Exercise 8

Software Development 2018
Department of Computer Science
University of Copenhagen

Anders Geil <ange@di.ku.dk>,
Sarah Hyatt <zfj900@alumni.ku.dk>

Version 1.0: April 20th

Due: April 26th, **15:00**

1 Introduction

For the remainder of this course, we will be working on a new game called Space Taxi. This is also the game you will be presenting during the final exams. If you don't already know the original Commodore 64 version from 1984, we encourage you to watch this short *YouTube video* to get an overview of the general game mechanics.

```
%#%#%#%#%#%#%#%#%^^^^^^%#%#%#%#%#%#%#%#%#%
#                 JW       JW              %
%       h2g                                #
#       222                     >          %
%       H2G                        o       #
#        3                           o     %
%        3                                 #
#        3                              o   %
%        3                          j%i     #
#        3                          W Xi    %
%        3                             %    #
#                                      xI    %
%     o                                xI    #
#                                     xI     %
%                                    xI      #
#   o    o                          xI       %
%                                  xI        #
#     o                           xI         %
%       o                        xI          #
#   o                           xI           %
%         o                     I            #
#           n1111111111111m                  %
%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%#%
```

Figure 1: ASCII text map

Figure 2: Game level

This week, we will be implementing functionalities to load some simple ASCII-based text file maps into game levels. At the end of this assignment, you should be able to turn a text file with contents like that of figure 1 into an actual game level like the one in figure 2.

The ASCII text maps can be found in `./SpaceTaxi-1/Levels/` and are structured as follows:

- The top section of the file contains an ASCII drawing of the map layout.
- The middle section of the file contains metadata about the map. In this case, only the name of the map and the number of platforms are provided.
- The bottom section of the file contains a key legend, which maps each ASCII symbol, of the map, to a graphical image.

While we offer you a greater amount of freedom to come up with your own approach to this problem, the following exercises will walk you through some of the considerations you will have to go through, and some of the tools you might find helpful in the process of coming up with a good solution design.

Finally, you may find this exercise vastly more abstract and conceptual than what you may be used to, but we encourage you not to immediately resort to panic-programming as the emphasis during this week's evaluation will be more on the conceptual reflections and arguments conveyed in your report than the concrete implementation in code.

2 Object-Oriented Analysis & Design

At this point everything you do should be possible using just pen and paper – you will start programming later. First, we need to distill the core structure of our solution implementation. To do this, start by performing a use case analysis, similarly to the one presented in the lecture, as follows:

2.1. Reduce the main problem to its core responsibilities. To get a better overview, you may choose to write them down, e.g. in a mind map.

2.2. Some of these responsibilities may seem conceptually related. The next step therefore involves clustering the responsibilities into wider conceptual categories.

2.3. After successfully performing these steps, you should be able to create a responsibility table like the one below (otherwise you may find more concrete inspiration in the lecture slides). It is important to be thorough during this first step, as additions or alterations may potentially ripple through the rest of your design later. Take your time and weigh in the suggestions (and objections!) of your group members.

Responsibility description	Concept name
⋮	⋮
⋮	⋮
⋮	⋮

2.4. So far so good! You may now use this table to deduct the associations between your concepts. That is, how are your concepts related? By explicitly filling in the table below, you can help discipline your own line of thinking to avoid confusion later on. Again, take your time and discuss alternative configurations with your group members. If something is not entirely clear to you at this point, it will most likely only be causing you more trouble later on!

Concept pair	Association description	Association name
⋮	⋮	⋮
⋮	⋮	⋮
⋮	⋮	⋮

2.5. Having identified the main responsibilities and their associations, you may now start to consider what kinds of attributes your concepts require. Again, the table below may prove helpful in formalizing your line of thought and bringing all members of the group into a similar state of mind.

Concept	Attributes	Attribute Description
⋮	⋮	⋮
	⋮	⋮
⋮	⋮	⋮
⋮	⋮	⋮
	⋮	⋮
	⋮	⋮
⋮	⋮	⋮

2.6. You are almost there! If you successfully completed the previous steps, you should now have a list of abstract concepts, as well as their concrete attributes and the associations between them. This should now enable you to visualize a domain model. If you are in doubt as to what this entails, you may want to frequent the lecture slides for a more concrete example.

2.7. Well done! You are now ready to critically assess your model.

Deliverable: All three tables must be included and described in the technical report.

3 Evaluation

Before we proceed to the concrete implementation of our model, it's healthy to take a step back to critically evaluate the quality of our model. After all, it's always more difficult to make changes to the design once it's already been implemented in code!

To determine whether the model is of high quality, you should consider how the model addresses the most basic design principles presented at the lecture. For instance, you may discuss to what extent the model satisfies the following principles:

- The Expert Doer Principle: Do the classes perform tasks in accordance to the information they possess or are some classes responsible for tasks they hold little to no information on?
- The High Cohesion Principle: Are the responsibilities evenly distributed among your classes or do some classes take on a disproportionate amount of the collective workload?
- The Low Coupling Principle: Does most of the information exchange in your model happen within your classes or is there a lot of communication between them?

Further, you may also want to consider the length of the chains of communication in your model. Generally, we strive to keep the chains of communication as short as possible in order to reduce the complexity of our model. Similarly, your model may be particularly sensitive to errors if the network of associations in your model is mostly centered around a few major hubs. Finally, you may find it useful to produce a traceability matrix, like the one shown in figure 3, to provide a convincing overview of how your model satisfies these design principles.

