Assignment 6, CSC/CPE 203

For this assignment you must modify the pathing behavior of all entities that move within the world.

Objectives

- Practice using a visitor pattern, specifically with the use of generic types, in order to eliminate all `instanceof/isInstance` code in the project.
- To modify the code to use the specified `PathingStrategy interface` (that in turn uses `streams` to build a list of neighbors)
- Further to integrate the use of this pathing strategy and understand the associated code example which uses `filter and collect`
- Implement A star pathing algorithm in the exisiting code by implementing a new `PathingStrategy` subclass building off prior exercises.

Overview

This assignment deviates from the pattern of previous assignments. Though this assignment does introduce/leverage some design strategies, the primary goal is to improve the functionality of some entities in the virtual world.

In particular, as you are likely very aware of by now, the blobs and miners movement is very simplistic. You have likely seen an entity get stuck on an obstacle or on another entity. You will improve the pathing strategy as part of this assignment.

Pathing algorithms are quite interesting, in and of themselves, but our exploration of pathing in this assignment also motivates the use of some design patterns and techniques. Applying these patterns will also improve the flexibility of the implementation.

Eliminating `instanceof`/`isInstance` — Visitor Pattern

Deriving from the original given code, your implementation of the project currently contains some uses of `isInstance` and `instanceof` (not counting uses within an `equals` method). For instance, based on the specifications of Assignment 4, `findNearest` takes a `Class` object and then uses the `isInstance` method to find the desired entity.

Instance checks, as currently used in the project, have two drawbacks. First, they circumvent the static typecheckeing algorithm (sometimes this is what you want) deferring error detection to runtime. Second, they decrease flexibility in that only a single type (hierarchy) will satisfy the check. Thus, if a miner wanted to search for, as an example, the nearest ore or blob, one would need to modify the code shared by multiple entities. Eliminating these checks is one interesting use of the Visitor pattern; and once the framework for the pattern is introduced, one can use Visitors for various other purposes (often as an alternative to adding a method to a class hierarchy).

Follow these steps

1. To the root of your Entity hierarchy, add an abstract method as follows (if your root is an `interface`, then the declaration is implicitly `public` and `abstract`).

```
public abstract <R> R accept(EntityVisitor<R> visitor);
```

This method accepts a visitor object that will return a value of some (parameterized) type `R` given an entity object. For instance, you might have a visitor that returns a `Boolean` `true` when given an `Ore` entity object, but `false` otherwise.

2. For each concrete (i.e., not `abstract`) entity, define the `accept` method as follows.

```

   public <R> R accept(EntityVisitor<R> visitor)
   {
      return visitor.visit(this);
   }
```

This method simply passes the current object to the visitor's `visit` method. But why? Well, within the concrete class, the actual type of the object (e.g., OreBlob) is known. As such, if the `EntityVisitor` overloads the `visit` method for each concrete entity class, then the `visit` method invoked will be particular to the entity type (i.e., the method taking an argument of which `this` is an instance will be invoked).

7. Define the `EntityVisitor` interface with a `visit` method for each concrete entity class.

```

   interface EntityVisitor<R>
      {
         R visit(Ore ore);
         ...
      }
```

14. Define an `AllFalseEntityVisitor` implementation of the `EntityVisitor` to return `false` from every `visit` method. This is a convenience class providing default implementations for each method so that subclasses need only override those that are meant to return `true`.
15. Find and replace all uses of `instanceof` and `isInstance` with uses of appropriate visitor objects (you will need to define the classes for these).

Supporting Variety — Strategy Pattern

When an entity attempts to move, it needs to know the next step to take. How that next step is computed is, in many respects, irrelevant to the code within the corresponding entity. In fact, we may want to change that strategy for different builds of the program (to experiment), each time the program is executed (based on configuration), or dynamically during execution. The Strategy pattern allows you to encapsulate each pathing algorithm and switch between them as desired.

Your implementation must use the given PathingStrategy interface (discussed below).

```
interface PathingStrategy
{
   /*
    * Returns a prefix of a path from the start point to a point
within reach
    * of the end point.  This path is only valid ("clear") when
returned, but
    * may be invalidated by movement of other entities.
    *
    * The prefix includes neither the start point nor the end
point.
    */
   List<Point> computePath(Point start, Point end,
      Predicate<Point> canPassThrough,
      BiPredicate<Point, Point> withinReach,
      Function<Point, Stream<Point>> potentialNeighbors);

   static final Function<Point, Stream<Point>> CARDINAL_NEIGHBORS
=
      point ->
         Stream.<Point>builder()
            .add(new Point(point.x, point.y - 1))
            .add(new Point(point.x, point.y + 1))
            .add(new Point(point.x - 1, point.y))
            .add(new Point(point.x + 1, point.y))
            .build();
}
```

This strategy declares only a single method, `computePath`, to compute a path of points (returned as a list) from the start point to the end point (this is only expected to be a prefix, excluding the start and end points, of a real path; it need not represent a full path).

