Tutoriel pour « PhyProject » avec l'exemple « BubbleDemo »

Contenu

2012-10-22

1 L'approche de « PhyProject »

En règle générale, les moteurs physiques sont destinés à être utilisés par des jeux nécessitant des calculs physiques réalistes. L'approche prise par « PhyProject » va à contre-courant. Il est destiné aux jeux nécessitant une physique spécifique. Typiquement ces jeux sont réalisés avec un moteur physique fait maison. « PhyProject » fournit une architecture modulaire qui permet aux développeurs de définir les calculs physiques spécifiques au monde de leur jeu. L'intérêt de « PhyProject » est de permettre de partager ces calculs afin qu'ils soient intégrés à d'autres jeux.

2 L'exemple « BubbleDemo »

2.1 Un jeu simpliste

Le but du jeu est d'appuyer sur tous les ronds rouges. Un appui sur un rond vert fait perdre la partie. Les ronds ne révèlent leur véritable couleur uniquement lorsqu'ils se cognent entre eux.

Ce mini-jeu permet de montrer une grande partie des fonctionnalités du moteur tout en étant relativement simple.

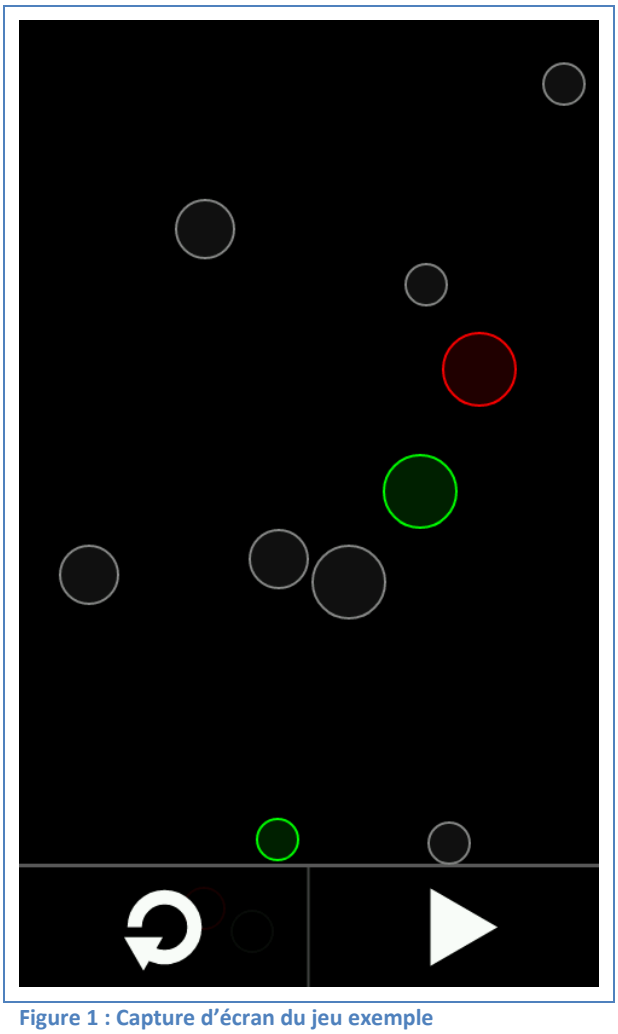

Figure 1 : Capture d'écran du jeu exemple

2.2 Lancer le jeu sur un PC

Dans le cas où vous n'auriez pas d'appareil fonctionnant sous Android, il peut être intéressant de se tourner vers une machine virtuelle. L'émulateur fourni avec « l'Android SDK Tools » est quasiment inutilisable pour tester des jeux. Un logiciel de virtualisation et un portage d'Android pour les processeurs de la famille x86 donnent un résultat correct même sur des machines peu puissantes.

2.3 Basé sur le patron MVC

Le jeu se base sur le patron MVC où le modèle est la classe « GameModel », la vue « GameView » et le contrôleur « GameActivity ». Le contrôleur reçoit des signaux du modèle concernant la fin du jeu (« GameListener ») ainsi que le calcul d'un nouvel état du monde physique (« WorldListener »).

