

Modulul 3

Design Patterns

Overview

- Bad design vs. Good design
- Principii de design
- Ce este un design pattern?
- Câteva design pattern-uri

Acesta este un modul interactiv!



Acest modul introduce noțiuniunile de **software design** și **design patterns**. Veți învăța în acest modul care sunt caracteristicile unui design bine făcut, principiile ce trebuie urmate pentru a face un design bun, ce este un design pattern precum și câteva design pattern-uri des utilizate.

La finalul acestui modul, veți:

- Înțelege caracteristicile unui design bun
- Putea aplica principiile de design pentru programele proprii
- Înțelege ce este un design pattern și cum trebuie folosit
- Putea să folosiți design pattern-uri în programele proprii

Bad design vs. Good design

- ▶ Rigiditate
- ▶ Fragilitate
- ▶ Imobilitate
- ▶ Mentenabilitate
- ▶ Modularitate
- ▶ Loose coupling
- ▶ Extensibilitate
- ▶ Scalabilitate

10/31/2014

3

A face un design pentru un program utilizând paradigma programării orientate pe obiecte nu este un lucru usor, iar a face un design reutilizabil este și mai greu. Pentru aceasta trebuie să găsiți obiectele potrivite specificațiilor date în limbaj natural, factorizate în clase folosind granularitatea corectă, definite interfețe pentru clase și ierarhii de moștenire, precum și definirea relațiilor cheie dintre obiecte. Un design trebuie să fie specific problemei date, dar în același timp trebuie să fie suficient de general încât să poată face față viitoarelor probleme și specificații.

Deasemenea se dorește pe cât posibil evitarea redesign-ului sau cel puțin minimizarea acestuia.

Deși este destul de greu de definit într-o singură frază ce înseamnă un design bun, există anumite caracteristici ce pot indica faptul că un anumit design este bun și reutilizabil sau nu. Unele dintre cele mai importante astfel de caracteristici sunt enumerate și explicate mai jos.

Bad design

- Rigiditate – orice schimbare afectează mai multe componente ale sistemului
- Fragilitate – o schimbare conduce la erori în componente neașteptate ale sistemului
- Imobilitate – cod greu de reutilizat

Good design

- Mentenabilitate – proprietatea unui sistem de a fi ușor de întreținut
- Modularitate – proprietatea unui sistem de a avea mai multe părți componente, individuale și bine definite care interacționează
- Loose coupling – proprietatea unui sistem de a avea componente care interacționează cu minim de cunoștiințe (ideal, fără cunoștiință) de implementarea concretă a celorlalte componente independente
- Extensibilitate – noi facilități pot fi adăugate programului fără a fi nevoie de schimbări majore ale arhitecturii fundamentale
- Scalabilitate – programul se adaptează ușor creșterii numărului de date sau utilizatori

Principii de design

- ▶ **S**ingle Responsibility Principle
- ▶ **O**pen Close Principle
- ▶ **L**iskov Substitution Principle
- ▶ **I**nterface Segregation Principle
- ▶ **D**evelopment Inversion Principle

După cum s-a menționat mai devreme, a face un design bun nu este un lucru ușor. Pentru obține un astfel de design, în special pentru un sistem complex, este nevoie de mai mult decât de cunoștiințele teoretice. Este nevoie deasemenea de experiență și poate și un gram de noroc. Mai jos sunt enumerate și explicate câteva principii de design importante, care deși nu oferă garanția realizării unui bun design de la prima încercare, dacă sunt înțelese și urmate pot reduce numărul de redesign-uri.

Single Responsability Principle

- ▶ ***Nu trebuie să existe mai mult de un motiv pentru ca o clasă să se schimbe.***

▶ Responsabilitate = Motiv de schimbare

Robert C. Martin

▶ Principiu intuitiv, de bază... și totuși greu de respectat

10/31/2014

5

- **Single Responsability Principle**

Nu trebuie să existe mai mult de un motiv pentru ca o clasă să se schimbe.

După cum spune și numele principiului, fiecare clasă trebuie să aibe un singur rol bine definit. În acest context, responsabilitatea a fost definită ca “motiv de schimbare”. Dacă într-o clasă există două motive pentru schimbare (clasa are două responsabilități), aceasta trebuie spartă în două clase distincte, fiecare cu responsabilitatea proprie. Acest principiu este unul foarte bun, deoarece anticipatează apariția unor noi cerințe și astfel a unor posibile schimbări. În momentul în care o clasă are mai mult de o responsabilitate, acestea sunt puternic cuplate. Schimbări aduse uneia dintre responsabilități pot afecta în mod neașteptat pe celelalte. Acest lucru conduce la un design fragil.

Deși acest principiu este destul de intuitiv, este totuși destul de greu de respectat. Găsirea și separarea responsabilităților reprezintă o mare pare a procesului de design.

Respectarea acestui principiu elimină apariția aşa numitelor clase “Manager”, cu multe responsabilități, în care fiecare contribuitor adaugă noi metode, pentru că nu este sigur unde s-ar potrivi, conducând în acest fel la clase mari greu de menținut.

Open Close Principle

- ▶ **Entitățile software (Clase, Module, Funcții, etc.) ar trebuie să fie deschise pentru extindere, dar închise pentru modificări.**
- ▶ Deschise pentru extindere: prin adăugare de cod (moștenire, agregare)
- ▶ Închise pentru modificari: Cod sursă nemodificat

10/31/2014

6

- **Open Close Principle**

Entitățile software (Clase, Module, Funcții, etc.) ar trebuie să fie deschise pentru extindere, dar închise pentru modificări.

Ceea ce spune acest principiu este că un cod odată scris și testat cu succes, nu ar trebui să mai fie modificat. În schimb funcționalitățile sale trebuie să poată fi extinse astfel încât programul să facă față noilor cerințe prin adăugare de cod nou. În momentul în care adăugarea de noi funcționalități necesită modificarea codului existent al unei entități, acest lucru poate conduce la schimbări în cascadă ale entităților dependente. Nerespectarea acestui principiu va crea un design ce va avea toate atrитеle unui design greșit: rigiditate (am menționat deja modificările în cascadă ale entităților dependente), fragilitate (Cum putem ști unde să ne oprim cu modificările? Când putem fi siguri că schimbările aduse ultimei verigi din lanțul presupus de dependințe nu va crea erori neașteptate următoarelor verigi pe care noi nu le-am luat în calcul), imobilitate (nu se poate numi că reutilizăm un cod dacă îl aducem modificări).

În cazul claselor, cheia respectării acestui principiu este reprezentată de abstractizare. Existența claselor abstracte va obliga clasele concrete să implementeze acele metode ce se pot modifica în funcție de cerințe și astfel nu va fi nevoie să schimbăm codul din clasei abstracte. O alta metodă prin care putem adăuga funcționalități, fară a modifica codul unei clase, este prin agregarea* obiectelor acestei clase. În acest fel de ne putem folosi de codul deja scris prin apelul metodelor clasei aggregate și putem adăuga cod nou pentru a îndeplini noile cerințe și a adăuga noi funcționalități.

Pe scurt:

- Deschis pentru extindere:
 - comportamentul entității poate fi extins
 - se pot adăuga noi funcționalități
 - acest lucru se face prin adăugare de cod
 - se folosesc abstractizări, moșteniri și agregări
- Închis pentru modificări:
 - codul sursă al entității nu este modificat

Beneficii:

- reutilizarea codului
- mentenabilitate
- extensibilitate

Liskov Substitution Principle

- ▶ **Fie $q(x)$ o proprietate dovedită a obiectelor x de tip T . Atunci $q(y)$ poate fi dovedită pentru obiectele y de tipul S , unde S este un subtip a lui T .**
- ▶ **Obiectele având tipuri derivate trebuie să substituie complet obiectele clasei de bază.**

10/31/2014

7

- **Liskov Substitution Principle**

Fie $q(x)$ o proprietate dovedită a obiectelor x de tip T . Atunci $q(y)$ poate fi dovedită pentru obiectele y de tipul S , unde S este un subtip a lui T .

În forma în care a fost enunțat acest principiu este destul de greu de înțeles la ce anume se referă el. O altă definiție ar putea fi dată în felul următor:

Obiectele având tipuri derivate trebuie să substituie complet obiectele clasei de bază.

Practic, nu trebuie să avem niciun fel de cunoștiințe despre felul obiectului (este al clasei de bază sau este al unei clase care mosteștește clasa de bază) pentru a putea lucra cu acesta, în momentul în care îl privim prin interfața clasei de bază. Acest principiu este unul destul de greu de respectat, pentru că nu există un mod simplu prin care să ne uităm la codul unei clase și să spunem dacă acesta încalcă sau nu principiul.

NOTĂ: Există însă formalisme matematice prin care ne putem da seama dacă este încălcăt acest principiu, folosind invariante semantică, teorema Shannon pentru a calcula ceea ce a fost definit ca entropia design-ului sau alte tehnici ce nu fac însă subiectului acestui modul.

Pentru a înțelege însă mai bine acest principiu să luăm un exemplu simplu. Vrem să facem un modul care face management la elipse și cercuri.

