TP 6 : Sherlock at Factory 06

Submission instructions

At the end of the practical, your Git repository must follow this architecture:

```
csharp-tp6-sherlock.holmes/
|-- README
|-- .gitignore
|-- Factory06/
    |-- Factory06.sln
    |-- Factory06/
        |-- Tout sauf bin/ et obj/
```

Do not forget to check the following requirements before submitting your work:

- You shall obviously replace `sherlock.holmes` with your login.
- The `README` file is mandatory.
- There must be no `bin` or `obj` folder in the repository.
- You must respect the prototypes of the given and asked functions.
- Remove all personal tests from your code.
- **The code MUST compile !**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

Contents

1 Introduction

1.1 Objectives

The goal of this practical is to allow you to get more practice using notions in **Object Oriented Programming**, that you discovered in the TP4 such as abstract classes. If you have troubles understanding the concept of **Classes**, we invite you to take a look at the TP4 once more, as this practical will mostly elaborate on those notions.

2 Courses

2.1 Reminder

We present here the concepts approached during the TP C#4. These concepts must be understood.

2.1.1 Classes and Objets

C# is an object oriented language. This means that it is possible to define and instantiate its own **objects**. But what is an **object** exactly ?

An **object** is a data structure described by the **classes** to which it belongs. Be careful, these two concepts are not same. The terms **class** and **object** must not be mixed up. The **classes** describes the structure and the behavior of the **object**. In C#, the **class** is used as a type like `int` or `float`. The variable will then be considered as an **object**.

```
1  class MyClass // This is a class
2  {
3  }
4
5  public static void Main(string[] args)
6  {
7      MyClass foo = new MyClass(); // foo is an object
8  }
```

In the example above, foo is an **object** of **class** MyClass. But there are other ways to describe the relationship between the class MyClass and its object **foo**:

- **foo** is an **instance** of **MyClass**.
- **MyClass** is the type of **foo**.

2.1.2 Encapsulation

The **classes** are used to reunite and link data (the **attributes**) and behaviors (the **methods**). This is called `encapsulation`. Of course, it is possible to modify the **attributes** of an **objet** from a **method**.

2.1.3 Abstraction

Abstract classes are classes which cannot be used to instantiate objects. They can be used to define a common structure to allow some other **classes** to inherit from them. Then, you can modify objects from different classes thanks to their common abstract parent class. It is also possible to prevent a class from being a parent class. This class must be created with the keyword **sealed**.

Here is an example of an abstract class:

```
1  public abstract class Vehicle // The abstract class
2  {
3      // Common attributes
4      protected int WheelsNumber;
5      protected float MaxSpeed;
6      protected float PosX;
7      protected float PosY;
8
9      // Common Methods
10     public abstract void Move(float speedX, float speedY);
11 }
```

We can notice the keyword **abstract** which completes, not only the definition of the class, but also the prototypes of the methods specific to this class which require a different implementation in the child classes.

Because this class is abstract, it cannot be instantiated. In order to be able to implement the abstract methods, you must use the keyword **override**.

Warning

These abstract methods declared in the parent class must be implemented in the child classes (If they are no abstract).

Going further

`https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract`

2.1.4 Inheritance

The Inheritance is a notion of object-oriented programming that will complement the use of our abstract classes. In fact, inheritance allows you to create classes that are called **child classes** which depend on a **parent class**. This then makes it possible to define the daughter class from a base provided by the parent class, the methods of the latter existing by default in the child classes.

Here is an example of inheritance (with the Vehicle class):

```
public class Monocycle : Vehicle
{
    // Attribute specific to the Monocyle class
    private bool Crashed = false;

    // Monocycle class' Constructor
    public Monocycle(float MaxSpeed, float x, float y)
    {
        this.MaxSpeed = MaxSpeed;
        this.PosX = x;
        this.PosY = y;
        this.WheelsNumber = 1;
    }

    // Implementation specific to the Monocycle class
    // Do not forget the override keyword
    public override void Move(float speedX, float speedY)
    {
        this.PosX += speedX;
        this.PosY += speedY;
        Crashed = true;
    }
}
```

```
public class Car : Vehicle
{
    // Attribute specific to the Car class
    public string BrandName;

    // Car class' Constructor
    public Car(string brand, float MaxSpeed, float x, float y)
    {
        this.BrandName = brand;
        this.WheelsNumber = 4;
        this.MaxSpeed = MaxSpeed;
        this.PosX = x;
        this.PosY = y;
    }

    // Implementation specific to the Car class
    // Do not forget the override keyword
    public override void Move(float speedX, float speedY)
    {
        PosX += speedX;
        PosY += speedY;
    }
}
```

