

### Table des matières

Introduction à la programmation et à la conception orientée objet.....	1
1. Déroulement des séances de TPs.....	5
1.1. Organisation des TPs.....	5
1.2. Sujet d'étude.....	5
1.3. Rôle des assistants de TP.....	5
2. Acquisition des compétences.....	6
Première partie : Les piliers de la POO.....	7
3. Installation de l'environnement de travail.....	7
3.1. Sur les machines de l'école.....	7
3.2. Sur votre machine personnelle.....	7
3.2.1. Installation d'un jdk (+jre inclus).....	7
3.2.2. Installation d'un IDE (eclipse).....	8
3.3. Installation de la documentation.....	8
3.3.1. Mode opératoire :.....	8
3.3.2. Bénéfices :.....	8
3.3.3. Behind the stage.....	9
3.3.3.1. Considérations annexes (et pour référence).....	9
3.3.3.2. Considérations associées.....	9
4. Premiers pas en objet : définition d'un modèle (une abstraction) d'Animal.....	9
4.1. Prise en main.....	9
4.1.1. Créez sous eclipse un projet java « simulateur ».....	9
4.1.2. La classe est-elle correctement commentée ?.....	10
4.1.3. Le code respecte-t-il les bonnes pratiques ? Pourquoi ?.....	10
4.1.4. Attributs.....	10
4.1.5. Constructeurs.....	10
4.1.6. Création des accesseurs et des mutateurs [cours 64].....	10
4.1.7. Programmation des méthodes d'Animal.....	11
4.1.7.1. La méthode vieillir.....	11
4.1.7.2. La méthode seDeplacer.....	11
4.1.7.3. La méthode getUniqueId.....	12
4.2. Spécialisation d'Animal par rapport à sa classe mère Object.....	12
4.2.1. Redéfinition de comportements par défaut hérités d'Object.....	12
4.2.1.1. Représentation d'un Animal sous forme de chaîne de caractère.....	12
4.2.1.2. Comparaison de deux instances d'Animal.....	13
5. Générez la documentation de votre projet en utilisant votre IDE.....	13
6. Héritage et polymorphisme.....	13
6.1. Nouvelles contraintes.....	13
6.1.1. Définition des acteurs.....	13
6.1.2. Les agents.....	14

6.1.3. Les éléments de décor.....	14
6.2. Mise en place de la hiérarchie des agents.....	15
6.3. Conception d'une hiérarchie d'agents.....	15
6.3.1. Travail préliminaire d'analyse sur papier.....	15
6.3.2. Réutilisation du travail précédent.....	15
6.3.3. Classe de test.....	15
6.3.4. Animal et Agent.....	16
6.3.5. Hiérarchie complète d'agents.....	16
6.4. Décor.....	16
7. Un peu d'algorithmie !.....	16
7.1. cycle de simulation.....	16
7.2. fonctionnalités spécifiques.....	17
8. Les collections (1ère partie).....	18
8.1. Objectifs pédagogiques.....	18
8.2. Mise en place.....	18
8.2.1. Nouveau projet.....	18
8.3. Compléter le projet avec votre existant.....	18
9. Échauffement sur les collections.....	19
9.1. Aggraver ou améliorer un état de santé.....	19
9.1.1. Étude de aggraverEtat().....	19
9.1.2. Codage de ameliorerEtat().....	19
9.2. Liste de proies du Frelon.....	20
9.2.1. Problématique.....	20
9.2.2. Codage.....	20
9.3. Population d'une ruche.....	20
9.3.1. Choix de la collection.....	21
9.3.2. Algorithmie.....	21
10. Le monde des abeilles.....	22
10.1. La classe Monde.....	22
10.1.1. prise en main.....	22
10.1.2. composition aléatoire du monde.....	22
10.2. Compérateurs.....	22
10.2.1. Comparable.....	22
10.2.2. Comparator.....	22
11. Bilan intermédiaire.....	23
12. Les collections (2ème partie).....	23
12.1. Finalisation du Monde.....	23
12.2. Implémentez les fonctionnalités suivantes.....	23
12.2.1. Rencontre entre Agent.....	23
12.2.1.1. Première approche naïve.....	23
12.2.2. Cycle complet du monde.....	23
12.3. Mode « nuit ».....	24
12.3.1. Mode nuit et Hebergeurs.....	24
12.3.2. Cas particulier de la Ruche.....	24
12.4. Liste des fonctionnalités du logiciel à implémenter (Rappel et nouvelles).....	24
Deuxième partie : objectifs complémentaires.....	26
13. Objectifs.....	26
14. Héritage et abstraction : les classes abstraites.....	26
15. Techniques et mises en œuvre s'appuyant fortement sur la POO.....	26
15.1. Clonage : comment améliorer l'encapsulation.....	26
15.1.1. Clonage d'un Point.....	26

15.1.2. Clonage d'un agent.....	27
15.2. Template Method canonique.....	27
16. Étude du code.....	27
16.1. Génération automatique des Agents du Monde.....	27
17. Nouvelles fonctionnalités.....	28
17.1.1. Arbres Hebergeur.....	28
17.2. Santé des agents.....	28
18. Simulateur graphique.....	28
18.1. Mise en place.....	28
18.1.1. Travail préliminaire au dessin des images.....	28
18.1.2. Étude du code.....	29
18.2. Panneaux extérieurs.....	29
18.2.1. taille des panneaux.....	29
18.2.2. Panneau de commande (en vert à droite).....	30
18.2.3. Panneau d'information (en bleu en bas).....	31
18.3. Panneau central : animation.....	31
18.3.1. Création du panneau.....	31
18.3.2. Testez !.....	32
Troisième partie : pour aller plus loin.....	33
19. Objectifs.....	33
20. Améliorer les faiblesses de conception.....	33
20.1. Adapter une classe existante.....	33
21. Nouvelles fonctionnalités.....	33
21.1. Le problème des Hebergeurs.....	33
21.2. Cartographie du monde.....	34
22. La classe Monde Animable.....	34
22.1. Classes/interfaces impliquées.....	34
22.2. Schéma des collaborations entre Monde et WorldFrame.....	34
23. Amélioration du client graphique.....	35
Annexes.....	36
24. Diagrammes de classe.....	36
24.1. Section 8, progression normale.....	36
24.2. Section 8, progression avancée.....	37
.....	37
25. Synthèse des objectifs pédagogiques.....	38
25.1. Piliers (et quelques principes) de la POO.....	38
25.1.1. Abstraction.....	38
25.1.2. Encapsulation.....	38
25.1.3. Héritage.....	38
25.1.4. Polymorphisme.....	38
25.1.5. Principes de la POO (introduction).....	38
25.2. Techniques et mise en œuvre en java.....	39
25.3. Bonnes pratiques.....	39
25.3.1. codage.....	39
25.3.2. documentation.....	39
25.3.3. IDE.....	39
25.4. Conception.....	40
26. Crédits et illustrations.....	40
26.1. crédits.....	40
26.2. illustrations.....	40
26.3. licences utilisées :.....	40



# 1. Déroulement des séances de TPs


## 1.1. Organisation des TPs


Les TPs sont progressifs et regroupés en 3 parties.


**La première partie** comprend les **compétences indispensables** à acquérir durant ce module. **Elle doit être totalement maîtrisée** pour valider les objectifs du module (validation à 10/20 max).

**La deuxième partie** introduit des **compétences plus spécifiques** : certaines de ces compétences peuvent simplement être abordées, d'autres approfondies suivant vos goûts et motivations. Elle permet d'augmenter la maîtrise des compétences visées en première partie (validation à 12/20) et d'augmenter ses connaissances en POO (validation à 14/20).


**La troisième partie** enfin approfondit encore plus les compétences précédentes, et a pour objectif de vous faire **progresser au-delà des attentes du module** (validation >16). Elle vous challenge sur vos acquis, vous incite à explorer un large champ des possibilités de la POO, et vous permettra d'acquérir des compétences utiles si vous souhaitez poursuivre votre formation dans des spécialités où l'objet est très utilisé (Info, Robotique, Image, Embarqué par exemple...).

Chaque activité détaille en première partie les objectifs pédagogiques à atteindre, puis précise en encadré pour chaque partie () ce qui va être modélisé et programmé.

En cas de difficultés, des activités complémentaires () sont proposées afin de vous permettre de mieux appréhender les concepts utilisés (section « un autre angle d'approche »).

Enfin, chaque activité se termine par des demandes « ouvertes » () où vous pourrez exprimer toute votre créativité !

## 1.2. Sujet d'étude

 Le sujet d'étude pour ce module consiste en la mise en place d'une simulation de type « jeu de la vie » appliquée à l'apiculture.

Nous modéliserons et programmerons une application dont le but final est de réaliser une représentation visuelle du déplacement des abeilles dans leur environnement, avec la prise en compte de contraintes de plus en plus complexes au fur et à mesure des TPs.

Ces contraintes vous seront données progressivement, afin de ne pas compliquer la compréhension. Elles sont identifiées dans un encadré comme celui-ci.

## 1.3. Rôle des assistants de TP

Comme leur nom l'indique, les **assistants de TP sont là pour vous aider à progresser, pas pour vous faire le travail à votre place.**

Ils peuvent vous indiquer des ressources, vous dépanner sur un problème de syntaxe, reformuler un point technique, vous challenger sur une compétence ou une activité s'ils sentent que vous avez la capacité d'aller plus loin. Ils n'ont pas vocation à faire un cours ou pallier le fait que vous n'avez pas préparé une séance de TP correctement, pas lu une documentation ou déjà essayé de régler un problème vous-même !

N'hésitez pas cependant à les solliciter ! **Nous sommes là pour vous aider, pas pour évaluer !**

## **2. Acquisition des compétences**

Chaque activité a pour objectif l'acquisition d'une ou plusieurs compétences du référentiel dont la synthèse est donnée en fin de sujet (section 25) et sur la grille d'acquisition des compétences du module (document annexe).

Votre objectif est bien l'acquisition de toutes les compétences indispensables, ainsi que du maximum de compétences complémentaires et d'approfondissement.

Cette acquisition peut se faire en réalisant une activité du TP, mais aussi par tout autre moyen mis à votre disposition (cours, documentation, TPs annexes, QCM d'auto-évaluation...) ou que vous aurez trouvé vous-même.