Presupunem că avem urmatorul cod pentru clasa elipsă:

```
class Elipsa
{
    private int axisX;
    private int axisY;

    public virtual int AxisX
    {
```

```

        get { return axisX; }
        set { axisX = value; }
    }

    public virtual int AxisY
    {
        get { return axisY; }
        set { axisY = value; }
    }
}

```

Cunoaștem faptul ca acest cod funcționează foarte bine și dorim să adăugăm cod astfel încăt să putem avea și cercuri. Ne gândim că geometric cercul este o elipsă ce are lungimile axelor egale ($axisX == axisY == R$) și hotărâm să derivăm clasa elipsă astfel făcând cele 2 axe egale.

```

class Cerc : Elipsa
{
    public override int AxisX
    {
        get { return base.AxisX; }
        set { base.AxisX = value; base.AxisY = value; }
    }

    public override int AxisY
    {
        get { return base.AxisY; }
        set { base.AxisX = value; base.AxisY = value; }
    }
}

```

Avem acum însă următoarea metoda de test:

```

class Test
{
    public static bool test(Elipsa e)
    {
        e.AxisX = 3;
        e.AxisY = 4;

        if (e.AxisX != e.AxisY)
        {
            Console.WriteLine("Pass");
            return true;
        }

        Console.WriteLine("Fail");
        return false;
    }
}

```

```
public static void Main(string[] Args)
{
    test(new Elipsa()); // Pass
    test(new Cerc()); // Fail

    Console.ReadKey();
}
```

Dacă ați rula secvența de cod anterioară, ați vedea că primul test ar trece, în timp ce al doilea test ar pică. Utilizatorul clasei Eplisă se aşteaptă ca cele două axe să fie diferite după apelarea celor două proprietăți. El nu știe de clasa Circle și nici nu ar trebui să știe conform principiului. El ar trebui să poată folosi obiectele clasei Cerc în aceleași scenarii și să primească același rezultat ca și când ar folosi obiectele clasei Elipsă. După cum s-a menționat anterior este dificil să ne dăm seama când este încălcăt Principul Substituției Liskov.

Probabil v-ați dat deja seama din exemplu, încălcarea acestui principiu face ca design-ul sa fie fragil.

Interface Segregation Principle

- ▶ **Clienții nu trebuie forțați să depindă de interfețe pe care nu le folosesc.**

```
interface Animal
{
    void eat();
    void sleep();
    void moveTail();
}
```

- ▶ Dacă animalul este o gorilă?

- **Interface Segregation Principle**

Clienții nu trebuie forțați să depindă de interfețe pe care nu le folosesc.

Acest principiu vine să limiteze creșterea interfețelor. Există în literatură un concept numit “poluarea interfeței” și înseamnă adăugarea în interfață a metodelor ce nu sunt folosite de anumiți clienți sau metode pe care clasele sunt obligate să le implementeze, chiar dacă nu au sens pentru acestea. Să luăm următorul exemplu. Avem interfața Animal:

```
interface Animal
{
    void eat();
    void sleep();
    void moveTail();
}
```

Ce facem însă dacă vrem să modelăm comportamentul unei gorile? Gorilele nu au coadă, astfel metoda moveTail nu are sens în acest context. Ce am putea face? Să aruncăm o excepție de tipul NotImplementedException? S-ar putea ca nimeni să nu se gândească să pună un bloc try catch în jurul acestei metode. Soluția în acest caz este să fim atenți în momentul în care adăugăm o metodă într-o interfață dacă aceasta chiar are sens pentru toate clasele ce vor implementa această interfață.

Un altfel de poluare este crearea de interfețe mari ce conține grupuri de metode disjuncte ce sunt apelate de clienți separați. Soluția în acest caz este spargerea interfeței în mai multe interfețe corespunzătoare grupurilor de metode. În acest fel se decuplează și logica clientilor, iar programul este mult mai ușor de înțeles.

Nerespectarea acestui principiu implica rigiditate.

Dependency Inversion Principle

- **Modulele High-level nu trebuie să depindă de modulele low-level. Ambele ar trebui să depindă de abstractizări.**
- **Abstractizările n-ar trebui să depindă de detalii. Detaliile ar trebui să depindă de abstractizări.**

10/31/2014

9

- **Dependency Inversion Principle**

*Modulele High-level nu trebuie să depindă de modulele low-level. Ambele ar trebui să depindă de abstractizări.
Abstractizările n-ar trebui să depindă de detalii. Detaliile ar trebui să depindă de abstractizări.*

Acest principiu spune că între modulele high-level și modulele low level ar trebui să existe un nivel de abstracțare (clasa abstractă sau interfață). În metoda clasică de dezvoltare, atunci când un modul/clasă are nevoie de funcționalitățile altor clase, o instanțiază cu tipul concret și menține o referință la aceasta. Acest lucru conduce la o cuplare puternică a componentelor software-ului și la un design rigid. Clasa client (clasa care instanțiază) trebuie să folosească fie o interfață, fie o clasă abstractă. Această soluție face ca mai târziu să fie foarte ușor de modificat comportamentul clasei client pur și simplu prin schimbarea modulului instanțiat și păstrând restul codului neschimbat.

Să presupunem că intenționăm să scriem o clasă ce folosește un ArrayList (implementat tot de noi), deoarece algoritmul pe care această clasă îl implementează adaugă elementele o singură dată și ulterior face multe accesări.

```
class Algorithm
{
    private ArrayList a = new ArrayList();
    public void Compute()
    {
        a.addToArrayList(x);
        a.addToArrayList(y);
        a.addToArrayList(z);

        // ... multe accesari
    }
}
```

Totuși mai târziu ne dăm seama că vrem să extindem algoritmul nostru, iar noua componentă are un număr mult mai

mare de adăugări și ștergeri decât numărul de accesări din metoda Compute și am vrea să folosim o listă înlanțuită.

Distingem 2 probleme (subliniate și în principiu):

- algoritmul depinde de o clasă low-level concretă low-level ArrayList
- chiar dacă ArrayList ar fi o interfață, abstractizarea ar depinde de detalii concrete (metoda se numește addToArrayList)

Solutie:

- construim o interfață IList care nu depinde ce nu are cunoștiință de detaliile de implementare. ArrayList va implementa interfața IList

```
interface IList {  
    void add(Object *o);  
    void remove(Object* o);  
    Object elementAt(int i);  
}
```

Și o folosim în implementare:

```
class Algorithm  
{  
    private IList a = new ArrayList();  
    public void Compute()  
    {  
        a.add (x);  
        a.add (y);  
        a.add (z);  
  
        // ... multe accesari  
    }  
}
```

În acest moment, dacă dorim să exindem algoritmul în modul prezentat anterior, singura modalitate pe care trebuie să o aducem codului existent este înstantierea IList a= new LinkedList();

Ulterior în acest modul vom învăța metode prin care putem să scoatem până și instantierea concretă a Array/LinkedList-ului.

Ce este un design pattern?

- ▶ "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander
- ▶ "Design Patterns : Elements of Reusable Object-Oriented Software" – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four)
- ▶ Design pattern-urile nu dau o soluție propriu-zisă, ele trebuie să inspire o soluție

10/31/2014

10

Christopher Alexander a fost primul care a venit cu ideea de design pattern-uri. Deși el se referea la arhitectură, citatul este valabil și în cazul software design-ului.

Pe 21 octombrie 1994, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, cunoscuți și sub numele de Gang of Four publică lucrarea "Design Patterns: Elements of Reusable Object Oriented Software". Această carte a rămas lucrarea de referință în materie de design pattern-uri și conținea o listă de 23 de pattern-uri.

Întâi vom menționa ce **NU** este un design pattern:

- Un design pattern nu este un şablon, cel puțin nu în ideea de şablon de croitorie sau desen.
- Un design pattern nu este o rețetă. Nu există niște pași clari ce trebuie urmați și conduc la soluția dorită.

Un design pattern este o soluție la o problema recurrentă din domeniul software design-ului. Un design pattern nu este un design finisat până la ultimele detalii ce poate fi transformat direct în cod sursă sau cod mașină. Un pattern este o idee generală de rezolvare a unei probleme de design, ce poate fi folosită în diferite situații și contexte diferite. Acestea arată de obicei relațiile și interacțiile dintre clase și obiecte, fără însă a da o implementare concretă.

Un design pattern nu trebuie să dea o soluție propriu-zisă pentru o problemă, el trebuie să inspire o soluție.

Avantajele folosirii design pattern-urilor

The wheel was already invented... and so were design patterns

- ▶ Accelerează procesul de dezvoltare
- ▶ Oferă niște soluții testate de-a lungul a peste 20 de ani
- ▶ Îmbunătățesc lizibilitatea codului pentru programatori și arhitecți familiarizați cu pattern-urile
- ▶ Permit programatorilor să folosesc să asocize un nume, cu o soluție complexă

	Purpose			
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visito

10/31/2014 12

Design pattern-urile se clasifică după două criterii:

- **Intenție** - ce anume face pattern-ul
- **Scop** - specifică unde se aplică (claselor sau obiectelor)

După **intenție** acestea sunt de trei tipuri:

- **Creational**

Abstractizează procesul de întâmpinare

Ajută la crearea unui sistem independent de modul în care obiectele sunt create, compuse și reprezentate.

- **Structural**

Acestea se referă la modul de organizare al claselor sau obiectelor astfel încât să creeze structuri complexe

- **Comportamental**

Acestea caracterizează modurile în care clasele sau obiectele interacționează sau distribuie responsabilități.

Descrierea pattern-urilor

- ▶ Intentie
- ▶ Nume alternative
- ▶ Motivatie
- ▶ Aplicabilitate
- ▶ Structura
- ▶ Participanti
- ▶ Colaboranti
- ▶ Consecinte
- ▶ Implementari

10/31/2014

13

- **Intentie**

Ce face acest design pattern? Ce problemă rezolvă?

