

Code branche PROGR	Ministère de l'Éducation nationale, de l'Enfance et de la Jeunesse EXAMEN DE FIN D'ETUDES SECONDAIRES TECHNIQUES Régime technique – Session 2015/2016	
Épreuve écrite	Branche	Division / Section
Durée épreuve 4h	Programmation	GI – Section informatique
Date épreuve <i>1-6-2016</i>		

Dans votre répertoire de travail, vous trouverez un sous-dossier nommé **EXAMEN_GI_xx**. Renommez ce dossier en remplaçant le nom par votre code de l'examen (Exemple de notation : **LAM_GI_07**). Tous vos fichiers devront être sauvegardés à l'intérieur de ce sous-dossier, qui sera appelé « votre dossier » dans la suite !

Question 1

60 points

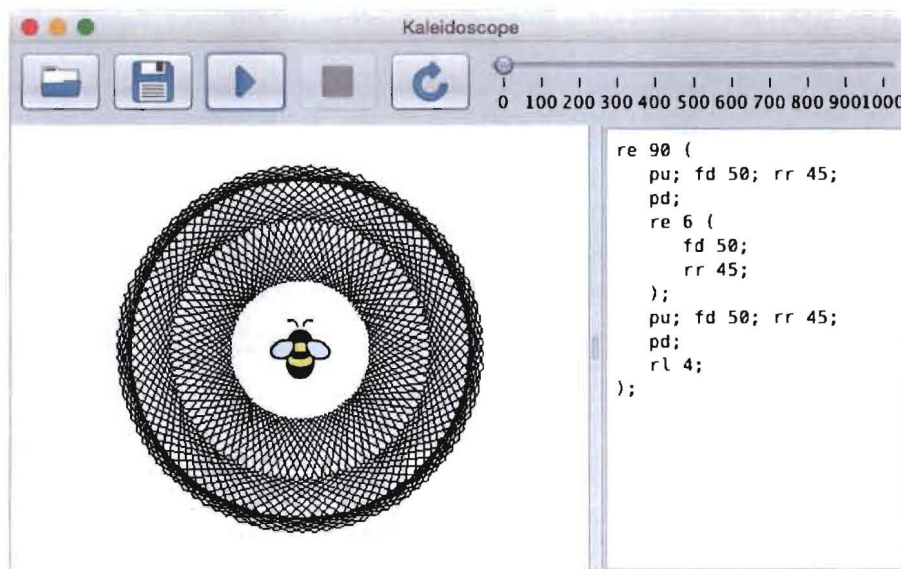
Dans la suite vous allez écrire une version très simpliste d'un clone du célèbre programme « Logo » dans lequel l'utilisateur a pu diriger une tortue à l'aide de commandes textuelles. Dans le temps ce programme était utilisé pour faire apprendre la programmation aux élèves.

Un programme de démonstration vous est fourni dans votre dossier. Veuillez le démarrer et l'analyser ! Il y a aussi quelques fichiers .kal de démonstration que vous pouvez charger et exécuter. Regardez aussi le diagramme des objets en dernière page.

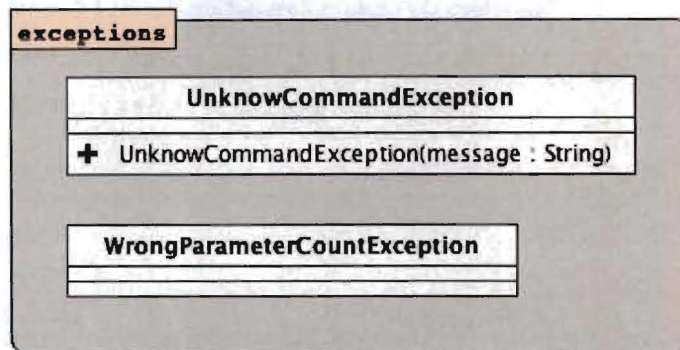
– ensuite –

Ouvrez le projet `kaleidoscope` qui se trouve déjà dans votre répertoire et complétez-le.

L'interface graphique vous est fournie presque entièrement. Il vous reste à la programmer.



Plus loin dans le code, nous aurons besoin de deux exceptions:



WrongParameterCountException – à créer – (0.5 points)

- Il s'agit d'une exception simple, qui, comme le nom l'indique déjà, est lancée si le nombre de paramètres n'est pas conforme.

