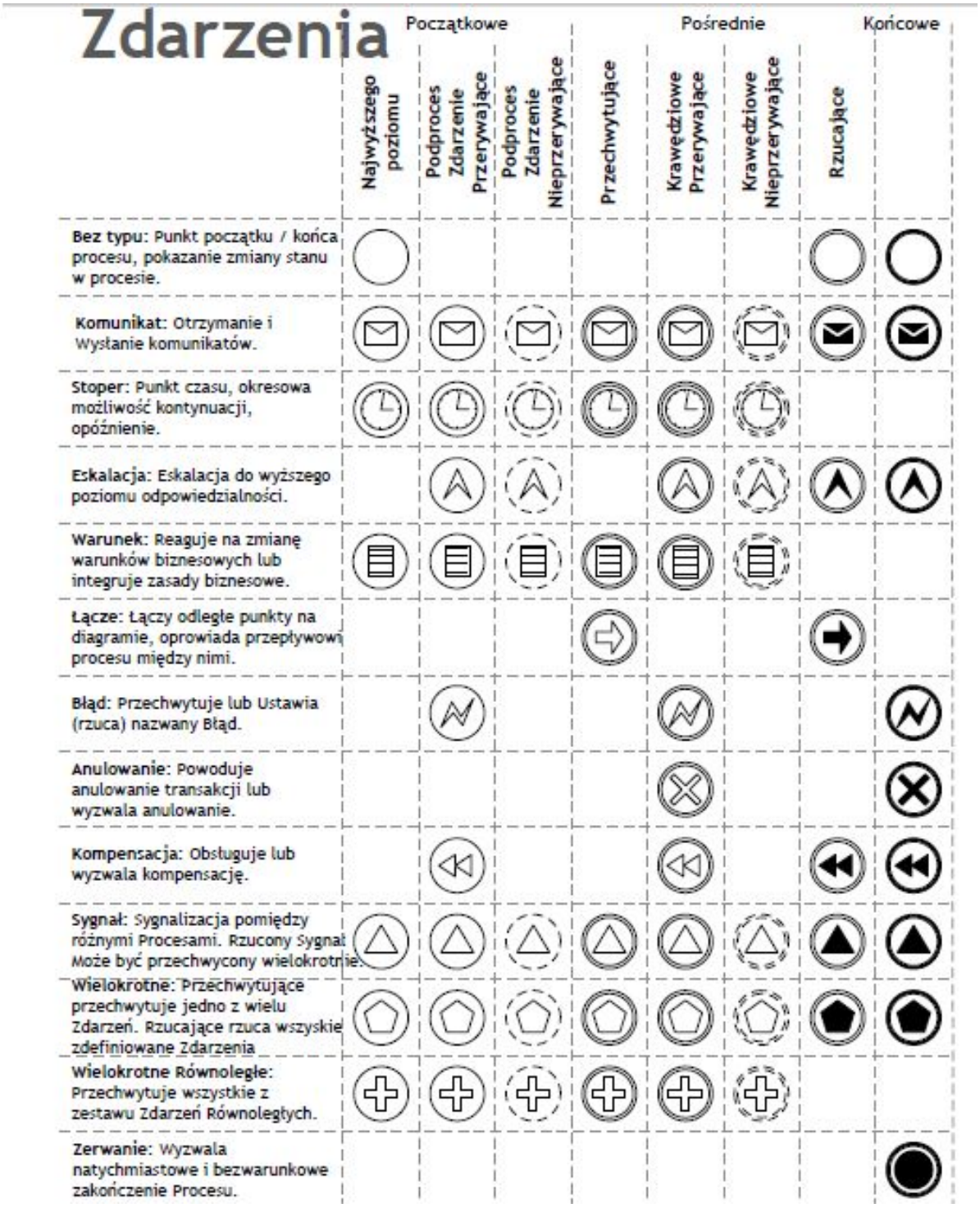
Contents

1 UML


Związki między klasami
Słabszy związek między klasami
Silniejszy związek między klasami
Zależność
strzałka przerywana
Asocjacja
pojedyncza linia łącząca
Agregacja częściowa
pusta strzałka zakończona rombem
Agregacja całkowita
pełna strzałka zakończona rombem
Dziedziczenie
pusta strzałka
W sytuacji gdy obiekty jednej klasy działają, wykorzystując przelotnie obiekty innej
W sytuacji gdy obiekty jednej klasy działają, wykorzystując obiekty innej przez pewnien dłuższy czas
W sytuacji kiedy jedna klasa zawiera, ale jednocześnie współdzieli odwołanie do obiektów innej
W sytuacji gdy jedna klasa zawiera obiekty innej
Kiedy jedna klasa jest rodzajem innej

