Curs 04

Principii de proiectare S.O.L.I.D.

- SRP (Single Responsibility Principle)
- OCP (Open Closed Principle)
- LSP (Liskov Substitution Principle)
- ISP (Interface Segregation Principle)
- DIP (Dependency Injection Principle)

OCP - Principiul închiderii-deschiderii

Enunț: O entitate software (clasă / metodă) trebuie să fie deschisă la extinderi şi închisă la modificări.

Explicație: O clasă se proiectează pentru implementarea unei / unor funcționalități (spunem că are niște responsabilități). Clasa trebuie proiectată astfel încât să putem schimba / optimiza / adaugă noi funcționalități fară modificarea codului sursă actual (fară să fie necesară recompilarea clasei). Închiderea se referă la faptul că, clasa este bine definită, compilabilă și nu trebuie să fie modificată dacă se doresc noi funcționalități.

Exemplu:

Să se implementeze o clasă pentru logging de erori, informații la consolă.

```
public enum LogLevel { Info, Error, Debug, Warn }

class Logger {
  public static void Log(string message, LogLevel level) {
    Console.WriteLine("{0} - {1} - {2}", DateTime.Now, level.ToString(),
message);
  }
}

// Utilizare
Logger.Log("Cocolino", LogType.Info);
```

Dacă se cere logging într-un fișier? Sau într-o baza de date? Clasa anterioară evident încalcă OCP din acest punct de vedere.

Soluția: abstractizăm această funcționalitate (printr-o clasă abstractă sau interfață)

```
public interface ILog {
  void Log(string message, LogLevel level);
}
```

```
class ConsoleLogging: ILog {
  public void Log(string message, LogLevel level) {
    Console.WriteLine("{0} - {1} - {2}", DateTime.Now, level.ToString(),
message);
  }
}

class FileLogging: ILog {
  public FilePath { get; private set; }
  public FileLogging(string filePath) {
    FilePath = filePath;
  }
  public void Log(string message, LogLevel level) {
    Console.WriteLine("{0} - {1} - {2}", DateTime.Now, level.ToString(),
message);
  }
}

// Inchisa la modificari, deschisa la extinderi
// deoarece posibilitatea de logging a fost izolata
// intr-o ierarhie separata de clase (prin interfata ILog)
class Logger {
  ILog _log; // punct de extensie
  public void Log(string message, LogLevel level) {
    _log.Log(message, level);
  }
}
```

În general, când dorim ca o clasa să adere la OCP dintr-un anumit punct de vedere (o anumită funcționalitate), soluția este **ABSTRACTIZAREA**: se va crea o clasă abstractă sau interfață (vezi `ILog`) ce joacă rol de contract între clasă și posibilele variante ale funcționalității respective.

Referința la interfață (clasa abstractă) se vă numit **punct de extensie**.

```
class Logger {
  ILog _log; // punct de extensie
}
```

Un exemplu clasic (des întâlnit) este cel în care avem două clase asociate, o clasă Client și una Server. Clasa Client beneficiază de un serviciu al clasei Server.

Dacă dorim să putem schimba serviciul fară să afectăm clasa Client, soluția este OCP.

```
class Client {
  IServer _server;
  public void DoJob() { _server.Job(); }
}

interface ISever { void Job(); }

class Server1: IServer {
  public void Job() { }
}
```

În mod real, cum se procedează pentru a obține aderare OCP.

Se identifică principala funcționalitate (vezi cardurile CRC) a clasei. Se verifică dacă repectiva funcționalitate poate varia (poate fi înlocuită/optimizată)
în timp.

Dacă da, aplicăm abstractizare și clasa inițială vă depinde de ceva abstract (deoarce ceea ce este abstract în general este **FIX**, nu variază).

Principiul LSP (Liskov Substitution Principle)

OCP recomandă utilizarea abstractizării, fapt ce conduce la ierarhii de clase (moștenire). **LSP** arată tocmai modul prin care se construim această ierarhie.

Enunț:

- (1) Dacă o entitate (metodă/clasă) depinde de o referință (pointer) către o clasă de bază, aceasta trebuie să rămână funcțională pentru toate clasele derivate fară să cunoască nimic despre acestea.
- (2) Entitata nu știe despre clasele derivate!

Altfel spus, dacă înlocuim referința respectivă cu **ORICE** clasă ce implementează interfața / clasa abstractă, entitatea rămâne funcțională.

Încălcare LSP => Încălcare OCP.

Exemplu de încălcare.

