TP C#15
My BattleShip

Submission

Archive

A template of the project can be found on the intranet. Your final submission must follow this architecture:

```
rendu-tp-firstname.lastname.zip
    |-- rendu-tp-firstname.lastname/
        |-- AUTHORS
        |-- README
        |-- MyBattleShip/
            |-- BattleShip.sln
            |-- BattleShip/
                |-- Map/
                    |-- Cell.cs
                    |-- Coordinate.cs
                    |-- Map.cs
                |-- Display/
                    |-- IDisplayable.cs
                    |-- Displayer.cs
                    |-- Grid.cs
                |-- Ressources/
                    |-- battleship.png
                    |-- boat.png
                |-- Game/
                    |-- SoloGameManager.cs
                    |-- OnlineGameManager.cs
                    |-- Player.cs
                    |-- OnlinePlayer.cs
                    |-- Ship.cs
                    |-- InvalidPositionException.cs
                |-- Program.cs
                |-- everything except bin/ and obj/
```

- Obviously, you should replace *firstname.lastname* by your name.
- Your code must compile and must not contain any warning.
- Please make sure your classes and methods have the correct name.

AUTHORS

This file should contain a star (*), a space and your login.

Here is an example:

```
* firstname.lastname
```

Note that the name of the file is AUTHORS without extension.

README

This file should contain information about your TP: what did you manage to do, where did you have a hard time, **what bonii have you done**...
Note that the name of the file is README without extension.

1 Reminder

The first part of the subject is about some notions covered earlier this year that haven't been perfectly understood by some of you. We encourage you to read them even if you think you have already mastered them, since this is the last TP before your final exam. If you have questions about these notions, please take a look at MSDN, or ask your assistants.

1.1 Enumerations

An enumeration is declared using the **enum** keyword. It is a type which allows you to declare a set of constants of the same type.

```
1 enum Months {
2   January, February, March, April, Other
3 }
```

Each enumeration has an underlying type. By default, this type is **int**, the first field is equal to 0, and the following fields are incremented by 1 each time. To access to these values, you have to cast the field into an **int**. You can modify the type of the enumeration (it must be a numeric type though), and the value of each type like this:

```
1 enum Months : long {
2   January = 1, February, March = 0, April, Other = 12
3 } //Here, February is equal to 2 and April is equal to 1.
```

Having multiple fields with the same value is not recommended, since it will prevent you from comparing them. Comparisons ('==', or **switch** for example) between fields of an enumeration are based on their values, which can cause some conflicts. Fields having the same value cannot appear at the same time in a switch.

```
enum Months : long {
  January = 1, February, March = 0, April, Other = 12
} //Here, February is equal to 2 and April is equal to 1.

public bool foo(Months m)
{
  return m == Months.April;
}

foo(Months.January); //true
```

1.2 Inheritance, abstract class, override, Interface and static

Here is a little reminder about important notions to know about classes.

1.2.1 Inheritance

You can make your classes inherit. It is a basic notion that is extremely important. A class inheriting from another class will inherit the attributes and the **public** and **protected** methods from the parent class. In C#, every class inherits from the **Object** class and has access to some methods (Tostring, GetType, etc...).

```
public class A
{
  protected int x;
}

public class B : A //B inherit from A
{
  //It works, since B inherits the protected attribute x.
  public int GetX() { return x; }
}
```

The constructor of B will always call the constructor of A before executing. Thus, when the constructor of A takes some parameters, you must specify them using the **base** keyword in the constructor of B.

```
public class A
{
  public A(int x) { this.x = x; }
  protected int x;
}

public class B : A
{
  //These two constructors are correct
  public B() : base (0) {}
  public B(int x) : base(x){}
  public int GetX() { return x; }
}
```

The **Object** class has a *ToString* method that is called when the object is used in a string context. Every class inherits from **Object**, so it is possible to "override" this method and add a particular behavior when it is used in a string context. Example :

```
public class A
{
  public override string ToString()
  {
    return "I'm an Object of type A";
  }
}
Console.WriteLine(new A()); //Displays "I'm an object of Type A".
```

