Robert HANSANA

Pierre BAUDOT

Xavier MIGNOT

Rapport

Projet de fin d'études

Animation de programmes avec sémaphores

Professeur : Jean-Michel COUVREUR

Licence Informatique

Sommaire

1 Introduction

Dans le cadre de notre troisième année de licence informatique en ingénierie informatique à l'université d'Orléans, il nous est proposé un projet ,pour ceux n'ayant pas eu de stage ou l'ayant perdu, nous permettant de mettre en pratique nos connaissances et nos compétences face à un projet ayant pour finalité la conception et le développement d'une API servant à animer les sémaphores.

Lors de la réception de notre sujet, notre seul directive et demande de la par de notre professeur a été de pouvoir voir des animations de sémaphore. Et pour le reste nous avions carte blanche.

La problématique à laquelle nous faisions face prenait le forme de cette « liberté », en effet, nous devions penser, conceptualiser et réaliser un programme de A à Z sans idées précises de ce que devais être le produit final.

2 Concept de l'API

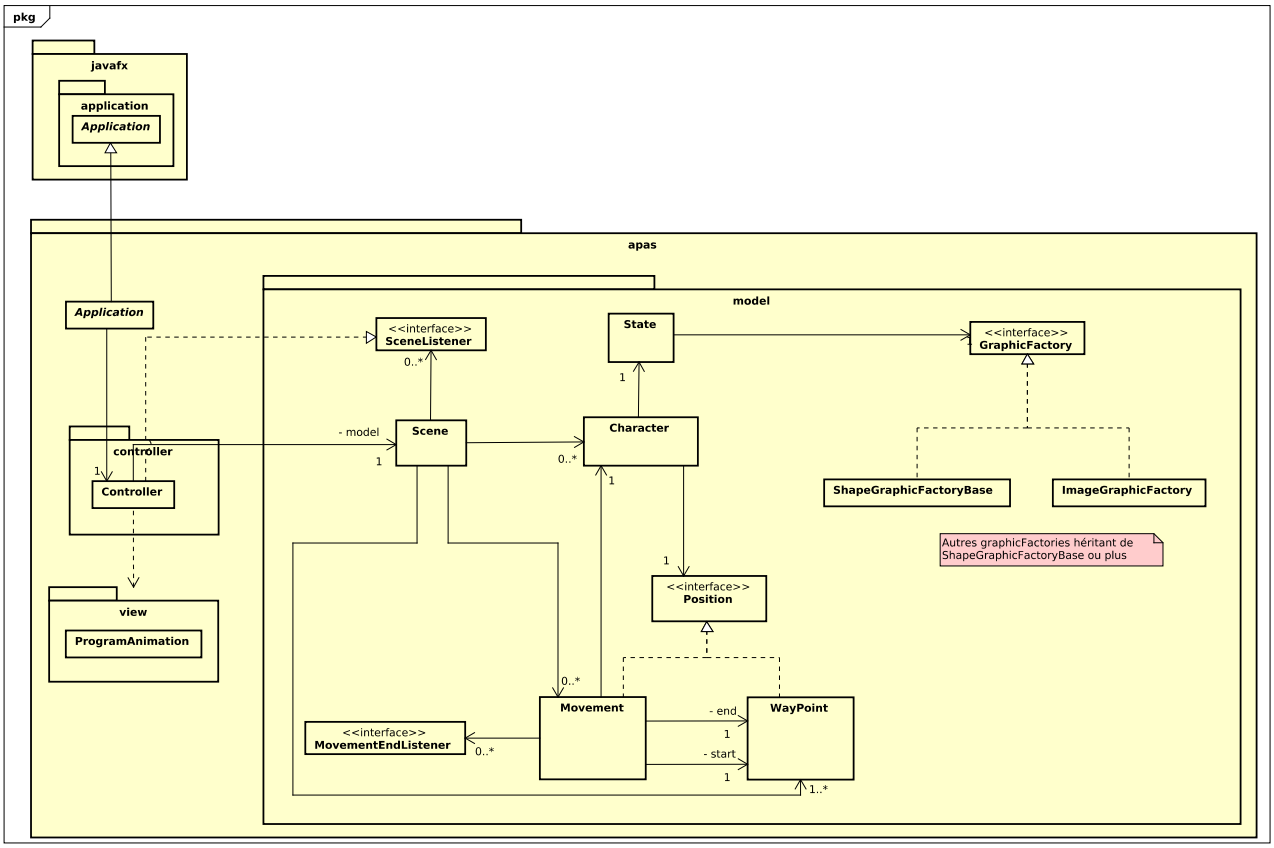
L'idée est que l'utilisateur devra faire hériter sa classe principale de la classe Application du package apas, et exactement comme à la manière de javafx avec la classe javafx.Application, l'utilisateur devra initialiser et manipuler les animations a partir d'un objet qui lui sera donné, dans javafx cet objet est un Stage, dans apas cette objet est un Scene (Attention pas celui de javafx, celui du package apas.model) qui sert a représenter la scène ou vont se dérouler les annimations et a manipuler les personnages (classe Character) qui vont se déplacer et changer d'états dessus.

