Projet sur DP Decorator - Application *Angry Balls*

1. Sujet

Le premier but de ce projet est d'exploiter le DP Decorator pour réécrire la partie maladroite d'une application. La deuxième tâche définissant ce projet consiste à ajouter une fonctionnalité importante à l'application (cf .2.2).
Pour comprendre le travail à réaliser le plus simple est de commencer par faire tourner l'application ***angryballs*** et d'observer ce qu'elle fait.

L'archive ***projet_angry_balls_2023_2024_ressources.zip*** fournie avec ce sujet contient tous les documents nécessaires à la réalisation du projet.
Après décompression de l'archive, on obtient les ressources suivantes :

```
exodecorateur_angryballs_maladroit_sources.jar
mesmaths_sources_avec_awt.jar
malibrairiegui.jar
malibsonique.jar
decompresser_archive.bat
docmesmaths.zip
lance_angryballs_maladroit_bruits.bat
sujet_projet_decorateur_angryballs_2023_2024.doc
Bruitages_malibsonique.doc
Introduction_gestion_fichiers_audio.doc
TestServeurFlammes.exe
config_serveur_flammes.txt
x86_64-6.1.0-release-posix-seh-rt_v5-rev0.7z
```

Angry Balls est une application client-serveur où le client est écrit en JAVA et le serveur en C++. Ce dernier est lancé directement au moyen de l'exécutable **`TestServeurFlammes.exe`**. Par défaut, ce serveur fonctionne avec l'adresse localhost (127.0.0.1) et sur le port 3718.
Ces paramètres peuvent être changés au moyen d'un fichier de configuration, organisé comme **`config_serveur_flammes.txt`** (cf. paragraphes 7.3 à 7.5). Dans la suite, on suppose que le serveur est lancé.
En cas d'échec de démarrage du serveur, la partie JAVA (le client) est conçue pour fonctionner dans une version minimaliste. L'exécutable **`TestServeurFlammes.exe`** ne fonctionne que sur le système Windows. Pour les autres systèmes, nous devrons recompiler le code source C++ avec le compilateur adéquat.
Le dernier document mentionné dans la liste ci-dessus, **`x86_64-6.1.0-release-posix-seh-rt_v5-rev0.7z`**, contient l'archive de la distribution du compilateur qui a été utilisé pour produire l'exécutable **`TestServeurFlammes.exe`**.
Si sur Windows, le serveur C++ ne se lance pas, il est possible qu'il manque une dll. Dans ce cas, installez sur votre ordinateur la même version du compilateur C++ que celle qui a été utilisée (cf. archive ci-dessus) puis ajoutez à votre variable d'environnement système ***PATH***, le chemin complet du compilateur (il s'agit du dossier contenant g++) . Il se termine par *bin*.

Les quatre premiers documents (.jar) contiennent les fichiers source, les fichiers de byte code et les ressources nécessaires à faire fonctionner la partie JAVA de l'application.

L'archive ***exodecorateur_angryballs_maladroit_sources.jar*** contient le code source et le code exécutable de l'application (fct *main*).
Décompressons d'abord cette archive afin de rendre les fichiers audio accessibles à l'application angryballs.
L'archive peut être décompressée avec la commande ***decompresser_archive.bat***, avec n'importe quelle application de compression (zip, rar, etc.) ou encore en tapant directement la ligne de commande contenue dans ***decompresser_archive.bat*** dans une fenêtre invite de commande ou shell linux.
Cette ligne de commande utilise le programme *jar* :

```
jar -xvfM exodecorateur_angryballs_maladroit_sources.jar
```

Après décompression, on obtient un nouveau dossier appelé ***exodecorateur_angryballs***.

L'application peut alors être lancée par la commande :

lance_angryballs_maladroit_bruits.bat

Si cette commande ne fonctionne pas (comme sur Linux par exemple), il faut taper directement le script contenu dans le fichier *.bat* dans une fenêtre *invite de commande* ou *shell*.

Ce script utilise la JVM java :

```
java -classpath .;malibsonique.jar;malibrairiegui.jar;
mesmaths_sources_avec_awt.jar
exodecorateur_angryballs.maladroit.TestAngryBalls
```

L'application utilise :

- la librairie *malibrairiegui.jar* qui contient quelques classes GUI utilitaires.
- la librairie *malibsonique.jar* qui contient la gestion des bruitages.
- la librairie géométrique *mesmaths_sources_avec_awt.jar*.