2.1.5 Enum

An `num` represents a list of states that an associated variable can take.

```
enum Card
{
    Spade, // Implicitly associated with the value 0
    Diamond, // Implicitly associated with the value 1
    Heart, // Implicitly associated with the value 2
    Club, // Implicitly associated with the value 3
    Joker = 9, //Explicitly associated with the value 9
    Unknown // Implicitly associated with the value 10
}
```

As you can see, states have names (Spades, Diamonds, etc, ...) and they are associated with numbers. By default, if no value is specified, the value of the first argument will be 0. Then the following ones will have the successive value of the previous state. On the other hand, we can also force a state to have a precise value associated (as above with the Joker).

```
Card tmp = Card.Spade;

if (tmp == Card.Heart)
    Console.WriteLine("I'm a Heart");
else if (tmp == (Card) 0)
    Console.WriteLine("I'm a Spade"); // prints "I'm a Spade"
else
    Console.WriteLine("I'm not a Heart nor a Spade");
```

The different states can be compared with each other either by name or with its associated value. The advantage of assigning a name to these states is for the sake of readability and understanding the code. The use of numbers is less relevant.

Going further

`https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum`

2.2 List

It is important to know what is a list in C# for this practical.

It is a class, **List <T>**, which provides methods to search, sort, and manipulate lists. A list, by contradiction to an array, is not of fixed size because its size varies.
In a list, every element is of the same type. Therefore it is not possible to have a List containing integers and string at the same time. Each element in a List can be accessed through its index. The first one being zero and the lastest is the size of the List minus one. Here is an example of the use of a list :

```
List<string> colors = new List<string>(); // colors : {}
colors.Add("Blue");       // colors : {"Blue"}
colors.Add("Red");        // colors : {"Blue", "Red"}
colors.Add("Green")       // colors : {"Blue", "Red", "Green"}

int length = colors.Count;
// The variable length has a value of 3.
```

The "colors" list created above has 3 elements, and these are all `strings`. It is not possible to add an element of another type to this list. Once the list has been created, it is of course possible to remove elements from it with the method `Remove`.

```
// We are still using the same list previously created.
// colors : {"Blue", "Red", "Green"}

colors.Remove("Green");      //Remove the first encountered "Green" element.
// colors: {"Blue", "Red"}

colors.RemoveAt(0);          //Remove the element at the index 0
// colors: {"Red"}
```

Note

Using the *Remove* method on a non-existent item in the list has no effect.

Carefull

Using the *RemoveAt* method on a non-existent index in the list raise an error.

Finally, to modify an element at a specific index, we can use the following method:

```
//colors : {"Blue", "Red", "Green"}
colors[2] = "Yellow";
//colors: {"Blue", "Red", "Yellow"}
```

Index

Similar to the *RemoveAt* method, using the method on a non-existent index will raise an error.

One last remark, the lists are located in memory blocks, and are accessible by pointers (the memory and reference lessons can be found in the TP3 C#). This means that copying a list in the following way will not work:

```
1  List<string> myList = new List<string>();
2  myList.Add("One");
3  myList.Add("Two");
4  myList.Add("Three");
5
6  List<string> otherList = myList;
7  otherList[0] = "Four";
8  // Now both myList and otherList have "Four" as their first element.
```

Changing one list will change the other. To copy a list without it being linked, you have to create a new list and copy each element from the first to the second.