In order to compute this path, the pathing algorithm needs to know the directions in which travel might be able to proceed (determined by `potentialNeighbors`). In addition, in order to explore potential paths, the pathing algorithm must be able to determine if a given point can be traversed (i.e., is both a valid position in the world and a location to which the traveler can move; determined by `canPassThrough`). Finally, it is unlikely that the pathing algorithm should actually attempt to move to the `end` point (it is quite likely occupied, of course). Instead, the pathing algorithm will determine that a path is complete when a point is reached that is `withinReach` of the `end` point.

Single-Step Pathing

As an example of defining a pathing strategy, consider the following implementation of the single-step pathing algorithm (SingleStepPathingStrategy) used to this point by the pathing entities (this specific implementation leverages the stream library).

Modify the appropriate entities to use a `PathingStrategy` (referencing the `interface`, of course). Use the given implementation to verify that your changes work.

```
class SingleStepPathingStrategy
   implements PathingStrategy
{
   public List<Point> computePath(Point start, Point end,
      Predicate<Point> canPassThrough,
      BiPredicate<Point, Point> withinReach,
      Function<Point, Stream<Point>> potentialNeighbors)
   {
      /* Does not check withinReach.  Since only a single step is
taken
       * on each call, the caller will need to check if the
destination
       * has been reached.
       */
      return potentialNeighbors.apply(start)
         .filter(canPassThrough)
         .filter(pt ->
            !pt.equals(start)
            && !pt.equals(end)
            && Math.abs(end.x - pt.x) <= Math.abs(end.x -
start.x)
            && Math.abs(end.y - pt.y) <= Math.abs(end.y -
start.y))
         .limit(1)
         .collect(Collectors.toList());
   }
}
```

Of course, this implementation only matches the original pathing algorithm if `potentialNeighbors` returns the same neighbor points (in the same order) as before. Experiment with adding other points to the `Stream` returned by `potentialNeighbors`; perhaps allow the addition of diagonal movement, only allow diagonal movement, or remove the option to move straight up or down and replace them with the corresponding diagonals. Each of these approaches can be tried simply by changing the function passed to `computePath`.

A* Pathing

Define a new `PathingStrategy` subclass that implements the A* search algorithm. As before, an entity will take only one step along the computed path so the `computePath` method will be invoked multiple times to allow movement to the intended destination (see below for alternatives). As such, take care in how you maintain state relevant to the algorithm.

Testing

You are strongly encouraged to write unit tests for this strategy. Since your implementation must conform to a specified interface, part of the grading will be based on instructor unit tests.

Alternate Traversal Approaches

After completing the above, you might notice an indecisive miner ping-ponging between two points. This is an artifact of attempting to move to the nearest ore and only following one step of any computed path. That one step moves the miner closer to a different ore, which results in the computation of a new path ... that brings the miner right back to the previous point.

Consider some alternatives (implementation of these is entirely optional; any such changes will be in the entity code, not in the pathing strategy).

- Non-fickle: Once a path is computed, continue to follow that path as long as the target entity (e.g., ore) has not been collected by another. This approach skips the check for the "nearest target" as long as the previous target is available.
- Determined: Once a path is computed, follow it to the end. This approach skips the check for "nearest target" until a new path must be computed.
- Ol' College Try: Once a path is computed, follow it at least X steps (or until exhausted) before giving up. This approach skips the check for "nearest" target until it has consumed a fixed number of steps (e.g., five) in the current path (or it has consumed the entire path). After this initial effort, if the destination has not been reached, then check for the "nearest target" and compute a new path.

Warning: Of course, it is **important** to note that an implementation of any of these alternate approaches (since each continues to traverse a computed path) must take care to not move into an occupied cell. Keep in mind that the path was clear when it was originally computed, but other entities will move during this path traversal.

Assignment Submission

Your submission must include all source files (even those that were unchanged). Your grader should be able to build your project based on the files submitted. (You do not need to submit the image files, the image list, or the world save file.) An explicit list of files is not given because you are creating new files for this assignment, so verify that you have submitted everything properly.