INSA RENNES

Projet Modélisation et Programmation Orientées Objet
Quatrième année - Informatique

Éric Anquetil
eric.anquetil@insa-rennes.fr
Ferran Argelaguet
ferran.argelaguet@gmail.com
Arnaud Blouin
arnaud.blouin@insa-rennes.fr
Maud Marchal
maud.marchal@insa-rennes.fr
Grégoire Richard
gregoire.richard@irisa.fr

Institut National des Sciences Appliquées
Ecole publique d'ingénieurs

Table des matières

Jeu Small World

1 Préambule

Avant tout, à la moindre interrogation posez vos questions et remarques concernant le projet et son sujet sur le forum qui lui est dédié : `http://coursonline.insa-rennes.fr/mod/forum/view.php?id=5522` car toute question peut intéresser l'ensemble de la promotion. De plus, vous pouvez poser des questions techniques sur comment faire telle ou telle chose, ainsi que répondre vous-même répondre aux questions des autres étudiants, le but étant d'avoir un forum d'aide au développement du projet. Pensez également à lire les messages du forum de l'an dernier.

2 Principes et But du Jeu

Il s'agit d'un jeu tour-par-tour où chaque joueur dirige un peuple. Le but du jeu est de gérer des unités sur une carte pour obtenir le plus de points possible à la fin d'un certain nombre de tours. Pour ce faire, chaque joueur commence avec des unités placés sur une même case de la carte. Il doit ensuite les repartir au mieux sur la carte. Le placement de chaque unité sur les cases rapporte plus ou moins de points (par défaut une case rapporte 1 point, des bonus et des malus sont possibles, cf. la description des peuples). Les unités d'un joueur peuvent également attaquer les unités d'un autre joueur pour détruire des unités (limitant ainsi l'acquisition de points de l'adversaire) et occuper une case de la carte. Les points sont recomptés à la fin de chaque tour pour que tous les joueurs connaissent leur nombre courant de points ainsi que celui de leurs adversaires. Le jeu se déroule sur une carte du monde sur laquelle les unités se déplacent.

2.1 Règles du Jeu

Les peuples

Il existe trois peuples [1], Elf, Orcs et Nain, ayant des caractéristiques très différentes influant sur les stratégies de jeu :

1. Elf.
   - Le coût de déplacement sur une case *Forêt* est divisé par deux.
   - Le coût de déplacement sur une case *Désert* est multiplié par deux.
   - Une unité Elf a 50% de chance de se replier lors d'un combat (provoqué ou subit) perdu devant normalement conduire à la destruction de l'unité : l'unité survie avec 1 point de vie.
2. Orc.
   - Le coût de déplacement sur une case *Plaine* est divisé par deux.

1. Vous pourrez créer d'autres peuples si vous en avez le temps.

— Une unité Orc n'acquière aucun point sur les cases de type *Forêt*.
— Lorsqu'une unité Orc détruit une autre unité, elle possède alors 1 point de bonus permanent. Cet effet est cumulable et est lié à chaque unité (i.e. si l'unité ayant le bonus meurt, le bonus disparaît).

3. Nains.
— Le coût de déplacement sur une case *Plaine* est divisé par deux.
— Une unité Nain n'acquière aucun point sur les cases *Plaine*.
— Lorsqu'elle se trouve sur une case *Montagne*, une unité Nain a la capacité de se déplacer sur n'importe quelle case montage de la carte à condition qu'elle ne soit pas occupée par une unité adverse.

À chaque tour, toutes unités peuvent être déplacées et attaquer. Par défaut (c.-à-d. hors bonus), chaque unité possède un point de mouvement ce qui correspond à un déplacement normal (bonus ou malus exclus). Cela signifie qu'une même unité peut se déplacer ou attaquer plusieurs par tour grâce à ses bonus. Chaque unité possède 2 d'attaque, 1 de défense et 5 points de vie. Les unités ne récupèrent pas leurs points de vie à la fin d'un tour.

La Carte du Monde

La carte du monde se compose de cases hexagonales[2]. Il existe différents types de case : plaine, désert, montagne, forêt. Par défaut, une case rapporte 1 point. Une carte doit contenir le même nombre de cases de chaque type (pour ne pas avantager un peuple). Nous vous fournissons des images pour chaque type de cases. Si vous avez le temps et si vous le souhaiter, vous pouvez faire vos propres cases.
La carte sera créée en début de partie de manière aléatoire.

Indice de modélisation : utilisez *poids-mouche* pour minimiser le nombre d'instances de cases (*cf.* exemple du cours).

Il existe 3 types de cartes :
— Démo : 2 joueurs, 6 cases × 6 cases, 5 tours, 4 unités par peuples.
— Petite : 2 joueurs, 10 cases × 10 cases, 20 tours, 6 unités par peuples.
— Normale : 2 joueurs, 14 cases × 14 cases, 30 tours, 8 unités par peuples.