To go further

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-6.0

3 Exercise : Factory06

3.1 Introduction

Sherlock Holmes has run out of coat, flask, and hat. He has unfortunately lost all of his stuff during his last investigation ...
But luckily you're here! You will be able to help him from your position as the director of the **Factory 06**, well known for its production of Sherlock's favorite stuff.

The **Factory06** is an important industrial complex which manages many machines producing, among other things, coats, flasks and hats. In this factory, you are able to build new production machine, in order to produce specific items that you can then resell !

Your objective is to collect as much money as possible in a finite amount of time.

3.1.1 Objectives

In this exercise, you will have to implement the **Factory 06**, then a program capable of managing the factory as best as possible (a bot). For this, a precise architecture has been put in place to prevent the bot from being able to cheat. Like for example, not being able to modify the remaining money.

3.1.2 Description of the factory

The factory is represented by a list of **Machine** (each with a specific role), and a certain amount of money that represents the funds available.

There are three types of Machine:

- "*Hat*": Machine that produces hats, at a low price.
- "*Flask*": Machine that produces flasks, at a high price.
- "*Coat*": Machine that produces coats, at a medium price.

The bot can perform 5 actions :

- Create a Machine "*Build*": if the factory has room and means, it is possible to create one.
- Upgrade a Machine "*Upgrade*": each Machine can be improved to have certain advantages.
- Destroy Machine "*Destroy*": each Machine can be destroyed for free.
- Start a Machine "*Produce*": if the factory has the means, it can produce the objects corresponding to the Machine.
- Sell Machine Items "*Sell*": each object produced by a Machine can be sold at a certain price.

3.2 Item

This part is necessary to move on to the next. The goal is to implement the representation of the objects that will be produced in the factory. For this, we will use 2 elements: a class (Item) which represents the produced object, and an enumeration (ItemType) which represents the differents objects.

The `ItemType` enumeration is in the file **ItemType.cs**, and the `Item` class in the file **Item.cs**.

Reminder

To access a state of the `ItemType` enumeration, for example Hat, you must do `ItemType.Hat`.

An Item contains two attributes:

- A "*price*": the price of the item.
- A "*type*": its type of object (hat, etc).

You must start by adding a **getter** on the price and type of the Item, in order to be able to access it from the corresponding Machine.

```
1  private readonly uint price;
2  private ItemType type;
3
4  // TODO
5  // Add a getter Price
6  // Add a getter Type
```

You can now implement the constructor of the class `Item`. It must initialize all its attributes.

```
1  public Item(ItemType type)
2  {
3      // TODO
4  }
```

Reminder enum

Remember that all states of an enumeration have a value ! Consider looking at the `ItemType.cs` file.

All you have to do is implement the single method of the `Item` class, **Sell**. This method returns the price of the Item once sold.

Knowing that :

- A `ItemType` chapeau "*Hat*" is worth 3 times its cost.
- A `ItemType` "*Flask*" is worth 6 times its cost.
- A `ItemType` "*Coat*" is worth 4 times its cost.

You have completed the Item part, excellent! We must now move on to step above: Machine.

3.3 Machine

Just like the Item part, this part is necessary to move on to the next. The objective is to implement the 3 different types of Machine that the `Factory 06` contains (which produce the Items mentioned in the previous part).

For this, you have the abstract class **Machine** in the file **Machine.cs** which will serve as mother class for the 3 Machines that you will need to implement.

Let's look at what a Machine is made of:

- A list of Item **items** which represents all the Items that the machine produced (and which are not yet sold).
- A state of the MachineType **type** enum which allows us to distinguish which Item this machine produces.
- An integer **level** that represents the current level of the machine (a level starts at 1 and each Machine has a maximum level).
- An uint **capacity** which represents the maximum number of Item that the machine can hold.