La documentation sur la librairie *malibsonique.jar* se trouve dans le fichier *Bruitages_malibsonique.doc* et les fichiers sources de cette libraire sont directement inclus dans l'archive *malibsonique.jar*.

La documentation sur la librairie *mesmaths_sources_avec_awt.jar* est fournie dans *docmesmaths.zip* et les fichiers source de cette librairie sont dans *mesmaths_sources_avec_awt.jar*.

Editez le code source de la classe suivante (fonction *main()*) :
exodecorateur_angryballs.maladroit.TestAngryBalls

Examinez les 24 lignes de code :

```
billes.add(new  BilleMvtRURebond(p0, rayon, v0, Color.red));
billes.add(new  BilleMvtPesanteurFrottementRebond(p1, rayon, v1,
                        new Vecteur(0,0.004), Color.yellow,coeffFrottement));

coeffFrottement = 4.5;
billes.add(new BilleMvtNewtonFrottementRebond(p2, rayon, v2,
Color.green,coeffFrottement ));
```

```
billes.add(new              BilleMvtRUPasseMurailles(p3, rayon, v3, Color.cyan));

BilleHurlanteMvtNewtonArret billeNoire;         // cas particulier de la bille
                                                // qui hurle

billes.add(billeNoire = new BilleHurlanteMvtNewtonArret(p4, rayon, v4,
Color.black,hurlements[choixHurlementInitial], cadre));

cadre.addChoixHurlementListener(billeNoire);  // à présent on peut changer le son
                                              // de la bille qui hurle

//-------------------- on ajoute la bille torche -----------------------------

int couleurBleuAzur = 0x003399;

billes.add(new  BilleRebondFrottementFlamme( p5, rayon, v5, new
Color(couleurBleuAzur)));
```

Parmi les six lignes commençant par ***billes.add***, mettez cinq lignes sur six en commentaire et faites tourner l'application. Renouvelez l'opération plusieurs fois en changeant à chaque fois la ligne gardée. Remarquez que les six billes ont toutes un comportement différent. Notez ce qui caractérise le mouvement de chaque bille. Finalement lancez simultanément les six billes en gardant les six lignes de code. Observez le mouvement des six billes et complétez vos remarques concernant leurs comportements respectifs.
Nous pouvons à présent dresser l'inventaire des "accélérations" possibles :

1. absence d'accélération (mouvement **R**ectiligne **U**niforme ou autrement dit en ligne droite à vitesse constante)
2. attirée par les autres billes (attraction universelle due à la force de **Newton**)
3. attraction vers le bas (pesanteur)
4. freinage dû aux frottements dans l'air (frottement visqueux)

Nous pouvons aussi faire l'inventaire des sortes de collisions avec les parois :

5. rebond sur les bords
6. bloquée par un bord
7. franchissement des parois et réapparition sur le côté opposé

Enfin nous pouvons noter les comportements suivants :

8. une bille pousse des « hurlements«.
9. une bille est complétée par une flamme, à la manière d'une comète ou d'une torche.

La bille rouge est munie des comportements (1) et (5)
La bille jaune est munie des comportements (3), (4) et (5)
La bille verte est munie des comportements (2), (4) et (5)
La bille bleu ciel est munie des comportements (1) et (7)
La bille noire est munie des comportements (2), (6) et (8)
La bille bleu marine est munie des comportements (4), (5) et (9)

Chacune de ces billes est une instance d'une classe particulière (cf. les six lignes commençant par ***billes.add***).
Les comportements (2), (3), (4), (8) et (9) sont cumulables et peuvent être combinés à (5), (6) et (7) ; ce qui fait au total 2^5 x 3 = 96 combinaisons possibles. Si l'on veut prévoir toutes les combinaisons à l'aide de classes, il en reste 90 à écrire ! Et on peut imaginer d'autres types

d'accélérations (force de Lorentz, etc.) ou encore changer l'aspect visuel d'une bille. Il est donc bien naïf de tenter d'associer une classe à chaque combinaison de comportements.

2. Amélioration de l'application à l'aide des DP

L'application doit être améliorée sur deux points :

2.1 Réorganisation du code source à l'aide du DP Decorator

Le but ici est donc de transformer les maladroites classes de bille de cette application en une version "bille avec décorateur" où le développeur final assemblera lui-même dynamiquement les billes nues et les comportements. Lorsque ceci sera fait, vous exploiterez votre solution en reconstruisant six billes ayant les mêmes comportements et couleurs que les billes de l'application fournie et ferez tourner votre application. Vous êtes libre d'inventer de nouvelles sortes de billes (Force de Lorentz, etc.).

