## horizontal line



Project Report

02.04.2021

**─  
DPR**

Stefan Rachev 3315517

Contents

[horizontal line 0](#_Toc68580955)

[Introduction 2](#_Toc68580956)

[Problem Description 2](#_Toc68580957)

[Justifications on selected patterns 2](#_Toc68580958)

[Abstract Factory pattern 2](#_Toc68580959)

[Observer pattern 2](#_Toc68580960)

[Command Pattern 2](#_Toc68580961)

[UML Diagram 3](#_Toc68580962)

[Describe the role of the classes of your solution in relation to general solution per pattern 3](#_Toc68580963)

[Abstract Factory pattern 3](#_Toc68580964)

[Command pattern 3](#_Toc68580965)

[Observer pattern 4](#_Toc68580966)

[Discuss the consequences of your solution (+) and (-) for each of the problem areas: Reusability, Extensibility and Maintainability 4](#_Toc68580967)

[Reusability 4](#_Toc68580968)

[Pros 4](#_Toc68580969)

[Cons 4](#_Toc68580970)

[Extensibility 5](#_Toc68580971)

[Pros 5](#_Toc68580972)

[Cons 5](#_Toc68580973)

[Maintainability 5](#_Toc68580974)

[Pros 5](#_Toc68580975)

[Cons 5](#_Toc68580976)

# 

# Introduction

This document's final DPR project is for a real-life Publisher, Distributor, and Customer interaction. A system in which consumers can subscribe and unsubscribe from a distributor which has a contract with publishers to sell newspapers and magazines to customers.

# Problem Description

The system should allow users to subscribe or unsubscribe to distributors to receive newspapers or magazines from contracted publishers to said, distributors. Three design patterns must be used to create the system. The design patterns chosen for this case were:

The Observer pattern is concerned with the relationship between customers and distributors. The Abstract factory pattern is concerned with the establishment of factories and the goods created by such factories. Finally, the Command pattern links the preceding design patterns by simulating the formation or termination of contracts between a distributor and a factory.

# Justifications on selected patterns

## Abstract Factory pattern

As mentioned above, this pattern deals with the creation of factories and the products they will provide. The pattern was a perfect fit for the generation of publisher factories and the type of products they provide. Also, the system needed a creational design pattern to increase the flexibility and reusability of the code. As an example, in the created system, there are two types of factories. One creates only sportsbooks and the other only fashion books.

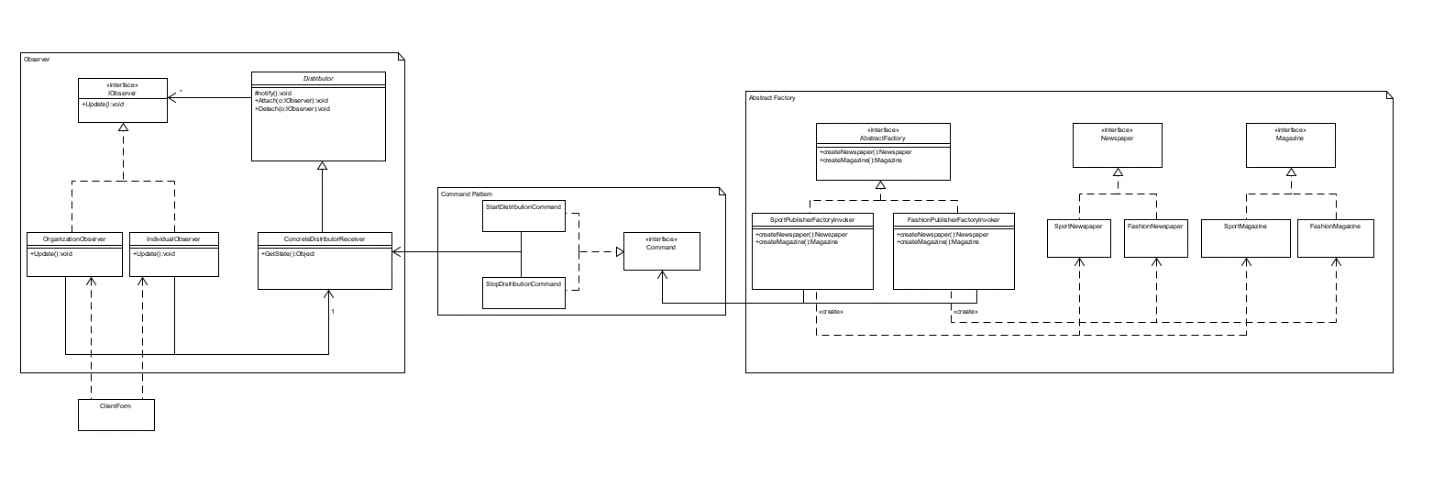
## Observer pattern

The Observer pattern was the first to be developed and served as the system's base. This pattern served as the basis for the whole structure. Customers would be able to subscribe to distributors. The distributors provide newspapers and/or magazines from publishers contracted to the them to the customers. This pattern defines a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing, in this case the distributors they have subscribed to. For this system the observer pattern was used with the pull mechanism.

