1/17/2019

Homework

Pacman3

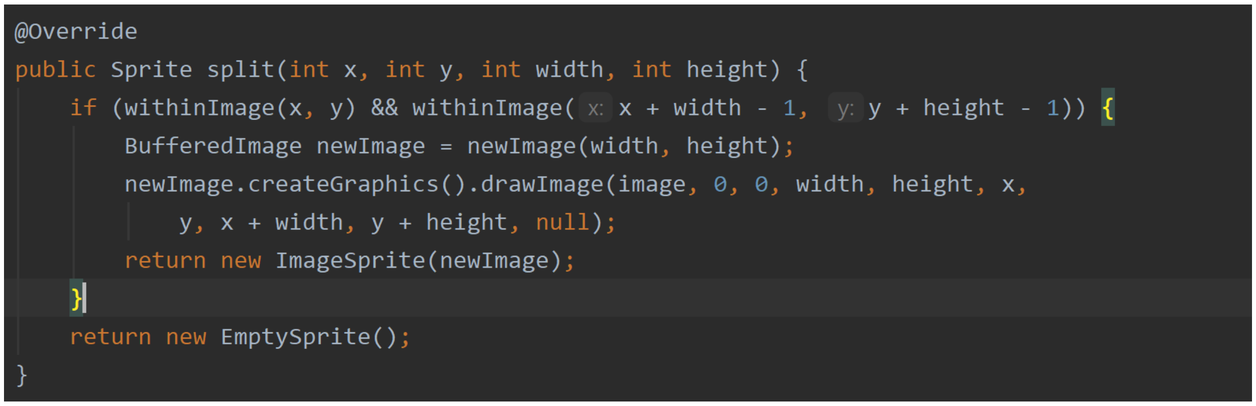
Rafael Oliveira
UNIVERSITY OF CALIFORNIA, IRVINE

Rafael Oliveira
SWE265

What is the role of EmptySprite?

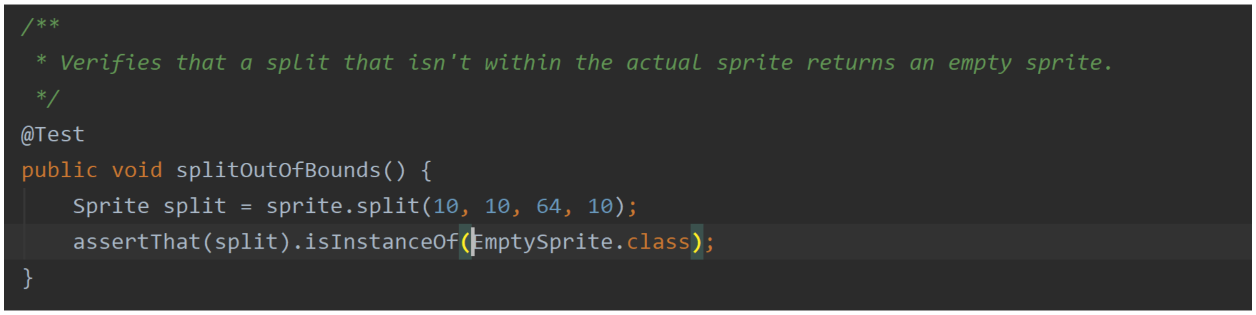
EmptySprite is an empty sprite that holds data. When it is drawn on the game screen, it will not appear, since there is no real sprite art bound to it at any point in the game. EmptySprite objects are created in only three classes in the program: ImageSprite, AnimatedSprite, and SpriteTest. ImageSprite, AnimatedSprite, and EmptySprite implement the same Sprite class.

The ImageSprite class has a method where if a sprite is split outside of bounds, it will become an empty sprite.



```java
@Override
public Sprite split(int x, int y, int width, int height) {
    if (withinImage(x, y) && withinImage( x: x + width - 1,  y: y + height - 1)) {
        BufferedImage newImage = newImage(width, height);
        newImage.createGraphics().drawImage(image, 0, 0, width, height, x,
            y, x + width, y + height, null);
        return new ImageSprite(newImage);
    }
    return new EmptySprite();
}
```

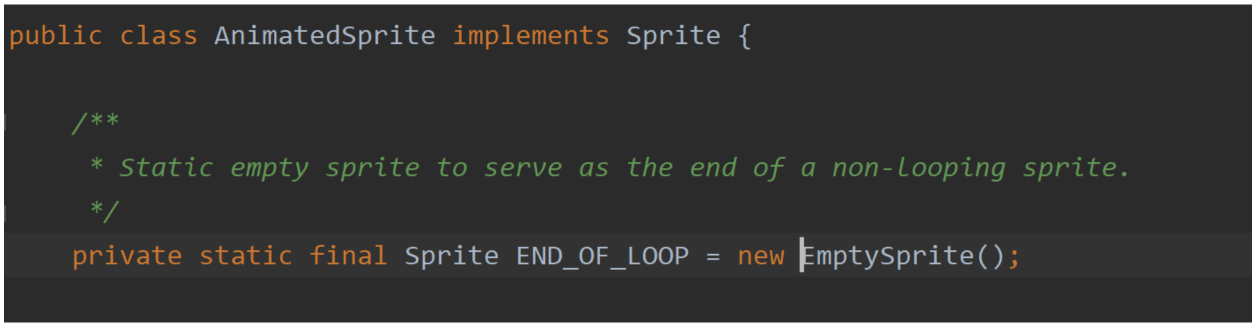
This is also tested in the SpriteTest class.



