Architektura oprogramowania

Instytut Informatyki Stosowanej PŁ

Wykład 5

Architektura oprogramowania

Instytut Informatyki Stosowanej PŁ

Definicja architektury

architektura [łac. < gr. architéktōn ‘budowniczy’],

sztuka tworzenia ładu w otoczeniu w celu dostosowania go do zaspokojenia wielorakich fizycznych, materialnych i kulturowych potrzeb ludzi przez planową przemianę naturalnego środowiska oraz budowanie form i wydzielanie przestrzeni o różnym przeznaczeniu.
http://encyklopedia.pwn.pl/

Artefakty architektoniczne świata rzeczywistego

- Normy podstawowe - Podstawy wymiarowania. Proporcje - Projektowanie - Realizacja budowy - Części budynku - Ogrzewanie. Wentylacja - Fizyka budowli. Ochrona budowli - Oświetlenie. Światło dzienne. Szkło - Okna. Drzwi - Schody. Dźwigi -Ulice - Ogrody - Dom. Pomieszczenia gospodarcze - Pomieszczenia mieszkalne - Pływalnie - Pralnie - Balkony - Drogi - Domy letniskowe - Rodzaje zabudowy mieszkaniowej - Schrony - Modernizacja starych budynków - Szkoły - Szkoły wyższe. Laboratoria - Ośrodki dla dzieci - Biblioteki. Biura. Banki - Budynki użyteczności publicznej - Sklepy - Magazyny - Warsztaty. Zakłady przemysłowe - Zmiana przeznaczenia - Budownictwo wiejskie - Koleje - Parkingi. Garaże. Stacje paliw - Porty lotnicze -Gastronomia - Hotele. Motele - Zoo -Teatry.Kina - Urządzenia sportowe - Szpitale - Domy seniora - Kościoły.Muzea - Cmentarze - Ochrona przeciwpożarowa - Miary. Normy

Oprogramowanie a konteksty architektoniczne

- System - projekt nad którym pracujemy (kontekst architektoniczny - dom).
- Podsystem - część systemu, która jest na tyle niezależna, że może być systemem w swoim kontekście np. kuchnia, pokój.
- Moduł - część systemu, która nie ma znaczenia poza jego kontekstem. Moduł jest hermetyczny i posiada API (np. noga od stołu).

Projektowanie architektoniczne

- Proces projektowania, którego celem jest identyfikacja podsystemów, na które należy podzielić system, oraz szkieletu pozwalającego na kontrolę oraz wzajemną komunikację podsystemów.
- Architektura oprogramowania jest rezultatem projektowania architektonicznego.

Reguły odkrywania architektury oprogramowania

- Nie ma wyraźnej granicy pomiędzy architekturą a projektem.
- Architektura jest ogólnym (najwyższym) poziomem projektu.
- Dokumentacja architektury musi zawierać wszystkie założenia kooperacji komponentów składowych systemu, w tym w zakresie ich odpowiedzialności za funkcjonalności.



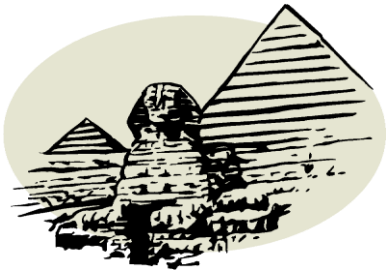


Reguły odkrywania architektury oprogramowania

- Architektura podobnych systemów powinna być również podobna.
- Stworzenie z elementów pewnej spójnej formy architektury może odbywać się przy pomocy stworzonych wcześniej wzorców.
- Wzorce i architektura istniejących systemów pomagają zrozumieć zasady konstrukcji i tworzenia nowego systemu.

Architektura oprogramowania – projekt architektoniczny

- Zbiór istotnych decyzji związanych z organizacją oprogramowania.
- Nie wskazuje sposobu realizacji.
- Wybór elementów strukturalnych i ich interfejsów.



- Współpraca pomiędzy elementami strukturalnymi określająca działanie systemu.
- Grupowanie elementów strukturalnych w podsystemy.

Architektura oprogramowania - projekt architektoniczny

- Pokazanie aspektu behawioralnego w trakcie rozwoju systemu.
- Określenie stylu architektnicznego tworzącego całość. Styl architektoniczny to przyporządkowanie do projektu określonego wzorca projektowego.
- **Ma stanowić skorelowaną z wizją rozwoju biznesowego, spójną wizję utrzymania i rozwoju systemu w przyszłości.**

Przeznaczenie architektury

Opis architektury systemu jest istotny dla:

- Procesu ewolucji systemu.
- Powtórnego użycia.
- Uzyskania dodatkowych cech jakościowych: wydajności, dostępności, przenośności, bezpieczeństwa.
- Przypisania zadań projektowych zespołom i/lub podwykonawcom.
- Wspomagania procesu podejmowania decyzji o wykorzystaniu gotowych komponentów.
- Wbudowania systemu w szerszy kontekst.

Zalety jawnej architektury

- Porozumienie stron - architektura może stanowić element dyskusji pomiędzy zainteresowanymi stronami.
- Analiza systemu – w kontekście odpowiedzi na pytanie czy system będzie w stanie zrealizować wymagania niefunkcjonalne.
- Wielokrotne użycie w wielkiej skali – architektura może być podstawą wielokrotnego użycia.

Architektura oprogramowania - pojęcia podstawowe

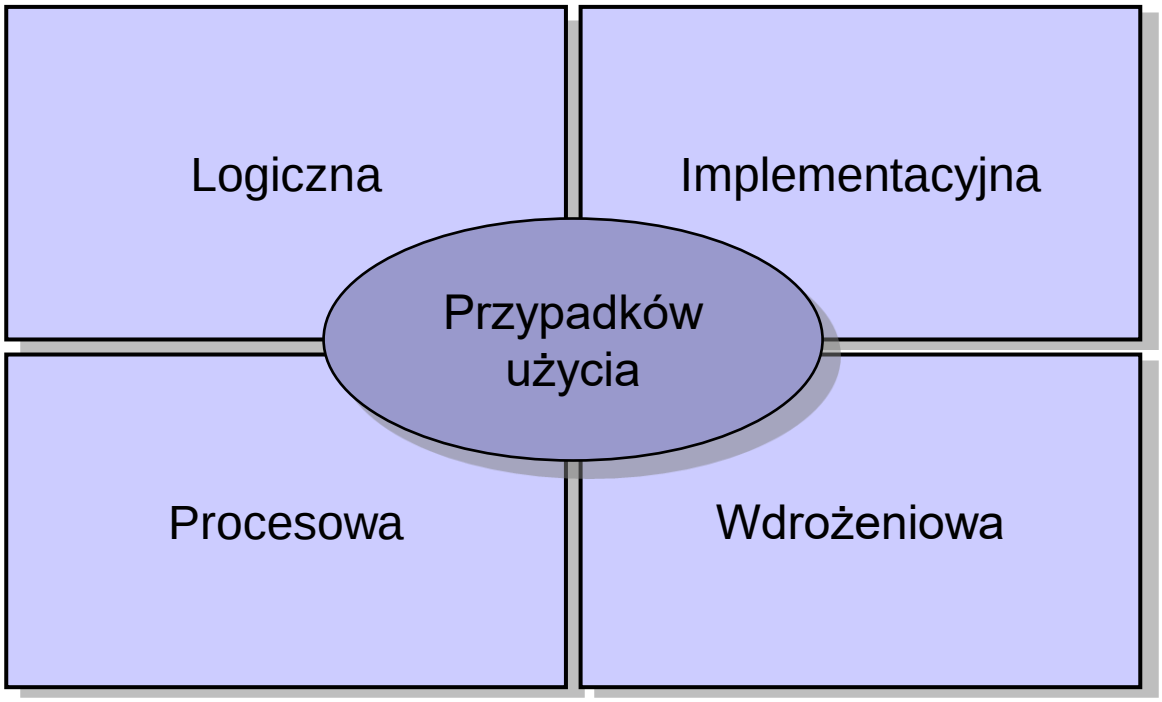
- **Komponen**t - jednostka oprogramowania zastępowana i aktualizowana niezależnie od systemu, hermetyzujący zbiór powiązanych funkcji (i/lub danych).
- **Biblioteka** – komponent łączony z programem i wywoływany za pomocą wywołań bezpośrednich.
- **Usługa** – komponent pozaprocesowy z którym komunikacja wymaga protokołu takiego jak np. usługi Internetowe czy inną formę zdalnego wywołania.
- W tej definicji komponent może być implementacją tak podsystemu jak i modułu.

Architektura oprogramowania – perspektywy (model) "4 + 1"

- Istnieje niewielki zbiór najczęściej występujących potrzeb przeglądania systemu do opisu jego charakterystki w aspekcie spełnienia wymagań.
- Umożliwia on pełen opis systemu i jego architektury.
- Perspektywy, które najlepiej ilustrują te potrzeby omówił Philippe Kruchten (1995) i nazwał perspektywami "4 + 1".

Architektura widziana z perspektywy „Przypadków Użycia”

Podzbiór modelu przypadków użycia obejmujący istotne z punktu widzenia architektury systemu przypadki użycia i scenariusze.



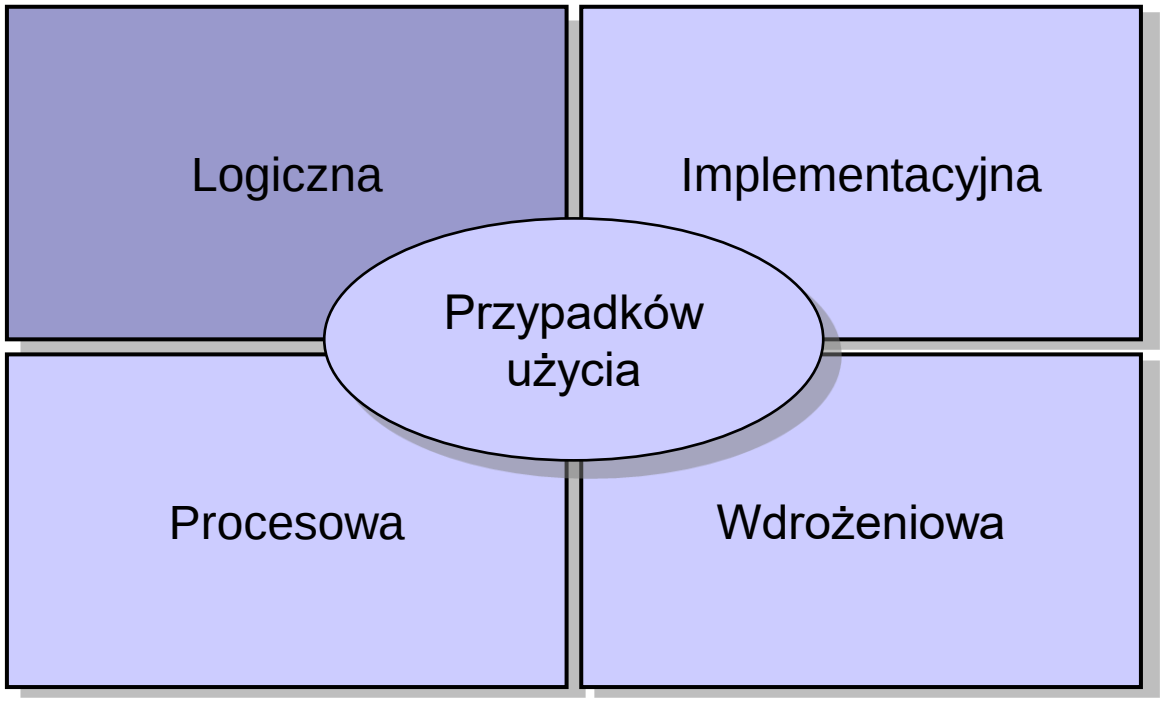
Scenariusz jako istota przypadku użycia

Scenariusze:

- opisują jak system udostępnia funkcjonalności użytkownikowi
- opisują jak system odpowiada na poszczególne akcje aktora
- ukazują zachowanie systemu jako czarnej skrzynki
- pozwalają kontrolować pozostałe perspektywy

Architektura widziana z perspektywy modelu logicznego

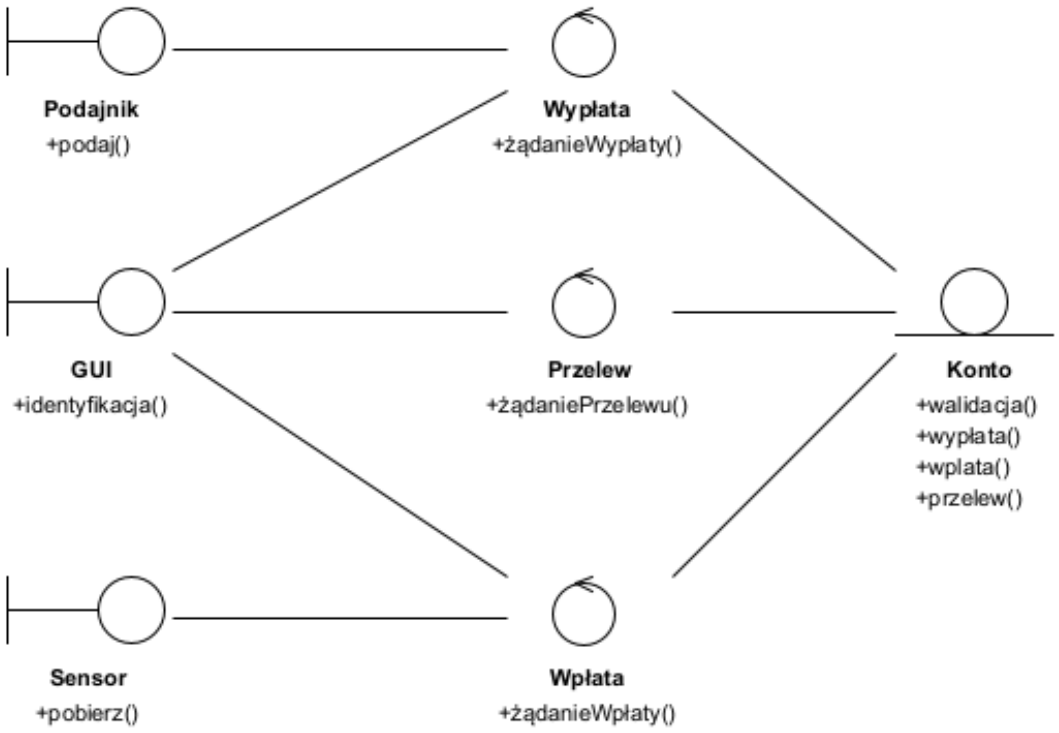
Zawiera najważniejsze klasy projektowe pogrupowane w pakiety i podsystemy. Pakiety i podsystemy są zorganizowane w warstwy (np. podsystem aplikacji, podsystem biznesowy, middleware, oprogramowanie systemowe).



Perspektywa logiczna - modelowanie wymagań funkcjonalnych

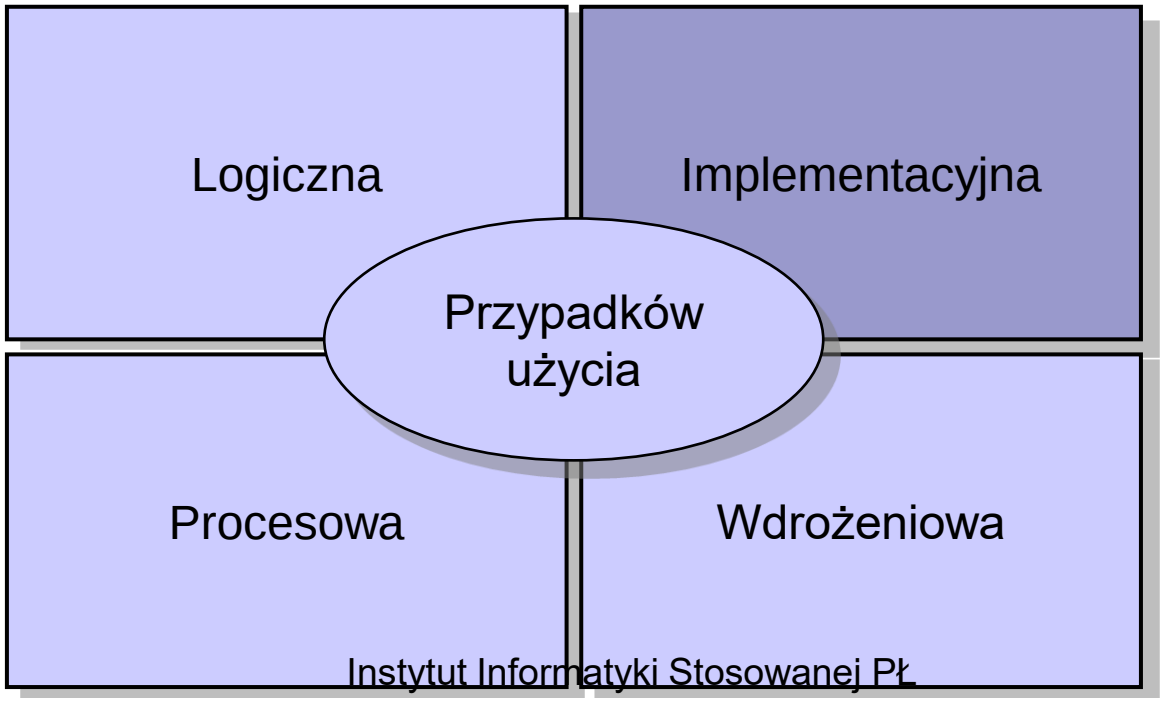
- Preferowany opis za pomocą modelu klas (lub obiektów) oraz modelu stanów dla istotnych klas (obiektów).
- Diagramy aktywności i diagramy sekwencji jako opis realizacji przypadków użycia.

Przykład diagramu klas



Architektura widziana z perspektywy implementacji

Opisuje organizacje modułów oprogramowania w pakietach. Opisuje również powiązania pomiędzy pakietami i klasami z modelu logicznego a pakietami i modułami z modelu implementacyjnego.