## Command Pattern

The main goal of the command pattern was to allow distributors and publishers to establish contracts (or terminate them). By having the publishers as the invokers of the contract commands it allows for the distributors who play the role of receivers to establish or terminate contracts with publishers. And update the state of distributor. Informing a distributors client of the change. This pattern was necessary to be able to pass requests and change of state as method arguments.

# UML Diagram



Above you see the general UML version made before any code was written.

# Describe the role of the classes of your solution in relation to general solution per pattern

## Abstract Factory pattern

The interfaces Newspaper and Magazine are blueprint classes for a set of distinct but related products which make up a product family. In this case Newspaper and Magazine. The concrete product classes are various implementations of the product interfaces, grouped by variants. Each interface (Newspaper/Magazine) must be implemented in all given variants which are the different type of factories. In this case Sport and Fashion. The Abstract Factory interface declares a set of methods inherited by the concrete factories for creating different variation for the content of newspapers and magazines. The Sport and Fashion Factories are concrete factories that implement the creation methods of the Abstract factory class. Each concrete factory corresponds to a different variation of products and creates only those product variations. In this case the Sport factory makes only sport newspapers and magazines and the fashion one does fashion newspapers and magazines.

## Command pattern

The concrete factory classes (SportPublisherFactoryInvoker and FashionPublisherFactoryInvoker) are also invokers for the command pattern. They are responsible for initiating requests. They trigger those commands instead of sending the request directly to the receiver (ConcreteDistributorReceiver). The Command interface declares just a single method for executing the command. The concrete StopDistribution and StartDistribution commands implement the Command interface. The concrete commands pass the call to a receiver object. Parameters required to execute a method on the receiving object are declared as fields in the concrete commands. The ConcreteDistributorReceiver class contains some business logic. The commands only handle the details of how a request is passed to the receiver, while the receiver itself does the actual work. In this case the methods StartContract to start a new distribution or TerminateContract to stop distributing a factory’s products.

## Observer pattern

The ConcreteDistributorReceiver is also the concrete subject class of the application. It issues events of interest to other objects. These events occur when the distributor changes its state, in the case of this application if there is a new publisher to distribute or a publisher stops distributing. The ConcreteDistributorReceiver contains a subscription infrastructure that lets new subscribers to that distributor to join and current subscribers to unsubscribe. The Distributor class is a blueprint for the ConcreteDistributorReceiver. When a new event happens, the distributor goes over the subscription list and calls the notification method declared in the IObsever interface on each observer object. The IObsever interface declares the notification interface. It consists of a single update method. The method has no parameters since the pull technique is being used and there is no need for the distributor to pass event details along with the update. The concrete observers, in this case individual and organization, perform some actions in response to notifications issued by the publisher. Both of these classes implement the same interface so the publisher isn’t coupled to concrete classes.

# Discuss the consequences of your solution (+) and (-) for each of the problem areas: Reusability, Extensibility and Maintainability

# Reusability

## Pros

The way the Observer and Command patterns are implemented in this application provide for the software to be adaptable to different distribution scenarios with different events and additional or less commands. The Abstract factory pattern however has adaptability only for the type of factories that can be created.

## Cons

In order to reuse the code from existing items the additions that are made have to comply with the structure of the interface already created. Due to this the abstract factory pattern can only have sport and fashion products.

# Extensibility

## Pros

The application can easily be extended due to the low number of core classes and interfaces. New functionality can easily be added and the modification of existing functionality can easily be made.

## Cons

Due to the fact that the concrete factory classes also play the role of invokers for the command pattern, and the concrete subject classes play the role of a receiver. The addition of more commands and factories will require more time and effort. Also if other separate from the observer of factory patterns are added that use the command pattern the structure of the command pattern will need to be changed as well as the functionality between patterns.

# Maintainability

## Pros

Due to the use of inheritance the core of the application is made easily maintainable. Problems can be discovered and dealt with fast and easy, the software can easily be kept usable if the environment changes.

## Cons

Due to the core of the application maintainability and performance cannot be improved significantly if at all. Also faults and bugs will not be easily detectable before they happen.