TTK4145 2024 Code review

Assignment description

The code review process takes place in two phases:

1. You will evaluate the code of several other groups, as well as your own. For each group, you will:
   a. award a score (on a scale from 5 to 9).
   b. give written feedback.

2. You will participate in a code review session with the course staff, where we will:
   a. discuss your code with you.
   b. quality-assure the scores given by other groups.
   c. evaluate the feedback given by other groups.

The code-related portion of your project grade will be based both on your own code, as well as your ability to evaluate others' code. In decreasing order of importance:

- how readable (understandable and easy to navigate) your own code is.
- how mature and useful your comments to the reviewed groups are.
- any hints of maturity and learning in your commentary on your own code.

This document contains information primarily about the first phase of this process.

Code peer review

- **About the code from other groups**

After delivering your code, the files from all groups will be collected, anonymized, and repackaged as a single zipped folder. A separate document will match your group number to several hashes, each of which matches the folders containing other groups' code. You should provide evaluations for each of the hashes assigned to your group.

About the scoring

You will give a score from 5 to 9 to each group you evaluate. This number represents how well you estimate the other group has achieved the code quality learning goals in this course. Assume these coarse categories:

5. The elevator project was too big for this group. The group has hit a wall with regard to maintainability. Their code cannot be fixed without starting over, but the group should retreat to smaller projects before trying.
6. The group failed at keeping their codebase reasonable for this project. It is probably more work to incrementally improve the code than to scrap it and start over. Though if this group *were* to redo it, we believe they would be able to improve its quality.
7. The code is not easily accessible, but we can glimpse the underlying designs from the code. If some days of tidying and aligning were put in, it could be made reasonable.
8. The code is reasonable, though there is some indication that the code quality could deteriorate if the project were to grow in size or scope.
9. This project was manageable for this group. Bugs can be fixed and features added, without any immediate deterioration.

- **About the written feedback**

Aim for about seven bullet points of feedback. Your feedback might be all positive or all negative – be honest and provide feedback that you think would be the most useful.

As you go through the code, take notes of the things that grabbed your attention. If everything is as you expect, it means you can navigate and understand the code effortlessly. This is usually not the case, so anything *unexpected* is often worth commenting on.

As you also will evaluate your own code, you will provide feedback to yourself. This can be very difficult, so set your aim somewhat lower at around two or three bullet points. Try to think of this in the context of your evaluations of the other groups – what have you learned about your own code from reading *their* code?

To streamline our automatic tooling, please submit your feedback in a plaintext file named “group-##.txt,” where ## is your two-digit group ID.

Use the following format EXACTLY as presented here for the content of the file:

```
hash1
score1
bullet point 1
bullet point 2
(or other plaintext)

hash2
score2
feedback for group 2
```

The hash should be on its own line, followed by a number on the next line indicating the score given to that group (a single digit, with no extra comments or decimals). The next lines contain your feedback, and then finally a blank line between the feedback for one group and the hash of the next. The blank lines should only be between the feedback for one group and the hash of the next group.

- **About criteria suggestions**

Appended to this document is a list of suggested criteria you can use when performing your evaluations.

You do not have to use this list. It is provided as an aid in case you have a hard time getting started, come across code that you find particularly hard to evaluate, or perhaps even as a conflict resolution guide in case you have disagreements within your group.

Look at the "main" function or other top-level entry points: ***The thing that "starts" the system***	
1	**Components:** Does the entry point document what components/modules the system consists of? ● You can see what threads/classes are initialized
2	**Dependencies:** Does the entry point document how these components are connected? ● You can see how different components interact and depend on each other ● This would imply making channels, thread IDs, or object pointers here, and explicitly passing them to the relevant components ● If there are any global variables, is their use immediately clear and are their names truly excellent?

3	**Functionality:**
	Do you know where to look to find out how the parts of the system are designed?
	● Examples:
	  ● If it is master-slave or peer-to-peer
	  ● How any acknowledgment procedure works
	  ● How any order assignment works
	  ● How orders for this elevator are executed
	  ● How orders are backed up

Look at the individual modules from the "outside": ***The header file, the public functions, the list of channels or types the process reads from, etc.***	
4	**Coherence:** Does the module appear to deal with only one subject? ● A large interface (lots of functions in a header, lots of channels as parameters, etc.) can be an indication that the module does too many things ● Pay particular attention to the outputs of a module (what it “does”, its “task”, its “role”) ● E.g., the thing that runs a single elevator should probably not perform order assignments
5	**Completeness:** Does the module appear to deal with “everything” concerning that subject? ● There are no cases where an interface shows an obvious lack of functionality ● It is obvious to you how you would use all of it
Look at the individual modules from the "inside": ***The contents/bodies of the functions, select- or receive-statements, etc.***	
6	**State:** Is state maintained in a structured and local way? ● “State” here refers to any data that changes over the life of the program, typically variables ● It is clear “who” is responsible for each piece of state ● The use of shared state is minimized, especially if shared across threads
7	**Functions:** Are functions as pure as possible? ● Functions do not modify variables outside their scope, preferring parameters and return values instead ● If there are any variables with a scope larger than the function, it is trivial to find out what their scope is, and the variables are very easy to keep track of
8	**Understandability:** Is each body of code easy to follow? ● You can see what it does, and you can see that it is correct ● E.g., nesting levels are kept under control, local variables have names that don't confuse you, etc.

Look at the interactions between modules: ***How information flows from one module to the next*** *For example, try to trace an event like a button press, and follow the information from its source (something reading the elevator hardware) to its destination (some other elevator starts moving)*	
9	**Traceability:** Can you trace the flow of information easily? ● A process or object that changes its state has a clear origin point for *why* it changed its state ● Think of debugging scenarios like "Why does this variable have this value now?"
10	**Direction:** Does the information (mostly) flow in one direction, from one module to the next? ● In order to trace an event (like a button press), you don't have to flip back and forth between some modules repeatedly, in order to find its "destination" (like the door opening) ● E.g., if A calls into B, then B does not immediately call back into A again - usually
Look at the details: The contents/bodies of the functions, select- or receive-statements, etc.	
11	**Comments:** Were the comments you found useful? ● The comments were not just a repetition of the code ● Or if there were no comments, you feel that no comments were necessary
12	**Naming:** Did the names of modules, functions, etc. help you navigate the code? ● You were never misled by a vague or incorrect name
Look at the whole:	
13	**Gut feeling:** Give a gut-feeling score from 0 to 10 Do not look at the sum of the points you have given for the other criteria
14	**Feedback:** Provide written feedback to the group that created this code Aim for about seven bullet points