Projet IHM – Simulation

Projet avec base de code fournie, à réaliser **en solo**. Durant les six séances (deux séances puis quatre fois une séance) du projet, l'organisation sera :

1. Un *tutoriel* (de deux séances) permettant de prendre le projet et le code en main, en suivant des instructions précises et du code pour la plupart fourni,
   Rendu du tutoriel : un fichier contenant les réponses aux questions **et** l'archive du projet à la fin du tutoriel.
2. Une phase de *projet* où vous devrez utiliser vos compétences pour réaliser un travail qui vous soit propre.
   Rendu : l'archive du projet.

Ces livrables sont à remettre sur le dépôt Moodle qui leur est consacré, au plus tard le **Lundi 31 mai à 7 : 55 le matin** afin qu'ils puissent être corrigés à temps. Attention, **tout plagiat sera considéré comme une fraude** et sévèrement sanctionné.

1 Sujet – Mise en Situation

Vous jouez le rôle d'une ingénieure ou un ingénieur en design et développement d'interfaces graphiques, venant d'être embauché dans une petite entreprise montée pour faire divers logiciels, dont du jeu vidéo. Ce qui suit est un monologue fictif, dans lequel votre directeur détaille votre nouvelle mission… Toute ressemblance avec une situation réelle serait assez inquiétante.

—

Bonjour, installez-vous, prenez un siège. J'étais à la recherche d'un concept pour lancer notre activité *casual gaming* pour *pitch*er notre boîte auprès d'investisseurs bien dotés, voyez, mais notre budget est extrêmement serré et il nous faut quelque chose de percutant, solide sans nous demander trop de *production value* dans les *assets* graphiques, son et vidéo. Donc j'ai revu *Sacré Graal* en VO dernièrement avec sa magnifique citation que je ne retrouve pas, attendez, voilà, *strange women lying in ponds distributing swords is no basis for a system of government*, et ça m'a frappé d'un coup. Arthur, en fait, c'est un gestionnaire qui fait avant tout de la politique économique, en plus de toute l'épopée, la geste et les amusements d'*heroic-fantasy* qui vont avec, et toute la tragédie, c'est qu'il n'a pas les moyens de mettre en œuvre la disruption du système qui lui permettrait d'avoir un pays moderne capable de trouver le Graal. Bon, il n'est pas totalement impossible que j'ai été influencé par mes cours à Science-Po'. Ou par Alexandre Astier. Mais bref, on peut faire quelque chose à base d'une simulation éco-politique qui soit capable d'attirer les fans de *Game of Thrones* et de Tolkien sans avoir les moyens nécessaires pour du RPG ou de la grosse action irréalisable. On ne va pas faire du *Civilization*, mais on peut faire du *Democracy* : c'est du jeu qui a l'air de se vendre, alors qu'il n'y a qu'un affichage de graphes avec des chiffres, c'est totalement idéal pour notre boîte. Et comme le *setting* est fantastique, pas d'objections de tel ou tel bord politique quand on examine les conséquences de financer les enchanteurs ou d'envoyer des chevaliers en quêtes, on évite le boycott des activistes divers qu'on aurait dans un jeu réaliste, et surtout on n'a pas à embaucher d'économiste pour que ça ait l'air vrai ; c'est beau.

Dans le détail, mon concept est de manipuler des valeurs numériques pour faire de la gestion. On veut un petit jeu simple, donc pas beaucoup de valeurs. D'abord les gens, avec trois groupes : les **paysans**, les **chevaliers**, les **barbares**, avec un pourcentage de satisfaction pour chaque ; les deux premiers devraient être à 100 et le troisième à 0 dans un monde idéal (mais le monde n'est pas idéal, sinon, pas de jeu). Ensuite, des indicateurs, comme le PIB et le taux de chômage dans un vrai truc réaliste. Tout ça indique la bonne santé ou non du royaume, et le prestige rapporte en plus de la gloire. Oui, parce que j'ai deux "monnaies" dans le jeu, avec une classique, l'argent en **pièces d'or** qui est dépensé ou apporté à chaque tour, et la **gloire** qu'il faut dépenser pour changer de politique, ce qui empêche qu'on change tout à chaque tout ; on est en tour par tour, d'ailleurs, c'est plus simple. Donc le prestige rapporte de la gloire, un peu. Ensuite, il y a les politiques, qui sont ce sur quoi le joueur agit pour faire changer le reste, il y en a trois types : les impôts, que tout le monde déteste et qui risquent de ruiner le pays mais qui apportent des sous pour financer le reste ; les quêtes qui font travailler les chevaliers et qui rapportent de la gloire ; et enfin des politiques qui permettent d'améliorer la situation mais qui réclament des sous. Sur tout ça, la difficulté est d'équilibrer les conséquences positives et négatives, avec les finances. Pour

ces politiques, j'ai pensé à un mécanisme qui permet à certaines d'arriver plus tard dans le jeu, pour qu'il y ait des nouveautés régulièrement. Et enfin, il y a des situations particulières, les bénéfices et les problèmes. Je vous ai mis tout ça en annexe.

Voilà. Bon, c'est assez compliqué et il y a quelques interactions difficiles, l'équilibrage est non trivial, il faut faire attention à ce que ça soit jouable mais *challenging*. Mais vous n'avez pas à vous préoccuper de ça, c'est réglé ; j'ai demandé au stagiaire de DUT de me coder tout ça, et il a fait un super boulot avec des objets, je crois ; il m'a fait deux classes différentes et d'autres classes statiques, c'est beau, vous verrez, on a même des fonctionnelles, quoi que ce puisse être. Bon, il m'a fait une interface en mode texte et il m'a dit que ça irait bien, mais c'est la partie qui pose problème. Tout est bien codé, mais honnêtement, je n'arrive pas à faire une partie ; taper des lignes dans un Minitel, c'est lourdingue et c'était bien au siècle dernier. Donc pour notre prototype, ça n'ira pas. Et c'est là que j'ai besoin de vous en urgence.

