Tutoriel moteur de jeu

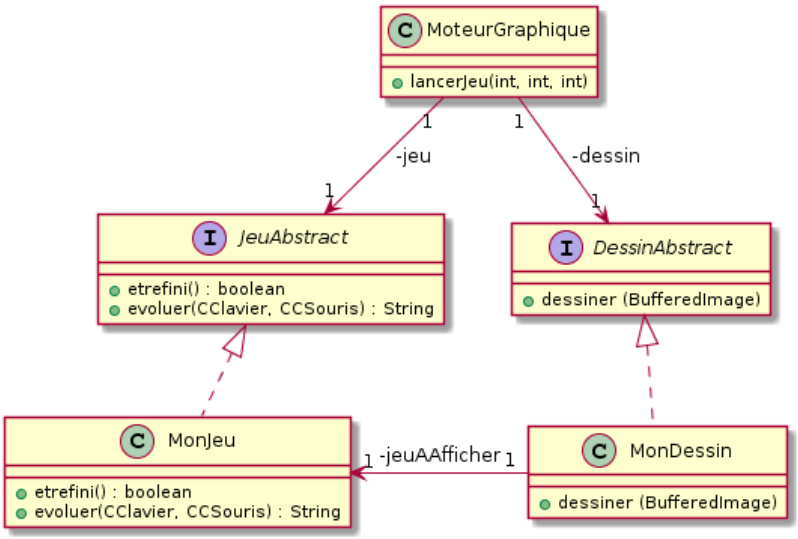
Ce TD a pour objectif de vous faire manipuler la bibliothèque fournie qui fonctionne en JAVA.

1 Récupération des fichiers

Le fichier « **`moteurJeuScCO.zip`** » contient les fichiers java à la base du moteur de jeu. Ces fichiers sont organisés en package

- le package **`moteur`** contient les classes permettant de gérer l'affichage et d'exécuter le jeu
- le package **`sprite`** contient les classes pour charger et dessiner des sprites.

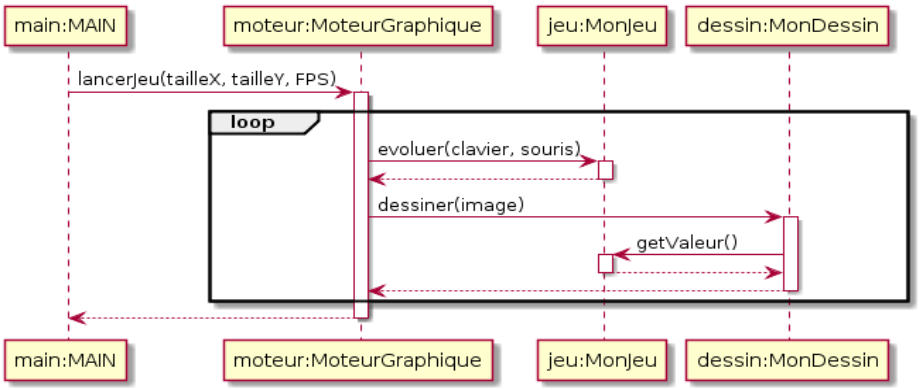
Le package moteur contient le Moteur de jeu qui possède un attribut de type **`DessinAbstract`** et un attribut de type **`JeuAsbtract`**. En redéfinissant des classes implémentant ces interfaces (ici MonJeu et MonDessin), il est possible de proposer son propre jeu.



A l'exécution, on créer le moteur avec un jeu et un dessin puis on appelle la méthode **`lancerJeu`** du moteur en passant en paramètre la taille en X et en Y ainsi que le nombre de frames par secondes.

Le **`moteur`** à son tour appelle de manière cyclique (cf diagramme de séquence ci-dessous)

- la méthode **`evoluer`** du **`jeu`** possédé en attribut
- puis la méthode **`dessiner`** du **`Dessin`** possédé en attribut, le **`dessin`** utilise le **`jeu`** pour savoir quelle information afficher et quoi en faire.



