Projet Logiciel Transversal

Projet SLAY

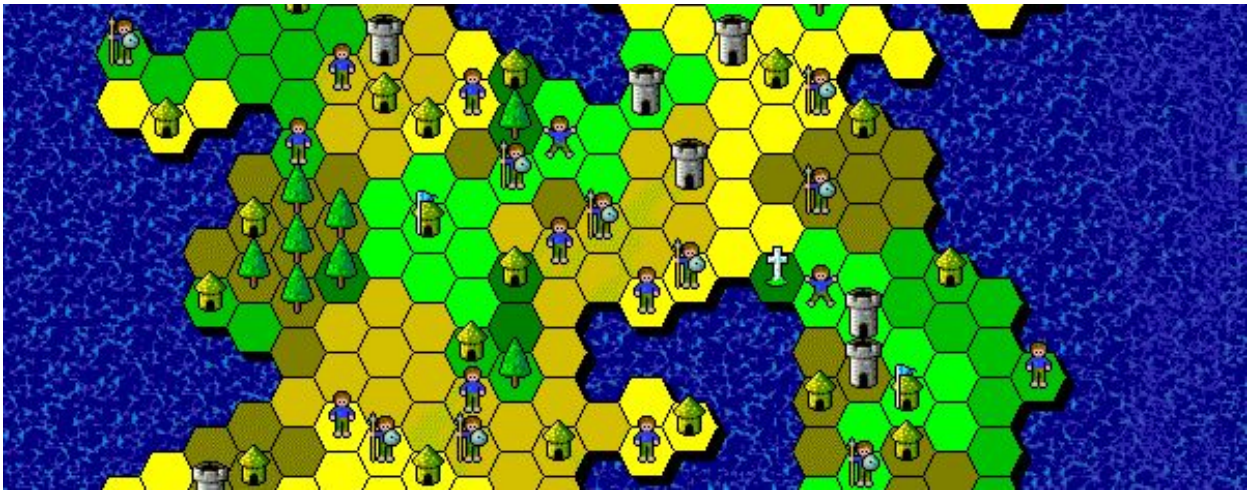


Figure 1 — Exemple du jeu *Slay*

Badisse BOUABDALLAH

Nicolas LARUE

Hicham SDIRI

Kaan UYGUN

Table des matières

1 Présentation Générale

1.1 Introduction

Notre jeu est basé sur le jeu de stratégie tour par tour *Slay*, créé par Sean O'Connor et sorti pour Microsoft Windows en mars 1995.

