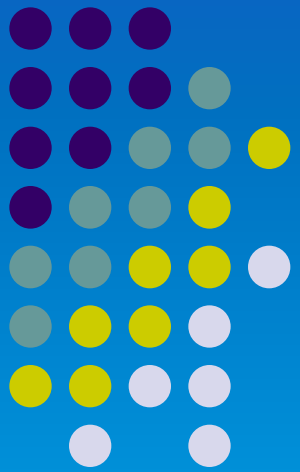
Projektowanie obiektowe
Wzorce projektowe



Gang of Four
Strukturalne wzorce projektowe
(Wzorce interfejsów)

Roadmap

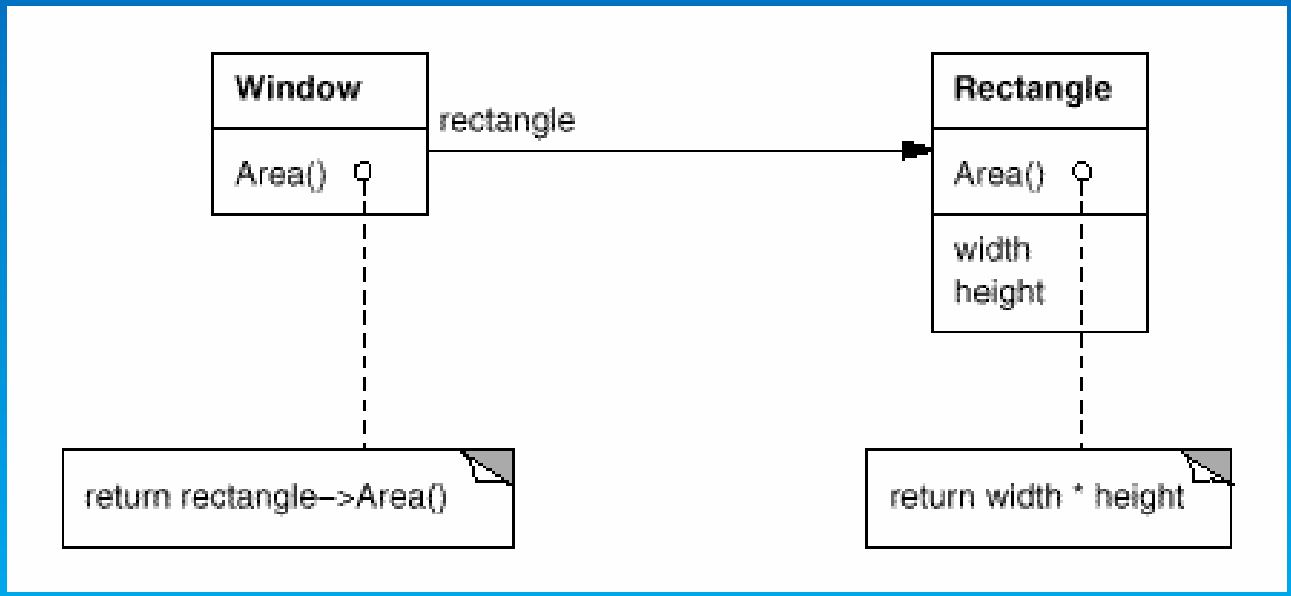
- Adapter
- Bridge
- Composite
- Facade

Pojęcia

- **obiekt …**
- **interfejs …**
- **typ …**
- **klasa …**

Co to jest delegacja?

- Po prostu: przekazywanie (delegowanie) żądania (operacji) przez obiekt odbierający komunikat do realizacji przez inny obiekt (tzw. delegat)
- Zwiększenie ponownego użycia poprzez zastosowanie agregacji zamiast dziedziczenia

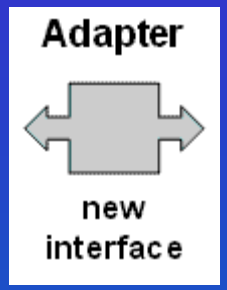


Wzorce strukturalne
Wzorce interfejsów

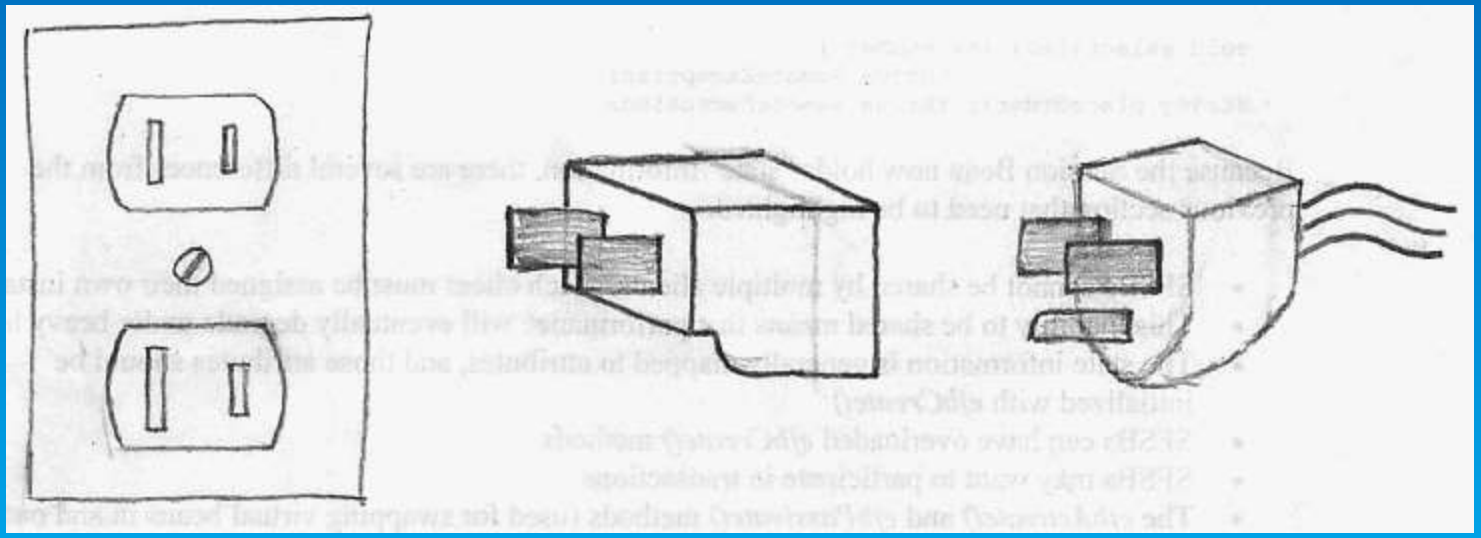
- ***wzorce strukturalne*** dotyczą powszechnych sposobów organizacji obiektów różnego typu, aby mogły współpracować ze sobą.

- ***wzorce interfejsów*** (**adapter**, **bridge**, **composite**, **facade**) dotyczą problemów wymagających zdefiniowania lub przedefiniowania dostępu do klasy lub grup klas

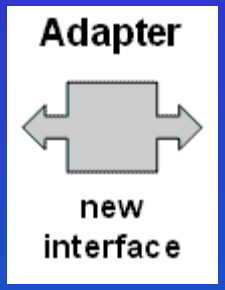
Adapter



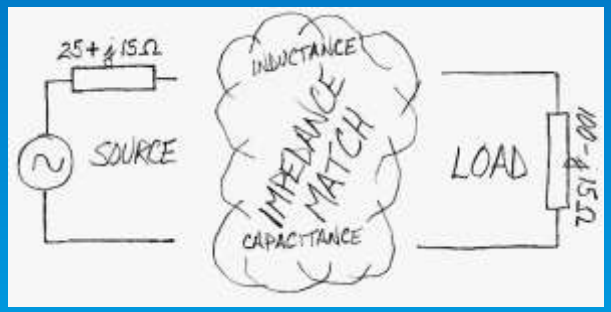
- Zaadoptowanie istniejącego interfejsu klasy do postaci oczekiwanej przez klienta.



