**„ALEXANDRU IOAN CUZA” UNIVERSITY OF IASI**

**FACULTY OF ECONOMICS AND BUSINESS ADMINISTRATION**

**SOFTWARE DEVELOPMENT AND BUSINESS INFORMATION SYSTEMS**

**MANAGEMENT SYSTEM FOR RESEARCH PUBLISHING**

**Part II: Applying design patterns**

**Scientific coordinator,**

**Ph.D. Professor Florin Dumitriu**

**Students,**

**Ignea Vlad**

**Nistor Carmen**

**TABLE OF CONTENTS**

[1st Problem 3](#_Toc440481169)

[1.1. Applying Composite pattern 3](#_Toc440481170)

[1.1.1. Defining the problem 3](#_Toc440481171)

[1.1.2. Solution without applying the Composite pattern 3](#_Toc440481172)

[1.1.3. Disadvantages of the solution proposed 4](#_Toc440481173)

[1.1.4. Solution when applying the Composite pattern 4](#_Toc440481174)

[1.1.5. Comments regarding the application of Composite pattern 7](#_Toc440481175)

[1.2. Applying Abstract Factory 7](#_Toc440481176)

[1.2.1. Defining the problem 7](#_Toc440481177)

[1.2.2. Solution without applying Abstract Factory 7](#_Toc440481178)

[1.2.3. Disadvantages of the solution proposed 8](#_Toc440481179)

[1.2.4. Solution when applying the Abstract Factory 8](#_Toc440481180)

[1.2.5. Comments regarding the application of Abstract Factory 10](#_Toc440481181)

[1.3. Applying Strategy pattern 10](#_Toc440481182)

[1.3.1. Defining the problem 10](#_Toc440481183)

[1.3.2. Solution without applying the Strategy pattern 10](#_Toc440481184)

[1.3.3. Disadvantages of the solution proposed 11](#_Toc440481185)

[1.3.4. Solution when applying the Strategy pattern 11](#_Toc440481186)

[1.3.5. Comments regarding the application of Strategy pattern 12](#_Toc440481187)

[2nd Problem 13](#_Toc440481188)

[2.1. Applying Factory method pattern 13](#_Toc440481189)

[2.1.1. Defining the problem 13](#_Toc440481190)

[2.1.2. Solution without applying the Factory Method pattern 13](#_Toc440481191)

[2.1.3. Disadvantages of the solution proposed 14](#_Toc440481192)

[2.1.4. Solution when applying the Factory Method pattern 14](#_Toc440481193)

[2.1.5. Comments regarding the application of Factory Method pattern 15](#_Toc440481194)

[2.2. Applying Chain of responsibility pattern 15](#_Toc440481195)

[2.2.1. Defining the problem 15](#_Toc440481196)

[2.2.2. Solution without applying the Chain of responsibility pattern 16](#_Toc440481197)

[2.2.3. Disadvantages of the solution proposed 16](#_Toc440481198)

[2.2.4. Solution when applying the Chain of responsibility pattern 16](#_Toc440481199)

[2.2.5. Comments regarding the application of Chain of responsibility pattern 17](#_Toc440481200)

[2.3. Applying Facade pattern 17](#_Toc440481201)

[2.3.1. Defining the problem 17](#_Toc440481202)

[2.3.2. Solution without applying the Facade pattern 17](#_Toc440481203)

[2.3.3. Disadvantages of the solution proposed 17](#_Toc440481204)

[2.3.4. Solution when applying the Facade pattern 18](#_Toc440481205)

[2.3.5. Comments regarding the application of Facade pattern 18](#_Toc440481206)

# 1st Problem

*The big problem that needs to be solved with three design patterns in the management system for research publishing regarding the discounted given by the foundation to the authors at the payment of articles according to several different criteria!*

a) The foundation has multiple discount strategies according to different criteria – we solve the problem applying the composite pattern;

c) A new type of discount can be added at any time in the system – we solve the problem applying Abstract factory.

b) Every discount applied needs a distinctive interface – we solve the problem applying the strategy pattern;

## 1.1. Applying Composite pattern

### 1.1.1. Defining the problem

In the management application for research publishing we need to resolve the problem of handling the case of multiple conflicting pricing policies for authors when they pay the articles. The foundation has the following policies regarding the prices:

1) 20% off for authors which are PhD students;

2) 30% off if an author pays more than one article;

3) 15% off if an author pays for an article to participate on a conference and not to publish it directly in one of the journals.