1.2.2 Abstract et override

A method defined with the **abstract** keyword is a method that has not been implemented yet. You can see it as a prototype declaration. An abstract method implies that its class is abstract. Thus, by definition, an abstract class cannot be instantiated since every method has not been implemented yet. When a class inherits from an abstract class, it has to "override" every abstract method of the abstract class (except if this class is also abstract).
Reminder : The **override** keyword allows you to re-implement a method of a parent class, abstract or not. The overriding method has the same arguments and the same return type as the overridden one.

```
//A cannot be instantiated
public abstract class A
{
  public void foo() {}
  public abstract void bar();
}

```

```
8
9  //B can be instantiated (after removing what doesn't compile)
10 public class B : A
11 {
12   //Doesn't compile without override
13   public void bar() {}
14
15   //Correct
16   public override void bar() {}
17
18
19   //Doesn't compile
20   public override int bar() { return 14; }
21
22   //Correct, we can define a method having the same name if the number of argument is different
23   public int bar(int x) { return 14; }
24 }
```

1.2.3 Interfaces

An interface is simply an abstract class that has only abstract methods. In C#, a class can inherit from only one class, but is able to implement multiple interfaces. Moreover, an interface cannot contain any attributes, but can contain some properties[1]. It is declared with the **interface** keyword. Usually, its name starts with the letter 'I', and ends with 'able' because it represents something that the object is able to do. (IComparable to be compared with **CompareTo**, IEnumerable to be used in a **foreach**, IDisposable which permits to release the object, etc).

```
1  public interface IDrawable
2  {
3    //the abstract keyword isn't needed.
4    void print();
5  }
6
7  public class Shape : IDrawable
8  {
9    public void print()
10   {
11     Console.WriteLine("This is a shape");
12   }
13 }
```

[1] https://msdn.microsoft.com/fr-fr/library/w86s7x04.aspx

1.2.4 Méthodes statiques

A **static** method is a method that doesn't need an instance of the object to be called. It allows us to access a method of the class without having to instantiate it. It implies that the **static** method never calls a non-static method or attribute. If the methods and the attributes are non-static, they don't exist when we call the **static** method.

```
public class A
{
  public static void print()
  {
    Console.WriteLine("14");
  }
}
A.print(); //Correct
A a = new A();
a.print(); //Correct
```

A **static** attribute works the same way. It is shared between every instance of the class and can be accessed even when the class has never been instantiated. This property is really interesting. For example, it allows you to count the number of instances, or it allows you to share an object with other classes without having to give it to the constructor. Here is a basic code that counts the number of instances of a class.

```
public class A
{
  public static int NbInstances = 0;
  public A()
  {
    NbInstances++;
  }
}
//Displays 0
Console.WriteLine("There is " + A.NbInstances + " object of type A");
A a = new A();
A b = new A();
//Displays 2
Console.WriteLine("There is " + A.NbInstances + " object of type A");
```

A **static** attribute must be initialized in the declaration (or in a static constructor[2]), not in the constructor. It doesn't belong to an instance and exists even when the constructor has never been called.

[2] https://msdn.microsoft.com/fr-fr/library/k9x6w0hc.aspx

1.3 Exceptions

Almost every language gives you the opportunity to handle exceptions. In C#, exceptions are handled with the **try**, **catch** and **finally** keywords. An exception can be raised using **throw**. When an exception is raised in a method, it will stop its execution, and will go up in every calling method until it is handled. Here is a example to show this behavior.

```
1  public bool foo()
2  {
3    bool b = true;
4    throw new IndexOutOfRangeException(); //Do not forget the new keyword
5    return b; //It will never reach this code.
6  }
7
8  public void bar()
9  {
10   try
11   {
12     foo();
13     return true; //It will never reach this code.
14   }
15   catch (OutOfBoundException e)
16   {
17     return false; //It will never reach this code.
18   }
19   catch (IndexOutOfRangeException e)
20   {
21     Console.WriteLine(e.msg);
22     return false;
23   }
24   finally //The finally block is optional
25   {
26     Console.WriteLine("Finally block"); //Code executed, even after a return.
27   }
28 }
```

If an exception is raised in the **try**, and isn't handled with **catch**, then the **finally** block will not be executed. Thus, if you never handle an exception, it will arrive in your *main* method and the program will stop. You can easily create your own exceptions by creating a class inheriting from **Exception**. You can then add your own methods to your exceptions.

