fsoft2025_1DA_2: Iteration 3 - Technical Report
FleetManager's C++ Code Implementation And Testing

Dinis Gonçalves, Lucas Santos, Rúben Silva, and Vítor Teixeira

ISEP, Instituto Superior de Engenharia do Porto,
Rua do Dr António Bernardino de Almeida 431, 4249-015 Porto, Portugal
{1241099,1241008,1240708,1232067}@isep.ipp.pt
https://www.isep.ipp.pt

Abstract. In the following pages, the reader will be able to be informed about the application's implementation (using the C++ programming language) and testing (using Google's Testing Framework).

Key words: Business, C++, Fleet Management, IPP, ISEP, Management, Programming, Software Development, Command-Line Interface, Data Persistence, Expense Tracking, JSON, Google, Software Testing, Google C++ Testing Framework

1 Introduction

In this report, we present the design and implementation of our application, FleetManager, a complete, feature-rich, fleet management system developed in C++. The application offers a integrated solution for managing a wide variety of sectors within a targeted company, such as fleet, driver, trip, financial and client order management.

Our primary focus was to create an all-in-one software for the companies in this rapidly evolving industry. We believe that, by centralizing everything in just one piece of software, companies will spend less time worrying and complaining about connection problems between their software and more time being productive and contributing to the economy.

This report provides details of the application's development process, including code examples, explanations of system functionality, and testing.

2 Implementation

The entire application is called in the `main.cpp` file. The content of such file is meant to be simple, you should only create the core object and give the instruction to run the program. We did a little more by encompassing everything in a `try-catch` and having a different returning code for every exception caught.

```cpp
#include <iostream>
#include "Enterprise.h"
#include "Controller.h"
#include "DuplicatedDataException.h"
#include "InvalidDataException.h"
#include "NonExistingDataException.h"

using namespace std;

int main() {
    try {
        Enterprise enterprise("fsoft2025_1DA_2");

        Controller controller(enterprise);
        controller.run();

        cout << "\nClosing FleetManager..." << endl;

        return 0;
    }
    catch (const DuplicatedDataException &error) {
        cout << error.what() << endl;
        return 1;
    }
    catch (const NonExistingDataException &error) {
        cout << error.what() << endl;
        return 2;
    }
    catch (const InvalidDataException &error) {
        cout << error.what() << endl;
        return 3;
    }
}
```

Exceptions are classes that manage and redirect the program's flow when an unexpected error occurs. FleetManager has three different types of exceptions that guarantee that integrity remains untouched.

```cpp
#include "DuplicatedDataException.h"

DuplicatedDataException::DuplicatedDataException(const string &data)
    : runtime_error("!! Error: \"" + data + "\" duplicated !!") {}

const char *DuplicatedDataException::what() const noexcept {
    return runtime_error::what();
}
```

```cpp
#include "InvalidDataException.h"

InvalidDataException::InvalidDataException(const string &data)
    : runtime_error("!! Error: \"" + data + "\" is invalid !!") {}

InvalidDataException::InvalidDataException(): runtime_error("") {}

const char *InvalidDataException::what() const noexcept {
    return runtime_error::what();
}
```

```cpp
#include "NonExistingDataException.h"

NonExistingDataException::NonExistingDataException(const string &data)
    : runtime_error("!! Error: \"" + data + "\" does not exist !!") {}

NonExistingDataException::NonExistingDataException() : runtime_error("") {}

const char *NonExistingDataException::what() const noexcept {
    return runtime_error::what();
}
```

When the user starts FleetManager, a menu will be shown to navigate through the different areas of the program. The *Login Menu* contains the options for all the different managers available. In subsequent menus, after choosing the manager, it will be shown all the manager's actions.