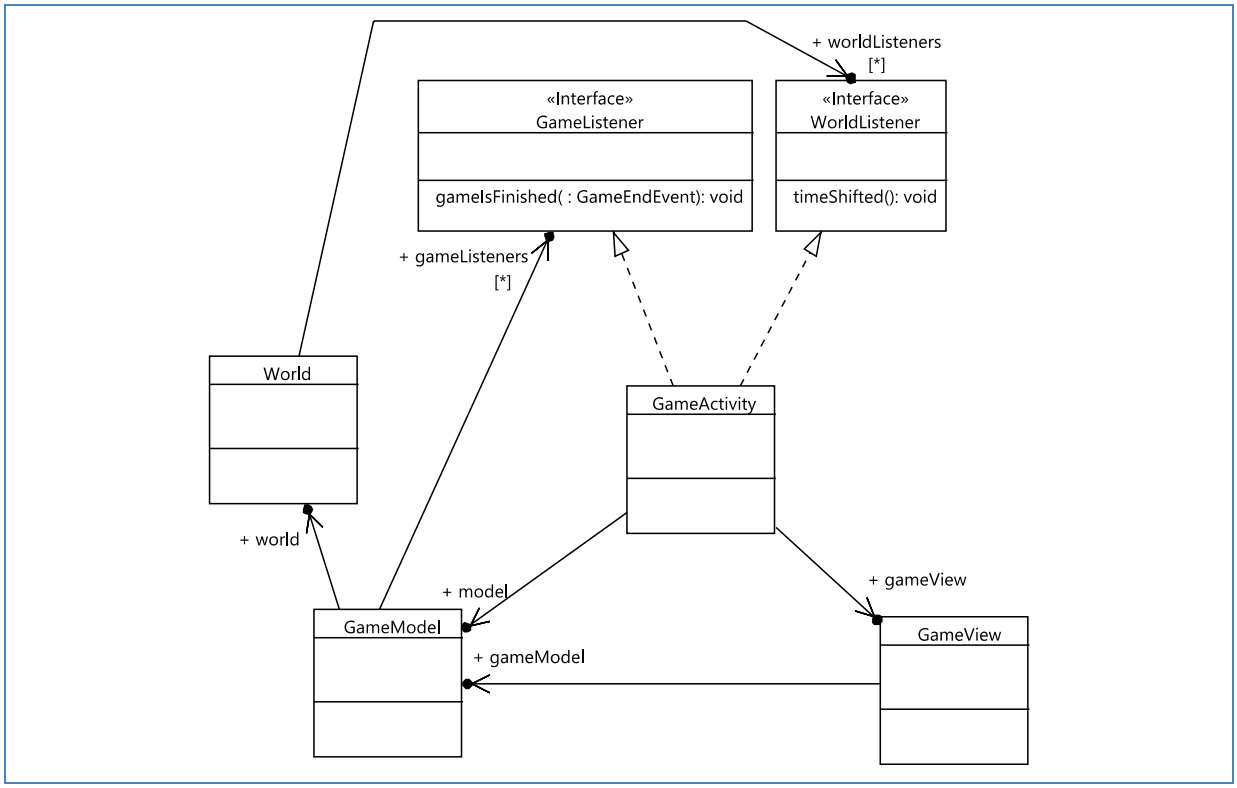

Figure 2 : Diagramme de classes du jeu exemple

3 La vue d'ensemble du moteur physique

Ce moteur physique n'a pas pour objectif de fournir des calculs prédéfinis, mais plutôt l'infrastructure autour de ces calculs. Le lien entre le moteur physique et les calculs ajoutés par les développeurs se fait par injection de dépendance. Comme on le voit sur le diagramme ci-dessous, « World », la classe qui orchestre les calculs physiques utilise des interfaces. Ces interfaces sont implémentées par les développeurs de jeux. Ils y définissent les calculs spécifiques à leur jeu et injectent ces implémentations par des manipulateurs. Nous reviendrons plus précisément sur les différentes injections avec des exemples dans la partie suivante.

