CPEN 321 Software Engineering
Winter 2025

M4: (Requirements, Design, Implementation), Testing, Code Review (Friday, November 7, 10:59pm)

The focus of this milestone is testing (your own code) and code review (your own and third-party code). The work on the milestone consists of three steps, which will be performed both in groups and then in communities. **Please read the milestone specification to the end before starting to work on the assignment!**

You should aim to complete Steps 1 and 2 of the assignment by Monday, November 3. **In any case, please make sure to meet and exchange artifacts with your community peer-group on or before that date, to leave both groups enough time to analyze each other's code and tests and write the final report** (Step 3).

You are permitted to use an assistive AI technology while working on the assignment. However, if you use such technology in any stage of working on the assignment, you must properly document and critically analyze its use (Step 4). No points will be deducted for any type of documented use. Undocumented use will be considered **academic misconduct** and will be treated accordingly. Please consult the course syllabus for more details.

STEP 1. Your group will perform:

1. Automated testing of your back-end using Jest (https://jestjs.io) – a testing and code coverage framework for TypeScript.
   - Back-end testing will focus on all externally triggered back-end APIs (typically, APIs exposed by the back-end to the front-end, but also back-end APIs called by a third-party service on a certain schedule, etc.).
   - For each exposed interface, create two "*describe*" groups (https://jestjs.io/docs/api).
     - The first group will contain tests for which no mocking is necessary, i.e., where the external components can be triggered and will perform as expected.
     - The second group will contain tests for which you need to use mocks (https://jestjs.io/docs/en/mock-functions) to simulate behaviors of external components that you cannot fully control, e.g., errors in databases and APIs of external components.
     - Annotate each group with the name of the tested interface and whether the tests in the group rely on mocking or not.
     - Furthermore, annotate each test with information about its inputs, expected returned status code, outputs, expected behavior, and (where applicable) information about the behavior of the mock, as in the examples below.
     - **Note:** You may find it beneficial to place all tests with mocks and tests without mocks in separate folders, so you can easily run each subset independently.
   - Make sure to configure the tests to report coverage information for individual files in your back-end (see *collectCoverageFrom* configuration option at https://jestjs.io/docs/configuration#collectcoveragefrom-array).

Example: Tests for POST /photo API, no mocks

```
// Interface POST /photo
describe("Unmocked: POST /photo", () => {
  // Input: test_photo.png is a valid photo
  // Expected status code: 201
  // Expected behavior: photo is added to the database
  // Expected output: id of the uploaded photo
  test("Valid Photo", async () => {
    const photo = fs.readFileSync("test/res/test_photo.png");
    const res = await app.post("/photo")
      .attach("photo", photo);
    expect(res.status).toStrictEqual(201);
    expect(typeof res.body).toBe("number"); // Expect returned id
    const insertedPhoto = database.getAllPhotos().find(photo =>
photo.id === res.body);
    expect(insertedPhoto).toBeDefined();
    expect(insertedPhoto.content).toStrictEqual(photo);
  });
```

```
  // Input: no photo attached to request
  // Expected status code: 400
  // Expected behavior: database is unchanged
  // Expected output: None
  test("No Photo", async () => {
    // ...
  });

  // Input: bad_photo.txt is a not a valid photo
  // Expected status code: 400
  // Expected behavior: database is unchanged
  // Expected output: None
  test("Invalid Photo", async () => {
    // ...
  });

  // more tests...
});
```

Example: Tests for POST /photo API, with mocks

```
// Interface POST /photo
describe("Mocked: POST /photo", () => {
  // Mocked behavior: database.uploadPhoto throws an error
  // Input: test_photo.png is a valid photo
  // Expected status code: 500
  // Expected behavior: the error was handled gracefully
  // Expected output: None
  test("Database throws", async () => {
    jest.spyOn(database, 'uploadPhoto').mockImplementation(() => {
      throw new Error('Forced error');
    });
    const photo = fs.readFileSync("test/res/test_photo.png");
    let res;
    expect(async () => {
      res = await app.post("/photo")
        .attach("photo", photo);
    }).toNotThrow();
```

```
    expect(res.status).toStrictEqual(500);
    expect(database.uploadPhoto).toHaveBeenCalledTimes(1);
  });

  // more tests...
});
```