2.2 Nouveau comportement : bille attrapable et lançable

L'application doit aussi permettre de créer une nouvelle sorte de bille que l'on peut attraper avec la souris puis lancer (comme on lancerait une balle avec la main). Le comportement obtenu doit être le plus réaliste et le plus intuitif possible. Par exemple une grosse bille doit être plus difficile à lancer qu'une petite bille :

☞ par conséquent la main qui tient la souris doit faire un geste plus rapide avec une grosse bille qu'avec une petite bille. De même, une bille, lorsqu'elle est attrapée conserve les comportements qu'elle avait avant d'être attrapée. La main exerce simplement une influence de plus, c'est tout (c'est l'idée même du DP Décorateur).

☞ Enfin, la bille attrapée est soumise aux collisions avec les autres billes de la même façon qu'elle l'était avant d'être attrapée. Elle ne doit donc pas traverser les autres billes en cas de collision !

Ce nouveau comportement est appelé *BillePilotée* (pour bille pilotée par l'utilisateur). Bien sûr, ce comportement doit être mis en oeuvre en exploitant, à l'instar des autres comportements, le DP Decorator. Une bille peut cumuler le comportement *BillePilotée* avec un ou plusieurs des comportements précédents.

☞ La réalisation du comportement *BillePilotée* entraîne nécessairement une mise en oeuvre des DP Decorator et State.

3. Organisation du travail à effectuer

L'application fournie est décomposée en 12 packages ou dossiers.
exodecorateur_angryballs.maladroit
exodecorateur_angryballs.maladroit.bruits
exodecorateur_angryballs.maladroit.modele
exodecorateur_angryballs.maladroit.modele.torche
exodecorateur_angryballs.maladroit.vues
mesmaths
mesmaths.cinematique
mesmaths.geometrie.base
mesmaths.mecanique

musique
musique.javax
outilsvues

3.1 Package exodecorateur_angryballs.maladroit

Le package *exodecorateur_angryballs.maladroit* contient la classe *TestAngryBalls* (fct *main*()) où les six lignes de code vues plus haut sont à changer. Le reste de la fonction *main()* peut rester intact sauf les deux dernières lignes qui pourraient être rendues indépendantes de la librairie graphique *awt*.
L'intérêt de rendre le code source indépendant de la libraire graphique prend tout son sens si on souhaite, par exemple, porter l'application sur le système Android.

Le package *maladroit* contient les classes *EcouteurBoutonLancer* et *EcouteurBoutonArrêter* qui peuvent rester intactes. Ce package contient enfin la classe *AnimationBilles* qui prend en charge l'animation des billes à travers un thread séparé.

☞ C'est cette classe qui appelle les opérations de gestion des mouvements et des collisions des billes ; elle aussi peut rester intacte.

Enfin le package *maladroit* contient la classe *OutilsConfigurationBilleHurlante* responsable du chargement des extraits de sons qui animent la bille hurlante. Cette classe peut elle aussi rester intacte.

3.2 Dossier exodecorateur_angryballs.maladroit.bruits

Les extraits de son qui animent la bille hurlante sont créés à partir de fichiers audio situés dans le dossier *maladroit.bruits*. Pour accomplir sa tâche, la classe *OutilsConfigurationBilleHurlante* utilise un petit fichier texte de configuration *config_audio_bille_hurlante.txt* qui indique quels fichiers audio prendre et quels passages extraire. Le nom du dossier et le nom du fichier de configuration sont bien sûr paramétrables. Le fichier de configuration respecte un format directement expliqué dans le même fichier.
Le dossier *maladroit.bruits* contient aussi les fichiers audio utilisés pour créer les sons de collisions bille-bille et bille-bord (cf. exemple vu en cours).

3.3 Package exodecorateur_angryballs.maladroit.vues

Le package *exodecorateur_angryballs.maladroit.vues* contient cinq classes chargées de visualiser les billes en mouvement. Ce sont les classes *VueBillard*, *CadreAngryBalls*, *Billard*, *BillardAR*, *BoutonChoixHurlement* et *PanneauChoixHurlement*.
C'est la classe *BillardAR* qui dessine les billes (à travers la méthode *dessine*()) (*AR* signifie Active Rendering).
Les deux classes *BoutonChoixHurlement* et *PanneauChoixHurlement* servent bien sûr à proposer les différents bruitages de la bille hurlante. La ligne de boutons est automatiquement remplie à partir de la liste des sons fournie par la méthode *OutilsConfigurationBilleHurlante.chargeSons()*.
Ces six classes peuvent rester intactes.

