Kardinal & König

Énoncés de TP de C# .NET et XML

Marc Chevaldonné
IUT Informatique Clermont1 • 2ème année • orientation GI • septembre 2010

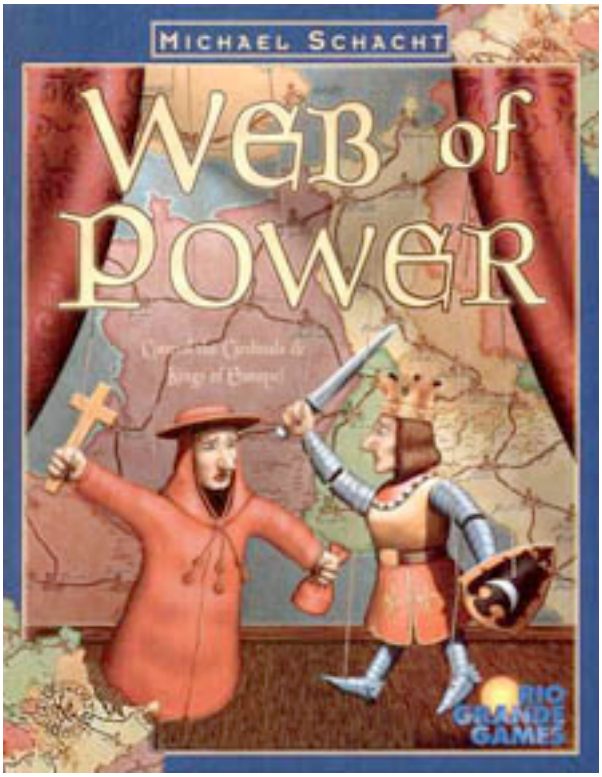
Introduction

Organisation et conseils

CONTEXTE

L'objectif de ces TP est avant tout d'initier les étudiants en 2ème année orientation GI à la programmation en C#, à la plateforme .NET et à l'XML (XML Schema, parseurs, XSLT).

Pour s'approcher de cas concrets, cette initiation s'appuie sur la réalisation d'un seul projet tout au long des 14 semaines de cours : la programmation du jeu Kardinal & König (Web Of Power) de Michael Schacht[1].



TRAVAIL EN ÉQUIPE

Dans un but pédagogique mais également pour résoudre le problème de la quantité de travail que nécessite la réalisation de ce jeu, l'application sera réalisée par des équipes de 4 étudiants réparties en 2 binômes. La séparation des tâches en deux groupes sera soutenue par l'utilisation d'un système de contrôle de versions : Git. Afin de garantir un code utilisable par les deux équipes, la documentation du code sera maintenue avec l'aide de Doxygen.

Une proposition de répartition des tâches est déjà proposée à travers les énoncés. Elle est observable par l'intermédiaire d'un code de couleur : les énoncés en vert sont conseillés pour la première équipe, ceux en rouge pour la seconde (les parties communes sont en bleu).

[1] Informations sur le jeu ici et ici. Règles du jeu ici.

Notez qu'il est préférable de suivre également le travail réalisé par l'autre groupe dans un but d'apprentissage.

L'utilisation de Git est également indispensable pour le suivi, l'aide et l'évaluation du travail des équipes.

ORGANISATION

Les énoncés ci-dessous propose une organisation du travail par binôme (cf. partie précédente), par thématique et par semaine. Ce découpage reste grossier mais il a pour but de conseiller une répartition du travail par binôme et dans le temps pour éviter les surcharges ainsi que les retards.

CONTRAINTES

Il est obligatoire de respecter les consignes données dans les règles de codage (aussi bien pour les TP que pour l'examen).

Semaine 1

Git, Doxygen, interfaces, structures, propriétés

OBJECTIFS

Les objectifs de cette première semaine, du point de vue de l'application finale, est de préparer les interfaces de nos assemblages en nous plaçant du point de vue du créateur de l'interface graphique et du point de vue des créateurs d'IA (Intelligence Artificielle).

Les objectifs pédagogiques sont :

- préparation d'un repository Git, l'utilisation des commandes principales (checkout, commit, merge, fetch, push, add...) et le travail en équipe,
- la prise en main de VisualStudio2010,
- la création automatique d'une documentation à l'aide de Doxygen et son intégration dans VisualStudio2010 à l'aide d'un makefile,
- les premiers pas en C# : écriture de structures, d'interfaces et de propriétés,
- la prise en compte de l'encapsulation,
- la découverte des types array et «nullable types»,
- l'utilisation de `ToString`

PRÉPARATION

Lisez l'énoncé ci-dessous (celui des deux parties !) et révisez les exemples de cours appropriés.

Partie 1 : préparation

VisualStudio2010, Git, Doxygen

PRÉPARATION DE GIT

Git

Préparation du repository Git

Sous Windows, créez l'arborescence suivante :

```
▸WebOfPower
  ▸VisualStudio2010
    ▸giWebOfPowerCore
      ▸giWebOfPowerCore.csproj
  ▸WebOfPower.sln
```

La solution devra contenir le projet.

Utilisez le document « Quick_Guide_Git_TortoiseGit» pour créer le repository et ajoutez lui les fichiers et dossiers créés précédemment.

Vérifiez par un clone que votre repository existe et contient bien les fichiers demandés.

Envoyez l'URL de votre repository à vos coéquipiers.

PRÉPARATION DE DOXYGEN

Doxygen
VisualStudio

Préparation de la génération automatique de la documentation

Utilisez le document «Quick_Guide_Doxygen» pour créer un makefile qui génère automatiquement la documentation du code. Pensez à bien exclure les dossiers de tests, comme indiqué dans le document.

Git
VisualStudio

Ajout de votre projet à la solution et au repository

Dès que vos coéquipiers ont créé le repository Git, clonez-le sur votre disque dur.

Ajoutez votre projet de Documentation à la solution `WebOfPower.sln` en respectant l'arborescence décrite dans le quick guide de Doxygen.

Soumettez votre travail sur le repository.

Partie 2 : API publique

interfaces, structures, enum, properties

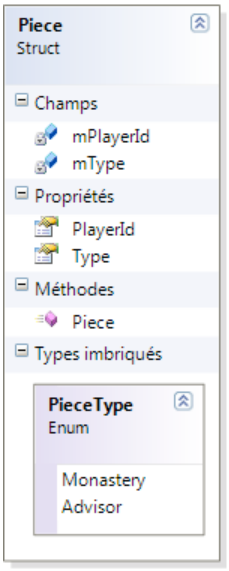
STRUCTURE PIECE ET ENUM PIECE_TYPE

struct
enum

Les types de pièces du jeu

Dans le projet `giWebOfPowerCore.csproj`, ajoutez une structure `Piece`.

Ajoutez le type imbriqué (*nested type*) `enum` `PieceType`, contenant les différents types de pièce possibles pour ce jeu (`Monastery` ou `Advisor`).



struct
readonly
property
constructor
ToString

Un type de pièce «write-once immutable»

Pour des raisons de sécurité vis-à-vis des utilisateurs (interface graphique et AI), on souhaite rendre ce type immutable *«write-once»*.

Utilisez pour cela le modificateur `readonly` et ajoutez les champs et propriétés permettant à la structure de stocker l'identifiant (`int`) du joueur possédant la pièce, le type de la pièce (`PieceType`) et un constructeur pour l'aspect *«write-once»*.

Réécrivez `ToString`.

Tests

Tests de la structure `Piece`

Ajoutez un projet de type `Console` à la solution et placez-le dans le dossier `Applications > Tests` (cf. règles de codage).

> *Note : Chaque projet doit compiler dans le dossier `bin > Debug` en phase debug et `bin > Release` en phase release. En conséquence, ouvrez les propriétés de votre projet (clic droit dessus), et dans l'onglet Build, modifiez le chemin de sortie en debug et en release.*

Un fichier `Program.cs` a été automatiquement créé avec votre projet. Ajoutez une référence à ce nouveau projet. Faites en clic droit, et allez chercher `giWebOfPowerCore`. Ajoutez votre namespace au début du fichier de code en ajoutant `using giWebOfPowerCore;` puis créez un programme qui teste la structure, ses propriétés et son constructeur.

interface
property
nullable type
tableau

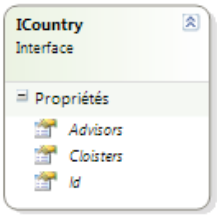
L'interface d'un cloître

Créez l'interface d'un cloître `ICloister`. Celle-ci doit contenir :

- un identifiant unique (`int`)
- l'identifiant unique (`int`) du pays auquel il appartient,
- un tableau d'identifiants uniques de cloîtres voisins par la route
- un monastère. Ce dernier sera de type `nullable type` de `Piece` ; ainsi, s'il n'y a pas de monastère sur le cloître, la valeur sera `null`, sinon on aurait une `Piece`.

interface
property
tableau

L'interface d'un pays

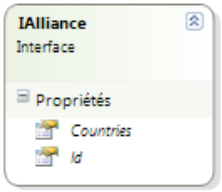


Créez l'interface d'un pays `ICountry`. Celle-ci doit contenir :

- un identifiant unique (`int`)
- un tableau d'identifiants uniques de cloîtres sur ses terres
- un tableau de conseillers (de type `Piece`). S'il n'y en a pas, le tableau sera vide, sinon, on aura l'ensemble des conseillers de ce pays dans le tableau.

interface
property
tableau

L'interface d'une alliance

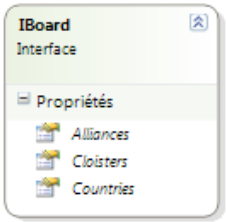


Créez l'interface d'un pays `IAlliance`. Celle-ci doit contenir :

- un identifiant unique (`int`)
- un tableau d'identifiants uniques de pays de cette alliance.

interface
property
tableau

L'interface du plateau de jeu



Créez l'interface du plateau de jeu `IBoard`. Celle-ci doit contenir :

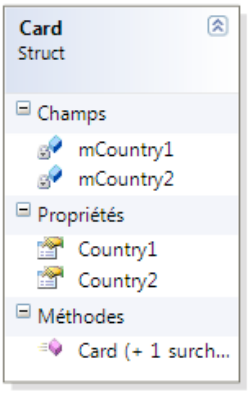
- un tableau de pays (`ICountry`),
- un tableau de cloîtres (`ICloister`),
- un tableau d'alliances (`IAlliances`).

Si vous n'avez pas encore soumis votre travail sur le repository distant, c'est peut-être un bon moment pour le faire :)

struct

Les cartes du jeu

Dans le projet `giWebOfPowerCore.csproj`, ajoutez une structure `Card`.



struct
readonly
property
constructor
ToString

Un type de carte «write-once immutable»

Pour des raisons de sécurité vis-à-vis des utilisateurs (interface graphique et AI), on souhaite rendre ce type immutable *«write-once»*.

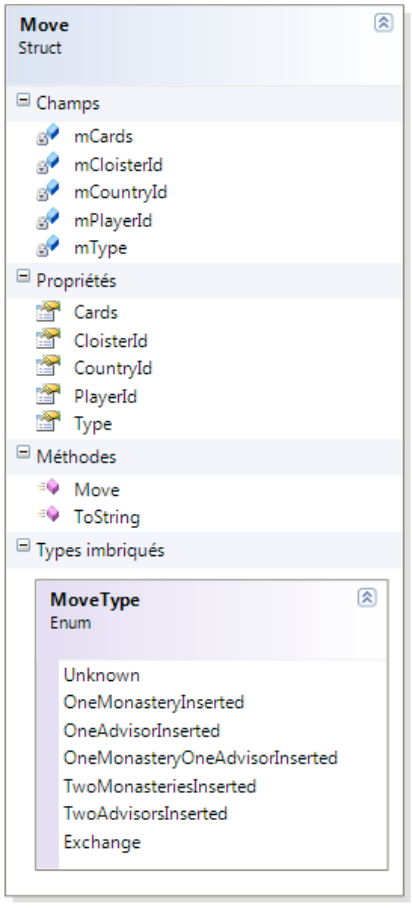
Utilisez pour cela le modificateur `readonly` et ajoutez-lui des propriétés représentant les identifiants des deux pays qu'il y a sur la carte. S'il n'y a qu'un seul pays, les 2 identifiants seront égaux. Ajoutez-lui également un constructeur pour l'aspect «*write-once*».

Réécrivez `ToString`.

struct
enum

Un coup dans le jeu (insertion d'une ou plusieurs pièces)

Dans le projet `giWebOfPowerCore.csproj`, ajoutez une structure `Move`. Ajoutez le type imbriqué (*nested type*) `enum MoveType`, contenant les différents types de coups possibles pour ce jeu (`Unknown Exchange OneMonasteryInserted, TwoMonasteriesInserted, OneAdvisorInserted, TwoAdvisorsInserted` ou `OneMonasteryAndOneAdvisorInserted`).



struct
readonly
property
constructor
ToString

Un type de coup «write-once immutable»

Pour des raisons de sécurité vis-à-vis des utilisateurs (interface graphique et AI), on souhaite rendre ce type immutable *«write-once»*.

Utilisez pour cela le modificateur `readonly` et ajoutez-lui des propriétés représentant les cartes jouées, les identifiants des cloîtres sur lesquels des monastères sont insérés, les identifiants des pays sur lesquels des

conseillers sont insérés, l'identifiant du joueur qui joue ce coup, le type de coup (du type de l'`enum` créé précédemment). Ajoutez-lui également un constructeur pour l'aspect «*write-once*», en utilisant des paramètres par défaut (`null` pour les tableaux de cartes, de cloîtres et de pays passés en paramètres).
Réécrivez `ToString`.

Tests

Tests des structures `Card` et `Move`

Ajoutez un projet de type `Console` à la solution et placez-le dans le dossier `Applications > Tests` (cf. règles de codage).

> *Note : Chaque projet doit compiler dans le dossier `bin > Debug` en phase debug et `bin > Release` en phase release. En conséquence, ouvrez les propriétés de votre projet (clic droit dessus), et dans l'onglet Build, modifiez le chemin de sortie en debug et en release.*

Un fichier `Program.cs` a été automatiquement créé avec votre projet. Ajoutez une référence à ce nouveau projet. Faites en clic droit, et allez chercher `giWebOfPowerCore`. Ajoutez votre namespace au début du fichier de code en ajoutant `using giWebOfPowerCore;` puis créez un programme qui teste la structure `Card`, ses propriétés et son constructeur.
Faites de même pour la structure `Move`.

Si vous n'avez pas encore soumis votre travail sur le repository distant, c'est le bon moment pour le faire :)

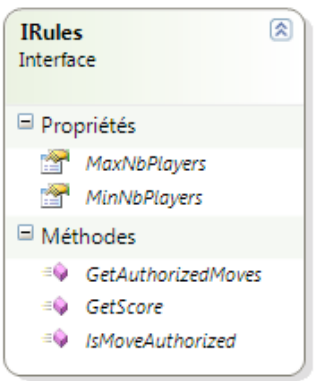
CRÉATION DES INTERFACES DES PRINCIPAUX OBJETS DE L'API 2/4

interface
property
nullable type
tableau

L'interface des règles du jeu

Créez l'interface des règles du jeu `IRules`. Celle-ci doit contenir :

- le nombre maximum de joueurs pour ces règles (`int`)
- le nombre minimum de joueurs pour ces règles (`int`)
- une méthode vérifiant si un coup est autorisé (elle prend évidemment une valeur de `Move` en paramètre)
- une méthode donnant un tableau des coups autorisés pour un joueur prenant en paramètres : l'identifiant du joueur et un tableau de cartes (`Card`) qu'il a en main



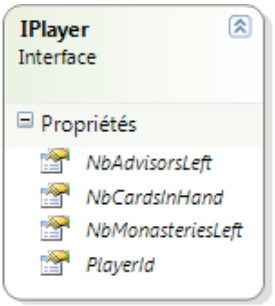
- une méthode calculant le score d'un joueur, prenant en paramètres : l'identifiant du joueur et la phase de jeu. Pour ce dernier paramètre, vous pourrez utiliser un type nullable : si la valeur est nulle, alors la méthode calcule le score du joueur seulement sur les phases terminées ; sinon, la méthode calcule le score du joueur dans la phase indiquée.

CRÉATION DES INTERFACES DES PRINCIPAUX OBJETS DE L'API 3/4

interface
property

L'interface d'un joueur

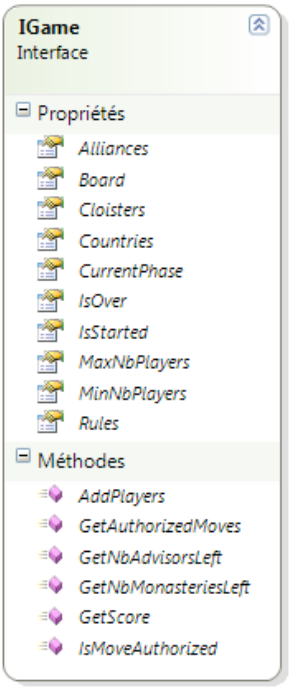
Créez l'interface d'un joueur `IPlayer`. Celle-ci doit contenir :



- un identifiant unique (`int`)
- le nombre de cartes en main,
- le nombre de monastères restants
- le nombre de conseillers restants

interface
partial
property
tableau

L'interface du jeu 1/3



Une partie de l'interface du jeu `IGame` sera créée par vos collaborateurs. Afin de ne pas écrire dans le même fichier et limiter les conflits, on peut profiter du modificateur `partial` en C# pour définir une interface (ou une classe sur plusieurs fichiers). De plus, pour améliorer la lisibilité de notre API, nous souhaitons faire en sorte qu'une seule classe serve d'intermédiaire pour la majorité des besoins. Ainsi, cette interface servira également de wrapper pour les opérations de `IBoard` et `IRules`.

En conséquence, créez l'interface partielle du jeu `IGame` (par exemple dans `IGame.BoardDetails.cs`) reprenant les propriétés de `IBoard` suivantes :

- `Alliances` (tableau de `IAlliance`)
- `Cloisters` (tableau de `ICloister`)
- `Countries` (tableau de `ICountry`)
- Enfin, rajoutez une propriété permettant de récupérer une instance du plateau `Board` de type `IBoard`.

interface
partial
property
tableau

L'interface du jeu 2/3

Créez maintenant l'interface partielle du jeu `IGame` (par exemple dans `IGame.cs`) propre au jeu lui-même :

- propriété `IsStarted` (`bool`) indiquant si la partie est commencée
- propriété `IsOver` (`bool`) indiquant si la partie est terminée
- propriété `CurrentPhase` (`int`) donnant le numéro de la phase de jeu (le jeu classique contient deux phases par exemple)
- la méthode `AddPlayers` prenant en paramètres un tableau de `IPlayer` permettant d'ajouter des joueurs au jeu
- les méthodes `GetNbAdvisorsLeft` et `GetNbMonasteriesLeft` prenant en paramètre l'identifiant d'un joueur et rendant respectivement le nombre de conseillers et le nombre de monastères qui lui restent.

Si vous n'avez pas encore soumis votre travail sur le repository distant, c'est le bon moment pour le faire :)

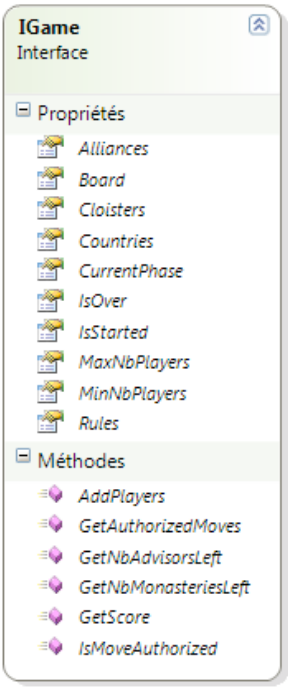
CRÉATION DES INTERFACES DES PRINCIPAUX OBJETS DE L'API 4/4

interface
partial
property
tableau

L'interface du jeu 3/3

Une partie de l'interface du jeu `IGame` sera créée par vos collaborateurs. Afin de ne pas écrire dans le même fichier et limiter les conflits, on peut profiter du modificateur `partial` en C# pour définir une interface (ou une classe sur plusieurs fichiers). De plus, pour améliorer la lisibilité de notre API, nous souhaitons faire en sorte qu'une seule classe serve d'intermédiaire pour la majorité des besoins. Ainsi, cette interface servira également de wrapper pour les opérations de `IBoard` et `IRules`.

En conséquence, créez l'interface partielle du jeu `IGame` (par exemple dans `IGame.RulesDetails.cs`) reprenant les propriétés et méthodes de `IRules`. Ajoutez également une propriété `Rules` qui rendra une instance des règles du jeu `IRules` de ce jeu.



Si vous n'avez pas encore soumis votre travail sur le repository distant, c'est le bon moment pour le faire :)

Semaine 2

classes, collections

OBJECTIFS

Les objectifs de la deuxième semaine, du point de vue de l'application finale, sont :

- d'implémenter la structure de données permettant le stockage et l'évolution du plateau de jeu.

Les objectifs pédagogiques sont :

- la création de types personnalisés, et en particulier des classes, leurs membres, propriétés et méthodes,
- l'implémentation d'interfaces,
- la création d'assemblages,
- l'utilisation de collections et les interfaces qu'elles implémentent.

PRÉPARATION

Lisez l'énoncé ci-dessous (celui des deux parties !) et révisez les exemples de cours appropriés.

Partie 3 : qqs améliorations

structures, interfaces, collections

UTILISATION DE COLLECTIONS DANS LES INTERFACES

IEnumerable
ReadOnlyCollection
interface

Modification des interfaces ICloister, ICountry, IAlliance et IBoard

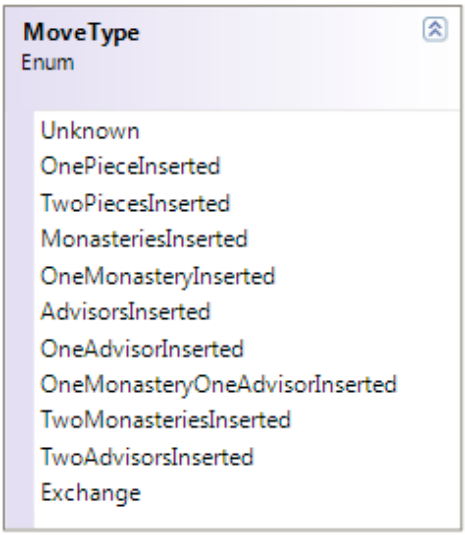
Modifiez ces quatre interfaces pour qu'elles ne contiennent plus de tableaux mais des collections. Préférez `IEnumerable<>` pour les types référence et les interfaces, et `ReadOnlyCollection<>` pour les types valeur. Ajoutez également une propriété `Name` de type `string` à `ICloister`, `ICountry` et `IBoard`.

UTILISATION DE COLLECTIONS DANS LA STRUCTURE MOVE

struct
enum

Un coup dans le jeu (insertion d'une ou plusieurs pièces)

Modifiez le type imbriqué `enum MoveType`, contenant les différents types de coups possibles (voir ci-contre) pour ce jeu, en utilisant des combinaisons. Par exemple, `OneMonasteryInserted` est la combinaison de `OnePieceInserted` et de `MonasteriesInserted`.



IEnumerable
ReadOnlyCollection
interface

Modification de la structure Move avec des collections

Modifiez la structure pour qu'elles ne contiennent plus de tableaux mais des collections (propriétés `Cards`, `Cloisters`, `Countries`).

Préférez `IEnumerable<>` pour les types référence et les interfaces, et `ReadOnlyCollection<>` pour les types valeur.

Note : `Cloisters` contient les cloîtres sur lesquels des monastères sont insérés dans le coup, et `Countries` les pays sur lesquels des conseillers sont insérés dans le coup.

