Chapitre 1

La SAÉ S2.01 en 2024

1.1 Éléments clés

— Travail en binôme
— Une séance de développement d'applications avec IHM – S2.01 Formation R2.02
— Une séance de développement objets – S2.01 Formation R2.01
— Quatre séances de travail en autonomie – S2.01 Dév. d'application
— Soit douze heures de travail prévues dans l'emploi du temps
— Trois livrables : code à la fin du tutoriel, rapport, code final
— Date limite : le 31 mars au soir

1.2 Évaluation

— Deux notes liées à votre travail sur la SAÉ : rapport évalué par l'équipe pédagogique de R2.01, coefficient 3 (sur 8) et rendu final de l'application évalué par l'équipe pédagogique de R2.02, coefficient 3 (sur 8).
— Deux notes liées aux compétences nécessaires pour la SAÉ : le DS1 de R2.01 (coefficient 1 sur 8) et le DS 1 de R2.02 (coefficient 1 sur 8).
— Ces quatre notes contribuent à hauteur de 33 % à la compétence 1 pour le S2 (UE 2.1).

Chapitre 2

Sujet

2.1 Concept

University deck build duel est un jeu de cartes (fictif) pour deux joueurs, dont l'objectif est de construire la plus belle université. Le travail est à effectuer sur la version numérique du jeu.

Le principe du jeu est que chaque joueur tire des cartes, dont certaines représentent des bâtiments ; à son tour, quand les conditions le permettent, le joueur construit certains de ses bâtiments en les posant devant lui. Certains bâtiments rapportent des points de victoire, et le joueur disposant du plus de points de victoire en fin de partie a gagné. À ce concept simple s'ajoutent des ressources, utilisées pour construire des bâtiments, des effets induits par les bâtiments, des actions...

Phases de jeu

Une fois terminé, le jeu se déroule comme suit :

1. Le tour n commence pour le joueur i,
2. Phase de pioche : le joueur i complète sa main entièrement en tirant le nombre de cartes qu'il lui manque,
3. Phase de jeu : le joueur i joue autant de cartes qu'il souhaite et s'il le peut,
4. Phase de défausse : le joueur i peut, s'il le souhaite, défausser les cartes de son choix,
5. Fin du tour n pour le joueur i, on passe à l'autre joueur ; si i était le deuxième joueur, on passe au tour $n+1$ si la partie n'était pas terminée.

2.2 Résumé du travail à effectuer

Il ne s'agit pas de partir de zéro, puisque les **spécifications du jeu** sont fournies dans ce document, ainsi qu'une **base de code** permettant de gérer quelques éléments dont l'**enchaînement des phases de jeu** ainsi qu'une première **interface graphique**. Le travail à réaliser sera donc multiple :

1. **Analyse** du code fourni, de la structuration des classes et d'un algorithme (méthode) en particulier,
2. **Conception** d'algorithmes supplémentaires et restructuration de code,
3. **Ajout** de fonctionnalités,
4. **Refonte et création** d'éléments d'interface graphique.

On demande de remettre un **dossier d'analyse et de conception**, une version du logiciel une fois fourni le **tutoriel**, et une dernière version du logiciel comprenant **tous** les éléments insérés.

Le tutoriel est accompagné de questions et sera évalué sur quatre points (sur 20) dans la partie réalisation.

La SAÉ, outre le tutoriel, présente dans une première partie le **travail obligatoire** à réaliser pour cibler une note maximale de 13 sur 16 (en cas de réalisation parfaite d'une excellence sans faille), et des **compléments** à aborder une fois le travail obligatoire réalisé ; suivant le temps qu'il vous reste, abordez un, plusieurs ou tous les compléments afin d'améliorer votre résultat – attention, la qualité sera privilégiée devant la quantité.

Le travail sera à effectuer en équipe de deux, éventuellement un ou trois avec accord explicite de l'enseignant. En dehors de la collaboration normale au sein de l'équipe, **tout partage de code ou de travail est interdit**, sera considéré comme du plagiat et sanctionné en tant que tel. En cas de difficultés, notamment sur la partie tutoriel, n'hésitez pas à solliciter votre enseignant.

2.3 Tutoriel

Le tutoriel est une partie d'éléments guidés vous permettant de prendre cette application en main. N'hésitez pas à solliciter votre enseignant durant la séance encadrée de R2.02. Répondez à chaque question dans un fichier texte, à rendre avec le tutoriel terminé dans la remise prévue à cet effet.

