TP C#15
My BattleShip

Submission

Archive

Un début de projet vous est fourni sur l'intranet. Votre rendu final doit suivre l'architecture suivante:

```
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
                |-- tout le reste sauf bin/ and obj/
```

- Bien entendu, vous devez remplacer *prenom.nom* par votre nom.
- Votre code doit compiler, et ne doit contenir aucun warning.
- Faites très attention aux noms de vos classes et méthodes afin que le GUI puisse fontionner correctement.

AUTHORS

Ce fichier doit contenir Une astérisque (*), un espace et votre login.
Voici un exemple:

```
* prenom.nom
```

Notez bien que le nom du fichier est AUTHORS SANS extension.

README

Ce fichier doit contenir au moins les informations suivantes : Où en êtes-vous dans l'avancée du tp, qu'est-ce que vous n'avez pas fait, **quels bonii avez-vous fait**...
Notez bien que le nom du fichier est README SANS extension.

1 Rappels

Voici quelques notions déjà vues plus tôt dans l'année qui restent floues pour beaucoup d'entre vous. Nous vous incitons à les lire même si vous les maitrisez déjà car ceci est votre dernier tp avant les partiels. Pour toute information supplémentaire, veuillez regarder MSDN, ou demander à vos assistants.

1.1 Enumérations

Une énumération est déclarée en utilisant le mot clé **enum**. Une énumération est un type permettant de déclarer un ensemble de constantes de même type.

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

Chaque énumération a un type sous-jacent. Par défaut, le type sous-jacent est **int**, le premier champ a pour valeur 0, les champs suivants sont incrémentés de 1 à chaque fois. Pour accéder à la valeur d'un champ, il suffit juste de caster ce champ en **int**. Vous pouvez modifier le type de l'énumération (tant que c'est un type numérique) et la valeur de chaque champ comme ci-dessous.

```
1 enum Months : long {
2   January = 1, February, March = 0, April, Other = 12
3 } //Ici, February est égal à 2, et April est égal à 1
```

Avoir plusieurs champs ayant la même valeur est fortement déconseillé, car cela va vous empêcher de comparer ces deux champs. Les comparaisons ('==', ou les **switch** par exemple) entre les champs d'une énumération se basent sur la valeur des champs comparés, ce qui crée des conflits. Les champs de valeurs égales ne peuvent pas apparaître en même temps dans un switch.

```
enum Months : long {
  January = 1, February, March = 0, April, Other = 12
} //Ici, February est égal à 2, et April est égal à 1

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

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

1.2 Héritage, classe abstraite, override, Interface et static

Voici un petit rappel de certaines notions importantes à connaître concernant les classes.

1.2.1 Héritage

Vous pouvez faire hériter vos classes. C'est une notion basique extrêmement importante de la programmation orientée objet. Une classe héritant d'une autre classe possèdera tous les attributs et les méthodes **public** et **protected** de la classe dite "mère". En C#, toute classe hérite de la classe **Object** et a donc accès à certaines méthodes (Tostring, GetType, etc...)

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

public class B : A //B hérite de A
{
  //Ca marche, car B hérite de l'attribut protected x
  public int GetX() { return x; }
}
```

Le constructeur de B appellera toujours le constructeur de A avant de s'exécuter. Du coup, quand le constructeur de A prend des paramètres, vous êtes obligé de spécifier ces paramètres avec le mot clé **base** dans le constructeur de B.

```
1  public class A
2  {
3    public A(int x) { this.x = x; }
4    protected int x;
5  }
6
7  public class B : A
8  {
9    //Ces deux constructeurs sont corrects
10   public B() : base (0) {}
11   public B(int x) : base(x){}
12   public int GetX() { return x; }
13 }
```

La classe **Object** possède une méthode *ToString* qui est appelée lorsque l'objet est utilisé dans le même contexte qu'un string. Du coup, sachant que toutes les classes héritent implicitement de **Object**, il est possible de "override" cette méthode et donc de changer son comportement. Example :

```
1  public class A
2  {
3    public override string ToString()
4    {
5      return "I'm an Object of type A";
6    }
7  }
8  Console.WriteLine(new A()); //Affiche "I'm an object of Type A".
```

