Projet Logiciel Transversal

JAMET Romain DORRA Benjamin

Table des matières

1 Présentation Générale **1**
1.1 Archétype . . . 1
1.2 Règles du jeu . . . 1
1.3 Ressources . . . 2

2 Description et conception des états **3**
2.1 Description des états . . . 3
2.2 Conception Logiciel . . . 6

3 Rendu : Stratégie et Conception **8**
3.1 Stratégie de rendu d'un état . . . 8
3.2 Conception logiciel . . . 10

4 Règles de changement d'états et moteur de jeu **12**
4.1 Règles . . . 12
4.2 Conception logiciel . . . 13

5 Intelligence Artificielle **13**
5.1 Stratégies . . . 13
5.2 Conception logiciel . . . 14

6 Modularisation **14**
6.1 Organisation des modules . . . 14
6.2 Conception logiciel . . . 15

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 VIIIe et le XVe siècle, où les joueurs jouent le rôle d'une dynastie de nobles régnant sur un domaine, qu'ils peuvent faire croitre 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 devrons 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 – royaume-uni

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

— GameState
  Elle contient :
  — 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.

— 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 diplomaty 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.

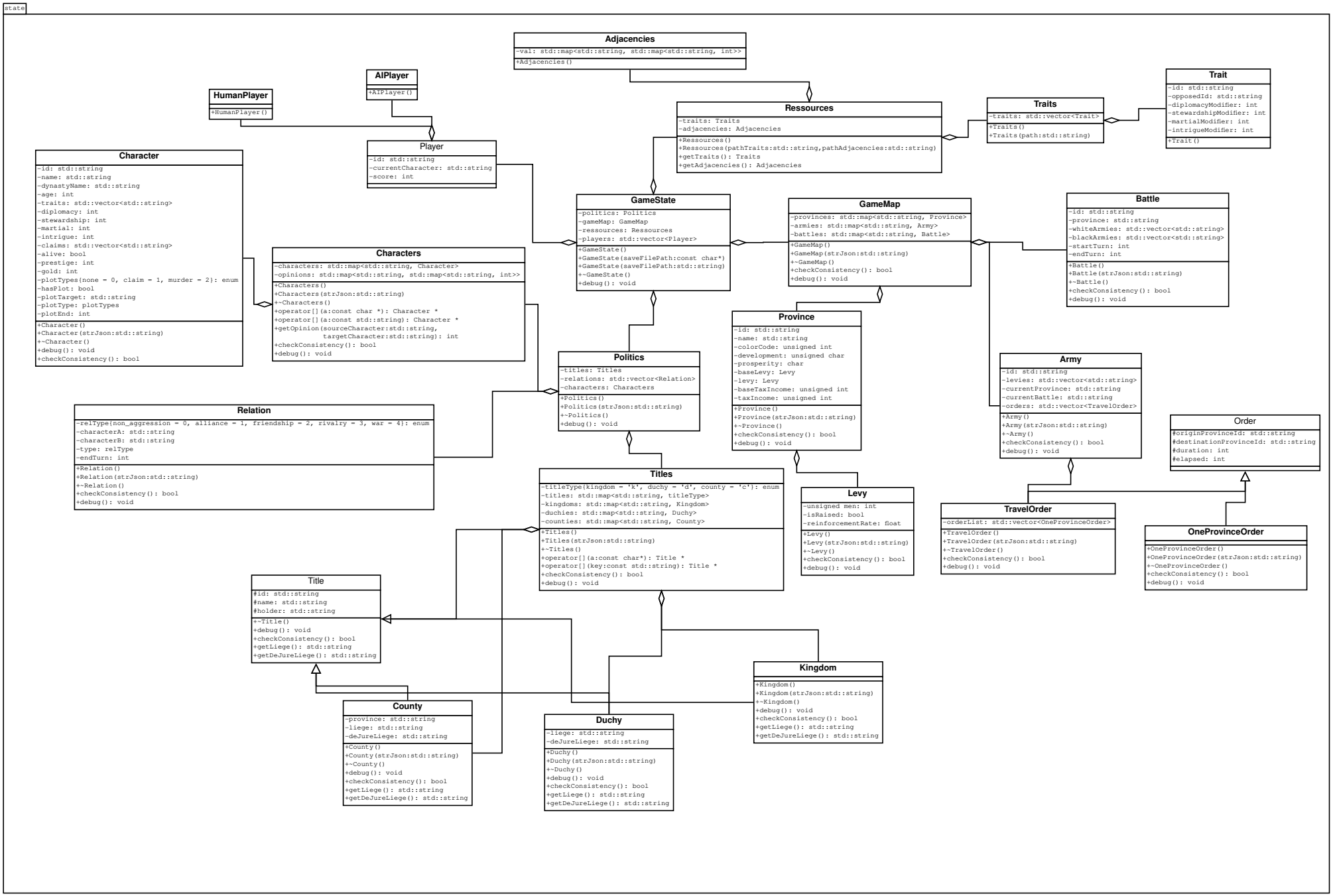
— TravelOrder
Ordre des provinces par lesquelles passe une armée. Hérite de Order. Contient un champ de type std : :vector <OneProvinceOrder>

