Ministerul Educaţiei, Culturii și Cercetării

Universitatea Tehnică a Moldovei

**Proiect de curs**

Disciplina: *Tehnici si mecanisme de testare softwere*



A elaborat: Nica Ion

Grupa: TI-203

Semnatura:\_\_\_\_\_\_\_\_\_\_\_\_

A verificat: Gaidau Mihai

Semnatura:\_\_\_\_\_\_\_\_\_\_\_\_

Chisinau, 2023

[**INTRODUCERE** 3](#_Toc137039919)

[**1.** **NOȚIUNI TEORETICE** 4](#_Toc137039920)

[1.1 SOLID 4](#_Toc137039921)

[1.2 DESIGN PATTERNS 5](#_Toc137039922)

[**2** **DESCRIEREA PRINCIPIULUI DE LUCRU AL TEMATICII ALESE** 7](#_Toc137039923)

[2.1 SOLID 7](#_Toc137039924)

[2.1.1 Principiul responsabilității unice 7](#_Toc137039925)

[2.1.2 Principiul deschiderii-închiderii 7](#_Toc137039926)

[2.1.3 Principiul substituției Liskov 8](#_Toc137039927)

[2.1.4 Principiul inversării dependențelor 8](#_Toc137039928)

[2.1.5 Principiul înlocuirii dependențelor 8](#_Toc137039929)

[2.2 DESIGN PATTERNS 9](#_Toc137039930)

[2.2.1 Factory Method 9](#_Toc137039931)

[2.2.2 Builder 9](#_Toc137039932)

[2.2.3 Adapter 10](#_Toc137039933)

[2.2.4 Observer 11](#_Toc137039934)

[**3** **MODELAREA SISTEMULUI UML** 12](#_Toc137039935)

[**CONCLUZII** 14](#_Toc137039936)

[**BIBLIOGRAFIE:** 15](#_Toc137039937)

# **INTRODUCERE**

Acest proiect este un exemplu practic de implementare a unor modele de design în JavaScript. Scopul este de a demonstra cum putem utiliza aceste modele pentru a obține un cod mai modular, mai ușor de înțeles și de întreținut.

Aplicația se concentrează pe gestionarea unui Depozit de mașini. Aceasta ne permite să adăugăm mașini noi în depozit, să căutăm mașini după anumite criterii și să le ștergem. De asemenea, putem afișa lista de mașini disponibile într-o interfață grafică.

Pentru a obține o structură coerentă și un cod extensibil, am implementat mai multe modele de design. Aceste modele include: Factory Method, Builder, Adapter și Observer. Fiecare model de design are un scop specific și contribuie la modularitatea și flexibilitatea aplicației noastre.

Pe lângă modelele de design, am implementat și funcționalități de interacțiune cu utilizatorul, precum ascultători de evenimente și funcții de utilitate. Acestea ne permit să adăugăm mașini prin intermediul unei interfețe grafice, să căutăm mașini in depozit și să afișăm lista de mașini disponibile.

Proiectul este conceput pentru a fi interactiv și ușor de utilizat. Puteți interacționa cu depozitul de mașini prin intermediul interfeței grafice, în funcție de preferințele dumneavoastră. Adăugați mașini noi, căutați mașini după marca sau model, ștergeți mașini și explorați funcționalitățile oferite de aplicație.

Acest proiect vă va oferi o perspectivă mai bună asupra modului în care putem utiliza modelele de design în dezvoltarea de aplicații și cum acestea contribuie la obținerea unui cod mai modular și mai ușor de gestionat.

# **NOȚIUNI TEORETICE**

## SOLID

SOLID este un acronim pentru cele cinci principii principale de proiectare în programarea orientată pe obiecte - Responsabilitate unică, Deschis-închis, substituție Liskov, Segregarea interfeței și inversarea dependenței. Tradus în rusă: principiile responsabilității unice, deschis/închis, înlocuirea Barbara Liskov, separarea interfeței și inversarea dependenței)

Acronimul SOLID a fost inventat de Robert Martin, autorul mai multor cărți binecunoscute în comunitatea dezvoltatorilor. Aceste principii fac posibilă construirea de produse software scalabile și care pot fi întreținute, bazate pe OOP, cu o logică de afaceri ușor de înțeles.

