Functional
Funkcjonalności, możliwości, bezpieczeństwo
Usability
Czynnik ludzki, pomoc, dokumentacja
Reliability
Awaryjność, odzyskiwanie, przewidywalność
Performance
Czas reakcji, przepustowość, dokładność, dostępność, wykorzystanie zasobów
Supportability
Dostosowanie, utrzymanie, konfiguracja, lokalizacja
Design
Wszelkie ograniczenia projektowe (np. relacyjna baza danych)
Implementation
Narzędznia, sprzęt, zasoby, standardy
Interface
Interfejsy zewnętrznych systemów
Physical

Jak ocenić sformułowane wymagania?

•Szczegółowy/prosty **(Simple)**
•Mierzalny **(Measurable)**
•Osiągalny/atrakcyjny (**Achievable)**
•Istotny/realistyczny **(Relevant)**
•Terminowy **(Time-specific)**

Projektowanie obiektowe = określanie odpowiedzialności obiektów (klas) i ich relacji względem siebie. Wszystkie dobre praktyki, zasady, wzorce sprowadzają się do tego jak właściwie rozdzielić odpowiedzialność na zbiór obiektów (klas).

GRASP
General Responsibility Assignment Software Patterns (Principles)
Nazwa
Creator (Twórca)
Problem
Kto tworzy instancje klasy A?
Rozwiązanie: Przydziel zobowiązanie tworzenia instancji klasy A klasie B, jeżeli zachodzi jeden z warunków:
• B „zawiera” A lub agreguje A (kompozycja)
• B zapamiętuje A
• B bezpośrednio używa A
• B posiada dane inicjalizacyjne dla A

SOLID
SRP, Single Responsibility Principle - żadna klasa nie może być modyfikowana z więcej niż jednego powodu
Open-closed Principle - Składniki oprogramowania (klasy, moduły) powinny być otwarte (na rozszerzenia, adaptowalne) i zamknięte (na modyfikacje wpływające na klientów)
Liskov Substitution Principle - zasada dobrego dziedziczenia - Musi istnieć moĪliwość zastępowania typów bazowych ich podtypami (w kontekście semantycznym, poprawności działania programu, a nie syntaktycznym – program się skompilował)
Interface Segregation Principle - Klient nie powinien być zmuszany do zależności od metod których nie używa
Dependency Inversion Principle - Moduły wysokiego poziomu nie powinny zaleĪeć od modułów niskiego poziomu tylko od abstrakcji. Abstrakcje nie powinny zaleĪeć od szczegółowych rozwiązań.

Wzorce podstawowe:

-klasa abstrakcyjna i interfejs
-delegacja (Prefer Delegation over Inheritance)
Wzorce kreacyjne:
-singleton
-monostatic - Usuwa ograniczenie liczby instancji w Singletonie, pozostawia właściwość współdzielenia stanu

- **(Parametrized) Factory**
-**Factory Method:**
- Delegowanie tworzenia obiektu użytkowego do metody tworzącej, zwykle abstrakcyjnej (FactoryMethod)
- Metoda fabrykująca mimo że nie ma implementacji, może już być używana (w AnOperation)
- Podklasy dostarczają implementacji metody fabrykującej

- **Abstract Factory**
-**Prototype**
- Istnieje kilka prototypowych instancji obiektów
- Tworzenie nowych polega na kopiowaniu prototypów
- Nie ma znaczenia kto i jak wyprodukował instancje prototypów

-**Object Pool:**
- Reużywanie/ współdzielenie obiektów które są kłopotliwe w tworzeniu (np. czasochłonne)
- Metoda tworzenia/pobierania obiektu bywa parametryzowana

-**Builder**:
- Ukrywanie szczegółów kodu służącego do kreowania obiektu/obiektów

Wzorce strukturalne:

-**Facade**: uproszczony interfejs dla podsystemu z wieloma interfejsami
-**Read-only interface**: Motto: interfejs do odczytu dla wszystkich, a do zapisu tylko dla wybranych
- **Flyweight**: Efektywne zarządzanie wieloma drobnymi obiektami
Kojarzyć: **Object Pool + immutable + bardzo dużo danych** –zapamiętać przykład z wykładu Board vs Checker
-**Decorator:** Dynamicznie rozszerzanie odpowiedzialności obiektów (alternatywa dla podklas)
Kojarzyć: w bibliotekach standardowych technologii przemysłowych od lat używa się wzorca Decorator do implementacji podsystemu strumieni
- **Proxy:** substytut (zamiennik) obiektu w celu sterowania dostępem do niego
- **Adapter:** uzgadnianie niezgodnych interfejsów
- **Bridge**: **SRP + ISP + DIP** dla hierarchii obiektowej o dwóch stopniach swobody; oddzielenie abstrakcji od implementacji
- **Composite:** składanie obiektów w struktury „drzewiaste”
Kojarzyć: Tree, Expression

