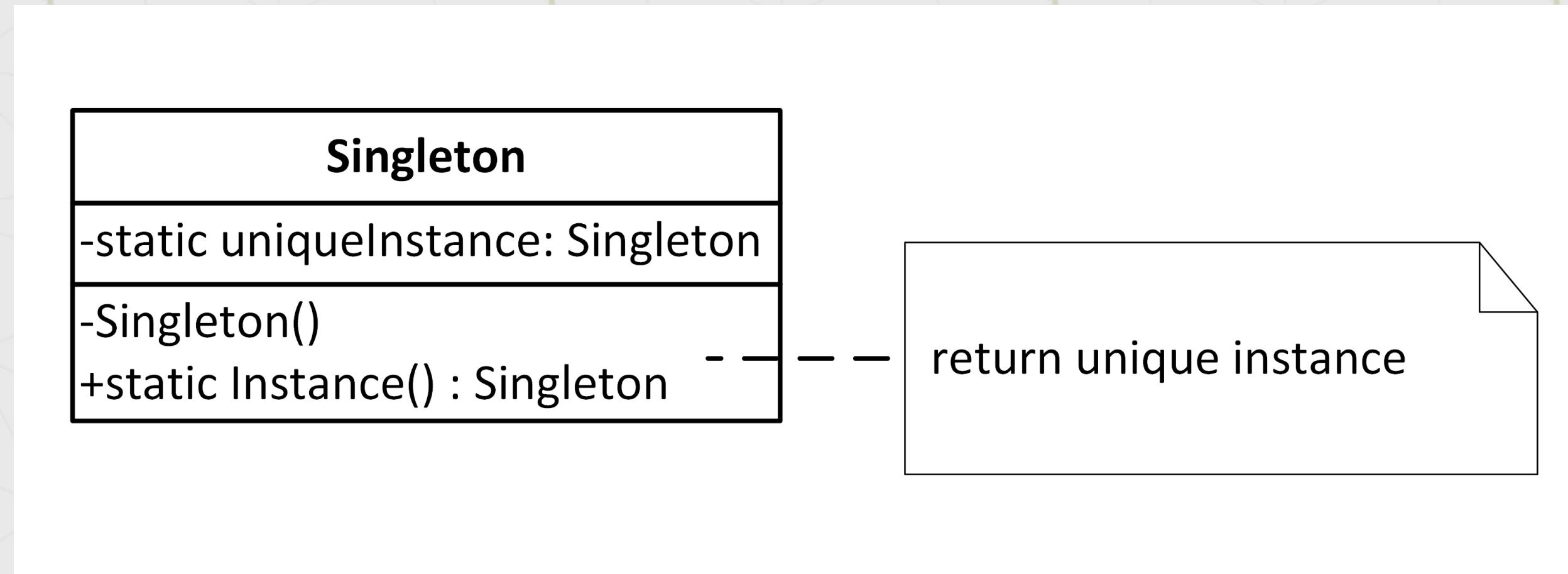
■ Kontekst

- w systemie istnieją obiekty, które z różnych powodów powinny występować tylko w jednym egzemplarzu

■ Problem

- chcemy utworzyć klasę, która może mieć tylko jeden egzemplarz instancji, dostępny dla klientów

Singleton - Struktura



Singleton - Implementacja

Singleton - Implementacja

- Zagwarantowanie unikatowego egzemplarza
 - prywatny konstruktor oraz usunięte z interfejsu operacje kopiowania
 - prywatna statyczna właściwość inicjalizowana jedyną instancją klasy
 - publiczna statyczna metoda oferująca dostęp do jedynego egzemplarza obiektu klasy Singleton

Singleton - Implementacja

- Zagwarantowanie unikatowego egzemplarza
 - prywatny konstruktor oraz usunięte z interfejsu operacje kopiowania
 - prywatna statyczna właściwość inicjalizowana jedyną instancją klasy
 - publiczna statyczna metoda oferująca dostęp do jedynego egzemplarza obiektu klasy Singleton
- Leniwa inicjalizacja instancji klasy

Singleton - Implementacja

- Zagwarantowanie unikatowego egzemplarza
 - prywatny konstruktor oraz usunięte z interfejsu operacje kopiowania
 - prywatna statyczna właściwość inicjalizowana jedyną instancją klasy
 - publiczna statyczna metoda oferująca dostęp do jedynego egzemplarza obiektu klasy Singleton
- Leniwa inicjalizacja instancji klasy
- Singleton wielowątkowy – obsługa jednoczesnych wywołań metody instance

Singleton - Implementacja C++

```
01 class Singleton
02 {
03     Singleton() = default;
04     ~Singleton() = default;
05 public:
06     Singleton(const Singleton&) = delete;
07     Singleton& operator=(const Singleton&) = delete;
08
09     static Singleton& instance()
10    {
11        static Singleton unique_instance;
12        return unique_instance;
13    }
14
15    void do_something();
16};
```

Singleton - Implementacja C++

```
01 class Singleton
02 {
03     Singleton() = default;
04     ~Singleton() = default;
05 public:
06     Singleton(const Singleton&) = delete;
07     Singleton& operator=(const Singleton&) = delete;
08
09     static Singleton& instance()
10    {
11        static Singleton unique_instance;
12        return unique_instance;
13    }
14
15    void do_something();
16};
```

Singleton - Implementacja C++

```
01 class Singleton
02 {
03     Singleton() = default;
04     ~Singleton() = default;
05 public:
06     Singleton(const Singleton&) = delete;
07     Singleton& operator=(const Singleton&) = delete;
08
09     static Singleton& instance()
10    {
11        static Singleton unique_instance;
12        return unique_instance;
13    }
14
15    void do_something();
16};
```

Singleton - Implementacja C++

```
01 class Singleton
02 {
03     Singleton() = default;
04     ~Singleton() = default;
05 public:
06     Singleton(const Singleton&) = delete;
07     Singleton& operator=(const Singleton&) = delete;
08
09     static Singleton& instance()
10    {
11        static Singleton unique_instance;
12        return unique_instance;
13    }
14
15    void do_something();
16};
```

Użycie singletona

```
01 Singleton::instance().do_something();
02
03 Singleton& single_object = Singleton::instance();
04 Single_object.do_something();
```

Generic Singleton

```
01 template <typename T>
02 class SingletonHolder
03 {
04     SingletonHolder() = default;
05 public:
06     SingletonHolder(const SingletonHolder&) = delete;
07     SingletonHolder& operator=(const SingletonHolder&) = delete;
08
09     static T& instance()
10    {
11        static T unique_instance;
12
13        return unique_instance;
14    }
15};
```

Użycie singletona w wersji generycznej

Dla klasy:

```
01 class Logger
02 {
03     public:
04         void log(const string&);
05         //... rest of impl
06 };
```

Utworzenie singletona i jego użycie:

```
01 using SingletonLogger = SingletonHolder<Logger>;
02
03 SingletonLogger::instance().log("msg1");
```

Singleton - Podsumowanie

- Singleton gwarantuje, że zostanie utworzony tylko jeden egzemplarz danej klasy
- Wszystkie obiekty korzystające z danej klasy używają tego samego egzemplarza

Prototype

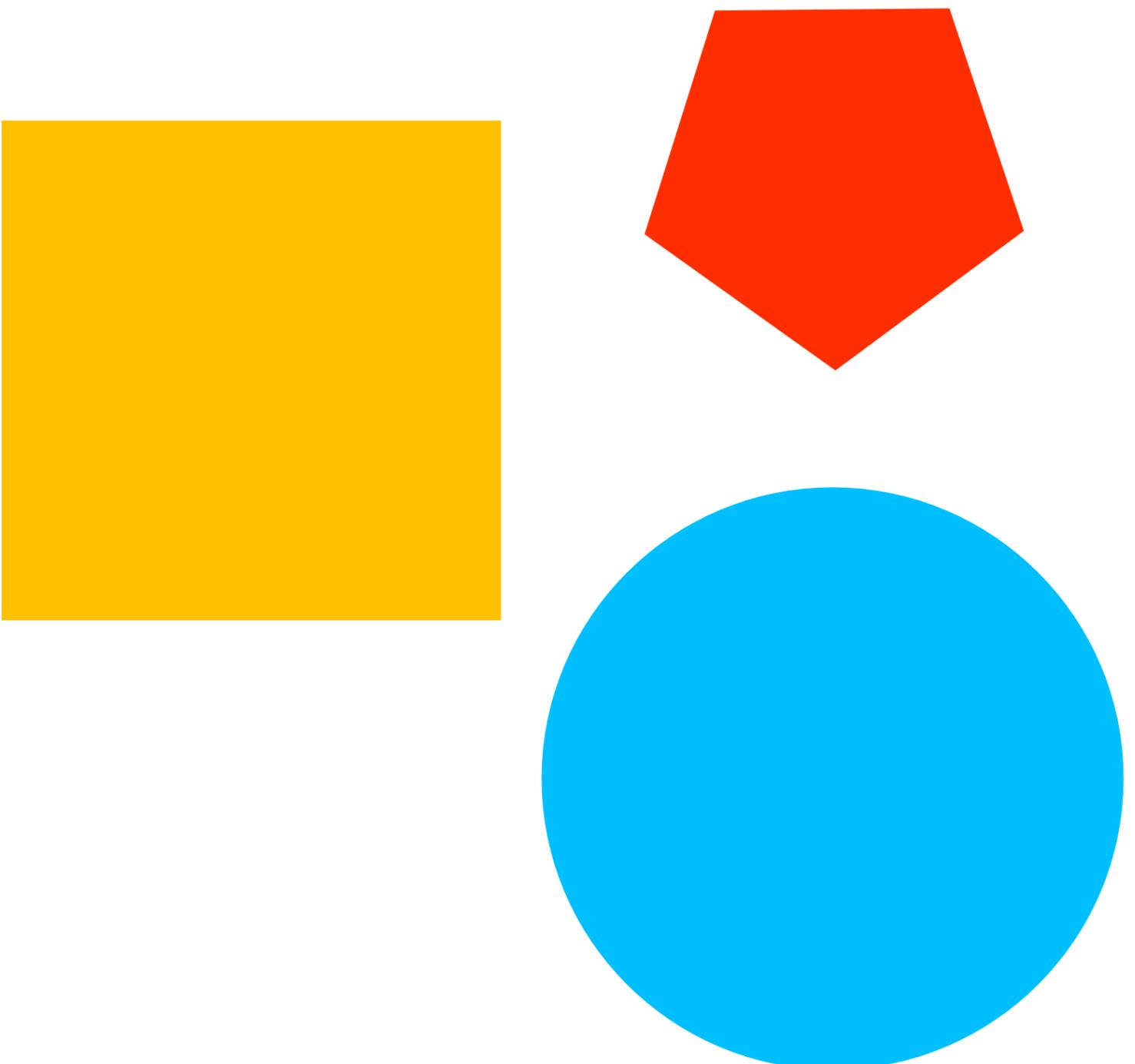
Prototype

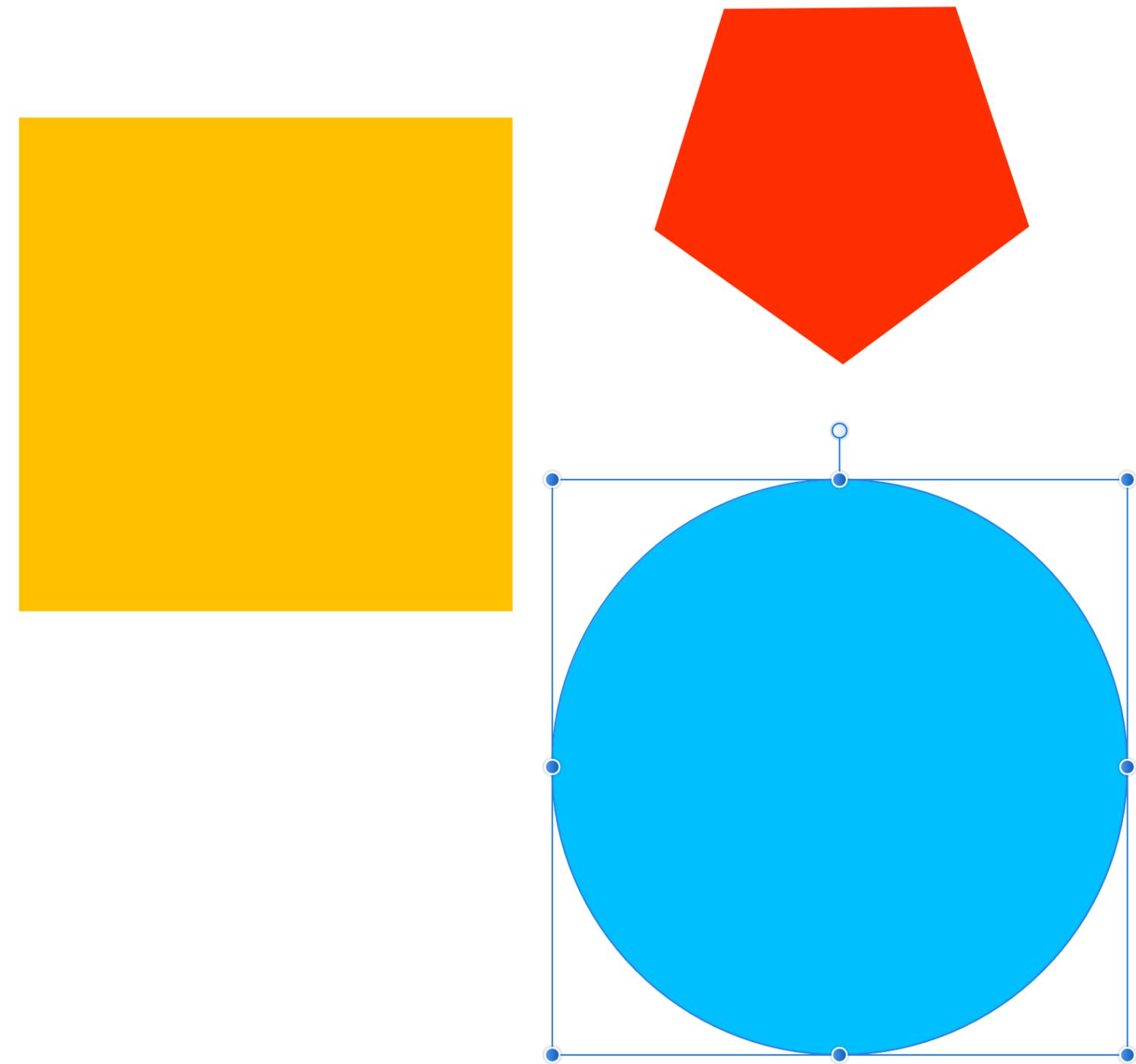
■ Przeznaczenie

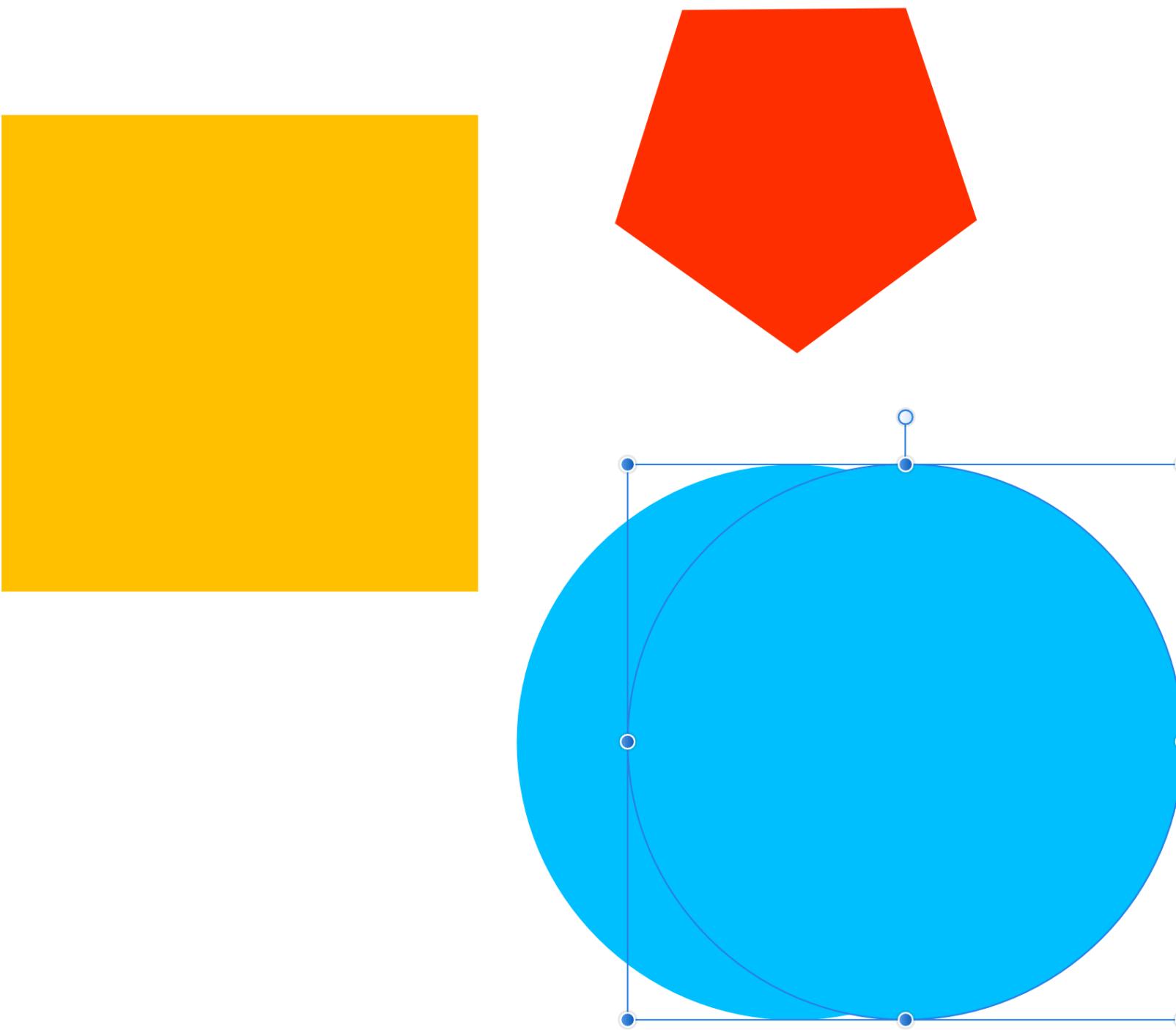
- określa rodzaj tworzonych obiektów, używając prototypowego egzemplarza
- tworzy nowe obiekty, kopując ten prototyp
- zapewnia klientowi możliwość generowania obiektów, których typ nie jest klientowi znany

Prototyp - Scenariusz

- Chcemy napisać aplikację graficzną umożliwiającą edycję rysunków
- Rysunki są reprezentowane w postaci agregatów typu Shape
- Klasa abstrakcyjna Shape definiuje podstawowe operacje wykonywane przez klienta - kod klienta zależy od tej abstrakcji
- Chcemy dodać możliwość kopiowania rysunków bez odwoływanego się do klas konkretnych reprezentujących jego elementy składowe



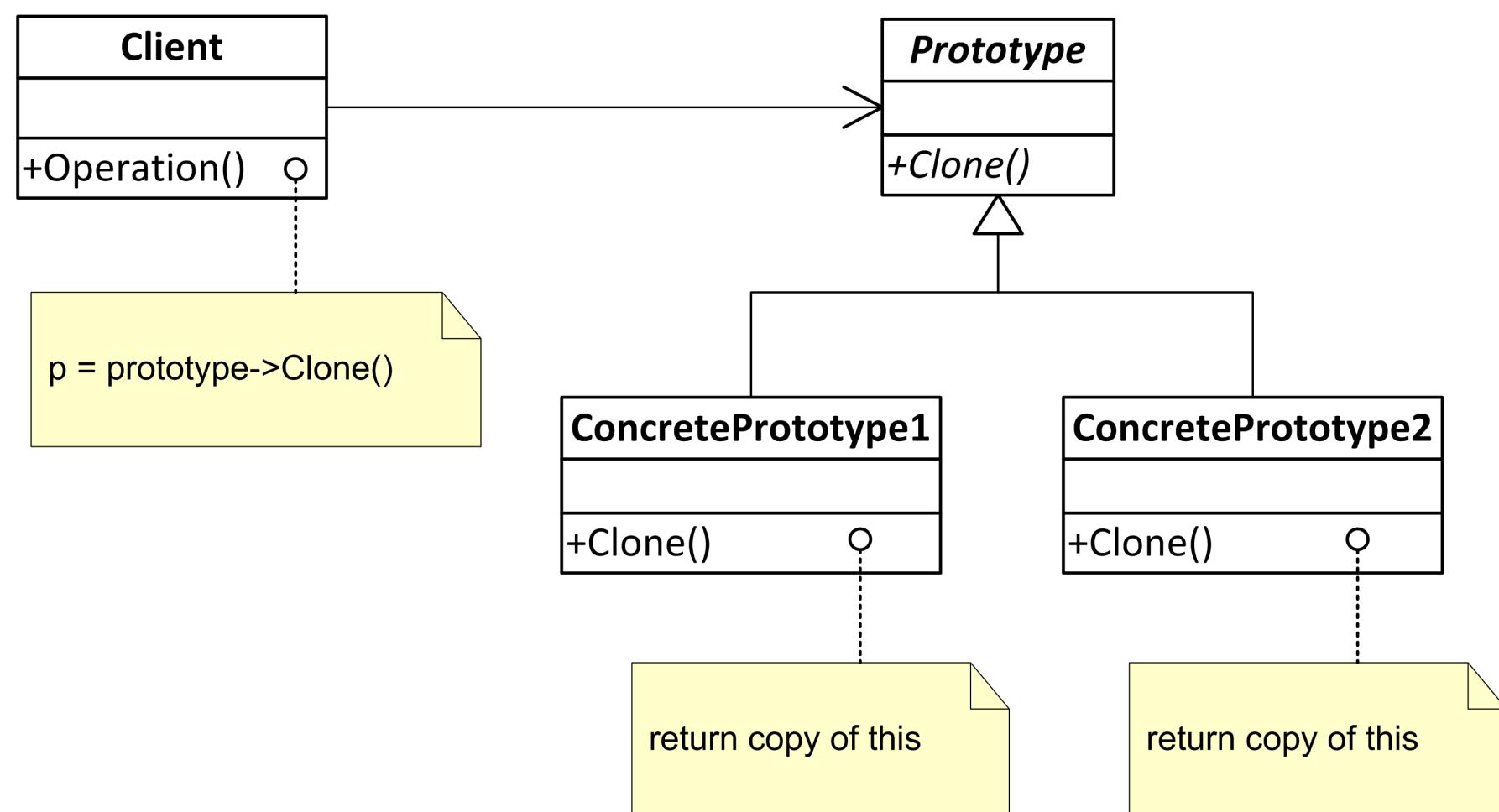




```
01 auto selected_shape = get_selected_shape(x, y);  
02  
03 std::unique_ptr<Shape> shape_copy;  
04 if (selected_shape)  
05 {  
06     shape_copy = ...;  
07 }
```

```
01 std::unique_ptr<Shape> shape_copy;
02
03 if (auto selected_shape = get_selected_shape(x, y); selected_shape)
04 {
05     shape_copy = selected_shape->clone();
06 }
```

Prototype - Struktura



Prototyp - implementacja C++

Definiujemy abstrakcyjną metodę `clone` w klasie bazowej:

```
01 class Shape {
02 public:
03     virtual std::unique_ptr<Shape> clone() const = 0;
04     virtual ~Shape() = default;
05
06     // rest of methods
07 };
```

Każda konkretna klasa pochodna **musi** zaimplementować operację `clone`:

```
01 class Rectangle : public Shape {
02 public:
03     std::unique_ptr<Shape> clone() const override
04     {
05         return std::make_unique<Rectangle>(*this);
06     }
07 };
```

CRTP

Aby uniknąć duplikacji kodu przy implementacji klas pochodnych możemy zastosować idiom CRTP:

```
01 template <typename ShapeType>
02 class CloneableShape : public Shape
03 {
04 public:
05     std::unique_ptr<Shape> clone() const override
06     {
07         return std::make_unique<ShapeType>(
08             static_cast<const ShapeType&>(*this));
09     }
10 };
```

Klasa pochodna może użyć `CloneableEngine` jako klasy bazowej:

```
01 class Rectangle : public CloneableShape<Diesel> {
02 public:
03     //...
04 };
```

CRTP

Aby uniknąć duplikacji kodu przy implementacji klas pochodnych możemy zastosować idiom CRTP:

```
01 template <typename ShapeType>
02 class CloneableShape : public Shape
03 {
04     public:
05         std::unique_ptr<Shape> clone() const override
06     {
07         return std::make_unique<ShapeType>(
08             static_cast<const ShapeType&>(*this));
09     }
10 };
```

Klasa pochodna może użyć **CloneableEngine** jako klasy bazowej:

```
01 class Rectangle : public CloneableShape<Diesel> {
02 public:
03     //...
04 };
```

CRTP

Aby uniknąć duplikacji kodu przy implementacji klas pochodnych możemy zastosować idiom CRTP:

```
01 template <typename ShapeType>
02 class CloneableShape : public Shape
03 {
04 public:
05     std::unique_ptr<Shape> clone() const override
06     {
07         return std::make_unique<ShapeType>(
08             static_cast<const ShapeType&>(*this));
09     }
10 };
```

Klasa pochodna może użyć **CloneableEngine** jako klasy bazowej:

```
01 class Rectangle : public CloneableShape<Diesel> {
02 public:
03     //...
04 };
```

CRTP

Aby uniknąć duplikacji kodu przy implementacji klas pochodnych możemy zastosować idiom CRTP:

```
01 template <typename ShapeType>
02 class CloneableShape : public Shape
03 {
04 public:
05     std::unique_ptr<Shape> clone() const override
06     {
07         return std::make_unique<ShapeType>(
08             static_cast<const ShapeType&>(*this));
09     }
10 };
```

Klasa pochodna może użyć **CloneableEngine** jako klasy bazowej:

```
01 class Rectangle : public CloneableShape<Diesel> {
02 public:
03     //...
04 };
```

Prototype - konsekwencje

- Dynamiczne dodawanie i usuwanie produktów w czasie wykonywania programu
 - ułatwia włączanie do systemu nowych produktów konkretnych przez rejestrowanie prototypowych egzemplarzy u klienta
 - bardziej elastyczne rozwiązanie niż w przypadku fabryk
 - zredukowana liczba podklas
- Umożliwia specyfikowanie nowych prototypowych obiektów przez urozmaicanie struktury
 - złożone struktury definiowane w trakcie działania programu mogą być również klonowane

Prototype - Pokrewne wzorce

- Abstract Factory
- Composite, Decorator, Command
 - wzorce te są często używane wraz ze wzorcem Prototype

Prototype - Podsumowanie

Prototype - Podsumowanie

- Wzorzec umożliwia tworzenie nowych obiektów poprzez kopiowanie prototypowego egzemplarza

Prototype - Podsumowanie

- Wzorzec umożliwia tworzenie nowych obiektów poprzez kopiowanie prototypowego egzemplarza
- Klasy, których egzemplarze są klonowane, mogą być specyfikowane w trakcie wykonania programu

Prototype - Podsumowanie

- Wzorzec umożliwia tworzenie nowych obiektów poprzez kopiowanie prototypowego egzemplarza
- Klasy, których egzemplarze są klonowane, mogą być specyfikowane w trakcie wykonania programu
- Pozwala uprościć hierarchię klas fabryk, która jest porównywalna z hierarchią klas produktów

Builder

Builder - Przeznaczenie

Builder - Przeznaczenie

- Oddziela konstrukcję złożonych obiektów od ich reprezentacji, umożliwiając tym samym powstawanie w jednym procesie konstrukcyjnym różnych reprezentacji

Builder - Przeznaczenie

- Oddziela konstrukcję złożonych obiektów od ich reprezentacji, umożliwiając tym samym powstawanie w jednym procesie konstrukcyjnym różnych reprezentacji
- Definiuje etapy tworzenia obiektu-produktu
 - etapy te są konfigurowalne z zewnątrz (to odróżnia go od fabryk obiektów)

