# Упражнения: Design Patterns

**Решенията на задачите са ви предоставени** почти изцяло (ако срещате затруднения), тъй като задачите от тази тема са много абстрактни. Не е лесно, когато нямате опит, да разберете какво се изисква от вас и как да решите радачата чрез някой **класически шаблон за проектиране**. Решенията са само примерни, така че не е необходимо да ги следвате изцяло.

Няма автоматизирани тестове за това упражнение, така че трябва сами да валидирате кода си.

Състояние на настоящия учебен материал:

A green and blue rectangle with white text

Description automatically generated

## Singleton

Задачата ви е да създадете конзолно приложение, в което се четат всички данни от файл (който се състои от градове и тяхното население), и след това се използват. За начало, нека да създадем един интерфейс:

Graphical user interface, text, application

Description automatically generated

След това създаваме клас, който имплементира интерфейса **ISingletonContainer**. Да го наречем **SingletonDataContainer**:

Graphical user interface, text, application, email

Description automatically generated

Имаме речник, в който са съхранени имената на **столиците** и тяхното **население** от нашия файл. Както виждаме, четем файла в нашия **конструктор**. Сега сме готови да използваме класа, като го инстанциираме. Но няма смисъл да инстанциираме множество пъти един клас, който чете данни от файл, който никога не се променя. Затова ще е полезно да използваме **Singleton** **pattern**. Нека да го имплементираме:

Първо, нека да скрием конструктора от класовете, като го направим частен. След това създаваме единична инстанция на класа чрез свойството **Instance**.

A picture containing text

Description automatically generated

На този етап можем да извикаме свойството **Instance** колкото пъти искаме, но нашият обект ще бъде инстанцииран **само веднъж** и **споделен** при всяко следващо извикване. Можете да го тествате сами:

Graphical user interface, text

Description automatically generated

Резултатът на конзолата е следният:

Text

Description automatically generated

Можем да видим, че извикваме инстанцията четири пъти, но обектът се инициализира само веднъж, което е желаният ефект.

Нека да проверим дали конзолната програма работи:

Text

Description automatically generated

Очакваният изход трябва да бъде нещо подобно:

Text

Description automatically generated

## Façade и Builder

Сега ще погледнем пример за имплементация на **Façade** и **Builder**.

Нека да започнем, като създадем клас, с който да работим:

Graphical user interface, text, application

Description automatically generated

Имаме част с **информация** и част с **адрес** в нашия обект, така че ще използваме два builder-а, за да създадем целия обект.

Нуждаем се от **Façade** – нека да създадем един:

Text

Description automatically generated

Инстанциираме обекта **Car**, който искаме да разкрием чрез метода Build().

Това, от което се нуждаем сега, е да създадем конкретни builder-и. Нека да започнем с **CarInfoBuilder**, който трябва да наследява **Facade** класа:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

През конструктора получаваме обект, който искаме да build-нем и да изполваме интерфейс за building цел.

Нека да направим същото с класа **CarAddresBuilder**:

Graphical user interface, text, application

Description automatically generated

В момента имаме и **двата** builder класа, но все още **не можем** да започнем да създаваме наш обект, защото не сме expose-нали нашите builder-и извън **Facade** класа. Нека да го направим:

Graphical user interface, text, application

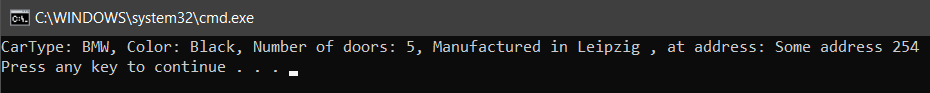
Description automatically generated

Нека да започнем да създаваме обекта:

Text

Description automatically generated with medium confidence

Изходът трябва да бъде:



## Command шаблон

Шаблонът за дизайн **Command** се състои от **Invoker** клас, **Command** клас/интерфейс, **ConcreteCommand** класове и **Receiver** клас.

Ще напишем приложение, в което ще модифицираме цената на продукт, който ще имплементира Command design pattern-а.

Нека да започнем с **Product** receiver клас, който трябва да съдържа базовата бизнес логика на нашето приложение:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Сега **Client** класът може да инстанциира **Product** класа и да изпълни очакваните действия. Но според шаблона за дизайн **Command** **не трябва** да използваме **receiver класовете директно**. Вместо това трябва да извлечем **всички детайли** на заявката в специален клас – **Command**. Нека да го направим.

Първото нещо, което ще направим, е да **добавим интерфейса ICommand**:

Graphical user interface, text

Description automatically generated

Нека да добавим проста **PriceAction** енумерация, за да **енумерираме действията** за модификация на цените:

Text

Description automatically generated

Нека сега да добавим класа **ProductCommand**:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Както можем да видим, класът **ProductCommand** има цялата информация за заявката и на база на нея изпълнява съответното действие.