1.2.2 abstract et override

Une méthode définie avec le mot clé **abstract** est une méthode qui n'a pas encore été implémentée. Vous pouvez voir cette méthode comme une déclaration de prototype. Une méthode abstraite implique que la classe soit abstraite. Par définition une classe abstraite ne peut pas être instanciée, vu que toutes ses méthodes ne sont pas implémentées. Lorsqu'une classe va hériter d'une classe abstraite, elle va devoir "override" toutes les méthodes abstraites de la classe "mère" (A moins d'être elle aussi abstraite).
Rappel : Le mot clé **override** permet de réimplémenter une méthode d'une classe mère, abstraite ou pas.

```
1  public abstract class A //A ne peut pas être instanciée
2  {
3    public void foo() {}
4    public abstract void bar();
5  }
```

```
6  //B peut être instanciée (En enlevant les cas ne compilant pas)
7  public class B : A
8  {
9    //Ne compile pas sans override
10   public void bar() {}
11
12   //Correct
13   public override void bar() {}
14
15   //Ne compile pas.
16   public override int bar() { return 14; }
17
18   //Correct, on a le droit de définir une méthode du même nom si le nombre
19   //d'argument est différent.
20   public int bar(int x) { return 14; }
21 }
```

1.2.3 Interfaces

Une interface est simplement une classe abstraite dont toutes les méthodes sont abstraites. En C#, une classe peut hériter que d'une seule classe, mais peut implémenter plusieurs interfaces. De plus, une interface ne peut pas contenir d'attributs, mais peut contenir des propriétés[1]. Elle est déclarée avec le mot clé **interface**. Habituellement, son nom commence son nom par la lettre 'I', et fini par 'able' car elle représente la plupart du temps le fait que l'objet puisse faire quelque chose (IComparable pour être comparé avec **CompareTo**, IEnumerable pour pouvoir être utilisé avec **foreach**, IDisposable qui permet de libérer un objet qui ne peut pas être libéré automatiquement, etc..).

```
1  public interface IDrawable
2  {
3    //Le mot clé abstract n'est pas précisé car c'est implicite.
4    //Il ne faut pas préciser la visibilité des méthodes dans les interfaces.
5    void print();
6  }
7
8  public class Shape : IDrawable
9  {
10   public void print()
11   {
12     Console.WriteLine("This is a shape");
13   }
14 }
```

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

1.2.4 Méthodes statiques

Une méthode **static** est une méthode qui n'a pas besoin que la classe soit instanciée pour pouvoir l'appeler. Cela permet d'avoir accès à une méthode d'une classe sans vraiment avoir besoin d'une instance de celle-ci. Cela implique que la méthode **static** ne fait jamais appel à des méthodes ou attributs non **static**. En effet, si les méthodes et attributs sont propres à un objet, ils n'existent pas au moment où l'on appelle la méthode **static**.

```
1  public class A
2  {
3    public static void print()
4    {
5      Console.WriteLine("14");
6    }
7  }
8  A.print(); //Correct
9  A a = new A();
10 a.print(); //Correct
```

Un attribut **static** fontionne de la même manière. Il est partagé entre toutes les instances et peut être accéder même lorsque la classe n'a jamais été instanciée. C'est une propriété extrêmement importante : Il vous permet entre autre de compter les instances d'une classe, ou de partager un objet à d'autres classes sans avoir à le passer au constructeur à chaque fois, ce qui évite du code redondant, et des copies inutiles de cet objet. Voici une manière toute bête de compter les instances d'une classe.

```
1  public class A
2  {
3    public static int NbInstances = 0;
4    public A()
5    {
6      NbInstances++;
7    }
8  }
9  //Affiche 0
10 Console.WriteLine("There is " + A.NbInstances + " object of type A");
11 A a = new A();
12 A b = new A();
13 //Affiche 2
14 Console.WriteLine("There is " + A.NbInstances + " object of type A");
```

Un attribut **static** doit toujours être initialisé en même temps que la déclaration (ou dans un constructeur statique[2]) et pas dans le constructeur pour les mêmes raisons que les méthodes. Il n'est pas propre à une instance et existe même quand le constructeur n'a jamais été appelé.