Builder - Kontekst

Builder - Kontekst

- Algorytm konstrukcji obiektu jest wieloetapowy

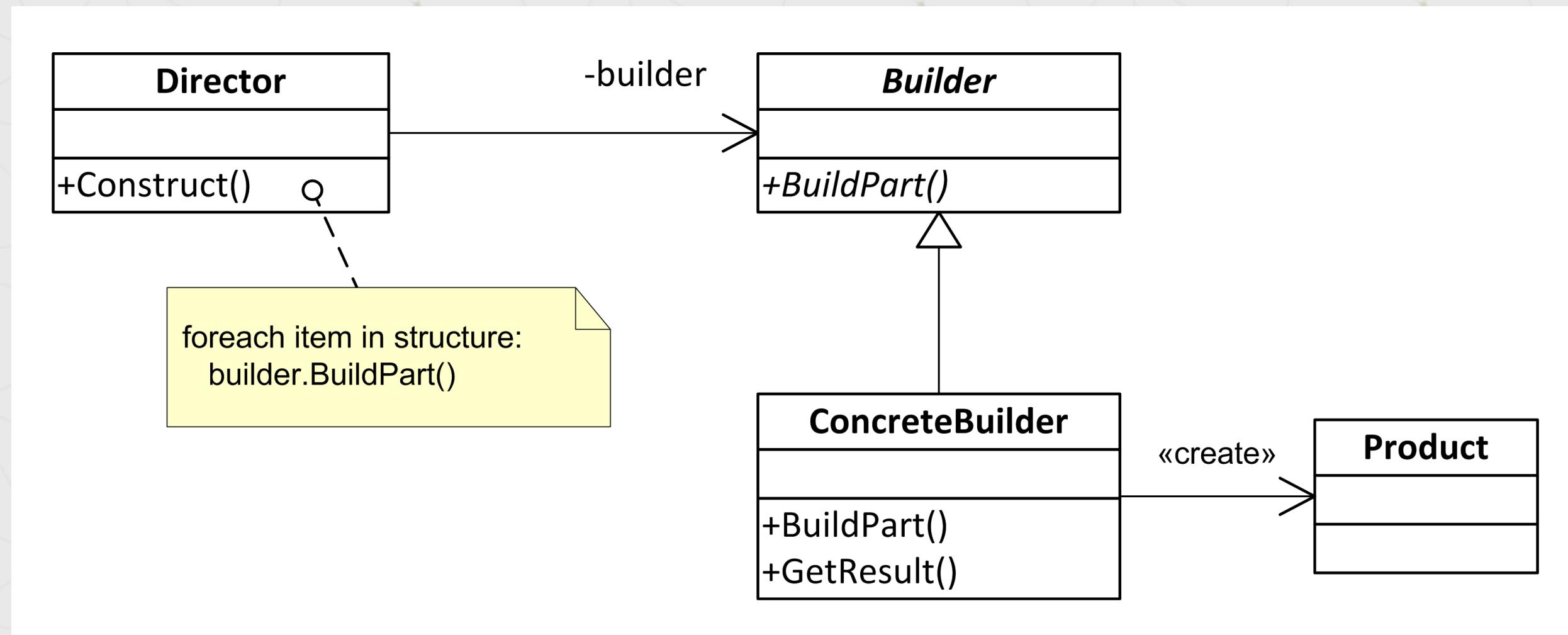
Builder - Kontekst

- Algorytm konstrukcji obiektu jest wieloetapowy
- Proces konstrukcji złożonego obiektu prowadzi do utworzenia różnych reprezentacji obiektu

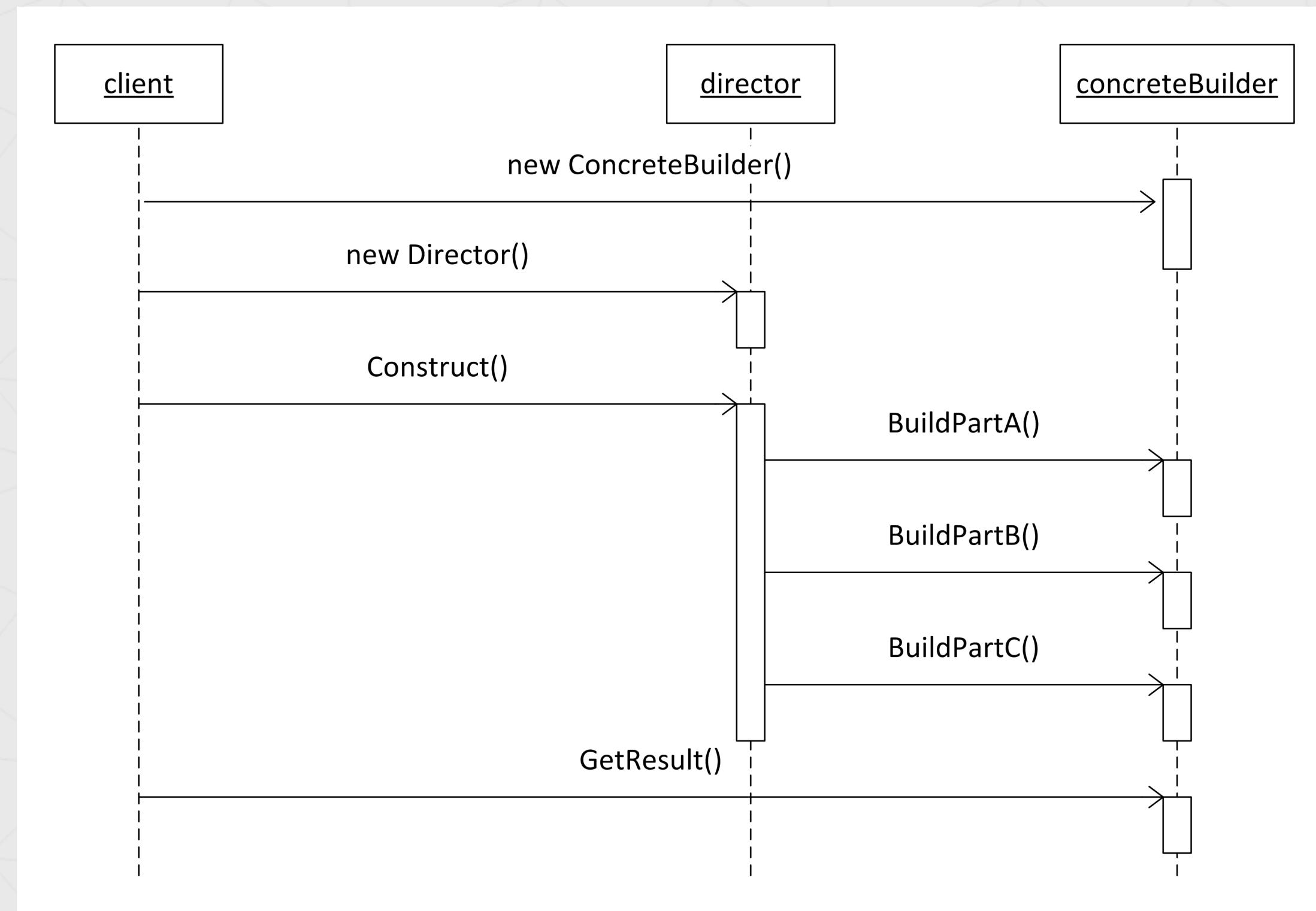
Builder - Problem

- Chcemy zhermetyzować operacje niezbędne do stworzenia złożonego obiektu oraz ukryć wewnętrzną reprezentację produktu przed klientem
- Chcemy mieć możliwość modyfikacji poszczególnych kroków algorytmu

Builder - Struktura



Builder - Współpraca



Builder - Konsekwencje

Builder - Konsekwencje

- Umożliwia zmiany wewnętrznej reprezentacji produktu

Builder - Konsekwencje

- Umożliwia zmiany wewnętrznej reprezentacji produktu
- Oddziela kod służący do konstruowania od reprezentacji

Builder - Konsekwencje

- Umożliwia zmiany wewnętrznej reprezentacji produktu
- Oddziela kod służący do konstruowania od reprezentacji
- Ulepsza kontrolę procesu konstruowania
 - w odróżnieniu od innych wzorców kreacyjnych, we wzorcu Builder obiekty konstruowane są krok po kroku pod nadzorem obiektu kierownika (Director)

Builder - Konsekwencje

- Umożliwia zmiany wewnętrznej reprezentacji produktu
- Oddziela kod służący do konstruowania od reprezentacji
- Ulepsza kontrolę procesu konstruowania
 - w odróżnieniu od innych wzorców kreacyjnych, we wzorcu Builder obiekty konstruowane są krok po kroku pod nadzorem obiektu kierownika (Director)
- Proces konstrukcji może trwać w czasie !!!
 - Dopiero po ukończeniu produktu kierownik (Director) odbiera go od budowniczego

Builder - Implementacja

- Interfejs montowania i konstruowania produktów
 - Interfejs Builder musi być na tyle ogólny, żeby było możliwe konstruowanie produktów przez wszystkich budowniczych konkretnych
- Brak klasy abstrakcyjnej produktów – nie jest wymagana

Builder - Pokrewne wzorce

■ Abstract Factory

- podobnie jak Builder może konstruować obiekty złożone
- różnica – wzorzec Builder kładzie nacisk na tworzenie produktów krok po kroku a Abstract Factory kładzie nacisk na rodziny produktów

■ Decorator, Composite

- Builder jest często używany do budowy łańcuchów obiektów

Builder - Podsumowanie

- Oddziela konstrukcję złożonych obiektów od ich reprezentacji
- Ten sam proces konstrukcyjny może prowadzić do powstania obiektów o różnej reprezentacji
- Jest często używany do budowania obiektów kompozytowych (wzorzec Composite)

Creational Patterns - Podsumowanie

- Factory Method
- Abstract Factory
- Singleton
- Prototype
- Builder

Wzorce strukturalne

Wzorce strukturalne

- Adapter

Wzorce strukturalne

- Adapter
- Decorator

Wzorce strukturalne

- Adapter
- Decorator
- Composite

Wzorce strukturalne

- Adapter
- Decorator
- Composite
- Proxy

Wzorce strukturalne

- Adapter
- Decorator
- Composite
- Proxy
- Facade

Wzorce strukturalne

- Adapter
- Decorator
- Composite
- Proxy
- Facade
- Bridge

Wzorce strukturalne

- Adapter
- Decorator
- Composite
- Proxy
- Facade
- Bridge
- Flyweight

Wzorce strukturalne rozwiążają problem poprzez złożenie klas i obiektów w większe struktury.

Adapter

Adapter - Przeznaczenie

- Dokonuje konwersji interfejsu danej klasy do postaci zgodnej z oczekiwaniami klienta
- Pozwala na wzajemną współpracę klas, które ze względu na niekompatybilne interfejsy wcześniej nie mogły ze sobą współpracować

Adapter - Kontekst / Problem

Adapter - Kontekst / Problem

■ Kontekst

- Interfejs wymagany przez klienta i interfejs klasy dostarczającej implementację nie są ze sobą zgodne

Adapter - Kontekst / Problem

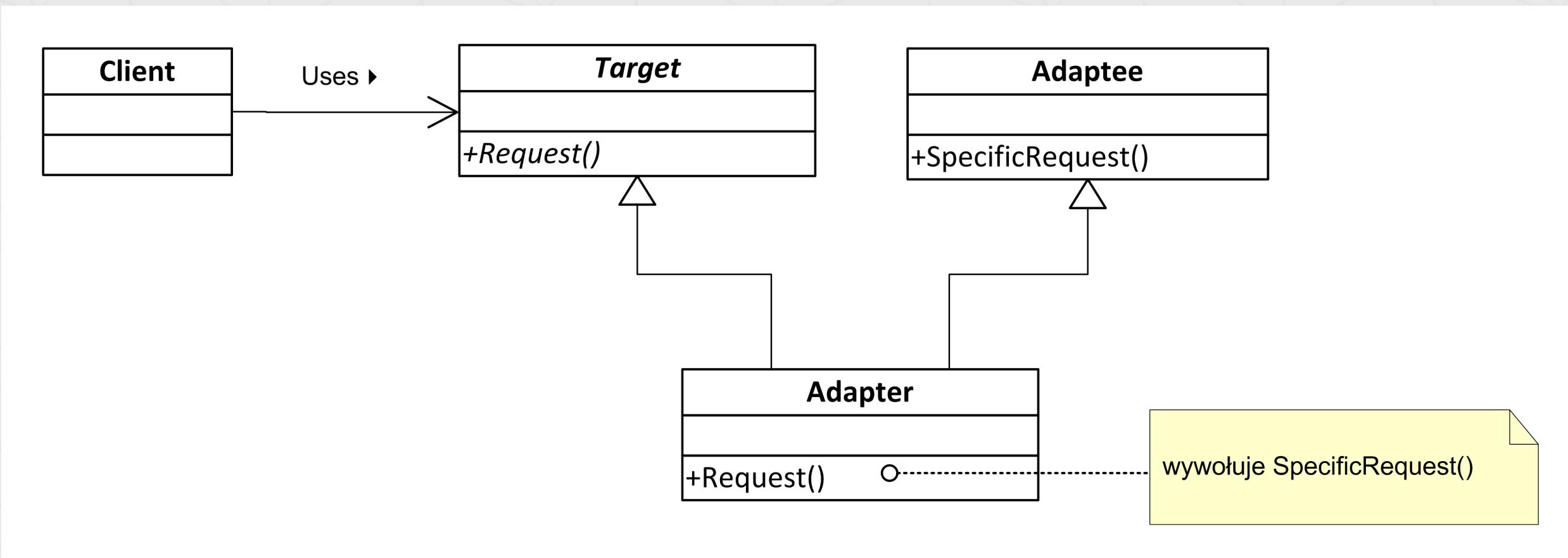
■ Kontekst

- Interfejs wymagany przez klienta i interfejs klasy dostarczającej implementację nie są ze sobą zgodne

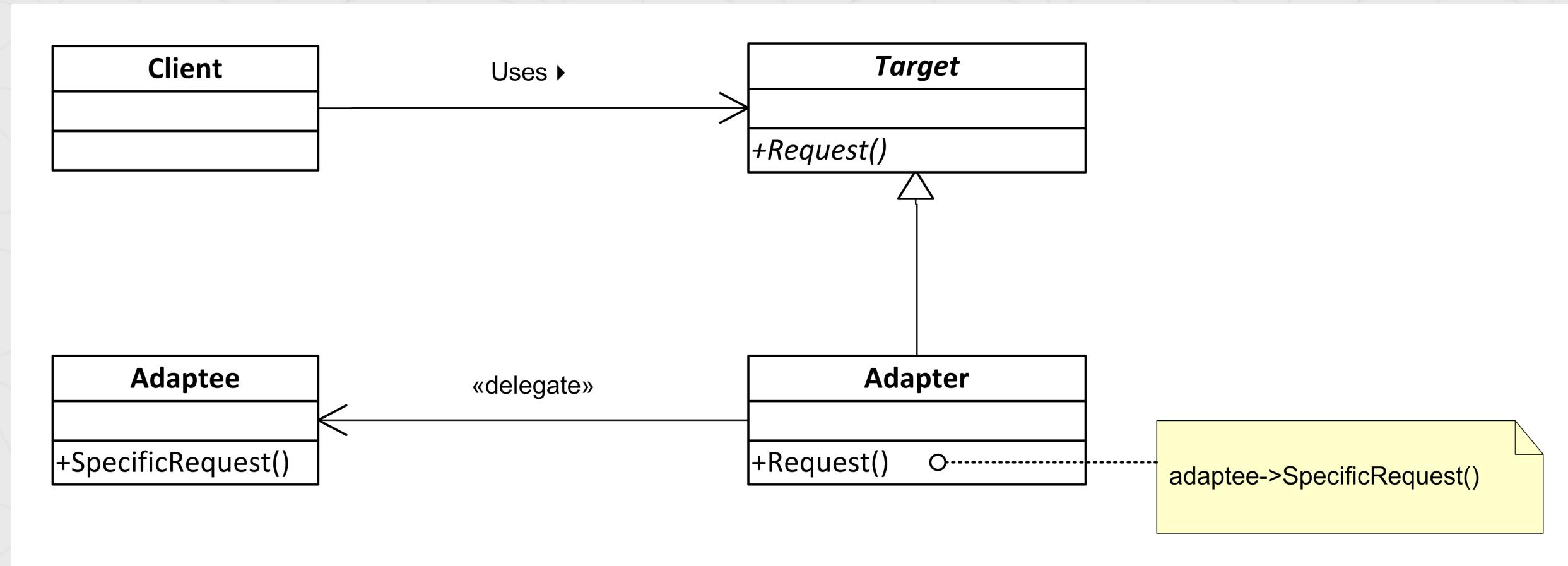
■ Problem

- Chcemy wykorzystać istniejącą klasę, a jej interfejs nie odpowiada temu, którego potrzebujemy

Adapter klas



Adapter obiektów



Adapter klas

- Adaptuje dostosowując się do klasy konkretnej Adaptee – nie będzie więc działał wtedy, gdy będziemy chcieli zaadaptować klasę oraz jej wszystkie podklasy
- Umożliwia klasie Adapter przeddefiniowanie części zachowania Adaptee
- Wprowadza tylko jeden obiekt, aby dostać się do adaptowanego

Adapter obiektów

- Umożliwia jednemu adapterowi działanie z wieloma adaptowanymi – z samą klasą Adaptee oraz jej wszystkimi podklasami
- Utrudnia przeddefiniowanie zachowania adaptowanego – wymaga w tym celu tworzenia podklas adaptowanego i odwoływanie się Adaptera do nich, a nie do samego adaptowanego (Adaptee)

Adapter - Podsumowanie

- Zmienia interfejs istniejącego obiektu i dostosowuje go do oczekiwania klienta
- Klasy nie związane ze sobą mogą współpracować pomimo niezgodnych interfejsów

Decorator

Decorator - Przeznaczenie

- Pozwala na dynamiczne przydzielanie danemu obiektowi nowych zachowań
- Zapewnia elastyczną alternatywę dla tworzenia podklas w celu rozszerzania funkcjonalności

Decorator - Scenariusz

Chcemy rozszerzyć możliwości obiektu Photo i dodać do zdjęcia ramkę oraz dwa tagi



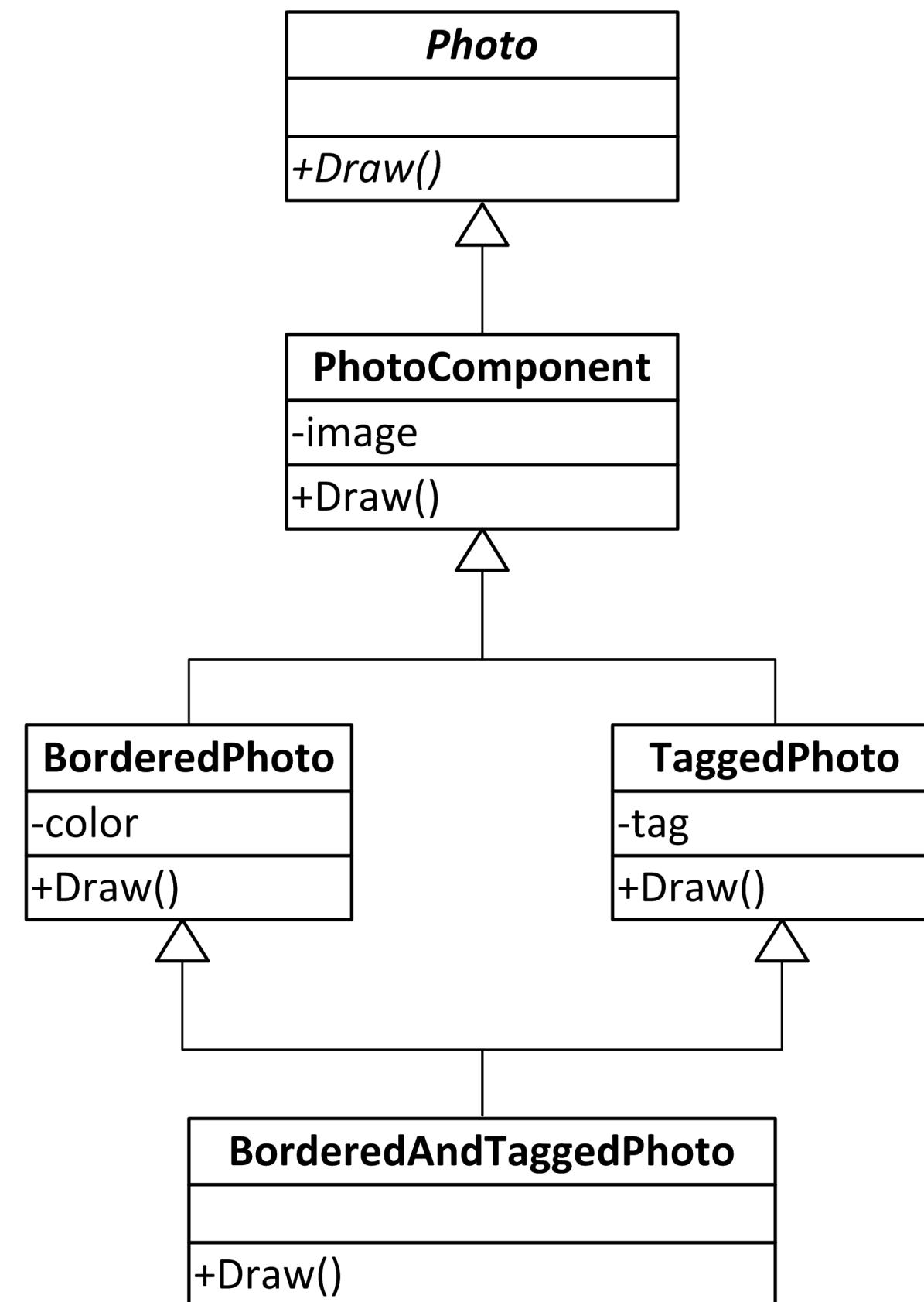


coffee

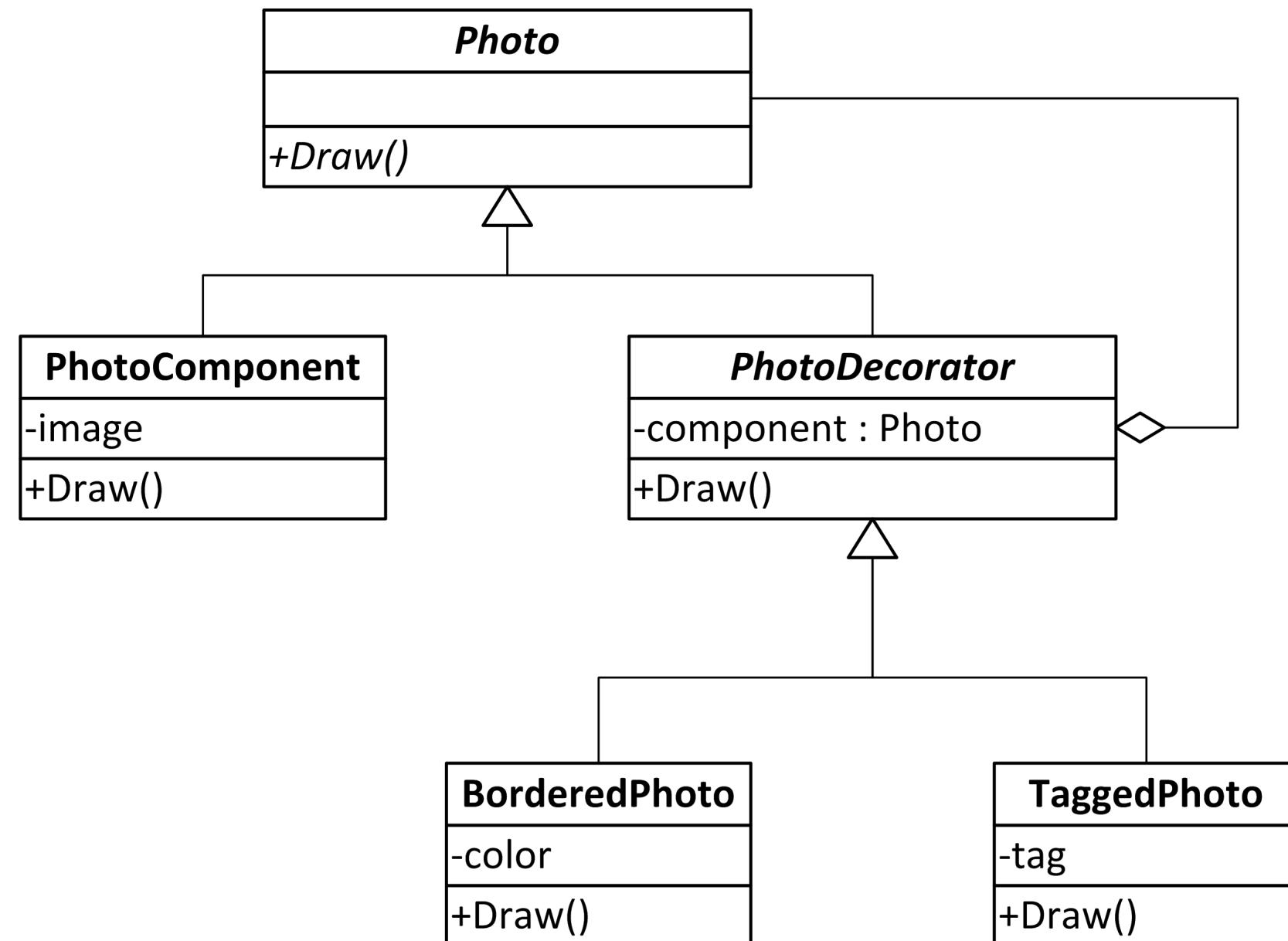


Dwie możliwe implementacje:

Rozwiązanie 1 - Dziedziczenie



Rozwiązanie 2 - Dekoracja komponentu



Dekoracja komponentu

Dekoracja komponentu

- Umieszczenie komponentu w innym obiekcie, który dodaje ramkę.