Wzorce czynnościowe:

-**Visitor:** Motto: definiowanie nowej operacji bez modyfikowania interfejsu
Kojarzyć: TreeVisitor z wykładu, ExpressionVisitor z biblioteki standardowej .NET
-**Null Object:** pusta implementacja zwalniająca klienta z testów if na null
-**Iterator:** sekwencyjny dostęp do obiektu zagregowanego bez ujawniania jego struktury
Kojarzyć: IEnumerator, IEnumerable
-**Interpreter(Little Language):** reprezentacja gramatyki języka i jego interpretera
Kojarzyć: kompozyt z interpreterem
-**Observer:** powiadamianie zainteresowanych o zmianie stanu, dzięki czemu nie odwołują się one do siebie wprost.
Kojarzyć: zdarzenia w C#
Przykład z życia: architektura aplikacji oparta o powiadomienia między różnymi widokami (w środku okienka – Mediator, pomiędzy okienkami – Observer)
Jeszcze inaczej –Observer ujednolica interfejs „Colleagues” Mediatora, dzięki czemu obsługuje dowolną liczbę „Colleagues”
- **Mediator:** Koordynator współpracy ściśle określonej grupy obiektów – dzięki niemu one nie odwołują się do siebie wprost (nie muszą nic o sobie wiedzieć), ale przesyłają sobie powiadomienia przez mediatora.
Kojarzyć: niby Observer bo też „powiadomienia”, ale zbiór współpracujących obiektów jest tu ściśle określony. Mediator może więc wykorzystać ten fakt do wyboru różnych technik przesyłania powiadomień (bezpośrednio, na styl observera itp.).
Druga różnica między Mediatorem a Observerem jest taka że to kolaborujące obiekty przesyłają sobie powiadomienia o zmianie swojego stanu, a stan Mediatora nie ma nic do tego. Observer ze wszyscy zainteresowani nasłuchują powiadomień o zmianie stanu obiektu obserwowanego. Nie ma więc zupełnie analogii między mediatorem a obserwowanym.
Przykład z życia: typowe okienka desktopowych technologii wytwarzania
GUI są mediatorami między konkretnymi kontrolami, które są zagregowane wewnątrz (w środku okienka –
Mediator, pomiędzy okienkami –Observer)
- **Event Aggregator:** rozwiąż problem Observera ogólniej – jeden raz dla różnych typów powiadomień

