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.

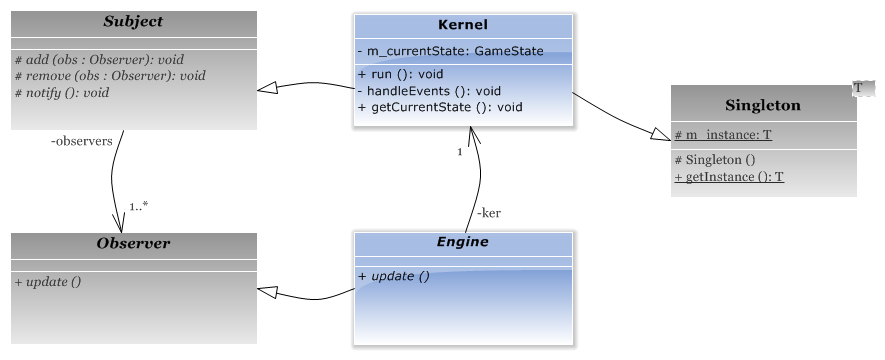


Fig.1 : *Diagramme de classe simplifié du noyau*

ker.getCurrentState()

for obs : observers

do obs.update()

done

### 

##### 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.