Aplikacje sieciowe i webowe

dr inż. Rafał Brociek

Katedra Zastosowań Matematyki i Metod Sztucznej Inteligencji
Wydział Matematyki Stosowanej
Politechnika Śląska

17.10.2023

Wzorzec CQRS

Command query responsibility segregation (CQRS) to wzorzec architektoniczny, w którym rozróżnia się pobieranie i aktualizowanie danych. Wzorzec ten został opracowany przez Bertranda Meyera jako sposób na zapewnienie, że metoda pracująca z danymi jest w stanie wykonać tylko jedno z dwóch zadań. Metoda może albo pobierać informacje, albo je modyfikować, ale nie może wykonywać obu tych czynności jednocześnie. W programowaniu obiektowym paradygmat ten dzieli obowiązki na dwie różne klasy — jedną do odczytu, a drugą do usuwania, tworzenia i aktualizowania.

Wzorzec CQRS

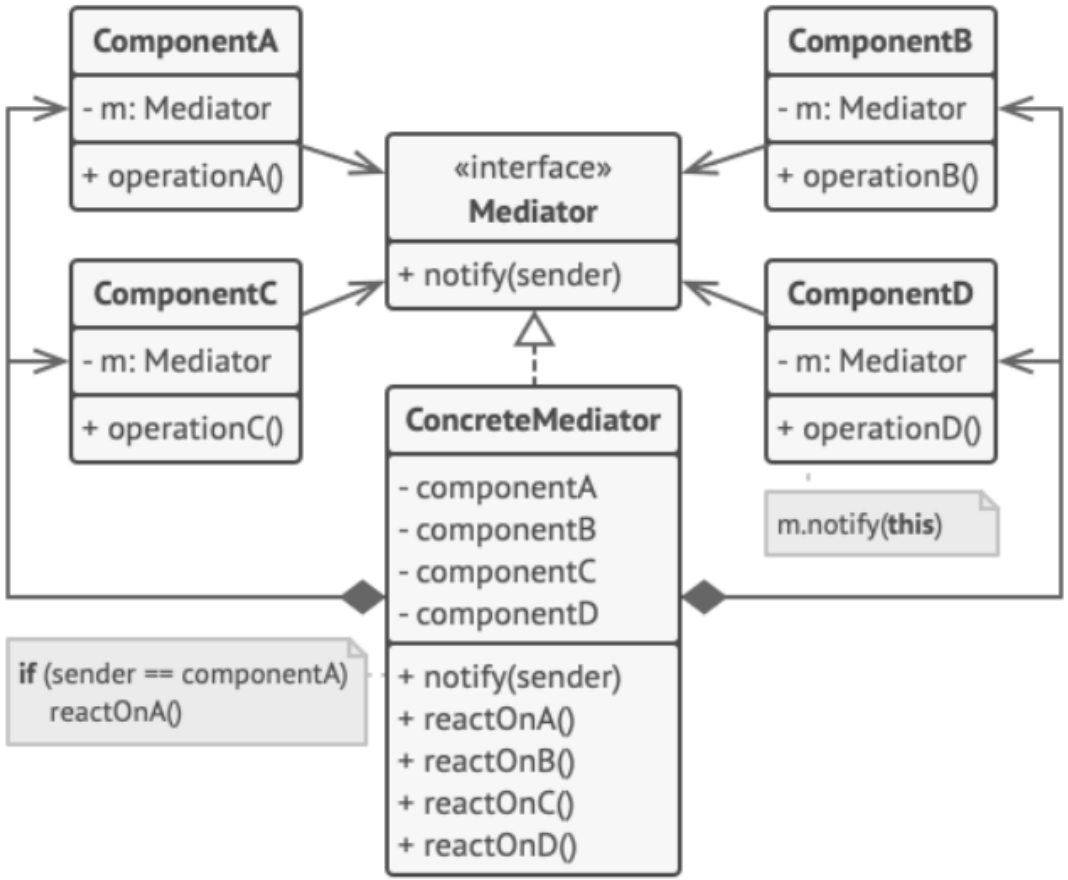
Każda metoda w systemie powinna być zaklasyfikowana do jednej z dwóch grup:

- **Command** (polecenia) - są to metody, które zmieniają stan aplikacji i nic nie zwracają.
- **Query** (zapytanie) - są to metody, które coś zwracają, ale nie zmieniają stanu.

