

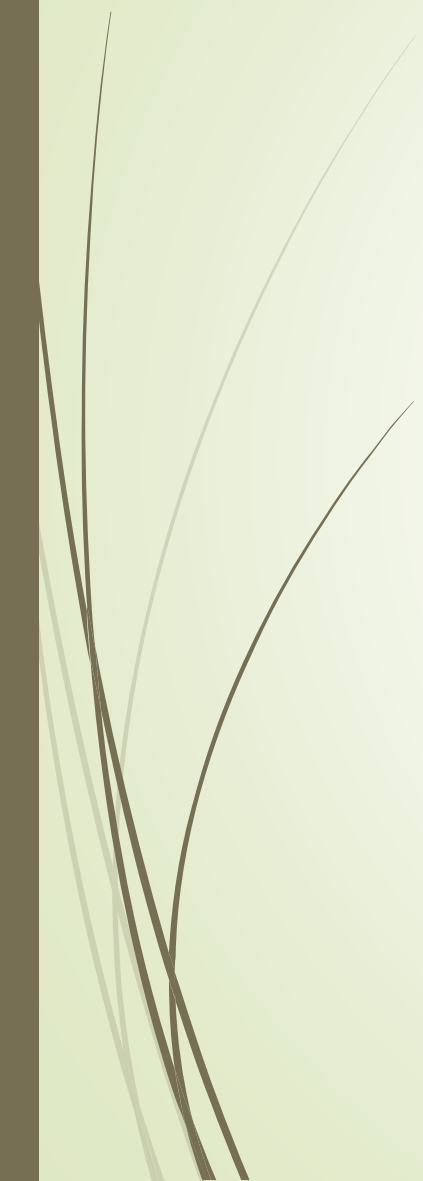


Технологія створення програмних продуктів



Лекція 7. Патерни проектування.

- 
- *Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес* Приемы объектно-ориентированного проектирования. Паттерны проектирования /Design Patterns: Elements of Reusable Object-Oriented Software. — СПб: «Питер», 2007. — С. 366. — ISBN 978-5-469-01136-1 (також ISBN 5-272-00355-1) - 23 шаблони проектування GoF
 - *Крэг Ларман* Применение UML 2.0 и шаблонов проектирования /Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development. — М.: «Вильямс», 2006. — С. 736. — ISBN 0-13-148906-2 - 9 шаблонів GRASP (General Responsibility Assignment Software Patterns, загальні зразки розподілу обов'язків)

- 
- 
- **Шаблони проектування** (*pattern, design pattern*) - це багаторазова архітектурна конструкція, що надає рішення для загальної проблеми проектування в рамках конкретного контексту й описує значимість цього рішення.
 - **Патерн** не є закінченим зразком проекту, який може бути прямо перетворений в код, скоріше це опис або *зразок* для того, як вирішити завдання, таким чином, щоб це можна було використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони часто показують відносини і взаємодії між класами або об'єктами, без визначення того, які кінцеві класи чи об'єкти додатки будуть використовуватися. Алгоритми не розглядаються як шаблони, так як вони вирішують завдання

Класифікація патернів

■ Твірні патерни

- Абстрактна фабрика (Abstract Factory)
- Одинак (Singleton)
- Прототип (Prototype)
- Фабричний метод (Factory Method)
- Будівельник (Builder)
- Віртуальний конструктор (Virtual Constructor)
- Творець примірників класу (Creator)

Э. Гамма,
Р. Хелм,
Р. Джонсон,
Дж. Влиссидес

Крэг Ларман

Класифікація патернів

■ Структурні патерни

- Адаптер (Adapter)
- Декоратор (Decorator)
- Сурогат (Surrogate)
- Компонувальник (Composite)
- Міст (Bridge)
- Пристосуванець (Flyweight)
- Фасад (Facade)
- Інформаційний експерт (Information Expert)
- Низька зв'язаність (Low Coupling)
- Стійкий до змін (Protected Variations)



Класифікація патернів

■ Патерни проектування поведінки класів

- Інтерпретатор (Interpreter)
- Ітератор (Iterator)
- Команда (Command)
- Спостерігач (Observer)
- Відвідувач (Visitor)
- Посередник (Mediator)
- Стан (State)
- Стратегія (Strategy)
- Зберігач (Memento)
- Ланцюжок обов'язків (Chain of Responsibility)
- Шаблонний метод (Template Method)
- Високе зачеплення (High Cohesion)
- Контролер (Controller)
- Поліморфізм (Polymorphism)
- Чиста видумка (Pure Fabrication)
- Перенаправлення (Indirection)



Абстрактна фабрика

Проблема	Створити сімейство взаємопов'язаних або взаємозалежних об'єктів (без специфікації конкретних класів).
Рішення	Створити абстрактний клас, в якому оголошений інтерфейс для створення конкретних класів.
Переваги	<ul style="list-style-type: none">○ Ізолює конкретні класи.○ Оскільки "Абстрактна фабрика" інкапсулює відповідальність за створення класів і сам процес їхнього створення, то вона ізолює клієнта від деталей реалізації класів.○ Спрощена заміна "Абстрактної фабрики" тому, що вона використовується у додатку тільки один раз при інстанціюванні.
Недоліки	<ul style="list-style-type: none">○ Інтерфейс "Абстрактної фабрики" фіксує набір об'єктів, які можна створити.○ Розширення "Абстрактної фабрики" для виготовлення нових об'єктів часто є важким.

Абстрактна фабрика

```
// абстрактна фабрика (abstract factory)
public interface IToyFactory
{
    Bear GetBear();
    Cat GetCat();
}

// конкретна фабрика (concrete factory)
public class TeddyToysFactory : IToyFactory
{
    public Bear GetBear()
    {
        return new TeddyBear();
    }
    public Cat GetCat()
    {
        return new TeddyCat();
    }
}

// і ще одна конкретна фабрика
public class WoodenToysFactory : IToyFactory
{
    public Bear GetBear()
    {
        return new WoodenBear();
    }
    public Cat GetCat()
    {
        return new WoodenCat();
    }
}
```


Абстрактна фабрика

// Базовий клас для усіх котиків, базовий клас AnimalToy містить
Name

```
public abstract class Cat : AnimalToy
{
    protected Cat(string name) : base(name) { }
}
```

// Базовий клас для усіх ведмедиків

```
public abstract class Bear : AnimalToy
{
    protected Bear(string name) : base(name) { }
}
```

// Конкретні реалізації

```
class WoodenCat : Cat
{
    public WoodenCat() : base("Wooden Cat") { }
}
```

```
class TeddyCat : Cat
{
    public TeddyCat() : base("Teddy Cat") { }
}
```

```
class WoodenBear : Bear
{
    public WoodenBear() : base("Wooden Bear") { }
}
```

```
class TeddyBear : Bear
{
    public TeddyBear() : base("Teddy Bear") { }
}
```

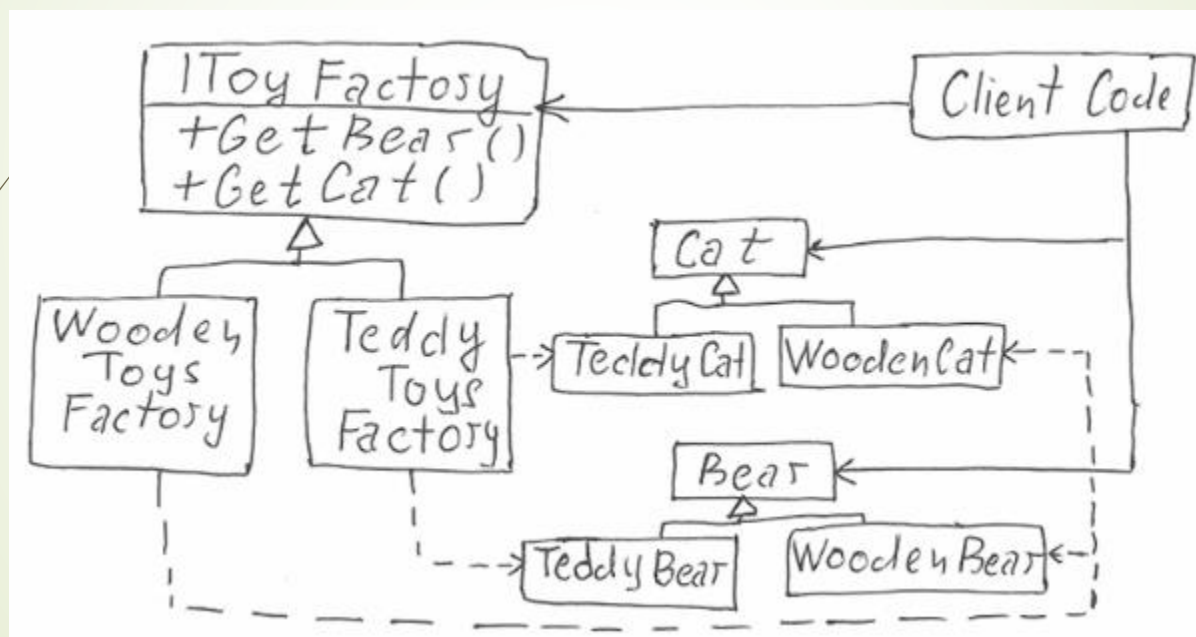
Абстрактна фабрика

```
// Спочатку створимо «дерев'яну» фабрику
IToyFactory toyFactory = new WoodenToysFactory();
Bear bear = toyFactory.GetBear();
Cat cat = toyFactory.GetCat();
Console.WriteLine("I've got {0} and {1}", bear.Name, cat.Name);
// Вивід на консоль буде: [I've got Wooden Bear and Wooden Cat]

// А тепер створимо «плюшеву» фабрику, наступна лінійка є єдиною
різницею в коді
IToyFactory toyFactory = new TeddyToysFactory();
// Як бачимо код нижче не відрізняється від наведеного вище
Bear bear = toyFactory.GetBear();
Cat cat = toyFactory.GetCat();
Console.WriteLine("I've got {0} and {1}", bear.Name, cat.Name);
// А вивід на консоль буде інший: [I've got Teddy Bear and Teddy
Cat]
```

Абстрактна фабрика

Структура в UML



Будівельник

Проблема

Відділити конструювання складного об'єкту від його представлення, так щоб в результаті одного і того ж конструювання могли утворюватися різні представлення. Алгоритм створення складного об'єкта не повинен залежати від того, з яких частин складається об'єкт і як вони стикуються між собою.

Рішення

"Клієнт" створює об'єкт – розпорядник "Директор" і конфігурує його об'єктом – "Будівельником". "Директор" повідомляє "Будівельника" про те, що слід побудувати чергову частину "Продукту". "Будівельник" обробляє запити "Директора" і додає нові частини до "Продукту", потім "Клієнт" забирає "Продукт" у "Будівельника"

Переваги

- Об'єкт "Будівельник" надає об'єкту "Директор" абстрактний інтерфейс для конструювання "Продукту", за яким може приховати представлення і внутрішню структуру продукту, і, крім цього, процес збірки "продукту".
- Для зміни внутрішнього представлення "Продукту" достатньо визначити новий вид "Будівельника".



Будівельник

```
abstract class LaptopBuilder
{
    protected Laptop Laptop { get; private set; }
    public void CreateNewLaptop()
    {
        Laptop = new Laptop();
    }
    // Метод, який повертає готовий ноутбук назовні
    public Laptop GetMyLaptop()
    {
        return Laptop;
    }
    // Кроки, необхідні щоб створити ноутбук
    public abstract void SetMonitorResolution();
    public abstract void SetProcessor();
    public abstract void SetMemory();
    public abstract void SetHDD();
    public abstract void SetBattery();
}
```

Будівельник

```
// Таким будівельником може бути працівник, що
// спеціалізується у складанні «геймерських» ноутів
class GamingLaptopBuilder : LaptopBuilder
{
    public override void SetMonitorResolution()
    {
        Laptop.MonitorResolution = "1900X1200";
    }
    public override void SetProcessor()
    {
        Laptop.Processor = "Core 2 Duo, 3.2 GHz";
    }
    public override void SetMemory()
    {
        Laptop.Memory = "6144 Mb";
    }
    public override void SetHDD()
    {
        Laptop.HDD = "500 Gb";
    }
    public override void SetBattery()
    {
        Laptop.Battery = "6 lbs";
    }
}
```

Будівельник

```
class BuyLaptop //Директор
{
private LaptopBuilder _laptopBuilder;
public void SetLaptopBuilder(LaptopBuilder lBuilder)
{
_laptopBuilder = lBuilder;
}
// Змушує будівельника повернути цілий ноутбук
public Laptop GetLaptop()
{
return _laptopBuilder.GetMyLaptop();
}
// Змушує будівельника додавати деталі
public void ConstructLaptop()
{
_laptopBuilder.CreateNewLaptop();
_laptopBuilder.SetMonitorResolution();
_laptopBuilder.SetProcessor();
_laptopBuilder.SetMemory();
_laptopBuilder.SetHDD();
_laptopBuilder.SetBattery();
}
}
```