Partie 4 : structure de données

Classes, membres, propriétés, méthodes, interfaces

IMPLÉMENTATION DES CLOÎTRES

class
interface

Implémentation de l'interface ICloister

Créez la classe `Cloister` (interne à l'assemblage) qui implémente l'interface `ICloister`:

static

- utilisez un membre statique privé qui incrémentera (dans le constructeur) l'identifiant de chaque cloître pour une assignation automatique à la construction.

constructeur
initialiseur

- Attention, le constructeur devra permettre également d'initialiser le pays contenant ce cloître et ajouter automatiquement ce cloître dans la liste des cloîtres de ce pays (à voir avec l'équipe rouge).

internal

- pour une gestion en interne plus rapide et plus pratique, ajoutez des membres, propriétés et méthodes (internes) pour :
  - accéder au pays contenant ce cloître (`Country` et non pas `ICountry`)

ReadOnlyCollection

  - accéder à ses voisins (`ReadOnlyCollection<Cloister>`)

(N'oubliez pas, même en interne de conserver le principe d'encapsulation : créez pour cela les membres privés correspondant à ces propriétés, par exemple `List<Cloister>` *pour les voisins)*

List

  - ajouter un voisin à ce cloître (en privé)

params

  - ajouter plusieurs voisins à ce cloître (à l'aide de `params`), en interne, en appelant plusieurs fois la méthode précédente.

ToString

- Réécrivez `ToString` de façon à obtenir sur une ligne : le nom et l'id du cloître ainsi que l'id du propriétaire du monastère (s'il y en a un).

Tests
Friend assembly

Test du cloître

Dès que l'équipe rouge aura également terminé de coder le pays, réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

class
interface

Implémentation de l'interface ICountry

Créez la classe `Country` (interne à l'assemblage) qui implémente l'interface `ICountry`:

static

- utilisez un membre statique privé qui incrémentera (dans le constructeur) l'identifiant de chaque pays pour une assignation automatique à la construction.

constructeur
initialiseur
params

- ajoutez un constructeur qui prend un nombre indéterminé de cloîtres en paramètre à l'aide du mot-clé `params`.

internal

- pour une gestion en interne plus rapide et plus pratique, ajoutez des membres, propriétés et méthodes (internes) pour :
  - accéder aux cloîtres contenus par ce pays (`Cloister` et non pas `ICloister`)

ReadOnlyCollection
List

(N'oubliez pas, même en interne de conserver le principe d'encapsulation : créez pour cela les membres privés correspondant à ces propriétés, par exemple `List<Cloister>` *pour les cloîtres)*

ToString

- Réécrivez `ToString` de façon à obtenir sur une ligne : le nom, l'id et le nombre de cloîtres du pays, puis sur des lignes suivantes, le ToString de chaque cloître du pays.

Tests
Friend assembly

Test du pays

Dès que l'équipe verte aura également terminé de coder le cloître, réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

IMPLÉMENTATION DES ALLIANCES

class
interface

Implémentation de l'interface IAlliance

Créez la classe `Alliance` (interne à l'assemblage) qui implémente l'interface `IAlliance`:

static
constructeur
initialiseur

- utilisez un membre statique privé qui incrémentera (dans le constructeur) l'identifiant de chaque alliance pour une assignation automatique à la construction.

internal

- pour une gestion en interne plus rapide et plus pratique, ajoutez des membres, propriétés et méthodes (internes) pour :
  - accéder aux pays alliés par cette alliance (`Country` et non pas `ICountry`)

ReadOnlyCollection
List

(N'oubliez pas, même en interne de conserver le principe d'encapsulation : créez pour cela les membres privés correspondant à ces propriétés, par exemple `List<Country>` *pour les pays alliés)*

ToString

- Réécrivez `ToString` de façon à obtenir sur une ligne : le nom et l'id de chaque pays allié.

Tests
Friend assembly

Test de l'alliance

Réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

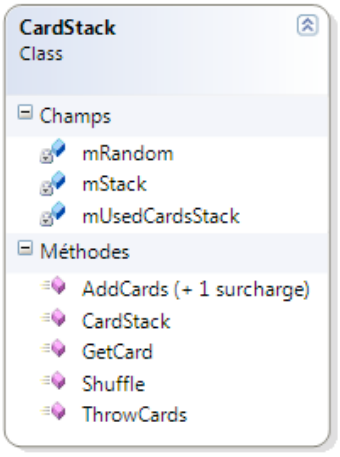
```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

IMPLÉMENTATION DE LA PILE DE CARTES

class

Implémentation de la classe CardStack

Nous allons maintenant créer un type pour représenter la pile de cartes : `CardStack`. Ajoutez cette classe au projet principal et ajoutez-lui les membres suivants :



List<>

- une liste de `Card` représentant la pile des cartes encore à tirer
- une liste de `Card` représentant la pile des cartes déjà utilisées par les joueurs

Random

- un générateur de nombre aléatoires entiers.

List<>
Random

Implémentation des méthodes de CardStack

Ajoutez les méthodes suivantes :

un constructeur,

- une méthode permettant d'ajouter n fois une même carte à la pile de cartes non utilisées (qui servira à l'initialisation de la pile),
- une méthode permettant de tirer une carte aléatoirement dans la pile de cartes non utilisées
- une méthode permettant de «jeter» des cartes dans la fosse,
- une méthode permettant de vider la fosse dans la pile de cartes non utilisées (pour réinitialiser la pile avant la phase 2).
- Réécrivez `ToString` de façon à obtenir le nombre de cartes restantes et le nombre de cartes déjà utilisées sur une seule ligne.

Tests
Friend assembly

Test de CardStack

Réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

IMPLÉMENTATION DU PLATEAU DE JEU 1/4

class
interface

Implémentation de la classe Board

Créez la classe `Board` (interne à l'assemblage) qui implémente l'interface `IBoard` (en parallèle de l'équipe verte, pensez à partial pour éviter les conflits inutiles) :

IEnumerable<>
List<>
covariance

- à chaque collection imposée par l'interface (`IEnumerable<ICloister>`, `IEnumerable<ICountry>`, `IEnumerable<Alliance>`), associez une liste du type concret correspondant (`List<Cloister>`, `List<Country>`, `List<Alliance>`). Le fait que `Cloister` implémente `ICloister` et que `List` implémente `IEnumerable`, ainsi que le fait que les interfaces génériques soient covariantes nous autorisent à rendre le membre de type `List<Cloister>` en lieu et place du getter de la propriété de type `IEnumerable<ICloister>`, tout en conservant une forte encapsulation puisque `Cloister` et `Board` restent `internal`. Pensez à initialiser correctement ces listes (via des initialiseurs ou le constructeur).

List<>
params

- ajoutez une méthode `AddCloister` et une méthode `AddCloisters` permettant d'ajouter respectivement un ou plusieurs (à l'aide de `params`) élément(s) de type `Cloister` à la liste de cloîtres précédente. Pensez à vérifier que les cloîtres ajoutés n'existent pas déjà dans la liste avant de les ajouter.
- recommencez la même opération avec `AddCountry` et `AddCountries`
- ajoutez une méthode `AddAlliance` prenant en paramètres un nombre indéterminé de pays (`Country`) à l'aide de `params`. Cette méthode vérifiera que ces pays existent, et le cas échéant, créera l'alliance et l'ajoutera à la liste correspondante.

method overload

- ajoutez les méthodes `InsertAdvisor` et `InsertMonastery` qui seront surchargées : elles pourront prendre en paramètre d'une part, un l'identifiant du pays ou du cloître où a lieu l'insertion et la pièce à insérer, et d'autre part, le pays ou le cloître où a lieu l'insertion et la pièce à insérer.
- ajoutez à l'interface `IBoard` une propriété en lecture seule `Name` contenant le nom du pays et implémenter la dans la classe `Board`

ToString

- Réécrivez `ToString` pour que cela affiche la liste des pays et des cloîtres du plateau de jeu (pensez pour cela à utiliser le `ToString` de `Cloister` et de `Country`).

Tests
Friend assembly

Test de la première partie du plateau de jeu

Réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

IMPLÉMENTATION DU PLATEAU DE JEU 2/4

class
interface

Implémentation de la classe Board

Créez la classe `Board` (interne à l'assemblage) qui implémente l'interface `IBoard` (en parallèle de l'équipe rouge, pensez à partial pour éviter les conflits inutiles) :

- L'équipe rouge s'occupe de l'implémentation des méthodes de IBoard
- Ajouter le membre et la propriété interne permettant de stocker et d'accéder à la pile de cartes du jeu (`CardStack`) en interne.

abstract

- Rendez la classe `Board` abstraite et ajoutez-lui la méthode abstraite `CreateBoard`.

IMPLÉMENTATION DU PLATEAU DE JEU 3/4

class

Implémentation d'une classe concrète de Board

Dans l'étape précédente, vous avez rendu la classe `Board` abstraite. Elle ne peut donc pas être instanciée. Ceci nous permettra d'implémenter différents plateaux de jeu concrets.

héritage
override

Créez la classe `EuropeanBoard`[2] qui dérive de la classe abstraite `Board` et réécrivez sa méthode `CreateBoard`. Dans cette méthode, vous devrez donc :

- construire les pays,
- construire les cloîtres,
- ajoutez les cloîtres aux pays,
- établir les relations de voisinage entre les cloîtres,
- construire les alliances,
- ajouter les cartes à la pile de cartes.

Tests
Friend assembly

Test du plateau de jeu européen

Réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

IMPLÉMENTATION DU PLATEAU DE JEU 4/4

class

Implémentation d'une classe concrète de Board

L'équipe verte a rendu la classe `Board` abstraite. Elle ne peut donc pas être instanciée. Ceci nous permettra d'implémenter différents plateaux de jeu concrets.

héritage
override

Créez la classe `GreeceBoard`[3] qui dérive de la classe abstraite `Board` et réécrivez sa méthode `CreateBoard`. Dans cette méthode, vous devrez donc :

---

[2] vous pouvez trouver une image du plateau de jeu européen ici

[3] vous pouvez trouver une image du plateau de jeu grec ici

- construire les pays,
- construire les cloîtres,
- ajoutez les cloîtres aux pays,
- établir les relations de voisinage entre les cloîtres,
- construire les alliances,
- ajouter les cartes à la pile de cartes.

Tests
Friend assembly

Test du plateau de jeu grec

Réalisez une application Console pour tester votre classe. Pour permettre le test des méthodes et propriétés `internal`, ajoutez la ligne suivante dans le fichier `AssemblyInfo.cs` du projet `giWebOfPowerCore`.

```
[assembly: InternalsVisibleTo(«nom_de_l_appli_de_test»)]
```

Semaine 3

classes, collections, LINQ

OBJECTIFS

Les objectifs de la deuxième semaine, du point de vue de l'application finale, sont :

- d'implémenter les règles du jeu classique et la variante.

Les objectifs pédagogiques sont :

- la création de types personnalisés, et en particulier des classes, leurs membres, propriétés et méthodes,
- l'implémentation d'interfaces,
- la création d'assemblages,
- l'utilisation de collections et les interfaces qu'elles implémentent,
- l'utilisation de dictionnaires,
- l'utilisation de LINQ.

PRÉPARATION

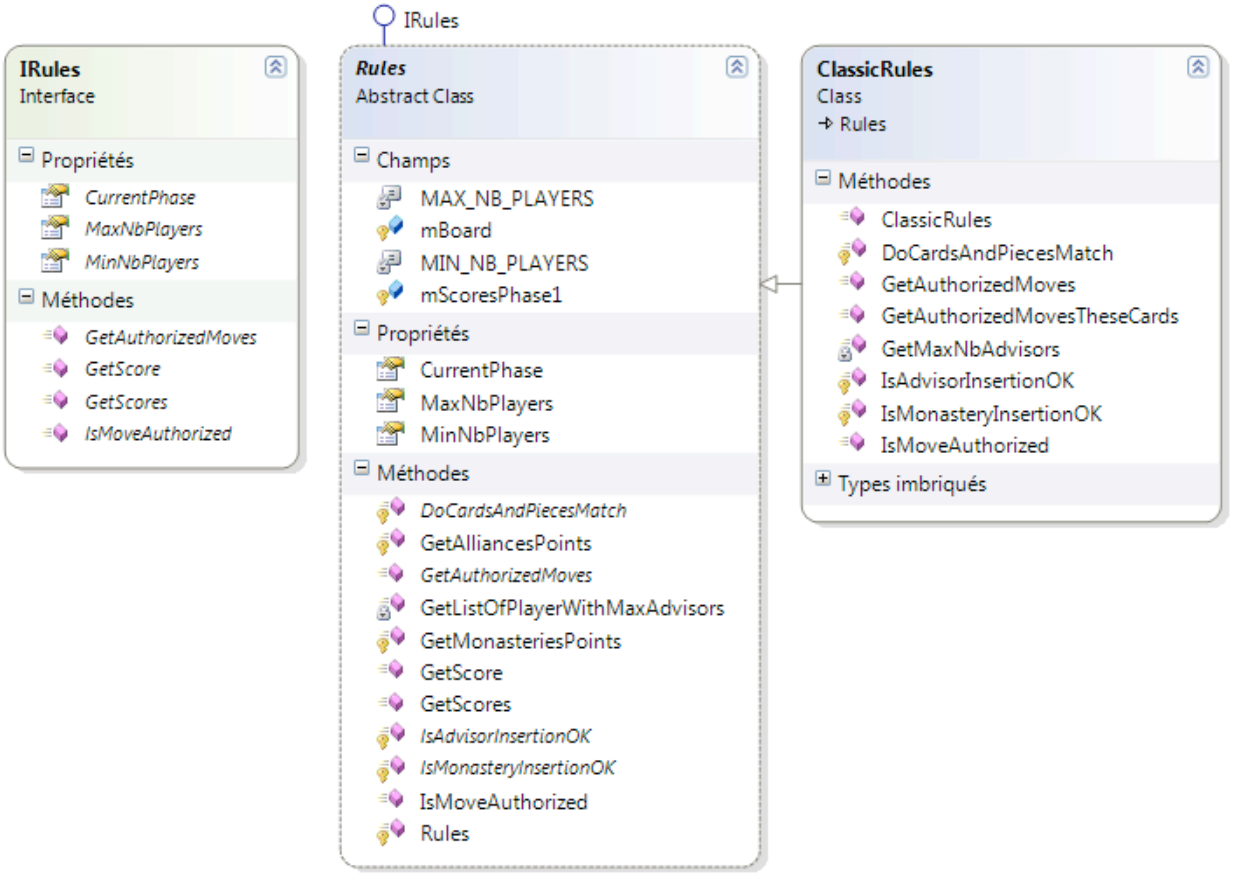
Lisez l'énoncé ci-dessous (celui des deux parties !) et révisez les exemples de cours appropriés.

Partie 5 : règles du jeu

collections, dictionnaires, LINQ, classes

INTRODUCTION

Dans cette partie, nous allons coder les règles du jeu. L'objectif est de coder les règles du jeu classiques et la variante. Nous allons donc répartir à nouveau les tâches entre deux binômes, mais nous allons également chercher les points communs entre ces deux classes pour éviter de les recoder deux fois. Pour cette raison, nous allons tout d'abord créer une classe abstraite Rules qui implémentera l'interface IRules et contiendra tous les points communs (propriétés et méthodes) des deux règles du jeu. Ensuite, nous allons créer deux classes ClassicRules et VariantRules représentant respectivement les règles du jeu classiques et la variante. Voici ci-dessous un résumé des classes et méthodes à implémenter :



IMPLÉMENTATION DE L'INTERFACE IRULES ET CRÉATION DE LA CLASSE ABSTRAITE RULES

interface
abstract
class

Implémentation de l'interface IRules

Implémentez l'interface `IRules` dans la classe `Rules`.

Implémentez particulièrement les propriétés `MaxNbPlayers` et `MinNbPlayers` en utilisant des constantes (pour les deux règles du jeu qui hériteront de `Rules`, le nombre de joueurs varie entre 3 et 5).

Ajoutez un membre `protected` de type `Board` qui représentera le plateau de jeu et qui sera initialisé dans le constructeur (`protected`) de `Rules`.

Pour permettre à l'autre équipe d'avancer, implémentez les méthodes de l'interface `IRules` en vide et faites un push.

IMPLÉMENTATION DES PROTOCOLES D'ÉGALITÉ 1/2

Equals
IEquatable<>
operator==
operator!=

Implémentation des protocoles d'égalité pour la structure Card

Implémentez les protocoles d'égalité pour la structure `Card` en respectant le pattern présenté en cours.

Equals
IEquatable<>

Implémentation des protocoles d'égalité pour la classe Alliance

Implémentez les protocoles d'égalité pour la classe `Alliance` en respectant le pattern présenté en cours.

IMPLÉMENTATION DES PROTOCOLES D'ÉGALITÉ 2/2

Equals
IEquatable<>

Implémentation des protocoles d'égalité pour la classe Country

Implémentez les protocoles d'égalité pour la classe `Country` en respectant le pattern présenté en cours.

Equals
IEquatable<>

Implémentation des protocoles d'égalité pour la classe Cloister

Implémentez les protocoles d'égalité pour la classe `Cloister` en respectant le pattern présenté en cours.

PRÉPARATION DE LA VÉRIFICATION ET DE LA RECHERCHE DES COUPS AUTORISÉS

Les questions suivantes ont pour but de coder les méthodes permettant de vérifier qu'un coup est autorisé (`IsMoveAuthorized`) et de donner la liste de tous les coups autorisés (`GetAuthorizedMoves`). Pour cela, nous allons décomposer ces méthodes en plus petites méthodes `protected`. Pour vérifier qu'un coup est autorisé, nous allons :

- vérifier que les pièces insérées sont compatibles avec les cartes jouées (par exemple, 2 monastères insérées en Angleterre et 2 cartes France + 1 carte Angleterre est un coup autorisé) à travers la méthode `DoCardsAndPiecesMatch`. Cette méthode sera abstraite dans la classe `Rules` et réécrite dans les deux classes filles car ceci diffère dans les deux règles du jeu (classique et variante). Vous pourrez d'ailleurs créer pour cette méthode plusieurs méthodes plus petites pour améliorer la lisibilité.
- vérifier que l'insertion du ou des monastères est autorisée (par exemple, les cloîtres sur lesquels sont insérés les monastères sont vides, il y a déjà au moins un monastère dans le pays d'insertion pour insérer deux monastères (pour les règles classiques uniquement)). Cette méthode (`IsMonasteryInsertionOK`) sera abstraite dans la classe `Rules` et réécrite dans les deux classes filles car les règles sont différentes à ce sujet.
- vérifier que l'insertion du ou des conseillers est autorisée (par exemple, on ne peut pas insérer un conseiller si le nombre de conseillers est égal au nombre de monastères du joueur majoritaire). Cette méthode (`IsAdvisorInsertionOK`) sera abstraite dans la classe `Rules` et réécrite dans les deux classes filles car les règles sont différentes à ce sujet.

Enfin, nous utiliserons ces méthodes pour l'implémentation de `IsMoveAuthorized` et de `GetAuthorizedMoves`.

IMPLÉMENTATION DE LA VÉRIFICATION D'UN COUP

abstract

Ajout des méthodes abstraites

Ajoutez les méthodes abstraites `DoCardsAndPiecesMatch`, `IsMonasteryInsertionOK` et `IsAdvisorInsertionOK` à la classe `Rules` et n'oubliez pas de faire le push pour l'autre équipe. Ces méthodes prendront un `Move` en paramètre et rendront un `bool`.

héritage

Ajout de la classe ClassicRules

Ajoutez la classe `ClassicRules` qui hérite de `Rules`.

override

Réécriture de la méthode DoCardsAndPiecesMatch

Réécrivez la méthode `DoCardsAndPiecesMatch` dans la classe `ClassicRules`. Vous pourrez pour cette méthode, vérifier que le type de `Move`, les cartes jouées et les pièces insérées sont cohérentes en testant les cas suivants :

- 1 carte jouée et 1 pion inséré
- 2 cartes jouées et 2 pions insérés
- 2 cartes jouées et 1 pion inséré
- 3 cartes jouées et 2 pions insérés.

override

Réécriture de la méthode IsMonasteryInsertionOK

Réécrivez la méthode `IsMonasteryInsertionOK` dans la classe `ClassicRules`. Vous pourrez notamment vérifier par exemple que deux monastères ne sont pas insérés dans un pays vide, que les deux monastères sont différents...

override

Réécriture de la méthode IsAdvisorInsertionOK

Réécrivez la méthode `IsAdvisorInsertionOK` dans la classe `ClassicRules`. Pour cela, vous devez d'abord vérifier que le nombre de conseillers + le nombre de conseillers que le joueur cherche à insérer est inférieur au nombre de monastères du joueur majoritaire dans le pays.

Créez pour cela la méthode `GetMaxNbAdvisors` qui rend un entier (le nombre de conseillers maximum autorisés dans le pays, soit le nombre de monastères du joueur majoritaires) et prend en paramètre un pays (`Country`). Dans cette méthode, créez une variable de type `Dictionary<int, int>` où les clés correspondront à l'identifiant des joueurs et les valeurs au nombre de monastères de chaque joueur.

Dictionary<,>

LINQ
Count
Max

Remplissez les valeurs à l'aide de LINQ, c'est-à-dire écrivez une instruction LINQ par joueur, qui comptera le nombre de monastères de ce joueur. Pour récupérer le maximum, écrivez une instruction LINQ à nouveau.

Finissez la méthode `IsAdvisorInsertionOK` à l'aide de cette méthode.

abstract

Ajout des méthodes abstraites

L'équipe rouge a ajouté (ou va ajouter) les méthodes abstraites `DoCardsAndPiecesMatch`, `IsMonasteryInsertionOK` et `IsAdvisorInsertionOK` à la classe `Rules`. Ces méthodes prennent un `Move` en paramètre et rendent un `bool`. Si le push n'est pas fait, faites-le, ajoutez vous-mêmes ces méthodes, c'est très rapide.

héritage

Ajout de la classe VariantRules

Ajoutez la classe `VariantRules` qui hérite de `Rules`.

override

Réécriture de la méthode DoCardsAndPiecesMatch

Réécrivez la méthode `DoCardsAndPiecesMatch` dans la classe `VariantRules`. Vous pourrez pour cette méthode, vérifier que le type de `Move`, les cartes jouées et les pièces insérées sont cohérentes en testant les cas suivants :

- 1 carte jouée et 1 pion inséré
- 2 cartes jouées et 2 pions insérés
- 2 cartes jouées et 1 pion inséré
- 3 cartes jouées et 2 pions insérés.

override

Réécriture de la méthode IsMonasteryInsertionOK

Réécrivez la méthode `IsMonasteryInsertionOK` dans la classe `VariantRules`. Vous pourrez notamment vérifier par exemple que si un monastère est inséré dans un pays vide, il n'est pas accompagné d'un conseiller...

override

Réécriture de la méthode IsAdvisorInsertionOK

Réécrivez la méthode `IsAdvisorInsertionOK` dans la classe `VariantRules`. Pour cela, vous devez d'abord vérifier que le nombre de conseillers + le nombre de conseillers que le joueur cherche à insérer est inférieur au nombre de monastères du joueur majoritaire dans le pays.

Créez pour cela la méthode `GetMaxNbAdvisors` qui rend un entier (le nombre de conseillers maximum autorisés dans le pays, soit le nombre de monastères du joueur majoritaires) et prend en paramètre un pays (`Country`). Dans cette méthode, créez une variable de type `Dictionary<int, int>` où les clés correspondront à l'identifiant

Dictionary<,>

LINQ
Count
Max

des joueurs et les valeurs au nombre de monastères de chaque joueur. Remplissez les valeurs à l'aide de LINQ, c'est-à-dire écrivez une instruction LINQ par joueur, qui comptera le nombre de monastères de ce joueur. Pour récupérer le maximum, écrivez une instruction LINQ à nouveau.

Finissez la méthode `IsAdvisorInsertionOK` à l'aide de cette méthode.

Implémentation de la méthode IsMoveAuthorized

À l'aide des 3 méthodes implémentées précédemment, écrivez la méthode `IsMoveAuthorized` dans la classe `Rules`. Elle sera ainsi accessible par tous.

RECHERCHE DE TOUS LES COUPS AUTORISÉS

List<>

Implémentation de GetAuthorizedMoves

Le but de cette méthode est de trouver, connaissant les cartes qu'un joueur a en main, l'ensemble des coups autorisés pour lui.

Pour écrire cette méthode, vous pourrez écrire une première méthode (`GetAuthorizedMovesWithTheseCards`) qui recherchera tous les coups autorisés pour ce joueur mais en utilisant exactement toutes les cartes passées en paramètre. Vous devrez donc tester l'insertion, en créant des `Move` avec ces cartes, d'un monastère quelconque, de deux monastères quelconques, d'un conseiller quelconque, de deux conseillers quelconques et d'un monastère et un conseiller quelconques. Vous utiliserez la méthode `IsMoveAuthorized` si les coups sont autorisés et vous les ajouterez à la liste de coups autorisés le cas échéant. `GetAuthorizedMoves` n'aura ainsi plus qu'à générer l'ensemble des combinaisons de cartes possibles et appeler la méthode précédente avec chacune de ces combinaisons. Par exemple, si le joueur possède en main deux cartes Frankreich et une carte England/Schwaben, il faudra tester les combinaisons suivantes : 1 carte Frankreich ; 1 carte England/Schwaben ; 2 cartes Frankreich ; 1 carte Frankreich et 1 carte England/Schwaben ; 3 cartes. Vous pourrez utiliser LINQ pour vérifier s'il existe des paires et déterminer plus rapidement les combinaisons.

LINQ
Where
Count
First

Ces deux méthodes sont implémentées dans la classe `Rules` et ne dépendent donc pas des règles du jeu classiques ou variantes.

PRÉPARATION AU CALCUL DES SCORES

Les questions suivantes ont pour but de coder les méthodes permettant de calculer les scores. L'équipe verte est en charge de compter les points des alliances. L'équipe rouge comptera les points des monastères. Les chaînes de monastères sont volontairement oubliées pour décharger les TP (mais vous pouvez les coder si vous le souhaitez :). L'équipe rouge fera ensuite l'inventaire des points en fonction de la phase.

CALCUL DES POINTS DUS AUX MONASTÈRES

Dictionary<,>
List<>
IEnumerable<>
LINQ
Where
OrderBy
OrderByDescending
Count
...

Implémentation d'une méthode GetMonasteriesPoints

Ajoutez à la classe `Rules` une méthode permettant de calculer les points dus aux monastères. Cette méthode ne rendra rien, mais prendra en paramètre une variable de type `Dictionary<int, int>` où la clé représentera l'identifiant d'un joueur et la valeur son nombre de points. Pensez à ajouter les points dans le dictionnaire et non à les remplacer : nous supposons ainsi que les joueurs ont déjà des points précédentes et que nous ne faisons qu'en ajouter.

L'idée est de faire cette méthode avec le moins d'instructions possibles en utilisant LINQ. Toutefois, il vous est proposé de suivre le plan suivant :

- à l'aide de LINQ, récupérez une collection de cloîtres possédant un monastère
- à l'aide de LINQ et de la collection précédente, ordonnez ces cloîtres en fonction de l'identifiant du joueur possédant le monastère,
- à l'aide de LINQ, d'une sous-requête et de la collection précédente, ordonnez ces cloîtres par joueur, tel que les cloîtres du joueur majoritaire arrivent en premier dans la collection, puis ceux du second, etc.
- inspirez-vous des trois lignes précédentes pour trouver, à l'aide de sous-requêtes et de LINQ une méthode pour obtenir le même résultat en une instruction.

À partir de la collection obtenue, déduisez les scores de chaque joueur (attention aux joueurs à égalité !).

CALCUL DES POINTS DUS AUX ALLIANCES

IEnumerable<>
LINQ
Empty
Max
Count
Where
Select
Distinct
...

Implémentation d'une méthode GetListOfPlayersWithMaxAdvisors

Ajoutez à la classe `Rules` une méthode permettant de trouver la liste des joueurs ayant le plus de conseillers dans un pays.

Pour cela, utilisez LINQ :

rendez une collection vide s'il n'y a pas de conseillers,

à l'aide de LINQ, calculez le nombre de conseillers maximum possédés par un joueur dans ce pays,

à l'aide de LINQ, déterminez tous les joueurs possédant ce nombre de conseillers (en cas d'égalité).

Dictionary<,>
IEnumerable<>
LINQ
Intersect
Sum
...

Implémentation d'une méthode GetAlliancesPoints

Ajoutez à la classe `Rules` une méthode permettant de calculer les points obtenus pour chaque joueur grâce aux alliances. Cette méthode ne rendra rien, mais prendra en paramètre une variable de type `Dictionary<int, int>` où la clé représentera l'identifiant d'un joueur et la valeur son nombre de points. Pensez à ajouter les points dans le dictionnaire et non à les remplacer : nous supposons ainsi que les joueurs ont déjà des points précédentes et que nous ne faisons qu'en ajouter.

Vous pourrez pour cela, utiliser la méthode précédente pour trouver la liste des joueurs majoritaires dans les pays d'une alliance, et utiliser LINQ pour trouver les joueurs communs à ces deux collections. Enfin, utilisez LINQ pour calculer le nombre de points pour les joueurs majoritaires dans tous les pays de l'alliance.

CALCUL DES SCORES

Dictionary<,>

Calcul des scores des joueurs

Ajoutez à la classe `Rules` une méthode (`GetScores`) permettant de calculer les scores de tous les joueurs en utilisant les méthodes `GetMonasteriesPoints` et `GetAlliancesPoints` précédentes.

Vous prendrez en compte la phase : s'il s'agit de la phase 1, le score ne prend en compte que les monastères ; s'il s'agit de la phase 2, le score prend en compte les monastères et les alliances.

Déduisez-en la méthode `GetScore` rendant le score d'un seul joueur.

Semaine 4

classes, collections, LINQ

OBJECTIFS

Les objectifs de la deuxième semaine, du point de vue de l'application finale, sont :

- de terminer l'implémentation des règles du jeu,
- de préparer le développement des événements.

Les objectifs pédagogiques sont :

- la création de types personnalisés, et en particulier des classes, leurs membres, propriétés et méthodes,
- l'implémentation d'interfaces,
- la création d'assemblages,
- l'utilisation de collections et les interfaces qu'elles implémentent,
- l'utilisation de dictionnaires,
- l'utilisation de LINQ.

PRÉPARATION

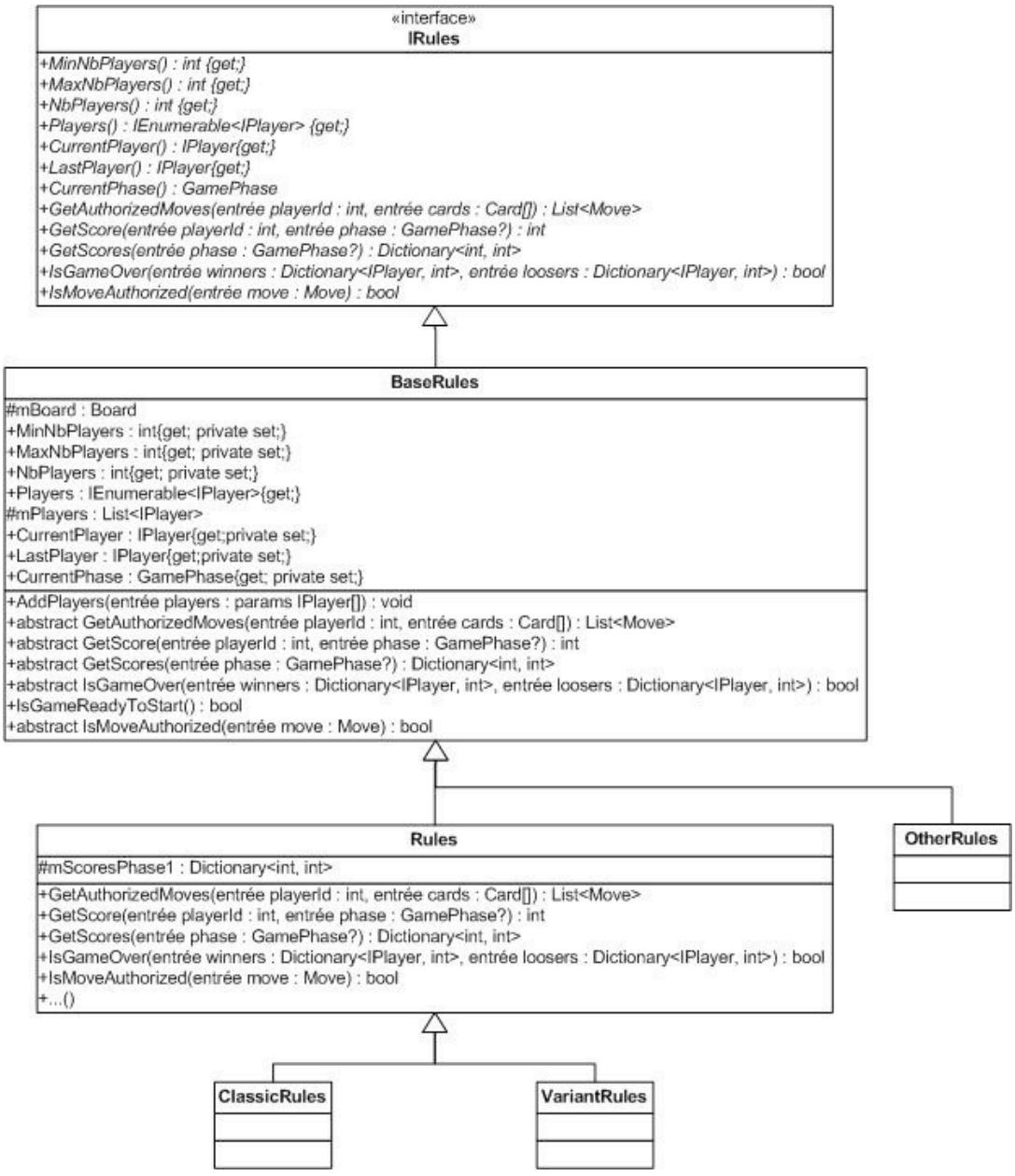
Lisez l'énoncé ci-dessous (celui des deux parties !) et révisez les exemples de cours appropriés.

Partie 6 : règles du jeu (suite) et préparation de Game

collections, dictionnaires, LINQ, classes

INTRODUCTION

Dans cette partie, nous allons terminer de coder les règles du jeu, avant de commencer le développement des événements, c'est-à-dire de la classe `Game` et du déroulement du jeu. En ce qui concerne les règles du jeu, l'implémentation proposée est la suivante :



L'interface `IRules` définit le comportement. La classe abstraite `BaseRules` implémente les méthodes communes à toutes les règles : en particulier celles gérant les joueurs (`AddPlayers`, `CurrentPlayer`, `LastPlayer`...). Les autres classes (`Rules`, `ClassicRules`, `VariantRules`, `OtherRules`...) héritent de `BaseRules` et définissent des règles concrètes. Pour le moment, nous n'avions pas géré les joueurs à travers les règles du jeu. Le but des questions suivantes est d'introduire la classe `BaseRules` permettant de gérer les joueurs et de modifier les règles du jeu en conséquence.

AMÉLIORATION DES CLASSES DÉCRIVANT LES RÈGLES DU JEU

enum

Ajout d'un enum GamePhase

Jusqu'à maintenant, nous avons décrit les phases par un entier. Pour améliorer la gestion des phases, nous avons besoin d'une description plus détaillée. Codez pour cela un `enum GamePhase` qui comprendra les valeurs :

- `Phase1` (i.e. jusqu'à ce que la pile soit vidée une première fois),
- `Phase2` (i.e. jusqu'à ce que la pile soit vidée une deuxième fois),
- `Phase3` (i.e. la toute fin de la partie, entre le moment où la pile a été vidée pour la deuxième fois et le dernier coup du dernier joueur),
- `Finished` (i.e. quand la partie est terminée).

interface
abstract

Amélioration de l'interface IRules

Modifiez l'interface comme dans le diagramme de classe de la page précédente :

- ajoutez une propriété `CurrentPhase` en modifiant son type de `int` à `GamePhase`,
- modifiez les méthodes `GetScore` et `GetScores` en utilisant `GamePhase`,
- ajoutez une méthode `AddPlayers` (permettant de rajouter différents joueurs) et une propriété `Players` (permettant d'y accéder publiquement)
- ajoutez une méthode `IsGameReadyToStart`, permettant de vérifier que les conditions du démarrage du jeu sont réunies
- ajoutez une méthode `IsGameOver` qui vérifie si la partie est terminée
- ajoutez des propriétés pour le joueur courant (`CurrentPlayer`) et le dernier joueur à avoir joué (`LastPlayer`).

interface
abstract
class

Ajout de la classe BaseRules

Ajoutez la classe `BaseRules` qui implémente l'interface `IRules`.

Implémentez les méthodes et propriétés suivantes de `IRules` :

- `MinNbPlayers`, `MaxNbPlayers`, `NbPlayers`,
- `CurrentPhase`,
- `IsMoveAuthorized` (`abstract`)
- `GetAuthorizedMoves` (`abstract`)
- `GetScores` (`abstract`) et `GetScore`
- `Players`, `AddPlayers`, `CurrentPlayer`, `LastPlayer`
- `GetNextPlayer` : qui permettra de setter `LastPlayer` comme le joueur courant et `CurrentPlayer` comme le joueur suivant,
- le membre `mBoard` protégé et `IsGameReadyToStart` : qui vérifie les conditions nécessaires au lancement du jeu : le nombre de joueurs est compris entre `MinNbPlayers` `MaxNbPlayers` et `mBoard` n'est pas `null`.
- la méthode abstraite `IsGameOver`

Modifiez la classe `Rules` en conséquence : supprimez les quelques propriétés déjà implémentées dans `BaseRules` et réécrivez `IsGameOver`. Modifiez également `IsMoveAuthorized` pour qu'elle vérifie que le coup est fait par le joueur courant.

IMPLÉMENTATION DE L'INTERFACE IPLAYER

interface

Amélioration de l'interface IPlayer

Ajoutez à l'interface `IPlayer` :

- une méthode `Play` (ne prenant rien et ne rendant rien),
- une propriété `Game` de type `IGame` (`protected`) représentant le jeu en cours,
- une propriété `Moves` de type `ReadOnlyCollection<Move>` (`protected`) permettant d'accéder aux différents coups possibles pour ce joueur,
- une propriété `CardsInHand` de type `ReadOnlyCollection<Card>` (`internal`) permettant d'accéder aux cartes de ce joueur.

interface
classe
Equals
IEquatable<>

Implémentation de l'interface IPlayer dans la classe Player

Créez une classe `Player` qui implémente l'interface `IPlayer`, notamment :

- en laissant la méthode `Play` abstraite
- en ajoutant un constructeur `protected`

Implémentez les protocoles d'égalité pour la classe `Player`.

IMPLÉMENTATION DE L'INTERFACE IGAME

interface

Amélioration de l'interface IGame

`IGame` doit servir de wrappers pour les classes implémentant `IRules` et `IBoard`, c'est-à-dire que les utilisateurs de notre assemblage passeront toujours par `IGame` pour accéder à quelques méthodes de `IRules` et `IBoard`.

Modifiez l'interface `IGame` pour qu'elle respecte les nouvelles signatures des méthodes de `IRules` et `IBoard` wrappées dans `IGame.BoardDetails.cs` et `IGame.RulesDetails.cs`

interface
class

Implémentation de l'interface IGame dans la classe Game

Créez la classe `Game` qui implémente l'interface `IGame`. Chaque méthode «wrappant» une méthode de `IRules` ou `IBoard` appellera la méthode correspondante de ce membre. Par exemple, la méthode `IGame.AddPlayers` appellera `Rules.AddPlayers`.

AMÉLIORATION DE CARDSTACK

class
nullable type

Amélioration de CardStack

J'ai oublié quelques règles du jeu dans les TP précédents : deux cartes de la pioche doivent être retournées face visible sur la table. Un joueur peut ainsi piocher dans ces deux cartes ou dans la pioche.

Rajoutez à la classe `CardStack` :

- un tableau de `Card?` représentant les cartes face visible et une propriété permettant de lire et d'écrire (`private`) le nombre de cartes de ce tableau. Le setter de cette propriété sera appelé dans le constructeur de `Card`.
- modifiez la méthode `GetCard` pour qu'elle rende un `Card?` : la valeur de retour sera `null` s'il n'y a plus de cartes dans la pile

• ajoutez la méthode `GetFaceUpCard` rendant un `Card?` et prenant un indice : si cet indice est valide (0 ou 1 par exemple), la carte correspondante dans le tableau de cartes visibles est rendue, sinon c'est `null`

• ajoutez une méthode permettant de remplir ce tableau de cartes visibles à l'aide de la pioche.

Semaine 5

délégués, événements

OBJECTIFS

Les objectifs de la cinquième semaine, du point de vue de l'application finale, sont :

- de définir les relations entre les joueurs et la classe Game,
- de définir les relations de communication entre la classe Game et les clients (interfaces graphiques et exécutables).

Les objectifs pédagogiques sont :

- la création d'événements,
- le respect du pattern standard,
- l'abonnement aux événements,
- l'implémentation d'une application cliente.

PRÉPARATION

Lisez l'énoncé et les explications ci-dessous (celui des deux parties !) et révisez les exemples de cours appropriés.

Partie 6 : fin de la préparation de l'API : événements

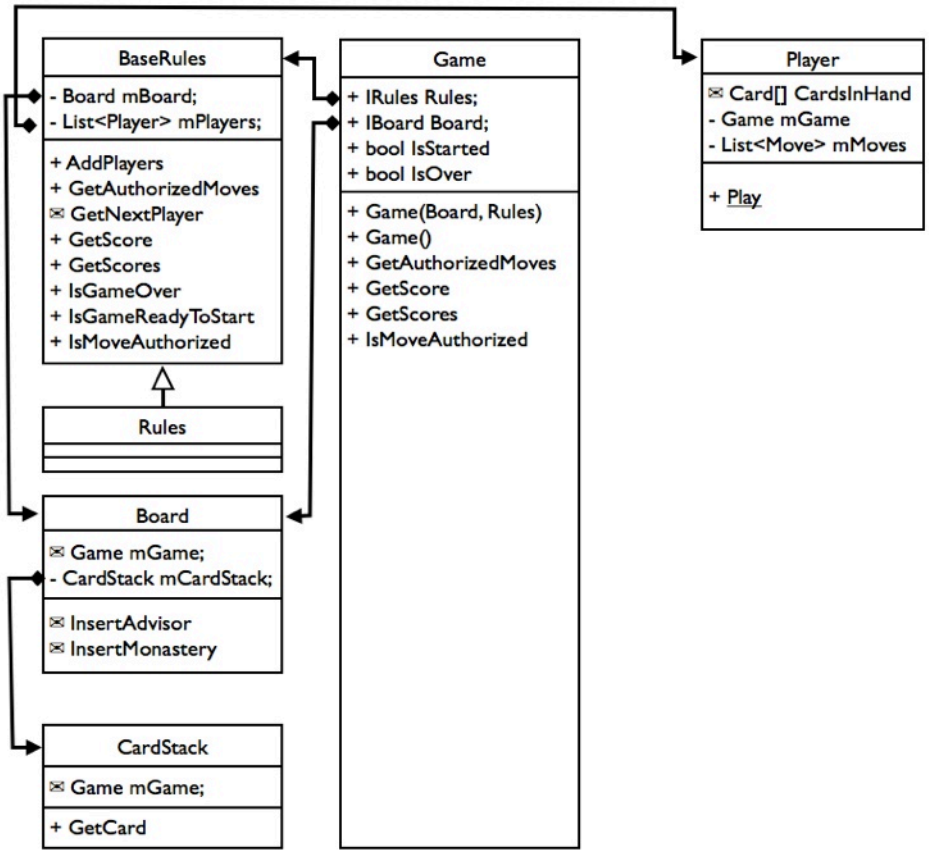
délégués, événements

INTRODUCTION

Dans un but pédagogique, nous allons simplifier pour cette partie les règles du jeu :

- le jeu ne comporte plus qu'une seule phase qui se termine lorsque la pile est vide,
- un joueur ne peut remplir sa main qu'avec les cartes de la pioche : nous ne prenons donc plus en considération les cartes qui étaient posées faces visibles sur la table,
- la partie se termine :
  - si la pioche est vide,
  - s'il n'est plus possible de placer de pièces pour aucun joueur
  - si tous les joueurs n'ont plus de pièces,
- les autres règles sont toujours valables.

À la fin de la partie 5, nous avons obtenu le résultat ci-contre (évidemment très simplifié...) (Les enveloppes représentent des membres, propriétés ou méthodes internes).



Nous souhaitons maintenant réaliser les tâches suivantes :

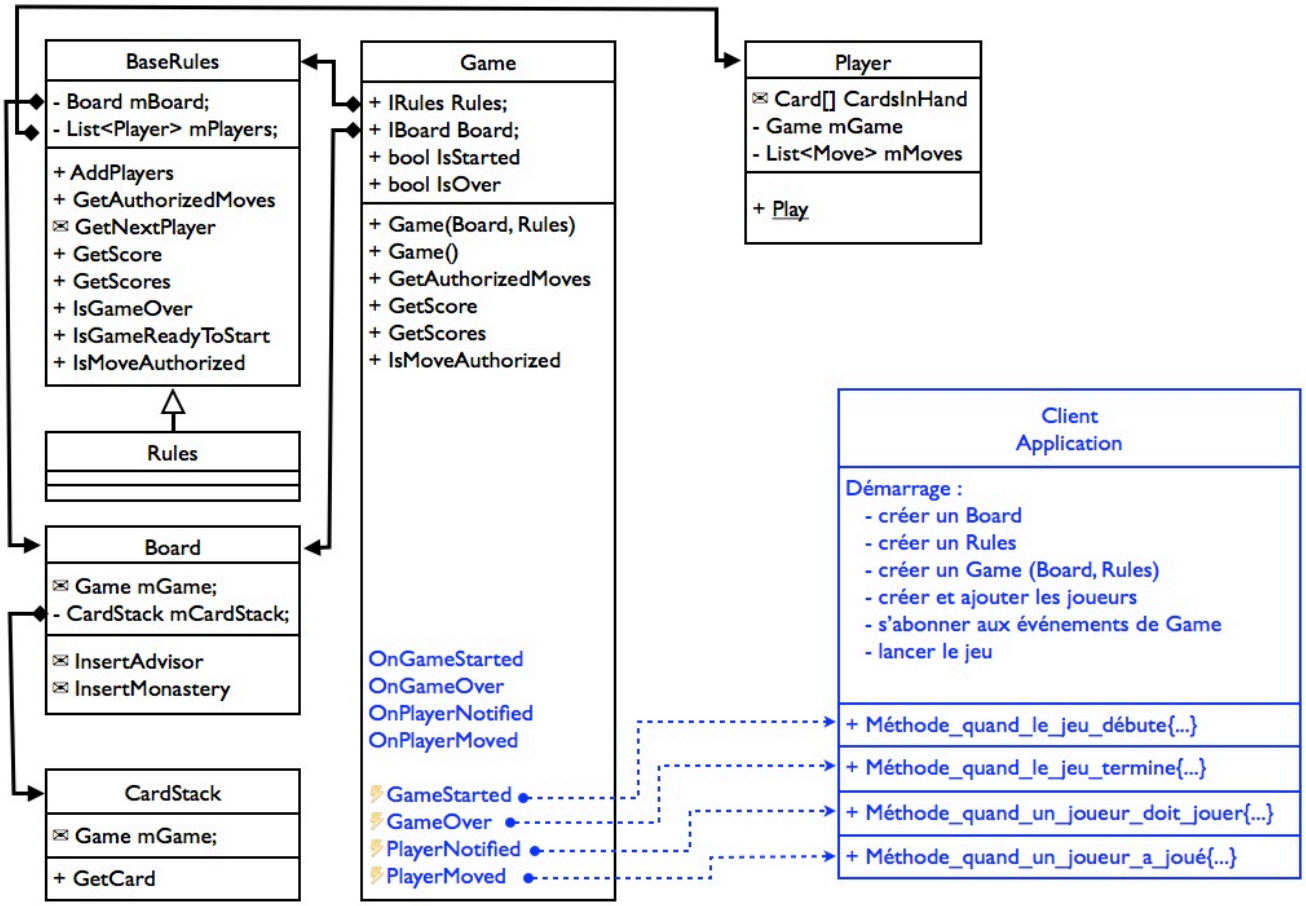
- permettre le dialogue entre le jeu (classe `Game`) et les joueurs (classes filles de `Player`),
- permettre le déroulement du jeu,
- offrir une API aux différents clients (interfaces graphiques notamment).

Cette API est déjà en grandes parties constitutée : `IGame` contient de nombreuses propriétés et méthodes publiques qui permettent à un client d'obtenir des informations quand il le souhaite. Par exemple, si le client veut connaître les scores à un moment donné, il lui suffit d'appeler la méthode `GetScores` de `IGame` et il récupère cette information.

En revanche, en ce qui concerne l'information plus «dynamique», afin que le client reste informé des modifications, la méthode la plus efficace est celle consistant à utiliser des événements auxquels le client s'abonnera. C'est pour cette raison que nous allons décrire un ensemble d'événements permettant de renseigner les utilisateurs externes de `IGame` sur l'évolution du jeu et notamment :

- le démarrage du jeu,
- le fait qu'un joueur soit invité à jouer,
- le fait qu'un joueur ait joué un coup,
- la fin du jeu.

Le client, avant de lancer le jeu, devra donc créer des méthodes et s'abonner aux événements qui l'intéressent et qui appelleront ces méthodes.



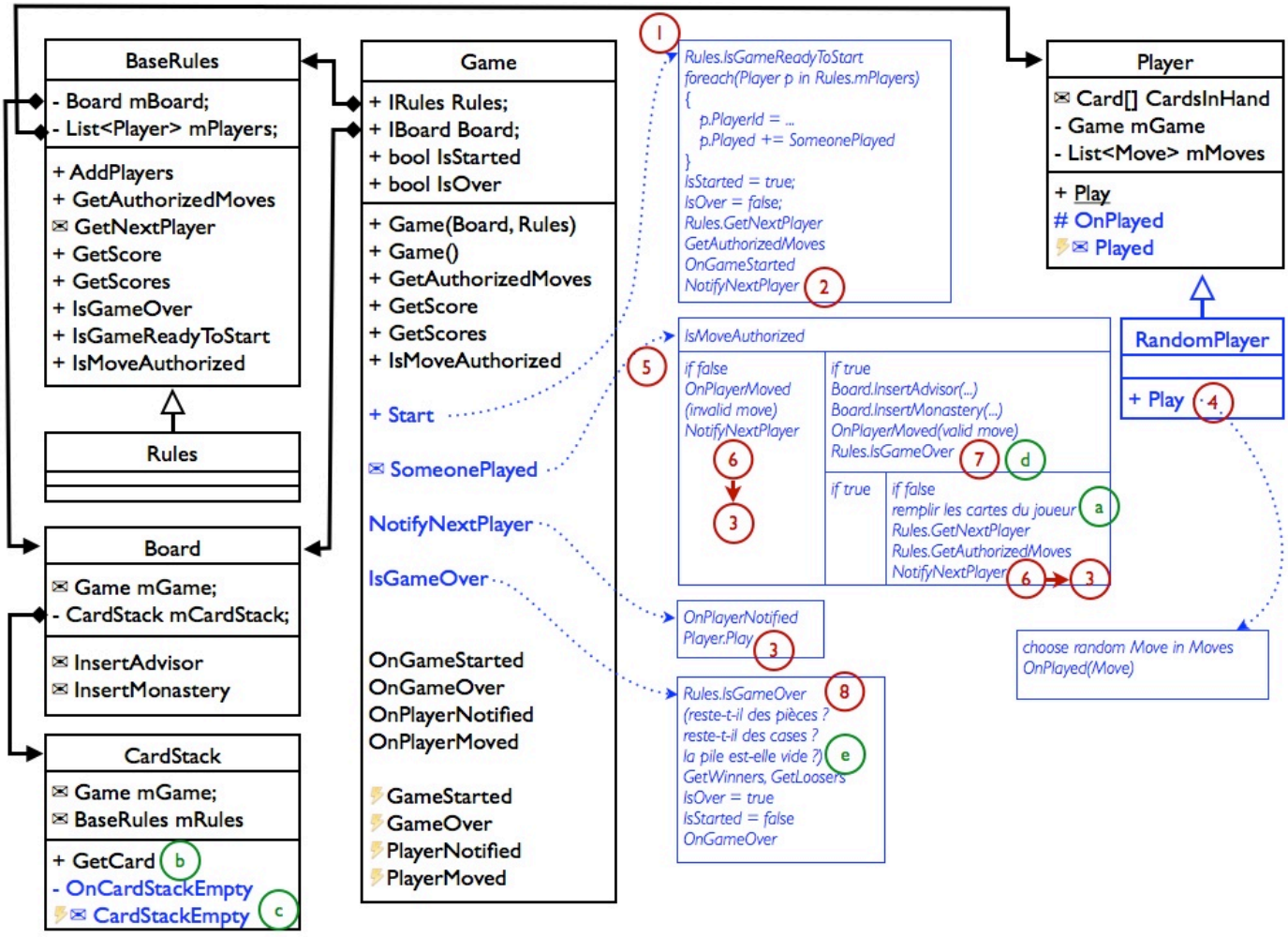
La création du client et les abonnements seront fait dans la partie 7. Tout d'abord, nous allons implémenter les événements publiques `GameStarted`, `GameOver`, `PlayerNotified` et `PlayerMoved`.

Dans un second temps, nous allons écrire les méthodes permettant le bon déroulement du jeu. À l'intérieur de ces méthodes, nous lancerons les événements en fonction de l'évolution et renseignerons ainsi le monde entier sur le déroulement du jeu.

Quatre méthodes principales sont à écrire :

- `Start` : qui démarre le jeu,
- `SomeonePlayed` : qui s'exécute à chaque fois qu'un joueur essaye de jouer un coup,
- `IsGameOver` : qui vérifie si le jeu est terminé ou non,
- `NotifyNextPlayer` : qui renseigne le joueur suivant que c'est à son tour de jouer.

Afin que la classe `Game` ne soit pas en perpétuelle attente des coups de joueurs, nous utiliserons un événement qui sera déclenché par la classe `Player` à chaque fois qu'un joueur veut faire un coup. À la réception de cet événement à travers la méthode `SomeonePlayed`, la classe `Game` pourra exécuter la suite du jeu (vérifier que le coup est valide, faire les insertions sur le plateau de jeu, informer le joueur suivant, vérifier que la partie n'est pas terminée...).



Le déroulement du jeu est le suivant :

1 `Start` est appelée :

elle doit d'abord vérifier, auprès de `Rules`, que les conditions initiales avant démarrage du jeu sont vérifiées : est-ce qu'un plateau de jeu existe, est-ce que le nombre de joueurs est valide...

elle permet ensuite à `Game` de s'abonner aux événements `Played` de chaque joueur (ces événements sont lancés par le joueur, chaque fois qu'un joueur souhaite faire un coup)

elle demande à `Rules` qui est le joueur suivant

elle demande à `Rules` quels sont alors les coups autorisés

puis renseigne le monde entier que le jeu démarre à l'aide de l'événement `GameStarted`

enfin, elle renseigne le premier joueur que c'est son tour de jouer à l'aide de la méthode `NotifyNextPlayer`

2 `NotifyNextPlayer` se contente de lancer un événement indiquant au monde entier qu'un joueur vient d'être informé que c'est son tour de jouer et qui est ce joueur. Elle demande ensuite à ce joueur de jouer en appelant sa méthode `Play` 3

4 La méthode `Play` de la classe `Player` a pour but de choisir un coup et d'informer les autres classes que cette instance de `Player` souhaite faire ce coup. La méthode `Play` est abstraite dans la classe `Player`. Prenons par exemple le cas de la classe `RandomPlayer` qui héritera de `Player`. Celle-ci choisit par exemple, un coup aléatoire dans la liste des coups autorisés et lance l'événement `Played` pour informer les autres classes qu'elle veut faire ce coup.

5 `Game` s'étant abonnée à l'événement `Played` de chacun des joueurs à travers la méthode `SomeonePlayed`, reçoit alors l'événement et cette méthode `SomeonePlayed` est donc exécutée.

Celle-ci vérifie alors auprès de `Rules` si le coup proposé est valide.

S'il ne l'est pas, elle redemande au même joueur de jouer à nouveau.

Si le coup est valide, elle réalise les insertions des pièces auprès de `Board` et informe le reste du monde qu'un coup a été joué en utilisant l'événement `PlayerMoved`.

Elle teste alors (auprès de `Rules`) si la partie est terminée.

Si elle ne l'est pas, elle remplit la main du joueur qui vient de jouer en redemandant des cartes à `Board.CardStack`.

Elle demande ensuite à `Rules` qui est le prochain joueur, lui demande quels sont les nouveaux coups autorisés, et appelle renseigne le nouveau joueur. (6)

Le jeu continue en cycle de cette manière, jusqu'à ce que la partie soit terminée. Dans ce cas, dans la méthode `SomeonePlayed`, le teste auprès de `Rules` de la fin de partie (7) arrête la partie et lance l'événement `GameOver` (8).

Il faut toutefois noter un dernier cas un peu particulier de fin de partie qui est celui de la pile vide. À chaque fois qu'un joueur doit remplir sa main (a), la pile de cartes se vide (b). Si celle-ci se retrouve complètement vide, nous devons alors renseigner le jeu que la partie est terminée (seulement si nous utilisons ces règles du jeu !!!). Cette règle pouvant varier, ce n'est pas à `CardStack` ou à `Game` de décider que la partie est terminée, mais à `Rules`. Pour cela, nous allons envoyer un événement `CardStackEmpty` au reste du monde depuis `CardStack` (c). Si une classe de règles du jeu est intéressée par cette information (ce qui est le cas pour nous), alors elle doit s'abonner à cet événement. Lorsqu'elle le reçoit, elle peut ainsi le traiter et en déduire la marche à suivre. Dans notre cas, elle indiquera par un drapeau la fin de partie, afin que le prochain appel de `IsGameOver` rende `true` (d) (e). On pourra alors, dans `SomeonePlayed`, appeler une deuxième fois `IsGameOver` après avoir rempli la main du joueur pour éviter de laisser un autre joueur jouer.

DÉROULEMENT DU JEU

La répartition des tâches est celle proposée dans le schéma de la page suivante. La suite du document propose l'ordre de la réalisation de ces tâches. Pour chaque événement, vous devrez respecter le pattern standard et choisir les arguments d'événement.

UN JOUEUR SOUHAITE JOUER UN COUP

event

Ajout d'un événement Played

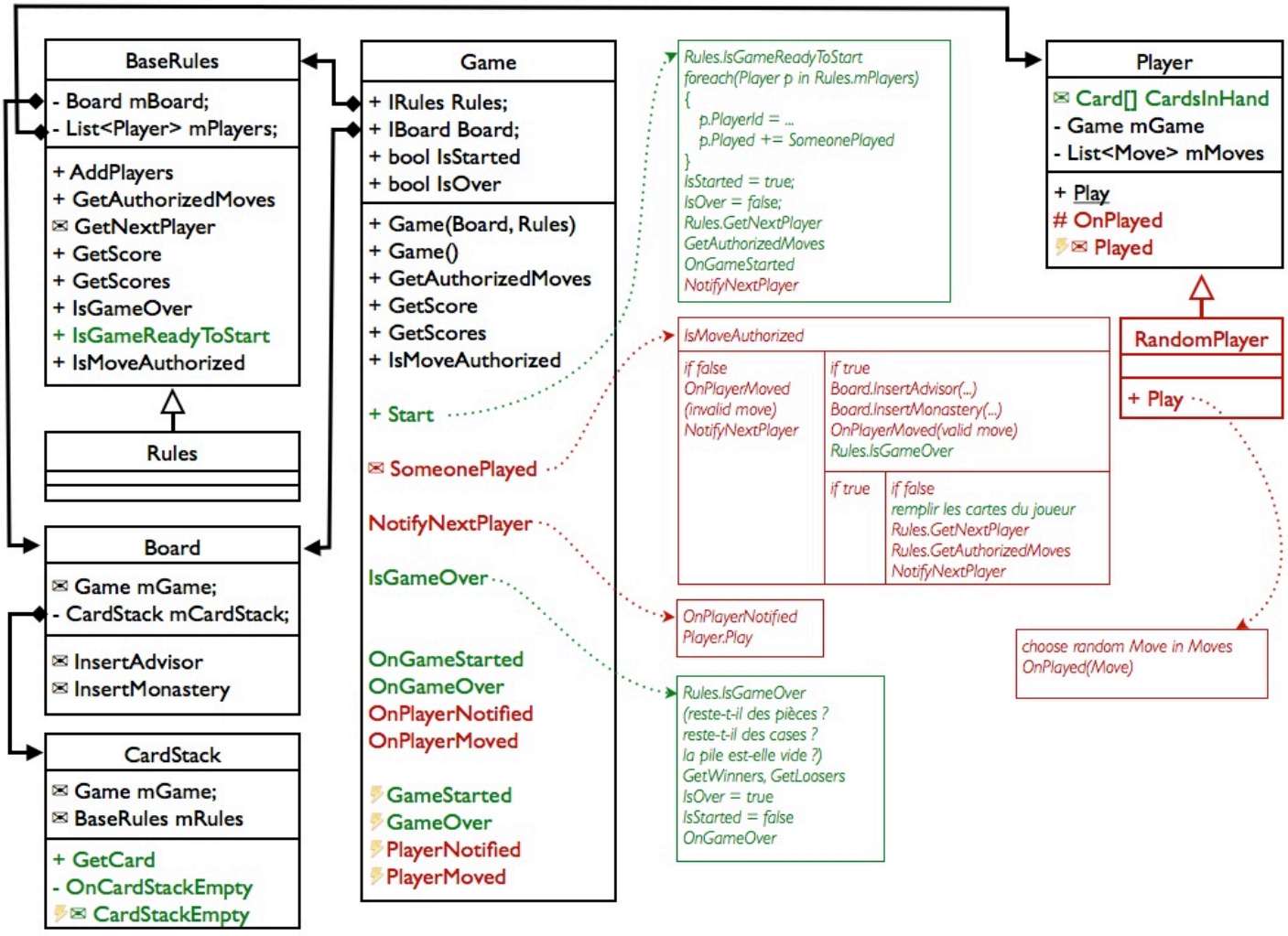
Dans la classe `Player`, ajoutez un événement `Played` en respectant le pattern standard.

event

RandomPlayer et implémentation de Play

Créez la classe `RandomPlayer` héritant de `Player` et écrivant la méthode `Play`. Lancez l'événement `Played` dans cette méthode en

choisissant aléatoirement un coup dans la liste de coups autorisés `Moves`.



propriétés

Amélioration de GetAuthorizedMoves

`GetAuthorizedMoves` est une méthode assez lente. Faites en sorte que l'appel de `GetAuthorizedMoves` dans `Game`, stocke le résultat dans une propriété interne. Ensuite, faites en sorte que la propriété `Moves` de `Player` ne fasse que récupérer ce résultat à travers la propriété de `Game`. Ceci devrait grandement améliorer les performances.

FIN DE PARTIE

event

Ajout d'un événement GameOver

Ajoutez un événement `GameOver` à la classe `Game`. Écrivez une méthode `IsGameOver` qui vérifiera auprès de `Rules` que la partie est terminée et lancera l'événement si elle l'est.

event

Ajout d'un événement CardStackEmpty

Ajoutez un événement `CardStackEmpty` à la classe `CardStack`. Notre classe `Rules` (et pas `BaseRules`) sera intéressée par cet événement. Abonnez-la et modifiez la méthode `IsGameOver` de façon à ce que la pile vide indique que la partie est terminée. Modifiez la méthode `GetCard` de `CardStack` pour qu'elle lance l'événement `CardStackEmpty` si elle est vide.

RENSEIGNER UN JOUEUR QU'IL DOIT JOUER

event

Ajout d'un événement PlayerNotified

Ajoutez l'événement `PlayerNotified` et la méthode `NotifyNextPlayer`.

DÉBUT DE PARTIE

event

Ajout d'un événement GameStarted

Ajoutez un événement `GameStarted` à la classe `Game`. Écrivez une méthode `Start` qui lancera le jeu après avoir :

- vérifié auprès de `Rules` que la partie peut commencer (le nombre de joueurs est valide...),
- abonné la classe `Game` à l'événement `Played` de chaque joueur,
- demandé à `Rules` qui est le premier joueur,
- demandé à `Rules` la liste des coups autorisés.

Lancez l'événement `GameStarted` et appelez la méthode `NotifyNextPlayer`.

CONTINUER LE JEU APRÈS LE COUP D'UN JOUEUR

event

Ajout d'un événement PlayerMoved

Ajoutez l'événement `PlayerMoved` qui sera lancé à chaque fois qu'un coup (valide ou non, à rajouter dans les arguments) a été fait.

méthode
event

Écrire la méthode SomeonePlayed

Cette méthode devra :

- vérifier que le coup reçu est autorisé,
- s'il ne l'est pas, renseigner le même joueur qu'il doit rejouer,
- s'il l'est :faire les insertions sur le plateau de jeu,
- lancer l'événement précisant qu'une insertion a été faite,
- vérifier que le jeu n'est pas terminé,
- s'il ne l'est pas, remplir la main du joueur,
- vérifier à nouveau que le jeu n'est pas terminé (pile vide...),
- s'il ne l'est pas, demander qui est le prochain joueur à `Rules`,
- demander quels sont les coups autorisés,
- renseigner ce joueur qu'il doit jouer.

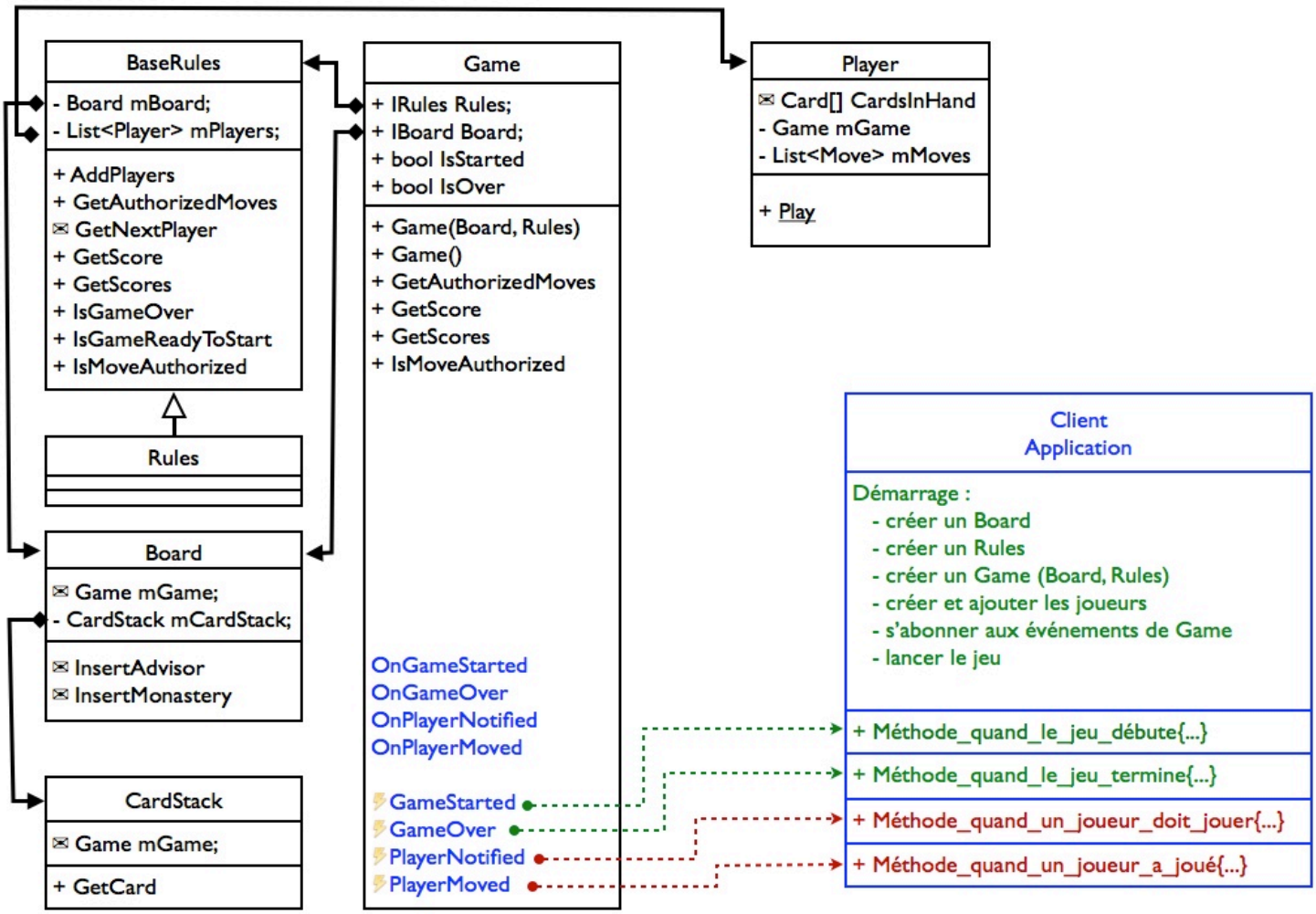
Partie 7 : jeu… en console

délégués, événements, application Console

INTRODUCTION

Le but de cette partie est de créer une application Console permettant de vérifier que le jeu se déroule normalement.

Relisez l'introduction de la partie 6 concernant le développement de l'application client, puis réalisez les tâches ci-dessous.



APPLICATION CLIENTE - DÉBUT ET FIN DE PARTIE

Console

Création d'une application cliente et lancement du jeu

Créez une application Console, dans laquelle vous créerez le jeu et l'initialiserez.

event
abonnement

Abonnement à l'événement GameStarted

Créez une méthode dans cette application qui affichera un message lorsque le jeu commence. Abonnez cette méthode à l'événement `GameStarted` de `Game`.

event
abonnement

Abonnement à l'événement GameOver

Créez une méthode dans cette application qui affichera un message (vainqueurs, perdants, scores...) lorsque le jeu se termine. Abonnez cette méthode à l'événement `GameOver` de `Game`.

APPLICATION CLIENTE - DÉBUT ET FIN DE PARTIE

event
abonnement

Abonnement à l'événement PlayerNotified

Créez une méthode dans cette application qui affichera un message (joueur renseigné), lorsqu'un joueur est renseigné qu'il doit jouer. Abonnez cette méthode à l'événement `PlayerNotified` de `Game`.

event
abonnement

Abonnement à l'événement PlayerMoved

Créez une méthode dans cette application qui affichera un message (état du tableau et coup réalisé) lorsque le jeu se termine. Abonnez cette méthode à l'événement `PlayerMoved` de `Game`.

DEBUG

Tout doit marcher. Bon debug.

Semaine 6

XML, DTD

OBJECTIFS

Les objectifs de la sixième semaine, du point de vue de l'application finale, sont :

- de décrire les règles du jeu dans un format XML.

Les objectifs pédagogiques sont :

- l'écriture de fichiers XML mixtes,
- l'utilisation et la modification de DTD,
- la validation de fichiers XML selon des DTD à l'aide de validateurs publiques.

PRÉPARATION

Relisez le cours sur XML et les DTD.

Partie 8 : les règles du jeu

XML, DTD

OBJECTIFS

L'objectif de cette partie est d'écrire un document XML contenant la sémantique des règles du jeu Web Of Power.

Les objectifs pédagogiques sont :

- l'écriture d'un fichier XML bien formé,
- l'utilisation d'une DTD externe pour l'écriture d'un document XML valide,
- la modification d'une DTD interne.

PRÉPARATION

Lisez l'énoncé ci-dessous et préparez un arbre représentant les éléments et les attributs du fichier XML que vous souhaitez réaliser (première question). Puis réalisez un autre arbre correspondant à la DTD externe fournie. Enfin, écrivez un troisième arbre correspondant à la DTD que vous devez modifier.

PRATIQUE

document XML

Écriture des règles du jeu dans un document XML

Relisez les règles du jeu (classiques pour le groupe rouge et la variante pour le groupe vert) et analysez-les. Écrivez un document XML contenant l'information liée à ces règles.

Rappel : un document XML ne contient pas d'information de mise en forme mais de la sémantique. Il vous faudra donc étudier attentivement l'information contenue dans la règle du jeu avant de vous lancer dans l'écriture du document XML.

Bonus : écrivez un document similaire pour un autre jeu (par exemple, le jeu de dames).

DTD externe
document XML

Utilisation d'une DTD externe

Vous trouverez une DTD pour les règles du jeu à l'URL suivant http://marc.chevaldonne.free.fr/ens_rech/Csharp_XML_GI_2010_2011_files/Rules.dtd.

Étudiez cette DTD et écrivez un (des) document(s) XML pour les règles du jeu Web Of Power (classiques pour le groupe rouge et la variante pour le groupe vert) (et du jeu des dames (bonus)). Le(s) document(s) devra (ont) bien entendu être valide(s). Pour cela, utilisez le validator gratuit à l'adresse suivante : http://www.w3schools.com/xml/xml_validator.asp, section «Validate your XML against a DTD» (uniquement avec Internet Explorer).

DTD interne
document XML

Modification d'une DTD en interne

La DTD précédente présente quelques inconvénients. Écrivez une DTD interne inspirée de celle-ci afin de permettre :

- pour un matériel, d'écrire son type puis sa quantité, ou sa quantité puis son type,
- pour un matériel, de ne pas être obligé d'écrire la quantité,
- pour un jeu, de ne pas obligatoirement indiquer l'âge minimum,
- pour un jeu, que le temps soit 45 min par défaut,
- pour un jeu, d'indiquer le(s) nom(s),
- pour un jeu, de choisir un type parmi : cartes, plateau, rôle,
- de rajouter des commentaires,
- ce que vous voulez.

Modifiez vos documents XML en conséquence pour vérifier que votre DTD est correcte.

Semaine 7

XML, espaces de noms, XML Schema

OBJECTIFS

Les objectifs de la septième semaine, du point de vue de l'application finale, sont :

- de décrire les règles du jeu dans un format XML,
- de décrire une partie (terminée ou en cours) dans un fichier XML,
- de décrire les résultats des parties terminées sur une machine dans un fichier XML.

Les objectifs pédagogiques sont :

- l'écriture de fichiers XML mixtes,
- l'écriture de fichiers XML de données,
- l'utilisation d'espaces de noms dans les schémas XML et les instances XML,
- l'utilisation et la modification de schémas XML,
- la validation de fichiers XML selon des schémas à l'aide de validateurs publiques (Xercès) et privés (VisualStudio2010).

PRÉPARATION

Relisez le cours sur XML, les espaces de noms et les schémas XML.

Partie 9 : un schéma pour les règles du jeu

XML, XML Schema, Xerces

OBJECTIFS

L'objectif de cette partie est d'écrire un document XML schéma valide à partir d'un XML Schema donné.

Les objectifs pédagogiques sont :

- l'écriture d'un fichier XML bien formé et schéma valide,
- l'utilisation d'un schéma-validateur Xerces.

PRÉPARATION

Lisez l'énoncé ci-dessous et préparez un arbre représentant les éléments et les attributs que doivent contenir les document XML validant le schéma fourni.

PRATIQUE

Xerces

Utilisation de Xerces

Téléchargez le zip de Xerces à l'adresse suivante : http://xerces.apache.org/xerces-c/download.cgi (choisissez le zip : `xerces-c-3.1.1-x86-windows-vc-10.0.zip`).

Dézippez l'archive dans votre dossier `C:/Travail`.

Ouvrez une fenêtre de commande et écrivez comme première ligne :

