Space Invaders - Piste Verte

Ce document va vous conduire à travers le projet Space Invaders étape après étape.

L'approche de développement adoptée est incrémentale. Cette approche consiste à développer une à une les fonctionnalités de manière à toujours avoir un programme fonctionnel (le programme doit preque toutjours être compilable et testable). Plutôt que de chercher tout de suite la structure idéale de notre programme (découpage en classes, fonctions...), on remaniera le code au fur et à mesure des besoins. On reviendra donc plusieurs fois sur le code déjà produit afin de l'améliorer : on parle de *refactoring*.

Cette approche est particulièrement bien adaptée lorsque l'on aborde un problème sur lequel on a peu d'expérience. En effet, sans expérience, il est impossible d'anticiper correctement les contraintes et les besoins et il est alors généralement moins risqué de construire un modèle que l'on rafine au fur et à mesure des développements plutôt que d'essayer d'imaginer dès le début le modèle idéal.

1) Préparation

Téléchargement du squelette de projet et création de la classe `Vecteur2d` déjà réalisés.

2) Classe SpaceShip

La première fonctionnalité que l'on va développer est d'avoir un vaisseau pour le joueur qui sera affiché à l'écran. Les vaisseau du joueur sera représenté par la classe `SpaceShip`.

Des images de vaisseaux sont déjà incluses dans le projet SpaceInvaders (Menu *Projet -> Propriétés -> Ressources -> Images*). Ces images sont incluses dans l'exécutable généré à partir de la solution et sont accessibles directement depuis le code. Pour dessiner une image `image` à la position `x, y` (coin supérieur gauche de l'image), utilisez la méthode `DrawImage` de l'objet `graphics`.

```
float positionX = ...;
float positionY = ...;
Bitmap image = SpaceInvaders.Properties.Resources.ship3;
g.DrawImage(image, positionX, positionY, image.Width, image.Height);
```

Ecrivez la classe `SpaceShip` qui hérite de la classe `GameObject`. Cette classe possède les membres suivants:

Membre	Type	Nom	Description
Champ privé	double	speedPixelPerSecond	Vitesse de déplacement du joueur
Propriété	Vecteur2D	Position	Position du vaisseau
Propriété	int	Lives	Nombre de vie du vaisseau
Propriété	Bitmap	Image	Image représentant le vaisseau
Constructeur		SpaceShip	Permet d'initialiser la position, le nombr

Elle implémente également les méthodes abstraites de la classe `GameObject`:

Nom	Description
Update	Ne fait rien pour le moment
Draw	Dessine le vaisseau à sa position
IsAlive	Retourne vrai si le nombre vie du vaisseau est supérieur à 0

3) Création du vaisseau du joueur

Dans la classe `Game`:

- Ajouter un champ `playerShip` de type `SpaceShip`.
- Dans le constructeur, initialisez le champ `playerShip` avec un nouveau vaisseau avec 3 vies, centré en bas de l'écran au démarrage du jeu.
- Dans le constructeur, ajoutez le nouveau vaisseau à la liste des objets du jeu.

Lancez votre jeu et vérifiez que le vaisseau est correctement affiché.

4) Déplacement du joueur

Toute la gestion des déplacements du joueur sera faite dans la méthode `Update` de la classe `SpaceShip` :

- Modifiez la méthode `Update` de la classe `SpaceShip` pour qu'un appui sur la touche gauche (resp. droite) déplace le vaisseau sur la gauche (resp. droite) en respectant la vitesse définie dans le champ `playerSpeed`. Assurez-vous que le vaisseau ne puisse pas sortir de la zone de jeu (champs `gameSize` de la classe `Game`) !

Lancez votre jeu et testez.

5) La classe Missile

Nous allons maintenant passer à la création d'une classe `Missile` pour permettre au joueur de tirer.

Ecrivez la classe `Missile` qui hérite de la classe `GameObject`. Cette classe possède les membres suivants:

Membre	Type	Nom	Description
Propriété	Vecteur2D	Position	Position du missile
Propriété	double	Vitesse	Vitesse du missile
Propriété	int	Lives	Nombre de vie du missile
Propriété	Bitmap	Image	Image représentant le missile
Constructeur		Missile	Permet d'initialiser la position, le nombre de vies et l'i

Elle implémente également les méthodes abstraites de la classe `GameObject` :

Nom	Description
Update	Le missile se déplace verticalement selon sa vitesse. Si le missile sort de l'écran son nomb
Draw	Dessine le vaisseau à sa position
IsAlive	Retourne vrai si le nombre vie du missile est supérieur à 0

6) Tir du joueur

Lorsque le joueur appuie sur la touche espace, son vaisseau doit tirer un missile sauf si un missile a déjà été tiré et est toujours en vie.

- Ajoutez un champ `missile` de type `Missile` à la classe `SpaceShip`.
- Ajoutez une méthode `Shoot`. Si il n'y a actuellement pas de missile ou si le missile existant est mort, cette méthode
    - Crée un nouveau missile, positionner au milieu du vaisseau.
    - Affecte le nouveau missile au champ `missile`.
    - Ajoute le nouveau missile aux objets du jeu.
- Modifiez la méthode `Update` de la classe `SpaceShip`. Si le joueur appuie sur `Espace`, elle appelle la méthode `Shoot`.

Lancez votre jeu et testez.

7) Refactoring `SpaceShip` et `Missile`

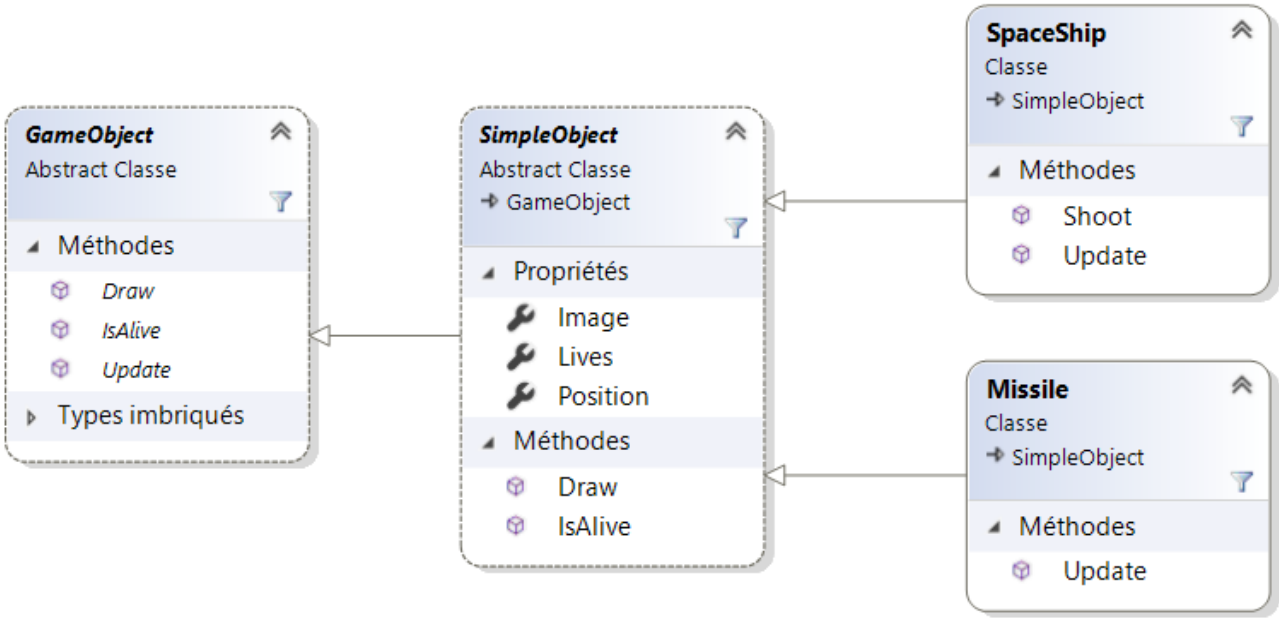
Les classes `SpaceShip` et `Missile` ont énormément de points communs : on retrouve beaucoup d'éléments dupliqués dans ces deux classes (le champs `image`, la propriété `Position`, la propriété `Lives`, la méthode `Draw`, la méthode `Alive`). Il est donc nécessaire de réorganiser notre code !

