TP Java sur Eclipse

Installation et découverte du simulateur de robotique Simbad

Objectifs du TP

- Etudier la création d'un projet utilisant des bibliothèques java externes existantes sous forme de fichiers .jar dans Eclipse...
- Utiliser une API importée : découvrir et développer sur la plate-forme robotique Simbad...
- Installer et utiliser java 3d en tant que librairie utilisateur...

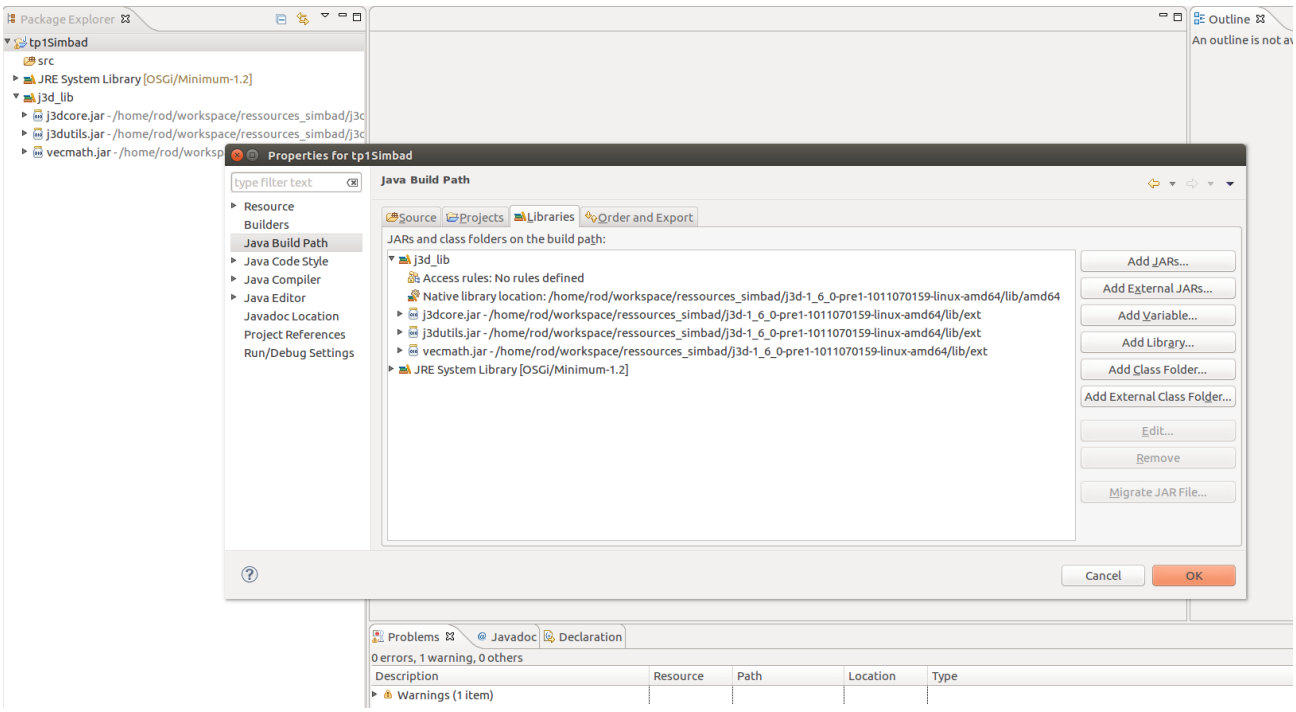
Documentation

Des liens vers la documentation officielle de Simbad sont disponibles sur le cours en ligne Eureka ainsi que les fichiers de bibliothèques à utiliser dans ce TP.

