Contents

[Type of Desing Pattern 2](#_Toc144664057)

[CoolShop 2](#_Toc144664058)

[Domain driven development 3](#_Toc144664059)

[Ado Connection Example 3](#_Toc144664060)

[Creational Pattern 3](#_Toc144664061)

[Simple factory 3](#_Toc144664062)

[Factory Method Pattern 3](#_Toc144664063)

[Abstract Factory Pattern 3](#_Toc144664064)

[Builder Pattern 4](#_Toc144664065)

[Prototype 4](#_Toc144664066)

[Singleton Pattern 4](#_Toc144664067)

[Structural Pattern 4](#_Toc144664068)

[Adapter Pattern 4](#_Toc144664069)

[Bridge Pattern 4](#_Toc144664070)

[Composite Pattern 4](#_Toc144664071)

[Decorator Pattern 4](#_Toc144664072)

[Facade Pattern 4](#_Toc144664073)

[FlyWeight Pattern 5](#_Toc144664074)

[Proxy Pattern 5](#_Toc144664075)

[Chain of Responsibility Pattern 5](#_Toc144664076)

[Template Pattern 5](#_Toc144664077)

[Behavioral Patterns 5](#_Toc144664078)

[Mediator Pattern 5](#_Toc144664079)

[Memento Pattern 5](#_Toc144664080)

[Interpreter Pattern 5](#_Toc144664081)

[Iterator Pattern 5](#_Toc144664082)

[Command Pattern 5](#_Toc144664083)

[State Pattern 5](#_Toc144664084)

[Strategy Pattern 5](#_Toc144664085)

[Observer Pattern 6](#_Toc144664086)

[Repository 6](#_Toc144664087)

[Visitor Pattern 6](#_Toc144664088)

[Lazy Loading 6](#_Toc144664089)

[Rip 6](#_Toc144664090)

[IOC, Inversion of Control 6](#_Toc144664091)

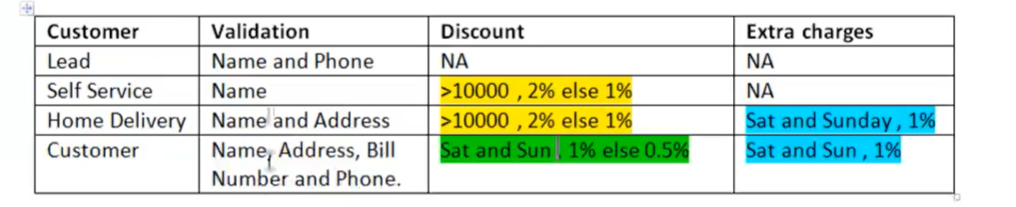
[Unit of work 6](#_Toc144664092)

# Type of Desing Pattern

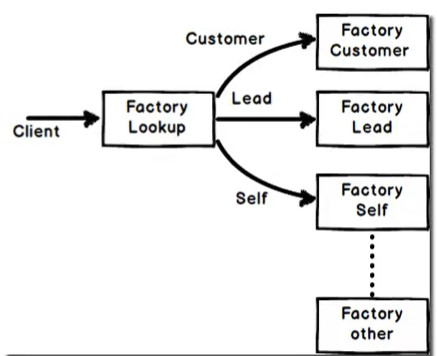
Creational, Structural and Behavioral Patterns

# CoolShop

Customer login



Object creation



# Domain driven development

Follow the vocabulary of you end user (ubiquitous language)

# Ado Connection Example

A computer screen shot of text

Description automatically generated

# Creational Pattern

## Simple factory

Create a new object in a different class.

## Factory Method Pattern

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. The Factory method lets a class defer instantiation to subclass.

## Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete class.

## Builder Pattern

Help us to separate the construction of a complex object from its representation. The same construction process can create difference representations. Common steps for creating differences objects.

**Builder** = Is responsible for defining the construction process for individual parts

**Director** = take those individual processes from the builder and defines the sequence to build the product.

**Product =** Is the final object which is produced from the builder and direction coordination.

## Prototype

Create a new copy of the object (Clone an object). By cloning, any changes to the cloned object does not affect the original object.

**Shallow Cloning**

Only the parent is cloning. By using MemberwiseClone

**Deep Cloning**

The parent and child object are cloning.

## Singleton Pattern

Only one instance of the object share for the hole application. Global Data.

No client can create an instance of this object from outside.

Static keywork create only one instance of one object.

# Structural Pattern

## Adapter Pattern

Helps to work in a unified way when we have incompatible interfaces from tree party. For example, we retrieved data in Json and xml format but we mapped this both types into a common object.

## Bridge Pattern

Helps us to decouple the abstraction from implementation.

## Composite Pattern

Treating difference objects in a similar fashion. Is just an interface implementation. Compose objects

## Decorator Pattern

Adding behavior by sum separate class.

## Facade Pattern

Allow to communicate with subsystems in a unified manner.

## FlyWeight Pattern

Is useful where we need to create many objects and all these objects share some of common data.

## Proxy Pattern

It is like a pointer to some outside data.

## Chain of Responsibility Pattern

Is used when we have series of processing which will be handle by a series of handler logic. Pasing the responsibility to the next class. **is like I cannot handle this, please move ahead and so on until someone can handler.**

## Template Pattern

We have an abstract class acts as a skeleton for its inherited classes calling sequence of methods.

# Behavioral Patterns

## Mediator Pattern

Handle all the communication between difference components. Hidden the communication complex.

## Memento Pattern

To reverse back to any old estate of one object

## Interpreter Pattern

Allow to interpret grammar into code solutions (xml, json… parse)

## Iterator Pattern

When just need to iterate over a list, do not allow add, remove end other method, just iterate. **Enumerable or Enumerator** encapsulate ADD, REMOVE methods.

## Command Pattern

Allows a request to exist as an object. Is like ctr+c or ctr+v created in a class.

## State Pattern

Allow an object to change its behavior depending on the current value of the object.

## Strategy Pattern

This is a behavioral pattern witch help to select algorithms on runtime. It chooses algorithms dynamically.

## Observer Pattern

Helps us to communicate between parent class and its associated or dependent class. By a condition sending (Publisher) notifications to the observers (Subscribers). Broadcasting

# Repository

It creates a commons interface, a common base point for creating a differences type of concrete data access methods.

## Visitor Pattern

Its way of separating the logic and algorithm from the current data structure. You can add new logic to the current data structure without altering the structure.

# Lazy Loading

Load static information just ones.

# Rip

Replacing If condition with polymorphism

# IOC, Inversion of Control

Solid Principle, take this unnecessary logic for this class and put it in another place.

# Unit of work

Solve the problems of repository pattern, create a single transaction for difference repositories.

# Inversion of control

Is passing an interface by constructor letting the parent class decide which concrete class inject.

# Fluent Interfaces

It simplifies your object consumption code by making your code more simple, readable and discoverable.

# Method chaining

Is a common technique where each method returns an object and all these methods can be chained together to form a single statement.

# Immutable Objects

Ones its are loaded cannot be changed internally or externally. (setting variable, singleton).

ReadOnly allows to set the variable just ones time in the constructor.

# Desing Pattern vs Arquitectural Pattern vs Arquitectural Style

A diagram of a diagram

Description automatically generated