

# **Projet Logiciel Transversal**

**JAMET Romain DORRA Benjamin**

# Table des matières

<b>1</b>	<b>Présentation Générale</b>	<b>1</b>
1.1	Archétype . . . . .	1
1.2	Règles du jeu . . . . .	1
1.3	Ressources . . . . .	2
<b>2</b>	<b>Description et conception des états</b>	<b>5</b>
2.1	Description des états . . . . .	5
2.2	Conception Logiciel . . . . .	8
<b>3</b>	<b>Rendu : Stratégie et Conception</b>	<b>10</b>
3.1	Classe principale . . . . .	11
3.2	Stratégie de rendu . . . . .	11
3.3	Interactivité . . . . .	12
3.4	Stratégie de rendu d'un état . . . . .	12
3.5	Conception logiciel . . . . .	12
<b>4</b>	<b>Règles de changement d'états et moteur de jeu</b>	<b>14</b>
4.1	Règles . . . . .	14
4.2	Conception logiciel . . . . .	14
<b>5</b>	<b>Intelligence Artificielle</b>	<b>16</b>
5.1	Stratégies . . . . .	16
5.1.1	IA aléatoire . . . . .	16
5.2	Conception logiciel . . . . .	17
<b>6</b>	<b>Modularisation</b>	<b>19</b>
6.1	Organisation des modules . . . . .	19
6.2	Conception logiciel . . . . .	20

# 1 Présentation Générale

## 1.1 Archétype

Le jeu d'origine est le jeu de stratégie/gestion Crusader Kings II, développé et édité par le studio Paradox Interactive.

Il s'agit d'un jeu se déroulant dans l'Europe et le Moyen-Orient entre le VIII<sup>e</sup> et le XV<sup>e</sup> siècle, où les joueurs jouent le rôle d'une dynastie de nobles régnant sur un domaine, qu'ils peuvent faire croître par une mécanique de titres de noblesse, de guerre et de diplomatie inspirés du Moyen-Âge. Les règles en seront bien évidemment simplifiées dans le cadre du projet.

Le descriptif des règles ci-dessous présente le projet tel qu'il devrait être au final.

## 1.2 Règles du jeu

Les joueurs sont des nobles qui règnent sur ces provinces, en possédant un ou plusieurs titres de noblesse. Les trois titres sont, dans l'ordre du plus puissant au moins puissant : les Rois, les Ducs, et les Comtes.

À chaque province est lié un titre de Comte. Posséder un titre de Comte, c'est posséder la province associée.

Une province possède plusieurs caractéristiques décrivant son niveau de développement, son état de dévastation ou de prospérité, qui influent sur les impôts qu'elle génère, et les levées militaires qu'elle peut supporter. Un Comte peut posséder plusieurs Comtés, cependant, le nombre en est limité. Si cette limite est dépassée, un malus d'impôts et de levées est appliqué.

Un Duc est plus puissant qu'un Comte. Pour pouvoir contrôler plus de territoire malgré la limite mentionnée précédemment, il doit confier la gestion de certaines provinces à des nobles qui seront ses vassaux.

Un Duc ne peut avoir que des Comtes pour vassaux.

Un Roi est plus puissant qu'un Duc. Il peut avoir comme vassaux des Ducs et des Comtes. Lorsqu'un seigneur est vassal, il paie un impôt en or et en hommes à son suzerain.

Pour faire la guerre à son voisin, il faut un Casus Belli. Il y aura la possibilité de dépenser de l'or pour fabriquer une revendication à un titre de Comte, pour essayer de s'en emparer à la guerre. Une fois la guerre déclarée, on peut mobiliser les troupes dans les provinces que l'on possède, ainsi que les troupes de ses vassaux si on en a, et leur donner des ordres de déplacement sur la carte.

Lorsque deux armées hostiles se rencontrent, un combat a lieu. L'armée défaite est mise en déroute et cherchera automatiquement à revenir à sa province d'origine.

Chaque Comté fait partie d'un Duché «de Jure», c'est-à-dire un Duché dont le titre n'existe pas, mais qui peut être créé si quelqu'un en contrôle plus de la moitié et dépense une certaine somme. Le fait de posséder un titre de Duc offre un Casus Belli permettant de vassaliser les Comtes dont les terres font partie «de Jure» de votre Duché.

De même, chaque Duché fait partie d'un Royaume de Jure. Posséder ou contrôler par le biais de vassaux une certaine proportion d'un Royaume de Jure vous permet de créer ce titre en échange d'argent. Posséder un tel titre offre un Casus Belli contre tous les Ducs et Comtes faisant partie de Jure de votre Royaume. On peut bien évidemment posséder ou contrôler par vassaux des provinces n'appartenant pas de Jure à votre domaine.

Il est possible d'interagir avec d'autres personnages, forger des alliances, des pactes de non-agression, proposer à un seigneur plus faible de devenir votre vassal.

Chaque personnage possède un score décrivant son opinion envers les autres, influencé par l'historique de leurs actions. Ce score permettra de définir le comportement de l'IA. Il sera également influencé par des traits de personnalité des personnages. L'IA prendra ses décisions en utilisant une mécanique de MMTH (mean time to happen, ou temps moyen d'attente). Pour chaque action, il y a un MTTH en nombre de tours, avec des modificateurs liés aux traits de personnalité aléatoires des personnages.