Afin de faciliter le suivi de votre progression et l'évaluation finale, vous devrez mettre à jour à chaque séance votre grille d'acquisition des compétences et la remettre en fin de module. Le graphe suivant vous donne également une organisation des compétences faisant apparaître les prérequis nécessaires à l'acquisition de chaque compétence.

# Première partie : Les piliers de la POO



## 3. Installation de l'environnement de travail

Nous allons avoir besoin de trois composants pour pouvoir programmer en java :

1. une machine virtuelle java (commande java) pour exécuter les programmes java (JRE, java runtime environment)
2. un kit de développement java (JDK, java development kit) contenant les outils de développement et notamment un compilateur (commande javac) et un générateur de documentation (commande javadoc) ; le jdk inclus un JRE
3. pour faciliter le développement, un environnement de développement intégré (IDE, integrated development environment) ; nous utiliserons **eclipse**<sup>1</sup> sur les machines de l'école et dans ce document, mais vous pouvez utiliser l'IDE de votre choix.

### 3.1. Sur les machines de l'école

Eclipse et le jdk sont déjà installés. Vous pouvez passer tout de suite à la section 3.3.

### 3.2. Sur votre machine personnelle

Quelque soit votre système d'exploitation, vous pouvez installer le jdk java et eclipse.

Vous pouvez suivre le tutoriel d'openclassroom : <https://openclassrooms.com/fr/courses/6106191-installez-votre-environnement-de-developpement-java-avec-eclipse/6250076-installez-un-jdk-et-eclipse>

#### 3.2.1. Installation d'un jdk (+jre inclus)

<https://openjdk.java.net/install/>

Tester l'installation en exécutant la commande dans une console<sup>2</sup> :

```
javac -version
```

Elle devrait vous répondre en vous indiquant la version du jdk utilisé.

---

<sup>1</sup> <https://www.eclipse.org/>

<sup>2</sup> pour ceux qui persistent à rester sous windows et qui ne savent pas afficher l'invite de commande MS DOS : <https://lecrabeinfo.net/ouvrir-linvite-de-commandes-sur-windows-10-8-et-7.html> (consulté en nov. 2020)

### 3.2.2. Installation d'un IDE (eclipse)

Les procédures sont détaillées sur le site d'eclipse : <https://www.eclipse.org/eclipseide/>

Tutoriel d'installation : <https://www.eclipse.org/downloads/packages/installer> ou <https://openclassrooms.com/fr/courses/6106191-installez-votre-environnement-de-developpement-java-avec-eclipse/6250076-installez-un-jdk-et-eclipse>

Choisissez bien *eclipse for java developers*.

### 3.3. Installation de la documentation

#### 3.3.1. Mode opératoire :

Après avoir lancé Eclipse :

1. Menu **Window** > **Preferences** (pour mémoire, sur Ubuntu, les menus de l'application qui a le focus sont tout en haut de votre écran)
2. Section **Java** > **Installed JREs** (dans la partie de gauche des préférences)

Sélectionner l'entrée **java7-openjdk-amd64**

Bouton **Edit...**

Dans la liste "*JRE system libraries*", sélectionnez **rt.jar** (c'est la 2ème entrée de la liste)  
2 actions à mener pour cette sélection :

1. bouton **Source Attachment...**

Indiquez **External location** et choisissez pour **External File...** :

[/softwares/INFO/jdk1.7.0\\_67/src.zip](#)

(validez par **OK**)

2. (cette 2ème action n'est pas nécessaire avec le src.zip indiqué ci-dessus, qui inclut la documentation dans les sources sous forme de javadoc)

bouton **JavaDoc Location...**

Sélectionnez **Javadoc URL** et indiquez :

<http://docs.oracle.com/javase/7/docs/api/>

(validez par **OK**)

(validez avec **Finish**)

(validez avec **OK**)

#### 3.3.2. Bénéfices :

- Au survol d'un type d'objet défini par la librairie standard (par exemple **String**), ou de ses méthodes, vous avez maintenant l'affichage de sa JavaDoc. Idem lors de l'auto-complétion (CTRL+SPACE), etc...
- Control + clic vous permet d'afficher le code source (d'une classe, d'une méthode) de la librairie standard



### 3.3.3. Behind the stage

#### 3.3.3.1. Considérations annexes (et pour référence)

Ces préférences sont stockées dans votre *workspace*, dans le fichier suivant :

(workspace)/.metadata/.plugins/org.eclipse.core.runtime/.settings/  
org.eclipse.jdt.launching.prefs

#### Conservez le même workspace d'un TP à l'autre !


Un workspace peut contenir plusieurs projets, indépendants ou liés. Dans notre contexte, où l'on travaille dans le même environnement (POO Java), il est parfaitement inutile d'en avoir plusieurs.

#### 3.3.3.2. Considérations associées

- vous n'avez plus qu'à répéter le mode opératoire si vous changez de *workspace*...
- ...à moins de copier ledit fichier d'un *workspace* à l'autre

(l'auteur de ces lignes n'a pas testé et ne garantit pas l'absence d'éventuels effets de bord d'une telle manipulation).

## 4. Premiers pas en objet : définition d'un modèle (une abstraction) d'Animal

 On choisit de modéliser un animal (abeille domestique, parasitaire ou solitaire<sup>3</sup> ; frelon asiatique ou européen<sup>4</sup> ; varroa<sup>5</sup>,...) le plus génériquement possible.

#### Principes mis en œuvre :

**Responsabilité unique** : une classe ne doit avoir qu'une seule raison de changer.

**Ouverture fermeture** : une classe doit être ouverte aux extensions mais fermée aux modifications.

#### Concepts :

classe, instance, encapsulation, constructeur, accesseurs, mutateurs

### 4.1. Prise en main

#### 4.1.1. Créez sous eclipse un projet java « simulateur »

Importez ensuite les fichiers donnés en annexe (TP1). Sous eclipse, faites « importer un système de fichiers ». Vous devez obtenir une arborescence en **package**.

Ouvrez le fichier *Animal.java* avec votre IDE (**eclipse, intelliJ, netbeans...**) et répondez aux questions suivantes.

3 [https://fr.wikipedia.org/wiki/Abeille#Les\\_abeilles\\_solitaires](https://fr.wikipedia.org/wiki/Abeille#Les_abeilles_solitaires)

4 <https://guepes.fr/frelon-europeen-asiatique.htm>

5 [https://fr.wikipedia.org/wiki/Varroa\\_destructor](https://fr.wikipedia.org/wiki/Varroa_destructor)

#### 4.1.2. La classe est-elle correctement commentée ?

1. Quels sont les différents types de commentaires utilisés ?

(💡) ((re)-lire <http://bruno.mascret.fr/dokuwiki/doku.php/java:bp#commentaires>).

2. Ajoutez les éventuels commentaires manquants, (taper `/**` puis « entrée: eclipse complète pour vous le format de commentaire javadoc).

#### 4.1.3. Le code respecte-t-il les bonnes pratiques ? Pourquoi ?

lire [http://bruno.mascret.fr/dokuwiki/doku.php/java:bp#en\\_programmant](http://bruno.mascret.fr/dokuwiki/doku.php/java:bp#en_programmant)

#### 4.1.4. Attributs

Parmi les attributs :

1. lesquels sont des types construits ?
2. lesquels sont des types primitifs ?

(💡) petit rappel ici : [http://bruno.mascret.fr/dokuwiki/doku.php/java:memoire#les\\_types\\_primitifs](http://bruno.mascret.fr/dokuwiki/doku.php/java:memoire#les_types_primitifs)

3. lesquels sont des attributs d'instance ? de classe ? [cours : 65 et 67-70]
4. que sont les types **Etat** et **Sexe** en java ? Comment s'utilisent-ils ? Où sont-ils déclarés ?  
**Ajoutez une valeur ASSEXUE à SEXE.**
5. **etat** est initialisé directement au moment de sa déclaration : est-ce une bonne idée ?  
Comment pourriez-vous faire autrement ?
6. quel pilier de la POO est mis en œuvre lors de la déclaration des attributs ? Comment ?

#### 4.1.5. Constructeurs

Complétez le constructeur sans paramètre et le constructeur `public Animal(Sexe sexe).`

Exécutez à l'aide de votre IDE la méthode **main** de la classe **animal** pour tester vos constructeurs.

1. quel mot-clef permet de faire référence à l'instance créée ?
2. un constructeur peut-il se servir d'un autre constructeur ?
3. peut-on créer un **Animal** d'âge 25 ? Est-ce une bonne chose ?
4. peut-on créer un **animal** sans lui indiquer de coordonnées ? Comment le programme se comporte-t-il ?

#### 4.1.6. Création des accesseurs et des mutateurs [cours 64]

(📖) Soient les contraintes suivantes :

- un âge ne peut qu'augmenter (on ne sait pas rajeunir...) et peut être consulté et modifié par d'autres classes.
- id, sexe et etat ne peuvent pas être modifiés mais peuvent être consultés par d'autres classes.
- (attention, réfléchissez bien, c'est plus compliqué que ça en a l'air...) une position peut être donnée mais ne peut pas être modifiée.

1. créez les assesseurs et les mutateurs utiles (vous pouvez utiliser votre IDE pour générer automatiquement ce que vous souhaitez).
2. dé-commentez les lignes de test dans la méthode main et vérifiez que tout se passe comme prévu...



Du mal à comprendre ce qui se passe ?

[http://bruno.mascret.fr/dokuwiki/doku.php/java:memoire#les\\_types\\_pointeurs](http://bruno.mascret.fr/dokuwiki/doku.php/java:memoire#les_types_pointeurs)

3. quelle différence fondamentale y a-t-il entre une énumération et un objet classique ?
4. ajouter à la méthode main de quoi tester vos autres assesseurs/mutateurs et vérifier leur bon fonctionnement.
5. pensez à documenter les comportements particuliers !
6. un mutateur ou un assesseur peut-il être privé ? Pourquoi ?

#### **4.1.7. Programmation des méthodes d'Animal**

##### **4.1.7.1. La méthode vieillir**

1. Programmez la méthode vieillir qui ajoute une unité de temps à l'âge de l'animal.

Pensez OBJET lorsque vous modifiez un attribut : une bonne manière de faire, et une moins bonne...

2. un mutateur ou un assesseur peut-il être privé ? Avez-vous changé d'avis ? Pourquoi ?

##### **4.1.7.2. La méthode seDeplacer**

L'objectif est de mettre à jour les coordonnées de l'Animal suite à un déplacement aléatoire simple.