2.4 Découverte de l'application

Mode d'emploi

Pour visualiser l'application, ouvrez simplement le projet sous Visual Studio et exécutez-le. Une fenêtre de jeu s'affiche, avec les deux joueurs qui occupent respectivement les parties haute et basse de l'écran. Trois boutons sont actifs : passer son tour, jouer toutes les cartes possibles, et quitter. Une zone de messages donne l'avancée de la partie. Cliquez quelques fois sur les deux premiers boutons pour jouer et sur le troisième pour quitter. Les scores et le numéro du tour sont mis à jour.

Capture d'écran commentée

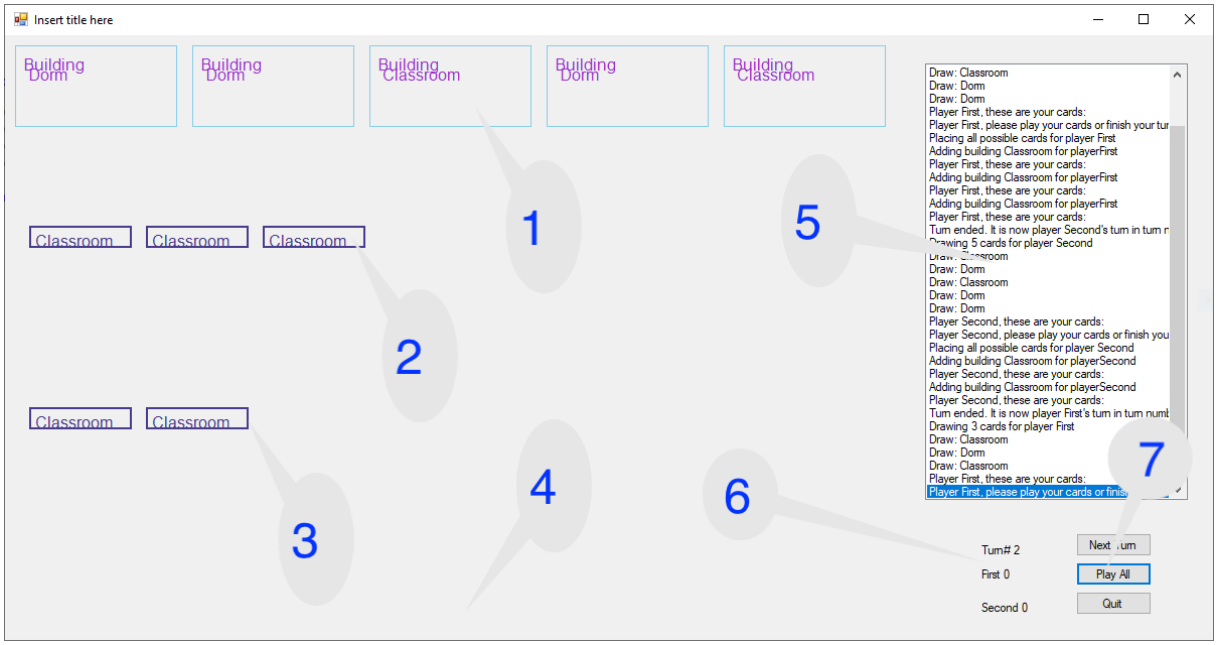

FIGURE 2.1 – Capture d'écran en cours de partie

Légende :

1. Affichage des cartes (`CardView`) dans la main du premier joueur (le joueur actif).
2. Affichage des bâtiments (`BuildingView`) posés par le premier joueur.
3. Affichage des bâtiments posés par le deuxième joueur.
4. Zone où les cartes du deuxième joueur seront affichés (vide car il n'est pas actif).
5. Affichage des messages émis par l'application (`ListBox`).
6. Trois `Label` affichant le numéro du tour en cours (à partir de 1), le nom et score du premier joueur, et le nom et score du deuxième joueur.
7. Trois `Button` : passer son tour (pour le joueur actif), tenter de jouer toutes les cartes (pour le joueur actif), et quitter l'application.

Explications

Le code fourni est fonctionnel en termes d'ordre de jeu, de passage d'une phase à l'autre et d'un joueur à l'autre, de gestion des cartes, des bâtiments ; la plus grande partie du *moteur* de jeu est réalisée. Une interface minimaliste, mais peu agréable, est fournie et fonctionnelle : à son tour, les cinq cartes du joueur actif s'affichent (en haut ou en bas suivant le joueur), et les bâtiments sont placés et s'affichent au milieu. Pour l'instant, le jeu contient deux types de cartes, des salles de cours et des résidences universitaires. Pour poser une résidence universitaire, il faut avoir déjà posé deux salles de cours ; chaque résidence universitaire rapporte un point.