Nous allons extraire une classe mère commune à partir de ces 2 classes. Faute de mieux nous appelerons cette classe `SimpleObject`.

Créez un nouvelle classe abstraite `SimpleObject` qui hérite de `GameObject` :

- Ramenez le champs `Position`, `image`, la propriété `Lives`, la méthode `Draw` et la méthode `Alive` dans cette nouvelle classe.
- Supprimez ces membres des classes `SpaceShip` et `Missile`.
- Les classes `SpaceShip` et `Missile` héritent maintenant de `SimpleObject`.

Voilà la nouvelle architecture du jeu:



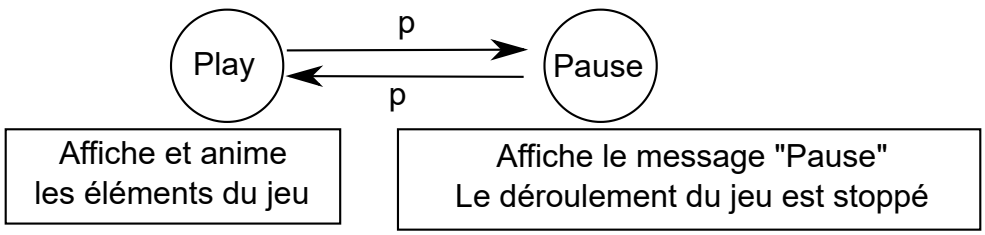
Lancez votre jeu et testez : il ne doit pas y avoir de régression (tout ce qui fonctionnait doit continuer à fonctionner après le refactoring) !

8) Gestion de l'option Pause

La touche `p` doit permettre de mettre le jeu en pause puis de revenir dans l'état initial. Un jeu vidéo est un exemple classique de machine à états :

- Il peut être dans différents états qui conduiront à des comportements différents (en jeu, en pause, menu de départ, ...).
- Certains évènements font passer d'un état à l'autre.

On va représenter les différents états du jeu par une énumération.



Travail de programmation dans la classe `Game` :

- Déclarez une énumération `GameState` contenant les valeurs `Play` et `Pause` (d'autres états seront ajoutés par la suite)
- Ajouter un champ `state` de type `GameState` qui représentera l'état courant.
- Modifiez la fonction `Draw` de manière à afficher le texte *Pause* si le jeu est en pause et le jeu en cours dans l'état `Play`.
- Modifiez la fonction `Update` de manière à gérer ce nouvel état (la méthode `ReleaseKey` de la classe `Game` peut vous aider) :

- Si le jeu est dans l'état `Play` et que le joueur appuie sur la touche `p`, il doit passer en état pause.
- Si le jeu est dans l'état `Pause` et que le joueur appuie sur la touche `p`, le jeu doit passer dans l'état `Play`.

9) La classe Bunker

Les bunkers permettent au joueur de se mettre à couvert mais peuvent être détruit par les missiles. Les bunkers ne se déplacent pas.

Ecrivez la classe `Bunker` qui hérite de `SimpleObject`. Les bunkers sont des objets très simples et ne font donc pas grand chose; les membres à définir sont :

- Le constructeur qui ne prend qu'un seul paramètre : la position du bunker
- La redéfinition de la méthode `Update` qui ne fait rien.

Modifiez le constructeur de la classe `Game` :

- Créez 3 bunker régulièrement espacés dans le bas de la fenêtre.
- Ajoutez les bunker à la liste des objets du jeu.

Lancez votre programme et vérifiez que les bunkers s'affichent.

10) Collisions - Mise en place

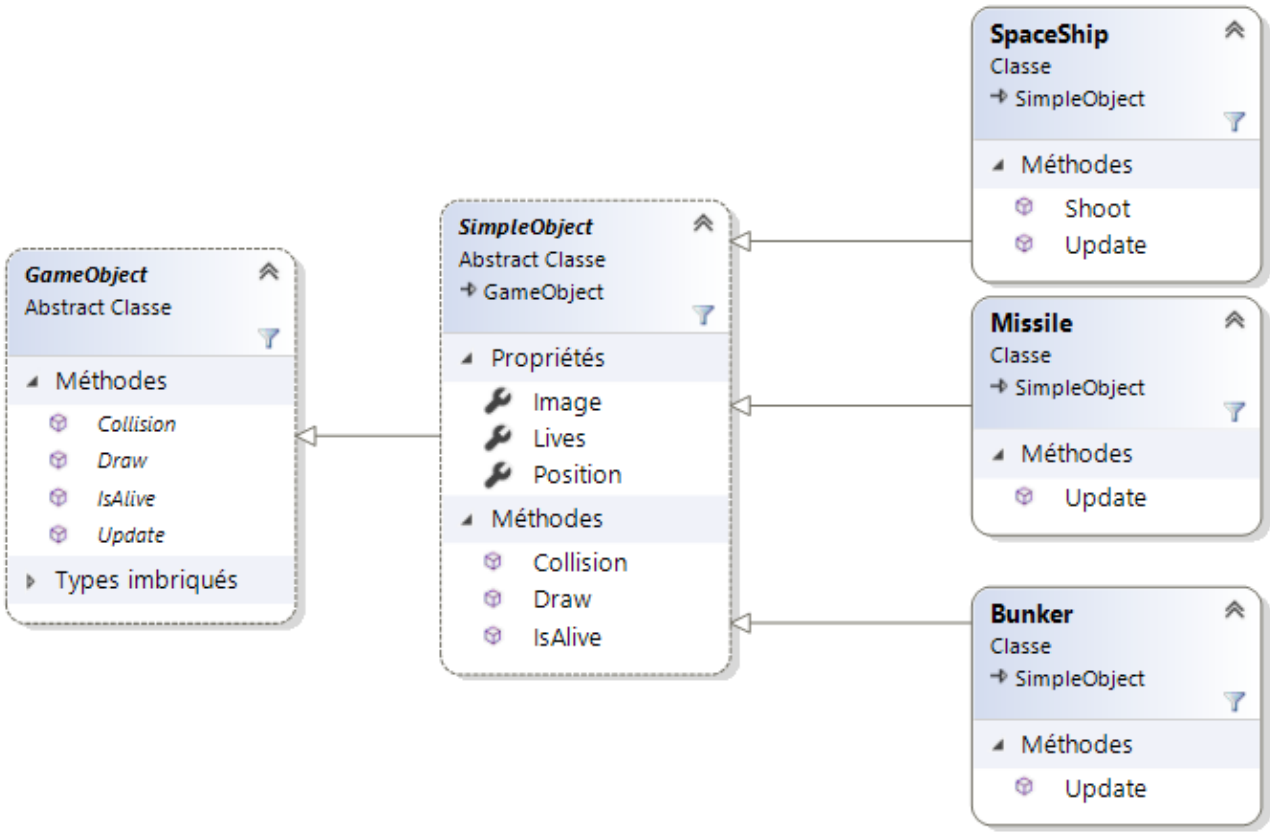
Nous pouvons maintant tirer des missiles, mais ceux ci traversent les bunkers. Afin de gérer les collisions entre les missiles et les autres objets, nous allons ajouter une nouvelle méthode abstraite dans la classe `GameObjet` :

```
public abstract void Collision(Missile m);
```

