~~Forest Project~~ YOUR GAME(Project 4)

For the first part of this project (Part 1/2) you must modify the pathing behavior of all entities that move within the world where at least one entity moves using A*.
Furthermore, you will modify the virtual world to your liking and under consideration of certain elements stated below. If your "game" is incompatible with those requirements stated in Part 2/2 reach out to your instructor and discuss if it will satisfy the requirements (at least of the same difficulty and complexity level).

Objectives

- 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 existing code by implementing a new `PathingStrategy` subclass building off prior exercises.
- Add new functionality to existing code base demonstrating an understanding of the existing design and functionality
- Be able to evaluate the current design based on the experience of adding to the code

Overview

This assignment deviates from the pattern of previous assignments. The primary goal is to improve the functionality of some entities in the virtual world by changing their pathing strategy as well as extending and modifying the game under carefully adhering to the design principles learned this quarter.

In particular, as you are likely very aware of by now, some entities 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.

Part 1/2 – Strategy Design Pattern

NOTE: You're AStar Algorithm of Part 1 is due BEFORE Part 2. You will have to use handin to submit your AStar and you will get autograder feedback. There is no collaboration with anyone allowed for your AStar algorithm.

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 called `AStarPathingStrategy` 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.

Additionally, the code for the testing program is found on Canvas. This is a great way to visualize the path your AStar code is returning.

Alternative Traversal Approaches

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 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.

Part 2/2 – Modifying the Game (the fun part)

- The world must look entirely different. You are free to modify all pictures however you like and are encouraged to be creative.

- Create a "world changing" event triggered by a mouse click that affects at least 2 tiles of the world in proximity to the mouse position when the click occurs. The event should affect no more than half of the world.
  The world event must be visualized by modifying the background image of the affected tiles.
  At least one of the existing mobile entities must be affected by the world event, based on the proximity to the event location. More specifically, this type of entity should change in appearance and behavior.
  The world event must cause a new type of mobile entity to spawn. This new entity should animate and move according to logic defined by you.

- Utilize the cursor keys to make an entity move around the world.

Project Partner???

Yes! You may work with **one** partner on this part of the project. Choose the code base of one of the partners to add to, and then both work on the project together. Both team members must contribute code to the project. See below.

Note that you do not have to work with a partner. You may choose to work alone.

Include a text file named `WORLD_EVENT.txt` in your submission that describes:

1. how to trigger the event
2. what should happen when the event is triggered
3. what the affected entities should do
4. what the new entity is and how it should behave
5. (if applicable) the names of each partner and what each partner was responsible for

Extra Credit???

Yes! You may go all out and create a truly awesome game that is fun to play and incooperates a bit more such as:

- At least 3 different kind of entities where one kind is moving using a "new" pathing algorithm which is not SingleStep or AStar, preferably Dijkstra for full EC credit.
- Must use Factory Design Pattern.
- Explanation of what other design you used. If yes, which one(s)?
- SUBMIT a short video clip of your game in action while explaining the game mechanics. Describe how it works! (if this is not submitted- you won't get any extra credit- absolutely NO exception)

If you have a different idea that will not allow to follow the requirements- please check with me first!

Grading

2 pts - Your code correctly uses the PathingStrategy interface and SingleStepPathingStrategy. Do this first and get it working before moving on to AStar!

3 pts - Your AStarPathingStrategy works correctly both with your project and in the given testing program.

10 pts - Modifications to your Game (Part 2/2)

5 pts - possible Extra Credit

Submission

For Part 1/2 there is a handin requirement for your AStar Testing Program posted on Canvas. Instructions will be given in lab/lecture.

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 need to submit the image files, the image list, or the world save file as you likely will have to modify them for Part 2) 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.

There is a UML required for Project 3. You are allowed to use IntelIJ feature the create the UML for you.

If you work with a partner, only submit once indicating in the submission comments the name of your partner.

Video Clip if you go for Extra Credit.