Il me faut un jeu à montrer, et pour ça, je veux une interface, des menus, de la souris, etc. Je veux du *design* ergonomique et moderne. Pour les images, je recruterai des infographistes intermittents à la fin, c'est la partie facile, donc n'en faites pas, mais je veux des éléments qui soient intuitifs, comme dans *Democracy 3*, voyez, où on a les indicateurs qui sont visibles à l'écran et où on voit les relations entre eux quand on passe la souris dessus, où on puisse les changer et voir l'historique en cliquant dessus, et ça fait comme un genre de graphe. Sachant qu'on va évacuer les choses compliquées comme élections et gouvernement, ça devrait être facile.

Bon, je compte sur vous pour me faire ça vite et bien.

—*En annexe : p. 14 - détails de la logique de jeu, p. 15 - inspiration pour le jeu.*

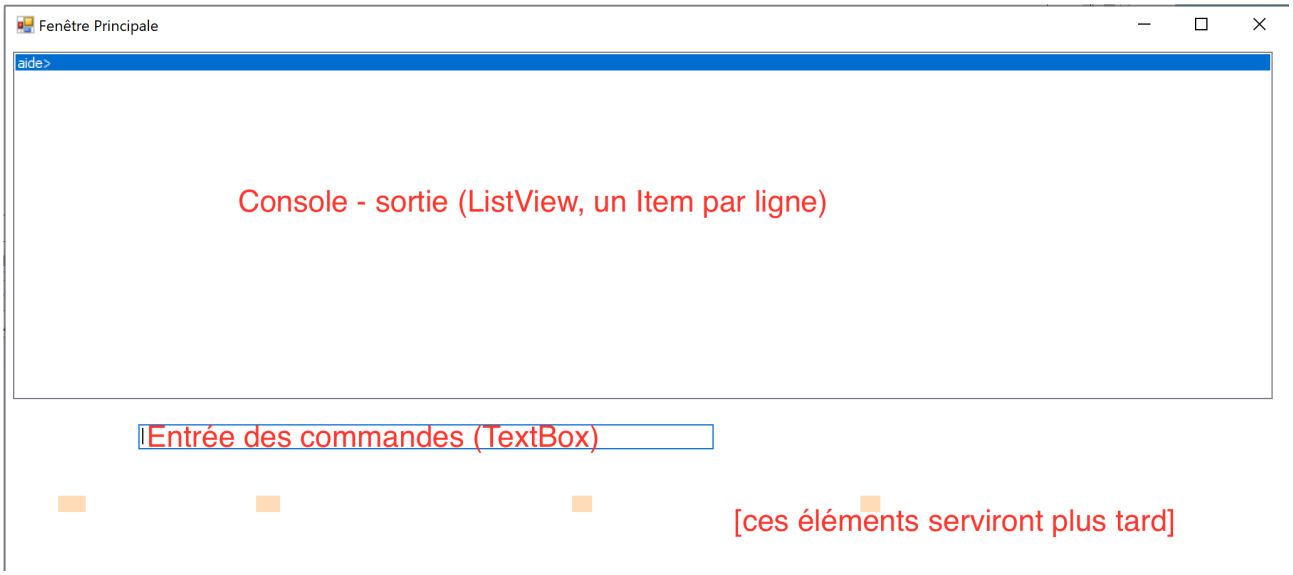
2 Tutoriel

2.1 Présentation de l'existant

Le programme fourni est une solution Visual Studio avec un seul projet. Il s'agit déjà d'une application Windows Forms (C#, .Net Framework 4.8 – à utiliser avec Visual Studio 2019, quelle que soit la version). Ouvrez la solution, générez-la et essayez-la.

L'interface est rudimentaire et comporte une `TextBox` pour entrer des commandes et une `ListView` pour voir le résultat – comme dans un terminal (l'application avait d'ailleurs été conçue pour la Console de Windows). Il suffit de taper des commandes (par exemple *aide*) et de valider par la touche Entrée. Il y a toujours une commande par défaut, *cmd*> dans la fenêtre de sortie, qui sera lancée si vous ne tapez rien (juste Entrée).

Pour l'instant, l'interface de jeu donne, par exemple en début de partie :



Vous êtes le dirigeant du royaume mythique de Logres ; vous disposez d'un budget, d'une situation et vous pouvez visualiser l'état de différents aspects du royaume, changer certaines politiques publiques, ou attendre. Et c'est tout.

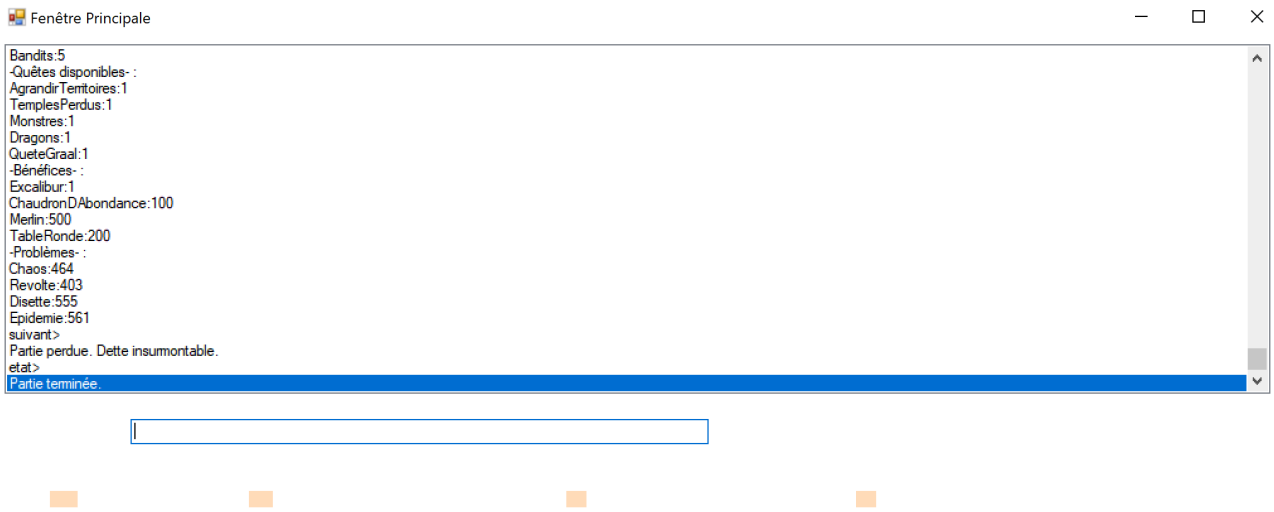
Dans le détail, les commandes à votre disposition sont :

