Software Engineering, University of Oulu

**Lab 4: Unit Testing**

*Disclosure: The material in this paper has not been reviewed, endorsed, or approved of by the Rust Foundation. For more information on the Rust Foundation Trademark Policy, click* [*here*](https://docs.google.com/document/d/1ErZlwz9bbSI43dNo-rgQdkovm2h5ycuW220mWSOAuok/edit) *.*

1. **Introduction**

In software development, it is important that the product works as expected after any new changes to the source code. Any patch or feature needs to go through appropriate testing to determine if it should be accepted to the main branch and/or end build.

Unit testing is an automatic software testing method where individual units of source code are tested. The primary goal is to validate that each unit of code performs as expected, adhering to its intended functionality.

**2. Unit Testing in Rust**

Unit testing is a built-in functionality in Rust. It is common in smaller projects to put unit tests into a test module in the same file as the source code.

A picture containing text, screenshot, font

Description automatically generated

More in-depth details for unit testing in Rust:

<https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html>

After adding unit tests to the project, they can be run with:

cargo test

**2.1. Test Coverage**

In larger projects, it is important to keep track of how much of the code is being tested. Some organizations may require that each push needs to keep the coverage over a certain percentage.

In Rust, we can generate coverage reports with cargo-tarpaulin.

cargo install cargo-tarpaulin

You can run the coverage check with:

cargo tarpaulin

We can also generate a coverage report file by typing:

cargo tarpaulin –-out html

You can simply open this file in your browser and examine line-by-line in each file which parts are covered by tests.

**3. Exercise**

1. Create a GitHub classroom environment: <link here>.
2. Create unit tests for lib.rs.
3. Generate a tarpaulin-report.html and make sure that you reached over 90% test coverage.
4. Complete the CI/CD-pipeline to run unit tests and check for coverage with the given script.
5. Commit your changes and push them to the created remote repository.
6. Return a link to your repository in Moodle.

Make sure that you understand what the pre-made script does!