Dekoracja komponentu

- Umieszczenie komponentu w innym obiekcie, który dodaje ramkę.
- Obiekt będący otoczką komponentu nazywa się **dekoratorem**

Dekoracja komponentu

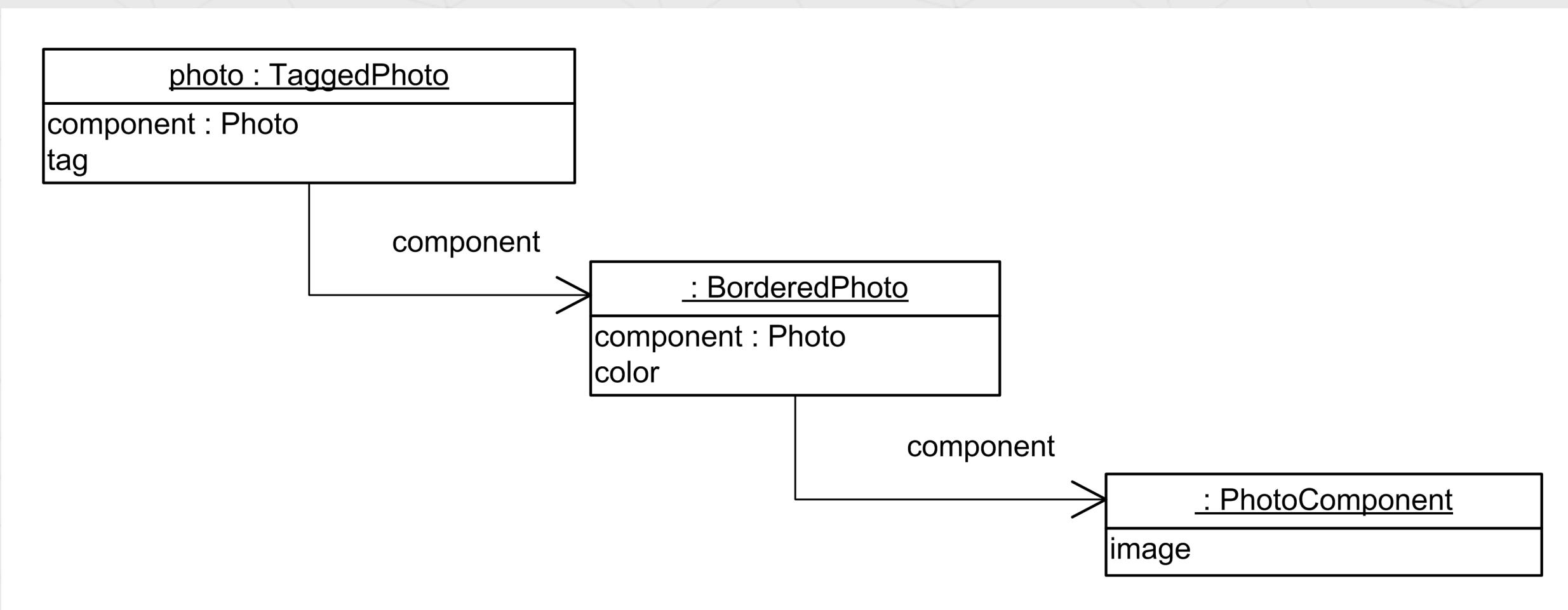
- Umieszczenie komponentu w innym obiekcie, który dodaje ramkę.
- Obiekt będący otoczką komponentu nazywa się **dekoratorem**
- Dekorator dostosowuje się do interfejsu ozdabianego obiektu, dzięki czemu staje się przezroczysty dla klientów

Dekoracja komponentu

- Umieszczenie komponentu w innym obiekcie, który dodaje ramkę.
- Obiekt będący otoczką komponentu nazywa się **dekoratorem**
- Dekorator dostosowuje się do interfejsu ozdabianego obiektu, dzięki czemu staje się przezroczysty dla klientów
- Dekorator przekazuje żądania do komponentu i może wykonywać dodatkowe akcje

Decorator - Współpraca

Wzorzec dekoratora umożliwia składanie obiektu Photo z dekoratorami BorderedPhoto i TaggedPhoto



Decorator - Kontekst

Decorator - Kontekst

- Dziedziczenie jest jedną z form rozszerzenia funkcjonalności klasy, ale niekoniecznie musi być najlepszym sposobem na osiągnięcie w pełni elastycznych projektów aplikacji

Decorator - Kontekst

- Dziedziczenie jest jedną z form rozszerzenia funkcjonalności klasy, ale niekoniecznie musi być najlepszym sposobem na osiągnięcie w pełni elastycznych projektów aplikacji
- Tworząc projekt aplikacji, należy go tak skonstruować, aby możliwe było rozszerzanie zachowań poszczególnych elementów bez konieczności modyfikowania istniejącego kodu

Decorator - Kontekst

- Dziedziczenie jest jedną z form rozszerzenia funkcjonalności klasy, ale niekoniecznie musi być najlepszym sposobem na osiągnięcie w pełni elastycznych projektów aplikacji
- Tworząc projekt aplikacji, należy go tak skonstruować, aby możliwe było rozszerzanie zachowań poszczególnych elementów bez konieczności modyfikowania istniejącego kodu
- Wykorzystując kompozycję oraz delegację, można dodawać nowe zachowania podczas działania programu

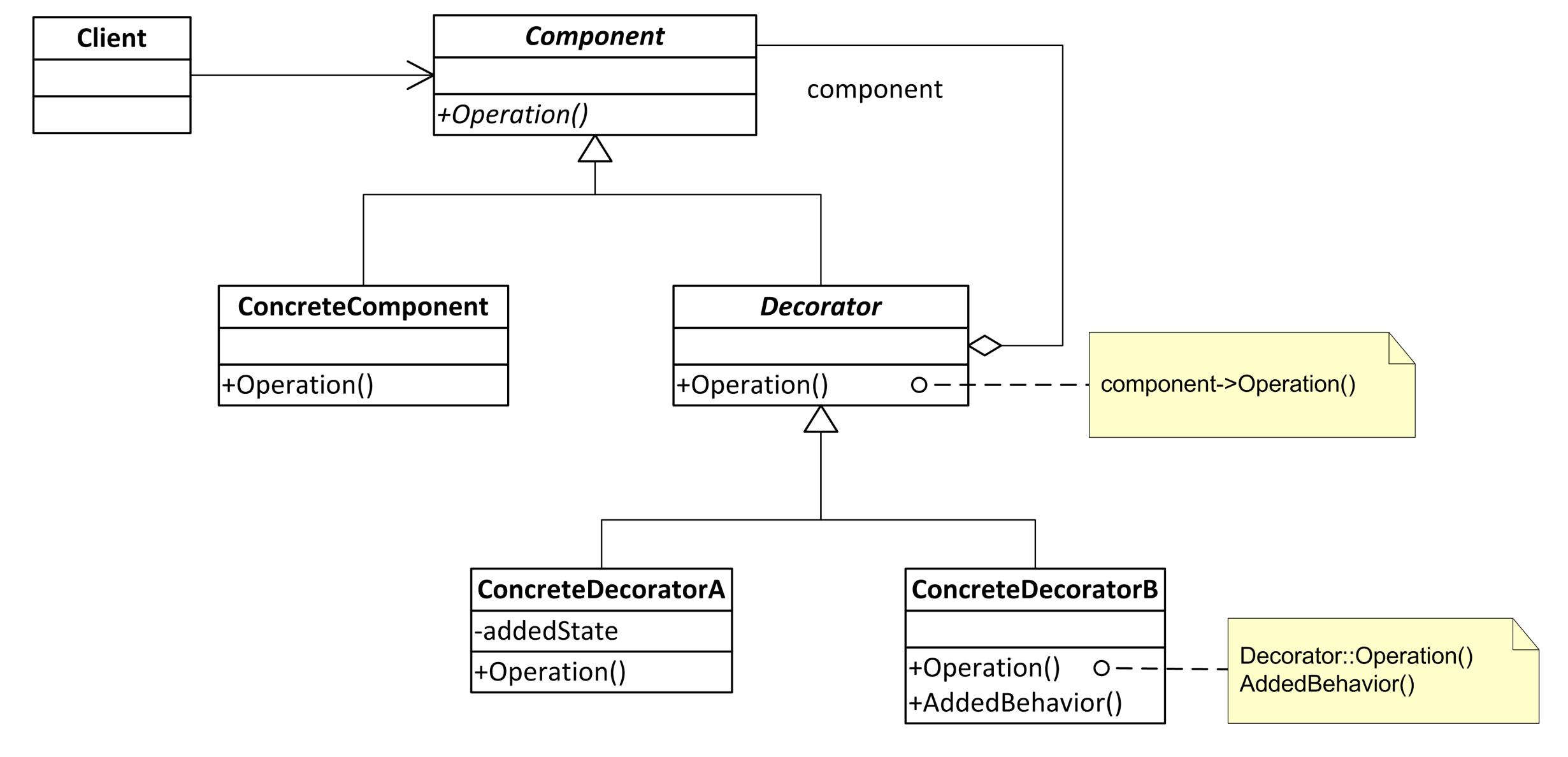
Decorator - Kontekst

- Dziedziczenie jest jedną z form rozszerzenia funkcjonalności klasy, ale niekoniecznie musi być najlepszym sposobem na osiągnięcie w pełni elastycznych projektów aplikacji
- Tworząc projekt aplikacji, należy go tak skonstruować, aby możliwe było rozszerzanie zachowań poszczególnych elementów bez konieczności modyfikowania istniejącego kodu
- Wykorzystując kompozycję oraz delegację, można dodawać nowe zachowania podczas działania programu
- Wzorzec Decorator posługuje się zbiorem klas dekorujących (dekoratorów), które są wykorzystywane do dekorowania poszczególnych obiektów (komponentów)

Decorator - Stosowalność

- Wzorzec Decorator powinien być stosowany:
 - aby dynamicznie i w przezroczysty sposób (tzn. nie wpływający na inne obiekty) dodać zobowiązania do pojedynczych obiektów
 - w wypadku zobowiązań, które mogą być cofnięte
 - gdy rozszerzanie funkcjonalności przez definiowanie podklas jest niepraktyczne
 - czasami jest możliwych wiele niezależnych rozszerzeń, które przy próbie uwzględnienia różnych ich kombinacji prowadzą do gwałtownego wzrostu liczby klas

Decorator - Struktura



Decorator - konsekwencje [1]

- Większa elastyczność niż przy stosowaniu statycznego dziedziczenia
 - mając dekoratory można dodawać i usuwać zobowiązania w czasie wykonywania programu
 - Uzgłađnienie różnych klas Decorator dla określonej klasy komponentu umożliwia mieszanie i dopasowywanie zobowiązań
 - dekoratory ułatwiają także dwukrotne dołączanie właściwości (np. fotografia z podwójną ramką)

Decorator - konsekwencje [2]

- Unikanie przeładowania właściwościami klas na szczycie hierarchii
 - możliwe jest zdefiniowanie prostej klasy i przyrostowe rozszerzanie jej funkcjonalności za pomocą obiektów dekoratora
 - nowe rodzaje dekoratorów są łatwe do zdefiniowania
- Dekorator i jego komponent nie są identyczne
 - dekorator działa jak przezroczysta otoczka, jednak z punktu widzenia identyczności obiektów udekorowany komponent nie jest taki sam jak ten wyjściowy

Decorator - konsekwencje [3]

- Wiele małych obiektów
 - projekty wykorzystujące dekoratory prowadzą często do powstawanie systemów z dużą liczbą małych, podobnych do siebie obiektów

Decorator - implementacja

■ Zgodność interfejsów

- interfejs obiektu będącego dekoratorem musi odpowiadać interfejsowi dekorowanego przez niego komponentu
- klasy ConcreteDecorator muszą dziedziczyć po wspólnej klasie

■ Pomijanie klasy abstrakcyjnej Decorator

- gdy zależy nam na dodaniu tylko jednego zobowiązania, nie musimy definiować klasy abstrakcyjnej Decorator

■ Utrzymanie klas Component w wadze lekkiej

- ekstrakcja interfejsu dla klasy komponentu

Decorator - pokrewne wzorce

■ Composite

- wzorzec Decorator można uważać za zdegenerowany kompozyt, z jednym komponentem. Decorator jednak dodaje dodatkowe zobowiązania, nie jest przeznaczony do agregacji obiektów

■ Strategy

- wzorzec Decorator umożliwia zmianę skóry obiektu, a Strategy zmianę jego wnętrza

■ Builder

- ułatwia tworzenie łańcuchów dekoratorów

Decorator - podsumowanie

Decorator - podsumowanie

- Umożliwia dynamiczne dodawanie zobowiązań do obiektu

Decorator - podsumowanie

- Umożliwia dynamiczne dodawanie zobowiązań do obiektu
- Posługuje się zbiorem klas dekorujących (dekoratorów), które są wykorzystywane do dekorowania poszczególnych obiektów (składników)

Decorator - podsumowanie

- Umożliwia dynamiczne dodawanie zobowiązań do obiektu
- Posługuje się zbiorem klas dekorujących (dekoratorów), które są wykorzystywane do dekorowania poszczególnych obiektów (składników)
- Dekoratory mają ten sam interfejs, co obiekty dekorowane

Decorator - podsumowanie

- Umożliwia dynamiczne dodawanie zobowiązań do obiektu
- Posługuje się zbiorem klas dekorujących (dekoratorów), które są wykorzystywane do dekorowania poszczególnych obiektów (składników)
- Dekoratory mają ten sam interfejs, co obiekty dekorowane
- Dekoratory zmieniają zachowania obiektów dekorowanych (składników), dodając nowe zachowania przed wywołaniami metod danego składnika i (lub) po nich lub nawet pomiędzy nimi

Decorator - podsumowanie

- Umożliwia dynamiczne dodawanie zobowiązań do obiektu
- Posługuje się zbiorem klas dekorujących (dekoratorów), które są wykorzystywane do dekorowania poszczególnych obiektów (składników)
- Dekoratory mają ten sam interfejs, co obiekty dekorowane
- Dekoratory zmieniają zachowania obiektów dekorowanych (składników), dodając nowe zachowania przed wywołaniami metod danego składnika i (lub) po nich lub nawet pomiędzy nimi
- Każdy składnik może być "otoczony" dowolną ilością dekoratorów

Composite

Composite - Przeznaczenie

Composite - Przeznaczenie

- Składa obiekty w struktury drzewiaste reprezentujące hierarchie typu część-całość

Composite - Przeznaczenie

- Składa obiekty w struktury drzewiaste reprezentujące hierarchie typu część-całość
- Umożliwia klientom jednakowe traktowanie pojedynczych obiektów i ich agregatów

Composite - Kontekst / Problem

Composite - Kontekst / Problem

■ Kontekst

- chcemy utworzyć reprezentację dla hierarchii obiektów
- hierarchia obiektów ma wspólną klasę bazową (klasę abstrakcyjną lub interfejs)

Composite - Kontekst / Problem

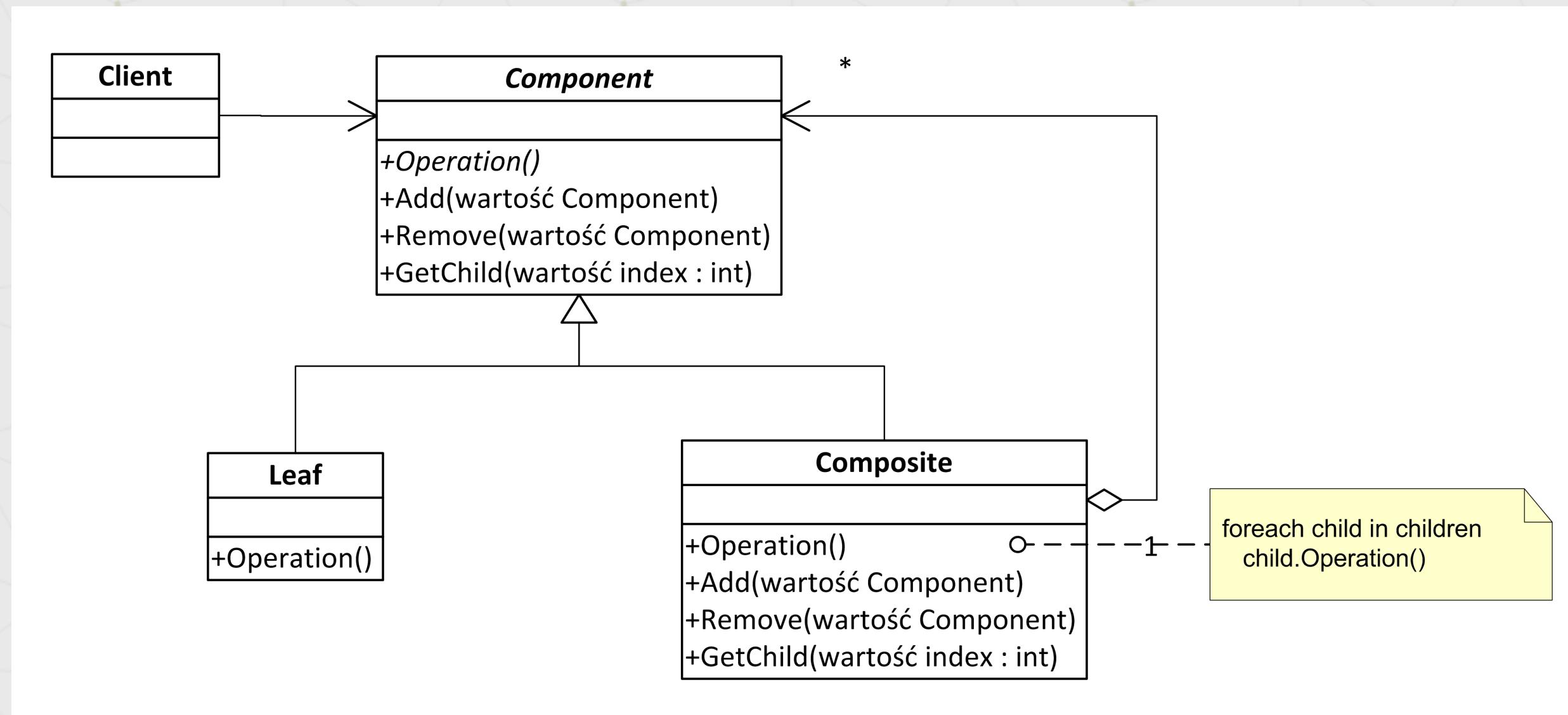
■ Kontekst

- chcemy utworzyć reprezentację dla hierarchii obiektów
- hierarchia obiektów ma wspólną klasę bazową (klasę abstrakcyjną lub interfejs)

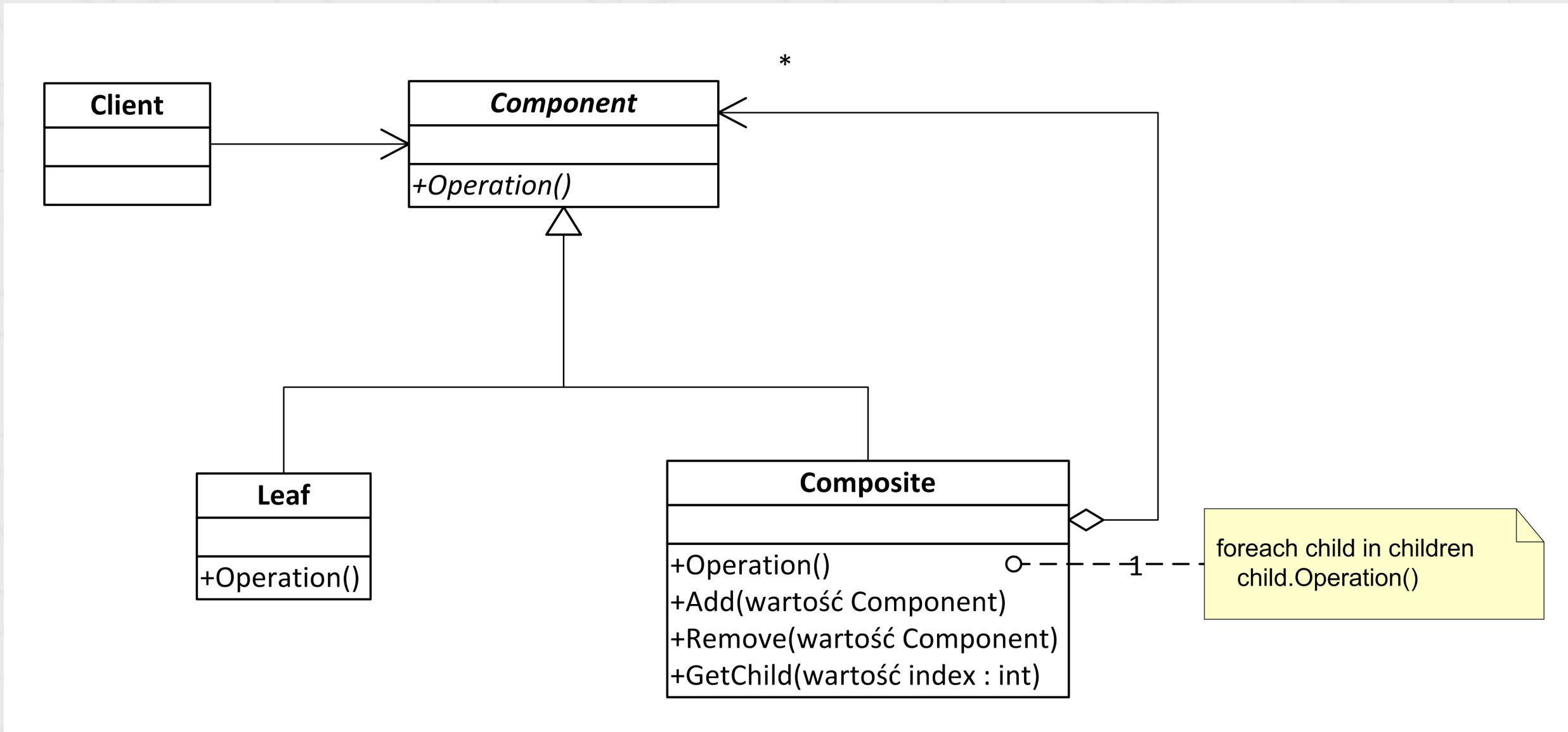
■ Problem

- chcemy, aby klienci mogli ignorować różnicę między agregatami obiektów a pojedynczymi obiektami – klienci będą wtedy jednakowo traktować wszystkie obiekty występujące w strukturze

Composite - Struktura



Composite - Struktura (Alternative Take)



Composite - implementacja

```
01 class ShapeGroup : public Shape
02 {
03     using ShapePtr = std::shared_ptr<Shape>;
04     std::vector<ShapePtr> shapes_;
05 public:
06     ShapeGroup() = default;
07
08     void draw() const
09     {
10         for(const auto& shp : shapes_)
11             shp->draw();
12     }
13
14     void add(ShapePtr shp)
15     {
16         shapes_.push_back(shp);
17     }
18 };
```

Composite - implementacja

```
01 class ShapeGroup : public Shape
02 {
03     using ShapePtr = std::shared_ptr<Shape>;
04     std::vector<ShapePtr> shapes_;
05 public:
06     ShapeGroup() = default;
07
08     void draw() const
09     {
10         for(const auto& shp : shapes_)
11             shp->draw();
12     }
13
14     void add(ShapePtr shp)
15     {
16         shapes_.push_back(shp);
17     }
18 };
```

Composite - implementacja

```
01 class ShapeGroup : public Shape
02 {
03     using ShapePtr = std::shared_ptr<Shape>;
04     std::vector<ShapePtr> shapes_;
05 public:
06     ShapeGroup() = default;
07
08     void draw() const
09     {
10         for(const auto& shp : shapes_)
11             shp->draw();
12     }
13
14     void add(ShapePtr shp)
15     {
16         shapes_.push_back(shp);
17     }
18 };
```

Composite - implementacja

```
01 class ShapeGroup : public Shape
02 {
03     using ShapePtr = std::shared_ptr<Shape>;
04     std::vector<ShapePtr> shapes_;
05 public:
06     ShapeGroup() = default;
07
08     void draw() const
09     {
10         for(const auto& shp : shapes_)
11             shp->draw();
12     }
13
14     void add(ShapePtr shp)
15     {
16         shapes_.push_back(shp);
17     }
18 };
```

Composite - implementacja

```
01 class ShapeGroup : public Shape
02 {
03     using ShapePtr = std::shared_ptr<Shape>;
04     std::vector<ShapePtr> shapes_;
05 public:
06     ShapeGroup() = default;
07
08     void draw() const
09     {
10         for(const auto& shp : shapes_)
11             shp->draw();
12     }
13
14     void add(ShapePtr shp)
15     {
16         shapes_.push_back(shp);
17     }
18 };
```

Composite - Współpraca

- Klienci używają interfejsu z klasy Component w celu komunikowania się z obiektami występującymi w składanej strukturze

Composite - Współpraca

- Klienci używają interfejsu z klasy Component w celu komunikowania się z obiektami występującymi w składanej strukturze
 - jeśli odbiorca jest liściem, to żądania są realizowane bezpośrednio

Composite - Współpraca

- Klienci używają interfejsu z klasy Component w celu komunikowania się z obiektami występującymi w składanej strukturze
 - jeśli odbiorca jest liściem, to żądania są realizowane bezpośrednio
 - jeśli odbiorca jest kompozytem (Composite), to zwykle przekazuje swoje żądania komponentom-dzieciom, wykonując ewentualnie przed i/ lub po przekazaniu dodatkowe operacje

Composite - Konsekwencje

- Wzorzec Composite definiuje hierarchie klas grupujących obiekty pierwotne i agregaty
- Uproszczenie budowy klienta – klienci mogą jednakowo traktować struktury złożone i pojedyncze obiekty
- Umieszczenie operacji dodawania nowych komponentów do agregatów w klasie bazowej Component może naruszać zasadę podstawiania Liskov

Composite - Implementacja (1)

Composite - Implementacja (1)

■ Jawne odwołania do rodziców

- przechowywanie odwołań z komponentów-dzieci do ich rodziców może ułatwiać poruszanie się po strukturze kompozytu i zarządzanie nią
- typowym miejscem do definiowania odwołania do rodzica jest klasa **Component**
- klasy **Leaf** i **Composite** mogą dziedziczyć to odwołanie i zarządzające nim operacje

Composite - Implementacja (2)

Composite - Implementacja (2)

- Iteracja po elementach agregatu
 - implementacja wzorca Iterator dla struktury kompozytowej

Composite - Implementacja (2)

- Iteracja po elementach agregatu
 - implementacja wzorca Iterator dla struktury kompozytowej
- Współdzielenie komponentów
 - bardzo często opłacalne jest współdzielenie komponentów, na przykład w celu ograniczenia wymagań pamięciowych
 - implementacja obiektów struktury jako `const` (*immutable*)

Composite - Implementacja (3)

Composite - Implementacja (3)

- Kto powinien usuwać komponenty?
 - czy obiekt kompozytu posiada prawo własności (*ownership*) do obiektów podrzędnych

Composite - Implementacja (3)

- Kto powinien usuwać komponenty?
 - czy obiekt kompozytu posiada prawo własności (*ownership*) do obiektów podrzędnych
- Jaka struktura danych jest najlepsza do przechowywania komponentów?
 - `std::vector<T>`
 - `std::list<T>`
 - `std::unordered_map<K, T>`

Composite - Pokrewne wzorce

Composite - Pokrewne wzorce

- Wzorzec Composite może być użyty do reprezentacji grupy obiektów we wzorcach
 - Chain Of Responsibility
 - Visitor

Composite - Pokrewne wzorce

- Wzorzec Composite może być użyty do reprezentacji grupy obiektów we wzorcach
 - Chain Of Responsibility
 - Visitor
- Flyweight
 - umożliwia implementację współdzielenia komponentów

Composite - Pokrewne wzorce

- Wzorzec Composite może być użyty do reprezentacji grupy obiektów we wzorcach
 - Chain Of Responsibility
 - Visitor
- Flyweight
 - umożliwia implementację współdzielenia komponentów
- Iterator
 - umożliwia iterację po całej hierarchii obiektów

Proxy

Proxy - Przeznaczenie

- Wzorzec Proxy zapewnia substytut lub reprezentanta innego obiektu w celu sterowania dostępem do niego

Proxy - Kontekst / Problem

Proxy - Kontekst / Problem

■ Kontekst

- tworzenie obiektów i ich inicjalizacja w trakcie działania programu jest kosztowne
- potrzebna jest kontrola dostępu do obiektu

Proxy - Kontekst / Problem

■ Kontekst

- tworzenie obiektów i ich inicjalizacja w trakcie działania programu jest kosztowne
- potrzebna jest kontrola dostępu do obiektu

■ Problem

- optymalizacja kosztownych procesów lub kontrola dostępu powinna być przezroczysta dla klienta