Because there can exist multiple co-existing discount strategies one payment may have several pricing strategies meaning to one payment can apply several discounts. Another point to note is that a pricing strategy can be related to the type of author (eg. PhD. Student). Similarly, a pricing strategy can be related to the type of article send it by the author (eg. The author can choose either to participate in a conference with the article or to publish ii directly in one of the journals).

### 1.1.2. Solution without applying the Composite pattern

In order to resolve the 1st issue of the multiple discounts without applying the pattern we can implement in the class Invoice every type of discount.

### 1.1.3. Disadvantages of the solution proposed

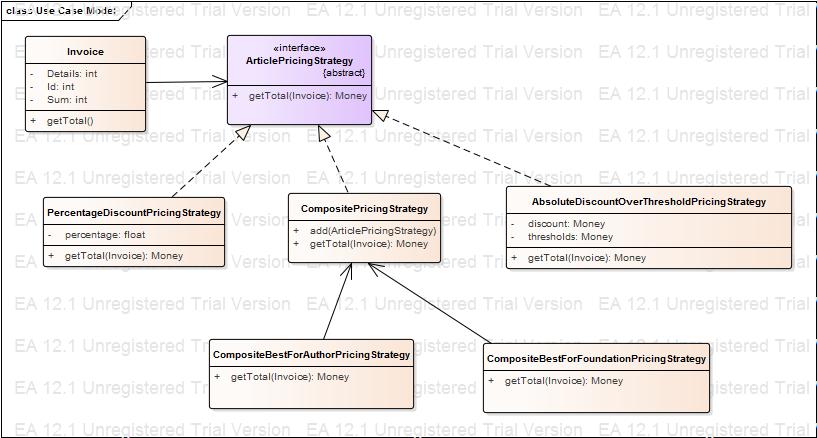
Every time a new type of discount appears we have to introduce it manually (we write cod).

### 1.1.4. Solution when applying the Composite pattern

Name of the pattern applied: Composite

Problem: How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object.

Solution: we defined classes for composite and atomic objects so that they implement the same interface.



public class Invoice {

private int Details;

private int Id;

private int Sum;

public ArticlePricingStrategy m\_ArticlePricingStrategy;

public Invoice(){

}

public void finalize() throws Throwable {

}

public getTotal(){

}

}//end Invoice

public interface ArticlePricingStrategy {

/\*\*

\*

\* @param X

\*/

public Money getTotal(Invoice X);

}

public class PercentageDiscountPricingStrategy implements ArticlePricingStrategy {

private float percentage;

public PercentageDiscountPricingStrategy(){

}

public void finalize() throws Throwable {

}

/\*\*

\*

\* @param X

\*/

public Money getTotal(Invoice X){

return null;

}

}//end PercentageDiscountPricingStrategy

public class AbsoluteDiscountOverThresholdPricingStrategy implements ArticlePricingStrategy {

private Money discount;

private Money thresholds;

public AbsoluteDiscountOverThresholdPricingStrategy(){

}

public void finalize() throws Throwable {

}

/\*\*

\*

\* @param X

\*/

public Money getTotal(Invoice X){

return null;

}

}//end AbsoluteDiscountOverThresholdPricingStrategy

public class CompositeBestForAuthorPricingStrategy {

public CompositePricingStrategy m\_CompositePricingStrategy;

public CompositeBestForAuthorPricingStrategy(){

}

public void finalize() throws Throwable {

}

/\*\*

\*

\* @param X

\*/

public Money getTotal(Invoice X){

return null;

}

}//end CompositeBestForAuthorPricingStrategy

public class CompositeBestForFoundationPricingStrategy {

public CompositePricingStrategy m\_CompositePricingStrategy;

public CompositeBestForFoundationPricingStrategy(){

}

public void finalize() throws Throwable {

}

/\*\*

\*

\* @param X

\*/

public Money getTotal(Invoice X){

return null;

}

}//end CompositeBestForFoundationPricingStrategy

public class CompositePricingStrategy implements ArticlePricingStrategy {

public CompositePricingStrategy(){

}

public void finalize() throws Throwable {

}

/\*\*

\*

\* @param X

\*/

public add(ArticlePricingStrategy X){

}

/\*\*

\*

\* @param X

\*/

public Money getTotal(Invoice X){

return null;

}

}//end CompositePricingStrategy

### 1.1.5. Comments regarding the application of Composite pattern

The composite classes such as CompositeBestForAuthorPricingStrategy inherit an attribute PricingStrategy that contains a list of more ArticlePricingStrategy objects. The outer composite object contains a list of inner objects, and both the outer and inner objects implement the same interface.