Les Combats

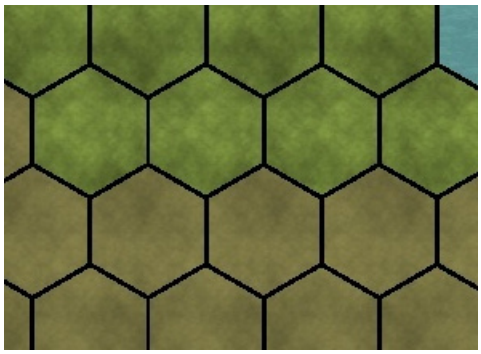
Le fonctionnement des combats ressemble plus à celui du jeu *Civilization* qu'à celui de *Small World*. Pour qu'une unité puisse lancer une attaque contre une unité d'un autre peuple, elles doivent se situées sur des cases juxtaposées. Lorsqu'une unité attaque une case contenant plusieurs unités, la meilleure unité défensive est choisie. Un combat se compose d'un certain nombre d'attaques. Ce nombre est choisi aléatoirement à l'engagement (entre 3 et le nombre de points de vie de l'unité ayant le plus de points de vie + 2 points). Le combat s'arrête lorsque ce nombre est atteint ou lorsque l'une ou autre des unités n'a plus de vie. Chaque combat calcul les probabilités de perte d'une vie de l'attaquant. Par exemple, Si l'attaquant a 4 en attaque et l'attaqué a 4 en défense (en tenant compte des bonus de terrain et du nombre de points de vie restant), l'attaquant à 50% de (mal-)chance de perde une vie. S'il a 3 att. contre 4 déf., le rapport de force est de 75% : 3/4 = 25%, 25% de 50% = 12.5%, 50%+12.5%=62.5% chance pour l'attaquant de perdre une vie. Explications du calcul : par défaut 2 unités égales ont 50% de gagner. Puisque dans le cas présent un écart de 25% est constaté entre les deux unités, il est nécessaire de pondérer le 50% par ces 25%

2. `https://en.wikipedia.org/wiki/Hex_map`
`https://fr.wikipedia.org/wiki/Hexagone`

ce qui donne 62.5% contre 37.5%. S'il a 4 att. contre 2 déf., le taux baisse à 25% (2/4 = 50%, 50% de 50% = 25%, 25%+50% = 75% pour l'attaqué, 100%-75%=25% pour l'attaquant). Évidemment, lorsque l'attaquant gagne cela signifie que l'adversaire perd un point de vie.
Les points de vie entrent en compte dans le calcul des probabilités : si une unité attaquante ayant 4 en attaque possède 50% de sa vie, alors son attaque sera au final de 4*50% = 2. L'unité attaquée suit le même calcul pour sa défense.
À la fin d'un combat gagné par l'attaquant et si la case du vaincu ne contient plus d'unité, l'attaquant se déplace automatiquement sur cette case. Cela coûte alors le coût d'un déplacement (bonus et malus inclus). Le même calcul s'applique lorsque l'unité attaquante ne détruit pas l'unité adverse. Cela signifie qu'une unité qui vient d'attaquer peut très bien se déplacer ensuite si elle possède les points de déplacement nécessaires. Lorsqu'un joueur n'a plus d'unité, il est éliminé. Lorsqu'il ne reste plus qu'un seul joueur dans une partie, celui-ci a gagner. Une unité ne regagne pas ses points de vie d'un tour à un autre.

La Vue

Voici un exemple du plateau du vrai jeu ainsi qu'un morceau de carte hexagonale pour vous donner une idée :



La carte, ses ressources, les unités de tous peuples sont visibles par tous les joueurs. Le jeu doit permettre de voir la carte du dessus (vue plateau) contrairement à beaucoup de jeux fournissant une vue isométrique.

Début de Partie

Au début du jeu, chaque joueur choisi son peuple. Chaque peuple débute la partie avec toutes ses unités sur la même case de la carte choisi de manière à ce que les joueurs ne soient pas trop proche. L'ordre de jeu est choisie aléatoirement en début de partie. Les joueurs jouent chacun leur tour sur leur même ordinateur. Deux joueurs ne peuvent sélectionner le même peuple.

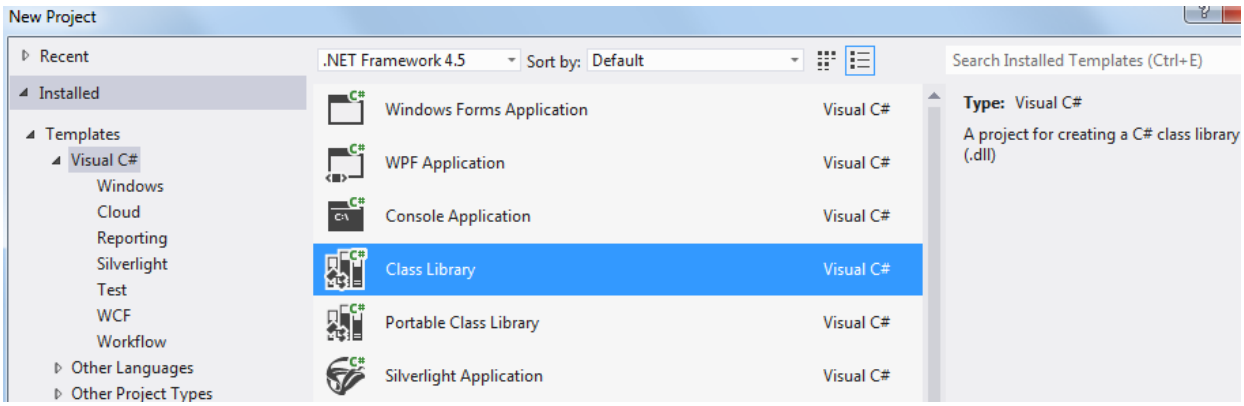
Tour de jeu

Lorsqu'un joueur peut jouer (c.-à-d. une fois par tour), il peut déplacer toutes ses unités suivant leur nombre de déplacements (un déplacement sur une case coûte un point de déplacement). Il est possible pour chaque unité de passer son tour (généralement par le biais de la touche *espace*). Une unité peut engager un combat s'il lui reste assez de points pour se déplacer sur la case visée (bonus inclus). Par exemple, si une unité A veut en attaquer une autre B se trouvant sur une case C. Si le coût de déplacement de A sur C coûte 0,5 grâce à un bonus, alors A peut attaquer s'il elle lui reste au moins 0,5 de déplacement.