— 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


state
Adjacencies
-val: std::map<std::string, std::map<std::string, int>>
+Adjacencies()
AIPlayer
+AIPlayer()
HumanPlayer
+HumanPlayer()
Player
-id: std::string
-currentCharacter: std::string
-score: int
Ressources
-traits: Traits
-adjacencies: Adjacencies
+Ressources()
+Ressources(pathTraits:std::string,pathAdjacencies:std::string)
+getTraits(): Traits
+getAdjacencies(): Adjacencies
Traits
-traits: std::vector<Trait>
+Traits()
+Traits(path:std::string)
Trait
-id: std::string
-opposedId: std::string
-diplomacyModifier: int
-stewardshipModifier: int
-martialModifier: int
-intrigueModifier: int
+Trait()
Character
-id: std::string
-name: std::string
-dynastyName: std::string
-age: int
-traits: std::vector<std::string>
-diplomacy: int
-stewardship: int
-martial: int
-intrigue: int
-claims: std::vector<std::string>
-alive: bool
-prestige: int
-gold: int
-plotTypes{none = 0, claim = 1, murder = 2}: enum
-hasPlot: bool
-plotTarget: std::string
-plotType: plotTypes
-plotEnd: int
+Character()
+Character(strJson:std::string)
+~Character()
+debug(): void
+checkConsistency(): bool
GameState
-politics: Politics
-gameMap: GameMap
-ressources: Ressources
-players: std::vector<Player>
+GameState()
+GameState(saveFilePath:const char*)
+GameState(saveFilePath:std::string)
+~GameState()
+debug(): void
GameMap
-provinces: std::map<std::string, Province>
-armies: std::map<std::string, Army>
-battles: std::map<std::string, Battle>
+GameMap()
+GameMap(strJson:std::string)
+~GameMap()
+checkConsistency(): bool
+debug(): void
Battle
-id: std::string
-province: std::string
-whiteArmies: std::vector<std::string>
-blackArmies: std::vector<std::string>
-startTurn: int
-endTurn: int
+Battle()
+Battle(strJson:std::string)
+~Battle()
+checkConsistency(): bool
+debug(): void
Characters
-characters: std::map<std::string, Character>
-opinions: std::map<std::string, std::map<std::string, int>>
+Characters()
+Characters(strJson:std::string)
+~Characters()
+operator[](a:const char *): Character *
+operator[](a:const std::string): Character *
+getOpinion(sourceCharacter:std::string,
targetCharacter:std::string): int
+checkConsistency(): bool
+debug(): void
Politics
-titles: Titles
-relations: std::vector<Relation>
-characters: Characters
+Politics()
+Politics(strJson:std::string)
+~Politics()
+debug(): void
Province
-id: std::string
-name: std::string
-colorCode: unsigned int
-development: unsigned char
-prosperity: char
-baseLevy: Levy
-levy: Levy
-baseTaxIncome: unsigned int
-taxIncome: unsigned int
+Province()
+Province(strJson:std::string)
+~Province()
+checkConsistency(): bool
+debug(): void
Army
-id: std::string
-levies: std::vector<std::string>