Proxy - Scenariusz

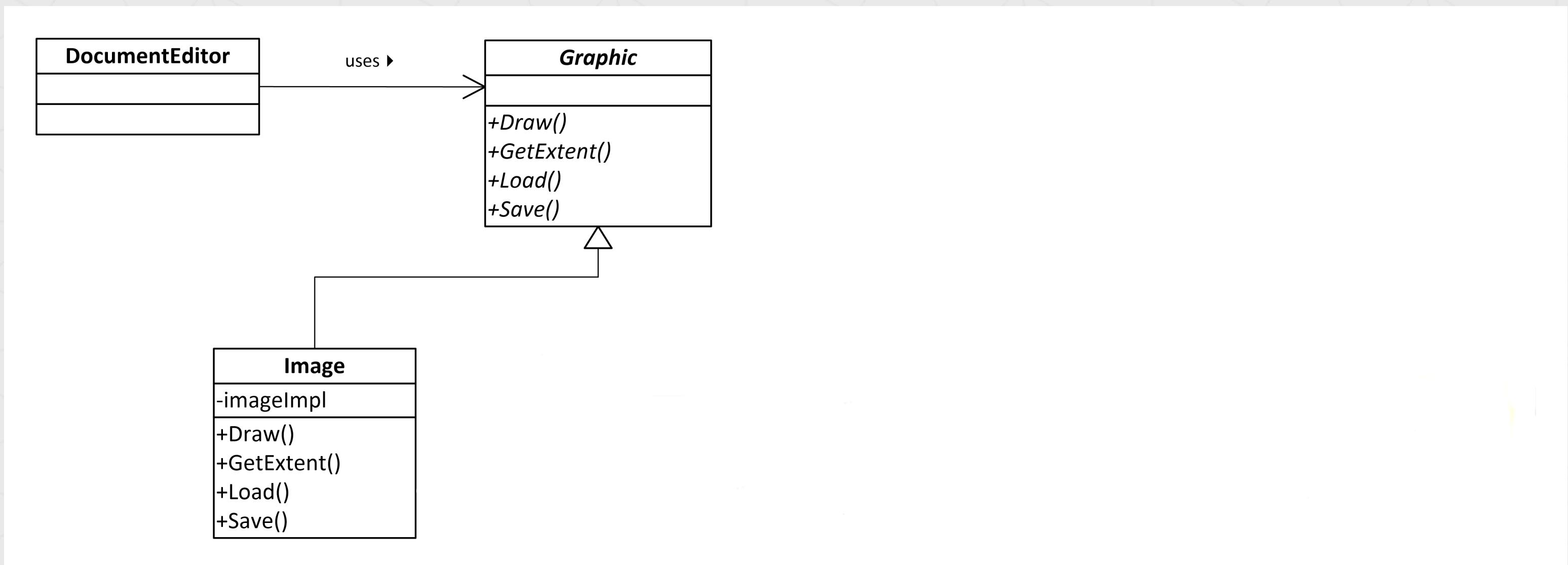
Proxy - Scenariusz

- Chcemy napisać edytor dokumentów, który umożliwia osadzanie obiektów graficznych
 - otwieranie dokumentów powinno być szybkie
 - optymalizacja nie powinna mieć wpływu na części programu związane z wyświetlaniem czy formatowaniem

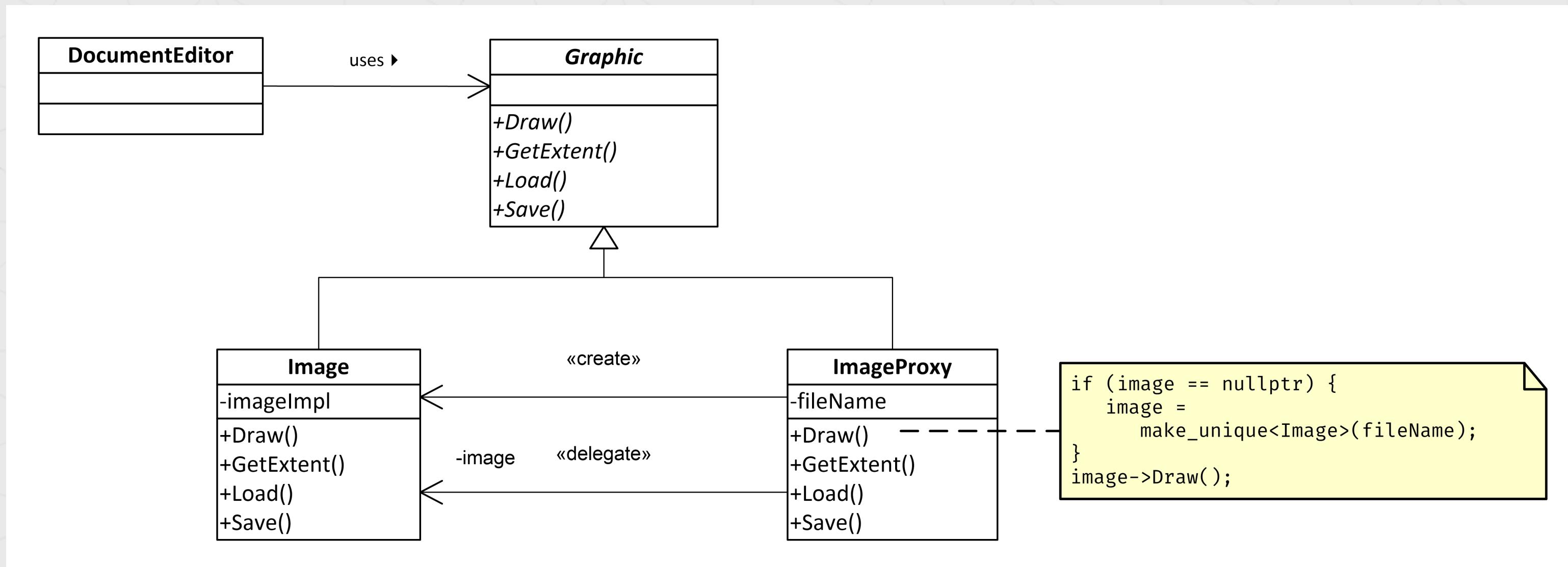
Proxy - Scenariusz

- Chcemy napisać edytor dokumentów, który umożliwia osadzanie obiektów graficznych
 - otwieranie dokumentów powinno być szybkie
 - optymalizacja nie powinna mieć wpływu na części programu związane z wyświetlaniem czy formatowaniem
- Rozwiązanie
 - użycie innego obiektu, pełnomocnika rysunku, który będzie zastępował prawdziwy rysunek
 - obiekt proxy zachowuje się jak rysunek i zajmuje się jego utworzeniem, gdy będzie to konieczne (*lazy loading*)

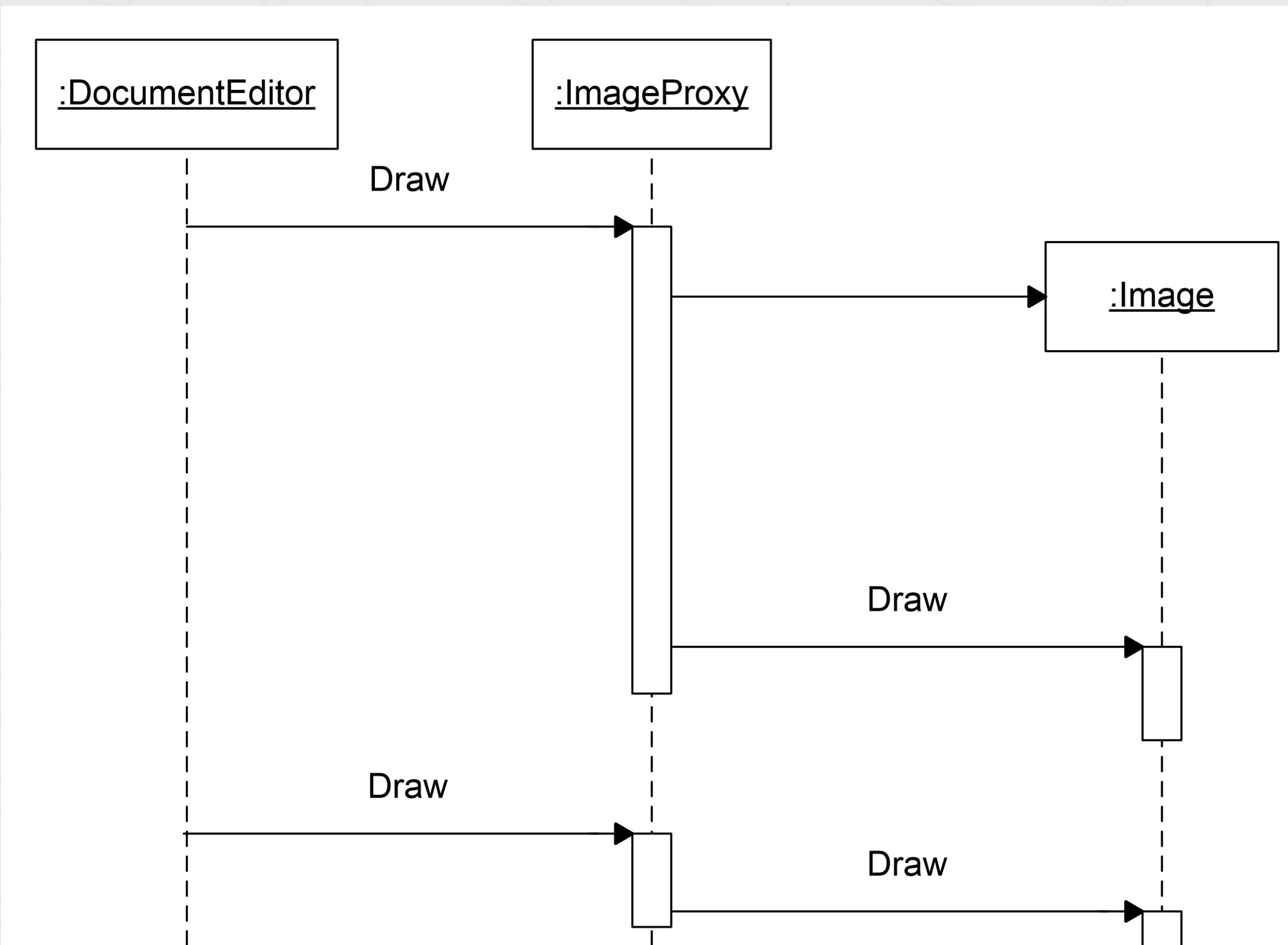
Proxy - Scenariusz - UML



Proxy - Scenariusz - UML



Proxy - Scenariusz - UML



Proxy - Stosowalność

- Proxy ma zastosowanie zawsze wtedy, gdy potrzeba bardziej uniwersalnego lub bardziej wyrafinowanego odwołania do obiektu

Rodzaje proxy

- Rodzaje obiektów Proxy

Rodzaje proxy

■ Rodzaje obiektów Proxy

- remote proxy – jest lokalnym reprezentantem obiektu znajdującego się w innej przestrzeni adresowej (RPC)

Rodzaje proxy

■ Rodzaje obiektów Proxy

- remote proxy – jest lokalnym reprezentantem obiektu znajdującego się w innej przestrzeni adresowej (RPC)
- virtual proxy – tworzy kosztowne obiekty na żądanie

Rodzaje proxy

■ Rodzaje obiektów Proxy

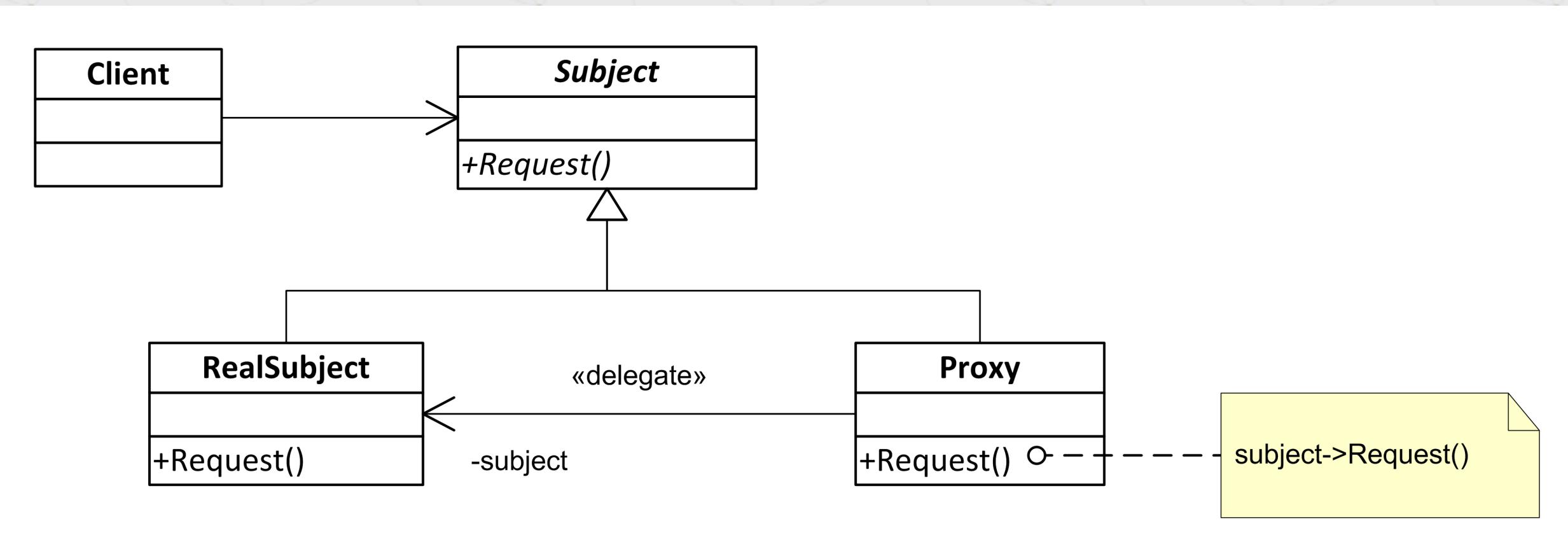
- remote proxy – jest lokalnym reprezentantem obiektu znajdującego się w innej przestrzeni adresowej (RPC)
- virtual proxy – tworzy kosztowne obiekty na żądanie
- protection proxy – kontroluje dostęp do oryginalnego obiektu

Rodzaje proxy

■ Rodzaje obiektów Proxy

- remote proxy – jest lokalnym reprezentantem obiektu znajdującego się w innej przestrzeni adresowej (RPC)
- virtual proxy – tworzy kosztowne obiekty na żądanie
- protection proxy – kontroluje dostęp do oryginalnego obiektu
- smart proxy – modyfikuje żądanie przed przesłaniem go do oryginalnego obiektu

Proxy - Struktura



Proxy - Współpraca

- Obiekt pełnomocnika (proxy), jeśli trzeba, przekazuje żądania do prawdziwego obiektu (subject)



Proxy - Konsekwencje

Proxy - Konsekwencje

- Wzorzec Proxy wprowadza dodatkowy poziom pośredniości przy dostępie do obiektu

Proxy - Konsekwencje

- Wzorzec Proxy wprowadza dodatkowy poziom pośredniości przy dostępie do obiektu
 - Remote Proxy – może ukryć fakt, że obiekt znajduje się w innej przestrzeni adresowej

Proxy - Konsekwencje

- Wzorzec Proxy wprowadza dodatkowy poziom pośredniości przy dostępie do obiektu
 - Remote Proxy – może ukryć fakt, że obiekt znajduje się w innej przestrzeni adresowej
 - Virtual Proxy – może wykonywać optymalizacje, np. tworzenie obiektu na żądanie, kopiowanie-przy-zapisaniu

Proxy - Konsekwencje

- Wzorzec Proxy wprowadza dodatkowy poziom pośredniości przy dostępie do obiektu
 - Remote Proxy – może ukryć fakt, że obiekt znajduje się w innej przestrzeni adresowej
 - Virtual Proxy – może wykonywać optymalizacje, np. tworzenie obiektu na żądanie, kopiowanie-przy-zapisaniu
 - Protection Proxy i Smart Proxy – umożliwiają wykonywanie dodatkowych czynności porządkowych przy dostępie do obiektu

Proxy - Pokrewne wzorce

- Decorator – pomimo podobnej implementacji, przeznaczenie wzorca Proxy jest inne:
 - Dekorator dodaje zobowiązania do obiektu
 - Proxy zarządza dostępem do obiektu

Proxy - Podsumowanie

Proxy - Podsumowanie

- Zapewnia obiekt pośrednika, dzięki któremu możemy optymalizować wywołanie kosztownych operacji lub kontrolować dostęp do oryginału

Proxy - Podsumowanie

- Zapewnia obiekt pośrednika, dzięki któremu możemy optymalizować wywołanie kosztownych operacji lub kontrolować dostęp do oryginału
- Interfejs obiektu Proxy jest taki sam jak interfejs oryginału

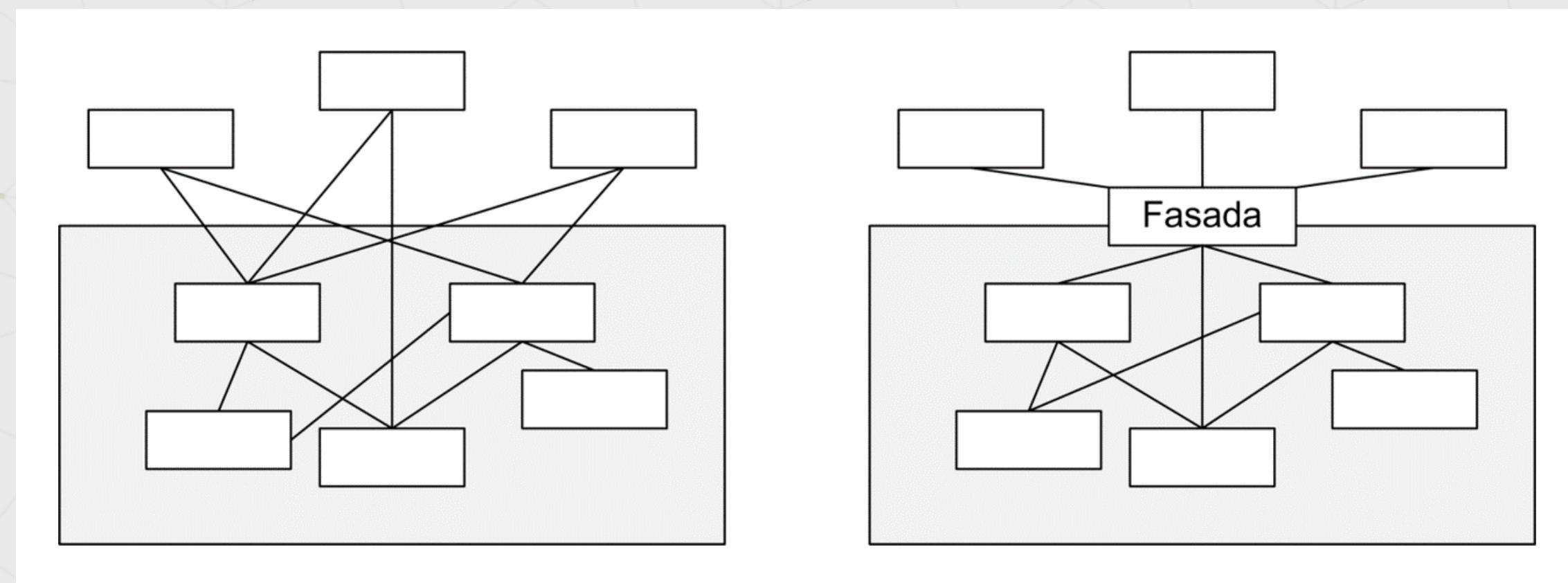
Façade

Façade

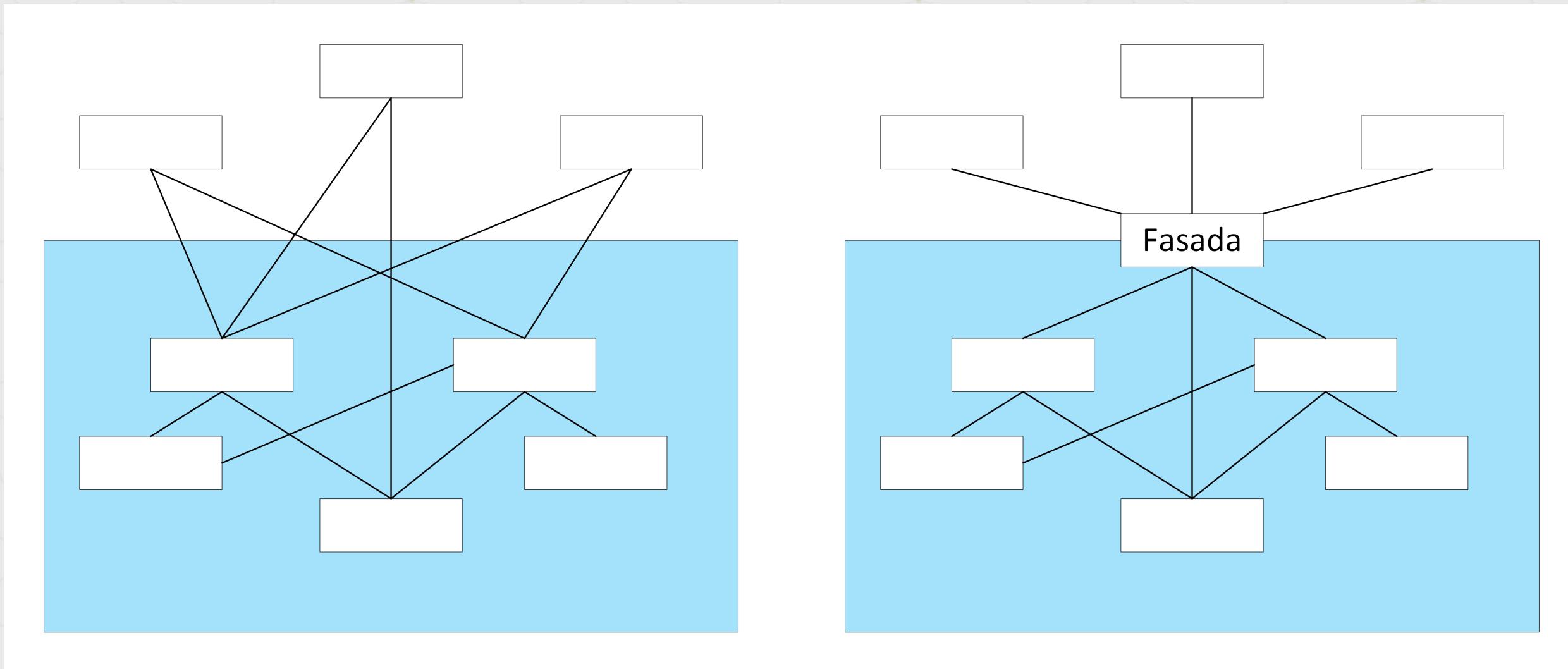
■ Przeznaczenie

- zapewnia jeden, zunifikowany interfejs dla całego zestawu interfejsów określonego podsystemu
- tworzy nowy interfejs wysokiego poziomu, który powoduje, że korzystanie z całego podsystemu staje się zdecydowanie łatwiejsze
- odseparowuje klienta od złożonych podsystemów

Façade



Façade - Struktura



Façade - Współpraca

Façade - Współpraca

- Klienci komunikują się z podsystemem, wysyłając żądania do fasady (Façade), która przekazuje je do odpowiednich obiektów podsystemu

Façade - Współpraca

- Klienci komunikują się z podsystemem, wysyłając żądania do fasady (Façade), która przekazuje je do odpowiednich obiektów podsystemu
- Klienci wykorzystujący fasadę nie muszą mieć bezpośredniego dostępu do obiektów z jej podsystemu

Façade - Konsekwencje

Façade - Konsekwencje

- Oddziela klientów od komponentów podsystemu, dzięki czemu zmniejsza się liczba obiektów, z którymi klienci mają do czynienia – podsystem staje się łatwiejszy do użycia

Façade - Konsekwencje

- Oddziela klientów od komponentów podsystemu, dzięki czemu zmniejsza się liczba obiektów, z którymi klienci mają do czynienia – podsystem staje się łatwiejszy do użycia
- Sprzyja słabemu powiązaniu podsystemu z jego klientami – umożliwia zmianę komponentów w sposób niewidoczny dla jego klientów

Façade - Konsekwencje

- Oddziela klientów od komponentów podsystemu, dzięki czemu zmniejsza się liczba obiektów, z którymi klienci mają do czynienia – podsystem staje się łatwiejszy do użycia
- Sprzyja słabemu powiązaniu podsystemu z jego klientami – umożliwia zmianę komponentów w sposób niewidoczny dla jego klientów
- Ułatwia ułożenie warstwami systemu i zależności między obiektami

Façade - Konsekwencje

- Oddziela klientów od komponentów podsystemu, dzięki czemu zmniejsza się liczba obiektów, z którymi klienci mają do czynienia – podsystem staje się łatwiejszy do użycia
- Sprzyja słabemu powiązaniu podsystemu z jego klientami – umożliwia zmianę komponentów w sposób niewidoczny dla jego klientów
- Ułatwia ułożenie warstwami systemu i zależności między obiektami
- Nie uniemożliwia aplikacjom bezpośredniego dostępu do podsystemu, jeśli go potrzebują

Façade - Pokrewne wzorce

■ Singleton

- zwykle potrzeby jest jeden obiekt będący fasadą, dlatego fasady często są implementowane jako singletony

■ Abstract Factory

- wzorca Façade można użyć ze wzorcem AbstractFactory
- zapewniajemy w ten sposób interfejs do tworzenia określonej rodziny obiektów podsystemu

Façade - Podsumowanie

- Udostępnia interfejs pozwalający ukryć przed klientem złożoność podsystemu
- Sprzyja słabemu powiązaniu klientów z podsystemem

Bridge

Bridge

■ Przeznaczenie

- umożliwia różnicowanie implementacji i abstrakcji przez umieszczenie obu elementów w osobnych hierarchiach klas
- bardziej elastyczny sposób separacji abstrakcji od implementacji niż stosowanie dziedziczenia

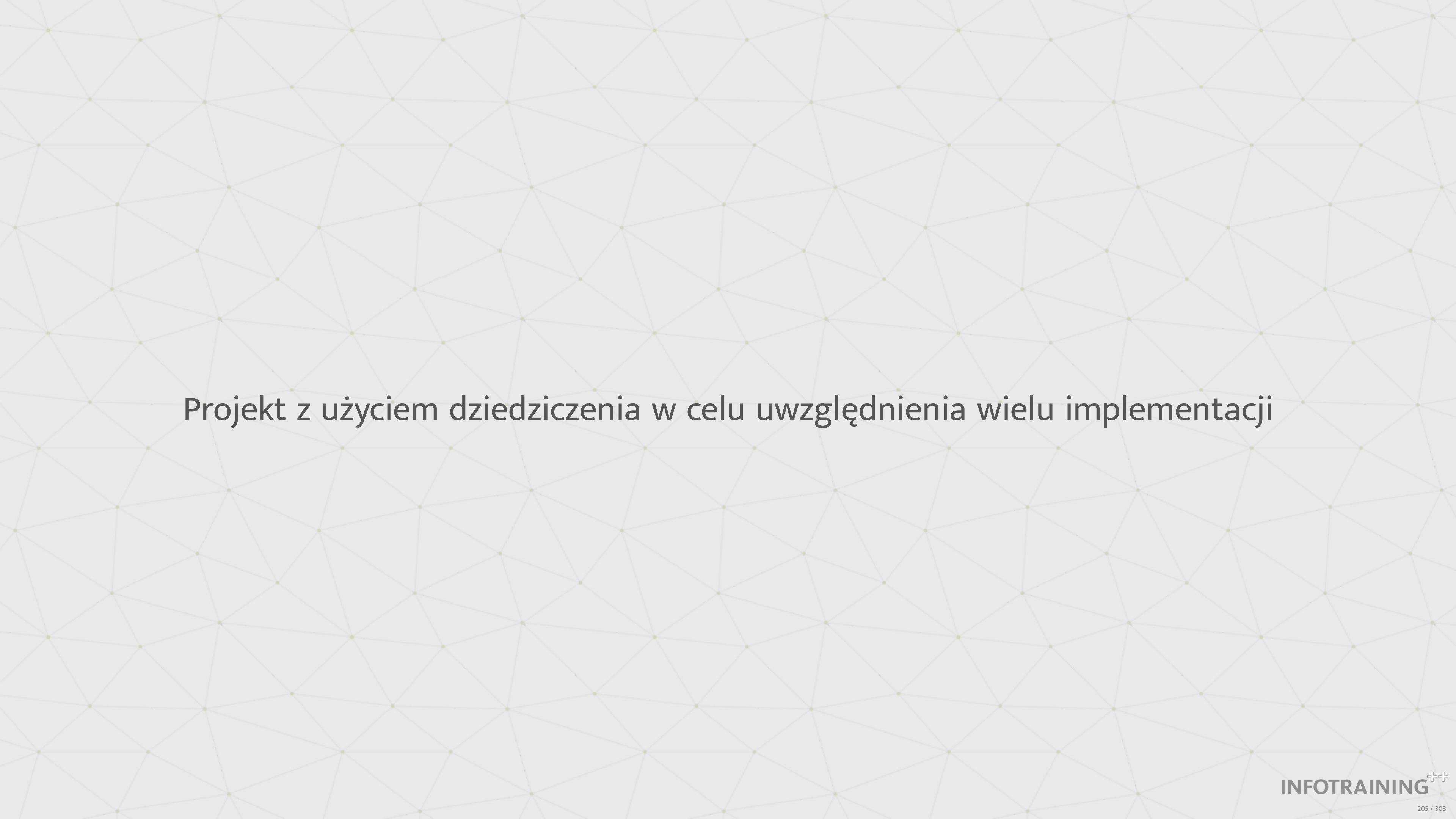
Bridge - Kontekst

- Istnieje wiele implementacji, które muszą być uwzględnione w projekcie
- Klient korzysta z abstrakcyjnych klas w celu ujednolicenia interfejsu