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

1.3 Exceptions

Pratiquement tous les langages vous offrent une gestion des exceptions, qui permettent comme le nom l'indique, de gérer des cas exceptionnels et non attendus. En C#, les exceptions sont gérés par les mots clés **try**, **catch** et **finally**. Une exception peut être levée en utilisant **throw**. Comme vous le savez, chaque méthode a son propre fil d'exécution. Lorsqu'une exception est levée, elle va arrêter le fil d'exécution de la méthode courrante, et va remonter dans toutes les méthodes "mères" tant que l'exception en question n'est pas gérée. Voici un exemple pour que cela soit plus clair.

```
1  public bool foo()
2  {
3    bool b = true;
4    throw new IndexOutOfRangeException(); //N'oubliez pas le mot clé new.
5    return b; //Ce code ne sera jamais executé.
6  }
7
8  public void bar()
9  {
10   try
11   {
12     foo();
13     return true; //Ce code ne sera jamais executé
14   }
15   catch (OutOfBoundException e)
16   {
17     return false; //Ce code ne sera jamais executé
18   }
19   catch (IndexOutOfRangeException e)
20   {
21     Console.WriteLine(e.msg);
22     return false;
23   }
24   finally //Le block finally est optionnel
25   {
26     Console.WriteLine("Finally block"); //Code executé, même après un return
27   }
28 }
```

Si une exception est levée dans le **try**, et n'est pas **catch**, alors le block **finally** ne s'exécutera pas. Du coup, vous l'aurez compris, si vous ne gérez jamais une exception, et qu'elle remonte jusqu'à la méthode *main*, le programme va s'interrompre. Vous pouvez facilement créer vos propres exceptions en créant une classe qui hérite de la classe **Exception**. Vous pouvez donc rajouter vos propres méthodes à vos exceptions.

Vous pensez peut-être que c'est inutile car "**throw** une exception" peut être remplacée par un **return**, mais imaginez que votre méthode renvoie un **int[][][]**. Cela voudrait dire que pour gérer un cas particulier arrivant au milieu de cette méthode, vous devriez créer ce tableau juste pour pouvoir le renvoyer. Ce n'est pas du tout pratique, d'où l'utilisation d'exceptions.

1.4 Réseau

Le réseau est géré par ce qu'on appelle les 'Socket'. Une socket est une interface de communication entre processus. Nous vous invitons très fortement à relire le TP C# 11 qui explique très bien le concept de Socket, d'ip, et de moyen de transmission, ainsi que la documentation MSDN des classes **Socket**[3], **IPAddress** et **IpEndPoint**. Lorsque vous voulez vous connecter à un autre appareil, quatre méthodes sont essentielles.

```
1  Socket socket = new Socket (AddressFamily.InterNetwork, SocketType.Stream,
2                              ProtocolType.Tcp);
3
4  //Côté client
5  void Connect (IPAddress address, int port);
6
7  //Côté serveur
8  void Bind (IPEndPoint endPoint);
9  void Listen (int maxWaitList);
10 Socket Accept();
```

Une fois que votre Socket est créée :

- Côté client, vous devez appeler la méthode *Connect* pour vous connecter à un serveur.
- Côté serveur, il faut relier la **Socket** à ce qu'on appelle un **endpoint** grace à la méthode *Bind*. Après avoir lié correctement, il faut *Listen*, ce qui va permettre d'écouter l'endpoint et de rajouter dans une file les connexions entrantes. Enfin, il faut *Accept*. Cette méthode est bloquante, c'est à dire qu'elle va arrêter le fil d'exécution tant que la file de connexion entrante est vide. Elle va ensuite *pop* une connexion, et renvoyer la **Socket** qui permet la communication avec cette connexion.

