SWE 265P: Lecture 2 Homework: jpacman3

Harry Duong

Due 22 Jan 2020 at 1500

1. **What is the role of EmptySprite?**

   From the file EmptySprite.java, nothing is supposed to happen when drawing such a sprite.

   My guess (before digging deeper into other source code) is that EmptySprite simply serves as a filler to blank out whatever space it occupies. Maybe an EmptySprite is really invisible such that when Pac-Man touches it, he dies. Or maybe it serves as an abstract base class from which other sprites derive, although the lack of an abstract keyword does not suggest so. These are all just hypotheses. One clue is that it implements a Sprite interface, so that's worth looking at.

   After analyzing any source code containing an EmptySprite, the EmptySprite seems to be a default sprite where it is inappropriate to use another sprite, much in the same way integers (usually) have a default value of 0, or how strings (usually) default to an empty string, pointers to null pointers, and so on.

   Here are some code snippets regarding EmptySprite:

```
// @author Jeroen Roosen
//
//
//
// From ImageSprite.java
@Override
public Sprite split(int x, int y, int width, int height)
{
if (withinImage(x, y) && withinImage(x + width - 1, 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();
}

// From Sprite.java
void draw(Graphics graphics, int x, int y, int width, int height);
```

```
/**
 * Returns a portion of this sprite as a new Sprite.
 *
 * @param x
 *            The x start coordinate.
 * @param y
 *            The y start coordinate.
 * @param width
 *            The width of the target sprite.
 * @param height
 *            The height of the target sprite.
 * @return A new sprite of width x height, or a new {@link EmptySprite} if
 *         the region was not in the current sprite.
 */

// From EmptySprite.java
/**
 * Empty Sprite which does not contain any data. When this sprite is drawn,
 * nothing happens.
 *
 * @author Jeroen Roosen
 */
public class EmptySprite implements Sprite {

@Override
public void draw(Graphics graphics, int x, int y, int width, int height) {
// nothing to draw.
}

@Override
public Sprite split(int x, int y, int width, int height) {
return new EmptySprite();
}

@Override
public int getWidth() {
return 0;
}

@Override
public int getHeight() {
return 0;
}

}

// From AnimatedSprite.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();
}


// From SpriteTest.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);
}
```

2. **What is the role of MOVE_INTERVAL and INTERVAL_VARIATION?**

MOVE_INTERVAL and INTERVAL_VARIATION are static variables in each ghost's class definition. Each ghost is a subclass of the Ghost class. The ghosts' constructurs call their superclass's constructur with MOVE_INTERNAL and INTERVAL_VARIATION as arguments.

```
super(spriteMap, MOVE_INTERVAL, INTERVAL_VARIATION)
```

The call is identical in all four ghosts.

Tracing through Ghost.java, those values become final state variables in the Ghost class. MOVE_INTERVA becomes moveInterval and INTERNAL_VARIATION becomes internalVariation. The comments in Ghost.java suggest that intervalVariation is a deviation from moveInterval such that the Ghost's move interval is the value MOVE_INTERVAL +/- INTERNAL_VARIATION.

The Ghost function getInterval() confirms this; getInterval() returns moveInterval plus a random number based on internalVariation. Here is the Java code for getInterval:

```
/**
     * 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);
}
```

Here is the subclass's call to the superclass's constructor:

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

I searched for all occurrences of image files, most likely JPEG or PNG format. One key file regarding sprite images is PacManSprites.java, a subclass of SpriteStore.java. This fil returns various resources based on *.png files, like an animated or dying Pac-Man. Functions like directionSprite(...) and loadSprite(...) take image files as inputs. loadSprite(...) and loadSpriteFromResource(...), both in SpriteStore.java, appear to not care about the meaning of the image they load; they merely load the images. So I probably won't have to change SpriteStore.java. I may have to change PacManSprites.java because there are various functions like getWallSprite(), getGroundSprite(), and getPelletSprite() that DO account for the image's meaning.

By searching for the keyword 'image', I found another file that I might change - ImageSprite.java. ImageSprite has one occurrence outside of its definition, in SpriteStore.java. ImageSprite is also a generic definition which I am unlikely to change since like the two functions in SpriteStore.java, it doesn't address the sprite's meaning.

Given the variable name 'resource', which comes up several times to hold the image information, I searched for the keyword 'resource'. Unfortunately I did not find anything meaningful; without any more dependencies to find, further things to search are subjective suggestions. The search ends. Maybe another segment of the code is useful - if there is some other sprite that shows up, then maybe there is a function that doesn't necessarily take an explicit image input. I assume that with the way the original author wrote up PacManSprites.java, he/she hardcoded many if not all of the image data. Since the main program doesn't take in any command line arguments, the images' names would have to show up in either the code itself or another in another file that the program reads, and no such file exists, at least when I searched.

The bundled code snippets from PacManSprites.java show functions that specifically name an image file, or in the case of the function getGhostSprite(), an image within a small set of images:

```
/**
 * Sprite Store containing the classic Pac-Man sprites.
 *
 * @author Jeroen Roosen
 */
public class PacManSprites extends SpriteStore {
/**
 * @return A map of animated Pac-Man sprites for all directions.
 */
public Map<Direction, Sprite> getPacmanSprites() {
return directionSprite("/sprite/pacman.png", PACMAN_ANIMATION_FRAMES);
}

/**
 * @return The animation of a dying Pac-Man.
 */
public AnimatedSprite getPacManDeathAnimation() {
String resource = "/sprite/dead.png";

Sprite baseImage = loadSprite(resource);
AnimatedSprite animation = createAnimatedSprite(baseImage, PACMAN_DEATH_FRAMES,
```

```
ANIMATION_DELAY, false);
animation.setAnimating(false);

return animation;
}

/**
 * Returns a map of animated ghost sprites for all directions.
 *
 * @param color
 *            The colour of the ghost.
 * @return The Sprite for the ghost.
 */
public Map<Direction, Sprite> getGhostSprite(GhostColor color) {
assert color != null;

String resource = "/sprite/ghost_" + color.name().toLowerCase()
+ ".png";
return directionSprite(resource, GHOST_ANIMATION_FRAMES);
}

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

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

/**
 * @return The sprite for the
 */
public Sprite getPelletSprite() {
return loadSprite("/sprite/pellet.png");
}
}
```

Overall There are other places that I did not look because I have only so much information. The best I have is to search and hit various parts of the code where image manipulation is obvious and trace dependencies from there. Besides that I cannot think of another obvious method and I will likely learn one in the near future.