3 Spécificités Techniques

Dans cette partie, nous allons voir plus en détails la structure de l'API.

3.1 Diagramme UML

Voici une version simplifiée du diagramme, vous pourrez en trouver une version complète et exhaustive à la racine du dépôt.



3.2 Point d'entrée de l'application

Le point d'entrée pour un programme APAS est la classe Application.

init(Map<String,WayPoint>) doit initialiser les balises entre lesquels vont se déplacer les personnages ainsi que les états possibles qu'ils peuvent avoir.

mainThread(Scene), est l'endroit où vous devez écrire votre programme principal, initialiser et lancer les différents threads dont vous avez besoin, etc. Elle doit créer et animer les personnages. Elle sera exécutée dans un thread different de celui de l'interface graphique.

Quand l'appel à mainThead(Scene) est terminé, l'application n'est pas fermée, vous devez fermer la fenêtre pour cela.

Notez que mainThread(Scene) et init(Map<String, WayPoint>,Map<String,State>) sont abstraite et doivent obligatoirement êtrent redéfinie.

Lors de la fermeture de la fenêtre contenant l'annimation, par défaut, tous les thread de l'application sont détruits "brutalement". Si vous voulez libérer des ressources proprement lors de la fermeture de la fenêtre redéfinissez la méthode stop() qui sera appelé lors de la fermeture de la fenêtre ou lors de l'appel à Platform.exit().

D'ailleurs, pour stopper totalement l'application, utiliser Platform.exit() est préférable à System.exit(int) car sinon la méthode stop ne sera pas appelée.
Il est important que les sous classes de Application doivent être déclarés public et avoir un constructeur sans arguments.

3.3 Le modèle

Description courte de chaque classes :

- **Character** (Personnage): Sert à représenter des personnages, ou toute autre entité pouvant se déplacer d'une balise (WayPoint) à une autre dans une scène (représenté par la classe Scene). Immuable en dehors du package apas.model car les méthodes de mise à jour sont déclarées package private. L'égalité de deux personnages repose uniquement sur leurs noms.

- **WayPoint** (balise): Balise permettant aux Character de se repérer sur une Scene. Immuable. L'egalité de deux WayPoint repose sur leurs noms uniquement.

- **Position** : Position sur une Scene, sert notamment a la classe Character.

- **Movement** : Position mobile, soit un déplacement entre deux WayPoints. Lors de la construction d'un mouvement, celui ci devient automatiquement la nouvelle Position de character, et à la fin de ce mouvement, le WayPoint d'arrivée devient automatiquement la Position du Character concerné. L'égalité de deux mouvements repose sur les WayPoints de départ et d'arrivée, sur le Character ainsi que sur la durée totale du mouvement uniquement. Immuable en dehors du package apas.model car les méthodes de mise à jour sont déclarées package private.

- **State** : État d'un Character. Cette class définit aussi l'apparence qu'aurra un Character dans l'interface grace à une graphicFactory (fabrique de représentations). Les State sont immuable et l'egalité de deux State repose sur leurs noms uniquement.

- **MovementEndListener** : Permet de réagir à la fin d'un Mouvement.

