# 24. Wzorce projektowe i architektoniczne

Wzorzec – powszechnie stosowane rozwiązanie często występującego problemu zaadoptowane do specyficznego kontekstu.

### Rodzaje wzorców:

* Wzorce architektury (style architektoniczne)
* Wzorce projektowe
* Idiomy językowe

### Styl architektoniczny

* Często występująca w praktyce specjalizacja elementów i związków należących do widoku wraz ze zbiorem ograniczeń, jak ich używać
* Rodzina architektur, która spełnia zdefiniowane dla stylu ograniczenia
* Może być używany przez wiele systemów
* System może bazować na wielu stylach

### Klasyfikacja Clementsa

* **Widok modułowy**
  + Dekompozycja  
    Elementami są moduły (pakiety) lub klasy. Relacje między elementami ograniczone do „A jest częścią B”.
  + Generalizacja  
    Elementami są moduły (pakiety) lub klasy. Relacje między elementami ograniczone do „A jest szczególnym przypadkiem B”.
  + Warstwy (Layers) - przykład: PCMEF+, DDD  
    Elementami są warstwy oraz elementy należące do warstw. Relacje ograniczone do „A używa B”. Wprowadza kilka ograniczeń:
    - Każdy element oprogramowania należy do dokładnie jednej warstwy.
    - Element w warstwie może używać elementów w {dowolnej niższej warstwie | sąsiedniej niższej warstwie}.
    - Element w warstwie {może | nie może} używać innych elementów w tej samej warstwie.
  + Uwaga – struktura modułów często jest mocno powiązana z widokiem C&C (np. dekompozycja na pakiety M-V-C)
* **Widok C&C**
  + Filtry i potoki  
    Filtry (komponenty) stopniowo dokonują transformacji danych wejściowych w dane wyjściowe. Transformacje mogą polegać na dodaniu, uściśleniu, usunięciu lub zmianie reprezentacji danych. Działają niezależnie (być może współbieżnie). Nie znają swoich sąsiadów, są bezstanowe.  
    Potoki (połączenia) przenoszą dane z wyjścia filtra A do wejścia filtra B. Są jednokierunkowe, nie zmieniają danych ani ich porządku.
  + Klient-server (Tiers),  
    Połączenia typu „żadanie-odpowiedź”. Zwykle tylko klient wykonuje żądanie (połączenia asymetryczne). Komunikacja zwykle synchroniczna. Bywa dwupoziomowa (warstwa klienta, warstwa danych), trzypoziomowa (warstwa klienta, warstwa aplikacji, warstwa danych) lub wielopoziomowa (pomiędzy warstwą klienta a warstwą danych wstawione dodatkowe warstwy).
  + Czarna tablica (Blackboard)  
    System wieloagentowy, wymiana danych poprzez wspólną „tablice”, po której „piszą” wszyscy agenci. Koordynator dba, żeby nie powstał bałagan.
  + Mikrojądro  
    Podstawowe funkcjonalności, wszystko ponadto jest dostarczane przez zewnętrzne (względem jądra) serwery.
  + MVC  
    W modelu pasywnym kontroler uaktualnia widoki. W modelu aktywnym wykorzystanie np. wzorca projektowego Obserwator (opis dalej).
  + SOA
* **Widok alokacji**
  + Rozmieszczenia
    - Pokazuje rozmieszczenie elementów software’owych (wykonywalny kod) do węzłów przetwarzających.
    - Można reprezentować UML-owym diagramem rozmieszczenia (stereotypy: <<executable>>, <<script>>).
  + Implementacji
    - Pokazuje rozmieszczenie elementów software’owych (kody źródłowe) w środowisku wytwórczym (system plików).
    - Można reprezentować UML-owym diagramem rozmieszczenia (stereotypy: <<source>>, <<file>>).
  + Przypisania pracy
    - Pokazuje przypisanie elementów programowych do jednostek organizacyjnych (zespołów, osób).
    - Można reprezentować w UML z wykorzystaniem metek, np. {autor = „Jan Kowalski”}.

