Compléments pour le cours 5

En complément du TP5, voici d'autres exercices sur des thèmes abordés lors des 5 premiers cours, que je vous conseille de faire afin de vous entrainer davantage.

EXERCICE 04_01

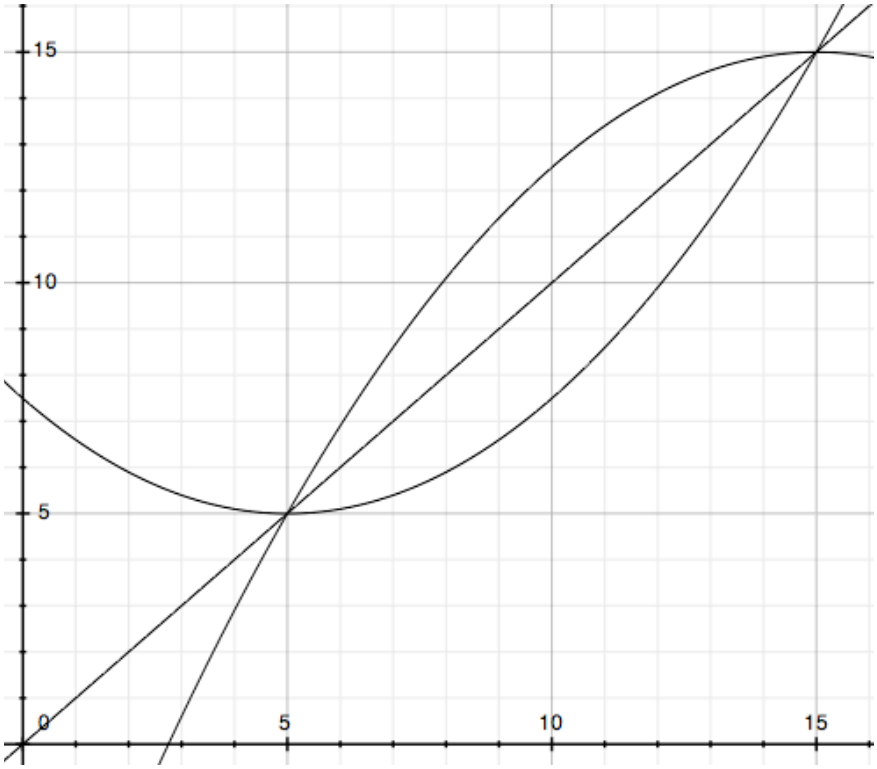
Interfaces
Strategy

Comme préambule aux exercices suivants, réalisez un programme en respectant le diagramme de classes suivant.

Un `Elève` possède un `Nom`, une note `Min`, une note `Max`. On a également un tableau d'`Elèves` prédéfinis avec au moins 3 `Elèves` dont les notes `Min` et `Max` sont différentes.

Une classe implémentant `IRépartition` devra comporter une méthode `Image`. Cette méthode prendra en paramètres un entier `min`, un entier `max` et un entier `nombre` et devra rendre l'image de ce nombre selon une équation associée à cette classe. Cette équation pourra (ou non) se baser sur les paramètres `min` et `max`. Ainsi :

- la classe `RépartitionLinéaire` rend l'image de x sur la droite y=x, i.e. rend le nombre directement, sans modifications,
- la classe `RépartitionCarrée` rend l'image de x sur la parabole $y=\frac{(x\text{-min})^2}{\text{max-min}}+\text{min}$, i.e. un nombre revu à la baisse,
- la classe `RépartitionCarréeNégative` rend l'image de x sur la parabole $y=-\frac{(\text{max-}x)^2}{\text{max-min}}+\text{max}$, i.e. un nombre revu à la hausse. Ci-dessous, les trois méthodes avec `min=5` et `max =15`.



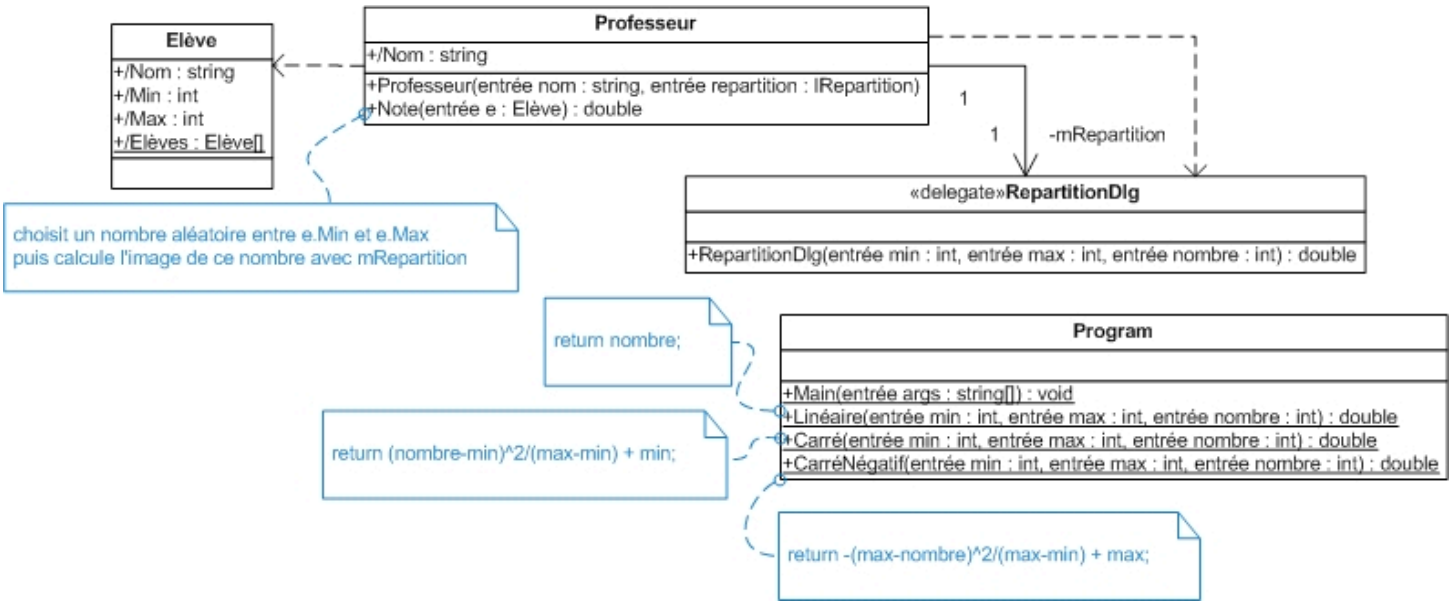
Un `Professeur` possède un `Nom` et une méthode de `IRépartition`. La méthode `Note` choisit un nombre aléatoirement entre `Min` et `Max` inclus de l'`Elève` passé en paramètre, puis utilise la classe concrète de `IRépartition` pour modifier cette note. La note obtenue est minorée à 0 et majorée à 20 puis rendue.

