CS 3307A Object-Oriented Design & Analysis

Full Fidelity Prototype

The Dollar Tracker

Western University

Dr. Umair Rehman

Kareem Rahme	Ahmad Omran
251219015	251220813
krahme@uwo.ca	aomran3@uwo.ca

Table of Contents

1. Introduction

A primary focus of *The Dollar Tracker* system has been to build a user-friendly, comprehensive tool for small businesses to manage their employees' roles, develop schedules, track hours worked by staff (clock-in/clock-out), and create payrolls, including calculating overtime. This system was designed and built using C++, along with the Qt framework, and the objective of the system is to provide small businesses with an alternative for employee scheduling, tracking time, and producing payrolls promptly.

The major focus of the project was to incorporate object-oriented design (OOD) into the application of design, modularity into the architecture, and adherence to best practices of software engineering.

In addition to demonstrating the progression from the project proposal (Deliverable 1) and partially implemented system (Deliverable 2) to a completely implemented, operational system (this document). This document will detail the progression of integrating multiple design patterns, improved error checking mechanisms, a layering of the architecture (User Interface, Business Logic Layer, Data Access Layer), and an integrated automated test suite utilizing GoogleTest and GoogleMock to verify that the system is maintainable, extendible, and reliable. This document also provides an overview of the system's architecture, a justification of the design patterns employed, updated UML diagrams, a comprehensive testing report detailing both unit-tested code and mock interaction between classes, and a final section of personal reflections regarding the difficulties, decisions, and lessons learned during the development of the system. Ultimately, this project represents a strong demonstration of practical software engineering skills and the ability to design, implement, and deploy a complete, realistic application.

2. Architecture Overview

System Modules Overview

These are the three primary system modules described in the code. They are the building blocks that are integrated together to make up the overall system.

---

1. UI Module (Qt Dialogs)

Location: src/ui/

This module contains all of the user-visible windows:

- **LoginWindow** – Allows users to authenticate themselves as either an employee or manager.
- **EmployeeWindow** – Contains clock-in/clock-out functionality, allows viewing pay stubs, and scheduling.
- **ManagerWindow** – Manages employee information, creates new shifts, previews/exports payroll, assigns roles, and configures the system.
- A number of auxiliary dialogs (e.g., Payview, Schedule, ViewPayStub, ViewWindow).

The UI layer communicates with the core logic solely through well-defined public interfaces (APIs), therefore performs no calculations, validation, or file operations internally. The separation of concerns here provides the core logic with both independence from the Qt UI and allows for full Google Test coverage of the core logic.

---

2. Core Logic Module

Location: src/core/

This module defines the business logic of the system:

- **Payroll** – Calculates regular hours, overtime, flat rate pay, and weekly payroll totals.
- **Validation** – Validates inputs for employee IDs, shift start/end times, etc.
- **Paths** – Determines the correct path for runtime-generated data.
- **FileGateway** – Provides abstracted file I/O.

The Core Logic Module is responsible for:

- Parsing the clock log
- Calculating payroll
- Interpreting roles
- Formatting files
- Detecting edge cases (invalid shifts)

It has been designed to have no dependencies on the UI; thus, it may be completely tested using Google Test.

---

3. Data Access Module (Gateway Pattern)

The FileGateway class is the system's single entry point into persistence (files):

- ids.txt
- mg_ids.txt
- emp_clock.txt
- jobs.txt
- schedule.txt
- PayStubs.txt

By making the FileGateway an abstraction of the filesystem:

- The Core Logic does not interact with file streams directly.

- The back end of the storage mechanism may be changed (i.e., a SQL database) without having to modify the business logic.
- Google Mock can replace the gateway during testing, allowing for interaction testing.

Data Flow

There are four phases of data flow in the system:

1. Login Phase

- LoginWindow → FileGateway → mg_ids.txt / ids.txt → Select Role → Launch Window

2. Employee Workflow

- Clock In/Out → FileGateway.writeAll(emp_clock.txt)
- View Pay Stub → Payroll.compute() → Payview

3. Manager Workflow

- ManagerWindow

- Employee Management (ids.txt)
- Role Management (jobs.txt)
- Scheduling (schedule.txt)
- Payroll Preview (Payroll.computeWeekly)
- Export Payroll (PayStubs.txt)

4. Payroll Computation Flow

- ids.txt → Roles + Rates
- emp_clock.txt → Extract time stamps → Calculate Hours

- Payroll.compute() → Generate Payroll::Line objects
- Payview → Display weekly summary

The data flow shows how there is a clear separation of responsibility in the system:

- UI → Core Logic → Gateway → Filesystem.

---

3. Design Patterns

Two creational patterns and one structural/behavioral pattern were implemented in the system to support modularity, maintainability, and testability.

3.1 Creational Patterns

Pattern 1: Singleton (FileGateway)

Where Used:

- FileGateway::instance()

Why Chosen:

- The application needs a single source of truth for writing/reading files. Multiple instances could cause race conditions, overwrite files, or result in inconsistent file paths.