Thus, we can attach either a composite CompositeBestForAuthorPricingStrategy object (which contains other strategies inside of it) or an atomic PercentageDiscountPricingStrategy object to the Inovice object and the Invoice does not know or care if its pricing strategy is an atomic or composite strategy (it looks the same for the Inovoice object).

## 1.2. Applying Abstract Factory

### 1.2.1. Defining the problem

The problem consists in the ability to add a new type of discount at any time in the system – we solve the problem applying Abstract factory.

### 1.2.2. Solution without applying Abstract Factory

Every time a new type of discount is

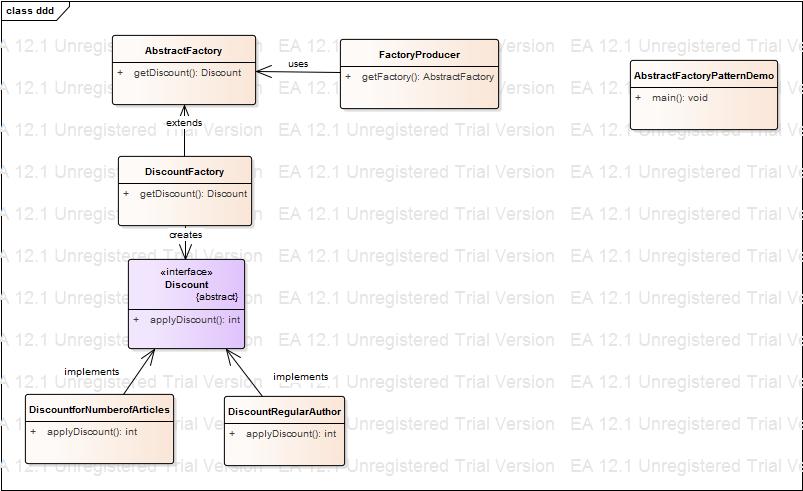
We should add the discount manually every time a new type of discount applies.

We should write code

### 1.2.3. Disadvantages of the solution proposed

The logic of creating the family of objects is not hidden being visible to the user

### 1.2.4. Solution when applying the Abstract Factory



public class AbstractFactory {

public AbstractFactory(){

}

public void finalize() throws Throwable {

}

public Discount getDiscount(){

return null;

}

}//end AbstractFactory

public class AbstractFactoryPatternDemo {

public AbstractFactoryPatternDemo(){

}

public void finalize() throws Throwable {

}

public void main(){

}

}//end AbstractFactoryPatternDemo

public interface Discount {

public int applyDiscount();

}

public class DiscountFactory {

private Discount m\_Discount;

private AbstractFactory m\_AbstractFactory;

public DiscountFactory(){

}

public void finalize() throws Throwable {

}

public Discount getDiscount(){

return null;

}

}//end DiscountFactory

public class DiscountforNumberofArticles {

private Discount m\_Discount;

public DiscountforNumberofArticles(){

}

public void finalize() throws Throwable {

}

public int applyDiscount(){

return 0;

}

}//end DiscountforNumberofArticles

public class DiscountRegularAuthor {

private Discount m\_Discount;

public DiscountRegularAuthor(){

}

public void finalize() throws Throwable {

}

public int applyDiscount(){

return 0;

}

}//end DiscountRegularAuthor

public class FactoryProducer {

private AbstractFactory m\_AbstractFactory;

public FactoryProducer(){

}

public void finalize() throws Throwable {

}

public AbstractFactory getFactory(){

return null;

}

}//end FactoryProducer

### 1.2.5. Comments regarding the application of Abstract Factory

Applying Abstract Factory we hide the logic of creating the family of objects.

Every factory class contains a factory method for each family of objects created.

## 1.3. Applying Strategy pattern

### 1.3.1. Defining the problem

The problem consists in resolving the selection of a certain discount depending on the situation at the moment of run-time when we don’t want to implement the algorithms and their logic to the client.