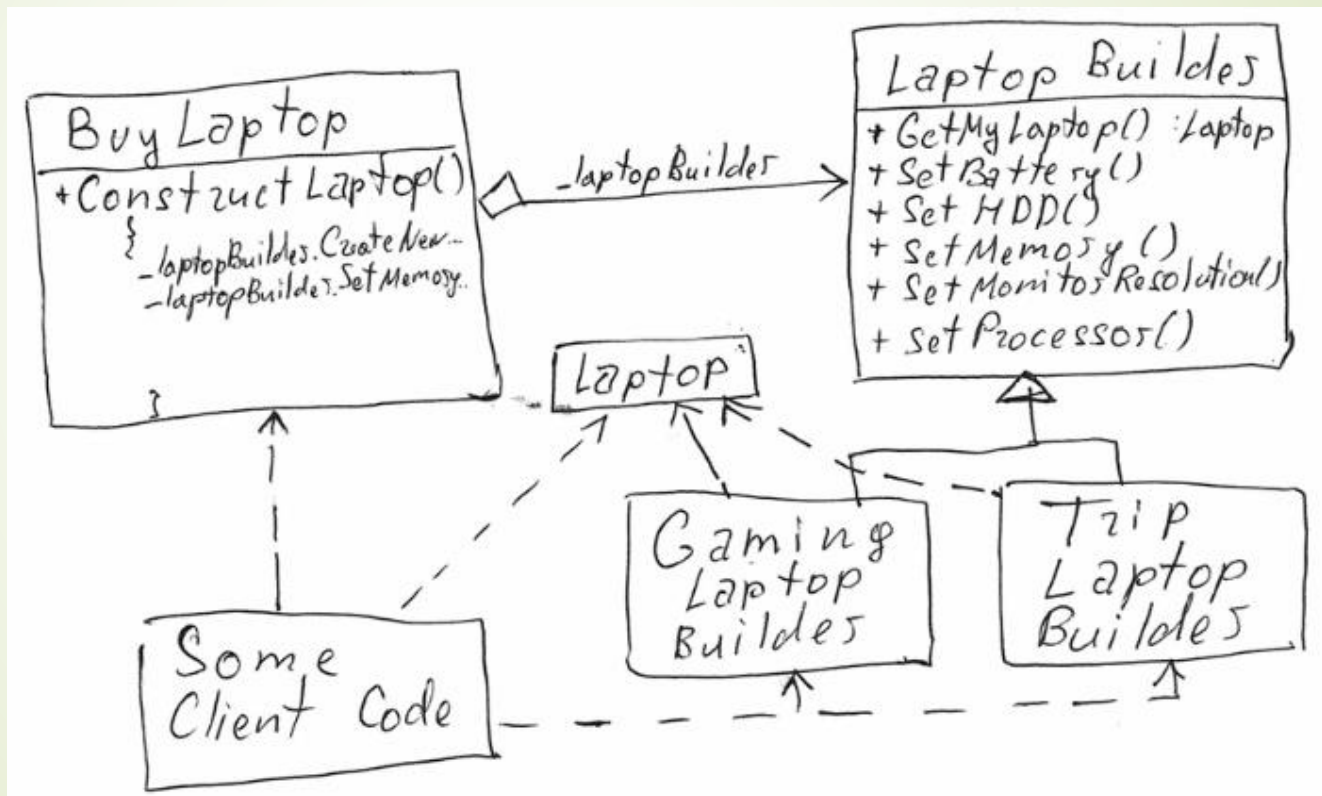

Будівельник

```
// А ось інший «збирач» ноутів
class TripLaptopBuilder : LaptopBuilder
{
    public override void SetMonitorResolution()
    {
        Laptop.MonitorResolution = "1200X800";
    }
    public override void SetProcessor()
    {
        //.. і так далі...
    }
}

// Ваша система може мати багато конкретних будівельників
var tripBuilder = new TripLaptopBuilder();
var gamingBuilder = new GamingLaptopBuilder();
var shopForYou = new BuyLaptop();//Директор
// Покупець каже, що хоче грати ігри
shopForYou.SetLaptopBuilder(gamingBuilder);
shopForYou.ConstructLaptop();
// Ну то нехай бере що хоче!
Laptop laptop = shopForYou.GetLaptop();
Console.WriteLine(laptop.ToString());
// Вивід: [Laptop: 1900X1200, Core 2 Duo, 3.2 GHz, 6144 Mb, 500 Gb, 6 lbs]
```

Будівельник

Структура в UML



Фабричний Метод

Проблема	Визначити інтерфейс для створення об'єкта, але залишити підкласам рішення про те, який клас інстанціювати, тобто, делегувати інстанціювання підкласам.
Рішення	Абстрактний клас "Творець" оголошує ФабричнийМетод, який повертає об'єкт типу "Продукт" (абстрактний клас, що визначає інтерфейс об'єктів, які створюються фабричним методом). "Творець" також може визначити реалізацію за замовчуванням ФабричногоМетоду, який повертає "КонкретнийПродукт". "КонкретнийТворець" заміщує ФабричнийМетод, що повертає об'єкт "КонкретнийПродукт". "Творець" "покладається" на свої підкласи у визначенні ФабричногоМетоду, що повертає об'єкт "КонкретнийПродукт".
Переваги	<ul style="list-style-type: none">○ Позбавляє проектувальника від необхідності вбудовувати в код залежні від додатку класи.
Недоліки	<ul style="list-style-type: none">○ Виникає додатковий рівень підкласів.



Фабричный метод

```
public enum LoggingProviders
{
    Enterprise,
    Log4Net
}

public interface ILogger
{
    void LogMessage(string message);
    void LogError(string message);
    void LogVerboseInformation(string message);
}

class EnterpriseLogger : ILogger
{
    public void LogMessage(string message)
    {
        Console.WriteLine(string.Format("{0}: {1}",
"Enterprise", message));
    }
    public void LogError(string message)
    {
        throw new NotImplementedException();
    }
    public void LogVerboseInformation(string message)
    {
        throw new NotImplementedException();
    }
}
```

Фабричний метод

Якщо треба додати провайдера
(реалізація фабричного методу)

```
class LoggerProviderFactory
{
    public static ILogger GetLoggingProvider(LoggingProviders
logProviders)
    {
        switch (logProviders)
        {
            case LoggingProviders.Enterprise:
                return new EnterpriseLogger();
            case LoggingProviders.Log4Net:
                return new Log4NetLogger();
            default:
                return new EnterpriseLogger();
        }
    }
}
```

Фабричний метод

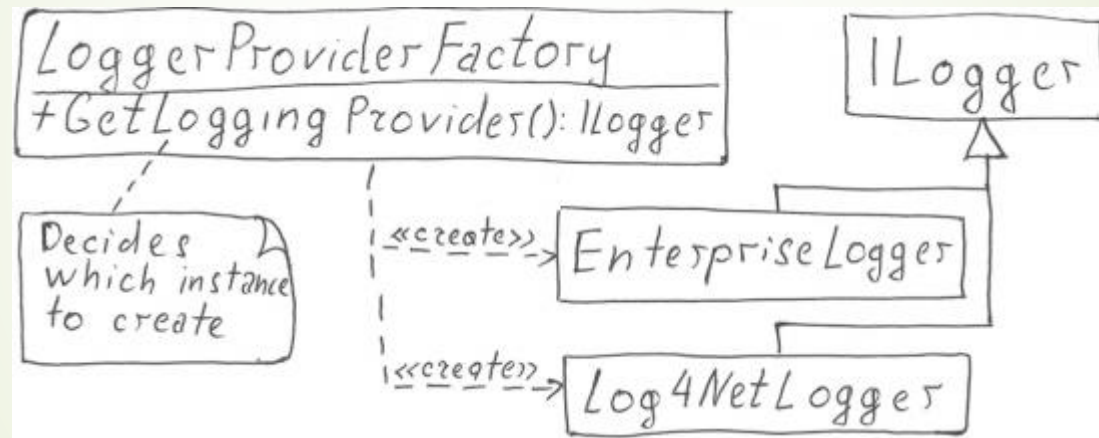
Використання фабричного методу

```
public static void Run()
{
    var providerType = GetTypeOfLoggingProviderFromConfigFile();
    ILogger logger =
        LoggerProviderFactory.GetLoggingProvider(providerType);
    logger.LogMessage("Hello Factory Method Design Pattern.");
    // Вивід: [Log4Net: Hello Factory Method Design Pattern]
}

private static LoggingProviders
GetTypeOfLoggingProviderFromConfigFile()
{
    // Це такий собі хакркод, щоб не ускладнювати прикладу
    return LoggingProviders.Log4Net;
}
```

Фабричный Метод

Структура в UML





Прототип

Проблема	Система не повинна залежати від того, як у ній створюються, компонуються і представляються об'єкти.
Рішення	Створювати нові об'єкти з допомогою патерна - прототипу. "Прототип" оголошує інтерфейс для клонування самого себе. "Клієнт" створює новий об'єкт, звертаючись до "Прототипу" з запитом клонувати "Прототип".
Переваги	
Недоліки	



Прототип

```
class CalendarPrototype
{
    public virtual CalendarPrototype Clone()
    {
        var copyOfPrototype =
            (CalendarPrototype)this.MemberwiseClone();
        return copyOfPrototype;
    }
}

class CalendarEvent : CalendarPrototype
{
    public Attendee[] Attendees { get; set; }
    public Priority Priority { get; set; }
    public DateTime StartDateAndTime { get; set; }

    // Зауважимо, що метод Clone не перевантажений (покищо)
}
```

Прототип

```
public class PrototypeDemo
{
    public static CalendarEvent GetExistingEvent()
    {
        var beerParty = new CalendarEvent();
        var friends = new Attendee[1];
        var andriy = new Attendee { FirstName = "Jakyjs", LastName = "Chuvak" };
        friends[0] = chuvak;
        beerParty.Attendees = friends;
        beerParty.StartDateAndTime = new DateTime(2018, 4, 24, 18, 0, 0);
        beerParty.Priority = Priority.High();

        return beerParty;
    }

    public static void Run()
    {
        var beerParty = GetExistingEvent();
        var nextFridayEvent = (CalendarEvent)beerParty.Clone();
        nextFridayEvent.StartDateAndTime = new DateTime(2018, 7, 21, 18, 0, 0);

        // про цей код побалакаємо трошки нижче
        nextFridayEvent.Attendees[0].EmailAddress = "MyEMail@liamg.com";
        nextFridayEvent.Priority.SetPriorityValue(0);

        if (beerParty.Attendees != nextFridayEvent.Attendees)
        {
            Console.WriteLine("GOOD: Each event has own list of attendees.");
        }
        if (beerParty.Attendees[0].EmailAddress == nextFridayEvent.Attendees[0].EmailAddress)
        {
            //В цьому випадку добре мати поверхневу копію кожного із учасників
            //таким чином моя адреса, ім'я і персональні дані залишаються тими ж
            Console.WriteLine("GOOD: If I updated my e-mail address it will be updated in all events.");
        }
        if (beerParty.Priority.IsHigh() != nextFridayEvent.Priority.IsHigh())
        {
            Console.WriteLine("GOOD: Each event should have own priority object, fully-copied.");
        }
    }
}
```



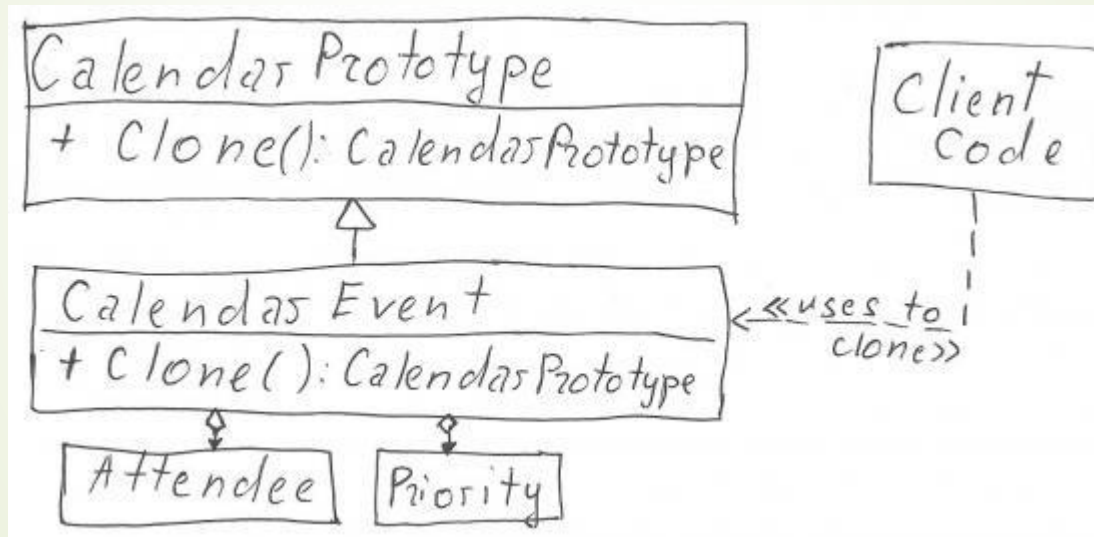
Прототип

```
public override CalendarPrototype Clone()
{
    var copy = (CalendarEvent)base.Clone();

    // Це дозволить нам мати інший список із посиланнями на тих же
    // відвідувачів
    var copiedAttendees = (Attendee[])Attendees.Clone();
    copy.Attendees = copiedAttendees;
    // Також скопіюємо пріоритет
    copy.Priority = (Priority)Priority.Clone();
    // День і час не варто копіювати – їх заповнять
    // Повертаємо копію події
    return copy;
}
```

Прототип

Структура в UML



ОДИНАК

Проблема Необхідний лиш один екземпляр спеціального класу, різні об'єкти повинні звертатися до цього екземпляру через єдину точку доступу.

Рішення Створити клас і визначити статичний метод класу, який повертає цей єдиний об'єкт.

Переваги

- Слід створювати саме статичний екземпляр спеціального класу, а не оголошувати потрібні методи статичними, оскільки при використанні методів екземпляра можна застосувати механізм наслідування і створювати підкласи. Статичні методи в мовах програмування не поліморфні і не допускають перекриття у похідних класах.
- Рішення на основі створення екземпляра є більш гнучким, оскільки в подальшому може знадобитись

ОДИНАК

Реалізація «Одинака»

```
public class LoggerSingleton
{
    private LoggerSingleton() { }
    private int _logCount = 0;
    private static LoggerSingleton _loggerSingletonInstance
= null;
    public static LoggerSingleton GetInstance()
    {
        if (_loggerSingletonInstance == null)
        {
            _loggerSingletonInstance = new
LoggerSingleton();
        }
        return _loggerSingletonInstance;
    }

    public void Log(String message)
    {
        Console.WriteLine(_logCount + ": " + message);
        _logCount++;
    }
}
```


ОДИНАК

Потокобезпечна реалізація «Одинака»
Необхідно, щоб не створювати велику
кількість екземплярів того ж класу

```
public class ThreadSafeLoggerSingleton
{
    private ThreadSafeLoggerSingleton()
    {
        // Читаємо дані із якогось файлу і дістаємо номер останнього запису
        // _logCount = вичитане значення
    }
    private int _logCount = 0;
    private static ThreadSafeLoggerSingleton _loggerInstance;
    private static readonly object locker = new object();

    public static ThreadSafeLoggerSingleton GetInstance()
    {
        lock (locker)
        {
            if (_loggerInstance == null)
            {
                _loggerInstance = new ThreadSafeLoggerSingleton();
            }
        }
        return _loggerInstance;
    }

    public void Log(String message)
    {
        Console.WriteLine(_logCount + ": " + message);
        _logCount++;
    }
}
```

ОДИНАК

Використання «Одинака»

```
public class SingletonDemo
{
    public static void Run()
    {
        DoHardWork();
    }

    public static void DoHardWork()
    {
        LoggerSingleton logger = LoggerSingleton.GetInstance();
        HardProcessor processor = new HardProcessor(1);
        logger.Log("Hard work started...");
        processor.ProcessTo(5);
        logger.Log("Hard work finished...");
    }
}

public class HardProcessor
{
    private int _start;
    public HardProcessor(int start)
    {
        _start = start;
        LoggerSingleton.GetInstance().Log("Processor just created.");
    }
    public int ProcessTo(int end)
    {
        int sum = 0;
        for (int i = _start; i <= end; ++i)
        {
            sum += i;
        }
        LoggerSingleton.GetInstance().Log("Processor just calculated some value: " + sum);
        return sum;
    }
}
```

Не цікавить, що він робить

Адаптер

Проблема Необхідно забезпечити взаємодію несумісних інтерфейсів або створити єдиний стійкий інтерфейс для декількох компонентів з різними інтерфейсами.

