

Contents

1	UML	4
2	BMPN	5
2.1	zdarzenia	5
3	wzorce kreacyjne	5
3.1	singleton	5
3.1.1	implementacja	6
3.2	metoda wytworcza (factory method)	7
3.3	fabryka abstrakcyjna (abstract factory)	8
3.4	budowniczy (builder)	8
3.4.1	problem	9
3.5	prototyp	9
4	wzorce behawioralne	9
4.1	Obserwator (observer)	9
4.1.1	kontekst	10
4.1.2	problem	10
4.1.3	implementacja	10
4.2	Stan (state)	10
4.2.1	kontekst	11
4.2.2	problem	11
4.2.3	implementacja	12
4.3	strategia (strategy)	12
4.4	iterator	13
4.5	mediator	13
4.6	Metoda szablonowa (template method)	13
4.7	Odwiedzajacy (visitor)	14
4.8	polecenie	14
5	wzorce strukturalne	15
5.1	kompozyt (composite)	15
5.1.1	kontekst	15
5.1.2	problem	16
5.2	dekorator (decorator)	16
5.3	pełnomocnik (proxy)	16
5.4	fasada (facade)	18
5.5	most (bridge)	19

5.6	adapter	20
5.7	pylek (cache, flyweight)	21
6	pytania zamknięte	21
6.1	zaznacz głównie rodzaje procesów biznesowych	21
6.2	stosując wzorzec <BLANK> gdy nie wiesz z góry jakie typy obiektów pojawiają się jakie twoim programie między nimi zależności	21
6.3	stosując wzorzec <BLANK> gdy istnieje potrzeba wykonywanie jakiego działania na elementach złożonej struktury obiektów (jak drzewo obiektów)	22
6.4	stosuj wzorzec <BLANK> gdy musisz zaimplementować drzewiastą strukturę obiektów	22
6.5	korzystając z wzorca <BLANK> gdy chcesz oszczędniej wykorzystać zasoby systemowe poprzez ponowne wykorzystanie już istniejących obiektów zamiast odbudowywać je raz za razem	22
6.6	stosuj wzorzec <BLANK> gdy chcesz przyjmować dodatków dodatkowe obowiązki obiektom w trakcie działania programu, bez pisania	22
6.7	stosowanie wzorca <BLANK> pozwala uprzątnąć logikę biznesową czynności pomocniczych	22
6.8	<BLANK> pozwala odizolować logikę biznesową klasy od szczegółów implementacyjnych algorytmów, które nie są istotne w kontekście tej logiki	22
6.9	stosuj wzorzec <BLANK> gdy chcesz aby kod klienci traktował zarówno proste, jak i złożone elementy jednakowo	22
6.10	stosuj wzorzec <BLANK> gdy istnieje potrzeba wykonania jakiegoś działania na wszystkich elementach złożonej struktury obiektów (jak drzewo obiektów)	23
6.11	korzystaj z wzorca <BLANK> gdy zamierzasz pozwolić użytkownikowi twoją bibliotekę lub framework rozbudowywać jej wewnętrzne komponenty	23
6.12	które stwierdzenia są prawdziwe, gdy aktor A uogólnia aktora B	23
6.13	które z poniższych stwierdzeń charakteryzują przypadki użycia	23
6.14	wybierz zdania prawdziwe określające pojęcie błędu logicznego w oprogramowaniu	23
6.15	Proces określania wymagań dla systemu informatycznego można podzielić na następujące fazy	23
6.16	Kontekst systemu	24

6.17	Zaznacz główne rodzaje procesów biznesowych	24
6.18	Strukturalne wzorce projektowe to	24
6.19	Wybierz zdania prawdziwie określające pojęcie złożoności cyk- lometrycznej	24
6.20	Które z poniższych stwierdzeń charakteryzuje przypadki użycia	25
7	pytania otwarte odpowiedz	25
7.1	kiedy nie należy stosować dziedziczenia opisz przynajmniej dwa przypadki	25
7.2	opisać silną agregację	25
8	pytania otwarte modelio	25
8.1	system w którym pracownicy mogą być również klientami, zaproponuj trzy rozwiązania opisując ich wady i zalety	25
8.2	zamodeluj podsystem obsługi klienta w sklepie internetowym Zacznij od opisu wymagań i procesów	25