- **Nume alternative**

- **Motivatie**

Un scenariu care să ilustreze folosirea design pattern-ului

- **Aplicabilitate**

Situatii in care acest pattern se aplică

- **Structura**

O diagramă UML

- **Participanti**

Clasele si/sau obiectele participante la design si responsabilitatile lor

- **Colaborari**

Cum interacționează participanții pentru a-și îndeplini responsabilitățile

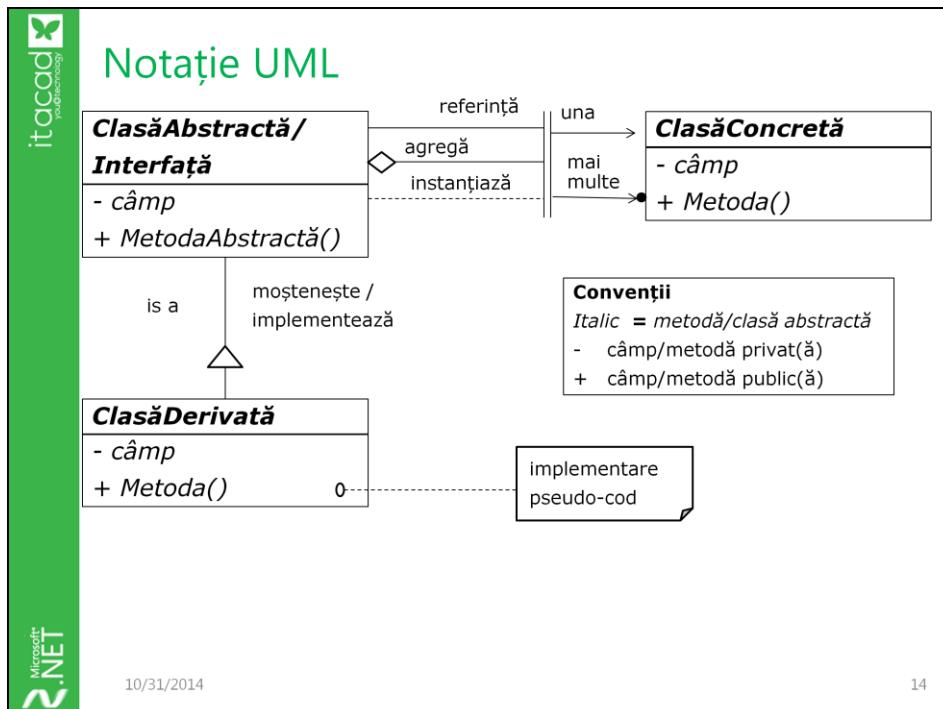
- **Consecinte**

Rezultatul aplicării, avantaje, dezavantaje, trade-off-uri

- **Implementari**

Possible implementare folosind caracteristicile limbajului C# (**pentru scenariul prezentat**)

ATENȚIE! După cum s-a accentuat până acum de-a lungul modulului, design pattern-urile nu au o singură implementare. Acestea sunt dependente de problemă și context. Implementarea sugerată se referă la scenariul prezentat.



Folosind notația UML (Unified Modeling Language) vom reprezenta câteva din cele mai cunoscute design pattern-uri cunoscute. Această notație ne permite să descriem succint, fară a oferi o implementare concretă diferitele design pattern-uri, arătând elementele componente și relațiile dintre acestea.

Scenariu

- ▶ Implementarea unui sistem de fișiere
 - ▶ Single user
 - ▶ Comenzile cat, ls, chmod
 - ▶ Scriere și citire din fișiere
 - ▶ Sistem de notificări la modificarea fișierelor

Pentru a înțelege mai bine cum se folosesc design pattern-urile, vom crea un design simplu pentru un sistem de fișiere, ce permite existența unui singur utilizator.

Vom implementa comenziile cat, ls, chmod, astfel încât să putem adăuga oricâte comenzi.

Vom implementa stream-uri de scriere și citire în și din fișiere.

Deasemenea vom implementa un sistem de notificări pentru a știi când sunt modificate fisierele.

Challenge 2

- Încercați să găsiți o soluție de implementare astfel încât în orice moment să nu existe mai mult de o instanță a clasei User în program

Challenge 1

- Încercați să găsiți o soluție de implementare astfel încât în orice moment să nu existe mai mult de o instanță a clasei User în program

Răspuns: Singleton Pattern

User
- static User instance
- User();
+ static User GetInstance()

Răspunsul la acest challenge este design pattern-ul **Singleton**.

Pentru a implementa acest pattern trebuie:

1. Să facem constructorul clasei User privat. Astfel nu se poate crea o instanță a clasei User din afara acesteia.
2. Creăm o variabilă statică de tip user privată.
3. Creăm o metodă statică publică ce va întoarce referința la variabila statică privată.

În forma în care este prezat aici pattern-ul Singleton avem două implementări posibile, fiecare cu avantaje și dezavantaje:

Implementarea 1

```
class User
{
    private static User instance = null; //referință statică privată

    public static User GetInstance() {
        if (instance == null)
            instance = new User();

        return instance;
    }

    private User() { } //constructor privat
}
```

Avantaje:

- Implementarea folosește tehnica lazy instantiation. Aceasta înseamnă că obiectul User este creat atunci când este apelată metoda GetInstance. Dacă metoda nu este apelată niciodată, înseamnă că obiectul nu este folosit niciodată și deci nu trebuie creat.

Dezavantaje:

- Implementarea aşa cum este prezentată aici nu este sigură pentru multi threading (veți discuta acest aspect

intr-un modul viitor).

- Există un overhead datorat executării if-ului la fiecare apelare a metodei GetInstance, chiar dacă el nu are efect decât prima dată

Recomandată atunci când obiectul este greu de instantiat și nu se dorește pierderea timpului la inițializare

Implementarea 2

```
class User
{
    private static User instance = new User(); //referință statică privată

    static User GetInstance() { return instance; }

    private User() { } //constructor privat
}
```

Avantaje:

- Nu are probleme în cazul multithreading-ului
- Nu mai am overhead-ul introdus de if-ul adițional

Dezavantaje:

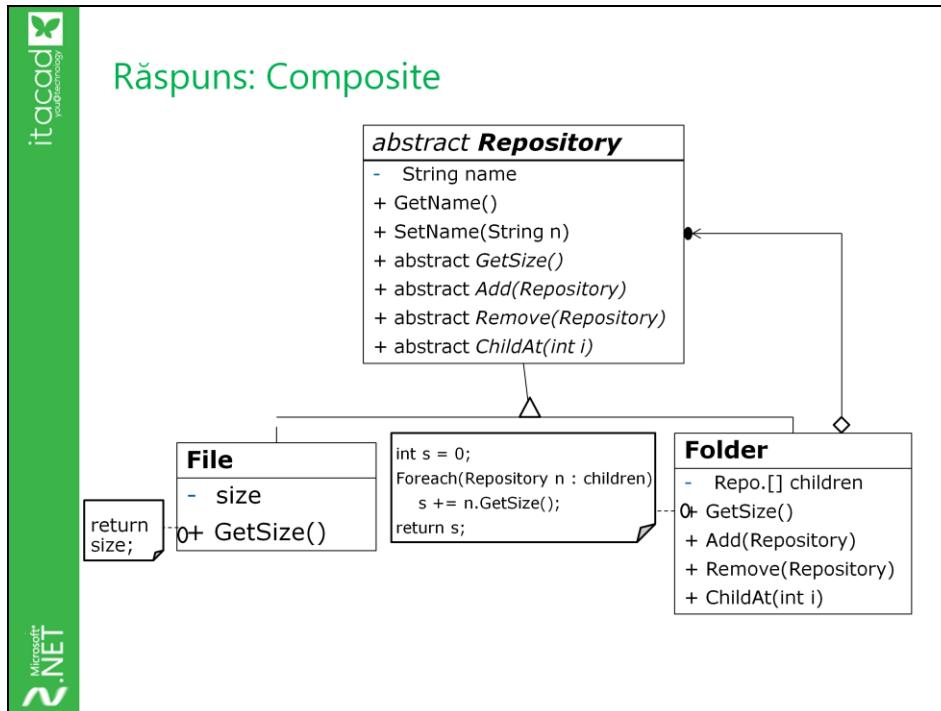
- Pierd timp la inițializare pentru a crea obiectul, chiar dacă poate nu o să am nevoie de el

Singleton

- ▶ Ne asigurăm că nu putem avea decât o singură instanță a unei clase la un moment dat
- ▶ Există situații în care nu are sens să avem mai mult de o instanță pt o clasă.
Ex. : Președinte, motor la mașină, etc.
- ▶ Acces controlat la instanță prin metoda statică
- ▶ Există un punct global de acces la instanță
- ▶ Probleme: Multithreading, Moștenire

Challenge 2

- ▶ Faceți design-ul sistemului de fișiere astfel încât:
 - ▶ să putem avea fișiere
 - ▶ să putem adăuga oricând noi tipuri de fișiere
 - ▶ fișierele/folderele trebuie să aibă un nume
 - ▶ să putem calcula dimensiunea fișierului/folder-ului



Răspunsul la acest challenge este designd pattern-ul **Composite**.