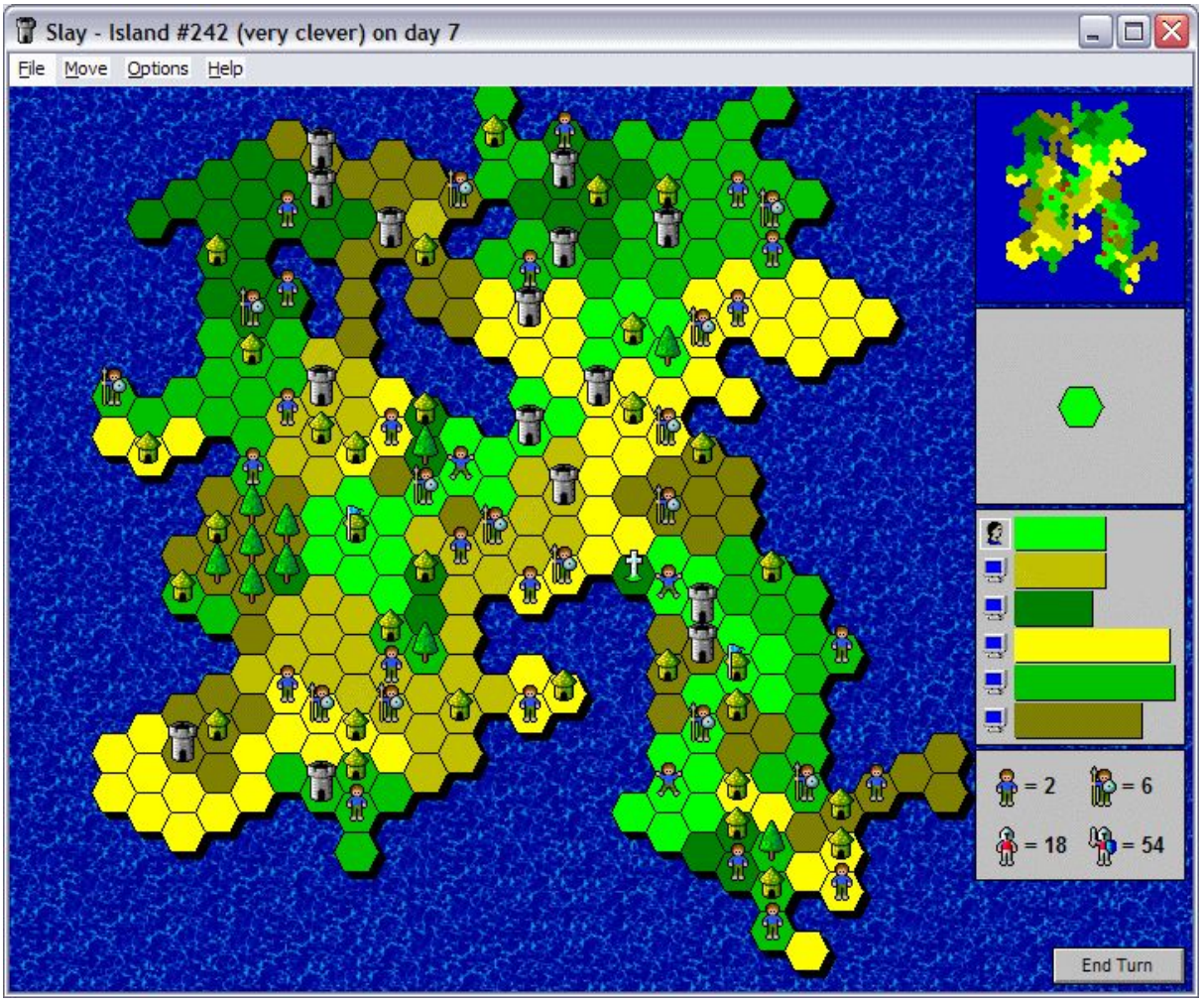


Figure 2 — Interface graphique du jeu Slay

1.2 Objectif du jeu

En tant que joueur, votre objectif est de conquérir toute l'île en achetant des hommes et en les utilisant pour capturer les territoires de vos ennemis.

1.3 Règles du jeu

Slay se joue jusqu'à six joueurs sur un plateau hexagonal. Les joueurs reçoivent d'abord un ou plusieurs petits territoires, qu'ils doivent développer en déplaçant des soldats. Un territoire est formé d'au moins deux cellules. Plus vous avez de cellules sur votre territoire, plus vous gagnez d'argent. Il existe différents niveaux de soldats qui coûtent différents prix et consomment différentes quantités de ressources. Vous pouvez également placer des tours sur la carte afin de protéger votre territoire des attaques adverses.

1.4 Ressources

Figure 3 — Sprites des différents personnages et objets

Figure 4 — Textures et couleurs des cellules

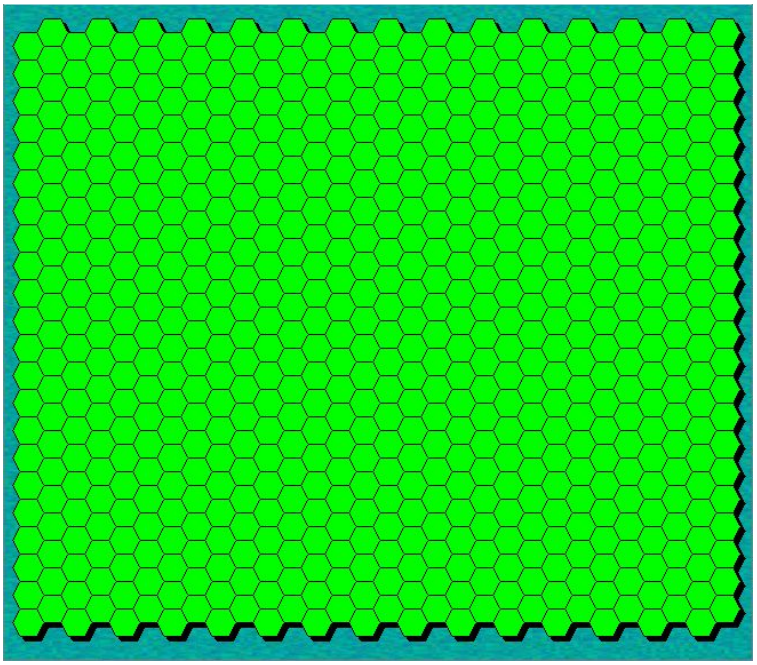


Figure 5 — Grille de jeu maximale

Figure 6 — Police d'écriture (Balsamiq Sans)

Remarque : Pour la suite du projet, il est fortement probable que l'on change certaines de ces ressources.

2 Description et conception des états

2.1 Description des états

Un état du jeu est caractérisé par les joueurs, la grille de jeu et les territoires. Généralement, l'état du jeu est déterminé premièrement si la partie est toujours en cours ou non. De plus, le nombre de joueurs et le joueur dont c'est le tour font aussi partie d'un état de jeu.

2.1.1 Grille de jeu

La carte est formée par une grille de jeu composée de cellules hexagonales. La dimension de cette grille est fixée au démarrage du niveau. Il y a deux types de cellules différentes :

- **Cellules Non Accessibles** : Les cellules dites "non accessibles" sont des cellules comprises dans la grille de jeu mais sur lesquelles il est impossible de jouer. Elles ne comportent donc aucune entité de jeu. Visuellement, ces cellules sont des cellules de type "mer". Le choix de la texture de ces cellules est purement esthétique et n'a pas d'influence sur l'évolution du jeu.
- **Cellules Accessibles :** Les cellules accessibles (visuellement représentées par des cellules de type "terre") sont des cellules qui contiennent des entités, qui se différencient les unes des autres par un identifiant unique. Toutes les entités possèdent les propriétés suivantes :
    - des points d'attaque
    - des points de défense
    - les revenus générés (positifs ou négatifs) par tour

  Ces entités peuvent être les suivantes :
    - Entité de type "arbre" : cellule qui comporte un arbre.
      Il existe deux types d'arbres :
        - Les palmiers
        - Les pins
    - Entité de type "édifice" : cellule qui comporte un édifice.
      Il en existe trois types :
        - Les capitales
        - Les châteaux

        - Les âmes
    - Entité de type “soldat” : cellule qui comporte un soldat.
      Il en existe quatre types :
        - Les paysans
        - Les lanciers
        - Les chevaliers
        - Les barons
    - Entité de type “vide” : cellule qui ne comporte aucune autre entité.