Tous ces éléments fonctionnent bien avec le bouton permettant de placer toutes les cartes, le score est correctement mis à jour. Par contre, la partie ne s'arrête jamais (un message *Game over* est bien affiché, mais sans effet) et, assez rapidement, les affichages débordent et ne sont plus lisibles.

Le bouton *jouer toutes les cartes possibles* essaie en fait de jouer chacune des cartes de droite à gauche et passe à la suivante s'il échoue, ce qui ne permet pas de jouer intelligemment : les bâtiments nécessaires à d'autres ne sont pas forcément joués en premier, par exemple. Un des premiers objectifs de la réalisation sera de s'en passer.

Tutoriel 1 : essayez quelques parties pour vérifier le fonctionnement (limité) du code fourni.

2.5 Découverte du code

Vos notes pour la réalisation du rapport

Toute cette partie (découverte de code) est à utiliser pour votre dossier de conception ; vous pouvez bien sûr avancer dans votre travail en parallèle de ce dossier, mais il est temps de prendre des notes, au moins, avant de changer le code. Vous n'avez pas besoin de comprendre tout le détail du code existant tant que vous pouvez l'utiliser et ajouter vos modifications ; des compléments de C# sont disponibles en annexe section 4.2 si besoin.

Lien entre cartes, types de cartes, catégories de cartes

Tutoriel 2 : Examinez les trois classes suivantes : `Card`, `CardType`, `Kind`.

Dans un jeu de cartes, il peut y avoir plusieurs cartes du même type (trois cartes *salle de cours – Classroom*, par exemple). Ici, `Card` représente **une** carte, dont le type est `CardType` (et donc *Classroom* ici); le type d'une carte définit son nom, sa descrip-

tion, ses effets, etc. Il y a également plusieurs **catégories** de cartes : des bâtiments, des ressources et des actions. Chaque type de carte est d'une catégorie (`Kind`).

Question 3 : d'après le code de la classe `CardType`, comment différencie-t-on deux types de cartes ?

Les classes de logique métier

Tutoriel 4 : Examinez maintenant les classes `Effect`, `GameStatus`, `Player`, et enfin `Game`.

L'ensemble de ses classes représente la **logique métier** du jeu, contenant les règles et les données concernant une partie. On ne s'occupe ni de l'affichage, ni de la gestion de la fin d'une partie ou du score final, par exemple. Ces éléments sont dans le paquet (espace de noms) `Logic` ; la plus grande partie se trouve dans la classe `Game`.

Question 5 : Dans quelle méthode une nouvelle partie est-elle lancée ? Dans quelle méthode gère-t-on les phases du tour décrites en section 2.1?

La création des données

Dans le paquet `Logic.GameData` se trouve une unique classe sans instances, `LoadData`. Les méthodes, attributs et propriétés de cette classe sont responsables de la création des données (types de cartes et cartes).

Question 6 : de quelle manière sont stockés les types de cartes ?

Question 7 : combien y a-t-il d'exemplaires de cartes *salles de cours* dans le paquet de cartes distribué initialement ?

Programme et gestion de la partie

La classe `Program` contient la méthode principale. De manière centrale, la classe `Controller` est utilisée pour faire le lien entre la logique et l'interface graphique humain-machine ; dans l'application fournie, pour donner un retour à l'utilisateur, la logique de jeu fait appel à cette classe, et pour effectuer une action, la fenêtre fait appel à cette classe. De cette manière, seule cette classe, et non la logique de jeu, sera à changer quand l'interface sera modifiée pour ajouter des messages ou retirer des éléments d'interfaces obsolètes, par exemple. Cette classe sert aussi de meneur de jeu et d'arbitre en déterminant si la partie est terminée, ou qui a gagné, par exemple.

Question 8 : dans la classe `Controller` quelles sont les méthodes qui font appel aux classes de l'interface graphique ? Quelles sont les méthodes qui font appel aux classes de la logique de jeu ?

Les classes de l'interface graphique

Tutoriel 9 : dans le paquet `View`, examinez les classes `BuildingView`, `CardView`, et le formulaire `Window` partie design et code.

Question 10 : Comment sont affichées les cartes ? Les bâtiments ? Il y a une différence claire dans leurs algorithmes d'affichage, au moins initialement, laquelle ? (Indice : voyez la première ligne de l'affichage des cartes...)

2.6 Premiers ajouts

Dans cette partie, le code sera modifié par vos soins en suivant les instructions ; n'hésitez pas à interroger votre enseignant pendant le tutoriel.

Restaurant universitaire

Tutoriel 11 . Trois cartes sont pour l'instant implémentées, dont deux qui fonctionnent effectivement, plus le restaurant universitaire (*cafeteria*), dont l'ajout est donné en commentaire. Trouvez où, puis retirez les commentaires et essayez.

Dialogue de début de jeu

Un formulaire `StartupDialog` est presque prêt à être utilisé ; son but est de permettre de choisir le nombre de tours d'une partie (cinq, c'est très peu).

Tutoriel 12 : trouvez où `StartupDialog` est prêt à être appelé en commentaires, et retirez ces commentaires pour appeler ce formulaire. Rectifiez le comportement du bouton de confirmation.

Tutoriel 13 : on souhaite aussi pouvoir personnaliser les noms des deux joueurs ; ajoutez les éléments nécessaires dans ce formulaire, puis modifiez le reste du code en conséquence. Vérifiez le comportement de votre application avant de continuer.

Fin de partie

Tutoriel 14 . Pour l'instant, le code se contente d'afficher un message de `game over` en fin de partie, mais la partie continue (déraisonnablement). En fin de partie, les boutons permettant de jouer devraient être désactivés ; on devrait aussi pouvoir rejouer, avec un nouveau bouton (indisponible avant la fin de la partie). Ceci peut amener une restructuration du code ; le dialogue de début de jeu devrait apparaître (et être fonctionnel) lors d'une nouvelle partie.

Tutoriel 15 . En fin de partie, affichez aussi le vainqueur dans les messages de sortie.

2.7 Le dossier d'analyse et de conception en développement objets

On demande, pour un des rendus, un **dossier d'analyse et de conception** au format PDF. Les figures et diagrammes peuvent être réalisés avec tout outil adapté – UMLet est recommandé, les lacunes des logiciels utilisés ne sont pas des excuses valables pour ne pas respecter les normes. La police de caractères doit être lisible (11pt minimum), les pages numérotées, les figures et diagrammes légendés, chacun sur une page au maximum. Le nom des auteurs doit être précisé en page de garde. Il doit comporter chacun des éléments numérotés ci-dessous avec la mention **Dossier**.

2.8 Analyse de structure

Dossier 16 . Donnez le diagramme de classes par paquets (un diagramme par paquet) du code fourni. Attention, ici, les paquets sont représentés par des dossiers et des espaces de noms différents. Vous pouvez omettre les éléments privés.

Les propriétés doivent être représentées par leurs méthodes `get` et `set`, lorsqu'elles ne sont pas privées :

— `public int Number { get; set; }` devient ainsi `+Number.get(): int` et `+Number.set(int): void`,
— `public int Count { get { return ... ;} }` devient simplement `+Count.get(): int`.

2.9 Analyse d'algorithme : validation de condition de construction

Dans la logique de jeu, une vérification (déjà codée) est effectuée lors de chaque construction de nouveau bâtiment : il est vérifié que chacun des bâtiments requis est présent.

Dossier 17 . Analysez cet algorithme en détaillant, en français et avec schémas si nécessaire :

— comment sont organisées les structures de données, collections, tableaux associatifs mis en jeu,
— comment la condition souhaitée est-elle spécifiée et par quels objets,
— quelles sont les étapes de la validation de la condition,
— quels sont les effets si la condition est vérifiée, et si elle n'est pas vérifiée.

2.10 Conception d'algorithmes

Plusieurs éléments de code (algorithmes) doivent être ajoutés pour créer les fonctionnalités demandées. On en demande deux :

Dossier 18 . Concevez et détaillez de la même manière que l'algorithme analysé précédemment la validation de la condition *chaque ressource nécessaire doit être présente, et ce en nombre suffisant*, et des effets subséquents.

Dossier 19 . L'effet *échange possible* (`CanExchange`) permet à un joueur qui a posé un bâtiment disposant de cet effet d'échanger une carte de sa main contre une nouvelle. On utilisera une nouvelle méthode pour cela : indiquez précisément quoi modifier, quelles vérifications sont à effectuer, et quels éléments (données, méthodes...) devraient, selon vous, être ajoutés, et dans quelles classes.

2.11 Modification de la structure

La réalisation du jeu détaillée en section 2.12 pourra nécessiter des changements : ajouts d'attributs, de méthodes, de classes, modifications d'associations.

Dossier 20 . Donnez les **modifications** à apporter au diagramme de classes, par rapport au code initial, en ne détaillant que les éléments **modifiés** sous forme d'un diagramme de classes par paquets.

2.12 La réalisation du jeu

Tout ce qui suit sera évalué dans le rendu final de l'application.

2.13 Choix des cartes à jouer

Il s'agit de remplacer le bouton *jouer toutes les cartes* par un choix réfléchi du joueur des cartes qu'il souhaite jouer, et dans quel ordre.