Benefits:

- Ensures all parts of the application use the same file I/O handler.
- Allows for the replacement of the gateway with a mock during testing.
- Prevents accidental creation of duplicate or conflicting file operations.

Pattern 2: Factory Method (Windows and Dialog Construction)

Where Used:

- All Windows/dialog construction follows a factory method as part of their implementation in Qt, such as:

- LoginWindow login;
- ManagerWindow w;
- EmployeeWindow w(id);

Why Chosen:

- Simplifies the creation of UI components.
- Keeps the initialization of the components contained within the component itself.
- Supports future extensibility (for example, creating different types of ManagerWindow).

Benefits:

- Reduces coupling between modules.
- Ensures consistent construction of UI components.
- Improves testability by keeping the creation of each window isolated.

3.2 Structural/ Behavioral Patterns

Pattern 3: Gateway/Facade (FileGateway)

- FileGateway is also a facade for the system, simplifying the interface for accessing multiple file operations:
  - readAll()
  - writeAll()

- Instead of having multiple places in the system where file I/O is done, all requests are routed through the FileGateway.

Why Chosen:

- Eliminates redundant file-I/O logic.
- Makes error handling easier.
- Enables testing: the gateway may be replaced with a mock using Google Mock.

Behavioral Outcome:

- When using EXPECT_CALL in Google Mock, we verify the behavior:
    - EXPECT_CALL(mock, writeAll(path, data)).Times(1);
- Thus, FileGateway also uses a command-based behavioral verification model.

How These Patterns Have Improved the System

Pattern	Improvement
Singleton	Allowed for centralized file I/O, preventing an inconsistent state, and made mocking possible.
Factory Method	Improved UI construction to be cleaner and more modular, reduced coupling.
Facade/Gateway	Unified all data access, made payroll/schedule logic simpler, and improved testability.