Une fois la **Socket** connectée à un client obtenue (avec *Accept*, ou en l'ayant créée et en ayant appelé *Connect*), les méthodes *Send* et *Receive* vous servent à envoyer ou recevoir des informations. Elles prennent en argument des tableaux de byte. Les méthodes de conversion peuvent vous servir.

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

L'attribut Available d'une **Socket** indique la quantité de données reçues (et qui n'ont pas été récupérées avec *Receive*) et peut éventuellement vous aider. Une **Socket** possède plusieurs méthodes intéressantes, comme *Connected* par exemple. De plus, n'oubliez pas que la méthode *TryParse* n'est pas seulement utilisée pour les **int**.

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

2 Exercice : MyBattleShip

Le but de ce tp sera d'implémenter le jeu 'Bataille Navale'. Il sera divisé en deux parties : L'implémentation du jeu, dont le GUI fourni par les ACDC va se servir pour afficher les grilles, et une partie réseau qui va permettre de jouer au jeu contre un autre joueur. Dans ce tp, tout attribut **DOIT** être 'private'. Les *getters/setters* ne sont pas obligatoires s'ils ne sont pas nécessaires. Rajouter des attributs privés est autorisé, au même titre que de rajouter des méthodes (toujours privées) si cela peut vous simplifier la vie.

2.1 Règles

La bataille navale, appelée aussi touché-coulé, est un jeu de société dans lequel deux joueurs doivent placer des bateaux sur une grille 10x10 tenue secrète et tenter de toucher les navires adverses en tirant chacun son tour. Le gagnant est celui qui parvient à couler tous les bateaux adverses avant l'autre. Vous l'aurez donc compris, chaque joueur possède deux grilles. Une **grille privée**, qui contient ses navires, et les tirs effectués par l'adversaire, pour pouvoir dire lorsque l'adversaire a raté, touché, ou coulé une cible, et une **grille publique** qui contient ses tirs, pour ne pas tirer deux fois au même endroit, et pour décider où effectuer son prochain tir. Chaque joueur doit positionner sur sa grille 5 bateaux, un de chaque type. Les types sont décrits plus bas dans le sujet.

2.2 Implémentation du jeu

Votre rendu doit s'organiser en 4 dossiers comme précisés dans l'architecture de rendu. Nous vous fournissons un GUI qui sera contenu dans le dossier Display/, et qui se servira des images placées dans le dossier Ressources/. Un début de projet est donné sur l'intranet. Les classes devront absolument suivre l'architecture précisée en haut du sujet.

2.2.1 Coordinate

Cette classe permet de représenter plus facilement un point sur une carte. Elle possède deux attributs : x et y. Les coordonées sont fixes, il n'y a donc pas besoin de *setters*. La case de coordonnées (0,0) correspond au coin en haut à gauche de la grille. (9,0) est en haut à droite de la grille.

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

2.2.2 Cell

Comme dit précédemment, chaque joueur possède deux grilles. Nous avons décidé de fusionner les deux grilles dans un seul objet.
Pour arriver à ça, une **Cell** possède deux états qui dépendent d'une énumération.

- **hState** qui représente une case de votre grille 'hidden' (grille qui contient vos bateau, et les tirs de l'adversaire).
- **pState** qui représente une case de votre grille 'public' (grille qui contient vos tirs, mais pas les bateaux de l'adversaire)

Ces états ont quatres possibilités représentés par l'énumération **State**.

- **WATER**: pour une case vide (valeur par défaut)
- **BOAT**: pour une case représentant un bateau
- **HIT**: pour une case qui possédait un bateau, mais vous (*pState*) ou votre adversaire (*hState*) l'a touché.
- **MISSED**: vous (*pState*) ou votre adversaire (*hState*) a tiré dans une case **WATER**

La classe possède donc comme vous l'avez compris *hState*, *pState* et des coordonnées.

```
public Cell(Coordinate coord);
public State GetPstate();
public State GetHstate();
public void SetHstate(State s);
public void SetPstate(State s);
public Coordinate GetCoordinate();
```