— *aide* : donne de l'aide (argument optionnel : une commande),
— *etat* : affiche l'état (sans argument : général, avec argument : de quelque chose),
— *historique* : avec une valeur comme argument, affiche l'évolution de cette valeur depuis le début du jeu,
— *liste* : pour lister certains éléments (argument optionnel),
— *politique* : change une politique avec comme arguments le nom de la politique à changer et la nouvelle valeur (entre 0 et 100 ; 0 pour désactiver, 100 pour activer au maximum, tout autre chiffre : un pourcentage) – vous pouvez changer les politiques disponibles uniquement (dépenses publiques, impôts, quêtes),
— *suivant* : pour attendre et passer au tour suivant,
— *quitter* : pour quitter l'application (cliquer sur la croix marche aussi).

2.1.1 Un exemple de donne en mode texte

Testez l'implémentation du jeu en mode texte.

Pour visualiser ce qui se passe dans une partie très courte, tapez *suivant*, puis 22 fois sur Entrée. Le jeu conserve son état initial et déroule une partie dans laquelle, sans intervention, le royaume périclite (diverses crises surviennent) et la partie est perdue.



Pour une partie plus complète, si vous le souhaitez, voici la procédure pour *gagner*.

Relancez l'application et entrez d'abord :

— etat
— liste politiques
— liste taxes

Cela vous donnera l'état initial du royaume et ce qu'il est possible de changer. Entrez ensuite (en confirmant à chaque étape) :

— politique gardes 100
— politique pretres 100
— politique impots 40

Cela crée des dépenses pour améliorer les conditions de vie et des revenus pour faire face à ces dépenses. Entrez ensuite *suivant* et visualisez de nouveau la situation (*etat*, *liste*).

Tapez ensuite :

— politique subventions 100
— politique doleances 100
— politique quetegraal 10
— suivant

Au début, de nouvelles politiques deviennent disponibles régulièrement. Poursuivez avec :

— politique ecoles 100
— politique enchanteurs 100
— politique taxeluxe 10
— suivant
— politique theatres 100
— politique taxealcool 5
— politique agrandirterritoires 2
— politique monstres 2
— suivant
— suivant
— politique thermes 100
— politique juges 100
— politique taxefonciere 5
— politique dragons 5

Une fois toutes ces politiques définies, vous pouvez laisser le champ vide et taper sur Entrée en continu pour visualiser l'état actuel du monde simulé et passer le tour plusieurs fois (ou juste laisser *suivant*) pour passer vite). Vous gagnerez au tour 26 :

Explication : en mode *facile* (la partie est créée en mode facile par défaut), la trésorerie de départ est énorme et permet de financer toutes les dépenses en minimisant les impôts ; cette difficulté offre aussi de multiples avantages qui facilitent les choses. Dans les autres modes de difficulté, il faudrait jongler entre les priorités dans les dépenses, les recettes et le budget.

2.2 Organisation du code

Le code est découpé en trois grandes parties.

2.2.1 Le modèle

Le *modèle* est la *logique métier* du jeu. Il contient tous les éléments du monde simulé d'une manière purement abstraite. Les fichiers du modèle sont `FunLibrary.cs`, `IndexedValue.cs`, `WorldState.cs` – pour faire simple, un unique objet `WorldState` contient les informations d'une partie en cours et est associé à un grand nombre d'objets `IndexedValue`, qui contiennent chacune une valeur du monde simulé. Une `IndexedValue` fait appel à certains objets de `FunLibrary` pour des calculs.

Vous ne devez jamais modifier les fichiers du modèle (même si vous estimez qu'ils sont mal codés, contiennent des erreurs ou devraient être améliorés).

Le modèle s'appuie sur un fichier XML externe pour construire l'état initial de la simulation.

2.2.2 La vue

La *vue* contient l'*interface utilisateur* : ce qui est visible à l'écran, permet de communiquer l'état du jeu (événements résultats) et à l'utilisateur d'effectuer des actions (événements externes). Elle contient pour l'instant le fichier `GameView.cs`. Il n'y a normalement qu'une instance de `GameView` à chaque instant à l'écran.

Ce projet consiste surtout à modifier l'interface ; vous devrez modifier cette partie et y ajouter des fichiers.

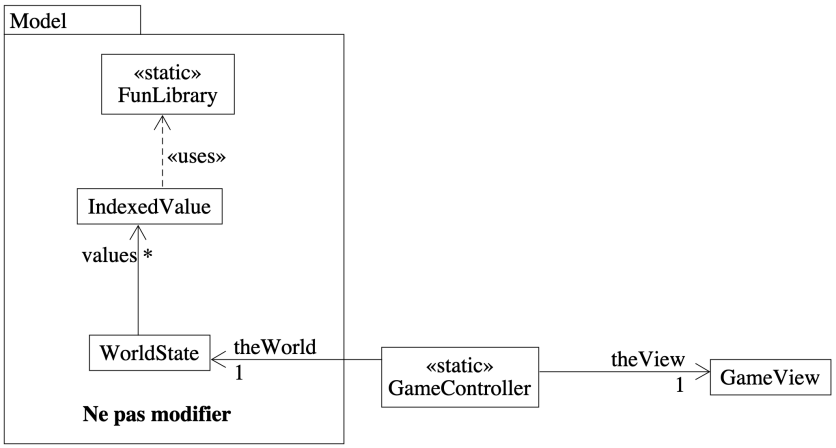
2.2.3 Le contrôleur

Le *contrôleur* sert à mettre en relation vue et modèle, en conservant l'indépendance de ce dernier. Ici, le fichier `GameController.cs` sert de contrôleur sous la forme d'une classe statique (ne comportant que des attributs, propriétés et méthodes de classe) ; ce n'est absolument pas la seule façon de faire : **l'architecture Modèle-Vue-Contrôleur (MVC) sera étudiée en détail en deuxième année**, ceci n'est qu'une première approche.