3.4 Packages exodecorateur_angryballs.maladroit.modele et exodecorateur_angryballs.maladroit.modele.torche

Les packages *exodecorateur_angryballs.maladroit.modele* et *exodecorateur_angryballs.maladroit.modele.torche* contiennent la hiérarchie des classes de billes.
La classe *OutilsBille* peut rester intacte. Les autres classes doivent toutes être revues. C'est là qu'intervient votre contribution. Vous devez "désassembler" ces classes et reconstruire des classes qui représentent bille nue et comportements élémentaires.

☞ **Examinez bien ce qu'elles font avant de les modifier !**

3.5 Package outilsvues

Le package *outilsvues* contient quelques outils utiles aux vues. Les classes de ce package peuvent rester intactes.

*3.6 Packages mesmaths.**

Les packages *mesmaths*, *mesmaths.cinematique*, *mesmaths.geometrie.base* et *mesmaths.mecanique* contiennent essentiellement les classes chargées des calculs de position, de vitesse, d'accélération et de gestion de collisions. Ces classes peuvent rester intactes. Ces classes constituent la librairie *mesmaths_sources_avec_awt.jar*.

3.7 Packages musique et musique.javax

Ces packages servent à gérer l'extraction de sons à partir de fichiers audio et la diffusion de ces sons. Une explication détaillée de cette partie peut être trouvée dans le document *Bruitages_malibsonique.doc* fournie avec ce sujet. Le code source complet de ces packages, les tests unitaires, les fichiers exécutables et les fichiers audio exploités constituent l'archive *malibsonique.jar*.

4. Géométrie, mécanique et calculs

☞ **Il ne vous est pas demandé de faire des calculs de géométrie, de cinématique ou de mécanique.**

Toutes les opérations de calcul nécessaires sont déjà définies dans les classes
OutilsBille
MesMaths
Vecteur (vecteur en 2 dimensions)
Geop (opérations géométriques de base)
Cinematique (mise à jour des vecteurs position et des vecteurs vitesse des billes)
Collisions (gestion des collisions bille-bille et bille-contour)
MecaniquePoint (calcul des accélérations dues aux frottements, à la pesanteur ou à l'attraction par d'autres billes)

5. Conseils

Pour bien réussir la transformation de l'application fournie, appuyez vous sur les quatre points suivants.

5.1 Examen du code source fourni

Pour faire l'inventaire puis écrire les méthodes dont vous devez munir les classes que vous allez définir :

1. Examinez bien les besoins des classes *Animation*, *Billard*, *OutilsBille* et *TestAngryBalls*. Ce sont les seules classes qui appellent les méthodes de la classe *Bille*.
2. Examinez le code des classes de billes fournies :

   - Bille (classe mère de toutes les classes de billes)
   - BilleMvtRURebond (bille rouge)
   - BilleMvtPesanteurFrottementRebond (bille jaune)
   - BilleMvtNewtonFrottementRebond (bille verte)
   - BilleMvtRUPasseMurailles (bille bleu ciel)
   - BilleHurlanteMvtNewtonArret (bille noire)
   - BilleRebondFrottementFlamme (bille bleu marine)

5.2 Définition d'une bille

Une bille, conceptuellement, doit être définie par :

- un vecteur position ***p*** (choisi à l'intérieur du rectangle définissant le composant *Billard*)
- un rayon (réel strictement positif)
- un vecteur vitesse ***v***
- un vecteur accélération ***a***
- une clef (identifiant unique utile à *OutilsBille.gestionCollisionBilleBille* (...) et à *OutilsBille.gestionAccélérationNewton* (...))
- une couleur

5.3 Animation d'une bille

Rappelons que le mouvement d'une bille est entièrement défini par ses trois caractéristiques (***p*** , ***v***, ***a***). La vie d'une bille en mouvement suit les étapes suivantes :

1. Initialisation de ***a*** au vecteur nul à la création de la bille (cf. constructeur classe *Bille*)
2. Initialisation aléatoire de ***p*** et de ***v*** à la création de la bille (cf. fct *main*())
3. mise à jour de ***p*** et de ***v*** (cf. fonction *run* de *AnimationBilles*) en fonction de ***a*** et de la durée *deltaT* (délai entre deux mises à jour successives, *deltaT* est géré par *AnimationBilles*)
4. calcul de ***a***
5. gestion de l'éventuelle collision de la bille avec les autres billes (m-à-j des vecteurs position et vecteurs vitesse des deux billes concernées)
6. gestion de l'éventuelle collision de la bille avec un côté (m-à-j de ***p*** et de ***v***)
7. retour à l'étape (3).