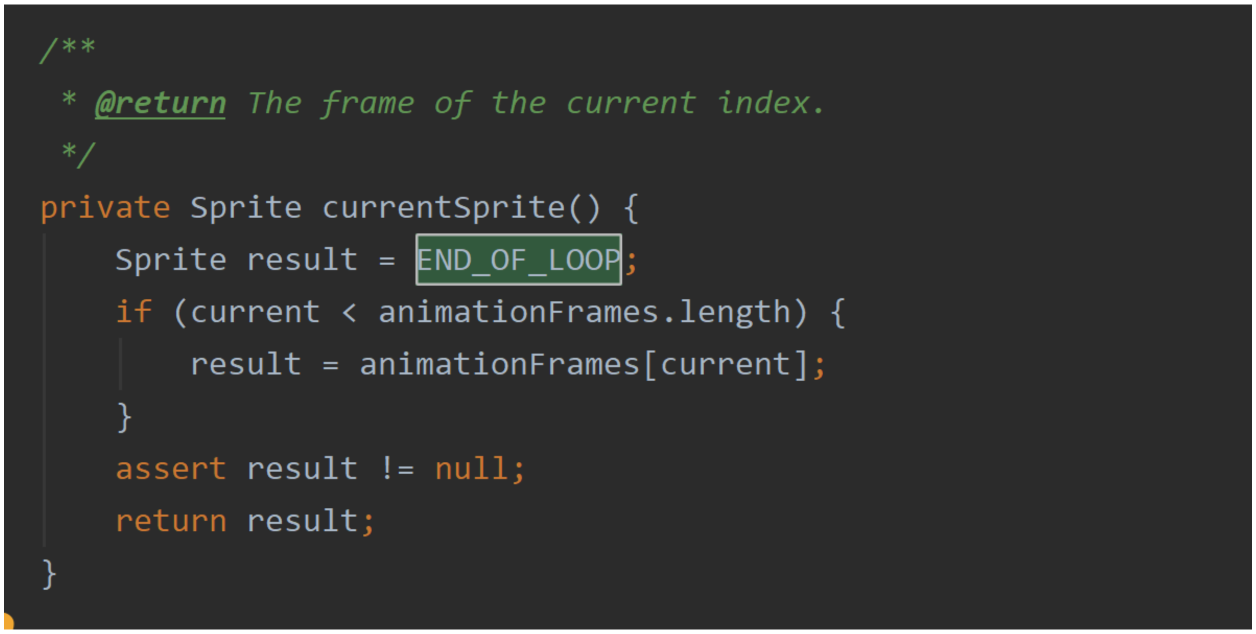
```java
/**
 * Verifies that a split that isn't within the actual sprite returns an empty sprite.
 */
@Test
public void splitOutOfBounds() {
    Sprite split = sprite.split(10, 10, 64, 10);
    assertThat(split).isInstanceOf(EmptySprite.class);
}
```

An animated sprite that has a finite loop will become an empty sprite at the end of the loop or if no animation is playing.

Rafael Oliveira
SWE265



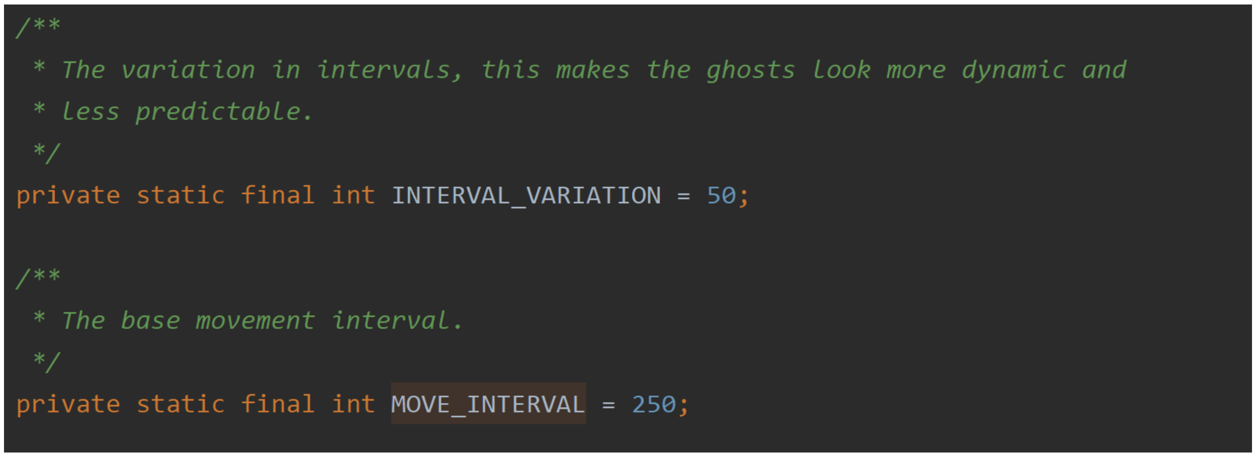
```java
public class AnimatedSprite implements Sprite {

    /**
     * Static empty sprite to serve as the end of a non-looping sprite.
     */
    private static final Sprite END_OF_LOOP = new EmptySprite();
```



```java
/**
 * @return The frame of the current index.
 */
private Sprite currentSprite() {
    Sprite result = END_OF_LOOP;
    if (current < animationFrames.length) {
        result = animationFrames[current];
    }
    assert result != null;
    return result;
}
```

