## DDD (Domain Driven Design) oraz architektura cebulowa

Domain Driven Design jest pewnym szczególnym podejściem do projektowania i wytwarzania oprogramowania. Przykłada ono szczególną uwagę do definiowania i modelowania obiektów i komponentów w projekcie, aby te maksymalnie odzwierciedlały rzeczywistość, z którą powiązany jest ten projekt. Dzięki takiemu założeniu, programiści i architekci oprogramowania w znacznym stopniu mogą być odciążeni od modelowania logiki biznesowej w danej aplikacji, przez ekspertów specjalizujących się w danym zagadnieniu. Model takiej domeny jest jądrem całej aplikacji, wokół której powstają kolejne komponenty i interfejsy.

Jednym z częstych aspektów idących w parze za podejściem DDD jest pewien specyficzny podział aplikacji na osobne warstwy, które zawierają w sumie moduły o zbliżonych właściwościach. Taki podział na warstwy często wiąże się z określeniem „architektura cebulowa (onion architecture)”. Najważniejszą warstwą takiej architektury jest warstwa domeny. Komponenty które się nią składają, specyfikują struktury danych, które odzwierciedlają modelowaną rzeczywistość. Ponadto warstwa ta może zawierać logikę biznesową, jaka jest związana z domeną. Osobiście jednak preferują specyfikować logikę biznesową na warstwie aplikacji. Domena powinna być lekka oraz posiadać minimalną zależność od bibliotek zewnętrznych. Jest ona jakby sercem całej architektury, wokół którego buduje się kolejne warstwy. Bardzo istotne jest, żeby w początkowej fazie projektowania aplikacji, jak najdokładniej przeanalizować i zamodelować strukturę domeny. Praktycznie każda zmiana, jaka zachodzi w strukturze domeny, może mieć swoje reperkusje w innych komponentach składających się na całość aplikacji.

Kolejnym elementem „onion architecture” jest tak zwana warstwa aplikacji. W przedstawionej aplikacji służy ona do dwóch celów. Pierwszym jej zadaniem jest implementacja pewnej logiki biznesowej, jaką niesie ze sobą domena. Drugim, nie mniej istotnym, jest bycie pomostem pomiędzy kolejnymi modułami warstwy infrastruktury. To tutaj scalamy ze sobą informacje domenowe, z konkretnymi modułami funkcjonalnymi z warstwy infrastruktury.

Warstwa infrastruktury służy nam do przygotowania bibliotek, które następnie będą wykorzystywane przez warstwę aplikacji, aby zrealizować logikę biznesową danego systemu. Moduły tworzące taką infrastrukturę mogą pełnić wielorakie działania, w zależności od funkcjonalności jakie musi zapewnić dana aplikacja. Jako przykłady mogą, służyć elementy aplikacji zajmujące się takimi funkcjonalnościami jak komunikacja ze środowiskiem zewnętrznym z wykorzystaniem REST, wysyłanie maili, dostęp do bazy danych, zapewnienie autoryzacji i autentykacji. Każdy z tych elementów służy do zapewnienia pewnej funkcjonalności, która potem zostanie wykorzystana na rzecz logiki biznesowej. Istotne jest, aby minimalizować zależności pomiędzy konkretnymi komponentami w tej warstwie. W idealnej sytuacji, żaden z komponent z warstwy infrastruktury nie posiada bezpośredniej zależności z innym. Dzięki temu, w późniejszym czasie programista może prościej zmienić sposób, w jaki jest wykonywana dana funkcjonalność, bez znacznej ingerencji w pozostałe elementy aplikacji. Zmiana taka zachodzi wtedy tylko na warstwie aplikacji, która spina ze sobą całą infrastrukturę.

Jeżeli w danej aplikacji integrujemy się z wieloma systemami zewnętrznymi, nad którymi nie mamy bezpośredniej kontroli, dobrym pomysłem jest wydzielenie z infrastruktury osobnego bytu – tak zwanej warstwy integracji. Moduły takiej warwy służą nam do definiowania i enkapsulacji sposobów komunikacji z odrębnymi systemami. W warstwie infrastrukturalnej zostają tylko te elementy, nad którymi mamy pełną władzę, jeśli chodzi o sposób ich działania. Głównym zadaniem tej warstwy, jest maksymalne uniezależnienie naszego produktu od zmian, jakie zachodzą w systemach, z którymi się integrujemy. Przykładem takiego uniezależnienia może być opakowanie metod wystawionych przez zewnętrznych klientów, w nasze własne metody, które zostaną potem wykorzystane na niższych. Dzięki takiemu podejściu, potencjalna zmiana w sposobie działania biblioteki klienckiej, będzie miała swoje następstwo tylko w module, który taką bibliotekę obsługuje. Cała reszta aplikacji nie powinna odczuć jakichkolwiek zmian w sposobie działania klienta. Idealnym przykładem takiego podejścia są wzorce projektowe Gateway lub Fasada, które w bardzo dobry sposób zapewniają nam jakże ważną niezależność. Zostaną one omówione w dalszej części pracy.