2. Automated end-to-end testing of your project using Jetpack Compose Testing APIs (https://developer.android.com/develop/ui/compose/testing) and UI Automator framework (https://developer.android.com/training/testing/other-components/ui-automator) for testing Compose screens and cross-app UI actions (like opening the settings menu or app launcher), respectively.
   - End-to-end testing focuses on frond-end GUI tests for **three** main app features.
   - You can create one test case for all use cases of each tested feature combined or create a test case for each use case separately. In any case, please make sure to include all success and failure scenarios from the formal use case specification (as stated in the Requirements_and_Design.md document).

Example: Assuming the formal use case specification below for the "Add Todo Items" use case:

Main Success Scenario

1. The user opens "Add Todo Items" screen.
2. The app shows an input text field and an "Add" button. The Add button is disabled.
3. The user inputs a new item for the list. The add button is enabled.
4. The user presses the add button.
5. The screen refreshes and the new item is at the bottom of the todo list.

Failure scenarios

3a. The user inputs an ill-formatted string.
 3a1. The app displays an error message prompting the user for the expected format.
5a. The list exceeds the maximum todo-list size.
 5a1. The app displays an error message to inform the user.
5b. The list is not updated due to recurrent network problems.
 5b1. The app displays an error message prompting the user to try again later.

A test case specification would be:

Scenario steps	**Test case steps**
1. The user opens "Add Todo Items" screen.	Open "Add Todo Items" screen
2. The app shows an input text field and an "Add button". The add button is disabled.	Check text field is present on screen Check button labelled "Add" is present on screen Check button labelled "Add" is disabled
3a. The user inputs an ill-formatted string.	Input "*^*^^OQ#$" in text field
3a1. The app displays an error message prompting the user for the expected format.	Check dialog is opened with text: "Please use only alphanumeric characters"
3. The user inputs a new item for the list. The add button is enabled.	Input "buy milk" in text field Check button labelled "Add" is enabled
4. The user presses the "Add" button.	Click button labelled "Add"
5. The screen refreshes and the new item is at the bottom of the todo list.	Check text box with text "buy milk" is present on screen Input "buy chocolate" in text field Click button labelled "Add" Check two text boxes on the screen with "buy milk" on top and "buy chocolate" at the bottom
5a. The list exceeds the maximum todo-list size.	Repeat steps 3 to 5 ten times Check dialog is opened with text: "You have too many items, try completing one first"

Note that for 5b, one would need to mock back-end responses, which is not required for this milestone, so you can omit this part.

3. Automated testing of two non-functional requirements of your project.

4. Automated code review of both front-end and back-end of your project using Codacy (https://www.codacy.com/) – an automated code review tool.
    - To install Codacy, please follow the instructions in Piazza post @113.
    - Run Codacy in the main branch of your project, on your front-end and back-end code and tests. Fix the reported issues.

5. Automated continuous integration pipeline using GitHub Actions (https://docs.github.com/en/actions) – an automated CI/CD platform that allows you to automate your build, test, and deployment pipeline.
    - Configure the tool to run your back-end tests every time code is pushed to the **main** branch of your Git repository.

6. Updated project scope, requirements, and design document (the "Requirements_and_Design.md" file and its corresponding PDF version stored in Git).
    - Make sure to update the "Change History" section of the document for each modification, to list the date the modification was made, the modified section(s), and the rationale for the modification. Please make sure the rationale is clear and reasonable.
    - If no changes were made, state "None". However, **mismatches** between your requirements/design and your implementation/tests will cause mark deductions.

STEP 2. After implementing the steps above, prepare a report in the markdown format. The report should be named "Testing_And_Code_Review" and be pushed into your Git repository, in the "documentation" folder. A readable/submittable version of the markdown file in PDF format should be pushed into Git as well.