What is the role of MOVE_INTERVAL and INTERVAL_VARIATION?

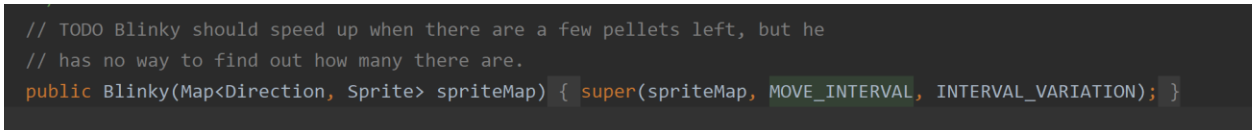
There are four ghosts in this PacMan version: Clyde, Blinky, Inky, and Pinky. They are represented by classes with the same name, and all extend the class Ghost. These ghosts all have their own values for MOVE_INTERVAL and INTERVAL_VARIATION. Blinky's values can be seen in the picture below.



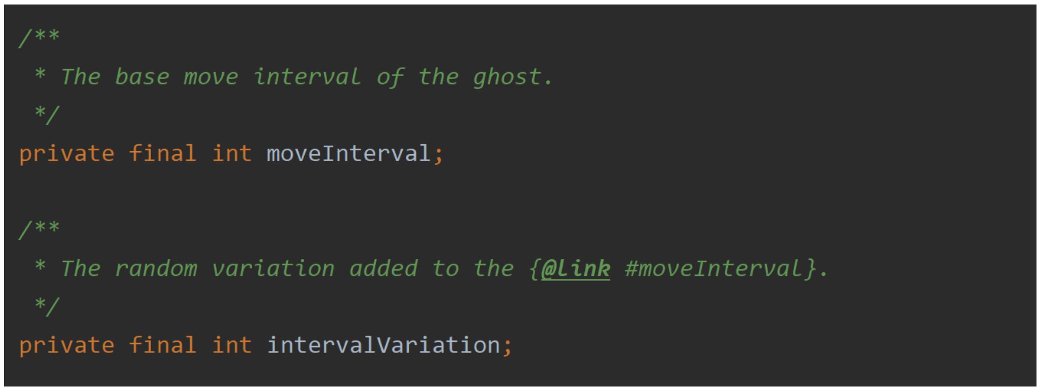
```java
/**
 * The variation in intervals, this makes the ghosts look more dynamic and
 * less predictable.
 */
private static final int INTERVAL_VARIATION = 50;

/**
 * The base movement interval.
 */
private static final int MOVE_INTERVAL = 250;
```

Rafael Oliveira
SWE265

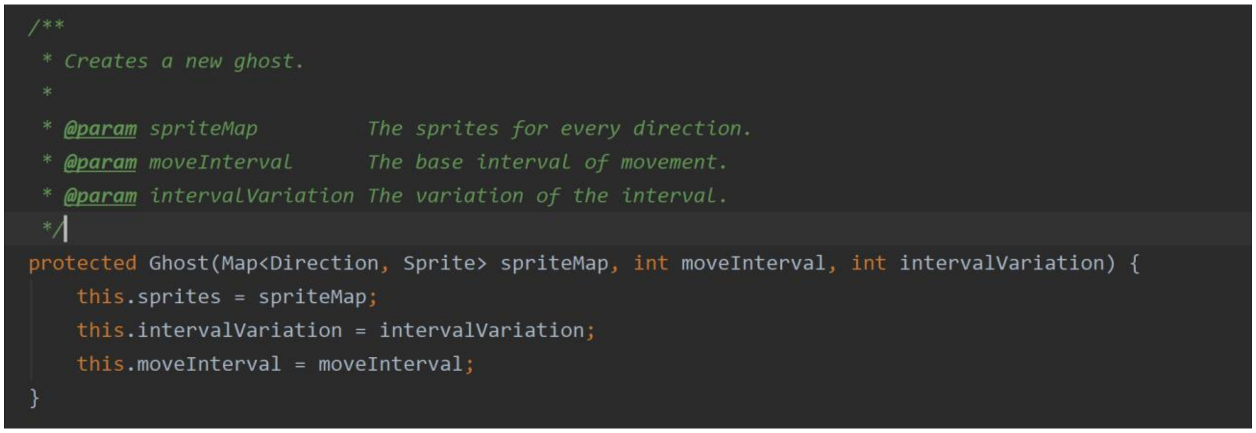
These values are passed into the constructor so movement can be used according to the parent class definition.



```
// TODO Blinky should speed up when there are a few pellets left, but he
// has no way to find out how many there are.
public Blinky(Map<Direction, Sprite> spriteMap) { super(spriteMap, MOVE_INTERVAL, INTERVAL_VARIATION); }
```