Réalisation 21 . Rendez les cartes visibles **cliquables** ; durant la phase de jeu, le joueur clique sur les cartes qu'il souhaite jouer.

Réalisation 22 . Une fois les cartes rendues jouables une par une, retirez le bouton permettant de jouer toutes les cartes – ou restreignez-le à la pose de bâtiments ne nécessitant pas de ressources, à votre choix.

2.14 Code des ressources

Il s'agit de rendre utilisable les `ressources` : ce sont des cartes consommées par la construction de certains bâtiments. Ces cartes ne doivent être retirées que si les

conditions sont remplies, elles sont dans ce cas immédiatement défaussées.

Réalisation 23 . Coder, selon l'algorithme préparé en section 2.10, l'utilisation des ressources.

2.15 Pile de défausse

À côté de la pioche (pile de jeu, `deck` en anglais et dans le code) se place la *pile de défausse* ; ce sont les cartes utilisées. Pour l'instant, les ressources consommées seront posées sur cette pile. Quand la pile de jeu est vide, la défausse est mélangée puis utilisée pour les cartes suivantes, sauf si la défausse est vide.

Réalisation 24 . Coder la pile de défausse.

Réalisation 25 . Ajouter, dans la fenêtre, des éléments d'interface permettant de visualiser la taille des piles de jeu et de défausse.

2.16 Phase de défausse et choix des cartes à défausser

La phase de défausse donnée en section 2.1 est prévue dans le code, mais elle est **ellipsée** (non effectuée) pour l'instant : les joueurs ne peuvent défausser que les ressources consommées ou les cartes action jouées...

Réalisation 26 . Ajoutez la phase de défausse dans le cycle des phases du tour en posant les messages adaptés, avec un bouton permettant d'y mettre fin (éventuellement un bouton existant).

Réalisation 27 . En phase de défausse, cliquer sur une carte doit permettre de s'en défausser.

2.17 Code des actions

Quelques cartes sont des **actions** qui sont à jouer pour un effet immédiat.

Réalisation 28 . Ajouter le code permettant de jouer une carte de catégorie `Action`, sans effet pour l'instant (voir en section suivante pour l'ajout des effets des actions). La carte est défaussée.

2.18 Code des effets

Pour avoir un jeu intéressant, le code des effets doit être implémenté. Attention, il s'agit de l'aspect qui nécessitera le plus de travail dans la logique de jeu ; procéder in-

crémentalement et précautionneusement... L'utilisation de la gestion de versions n'est ni exigée ni évaluée, mais fortement recommandée.

Réalisation 29 . Coder les effets des bâtiments et actions suivant le descriptif ci-dessous :

- **Peut échanger** – `CanExchange`, pour un `CardType` spécifié : le joueur qui a posé un bâtiment avec cet effet peut, pendant sa phase de jeu, défausser autant de cartes du type spécifié dans le bâtiment que souhaité et tirer une nouvelle carte à chaque fois. **Nécessite des ajouts dans l'interface utilisateur.**
- **Une carte en plus** – `OneMoreCard` : la taille maximale de la main du joueur passe de cinq à six cartes s'il a posé ce bâtiment. Non cumulable.
- **Tirer une fois par tour** – `DrawOncePerTurn`, pour un `CardType` spécifié : le joueur qui a posé un bâtiment avec cet effet peut, pendant sa phase de jeu, tirer la prochaine carte du type indiqué de la pioche. S'il n'y a aucune carte de ce type dans la pioche, par exemple si la pioche est vide, il ne se passe rien. **Nécessite des ajouts dans l'interface utilisateur.**
- **Récupérer dans la défausse une fois par tour** – `DrawFromDiscardOncePerTurn`, pour un `CardType` spécifié : le joueur qui a posé un bâtiment avec cet effet peut, pendant sa phase de jeu, tirer la prochaine carte du type indiqué de la défausse. S'il n'y a aucune carte de ce type dans la défausse, par exemple si la défausse est vide, il ne se passe rien. **Nécessite des ajouts dans l'interface utilisateur.**
- **Produire un exemplaire** – `ProducesOne`, pour un `CardType` spécifié : le joueur qui a posé un bâtiment avec cet effet est considéré comme ayant une carte de plus du type spécifié pour chaque placement de bâtiment.
- **Nouveau tour** – `PlayAgain` : le joueur qui joue cette action rejoue immédiatement, en tirant de nouveau les cartes nécessaires pour refaire sa main, après avoir défaussée cette carte.
- **Substitut** – `Substitute`, pour un `CardType` spécifié : après avoir joué cette carte et uniquement pour le reste du tour, le joueur est considéré comme ayant une quantité illimitée de cartes du type spécifié.