Pour cela, on calcule un dx et un dy de -1,0 ou 1 et on applique les changements sur les coordonnées.

Exemple : dx = -1 ; dy = 0 ; coord.x=coord.x-1 ; coord.y ne change pas

Regarder la documentation en ligne de la classe Math et celle de la classe Point :

<https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Math.html>

<https://docs.oracle.com/javase/8/docs/api/index.html?java/awt/Point.html>

1. Comment peut-on qualifier la méthode random() de Math ?
2. Que renvoie-t-elle comme type ?
3. Qu'est que le transtypage de type primitif ? Comment l'utiliser ici ?

### 4.1.7.3. La méthode getUniqueId

Cette méthode permet d'affecter un numéro unique (identifiant) à un animal.

Deux animaux ne peuvent pas avoir le même id.

1. cette méthode est déclarée statique, pourquoi ?
2. pour fonctionner, vous allez avoir besoin d'utiliser quelque chose de la classe Animal. De quoi s'agit-il ?



Indice si vous bloquez : <http://bruno.mascret.fr/dokuwiki/doku.php/java:moutons>

3. codez cette fonctionnalité.
4. pourquoi getUniqueId est-elle private ?
5. testez-là (dans le main)

## 4.2. Spécialisation d'Animal par rapport à sa classe mère Object

Consultez la documentation de la classe Object et le cours (chapitre 3, début).

### 4.2.1. Redéfinition de comportements par défaut hérités d'Object

#### 4.2.1.1. Représentation d'un Animal sous forme de chaîne de caractère

1. Quelle méthode permet de renvoyer une représentation sous forme de chaîne de caractères d'un objet ? D'où vient-elle ? Quel est son comportement par défaut ?
2. Sans modifier la fonction main de Animal, faire en sorte que les instructions `System.out.println(animal)` affichent :

```
NomDeLaClasse n° id_animal(sexe, (position x; position y)).
```

Exemple :

```
Animal 3 (Femelle ((25;30))
```

3. Comment peut-on récupérer le nom d'une classe ? Regardez la documentation de la classe Class (<https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html>) et de la méthode getClass(<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass>) de la classe Object ().
4. Pourquoi est-il préférable d'utiliser cette manière de faire plus complexe au lieu de d'écrire directement la chaîne « Animal » dans le code ?
5. Pourquoi n'y a-t-il pas de différence entre les deux affichages suivants :

```
Animal a = new Animal();  
System.out.println(a);  
System.out.println(a.toString());
```



Indice : « out » est un attribut statique de la classe System, de type PrintStream. (<https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out>).

Consultez la documentation des méthodes println de la classe PrintStream :

<https://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html>

6. Quel(s) pilier(s) est(sont) mis en œuvre ici ?
7. Comment afficheriez-vous un message d'erreur dans la console ?

#### 4.2.1.2. Comparaison de deux instances d'Animal

1. dé-commentez les lignes du test de comparaison de la méthode main.
2. interprétez le résultat

On considère que deux animaux sont identiques si ils ont le même **age**, le même **sexe** et le même **état**.

3. en utilisant les fonctionnalités de votre IDE (menu source sous éclipse), générez la méthode equals qui répond à ces contraintes.
4. exécutez et interprétez le résultat pour les animaux.
5. Pourquoi le comportement pour les chaînes de caractères semble incohérent ?

 Indice si vous bloquez : <http://www.touilleur-express.fr/2006/09/12/string-et-intern-on-ne-sait-pas-tout-sur-les-strings/>

### 5. Générez la documentation de votre projet en utilisant votre IDE.

Utilisez pour cela les outils de gestion javadoc de l'IDE.

Si besoin, complétez vos commentaires pour obtenir une jolie documentation !

### 6. Héritage et polymorphisme

L'objectif de l'activité est d'augmenter le niveau d'**abstraction** de la modélisation de notre sujet d'étude. Pour cela, on se propose de mettre en place des relations d'**héritage** ou d'implémentation entre différents acteurs du sujet d'étude.

Vous allez disposer d'une partie de la modélisation (sources de l'étape 6), et vous devrez trouver une autre partie de cette modélisation. Au départ, vous serez très guidés, et au fur et à mesure de l'activité vous aurez de moins en moins d'indications.

**Ne restez cependant pas bloqués plus de 10 minutes sur une question de conception.** Vos intervenants sont là pour vous aider et vous aiguiller vers une solution pertinente.

#### 6.1. Nouvelles contraintes

##### 6.1.1. Définition des acteurs

Pour rappel, notre objectif est de produire un simulateur pour l'apiculture.

L'augmentation du cahier des charges concerne essentiellement les acteurs impliqués dans le modèle.

Les acteurs sont :

1. des agents (animaux, végétaux) : regardez la définition dans les commentaires de la classe **Agent** donnée en support.
2. des éléments de décor (idem).

### 6.1.2. Les agents

Les animaux sont des agents qui ont un niveau de santé et un sexe. Les animaux **peuvent se déplacer**.

Les végétaux sont des agents qui produisent une certaine quantité de nectar ou de pollen et qui **ne se déplacent pas** (leur position est fixée une fois pour toute).

Parmi les Animaux, nous avons :

1. des Abeilles : elles se nourrissent uniquement de nectar et de pollen. Les abeilles domestiques ont toutes une Ruche où elles passent la nuit. Les abeilles solitaires dorment dans des arbres. Une abeille peut polliniser un végétal.
2. des frelons : un frelon se nourrit de nectar et de pollen, mais aussi d'abeilles. Il n'a pas de ruche (il dort en général dans un arbre). Il existe des frelons asiatiques et des frelons européens. Le frelon européen est le seul prédateur du frelon asiatique, qui est très vorace (un seul frelon asiatique peut décimer une ruche).
3. des Varroa (eurk....) : ce sont des parasites qui vivent sur les abeilles et qui mangent leurs larves.

Parmi les végétaux, nous retiendrons pour le moment :

1. les fleurs
2. les arbres : ce sont des végétaux qui peuvent héberger des insectes comme les frelons.

### 6.1.3. Les éléments de décor

Les éléments de décor :

1. ont une position fixe et ne se déplacent pas ;
2. certains peuvent héberger des insectes : Ruche par exemple.  
Pour le moment, nous retiendrons :
  1. les Ruches : elles hébergent des abeilles et contiennent une certaine quantité de miel
  2. les Champs : ce sont des zones cultivées qui peuvent contenir des pesticides qui diminuent le niveau de santé des abeilles et des frelons.

## 6.2. Mise en place de la hiérarchie des agents



La première partie de l'activité concerne la modélisation des **agents**.

### Principes mis en œuvre :

**Responsabilité unique** : une classe ne doit avoir qu'une seule raison de changer.

**Ouverture fermeture** : une classe doit être ouverte aux extensions mais fermée aux modifications.

**Substitution de LISKOV** : une méthode utilisant une référence vers une classe de base doit pouvoir référencer des objets de ses classes dérivées sans les connaître (polymorphisme).

### Concepts :

héritage, implémentation, classe abstraite, interfaces

## 6.3. Conception d'une hiérarchie d'agents

### 6.3.1. Travail préliminaire d'analyse sur papier

1. Sans écrire de code pour le moment, déterminez **sur papier** (diagramme de classe simple) les relations d'héritage entre les agents suivants (relation « *est-un* » ou « *est une sorte de* »). Chaque concept sera modélisé sous forme de classe.

concepts : **Abeille, Frelons, Animal, Fleur, Abeille solitaire, Abeille domestique, Frelon asiatique, Frelon européen, Varroa, Arbre, Vegetal, Agent.**

### 6.3.2. Réutilisation du travail précédent

Lors de la précédente activité (section 4), vous avez modélisé une classe **Animal**.

On se propose d'ajouter un niveau d'abstraction en codant une nouvelle classe **Agent** dans le package **model.agents**.

1. récupérez le code de l'archive pour la section 6 et étudiez-le. Corrigez éventuellement votre code, ou utilisez directement le code de la solution proposée dans l'archive.
2. identifiez les éléments de la classe **Animal** que vous avez réalisée en section 4 à transférer dans la nouvelle classe **Agent** (attributs, méthodes...).
3. procédez à ces modifications en utilisant les outils de refactoring :
  1. placez-vous sur la classe **Animal**
  2. menu Refactor → extract superclass
  3. nom de la superclasse à générer : **Agent**
  4. sélectionnez les méthodes et attributs à faire remonter dans la classe **Agent**

### 6.3.3. Classe de test

Au premier TP, la méthode **main** était programmée dans la classe **Animal**.

Aujourd'hui, nous allons utiliser une classe spécifique dont le rôle est de lancer le programme. C'est une bonne pratique car on sépare les tests et les exécutions du modèle de programme dans la conception.

1. Quel principe est mis en œuvre ?
2. Étudiez le code de la classe `Launcher`. En reprenant la méthodologie de test vue en section 4, **vous complétez cette classe au fil de l'eau** pendant le TP.
3. pourquoi y-a-t-il une erreur ? Corrigez le programme.

#### 6.3.4. *Animal et Agent*

1. Quel avantage en terme de taille de code la hiérarchie entre `Agent` et `Animal` offre-t-elle ?
2. Quel principe est ainsi mis en œuvre ?
3. la méthode `toString` de `Animal` doit afficher : `NomDeLaClasse n° id_agent (position x; position y), SEXE`

Une bonne manière de faire réutilise l'existant !

Regardez notamment les méthodes de la classe **Object** et de la class **Class** : lesquelles peuvent être utilisées ?

#### 6.3.5. *Hiérarchie complète d'agents*

1. codez la hiérarchie avec les classes **Abeille, Frelons, Fleur, AbeilleSolitaire, AbeilleDomestique, FrelonAiatique, FrelonEuropéen, Varroa, Arbre, Vegetal**.
2. quel(s) animaux se comportent comme des **Hebergeurs** (regardez l'interface donnée) ? Codez cette fonctionnalité. Testez.
3. seuls les animaux se déplacent. Codez cette fonctionnalité. Testez.
4. Les animaux disposent tous d'un **Hebergeur (Ruche, Arbre...)**. Ajoutez un attribut pour tous les animaux indiquant où ils sont hébergés.
5. les abeilles transportent une certaine quantité de miel

### 6.4. **Décor**

En appliquant la même méthodologie qui pour les Agents, mettez en place la hiérarchie pour le décor.