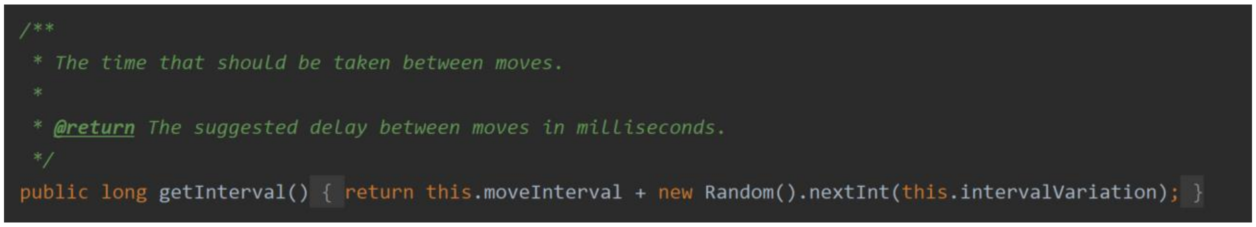


```
/**
 * The base move interval of the ghost.
 */
private final int moveInterval;

/**
 * The random variation added to the {@link #moveInterval}.
 */
private final int intervalVariation;
```



```
/**
 * Creates a new ghost.
 *
 * @param spriteMap         The sprites for every direction.
 * @param moveInterval      The base interval of movement.
 * @param intervalVariation The variation of the interval.
 */
protected Ghost(Map<Direction, Sprite> spriteMap, int moveInterval, int intervalVariation) {
    this.sprites = spriteMap;
    this.intervalVariation = intervalVariation;
    this.moveInterval = moveInterval;
}
```

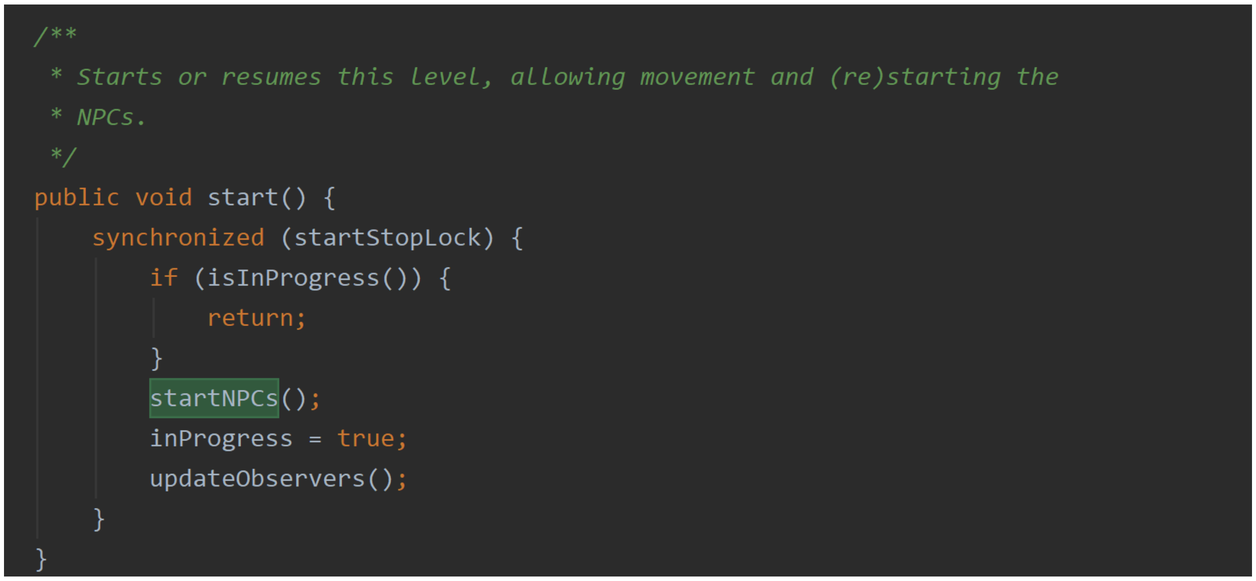
According to the comments, MOVE_INTERVAL is the base interval of movement and INTERVAL_VARIATION is the variation of the interval. This doesn't make the code that clear, since these definitions are not very specific. However, these private attributes are used in the method below.