2.2.3 Ship

Le *type* du bateau dépends de l'énumération **ShipType**

- **AIRCRAFT**: de taille 5
- **BATTLESHIP**: de taille 4
- **SUBMARINE**: de taille 3
- **DESTROYER**: de taille 3
- **PATROLBOAT**: de taille 2

La classe contiendra donc un *type*, un booléen *horizontal*, un booléen *sunk* et une liste de coordonnées.

```
public Ship(ShipType type, bool horizontal, Coordinate origin)
public static int GetSize(ShipType type);
public bool IsHorizontal();
public bool IsSunk();
public void SetSunk(bool b);
public bool IsAtCoordinate(Coordinate coord);
public ShipType GetShipType();
public List<Coordinate> GetCoordinates();
```

Le constructeur va créer une liste de coordonnées en fonction du booléen *horizontal* et de l'origine qui représente le point le plus en haut à gauche du bateau.

2.2.4 Map

La classe **Map** va servir à contenir la matrice de **Cell**, et les bateaux du joueurs. Elle contiendra une liste de **Ship** et la matrice en question.

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

La méthode *AddShip* va vérifier s'il est possible d'insérer le bateau passé en argument dans la grille (hState, si vous avez bien suivi), et l'insérer si possible. N'oubliez pas d'update l'état des cases.

2.2.5 InvalidPositionException

Comme précisé au début du sujet, créez votre propre exception. Le constructeur prendra une **Coordinate**. Le message d'erreur et son *getter* est laissé libre.

2.2.6 IDisplayable

Vous devez ici créer une interface que devra implémenter la classe **Player**. Elle doit contenir au moins les méthodes suivantes.

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

2.2.7 Player

Maintenant que les classes de bases sont implémentées, nous allons implémenter la classe manipulant la grille : **Player**. Elle va donc contenir un nom, un booléen *display* qui servira à activer le gui, et une **Map**.

```
1  public Player (string name, bool display);
2  public Player(string name); //display set to false
3  public bool IsDisplay();
4  public Coordinate Shoot();
5  public bool ReceiveShot(Coordinate coord);
6  public bool HasLost(); //Tous les bateaux ont coulé.
7  public void SetCell(Coordinate coord, bool success);
8  public void CheckPosition(Ship sip);
9  public void GenerateShip(ShipType ship);
10 public bool AddShip(Ship ship);
```

- *Shoot* est une IA qui choisi où tirer. Vous pouvez facilement faire une AI à base de **Random** sur toute la grille, mais cela ne sera vraiment pas efficace.
- *ReceiveShot* est appelée quand vous recevrez un tir. Cela update la case en question, et renvoie si le tir a touché un bateau. Elle mettra aussi à jour *sunk*.
- *SetCell* est appelée après avoir reçu le résultat du tir, update la case en fonction du booléen *success* (true si le tir a touché un bateau)
- *CheckPosition* vérifie que toutes les coordonnées du bateaux sont valides. La méthode va **throw** une InvalidPositionException.
- *AddShip* prend un bateau, vérifie sa position et l'ajoute si elle est correcte (Elle doit appeler *CheckPosition*).
- *GenerateShip* génère un bateau sur la map. Ici encore, vous pouvez utiliser la class **Random**. Cette méthode doit appeler *AddShip*.

2.2.8 SoloGameManager

Maintenant que tout est prêt, nous pouvons implémenter la classe qui va faire tourner notre jeu. Pour l'instant nous allons juste faire jouer deux IA l'une contre l'autre. Cette classe va contenir les deux joueurs, et une liste de **Displayer**, classe qui permet d'activer le GUI. Le GUI est extremement simple à utiliser. Le constructeur prend en paramètre un **Player**. La méthode *Create* l'initialise, et la méthode *UpdateGui* l'actualise.

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

Le constructeur initialize les joueurs et les displayer (N'oubliez pas de prendre en compte le booléen *Display*). La méthode *Play* va représenter la partie. Si vous avez bien suivi, tant que les joueurs n'ont pas perdu, vous devez alterner *Shoot*, *ReceiveShot* et *SetCell*. N'oubliez pas d'appeler les méthodes *UpdateGui* de chaque displayer.

