# ***CONCEPTION***

## *ESSENTIEL :*

**1.** il y a 2 jeux, le premier BikeGame est le jeu qui représente le travail attendu dans la partie 5. Le deuxième BikeGame2 est le jeu sur lequel nous avons ajouté des extensions, ajouté plus d'options etc ( la partie 6).
**2.** Afin de ne pas nous répéter nous tenons a vous dire que vous trouverez dans la quasi totalité des nos classes des fonctions comme « graphicCreator(), partCreator(), ***Creator() » etc… Ces fonctions ont pour but de simplifier le code, de le rendre plus facile à modifier et plus propre à lire. Cependant d'une classe à l'autre les besoins étaient différents, nous n'avons donc pas pu créer (par exemple) une interface qui implementerait toutes nos classes.
**3.** Nous avons essayé de commenter au maximum le code afin d'expliquer clairement quelles méthodes réalisent quelles actions etc… Si ce fichier « CONCEPTION » n'est pas assez claire à certains endroits pour vous, n'hésitez pas a regarder le code et ses commentaires. Merci
4. N'ayant pas appris a utiliser les *Timers* pendant le semestre nous avons bricolé nos propres Timer en utilisant le temps « *deltaTime* ». Nous savons que ce n'est pas parfaitement précis mais nous avons fait en sorte que ce le soit le plus possible.

## *Partie 3 (Tutoriel) :*

- Nous avons strictement suivie le tutoriel afin de comprendre le fonctionnement de l'API et du moteur graphique. Aucune modification significative a retenir.

## *Partie 4 (Architecture) :*

- Nous avons gardé l'architecture de base (Actor / ActorGame / GameEntity)

**Actor :**
- Nous avons suivie les instructions du PDF, Actor est donc une interface qui hérite des interfaces Graphics et Positionable. L'interface est constitué de 3 méthodes : update (pour actualiser un acteur pendant le jeu), destroy (pour détruire/supprimer un acteur du jeu) et de getEntity (qui sert lorsqu'on regarde si l'acteur en contact avec x est bien celui qu'on veut).

**ActorGame :**
-Nous avons suivie les instructions du PDF. Mais nous avons rajouté quelques méthodes. Premièrement la méthode getPayLoad() qui est public car elle doit être accessible et qui retourne l'acteur principal du jeu (le vélo Bike). De plus afin de garder une encapsulation correcte nous n'avons pas mit de getWorld() (trop intrusif) et avons donc opté pour le rajout de fonction tel que entityConstructor(), WheelConstraintBuilder(), RopeConstraintBuilder() et RevoluteConstraintBuilder() qui permette de construire des entités ou contrainte lorsqu'on se situ dans d'autre classe sans utiliser de getWorld(). Ces méthodes sont déclaré publique SAUF entityConstructor() car elles doivent être accessible pour n'importe quelle classe qui souhaite construire une contrainte. EntityConstructor() est déclaré protected car elle n'est utile que pour GameEntity situé dans le meme package donc ce n'est pas la peine d'entendre son autorisation à publique. Enfin nous avons rajouté une fonction

deleteAllActor() qui permet de supprimer tout les acteur de la liste (en cas de reset par exemple) et qui est déclarée protégée car elle n'est utile que dans les jeu (BikeGame/BikeGame2) qui hérite de ActorGame.

**GameEntity :**
-Nous avons ici aussi suivie les instructions du PDF. Nous avons cependant rajouté quelque fonction utile a notre programme. Tout d'abord une fonction PartIn() qui vérifie que la Part passée en argument appartient a l'entité en question (l'acteur). Typiquement cette fonction permet de faire en sorte qu'il n'y ai pas de collision entre la hitbox du cycliste et les roues car elles sont tous dans le même acteur : Bike. Cette fonction est déclaré protégé car elle n'est utile que dans ce package et pour les classes héritant de GameEntity.