Рішення Конвертувати вихідний інтерфейс компонента до іншого виду за допомогою проміжного об'єкта – адаптера, тобто, додати спеціальний об'єкт з спільним інтерфейсом у рамках даного додатку і перенаправити зв'язки від зовнішніх об'єктів до цього об'єкту – адаптеру.

Адаптер

```
// Клас який буде адаптовуватися
class OldElectricitySystem
{
    public string MatchThinSocket()
    {
        return "220V";
    }
}
// широковикористовуваний інтерфейс
interface INewElectricitySystem
{
    string MatchWideSocket();
}
class NewElectricitySystem : INewElectricitySystem
{
    public string MatchWideSocket()
    {
        return "220V";
    }
}
// Адаптер
class Adapter : INewElectricitySystem
{
    private readonly OldElectricitySystem _adaptee;
    public Adapter(OldElectricitySystem adaptee)
    {
        _adaptee = adaptee;
    }
    // наш адаптер перекладає із того, що ми (код) не можемо використати наразі у те що ми можемо
    public string MatchWideSocket()
    {
        return _adaptee.MatchThinSocket();
    }
}
```

Адаптер

```
class ElectricityConsumer
{
    // Зарядний пристрій розуміє тільки нову систему
    public static void ChargeNotebook(INewElectricitySystem electricitySystem)
    {
        Console.WriteLine(electricitySystem.MatchWideSocket());
    }
}

public class AdapterDemo
{
    public static void Run()
    {
        // 1)
        // Ми можемо і надалі користувати нашою новою системою
        var newElectricitySystem = new NewElectricitySystem();
        ElectricityConsumer.ChargeNotebook(newElectricitySystem);

        // 2)
        // Ми повинні адаптуватися до старої системи, використовуючи адаптер
        var oldElectricitySystem = new OldElectricitySystem();
        var adapter = new Adapter(oldElectricitySystem);
        ElectricityConsumer.ChargeNotebook(adapter);
    }
}
```

Міст

Проблема Потрібно відділити абстракцію від реалізації так, щоб і те і інше можна було змінювати незалежно. При використанні наслідування реалізація жорстко прив'язується до абстракції, що утруднює незалежну модифікацію.

Рішення Помістити абстракцію і реалізацію в окремі ієрархії класів.

Переваги

- Відмежовування реалізації від інтерфейсу, тобто, "Реалізацію" "Абстракції" можна конфігурувати під час виконання. Крім того, слід згадати, що розділення класів "Абстракція" і "Реалізація" знімає залежності від реалізації, що встановлюються на етапі компіляції: щоб змінити клас "Реалізація" зовсім не обов'язково перекомпільовувати клас "Абстракція".

MiCT

```
internal interface IBuldingCompany
{
    void BuildFoundation();
    void BuildRoom();
    void BuildRoof();
    IWallCreator WallCreator { get; set; }
}

class BuldingCompany : IBuldingCompany
{
    public void BuildFoundation()
    {
        Console.WriteLine("Foundation is built.{0}", Environment.NewLine);
    }
    public void BuildRoom()
    {
        WallCreator.BuildWallWithDoor();
        WallCreator.BuildWall();
        WallCreator.BuildWallWithWindow();
        WallCreator.BuildWall();
        Console.WriteLine("Room finished.{0}", Environment.NewLine);
    }
    public void BuildRoof()
    {
        Console.WriteLine("Roof is done.{0}", Environment.NewLine);
    }
    public IWallCreator WallCreator { get; set; }
}
```



```
internal interface IWallCreator
{
    void BuildWallWithDoor();
    void BuildWallWithWindow();
    void BuildWall();
}

class ConcreteSlabWallCreator : IWallCreator
{
    public void BuildWallWithDoor()
    { Console.WriteLine("Concrete slab wall with door.");
    }
    public void BuildWallWithWindow()
    { Console.WriteLine("Concrete slab wall with window.");
    }
    public void BuildWall()
    { Console.WriteLine("Concrete slab wall.");
    }
}

class BrickWallCreator : IWallCreator
{
    public void BuildWallWithDoor()
    { Console.WriteLine("Brick wall with door.");
    }
    public void BuildWallWithWindow()
    { Console.WriteLine("Brick wall with window.");
    }
    public void BuildWall()
    { Console.WriteLine("Brick wall.");
    }
}
```

MiCT

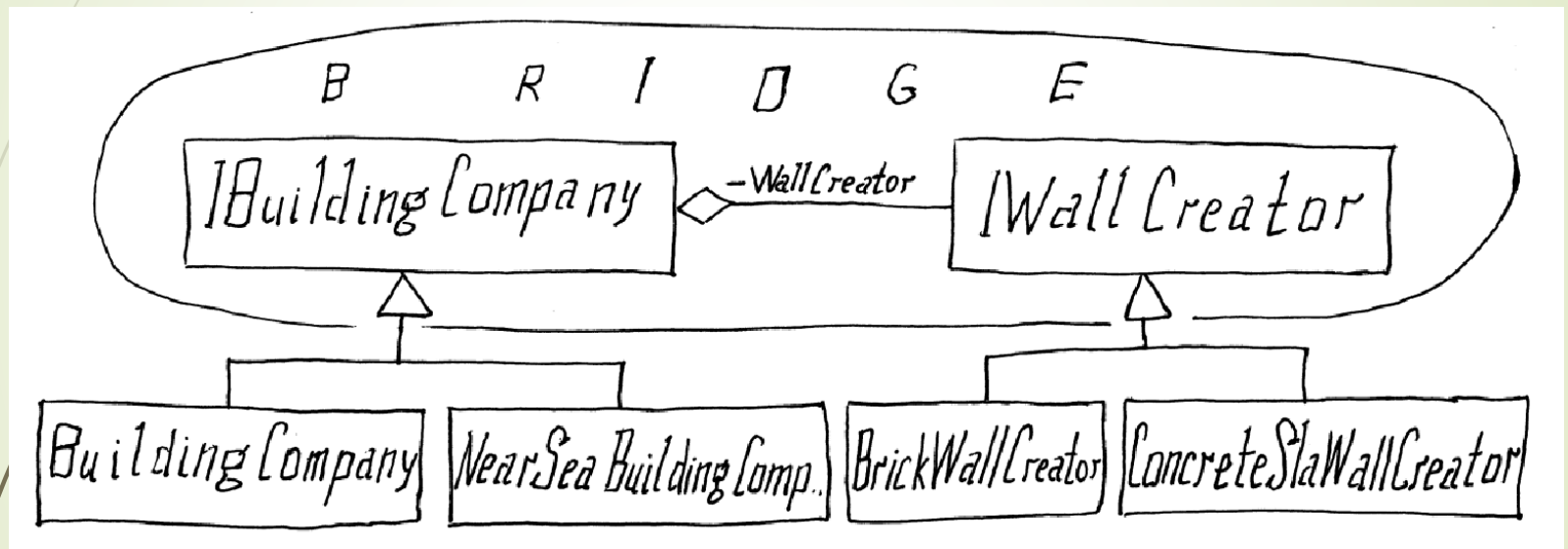
```
public class BridgeDemo
{
    public static void Run()
    {
        // We have two wall creating crews - concrete blocks one and bricks one
        var brickWallCreator = new BrickWallCreator();
        var concreteSlabWallCreator = new ConcreteSlabWallCreator();

        var buildingCompany = new BuildingCompany();
        buildingCompany.BuildFoundation();

        buildingCompany.WallCreator = concreteSlabWallCreator;
        buildingCompany.BuildRoom();

        // Company can easily switch to another wall crew to continue building rooms
        // with another material
        buildingCompany.WallCreator = brickWallCreator;
        buildingCompany.BuildRoom();
        buildingCompany.BuildRoom();

        buildingCompany.BuildRoof();
    }
}
```



Компонувальник

Проблема Як обробляти групу або композицію структур об'єктів одночасно?

Рішення Визначити класи для композитних і атомарних об'єктів таким чином, щоб вони реалізовували один і той же інтерфейс.

КОМПОНУВАЛЬНИК

```
interface IDocumentComponent
{
    string GatherData();
    void AddComponent(IDocumentComponent documentComponent);
}

class DocumentComponent : IDocumentComponent
{
    public string Name { get; private set; }
    public List<IDocumentComponent> DocumentComponents { get; private set; }
    public DocumentComponent(string name)
    {
        Name = name;
        DocumentComponents = new List<IDocumentComponent>();
    }
    public string GatherData()
    {
        var stringBuilder = new StringBuilder();
        stringBuilder.AppendLine(string.Format("<{0}>", Name));
        foreach (var documentComponent in DocumentComponents)
        {
            documentComponent.GatherData();
            stringBuilder.AppendLine(documentComponent.GatherData());
        }
        stringBuilder.AppendLine(string.Format("</{0}>", Name));
        return stringBuilder.ToString();
    }
    public void AddComponent(IDocumentComponent documentComponent)
    {
        DocumentComponents.Add(documentComponent);
    }
}
```

КОМПОНУВАЛЬНИК

```
class CustomerDocumentComponent : IDocumentComponent
{
    private int CustomerIdToGatherData { get; set; }
    public CustomerDocumentComponent(int customerIdToGatherData)
    {
        CustomerIdToGatherData = customerIdToGatherData;
    }
    public string GatherData()
    {
        string customerData;
        switch (CustomerIdToGatherData)
        {
            case 41:
                customerData = "Andriy Buday";
                break;
            default:
                customerData = "Someone else";
                break;
        }
        return string.Format("<Customer>{0}</Customer>", customerData);
    }
    public void AddComponent(IDocumentComponent documentComponent)
    {
        Console.WriteLine("Cannot add to leaf...");
    }
}
```

КОМПОНУВАЛЬНИК

```
class OrderDocumentComponent : IDocumentComponent
{
    private int OrderIdToGatherData { get; set; }

    public OrderDocumentComponent(int orderIdToGatherData)
    {
        OrderIdToGatherData = orderIdToGatherData;
    }
    public string GatherData()
    {
        string orderData;
        switch (OrderIdToGatherData)
        {
            case 0:
                orderData = "Kindle;Book1;Book2";
                break;
            default:
                orderData = "Phone;Cable;Headset";
                break;
        }
        return string.Format("<Order>{0}</Order>", orderData);
    }

    public void AddComponent(IDocumentComponent documentComponent)
    {
        Console.WriteLine("Cannot add to leaf...");
    }
}

class HeaderDocumentComponent : IDocumentComponent
{
    public string GatherData()
    {
        return string.Format("<Header><MessageTime>{0}</MessageTime></Header>",
            DateTime.Now.ToLongTimeString());
    }
    public void AddComponent(IDocumentComponent documentComponent)
    {
        Console.WriteLine("Cannot add to leaf...");
    }
}
```


КОМПОНУВАЛЬНИК

```
public class CompositeDemo
{
    public static void Run()
    {
        var document = new DocumentComponent("ComposableDocument");
        var headerDocumentSection = new HeaderDocumentComponent();
        var body = new DocumentComponent("Body");
        document.AddComponent(headerDocumentSection);
        document.AddComponent(body);

        var customerDocumentSection = new CustomerDocumentComponent(41);
        var orders = new DocumentComponent("Orders");
        var order0 = new OrderDocumentComponent(0);
        var order1 = new OrderDocumentComponent(1);
        orders.AddComponent(order0);
        orders.AddComponent(order1);

        body.AddComponent(customerDocumentSection);
        body.AddComponent(orders);

        string gatheredData = document.GatherData();

        Console.WriteLine(gatheredData);
    }
}
```

КОМПОНУВАЛЬНИК

```
<ComposableDocument>
  <Header>
    <MessageTime>8:47:23</MessageTime>
  </Header>
  <Body>
    <Customer>Andriy Buday</Customer>
    <Orders>
      <Order>Kindle;Book1;Book2</Order>
      <Order>Phone;Cable;Headset</Order>
    </Orders>
  </Body>
</ComposableDocument>
```

Декоратор

Проблема	Покласти додаткові обов'язки (прозорі для клієнтів) на окремий об'єкт, а не на клас в цілому.
Рішення	Динамічно додати об'єкту нові обов'язки не застосовуючи при цьому породження підкласів (наслідування). "Компонент" визначає інтерфейс для об'єктів, на які можуть бути динамічно покладені додаткові обов'язки, "КонкретнийКомпонент" визначає об'єкт, на який покладаються додаткові обов'язки, "Декоратор" – зберігає посилання на об'єкт "Компонент" і визначає інтерфейс, що відповідає інтерфейсу "Компонент". "КонкретнийДекоратор" покладає додаткові обов'язки на компонент.
Переваги	Відмежовування реалізації від інтерфейсу, Біліша гнучкість, ніж у статичного наслідування: можна додавати і видаляти обов'язки під час виконання програми тоді як при використанні наслідування треба було б створювати новий клас для кожного додаткового обов'язку. Даний патерн дозволяє уникнути переважаних методами класів на верхніх рівнях ієрархії – нові обов'язки можна додавати за необхідності. Тобто, "Реалізацію" "Абстракції" можна конфігурувати під час виконання. Крім того, слід згадати, що розділення класів "Абстракція" і "Реалізація" знімає залежності від реалізації, що встановлюються на етапі компіляції: щоб змінити клас "Реалізація" зовсім не обов'язково перекомпільовувати клас "Абстракція".
Недоліки	"Декоратор" та його "Компонент" не ідентичні, і, крім того, виходить що система складається з великого числа малих об'єктів, які схожі один на одного і різняться тільки способом взаємозв'язку, а не класом і не значеннями своїх внутрішніх змінних – така система складна для вивчення та налагодження.