Le contrôleur devra être mis à jour pour utiliser les fonctionnalités que vous ajouterez dans l'interface. Le fichier `Program.cs` qui sert de point d'entrée aux applications C# pourra aussi être modifié.

2.2.4 Diagramme de classes

Un diagramme de classes approximatif et général :



2.3 Questions préalables

Votre premier objectif est de comprendre le fonctionnement général du projet. Il y a beaucoup de code, dont des passages obscurs, des détails peuvent vous échapper, il est même possible qu'il y ait des erreurs de programmation, d'algorithmique ou de mathématiques ; néanmoins, ne modifiez **pas** le code fourni. En particulier, il vous est demandé de faire confiance assez aveuglément aux données et au code des classes `IndexedValue` et `WorldState`, bien qu'il faille savoir quand employer les propriétés et méthodes publiques qu'elles fournissent. Essayez d'avancer efficacement et donnez-vous des objectifs intermédiaires raisonnables.

Question 1 *Combien y a-t-il de façons de gagner, de perdre ? À quel endroit dans le code déclenche-t-on la victoire et la défaite ?*

Question 2 *À quoi correspond la propriété* `DefaultCommand` *de la classe* `GameController` *?*

2.4 Début des modifications

Commencez par les bases.

Exercice 1 *Renommez la solution* ***en incluant votre nom****, et organisez-vous (par exemple en utilisant* gitlab*) pour* ***toujours disposer d'une sauvegarde à jour*** *de votre projet – il est* ***essentiel*** *que vous ne perdiez pas du travail d'une séance sur l'autre, quelles que soient les circonstances de ce travail.*

2.5 Choix de la difficulté

Pour l'instant, la partie est lancée en mode Facile. Le but est de permettre de sélectionner entre Facile, Moyen, Difficile en début de partie – *avant* son lancement. Un fichier `DifficultyView.cs` est déjà prévu à cet effet.

Exercice 2 *Décommentez les lignes 22 à 24 de* `Program.cs`*, puis essayez.*

Question 3 *Comment la difficulté est-elle choisie et pourquoi le choix* Difficile *ne fonctionne-t-il pas ?*

Question 4 *Comment la boîte de dialogue se ferme-t-elle et pourquoi le bouton* Difficile *ne permet-il pas de la fermer ?*

Exercice 3 *Rectifiez le problème en rendant fonctionnel le bouton* Difficile*.*

2.6 Première amélioration de l'interface principale

Il est assez simple d'ajouter des éléments d'interface permettant de visualiser l'état global : difficulté, numéro de tour, trésor, gloire (les informations qui s'affichent en appuyant sur *état*. Une grande partie du code nécessaire est déjà positionné dans `GameView.cs`. Le principe : des étiquettes sont là (avec un fond coloré et sans contenu pour l'instant) pour afficher ces informations, et un bouton est utilisé pour passer au tour suivant (au lieu de devoir taper *suivant*).

Exercice 4 *Décommentez le code de l'événement* Paint *de* `GameView.cs`*. Cherchez quelles propriétés de l'objet* `theWorld` *sont à utiliser (en lecture !) pour cet affichage et remplacez les* ? *par ces valeurs, elles sont mises à jour à chaque rafraichissement. Vérifiez le résultat.*

Attention : il faut désormais penser à rafraîchir la fenêtre lors d'un changement d'état pour tenir le joueur informé de l'évolution de la situation. Des appels à `theView.Refresh()` (qui déclenche l'événement 'Paint') sont prévus à cet effet dans le *GameController*, n'oubliez pas d'ajouter les vôtres là où ce sera nécessaire.

Question 5 *Lors du passage au tour suivant (la commande tapée est* suivant*), comment* GameView *et* GameController *interagissent-elles pour faire le passage au tour suivant (quelles sont les méthodes appelées, quels sont leurs effets) ?*

À partir de ce moment, il faudra probablement modifier le `GameController`.

Exercice 5 *Rendez visible le bouton `nextButton` et faites en sorte qu'il permette de passer au tour suivant, vérifiez le résultat.*

Exercice 6 *Faites en sorte que la partie s'arrête avec des boîtes de dialogue (en plus des messages de la `ListBox` qui disparaîtront bientôt). Voici par exemple du code pour afficher dans `GameView` un échec :*

```
public void LoseDialog(IndexedValue indexedValue) {
    if (indexedValue==null) {
        MessageBox.Show("Partie perdue : dette insurmontable.");
    }
    else {
        MessageBox.Show("Partie perdue :
        +indexedValue.CompletePresentation());
    }
    nextButton.Enabled=false;
}
```

Modifiez `GameController` pour que ceci fonctionne, et ajoutez aussi un code similaire pour la victoire.

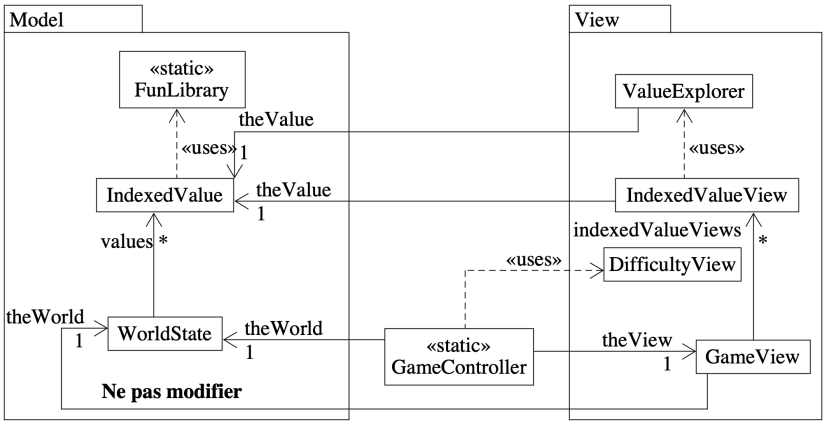
3 Projet

3.1 Création d'une interface complète

Après cette partie *tutoriel*, votre but sera de créer une interface *sans commandes texte*, en utilisant des éléments graphiques si possible lisibles et ergonomiques. Vous pouvez supprimer la `ListBox` et la `TextBox` (ou les conserver en plus petit tant que vous n'avez pas quelque chose de pleinement fonctionnel).