2.3 PrettyPrinter

Pour cette partie du sujet, vous devez rajouter les méthodes suivantes à la classe **Map**.

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

Le string renvoyé par la méthode *ToString* devra suivre le format suivant :

```
+ - - - - - - - - - - +  + - - - - - - - - - - +
| 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                 |  |                     |
+ - - - - - - - - - - +  + - - - - - - - - - - +
```

Chaque case est séparée par un espace. Le string ne doit pas finir par un retour à la ligne. Les cases sont représentées suivant ce code :

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

La méthode *PrettyPrint* devra afficher le string en couleur. Le choix des couleurs est laissé libre. Grâce à ce *PrettyPrinter*, vous pouvez facilement reproduire le comportement du gui. Il vous suffit d'appeler *PrettyPrint(player.GetMap()* à la place d'*UpdateGui()*.

2.4 Réseau

Pour pouvoir jouer en réseau avec un autre joueur, vous devez rajouter quelques méthodes de votre jeu. Comme nous savons qu'à epita tous les joueurs sont très honnêtes, nous allons considérer que la grille de l'adversaire est bien de taille 10x10, que les 5 bateaux différents sont présents et qu'ils sont correctement placés. (Un bateau dans la 11ème colonne de la grille serait gênant, mais ce n'est pas le genre des épitéens.....)

2.4.1 OnlinePlayer

La classe **OnlinePlayer** hérite de la classe **Player**. Comme vu dans le cours, *ReceiveShot* ne peut pas être **override** car elle possède les même arguments et un type de retour différent.

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

La méthode *Shoot* va lire ce qu'écrit le joueur tant que l'input est mal formé, ou incorrect. Le format est le suivant : "x y". Ne connaissant pas la grille de l'adversaire, nous ne pouvons pas utiliser la méthode *HasLost*, et nous devons recevoir l'information de l'adversaire si un tir a coulé un bateau ou non. L'implémentation de *HasWon* est libre. *ReceiveShotVersus* devra renvoyer 0, 1 ou 2, en fonction de si le tir a raté, touché, ou coulé un bateau. *setCell* agira donc en fonction de cet entier, et non plus en fonction d'un booléen *success*. Le positionnement des bateaux est toujours aléatoire (cf. liste des bonii).

2.4.2 OnlineGameManager

Vous devez au moins implémenter les méthodes suivantes. Libre à vous d'implémenter d'autres méthodes qui gèreront la connexion avec l'autre joueur, ou de faire ca dans la méthode *PlayVersus*.

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

Contrairement à l'ancienne méthode *Play*, vous devez tirer, attendre le résultat, attendre le tir, puis envoyer le résultat. Une idée serait que dans la boucle de jeu, un booléen indique si c'est à votre tour de jouer. Ainsi, à chaque tour, chaque joueur doit envoyer 3 entiers (résultat du tir, coordonnées x et y du prochain tir). Que ce soit le premier tir ou que le tir reçu ait gagné la partie, il doit toujours y avoir 3 entiers. Les entiers inutiles seront juste ignorés.
Note: Les entiers sont inférieurs à 128, ils peuvent être facilement convertis en charactères. L'envoi sera donc un string de taille 3.

Le premier joueur va jouer le rôle du serveur, et le deuxième joueur va se connecter au premier joueur. Le premier joueur est donc défini par le fait que *ip* soit **null**.

Bonii

Le premier bonus serait que le positionnement des bateaux ne soit pas aléatoire, mais à lire dans un fichier. Chaque ligne contiendrait les coordonnées, le type du bateau et le booléen *horizontal*, séparés par un espace, pour pouvoir appeler le constructeur de la classe **Ship**. Vous devrez vérifier que le fichier peut etre ouvert/lu, qu'il est au bon format et que les bateaux sont bien placés.

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

Les autres bonii sont libres (**Une idée serait de bien réviser tous les tp du second semestre pour gérer vos partiels**).

The code is the law.