Kojarzyć: ogólniejszy Observer, „hub” komunikacyjny (Observer zaimplementowany jako „słownik list” słuchaczy indeksowany typem powiadomienia) Event Aggregator znosi najważniejsze ograniczenie Observera – klasy obserwatorów muszą tam znać klasę obserwowanego. W EventAggregatorze zarówno obserwowani jak i obserwujący muszą tylko wiedzieć gdzie szukać EventAggregatora. W efekcie klasy obserwowane i obserwujące mogą być zdefiniowane np. w niezależnych od siebie modułach (co jest niemożliwe w przypadku Observera).
Uwaga: jeden z ważniejszych wzorców dobrej architektury aplikacji
- **Memento:** Zapamiętuj i pozwalaj odzyskać stan obiektu
Uwaga: stan obiektu i stan pamiątki nie muszą być takie same. W szczególności duże obiekty mogą tworzyć małe, przyrostowe pamiątki
Kojarzyć z: Undo (i opcjonalnym Redo).
- **Chain of Responsibility:** uniknięcie związania obiektu wysyłającego żądanie z konkretnym odbiorcą w sytuacji gdy zbiór możliwych odbiorców jest dynamiczny
Kojarzyć: łańcuch niezależnych odbiorców wiadomości; przetwarzanie w skomplikowanej logice
- **Command:** kapsułkowanie żądań w postaci obiektów o jednolitym interfejsie dostępu, oddziela wywołującego (Invoker) od odbiorcy (Receiver). Wiele różnych klas Receiver może mieć różny interfejs, ale dzięki Command dla Invokera wyglądają one tak samo. Można powiedzieć, że Command wciela w życie filozofię Adaptera na szeroką skalę –każdy ConcreteCommand jest adapterem jakiegoś Receiver do jednolitego interfejsu Command.
Uwaga: prostszą alternatywą byłoby zapamiętanie funkcji zwrotnej, ale komenda może mieć szerszy interfejs niż tylko Execute (), np. Undo() - o ile sam Receiver może nie implementować Undo wprost, o tyle ConcreteCommand może użyć innego sposobu implementacji Undo (zwykle – z pomocą innego obiektu).
- **Template Method:** określ szkielet algorytmu i zrzucając odpowiedzialność za implementację szczegółów do podklasy.
Kojarzyć z: Template Method = Strategy przez dziedziczenie
Refaktoryzacja Template Method do Strategy polega na zamianie klasy abstrakcyjnej na interfejs i wyłączeniu TemplateMethod do osobnej klasy do której wstrzykiwana jest implementacja interfejsu.
- **Strategy:** kreśl szkielet algorytmu i zrzucając odpowiedzialność za implementację szczegółów do klasy, do której delegujesz żądania.
Kojarzyć z: Strategy = Template Method przez delegację
Refaktoryzacja Strategy do Template Method polega na likwidacji wstrzykiwania – metoda szablonowa staje się częścią klasy abstrakcyjnej, a konkretną funkcjonalność uzyskuje się przez dziedziczenie.
- **State:** maszyna stanowa
Uwaga: zbyt rzadko stosowana (można częściej niż się wydaje)
Przykład zastosowania: zarządzanie złożonym GUI. Formularz jest kontekstem, a klasy stanów odpowiadają aktualnie wyświetlanej zawartości. W każdym ze stanów inaczej obsługuje się zestaw zdarzeń formantów.

Wzorce architektury aplikacji:

- **automated code generation**
- **Object-relational mapping**
- **Inversion of Control (Dependency Inversion)** = zestaw technik pozwalających tworzyć struktury klas o luźniejszym powiązaniu.
Trzy kluczowe skojarzenia:
1.**Późne wiązanie** – możliwość modyfikacji kodu bez rekompilacji, wyłącznie przez rekonfigurację, „programming against interfaces”(DIP)
2.**Ułatwienie tworzenia testów jednostkowych** – zastąpienie podsystemów przez ich stuby/fake’i
3. **Uniwersalna fabryka** – tworzenie instancji dowolnych typów według zadanych wcześniej reguł
Dependency Injection = konkretny sposób realizacji IoC w językach obiektowych
… Inversion = ogólna zasada
… Injection = jej implementacja

Jeszcze inne spojrzenie na modularność:
1. **Sztywna zależność** (stable dependency) – klasyczna modularność; zależne moduły już istnieją, są stabilne, znane i przewidywalne (np. biblioteka standardowa)
2. **Miękka zależność** (volatile dependency) – modularność dla której zachodzi któryś z powodów wprowadzenia spoiny:
a. Konkretne środowisko może być konfigurowane dopiero w miejscu wdrożenia (późne wiązanie)
b. Moduły powinny być rozwijane równolegle
3. **Spoina (seam)** – miejsce, w którym decydujemy się na zależność od interfejsu zamiast od konkretnej klasy
Uwaga. O ile zastosowanie technik DI pozwala na wprowadzenie miękkich zależności w miejscach spoin, o tyle zwykle zależności do samych ram (frameworków) DI mają charakter sztywny.

Innymi słowy, nie projektuje się aplikacji w taki sposób, żeby móc miękko przekonfigurowywać je na różne implementacje ram DI. Zobaczymy jednak wzorzec Local Factory, który w praktyce na tyle izoluje użycie ramy DI w aplikacji, że jej wymiana na inny nie stanowi większego problemu.

Service Locator = schowanie singletona kontenera DI za fasadą, pozwalającą z dowolnego miejsca aplikacji na rozwiązanie zależności do usługi. Pozwala znacznie zredukować jawne zależności miedzy klasami.
Service Locator nie musi być przekazywany jako zależność, bo jako singleton, może być osiągalny z dowolnego miejsca.