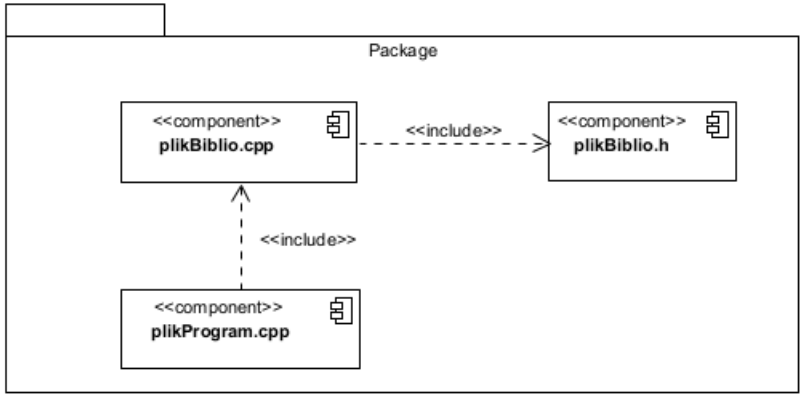
Perspektywa implementacyjna

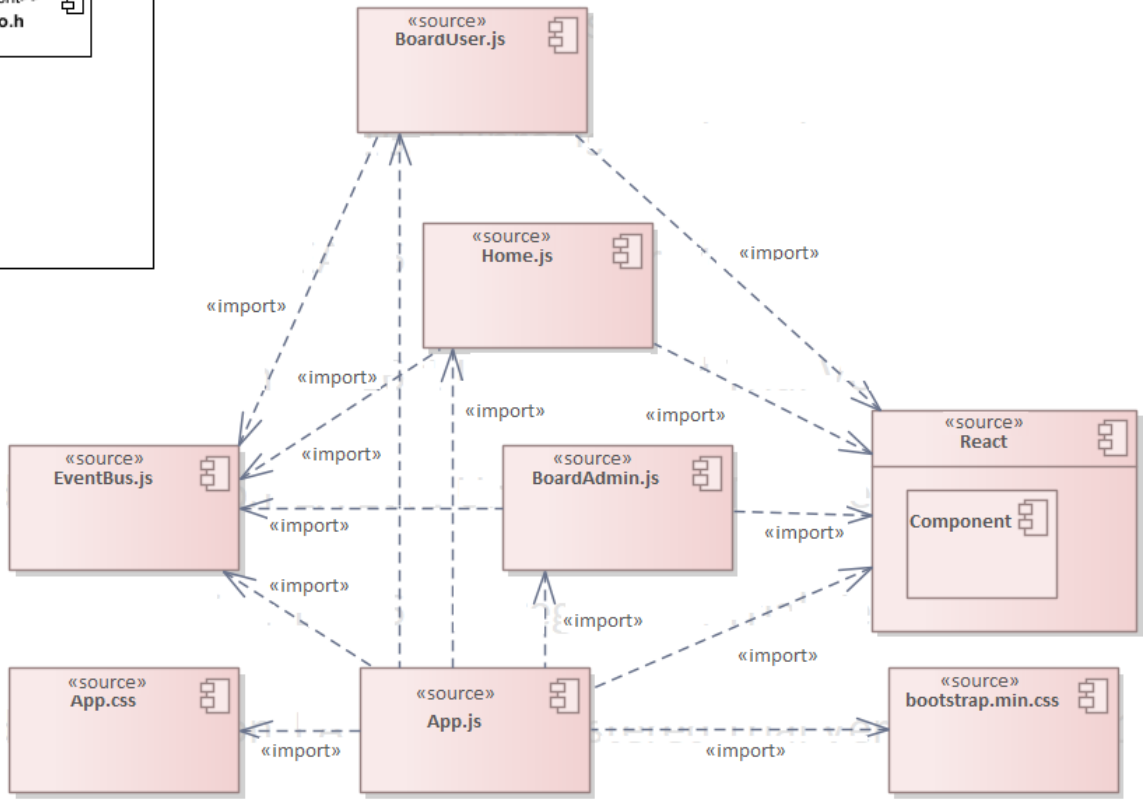
Opisuje podział oprogramowania na elementy składowe zawierające kod źródłowy: moduły, pakiety, biblioteki.

Ukazuje konfigurację systemu i jest używana do zarządzania wersjonowaniem konfiguracji.

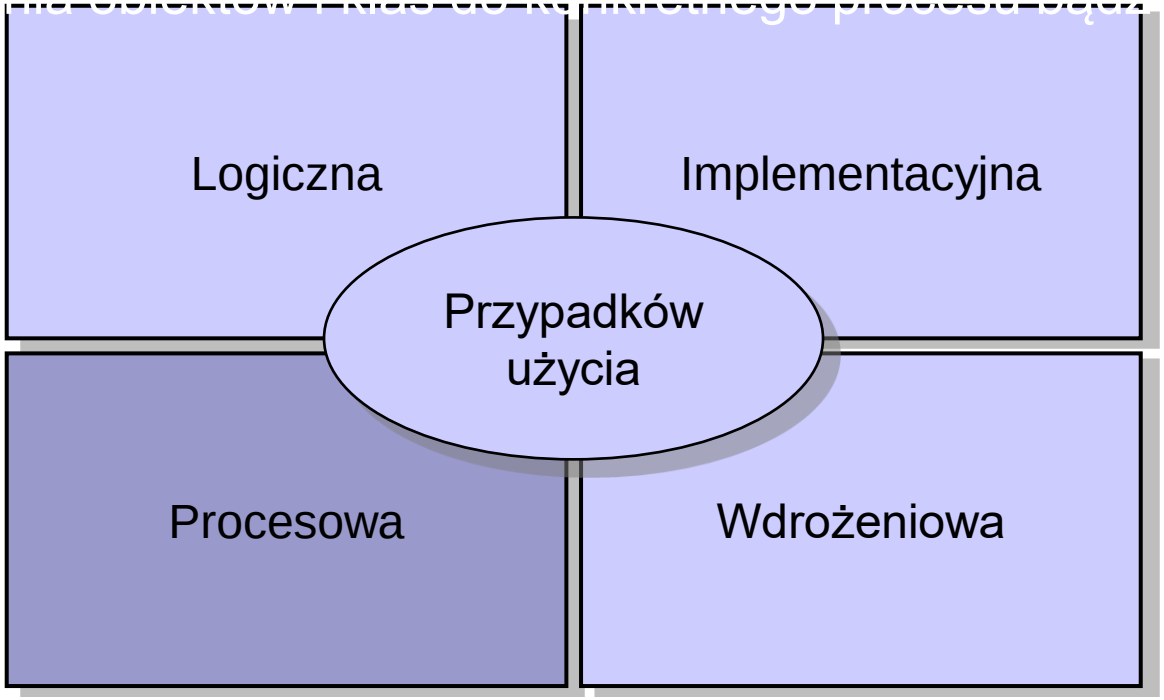
Stosuje się diagramy komponentów w obszarze statycznym modelu oraz diagramy interakcji, aktywności i stanów.

Przykład diagramu komponentów





Architektura widziana z perspektywy procesów i wątków



Perspektywa procesowa

Prezentuje klasy aktywne (źródła wątków) w pespektywie projektowej, uszczegóławiając model.

Opisuje funkcjonowanie systemu z punktu widzenia komunikacji oraz synchronizacji wątków i procesów systemu.

Charakteryzuje pewne elementy wymagań niefunkcjonalnych, takie jak przepustowość i efektywność systemu (przez aspekt wielowątkowosci).

Klasa aktywna

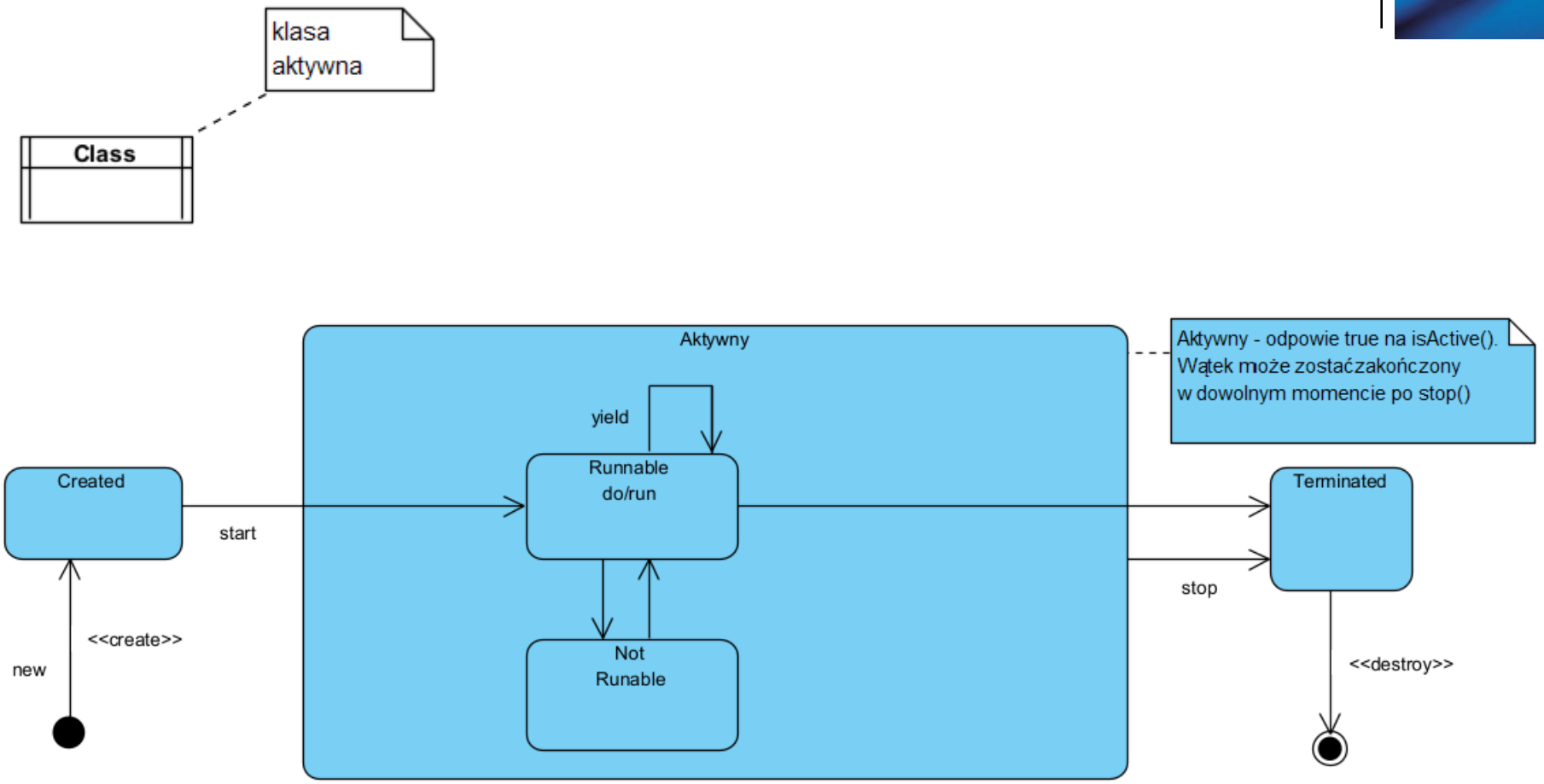
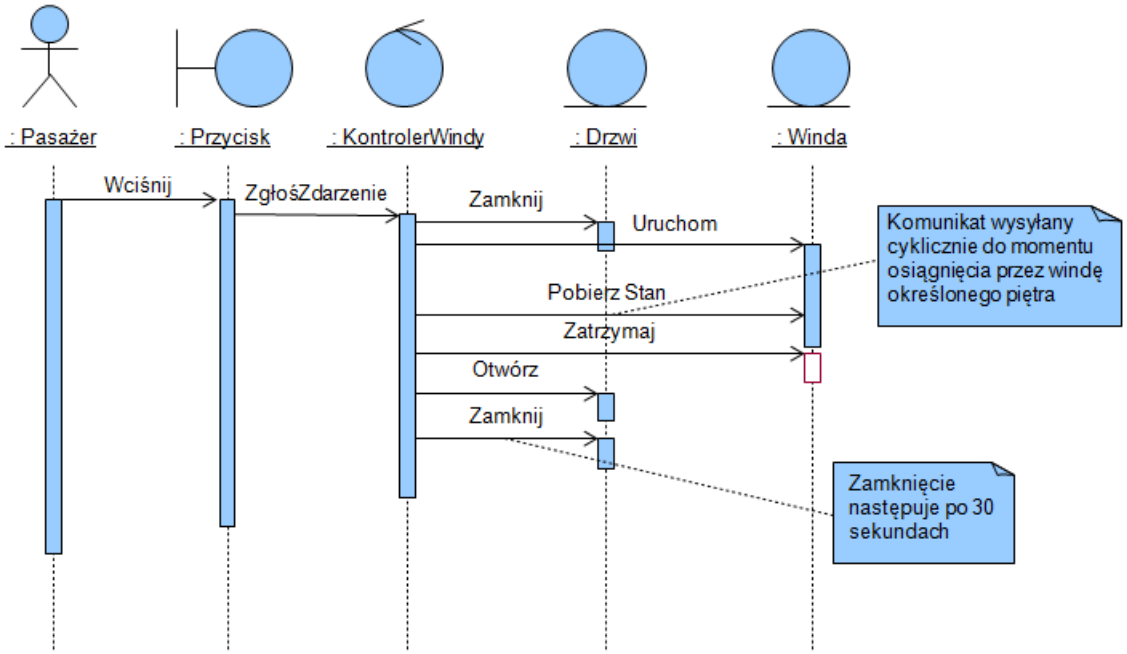
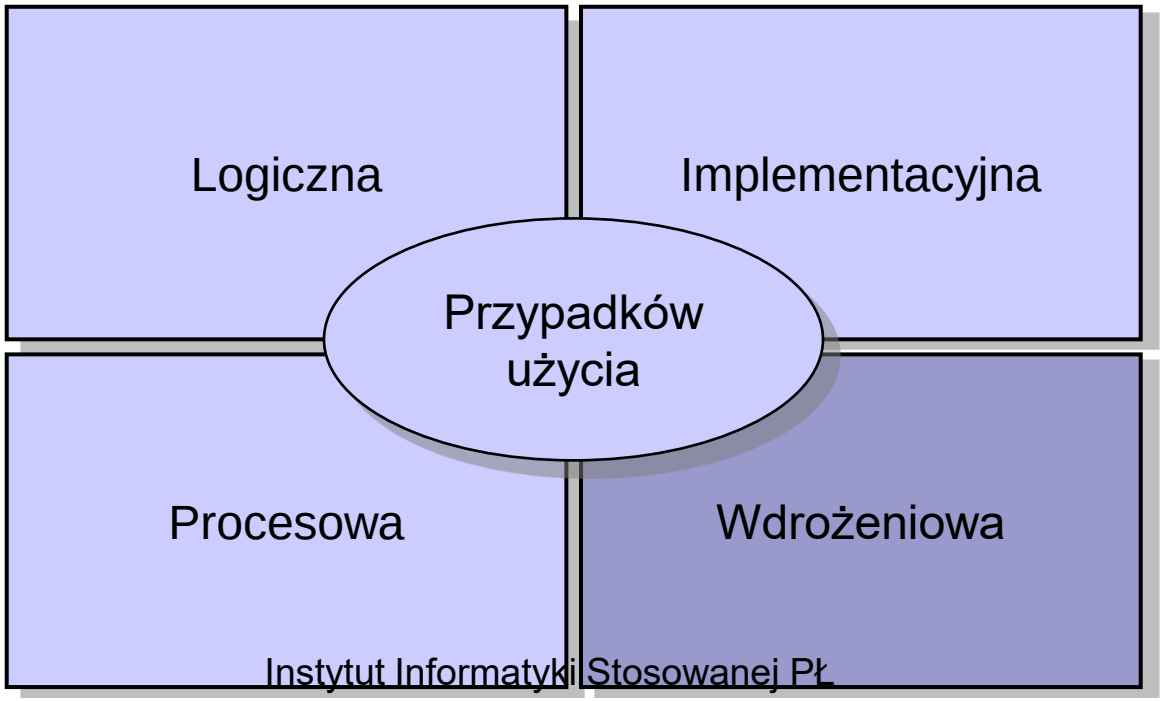


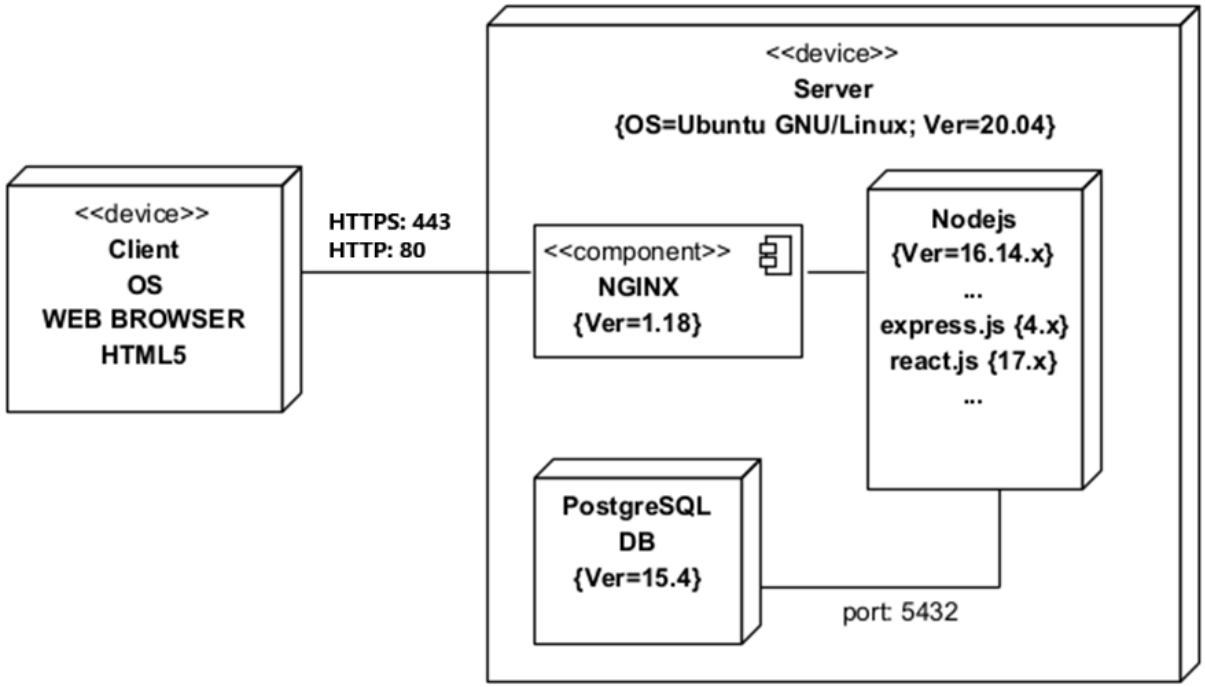
Diagram sekwencji



Architektura widziana z perspektywy wdrożeniowej



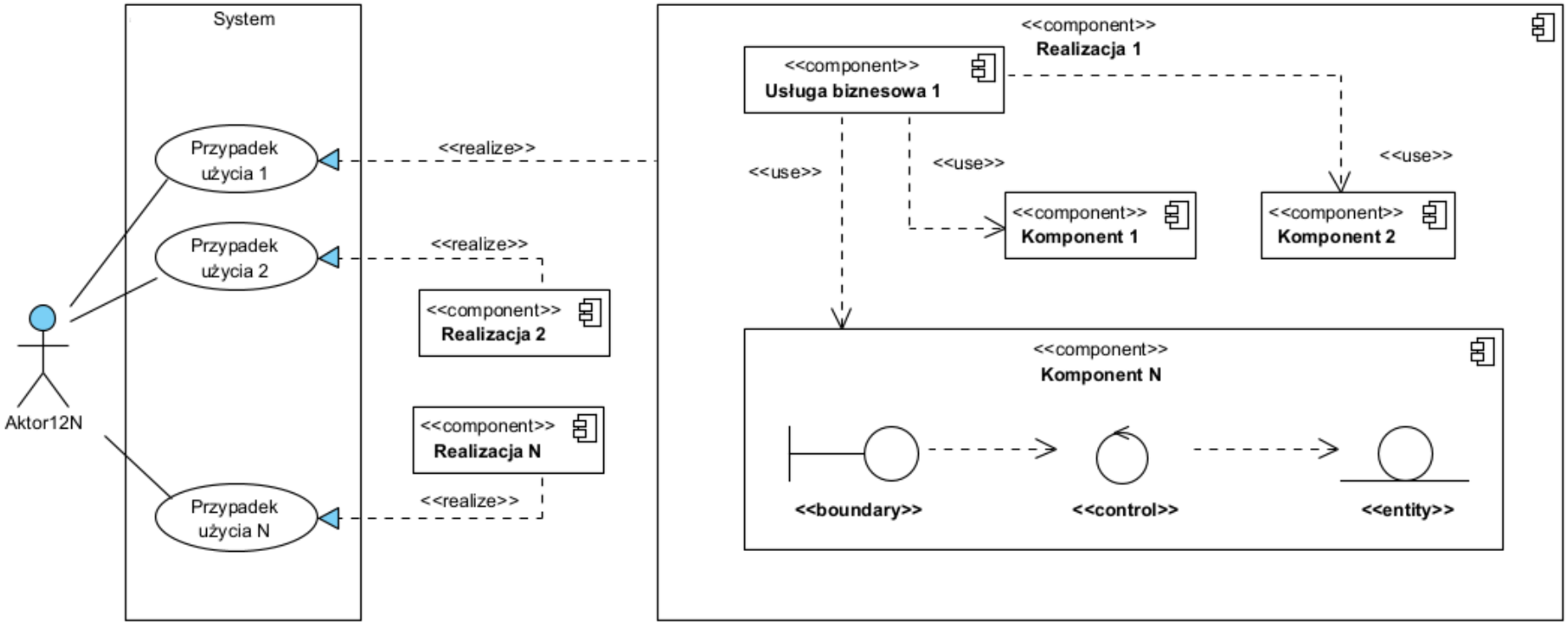
Perspektywa wdrożenia



Opisuje rozmieszczanie, dostarczanie i instalację systemu.