2 Affichage simple

Pour utiliser le moteur, il suffit simplement de créer deux classes et de créer un objet moteur de jeu en passant ces classes en paramètre.

Les classes concernées sont

- une classe destinée à représenter le jeu, cette classe hérite de **`JeuAsbtract`**
- une classe destinée chargée d'afficher le jeu, cette classe hérite de **`DessinAbstract`**

Dans un premier temps, on va créer un jeu simple dans lequel il ne se passe rien avec un affichage simple constitué d'un rectangle vert avec des cercles bleus au coins.

Pour cela

- faire une classe **`Jeu`** vide héritant de **`JeuAsbtract`**
- faire une classe **`Dessin`** héritant de **`DessinAbstract`** en remplissant le contenu de la méthode dessiner.

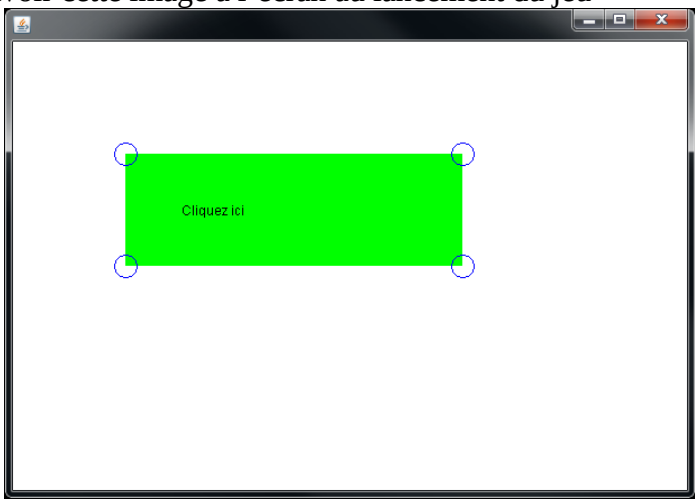
Pour effectuer un dessin dans dessiner, il suffit

- de récupérer un objet **`graphics2D`** à partir de l'image en paramètre
- d'appeler des primitives d'affichage sur ce **`graphics`**
- puis de libérer le **`graphics`** avec la méthode **`dispose`**()

Les primitives d'affichages sont

- toutes les méthodes de **`graphics2D`** (cf API Java)
- parmi celles ci, vous utiliserez
  - **`drawRect`** pour dessiner un rectangle
  - **`drawOval`** pour dessiner un ovale
  - **`drawText`** pour écrire à l'écran
  - **`filleRect`** et **`fillOval`** pour dessiner un rectangle/ovale plein
  - **`changeColor`** pour changer la couleur du crayon

A l'issue, vous devez avoir cette image à l'écran au lancement du jeu



3 Gestion de la souris

Le moteur de jeu fournit aussi toutes les méthodes permettant de suivre et de réagir à la souris.
En effet, la méthode **`evoluer`** de la classe **`Jeu`** prend en paramètre deux objets

- un objet de type **`CClavier`** destiné à suivre les actions effectuées au clavier

- un objet de type **`CSouris`** destiné à suivre les actions effectuées à la souris

C'est de l'objet de type **`CSouris`** que cette partie traitera.

3.1 Suivi de la souris (v02)

Il est possible d'interroger la position de la souris grâce aux méthode **`getX`** et **`getY`** de l'objet **`CSouris`**.

Dans le principe, l'affichage ne doit pas contenir les données à afficher. Ces données doivent être stockées au niveau du jeu qui contient les données utiles.