### Wzorce projektowe

Wzorce projektowe klasyfikujemy wg dwóch kryteriów. Pierwszy – rodzaj – określa to, co robi wzorzec:

* wzorce kreacyjne – związane z procesem tworzenia obiektów,
* wzorce strukturalne – dotyczą składania klas lub obiektów,
* wzorce czynnościowe – charakteryzują sposób, w jaki klasy lub obiekty oddziałują na siebie i rozdzielają zobowiązania.

Drugie kryterium – zakres – określa, czy wzorzec odnosi się przed wszystkim do klas, czy obiektów. Wzorce klasowe zajmują się związkami między klasami i ich podklasami. Związki te są ustanawiane za pomocą dziedziczenia (są to związki statyczne, tj. ustalane podczas kompilacji). Wzorce obiektowe zajmują się związkami między obiektami. Mogą one zmieniać się w czasie wykonania programu i są bardziej dynamiczne. Prawie wszystkie wzorce wykorzystują dziedziczenie. Jako „wzorce klasowe” określono więc tylko wzorce skupiające się na związkach między klasami. Poniższa tabela zawiera podział wzorców. Gwiazdką oznaczono opisane dalej dokładniej wzorce, natomiast poniżej opisano wszystkie pokrótce.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Rodzaj | | |
| Kreacyjne | Strukturalne | Czynnościowe |
| Zakres | Klasy | Metoda wytwórcza | Adapter (klasy) | Interpreter  Metoda szablonowa |
| Obiekty | Budowniczy  Fabryka abstrakcyjna\*  Prototyp  Singleton\* | Adapter (obiekty)  Dekorator\*  Fasada\*  Kompozyt\*  Most  Pełnomocnik  Pyłek | Iterator\*  Łańcuch zobowiązań  Mediator  Obserwator\*  Odwiedzający  Pamiątka  Polecenie  Stan  Strategia\* |

**Wzorce kreacyjne**

* Budowniczy (ang. Builder) – oddziela konstrukcję obiektów złożonych od ich reprezentacji. Umożliwia dzięki temu powstawanie w jednym procesie konstrukcyjnym różnych reprezentacji.
* Fabryka abstrakcyjna (ang. Abstract Factory) – zapewnia interfejs umożliwiający tworzenie rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych.
* Metoda wytwórcza (ang. Factory Method) – określa interfejs do tworzenia obiektów, lecz umożliwia podklasom decydowanie o tym, której klasy ma to być obiekt. Dzięki temu wzorcowi klasy mogą zdać się na podklasy w kwestii tworzenia egzemplarzy.
* Prototyp (ang. Prototype) – specyfikuje rodzaje tworzonych obiektów, używając prototypowego egzemplarza, a także tworzy nowe obiekty, kopiując ten prototyp.
* Singleton – gwarantuje, że klasa ma tylko jeden egzemplarz i zapewnia globalny dostęp do niego.

**Wzorce strukturalne**

* Adapter (albo: Wrapper) – przekształca interfejs klasy w taki, jakiego klienci oczekują. Dzięki temu wzorcowi klasy mogą współpracować, co bez niego nie byłoby możliwe, ponieważ mają niezgodne interfejsy.
* Dekorator (ang. Decorator) – dynamicznie dołącza do obiektów dodatkowe zobowiązania. Ten wzorzec zapewnia elastyczną alternatywę dla tworzenia podklas w celu rozszerzania funkcjonalności.
* Fasada (ang. Facade) – zapewnia jednolity interfejs dla podsystemu zawierającego wiele interfejsów. Definiuje interfejs wyższego poziomu, co ułatwia korzystanie z podsystemu.
* Kompozyt (ang. Composite) – składa obiekty w struktury drzewiaste reprezentujące hierarchie typu część-całość. Umożliwia klientom jednakowe traktowanie pojedynczych obiektów i złożeń obiektów.
* Most (ang. Bridge) – oddziela abstrakcję od implementacji, tak by mogły się zmieniać niezależnie jedna od drugiej.
* Pełnomocnik (ang. Proxy) – zapewnia substytut lub reprezentanta innego obiektu w celu sterowania dostępem do niego.
* Pyłek (ang. Flyweight) – wykorzystuje współdzielenie obiektów w celu efektywnej obsługi wielkiej liczby drobnych obiektów.