Pensez bien à utiliser les comportements existants définis dans les interfaces.

Testez les Hebergeurs : Arbres et Ruches.

2. Parmi ces classes, quelles sont celles qui pourraient être abstraites ? Pourquoi ?

## 7. **Un peu d'algorithmie !**

### 7.1. **cycle de simulation**

La simulation va réaliser des cycles (ou tour) régulièrement. Chaque agent exécutera alors le même algorithme à chaque tour :

1. vieillir
2. `seDeplacer` (si il le peut bien entendu...)
3. `seNourrir` (laisser cette méthode vide pour le moment).



4. mettre à jour ses données (état de santé, quantité de pollen/nectar, etc.).

En vous inspirant de l'exemple du cours sur les boissons caféinées (*[cours 103-105]*), mettez en place un algorithme générique de type **template method** dans la classe agent.

Vous pouvez également consulter : <https://refactoring.guru/fr/design-patterns/template-method>

La mise en place détaillée de cette template method est reprise de manière exhaustive en section 15.2. Vous pouvez vous contenter pour l'instant de faire fonctionner le programme sans regarder les subtilités.

## 7.2. fonctionnalités spécifiques

Codez les fonctionnalités suivantes :

1. si un frelon rencontre une abeille, le niveau de santé de l'abeille passe à EnDetresse.
2. si un frelon asiatique rencontre un frelon européen, son niveau de santé passe à EnDetresse
3. si le frelon a faim, le niveau de santé de sa proie passe à mourant et le frelon la mange.
4. si une abeille rencontre un Varroa et qu'elle n'est pas déjà parasitée :
  1. son niveau de santé diminue
  2. le Varroa se déplace maintenant avec elle
  3. une fois parasitée, une abeille ne peut plus se débarrasser de son parasite.
5. Les végétaux produisent de la nourriture (pollen ou nectar) selon les règles suivantes :
  1. fleur : création d'une unité de pollen/nectar par tour
  2. arbre : création de  $2^{(\text{tailleArbre}(\text{réel, en m}))}$   $2^{\text{taille arbre}}$
6. codez la méthode seNourrir. Si un animal ne se nourrit pas pendant un certain nombre de tours, son niveau de santé passe à mourant. Lorsqu'il se nourrit, son niveau de santé remonte. Pensez à mettre à jour la quantité de nourriture disponible sur le végétal.
7. un animal mourant est condamné, son niveau de santé ne peut plus remonter.

**Si vous n'avez pas encore commencé la 3ème séance de TP, plutôt que de passer à la section suivante, vous pouvez essayer de traiter les sections 14, 15.1 et 20.1 des objectifs complémentaires.**

## 8. Les collections (1ère partie)

### 8.1. Objectifs pédagogiques

L'objectif principal de cette activité est de vous permettre d'appréhender en douceur la notion de collection en POO et en JAVA (cours n°3, *cours [chap. 4 124-184]*).

Pour cette activité, vous disposez de code facilitant la génération automatique de monde. Pour le moment, n'essayez pas de comprendre comment cela fonctionne en détail ni de la modifier si ce n'est pas clairement demandé.

### 8.2. Mise en place

#### 8.2.1. Nouveau projet

Importez le code source de l'activité dans un nouveau projet. Choisissez le code qui correspond à votre niveau d'avancement :

- vous n'avez traité aucune activité de la partie 2 ou 3 : **srcSection8-normal** et le diagramme en annexe 24.1
- vous avez mis en place l'adapter *PointPositif* de *Point* (20.1) et le clonage d'*Agent* (15.1.2) : **srcSection8-avance** et le diagramme en annexe 24.2
- vous avez fait l'un ou l'autre mais pas les deux : par sécurité, partez de **srcSection8-normal** et intégrez vos modifications, ou si vous vous sentez à l'aise terminez les activités complémentaires et continuez ensuite la progression à partir d'ici en mode **avancé** ;

Le code source est incomplet (il ne contient pas tout le code des classes concrètes) mais vous donne une structure générale qui répond à ce que vous avez étudié dans les activités précédentes.

Le diagramme de classes général donné en annexe a pour but de vous faciliter le travail, de vous permettre d'identifier des oublis et surtout de fusionner rapidement le code donné avec ce que vous avez déjà réalisé.

### 8.3. Compléter le projet avec votre existant

1. Comparez les classes que vous avez programmées avec celles données dans les sources ; interprétez les différences éventuelles.
2. Intégrez le code de vos classes dans le nouveau projet ; faites attention à ne pas changer les visibilités des méthodes, leur signature, à respecter les attributs et leurs types (pas d'ajout d'autre attribut), etc.

#### Principes mis en œuvre :

**Responsabilité unique** : une classe ne doit avoir qu'une seule raison de changer.

Même si cela est possible, une classe ne devrait jamais s'occuper de manipuler les attributs d'une autre classe.

Quand un Frelon rencontre une Abeille, il ne doit pas mettre à jour le niveau de santé de l'abeille. C'est l'abeille qui déterminera son niveau de santé avec sa méthode rencontrer.

Même principe pour les hébergeurs.

**Substitution de LISKOV** : une méthode utilisant une référence vers une classe de base doit pouvoir référencer des objets de ses classes dérivées sans les connaître (polymorphisme).

La classe Frelon ne doit pas utiliser de référence vers les classes AbeilleDomestique ou AbeilleSolitaire.

La classe Abeille ne doit pas avoir de référence ni de cast vers Ruche, mais vers Herbergeur.

**Répondez aux points suivants (et modifiez votre code si besoin en conséquence) :**

- pourquoi la méthode `sInstaller` d'`Animal` est-elle final ?
- quel est l'intérêt d'utiliser une interface `Deplacable` plutôt que de donner une méthode abstraite `seDeplacer` à la classe `Animal` ? On parle des méthodes abstraites en deuxième partie (Erreur : source de la référence non trouvée) et *cours [100-101]*
- quelle est la différence entre `peutAccueillir` de `Hebergeur` et `accueillir` ? Pourquoi `accueillir` renvoie-t-elle un booléen ?
- une abeille domestique a forcément une Ruche comme hébergeur dès sa création. Comment prendre en compte cette contrainte ?

## 9. Échauffement sur les collections

### 9.1. Aggraver ou améliorer un état de santé

L'énumération `Etat` propose des valeurs d'état possible mais il n'est pas facile de passer logiquement d'un état à un autre.

C'est le but des méthodes `protected final void aggraverEtat()` et `protected final void ameliorerEtat()` de la classe `Animal`.

#### 9.1.1. Étude de `aggraverEtat()`

Regardez le code de la méthode `aggraverEtat` dans la classe `Animal`. Consultez également la documentation sur l'énumération `Etat`.

Répondez aux questions suivantes :

- quels sont les types concrets de collection utilisés dans l'algorithme ?
- pourquoi parle-t-on des valeurs possible d'une énumération comme d'un `EnumSet` et non d'une `EnumList` (ce dernier terme **n'existe pas** dans l'API java) ?
- quel est l'intérêt d'utiliser un `ListIterator` plutôt qu'un `Iterator` simple ?
- est-on obligé d'utiliser une liste chaînée ? Pourquoi ne pas utiliser une `ArrayList` ?
- pourquoi les méthodes `ameliorerEtat()` et `aggraverEtat()` sont-elles **final** ? Qu'est que cela implique ?

#### 9.1.2. Codage de `ameliorerEtat()`

En vous inspirant du code étudié, codez la méthode `ameliorerEtat()`.

## 9.2. Liste de proies du Frelon

### 9.2.1. Problématique

Les frelons utilisent des instructions `instanceof` afin de déterminer quelles sont leur proies.

1. Quel est l'inconvénient de cette solution en terme de factorisation de code ?

Au lieu de cela, on se propose d'utiliser une collection contenant des objets de type `Class` représentant les proies potentielles d'un `Frelon`.

2. En quoi cette solution est-elle préférable ? Quel(s) principe(s) de la POO sont ainsi pris en compte ?

### 9.2.2. Codage

Regardez la déclaration de la collection `proies` et sa construction dans la classe `Frelon`.

Remarquer au passage la puissance qu'offre Java dans le paramétrage des classes et des collections.

1. Comment interpréteriez-vous la déclaration `ArrayList<? extends Agent> liste;`
2. Que peut contenir cette liste ?

Le mécanisme reste le même avec les interfaces :

`ArrayList<? extends Hebergeur> liste;` et non pas :

`ArrayList<? implements Hebergeur> liste;`

3. Regardez la documentation de la classe `Object`

(<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.htm>)

Quelle méthode renvoie un objet de type `Class` ?

Il est également possible d'utiliser la syntaxe : `NomDeLaClasse.class` pour récupérer un objet de type `class`.

4. complétez le constructeur en ajoutant la classe `Abeille` aux proies du `Frelon`.
5. Testez. Pourquoi cela ne fonctionne-t-il pas ? En quoi cette technique diffère de `instanceof` `Abeille` ?
6. Corrigez le constructeur de `Frelon`.
7. Appliquez le même principe à `FrelonEuropeen` (réutilisez l'existant!)
8. Dans la méthode `protected void gestionProie(Animal a)` de `Frelon`, supprimez le `instanceof` et remplacez par un test indiquant si la classe de l'animal `a` appartient à la liste .

## 9.3. Population d'une ruche

Lorsqu'une `Ruche` accueille une `AbeilleDomestique`, elle va conserver cette information afin de maintenir une collection d'`AbeilleDomestique` (population de la Ruche).

Une ruche ne peut pas accueillir plus d'abeilles qu'une constante définie dans la classe (`private static int populationMax = 1000;`). Cette propriété sera utilisée lorsque la Ruche créera de nouvelles Abeille (régulation de sa population).

### 9.3.1. Choix de la collection

Ajouter une collection `population` en attribut de la classe Ruche.

Vous choisirez la collection la plus pertinente pour la Ruche (une `ArrayList` n'est pas la solution attendue...) parmi `LinkedList`, `HashMap` et `HashSet`.

