Wzorce projektowe [wstęp]

Motywacje definiowania wzorców projektowych

Za twórcę uważany jest amerykański architekt Christopher Alexander
Alexander, C., Ishikawa, S., Silverstein, M., *The Timeless Way of Building*,
New York: Oxford University Press, 1979.

Ocena, czy dana budowla jest piękna, jest nie tylko kwestią smaku i subiektywnego punktu widzenia. Podlega weryfikacji w oparciu o mierzalne, zdefiniowane wielkości. Tymi wielkościami są wzorce projektowe. Co jest obecne w dobrym projekcie, a czego nie ma w złym (i vice versa)? Jakość projektu jest rzeczą podlegającą obiektywnej analizie i ocenie, zatem powinniśmy potrafić zdefiniować co dany projekt czyni dobrym lub złym. Potrzeba zatem przeanalizować różnego rodzaju struktury rozwiązujące ten sam problem. Odnajdując w różnych projektach wspólne części stanowiące dobre rozwiązanie, możemy zdefiniować te wspólne części jako wzorce.

Wzorzec – rozwiązanie problemu w danym kontekście

Wzorzec zawsze ma nazwę i cel.
Szukając właściwego wzorca należy wprost określić dany problem, sytuację do rozwiązania.
Zaletą używania wzorców jest to, że można czerpać wprost z rozwiązania wypracowanego przez innych jako dobre (najlepsze) w danym zagadnieniu, unikając błędów.

- Zostały pomyślane jako zestaw sprawdzonych koncepcji architektonicznych
- Dzięki nim każdy miał mieć możliwość zbudować swój własny dom
- Wzorce Alexandra nie znalazły uznania wśród innych architektów

Wzorce projektowe [programowanie]

Wzorce w programowaniu

Wprowadzono je do inżynierii oprogramowania w 1995 („**G**ang **o**f **F**our” – „banda czworga” – E. Gamma, R. Helm, R. Johnson, J. Vlissides „*Design Patterns: Elements of Reusable Object-Oriented Software*”)

Czym są wzorce projektowe?

- Rozwiązanie problemu w określony sposób
- Opisuje problem który się stale powtarza
- Stanowią abstrakcyjny opis zależności pomiędzy klasami
- Znaczna ich część stanowi obszerna dokumentacja opisująca sposób użycia wzorca
- Są łączone w celu rozwiązania bardziej złożonego problemu
- Algorytmy nie są wzorcami projektowymi jako że rozwiązują problemy obliczeniowe, a nie projektowe

Klasyfikacja

Konstrukcyjne - Strukturalne - Operacyjne

Wzorce projektowe [systematyka]

Nazwa wzorca
Odzwierciedla problem, rozwiązanie i konsekwencje danego wzorca

Problem
Opisuje zagadnienie i kontekst wystąpienia wzorca

„Każdy wzorzec opisuje problem, który ciągle na nowo pojawia się w naszym otoczeniu i opisuje rdzeń jego rozwiązania w taki sposób, że można go używać milion razy i nigdy w ten sam sposób."

Christopher Alexander

Rozwiązanie
Opisuje elementy tworzące projekt, ich relacje, odpowiedzialności oraz współpracę

Konsekwencje
Rezultaty zastosowania wzorca – korzyści i straty

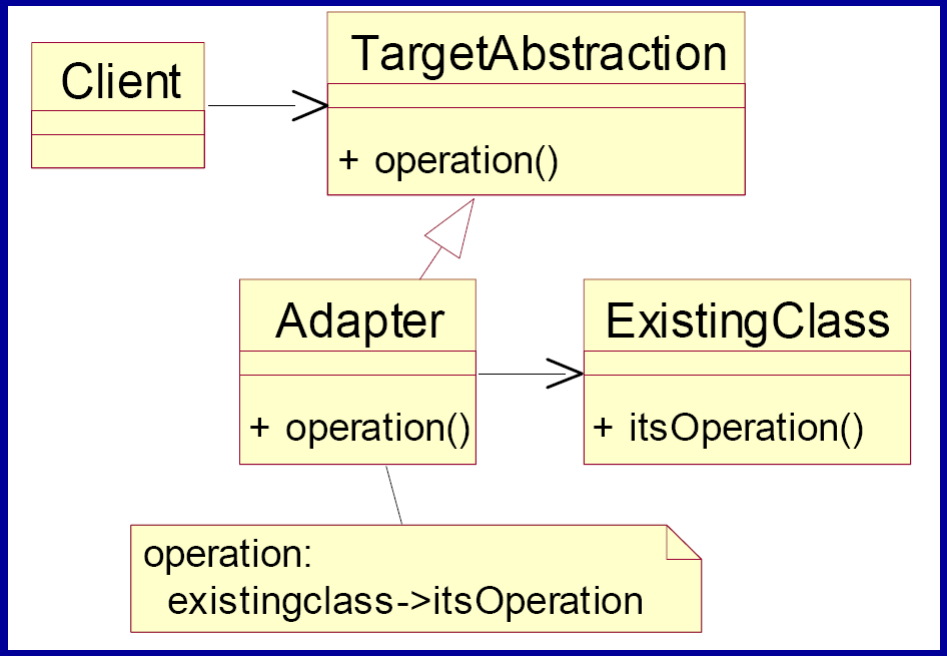
Wzorce projektowe pomagają w realizacji zasady „**open-closed**" (sformułowanej przez Bertranda Meyera): moduły, metody i klasy powinny być otwarte na rozszerzenia, pozostając zarazem zamkniętymi na modyfikacje. Program należy tak projektować, żeby dało się powiększać jego funkcjonalność bez zmieniania programu (interfejsów).

Wzorce projektowe [systematyka]

		Przeznaczenie		
		Konstrukcyjne (Creational)	**Strukturalne (Structural)**	**Behawioralne (Behavioral)**
Zakres	**Klasa**	Factory Method	Adapter	Interpreter Template Method
	Obiekt	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Wzorce projektowe [adapter]

Cel: dostosować (zaadaptować) istniejący obiekt do używanego interfejsu
Problem: obiekt posiada właściwe atrybuty i zachowanie, ale inaczej zdefiniowany interfejs niż ten przez nas używany
Rozwiązanie: Adapter dostarcza opakowanie wraz z pożądanym interfejsem, dzięki czemu pozwala na dopasowanie użycia istniejących obiektów do nowych klas
Implementacja: poprzez zawarcie istniejącej klasy w nowej klasie, która wywołuje metody starej klasy

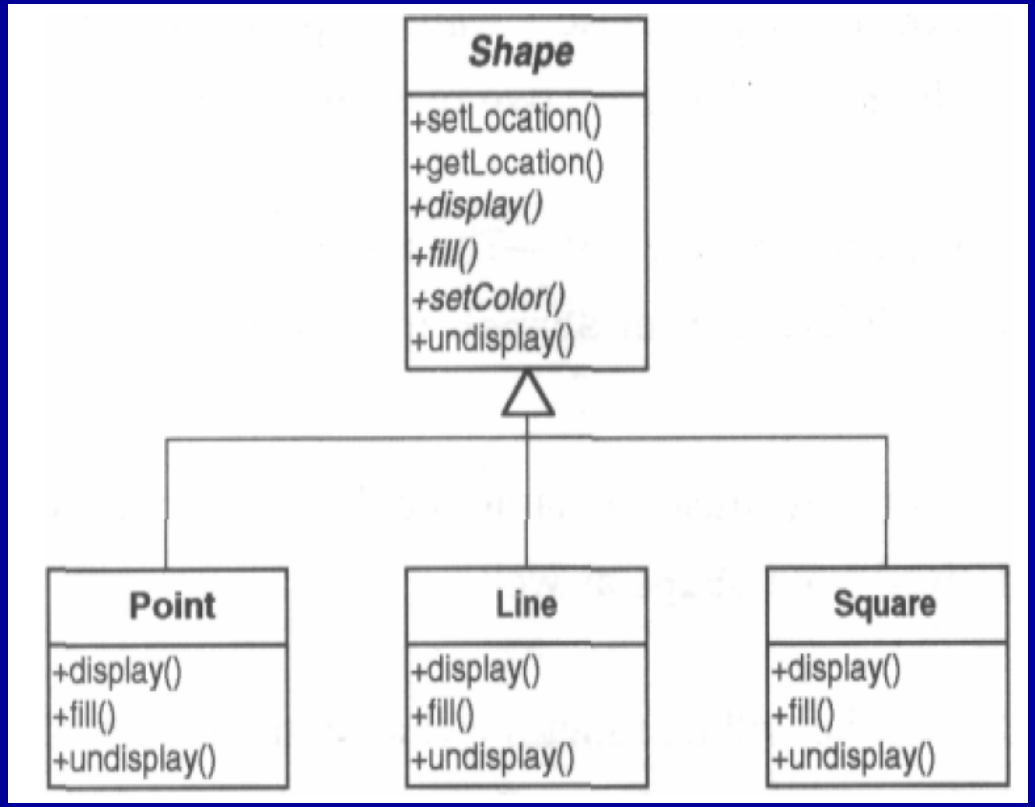


Definicja GoF:

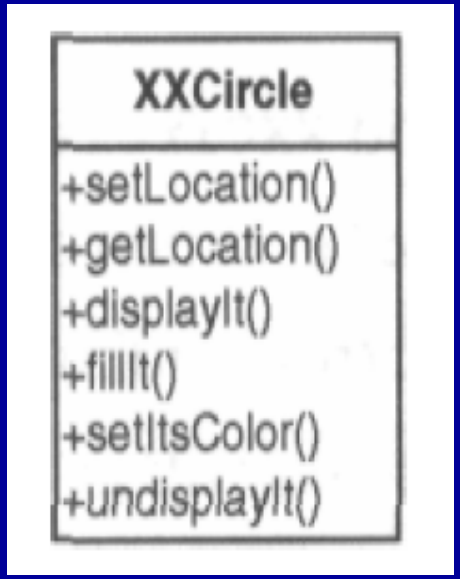
Adapter konwertuje interfejs klasy na inny interfejs, oczekiwany (używany) przez klienta. Umożliwia dzięki temu współpracę klas, która w innym wypadku nie byłaby możliwa ze względu na niezgodność interfejsów.

Wzorce projektowe [adapter - przykład]

- Mamy system do rysowania różnych figur geometrycznych. Oparty jest on na klasie abstrakcyjnej `Shape` i wywodzących się z niej klasach konkretnych.
- Dostajemy zadanie dodać obsługę okręgu (`circle`) – nowe wymaganie.
- Ktoś już napisał klasę realizującą nasze potrzeby, ale zupełnie innych nazwach metod (interfejs niezgodny z naszym).