Au cours de son règne, un personnage accumule des points de prestige selon la quantité de titres qu'il possède, ainsi que les titres de ses vassaux. Lorsqu'il meurt, un héritier apparaît, et le score de prestige s'ajoute au score du joueur. Le score final d'un joueur correspond donc à la somme de points de prestige accumulés par tous les seigneurs successifs qu'il a joué.

### 1.3 Ressources

Pour réaliser ce projet, nous aurons besoin en termes de ressources d'une carte du monde du jeu, avec les différentes provinces découpées.

Nous aurons besoin d'une version pour l'affichage, et d'une autre qui ne sera pas affichée pour identifier la province sur laquelle le joueur clique, par un code couleur unique par province.

Nous aurons besoin de sprites d'armée immobile, en mouvement, et à la bataille.

Nous devons pouvoir coloriser facilement les bannières pour identifier immédiatement le contrôleur de l'armée.

Nous pouvons également avoir besoin d'images pour les personnages, même si ce n'est pas obligatoire il s'agirait d'un plus. Dans le même ordre d'idées, nous pourrions employer diverses images sur le thème médiéval pour illustrer les différents menus du jeu. Nous pouvons aussi utiliser différentes musiques d'ambiance, des bruitages de bataille, et quelques sons génériques pour l'appui sur les boutons de l'interface.

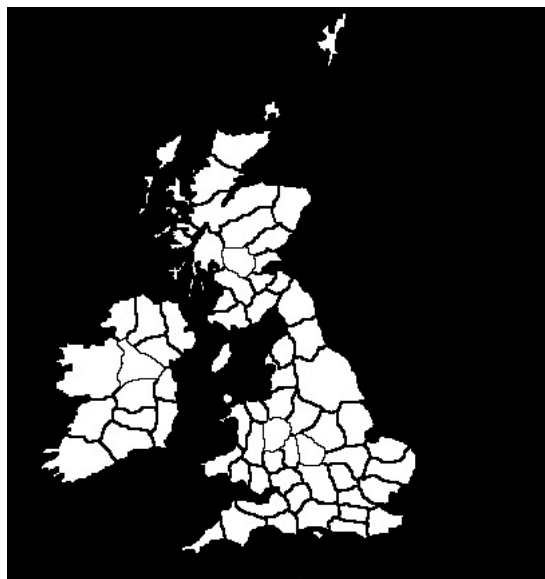


FIGURE 1 – carte affichée - royaume-uni

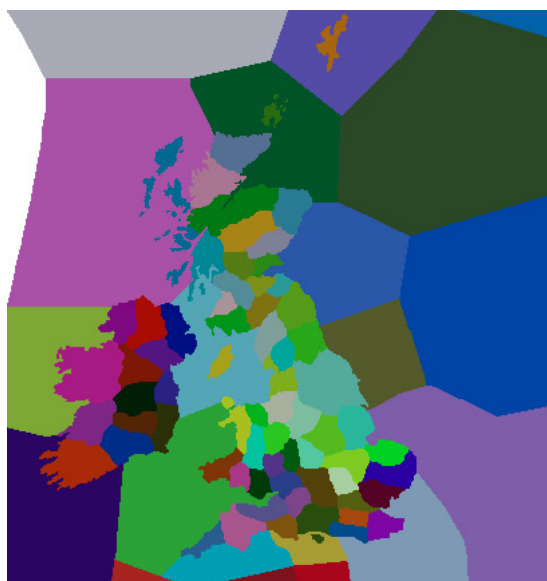


FIGURE 2 – carte non affichée - royaume-uni



FIGURE 3 – sprite pour une armée (temporaire)

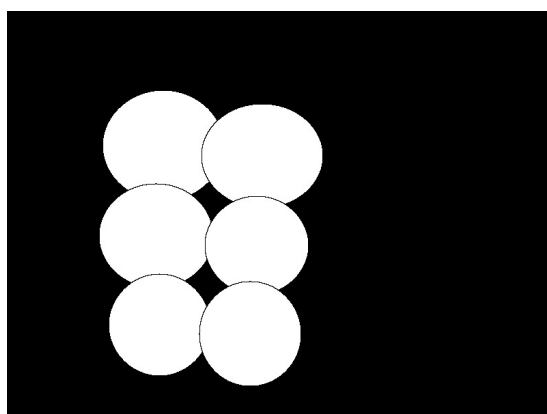


FIGURE 4 – carte affichée - test map (temporaire)

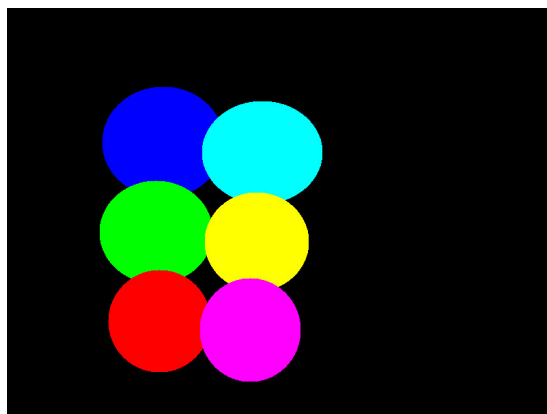


FIGURE 5 – carte non affichée - test map (temporaire)

## 2 Description et conception des états

L'état de jeu est contenu dans une classe principale : GameState, qui contient tous les éléments du jeu propres à l'état d'une partie de jeu. Certains éléments de l'état de jeu ont besoin de références à d'autres éléments. Afin d'être compatible avec une exécution rapide en jeu et un système de sauvegarde sûr, les éléments auxquels il peut être fait référence sont dotés d'un champ id unique, qui sert à les identifier. Ainsi, pour référencer le personnage propriétaire d'un titre dans la classe Title on doit mettre l'id unique correspondant à ce personnage. Pour ce faire, on emploie des std : :map qui associent id et objet.