Outre l’entité qu’elle contient, une cellule est aussi liée à un joueur et potentiellement à un territoire. Dans le cas où la cellule est neutre, c’est-à-dire qu’elle n’appartient à aucun joueur, nous considérons qu’elle appartient toutefois à un joueur “neutre” qui sera défini en plus des joueurs.

2.1.2 Territoires

Un territoire est un ensemble de cellules adjacentes — au moins deux — appartenant au même joueur. Il est alors possible qu’un joueur dispose de plusieurs territoires. Pour chaque territoire, une cellule est réservée pour comporter une capitale. Visuellement, tous les territoires d’un même joueur sont de même couleur.

De plus, chaque territoire comporte plusieurs propriétés :

- L’argent gardé dans la capitale
- Les revenus positifs générés par ce territoire par tour ; autrement dit, c’est la somme des revenus positifs de toutes les cellules du territoire.
- Les revenus globaux générés par ce territoire par tour ; autrement dit, c’est la somme de tous les revenus (positifs et négatifs) de toutes les cellules du territoire.

Les revenus positifs et les revenus globaux sont distingués dans le but d’afficher ces deux propriétés sur l’interface de jeu.

2.1.3 Joueurs

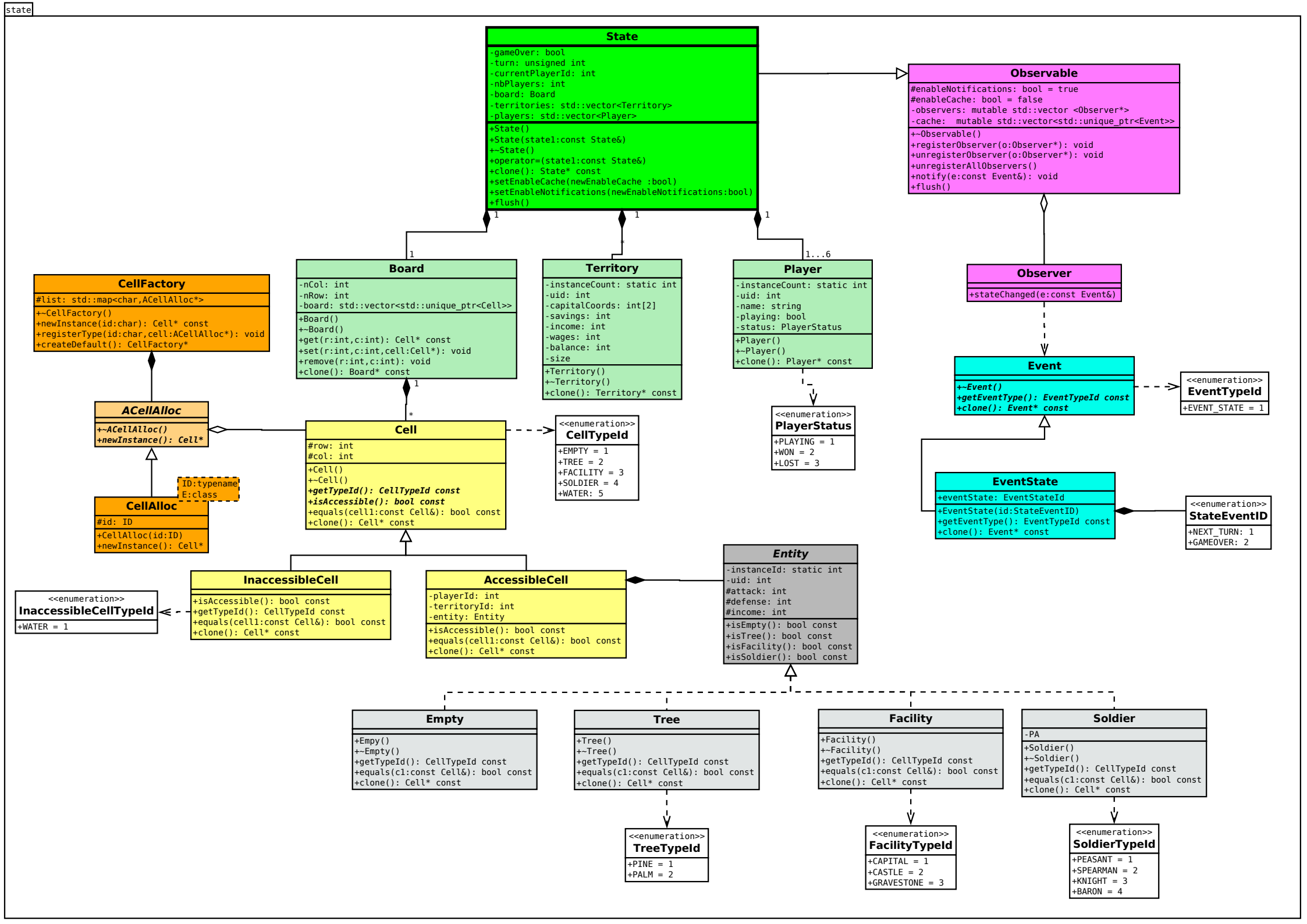
Chaque joueur dispose d’un nom et se différencie des autres joueurs par un identifiant unique et possède une propriété “statut” qui peut prendre les valeurs suivantes :