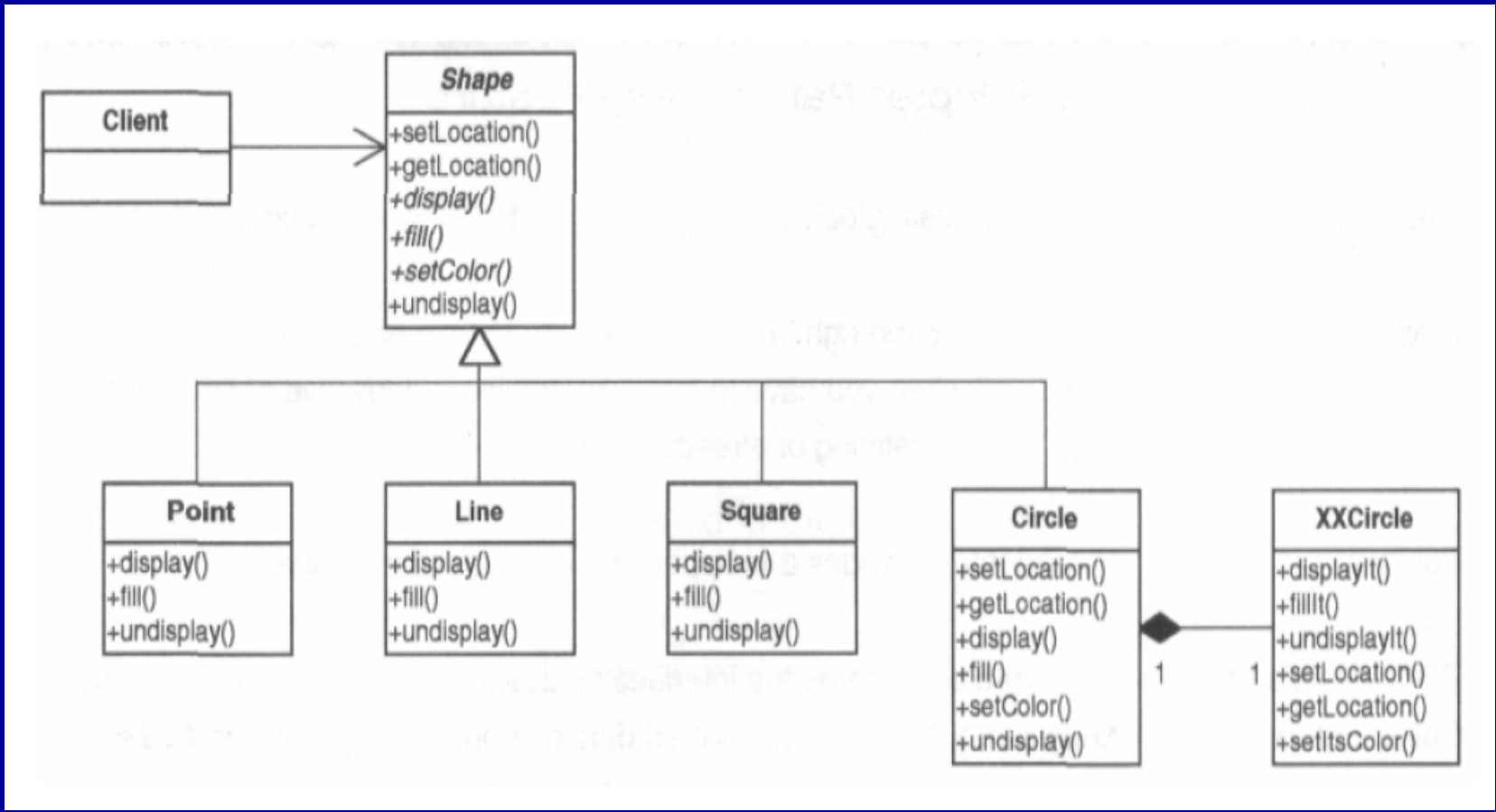


Kod już istnieje i nie chcemy go przepisywać od nowa. Chcemy użyć go w naszym systemie – adapter!



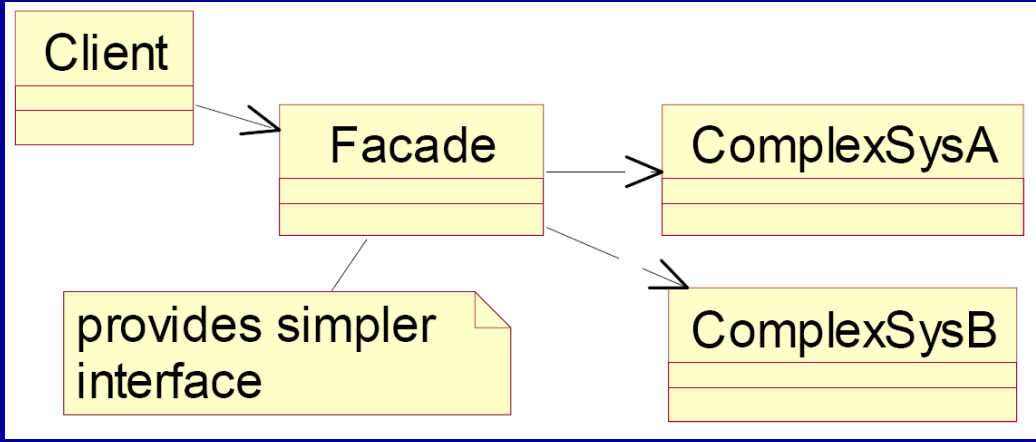
Wzorce projektowe [adapter - rozwiązanie]

- Klasa `Circle` dziedziczy z klasy `Shape` – dzięki temu mamy polimorficzne zachowanie
- Klasa `Circle` zawiera w sobie klasę `XXCircle`
- Klasa `Circle` przekazuje wywołania do klasy `XXCircle`, używając jej metod



Wzorce projektowe [facade - fasada]

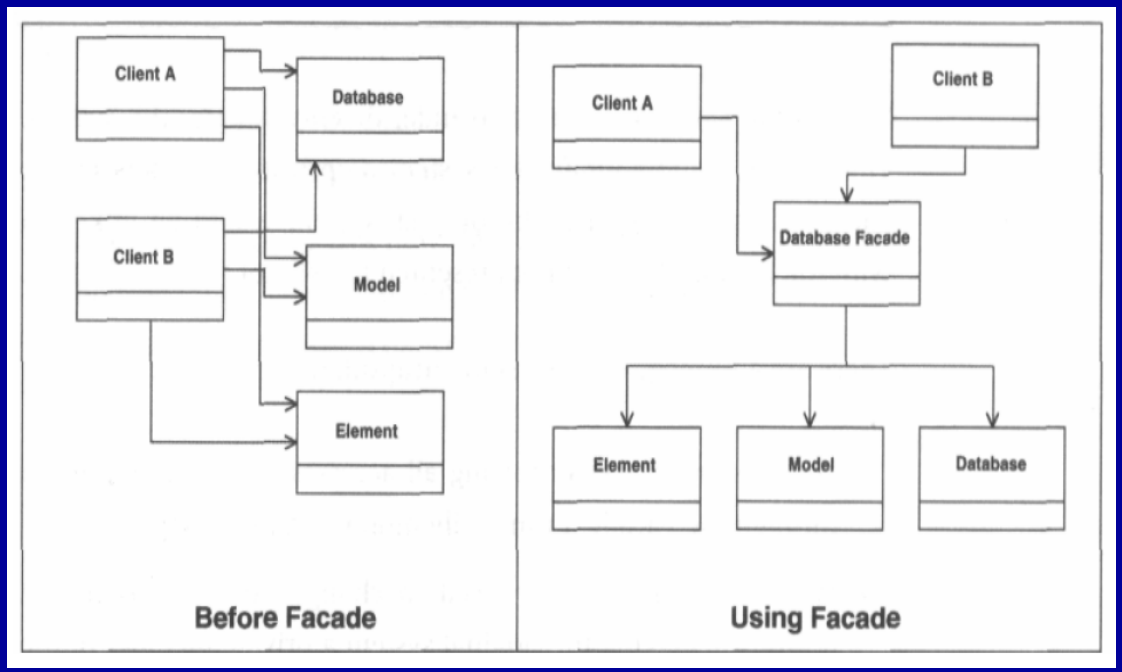
Cel: uprościć użycie istniejącego systemu poprzez prostszy interfejs
Problem: system posiada złożony i skomplikowany interfejs, a my chcemy użyć tylko części z możliwości systemu, albo współpracować z systemem w jakiś jeden określony sposób
Rozwiązanie: Fasada dostarcza nowy interfejs, specjalizowany (uproszczony, zredukowany) do (mniejszych, specyficznych) potrzeb użytkownika
Implementacja: zdefiniuj nową klasę (klasy) z interesującym cię interfejsem i niech ona używa skomplikowany istniejący system



Definicja GoF:
Fasada dostarcza zunifikowany interfejs do zbioru interfejsów złożonego systemu, definiuje interfejs wyższego poziomu ułatwiający użycie systemu.

Wzorce projektowe [fasada – przykłady zastosowań]

- Wzorca fasady użyjemy nie tylko do uproszczenia interfejsu, ale również do zredukowania liczby obiektów, z którymi musi współpracować klient.



Przykładowo, klient, aby dostać informację o elemencie (Element) musi kolejno nawiązać połączenie z bazą danych, uzyskać z niej Model. Na podstawie modelu wysyła zapytanie o Element. Cała operacja byłaby prostsza gdyby napisać fasadę dla tych operacji na bazie danych.

- Wzorzec fasady może nie tylko redukować (upraszczać) interfejs, ale dodawać do niego jakąś nową funkcjonalność.
- Wzorca można użyć w przypadku chęci enkapsulacji systemu, np. za pomocą fasady można kontrolować wszystkie odwołania do systemu albo przewidywana jest możliwość zmiany systemu – wtedy używany interfejs się dla klienta nie zmieni.
- Wreszcie – fasadę pisze się jeśli koszt napisania nowej klasy (klas) jest mniejszy niż koszt szkolenia każdego z klientów na temat użytkowania całego oryginalnego systemu lub utrzymanie kodu fasady jest łatwiejsze (mniej kosztowne).