2 BMPN

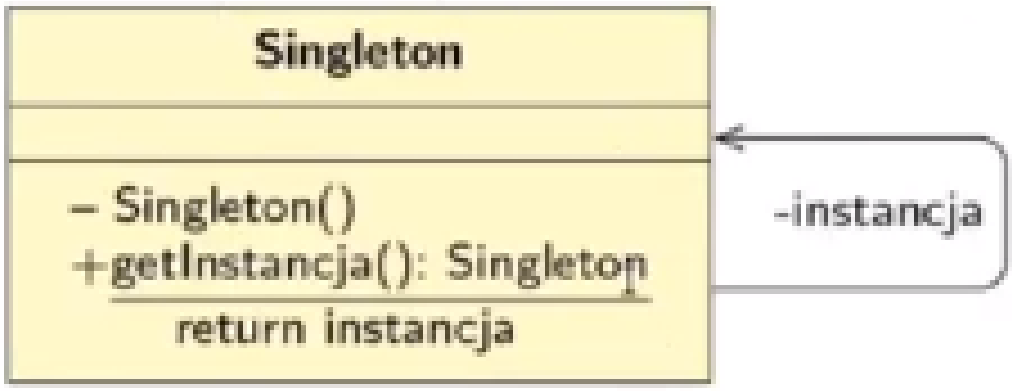
2.1 zdarzenia



3 wzorce kreacyjne

3.1 singleton

`https://refactoring.guru/design-patterns/singleton`



Rysunek 9: Schemat wzorca *Singleton*

- zagwaratowac ze 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.

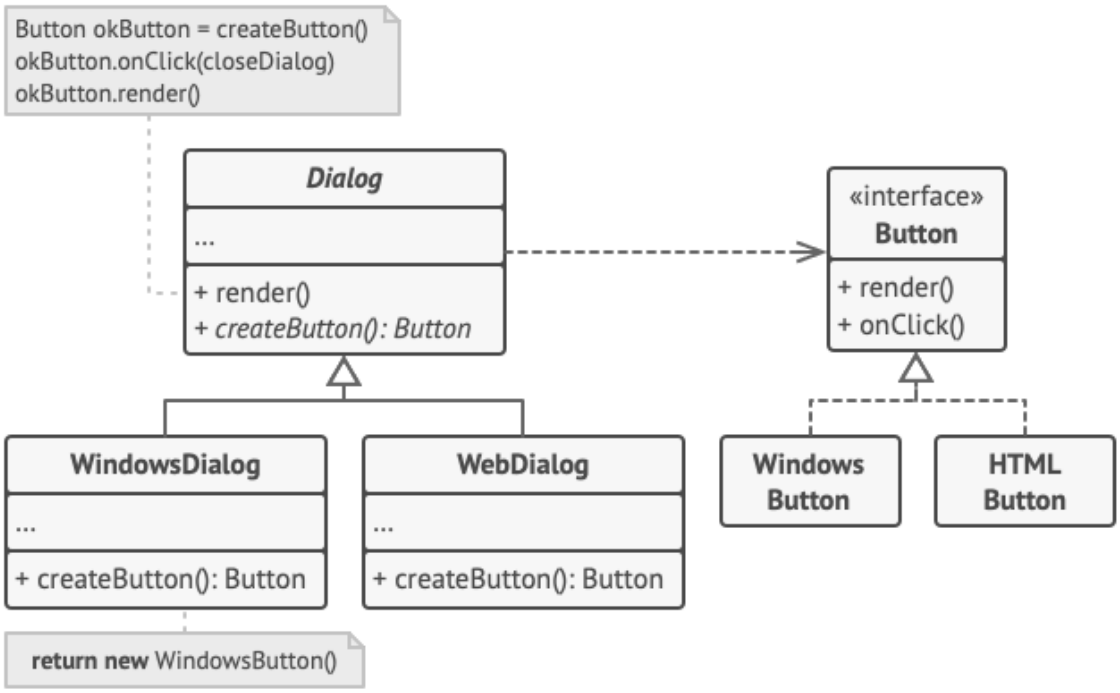
3.1.1 implementacja

```
class singleton {

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

return singleton;
}
};
```