- Statut “en jeu” : le joueur est encore en lice
- Statut “gagné” : le joueur a gagné la partie

- Statut “perdu” : le joueur n’a pas gagné la partie

2.2 Conception Logiciel

Le diagramme des classes pour les états est présenté en Figure 7.


state
State
-gameOver: bool
-turn: unsigned int
-currentPlayerId: int
-nbPlayers: int
-board: Board
-territories: std::vector<Territory>
-players: std::vector<Player>
+State()
+State(state1:const State&)
+~State()
+operator=(state1:const State&)
+clone(): State* const
+setEnableCache(newEnableCache :bool)
+setEnableNotifications(newEnableNotifications:bool)
+flush()
Observable
#enableNotifications: bool = true
#enableCache: bool = false
-observers: mutable std::vector <Observer*>
-cache: mutable std::vector<std::unique_ptr<Event>>
+~Observable()
+registerObserver(o:Observer*): void
+unregisterObserver(o:Observer*): void
+unregisterAllObservers()
+notify(e:const Event&): void
+flush()
Observer
+stateChanged(e:const Event&)
Board
-nCol: int
-nRow: int
-board: std::vector<std::unique_ptr<Cell>>
+Board()
+~Board()
+get(r:int,c:int): Cell* const
+set(r:int,c:int,cell:Cell*): void
+remove(r:int,c:int): void
+clone(): Board* const
Territory
-instanceCount: static int
-uid: int
-capitalCoords: int[2]
-savings: int
-income: int
-wages: int
-balance: int
-size
+Territory()
+~Territory()
+clone(): Territory* const
Player
-instanceCount: static int
-uid: int
-name: string
-playing: bool
-status: PlayerStatus
+Player()
+~Player()
+clone(): Player* const
CellFactory
#list: std::map<char,ACellAlloc*>
+~CellFactory()
+newInstance(id:char): Cell* const
+registerType(id:char,cell:ACellAlloc*): void
+createDefault(): CellFactory*
ACellAlloc
+~ACellAlloc()
+newInstance(): Cell*
CellAlloc
#id: ID
+CellAlloc(id:ID)
+newInstance(): Cell*
ID:typename
E:class
Cell
#row: int
#col: int
+Cell()
+~Cell()
+getTypeId(): CellTypeId const
+isAccessible(): bool const
+equals(cell1:const Cell&): bool const
+clone(): Cell* const
<<enumeration>>
CellTypeId
+EMPTY = 1
+TREE = 2
+FACILITY = 3
+SOLDIER = 4
+WATER: 5
<<enumeration>>
PlayerStatus
+PLAYING = 1
+WON = 2
+LOST = 3
Event
+~Event()
+getEventType(): EventTypeId const
+clone(): Event* const
<<enumeration>>
EventTypeId
+EVENT_STATE = 1
EventState
+eventState: EventStateId
+EventState(id:StateEventID)
+getEventType(): EventTypeId const
+clone(): Event* const
<<enumeration>>
StateEventID
+NEXT_TURN: 1
+GAMEOVER: 2
InaccessibleCell
+isAccessible(): bool const
+getTypeId(): CellTypeId const
+equals(cell1:const Cell&): bool const
+clone(): Cell* const
<<enumeration>>
InaccessibleCellTypeId
+WATER = 1
AccessibleCell
-playerId: int
-territoryId: int
-entity: Entity
+isAccessible(): bool const
+equals(cell1:const Cell&): bool const
+clone(): Cell* const
Entity
-instanceId: static int
-uid: int
#attack: int
#defense: int
#income: int
+isEmpty(): bool const
+isTree(): bool const
+isFacility(): bool const
+isSoldier(): bool const
Empty
+Empy()
+~Empty()
+getTypeId(): CellTypeId const
+equals(c1:const Cell&): bool const
+clone(): Cell* const
Tree
+Tree()
+~Tree()
+getTypeId(): CellTypeId const
+equals(c1:const Cell&): bool const
+clone(): Cell* const
Facility
+Facility()
+~Facility()
+getTypeId(): CellTypeId const
+equals(c1:const Cell&): bool const
+clone(): Cell* const
Soldier
-PA
+Soldier()
+~Soldier()
+getTypeId(): CellTypeId const
+equals(c1:const Cell&): bool const
+clone(): Cell* const
<<enumeration>>
TreeTypeId
+PINE = 1
+PALM = 2
<<enumeration>>
FacilityTypeId
+CAPITAL = 1
+CASTLE = 2
+GRAVESTONE = 3
<<enumeration>>
SoldierTypeId
+PEASANT = 1
+SPEARMAN = 2
+KNIGHT = 3
+BARON = 4

3 Rendu : Stratégie et Conception

3.1 Stratégie de rendu d'un état

Pour définir une stratégie de rendu, il nous faut d'abord définir les éléments qui fixent un état précis. Trois éléments permettent sont nécessaires pour effectuer le rendu d'un état : la **carte de jeu** composée de cellules hexagonales, les **entités** présentes ou non sur les cellules, et enfin les **informations** autour de l'état jeu (argent des joueurs, nombre de cellules possédées, etc.). Pour le rendu de cet état, nous avons fait le choix d'un rendu par tuiles à l'aide de la bibliothèque SFML. Notre affichage est donc composé de trois surfaces :

- Une couche contenant les cellules hexagonales, qui sont de couleurs différentes selon le territoire (cf. Figure 4)
- Une couche pour les entités (soldats, arbres et bâtiments) (cf. Figure 3)
- Une couche pour les informations de l'état du jeu

Les informations de l'état du jeu qui seront affichées à l'écran seront une carte miniature pour avoir une vue d'ensemble sur la partie, l'argent détenu par les joueurs, les achats possibles pour chaque territoire et une barre d'avancement général présentant combien de cellules chaque joueur possède, en pourcentage.