On va modifier jeu pour qu'il suive la souris :

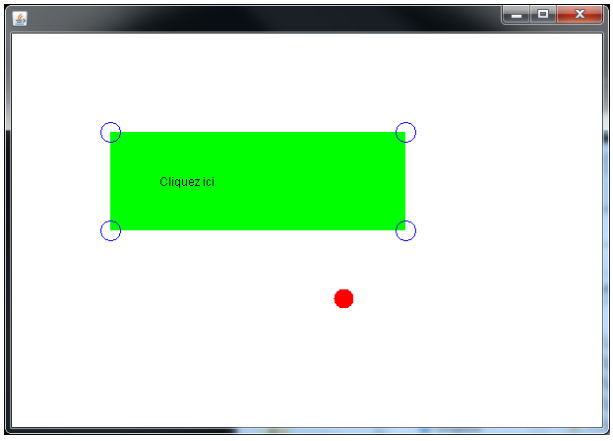
- il faut donc ajouter dans la classe Jeu deux attributs entier **`sx`** et **`sy`** destinés à stocker les coordonnées de la souris
- quand la méthode **`evoluer`** est appelée, le jeu met à jour ces attributs pour y stocker les coordonnées de la souris obtenues grâce avec les méthodes **`getX`** et **`getY`**.
- Pour suivre dans un premier temps, la méthode **`evoluer`** affichera les valeurs de la souris dans la console avec un **`System.out.println`**.

Une fois cette étape faite, on va simplement modifier l'affichage Dans un second temps, on va modifier l'affichage en y ajoutant un cercle tracé en fonction de la position de la souris stockée dans les attributs **`sx`** et **`sy`** de Jeu.

Ainsi, à l'exécution, à chaque itération,

1. Le moteur appelle la méthode **`evoluer`** qui met à jour **`sx`** et **`sy`** en fonction la position de la souris ;
2. Le moteur appelle l'affichage qui affiche un cercle à cette position ;
3. Le moteur réitéré ces opérations.

On obtient la figure suivante :



3.2 Détection de l'appui d'un bouton (v02)

Il est possible aussi d'utiliser l'objet **`CSouris`** pour detecter quand un bouton de la souris est appuyé.

La méthode **`boolean getPressed()`** de la classe **`CSouris`** renvoie **`true`** si et seulement si un bouton de la souris est appuyé.

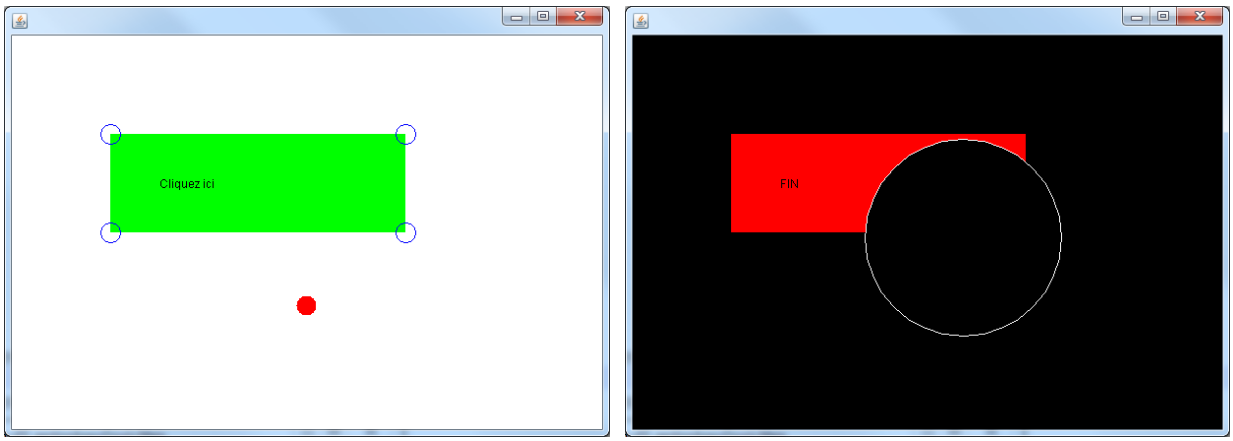
Stockez cette nouvelle information dans un nouvel attribut et modifier la couleur du cercle en fonction de l'appui :

- si un bouton est appuyé, le cercle doit etre bleu
- si aucun bouton n'est appuyé, le cercle qui suit la souris est rouge.