### 2.1 Description des états

L'organisation de l'état du jeu s'organise autour d'une classe principale, GameState. Elle contient les objets qui regroupent les différents éléments du jeu, ainsi que les principales fonctions permettant d'interagir avec l'état de jeu, que ce soit pour le consulter (fonctions fetch<...>, getCurrentTurn) ou le modifier (fonctions setArmyOrder pour assigner des ordres de déplacement aux armées, turnAdvance pour passer au tour suivant, updatePlayerCharacter pour assigner un nouveau personnage à un joueur, etc.). Les objets contenus dans la classe GameState sont :

- un objet ressources de la classe Ressources, chargé de contenir toutes les ressources statiques servant au fonctionnement du jeu.
- un objet gameMap de la classe GameMap, décrivant l'état des éléments dynamiques de la carte,
- un objet politics de la classe Politics, décrivant l'organisation géopolitique de la partie.
- une liste de joueurs (players) de la classe std : :vector<Player> pour suivre l'évolution des différents joueurs, humains ou IA.

La classe GameState charge les différents objets au démarrage du jeu à partir d'un fichier de sauvegarde json.

- Politics  
La classe Politics décrit l'ensemble des entités politiques existantes.
  - On a d'abord la liste des titres (titles), de la classe Titles. Il s'agit de tous les titres qui existent ou peuvent être créés.
  - On a ensuite la liste de tous les personnages (characters), de la classe Characters, qui contient tous les personnages en vie ou morts.
  - Enfin, on a la liste de toutes les relations (relations) qui peuvent exister entre deux personnages (alliances, etc.), de la classe std : :vector<Relation>.
- Titles  
La classe Titles décrit tous les titres qui existent ou peuvent être créés.
  - un champ titleType de type enum, qui peut prendre la valeur kingdom, county, ou duchy,
  - un champ titles de type std : :map<std : :string, titleType>,
  - un champ kingdoms de type std : :map<std : :string, Kingdom>
  - un champ duchies de type std : :map<std : :string, Duchy>
  - un champ counties de type std : :map<std : :string, County>
- Title  
La classe Title décrit un titre de noblesse, que celui-ci soit possédé par quelqu'un ou non. Il contient
  - un champ id de type std : :string,
  - un champ nom(name) de type std : :string,
  - un champ propriétaire (holder) de type std : :string

De la classe Title, héritent les classes County, Duchy, Kingdom (Comté, Duché, Royaume).

— County

- un champ province de type `std::string` qui désigne la province associée au titre
- un champ liege de type `std::string` qui désigne le suzerain
- un champ deJureLiege de type `std::string` qui désigne le suzerain de jure

— Duchy

- un champ liege de type `std::string` qui désigne le suzerain
- un champ deJureLiege de type `std::string` qui désigne le suzerain de jure

— Kingdom

— Characters

- le champ characters de type `std::map<std::string, Character>`
- un champ opinions de type `std::vector<std::string, std::map<std::string, int>` qui contient les opinions d'un personnage sur les autres,

— Character

La classe Character décrit les personnages du jeu. Les personnages peuvent posséder des titres, ont des statistiques, une opinion, des traits de caractère et peuvent mener des guerres ou comploter. Elle possède de nombreux attributs :

- un champ id de type `std::string`
- un champ name de type `std::string`
- un champ dynastyName de type `std::string` qui décrit la dynastie
- un champ age de type `int`
- un champ traits de type `std::vector<std::string>` qui contient les id des traits de caractère que possède le personnage
- un champ diplomacy de type `int` qui décrit son talent diplomatique
- un champ stewardship de type `int` qui décrit sa capacité à gérer les fiscalités de provinces
- un champ martial de type `int` qui décrit son talent martial
- un champ intrigue de type `int` qui décrit son talent pour les complots
- un champ claims de type `std::vector<std::string>` qui décrit les revendications du personnage. Cela lui permet de déclarer la guerre pour récupérer un titre sur lequel il a des revendications.
- un champ alive de type `bool`
- un champ prestige de type `int` qui décrit le score du personnage
- un champ gold de type `int`
- un champ plotTypes, énumération qui regroupe none, claim, murder
- un champ hasPlot de type `bool` qui décrit si le personnage a ou non un complot en cours
- un champ plotTarget de type `std::string` qui donne l'id de la cible d'un hypothétique complot, cette cible pouvant être un personnage ou un titre selon le type de complot.
- un champ plotType de type `plotTypes` qui décrit le type de complot (au plus un) mené actuellement par le personnage.
- un champ plotEnd de type `int` qui décrit le tour où le complot en cours se termine.

