ENSTA BRETAGNE

Programmation orientée objet en Java

Projet

Raúl Mazo & Hiba Hnaini

Année scolaire 2023-2024

Indications pour bien réussir votre projet

Recommandations et consignes pour la formation des groupes et l'organisation du travail

1. Veuillez faire des groupes de 3 étudiants au plus.
2. Le mode Agile s'invite ici pour éviter l'effet tunnel et pour vous aider à livrer des versions exécutables dès le début !
3. Donnez-vous les moyens de travailler correctement. Prenez le temps de vous organiser et de travailler en utilisant les outils qu'il vous faut : un bon IDE (IntelliJ Ultimate ou Eclipse), un outil de gestion de versions et de travail collaboratif (Git et GitHub) …
4. Les **difficultés avec votre ordinateur**, **vos outils** (performances, configuration, bugs, licences, ...) **et votre équipe de travail** (horaires de travail, répartition des responsabilités, manque de confiance, manque d'engagement et de communication ...) **sont de la responsabilité de l'étudiant ou de l'équipe et NON des enseignants**.

Recommandations et consignes pour la réalisation du projet

1. Réfléchissez bien avant de coder : attention à la modélisation, au découpage, à la « duplication » de code, … Ceci est un cours de PROGRAMMATION ORIENTEE OBJET et non de programmation impérative.
2. Testez, testez et... testez encore (mieux : écrivez des tests !).
3. Pensez à rédiger une javadoc parfaite (la compiler et la relire). La javadoc ne fait que "formater" la documentation que vous avez renseignée dans le code source. Cela ne sera pas magique si vous ne documentez pas bien les classes, méthodes, paramètres, les exceptions propagées, … de votre application. Voici un tutoriel présentant l'outil Javadoc de Sun, qui permet de générer les documentations d'un code Java : https://simonandre.developpez.com/tutoriels/java/presentation-javadoc/
4. Lorsque vous êtes satisfaits de votre application, prenez le temps, à tête reposée, de relire tout le code. Interrogez-vous, étonnez-vous, vérifiez la cohérence entre les classes, les éventuelles redondances de code, et le besoin d'abstraire ou de rendre générique, réfléchissez aux éventuelles modifications futures et à leurs répercussions sur l'évolution du code, ...
5. Je vous conseille ensuite de soumettre votre code à Sonarcube (et/ou Codacy) ; ça devrait vous permettre d'encore améliorer votre code. Ces outils vous permettront de faire une analyse statique qui a pour but de mesurer la qualité du code de vos applications et de vous fournir des métriques portant sur la qualité du code et permettant d'identifier précisément les points à corriger (code mort, bugs potentiels, non-respect des standards, manque ou excès de commentaires...). L'utilisation de ce type d'outils ne remplace en aucun cas les tests unitaires, mais permet d'identifier rapidement certains défauts du code.

Recommandations pour la partie rapport et rendu du travail

Voici quelques recommandations pour préparer et rendre vos projets :

1. Vous rendrez votre travail en exportant votre projet Java en format .zip. Préciser le nom du fichier .zip en y incluant les noms des étudiants de l'équipe.
2. Préparez la documentation de votre projet : l'architecture (au moins le diagramme représentant les modules/composants et classes de votre projet) doit être faite avant de commencer à coder ; à chaque fois que vous avancez sur votre projet, préparez votre javadoc et documentez ce que vous venez de faire et la manière dont vous l'avez faite (dans un fichier partagé entre tous les membres de l'équipe).
3. Faites un rapport avec, au moins :
   a. Une page de garde avec l'information de l'équipe de travail

   b. Le cahier de charges
   c. Les nouvelles fonctionnalités que vous avez ajoutées au cahier de charges de base
   d. L'architecture du logiciel (au moins le diagramme représentant les modules/composants et classes de votre projet)
   e. Garanties de qualité telles que les tests, le plan de couverture des tests, des algorithmes d'optimisation si vous en avez implémentés, comment vous avez réussi à améliorer la généricité et l'extensibilité de vos implémentations, comment vous avez rendu votre projet facile à évoluer…)
   f. Méthode de travail (processus suivi, planning du projet, distribution des responsabilités, comment vous avez travaillé ensemble ...)
4. Déposer sous le Moodle du cours (si cette application ne fonctionne pas, veuillez envoyer vos projets dans un email aux enseignants du cours et UN LIEN POUR TÉLÉCHARGER LE PROJET ET LA DOCUMENTATION : surtout, n'incluez pas le projet en tant que pièce jointe de votre courriel, car les filtres de sécurité du serveur SMTP ne laisseront pas passer du code source) le fichier .zip et toute la documentation (ficher .docx ou PDF) de votre projet.
5. Veuillez respecter le dernier délai pour déposer votre projet (voir le planning dans les slides du cours)