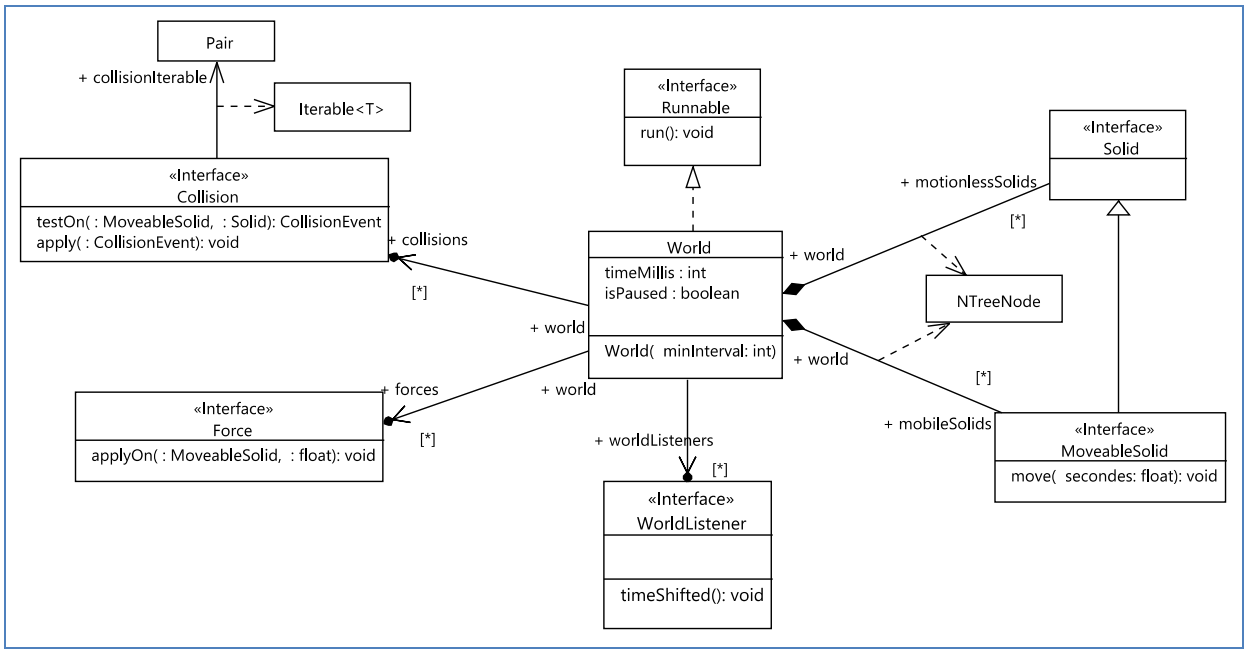

Figure 3 : Diagramme de classes du cœur du moteur physique

4 Comment faire pour…

4.1 Délimiter les sous-ensembles de solides

Les solides contenus dans le monde sont organisés en sous-ensembles, sous-ensemble de sous-ensemble… Les sous-ensembles sont représentés par un arbre N-aire (« NTreeNode »). Le monde définit deux sous-ensembles : celui des solides immobiles (« motionlessSolids ») et celui des solides mobiles (« mobileSolids »).