Maybe your think that **throw** an exception is useless since you can just **return** to leave the function. Imagine your method returns an **int[][]**. You would create a 2-dimension array to return, just to handle a particular error case, it would be quite inconvenient. It is better to **throw** an exception.

1.4 Network

Network is handled by something called a 'Socket'. A socket is an communication interface between processes. We strongly encourage you to read again the 'TP C# 11' which explains really well what a socket is, an ip, the ways of transmission, and of course the documentation on MSDN of some classes: **Socket**[3], **IPAddress** and **IpEndPoint**. When you connect to another computer, you will need at least these 4 methods.

```
Socket socket = new Socket (AddressFamily.InterNetwork, SocketType.Stream,
                            ProtocolType.Tcp);

//Client side
void Connect (IPAddress address, int port);

//Server side
void Bind (IPEndPoint endPoint);
void Listen (int maxWaitList);
Socket Accept();
```

When your socket has been created:

- Client side, you need to *Connect* your socket to connect to a server.
- Server side, you need to *Bind* the socket to something called **endpoint**. After binding, you need to *Listen*, which will allow the endpoint to add the incoming connexions to a queue. Then you need to *Accept*. This is a blocking method, which means that if the queue of incoming connexions is empty, it will stop the execution of the current method until a connexion income. It will then *pop* a connexion out of the queue, and return the **Socket** which allows the communication with this connexion.

Once you have the right **Socket** (after *Acccept*, or after connecting it with *Connect*), the *Send* and *Receive* methods are used. They take byte arrays as parameters. Thus, conversion methods can be useful.

```
byte[] bt = System.Text.Encoding.UTF8.GetBytes("Martin, un ACDC en platine");
string str = System.Text.Encoding.UTF8.GetString(bt);
```

The *Available* attribute of a **Socket** shows the quantity of information received by the socket (not yet retrieved using *Receive*). A **Socket** has some interesting methods, such as *Connected*. Moreover, do not forget that the *TryParse* method doesn't only apply to **int**.

[3] https://msdn.microsoft.com/fr-fr/library/system.net.sockets.socket(v=vs.110).aspx

2 Exercise : MyBattleShip

The goal of this practical is to implement the 'BattleShip' game. It will be divided into two parts: The game implementation, used by the given GUI to display the grids, and the network part that will allow you to battle against someone. Every attribute **MUST** be 'private'. *Getters* and *setters* are not mandatory if they are not needed. You may add your own private attributes and methods.

2.1 Rules

Battleship is a game where the two players have to place ships on a 10x10 hidden grid, and has to sink the opponent's ships. Players alternate turn, and shoot one cell each turn. The winner is the first player to sink all his opponents ship. Thus, each player has two grids: A **hidden grid** which contains his ships and the opponent's shots (in order to say when a ship is missed, hit, or sunk), and a **public grid** which contains only his shot (in order not to shoot twice at the same place, and to choose where to shoot). Each player has to put 5 ships on his hidden grid, one of each type. Types will be described later in the subject.

2.2 Game implementation

Your submission is organized in four folders as shown in the submission architecture. We will give you a GUI which will be contained in the Display/ folder, and which will use pictures placed in Ressources/. A beginning of the project is given on the intranet. Classes must follow the submission architecture.

2.2.1 Coordinate

This class allows the user to manipulate a point more easily on a map. It has two attributes : x and y. Coordinates are fixed, so *setters* are not needed. The cell at coordinates (0,0) corresponds to the upper left corner. (9,0) is the upper right corner.

```
1 public Coordinate(int x, int y);
2 public int GetX();
3 public int GetY();
```

2.2.2 Cell

As said previously, each player has two grids. We have decided to merge these two grid into one object.
To achieve this, a **Cell** has two states depending on an enumeration.