Decriptare:

* Responsabilitate unică - principiul răspunderii unice
* Deschis-închis - principiul deschiderii/apropierii
* Substituția Liskov - principiul substituției Barbara Liskov
* Segregarea interfeței - principiul separării interfeței
* Inversarea dependenței - principiul inversării dependenței

Principiul responsabilității unice / SRP înseamnă că fiecare obiect trebuie să aibă o singură responsabilitate și această responsabilitate trebuie să fie complet încapsulată într-o clasă. Toate serviciile sale ar trebui să aibă ca scop exclusiv asigurarea acestei obligații

Principiul deschis-închis / OCP declară că entitățile software (clase, module, funcții etc.) ar trebui să fie deschise pentru extindere, dar închise pentru modificare. Aceasta înseamnă că aceste entități își pot schimba comportamentul fără a-și schimba codul sursă.

Principiul de substituție al lui Barbara Liskov / LSP așa cum a afirmat Robert Martin: „funcțiile care folosesc un tip de bază trebuie să poată folosi subtipuri ale tipului de bază fără să știe acest lucru”.

Principiul segregării interfeței / ISP așa cum a afirmat Robert Martin: „Clienții nu ar trebui să depindă de metodele pe care nu le folosesc”. Principiul separării interfețelor spune că interfețele care sunt prea „groase” ar trebui împărțite în altele mai mici și mai specifice, astfel încât clienții interfețelor mici să cunoască numai metodele de care au nevoie în munca lor. Ca urmare, atunci când se schimbă o metodă de interfață, clienții care nu folosesc această metodă nu ar trebui să se schimbe.

Principiul inversării dependenței / DIP - modulele de nivel superior nu ar trebui să depindă de modulele de nivel inferior, iar ambele tipuri de module ar trebui să depindă de abstracții; abstracțiile în sine nu ar trebui să depindă de detalii, dar detaliile ar trebui să depindă de abstracții.

## DESIGN PATTERNS

În ingineria software , un model de proiectare software este o soluție generală, reutilizabilă , la o problemă care apare frecvent într-un context dat în proiectarea software . Nu este un design finit care poate fi transformat direct în cod sursă sau mașină . Mai degrabă, este o descriere sau un șablon pentru a rezolva o problemă care poate fi folosită în multe situații diferite. Modelele de proiectare sunt cele mai bune practici formalizate pe care programatorul le poate folosi pentru a rezolva probleme comune atunci când proiectează o aplicație sau un sistem.

Modelele de proiectare orientate pe obiecte arată de obicei relații și interacțiuni între clase sau obiecte , fără a specifica clasele finale de aplicație sau obiectele care sunt implicate. Modelele care implică o stare mutabilă pot fi nepotrivite pentru limbaje de programare funcționale . Unele modele pot deveni inutile în limbaje care au suport încorporat pentru rezolvarea problemei pe care încearcă să o rezolve, iar modelele orientate pe obiecte nu sunt neapărat potrivite pentru limbaje neorientate pe obiecte.

Modelele de proiectare pot fi privite ca o abordare structurată a programării computerelor intermediare între nivelurile unei paradigme de programare și un algoritm concret  
Modelele de design au două utilizări principale în dezvoltarea de software.

Modelele de design oferă o terminologie standard și sunt specifice unui anumit scenariu. De exemplu, un model de design singleton înseamnă utilizarea unui singur obiect, astfel încât toți dezvoltatorii familiarizați cu modelul de design unic vor folosi un singur obiect și își pot spune reciproc că programul urmează un model singleton.

Modelele de design au evoluat pe o perioadă lungă de timp și oferă cele mai bune soluții la anumite probleme cu care se confruntă în timpul dezvoltării software. Învățarea acestor modele îi ajută pe dezvoltatorii neexperimentați să învețe designul software într-un mod ușor și mai rapid.  
Conform cărții de referință pentru modele de design Design Patterns - Elements of Reusable Object-Oriented Software , există 23 de modele de design care pot fi clasificate în trei categorii: modele creaționale, structurale și comportamentale.