Créez trois `Professeurs` (un pour chaque méthode de `Répartition`) et notez les `Elèves` du tableau statique d'`Elèves` avec chacun d'entre eux et affichez les résultats.

EXERCICE 04_02

delegate
Strategy

Réécrire le programme précédent en utilisant un délégué à la place de l'interface comme le propose le diagramme de classes page suivante.



EXERCICE 04_03

delegate
Strategy
Func<,,,>

.NET contient déjà un certain nombre de types délégués dans l'espace de nom System. Parmi eux, on peut distinguer notamment la famille des Func :

```
delegate TResult Func<TResult>();
delegate TResult Func<T, TResult>(T arg);
delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
delegate TResult Func<T1, T2, T3, TResult>(T1 arg1,
                                           T2 arg2, T3 arg3);
delegate TResult Func<T1, T2, T3, T4, TResult>(T1 arg1,
                                T2 arg2, T3 arg3, T4 arg4);
```

Ces types délégués décrivent des méthodes qui rendent un `TResult` et prennent en paramètre différents arguments de type `T1`, `T2`, `T3`, `T4`...

Modifiez l'exercice précédent en utilisant un type délégué `Func` plutôt que de déclarer un délégué `RepartitionDlg`.

Réécrire le programme précédent en utilisant un délégué à la place de l'interface comme le propose le diagramme de classes suivant.

EXERCICE 04_04

delegate
Strategy

Reprenez l'exercice 02_18 avec le pattern Strategy sur `MonPanier`, en utilisant un delegate à la place de l'interface et des classes concrètes.

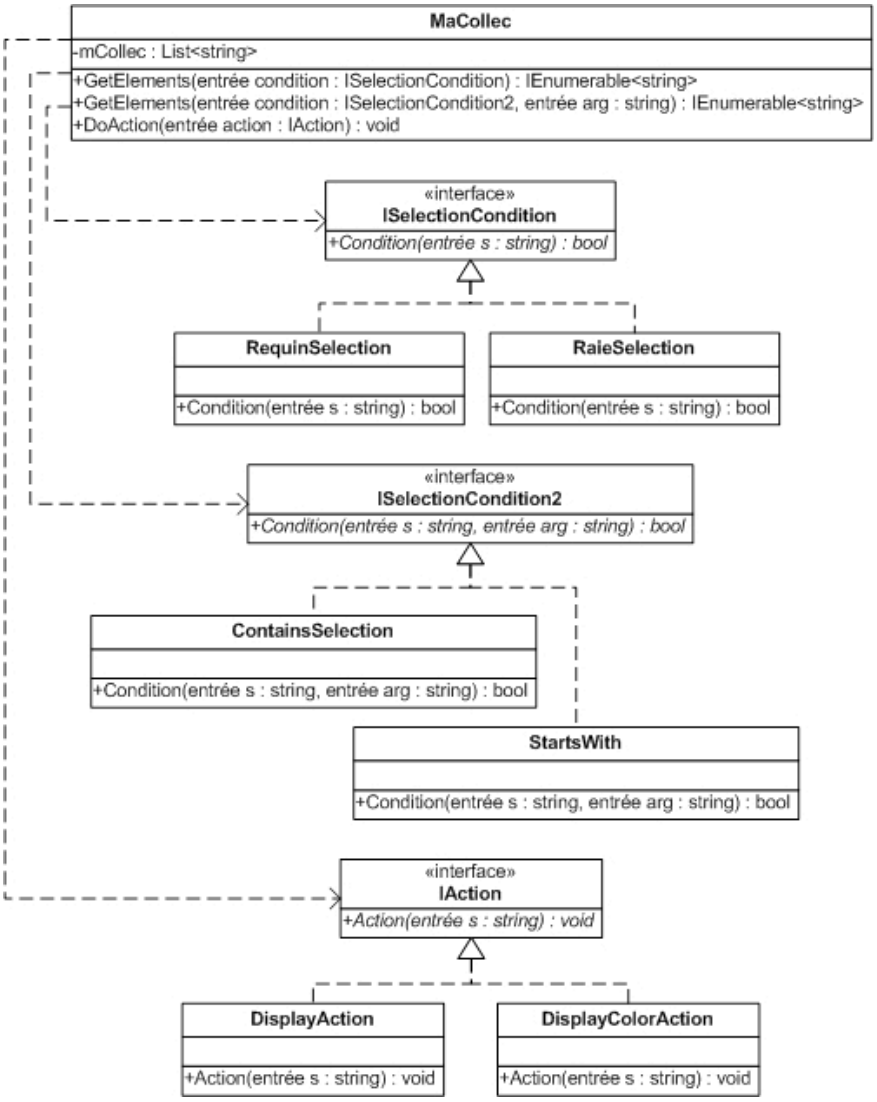
EXERCICE 04_05

delegate
Strategy
Func

Reprenez l'exercice précédent en utilisant un delegate prédéfini `Func`.