La carte est construite à partir d'un fichier texte contenant l'ID associé à chaque cellule. Ainsi, pour définir si une cellule est terrestre (donc accessible) ou maritime (donc inaccessible), il suffit d'assigner un ID spécifique à cette cellule dans le fichier texte. Il en est de même concernant les entités présentes sur la carte.

À chaque nouvel événement, les informations du jeu sont changées et l'affichage doit alors être mis à jour.

3.2 Conception Logiciel

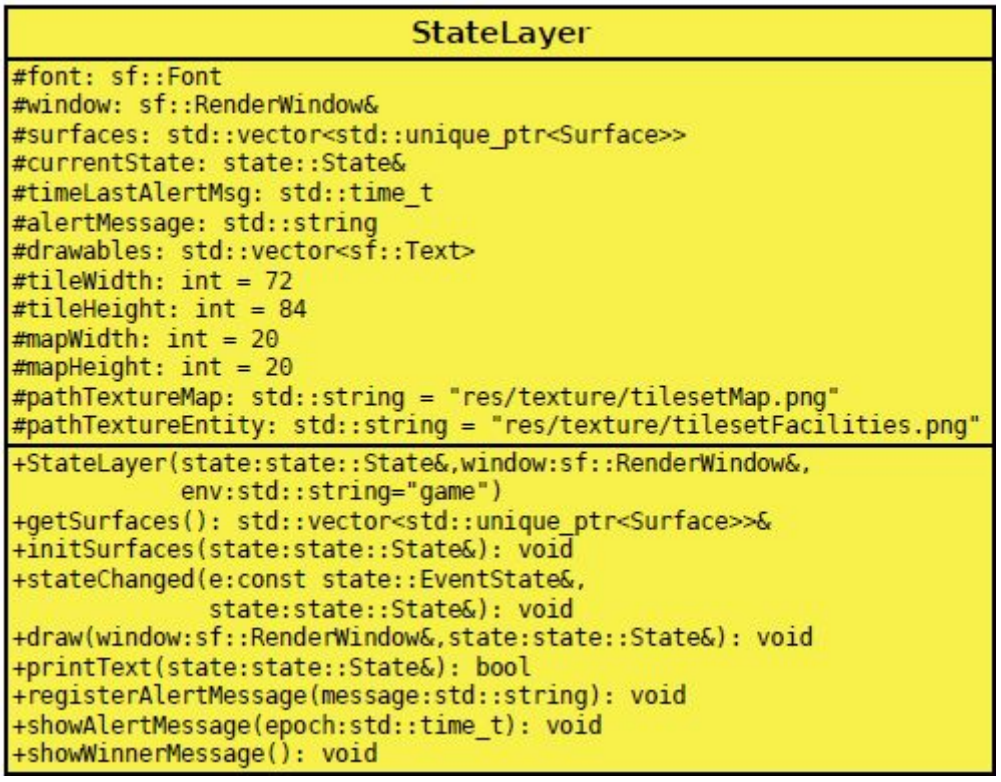


Figure 9 — Classe StateLayer

Cette classe permet de faire l'intermédiaire entre le *state* et le *render*. Elle contient les paramètres d'affichages de la fenêtre, les paramètres et le chargement des tuiles et la gestion générale de l'affichage de la classe State. Ses méthodes permettent d'initialiser les couches et d'afficher des messages.