As with the `Item` class, you must first add **getters** for all these attributes so that you can access them from your Factory (the `Factory` class).

```
protected List <Item> items;
protected MachineType type;
protected int level;
protected uint capacity;

// TODO
// Add a getter Items
// Add a getter Type
// Add a getter Level
// Add a getter Capacity
```

We can now move on to the implementation of the methods of the abstract class `Machine`. To do this, let's start the **Hat** class which is in the file `Hat.cs`. This class implements the abstract class `Machine`, and will produce Item of type *ItemType.Hat*. We can find the abstract methods declared in the Machine class:

```
public abstract bool Upgrade(ref long money);
public abstract bool Produce(uint count, ref long money);
public abstract void Clear();
public abstract uint Destroy();
```

But we have two more attributes:

- An array of uint *upgrades* which represents the cost of each of the Hat Machine upgrades.
- An integer *maxLevel* which represents the maximum possible level for the Machine Hat.

For these attributes, we don't need a `getter`. Indeed, their use is specific to the machine, and the Factory or other Machines have no interest in knowing their values.

You can therefore go directly to the implementation of the constructor of the Machie **Hat** which must initialize all the attributes of the Machine **Hat**. In the case of a Machine **Hat**, the capacity at level 1 is 300.

```
1 public Hat()
2 {
3     // TODO
4 }
```

Reminder

Remember that `Hat`, is a class that inherits from the abstract class **Machine**.

As said above, the levels of a machine start at one, and not zero.

The `Upgrade` method makes it possible to improve the level of the Machine (here, Hat). It takes as a parameter a long, as a reference, which corresponds to the remaining money from the Factory. The method should return true if the improvement is possible, false otherwise. In addition, if the upgrade is possible, it should subtract the cost of the upgrade to *money*, as well as increase the capacity and level of the Machine. Regarding the Machine Hat, its capacity increases by 300 at level 2, otherwise it doubles.

```
1 public override bool Upgrade(ref long money)
2 {
3     // TODO
4 }
```

Conseil

You must use the constant array `upgrades` declared above. It corresponds to the prices of each improvement according to the level, up to the maximum level. But don't forget that the levels start at one.

You then need to implement the `Produce` method, which is essential for the proper functioning of your Factory. This method returns true if the Machine succeeded in producing *count* Item, based on the remaining money of the Factory: *money*. For each Item produced it is also necessary to subtract the cost of the latter to *money*. We will assume that a production request for 0 Item does not produce anything, and therefore returns false.

```
1 public override bool Produce(uint count, ref long money)
2 {
3     // TODO
4 }
```

The next method, `Clear`, removes the list of `Items` that the Machine **Hat** contains. This will be useful to you right after, when you wish to destroy a Machine.

```
1  public override void Clear()
2  {
3      // TODO
4  }
```

Advice

Remember to look at the different methods available on the class **List**. This function is only about calling a specific method.

Warning

We only ask you to clean the list, which means the machine remains usable after a clear. There is just no more Item in his Item list.

Finally, implement the `Destroy` method which must, like its name indicates, destroy the Machine. This involves deleting all of its items. This method also returns a third of the machine's price (you can find the values in the MachinePrice enumeration in the file `MachinePrice.cs`), since, as a good director, you waste nothing. So you can recycle part of this Machine and its `Items` (like the previous method, this implies that the Item list does not have to be destroyed, become null).

```
1  public override uint Destroy()
2  {
3      // TODO
4  }
```

Once the **Hat** class is complete, you need to implement the **Coat** and **Flask** classes which are respectively in the files, `Coat.cs` and `Flask.cs`.

The only differences between these Machines are the following:

- Machine Hat : original capacity of 300, increased its capacity by 300 at level 2. Machine price: 90.
- Machine Coat : original capacity of 30, increased its capacity by 10 at level 2. Price of the machine: 120.
- Machine Flask : original capacity of 20, increased its capacity by 4 at level 2. Machine price: 200.

3.4 Factory

We are now going to take care of the Factory itself, it is the class **Factory** which is located in the file `Factory.cs`. This class contains three attributes:

- A long *money*, representing the remainig money of the factory.
- An integer *maxNbMachine* which represents the maximum capacity Factory machine storage.
- A list of Machine *machines* which groups together all the Machines used in the factory.