Wzorzec mediatora

Wzorzec mediatora jest behawioralnym wzorcem projektowym. W uproszczeniu obiekt mediatora ma za zadanie zarządzać komunikacją między obiektami. W jakiej sytuacji wzorzec ten jest przydatny? Jeśli wiele obiektów jest od siebie zależnych i muszą się one ze sobą komunikować, to sterowanie procesem komunikacji może być kłopotliwe. W takiej sytuacji powinno ograniczać się bezpośrednią komunikację między obiektami na rzecz komunikacji pośredniej. To właśnie obiekt mediatora jest za to odpowiedzialny. Wówczas odpowiednie klasy zależą od mediatora, a nie od siebie nawzajem. Wzorzec mediatora pozwala hermetyzować złożone plątaniny relacji pomiędzy obiektami w jednym obiekcie. Im mniej zależności ma klasa, tym łatwiej ją modyfikować, rozszerzać lub użyć ponownie.

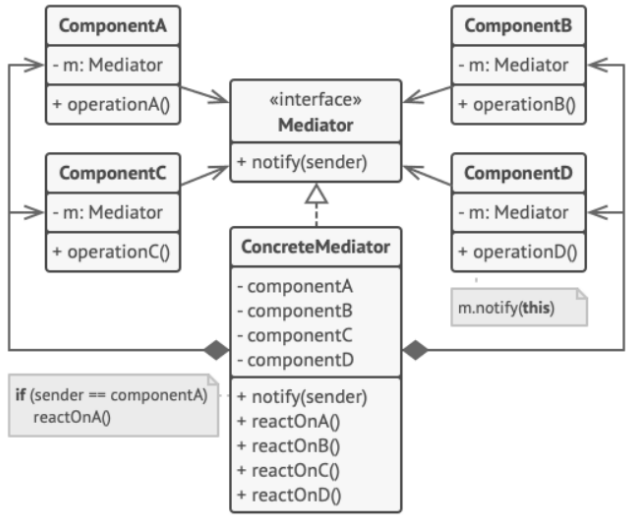
Wzorzec mediatora - schemat



Rysunek: Schemat wzorca mediatora (źródło: `https://refactoring.guru/pl/design-patterns/mediator`)

Component

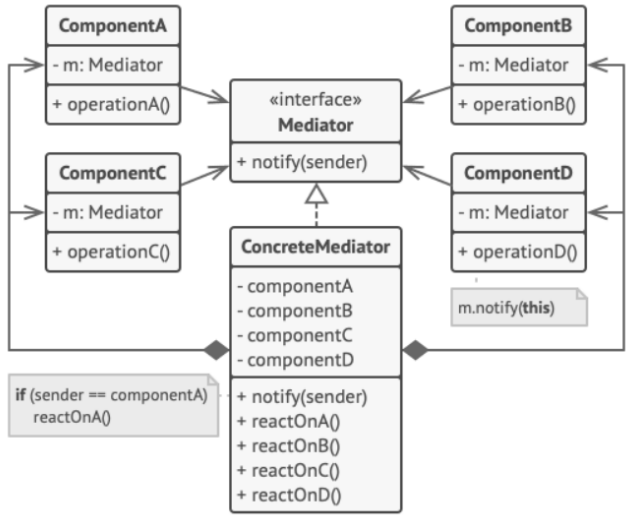
Obiekty komponenty (`Component`) muszą się ze sobą komunikować, zależności między nimi są dosyć skomplikowane. Obiekty te nie są ze sobą powiązane bezpośrednio, a komunikacja odbywa się przez obiekt mediatora. Każdy z tych komponentów jest zależny od obiektu implementującego interfejs wzorca mediatora (nic nie wiedzą o konkretnym mediatorze).



Rysunek: Schemat wzorca mediatora (źródło: `https://refactoring.guru/pl/design-patterns/mediator`)

Interfejs mediatora

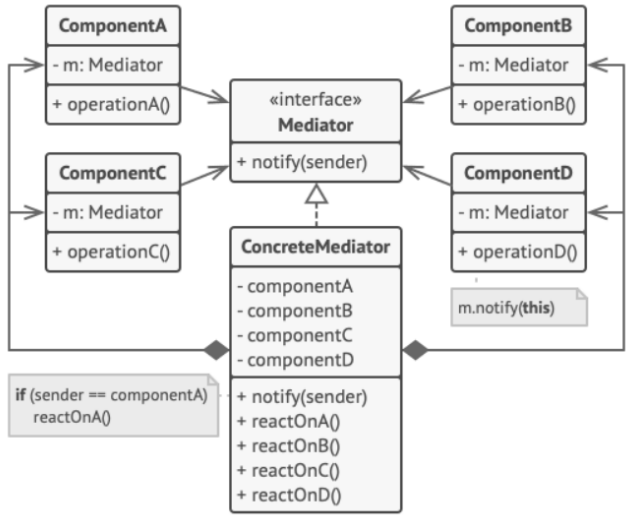
Interfejs mediatora zawiera metody komunikacji z komponentami, np. metodę `notify(sender)`. Komponenty jako argument metody `notify` mogą przekazywać dowolne obiekty, np. instancję do samego siebie. Nie może to spowodować bezpośredniej relacji między komponentami.



