# **Kompozyt**

Podczas rozwoju aplikacji często można spotkać komponenty, które mogą być pojedynczymi obiektami lub reprezentować hierarchię obiektów. Wzorzec Kompozytu pozwala klientowi operowanie w sposób ogólny na obiektach które mogą (lub nie) reprezentować hierarchię obiektów.

## ZALETY

* klient jednolicie wykonuje operacje na obiekcie złożonym i "prymitywnym"
* łatwo dodawać nowe rodzaje komponentów

## UCZESTNICY

* Komponent
  + deklaruje interfejs składanych obiektów;
  + implementuje, tam gdzie to możliwe, domyślne zachowanie w wypadku interfejsu wspólnego dla wszystkich klas;
  + definiuje interfejs umożliwiający dostęp i zarządzanie komponentami-dziećmi;
  + (czasami) definiuje interfejs umożliwiający dostęp do rodzica komponentu w strukturze rekurencyjnej i implementuje go, o ile jest to potrzebne.
* Liść
  + reprezentuje obiekty będące liśćmi w składanej strukturze; liść nie ma dzieci;
  + definiuje zachowanie obiektów pierwotnych w strukturze.
* Kompozyt
  + definiuje zachowanie komponentów mających dzieci;
  + przechowuje komponenty będące dziećmi;
  + implementuje operacje z interfejsu Komponentu związane z dziećmi.
* Klient
  + manipuluje obiektami występującymi w strukturze, wykorzystując do tego interfejs komponentu.

## WSPÓŁPRACA

Klienci używają interfejsu z klasy Komponent w celu komunikowania się z obiektami występującymi w składanej strukturze. Jeśli odbiorca jest Liściem, to żądania są realizowane bezpośrednio. Jeśli natomiast odbiorca jest Kompozytem, to zwykle przekazuje żądania swoim komponentom-dzieciom, być może wykonując przed i/lub po przekazaniu dodatkowe operacje.

# **Obserwator**

W [programowaniu obiektowym](https://pl.wikipedia.org/wiki/Programowanie_obiektowe) obiekty posiadają pewien stan, tj. zbiór aktualnych wartości pól obiektu, który w wyniku wykonywania na nich operacji może ulegać zmianie. Od bieżącego stanu mogą być zależne inne obiekty, dlatego musi istnieć możliwość ich powiadomienia o jego zmianie tak, aby mogły one się do niej dostosować. Możemy także żądać, aby inne obiekty były powiadamiane o tym, że inny obiekt próbuje wykonać konkretną czynność, np. ponownie nawiązywać utracone połączenie z bazą danych. Pragniemy zaimplementować ogólny mechanizm, który umożliwi nam osiągnięcie tych celów.

## BUDOWA

We wzorcu obserwator wyróżniamy dwa podstawowe typy obiektów:

* **obserwowany** - obiekt, o którym chcemy uzyskiwać informacje,
* **obserwator** - obiekt oczekujący na powiadomienie o zmianie stanu obiektu obserwowanego.

## ZALETY

* luźna zależność między obiektem obserwującym i obserwowanym. Ponieważ nie wiedzą one wiele o sobie nawzajem, mogą być niezależnie rozszerzane i rozbudowywane bez wpływu na drugą stronę,
* [relacja](https://pl.wikipedia.org/wiki/Relacja) między obiektem obserwowanym a obserwatorem tworzona jest podczas wykonywania programu i może być dynamicznie zmieniana,
* domyślnie powiadomienie otrzymują wszystkie obiekty. Obiekt obserwowany jest zwolniony z zarządzania subskrypcją — o tym czy obsłużyć powiadomienie, decyduje sam obserwator.

## WADY

* obserwatorzy nie znają innych obserwatorów, co w pewnych sytuacjach może wywołać trudne do znalezienia [skutki uboczne](https://pl.wikipedia.org/wiki/Skutek_uboczny_(informatyka)).

# **Strategia**

## CEL WZORCA STRATEGII

* Zdefiniowanie rodziny algorytmów, hermetyzacji każdego z nich, i uczynienie ich zamiennych. Strategia pozwala na zmianę algorytmu niezależnie od klientów, którzy go używają.
* Uchwycenie abstrakcji w interfejsie, zakopanie szczegółów implementacji w klasach pochodnych.

## STRUKTURA

W strategii definiujemy wspólny interfejs, dla obsługiwanych algorytmów, posiadający dozwolone metody. W kolejnym kroku implementujemy poszczególne strategie w poszczególnych klasach. Następnie budujemy klasę klienta, która będzie pozwalała na określenie strategii (na przykład poprzez jej wstrzyknięcie) oraz będzie posiadała referencję do aktualnie wybranej strategii. Klient współpracuje z wybraną strategią w celu wykonania określonego zadania.

## KIEDY NALEŻY STOSOWAĆ ?

* gdy istnieje potrzeba rozwiązania danego problemu na parę różnych sposobów
* gdy system musi być otwarty na rozszerzanie
* gdy chcesz zwiększyć czytelność swojego kodu
* gdy chcesz jasno i jawnie wyrazić intencje w kodzie

# **Metoda Wytwórcza**

Metoda wytwórcza jest **kreacyjnym** wzorcem projektowym rozwiązującym problem tworzenia obiektów-produktów bez określania ich konkretnych klas.

Metoda wytwórcza definiuje metodę która ma służyć tworzeniu obiektów bez bezpośredniego wywoływania konstruktora (poprzez operator ***new***). Podklasy mogą nadpisać tę metodę w celu zmiany klasy tworzonych obiektów.

Metoda wytwórcza, albo inaczej metoda fabrykująca jest **wzorcem konstrukcyjnym**. Służy do tworzenia nowych obiektów, nieokreślonych, lecz związanych z jednym, wspólnym interfejsem. Następnie dodajemy kilka klas, które będą implementować owy interfejs (to właśnie one będą tworzyć konkretne produkty). Klasy te reprezentują bardziej szczegółowe postacie produktu. Metoda fabrykująca nadaje klasom pełną odpowiedzialność, dotyczącą produkcji obiektów.

# **Dekorator**

**Opis wzorca projektowego dekorator**

Wzorzec projektowy dekorator należy do wzorców strukturalnych. Wzorzec tego typu umożliwia rozszerzenie funkcjonalności podstawowego obiektu poprzez opakowanie go. Obiekt opakowujący nazywa się dekoratorem i zawiera on wskaźnik do interfejsu głównego obiektu po którym również on dziedziczy.

Dekorator polega na opakowaniu bazowej klasy **w klasę dekorującą**. Wykorzystuje się do tego kompozycję. Wzorzec ten jest alternatywą dla dziedziczenia, które posiada szereg ograniczeń w tym zakresie. Przede wszystkim dekorator pozwala na dekorowanie w trakcie działania programu, a nie podczas kompilacji. Dodatkowo umożliwia „**składanie**” dekoratorów, a więc daje elastyczność w kwestii doboru zestawu nowych funkcjonalności.

Ważne w tym wzorcu jest to, że pierwotne zachowanie klasy, którą dekorujemy pozostaje bez zmian. Pozwala to zachować pełną kompatybilność z już istniejącym kodem i zapewnia transparentność naszego dekoratora.

Poniżej prosty przykład jak wygląda struktura omawianego wzorca projektowego:

