Assignment 3: Static and Dynamic Entities

Due 3/7, 2016 10:00am

1 Overview

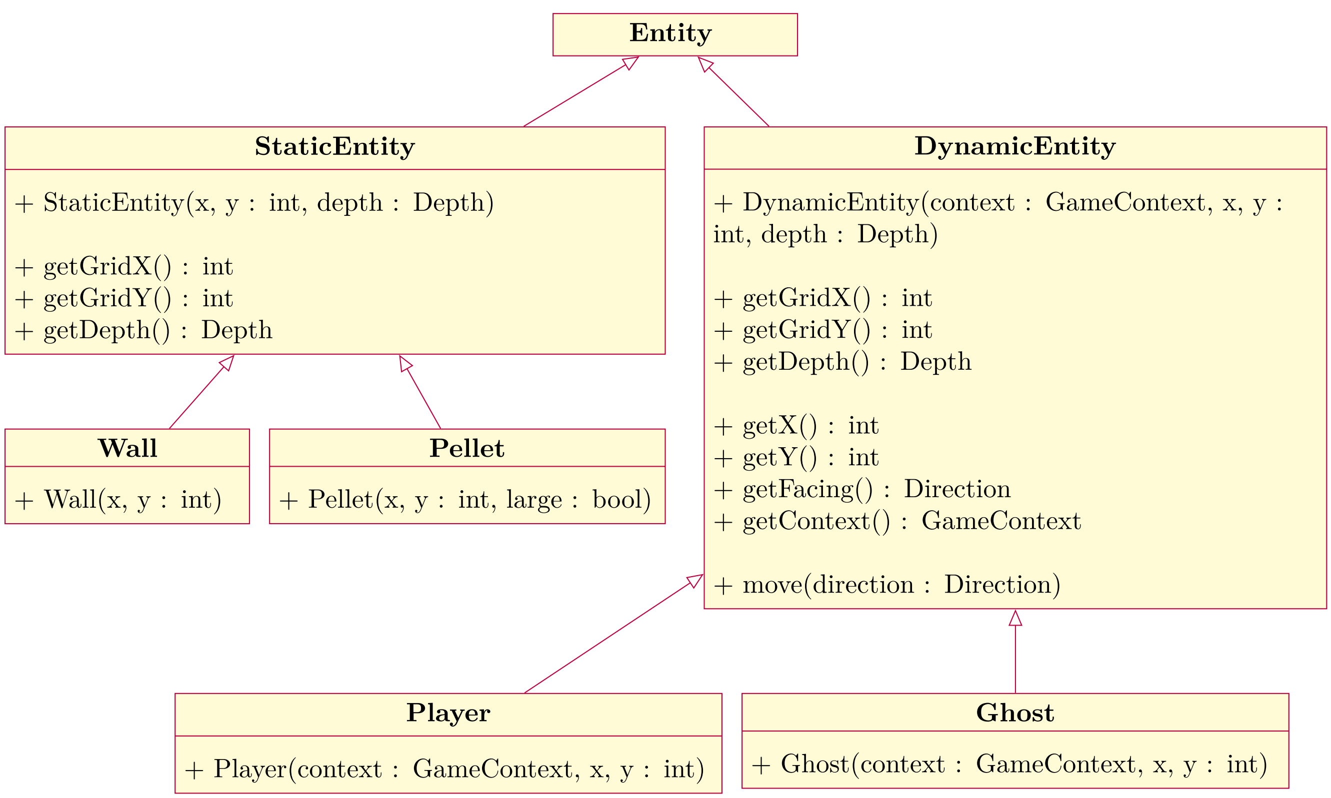
In this assignment you will be adding entities for pellets (*nom nom!*) and ghosts. This will require us to split the *entity hierarchy* into something more manageable, and moving *common functionality* into superclasses.

