Homework #3: Virtual World

Due Thursday, September 25th at 11:59 p.m.

In this assignment you will complete and extend a virtual world. This world contains many different items—animate and inanimate—which can interact with each other in complex ways. For example, `Fox`es hunt for `Rabbit`s, which in turn try to outwit their predators. It's up to you to determine just how complex the interaction will be.

Your goals for this assignment are to:

- Demonstrate mastery of earlier learning goals, especially the concepts of encapsulation and polymorphism, software design based on informal specifications, and Java coding and testing practices and style.
- Use inheritance and delegation effectively to achieve design flexibility and code reuse.
- Apply design principles such as low coupling and high cohesion to achieve informally-specified design goals.
- Learn and apply the Template Method design pattern.
- Explicitly specify behavioral contracts and test against those specifications.
- Gain experience working with a medium-sized application, including programming against existing interfaces and implementations.

Overview

We have developed a virtual world environment that can simulate the interaction of many items and actors. This world is flat and consists of many fields that can each have one `Item`. In the beginning the world will contain `Grass`, `Rabbit`s, and `Fox`es, and you will add additional `Item`s as part of this assignment. Time in the virtual world simulation progresses in discrete steps; in every step an `Actor` may act, for example, by moving, eating, or breeding. We have provided some simple `AI`s for rabbits and foxes; you will implement more intelligent `AI`s for them and other items you create.

This assignment has three parts. In the first part you will refactor the `Rabbit` and `Fox` classes, using the Template Method pattern to maximize code reuse. In Part 2 you will fill the world with additional items; in this part you will also specify the behavioral contracts for and test some of your items. Finally, in Part 3 you will create intelligent `AI`s for `Rabbit`s and `Fox`es.

Part 1: Maximize code reuse with the Template Method design pattern

The Template Method design pattern is a common design pattern that allows a superclass method to specialize its behavior for subclasses while ensuring that the superclass's overall high-level algorithm is still followed. See Chapter 19 in *Design Patterns Explained*, our required course textbook, for more details on the Template Method pattern.

In our initial code there is substantial duplication in the `Rabbit#breed()` and `Fox#breed()` methods. Use the Template Method pattern to eliminate this duplication. You may modify any classes and interfaces in the `edu.cmu.cs.cs214.hw3.items` package (and its subpackages) necessary to maximize reuse.

Part 2: Fill the virtual world with life

This part has two major requirements: (1) you will add many new classes to the virtual world, and (2) you will specify the behavior of and test some of your new class implementations.

Add at least 9 new classes to the virtual world—three `Item` classes for each of the three different categories listed below:

- **Animals**: In the `animals` package create three additional classes for animals, such as lions, flies, elephants, etc.
- **Vehicles**: In the `vehicles` package create three classes for vehicles, which can run over (destroy) everything with a lower strength but will crash (be destroyed) when running into something with a greater strength. Like real vehicles, your vehicles should build momentum when moving, so it takes time for them to accelerate or brake or turn; they can change directions only at low speed. Note that the speed of a `Vehicle` is controlled by the cool-down period.
- **Your own category**: In a separate package implement three classes of `Item`s that share some similarity. Examples might include tornadoes and earthquake, mountains and cliffs, Master Yoda and Skywalker... Use your imagination!

You have considerable freedom in this assignment for which items you add and how your items behave. Your items might range from a simple stone to sophisticated characters and weapon systems, from real-world animals to science fiction creatures, or include technical objects.

Overall, design your items so that you can achieve code reuse with inheritance and/or delegation. We strongly recommend that you introduce additional interfaces or abstract classes to achieve appropriate modeling flexibility and code reuse. If your additional animals, vehicles, and category do not allow you to demonstrate modeling flexibility and code

reuse with inheritance and delegation, you should rethink your design so that you can successfully demonstrate the key learning goals of this assignment.

You must specify the behavior of, and then test, all items in your **Vehicles** category. Write your specification in natural language as a comment for each class and comments for each method. Your specification should include all preconditions, postconditions, and invariants for your vehicles' designs, and also specify appropriate exceptional behavior for abnormal conditions such as when preconditions are violated. Submit your tests as JUnit tests and follow best practices for Java testing, including an appropriate directory and package structure for testing, importing any necessary libraries for JUnit, and configuration of `ant` to build and test your solution on Travis-CI.