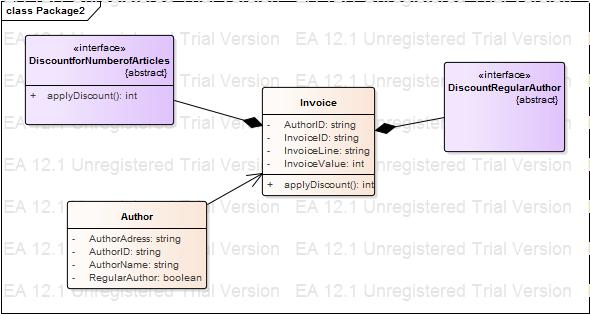
### 1.3.2. Solution without applying the Strategy pattern

We should treat every discount separately

### 1.3.3. Disadvantages of the solution proposed

Every time a new type of discount appears we have to write code for the new type.

### 1.3.4. Solution when applying the Strategy pattern



public class Author {

private string AuthorAdress;

private string AuthorID;

private string AuthorName;

private boolean RegularAuthor;

public Invoice m\_Invoice;

public Author(){

}

public void finalize() throws Throwable {

}

}//end Author

public interface DiscountforNumberofArticles {

public int applyDiscount();

}

public interface DiscountRegularAuthor {

}

public class Invoice {

private string AuthorID;

private string InvoiceID;

private string InvoiceLine;

private int InvoiceValue;

public DiscountforNumberofArticles m\_DiscountforNumberofArticles;

public DiscountRegularAuthor m\_DiscountRegularAuthor;

public Invoice(){

}

public void finalize() throws Throwable {

}

public int applyDiscount(){

return 0;

}

}//end Invoice

### 1.3.5. Comments regarding the application of Strategy pattern

It allows the implementation of several algorithms to apply discounts.

# 2nd Problem

The articles from the application are of several types. There has to exist the possibility to create a new type of article. Depending of the type of article and the field that each article has there has to be assigned a reviewer for articles. A reviewer is responsible for reviewing several articles. The system provides a single interface in order to select reviewers.

The problem of adding a new type of article will be resolved using the Factory method.

The problem of assigning reviewers will be resolved using Chain of responsibility.

In order to avoid the access of multiple interfaces for each class of reviewers we

The problem of accessing multiple classes of reviewers through a single interface will be resolved using Facade.

## 2.1. Applying Factory method pattern

### 2.1.1. Defining the problem

The articles submitted by the authors are of different types (methodological, theoretical, exploratory, case study etc.). In this case we will have several types of classes for each article which will implement the method submit().

The author can choose the type of article that he wants to submit from the form of the application and submits the article.

We will use factory method to resolve the problem of creating a new type of articles.

### 2.1.2. Solution without applying the Factory Method pattern

Considering the author’s choice the application will create an object methodological, theoretical, exploratory etc and will invoke method submit()

//author enters their articletype

//

If (articletype == "methodological")

{

// call methodologicaltype and it's methods

Methodological m = **new** methodological()

m.submit();

}

If (articletype == "theoretical")

// call theoreticaltype class and it's methods

Theoretical t = **new** theoretical()

t.(submit)

}

### 2.1.3. Disadvantages of the solution proposed

- A new type of article submitted involves modifying the code of application;

- The code has no security because the two classes and the methods involved can be seen.

### 2.1.4. Solution when applying the Factory Method pattern

Using Factory Method:

- we will create an interface for defining the submit() method, which will be implemented by the classes Methodological and Theoretical.

- we will create a class with a Factory role which will have a method with the purpose of “fabricating” methodological and theoretical objects;

- the class will be used between the client application and the two specific classes of the domain.

//

//interface

//

**public** **interface** iChoice

{

string submit()

}

//Article classes

//Methodological

**public** **class** clsMethodological:IChoice

{

#region IChoiceMmembers

**public** string submit()

{

**return** ("You choose methodological")

}

#endregion

}

//theoretical

**public** **class** clsTheoretical:IChoice

{

#region IChoiceMmembers

**public** string submit()

{

**return** ("You choose theoretical")

}

#endregion

}

//

//Factory Class

//

**public** **class** FactoryChoice

{

**static** **public** IChoice getChoiceObj(String cChoice)

{

IfChoice objChoice=**null**;

**if**(cChoice.toLower()=="methodological")

{

objChoice=newclasmethodological();

}

**else** **if** (c.Choice.toLower()=="theoretical")

{

objChoice=newclstheoretical();

}

**else**

{

objChoice=newInvalidChoice();

}

**return** objChoice

}

}

//Author class

IChoice objArticle;

objArticletype = FactoryChoice.getChoiceobj(txtChoice.Text.Trim());

MessageBox.Show(obArticle.Submit());

### 2.1.5. Comments regarding the application of Factory Method pattern

- The logic of creating objects is not exposed to the client part of the application;

- The reference to the new object created is done through an interface, the client will see and interact with a generic object;

- It avoids implementing the logic of choosing a class in the client part – it will involve a too high dependency;

- The code from the client part doesn’t changes when we introduce a new type of of object or the logic of choosing a class to instantiate changes;

- The modifications are localized in one class and one method.