Декоратор

```
class Car
{
    protected String BrandName { get; set; }
    public virtual void Go()
    {
        Console.WriteLine("I'm " + BrandName + " and I'm on my way...");
    }
}

class DecoratorCar : Car
{
    protected Car DecoratedCar { get; set; }
    public DecoratorCar(Car decoratedCar)
    {
        DecoratedCar = decoratedCar;
    }
    public override void Go()
    {
        DecoratedCar.Go();
    }
}

class AmbulanceCar : DecoratorCar
{
    public AmbulanceCar(Car decoratedCar)
        : base(decoratedCar)
    {
    }
    public override void Go()
    {
        base.Go();
        Console.WriteLine("... beep-beep-beeeeeep ...");
    }
}
```

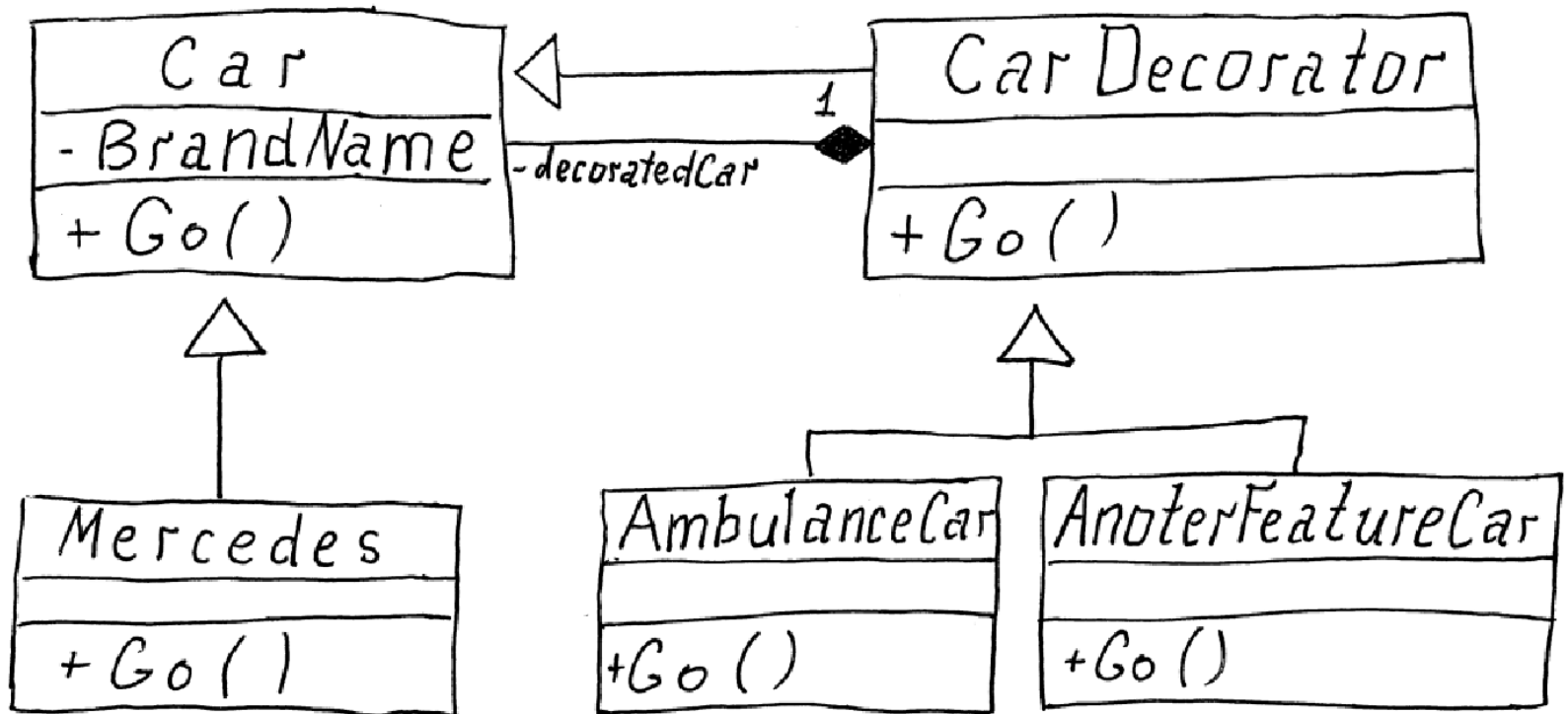
Декоратор

```
//конкретна реалізація класу Car
class Mercedes : Car
{
    public Mercedes()
    {
        BrandName = "Mercedes";
    }
}

public static class DecoratorDemo
{
    public static void Run()
    {
        var doctorsDream = new AmbulanceCar(new Mercedes());
        doctorsDream.Go();
    }
}
```

I'm Mercedes and I'm on my way... ... beep-beep-beeeeeep ...

Декоратор



Фасад

Проблема Як забезпечити уніфікований інтерфейс з набором розрізнених реалізацій чи інтерфейсів, наприклад, з підсистемою, якщо небажаним є високе зв'язування з цією підсистемою або реалізація підсистеми може змінитися?

Рішення Визначити одну точку взаємодії з підсистемою – фасадний об'єкт, що забезпечує загальний інтерфейс з підсистемою, та покласти на нього обов'язки з взаємодії з її компонентами.

Фасад – це зовнішній об'єкт, який забезпечує єдину точку входу для служб підсистеми. Реалізація інших компонентів підсистеми закрита і її “не видно” зовнішнім компонентам.

ΦασαΔ

```
class SkiRent
{
    public int RentBoots(int feetSize, int skierLevel)
    {
        return 20;
    }
    public int RentSki(int weight, int skierLevel)
    {
        return 40;
    }
    public int RentPole(int height)
    {
        return 5;
    }
}

class SkiResortTicketSystem
{
    public int BuyOneDayTicket()
    {
        return 120;
    }
    public int BuyHalfDayTicket()
    {
        return 60;
    }
}

class HotelBookingSystem
{
    public int BookRoom(int roomQuality)
    {
        switch (roomQuality)
        {
            case 3:      return 250;
            case 4:      return 500;
            case 5:      return 900;
            default:     throw new ArgumentException("roomQuality should be in range [3;5]");
        }
    }
}
```

ΦασαΔ

```
class SkiResortFacade
{
    private SkiRent _skiRent = new SkiRent();
    private SkiResortTicketSystem _skiResortTicketSystem = new SkiResortTicketSystem();
    private HotelBookingSystem _hotelBookingSystem = new HotelBookingSystem();

    public int HaveGoodRest(int height, int weight, int feetSize, int skierLevel, int roomQuality)
    {
        int skiPrice = _skiRent.RentSki(weight, skierLevel);
        int skiBootsPrice = _skiRent.RentBoots(feetSize, skierLevel);
        int polePrice = _skiRent.RentPole(height);
        int oneDayTicketPrice = _skiResortTicketSystem.BuyOneDayTicket();
        int hotelPrice = _hotelBookingSystem.BookRoom(roomQuality);

        return skiPrice + skiBootsPrice + polePrice + oneDayTicketPrice + hotelPrice;
    }

    public int HaveRestWithOwnSkis()
    {
        int oneDayTicketPrice = _skiResortTicketSystem.BuyOneDayTicket();
        return oneDayTicketPrice;
    }
}

public class FacadeDemo
{
    public static void Run()
    {
        var skiResortFacade = new SkiResortFacade();
        int weekendRestPrice = skiResortFacade.HaveGoodRest(175, 60, 42, 2, 3);
        Console.WriteLine("Price: {0}", weekendRestPrice);
    }
}
```

Пристосуванець (Легковаговик)

Проблема	Необхідно забезпечити підтримку множини малих об'єктів.
Рекомендації	<p>Пристосуванці моделюють сутності, число яких надто велике для представлення об'єктами. Має зміст використовувати даний патерн якщо одночасно виконуються наступні умови:</p> <ul style="list-style-type: none">•у додатку використовується велике число об'єктів, через це видатки на зберігання високі,•більшу частину стану об'єктів можна винести назовні,•численні групи об'єктів можна замінити відносно невеликою кількістю об'єктів, оскільки стіни об'єктів винесені назовні.
Рішення	<p>Створити розподілений об'єкт, який можна використовувати одночасно у декількох контекстах, причому, в кожному контексті він виглядає як незалежний об'єкт (не відрізняється від екземпляра, що не розподіляється). "Пристосуванець" оголошує інтерфейс, з допомогою якого пристосуванці можуть отримати зовнішній стан або якось подіяти на нього, "КонкретнийПристосуванець" реалізує інтерфейс класу "Пристосуванець" і додає при необхідності внутрішній стан. Внутрішній стан зберігається в об'єкті "КонкретнийПристосуванець", тоді як зовнішній стан зберігається або обчислюється "Клієнтами" ("Клієнт" передає його "Пристосуванцю" при виклику операцій).</p>
Переваги	Внаслідок зменшення загального числа екземплярів і винесення стану економиться пам'ять.

Пристосуванець

```
class Image
{
    private List<string> _imitatesHugeImage = new List<string>();
    public static Image Load(string fileName)
    {
        var img = new Image();
        for (int i = 0; i < 10000; i++)
        {
            img._imitatesHugeImage.Add(string.Format("abcdefgh:{0}", i));
        }
        return img;
    }
}

class Parser
{
    public List<Unit> ParseHTML()
    {
        var units = new List<Unit>();
        for (int i = 0; i < 150; i++)
            units.Add(new Dragon());
        for (int i = 0; i < 500; i++)
            units.Add(new Goblin());
        Console.WriteLine("Dragons and Goblins are parsed from HTML page.");
        return units;
    }
}
```

Пристосування

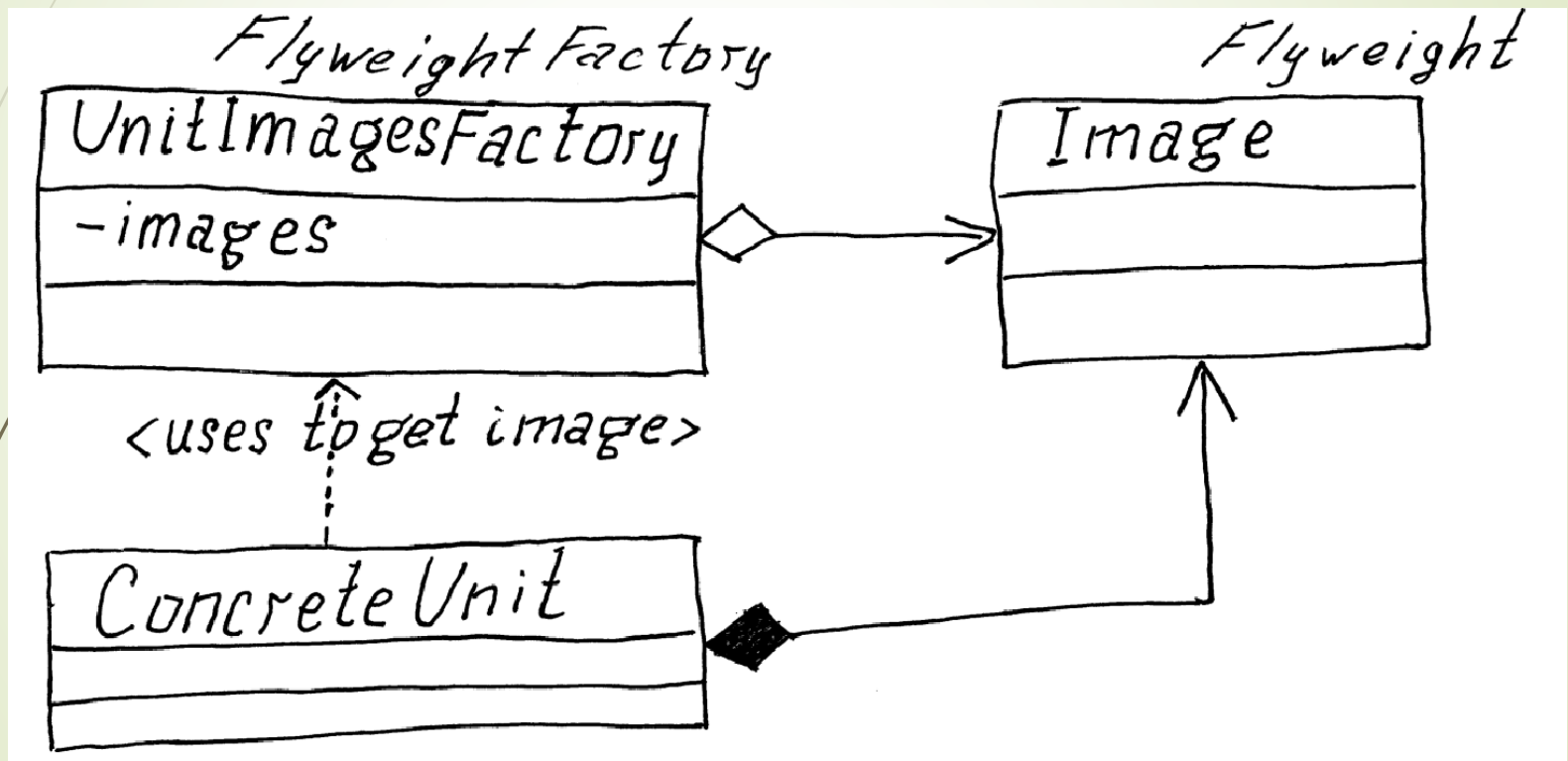
```
abstract class Unit
{
    public string Name { get; protected set; }
    public int Health { get; protected set; }
    public Image Picture { get; protected set; }
}
class Goblin : Unit
{
    public Goblin()
    {
        Name = "Goblin";
        Health = 8;
        //old
        Picture = Image.Load("Goblin.jpg");
        //new
        Picture = UnitImagesFactory.CreateGoblinImage();
    }
}

class Dragon : Unit
{
    public Dragon()
    {
        Name = "Dragon";
        Health = 50;
        //old
        Picture = Image.Load("Dragon.jpg");
        //new
        Picture = UnitImagesFactory.CreateDragonImage();
    }
}
```

Пристосуванець

```
class UnitImagesFactory
{
    public static Dictionary<Type, Image> Images = new Dictionary<Type, Image>();
    public static Image CreateDragonImage()
    {
        if (!Images.ContainsKey(typeof(Dragon)))
        {
            Images.Add(typeof(Dragon), Image.Load("Dragon.jpg"));
        }
        return Images[typeof(Dragon)];
    }
    public static Image CreateGoblinImage()
    {
        if (!Images.ContainsKey(typeof(Goblin)))
        {
            Images.Add(typeof(Goblin), Image.Load("Goblin.jpg"));
        }
        return Images[typeof(Goblin)];
    }
}
```

Пристосування



Заступник (Проксі)

Проблема	Необхідно управляти доступом до об'єкта так, щоб створювати громіздкі об'єкти "на вимогу".
Рішення	<p>Створити сурогат громіздкого об'єкта. "Замісник" зберігає посилання, яке дозволяє заміснику звернутися до реального суб'єкта (об'єкт класу "Замісник" може звертатися до об'єкта класу "Суб'єкт", якщо інтерфейси "РеальногоСуб'єкта" та "Суб'єкта" однакові). Оскільки інтерфейс "РеальногоСуб'єкта" ідентичний інтерфейсу "Суб'єкта", таким чином, що "Замісника" можна підставити замість "РеальногоСуб'єкта", контролює доступ до "РеальногоСуб'єкта", може відповідати за створення чи видалення "РеальногоСуб'єкта". "Суб'єкт" визначає спільний для "РеальногоСуб'єкта" і "Замісника" інтерфейс, таким чином, що "Замісник" може бути використаний всюди, де очікується "РеальнийСуб'єкт". За необхідності запити можуть бути переадресовані "Замісником" "РеальномуСуб'єкту". "Замісник" може мати і інші обов'язки, а саме:</p> <ul style="list-style-type: none">• віддалений "Замісник" може відповідати за кодування запиту і його аргументів та відсилання закодованого запиту реальному "Суб'єкту",• віртуальний "Замісник" може кешувати додаткову інформацію про реальний "Суб'єкт", щоб відкласти його створення,• захисний "Замісник" може перевіряти, чи має об'єкт, що викликає, необхідні для виконання запиту права.