UnknowCommandException – à créer – (0.5 points)

- Cette exception est lancée si une commande non connue sera lue. Son constructeur possède un paramètre via lequel des détails sur la commande non valide devront être passés.

L'abeille, contrôlée par l'utilisateur, laissera des traces sous formes de lignes:

Line – fournie

- La classe `Line` représente un trait entre les points (x_1, y_1) et (x_2, y_2) .
- La méthode `setEnd` repositionne l'extrémité (x_2, y_2) de la ligne.

Line
- x1 : int
- y1 : int
- x2 : int
- y2 : int
+ Line(x1 : int, y1 : int, x2 : int, y2 : int)
+ getX1() : int
+ getY1() : int
+ getX2() : int
+ getY2() : int
+ setEnd(x : int, y : int) : void

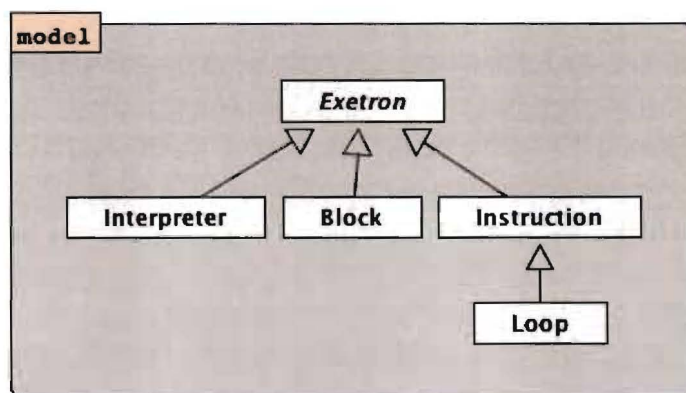
Ensuite il faut créer la classe principale qui représente l'avatar à manipulé, représenté dans le présent programme par une abeille.

Avatar
- x : double
- y : double
- angle : int
- penDown : boolean
- lines : ArrayList<Line>
+ drawImageRotated(g : Graphics, image : Image, x : int, y : int, angle : double) : void
+ draw(g : Graphics) : void
+ reset() : void
+ moveForward(pixel : int) : void
+ moveBackward(pixel : int) : void
+ rotateLeft(degree : int) : void
+ rotateRight(degree : int) : void
+ penUp() : void
+ penDown() : void
+ setX(x : double) : void
+ setY(y : double) : void
+ setAngle(angle : int) : void

Avatar – à créer – (9 points)

- Cette interface représente l'avatar que l'utilisateur pourra faire bouger. Dans l'exercice que voici nous allons utiliser l'image d'une abeille.
- Définissez l'interface suivant le schéma UML donné ci-dessus !
- Les attributs `x` et `y` représentent la position de l'avatar tandis que l'attribut `angle` donne son orientation. Ces attributs possèdent des modificateurs.
- L'attribut `penDown` indique si l'avatar doit laisser des traces ou non.
- L'attribut `lines` est la liste de traces laissées par l'avatar.
- La méthode `drawImageRotated` vous est fournie. Elle se trouve dans le fichier `drawImageRotated.txt` dans votre répertoire de base.
- La méthode `drawImageRotated` affiche une image avec un certain angle donné. Cette méthode vous est fournie dans le fichier « `drawImageRotated.txt` » qui se trouve dans votre répertoire de base.
- La méthode `draw` dessine d'abord toutes les traces (en noir) qu'a laissées l'avatar. A la fin l'avatar est dessiné (voir code donné dans le fichier « `drawImageRotated.txt` »).
- La méthode `reset` vide la liste des traces et remet `true` l'attribut `penDown`.
- La méthode `moveForward` fait ceci :
 - Elle fait avancer l'avatar suivant les formules ci-dessous:
 - $x = x - pixel \cdot \cos(angle)$
 - $y = y - pixel \cdot \sin(angle)$
 - Si l'avatar est supposé laisser des traces, il faut ajouter une nouvelle ligne.
- La méthode `moveBackward` fait reculer l'avatar.
- La méthode `rotateRight` change l'orientation de l'avatar vers la droite. L'angle du changement est donné en degrés par le paramètre.
- La méthode `rotateLeft` change l'orientation de l'avatar vers la gauche. L'angle du changement est donné en degrés par le paramètre.
- Les méthodes `penUp` fait en sorte que l'avatar ne laisse plus de traces.
- Les méthodes `penDown` fait en sorte que l'avatar laisse des traces.