3.3 Prise en compte de mode de jeu (v03)

Il est utile de découper un jeu en plusieurs modes, pour cela on va s'intéresser à un cas simple avec un jeu en deux modes :

1. au lancement le jeu est au début, un rectangle est affiché à l'écran
2. quand on clique sur le rectangle, il passe en mode fin, le rectangle est affiché différemment.



Pour cela, on va ajouter un nouvel attribut dans la classe **`Jeu`**. Cet attribut est un **`String`** nommé **`mode`**.

En fonction de la valeur de l'attribut, le jeu et le dessin fonctionneront différemment. Cela se fait en utilisant simplement un switch selon cet attribut

- si l'attribut vaut « **`debut`** »
  - le jeu fonctionnera comme avant et s'il y a un click au centre du rectangle, le mode prendra la valeur « **`fin`** »
  - le jeu s'affichera comme avant
- si l'attribut vaut « **`fin`** »
  - le jeu consistera simplement à mettre à jour les valeurs de **`sx`** et **`sy`**
  - l'affichage se fera différemment

3.4 Gestion appui souris (v04)

Dans cette partie, on va chercher à passer d'un mode de jeu à un autre en fonction des clicks de souris. Désormais, lorsqu'on se trouve dans le mode « fin », une détection de click de souris fait à nouveau passer le mode en mode « **`debut`** ».

Une fois que cela est codé, on peut observer qu'on oscille d'un mode à un autre sans arrêt. En effet, la méthode ne détecte par la présence d'un click de souris mais renvoie vrai tant que le bouton de la souris est appuyé. Il faut donc faire appel à une autre méthode de la classe **`CSouris`**..

3.5 Prise en compte du click (v05)

Afin de résoudre le problème d'oscillation, on peut

- soit valider un changement de mode uniquement quand le booléen **`getPressed`** passe de vrai à faux

- soit valider un changement en utilisant la méthode **`boolean getClicked()`** qui détecte le click (détection du changement d'appui) et pas l'appui seul du bouton de la souris.

Modifier le changement d'état en utilisant **`getClicked`** et pas **`getPressed`**.

4 Gestion du clavier

De la même manière que pour la souris, le moteur de jeu fournit une classe **`CClavier`** destinée à étudier les actions effectuées au clavier.

4.1 Appui d'une touche (v06)

On souhaite changer de mode toujours quand il y a un click de souris, mais aussi quand une touche est appuyée (par exemple la touche espace).

La classe **`CClavier`** propose la méthode **`boolean isAppuyee(int codeTouche)`** qui retourne un booléen valant **`true`** si et seulement la touche dont le code est passé en paramètre est appuyée. Les codes de touches sont fondés sur les keycode java (disponibles à l'adresse : https://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyEvent.html)

Si on souhaite change de mode avec la touche espace, le code à passer est la constante **`KeyEvent.VK_SPACE`**.

De la même manière qu'avec la souris, on appuie à une oscillation puisque tant que la touche est appuyée, la méthode **`isAppuyee`** renvoie vrai et le mode change.

4.2 Touche tapée (v07)

Pour éviter cela, on utilise une autre méthode qui détecte l'événement correspondant à l'appui d'une touche. Cette méthode est la méthode **`boolean getTyped(int codeTouche)`** de la classe **`CClavier`**.

A noter que quand la touche reste appuyée, java considère que plusieurs touches sont envoyées à l'application (fonctionnement du clavier et du **`KeyListener`**).

5 Structuration du jeu

Maintenant que vous savez comment manipuler les classes qui gèrent les interactions, cette partie va se concentrer sur la manière de structurer simplement une application.

5.1 Séparation des dessin et des modèles de jeu (v08)

Jusqu'à présent, nous avions séparés les deux modes en utilisant un **`switch`**, mais les codes restaient dans la même classe.

L'idée est de faire pour chaque mode de jeu une classe de Jeu et une classe Dessin. Ainsi nous aurons

- pour le mode début, une classe **`JeuDebut`** et une classe **`DessinDebut`**
- pour le mode fin, une classe **`JeuFin`** et une classe **`DessinFin`**

Les classes de jeu mettent à jour les données à partir des actions utilisateurs et en fonction du mode en cours, les classes de dessin affichent les données en fonction du mode de jeu actuel.

