CSCI 2134 Assignment 4

Fall 2024. Due date: 11:59pm, Friday, November 29, 2024, submitted via Git

Objectives

Extend an existing code-base and perform some basic class-level refactoring in the process.

Preparation:

Clone the Assignment 4 repository

https://git.cs.dal.ca/courses/2024-fall/csci-2134/assignment4/????.git

where ???? is your CSID.

Problem Statement

Take an existing codebase, add required features, test your features, and

Background

The CacheSim application is ready to move on to version 2! Now that all the bugs are fixed, it's time to add some new features, as requested by the client. Your boss has hired you to extend the code. She will provide you with (i) the "bug free" codebase, (ii) the existing code specification, and (iii) the requirements of the additions to be made.

Your job is to (i) create a design for the additions, (ii) implement the additions, and (iii) create unit tests for the additions, while following good development practices. This includes: proper git usage, writing comments/documentation, writing unit tests, and doing regression testing. This assignment is meant to emulate how you might work and implement features in the real world.

Note1:
Although these classes and code are very similar to Assignment 3, this is a standalone assignment. You do not need to "import" your unit tests or bug fixes from Assignment 3. Everything you need is included directly in the Assignment 4 repo. Simply use the unit tests and existing code provided.

Note2:
It's a good learning exercise to compare the bug fixes you completed in Assignment 3 to the code provided in Assignment 4. Look to see if you missed any bugs. Compare the way you fixed the bugs versus the way the bugs are fixed in the Assignment 4 code. Good questions for self reflection are: Are the two fixes for the same bug different? Which fix do you like better? Why?

Tasks

0. **Follow these tasks in sequence** to achieve the best results!

1. Review the old specification (**`specification.pdf`**) in the **`docs`** directory of the repo. You will absolutely need to understand it and the code you are extending.

2. Review the **new requirements** at the end of this document, which describes all the extensions to be performed.

3. Implement, document, and test **Feature #1**.
    a. Start by creating a new git branch `feature1` on which to implement the feature. Commit as often as needed on this branch.
    b. Implement the requirements as specified below. This includes creating a new `BackingStore` class and `BackingStoreResponse` class.
    c. Create a `BackingStoreTest` class in the test directory and move the unit tests for `fetchData()` and `pushData()` into the new test class.
    d. Modify the unit tests of `fetchData()` to also check that the correct timing is produced. (You'll have to manually compute and then hard-code the expected timings)
    e. Create a `BackingStoreResponseTest` class in the test directory and implement sufficient unit tests to test your new `BackingStoreResponse` class.
    f. Add additional unit tests to `CacheResponseTest` to test the new methods added to `CacheResponse`.
    g. **Do regression testing.** Ensure all unit tests pass. Ensure all input test cases pass. Note that the `GoldX.txt` files now include the correct timing as well.
    h. Commit and push the feature branch!

4. Once Feature #1 is implemented and tested, merge it into the main branch.

5. Implement, document, and test **Feature #2**.
    a. Start by creating a new git branch `feature2` on which to implement the feature. Ensure that you are branching from `main`, and not from `feature1`. Commit as often as needed on this new branch.
    b. Implement the requirements as specified below. This includes creating a new class `DirectMappedCache`.
    c. Create a `DirectMappedCacheTest` class in the test directory and implement new unit tests. Each method to be tested requires several test cases. Use the test cases from `CacheTest` as a template. You should implement test cases for:
        i. `findData()`
        ii. `evictData()`
        iii. `getContents()`

        For brevity, you do not need to write unit tests for other methods.

    d. **Do regression testing.** Ensure all unit tests pass. Ensure all input test cases pass. For `TestX.txt` as input, simulation with a direct-mapped cache should produce `GoldX-DM.txt` and simulation with a least-recently used cache should produce `GoldX.txt`.
    e. Commit and push the feature branch.

6. Once Feature #2 is implemented and tested, merge it into the main branch.

7. Provide a readable, professional looking UML diagram of your final implementation. This includes both pre-existing classes and any classes you added to implement the extensions.
    a. You **do not** need to include protected methods, private methods, or getter/setter methods in the UML diagram.
    b. There are a lot of online tools available to draw UML diagrams. Creately, draw.io, canva, miro, and lucidchart are all great options.
    c. Commit your UML diagram as **`design.pdf`** in the **`docs`** directory of the repository. Make sure you're committing on to the main branch.

8. **Commit and push back** everything to the remote repository.
    a. **Note:** you really should be committing frequently throughout this assignment. Recall best practices for debugging, refactoring, and software development generally: make many small commits!

Submission

All extensions and files should be committed and pushed back to the remote Git repository.

Grading

The following grading scheme will be used:

Task	4/4	3/4	2/4	1/4	0/4
Git Usage (10%)	The same as (3/4) PLUS: each feature branch contains multiple commits showing implementation and testing process.	Each feature is implemented on its own branch. Branches are tested before merging. Branches are merged into main.	Branches are not adequately tested before merging.	Fewer than two branches are used.	No branches are used.
UML (10%)	Diagram accurately reflects the implementation. All necessary relationships and class attributes are correctly included. Diagram is neat and professional.	Diagram accurately reflects the implementation. Most relationships and class attributes are correctly included.	Diagram mostly reflects the implementation. Most relationships and class attributes are correctly included. Diagram is not very well organized or legible.	UML class diagram generally reflects the implemented design. Only some relationships and class attributes are correctly included.	No UML diagram provided.
Code Clarity (5%)	Code looks professional and follows style guidelines	Code looks good and mostly follows style guidelines	Code occasionally follows style guidelines	Code does not follow style guidelines	Code is illegible or not provided
Documentation (10%)	Code is thoroughly documented, including ID blocks for new classes and documentation for new methods.	Code is mostly documented. A few parameters, returns, or ID blocks are not sufficiently documented.	Code is partly documented. Roughly half of the expected details are missing.	Code is barely documented. Most ID blocks are missing. Most methods are not documented.	No code documentation.