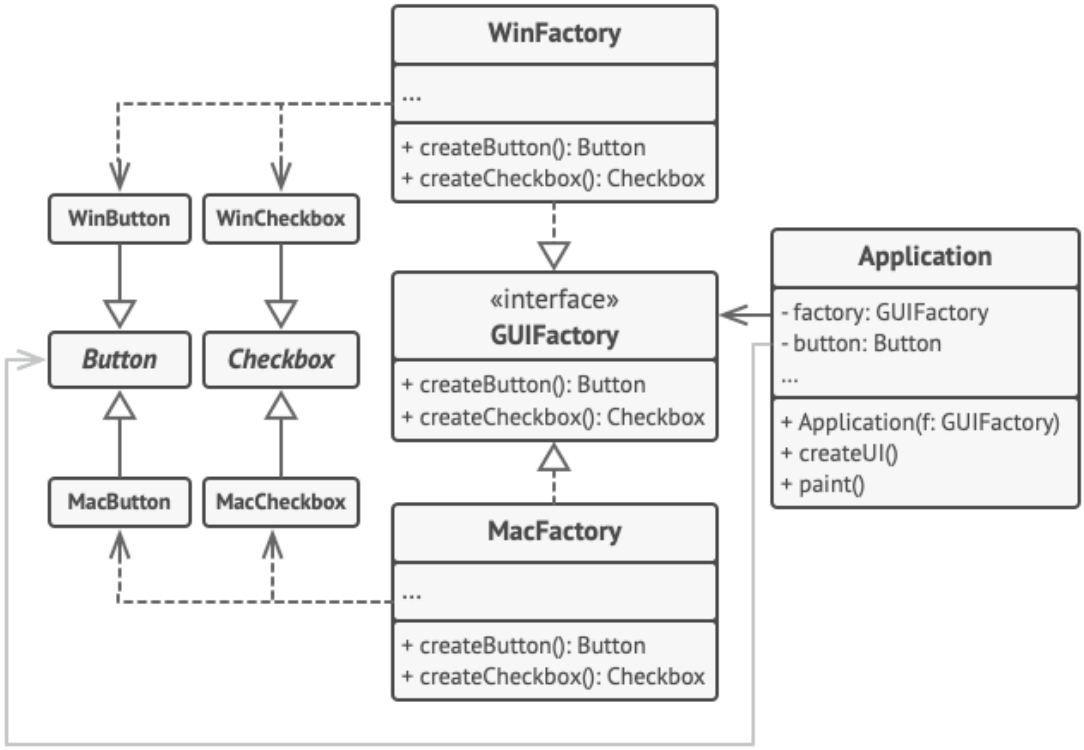
3.2 metoda wytworcza (factory method)



- 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 wariancie.

3.4 budowniczy (builder)

- **SKLADANIE OBIEKTU Z MALYCH CZESCI** 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 wzorzec 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

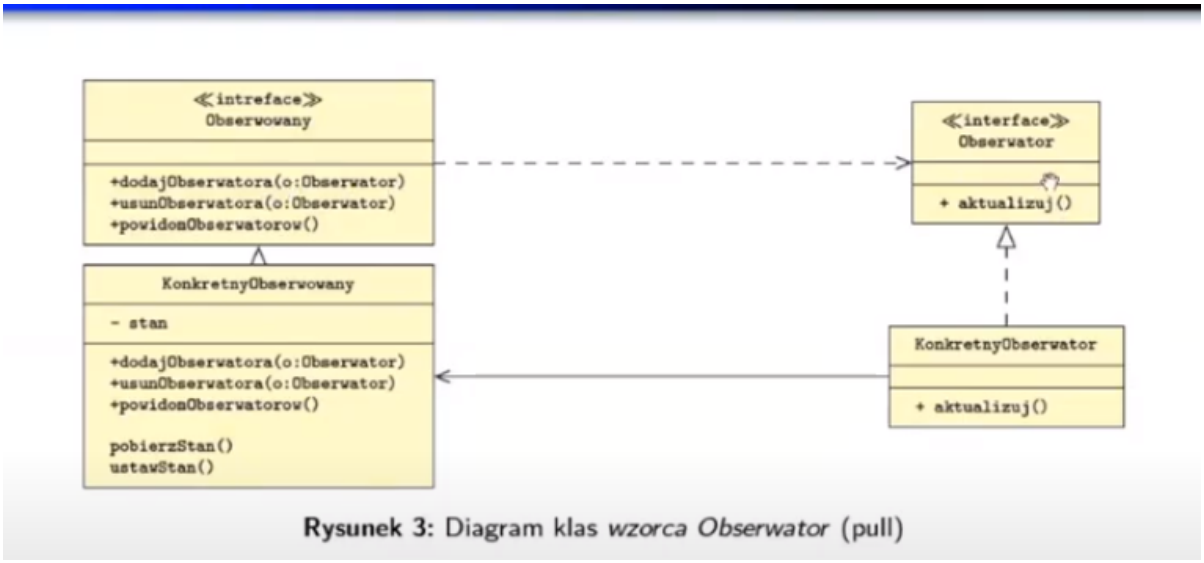
Wyobraź sobie jakiś skomplikowany obiekt, którego inicjalizacja jest pracochłonnym, wieloetapowym procesem obejmującym wiele pól i obiektów zagnieżdżonych. Taki kod inicjalizacyjny jest często wrzucany do wielgachnego konstruktora, przyjmującego mnóstwo parametrów. Albo jeszcze gorzej: kod taki rozrzucono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



Rysunek 3: Diagram klas *wzorca Obserwator* (pull)