2 What has changed

Take a look at the template and compare it to your solution. The `GameContext` class has an instance variable (and an associated accessor/mutator pair) for the *player* of the game. Notice that a `setPlayer` method implies that there is (*at most*) one player (*at a time*) relevant to a game.

3 What you need to do

We will be fleshing out the *inheritance hierarchy* of the game entities. Namely, we are going to break entities into two coarse groups: ones that move (*dynamic* entities), and ones that don't (*static* entities). The class hierarchy should look something like the following.



(1) The `Wall` and `Player` classes both currently extend the `Entity` class. You should create a `StaticEntity` and a `DynamicEntity` class, and change the *parents* of the four classes to match the hierarchy above.

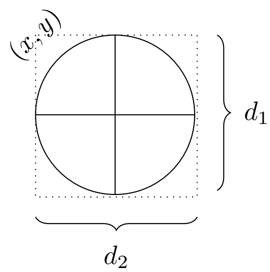
(2) Now, determine what parts of the implementation of `Wall` and `Player` can be moved to the superclasses `StaticEntity` and `DynamicEntity` (respectively). These should be the state and behavior that should apply to **all** subclasses of this superclass. Follow the UML diagram above for specific instructions.

However, not **all** of the methods should be moved to the super class. Specifically, the `draw` methods of each concrete subclass will be unique, as will the `update` method for the dynamic entities. Also notice that walls are the **only** solid object in the game (you can move over pellets, and we will handle player-ghost collisions in a later assignment).

Notice that the new superclasses have a constructor taking a *depth*. This way no subclass needs to implement the `getDepth` method. This value can be supplied to the superclass in the subclass's constructor. For example, the `Player`'s constuctor should look like the following:

```
public Player(GameContext context, int x, int y) {
    super(context, x, y, Depth.FRONT);
}
```

(3) You will now need to make two additional concrete subclasses: `Pellet` and `Ghost`. The pellet sits at the *middle* depth (on top of walls but below the player). The pellet should be drawn as a yellow circle with a radius of 3 (`large = false`) or 7 (`large = true`). You can draw a colored circle by calling the `fillOval` method of the graphics object and giving it the parameters representing: (x, y, d_1, d_2). To draw a circle from an oval, simply set $d_1 = d_2 = \textit{target radius} \times 2$.



The ghost sits at the *front* depth, with the player, and should be rendered as a blue square (in the same manner as walls and the player). Ghosts must additionally *update*, but should not use the `input` argument as the player does (otherwise Pacman and all the ghosts will be moving the same ways, but offset their initial positions). Instead we'll make a rudimentary AI based on a *drunken walk*. The ghost will move around the grid randomly based on the following rules:

1. If the ghost is at an intersection, it will choose a direction to move *at random*
2. If the ghost is **not** at an intersection, it will continue to move according to its current facing

You can determine if a ghost is at an intersection by determining if its *pixel coordinates* are both multiples of the grid size. You can get the ghosts's current facing by using a new methods inside `DynamicEntity`.

4 Submission

Create a *zip archive* of your Eclipse project (including all template files) and upload it to the correct D2L dropbox before the due date. Again, no late work will be accepted.

(4 - Double Credit) For up to 100% extra credit you may implement a *chasing AI* for the ghost. The solution does this in about 32 lines of code (simply replacing the random choice part of the `update` method). This additional feature can be submitted to the `Assignment 3b` dropbox before `3/28`.

The chasing algorithm for *Blinky* (the red ghost) in the original Pacman follows the algorithm described here. At each intersection (defined the same as above), Blinky will choose a new facing according to *lowest Manhattan Distance* (the sum of the x and y distances) of a free cell adjacent to the ghost and the player's *current* cell. You can get the player from the context object. The ghost may move backwards **only** if trapped (the ghost will never do a 180-degree turn unless it reaches a dead-end).

The implementation of this algorithm will be left (almost) completely up to you. No additional direction will be given, but feel free to contact one of us for a subtle push in the right direction.