Заступник

```
public class RobotBombDefuser
{
    private Random _random = new Random();
    private int _robotConfiguredWavelength = 41;
    private bool _isConnected = false;

    public void ConnectWireless(int communicationWavelength)
    {
        if (communicationWavelength == _robotConfiguredWavelength)
        {
            _isConnected = IsConnectedImmitatingConnectivitiyIssues();
        }
    }

    public bool IsConnected()
    {
        _isConnected = IsConnectedImmitatingConnectivitiyIssues();
        return _isConnected;
    }

    private bool IsConnectedImmitatingConnectivitiyIssues()
    {
        return _random.Next(0, 10) < 4; // immitates 40% good connection, aka. very bad
    }

    public virtual void WalkStraightForward(int steps)
    {
        Console.WriteLine("Did {0} steps forward...", steps);
    }

    public virtual void TurnRight()
    {
        Console.WriteLine("Turned right...");
    }

    public virtual void TurnLeft()
    {
        Console.WriteLine("Turned left...");
    }

    public virtual void DefuseBomb()
    {
        Console.WriteLine("Cut red or green or blue wire...");
    }
}
```

Заступник

```
public class RobotBombDefuserProxy : RobotBombDefuser
{
    private RobotBombDefuser _robotBombDefuser;
    private int _communicationWaveLength;
    private int _connectionAttempts = 3;

    public RobotBombDefuserProxy(int communicationWaveLength)
    {
        _robotBombDefuser = new RobotBombDefuser();
        _communicationWaveLength = communicationWaveLength;
    }
    public override void WalkStraightForward(int steps)
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.WalkStraightForward(steps);
    }
    public override void TurnRight()
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.TurnRight();
    }
    public override void TurnLeft()
    {
        EnsureConnectedWithRobot();
        _robotBombDefuser.TurnLeft();
    }
}
```

Заступник

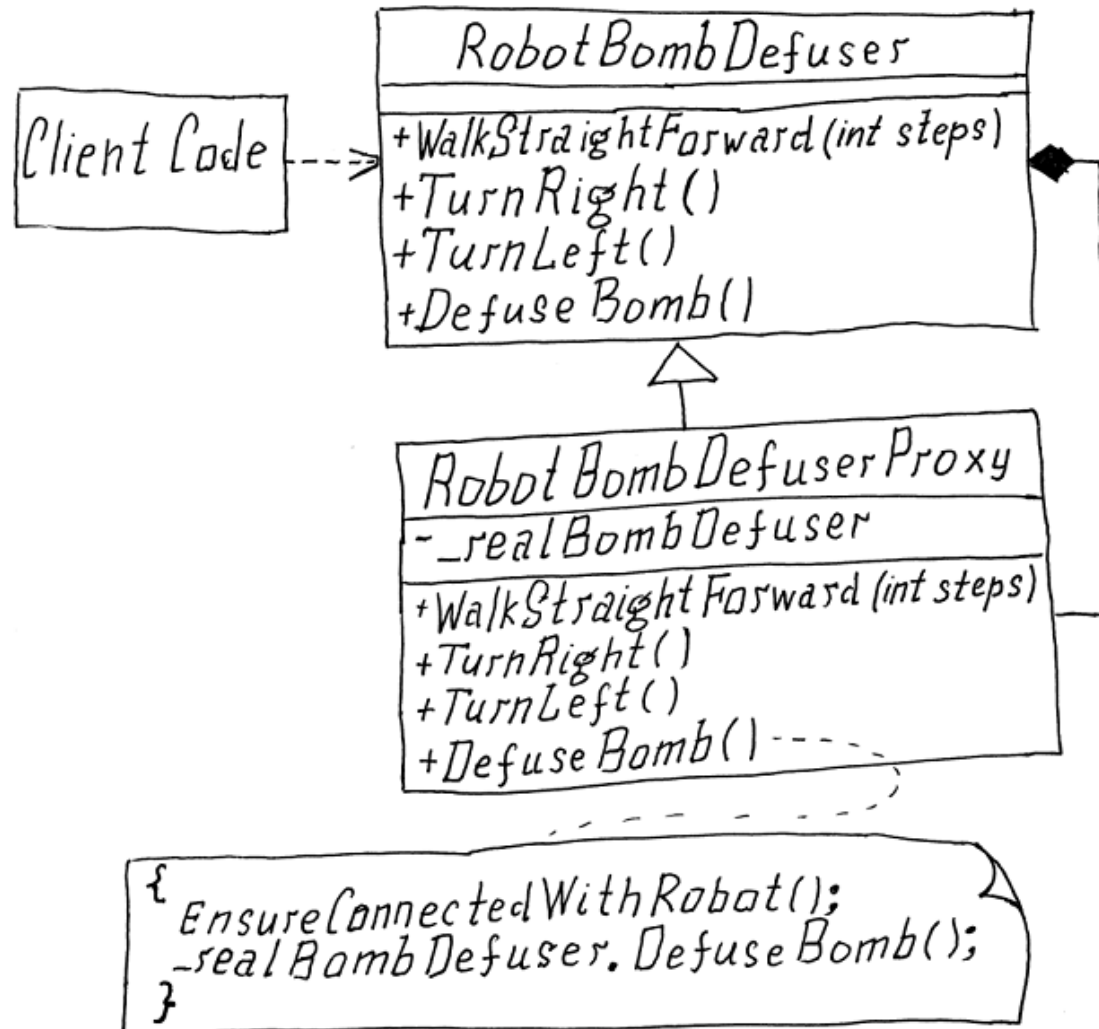
```
public override void DefuseBomb()
{
    EnsureConnectedWithRobot();
    _robotBombDefuser.DefuseBomb();
}
private void EnsureConnectedWithRobot()
{
    if (_robotBombDefuser == null)
    {
        _robotBombDefuser = new RobotBombDefuser();
        _robotBombDefuser.ConnectWireless(_communicationWaveLength);
    }
    for (int i = 0; i < _connectionAttempts; i++)
    {
        if (_robotBombDefuser.IsConnected() != true)
        {
            _robotBombDefuser.ConnectWireless(_communicationWaveLength);
        }
        else
        {
            break;
        }
    }
    if (_robotBombDefuser.IsConnected() != true)
    {
        throw new BadConnectionException("No connection with remote bomb
diffuser robot could be made after few attempts.");
    }
}
}
```

Заступник

```
public static void Run()
{
    int opNum = 0;
    try
    {
        var proxy = new RobotBombDefuserProxy(41);
        proxy.WalkStraightForward(100);    opNum++;
        proxy.TurnRight();                opNum++;
        proxy.WalkStraightForward(5);      opNum++;
        proxy.DefuseBomb();                opNum++;
        Console.WriteLine();
    }
    catch (BadConnectionException e)
    {
        Console.WriteLine("Exception has been caught with message: ({0}). Decided to have human
operate robot there.", e.Message);
        PlanB(opNum);
    }
}

private static void PlanB(int nextOperationNum)
{
    RobotBombDefuser humanOperatingRobotDirectly = new RobotBombDefuser();
    if (nextOperationNum == 0)
    {
        humanOperatingRobotDirectly.WalkStraightForward(100);
        nextOperationNum++;
    }
    if (nextOperationNum == 1)
    {
        humanOperatingRobotDirectly.TurnRight();
        nextOperationNum++;
    }
    if (nextOperationNum == 2)
    {
        humanOperatingRobotDirectly.WalkStraightForward(5);
        nextOperationNum++;
    }
    if (nextOperationNum == 3)
    {
        humanOperatingRobotDirectly.DefuseBomb();
    }
}
```

Заступник



Ланцюжок обов'язків

Проблема	Запит має бути оброблений кількома об'єктами.
Рекомендації	Логічно використовувати даний паттерн, якщо є більше одного об'єкта, який здатен обробити запит і обробник наперед невідомий (і має бути знайдений автоматично) або якщо весь набір об'єктів, які здатні обробити запит, повинен задаватись автоматично.
Рішення	Зв'язати об'єкти-отримувачі в ланцюжок і передати запит вздовж цього ланцюжка, поки він не буде оброблений. "Обробник" визначає інтерфейс для обробки запитів, і, можливо, реалізує зв'язок з наступником, "КонкретнийОбробник" обробляє запит, за який відповідає, має доступ до свого наступника ("КонкретнийОбробник" скеровує запит до свого наступника, якщо не може обробити запит самостійно).
Переваги	Послаблюється зв'язність (об'єкт не зобов'язаний "знати", хто саме обробить його запит).
Недоліки	Нема гарантій, що запит буде оброблений, оскільки він не має конкретного отримувача.

ЛАНЦЮЖОК ОБОВ'ЯЗКІВ

```
abstract class WierdCafeVisitor
{
    public WierdCafeVisitor CafeVisitor { get; private set; }
    protected WierdCafeVisitor(WierdCafeVisitor cafeVisitor)
    {
        CafeVisitor = cafeVisitor;
    }
    public virtual void HandleFood(Food food)
    {
        // If I cannot handle other food, passing it to my successor
        if (CafeVisitor != null)
        {
            CafeVisitor.HandleFood(food);
        }
    }
}

class Girlfriend : WierdCafeVisitor
{
    public Girlfriend(WierdCafeVisitor cafeVisitor)
        : base(cafeVisitor)
    {
    }

    public override void HandleFood(Food food)
    {
        if (food.Name == "Cappuccino")
        {
            Console.WriteLine("GirlFriend: My lovely cappuccino!!!");
            return;
        }
        base.HandleFood(food);
    }
}
```

ЛАНЦЮЖОК ОБОВ'ЯЗКІВ

```
class Me : WierdCafeVisitor
{
    public Me(WierdCafeVisitor cafeVisitor)
        : base(cafeVisitor)
    {
    }
    public override void HandleFood(Food food)
    {
        if (food.Name.Contains("Soup"))
        {
            Console.WriteLine("Me: I like Soup. It went well.");
        }
        base.HandleFood(food);
    }
}

class BestFriend : WierdCafeVisitor
{
    public List<Food> CoffeeContainingFood { get; private set; }
    public BestFriend(WierdCafeVisitor cafeVisitor)
        : base(cafeVisitor)
    {
        CoffeeContainingFood = new List<Food>();
    }
    public override void HandleFood(Food food)
    {
        if (food.Ingradients.Contains("Meat"))
        {
            Console.WriteLine("BestFriend: I just ate {0}. It was testy.", food.Name);
            return;
        }
        if (food.Ingradients.Contains("Coffee") && CoffeeContainingFood.Count < 1)
        {
            CoffeeContainingFood.Add(food);
            Console.WriteLine("BestFriend: I have to take something with coffee. {0} looks
fine.", food.Name);
            return;
        }
        base.HandleFood(food);
    }
}
```

ЛАНЦЮЖОК ОБОВ'ЯЗКІВ

```
class Food
{
    public Food(string name, List<string> ingredients)
    {
        Name = name;
        Ingredients = ingredients;
    }
    public List<string> Ingredients { get; private set; }
    public string Name { get; private set; }
}

class ChainOfResponsibility
{
    public static void Run()
    {
        var cappuccino1 = new Food("Cappuccino", new List<string> { "Coffee", "Milk", "Sugar" });
        var cappuccino2 = new Food("Cappuccino", new List<string> { "Coffee", "Milk" });
        var soup1 = new Food("Soup with meat", new List<string> { "Meat", "Water", "Potato" });
        var soup2 = new Food("Soup with potato", new List<string> { "Water", "Potato" });
        var meat = new Food("Meat", new List<string> { "Meat" });

        var girlFriend = new GirlFriend(null);
        var me = new Me(girlFriend);
        var bestFriend = new BestFriend(me);

        bestFriend.HandleFood(cappuccino1);
        bestFriend.HandleFood(cappuccino2);
        bestFriend.HandleFood(soup1);
        bestFriend.HandleFood(soup2);
        bestFriend.HandleFood(meat);
    }
}
```

ЛАНЦЮЖОК ОБОВ'ЯЗКІВ

BestFriend: I have to take something with coffee. Cappuccino looks fine.

GirlFriend: My lovely cappuccino!!!

BestFriend: I just ate Soup with meat. It was tasty.

Me: I like Soup. It went well.

BestFriend: I just ate Meat. It was tasty.

Команда

Проблема	Потрібно надсилати об'єктам запити, нічого не знаючи про те, виконання якої операції запрошено і хто є одержувачем.
Рішення	Інкапсулювати запит як об'єкт, що дозволить задавати параметри клієнтів для обробки відповідних запитів, ставити запити в чергу, протоколювати їх та підтримувати відміну операцій.

