## **Sprawozdanie z Analizy Projektu**

### **1. Naruszenie Zasady Odwracania Zależności (DIP)**

**Analiza:**

Zasada odwracania zależności (Dependency Inversion Principle) stanowi, że moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych, lecz oba powinny zależeć od abstrakcji. W analizowanym projekcie klasa Nadzorca (moduł wysokopoziomowy, orkiestrujący proces tworzenia sporu) zależy bezpośrednio od konkretnej implementacji ConcreteDisputeBuilder, a nie od interfejsu DisputeBuilder.

**Wnioski i rekomendacje:**

To naruszenie powoduje silne sprzężenie (high coupling) między klasą Nadzorca a konkretną implementacją budowniczego. Utrudnia to testowanie oraz ewentualną podmianę implementacji budowniczego w przyszłości (np. na AdvancedDisputeBuilder) bez modyfikacji kodu klasy Nadzorca.

**Sposób poprawy:**

Należy zmienić typ pola builder w klasie Nadzorca z ConcreteDisputeBuilder na interfejs DisputeBuilder. Obiekt konkretnej implementacji powinien być wstrzykiwany z zewnątrz, np. przez konstruktor.

**Fragment diagramu UML ilustrujący poprawę:**

A diagram of a server

AI-generated content may be incorrect.

### **2. Naruszenie Zasady Pojedynczej Odpowiedzialności (SRP)**

**Analiza:**

Zasada pojedynczej odpowiedzialności (Single Responsibility Principle) mówi, że klasa powinna mieć tylko jeden powód do zmiany. W diagramie klasa Item posiada metody takie jak buy, edit oraz delete. Odpowiedzialność za sam proces zakupu (buy) nie powinna leżeć w gestii obiektu Item. Item jest obiektem domenowym, którego odpowiedzialnością powinno być przechowywanie informacji o sobie samym (tytuł, cena, opis). Proces transakcyjny to osobna odpowiedzialność.

**Wnioski i rekomendacje:**

Umieszczenie logiki biznesowej zakupu w klasie Item prowadzi do jej "puchnięcia" i mieszania odpowiedzialności. Zmiana w procesie zakupu (np. dodanie walidacji kupującego) wymuszałaby modyfikację klasy Item, co jest niezgodne z SRP.

**Sposób poprawy:**

Logika zakupu przedmiotu powinna zostać przeniesiona do dedykowanej klasy serwisowej lub, jak to częściowo zrobiono w projekcie, do fasady (MarketplaceFacade.buyItem). Metoda buy powinna zostać usunięta z klasy Item.

**Fragment diagramu UML ilustrujący poprawę:**A white rectangular object with black dots

AI-generated content may be incorrect.

### **3. Zastosowanie antywzorca "Duplikacja Kodu" (Duplicate Code)**

**Analiza:**

W projekcie występuje rażąca duplikacja kodu i struktur. Zdefiniowano dwie niemal równoległe hierarchie dziedziczenia dla użytkowników:

1. Abstrakcyjna klasa User i dziedziczące po niej Buyer(User), Seller(User), Admin(User).
2. Klasa User2 i dziedziczące po niej Buyer1(User), Seller1(User), Admin1(user).

Obie struktury modelują ten sam koncept (użytkownika systemu), posiadają zduplikowane atrybuty (np. email, name) oraz metody (register, login). Jest to klasyczny przykład antywzorca **Duplicate Code**.

**Wnioski i rekomendacje:**

Taka duplikacja jest źródłem błędów, utrudnia utrzymanie i rozwój systemu. Każda zmiana w logice użytkownika musi być wprowadzana w dwóch miejscach.

**Sposób poprawy:**

Należy scalić obie hierarchie w jedną, spójną strukturę. Proponuje się stworzenie jednej abstrakcyjnej klasy bazowej User z podstawowymi danymi i metodami autoryzacji, oraz konkretnych klas Buyer, Seller, Admin dziedziczących po niej i implementujących specyficzne dla siebie zachowania.

**Fragment diagramu UML ilustrujący poprawę (uproszczona hierarchia):**

A diagram of a company

AI-generated content may be incorrect.

### **4. Zachowanie Zasady Otwarte-Zamknięte (OCP)**

**Analiza:**

Zasada otwarte-zamknięte (Open/Closed Principle) mówi, że byty programistyczne (klasy, moduły) powinny być otwarte na rozszerzenia, ale zamknięte na modyfikacje. W analizowanym projekcie zasada ta jest **poprawnie zaimplementowana** za pomocą wzorca projektowego **Fabryka Abstrakcyjna** (UserFactory).

**Dowód:**

System wykorzystuje interfejs UserFactory oraz jego konkretne implementacje (BuyerFactory, SellerFactory, AdminFactory). Dzięki temu, jeśli w przyszłości zajdzie potrzeba dodania nowego typu użytkownika (np. Moderator), wystarczy stworzyć nową klasę Moderator oraz implementację ModeratorFactory. Nie będzie wymagana żadna modyfikacja istniejącego kodu klienckiego (np. w MarketplaceFacade), który korzysta z interfejsu UserFactory. System jest więc otwarty na dodawanie nowych typów użytkowników, a jednocześnie zamknięty na modyfikacje w zakresie ich tworzenia. Podobnie działa wzorzec **Obserwator** (Subject/Observer), pozwalając na dodawanie nowych obserwatorów bez zmiany kodu obserwowanego obiektu (Item).

### **5. Zastosowanie pryncypium GRASP "Kontroler" (Controller)**

**Analiza:**

Pryncypium GRASP **Kontroler** (Controller) odpowiada na pytanie, który obiekt powinien odbierać i obsługiwać zdarzenia systemowe pochodzące od interfejsu użytkownika. Zadaniem kontrolera jest delegowanie zadań do odpowiednich obiektów w warstwie domenowej. W analizowanym projekcie rola ta jest **poprawnie pełniona** przez klasę MarketplaceFacade.

**Dowód:**

Klasa MarketplaceFacade stanowi jednolity punkt wejścia do głównych funkcjonalności systemu, takich jak register(), addItem(), buyItem(). Odbiera ona żądania (które w realnym systemie pochodziłyby z warstwy UI) i koordynuje pracę innych obiektów – UserFactory do tworzenia użytkowników, ItemRepository (domyślnie) do zarządzania przedmiotami, czy tworzenia Transaction. Takie podejście izoluje logikę biznesową od interfejsu użytkownika, zmniejsza sprzężenie między tymi warstwami i promuje wysoką spójność (high cohesion) w obiektach domenowych, które wykonują specyficzne zadania.