Travail à réaliser

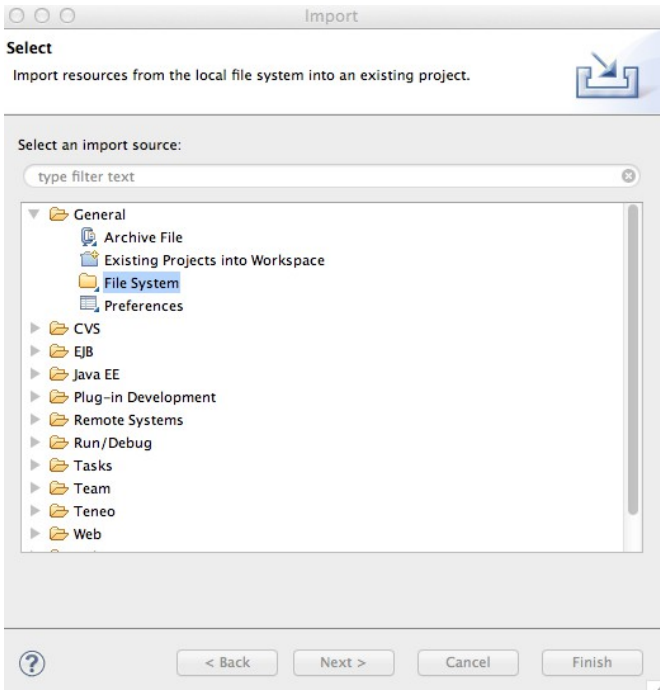
Création du projet et installation de l'API java3D

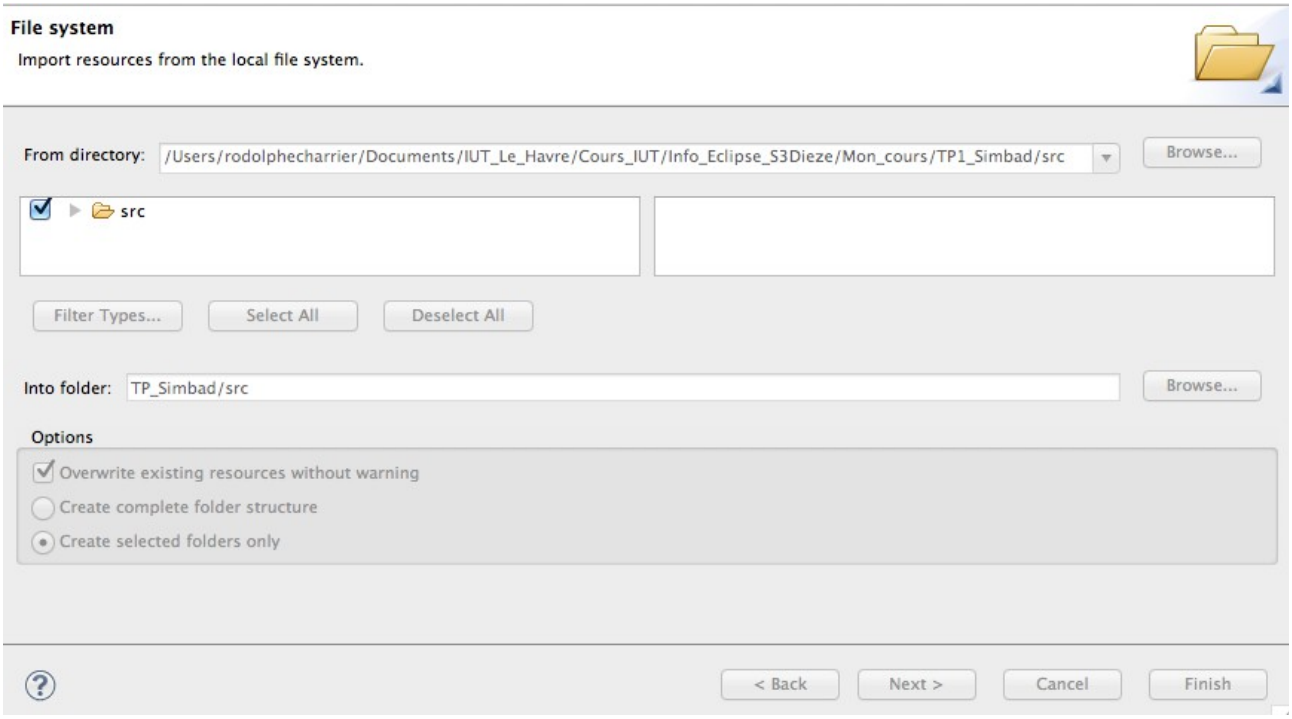
1. Ouvrir Eclipse dans votre workspace java habituel.
2. Créer un nouveau projet **tp1Simbad** dans un dossier de même nom (créé par défaut par Eclipse), avec deux dossiers distincts pour les fichiers sources (src) et compilés (bin).
3. Télécharger sur le cours en ligne Eureka l'archive **j3d-1_6_0-pre1-1011070159-linux-amd64.zip** disponible dans le lien **TP1 – Installation** que vous placerez à la racine de votre dossier workspace Eclipse
4. Décompactez l'archive zip précédente. Vous y trouverez un répertoire **lib** qui contient lui-même un sous-répertoire **ext** contenant les fichiers/archives jar de l'API java3d :
    - Ouvrir les propriétés de votre projet Eclipse dans le menu contextuel du projet tp1Simbad créé précédemment, dans l'onglet des librairies du projet (**Properties/Java Build Path/Librairies)** et créer une nouvelle bibliothèque utilisateur (**Add Library/User Library/User Librairies/New** puis entrer le nom **j3d_lib** et valider) .
    - Dans la fenêtre des librairies utilisateurs, sélectionner la librairie **j3d_lib** et ajouter tous les jars externes (**Add External JARS**) situés dans le répertoire j3d décompacté au point (3) dans le dossier **ext**. Il y en a 3, qui contiennent les classes java pour l'utilisation de la 3D.
    - Validez ensuite l'ensemble jusqu'à revenir à votre onglet **Librairies** des propriétés du projet. Déplier votre librairie **j3d_lib** et sélectionner **Native Library Location** (les librairies natives sont des librairies écrites en langage de bas niveau comme le C ou le langage machine) et cliquer sur le bouton **Edit**. Renseignez alors le chemin vers le dossier **lib/amd64** de votre répertoire java3d décompacté au point (3). Il s'y trouve en effet une librairie native **libj3dcore-ogl.so** à référencer également dans le projet.