Команда

```
class CommandDemo
{
    public static void Run()
    {
        var customer = new Customer();
        // із певних міркувань, босс завжди знає, що грошей стає тільки на бригаду Z
        var team = new Team("Z");
        // також отримав список вимог, що треба буде передати бригаді
        var requirements = new List<Requirement>() { new Requirement("Cool web site"), new
Requirement("Ability to book beer on site") };
        // ви повинні бути готові бути викликаним замовником
        ICommand commandX = new YouAsProjectManagerCommand(team, requirements);

        customer.AddCommand(commandX);

        // в компанії також є програміст-герой, що кодує на швидкості світла
        var heroDeveloper = new HeroDeveloper();
        // босс вирішив віддати йому проект A
        ICommand commandA = new HeroDeveloperCommand(heroDeveloper, "A");

        customer.AddCommand(commandA);

        // як тільки замовник підписує контракт із вашим боссом
        // ваша бригада і програміст-герой готові виконати все що треба
        customer.SignContractWithBoss();
    }
}
```

Команда

```
class Customer
{
    protected List<ICommand> Commands { get; set; }
    public Customer()
    {
        Commands = new List<ICommand>();
    }
    public void AddCommand(ICommand command)
    {
        Commands.Add(command);
    }
    public void SignContractWithBoss()
    {
        foreach (var command in Commands)
        {
            command.Execute(); \\запускає виконання всіх проектів
        }
    }
}
interface ICommand
{
    void Execute();
}
class YouAsProjectManagerCommand : ICommand \\одна реалізація команди
{
    public YouAsProjectManagerCommand(Team team, List<Requirement> requirements)
    {
        Team = team;
        Requirements = requirements;
    }
    public void Execute()
    {
        // реалізація делегує роботу до потрібного отримувача
        Team.CompleteProject(Requirements);
    }
    protected Team Team { get; set; }
    protected List<Requirement> Requirements { get; set; }
}
```


Команда

```
class HeroDeveloperCommand : ICommand \\ інша реалізація команди
{
    public HeroDeveloperCommand(HeroDeveloper heroDeveloper, string projectName)
    {
        HeroDeveloper = heroDeveloper;
        ProjectName = projectName;
    }
    public void Execute()
    {
        // реалізація делегує роботу до потрібного отримувача
        HeroDeveloper.DoAllHardWork(ProjectName);
    }
    protected HeroDeveloper HeroDeveloper { get; set; }
    public string ProjectName { get; set; }
}

class HeroDeveloper
{
    public void DoAllHardWork(string projectName)
    {
        Console.WriteLine("Hero developer completed project ({0}) without requirements in manner
of couple of hours!", projectName);
    }
}

class Requirement
{
    public string UserStory { get; private set; }

    public Requirement(string userStory)
    {
        UserStory = userStory;
    }
}
```

Команда

```
class Team
{
    public string Name { get; private set; }
    public Team(string name)
    {
        Name = name;
    }

    public void CompleteProject(List<Requirement> requirements)
    {
        AnalyzeRequirements(requirements);
        foreach (var requirement in requirements)
        {
            WorkOnRequirement(requirement);
        }
    }

    private void WorkOnRequirement(Requirement requirement)
    {
        Console.WriteLine("User Story ({0}) has been completed", requirement.UserStory);
    }

    private void AnalyzeRequirements(List<Requirement> requirements)
    {
        foreach (var requirement in requirements)
        {
            if (string.IsNullOrEmpty(requirement.UserStory))
            {
                throw new ArgumentException("not enough information on some of the
requirements...");
            }
        }
    }
}
```

Команда

User Story (Cool web site) has been completed

User Story (Ability to book beer on site) has been completed

Hero developer completed project (A) without requirements in manner of couple hours!

Інтерпретатор

Проблема	Є частина, підвладна зміні задачі.
Рішення	Створити інтерпретатор, який вирішує дану задачу.
Приклад	Задача пошуку рядків за зразком може бути вирішена за допомогою створення інтерпретатора, що визначає граматику мови. "Клієнт" будує пропозицію у вигляді абстрактного синтаксичного дерева, у вузлах якого знаходяться об'єкти класів "НетермінальнийВираз" і "ТермінальнийВираз" (рекурсивно), потім "Клієнт" ініціалізує контекст і викликає операцію Розібрати(Контекст). На кожному вузлі типу "НетермінальнийВираз" визначається операція Розібрати для кожного підвиразу. Для класу "ТермінальнийВираз" операція Розібрати визначає базу рекурсії. "АбстрактнийВираз" визначає абстрактну операцію Розібрати, загальну для всіх вузлів в абстрактному синтаксичному дереві. "Контекст" містить інформацію, глобальну по відношенню до інтерпретатора.
Переваги	Граматику стає легко розширювати і змінювати, реалізації класів, що описують вузли абстрактного синтаксичного дерева схожі (легко кодуються). Можна легко змінювати спосіб обчислення виразів.
Недоліки	Супровід граматики з великим числом правил накладний.

Интерпретатор

```
// Context
class CurrentPricesContext
{
    Dictionary<string, int> _prices = new Dictionary<string, int>();

    public CurrentPricesContext()
    {
        _prices.Add("Bed", 3000);
        _prices.Add("TV", 400);
        _prices.Add("Laptop", 1500);
    }

    public int GetPrice(string goodName)
    {
        if (_prices.ContainsKey(goodName))
        {
            return _prices[goodName];
        }
        else
        {
            throw new ArgumentException("Could not get price for the good that is not
registered.");
        }
    }

    public void SetPrice(string goodName, int cost)
    {
        if (_prices.ContainsKey(goodName))
        {
            _prices[goodName] = cost;
        }
        else
        {
            _prices.Add(goodName, cost);
        }
    }
}
```

Интерпретатор

```
abstract class Goods// Abstract expression
{
    public abstract int Interpret(CurrentPricesContext context);
}
// Nonterminal expression
class GoodsPackage : Goods
{
    public List<Goods> GoodsInside { get; set; }
    public override int Interpret(CurrentPricesContext context)
    {
        var totalSum = 0;
        foreach (var goods in GoodsInside)
        {
            totalSum += goods.Interpret(context);
        }
        return totalSum;
    }
}
// Terminal expression // Terminal expression
class TV : Goods
{
    public override int Interpret(CurrentPricesContext context)
    {
        int price = context.GetPrice("TV");
        Console.WriteLine("TV: {0}", price);
        return price;
    }
}
class Laptop : Goods // Terminal expression
{
    public override int Interpret(CurrentPricesContext context)
    {
        int price = context.GetPrice("Laptop");
        Console.WriteLine("Laptop: {0}", price);
        return price;
    }
}
class Bed : Goods // Terminal expression
{
    public override int Interpret(CurrentPricesContext context)
    {
        int price = context.GetPrice("Bed");
        Console.WriteLine("Bed: {0}", price);
        return price;
    }
}
```

Интерпретатор

```

class InterpreterDemo
{
    public static void Run()
    {
        new InterpreterDemo().RunInterpreterDemo();
    }
    public void RunInterpreterDemo()
    {
        var truckWithGoods = PrepareTruckWithGoods(); // create syntax tree that represents sentence
        var pricesContext = GetRecentPricesContext(); // get latest context
        var totalPriceForGoods = truckWithGoods.Interpret(pricesContext); // invoke Interpret
        Console.WriteLine("Total: {0}", totalPriceForGoods);
    }

    private CurrentPricesContext GetRecentPricesContext()
    {
        var pricesContext = new CurrentPricesContext();
        pricesContext.SetPrice("Bed", 400);
        pricesContext.SetPrice("TV", 100);
        pricesContext.SetPrice("Laptop", 500);
        return pricesContext;
    }
    public GoodsPackage PrepareTruckWithGoods()
    {
        var truck = new GoodsPackage() { GoodsInside = new List<Goods>() };
        var bed = new Bed();
        var doubleTriplePackedBed = new GoodsPackage()
        {
            GoodsInside = new List<Goods>() { new GoodsPackage() { GoodsInside = new List<Goods>() { bed } } }
        };
        truck.GoodsInside.Add(doubleTriplePackedBed);
        truck.GoodsInside.Add(new TV());
        truck.GoodsInside.Add(new TV());
        truck.GoodsInside.Add(new GoodsPackage()
        {
            GoodsInside = new List<Goods>() { new Laptop(), new Laptop(), new Laptop() }
        });
        return truck;
    }
}

```


Интерпретатор

Bed: 400
TV: 100
TV: 100
Laptop: 500
Laptop: 500
Laptop: 500
Total: 2100

Ітератор

Проблема	Складений об'єкт, наприклад, список, повинен надавати доступ до своїх елементів (об'єктів), не розкриваючи їхню внутрішню структуру, причому перебирати список вимагається по-різному в залежності від завдання.
Рішення	Створюється клас "Ітератор", який визначає інтерфейс для доступу та перебору елементів, "КонкретнийІтератор" реалізує інтерфейс класу "Ітератор" і стежить за поточною позицією при обході "Агрегату". "Агрегат" визначає інтерфейс для створення об'єкта - ітератора. "КонкретнийАгрегат" реалізує інтерфейс створення ітератора і повертає екземпляр класу "КонкретнийІтератор", "КонкретнийІтератор" відстежує поточний об'єкт в агрегаті і може обчислити наступний об'єкт при переборі.
Переваги	Підтримує різні способи перебору агрегату, одночасно можуть бути активні кілька переборів.

Ітератор

```
class Army \\ створення армії
{
    public Hero ArmyHero;
    public List<Group> ArmyGroups { get; private set; }

    public Army(Hero armyHero)
    {
        ArmyHero = armyHero;
        ArmyGroups = new List<Group>();
    }

    public void AddArmyGroup(Group group)
    {
        ArmyGroups.Add(group);
    }
}

class Group \\ створення груп
{
    public List<Soldier> Soldiers { get; private set; }

    public Group()
    {
        Soldiers = new List<Soldier>();
    }

    public void AddNewSoldier(Soldier soldier)
    {
        Soldiers.Add(soldier);
    }
}
```

Iterator

```
class IteratorDemo
{
    public static void Run()
    {
        var andriybuday = new Hero("Andriy Buday");
        var earthArmy = new Army(andriybuday);

        var groupA = new Group();
        for (int i = 1; i < 4; ++i)
            groupA.AddNewSoldier(new Soldier("Alpha:" + i));

        var groupB = new Group();
        for (int i = 1; i < 3; ++i)
            groupB.AddNewSoldier(new Soldier("Beta:" + i));

        var groupC = new Group();
        for (int i = 1; i < 2; ++i)
            groupC.AddNewSoldier(new Soldier("Gamma:" + i));

        earthArmy.AddArmyGroup(groupB);
        earthArmy.AddArmyGroup(groupA);
        earthArmy.AddArmyGroup(groupC);

        var iterator = new SoldiersIterator(earthArmy); \\ лікується вся армія
        while (iterator.HasNext())
        {
            var currSoldier = iterator.Next();
            currSoldier.Treat();
        }
    }
}
```

Iterator

```
class Soldier
{
    public String Name;
    public int Health;
    private const int SoldierHealthPoints = 100;
    protected virtual int MaxHealthPoints { get { return SoldierHealthPoints; } }

    public Soldier(String name)
    {
        Name = name;
    }
    public void Treat()
    {
        Health = MaxHealthPoints;
        Console.WriteLine(Name);
    }
}

class Hero : Soldier
{
    private const int HeroHealthPoints = 500;
    protected override int MaxHealthPoints { get { return HeroHealthPoints; } }

    public Hero(String name)
        : base(name)
    {
    }
}
```

Iterator

```
class SoldiersIterator
{
    private readonly Army _army;
    private bool _heroIsIterated;
    private int _currentGroup;
    private int _currentGroupSoldier;
    public SoldiersIterator(Army army)
    {
        _army = army;
        _heroIsIterated = false;
        _currentGroup = 0;
        _currentGroupSoldier = 0;
    }
    public bool HasNext()
    {
        if (!_heroIsIterated) return true;
        if (_currentGroup < _army.ArmeyGroups.Count) return true;
        if (_currentGroup == _army.ArmeyGroups.Count - 1)
            if (_currentGroupSoldier < _army.ArmeyGroups[_currentGroup].Soldiers.Count) return true;
        return false;
    }
}
```

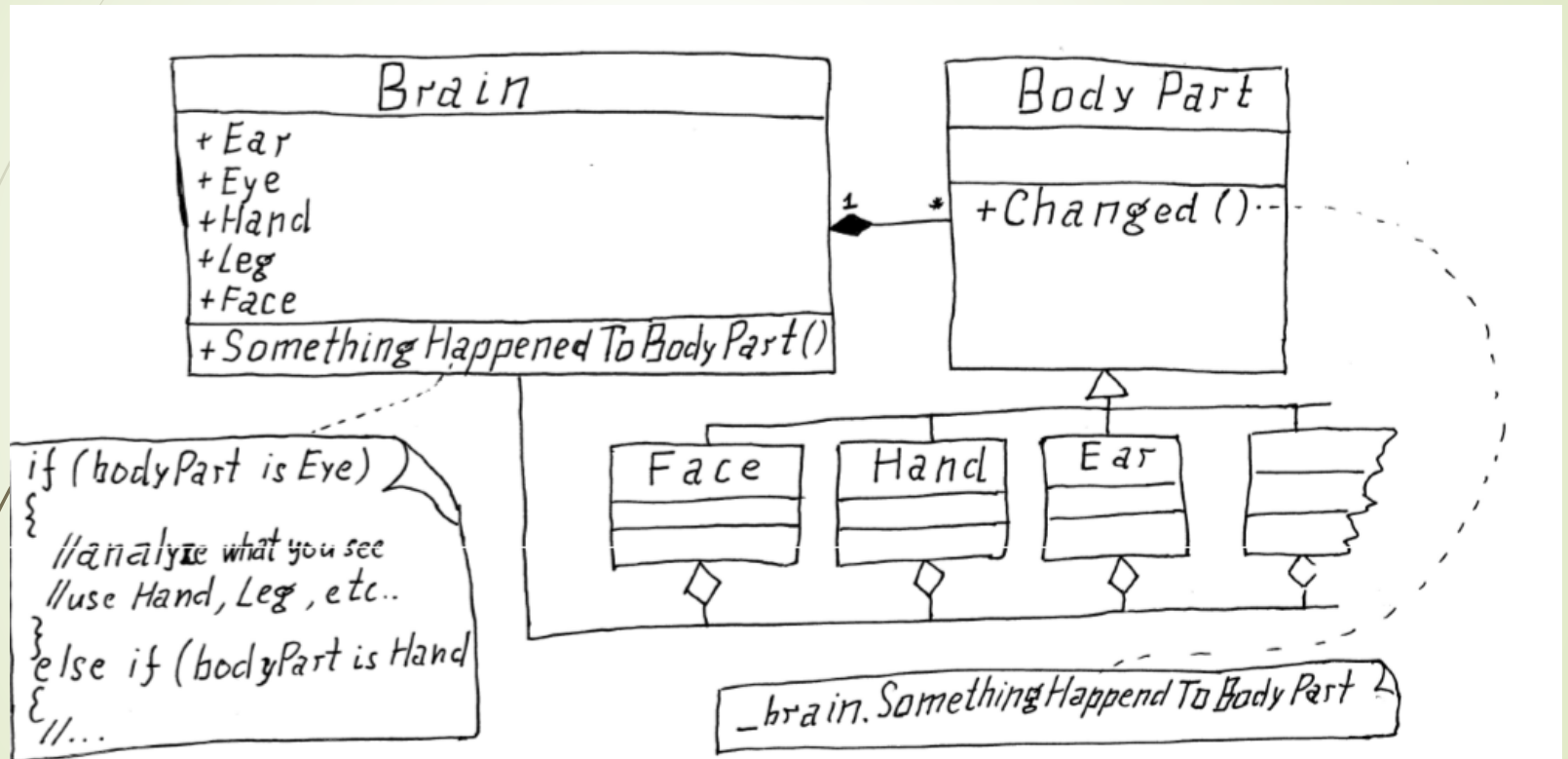
Iterator

```
public Soldier Next()
{
    Soldier nextSoldier;
    // we still not iterated all soldiers in current group
    if (_currentGroup < _army.ArmyGroups.Count)
    {
        if (_currentGroupSoldier < _army.ArmyGroups[_currentGroup].Soldiers.Count)
        {
            nextSoldier = _army.ArmyGroups[_currentGroup].Soldiers[_currentGroupSoldier];
            _currentGroupSoldier++;
        }
        // moving to next group
        else
        {
            _currentGroup++;
            _currentGroupSoldier = 0;
            return Next();
        }
    }
    // hero is the last who left the battlefield
    else if (!_heroIsIterated)
    {
        _heroIsIterated = true;
        return _army.ArmyHero;
    }
    else
    {
        // THROW EXCEPTION HERE
        throw new Exception("End of collection");
        //or set all counters to 0 and start again, but not recommended
    }
    return nextSoldier;
}
```