- okresla zaleznosc jeden do wiele miedzy obiektami
- gdy jeden obiekt zmienia stan wszystkie obiekty od niego zalezne sa o tym automatycznie powiadamiane i uaktualniane (np. w kalkulatorze mamy 3 klasy wypisywania ktore maja w sobie string do wypisywania, kiedy wprowadzamy nowe dzialanie wszyskie sa updatowane)
- wydaje mi sie ze realizowany w grach -> bo trzeba updatowac stan obiektow a one musza znac stan innych
- gdy zmiany stanu jednego obiektu mogą wymagać zmiany w innych obiektach, a konkretny zestaw obiektów nie jest zawczasu 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 obiektow trzeba zmienic

4.1.2 problem

obiekt powinien byc w stanie powiadamiac inne obiekty, nie przyjmujac zadnych zalozen co do tego, co te obiekty reprezentuja - wynikiem sa luzniejsze powiazania miedzy obiektami

4.1.3 implementacja

`https://refactoring.guru/design-patterns/observer` zagwarantowanie ze przed rozeslaniem powiadomienia stan obserwowanergo jest wewnetrznie spojny

model push (obserwowany wysyla wszystkie informacje same) model pull (obserwowany wysyla POWIADOMIENIE a kazdy inny pyta sie to czego potrzebuje z jakiejs zmiany)

4.2 Stan (state)

`https://refactoring.guru/design-patterns/state`

- umozliwia obiektowi zmiane zachowania, gdy zmienia sie jego stan wewnetrzny (np. ktos 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.

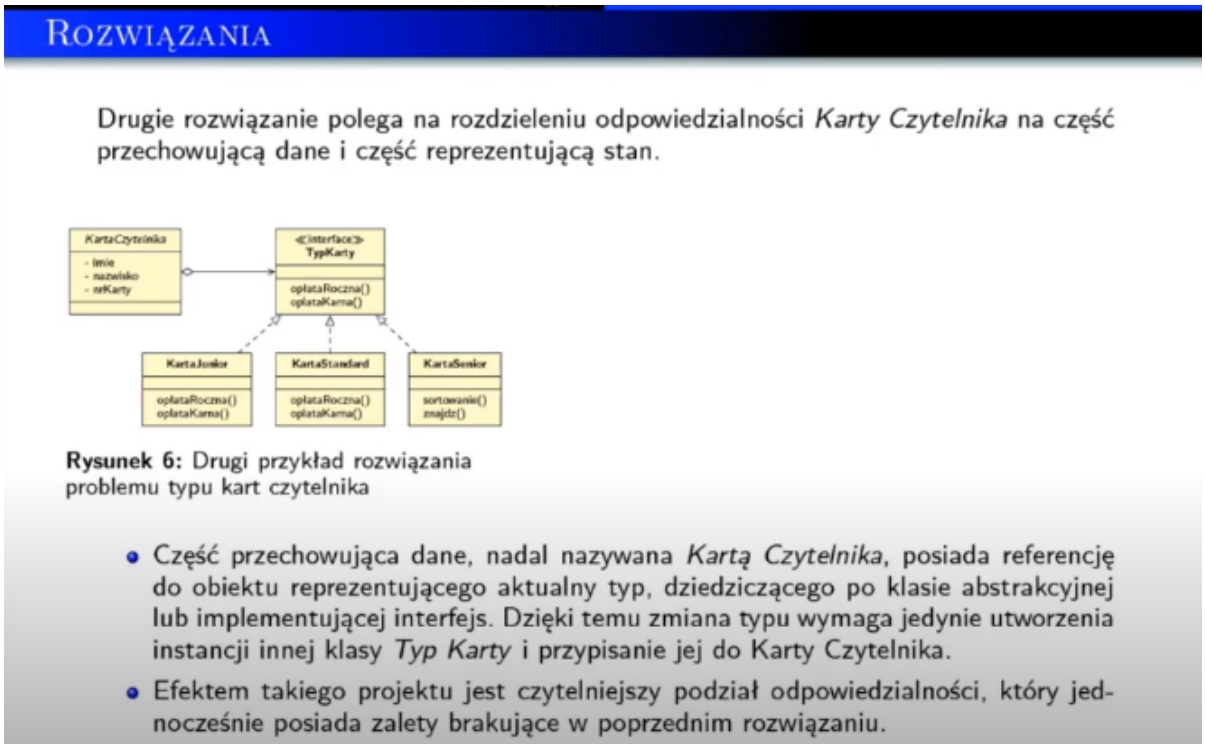
4.2.1 kontekst

- zachowanie obiektu zalezy od jego stanu, a obiekt ten musi zmieniac swoje zachowanie w czasie wykonywania programu w zaleznosci od stanu
- operacje zawieraja duze, wieloczesciowe instrukcje warunkowe ktore zaleza od stanu obiektu - wzorzec State przenosi kazde rozgalezienie do specjalnej klasy z inna implementacja np. pobierz podatek

4.2.2 problem

chemy umozliwic obiektowi zmiane zachowania w momencie zmiany wewnetrzengo stanu obiektu hermetyzujac stan w postaci klasy

4.2.3 implementacja



4.3 strategia (strategy)

`https://refactoring.guru/design-patterns/strategy`

- roznica w implementacji ze stanem
- w stanie klient nie widzi z kim dziala
- w strategi klient zna wewnetrzna strukture - wie kto uzywa
- pomaga 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 zbytniej jego zależności od innych komponentow

gdy zauważysz, że tworzysz mnóstwo podklas komponentu tylko aby móc ponownie użyć jakieś 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 Odwiedzajacy (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.