2.19 Toutes les cartes

Réalisation 30 . À mesure que les effets et interfaces graphiques sont codées, ajouter les cartes à la pioche de base, en quantité déterminée en annexe, section 4.1.

2.20 Utilisation des effets

Certains effets de cartes nécessitent une interface spécifique pour être utilisés.

Réalisation 31 . Ajouter une visualisation de la **liste des effets disponibles à ce tour pour le joueur actif** qui permette de savoir ce qui peut être fait.

Réalisation 32 . Pour l'effet `CanExchange`, ajoutez une interface permettant de sélectionner les cartes à échanger, par exemple un clic droit ou un bouton à cocher.

Réalisation 33 . Pour les effets `DrawOncePerTurn` et `DrawFromDiscardOncePerTurn`, permettre de cliquer sur la pioche et la défausse pour récupérer la ou les cartes concernées.

Réalisation 34 . Pour chaque effet utilisé, affichez un message approprié.

2.21 Interface plus utilisable

À ce stade, le jeu devrait être *jouable*, mais son interface peut sans doute être améliorée...

Thème visuel

La fenêtre de départ pourra vous sembler peu agréable...

Réalisation 35 . Personnalisez-la avec vos couleurs, tailles, dispositions et valeurs de la classe `ViewSettings` prévue à cet effet.

Sixième carte

Réalisation 36 . Pour l'effet `OneMoreCard`, la fenêtre n'est (peut-être) pas adaptée à la réception de six cartes au lieu de cinq, ajustez.

Dialogues de confirmation

Beaucoup d'actions sont coûteuses : échanger des cartes, poser une carte nécessitant des ressources, etc. Sans demande à l'utilisateur, ce dernier peut se retrouver dans une situation problématique.

Réalisation 37 . À chaque action le nécessitant, affichez un dialogue de confirmation (soit un formulaire créé par vos soins, soit une `MessageBox`, par exemple). N'effectuez pas l'action si le joueur ne confirme pas.

Regroupement des bâtiments identiques

Les bâtiments ont peu d'espace pour leur affichage, et il peut s'en accumuler beaucoup plus que l'espace disponible.

Réalisation 38 . Au lieu d'afficher un rectangle par bâtiment, n'utiliser qu'un rectangle par nouveau type de bâtiment et ajouter le nombre de bâtiments identiques à côté du nom de l'élément concerné.

2.22 Compléments

N'abordez cette partie qu'après tous les éléments précédents !

2.23 Jeu à score ciblé

Le jeu est en **nombre de tours limité** ; pour ne pas avoir de limite au nombre de tours, il suffit d'initialiser la propriété voulue avec une valeur nulle ou négative.

Complément 39 . Ajoutez un mode de jeu *à score ciblé* : au lieu d'attendre un certain nombre de tours, le gagnant sera **le premier à atteindre un certain score** (par défaut 60, avec le jeu complet). Modifiez le code de `Controller` et le dialogue de début de jeu en conséquence.

2.24 Interface plus agréable

Détails suivant pointeur

Complément 40 . Faites en sorte que passer la souris au-dessus d'un bâtiment ou d'une carte donne ses détails (la description notamment) dans une zone visible de la fenêtre principale.

Messages de sortie

Complément 41 . L'objectif est de se passer des messages de sortie et de la `ListBox` en utilisant au mieux l'interface utilisateur.

2.25 Interface plus agréable : préférences d'affichage

Complément 42 . Ajoutez une boîte de dialogue permettant de modifier les préférences d'affichage de la classe `ViewSettings`, dans des valeurs raisonnables.

Réaménagements utilisateur

Complément 43 . Ajoutez un *mode déplacement* permettant à l'utilisateur de réagencer les éléments visibles : bâtiments, cartes, éventuellement autres éléments de l'interface.

2.26 Interface plus esthétique

Complément 44 . Améliorez l'esthétique et l'ergonomie de l'interface au maximum pour donner envie de jouer. Profitez-en pour écrire de meilleures descriptions.

2.27 Jeu contre l'ordinateur

Complément 45 . Ajoutez une option au dialogue de départ permettant d'avoir un joueur simulé, dit *IA* (qui joue au hasard parmi les actions possibles... ou qui joue intelligemment, suivant le temps que vous souhaitez investir).

2.28 Construction de jeu

Ce complément est à faire en dernier, pour aller plus loin. Dans un jeu de *deck building*, les joueurs commencent par personnaliser la pile de cartes dans laquelle ils vont piocher, avant de jouer la partie.

