Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Lucrare de laborator nr.4**

*Disciplina: Tehnici și mecanisme de proiectare software*

**Tema:** Principiile SOLID



**Studentul: gr. TI-206 Raileanu Daniel**

**Coordonator: asist. univ. Buldumac Oleg**

**Chișinău 2023**

**Sarcina:** Să se implementeze la nivel de cod principiile SOLID.

**Principiile SOLID** sunt principiile de proiectare care ne permit să gestionăm cele mai multe probleme de proiectare a software-ului. Acestea reprezintă un set de recomandări care ne ajută în a evita conceperea greșită a programului. Principiile SOLID reprezintă cele 5 concepte de bază, pe care ar trebui să le îndeplinească orice cod care este scris în spiritul programării obiectate pe obiecte (POO). Ele sunt principii generale, care pot fi implementate în orice limbaj de programare.

(SRP) Single Responsibility Principle

**(SRP) Single Responsibility Principle**

Fiecare clasă ar trebui să aibă o singură responsabilitate. Orice modul sau clasă trebuie să încapsuleze o singură funcționalitate. O clasă sau funcție, nu ar trebui să rezolve sau sa trateze mai mult decât un singur scop deoarece aceasta ar introduce cuplarea între cele două funcționalități.

**(OCP) Open closed Principle**

Acest principiu exprimă faptul că fiecare clasă ar trebui să fie deschisă pentru extindere și închisă pentru modificări. Asta inseamna ca (clasele, functiile, modulele, etc) pot sa-si schimbe comportamentul fara a modifica codul initial (dar prin adaugare unei clase noi).

**(LSP) Liskov substitution Principle**

Orice clasă derivată poate înlocui clasa de bază și codul să funcționeze corect în continuare. In cazul exemplului meu, 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.

(ISP) Interface Segregation Principle

**(ISP) Interface Segregation Principle**

In loc să avem o interfață cu foarte multe metode, din care doar câteva sunt folosite, mai bine avem mai multe interfețe mai mici și le folosim doar pe cele necesare.

Acest principiu expune ideea că niciun client nu are voie să fie forțat să depindă de metodele pe care nu le folosește.

(DIP) Dependency Inversion Principle

(DIP) Dependency Inversion Principle

**(DIP) Dependency Inversion Principle**

Dacă o clasă are o dependență, această dependență ar trebui să fie o interfață, nu o clasă

concretă – acest principiu încurajează decuplarea claselor și e necesar pentru testarea unitară