4. UML Diagrams

4.1 Use Case Diagram 1

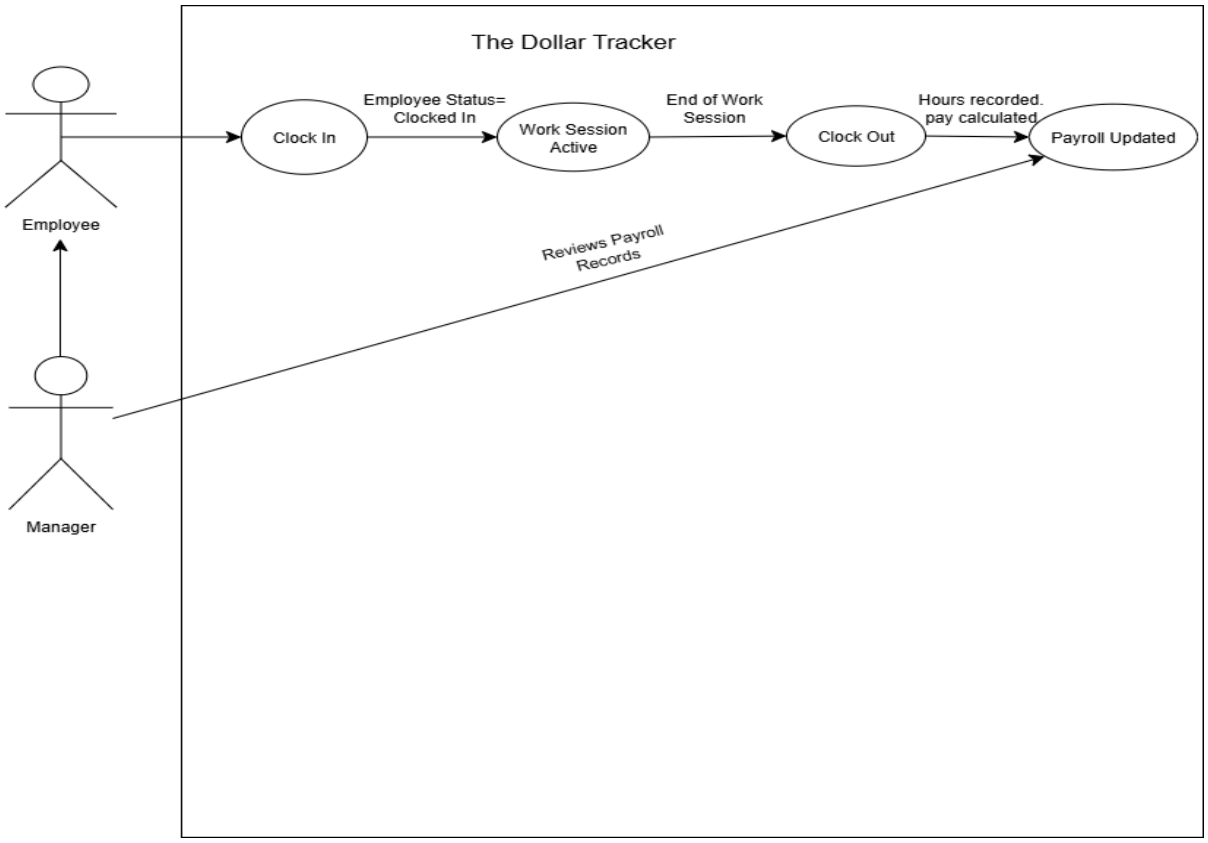


Figure 1: Use Case Diagram 1

Table 1.1

Use Case	Employee Work Session
Primary Actor	Employee
Secondary Actor	Manager
Goal in Context	Allow an employee to start their workday, record their attendance accurately, and update payroll records at the end of the session.
Preconditions	Employee exists in the system, is not already clocked in, and the system clock and data storage are available.
Trigger	Employee presses ‘Clock In’ as the start of the shift

Scenario	1) Employee presses **Clock In**. 2) System validates ID and ensures no active session. 3) System records start time in the time log. 4) Employee works while the session remains open. 5) Employee presses **Clock Out** at the end of the shift. 6) System records end time, calculates total hours, and updates attendance. 7) Payroll module computes pay for that shift and generates a pay stub entry. 8) The manager may review payroll records as part of oversight.
Exception	- Employee already clocked in → show error, block action. - Invalid ID → reject. - System cannot write to log → abort with warning. - Employee forgets to clock out → flagged for correction by manager.
Priority	High