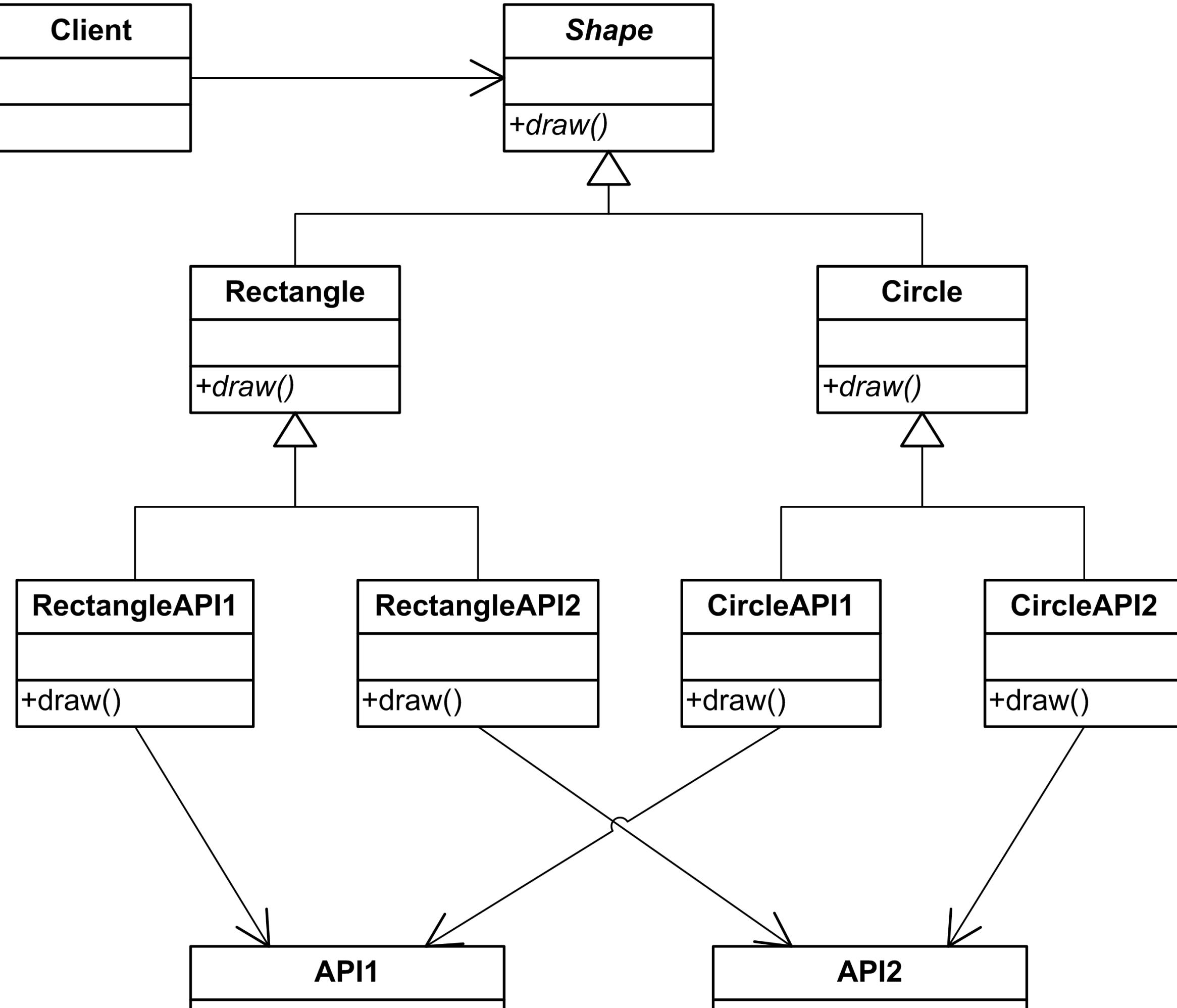
Bridge - Problem

- Chcemy uniknąć stałego powiązania abstrakcji z jej implementacją – implementacja może być wybierana lub zmieniana w czasie wykonywania programu
- Oczekujemy zmian zarówno w interfejsie abstrakcji jak i w implementacjach
 - zmiany w implementacji abstrakcji nie powinny mieć wpływu na klientów
- Chcemy całkowicie ukryć implementację abstrakcji przed klientami

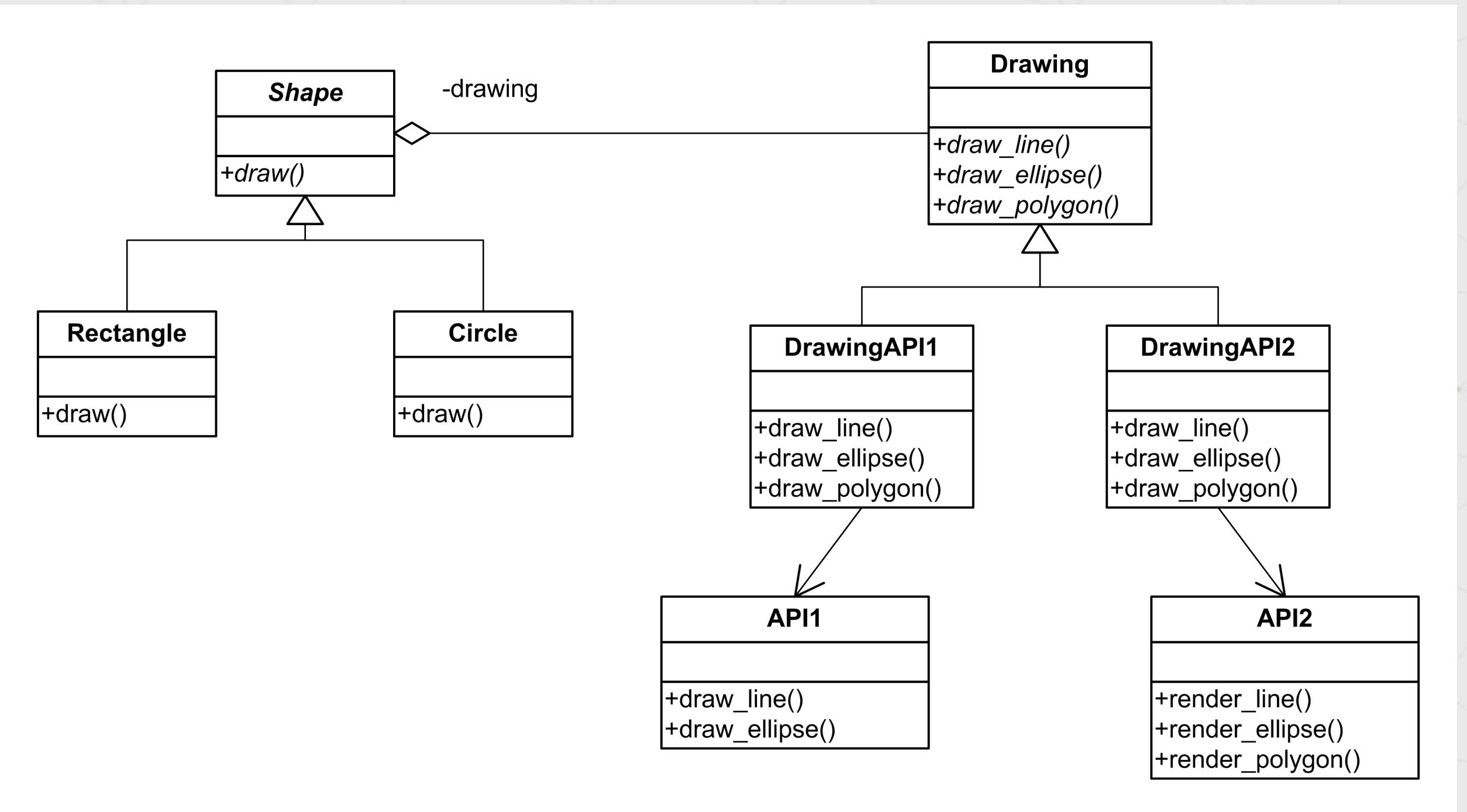
Bridge - Scenariusz



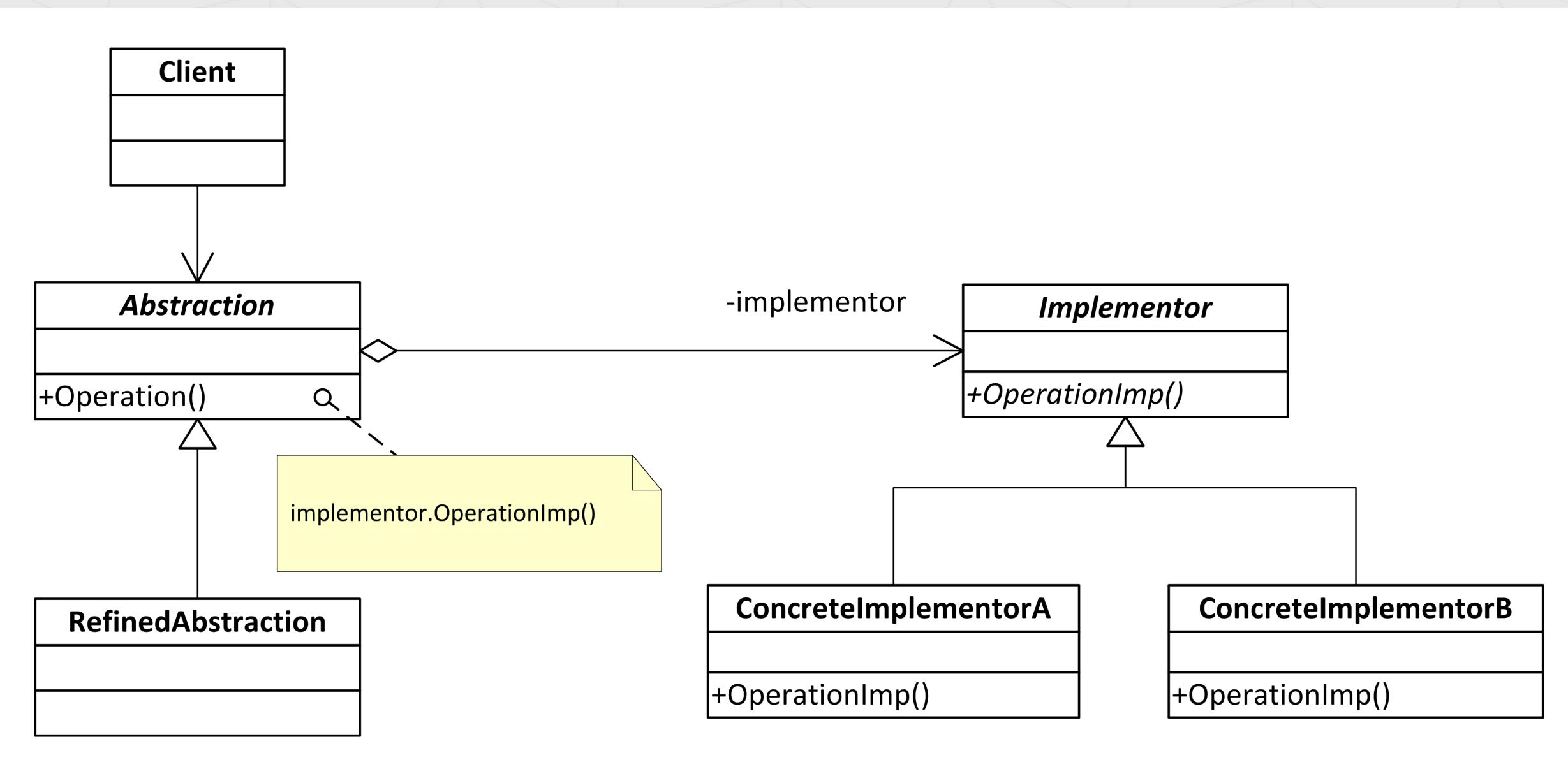
Projekt z użyciem dziedziczenia w celu uwzględnienia wielu implementacji



Projekt z użyciem wzorca Bridge



Bridge - Structure



Bridge - Konsekwencje

Bridge - Konsekwencje

- Separuje implementację, usuwając jej trwałe powiązanie z interfejsem
 - implementacja abstrakcji może być ustalana w czasie wykonywania programu
 - eliminuje zależność w czasie kompilacji od określonej implementacji

Bridge - Konsekwencje

- Separuje implementację, usuwając jej trwałe powiązanie z interfejsem
 - implementacja abstrakcji może być ustalana w czasie wykonywania programu
 - eliminuje zależność w czasie komplikacji od określonej implementacji
- Ułatwiona rozszerzalność – można niezależnie rozbudowywać hierarchie Abstrakcji i Implementacji
 - zmiany w klasach konkretnych abstrakcji nie wpływają na klienta

Bridge - Konsekwencje

- Separuje implementację, usuwając jej trwałe powiązanie z interfejsem
 - implementacja abstrakcji może być ustalana w czasie wykonywania programu
 - eliminuje zależność w czasie komplikacji od określonej implementacji
- Ułatwiona rozszerzalność – można niezależnie rozbudowywać hierarchie Abstrakcji i Implementacji
 - zmiany w klasach konkretnych abstrakcji nie wpływają na klienta
- Przydatny w systemach graficznych i okienkowych, które muszą pracować na różnych platformach

Bridge - Konsekwencje

- Separuje implementację, usuwając jej trwałe powiązanie z interfejsem
 - implementacja abstrakcji może być ustalana w czasie wykonywania programu
 - eliminuje zależność w czasie komplikacji od określonej implementacji
- Ułatwiona rozszerzalność – można niezależnie rozbudowywać hierarchie Abstrakcji i Implementacji
 - zmiany w klasach konkretnych abstrakcji nie wpływają na klienta
- Przydatny w systemach graficznych i okienkowych, które muszą pracować na różnych platformach
- Zwiększa złożoność projektu

Bridge - przypadek szczególny PIMPL

bitmap.hpp

```
01 class Bitmap
02 {
03     struct BitmapImpl;
04
05     std::unique_ptr<BitmapImpl> impl_;
06 public:
07     explicit Bitmap(const std::string& file_name);
08     ~Bitmap();
09     Bitmap(Bitmap&&) = default;
10     Bitmap& operator=(Bitmap&&) = default;
11     void draw() const;
12 };
```

Bridge - przypadek szczególny PIMPL

bitmap.hpp

```
01 class Bitmap
02 {
03     struct BitmapImpl;
04
05     std::unique_ptr<BitmapImpl> impl_;
06 public:
07     explicit Bitmap(const std::string& file_name);
08     ~Bitmap();
09     Bitmap(Bitmap&&) = default;
10     Bitmap& operator=(Bitmap&&) = default;
11     void draw() const;
12 };
```

forward declaration

Bridge - przypadek szczególny PIMPL

bitmap.hpp

```
01 class Bitmap
02 {
03     struct BitmapImpl;
04
05     std::unique_ptr<BitmapImpl> impl_;
06
07     explicit Bitmap(const std::string& file_name);
08     ~Bitmap();
09     Bitmap(Bitmap&&) = default;
10     Bitmap& operator=(Bitmap&&) = default;
11     void draw() const;
12 };
```

handle to an implementation

Bridge - przypadek szczególny PIMPL

bitmap.hpp

```
01 class Bitmap
02 {
03     struct BitmapImpl;
04
05     std::unique_ptr<BitmapImpl> impl_;
06 public:
07     explicit Bitmap(const std::string& file_name);
08     ~Bitmap();
09     Bitmap(Bitmap&&) = default;
10     Bitmap& operator=(Bitmap&&) = default;
11     void draw() const;
12 };
```

declaration of destructor

bitmap.cpp

```
01 struct Bitmap::BitmapImpl
02 {
03     std::vector<std::byte> bmp;
04 };
05
06 Bitmap::Bitmap(const std::string& file_name)
07     : impl_{std::make_unique<BitmapImpl>()}
08 {
09     impl_->bmp.reserve(1024);
10     //...
11 }
12
13 Bitmap::~Bitmap() = default;
14
15 void Bitmap::draw() const
16 {
17     for(const auto& pixel : impl_->bmp)
18     {
19         //...
20     }
21 }
```

bitmap.cpp

```
01 struct Bitmap::BitmapImpl
02 {
03     std::vector<std::byte> bmp;
04 };
05
06 Bitmap::Bitmap(const std::string& file_name)
07     : impl_{std::make_unique<BitmapImpl>()}
08 {
09     impl_->bmp.reserve(1024);
10     //...
11 }
12
13 Bitmap::~Bitmap() = default;
14
15 void Bitmap::draw() const
16 {
17     for(const auto& pixel : impl_->bmp)
18     {
19         //...
20     }
21 }
```

definition of implementation structure

bitmap.cpp

```
01 struct Bitmap::BitmapImpl
02 {
03     std::vector<std::byte> bmp;
04 };
05
06 Bitmap::Bitmap(const std::string& file_name)
07     : impl_{std::make_unique<BitmapImpl>()}
08 {
09     impl_->bmp.reserve(1024);
10     //...
11 }
12
13 Bitmap::~Bitmap() = default;
14
15 void Bitmap::draw() const
16 {
17     for(const auto& pixel : impl_->bmp)
18     {
19         //...
20     }
21 }
```

instantiation of implementor

bitmap.cpp

```
01 struct Bitmap::BitmapImpl
02 {
03     std::vector<std::byte> bmp;
04 };
05
06 Bitmap::Bitmap(const std::string& file_name)
07     : impl_{std::make_unique<BitmapImpl>()}
08 {
09     impl_->bmp.reserve(1024);
10     //...
11 }
12
13 Bitmap::~Bitmap() = default;
14
15 void Bitmap::draw() const
16 {
17     for(const auto& pixel : impl_->bmp)
18     {
19         //...
20     }
21 }
```

definition of destructor

bitmap.cpp

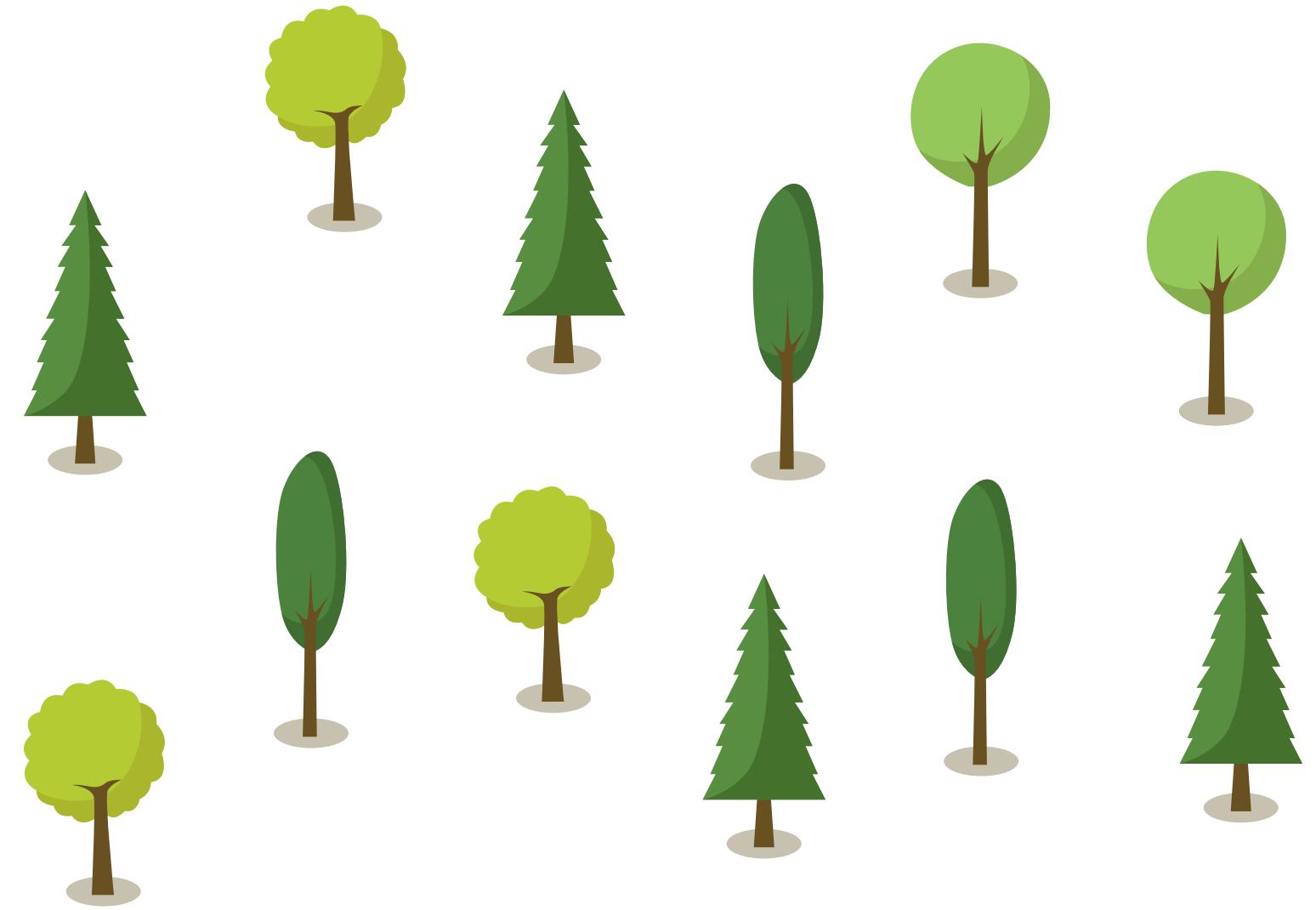
```
01 struct Bitmap::BitmapImpl
02 {
03     std::vector<std::byte> bmp;
04 };
05
06 Bitmap::Bitmap(const std::string& file_name)
07     : impl_{std::make_unique<BitmapImpl>()}
08 {
09     impl_->bmp.reserve(1024);
10     //...
11 }
12
13 Bitmap::~Bitmap() = default;
14
15 void Bitmap::draw() const
16 {
17     for(const auto& pixel : impl_->bmp)
18     {
19         //...
20     }
21 }
```