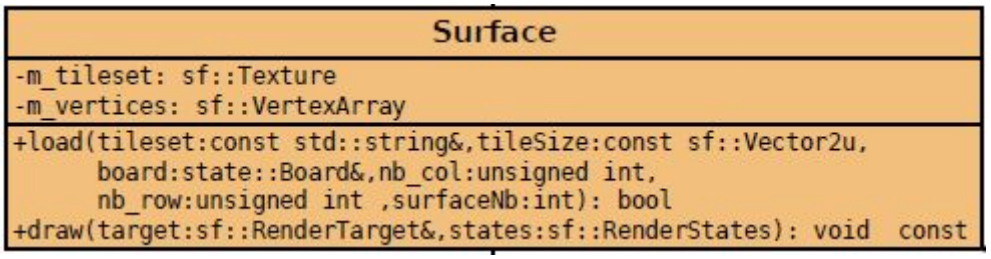
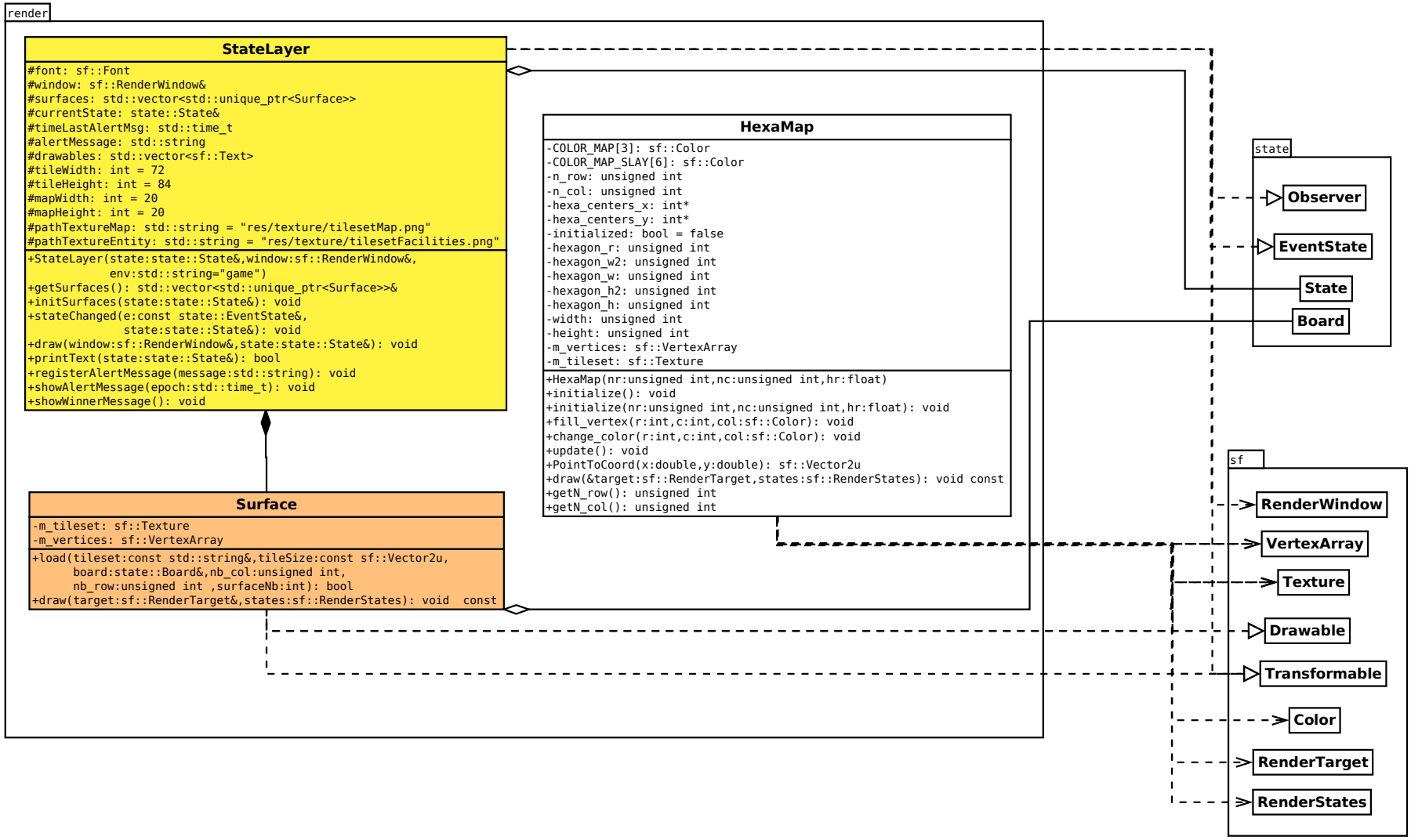


Figure 10 — Classe Surface

La classe Surface permet de gérer les différentes couches pour l'affichage de la carte. Elle gère aussi le chargement des ressources visuelles.

Le diagramme des classes pour le rendu d'un état est présenté en Figure 11.


render
StateLayer
#font: sf::Font
#window: sf::RenderWindow&
#surfaces: std::vector<std::unique_ptr<Surface>>
#currentState: state::State&
#timeLastAlertMsg: std::time_t
#alertMessage: std::string
#drawables: std::vector<sf::Text>
#tileWidth: int = 72
#tileHeight: int = 84
#mapWidth: int = 20
#mapHeight: int = 20
#pathTextureMap: std::string = "res/texture/tilesetMap.png"
#pathTextureEntity: std::string = "res/texture/tilesetFacilities.png"
+StateLayer(state:state::State&,window:sf::RenderWindow&,
env:std::string="game")
+getSurfaces(): std::vector<std::unique_ptr<Surface>>&
+initSurfaces(state:state::State&): void
+stateChanged(e:const state::EventState&,
state:state::State&): void
+draw(window:sf::RenderWindow&,state:state::State&): void
+printText(state:state::State&): bool
+registerAlertMessage(message:std::string): void
+showAlertMessage(epoch:std::time_t): void
+showWinnerMessage(): void
HexaMap
-COLOR_MAP[3]: sf::Color
-COLOR_MAP_SLAY[6]: sf::Color
-n_row: unsigned int
-n_col: unsigned int
-hexa_centers_x: int*
-hexa_centers_y: int*
-initialized: bool = false
-hexagon_r: unsigned int
-hexagon_w2: unsigned int
-hexagon_w: unsigned int
-hexagon_h2: unsigned int
-hexagon_h: unsigned int
-width: unsigned int
-height: unsigned int
-m_vertices: sf::VertexArray
-m_tileset: sf::Texture
+HexaMap(nr:unsigned int,nc:unsigned int,hr:float)
+initialize(): void
+initialize(nr:unsigned int,nc:unsigned int,hr:float): void
+fill_vertex(r:int,c:int,col:sf::Color): void
+change_color(r:int,c:int,col:sf::Color): void
+update(): void
+PointToCoord(x:double,y:double): sf::Vector2u
+draw(&target:sf::RenderTarget,states:sf::RenderStates): void const
+getN_row(): unsigned int
+getN_col(): unsigned int
Surface
-m_tileset: sf::Texture
-m_vertices: sf::VertexArray
+load(tileset:const std::string&,tileSize:const sf::Vector2u,
board:state::Board&,nb_col:unsigned int,
nb_row:unsigned int ,surfaceNb:int): bool
+draw(target:sf::RenderTarget&,states:sf::RenderStates): void const
state
Observer
EventState
State
Board
sf
RenderWindow
VertexArray
Texture
Drawable
Transformable
Color
RenderTarget
RenderStates