Chaque missile va ensuite tester si il est en collision avec les autres objets du jeu. Dans la méthode `Update` de la classe `Missile` :

- Parcourez l'ensemble des objets du jeux (liste `gameInstance.gameObjects`).
- Appelez la méthode `Collision` sur ces objets.

Pour le moment, implémentez la nouvelle méthode abstraite dans la classe `SimpleObject` avec une méthode qui ne fait rien. On obtient la structure suivante:



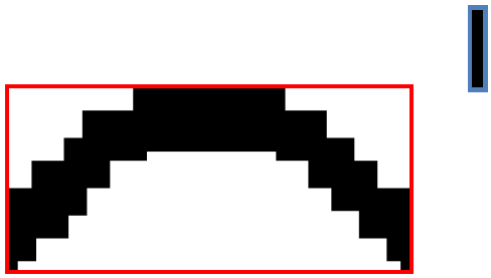
Lancez votre programme et vérifiez qu'il n'y a pas de régression.

11) Collisions Missile/Bunker

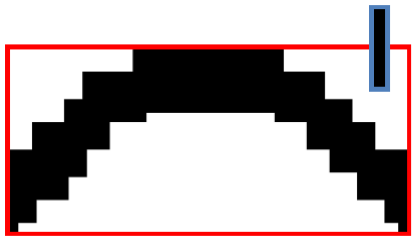
Le test de collision entre 2 objets (missile/bunker, missile/vaisseau ou missile/missile) fait intervenir le calcul de l'intersection entre deux sprites situés à des positions différentes. Pour des raisons de performances, le test est effectué en 2 temps. On commence par tester si le rectangle englobant du bunker/vaisseau intersecte le rectangle englobant du missile. Si ce n'est pas le cas, on sait qu'il n'y a pas de collision possible, sinon il faut tester plus précisément.

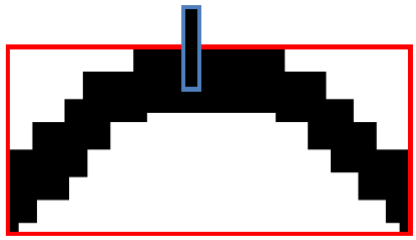
Test des rectangles englobants

- les rectangles englobants sont disjoints; aucune collision possible:

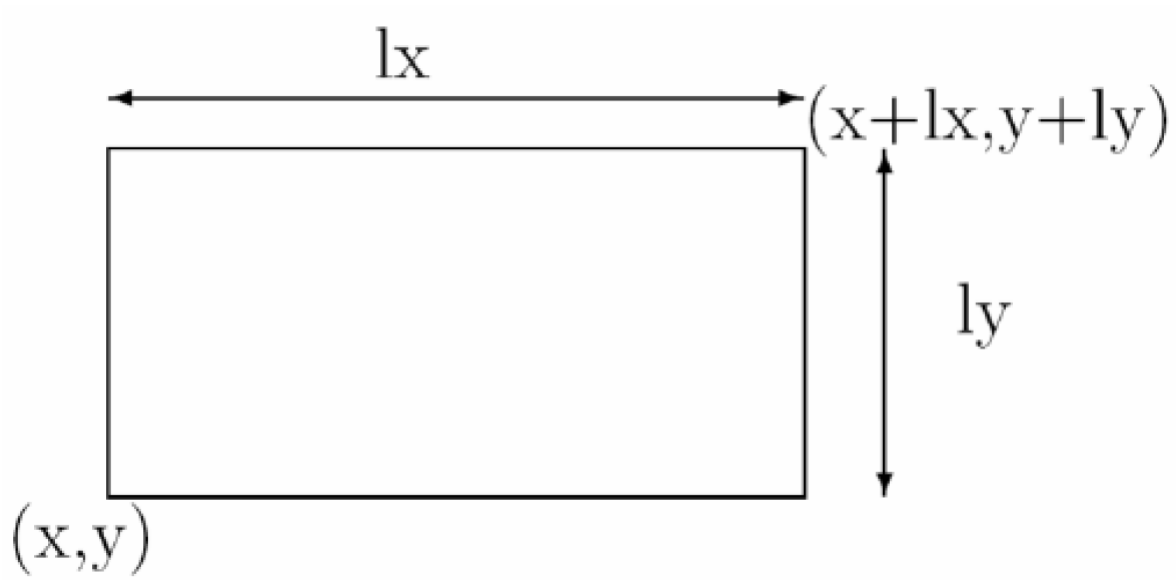


- les rectangles englobant s'intersectent; on ne peut pas conclure:





Pour tester si deux rectangles s'intersectent, on considère qu'un rectangle est paramétré de la façon suivante:



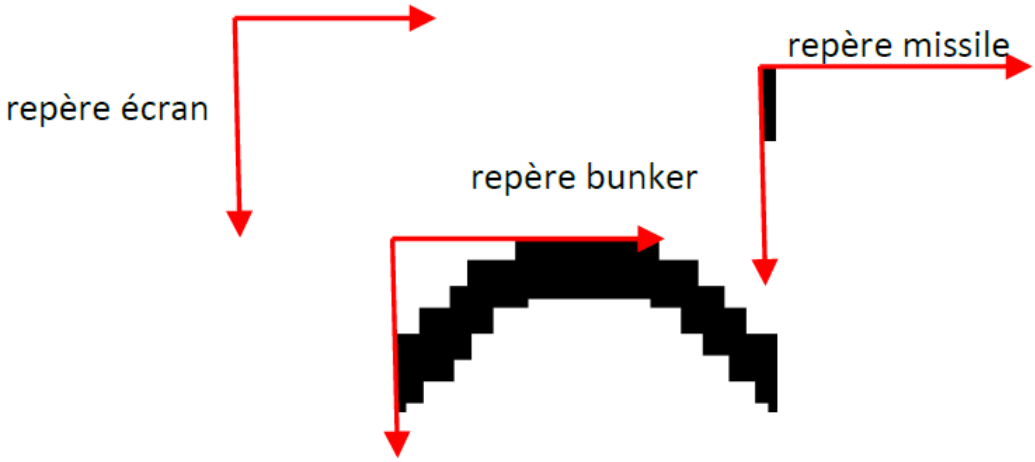
Si on dispose de deux rectangles (x_1, y_1, lx_1, ly_1) et (x_2, y_2, lx_2, ly_2), les deux rectangles sont disjoints si

- $x_2 > x_1 + lx_1$: le deuxième rectangle est à droite du premier **OU**
- $y_2 > y_1 + ly_1$: le deuxième rectangle est haut dessus du premier **OU**
- $x_1 > x_2 + lx_2$: le premier rectangle est à droite du deuxième **OU**
- $y_1 > y_2 + ly_2$: le premier rectangle est haut dessus du deuxième.

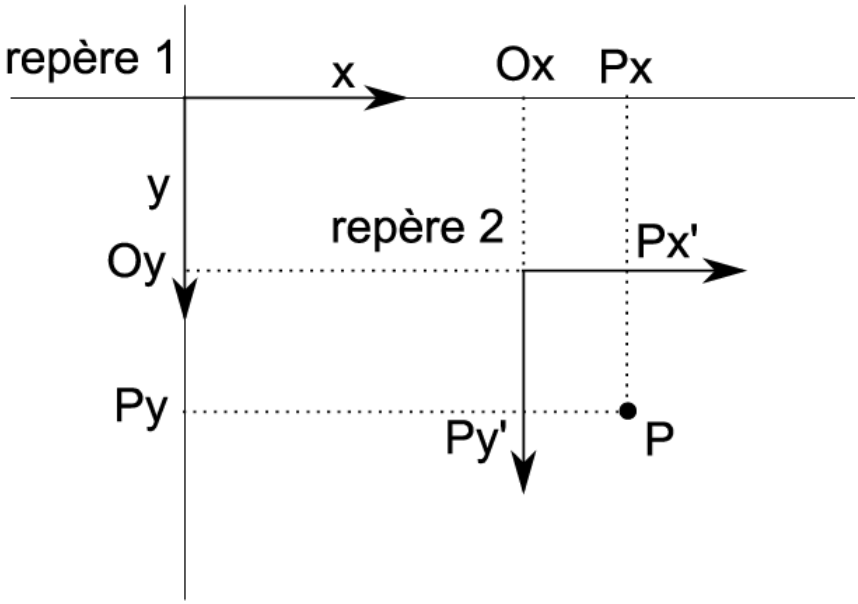
Test pixel à pixel

Si les rectangles englobant s'intersectent, il faut tester pixel par pixel si il y a une intersection. On parcourt l'ensemble des pixels du missile et on calcule la position correspondante sur l'autre objet (on connait la position du missile et de l'autre objet par rapport au coin supérieur

gauche de la fenêtre, et on connait les coordonnées du pixel par rapport à la position du missile, on peut donc en déduire les coordonnées du pixel par rapport à la position de l'autre objet). Si le pixel de l'image de l'autre objet qui correspond est noir, il y a collision au niveau de ce pixel.



Le changement de repère, lorsque les axes sont alignés, est une opération simple. Imaginons que l'on dispose de 2 repères. On connait la position du deuxième repère par rapport au premier (O_x, O_y). On connait la position d'un point P dans le second repère $(P_{x'}, P_{y'})$ et on souhaite obtenir les coordonnées de P dans le premier repère (P_x, P_y). On a directement $P_x = P_{x'} + O_x$ et $P_y = P_{y'} + O_y$.



Dans le cas qui nous intéresse il sera nécessaire d'effectuer deux changements de repères consécutifs: 1) du repère missile vers le repère écran, et 2) du repère écran vers le repère de l'autre objet.

Attention

Les images fournies avec le projet possèdent une couche *alpha* pour gérer la transparence. Les pixels noirs de l'image correspondent à la couleur (255, 0, 0, 0) : du noir (0 pour les 3 composantes rouge, verte et bleue) opaque (255 sur la composante alpha)

alors que les pixels en dehors du bunker sont représentés par la couleur (0, 255, 255, 255) : du blanc (255 pour les 3 composantes rouge, verte et bleue) transparent (0 sur la composante alpha).

Codage

Dans la classe `Bunker`, redéfinissez la méthode `Collision` avec l'algorithme décrit ci-dessus. Cette méthode doit:

- Effacer les pixels du bunker qui sont en collision avec le missile.
- Compter le nombre de pixels en collision et retirer ce nombre du nombre de vie du missile.

Cette fonction est complexe, il est donc nécessaire de :

- **Créer des sous-fonctions**
- **Définir des variables avec des noms explicites à chaque étape des calculs**

