Single responsibility – clasa are o singura responsabilitate

Interface Segregation – mai multe interfete si fiecare interfata cu metoda sa

Open Closed – putem adauga o clasa nou / putem modifica codul

Liskov – clasa copil mosteneste clasa parinte si poate fi mai dezvoltata ca parintele

Dependency Inversion - eliminam dependentele / abstractizam sistemul

În acest exemplu, avem interfețele ITakeOff și ILand, care definesc metodele TakeOff() și Land(). Clasele PassengerFlight și CargoFlight implementează aceste interfețe și definesc logica specifică pentru decolare și aterizare în funcție de tipul de zbor.

Clasa FlightManager respectă principiul SRP, deoarece are o singură responsabilitate - gestionarea operațiunilor de decolare și aterizare. Prin intermediul constructorului, managerul primește instanțe de zboruri specifice și le utilize

În acest exemplu, avem clasa abstractă **Ticket** care servește drept bază pentru bilet. Avem două clase derivate: **ETicket** pentru biletul electronic și **PhysicalTicket** pentru biletul fizic. Aceste clase implementează metoda **PrintTicket()** specifică tipului de bilet.

Clasa **TicketManager** respectă principiul OCP prin faptul că este deschisă pentru extensie prin adăugarea de noi tipuri de bilete, dar închisă pentru modificare. Putem adăuga noi tipuri de bilete derivate ale clasei **Ticket** fără a modifica codul existent al managerului de bilete.

Prin respectarea Liskov Substitution Principle, putem înlocui orice instanță de **Ticket** cu oricare dintre clasele derivate (**ETicket**, **PhysicalTicket**) fără a afecta funcționalitatea corectă a programului.

Astfel, OCP ne permite să extindem funcționalitatea adăugând noi tipuri de bilete, iar LSP ne asigură că putem utiliza diferite tipuri de bilete în mod transparent în cadrul managerului de bilete.

În acest exemplu, avem două implementări ale interfeței **ISecurityService**: **CustomsSecurityService** pentru controlul vamal și **PassengerSecurityService** pentru controlul pasagerilor. Ambele implementări au metoda **Scan()** care conține logica specifică pentru scanarea bagajelor sau a pasagerilor.

Clasa **AirportSecurity** utilizează o dependență prin intermediul constructorului său. Prin injecția unei implementări de **ISecurityService**, clasa **AirportSecurity** devine independentă de detaliile specifice ale serviciului de securitate și respectă principiul Dependency Inversion.

În metoda **Main**, cream instanțe ale serviciilor de securitate și le injectăm în instanțele **AirportSecurity** pentru a efectua scanarea în controlul vamal și în controlul pasagerilor. Acest lucru demonstrează flexibilitatea și extensibilit