Да продължим, като добавим клас **ModifyPrice** клас, който ще има ролята на **Invoker**:

Graphical user interface, text, application, email

Description automatically generated

Този клас може да работи с всяка команда, която имплементира интерфейса **ICommand** и да съхранява всички операции.

Нека сега да започнем да работим с **клиентската част**:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Резултатът трябва да е следният:

Text

Description automatically generated

## Прототип

Задачата ви е да създадете конзолно приложение за създаване на сандвичи, имплементирайки шаблона за дизайн **Prototype**.

### 1.1 Абстрактен клас

Първо, нека да създадем абстрактен клас **Sandwich**, който да представлява сандвич, и да дефинираме метод, чрез който абстрактният клас **Sandwich** може да се клонира:

Graphical user interface, text

Description automatically generated

### 1.2 ConcretePrototype участник

Сега се нуждаем от класа **ConcretePrototype**, който може да се клонира, за да създава **повече инстанции** на сандвичи. Нека да кажем, че един сандвич се състои от **четири части**: хляб (bread), месо (meat), кашкавал (cheese) и зеленчуци (veggies):

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

### 1.3 Меню от сандвичи

Нека да създадем клас, който има за цел да **съхранява** сандвичите, които сме направили:

Graphical user interface, text, application

Description automatically generated

### 1.4 Тестване на кода

Сега е време да тестваме това, което сме направили, като се опитаме да го използваме. В **Main()** метода можем да го направим, като **инстанцираме прототипа** и след това го **клонираме**, популирайки нашето **меню със сандвичи**:

A picture containing text

Description automatically generated

Text

Description automatically generated with low confidence

Text

Description automatically generated

## Composite шаблон

Вашата задача е да създадете конзолно приложение, което изчислява **общата цена на подаръците**, които са продадени в даден магазин. Подаръкът може да е **единичен** (напр. играчка) или **комплексен**, който се състои от кутияс две играчкии друга кутия, например с една играчка и друга кутия с играчка вътре. Ще имаме **дървовидна структура**, която да представя комплексните подаръци, така че ще имплементираме шаблона за дизайн **Composite**.

### 1.1 Component

Първо трябва да създадем **абстрактен клас**, който представлява **базовия подарък**. Той трябва да има две полета (**име** и **цена**) и метод, който **изчислява общата цена**. Ще използваме тези полета и метод като интерфейс между **Leaf** и **Composite** частите на нашия шаблон.

Graphical user interface, text, application

Description automatically generated

#### Базови операции

Нека да създадем интерфейс **IGiftOperations**, който ще съдържа две операции – **Add** (добави) и **Remove** (премахни). Създаваме този интерфейс, защото класът **Leaf** няма методи за операции.

Graphical user interface, text

Description automatically generated

### 1.2 Composite клас

Нека сега да създадем класа **CompositeGift**. Той трябва да наследява класа **GiftBase** и да имплементира интерфейса **IGiftOperations**. Следователно имплементацията ще се състои от много обекти от класа

Now you have to create the composite class (**CompositeGift**). It should inherit the **GiftBase** class and **GiftBase**. Методът **Add** ще добавя подарък, а методът **Remove** ще премахва един подарък. Методът **CalculateTotalPrice** ще връща цената на един **CompositeGift**.

Graphical user interface, text, application, email

Description automatically generated

### 1.3 Leaf клас

Нека да създадем Leaf клас (**SingleGift**). Той няма да има поднива, така че няма да изисква операции за добавяне и премахване. Следователно трябва да наследява **само** класа **GiftBase**. Ще бъде като **единичен подарък**, без компонентни подаръци.

Graphical user interface, text, application, email

Description automatically generated

#### Тестване на кода

Сега е време да тестваме това, което сме направили, като се опитаме да го използваме. В метода **Main()** можем да инстанцираме класа **SingleGift** и **CompositeGift** и да използваме техните методи.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

## Template шаблон

Има стотици видове хляб, които се правят по света, но всеки вид включва специфични стъпки, за да се приготви. Вашата задача е да моделирате **няколко различни вида хляб**, които ще използват един и същи **шаблон**, което е добър повод да използваме шаблона за дизайн **Template**.

### 1.1 Абстрактен клас

Първо, нека да създадем абстактен клас (**Bread**), който да представлява **всички видове хляб**, които може да се направят. Трябва да има два абстрактни void метода,  **MixIngredients()** и **Bake()**, един виртуален метод **Slice()** и шаблонния метод - **Make()**.

Graphical user interface, text, application, email

Description automatically generated

### 1.2 Concrete класове

Нека да разширим приложението, като добавим няколко Concrete класа за **различни типове хляб**. Например: **TwelveGrain**, **Sourdough**, **WholeWheat**.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

#### Тестване на кода

Сега е време да тестваме това, което сме направили, като се опитаме да го използваме. В метода **Main()** можем да го направим, като инстанциираме обекти от класовете, които току-що направихме.

A screenshot of a computer

Description automatically generated with low confidence

Text

Description automatically generated