```
set path=C:/Travail/xerces-c-3.1.1-x86-windows-vc-10.0/bin
```

Afin de vérifier que tout fonctionne correctement, toujours dans la fenêtre de commande, utilisez `SAXPrint` sur le fichier exemple de la manière suivante :

```
SAXPrint -f -n C:/Travail/xerces-c-3.1.1-x86-windows-vc-10.0/samples/data/personal-schema.xml
```

Si le fichier est écrit dans la fenêtre de commande sans erreur, `SAXPrint` est configuré correctement et permet de schéma valider.

XML Schema
document XML
espace de noms

Utilisation d'un schéma XML

Vous trouverez un schéma XML de règles du jeu à l'URL suivant : http://marc.chevaldonne.free.fr/ens_rech/Csharp_XML_GI_2010_2011_files/rules_TP7a.xsd.

Étudiez ce schéma et écrivez des documents XML valides pour des règles du jeu de Web Of Power (classiques pour le groupe rouge et la variante pour le groupe vert). Vous pouvez aussi utiliser ce schéma pour écrire les règles du jeu de dames par exemple. Les documents devront bien entendu être valides. Pour cela, utilisez SAXPrint que vous avez utilisé dans la question précédente, avec les options `-s -n`. Faites bien attention d'utiliser les espaces de noms et le schéma.

XML Schema
document XML
espace de noms
xs:element
mixed
xs:complexType
xs:attribute
use
xs:any
xs:enumeration

Modification d'un schéma XML

Rajoutez un élément `description`, qui contiendra une description du jeu à travers deux sous-éléments (`introduction`, `history`). Le sous-élément `history` sera mixte et pourra contenir des éléments `author`, `origin`, `creation_date` et `name`.

Rajoutez à `material` la possibilité d'avoir des sous-éléments `item`, contenant eux-mêmes l'attribut obligatoire `id`, entier positif, et pouvant éventuellement contenir l'attribut `name` et n'importe quel contenu. On pourra ainsi dans l'instance, faire un élément `material` avec l'attribut `type` égal à `piece`, contenant différents `item`, un pour chaque type de pièce. Ces items pourront contenir un sous-élément `nb`, donnant le nombre de pièces de ce type dans l'échiquier. Imaginez quelque chose de similaire pour l'élément `material` ayant un attribut type égal à `card` et ses sous-éléments.

Rajoutez à l'attribut `type` de `rule` la valeur possible `objectives`. Modifiez `rule` pour qu'il ne dépende pas d'un jeu en particulier, i.e. modifiez `rule` pour qu'il ne contienne plus de sous-éléments `rules:place`, `rules:chip`, etc... mais que des sous-éléments `material` avec un attribut `id` contenant la valeur d'un `id` d'un élément `materials/material/item`.

Pour s'assurer que l'`id` dans `rule/material` correspond bien à un `id` de `materials/material/item` et que ce dernier est bien unique, utilisez des key/keyref.

Partie 10 : un schéma pour le déroulement d'une partie

XML, XML Schema

OBJECTIFS

On souhaite enregistrer dans des fichiers XML, le déroulement de chaque partie, pour pouvoir ensuite : sauvegarder les parties, reprendre une partie ou encore les rejouer à l'écran. On propose donc d'enregistrer les informations suivantes dans ces fichiers XML :

- les règles du jeu et le plateau de jeu utilisés,
- les noms, types, fichiers, classes[4] des joueurs ayant joué la partie enregistrée dans le fichier,
- chaque coup joué, en précisant l'heure du coup, le joueur réalisant ce coup, la liste des monastères introduits, la liste des conseillers introduits, les cartes jouées, les cartes tirées de la pioche pour remplir la main après le coup.

L'objectif de cette partie 10 est d'écrire un schéma XML stockant toutes ces informations (et une instance XML [bidon, juste pour tester] de ce schéma).

Les objectifs pédagogiques sont :

- l'écriture d'un schéma XML,
- l'écriture d'un document XML schéma valide.

PRÉPARATION

Lisez l'énoncé ci-dessous et préparez un arbre qui permettra de stocker les informations liées aux parties réalisées.

Pensez à écrire une instance de ce schéma que vous mettrez à jour au fur et à mesure de l'avancement pour vous garantir que votre schéma correspond bien aux objectifs.

---

[4] Un joueur «intelligence artificielle» sera représenté dans le jeu par une classe dérivant de Player et provenant d'un assemblage (dll) externe. Nous pourrons donc le représenter de manière unique dans le fichier XML avec les paramètres suivants : classe + assemblage (dll). S'il s'agit d'un joueur humain, ces paramètres ne seront pas considérés.

namespace

Préparation du schéma

Préparez un fichier xsd pour votre schéma et choisissez un espace de noms et un préfixe. Qualifiez les éléments et les attributs. Ajoutez un attribut `date` obligatoire, correspondant à la date et l'heure de l'enregistrement de la partie, à l'élément racine.

XML Schema

Informations sur les règles du jeu et le plateau de jeu

Les règles du jeu et le plateau de jeu seront représentés de la même manière dans le fichier XML, c'est-à-dire par : la classe les représentant, l'assemblage dans lequel elles se trouvent et un nom. Par exemple, les règles du jeu classiques seront représentés par {nom : «`Classic Rules`» ; classe : «`ClassicRules`» ; assemblage : «`giWebOfPowerCore.dll`»} et le plateau européen par {nom : «`Europe`» ; classe : «`EuropeanBoard`» ; assemblage : «`giWebOfPowerCore.dll`»}. Si d'autres règles ou d'autres plateaux sont créées dans d'autres assemblages, nous pourrons ainsi les référencer de cette manière.

Pour cela, nous allons créer des types globaux qui pourront être réutilisés aussi bien par le plateau de jeu que par les règles.

simpleType
pattern (facettes)

Créez un type simple `dllType`, permettant d'imposer à une chaîne de caractères de se finir par la chaîne «`.dll`».

complexType
attribute

Créez une type complexe globale `classType` contenant :

- un attribut `name` de type chaîne de caractères,
- un attribut `dll` de type `dllType`,
- un attribut `class` de type chaîne de caractères.

Vous pouvez rendre les attributs `dll` et `class` optionnels et l'attribut `name` obligatoire. De cette façon, nous ne serons pas obligés de renseigner la classe et la dll si les règles et/ou le plateau de jeu font partie de l'assemblage principal (cas le plus fréquent).

element

Modifiez votre schéma pour que les instances XML possèdent un et un seul plateau de jeu, et une et une seule règle du jeu.

Informations sur les joueurs 1/2

Nous allons maintenant créer la partie permettant de stocker les joueurs. Pour cela, nous allons créer différents types globaux complexes. Le joueur doit contenir les informations suivantes :

- son nom, sa classe, son assemblage, son type IA, dans le cas d'un joueur IA
- son nom dans le cas d'un joueur humain,
- l'ordre de jeu (s'il est le 1er, le 2nd, le 3ième... joueur a joué),
- la liste des cartes qu'il a en main au début du jeu (afin de pouvoir recréer le même état dans le cas d'une reprise ou d'un replay).

enumeration

Type de joueur

Créez un type simple qui permettra de choisir entre deux types possibles pour les joueurs : «`Human Player`» ou «`AI Player`».

list
pattern

Liste de cartes

Créez un type simple qui permettra de lister des cartes. Chaque carte sera représentée par une chaîne de caractères de la forme suivante : «`0/6`», «`1/5`» ou «`8/8`»...

extension

Informations sur les joueurs 2/2

Créez un type complexe `playerType` dérivant par extension de `classType`. Il pourra ainsi contenir le nom, la classe et l'assemblage du joueur.

Ajoutez à ce type :

- un attribut pour le type du type créé plus haut,
- un entier positif ou nul représentant son ordre du jeu,
- un attribut permettant de lister les cartes qu'il a en main au début du jeu, en utilisant le type créé dans la question précédente.

Ajoutez maintenant au fichier XML, la possibilité d'avoir entre 3 et 5 joueurs de ce type, dans un élément `players` par exemple.

Informations sur les coups joués 1/2

Un coup joué se compose des informations suivantes :

- l'identifiant du joueur effectuant le coup,
- le temps écoulé depuis le début du jeu au moment du coup,
- la liste des monastères placés,
- la liste des conseillers placés,
- la liste des cartes jouées,
- la liste des cartes prises de la pioche pour remplir la main.

Afin de préparer la création de ce type complexe, créez d'abord les deux types simples suivants pour les listes de monastères et de conseillers :

simpleType
restriction

- créez un type simple `cloisterList` qui sera une liste d'entiers pouvant varier entre 0 et 50 (nombre de cloîtres),
- créez un type simple `countryList` qui sera une liste d'entiers pouvant varier entre 0 et 8 (nombre de pays).

Note : limiter le nombre de cloîtres et de pays n'est pas très intéressant du point de vue de l'application, car il ne permet pas de s'adapter facilement à différents types de plateaux. Néanmoins, j'ai préféré vous demander de limiter cette liste dans un but pédagogique, afin de manipuler plus d'éléments et d'attributs dans les schémas XML.

Informations sur les coups joués 2/2

complexType
simpleType
restriction

Créez maintenant le type complexe `moveType` contenant :
l'identifiant du joueur réalisant ce coup, c'est-à-dire un entier entre 0 et 4 inclus,

time

- le temps écoulé depuis le début de la partie,
- la liste des monastères placés (de type `cloisterList`),
- la liste des conseillers placés (de type `countryList`),
- la liste des cartes jouées (de type `cardList`),
- la liste des cartes tirées de la pioche pour remplir la main après le coup (de type `cardList`).

Ajoutez maintenant au schéma XML une balise `moves` contenant une séquence de coups de ce type (entre 0 et beaucoup...).

abstract

Amélioration n°1/2

Améliorez le schéma en créant deux types de joueurs : `humanPlayerType` et `aiPlayerType`. Pour cela, rendez le type `playerType` abstrait et utilisez l'héritage pour créer les types `humanPlayerType` et `aiPlayerType` à partir de `playerType` pour permettre :

- pour le type humain, de ne pas avoir à remplir la dll et la classe obligatoirement, et de lui imposer le type «`Human Player`»,
- pour le type ai, d'obliger à remplir la dll et la classe, et de lui fixer le type «`AI Player`».

key
keyref

Amélioration n°2/2

À l'aide de `key/keyref` :

- rendez le paramètre «ordre dans le jeu» d'un joueur unique dans le fichier XML,
- assurez-vous que dans un coup joué, l'identifiant du joueur fait référence à un «ordre dans le jeu» d'un joueur.

document XML

Instance de schéma

Créez une instance de ce schéma pour vérifier qu'il décrit les documents que vous souhaitez écrire.

Partie 11 : un schéma pour les parties réalisées

XML, XML Schema

OBJECTIFS

On souhaite enregistrer dans des fichiers XML, toutes les parties de Web Of Power réalisées sur une même machine, pour permettre le calcul de statistiques sur les joueurs (meilleurs scores, profil d'un joueur...). On propose donc d'enregistrer les informations suivantes dans le fichier XML :

- les noms, types, fichiers, classes[5] et points[6] des joueurs ayant effectué des parties sur cette machine,
- les parties réalisées, ainsi que le(s) vainqueur(s), le(s) perdant(s), le type de victoire, la date et l'heure de la partie, les règles du jeu utilisées dans la partie, le plateau de jeu utilisé dans la partie.

L'objectif de cette partie 11 est d'écrire un schéma XML stockant toutes ces informations (et une instance XML de ce schéma).

Les objectifs pédagogiques sont :

- l'écriture d'un schéma XML,
- l'écriture d'un document XML schéma valide.

PRÉPARATION

Lisez l'énoncé ci-dessous et préparez un arbre qui permettra de stocker les informations liées aux parties réalisées.

---

[5] Un joueur «intelligence artificielle» sera représenté dans le jeu par une classe dérivant de Player et provenant d'un assemblage (dll) externe. Nous pourrons donc le représenter de manière unique dans le fichier XML avec les paramètres suivants : classe + assemblage (dll). S'il s'agit d'un joueur humain, ces paramètres ne seront pas considérés.

[6] points qui serviront à faire un classement. Nous déterminerons le mode de calcul de ces points dans une partie future

namespace

Préparation du schéma

Préparez un fichier xsd pour votre schéma et choisissez un espace de noms et un préfixe. Qualifiez les éléments et les attributs.

XML Schema

Informations sur les règles du jeu et le plateau de jeu

Les règles du jeu et le plateau de jeu seront représentés de la même manière dans le fichier XML, c'està-dire par : la classe les représentant, l'assemblage dans lequel elles se trouvent et un nom. Par exemple, les règles du jeu classiques seront représentés par {nom : «`Classic Rules`» ; classe : «`ClassicRules`» ; assemblage : «`giWebOfPowerCore.dll`»} et le plateau européen par {nom : «`Europe`» ; classe : «`EuropeanBoard`» ; assemblage : «`giWebOfPowerCore.dll`»}. Si d'autres règles ou d'autres plateaux sont créées dans d'autres assemblages, nous pourrons ainsi les référencer de cette manière.

Pour cela, nous allons créer des types globaux qui pourront être réutilisés aussi bien par le plateau de jeu que par les règles.

simpleType
pattern (facettes)

Créez un type simple `dllType`, permettant d'imposer à une chaîne de caractères de se finir par la chaîne «`.dll`».

attributeGroup
attribute

Créez un groupe d'attributs `classAttributeGroup` contenant :

- un attribut `name` de type chaîne de caractères,
- un attribut `dll` de type `dllType`,
- un attribut `class` de type chaîne de caractères.

Vous pouvez rendre les attributs `dll` et `class` optionnels et l'attribut `name` obligatoire. De cette façon, nous ne serons pas obligés de renseigner la classe et la dll si les règles et/ou le plateau de jeu font partie de l'assemblage principal (cas le plus fréquent).

Informations sur les joueurs 1/3

Nous allons maintenant créer la partie permettant de stocker les joueurs. Pour cela, nous allons créer différents types globaux complexes. Le joueur doit contenir les informations suivantes :

- son nom, sa classe, son assemblage, son type IA, dans le cas d'un joueur IA
- son nom dans le cas d'un joueur humain.

enumeration

Informations sur les joueurs 2/3

Créez un type simple qui permettra de choisir entre deux types possibles pour les joueurs : «`Human Player`» ou «`AI Player`».

extension

Informations sur les joueurs 3/3

Créez un nouveau groupe d'attributs contenant le groupe d'attributs `classAttributeGroup` créé précédemment et un nouvel attribut du type créé dans la question précédente. Ce nouveau groupe d'attributs qu'on pourra appelé `playerAttributeGroup` pourra ainsi contenir le nom, le type, la classe et l'assemblage du joueur.

Ajoutez à ce type des attributs :

- représentant le nombre de points obtenus dans les parties à 3 joueurs,
- représentant le nombre de points obtenus dans les parties à 4 joueurs,
- représentant le nombre de points obtenus dans les parties à 5 joueurs.

Ajoutez maintenant au fichier XML, la possibilité d'avoir plusieurs joueurs de ce type, dans un élément `players` par exemple.

XML Schema
attributeGroup
enumeration
xs:date

Informations sur les parties

Créez la partie du schéma permettant de stocker les informations sur les parties, et en particulier, pour chaque partie :

- les règles du jeu utilisées (en utilisant le `classAttributeGroup`),
- le plateau de jeu utilisé (en utilisant le `classAttributeGroup`),
- le ou les vainqueur(s) (utilisez le `playerAttributeGroup` de la question précédente) et imposez le nombre entre 1 et 5,
- le(s) perdant(s) (utilisez le `playerAttributeGroup`) ; imposez le nombre entre 0 et 4,
- la date et l'heure de la victoire (`xs:date`).

Modifiez l'instance précédente pour vérifier que le schéma correspond bien à ce que vous souhaitiez.

abstract

Amélioration n°1/2

Améliorez le schéma en créant deux types de joueurs : `humanPlayerType` et `aiPlayerType`. Pour cela, rendez le type `playerType` abstrait et utilisez l'héritage pour créer les types `humanPlayerType` et `aiPlayerType` à partir de `playerType` pour permettre :

- pour le type humain, de ne pas avoir à remplir la dll et la classe obligatoirement, et de lui imposer le type «`Human Player`»,
- pour le type ai, d'obliger à remplir la dll et la classe, et de lui fixer le type «`AI Player`».

key
keyref

Amélioration n°2/2

À l'aide de `key/keyref` :

- rendez les joueurs sous-éléments de `players` uniques en utilisant une combinaison de leur nom, leur type, le nom de la classe et le nom de l'assemblage,
- assurez-vous que dans un match, les vainqueurs et les perdants correspondent toujours à un joueur sous-élement de `players`, en utilisant cette combinaison d'unicité.

document XML

Instance de schéma

Créez une instance de ce schéma pour vérifier qu'il décrit les documents que vous souhaitez écrire.

Semaine 8

XML, C# DOM Parser

OBJECTIFS

Les objectifs de la huitième semaine, du point de vue de l'application finale, sont :

- de remplir un fichier XML contenant l'ensemble des parties de Kardinal & König effectuées sur cet ordinateur (i.e. enregistrer les données d'un match dans ce fichier à la fin de chaque match),
- de lire ce fichier XML pour le transformer en visualisation des «High Scores».

Les objectifs pédagogiques sont :

- l'utilisation d'espace de noms,
- l'écriture et l'utilisation d'un parseur DOM en C#,
- l'intégration de ce parseur au reste de l'application.

PRÉPARATION

Relisez le cours sur XML, les espaces de noms, les schémas XML et les parseurs DOM, ainsi que les exemples.

Partie 12 : un parseur des parties réalisées

XML, XML Schema, DOM

OBJECTIFS

L'objectif de cette partie est d'écrire un parseur DOM des fichiers XML contenant les parties réalisées. Un schéma XML a déjà été réalisé par l'équipe verte dans la partie 11 et il devra être utilisé. L'équipe rouge est responsable de la lecture du fichier XML. L'équipe verte est responsable de la mise à jour du fichier XML.

Les objectifs pédagogiques sont :

- l'écriture et l'utilisation d'un parseur DOM en C# pour lire et écrire des fichiers XML «schéma valides»,
- l'utilisation d'un schéma XML.

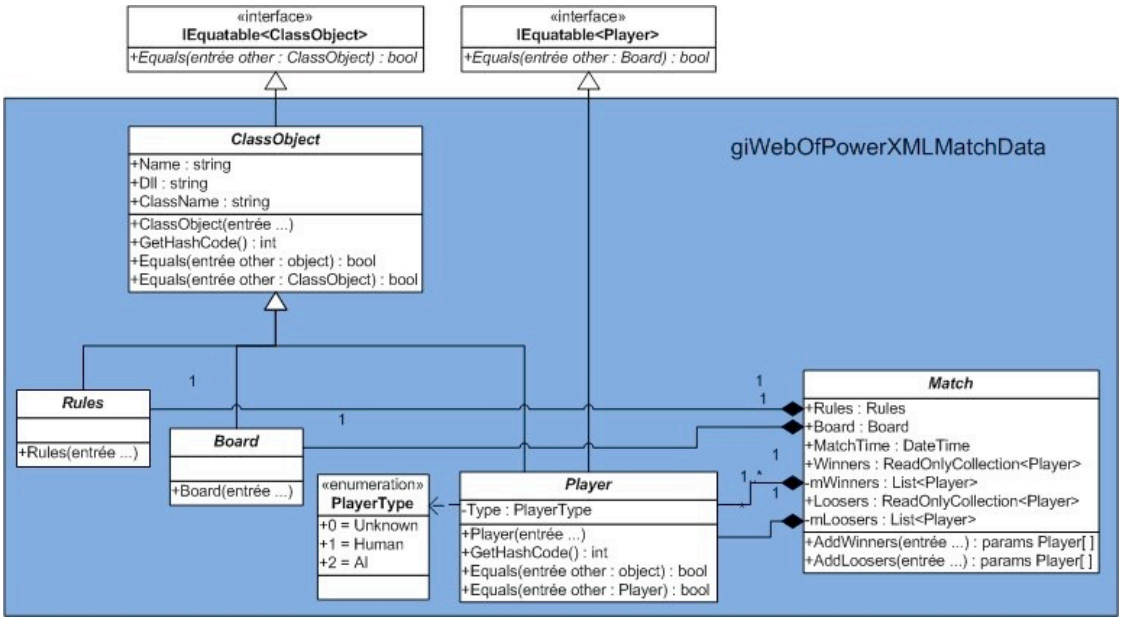
PRÉPARATION

Lisez l'énoncé ci-dessous et préparez un arbre représentant les éléments et les attributs que doivent contenir les document XML validant le schéma réalisé par l'équipe verte dans la partie 11. Préparez au brouillon un diagramme de classes du parseur.

DOCUMENTS FOURNIS

Pour ne pas vous faire perdre du temps sur l'écriture de classes C# que vous savez désormais parfaitement écrire, celles-ci vous sont fournies. Vous trouverez sur l'ENT ou sur la page web, les fichiers suivants : `ClassObject.cs`, `RulesAndBoard.cs`, `Player.cs` et `Match.cs`. Ces fichiers contiennent des classes qui permettent de stocker le contenu de l'instance XML contenant les parties réalisées, suivant le diagramme de classes ci-dessous. `Rules` permet de stocker les données propres aux règles du jeu utilisées dans un match ; `Board` permet de stocker les informations propres au plateau de jeu utilisé dans un match ;

`Player` permet de stocker les informations d'un joueur ; `Match` stocke les informations d'un match.



Vous trouverez également un fichier `MatchDataXMLTags.cs` qui contient la classe statique `XMLTags` stockant sous forme de chaîne de caractères constantes et publiques les balises que peuvent contenir les instances XML stockant les parties réalisées. Ceci est important car il permet de gagner du temps lors de modifications mineures du schéma XML et d'éviter les fautes de frappe. Si ces fichiers vous sont fournis pour le TP, il est toutefois sage de comprendre leur intérêt et la façon dont ils sont écrits : il pourrait vous être demandé d'en écrire de semblables lors de l'examen sur les parseurs.

Puisque cette partie est dépendante de la partie 11, le schéma XML de la partie 11 vous est également fourni. Vous pouvez toutefois utiliser celui réalisé par l'équipe verte si vous le désirez. Il faudra très certainement dans ce cas, faire quelques modifications à la classe `XMLTags`, pour que les noms des éléments et attributs concordent.

PRÉPARATION DU PROJET (PARTIE À FAIRE EN COMMUN PAR LES DEUX ÉQUIPES)

Projet Visual Studio

Bibliothèque de classes et préparation du projet

Ajoutez un nouveau projet à votre solution `giWebOfPowerXML` qui contiendra nos parseurs XML. Ajoutez les fichiers `ClassObject.cs`, `Rules.cs`, `Board.cs`, `Player.cs`, `Match.cs` et `MatchDataXMLTags.cs` qui vous ont été fournis.

Schéma XML

Dans votre arborescence, ajoutez les dossier `data` (au même niveau que bin et VisualStudio2010) qui contiendra les dossiers `XML` et `XSD`.

Ajoutez le schéma XML au dossier `data/XSD`. Trop c'est trop... J'ai modifié mon schéma pour qu'il ne fasse plus la différence entre les points obtenus par les monastères et les points obtenus par les conseillers. Si vous voulez utiliser votre schéma, modifiez-le pour que les éléments `winner` et `looser` dans un match stocke un attribut score au lieu de `scoreMonasteries` et `scoreAdvisors`.
Ajoutez une classe `MatchDataParser`. Ajoutez-lui un champ `mXDoc` de type `XmlDocument` et pushez le tout sur le serveur.

CHARGEMENT DU FICHIER DANS L'ARBRE DOM

arbre DOM

Chargement du fichier XML dans un DOM

Inspirez-vous des exemples de cours et écrivez une méthode `LoadXMLFile` qui chargera un fichier XML contenant les parties réalisées (cf. `XMLTags` pour son nom), vérifiera qu'il est valide par rapport à son schéma. Cette méthode lancera une exception dans le cas contraire. Appelez cette méthode dans le constructeur.
Pour vérifier le bon fonctionnement de cette classe, testez-la dans un projet de tests «ami» de votre bibliothèque avec une instance XML bidon que vous placerez dans `data/XML`.
Pushez sur le serveur (le test aussi) : l'équipe verte aura très bientôt besoin de cette méthode.

CRÉATION DU FICHIER DES PARTIES RÉALISÉES

écriture de l'arbre DOM

Création d'une instance XML depuis le parseur

Si le fichier XML des parties réalisées n'existe pas (le jeu vient d'être installé par exemple, ou un utilisateur l'a effacé), le parseur ne le trouvera pas et ne pourra pas le modifier. Il faut donc pouvoir le créer dans ce cas.
Inspirez-vous des exemples de cours et créez une méthode `CreateEmptyFile` qui va créer l'arbre DOM puis l'instance XML la plus petite possible (mais «schéma valide»), et ne contenant pour le moment aucun match et aucun joueur.
Pour cela, écrivez ensuite une méthode `Save` permettant d'enregistrer l'arbre DOM en un fichier XML et appelez cette méthode à la fin de `CreateEmptyFile`.

L'équipe rouge a déjà créé un constructeur à la classe `MatchDataParser`. Faites un fetch/merge et récupérez ce constructeur. À l'aide d'un bloc `try/catch`, arrangez-vous pour qu'une exception lors de l'appel de `LoadXMLFile` entraine l'appel de `CreateEmptyFile`. De cette manière, si le fichier XML n'existe pas ou s'il n'est pas schéma valide, un nouveau sera créé. L'équipe rouge a également dû rajouter un test très simple «ami» de votre assemblage : utilisez-le pour vérifier la bonne création de votre fichier XML en l'absence de celui-ci.

LECTURE DE L'ARBRE DOM

arbre DOM

Lecture des joueurs et des matchs dans l'arbre DOM

Ajoutez à la classe `MatchDataParser` quatre listes stockant : les règles du jeu utilisées, les plateaux de jeu utilisés, les joueurs et les parties. Utilisez pour cela les classes qui vous ont été fournies. Ajoutez deux méthodes `ReadPlayers` et `ReadMatches` (et plus selon vos besoins) qui vont remplir ces listes en parcourant l'arbre DOM. Appelez ces méthodes dans le constructeur de votre parseur.

Tests

Tests des résultats

Ajoutez deux `ReadOnlyCollections` (une pour les joueurs et une pour les matches) qui permettront de récupérer la liste des joueurs et la liste des matches.

Dans votre application de test «ami», utilisez LINQ pour afficher le score de chaque joueur et ainsi vérifier que vos méthodes fonctionnent bien.

Pushez.

ÉCRITURE DANS L'ARBRE DOM

écriture de l'arbre DOM

Ajout de nouveaux éléments dans l'arbre DOM

Rajoutez dans l'arbre DOM un match qui vient d'avoir lieu. Pour cela, vous devez ajouter les joueurs de ce match qui ne sont pas encore présents dans la liste des joueurs et ainsi que le nouveau match.

Ajoutez une méthode `AddPlayer` qui ajoute un `Player` dans l'arbre DOM (en fils de l'élément `players`). Ajoutez une méthode `AddMatch` qui ajoute un `Match` dans l'arbre DOM (en fils de l'élément `matches`). Pour cela, utilisez les classes qui vous ont été fournies et abusez de

