Paradygmaty Programowania

Style architektury oprogramowania

dr inż. Marcin Bączyk

Wykład 10

16 maja 2022

Treść dzisiejszego wykładu

- Style architektury oprogramowania
- Wzorce architektoniczne

Literatura

- https://www.oreilly.com/content/software-architecture-patterns

Kryteria oceny architektury oprogramowania

- Łatwość tworzenia (ang. *ease of development*)
- Łatwość testowania (ang. *testability*)
- Łatwość wdrożenia (ang. *ease of deployment*)
- Skalowalność (ang. *scalability*)
- Elastyczność (ang. *agility*)
- Wydajność (ang. *performance*)

Wybrane style architektury oprogramowania

- „Bryła błota”
- Styl architektury warstwowej
- Styl architektury mikrojądra
- Styl architektury bazującej na usługach
- Styl architektury sterowanej zdarzeniami

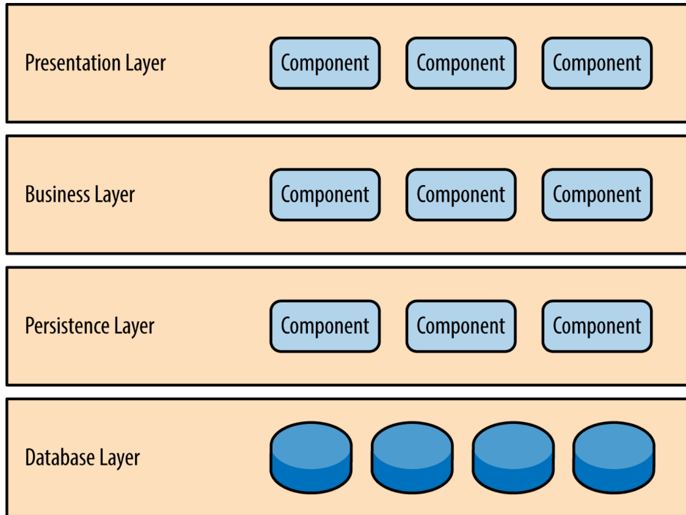
„Bryła błota" (ang. *big ball of mud*)

- Brak jakiejkolwiek zauważalnej struktury architektonicznej określa się mianem bryły błota.
- Systemy wykazują oznaki nieuregulowanego wzrostu i powtarzających się doraźnych napraw.
- Informacje przekazywane są bezładnie pomiędzy odległymi elementami systemu.
- Często wszystkie ważne informacje stają się globalne lub są duplikowane.
- Efekt: praktycznie każda zmiana w klasach systemu powoduje trudne do przewidzenia skutki uboczne dla systemu.
- Każdy element zależy od wszystkich innych – „spaghetti code".

Architektura warstwowa (ang. *layered architecture*)

- Składniki zorganizowane są w logiczne poziome warstwy, przy czym każda warstwa pełni określoną rolę w aplikacji.
- Składniki w obrębie konkretnej warstwy mają ograniczony zakres, zajmują się logiką, która odnosi się do konkretnej warstwy.
- Podstawowe warstwy to:
  - prezentacji,
  - biznesowa,
  - dostępu do danych (trwałości/utrwalania danych) – ang. *data access layer* / *persistence layer*,
  - bazodanowa.
- Podział na warstwy – według ról "technicznych".
- Podział w ramach warstwy – według funkcji "z dziedziny problemu".

Architektura warstwowa



Źródło: https://www.oreilly.com/content/software-architecture-patterns

Architektura warstwowa – izolacja

- Warstwy **zamknięte**: elementy z danej warstwy wysyłają żądania wyłącznie do warstw sąsiednich.
- Warstwy **otwarte**: możliwa bezpośrednia komunikacja między różnymi warstwami – niekoniecznie sąsiednimi, np. prezentacji i baz danych.
- Użycie warstw zamkniętych jest *zalecane*:
  - redukuje zależności między elementami różnych warstw, umożliwia używanie uniwersalnych komponentów,
  - często zwiększa wydajność, np. warstwa utrwalania danych (ang. *persistence*) agreguje połączenia z serwerem czy zapytania do baz danych.
- Niektóre elementy w warstwach pośredniczących służą *wyłącznie* przekazaniu żądania do kolejnej warstwy ("**lej architektoniczny**" – ang. *achitecture sinkhole anti-pattern*)
  - przykład: `gui.getMathModule().makeDataBaseReader().getPi();`
  - pogorszenie wydajności (tworzone są nowe instancje) i czytelności,
  - jeśli większość żądań przechodzi przez przynajmniej warstwę bez dodatkowej obsługi, warto otworzyć warstwy.