— Relation

La classe Relation décrit un lien particulier entre deux personnages actifs (vivants). Elle contient

- un champ characterA et
- un champ characterB, tous deux de type `std::string`,
- une énumération relType qui peut valoir : non\_aggression, alliance, friendship, rivalry, war et



- désigne les types de relation possibles,
- un champ type de type relType qui désigne le type de relation qui existe entre deux personnages,
- une date (numéro de tour) de fin (endTurn) qui vaut 0 dans le cas d'une durée indéterminée.
- Player
 

La classe Player décrit un joueur (humain ou IA). Les champs sont :

  - id de type std::string
  - currentCharacter de type std::string qui décrit le personnage actuellement contrôlé
  - score de type int qui décrit le score total du joueur
- AIPlayer
 

Une sous-classe de Player.
- HumanPlayer
 

Une sous-classe de Player.
- Ressources
 

La classe Ressources décrit les ressources statiques utilisées par le jeu. Elle contient les champs suivants :

  - un champ traits de type Traits qui contient tous les traits de caractère que peuvent avoir les personnages
  - un champ adjacencies de type Adjacencies qui contient le nombre de tours nécessaires à une armée pour passer d'une province à une autre.
- Adjacencies
 

La classe Adjacencies regroupe le nombre de tours nécessaires pour passer d'une province à une autre dans le champ val de type std::map<std::string, std::map<std::string, int>. Si les provinces ne sont pas adjacentes, la distance est 0. Entre une province et elle-même, la distance est -1.
- Traits
 

Cette classe regroupe tous les traits de caractère que peut avoir un personnage dans le champ traits de type std::vector<Trait>.
- Trait
 

Cette classe décrit un trait de caractère. Elle comprend :

  - un champ id de type std::string
  - un champ opposedId de type std::string qui donne l'id du trait de caractère opposé
  - diplomacyModifier de type int qui décrit l'effet qu'a le trait sur la statistique diplomacy
  - stewardshipModifier de type int qui décrit l'effet qu'a le trait sur la statistique stewardship
  - martialModifier de type int qui décrit l'effet qu'a le trait sur la statistique martial
  - intrigueModifier de type int qui décrit l'effet qu'a le trait sur la statistique intrigue
- GameMap
 

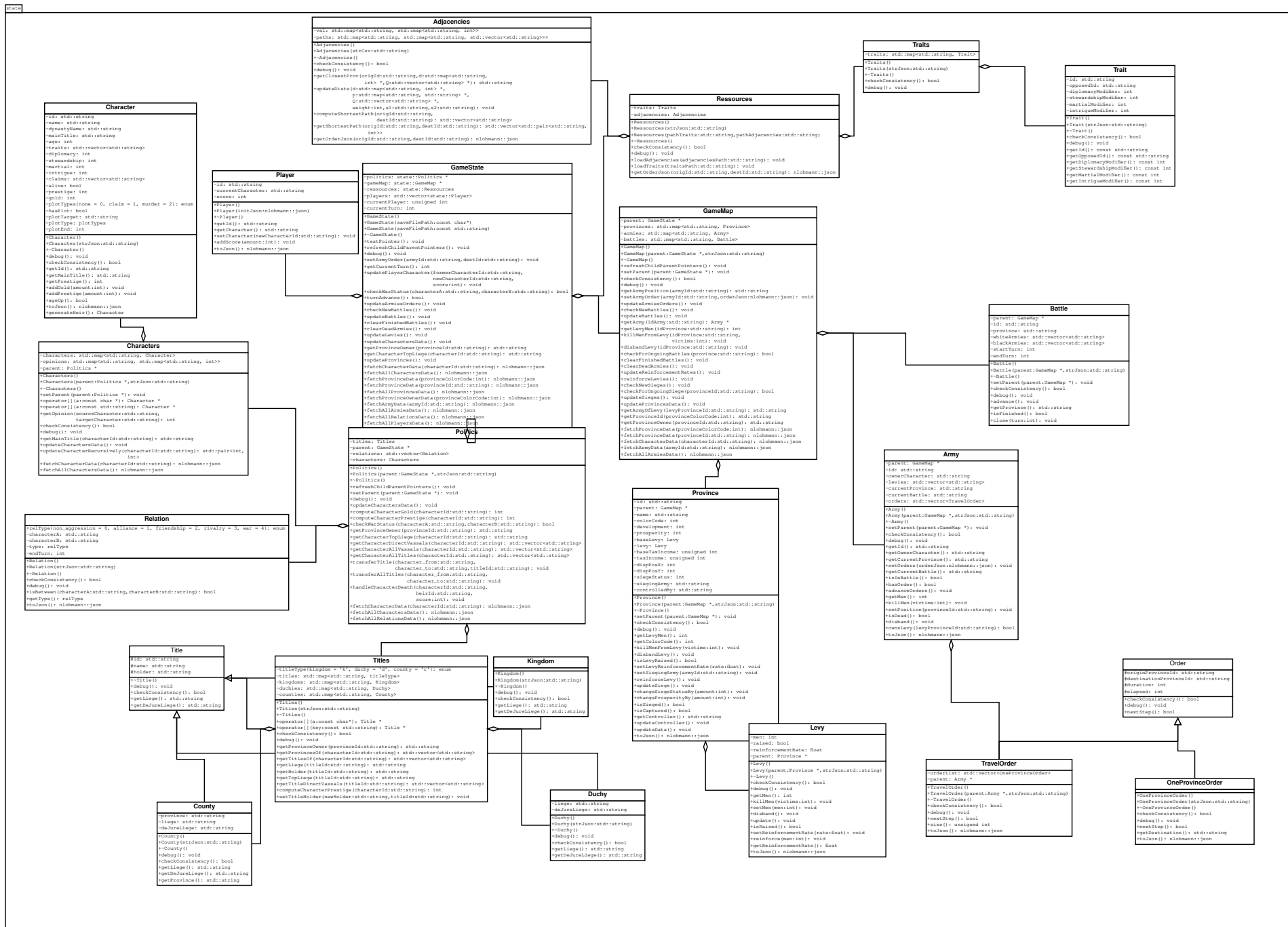
La classe GameMap décrit la situation des différentes zones, les batailles et les armées. Elle présente les champs suivants :