4.2 Use Case Diagram 2

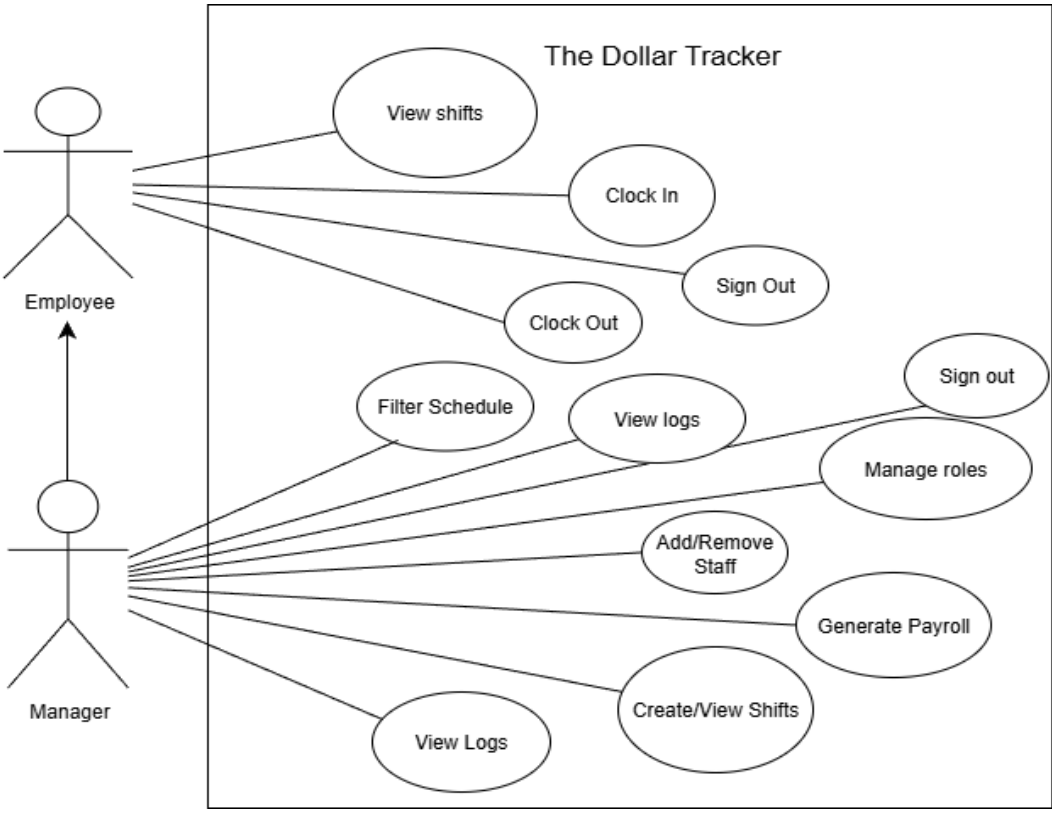


Figure 2: Use Case Diagram 2

4.3 Sequence Diagram

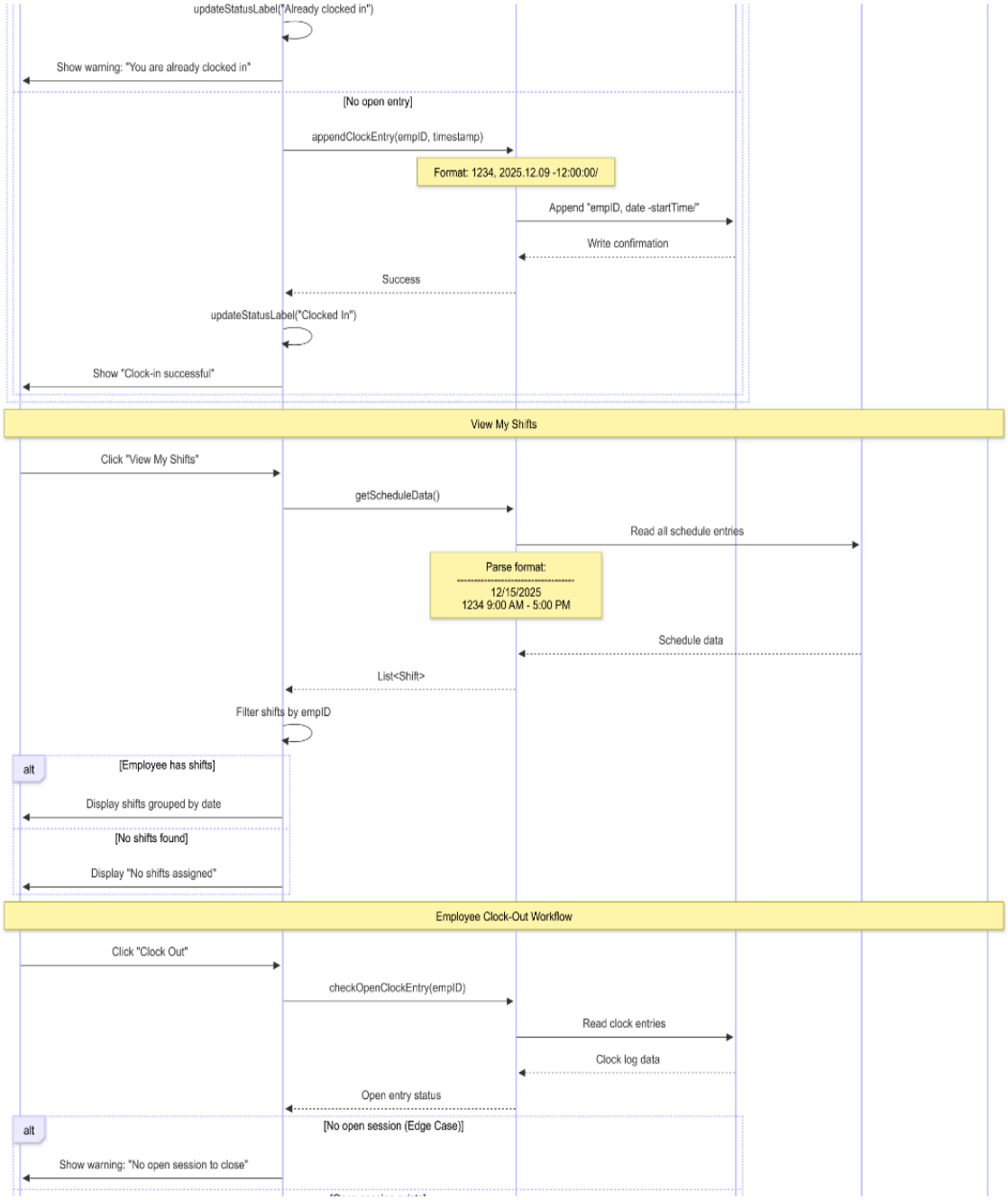


Figure 3: Sequence Diagram

4.4 Class Diagram

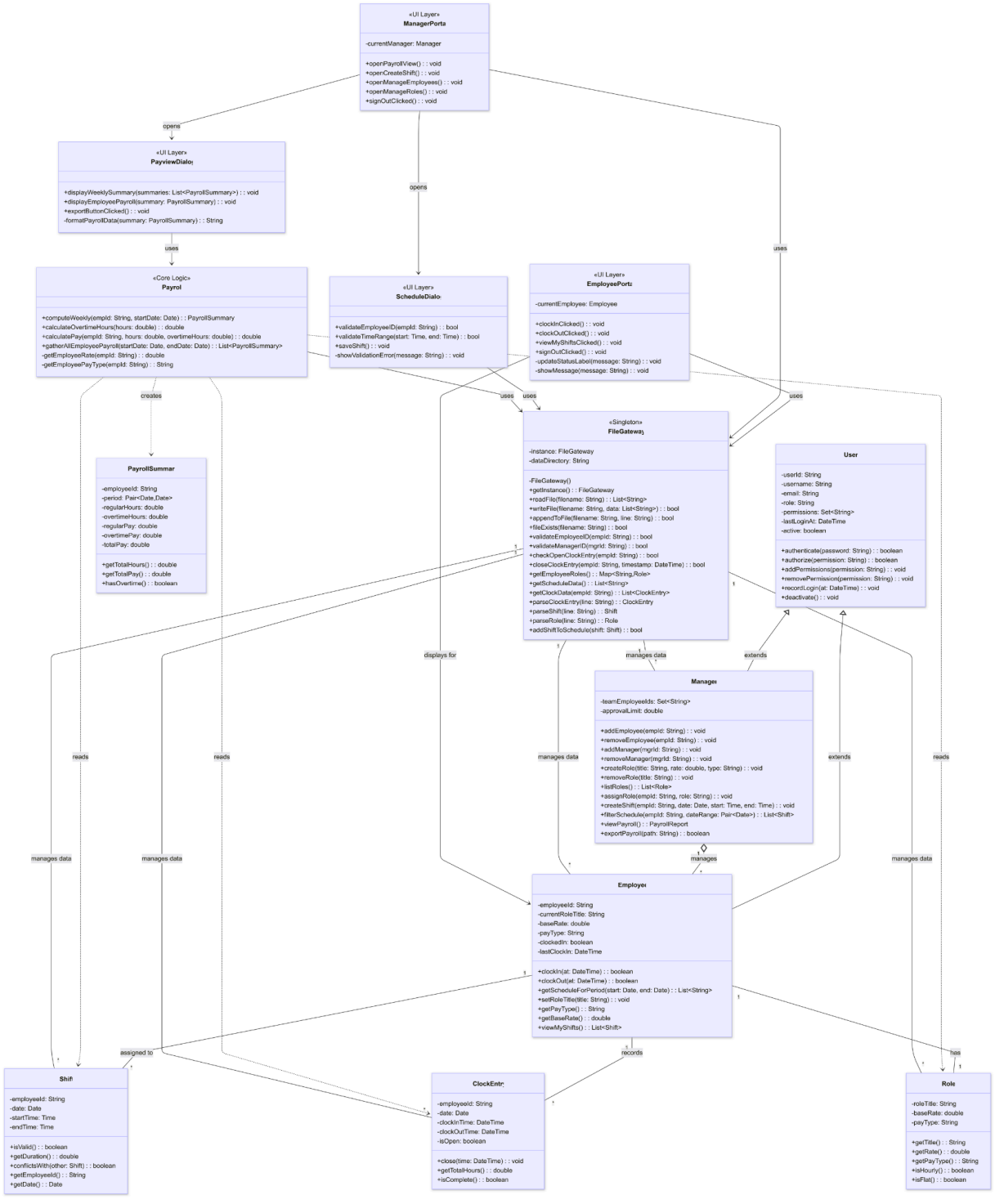


Figure 4: Class Diagram

5. Testing Report

5.1 Testing Strategy

For Deliverable 3, automated unit testing was the centrepiece, which was used to confirm the correctness, robustness, and maintainability of *The Dollar Tracker*. The tests were performed on the business logic that was the most critical, which was also independent of the Qt UI layer to allow for deterministic and repeatable tests.

GoogleTest and **Google Mock** were used for the implementation of all tests, and CMake was used for integrating them into a separate executable target called *DollarTrackerTests.* The tests were then run directly in Qt Creator, and the results were printed in the Application Output panel.

The testing approach focused on:

- Validating the *happy-path behavior* for regular system use,
- Testing the limits and the system's ability to process invalid inputs.
- Establishing proper mock and interpretation testing by Google Mock.

The core components that were tested are:

- Validation (input validation logic),
- Payroll (workers' hours, overtime, and pay calculations),
- FileGateway (through a mock interface to illustrate controlled interaction testing).

5.2 Unit Tests with GoogleTest

Validation Tests

File: tests/test_validation.cpp

Class under test: Validation

These tests verify that any invalid user input is rejected before it can corrupt application data. Test cases include:

- **ValidEmployeeIdAccepted**
  Confirms acceptance of numeric IDs with 4 to 8 digits (e.g., "1234", "987654").
- **InvalidEmployeeIdRejected**
  Rejects IDs that are too short, too long, or contain non-digit chars (e.g., "12", "abc1", "123456789").
- **ValidShiftTimesAccepted**
  Confirms that the start/end times of the shift are valid if the end time is after the start time and the total duration is reasonable.
- **InvalidShiftTimesRejected**
  Verifies the rejection of invalid shifts, for example, end time before start time or shifts exceeding the maximum duration.

These tests ensure that the system applies strict validation rules and does not allow invalid data entry at the earliest stage.

Payroll Tests

File: tests/test_payroll.cpp

Class under test: Payroll

The Payroll logic was tested on **in-memory text data** that simulated the ids.txt and emp_clock.txt. This way, input was controlled very precisely without doing anything with real files.

Test cases include:

- **ComputesRegularHoursAndPay**
  Confirms that an hourly employee is paid according to the role rate configured for standard (non-overtime) hours.