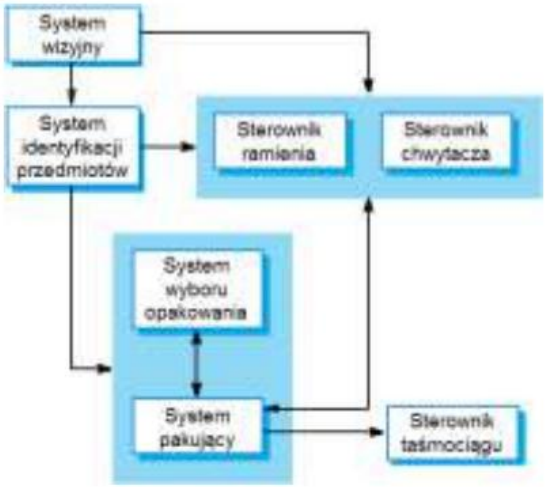
Pokazuje elementy struktury technicznej i umieszczone w jej węzłach komponenty.

Przykład – struktury realizujące przypadki użycia systemu



Dokumentacja architektury

- Diagramy blokowe o prostej strukturze ukazujące jednostki systemu i relacje między nimi.
    - Abstrakcyjna - bez reprezenacji natury powiązań pomiędzy komponentami i bez widocznych cech wewnętrznych podsystemów.



Projektowanie architektoniczne

- **Architektura małej skali -** architekura pojedynczych programów – sposób w jaki pojedynczy program jest dekomponowany do poziomu komponentów.
- **Architektura wielkoskalowa** - dotyczy systemów korporacjnych, które składaja się z systemów, aplikacji, komponentów. Systemy korporacyjne są rozproszone w sposób dopuszczający sytuację, w której poszczególne jego elementy mogą być zarządzane i stanowić własność różnych organizacji.

Wzorce architektoniczne

- Gotowe do wielokrotnego współdzielenia i wykorzystania formy rozwiązujące popularne problemy architektoniczne.
- Rama/zrąb architektury (ang. architectural framework) lub infrastruktura (ang. middleware) – zbiór komponentów na których buduje się określony rodzaj architektury.
- Opis wzorca powinien określać jego stosowalność.
- Wzorzec może być reprezentowany w postaci tabelarycznej lub graficznej.

Przykłady wzorców architektury

Systemy adaptacyjne:

- microkernel
- reflection

Systemy rozproszone – broker

Systemy interaktywne:

- model-view-controller
- presentation-abstraction-control

Systemy strukturalne:

- blackboard
- pipes and filters
- layers
- hexagonals

Microkernel

Mikrojądro służy również jako gniazdo dla rozszerzeń wraz mechanizmem umożliwiającym ich koordynację.

Wzorzec dla systemów, które muszą być zdolne do adaptacji w zmieniających się wymaganiach systemowych.

Separuje minimalną podstawową funkcjonalność od rozszerzeń specyficznych dla określonego systemu.

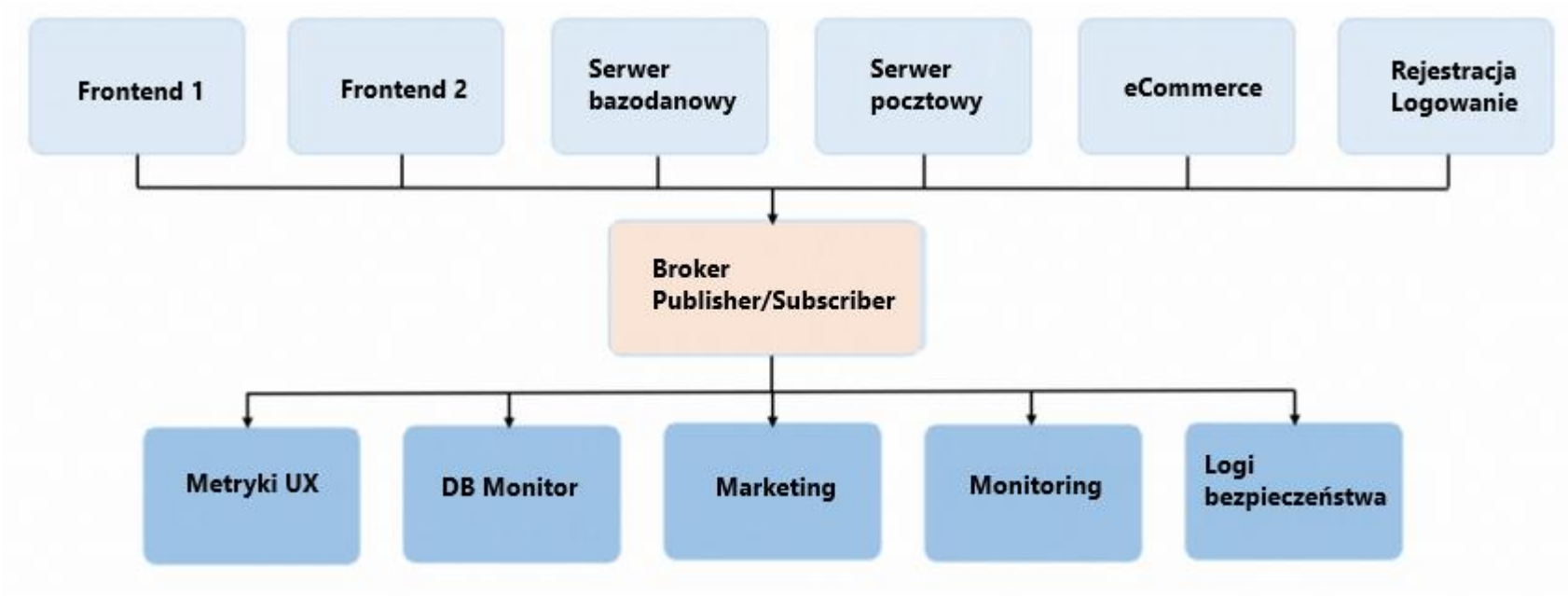
Reflection

Wzorzec architektoniczny Reflection zapewnia mechanizm dynamicznej zmiany struktury i zachowania systemów oprogramowania. Wspiera on modyfikację podstawowych aspektów, takich jak struktury typów i mechanizmy wywoływania funkcji. W tym wzorcu aplikacja jest podzielona na dwie części. Poziom meta dostarcza informacji o wybranych właściwościach systemu i sprawia, że oprogramowanie jest samoświadome. Poziom bazowy zawiera logikę aplikacji. Jego implementacja opiera się na metapoziomie. Zmiany w informacjach przechowywanych na metapoziomie wpływają na późniejsze zachowanie na poziomie podstawowym.

Broker

Nazwa	Architektura typu broker
Opis	Rozwiązanie integrujące usługi kooperujące poprzez systemy kolejkowe. W miejsce bezpośrednich integracji miedzy usługami, tworzących pajęczynę, wprowadza się pośrednika. Technologia ta umożliwia asynchroniczną współpracę rozproszonych usług, z których część jest producentami danych, a część konsumentami danych.
Przykład	Implementacje brokera kolejki w postaci Apache Kafka, RabbitQM (ogólnie wykorzystanie Publish/Subscribe).
Stosowalność	Jest konieczna tam, gdzie istnieje zastosowania koncepcji mikrousług, zastosowanie w koncepcji IoT (np. odczyty pomiarów z sieci sensorycznych).
Zalety	Asynchroniczna komunikacja między (mikro)usługami, możliwość przesuwania ciężkich zadań w tło, odciążenie producenta i konsumenta w procesie komunikacji, sprowadzonej do jednorazowego kontaktu z kolejką w koniecznym momencie.
Wady	Możliwość utraty informacji przy awarii brokera. Nie ma możliwości uruchomienia transakcji biznesowej.

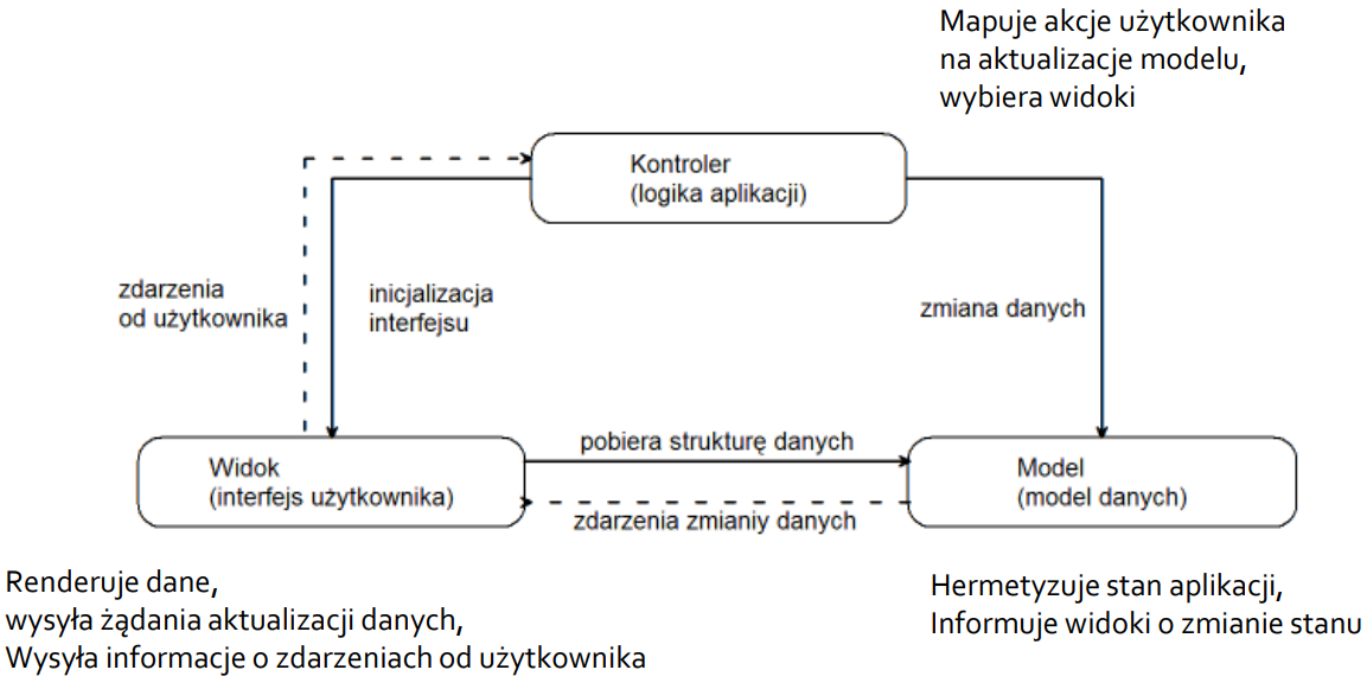
Object Request Broker



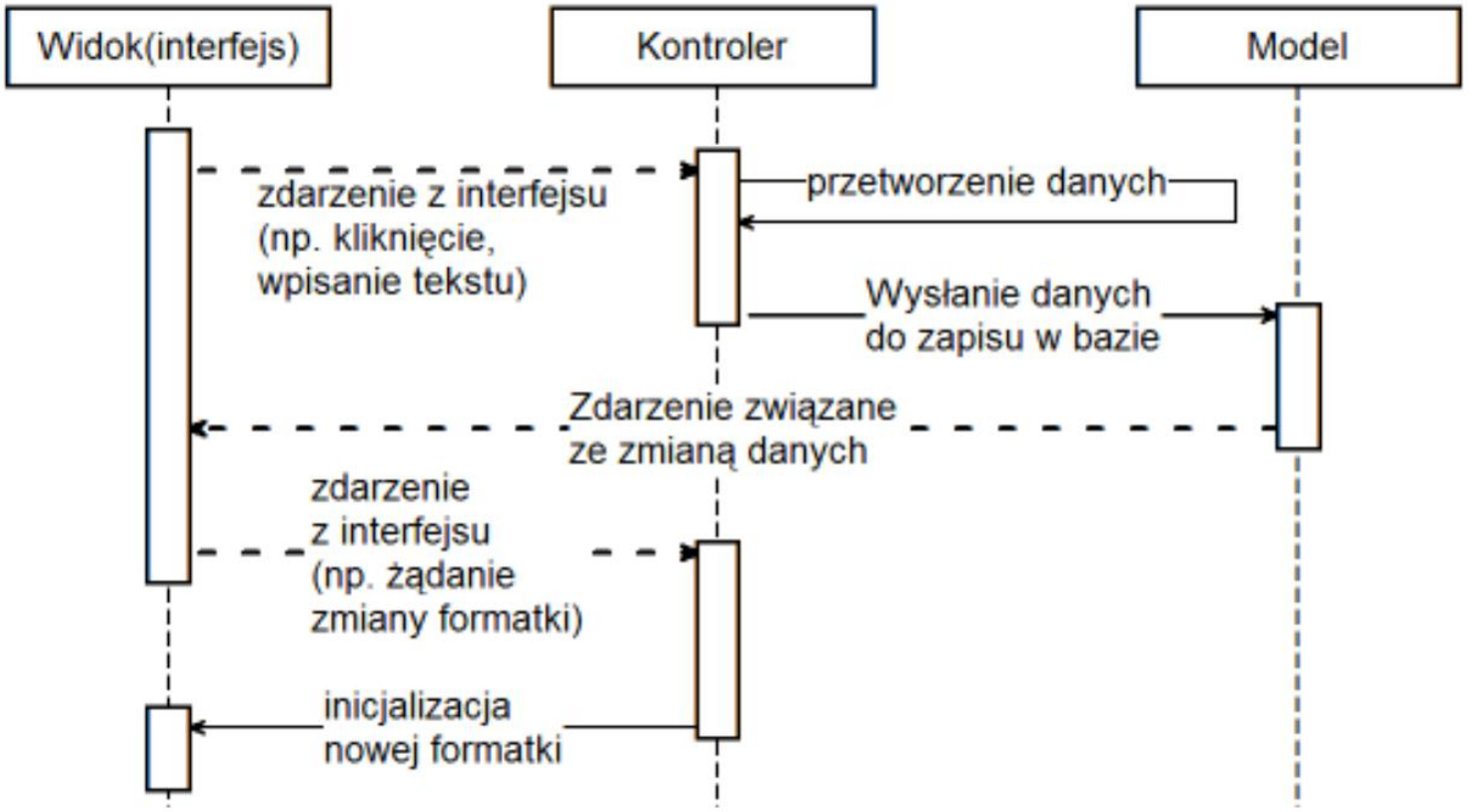
MVC

Nazwa	Architektura typu MVC
Opis	Oddziela prezentację oraz interakcję od danych systemu. System podzielony jest na trzy, oddziaływujące na siebie, logiczne komponenty: Model – to komponent zarządzający danymi oraz operacjami na nich wykonywanymi. Widok – to komponent definiujący oraz zarządzający sposobem prezentacji danych użytkownikom. Kontroler – to komponent zarządzający interakcją użytkownika z aplikacją (np. kliknięcie myszą, naciśnięcie przycisku, itp.), który przekazuje interakcję tak do Widoku, jak i Modelu.
Przykład	Następny slajd - architektura aplik. web. zorganizowanej z wykorzystaniem wzorca MVC.
Stosowalność	Wiele widoków oraz mechanizmów interakcji powiązanych z danymi. Możliwe zmiany w wymaganiach dotyczących sposobu interakcji z danymi oraz metod ich prezentacji.
Zalety	Pozwala na niezależne zmiany w obrębie danych systemu oraz w obrębie ich reprezentacji. Ułatwia prezentowanie tych samych danych na różne sposoby oraz pozwala na zmianę wszystkich widoków w odpowiedzi na modyfikację danych (np. wykonaną za pośrednictwem jednego z widoków).
Wady	W przypadku gdy dane oraz model interakcji użytkownika z aplikacją jest prosty, może zwiększać się złożoność kodu przy ewolucji wymagań biznesowych.