  - un champ provinces de type std::map<std::string, Province> qui regroupe les différentes provinces.
  - un champ armies de type std::map<std::string, Army> qui regroupe les différentes armées.
  - un champ battles de type std::map<std::string, Battle> qui regroupe les différentes batailles.
- Battle
 

Les champs sont :

- id de type `std : :string`
- province de type `std : :string` la province où a lieu la bataille
- whiteArmies de type `std : :vector<std : :string>`
- blackArmies de type `std : :vector<std : :string>`
- startTurn de type `int` le tour de début de la bataille
- endTurn de type `int` le tour de fin de la bataille
  
- Army  
 Décrit une armée. Contient les champs :
  - id de type `std : :string`
  - levies de type `std : :vector<std : :string>` les id des levées qui composent l'armée
  - currentProvince de type `std : :string` la position actuelle de l'armée
  - currentBattle de type `std : :string` la bataille actuellement menée par l'armée
  - orders de type `std : :queue<TravelOrder>` les ordres que suit l'armée.
- Order  
 Ordre donné à une armée. contient les champs :
  - originProvinceId de type `std : :string` id de la province de départ
  - destinationProvinceId de type `std : :string` idée de la province d'arrivée
  - duration de type `int` qui contient le temps total (en nombre de tours) nécessaire pour exécuter l'ordre
  - elapsed de type `int`, le nombre de tours écoulés depuis que l'ordre a été donné.
- TravelOrder  
 Ordre des provinces par lesquelles passe une armée. Hérite de Order. Contient un champ de type `std : :vector <OneProvinceOrder>`
- OneProvinceOrder  
 Hérite de Order. Détermine l'ordre de déplacement d'une armée province par province.
  
- Province  
 Cette classe décrit une province. Elle comprend les champs suivants :
  - id de type `std : :string`
  - name de type `std : :string`
  - colorCode de type `unsigned int`, identifiant de couleur unique qui permet d'identifier la province à sa position sur la carte.
  - development de type `unsigned char`
  - prosperity de type `char` qui détermine le niveau de prospérité de la province
  - baseLevy de type `Levy` la levée de base
  - levy de type `Levy` qui calcule la levée réelle après prise en compte de l'état de la province
  - baseTaxIncome de type `unsigned int` l'argent que la province rapporte de base
  - taxIncome de type `unsigned int` les impôts réels après prise en compte de l'état de la province
- Levy  
 Décrit une levée.
  - unsigned men de type `int` le nombre d'hommes dans la levée
  - isRaised de type `bool` qui décrit si la levée a été mobilisée
  - reinforcementRate de type `float` la vitesse de renouvellement de la levée