### 9.3.2. Algorithmie

1. initialisez la collection dans le constructeur de ruche.
2. complétez la méthode `peutAccueillir` en ajoutant les contrôles opportuns
3. complétez la méthode `accueillir` afin qu'elle ajoute (ou non) une nouvelle abeille à la population
4. écrivez une méthode `toString` pour la classe ruche qui produira un affichage de la méthode `main` de test de la Ruche *similaire* au suivant :

```
Ruche (0;0) : population 1 abeilles
    *AbeilleDomestique 1 ((10;20)), Assexue
```

```
Ruche (0;0) : population 2 abeilles
    *AbeilleDomestique 1 ((10;20)), Assexue
    *AbeilleDomestique 2 ((10;20)), Assexue
```

```
Ruche (0;0) : population 3 abeilles
    *AbeilleDomestique 1 ((10;20)), Assexue
    *AbeilleDomestique 3 ((5;10)), Femelle
    *AbeilleDomestique 2 ((10;20)), Assexue
```

5. Pourquoi les abeilles ne sont-elles pas nécessairement classées par ordre d'insertion dans la collection ?
6. Quelle(s) modification(s) faudrait-il apporter, et dans quelle(s) classe(s), pour que les abeilles soient classées par ordre d'identifiant (donc par ordre de création dans le programme) ?

Regardez les différents comportements des `HashSet`, `LinkedHashSet`, `TreeSet`, etc...

7. Est-ce une bonne pratique de proposer une méthode `main` directement dans la classe à tester ? Pourquoi ? Adaptez en fonction de votre réponse.

## 10. Le monde des abeilles

Nous allons mettre en place le monde qui permettra de faire évoluer les Agents.

### 10.1. La classe Monde

Le danger est d'obtenir une classe « dieu », c'est à dire qui ne respecte pas le principe de responsabilité unique. Veillez à ne coder que le minimum dans la classe Monde et à déléguer aux autres classes les gestions spécifiques !

#### 10.1.1. *prise en main*

Consultez la classe Monde et répondez aux questions suivantes :

1. quelle collection est utilisée pour contenir les Agents ? est-ce une bonne idée ?
2. grâce à quelle méthode cette collection est-elle initialisée ? A quel endroit ?

#### 10.1.2. *composition aléatoire du monde*

Afin de faciliter les tests, la classe comporte un système permettant de tirer aléatoirement un objet.

Le mécanisme est étudié dans la deuxième partie du TP (section 16.1)

Pour fonctionner, il est nécessaire de rendre clonable les Agents (la notion de clonage est abordée en deuxième partie de TP section 15.1).

A ce stade, si vous êtes en mode normal, la méthode de tirage a été codée en utilisant une méthode en dur afin de ne pas compliquer les choses, et ne génère que quelques types possible de configuration d'agents (une seule Ruche, arbres de la même taille, etc.). Elle sera modifiée plus tard afin de permettre la création de tout type d'agent (partie 2).

## 10.2. Comparateurs

#### 10.2.1. *Comparable*

Lorsque vous exécutez le launcher, vous obtenez une erreur d'exécution. Pourquoi ?

1. Implémentez l'interface qui permet de lever cette erreur.

La comparaison se fera sur l'id de l'Agent.

2. Était-ce une bonne idée de choisir un TreeSet plutôt qu'un HashSet pour agents ?

#### 10.2.2. *Comparator*

Regardez le code commenté de la méthode toString du monde.

Afin de faciliter l'affichage, il serait préférable de disposer d'un TreeSet triant d'abord sur la coordonnée x puis en second sur la coordonnée y.

Ainsi (3,2) est avant (10,1) ; (3,6) est avant (3,8) ; on choisira en cas d'égalité de coordonnées que la deuxième viendra après la première dans la liste.

1. créez un comparateur CoordComparator qui implémente Comparator<Agent> dans une classe indépendante (ne faites pas de classe interne dans monde).
2. testez ce comparateur et vérifiez que l'ordre de parcours correspond à l'algorithme indiqué.

## 11. Bilan intermédiaire

Faites le point sur l'ensemble des Collections et de leurs classes associées étudiées jusqu'à présent.

1. Quel(s) type(s) de collection(s) n'a(ont) pas encore été utilisé(s) pour l'instant ?
2. Commencez un arbre de décision vous permettant de choisir la collection qui répond le plus à vos besoins (doublons/uniques, trié, etc.)
3. Complétez cet arbre de décision au fur et à mesure des activités (c'est un bon outil pour le DS!).

## 12. Les collections (2ème partie)

### 12.1. Finalisation du Monde

Cette dernière partie a pour objectif de vous trouver seul des solutions à des problèmes donnés.

Vous disposez à présent de suffisamment de connaissances pour vous débrouiller (presque) seuls !

À vous de jouer !

### 12.2. Implémentez les fonctionnalités suivantes.

#### 12.2.1. Rencontre entre Agent

Le monde a la responsabilité de gérer les rencontres entre ses agents.

Deux agents a et b aux positions respectives  $(x_a, y_a); (x_b, y_b)$  se rencontrent si :

$$|x_b - x_a| \leq \text{rayon} \quad \text{et} \quad |y_b - y_a| \leq \text{rayon}$$

Rayon est une constante entière définie dans monde (une valeur de 10 est bien).

#### 12.2.1.1. Première approche naïve

1. Créez une méthode *gererRencontre()* dans le monde qui, pour chacun des agents, renvoie une liste des agents qu'il peut rencontrer (on les appellera les voisins de l'agent).
2. Appeler la méthode rencontrer de l'agent avec chacun de ses voisins.
3. Quels sont les problèmes que posent cette approche ?
4. Proposez un tri de la collection d'Agent du monde qui corrige un de ses problèmes.

Une approche plus performante est proposée en 21.2 (Cartographie du monde).

Sous windows, pour obtenir un affichage correct en console, fixer la longueur des colonnes du terminal et enlever la limite du nombre de lignes affichables.

#### 12.2.2. Cycle complet du monde

Voici l'algorithme complet du cycle pour le monde à coder :

1. le monde gère les rencontres entre ses agents ;
2. le monde appelle la méthode cycle de tous ses agents ;
3. chaque agent doit définir sa méthode maj (elle est appelée par la méthode cycle de chaque agent) ;

4. le monde retire de ses Agents les Animaux qui sont Mourant. L'interface Hebergeur doit disposer d'un nouveau comportement « supprimer(Animal) » ; penser à déréférencer TOUTES les références de l'Animal. Vous pouvez également ajouter une méthode « mourrir » à Animal qui sera chargée de gérer une partie des opérations (respectez au mieux cependant le principe de Responsabilité Unique) ;

### 12.3. Mode « nuit »

Lorsqu'il fait nuit, les Animaux changent de méthode de déplacement et cherchent à rejoindre leur hebergeur. Une fois positionnés aux mêmes coordonnées que leur Hebergeur, ils ne bougent plus tant que c'est la nuit.

#### 12.3.1. *Mode nuit et Hebergeurs*

Un hebergeur doit donc pouvoir transmettre ses coordonnées.

1. Est-ce un problème d'ajouter une méthode getCoord() à Hebergeur alors que certains Hebergeurs (Agents) ont déjà cette méthode héritée d'Agent ? Pourquoi ?
2. Qui est responsable du changement de mode ?
3. implémentez cette fonctionnalité.

#### 12.3.2. *Cas particulier de la Ruche*

1. La première abeille qui rentre dans la Ruche (pour apporter du miel ou à la nuit) sera également celle qui en ressort en premier.
2. Le comportement « nuit » chez l'abeille domestique ressemble fortement à celui qui consiste à ramener du nectar à la ruche (pas forcément implémenté à ce stade). Comment organiser votre code au mieux en tenant compte de cette information ?

### 12.4. Liste des fonctionnalités du logiciel à implémenter (Rappel et nouvelles)

1. si un frelon rencontre une abeille, le niveau de santé de l'abeille passe à EnDetresse.
2. si un frelon asiatique rencontre un frelon européen, son niveau de santé passe à EnDetresse
3. si le frelon a faim, le niveau de santé de sa proie passe à mourant et le frelon la mange : attention, le frelon ne doit pas mettre à jour les données de la proie : la proie se mettra à jour elle-même quand on appellera sa méthode rencontrer.
4. si une abeille rencontre un Varroa et qu'elle n'est pas déjà parasitée :
  1. son niveau de santé diminue
  2. le Varroa se déplace maintenant avec elle
  3. une fois parasitée, une abeille ne peut plus se débarrasser de son parasite.
5. Les végétaux produisent de la nourriture (pollen ou nectar) selon les règles suivantes :
  1. fleur : création d'une unité de pollen/nectar par tour
  2. arbre : création de  $2^{tailleArbre}$  (réel, en m)  $2^{taille arbre}$
6. Corrigez l'erreur de la méthode aggraverEtat() de Animal.



7. codez la méthode seNourrir. Si un animal ne se nourrit pas pendant un certain nombre de tour, son niveau de santé **diminue**. Lorsqu'il se nourrit, son niveau de santé remonte. Pensez à mettre à jour la quantité de nourriture disponible sur le végétal.
8. Les abeilles domestiques rapportent du nectar à la ruche (cf. plus haut 21.2 (Cas particulier de la Ruche).)

## Deuxième partie : objectifs complémentaires

### 13. Objectifs

Cette deuxième partie vise des compétences plus spécifiques et plus approfondies que celles de la première partie. Elle doit vous permettre de mieux comprendre le fonctionnement des piliers de la POO dans des mises en œuvre plus subtiles ou plus techniques que lors de la première partie.

**Ces compétences sont au programme, et l'idéal est donc d'en maîtriser le plus possible !**

Les différentes activités reprennent des activités précédemment, en les complétant, en vous faisant réfléchir sur un point rapidement évoqué et qui mérite plus de réflexion, ou bien vous propose de nouveaux problèmes dont la résolution améliorera les qualités du programme et vous fera monter en compétence.

### 14. Héritage et abstraction : les classes abstraites

Lors de la mise en place de la hiérarchie des agents (), vous vous êtes peut-être rendu compte que certaines classes servaient uniquement à regrouper des classes concrètes, mais qu'il n'était pas utile de les utiliser pour créer des instances.

Par exemple, la création suivante n'a aucun intérêt :

```
Agent a = new Agent();
```

Pour préciser cela, il est possible de déclarer ces classes abstraites avec le mot-clef **abstract**.

Il ne sera pas possible d'appeler directement un constructeur d'une classe abstraite.



Ca reste abstrait ? Regardez :

- *cours [100-101]*

- sur les classes abstraites, les méthodes abstraites et les interfaces, un petit TP guidé et corrigé :

<http://bruno.mascret.fr/dokuwiki/doku.php/java:abstract>

1. indiquez quelles classes sont abstraites dans la hiérarchie.
2. appliquez les changements et corrigez les erreurs éventuelles.
3. nous avons vu que seuls les Animaux se déplacent en section 6.3.5. Comment leur imposer de disposer d'une méthode seDeplacer sans pour autant expliquer comment un animal se déplace dans la classe Animal ?