4 Règles de changements d'états et moteur de jeu

4.1 Horloge globale

Un changement d'état est réalisé dès lors qu'un joueur réalise une action. Toutes les actions sont donc soumises à la même temporalité.

4.2 Changements extérieurs

Les changements extérieurs sont des changements de l'état du jeu suite à des actions d'un joueur. Il existe plusieurs catégories de changements extérieurs dans notre jeu :

- le déplacement des soldats
- l'achat ou l'amélioration d'entités

Le déplacement d'un soldat se fait à la souris, en sélectionnant un soldat d'un territoire et en cliquant sur la case sur laquelle on veut l'amener. Un déplacement en dehors du territoire permet de conquérir de nouvelles cases. Un soldat peut être déplacé dans tout le territoire auquel il appartient mais lorsqu'il est déplacé en dehors du territoire, il ne peut conquérir qu'une seule case. Par ailleurs, les arbres présents sur la carte peuvent être abattus par n'importe quel soldat. Toutefois, dès lors qu'un soldat a coupé un arbre, il ne pourra plus être déplacé lors de ce tour.

L'achat de nouvelles entités se fait via la boutique. Suite à l'achat, il faut placer l'entité au sein du territoire concerné. Il est de plus possible d'améliorer le niveau d'un soldat en en superposant deux.

Une fois toutes ses actions réalisées, le joueur choisit de finir son tour en cliquant sur le bouton associé.

4.3 Changements autonomes

Les changements autonomes sont appliqués après chaque fin de tour. Dans notre cas, les changements autonomes sont les suivants :

- Affichage du “Game over” pour un joueur qui a perdu
- Affichage de l’écran de victoire pour le joueur gagnant
- Recalcul des statistiques (argent, entre autres) de chaque joueur

4.4 Conception Logiciel

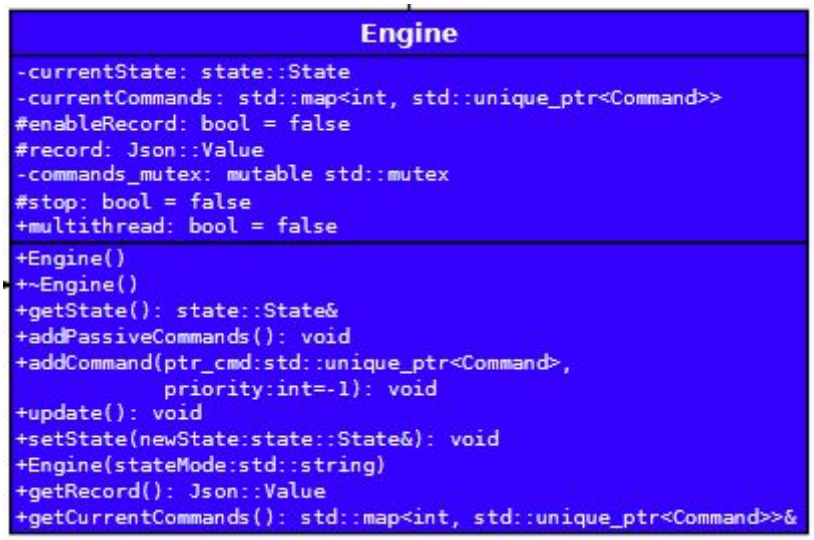


Figure 12 — Classe Engine

La classe Engine permet de gérer tout le moteur de jeu et elle permet de gérer l’échange avec la classe State.