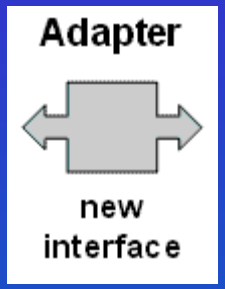
Adapter - Problem



- Niepowiązane klasy, komponenty rozwijane w różnym czasie lub równolegle mają ze sobą współpracować.
- Dobry program, do którego kodu nie mamy dostępu, ma działać i być odpowiednio wykorzystany .
- Niekompatybilny interfejs.
- Problem niezgodności impedancji (impedance mismatch).

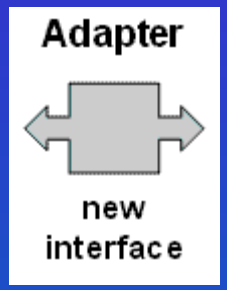


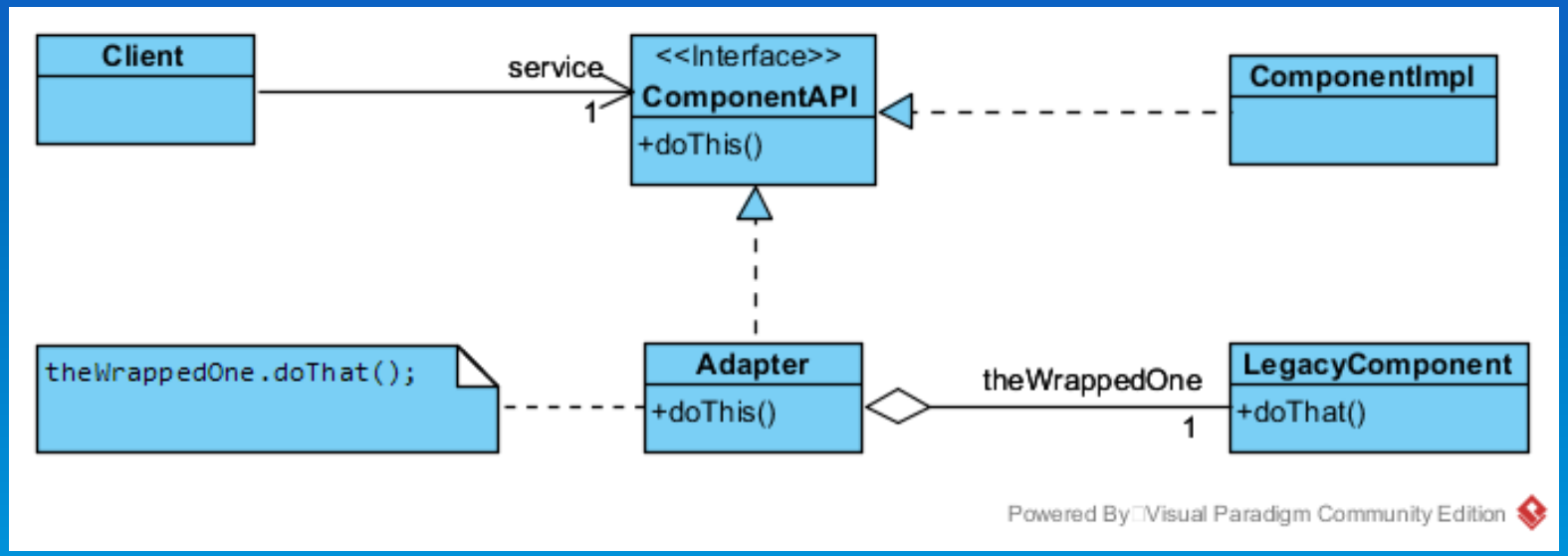
Adapter - Rozwiązanie



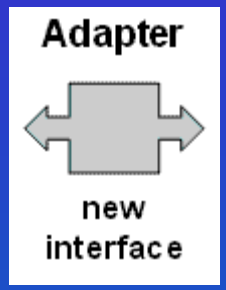
- Osłaniamy istniejący kod nowymi interfejsami.
- Dopasowujemy impedancje starych komponentów do nowego systemu (*często rozwiązanie pozorne! – tj. jak przyszycie starej łaty do nowych spodni*)
- Struktura: Osłona/Delegacja.