`XMLTags`. Il est conseillé également de s'aider de plusieurs méthodes privées pour simplifier l'algo.

N'oubliez pas de sauver l'arbre DOM pour qu'il s'écrive dans le fichier XML.

Tests

Tests de l'ajout

Dans votre application de test «ami», l'équipe rouge a ajouté (ou va rajouter) l'affichage du nombre de parties de chaque joueur. Dans ce test, ajoutez un match bidon et affichez les matchs gagnés par chaque joueur avant et après votre ajout pour vérifier que vos méthodes fonctionnent bien.

Pushez.

UTILISATION DU PARSEUR

arbre DOM

Affichage des meilleurs joueurs

À l'aide de LINQ, ajoutez trois méthodes dans le projet Test permettant : d'afficher les x (passé en paramètres) meilleurs joueurs, c'est-à-dire, ceux ayant le meilleur pourcentage de victoires. Si deux joueurs sont à égalité, c'est celui qui a le plus de victoires qui est considéré comme meilleur. S'ils sont toujours à égalité, c'est celui qui a le moins de défaites.

MISE À JOUR DE L'INSTANCE XML

écriture de l'arbre DOM

Ajout d'un match après une partie

Modifiez la classe `Game` pour que le match qui se termine soit enregistré dans le fichier XML.

Pour cela, vous pourrez rajouter une propriété virtuelle `Name` à `Player` et à `Rules` (elle existe déjà pour `Board` normalement).

Après le match, vérifiez que le fichier XML a bien été modifié.

Semaine 9

XML, C# XmlReader XmlWriter

OBJECTIFS

Les objectifs de la neuvième semaine, du point de vue de l'application finale, sont :

- d'enregistrer une partie dans une fichier XML,
- de rejouer avec une partie enregistrée dans un fichier XML avec des joueurs automatiques.

Les objectifs pédagogiques sont :

- l'écriture et l'utilisation d'un parseur XmlWriter et d'un parseur XmlReader en C#,
- l'intégration de ce parseur au reste de l'application.

PRÉPARATION

Relisez le cours sur XML, les espaces de noms, les schémas XML et les parseurs XmlReader et XmlWriter, ainsi que les exemples.

Partie 13 : un parseur des parties réalisées

XML, XML Schema, XmlWriter, XmlReader

OBJECTIFS

L'objectif de cette partie est d'écrire un parseur XmlWriter pour enregistrer une partie et un parseur XmlReader pour la rejouer. Un schéma XML a déjà été réalisé par l'équipe rouge dans la partie 10 et il devra être utilisé. L'équipe verte est responsable de la lecture du fichier XML. L'équipe rouge est responsable de l'écriture du fichier XML.

Les objectifs pédagogiques sont :

- l'écriture et l'utilisation d'un parseur XmlWriter en C# pour écrire des fichiers XML «schéma valides»,
- l'écriture et l'utilisation d'un parseur XmlReader en C# pour lire des fichiers XML «schéma valides»,
- l'utilisation d'un schéma XML.

PRÉPARATION

Lisez l'énoncé ci-dessous et préparez un arbre représentant les éléments et les attributs que doivent contenir les document XML validant le schéma réalisé par l'équipe rouge dans la partie 10. Préparez au brouillon un diagramme de classes du parseur.

La solution proposée pour l'enregistrement est la suivante :

- vous allez créer une SortedList qui contiendra la date et l'heure de chaque coup effectué (ainsi que le coup lui-même),
- le coup stocké ressemblera à la structure Move (il contiendra seulement en plus, les cartes tirées par le joueur dans la pile),
- au début de la partie, vous stockerez les joueurs et les cartes reçues au démarrage du jeu,
- à chaque coup joué, vous mettrez à jour l'historique,
- lorsque la partie sera terminée, vous enregistrez l'historique dans un fichier XML à l'aide du parseur.

La réalisation de ces tâches se fera donc dans l'ordre suivant :

- création de l'historique,
- création du parseur XmlWriter,
- utilisation du parseur.

La solution proposée pour la lecture des parties est la suivante :

- vous allez écrire un parseur XmlReader offrant toutes les méthodes nécessaires pour lire les éléments du fichier XML,
- vous allez créer un nouveau type de Player : ReplayPlayer, dont la méthode Play lira le coup suivant dans le fichier XML et le jouera.

La réalisation de ces tâches se fera donc dans l'ordre suivant :

- création du parseur XmlReader,
- création de la classe ReplayPlayer,
- utilisation du parseur.

DOCUMENTS FOURNIS

Aucun. Débrouillez-vous.

HISTORIQUE DU JEU

struct
List<>

AdvancedMove

Ajoutez le type imbriqué AdvancedMove à Game. Ce type sera une structure contenant : une valeur de type Move et une liste de cartes (correspondant aux cartes tirées par le joueur après avoir joué, ou reçues initialement au début du jeu).

SortedList<,>

Historic

Ajoutez à Game, le membre mHistoric de type SortedList. Les clés seront de type DateTime et représenteront la date et l'heure d'exécution du coup. Les valeurs seront de type AdvancedMove et contiendront donc le coup et les cartes tirées.

PARSEUR POUR ENREGISTRER

XmlWriter

Parser XmlWriter

Créez un parseur XmlWriter qui enregistra dans le fichier XML vérifiant le schéma de la partie 10, la date et l'heure de l'enregistrement, les joueurs de cette partie (et leurs cartes en main au démarrage du jeu), l'ensemble des coups réalisés, en utilisant notamment l'historique créé précédemment.

Pour réaliser cette tâche, vous pourrez notamment créer un fichier XMLTags (inspiré de la partie précédente de la semaine 8) que vous pourrez préparer avec l'équipe verte, ainsi que des classes de stockage intermédiaires (en particulier pour les joueurs).

ENREGISTREMENT DANS UN FICHIER XML

Game

Mettez Game à jour pour que :

au démarrage du jeu, il ajoute un AdvancedMove par joueur à l'historique (contenant un Move de type inconnu, sans cloîtres choisis, sans pays choisis et sans cartes jouées, mais contenant les 3 cartes reçues au lancement du jeu),

à chaque coup joué, il ajoute un AdvancedMove avec le coup joué,

à la fin de la partie, il ajoute bien le dernier coup, puis enregistre la partie à l'aide du parseur.

LECTURE D'UN FICHIER XML

XmlReader

Création d'un parseur pour lire une partie enregistrée

Créez un parseur XmlReader qui contiendra des méthodes permettant de :

lire les joueurs (en les stockant à l'aide d'une classe de stockage temporaire, inspirée de celle de la partie 12),

lire un coup.

Pour réaliser cette tâche, vous pourrez notamment créer un fichier XMLTags (inspiré de la partie précédente de la semaine 8) que vous pourrez préparer avec l'équipe verte, ainsi que des classes de stockage intermédiaires (en particulier pour les joueurs).

REPLAY PLAYER

class

Création d'un joueur automatique

Créez un type de joueur ReplayPlayer dérivant de Player dans la bibliothèque du noyau. ReplayPlayer aura un membre du type du parseur créé précédemment, et la méthode Play lira le prochain coup dans le fichier XML et le jouera.

Vous remplirez ensuite la main avec les cartes indiquées dans le fichier XML.

LANCEMENT DU JEU

Game

Lancement du jeu depuis Game

Ajoutez un nouveau constructeur à Game pour qu'il lance un jeu à partir d'un fichier XML. Game devra alors créer les joueurs de type ReplayPlayer nécessaire, distribuer les cartes de départ et lancer le jeu.

AMÉLIORATIONS

N'enregistrez pas les replays !

Une fois le travail des deux équipes terminé, débrouillez-vous pour que les parties rejouées (replays) ne soient pas enregistrées.

Semaine 10

XSLT

OBJECTIFS

Les objectifs de la dixième semaine, du point de vue de l'application finale, sont :

- d'afficher les règles du jeu en HTML, à partir du fichier XML.

Les objectifs pédagogiques sont :

- l'écriture d'une feuille XSLT pour transformer tous les fichiers XML valides par rapport à un schéma en HTML,
- la vérification du fonctionnement de cette feuille de transformation XSLT à l'aide des règles du jeu de Web Of Power et du jeu de Bataille.

PRÉPARATION

Relisez le cours sur XLST, reprenez le schéma réalisé en semaine 7, et étudiez les fichiers XML des règles du jeu de Web Of Power et du jeu de Bataille fourni.

Partie 14 : règles du jeu au format HTML

XSLT

OBJECTIFS

L'objectif de cette partie est d'écrire une feuille XSLT pour transformer les règles du jeu au format HTML.

Les objectifs pédagogiques sont :

- l'écriture et l'utilisation d'une feuille XSLT.

PRÉPARATION

Lisez l'énoncé ci-dessous ainsi que le cours. Deux pages HTML vous sont fournies : elles représentent le résultat à obtenir. Étudiez donc la page source de ces deux pages.

DOCUMENTS FOURNIS

Les documents sont fournis pour la première fois (ou plus) :

- le schéma `rules_TP7b.xsd` : la feuille XSLT devra transformer n'importe quel fichier XML valide par rapport à ce schéma ;
- deux fichiers XML valides par rapport au schéma précédent, `rules_TP7b.xml` et `rules_bataille.xml`, représentant respectivement les règles du jeu de Kardinal & König et de la Bataille (vous pourrez vérifier le bon fonctionnement de votre feuille XSLT avec ces deux fichiers XML) ;
- deux pages HTML, `WebOfPower.html` et `Bataille.html`, qui représentent les résultats à obtenir, respectivement pour le jeu Kardinal & König et pour la Bataille.

XSLT

XSLT : XML => HTML

Écrire une feuille XSLT pour transformer les fichiers XML fournis en page HTML (comme celles fournies).

Quelques conseils :

- utilisez des modèles (...),
- pour le nom du jeu, le premier est mis en gros et en rouge, les suivants en petit, en italique, entre parenthèses : pour cela, différenciez-les à l'aide de la fonction XPath `position()`,
- utilisez des fonctions XPath pour transformer les temps (par exemple `translate`),
- pour le tableau de matériel, pour pouvoir réaliser des `rowspan` et n'afficher les bordures que de certaines cellules, créez deux modèles différents (en utilisant des modes), pour différencier les premières lignes de chaque type et les suivantes,
- le plus dur est certainement d'afficher les types des règles en titre 2 : vous devriez pouvoir y arriver à l'aide de deux boucles `for-each` imbriquées et de l'utilisation de l'axes `preceding-sibling` pour supprimer les doublons (ici, pour pouvoir n'afficher qu'une fois chaque titre en bleu).

Semaine 11

XSLT

OBJECTIFS

Les objectifs de la onzième semaine, du point de vue de l'application finale, sont :

- d'afficher le classement des joueurs en HTML, à partir du fichier XML.
- d'afficher le profil d'un joueur en HTML, à partir du fichier XML.

Les objectifs pédagogiques sont :

- l'écriture d'une feuille XSLT pour transformer tous les fichiers XML valides par rapport à un schéma en HTML.

PRÉPARATION

Relisez le cours sur XLST et les suppléments XSLT, reprenez le schéma réalisé en partie 11 de la semaine 7, et étudiez le fichier XML matchData.xml.

Partie 15 : transformation du classement des joueurs

XSLT

OBJECTIFS

On souhaite afficher le classement des joueurs de Kardinal & König à travers une page web. Pour cela, nous nous proposons d'utiliser le contenu du fichier `matchData.xml` et une transformation XSLT. Le résultat permettra d'afficher :

- le rang de chaque joueur (en fonction de différents critères : plus de victoires, plus de défaites ou pourcentage de victoires),
- une icône pour chaque joueur attribuée en fonction du nombre de victoires de ce joueur : de 0 à 2, l'icône du niveau 1 ; de 3 à 9, l'icône du niveau 2 ; de 10 à 24, l'icône du niveau 3 ; de 25 à 49, l'icône du niveau 4 ; plus de 50, l'icône du niveau 5,
- le nom de chaque joueur,
- le type de chaque joueur : «Human Player» dans le cas d'un joueur humain, ou la classe puis la dll (entre parenthèses) dans le cas d'un joueur AI,
- le pourcentage de victoires,
- le nombre de victoires de chaque joueur,
- le nombre de défaites de chaque joueur.

Une variable dans le fichier XSLT permettra d'afficher soit le classement tous types confondus, soit le classement des joueurs humains, soit le classement des intelligences artificielles.

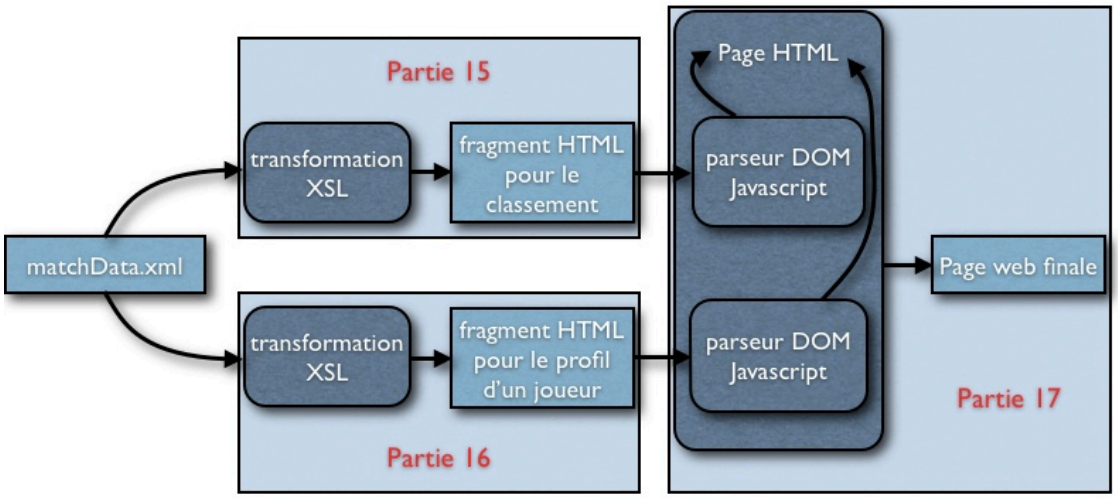
Une autre variable dans le fichier XSLT permettra de choisir si on classe les joueurs par le pourcentage de victoires, ou le nombre de victoires (puis le plus petit nombre de défaites), ou le nombre de défaites (puis le plus petit nombre de victoires).

Les objectifs pédagogiques sont :

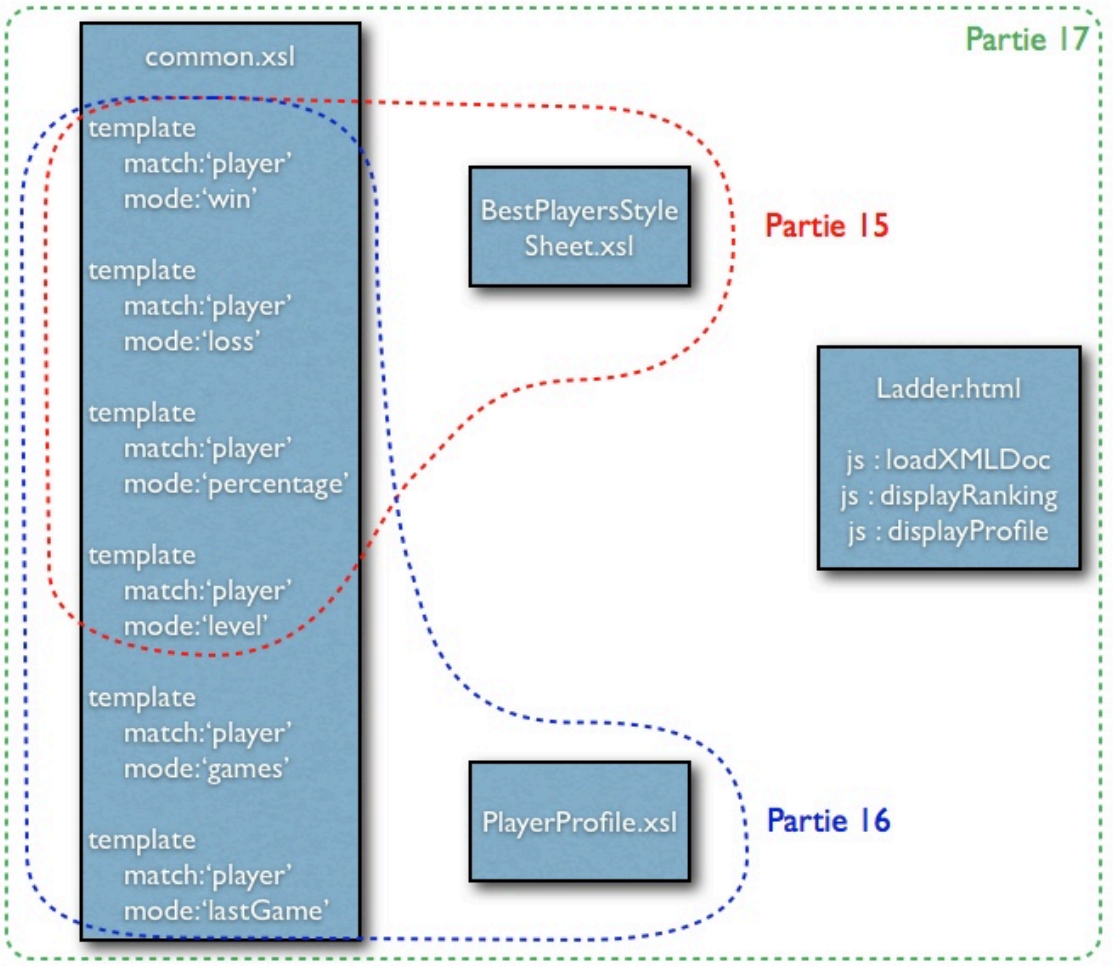
- l'écriture et l'utilisation d'une transformation (ou feuille de style) XSLT.

Note : pour simplifier le travail, j'ai rajouté des attributs `nb_wins` *et* `nb_losses` *aux éléments* `player` *dans le fichier* `matchData.xml`.

Les parties 15, 16 et 17 sont liées et ont un même objectif : réaliser une «page web» permettant de visualiser le classement des joueurs en fonction de leur type, et le profil de chaque joueur. La partie 15 vous aidera à écrire une transformation XSLT pour afficher le classement. La partie 16 vous aidera à écrire une transformation XSLT pour afficher le profil d'un joueur. Enfin, la partie 17 vous guidera à travers la réalisation d'une page web très simple bénéficiant des résultats des deux transformations précédentes grâce à des parseurs DOM en Javascript.



Pour atteindre ce résultat, vous devrez écrire 4 fichiers : `common.xsl`, `BestPlayersStyleSheet.xsl`, `PlayerProfile.xsl` et `Ladder.html`. Le premier d'entre eux contiendra plusieurs modèles xslt et sera utilisé dans les parties 15 et 16. Vous écrirez le 2ème dans la partie 15, le 3ème dans la partie 16, et le dernier sera parsé et modifié par des parseurs DOM Javascript dans la partie 17. Le schéma suivant résume les dépendances entre les parties 15 à 17.



Il est donc possible de réaliser la partie 16 sans avoir fait la partie 15, mais dans les deux cas, vous aurez à faire les modèles en commun dans `common.xsl`. L'équipe rouge sera responsable de la partie 15, l'équipe verte de la partie 16. La partie 17 concerne seulement ceux qui ont envie de s'amuser.

Préparation de la partie 15

Dans la partie 15, vous devrez donc transformer le fichier `matchData.xml` en un fragment HTML qui pourra ressembler à la page suivante : http://marc.chevaldonne.free.fr/WebOfPower/BestPlayersStyleSheet.htm. Pensez notamment à visualiser le code source de la page pour vous en inspirer dans l'écriture de la transformation XSLT.

PRATIQUE

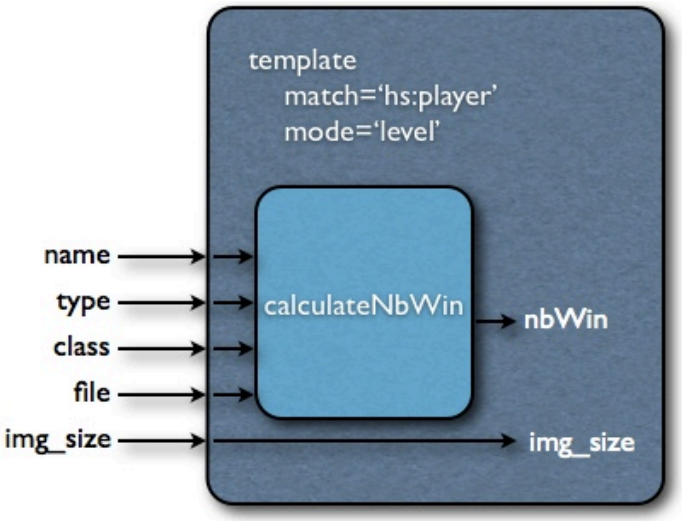
Cette partie 15 est divisée en deux sous-parties : l'écriture de modèles dans le fichier `common.xsl` et l'écriture de la transformation `BestPlayersStyleSheet.xsl` qui utilisera entre autres les modèles écrits dans `common.xsl`. Ces deux sous-parties doivent donc en conséquence être réalisées dans l'ordre.

Quatre modèles seront créés dans `common.xsl` :

- un modèle permettant d'afficher l'icône correspondant au niveau d'un joueur,
- un modèle permettant d'afficher le nombre de victoires d'un joueur
- un modèle permettant d'afficher le nombre de défaites d'un joueur,
- un modèle permettant d'afficher le pourcentage de victoires d'un joueur.

L'équipe verte a également besoin de ces modèles, arrangez-vous avec eux.

Modèle permettant d'afficher l'icône correspondant au niveau d'un joueur



Ce modèle a pour objectif d'afficher une icône associée au joueur, en fonction du nombre de victoires de ce joueur. Puisqu'il est donc nécessaire de connaître le nombre de victoires, nous allons d'abord créer un modèle (fonction) calculant le nombre de victoires d'un joueur. Le nombre de victoires pourrait être (et en fait sera) réutilisé ailleurs : c'est pour cette raison que nous allons placer ce calcul dans un modèle. Ensuite, nous récupérerons le résultat de ce calcul dans une variable à l'aide d'un `call-template`, et nous afficherons l'icône recherchée.

xsl:variable

Préparation des variables

Pour améliorer la maintenance de notre transformation, le plus sage est de construire des variables pour chaque valeur «constante» au moment de la transformation, mais susceptible d'être modifiée avant que la transformation ne soit exécutée. Nous placerons donc en variables globales, les valeurs permettant de choisir combien de victoires sont nécessaires pour passer d'un niveau au niveau supérieur et les url des icônes.

xsl:template
xsl:param
xsl:value-of

calculateNbWin template

Écrivez un modèle qui prendra en paramètres

- le nom du joueur dont on calcule le nombre de victoires,
- le type de ce joueur,
- la classe de ce joueur,
- et le fichier contenant cette classe.

Rendez le nombre de victoires à l'aide d'un élément `xsl:value-of` et d'un prédicat adapté.

xsl:template
xsl:param
xsl:variable
xsl:call-template

Modèle affichant l'icône du joueur
template : match:player / mode:level

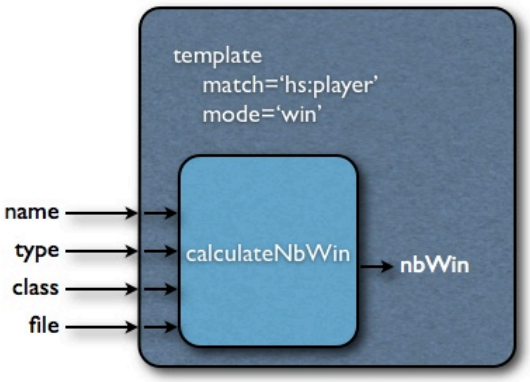
Écrivez un modèle qui prendra en paramètres

- le nom du joueur dont on veut afficher l'icône,
- le type de ce joueur,
- la classe de ce joueur,
- le fichier contenant cette classe,
- et la taille de l'image (largeur de l'image).

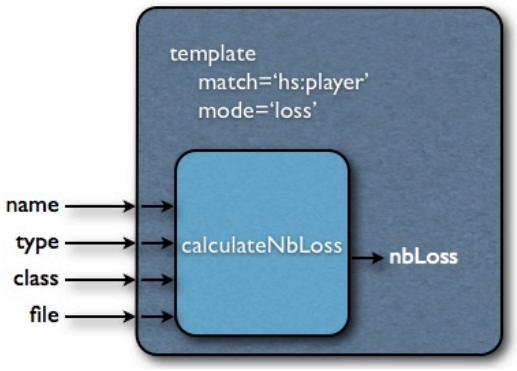
Utilisez le modèle `calculateNbWin` et les 4 premiers paramètres à l'aide d'un `call-template` pour en déduire le nombre de victoires de ce joueur, qui vous placerez dans une variable. En fonction du résultat obtenu et en utilisant les variables globales, affichez l'icône correspondante avec la bonne taille.

Modèle permettant d'afficher le nombre de victoires d'un joueur

Ce modèle a pour objectif d'afficher le nombre de victoires d'un joueur. Réutilisez le modèle `calculateNbWin` créé dans la question précédente, récupérez le nombre de victoires et affichez-le. Utilisez pour cela la même méthode que précédemment, c'est-à-dire une variable remplie à l'aide d'un `call-template`.

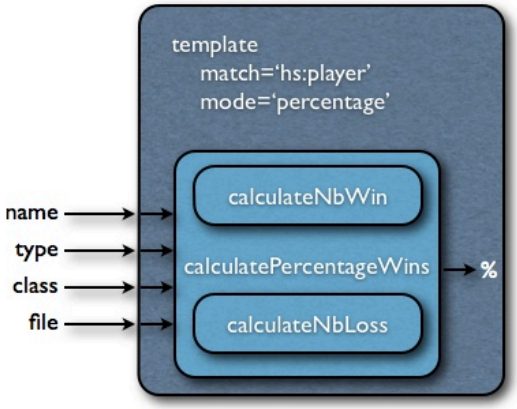


Modèle permettant d'afficher le nombre de défaites d'un joueur



Ce modèle a pour objectif d'afficher le nombre de défaites d'un joueur. Inspirez-vous des questions précédentes pour créer un modèle `calculateNbLoss` et réutilisez-le dans le modèle qui affichera le nombre de défaites.

Modèle permettant d'afficher le pourcentage de victoires d'un joueur



Ce modèle a pour objectif d'afficher le pourcentage de victoires d'un joueur. Inspirez-vous des questions précédentes pour créer un modèle `calculatePercentageWins` et réutilisez-le dans le modèle qui affichera le nombre de défaites. Vous pourrez par exemple réutilisez les modèles `calculateNbWin` et `calculateNbLoss`.

BESTPLAYERSSTYLESHEET.XSL

`BestPlayersStyleSheet.xsl` doit transformer le contenu du document `matchData.xml` en un fragment HTML ressemblant à http://marc.chevaldonne.free.fr/WebOfPower/BestPlayersStyleSheet.htm.

Il doit être possible d'afficher le classement de tous les joueurs, le classement des joueurs humains seulement ou le classement des joueurs AI seulement. Pour cela, utilisez une variable globale que vous pourrez nommer `player_types`, qui pourra prendre la valeur `string('all')`, `string('human')` ou `string('ai')`. En fonction de la valeur donnée à la variable, le résultat sera évidemment différent.

Les joueurs pourront être classés selon trois modes. Pour cela, utilisez une variable globale `ranking_criterion` qui pourra prendre les valeurs suivantes :

- par ordre décroissant du nombre de victoires (puis en cas d'égalité, par ordre croissant du nombre de défaites),
- par ordre décroissant du nombre de défaites (puis en cas d'égalité, par ordre croissant du nombre de victoires),

- par ordre décroissant du pourcentage de victoires (puis en cas d'égalité, par ordre décroissant du nombre de victoires).

Chaque ligne devra comprendre :

- le rang du joueur,
- l'icône du joueur en fonction de son nombre de victoires,
- le nom du joueur,
- le type du joueur : `HumanPlayer` pour les joueurs humains, `Class (File)` pour les joueurs AI,
- le pourcentage de victoires du joueur,
- le nombre de victoires du joueur,
- le nombre de défaites du joueur.

En ce qui concerne l'icône, le nombre de victoires et le nombre de défaites, vous devrez naturellement utiliser les modèles écrits dans `common.xsl`. Pour cela, utiliser l'élément `xsl:include` pour inclure `common.xsl`.

Pour la plupart des modèles, vous devrez passer les paramètres pour décrire le joueur (`name`, `type`, `class`, `file`) et éventuellement d'autres. Utilisez pour cela des éléments `xsl:with-param` en sous-éléments de `xsl:apply-templates`.

Enfin, pour pouvoir changer facilement l'apparence du résultat, il est conseillé de placer un maximum de constantes en `xsl:variable` (par exemple, les couleurs des lignes, les couleurs de texte, etc.).

Partie 16 : transformation du profil d'un joueur

XSLT

OBJECTIFS

On souhaite afficher le profil et les statistiques d'un joueur de WebOfPower à travers une page web. Pour cela, nous nous proposons d'utiliser le contenu du fichier `matchData.xml` et une transformation XSLT. Le résultat permettra d'afficher :

- une icône pour chaque joueur attribuée en fonction du nombre de victoires de ce joueur : de 0 à 2, l'icône du niveau 1 ; de 3 à 9, l'icône du niveau 2 ; de 10 à 24, l'icône du niveau 3 ; de 25 à 49, l'icône du niveau 4 ; plus de 50, l'icône du niveau 5,
- le nom du joueur,
- le type du joueur : «Human Player» dans le cas d'un joueur humain, ou la classe puis la dll (entre parenthèses) dans le cas d'un joueur AI,
- les statistiques du joueur :
  - la date et l'heure de la dernière partie jouée par ce joueur,
  - le nombre de parties jouées par ce joueur,
  - le nombre de victoires de ce joueur,
  - le nombre de défaites de ce joueur,
- les informations sur les 2 dernières parties jouées par ce joueur, avec pour chaque partie :
  - la date et l'heure de la partie,
  - le nom et le type du vainqueur,
  - le nom et le type du perdant.

Des variables globales dans le fichier XSLT permettront de choisir le nom, le type, la classe et le fichier du joueur dont on souhaite afficher le profil et les statistiques.

Les objectifs pédagogiques sont :

- l'écriture et l'utilisation d'une transformation (ou feuille de style) XSLT.

Note : pour simplifier le travail, j'ai rajouté des attributs `nb_wins` *et* `nb_losses` *aux éléments* `player` *dans le fichier* `matchData.xml`.

PRÉPARATION

Les parties 15, 16 et 17 sont liées et ont un même objectif : réaliser une «page web» permettant de visualiser le classement des joueurs en fonction de leur type, et le profil de chaque joueur. Des informations concernant ces liens ont été données dans la partie Préparation de la partie 15. Il est fortement conseillé de les lire si vous n'avez pas fait la partie 15.

Préparation de la partie 16

Dans la partie 16, vous devrez transformer le fichier `matchData.xml` en un fragment HTML qui pourra ressembler à la page suivante : http://marc.chevaldonne.free.fr/WebOfPower/PlayerProfile.htm. Pensez notamment à visualiser le code source de la page pour vous en inspirer dans l'écriture de la transformation XSLT.

PRATIQUE

Cette partie 16 est divisée en deux sous-parties : l'écriture de modèles dans le fichier `common.xsl` et l'écriture de la transformation `PlayerProfile.xsl` qui utilisera entre autres les modèles écrits dans `common.xsl`. Ces deux sous-parties doivent donc en conséquence être réalisées dans l'ordre.

COMMON.XSL

Six modèles[7] seront créés dans `common.xsl` :

- un modèle permettant d'afficher l'icône correspondant au niveau d'un joueur,
- un modèle permettant d'afficher le nombre de victoires d'un joueur,
- un modèle permettant d'afficher le nombre de défaites d'un joueur,
- un modèle permettant d'afficher le pourcentage de victoires d'un joueur,
- un modèle permettant d'afficher le nombre de parties réalisées par un joueur,
- un modèle permettant d'afficher la date et l'heure de la dernière partie jouée par un joueur.

[7] L'équipe rouge a besoin également de 4 de ces modèles, arrangez-vous avec eux pour la répartition des tâches.

Modèle permettant d'afficher l'icône correspondant au niveau d'un joueur

Si vous avez fait la partie 15, il vous faut juste compléter le fichier `common.xsl`. Ce modèle est alors déjà fait. Les indications se trouvent ici.

Modèle permettant d'afficher le nombre de victoires d'un joueur

Si vous avez fait la partie 15, il vous faut juste compléter le fichier `common.xsl`. Ce modèle est alors déjà fait. Les indications se trouvent ici.

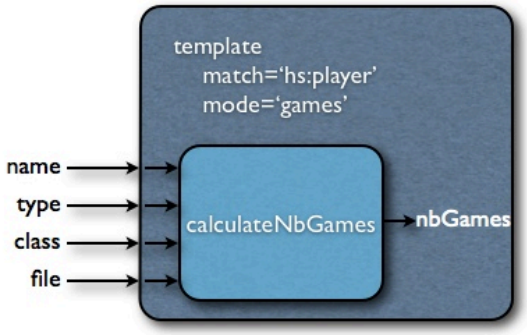
Modèle permettant d'afficher le nombre de défaites d'un joueur

Si vous avez fait la partie 15, il vous faut juste compléter le fichier `common.xsl`. Ce modèle est alors déjà fait. Les indications se trouvent ici.

Modèle permettant d'afficher le pourcentage de victoires d'un joueur

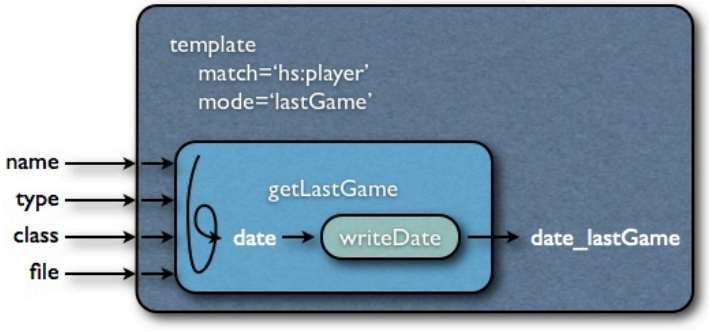