Complément 46 . Créez un formulaire permettant de composer un paquet de cartes, à partir des cartes disponibles. Il sera appelé pour chaque joueur. Les joueurs vont donc jouer avec des pioches et défausses différentes.

Chapitre 3

Récapitulatif des livrables

3.1 Rendus en R2.01

Un **unique fichier PDF**, le dossier d'analyse et de conception décrit en section 2.7, à rendre **au plus tard le 31 mars 2024 à 23:59**, délai de rigueur, sur Moodle.

3.2 Rendus en R2.02

- Une archive contenant : un fichier texte avec le nom des auteurs et les réponses aux questions du tutoriel ; le code (projet avec solution) avec le tutoriel terminé.
- Une archive contenant tout le code terminé (projet avec solution) et un fichier texte contenant :
  - le nom des auteurs,
  - la liste de tous les éléments **effectivement** réalisés, compléments compris, en donnant tous les numéros des éléments traités.

Chapitre 4

Annexes

4.1 Récapitulatif de toutes les cartes avec leurs règles et fonctionnalités

Tableau des cartes :

#	Nom ang.	Nom fr.	Catégorie	Description	Commentaires
30	Classroom	Salle de cours	Bâtiment	*Une simple salle de cours.*	Aucun pré-requis, aucun effet
10	Dorm	Résidence universitaire	Bâtiment	*Permet de loger des étudiants.*	Pré-requis : deux salles de cours construites. Rapporte 1 point de victoire.
2	Cafeteria	Restaurant universitaire	Bâtiment	*Permet de sustenter des étudiants.*	Pré-requis : trois salles de cours et deux résidences universitaires construites. Rapporte 5 points de victoire.
3	Faculty Lounge	Salle des professeurs	Bâtiment	*Habitat pour les universitaires.*	Pré-requis : deux salles de cours construites. Rapporte 1 point de victoire, et permet d'échanger des salles de cours contre des cartes à piocher.

7	Funding	Financement	Ressource	*Nourriture consommée par les universitaires.*	
7	A-List Publication	Publication de rang A	Ressource	*Substance produite par les universitaires.*	
10	Research Lab	Laboratoire de Recherche	Bâtiment	*Lieu favorable à la production des universitaires.*	Pré-requis : une salle de cours et une salle des professeurs construites, consomme deux financements. Rapporte 7 points de victoire et permet d'échanger des publications de rang A contre de nouvelles cartes à piocher.
2	Library	Bibliothèque	Bâtiment	*Mémoire collective.*	Unique (ne peut être construit qu'une fois par joueur). Pré-requis : deux résidences universitaires et un restaurant universitaire construites. Rapporte 7 points de victoire et permet de conserver une carte supplémentaire en main (jusqu'à six, donc).
2	Grand Hall	Salon présidentiel	Bâtiment	*Appeau à financements.*	Unique. Pré-requis : deux salles des professeurs et deux laboratoires de recherche construits, consomme trois financements. Rapporte 13 points de victoire et permet de tirer un financement par tour.

2	Server farm	Enclos à serveurs	Bâtiment	*Salle de consommation de ressources diverses.*	Unique. Pré-requis : cinq salles de cours et deux laboratoires de recherche construits. Rapporte 7 points de victoire et permet d'échanger des financements contre de nouvelles cartes à piocher.
2	Archive	Archives	Bâtiment	*Stockage de long terme.*	Unique. Pré-requis : avoir construit la bibliothèque, consomme un financement. Rapporte 17 points de victoire et permet de récupérer une publication de rang A de la défausse à chaque tour.
2	Museum	Musée	Bâtiment	*Permet au public de voir des bidules compliqués*	Unique. Pré-requis : avoir construit les archives. Rapporte 19 points de victoire et produit une publication de rang A par tour.
2	Aquarium	Aquarium	Bâtiment	*Permet au public de dépenser son argent.*	Unique. Pré-requis : avoir construit les archives. Rapporte 19 points de victoire et produit un financement par tour.
2	Fablab	Fablab	Bâtiment	*Permet de détourner des financements.*	Unique. Pré-requis : trois laboratoires de recherche construits, consomme trois financements. Rapporte 19 points de victoire et permet de tirer un financement par tour.