Of these three attributes, only one requires a getter: **money**.

```
1  private long money;
2  private readonly int maxNbMachine = 50;
3  private List <Machine> machines;
4
5  // TODO
6  // Add a getter Money
```

Once done, you can implement the constructor of the class. **Factory**, by properly initializing all the attributes.

```
1  public Factory(long initialMoney)
2  {
3      // TODO
4  }
```

The first two methods will be useful for implementing some methods thereafter. The first is *GetMatchMachines*. This method returns a list only composed of Machines which have the same MachineType *type* than the one passed in parameter. To do this you must therefore create a list, and return it at the end.

```
1  public List <Machine> GetMatchMachines(MachineType type)
2  {
3      // TODO
4  }
```

The next method is *FindAvailableMachine*. Like his name indicates, returns the first Machine that is available and of the same MachineType than the *type* passed as a parameter. Available means that the Machine in question can produce `at least` one Item. If no machine is available, you must return *null*.

```
1  public Machine FindAvailableMachine(MachineType type)
2  {
3      // TODO
4  }
```

Let's move on to the first method that modifies your Factory! This is the method *Build*. The latter returns true if a Machine of MachineType *type* can be built in the factory, otherwise false. Do not forget to take into account the maximum capacity of the Factory. In addition, this method updates the remaining money from the Factory if the construction is doable.

```
1  public bool Build(MachineType type)
2  {
3      // TODO
4  }
```

The *Produce* method allows you to start your factory. It tries to produce *count* Item from Machine of MachineType *type*. This method returns true only if *count* Item have been produced, otherwise false.

```
1  public bool Produce(MachineType type, int count)
2  {
3      // TODO
4  }
```

Warning

A negative production request is, of course, not valid.

The next methods are divided into two parts: the **All** and the **Match**. The only difference is that the *All* applies the method on all Machines, whereas the *Match* olny applies it on a specific part.

The *UpgradeAll* and *UpgradeMatch* methods respectively allow to improve `all` and `some` Machines. They return true if all the desired upgrades have been made.
Which means all the Machines have been improved in the case of **UpgradeAll**. And in the case of **UpgradeMatch**, that *count* Machines which have for MachineType *type*, as well as the same *level* given as parameter have been upgraded.

```
1  public bool UpgradeAll()
2  {
3      // TODO
4  }
5
6  public bool UpgradeMatch(MachineType type, int level, int count)
7  {
8      // TODO
9  }
```

The next two methods are *DestroyAll* and *DestroyMatch*. Their job is, respectively, to destroy *all* and *some* Machines (those with the same MachineType as *type* passed as a parameter). Both methods return an uint, which represents the money gained from these destructions of Machines. If no Machine is destroyed, they return zero. In addition, they need to update the remaining money from the factory.

```
public uint DestroyAll()
{
    // TODO
}

public uint DestroyMatch(MachineType type)
{
    // TODO
}
```

You have previously implemented the *Produce* method to create Items. You must now implement the methods *CollectAll* as well as *CollectMatch*. They respectively allow to recover the Items of *all* Machines, and of *some* Machines (which have the same MachineType as the *type* in parameter). For this you will need to create a list of Items, which you will return when completed. These two methods will be useful for the following ones.

```
public List <Item> CollectAll()
{
    // TODO
}

public List <Item> CollectMatch(MachineType type)
{
    // TODO
}
```

You can now move on to the methods *SellAll* and *SellMatch*. These two methods will respectively collect *all* and *some* Items of the Factory's Machines. Then sell them, in order to return the total money earned. If none are sold, they therefore return zero. In addition, they must update the remaining money of the factory.

```
public uint SellAll()
{
    // TODO
}

public uint SellMatch(MachineType type)
{
    // TODO
}
```

Warning

Once the Items are sold, they are no longer in the factory. This means that the Machines which have their Items sold must not have them in their lists of Items after the sale!

You have previously implemented the methods to sell your Items, the last two methods *ClearAll* and *ClearMatch* can simplify this task for you. They respectively allow to remove *all*, and *some* Items of the Plant Machines.

```
1  public void ClearAll()
2  {
3      // TODO
4  }
5
6  public void ClearMatch(MachineType type)
7  {
8      // TODO
9  }
```

At this stage, you must therefore have a functional factory! You still have to do all the necessary actions to make it work by yourself of course. But it's a great stopover!
All you have to do now is implement the `Bot` part of this practical.