1 UML



2 BPMN

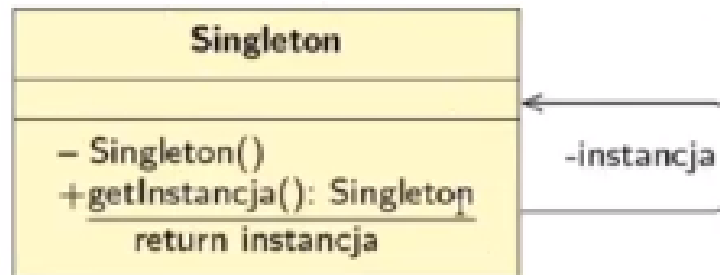
2.1 zdarzenia

Zdarzenia	Początkowe				Pośrednie		Końcowe	
	Najwyższego poziomu	Podproces Zdarzenie Przerywające	Podproces Zdarzenie Nieprzerwywające	Przechwytyujące	Krawędziowe Przerywające	Krawędziowe Nieprzerwywające	Rzucające	Końcowe
Bez typu: Punkt początku / końca procesu, pokazanie zmiany stanu w procesie.								
Komunikat: Otrzymanie i Wystanie komunikatów.								
Stoper: Punkt czasu, okresowa możliwość kontynuacji, opóźnienie.								
Eskałacja: Eskalacja do wyższego poziomu odpowiedzialności.								
Warunek: Reaguje na zmianę warunków biznesowych lub integruje zasady biznesowe.								
Łączy: Łączy odległe punkty na diagramie, opowiada przepływowi procesu między nimi.								
Błąd: Przechwytyje lub Ustawia (rzuca) nazwany Błąd.								
Anulowanie: Powoduje anulowanie transakcji lub wyzwala anulowanie.								
Kompensacja: Obsługuje lub wyzwala kompensację.								
Sygnal: Sygnalizacja pomiędzy różnymi Procesami. Rzucony Sygnal. Może być przechwycony wielokrotnie.								
Wielokrotne: Przechwytyujące przechwytyje jedno z wielu Zdarzeń. Rzucające rzuca wszystkie zdefiniowane Zdarzenia								
Wielokrotne Równoległe: Przechwytyje wszystkie z zestawu Zdarzeń Równoległych.								
Zerwanie: Wyzwala natychmiastowe i bezwarunkowe zakończenie Procesu.								

3 wzorce kreacyjne

3.1 singleton

<https://refactoring.guru/design-patterns/singleton>



Rysunek 9: Schemat wzorca *Singleton*

- zagwarantować że jest jeden obiekt tego typu (np. konfiguracja/stan globalny)
- gdy w twoim programie ma prawo istnieć wyłącznie jeden ogólnodostępny obiekt danej klasy. Przykładem może być połączenie z bazą danych, którego używa wiele fragmentów programu.
- gdy potrzebujesz ściślejszej kontroli nad zmiennymi globalnymi.

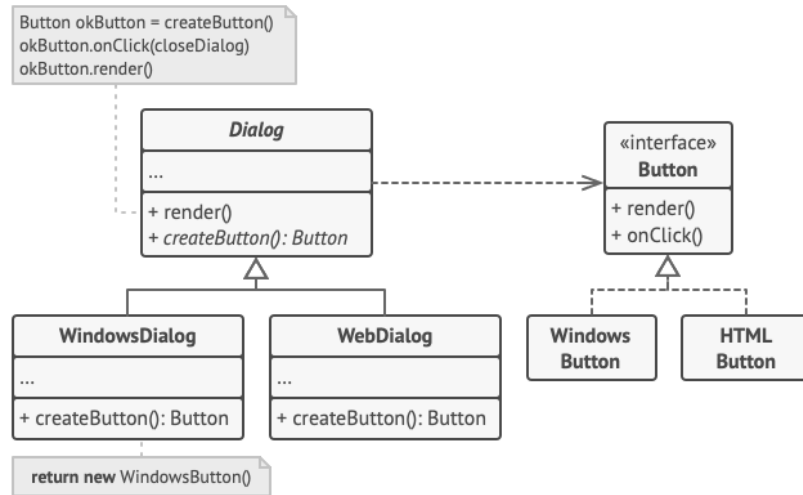
3.1.1 implementacja

```
class singleton {

private static singleton; //nasz obiekt
public static singleton getSingleton() //statyczna publiczna funkcja do otrzymywania t
{
if(instancja==null)
instancja = new Singleton();

return singleton;
}
};
```

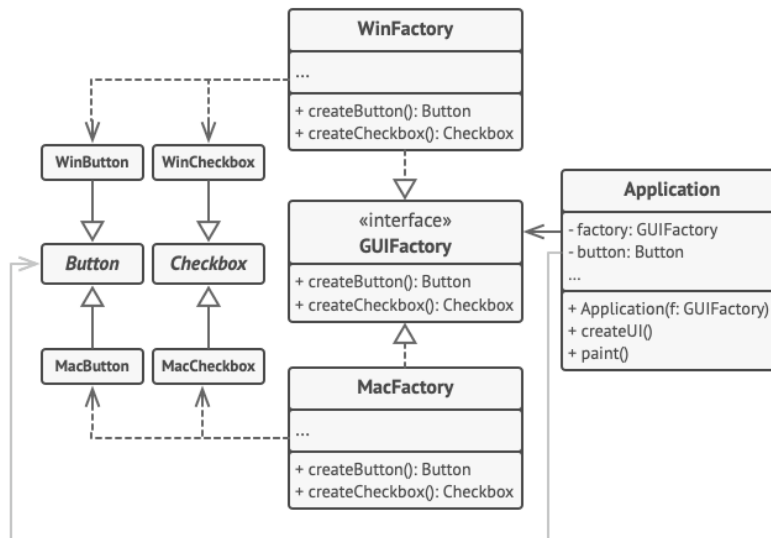
3.2 metoda wytworcza (factory method)



- udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.
- Stosuj Metodę Wytwórczą gdy nie wiesz z góry jakie typy obiektów pojawią się w twoim programie i jakie będą między nimi zależności.
- Metody Wytwórczej gdy zamierzasz pozwolić użytkującym twą bibliotekę lub framework rozbudowywać jej wewnętrzne komponenty.
- gdy chcesz oszczędniej wykorzystać zasoby systemowe poprzez ponowne wykorzystanie już istniejących obiektów, zamiast odbudowywać je raz za razem.

Metoda Wytwórcza oddziela kod konstruujący produkty od kodu który faktycznie z tych produktów korzysta. Dlatego też łatwiej jest rozszerzać kod konstruujący produkty bez konieczności ingerencji w resztę kodu.

3.3 fabryka abstrakcyjna (abstract factory)



Przykład międzyplatformowych klas UI.

- który pozwala tworzyć rodziny spokrewnionych ze sobą obiektów bez określania ich konkretnych klas.
- gdy twój kod ma działać na produktach z różnych rodzin, ale jednocześnie nie chcesz, aby ściśle zależał od konkretnych klas produktów. Mogą one bowiem być nieznane na wcześniejszym etapie tworzenia programu, albo chcesz umożliwić przyszłą rozszerzalność aplikacji.
- dostarcza ci interfejs służący tworzeniu obiektów z różnych klas danej rodziny produktów. O ile twój kod będzie kreował obiekty za pośrednictwem tego interfejsu — nie musisz się martwić stworzeniem produktu w niezgodnym z innymi wariantami.

3.4 budowniczy (builder)

- **SKŁADANIE OBIEKTU Z MAŁYCH CZĘŚCI** np fabryka pizzy, konstruujesz ciasto, dodatki i sos
- gdy potrzebujesz możliwości tworzenia różnych reprezentacji jakiegoś produktu (na przykład, domy z kamienia i domy z drewna).
- Stosuj ten wzorzec do konstruowania drzew Kompozytowych lub innych złożonych obiektów.

- Stosuj wzorec Budowniczy, aby pozbyć się “teleskopowych konstruktorów”.

```
Pizza(int size) { }
Pizza(int size, boolean cheese) { }
Pizza(int size, boolean cheese, boolean pepperoni) { }
```

3.4.1 problem

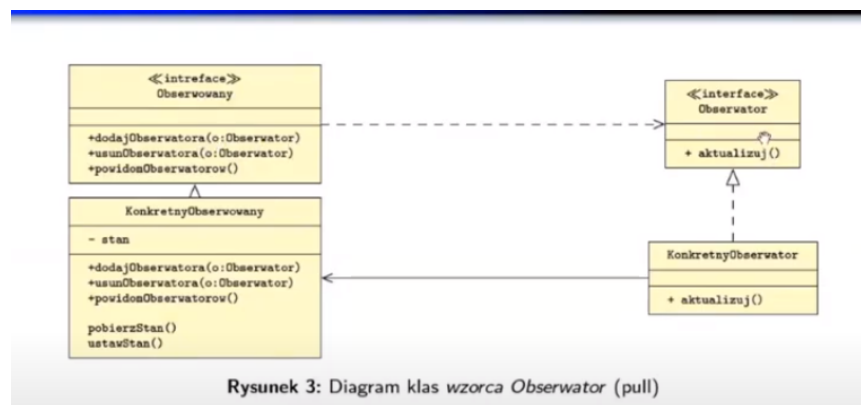
Wyobraź sobie jakiś skomplikowany obiekt, którego inicjalizacja jest pracochłonnym, wieloetapowym procesem obejmującym wiele pól i obiektów zagnieźdzonych. Taki kod inicjalizacyjny jest często wrzucany do wielgachnego konstruktora, przyjmującego mnóstwo parametrów. Albo jeszcze gorzej: kod taki rozrzucono po całym kodzie klienckim.

3.5 prototyp

- który umożliwia kopiowanie już istniejących obiektów bez tworzenia zależności pomiędzy twoim kodem, a klasami obiektów.
- deleguje proces klonowania samym obiektom, które mają być sklonowane. We wzorcu tym deklarowany jest wspólny interfejs dla wszystkich obiektów wspierających funkcjonalność klonowania.