EXERCICE 04_06

interface
collections



Le but de cet exercice est de réaliser des filtrages et des actions sur les éléments d'une collection, sans utiliser LINQ ni les délégués pour le moment. Pour cela, aidez-vous du diagramme de classes ci-dessus et réalisez les opérations suivantes :

- utilisez la classe `MaCollec` partielle qui vous est fournie et qui encapsule une collection de chaîne de caractères,
- conditions simples :

- créez une interface `ISelectionCondition` qui contiendra une méthode `Condition` rendant `true` ou `false` en fonction de la chaîne de caractères passée en paramètres,
- écrivez deux classes concrètes implémentant cette interface, par exemple : `RaieSelection` recherche si la chaîne de caractères contient la chaîne «raie» (quelle que soit la casse) ; `RequinSelection` recherche si la chaîne de caractères contient la chaîne «requin»(quelle que soit la casse)
- ajoutez une méthode à `MaCollec` prenant en paramètre une condition (de type `ISelectionCondition`) et rendant une collection de string vérifiant cette condition
- testez-la avec les deux classes concrètes

- conditions plus élaborées :
  - créez une interface `ISelectionCondition2` qui contiendra une méthode `Condition` rendant `true` ou `false` en fonction de la chaîne de caractères passée en paramètres ainsi que d'une chaîne de caractères supplémentaires passées en argument,
  - écrivez deux classes concrètes implémentant cette interface, par exemple : `StartsWith` recherche si la chaîne de caractères commence par la deuxième et `ContainsWith` recherche si la chaîne de caractères contient la deuxième (n'utilisez pas LINQ pour cet exercice),
  - ajoutez une méthode à `MaCollec` prenant en paramètre une condition (de type `ISelectionCondition2`), un argument supplémentaire, et rendant une collection de string vérifiant cette condition
  - testez-la avec les deux classes concrètes
- actions :
  - créez une interface `IAction` qui contiendra une méthode `Action` ne rendant rien et réalisant quelque chose en fonction de la chaîne de caractères passée en paramètres,
  - écrivez deux classes concrètes implémentant cette interface, par exemple : `DisplayAction` qui affiche la chaîne de caractères passée et `DisplayColorAction` qui affiche en jaune la chaîne de caractères si elle fait moins de 5 caractères et en rouge si elle fait

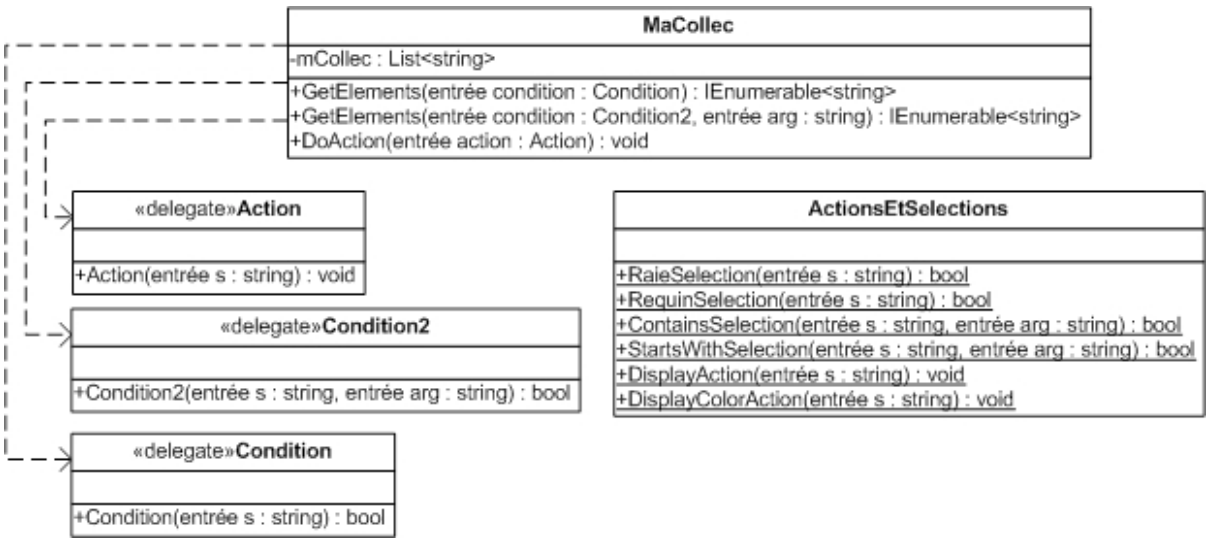
plus de 10 caractères

- ajoutez une méthode à `MaCollec` prenant en paramètre une action (de type `IAction`) et ne rendant rien, qui effectuera l'action sur chaque élément
- testez-la avec les deux classes concrètes

EXERCICE 04_07

delegate

Reprenez l'exercice 04_06, en utilisant trois delegates à la place des interfaces et des classes concrètes. Vous pourrez suivre par exemple le diagramme de classes suivants, qui définit trois types délégués internes à `MaCollec`, puis une classe statique qui contient 6 méthodes statiques (2 pour chaque type délégué). Testez le tout dans une application Console.