(1) - clasa ştie despre clasele derivate

```
class GraphicTool {
  public void Draw(ShapeType type) { }
}

public enum ShapeType { Circle, Triangle, Rectangle, Line }

abstract class Shape {
  public ShapeType Type { get; set; }
  public abstract void Draw();
}

class Circle: Shape {
  public override void Draw() { }
  public double Radius { get; set; }
}

class Line: Shape {
  public override void Draw() { }
}

class GraphicTool {
  List<Shape> _shapes;

  // incalcare LSP => incalcare OCP (clasa are referinta catre  Shape si totusi
stie de clasele derivate (depinde de acestea)
  public void DrawAll() {
    // incalca OCP deoarece la un eventual nou tip de figura geom. instr.case se
va modifica
    foreach(var shape in _shapes)
```

```
      switch (shape.Type) {
        case ShapeType.Circle:
          var circle = (Circle) shape;
          circle.Draw();
          var r = circle.Radius;
          // ...
          break;
        }
  }
}
```

(2) - Încălcare LSP mai subtilă. Moştenirea şi redefinirea metodelor au fost utilizate incorect.

```
class Rectangle {
  public virtual int Width { get; set; }
  public virtual int Height { get; set; }
  public int Area() {
    return Width * Height;
  }
}

class Square: Rectangle {
  public Square(int length) {
    Width = Height = length;
  }
  // Solutie temporara!!
  public override int Width {
    get { /*...*/ }
    set { Height = value; }
  }
}
```

```
// Conform LSP, nu cunosc nimic despre clasele derivate 200
void Draw(Rectangle r) {
  r.Width + = 20;
  // Daca substituim pe r cu Square, instr. de mai sus va modifica inclusiv
Height
  var w = r.Width;
  var h = r.Height;
  var area = w * h; //66000
  if (area == someValue) { /*desenez*/ }
}
// Nu functioneaza corect pentru clasa Square!!
```

(a) clasa `Square` are cel puţin o variabilă suplimentară
(b) pot apărea probleme dacă cineva apelează separat una din cele două proprietăţi

```
var s = new Square(20);
s.Width = 10;
var area = s.Area(); // 100 // 200
```

- Tehnica `Design By Contract` (Vezi cartea `Object Oriented Software Construction, B. Meyer`)

Se considera o clasă `Server` (ce oferă servicii) şi o clasă `Client` care beneficiază de serviciile clasei `Server`. Clasa `Server` "semnează" un contract cu clasa `Client` (prin intermediul unei clase abstracte sau interfețe): totalitatea metodelor publice din interfață / clasa abstractă ce pot fi utilizate de `Client`.

Pentru fiecare metode se stabilesc doua proprietăți: `Precondiție` și `Postcondiţie`.

- **Precondiţia**: o formulă adevărată înainte să apelez serviciul (metodă)
- **Postconditia**: o forumula adevărată după apelarea serviciului (metodei).

Conform tehnicii **Design prin Contract**, clientul se obligă să apeleze metoda într-un context în care preconditia este adevărată. Serverul se obligă să asigure `Postcondiţia`.

Avantaje:
(1) Codul simplificat al clasei `Server`.
(2) Aderare la LSP în următoarele condiţii: orice clasă derivată care redefineşte o metodă a clasei de bază trebuie să asigure o precondiţie "mai slabă" şi o postcondiţie "mai tare" decât cele din clasa de bază.

```
class ServerA {
  // P
  public virtual void Method() {}
  // Q
}

class ServerB: ServerA {
  // P'
  public override void Method() {}
  // Q'
}
// P  => P' (adevarata)
// Q' => Q (adevarata)
```

În aceste condiţii de "moştenire", se asigură LSP. De ce?

```
class Client {
  ServerA _server;
  public void Job() {
    // aici P este adevarata!! Din P => P' rezulta si P' adevarata
    _server.Method(); //aici, de ce codul ramane functional si pentru ServerB?
    //P' adevarata, conform DbC, Q' adevarata!, dar Q' => Q;
    // aici Q este adevarata
  }
}
```

(Se pot stabili invarianți de clasă. Ce înseamnă o clasă corectă? Invarianți pentru constructori default.)

SRP (Single Responsibility Principle) - Principiul Responsabilității unice.

O clasă trebuie să conțină o singură responsabilitate de bază. Drept consecință, clasa repectiva vă avea "un singur motiv" pentru a fi modificată, schimbată.

Codul este uşor de întreţinut, de testat.