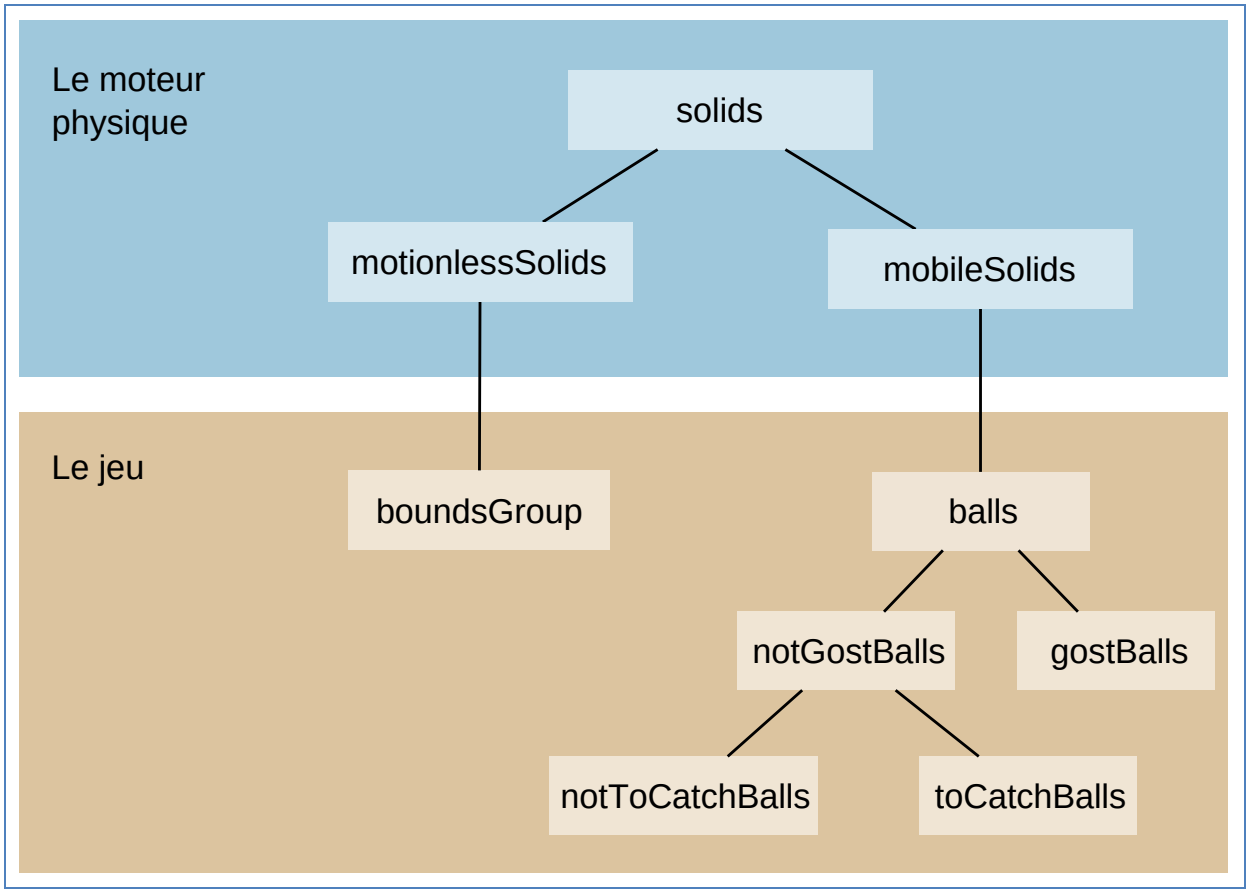

Figure 4 : Arborescence des solides de l'exemple

On commence par récupérer le sous-ensemble des solides immobiles pour y ajouter un sous-ensemble contenant une bordure rectangulaire correspondant à l'écran. Ce solide permettra de faire rebondir les balles à l'intérieur de l'écran. En parlant des balles, elles sont ajoutées au sous-ensemble des solides mobiles. Le sous-ensemble des balles est décomposé en plusieurs sous-ensembles :

- les balles fantômes qui ne rebondissent pas sur les autres balles et restent toujours grises (« ghostBalls »)
- les balles devant être attrapées (« toCatchBalls »)
- les balles ne devant pas être attrapées (« notToCatchBalls »)

On remarque que les collections de solides utilisées par le modèle du jeu sont les mêmes que celles utilisées par le moteur physique. Il n'y a donc pas à s'inquiéter de conserver la consistance entre les collections manipulées par le moteur et le modèle.

```
ArrayList<BoundsSolid> boundsGroup;
NTreeNode<GameBall> balls;
NTreeNode<GameBall> notGhostBalls;

NTreeNode<Solid> motionlessSolids = world.getMotionlessSolids();
{
	boundsGroup = new ArrayList<BoundsSolid>();
	{
		BoundsSolid boundsSolid = new BoundsSolid(world, bounds);
		boundsGroup.add(boundsSolid);
	}
	motionlessSolids.getChildren().add(boundsGroup);
}
NTreeNode<MoveableSolid> mobileSolids = world.getMobileSolids();
{
	balls = new NTreeNode<GameBall>();
	{
		ghostBalls    = new ArrayList<GameBall>();
		notGhostBalls = new NTreeNode<GameBall>();
		{
			toCatchBalls    = new ArrayList<GameBall>();
			notToCatchBalls = new ArrayList<GameBall>();

			notGhostBalls.getChildren().add(toCatchBalls);
			notGhostBalls.getChildren().add(notToCatchBalls);
		}
		balls.getChildren().add(ghostBalls);
		balls.getChildren().add(notGhostBalls);
	}
	mobileSolids.getChildren().add(balls);
}
```