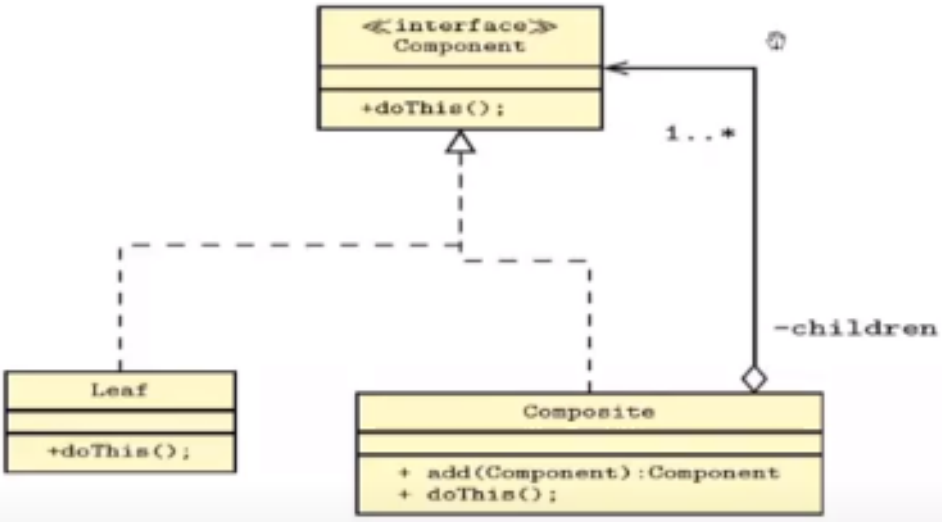
4.8 polecenie

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 kolejkowanie 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)



Rysunek 2: Schemat *wzorca Kompozyt*

TLDR: Drzewko w ktorym lisc zawiera siebie + liste dzieci

- zadaniem jest laczenie obiektow w struktura tak, ze reprezentuja hierarchie czesci-calosci, unifikujac dostep do kolekcji jak i pojedynczego obiektu.
- umozliwa to klientom jednolite traktowanie pojedynczych obiektow i rowniez 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.