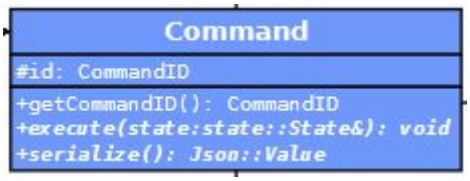
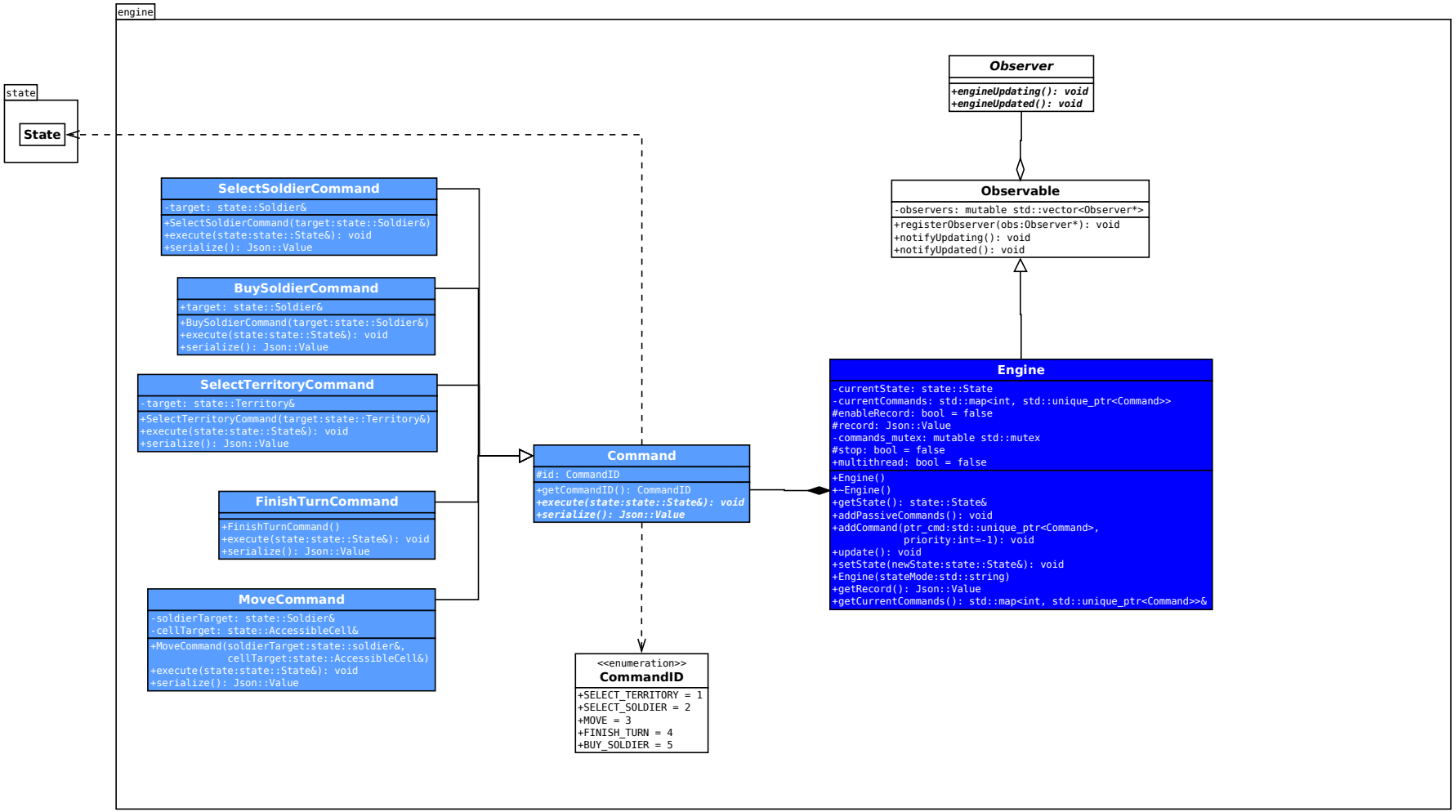


Figure 13 — Classe Command

Toutes les commandes du jeu (Sélection de soldat, Achat d’entité, Sélection de territoire, Fin de tour et Mouvement de soldat) héritent de la classe Command. Cette dernière contient un attribut *CommandID* qui diffère selon la commande effectuée.

Le diagramme des classes pour le moteur du jeu est présenté en Figure 14.


engine
state
State
Observer
+engineUpdating(): void
+engineUpdated(): void
Observable
-observers: mutable std::vector<Observer*>
+registerObserver(obs:Observer*): void
+notifyUpdating(): void
+notifyUpdated(): void
SelectSoldierCommand
-target: state::Soldier&
+SelectSoldierCommand(target:state::Soldier&)
+execute(state:state::State&): void
+serialize(): Json::Value
BuySoldierCommand
+target: state::Soldier&
+BuySoldierCommand(target:state::Soldier&)
+execute(state:state::State&): void
+serialize(): Json::Value
SelectTerritoryCommand
-target: state::Territory&
+SelectTerritoryCommand(target:state::Territory&)
+execute(state:state::State&): void
+serialize(): Json::Value
FinishTurnCommand
+FinishTurnCommand()
+execute(state:state::State&): void
+serialize(): Json::Value
MoveCommand
-soldierTarget: state::Soldier&
-cellTarget: state::AccessibleCell&
+MoveCommand(soldierTarget:state::soldier&,
cellTarget:state::AccessibleCell&)
+execute(state:state::State&): void
+serialize(): Json::Value
Command
#id: CommandID
+getCommandID(): CommandID
+execute(state:state::State&): void
+serialize(): Json::Value
Engine
-currentState: state::State
-currentCommands: std::map<int, std::unique_ptr<Command>>
#enableRecord: bool = false
#record: Json::Value
-commands_mutex: mutable std::mutex
#stop: bool = false
+multithread: bool = false
+Engine()
+~Engine()
+getState(): state::State&
+addPassiveCommands(): void
+addCommand(ptr_cmd:std::unique_ptr<Command>,
priority:int=-1): void
+update(): void
+setState(newState:state::State&): void
+Engine(stateMode:std::string)
+getRecord(): Json::Value
+getCurrentCommands(): std::map<int, std::unique_ptr<Command>>&
<<enumeration>>
CommandID
+SELECT_TERRITORY = 1
+SELECT_SOLDIER = 2
+MOVE = 3
+FINISH_TURN = 4
+BUY_SOLDIER = 5