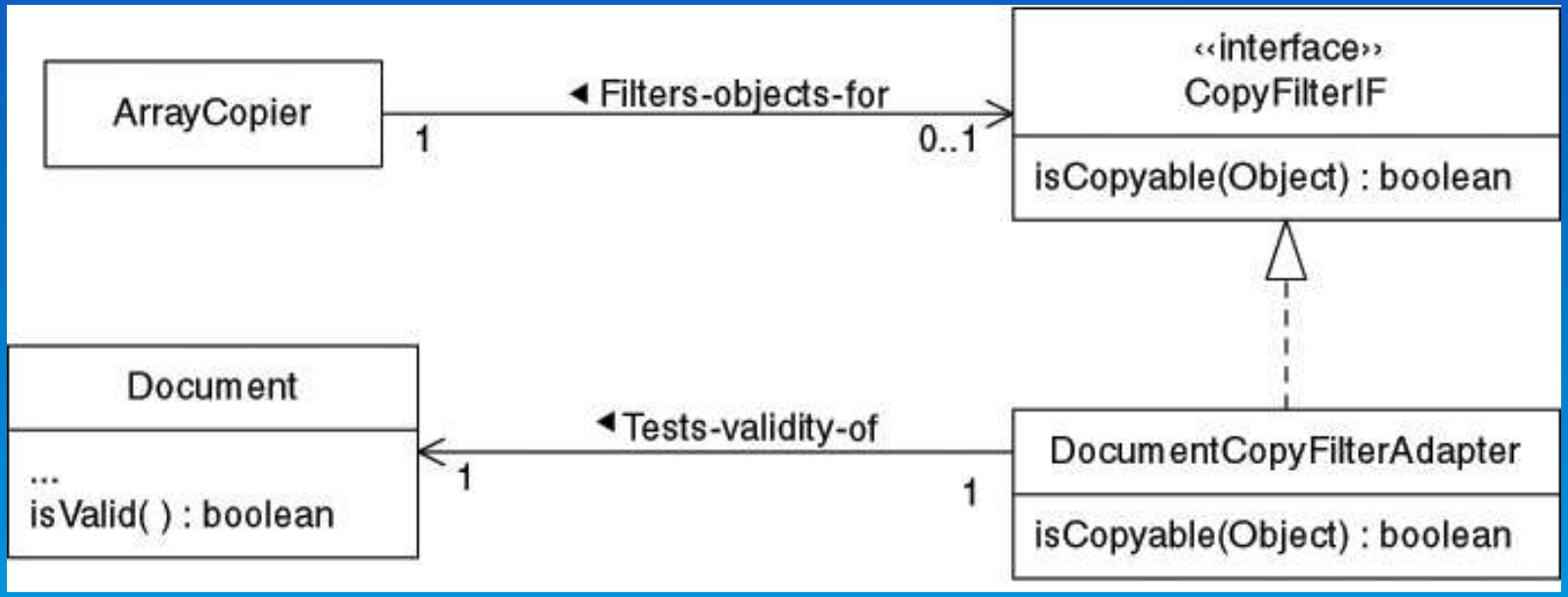
Adapter – diagram klas



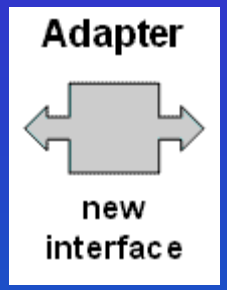


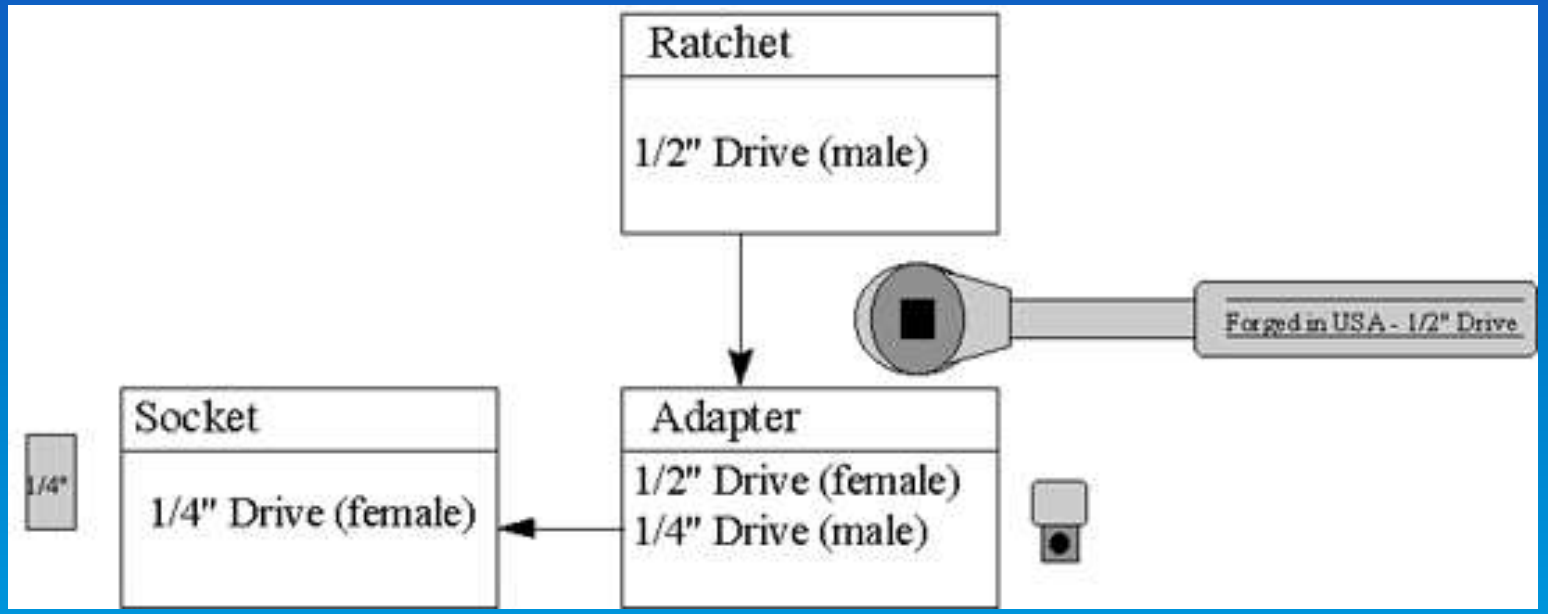
Adapter – diagram klas (przykład)



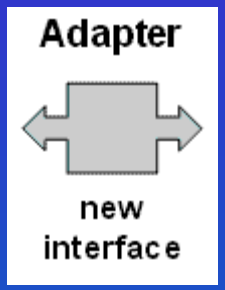


Adapter – przykład



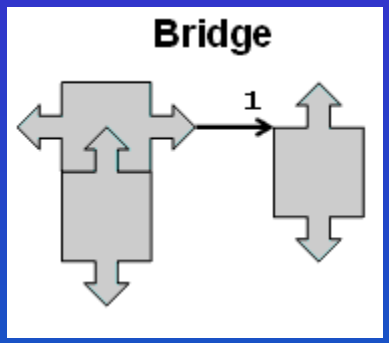


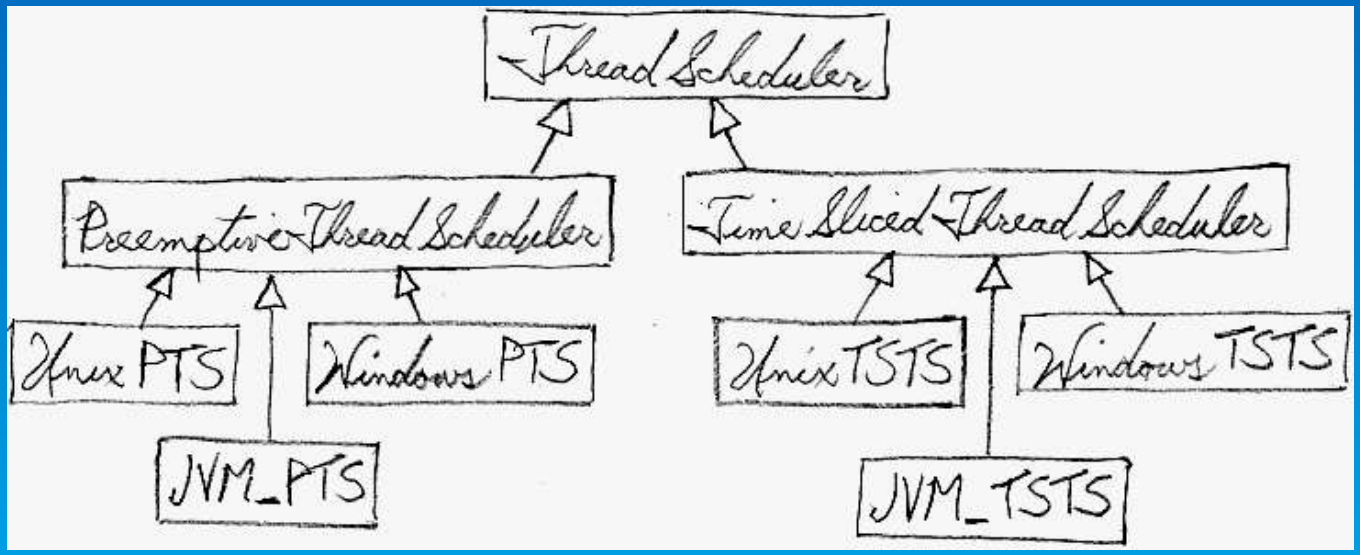
Adapter - Konsekwencje



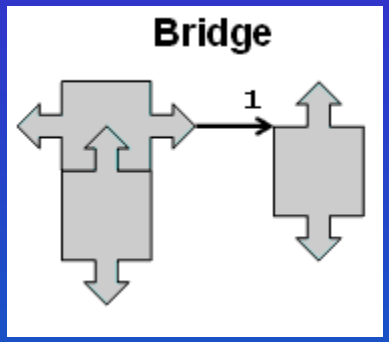
- Klient i adaptowany komponent (klasa, metoda, itp.) pozostają niezależne.
- Można używać klas adaptera do określenia jaka metoda obiektu ma być wywołana przez klienta (*np. jeden adapter wywołuję metodę rysującą linię ciągłą, a drugi wywołuję metodę rysującą linię przerywaną*)
- Adapter dodaje warstwę pośrednią w programie:
  - negatywny wpływ na wydajność,
  - trudność zrozumienia aplikacji.