Wzorce projektowe [adapter / fasada – porównanie]

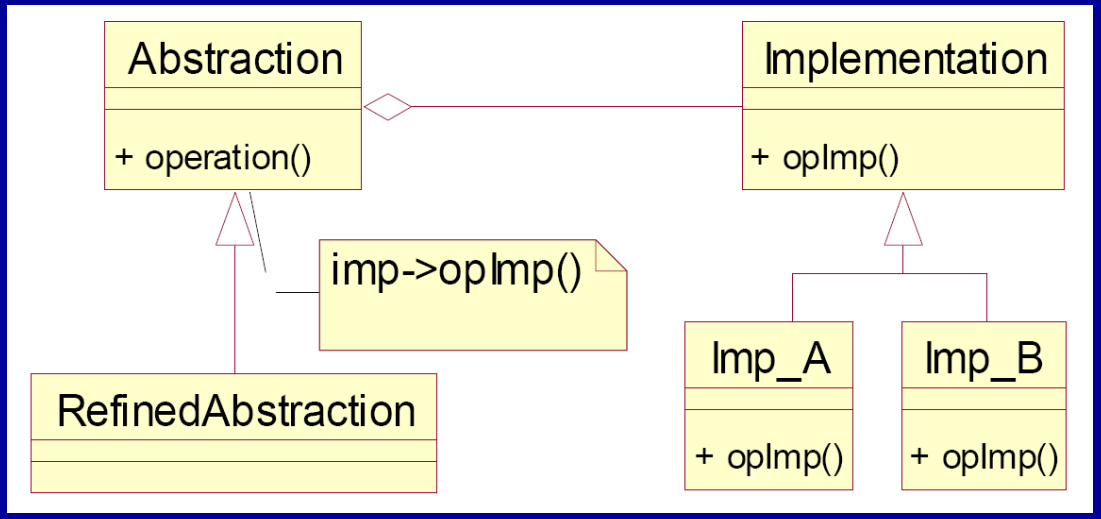
- Oba wzorce oznaczają napisanie klasy opakowującej, ale posiadają kilka wyróżniających je cech:

	Fasada	Adapter
Czy istnieją klasy, w oparciu o które napisane zostaną wzorce?	tak	tak
Czy musimy zaprojektować interfejs dla wzorca?	tak	nie
Czy obiekt ma zachowywać się polimorficznie?	nie	być może tak
Czy jest potrzebny prostszy interfejs?	tak	nie

Fasada upraszcza interfejs, zaś Adapter konwertuje interfejs

Wzorce projektowe [bridge - most]

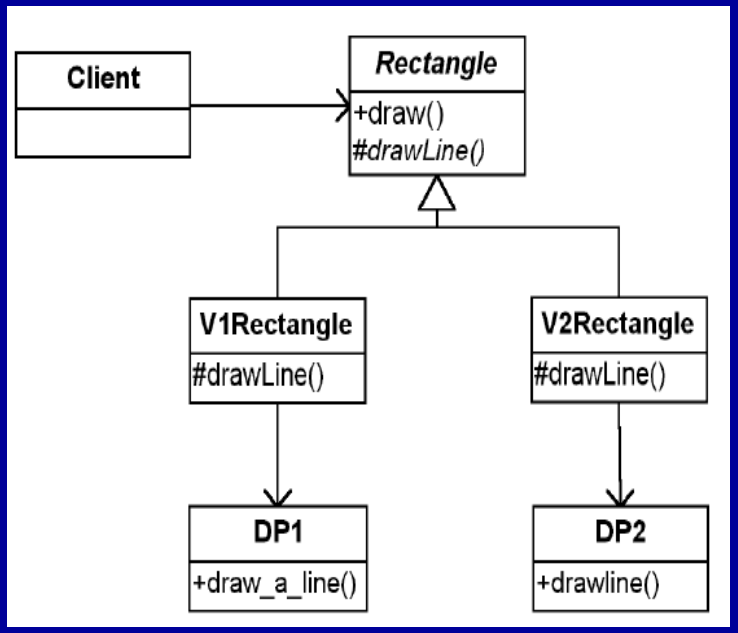
Cel: rozdzielić zestaw implementacji od zbioru obiektów ich używających
Problem: pochodne z klasy abstrakcyjnej muszą używać wielu różnych implementacji, nie powodując przy tym kombinatorycznej eksplozji wszystkich możliwości (połączenia wszystkich pochodnych z wszystkimi implementacjami)
Rozwiązanie: zdefiniuj interfejs dla wszystkich implementacji i niech on będzie używany przez obiekty klas pochodnych z klasy abstrakcyjnej
Implementacja: ukryj implementacje w bazowej klasie abstrakcyjnej (dla wszystkich implementacji), zaś w klasie abstrakcyjnej obiektów dostarcz uchwyt do klasy abstrakcyjnej implementacji



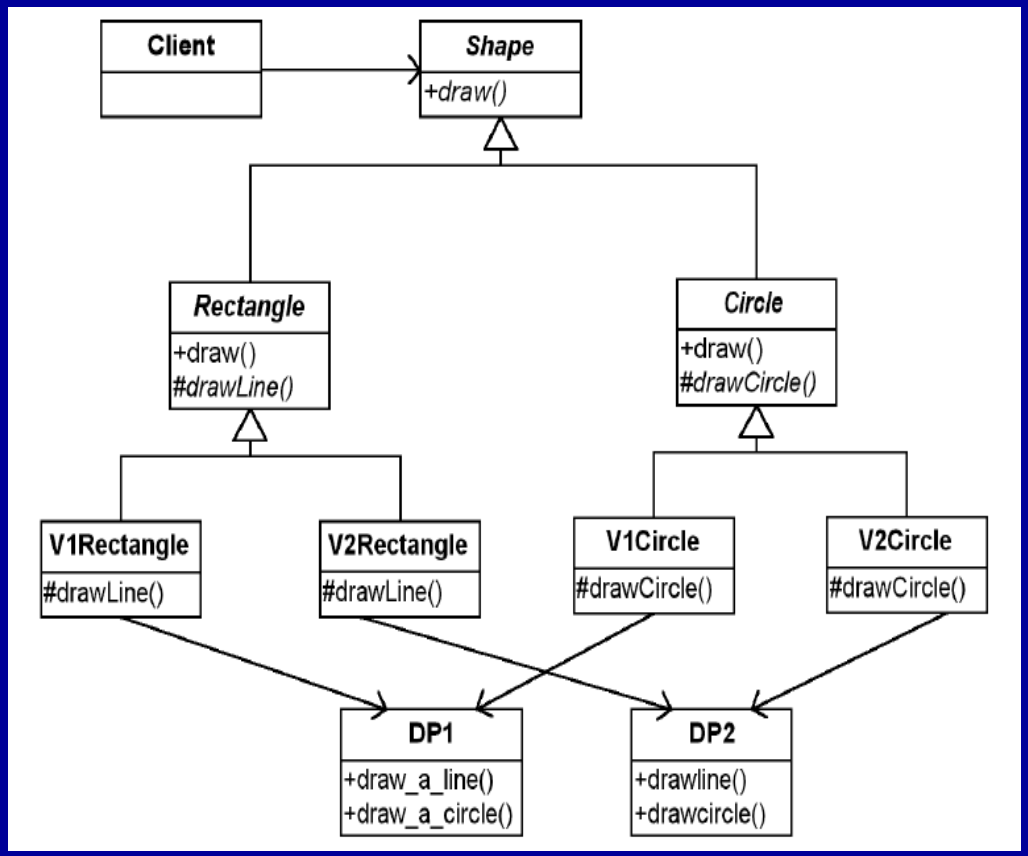
Definicja GoF:
Wzorzec mostu służy dokonaniu rozdziału pomiędzy abstrakcją a jej implementacją, tak że każda może zmieniać się niezależnie.

Wzorce projektowe [most – przykład]

Mamy napisać program, który będzie rysował prostokąty za pomocą dwóch różnych bibliotek graficznych, w momencie utworzenia konkretnego prostokąta jest wiadome, która biblioteka zostanie użyta do wizualizacji.



Po zrealizowaniu tego zadania (lewa strona) okazuje się, że mamy dopisać możliwość rysowania okręgów… (poniżej).

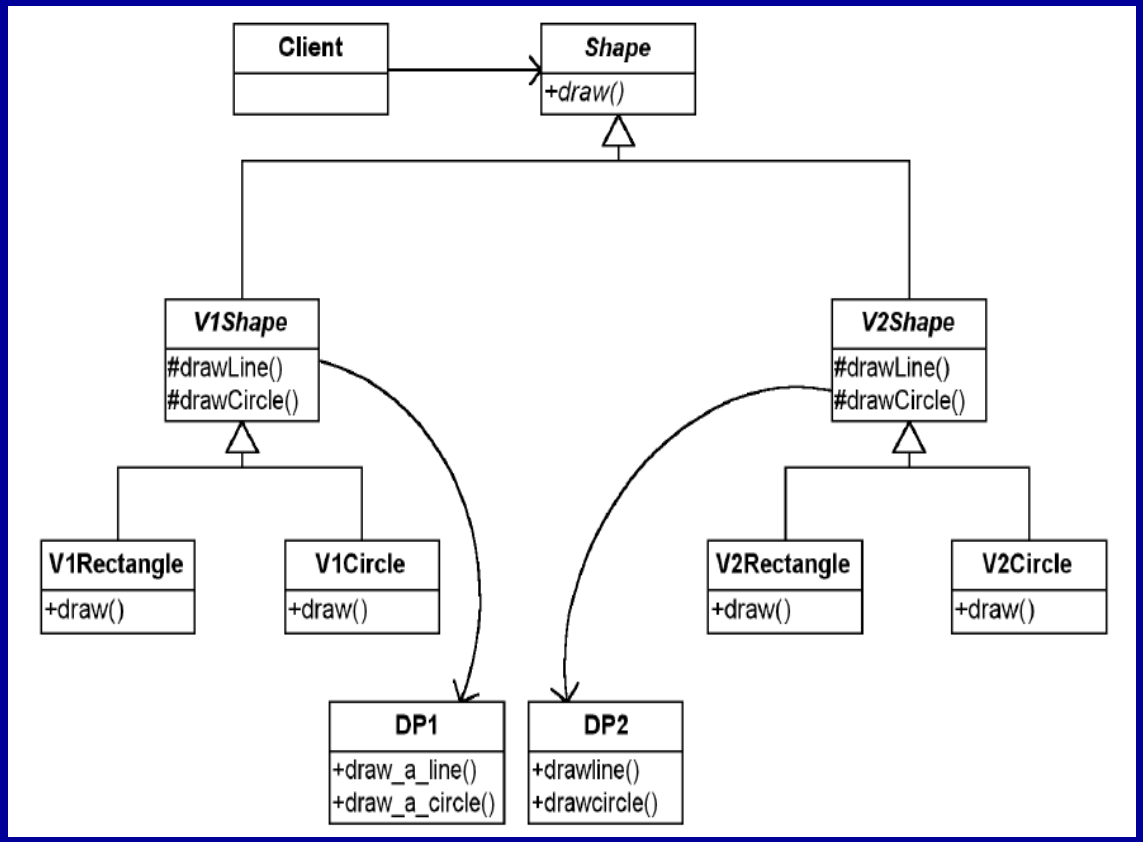


Wymagania klienta bardzo często się zmieniają i nasz projekt powinien być na nie przygotowany… według pomysłu po prawej, jesteśmy bliscy **kombinatorycznej eksplozji**! Co się stanie, jeśli trzeba będzie dodać jakąś trzecią bibliotekę graficzną?

Wzorce projektowe [most – przykład c.d.]

Problem w takiej realizacji wynika ze ścisłego powiązania realizacji abstrakcji (rodzaje kształtów) i implementacji (biblioteki graficzne). Każdy rodzaj kształtu musi wiedzieć którą bibliotekę używa do wizualizacji!

Podejmujemy próbę innego zaprojektowania hierarchii dziedziczenia:



Niestety w tym rozwiązaniu nadal mamy nadmiarowość w ilości klas opisujących prostokąty i okręgi, wizualizowane za pomocą różnych bibliotek graficznych.

Szukając rozwiązania we wzorcach należy zwracać uwagę nie na to „co zrobić” ale „kiedy” oraz „dlaczego” to zrobić – czyli spojrzeć na kontekst sytuacji.