### 15. Techniques et mises en œuvre s'appuyant fortement sur la POO

#### 15.1. Clonage : comment améliorer l'encapsulation

##### 15.1.1. Clonage d'un Point

Une technique d'encapsulation en objet consiste à fournir des clones au lieu de références directe lorsqu'on utilise un accesseur.

En java, tout objet dispose d'une méthode clone héritée d'Object **MAIS** elle ne permet pas forcément de cloner un objet.

Une classe est clonable uniquement si elle implémente l'interface **Cloneable** de java.

La classe Point implémente déjà **Cloneable**.

1. consultez <https://ydisanto.developpez.com/tutoriels/java/cloneable/>
2. modifiez l'accesseur de coordonnées afin qu'il renvoie un clone de coord.
3. quelle différence y a-t-il entre les deux manières de protéger coord ? Laquelle est la meilleure d'un point de vue Objet ?

### 15.1.2. Clonage d'un agent

Contrairement à Point, Agent n'implémente pas Cloneable (c'est normal, c'est nous qui avons réalisé cette classe).

1. faites implémenter par Agent l'interface Cloneable et ajoutez dans la classe Agent :  
`public abstract Object clone();`
2. quel est l'intérêt de cette manipulation ? Regardez les méthodes héritées d'Object : que veut-on éviter en utilisant le mot-clef **abstract** ?
3. quelles sont les conséquences sur le code ?
4. En vous inspirant de la méthode donnée dans AbeilleDomestique, ajoutez les méthodes de clonage *clone()* dans les classes terminales.

## 15.2. Template Method canonique

Template Method est ce qu'on appelle un design pattern en POO : une manière de répondre proprement à un problème de conception objet.

En section 7.1, nous avons mis en place un cycle de simulation dans la classe **Agent**.

Pour appliquer pertinemment le template method, vérifiez :

1. que vous avez bien pensé à ce que la méthode cycle d'Agent ne soit pas redéfinissable par ses héritiers (pourquoi d'ailleurs ?)
2. que les méthodes qui varient chez les héritiers sont bien abstract dans la classe Agent
3. réfléchissez à la visibilité des méthodes impliquées dans le cycle : **public**, **private** ou **protected** ? Quelles sont les intentions du concepteur dans chacun des cas ?
4. quel principal pilier est mis à contribution avec une template method ?

## 16. Étude du code

### 16.1. Génération automatique des Agents du Monde

La classe Monde dispose d'un mécanisme de tirage aléatoire permettant d'obtenir un Agent et de le cloner ensuite avant de l'introduire dans la collection d'agents.

1. dé-commentez le code de la méthode tirage concernant la partie 2 et commentez le code concernant la partie 1. Attention à bien conserver les dernières lignes qui attribuent une position à l'agent.
2. étudiez le fonctionnement de ce mécanisme de tirage
3. sur quels objets de l'API Collection repose-t-il ? Pourquoi avoir fait ce choix ?
4. comment est initialisé la table de statistiques ? que comporte-t-elle ?
5. comparez ce que vous avez commenté et dé-commenté : quel est l'avantage de la nouvelle solution ?

## **17. Nouvelles fonctionnalités**

### **17.1.1. Arbres Hebergeur**

Un arbre peut accueillir un nombre d'animaux qui est fonction de sa taille : regardez le code proposé.

1. En vous servant de ce qui a été réalisé pour la Ruche, adaptez ce code afin qu'un arbre connaisse également ses hébergés.

### **17.2. Santé des agents**

1. un animal mourant est condamné, son niveau de santé ne peut plus remonter.
2. un champ dans lequel il y a eu des pesticides diminue la santé d'un animal qui le traverse.
3. Si une abeille domestique a faim ET qu'elle se trouve dans la ruche ET que la ruche contient du miel, elle peut utiliser le miel de la ruche pour se nourrir (pensez bien aux responsabilités de chaque classe).

## **18. Simulateur graphique**

Nous vous proposons de réaliser un petit simulateur graphique plus agréable qu'une sortie en mode console. Vous pouvez si vous préférez passer tout de suite à la partie trois, puis revenir ici.

Vous devrez néanmoins savoir pour continuer :

1. utiliser un gestionnaire de positionnement ;
2. gérer une action événementielle (source, événement, écouteur)
3. connaître les principes de swing (containers, gestion des événements)

Enfin, les images du programme proviennent du site <http://gif.toutimages.com/>.

### **18.1. Mise en place**

1. Récupérez le code donné pour la classe Monde, le Launcher GUI et le package ui fournis avec le sujet.
2. insérez ces classes dans votre projet. Corrigez les éventuelles erreurs

#### **18.1.1. Travail préliminaire au dessin des images**

1. Afin d'être dessinables sous forme d'image, les décors et les agents doivent disposer d'une chaîne de caractère représentant le nom de leur image.

- ajoutez une interface `Dessinable` et faites-la implémenter par `Decor` et `Agent`.

Voici le code de la méthode à ajouter :

```
public String getImage() {  
    return "images/"+getClass().getSimpleName()+".png";  
}
```

Attention, si vous êtes sous windows, vous aurez peut-être des problèmes car les chemins de fichiers ne sont pas les mêmes que sous mac ou linux. Utilisez dans ce cas `File.separator` qui donne le séparateur de fichier de votre OS au lieu d'écrire « / »

Soit : 

```
return "images"+File.separator+getClass().getSimpleName()+".png";
```

les images portent le nom de la classe et se trouvent dans le répertoire « images » à la racine du projet (doit apparaître dans les ressources du projet, pas les sources).

### 18.1.2. Étude du code

- Consultez la classe `WorldFrame`, regardez ce qu'elle fait.
- utilisez le Launcher GUI
- Vous devriez obtenir ceci, centré au milieu de votre écran (comportement par défaut de `setLocationRelativeTo(null)`) :



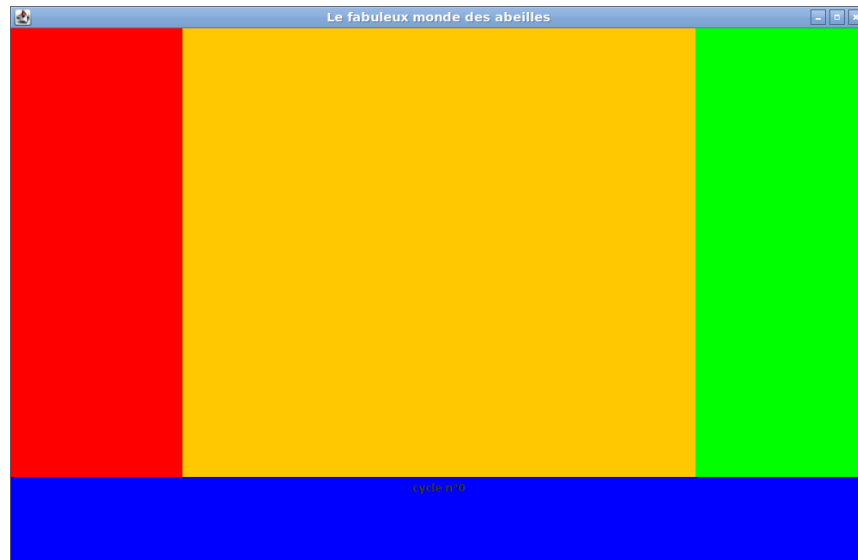
## 18.2. Panneaux extérieurs

### 18.2.1. taille des panneaux

Les panneaux extérieurs (vert, bleu et rouge) sont tout petits car ils ne contiennent aucun composant.

- améliorer l'affichage en leur donnant une taille idéale dans la méthode `private void fabriqueFenetre()` de la classe `WorldFrame`. (200 pixels de large pour les panneaux latéraux et 100 pixel de haut pour le panneau du bas).
- quelle est la bonne méthode à utiliser pour que les contraintes de dimension soient appliquées ? `setSize` ou `setPreferredSize` ? Pourquoi ?
- quelle dimension est prise en compte par l'utilisation du gestionnaire de positionnement `BorderLayout` pour les panneaux extérieurs ?

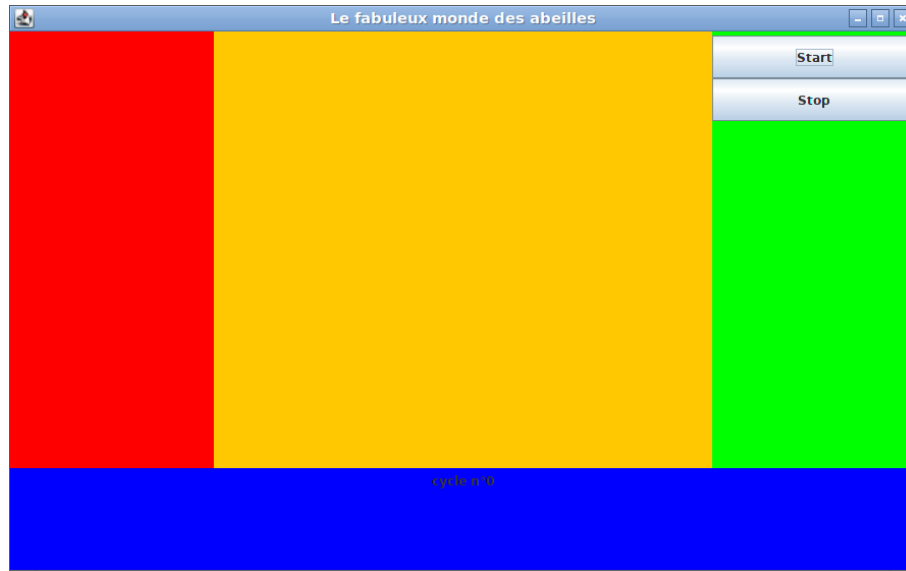
Vous devriez obtenir :



### 18.2.2. **Panneau de commande (en vert à droite)**

L'objectif de ce panneau est d'accueillir les boutons et commandes permettant de lancer l'animation, de la mettre en pause et plus généralement d'accueillir les commandes de paramétrages de l'animation (de régler la vitesse par exemple dans le prochain TP).