Implementare posibilă

```

abstract class Repository
{
    private string name;

    public Repository(string name)
    {
        this.name = name;
    }

    public string Name
    {
        get { return name; }
    }

    public abstract void Add(Repository r);
    public abstract void Remove(Repository r);
    public abstract int Size { get; }
}

class Folder : Repository {
    IList<Repository> children = new List<Repository>();

    public override void Add(Repository r)
    {
        children.Add(r);
    }
}

```

```
public override void Remove(Repository r)
{
    children.Remove(r);
}

public IList<Repository> Children
{
    get { return children; }
}

public Folder(String s) : base(s) { }

public override int Size
{
    get {

        int s = 0;
        foreach (Repository r in children)
            s += r.Size;
        return s;
    }
}
}

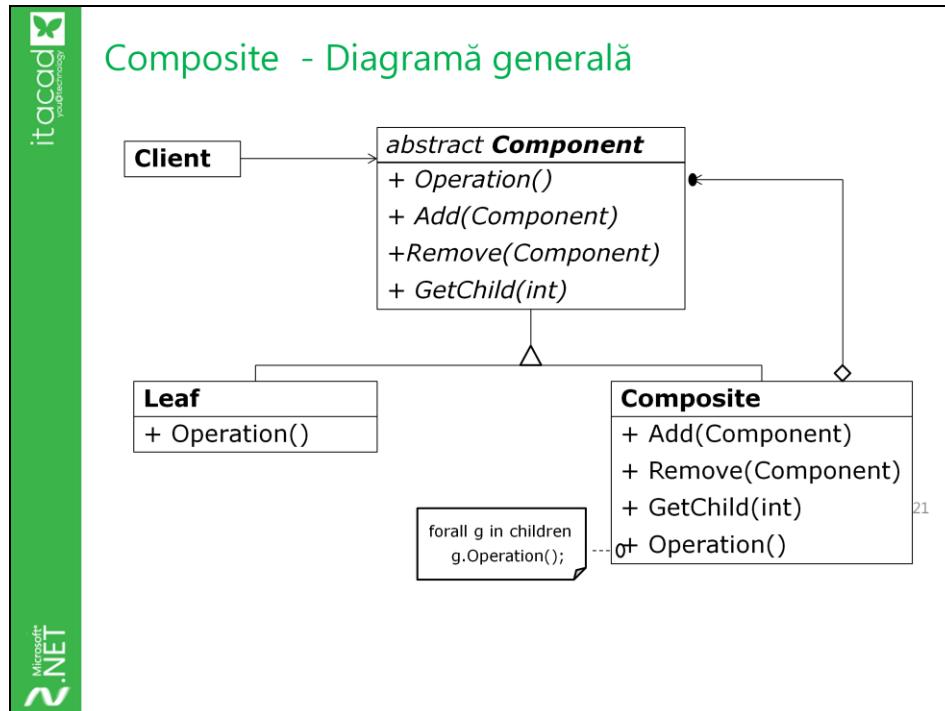
class File : Repository
{
    private int size;

    public override int Size
    {
        get { return size; }
    }

    public override void Add(Repository r)
    {
        throw new NotImplementedException();
    }

    public override void Remove(Repository r)
    {
        throw new NotImplementedException();
    }

    public File(String s, int size) : base(s)
    {
        this.size = size;
    }
}
```



Composite

- ▶ Folosit atunci când avem nevoie să creăm ierarhii
- ▶ Obiectele individuale și compuse sunt tratate uniform
- ▶ **Component**
 - ▶ Declara interfața pentru obiecte
- ▶ **Leaf**
 - ▶ Ultimul nod din ierarhie. Nu are noduri copil
 - ▶ Definește comportamentul pentru primitive
- ▶ **Trade-off**
 - ▶ Se încalcă principiul **Interface Segregation**. Nodurile frunză sunt obligate să implementeze metodele pentru obiectele compuse

➤ Intenție

Folosit atunci când avem nevoie să creăm ierarhii
Obiectele individuale și compuse sunt tratate uniform

➤ Motivație

Vezi exemplul.

➤ Aplicabilitate

Se dorește reprezentarea unei ierarhii de obiecte
Nu trebuie să existe distincții între obiectele individuale și compuse. Acestea sunt tratate uniform.

➤ Structură

Vezi diagrama UML generală.

➤ Participanți

Component – Declara interfața comună pentru obiecte

Leaf - Reprezintă obiectele frunză ale ierarhiei. Acestea implementează primitivele operațiilor

Comp - Definește obiectele compuse ale ierarhiei. Are o listă de obiecte Component. Implementează operațiile de management ale copiilor (Add/Remove/etc.). Se folosește de lista de copii pentru a implementa diferitele operații (Vezi metoda GetSize – exemplul cu fișiere sau metoda Operation – caz general)

➤ Colaborări

Clientul folosește interfața/clasa abstractă **Component** fară a știi dacă în spate se află un obiect **Leaf** sau **Composite**. Dacă obiectul este o frunză, atunci de obicei va returna un rezultat, dacă este un composite se va folosi de metodele obiectelor copil. (Vezi metoda GetSize – exemplul cu fișiere sau metoda Operation – caz general)

➤ Consecințe

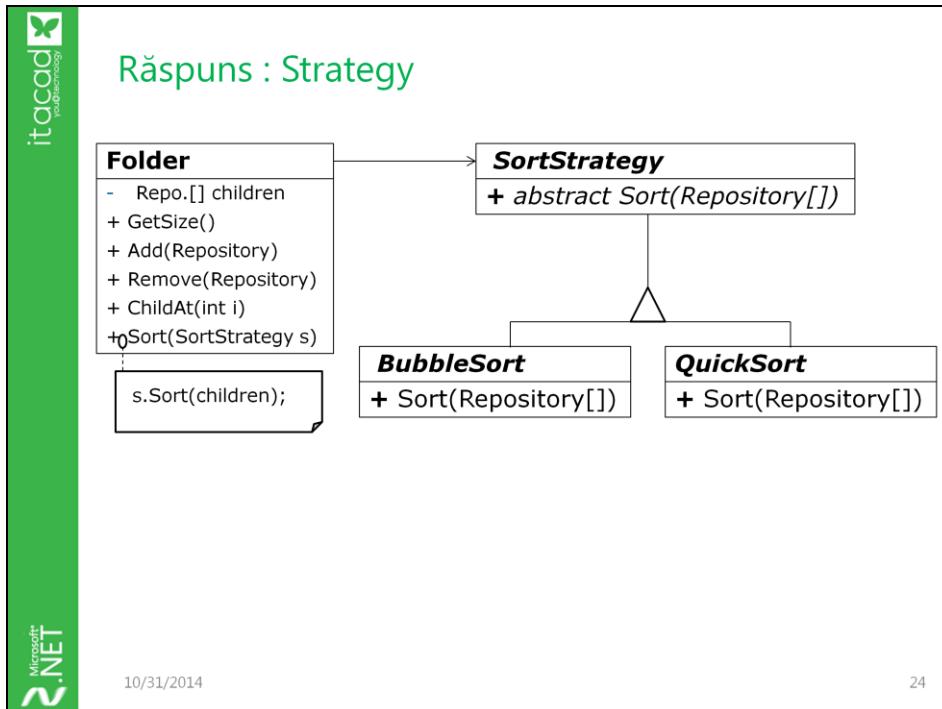
Se definește o structură care permite crearea ușoară a ierarhiilor și implementarea de operații pe acestea.
Nu trebuie să știm dacă avem un obiect frunză sau un obiect composite. Le putem trata uniform.
Avem o interfață simplă pentru client.

Se pot adauga noi tipuri în ierarhie fară a fi nevoie de modificări în codul existent sau informat Clientul.
Noul tip trebuie doar să moștenească/implementeze Component.

Trade-off - Se încalcă principiul **Interface Segregation**. Nodurile frunză sunt obligate să implementeze metodele pentru obiectele compuse.

Challenge 3

- Sortați vectorul de Repository-uri al unui Folder după numele acestora, făcând sortarea independentă de algoritmul de sortare



Răspunsul la acest challenge este design pattern-ul **Strategy**.

Implementare posibilă

```

class Folder : Repository {
    IList<Repository> children = new List<Repository>();
    ...

    public void sort(SortStrategy s)
    {
        s.Sort(children);
    }
    ...
}

abstract class SortStrategy
{
    public abstract void Sort(IList<Repository> r);
}

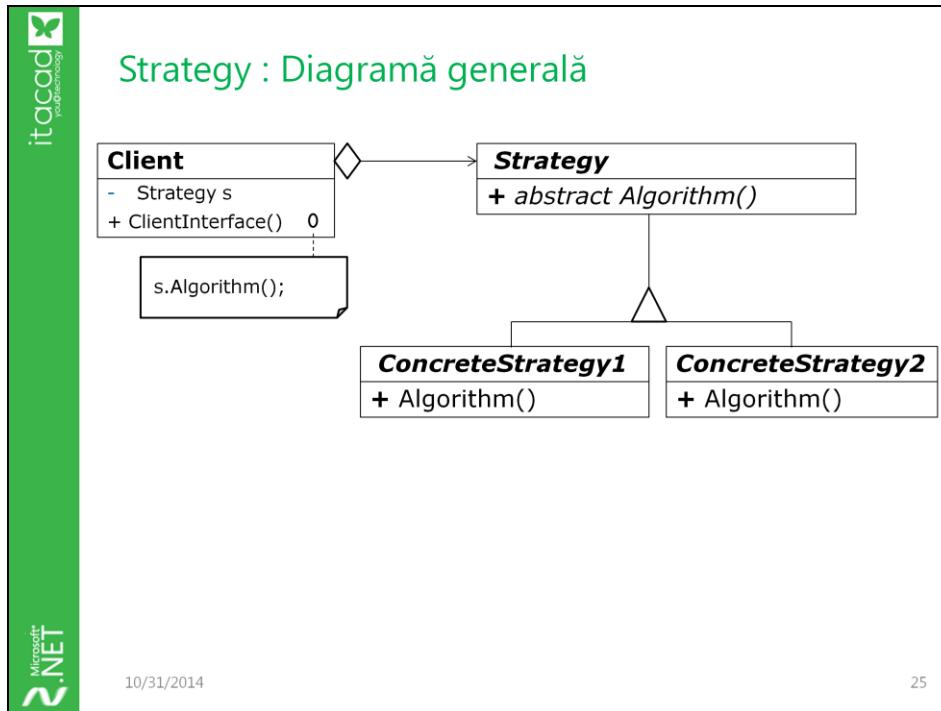
class BubbleSort : SortStrategy
{
    public override void Sort(IList<Repository> r) {
        bool ord;
        do
        {
            ord = true;
            for (int i = 0; i < r.Count - 1; i++)
            {
                if (r[i].Name.CompareTo(r[i + 1].Name) > 0)
                {
                    ...
                }
            }
        } while (!ord);
    }
}
  