Pattern-urile de proiectare pot fi clasificate în trei categorii principale:

* Creational Patterns (Pattern-uri de Creare): Aceste pattern-uri se referă la modul de creare a obiectelor sau a claselor. Ele ajută la abstractizarea procesului de instantiere a obiectelor. Exemple de astfel de pattern-uri includ Singleton, Factory, Abstract Factory, Builder și Prototype.
* Structural Patterns (Pattern-uri Structurale): Aceste pattern-uri se referă la modul în care clasele și obiectele sunt compuse pentru a forma structuri mai mari. Ele asigură că, atunci când o parte a unui sistem se schimbă, întregul sistem nu trebuie să fie restructurat. Exemple de astfel de pattern-uri includ Adapter, Bridge, Composite, Decorator, Facade, Flyweight și Proxy.
* Behavioral Patterns (Pattern-uri de Comportament): Aceste pattern-uri se referă la algoritmi și la asignarea responsabilităților între obiecte. Ele nu descriu doar modele de obiecte sau de clase, ci și modele de comunicare între obiecte. Exemple de astfel de pattern-uri includ Observer, Mediator, Strategy, Chain of Responsibility, Command, State, Visitor, Template Method, Iterator și Memento.

In aceasta lucrare au fost folosite urmatorele pattern-uri:

* Factory Method: Clasa MasinaFactory oferă o metodă statică creazaMasina care se ocupă de crearea unui obiect de tip Masina. Această abordare abstractizează procesul de creare a unei instanțe de Masina și permite o flexibilitate mai mare în crearea obiectelor.
* Builder: Clasa DepozitMasiniBuilder are rolul de a construi un obiect de tip DepozitMasini. Aceasta permite adăugarea mai ușoară a mașinilor în depozit, utilizând metoda adaugaMasina, și apoi finalizarea construcției depozitului prin apelarea metodei build. Aceasta oferă o abordare mai fluentă și mai expresivă pentru construirea depozitului de mașini.
* Adapter: Clasa DepozitMasiniAdapter acționează ca un adapter între o instanță a clasei DepozitMasini și o interfață comună de adăugare, ștergere și căutare a mașinilor. Prin intermediul adapterului, se poate utiliza DepozitMasini într-un mod compatibil cu alte componente ale sistemului.
* Observer: Clasa DepozitMasiniObserver este o implementare a pattern-ului Observer. Aceasta permite înregistrarea de observatori (în cazul dat, obiecte de tip Observer) care vor fi notificate când se produc modificări în depozitul de mașini. Astfel, se asigură o separare între manipularea datelor și afișarea acestora.

# **DESCRIEREA PRINCIPIULUI DE LUCRU AL TEMATICII ALESE**

## SOLID

### Principiul responsabilității unice

Principiul responsabilității unice (SRP): Fiecare clasă are o singură responsabilitate și se concentrează doar pe acea responsabilitate. Aceasta ajută la menținerea coeziunii și la separarea preocupărilor.

Clasa "Masina" se ocupă doar de definirea unei mașini și nu are alte responsabilități.

class Masina {

    constructor(marca, model, culoare, an, pret) {

      this.marca = marca;

      this.model = model;

      this.an = an;

      this.pret = pret;

      this.culoare = culoare;

    }

  }

Clasa "DepozitMasini" gestionează operațiile pe colecția de mașini, cum ar fi adăugarea, ștergerea și căutarea mașinilor.

class DepozitMasini {

    constructor() {

      this.masini = [];

    }

    adaugaMasina(masina) {

      this.masini.push(masina);

    }

    stergeMasina(marca, model) {

      const index = this.masini.findIndex(m => m.marca === marca && m.model === model);

      if (index !== -1) {

        this.masini.splice(index, 1);

      }

    }

    cautaMasina(cuvantCheie) {

      return this.masini.filter(m => m.marca.includes(cuvantCheie) || m.model.includes(cuvantCheie));

    }

  }

### Principiul deschiderii-închiderii

Principiul deschiderii-închiderii (OCP): Acest principiu prevede că o entitate software trebuie să fie deschisă pentru extindere și închisă pentru modificare. În codul dat, putem vedea respectarea acestui principiu prin utilizarea unor design patterns precum Factory Method și Adapter.

Clasa "MasinaFactory" oferă o metodă statică "creazaMasina" pentru a crea obiecte de tipul "Masina". Astfel, adăugarea unui nou tip de mașină se poate face prin extinderea acestei clase și implementarea metodei corespunzătoare.

class MasinaFactory {

    static creazaMasina(marca, model, culoare, an, pret) {

      return new Masina(marca, model, culoare,  an, pret);

    }

  }

### Principiul substituției Liskov

Principiul substituției Liskov (LSP): Acest principiu prevede că un obiect de tipul unei clase derivate trebuie să poată fi înlocuit cu un obiect de tipul clasei de bază, fără a afecta corectitudinea programului. În codul dat, nu există o ierarhie de clase, dar principiul poate fi aplicat în contextul utilizării interfeței "Observer". Astfel, orice obiect care implementează interfața "Observer" poate fi adăugat ca observator în clasa "DepozitMasiniObserver".