- **ComputesOvertimeAboveThreshold**

  Confirms that when total hours surpass the overtime threshold:

  - Regular hours are limited
  - Overtime hours are calculated correctly
  - Overtime pay is multiplied by the specified rate
  - Net pay is the sum of regular and overtime pay.
- **FlatRateRolePaysPerShift**

  Guarantees that flat-rate employees (F) are paid per shift completed, regardless of duration, and no overtime is applied.

These tests validate the correctness of payroll calculations for both hourly and flat-rate roles and confirm that overtime rules are applied consistently.

5.3 Mocking with Google Mock

File: tests/test_filegateway_mock.cpp

Google Mock provides the ability to create mock versions of classes to verify what your code is doing by interacting with a mock object in place of an existing object. This is useful for unit testing the interaction of your code with other parts of your application, as well as third-party libraries or APIs.

In this example, we have created a mock version of the FileGateway class. In the mock, the writeAll() method is declared as a mocked method. This allows us to declare our expectations about how it will be used.

Example:

EXPECT_CALL(mock, writeAll(path, payload))

Times(1)

WillOnce(::testing::Return(true));

We then check in the associated test that the writeAll() method is called precisely one time, it is called with the correct file path and data, and the calling method responds appropriately to a successful write.

This is an example of interaction testing, which is a major focus of Google Mock and helps to ensure the file-writing functionality behaves as expected without actually writing to the file system.

5.4 Edge Case Handling

There were several edge cases specifically tested throughout the test suite:

- Employee ID Validation

Non-numeric characters in an employee's ID, and/or an ID of an improper length, will cause the program to reject them to prevent bad records.

- Shift Time Validation

If there is a shift in the schedule with an invalid order, or if the shifts overlap for too long, the program will block the scheduling of those shifts, because they are impossible to schedule.

- Payroll Overtime Edge Cases

Payroll tests verified that payroll calculations behave correctly both when the total number of hours worked is at the exact overtime boundary and when it is slightly greater than the boundary.

- Hourly vs Flat-Rate Roles

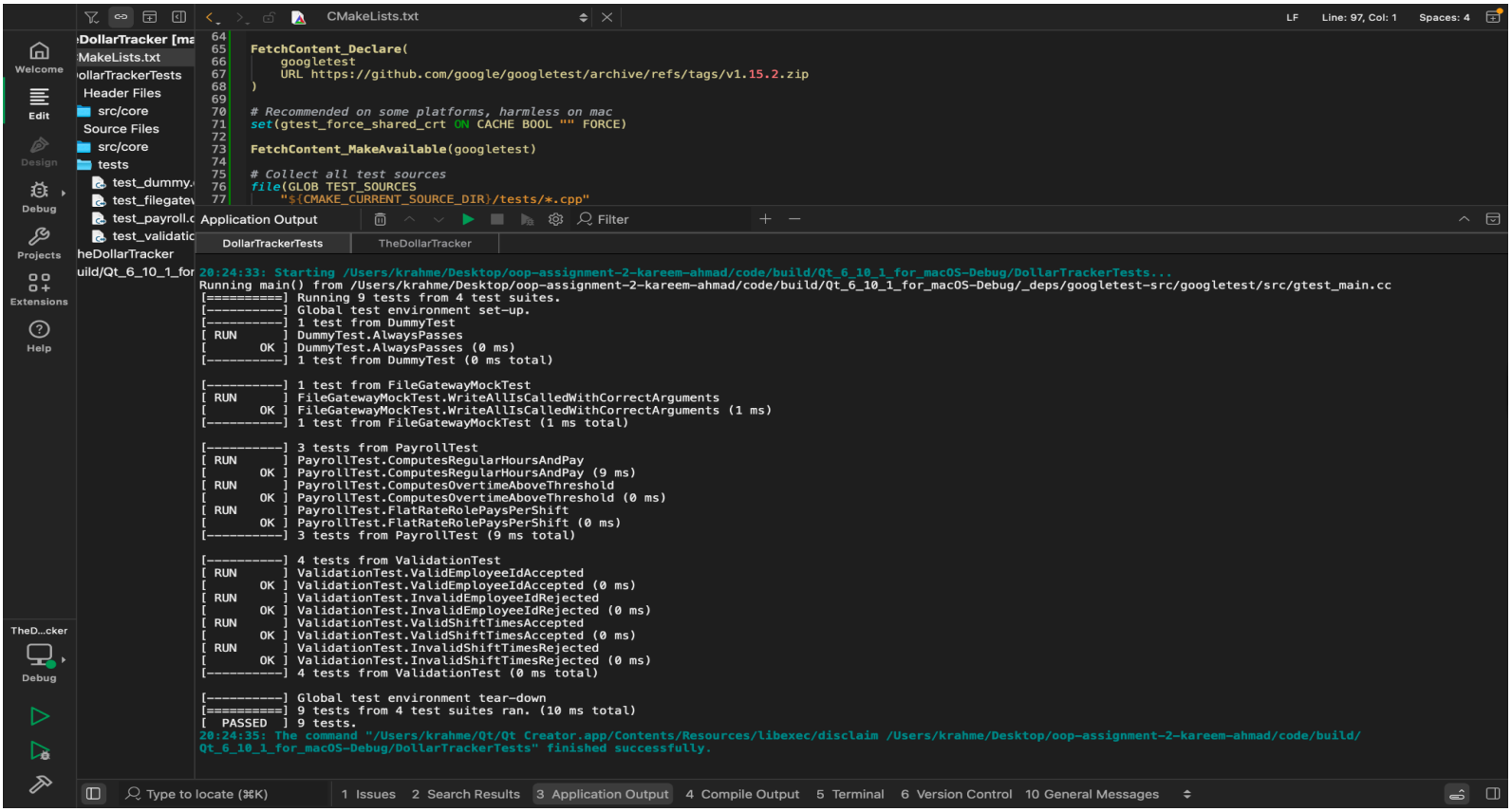
Tests confirmed that overtime rules apply only to hourly employees, while flat-rate employees are handled separately. Through the inclusion of these edge case tests, the program demonstrates its robustness to errors and its predictability in response to unexpected inputs.