Dans les prochaines étapes vous aller développer le moteur qui permet de lire, d'interpréter et d'exécuter les instructions textuelles écrites par l'utilisateur.



Exetron – à créer – (1 point)

- Cette classe abstraite représente un élément exécutable.
 - Comme les éléments exécutables peuvent s'enchaîner, l'attribut `next` pointe vers un autre `Exetron`. Un programme est donc une liste chaînée d'exétrons.
 - La méthode `executeStep` exécute un pas de l'élément. Comme un élément peut nécessiter plusieurs pas à terminer, elle retourne `true` si et seulement si son exécution est totalement terminée.
- Exemple: Une boucle (**Loop**) qui doit exécuter une instruction 5 fois aura besoin de 5 pas. Ce n'est qu'après le cinquième appel de la méthode `executeStep` qu'elle retourne `true`.
- La méthode `reset` met l'élément dans son état initial. Cette méthode sera par exemple utilisée pour réinitialiser le compteur d'une boucle.

Exetron
+ next : Exetron
+ executeStep(avatar : Avatar) : boolean
+ reset() : void

Instruction – à créer – (9.5 points)

- Cette classe représente une instruction.
- L'attribut `cmd` représente le nom de la commande.
- L'attribut `param` représente le paramètre optionnel de l'instruction.
- L'attribut `knownCommands` est une liste avec les noms de toutes commandes connues.
- Les commandes que `Kaleidoscope` devra reconnaître et exécuter, sont les suivantes :

Instruction
cmd : String
param : int
- knownCommands : String[]
- isKnownCommand(cmd : String) : boolean
+ Instruction(command : String)
+ executeStep(avatar : Avatar) : boolean
+ reset() : void

Commande	Paramètre?	Exemple	Description
fd	Oui: nombre de pixels	fd 50;	« forward », fait avancer l'avatar
bd	Oui: nombre de pixels	bd 100;	« backward », fait reculer l'avatar
rr	Oui: degrés	rr 90;	« rotate right », fait tourner l'avatar à droite
rl	Oui: degrés	rl 45;	« rotate left », fait tourner l'avatar à gauche
pu	Non	pu;	« pen up », l'avatar ne laisse plus de trace
pd	Non	pd;	« pen down », l'avatar trace son chemin
re	Oui: nombre de répétitions et instructions à répéter	re 10 (fd 20; rr 45;);	« repeat », boucle servant à répéter d'autres instructions un nombre précis de fois

- On remarque que :
 - Les paramètres sont toujours des nombres entiers.
 - Toute instruction se termine par un point-virgule.
 - Un bloc est entouré de parenthèses.
- La méthode `isKnownCommand` teste si le nom de la commande passé en tant que paramètre se trouve dans la liste des noms de commandes connues.
- Le constructeur est supposé de recevoir en paramètre une commande, sans le point-virgule à la fin.
 - Sa tâche est de séparer le nom de la commande du paramètre éventuel et de les sauvegarder dans les attributs y relatifs. A part l'espace entre le nom de la commande et le paramètre, tous les autres espaces sont à ignorer !
 - S'il s'agit d'une commande non connue, l'exception `UnknownCommandException` devra être lancée.

- S'il s'agit d'une commande avec paramètre et que le paramètre n'est pas présent, l'exception `WrongParameterCountException` devra être lancée.
- Pour la commande « repeat », le bloc d'instructions à exécuter n'est pas passé dans le constructeur. Il sera rattaché ultérieurement. Il n'y a donc pas besoin de prévoir un cas spécial à cet effet !
- La méthode `executeStep` exécute l'instruction en effectuant l'action adéquate sur l'avatar passé en tant que paramètre. Elle retourne donc toujours `true`.
- La méthode `reset` est vide.