Wzorce projektowe [most – przykład c.d. – analiza problemu]

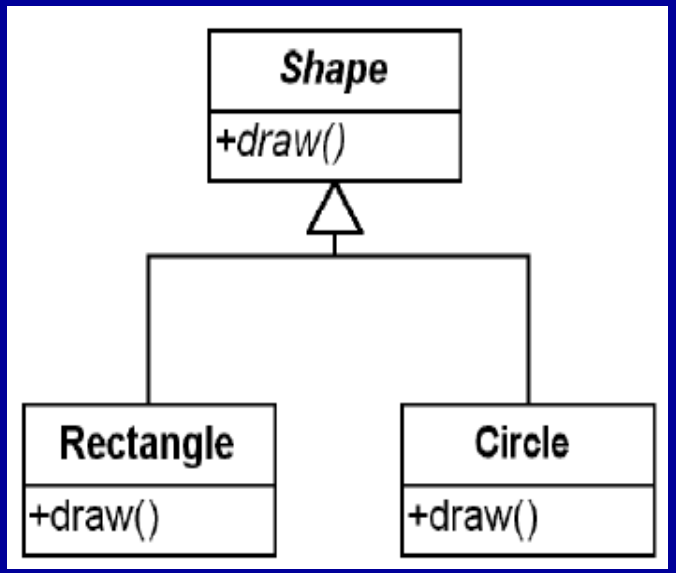
Jim Coplien: zlokalizuj gdzie rzeczy się mogą zmieniać („commonality analysis”), a następnie określ jak się zmieniają („variability analysis”). Wspólne części wyrażone zostaną przez klasę abstrakcyjną. Zmiany powinny znaleźć odzwierciedlenie w klasach konkretnych.

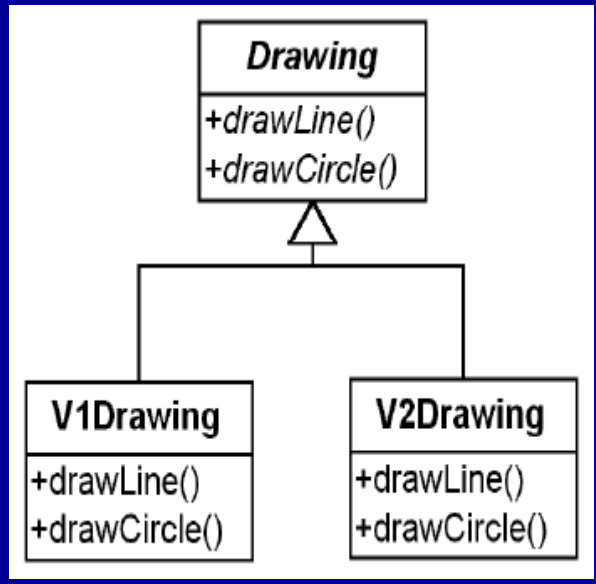
Podstawowe reguły projektowania:
1. zidentyfikuj co się może zmieniać i ukryj to
2. preferuj kompozycję przed dziedziczeniem

W naszym przykładzie mamy: różne typy kształtów i różne typy bibliotek graficznych. Wspólną ich częścią są zatem pojęcia „kształty” i „biblioteki graficzne”. Zatem klasa `Shape` zamknie w sobie część wspólną kształtów. Konkretne kształty muszą wiedzieć jak się narysować. Z kolei obiekty klasy `Drawing` są odpowiedzialne za to jak rysować linie, okręgi etc.

Która klasa ma używać drugiej? Jeśli biblioteka graficzna miałaby bezpośrednio rysować kształty, to musiałaby coś wiedzieć o kształtach. To pogwałca podstawową regułę obiektowości:

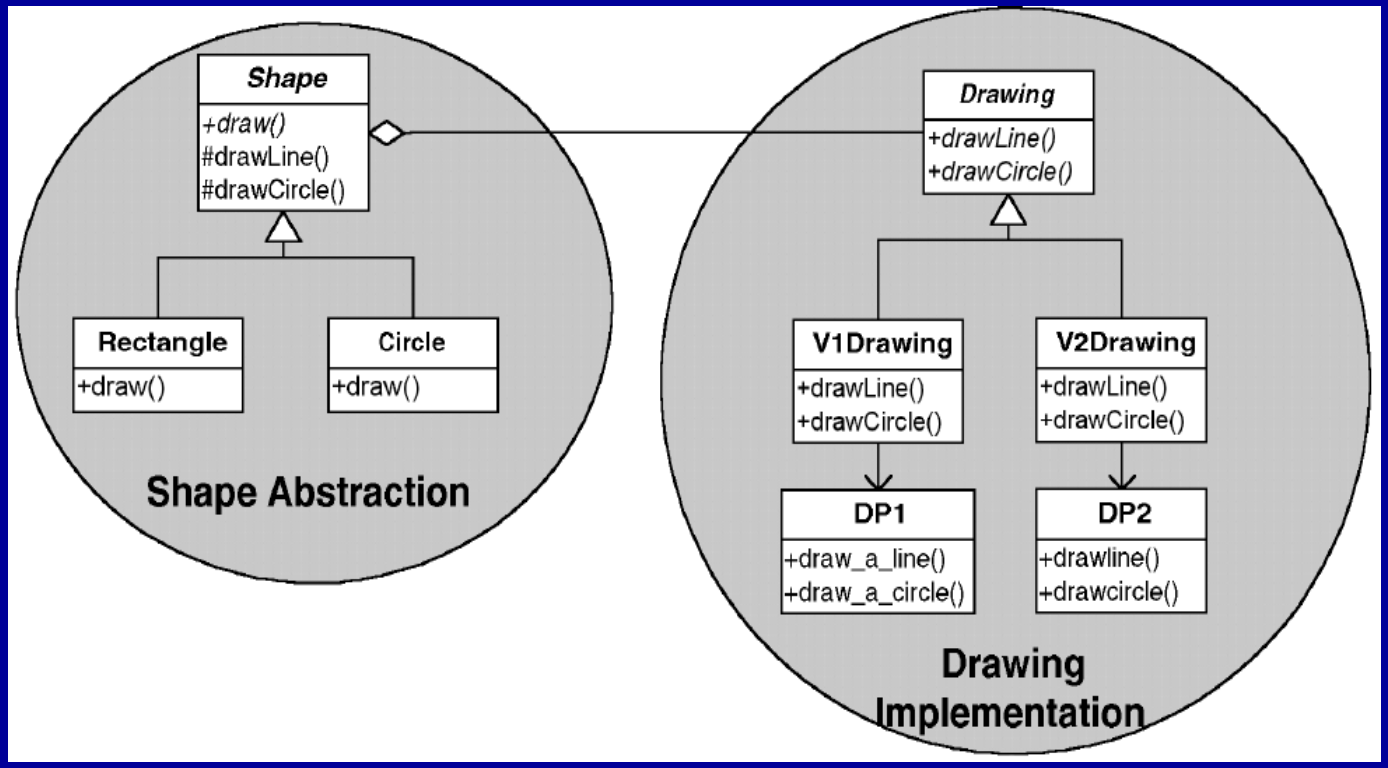
Obiekt odpowiada tylko za siebie!





Wzorce projektowe [most – rozwiązanie]

Obiekty klasy Shape używają obiektów klasy Drawing, nie wiedząc jaki dokładnie typ pochodny klasy Drawing jest użyty (w klasie abstrkacyjnej Shape odnosimy się do klasy bazowej Drawing).



Zauważmy, że klasy V1Drawing i V2Drawing realizują wzorzec Adaptera, dostosowując interfejsy różnych bibliotek graficznych do interfejsu klasy Drawing. Takie połączenie (Adapter + inny wzorzec) jest często spotykane.

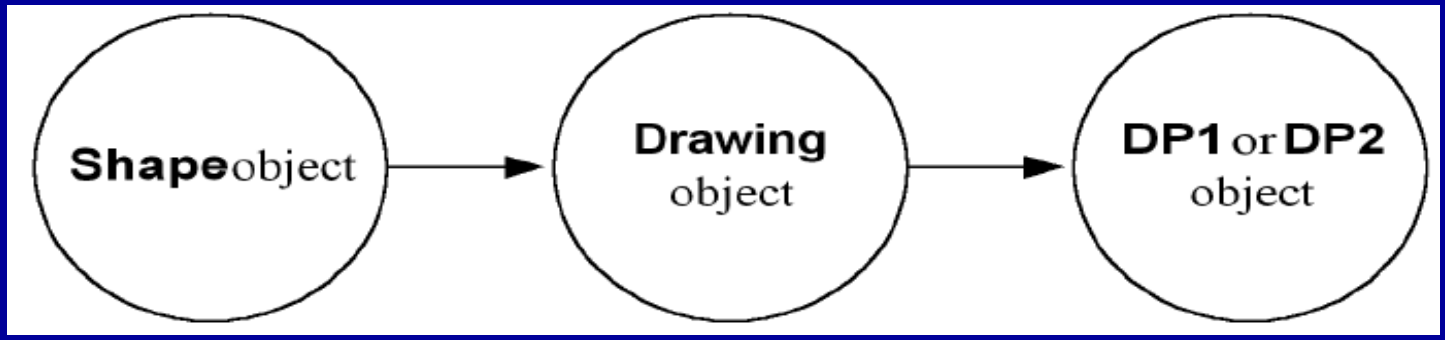
Wzorzec mostu pozwala na wydzielenie implementacji jako czegoś zewnętrznego, używanego przez obiekty (pochodne klasy Shape), dając znacznie większą swobodę w ukrywaniu zmian w implementacji (np. dodanie kolejnej biblioteki graficznej).

Wzorce projektowe [most – strategie implementacji]

Ważna strategia dotycząca implementacji: „**one rule, one place**”
Jeśli wiesz jak zrealizować daną rzecz, zaimplementuj ją raz. Odzwierciedleniem tego jest zwykle kod z większą ilością metod.

Przykładowo, metoda `draw()` z klasy Rectangle mogłaby bezpośrednio wołać metodę `drawLine()` z dowolnego obiektu klasy Drawing używanego przez klasę Shape. Lepiej jednak gdy umiejscawiamy daną strategię (tu: rysowanie za pomocą metod z Drawing) w jednym miejscu (tu: klasa Shape, która poprzez metody `drawLine()` i `drawCircle()` odnosi się do takich metod w klasie Drawing).