Installation des fichiers sources de l'API Simbad dans Eclipse

- *Depuis le cours en ligne Eureka* :
  Télécharger le fichier **simbsrc.tar.gz** situé dans le dossier **TP1 - Installation** du cours en ligne et décompactez-le dans un dossier temporaire TP_Simbad sur votre système.
  (Rappel de la commande pour décompacter : tar -xvf simb.tar.gz)
  Il s'agit d'un dossier comportant les fichiers sources .java de l'API Simbad.
- *Retour dans Eclipse* :
  Sélectionnez le dossier **src** du projet Eclipse précédent **tp1Simbad** et appliquez la commande **Import** du menu contextuel sur ce dossier, qui va vous permettre de rapatrier les fichiers java de Simbad : dans la fenêtre qui s'ouvre alors, sélectionnez l'item **General/File System** puis cliquez sur le bouton Next de la fenêtre. Indiquez alors dans le champ **From Directory** le chemin vers le dossier **TP_Simbad/src** que vous avez décompressé précédemment, puis cochez la case du dossier **src** qui s'affiche alors dans le cadre en dessous. Enfin validez.

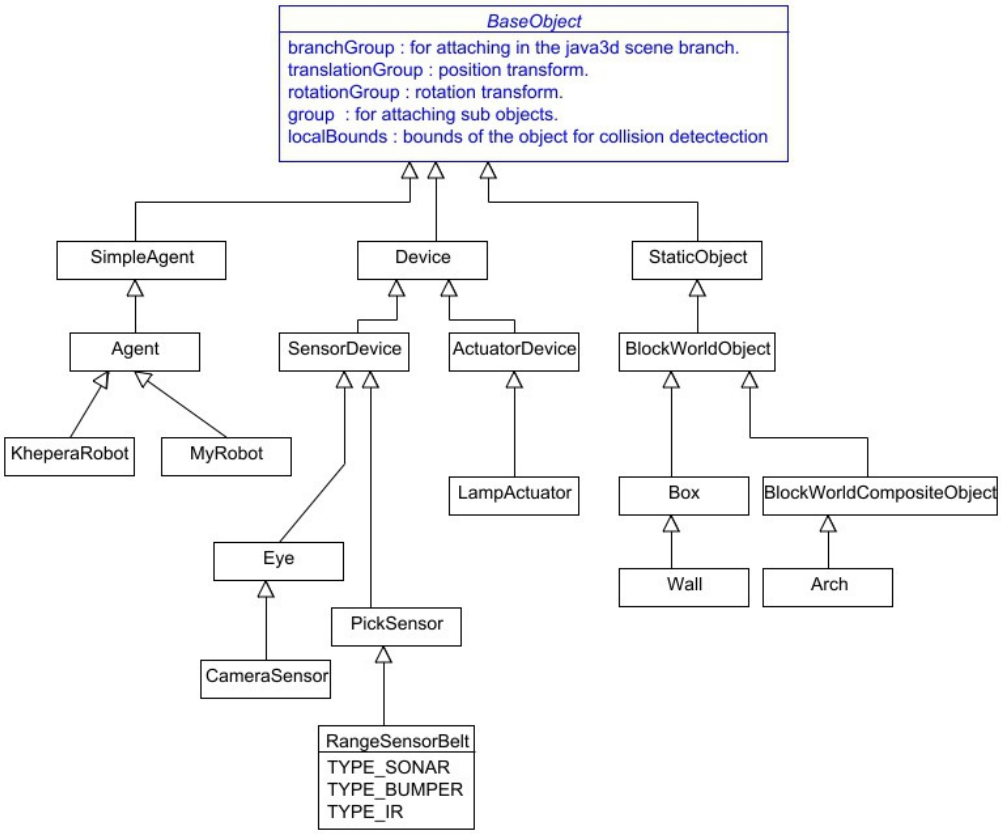




- Vous devez voir apparaître un ensemble de packages dans le dossier **simbsrc** du projet qui regroupe l'ensemble des fichiers java de Simbad.
- Consultez ces packages et leurs classes. Analysez la structure des packages. Utilisez aussi la documentation officielle de Simbad disponible en lien dans le cours.
- Exécuter la classe **Simbad.java** du package **simbad.gui** pour tester votre configuration et découvrir l'interface.
- Ouvrir le fichier **BaseDemo.java** et analyser sa structure : comment est déclarée la classe Robot et pourquoi est-elle placée ici... ?

Rem : on aurait pu aussi utiliser uniquement l'archive/librairie **simbad.jar** disponible dans le dossier TP1-installation au lieu d'installer les sources. Cependant, nous serons amenés à modifier/ajouter certaines classes dans les packages d'origine de Simbad dans les derniers TP pour changer les éléments de l'interface.

Réalisation d'un premier exemple et test de Simbad :



Simbad est fondé sur quelques classes de base schématisées dans le diagramme UML ci-dessus.

- Créer un nouveau package **ex1** dans le projet importé précédent au même niveau que les autres packages et créer un nouveau fichier java **Exemple1.java** comprenant la classe de même nom qui inclut une méthode principale **main**; cette méthode crée la classe principale pour une simulation Simbad :

```
public class Exemple1 {
        public static void main(String[] args) {
        Simbad frame = new Simbad(new MyEnv() ,false);
        }
}
```

Des erreurs apparaissent. Pourquoi ?
Avec l'aide en ligne d'Eclipse, importer ce qui manque. Cependant il reste des erreurs car il manque des classes, lesquelles ?