4 wzorce behawioralne

4.1 Obserwator (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 (np. w kalkulatorze mamy 3 klasy wypisywania które mają w sobie string do wypisywania, kiedy wprowadzamy nowe działanie wszystkie są aktualizowane)
- wydaje mi się że realizowany w grach -> bo trzeba aktualizować stan obiektów a one muszą znać stan innych
- gdy zmiany stanu jednego obiektu mogą wymagać zmiany w innych obiektach, a konkretny zestaw obiektów nie jest z góry znany lub ulega zmianom dynamicznie
- gdy jakieś obiekty w twojej aplikacji muszą obserwować inne, ale tylko przez jakiś czas lub w niektórych przypadkach.

4.1.1 kontekst

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

4.1.2 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

4.1.3 implementacja

<https://refactoring.guru/design-patterns/observer> zagwarantowanie że przed rozesłaniem powiadomienia stan obserwowanego jest wewnętrznie spójny

model push (obserwowany wysyła wszystkie informacje same) model pull (obserwowany wysyła POWIADOMIENIE a każdy inny pyta się o to czego potrzebuje z jakichś zmian)

4.2 Stan (state)

<https://refactoring.guru/design-patterns/state>

- umożliwia obiektowi zmianę zachowania, gdy zmienia się jego stan wewnętrzny (np. ktoś zmienia typ konta bankowego)

- gdy masz do czynienia z obiektem którego zachowanie jest zależne od jego stanu, liczba możliwych stanów jest wielka, a kod specyficzny dla danego stanu często ulega zmianom.
- gdy masz klasę zaśmieconą rozbudowanymi instrukcjami warunkowymi zmieniającymi zachowanie klasy zależnie od wartości jej pól.
- pomaga poradzić sobie z dużą ilością kodu który się powtarza w wielu stanach i przejściach między stanami automatu skończonego, bazującego na instrukcjach warunkowych.

4.2.1 kontekst

- zachowanie obiektu zależy od jego stanu, a obiekt ten 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 do specjalnej klasy z inną implementacją np. pobierz podatek

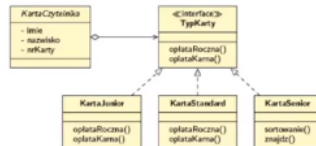
4.2.2 problem

chcemy umożliwić obiektowi zmianę zachowania w momencie zmiany wewnętrzznego stanu obiektu hermetyzując stan w postaci klasy

4.2.3 implementacja

ROZWIĄZANIA

Drugie rozwiązanie polega na rozdzieleniu odpowiedzialności *Karty Czytelnika* na część przechowującą dane i część reprezentującą stan.



Rysunek 6: Drugi przykład rozwiązania problemu typu kart czytelnika

- Część przechowująca dane, nadal nazywana *Kartą Czytelnika*, posiada referencję do obiektu reprezentującego aktualny typ, dziedziczącego po klasie abstrakcyjnej lub implementującej interfejs. Dzięki temu zmiana typu wymaga jedynie utworzenia instancji innej klasy *Typ Karty* i przypisanie jej do *Karty Czytelnika*.
- Efektem takiego projektu jest czytelniejszy podział odpowiedzialności, który jednocześnie posiada zalety brakujące w poprzednim rozwiązaniu.

4.3 strategia (strategy)

<https://refactoring.guru/design-patterns/strategy>

- różnica w implementacji ze stanem
- w stanie klient nie widzi z kim działa
- w strategii klient zna wewnętrzną strukturę - wie kto używa
- pomagają poradzić sobie z dużą ilością kodu, który się powtarza w wielu stanach i przejściach między stanami automatu skończonego, bazującego na instrukcjach warunkowych.
- gdy masz w programie wiele podobnych klas, różniących się jedynie sposobem wykonywania jakichś zadań.
- odizolować logikę biznesową klasy od szczegółów implementacyjnych algorytmów, które nie są istotne w kontekście tej logiki.
- gdy twoja klasa zawiera duży operator warunkowy, którego zadaniem jest wybór odpowiedniego wariantu tego samego algorytmu.

4.4 iterator

- hermetyzacja iteracji
- gdy kolekcja z którą masz do czynienia posiada skomplikowaną strukturę, ale zależy ci na ukryciu jej przed klientem (dla wygody, lub dla bezpieczeństwa).
- w celu redukcji duplikowania kodu przeglądania elementów zbiorów na przestrzeni całego programu.
- gdy chcesz, aby twój kod był w stanie przeglądać elementy różnych struktur danych, lub gdy nie znasz z góry szczegółów ich struktury.
- abstrakcja dla skomplikowanych struktur danych np. drzewo lista

```
Iterator iterator = menuCostam.utworzIterator();
while (iterator.hasNext())
{
    pozycjaMenu pozycja = iterator.next();
}
```

4.5 mediator

pozwalający zredukować chaos zależności pomiędzy obiektami. Wzorzec ten ogranicza bezpośrednią komunikację pomiędzy obiektami i zmusza je do współpracy wyłącznie za pośrednictwem obiektu mediatora

- pozwalający zredukować chaos zależności pomiędzy obiektami. Wzorzec ten ogranicza bezpośrednią komunikację pomiędzy obiektami i zmusza je do współpracy wyłącznie za pośrednictwem obiektu mediatora
- gdy nie możesz ponownie użyć jakiegoś komponentu w innym programie, z powodu zbytnej jego zależności od innych komponentów

gdy zauważysz, że tworzysz mnóstwo podklas komponentu tylko aby móc ponownie użyć jakiegoś zachowanie w innych kontekstach.

