Exercise 9

Software Development 2018
Department of Computer Science
University of Copenhagen

Line Hagenow <`liha@di.ku.dk`>,
Kristian Fogh Nissen <`hmf727@alumni.ku.dk`>

Version 1.0: Maj 1st

Due: Maj 11th, **15:00**

Abstract

For this exercise you have both an individual part and a group part. The individual part ***must*** be done alone. For the group part you will be continue working on the *SpaceTaxi Game* while keeping the designed developed in exercise 8 in mind. You must submit your code from the group part on **GitHub** and include the link to the repository in your report. The individual report and the groups technical report must be submitted on Absalon.

Contents

1 Individual Part

You find yourself working in the video game industry. Your current task is to create classes to handle drawing a character on the screen. One big problem though, is that there are a lot of different ways the character can appear! The character starts out short, but when he gets a mushroom, becomes tall. The character may also get a flower, and change into a white and red costume. Finally, regardless of size and costume, if the character gets a star, then he flashes for a short while.

A quick look at the combinations shows that, if you aren't careful, you'll be creating 7 subclasses using inheritance for this game, and with the game designer's love of costumes, this could easily explode in future games. With just 8 different costumes you would have 5041 combinations. Luckily, you notice that each possible character is really just some transform of the base character: either the character gets tall, or changes outfit colors, or flashes.

Your task is to efficiently create a set of classes, using one or more of the 11 design patterns presented in the lectures, which provide the various different image combinations and implement a draw() method. For practical purposes, you do not need to actually draw images or make the game, it is sufficient to output a description of the character to the console. The descriptions should be:

- Base: "It's me, SUrio!"
- Tall: "I'm tall!"
- Flower: "And I can throw fireballs!"
- Star: "Wow! I'm invincible!"

For example, if the character is tall and has the flower, then a call to draw should result in "It's me, SUrio! I'm tall! And I can throw fireballs!" to be printed to the console. Notice how the Base message is still part of the character.

Deliverable: A report presenting the following:

1.1. A discussion of why you chose the design pattern(s) you used and not any of the others.

1.2. A discussion of what advantages and disadvantages there are to using your chosen design pattern.

1.3. A UML class diagram describing the structure and relations of the system you designed.

1.4. Show in the report your implementation code as well as a main method running the example described above.

Be aware that for this assignment the discussions in the report are significantly more important than your implementation.

2 Group Part

This weeks group part will focus on implementing physics and collision detection to your Space Taxi implementation, as well as adding a statemachine to make the game more enjoyable. Remember to test your implementation.

2.1 State Machine

Like in exercise 7, we would like to start the game with a main menu and want to pause the game and so on. In order to do this we need to create a state machine. For this assignment you are welcome to use the state machine you made in `Galaga`, but you are also welcome to create your own one from scratch.

The state machine *must* have the same functionality as in `Galaga`, with the extension of choosing levels.

2.2 Physics

To move the taxi around the maps we need it to be affected by both gravity and it's own thrusters. We will in this section have you implement the physics needed for this to work.

Although it's difficult to imitate actual physics correctly, it is fairly easy to make an approximation that comes close. You might recall that, from Newton's laws of motion, $F = ma$, that is: The net force applied to an object is equal to the mass of the object multiplied with its acceleration. Notice how you need to add together all forces applied to the object for this to work. If we imagine the objects mass to be of 1 unit, then that mass can be neglected in our calculations. You might also recall that an objects acceleration is equal to the derivative of that objects velocity with respect to time: $v' = a$. Through numerical integration we can approximate a velocity of an object to be:

$$v(t_1) \approx a(t_0) \times \Delta t + v(t_0)$$

To sum up: if we apply one or more forces to an object (like thrusters and gravity to the taxi in our game) we can approximate it's velocity at each frame as the addition between the previous velocity and the product of the sum of forces and delta time.

Hint: You can use GameTimer.CapturedUpdates to calculate how many updates happened during the last second. Just be aware that this is 0 when the game starts.