- **Scene** : Représente la scène sur laquelle évoluent les personnages (Character). Il s'agit de la façade du modèle, toute modification du modèle devra se faire par cette classe (elles sont d'ailleurs impossibles autrement). Les dimensions de la scène sont calculées en fonction de la position des WayPoint les plus éloignés du point (0,0). Cette classe est thread-safe.

- **SceneListener** : Permet de réagir aux changements qui surviennent dans Scene.
  ATTENTION, Les méthodes de SceneListener peuvent être appelés depuis des thread différents, faites donc très attention à la synchronisation. Note: En javafx, toutes les opérations sur l'interface graphique doivent être faite depuis le JavaFX application thread, pour cela utilisez Platform.runLater(Runnable) ou utilisez les classe Task et Worker dans l'implémentation des méthodes de SceneListener).

3.4 La vue

La vue, c'est-à-dire la classe ProgramAnimation, gère elle la fenêtre d'affichage et les animations tels que : afficher les points de départs et d'arrivés, afficher et supprimer la représentation graphique des Character et faire se déplacer les Character .

3.5 Le contrôleur

Le contrôleur, lui sera réactif au changements du modele initialisés par les utilisateurs. En effet, le contrôleur implémentera l'interface SceneListener et de ce fait agira en fonction de ce qui est notifié lors d'un changement sur l'objet Scene que manipule l'utilisateur.

4 Guide de démarrage

Cette partie est destinée à vous montrer comment installer l'environnement pour créer et lancer une application utilisant l'API APAS au travers d'un exemple sans rentrer dans les détails.

Veuillez vous référer à la javadoc pour plus de détails.

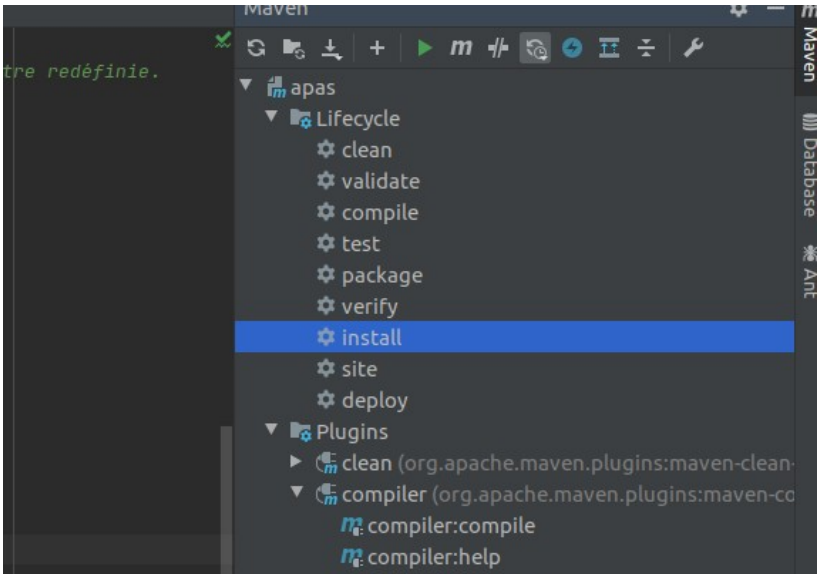
Le code complet de l'exemple ci-dessous est disponible sur le dépôt https://o2171252@pdicost.univ-orleans.fr/git/scm/as/apas_exemple_pont.git

4.1 Procédure d'installation du package apas

1) Clonez le dépôt apas

https://o2171252@pdicost.univ-orleans.fr/git/scm/as/apas.git

2) Installer apas avec maven. (fait avec IntelliJ)



si vous n'avez pas intellij, utilisez simplement la commande mvn install en vous plaçant à la racine du projet.