MVC w aplikacji webowej

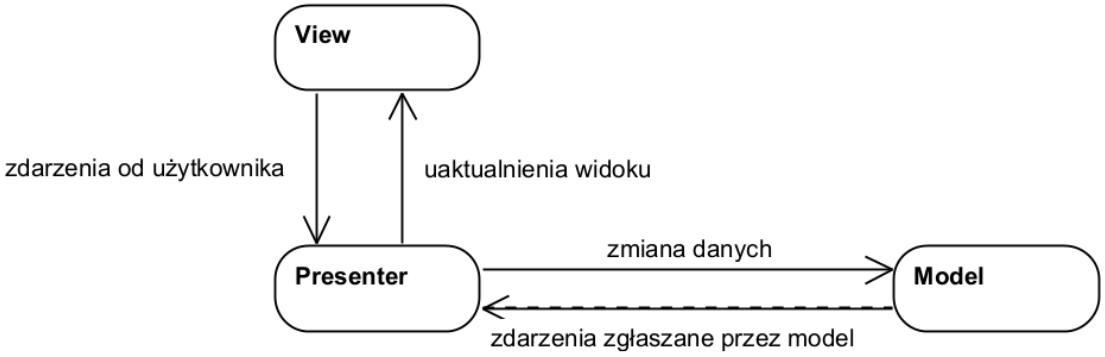


MVC

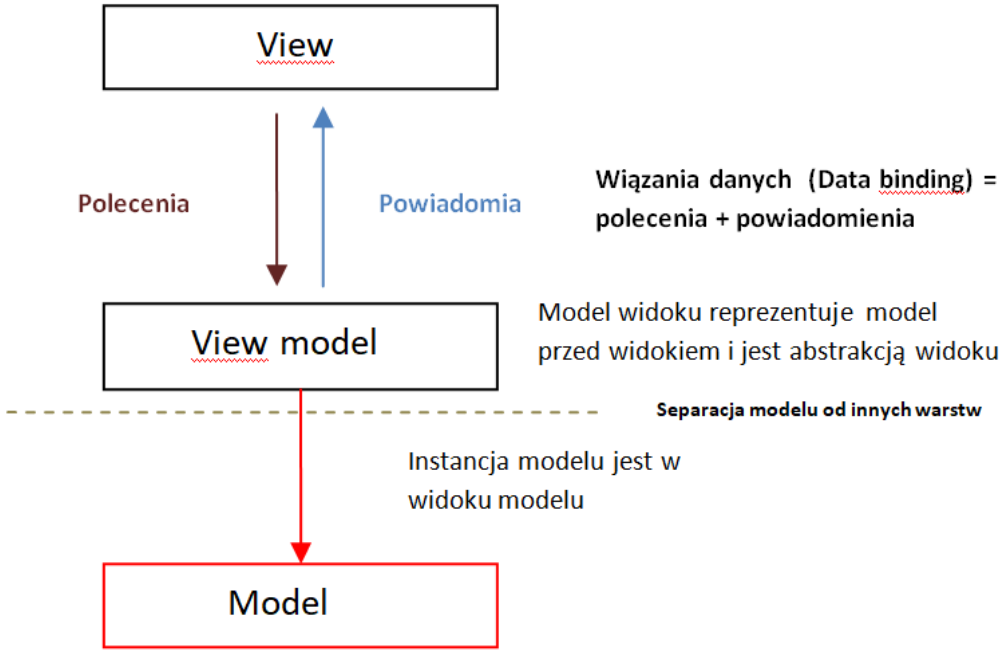

Widok(interfejs)
Kontroler
Model
zdarzenie z interfejsu
(np. kliknięcie,
wpisanie tekstu)
przetworzenie danych
Wysłanie danych
do zapisu w bazie
Zdarzenie związane
ze zmianą danych
zdarzenie
z interfejsu
(np. żądanie
zmiany formatki)
inicjalizacja
nowej formatki

MVP vs MVVM

MVP



MVVM

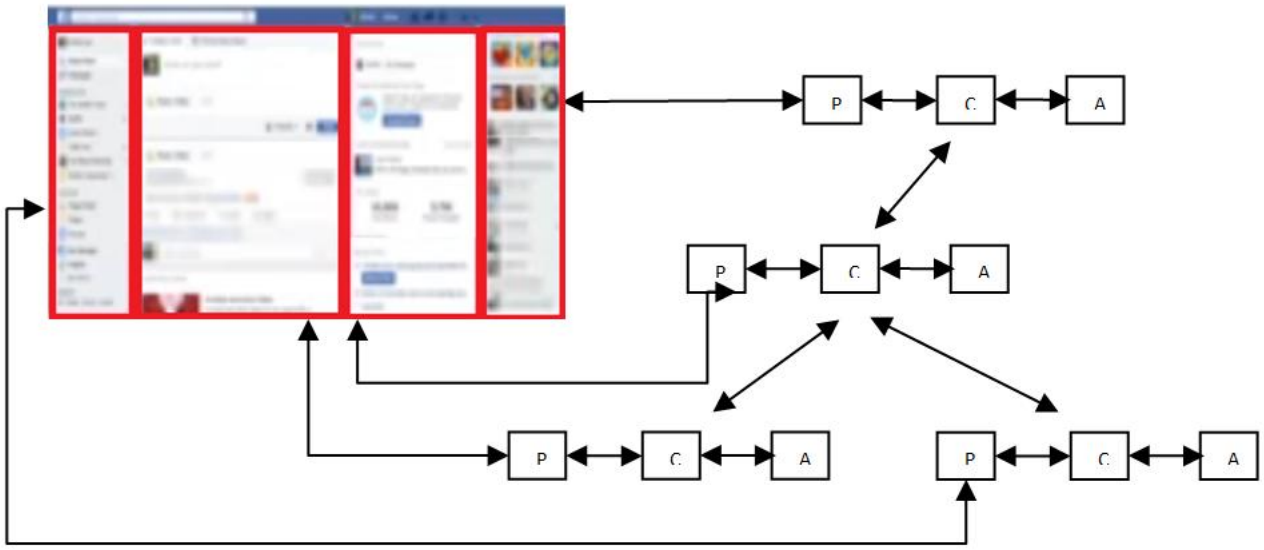


Presentation-Abstraction-Control

System przedstawiony w postaci struktury interaktywnych systemów (agentów). Każdy z agentów jest odpowiedzialny za część funkcjonalności i składa się z trzech warstw: prezentacji, abstrakcji i kontroli.

P – View C – Controller A - Model

Przykład PAC



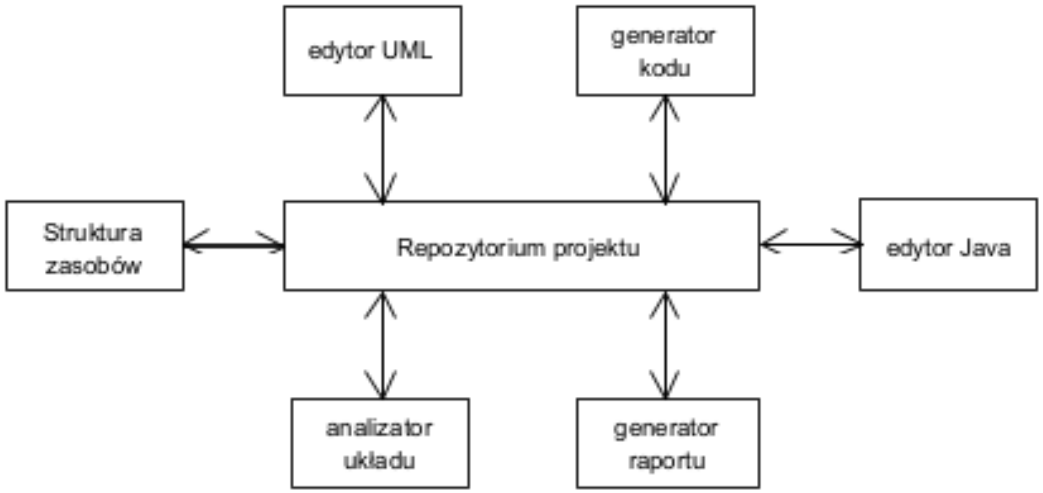
Blackbord

Nazwa	Architektura blackboard
Opis	Grupa wyspecjalizowanych podsystemów (agentów) operujących na wspólnej strukturze danych. Zarządzanie wszystkimi danymi systemu odbywa się za pośrednictwem centralnego repozytorium. Repozytorium dostępne jest dla wszystkich komponentów systemu. Komponenty współdziałają ze sobą tylko za pośrednictwem repozytorium.
Przykład	Następny slajd prezentuje przykład IDE, którego komponenty wykorzystują repozytorium informacji o projekcie. Każde narzędzie wchodzące w skład IDE generuje informacje, które mogą być wykorzystane przez pozostałe.
Stosowalność	System, w którym generowane są duże ilości danych, które muszą być przechowywane przez długi czas. Systemy sterowane danymi, w których wprowadzenie danych wyzwala akcje (np. uruchamia narzędzie).
Zalety	Komponenty systemu mogą być niezależne – nie musza wiedzieć o swoim istnieniu. Zmiany wprowadzone przez jeden z komponentów mogą być propagowane do innych. Dane mogą być zarządzane w spójny sposób (np. kopie zapasowe wykonywane w tym samym czasie).
Wady	Repozytorium jest pojedynczym punktem awarii (single point of failure) – problem z repozytorium wpływa na cały system. Organizowanie całej komunikacji za pośrednictwem repozytorium może być nieefektywne. Rozproszenie repozytorium na kilka komputerów może być trudne.

Blackbord

Nazwa	Architektura blackboard
Opis	Grupa wyspecjalizowanych podsystemów (agentów) operujących na wspólnej strukturze danych. Zarządzanie wszystkimi danymi systemu odbywa się za pośrednictwem centralnego repozytorium. Repozytorium dostępne jest dla wszystkich komponentów systemu. Komponenty współdziałają ze sobą tylko za pośrednictwem repozytorium.
Przykład	Następny slajd prezentuje przykład IDE, którego komponenty wykorzystują repozytorium informacji o projekcie. Każde narzędzie wchodzące w skład IDE generuje informacje, które mogą być wykorzystane przez pozostałe.
Stosowalność	System, w którym generowane są duże ilości danych, które muszą być przechowywane przez długi czas. Systemy sterowane danymi, w których wprowadzenie danych wyzwala akcje (np. uruchamia narzędzie).
Zalety	Komponenty systemu mogą być niezależne – nie musza wiedzieć o swoim istnieniu. Zmiany wprowadzone przez jeden z komponentów mogą być propagowane do innych. Dane mogą być zarządzane w spójny sposób (np. kopie zapasowe wykonywane w tym samym czasie).
Wady	Repozytorium jest pojedynczym punktem awarii (single point of failure) – problem z repozytorium wpływa na cały system. Organizowanie całej komunikacji za pośrednictwem repozytorium może być nieefektywne. Rozproszenie repozytorium na kilka komputerów może być trudne.

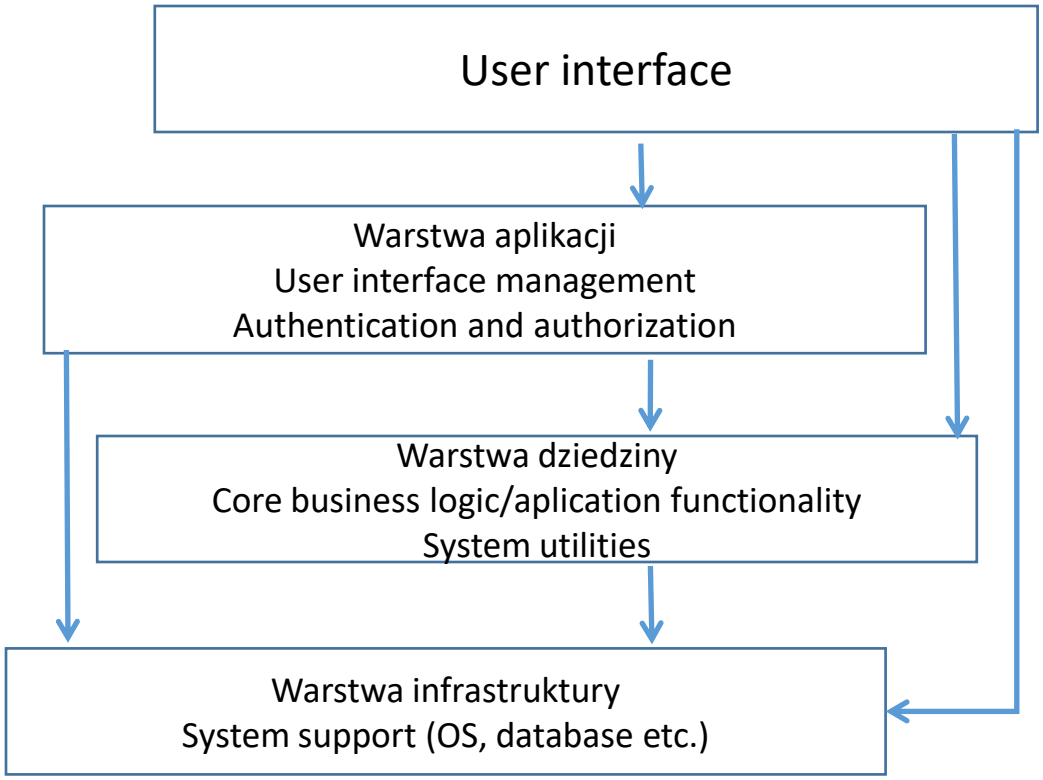
Architektura repozytorium dla IDE



Opis tabelarczyny architektury warstwowej

Nazwa	Architektura warstwowa
Opis	Organizuje system w warstwy. Powiązana funkcjonalność znajduje się w określonej warstwie. Warstwa udostępnia usługi warstwie znajdującej się bezpośrednio nad nią. Warstwa położona najniżej reprezentuje podstawowe usługi, najczęściej wykorzystywane w systemie. Istnieją wersje ścisłej (strict - sprzężenie tylko z warstwą poniżej) i rozluźnionej (relaxed - sprzężenie z dowolną warstwą poniżej, co jest w większości ze względu na konieczność korzystania z infrastruktury).
Przykład	Warstwowy model systemu umożliwiający współdzielenie dokumentów chronionych prawami autorskimi, przechowywanymi w różnych bibliotekach.
Stosowalność	Budowanie nowych elementów na „czubku” istniejących systemów. Rozwój jest podzielony pomiędzy odrębne zespoły i każdy z zespołów odpowiedzialny jest za określoną warstwę. Istnieje wymaganie wielopoziomowych zabezpieczeń.
Zalety	Pozwala na podmianę całej warstwy tak długo, jak interfejs pozostaje bez zmian. Nadmiarowe udogodnienia mogą być wprowadzane do każdej warstwy (np. uwierzytelnianie) w celu zwiększenia pewności systemu.
Wady	W praktyce pełna realizacja założeń wzorca jest trudna i może prowadzić do problemów z wydajnością (potrzeba przetwarzania żądania usługi przez każdą warstwę).

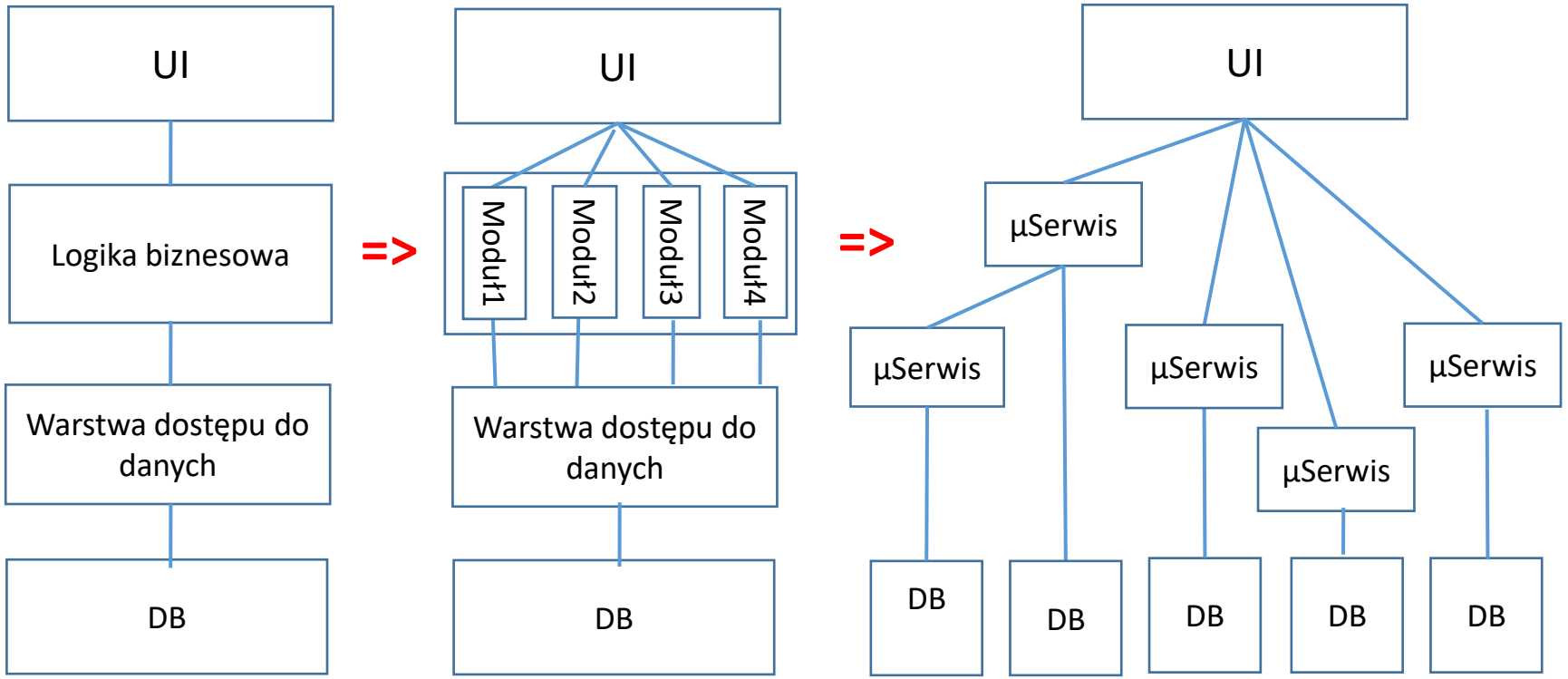
Graficzna prezentacja tradycyjnej architektury warstwowej



Architektura warstwowa – wykorzystanie Mediatora

wyższa warstwa implementuje w obiekcie interfejs Mediatora zdefiniowany przez niższą warstwę, a następnie przekazuje ten obiekt jako argument do warstwy niższej, która korzysta z niego bez wiedzy o tym, skąd on pochodzi.

Monolit => Modularny Monolit => Mikroserwisy



Korzyści stosowania architektury mikroserwisów

1. Łatwe śledzenie kodu jako odrębnego bytu realizujacego konkretną funkcjonalność (przypadek użycia).
2. Łatwość we wdrażaniu aktualizacji takiego bytu.
3. Łatwość ponownego użycia komponentów.
4. Łatwa izolacja wadliwych serwisów i szybsza reakcja na odkryte błędy
5. Łatwiejsze testowanie pojedyńczych serwisów.
6. Możliwość stosowania różnych technologii w poszczególnych serwisach.