- **hState** represents a cell of your hidden grid (grid which contains your ship and the opponent's shots).
- **pState** represents a cell of your public grid (grid which contains your shots, but not the opponent's ships).

These states have four possibilities represented by the **State** enumeration.

- **WATER**: empty cell (default value)
- **BOAT**: cell containing a ship
- **HIT**: cell showing that a boat has been hit by you (*pState*) or your opponent (*hState*)
- **MISSED**: cell not containing a boat that received a shot by you (*pState*) or your opponent (*hState*)

Thus, this class has *hState*, *pState* and coordinates.

```
1 public Cell(Coordinate coord);
2 public State GetPstate();
3 public State GetHstate();
4 public void SetHstate(State s);
5 public void SetPstate(State s);
6 public Coordinate GetCoordinate();
```

2.2.3 Ship

The *type* of the ship depends on the **ShipType** enumeration.

- **AIRCRAFT** of size 5
- **BATTLESHIP** of size 4
- **SUBMARINE** of size 3
- **DESTROYER** of size 3
- **PATROLBOAT** of size 2

Ship will thus contain a *type*, a boolean *horizontal*, a boolean *sunk* and a list of coordinates.

```
1 public Ship(ShipType type, bool horizontal, Coordinate origin)
2 public static int GetSize(ShipType type);
3 public bool IsHorizontal();
4 public bool IsSunk();
5 public void SetSunk(bool b);
6 public bool IsAtCoordinate(Coordinate coord);
7 public ShipType GetShipType();
8 public List<Coordinate> GetCoordinates();
```

The constructor will create a list of coordinates according to the boolean *horizontal* and the *origin* which represents the upper left point of the ship.

2.2.4 Map

The class **Map** will be used to contain the matrix of **Cell**s, and the player's ships. Thus, it will contain a list of **Ship**s and the matrix.

```
public Cell GetCell(Coordinate coord);
public List<Ship> GetShips();
public Cell[][] GetMatrix();
public bool AddShip(Ship ship);
```

The *AddShip* method will verify if it's possible to insert the given ship into the grid (hState, if you understood well), and insert it if possible. Do not forget to update the cells' state.

2.2.5 InvalidPositionException

As specified earlier, you may create your own exceptions. The constructor will take a **Coordinate**. The error message and its *getter* is free.

2.2.6 IDisplayable

Here, you have to create an interface that will be implemented by the **Player** class. It has to contain at least the following methods.

```
public string GetName();
public Map GetMap();
```

2.2.7 Player

Now that every basic class is implemented, we can implement the class that manipulates the map : **Player**. It will contain a name, a boolean *display* which will be used to enable the GUI, and a **Map**.

```
public Player (string name, bool display);
public Player(string name); //display set to false
public bool IsDisplay();
public Coordinate Shoot();
public bool ReceiveShot(Coordinate coord);
public bool HasLost(); //Tous les bateaux ont coulé.
public void SetCell(Coordinate coord, bool success);
public void CheckPosition(Ship sip);
public void GenerateShip(ShipType ship);
public bool AddShip(Ship ship);
```

- *Shoot* is an AI that chooses where to shoot. You can easily implement a **Random** AI through the whole grid, but it won't really be effective.
- *ReceiveShot* is called when you receive a shot. It will update the cell, and return if the shot has hit a ship. It will also update *sunk*.
- *SetCell* is called after receiving the result of the previous shot. It will update the cell according to *success* (true if it's a hit)
- *CheckPosition* verifies that every coordinate of the ship is valid (in the grid, and right cell state). It will **throw** an InvalidPositionException if it is not.
- *AddShip* takes a ship as argument, verifies its position, and add it if it is correct. (It has to call *CheckPosition*).
- *GenerateShip* is creating a ship on the map. Again, you can use the class **Random**. It has to call *AddShip*.