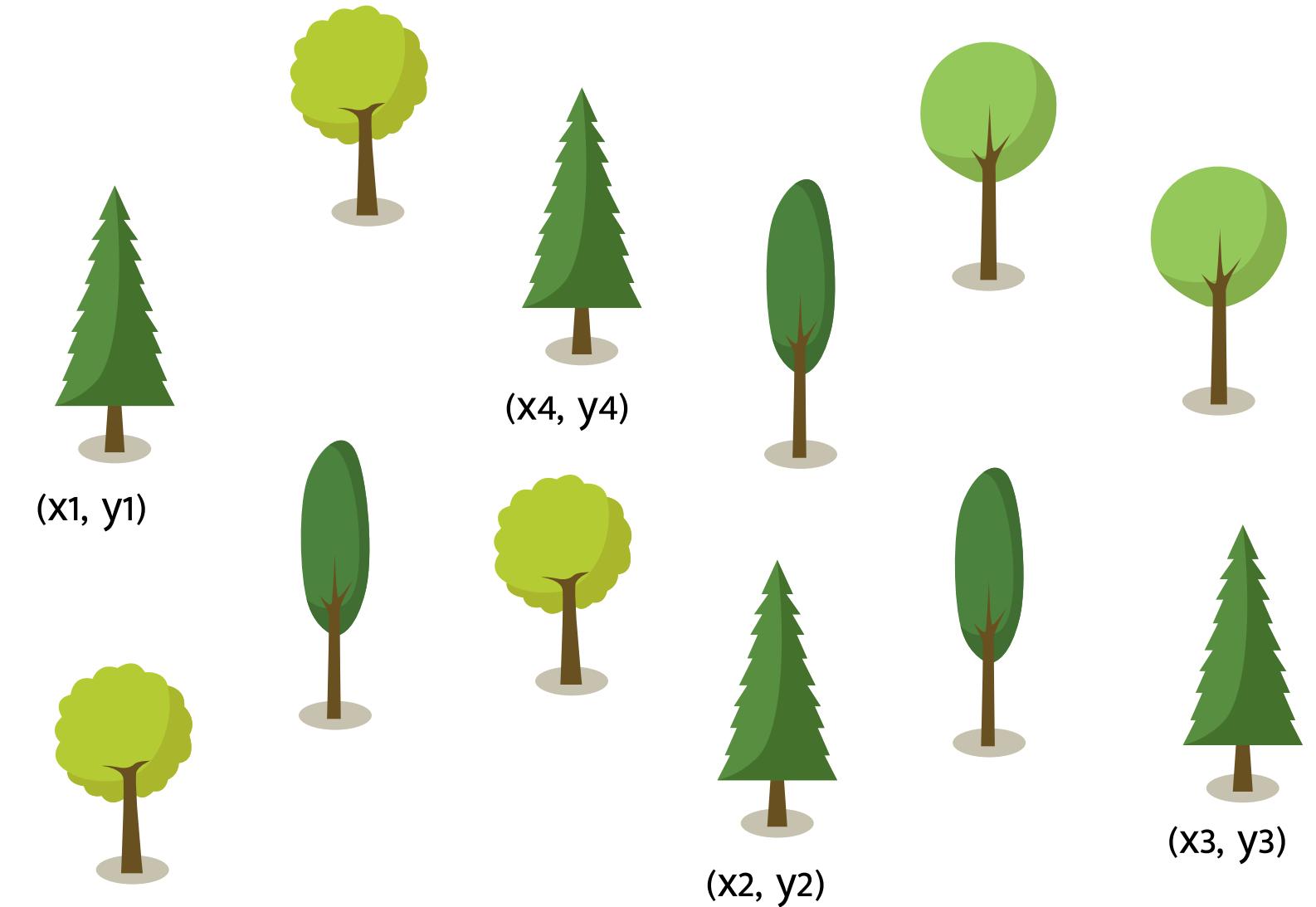
usage of implementor

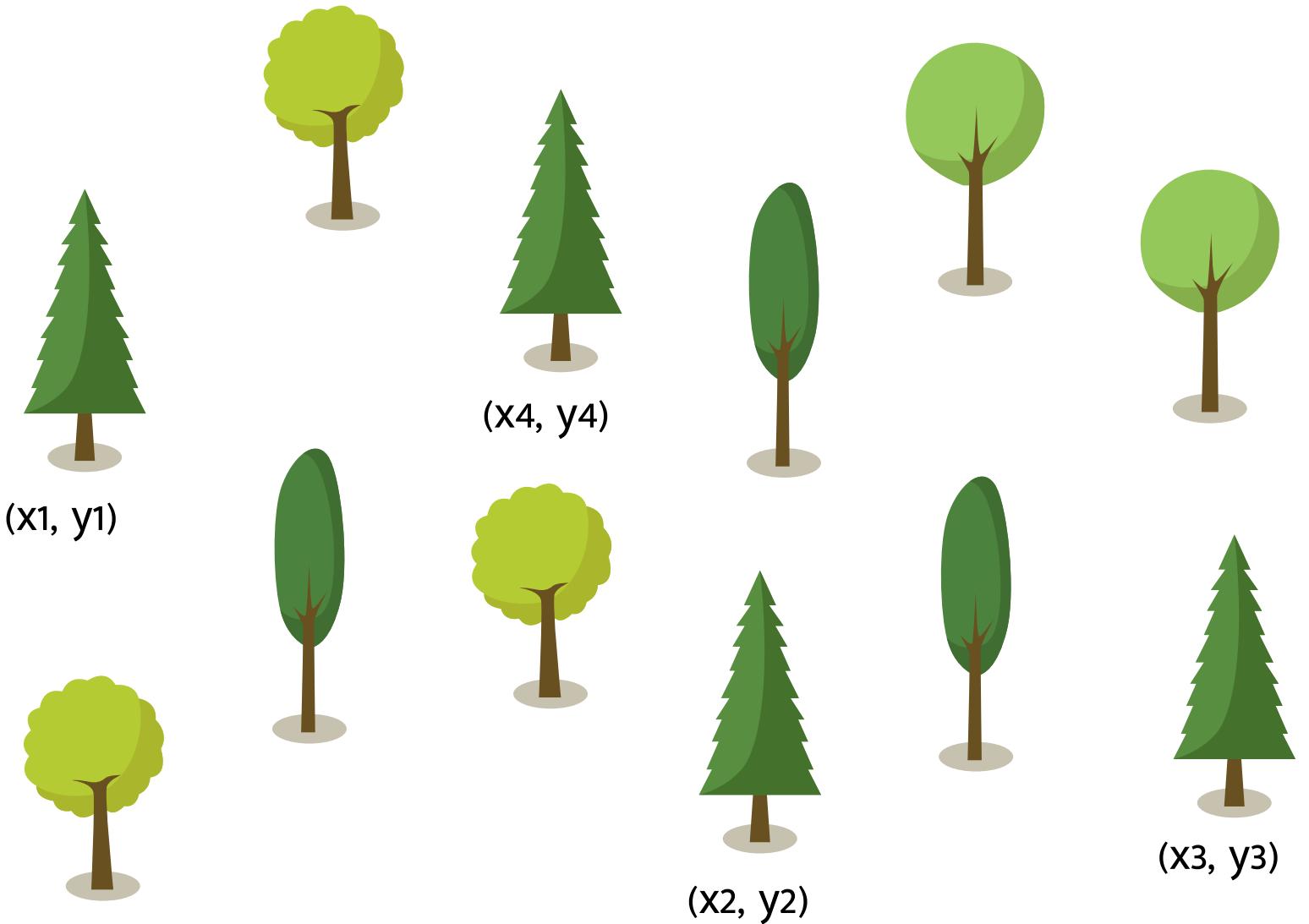
Flyweight

Flyweight

Flyweight - Scenariusz



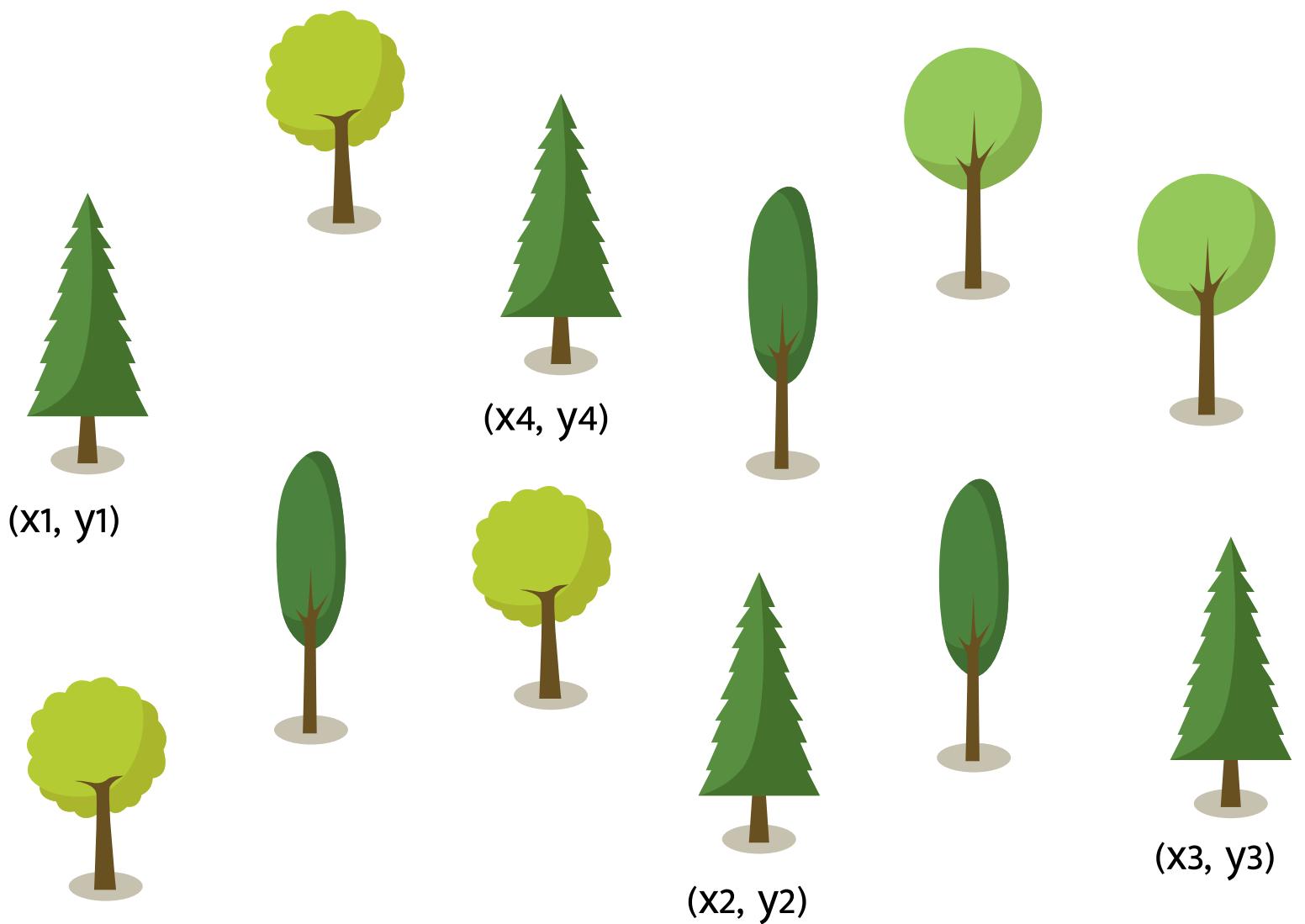




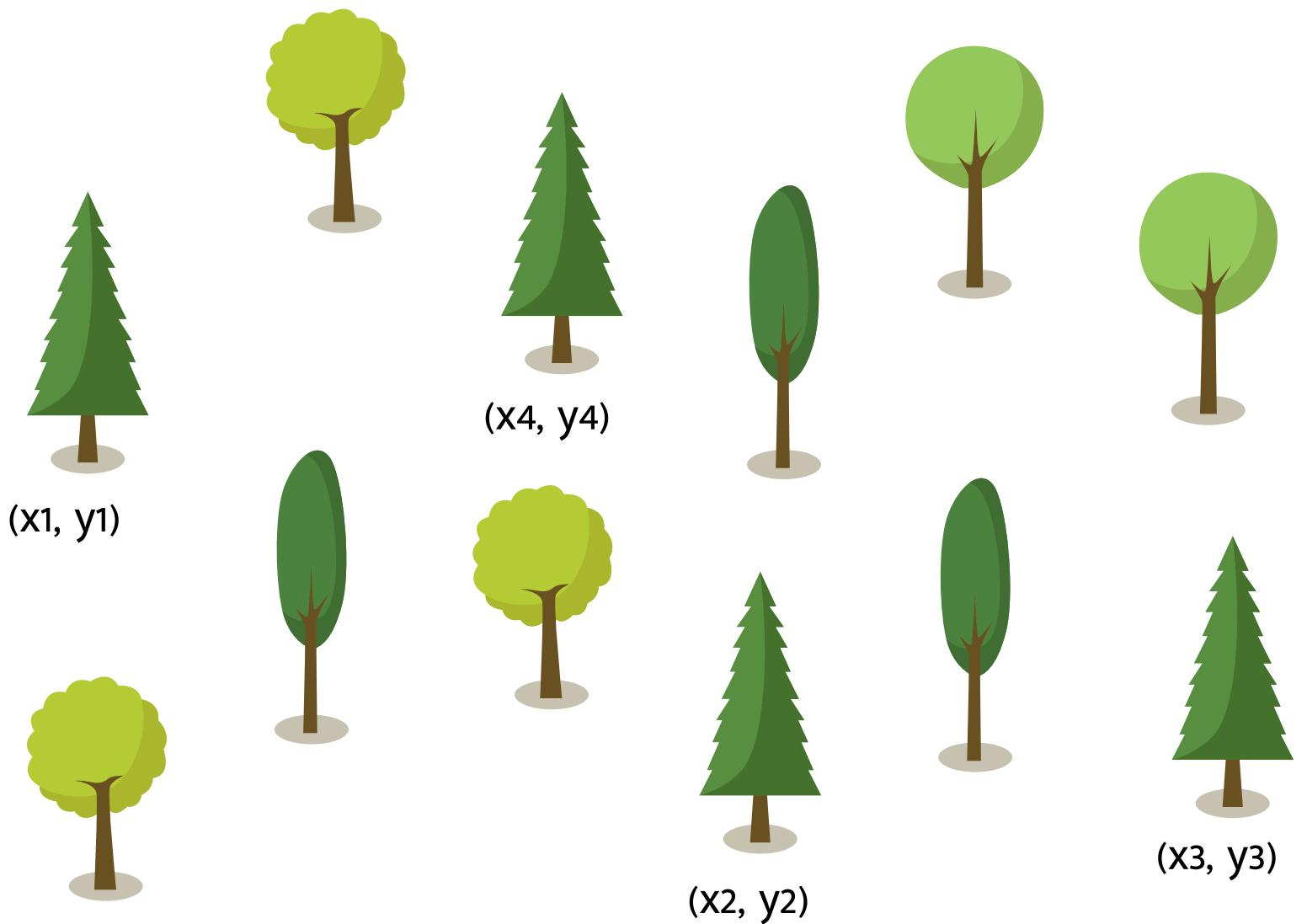
TreeSprite

-position
-bitmap

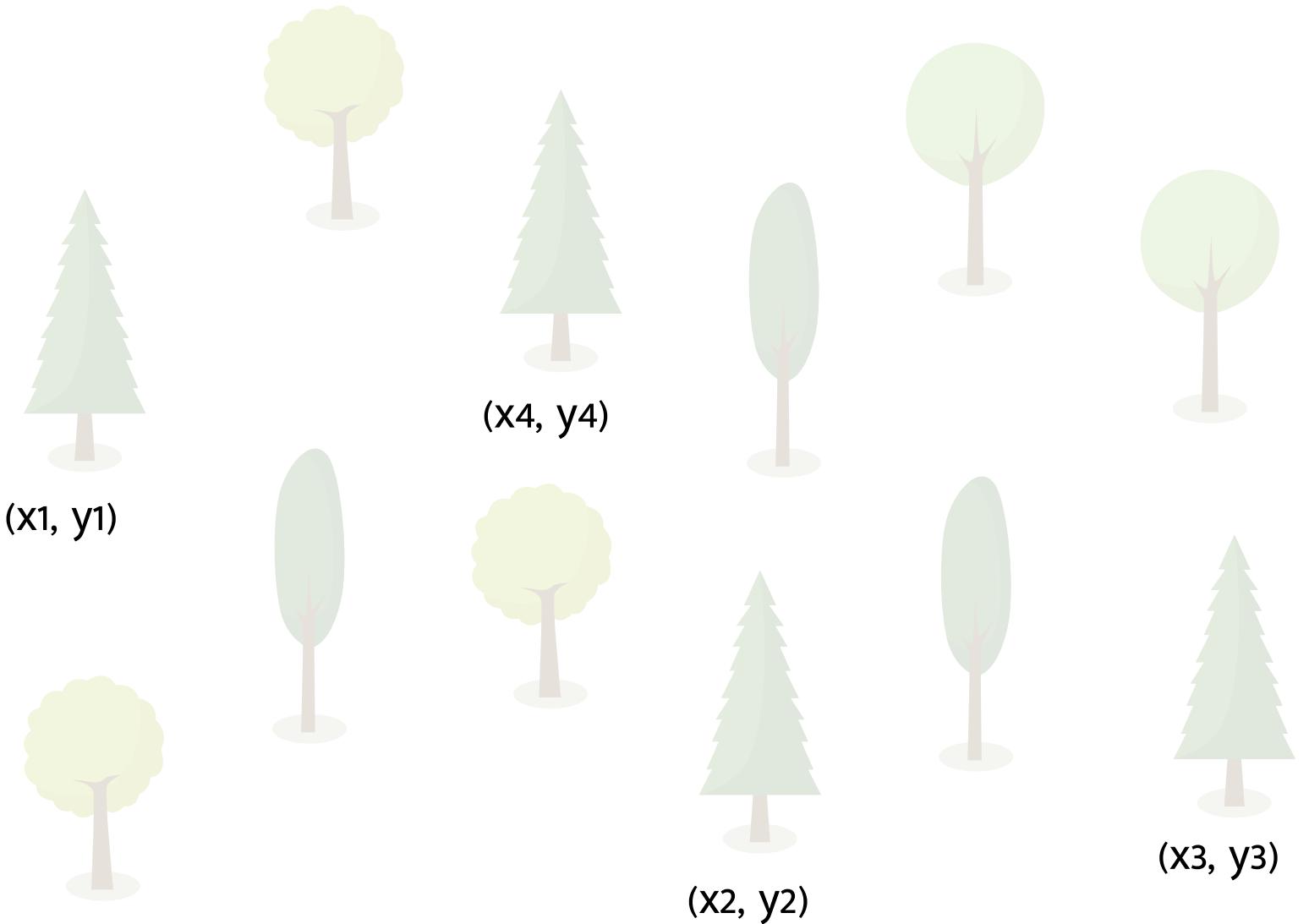
+render()



TreeSprite
-position
-bitmap
+render()

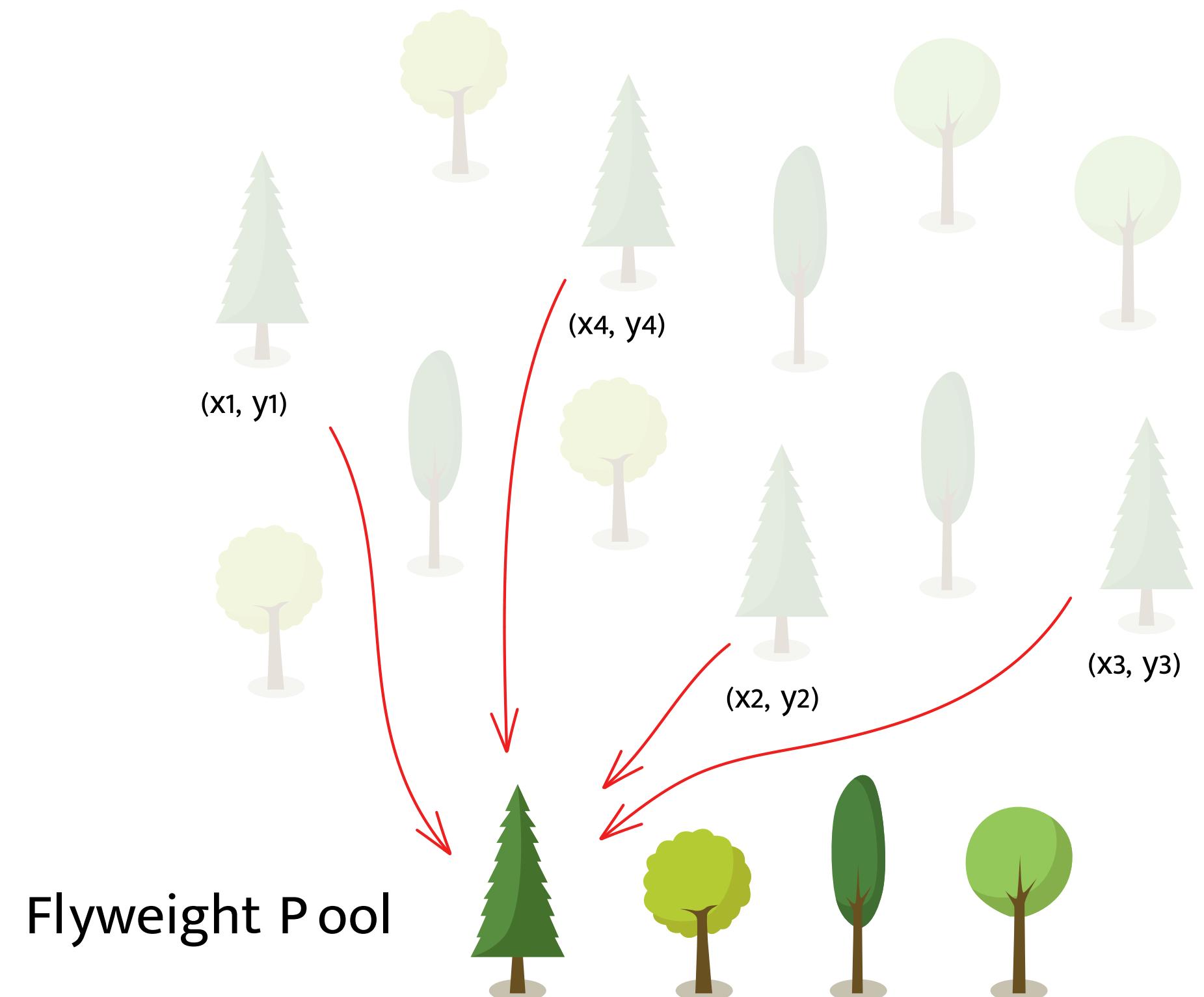


TreeSprite
-bitmap
+render(position: x, y)



Flyweight Pool





Flyweight Pool

Flyweight - Kontekst / Problem

■ Kontekst

- w projekcie istnieje olbrzymia liczba obiektów
- koszt związany z przechowywaniem tych obiektów w pamięci jest znaczący

■ Problem

- chcemy ograniczyć koszty, związane z przechowywaniem obiektów w pamięci współdzieląc obiekty w postaci pyłków

Obiekt flyweight

■ Obiekt Flyweight (Pyłek)

- współdzielony obiekt, który może być używany jednocześnie w wielu kontekstach
- działa jako obiekt niezależny w każdym kontekście dzięki rozróżnieniu stanu na **wewnętrzny i zewnętrzny**

Obiekt flyweight

Obiekt flyweight

- Stan wewnętrzny – jest przechowywany w pyłku, składa się z informacji, które są niezależne od kontekstu

Obiekt flyweight

- Stan wewnętrzny – jest przechowywany w pyłku, składa się z informacji, które są niezależne od kontekstu
- Stan zewnętrzny – zależy od kontekstu i zmienia się w zależności od niego, nie może być współdzielony

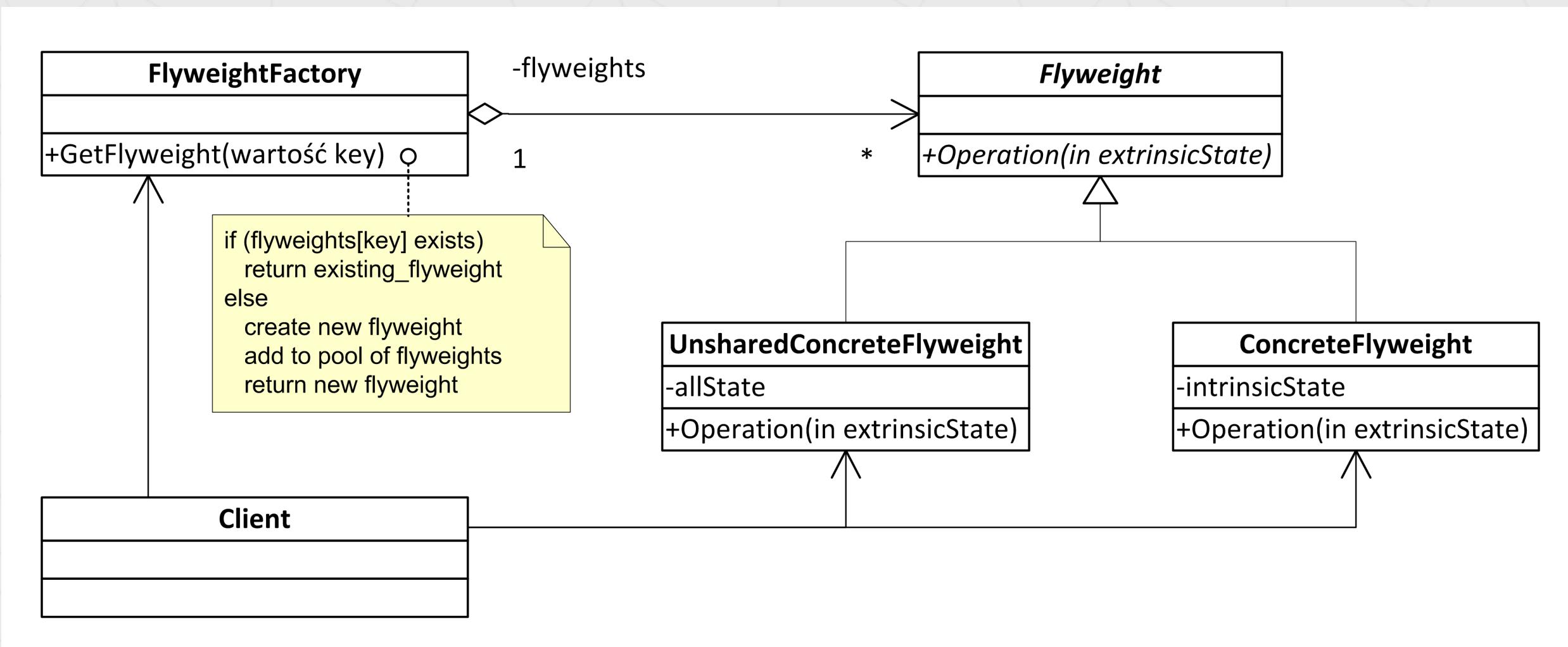
Obiekt flyweight

- Stan wewnętrzny – jest przechowywany w pyłku, składa się z informacji, które są niezależne od kontekstu
- Stan zewnętrzny – zależy od kontekstu i zmienia się w zależności od niego, nie może być współdzielony
- Pyłki modelują pojęcia lub byty, które zwykle są zbyt liczne, żeby przedstawiać je za pomocą w pełni samodzielnych obiektów

Flyweight - wpółdzielenie obiektów

- Po usunięciu stanu zewnętrznego wiele grup obiektów można zastąpić stosunkowo niewielką liczbą współdzielonych obiektów

Flyweight - Struktura



Flyweight - Konsekwencje

- Zmniejszenie zużycia pamięci kosztem wydłużenia czasu wykonywania
- Oszczędności pamięci zależą od kilku czynników:
 - zmniejszenia łącznej liczby egzemplarzy, wynikającego ze współdzielenia
 - wielkości stanu wewnętrznego przypadającego na obiekt
 - tego, czy stan zewnętrzny jest wyliczany, czy przechowywany

Flyweight - Podsumowanie

- Wykorzystuje współdzielenie obiektów w celu efektywnej obsługi wielkiej liczby obiektów
- Zmniejsza zużycie pamięci kosztem wydłużenia czasu wykonywania
- Skuteczność pyłku zależy od tego, ile informacji uda się współdzielić jako stan wewnętrzny pyłku

Wzorce behawioralne

■ Template Method

- Template Method
- Strategy

- Template Method
- Strategy
- State

- Template Method
- Strategy
- State
- Chain Of Responsibility

- Template Method
- Strategy
- State
- Chain Of Responsibility
- Observer

- Template Method
- Strategy
- State
- Chain Of Responsibility
- Observer
- Command

- Template Method
- Strategy
- State
- Chain Of Responsibility
- Observer
- Command
- Memento

- Template Method
- Strategy
- State
- Chain Of Responsibility
- Observer
- Command
- Memento
- Mediator

- Template Method
- Strategy
- State
- Chain Of Responsibility
- Observer
- Command
- Memento
- Mediator
- Visitor

- Dotyczą algorytmów i przydzielania zobowiązań obiektom

- Dotyczą algorytmów i przydzielania zobowiązań obiektom
- Charakteryzują złożone przepływy sterowania między obiektyami, które są trudne do prześledzenia w czasie wykonywania programu

- Dotyczą algorytmów i przydzielania zobowiązań obiektom
- Charakteryzują złożone przepływy sterowania między obiektyami, które są trudne do prześledzenia w czasie wykonywania programu
- Są wykorzystywane do organizowania, zarządzania i łączenia zachowań

Template Method

Template Method

- Przeznaczenie

Template Method

■ Przeznaczenie

- definiuje szkielet algorytmu, odkładając implementację niektórych kroków algorytmu do klas pochodnych

Template Method

■ Przeznaczenie

- definiuje szkielet algorytmu, odkładając implementację niektórych kroków algorytmu do klas pochodnych
- metoda szablonowa ustala kolejność kroków w algorytmie, ale umożliwia klasom pochodnym zmianę implementacji tych kroków zależnie od ich potrzeb

Template Method - Kontekst / Problem

Template Method - Kontekst / Problem

■ Kontekst

- istnieje algorytm wymagający zmiany implementacji poszczególnych kroków

Template Method - Kontekst / Problem

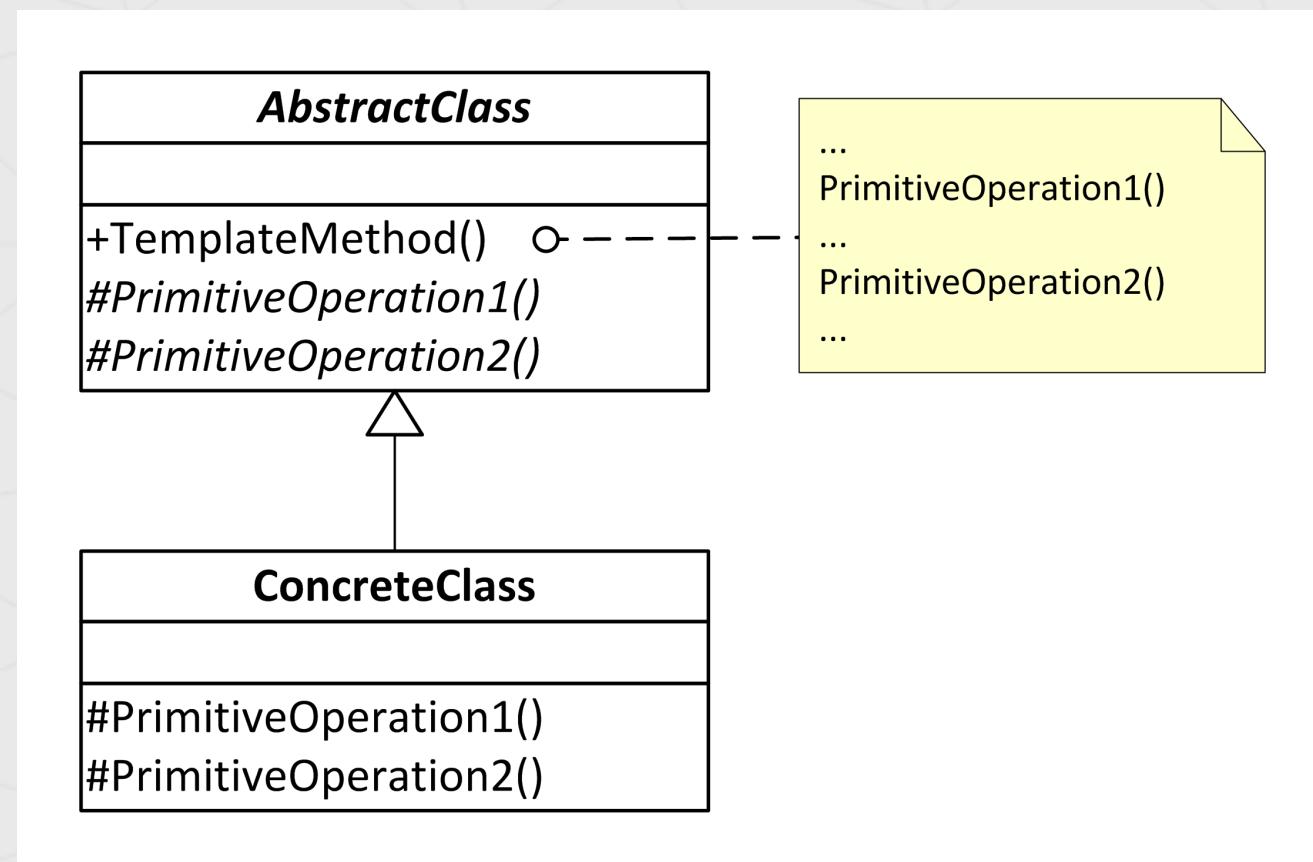
■ Kontekst

- istnieje algorytm wymagający zmiany implementacji poszczególnych kroków

■ Problem

- chcemy jednorazowo zaimplementować stałą część algorytmu i pozostawić klasom pochodnym zaimplementowanie zachowania, które może się zmieniać

