# TTK4145 Peer code review specification

The code review happens in two phases: you will review the code of several of your classmates' (i.e. "peers'") projects, and you will participate in a code review session with the course staff (a.k.a. "final code review") wherein you will receive feedback on your final project code. The peer review phase should result in a report which will be distributed back to your peer groups. This document specifies what the report should contain, as well as some information about the process.

## The process

After delivering your code in the code snapshot hand-in, 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 a folder containing another group's code. You should provide evaluations for each of the hashes assigned to your group.

- Your first task is to download the single folder containing every project and pick out the correct hashes (projects) according to the provided document. Your group number should be associated with four hashes, one of which will correspond to your own group's project code.

The review process starts when you've found the correct project folders. Please ask either a student assistant or the scientific assistant if you are uncertain about how to do choose the correct ones.

- Your second task is to write a peer review report according to the specification given in the next section. You will produce one (1) plaintext document which will contain feedback for all of your assigned groups.

- Your third and final task is to upload your report to Blackboard within the given deadline, as usual.

After the deadline, the reports will be processed by the course staff (i.e. sorted and anonymized), and you will receive the feedback written by your peers. You will then need to consider whether the feedback given to you is actionable and, optionally, implement it in your project before the final hand-in.

The second phase, i.e. the final code review, happens after the final code hand-in and after/during the FAT. We will then have a meeting with you, similar to the post-PDD meetings, where we discuss your code with you as well as do quality assurance on the reviews and scores given to you 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.
- maturity and learning in your commentary on your own code.
- how mature and ***useful*** your comments to the reviewed groups are.

The course staff does not have the capacity to make a detailed analysis of the contents of the peer review, and therefore it will not account for a significant portion of the percentage of your final grade for which the project code hand-in counts. It will primarily be used as a conversation starter in the final code review. However, in accordance with the above bullet points, the peer review *given* may be used to determine your grade in cases of uncertainty or obvious malice. Peer reviews *received* will not inherently affect your grade. In plain: do not worry that a poor review will negatively affect you, but make an effort to provide honest and useful feedback to your classmates.

## The peer review report

### 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.

### 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?

**Formal requirements**

- The report is a plaintext document, i.e. a .txt file
- The file is named **peer-review-##.txt**, where ## is your group number
    - Do not use underscores or otherwise deviate from the format.
    - Single digit groups do 0#, i.e. a zero followed by your group number.
- The contents of the report are structured **exactly** as presented here:

```
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 line(s) should only be between the feedback for one group and the hash of the next group. It is important that you adhere to this pattern, as your report will be fed to an automatic tool for further processing.

**Suggested criteria**

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:***<br>**The thing that "starts" the system** | |
| 1 | **Components:**<br>Does the entry point document what components/modules the system consists of?<br>• You can see what threads/classes are initialized |
| 2 | **Dependencies:**<br>Does the entry point document how these components are connected?<br>• You can see how different components interact and depend on each other<br>  ○ This would imply making channels, thread IDs, or object pointers here, and explicitly passing them to the relevant components<br>• If there are any global variables, is their use immediately clear and are their names truly excellent? |
| 3 | **Functionality:**<br>Do you know where to look to find out how the parts of the system are designed? Ex.:<br>• Whether it is master-slave or peer-to-peer<br>• How any acknowledgement procedure works<br>• How any order assignment works<br>• How orders for this elevator are executed<br>• How orders are backed up |
| ***Look at the individual modules from the "outside":***<br>**The header file, the public functions, the list of channels or types the process reads from, etc.** | |
| 4 | **Coherence:**<br>Does the module appear to deal with only one subject?<br>• 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.<br>• Pay particular attention to the outputs of a module (what it does, its task, its role)<br>• E.g.: The thing that runs a single elevator should probably not perform order assignments |
| 5 | **Completeness:**<br>Does the module appear to deal with everything concerning that subject?<br>• There are no cases where an interface shows an obvious lack of functionality<br>• It is obvious to you how you would use all of it |
| ***Look at the individual modules from the "inside":***<br>**The contents/bodies of the functions, select- or receive-statements, etc.** | |
| 6 | **State:**<br>Is state maintained in a structured and local way?<br>• "State" here refers to any data that changes over the life of the program, typically variables<br>• It is clear "who" is responsible for each piece of state<br>• The use of shared state is minimized, especially if shared across threads |
| 7 | **Functions:**<br>Are functions as pure as possible?<br>• Functions do not modify variables outside their scope, preferring parameters and return values instead<br>• 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:**<br>Is each body of code easy to follow?<br>• You can see what it does, and you can see that it is correct<br>• E.g., nesting levels are kept under control, local variables have names that don't confuse you, etc. |
| ***Look at the interactions between modules:***<br>**How information flows from one module to the next**<br>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:**<br>Can you trace the flow of information easily?<br>• A process or object that changes its state has a clear origin point for why it changed its state<br>• Think of debugging scenarios like “Why does this variable have this value now?” |
| 10 | **Direction:**<br>Does the information (mostly) flow in one direction, from one module to the next?<br>• 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)<br>• E.g., if A calls into B, then B does not immediately call back into A again - usually |
| ***Look at the details:***<br>The contents/bodies of the functions, select- or receive-statements, etc. | |
| 11 | **Comments:**<br>Were the comments you found useful?<br>• The comments were not just a repetition of the code<br>• Or if there were no comments, you feel that no comments were necessary |
| 12 | **Naming:**<br>Did the names of modules, functions, etc. help you navigate the code?<br>• You were never misled by a vague or incorrect name |
| ***Look at the whole*** | |
| 13 | **Gut feeling:**<br>Give a gut-feeling score from 0 to 10<br>Do not look at the sum of the points you have given for the other criteria |
| 14 | **Feedback:**<br>Provide written feedback to the group that created this code<br>Aim for about seven bullet points |