Lorsqu'un joueur a fini son tour, il clique sur le bouton correspondant ("Fin tour"). C'est alors au joueur suivant de commencer son tour. La partie se termine lorsque le nombre de tours prédéfini en début de partie à été effectué, ou lorsqu'il ne reste qu'un seul joueur sur le plateau.

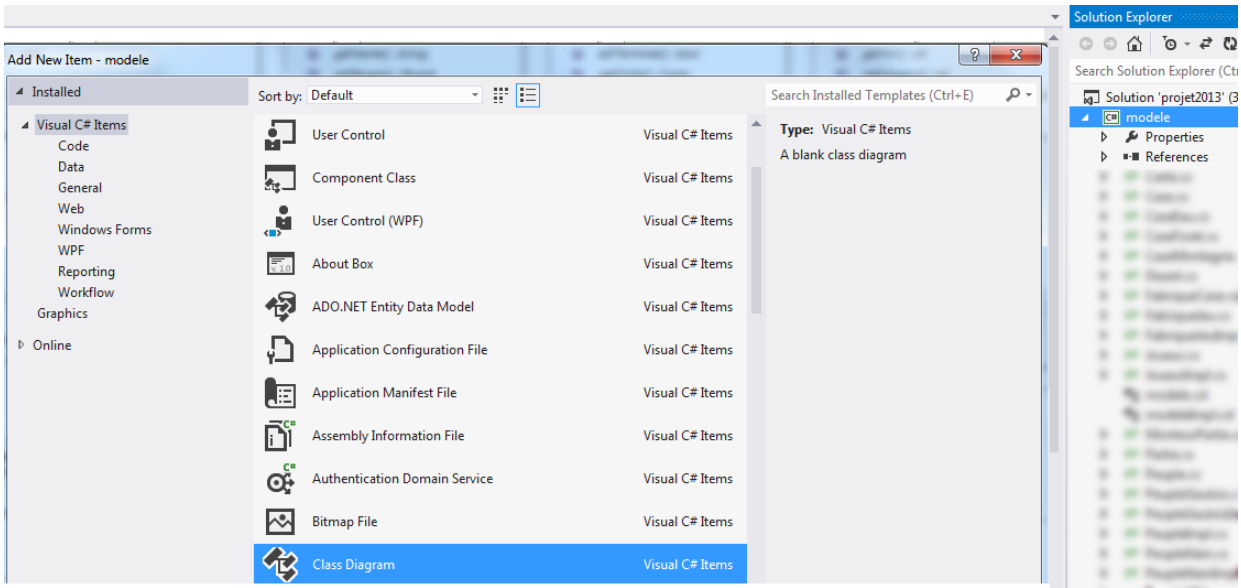
3 Modélisation (séances 1, 2 et 3)

Vous utiliserez Visual Studio 2013 Ultimate pour réaliser la modélisation de votre projet [3]. La méthode la plus facile pour créer les diagrammes de classes dans le contexte de ce projet est de créer un projet "Class Library" comme le montre la figure suivante :



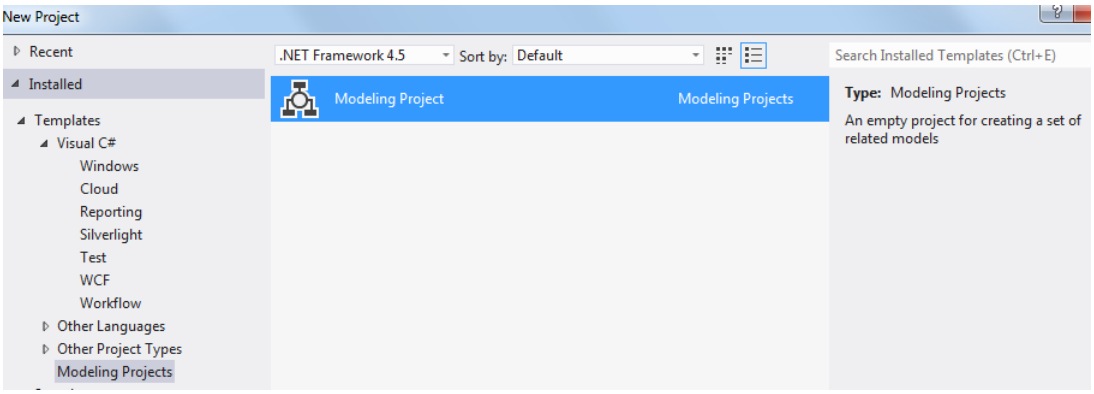
Dans ce nouveau projet vous pourrez alors créer un diagramme de classes :

3. Visual Studio 2013 Ultimate est disponible à l'INSA *via* DREAMSPARK (*cf.* `http://intranet.insa-rennes.fr/index.php?id=652`. Notez que seule la version Ultimate possède le module UML.

