☐	Ogranicz poziomy wcięć w metodach	Złożone warunki if często **bywają trudne do zrozumienia.** Dlatego bloki w instrukcjach if, else, while **powinny być jak najprostsze.** Najlepiej jakby zawierały tylko jeden wiersz. Na pewno warto wystrzegać się poziomu wcięcia większego niż dwa. Dzięki temu kod jest o wiele czytelniejszy. Dobrym pomysłem jest **również stosowanie tzw. warunku "guard",** który pozwala na szybkie zakończenie działania na początku metody, gdy pewne warunki nie zostaną spełnione. Dzięki temu eliminujemy głęboko zagnieżdżone struktury if. To upraszcza kod i ułatwia jego utrzymanie.				
☐	Twórz metody bez efektów ubocznych	Ważne, aby metoda robiła dokładnie to, co sugeruje jej nazwa. Jeśli metoda wykonuje dodatkowe operacje, o których **jej nazwa może nie wskazywać**, osoba korzystająca z niej nie będzie tego świadoma. To **zwiększa trudność zrozumienia kodu**, ponieważ trzeba będzie zagłębiać się w metodę i kolejne wywołania, aby pojąć, co faktycznie się dzieje. Takie podejście **będzie powodowało błędy**. Najpierw zastanów się, czy można to zrefaktoryzować. Jeśli nie, zmień nazwę metody tak, aby **jasno komunikowała, co robi**, bez konieczności zaglądania w jej wnętrze.				
	Obiektowość					
☐	Pisz kod obiektowy	Kod obiektowy ukrywa dane za odpowiednią abstrakcją, **udostępniając metody operujące** na tych danych, co **zwiększa czytelność i zrozumiałość** kodu. W przeciwieństwie do struktur danych, które eksponują swoje pola i **oferują jedynie gettery i settery**. Struktury danych mają swoje zalety i bywają użyteczne w specyficznych przypadkach, takich jak klasy DTO czy entity. Ale jednak warto pamiętać o korzyściach obiektowości. Dlatego, w większości przypadków, **zaleca się stosowanie podejścia obiektowego**, które sprzyja lepszej organizacji i **łatwiejszemu zrozumieniu kodu**.				
☐	Hermetyzuj dane, ukrywaj szczegóły za odpowiednią abstrakcją	Hermetyzacja **to nie tylko ustawianie pól jako prywatne** i ograniczanie liczby setterów, ale przede wszystkim **ukrywanie szczegółów implementacji** obiektu. Zamiast używać setterów, powinniśmy operacje umieszczać za odpowiednią abstrakcją, **by nie odkrywać wewnętrznych pól**. Logika związana z obiektem powinna być zawarta w danym obiekcie, a **nie wyciekać do serwisów** czy klas pomocniczych. W serwisach powinna znajdować się **jedynie logika procesowa**, natomiast szczegóły działania obiektów **muszą pozostać ukryte.**				
☐	Twórz obiekty niezmienne	Obiekty niezmienne to obiekty, których **cały stan inicjujemy** w konstruktorze i później ten **stan nie może być później zmieniony.** Możemy na nich wykonywać operacje, ale każda operacja tworzy nowy obiekt, który również pozostaje niezmienny.				
☐	Twórz Value Objecty	Value Object to obiekt **przechowujący jakąś wartość**, grupuje dane i reguły w jednym miejscu. Taki obiekt **nadaje sens biznesowy** wartości. np. zmienna Long czy String nie daje nam sensu biznesowego. Jeśli natomiast **opakujemy tą zmienną w Value Object** np. Distance. To wiemy już co kryje się pod tą zmienną. Dodatkowo możemy umieścić tam walidację jak i potrzebne metody. Tworzenie Value Objectów ułatwia testowanie, **poprawia czytelność kodu** i wprowadza bardziej zrozumiały język biznesowy. Choć nie zawsze są potrzebne, szczególnie w prostych projektach CRUD, warto je stosować, gdy zauważymy korzyści.				
☐	Staraj się przestrzegać Prawa Demeter	Prawo Demeter mówi, że **moduł nie powinien znać szczegółów** wewnętrznych obiektów, **którymi manipuluje.** Klasa może komunikować się tylko z obiektami przekazanymi jako parametry, polami klasy, obiektami globalnymi lub stworzonymi w metodzie. Prawo jest łamane, gdy w jednej linii kodu **pojawia się więcej niż jedna kropka**, co oznacza zbyt głębokie zależności. Choć delegowanie zadań między obiektami jest korzystne, warto zachować równowagę, by nie tworzyć klas, które **jedynie przekazują zadania** bez żadnej logiki. To prawo nie dotyczy struktur danych.				
	SOLID					
☐	Single Responsibility Principle	Zasada Pojedynczej Odpowiedzialności mówi, że każdy moduł powinien mieć jedną i tylko jedną przyczynę zmian. **Jedna odpowiedzialność to jedna przyczyna zmian**, a tą przyczyną zmian jest tzw. aktor, czyli grupa osób, które chcą zmienić system w określony sposób. SRP można więc opisać jako: "Każdy moduł powinien odpowiadać przed jednym i tylko jednym aktorem." **Moduł to spójny zbiór metod i struktur danych, które zmieniają się jednocześnie.** Dlatego można też przedstawić inną wersję tej zasady: "Grupuj te rzeczy, które zmieniają się w tym samym czasie z tego samego powodu, i **oddzielaj te, które zmieniają się w różnych momentach z różnych powodów."**				
☐	Open Closed Principle	Zasada Otwarte Zamknięte mówi, że element oprogramowania (klasa, moduł, metoda) **powinien być otwarty na rozbudowę, ale zamknięty na modyfikacje**. Oznacza to, że można **rozszerzać jego funkcjonalność** bez zmieniania istniejącego kodu. Aby pisać kod zgodny z tą zasadą, warto stosować abstrakcje, polimorfizm, hermetyzację, zasady SOLID, GRASP czy wzorce projektowe. Kluczowym celem tej zasady jest umożliwienie łatwej rozbudowy oprogramowania, **minimalizując konieczność wprowadzania zmian** w już istniejącym kodzie.				