Liste de contrôle (checklist) pour L'IMPLEMENTATION de votre projet

1. **Toutes** les fonctionnalités de mon projet son **correctement documentées** ?
2. Mon projet a deux interfaces (une interface en mode de comande et autre graphique) ?
3. Mon code respecte **la règle DRY** (Do not Repeat Yourself) ?
4. Mon projet a **au moins 10 héritages et 5 interfaces** (où cela soit possible pour l'utilisation actuelle ou pour des éventuelles évolutions de mon programme) ?
5. Mon code respecte **les principes de SOLID** https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898 ?
6. Mes tests unitaires **couvrent au moins le 90% des fonctionnalités** de mon programme ?
7. Mon code respecte **les conventions de nommage** des variables, des méthodes, des classes ?
8. Mon code traite **les exceptions** de manière adéquate et donner la meilleure résolution à chaque exception soulevée ?
9. Les données sont **correctement encapsulées** ?
10. Mon projet **est portable** (toutes les dépendances son sur un *dependency manager* de manière à ne pas avoir besoin de reconfigurer ou télécharger manuellement quoi que ce soit) ?

Conseils et consignes pour la présentation et démonstration du projet

Vous avez ***20 minutes de présentation et 10 minutes de questions par équipe*** pendant les deux dernières séances du cours pour faire la présentation (en équipe) de vos projets. La note et les commentaires la justifiant vous seront donnés à la fin de chaque présentation/démonstration.

1. Préparez-vous pour la présentation et démonstration de votre projet (avez-vous les connecteurs qu'il faut ? avez-vous assez de batterie pour faire la présentation sans interruptions ? vous êtes-vous coordonnés pour l'utilisation de la parole (qui présente quoi) ? …)
2. Respectez le temps alloué
3. Pour la partie présentation, préparez des slides présentant de manière succincte le cahier des charges, le problème que vous voulez résoudre et en particulier les nouvelles fonctionnalités que vous avez ajoutées au cahier de charges de base, l'architecture du logiciel (au moins le diagramme représentant les modules et classes de votre projet), des éléments de qualité (si vous avez fait des options donc les algorithmes d'optimisation, de généricité, d'extensibilité, etc…) et des éléments méthodologiques (planning du projet,

distribution des responsabilités, comment vous avez travaillé ensemble, refactoring du code...)

4. Pour la démonstration il faudra montrer en quoi le programme correspond au cahier de charges en déroulant des scénarios de simulation ; veuillez avoir un esprit critique et objectif de l'outil obtenu, du code produit, et veuillez présenter des perspectives d'évolution.
5. Il vous faudra répondre aux questions de manière claire, concise et avec du recul sur votre projet.

Project: Ecosystem – The case of a pond

Ecosystems are dynamic natural systems. Through their interactions with each other and with their biotope, living species transform the ecosystem, which thus evolves over time: it is a dynamic whole resulting from a co-evolution between life and its habitat. Although it tends to evolve towards a theoretically stable state, known as climactic, external events and pressures constantly divert it from this. The biocenosis then deploys its evolutionary and adaptive capacities in the face of a constantly changing ecological and abiotic context. This ability to withstand impacts without altering ecosystem structure, or to return to a previous state following a disturbance, is known as ecological resilience.

In a nutshell! Our ecosystem will therefore be a living whole of different species interrelated with each other and with their environment on a given spatial scale. For the project, we'll be adding different laws of behavior to aid survival, depending on the nature of each ecosystem.

For example, for human ecosystems, in 1974, philosopher and psychologist Anatole Rapaport of the University of Toronto put forward the idea that the most "effective" way to behave towards others is :

1) cooperation

2) reciprocity

3) forgiveness.

In other words, when an individual, a structure or a group encounters another individual, structure or group, it is in their best interest to propose an alliance. Then, according to the rule of reciprocity, it's important to give to the other in proportion to what you receive. If the other helps, we help; if the other aggresses, we must aggress in return, in the same way and with the same intensity. Finally, you must forgive and offer cooperation once again.

And for animal ecosystems, there are also rules linked both to the resources available and to the relationships between the animals themselves and with the resources. For example, the ecosystem's terrain will be a limited area containing various resources (water, grass, etc.). Several animals will evolve in this environment (herbivores, scavengers, predators). These animals will try to survive in this hostile environment, and to do so they will need to eat and drink every day.

All the animals will have common characteristics, such as speed, their necessary daily ration of food and water, their field of vision, and so on. On the other hand, animals will behave differently depending on their type.