EXERCICE 04_08

delegate
Predicate
Func
Action

.NET contient déjà un certain nombre de types délégués dans l'espace de nom System. Parmi eux, on peut distinguer notamment la famille des `Func` (comme nous l'avons déjà vu dans l'énoncé de l'exercice 04_03) ou bien la famille des `Action` et celle des `Predicate` :

```
delegate void Action<T>(T obj);
delegate bool Predicate<T>(T obj);
delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
```

Le délégué `Action` prend un objet générique en paramètre et en fait quelque chose (sans rien rendre).

Le délégué `Predicate` prend un objet générique en paramètre et rend un `true` ou `false`.

Le délégué `Func` prend 1 ou plusieurs paramètres de différents types et rend un quelque chose d'un autre type.

Modifiez l'exercice précédent en utilisant les types délégués prédéfinis `Func`, `Predicate` et `Action`, plutôt que d'utiliser les déclarations internes des types délégués `Condition`, `Condition2` et `Action` dans `MaCollec`.

Ex04_09 : examen du 2 novembre 2010

Sujet - Jeu du morpion

OBJECTIFS

L'objectif de cet examen est de réaliser une application Console qui permet de regarder deux joueurs automatiques jouer au Morpion. Pour vous aider, deux bibliothèques vous sont fournies :

• la première `giMorpionCore.dll` contient des classes et des méthodes gérant une grande partie du jeu (démarrage du jeu, insertion de pièces, indication au prochain joueur qu'il doit jouer, test de la fin du jeu...),

• la deuxième `giMorpionTools.dll` contient une classe et une méthode statiques permettant d'afficher la grille du Morpion dans une fenêtre Console.

Mais alors que reste-t-il alors à faire ?

Ces bibliothèques ne communiquent pas, et il n'y a pas d'application. Dans la classe `Game` :

• la méthode `NotifyNextPlayer` appelle la méthode `Play` du prochain joueur (qui joue ainsi son coup),

• la méthode `Start` appelle la méthode `NotifyNextPlayer` sur le premier joueur,

• la méthode `InsertPiece` se contente d'insérer ou non la pièce et d'indiquer qui sera le prochain joueur, mais la liaison avec le reste du jeu n'est pas réalisée,

• la méthode `IsGameOver` teste si le jeu est terminé mais n'est jamais appelée.

Dans la classe `Player`, l'appel de la méthode `Play` appelle la méthode `InsertPiece` de `Game` (notez que cette méthode `InsertPiece` est virtuelle !!)

Vous avez la charge de :

• réutiliser cette bibliothèque, sans avoir accès au code source,

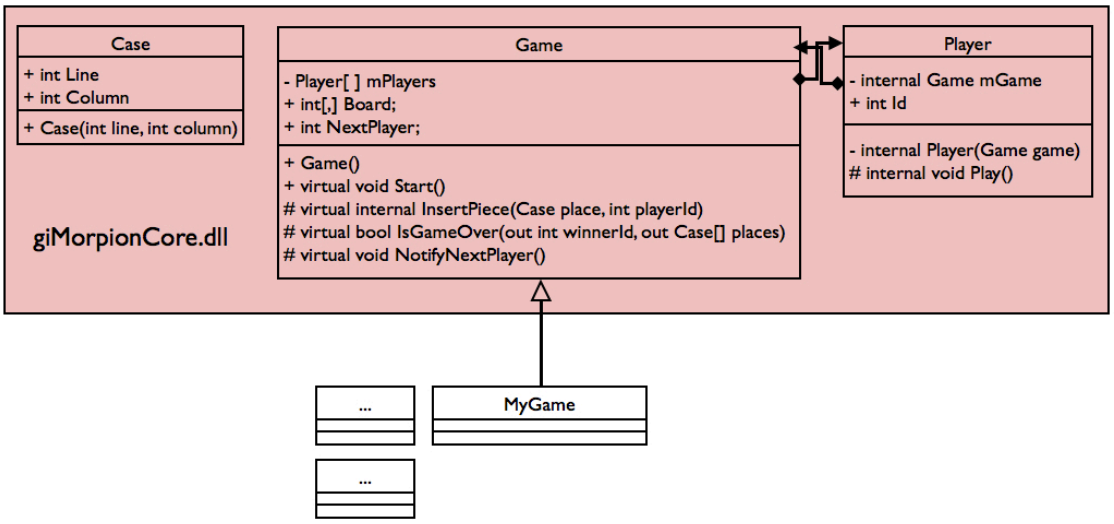
• écrire la liaison entre toutes ces méthodes pour que le jeu puisse se dérouler,

• créer des événements pour pouvoir renseigner une application extérieure sur le déroulement du jeu,

• créer une application Console permettant de visualiser le jeu et son déroulement en s'abonnant à ces événements.

QUESTION 1 : BIBLIOTHÈQUE ET ÉVÉNEMENTS

Vous devez réaliser une bibliothèque de classes `giMorpionExam.dll` contenant vos nouvelles classes pour le déroulement du jeu du Morpion, c'est-à-dire une classe `MyGame` dérivant de la classe `Game`, dans laquelle vous devez :



• ajouter des événements (et donc peut-être des classes :-) permettant de renseigner le reste du monde que :

  • le jeu a commencé,

  • le jeu est terminé (en renseignant qui a gagné, et quelle est la ligne gagnante),

  • un joueur vient d'être renseigné qu'il doit jouer (et qui est ce joueur),

  • une pièce a été insérée (où et par qui).

