SOFTWARE DESIGN AND TESTING LABORATORY

PROJECT DESCRIPTION

For the LDTS course unit's project, you will develop a **text-based game**, in **Java**, taking into consideration the following:

- The game ought to be **text-based** and use lanterna as its GUI framework
  - 🤔 discuss in class with your professor the game you plan to implement.
- You ought to follow *good* object oriented **design practices** (as discussed in class, at the very least).
- The main focus of the project is not its complexity but good usage of **design patterns**.
- Together with the project, you will write a **report** (writting in GitHub's markdown) addressing, amongst others, information on
  - the usage of **design patterns** in your project;
  - which **code smells** are still present at the end, and
  - which **refactorings** could be used to fix them.
- Your **source code** and **report** will be stored at a **GitHub** repository that will be **given to you**.
- Using good **git** practices (like using **feature branches**) is a plus.
- Code needs to be **tested** extensively. Having a good design will help with this. Use **mocks** and **stubs** when needed.

    - 📣 🧑‍💻 In fact, we expect the project to be developed using Test Driven Development. (see more below)

⁉️ **IMPORTANT**: Having a good design will help with some requirements changing mid-project. Be prepared for **some surprises**.:

Groups

The project should be done in groups of **three students** (🔴👀 teams are not allowed to be more than 3 elements!). If the number of students in a class is not a multiple of three, some groups will have only **two** elements.

DELIVERABLES AND DELIVERIES

You should deliver the following two artifacts (both in your GitHub repository):

- **The source code**, which should follow good object oriented design practices, with close attention to the use of appropriate **design patterns**, in the **root** folder of the repository. All functionality should be adequately **unit-tested**. It should be possible to import the repository directly into *IntelliJ* and just run it using **"gradlew run"**. Your final submission must be in the **main** branch.
- **A report** written in markdown syntax, following the provided template, in a *README.md* file on the **/docs/** folder. It should describe the features that are **implemented**, the features that are **planned**, the **design patterns** that have been used, the existing **code smells** and what **refactoring techniques** could be used to fix them. It should also reflect the **quality** of the **unit tests** with **screenshots** and **links** to the **coverage** and **mutation** testing reports.
- You should also have an extra **simplified** version of the *README.md* file in your **root** folder with a simple description of the game, and some screenshots.

These two artifacts (code and report) should be continuously available in the project's git repository and will be evaluated in two distinct points in time:

- **Intermediate delivery** @ 2022-01-08.
- **Final delivery**
    - Code for Demo @ 2022-01-17 to 2022-01-22 week (ready before your last practical class).

- Final Code & Report @ 2021-01-29.

In both of these deliveries the **code** and the **report** should reflect your implementation up to that point in time. The features reported as implemented should be properly **tested** and **bug-free**. Existing **unit tests** should all pass. The use of **design patterns** should be properly **motivated** and **described** using UML.

More on Test Driven Development

The following sequence is based on the book *Test-Driven Development by Example*: (2)

- 1. **Add** a test

  The adding of a new feature begins by writing a test that passes iff the feature's specifications are met. The developer can discover these specifications by asking about use cases and user stories. A key benefit of test-driven development is that it makes the developer focus on requirements *before* writing code. This is in contrast with the usual practice, where unit tests are only written *after* code.
- 2. **Run** all tests. The new test *should fail* for expected reasons

  This shows that new code is actually needed for the desired feature. It validates that the test harness is working correctly. It rules out the possibility that the new test is flawed and will always pass.
- 3. **Write** the simplest code that passes the new test

  Inelegant or hard code is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.
- 4. All **tests** should now pass

  If any fail, the new code must be revised until they pass. This ensures the new code meets the test requirements and does not break existing features.
- 5. **Refactor** as needed, using tests after each refactor to ensure that functionality is preserved

Code is refactored for readability and maintainability. In particular, hard-coded test data should be removed. Running the test suite after each refactor helps ensure that no existing functionality is broken.Examples of refactoring:moving code to where it most logically belongsremoving duplicate codemaking names self-documenting splitting methods into smaller piecesre-arranging inheritance hierarchies

Repeat

The cycle above is repeated for each new piece of functionality. Tests should be small and incremental, and commits made often. That way, if new code fails some tests, the programmer can simply undo or revert rather than debug excessively. When using external libraries, it is important not to write tests that are so small as to effectively test merely the library itself,(4) unless there is some reason to believe that the library is buggy or not feature-rich enough to serve all the needs of the software under development.

TDD and Git

You should consider creating a new branch for each new feature you plan to develop. It should also be clear for the commits that tests were developed first, and only then the solution.