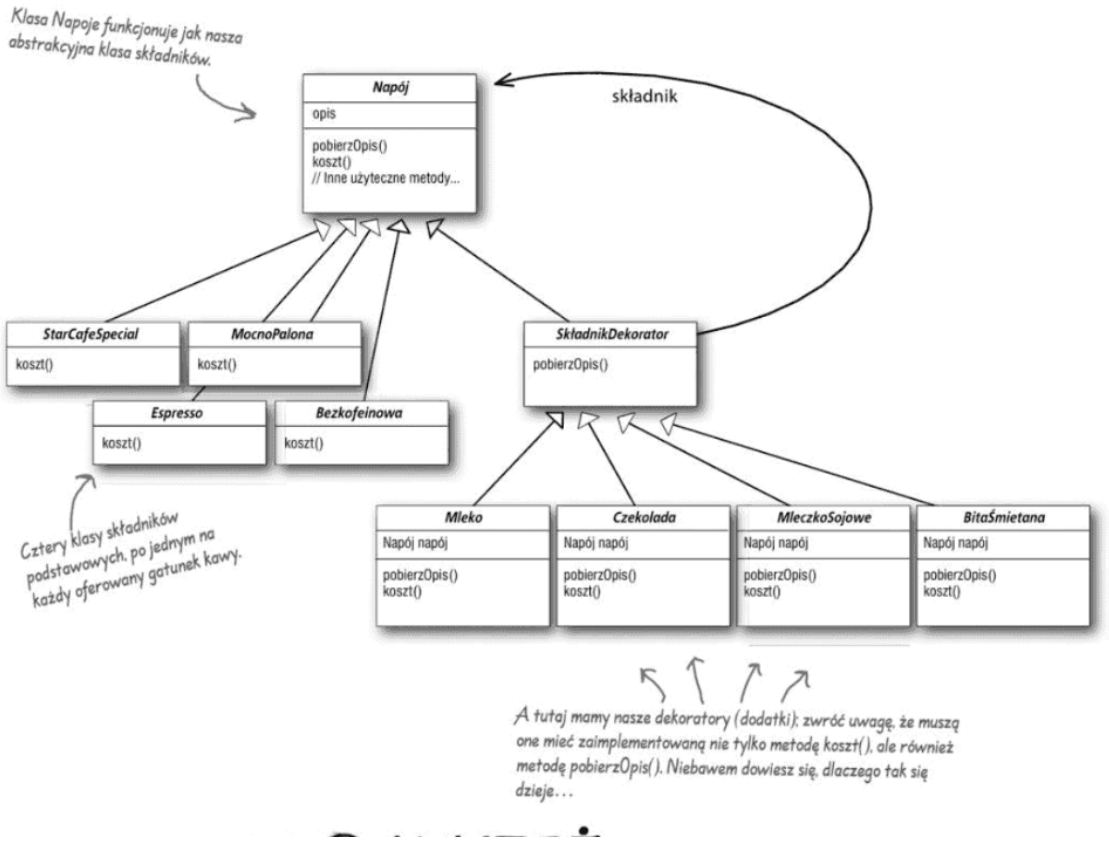
5.1.1 kontekst

chcemy przedstawic hierarchie obiektow czesc-calosc Hierarchia obiektow ma wspolna klase bazowa (abstrakcyjną)

5.1.2 problem

chcemy, aby klienci mogli ignorowac roznice miedzy zlozeniami obiektow a pojedynczymi obiektami - klienci beda wtedy jednakowo traktowac wszyskie obiekty wystepujace w strukturze

5.2 dekorator (decorator)



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

- dodawanie dodatkowej funkcjonalnosci do obiektow
- 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.

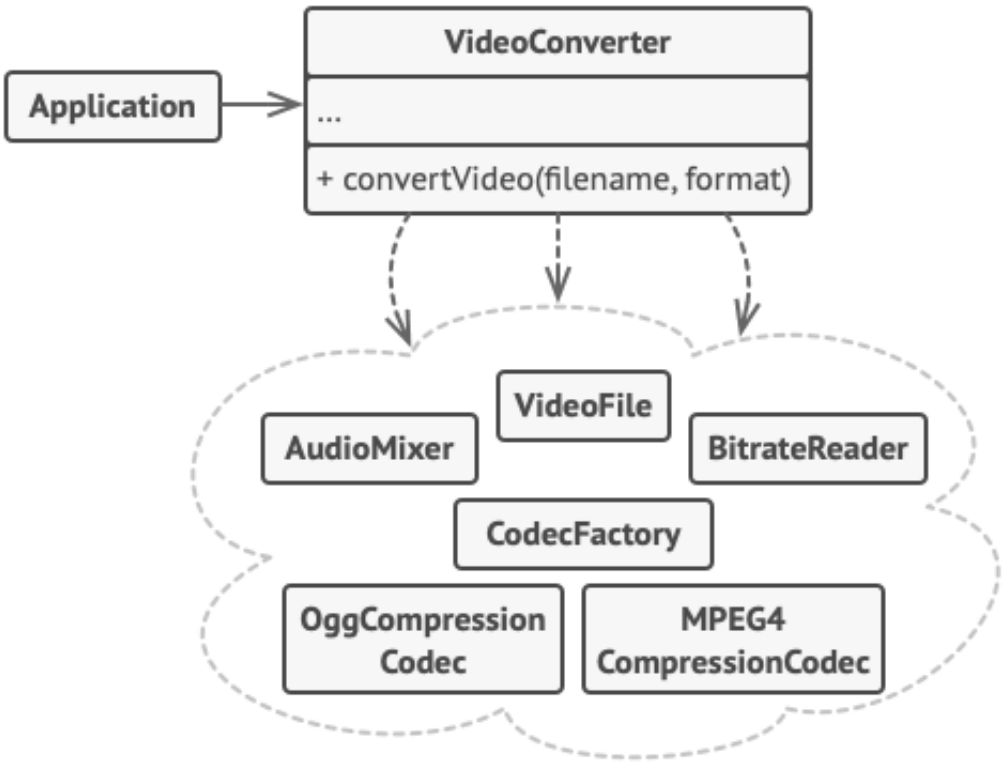
5.3 pelnomocnik (proxy)

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.

