Curs 01

Examen

- Examen scris - 50%
- Proiecte de laborator (5 sau 6) - 50%

Documentaţie

1. **Head First Design Patterns** By Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra
2. **Agile Software Development, Principles, Patterns, and Practices** by Robert C. Martin
3. **Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition** by Erich Gamma (Author), Richard Helm (Author), Ralph Johnson (Author), John Vlissides (Author), Grady Booch (Foreword)
4. **Object Oriented Software Construction** by Bertrand Meyer

Structura Cursului

1. **Elemente de bază ale programării orientate pe obiecte** (recapitulare)
2. **Elementele de UML** (Unified Modelling Language)
   - diagrame de cazuri de utilizare
   - diagrame de clase
   - diagrame de secvenţă
3. **Principii de proiectare S.O.L.I.D** (5 principii)
   - **SRP** (Single Responsibility Principle)

    - **OCP** (Open Closed Principle)
    - **LSP** (Liskov Substitution Principle)
    - **ISP** (Interface Segregation Principle)
    - **DIP** (Dependency Inversion Principle)
4. **Design Patterns** - şabloane de proiectare pe studii de caz
    - se introduce un scenariu
    - se propune o soluţie
    - sunt analizate avantajele / dezavantajele
    - se extrage şablonul / şabloanele utilizat(e) în cadrul studiului de caz
    - prezentarea generală a şablonului de proiectare
      Sabloane
        - Structurale (1)
        - Comportamentale (2)
        - Creationale (3)
        - State Pattern
        - Template Method Pattern
        - Command Pattern
        - Composite Pattern
        - Strategy Pattern
        - Singleton Pattern
        - Abstract Factory Pattern
        - Factory Method Pattern
        - Observer Pattern
        - Facade Pattern
        - Adapter Pattern
5. **Elemente de arhitectură software**

Obiectivul cursului:

- utilizarea metodologiei obiectuale în proiectarea sistemelor software extensibile mentenabile.
- utilizarea şabloanelor de proiectare a soluţiei eficiente în rezolvarea problemelor de design.

Review POO

Clase

1. Prototip care indică atributele şi comportamentul unei familii de obiecte
2. Structură ce conţine date şi metode

Obiecte

1. O entitate care are un nume, o anumită stare şi un comportament în starea respectivă.
2. O variabilă ce are ca tip o clasă, o adresă de memorie, spaţiu alocat prin constructor pentru fiecare variabilă a sa şi o mulţime de metode.

Metode / Operaţii

- O funcţie membră a unei clase. Acestea implementează comportamentele obiectelor.

Constructori

- Metode speciale de creare a obiectelor în starea lor iniţială

Destructori

- Metoda de eliberare a resurselor ocupate de un obiect cand i se termină durata de viaţă.

Încapsulare

- Separarea detaliilor de implementare de interfaţa publică cu exteriorul şi ascunderea acestora poartă numele de încapsulare. Comunicarea cu obiectul se face doar prin intermediul interfeţei publice.

Implementare ascunsă

- Toate datele şi metodele ascunse ale clasei.

```
public Class C {
  // Detalii de implementare ascunse
  private int _data;
  private void PrivateMethod();

  // Interfata publica
  public C() {}
  public void PublicMethod();
}
```

Interfaţa publică a obiectelor

- Mulţimea metodelor publice ale obiectelor.
- Mesaj - Apelarea metodei unui obiect

```
Class A {
  public void a() {}
}
class B {
  A refA;
  public void b() {
    refA.a(); // b trimite un mesaj "a" catre obiectul refA
  }
}
```

Comportamentul obiectelor

- Se poate discuta despre comportament într-o anumită stare.
- Comportamentul reprezintă modul în care este afectată starea curentă a unui obiect când i se apelează orice metodă a sa (din interfaţa publică)

Starea obiectelor

- Valorile particulare ale atributelor obiectului.

Getter / Setter

- Operaţiile prin care putem obţine / modifica în mod controlat (vezi încapsularea)
- `get`: îmi permite să obţin starea curentă a obiectului (totdeauna metodă publică)
- `set`: permit modificarea stării curente

Moştenire / Clase derivate / Generalizare / Specializare

- Spunem că o clasă `A` este generalizare a unei clase `B` dacă orice obiect de tip `B` putem spune că este "un fel de" obiect de tip `A`.
  ATENŢIE!: Din punct de vedere comportamental (nu structural)
- Motor - Maşină (compunere între motor şi maşină: motorul face parte din maşină)
- Patrat - Dreptunghi
  - "Patratul este un fel de dreptunghi?" - Nu
  - "Dreptunghiul este derivat din patrat" - Corect (d.p.d.v. al comportamentului)

Polimorfism

Abilitatea de a transmite un mesaj cu aceeaşi semnătură către obiecte diferite, instanţe diferite ale unei ierarhii de clase, fără a cunoaşte destinatarii.