1. Regardez comment fonctionne le gestionnaire de positionnement GridLayout.
2. Dans la méthode `fabriqueFenetre()`:
  1. ajoutez un GridLayout gérant une grille de 10 lignes et d'une seule colonne.
  2. créez et ajoutez un bouton « start » et un bouton « stop » que vous ajouterez au panneau latéral droit.
  3. ajoutez à chacun de ces boutons un ActionListener à l'aide d'une classe anonyme qui utilisera les méthodes `start()` et `stop()` de la classe Monde.
  4. testez si les événements sont bien interceptés
3. vous devriez obtenir le résultat suivant :



### 18.2.3. **Panneau d'information (en bleu en bas)**

1. Ajoutez un label à ce panneau qui permettra d'envoyer des informations à l'utilisateur.
2. Comment ce label doit-il être déclaré pour pouvoir être modifié ensuite par la classe WorldFrame ?
3. Affichez un message d'information indiquant l'état de l'animation (non démarré, en cours, en pause).

**Bon à savoir :** il est possible d'utiliser HTML pour la mise en forme des textes des JLabel et des swings en général.

Voir à ce sujet <https://docs.oracle.com/javase/tutorial/uiswing/components/html.html>

## 18.3. **Panneau central : animation**

L'objectif de ce panneau est de contenir l'animation principale.

### 18.3.1. **Création du panneau**

Il va falloir utiliser un panneau qui change le comportement standard de JPanel.

1. Créez une classe `PanneauPrincipal` qui hérite de `JPanel`
2. Décommentez/commentez les lignes créant le panneau principal dans `fabriqueFenetre` et assurez-vous que le constructeur de `PanneauPrincipal` dispose bien d'un paramètre de type `MondeAnimable`.
3. Que fait la méthode suivante si elle appartient à `PanneauPrincipal` :

```
public PointPositif convertir(PointPositif p) {
    int x = p.getX()*this.getWidth()/Monde.getLongueur();
    int y = p.getY()*this.getHeight()/Monde.getLargeur();
    return new PointPositif(new Point(x,y));
}
```

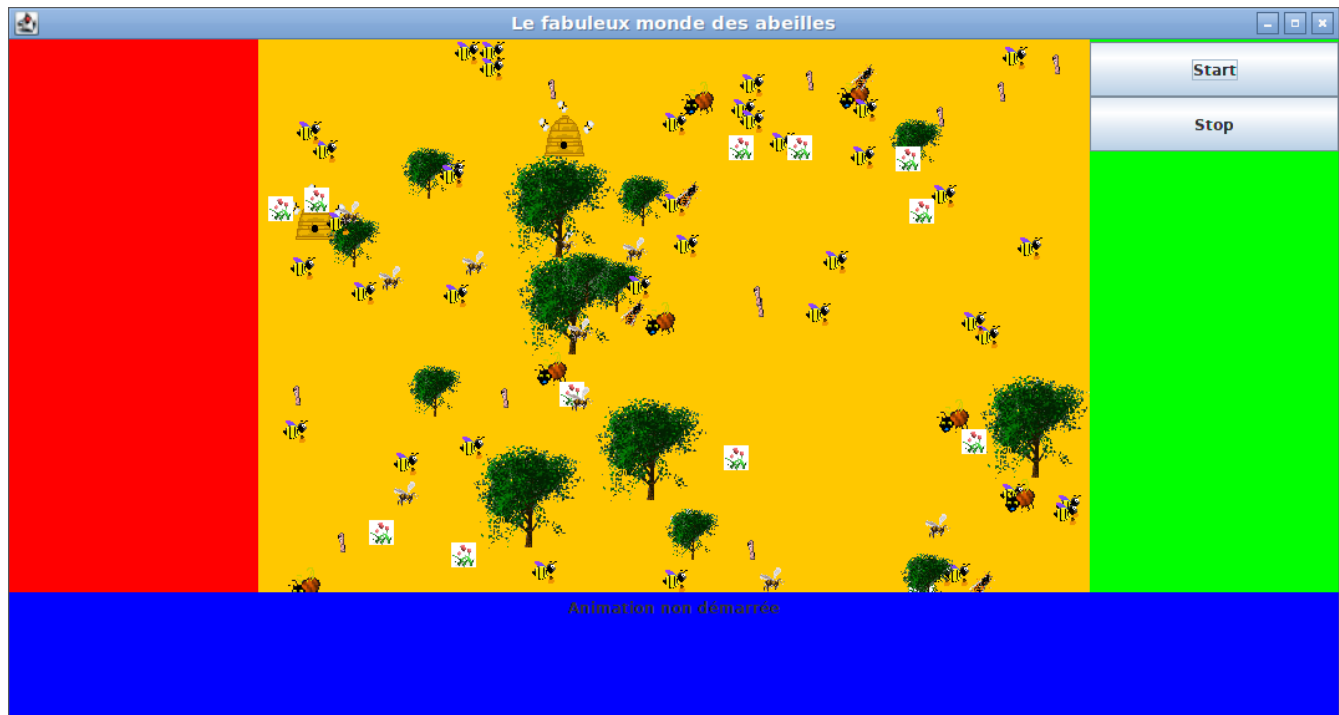
4. Redéfinissez la méthode `paintComponent` de `JPanel`. Voici le code permettant de dessiner une image pour un élément dessinaable avec un `Graphics g` :

```
Dessinable elem = ... ;
PointPositif coord = convertir(elem.getCoord());
```

```
try {
    Image img = ImageIO.read(new File(elem.getImage()));
    g.drawImage(img, coord.getX(), coord.getY(), elem.getWidth(), elem.getHeight(),
    this);
} catch (IOException e) {
    g.drawString(elem.toString(), coord.getX(), coord.getY());
}
}
```

### 18.3.2. Testez !

Vous devriez obtenir une fenêtre ressemblant à la fenêtre donnée ci-après.





## Troisième partie : pour aller plus loin

### 19. Objectifs

Cette dernière partie vise des compétences encore plus spécifiques et plus approfondies que celles des deux premières parties. Elle doit vous permettre d'aller plus loin dans votre découverte de la philosophie objet, ou vous challenge sur des difficultés techniques de mise en œuvre, des faiblesses de conceptions des parties précédentes.

**Ces compétences ne sont pas au programme, mais elles vous permettront de vous améliorer !**

**Qui peut le plus, peut le moins !**

Comme pour la partie 2, les différentes activités reprennent des activités précédemment, en les complétant, en vous faisant réfléchir sur un point rapidement évoqué et qui mérite plus de réflexion, ou bien vous propose de nouveaux problèmes dont la résolution améliorera les qualités du programme et vous fera monter en compétence.

### 20. Améliorer les faiblesses de conception

#### 20.1. Adapter une classe existante

Le système de coordonnées pose un problème : les valeurs de x et y peuvent être négatives, ce qui n'a pas de sens si on considère que le coin en haut à gauche de l'écran a les coordonnées (0,0).

Le problème, c'est que la classe Point utilisée fait partie de l'API java ! On ne peut pas la modifier.

Un autre problème : la classe point permet de modifier directement les attributs x et y qui sont déclarés public. Ce n'est pas une bonne pratique objet !

1. rappelez pourquoi mettre un attribut public est une mauvaise idée en général.
2. à votre avis, pourquoi les concepteurs de l'API java ont-ils enfreints cette bonne pratique ?
3. créez une classe PointPositif qui permet d'utiliser des coordonnées qui ne peuvent pas être négatives, et qui encapsule x et y. Réutilisez la classe Point, mais cherchez une manière intelligente de le faire (regardez comment fonctionne un adaptateur en POO : <https://refactoring.guru/fr/design-patterns/adapter> ).
4. Mettez en place un système de clonage pour la classe PointPositif. Faut-il également cloner le Point adapté ? Est-ce indispensable ?

### 21. Nouvelles fonctionnalités

#### 21.1. Le problème des Hébergeurs

L'interface Hébergeur est pénible à utiliser, car un Hébergeur doit à chaque fois préciser quels types de classes il est en mesure d'héberger.

1. Quelle solution basée sur les collections serait envisageable pour limiter ce problème ?
2. Après avoir réfléchi à une solution, consultez cet article sur les classes paramétrées :

<https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java/22404-la-genericite-en-java>

Essayez de mettre en œuvre cette technique pour paramétrer les Hebergeurs.

Parmi les deux solutions envisagées( collection ou classe paramétrée), laquelle peut-on mettre en œuvre étant donné les choix de modélisation qui ont été fait ?

### 3. Mettez en place cette solution.

## 21.2. Cartographie du monde

Afin d'améliorer les performances de l'algorithme de rencontre, on se propose d'optimiser le monde en le cartographiant avec des cases. Le monde disposera d'une collection de type Map avec les cases comme clef et la liste des agents présents à cet endroit comme valeur.

Une rencontre ne pourra se faire qu'entre agents se trouvant sur la même case.

1. en quoi cette organisation optimise-t-elle l'algorithme ?
2. quel en est la principale contrainte ?
3. codez cette nouvelle organisation

Dans un premier temps, utilisez deux collections d'agents : une pour la population (celle qui existe déjà) et une nouvelle pour la cartographie.

Vous pouvez définir une case comme possédant un coin supérieur et un coin inférieur (donc deux objets PointPositifs). Cela facilite le test pour savoir si un Agent appartient ou non à la case.

Attention : **case** est un mot-clef de java. Vous ne pouvez pas l'utiliser comme nom de variable. Préférez « zone ».

## 22. La classe Monde Animable

L'objectif pour vous est de comprendre comment fonctionne l'animation, quels événements sont générés et comment ils sont écoutés et gérés.

### 22.1. Classes/interfaces impliquées

Regardez notamment à quoi servent les classes/interfaces suivantes :

1. Thread
2. Timer (celui de swing)
3. Runnable

### 22.2. Schéma des collaborations entre Monde et WorldFrame

Proposez un schéma de l'organisation événementielle entre Monde et WorldFrame.

Répondez également aux questions suivantes :

1. Pourquoi utiliser une interface MondeAnimable ? Est-ce une protection suffisante ?
2. Même question pour Dessinable ?

3. Quelle autre solution, que nous avons déjà vu avec PointPositif, permettrait de corriger les problèmes ?
4. Mettez -la en œuvre **sans changer une ligne de la classe WorldFrame**.