Bridge – wprowadzenie do problemu





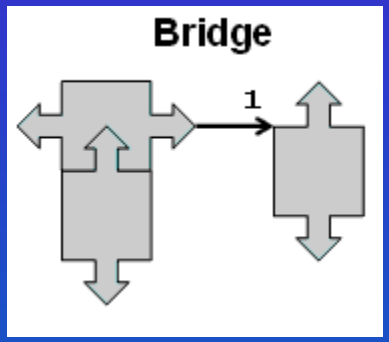
Bridge



- Oddzielenie operacji abstrakcyjnych od ich implementacji w celu umożliwienia wprowadzenia w nich niezależnych zmian

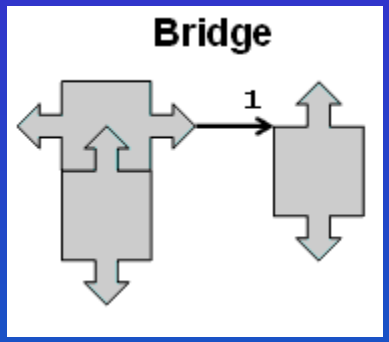
- Użyteczny, gdy mamy do czynienia z hierarchią abstrakcji i odpowiadającą hierarchią implementacji, w celu ich rozłączenia.

Bridge - Problem



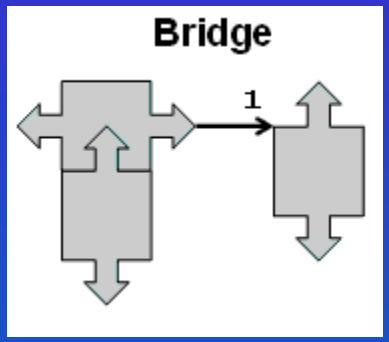
- Brak odseparowania implementacji od interfejsu.
- Potrzeba poprawienia możliwości rozbudowy klas, zarówno implementacji, jak i interfejsu (m.in. przez dziedziczenie),
- Potrzeba ukrycia implementacji od klienta, w celu umożliwienia zmiany implementacji bez zmian interfejsu
- Kilka odpowiednich abstrakcji ma korzystać z tej samej implementacji

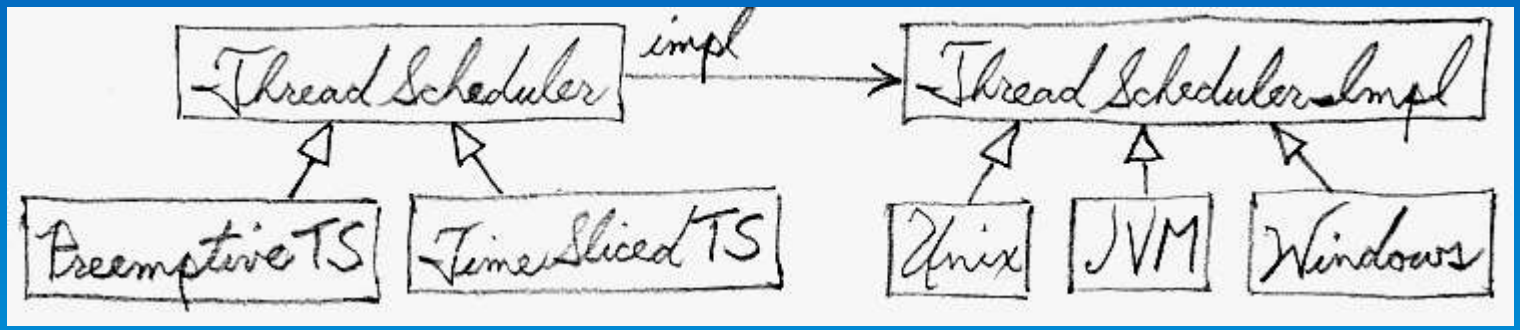
Bridge - Rozwiązanie



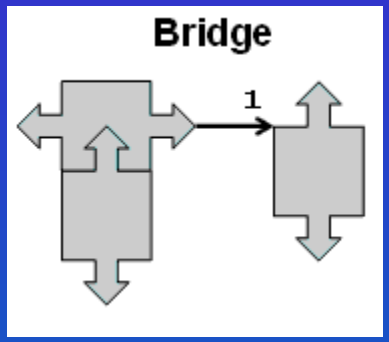
- Pozwolenie na zmiany implementacji oraz pozostawienie stabilnego interfejsu.
- Struktura: Osłona/Delegacja
  - osłona to hierarchia, która dostarcza interfejs
  - delegacja to hierarchia, która ukrywa bagaż implementacji
- Izolacja: *koperta/list*, (więcej niż enkapsulacja)

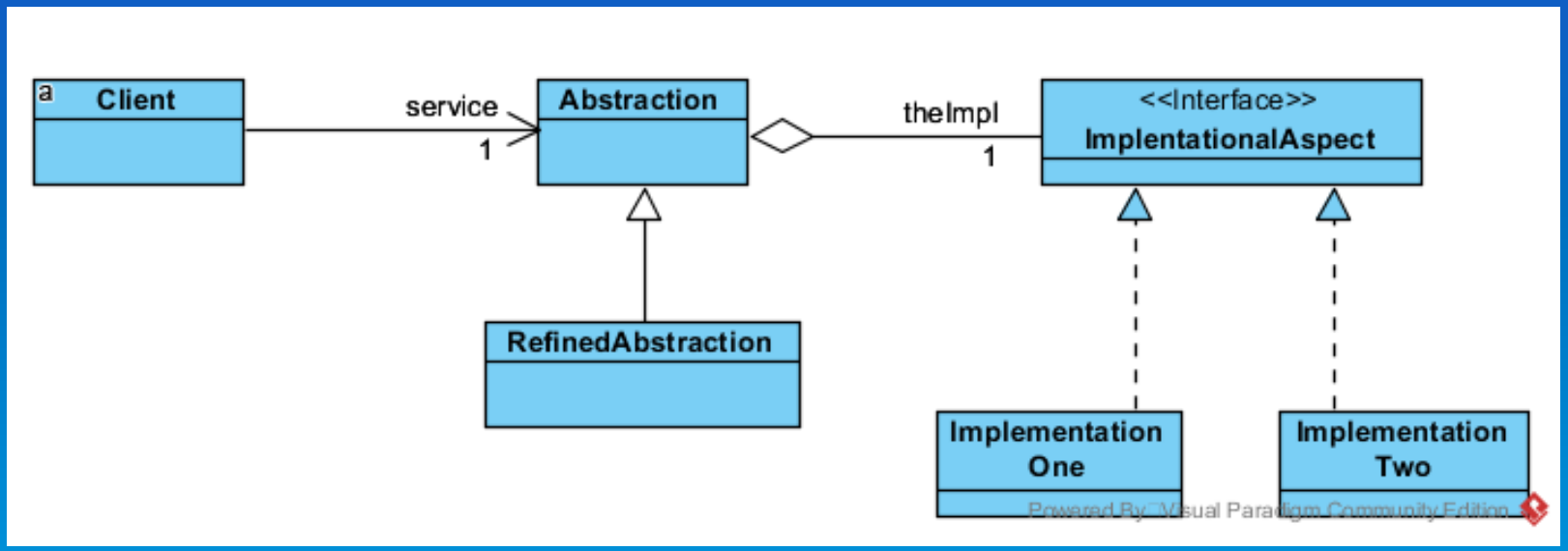
Bridge – przykład (szeregowanie wątków)