```

```
        ord = false; ;
        Repository aux = r[i];
        r[i] = r[i + 1];
        r[i + 1] = aux;
    }
}
} while (!ord);
}

class QuickSort : SortStrategy
{
    public override void Sort(IList<Repository> r)
    {
        //...QuickSort
    }
}

class Test
{
    public static void Main(string[] args)
    {
        Folder root = new Folder("D:");
        //... create ierarchy

        root.Sort(new Bubblesort());
    }
}
```



Strategy

- ▶ Folosit atunci când avem nevoie să creăm o familie de algoritmi din care clientii să poată alege
- ▶ Ajută la eliminarea instrucțiunilor condiționale
- ▶ Permite schimbarea algoritmilor la runtime
- ▶ Poate fi implementat în 2 moduri
 - ▶ Domain coupling – clientii cunosc și diferitele strategii și aleg singuri ce strategie să folosească
 - ▶ No domain coupling – clientii nu au informații despre strategii însă sunt configurați să folosească o strategie anume

10/31/2014

26

➤ Intenție

Se definește o familie de algoritmi, se încapsulează fiecare dintre aceștia, se creează o interfață comună și se fac interschimbabili. Strategy va lăsa algoritmi să varieze independent de clientii care îi folosesc.

➤ Nume alternative

Policy

➤ Motivație

Vezi exemplul.

➤ Aplicabilitate

Mai multe clase diferă numai prin comportament.

Avem nevoie de variante diferite de implementare ale unui algoritm

Avem structuri condiționale prin care se aleg comportamente diferite

➤ Structură

Vezi diagrama UML

➤ Participanți

Client – Utilizează strategia. Există două tipuri de clienti:

1. Clientii care cunosc diferitele strategii și sunt capabili să aleagă ce strategie să folosească (**Domain coupling**). Aceștia de obicei au o agregă de obicei o strategie și stiu să instanțeze și să folosească o strategie.
2. Clientii care nu au cunoștiință de diferitele strategii (**No domain coupling**). Aceștia au de obicei un mod de a configura ce strategie să fie folosită sau primesc o strategie din afară. Aceștia pot să nu țină o referință internă la un obiect strategy (pot să nu folosească agregarea).

Existența celor două tipuri de clienti este motivul pentru care diagrama generală diferă de cea particulară exemplului cu sortarea. Exemplul cu sortarea folosește **No domain coupling**.

Strategy – definește interfața comună a algoritmilor

ConcreteStrategy – Implementează o variantă a algoritmului

➤ **Colaborări**

Clientul nu îndeplinește acțiunea cerută, el o va retrimit request-ul obiectului Strategy.
În funcție de tipul de cuplare, clientul poate instația sau primi obiectul Strategy.

➤ **Consecințe**

Se elimină instrucțiunile condiționale.

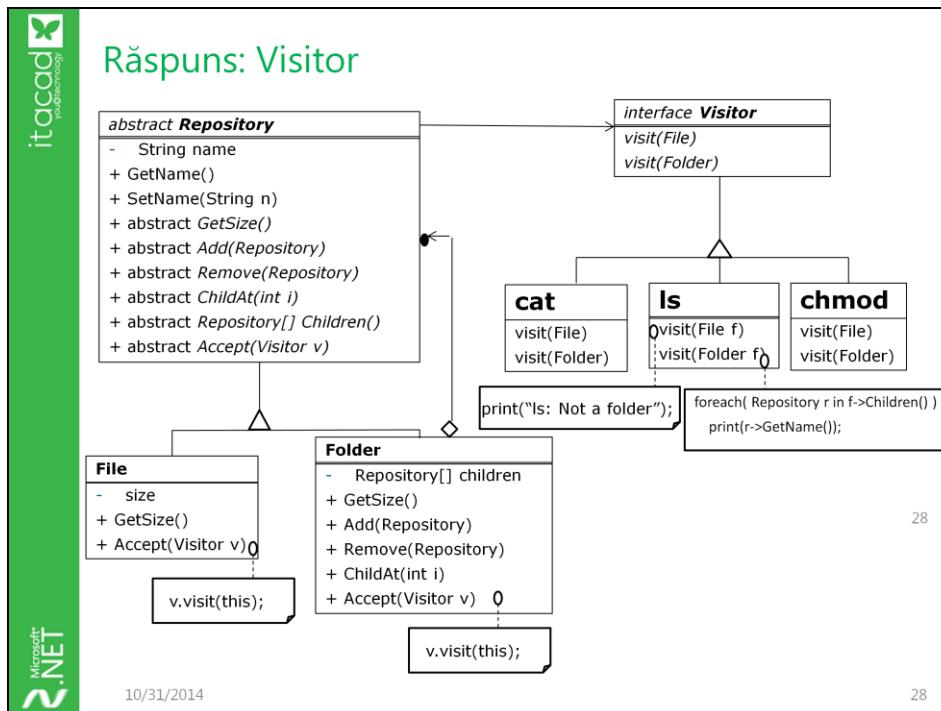
Algoritmii se pot inter schimba la runtime

Domain coupling – se creează o cuplare strânsă între client și strategii. Dacă adăugăm un nou tip de strategie, clientul va trebui informat despre acest lucru

No domain coupling – cineva (o altă clasă/un fișier de configurare) tot va trebui să știe care este diferența dintre strategii și când trebuie folosite

Challenge 4

- Încercați să găsiți o soluție de implementare astfel încât pentru comenzi de tipul ls, cat, chmod, mv, cp, etc.
- Implementarea trebuie să permită adăugarea unui număr mare



Răspunsul la acest challenge este design pattern-ul **Visitor**.

Implementare posibilă

```

interface Visitor
{
    void visit(Folder f);
    void visit(File f);
}

class ls : Visitor
{
    public void visit(Folder f)
    {
        Console.WriteLine(f.Name);
        foreach (Repository r in f.Children())
            Console.WriteLine('\t' + r.Name);
    }

    public void visit(File f)
    {
        Console.WriteLine(f.Name + " is not a folder");
    }
}

abstract class Repository
{
    ...
    private string name;
}
  