## 2.2 Conception Logiciel



### 3 Rendu : Stratégie et Conception

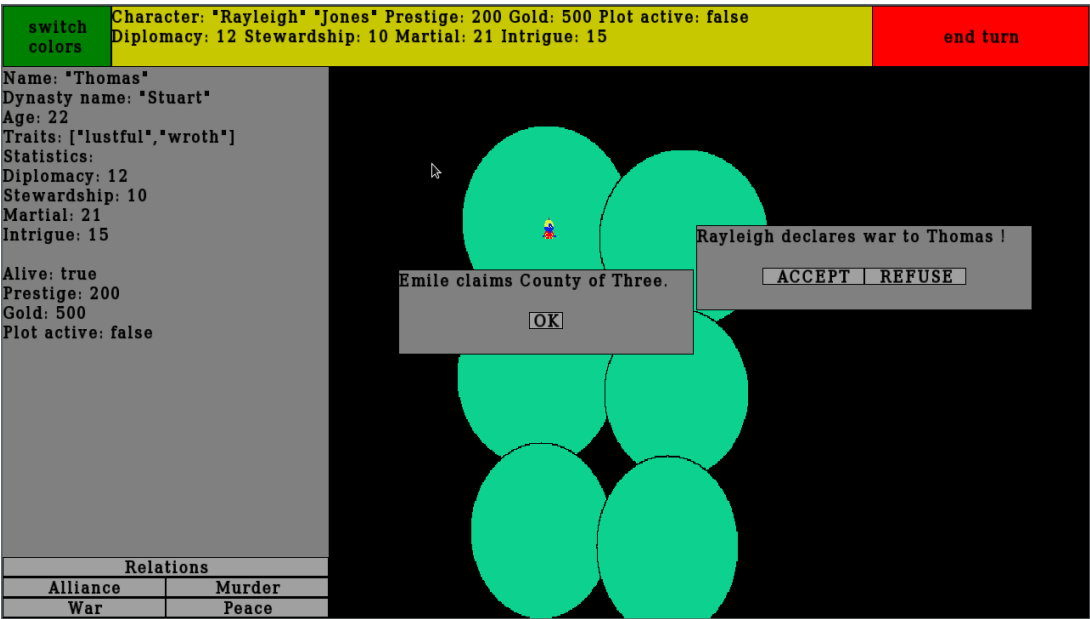


FIGURE 7 – Affichage du rendu

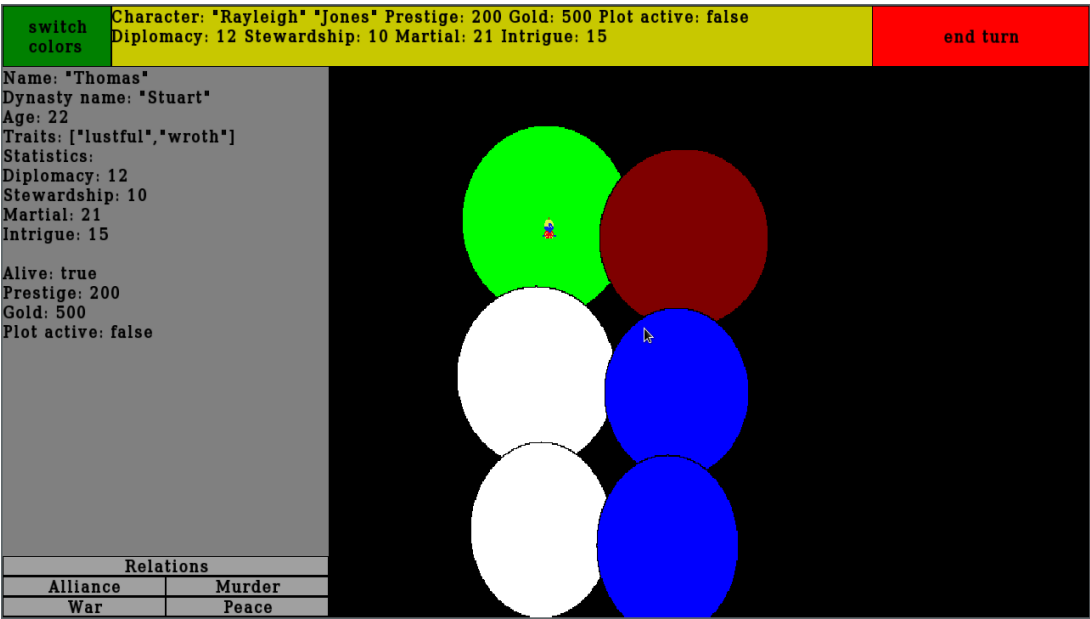


FIGURE 8 – Affichage des relations entre le personnage joueur et les autres

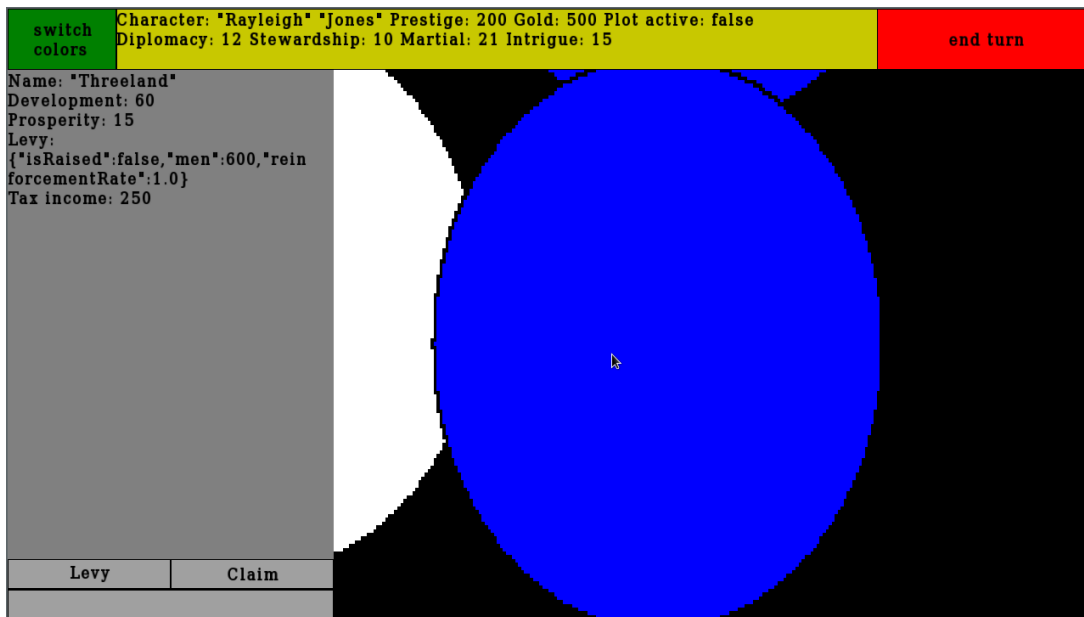


FIGURE 9 – Zoom et déplacement dans la carte

### 3.1 Classe principale

Notre stratégie générale consiste à créer un objet principal Render qui comprend une fenêtre qui va se rafraîchir régulièrement en utilisant la bibliothèque sfml (fonction `renderLoop()`).

Lors du rafraîchissement de la fenêtre, les divers éléments composant la page sont rafraîchis à partir des données du game state, puis affichés. La boucle principale dans `renderLoop()` comprend aussi la gestion des events, qui permet d'avoir une interactivité.