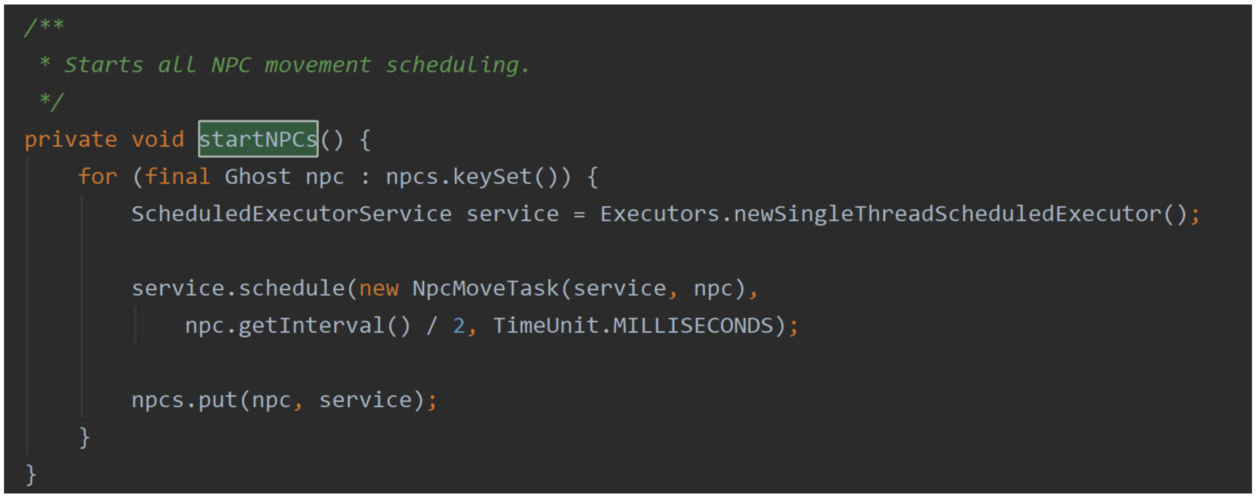
```
/**
 * The time that should be taken between moves.
 *
 * @return The suggested delay between moves in milliseconds.
 */
public long getInterval() { return this.moveInterval + new Random().nextInt(this.intervalVariation); }
```

This method is used in the Level class, where the movement of the NPCs is set up.

Rafael Oliveira
SWE265



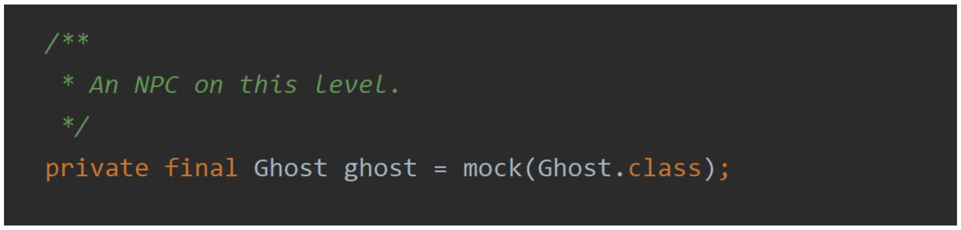
```java
/**
 * Starts or resumes this level, allowing movement and (re)starting the
 * NPCs.
 */
public void start() {
    synchronized (startStopLock) {
        if (isInProgress()) {
            return;
        }
        startNPCs();
        inProgress = true;
        updateObservers();
    }
}
```



```java
/**
 * Starts all NPC movement scheduling.
 */
private void startNPCs() {
    for (final Ghost npc : npcs.keySet()) {
        ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();

        service.schedule(new NpcMoveTask(service, npc),
            npc.getInterval() / 2, TimeUnit.MILLISECONDS);

        npcs.put(npc, service);
    }
}
```

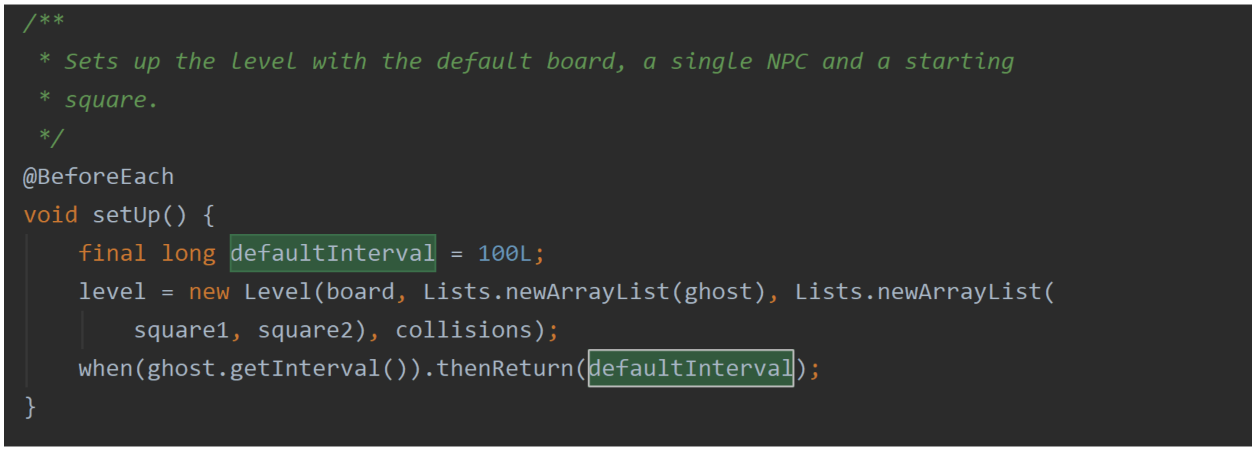
This means that whenever the game starts, for every ghost in the level, their movements are scheduled based on the ghost’s base movement value (MOVE_INTERVAL) and a random value generated using the ghost’s interval variation (INTERVAL_VARIATION).