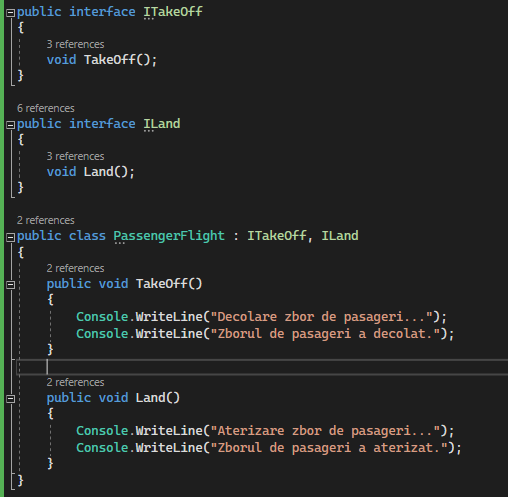


Figura 1 – Implementarea principiilor SRP + ISP



Figura 2 – Crearea/primirea instanței

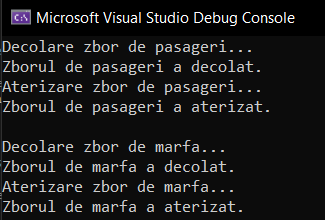


Figura 3 – Rezultatele in urma executarii programului

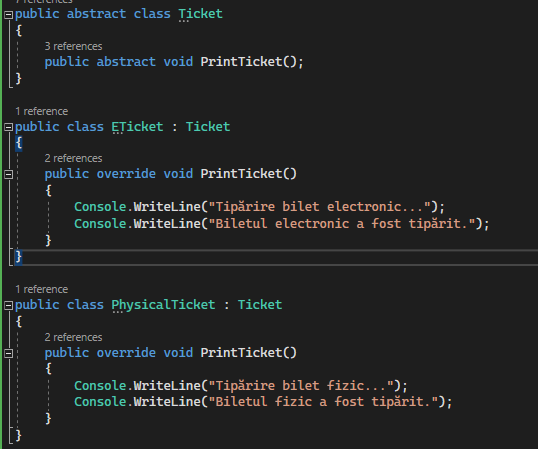


Figura 4 – Implementarea principiilor OCP + LSP

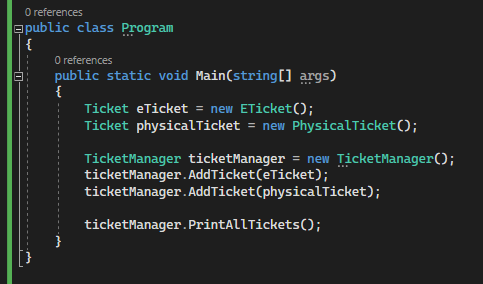


Figura 5 – Crearea/primirea instanței

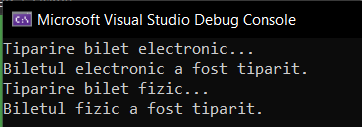


Figura 6 – Rezultatele in urma executarii programului

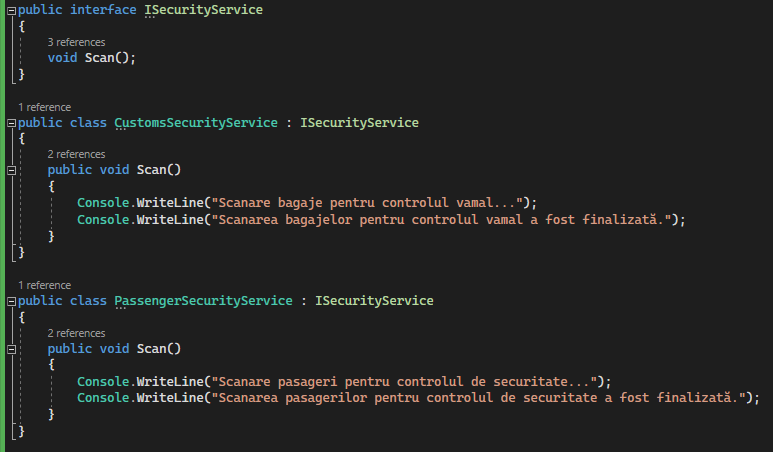


Figura 7 – Implementarea principiului Dependency Inversion



Figura 8 – Crearea/primirea instanței

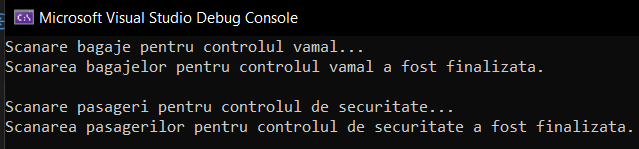


Figura 9 – Rezultatele in urma executarii programului

**Concluzie**

În urma efectuării lucrării de laborator am învățat despre principiile SOLID scopul lor în programare. Principiile SOLID sunt principiile de proiectare care ne permit să gestionăm cele mai multe probleme de proiectare a software-ului. Acestea reprezintă un set de recomandări care ne ajută să evităm conceperea greșită a programului. Prin respectarea acestor principii, putem crea un cod robust, extensibil și ușor de întreținut în cadrul programării obiectate pe obiecte (POO). În continuare, vom analiza câteva exemple și vom vedea cum aceste principii sunt aplicate.

În primul exemplu, avem interfețele ITakeOff și ILand, care definesc metodele TakeOff() și Land(). Aceste interfețe sunt implementate de clasele PassengerFlight și CargoFlight, care definesc logica specifică pentru decolare și aterizare în funcție de tipul de zbor. Prin separarea responsabilităților în clase separate și prin respectarea principiului SRP, obținem un cod modular și ușor de întreținut.

În al doilea exemplu, avem clasa abstractă Ticket, care servește drept bază pentru bilet, și două clase derivate: ETicket și PhysicalTicket. Aceste clase implementează metoda PrintTicket() specifică tipului de bilet. Prin respectarea principiului OCP, putem adăuga noi tipuri de bilete derivate fără a modifica codul existent al managerului de bilete. De asemenea, respectarea principiului LSP ne asigură că putem utiliza diferite tipuri de bilete în mod transparent în cadrul managerului de bilete.

În al treilea exemplu, avem două implementări ale interfeței ISecurityService: CustomsSecurityService și PassengerSecurityService, care au metoda Scan() cu logica specifică pentru scanarea bagajelor sau a pasagerilor. Clasa AirportSecurity utilizează o dependență prin injecția unei implementări de ISecurityService în constructorul său, respectând principiul Dependency Inversion. Acest lucru ne oferă flexibilitate și extensibilitate în ceea ce privește serviciile de securitate utilizate în controlul vamal și în controlul pasagerilor.

În concluzie, prin aplicarea principiilor SOLID, putem obține un cod bine structurat, modular și ușor de întreținut. Aceste principii ne ajută să evităm dependențe strânse între componente, să separăm responsabilitățile și să creăm un cod flexibil și extensibil. Prin respectarea acestor principii, putem dezvolta software de calitate și ușor de gestionat în orice limbaj de programare.

**Bibliografie**

https://github.com/442god/TMPS