Les étapes (3) à (7) sont réalisées dans la fonction *run* de AnimationBilles.

5.4 Calcul du vecteur accélération a

Le vecteur ***a*** est obtenu en faisant la somme des contributions de toutes les sortes d'accélérations que subit la bille (principe fondamental de la dynamique, cf. vos cours de méca de terminale).
Exemple : Pour une bille subissant les frottements dus à l'air, l'attraction par les autres billes et la pesanteur, on a l'équation suivante :

$$\boldsymbol{a} = \mathbf{0} + \boldsymbol{a}_{frottement} + \boldsymbol{a}_{Newton} + \boldsymbol{a}_{pesanteur} \qquad (1)$$

L'affectation ***a*** = **0** est réalisée directement par la méthode *gestionAccélération* de la classe *Bille*.

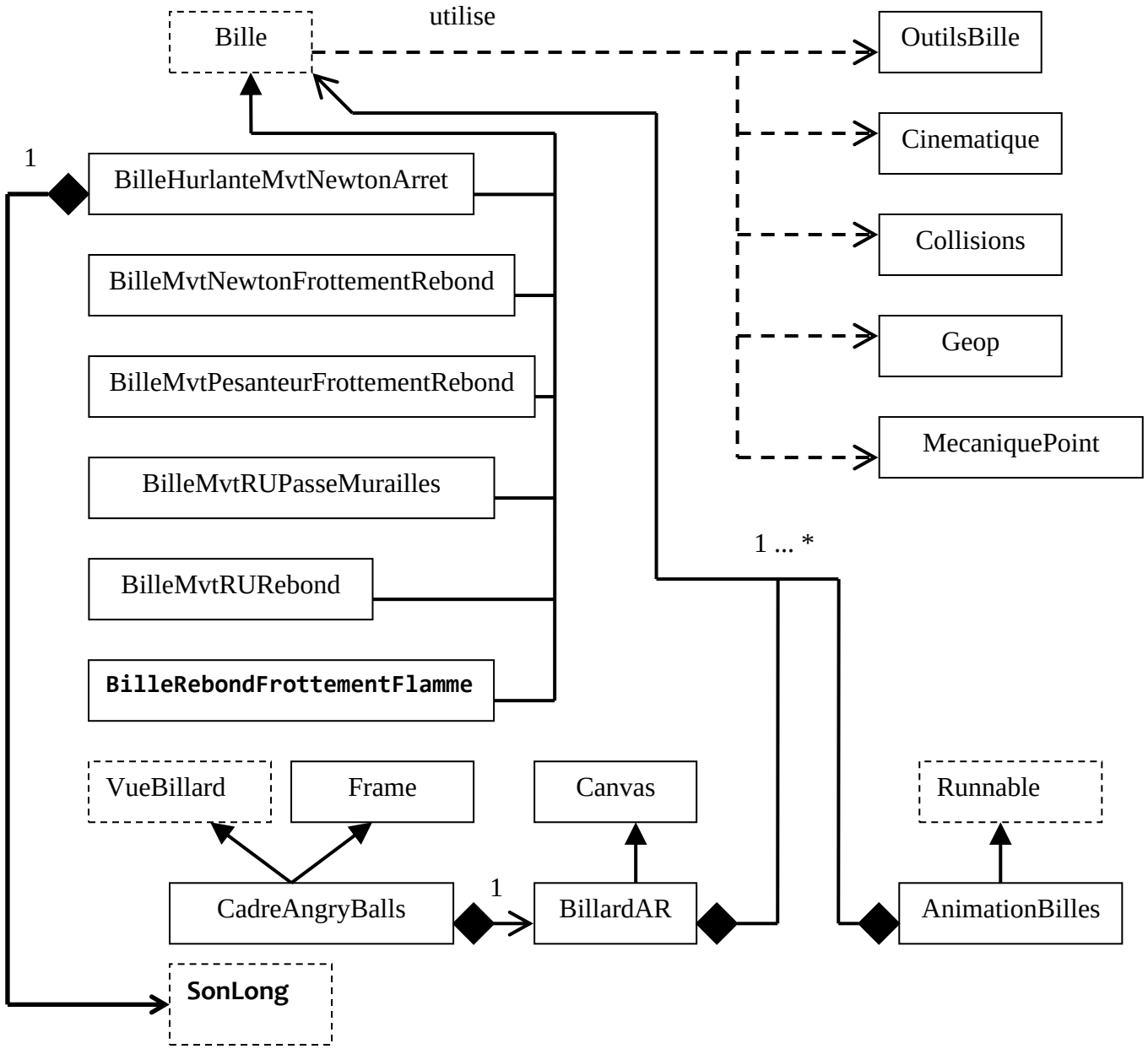
Les termes successifs de l'équation (1) sont ajoutés par les classes dérivées (cf. par exemple méthode *gestionAccélération* de la classe *BilleMvtNewtonFrottementRebond*).