3.5 Bot

While your goal is to gain as much money as possible within a finite time, you will need to create a program tha will do the job for you. This part is about its creation, and also the implementation of the class *Game* which will simulate the famous Factory 06 for your bot!

3.5.1 Game

The *Game* can be found in the file **Game.cs**, it contains more or less the same methods as the *Factory* class. Indeed, this class is just a wrapper around the *Factory* class.
There are various reasons for this choice:

- The clanliness of the code : they are already a lot of methods in the Factory class, adding more of them could make it easier to get lost while browsing the code. Moreover, some attributes we want to add in the `Game` class are not relevant for the `Factory` class.
- Restrict the modifcations made to the factory : the `Factory` class is not supposed to be accessible from the bot itself. Otherwise, it could directly modify it, while we want him to follow the rules we made.
- Consistency of attributes and methods : the *Factory* class represents a factory, whereas the *Game* class represents a simulation of a factory. It means with this class we will "play" with the factory, with game round (and a limited number of them), for example. It is therefore not consistent to have attributes such as the current round number in the `Factory` class.

Most of the methods in the Game class are already implemented. The only ones not implemented are the following:

Getters : there are three attributes, and two of them need a `getter`. The integer *nbRound* representing the maximum number of round on which the simulation will be ran. Another integer, *round*, representing the current round the simulation actually is. And lastly, an instance of the `Factory` class *factory* which is the factory used for the simulation.
You must add a getter **NbRound** for *nbRound*, **Round** for *round*. And also a getter **Money**, of type *long*, which should returns the remaining money of the attribute *factory*.

```
1  private int nbRound;
2  private int round;
3  private readonly Factory factory;
4
5  // TODO
6  // Add getter NbRound
7  // Add getter Round
8  // Add getter Money
```

Constructor : you must implement the **Game** class' constructor which instanciates all of its attributes. It takes an integer as parameters, representing the maximum number of round for the simulation, and a long which is the initial money of the *factory*. The round is the same as the level attribute for Machine, it starts at one, not zero.

```
1  public Game(int nbRound, long initialMoney)
2  {
3      // TODO
4  }
```

A method : *Launch* which allows you to lauch a simulation, it takes as sole parameter a *Bot*. You must implement this method in charge of running a simulation of *nbRounbd*. It must call various *Bot* methods a the right time, and the right number of times. Moreover, it must keep the variable `round` updated. This method returns the final `score` (the remaing money of the factory) made by the `Bot`.

```
1  public long Launch(Bot bot)
2  {
3      // TODO
4  }
```

3.5.2 MyBot

You are now in the last step, excellent! All you have to do is implement your own bot, and you will be able to enter the competion. In order to have the most money in your factory at the end of the simualtion !

To do so, you have at your disposal an abstract class called **Bot** in the file `Bot.cs`, and a class **MyBot** which implements Bot, in the file `MyBot.cs`.
There are only three methods in the `Bot` class:

- Start : the method called right before the simulation starts running.
- Update : the method called every round.
- End : the method called at the end of the simulation (when all the round are finished).

The `MyBot` class implements these methods. Start and Update already contains some basic code. Of course, this code is far from enough to have a good score. You will need to modify them.

```
1  public override void Start(Game game)
2  {
3      game.Build(MachineType.Hat);
4      game.Build(MachineType.Coat);
5      // Feel free to modify this method
6  }
7
8  public override void Update(Game game)
9  {
10     // TODO modfiy it
11     game.UpdateMoneyAll();
12 }
13
14 public override void End(Game game)
15 {
16     // Feel free to modify it
17 }
```

In order to create your own bot, you are free to add other functions, classes (or enumeration and so on) in the file `MyBot.cs`.

An example of algorithm that could help you is the greedy algorithm. This algorithm follows the principle of a local maximum, it will choose the best option at the moment.

Going further

`https://en.wikipedia.org/wiki/Greedy_algorithm`

There is nothing more deceptive,
than an obvious fact.