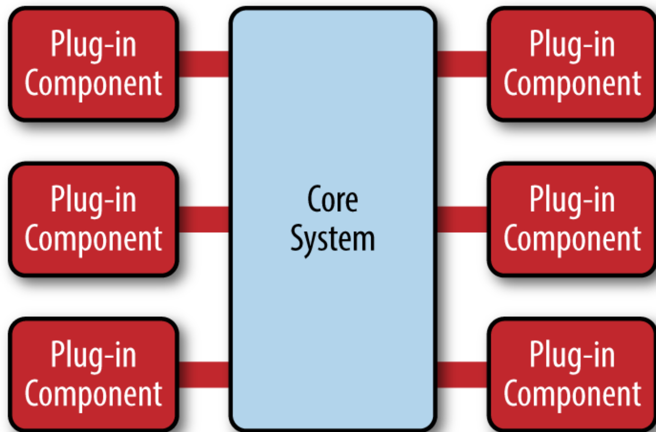
Architektura warstwowa – wady i zalety

- Jest dobrym rozwiązaniem dla małych prostych aplikacji lub witryn internetowych.
- Łatwa do wytworzenia – podział na warstwy często odzwierciedla strukturę firmy programistycznej: specjaliści od interfejsu graficznego, problemów obliczeniowych, baz danych itp. (prawo Conwaya)
- Ze względu na prostotę i znajomość wśród programistów jest jedną z najtańszych architektur.
- Dobry punkt wyjścia, gdy jeszcze nie jest wiadomo, jaka architektura zostanie ostatecznie wybrana.
- Wraz ze wzrostem systemu spada elastyczność, trudniejsze stają się: testowanie i utrzymanie.
- Trudna do wdrożenia – dodanie nowych funkcji wymaga modyfikacji (i ponownej instalacji) wszystkich warstw.

Architektura mikrojądra (ang. *microkernel architecture*)

- Architektura złożona z dwóch rodzajów składników: podstawowego systemu (rdzenia) i dołączanych wtyczek.
- Logika podzielona jest na niezależne (z reguły), dołączane składniki i podstawowy system.
- System podstawowy może mieć strukturę warstwową (podzielony technicznie) lub strukturę modułową (podzielony dziedzinowo).
- Może być stosowana do całego systemu lub do jego części.
- Dołączane składniki to samodzielne niezależne wtyczki zawierające wyspecjalizowane mechanizmy przetwarzania, dodatkowe funkcje lub niestandardowy kod.

Architektura mikrojądra



Źródło: https://www.oreilly.com/content/software-architecture-patterns

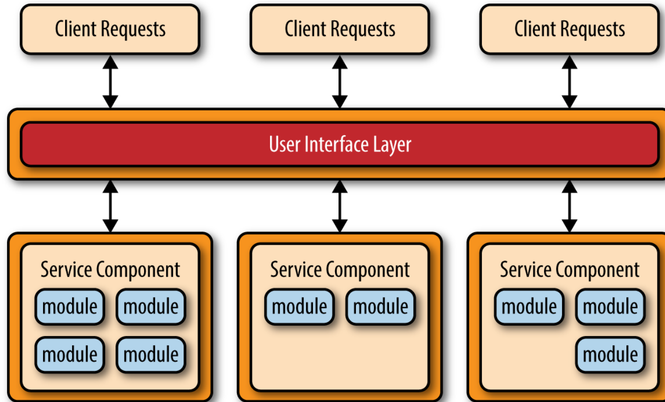
Architektura mikrojądra – wady i zalety

- Podstawowa zaleta – konfigurowalność.
- Tworzenie może być trudne – skomplikowane interfejsy rdzeń-wtyczki.
- Duża elastyczność
  - podstawowy system – prosty i stabilny,
  - wtyczki – rozwijane niezależnie i również proste.
- Łatwość wdrożenia – wtyczki mogą być dołączane bez przerw w pracy podstawowego systemu.
- Łatwe testowanie:
  - wtyczki testowane niezależnie,
  - funkcjonalność wtyczek może być emulowana w podstawowym systemie, żeby przetestować ich współpracę.
- Wysoka wydajność – przy ograniczeniu do najbardziej potrzebnych wtyczek.
- Skalowalność – raczej słaba:
  - jeśli podstawowy system musi nadzorować pracę całości,
  - jeśli wtyczki nie są skalowalne.

Architektura oparta na usługach (ang. *microservices architecture*)

- Uważana za jeden z najbardziej pragmatycznych stylów architektonicznych.
- Odpowiedź na wady architektur monolitycznych.
- Architektura rozproszona o relatywnie niskim poziomie złożoności i kosztach wytworzenia.
- Opiera się na rozproszonej makrowarstwowej strukturze składającej się z osobno wdrażanych elementów:
  - interfejsu użytkownika,
  - usług o większych stopniu ogólności,
  - monolitycznej bazy danych.
- Usługi są niezależnymi i wdrażanymi osobno częściami aplikacji.