☞ **Notez que :**

1) Le package *mesmaths.cinematique* contient le fichier ***mouvement_point_materiel.doc*** qui explique plus en détail comment modéliser le mouvement d'une bille.
2) Le package *mesmaths.mecanique* contient le fichier ***collision_de_billes_de_billard.doc*** qui explique plus en détail comment modéliser une collision entre deux billes.

6. Hiérarchie des classes de l'application fournie

Voici une vue d'ensemble simplifiée des classes intervenant dans l'application fournie :



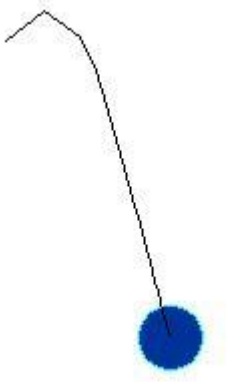
7. Cas particulier de la bille flamme

La bille « flamme » (cf. image ci-dessous) est incontestablement la bille dont le comportement est le plus complexe.

7.1 Classe Echine

La flamme est définie par une *Echine* qui représente la colonne vertébrale (ou l'échine) de la flamme. L'échine est définie par une suite de sommets (cf. classe *mesmaths.geometrie.base.Vecteur* et image ci-dessous). Les segments reliant les sommets sont de longueur constante au cours du temps, ils peuvent être considérés comme des vertèbres rigides. L'échine se déforme constamment pour suivre la bille dans sa trajectoire (comme le sillage d'un bateau). Les calculs de déformation et de déplacement de l'échine sont partagés entre les classes *BilleRebondFrottementFlamme* et *Echine*.
Concernant l'échine, la classe *BilleRebondFrottementFlamme* permet de définir les paramètres suivants :

- nombre de sommets composant l'échine
- longueurs des vertèbres de l'échine



7.2 Flamme

La flamme qui habille l'échine est constituée d'un nuage d'étincelles. Pour chaque étincelle, on définit la position et la couleur. Les attributs *etincelles* et *couleursEtincelles* de la classe *BilleRebondFrottementFlamme* représentent respectivement le tableau des positions des étincelles et le tableau des couleurs des étincelles. Ces deux tableaux sont évidemment mis à jour à chaque rafraîchissement de l'image. Les calculs pour déterminer les positions des étincelles et leurs couleurs sont effectués par un objet *CreateurBellesFlammes*, lui aussi attribut de la classe *BilleRebondFrottementFlamme*.

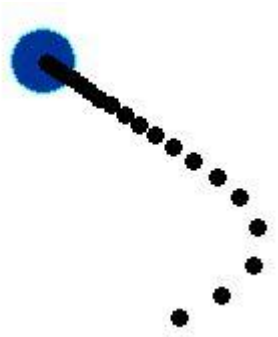
Concernant la flamme, la classe *BilleRebondFrottementFlamme* permet de définir les paramètres suivants :

- *rayonEtincelle* qui définit le rayon d'une étincelle
- *mRangees* tel que *mRangees* + 1 définit le nombre d'étincelles dans la direction de la largeur de la flamme.
- *mEtincelles* tel que *mEtincelles* + 1 définit le nombre d'étincelles dans la direction de la longueur de la flamme.

Le nuage d'étincelles compte donc (*mRangees* +1) x (*mEtincelles* +1) étincelles.