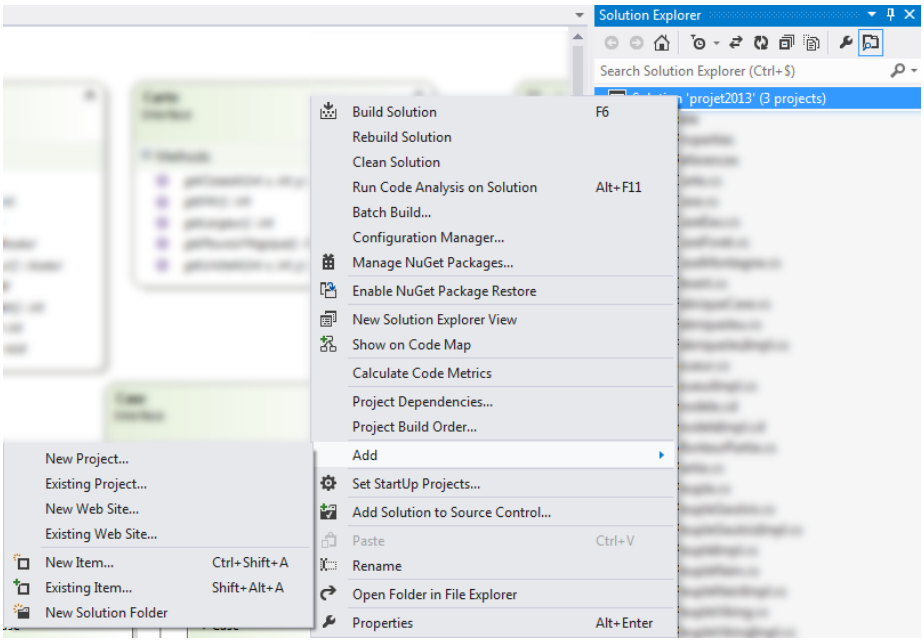


Vous n'utiliserez pas le diagramme de classes UML fourni dans la section UML de Visual Studio car la génération du code C# y est plus compliquée. En effet, avec le diagramme de classes de l'image précédente chaque modification dans le diagramme influe directement sur du code généré automatiquement. Il est également possible d'éditer le code C# ; le diagramme se mettra alors à jour automatiquement. Cependant, cet éditeur de diagrammes de classes ne permet par de définir des packages. Vous pouvez cependant créer dans dossiers et y déplacer les fichiers générés, voire définir des namespaces. L'éditeur de diagrammes de classes gère de tels changements.

Concernant les autres diagrammes UML demandés, vous devrez créer un projet de modélisation (dans Visual Studio, une *solution* consiste en un ensemble de projets dédiés à la conception, au développement et au test d'une application) :



Pour ajouter un projet à une "solution" – terme employé par Visual Studio pour parler d'un ensemble de projets liés à une application donnée – vous devrez faire comme suit :



3.1 Description de l'étape de modélisation

Les thèmes essentiels que vous devez aborder lors de la phase d'analyse et de conception sont les suivants :

- modélisation des données du jeu (joueur, case, carte, unité, *etc.*) à l'aide de différents diagrammes de classes ;
- modélisation du comportement du jeu à l'aide de diagrammes d'interaction, d'états-transitions (déroulement des combats, fonctionnement d'un tour, *etc.*) ;
- utilisation des patrons de conception suivants (cela est lié aux trois points précédents) :
  - *Fabrique*, pour gérer les différentes peuples.
  - *Monteur*, pour la création d'une partie. Il devra exister 2 monteurs. Le premier sera dédié au montage d'une nouvelle partie. Le second au montage d'une partie enregistrée que l'on souhaite charger.
  - *Poids-mouche* pour la modélisation de la carte.
  - *Stratégie*, pour la création des différents types de carte.

Il existe également de nombreux autres scénarios que vous pouvez développer (scénarios nominaux ou bien gestion des erreurs notamment).

3.2 Tâches à réaliser

Votre modélisation du jeu à l'aide de diagrammes UML sera fondée sur votre analyse. Votre rapport contiendra les diagrammes suivants :

- 2 illustrations des fonctionnalités du jeu à l'aide de cas d'utilisation.
- Les diagrammes de classe représentant :
  - la modélisation globale du jeu (carte, joueurs, jeu, peuples, *etc.*) ;
  - les patrons de conception utilisés.
- 1 diagramme d'états-transitions pour modéliser le fonctionnement des différents objets (par exemple le cycle de vie d'une des unités). Visual Studio ne permet pas de réaliser des diagrammes d'états-transitions. Modelio (disponible sous les machines Linux du département) permet de faire de tels diagrammes.

— 2 diagrammes d'interaction pour vous aider à définir les diagrammes de classes (création d'une partie, déroulement globale d'une partie jusqu'à la victoire d'un joueur, déroulement d'un tour pour 1 joueur, *etc.*).

Tout diagramme supplémentaire, correct et ayant une utilité sera considéré positivement lors de l'évaluation du projet.