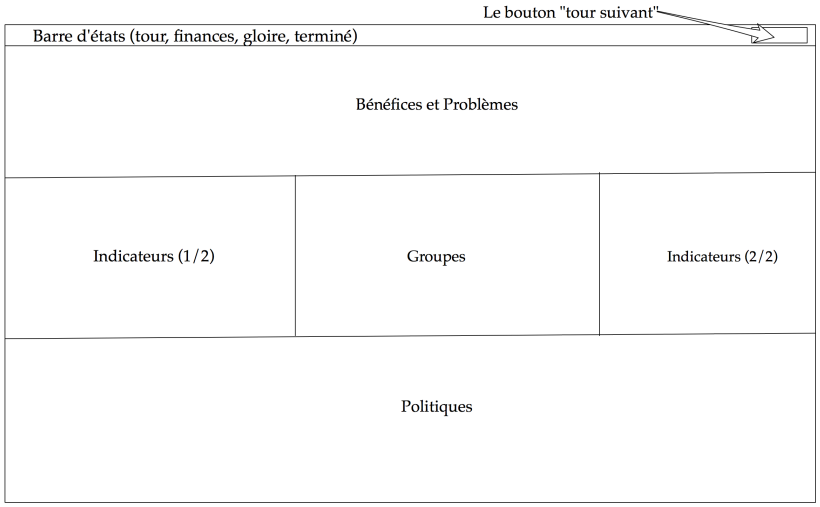
La première tâche est d'utiliser la fenêtre pour afficher une représentation graphique des éléments pertinents, c'est à dire les valeurs.

Le diagramme de classes du projet ressemblera bientôt à ceci :



Exercice 7 *Créez une nouvelle classe `IndexedValueView` dans un fichier impérativement nommé `IndexedValueView.cs`. Cette classe contiendra une `IndexedValue`, plus de quoi l'afficher à l'écran (suggestion : coordonnées, taille pour un rectangle, éventuellement couleur·s et épaisseur). Elle nécessite au moins un constructeur et une méthode de dessin (`Draw` ou similaire).*

Comment positionner les 46 valeurs ? Une suggestion de maquette est donnée dans le schéma suivant :



Si nous avons une fenêtre 2100x900, les coordonnées des zones sont alors :
— Bénéfices et Problèmes : 0,0 ; 2100x300 (moins la barre),
— Indicateurs : 0,300 ; 700x300 et 1400,300 ; 700x300
— Groupes : 700,300 ; 700x300
— Politiques : 0,600 ; 2100x300
Il faut bien sûr adapter selon votre design (vous pouvez avoir d'autres idées) et votre fenêtre (l'idéal serait de calculer ces coordonnées en fonction de la largeur et de la hauteur de la fenêtre).

Exercice 8 *Avant de commencer à poser ces éléments, préparez votre propre maquette et déplacez les éléments déjà présents (étiquettes et boutons) à l'endroit souhaité (par exemple barre d'outils, barre d'états ou ailleurs). Redimensionnez la fenêtre si vous le souhaitez.*

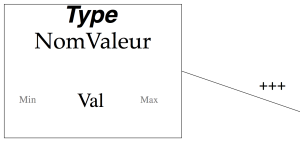
Exercice 9 *Construisez une (ou plusieurs) listes de* `IndexedValueView` *en leur affectant à chacun une coordonnée raisonnable, à partir des listes de* `IndexedValue` *de l'instance de* `WorldState` *disponibles au travers des propriétés publiques de cette classe.*

Exemple : en supposant l'utilisation de carrés de taille 80x80 pixels et d'une marge de 10 pixels pour chaque valeur, le code suivant affecte une coordonnée (coin en haut à gauche) à toutes les politiques (tant qu'il y en a vingt ou moins) :

```
// PolRectangle:0,600,2100,300; w:80, h:80, margin:10
int x = PolRectangle.X + margin, y=PolRectangle.Y + margin;
List<IndexedValueView> polViews = new List<IndexedValueView>();
foreach (IndexedValue p in theWorld.Policies) {
    polViews.Add(new IndexedValueView(p, new Point(x, y);
    x += w+margin;
    if (x>PolRectangle.Right) {
        x=PolRectangle.x;
        y += h+margin;
    }
}
```

Il suffit maintenant de dessiner de manière adaptée chaque valeur. Voici une suggestion :

Exercice 10 *Complétez la méthode de dessin de la classe* `IndexedValueView`*, ainsi que la réponse à l'événement* `Paint` *de la fenêtre principale, et affichez l'ensemble des valeurs.*



Vous devriez voir chaque valeur définie affichée à l'écran. Si vous affichez au moins le nom et la valeur numérique, le jeu devrait pouvoir être jouable. Ou presque, il faut permettre de modifier les politiques.

Exercice 11 *Créez un dialogue simple de modification (suggestion : une Windows Form utilisant un* `NumericUpDown` *ou une* `TextBox`*, un bouton* `Ok` *et un bouton* `Annuler`*) permettant de modifier une politique quand on clique dessus (réponse à l'événement* `MouseDown`*). Mettez en œuvre les méthodes publiques des classes fournies en vous inspirant de* `GameController.ApplyPolicyChanges`*. (Suggestion : si vous avez besoin d'émettre un message de confirmation ou d'erreur, utilisez la classe .Net* `MessageBox`*.)*

Vérifiez que les fonctionnalités sont bien présentes comme précédemment. Un des problèmes est que, parmi les valeurs, certaines sont actives (`val.Active !=false`), certaines sont inactives (`val.Active==false`) et certaines politiques peuvent aussi être inactives, mais disponibles (`val.Active==false && val.AvailableAt <= Turns`). Il faudrait les afficher différemment (en changeant la couleur, leur épaisseur, voire leur `StrokePattern` ou leur opacité, ou simplement en ne les affichant plus), et ne permettre de modifier que les politiques actives ou disponibles à ce tour.