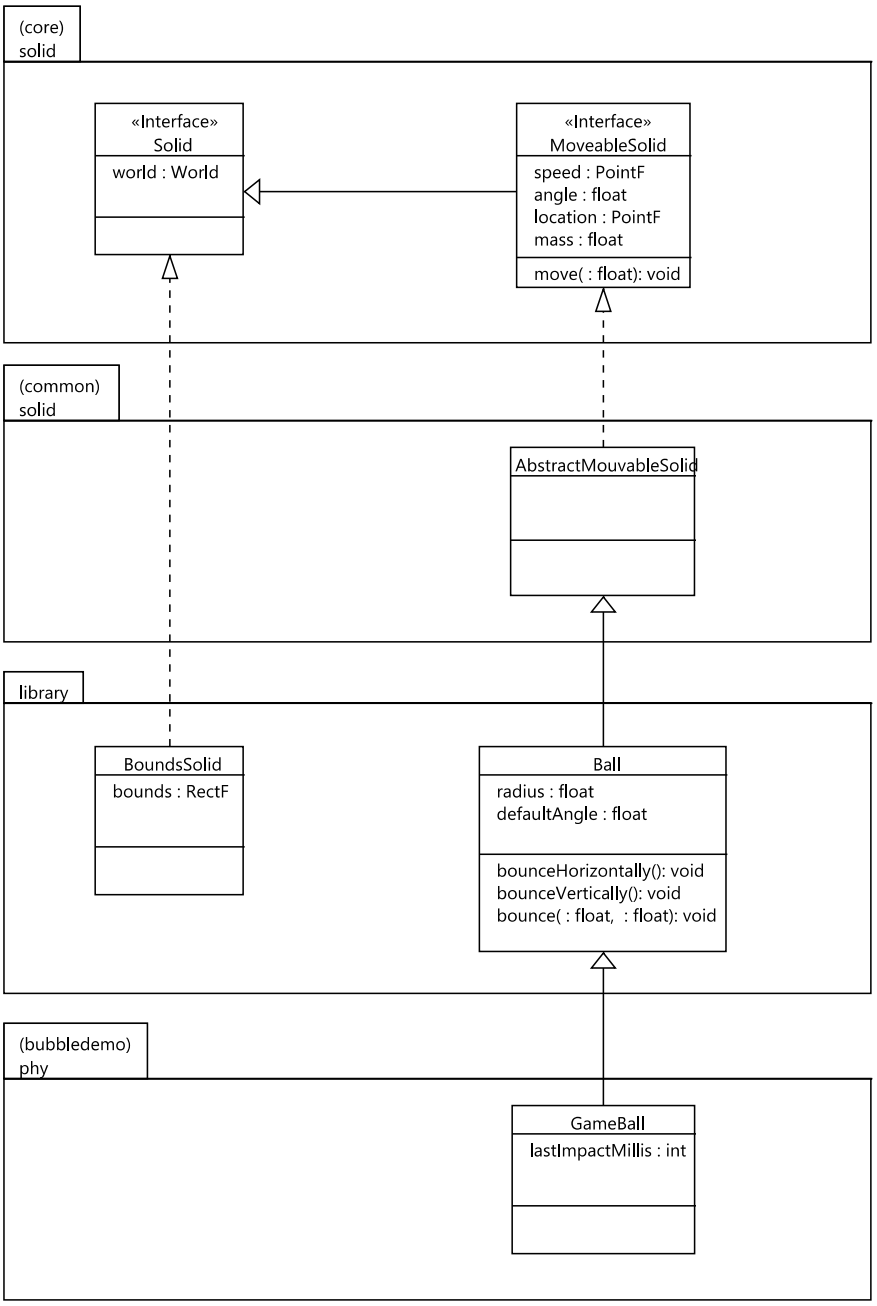

Figure 5 : Diagramme de classes de l'arborescence des types de solides

4.2 Ajouter une force de frottement

Une force est appliquée à un « Iterable » de solides mobiles, ici toutes les balles. Dans notre cas, on ajoute une force de frottement pour réduire progressivement la vitesse des balles.

```
world.addForceDefinition(new FrictionForce(0.0625f), balls);
```

4.3 Définir les collisions entre les solides

Implémenter l'interface « Collision » permet de définir un nouveau comportement de collision entre deux types de solides. Pour cet exemple, on utilise deux implémentations : l'une pour gérer les collisions entre les balles et le bord de l'écran et l'autre pour les collisions entre les balles elles-mêmes. Ces implémentations peuvent être utilisées dans d'autres projets utilisant le même genre de solides. Une bibliothèque de collisions pourrait se constituer au fur et à mesure des projets réalisés. Il serait alors possible de choisir les comportements de collision qui nous conviennent et assembler ainsi le moteur physique d'un nouveau jeu.