Hint: You can use a 2 dimensional vector to represent a force. Notice how the math namespace of DIKUArcade gives you methods for working with vectors.

More specifically you should implement the following features:

2.1. A source of force that the space taxi is subject to is gravity (The space taxi is really a stratosphere taxi). The bottom of the arena should pull on the taxi, effectively, gradually reducing the velocity of the taxi in its upwards direction.

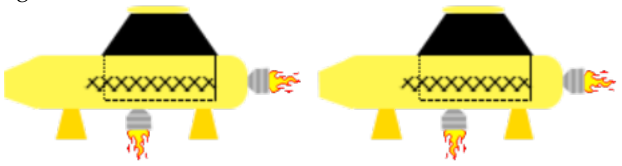
2.2. Two sources of force that a space taxi has are its thrusts. The user can control these thrusts with the arrow keys:

- By pressing UP the bottom thruster is activated and applies an upwards force to the taxi. This force has to be stronger than the constant gravitational force, since the Taxi should be able to go up.
- By pressing one of the side arrows a force will be applied to the taxi from the side. However only one force can be applied from the side at a time (the taxi only has a thruster on the back, not on the front).
- The two previous rules can be combined so that the taxi can accelerate diagonally by holding down both the UP arrow and either of the side arrows.

It is important to note that if you implement this right, the Taxis velocity will rise and fall gradually like in the real world. Remember to design your physics to follow the SOLID principles presented in the lectures.

2.3 Taxi images

You should now have implemented the physics of the taxi. You now have to implement the images of the taxi to match the taxis direction and whether or not it has its thrusters on. In the handout from the last assignment you got 8 different images of the taxi showing the different directions of the taxi with and without the thrusters on. *Notice* that some of these images are imagestrides like the enemies in GalagaGame.
For example if you prees **UP** and **LEFT** the taxi should have following imagestride:



The assignment is now to match these images with the taxis direction. To do this you can use the orientation given in the last assignment.

2.4 Collision detection

At this point you should now have a taxi that moves within the law of physics and whose images match its direction. The last thing we need to do is to make a collision detection, similar to the one you made for GalagaGame, *Hint:* Take a look at the DIKUArcades physics. However this collision detection should do a little more than the one in GalagaGame. You should check for three things.

2.1. Collision with an obstacle, if the taxi touch an obstacle it *must* die (possibly with an explosion) and the running game *must* end.

2.2. Landing on a platform *must* be from above and *must* be with a reasonable speed. If the taxi collides with a platform from any other way than above the taxi *must* die and the running game end. If the taxi touch a platform from above with a velocity that's too high the taxi *must* die and the running game end.

2.3. Collision with the portal must *not* kill the taxi, but should switch the game to a new higher level.

2.5 Technical Report

Write a technical report:
Discuss your implementation of `SpaceTaxi` at the point of submission. Your report should be written LaTeX, and use the su18.sty package.

Your report must at least include:

- Give an overview of the requirements for `SpaceTaxi`.
- Give an overview of your `SpaceTaxi` implementation.
- Discuss your design decisions and how well you followed the SOLID principles.
- Discuss how you designed and implemented your tests.
- Discuss the non-trivial parts of your implementation.
- Disambiguate any ambiguities in the exercise sets.

You must read and follow our guide on writing technical reports. `https://github.com/diku-dk/su18-guides/blob/master/files/techReport.pdf`

You can also look at our example report for the `TicTacToe` program if you need some inspiration. `https://github.com/diku-dk/su18-guides/blob/master/files/su17-tic-tac-toe.pdf`

3 Submission

For the submission you must hand in:

- One *per person*, individual report.
- One *per group*, technical report with the git url in the report.
- One *per group* group.txt file containing your group information.

4 Cleaning up your code

To keep your TA happy, and receive more valuable feedback, you should:

- Remove commented out code.
- Make sure that your files and folders have the correct names.
- Make sure the code compiles without errors and warnings.
- Make the code comprehensible (perform adequate renaming, separate long methods into several methods, add comments where appropriate).
- Make sure the code follows our style guide.