- **Feature #1 Implementation (20%):**
    1. `BackingStore` implemented correctly (5%)
    2. `BackingStoreResponse` implemented correctly (5%)
    3. `CacheResponse` extended as required (5%)
    4. `CacheSim` and `CacheSimMain` extended as required (5%)
- **Feature #1 Testing (15%):**
    1. `BackingStore` unit tests (4%)
    2. `BackingStoreResponse` unit tests (4%)
    3. `CacheResponse` unit tests (4%)
    4. All unit tests and input test cases pass (3%)
- **Feature #2 Implementation (15%):**
    1. `DirectMappedCache` class created and method overridden (5%)
    2. `DirectMappedCache` behaviour is correct; input test cases all pass (5%)
    3. `CacheSimMain` extended as required (5%)
- **Feature #2 Testing (15%):**
    1. Unit tests are implemented for each method (3% for each method under test)
    2. New unit tests pass (2% for each method under test)

Specification of Required Extensions

Feature #1: Timing the cache simulation.

The client wants additional metrics from the cache simulation beyond number of misses. They want to account for the time taken to access the cache and the backing store during a sequence of requests. Use a float for the timing. We will assume the following:

a) Accessing the cache using `requestData()` or `writeData()` requires at least 5.0 seconds.
b) Calling `fetchData()` within `requestData()` or `writeData()` incurs additional time based on how many lines it reads from the backing store to find the requested data. Assume each call to `readLine()` requires 1.0 seconds.
c) `pushData()` incurs no additional time. (i.e. writing to the backing store occurs asynchronously and does not count toward the simulation time).

Before implementing this feature, we notice that the `Cache` class itself is currently responsible for accessing the backing store to read and write data. This does not follow the Single Responsibility Principle. We will begin by first encapsulating interaction with the backing store and then extending its functionality to include timing.

You can follow this procedure to implement the feature:

1. Create a `BackingStore` class whose constructor takes a `String` as argument. This `String` is the file name of the underlying file representing the backing store. The `Cache` constructor should then create and store a new `BackingStore` object rather than storing the file name.
2. Refactor `fetchData()` and `pushData()` from `Cache` into the `BackingStore` class. `Cache` should then call those methods from `BackingStore` to fetch and push data as needed.
3. Extend `fetchData()` to return the fetched data and the amount of time required to fetch that data.
    a. Create a `BackingStoreResponse` class which encapsulates the retrieved data and the time to access the backing store. This class should have getters and setters for its encapsulated data members.
4. Extend `CacheResponse` to include a new field (i.e. instance variable) for the time.
    a. Add a new constructor which receives a third parameter for time.
    b. Provide getters and setters for the new time field.
5. Add a method to `CacheSim` to compute the total time for the simulation. This should be similar to `getCacheMisses()`.
6. Extend `CacheSimMain` to get and print out the total time for the simulation to its output file. The total time should be printed on its own line after the cache misses line. It should be formatted as "`Total Time: X`", where `X` is the time returned from `CacheSim`.

Feature #2: Direct-Mapped Cache

The client wants an additional kind of cache beyond the current "least recently used" cache. In a direct-mapped cache, a cache item is always installed into a specific position within the cache (whether or not the cache is full). Therefore, rank is no longer used to determine eviction. Every cache item can only be installed in exactly one position (i.e. index).

Where the cache item is installed depends on its key: **the index to install the cache item is the key modulo the cache capacity**. For example, in a cache with capacity 4:

- key 2 would be installed in index 2 (2 % 4 = 2).
- key 18 would be installed in index 2 (18 % 4 = 2).
- key 23 would be installed in index 3 (23 % 4 = 3).

`CacheSimMain` must also be updated now to ask the user for which type of cache they would like to simulate. Ask the user which cache they would like to simulate and receive their choice using `System.in`**.** Users should input "L" for the pre-existing least-recently used cache or "D" for the direct-mapped cache. The program should continue to ask for the user's choice until a valid choice is given. (*Hint:* see the mailbox example in the defensive programming slides.)

You can follow this procedure to implement the feature:

1. Create a subclass of `Cache` named `DirectMappedCache`.
2. In `DirectMappedCache`, override `findEvictCandidate()`, `evictData()`, and `installData()` to achieve the desired behaviour. These are the only three methods you need to modify to obtain the correct behaviour.
3. Extend `CacheSimMain` as specified to choose between the basic `Cache` or the new `DirectMappedCache`. `CacheSimMain` then constructs the desired cache type and performs the simulation as before. Notice that `CacheSim` does not need to change to handle the new type of cache. Polymorphism will handle it.

Non-Functional Requirements

1. All methods and classes should be thoroughly documented.
2. Code should consistently follow good coding practices and style guidelines.