Herbivores, for example, have a gregarious instinct and will therefore tend to seek out the company of their fellow creatures (when they're not looking for food). What's more, they flee as soon as a predator approaches. Their probability of detecting a predator is inversely proportional to the predator's distance.

There will be several types of predators, each with its own strategy for obtaining food: some will try to get as close as possible to their prey (with stealth, and therefore with a reduced probability of detection), then sprint to catch an animal, others will try to surround the herd (try to generate this behavior in a simplified way), ...

The simulation should be both command-line and graphical (using JavaFX, for example). It should also take into account the birth and death of animals (at least in a simplified way).

To give you an example of what a mini-ecosystem can be and how to implement it, please consider the following case of a pond.

Solution Description

Create files `Fly.java`, `Frog.java`, and `Pond.java` that will simulate how a pond ecosystem works. You will be creating a number of fields and methods for each file. Based on the description given for each variable and method, you will have to decide whether or not the variables/method should be static, and whether it should be private or public. To make these decisions, you should carefully follow the guidelines on these keywords as taught in lecture. In some cases, your program will still function with an incorrect keyword.

`Fly.java`

This Java file defines fly objects that exists within the pond.

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint:** there is a specific visibility modifier that can do this!

The `Fly` class must have these variables.

- `mass` - the mass of the `Fly` in grams (it must allow decimal values)
- `speed` - how quickly this `Fly` can maneuver through the air while flying, represented as a `double`

Constructors

You **must** use constructor chaining in at least two of your constructors. Duplicate code cannot exist in multiple constructors.

- A constructor that takes in `mass` and speed of a `Fly`.
- A constructor that takes in only `mass`.
    - By default, the `Fly` will have 10 speed.
- A constructor that takes in no parameters.

    - By default, the `Fly` will have 5 mass and 10 speed.

Methods

All methods must have the proper visibility to be used where it is specified they are used.

- Setters and getters (using appropriately named methods) for all variables in `Fly.java`.
- `toString` - takes in no parameters and returns a String describing the `Fly` as follows:
  (Note: replace the values in brackets [] with the actual value. Do not include the double quotes "" or the square brackets [] in the output. Specify all decimal values to 2 decimal points.)
    - If `mass` is 0: "I'm dead, but I used to be a fly with a speed of [`speed`]."
    - Otherwise: "I'm a speedy fly with [`speed`] speed and [`mass`] mass."
- `grow` - takes in an integer parameter representing the added `mass`. Then it increases the `mass of the Fly by the given number of mass.`
  `As mass increases, speed changes as follows:`
    - If `mass` is less than 20: increases `speed` by 1 for every `mass` the `Fly` grows until it reaches 20 `mass`.
    - If the `mass` is 20 or more: decreases `speed` by 0.5 for each mass unit added over 20.
- `isDead` – if `mass` is zero, return true. Otherwise, return false.

`Frog.java`

This Java file defines frog objects that exist within the pond.

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint:** there is a specific visibility modifier that can do this!

The `Frog` class must have these variables:

- `name` - the name of this `Frog`, which can be made of any combination of characters
- `age` - the age of the `Frog` as an integer number of months
- `tongueSpeed` - how quickly this `Frog`'s tongue can shoot out of its mouth, represented as a double
- `isFroglet` - a value that represents if this `Frog` is young enough to be a froglet (the stage between tadpole and adult frog), which must only have two possible

values - `true` or `false`. A `Frog` is a froglet if it is more than 1 month old but fewer than 7 months old. Whenever `age` is changed, this variable must be updated accordingly.

- `species` - the name of the species of this `Frog`, which must be the same for all instances of `Frog` (Hint: there is a keyword you can use to accomplish this). By default, its value must be “Rare Pepe”.

Constructors

You **must** use constructor chaining in your constructors. Duplicate code cannot exist in multiple constructors.

- A constructor that takes in `name`, `age`, and `tongueSpeed` and sets all variables appropriately.
- A constructor that takes in `name`, `ageInYears` representing the age of the `Frog` in years as a `double`, and `tongueSpeed` and sets all variables appropriately.
    - When converting `ageInYears` to `age` (in an integer number of months), round down to the nearest month without using any method calls (Hint: Java can automatically do this for you with casting).
- A constructor that takes in just a name.
    - By default, a `Frog` is 5 months old and has a `tongueSpeed` of 5.

Methods

You **must** use method overloading at least once. All methods must have the proper visibility to be used where it is specified they are used.