2.2.8 SoloGameManager

Now that everything is ready, we can implement the running class of our game. For the moment, we will just have two AI play against each other. This class will contain the two players, and a list of **Displayer**, a class which enables the GUI. The GUI is extremely simple to use. The constructor takes the **Player** to display. The *Create* method initializes it and *UpdateGui* updates it.

```
1 public SoloGameManager(Player player1, Player player2);
2 public void Play();
```

The constructor initializes players and displayers. (Do not forget to take into account the *Display* boolean) The *Play* method will represent the game. If you understood everything, until the game is over, it has to alternate *Shoot*, *ReceiveShot* and *SetCell*. Do not forget to call *UpdateGui* of each displayer.

2.3 PrettyPrinter

For this part, you have to add the following methods to your **Map** class.

```
1 public override string ToString();
2 public void PrettyPrint(string map);
```

The string returned by *ToString* should follow this format:

```
+ - - - - - - - - - - +  + - - - - - - - - - - +
| S   P P       O     |  |     X X X X X       |
| S                   |  |                     |
| S         O         |  |                     |
|                   O |  |                     |
| A A X X A           |  |    O   O            |
|                     |  |                     |
| B B B B             |  | O                   |
|   D         O       |  |                   O |
|   D                 |  | O                   |
|   D                 |  |                     |
+ - - - - - - - - - - +  + - - - - - - - - - - +
```

Each cell is separated by a space. The string must not finish by a newline. Cells are represented this way:

- A : aircraft
- B : battleship
- S : submarine
- D : destroyer
- P : patrolboat
- O : missed
- X : hit
- : water

PrettyPrint must simply display the string using colors. You are free to chose the colors used for this (not too much pink please). Thanks to this *PrettyPrinter*, you can easily reproduce the behavior of the GU by calling *PrettyPrint(player.GetMap())*instead of d'*UpdateGui()*.

2.4 Network

To be able to play with another player, you have to add some methods to your game. We all know that at EPITA, everyone is very honest. We will consider that the grid of the opponent will be of size 10x10, and that the five ships are placed and well positioned. (A ship on the 11th column of the grid would be annoying and really mean, but we are at EPITA, country of fairness, so it's okay)

2.4.1 OnlinePlayer

The **OnlinePlayer** class inherits from **Player**. As we said in the reminders, *ReceiveShot* can not be overridden because it has the same arguments, but not the same return type.

```
public override Coordinate Shoot();
public int ReceiveShotVersus(Coordinate coord);
public void setCell(Coordinate coord, int success);
public bool HasWon();
```

Shoot will read the user input as long as the input is not correct. The requested format is "x y". You don't know the opponent's grid, so you cannot use *HasLost* anymore. You also need to receive the information that a ship has been sunk (not only missed/int like before). *ReceiveShotVersus* must return 0, 1 or 2, if the shot has missed, hit, or sunk a ship. *setCell* will now take this integer as an argument, instead of a boolean.
Ship positioning is still random (see bonii's list).

2.4.2 OnlineGameManager

You need to implement at least the following methods.

```
1 public OnlineGameManager(Player player, string port, string ip = null);
2 public Socket Connect(string ip, string port, bool first);
3 public void Play();
```

In the new *Play* method, you need to shoot, wait for the result, wait for the opponent's new shot, and send the result. An idea would be to have a boolean that indicates if it's your turn or not. For each turn, each player has to send 3 integers (result of the shot, x and y of your next shot). Useless integers will be ignored, but there must be 3 integers sent, even when it's the first or last shot.
Note: These integers are lower than 128, and can easily be converted to characters. You should then send a string of length 3.

The first player will be the server, and the second player will connect to the first one. The first player is thus defined by *ip*, which will be **null** when it is the first player.

Bonii

The first bonus could be about the ship placement. You have to make your program read a file in which each line would contain the coordinates, the type of the ship, and the boolean *horizontal* separated by a space, in order to call the **Ship**'s constructor. You need to check the correctness of the file, and make sure that the ships are well positioned.

```
1 public void GenerateShipFile(string path);
```

Other bonii are free (**A good idea would be to study every other practical to have a good grade at your final exams**).

The code is the law.