**Crate et CrateGame :**
-Pour Crate vous pouvez observer l'architecture typique que ne utilisons pour nos classes, a savoir quelques attributs indispensables, deux constructeurs (ce ne sera qu'un parfois) des méthodes de type ***Creator() cf. ESSENTIEL 2. Afin de permettre la destruction de la boite, nous ajoutons dans la méthode destroy une fonctionnalité qui supprime l'acteur de la liste afin qu'il ne soit plus simulé. CrateGame est quand a lui très simple, on declare les caisses dans l'initialisation (begin) et ajoute les caisses dans la liste d'acteurs de ActorGame. Pas besoin de réécrire update ou end car nous n'ajoutons rien ici.

## *Partie 5 (BikeGame) :*

**Terrain :**
- La classe terrain est une classe assez simpliste, nous n'avons rien ajouté de particulier et nous avons utilisé notre architecture de classe habituelle. CEPENDANT, comme nous devons pouvoir initialiser la forme du terrain depuis sa création (c'est a dire dans BikeGame) nous avons rajouté dans Bike game quelque attributs/fonctions utile. Nous y viendrons plus tard.

**Wheel :**
- Nous avons suivie les instructions du PDF et utilisé l'architecture habituelle pour cette classe. Nous avons rajouté cependant plusieurs éléments. Tout d'abord étant donné que nous avons fait le choix d'utiliser des Shape pour dessiner les roues et non pas des image, nous avons créé un « rayon » de roue sous forme d'une Polyline qui permet de rendre l'esthétique du jeu plus agréable et aussi de contrôler si les roues tournent ou pas. (fonction graphicRayon() privée bien entendu car utilisé que dans cette classe). Pour le reste nous avons, comme nous l'avons dit, suivi le pdf afin de créer les roues, les fonction permettant la motorisation, et la contrainte liant la roue a un véhicule etc…

**Bike :**
- Pour la class Bike nous avons également suivie le PDF et utilisé notre architecture de base. Nous avons créer le « graphisme » du personnage a l'aide de Polyline qui se base sur des getters que nous avons déclaré privé car ils ne sont utilisé qu'au sein même de la classe. Afin de contrôle le regard du personnage (et donc le graphique de celui-ci mais aussi la roue motorisé etc..) nous avons déclaré un boolean « lookRigh » qui indique si vrai qu'on regarde a droite sinon a gauche. Afin

de pouvoir modifier ce paramètre et de le vérifier nous avons créé un get et un set (publique car ils doivent être accessible dans le contrôle du jeu par exemple). Nous avons aussi dans la classe bike définie des fonction publiques telles que accel(), stop() ou relaxMotor() qui on pour but d'épurer la class BikeGame en vérifiant les conditions d'accélération de freinage et de désactivation du moteur au sein même de la class Bike. De plus on trouve une fonction changeWayOfLook() qui permet de changer le graphique du cycliste (sa direction de regard).

**Contrôles :**
Nous avons utilisé les contrôles recommandés dans le PDF

**Chute :**
Nous avons implémenté un contact listener a Bike afin de pouvoir détecter les chutes/contacts avec la hitBox qui est couplé a un boolean qui « devient » vrai lorsque la hitBox est touchée. Nous avons placé un getter de ce boolean afin de pouvoir contrôler dans BikeGame si le personnage a chuté ou non (voir update() dans BikeGame).

**Finish :**
- Pas de modification par rapport au PDF

**Gestion des fin de partie :**
-idem

**BikeGame :**
-Pour BikeGame nous avons suivie simplement les instruction du PDF. Nous avons laissé la déclaration et l'initialisation des messages dans la classe BikeGame. Nous avons cependant ajouté une liste de vecteur et un méthode permettant d'ajouter des vecteur a cette liste addVector(). Cette liste est ensuite passé en paramètre dans le constructeur de terrain, ainsi nous avons bien comme demandé la possibilité de changer le terrain lors de l'initialisation.