```

```
public string Name
{
    get { return name; }
}

public abstract void Accept(Visitor v) ;
...
}

class Folder : Repository {
    IList<Repository> children = new List<Repository>();

    public IList<Repository> Children
    {
        get { return children; }
    }

    public override void Accept(Visitor v)
    {
        v.Visit(this);
    }
}

class File : Repository
{
    ...
    public override void Accept(Visitor v)
    {
        v.Visit(this);
    }
    ...
}

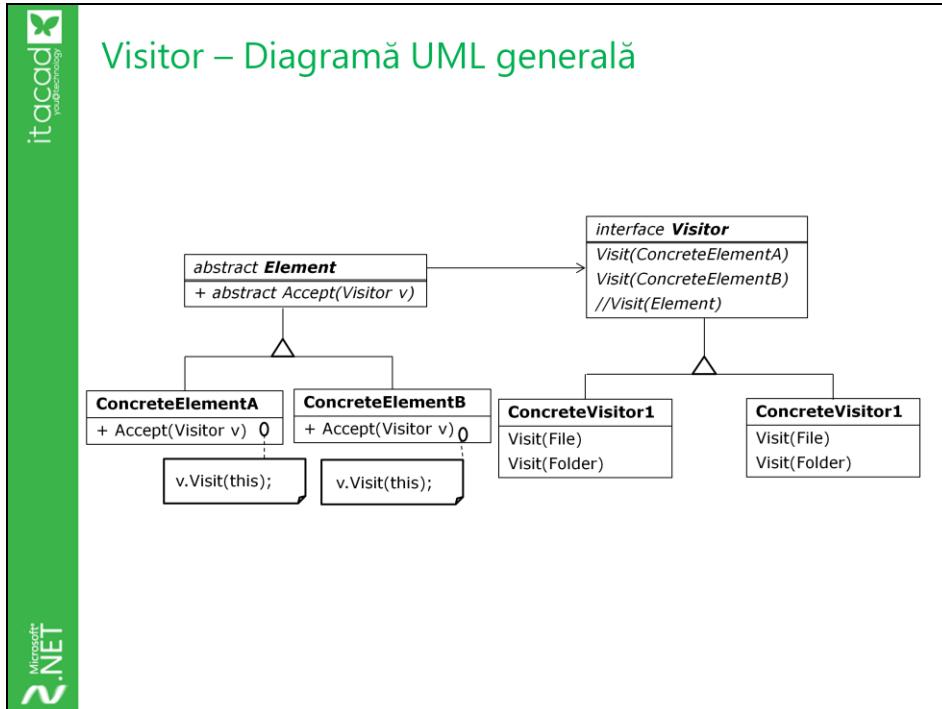
class Test
{
    static void Main(string[] args)
    {
        Visitor operation = new ls();
        Repository D = new Folder("D:");
        Repository important = new File("Examen", 20);

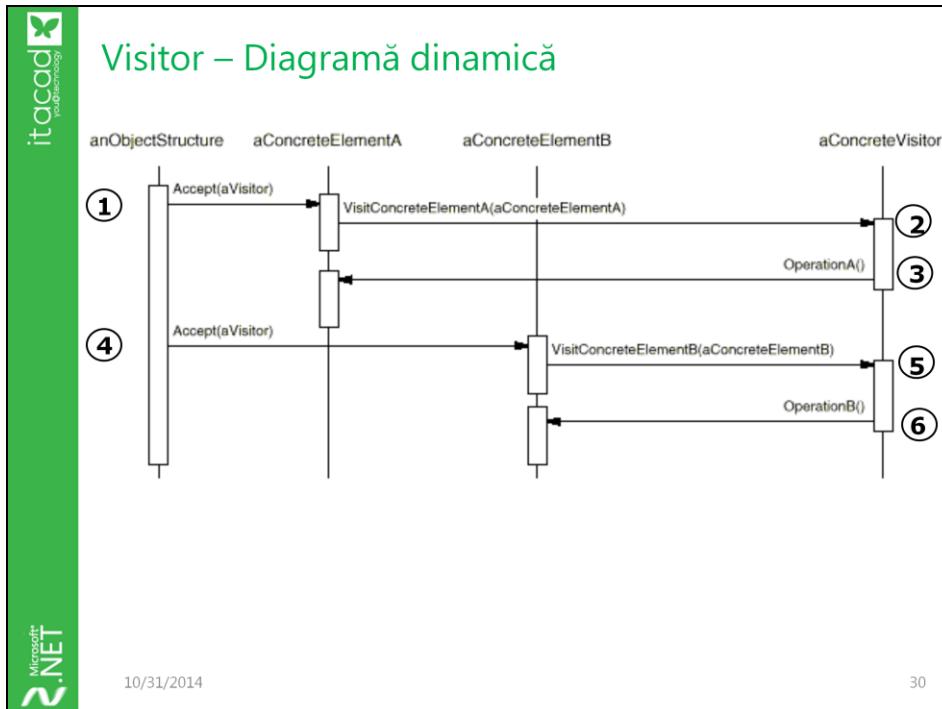
        D.Add(important);
        D.Add(new Folder("Referat"));
        D.Add(new File("Fisier", 15));

        Console.WriteLine("Visiting Folder");
        D.Accept(operation);
        Console.WriteLine();

        Console.WriteLine("Visiting File");
    }
}
```

```
important.Accept(operation);  
Console.ReadKey();  
}
```





Pe slide este prezentată diagrama dinamică a **Visitor** pattern-ului. Aceasta arată ordinea în care se fac apelurile diferitelor metode din obiecte participante. În continuare vom explica fiecare pas/apel în parte, pentru a înțelege ce anume face ca acest design pattern să fie o soluție extrem de puternică. Ideea acestui design pattern se folosește foarte mult de sistemul de tipuri, de proprietatea limbajului C# de a fi puternic tipat și de facilitatea de supraîncărcare oferită de acesta.

1. Având o structură `anObjectStructure` de Element, apelăm `anObjectStructure.Accept(Visitor v)`. Nimic magic până aici. Apelul `Accept` se va duce pe lanțul de moștenire până în ultima suprascriere determinând astfel tipul concret al elementului. În exemplul de pe slide acesta este un element de tipul `ConcreteElement A`.
2. Apelul `v.Visit(this)`: Știind în acest moment tipul concret al obiectului, se va apela metoda `Visit(ConcreteElementA)`, deoarece "this" are același tip ca și obiectul.
3. Apelul metodei `Visit(Concrete ElementA)` va fi îndeplinit deasemea de ultima suprascriere a metodei, determinând astfel tipul concret al visitor-ului. Se va executa codul acestei implementări. Pe slide se apelează o metodă oarecare `OperationA` a obiectului cu tip `ConcreteElementA`. ATENȚIE: În acest moment putem apela orice metodă publică a clasei `ConcreteElementA`, deoarece știm sigur că acesta este tipul obiectului.

Acest mecanism se numește **double dispatch**. Practic comportamentul și rezultatul apelurilor depinde de tipurile concrete de la runtime ale obiectelor implicate.

Pentru a înțelege mai bine ce înseamnă double dispatch, să studiem puțin **single dispatch-ul**. Single dispatch-ul este simplul apel al metodelor virtuale. Cuplarea între apelul metodei și implementarea concretă va depinde de tipul obiectului de la runtime și va fi dinamică. Double dispatch-ul apare ca urmare a apelurilor `Accept(v)` și `v.Visit(this)`. În acest caz, rezultatul apelului va fi determinat la runtime de tipurile concrete ale celor două – de unde double – obiecte. După cum puteți observa, nu s-a folosit niciun cast sau verificare de tipuri.

Visitor

- ▶ Folosit atunci când vrem să implementăm operații pe structuri/ierarhii ce depind de tipurile concrete din ierarhie (se apelează metode ale clasei concrete)
- ▶ Ne permite să implementăm oricâte operații fără a modifica clasele asupra căroră se execută operațiile (se evită poluarea interfețelor clasei cu câte o metodă pentru fiecare operație)
- ▶ **Trade-off** putem adăuga cu ușurință noi operații, dar este greu să adaugăm noi tipuri în ierarhie

➤ Intenție

Folosit atunci când vrem să implementăm operații pe structuri/ierarhii ce depind de tipurile concrete din ierarhie (se apelează metode ale clasei concrete).

Dorim să adăugăm noi operații, fără a modifica clasele existente.

➤ Motivatie

Vezi exemplul cu operațiile pe sistemul de fișiere.

➤ Aplicabilitate

Se folosește atunci când există multe tipuri într-o ierarhie și vrem să definim multe operații asupra ierarhiei, dar vrem deosemenea să folosim și metode ale claselor concrete nu doar cele ale interfeței comune.

Ştim că numărul operațiilor poate crește oricând, dar numărul tipurilor din ierarhie se modifică foarte greu (ideal de loc).

➤ Structură

Vezi diagrama UML.

➤ Participanți

- Visitor definește câte o metodă Visit pentru fiecare tip concret din ierarhie. Opțional poate defini și implementa o metodă Visit(Element), dând astfel o implementare default pentru noile tipuri ce apar în ierarhie. Avantajul este că nu vom mai fi obligați să implementăm metoda Visit pentru noul tip aparut în fiecare visitor concret, dezavantajul este că nu vom mai fi anunțați de compilator dacă am dorit să implementăm operația în fiecare visitor concret, dar am omis să facem acest lucru în anumite locuri. În acest caz (cazul din urmă) putem avea comportamente neașteptate, deoarece noi credeam că se va întâmpla un lucru, când de fapt vedem rezultatul implementării default (de obicei un mesaj afișat sau o excepție aruncată).

- Elementul – definește o metodă Accept abstractă
- ConcreteElement – Implementează metoda Accept apelând metoda Visit a obiectului primit prin intermediul parametrului asupra sa

➤ Colaborări

[Vezi diagrama dinamică](#)

➤ **Consecințe**

Adăugăm cu ușurință noi operații.

Nu poluăm interfețele claselor cu metode pentru fiecare operație.

Double dispatch

Putem avea implementări complexe pentru operații, visitorii putând reține starea

Greu de adăgat noi tipuri concrete – trebuie implementată metoda Visit în fiecare visitor concret pentru noul tip din ierarhie

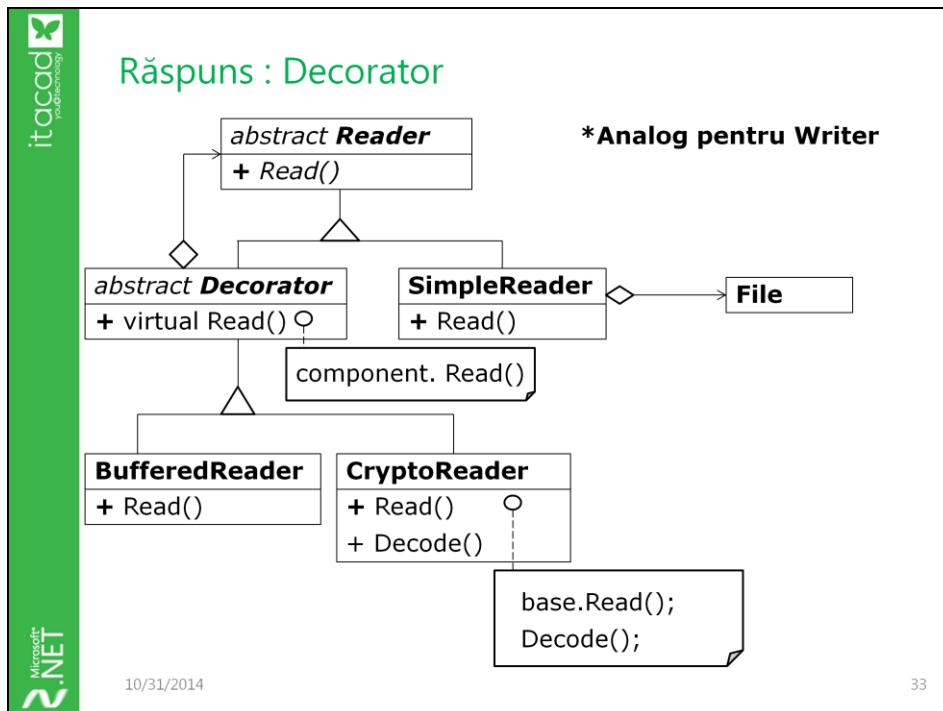
S-ar putea ca unele operații să nu aibă sens pentru anumite tipuri – Încălcarea a principiului **Interface Segregation**