  Pour cela, vous pourrez choisir de réécrire des méthodes virtuelles, ou d'écrire de nouvelles méthodes. Dans tous les cas, vous devrez profiter de l'héritage de `Game` et éviter de réécrire des choses existantes, et bien choisir à quel moment lancer l'événement.

• faire en sorte que le jeu se déroule automatiquement, c'est-à-dire en respectant l'ordre suivant :

  • un joueur est renseigné que c'est son tour de jouer,

  • il joue son coup,

  • une pièce est insérée

  • si cette insertion n'est pas bonne (case occupée ou en dehors des limites), le même joueur est renseigné qu'il doit jouer,

  • si l'insertion est bonne, la fin du jeu est testée, puis le joueur suivant est renseigné qu'il doit jouer si le jeu n'est pas terminé.

  Vous pourrez choisir par exemple de réécrire la méthode `InsertPiece`, et dans cette méthode, d'appeler les autres méthodes pour garantir l'ordre de ces appels. Il

peut être d'autant plus judicieux de réécrire `InsertPiece` que `Player.Play` appelle cette méthode virtuelle.

Attention, beaucoup de choses sont déjà réalisées dans la bibliothèque `giMorpionCore.dll`. Étudiez la documentation `html` fournie pour en déduire le code à écrire et éviter de réécrire ce qui existe déjà. Apportez également une attention particulière au lancement des événements.

Un rappel utile : si vous réécrivez une méthode virtuelle `MethodA` dans une classe fille, vous pouvez appeler la méthode virtuelle `MethodA` correspondante de la classe mère à l'aide du mot clé `base` : `base.MethodA(...)`.

Conseil : respectez au maximum le pattern standard des événements.

QUESTION 2 : APPLICATION CONSOLE

Vous devez réaliser une application Console qui s'abonnera aux 4 événements créés dans la question précédente, et permettra d'afficher le jeu de la manière suivante :

- quand le jeu débute : la grille vide et un message de début

- quand un joueur est renseigné qu'il doit jouer : une phrase lui demandant de jouer

```
Player 0, it's your turn !
```

- quand un joueur a inséré une pièce : la grille avec la pièce insérée mise en évidence

```
  0 1 2
0 X   O
1     O
2   X
```

- quand la partie est terminée avec un vainqueur : la grille avec la ligne gagnante mise en évidence, et un message au vainqueur,

```
  0 1 2
0 X O O
1 O   O
2 X X X
Congratulations Player 1 for your victory !
```

- quand la partie est terminée sans vainqueur : la grille et un message pour dire qu'il n'y a pas de vainqueur

```
  0 1 2
0 X X O
1 O O X
2 X O O
Deuce
```

Conseil important : utilisez la bibliothèque `giMorpionTools.dll` pour l'affichage.

Ex04_10 : examen du 2 novembre 2011

Sujet - Tennis 2

OBJECTIFS

La fin de la saison de Tennis approche. Dwight Schrute souhaite suivre en direct pendant son travail les derniers matchs à l'aide d'un «Match Tracker» qui lui permet de connaître le score de chaque match en cours ou terminé, sans avoir à rafraîchir son application.

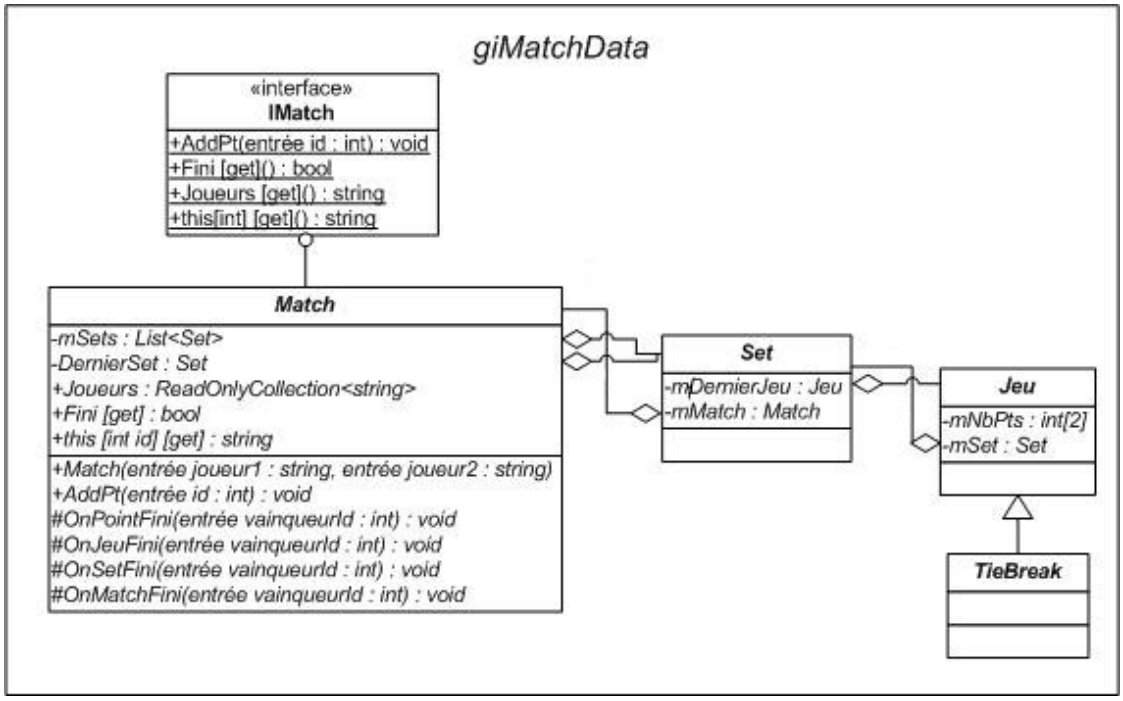
Il a déjà codé un certain nombre de classes. Malheureusement, il a beaucoup de travail à effectuer cette semaine et il n'a pas le temps de terminer son programme. C'est la raison pour laquelle il vous demande de terminer son application, afin de pouvoir suivre les résultats pendant sa semaine de travail chargée. Mais, les Schrute sont méfiants de père en fils, et, s'il accepte de vous livrer ses dlls, il refuse de vous donner son code source. Il est persuadé, que la qualité de son code vous permettrait de vous faire beaucoup d'argent sur son dos.