Z obiektowego punktu widzenia nasza realizacja wygląda następująco:



W rzeczywistości to jest Rectangle lub Circle ale klient tego nie wie, bo oba wyglądają tak samo (obiekty klasy Shape)	Realnie to jest V1Drawing lub V2Drawing, ale obiekt klasy Shape tego nie wie, bo oba wyglądają tak samo (obiekty klasy Drawing)	Tu musi być odpowiedni typ obiektu, ale obiekt klasy Drawing używający go będzie wiedział który

Wzorce projektowe [most – realizowane zasady OOP]

- **obiekty są odpowiedzialne tylko same za siebie** – mamy różne rodzaje obiektów Shape, ale każdy z nich dba o to jak się narysować (metoda draw). Obiekty klasy Drawing dbają o rysowanie elementów składowych obiektów

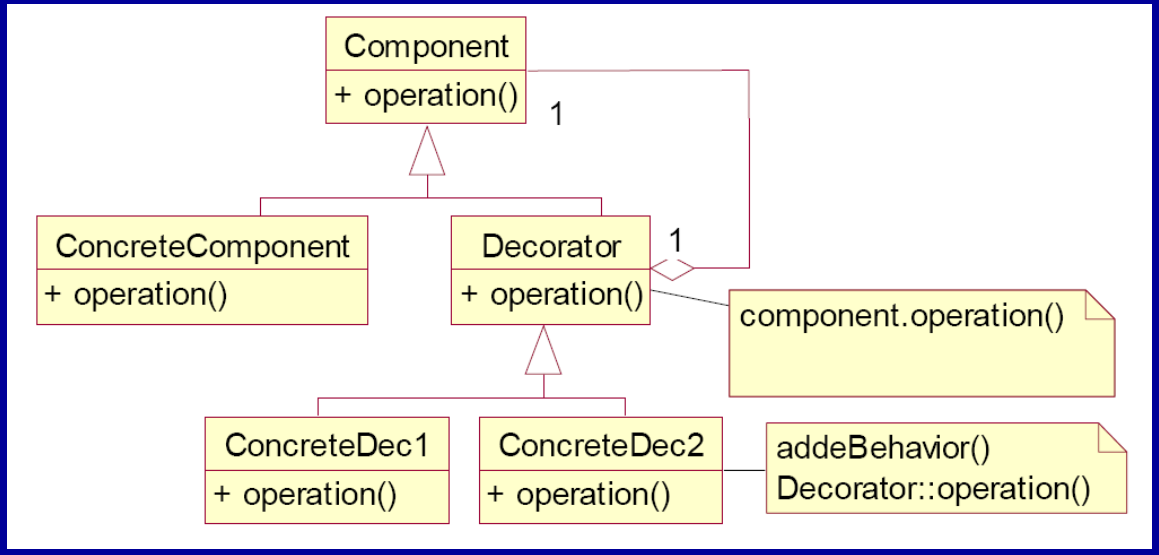
- **klasa abstrakcyjna** – realizujemy ogólną ideę za pomocą klasy, która nigdy nie będzie skonkretyzowana (klasa Shape reprezentująca wszystkie kształty)

- **enkapsulacja poprzez klasę abstrakcyjną** – należy zrozumieć wielorakie znaczenie enkapsulacji:
  - klient mający do czynienia z naszym wzorcem mostu będzie obsługiwał jakiś obiekt pochodny z klasy Shape, ale nie będzie znał jego typu (dla klienta będzie to po postu obiekt typu Shape), zatem dzięki takiej enkapsulacji dodanie jakiegoś nowego kształtu nie sprawi problemu
  - klasa Drawing ukrywa różne swoje pochodne klasy przed obiektami klasy Shape

- **jedna reguła, jedno miejsce** – klasa abstrakcyjna często posiada metody używane przez obiekty klas pochodnych, realizując ideę realizowania danej reguły (zadania) w jednym miejscu

Wzorce projektowe [decorator - dekorator]

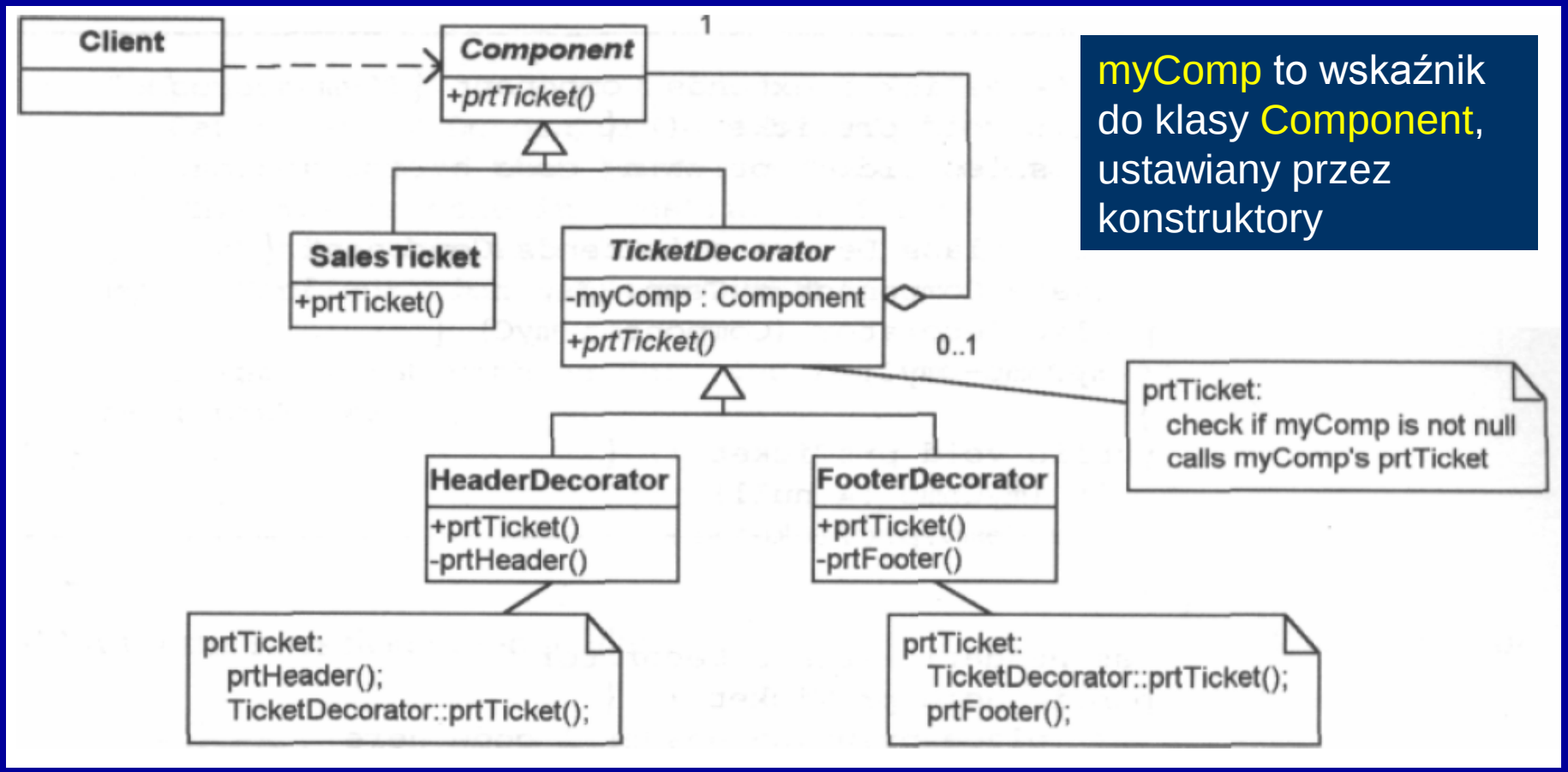
Cel: dynamicznie dodać jakąś funkcjonalność do obiektu
Problem: używany obiekt realizuje podstawową funkcjonalność, którą chcielibyśmy uzupełnić o dodatkową funkcjonalność mającą miejsce przed lub po funkcjonalności własnej tego obiektu
Rozwiązanie: pozwól na zrealizowanie dodatkowej funkcjonalności bez definiowania kolejnych klas pochodnych zawierających taki dodatek
Implementacja: utwórz klasę abstrakcyjną reprezentującą zarówno klasę pierwotną jak i klasę z nowymi funkcjami do niej dodanymi (Decorator), w pochodnych klasy Decorator umieść wywołania dodatkowych metod, przed lub po metodzie pierwotnej



Definicja GoF:
Dekorator pozwala na dynamiczne dodanie dodatkowych odpowiedzialności do obiektu, stanowiąc plastyczną alternatywę dla kreowania pochodnych podklas realizujących dodatkową funkcjonalność.

Wzorce projektowe [dekorator - przykład]

System drukowania biletów, które mają różne (dynamicznie zmienne) nagłówki i stopki. Zasadniczą częścią jest klasa SalesTicket z metodą prtTicket() drukującą bilet bez nagłówka i stopki – te są dodane poprzez klasę dekoratora.



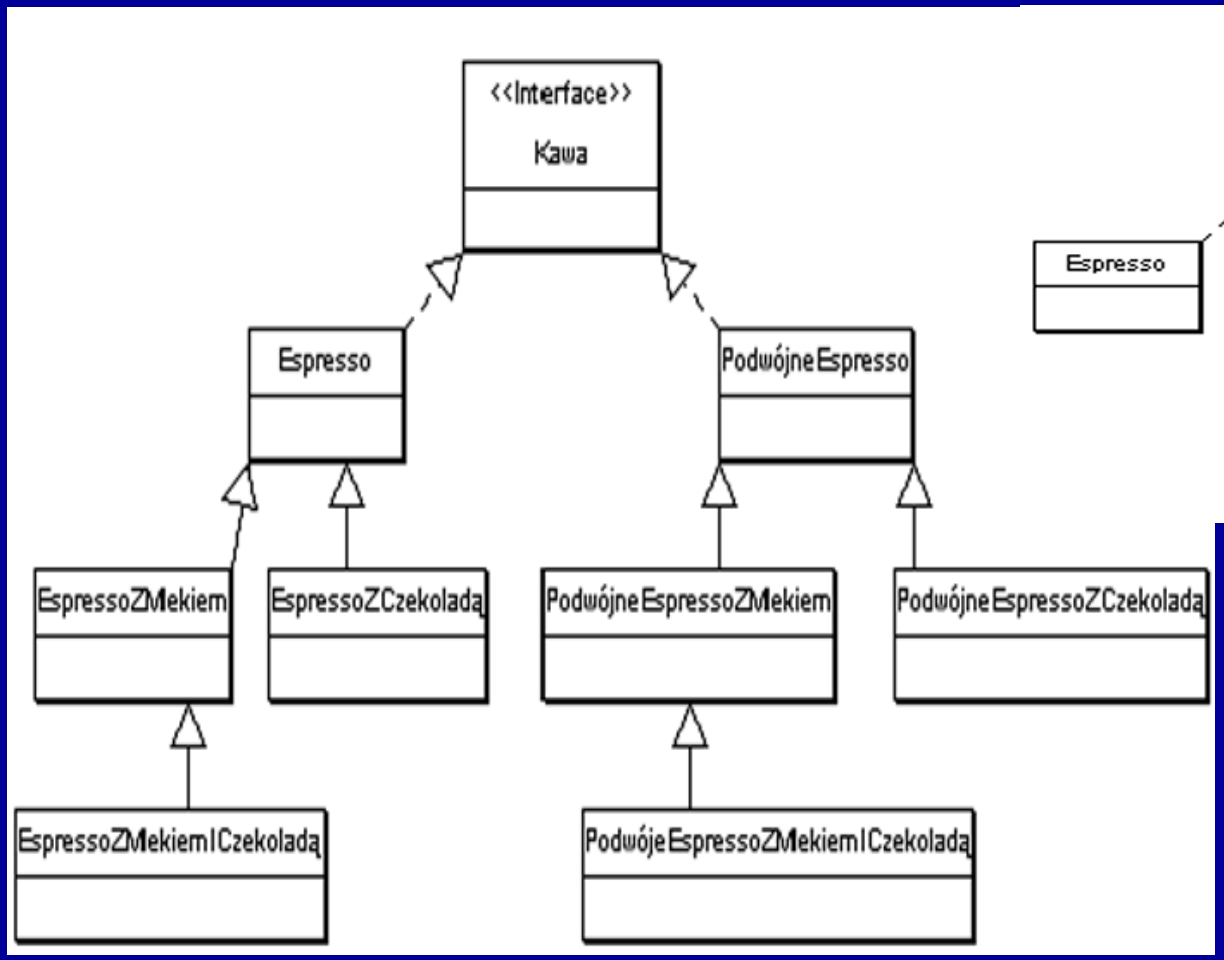
Klient używa wskaźnika do klasy Component. Wybór konfiguracji drukowania, np.:

```
Component *ptr = new HeaderDecorator( new FooterDecorator( new SalesTicket() ) );
ptr->prtTicket();
```