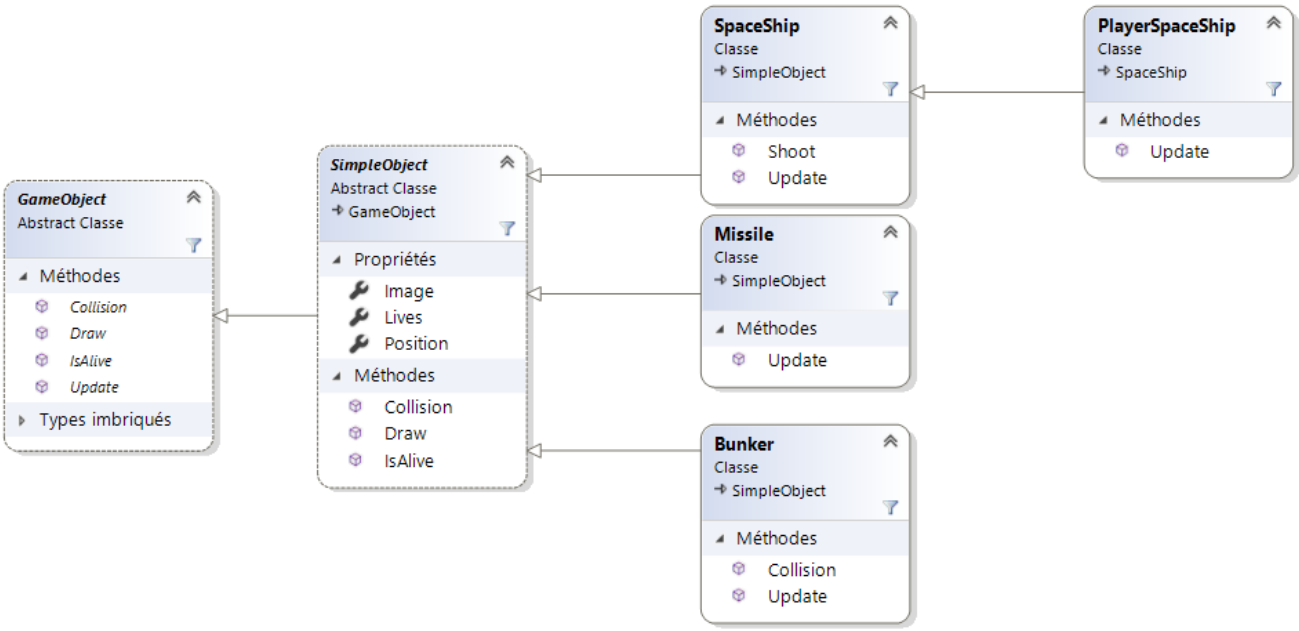
12) Vaisseaux ennemis

Les vaisseaux ennemis sont en fait similaire au vaisseau du joueur à une différence prêt : ils ne se déplacent pas lorsque le joueur appuis sur les flêche du clavier (méthode `Update`).

Afin de modéliser cela:

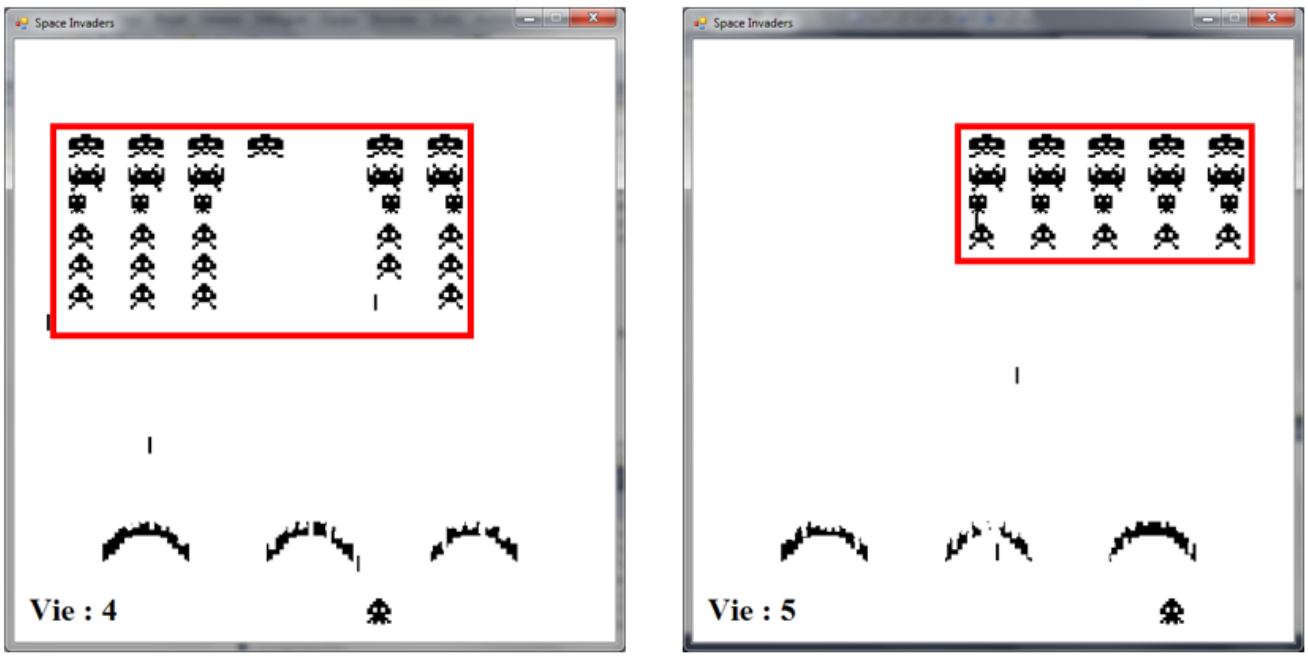
- Créez une sous classe `PlayerSpaceship` de la classe `SpaceShip`.
- Déplacez la méthode `Update` de la classe `SpaceShip` (celle avec la gestion des flêches du clavier) dans la classe `PlayerSpaceship`.
- Définissez une méthode `Update` ne faisant rien dans la classe `SpaceShip` (par défaut le vaisseau ennemi est immobile).
- Adaptez le code de la classe `Game` pour créer un `PlayerSpaceship` à la place d'un `SpaceShip`.

On obtient le diagramme de classe:



13) Bloc d'ennemis - Création

Les ennemis de Space Invaders ont un comportement assez particulier : ils arrivent par ligne de vaisseaux identiques et ils se déplacent de manière synchronisée. Ainsi, dès qu'un vaisseau atteint le bord de l'écran, c'est l'ensemble des ennemis qui descendent et inversent leur direction de déplacement horizontale. En fait on peut observer que c'est le rectangle englobant de l'ensemble des vaisseaux qui importe.



En rouge : rectangle englobant du bloc d'ennemis.

Afin de gérer cette situation, nous allons créer une nouvelle classe `EnemyBlock` qui va gérer un groupe d'ennemis. Cette classe représente un objet du jeu mais ne correspond pas à un objet simple (elle n'est pas représentée par un sprite) : elle héritera donc de `GameObject`.

Ecrivez la classe `EnemyBlock` qui hérite de la classe `GameObject`. Cette classe possède les membres suivants:

Membre	Type	Nom	Descr
Champ privé	`HashSet<SpaceShip>`	`enemyShips`	Ensen
Champ privé	`int`	`baseWidth`	Large
Propriété	`Size`	`size`	Taille
Propriété	`Vecteur2D`	`Position`	Positi
Constructeur		`EnnemyBloc`	Perme
Méthode	`void AddLine(int nbShips, int nbLives, Bitmap shipImage)`	`AddLine`	Ajout
Méthode	`void UpdateSize()`	`UpdateSize`	Recalc

