Sujet de Projet CPS : *Street Fighter*

Le projet a pour objectif la réalisation d'un programme entièrement spécifié selon la méthode Design-by-Contract. Le rendu est composé d'une unique archive contenant:

- la spécification semi-formelle des services utilisés par le projet sous forme d'interface **Java** (e.g. une interface `Engine`),
- l'implémentation de spécification par des contrats selon le motif *Decorator* vu en cours (e.g. deux classes `EngineDecorator` et `EngineContract`),
- (au moins) deux implémentations des services spécifiés, une implémentation correcte et une implémentation buggée (e.g. deux classes `EngineImpl` et `EngineBugImpl`).
- une série de test **JUnit** pertinents, élaborés selon la méthode MBT.
- un fichier `build.xml` permettant la compilation, l'execution et le test des implémentations.
- un rapport en `pdf`.

En outre, le projet doit être présenté lors d'une soutenance de 15 minutes.

1 Contenu Nécessaire

Le projet doit contenir les fonctionnalités suivantes:

1. **Noyau**: Le projet doit implémenter **et spécifier**[1] les services décrits dans le partiel: les hitbox, le moteur de jeu, les personnages avec déplacement gauche et droite, mise en garde et techniques et le compteur de combo.
2. **Affichage**: Le projet doit implémenter (mais pas forcément spécifier, bien que toute spécification soit valorisée) un affichage du jeu. Il est conseillé de se concentrer sur un affichage minimum (hitbox de couleurs dans une fenêtre graphique pour les personnages et affichage minimal de la vie et du compteur de combo) qui permet une compréhension rapide de l'implémentation, sans forcément utiliser des techniques d'affichage avancées (sprites, animations, . . .) (même si elles seront appréciées).
3. **Commande**: Le projet doit implémenter (mais pas forcément spécifier) des commandes pour le jeu, qui permettent à deux personnes de jouer sur un même clavier.
4. **Déplacements avancés**: Le projet doit implémenter **et spécifier** la posture accroupie et le saut. Quand un joueur envoie la commande DOWN (ou DOWNRIGHT ou DOWNLEFT), le personnage s'agenouille, il devient plus difficile à toucher: sa hitbox (et la palette des coups disponibles, si possible) doit se modifier en conséquence. Un personnage qui se baisse peut se protéger. Quand un joueur envoie la commande UP ou UPRIGHT ou UPLEFT, son personnage saute sur place, vers la gauche ou vers la droite. Sa hitbox suit une trajectoire fixée à l'avance (on ne peut pas se déplacer pendant un saut). Quand deux personnages se retrouvent dos à dos apres un saut, ils se retournent automatiquement pour se faire face dès qu'ils reprennent le contrôle (un personnage bloqué dans une technique ou un *stun* attend la fin de sa technique ou du *stun* avant de se retourner).
5. **Techniques avancées**: Un personnage doit disposer d'au moins 3 techniques différentes, avec des caractéristiques (frames, stun, dégats) variées.

[1] avec contrats et tests

2 Contenu Supplémentaire

Pour obtenir une bonne évaluation, le projet doit en outre intégrer des extensions ou des fonctionnalités, prises dans la liste suivante ou imaginée (ou inspirée de jeux existants). **Attention:** toute extension ou fonctionnalité doit être accompagnée de sa spécification (avec contrats et tests).

Extensions possibles:

- **Combattants:** Avant la bagarre, Les joueurs peuvent choisir entre un certain nombre de combattants prédéterminés. Chaque combattant possède des caractéristiques (vie, vitesse, hitbox, palette de techniques, et, pourquoi-pas, représentation dans l'affichage) différentes.
- **Techniques spéciales:** Une suite de commande précise d'un joueur (par exemple DOWN, DOWNRIGHT, RIGHT, ATTACK pour un personnage regardant vers la droite) déclenche des techniques spéciales. On fera attention à la fenêtre autorisée pour rentrer la séquence (par exemple, il faut décider si les quatre commandes précédentes doivent être entrées durant 4 frames successives ou si l'on autorise une entrée des commandes moins précise).
- **IA:** Le jeu propose des parties pour un seul joueur dans lesquelles l'adversaire est contrôlé par le programme selon un comportement simple incluant de l'aléatoire (il essaye de s'approcher du joueur, lance des techniques et se protège).