Alternatywą dla SL jest Compositon Root + Local Factory (Dependency Resolver)
Composition Root = fragment kodu wykonywany zwykle na starcie aplikacji, odpowiedzialny za zdefiniowanie wszystkich zależności. W idealnej rzeczywistości, tylko w Composition Root pojawia się zależność do DI, a cała reszta aplikacji jest jej pozbawiona.
W praktyce – w aplikacji typu desktop, CR jest funkcja wywoływaną z Main lub jej okolic, w aplikacji typu web to funkcja wywoływana w potoku z handlera zdarzenia Application_Start lub jego okolic. W stosie aplikacyjnym jest to warstwa najwyższa.Każde inne miejsce na konfigurację grafu zależności to już potknięcie projektowe

Local Factory (Dependency Resolver) = fabryka odpowiedzialna za tworzenie instancji jednej lub wielu klas której funkcja fabrykująca nie ma gotowej implementacji. Zamiast tego, konkretna implementacja jest wstrzykiwana do fabryki z poziomu Compositon Root.
Z kolei fabryka jest jedynym legalnym z punktu widzenia API danego podsystemu sposobem wytwarzania instancji obiektów tej jednej lub kilku klas.
Obrazowo można powiedzieć, że w stosie aplikacyjnym cała logika od miejsca określenia fabryki w górę korzysta z tej fabryki do tworzenia instancji, ale konkretna implementacja jest „wstrzyknięta do fabryki" z samego wierzchu stosu aplikacyjnego, z poziomu CR. Na Local Factory można patrzeć jak na lokalnego Service Locatora, który tym się różni od swojego „dużego" brata że rozwiązuje problem lokalnie, na potrzeby jednej/kilku klas z jakiegoś konkretnego podsystemu i jest częścią dziedziny (API) tego podsystemu, a nie częścią obcego API, wymuszającego zewnętrzne zależności (jak SL). W prawdziwej aplikacji takich LocalFactories jest więc wiele – występują one wszędzie tam, gdzie pojawiają się miękkie zależności w ramach podsystemów.

Repository – dodatkowa warstwa abstrakcji na obiektową warstwę dostępu do danych. Zwykle Repository kontroluje dostęp do jednej „kategorii" danych (np. jednej tabeli z bazy danych)
Unit of Work – kompozyt wielu repozytoriów – zarządza ich czasem życia i pozwala na dostęp do nich z jednego miejsca. Dodatkowo bierze na siebie m.in. zarządzanie transakcjami.

Wzorce warstwy interfejsu użytkownika mają na celu zapewnienie możliwości łatwiejszego utrzymania kodu oraz podniesienie wiarygodności – osiągają to oddzielając logikę przetwarzania od logiki prezentacji. Dzięki lepszej izolacji, możliwe jest testowanie obu warstw niezależnie za pomocą testów zautomatyzowanych, nie wymagających interakcji użytkownika.

Model-View-Controller (MVC)
Wzorzec architektury interfejsu użytkownika zarezerwowany dla aplikacji typu Web Application.
• Interakcja użytkownika Controller → Model + View
• Kontroler i widoki są zwykle połączone relacją 1 - wiele (1 kontroler obsługuje wiele widoków)
• Akcje użytkownika trafiają do właściwych kontrolerów; kontrolery są tworzone przez środowisko uruchomieniowe (tu: serwer aplikacji) na podstawie parametrów żądań HTTP
• Kontroler na podstawie akcji użytkownika tworzy model danych, ustala widok do wyrenderowania i do widoku przekazuje model

Model-View-Presenter
Model-View-ViewModel

Test-Driven Development(TDD) – rozwijanie oprogramowania sterowane testami
System Under Test (SUT) – klasa użytkowa, która jest podmiotem testu jednostkowego.
Collaborators – klasy usług pomocniczych, z których korzysta klasa SUT, ale które nie są podmiotami testów. Myślimy o architekturze, w której usługi pomocnicze są wstrzykiwane do klas użytkowych.

Test Double (Dubler) – klasa implementująca usługę, zastępująca prawdziwą implementację podczas testowania.
Interfejs „udawanej" usługi można zaimplementować na różne sposoby:
• **Dummy** - implementacja, która w ogóle nie jest wykorzystywana, a jej jedynym celem jest wypełnienie listy usług wstrzykiwanych do klasy SUT. Poszczególne metody implementacji typu Dummy mogą nawet wyrzucać wyjątki typu NotImplementedException, bo klasa SUT w ogóle tej usługi nie będzie wykorzystywać w danym teście.