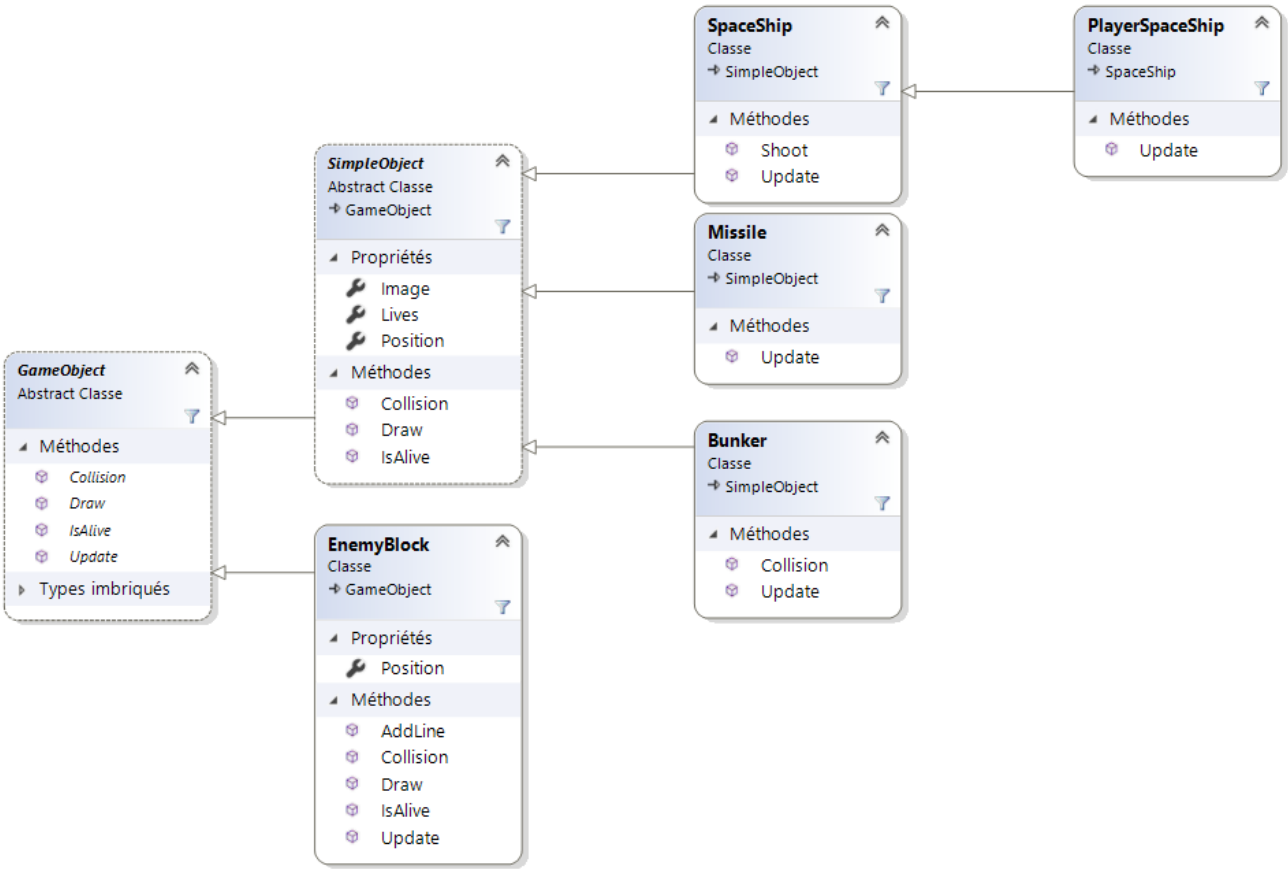
Elle implémente également les méthodes abstraites de la classe `GameObject`:

Nom	Description
Update	Ne fait rien pour le moment
Draw	Dessine les vaisseaux du bloc
IsAlive	Retourne vrai si il y a au moins 1 vaisseau vivant dans le bloc

La méthode `AddLine` doit ajouter une nouvelle ligne d'ennemis au bloc. La nouvelle ligne contient `nbShips` vaisseaux avec `nbLives` vies représentés par l'image `shipImage`. Les nouveaux vaisseaux doivent être répartis de manière homogène sur la ligne de largeur `baseWidth`. Par exemple, après ajout d'une ligne de 9 ennemis on obtient (le rectangle représentant la taille du bloc est dessiné en rouge):

Après ajout d'une deuxième ligne de 5 ennemis:

La nouvelle structure de classe est la suivante (notez qu'on pourrait factoriser la propriété `Position` dans la classe `GameObject`):



14) Bloc d'ennemis - Intégration

Dans la classe `Game` :

- Ajoutez un champ privé `enemies` de type `EnemyBlock`,
- Initialisez le champs dans le constructeur,
- Ajoutez des lignes d'ennemis, et
- Ajoutez le bloc d'ennemis aux objets du jeu.

Lancez votre jeu et vérifiez que le bloc d'ennemis est correctement construit et affiché.

15) Bloc d'ennemis - Déplacement

Modifiez la méthode `Update` de la classe `EnemyBlock`. Cette méthode doit gérer le déplacement du bloc et de tous les vaisseaux qu'il contient. De base le bloc se déplace horizontalement. Lorsque le bloc arrive au bord de la zone de jeu (à gauche ou à droite), il faut:

1. Inverser le sens de déplacement horizontal.
2. Décaler le bloc vers le bas.
3. Augmenter la vitesse de déplacement horizontale.

Lancez votre jeu et vérifiez le déplacement des ennemis.

16) Bloc d'ennemis - Destruction

Actuellement la classe `SpaceShip` ne gère pas les collisions : les vaisseaux ennemis sont indestructibles. La gestion des collisions missiles/vaisseaux est très proche de la gestion des collisions missiles/bunkers. Dans les 2 case, on doit vérifier si il y a des pixels du missile en intersection avec des pixels de l'autre objet. Néanmoins dans le case missile/bunker, le nombre de vie du missile est décrémenté du nombre de pixels en collision alors que dans le cas missile/vaisseau, il faut décrémenter les nombres de vies du missile et du vaisseau.

Cela suggère de factoriser le code de détection des pixels en collision dans la classe `SimpleObject` et de déporter l'action à réaliser en cas de collision dans les sous-classes. Il faut alors également vérifier que les collisions missile/missile entrent également dans ce schéma. C'est bien le cas, si il y a collision entre 2 missiles, les 2 missiles sont détruits (quelquesoit leurs nombres de vies).