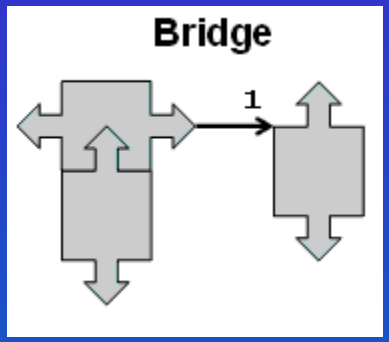
Bridge – diagram klas





np. sterowniki urządzeń i baz danych

Bridge - Konsekwencje



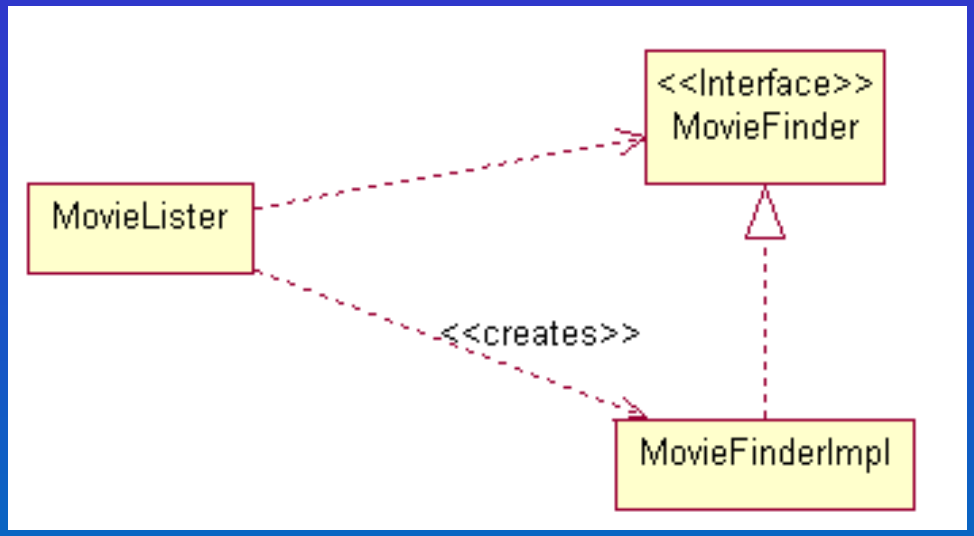
- Bridge utrzymuje niezależność między klasami reprezentującymi abstrakcje i dostarczającymi implementacje abstrakcji.
- Abstrakcja i jej implementacje mają osobną hierarchie klas, które można rozszerzać bez wzajemnego wpływu.
- Można mieć wiele klas implementujących dla abstrakcyjnej klasy lub wiele abstrakcji używających tej samej implementacji.
- Obiekty abstrakcji mogą zmieniać implementacje bez wpływu na klienta.

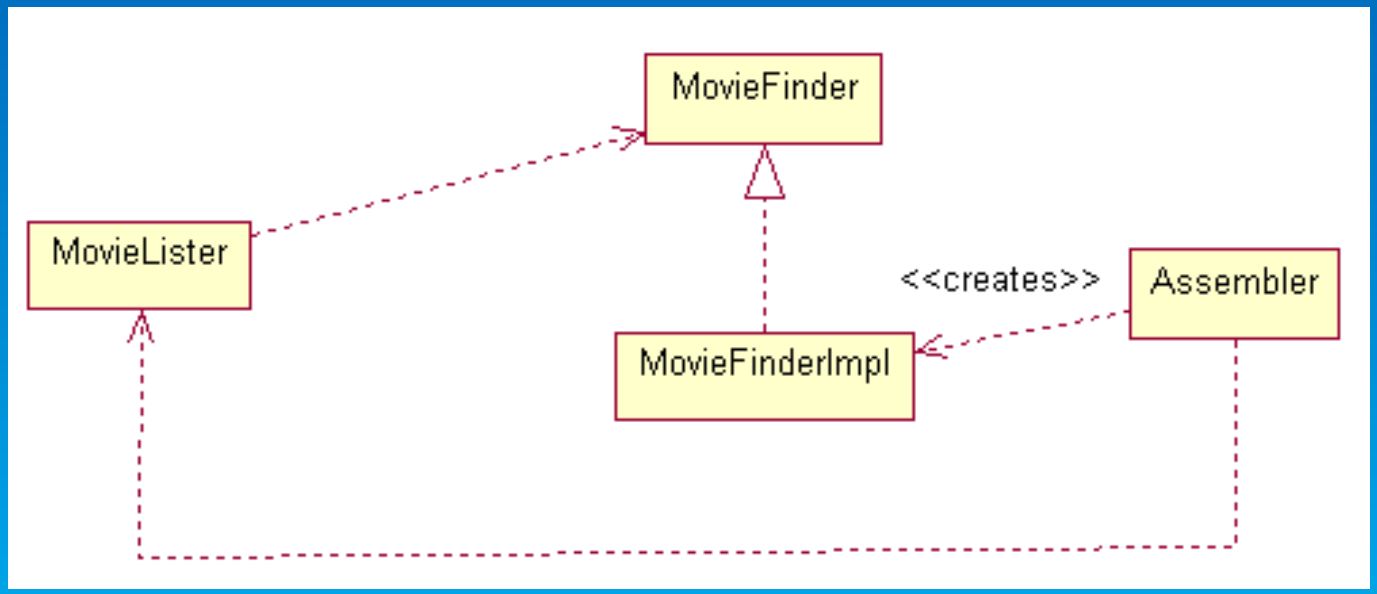
Konfiguracja na przykładzie Spring Framework

- **Inversion of Control (Hollywood Princilple) –**

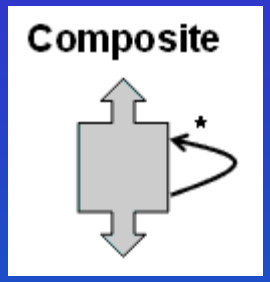
"don't call us, we will call you„.

  - **Framework konfiguruje aplikacje i woła komponenty użytkowe.**
  - **To tak na dobrą sprawę odróżnia framework od biblioteki.**
- **Depencendy Injection – zmniejszenie zależności pomiędzy komponentami tylko do interfejsów.**


<<Interface>>
MovieFinder
MovieLister
<<creates>>
MovieFinderImpl

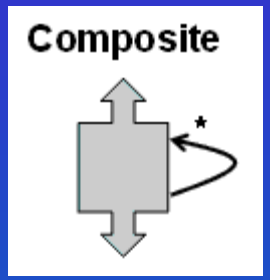

MovieFinder
MovieLister
<<creates>>
Assembler
MovieFinderImpl

Composite



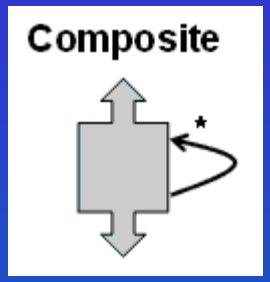
- Zdefiniowanie interfejsu uwzględniającego zarówno pojedyncze obiekty, jak i grupy obiektów.
- Umożliwienie odnoszenia się tak samo do pojedynczych obiektów, jak do *kompozytów*

Composite - Problem



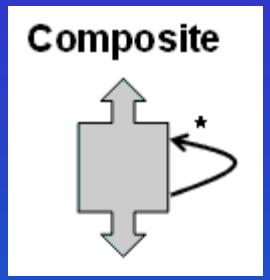
- Dekompozycja złożonego obiektu na hierarchię obiektów część-całość
- Klient nie powinien rozróżniać między kompozycją wielu obiektów, a pojedynczym obiektem.
- Wiele różnych obiektów jest używanych w podobny sposób i mają prawie identyczny kod obsługi.

