Dossier de conception

Game Foot+ 2D

Auteurs	Approbateurs	Validation
Nouaïm Souiki		
Rédigé le : 04/11/2014	Approuvé le :	Validé le :

Diffusion	Externe
À :	Aznam Yacoub Amine Hamri
Copies à :	Amine Rhazi Aimad Goussa Nouaïm Souiki Imane Srhir Amal Weslati Naoufal Sabihi

Documents de réferences

Libellé	Document

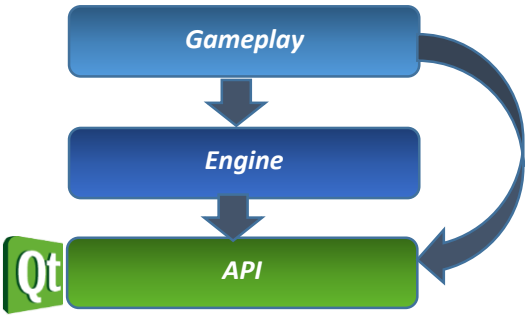
Historique

N° Version	Auteurs	Approbateurs	Date	Historique des évolutions
1.0	Nouaïm Souiki	Toute l'équipe	04/11/2014	Création du document. Rajout et rédaction des sections **1**-**2**-**3**
1.1	Nouaïm Souiki	Toute l'équipe	07/11/2014	Complétion de la rédaction.
1.2	Nouaïm Souiki	Toute l'équipe	09/11/2014	Rajout des classes `QTimer` et `PlayingGameState`

1. Introduction

Architecture du système

La conception du jeu *Game Foot+ 2D* se base sur l'architecture à 3 couches suivante :



Les flèches représentent des relations de dépendances (donc d'obsolescences : la modification d'une couche de base risque d'entraîner la modification de la couche dépendante).

La couche *API* nous est fournie par le *Framework Qt*, il nous reste donc à implémenter les deux couches supérieures, *Engine* et *Gameplay*.

Ligne à suivre

Notre conception se base autant que possible sur :

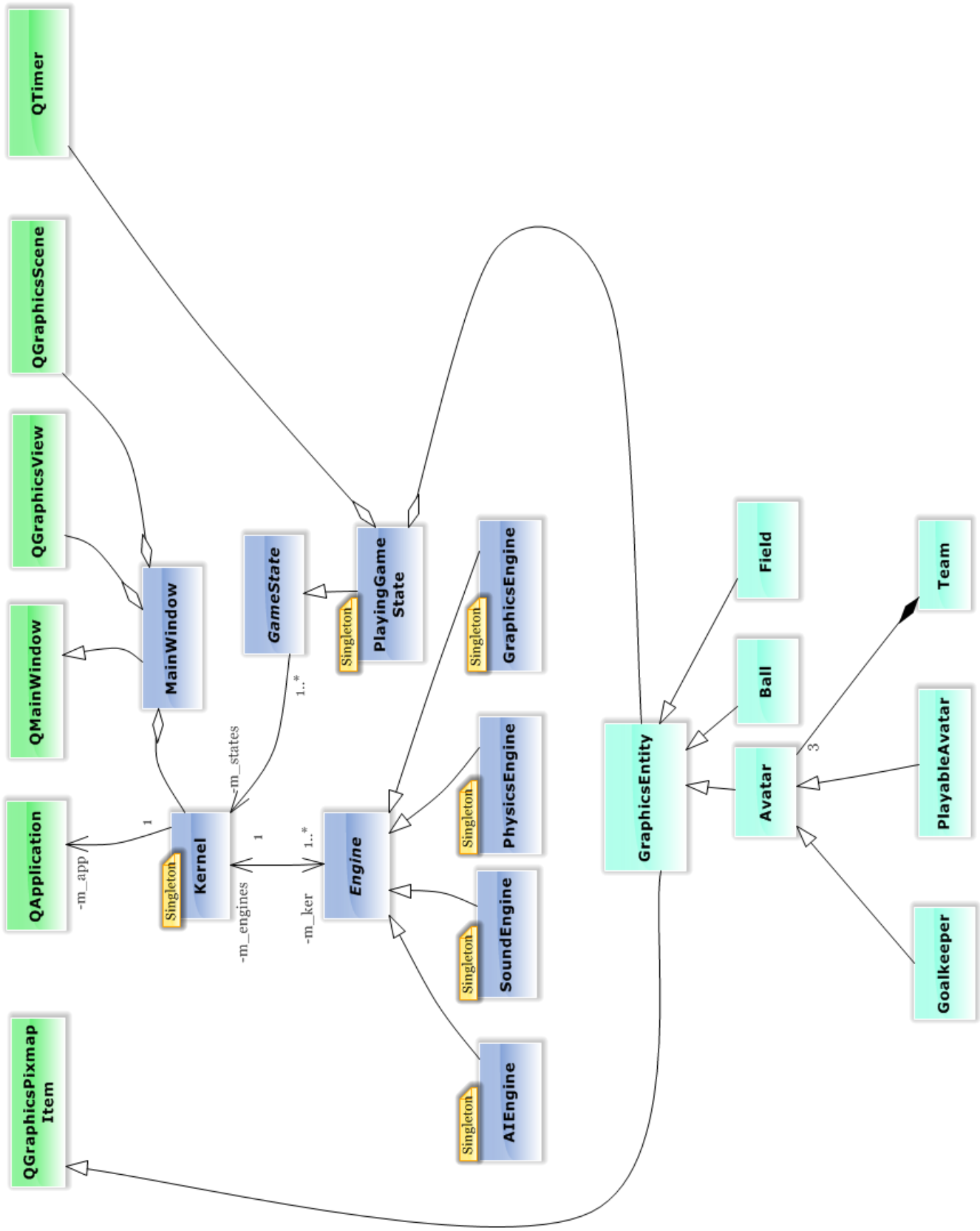
- L'utilisation optimale de la couche *API*, afin de ne pas « réinventer la roue » pour les procédures déjà fournies par cette couche primaire,
- Le respect des principes de conception orientée objets (en particuliers les cinq principes *SOLID*),
- Une utilisation –aussi fréquente que possible- des patrons de conception, afin de produire une application de qualité, robuste, et assurer à la fois la *clarté* et la *maintenabilité* du code source.