Effectuez le refactoring suivant:

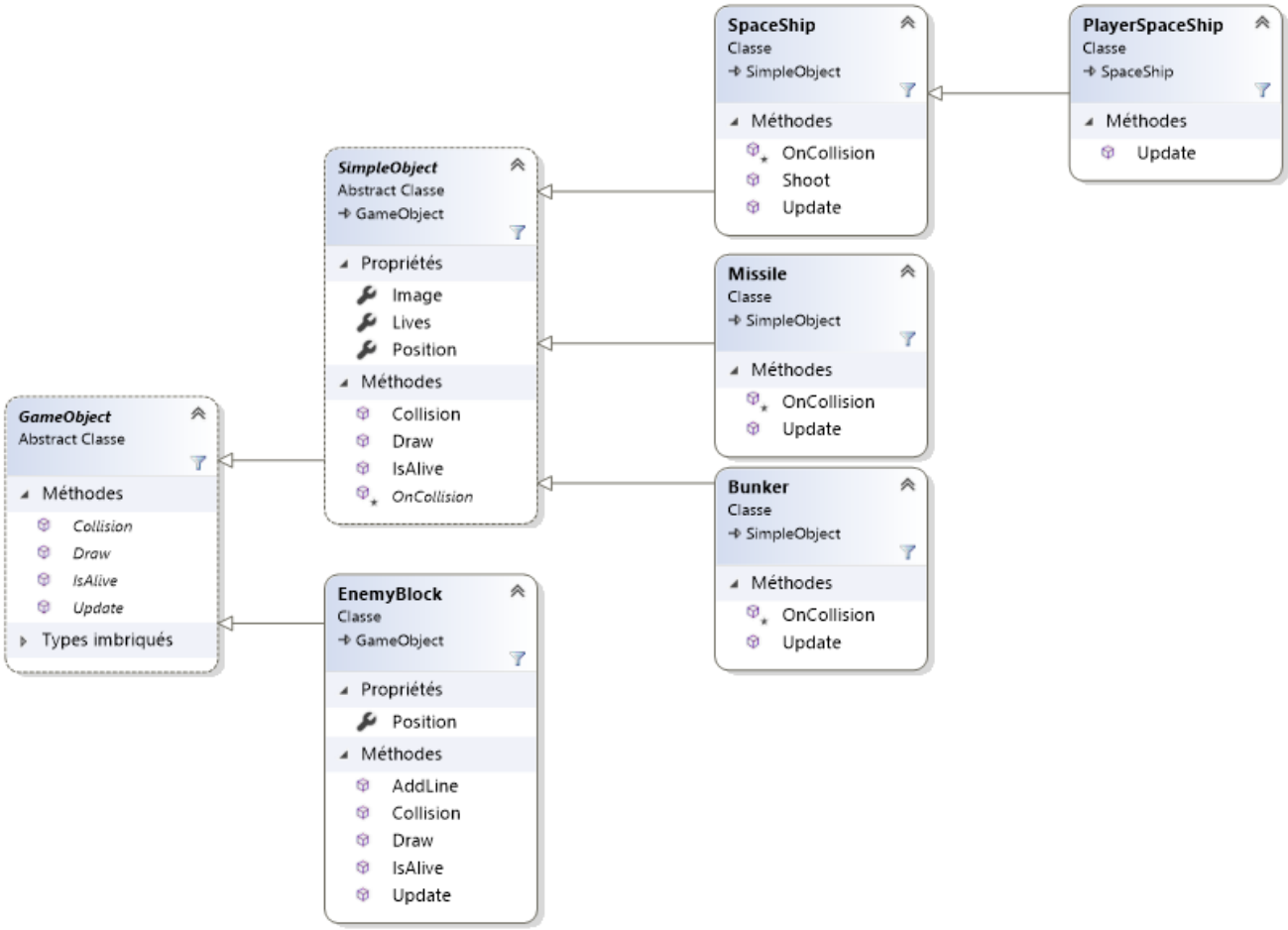
1. Ramenez le code de la méthode `Collision` de la classe `Bunker` dans la classe `SimpleObject`.
2. Afin d'implémentez les différents comportement en cas de collision, ajoutez une méthode abstraite `OnCollision` dans la classe `SimpleObject` :

```
protected abstract void OnCollision(Missile m, int numberOfPixelsInCollision);
```

3. Adaptez la nouvelle méthode `Collision` de la classe `SimpleObject` pour appeler la méthode `OnCollision` en cas de collision.
4. Implémentez la méthode `OnCollision` dans les classes `Bunker`, `SpaceShip` et `Missile` :
    - Dans la classe `Bunker` : le nombre de vie du missile est décrémenté du nombre de pixels en collision
    - Dans la classe `Missile` : le nombre de vie des deux missiles est mis à zéro (les missiles se détruisent mutuellement).
    - Dans la classe `SpaceShip` : les nombres de vies du missile et du vaisseau sont décrémentés de la valeur minimum entre le nombre de vies du missile et du vaisseau.

Il ne reste plus qu'à implémenter la méthode `Collision` de la classe `EnnemyBlock` : tester si un missile est en collision avec un bloc d'ennemis implique simplement de tester si le missile est en collision avec chacun des vaisseaux du bloc.

On obtient la nouvelle structure:



Testez et vérifiez:

- qu'il n'y a pas de régression (la destruction des bunkers fonctionne toujours),
- que la destruction des vaisseaux ennemis fonctionne,
- que le nombre de vies des vaisseaux et des missiles sont bien pris en compte,
- que le déplacement du bloc d'ennemis s'adapte bien à la destruction des vaisseaux ennemis.

17) Friendly Fire

Dans Space Invaders, un tir ennemi peut endommager un bunker ou le vaisseau du joueur mais pas un autre vaisseau ennemi : dit autrement il n'y a pas de *friendly fire*.

Cela signifie que l'on doit pouvoir différencier :