Pentru a beneficia de polimorfism (semantic) avem nevoie de următoarele elemente:

- o ierarhie de clase (o clasă de bază şi o mulţime de clase derivate)
- o metodă (virtuală) comun declarată în clasa de bază şi redefinită în clasele derivate
- apelul metodei se face prin intermediul unei referinţe (pointer) către clasa de bază

```
class B {
  public virtual void method() {...}
}
class D: B {
  public override void method() {...}
}
class C: B {
  public override void method() {...}
}

class Client {
  public void Process(B refB) {
    refB.method(); // Acesta este un mesaj (apel de metoda) polimorfic
  }
}

Client c = new Client();
c.Process(new B()); // Catre B
c.Process(new C()); // Catre C
c.Process(new D()); // Catre D
```

Observaţie: Polimorfismul este extrem de important pentru scrierea de componente software (librării) reutilizabile / extensibile. De exemplu, la o eventuală nouă derivare a clasei `B` (o nouă extensie), codul clasei `C` va rămâne **FUNCŢIONAL** fără a fi recompilat. (spunem ca este închis la modificări).

Abstractizare

Procedeul prin care obiecte diferite care fac parte din acelaşi domeniu (înrudite) sunt tratate în mod uniform printr-o noţiune generală, abstractă. `Line`, `Circle`, ... pot fi abstractizate prin noţiunea generală `Shape`.

Marele avantaj al abstractizării este acela că noţiunea foarte generală, abstractă, nu se modifică in timp! Prin urmare, dacă reuşim să construim componente software care să depindă **DOAR** de elemente abstracte, generale, acestea vor rămâne nemodificate (la noi cerinţe).

REGULĂ IMPORTANTĂ: Clasele `Client` trebuie să depindă de lucruri cât mai generale! Clasele concrete / abstracte trebuie să depindă de **ABSTRACTIZĂRI** (deoarece aceste sunt fixe în timp)

Clasă abstractă

O clasă abstractă reprezintă abstractizarea lumii reale în lumea programării orientate pe obiecte.

- nu poate fi instanţiată (este mult prea generală pentru a cunoaşte toate detaliile de implementare. Prin urmare, anumite metode vor fi declarate abstracte - nu au implementare)
- apare în vârful ierarhiilor de clase
- reprezintă un contract între clasele `Client` şi clasele unei ierarhii care oferă anumite servicii (numite clase `Server`).

Contractul este reprezentat de mulţimea metodelor publice expuse în clasa abstractă, care pot (şi trebuie) fi utilizate de clasa `Client` printr-o referinţă catre clasa de bază (ca să beneficiem de polimorfism, metode virtuale).

Interfaţă

Interfaţa poate fi considerată un caz particular de clasă abstractă: conţine doar declarări de metode. Nici o metodă nu conţine implementare (spre deosebire de clasa abstractă).

Interfaţă - rol de contract. Evidenţiază (ca tip) interfaţa publică comună a unei ierarhii de clase.

```
public interface ILogger {
  void Log(LogModel model);
}

// Clasa client
class LoggerManager {
  ILogger Logger { get; set; }
  public void Log(LogModel model) {
    Logger.Log(model);
  }
}

internal FileLogger: ILogger {
  public void Log(LogModel model) { }
}
```

Clasa `LoggerManager` este clasa client care beneficiază de ierarhia de clase `ILogger`. Utilizează diverse implementări ale interfeţei fără să le cunoască!

Funcţii virtuale

O metodă a unei clase de bază (generale), care este apelată sau nu, funcţie
de obiectul efectiv care prefixează apelul. Varianta apelată este dată de tipul obiectului creat, nu de tipul declarat al acestuia.

```
class B {
  public virtual void m(){...}
}
class D : B {
  public override void m(){...}
}
B ref = new D();
ref.m(); // Aici se alege varianta din clasa D (conform principiului functiilor virtuale)
```

Un obiect al clasei `D` conţine două variante pentru metoda `m`, salvate într-o tabelă numită `VFT` (Virtual Functions Table).

Legare dinamică (întârziată - late binding)

În strânsă legătură cu funcţiile virtuale. Obiectele care prefixează apelul unei funcţii virtuale sunt create dinamic, cu operatorul `new`. Acest fapt împiedică compilatorul să cunoască la momentul compilării secvenţa de cod care se va executa.

De aici şi numele de `legare intârziată`: asocierea unui apel de funcţie virtuală
cu o anumită implementare se numeşte `binding` (legare - legarea este amânată până la momentul execuţiei).