4.6 Metoda szablonowa (template method)

`./template` definiujący szkielet algorytmu w klasie bazowej, ale pozwalający podklasom nadpisać pewne etapy tego algorytmu bez konieczności zmiany jego struktury.

- gdy chcesz pozwolić klientom na rozszerzanie niektórych tylko etapów algorytmu, ale nie całego, ani też jego struktury.
- gdy masz wiele klas zawierających niemal identyczne algorytmy różniące się jedynie szczegółami. W takiej sytuacji bowiem konieczność modyfikacji algorytmu skutkuje koniecznością modyfikacji wszystkich klas.

4.7 Odwiedzający (visitor)

- gdy istnieje potrzeba wykonywania jakiegoś działania na wszystkich elementach złożonej struktury obiektów (jak drzewo obiektów).
- pozwala uprzątnąć logikę biznesową czynności pomocniczych.
- Warto stosować ten wzorzec gdy jakieś zachowanie ma sens tylko w kontekście niektórych klas wchodzących w skład hierarchii klas, ale nie wszystkich.

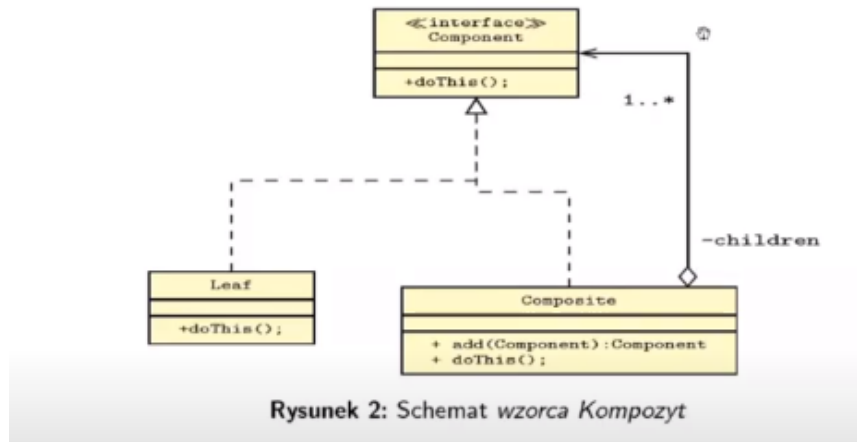
4.8 polecenie

który zmienia żądanie w samodzielny obiekt zawierający wszystkie informacje o tym żądaniu. Taka transformacja pozwala na parametryzowanie metod przy użyciu różnych żądań. Oprócz tego umożliwia opóźnianie lub kolejkovanie wykonywania żądań oraz pozwala na cofanie operacji.

- gdy chcesz parametryzować obiekty za pomocą działań.
- pozwala układać kolejki zadań, ustalać harmonogram ich wykonania bądź uruchamiać je zdalnie.

5 wzorce strukturalne

5.1 kompozyt (composite)



TLDR: Drzewko w którym liść zawiera siebie + listę dzieci

- zadaniem jest łączenie obiektów w strukturę tak, że reprezentują hierarchie części-całości, unifikując dostęp do kolekcji jak i pojedynczego obiektu.
- umożliwia to klientom jednolite traktowanie pojedynczych obiektów i również ich kompozycji
- Stosuj wzorzec Kompozyt gdy musisz zaimplementować drzewiastą strukturę obiektów.
- Stosuj ten wzorzec gdy chcesz, aby kod kliencki traktował zarówno proste, jak i złożone elementy jednakowo.

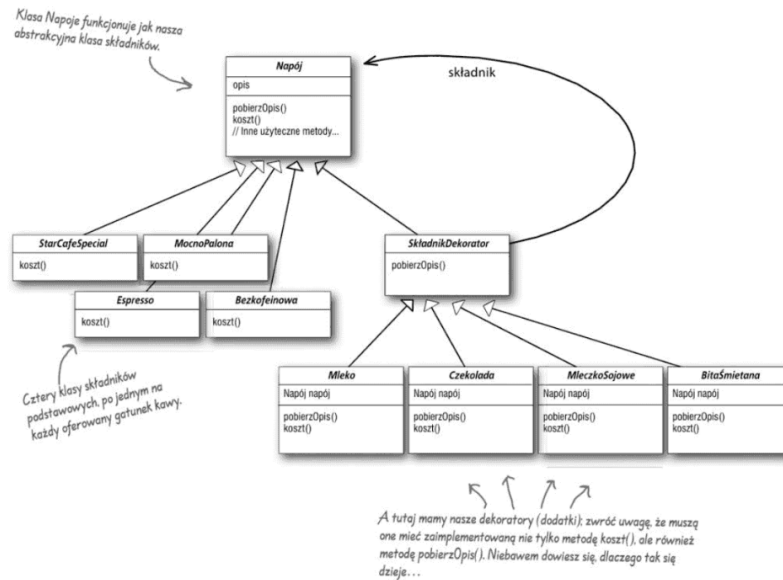
5.1.1 kontekst

chcemy przedstawić hierarchie obiektów część-całość Hierarchia obiektów ma wspólną klasę bazową (abstrakcyjną)