Part 3: Create intelligent `AI`s for `Rabbit`s and `Fox`es

Provide a more intelligent behavior for `Rabbit`s and `Fox`es by providing an implementation of their `AI` classes. In a competition, we will add rabbits and foxes from all students into a large virtual world (otherwise empty except for `Grass`) and evaluate their success over a large period of time (e.g., 10,000 steps). The best AIs are those that generate the largest average animal population over the entire time, measured separately for rabbits and foxes. To make this a fair competition, there are restrictions on how rabbits and foxes can behave (eat, breed, and move), and you will provide the AI under those restrictions.

Be warned: This part is a small part of our overall evaluation of your solution, but you may (if you want) invest much time developing high-quality AIs. You should not invest time developing high-quality AIs until you have successfully completed Parts 1 and 2 of this assignment.

Do not develop AIs for any items except the `Rabbit` and `Fox`.

The virtual world

This section describes the design of the virtual world and its rules.

Objects in the world

The world contains the following object types, with each type having different properties and specifications described here.

- **Item:** An `Item` represents a physical object in the virtual world that occupies a field in a specific location, where it is represented with a picture. For example, `Fox`es, `Rabbit`s, and `Grass` are `Item`s.

- **Actor:** An `Actor` can actively affect the state of the world. Many `Item`s are `Actor`s; they can decide to move, eat, breed, or perform other actions. The world regularly determines each `Actor`'s next action by calling `getNextAction`; the `Actor` returns a `Command` that represents the next action. `Actors` can act at different speeds, acting on every step in the world or only every nth step—the speed of an actor is determined by its cool-down period.
- **ArenaAnimal:** For the arena competition rabbits and foxes are special, and we already provide an implementation of them. Rabbits and foxes can only see the immediate world around them, determined by a view range that defines a viewable square surrounding the animal; e.g., a view range of 2 means the animal can view a 5 by 5 square, with the animal at the square's center. They have energy, starting with a default value and increasing when they eat something, but also slowly decreasing each time they act. Your additional items may behave like arena animals, but are not required to do so.

Additionally, you will find interfaces representing `MoveableItem`s (that can be moved to an adjacent location at each step), `LivingItem`s (that are actors which can move, eat, breed, and have energy), `Food` (representing edible items providing plant or meat-based calories), and so forth.

There are also implementations of `Grass`, `Gardener`s, `Rabbit`s (eating `Grass`), `Fox`es (eating `Rabbit`s), and `Gnat`s (generally behaving just stupidly) provided in the world already. You may change these implementations to foster reuse, but the behavior of these existing items should not be changed. For rabbits and foxes you should provide corresponding AIs that survive the best in the arena by implementing the `AI` interfaces.

Commands and behaviors

All actors are periodically asked for their next action, which they provide by returning a `Command`. How often they are asked depends on their speed; `getCoolDownPeriod` returns the number of steps to be skipped before the next action. AIs of rabbits and foxes may only return instances of the predefined commands `BreedCommand`, `EatCommand`, `MoveCommand`, and `WaitCommand`. The following are the predefined rules for breeding, eating, and moving that apply to rabbits and foxes:

- **BreedCommand:** When an `ArenaAnimal` breeds, it makes a copy of itself on an adjacent tile (one of the 8 tiles around it). The `LivingItem` can only breed when it has enough energy (`getBreedLimit` returns the minimum required energy to breed) and has a valid empty adjacent location. Breeding occurs alone; there is no mating between rabbits or between foxes. When an `ArenaAnimal` breeds, its energy is reduced to 50 percent of its former energy (rounded down), and the newly-bred animal

also starts at 50 percent of its parent's energy. Finally, a newly bred `ArenaAnimal` must be placed in an empty location that is adjacent to the parent.

- **EatCommand:** An `ArenaAnimal` is only able to eat something that is adjacent to it. By eating a `Food`, the `ArenaAnimal` increases its energy by the food's calories (plant calories for rabbits and meat for foxes) up to its maximum energy level. The food (vegetable/non-vegetable) must be edible by the eater (herbivorous/carnivorous) and the eater must possess greater strength than the food, i.e., foxes should not attempt to eat grass or other foxes.
- **MoveCommand:** An `ArenaAnimal` can only move once at a time, and moving distance is restricted by its moving range. Also, it must move only to valid, empty locations.
- **WaitCommand:** Simply doing nothing is the final option. Note that all living items lose energy each time `getNextAction` is called, even if they choose to do nothing, so they may eventually die of hunger.

The above rules needed to be obeyed strictly by the AI for rabbits and foxes competing in the arena. Your own items may handle them more flexibly (e.g., they may jump around further on the world) and may add additional `Command`s.

Implementing items with reuse

When implementing your own items you should maximize code reuse. You may modify the implementation (but not the behavior) of existing items. Your implementation of your own items should avoid duplicated code.