Votre mission, si vous l'acceptez (si vous ne l'acceptez pas, c'est o d'office...), est de terminer le Match Tracker de Dwight Schrute. Pour cela, vous pourrez suivre les conseils qui vous sont donnés dans cet énoncé.

On ne considérera ici que les matchs en deux sets gagnants. En pièce jointe à cet énoncé, vous trouverez la page wikipedia sur le Tennis avec notamment, un paragraphe Règles résumant les règles du jeu.

Dwight Schrute a implémenté trois bibliothèques :

• `giMatchData` : contient les classes permettant de stocker les résultats (ou le score actuel) d'un match, i.e. `Match`, `Set`, `Jeu` (et `TieBreak`). Le schéma suivant présente une vision simplifiée de cette bibliothèque. La méthode `AddPt` permet d'ajouter un point à une instance de `Match` en donnant l'identifiant du joueur (0 ou 1) ayant gagné le point. Les méthodes `OnPointFini`, `OnJeuFini`, `OnSetFini` et `OnMatchFini` sont protégées et virtuelles.



• `giTennisCaster` : contient une classe `TennisCaster` qui permet de simuler le flux d'informations sur chaque nouveau point dans un match. On pourra noter la présence d'un événement `PointGagné` qui est lancé à chaque fois qu'un point dans un match vient de se terminer. La méthode `Cast` permet de lancer l'écoute du flux d'informations. La propriété statique `COEFF_TEMPS` permet de modifier la vitesse de simulation du flux pour vous permettre de débugger plus rapidement (par exemple au début de l'application).

• `giMatchDisplayTool` : contient une classe statique avec une seule méthode statique permettant d'afficher un `Match` dans une fenêtre Console.

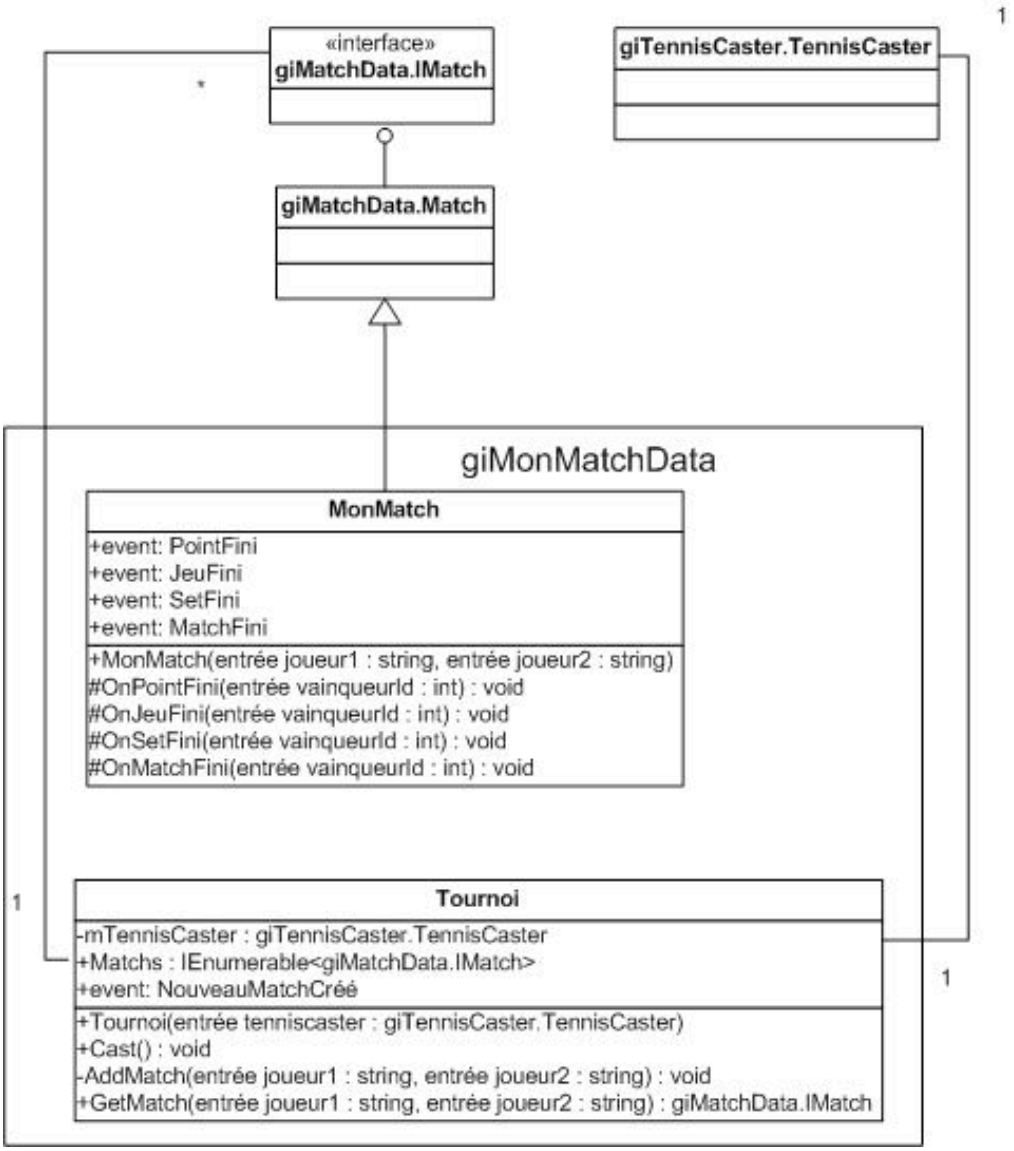
La documentation Doxygen vous est fournie !