La méthode **`evoluer`** de jeu consiste simplement à appeler la méthode **`evoluer`** du bon objet jeu

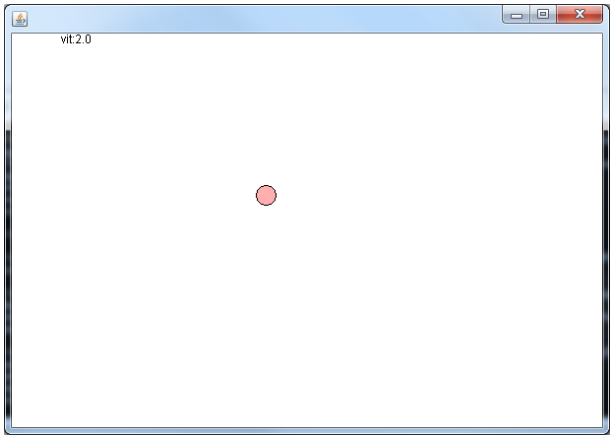
en fonction du mode actuel. Il en va de même pour la méthode dessiner de la classe **`Dessin`**.

Cette structuration permet facilement de gérer des menus, des pauses et des modes de jeu différents : il suffit simplement de créer une nouvelle classe de jeu et de dessin chargées de gérer ce nouveau mode et de modifier les **`switch`** en ajoutant ces modes.

5.2 Ajout jeu simple (v09)

Dans cette partie, on vous demande de créer un jeu et un dessin très simples qui correspondront à un nouveau mode :

- Le jeu a pour attributs les coordonnées d'un cercle et met à jour ces coordonnées en fonction des touches directionnelles appuyées par le joueur (**`VK_LEFT`**, **`VK_RIGHT`**, **`VK_UP`**, **`VK_DOWN`**)
- L'affichage affiche simplement un cercle ayant pour centre les coordonnées stockées dans l'objet jeu.



On passe à ce mode de jeu « **`PJ`** » quand on est au mode début et qu'on clicke sur le rectangle. Une fois entré dans ce mode de jeu, on n'en sort plus.

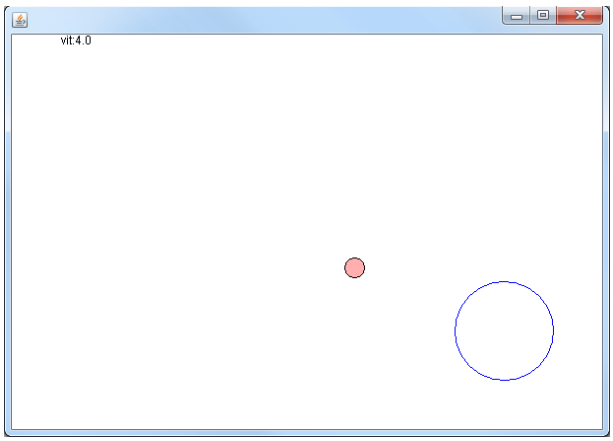
5.3 Modifier la vitesse de déplacement (v10)

Jusqu'à présent, on déplaçait le personnage avec une vitesse fixe. Dans cette partie, on va ajouter un attribut vitesse correspondant à la vitesse de déplacement.

Modifier le jeu pour modifier la vitesse de déplacement avec les touches A et Q.

5.4 Fin du jeu Personnage (v11)

En utilisant la structuration en mode, on va ajouter une fin de jeu. Cela consiste à ajouter un objectif dans le mode de jeu « **`jeu`** » qui sera dessiné à l'écran. Lorsque le personnage passe à coté de cet emplacement, le jeu passe en mode « **`fin`** ».



Le mode fin permet de revenir au mode début d'un click de souris.

6 Gestion des sprites du jeu

Cette partie montre comment tirer partie de planches de sprite à l'aide des classes fournies dans le moteur.