5.1.2 problem

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

5.2 dekorator (decorator)



pozwalający dodawać nowe obowiązki obiektom poprzez umieszczanie tych obiektów w specjalnych obiektach opakujących, które zawierają odpowiednie zachowania.

- dodawanie dodatkowej funkcjonalności do obiektów
- gdy chcesz przypisywać dodatkowe obowiązki obiektom w trakcie działania programu, bez psucia kodu, który z tych obiektów korzysta.
- gdy rozszerzenie zakresu obowiązków obiektu za pomocą dziedziczenia byłoby niepraktyczne, lub niemożliwe.

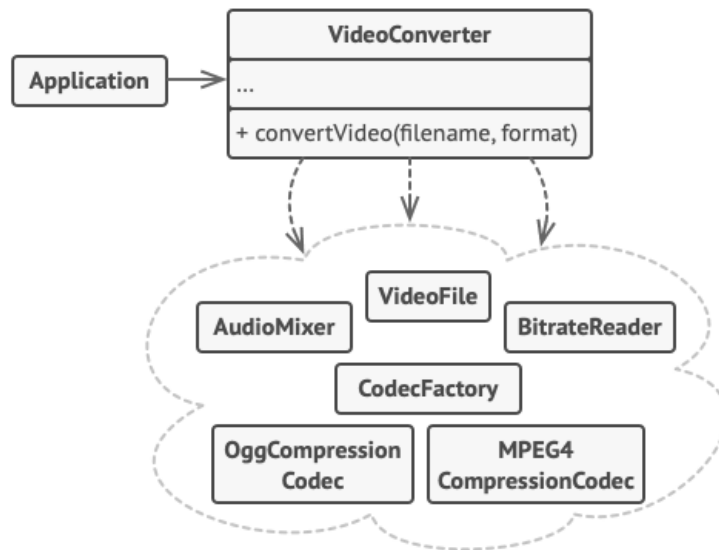
5.3 pełnomocnik (proxy)

pozwalający stworzyć obiekt zastępczy w miejsce innego obiektu. Pełnomocnik nadzoruje dostęp do pierwotnego obiektu, pozwalając na wykonanie

jakiejś czynności przed lub po przekazaniu do niego żądania

- Leniwa inicjalizacja (wirtualny pełnomocnik). Gdy masz do czynienia z zasobożernym obiektem usługi, którego potrzebujesz jedynie co jakiś czas.
- Kontrola dostępu (pełnomocnik ochronny). Przydatne, gdy chcesz pozwolić tylko niektórym klientom na korzystanie z obiektu usługi. Na przykład, gdy usługi stanowią kluczową część systemu operacyjnego, a klienci to różne uruchamiane aplikacje (również te szkodliwe).
- Lokalne uruchamianie zdalnej usługi (pełnomocnik zdalny). Użyteczne, gdy obiekt udostępniający usługę znajduje się na zdalnym serwerze.
- Prowadzenie dziennika żądań (pełnomocnik prowadzący dziennik). Pozwala prowadzić rejestr żądań przesyłanych do obiektu usługi.
- Przechowywanie w pamięci podręcznej wyników działań (pełnomocnik z pamięcią podręczną). Pozwala przechować wyniki przekazywanych żądań i zarządzać cyklem życia pamięci podręcznej. Szczególnie ważne przy dużych wielkościach danych wynikowych.
- Sprytne referencje. Można likwidować zasobożerny obiekt, gdy nie ma klientów którzy go potrzebują.