**Gestion des exceptions/erreurs :**
**-** Comme demandé dans le pdf nous avons codé la gestions de exceptions. Pour ce faire nous avons fait en sorte que dans chacune des classes possédant des constructeur (typiquement les classes représentant des *Actor*) si un argument passé d'est pas valide (par exemple une hauteur négative) alors on lance « throw » une exception de type IllégalArgumentException a laquelle on ajouter un petit message comme « The width must be positive » ou autre en fonction de l'argument. Ensuite dans l'initialisation du jeu (méthode Begin() de BikeGame/BikeGame2) en créer le bloc « try » dans lequel on place toute la construction de nos acteurs et finalement on retrouve après ce bloc un bloc « catch » permettant d'intercepter les possibles erreurs lancé dans les constructeur. (On retrouve également un bloc catch vérifiant que le jeu game de « ActorGame » n'est pas null ou que la position d'un objet n'est null non plus)

## *Partie 6 (Extensions) :*

-La partie 6 est sans doute la partie la plus aboutit du projet, vous trouverez dans BikeGame2 le jeu avec toute les extensions.

**1.Trigger**
-Nous avons fait l'extension Trigger (avec sa dérivé object collectable ET checkpoint). Nous avons choisie de créer cette classe abstraite et de la faire hérité de GameEntity, ainsi lorsque que nous voudront créer un objet héritant de Trigger il héritera directement de GameEntity. Dans cette classe abstraite on retrouve la création d'une classe « anonyme » permettant de créer un système de ContactListener propre a Trigger. Nous avons un boolean « bikeTouch » qui par défaut est faux. La classe anonyme permet de detecter les contacts et SI l'objet en contact est le vélo (son entité plus précisément) alors « bikeTouch » est passé a true et ainsi nous savons si l'objet a était en contact avec le vélo ou non. Nous avons également ajouté des fonctions telles que partBuilderCreator(), graphicCreator() (protégées car seulement utile pour les classes qui hérite de Trigger) qui sont générales aux sous classes de Trigger. Nous avons également un getter et setter de « bikeTouch » a accès publique pour permettre d'y avoir accès (typiquement dans la mettre update du jeu). Une méthode addContact() permettant de lier le ContactListener a l'objet héritant de Trigger.

**2. Checkpoint (Trigger)**
-Nous avons créer une classe Checkpoint héritant de Trigger. Nous reprenons exactement l'architecture fourni par Trigger. Dans le BikeGame2 (méthode update()) Nous faisons en sorte que lorsque nous avons passé le checkpoint nous pouvons y réapparaitre en gardant nos points acquis AVANT le passage du checkpoint ou autre (cf. code BikeGame2 fonction update()).

**3. FinishTrigger (Trigger)**
- Ici on a recréer une classe Finish mais qui cette fois-ci hérite de Trigger. On a reprit exactement l'architecture fournit par Trigger. Pour voir les actions liée a la ligne d'arrivé il faut aller dans BikeGame2(on a l'affichage du score final, un affichage de victoire etc…)

**4. Star (object collectionnante Trigger)**
-Nous avons créé une classe Star qui hérite de Trigger et qui a pour but de représenter des objets collectionnables (des étoiles) qui rapporte 100 points si vous les attraper. L'architecture est ici encore typiquement celle fournit par Trigger sans rajout particulier.

**5. Flamme (Trigger)**
- Cette classe hérite également de Trigger. Comment son nom l'indique c'est une casse permettant de créer des flammes qui nous tue si l'on tombe dedans. On a ici également reprit l'architecture fournit part Trigger sans rajouter de fonction particulière mise a part le fait que cette classe possède un émetteur de particules. On a donc ajouté un émetteur et crée des particules adaptées aux flammes. On va donc forcément retrouvé les fonction destroyEmitter() ou encore update() qui permette d'actualiser les particules de l'émetteur. (cf. Emetteur/Particules/ ImageParticule).