Template Method - Struktura



Template Method - Konsekwencje

- Użycie metod szablonowych jest podstawową techniką stosowaną w celu zagwarantowania możliwości ponownego wykorzystania kodu
- Metody szablonowe prowadzą do odwróconej struktury sterowania – klasa bazowa wywołuje operacje klasy pochodnej, a nie odwrotnie

Template Method - Konsekwencje

Metody szablonowe wywołują:

Template Method - Konsekwencje

Metody szablonowe wywołują:

- operacje konkretne z `ConcreteClass`, `AbstractClass` lub z klas klienta

Template Method - Konsekwencje

Metody szablonowe wywołują:

- operacje konkretne z `ConcreteClass`, `AbstractClass` lub z klas klienta
- operacje abstrakcyjne

Template Method - Konsekwencje

Metody szablonowe wywołują:

- operacje konkretne z `ConcreteClass`, `AbstractClass` lub z klas klienta
- operacje abstrakcyjne
- metody wytwarzcze

Template Method - Konsekwencje

Metody szablonowe wywołują:

- operacje konkretne z `ConcreteClass`, `AbstractClass` lub z klas klienta
- operacje abstrakcyjne
- metody wytwarzające
- *hook methods*, zapewniające zachowanie domyślne, które może być rozszerzane przez klasy pochodne

Template Method - Pokrewne wzorce

■ Strategy

- zarówno wzorzec Strategy jak i Template Method dokonują hermetyzacji algorytmów odpowiednio wykorzystując kompozycję i dziedziczenie

■ Factory Method

- Template Method często wywołuje metody wytwarzacze

Template Method

- Definiuje szkielet danego algorytmu w określonej metodzie, przekazując realizację niektórych kroków algorytmu do klas podrzędnych
- Pozwala klasom podrzędnym na redefiniowanie pewnych kroków algorytmu, ale jednocześnie uniemożliwia zmianę jego struktury
- Spełnia tzw. Regułę Hollywood
 - proces podejmowania decyzji powinien być umieszczony w modułach wysokiego poziomu , które mogą samodzielnie decydować, jak i kiedy wywoływać moduły niskiego poziomu

Strategy

Strategy

■ Przeznaczenie

- definiuje rodziny algorytmów
- dokonuje ich hermetyzacji i sprawia, że stają się one wymienne
- pozwala na modyfikację danego algorytmu niezależnie od klienta, który tego algorytmu używa

Strategy - Kontekst / Problem

Strategy - Kontekst / Problem

■ Kontekst

- potrzebne są różne warianty jakiegoś algorytmu
- wiele powiązanych ze sobą klas różni się tylko zachowaniem
- w algorytmie są używane dane, o których klient nie powinien wiedzieć
- klasa definiuje wiele zachowań, które w operacjach są uwzględnione w postaci wielokrotnych instrukcji warunkowych

Strategy - Kontekst / Problem

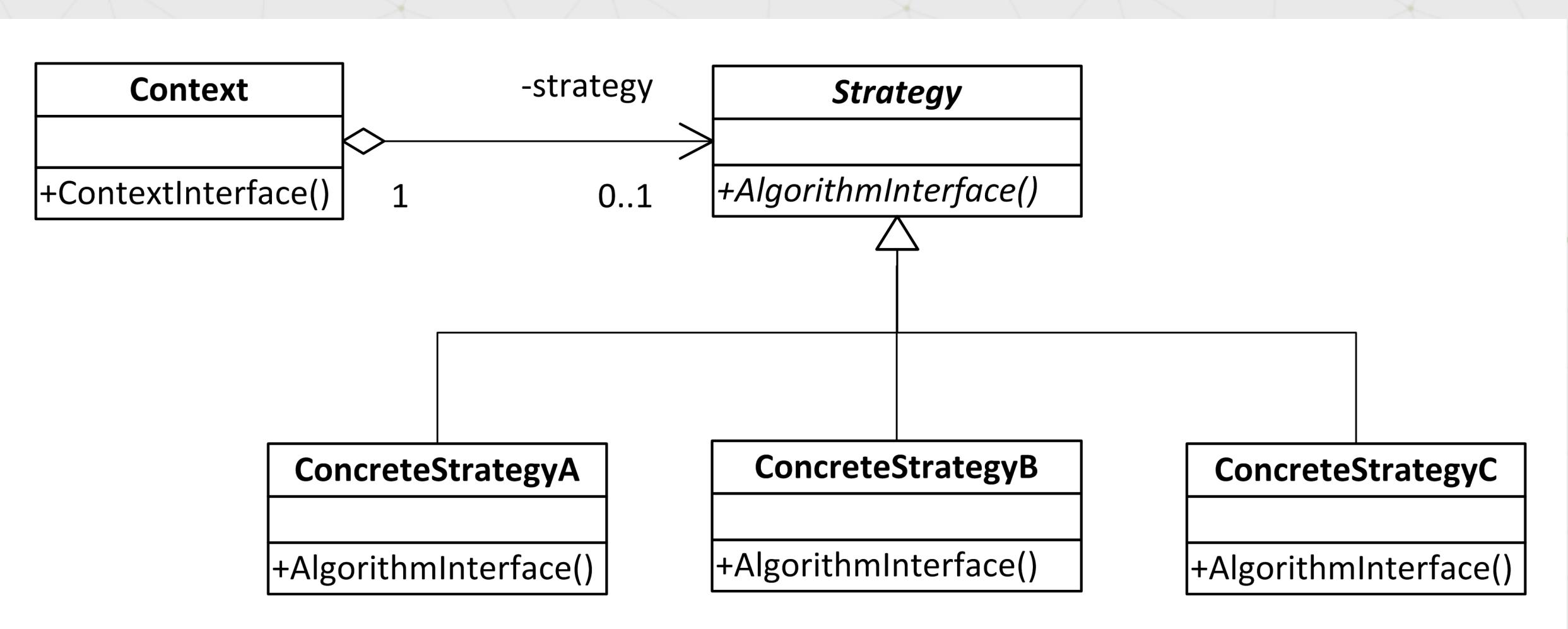
■ Kontekst

- potrzebne są różne warianty jakiegoś algorytmu
- wiele powiązanych ze sobą klas różni się tylko zachowaniem
- w algorytmie są używane dane, o których klient nie powinien wiedzieć
- klasa definiuje wiele zachowań, które w operacjach są uwzględnione w postaci wielokrotnych instrukcji warunkowych

■ Problem

- chcemy dokonać hermetyzacji algorytmu w klasie i stosując kompozycję umożliwić wymianę implementacji algorytmu

Strategy - Struktura



Strategy - Konsekwencje

Strategy - Konsekwencje

- Enkapsulacja algorytmu w oddzielnych klasach umożliwia modyfikację jego implementacji niezależnie od kontekstu

Strategy - Konsekwencje

- Enkapsulacja algorytmu w oddzielnych klasach umożliwia modyfikację jego implementacji niezależnie od kontekstu
- Strategie eliminują instrukcje warunkowe
 - alternatywa dla stosowania instrukcji warunkowych w celu wybrania pożądanego zachowania

Strategy - Konsekwencje

- Enkapsulacja algorytmu w oddzielnych klasach umożliwia modyfikację jego implementacji niezależnie od kontekstu
- Strategie eliminują instrukcje warunkowe
 - alternatywa dla stosowania instrukcji warunkowych w celu wybrania pożądanego zachowania
- Klienci muszą być świadomi istnienia różnych strategii i różnic pomiędzy nimi - potencjalna wada

Strategy - Konsekwencje

- Enkapsulacja algorytmu w oddzielnych klasach umożliwia modyfikację jego implementacji niezależnie od kontekstu
- Strategie eliminują instrukcje warunkowe
 - alternatywa dla stosowania instrukcji warunkowych w celu wybrania pożądanego zachowania
- Klienci muszą być świadomi istnienia różnych strategii i różnic pomiędzy nimi - potencjalna wada
- Zwiększoną liczbą obiektów

Strategy - Implementacj (1)

- Definiowanie interfejsów strategii (Strategy) i kontekstu (Context)
 - Context przesyła dane do operacji Strategy w argumentach
 - Context przesyła siebie samego jako argument, a strategia jawnie żąda danych od kontekstu
- Klasa kontekstu jest prostsza w użyciu, jeżeli użycie jej bez obiektu strategii ma sens
 - implementacja strategii domyślnej – klienci nie muszą zajmować się obiektem strategii

Strategy - Implementacj (2)

- Jeśli strategie są prostymi metodami, można zrezygnować z enkapsulacji w postaci odrębnej klasy na rzecz `std::function`

```
01  class Context
02  {
03      std::function<Result (Args)> strategy_;
04  public:
05      Context(std::function<Result (Args)> strategy)
06          : strategy_{strategy}
07      {}
08
09      void run()
10      {
11          auto result = strategy_();
12          //...
13      }
14  };
```

Strategy - Podsumowanie

- Strategia hermetyzuje algorytm w postaci obiektu
- Program wykorzystujący wzorzec Strategy może oferować wiele wersji algorytmu lub zachowania
- Zachowanie obiektów może się dynamicznie zmieniać w czasie wykonywania programu
- Delegowanie zachowania do oddzielnego obiektu z określonym interfejsem prowadzi do powstania klas o wysokiej kohezji
 - lepsza zgodność z SRP

State

State

Umożliwia obiektowi zmianę zachowania, gdy zmienia się jego stan wewnętrzny

State - Kontekst / Problem

State - Kontekst / Problem

■ Kontekst

- obiekt musi zmieniać swoje zachowanie w czasie wykonywania programu w zależności od stanu
- operacje zawierają duże, wieloczęściowe instrukcje warunkowe, które zależą od stanu obiektu - wzorzec State przenosi każde rozgałęzienie warunkowe do oddzielnej klasy

State - Kontekst / Problem

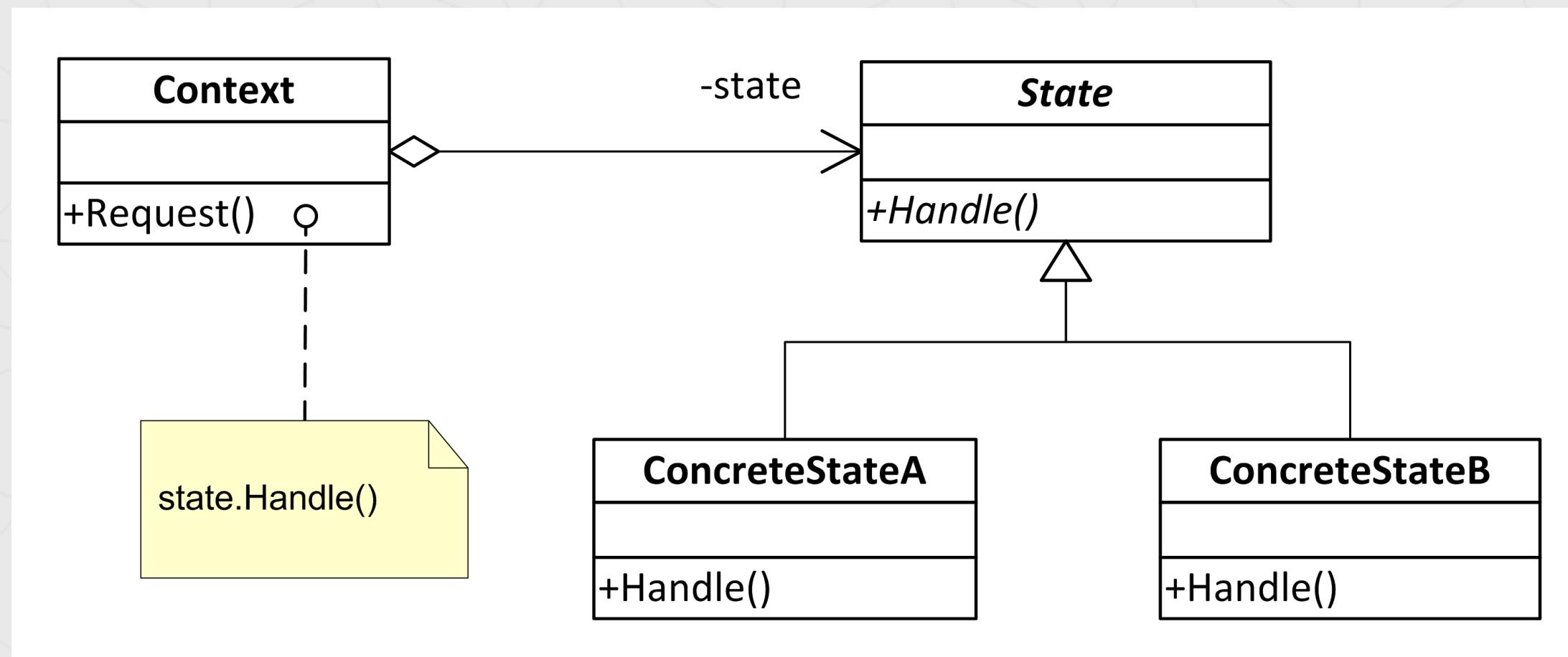
■ Kontekst

- obiekt musi zmieniać swoje zachowanie w czasie wykonywania programu w zależności od stanu
- operacje zawierają duże, wieloczęściowe instrukcje warunkowe, które zależą od stanu obiektu - wzorzec State przenosi każde rozgałęzienie warunkowe do oddzielnej klasy

■ Problem

- chcemy umożliwić obiektyowi zmianę zachowania w momencie zmiany wewnętrznego stanu obiektu hermetyzując stan w postaci klasy

State - Struktura



State - Konsekwencje

- Umiejscowienie zachowania specyficznego dla stanu i rozdzielenie zachowania w wypadku różnych stanów
 - kod dla każdego stanu znajduje się w osobnej klasie – ułatwia to dodawanie nowych stanów (nie wymaga daleko idących modyfikacji istniejącego kodu)
- Eliminuje konieczność dzielenia kodu metod na bloki właściwe dla poszczególnych stanów (bloki switch)
- Jawność przejść między stanami
 - z perspektywy kontekstu przejścia między stanami są atomowe – dochodzi do nich poprzez wymianę pojedynczego obiektu stanu

State - Implementacja

- Możliwość współdzielenia obiektów typu State
 - jeśli obiekty typu State są *immutable* to konteksty mogą je współdzielić
- Który z uczestników definiuje przejścia między stanami?

State - Pokrewne wzorce

- **Strategy**
 - podobny schemat UML
 - o zmianie zachowanie zawsze decyduje klient
- **Flyweight** – stosowany gdy można współdzielić obiekty reprezentujące stan

State - Podsumowanie

- Wzorzec projektowy State opisuje sytuację, w której zachowanie obiektu jest determinowane przez wewnętrzny stan, który może się zmieniać w reakcji na zachodzące zdarzenia
- Zapewnia lepszą skalowalność logiki zawiązanej z zarządzaniem stanów obiektu
- Poprzez hermetyzację stanów w klasach lokalizujemy przyszłe zmiany w kodzie

Chain Of Responsibility

Chain Of Responsibility

Chain Of Responsibility

- Umożliwia uniknięcie związania wysyłającego żądanie z odbiorcą żądania przez danie więcej niż jednemu obiekowi szansy obsłużenia tego żądania

Chain Of Responsibility

- Umożliwia uniknięcie związania wysyłającego żądanego z odbiorcą żądania przez danie więcej niż jednemu obiektowi szansy obsłużenia tego żądania
- Żądanie jest przesyłane wzdłuż łańcucha obiektów, aż któryś je obsłuży

Chain Of Responsibility

- Umożliwia uniknięcie związania wysyłającego żądanie z odbiorcą żądania przez danie więcej niż jednemu obiekowi szansy obsłużenia tego żądania
- Żądanie jest przesyłane wzdłuż łańcucha obiektów, aż któryś je obsłuży
- Obiekt, który wygenerował żądanie, nie wie, kto je obsłuży - żądanie ma **niejawnego odbiorcę**

Chain Of Responsibility

- Umożliwia uniknięcie związania wysyłającego żądanie z odbiorcą żądania przez danie więcej niż jednemu obiekowi szansy obsłużenia tego żądania
- Żądanie jest przesyłane wzdłuż łańcucha obiektów, aż któryś je obsłuży
- Obiekt, który wygenerował żądanie, nie wie, kto je obsłuży - żądanie ma **niejawnego odbiorcę**
- Aby przekazywać żadania wzdłuż łańcucha i zagwarantować, że odbiorcy pozostaną niejawni, każdy obiekt w łańcuchu korzysta ze wspólnego interfejsu obsługi żądań i uzyskiwania dostępu do następnika w łańcuchu

Chain of Responsibility - Kontekst / Problem

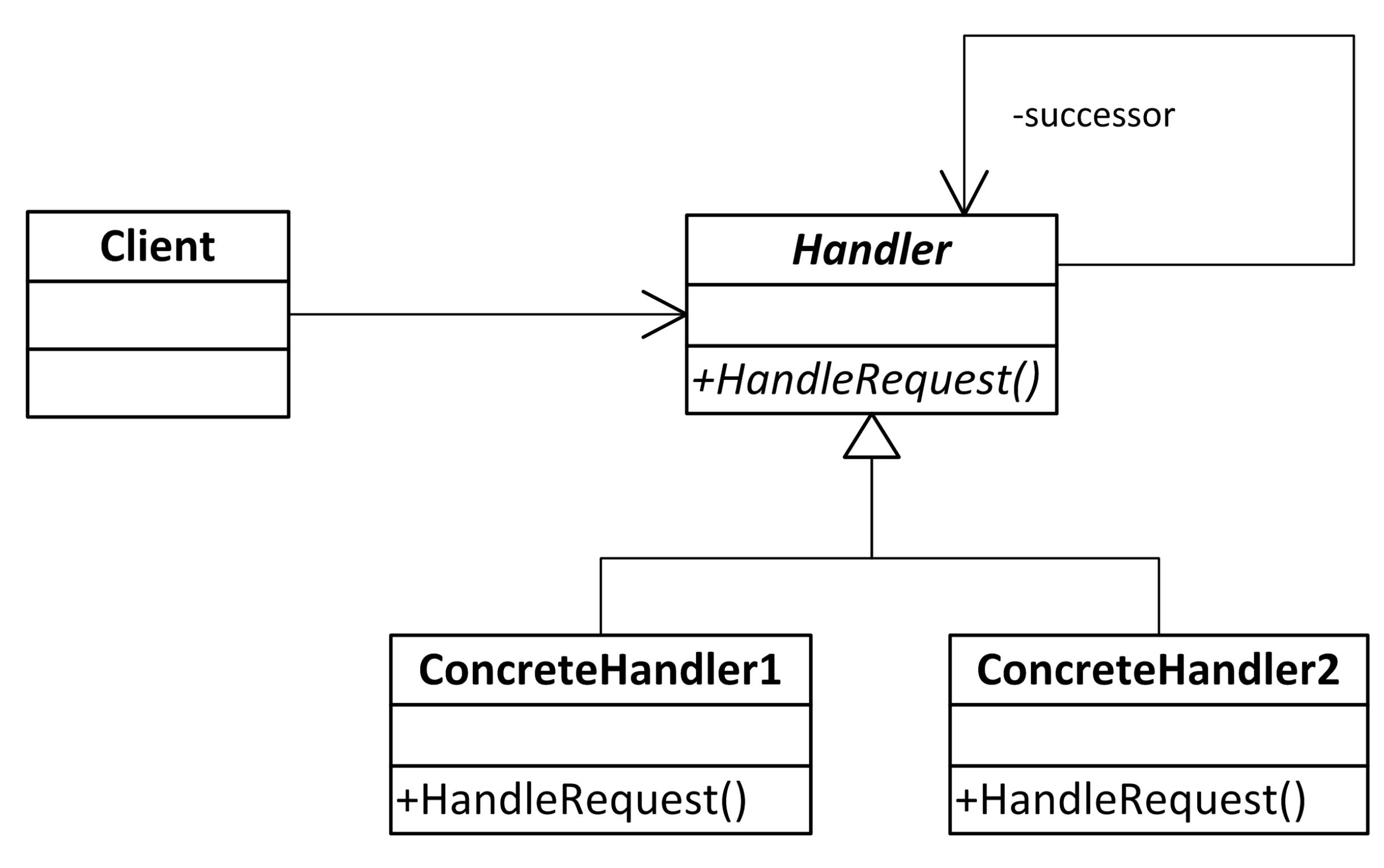
■ Kontekst

- więcej niż jeden obiekt może obsłużyć żądanie, a obiekt obsługujący nie jest znany a priori
- wykonanie żądania nie jest gwarantowane
- zbiór obiektów, które mogą obsłużyć żądanie, może być określony dynamicznie

■ Problem

- chcemy wysłać żądanie do jednego z kilku obiektów, nie określając jawnie odbiorcy
- chcemy odseparować nadawcę żądania od jego odbiorców

Chain of Responsibility - Struktura



Chain of Responsibility - Konsekwencje

- Zredukowanie powiązań
 - dany obiekt nie musi wiedzieć, który inny obiekt obsłuży żądanie
- Dodatkowa elastyczność w przydzielaniu obiektom zobowiązań
- Brak gwarancji odebrania żądania
 - ponieważ odbiorca żądania nie jest jawnie znany, nie ma gwarancji, że zostanie ono obsłużone – żądanie może wypaść z łańcucha, nie zostawiając w ogóle obsłużone

Chain of Responsibility - Implementacja

- Implementacja łańcucha następników:
 - zdefiniowanie nowych powiązań
 - użycie istniejących powiązań

Chain of Responsibility - Pokrewne wzorce

■ Composite

- często stosowany w połączeniu z Chain of Responsibility
- rodzic komponentu może odgrywać rolę jego następnika

Chain of Responsibility - Podsumowanie

Chain of Responsibility - Podsumowanie

- Separuje nadawcę żądania od jego odbiorców

Chain of Responsibility - Podsumowanie

- Separuje nadawcę żądania od jego odbiorców
- Wykonanie żądania nie jest gwarantowane

Chain of Responsibility - Podsumowanie

- Separuje nadawcę żądania od jego odbiorców
- Wykonanie żądania nie jest gwarantowane
- Powszechnie używany w systemach okienkowych do obsługi zdarzeń
 - np. kliknięcie myszy, wcisnięcie klawisza

Observer

Observer

- Określa zależność jeden-do-wiele między obiektami
- Gdy jeden obiekt zmienia stan, wszystkie obiekty od niego zależne są o tym automatycznie powiadamiane i uaktualniane

Observer - Kontekst

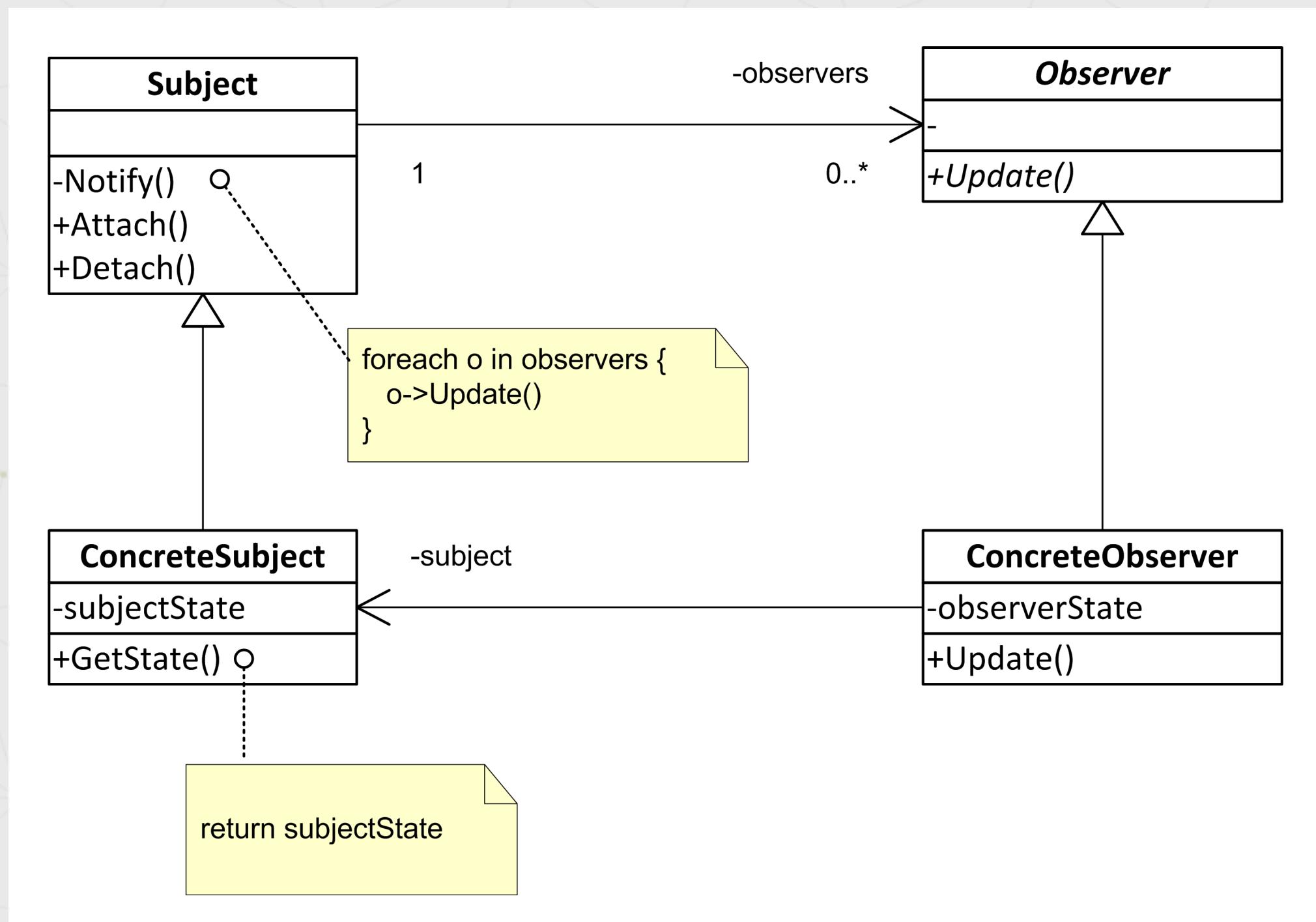
■ Kontekst

- zmiana stanu jednego obiektu wymaga zmiany innych i nie wiadomo, ile obiektów trzeba zmienić

■ Problem

- obiekt powinien być w stanie powiadamiać inne obiekty, nie przyjmując żadnych założeń co do tego, co te obiekty reprezentują – wynikiem są luźniejsze powiązania między obiektami