VOUS devrez créer deux assemblages :

• `giMonMatchData` : de type bibliothèque de classes, qui devra contenir :

  • une classe `Tournoi` qui permettra de gérer les différents `Match` observés par le flux,

  • une classe `MonMatch` qui dérivera de `giMatchData.Match` et permettra l'utilisation d'événements.



• `MatchTracker` : de type Application Console, qui permettra l'affichage du déroulement des matchs en cours.

Conseil : faites des schémas au brouillon (diagramme de classes, diagramme de séquence...).

QUESTION 1 : TOURNOI ET GESTION DES MATCHS

Dans votre bibliothèque de classes `giMonMatchData`, créez une classe `Tournoi` qui permettra la gestion des `Matchs` (liste de matchs) à travers notamment :

- `GetMatch` qui permettra de récupérer le seul et unique match de la liste de matchs de ce `Tournoi` dans lequel s'affrontent les deux joueurs passés en paramètres. Si ce match n'existe pas encore dans la liste, il sera ajouté via la méthode `AddMatch`,
- `AddMatch` qui permettra d'ajouter un nouveau match dans lequel s'affrontent les deux joueurs passés en paramètres, à la liste des matchs de ce `Tournoi`.

Ces deux méthodes font que, pour récupérer un match du `Tournoi`, vous utiliserez toujours `GetMatch`, que ce match existe ou non dans le Tournoi.

- le constructeur qui permettra de s'abonner à l'événement `PointGagné` du `TennisCaster` afin d'ajouter un point au match porté par l'événement à chaque fois qu'un point est terminé.
- la méthode `Cast`, appellera la méthode `Cast` du membre de type `TennisCaster` associé à ce `Tournoi`, afin de lancer l'écoute du flux.

QUESTION 2 : ÉVÉNEMENTS SUR LE DÉROULEMENT D'UN MATCH

Ajoutez une classe `MonMatch` à votre bibliothèque de classes `giMonMatchData`, qui dérivera de `giMatchData.Match` en lui ajoutant 4 événements correspondant à la fin d'un point, la fin d'un jeu, la fin d'un set, la fin d'un match. Vous êtes responsable de la création des événements, et de leur lancement, mais pas encore de l'abonnement (question suivante). Modifiez `Tournoi` pour qu'il contienne une liste de matchs lançant ces événements.

QUESTION 3 : APPLICATION FINALE

Créez une application `MatchTracker` qui permettra d'afficher les résultats des matchs dans une fenêtre Console. Pour cela, vous devrez bien sûr vous abonner aux 4 événements de toutes les instances de `MonMatch` alors que celles-ci ne sont pas encore créées ! Pour cela, la solution proposée est la suivante :

- ajoutez à `Tournoi` un événement `NouveauMatchCréé`, qui sera lancé à chaque fois qu'un nouveau match est ajouté à la liste de matchs du `Tournoi` avec comme argument, le match créé ;
- abonnez l'application à cet événement ;
- à la réception de cet événement, l'application finale s'abonne aux 4 événements du match passé en argument.

Ex04_11 : examen du 25 octobre 2012

Sujet Event - Chef et Larbins

OBJECTIFS

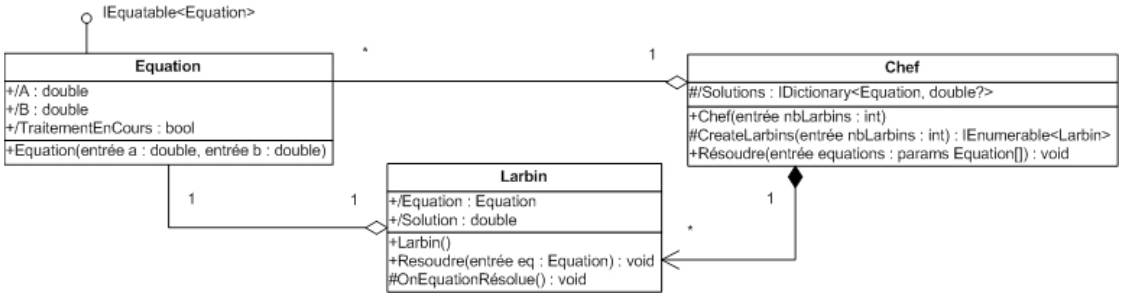
Définitions

- **Chef** : ingénieur ou chef d'entreprise qui ne sait pas résoudre des équations du premier degré mais qui possède des larbins
- **Larbin** : être sans intérêt mais sachant résoudre des équations du premier degré.

Le but de cet examen est de permettre à un Chef, à qui on demande de résoudre plusieurs équations du premier degré, de faire faire ce travail à une bande de Larbins.