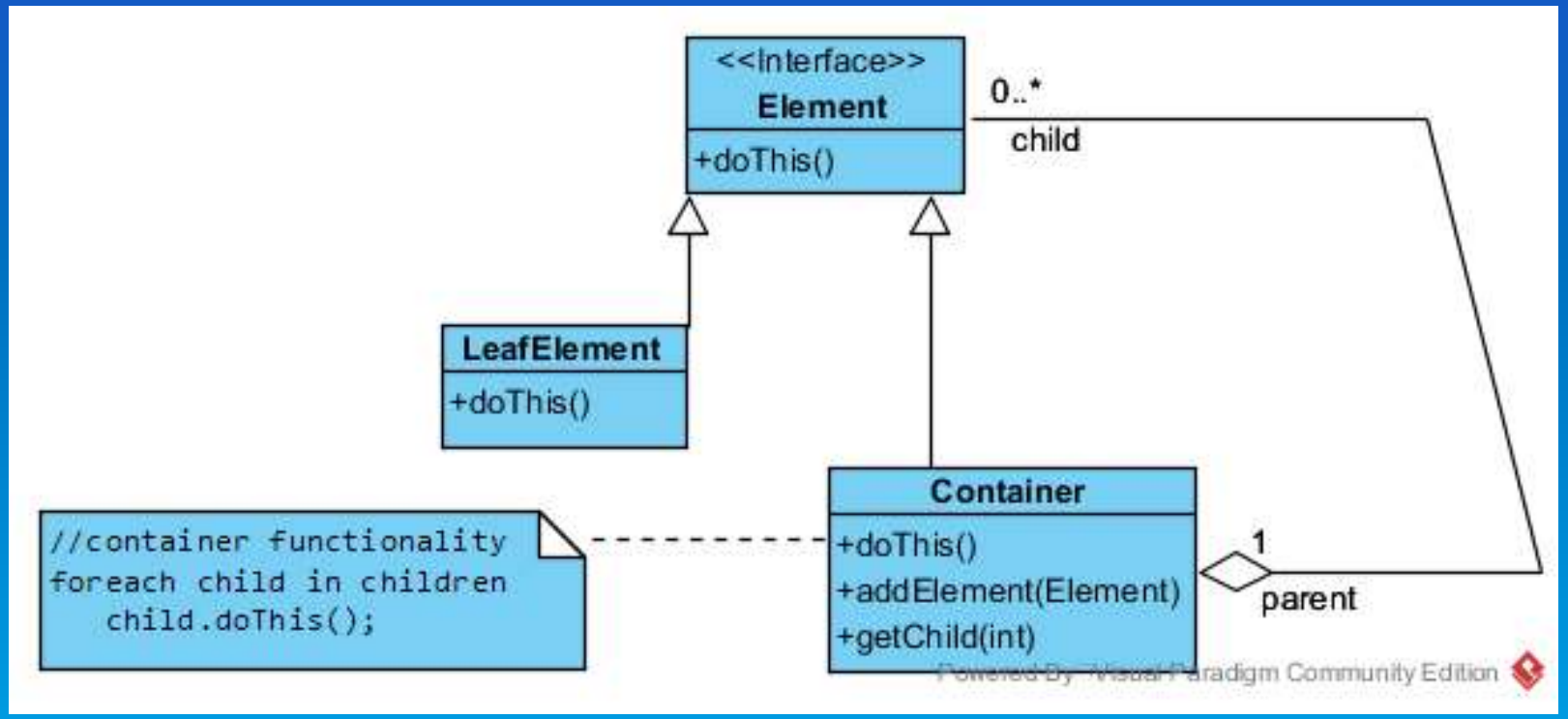
Composite - Rozwiązanie



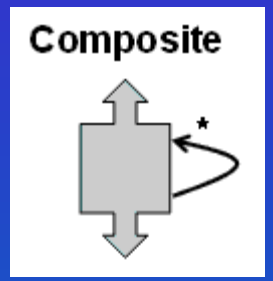
- Zastosowanie rekursywnej kompozycji.
- Agregacja 1 do wielu: „składa się” w górę hierarchii dziedziczenia.