Diagramme global des classes participantes

Dans cette section, nous donnons le diagramme des classes qui sert de base à notre conception.

Ce diagramme a été allégé des méthodes et attributs, ainsi que des patrons utilisés, pour ne laisser que le squelette global des classes et de leurs interactions.

N.B. : le lecteur trouvera les détails omis dans les sections suivantes de ce document, qui offrent une analyse détaillée de chacune des classes présentées ici.


QTimer
QGraphicsScene
QGraphicsView
QMainWindow
QApplication
QGraphicsPixmapItem
MainWindow
Kernel
Singleton
-m_app
1
-m_engines
1
-m_states
1..*
GameState
PlayingGameState
Singleton
Engine
1..*
-m_ker
GraphicsEngine
Singleton
PhysicsEngine
Singleton
SoundEngine
Singleton
AIEngine
Singleton
GraphicsEntity
Field
Ball
Avatar
3
Team
PlayableAvatar
Goalkeeper

2. Couche *Engine*

2.1 Noyau du système

Les tâches que le noyau effectue sont :

1. Echange de données (coordination, synchronisation) entre les différentes composantes (moteurs et gestionnaires) du système,
2. Gérer les entrées utilisateurs : capturer les évènements produits par le gamer, et les aiguiller vers les composantes adéquates du système,

Pour ne pas rester dans la théorie, prenons quelques exemples pratiques pour chacune de ces tâches :

- Le *Physics Engine* (via le *Collision Manager*) traite un évènement *« but »*, il doit le déléguer au *Sound Engine* afin de produire le son « GOOOOAAAAAL » attendu ; le noyau se charge de la communication et de la synchronisation de cet évènement entre les deux moteurs,
- Pendant que son avatar contrôle la balle, le gamer appuie sur la touche **T** : l'avatar va donc tirer la balle, qui va suivre une certaine trajectoire vers les cages adverses ; le noyau se charge de communiquer cet évènement au *Physics Engine* (pour le calcul de la trajectoire), et au *Graphics Engine* (pour l'animation du tir de la balle),
- Pendant une partie de jeu, l'utilisateur appuie sur la touche **Escape** ; le noyau, via l'état courant, active l'état *PauseGame* (cf.