Implementing the AI for the arena

You must implement the AI for rabbits and foxes by implementing the `AI` interface. For technical reasons, your AI classes must have a zero-argument constructor (you cannot participate in our virtual world tournament if your `AI`s require constructor arguments). The AI for rabbits and foxes is restricted in flexibility compared what other actors can do. They can only see nearby parts of the world through the `ArenaWorld` interface and may only return the predefined commands obeying the rules above.

Note that the AI should only rely on the interface contracts of arena animals, but not on specific implementations. For instance, we may chose to modify the size of the world, the energy limits, or the view ranges of animals in the actual competition. Returning invalid commands or attempting to cheat by casting the `ArenaWorld` to `World`, or casting other objects to specific implementations, may lead to the exclusion from the competition.

The world

The `World` class is the core engine of the game. It tracks all items (and removes dead ones) and actors. The world regularly gets the next action for each actor and performs the actions. The world organizes items in a 2-dimensional grid ($n \times n$, for arbitrary n) of `Items`, with the top left corner being $(0, 0)$. Locations are represented by the `Location` class, which contains several potentially useful methods. We also provide a utility class with potentially useful functionality.

We provide a GUI to visualize the world with its items. The GUI has a simple interface containing two buttons: a "Step" to execute a single step; and a "Start/Stop" toggle button to run indefinitely until the toggle button is pressed again. For completing the homework, it is not necessary to understand the implementation of the GUI.

To initialize the world with your items, modify the `Main` class. In the arena competition, we will initialize a large virtual world with grass and all competing rabbits and foxes.

Evaluation

Overall this homework is worth 120 points, plus up to 10 points of extra credit. To earn full credit you must do the following:

- Design your items to reuse as much code as possible. Avoid copying-and-pasting code; instead, design your code with useful abstractions, class hierarchies, and/or delegation.
- You must name your Fox AI class `FoxAI` and your Rabbit AI class `RabbitAI`, including the exact capitalization here. Place both classes in the `edu.cmu.cs.cs214.hw2.ai` package. Both classes must implement the provided `AI` interface and have a constructor that requires no parameters.
- Your AIs may rely on the `ArenaAnimal` interfaces we provide with the assignment, but they should not depend on specific implementations of these interfaces. For the arena-style tournament, your RabbitAI will be used to control some provided rabbit implementation and your FoxAI will control some provided fox implementation. If you unsure whether your implementation will work in our arena, please ask the course staff using a private question on Piazza.
- In general, adhere to the code organization. You may add classes, abstract classes, and interfaces that you desire to the code base. Place any new files in the appropriate package. For example, a `Snake` would go in the `animals` package, etc.
- You may not delete or modify the behavior of methods in any existing interfaces.

- As usual, make sure your code is readable. Use proper indentation and whitespace, abide by standard Java naming conventions, and add additional comments as necessary to document your code. Hint: use **Ctrl + Shift + F** to auto-format your code!

Additional hints:

- The tasks may be underspecified. In case of doubt use your judgment. If you want to communicate your assumptions, use comments in the source code or a README.md file in your homework directory.
- Avoid using `instanceof` and downcasts. Avoid casting an interface to a specific implementation. Do not use the `java.lang.Class` class or the `java.lang.reflect` package. You do not need—and should not use—those techniques for this homework.
- Aside from testing your vehicles, we do not have any testing-related requirements. You may write test code for other parts of your implementation, but they will not be an evaluated part of this assignment.

We will use the following approximate rubric to grade your work:

- A complete implementation matching our specification, including all nine additional item classes and working `RabbitAI` and `FoxAI` implementations: 44 points
- Program design that demonstrates understanding of design principles and the use of inheritance and delegation to achieve design flexibility and code reuse: 30 points
- A solution that demonstrates understanding of the use of the Template Method pattern to achieve code reuse: 12 points
- Behavioral specification and testing of your vehicles classes: 24 points
- Java coding and testing practices and style, including a solution that builds and tests on Travis-CI: 10 points
- Bonus: up to 10 points. After the homework deadline, we will run all students' `Rabbit` and `Fox` AIs in our own virtual arena with our own `Fox` and `Rabbit` implementations and award bonus points to the best competitors. Our animal implementations may use different values for the cool-down periods, the breed limit and similar values. Our `World` implementation will strictly enforce the `World` rules such as the `ArenaAnimal` view limits. Your solution will be disqualified from the tournament if your implementations do not respect the rules (such as the view limit) of our world.

Have fun!