### Principiul inversării dependențelor

Principiul inversării dependențelor (DIP): Acest principiu prevede că modulele de nivel superior nu ar trebui să depindă direct de modulele de nivel inferior, ci de abstracțiuni.

Clasa DepozitMasiniObserver depinde de interfața Observer, ceea ce permite injectarea oricărui obiect care implementează acea interfață în lista de observatori.

class DepozitMasiniObserver {

    constructor() {

      this.observers = [];

    }

    //....

### Principiul înlocuirii dependențelor

Principiul înlocuirii dependențelor (ISP): Acest principiu prevede că clienții nu ar trebui să depindă de interfețe pe care nu le utilizează.

Interfața Observer este separată de clasa DepozitMasiniObserver, astfel încât observatorii să poată implementa doar metodele necesare și să nu fie obligați să implementeze metode inutile.

class Observer {

    update() {}

  }

## DESIGN PATTERNS

### Factory Method

Factory Method este un pattern de proiectare creational care permite crearea de obiecte fără a specifica clasa exactă a obiectului care va fi creat. În schimb, se folosește o metodă factory care este responsabilă de crearea obiectului.

class MasinaFactory {

    static creazaMasina(marca, model, culoare, an, pret) {

      return new Masina(marca, model, culoare,  an, pret);

    }

  }

Clasa MasinaFactory oferă o metodă statică creazaMasina care se ocupă de crearea unui obiect de tip Masina. Această abordare abstractizează procesul de creare a unei instanțe de Masina și permite o flexibilitate mai mare în crearea obiectelor.

### Builder

Pattern-ul Builder este o metodă de construire a obiectelor complexe, în care obiectul este construit pas cu pas. Fiecare pas este reprezentat de o metodă a clasei Builder, care setează o proprietate a obiectului ce urmează să fie creat.

Clasa DepozitMasiniBuilder are rolul de a construi un obiect de tip DepozitMasini. Aceasta permite adăugarea mai ușoară a mașinilor în depozit, utilizând metoda adaugaMasina, și apoi finalizarea construcției depozitului prin apelarea metodei build. Aceasta oferă o abordare mai fluentă și mai expresivă pentru construirea depozitului de mașini.

class DepozitMasiniBuilder {

    constructor() {

      this.depozitMasini = new DepozitMasini();

    }

În constructorul clasei DepozitMasiniBuilder, se inițializează obiectul DepozitMasini ca un obiect gol, cu proprietățile necompletate. Această metodă este utilă atunci când dorim să creăm un obiect DepozitMasini, dar nu știm încă valorile pentru proprietățile acestuia.

Pentru a completa proprietățile obiectului DepozitMasini, clasa DepozitMasiniBuilder are metode specifice pentru fiecare proprietate a obiectului DepozitMasini. Aceste metode sunt definite ca metode ale clasei și primesc valoarea corespunzătoare proprietății ca argument, setând astfel valoarea proprietății obiectului book ce urmează să fie creat.

adaugaMasina(marca, model, culoare, an, pret) {

      const masina = MasinaFactory.creazaMasina(marca, model, culoare, an, pret);

      this.depozitMasini.adaugaMasina(masina);

      return this;

    }

Toate aceste metode returnează instanța curentă a obiectului DepozitMasiniBuilder (this), permițând astfel setarea mai multor proprietăți la un moment dat, prin concatenarea metodelor de setare a proprietăților.

La final, metoda build este utilizată pentru a crea obiectul DepozitMasini folosind valorile setate pentru proprietățile acestuia prin apelurile anterioare ale metodelor din clasa DepozitMasiniBuilder. În cadrul metodei build, este creată o nouă instanță a clasei DepozitMasini utilizând obiectul this.depozitMasini ca argument.

build() {

  return this.depozitMasini;

}

Acest pattern de design este util atunci când dorim să creăm obiecte complexe cu proprietăți multiple, dar nu dorim să definim constructori cu un număr mare de parametri. Prin intermediul pattern-ului Builder, putem construi obiectul pas cu pas, în funcție de necesități, permițând astfel o construcție mai flexibilă și mai ușor de înțeles a obiectelor complexe.

### Adapter

Pattern-ul Adapter este utilizat atunci când trebuie să adaptăm o interfață existentă a unei clase la o altă interfață a unei alte clase sau a unei biblioteci, astfel încât să putem utiliza acea clasă sau bibliotecă într-un mod compatibil cu codul nostru existent.