3.3 Aide

— **Pas d'attributs dans les interfaces ni de relations autres que l'héritage entre les interfaces (même si Visual Studio le permet) !**
— Débuter la modélisation d'un projet est souvent fastidieux et déroutant ("Par où commencer ?", "Que dois-je modéliser ?", *etc.*), Il est généralement recommandé de commencer un diagramme, voire plusieurs en même temps, sur une feuille de papier.
— Les diagrammes de séquence aident à identifier les opérations des classes de vos diagrammes de classes. Essayez de faire en parallèle ces deux types de diagrammes.
— Vous pouvez utiliser les classes issues de la librairie .NET comme type d'un attribut, *etc.* Pour cela, il vous suffit d'écrire dans le champ type le nom complet de la classe. Par exemple, pour utiliser la classe *Color*, il faut mettre dans le champ type *System.Drawing.Color*. Le nom complet de chaque classe peut se trouver sur Internet.
— N'oublier pas de modéliser les constructeurs des classes ainsi que leurs paramètres.
— Comme présenté en cours dans les exemples WPF, C# gère propose un moyen pour implémenter le patron de conception *observateur* (il n'est apparemment pas possible d'importer une classe/interface C# dans un diagramme de classes, donc vous ne modéliserez donc pas dans les diagrammes de classes la notion d'observabilité mais la gérerez lors du codage, si vous voulez utiliser le patron *Observateur*). Pour cela il faut utiliser du côté du modèle l'interface *INotifyPropertyChanged*. L'implémentation de cette interface requière la déclaration d'un attribut d'un type spécial, un *event*. Les *events* permettent à une classe de notifier d'autres classes (*cf.* Vous pouvez étudier et vous inspirer du code suivant :

```
public event PropertyChangedEventHandler PropertyChanged;
protected void OnPropertyChanged(string name) {
  PropertyChanged(this, new PropertyChangedEventArgs(name));
}
```

Ainsi, lors d'une modification d'un attribut, il est possible de notifier la vue écoutant le modèle modifié :

```
protected Point pos;
public Point Position {
  set {
    this.pos = value;
    OnPropertyChanged("Position");
  }
}
```

Mais cela implique que la vue s'abonne au modèle, par exemple :

```
model.PropertyChanged += new PropertyChangedEventHandler(update);
override public void update(object sender, PropertyChangedEventArgs e){...}
```

Dans ce code tiré d'une vue, on abonne la vue au modèle *model* : la méthode *update* sera appelée à chaque appel de *OnPropertyChanged* dans le modèle.

Note : Les accesseurs en lecteur et écriture (getter et setter) fonctionnent différemment qu'en Java, il s'agit des "properties" (qui existent en Java 8 désormais). Lisez les documents suivants pour comprendre leur fonctionnement : `http://msdn.microsoft.com/en-us/library/w86s7x04.aspx` et `http://msdn.microsoft.com/en-us/library/64syzecx.aspx` pour utiliser les "properties" dans les interfaces.

4 Implémentation (à partir de la séance 4)

4.1 Description de l'étape

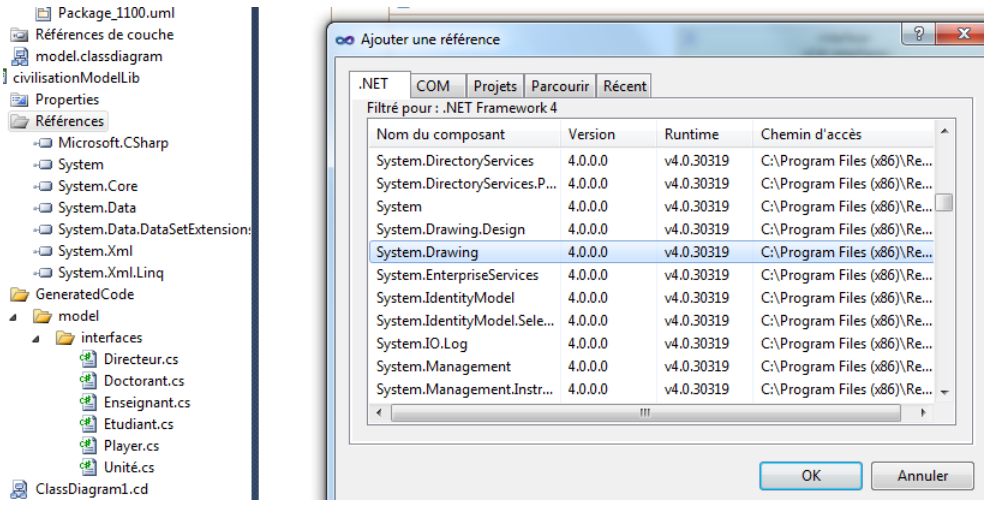
Votre application sera développée en C# et en C++ : C++ pour les algorithmes requérant des calculs (*cf.* la section concernée) ; C# pour le reste dont une partie est automatiquement générée grâce aux diagrammes de classes.
Vous devrez compléter le code généré (le code du diagramme concernant l'implémentation) automatiquement en n'oubliant pas de compiler régulièrement.

4.2 Tâches à réaliser

Lors de ces séances, vous devrez compléter le code généré, *i.e.* implémenter les opérations.

4.3 Aide

— N'oublier pas de compiler au fur et à mesure du codage pour corriger plus facilement les erreurs.
— Vous découvrirez très certainement des erreurs de modélisation lors du codage (c'est généralement le moment où l'on se dit que l'on aurait dû faire un diagramme de séquence pour éviter ce genre de problème). Il faudra donc modifier les diagrammes.
— Sous Visual Studio, utiliser un type issu d'une bibliothèque .NET nécessite l'ajout de celle-ci dans les références du projet généré. Pour cela, allez dans l'explorateur de solutions, cliquez-droit sur le dossier *Références* de votre projet, sélectionnez ajout d'une référence et allez dans l'onglet *.NET* comme le montre l'image suivante (*System.Drawing* pour utiliser la classe *Color*, *WindowsBase* pour utiliser la classe *Point*, *etc.*) :



— Pour exécuter l'application WPF vous devez définir le projet WPF comme le projet de la solution à exécuter : clic-droit sur le projet WPF, puis "Définir comme projet de démarrage".