Rysunek: Schemat wzorca mediatora (źródło: `https://refactoring.guru/pl/design-patterns/mediator`)

Instancja obiektu mediatora

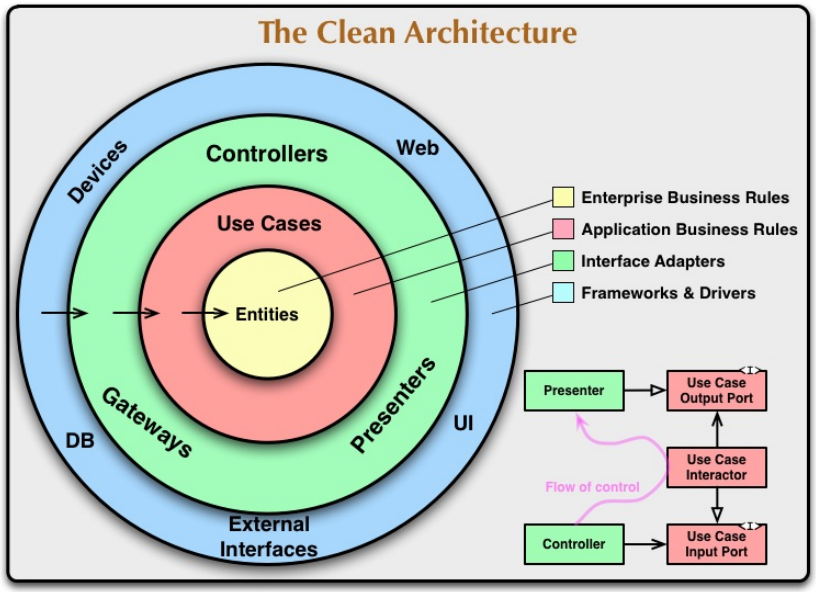
Konkretny obiekt mediatora definiuje relacje między komponentami. Często przechowują referencje do obiektów komponentów, których komunikacją zarządzają. Komponenty nie są świadome samych siebie. Nadawca przesyła informację, ale nie wie nic o odbiorcy, zaś odbiorca nie jest świadomy nadawcy.



Rysunek: Schemat wzorca mediatora (źródło: `https://refactoring.guru/pl/design-patterns/mediator`)

Omówienie tworzonego projektu

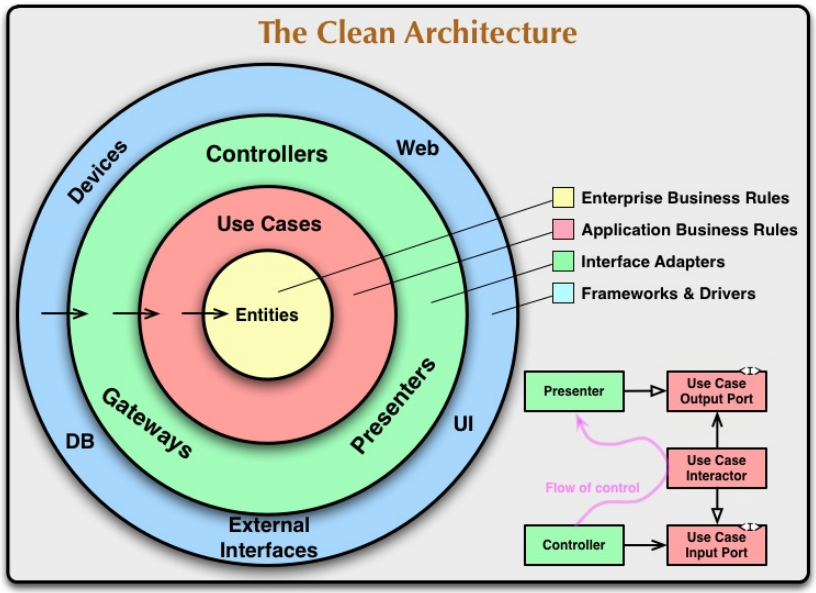
Entities. Odnosi się do warstwy `Cars.Domain` i klasy `Car`.