Exercice 12 *Afficher différemment les valeurs en fonction de leur état. Ne pas permettre de modifier une valeur quand il s'agit d'autre chose que d'une politique active ou disponible.*

On pourrait aussi afficher des formes différentes suivant le type de la valeur.

Complément 1 *Difficulté : variable, temps : variable. Utiliser des formes différentes (rectangle, cercle…ou encore losange, triangle, polygones divers) pour différencier les valeurs (les politiques, qui sont modifiables, des autres valeurs, qui sont en lecture seule, par exemple).*

Complément 2 *Difficulté : simple, temps : assez rapide. Si l'interface n'est pas au goût de l'utilisateur, il pourrait la modifier lui-même : ajoutez la possibilité de repositionner ces valeurs "à la main" (déplacement).*

Pour les valeurs autres que les politiques, on peut souhaiter afficher leur état complet.

Exercice 13 *Permettre, lors du clic sur une valeur autre qu'une politique, d'afficher leur description (suggestion : une simple* `MessageBox`*).*

Ce qui n'est absolument pas fait dans la version "console" de ce jeu et qui reste extrêmement utile, c'est l'affichage des **liens**, c'est-à-dire des valeurs influencées par chacune des valeurs. Il y en a beaucoup (dans le fichier XML fourni représentant le monde, 314…). Il est donc indispensable de **n'afficher que les liens d'une valeur à la fois**, pour cela, on ne considère que la valeur sur laquelle passe le pointeur de la souris à un instant t.

Exercice 14 *Repérez avec l'événement* `MouseMove` *la valeur sous laquelle se trouve la souris et sauvegardez-là (dans un attribut/une propriété). Établissez également la liste des valeurs auxquelles cette valeur contribue (elles apparaissent en clef dans la propriété* `OutputWeights` *de la valeur sélectionnée).*

Il faut ensuite afficher ces liens. Ils ont quelques propriétés différentes :
— lien positif (la valeur dans `OutputWeights` est > 0) ou négatif (< 0),
— lien mineur, moyen ou majeur (la valeur dans OutputWeights est de l'ordre de $0,00x$, $0,0x$ ou $0,x$).

Il y a différents moyens d'envisager l'affichage de chaque trait, en variant sa couleur, son épaisseur, sa `StrokePattern`, ou (mais c'est plus difficile) en ajoutant des étiquettes (suggestion : un disque contenant un symbole tel que $+, -, >, <$ au milieu du trait, qui est facile à calculer).

Exercice 15 *Réalisez l'affichage des liens : toutes les valeurs autres que celle sélectionnée et celles qui y sont liées doivent disparaître (ou être affichées avec une opacité inférieure à 100%), des traits entre la valeur sélectionnée et celles qui y sont liées doivent rendre explicites la nature de ces liens.*

3.2 Design d'un dialogue spécifique

Si vous avez réalisé l'ensemble des exercices précédents, vous avez répondu au cahier des charges. Mais on peut aller un peu plus loin... Le but de cette partie est que vous réalisiez la maquette (*design*) complet d'un formulaire permettant l'exploration complète d'une valeur, et sa modification. Il devrait y avoir toutes les informations disponibles pour l'état d'une valeur (voyez `CompletePresentation`), mises en forme avec des contrôles adaptés. On souhaite aussi afficher les liens sur les autres valeurs dans ce dialogue, par exemple sous la forme d'un histogramme : une ligne par valeur affectée, avec une barre vers la gauche ou la droite suivant que l'influence est positive ou négative. Dans le cas d'une politique, on souhaite effectuer une modification avec pré-visualisation en temps réel du coût (argent, gloire) s'affichant à mesure que la valeur visée est modifiée à l'aide d'un contrôle adapté (`NumericUpDown` ou réglette coulissante, par exemple).

Exercice 16 *Effectuez le design de cette fenêtre de dialogue améliorée, puis créez-la dans un ficher impérativement nommé `ValueExplorer.cs`. Testez.*

Parmi les fonctionnalités supplémentaires, on peut également afficher l'historique d'une valeur. Cette fonctionnalité est gérée en mode console par des chaînes de caractères.

Complément 3 *Difficulté : assez complexe, temps : assez long. Ajoutez à votre fenêtre l'affichage de l'historique de la valeur sous forme de graphique (suggestion : une zone de taille fixée, deux axes tracés par du code avec les tours en abscisses et la valeur du minimum au maximum en ordonnée, une ligne segmentée).*

Il y a aussi deux propriétés qui peuvent être détaillées ici : le coût et l'effet.

Complément 4 *Difficulté : complexe, temps : assez long. Profitez du fait que vous ayez défini de quoi afficher des graphiques pour afficher ceux des fonctions de coût et d'effet dans le domaine* $[-1,+1]^2$.

3.3 Finalisation des Fonctionnalités

La dernière partie concerne des fonctionnalités additionnelles rendant l'application plus conviviale. Faites-en autant (ou aussi peu) que vous le souhaitez (il vaut mieux avoir fait parfaitement tous les exercices que d'avoir tenté un complément). Certains compléments peuvent nécessiter d'analyser le code fourni.