S-ar putea să fie nevoie să definim metode adiționale de get – Spargerea încapsulării

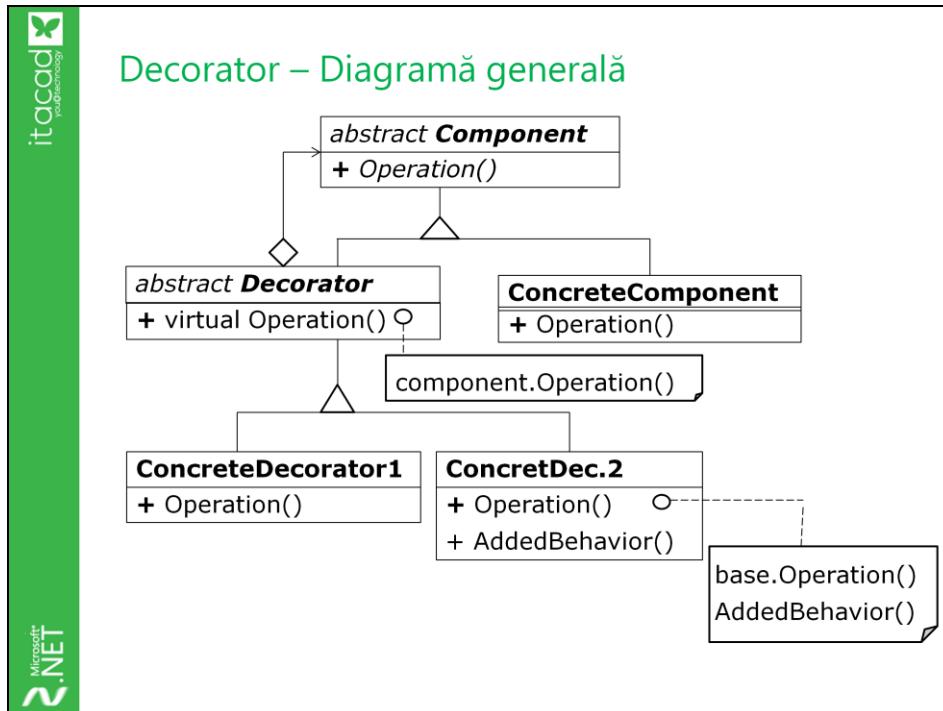
Challenge 4

- ▶ Presupunând că obiectele de tip File au următoarele metode:
 - ▶ byte ReadByte()
 - ▶ void WriteByte(byte)

Implementați un sistem de scriere și citire astfel încât să putem face atât scrierea cât și citirea în moduri diferite (simplu, buffered, criptat, compresat, etc.)



Răspunsul la acest challenge este design pattern-ul **Decorator**.



Decorator

- ▶ Folosit atunci când dorim să adăugăm/luăm responsabilități unui obiect în mod dinamic
- ▶ Este o alternativă flexibilă la simpla moștenire
- ▶ Datorită modului în care se comportă mai poartă numele de **Wrapper**

35

➤ Intenție

Folosit atunci când dorim să adăugăm/luăm responsabilități unui obiect în mod dinamic

➤ Denumiri alternative

Wrapper

➤ Motivație

Vezi exemplul cu operațiile pe sistemul de fișiere.

➤ Aplicabilitate

Vrem să putem atât adăuga cât și lua responsabilități unui obiect. Dintr-un motiv sau altul moștenirea nu este o soluție fiabilă.

➤ Structură

Vezi diagrama UML.

➤ Participanți

- **Component** – definește interfața comună pentru obiectele cărora vrem să le adăugăm dinamic responsabilități
- **ConcreteComponent** – definește obiectul căruia vrem să-i adăugăm responsabilități dinamic. Acest obiect trebuie instantiat înaintea decoratorilor și trebuie dat acestora (de obicei în constructor) pentru a fi împachetat
- **Decorator** - Menține o referință la un obiect Component (acesta poate fi după caz un alt decorator sau chiar un ConcreteComponent).
- **ConcreteDecorator** – adaugă noi responsabilități componentelor

➤ Colaborări

Decoratorul retrimitre requesturile Componentului pe care îl agregă și poate să facă propriile operații înainte și/sau după retrimitere.

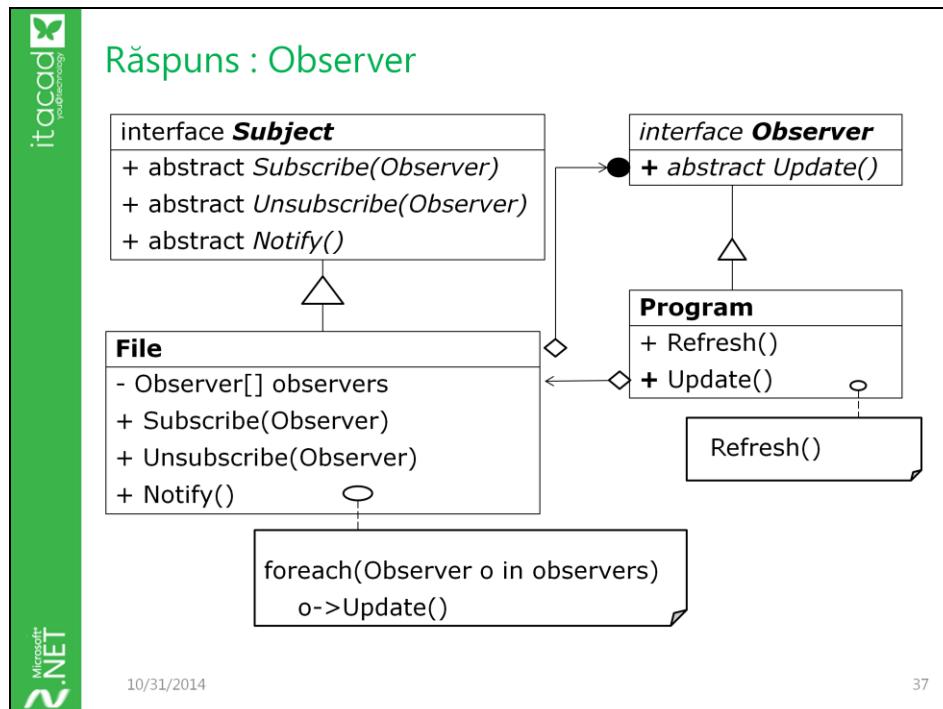
➤ Consecințe

Avem o alternativă la moștenire pentru adăugarea de funcționalități

Odată împachetat un obiect, nu mai putem accesa metodele specifice ale acestuia. Vom privi obiectul prin intermediul Decoratorului până când decidem să-i lăum responsabilitățile adiționale

Challenge 5

- ▶ Implementați un sistem astfel încât aplicațiile ce urmăresc un fișier să primescă un fișier în momentul în care acestea se modifică
- ▶ Presupunere: Un program poate deschide un singur fișier la un moment dat



Răspunsul la acest challenge este design pattern-ul **Observer**.

```

delegate void Notification(File f);

class File : IObservable<File>
{
    IList<IObserver<File>> observers = new List<IObserver<File>>();
    event Notification writeEvent;

    private class Unsubscriber : IDisposable
    {
        private File parent;
        private Notification delegat;
        private IObserver<File> observer;

        public Unsubscriber(File parent, IObserver<File> observer, Notification handler)
        {
            this.parent = parent;
            this.delegat = handler;
            this.observer = observer;
        }

        public void Dispose()
        {
            if (parent.observers != null && parent.observers.Contains(observer))
            {
                parent.observers.Remove(observer);
                parent.writeEvent -= delegat;
            }
        }
    }
}
  