Rysunek: Schemat wzorca czystej architektury zaproponowany przez Roberta C. Martina (źródło: `https://blog.cleancoder.com/`)

Omówienie tworzonego projektu

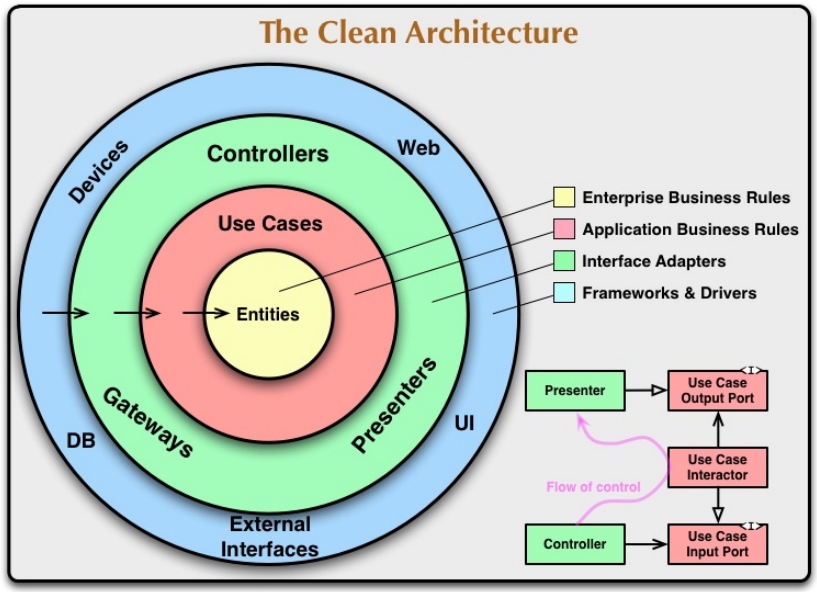
Use Cases. Odnosi się do warstwy `Cars.Application`. W tej warstwie będzie zaimplementowana logika biznesowa aplikacji. Dla każdej funkcjonalności (use case) implementować będzie osobną klasę, która będzie obsługiwać dany przypadek. Rozpatrywane przypadki: zwrócenie listy wszystkich samochodów, zwrócenie informacji o danym samochodzie (na podstawie id), tworzenie nowego obiektu, edycja istniejącego obiektu, usuwanie auta. Warstwa ta jest odseparowana od interfejsu użytkownika i nic o nim nie wie.



Rysunek: Schemat wzorca czystej architektury zaproponowany przez Roberta C. Martina (źródło: `https://blog.cleancoder.com/`)

Omówienie tworzonego projektu

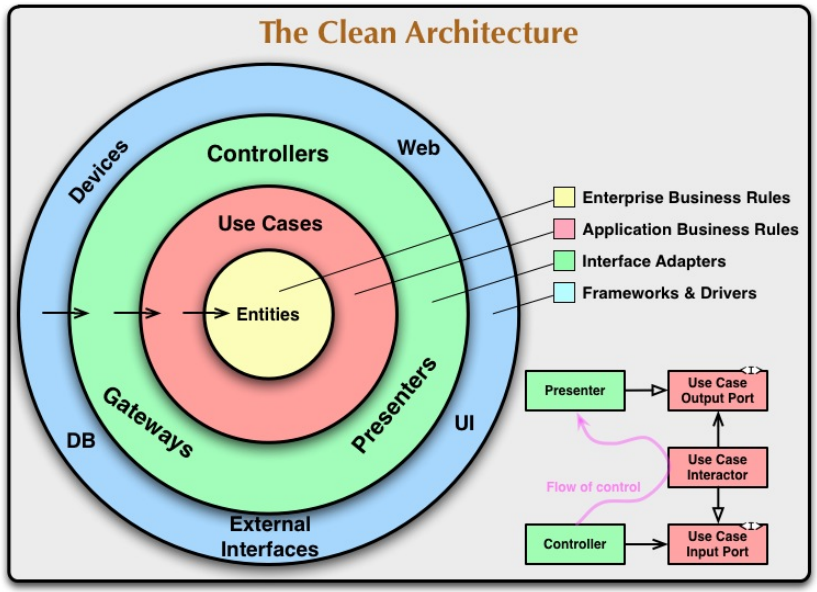
Interface Adapters (Controllers, Gateways, Presenters). Odnosi się do `Cars.API` i odgrywa rolę łącznika pomiędzy logiką biznesową (use cases), a widokiem. W naszym przypadku będziemy używać kontrolerów.