Block – à créer – (10 points)

- Cette classe représente un bloc d'une ou de plusieurs instructions (= exétron) qui sont enchaînées l'une après l'autre et dont la première est référencée par l'attribut `first`.
- L'attribut `actual` pointe vers la prochaine instruction à exécuter.
- La méthode `add` ajoute une instruction à la fin de la liste des instructions.
- La méthode `size` retourne le nombre d'instructions contenues dans la liste d'instructions.
- La méthode `executeStep` exécute un pas du bloc :
 - si l'attribut `actual` ne pointe pas vers une instruction, c'est la première qu'il faut choisir,
 - elle exécute un pas de l'instruction à exécuter,
 - si l'instruction à exécuter est terminée, l'attribut `actual` doit pointer vers la prochaine instruction du bloc d'instructions à exécuter,
 - s'il n'y a plus de prochaine instruction à exécuter, le bloc est terminé et devra être réinitialisé.
- La méthode `reset` fait pointer l'attribut `actual` vers la première instruction du bloc.

Block	
–	<code>actual : Exetron</code>
–	<code>first : Exetron</code>
+	<code>add(exetron : Exetron) : void</code>
–	<code>size() : int</code>
+	<code>executeStep(avatar : Avatar) : boolean</code>
+	<code>reset() : void</code>

Loop – à créer – (5 points)

- Cette classe représente une boucle.
- L'attribut `block` contient les instructions à répéter.
- L'attribut `counter` est le compteur interne indiquant le nombre de fois qu'elle a déjà été exécutée.
- Étant donné que la classe `Loop` hérite de la classe `Instruction`, le constructeur fait appel au constructeur hérité.
- La méthode `executeStep` exécute un pas de la boucle:
 - elle exécute un pas du bloc d'instructions (voir description de la classe `Block` ci-dessus),
 - si le bloc d'instructions a terminé, le compteur interne de la boucle est incrémenté et le bloc d'instructions est réinitialisé,
 - si le compteur interne a dépassé le nombre de répétitions à effectuer (attribut `param`), la boucle est terminée et devra aussi être réinitialisée.
- La méthode `reset` réinitialise la boucle, c'est-à-dire elle remet à zéro le compteur interne et réinitialise le bloc d'instructions.

Loop	
–	<code>block : Block</code>
–	<code>counter : int</code>
+	<code>Loop(command : String, block : Block)</code>
+	<code>executeStep(avatar : Avatar) : boolean</code>
+	<code>reset() : void</code>

Interpreter – à créer – (11.5 points)

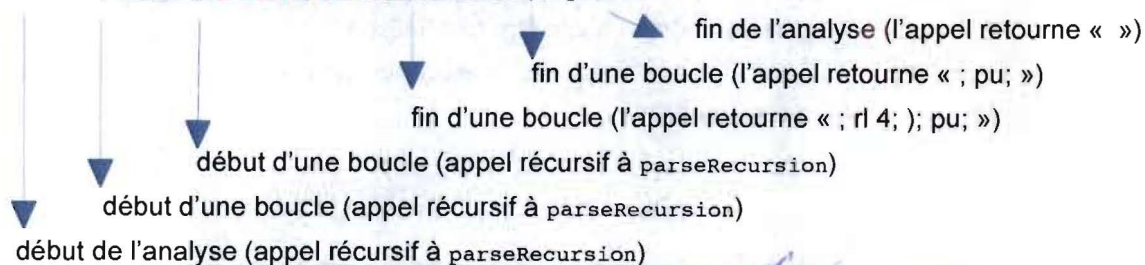
- Cette classe a comme tâche de décoder les commandes données par l'utilisateur et de les interpréter / exécuter. Elle hérite de la classe `Exetron` et utilise l'attribut `next` pour stocker son bloc d'instructions.
- La méthode `executeStep` exécute le bloc d'instruction et retourne le résultat obtenu.
- La méthode `reset` réinitialise le bloc d'instructions.
- La méthode `parse` analyse les commandes qui se trouvent dans le texte passé en tant que paramètre et remplit le bloc d'instructions de manière récursive.
- La méthode `parseRecursion` analyse le texte passé en tant que paramètre et en construit le modèle qu'elle sauvegarde dans l'objet `block` passé aussi en tant que paramètre.
 - Lors de l'analyse la casse (majuscule ou minuscule) des caractères est ignorée de même que les espaces au début et à la fin du texte passé en tant que paramètre.
 - L'analyse se fait symbole par symbole en utilisant une mémoire tampon (EN : buffer). Le tampon n'est rien d'autre qu'une chaîne de caractères qui a le rôle de retenir la suite de caractères jusqu'à ce qu'on ait trouvé la fin de l'instruction ou de la boucle actuelle.
 - Si le symbole « ; » ou «) » est rencontré ou si on est arrivé à la fin du texte, alors on est arrivé à la fin d'une instruction ou d'une boucle qu'il faudra ajouter à la liste des instructions à exécuter.
 - Avant de créer et de sauvegarder une nouvelle instruction, il faut vérifier que celle-ci ne soit pas vide. Tous les espaces au début et à la fin doivent être ignorés !
 - Le tampon est vidé.
 - S'il s'agit du symbole «) », la méthode doit se terminer et retourner la partie non encore analysée du texte passé en tant que paramètre.
 - Si le symbole « (» est rencontré, on est arrivé au début d'un nouveau bloc d'instructions d'une boucle.
 - Il faut donc créer un nouveau bloc vide qui sera rempli par un appel récursif.
 - Finalement une nouvelle boucle doit être créée pour le bloc et cette boucle doit être ajoutée à la liste des instructions.
 - Le tampon doit être vidé, puis l'analyse se poursuivra à la prochaine position non analysée.
 - Si aucun symbole spécial n'est rencontré, le symbole actuel est ajouté à la fin du tampon.