A template of the markdown file is available at: https://people.ece.ubc.ca/mjulia/teaching/CPEN321-W25T1/Testing_And_Code_Review.md

Use it as a starting point. It is your responsibility to ensure that the document that you produce is well-structured (all sections and subsections are clearly identified), clear, and readable.

The report should include the following high-level sections:

1. Change history
2. Back-end test specification: APIs
3. Back-end test specification: Tests of non-functional requirements
4. Front-end test specification
5. Automated code review results

1. Change history

The change history part will currently be empty as this is the first version. As you refine the document for the final milestone, you will need to document the change date, the modifications made, and the rationale for the modifications.

2. Back-end test specification: APIs

2.1. Locations of your back-end test and instructions to run them. Specifically, please provide:

2.1.1 A table listing, for each API, the location of each "describe" group (without and with mocks) and what components were mocked to test the API in the mocked "describe" group. See the example below.

Interface	**Describe Group Location, No Mocks**	**Describe Group Location, With Mocks**	**Mocked Components**
POST /photo	https://github.com/xx/backend/tests/unmocked/photoNM.test.js#L3	https://github.com/xx/backend/tests/mocked/photoM.test.js#L5	Photos DB
GET /photo	…	…	Photos DB
…	…	…	…

2.1.2. The hash of the commit on the **main** branch where your tests run.
2.1.3. Explanations on how to run the tests.
- The reviewer will run the tests on this version. Some tests may still fail at this point, but they will need to pass by the final release.
- If no clear explanations are provided and the reviewer cannot run the tests, you will lose marks.

2.2. The location of the .yml files that run all your back-end tests in GitHub Actions.
- The reviewer will use your configuration files to run all tests on the latest commit in the main branch. If there are tests that are not triggered from the configuration file, continuous integration automation will be considered incomplete.

2.3. Screenshots of Jest coverage reports for all files in your back-end (individual and combined), when running back-end tests **without mocking**.
- We expect to see high coverage for each back-end file, but less than 100% coverage due to missing error cases.

2.4. Screenshots of Jest coverage reports for all files in your back-end (individual and combined), when running back-end tests **with mocking**.
- Here, the coverage can be lower, as you mostly focus on error handling.

2.5. Screenshots of Jest coverage reports for all files in your back-end (individual and combined), when running both back-end tests **with and without mocking**.
- Here, we expect to see high coverage
- If the coverage is lower than 100%, provide a well-formed reason for not achieving 100% coverage.

Note: Some back-end tests may still fail at this point, but they will need to pass by the final release.

3. Back-end test specification: Tests of non-functional requirements
3.1. Provide the location of tests of the two non-functional requirements that you verify automatically.
3.2. For each test, specify, in one paragraph, how you verified that the requirement was met and provide logs of your verification result.

4. Front-end test specification
4.1. The location of your front-end test suite.
4.2. For each test, list the use case the test is verifying, the test's expected behaviors (as in the example table given in STEP 1, item 2), and the execution logs for the automated test runs (with passed/failed status).
Note: Some front-end tests may still fail at this point, but they will need to pass by the final release.

5. Automated code review results
5.1. The hash of the commit on the main **branch** where Codacy ran.
5.2.The number of unfixed issues per Codacy categories. For this, please take a screenshot or copy the "Issues breakdown" table in the "Overview" page from Codacy:
https://app.codacy.com/gh/<github_username>/<github_repo_name>/dashboard
5.3. The number of unfixed issues per Codacy code patterns. For this, please take a screenshot or copy the "Issues" page from Codacy:
https://app.codacy.com/gh/<github_username>/<github_repo_name>/issues/current
5.4. For each unfixed issue, provide a justification for why it was not fixed
- We expect to either see 0 issues left or have every issue that is left thoroughly justified, with citations to reputable sources. Opinion-based justifications (e.g., an opinion on Stack Overflow, without proper citations or acknowledgement from Codacy developers themselves) will not be accepted.

