INSA RENNES

Projet Analyse, Conception et P.O.O.
Quatrième année - Informatique

Éric Anquetil, Département Informatique
eric.anquetil@insa-rennes.fr
Arnaud Blouin, Département Informatique
arnaud.blouin@insa-rennes.fr
Manuel Bouillon, Département Informatique
manuel.bouillon@irisa.fr
Grégoire Richard, Département Informatique
gregoire.richard@irisa.fr
Maud Marchal, Département Informatique
maud.marchal@insa-rennes.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 qu'il doit repartir sur la carte. Le placement de chaque unité rapporte plus ou moins de points. 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 comptés à la fin de la partie, c.-à-d. après un nombre prédéfini de tours. 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], Gaulois, Viking et Nain, ayant des caractéristiques très différentes influant sur les stratégies de jeu :

1. Gaulois.
   - Le coût de déplacement sur une case *Plaine* est divisé par deux.
   - Une unité Gauloise fournit 1 point de plus lorsqu'elle occupe une case du type plaine.
   - Une unité Gauloise n'acquière aucun point sur les cases de type montagne.
2. Vikings.
   - L'unité Viking a la capacité de se déplacer sur l'eau. L'occupation d'une case eau ne rapporte cependant aucun point.
   - Une unité Viking fournit 1 point de plus lorsqu'elle occupe une case au bord de l'eau.
   - Une unité Viking n'acquière aucun point sur les cases de type désert.
3. Nains.

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

- 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.
- Une unité Nain fournit 1 point de plus lorsqu'elle occupe une case forêt.
- Une unité Nain n'acquière aucun point sur les cases de type plaine.

À chaque tour, toutes unités peuvent être déplacées ou attaquer. Par défaut (c.-à-d. hors bonus), chaque unité peut se déplacer d'une case par tour. Chaque unité possède 2 d'attaque, 1 de défense et 2 points de vie.

La Carte du Monde

La carte du monde se compose de cases carrées. La largeur d'une case est de 50 pixels. Il existe différents types de case : montagne, plaine, désert, eau, forêt.
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, 5 cases × 5 cases, 5 tours, 4 unités par peuples.
- Petite : 2 joueurs, 10 cases × 10 cases, 20 tours, 6 unités par peuples.
- Normale : 2 joueurs, 15 cases × 15 cases, 30 tours, 8 unités par peuples.

Les Combats

Pour qu'une unité puisse lancer une attaque contre une unité d'un autre peuple, elles doivent se situées sur des cases juxtaposées (attaque en diagonale impossible cependant). Lorsqu'une unité attaque une case contenant plusieurs unités, la meilleure unité défensive est choisie. Une unité attaquée possédant 0 en défense meurt immédiatement. Sinon, un certain nombre de combats a lieu. 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% 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 75% de sa vie, alors son attaque sera au final de 4*75% = 3. 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, ce dernier se déplace automatiquement sur cette case. 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

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

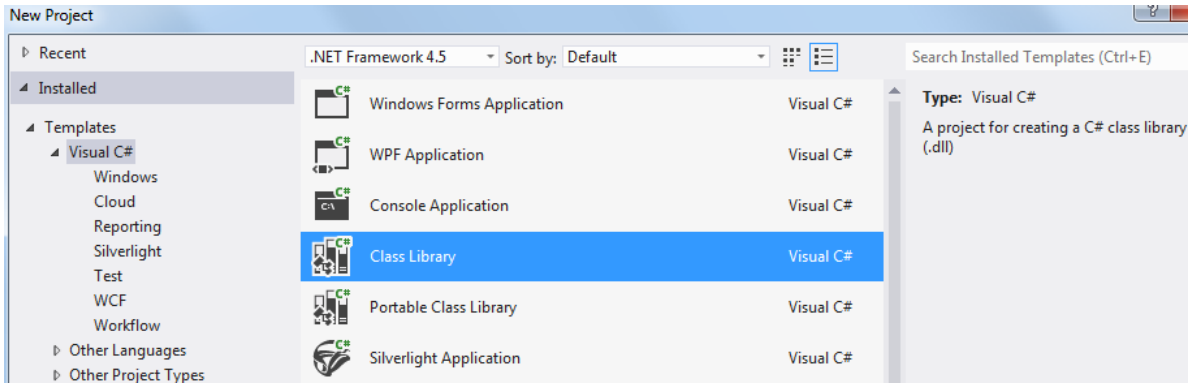
Au début du jeu, chaque joueur choisi son peuple. Chaque peuple débute la partie avec toutes ses unités sur la même case de la carte choisi de manière à ce que les joueurs ne soient pas trop proche. L'ordre de jeu est choisie aléatoirement en début de partie. Les joueurs jouent chacun leur tour sur leur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