Si vous avez fait la partie 15, il vous faut juste compléter le fichier `common.xsl`. Ce modèle est alors déjà fait. Les indications se trouvent ici.

Modèle permettant d'afficher le nombre de parties réalisées par un joueur



Ce modèle a pour objectif d'afficher le nombre de parties d'un joueur. Inspirez-vous des questions précédentes pour créer un modèle `calculateNbGames` et réutilisez-le dans le modèle qui affichera le nombre de parties.

Modèle permettant d'afficher la date et l'heure de la dernière partie jouée par un joueur



Ce modèle a pour objectif d'afficher la date et l'heure de la dernière partie jouée par un joueur. Inspirez-vous des questions précédentes pour créer un modèle `getLastGame` et réutilisez-le dans le modèle qui affichera la date et l'heure, dans le format suivant : `2010/12/03 (02h26)`. Notez également, que dans la partie sur les deux

dernières parties réalisées, nous aurons également besoin d'afficher des dates dans ce format. Pour cette raison, écrivez un modèle `writeDate` qui affichera une date `date`, passée en paramètre (avec `xs:param`) au format `xs:dateTime`, dans le format indiqué précédemment. `getLastGame` cherchera donc la date avec les informations du joueur passées en paramètres, puis utilisera `writeDate` pour l'affichage.

PLAYERPROFILE.XSL

`PlayerProfile.xsl` doit transformer le contenu du document `matchData.xml` en un fragment HTML ressemblant à http://marc.chevaldonne.free.fr/WebOfPower/PlayerProfile.htm.

Il doit être possible d'afficher le profil de n'importe quel joueur (humain ou ia). Pour cela, utilisez des variables globales que vous pourrez nommer `playerName`, `playerType`, `playerClass` et `playerFile`, qui pourront prendre les valeurs correspondantes à leurs noms et permettant de retrouver un joueur dans le fichier `matchData.xml`. En fonction des valeurs données aux variables, le résultat sera donc évidemment différent.

Chaque profil devra comprendre :

- une icône attribuée en fonction du nombre de victoires de ce joueur (tous types confondus) : de 0 à 2, l'icône du niveau 1 ; de 3 à 9, l'icône du niveau 2 ; de 10 à 24, l'icône du niveau 3 ; de 25 à 49, l'icône du niveau 4 ; plus de 50, l'icône du niveau 5.
- le nom du joueur,
- le type du joueur : «Human Player» dans le cas d'un joueur humain, ou la classe puis la dll (entre parenthèses) dans le cas d'un joueur AI,
- les statistiques du joueur :
    - la date et l'heure de la dernière partie jouée par ce joueur,
    - le nombre de parties jouées par ce joueur,
    - le nombre de victoires de ce joueur,
    - le nombre de défaites de ce joueur,
- les informations sur les 2 dernières parties jouées par ce joueur, avec pour chaque partie :
    - la date et l'heure de la partie,
    - le nom et le type du vainqueur,
    - le nom et le type du perdant.

En ce qui concerne l'icône, le nombre de victoires, le nombre de défaites, l'affichage de la date et de l'heure de la dernière partie (et des autres parties également) vous devrez

naturellement utiliser les modèles écrits dans `common.xsl`. Pour cela, utiliser l'élément `xsl:include` pour inclure `common.xsl`.

Pour la plupart des modèles, vous devrez passer les paramètres pour décrire le joueur (`name`, `type`, `class`, `file`) et éventuellement d'autres. Utilisez pour cela des éléments `xsl:with-param` en sous-éléments de `xsl:apply-templates`.

Enfin, pour pouvoir changer facilement l'apparence du résultat, il est conseillé de placer un maximum de constantes en `xsl:variable` (par exemple, les couleurs des lignes, les couleurs de texte, etc.).

Partie 17 : site web dynamique du jeu

parser DOM Javascript, XSLT, HTML

OBJECTIFS

Pour achever l'affichage web d'informations sur le jeu Kardinal & König, nous allons créer une page web qui permettra d'afficher le classement des joueurs de manière dynamique (c'est-à-dire qu'au fur et à mesure des parties, le fichier XML sera modifié et le classement en conséquence, de manière automatique) et le profil des joueurs de manière dynamique également. Pour cela, nous rajouterons des liens.

Cette page web contiendra donc :

- 3 boutons permettant de choisir l'affichage du classement de tous les joueurs, des joueurs humains seulement ou des joueurs IA seulement,
- dans la page de classement, les intitulés des colonnes `%` (pourcentage de victoires), `Wins` et `Losses` seront «cliquables» et permettront de choisir le critère de classement,
- en fonction des boutons cliqués, l'affichage de la transformation XSLT du classement,

Kardinal & König Ladders

All Human Ai

Rank	Level	Player Name	Player Type	▼ %	Wins	Losses
1st		Barbidur	giWebOfPowerCore.FirstMovePlayer (giWebOfPowerCore.dll)	50%	2	2
2nd		Barbapapa	giWebOfPowerCore.RandomPlayer (giWebOfPowerCore.dll)	41%	13	19
3rd		Jim Halpert	Human Player	29%	8	20
4th		Barbalala	giWebOfPowerCore.FirstMovePlayer (giWebOfPowerCore.dll)	25%	1	3
5th		Dwight Schrute	Human Player	22%	7	25
6th		Michael Scott	Human Player	21%	6	22
7th		Ellen Ripley	giWebOfPowerCore.RandomPlayer (giWebOfPowerCore.dll)	11%	3	25
8th		Barbidoux	giWebOfPowerCore.RandomPlayer (giWebOfPowerCore.dll)	0%	0	4

Classement de tous les joueurs selon le pourcentage de victoires

Classement des joueurs humains seulement, selon le pourcentage de victoires

Classement des joueurs IA seulement, selon le pourcentage de victoires

Classement des joueurs humain selon le nombre de victoires

- sur le nom de chaque joueur, un lien qui permettra d'afficher le profil du joueur sur lequel on clique à la place du classement,
- dans le profil du joueur, des liens permettant de cliquer sur les noms des joueurs (dans la section «2 dernières parties») pour afficher le profil de ces joueurs.

Profil d'un joueur IA

Profil d'un joueur humain

Pour réaliser tout cela, nous devrons donc parser les fichiers de transformation, afin de modifier le contenu des variables permettant de choisir le type du classement, ou le joueur dont on veut afficher le profil. Ces parseurs DOM seront réalisés en Javascript dans le code de la page HTML.

PRÉPARATION

Les parties 15, 16 et 17 sont liées et ont un même objectif : réaliser une «page web» permettant de visualiser le classement des joueurs en fonction de leur type, et le profil de chaque joueur.

Des informations concernant ces liens ont été données dans la partie Préparation de la partie 15. Il est conseillé de la lire si vous n'avez pas fait les parties 16 et 17 avant la partie 17.

Préparation de la partie 17

Dans la partie 17, vous devrez créer une page web dynamique affichant le contenu des transformations XSLT des parties précédentes après avoir modifié le contenu de variables globales des ces transformations.

PRATIQUE

HTML

Création de la page web

Créez une page web qui contiendra un titre, et trois liens pour les trois types de classement[8]. Ajoutez également une division (`div`) qui contiendra ensuite les résultats des transformations XSLT.

Javascript
DOM parser

Chargement des fichiers XML

Ajoutez dans la section `head` de ce fichier HTML, une méthode `loadXMLDoc` en javascript qui permettra de charger un fichier XML dont on passera le nom en paramètre. Pour créer cette méthode, inspirez-vous de (pour ne pas dire, copier) la méthode proposée par le w3schools.

Toujours dans la section `head`, ajoutez trois variables globales `xml`, `xsl` et `xslProfile`, chargeant respectivement les fichiers `matchData.xml`, `BestPlayersStyleSheet.xsl` et `PlayerProfile.xsl` (sur le web naturellement).

Javascript
DOM parser

Parsing et préparation de BestPlayersStyleSheet.xsl

Ajoutez dans la section `head` de ce fichier HTML, une méthode `displayRanking` en javascript permettant de parser le fichier `BestPlayersStyleSheet.xsl` et de le modifier.

Cette méthode prendra en paramètre le type du classement à préparer (`all`, `human` ou `ai`).

[8] Les liens pointent en fait vers la même page, mais nous ajouterons ensuite un événement onclick.

Voici une proposition de plan à suivre pour cette fonction :

- recherchez tous les éléments `xsl:variable` du fichier xsl,
- parmi ceux-là, recherchez celui qui a pour attribut `name`, la valeur `player_types`,
- modifiez la valeur de son attribut `select` en `string ('paramètre')`, où *`paramètre`* est le paramètre passé dans la fonction `displayResult`,
- effectuez la transformation en vous inspirant de la solution proposée par le w3schools. Vous devrez rajouter notamment la suppression de tous les sous-éléments de l'élément `div` avant de rajouter un nouvel enfant correspondant à la transformation xslt.

Javascript
HTML

Affichage du résultat de la transformation

Ajoutez un événement `onload` à l'élément `body` de votre page, qui appellera la méthode `displayRanking` précédente avec la valeur `all`.

Ajoutez des événements `onclick` aux éléments correspondant aux boutons `All`, `Human` et `AI` qui appelleront la méthode `displayRanking` avec les valeurs respectives `all`, `human` et `ai`.

Notre page permet maintenant d'afficher le classement de tous les types de joueurs, des joueurs humains ou des joueurs ia ! Tout cela en n'utilisant qu'une feuille de transformation XSLT.

Javascript
DOM Parser

Parsing et préparation de PlayerProfile.xsl

Ajoutez dans la section `head` de ce fichier HTML, une méthode `displayProfile` en javascript permettant de parser le fichier `PlayerProfile.xsl` et de le modifier.

Cette méthode prendra en paramètre le nom, le type, la classe et le fichier du joueur dont on souhaite afficher le profil.

Voici une proposition de plan à suivre pour cette fonction :

- recherchez tous les éléments `xsl:variable` du fichier xsl,
- parmi ceux-là, recherchez ceux qui ont pour attribut `name`, une des valeurs suivantes `playerName`, `playerType`, `playerClass` ou `playerFile`,
- modifiez la valeur de l'attribut `select` en `string('name')`, `string('type')`, `string('_class')`, `string('file')`, où `name`, `type`, `_class` et `file` sont les paramètres de la fonction `displayProfile`,

- effectuez la transformation en vous inspirant de la solution proposée par le w3schools. Vous devrez rajouter notamment la suppression de tous les sous-éléments de l'élément `div` avant de rajouter un nouvel enfant correspondant à la transformation xslt.

Javascript
DOM Parser

Liens HTML pour l'affichage du résultat de la transformation

On veut maintenant afficher le résultat de la transformation lorsqu'on cliquera sur le nom d'un joueur dans le classement affiché. Il nous faut donc rajouter des balises `<a>` autour de l'affichage du nom du joueur.

Nous pouvons faire cela en parsant le fichier xsl `BestPlayersStyleSheet.xsl` avant que la transformation ne soit exécutée, dans la fonction `displayResult`.

Pour cela, vous pouvez par exemple suivre la méthode suivante :

- recherchez dans le fichier xsl, tous les éléments `xsl:template`,
- parmi eux, recherchez celui qui a pour attribut `match` la valeur `hs:player` et pour attribut `mode` la valeur `line`,
- récupérez son 3ème élément `td`,
- supprimez tous les sous-éléments de cet élément `td`,
- ajoutez lui un sous-élément `<a>` avec un attribut `onclick=displayProfile('{$playerName}', '{$playerType}', '{$playerClass}', '{$playerFile}')`[9] et un attribut `href` pointant sur cette même page
- ajoutez à cet élément `<a>` un sous-élément `<b>`[10]
- ajoutez à ce sous-élément `<b>` un sous-élément `<xsl:value-of>` affichant le nom du joueur en cours (c'est-à-dire rajoutant la valeur «`@hs:name`» à l'attribut `select`,

Visualisez maintenant la transformation : vous devriez pouvoir cliquer sur les noms des joueurs dans la fenêtre du classement et voir apparaître le profil de ce joueur à la place du classement.

---

[9] `$playerName`, `$playerType`, `$playerClass` et `$playerFile` correspondent à celles du `template` en question

[10] si vous avez garder la même apparence que celle proposée

Javascript
DOM Parser

Liens HTML pour l'affichage du résultat de la transformation - 2

On veut maintenant afficher le résultat de la transformation lorsqu'on cliquera sur le nom d'un joueur dans les dernières parties jouées affichées dans le profil d'un joueur. Inspirez-vous de la question précédente et modifiez la fonction `displayProfile` en conséquence.

Javascript
DOM Parser

Critère de classement

Inspirez-vous des questions précédentes pour gérer les clics sur les colonnes `%`; `Wins` et `Losses`.

Note : le caractère ▼ a le code UTF-8 \u25BC.

Semaine 12

Chargement dynamique des IAs

OBJECTIFS

Les objectifs de la douzième semaine, du point de vue de l'application finale, sont :

- de charger n'importe quelle IA non connue de l'application (dans un assemblage externe).
- de créer un jeu qui permet de jouer avec ces IAs et des joueurs Humains.

Les objectifs pédagogiques sont :

- l'utilisation de la Reflection en C#.

PRÉPARATION

Étudiez les exemples de cours 82 de C#.

Partie 18 : la gestion des IA

Reflection, Type, Métadonnées, analyse d'assemblage

OBJECTIFS

L'objectif de cette partie est de permettre d'instancier dynamiquement des classes de joueurs sans connaître leur type à la compilation. En d'autres termes, le but est de permettre d'exécuter n'importe quel algorithme d'Intelligence Artificielle sans savoir de quelle classe ni de quel assemblage il provient.

Les objectifs pédagogiques sont :

- l'utilisation de la reflection : utilisation durant l'exécution des métadonnées de type d'un assemblage.
- l'utilisation de la classe Type.

PRÉPARATION

Étudiez les exemples du cours sur la réflexion (82).

PRATIQUE

héritage

Classe mère des IA

Créez une classe abstraite `AIPlayer` qui possède une propriété `AIName` en lecture seule (représentant le nom du type de l'IA qui sera affiché dans l'interface graphique), et un constructeur `protected` prenant en paramètre une chaîne de caractères représentant le nom du joueur IA (et pas son type).

reflection
Type

Chargement des IA

Créez une classe statique `AILoader` qui possédera deux méthodes.
La première méthode, `CreateInstance`, permettra d'instancier une classe par son type, si et seulement si elle dérive de `AIPlayer`.
La seconde, `GetAIClasses`, devra chercher dans les assemblages contenus dans un dossier (passé en paramètre), l'ensemble des classes

dérivant de `AIPlayer` et les stocker dans une `SortedList<K, V>` où la clé sera le contenu de la propriété `AIame` d'une classe et `V` le type de cette même classe. Elle devra donc utiliser la première méthode pour récupérer la propriété `AIame`.

Type

Ajout d'un joueur IA au jeu

Ajoutez une méthode `AddPlayer(Type, string)` à `Game` et utilisez la classe `AILoader` pour charger cette classe.

Testez en codant de stupides IA.

Partie 19 : dernière version du jeu Console

IA et HumanPlayer

PRATIQUE

Tests

Amélioration du jeu Console

Améliorez le jeu Console (Tests) pour qu'il permette de choisir son IA et son Human Player ainsi que leur nom, pour chaque joueur. On doit ainsi pouvoir faire jouer des IAs contre des humains.

Semaine 13

Interface graphique pour la préparation du jeu

OBJECTIFS

Les objectifs de la treizième semaine, du point de vue de l'application finale, sont :

- de créer une interface graphique permettant aux joueurs de préparer les règles du jeu à utiliser, le plateau de jeu, et les joueurs (nom, type, couleur).

Les objectifs pédagogiques sont :

- la création d'une interface graphique en XAML,
- la liaison entre le XAML et la bibliothèque principale à l'aide du «code behind».

PRÉPARATION

Étudiez les exemples de cours 83 à 94 de C#.

Partie 20 : interface graphique

XAML, WPF, code-behind

OBJECTIFS

L'objectif de cette partie est de permettre de préparer le jeu (joueurs, règles, plateau de jeu).

Les objectifs pédagogiques sont :

- la découverte de XAML et de WPF,
- l'utilisation d'outils de `Layout` (`DockPanel`, `Grid`, `WrapPanel`...),
- l'utilisation de contrôles XAML (boutons, combo boxes, text blocks, ...),
- l'utilisation d'outils graphiques (`ViewBox`, `Rectangle`, `Image`...),
- l'utilisation de dialogues (`MessageBox`, `ColorPickerDialog`),
- l'utilisation des événements des différents contrôles XAML,
- le lien avec la bibliothèque principale de notre projet (grâce au code-behind).

PRÉPARATION

Étudiez les exemples du cours sur XAML et WPF (83 à 94).

PRATIQUE

Window
Layout

Projet WPF + organisation de la fenêtre principale

Ajoutez à la solution un nouveau projet WPF. Préparez au brouillon l'interface à réaliser (en réfléchissant en particulier aux `Layout`). Vous pourrez par exemple réaliser l'interface suivante :

- un menu en haut contenant deux options (High Scores et Replay),
- un bas de la fenêtre, la possibilité de choisir le plateau de jeu et les règles du jeu, ainsi qu'un bouton pour démarrer le jeu,
- au milieu, une interface pour définir et préparer l'ensemble des joueurs (nom, type, couleur).

Choisissez et utilisez les outils de `Layout` adaptés (en particulier le `DockPanel` et la `Grid`) puis organisez vos contrôles (`TextBlock`, `ComboBox`, `TextBox`...) dans la fenêtre.

ComboBox (items en «dur»)

Plateau et règles du jeu

Dans les combo box pour le plateau de jeu et les règles du jeu, ajoutez respectivement les 2 options suivantes en XAML :

- `Europe` ou `Greece`,
- `Classic Rules` ou `Variant`.

Ajoutez deux membres à votre classe (en *code-behind*) :

- un champ de type `BaseRules`,
- un champ de type `Board`.

Construisez-les avec le constructeur de `EuropeanBoard` ou `Greek4_5Board` (en fonction de la selection de la combo box du plateau de jeu), et avec le constructeur de `ClassicRules` ou de `VariantRules` (en fonction de la sélection de la combo box des règles du jeu).

Grid

Préparation des joueurs

À l'aide d'une `Grid` et de contrôles, préparez la répartition des différents contrôles pour la définition des joueurs en XAML.

Style

Styles pour les titres

Pour factoriser et améliorer la maintenance de votre XAML, créez et utilisez un `Style` qui sera appliqué à tous les titres (`Board`, `Rules`, `Player i`, `Name`, `Type`, `Color`...).

ComboBox (items dynamiques)

Type des joueurs

Dans les combo box pour le type des joueurs, remplissez automatiquement les choix autorisés dans le *code-behind*, en utilisant notamment la classe `AILoader` (cf. semaine 12). Vous pouvez étudier pour cela le code de l'application de tests d'`AILoader`.Vous rajouterez au début de la liste des types autorisés, le type `Human Player`.

Rectangle
ColorPickerDialog

Choix des couleurs des joueurs

Il n'existe pas de `ColorDialog` en WPF. Néanmoins, il existe un exemple téléchargeable sur le MSDN pour VisualStudio2005 que vous pouvez recompiler sans erreur pour VisualStudio2010. Il est accessible en suivant ce lien[11].

Étudiez l'exemple d'utilisation fourni pour apprendre à l'utiliser.

Écrivez une (et une seule méthode) permettant aux contrôles représentant la couleur des joueurs (un `Rectangle` par exemple) d'ouvrir le `ColorPickerDialog` lorsqu'on clique dessus. Permettez ainsi de modifier la couleur d'un joueur.

MessageBox

Bouton Start

On veut démarrer le jeu lorsqu'on appuie sur `Start`. Néanmoins, il faut avoir remplir certaines conditions pour cela. Lorsque les conditions ne sont pas remplies, l'appui sur le bouton `Start` lancera un `MessageBox` indiquant aux joueurs pourquoi la partie ne peut pas commencer.

- Le nombre de joueurs doit être compris dans les bornes des règles du jeu choisies. Nous considérons que les joueurs sans nom ne sont pas des joueurs. Vérifiez que le nombre de joueurs est valide.
- Vérifiez que tous les joueurs ont des noms différents.
- Vérifiez que tous les joueurs ont des couleurs différentes.

Si toutes les conditions sont remplies, le jeu devra se lancer (semaine 14). En attendant, ouvrez une `MessageBox` indiquant que le jeu se lancera ... en 2011.

[11] http://blogs.msdn.com/b/wpfsdk/archive/2006/10/26/uncommon-dialogs--font-chooser-and-color-picker-dialogs.aspx

Partie 21 : interface graphique améliorée

XAML, WPF, code-behind, UserControl

OBJECTIFS

L'objectif de cette partie est d'améliorer la préparation du jeu à l'aide d'un UserControl pour la préparation d'un joueur. Cette partie est optionnelle.

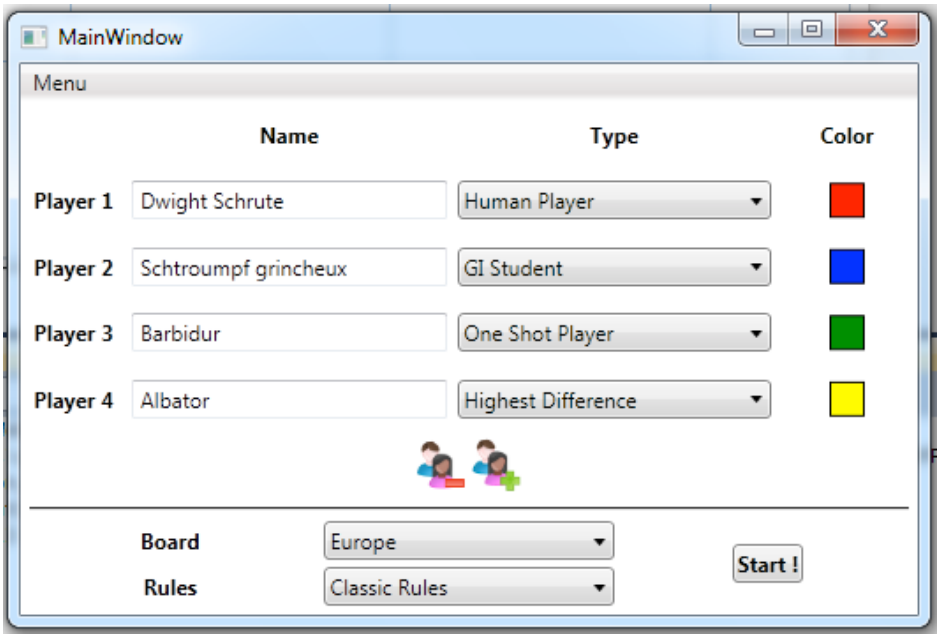
Les objectifs pédagogiques sont :

- la création et l'utilisation d'un UserControl,
- la liaison de ce UserControl dans l'Application et la fenêtre principale.

PRÉPARATION

On veut réaliser une interface graphique semblable à la précédente mais avec deux améliorations :

- le nombre de joueurs n'est pas de 5 mais variable,
- on factorise la définition des joueurs à travers un UserControl.



UserControl

UserControl pour la préparation d'un joueur

Ajoutez au projet de la partie précédente un `UserControl` `PlayerSettingControl`. Ce contrôle correspondra à une ligne de la grille des joueurs avec : un texte : *Player i* ; une `TextBox` avec le nom du joueur ; une `ComboBox` avec le type du joueur ; un `Rectangle` (ou un bouton) avec la couleur du joueur.

UserControl
propriétés

Propriétés du UserControl

Les utilisateurs du `UserControl` (ie. les concepteurs de la fenêtre principale) auront accès aux propriétés publiques de notre `UserControl`, aussi bien en *code-behind* qu'en XAML. Ajoutez les propriétés suivantes (dans le *code-behind*) :

- `PlayerName` (en lecture seule) donnant accès au nom du joueur (contenu de la `TextBox`) ;
- `PlayerType` (en lecture seule) donnant accès au nom du joueur (contenu de la `ComboBox`) ;
- `PlayerId` (entier) dont le setter modifiera le contenu du `TextBlock` (par exemple, si `PlayerId` vaut 2, le `TextBlock` affiche *Player 2*) ;
- `PlayerColor`, dont le setter modifie la couleur du `Rectangle`.

ColorPickerDialog

Modification de la couleur du joueur

Créez une méthode abonnée au click sur le Rectangle pour que la couleur du joueur soit modifiée, en utilisant le ColorPickerDialog (cf. partie 19).

Application
UserControl
ComboBox

Type du joueur

On veut à nouveau proposer l'ensemble des types de joueurs dans la ComboBox à l'aide d'un chargement dynamique (classe `AILoader` de la semaine 12), comme dans la partie 19. Par contre, il ne serait pas très judicieux de réaliser le chargement de toutes les IAs dans chaque `UserControl`.

Pour cette raison, ajoutez un membre interne à la classe `App` (dans le *code-behind*) qui contiendra l'ensemble des types chargés grâce à `AILoader` (cf. Partie 19 pour plus d'explications).

Dans le constructeur du `UserControl`, utilisez cette propriété de `App`[12] pour remplir la `ComboBox`. N'oubliez pas de rajouter le type `Human Player` (comme dans la partie 19).

Image

Ajout et suppression de joueurs

On veut permettre d'ajouter ou de supprimer des joueurs. Pour cela, ajoutez 2 boutons[13] pour ajouter un joueur ou supprimer le dernier.

Écrivez une méthode qui ajoutera un joueur, c'est-à-dire :

- créera une nouvelle instance de votre `PlayerSettingControl` (le `UserControl` précédent),
- initialisera ses propriétés (`PlayerId` automatiquement en fonction du nombre de joueurs déjà créé, `PlayerName` à vide, `PlayerType` à `Human Player` et la couleur avec une couleur par défaut),
- l'ajoutera à une liste privée de votre fenêtre qui permettra de gérer l'ensemble de ces joueurs,
- l'ajoutera à la grille du milieu en lui rajoutant une ligne et en l'insérant correctement dedans[14].

Écrivez une méthode qui supprimera un joueur, c'est-à-dire :

- supprimera la dernière ligne de la grille du milieu,
- supprimera le `PlayerSettingControl` des enfants de la grille,
- le supprimera de la liste privée des joueurs.

On veut qu'à l'ouverture de la fenêtre ou à la modification des règles :

- si le nombre de joueurs est supérieur au nombre de joueurs autorisés par les règles, les joueurs en trop soient supprimés,
- si le nombre de joueurs est inférieur au nombre minimum de joueurs, des joueurs soient rajoutés.

Enfin, on veut que le bouton `Add` soit invisible quand le nombre de joueurs est au maximum autorisé par les règles (idem pour `Remove`).

Utilisez ces méthodes avec les boutons `Add` et `Remove` puis lors de l'ouverture de la fenêtre et des modifications des règles du jeu.

---

[12] vous pouvez atteindre `App` de la manière suivante : `Application.Current as App`

[13] vous pourrez trouver des icônes gratuites sur internet

[14] pour les propriétés `Grid.Row`, `Grid.Column` et `Grid.ColumnSpan`, vous pourrez utiliser par exemple la syntaxe suivante :