Do prezentacji danych dla użytkowników, służy warstwa UI (User Interfaces). W niej definiujemy sposób prezentacji danych oraz interakcji z użytkownikiem. Ponadto określamy tam również technologie, jakie zostaną do tego wykorzystane. W przypadku aplikacji webowych najczęściej są to wariacje HTML5 oraz różnych frameworków javascriptowych. Nic nie stoi jednak na przeszkodzie, żeby stworzyć klienta mobilnego lub desktopowego, który będzie komunikował się z serwerem, na którym stoi nasza aplikacja. Do komunikacji elementów warstwy UI z resztą aplikacji, służą odpowiednie komponenty w zdefiniowane w warstwie infrastrukturalnej lub rzadziej – integracyjnej. W przypadku aplikacji webowej, zazwyczaj wykorzystuje się protokół http.

Miejsce na graf.

## CQRS (Command Query Responsibility Segregation)

CQRS jest to rozwinięcie idei stworzonej w latach osiemdziesiątych przez Bertranda Meyera o nazwie CQS (Command Query Separation). Idea ta głosiła, że zapytanie o pewne dane, nie powinno zmieniać odpowiedzi jaką otrzymamy. Sama logika rozdzielenia zapytań o dane do ich alteracji i zmian miała być zrealizowana poprzez osobne metody w danych klasach bądź komponentach. CQRS idzie o krok dalej. Po pierwsze rozdzielenie tych dwóch kluczowych funkcjonalności zachodzi już na poziomie nie metod, a klas. Po drugie, w przypadku CQRS często zachodzi dublowanie domeny – tworzymy przykładowo osobne modele do modyfikacji danych oraz do ich pobierania. Ponadto w ramach jednego typu operacji również może istnieć taki podział. Weźmie na przykład jakże często stosowany interfejs master-detail. Z poziomu użytkownika widzimy listę wielu rekordów (master) zawierającą podstawowe informacje o danej strukturze. Jednak dany wiersz listy może na przekierować do widoku detail, gdzie dostajemy znacznie więcej szczegółów. W tradycyjnym podejściu do tworzenia domeny, posiadalibyśmy jeden model obrazujący informację wyświetlane na obu widokach. Przy takim podejściu programista musiałby szukać kompromisu pomiędzy optymalizacją zapytania o dane w przypadku widoku master, a szczegółowością w przypadku widoku detail. Prowadziłoby to do psucia modelu – z jednej strony mógłby być zbyt mało szczegółowy, z drugiej zapytanie do bazy danych mogłoby być nadal zbyt czasochłonne. W podejściu CQRS tworzymy dwa osobne modele obrazujące ten sam byt, lecz wykorzystywane w różnych sytuacjach. Dla widoku master przygotujemy lekki model zawierający minimum informacji. Jeśli chcemy poznać pełną ilość danych wykorzystujemy zapytanie z bardziej szczegółową strukturą.

W bardziej szczegółowym przypadku traktujemy CQRS jak wzorzec architektoniczny, który składa się z kilku istotnych elementów. Większość jego logiki znajduje się na warstwie integracji, jednak ma on również wpływ na działanie pozostałych komponentów. W skład implementacji tego wzorca wchodzą takie elementy jak Command, CommandHandler, CommandHandlerInvoker oraz QueryHandler. Implementacja tych modułów jest ściśle związana z technologią, w jakiej została wykonana nasza aplikacja. Zostanie to szczerzej omówione w opisie komponentów.

……………………….

## IoC oraz DI.

Z pojęciem IoC (Inversion of Controll) nierozerwalnie wiąże się wzorzec architektoniczny i projektowy DI (Dependency Incjection), więc to jemu przyjrzymy się najpierw.