7.3 Client-Serveur de flammes

L'objet *CreateurBellesFlammes* est en fait un client vers un serveur de calcul C++ (mise en œuvre du DP Proxy). En effet, comme tous les langages interprétés, JAVA offre des performances de temps de calcul très médiocres. Aussi, calculer en temps réel les positions et couleurs des 500 étincelles de l'exemple à l'aide de JAVA est une impasse. Ces calculs sont donc confiés à un serveur de flammes écrit en C++.
Le client envoie au serveur les positions des sommets de l'échine ainsi que le rayon de la bille. Le serveur calcule alors les positions et les couleurs des étincelles et renvoie ces informations au *CreateurBellesFlammes*. La classe *BilleRebondFrottementFlamme* prend alors en charge le dessin des étincelles. Si le serveur de flammes n'est pas disponible, un créateur de flammes « *mock* » est créé (classe *CreateurFlammesMock*). Celui-ci calcule simplement un nuage d'étincelles monochrome correspondant aux positions des sommets de l'échine (cf. image ci-dessous).



7.4 Caractéristiques de la flamme – paramétrage du serveur de flammes

Comme la flamme est calculée côté C++, certains paramètres de celle-ci sont définis par le serveur.
Celui-ci impose donc les valeurs pour les cinq paramètres suivants, fixées à la compilation :

1. nombre maximal de sommets de l'échine : 50
2. nombre maximal de lignes d'étincelles (nombre maximal d'étincelles dans la direction de la largeur de la flamme) : 50
3. nombre maximal d'étincelles dans la direction de la longueur : 100
4. nombre maximal de couleurs de la flamme : 10

Il est donc possible de créer une flamme composée de 50 x 100 = 5000 étincelles (si toutefois le processeur est assez rapide pour l'afficher 100 fois par seconde !).

Les valeurs des paramètres suivants sont fixées par défaut mais peuvent être redéfinies au démarrage du serveur au moyen d'un petit fichier de configuration :

5. adresse du serveur, valeur par défaut : 127.0.0.1
6. port du serveur, valeur par défaut : 3718

7. vitesse d'écoulement des étincelles dans la flamme, valeur par défaut : 0.005
8. coefficient de dispersion aléatoire des étincelles, valeur par défaut : 0.01
9. nombre de couleurs, valeur par défaut : 3

Liste des couleurs (données au format RGB 24 bits en hexa), valeurs par défaut :

10. FFB81C (correspond à goldenyellow),
11. DC143C (correspond à rouge crimson),
12. 000000 qui correspond au noir.

Les couleurs sont données dans l'ordre de la bille vers la pointe de la flamme.

Tous ces paramètres sont affichés lors du lancement du serveur.

7.5 Fichier de configuration

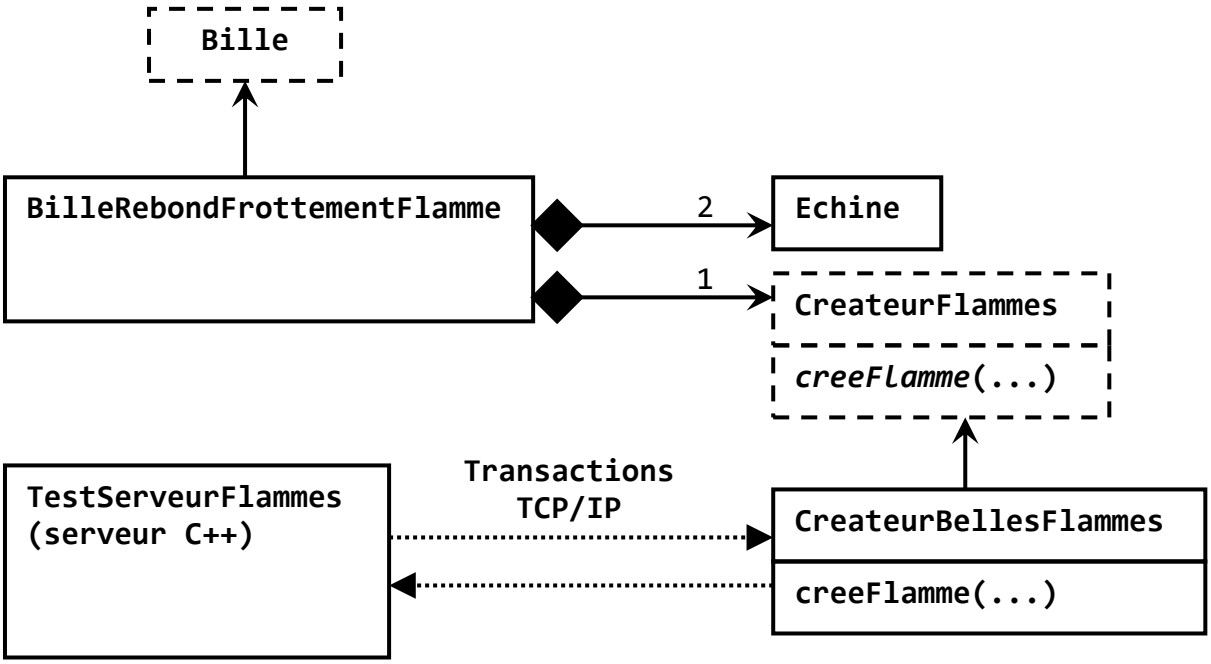
Les paramètres de (5) à (12) peuvent être redéfinis au moyen d'un petit fichier texte de configuration qui respecte la syntaxe du fichier **`config_serveur_flammes.txt`** (cf. archive du projet). Le fichier lui-même contient la description de sa syntaxe. Le nom complet du fichier de configuration (avec extension) doit être placé en paramètre de ligne de commande lors du lancement du serveur. Si le nom du fichier ne contient pas le chemin, alors le fichier de configuration doit être placé dans le même dossier que le fichier exécutable **`TestServeurFlammes.exe`**.
Si le fichier de configuration est introuvable ou s'il contient des erreurs, les valeurs par défaut sont utilisées.