## 2.2. Applying Chain of responsibility pattern

### 2.2.1. Defining the problem

To every article send to the foundation can be assigned multiple reviewers.

The problem of assigning reviewers to every article can be resolved using Chain of responsibility because there are multiple reviewers that can analyze an article

### 2.2.2. Solution without applying the Chain of responsibility pattern

Every time a new type of article is added from a different field of interest (geography, for example) we have to check the type of the article and assign the reviewer manually.

### 2.2.3. Disadvantages of the solution proposed

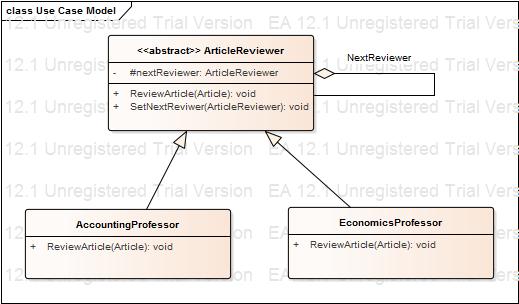
The appearance of another type of article will determine the manual selection of a reviewer to be assigned.

### 2.2.4. Solution when applying the Chain of responsibility pattern

The purpose of applying the Chain of responsibility pattern is to avoid the direct coupling between the expeditor of a request and the receiver of the request.

The client sends the request to a chain of objects without knowing which object will solve his request.

The objects from the chain decide themselves who will solve the request.



**class** AccountingProfessor: ArticleReviewer

{

publicoverrirde **void** ArticleReviewer(Article a)

**if** (a.ArticleDomain = "Accounting")

Console.WriteLine("Review of article"+a.ArticleDomain+"reviewed by AccountingProfessor");")

**else**

nextReviewer.ReviewArticle(a);

}

}

**class** EconomicsProfessor: ArticleReviewer

{

publicoverrirde **void** ArticleReviewer(Article a)

{

Console.WriteLine("Review of article"+a.ArticleDomain+"reviewed by EconomicsProfessor");")

**else**

nextReviewer.ReviewArticle(a);

}

}

### 2.2.5. Comments regarding the application of Chain of responsibility pattern

The abstract class Article reviewer is a parent for all the objects that can solve the requests

The variable NextReviewer is a reference to the next object in the chain

The method RevieArticle will be implemented by all child classes

The classes AccountingProfessor, EconomicsProfessor represent the classes which will review the articles. The method SetNetReviewer checks if the if the article can be reviewed by a certain class (accountingprofessors) and if not it sends it to another class.

## 2.3. Applying Facade pattern

### 2.3.1. Defining the problem

The problem: providing a single interface to access several classes of reviewers.

### 2.3.2. Solution without applying the Facade pattern

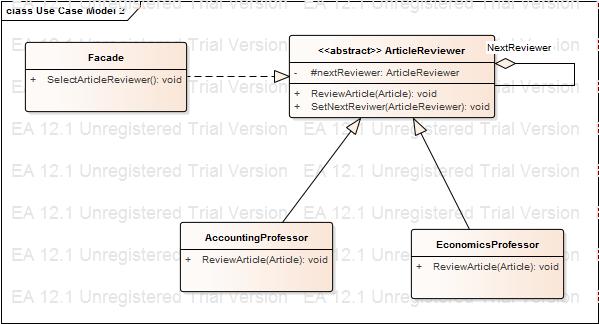
To create several interfaces for every type of reviewers.

### 2.3.3. Disadvantages of the solution proposed

Every time you add a new reviewer you have to create a specific interface.

When the Facade pattern is not applied the logic of specific methods should be implemented directly into articles.

### 2.3.4. Solution when applying the Facade pattern



Public class AccountingProfessor

{

Public int ReviewArticle(Article)

Public class EconomicsProfessor

{

Public int ReviewArticle(Article)

### 2.3.5. Comments regarding the application of Facade pattern

Applying facade we have a single simplified interface for every reviewer in the application;

We created a single access point in the services of the system;

The system becomes easier to be utilized, every reviewers access the same interface.