Fonctionnalités possibles:

- **Low/Overhead:** Certaines techniques (appelées *low*) peuvent toucher (hit) un personnage qui se protège sans se baisser. Inversement, certaines techniques (appelées *overhead*) peuvent toucher un personnage qui se protège en se baissant.
- **Counter:** Une technique qui touche un adversaire qui est lui-même en train d'effectuer une technique fait plus de dégats et de hitstun que si elle touchait un adversaire qui ne fait rien.
- **Knockdown:** Certaines techniques font tomber l'adversaire au sol. Un adversaire au sol perd le contrôle de son personnage et ne peut pas être touché. Il se relève automatiquement au bout d'un certain nombre de frames.
- **Prises:** Certaines techniques peuvent toucher un adversaire qui se protège et le font tomber.
- **Multijoueur:** Gestion des bagarres à plus de 2 joueurs (par exemple 4).
- **Arenes:** Introduction d'arènes particulières (avec des trous ou des plateformes).
- **Multiple Hit:** Certaines techniques peuvent toucher plusieurs fois. Par exemple un coup qui touche 3 fois voit ses 9 hitframes séparées en 3 groupes de 3 frames, chaque groupe a une hitbox différente, des caractéristiques (stun / dégats) différents et chaque partie peut toucher un même adversaire successivement (ce qui viole la règle de "une technique ne peut toucher un adversaire qu'une seule fois").

3 Rapport

Le rapport se compose de 3 parties:

1. Un manuel d'utilisation succint de l'implémentation.
2. La spécification formelle complète du projet.
3. Un rapport de projet proprement dit, se concentrant sur les choix et les difficultés de spécification, d'implémentation, de tests, exhibant des exemples choisis pour leur pertinence.

4 Soutenance

Une soutenance de 15 minutes se compose d'environ:

1. 10 minutes de rapport: présentation (rapide) du projet, puis exploration de différents cas de spécification, contrats ou tests pertinents.
2. 5 minutes de démonstration.

A Correction du Partiel

Voici un début de proposition de "correction" pour le partiel. Il n'est absolument pas nécessaire de suivre cette proposition pour le projet.

A.1 Préliminaires: Collisions

Attention: la hitbox n'a, a priori, aucun moyen de connaître les limites du terrain de jeu. A moins qu'elle les récupère à la création, la hitbox n'a pas à tester la sortie des limites de jeu (le moteur doit s'en occuper). Ainsi, ce service n'a pas de préconditions.

Service: Hitbox
Types: bool, int
Observators: `PositionX`: [Hitbox] → int
`PositionY`: [Hitbox] → int
`BelongsTo`: [Hitbox] × int × int → bool
`CollidesWith`: [Hitbox] × Hitbox → bool
`EqualsTo`: [Hitbox] × Hitbox → bool
Constructors: `init`: int × int → [Hitbox]
Operators: `MoveTo`: [Hitbox] × int × int → [Hitbox]
Observations:
[invariant]:
`CollidesWith(H,H1)` = ∃ `x,y`:int × int, `BelongsTo(H,x,y)` ∧ `BelongsTo(H1,x,y)`
`EqualsTo(H,H1)` = ∀ `x,y`:int × int, `BelongsTo(H,x,y)` = `BelongsTo(H1,x,y)`
[init]:
`PositionX(init(x,y))` = `x`
`PositionY(init(x,y))` = `y`
[MoveTo]:
`PositionX(MoveTo(H,x,y))` = `x`
`PositionY(MoveTo(H,x,y))` = `y`
∀ `u,v`:int × int, `BelongsTo(MoveTo(H,x,y),u,v)` =
`Belongsto(H,u-(x-PositionX(H)),v-(y-PositionY(H))`