Interpreter	
+	<code>parse(commands : String) : void</code>
-	<code>parseRecursion(commands : String, block : Block) : String</code>
+	<code>executeStep(avatar : Avatar) : boolean</code>
+	<code>reset() : void</code>

Exemple:

Supposons que `text` contienne le code suivant, voici ce que la méthode `parse` va faire :

`re 90 (re 8 (fd 50; rr 45;); rl 4;); pu;`



Finalement il ne reste qu'à implémenter l'interface graphique.

Drawpanel – fournie

- Cette classe est le canevas sur lequel est dessiné l'avatar ainsi que les traces qu'il laisse sur son chemin.
- Le code source vous est fourni, mais vous devez encore supprimer les instructions mises en commentaire.

DrawPanel
– avatar : Avatar
+ DrawPanel()
+ setProgram(program : Avatar) : void
paintComponent(g : Graphics) : void
– initComponents() : void

Mainframe – à compléter – (13 points)

- Le projet contient déjà quelques lignes de code que vous n'avez plus besoin d'écrire :

MainFrame
– avatar : Avatar
– interpreter : Interpreter
– timer : Timer
+ MainFrame()
– resetAvatar() : void
– initComponents() : void
– openButtonActionPerformed(evt : java.awt.event.ActionEvent) : void
– saveButtonActionPerformed(evt : java.awt.event.ActionEvent) : void
– runButtonActionPerformed(evt : java.awt.event.ActionEvent) : void
– formComponentResized(evt : java.awt.event.ComponentEvent) : void
– stepSliderStateChanged(evt : javax.swing.event.ChangeEvent) : void
– stopButtonActionPerformed(evt : java.awt.event.ActionEvent) : void
– resetButtonActionPerformed(evt : java.awt.event.ActionEvent) : void
+ main(args[] : String) : void

- Le bouton `stopButton` arrête le chronomètre, s'il existe et s'il est actif, et adapte les boutons.
- Si l'utilisateur déplace le curseur du `stepSlider`, la fréquence du chronomètre est adaptée en fonction de sa valeur.
- Le bouton `resetButton` réinitialise l'avatar.
- Compléter la méthode `resetAvatar` qui réinitialise l'avatar et qui le positionne de manière à ce qu'il se trouve au milieu du canevas et regarde vers le haut.
- Un clic sur le bouton `openButton` doit permettre à l'utilisateur de charger le contenu d'un fichier texte « Kaleidoscope Program » avec l'extension « .kal » dans la zone code à droite de l'application. Le contenu de l'éditeur de code est remplacé sans avertissement préalable de l'utilisateur.
- Un clic sur le bouton `saveButton` doit permettre à l'utilisateur de sauvegarder le contenu de l'éditeur de code dans un fichier texte « Kaleidoscope Program » avec l'extension « .kal ». Si l'utilisateur a omis l'extension, le programme l'ajoute automatiquement. Si le fichier de destination existe déjà, son contenu est remplacé sans avertissement préalable de l'utilisateur.
- En cliquant sur le bouton `runButton` :
 - Appel à la méthode de réaction de `stopButton` afin de s'assurer que le chronomètre n'est plus actif.
 - Un nouveau chronomètre est créé qui exécute un pas du programme (contenu dans l'interpréteur) à chaque fois qu'il se déclenche. Si l'interpréteur a terminé son travail, le chronomètre se désactive soi-même. Attention à l'état des boutons.
 - Avant de pouvoir lancer le chronomètre, il faut réinitialiser l'avatar et charger le code contenu dans l'éditeur dans l'interpréteur, puis il faut adapter les boutons (voir programme de démonstration).