******* Login Menu *******

1. Fleet Manager
2. Driver Manager
3. Trip Manager
4. Financial Manager
5. Order Manager

0. Exit

Your Option: 2

******* Driver Manager's Menu *******

1. Add Driver
2. Remove Driver
3. List Drivers
4. List Available Drivers
5. List Unavailable Drivers
6. Add Vacation

0. Go Back

Your Option: 1

The classes are a important part of any program, however, in `Controller.cpp` is where the magic happens. By that it is meant that the instructions of every option shown in the menu are delimited in the controller. In order not to make this report too long we'll only analyze code from `Driver`, as the others follow, more or less, the same pattern in some characteristics. Let's look at the example of "Add Driver", the first option in the Driver Manager's Menu.

```
void Controller::runDriver() {
int op = -1;
do {
    op = this->view.menuDriver();
    switch (op) {
        case 1: {
            Driver driver = this->driverView.addDriver();
            DriverContainer &containerDriver =
            ↪  this->model.getDriverContainer();
            containerDriver.add(driver);
        }
            break;
(...)
```

We've implemented data persistence into FleetManager. When the user terminates the application, all the data is stored in JSON files. Also, when the application is launched, whatever data existent in those JSON files is restored back into objects.

To implement this process were needed four different methods for all class containers:

- `toJSON` - parse class objects to JSON key-value pairs
- `fromJSON` - parse JSON key-value pairs to class objects
- `loadAllData` - restore data from the JSON files into objects (occurs every time the program starts)
- `saveAllData` - write the objects, already parsed to JSON key-value pairs, into the JSON files (occurs every time the program finishes)

```
void DriverSerialization::toJSON(json &j, const Driver &driver) {
    j["id"] = driver.getID();
    char licenseChar = driver.getLicense();
    string licenseString = string(1, licenseChar);
    j["license"] = licenseString;
    j["age"] = driver.getAge();
    j["timeToRetire"] = driver.getTimeToRetire();
    j["available"] = driver.getAvailability();

    j["vacations"] = json::array();
    const list<Vacation*>& driverVacations =
    ↪  driver.getVacations();
    for(const Vacation* vacation : driverVacations) {
        json vacationJSON;

        vacationJSON["id"] = vacation->getID();
        vacationJSON["driverID"] = vacation->getDriver()->getID();
        vacationJSON["startDate"] =
        ↪  vacation->getStartDate().dateToString();
        vacationJSON["endDate"] =
        ↪  vacation->getEndDate().dateToString();
        vacationJSON["status"] = vacation->getStatus();

        j["vacations"].push_back(vacationJSON);
    }
}
```

```
void DriverSerialization::fromJSON(const json &j, Driver &driver)
↪  {
    driver.setID(j["id"]);
    string licenseString = j["license"];
    char char_license = driver.stringToChar(licenseString);
    driver.setLicense(char_license);
    driver.setAge(j["age"]);
    driver.setTimeToRetire(j["timeToRetire"]);
    driver.setAvailability(j["available"]);

    driver.getVacations().clear();

    if (j["vacations"].is_array() && !j["vacations"].empty()) {
        for(const basic_json<> &vacationJSON : j["vacations"]) {
            Vacation *vacation = new Vacation();
            Date startDate;
            Date endDate;
```

```
            vacation->setID(vacationJSON["id"]);

            if (vacationJSON["driverID"] == driver.getID()) {
                vacation->setDriver(&driver);
                vacation->setStartDate(startDate.stringToDate(
                                vacationJSON["startDate"]));
                vacation->setEndDate(endDate.stringToDate(
                                vacationJSON["endDate"]));
                vacation->setStatus(vacationJSON["status"]);

                driver.setVacation(vacation);
            }
        }
    }
}
```

```
void DataContainer::loadAllData() {
    (...)
    loadDrivers();
    (...)
}
void DataContainer::loadDrivers() {
    ifstream file(driverFilePath);
    cout << "Loading \"driver.json\"... ";
    try {
        if (!file.is_open()) {throw
        ↪  NonExistingDataException("Driver JSON File");}
    } catch (NonExistingDataException &error) {
        cout << "\n" << error.what() << endl;
    }
    cout << " ✓" << endl;
    json j;
    file >> j;
    DriverSerialization serializer;
    if (j.contains("drivers")) {
        for (const basic_json<> &driverJSON : j["drivers"]) {
            Driver driver;
            serializer.fromJSON(driverJSON, driver);
            containerDriver->add(driver);
        }
    }
    file.close();
}
```

```
void DataContainer::saveAllData() {
    (...)
    DriverSerialization serializerDriver;
    (...)
    json dataDriver;
    json driverArray = json::array();
    for (const Driver &driver : containerDriver->listDrivers()) {
        json driverJSON;
        serializerDriver.toJSON(driverJSON, driver);
        driverArray.push_back(driverJSON);
    }
    dataDriver["drivers"] = driverArray;

    ofstream driverOutputFile(driverFilePath);
    cout << "Saving into \"driver.json\"...";
    try {
        if (!driverOutputFile.is_open()) {
            throw NonExistingDataException("Driver JSON File");
        }
        driverOutputFile << setw(4) << dataDriver << endl;
        driverOutputFile.close();
        cout << " ✓" << endl;
    } catch (NonExistingDataException &error) {
        cout << error.what() << endl;
    }
    (...)
}
```