5 Test Logiciel (à partir de la séance 4)

Il est important de tester les différentes parties d'un logiciel *au fur et à mesure du développement.* Dans le cas présent, vous devez tester les librairies C++ et votre code C#. Pour simplifier ce processus de test, vous testerez votre code C++ au travers de tests écrit en C#.
Effectuer des tests sur du code C#. Il faut ajouter un projet de test C# ("Unit Test Project") à la solution. Il faut ensuite aller dans le code C# que l'on veut tester et sélectionner une opération ->clic droit ->créer des tests unitaires. Puis choisir le projet de test C# créé. Note : cette fonctionnalité n'est plus disponible dans VS 2012. Voici un tutoriel Microsoft sur la définition de tests dans VS 2012 : `http://msdn.microsoft.com/en-us/library/ms182532%28v=vs.110%29.aspx`
Le but de chaque opération de test est de vérifier certaines propriétés de votre code en utilisant des assertions : `http://msdn.microsoft.com/fr-fr/library/ms182532(v=vs.80).aspx`
Par exemple voici un test, tiré d'un projet 2012-2013 concernant le jeu Civilisation, vérifiant que la prise d'une ville (les ville n'existent plus dans votre version du sujet) lors d'un combat (merci de ne pas utiliser de "_" dans les types et variables !) :

```
    [TestClass]
    public class TestCombat {
        [TestMethod]
        public void TestPriseVillage() {
            //Test de prise d'un village
            Monteur_Partie monteur = new Monteur_Partie_IMPL();

            Hashtable joueurs = new Hashtable();
            joueurs.Add("Flo", new Tuple<string, Color>("INFO", Colors.Blue));
            joueurs.Add("Nico", new Tuple<string, Color>("INFO", Colors.Red));

            monteur.genererPartie(joueurs, false);

            Joueur j1 = monteur._partie._joueurs[0];
            Joueur j2 = monteur._partie._joueurs[1];

            Case depart = monteur._partie._carte.getCase(10, 10);
            Case arrivee = monteur._partie._carte.getCase(10, 11);

            Fabrique_Unite_INFO.FABRIQUE_UNITE.creerEtudiant(depart, j1);
            Fabrique_Unite_INFO.FABRIQUE_UNITE.creerEnseignant(arrivee, j2);

            ((Enseignant)arrivee.Unites[0]).creerVillage();

            //Le village appartient au joueur 2
            Assert.AreEqual(j2, arrivee.Ville.Joueur);
            //Le joueur 1 attaque cette ville
            ((Etudiant)depart.Unites[0]).deplacer(arrivee);
            //Le village appartient au joueur 1
            Assert.AreEqual(j1, arrivee.Ville.Joueur);
        }
}
```

Vous devrez également implémenter des tests à partir des diagrammes de cas d'utilisation / séquence pour valider que l'implémentation du jeu correspond bien à vos spécifications.

Pour exécuter des tests, allez dans le fichier de tests, cliquez-droit dans l'éditeur de texte, puis "exécuter les tests".

5.1 Aide

— N'oubliez pas que si les diagrammes de séquences, et compagnies, ne sont pas utilisés pour générés du code, ils sont très utiles pour définir des cas de test. Par exemple, vérifier le bon fonctionnement du changement d'état d'une unité.
— Vous pouvez avoir des problèmes de compilation ou d'exécution des tests lorsque ces derniers se trouvent sur un disque réseau. Vous pouvez alors soit déplacer les fichiers en local, soit configurer *Visual Studio* (cf. `http://coursonline.insa-rennes.fr/mod/forum/discuss.php?d=399`).

6 Développement de la librairie C++ (séance 5)

Différents algorithmes devront être developés en C++. Ils seront utilisés sous la forme d'une librairie dans votre projet. Pour ce faire, vous aurez besoin de développer un *wrapper* faisant le lien entre le C# et le C++. Vous pouvez utiliser l'une ou l'autre des deux méthodes vues en cours.

6.1 Tâches à réaliser

Développer une librairie C++ réalisant les algorithmes suivants :

1. Création de la carte et placement des unités. La création d'une carte peu s'avérer extrêmement complexe. C'est pourquoi il vous est fixé un certain de contraintes visant à encadrer le fonctionnement d'un tel algorithme [4] :
   (a) Une carte doit contenir tous les types de terrain. Cependant, il n'y a pas de stratégie particulière pour regrouper les types de terrain.
   (b) Les joueurs doivent être placés le plus loin des uns des autres.
2. Suggestion des déplacements : où une unité peut se déplacer dans le tour ; pouvant intéresser à plus long terme une unité (bonus, bloquer un ennemi, *etc.*). Cet algorithme doit analyser la carte (terrains, peuples ennemies, *etc.*) afin de suggérer jusqu'à 3 emplacement (cases).

6.2 Aide

1. Visual Studio n'apprécie pas vraiment lorsque vous utilisez des fichiers (par exemple votre DLL) stockés sur un disque réseau. Si vous observez des problèmes de compilation ou d'exécution étranges, pensez à vérifier que le problème ne provienne pas de cela.
2. Si vous avez des problèmes de lien avec votre librairie C++ lors de la compilation en mode "*Release*" de votre projet, vérifier cette solution : allez dans les propriétés du projet wrapper (mode *release*) -> éditions de liens -> entrée -> dépendances supplémentaires -> ajoutez le chemin vers le *.lib* (entre guillemets).