- Nous allons compléter ce code par deux autres classes dans deux fichiers placés dans le même package **ex1**. Dans le code précédent, revenir sur l'erreur située sur **MyEnv** et cliquer sur le pictogramme d'erreur pour déclencher la création de la classe **MyEnv** à compléter avec le code suivant :

```
public class MyEnv extends EnvironmentDescription {
        public MyEnv(){
                add(new Arch(new Vector3d(3,0,-3),this));
                add(new MyRobot(new Vector3d(0, 0, 0),"mon robot"));
        }
}
```

Analyser ce code, notamment en utilisant la commande **Open Declaration** du menu contextuel disponible par clic droit sur les classes nouvelles.

Comme précédemment, utiliser l'aide d'Eclipse pour compléter les importations de classes manquantes signalées par des erreurs.

Puis par la même technique, créer la classe **MyRobot** qui hérite de la classe Agent, dans un autre fichier du même package, puis compléter-la avec le code suivant :

```
public class MyRobot extends Agent {
	public MyRobot (Vector3d position, String name) {
		super(position,name);
	}
	public void initBehavior() {}

	public void performBehavior() {
			// avance à 0.5 m/s
			setTranslationalVelocity(0.5);
			// changer l'angle fréquemment
			if ((getCounter() % 100)==0)
				setRotationalVelocity(Math.PI/2 * (0.5 - Math.random()));
	}
}
```

A chaque pas de temps (renvoyé à tout instant par **getCounter()**), un robot exécute ses méthodes **initBehavior()**, s'il a une spécificité au démarrage, et **performBehavior()** ensuite à chaque pas de temps.
La vitesse du robot est fixée par **setTranslationalVelocity()** pour la vitesse en translation (tout droit) et par **setRotationalVelocity()** pour la rotation du robot.

Compilez et exécutez ce code à partir de la classe Exemple1 qui comporte la classe main.

Gestion événementielle :

- la classe Agent possède une méthode **collisionDetected()** qui renvoie true si l'agent a une collision avec un obstacle.
  Intégrer une gestion liée à cet événement dans la classe précédente pour éviter les collisions avec les objets placés dans l'environnement, en modifiant la vitesse du robot.
- Explorez la classe Agent et ses méthodes grâce à **Open Declaration** et à **Open Type Hierarchy** du menu contextuel.

Ajouter des capteurs

Les capteurs d'un robot sont par exemple une ceinture de sonars (capteurs à ultra-sons) dont voici un exemple de réalisation à tester :

```
public class RobotSonars extends Agent {
	RangeSensorBelt sonars;

	public RobotSonars(Vector3d position, String name) {
		super(position, name);
		sonars = RobotFactory.addSonarBeltSensor(this,8);
	}

	public void performBehavior() {
		//toutes les 20 frames
		if (getCounter()%20==0){
```

```
                // afficher les mesures de chaque sonar
                for (int i=0;i< sonars.getNumSensors();i++) {
                        double range = sonars.getMeasurement(i);
                        double angle = sonars.getSensorAngle(i);
                        boolean hit = sonars.hasHit(i);
                        System.out.println("Sonar at angle "+ angle +
                        "measured range ="+range+ " has hit something:"+hit);
                }
        }
    }
}
```

Quelle classe réalise cette création de capteurs ?

Combien de sonars constituent la ceinture de capteurs ici ?

Utilisez l'API pour construire une autre classe de robots RobotBumpers semblable à la précédente utilisant des capteurs de contacts ('bumpers').

Vous pouvez également ajouter une caméra au robot qui peut renvoyer une image de ce que voit le robot sur la console de contrôle. Pour cela placer dans le constructeur du robot l'instruction suivante :

```
// ajoute la caméra sur le robot
camera = RobotFactory.addCameraSensor(this);
```

NB : Attention, cette fonctionnalité caméra n'est visiblement pas supportée par tous les pilotes de cartes graphiques et peut faire échouer la simulation Simbad (possible sur les postes plus anciens dans certaines salles...)!! Pour cette raison, cette fonctionnalité ne sera pas exploitée dans les TP suivants.

Pour finir, mettez les trois robots créés jusqu'ici ensemble dans le même environnement.

Réfléchir à des solutions pour résoudre les problèmes observés/rencontrés.