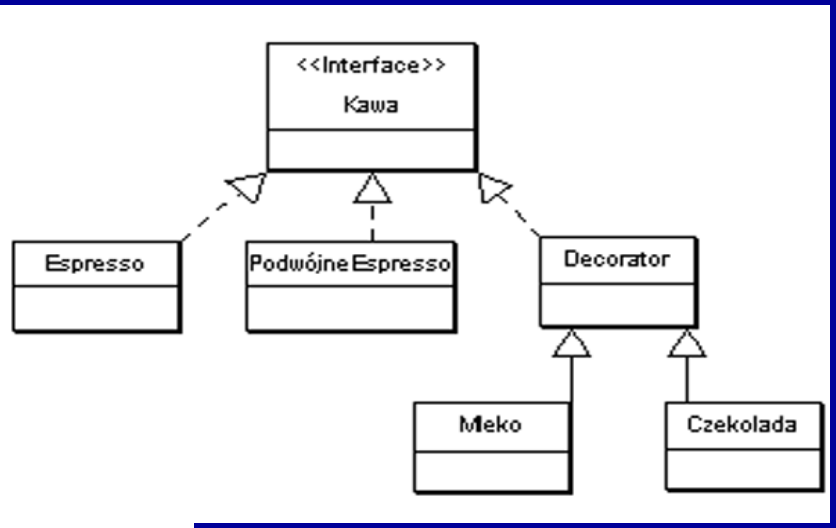
Właściwy obiekt (podlegający „dekorowaniu”) zawsze kończy ten łańcuch.

Wzorce projektowe [dekorator – jeszcze jeden przykład]

kawiarnia – rozwiązanie bez wzorca dekoratora prowadzi do eksplozji kombinatorycznej i kodu trudnego w utrzymaniu i rozwoju…



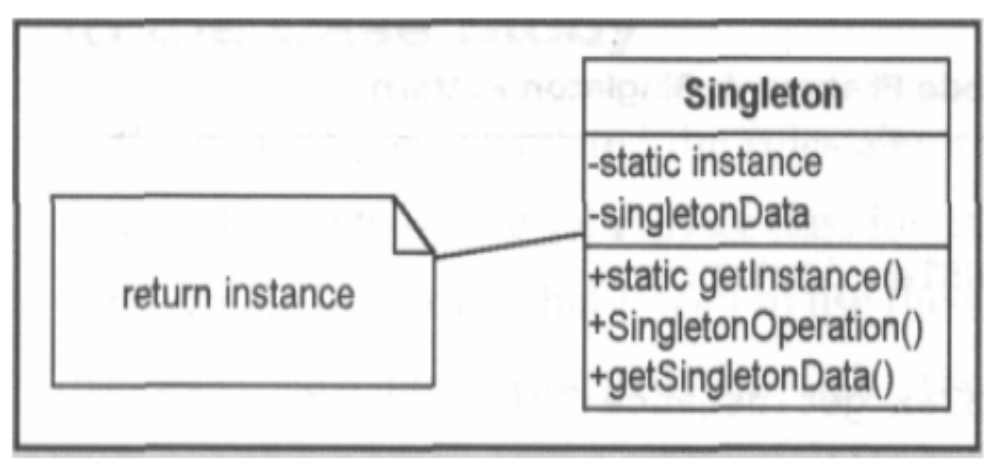
z dekoratorem –



uproszczenie i dynamizacja

Wzorce projektowe [singleton]

Cel: obiekt występujący tylko raz ale nie jako obiekt globalny
Problem: wiele różnych klientów chce mieć dostęp do tej samej rzeczy, a ty chcesz zagwarantować że ta rzecz jest dokładnie jedna
Rozwiązanie: zagwarantuj dokładnie jedną instancję danej klasy (Singletonu)
Implementacja: dodaj do klasy prywatne statyczne pole odwołujące się do pożądanego obiektu (początkowo ustawione na 0), dodaj metodę statyczną konkretyzującą obiekt (jeśli pole statyczne było 0) i / lub zwracającą referencję do obiektu jeśli wcześniej został już utworzony, zabezpiecz konstruktory (czyli uczyń je prywatne lub chronione), żeby nikt nie mógł ominąć statycznej metody powoływania obiektu do życia

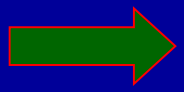


Definicja GoF:
Singleton gwarantuje, że klasa ma tylko jedną instancję (istnieje jeden obiekt – singleton dostarcza globalny sposób dostępu do niego).

Wzorce projektowe [singleton – przykład]

Singleton – w bardziej ograniczonym przypadku klasa może określać, że w działającym programie powinien istnieć tylko jeden jej egzemplarz (jedyny obiekt klasy), przy każdej próbie utworzenia nowego obiektu klasy klient będzie zawsze otrzymywać ten sam obiekt

```
class TClock {
  private:
    TClock();
    static TClock *cp;
  public:
    static TClock* makeObject() {
      if (cp == 0) cp = new TClock();
      return cp;
    }
};
```



```
TClock *TClock::cp = 0;

int main() {
  TClock *cp = TClock::makeObject();
  TClock *cp2 = TClock::makeObject();
}
```

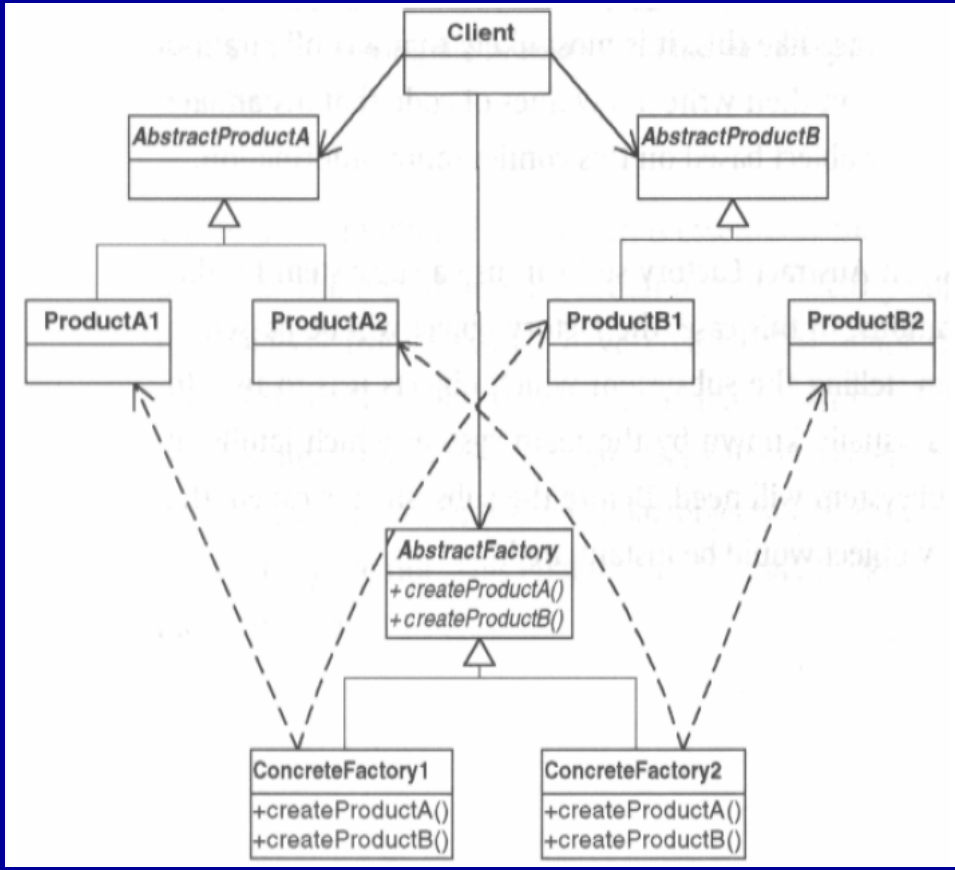
- pierwsze wywołanie funkcji makeObject() i sprawdzenie statycznej składowej równej 0 powoduje utworzenie obiektu
- każda kolejna próba utworzenia obiektu powoduje zwrócenie wskaźnika na ten sam obiekt (skoro już wcześniej został utworzony)

wady

- statyczna dana składowa musi być zdefiniowana / zainicjalizowana przed użyciem
- jakiś inny kod musi być odpowiedzialny za usunięcie obiektu singleton

Wzorce projektowe [abstract factory – fabryka abstrakcyjna]

Cel: grupy obiektów dla pewnych klientów (lub zastosowań)
Problem: grupy obiektów trzeba w jakiś sposób utworzyć
Rozwiązanie: koordynuj tworzenie grupy obiektów dostarczając sposobu ich stworzenia z klienta, który będzie ich używał
Implementacja: zdefiniuj klasę abstrakcyjną specyfikującą, które obiekty mają być tworzone i stwórz po jednej konkretnej klasie dla każdej rodziny (grupy) obiektów

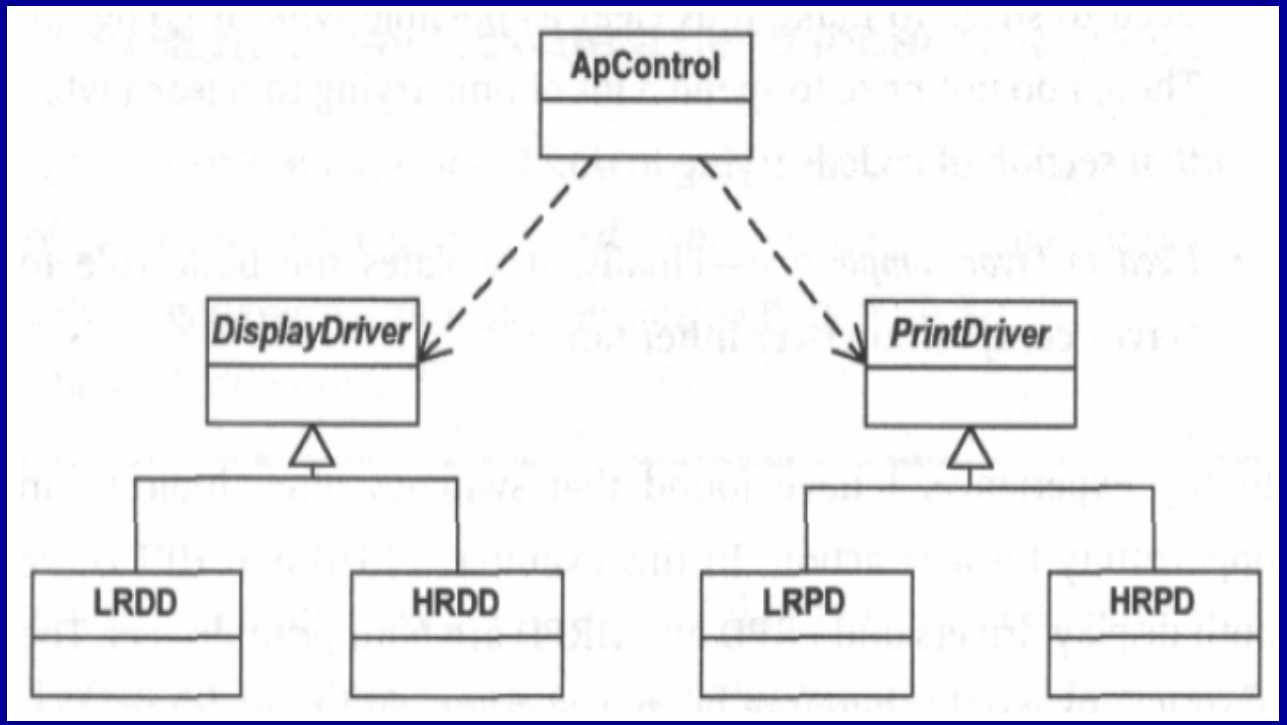


Definicja GoF:
Fabryka abstrakcyjna dostarcza interfejs do tworzenia grup związanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych.