**6. Spikes (Trigger)**
- Nous avons créer la classe Spires qui hérite de Trigger. Cette classe a pour but de représenter des pics qui, lorsqu'on roule dessus, nous immobilise pendant 5 sec

(grâce a un aimer créer dans BikeGame2). Cette perte de 5seconde entraine une perte de points automatique a la fin du jeu car le temps mit pour faire le parcours rapporte des points. L'architecture utilise ici est celle fournit par Trigger, on n'y a pas ajouté fonction particulière.

**7. Block**

-Cette classe est fondamentale, elle permet de créer un block en choisissant son graphic, sa hauteur, sa largeur, sa position, a profondeur. Elle est indispensable dans la creation d'autre objet (comme des pendules, des bascules) ou meme pour des elements du decors( comme la lune ou le nuage que nous avons placé en niveau du start). L'architecture de cette classe est simple, elle utilise un constructeur et des ethos de type ***Creator() (cf. ESSENTIEL 2).

**8. Ball**

-A la meme manière que Block, Ball est une classe basique permettant de créer des objet circulaire en tout genre. 2 constructeur y sont fouine, l'un pour faire des balles avec des ImageGraphics, l'autre pour faire des balles avec des ShapeGrahics. On a utilisé une Architecture classique ici aussi.

**8. Pendules**

-Nous avons créer deux classes pendule Pendulum et PendulumBall. La premiere est un dérive d'un pendule, on y attache en fait un petit bloc a un plus grand (relier par 2 contraintes) afin de fair une plate-forme non stable qui oscille a la manière d'un pendule simple. Pour se faire nous utilisons des fonctions permettant la création de contrainte qui passe par des fonction de ActorGame( voir la section sur ActorGame).

-Le deuxième pendule PendulumBall est un pendule beaucoup plus simple. C'est un pendule classique au bout duquel un balle oscille et il faut l'éviter. Afin de créer ce pendule on a utiliser des objets comme Block et Ball et nous les avons liée par une contrainte .

**10. Bascule**

-Afin de créer la bascule (classe Swing) nous nous sommes grandement inspiré du tutoriel. Nous avons donc utilise Block pour créer 2 block et nous les avons liée face a une contraint que nous créons dans ActorGame (voir section ActorGame).

**11. Terrain glissant**

- Nous avons fait le choix de créer une classe IceBlock, qui permet de créer des blocks de glace comme son nom l'indique. Nous avons fais en sorte que la friction de ces bloques soient quasiment nul (0.001) afin d'avoir un effet « glissant » lorsque le vélo se déplace sur ces blocks. Nous avons fait le choix de ne pas laisser le choix du graphique avant de garder une certaine cohérence (la glace rend cohérent le coté glissant, si on créer un block de pierre glissant il n'y pas plus de sens). On retrouver l'architecture classique de nos classes, avec des méthodes ****Creator() et un constructeur nous permettant de définir la position et la taille du block de glace. De plus nous avons décidé d'ajouter un émetteur lié a la création d'un block de glace afin de donner l'impression qu'il neige sur cette partie du terrain. Ainsi on retrouve les fonction propres aux émetteurs (cf Emétteur/Particle/ImageParticle). Cela n'apporte pas de modification au constructeur car on impose la construction d'un émetteur a neige a partir du moment ou on construit un terrain glissant.

**12. Les decors**
-Plusieurs de nos classes ne sont que des « décors », on aurai pu les créer a partir de Block ou de Ball. Met certain décors peut être répétitif/général et donc il nous ait paru plus logique de créer des classes telles que TreeSnow qui représente des arbres enneigés, ou bien Start qui représente le drapeau de départ d'un partie. En créant ces classes a part entière on donne la possibilité d'utilise plus facilement ces décors dans le cas ou nous voudront rajouter du décors ou codé un autre niveau. Ces classes sont simple, hérite de GameEntity et sont implémentée par Actor. Les mettons sont des simples *****Creator() et un constructeur.