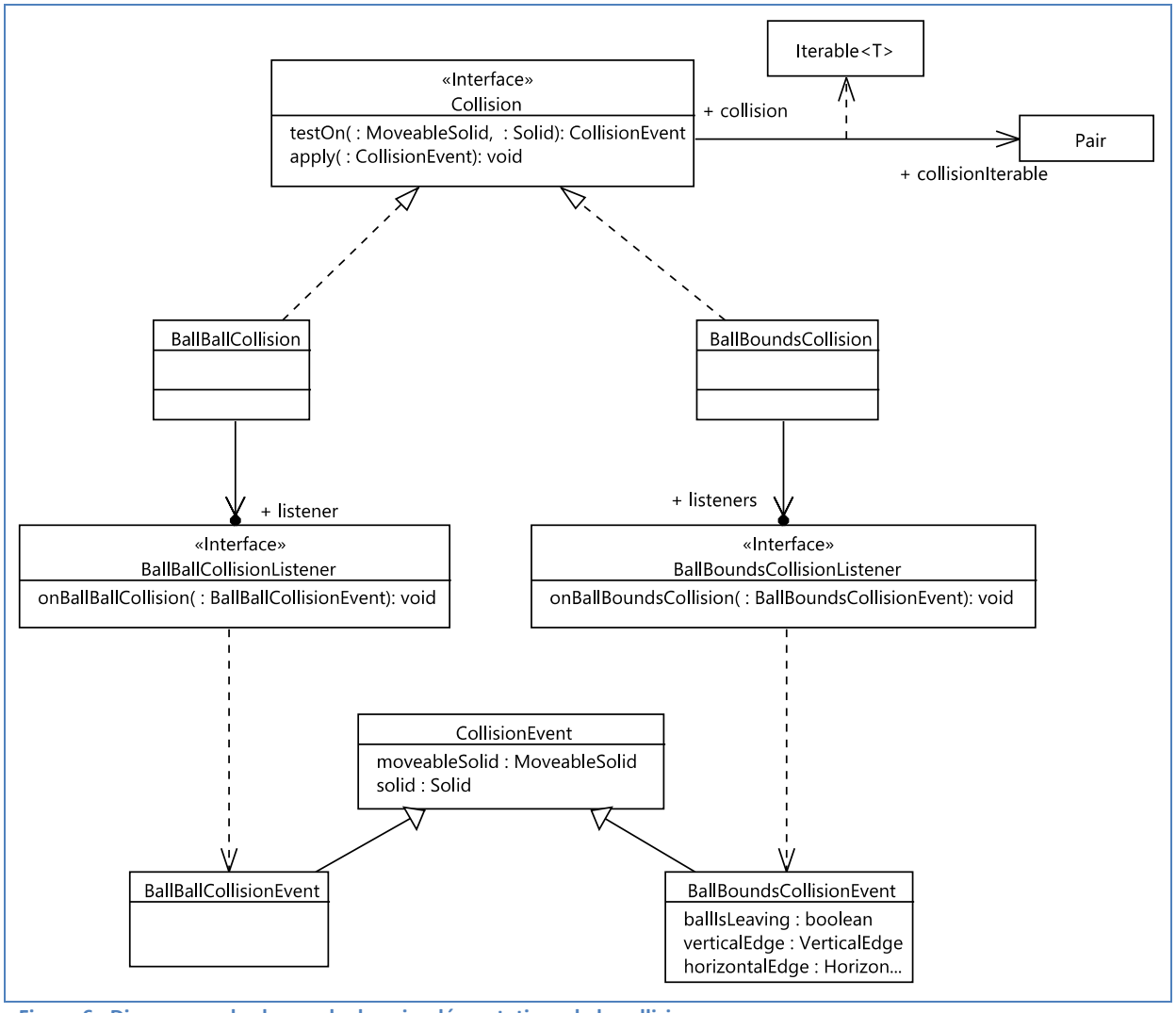

Figure 6 : Diagramme de classes de deux implémentations de la collision

Pour définir quels solides rentreront en collision avec quels autres ; on utilise un iterable de paire de solides. Deux types d'itérable sont fournis : l'un permet d'itérer sur les paires de solides un même ensemble et l'autre entre deux ensembles distincts. Les deux cas sont utilisés dans l'exemple.

Dans le cas de la collision entre les balles et l'écran, deux ensembles entrent en collision : les balles «balls » et les bordures de l'écran « boundsGroup ». On utilise donc « InterIterablePairIterable ».

```
InterIterablePairIterable<Ball, BoundsSolid> ballsBoundsCollisionIterable;
ballsBoundsCollisionIterable = new InterIterablePairIterable<>(balls, boundsGroup);

BallBoundsCollision ballBoundsCollision = new BallBoundsCollision
(
	ballsBoundsCollisionIterable,
	new BounceBallBoundsCollisionEffect(true)
);
world.getCollisions().add(ballBoundsCollision);
```

Dans le cas de la collision entre les balles, pour définir que l'ensemble des balles entre en collision entre elles, on utilise « IntraIterable ».

```
IntraIterablePairIterable<Ball> ballsBallsCollisionIterable;
ballsBallsCollisionIterable = new IntraIterablePairIterable<>(notGhostBalls);

BallBallCollision ballBallCollision = new BallBallCollision
(
	ballsBallsCollisionIterable,
	new BounceBallBallCollisionEffect()
);
world.getCollisions().add(ballBallCollision);
```