1	Super computer	Super-calculateur	Bâtiment	*Apothéose de l'esprit mathématique.*	Unique. Pré-requis : avoir construit l'enclos de serveurs et cinq laboratoires de recherche, consomme cinq financements et trois publications de rang A. Rapporte 50 points de victoire.
1	Particle Collider	Accélérateur-collisionneur de particules	Bâtiment	*Machine de l'apocalypse.*	Unique. Pré-requis : avoir construit le salon présidentiel, les archives, trois laboratoires de recherche, consomme trois financements et cinq publications de rang A. Rapporte 50 points de victoire.
3	Official Visit	Visite officielle	Action	*Captation d'une personnalité à des fins d'hyperactivité.*	Pour une publication de rang A, permet de rejouer immédiatement.
2	Over-hyped Keyword	Mot-clé de par trop surfait	Action	*Si c'est à la mode, tirons-en le maximum.*	Permet d'ignorer, à ce tour, les consommations de financements.
2	Prestigious Award	Prix prestigieux	Action	*Carpe diem.*	Permet d'ignorer, à ce tour, les consommations de publications de rang A.

4.2 Compléments de C#

Visibilité par défaut

Les membres des `enum` et `interface` sont par défaut publics en C#, tandis que les membres des classes sont privés par défaut. La visibilité dans l'espace de noms de la classe (paquet) est obtenue par le mot-clé `internal`. Par défaut, les classes sont `internal`, déclarez-les explicitement publiques.

Énumération

Comme en Java, les énumérations ou **types énumérés** sont déclarées comme des classes en remplaçant le mot-clé `class` par `enum`. Les éléments d'un type énuméré ont un nom commençant par une majuscule (mais pas tout en majuscule), comme les constantes. Contrairement à Java, les types énumérés ne peuvent pas contenir de méthodes ou constructeurs, juste la liste des valeurs qu'ils peuvent prendre ; une variable déclarée avec un type énuméré a forcément une valeur, `null` ne peut pas être utilisé.

Abbréviations

Les méthodes simples comme :

```
1 public X Y() {
2   return Z;
3 }
```

...sont souvent abrégées en : `public X Y() => Z;` – rien ne change, mais l'écriture est faite sur une ligne.

Types nullables

En C#, poser un `?` après un type représentant une valeur (une énumération, un entier, un booléen...) le rend **nullable**, c'est-à-dire capable de recevoir la valeur `null`. Par exemple : `int? a=null;`.

Tuples

C# permet d'utiliser des `tuples`, connus en Python, avec la notation `(a, b, c, ...)` : éléments du tuple entre parenthèses séparés par des virgules. On peut ainsi écrire l'échange de deux variables a et b avec des couples : `(a, b)=(b, a);`.

Conteneurs

L'équivalent du type `ArrayList` en Java s'écrit `List`, l'accès (`get`) se faisant avec les méthodes `ElementAt` ou simplement les crochets `[]`. Le type `HashSet` existe et fonctionne de manière similaire à Java. L'équivalent d'une `HashMap` est donnée par le type `Dictionary`, les crochets `[]` permettant d'accéder à une valeur à partir de la clé.

Classes statiques

C# permet d'utiliser le mot-clé *static* pour qualifier une classe ; cette classe ne pourra alors contenir que des éléments de classe (*static* aussi). Elle ne sera jamais instanciée (comme une classe abstraite).

Polymorphisme

Le polymorphisme n'est pas utilisé dans le code fourni, mais vous pouvez l'utiliser si vous le souhaitez.

Héritage et implémentation d'interfaces sont notées par `:` (deux points). Les méthodes et propriétés redéfinies le sont avec le mot-clé `override` ; la méthode dont la définition est ainsi remplacée doit être abstraite (`abstract`) ou déclarée explicitement avec le mot-clé `virtual`. Une méthode disposant du mot-clé `override` pourra ensuite être redéfinie dans ses propres sous-classes. Le mot-clé `sealed` peut être utilisé pour interdire d'hériter d'une classe ou de redéfinir une méthode, quand cette dernière a déjà été redéfinie (pour interdire de redéfinir une méthode en donnant sa définition avec sa déclaration, il suffit de ne pas la déclarer `virtual`).

L'équivalent du mot-clé `super` en Java (référence au constructeur ou à une méthode de la super-classe immédiate) est `base` en C#.

Attention, C# définit trois niveaux d'accessibilités – donc trois visibilités – en lien avec l'héritage :

1. `private protected` : le plus restreint. Uniquement la classe ou ses classes héritées si elles sont dans l'espace de nom (paquet) de la classe.

2. `protected` : intermédiaire. Uniquement la classe ou ses classes héritées, y compris hors de l'espace de nom (paquet) de la classe.

3. `protected internal` : le plus ouvert. Toutes les classes de l'espaces de nom (paquet) de la classe, plus les classes héritées en-dehors. Équivalent à `protected` de Java.

Égalité et code de hachage

On peut redéfinir les méthodes `Equals` et `GetHashCode`, Visual Studio peut également s'en charger, voir un exemple dans la classe `CardType`.

Table des matières