6.1 Gestion d'un sprite (v12)

On fournit une feuille de sprite contenant un seul sprite (« megaman-solo.png »).

Pour charger et accéder aux sprites, il suffit d'utiliser la classe **`Sprites`** et les méthodes statiques qu'elle possède.

Le chargement d'un simple sprite se fait avec la méthode **`void chargerImage(String nom, String nomFichier)`** qui prend en paramètre

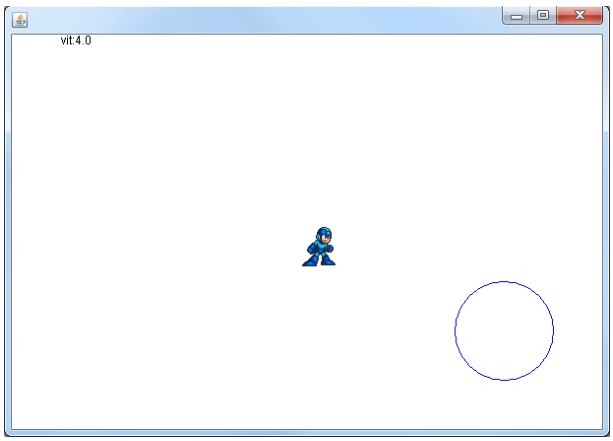
- **`nom`** le nom pour identifier le sprite
- **`nomfichier`** le nom du fichier contenant le sprite.

Le chargement n'a besoin d'être fait qu'une fois. Il est conseillé de faire ce chargement à la création des objets **`Dessin`** pour s'assurer de cela.

Un fois le sprite chargé, il suffit d'appeler dans les classes de **`Dessin`** la méthode **`void dessinerCentre(Graphics g, String nom, int x, int y)`** qui prend en paramètre

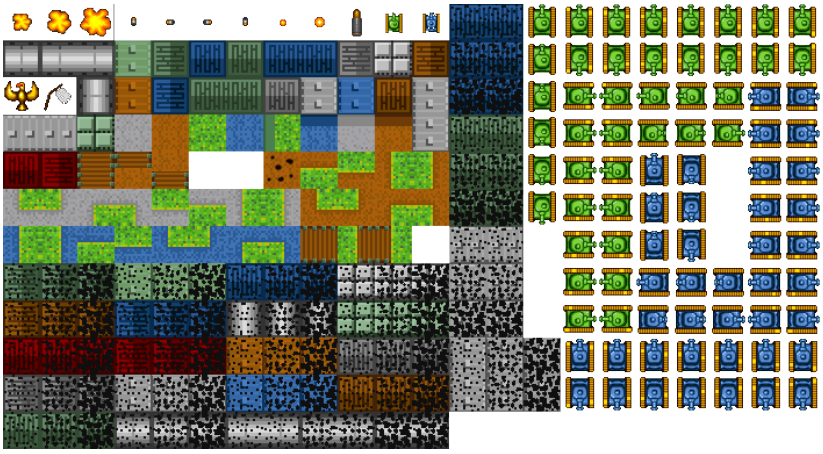
- le **`graphics g`** correspond au graphics de l'image dans laquelle dessiner
- nom désigne l'identifiant du sprite
- **`x`** et **`y`** désignent à quel endroit dessiner le sprite (x, y) correspond au centre du dessin du sprite.

Remplacer l'affichage du cercle du mode « **`jeu`** » par le personnage megaman.



6.2 Gestion d'une feuille de sprites (v13)

Dans certains cas, une image contient plusieurs sprites répartis de manière uniforme (cf feuille tank_tiles)



Il est possible de charger tous les sprites d'un coup à l'aide de la méthode **`void chargerFeuille(String nomRacine, String nomFichier, int nx, int ny)`** de la classe **`Sprite`**.