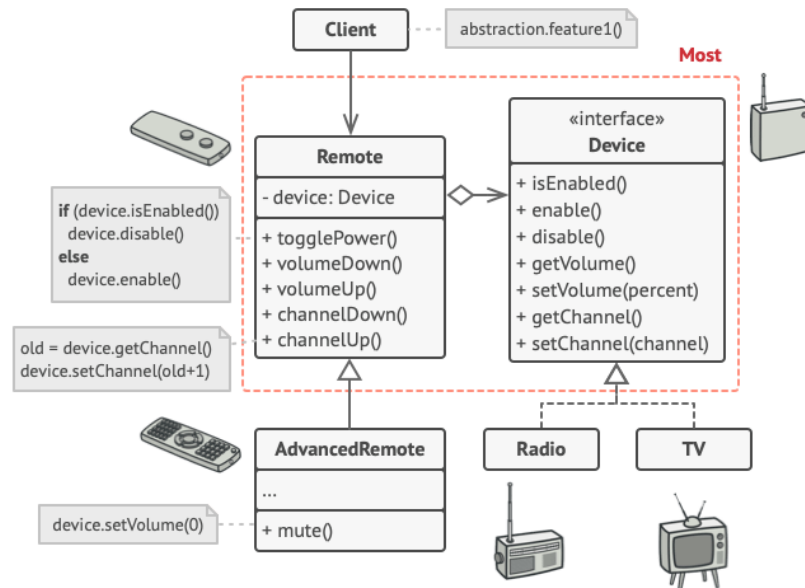
5.4 fasada (facade)



który wyposaża bibliotekę, framework lub inny złożony zestaw klas w uproszczony interfejs.

- taki wrapper na wiele rzeczy
- gdy potrzebujesz ograniczonego, ale łatwego w użyciu interfejsu do złożonego podsystemu.
- gdy chcesz ustrukturyzować podsystem w warstwy.

5.5 most (bridge)

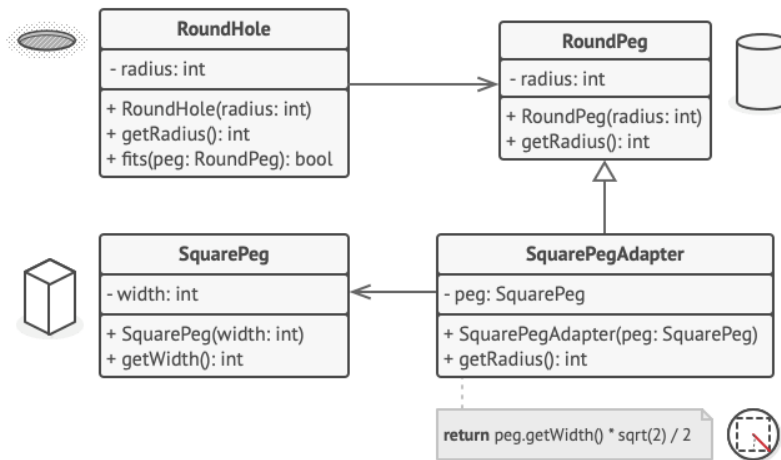


Pierwotna hierarchia klas podzielona na dwie części: urządzenia i piloty zdalnego sterowania.

pozwalającym na rozdzielenie dużej klasy lub zestawu spokrewnionych klas na dwie hierarchie — abstrakcję oraz implementację. Nad oboma można wówczas pracować niezależnie.

- gdy chcesz rozdzielić i przeorganizować monolityczną klasę posiadającą wiele wariantów takiej samej funkcjonalności (na przykład, jeśli klasa ma współpracować z wieloma serwerami bazodanowymi).
- gdy chcesz rozszerzyć klasę na kilku niezależnych płaszczyznach.
- pozwala spełnić wymóg możliwości wyboru implementacji w trakcie działania programu.

5.6 adapter



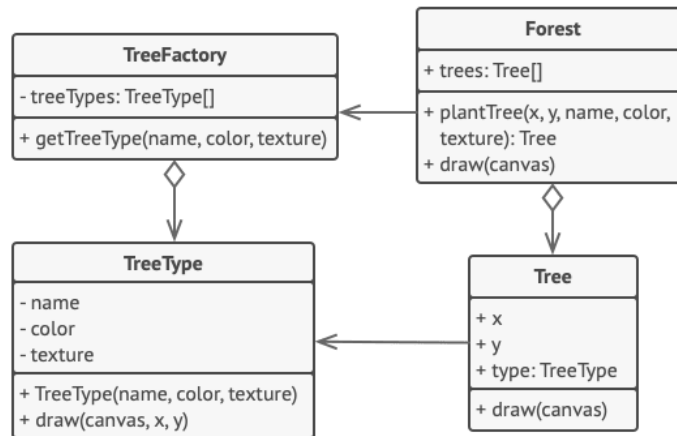
Adaptacja kwadratowych klocków do okrągłych otworów.

pozwalającym na współdziałanie ze sobą obiektów o niekompatybilnych interfejsach.

- gdy chcesz wykorzystać jakąś istniejącą klasę, ale jej interfejs nie jest kompatybilny z resztą twojego programu.
- gdy chcesz wykorzystać ponownie wiele istniejących podklas którym brakuje jakiejś wspólnej funkcjonalności, niedającej się dodać do ich nadklasy.

5.7 pylek (cache, flyweight)

W poniższym przykładzie, wzorzec **Pyłek** pomaga zredukować zużycie pamięci podczas renderowania milionów obiektów-drzew na ekranie.



pozwalającym zmieścić więcej obiektów w danej przestrzeni pamięci RAM poprzez współdzielenie części opisu ich stanów.