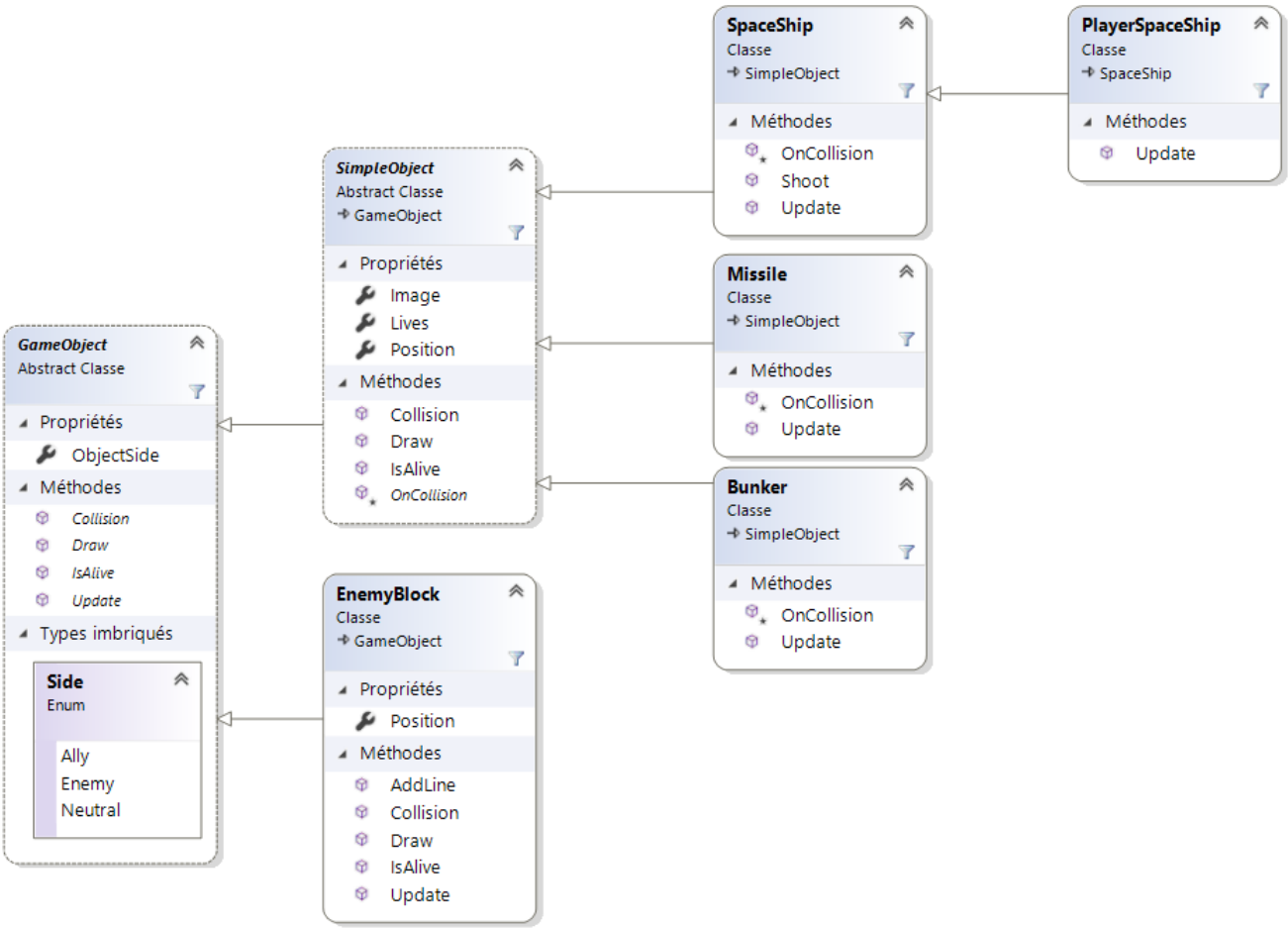
- le camp *allié* composé du vaisseau du joueur et des missiles du joueur,
- le camp *ennemi* composé des vaisseaux et des missilles des ennemis,
- le camp *neutre* composé des bunkers.

Les missiles ne pourront entrer en collision qu'avec un objet d'un camp différent.

Afin d'implémenter cela:

- Dans la classe `GameObject`, ajoutez une énumération `Side` composé des trois valeurs `Ally`, `Enemy` et `Neutral`.
- Dans la classe `GameObject`, ajoutez une propriété de type `Side`.
- Ajoutez un paramètre de type `Side` au constructeur de `GameObject` afin d'initialiser le camp d'un objet.
- Ajustez les constructeurs des classes descendantes de `GameObject` pour prendre en compte cette modification.
- Modifiez la fonction `Collision` de la classe `SimpleObject` pour ignorer les collisions entre 2 entités d'un même camp.

On obtient la nouvelle structure:



Testez votre jeu et vérifiez qu'il n'y a pas de régression.

18) Bloc d'ennemis - Tirs

Nous allons maintenant pouvoir faire tirer les ennemis. Les ennemis tirent aléatoirement et, plus le bloc de vaisseau se rapproche du joueur, plus la fréquence de tir est élevée.

Dans la classe `EnnemyBlock`:

- Ajoutez un champ privé `randomShootProbability` de type `double` à la classe `EnemyBlock` qui représente la probabilité qu'un vaisseau ennemi produise un tir en 1 seconde.

- Modifiez la fonction `Update` de manière à appeler aléatoirement la fonction `Shoot` des vaisseaux. Etant donné un nombre aléatoire `r` entre 0 et 1 (fonction NextDouble de la classe `Random`) et le temps passé `deltaT` depuis le dernier update, on peut effectuer le test suivant pour savoir si un objet doit tirer:

```
r <= randomShootProbability * deltaT
```

- Augmentez `randomShootProbability` lorsque le bloc d'ennemis descend.

Attention : les missiles des ennemis doivent partir vers le bas !

Testez votre jeu et vérifiez que les tirs ennemis :

- sont aléatoires,
- détruisent les bunkers,
- détruisent les missiles du joueur,
- enlève des vies au joueur.

19) Affichage des vies

Etendez la méthode `Draw` dans la classe `PlayerSpaceShip` afin d'afficher le nombre de vies du joueur en plus de son sprite : **utilisez le chainage de méthode**.

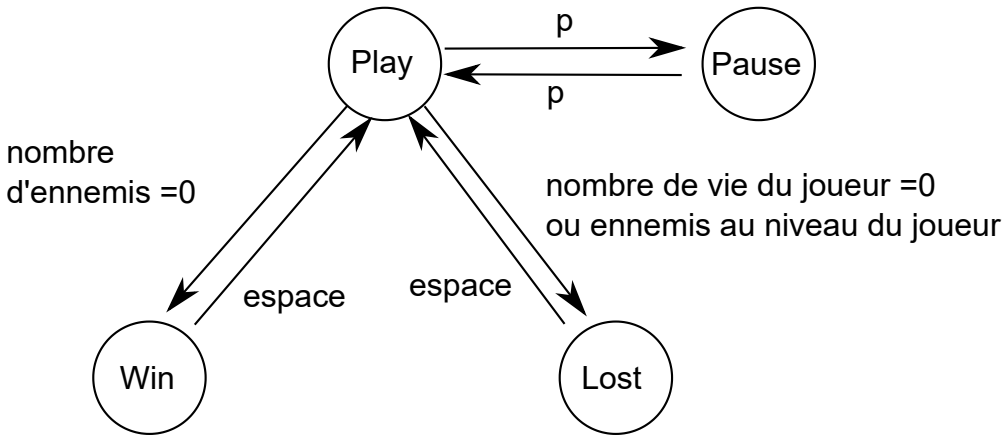
20) Gagné perdu

Toutes les mécaniques du jeu sont maintenant en place, il ne reste plus qu'à finaliser la machine à état pour gérer la victoire ou la défaite du joueur:

- Ajoutez les états `Win` et `Lost` à l'énumération `GameState`.
- Si le bloc d'ennemis arrive au niveau du joueur, alors le joueur perd toutes ses vies.
- Adaptez la méthode `Update` de la classe `Game` pour prendre en compte ces nouveaux états:
    - En phase de jeu, si le joueur est mort alors le jeu passe dans l'état perdu,
    - En phase de jeu, si le bloc d'ennemis est mort alors le jeu passe dans l'état gagné,
    - En phase gagné ou perdu, si le joueur appuie sur espace alors je jeu recommence.

- Adaptez la méthode `Draw` de la classe `Game` pour réaliser des affichages pertinents dans les nouveaux états.

La machine à état complète peut être représentée de la manière suivante :



21) DLC

Ajoutez les fonctions que vous voulez pour enrichir votre jeu !