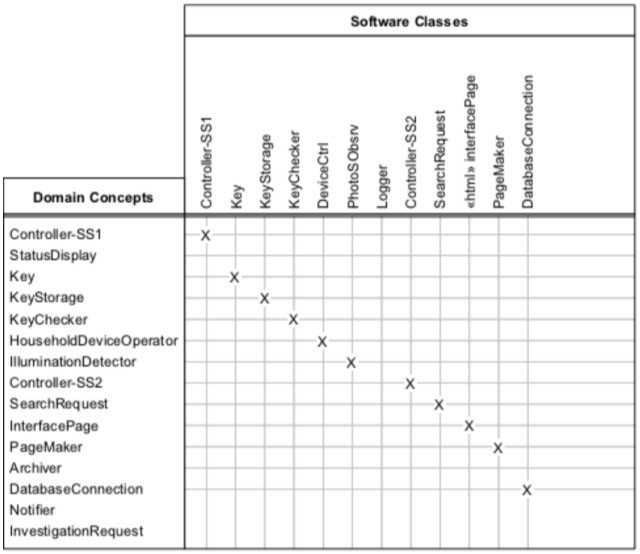

Figure 3: Example of a traceability matrix (from the lecture slides).

In practice, you will probably find yourself negotiating trade-offs between these principles. It is, however, *always* better to be honest and up-front about your specific design decisions. Presenting the arguments and reasoning behind these decisions in a convincing manner is guaranteed to bring you a lot further than neglecting (or even attempting to conceal) the weaknesses of your model.

As you discuss these principles with your group members, make sure to note down your arguments, so you don't forget to include them in the report. Also, do not be afraid to go back and revise your model if you think you can now create a better solution.

In fact, if the evaluation phase goes well, you may well end up with multiple, competing models emphasizing different aspects of the design principles. If this is the case, feel free to (briefly) present and discuss the alternative model, in light of your chosen one, in the report, too.

Once you have decided on a model to implement, we ask you to visualize it using at least one of the different types of UML diagrams. Carefully consider which of the following types best conveys your idea: Class diagram, sequence diagram, and/or state diagram. You might find that the configuration and functionality of your model is best portrayed using a combination of these.

Deliverable: The technical report must present:

3.1. At least one type of UML diagram presenting your design solution.

3.2. An analysis of the strengths and weaknesses identified in your design.

3.3. A discussion showcasing the arguments for using your chosen design (over potential alternatives) in the concrete code implementation.

4 Implementation

4.1 Getting Started

Since the exercise needs to be handed in on Thursday, we decided to offer you a little headstart. Along with this assignment text, you will find a folder, `./SpaceTaxi-1/`, with a basic game environment. If you feel short on time, feel free to simply import this setup to your existing solution as a new project, `SpaceTaxi-1`. Of course, if you already have another setup in mind or would rather reuse some of your features from Galaga, feel free to create your own basic game environment instead. In any case, this part is not central to the evaluation of this week's exercise and will be expanded in later exercises. This week, we will be using it simply as a basic platform from which to deploy our implementation design.

4.2 Implementing your design

In sections 2-3 you created a recipe; now is the time to carry it out. Implement your design in accordance to the specifications provided in your plan. We recommend that you use your UML diagram to implement the classes and methods one by one, making sure that each indvidual component works as intended before proceeding to the next one. This also leads us to the next important point, testing.

4.3 Testing

Testing is a critical component of software development. For the remainder of this course, we require that your handed in code is thoroughly tested. This effectively means you will be providing an additional unit test project at least containing unit tests of your code, although we encourage you to also perform other kinds of testing (e.g. integration testing) as necessary.

Tests will also be a critical component of your final exam, so we urge you to simply write the tests as you go along. It is much easier to write good tests when the code is still fresh in mind.

4.4 Documentation

Finally, we require you to also document your code using the built-in XML documentation going forward. Similar to testing, it's also a good habit to simply write the documentation as you go along rather than saving it for later. If you used the basic code setup in part 4.1, don't forget to show you understand how it works by documenting it too!

4.5 The Expected Outcome

At the end of this implementation we expect you to have:

- A basic game implementation with a simple, controllable space taxi.

- A full implementation of your design, including tables and a UML diagram.
- A game running with two different levels, created by loading the handout ASCII files.
- Thorough testing of your public methods.
- Documentation of all the code used in this assignment.

5 Technical Report

Good news! If you have successfully completed the previous tasks, you are already in possession of the most important ingredients to a great report! All you need to do now is stitch the pieces together in a structured and convincing manner. If you are still unsure of how to construct a good technical report, and you haven't already done so, we highly encourage you to take a look at the overall structure of the example report (available on Absalon) for the previous TicTacToe game.

It may be fruitful to think of the technical report as the pinnacle of your work. Even though the report is chronologically produced last, it is also the part of your hand-in that your recipient is most likely to read first. While you cannot expect the recipient to read and understand the full logic of your code base, you can, however, expect your technical report to be read cover to cover. Your technical report is therefore not only important for reasons of making a good first impression, but also because it provides a unique opportunity to provide a controlled window into the structure and reasoning behind your code. Do not just take this opportunity for granted.

6 Submission - 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.