Objectif

Nous voulons concevoir la classe `Kernel`, implémentant le noyau, de façon à ce qu'elle puisse réaliser les tâches citées ci-dessus, et informer ainsi les différentes composantes du système de l'état général du programme. Nous voulons réaliser cette conception selon la ligne de conduite énoncée précédemment (cf.).

Conception

Utilisation du patron Observateur

Nous souhaitons réaliser cet objectif en appliquant le patron de conception *Observer*, ce qui constitue une stratégie pertinente puisque le problème résolu par ce patron de conception est :

« Créer un lien entre un objet source et plusieurs objets cibles permettant de notifier les objets cibles lorsque l'état de l'objet source change. De plus, il faut pouvoir dynamiquement lier à (ou délier de) l'objet source autant d'objets cibles que nous le voulons. »

Or dans notre cas, l'objet `Observer` (l'objet cible) est un moteur du jeu et l'objet `Subject` (l'objet source) est le noyau, puisque notre problème est d'informer le moteur (cible) des évènements reçus par le noyau (source).

La classe `Observer` du patron de conception correspond à une abstraction de la classe `Engine` dont cette dernière hérite. C'est elle qui doit contenir la méthode par laquelle un moteur est informé des changements perçus dans le noyau.

La classe `Subject` du patron de conception correspond à une abstraction de la classe `Kernel` dont cette dernière hérite. Dans notre cas, cette classe contient les opérations abstraites *`add(Observer)`* `et remove (Observer)`, ainsi que l'opération de notification à tous les observateurs (i.e. à tous les moteurs), *`notify()`*.

Utilisation du patron Singleton

Le noyau (comme tous les moteurs et gestionnaires du jeu) ne doit pas exister en plusieurs exemplaires durant l'exécution du système. Pour cela il nous faut empêcher qu'il en soit instancié plus d'un objet, et assurer ainsi son unicité durant tout le programme. Une solution a été de l'implémenter en tant que classe *Singleton*.

Voici un premier aperçu de la classe `Kernel`, qui sera bien entendu enrichie au fur et à mesure de la conception.