### 3.2 Stratégie de rendu

Les éléments graphiques héritent de la classe élément, qui contient un attribut pour la position horizontale, verticale, la hauteur et la largeur. Pour des raisons de simplicité, chaque élément graphique est en effet considéré comme un rectangle.

D'autre part chaque élément graphique dispose d'un pointeur `mainRender` vers l'objet Render, qui va leur permettre de demander à la fenêtre de les dessiner.

La stratégie de rendu est en effet la suivante :

- La boucle principale de `renderLoop()` appelle la fonction `draw()` des objets principaux (`ViewMap`, `Data`, `playerData` et le bouton de fin de tour)
- La fonction `draw()` de chacun de ces objets appelle la fonction `draw()` des objets qu'ils contiennent
- La fonction `draw()` des objets au bout de la chaîne utilisent le pointeur `mainRender` vers l'objet principal Render pour utiliser la méthode `RenderWindow.draw()` de sfml sur la fenêtre d'affichage.

La classe `Image` permet de simplifier la gestion des sprites et des textures de sfml.

La classe `ShowArmy` permet d'afficher une armée, et la classe `ShowArmies` permet de gérer une liste des armées à afficher.

La classe `PlayerData` permet d'afficher les données du joueur et du personnage qui lui est associé.

La classe Data permet d'afficher les informations d'une province, d'un joueur, d'une armée, ou des relations d'un joueur.

### 3.3 Interactivité

La stratégie d'interactivité est basée sur les events de sfml, gérés dans la boucle principale de la méthode renderLoop() de la classe Render.

Cette stratégie se base sur un switch selon le type d'événement, qui décrit les actions à effectuer selon le type d'évènement et l'objet qui doit être affecté. Un système de vérification auprès de chaque objet sur lequel ce type d'événement peut agir est déclenché pour savoir si la souris est dans cet objet (méthode contains() présente dans tous les objets interactifs, qui vont éventuellement appeler les méthodes contains des sous-objets contenus dans cet objet), puis une requête qui dépend du type d'événement est faite à cet objet. L'objet traite la requête (éventuellement en la transmettant au sous-objet concerné), et exécute l'action demandée.

Ainsi il est possible de scroller dans l'objet Data, de cliquer sur les boutons ainsi que de se déplacer (zqsd) et de zoomer (molette) dans la carte.

D'autre part les clics sur la carte chargent l'armée ou la province concernée (clic gauche) ou le propriétaire de la province (clic droit). Cela permet ensuite d'agir sur l'objet concerné. Si pour l'armée nous pouvons utiliser la méthode d'appels emboîtés décrite précédemment, il faut en revanche une stratégie particulière pour les provinces qui utilise les couleurs (uniques) d'une carte cachée (l'objet ColorMap, qui contient un objet sf : :Image de la carte cachée.).

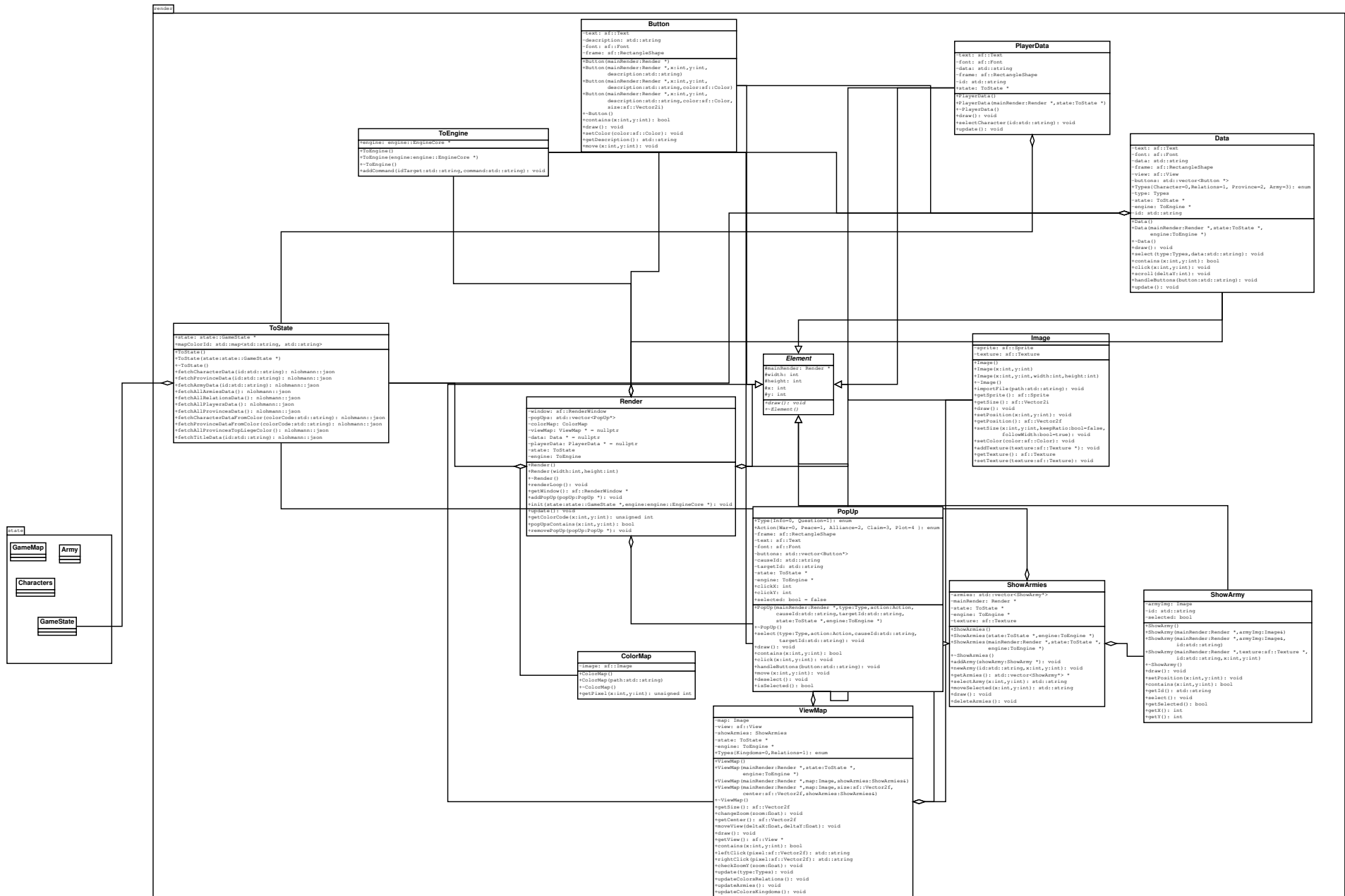
Les pop-ups sont affichés devant tous les autres objets peuvent être déplacés avec la souris, ou fermés en appuyant sur leurs boutons.