It is also worth noting that before the game begins, the level is set up with a single NPC (a mocked ghost), just so the game doesn’t look empty.



```java
/**
 * An NPC on this level.
 */
private final Ghost ghost = mock(Ghost.class);
```

Rafael Oliveira
SWE265

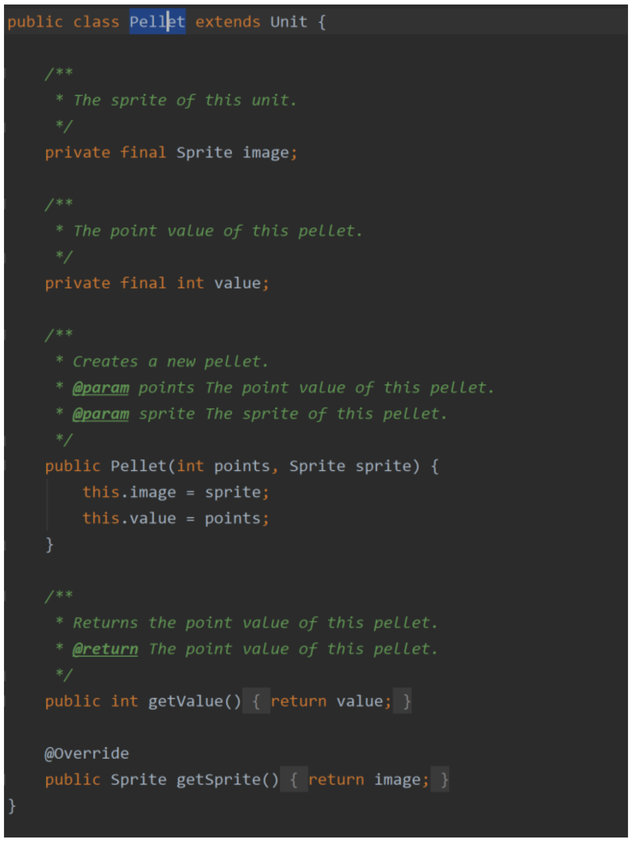


```java
/**
 * Sets up the level with the default board, a single NPC and a starting
 * square.
 */
@BeforeEach
void setUp() {
    final long defaultInterval = 100L;
    level = new Level(board, Lists.newArrayList(ghost), Lists.newArrayList(
        square1, square2), collisions);
    when(ghost.getInterval()).thenReturn(defaultInterval);
}
```

If you wanted to add a fruit, which files would you need to change?

There are many possible answers to this question, since there are currently no fruits in the game, and thus the possibilities are endless. However, the answer that feels the most correct, and is the one I am going to use is the one that follows the current code hierarchy and structure.

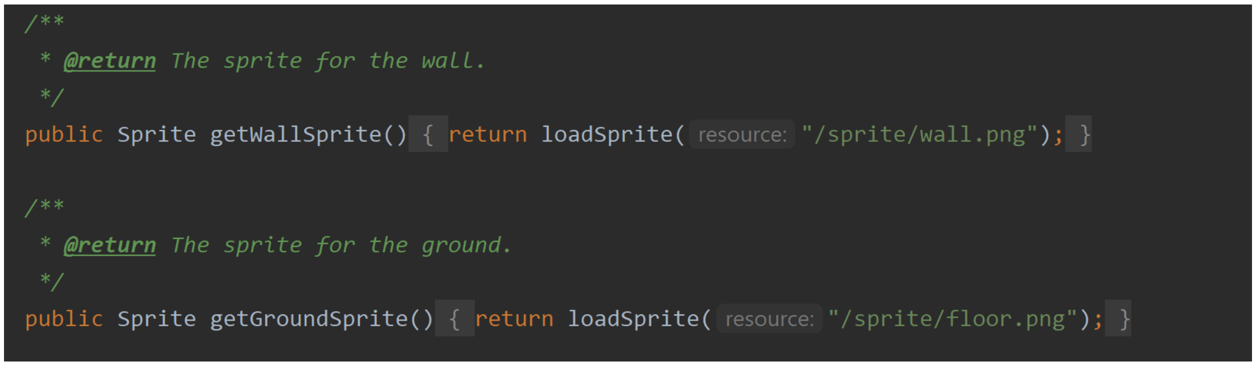
Fruits in the Pacman game are like Pellets, since they are also collectibles that can increase the user score. Therefore, a new class called Fruit should be created. It should be very similar to the Pellet class.



```java
public class Pellet extends Unit {

    /**
     * The sprite of this unit.
     */
    private final Sprite image;

    /**
     * The point value of this pellet.
     */
    private final int value;

    /**
     * Creates a new pellet.
     * @param points The point value of this pellet.
     * @param sprite The sprite of this pellet.
     */
    public Pellet(int points, Sprite sprite) {
        this.image = sprite;
        this.value = points;
    }

    /**
     * Returns the point value of this pellet.
     * @return The point value of this pellet.
     */
    public int getValue() { return value; }

    @Override
    public Sprite getSprite() { return image; }
}
```