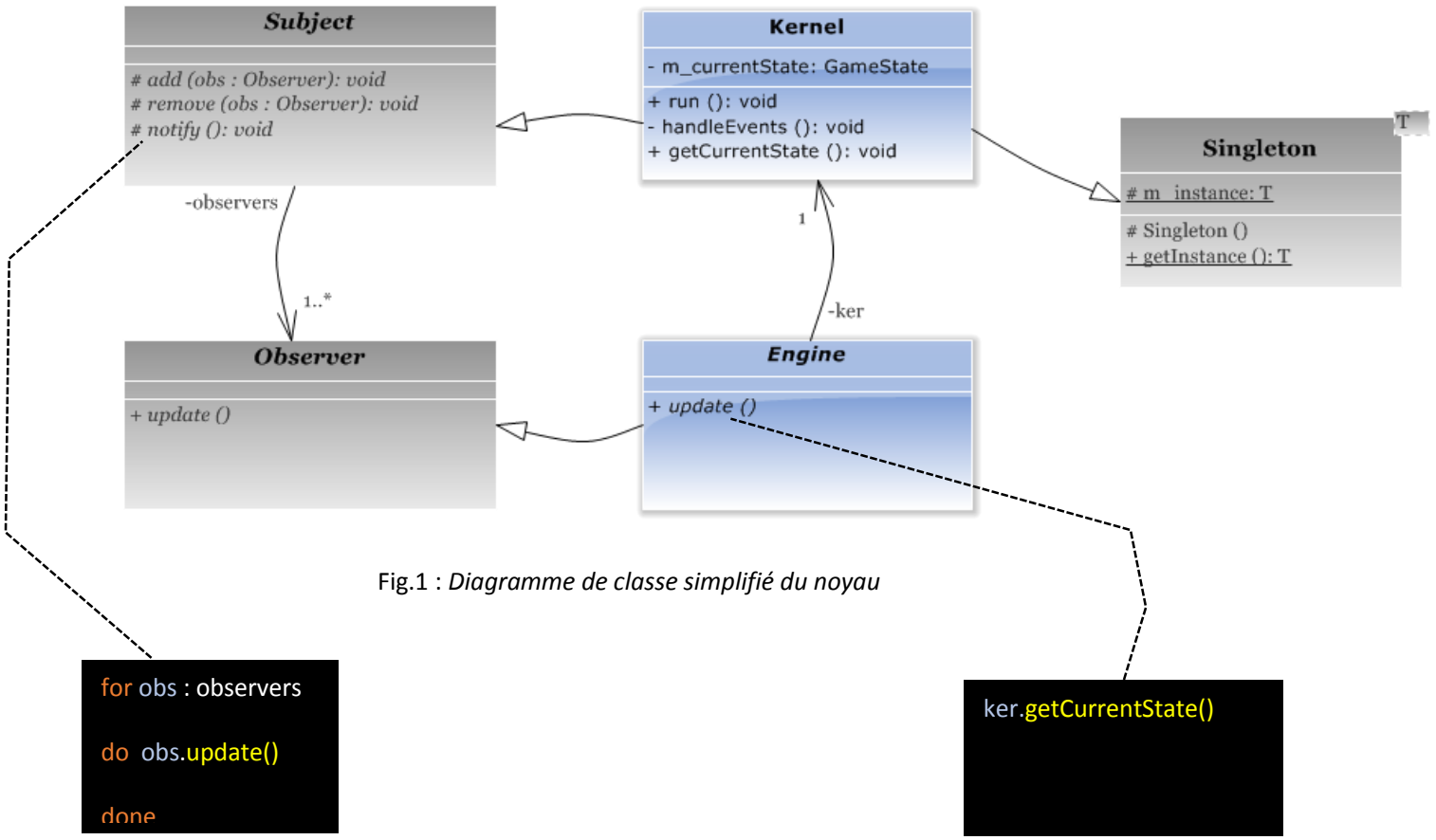


Fig.1 : *Diagramme de classe simplifié du noyau*

Association Kernel-GameState

Malgré la réalisation du patron observateur et la notification par le noyau à tous les moteurs de jeu, il reste que la conception d'une telle classe risque d'être complexe, si l'on veut que l'implémentation reste *lisible* et *maintenable*. Pour cette raison nous nous proposons d'associer un gestionnaire d'états au noyau, afin de décharger le noyau des tâches qu'un état en cours peut réaliser lui-même.

Pertinence de l'association

Cette association (qui est en fait une agrégation) permet au noyau de déléguer certaines de ses tâches à l'objet `m_currentState` –qui est une instance de la classe `GameState`– et se contente de notifier le changement d'état aux moteurs observateurs. Chaque moteur a ainsi accès à une référence vers l'état courant, et réalisera ses tâches sur cet état. (cf.).

Le rôle du noyau sera relégué ainsi, *grosso modo*, aux tâches de gestion des différents états.

Considérations techniques

- ✓ En tant que classe singleton, le noyau héritera d'une classe générique *Singleton* (qui servira ainsi pour d'autres classes du système) afin de factoriser le code, mais qui introduira un héritage multiple dans notre implémentation (puisque le noyau hérite déjà de *Subject*). Cependant, étant donné la spécificité et la simplicité de la classe template *Singleton*, cet héritage multiple ne devrait poser aucun problème de maintenance.

- ✓ Afin de respecter de façon puriste les principes de la conception objet, le *State Manager* devrait être une classe à part entière, associée au noyau, car le rôle du noyau n'est pas de « switcher » entre les différents états du jeu, mais juste de démarrer le jeu. Cependant, pour des questions de temps et de simplicité, nous choisissons d'implémenter les méthodes de gestion d'états directement dans le noyau. Ainsi le *StateManager* sera le noyau lui-même.

- ✓ Le noyau gère les changements d'état via une pile d'états `m_states`. L'état courant `m_currentState` sera simplement le sommet de cette pile.

- ✓ Une considération importante est celle-ci : comment le noyau sait quand passer d'un état à un autre ?
  En fait il ne le sait pas : il délègue cette tâche à l'objet `m_currentState` : en effet seul l'état courant peut savoir quand le gamer décide un changement d'état, ce qui lui permet de réaliser ce changement.
  Ceci permettra au noyau de mettre à jour son attribut `m_states`, et puis de notifier –grâce au patron observateur- ce changement d'état aux différents moteurs du jeu.

2.2 Gestion des états du jeu

Problématique

A ce stade de la conception, nous souhaitons faire en sorte que la gestion des différents états du jeu soit bien structurée au niveau de l'implémentation, afin d'augmenter la lisibilité et la maintenabilité du code source.

Cette stratégie, qui doit être adoptée pour toutes les composantes du système, augmentera l'efficacité et la facilité du travail des développeurs.