Complément 5 *Difficulté : très simple, temps : très court. Ajoutez un mode de jeu "nombre de tours limités" (un attribut de `WorldState` est prévu avec un paramètre optionnel).*

Complément 6 *Difficulté : assez simple, temps : assez long. Ajoutez un écran de préférences et un contrôle (bouton ou article de menu) permettant de définir le fichier de configuration à charger et la difficulté par défaut (avec un item affichant le dialogue de choix de la difficulté systématiquement). À noter qu'il faudra un fichier de préférences (d'application), en plus du fichier de configuration (du monde).*

Complément 7 *Difficulté : simple, temps : assez long. Permettez de changer les paramètres de l'affichage en permettant de choisir formes et couleurs pour chaque état/catégorie de valeur.*

Complément 8 *Difficulté : très variable, temps : très variable. Changez l'affichage pour le rendre dynamique et plus représentatif. Suggestion : changer la taille, l'épaisseur ou la couleur des "vues" des valeurs en fonction de leur `Impact` (normalisé entre -1 et +1) ; animer les liens avec des ronds glissant le long, d'autant plus nombreux et/ou rapides que le lien est important.*

Complément 9 *Difficulté : assez simple, temps : moyen. Réalisez un vrai historique des valeurs pour pouvoir revenir en arrière d'un tour.*

Complément 10 *Difficulté : simple, temps : moyen. Ajoutez des fonctionnalités de "triche" dans un menu, avec le retour en arrière (s'il est fait) et de quoi changer les valeurs (arbitrairement.*

Annexes

Quelques compléments de C#

Le code fourni utilise quelques aspects amusants de C# (.Net Framework 4.8 ou plus, entres autres). Il n'est pas nécessaire de tout connaître. En vrac :

Paramètres en sortie

Les types de base peuvent être rendus *modifiables en entrée et sortie* de méthode avec le mot-clé `ref`, et en sortie seule avec le mot-clé `out`.

```
private void increment(ref number, out old) {
  old = number;
  number++;
}
```

Paramètres optionnels

Parfois, la valeur de certains paramètres peut être quasiment toujours la même, sinon, il faut la spécifier. C'est faisable avec des paramètres optionnels (la valeur par défaut est donnée dans la signature).

```
private void doThing(int val=0) { ... }
doThing(); // call w/ val being 0
doThing(7); // call w/ val as the argument
```

Constantes

Les constantes en C# sont dénotées par le mot-clé `readonly` (équivalent de `final` en Java ou `const` en C ou certains de ses dérivés). Attention, comme en Java, les objets sont des références, ce qui signifie que, dans le code suivant :

```
private readonly List<Thing> list = new List<Thing>();
```

…la constante est la *référence à la liste* (impossible de changer l'emplacement de cet objet), son *contenu* peut, lui, être modifié (par `Add`, `Remove`, `Clear`…).

Définitions abrégées

Pour une propriété donnant un accès public en lecture à un champ donné, on peut écrire par exemple :

```
private Thing exampleField;
public Thing ExampleField { get { return exampleField; } }
```

Cette structure étant très fréquente, C# permet d'abréger (un peu) en :

```
private Thing exampleField;
public Thing ExampleField => exampleField;
```

De même, certaines méthodes n'ont qu'une ligne, par exemple :

```
public void m(Thing arg)
{
    doThing(arg);
}
```

Dans ce genre de cas, les lignes d'accolades sont superflues et on peut écrire :

```
public void m(Thing arg) => doThing(arg);
```

Vérification de non-nullité

Un idiome assez fréquent en programmation objet, quand des références peuvent être nulles, consiste à vérifier que ce n'est pas le cas avant d'appeler une méthode sur l'objet référencé. Par exemple :

```
if (myObject != null)
{
    myObject.m();
}
```

Ici, on peut plus simplement écrire :

```
myObject?.m();
```

Types nullables

Équivalent de `Optional` en Java, par exemple, un élément qui peut représenter une valeur entière (le maximum d'une liste) ou non (si la liste est vide)...

```
int? maybeVal; // déclaration
maybeVal = list.Count>0 ? list.Max() : null; // exemple d'initialisation
if (maybeVal.HasValue) // test, peut aussi s'écrire maybeVal != null
{
    return maybeVal.Value; // utilisation : le retour est un int et non un int?
}
```

Rejet

Indique qu'une méthode a un effet et une valeur de retour, mais qu'on souhaite ignorer cette dernière. Permet de préparer du code pour le moment où elle sera utile. Exemple : 'Remove' retire un élément d'une liste et on peut se contenter d'écrire `list.Remove(e);`, mais elle permet aussi de savoir si cet élément a bien été retiré en renvoyant un booléen. Si on souhaite noter ce fait mais en ignorant cette valeur de retour, on écrit :

```
_ = list.Remove(e);
```

On peut également capturer une exception sans utiliser la valeur de cette dernière, il suffit d'écrire :

```
try ...
catch (Exception)
...
```

Types fonctionnels

On peut sauvegarder des λ-expressions ou des références de méthodes en utilisant des variables de types fonctionnels. Le type `Action<T>` correspond ainsi aux méthodes prenant un `T` en paramètre et ne renvoyant aucune valeur. Par exemple, on peut utiliser un dictionnaire (équivalent d'une `Map` Java : un tableau associatif) pour stocker une association entre une commande (chaîne de caractères) et la méthode à appeler quand la commande est entrée. Commentaire de code :

```
Dictionary<string, Action<string>> commands;
commands = new Dictionary<string, Action<string>>();
                // déclaration et initialisation
commands.Add("help", DisplayHelp);
               // la méthode doit avoir pour signature void DisplayHelp(string s)
commands.Add("quit", _=>Quit());
               // si une commande n'a pas besoin d'argument,
               //on y associe une méthode qui ignore cet argument
...
commands[cmd](arg); // appel de la commande correspondant à la chaîne cmd
               // avec la chaîne arg comme argument
```

