# Assignment: Issue resolution (refactoring)

## 1 Goal

The goal of this project to realize the complexity of issue resolution (or refactoring, but henceforth we will use "issue") in a real project. What is being graded is the **process** through which you carry out your work, the quality of your work, and your reflections on it. Note that **even a partially resolved issue can give you maximal points**, as long as it is well managed and documented.

### Deliverables

You will fork or clone the source code repository of the project of your choice, and perform modifications there. The URL of the cloned repository will be submitted on Canvas. You can use the report template from Canvas. You are free to adapt it or use a different format as long as you cover the same content. The report is uploaded as a file.

## 2 Tasks

### 2.1 Part 0: Project choice

**Choose a project**, or a module of a project, as per the previous assignment (the same criteria apply **but the repo must NOT be a loose collection of algorithms that have no underlying common design or architecture**).[1] Identify an issue you will work on. The issue should be open and have no assignee (or you should get in touch with the assignee to ensure that the work has not been done yet). Please register your issue(s) in the project sheet (see Canvas for the link). Note that each group must work on different issues!

### 2.2 Part 1: Setup

Set up the project on your system(s). How good is the "onboarding" documentation?

- Did you continue with the previous project or choose a new one?
- If you chose a new project: How does the experience compare to the previous one?

Register the task you are working on the spreadsheet (link is on Canvas). Create an account on the issue tracker of the project, and register yourself as an assignee of that task.
**Note:** It is OK if multiple groups choose the same project, but they have to choose different issues/tasks. Maximum is three groups per project (one group per issue!) unless the project is large enough to handle more contributors in a short time.

### 2.3 Part 2: Issue resolution

1. Run the existing regression tests; make sure they succeed. If any tests fail, check if they may interfere with your task. **Keep copies of the test logs.** If you there is a problem with the existing tests that interferes with your task, document that as a new issue (in the issue tracker), and either try to resolve it, or change your task to a different issue, perhaps in a different project.
2. Identify requirements related to the issue. If the requirements are not documented yet, try to describe them based on code reviews and existing test cases. Create a **project plan** for testing these requirements, and working on the issue.[2]
3. Analyze existing test cases that relate to the issue. Do they contain good assertions? Do the assertions cover the requirements as you understand them? If necessary, enhance existing regression tests with properties derived from work done above, or write additional tests. Newly added tests related to refactorings should pass, but if your task is a new feature, new tests should (initially) fail.

[1] If your project fulfills these criteria, you may choose the same project as in the previous assignment, or a different one.
[2] If you choose a task where the goal is to add, update, or refactor tests, add mechanisms that ensure that you have some kind of validation of the test behavior. This could include coverage, test output, or mutations detected by the tests.

4. Make sure your new tests are actually executed. You can use code coverage measurement (especially if this is available in the project you are using), or temporarily inject a fault (mutation) in the code, and see that at least one test fails.

5. Resolve the issue itself.

6. Run the tests on the new code. New tests should pass. Verdicts of old tests on the original and new code should be the same, unless there is a good reason, such as a bug fix that makes a previously failing test now pass.

7. Create a patch in a format that is acceptable to the project.

### 2.4 Part 3: Documentation

Document your experience. Keep track of how your team spends its time.[3] If you were not able to finish all tasks, how much progress have you made? How much time do you think you would need to complete the task? If the process turned out to be less difficult than expected, attempt another issue (not necessarily in the same project).

Assess your **team** (p. 51 in the Essence standard v1.2) by evaluating the checklist on p. 52:
In what state are you in? Why? What are obstacles to reach the next state? How have you improved during the course, and where is more improvement possible?

### 2.5 What to do if there is time left (or time is running out)

If your task is too easy, and you have still enough time left to try another issue, try another one, as written above. If you have more than five hours left and think that, despite your experience gained, cannot find another task that you can do in the remaining time, you should still study other tasks.

For the last issue, when time is running out, create a **work plan.** Depending on the time remaining, you can work on requirements, (missing?) test cases, and other documentation, to give your effort estimate a solid base.

Then, report your outline of the work to be done as a comment in the issue tracker. Even if you thus end up not having enough time to carry out the task (whether it is your primary or secondary task), you will still end up helping others who will work on this later.

## 3 Grading criteria

### 3.1 Pass (P)

1. The onboarding experience of the group is briefly documented.

2. The contribution of each group member is documented. The work carried out is commensurate with the expected time spent (20–25 hours per person).

3. At least some issues are non-trivial in the sense that they do not all affect only single, local functions, but at least multiple functions, units, classes, files, or modules.
   The (planned) changes affect the code and/or functionality of the software, not just documentation or other non-executable artifacts.

4. Requirements related to the functionality are identified and described in a systematic way. Each requirement has a name (ID), title, and description. The description can be one paragraph per requirement.

5. Changes to the code and test suite are shown and documented, e.g., as a patch.

6. Tests are automated, their outcome is documented, e.g., as a test log.

7. Key features affected by the issue are shown in UML class diagrams (for refactorings: include before/after).
   **Note:** you do not have to show classes, fields, or methods that are not relevant, unless they help with the overall understanding. Typically, the diagram would contain 5–10 classes.

[3] For the choice and setup of the project, document which projects and required libraries you are trying to install and run. Do this at a granularity of 30–60 minutes; do not just add five hours for "setup".

8. The overall work carried out, and experience gained, are documented. In particular, mention your experience about the given documentation/examples of the project, its tool framework, and the interaction within your team (using the Essence framework) and with the community of the project.

#### 3.1.1 Regression tests

If all tests pass on the original version of the software:

- The issue is (perhaps partially) implemented and documented.
- Previously existing tests pass on the new version. If the issue is fully resolved, new tests must pass as well.
- There is a patch in an acceptable format.

If some tests related to your issue fail on the original version, and pending bug fixes prevent a full resolution:

- Each bug or issue is documented on the issue tracker of the project.
- Each report is accompanied by at least one automated test case that fails.

#### 3.1.2 Presentation

In the grading session, the group has to

- present the issue at hand,
- show the code changes and documentation,
- demonstrate the work (by running some unit tests or a demo).

### 3.2 Pass with distinction (P+)

#### 3.2.1 4 out of 7 points:

1. The architecture and purpose of the system are presented in an overview of about 1–1.5 pages; consider using a diagram. Note: If you manage to improve on existing documentation or fill a gap in the project here, please consider making your documentation available to the project; they may be grateful for it!
2. Updates in the source are put into context with the overall software architecture and discussed, relating them to design patterns and/or refactoring patterns.
3. Relevant test cases (existing tests and updated/new tests related to the refactored code) are traced to requirements.
4. Your patch is clean in that it (a) removes but does not comment out obsolete code and (b) does not produce extraneous output that is not required for your task (such as debug output) and (c) does not add unnecessary whitespace changes (such as adding or removing empty lines).
5. Patches are accepted by the project, or considered for acceptance. (This requires a link to an accepted commit, or a discussion item.) Note: the patch must be submitted by the assignment deadline, but it may be accepted later. (Please notify us if this extra point is necessary for a P+.)
6. You can argue critically about the benefits, drawbacks, and limitations of your work carried out, in the context of current software engineering practice, such as the SEMAT kernel (covering alphas other than Team/Way of Working).
7. You have done something extraordinary that exceeds the scope of the assignment, and which you can be proud of.

## 4 Ethics

Recall that the student code of conduct forbids any kind of plagiarism, between groups in the course, or based on content taken on the Internet.