**13. Bike (ajout)**
- Nous avons ajouté certaine fonctionnalité dans la class Bike. Notamment des fonction comme ArmUp ou NomalWay qui permettent de faire lever les bras du personnage lorsqu'il passe un checkpoint par exemple. Tout ces fonctions sont au sein même de la classe Bike: cela nous permet de garder une bonne encapsulation en limitant au maximum les getter/setter.

**14. BikeExtends**
-Cette classe est une classe qui hérité de Bike. Nous l'avons créer afin de pouvoir faire une vélo qui émet des particules. On aurait pu coder directement sur la classe Bike mais on aurait alors eu aussi le particules sur le vélo du jeu de la partie 5 (BikeGame) (On aurais aussi pu ajouter un boolean dans le constructeur pour decider lors de la creation si oui ou non l'Emetteur sera créé, ce sont deux méthodes possibles. Nous avons choisis la première pour ne pas trop modifier ce que nous avions fait avant). Les fonctions de cette classes sont donc typiquement celle de Bike, à la seule différence qu'on a rajouté un émetteur de particule sur le vélo et rajouté une fonction permettant de détruire l'émetteur seul et également une fonction Update() pour actualiser les particules de l'emetteur (nous verrons par la suite pourquoi).

**15. Particle :**
- Cette classe abstraite définit les caractéristiques liés a la position de la particle. En effet, celle-ci n'est ni une GameEntity ni un Actor (elle se limitera d'implementer ses super-Interfaces: Graphics et Positionable). Le plus important a retenir de ceci est que la particle n'a pas d'**Entity**, elle n'a pas de représentation graphique: ce n'est qu'une representation graphique (pouvant être positionné). Il faut donc trouver une autre façon de la faire "bouger". C'est cela dont la classe Particle va se charger. Elle composé d'attributs caractérisant la position, vitesse et accélération, et d'une méthode qui fait évoluer celles-ci au cours du temps. Elle est également composé d'une méthode abstraite copy() qui permettra de copier des particules aisément.

**16. ImageParticle :**
- Cette classe hérite de Particle. Elle nous permettra de formaliser le dessin de la particule dans le jeu. Comme son nom l'indique, elle liera une image png à chaque Particule. La méthode draw(), la plus importante de cette classe, affichera la particle selon le Transform de Particle (calculé comme on l'a dit avec les attributs liés a la position) et la rendra moins visible à chaque affichage, à l'aide de l'attribut alpha (on aurait également pu controller cela depuis l'émetteur et ajouter une méthode

reduceAlpha(float a) mais nous avons finalement décider de simplifier la chose) . Cette classe est également doté de tout les setters et getters qui pourront être utiles.

**17. Emitter :**

- Cette classe regroupera une quantité d'instances identique (si ce n'est par la transparence et la position/vitesse) de la classe Particle (On pourra plus tard en effet distinguer les en un ArrayList. Elle sera définie par un forme (Shape) concrète à l'intérieur laquelle elle dessinera les particules **seulement** si celles-ci ont une transparence non nulle (elle sont effacés et remplacés par de nouvelles si c'est le cas). Á chaque création d'instance de Particle, on utilise la très utile méthode copy(). On a également ajouté un methode attach() similaire a celle de Wheel mais avec un WeldConstraint cette fois, puisqu'on ne veux que l'émetteur (en théorie) ne tourne comme une roue. L'émetteur va également offrir une methode update qui va permettre a son utilisateur de mettre a jour la position des particules, puisque celles-ci ne sont pas de Actor et ne peuvent donc pas être misent a jour comme les jusqu'ici (par le biais d'ActorGame).