ASSEMBLAGE FOURNI

Avec cet énoncé, vous trouverez une bibliothèque contenant trois classes : `Equation`, `Chef` et `Larbin`. Le diagramme de classes partiel ci-dessous les décrit.



Le principe est le suivant :

- on construit un `Chef` en lui donnant un certains nombre de `Larbins` (le constructeur de `Chef` construit les `Larbins` en appelant `CreateLarbins` (protégée et virtuelle), vous n'avez donc pas à les construire).
- on demande au `Chef` de résoudre des `Equations` du premier degré en appelant `Résoudre`. Celle méthode répartit le travail en distribuant une `Equation` par `Larbin`. En conséquence, s'il y a moins d'`Equations` que de `Larbins`, certains ne feront rien. S'il y a moins de `Larbins` que d'`Equations`, certaines `Equations` ne seront pas résolues.
- la méthode `Resoudre` de `Larbin` lance la résolution de l'`Equation` dans une méthode asynchrone, c'est-à-dire dans un autre thread. Vous n'avez pas besoin d'en comprendre le

fonctionnement. Sachez simplement que le reste de votre programme continue, même pendant la résolution de l'`Equation`, et qu'ainsi, le `Chef` peut lancer plusieurs résolutions sur plusieurs `Larbins` à la fois.

- une fois que la résolution d'une `Equation` par un `Larbin` est terminée, il appelle sa méthode `OnEquationRésolue` (protégée et virtuelle) qui ne fait rien. Sachez toutefois que les propriétés `Equation` et `Solution` possèdent l'équation résolue et sa solution au moment de l'appel de cette méthode.

CE QU'IL MANQUE... ET QU'IL FAUDRA DONC CORRIGER

- Lorsqu'une `Equation` est résolue par un `Larbin`, le dictionnaire de `Solutions` du `Chef` n'est pas mis à jour.
- S'il y a plus d'`Equations` que de `Larbins`, le `Chef` n'a donné à résoudre qu'autant d'`Equations` que de `Larbins`, mais ne redonne jamais d'`Equation` à résoudre à un `Larbin` qui vient de terminer sa première résolution d'`Equation`.
- Personne n'est au courant de l'avancement du travail (nombre d'équations résolues) et des résultats.

QUESTION 1 : BIBLIOTHÈQUE

Créez une nouvelle bibliothèque de classes qui contiendra 2 nouvelles classes : `LarbinEx` et `ChefEx`, dérivant respectivement de `Larbin` et `Chef`. Écrivez `LarbinEx` et `ChefEx` pour que :

- une instance de `LarbinEx` envoie un événement interne lorsqu'il aura terminé de résoudre une `Equation`.
- les items de la collection de `Larbins` de `ChefEx` soient de type `LarbinEx`.
- `ChefEx`, à la réception d'un événement de `LarbinEx`, mette à jour son dictionnaire de solutions,
- `ChefEx`, à la réception d'un événement de `LarbinEx`, envoie un événement pour informer le monde du nombre d'équations déjà résolues
- `ChefEx`, une fois que toutes les équations sont résolues, envoie un événement pour informer le monde que le travail est terminé et donne le dictionnaire de solutions.

QUESTION 2 : EXÉCUTABLE

Ajoutez une nouvelle application Console utilisant votre assemblage précédent. Créez une instance de `ChefEx` et donnez-lui à résoudre plus d'`Equations` que de `Larbins`. Votre application, à la réception des deux événements de `ChefEx` décrits dans la questions précédente, indiquera respectivement le nombre d'`Equations` résolues et les solutions de toutes les `Equations`.

Attention : puisque les `Larbins` exécutent leur résolution dans un thread à part (pendant 1 à 2,5 secondes), ne fermez pas la fenêtre trop tôt. Vous pourrez pour cela par exemple, utilisez à la fin de la méthode `Main()`, les deux lignes suivantes :

```
Console.WriteLine("Attendez la fin de la résolution, puis appuyez sur entrée");
Console.ReadLine();
```

Vous pourrez ensuite attendre sagement la fin des résolutions d'`Equation` avant de fermer l'application.

EXERCICE 04_12

Méthodes anonymes

Modifiez l'exercice 04_03 en utilisant des méthodes anonymes pour remplacer les méthodes `Linéaire`, `Carré` et `CarréNégatif`.

EXERCICE 04_13

Méthodes anonymes

Modifiez l'exercice 04_08 en utilisant des méthodes anonymes et supprimer la classe statique `ActionsEtSélections`.

EXERCICE 04_14

expressions lambda

Reprenez l'exercice 04_12 en utilisant des expressions lambda.

EXERCICE 04_15

expressions lambda

Reprenez l'exercice 04_13 en utilisant des expressions lambda.

EXERCICE 04_16

méthodes d'extension

Écrivez les méthodes d'extension suivantes pour des entiers :

- `Abs` : une méthode qui transforme un entier en un sa valeur absolue,
- `Borne` : une méthode qui prend une borne inférieure et une borne supérieure en paramètres, et rend l'entier s'il est compris entre les deux, la borne inférieure si l'entier est inférieur à celle-ci, la borne supérieure si l'entier est supérieur à celle-ci,
- `ToLetter` : une méthode qui rend l'entier en lettres («zéro», «un», «deux»...) si l'entier est compris entre 0 et 9 (inclus) ou «je sais pas» s'il n'est pas compris entre ces bornes.

Testez vos méthodes dans une application Console. Vous pourrez notamment tester l'enchaînement de méthodes d'extension en réalisant par exemple :

```
Console.WriteLine((-13).Abs().Borne(0, 9).ToLetter());
```