Медіатор

Проблема	Забезпечити взаємодію безлічі об'єктів, сформувавши при цьому слабку зв'язаність і позбавивши об'єкти від необхідності явно посилатися один на одного.
Рішення	Створити об'єкт, який інкапсулює спосіб взаємодії безлічі об'єктів.
Приклад	"Посередник" визначає інтерфейс для обміну інформацією з об'єктами "Колеги", "КонкретнийПосередник" координує дії об'єктів "Колеги". Кожен клас "Колеги" знає про свій об'єкт "Посередник", всі "Колеги" обмінюються інформацією тільки з посередником, при його відсутності їм довелося б обмінюватися інформацією безпосередньо. "Колеги" посилають запити посереднику і отримують запити від нього. "Посередник" реалізує кооперативне поводження, пересилаючи кожен запит одному або декільком "Колегам".
Переваги	Усувається зв'язаність між "Колегами", централізується управління

Медіатор



Медіатор

```
class Brain // власне медіатор
{
    public Brain()
    {
        CreateBodyParts();
    }

    private void CreateBodyParts()
    {
        Ear = new Ear(this);
        Eye = new Eye(this);
        Face = new Face(this);
        Hand = new Hand(this);
        Leg = new Leg(this);
    }

    public Ear Ear { get; private set; }
    public Eye Eye { get; private set; }
    public Face Face { get; private set; }
    public Hand Hand { get; private set; }
    public Leg Leg { get; private set; }

    public void SomethingHappenedToBodyPart(BodyPart bodyPart)
    {
        if (bodyPart is Ear)
        {
            string heardSounds = ((Ear)bodyPart).GetSounds();

            if (heardSounds.Contains("stupid"))
            {
                // attacking offender
                Leg.StepForward();
                Hand.HitPersonNearYou();
                Leg.Kick();
            }
        }
    }
}
```

Медіатор

```
else if (heardSounds.Contains("cool"))
{
    Face.Smile();
}
}
else if (bodyPart is Eye)
{
    // brain can analyze what you see and
    // can react appropriately using different body parts
}
else if (bodyPart is Hand)
{
    var hand = (Hand)bodyPart;

    bool hurtingFeeling = hand.DoesItHurt();
    if (hurtingFeeling)
    {
        Leg.StepBack();
    }

    bool itIsNice = hand.IsItNice();
    if (itIsNice)
    {
        Leg.StepForward();
        Hand.Embrace();
    }
}
else if (bodyPart is Leg)
{
    // leg can also feel something if you would like it to
}
}
```

Медіатор

```
class BodyPart // Colleague
{
    private readonly Brain _brain;

    public BodyPart(Brain brain)
    {
        _brain = brain;
    }

    public void Changed()
    {
        _brain.SomethingHappenedToBodyPart(this);
    }
}

class Ear : BodyPart
{
    private string _sounds = string.Empty;

    public Ear(Brain brain)
        : base(brain)
    {
    }

    public void HearSomething()
    {
        Console.WriteLine("Enter what you hear:");
        _sounds = Console.ReadLine();

        Changed();
    }

    public string GetSounds()
    {
        return _sounds;
    }
}
```

Медіатор

```
class Face : BodyPart
{
    public Face(Brain brain)
        : base(brain)
    {
    }
    public void Smile()
    {
        Console.WriteLine("FACE: Smiling...");
    }
}

class Eye : BodyPart
{
    private string _thingsAround = string.Empty;
    public Eye(Brain brain)
        : base(brain)
    {
    }

    public void SeeSomething()
    {
        Console.WriteLine("Enter what you see:");
        _thingsAround = Console.ReadLine();

        Changed();
    }
}
```

Медіатор

```
class Hand : BodyPart
{
    private bool _isSoft;
    private bool _isHurting;
    public Hand(Brain brain)
        : base(brain)
    {
    }
    public void FeelSomething()
    {
        Console.WriteLine("What you feel is soft? (Yes/No)");
        var answer = Console.ReadLine();
        if (answer != null && answer[0] == 'Y')
        {
            _isSoft = true;
        }
        Console.WriteLine("What you feel is hurting? (Yes/No)");
        answer = Console.ReadLine();
        if (answer != null && answer[0] == 'Y')
        {
            _isHurting = true;
        }

        Changed();
    }
    public void HitPersonNearYou()
    {
        Console.WriteLine("HAND: Just hit offender...");
    }
    public void Embrace()
    {
        Console.WriteLine("HAND: Embracing what is in front of you...");
    }
    public bool DoesItHurt()
    {
        return _isHurting;
    }
    public bool IsItNice()
    {
        return !_isHurting && _isSoft;
    }
}
```


Медіатор

```
class Leg : BodyPart
{
    public Leg(Brain brain)
        : base(brain)
    {
    }
    public void Kick()
    {
        Console.WriteLine("LEG: Just kicked offender in front of you..");
    }
    public void StepBack()
    {
        Console.WriteLine("LEG: Stepping back...");
    }
    public void StepForward()
    {
        Console.WriteLine("LEG: Stepping forward...");
    }
}
```

Медіатор

```
public static class MediatorDemo
{
    public static void Run()
    {
        var human = new Brain();

        string line;
        AskForInput();
        while (!string.IsNullOrEmpty(line = Console.ReadLine()))
        {
            switch (line)
            {
                case "Ear":
                    human.Ear.HearSomething();
                    break;
                case "Eye":
                    human.Eye.SeeSomething();
                    break;
                case "Hand":
                    human.Hand.FeelSomething();
                    break;
            }
            AskForInput();
        }
    }

    private static void AskForInput()
    {
        Console.ForegroundColor = ConsoleColor.DarkGray;
        Console.WriteLine("Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):");
        Console.ForegroundColor = ConsoleColor.Green;
    }
}
```

Медіатор

Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):

Ear

Enter what you hear:

You are cool!

FACE: Smiling...

Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):

Hand

What you feel is soft? (Yes/No)

Yeah!

What you feel is hurting? (Yes/No)

No

LEG: Stepping forward...

HAND: Embracing what is in front of you...

Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):

Ear

Enter what you hear:

You are dumbass stupid guy!

LEG: Stepping forward...

HAND: Just hit offender...

LEG: Just kicked offender in front of you...

Enter body part ('Ear', 'Eye', 'Hand' or empty to exit):

Hand What you feel is soft? (Yes/No)

No

What you feel is hurting? (Yes/No)

Yes

LEG: Stepping back...

Зберігач

Проблема	Необхідно зафіксувати поведінку об'єкта для реалізації, наприклад, механізму відкату.
Рішення	Зафіксувати і винести (не порушуючи інкапсуляції) за межі об'єкта його внутрішній стан так, щоб згодом можна було відновити в ньому об'єкт. "Зберігач" зберігає внутрішній стан об'єкта "Господар" і забороняє доступ до себе всім іншим об'єктам крім "Господаря", який має доступ до всіх даних для відновлення в колишньому стані. "Посильний" може лише передавати "Зберігача" іншим об'єктам. "Господар" створює "Зберігача", що містить знімок поточного внутрішнього стану і використовує "Зберігач" для відновлення внутрішнього стану. "Посильний" відповідає за збереження "Зберігача", при цьому не робить ніяких операцій над "Зберігачем" і не досліджує його внутрішній вміст. "Посильний" запитує "Зберігач" у "Господаря", деякий час тримає його у себе, а потім повертає "Господарю".
Переваги	Не розкривається інформація, яка доступна тільки "Господарю", спрощується структура "Господаря".
Недоліки	З використанням "Зберігачів" можуть бути пов'язані значні витрати, якщо "Господар" повинен копіювати великий обсяг інформації, або якщо копіювання повинно проводитися часто.

Зберігач

```
public class Caretaker
{
    private readonly GameOriginator _game = new GameOriginator();
    private readonly Stack<GameMemento> _quickSaves = new Stack<GameMemento>();

    public void ShootThatDumbAss()
    {
        _game.Play();
    }

    public void F5()
    {
        _quickSaves.Push(_game.GameSave());
    }

    public void F9()
    {
        _game.LoadGame(_quickSaves.Peek());
        _quickSaves.Peek().GetState().Health = 8;
    }
}
```

Зберігач

```
public class GameOriginator
{
    private GameState _state = new GameState(100, 0); //Health & Killed Monsters

    public void Play()
    {
        //During this Play method game's state is continuously changed
        Console.WriteLine(_state.ToString());
        _state = new GameState((int)(_state.Health * 0.9), _state.KilledMonsters + 2);
    }

    public GameMemento GameSave()
    {
        return new GameMemento(_state);
    }

    public void LoadGame(GameMemento memento)
    {
        _state = memento.GetState();
    }
}

public class GameMemento
{
    private readonly GameState _state;

    public GameMemento(GameState state)
    {
        _state = state;
    }

    protected internal GameState GetState()
    {
        return _state;
    }
}
```

Зберігач

```
public class GameState
{
    public GameState(double health, int killedMonsters)
    {
        Health = health;
        KilledMonsters = killedMonsters;
    }

    public double Health { get; set; }
    public int KilledMonsters { get; set; }

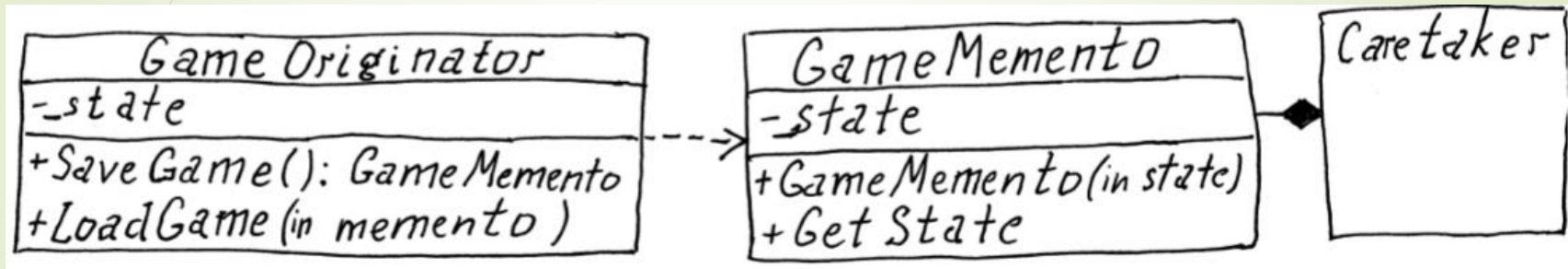
    public override string ToString()
    {
        return string.Format("Health: {0}{1}Killed Monsters: {2}", Health, Environment.NewLine,
KilledMonsters);
    }
}

class MementoDemo
{
    public static void Run()
    {
        var caretaker = new Caretaker();
        caretaker.F5();
        caretaker.ShootThatDumbAss();
        caretaker.F5();
        caretaker.ShootThatDumbAss();
        caretaker.ShootThatDumbAss();
        caretaker.ShootThatDumbAss();
        caretaker.ShootThatDumbAss();
        caretaker.F9();
        caretaker.ShootThatDumbAss();
    }
}
```


Зберігач

Health: 100
Killed Monsters: 0
Health: 90
Killed Monsters: 2
Health: 81
Killed Monsters: 4
Health: 72
Killed Monsters: 6
Health: 64
Killed Monsters: 8
Health: 90
Killed Monsters: 2

Зберігач



Спостерігач

Проблема	Один об'єкт («Спостерігач») повинен знати про зміну станів або деякі події іншого об'єкта. При цьому необхідно підтримувати низький рівень зв'язування з об'єктом – «Спомтерігачем».
Рішення	Визначити інтерфейс «Спостерігача». Об'єкти – спостерігачі реалізують цей інтерфейс і динамічно реєструються для отримання інформації про деяке подію. Потім при реалізації домовленої події сповіщаються всі об'єкти - спосткрігачі.