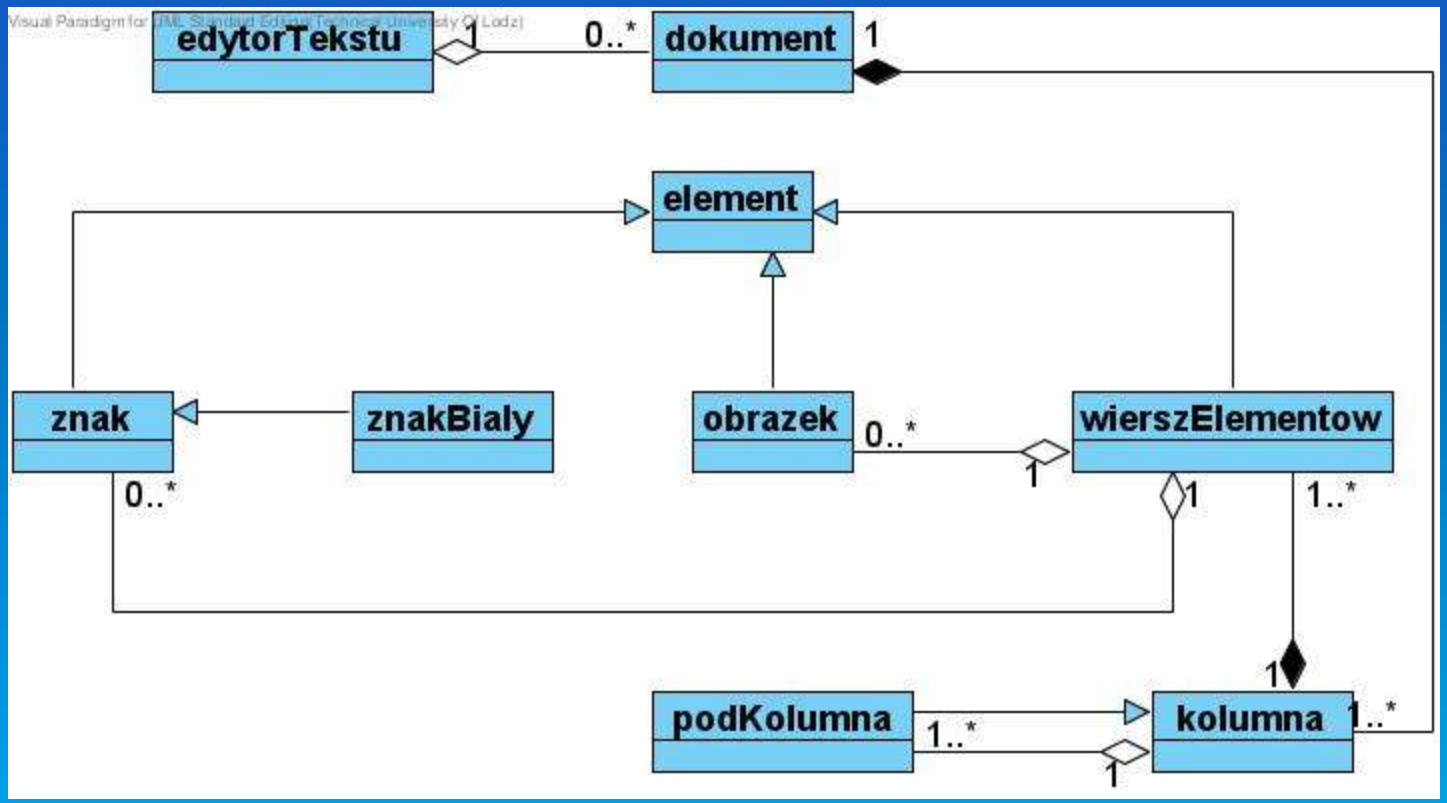
Composite – diagram klas



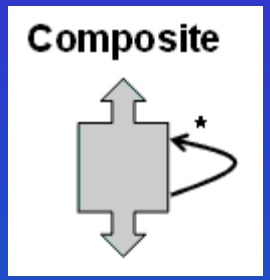


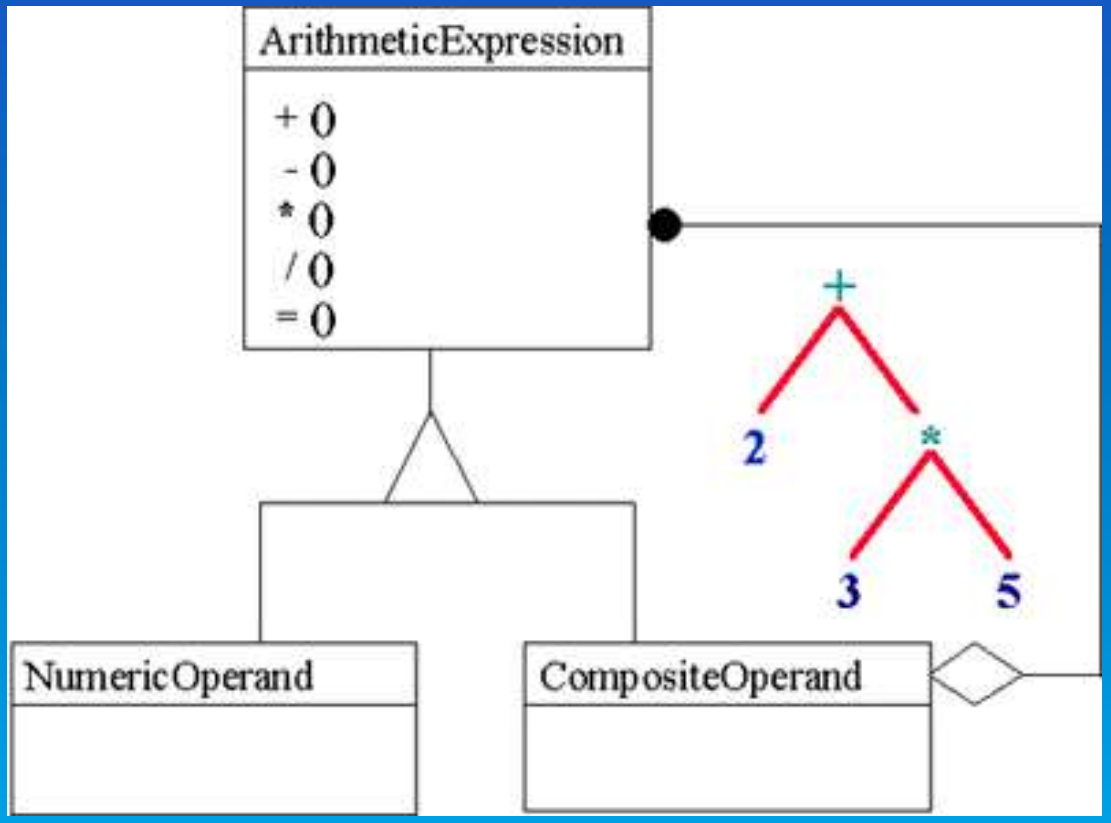
Composite – diagram klas (przykład)



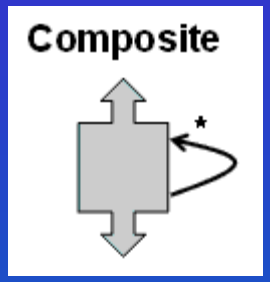


Composite – przykład



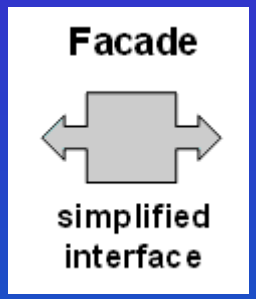


Composite - Konsekwencje



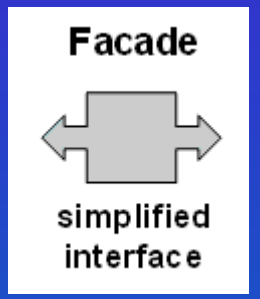
- Udostępnienie wspólnego interfejsu do obiektów składających się na drzewiastą strukturę.
- Klienta nie musi interesować hierarchia komponentów.
- Komponenty mogą rekurencyjnie delegować przetwarzanie żądanie klienta w dół lub górę hierarchii.
- Konkretne komponenty mogą implementować własne unikalne operacje (*ale dla uproszczenia można je przesuwać do klas ogólniejszych*)
- Dowolna klasa wzorca Composite może być dzieckiem obiektu kompozytu. Wprowadzenie zaostrzonych reguł wymaga kodu świadomego występujących typów.

Facade



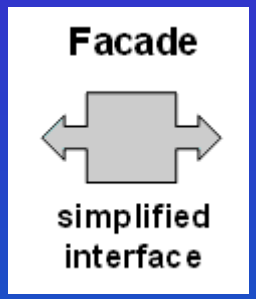
- Udostępnienie prostego interfejsu ułatwiającego korzystanie z zestawu klas lub podsystemu.
- Dostarczenie jednego obiektu na zewnątrz w celu umożliwienia komunikacji z zestawem klas.

Facade - Problem



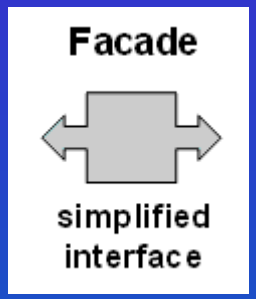
- Istnieje wiele zależności między klasami implementującymi abstrakcje i klasami klienta, zwiększając zauważalnie jego złożoność.
- Potrzeba uproszczenia klienta (np. zmniejszenie ryzyka błędów).
- Zwiększenie stopnia ponownego użycia systemu lub biblioteki.
- Zaprojektowanie klas by działały w jasno odseparowanych warstwach.