Rysunek: Schemat wzorca czystej architektury zaproponowany przez Roberta C. Martina (źródło: `https://blog.cleancoder.com/`)

Omówienie tworzonego projektu

Framework and Drivers. Warstwa odnosząca się do zewnętrznych frameworków. W naszym przypadku chodzi o interfejs użytkownika (aplikacja kliencka w React) oraz bazy danych (SQLite). Dwie wewnętrzne warstwy nie wiedzą nic o stosowanej bazie danych oraz interfejsie użytkownika.

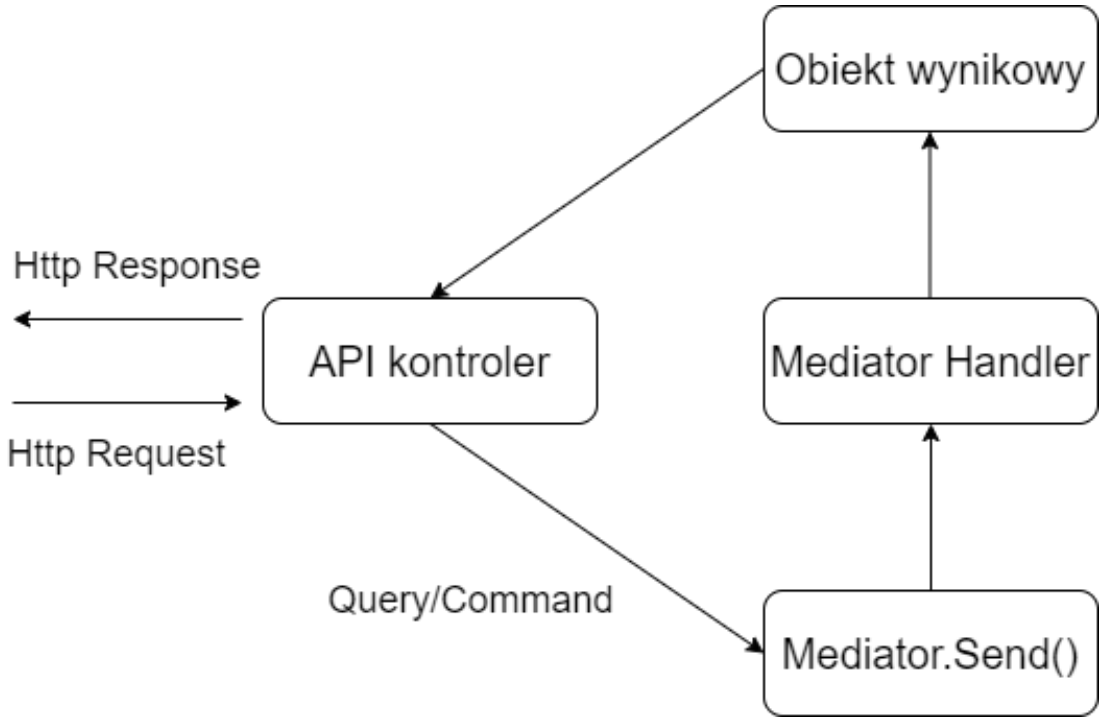


Rysunek: Schemat wzorca czystej architektury zaproponowany przez Roberta C. Martina (źródło: `https://blog.cleancoder.com/`)

Przepływ danych w aplikacji

W tworzonej na zajęciach aplikacji, do komunikacji, przepływu danych skorzystamy ze wzorca mediatora. Za każdym razem jak kontroler otrzyma żądanie, przesyła zapytanie/polecenie (query/command) po przez mediatora (metoda `Mediator.Send()`) do odpowiedniego obiektu typu `Mediator Handler` (warstwa `Cars.Application`). Obiekt ten jest odpowiedzialny za przetworzenie zapytania/polecenia i zwrócenie odpowiedzi do kontrolera, a ten zwraca odpowiedź http klientowi. W celu implementacji tego wzorca oraz CQRS skorzystamy z pakietu NuGet `MediatR`.

Przepływ danych w aplikacji



Rysunek: Schemat przepływu kontroli w aplikacji

Dziękuję za uwagę