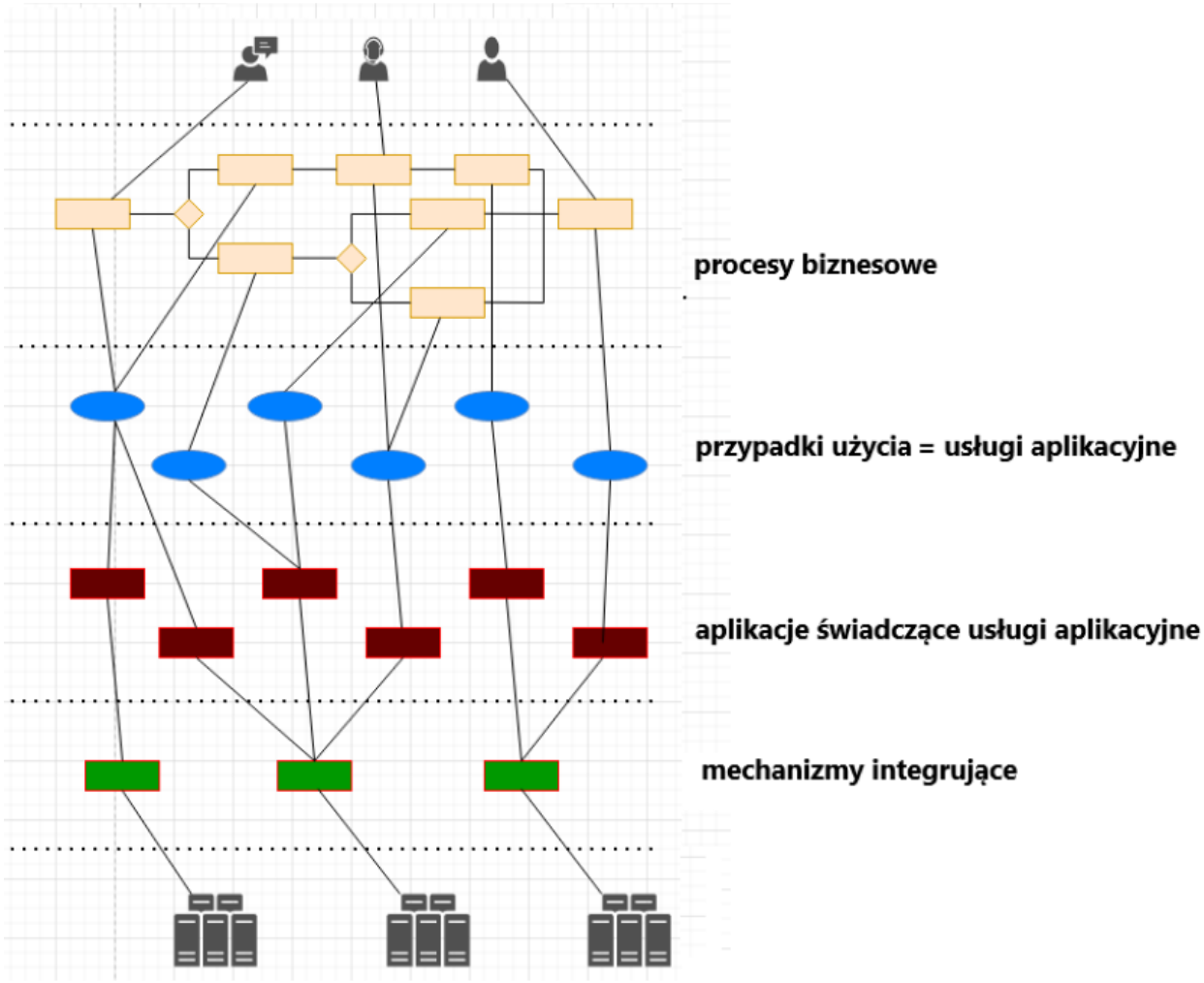
Inne kwestie związane z mikrousługami

Rozproszone transakcje - wykorzystują technikę rozproszonych transakcji znaną jako transakcję BASE (od ang. basic availability, soft state, eventual consistency, czyli podstawowa dostępność, miękki stan, ostateczna spójność), którego polegają na ostatecznej spójności i dlatego nie obsługują tego samego poziomu integralności bazy danych co transakcje ACID.

Rozproszone dzienniki zdarzeń.

Mikrousługi można uznać za pochodną stylu architektonicznego DDD (Domain Driven Design).

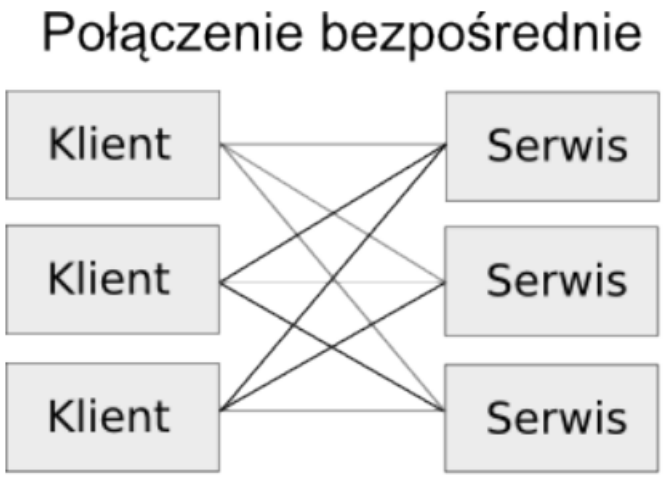
Service Oriented Architecture SOA vs mikroserwisy

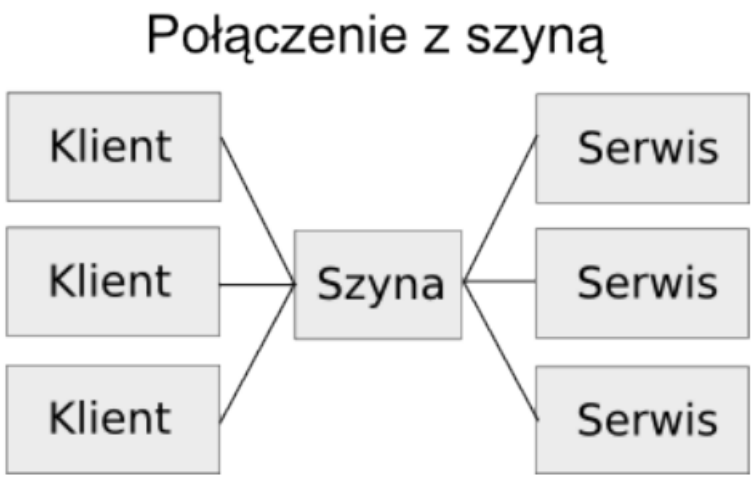


Elmenty projektowania usług

1. Kontrakt usługi – wyraża cel i możliwości usługi.
2. Słabe (luźne) sprzężenia – możliwie minimalne zależności miedzy usługami.
3. Abstrakcja – ukrywanie przed klientami logiki wewnętrznej przez publikację jedynie kontraktu usług.
4. Reusing – wielokrotne użycie usług przez usługi złożone.
5. Komponowalność – wkomponowanie w obszar usług (patrz reusing) bez względu na bieżącą ich złożoność
6. Niezależność – usługi zachowują swoją autonomię (środowiska, zasoby).
7. Bezstanowość – usługa pozwala zarządzać swym stanem aktorom.
8. Wykrywalność – metadane pozwalają odkryć i rozumieć kontrakt.

Mechanizmy integrujące serwisy w SOA



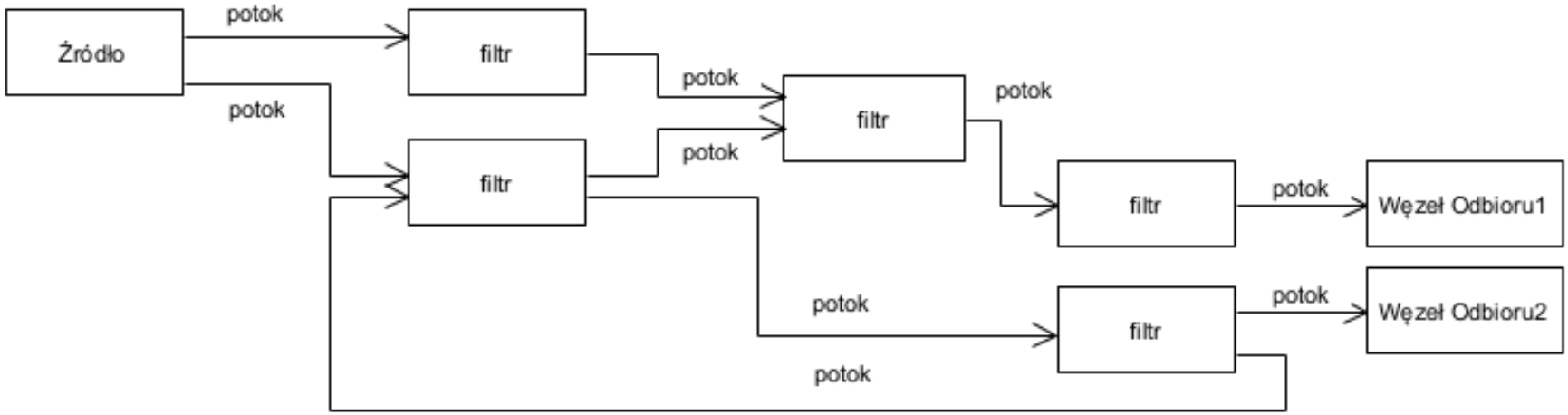


Szyna ESB – złożona komunikacja

Oprócz zwykłego łączenia usług, posiada jeszcze funkcje:

1. Mapowanie żądań usług z konkretnego protokołu i adresu na inny protokół/adres.
2. Konwersja danych na inny format.
3. Umiejętność pracy z wieloma modelami transakcji i bezpieczeństwa oraz łączenie różnych modeli integrowanych serwisów.
4. Integracja żądań do serwisów w zbiór pul żądań.
5. Zachowanie jakości usług (QoS) przy obsłudze protokołów sieciowych między różnymi platformami.

Filtry i potoki



Przypadek wiersza poleceń: $ cat text.txt | grep potok | wc –l

jest uproszczoną wersją dla jednego potoku i jednego filtra.

Równoległe korzystanie z filtrów zwiększa przepustowość architektury.

Filtry i potoki

Nazwa	Architektura filtrów i potoków
Opis	Filtr jako komponent przetwarzania wykonuje jeden typ transformacji danych w strumień wyjściowy. Potoki łączą filtry. Dane przepływają od filtru do filtru w celu dalszego przetworzenia. Transformacje mogą być wykonywane współbieżnie. Powszechnie stosowane w systemach przetwarzania wsadowego i wbudowanych systemach sterowania.
Przykład	Filtry i potoki mogą być wykorzystane w przetwarzaniu dokumentów. Także API jest przykładem takiej architektury.
Stosowalność	Powszechne zastosowanie w aplikacjach przetwarzania danych (zarówno wsadowych, jak i transakcyjnych), gdzie dane wejściowe są przetwarzane w oddzielnych etapach w celu wygenerowania powiązanych danych wyjściowych.
Zalety	Łatwe do zrozumienia i wspierające ponowne wykorzystanie transformacji. Przepływ pasuje do struktury wielu procesów biznesowych. Ewolucja poprzez dodawanie transformacji jest prosta. Może być zaimplementowany jako system sekwencyjny lub współbieżny.
Wady	Format przesyłania danych musi zostać uzgodniony między stronami. To zwiększa obciążenie systemu i może oznaczać, że niemożliwe jest ponowne użycie funkcjonalnych transformacji, które używają niekompatybilnych struktur danych. Trudne do zastosowania w systemach interaktywnych.

Filtry i potoki (c.d.)

Filtry są procesorami (komponentami). Każdy taki procesor-komponent jest niezależny od pozostałych (tzw. luźne sprzężenia).

Zgodnie z wymaganiami przypadków użycia można modyfikować kolejność w jakiej procesory-kompenty odbierają komunikaty.

Potoki są kanałami komunikatów i służą odbieraniu (potok wejściowy) oraz wysyłaniu (potok wyjściowy) komunikatów.

Można wyróżnić 4 rodzaje filtrów:

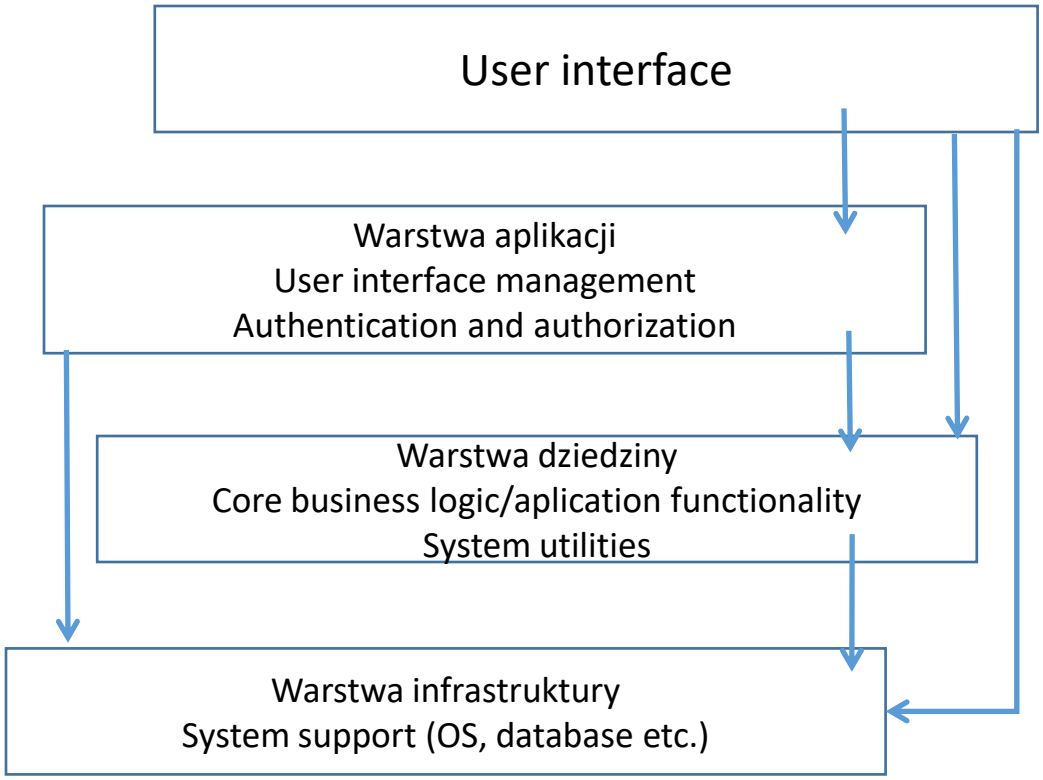
- wytwórczy (*producer*)
- przekształcający (*transformer*)
- testujący (*tester*)
- odbiorczy (*consumer*)

Wykład 6

Architektura oprogramowania (c.d.):

- architektura heksagonalna
- styl architektoniczny DDD

Graficzna prezentacja tradycyjnej architektury warstwowej



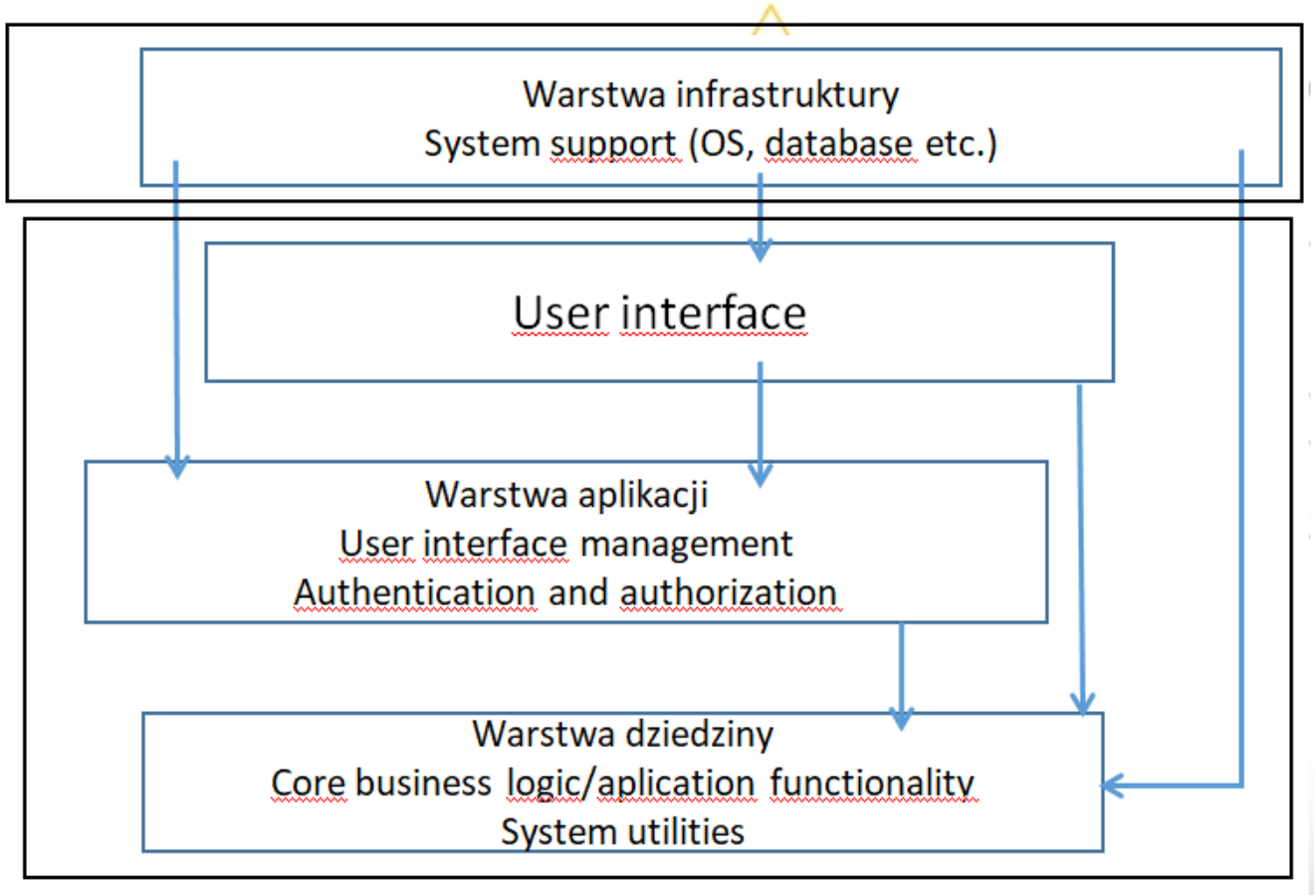
Zasada odwracania zależności
Dependency inversion principle

"High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces)." (źródło – wikipedia)

"Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions." (źródło – wikipedia)

Po zastosowaniu Dependency Inversion Principle tak wysokopoziomowe, jak i niskopoziomowe komponenty warstw zależą wyłącznie od abstrakcji (interfejsów), co prowadzi do spostrzeżenia o odwróceniu stosu.