Architektura oparta na usługach



Źródło: https://www.oreilly.com/content/software-architecture-patterns

Architektura oparta na usługach

- Struktura jest wyznaczona przez dziedziny (podział dziedzinowy), a nie względy techniczne (jak np. logika prezentacji czy biznesowa).
- Zmiany dokonywane w jednej z dziedzin mają wpływ tylko na jedną konkretną usługę.
- Wszystkie usługi mogą współdzielić ten sam interfejs użytkownika i tę samą bazę danych.
- Wysoka odporność na błędy.
- Zapewnia dostatecznie dobry poziom modułowości bez konieczności wnikania w złożoność i pułapki związane z rozdrobnieniem usług.

Architektura oparta na usługach – wady i zalety

- Słabe sprzężenia między elementami – łatwość tworzenia, wdrażania, testowania, duża elastyczność.
- Dobra skalowalność – można rozbudowywać najbardziej obciążone moduły.
- Słaba wydajność – duży koszt komunikacji między niezależnymi modułami.

Architektura sterowana zdarzeniami (ang. *event-driven architecture*) – wady i zalety

- Wysoka wydajność uzyskiwana dzięki asynchronicznej komunikacji i wysokiemu stopniowi przetwarzania równoległego.
- Wysoka skalowalność uzyskiwana poprzez programowe równoważenie obciążenia procesorów zdarzeń.
- Odporność na błędy osiągana dzięki stosunkowo niezależnym i asynchronicznym procesorom zdarzeń.
- Dodawanie nowych funkcji jest stosunkowo proste.
- Duża złożoność i słaba testowalność ze względu na niedeterministyczny i dynamiczny przepływ zdarzeń.

Porównanie architektur

Architektura	Tworzenie	Testowanie	Wdrażanie	Skalowalność	Elastyczność	Wydajność
Warstwowa	łatwe	trudne	trudne	mała	mała	mała
Zdarzeniowa	trudne	trudne	łatwe	duża	duża	duża
Mikrojądro	trudne	łatwe	łatwe	mała	duża	duża
Mikrousługi	łatwe	łatwe	łatwe	duża	duża	mała

Architektura oprogramowania

Wzorzec architektoniczny

Uznany i sprawdzony sposób rozwiązania danego problemu z zakresu architektury oprogramowania.

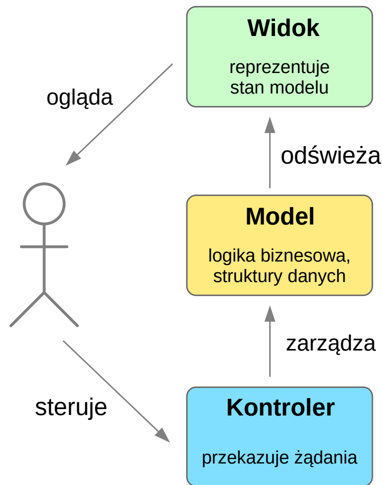
Wzorce architektoniczne, w przeciwieństwie do wzorców projektowych, dotyczą całego systemu informatycznego bądź jego modułów.

Przykłady wzorców architektonicznych

- Architektura trójwarstwowa: warstwa prezentacji, biznesowa i dostępu do danych.
- Model-Widok-Kontroler (MVC)
- Model-Widok-Prezenter (MVP)
- Peer-to-peer
- Architektura zorientowana na usługi

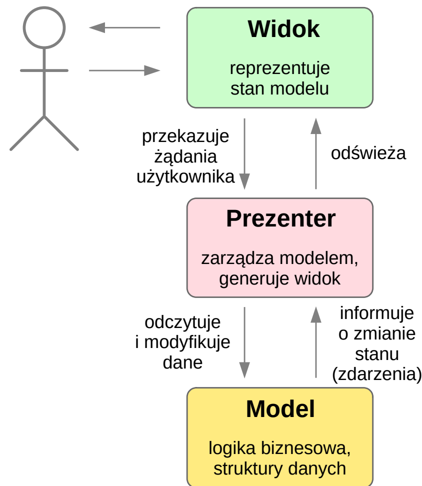
Model-Widok-Kontroler (ang. *Model-View-Controller, MVC*)

- **Widok** i **Kontroler** odpowiadają warstwie prezentacji w modelu trójwarstwowym.
- Użytkownik obserwuje **Widok** i steruje **Kontrolerem**.
- **Model** reprezentuje stan programu i zarządzającą nimi logikę.
- Kontrolki GUI zawierają elementy **Widoku** (np. etykiety) lub **Widoku** i Kontrolera (np. pola tekstowe, przyciski).



Model-Widok-Prezenter (ang. *Model-View-Presenter, MVP*)

- Użytkownik komunikuje się jedynie z **Widokiem** (GUI).
- **Model** reprezentuje stan programu i zarządzającą nimi logikę.
- **Prezenter**
  - przetwarza żądania Użytkownika,
  - informuje Widok, *co i jak* ma być wyświetlone



Model-Widok-Kontroler