STEP 3. You will review front-end and back-end code, tests, and the Testing_And_Code_Review report of your peer-team. After the review, you will prepare a report named "M4_<YourPeerGroupName>_Review.pdf", which will constitute your review of your peer group work. The report should include the sections below and your assessment of the quality of work for each item in sections 1-3, on the 0-10 scale:

1. Manual code review
   a. Code is maintainable (well-documented, uses self-explanatory variable names, does not have complicated and long methods, magic numbers, etc.), efficient, handles error cases well, etc.: x/10
2. Manual test review
   a. Tests are complete (all APIs exposed to the frontend are tested, three main features are tested), errors and edge cases are thoroughly tested, correct assertions are used: x/10
   b. Test code is maintainable and well-structured: x/10
   c. Test implementation matches the requirements and design: x/10
   d. Non-functional requirements are tested well: x/10
   e. Tests achieve high coverage and cases of $< 100\%$ coverage are well-justified: x/10
   f. All back-end tests can be run automatically: x/10
3. Automated code review
   a. Codacy runs with the required setup: x/10
   b. All remaining Codacy issues are well-justified: x/10
4. Fault: report one major issue you found in your peer-team app. The report should contain the details about the issue (with screenshots) and the severity of the issue
   - Issues that can be easily found with automated tools, like Codacy, ChatGPT, etc. are not considered major. Look for deep "semantic" issues that require human intelligence.
   - Yes, you will find some major issues – there is no app without issues a few weeks before the final deadline.
   - If you report that you cannot find any major fault and then a TA does, you will lose marks. Otherwise, you will be given the full mark.

STEP 4. Prepare your reflections and peer evaluation documents (same as in M3).

4-a: Prepare a PDF file named "M4_Reflections.pdf", with the following information.

1. Fill out "Table 1 – Task Distribution" below:

ID	**Task (a short description)**	**Team Member**	**Duration**
1.			*3 hours*
2.			*1 hour*
	…		

2. Fill out "Table 2 – Time and AI Reliance Distribution" below:

	Time Spent on the Assignment (hours)	**AI Reliance for the assignment (0 – 100%) (Estimate)**
Team Member 1 Name		
Team Member 2 Name		
Team Member 3 Name		
Team Member 4 Name		
Group Overall		

3. Your reflections on the use of AI:

→ If you, as a group, did not use any AI tools:
3.1 Why did you decide not to use AI?
3.2 Provide 2-3 concrete examples of AI tools being inadequate for tasks in this milestone.

→ If you did use AI tools:
3.1 Pick 4-5 most major tasks from the table above and specify:
- 3.1.1 What was your task? (a longer description)
- 3.1.2 Which AI Interfaces, Tools, and Models have you used for this task?
- 3.1.3 What was your strategy for utilizing the tool for this task?
- 3.1.4 What are the advantages of using AI tools (and particular models, if relevant) for this task?
- 3.1.5 What are the disadvantages of using AI tools (and particular models, if relevant) for this task?

3.2 Anything else you would like to share about this process.

4-b: Prepare a Zip file named "M4_AI_Interactions.zip", with PDFs of all your group's conversations with AI tools. Each PDF file name should contain the name of the group member involved in the conversation.

4-c: For your **iPeer** evaluation mark, please log into the iPeer website (https://ipeer.elearning.ubc.ca/login) using your CWL username and password and follow the "M4 Evaluation" instructions there.

The evaluation period will end 24 hours after you submit your assignment. *Please note that you will not get any marks for the assignment if you do not evaluate your peers and each day of delay in the evaluation (counted as full integers) will reduce your individual assignment mark by 20%.*

SUBMISSION. The submission for this milestone will include three parts.

PART I: A PDF file named "Testing_And_Code_Review.pdf", which includes the information about **your** project (from Step 2).

PART II: A PDF file named "M4_<YourPeerGroupName>_Review.pdf", which includes your assessment of **your peer-team** work (from Step 3).

PART III (from Step 4):
- A PDF file named "**M4_Reflections.pdf**".
- A zip file named "**M4_AI_Interactions.zip**".
- (No Canvas submission is needed for iPeer evaluation; the submission is done via iPeer).

Good luck!