4.2 Votre premier programme

Créez un nouveau projet maven pour la nouvelle application avec un pom.xml ayant la structure suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                            http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>myApplication</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>fr.orleans.univ.apas</groupId>
            <artifactId>apas</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <release>11</release>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.openjfx</groupId>
                <artifactId>javafx-maven-plugin</artifactId>
                <version>0.0.4</version>
                <configuration>
                    <mainClass>MainClass</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

N'oubliez pas de remplacer l' **artifactId** et le **groupId** au début du fichier avec les infos de votre application, et de remplacer **MainClass** dans la partie **<configuration>** du plugin **javafx-maven-plugin** par le nom de votre classe principale (qui doit hériter de apas.**Application** pour rappel)

1) Faites hériter à votre classe principale la classe abstraite apas.Application.

```
public class PontBis extends apas.Application {
```

2) Redéfinissez la méthode init(Set <WayPoint> points, Set<State> etats).

```
@Override
public void init(Set<WayPoint> points, Set<State> etats) {

}
```

Dans init, vous pouvez initialiser des State qui feront office de représentation graphique, ainsi que les WayPoints qui feront office de points de départ et d'arrivé.

```
@Override
public void init(Set<WayPoint> points, Set<State> etats) {
    points.add(new WayPoint( name: "A",  x: 50,  y: 50));
    points.add(new WayPoint( name: "B",  x: 750,  y: 50));
    etats.add(new State( name: "initial", new State.RectangleGraphicFactory()));
    etats.add(new State( name: "cercle", new State.CircleGraphicFactory()));
}
```

3) Pour pouvoir faire se déplacer des objets,ces mêmes objets doivent hériter de la classe Thread et avoir en attribut la scene.

```
static class Voiture extends Thread {
    private final Scene scene;
    private final String name;
    private final String pointDepart;
    private final SuperInterrupteur interrupteur;
    private final SuperInterrupteur lAutreInterrupteur;

    public Voiture(String name, String pointDepart, Scene scene, SuperInterrupteur interrupteur,
                   SuperInterrupteur lAutreInterrupteur) {
        this.name = name;
        this.scene = scene;
        this.interrupteur = interrupteur;
        this.lAutreInterrupteur = lAutreInterrupteur;
        this.pointDepart = pointDepart;
        scene.addCharacter(name, pointDepart,  state: "initial");
    }
```

4) Pour décrire les actions de ces objets vous pouvez écrire du code dans la fonction run() associée.

```
static class Voiture extends Thread {
    private final Scene scene;
    private final String name;
    private final String pointDepart;
    private final SuperInterrupteur interrupteur;
    private final SuperInterrupteur lAutreInterrupteur;

    public Voiture(String name, String pointDepart, Scene scene, SuperInterrupteur interrupteur,
                   SuperInterrupteur lAutreInterrupteur) {
        this.name = name;
        this.scene = scene;
        this.interrupteur = interrupteur;
        this.lAutreInterrupteur = lAutreInterrupteur;
        this.pointDepart = pointDepart;
        scene.addCharacter(name, pointDepart, state: "initial");
    }

    public void run() {
        lAutreInterrupteur.aVous();
        interrupteur.entre();
        scene.moveCharacter(name, pointDepart.equals("A") ? "B" : "A",
                m -> {
                    scene.updateState(name, newState: "cercle");
                    interrupteur.sort();
                });
    }
}
```

5) Dans le code ci-dessus, vous pouvez définir les actions de votre objet à la fin de son mouvement, ici on demande de changer la forme de notre objet (ici un rectangle) à un cercle.

```
m -> {
    scene.updateState(name, newState: "cercle");
    interrupteur.sort();
});
```

6) Redéfinissez la méthode mainThread(Scene scene).

```
@Override
public void mainThread(Scene scene) {

}
```

Les instructions dans cette fonction seront exécutées après l'initialisation de l'interface.

Par exemple :

```java
@Override
public void mainThread(Scene scene) {
    try {
        for (int i=0; i<4;i++) {
            new Voiture( name: "A"+i, pointDepart: "A",scene, interrupteurA, interrupteurB).start();
            Thread.sleep( l: 7000);
            new Voiture( name: "B"+i,  pointDepart: "B",scene,interrupteurB, interrupteurA).start();
            Thread.sleep( l: 7000);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Le code ci-dessus crée des voitures toutes les 7 secondes et les fait démarrer puis endors le thread actuelle pour changer de thread donc de voiture.