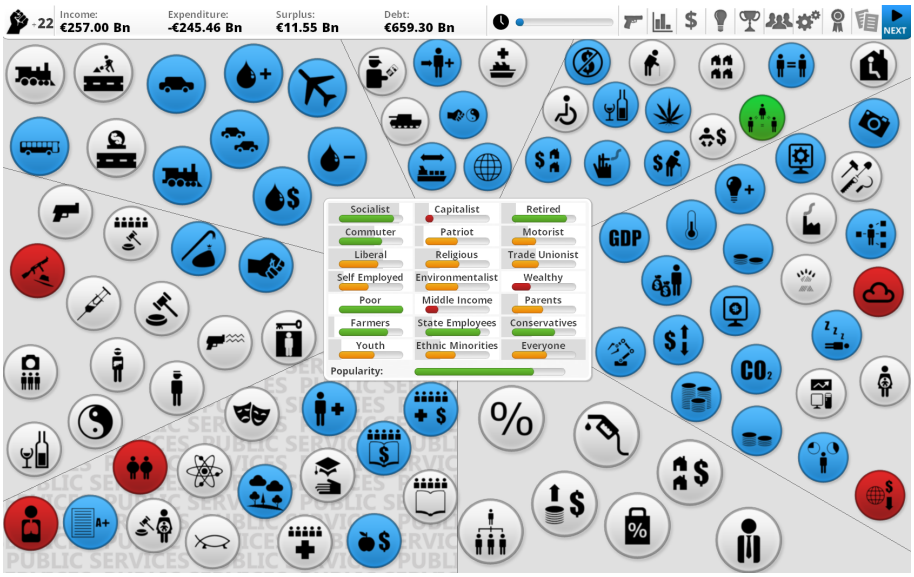
Détails de jeu – modèle

Code (EN)	Texte (FR)	Qu'est-ce	Liste
Glory	Gloire	Capital politique (limitant les décisions par tour)	
Money	Trésor	Capital financier (dépensé et gagné)	
Policy	Politique	Décision pouvant être prise par le joueur (associée à un pourcentage)	**Dépenses** : politiques générales permettant d'améliorer des situations, mais qui coûtent de l'argent (trésor). `subventions`, `ecoles`, `theatres`, `thermes`, `gardes`, `juges`, `doleances`, `enchanteurs`, `pretres`. **Taxes :** politiques permettant d'avoir des recettes, mais pouvant causer des difficultés. `impots`, `taxeLuxe`, `taxeAlcool`, `taxeFonciere`, `emprunts`. **Quêtes** : politiques permettant d'avoir des résultats spectaculaires, mais couteuses et exigeantes. `repousserBarbares`, `agrandirTerritoires`, `templesPerdus`, `monstres`, `dragons`, `bandits`, `queteGraal`.
Group	Groupe	Indicateur de popularité chez un groupe de personnes.	Les Paysans et les Chevaliers devraient être au maximum, les Barbares au minimum.
Indicator	Indicateur	Valeur permettant de visualiser l'état d'un aspect précis de la population entre un minimum et un maximum donnés.	`pauvrete`, `education`, `hygiene`, `egalite`, `criminalite`, `prestige`.
Perk	Bénéfice	Situation positive ou artefact donnant de grands avantages.	`abondance`, `ageDOr`, `securite`, `bonneSante`, `legende`, `Graal`, `excalibur`, `chaudronDAbondance`, `lanceDeLug`, `Merlin`, `tableRonde`. **Attention** : l'activation du *Graal* est le seul moyen de gagner la partie.
Crisis	Crise	Problème grave à régler.	`Chaos`, `Revolte`, `disette`, `epidemie`, `malediction`, `Mordred`. **Attention** : la crise *Mordred* fait perdre la partie.

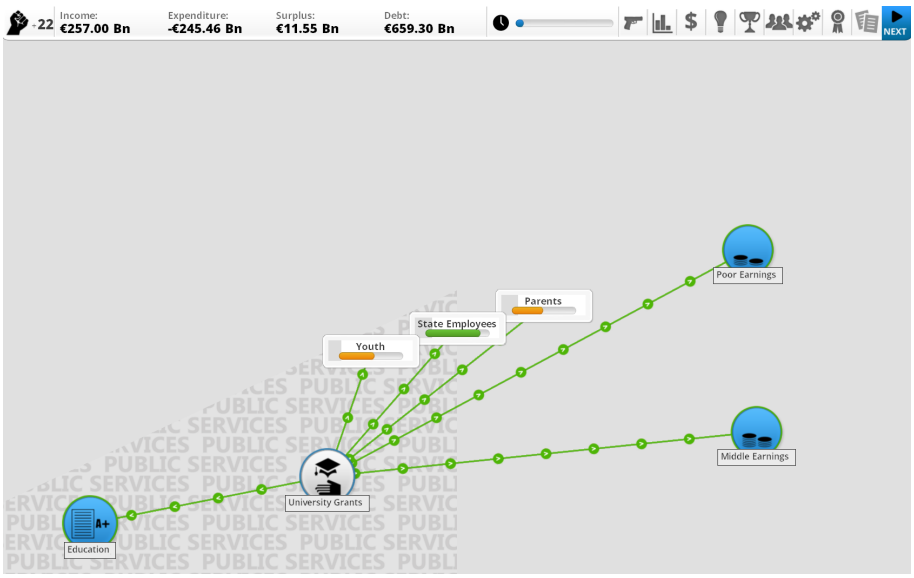
Quelques captures d'écran

Ce sujet est honteusement inspiré de *Democracy 3*. Sans conseiller l'achat de cette œuvre vidéoludique (honnêtement destinée à un public de niche), voici quelques captures d'écran du jeu. Il n'est absolument pas attendu d'arriver à ce niveau d'interface, il s'agit juste un exemple. De nombreux *let's play* sont disponibles sur internet pour plus de précisions, dont par exemple `https://www.youtube.com/watch?v=pdps5ZIdzEw` (attention, ce type de jeu entraine souvent des commentaires politiques – la simulation n'est pas forcément réaliste et les opinions des testeurs n'engagent qu'eux, notre seul but est de faire découvrir l'interface).

Le scénario est le suivant : le joueur dirige la France et examine son état - les éléments en bleu sont les *indicateurs* (valeurs visibles mais non modifiables directement), en rouge, les *crises* (valeurs visibles ayant un effet négatif, pouvant s'activer ou se désactiver), en vert, les *bénéfices* – et au centre dans une interface différente, les *groupes* :



Il examine ensuite précisément les effets du financement public de l'université (en survolant l'icône correspondante), les flèches vertes sortantes indiquent un effet positif sur certains éléments (il pourrait aussi y avoir des flèches rouges – effet négatif, ou noires – effet négligeable, ainsi que des flèches entrantes) :



Il ouvre alors un dialogue permettant de maximiser ce financement (accessible en cliquant sur la même icône) :