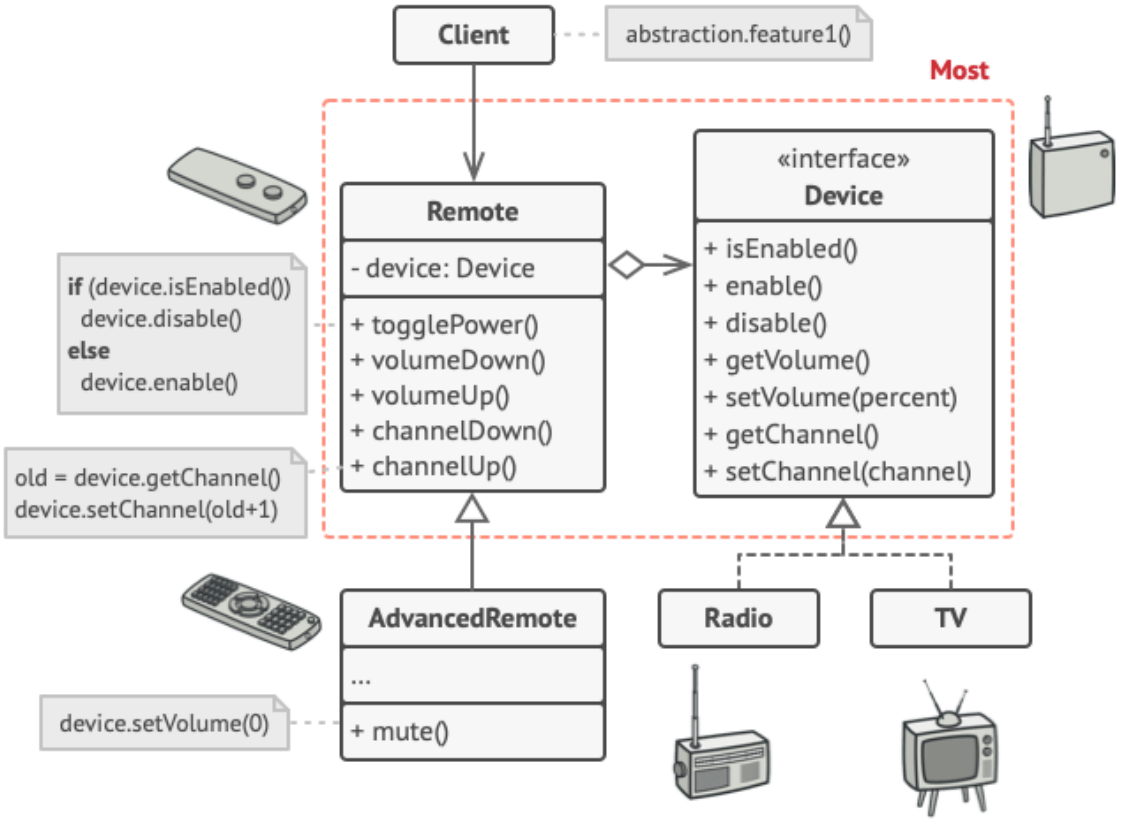
5.4 fasada (facade)



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)

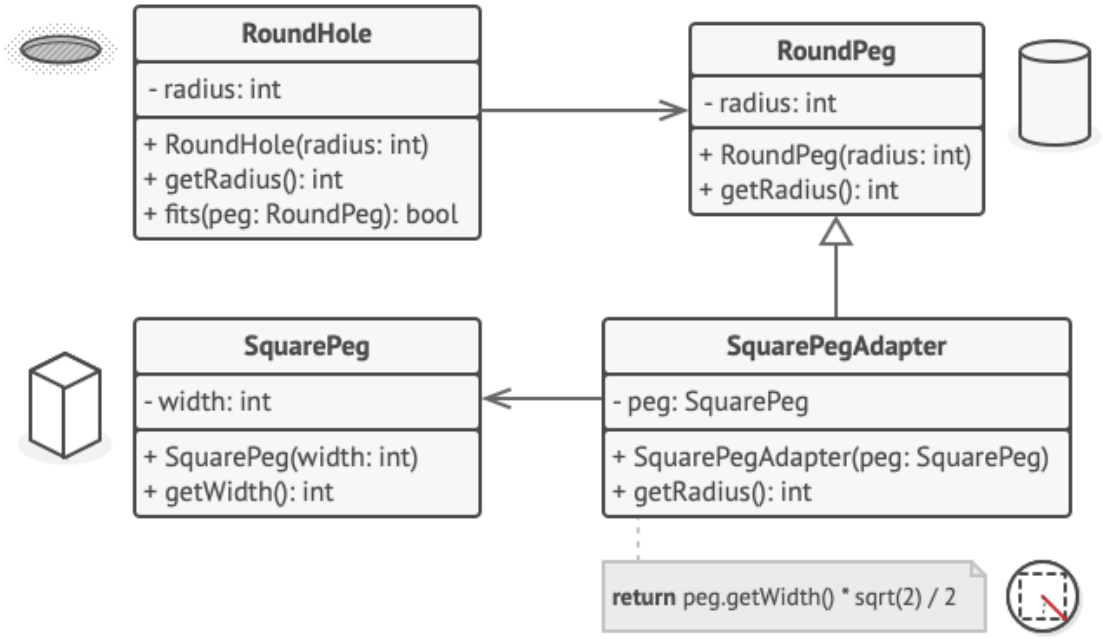


Pierwotna hierarchia klas podzielona na dwie części: urządzenia i piloty zdalnego sterowania.

pozwalającym na rozdzielenie dużej klasy lub zestawu spokrewnionych klas na dwie hierarchie — abstrakcję oraz implementację. Nad obiema 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