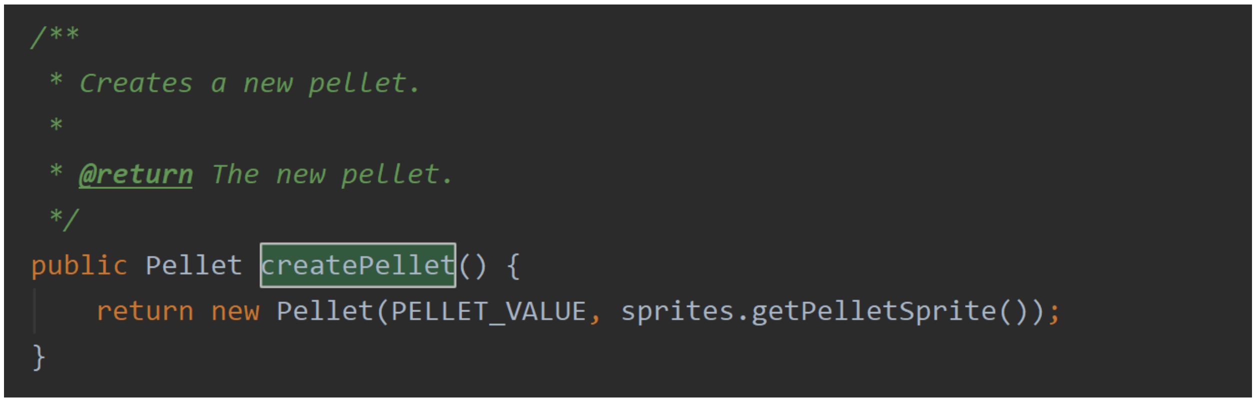
Rafael Oliveira
SWE265

A method to get each fruit sprite (cherry, apple, etc) should be created in the PacManSprites class. Since the image resources for each fruit already exist in the resources folder, then all there is to do is create the methods and link the images.



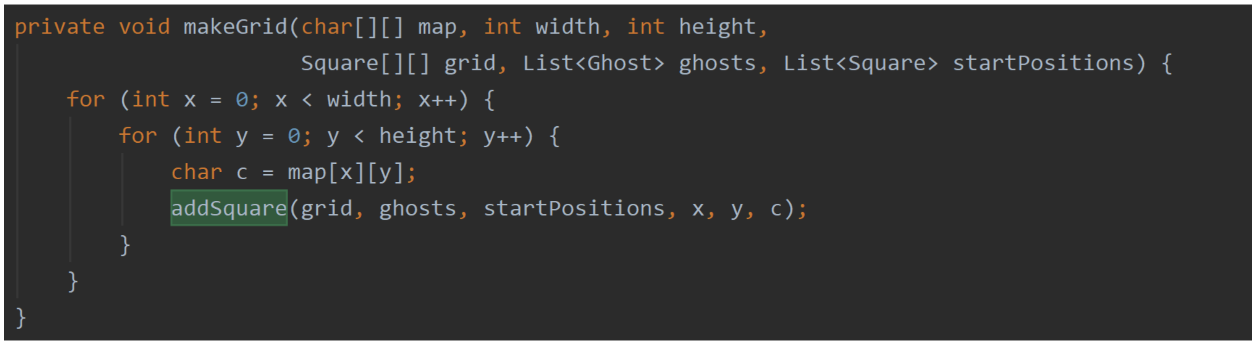
```java
/**
 * @return The sprite for the wall.
 */
public Sprite getWallSprite() { return loadSprite( resource: "/sprite/wall.png"); }

/**
 * @return The sprite for the ground.
 */
public Sprite getGroundSprite() { return loadSprite( resource: "/sprite/floor.png"); }
```

LevelFactory should also be changed. It will now be required to have a method for each fruit created. This method will create a fruit with a fruit value and a fruit sprite. Then it will return that fruit. It is similar to the create pellet value, but instead of being a generic method, it should be specific to each fruit. So apple for example should be created with APPLE_VALUE and sprites.getAppleSprite().