**Wzorce czynnościowe**

* Interpreter – definiuje reprezentację dla gramatyki zadanego języka, a także interpreter, który wykorzystuje tę reprezentację do interpretowania zdań w zadanym języku.
* Iterator – zapewnia sekwencyjny dostęp do elementów obiektu zagregowanego bez ujawniania jego reprezentacji wewnętrznej.
* Łańcuch zobowiązań (ang. Chain of Responsibility) – umożliwia uniknięcie związania wysyłającego żądanie z odbiorcą żądania przez danie więcej niż jednemu obiektowi szansy obsłużenia tego żądania. Tworzy łańcuch odbierających obiektów i przekazuje wzdłuż niego żądanie, aż jakiś obiekt je obsłuży.
* Mediator – definiuje obiekt enkapsulujący informacje o tym, jak obiekty współdziałają. Przyczynia się do rozluźnienia powiązań między obiektami, gdyż sprawia, że obiekty nie odwołują się do siebie wprost. Umożliwia też oddzielne zmienianie ich sposobu porozumiewania się.
* Metoda szablonowa (ang. Template Method) – definiuje szkielet algorytmu jako operację, odkładając definicję niektórych kroków algorytmu do podklas. Umożliwia podklasom przedefiniowanie pewnych kroków algorytmu bez zmiany jego struktury.
* Obserwator (ang. 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.
* Odwiedzający (ang. Visitor) – określa operację, która ma być wykonana na elementach struktury obiektowej. Umożliwia definiowanie nowej operacji bez modyfikowania klas elementów, na których ona działa.
* Pamiątka (ang. Memento) – bez naruszania enkapsulacji, zapamiętuje i udostępnia na zewnątrz stan wewnętrzny obiektu tak, że obiekt może być później przywrócony do zapamiętanego stanu.
* Polecenie (ang. Command) – enkapsuluje żądania w postaci obiektu, co umożliwia parametryzowanie klientów różnymi żądaniami, kolejkowanie żądań lub zapisywanie ich w dziennikach, a także ułatwia implementację anulowanych operacji.
* Stan (ang. State) – umożliwia obiektowi zmianę zachowania, gdy zmienia się jego stan wewnętrzny. Dzięki temu obiektu zdaje się zmieniać wówczas swoją klasę.
* Strategia (ang. Strategy) – definiuje rodzinę algorytmów, enkapsuluje każdy z nich i umożliwia ich wymianę. Sprawia, że możliwe staje się zmienianie algorytmu niezależnie od używających go klientów.

**Fabryka abstrakcyjna**

Używany, gdy:

* system powinien być niezależny od tego, jak jego produkty są tworzone, składane i reprezentowane,
* system powinien być konfigurowany przy użyciu jednej z wielu rodzin produktów,
* rodzina związanych ze sobą obiektów-produktów jest zaprojektowana tak, by te obiekty były używane razem, a nam zależy na zachowaniu tego ograniczenia,
* chcemy dostarczyć bibliotekę klas produktów, ujawniając jedynie ich interfejsy, a nie implementacje.



* FabrykaAbstrakcyjna – deklaruje interfejs tworzenia produktów abstrakcyjnych,
* FabrykaKonkretna – implementuje operacje tworzenia produktów konkretnych,
* ProduktAbstrakcyjny – deklaruje interfejs dla pewnego rodzaju produktu,
* ProduktKonkretny – definiuje obiekt będący produktem, który ma być utworzony przez odpowiednią fabrykę konkretną; implementuje interfejs klasy ProduktAbstrakcyjny,
* Klient – używa jedynie interfejsów deklarowanych przez klasy FabrykaAbstrakcyjna i ProduktAbstrakcyjny.

Podczas działania programu zwykle tworzony jest jeden egzemplarz klasy FabrykaKonkretna. Tworzy on obiekty-produkty o pewnej szczególnej implementacji. Aby tworzyć inne produkty, klienci powinni skorzystać z innej fabryki konkretnej.