Clasa DepozitMasiniAdapter acționează ca un adapter între o instanță a clasei DepozitMasini și o interfață comună de adăugare, ștergere și căutare a mașinilor. Prin intermediul adapterului, se poate utiliza DepozitMasini într-un mod compatibil cu alte componente ale sistemului.

class DepozitMasiniAdapter {

    constructor(depozitMasini) {

      this.depozitMasini = depozitMasini;

    }

    adauga(marca, model, culoare, an, pret) {

      this.depozitMasini.adaugaMasina(MasinaFactory.creazaMasina(marca, model, culoare, an, pret));

    }

    sterge(marca, model) {

      this.depozitMasini.stergeMasina(marca, model);

    }

    cauta(cuvantCheie) {

      return this.depozitMasini.cautaMasina(cuvantCheie);

    }

  }

Acest pattern de design este util atunci când trebuie să utilizăm o clasă existentă sau o bibliotecă care nu corespunde interfeței de cod existentă, fără a modifica codul acelei clase sau biblioteci. Prin intermediul unui adapter, putem adapta interfața clasei existente la interfața de cod existentă, permițând astfel utilizarea acelei clase sau biblioteci într-un mod compatibil cu codul nostru existent.

### Observer

Pattern-ul Observer este un pattern de proiectare behavioral care permite comunicarea între obiecte într-o relație de tip "unu-la-mulți". Atunci când un obiect (subiectul) se schimbă, toate obiectele dependente (observatorii) sunt notificate și actualizate automat.

Clasa DepozitMasiniObserver este o implementare a pattern-ului Observer. Aceasta permite înregistrarea de observatori (în cazul dat, obiecte de tip Observer) care vor fi notificate când se produc modificări în depozitul de mașini. Astfel, se asigură o separare între manipularea datelor și afișarea acestora.

class DepozitMasiniObserver {

    constructor() {

      this.observers = [];

    }

    addObserver(observer) {

      this.observers.push(observer);

    }

    removeObserver(observer) {

      this.observers = this.observers.filter(obs => obs !== observer);

    }

    notifyObservers() {

      this.observers.forEach(observer => observer.update());

    }

  }

Acest pattern de design este util atunci când dorim sa sepărâm subiectul si observatori, permițând astfel o extensibilitate si flexibilitate mai mare în gestionarea comunicării între obiecte.

# **MODELAREA SISTEMULUI UML**

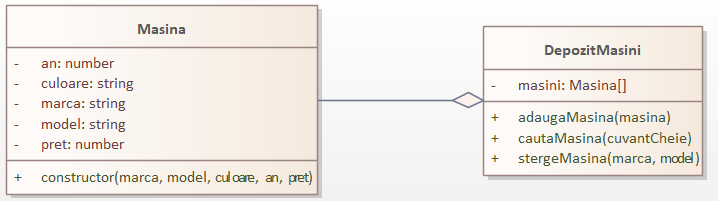


Figura 3.1 Diagrama UML pentru Principiul responsabilității unice

În schema UML (3.1) observăm două clase Masina si DepozitMasini.

* Clasa DepozitMasini are o relație de agregare cu clasa Masina. Adică, un depozit de mașini conține mai multe obiecte de tip mașină.
* Clasa Masina reprezintă o entitate care are atributele marca, model, culoare, an și pret. Aceasta este clasa de bază pentru obiectele de tip mașină.
* Clasa DepozitMasini este responsabilă de gestionarea mașinilor din depozit. Are un atribut masini, care este un vector de obiecte de tip Masina. Metodele sale includ adaugaMasina, stergeMasina și cautaMasina, care sunt utilizate pentru adăugarea, ștergerea și căutarea mașinilor în depozit

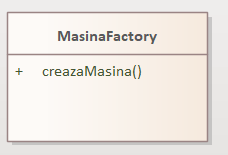


Figura 3.2 Diagrama UML pentru Factory Method Pattern

În figura 3.2 clasa MasinaFactory este reprezentată ca o clasă goală care definește metoda creazaMasina. Clasa MasinaFactory este o clasă statică care oferă un factory method creazaMasina. Acesta permite crearea ușoară a obiectelor de tip Masina și ascunde detaliile de creare în interiorul clasei.