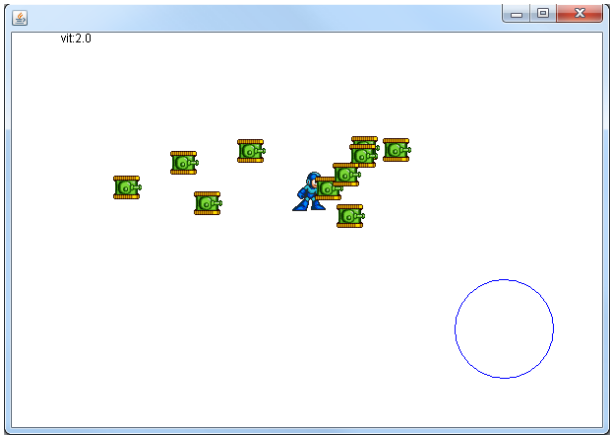
Cette méthode charge la feuille et construit différents sprites en découpant régulièrement la feuille :

- **`nomRacine`** désigne le début des identifiants des sprites qui seront complétés par les coordonnées du sprite
- **`nomFichier`** désigne le fichier image
- **`nx`** et **`ny`** désignent les tailles d'un sprite, c'est à partir de ces valeurs que la planche de sprites sera découpée.

La méthode pour dessiner les sprites est toujours la méthode **`void dessinerCentre(Graphics g, String nom, int x, int y)`**, l'identifiant à fournir correspond à la chaine « **`nomRacine_i_j`** » pour dessiner le sprite (i,j) de la planche.

Par exemple si la feuille est lue avec l'id « tank », « tank_0_0 » désignera le sprite en haut à gauche (début d'explosion) et « **`tank_2_0`** » désignera la grosse explosion.

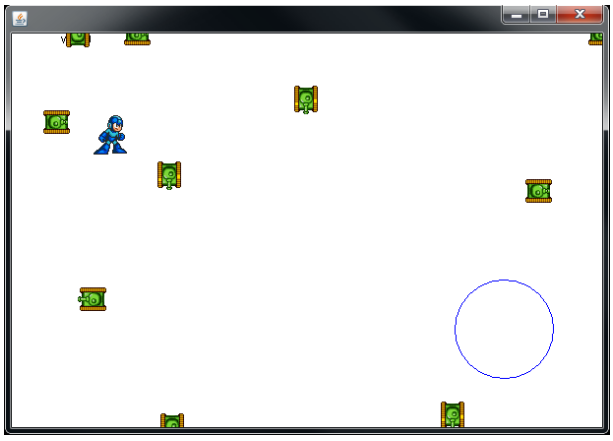
Remplacer le sprite de megaman par un tank simple. Ensuite, modifiez le jeu et les dessin pour ajouter des tanks fixes. En fonction de l'ordre dans lequel se fait l'affichage des sprites, ceci seront en dessus ou en dessous les uns des autres (ici megaman est dessiné avant les tanks).



6.3 Déplacement des tanks (v14)

Une fois que ces éléments sont posés, il est facile de déplacer les tanks en ajoutant des méthodes d'évolution appelées chaque fois le jeu évolue.