### 3.4 Stratégie de rendu d'un état

Pour effectuer le rendu d'un état nous utilisons les méthodes update() des classes concernées. Ces méthodes font appel à des méthodes de la classe ToState pour appeler les données de l'état de jeu, et adapter l'affichage à ces données. Elles sont appelées à chaque mise à jour du rendu par l'objet Render.

La classe ToState est une interface vers le gamestate, qui va appeler des méthodes fetch<...> de l'objet GameState par l'intermédiaire d'un pointeur.

### 3.5 Conception logiciel



## 4 Règles de changement d'états et moteur de jeu

Le moteur de jeu est constitué des classes EngineCore et Command.

La classe Command décrit une commande transmise au moteur de jeu. Elle comprend une énumération type, qui décrit le type de la commande et peut être un ordre de déplacement donné à une armée (moveArmy), une commande lancée par un bouton d'action ActionButton (qui peut désigner un assassinat, une requête sur une province, une déclaration de guerre ou de paix, une levée de troupes, ...), ou la commande de fin de tour (TurnButton), ainsi qu'un argument qui permet de désigner la cible de l'action.

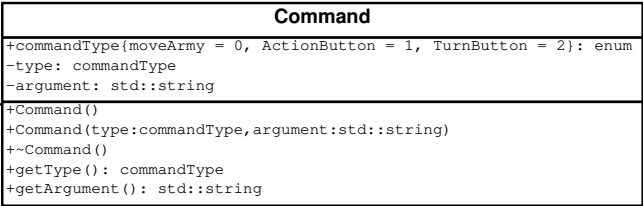
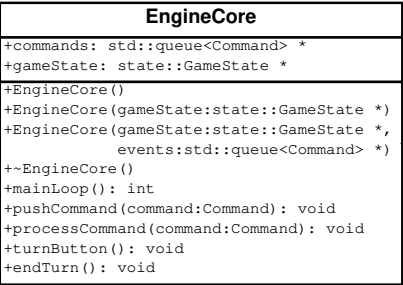
La classe EngineCore ajoute cette commande dans une liste de commandes à l'aide de la méthode pushCommand (commands), puis va traiter cette commande avec la fonction processCommand. si l'action est possible (par exemple on ne peut déplacer que ses propres armées), la commande est utilisée pour modifier l'état de jeu via les commandes de GameState, par l'intermédiaire d'un pointeur vers GameState.

### 4.1 Règles

Le moteur de jeu gère les fins de tour.

### 4.2 Conception logiciel





## 5 Intelligence Artificielle

### 5.1 Stratégies

#### 5.1.1 IA aléatoire

Pour commencer l'IA nous avons développé une classe simple RandomAI qui envoie des commandes au hasard (1 chance sur 3) parmi les différentes actions possibles : finir son tour, déplacer une armée, ou une action généraliste. La méthode randomAction de RandomAI répète des actions jusqu'à arriver à la fin du tour.

Pour les déplacements des armées l'armée et la province vers laquelle l'IA lui ordonne de se déplacer sont choisies aléatoirement.

Pour les actions généralistes, les actions sont choisies au hasard parmi toutes les actions possibles (lever une armée, déclarer la guerre, demander la paix, demander une alliance, exiger une province, assassiner un personnage) et la cible est choisie aléatoirement, avec une plus grande probabilité d'interagir avec un personnage qui possède un grand territoire.

## **5.2 Conception logiciel**

ai

### RandomAI

-toState: render::ToState  
-toEngine: render::ToEngine

+RandomAI(state:state::GameState \*,engine:engine::EngineCore \*)  
+~RandomAI()  
+randomActions(): void

## **6 Modularisation**

### **6.1 Organisation des modules**

## **6.2 Conception logiciel**