• **Stub** - implementacja, która niekoniecznie działa zgodnie ze specyfikacją funkcjonalną; poszczególne metody zwracają wyniki spreparowane pod kątem konkretnego testu/testów. Przydatne do testów na konkretnych danych i konkretnej sekwencji wywołań metod.
• **Fake** – implementacja, która faktycznie działa i nawet robi to co powinna; co prawda sposób jej implementacji wyklucza jej produkcyjne wykorzystanie, ale równocześnie pozbawiona skutków ubocznych rzeczywistej implementacji. Przykład to implementacja repozytorium, która utrwala obiekty w pamięci operacyjnej zamiast w bazie danych. Przydatne do testów dowolnych danych i dowolnej sekwencji wywołań metod, na których miałaby pracować rzeczywista implementacja.
• **Mock** – Mocking Object, typ zastępczy, gotowa rama aplikacyjna dostarczająca implementacji usług pod kątem testowania BDD

Behavior Driven Development(BDD) – TDD, w którym testuje się zachowanie implementacji SUT i usług pomocniczych a nie ich stanu. Testowanie zachowania polega na sprawdzaniu:
• Czy SUT wywołuje właściwe metody ze swoich collaborators
• Czy wywołuje je z właściwymi parametrami
• Czy wywołuje je właściwą liczbę razy
• Czy wywołuje je we właściwej kolejności
Testowanie zachowania jest ogólniejsze od testowania stanu, w wielu wypadkach pozwala unikać powtarzania sekwencji Act-Assert-Act-Assert, które przy testowaniu stanu są niezbędne do rozpoznawania sekwencji stanów.

Design by Contract – technika projektowania obiektowego, w której częścią interfejsu metod i klas są zobowiązania dotyczące stanu w określonych momentach obliczeń:
• **precondition**(warunek wejścia) – stan w chwili rozpoczęcia wykonywania się metody
• **postcondition** (warunek wyjścia) – stan w chwili zakończenia wykonywania się metody
• **invariant**(niezmiennik) – stan w określonym momencie wykonywania się metody
Warunki mają zwykle postać predykatów (formuł logicznych typu boole'owskiego) wyrażonych w języku logiki pierwszego rzędu.
OCL(Object Constraint Language) – uniwersalny formalizm zaprojektowany do wyrażania kontraktów DbC w językach obiektowych, stosunkowo mało rozpowszechniony.

Aplikacje rozległe (ang. Enterprise applications) – to wielomodułowe systemy informatyczne, często rozwijane przez lata lub powstające w wyniku połączenia kilku niezależnych elementów, zbudowanych w różnych technologiach i w oparciu o różne konstrukcje architektury

Single sign-on(pojedyncze logowanie) – to właściwość aplikacji rozległych, w których dostęp do tych części poszczególnych modułów które wymagają autentykacji i autoryzacji, możliwy jest po jednokrotnym potwierdzeniu tożsamości użytkownika.
Z uwagi na różne implementacje realizujące ten sam cel, można mówić o wzorcu dla aplikacji rozległych.

Claim(stwierdzenie/oświadczenie) – informacja o Kimś wydane przez jakiegoś Wystawcę. Claim powinien być „podpisany" tzn. nie powinno być wątpliwości że wydał go Wystawca. Zwykle nie da się nijak inaczej stwierdzić czy claim jest prawdziwy czy nie, ale ufamy wystawy wobec czego akceptujemy informację.

Message-oriented architecture – model architektury zorientowany na komunikaty wymieniane między modułami. Ciekawie zaczyna robić się wtedy, kiedy mówimy o architekturze systemów ponieważ wtedy do wymiany komunikatów niezbędny staje się zewnątrzprocesowy pośrednik – Message-oriented Middleware(MOM).

Współcześnie preferuje się specyfikacje typu Protocol Driven, specyfikacje typu API Driven są uznawane za przestarzałe z uwagi na problemy z interoperacyjnością.

Enterprise Service Bus – usługa zewnątrzprocesowa typu MOM, dostarczająca narzędzi komunikacyjnych w rozległym środowisku aplikacyjnym

Command-Query Responsibilibty Separation – filozofia budowy systemu transakcyjnego, w którym rola odczytu i zapisu (modyfikacji) jest wyraĨnie koncepcyjnie rozdzielona.