## 23. Amélioration du client graphique

Voici en vrac quelques idées de fonctionnalités :

1. facile : utilisez un composant graphique pour accélérer/diminuer la vitesse des cycles. Vous placerez ce composant dans le panneau de commande.
2. moyen : les images sont normalement animées mais ce n'est pas le cas sur le panneau d'affichage.

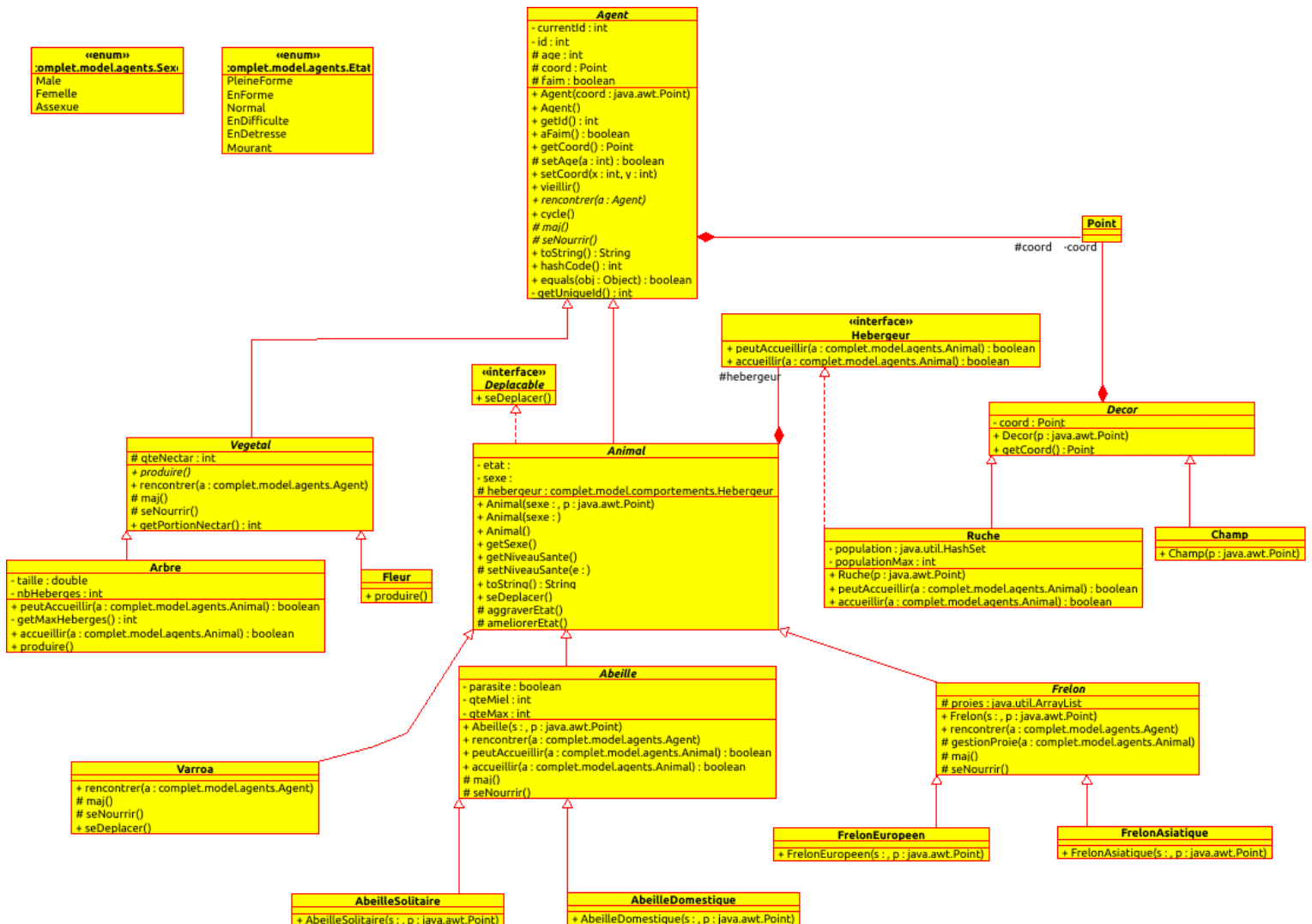
La raison est que pour obtenir une gif animée, il faut créer un JLabel avec une ImageIcon en paramètre. Ce changement d'organisation du PanneauPrincipal facilitera aussi le point 4.

3. moyen : réalisez un panneau indiquant les statistiques sur le monde : nombre d'agent, par type, temps écoulé, etc. Attention : vous ne devez pas récupérer la liste des agents de la classe monde mais imaginer une solution plus objet (qui est responsable de cette fonctionnalité, qui fait le travail, qui s'occupe de l'affichage, etc.).
4. difficile : mettez en place un système qui permet d'afficher dans le panneau du bas (information) les informations sur l'agent qui se trouve sous le pointeur de la souris (le toString de l'agent concerné suffira dans un premier temps!)

# Annexes

## 24. Diagrammes de classe

### 24.1. Section 8, progression normale



## POO – 4ETI – B.Mascret



## 25. Synthèse des objectifs pédagogiques

**Les objectifs pédagogiques indispensables sont en gras.**

Les objectifs pédagogiques complémentaires sont écrits sans mise en forme particulière.

*Les objectifs pédagogiques d'approfondissement sont en italique.*

### 25.1. Piliers (et quelques principes) de la POO

#### 25.1.1. **Abstraction**

1. **définir un modèle (une abstraction) : la classe** (4.1)
2. **créer et utiliser des objets** (4.1.5,4.1.7.2)
3. **identifier les comportements (méthodes) de classes** et les méthodes d'instance (4.1.7.2)
4. programmer avec des abstractions de haut niveau ()

#### 25.1.2. **Encapsulation**

1. **encapsuler des propriétés (attributs)** (4.1.6,4.1.7.1,4.1.7.2)
2. **encapsuler des comportements (méthodes)** (4.1.7.1)
3. protéger des références et garantir l'encapsulation
  1. **en utilisant des assesseurs et des mutateurs à bon escient** (4.1.6)
  2. par clonage ()
  3. *en mettant en place des mécanismes architecturaux s'appuyant sur les principes de l'objet* ()

#### 25.1.3. **Héritage**

1. **utiliser l'héritage entre classe** (4.2, 6)
2. identifier et utiliser des classes abstraites (Erreur : source de la référence non trouvée)

#### 25.1.4. **Polymorphisme**

1. **proposer plusieurs signatures de méthode** (surcharge, polymorphisme ad-hoc) (4.1.5)
2. polymorphisme d'opérateurs (coercition, polymorphisme ad-hoc) (ex. cours n°2)
3. **redéfinir des comportements** (polymorphisme d'héritage, interfaces) (4.2, 4.1.7.2, 6.3.4))
4. *polymorphisme paramétrique* (21.1)
5. principe de substitution de Liskov ()

#### 25.1.5. **Principes de la POO (introduction)**

1. **introduction au principe de responsabilité unique (SRP)** (4, 6.2)
2. introduction au principe d'ouverture-fermeture (OCP) (4, 6.2)
3. **introduction au principe de substitution de Liskov (LSP)** (6.2)
4. *introduction au principe de ségrégation des interfaces (ISP)* ()
5. *introduction au principe d'inversion de dépendance (DIP)* (15.2)

## 25.2. Techniques et mise en œuvre en java

1. **savoir programmer une classe** (4)
2. **reconnaître une interface, comprendre ce qu'est une implémentation d'interface en java** ()
3. **organiser son code en package** (4.1.1)
4. **distinguer les types primitifs des types construits** (4.1.4)
5. comprendre les conversions de type primitif (4.1.7.2)
6. transtypage de types construits ()
7. **comprendre et utiliser la classe Object** (4.2.1)
8. **comprendre et utiliser la classe Class** (4.2.1.1)
9. **redéfinir des comportements standards existants** (4.2.1)
10. **comprendre le mécanisme des exceptions java** ()
11. savoir intercepter une exception et la traiter ()
12. *programmer sa propre exception* ()
13. **utiliser une énumération java** (4.1.4, 4.1.5)
14. utiliser les types paramétrés ()
15. *créer une classe paramétrée* ()
16. **utiliser des composants de l'API java** (4.1.7.2, 4.2.1)
17. *manipuler une interface graphique en swing* ()
18. comprendre le fonctionnement évènementiel de swing ()
19. *gérer des évènements* ()

## 25.3. Bonnes pratiques

### 25.3.1. *codage*

1. **présenter son code correctement** (4.1.3)
2. **nommer correctement classes, attributs, méthodes...**
3. **commenter son code** avec le bon style de commentaire (4.1.2)

### 25.3.2. *documentation*

4. **savoir lire une documentation javadoc** (3.3.2, 4.1.7.2)
5. générer automatiquement une documentation (5)

### 25.3.3. *IDE*

6. **prendre en main un IDE** (eclipse) (4.1.1)
7. **utiliser les fonctionnalités de base d'un IDE** : génération de code, lancer, refactoring, completion (4.1.6, 4.2.1.2, 6.3.2)

8. utiliser un debugueur, définir des points d'arrêt ()
9. *utiliser des conditions d'interruption lors d'un debugage* ()

## 25.4. Conception

1. **comprendre l'organisation d'un programme objet** (4,4.1.7.2)
2. modéliser un problème à l'aide de classes (4, 4.1.7.2,4.1.7.3)
3. *critiquer une conception donnée et proposer des corrections* ()

## 26. Crédits et illustrations

### 26.1. crédits

*Installation de la documentation*, section 3.3 : **Alain Becker** (disponible également sur moodle, dans la rubrique trucs et astuces).

### 26.2. illustrations

*Les quatre piliers de la POO* : temple d'Athéna Lindia, acropole de Lindos à Rhodes, Grèce, [https://commons.wikimedia.org/wiki/File:Temple\\_of\\_Athena\\_Lindia\\_from\\_the\\_north\\_2010.jpg](https://commons.wikimedia.org/wiki/File:Temple_of_Athena_Lindia_from_the_north_2010.jpg)  
Wknight94 talk, CC BY-SA 3.0, via Wikimedia Commons

*Icônes* : package latex todonotes, <https://ctan.org/pkg/todonotes>, LaTeX Project Public License 1.3

### 26.3. licences utilisées :

CC BY-SA 3.0,    : <https://creativecommons.org/licenses/by-sa/3.0>

LaTeX Project Public License 1.3 : <https://ctan.org/license/lppl1.3>