-currentProvince: std::string
-currentBattle: std::string
-orders: std::vector<TravelOrder>
+Army()
+Army(strJson:std::string)
+~Army()
+checkConsistency(): bool
+debug(): void
Order
#originProvinceId: std::string
#destinationProvinceId: std::string
#duration: int
#elapsed: int
Relation
-relType{non_aggression = 0, alliance = 1, friendship = 2, rivalry = 3, war = 4}: enum
-characterA: std::string
-characterB: std::string
-type: relType
-endTurn: int
+Relation()
+Relation(strJson:std::string)
+~Relation()
+checkConsistency(): bool
+debug(): void
Titles
-titleType{kingdom = 'k', duchy = 'd', county = 'c'}: enum
-titles: std::map<std::string, titleType>
-kingdoms: std::map<std::string, Kingdom>
-duchies: std::map<std::string, Duchy>
-counties: std::map<std::string, County>
+Titles()
+Titles(strJson:std::string)
+~Titles()
+operator[](a:const char*): Title *
+operator[](key:const std::string): Title *
+checkConsistency(): bool
+debug(): void
Levy
-unsigned men: int
-isRaised: bool
-reinforcementRate: float
+Levy()
+Levy(strJson:std::string)
+~Levy()
+checkConsistency(): bool
+debug(): void
TravelOrder
-orderList: std::vector<OneProvinceOrder>
+TravelOrder()
+TravelOrder(strJson:std::string)
+~TravelOrder()
+checkConsistency(): bool
+debug(): void
OneProvinceOrder
+OneProvinceOrder()
+OneProvinceOrder(strJson:std::string)
+~OneProvinceOrder()
+checkConsistency(): bool
+debug(): void
Title
#id: std::string
#name: std::string
#holder: std::string
+~Title()
+debug(): void
+checkConsistency(): bool
+getLiege(): std::string
+getDeJureLiege(): std::string
Kingdom
+Kingdom()
+Kingdom(strJson:std::string)
+~Kingdom()
+debug(): void
+checkConsistency(): bool
+getLiege(): std::string
+getDeJureLiege(): std::string
County
-province: std::string
-liege: std::string
-deJureLiege: std::string
+County()
+County(strJson:std::string)
+~County()
+debug(): void
+checkConsistency(): bool
+getLiege(): std::string
+getDeJureLiege(): std::string
Duchy
-liege: std::string
-deJureLiege: std::string
+Duchy()
+Duchy(strJson:std::string)
+~Duchy()
+debug(): void
+checkConsistency(): bool
+getLiege(): std::string
+getDeJureLiege(): std::string

3 Rendu : Stratégie et Conception

Notre stratégie générale consiste à créer une fenêtre (Window) qui va se rafraîchir régulièrement en utilisant la bibliothèque sfml.
Lors du rafraîchissement de la fenêtre, les divers élément composant la page sont affichés.

Les élément sont affichés dans des cadres (Frames), eux-mêmes compris dans des Canvas. Les canvas sont placés dans la fenêtre. L'idée est que les Frames peuvent se superposer et potentiellement être ouverts ou fermés à n'importe quel moment, tandis que les canvas sont plus statiques. Cela permet surtout de compartimenter l'affichage.