```
playerSettingControl.SetValue(Grid.RowProperty, 2);
```

(ajoute le `playerSettingControl` à la 3ème ligne de la grille à laquelle il appartient).

Semaine 14

Interface graphique pour le déroulement du jeu

OBJECTIFS

Les objectifs de la dernière semaine, du point de vue de l'application finale, sont :

- de créer une interface graphique permettant de jouer.

Les objectifs pédagogiques sont :

- la création de UserControl,
- la liaison entre le XAML et la bibliothèque principale à l'aide du «code behind».

Partie 22 : interface graphique

XAML, WPF, code-behind, UserControl, RoutedEvents

OBJECTIFS

L'objectif de cette partie est de permettre de jouer via une interface graphique.

Les objectifs pédagogiques sont :

- la découverte de XAML et de WPF,
- l'utilisation d'outils de `Layout` (`StackPanel`, `Grid`, `WrapPanel`...),
- l'utilisation d'outils graphiques (`ViewBox`, `Border`, `Image`...),
- l'utilisation de dialogues (`MessageBox`),
- l'utilisation des événements des différents contrôles XAML,
- le lien avec la bibliothèque principale de notre projet (grâce au code-behind),
- la création et l'utilisation de UserControl.

PRATIQUE

Dans la partie précédente, nous avons créé la fenêtre PlayerSettingsWindow qui permet de préparer le jeu. Ajoutez dans le même projet une fenêtre GameWindow qui permettra de jouer. Cette deuxième fenêtre pourra notamment utiliser des UserControl pour factoriser et simplifier le code. Vous pourrez dans un premier temps ne considérer que des joueurs humains, que le plateau européen et les règles du jeu classiques pour simplifier. Si le coeur vous en dit, vous pourrez rajouter le reste. Voici une solution proposée.

- Les deux fenêtres sont regroupées dans la même application App et peuvent l'utiliser pour dialoguer (en particulier les informations sur le jeu créé par la première fenêtre, le plateau de jeu et les règles choisis, les joueurs...).
- Vous pourrez ensuite créer votre fenêtre en deux parties : la partie de gauche représentera le plateau de jeu, et la partie de droite les informations sur les joueurs et le déroulement du jeu.
- Vous pourrez ensuite créer un UserControl (contrôle utilisateur) pour les informations sur les joueurs : celui-ci permettra de stocker l'identifiant, la couleur, le nom et le type du joueur, ainsi que son score, et le nombre de monastères et de conseillers restants.
- Vous pourrez garder une partie dans la grille de droite pour afficher les cartes du joueur en cours, les dernières cartes jouées par le joueur précédent, ainsi qu'un bouton pour valider


MainWindow
Menu
Name
Type
Color
Player 1
Dwight Schrute
Human Player
Player 2
Schtroumpf grincheux
GI Student
Player 3
Barbidur
One Shot Player
Player 4
Albator
Highest Difference
Board
Europe
Rules
Classic Rules
Start !
App
Game
Board
Rules
...
Monastery Control
GameWindow
Country Control
PlayerInfoControl
Cartes du joueur à sélectionner
Cartes jouées par le joueur précédent
England
Franken
Lothringen
Schwaben
Bayern
Burgund
Italien
Massif Central
Aragon
Player 1 Dwight Schrute (Human Player)
Score: 14
Monasteries: 18 Advisors: 5
Player 2 Schtroumpf grincheux (GI Student)
Score: 7
Monasteries: 16 Advisors: 8
Player 3 Barbidur (One Shot Player)
Score: 6
Monasteries: 16 Advisors: 7
Player 4 Albator (Highest Difference)
Score: 8
Monasteries: 17 Advisors: 5
19 cards left
Player 1 (Dwight Schrute) is playing...
Move !

son coup.

- Le plateau de jeu pourra être représenté par un UserControl (si vous faîtes les deux, vous pourrez créer une classe de base pour les deux (BoardControl) qui sera dérivée en EuropeanBoardControl et en GreekBoardControl). Les plateaux de jeu comprendront différents UserControl :
  - des MonasteryControl permettant de cliquer sur un cloître pour insérer ou enlever un monastère,
  - des CountryControl permettant de cliquer sur le sceau d'un pays pour y insérer un ou deux conseillers ou les enlever.
- Vous devrez bien sûr vous abonner au événements principaux du jeu (GameStarted, GameOver, PlayerMoved et PlayerNotified) pour que votre interface suive l'évolution du jeu.

Vous avez accès aux différentes ressources nécessaires (images) sur l'ENT.