- gdy twój program musi pracować z wielką ilością obiektów, które ledwo mieszczą się w dostępnej pamięci RAM.

6 pytania zamkniete

6.1 zaznacz glownie rodzaje procesow biznesowych

procesy operacyjne, zarzadcze i pomocnicze

6.2 stosujac wzorzec <BLANK> gdy nie wiesz z gory jakie typy obiektow pojawiaja sie jakie twoim programie miedzy nimi zaleznosci

factory method

- 6.3 stosuj wzorzec <BLANK> gdy istnieje potrzeba wykonywanie jakiego działania na elementach złożonej struktury obiektów (jak drzewo obiektów)

iterator

- 6.4 stosuj wzorzec <BLANK> gdy musisz zaimplementować drzewiastą strukturę obiektów

composite

- 6.5 korzystając z wzorca <BLANK> gdy chcesz oszczędniej wykorzystać zasoby systemowe poprzez ponowne wykorzystanie już istniejących obiektów zamiast odbudowywać je raz za razem

factory method

- 6.6 stosuj wzorzec <BLANK> gdy chcesz przyjmować dodatków dodatkowe obowiązki obiektom w trakcie działania programu, bez pisania ... który z tych obiektów korzysta

DEKORATOR

- 6.7 stosowanie wzorca <BLANK> pozwala uprzątnąć logikę biznesową czynności pomocniczych

visitor

- 6.8 <BLANK> pozwala odizolować logikę biznesową klasy od szczegółów implementacyjnych algorytmów, które nie są istotne w kontekście tej logiki

strategy

- 6.9 stosuj wzorzec <BLANK> gdy chcesz aby kod klienta traktował zarówno proste, jak i złożone elementy jednako

composite

6.10 stosuj wzorzec <BLANK> gdy istnieje potrzeba wykonania jakiegoś działania na wszystkich elementach złożonej struktury obiektów (jak drzewo obiektów)

visitor

6.11 korzystaj z wzorca <BLANK> gdy zamierzasz pozwolić użytkownikowi twojej biblioteki lub frameworka rozbudowywać jej wewnętrzne komponenty

factory method

6.12 które stwierdzenia są prawdziwe, gdy aktor A uogólnia aktora B

- B może komunikować się z tymi samymi przypadkami użycia co A
- B dziedziczy wszystkie związki A

6.13 które z poniższych stwierdzeń charakteryzują przypadki użycia

- przypadki użycia posiadają procedury stosowane w systemie
- ???przypadki użycia posiadają funkcjonalność lub zachowanie oczekiwane od opracowanego systemu???

6.14 wybierz zdania prawdziwe określające pojęcie błędów logicznych w oprogramowaniu

- większość wysiłków, podczas testowania programu, koncentruje się na ich znajdowaniu
- błąd logiczny powstaje, gdy zewnętrzne zdarzenie lub nie wykryty błąd składni zmusza proces do zatrzymania swojego działania

6.15 Proces określania wymagań dla systemu informatycznego można podzielić na następujące fazy

- Faza ustalania wymagań
- Faza specyfikacji wymagań
- Faza atestacji wymagań

6.16 Kontekst systemu

- Jest częścią środowiska systemu, która jest istotna ze względu na definiowanie i zrozumienie wymagań dla tworzonego systemu.
- Odseparowania kontekstu systemu od samego systemu oraz części rzeczywistości, która jest nieistotna dla tworzonego systemu. Definiowanie granic systemu polega na podjęciu decyzji, które aspekty będą implementowane w systemie, a które należą tylko do jego kontekstu.

6.17 Zaznacz główne rodzaje procesów biznesowych

- Procesy operacyjne
- Procesy zarządzania
- Procesy pomocnicze

6.18 Strukturalne wzorce projektowe to

- Adapter
- Most
- Kompozyt
- Dekorator
- Fasada
- Pylek
- Pełnomocnik

6.19 Wybierz zdania prawdziwie określające pojęcie złożoności cyklometrycznej

- Złożoność cyklometryczna jest to liczba niezależnych ścieżek w programie
- Złożoność cyklometryczna jest podstawową miarą złożoności dowolnego fragmentu kodu programu

6.20 Które z poniższych stwierdzeń charakteryzuje przypadki użycia

- Przypadki użycia opisują procedury stosowane w systemie
- Przypadki użycia opisują funkcjonalność lub zachowanie oczekiwane od opracowywanego systemu

7 pytania otwarte odpowiedz

7.1 kiedy nie należy stosować dziedziczenia opisz przynajmniej dwa przypadki

7.2 opisać silną agregację

8 pytania otwarte modelio

8.1 system w którym pracownicy mogą być również klientami, zaproponuj trzy rozwiązania opisując ich wady i zalety

8.2 zamodeluj podsystem obsługi klienta w sklepie internetowym. Zacznij od opisu wymagań i procesów