Ces éléments sont ou bien statiques, ou bien liés à des attributs d'objets composant l'état. Dans le second cas, ils décrivent comment afficher ces données.
D'autre part certains éléments sont interactifs, c'est-à-dire qu'ils peuvent provoquer des changements graphiques ou bien modifier l'état de jeu lorsqu'on interagit avec eux. Ainsi cliquer sur une des provinces de la carte ouvre un Frame contenant des cadres de texte (Text) qui décrit l'état de la province en utilisant les informations disponible dans la classe Province des états. Les Frames ainsi créés doivent pouvoir être fermés à tout moment par le joueur à l'aide d'un bouton qui lance le destructeur de l'objet.
Si un élément doit modifier l'état de jeu, une commande est envoyée au moteur de jeu qui se charge des modifications.
L'affichage de la carte du monde se fait à partir de l'objet View de sfml qui permet de n'afficher qu'une partie d'une scène.
Les autres éléments sont des rectangles qui contiennent des types de données variables.

3.1 Stratégie de rendu d'un état

— Window
Description :
Le cœur de la partie rendu. Cette classe utilise un objet RenderWindow de sfml pour créer une fenêtre. Window comprend aussi une fonction renderLoop() qui va rafraîchir l'affichage régulièrement.
Attributs :
— window de type sf : :RenderWindow, la fenêtre affichée
— width de type int la largeur de la fenêtre
— height de type int la hauteur de la fenêtre

— Canvas
Description :
Un cadre lié à une fenêtre.
Attributs :
— window de type Window *, pointeur vers la fenêtre à laquelle appartient le canvas
— width de type int la largeur du canvas
— height de type int la hauteur du canvas
— x de type int la position horizontale du canvas dans la fenêtre
— y de type int la position verticale du canvas dans la fenêtre

— Frame
Description :

Un cadre lié à un canvas. Des éléments peuvent y être placés.
Attributs :
— canvas de type Canvas *, pointeur vers le canvas auquel appartient le Frame
— width de type int la largeur du Frame
— height de type int la hauteur du Frame
— x de type int la position horizontale du Frame dans le Canvas
— y de type int la position verticale du Frame dans le Canvas

— Element
Description :
Un élément généraliste dont tous les autres éléments héritent.
Attributs :
— frame de type Frame *, pointeur vers le Frame dans lequel est placé l'élément
— width de type int la largeur de l'élément
— height de type int la hauteur de l'élément
— x de type int la position horizontale de l'élément dans le Frame
— y de type int la position verticale de l'élément dans le Frame

— interactiveElement
Description :
Un élément iteractif : décrit les éléments qui déclenchent des réactions lors de certaines actions du joueur. Hérite de la classe Element.
Attributs :
— N/A

— IObserver
Description :
Une interface d'observer.
Attributs :
— N/A

— Button
Description :
Un bouton. cliquer dessus déclenche une fonction, si le joueur a la possibilité d'effectuer cette action. Hérite de InteractiveElement.
Attributs :
— available de type bool, décrit si le joueur a la possibilité d'effectuer l'action liée au bouton.
— text de type string, le texte affiché sur le bouton

— ObserverButton
Description :
Observer lié à un bouton, en cas de clic sur le bouton déclenche la fonction associée. Hérite de IObserver.
Attributs :
— N/A

— ViewMap
Description :
Permet de visualiser une partie plus ou moins grande de la carte du monde. Hérite de InteractiveE-

lement.
Attributs :
— zoom de type int, décrit le niveau de zoom par rapport à la carte complète
— mapX de type int, décrit la position horizontale de ViewMap sur la carte
— mapY de type int, décrit la position verticale de ViewMap sur la carte

— ObserverMap
Description :
Observer lié à ViewMap, permet d'observer les commandes liées au zoom et au déplacement de la vue dans la carte. Hérite de IObserver.
Attributs :
— N/A