7.6 Forme de l'échine

A chaque rafraîchissement de l'image, la forme de l'échine est recalculée à l'aide d'une loi empirique ... qui pourrait certainement être améliorée. Si cela vous intéresse, venez en discuter avec moi.

7.7 Organisation des classes impliquées dans la gestion de la bille-flamme

Le package *exodecorateur_angryballs.maladroit.modele.torche* contient toutes les classes participant à la réalisation de la bille-flamme.
L'organisation de ces classes peut être résumée par le schéma suivant :



Notez que cette organisation doit être revue puisqu'elle ne met pas en œuvre le DP Decorator (cf. paragraphe 2.1).

8. Quelques cerises sur le gâteau (extensions possibles et facultatives dont la réalisation sera prise en compte dans la notation)

1. Ajoutez des sons d'impact lors des collisions. Pour plus de réalisme, ces sons peuvent être rendus proportionnels à l'intensité du choc et la source du son peut suivre la position de la collision. Dans ce cas, explorez la classe *mesmaths.cinematique.Collisions* pour comprendre où vous devez intervenir pour modifier le code source. Le son peut encore être mis en oeuvre par l'intermédiaire de DPs. Pour réaliser cette fonctionnalité, vous pouvez bien sûr vous inspirer ou réutiliser telles quelles les classes *SonBref* et *SonLong* (cf. packages musique et musiqque.javax de la librairie *malibsonique*). Vous pouvez enfin aussi réutiliser les sons de collisions inclus dans le dossier maladroit.bruits.

2. Modifiez le comportement de la bille flamme. Lorsque celle-ci entre en collision avec une autre bille, elle l'enflamme aussi.

9. Organisation du travail

Les étudiants peuvent se grouper par équipe de 1 à 2 pour réaliser le travail.

10. Documents à rendre

L'application *Angry Balls* complète (sources JAVA + bytecode) améliorée sur les points définis dans le paragraphe 2.

11. Date de remise

Tous les documents concernant le projet sont à rendre au plus tard le dimanche 17 décembre 2023 à minuit à D. Michel. N'oubliez pas d'indiquer : noms et prénoms des membres de l'équipe, nom de la filière et nom du projet. Le projet doit être rendu par courrier électronique à l'adresse suivante : domic62@hotmail.com.

12. Soutenance

Les soutenances seront organisées à partir du lundi 18 décembre 2023, elles se dérouleront à l'UFR MIM courant décembre 2023 et janvier 2024.
Le planning des soutenances sera communiqué par courrier électronique la semaine du lundi 27 novembre 2023. Il aura la forme d'un fichier excel partagé qui définira les créneaux disponibles pour placer les soutenances. Les étudiants seront invités à s'y inscrire eux-mêmes pour prendre RDV. Tous les membres de l'équipe devront être présents. La soutenance durera environ 45 minutes.

13. Notation

La notation prendra en compte :

- la qualité du code source et la rigueur de programmation
- la mise en oeuvre de Design Patterns
- Le fonctionnement de l'application

14. Mots de la fin

Prenez votre temps, examinez bien les classes de l'application fournie et n'hésitez pas à poser des questions ou à solliciter des RDVs pour discuter de points de programmation.

Bon travail, j'espère que vous vous amuserez autant que moi à réaliser ce projet !