Voici un tutoriel concernant la création et l'utilisation d'une DLL C++ dans du code C# en complément du cours.
Étape 1 : créer un projet C++ ("*projet vide*", Visual C++) Ce projet contiendra vos algorithmes. Dans les propriétés du projet, changez le type de configuration de "*Application*" à "*Bibliothèque statique (.lib)*". Exemple d'un *.h* (comme expliqué en cours) :

4. Ne perdez pas trop de temps dans cet algorithme qui peut vite devenir chronophage. Commencez par une version basique que vous pourrez améliorer si vous en avez le temps à la fin du projet.

```
#ifdef WANTDLLEXP
  #define DLL _declspec(dllexport)
  #define EXTERNC extern "C"
#else
  #define DLL
  #define EXTERNC
#endif

class DLL Algo {
public:
  Algo() {}
  ~Algo() {}
  int computeFoo();
};

// A ne pas implémenter dans le .h !
EXTERNC DLL Algo* Algo_new();
EXTERNC DLL void Algo_delete(Algo* algo);
EXTERNC DLL int Algo_computeAlgo(Algo* algo);
```

Exemple d'un *.cpp* :

```
#include "Algo.h"

int Algo::computeFoo() { return 1; }

Algo* Algo_new() { return new Algo(); }
void Algo_delete(Algo* algo) { delete algo; }
int Algo_computeAlgo(Algo* algo) { return algo->computeFoo(); }
```

la compilation du projet doit générer un fichier *.lib* dans le répertoire *Debug* ou *Release* de la solution (et non pas du projet).
Étape 2 : création d'un projet de wrapping ("*Projet Vide CLR*", Visual C++) Ce projet définira le wrapper en le code C++ et C#. Dans les propriétés du projet, changez le type de configuration de "*Application*" à "*Bibliothèque dynamique (.dll)*". Ajouter une référence vers le projet C++.
Exemple d'un *Wrapper.h* :

```
#ifndef __WRAPPER__
#define __WRAPPER__

#include "../libCPP/Algo.h" // A changer
// Le bon chemin pour être "../Debug/nomProjetLib.lib"
#pragma comment(lib, "libCPP.lib") // A changer

using namespace System;

namespace Wrapper {
public ref class WrapperAlgo {
private:
  Algo* algo;
```

```
public:
  WrapperAlgo(){ algo = Algo_new(); }
  ~WrapperAlgo(){ Algo_delete(algo); }
  int computeFoo() { return algo->computeFoo(); }
};
}
#endif
```

Il faut définir un *Wrapper.cpp* pour que le projet compile :

```
#include "Wrapper.h"
```

la compilation du projet doit générer un fichier *.dll* dans le répertoire *Debug* ou *Release* de la solution (et non pas du projet).

Étape 3 : Utilisation dans un projet WPF (ou dans un projet de test).

- Créez un projet WPF (ou de test).
- Ajoutez une référence vers le projet wrapper.
- Définir le projet WPF comme le projet de la solution à exécuter : clic-droit sur le projet WPF, puis "*Définir comme projet de démarrage*".

Il est maintenant possible d'utiliser les classes du wrapper (ex : WrapperAlgo algo = new WrapperAlgo() ;) dans votre code C# (pensez à importer le package du wrapper : `using Wrapper;`

Vos opérations C++ peuvent retourner un pointeur. Exemple :

```
EXTERNC DLL int* Algo_computeAlgo(Algo* algo);
```

Dans ce cas, l'opération de wrapping correspondante retourne également un pointeur. Exemple dans *Wrapper.h* :

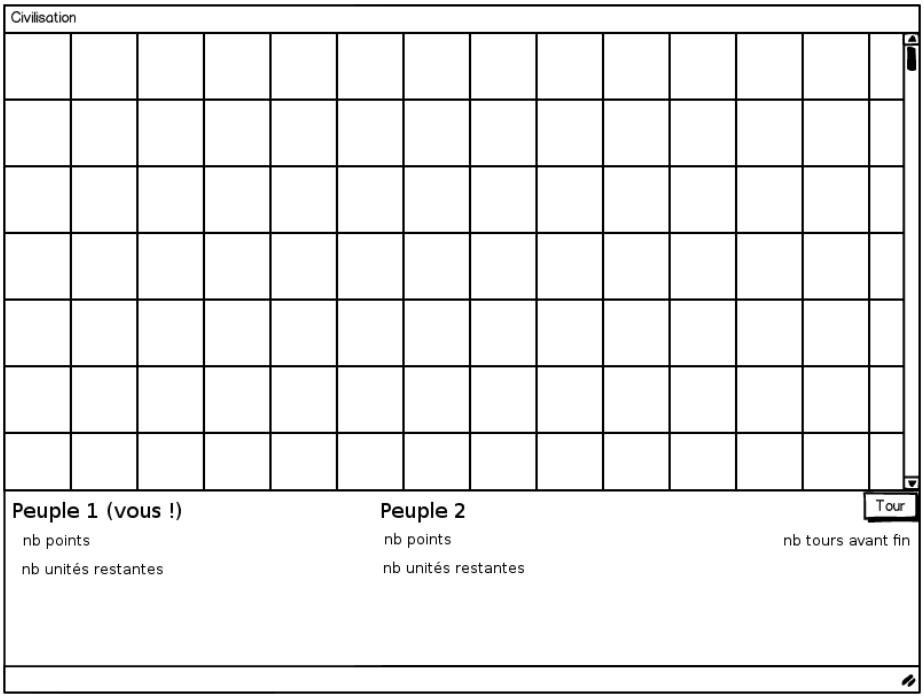
```
int* computeFoo() { return algo->computeFoo(); }
```