5.5 Test Execution Results

The entire test suite was run from within Qt Creator, utilizing the *DollarTrackerTests* target, and generated the following results.

- **4 test suites**
- **9 total tests**
- **100% pass rate**

Output:


CMakeLists.txt
LF
Line: 97, Col: 1
Spaces: 4
Welcome
Edit
Design
Debug
Projects
Extensions
Help
DollarTracker [ma
MakeLists.txt
ollarTrackerTests
Header Files
src/core
Source Files
src/core
tests
test_dummy.
test_filegatev
test_payroll.c
test_validatic
heDollarTracker
uild/Qt_6_10_1_for
TheD...cker
Debug
64 FetchContent_Declare(
65 googletest
66 URL https://github.com/google/googletest/archive/refs/tags/v1.15.2.zip
67)
68
69 # Recommended on some platforms, harmless on mac
70 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
71
72 FetchContent_MakeAvailable(googletest)
73
74 # Collect all test sources
75 file(GLOB TEST_SOURCES
76 "${CMAKE_CURRENT_SOURCE_DIR}/tests/*.cpp"
77
Application Output
Filter
DollarTrackerTests
TheDollarTracker
20:24:33: Starting /Users/krahme/Desktop/oop-assignment-2-kareem-ahmad/code/build/Qt_6_10_1_for_macOS-Debug/DollarTrackerTests...
Running main() from /Users/krahme/Desktop/oop-assignment-2-kareem-ahmad/code/build/Qt_6_10_1_for_macOS-Debug/_deps/googletest-src/googletest/src/gtest_main.cc
[==========] Running 9 tests from 4 test suites.
[----------] Global test environment set-up.
[----------] 1 test from DummyTest
[RUN] DummyTest.AlwaysPasses
[OK] DummyTest.AlwaysPasses (0 ms)
[----------] 1 test from DummyTest (0 ms total)
[----------] 1 test from FileGatewayMockTest
[RUN] FileGatewayMockTest.WriteAllIsCalledWithCorrectArguments
[OK] FileGatewayMockTest.WriteAllIsCalledWithCorrectArguments (1 ms)
[----------] 1 test from FileGatewayMockTest (1 ms total)
[----------] 3 tests from PayrollTest
[RUN] PayrollTest.ComputesRegularHoursAndPay
[OK] PayrollTest.ComputesRegularHoursAndPay (9 ms)
[RUN] PayrollTest.ComputesOvertimeAboveThreshold
[OK] PayrollTest.ComputesOvertimeAboveThreshold (0 ms)
[RUN] PayrollTest.FlatRateRolePaysPerShift
[OK] PayrollTest.FlatRateRolePaysPerShift (0 ms)
[----------] 3 tests from PayrollTest (9 ms total)
[----------] 4 tests from ValidationTest
[RUN] ValidationTest.ValidEmployeeIdAccepted
[OK] ValidationTest.ValidEmployeeIdAccepted (0 ms)
[RUN] ValidationTest.InvalidEmployeeIdRejected
[OK] ValidationTest.InvalidEmployeeIdRejected (0 ms)
[RUN] ValidationTest.ValidShiftTimesAccepted
[OK] ValidationTest.ValidShiftTimesAccepted (0 ms)
[RUN] ValidationTest.InvalidShiftTimesRejected
[OK] ValidationTest.InvalidShiftTimesRejected (0 ms)
[----------] 4 tests from ValidationTest (0 ms total)
[----------] Global test environment tear-down
[==========] 9 tests from 4 test suites ran. (10 ms total)
[PASSED] 9 tests.
20:24:35: The command "/Users/krahme/Qt/Qt Creator.app/Contents/Resources/libexec/disclaim /Users/krahme/Desktop/oop-assignment-2-kareem-ahmad/code/build/Qt_6_10_1_for_macOS-Debug/DollarTrackerTests" finished successfully.
Type to locate (⌘K)
1 Issues
2 Search Results
3 Application Output
4 Compile Output
5 Terminal
6 Version Control
10 General Messages

6. Video Walk-through

Link:

https://youtu.be/buFA73xWPoM?si=Pm07eGsRYWNOY8KP

7. Reflection

Many issues arose while creating *The Dollar Tracker* that greatly influenced the growth of knowledge in how to create quality software applications through the use of software design, Qt application development, and software architecture.