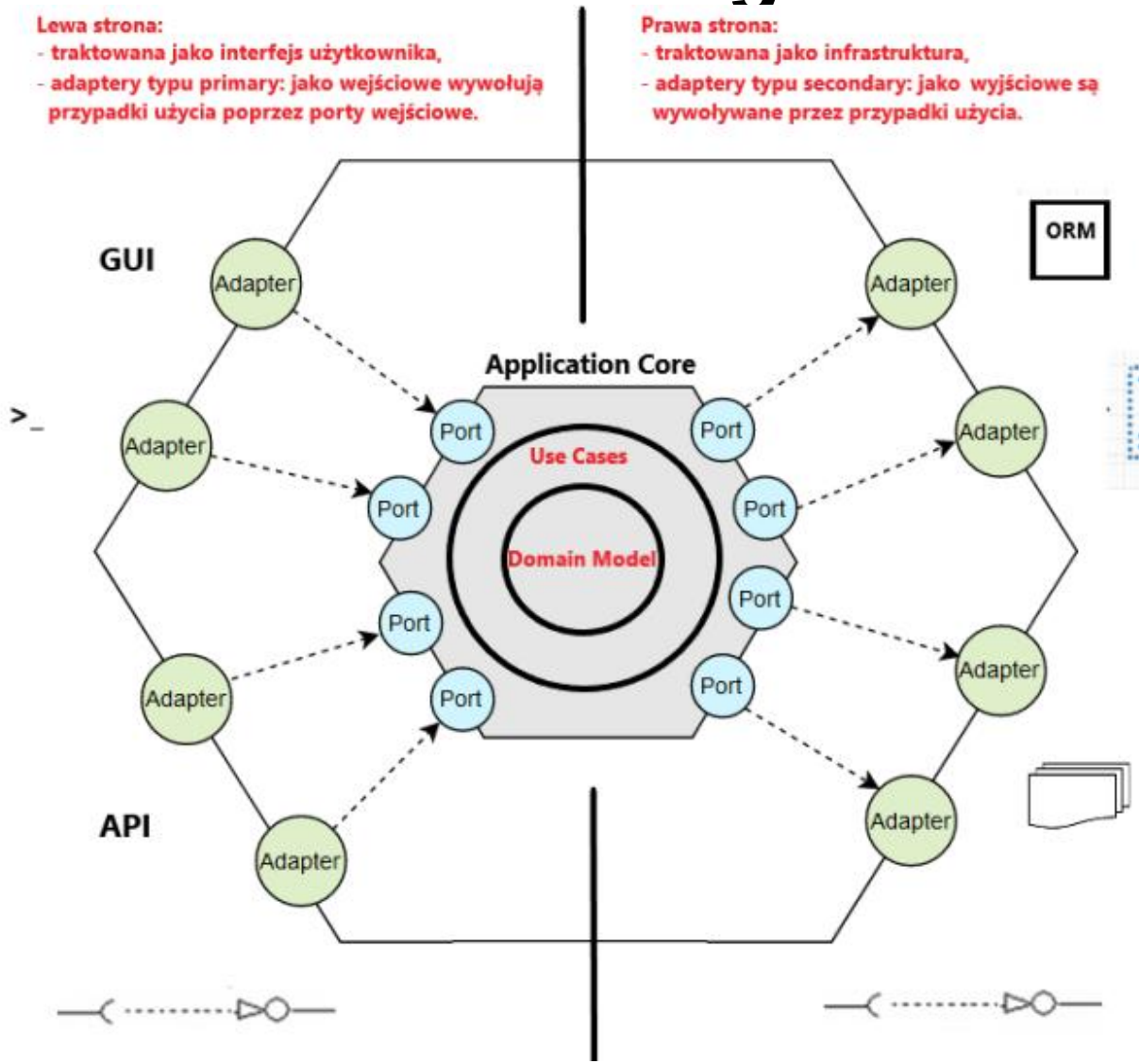
Postać architektury zależna od DIP – dwie quasi superwarstwy



Zasada odwracania zależności Dependency Inversion Principle (c.d.)

- Komponent infrastruktura, który dostarcza niskopoziomowych usług, zależy dzięki zastosowaniu DIP od interfejsów zdefiniowanych przez komponenty: UI, warstwa aplikacji, warstwa dziedziny. Infrastruktura implementuje interfejsy z tych warstw.
- Warstwa aplikacji bezpośrednio korzysta z dziedziny i zależy od interfejsów warstwy dziedziny i może korzystać pośrednio z klas dostarczanych przez infrastrukturę, które implementują usługi dziedziny.

Architektura heksagonalna



Architektura heksagonalna

Architektura heksagonalna pozwala na separację logiki biznesowej (tego co mamy zrobić) od szczegółów implementacji technicznych (w jaki sposób to robimy). Pozwala to na wygodne wprowadzanie zmian dotyczących aspektów technicznych, które są odseparowane od logiki biznesowej aplikacji. Dodatkowo daje to możliwość łatwiejszego testowania logiki biznesowej, która sama w sobie nie jest zależna od szczegółów technicznych.

Domain model

Model (dane) + złożoność biznesowa (w tym klasy usługowe) = model domenowy.

Model domenowy to mechanizm opisujący dane biznesowe i metody ich przetwarzania.

Model domenowy nie powinien być anemiczny (anemiczny antywzorzec) – anemiczny powstaje wskutek projektowania aplikacji w oparciu o stworzony projekt relacyjnej bazy danych, gdy mapuje się tabele na klasy bez zawartych w nich operacji (przypomina to koncepcyjny model danych).

Używanie anemicznego modelu może wynikać ze stosowania konkretnych narzędzi w procesie implementacji.

Anemic Domain Model vs Rich Domain Model

Anemiczny model danych to zwykły obiekt DTO (Data Transfer Object) czyli klasa z prywatymi polami, geterami i seterami.

W anemicznym modelu domenowym logika biznesowa jest wyniesiona do zewnętrznych klas a sam anemiczny model domenowy jest rozhermetyzowany.

W bogatym modelu danych obiekt bogatej domeny udostępnia zachowania modelu, ograniczając możliwość modyfikacji wewnętrznych danych modelu. Logika znajduje się wewnątrz modelu.

Inna nazwa: Adaptery i Porty

Port jest pojęciem elastycznym – do portów dochodzą żądania klientów, a właściwy adapter zajmuje się transformacją ich danych wejściowych. Adapter związany jest z typem klienta.

Port pełni rolę mechanizmu wymiany komunikatów, a adapter jest ich słuchaczem. Adapter przekształca protokół wejściowy na dane wejściowe zgodne z inferfejsem API aplikacji (klientem modelu dziedziny).

Granica przypadku użycia dąży do granicy wewnętrznego sześciokąta.

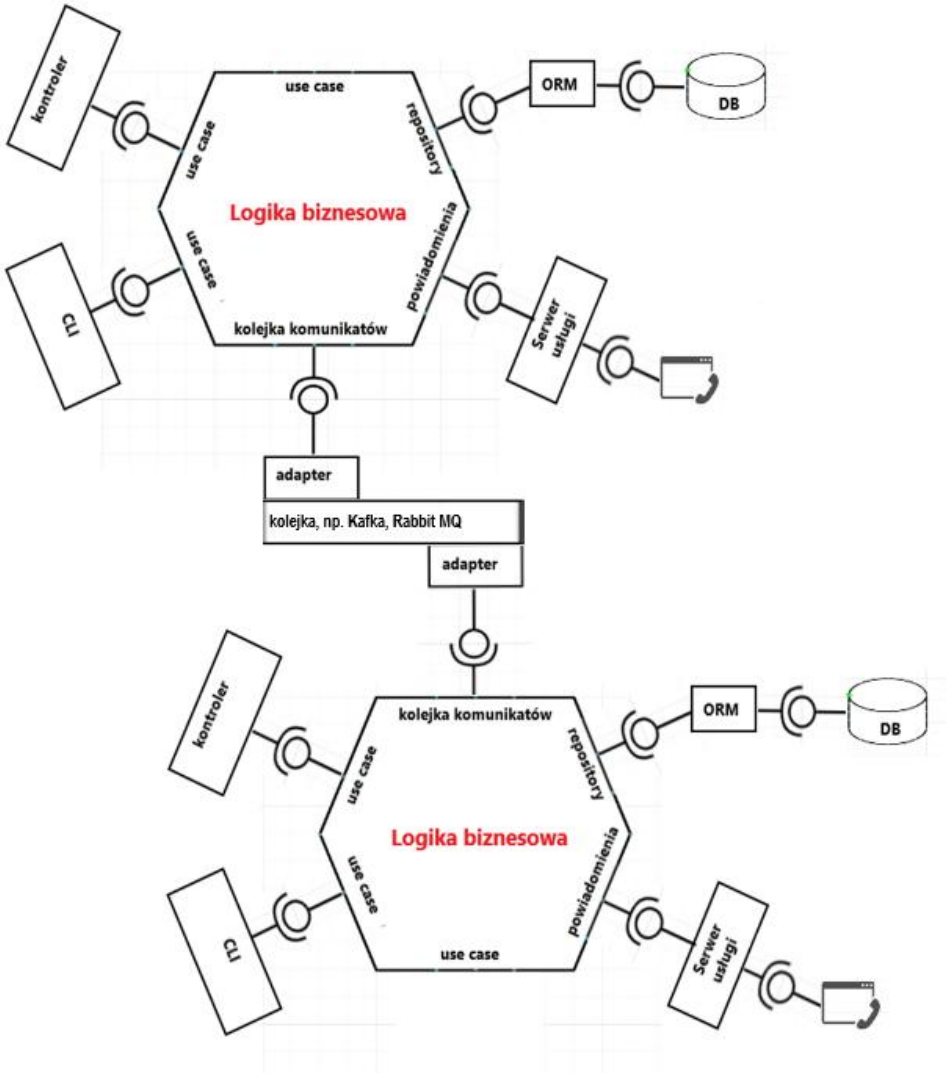
Adaptery i Porty

Primary adapter – np. kontroler pobierający dane wejściowe i przekazujący je do app poprzez port.

Serwis w Application Core implementuje interfejs definiowany przez port – interfejs portu i jego implementacja są wewnątrz haksagonu.

Secondary adapter – np. adapter bazy danych implementuje port, który jest używany przez Application Core – w tym przypadku port (interfejs) znajduje się wewnątrz heksagonu a jego adapter na zewnątrz.

Inny przykład hexagonu



Dobre praktyki implementacji architektury heksagonalnej

1. Modułowość komponentów - pozwala zamieniać implementacje bez zmian logiki bizensowej.
2. Separacja logiki biznesowej od infrastruktury dla wydajnego testowania i przyszłego rozwoju aplikacji.
3. Interfejsy dla portów mają mieć precyzyjnie definiowane kontrakty.
4. DIP do zarzadzania zależnościami między komponentami, co prowadzi do luzowania zależności i upraszcza testowanie.

Styl architektoniczny DDD

- Domain Driven Design czyli projektowanie zorientowane na dziedzinę systemu (zrozumienie, język, reprezentacja).
- Jest narzutem na sposób wytwarzania oprogramowania.
- Dobrze modeluje rzeczywistość – nadaje się w przyszłości do "liniowej" rozbudowy i konserwacji a nie "skokowej" zmiany na skutek zmian w logice biznesowej.
- Stosuje strategiczne i taktyczne podejście do modelowania dziedziny.
- Systemy projektowane zgodnie z duchem DDD mają strukturę warstwową.

DDD – strategia czy taktyka ?

Strategia – mechanizm identyfikacji serwisów poprzez wydzielanie kontekstów (granic), pozwala definiować zasadniczo optymalną komunikację pomiedzy kontekstami.

Taktyka – sposób pracy z kodem z zachowaniem enkapsulacji przy wykorzystaniu szeregu technik (wzorców taktycznych).

Nie każdy kontekst wymaga wzorców taktycznych (np. prosty CRUD o małym stopniu złożoności nie wymaga).

Zastosowanie strategii nie musi pociągać za sobą stosowania wzorców taktycznych.

Wzorce DDD – podejście praktyczne

Strategiczne: Ubiquitous Language, Bounded Context, Context Mapping

Bounded Context=core domain (domeny gdzie warto stosować taktyczne) + supporting subdomain + generic subdomain. Subdomeny to to gdzie nie warto stosować taktyki.

Taktyczne: Aggregate, Entity, Value Object (te trzy są odpowiedzialne za język), Factory, Repository (te dwa ostatnie to cykl życia)

Ubiquitous Language

Komunikacja

Zrozumienie

Wiedza

Odkrywanie

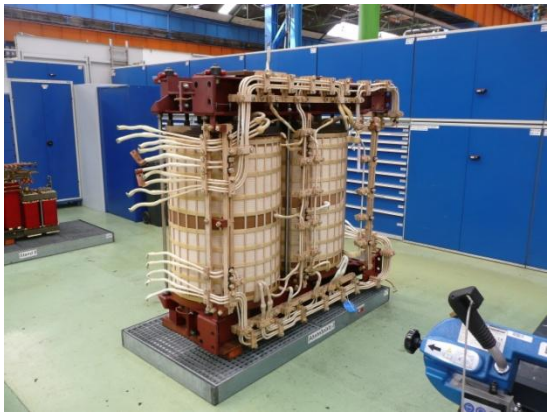
Te w/w pozwalają na jednoznaczne rozumienie wymagań. W trakcie postępów w przyswajaniu wiedzy ma miejsce odkrywanie nowych pojęć oraz ewentualnie doskonalenie wymagań.

Wiele aspektów tej samej rzeczy

Przykład transformatora

kontekst elektryczny – moc, przekładnia, straty, sprawność …

kontekst majątkowy - wartość , wymiary, masa …



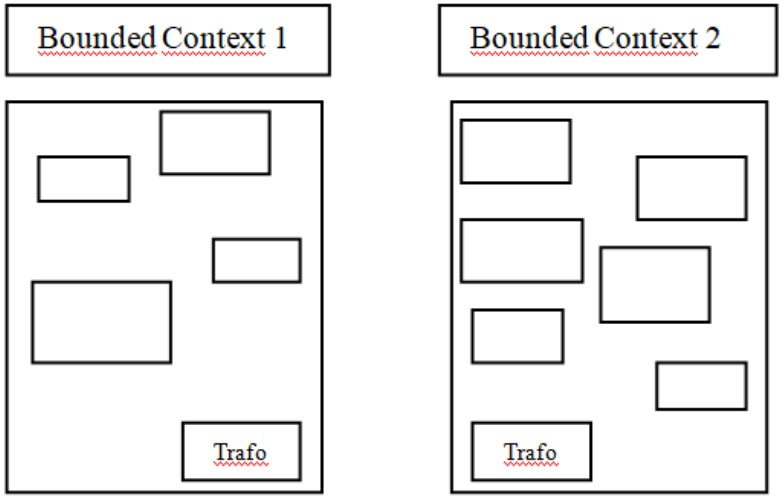
Źródło:

https://commons.wikimedia.org/wiki/Category:Transformers#/media/File:Aktivteil_Transformator_BR401_Trafowerkstatt_Aw_Dessau.jpg

Bounded Context jako strategia stylu architektonicznego DDD

Wyznaczamy obszary Bounded Context. Unikamy uwspólniania pojęć – to kontekst określa znaczenie.

Kontekst elektryczny



Kontekst majątkowy

Wyróżnione obiekty w obu kontekstach posiadają to samo UUID

Pojecia opisujące obiekt w DDD

Tożsamość – sprawia, że obiekt jest odróżnialny od innych obiektów.

Stan – zbiór wartości wszystkich cech (atrybutów) obiektu oraz powiązań między obiektami.

Zachowanie – zbiór wszystkich usług, jakie obiekt potrafi wykonać na rzecz innych obiektów.

Wzorce taktyczne - ValueObject

Value Object (obiekt wartości|) – np. cena, waluta, ilość (liczy się informacja a nie tożsamość – nie posiadają zatem tożsamości). Ta sama klasa Value Object może być atrybutem wielu różnych encji i agregatów (pojęcia DDD).

Uwaga: w jednym kontekście imie, nazwisko, adres mogą być Value Object, ale w innym mogą być encją.

ValueObject (c.d.)

Value Object – raz utworzony nie może być zmieniony, a zmieniony nie jest tym obiektem którym był (jest to cecha immutable).

Value Object można przekazywać jako argumenty metod bez obawy o skutki uboczne (jest to cecha immutable).

W porównaniu do DTO (*Data Transfer Object*), Value Object zawierają nie tylko atrybuty ale również metody (np. do walidacji – to ważne ponieważ w cyklu życia VO nie może zmienić stanu i należy sprawdzić, czy Value Object posiada poprawny stan).

Entity w DDD

Entity w DDD jest pojęciem dla DDD bez skojarzeń z definicjami z baz danych– określa jakiś byt i posiada jego wyróżniki:

Tożsamość (identity) definiuje i sprawia, że odróżnia dany byt od innych i jest trwała podczas całego cyklu życia bytu.

Relacja – byt może posiadać inne byty-obiekty: entities lub value objects. Relacja jest odpowiedzialna za koordynacje zachowania obiektów, które posiada.

Wzorce taktyczne – Encja ENTITY

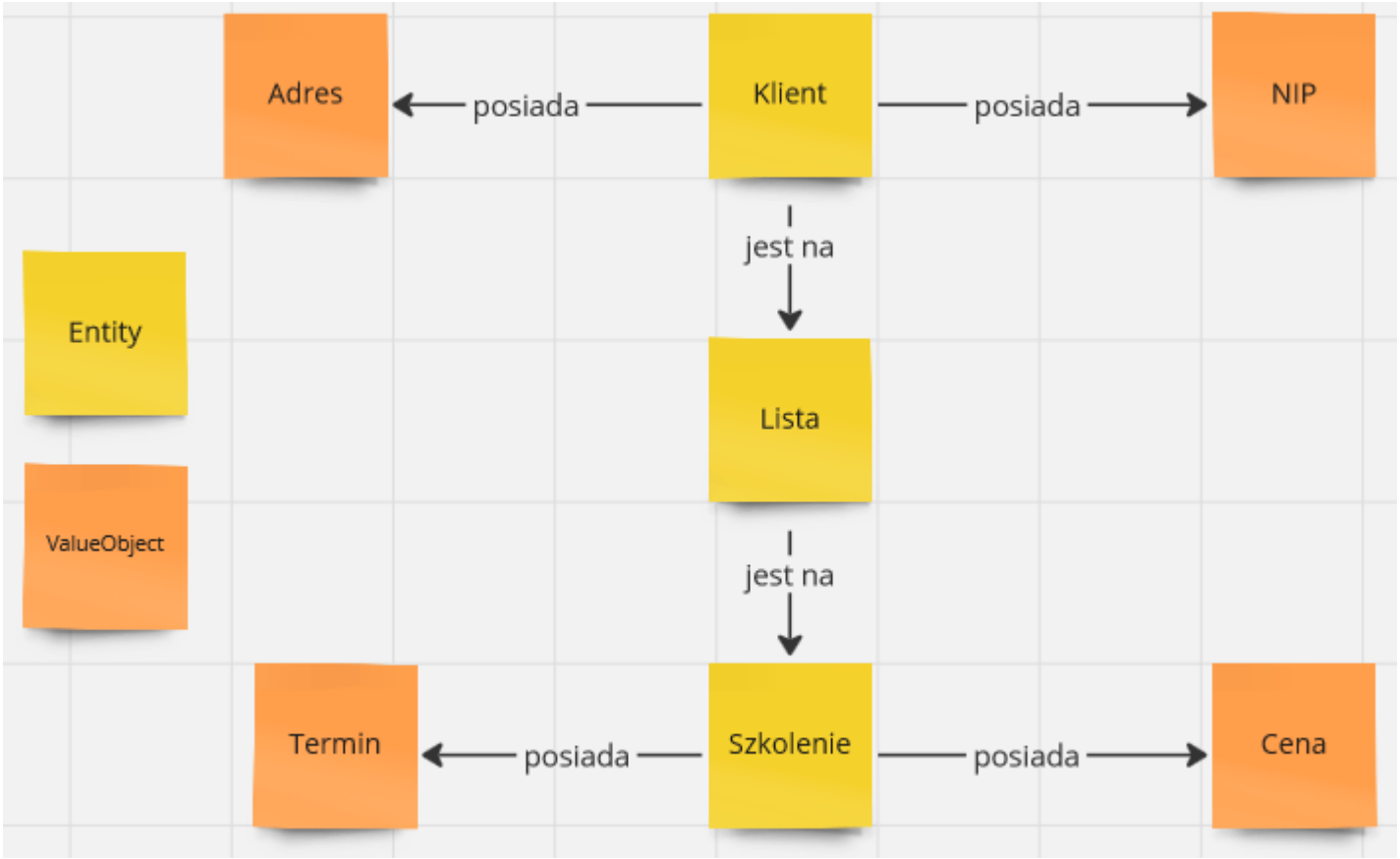
Encja posiada Tożsamość (2 obiekty mogą nie być tymi samymi nawet gdy dane są takie same – musimy znaleźć odróżniające Id jak np. PESEL).

Możliwa jest zmiana stanu, ale nie tozsamości.