Klient nie wie którego konkretnie obiektu używa (np. ProductA1 czy ProductA2), nie wie nawet której fabryki użył do stworzenia obiektów (np. ConcreteFactory1 – ale widzianej przez niego jako AbstractFactory)

Wzorce projektowe [fabryka abstrakcyjna – przykład]

Mamy napisać program, który będzie wyświetlał i drukował różne kształty pobrane z bazy danych. Program ma kontrolować używanie sterowników: niskiej rozdzielczości i wysokiej rozdzielczości, w zależności od komputera, na którym będzie wykonywany.

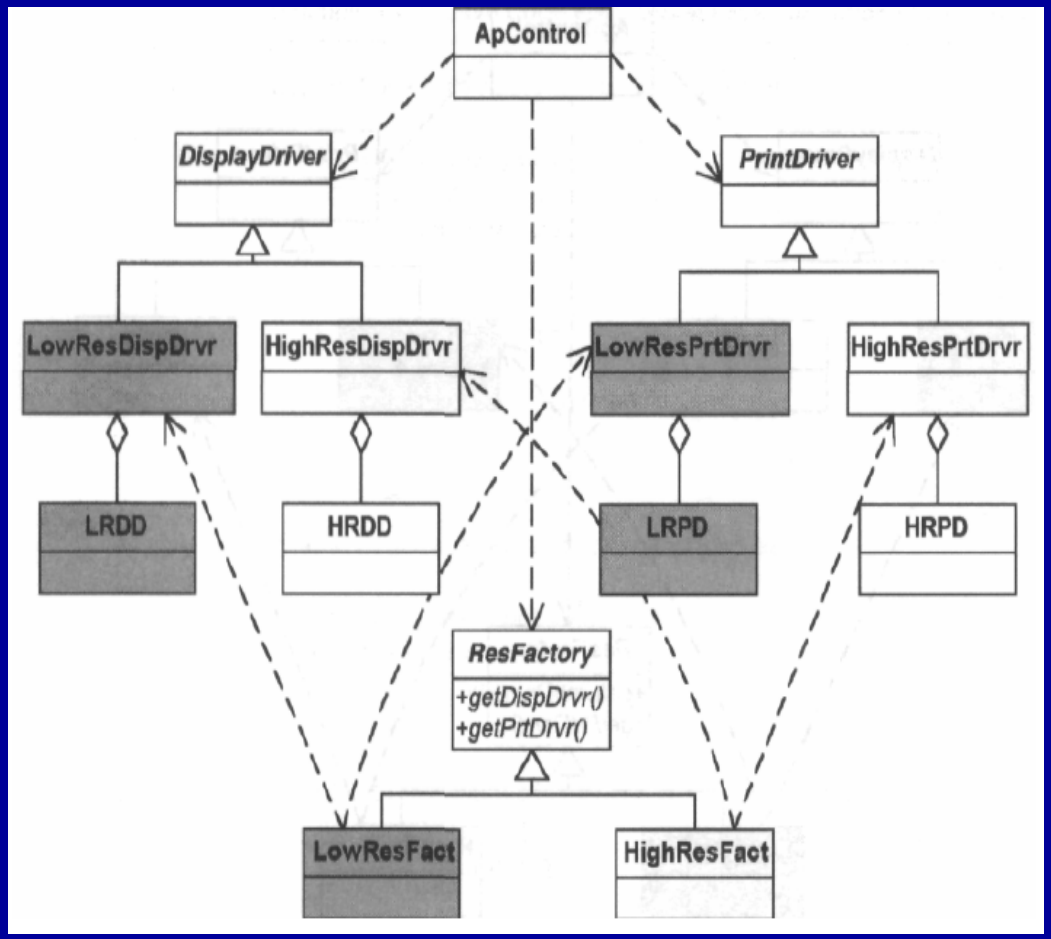


W pierwszym podejściu można by napisać po prostu kod oparty o komendę „switch” i jakieś dane konfiguracyjne… Komenda „switch” nierzadko wskazuje na możliwość wykorzystania abstrakcji: klasami abstrakcyjnymi będą sterowniki ekranu i sterowniki drukarki.

Wadą tego rozwiązania jest to, że każda zmiana pociąga za sobą konieczność zmian w aplikacji kontrolującej ApControl, dlatego wprowadzimy dodatkowy obiekt – fabrykę, dzięki któremu stworzymy właściwy typ sterowników.

Wzorce projektowe [fabryka abstrakcyjna – rozwiązanie]

ApControl odpowiada za właściwą współpracę z właściwymi obiektami (sterownikami). ResFactory odpowiada za to, które obiekty są tymi właściwymi (i tworzy je). ApControl nie musi się martwić tym czy obsługuje sterowniki wysokiej (HR) czy niskiej (LR) rozdzielczości, bo z oboma pracuje tak samo.

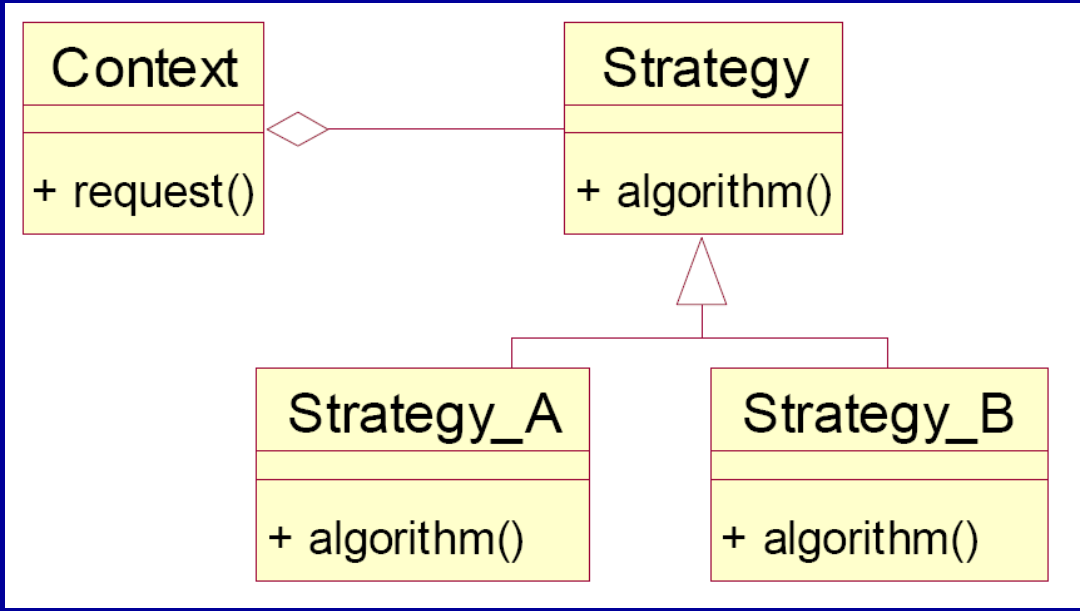


W tym przypadku również często zdarza się, że dostosowanie interfejsów odbywa się poprzez wzorzec Adaptera.

Fabryka abstrakcyjna dokonuje nowego rodzaju dekompozycji – rozdziału odpowiedzialności. Używając jej oddzielamy:
- tego kto używa poszczególnych obiektów (tu ApControl używająca sterowniki) od
- tego kto decyduje który obiekt należy użyć (i tworzy go – tu ResFactory)

Wzorce projektowe [strategy – strategia]

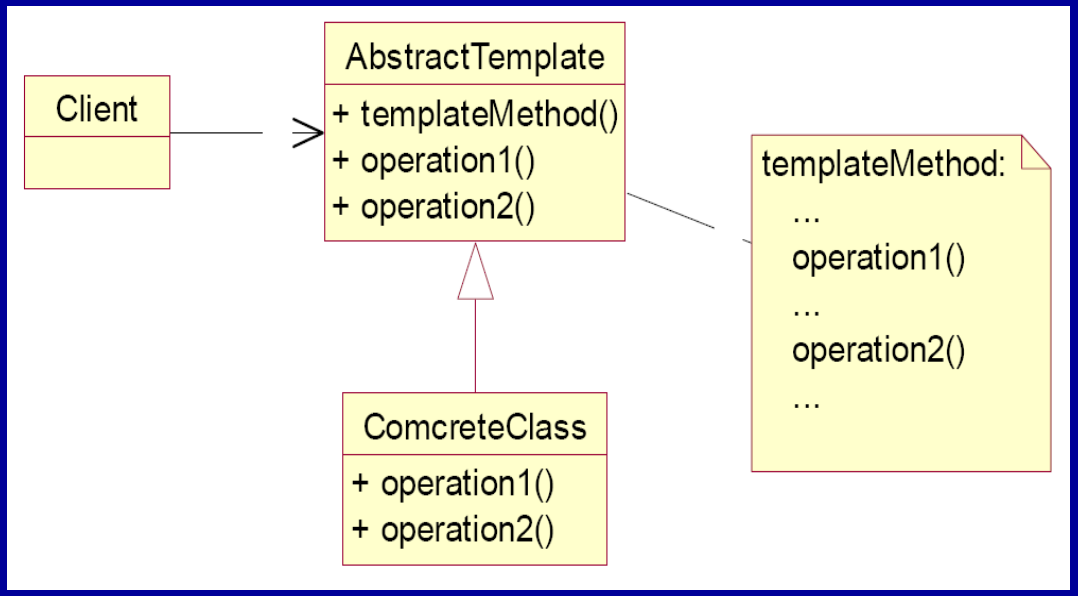
Cel: użycie różnych reguł biznesowych lub algorytmów, w zależności od występującego kontekstu
Problem: wybór potrzebnego algorytmu zależy od zapotrzebowania klienta lub danych, na których się pracuje
Rozwiązanie: odseparuj wybór algorytmu od implementacji algorytmu, wybór algorytmu może nastąpić w zależności od kontekstu
Implementacja: mamy klasę używającą algorytmu (Context), zawierającą klasę abstrakcyjną (Strategy) posiadającą metodę specyfikującą jak wywołać algorytm, każda pochodna klasa implementuje wymagany algorytm



Definicja GoF:
Definiuje rodzinę algorytmów, kapsułkuje każdy z nich i czyni wymiennymi. Strategia pozwala algorytmom na zmiany niezależne od używających je klientów.