- `grow` - takes in a whole number parameter representing the number of months.
    - Then it ages the `Frog` by the given number of months and increases `tongueSpeed` by 1 for every month the `Frog` grows until it becomes 12 months old.
    - If the `Frog` is 30 months old or more, then decrease `tongueSpeed` by 1 for every month that it ages beyond 30 months.
    - You must not decrease `tongueSpeed` to less than 5.
    - Remember to update `isFroglet` accordingly
- `grow` - takes in no parameters and ages the `Frog` by one month and updates `tongueSpeed` accordingly as for the other `grow` method
- `eat` – takes in a parameter of a `Fly` to attempt to catch and eat.
    - Check if `Fly` is dead, and if it is dead then terminate the method.

    - The `Fly` is caught, if `tongueSpeed` is greater than the speed of the `Fly` the `Frog` ages by one month using the method `grow`. If the `Fly` is caught, the `mass` of the `Fly` must be set to 0.
    - If the `Fly` is NOT caught, the `mass` of the `Fly` must be increased by 1 while updating the `speed` of the `Fly` using only one `Fly` method.
- `toString` - returns a String describing the `Frog` as follows:
  (Note: replace the values in brackets [] with the actual value. Do not include the double quotes "" or the square brackets [] in the output. Specify all decimal values to 2 decimal points.)
    - If frog is a froglet, returns "My name is [`name`] and I'm a rare froglet! I'm [`age`] months old and my tongue has a speed of [`tongueSpeed`]."
    - Otherwise, returns "My name is [`name`] and I'm a rare frog. I'm [`age`] months old and my tongue has a speed of [`tongueSpeed`]."
- Setter and getter for `species` which must change the value for all instances of the class. Points will be deducted if you include an unnecessary or inappropriate setter/getter.

`Pond.java`

This Java file is a driver, meaning it will contain and run `Frog` and `Fly` objects and "drive" their values according to a simulated set of actions. You can also use it to test your code. We require the following in your `Pond` class:

- Main method must act as such:
    - Must create at least 4 `Frog` objects:
        - `Frog` with `name` Peepo.
        - `Frog` with `name` Pepe, `age` 10 months, and `tongueSpeed` of 15.
        - `Frog` with `name` Peepaw, `age` 4.6 years, and `tongueSpeed` of 5.
        - `Frog` of your liking :)
    - Must create at least 3 `Fly` objects:
        - `Fly` with 1 `mass` and `speed` of 3.
        - `Fly` with 6 `mass`.
        - `Fly` of your liking :)
    - Perform the following operations in this order:
        - Set the `species` of any `Frog` to "1331 Frogs"
        - Print out on a new line the description of the `Frog` named Peepo given by the `toString` method.

- Have the `Frog` named Peepo attempt to `eat` the `Fly` with a `mass` of 6.
- Print out on a new line the description of the `Fly` with a `mass` of 6 given by the `toString` method.
- Have the `Frog` named Peepo `grow` by 8 months.
- Have the `Frog` named Peepo attempt to `eat` the `Fly` with a `mass` of 6.
- Print out on a new line the description of the `Fly` with a `mass` of 6 given by the `toString` method.
- Print out on a new line the description of the `Frog` named Peepo given by the `toString` method.
- Print out on a new line the description of your own `Frog` given by the `toString` method.
- Have the `Frog` named Peepaw `grow` by 4 months.
- Print out on a new line the description of the `Frog` named Peepaw given by the `toString` method.
- Print out on a new line the description of the `Frog` named Pepe given by the `toString` method.

Logs (traces)

Save game results and stages in a file.

Tests

Add the tests needed to test the game/simulation

Rendu graphique

You should have two versions for the user interface: a command-line interface and a graphical interface. Before starting with the graphical interface, make sure that the interface in command-line mode and that your functional code is working properly! Beware: debugging can be potentially laborious, and many concepts are beyond the scope of this course.

Challenge: Transforming the Pond Ecosystem Simulation into an Interactive Game

Your task is to convert the existing ecosystem simulation, featuring a pond, frogs, and flies, into an interactive game with intuitive user interactions. Here's how you can approach this challenge:

1. **Interactive Elements:**

   Implement clickable functionality for both frogs and flies. When a player clicks on a fly, it should move, and when a frog is clicked, it should extend its tongue in an attempt to catch a nearby fly.

2. **Game Dynamics:**

   Implement a scoring system. Players earn points when a frog catches a fly. Display the score prominently on the screen.

Consider adding levels or challenges. For example, as the game progresses, the speed of flies could increase, or more frogs could be added to the pond.

3. Visual and Audio Enhancements:

Enhance the visual and audio elements to make the game more immersive. Add animations for frog tongue extension and realistic fly movement. Incorporate sound effects like frog croaks and buzzing flies to create a lively atmosphere.

4. User Experience:

Focus on creating a seamless and enjoyable user experience. Ensure that the game controls are intuitive and responsive.