Issues Encountered

1. Separating UI & Business Logic

There existed a natural tendency to put logic inside UI classes. Transferring this type of logic to the business logic area of the system allowed for a clean and maintainable design once the architectural discipline was exercised.

2. Text Data (Parsing & Management)

The application will be using plain text files instead of a database; therefore, great care must be taken when processing the text in regards to format, white space, and edge cases. Examples of such processing can be seen in the following areas:

- Time parsing from Payroll information.
- Creating and inserting Shifts
- Updating Employees and Roles

3. Adding GoogleTest & GoogleMock

Adding a testing environment to a Qt/CMake-based application was not easy. Problems encountered were:

- Building tests as a different target in CMake and configuring it.
- Setting up the include path for Qt in Tests and using Google Mock to simulate file operations.
- This learning experience significantly expanded on knowledge of CMake, linking, and TDD.

4. Providing Good Error Handling

Each edge case (i.e., Invalid shifts, Malformed IDs, missing files, etc.) had to be properly accounted for to ensure the system would work reliably and required a lot of thought regarding defensive programming.

Trade-Offs Made

1. Using .txt files vs A Database

The use of .txt files increased the ease of use and portability of the application, but decreased the amount of data integrity.

The trade-offs we made fit well with the course scope:

- Easy to Parse
- Easy to Inspect Manually
- Easier to write Unit Tests for

2. Trade Off: UI Complexities vs Simplicity

Some of the Dialog Interactions could have been created with more complex MVC designs; however, we used simpler models to keep the overall system understandable and within the scope of the project.

3. Trade Off: Test Coverage vs Time

We have full coverage of all core logic functionality, but we have limited unit testing of some UI elements (e.g., Qt dialogs) due to their complexity.

We made this choice intentionally since we needed a tool like Qt Test to perform UI testing, and the value of having the business logic functionally tested is greater than the cost of unit testing each UI element.

Lessons Learned

1. Architecture Layering is Important

By separating the UI layer from the logic layer and input/output layers, we created a system that is both more maintainable and scalable, and future additions to the system (i.e., a database backend) could be done independently of any future changes to the UI.

2. Design Patterns Provide Clarity

Design patterns make the code more readable by reducing the number of ways a task can be implemented:

- Singleton provides a single way to access the file I/O system,
- Factory Method allows the UI to be constructed in one location and decouples the UI from the rest of the system.
- Gateway/Facade reduces the complexity of getting data from one location to another and makes testing simpler.

3. Writing Code to Be Tested Saves Time Later

With the testing established, finding logical errors in the payroll rule set became much faster. Additionally, it gave us a level of confidence that the payroll rules worked as expected across multiple scenarios.

4. Good Documentation and Modularity Improves Communication & Debugging

Final Reflection

This project has driven home the fact that good software design is not an accident. It takes effort, consideration, and a disciplined approach to designing and building systems that apply industry standards for software engineering and provide a positive user experience. *The Dollar Tracker* evolved from a basic scheduling concept to a structured application that demonstrates many industry-relevant principles of software engineering. The process has given me first-hand experience with actual software tools, including Qt, CMake, and GoogleTest, and has further emphasized the importance of taking the time to think through your design and make iterative improvements to your design.

8. Conclusion

Through the successful execution of object-oriented design techniques, modularization, and design patterns, *The Dollar Tracker* has transformed from a conceptual idea into a fully functioning desktop application that integrates all aspects of employee management, scheduling, time-tracking, and payroll processing under one roof. The clarity, robustness, and maintainability of the final product were made possible by applying well-established design principles to achieve key goals (access control based on roles, accurate payroll calculations including overtime, persistent and consistent data storage, and user-friendliness) and validating them through an extensive series of automated tests utilizing GoogleTest and GoogleMock. Additionally, the use of unit tests and dependency mocking enabled confident system performance and the ability to support future extension or refactoring.

This project further emphasized the significance of early architectural design, separating concerns, and defensive programming. As well, this project illustrated how automated testing enhances the degree of confidence in correctness as system complexity grows. In summary, *The Dollar Tracker* is a representative example of translating theoretical object-oriented concepts into practical software engineering problems and represents a completely professional quality solution meeting the objectives outlined for CS 3307A.