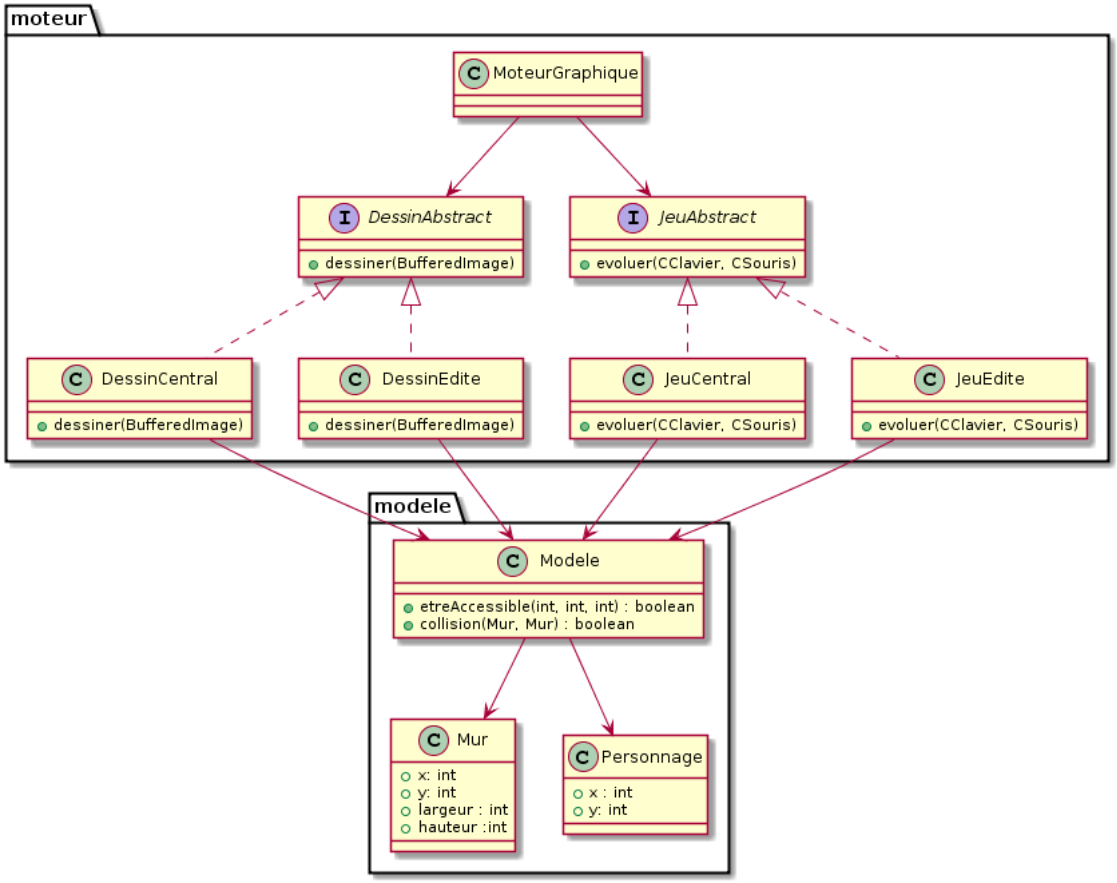
Un premier cas simple est de faire des déplacements aléatoires, mais il est possible de faire mieux par exemple en choisissant une direction aléatoire, en avançant selon cette direction et en modifiant légèrement cette direction avec un facteur aléatoire. Le sprite affiché pourra être différent en fonction de la direction suivie par le tank.



7 Gestion d'information commune (v15)

Dans certains cas, il peut être utile d'avoir de l'information commune à plusieurs modes de jeu car ces modes de jeu correspondent au même jeu.

Le plus intéressant consiste à partager les informations du jeu dans une classe **`Modele`** destinée à contenir les données du jeu. Cette classe peut ainsi être accessible à tous les modes de jeux et à tous les dessins.



On souhaite ainsi un jeu de déplacement dans un labyrinthe avec trois modes différents

- un menu au début
- un mode « **`central`** » dans lequel on déplace un personnage, on passe en mode « **`edite`** » en appuyant sur « **`escape`** »
- un mode « **`edite`** » dans lequel on peut modifier le labyrinthe, on revient en mode jeu en appuyant sur « **`escape`** »

Les modes « **`central`** » et « **`edite`** » ont tous les deux besoins des informations du jeu : la position du personnage et le labyrinthe. La classe **`Modele`** contient ces données.

Le mode « **`central`** » consiste

- a prendre en compte les touches de déplacements
- à déplacer le personnage s'il n'y a pas collision avec un mur
- à revenir au mode « edit » si on détecte l'appui sur touche escape

Le mode « **`edite`** » consiste

- à étudier les actions faites à la souris : un drag and drop permet de créer un mur et de l'ajouter aux murs du modèle.
- à revenir au mode « jeu » si on détecte l'appui sur touche escape

Les deux affichages sont légèrement différents mais ont une base commune.

Globalement

- Les classes Jeu ont pour objectif de manipuler et de modifier les données dans Modele
- Les classes dessin ont pour objectif d'afficher les données contenues dans Modele