Tour de jeu

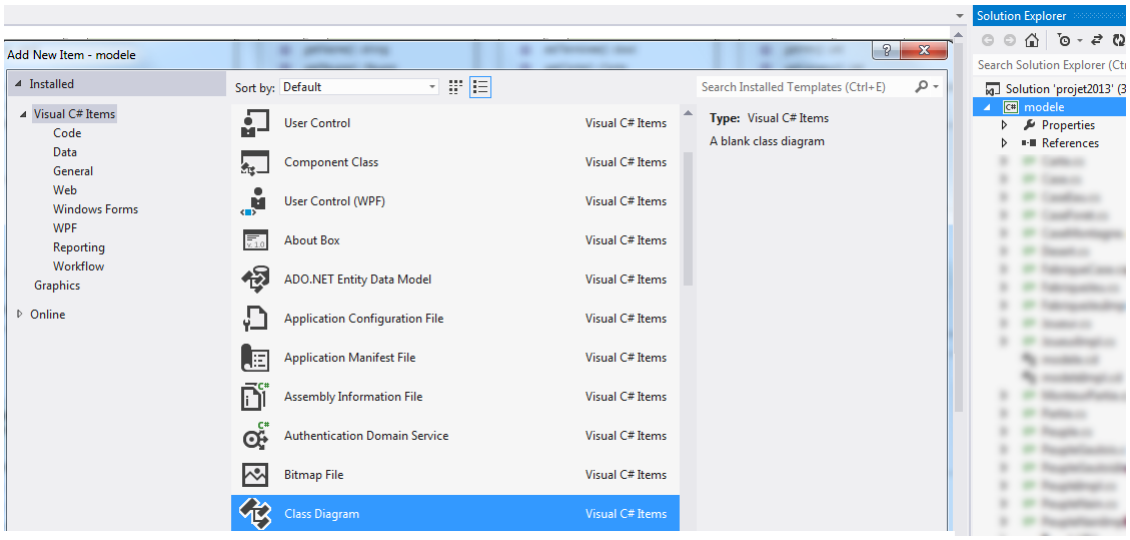
Lorsqu'un joueur peut jouer (c.-à-d. une fois par tour), il peut déplacer toutes ses unités suivant leur nombre de déplacements (un déplacement sur une case coûte un point de déplacement). Il est possible pour chaque unité de passer son tour (généralement par le biais de la touche *espace*). Une unité combattante peut engager un combat s'il lui reste au moins un point de mouvement. 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.
Voici un exemple du plateau du vrai jeu pour vous donner une idée :

3 Modélisation (séances 1, 2 et 3)

Vous utiliserez Visual Studio 2012 Ultimate pour réaliser la modélisation de votre projet [2]. 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 :



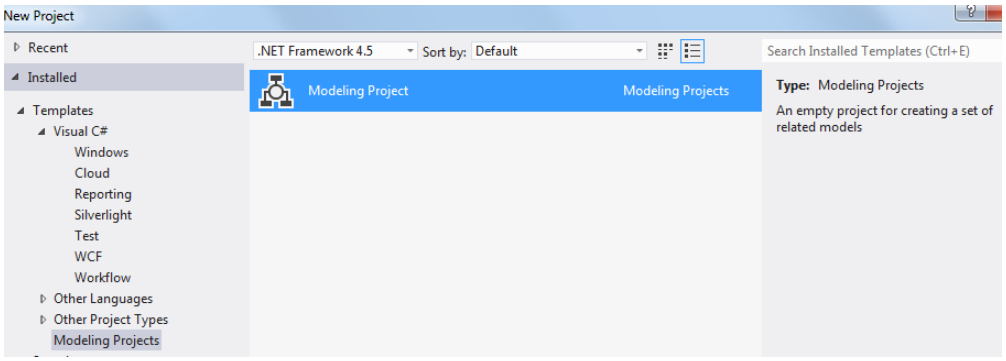
Dans ce nouveau projet vous pourrez alors créer un diagramme de classes :