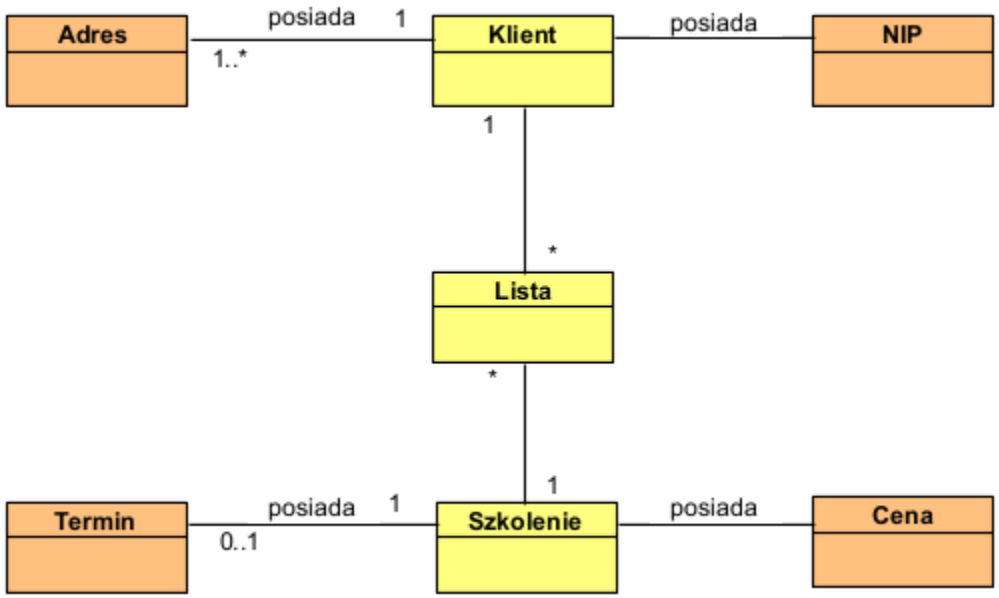
Encje są unikalne wewnątrz agregatu.

UWAGA: gdy wiele agregatów zawiera referencje do tej samej klasy, to oznacza, że te agregaty modyfikują tę samą klasę.

Entity vs ValueObject



Entity vs ValueObject (c.d.)



Entity vs ValueObject (c.d.)

Entity – obiekt klasy Klient może zmienić wartość atrybutów, jednak nadal będzie to ten sam Klient o znanej Tożsamości.

ValueObject – np. obiekt klasy Adres po zmianie wartości dowolnego atrybutu (np. ulica) będzie innym obiektem.

Wzorzec AGREGATE

Agregat – główny element logiki domenowej w DDD; agreguje on dane w hermetycznej strukturze (ma określone granice) o hierarchicznej postaci (np. JSON, XML), manifestując siebie na zewnątrz encją nazwaną korzeniem (Root). Tylko przez Aggregate Root można komunikować się z agregatem.

Korzeniem agregatu jest obiekt, który może występować samodzielnie. Obiekty innych klas posiadające jedynie powiązania z obiektem korzenia składają się na agregat zbudowany wokół tego korzenia.

Uwaga: agregat nie chroni swoich atrybutów, a logika biznesowa może być rozsiana po serwisach.

Agregat (c.d.)

Kandydatem na agregat są klasy powiązane kompozycją.

Agregat ma wpływ na zachowanie integralności danych dlatego, że:

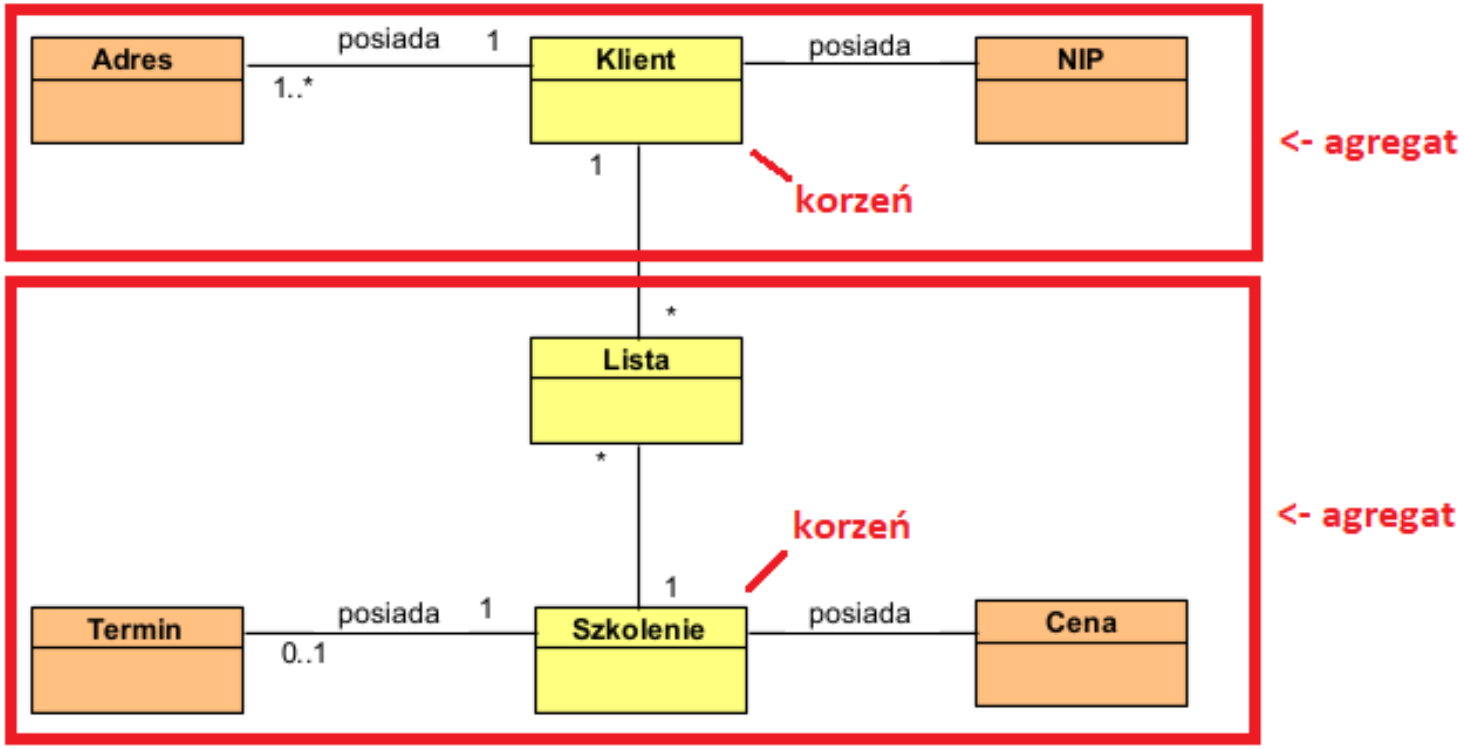
- jest zapisywany, odtwarzany i usuwany jako całość,

- izoluje swoje obiekty i operacje na nich można wykonać jedynie za pośrednictwem korzenia, co daje kontrolę na takim działaniem.

Zbiór agregatów i związków między nimi definiuje system i upraszcza się w ten sposób model systemu.

Agregat może być traktowany jako maszyna stanowa ze względu na statusy modelowanej domeny.

Przykład agregatu

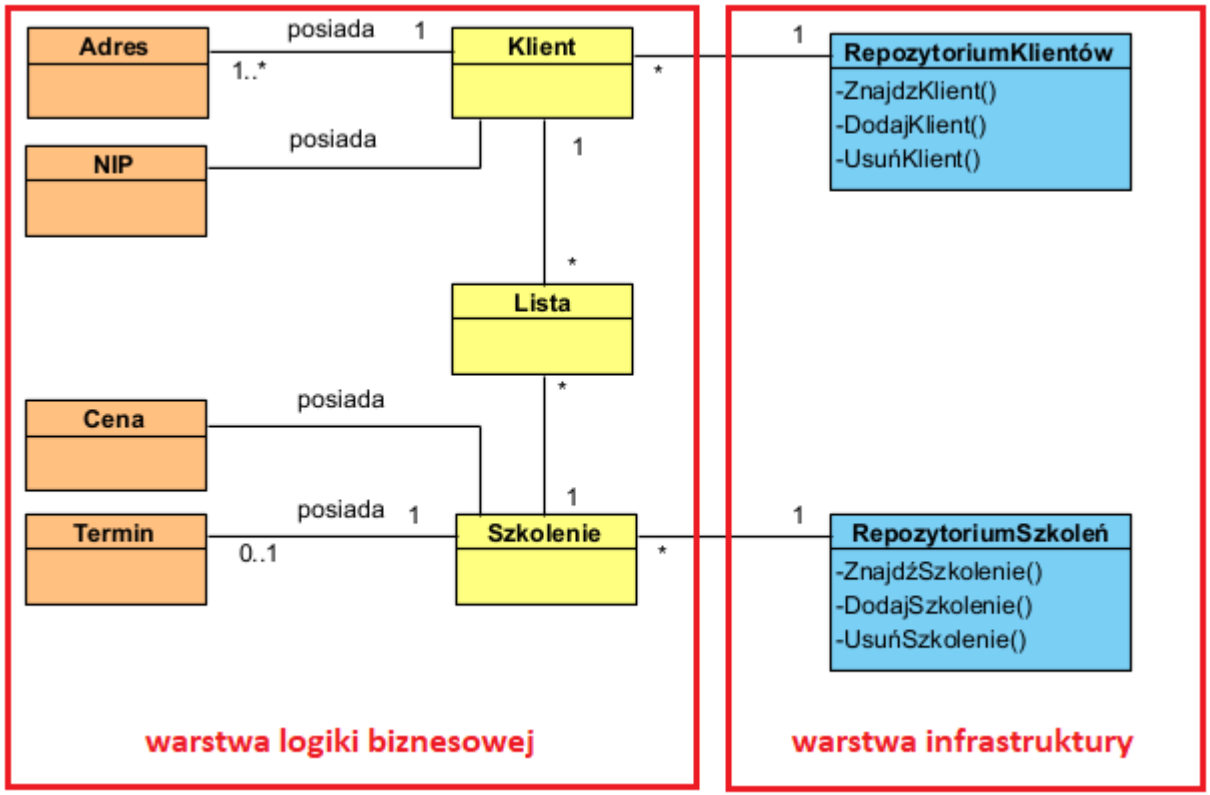


Repozytorium

Wzorzec repozytorium zarządza trwałością agregatu (zapisywany jest cały agregat), ukrywając jednocześnie mechanizmy dostępu do bazy danych.

W DDD repozytorium to niezależny interfejs, który mówi co w danym Bounded Context dzieje się z danym agregatem. Taki interfejs nie może z założenia rozszerzać innych interfejsów (np. interfejsu ORM, poineważ to rozszerzenie wprowdziłoby szereg metod, co do których nie będziemy pewni potrzeby ich istnienia).

Repozytorium



Serwis w DDD

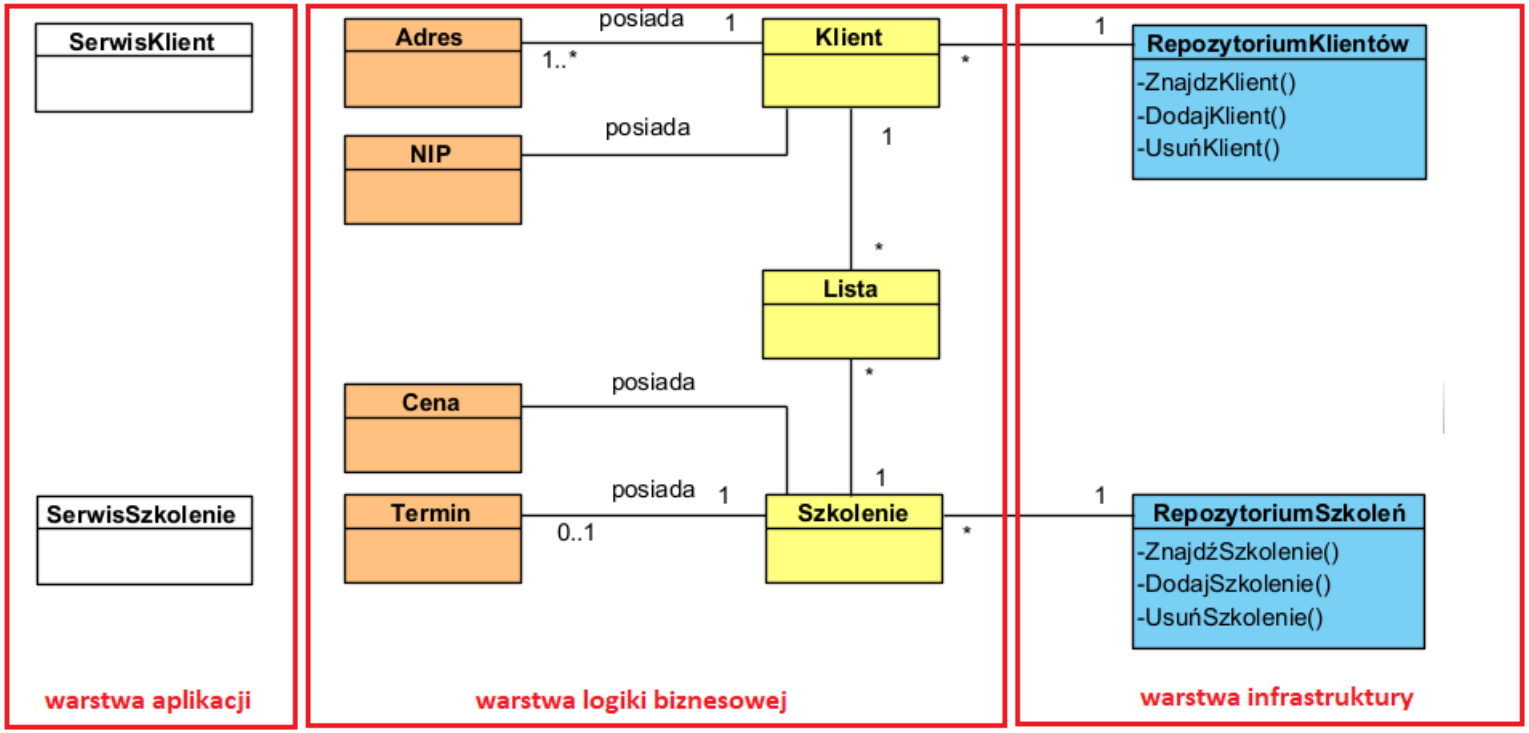
Serwis - wyspecjalizowana klasa, której obiekty służą do realizacji takich operacji biznesowych, jakich nie da się utożsamić z jednym obiektem biznesowym.

Serwis rozpoczyna obsługę operacji systemowych w kaskadzie skoordynowanych działań (orkiestracja), związanych z realizacja jednego przypadku użycia.

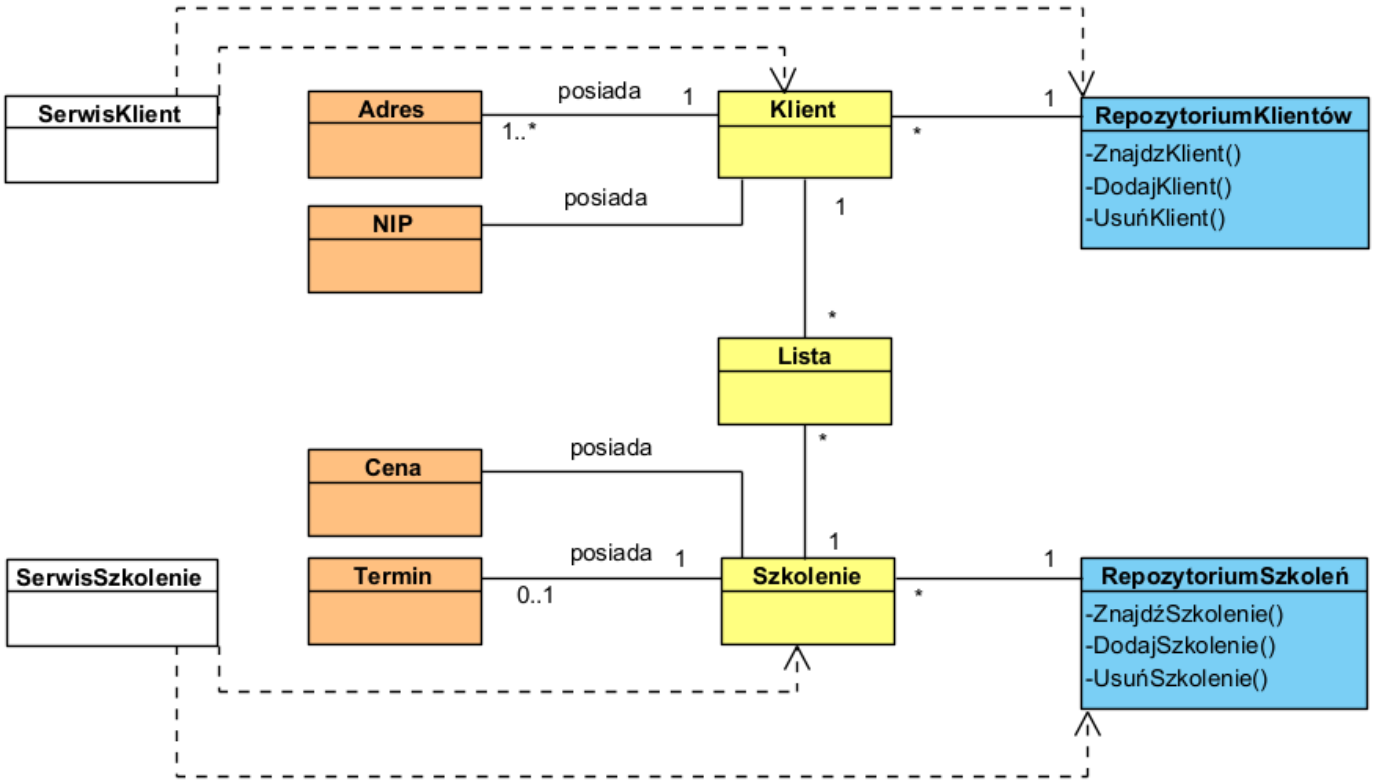
Serwisy występują w warstwie aplikacji (w/w orkiestracja) lub w warstwie logiki biznesowej (np. w transformacji agregatów – np. generacja raportu).

Serwis nie jest trwały (po wykonaniu akcji może być usunięty) i nie jest trwały jego związek z innymi obiektami.

Serwis w warstwie aplikacji



Krótkotrwały związek serwisów z agregatami i repozytoriami



Wzorce taktyczne - Fabryka

Fabryka - obiekt którego, celem jest wytworzenie produktu jaki zamawiamy lub jego odtworzenie. Fabryka jest wykorzystywana do operacji dot. Value Objects, Entities, Agreggates. W przypadku agragatu (encje i obiekty wartości) znacznie upraszcza się tworzenie tych wewnętrznych bytów.

Fabryka może generować unikalny identyfikator Id encji lub on może być do niej przekazany – aby encja nie zmieniła tożsamości nie można generować Id na nowo.