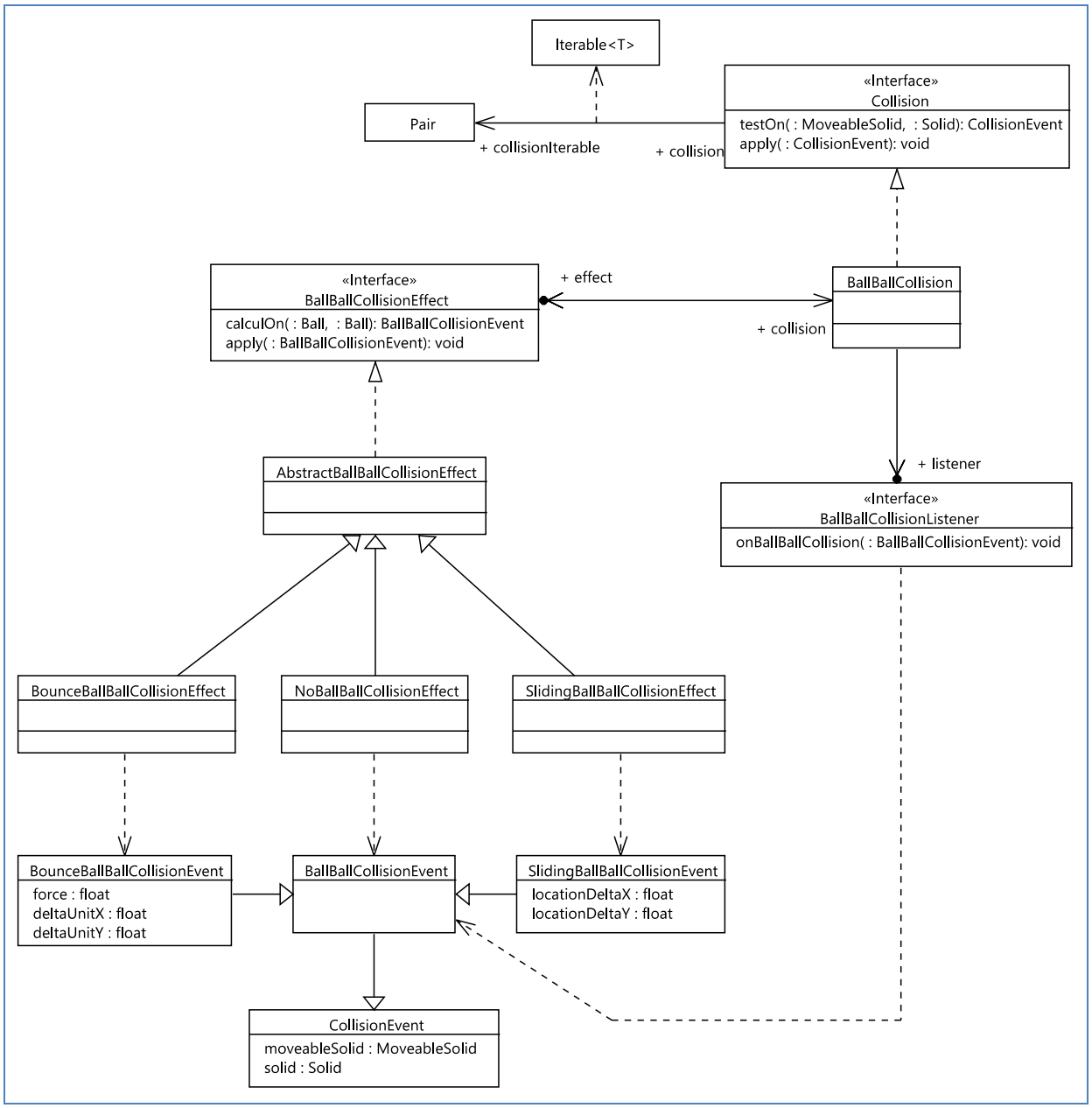

Figure 7 : Diagramme de classes des effets de la collision balle/balle

4.4 Réagir aux collisions

Il est probable que le jeu définisse des règles de jeu lorsque deux solides entrent en collision. Afin que le modèle du jeu puisse appliquer les règles du jeu, les implémentations de l'interface « Collision » permettent de s'abonner aux évènements de collision.

Dans l'exemple, les balles qui ne sont pas des fantômes révèlent leur véritable couleur après l'impact. On s'abonne donc aux évènements de la collision « ballBallCollision ».

```
NotGhostBallsCollisionListener notGhostBallsCollisionListener;
notGhostBallsCollisionListener = new NotGhostBallsCollisionListener();
ballBallCollision.addCollisionListener(notGhostBallsCollisionListener);
```

Un message est envoyé aux deux balles concernées. Ces balles retiennent la date courante de leur monde comme date de dernier impact. Lorsque la vue appellera leur méthode « isHidden() » celle-ci retournera faux pendant un certain temps. La vue dessinera donc les balles avec leur véritable couleur.

```
private class NotGhostBallsCollisionListener implements BallBallCollisionListener
{
	@Override
	public void onBallBallCollision(BallBallCollisionEvent event)
	{
		((GameBall) event.getMoveableSolid()).setImpacted();
		((GameBall) event.getSolid()).setImpacted();
	}
}
```

5 À vous de jouer !

Eh bien voilà, je vous laisse donner libre cours à votre imagination. Si ce tutoriel laisse certaines interrogations sans réponse ou que vous désirez participer au projet, vous pouvez me contacter à android.phy@gmail.com.