3 Testing

In this part of the project we used the `Google C++ Testing Framework`. We've tested the vast majority of FleetManager's features and methods.

Each class follows a strict testing pattern to encounter suspicious behavior during FleetManager's execution. The following code is a test for the initialization of a Driver's empty constructor:

```
class DriverTest : public ::testing::Test {
protected:
    void SetUp() override {
        driver = new Driver();
    }
    void TearDown() override {
        delete driver;
        driver = nullptr;
    }
    Driver* driver;
};

TEST_F(DriverTest, ConstructorAndInicialization) {
    EXPECT_EQ(driver->getID(), 0);
    EXPECT_EQ(driver->getLicense(), '\0');
    EXPECT_EQ(driver->getAge(), 0);
    EXPECT_TRUE(driver->getAvailability());
    EXPECT_TRUE(driver->getVacations().empty());
    EXPECT_EQ(driver->getTimeToRetire(), 65);
}
```

To make a clean and error-proof data persistence test is an art. It requires thinking on the most sophisticated error scenarios and being able to foresee it.

```
TEST_F(DataTest, LoadAllData) {
    containerData->loadAllData();
    (...)
    EXPECT_EQ(containerDriver->listDrivers().size(), 2);
    EXPECT_NE(containerDriver->get(1), nullptr);
    EXPECT_EQ(containerDriver->get(1)->getLicense(), 'B');
    EXPECT_TRUE(containerDriver->get(1)->getVacations().empty());
    EXPECT_NE(containerDriver->get(2), nullptr);
    EXPECT_EQ(containerDriver->get(2)->getLicense(), 'C');
    EXPECT_TRUE(containerDriver->get(2)->getVacations().empty());
    (...)
```

```
TEST_F(DataTest, SaveAllData) {
    containerData->loadAllData();
    (...)
    Driver *driver = new Driver(999, driverLicense, 20);
    containerDriver->add(*driver);
    (...)
    containerData->saveAllData();
    (...)
    DriverContainer *loadedContainerDriver = new
    ↪  DriverContainer();
    (...)
    DataContainer *loadedContainerData = new DataContainer(
        (...)
        loadedContainerDriver,
        (...)
        "dummy-json-data/driver.json",
        (...)
    );
    loadedContainerData->loadAllData();
    (...)
    EXPECT_EQ(loadedContainerDriver->listDrivers().size(), 3);
    EXPECT_NE(loadedContainerDriver->get(999), nullptr);
    EXPECT_EQ(loadedContainerDriver->get(999)->getLicense(), 'C');
    EXPECT_TRUE(
        loadedContainerDriver->get(999)->getVacations().empty());
    (...)
```