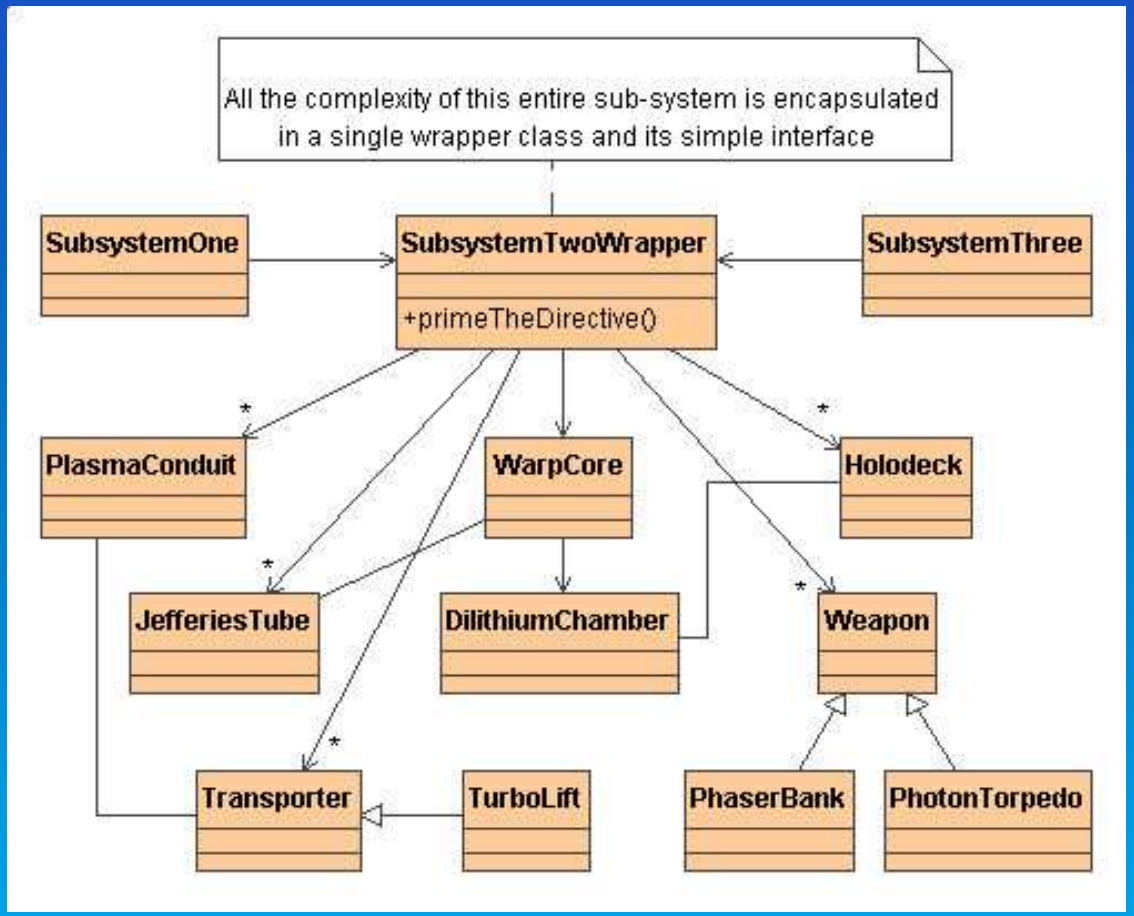
Facade - Rozwiązanie



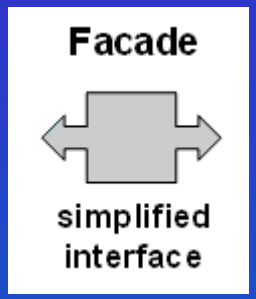
- Osłonienie istniejącego systemu nowym interfejsem.
- Prosty punkt wejścia dla dużego podsystemu.
- Dodanie warstwy pośredniczącej ukrywającej złożoność spadkowego systemu (legacy)

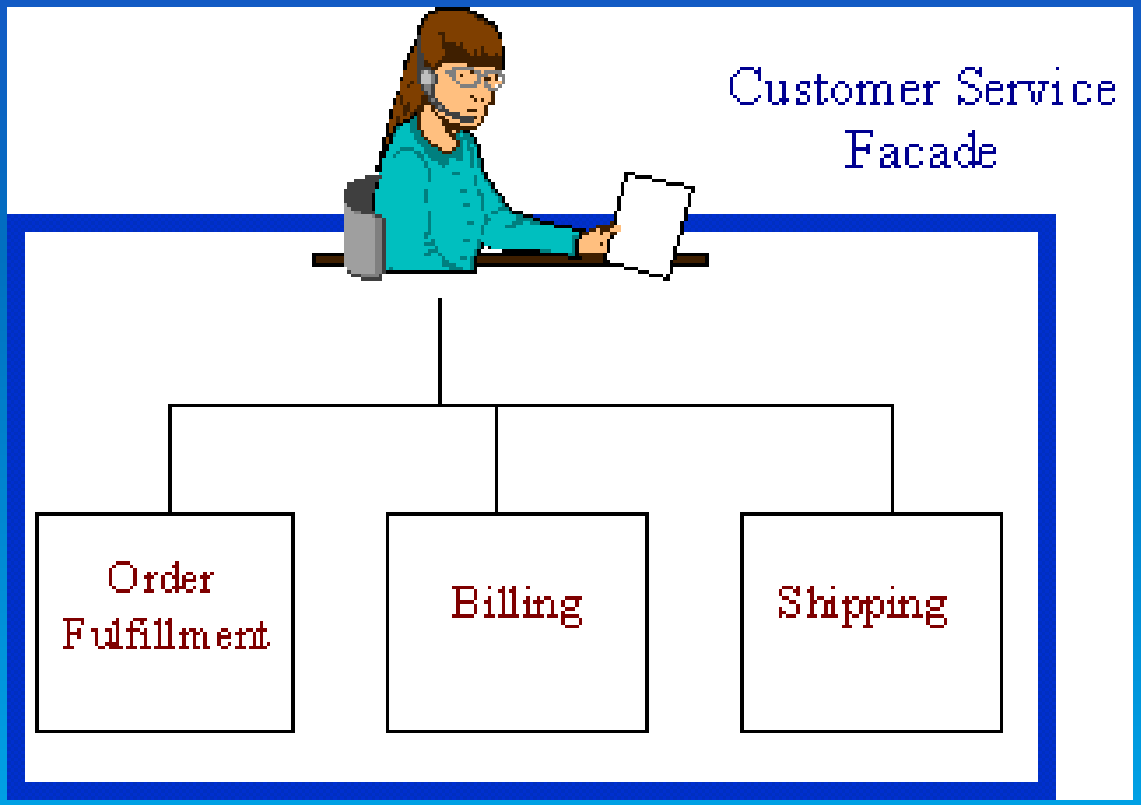
Facade – diagram klas



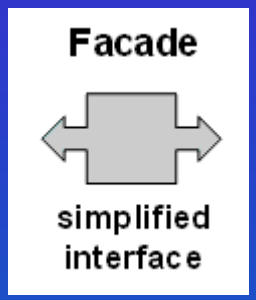


Facade – przykład



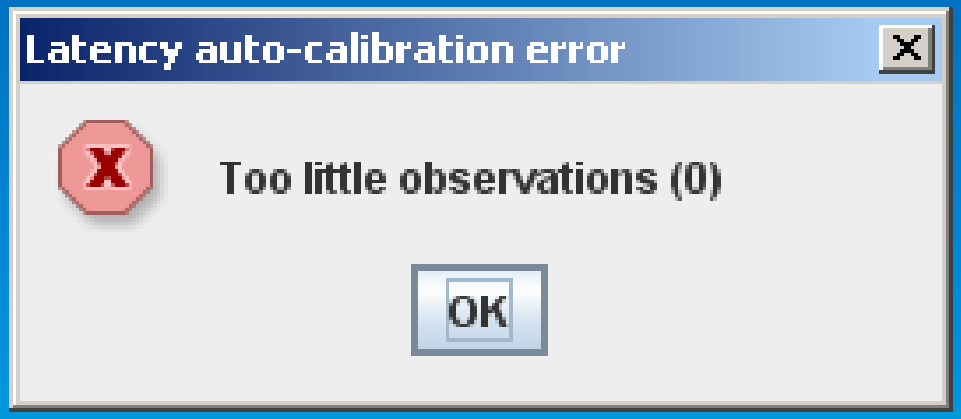


Facade – przykłady

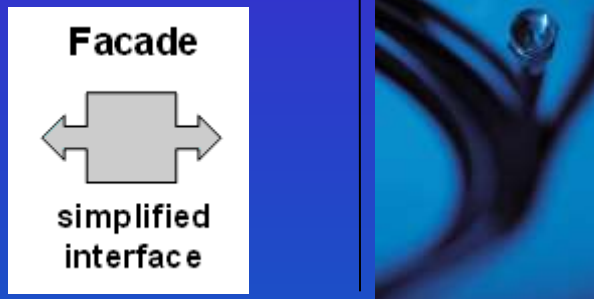


Java – klasa *JOptionPane* pakietu javax.swing

C# - klasa *MessageBox* pakietu System.Windows.Forms



Facade - Konsekwencje



- Wstawienie klas fasady upraszcza klasy klienta przesuwając zależności z klienta do fasady.
- Klient nie musi znać klas za fasadą.
- Zmiana implementacji (np. poprawa) klas znajdujących się za fasadą, czyli tych, które implementują abstrakcje, jest możliwa bez wpływu na kod klienta.

Zależności między wzorcami

- Umożliwienie wykorzystania elementów:
  - już gotowych – ***Adapter***,
  - przed ich powstaniem – ***Bridge***.
- Interfejs:
  - nowy, całkowicie zdefiniowany – ***Facade***.
  - stary, ponowne użycie – ***Adapter*** .