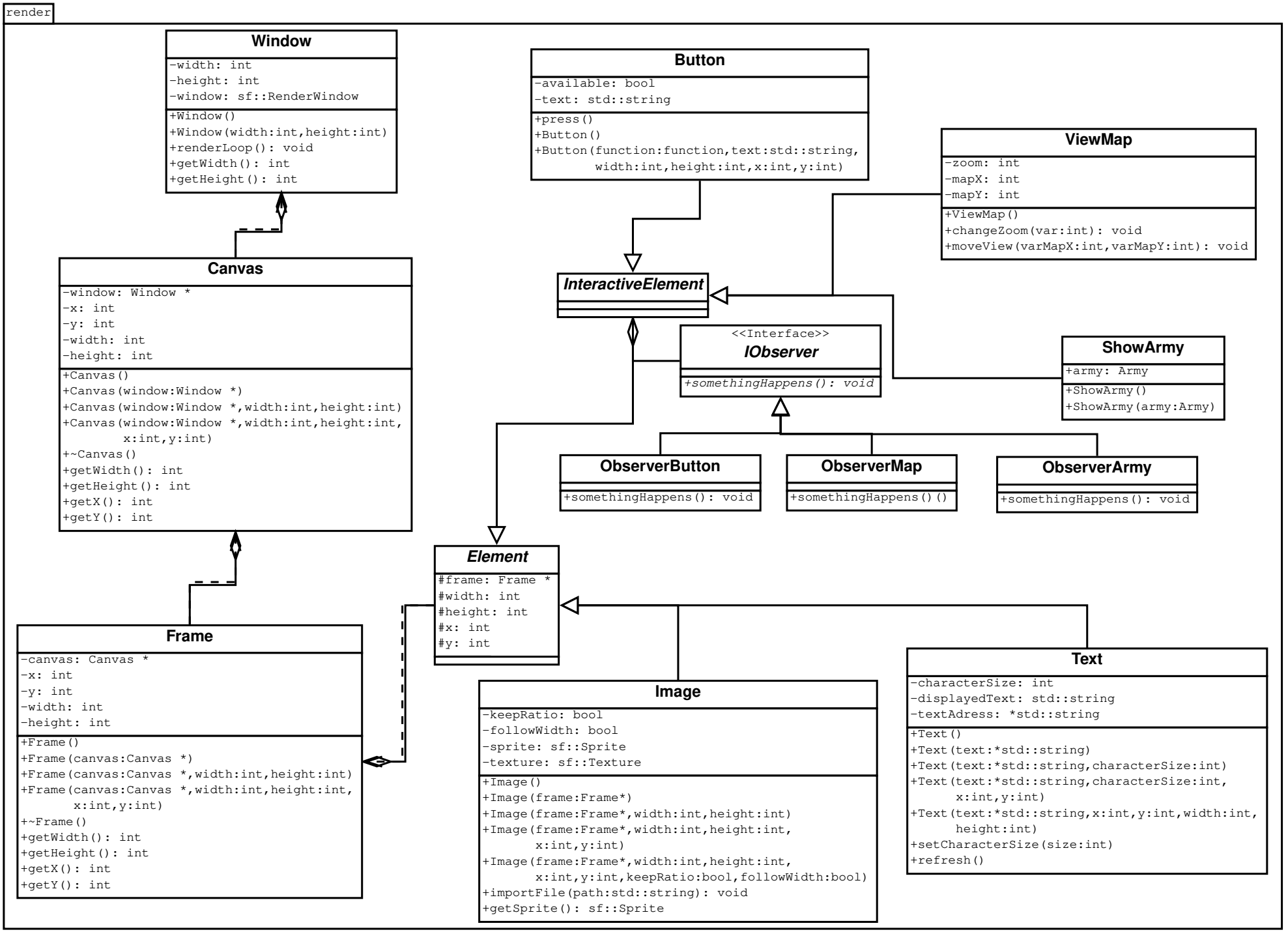
— ShowArmy
Description :
Permet de visualiser une armée sur la carte et d'interagir avec elle. Hérite de InteractiveElement.
Attributs :
— army de type Army, l'armée à laquelle est liée l'objet ShowArmy.