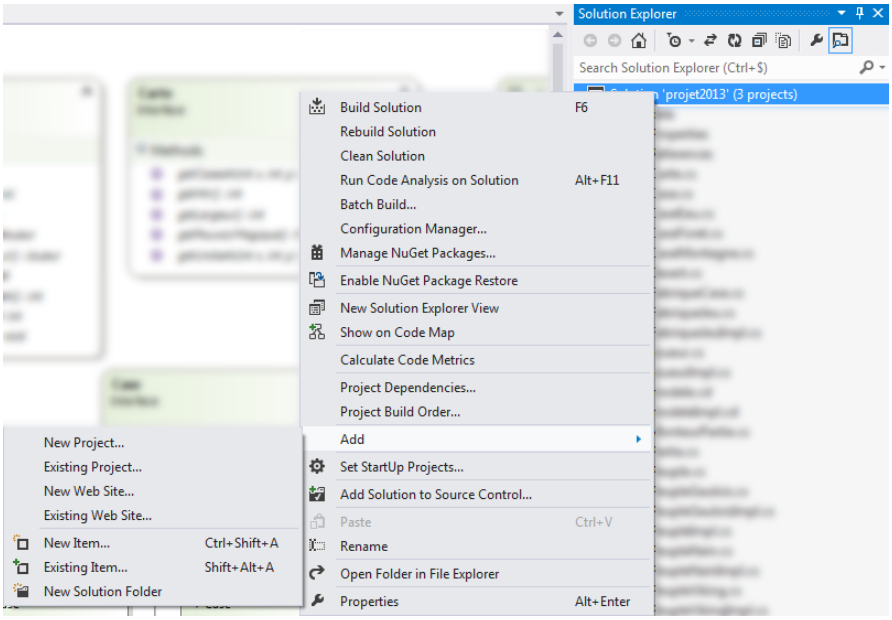
Vous n'utiliserez pas le diagramme de classes UML fourni dans la section UML de Visual Studio car la génération du code C# y est plus compliquée. En effet, avec le diagramme de classes de l'image précédente chaque modification dans le diagramme influe directement sur du code généré automatiquement. Il 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. Le plus simple est de définir d'abord les interfaces dans un diagramme, puis leur implémentation dans le même diagramme, pour enfin couper-coller les classes de l'implémentation dans un nouveau diagramme.

Concernant les autres diagrammes UML demandés, vous devrez créer un projet de modélisation :

2. Visual Studio 2012 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.



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, vue, *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, choix de la case de ville lors de l'agrandissement de la population, fonctionnement d'un tour, *etc.*) ;
- utilisation des patrons de conception suivants (cela est lié aux trois points précédents) :
  - *Fabrique abstraite*, pour gérer les différentes peuples.
  - *Monteur*, pour la création d'une partie.
  - *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 pourra comporter les parties suivantes (à titre indicatif) :

- 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. Vous pouvez donc les faire avec un quelconque éditeur de dessins.
- 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). 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.* http ://msdn.microsoft.com/en-us/library/aa645739.aspx). 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". 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

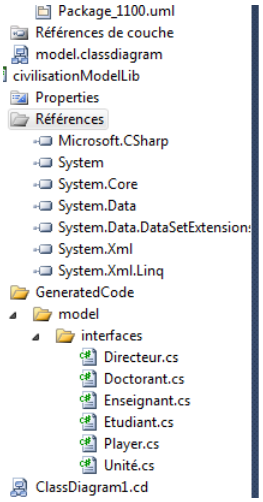
Votre application sera développée en C# et en C++ : C++ pour les algorithmes requérant des calculs (par ex. gestion des combats, création de la carte) ; C# pour le reste dont une partie est automatiquement générée grâce aux diagrammes de classes.
Vous devrez compléter le code généré (le code du diagramme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