Wstrzykiwanie zależności ma na celu usuwanie bądź minimalizowanie bezpośrednich zależności pomiędzy komponentami na rzecz asocjacji w ramach klas. Klasa taka sama nie nigdy nie tworzy obiektów, które będzie wykorzystywała w czasie swojego działania. Obiekty te wstrzyknięte zostają z zewnątrz, przez co programista zyskuje kontrolę nad działaniem takiej klasy. Wyróżniamy kilka różnych sposoby na wstrzykiwanie zależności.

Pierwszy z nich to wstrzykiwanie przez konstruktor. W konstruktorze danej klasy definiujemy jakiego typu obiektów będziemy potrzebować do jej działania. Następnie w momencie tworzenia jej instancji, dostarczamy obiekty o zdefiniowanych przez nas właściwościach. Daje nam to w pływ na sposób, w jaki dana klasa będzie spełniać swoje działanie. Nietrudno sobie wyobrazić sytuację, w której wstrzyknięcie obiektów o różnych parametrach, zapewni nam różnorakie działanie takiej klasy.

Drugą metodą stosowania dependency injection, jest wykorzystanie tak zwanej metody ustawiającej bądź mutującej. Zasada działania jest podobna jak w przypadku konstruktora, w parametrach metody podajemy parametry, które ustawią w wybrany przez nas sposób naszą klasę. Zaletą tego sposobu jest fakt, że może być on zastosowany wielokrotnie dla jednej instancji danej klasy w czasie działania programu. Powoduje to, że jeden obiekt w czasie swojego rzycie, może wykonać swoje operacje na różne sposoby. Wadą takiego rozwiązania jest fakt, że pomiędzy utworzeniem instancji danej klasy a uruchomieniem metody mutującej, część parametrów definiujących jej zachowania jest w stanie nieokreślonym. Może to powodować problem z działaniem takiego obiektu. Aby zabiec takiemu przypadkowi istnieje możliwość ustawienia stanu początkowego obiektu poprzez konstruktor.

Trzecią metodą na wstrzykiwania zależności jest wykorzystanie interfejsów. Metodę tę można rozumieć dwojako. Po pierwsze sama implementacja danego interfejsu wymaga na autorze danej klasy, aby zrealizował pewną funkcjonalność. Po drugie wykorzystując kontenery DI (lub szerzej IoC) o których będzie poniżej, jesteśmy wstanie spiąć ze sobą dany interfejs z jego wybraną implementacją. Nic nie stoi na przeszkodzie, aby w czasie działania aplikacji podmieniać daną implementację na inną. Dzięki temu z zewnątrz sprawujemy kontrolę nad danym zachowaniem. Ponadto wstrzykiwanie przez interfejs bardzo dobrze współpracuje z konstruktorami. Te zamiast przyjmować jako parametr typy danych obiektów, powinny wymagać tylko interfejsów, jakie te typy implementują.

W powyższych akapitach została poruszona kwestia sterowania z wykorzystaniem DI. Właśnie sterowanie i zależności są istotą IoC. Celem programisty jest uniezależnienie modułów z warstw wyższych, od działania komponentów z warstw niższych. Dzięki temu, twórca systemu ma większą władzę nad sposobem w jaki pracują dane komponenty oraz może w odpowiedni sposób je konfigurować. Ponadto dzięki luźnym wiązaniom, które zapewnia nam Dependency Injection, w stosunkowo bezbolesny sposób jesteśmy wstanie podmieniać dane klasy, a nawet całe komponenty na nowe, bez wchodzenia w implementację modułów, które z nich korzystają.

W przypadku stosowania praktycznie każdego z nowoczesnych frameworków służących do łatwiejszej implementacji IoC, spotkamy się z modułami służącymi do konfiguracji zależności, fachowo zwanymi kontenerami IoC bądź kontenerami DI. W takich kontenerach rejestrujemy w jaki sposób mają przebiegać dane powiązania między konkretnymi klasami. Ponadto taki kontener zajmuje się dostarczaniem odpowiedniej implementacji interfejsu do odpowiedniego komponentu.

Głównym założeniem jest przeniesienie tworzenia obiektów oraz wiązania ich między sobą poza kod aplikacji. Obiekty tworzy i wiąże osobna biblioteka nazywana kontener DI. Kontener w celu powiązania obiektów posługuje się konfiguracją, która określa jak obiekty powinny być powiązane.