Fonctionnalité à implémenter

Tout jeu vidéo s'exécute en défilant devant le gamer des *«écrans »* clés du jeu, chacun représentant un état particulier, à travers lequel le jeu permet au gamer d'accéder à certains services.

Par exemple, tout jeu vidéo possède un état d'introduction (avec souvent une image caractéristique du jeu, ou même, dans les jeux les plus aboutis, une cinématique), puis un état de menu, qui donne à son tour la possibilité de démarrer la partie, et accessoirement aussi à un état d'options … etc. Lorsque la partie est terminée, le jeu affiche un état de *game over*, qui donne ensuite souvent la possibilité de retour au menu initial. Plus encore : un jeu peut même être simultanément dans plus d'un état à la fois, ainsi lorsque le joueur accède au menu des options pendant le déroulement de la partie.

Nous désirons mettre en place cette fonctionnalité d'états multiples pour notre jeu *Game Foot+ 2D*, en nous basant sur une conception robuste, qui permette la lisibilité et la réutilisabilité de notre code source.

La mauvaise méthode

Une méthode commune, immédiate mais maladroite, consisterait à gérer les différents états à travers une grande boucle, qui met en place un aiguillage[1] sur le type de l'état en cours.

Ainsi le programme démarre sur l'état d'introduction, puis boucle sur les entrées utilisateurs jusqu'à ce que le gamer réalise la manœuvre demandée (généralement, *« Appuyer sur une touche »*), puis l'écran de l'état de menu est affichée, jusqu'à ce qu'une sélection soit faite (par exemple, *« Commencer partie »*). Ensuite la partie du jeu commence, et boucle jusqu'à la fin de partie.

La boucle principale du jeu en devient très vite illisible et vulnérable aux erreurs de codage. En conséquence, le noyau du jeu, dans lequel cette boucle est en principe implémentée, et qui doit gérer entre autres les entrées utilisateurs, risque d'être une composante ambigüe de notre système, ce qui provoquera l'instabilité de l'application.

La bonne méthode

Définition formelle d'un état

Un état est une composante de l'application, capable de prendre soin d'elle-même, ce qui déchargera le noyau et les moteurs de la gestion tous de tous les évènements destinés à l'état en cours.

Ainsi, un état configurera sa propre scène (puis l'envoie au noyau, qui se charge de l'aiguiller au moteur graphique, qui la dessine).

Automate de gestion d'états du jeu

Au sein de notre programme de jeu, les transitions entre les différents états peuvent être modélisées par un automate à états finis déterministe.

N.B. : comme spécifié dans le GDD, dans un premier temps, l'application n'implémentera que l'état *PlayingGame.* L'implémentation des autres états est optionnelle.

Utilisation du patron de conception *Etat*

Sur la base de l'analyse précédente, et en nous référant aux travaux connus sur les patrons de conception en informatique, nous avons choisi d'implémenter, pour les états multiples du jeu, le patron de conception *« Etat »* (*pattern design « State »)*, afin d'en optimiser la gestion, et ainsi augmenter la robustesse de notre système.

Conception

Considérations techniques

- ✓ La classe `GameState` étant abstraite, on ne peut pas la déclarer *Singleton* (du fait que notre template `Singleton` instancie les classes qui l'implémentent). Ainsi tout objet héritant de GameState (donc un état concret) sera chacun déclaré individuellement *Singleton*.

2.3 Moteurs du jeu

[1] Qui peut être soit une séquence d'instructions `if`, soit une instruction `switch`.

La structure de base de la classe abstraite `Engine` a été déjà été discutée dans la section traitant du noyau.

Il s'agit ici d'en donner une conception plus approfondie, notamment en donnant la conception des classes des moteurs concrets (sous-classes d'`Engine`).

La classe `Engine` étant abstraite, on ne peut pas la déclarer *Singleton* (du fait que notre template `Singleton` instancie les classes qui l'implémentent). Ainsi tout objet héritant d'`Engine` (donc un moteur concret) sera déclaré individuellement *Singleton*.

Le moteur graphique a pour tâche de dessiner la scène de l'état courant, pour cela il doit avoir accès aux entités graphiques définies par l'objet `m_currentGameState`.

3. Couche *Gameplay*

3.1 Gestion de la scène

3.2 Entités graphiques