```

```
        }

    }

    public IDisposable Subscribe(IObserver<File> observer)
    {
        if (!observers.Contains(observer))
        {
            Notification n = new Notification(observer.OnNext);
            writeEvent += n;
            return new Unsubscriber(this, observer, n);
        }

        return null;
    }

    public void Write(Byte b)
    {
        //...Writing to file
        if (writeEvent != null)
            writeEvent(this);
    }

    public File(String name)
    {
        this.name = name;
    }

    private String name;

    public string Name { get { return name; } }
}

class Program : IObserver<File>
{
    private File file;
    IDisposable unsubscriber;

    public void Refresh()
    {
        Console.WriteLine("The file " + file.Name + " has been modified from outside. Refreshing...");
    }

    public void OnCompleted()
    {
        unsubscriber.Dispose();
    }

    public void OnError(Exception error)
    {
```

```
        unsubscriber.Dispose();
    }

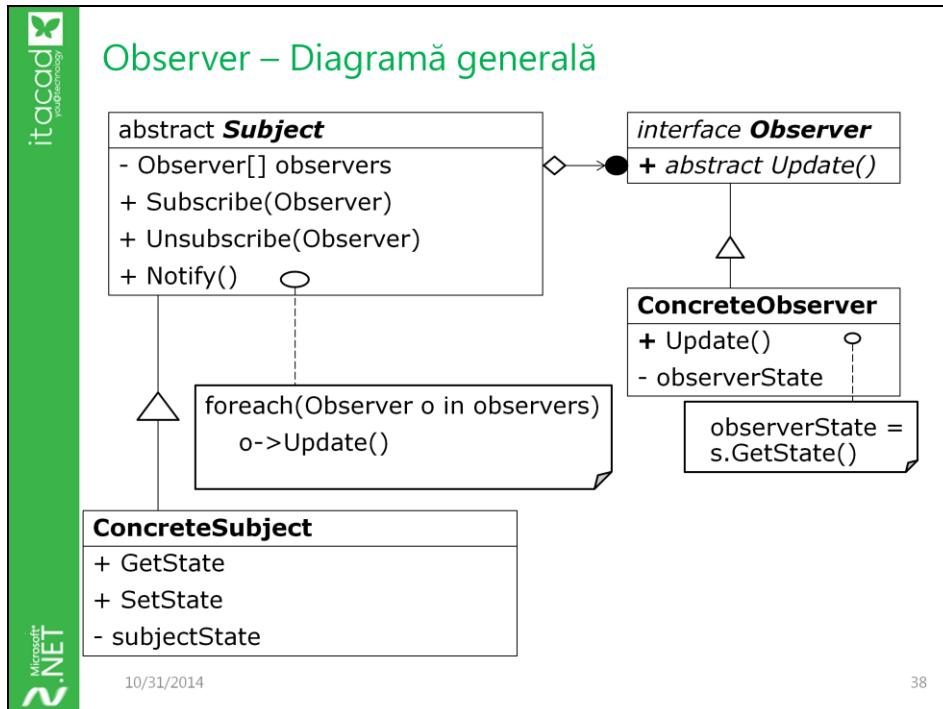
    public void Close()
    {
        unsubscriber.Dispose();
    }

    public Program(File f)
    {
        this.file = f;
        unsubscriber = f.Subscribe(this);
    }

    public void OnNext(File f)
    {
        Refresh();
    }
}

class Test
{
    public static void Main(String[] args)
    {
        File f = new File("Exemplu");
        Program program = new Program(f);

        f.Write(0);
        program.Close();
        Console.ReadKey();
    }
}
```



Observer

- ▶ Definește o dependință de tip one-to-many între obiecte, astfel încât în momentul în care un obiect își schimbă starea, toate obiectele care depinde de acesta sunt notificate și update automat
- ▶ .NET pune la dispoziție interfețele **IObserver<T>**, **IObservable<T>**, precum sistemul de evenimente și delegați ce ușurează implementarea acestui pattern
- ▶ **Loose-coupling** între Subiect și Observator
- ▶ **Referința din Subiect poate împiedica Garbage Collectorul să distrugă obiectul!**

10/31/2014

39

➤ Intenție

Definește o dependință de tip one-to-many între obiecte, astfel încât în momentul în care un obiect își schimbă starea, toate obiectele care depinde de acesta su notificate și update automat.

➤ Nume alternative

Publish-Subscribe

➤ Motivație

Vezi exemplul.

➤ Aplicabilitate

Design pattern-ul este aplicabil ori de câte ori o abstractizare are două aspecte, unul dependent de celălalt și trebuie implementat un sistem de notificări între acestea.

➤ Structură

Vezi diagrama UML.

Adițional putem adăuga parametrii metodei Update a observer-ului, astfel încât acesta să primească obiectul care trimite notificarea și alte informații de care are nevoie. Acest lucru este necesar în momentul în care un observer se poate abona la mai mulți subiecți.

➤ Participanți

Subject – definește o interfață pentru abonare și dezabonarea observer-ilor. Agregă mai multe instanțe de observatori. Implementează notificarea observatorilor.

Observer – definește interfața prin care primește notificări

ConcreteSubject – trimite efectiv notificările în momentul în care starea sa se modifică

ConcreteObserver – implementează interfața Observer pentru a-și reactualiza starea în aşa fel încât să fie consistentă cu a subiectului

➤ Colaborări

De câte ori intervine o modificare, subiectul concrete va anunța toti observatorii abonați astfel încât aceștia să se

poate reactualiza. Observeri pot cere noua stare a subiectului pentru a vedea cum să se actualizeze.

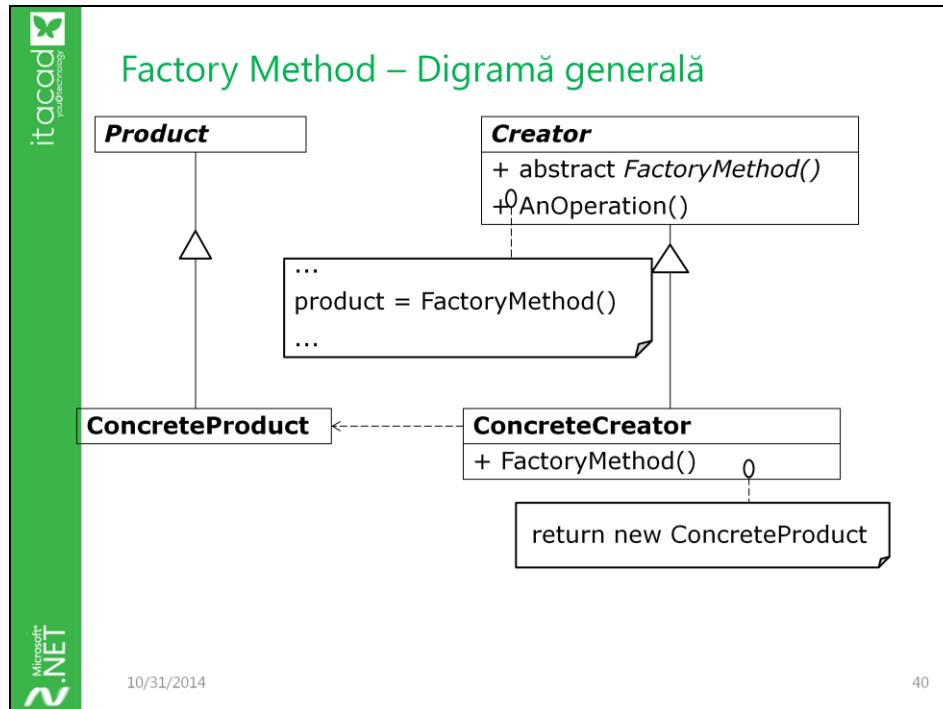
➤ Consecințe

Loose-coupling între subiecți și observatori. Subiectul nu știe nimic despre tipurile de observeri. Acesta trimit notificările în mod uniform. Tot ce cunosc Subiecții este ce observeri s-au abonat.

Putem adăuga și șterge observatori în orice moment.

IMPORTANT! Putem crea memory leak-uri și chiar avea un comportament neașteptat dacă uitam să dezabonăm observatorii de la subiecti. Subiecții vor continua să țină referința către observator și vor trimit în continuare notificări către aceștia. Cea mai simplă soluție pentru a ne asigura că nu avem astfel de probleme este să nu uităm să apelăm metoda de unsubscribe. Totuși, nu este întotdeauna evident unde trebuie să facem acest apel într-un program complex. .NET pune la dispoziție o clasă numită WeakReference care permite Garbage Collectorului să distrugă un obiect, dacă toate referințele la acesta sunt făcute prin intermediul acestei clase. Folosirea evenimentelor și delegațiilor cu WeakReference este peste programa acestui curs, totuși dacă doriți mai multe informații vă invităm să citiți acest articol:

<http://diditwith.net/2007/03/23/SolvingTheProblemWithEventsWeakEventHandlers.aspx>



Factory Method

- ▶ Creăm o interfață pentru instanțierea obiectelor, dar lăsăm clasele derivate să decidă pe cine să instanțieze
- ▶ Îl folosim atunci când o clasă nu poate anticipa ce obiecte să instanțieze
- ▶ Conectează ierarhii paralele de clase

10/31/2014

41

➤ Intenție

Creăm o interfață pentru instanțierea obiectelor, dar lăsăm clasele derivate să decidă pe cine să instanțieze

➤ Nume alternative

Virtual constructor

➤ Aplicabilitate

Îl folosim atunci când o clasă nu poate anticipa ce obiecte să instanțieze, dar suntem siguri că tipurile derivate se vor ocupa de acest lucru.

➤ Structură

Vezi diagrama UML

➤ Participanți

Product – definește o interfață obiectelor pe care FactoryMethod le crează

ConcreteProduct – Implementează interfața product

Creator – declară metoda FactoryMethod, ce va returna un obiect de tip Product. Poate apela această metodă el însuși.

ConcreteCreator – suprascrie FactoryMethod pentru a returna o instanță a clasei ConcreteProduct

➤ Colaborări

Creatorul se bazează pe clasele derivate pentru a implementa FactoryMethod și a returna obiectul derivat din Product corespunzător.

➤ Consecințe

Putem conecta ierarhii paralele de clase. Se oferă o metodă elegantă de abstractizare și decuplare a două ierarhii de atfel puternic cuplate.

Best Practices

- Înainte de a ne pune problema codului reutilizabil, avem nevoie de cod utilizabil
- Încercați pe cât posibil să respectați principiile
- Evitați instructiunile de tip if sau switch în funcție de tipul obiectelor
- Nu faceți overdesign
- Faceți design într-un mod iterativ. Nu încercați să faceți de la început design-ul până în cele mai mici detalii. Acesta trebuie făcut pe măsură ce codul avansează.
- Nu există design-ul perfect. Căutați cel mai bun trade-off.
- Încercați metoda de dezvoltare Test Driven Development

Review

- Principii de design
- Ce este un design pattern?
- Câteva design pattern-uri importante