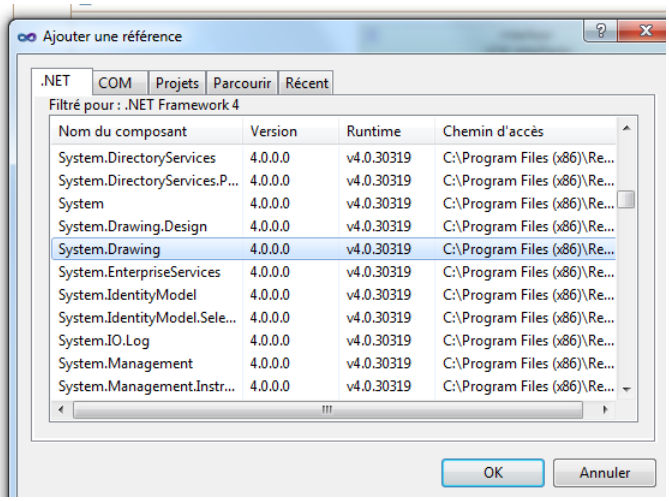
4.2 Tâches à réaliser

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 généré, 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.*) :





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.

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%28v=vs.80%29.aspx

Par exemple voici un test, tiré d'un projet 2012-2013 concernant le jeu Civilisation, vérifiant que la prise d'une ville 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.

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 joueurs. 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 [3] :
   (a) Une carte doit contenir tous les types de terrain et de pouvoirs magiques. Cependant, il n'y a pas de stratégie particulière pour regrouper les types de terrain.
   (b) Une carte de dimension $n \times n$ (une carte est forcement carrée), le nombre de ressources doit être de $n/2$.
   (c) Les joueurs doivent être placés le plus loin des uns des autres.
2. Suggestion des cases : 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 (pourvoir magiques, terrains, peuples ennemies, *etc.*) afin de suggérer jusqu'à 3 emplacement (cases).

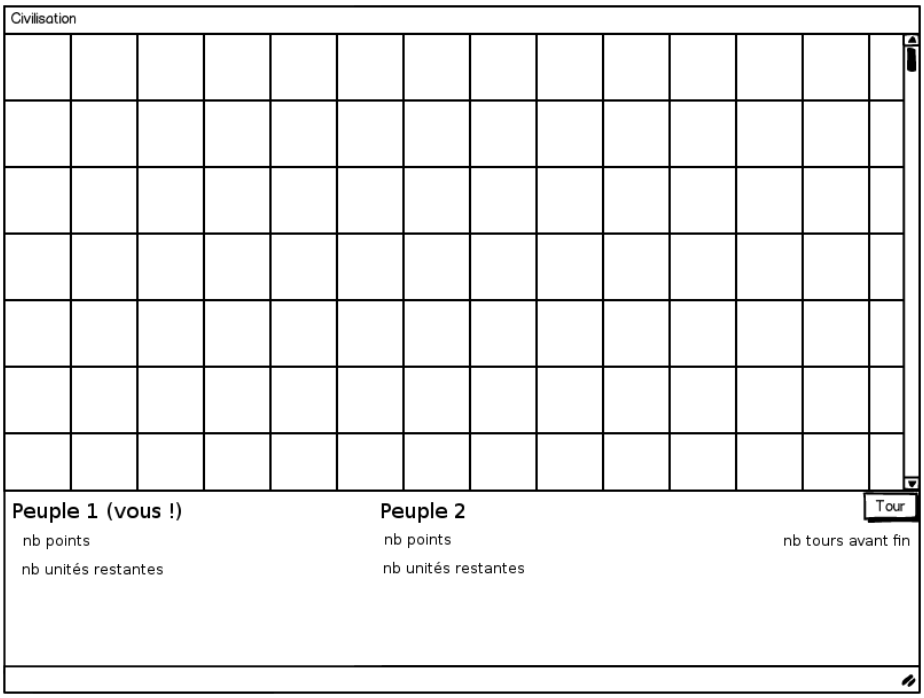
6.2 Aide

1. Si vous avez des problèmes de lien avec votre librairie C++ lors de la compilation en mode "Release" de votre problème, 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).

3. 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.

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 et de ces ressources. 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 et les ressource, 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 et pouvoirs magiques ;
– 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 19 novembre 2013 avant minuit sur la plate-forme moodle (aucun rapport envoyé par mail ne sera pris en compte pour la notation**).

9.3 Soutenance de projet

Vous devrez aussi préparer une soutenance de 10 min. Les soutenances auront lieu le jeudi 16 janvier 2014. 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 mercredi 15 janvier 2014 à minuit, 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, code (commentaires, propreté, *etc.*) et fichiers exécutables $\approx$ 6 pts ;
3. Présentation et démonstration $\approx$ 6 pts.