Schéma d'un code possible de la méthode `MoveTo` de la classe `HitboxContract`:

```
public void MoveTo(int x, int y){
  checkInvariant();
  (* Capture du centre *)
  belongsTo_centre_at_pre = belongsTo(getPositionX(), getPositionY());
  (* Capture du centre  + 100 *)
  belongsTo_centre_100_at_pre = belongsTo(getPositionX() + 100, getPositionY() + 100);
  (* Capture d'un point absolu *)
```

```
  getPositionX_at_pre = PositionX()
  getPositionY_at_pre = PositionY()
  belongsTo_abs_at_pre = belongsTo(300, 0);
  super.MoveTo(x,y);
  checkInvariant();
  (* Test du centre *)
  if(! belongsTo(PositionX(), PositionY()) == belongsTo_centre_at_pre)
    {throw new PostConditionError(...)}
  (* Test du centre + 100 *)
  if(! belongsTo(PositionX() + 100, PositionY() + 100) == belongsTo_centre_100_at_pre)
    {throw new PostConditionError(...)}
  (* Test d'un point absolu *)
  if(! belongsTo(300 + (x - PositionX_at_Pre), 0 + (y - PositionY_at_Pre)) == belongsTo_abs_at_pre)
    {throw new PostConditionError(...)}
}
```

A.2 Moteur de Jeu

Service: Engine
Types: bool, int, Commande
Observators: **const** `height`: [Engine] $\rightarrow$ int
const `width`: [Engine] $\rightarrow$ int
`char`: [Engine] $\times$ int $\rightarrow$ Character
pre char(E,i) **requires** i $\in \{1, 2\}$
`player`: [Engine] $\times$ int $\rightarrow$ Player
pre player(E,i) **requires** i $\in \{1, 2\}$
`gameOver`: [Engine] $\rightarrow$ bool
Constructors: `init`: int $\times$ int $\times$ int $\times$ Player $\times$ Player $\rightarrow$ [Engine]
pre init(h,w,s,p1,p2) **requires** $\mathrm{h} > 0 \wedge \mathrm{s} > 0 \wedge \mathrm{w} > \mathrm{s} \wedge \mathrm{p1} \neq \mathrm{p2}$
Operators: `step`: [Engine] $\times$ Commande $\times$ Commande $\rightarrow$ [Engine]
pre `step(E)` **requires** $\neg$`gameOver(E)`
Observations:
[invariant]:

$$\texttt{gameOver}(\texttt{E}) = \exists \texttt{i} \in \{1, 2\}\ \mathbf{Character}::\texttt{dead}(\texttt{player}(\texttt{E}, \texttt{i}))$$

[init]:

$$\texttt{height}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2})) = \texttt{h}$$
$$\texttt{width}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2})) = \texttt{w}$$
$$\texttt{player}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 1) = \texttt{p1}$$
$$\texttt{player}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 2) = \texttt{p2}$$
$$\mathbf{Character}::\texttt{positionX}(\texttt{char}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 1)) = \texttt{w}//2 - \texttt{s}//2$$
$$\mathbf{Character}::\texttt{positionX}(\texttt{char}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 2)) = \texttt{w}//2 + \texttt{s}//2$$
$$\mathbf{Character}::\texttt{positionY}(\texttt{char}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 1)) = 0$$
$$\mathbf{Character}::\texttt{positionY}(\texttt{char}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 2)) = 0$$
$$\mathbf{Character}::\texttt{faceRight}(\texttt{char}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 1))$$
$$\mathbf{Character}::\neg\texttt{faceRight}(\texttt{char}(\texttt{init}(\texttt{h}, \texttt{w}, \texttt{s}, \texttt{p1}, \texttt{p2}), 2))$$

[step]:

$$\texttt{char}(\texttt{step}(\texttt{E}, \texttt{C1}, \texttt{C2}), 1) = \texttt{step}(\texttt{char}(\texttt{E}, 1), \texttt{C1})$$
$$\texttt{char}(\texttt{step}(\texttt{E}, \texttt{C1}, \texttt{C2}), 2) = \texttt{step}(\texttt{char}(\texttt{E}, 2), \texttt{C2})$$

Service: Character
Types: bool, int, Commande
Observators: `positionX`: [Character] $\rightarrow$ int
`positionY`: [Character] $\rightarrow$ int
`engine`: [Character] $\rightarrow$ Engine
`charBox`: [Character] $\rightarrow$ Hitbox
`life`: [Character] $\rightarrow$ int
const `speed`: [Character] $\rightarrow$ int
`faceRight`: [Character] $\rightarrow$ bool
`dead`: [Character] $\rightarrow$ bool
Constructors: `init`: int $\times$ int $\times$ bool $\times$ Engine $\rightarrow$ [Character]
pre `init(l,s,f,e)` **requires** $\mathtt{l} > 0 \wedge \mathtt{s} > 0$
Operators: `moveLeft`: [Character] $\rightarrow$ [Character]
`moveRight`: [Character] $\rightarrow$ [Character]
`switchSide`: [Character] $\rightarrow$ [Character]
`step`: [Character] $\times$ Commande $\rightarrow$ [Character]
pre `step()` **requires** $\neg$`dead`
Observations:
[invariant]:

$\mathtt{positionX(C)} > 0 \wedge \mathtt{positionX(C)} < \mathbf{Engine}::\mathtt{width(engine)}$
$\mathtt{positionY(C)} > 0 \wedge \mathtt{positionY(C)} < \mathbf{Engine}::\mathtt{height(engine)}$
$\mathtt{dead(C)} = \neg(\mathtt{life} > 0)$

[init]:

$\mathtt{life(init(l,s,f,e))} = \mathtt{l} \wedge \mathtt{speed(init(l,s,f,e))} = \mathtt{s} \wedge \mathtt{faceRight(init(l,s,f,e))} = \mathtt{f}$
$\wedge \mathtt{engine(init(l,s,f,e))} = \mathtt{e}$
$\exists \mathtt{h} : \mathrm{Hitbox}, \mathtt{charbox(init(l,s,f,e))} = \mathtt{h}$

[moveLeft]:

$(\exists i, \mathtt{player(engine(C),i)} \neq \mathtt{C} \wedge \mathtt{collisionwith(hitbox(moveLeft(C)), hitbox(player(engine(C),i)))})$
$\Rightarrow \mathtt{positionX(moveLeft(C))} = \mathtt{positionX(C)}$
$\mathtt{positionX(C)} \leq \mathtt{speed(C)}$
$\wedge(\forall i, \mathtt{player(engine(C),i)} \neq \mathtt{C} \Rightarrow \neg\mathtt{collisionwith(hitbox(moveLeft(C)), hitbox(player(engine(C),i)))})$
$\Rightarrow \mathtt{positionX(moveLeft(C))} = \mathtt{positionX(C)} - \mathtt{speed(C)}$
$\mathtt{positionX(C)} > \mathtt{speed(C)}$
$\wedge(\forall i, \mathtt{player(engine(C),i)} \neq \mathtt{C} \Rightarrow \neg\mathtt{collisionwith(hitbox(moveLeft(C)), hitbox(player(engine(C),i)))})$
$\Rightarrow \mathtt{positionX(moveLeft(C))} = 0$
$\mathtt{faceRight(moveLeft(C))} = \mathtt{faceRight(C)} \wedge \mathtt{life(moveLeft(C))} = \mathtt{life(C)}$
$\mathtt{positionY(moveLeft(C))} = \mathtt{positionY(C)}$

[moveRight]:

...

[switchSide]:

$\mathtt{faceRight(switchSide(C))}! = \mathtt{faceRight(C)}$
$\mathtt{positionX(switchSide(C))} = \mathtt{positionX(C)}$

[step]:

$\mathtt{step(C, LEFT)} = \mathtt{moveLeft(C)}$
$\mathtt{step(C, RIGHT)} = \mathtt{moveRight(C)}$
$\mathtt{step(C, NEUTRAL)} = \mathtt{C}$

A.3 Techniques

On ne donne qu'un squelette pour cette section (et quelques préconditions). La rédaction d'observations correctes fait partie du projet.

Data: Tech
Observators: `damage`: [Tech] → int
`hstun`: [Tech] → int
`bstun`: [Tech] → int
`sframe`: [Tech] → int
`hframe`: [Tech] → int
`rframe`: [Tech] → int
`hitbox`: [Tech] × int × int → Hitbox

Service: FightChar **refines** Character
Observators: `isBlocking`: [FightChar] → bool
`isBlockstunned`: [FightChar] → bool
`isHitstunned`: [FightChar] → bool
`isTeching`: [FightChar] → bool
`tech`: [FightChar] → Tech
pre `tech(C)` **requires** `isTeching(C)`
`techFrame`: [FightChar] → bool
pre `techFrame(C)` **requires** `isTeching(C)`
`techHasAlreadyHit`: [FightChar] → bool
pre `techHasAlreadyHit(C)` **requires** `isTeching(C)`
Operators: `startTech`: [FightChar] × Tech → [FightChar]
pre `startTech(C,T)` **requires** ¬`isTeching(C)`
Observation:
...