**Singleton**

Używany, gdy:

* klasa musi mieć właśnie jeden egzemplarz, dostępny dla klientów,
* powinno być możliwe rozbudowywanie tego jedynego egzemplarza przez definiowanie podklas, a klienci powinni móc używać rozszerzonego egzemplarza bez modyfikowania swojego kodu.



* Singleton – definiuje operację Egzemplarz, umożliwiającą klientom uzyskiwanie dostępu do tego unikatowego egzemplarza; operacja ta jego operacją klasową (statyczną); singleton może być odpowiedzialny za tworzenie swojego jedynego egzemplarza.

Klienci dostają się do egzemplarza single tonu wyłącznie poprzez operację Egzemplarz zdefiniowaną w tej klasie.

**Dekorator**

Używany:

* żeby dynamicznie i w przezroczysty sposób dodać zobowiązania do pojedynczych obiektów,
* w wypadku zobowiązań, które mogą być cofnięte,
* gdy rozszerzanie przez definiowanie podklas jest niepraktyczne (np. możliwych jest mnóstwo niezależnych rozszerzeń, które przy próbie uwzględnienia każdej kombinacji prowadzą do gwałtownego wzrostu liczby klas; definicja klasy może też być ukryta lub niedostępna do tworzenia podklas)



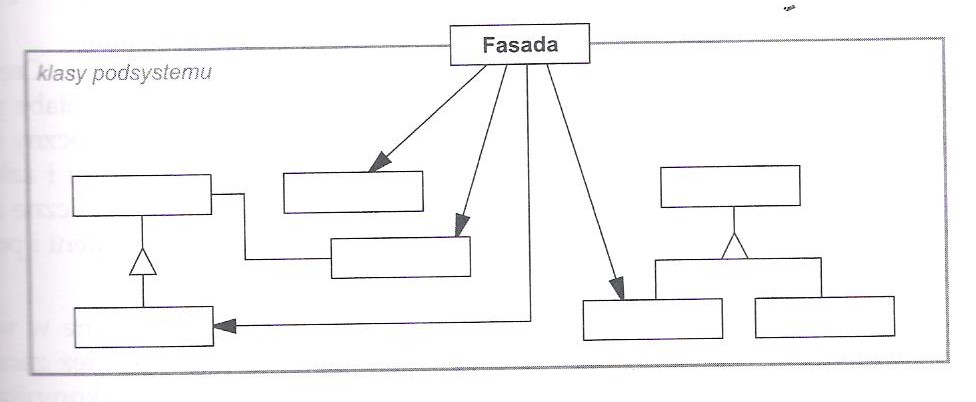
* Komponent – definiuje interfejs obiektów, do których można dynamicznie dołączyć zobowiązania,
* KomponentKonkretny – definiuje obiekt, do którego można dołączać dodatkowe zobowiązania,
* Dekorator – zarządza odwołaniem do obiektu Komponent i definiuje interfejs dopasowany do interfejsu Komponentu,
* DekoratorKonkretny – dodaje zobowiązania do komponentu.

Dekorator przesyła żądania do swojego obiektu Komponent. Może wykonywać dodatkowe operacje przed lub po przesłaniu żądania.

**Fasada**

Używany, gdy:

* chcemy zapewnić prosty interfejs do złożonego systemu,
* istnieje wiele zależności między klientami abstrakcji a klasami ją implementującymi,
* chcemy ułożyć podsystemy warstwami – fasada stanowi wtedy punkt wejścia do każdego poziomu podsystemów.



* Fasada – wie, jakie klasy podsystemu są odpowiedzialne za spełnienie żądania; przekazuje żądania klienta do odpowiednich obiektów podsystemu,
* Klasy podsystemu – implementują funkcje podsystemu; wykonują pracę przydzieloną przez obiekt klasy Fasada; nie wiedzą nic o fasadzie.