Wzorce projektowe [template method – metoda szablonowa]

Cel: mamy kilka różnych procedur postępowania, które wykonuje się zasadniczo tak samo za wyjątkiem tego, że każdy z etapów tych procedur robi te rzeczy w różny sposób, chcemy więc wydzielić szkielet algorytmu
Problem: procedury są realizowane tak samo (mają takie same etapy), choć szczegóły implementacyjne etapów różnią się
Rozwiązanie: określ co jest zmienne na poziomie etapów, wyodrębniając stałość i powtarzalność danej procedury
Implementacja: stwórz klasę abstrakcyjną implementującą procedurę za pomocą abstrakcyjnych metod, metody te skonkretyzowane będą w konkretnych klasach pochodnych



Definicja GoF:
Definiuje szkielet algorytmu danej operacji, odraczając realizację niektórych kroków do podklas, umożliwia redefinicję etapów algorytmu bez zmiany struktury algorytmu.

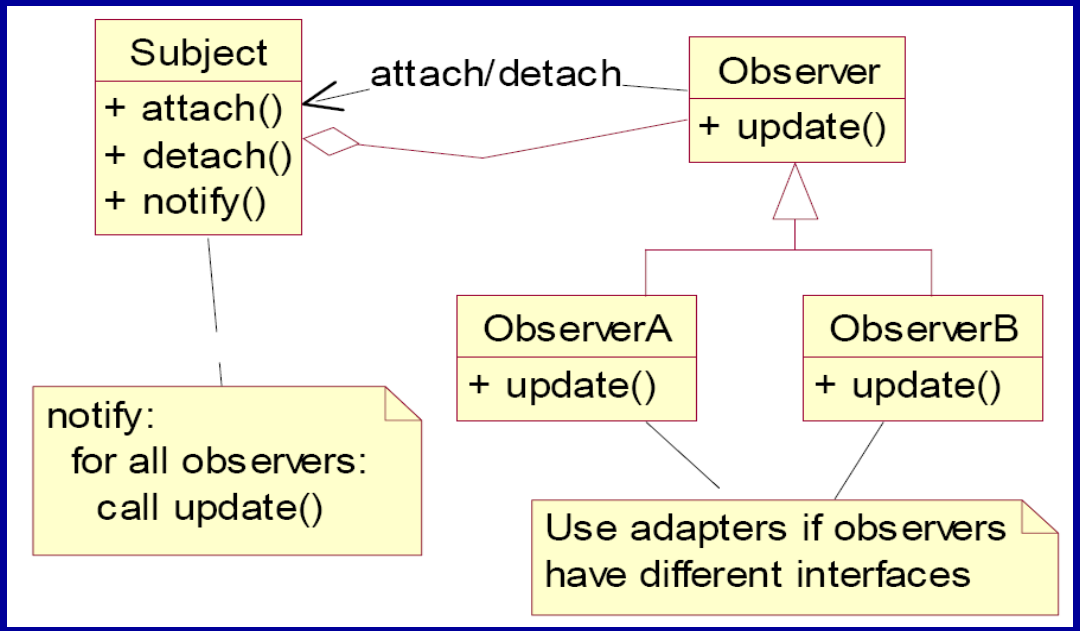
Wzorce projektowe [observer – obserwator]

Cel: obserwujemy stan jednego obiektu a gdy ten się zmieni, wysyła informację do wszystkich obiektów zarejestrowanych jako obserwatorzy w celu wywołania metody aktualizującej je

Problem: potrzebujemy zawiadomić zmienną ilość (listę) obiektów jeśli zaszło jakieś zdarzenie (zmiana stanu obserwowanego obiektu)

Rozwiązanie: obserwatorzy delegują odpowiedzialność za monitorowanie zdarzenia do centralnej klasy Subject

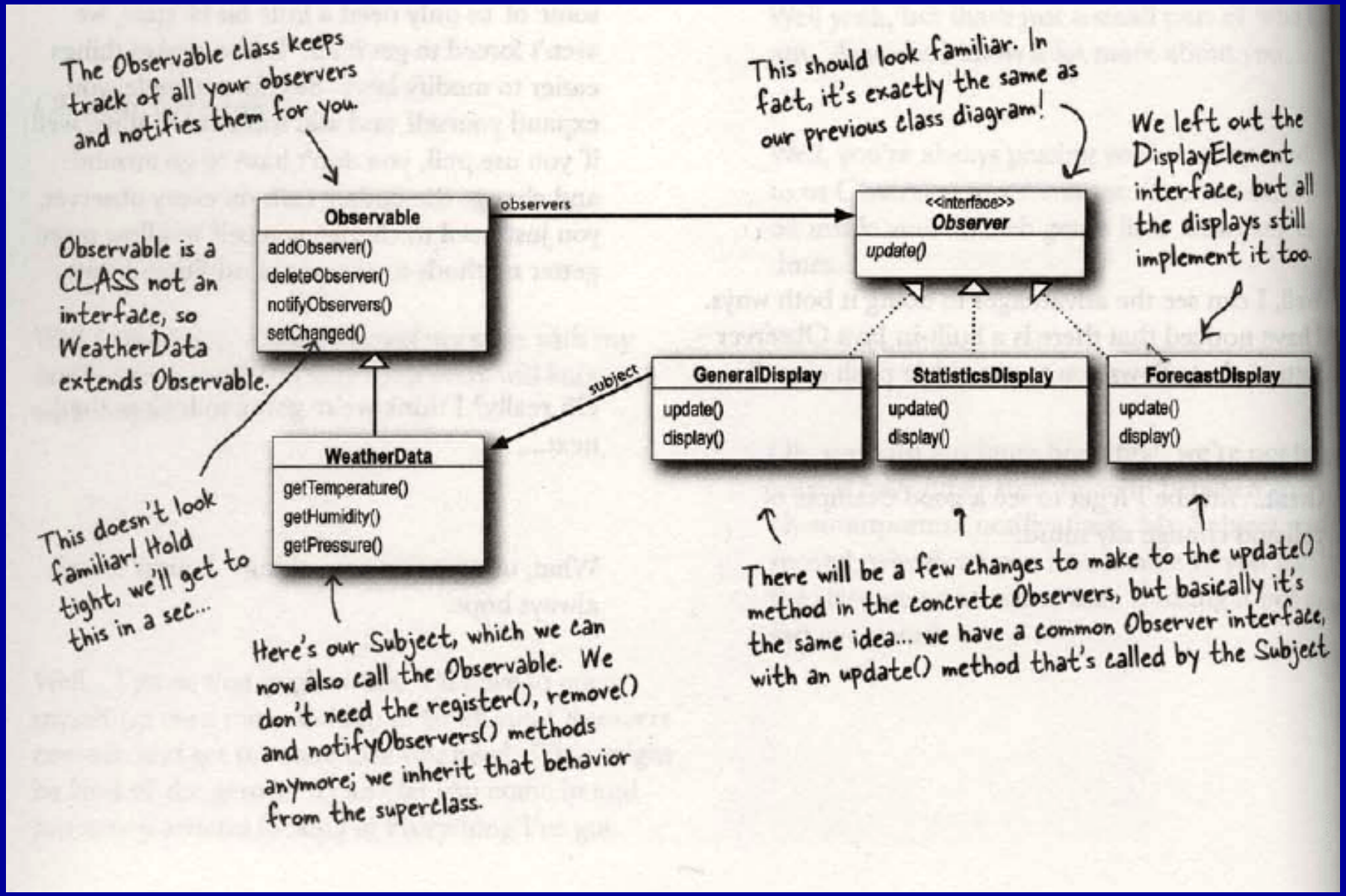
Implementacja: obserwatorzy są na liście posiadanej przez klasę Subject i kiedy nastąpi zdarzenie, następuje iteracja po liście wszystkich obserwatorów (wskaźniki do nich) i wywołanie ich metody update()



Definicja GoF:

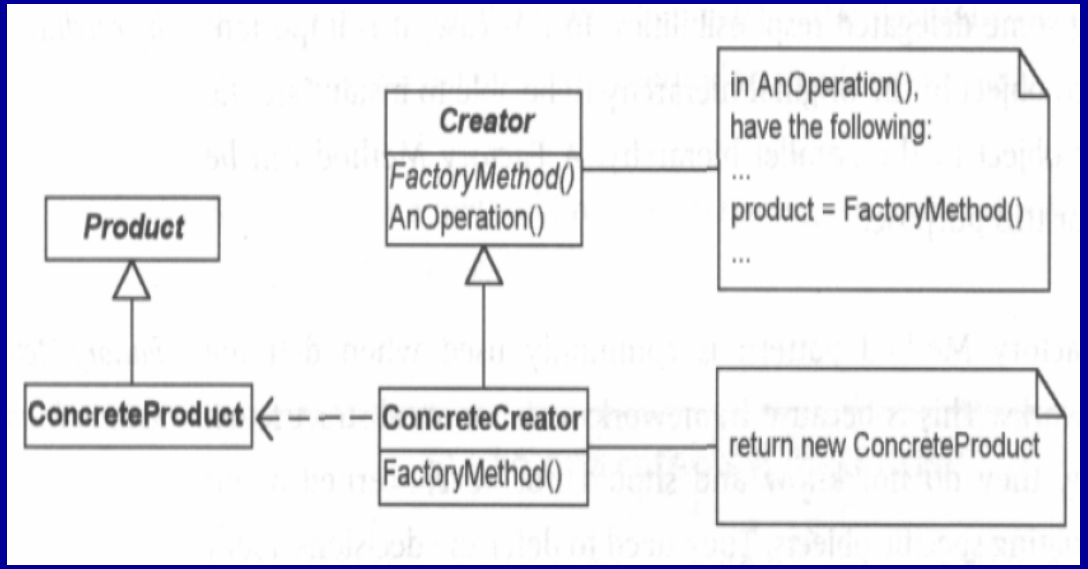
Definiuje zależność wielu od jednego tak, że jeśli jeden obiekt zmienia stan, wszystkie zależne od niego są o tym poinformowane i odpowiednio zaktualizowane.

Wzorce projektowe [observer – przykład]



Wzorce projektowe [factory method – metoda fabryczna]

Cel: zdefiniujmy interfejs do tworzenia obiektu, ale klasy konkretne zdecydują którą klasę zrealizować w postaci obiektu
Problem: klasa chce stworzyć obiekt innej klasy, ale nie wie której, decyzję tą podejmuje klasa konkretna
Rozwiązanie: klasa pochodna podejmuje decyzję obiekt której klasy zrealizować i w jaki sposób to zrobić
Implementacja: w klasie abstrakcyjnej Creator użyj metody czysto wirtualnej, która ma zrealizować utworzenie obiektu, ale nie posiada wiedzy na temat tego jaki obiekt jest wymagany



Definicja GoF:
Definiuje interfejs do tworzenia obiektu, ale zleca jego tworzenie podklasom podjęcie decyzji obiekt której klasy utworzyć.