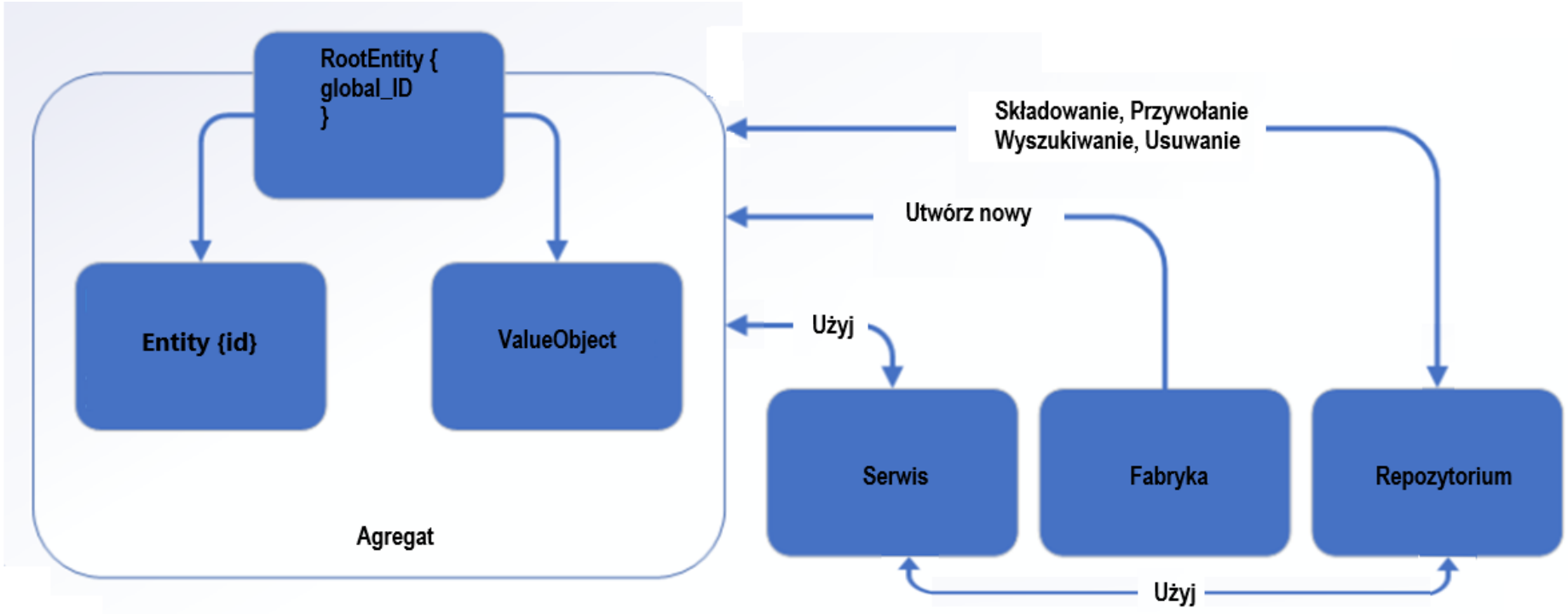
Stosowanie Fabryki – to nie tylko abstract factory ale dowolna struktura która tworzy obiekt. To ogólnie jakiś builder, który pozwoli tworzyć poprawny obiekt.

Fabryka (c.d.)

Wykorzystanie Fabryki może być nadmiarowe gdy:

- konstruktor jest atomowy, tzn. że wszystkie informacje niezbędne do wykonania obiektu znajdują się w konstruktorze,
- wszystkie atrybuty tworzonego obiektu są publiczne,
- klasa jest typem.

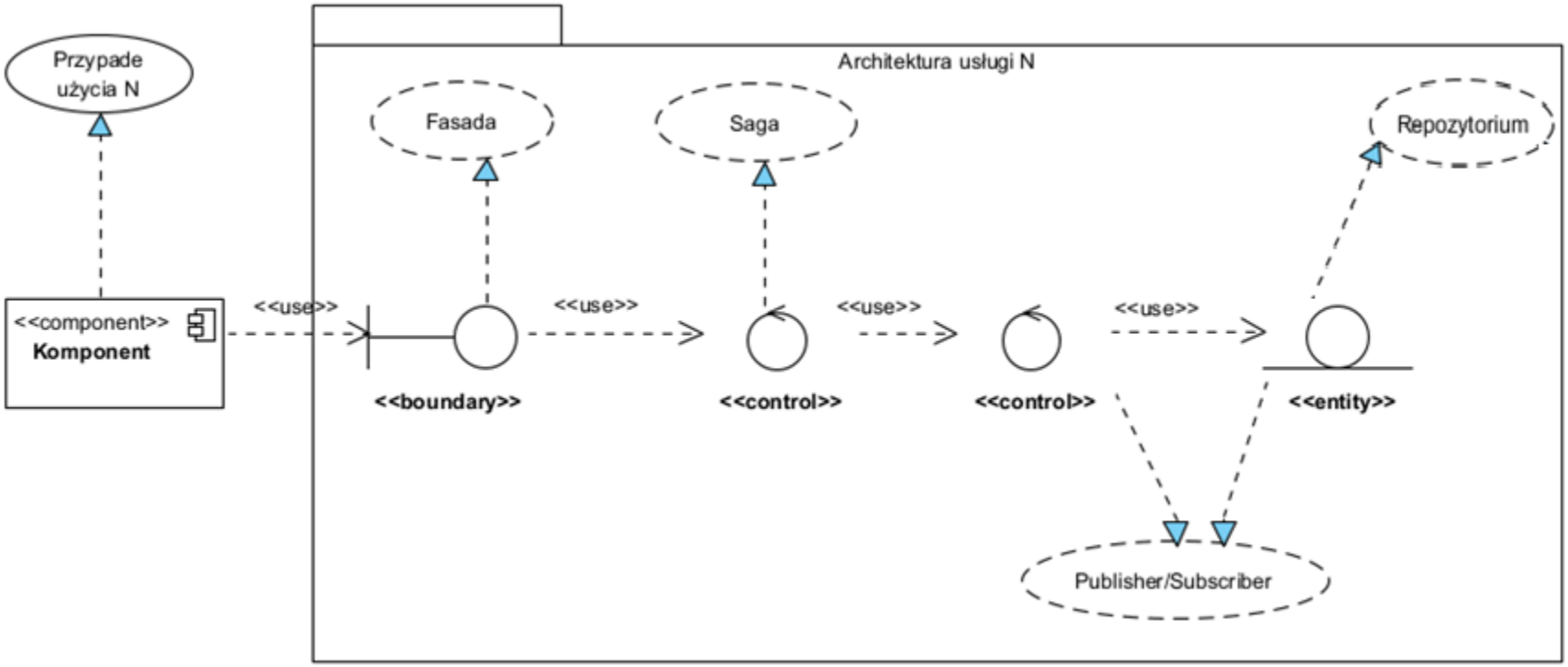
Podsumowanie zależności



Inne wzorce

- Event – model wydarzeń biznesowych, który może być wykorzystany do przetwarzania równoległego lub komunikacji pomiędzy Bounded Context.
- Policy – domknięcie/dostrojenie agregatu.
- Specification – złożone reguły biznesowe.
- Saga – modeluje złożony proces, którego stan jest trwały i zależy od wielu zdarzeń; obiekty Sagi nasłuchują wiele zdarzeń oraz ich stan jest utrwalany pomiędzy kolejnymi zdarzeniami.
- Envelope – przechowywanie danych (agregatu) w indeksowanych i mało złożonych obiektach.

Architektura usługi



DDD - rich model vs anemiczy model danych

Anemiczny model danych odzwierciedla relacyjną bazę danych, w której zapisywane są dane - nie zawiera żadnej logiki związanej z danymi, które są przechowywane przez obiekty. Anemiczny model danych posiada dane (pola/atrybuty), zestawy geterów i seterów oraz połączenia do innych modeli (referencje).

Model anemiczny składa się tylko z encji, które nie są pogrupowane w agregaty – wszystkie te encje tworzą jeden, monolityczny model danych.

W modelu anemicznym serwisy są konieczne do dokonywanie walidacji stanu obiektów, które są modyfikowane.

Rich model zawiera w klasie także operacje, które realizują logikę domenową (biznesową). Dane i zachowania koegzystują spójnie (zachowanie jest elementem i rezultatem modelu) – to logika realizuje inicjację obiektów, walidację i operowanie na modelu. W rich modelu serwisy nie są tak często wykorzystwane jak w modelu anemicznym.

Wzorzec CQRS (c.d.)

https://martinfowler.com/bliki/CQRS.html

Założenia CQRS

• Separacja rozkazów kierowanych do domeny i kwerend czytających stan domeny (query nie mutuje stanu, podczas gdy komenda mutuje stan).

• Osobny model do odczytu, odświeżany np. przez zdarzenia tworzone przez obiekty domenowe w ważnych momentach ich życia. Osobnych modeli do odczytu może być wiele i mogą być implementowane na innych bazach niż relacyjne.

• Read model często implementujemy jako widok zmaterializowany (rdbms w III postaci normalnej szybko zapisuje ale wolno czyta, a w systemach biznesowych częściej się czyta niż zapisuje).

CQRS – podstawowe pojęcia

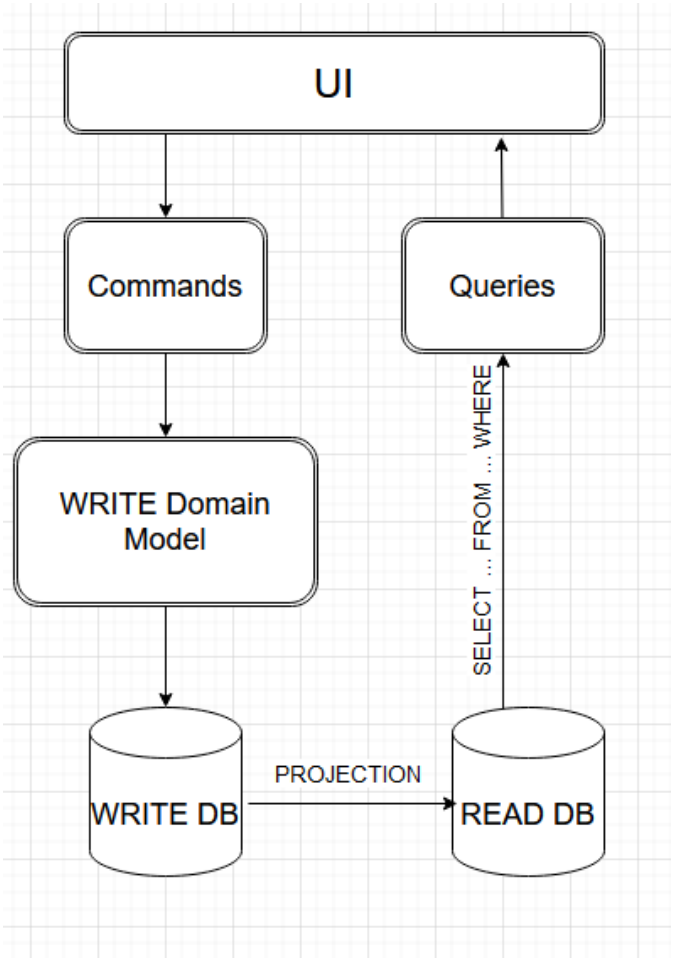
Commands – reprezentują zapisy do systemu i nic nie zwracają. Muszą mieć poprawne dane (muszą być zwalidowane). Są niemutowalne. Jak poznać ID utworzonego obiektu biznesowego?

Queries – reprezentują odczyty z systemu

READ Model – model dostosowany do odczytów

WRITE Domain Model – model do zapisów

Założenia CQRS (c.d.)



Dygresja - CQS

CQS (Command Query Separation) powstały przy okazji prac nad językiem Eiffel (autor: Bertrand Meyer) i CQRS (Command Query Responsibility Segregation) są bardzo ze sobą powiązane.

W CQS wprowadzono wcześniej zasadę, że ***„Pytanie nie powinno zmieniać odpowiedzi.” ,*** co jest w CQRS tym samym w kontekście Query.

W CQRS zadaniami podziału zajmują się osobne klasy, podczas gdy w CQS były to metody.

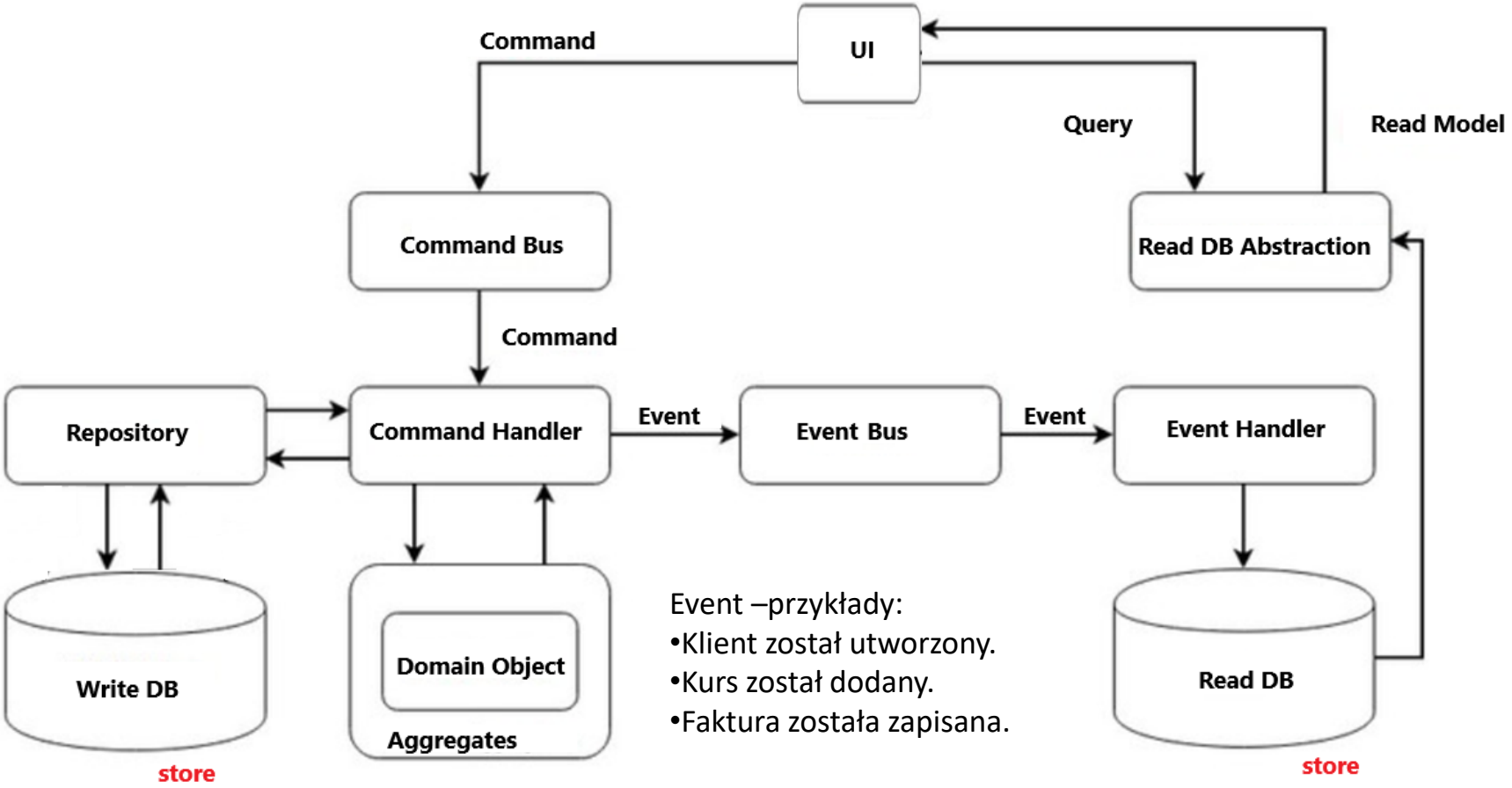
CQRS – podstawowe pojęcia

Command Handler – podstawowe zadanie to walidacja komendy a następnie modyfikacja wewnętrznego stanu aplikacji/domeny albo samodzielnie albo poprzez wywołanie Serwisu.

Command Handler może odkładać eventy domenowe na Event Bus.

Command Handler Używa repozytoriów (write) do persystencji/odtwarzania stanu

CQRS – events i Events Sourcing



Event –przykłady:
- Klient został utworzony.
- Kurs został dodany.
- Faktura została zapisana.

CQRS – Command Bus

Events - używane jako mechanizm do rejestrowania faktów, które wystąpiły w wyniku operacji zapisu.

Command Bus – kolejkuje komendy, łączy komendy z handlerami, wywołuje obsłużenie komendy przez handler.

Middleware – otacza Command Bus i dodaje takie funkcjonalności jak: logowanie, automatyczne zapisywanie w repozytorium, transakcje itp..

CQRS

Query – pobranie danych z systemu

Query Handler – 1 query na 1 handler, używa repozytoriów READ (pobierają dane w najprostszej postaci np. prosty SQL), które są wstrzykiwane przez interfejsy.

Query Bus – kolejkuje obiekty query i łączy je z handlerami; wywołuje obsłużenie zapytania przez handler queryBus. Tutaj Middleware otacza query bus i dodaje extra w postaci np.. logowanie, obsługa blędów …

CQRS != Event Sourcing

Event Sourcing - mechanizm odtwarzania aktualnego stanu aplikacji (patrz obiektów domenowych) na podstawie zdarzeń składowanych w magazynie danych Event Store. Event stories z założenia to bazy klucz-wartość, gdzie kluczem jest identyfikator strumienia.

Przykład lokalnego EventStore:

```
docker run -d --name eventstore -p 2113:2113 -p 1113:1113 eventstore/eventstore
```

Event Sourcing jest realizowane w ramach CQRS, ale CQRS nie musi posiadać Event Sourcing, stąd CQRS != Event Sourcing.

Zalety CQRS

- Separacja zapisu i odczytu – możemy skupić się na optymalizacji zapytań w READ model, także w Write model można użyć EventSourcing.
- System mniej podatny na regresję – zmiany robimy w commands i querys, ogranicza awarie w separacji.
- Pojedyńcza odpowiedzialność (S z SOLID).
- Kod jest przejrzysty a testowanie łatwiejsze (bo wstrzykujemy zależności przez interfejsy).
- Rozdzielanie pracy zespołów.
- Przeniesienie części funckjonalnośći na mikroserwisy.
- Reużywalność.

Wady CQRS

- Większa złożoność projektu – większa liczba klas (na skutek rozdzielania domeny).
- Rozdzielenie bazy danych – możliwe problemy ze spójnością w zakresie synchronizacji przy rozdzieleniu Store.

Uwagi dotyczące CQRS

Używać, gdy pojawia się złożona logika biznesowa.

Używać, gdy mamy na uwadze stosowanie różnych silników baz danych.

Używać, gdy chcemy zrównoleglić prace.

W CQRS nie realizujemy CRUD.

W CQRS jako READ DB może być użyty CACHE.

Może zachodzić READ DB > WRITE DB gdyż READ DB to suma różnych specjalizowanych READ DB.

Przykładowe informacje dodatkowe

https://docs.microsoft.com/pl-pl/azure/architecture/patterns/pipes-and-filters

https://pl.wikipedia.org/wiki/Model-View-Controller

DDD dla architektów oprogramowania Vaughn Vernon