Concernant le projet WPF, il faut autoriser la compilation de code "*unsafe*" : *propriété -> Générer -> Autoriser du code unsafe* et rajouter le mot-clé *unsafe* aux opérations C# utilisant des pointeurs. Exemple :

```
unsafe public MainWindow() {
WrapperAlgo algo = new WrapperAlgo();
int* tab = algo.computeFoo();
for (int i = 0; i < 5; i++)
System.Console.WriteLine(tab[i]);
```

7 Aspects interactif et graphique (séances 6, 7 et 8)

L'interface graphique possède une vue affichant une partie de la carte, un panneau contenant les informations sur la partie ou l'unité sélectionnée et un bouton de fin de tour. Le mockup suivant vous donne une idée d'une possible IU :



7.1 Tâches à réaliser

Essayez de progresser de manière itérative :

- Créer un projet "WPF application" qui utilisera les librairies C++ et C#.
- Affichage de la carte. La carte étant trop grande pour être vue dans sa totalité, des moyens (par ex. des ascenseurs : *ScrollViewer*) doivent être mis en place pour déplacer la vue. Pour afficher les terrains, vous pouvez soit les faire dessiner de manière basique (p. ex. *DrawRectangle*) dans des méthodes *OnRender(DrawingContext dc)*, soit utiliser les textures que l'on vous fournies et les afficher avec *DrawImage*.
  Note : La surcharge de *OnRender()* est considérée comme une mauvaise pratique XAML/WPF. Vous l'utiliserez ici uniquement dans un but pédagogique pour mettre en œuvre le patron poids-mouche. Dans la vraie vie, deux solutions seraient possibles : utiliser un *<ItemsControl>* fondé sur un *canvas* : WPF gère seul le poids-mouche en mettant les images en ressource ; créer un bitmap une fois pour toute pour le fond de carte et utiliser un contrôle *<Image>*, puis se fonder sur un système de couches superposées pour les unités, la sélection, *etc.*
- Affichage des unités.
- La sélection d'une unité s'effectue en cliquant dessus.
- Création d'un panneau pour voir les caractéristiques d'une unité sélectionnée et des informations sur la partie en cours.
- Une unité peut être déplacée ou attaquer à l'aide du clavier (barre espace pour passer son tour) ou éventuellement de la souris. Nous vous conseillons d'utiliser le pavé numérique pour bouger les unités (*Key.NumPadX*).
- La fin d'un tour peut s'effectuer en cliquant sur un bouton *fin de tour* ou en appuyant sur la touche *entrée*.
- Sauvegarder et charger une partie. Cela vous sera fort utile lors de votre démonstration.

7.2 Aide

— Si vous utiliser les textures fournies, faites attention à utiliser un poids-mouche pour ne les charger qu'une seule fois.

8 Fonctionnalités avancées (facultatif et bonus)

Pour ceux disposant de temps, il est possible de développer des fonctionnalités plus avancées, qui seront considérées comme bonus lors de l'évaluation, dont voici une liste non-exhaustive :

— Ajout de nouveaux peuples ;
— Ajout de nouvelles cases ;
— Ajout de bonus d'attaque et de défense en fonction du type de terrain ;
— Jouer en réseau ;
— Jouer contre l'ordinateur.

9 Consignes générales pour le projet

9.1 Fonctionnalités de base attendues

Les fonctionnalités de base attendues sont toutes décrites dans les sections "Tâches à réaliser" de cet énoncé.

9.2 Rapport de conception

Vous devrez rendre votre rapport de conception **au plus tard le 12 novembre 2014 avant 23h59.999 sur la plate-forme moodle (aucun rapport envoyé par mail ne sera pris en compte pour la notation**). Étant donnée la taille de certains diagrammes, il est fortement conseillé de rendre une archive contenant le document PDF du rapport ainsi que les images lisibles des diagrammes (au format vectoriel, PDF ou SVG, de préférence).

9.3 Soutenance de projet

Vous devrez aussi préparer une soutenance de 8 min. Les soutenances auront lieu le 17 janvier 2015. Cette soutenance devra se diviser entre : une démonstration de votre jeu (prévoyez plusieurs cas d'utilisation) ; une présentation. Il est aussi impératif qu'au plus tard le 16 janvier 2015 à 23h59.999, vous ayez déposé sur la plate-forme Moodle les documents suivants :

1. la documentation utilisateur de votre jeu ;
2. le code source (en entier) ;
3. un exécutable qui fonctionne.

Chaque jour de retard entraînera un point de pénalité sur la note finale.

9.4 Notation

Le barème suivant vous est donné à titre indicatif :

1. Rapport de conception $\approx$ 8 pts ;
2. Documentation utilisateur (un manuel utilisateur du jeu pouvant inclure un tutoriel, les règles, le fonctionnement du jeu), code (commentaires, propreté, *etc.*) et fichiers exécutables $\approx$ 6 pts ;
3. Présentation et démonstration $\approx$ 6 pts.