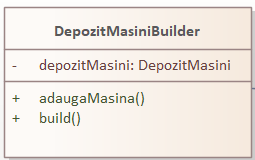


Figura 3.3 Diagrama UML pentru Builder Pattern

Clasa DepozitMasiniBuilder (3.3) implementează un builder pentru obiectul DepozitMasini. Acesta permite construirea treptată a unui depozit de mașini și furnizează metode pentru adăugarea mașinilor în timpul procesului de construcție. Clasa DepozitMasiniBuilder utilizează clasa MasinaFactory pentru a crea obiecte de tip Masina în timpul procesului de construcție a depozitului de mașini.

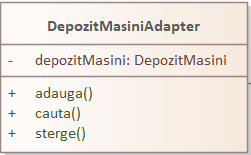


Figura 3.4 Diagrama UML pentru Adapter Pattern

În figura 3.4 clasa DepozitMasiniAdapter este un adapter care încapsulează un obiect DepozitMasini și îl adaptează pentru a oferi o interfață mai simplă pentru funcționalitățile de adăugare, ștergere și căutare a mașinilor. Aceasta utilizează metodele obiectului adaptat pentru a realiza operațiile respective. Clasa DepozitMasiniAdapter utilizează clasa MasinaFactory pentru a crea obiecte de tip Masina în cadrul operațiunilor de adăugare.

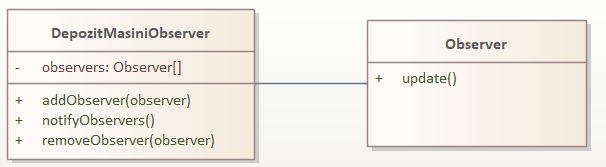


Figura 3.5 Diagrama UML pentru Observer Pattern

În figura (3.5) avem douaă clase DepozitMasiniObserver si Observer.

Clasa DepozitMasiniObserver are o relație de asociere cu clasa Observer, deoarece utilizează obiecte de tip Observer pentru a notifica modificările depozitului de mașini. Acestă clasă implementează un mecanism de observator. Are un atribut observers, care este un vector de obiecte Observer. Metodele sale includ addObserver, removeObserver și notifyObservers, care permit adăugarea, eliminarea și notificarea observatorilor atunci când se produc modificări în depozitul de mașini. Iar clasa Observer este o clasa goală care definește metoda update. Aceasta este implementată de observatori și este apelată atunci când sunt notificați.

# **CONCLUZII**

Folosirea pattern-urilor de design – Factory Method, Adapter, Builder și Observer în acest proiect a condus la un cod  modular și eficient. Aceste pattern-uri au permis separarea responsabilităților și crearea unui cod curat și lizibil, ceea ce a facilitat înțelegerea, modificarea și întreținerea acestuia.

Prin utilizarea acestor pattern-uri, s-a reușit crearea unui deposit de mașini cu o interfață intuitivă, care a putut interpreta comenzi, a adăugat și a șters mașini într-un mod controlat, a afișat mașinile într-un format atractiv și a permis crearea ușoară de noi mașini.

În plus, abordarea bazată pe aceste pattern-uri de design a îmbunătățit flexibilitatea sistemului, facilitând adăugarea de noi funcționalități sau modificarea celor existente fără a modifica structura generală a codului. În ansamblu, această abordare a contribuit la crearea unui cod mai durabil, care poate fi adaptat cu ușurință la schimbările viitoare.

Aceste pattern-uri reprezintă cele mai bune practici în programare, permițându-ne să ne concentrăm pe rezolvarea problemelor în mod eficient, în loc să reinventăm roata. Deci, prin utilizarea acestor pattern-uri, proiectul s-a dovedit a fi mai eficient, scalabil și ușor de gestionat.

# **BIBLIOGRAFIE:**

1. Solid [resursă electronică] – Regim de acces:

<https://web-creator.ru/articles/solid>

1. Design patterns[resursă electronică] – Regim de acces:

<https://en.wikipedia.org/wiki/Software_design_pattern>

1. Design patterns [resursă electronică] – Regim de acces:

<https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm>

1. Learn Design patterns [resursă electronică] – Regim de acces:

<https://refactoring.guru/design-patterns/creational-patterns>

1. Curs Design patterns [resursă electronică] – Regim de acces:

<https://www.youtube.com/watch?v=NU_1StN5Tkk&ab_channel=ProgrammingwithMosh>

1. Design patterns explained [resursă electronică] – Regim de acces:

<https://www.youtube.com/watch?v=tv-_1er1mWI&ab_channel=Fireship>