**BIKEGAME2**

-Beaucoup de changement ont été nécessaire pour le bon fonctionnement du jeu. Tout d'abord afin de pouvoir contrôler au mieux les événements du jeu nous avons du déclarer certain Acteur au sien même de la classe BikeGame2 (notamment les éléments héritant de Trigger  mais aussi le vélo (Bike)). En déclarant ces éléments nous avons accès au fonction publiques qu'ils proposent. Nous avons également créer des timer (ne sachant pas utiliser Timer correctement nous avons créé nos propre timer : des variables qui vont compter le temps qui s'écoule après un certain événement). Afin de respecter les consigne données dans la partie 5 les textes que nous avons créé (comme l'affichage du temps, du score, des instructions pour reset etc…) le sont directement dans le jeu Bikegame2. Pour le méthode begin() on retrouve quelque chose de similaire a BikeGame sauf que biensûr on a maintenant beaucoup plus d'acteurs. Le gros du changement se situe dans la fonction Update(). En effet grâce au classe héritant de Trigger nous pouvons créer bien plus de commandes. Par exemple lorsqu'on franchit la ligne on affiche en grand notre score final et un message de victoire. Si l'on passe le checkpoint on enregistre la position pour pouvoir donner cette position au vélo lorsqu'on le fera réapparaitre au niveau du checkpoint. C'est aussi ici que l'on gère nos objets collectage, pour notre part lorsqu'on collecte une étoile celle ci disparait du jeu et on obtient 100 points supplémentaire. Si on touche les pics (spikes) alors on est immobilisé pendant 5secondes.  On définie aussi nos conditions en cas de reset ou restart depuis le checkpoint (remettre les timer a 0 en cas de reset, supprimer les points etc…)

**.PNG**

Nous avons ajouté certaines image en .png dans le dossier ressource. Voici la liste :

- gliss.png qui représente un panneau de signalisation « sol glissant »
- ice.png qui représente une block de glace.
- tree.snow.png qui représente un sapin enneigé.
- flam.png qui représente des flammes.

## *ARCHITECHTURE*

Nous avons essayé de garder une architecture cohérente et nous allons essayer de la résumer ici :

**1.** Nos classe « classiques » (Block, Ball, Pendulum…) sont des classes héritant de GameEntity et implémentés par Actor. Elle redéfinisse les fonctions fournies et on y retrouve les fonctions de type ****Creator() (cf. ESSENTIEL 2).On retrouve toujours au moins 1 constructeur (parfois 2 cela dépend de la liberté qu'on a voulu donner pour la création de ces objets). Ces classes représentent des objets généraux, qu'on peut retrouver a plusieurs endroit du jeu et dans différents jeux.

**2.** Les classes héritant de Trigger. Ces classes ressemblent en tout points aux classes « classiques » a la différence que celles ci on un ContactListenner qui permet d'enregistrer si il y a eu contact ou non avec l'acteur principal du jeu (pour nous le vélo). Ces classes ne possède PAS les actions a faire en cas de contact car cela dépend du jeu qu'on veut créer et donc on retrouve tout cela dans la classe BikeGame2. On retrouve dans ces classes les objets qui doivent typiquement « interagir » sur le jeu (Finish, Checkpoint, object collectionnable, pics etc…)

**3.** Les classes possédant des émméteur de particules, typiquement celle ci peuvent être soit classiques soit des Trigger. Ces classes ont la particularité d'être couplées à un émetteur qui va propulser des particules dont on choisie la taille, l'image etc… On retrouve dans Flamme un émetteur de particule qui donne l'impression que le feu est vivant. On a aussi créer une classe BikeExtends qui un Bike mais auquel on rajouter un émetteur de particules pour rajouter un coté esthétique et vivant au jeu ou encore nous avons lié un émetteur à IceBlock pour donner une impression de neige au dessus du terrain glissant.

**4.** Le jeu en lui même (BikeGame2). Son rôle est de définir les règles du jeu. C'est dans cette classe qu'on va retrouver les actions a faire en cas de contact avec les classes héritant de Trigger(ajouter 100points si on récupère une étoile, immobiliser le personnage si on roule sur les pics etc…). Ils contrôles également le reset et le restart du checkpoint en réinitialisant les timer, les points etc…). Cette classe gère également la création du monde.