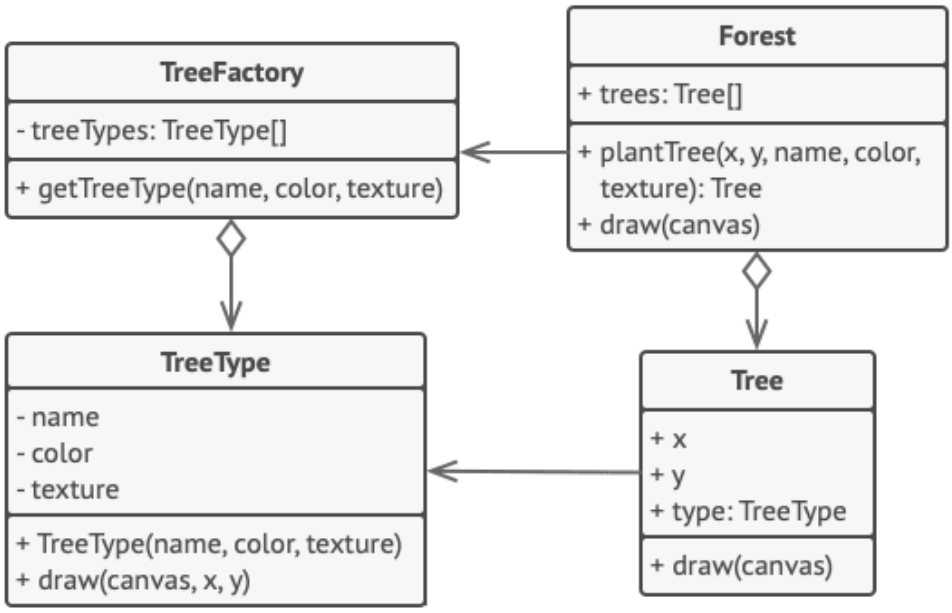
Adaptacja kwadratowych klocków do okrągłych otworów.

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, zarzadzcze 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 stosujac wzorzec <BLANK> gdy istnieje potrzeba wykonywanie jakiego dzialania na elementach zlozonej strukty obiektow (jak drzewo obiektow)

iterator

6.4 stosuj wzorzec <BLANK> gdy musisz zaimplementowac drzewiasta strukture obiektow

composite

6.5 korzystajac z wzorcza <BLANK> gdy chcesz oszczedniej wykorzystac zasoby systemowe poprzez ponownie wykorzystanie juz istniejacych obiektow zamiast odbudowywyac je raz za razem

factory method

6.6 stosuj wzorczec <BLANK> gdy chcesz przyjmowac dodatkow dodatkowe obowiazki obiektom w trajcie dziala programu, bez pisania ... ktory z tych obiektow korzysta

DEKORATOR

6.7 stosowanie wzorcza <BLANK> pozwala uprzatnac logike biznesowa czynnosci pomocniczych

visitor

6.8 <BLANK> pozwala odizolowac logike biznesowa klasy od szczegolow implementacyjnych algorytmow, ktore nie sa istotne w kontekscie tej logiki

strategy

6.9 stosuj wzorzec <BLANK> gdy chcesz aby kod klienci traktowal zarowno proste, jak i zlozone elementy jednakowo

composite

6.10 stosuj wzorzec <BLANK> gdy istnieje potrzeba wykonania jakiegos na dzialania na wszystkich elementacj zlozonej struktury obiektow (jak drzewo obiektow)

vistor

6.11 korzystaj z wzorcza <BLANK> gdy zamierzasz pozwolic uzytkujacym twa biblioteke lub framework rozbudowywac jej wewnetrzne komponenty

factory method

6.12 ktore stwierdzenia sa prawdziwy, gdy aktor A uogulnia aktora B

- B moze komunikowac sie z tymi samymi przypadkami uzycia co A
- B dziedziczy wszystkie zwiazki A

6.13 ktore z ponizszych stwierdzen charaktyryzuja przypadki uzycia

- przypadki uzycia posuja procedyury stosowane w systemie
- ???przypadki uzycia posuja funkcjonalnosc lub zachowanie oczekiwane od opracowanego systemu???

6.14 wybierz zdania prawdziwe okreslajace pojecie bledu logicznego w oprogramowaniu

- wiekszosc wysilkow, podzas testowania programu, koncentruje sie na ich znajdowaniu
- blad logiczny powstaje, gdy zewnetrzne zdarzenie lub nie wykryt blad skladni zmusza proces do zatrzymania swojego dzialania

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
- Pyłek
- 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 opisują funkcjonalność lub zachowanie oczekiwane od opracowywanego systemu

7 pytania otwarte odpowiedz

7.1 kiedy nie nalezy stosowac dziedziczenia opisz przynajmniej dwa przypadki

7.2 opisac silna agregacja

8 pytania otwarte modelio

8.1 system w ktorym pracownicy moga byc rowniez klientami, zaproponuj trzy rozwiazania opisujac i wady i zalety

8.2 zamodeluj podsystem obslugi klienta w sklepie internetowych Zacznij od opisu wymagan i procesow