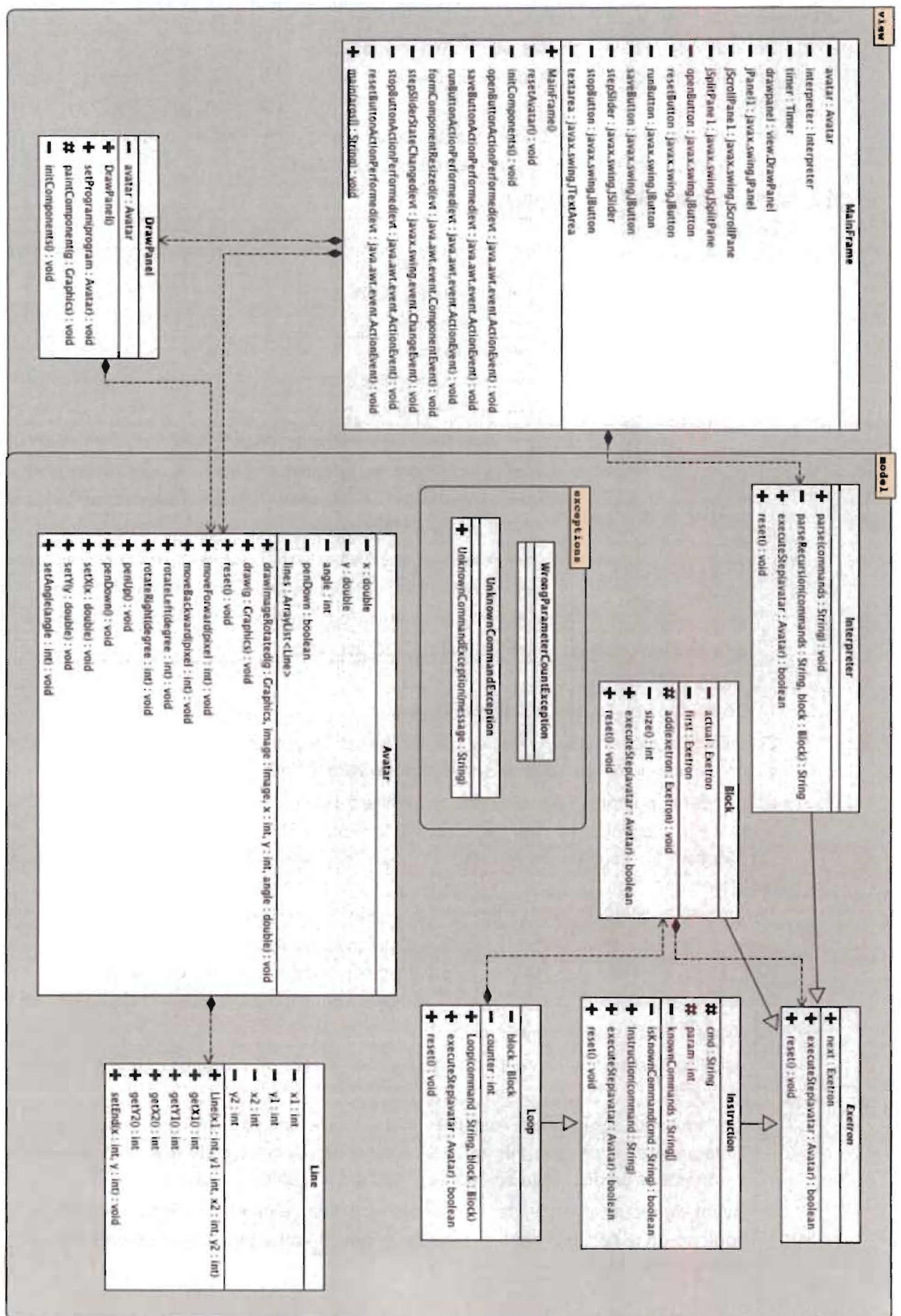


Diagramme des objets correspondant au fichier de démonstration « flower.kal ».