Спостерігач

```
interface IObserver
{
    void Update(ISubject subject);
}
//
class RiskyPlayer : IObserver
{
    public string BoxerToPutMoneyOn { get; set; }

    public void Update(ISubject subject)
    {
        var boxFight = (BoxFight)subject;

        if (boxFight.BoxerAScore > boxFight.BoxerBScore)
            BoxerToPutMoneyOn = "I put on boxer B, if he win I get more!";
        else BoxerToPutMoneyOn = "I put on boxer A, if he win I get more!";
        Console.WriteLine(BoxerToPutMoneyOn);
        Console.WriteLine("RISKYPLAYER:{0}", BoxerToPutMoneyOn);
    }
}
//
class ConservativePlayer : IObserver
{
    public string BoxerToPutMoneyOn { get; set; }

    public void Update(ISubject subject)
    {
        var boxFight = (BoxFight)subject;

        if (boxFight.BoxerAScore < boxFight.BoxerBScore)
            BoxerToPutMoneyOn = "I put on boxer B, better be safe!";
        else BoxerToPutMoneyOn = "I put on boxer A, better be safe!";

        Console.WriteLine("CONSERVATIVEPLAYER:{0}", BoxerToPutMoneyOn);
    }
}
```

Спостерігач

```
interface ISubject
{
    void AttachObserver(IObserver observer);
    void DetachObserver(IObserver observer);
    void Notify();
}
class BoxFight : ISubject
{
    public List<IObserver> Observers { get; private set; }
    public int RoundNumber { get; private set; }
    private Random Random = new Random();
    public int BoxerAScore { get; set; }
    public int BoxerBScore { get; set; }
    public BoxFight()
    {
        Observers = new List<IObserver>();
    }
    public void AttachObserver(IObserver observer)
    {
        Observers.Add(observer);
    }
    public void DetachObserver(IObserver observer)
    {
        Observers.Remove(observer);
    }
    public void NextRound()
    {
        RoundNumber++;
        BoxerAScore += Random.Next(0, 5);
        BoxerBScore += Random.Next(0, 5);
        Notify();
    }
    public void Notify()
    {
        foreach (var observer in Observers)
        {
            observer.Update(this);
        }
    }
}
```

Спостерігач

```
public static class ObserverDemo
{
    public static void Run()
    {
        var boxFight = new BoxFight();

        var riskyPlayer = new RiskyPlayer();
        var conservativePlayer = new ConservativePlayer();

        boxFight.AttachObserver(riskyPlayer);
        boxFight.AttachObserver(conservativePlayer);

        boxFight.NextRound();
        boxFight.NextRound();
        boxFight.NextRound();
        boxFight.NextRound();
    }
}
```

Спостерігач

RISKYPLAYER:I put on boxer A, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

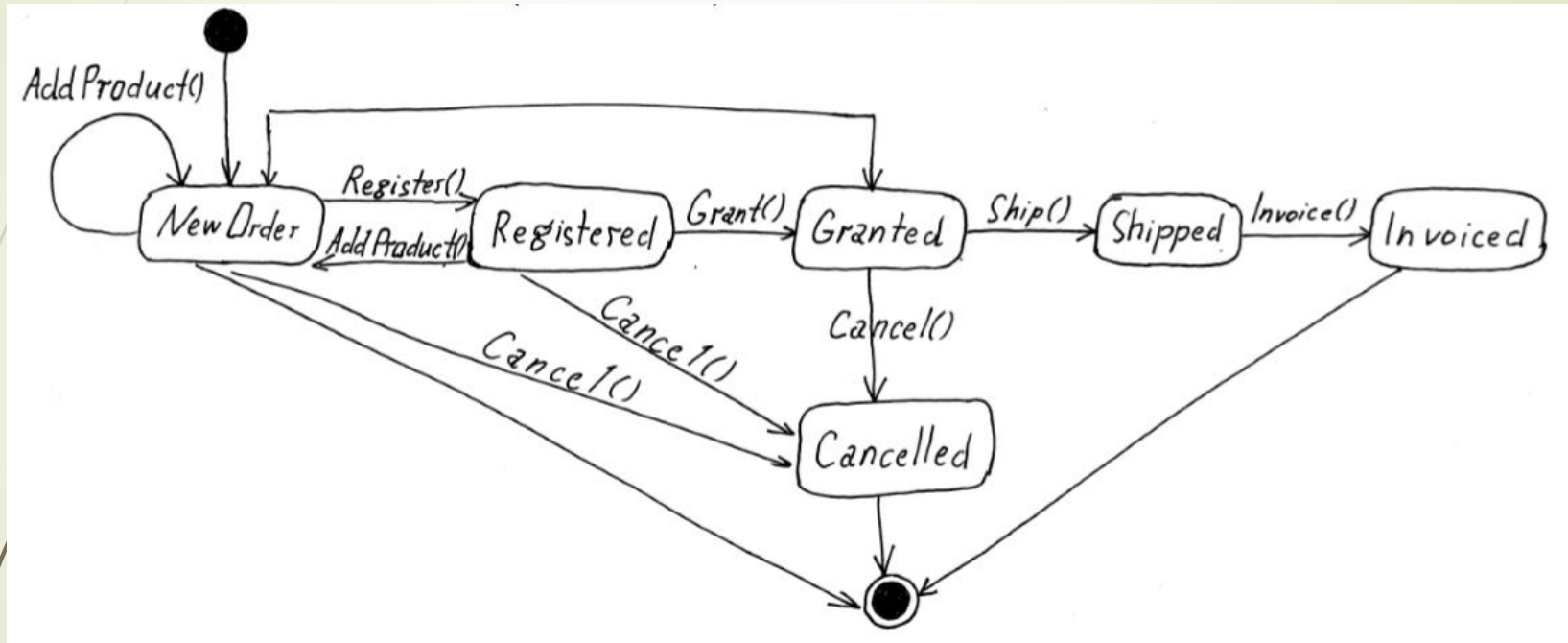
RISKYPLAYER:I put on boxer B, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

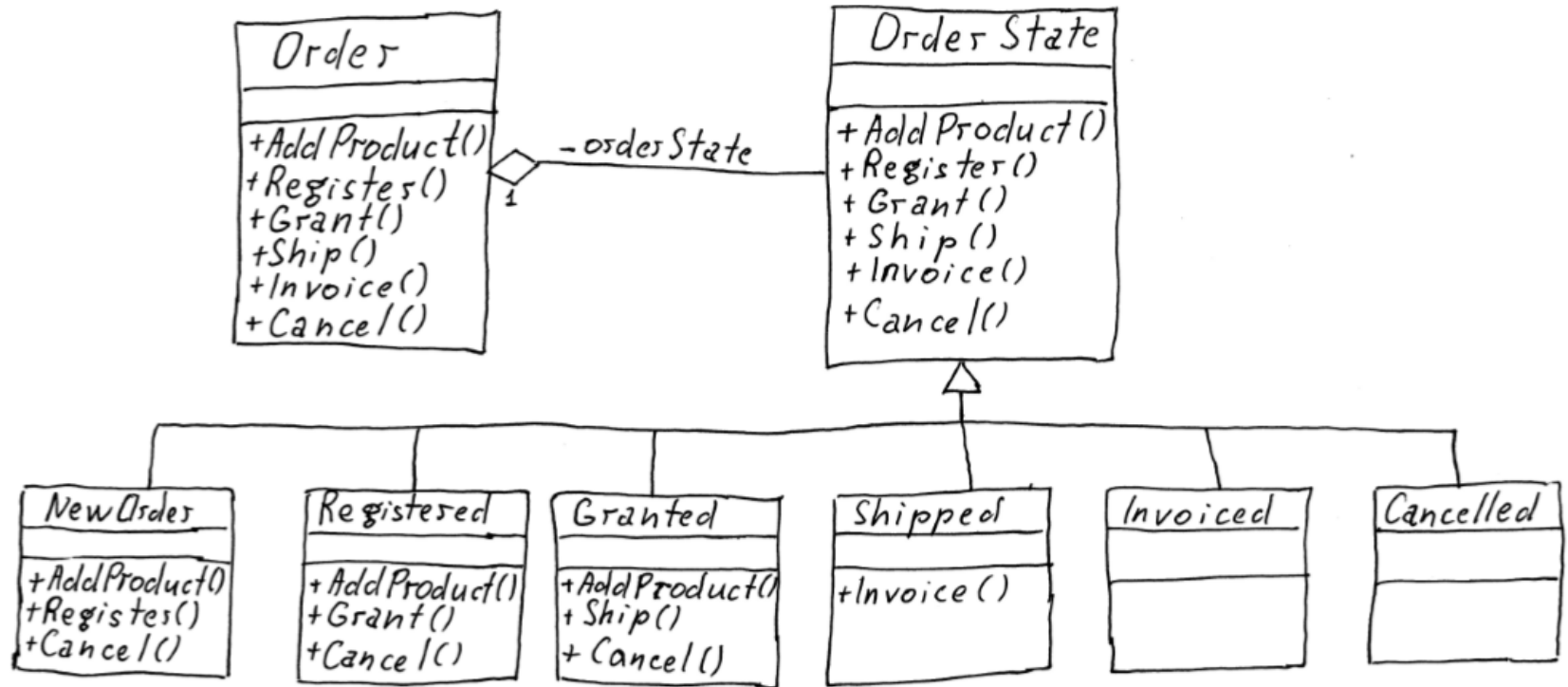
RISKYPLAYER:I put on boxer B, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

RISKYPLAYER:I put on boxer B, if he win I get more!
CONSERVATIVEPLAYER:I put on boxer A, better be safe!

Стан

Проблема	Варіювати поведінку об'єкта в залежності від його внутрішнього стану
Рішення	Клас "Контекст" делегує залежні від стану запити поточному об'єкту "КонкретнийСтан" (зберігає примірник підкласу "КонкретнийСтан", яким визначається поточний стан), і визначає інтерфейс, що представляє інтерес для клієнтів. "КонкретнийСтан" реалізує поведінку, асоційовану з якимсь станом об'єкта "Контекст". "Стан" визначає інтерфейс для інкапсуляції поведінки, асоційованого з конкретним екземпляром "Контексту".
Переваги	Локалізує залежну від стану поведінку і ділить її на частини, що відповідають станам, переходи між станами стають явними.





CTAH

```
class Product
{
    public string Name { get; set; }
    public int Price { get; set; }
}

class Order
{
    private OrderState _state;
    private List<Product> _products = new List<Product>();
    public Order()
    {
        _state = new NewOrder(this);
    }
    public void SetOrderState(OrderState state)
    {
        _state = state;
    }
    public void WriteCurrentStateName()
    {
        Console.WriteLine("Current Order's state: {0}", _state.GetType().Name);
    }
    public void AddProduct(Product product)
    {
        _products.Add(product);
        _state.AddProduct();
    }
    public void Register()
    {
        _state.Register();
    }
    public void Grant()
    {
        _state.Grant();
    }
    public void Ship()
    {
        _state.Ship();
    }
    public void Invoice()
    {
        _state.Invoice();
    }

    public void Cancel()
    {
        _state.Cancel();
    }
}
```

СТАН

```
public void DoShipping()
{
    Console.WriteLine("Shipping...");
}

public void DoAddProduct()
{
    Console.WriteLine("Adding product...");
}

public void DoCancel()
{
    Console.WriteLine("Cancelation...");
}

public void DoGrant()
{
    Console.WriteLine("Granting...");
}

public void DoRegister()
{
    Console.WriteLine("Registration...");
}

public void DoInvoice()
{
    Console.WriteLine("Invoicing...");
}
}
```

CTAH

```
#region States
class Registered : OrderState
{
    public Registered(Order order)
        : base(order)
    {
    }

    public override void AddProduct()
    {
        _order.DoAddProduct();
    }

    public override void Grant()
    {
        _order.DoGrant();
        _order.SetOrderState(new Granted(_order));
    }

    public override void Cancel()
    {
        _order.DoCancel();
        _order.SetOrderState(new Cancelled(_order));
    }
}
class Shipped : OrderState
{
    public Shipped(Order order)
        : base(order)
    {
    }

    public override void Invoice()
    {
        _order.DoInvoice();
        _order.SetOrderState(new Invoiced(_order));
    }
}
```

CTAH

```
class Invoiced : OrderState
{
    public Invoiced(Order order)
        : base(order)
    {
    }
}
class NewOrder : OrderState
{
    public NewOrder(Order order)
        : base(order)
    {
    }

    public override void AddProduct()
    {
        _order.DoAddProduct();
    }

    public override void Register()
    {
        _order.DoRegister();
        _order.SetOrderState(new Registered(_order));
    }

    public override void Cancel()
    {
        _order.DoCancel();
        _order.SetOrderState(new Cancelled(_order));
    }
}
```


CTAH

```
class OrderState
{
    public Order _order;
    public OrderState(Order order)
    {
        _order = order;
    }
    public virtual void AddProduct()
    {
        OperationIsNotAllowed("AddProduct");
    }
    public virtual void Register()
    {
        OperationIsNotAllowed("AddProduct");
    }
    public virtual void Grant()
    {
        OperationIsNotAllowed("Grant");
    }
    public virtual void Ship()
    {
        OperationIsNotAllowed("Ship");
    }
    public virtual void Invoice()
    {
        OperationIsNotAllowed("Invoice");
    }
    public virtual void Cancel()
    {
        OperationIsNotAllowed("Cancel");
    }
    private void OperationIsNotAllowed(string operationName)
    {
        Console.WriteLine("Operation {0} is not allowed for Order's state {1}", operationName,
            this.GetType().Name);
    }
}
```

CTAH

```
class Cancelled : OrderState
{
    public Cancelled(Order order)
        : base(order)
    {
    }
}
class Granted : OrderState
{
    public Granted(Order order)
        : base(order)
    {
    }
    public override void AddProduct()
    {
        _order.DoAddProduct();
    }
    public override void Ship()
    {
        _order.DoShipping();
        _order.SetOrderState(new Shipped(_order));
    }
    public override void Cancel()
    {
        _order.DoCancel();
        _order.SetOrderState(new Cancelled(_order));
    }
}
```

CTaH

```
#endregion States
class StateDemo
{
    public static void Run()
    {
        Product beer = new Product();
        beer.Name = "MyBestBeer";
        beer.Price = 78000;

        Order order = new Order();
        order.WriteCurrentStateName();

        order.AddProduct(beer);
        order.WriteCurrentStateName();

        order.Register();
        order.WriteCurrentStateName();

        order.Grant();
        order.WriteCurrentStateName();

        order.Ship();
        order.WriteCurrentStateName();

        //trying to add more beer to already shipped order
        order.AddProduct(beer);
        order.WriteCurrentStateName();

        //order.Invoice();
        //order.WriteCurrentStateName();
    }
}
```

CTaH

Current Order's state: NewOrder
Adding product...
Current Order's state: NewOrder
Registration...
Current Order's state: Registered
Granting...
Current Order's state: Granted
Shipping...
Current Order's state: Shipped
Invoicing...
Current Order's state: Invoiced

Current Order's state: NewOrder
Adding product...
Current Order's state: NewOrder
Registration...
Current Order's state: Registered
Granting...
Current Order's state: Granted
Shipping...
Current Order's state: Shipped
Operation AddProduct is not allowed for Order's state Shipped
Current Order's state: Shipped

- **Стратегія** зберігає сім'ю алгоритмів і дозволяє змінювати їх незалежно та переключатися між ними.
- **Шаблонний Метод** задає покроково алгоритм, а елементи алгоритму можуть бути довизначені в похідних класах.
- **Відвідувач** (Visitor) дозволяє відділити певний алгоритм від елементів, на яких алгоритм має бути виконаний, таким чином ми можемо легко додати або ж змінити алгоритм без змін до елементів системи.