— ObserverArmy
Description :
Observer lié à ShowArmy, permet d'observer les commandes liées à un objet ShowArmy. Hérite de IObserver.
Attributs :
— N/A

— Text
Description :
Cadre contenant du texte. Hérite de Element.
Attributs :
— characterSize de type int, définit la taille du texte affiché dans l'objet Text.
— displayedText de type std : :string, décrit le texte affiché
— textAdress de type *std : :string, l'adresse du texte à afficher

— Image
Description :
Element permettant d'afficher une image fixe. Hérite d'Element.
Attributs :
— keepRatio de type bool, décrit si l'image doit être adaptée aux dimensions de l'objet ou si elle doit garder son ratio
— followWidth de type bool, dans le cas où keepRatio est vrai, cet attribut permet de décider si l'image doit être adaptée selon la largeur ou la hauteur de l'objet
— sprite de type sf : :sprite, support utilisé pour afficher l'image
— texture de type sf : :texture, variable dans laquelle on charge l'image à afficher

3.2 Conception logiciel


render
Window
-width: int
-height: int
-window: sf::RenderWindow
+Window()
+Window(width:int,height:int)
+renderLoop(): void
+getWidth(): int
+getHeight(): int
Button
-available: bool
-text: std::string
+press()
+Button()
+Button(function:function,text:std::string,
width:int,height:int,x:int,y:int)
ViewMap
-zoom: int
-mapX: int
-mapY: int
+ViewMap()
+changeZoom(var:int): void
+moveView(varMapX:int,varMapY:int): void
Canvas
-window: Window *
-x: int
-y: int
-width: int
-height: int
+Canvas()
+Canvas(window:Window *)
+Canvas(window:Window *,width:int,height:int)
+Canvas(window:Window *,width:int,height:int,
x:int,y:int)
+~Canvas()
+getWidth(): int
+getHeight(): int
+getX(): int
+getY(): int
InteractiveElement
<<Interface>>
IObserver
+somethingHappens(): void
ShowArmy
+army: Army
+ShowArmy()
+ShowArmy(army:Army)
ObserverButton
+somethingHappens(): void
ObserverMap
+somethingHappens()()
ObserverArmy
+somethingHappens(): void
Element
#frame: Frame *
#width: int
#height: int
#x: int
#y: int
Frame
-canvas: Canvas *
-x: int
-y: int
-width: int
-height: int
+Frame()
+Frame(canvas:Canvas *)
+Frame(canvas:Canvas *,width:int,height:int)
+Frame(canvas:Canvas *,width:int,height:int,
x:int,y:int)
+~Frame()
+getWidth(): int
+getHeight(): int
+getX(): int
+getY(): int
Image
-keepRatio: bool
-followWidth: bool
-sprite: sf::Sprite
-texture: sf::Texture
+Image()
+Image(frame:Frame*)
+Image(frame:Frame*,width:int,height:int)
+Image(frame:Frame*,width:int,height:int,
x:int,y:int)
+Image(frame:Frame*,width:int,height:int,
x:int,y:int,keepRatio:bool,followWidth:bool)
+importFile(path:std::string): void
+getSprite(): sf::Sprite
Text
-characterSize: int
-displayedText: std::string
-textAdress: *std::string
+Text()
+Text(text:*std::string)
+Text(text:*std::string,characterSize:int)
+Text(text:*std::string,characterSize:int,
x:int,y:int)
+Text(text:*std::string,x:int,y:int,width:int,
height:int)
+setCharacterSize(size:int)
+refresh()

4 Règles de changement d’états et moteur de jeu

4.1 Règles

4.2 Conception logiciel

5 Intelligence Artificielle

5.1 Stratégies

5.2 Conception logiciel

6 Modularisation

6.1 Organisation des modules

6.2 Conception logiciel