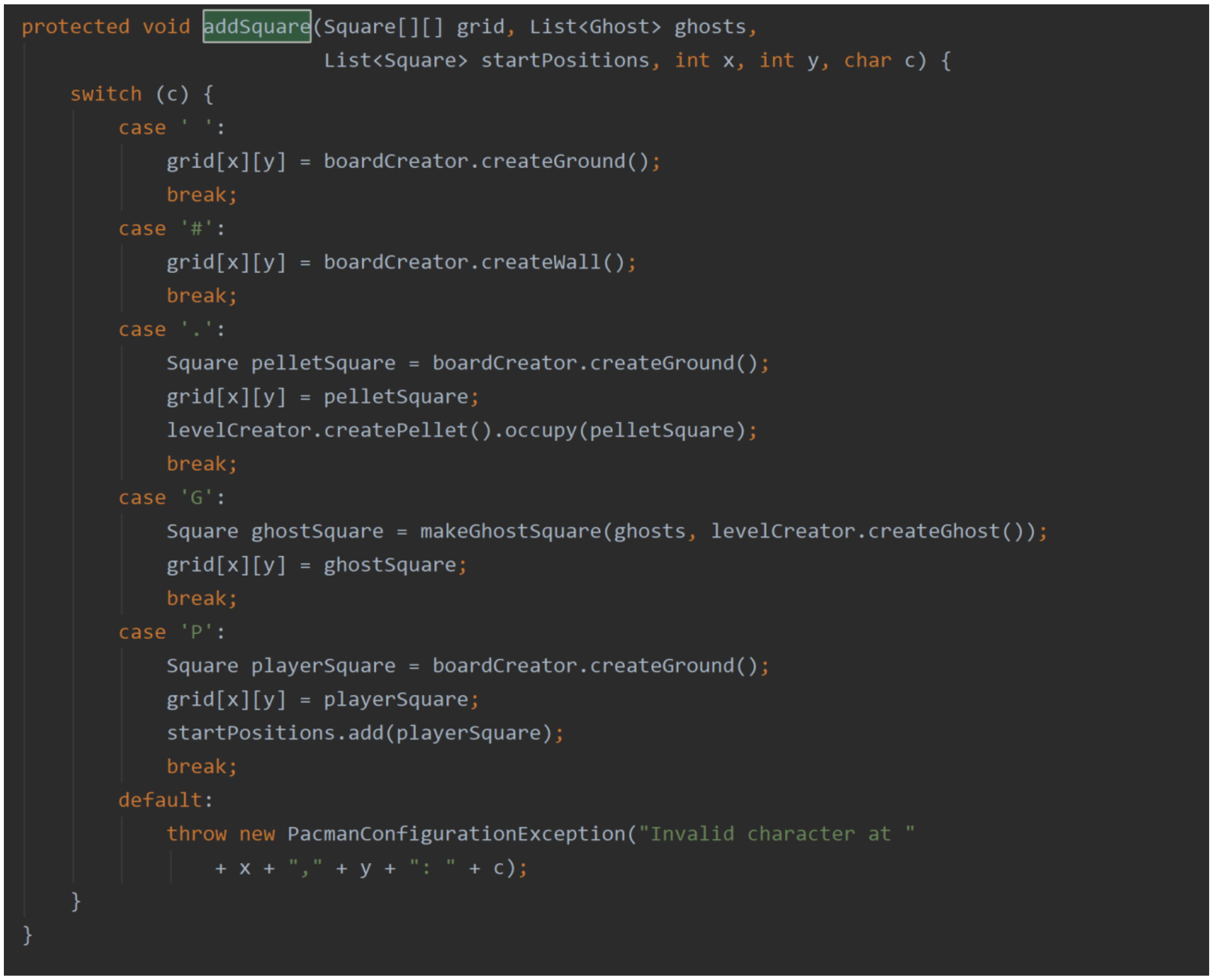
```java
/**
 * Creates a new pellet.
 *
 * @return The new pellet.
 */
public Pellet createPellet() {
    return new Pellet(PELLET_VALUE, sprites.getPelletSprite());
}
```

The final class that needs to be changed is the MapParser. The methods makeGrid and addSquare should be changed so it doesn’t only the already included presets, but also the fruits.



```java
private void makeGrid(char[][] map, int width, int height,
                      Square[][] grid, List<Ghost> ghosts, List<Square> startPositions) {
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            char c = map[x][y];
            addSquare(grid, ghosts, startPositions, x, y, c);
        }
    }
}
```

Rafael Oliveira
SWE265


```java
protected void addSquare(Square[][] grid, List<Ghost> ghosts,
                         List<Square> startPositions, int x, int y, char c) {
    switch (c) {
        case ' ':
            grid[x][y] = boardCreator.createGround();
            break;
        case '#':
            grid[x][y] = boardCreator.createWall();
            break;
        case '.':
            Square pelletSquare = boardCreator.createGround();
            grid[x][y] = pelletSquare;
            levelCreator.createPellet().occupy(pelletSquare);
            break;
        case 'G':
            Square ghostSquare = makeGhostSquare(ghosts, levelCreator.createGhost());
            grid[x][y] = ghostSquare;
            break;
        case 'P':
            Square playerSquare = boardCreator.createGround();
            grid[x][y] = playerSquare;
            startPositions.add(playerSquare);
            break;
        default:
            throw new PacmanConfigurationException("Invalid character at "
                + x + "," + y + ": " + c);
    }
}
```

A specific character should be created for the fruit or the same character for pellet should be used, but then the coordinate could be checked to see if a fruit should spawn there. In the traditional PacMan game, the fruit only spawns after PacMan has eaten a certain number of pellets. If that is the implementation that we are going with, there should be a mechanism to check the number of pellets every time PacMan gets one, and if it is equal to a certain number, a fruit should spawn. This should be changed in the playerVersusPellet method in the PlayerCollisions class.