Klienci komunikują się z podsystemem poprzez wysyłanie żądań do Fasady, która przekazuje je do odpowiednich obiektów podsystemu. Fasa może tłumaczyć własny interfejs na interfejsy podsystemu. Klienci wykorzystujący Fasadę nie muszą mieć bezpośredniego dostępu do obiektów z jej podsystemu.

**Kompozyt**

Używany, gdy:

* chcemy przedstawić hierarchie obiektów całość-część,
* chcemy, aby klienci mogli ignorować różnicę między złożeniami obiektów a pojedynczymi obiektami; klienci będą wtedy jednakowo traktować wszystkie obiekty występujące w strukturze.



* Komponent – deklaruje interfejs składanych obiektów; implementuje 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 i implementuje go,
* 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.

Klienci używają interfejsu z klasy Komponent w celu komunikowania się z obiektami występującymi w składanej strukturze. Jeśli odbiorca jest Liściem, żądania te są realizowane bezpośrednio. Jeśli odbiorca jest Kompozytem, to zwykle przekazuje żądania swoim komponentom-dzieciom, ewentualnie dokonując przed i/lub po ich wykonaniu dodatkowe operacje.

**Iterator**

Używany w celu:

* uzyskania dostępu do zawartości obiektu-agregatu bez ujawniania jego struktury wewnętrznej,
* umożliwienia wielokrotnego przechodzenia obiektów-agregatów,
* zapewnienia jednakowego interfejsu przechodzenia różnych struktur zagregowanych.



* Iterator – definiuje interfejs dostępu do elementów i przechodzenia ich,
* IteratorKonkretny – implementuje interfejs Iteratora; pamięta bieżącą pozycję osiągniętą przy przechodzeniu agregatu,
* Agregat – definiuje interfejs tworzenia obiektów-iteratorów,
* AgregatKonkretny – implementuje interfejs tworzenia Iteratora tak, by przekazywał egzemplarz odpowiedniego IteratoraKonkretnego.

Klasa IteratorKonkretny śledzi, który obiekt w agregacie jest bieżący i potrafi wskazać następny obiekt.

**Obserwator**

Stosowany, gdy:

* jakaś abstrakcja ma dwa aspekty, jeden zależny od drugiego,
* zmiana jednego obiektu wpływa na zmiany innych i nie wiadomo, ile obiektów trzeba zmienić,
* obiekt powinien być w stanie powiadamiać inne obiekty, nie przyjmując żadnych założeń co do tego, co te obiekty reprezentują.



* Obserwowany – zna swoich obserwatorów; zapewnia interfejs dołączania i odłączania Obserwatorów,
* Obserwator – definiuje interfejs uaktualniania dla obiektów, które powinny być powiadamiane o zmianach, jakie zaszły w Obserwowanym,
* ObserwowanyKonkretny – przechowuje stan, który interesuje obiekty ObserwatorKonkretny; wysyła powiadomienia do swoich obserwatorów, gdy zmienia się jego stan,
* ObserwatorKonkretny – utrzymuje odwołanie do obiketu ObserwowanyKonkretny; przechowuje stan, który powinien być spójny ze stanem obserwowanego; implementuje interfejs obserwatora służący do uaktualniania w celu zachowania spójności swojego stanu ze stanem obserwowanego.

ObserwowanyKonkretny zawsze powiadamia swoich obserwatorów, gdy wystąpi zmiana mogąca prowadzić do niespójności stanu. Po otrzymaniu informacji o zmianie, ObserwatorKonkretny może zapytać obserwowanego o informacje dotyczące tej zmiany. Używa ich do uaktualnienia swojego stanu na podstawie stanu obserwowanego.

**Strategia**

Używany, gdy:

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



* Strategia – deklaruje interfejs wspólny dla wszystkich obsługiwanych algorytmów; Kontekst wykorzystuje ten interfejs do wykonania algorytmu,
* StrategiaKonkretna – implementuje algorytm, wykorzystując interfejs z klasy Strategia,
* Kontekst – konfigurowany za pomocą obiektu StrategiaKonkretna; utrzymuje odwołanie do obiektu Strategia,; może definiować interfejs umożliwiający Strategii uzyskanie dostępu do jego danych.