Observer - Struktura



Observer - Konsekwencje

- Abstrakcyjne powiązanie między *Obserwowanym* a *Obserwatorem*
 - mogą należeć do różnych warstw abstrakcji w systemie
- Wsparcie dla rozsyłania komunikatów
 - powiadomienie jest automatycznie nadawane do wszystkich zainteresowanych obiektów, które je zaprenumerowały
- Nieoczekiwane uaktualnienia
 - pozornie nieszkodliwa operacja zmiana stanu dotycząca *Obserwowanego* może spowodować kaskadę uaktualnień w *Obserwatorach* i obiektach od nich zależnych

Observer - Implementacja

- Obserwowanie więcej niż jednego *Obserwowanego*
 - *Obserwowany* może po prostu przekazać siebie jako argument operacji `Update()`, informując w ten sposób *Obserwatora*, który obiekt powinien sprawdzić
- Przesyłanie informacji o zmianie do *Obserwatora* – dwa modele:
 - **model push** – *Obserwowany* wysyła szczegółową informację o zmianie (bez względu, czy *Obserwatorzy* tego chcą, czy nie)
 - **model pull** – *Obserwowany* nie wysyła niczego poza powiadomieniem, a *Obserwatorzy* jawnie pytają potem o szczegóły

Observer - Podsumowanie

- Umożliwia obiektom dynamiczne rejestrowanie zależności między obiekta, dzięki czemu obiekty mogą powiadamać swoje obiekty zależne o istotnych zmianach swoich stanów
- Obiekty obserwujące są luźno powiązane z obiektem obserwowanym
- Informacje o zmianach stanu obiektu obserwowanego mogą być wysyłane lub pobierane

Command

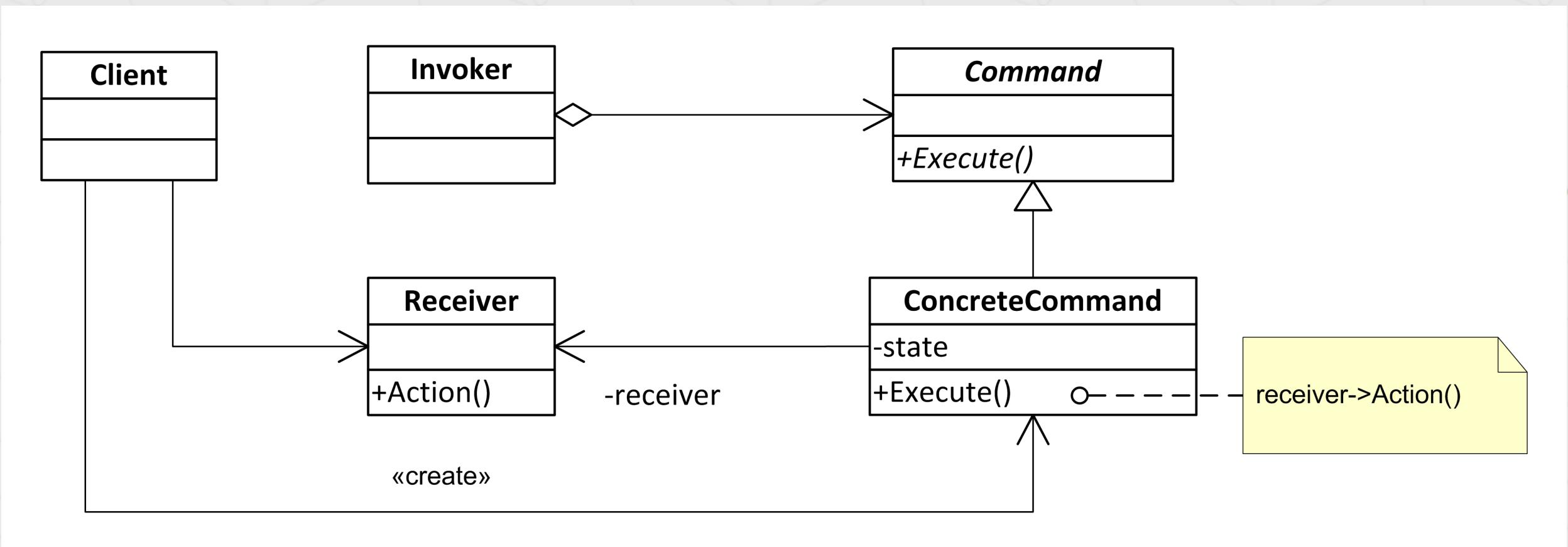
Command

- Hermetyzuje żądania w postaci obiektu
- Usuwa sprzężenie pomiędzy obiektem wystawiającym żądanie, a obiektem, który wie, jak należy je zrealizować
- Umożliwia
 - parametryzowanie klientów różnymi żądaniami
 - kolejkowanie żądań oraz zapisywanie w dziennikach
- Ułatwia implementację anulowania operacji
- Enkapsuluje odbiorcę (obiekt realizujący) z operacją lub szeregiem operacji, które mają być zrealizowane

Command - Kontekst / Problem

- Wzorca Command należy używać, gdy chcemy:
 - sparametryzować obiekt wykonywaną akcją - Command jest obiektową implementacją wywołań zwrotnych (*callbacks*)
 - uwzględnić możliwość anulowania wprowadzonych zmian
 - umożliwić wpisywanie zmian do dziennika, tak by można je było ponownie wykonać, gdy dojdzie do awarii systemu
 - stosować semantykę transakcji - transakcja hermetyzuje zbiór zmian danych w systemie

Command - Struktura



Command - Konsekwencje

- Wzorzec Command oddziela obiekt, który wywołał operację, od tego, który wie, jak ją wykonać – separacja interfejsów wywołującego od odbiorcy
- Polecenia są obiektami – mogą być przetwarzane i rozszerzane tak jak inne obiekty
- Można łatwo dodawać nowe polecenia, gdyż nie wymaga to modyfikowania istniejących klas

Command - Implementacja

- Polecenia mogą wspomagać anulowanie i przywracanie operacji, o ile zapewniają odpowiednie narzędzia do tego (np. operację Undo lub Redo)
- Klasa `ConcreteCommand` może wymagać w tym celu pamiętania dodatkowego stanu
- Stan ten może zawierać:
 - obiekt odbiorcy (Receiver), który faktycznie wykonuje operacje w odpowiedzi na żądanie
 - argumenty operacji wykonywanych przez odbiorcę
 - wszystkie początkowe wartości z odbiorcy, które mogą zmienić się w wyniku obsługi żądania – odbiorca musi zapewnić operacje, które umożliwiają przywrócenie odbiorcy do poprzedniego stanu

Command - Pokrewne wzorce

- Composite – wzorzec ten może zostać użyty do implementacji poleceń typu makro
- Prototype – polecenie, które musi być skopiowane przed umieszczeniem go na liście poleceń, działa jak Prototyp
- Memento – wzorzec często wykorzystywany do implementacji anulowania operacji

Command - Podsumowanie

- Oddziela obiekt żądający wykonania danej operacji od obiektu, który wie jak tą operację wykonać
- Ułatwia kolejkowanie, selekcję i sterowanie czasem wykonywania poleceń
- Ułatwia wycofywanie i ponowne wykonywanie poleceń
- Ułatwia utrzymywanie trwałej historii wykonanych poleceń

Memento

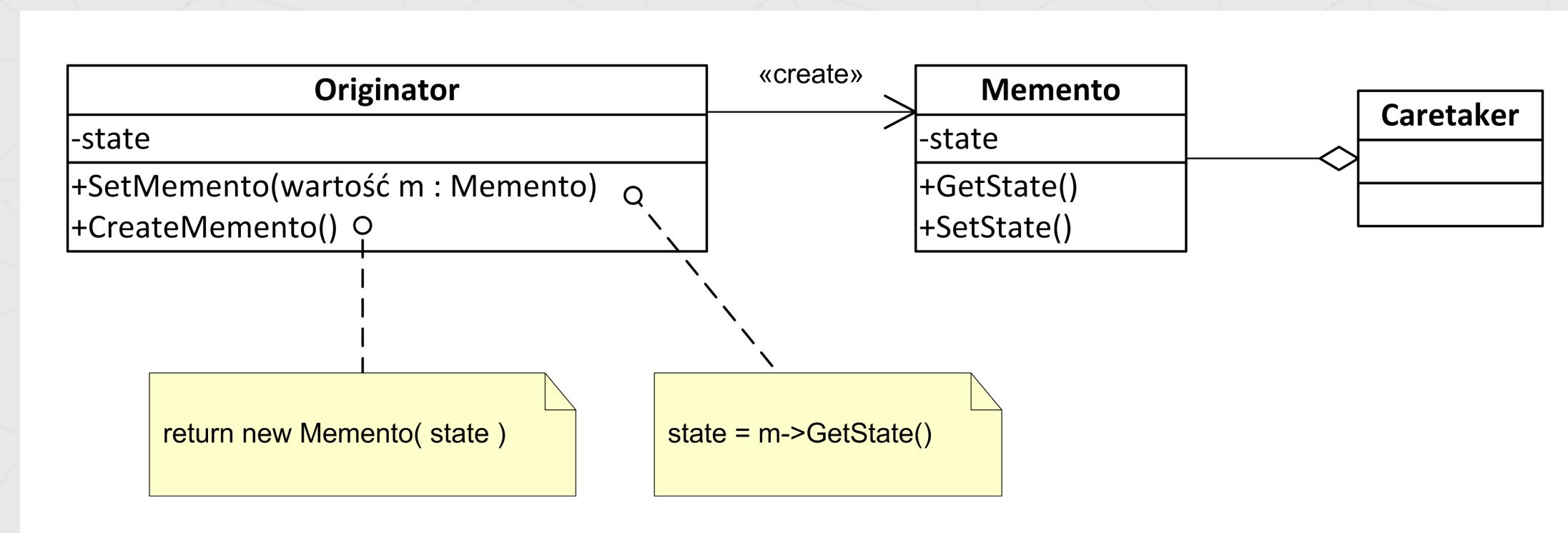
Memento

- Nie naruszając enkapsulacji, zapamiętuje i udostępnia na zewnątrz stan wewnętrzny obiektu, tak że obiekt może być później przywrócony do zapamiętanego stanu
- Memento jest obiektem przechowującym migawkę wewnętrznego stanu innego obiektu – jej źródła

Memento - Problem / Kontekst

- Istnieje potrzeba zapamiętania migawki stanu obiektu (lub części stanu), tak by można go było potem przywrócić do tego stanu
- Bezpośredni interfejs do uzyskania stanu ujawniałby szczegóły implementacji i naruszał hermetyzację obiektu

Memento - Struktura



Memento - Podsumowanie

- Zapamiętuje i udostępnia bez naruszenia zasad hermetyzacji wewnętrzny stan obiektu
- Dzięki pamiętce obiekt może być przywrócony do poprzedniego stanu

Mediator

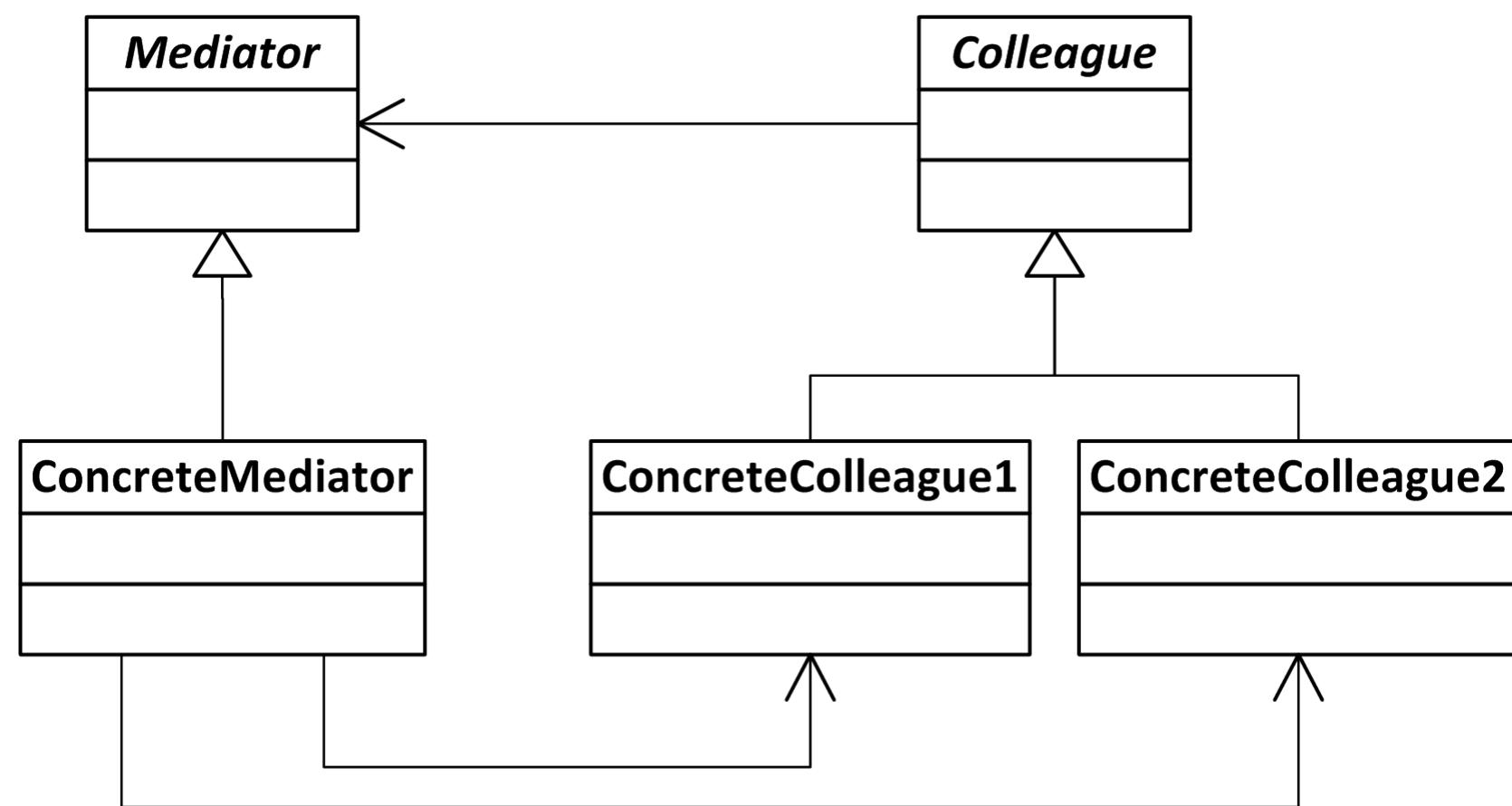
Mediator

- Definiuje obiekt enkapsulujący informacje o tym, jak obiekty współpracują
- Przyczynia się do rozluźnienia powiązań między obiektami, ponieważ obiekty nie odwołują się do siebie wprost
- Zapewnia separację współpracujących obiektów od systemu

Mediator - Stosowalność

- Zbiór obiektów porozumiewa się w dobrze zdefiniowany, lecz skomplikowany sposób
 - wynikające stąd zależności są nieuporządkowane i trudne do zrozumienia
- Ponowne użycie obiektu jest utrudnione, ponieważ odwołuje się i komunikuje się z wieloma innymi obiekttami
- Implementacja zachowania jest rozproszona w wielu klasach - zmiana wymaga utworzenia wielu klas pochodnych

Mediator - Struktura



Mediator - Podsumowanie

- Hermetyzuje proces komunikacji między obiektami
- Definiuje luźne powiązania między obiektami
- Umożliwia łatwą reimplementację sposobu współpracy obiektów typu **Coleague**, bez konieczności definiowania klas pochodnych dla nich

Visitor

Visitor

- Określa operację, która ma być wykonana na elementach struktury obiektowej
- Umożliwia definiowanie nowej operacji bez modyfikowania klas elementów, na których ona działa

Visitor - Kontekst / Problem

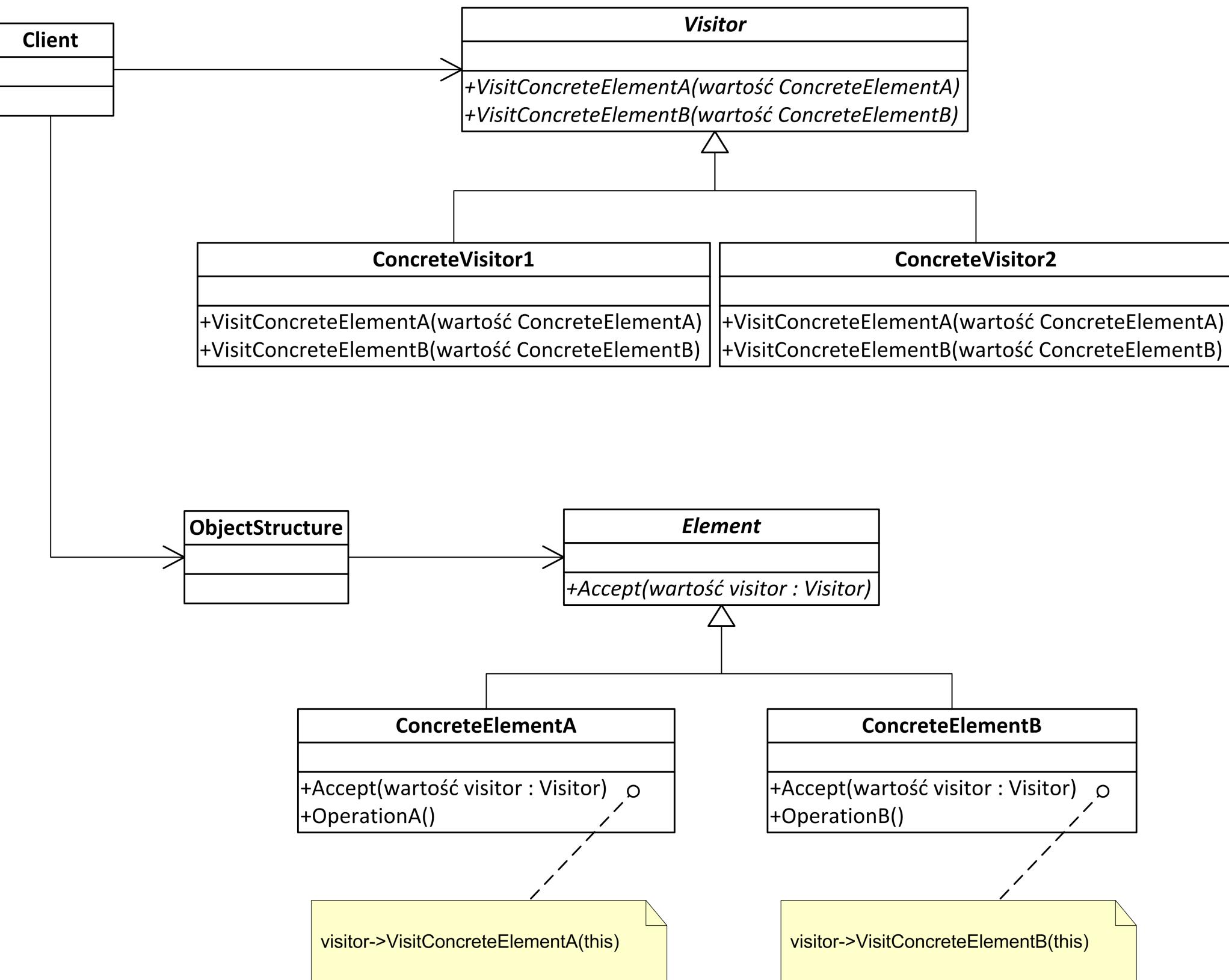
■ Kontekst

- klasy definiujące strukturę obiektową rzadko się zmieniają, ale chcemy często definiować nowe operacje w tej strukturze

■ Problem

- chcemy umożliwić łatwe dodawanie nowej operacji do struktury obiektowej bez konieczności otwierania klas tej struktury

Visitor - Struktura



Visitor - Konsekwencje

Visitor - Konsekwencje

■ Łatwe dodawanie nowych operacji

- odwiedzający ułatwiają dodawanie operacji zależących od komponentów złożonych obiektów
- nową operację na strukturze obiektowej definiuje się po prostu przez dodanie nowego odwiedzającego (Visitor)

Visitor - Konsekwencje

- Łatwe dodawanie nowych operacji
 - odwiedzający ułatwiają dodawanie operacji zależących od komponentów złożonych obiektów
 - nową operację na strukturze obiektowej definiuje się po prostu przez dodanie nowego odwiedzającego (Visitor)
- Zebranie razem powiązanych ze sobą operacji, a rozdzielenie tych niepowiązanych
 - powiązane ze sobą zachowania nie są rozsiane po wszystkich klasach definiujących strukturę obiektową, lecz są umiejscowione w klasie Visitor
 - upraszcza to zarówno klasy definiujące elementy, jak i algorytmy zdefiniowane w odwiedzających

Visitor - Konsekwencje

- Łatwe dodawanie nowych operacji
 - odwiedzający ułatwiają dodawanie operacji zależących od komponentów złożonych obiektów
 - nową operację na strukturze obiektowej definiuje się po prostu przez dodanie nowego odwiedzającego (Visitor)
- Zebranie razem powiązanych ze sobą operacji, a rozdzielenie tych niepowiązanych
 - powiązane ze sobą zachowania nie są rozsiane po wszystkich klasach definiujących strukturę obiektową, lecz są umiejscowione w klasie Visitor
 - upraszcza to zarówno klasy definiujące elementy, jak i algorytmy zdefiniowane w odwiedzających
- Trudne dodawanie nowych klas do wizytowanej hierarchii

Visitor - Konsekwencje

Visitor - Konsekwencje

- Odwiedzanie całej hierarchii klas
 - odwiedzający może odwiedzać obiekty nie mające wspólnego rodzica
 - do interfejsu Visitor można dodać operacje na obiektach dowolnego typu

Visitor - Konsekwencje

- Odwiedzanie całej hierarchii klas
 - odwiedzający może odwiedzać obiekty nie mające wspólnego rodzica
 - do interfejsu Visitor można dodać operacje na obiektach dowolnego typu
- Kumulowanie stanu
 - odwiedzający mogą kumulować stan w miarę odwiedzania poszczególnych elementów struktury obiektowej.
 - bez wzorca ten stan byłby przekazywany jako dodatkowe argumenty do operacji wykonujących przechodzenie lub mógłby pojawić się w postaci zmiennych globalnych

Visitor - Implementacja

- Wykorzystanie idiomu CRTP

```
01 class Expression
02 {
03     public:
04         virtual void accept(Visitor& v) = 0;
05         virtual ~Expression() = default;
06 };
```

```
01 template <typename VisitableType>
02 class VisitableExpression : public Expression
03 {
04 public:
05     void accept(Visitor& v)
06     {
07         v.visit(static_cast<VisitableType&>(*this));
08     }
09 };
```

```
01 class IntExpr : public VisitableExpression<IntExpr>
02 {
03     //...
04 };
```

Visitor - Pokrewne wzorce

■ Composite

- odwiedzających można użyć do wykonania operacji dla wszystkich elementów struktury obiektowej, zdefiniowanej przez wzorzec Composite

■ Interpreter

- wzorzec Visitor można zastosować do interpretowania AST

Designing for Change

Designing for Change

- A design that doesn't take change into account risks major redesign in the future
- Those changes might involve class redefinition and reimplementations, client modification, and retesting
- Design patterns help you avoid this by ensuring that a system can change in specific ways - each design pattern lets some aspect of system structure vary independently of other aspects

■ Creating an object by specifying a class explicitly

- Factory Method
- Abstract Method
- Prototype

- **Dependence on specific operations**
 - Command
 - Chain of Responsibility

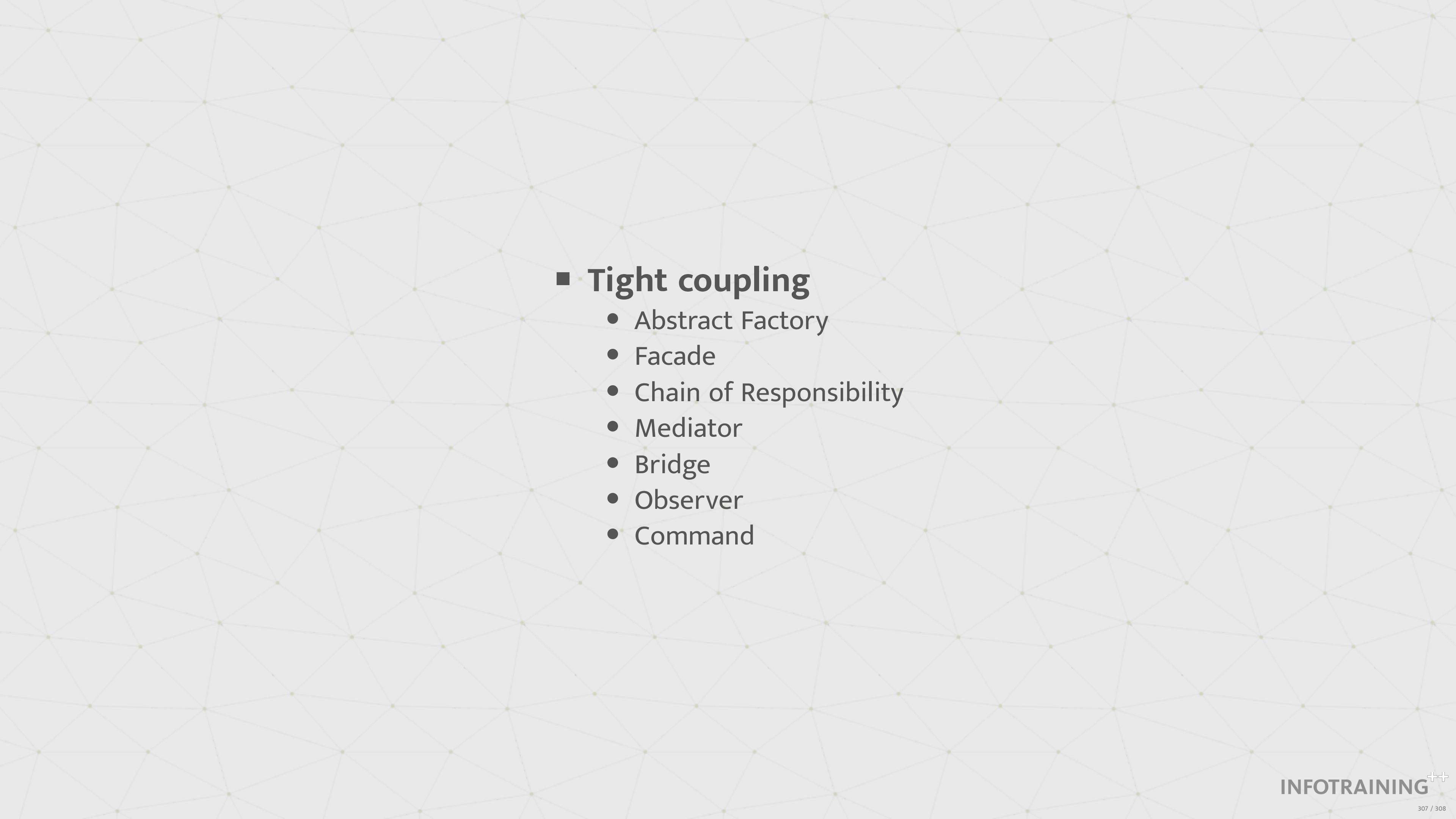
- **Dependence on hardware and software platform**
 - Abstract Factory
 - Bridge

■ Dependence on object representations or implementations

- Abstract Factory
- Bridge
- Memento
- Proxy

■ Algorithmic dependencies

- Builder
- Iterator
- Template Method
- Strategy
- Visitor

A faint, light-gray network graph serves as the background for the slide. It consists of numerous small, semi-transparent green dots representing nodes, connected by a web of thin, light-gray lines representing edges. This creates a sense of a complex, interconnected system.

■ Tight coupling

- Abstract Factory
- Facade
- Chain of Responsibility
- Mediator
- Bridge
- Observer
- Command

A faint, light-gray network graph serves as the background for the slide. It consists of numerous small, semi-transparent green dots representing nodes, connected by a web of thin, light-gray lines representing edges. This pattern creates a sense of a complex, interconnected system.

■ Extending functionality by subclassing

- Decorator
- Composite
- Strategy