SmartFin

Tyler Spring
3/2/2025

Introduction

This paper will cover the designs, functionally, and testing of the project called, “SmartFin”. SmartFin is a backend, Java based project. Its main purpose is to serve as a learning experience in developing projects at scale with an emphasis on data structures, modularity, data integrity and unit testing. The modules Password Manager, ExpenseTracker and SavingsGoalEst demonstrate this. PredictCat and nbTrainer focus more on dependency Injection and testing while introducing new features like Naives Bayes and tokenization. PWMGUI, PasswordManager and SmartFin focus on user experience(UI) while implementing security through encryption and safing storing data via .dat files.

Design Decisions and Architecture

Modular Design

Modular design of a program is meant to keep functionality of a program in separate modules in order to limit complexity while maintaining efficiency, reusability, and scalability. This design technique allows each module to focus on one sole task with minimal impact on others while allowing them to work together to create the overall functionality of the program and allow for effective testing on each module. Keeping the modules PWMGUI and PasswordManager separate is a prime example of this. PWMGUI serves as the UI for passwords allowing for user input, while PasswordManager performs the logic with encryption and securely storing of passwords. The two files interact with each other, but PasswordManager remains independent of PWMGUI, ensuring separation of concerns.

Data Encapsulation and Integrity

Data encapsulation is a principle of object-oriented programming where data or variables are encapsulated within methods that perform operations on them within a class. This practice essentially hides the data from unauthorized or accidental manipulation. These variables are declared as private and exposed only through controlled methods known as “getters and setters”. Getters allow access to the variable in a controlled method without allowing direct modification. A setter method allows controlled updates to the internal state while enforcing constraints and rules.

The Expense and Loan classes follow this principle in validating, storing, and protecting their variables inside their respective class files. The Expense class is responsible for creating the variables “amount”, “description”, “date” and “category”. These variables are declared as private and are accessible through the defined getter methods, with the exception of category, which has both a getter and setter method. This is because Category can be modified dynamically for training and predictions in the nbTrainer and PredictCat classes.

Loan is much smaller in comparison, with only three getter methods for “Principal”, “Rate” and “Term”. These values remain unchanged after initialization and are used in calculations performed in the LoanCalc class.

Dependency Injection

Dependency Injection (DI) is the practice of keeping creation and usage of objects in object oriented programming (OOP) separate. Objects define their dependencies through constructor parameters or setter methods. An external component provides the dependencies, rather than the objects creating them itself. The object receiving these dependencies does not need to know how to instantiate them, sticking to modular design making it more flexible and testable.

Aside from being a modular design choice and aiding in code reuse and maintainability, in SmartFin it aided in unit testing greatly through Maven. Maven is an open source build automation and dependency management tool for Java projects. Through its Project Object Model (POM) file, Maven automates dependency solutions, compilation and testing. Using DI, SmartFin decouples its AI training module (nbTrainer) from fixed datasets, allowing flexible data injection. This improves testability, adaptability, and separation of concerns. Combined with Maven for dependency management and automated testing, this approach enhances SmartFin’s ability to evolve its AI predictions without affecting its core functionality.

Key Features and Functionalities

Password Management

SmartFin handles password storage and retrieval using an encrypted .dat file, with an in-memory, static hashmap for managing passwords during runtime. When a password is stored, it is encrypted before being written to the file, ensuring that plaintext credentials are never exposed. Upon retrieval, the password is read from the file, decrypted, and returned to the user. Validation in PasswordManager ensures that only properly formatted usernames and passwords are processed before encryption and storage. Security logic is centralized, meaning passwords can only be accessed through controlled methods that enforce encryption and decryption, rather than being exposed directly.

PasswordManager serves as the security backbone of SmartFin, safeguarding credentials by handling encryption, decryption, and validation. By maintaining an encrypted storage mechanism, SmartFin ensures that password management remains both secure and structured.

Expense Management and Categorization

SmartFin organizes expenses through the Expense class, which serves as the core data structure for storing individual expense entries. Each expense consists of an amount, description, category, and an optional date. To ensure data integrity, Expense validates inputs upon creation, enforcing rules like requiring a positive amount. The class also provides controlled access through getter methods while allowing category updates via a setter, enabling both manual classification and AI-assisted categorization. By default, expenses are labeled as "Uncategorized" if no category is assigned, ensuring consistency in data management.

Additionally, the class overrides toString(), simplifying debugging, logging, and verification of expense records. Through this structured design, Expense seamlessly integrates with SmartFin's AI-powered categorization system while maintaining flexibility for the user adjustments.

Predictive Categorization

To enhance automation, SmartFin employs a Naive Bayes-AI prediction system for categorization. The nbTrainer and predictCat classes handle this prediction process by analyzing past expense descriptions and calculating the probability of each category based on learned patterns. PredictCat tokenizes expense descriptions and assigns a category by applying Bayes's Theorem to determine the most likely match. This allows SmartFin to make informed predictions, improving with more data over time. Once a category is predicted, the Expense class integrates the result, ensuring that new expenses are categorized consistently while still allowing users to make manual adjustments. This predictive approach streamlines expense tracking, reduces manual input, and enhances SmartFin's ability to adapt to evolving spending patterns.

Testing Strategies

Before Each

Unit testing is the process of testing individual parts of a project to ensure the quality of the code and prevent bugs early and through the development of the project. A unit is a small testable part of the project like a function, method, module or entire class.

BeforeEach is an annotation that ensures a method runs before each individual test case within a test class. This helps create a predictable, isolated test environment by resetting shared resources before each test, preventing unintended interference between test cases. Reducing the effect each test has on each other following it. This is practiced in nbTrainer and PasswordManager where the objects being tested in those classes are initialized before each test to avoid inference from previous tests.

InputStream

Testing methods or classes that process user input typically involves verifying that an InputStream, such as a Scanner, correctly handles data.

In SmartFin's unit testing, this was achieved by simulating user input through a command-line interface (CLI). A Scanner object is used to read from a simulated input stream, allowing automated testing of menu selections, user choices, and other input methods without manual intervention.

User Input

In SmartFin, ensuring robust handling of user input is critical for maintaining the system's reliability and preventing unexpected behavior. Extended testing was conducted to evaluate how the system processes both valid and invalid data, improving overall robustness and the user experience. The tests focus on ensuring valid input, such as positive numeric values for amounts or properly formatted categories, is processed correctly. Invalid input, including negative amounts, empty categories, and non-numeric characters where numbers are expected, is tested to ensure the system responds appropriately, either by returning error messages or prompting for re-entry. Additionally, boundary testing was performed to check how the system handled extreme values, such as very large or small numeric inputs, ensuring no overflow errors or incorrect calculations occur. These tests ensure that SmartFin handles various input cases gracefully, preventing crashes and maintaining accuracy, thus enhancing the system's overall stability.

Static Method Testing

Testing static methods directly plays a crucial role in ensuring the correctness and efficiency of core functionality without the need for unnecessary overhead related to the user interface (UI) logic. By isolating and testing methods that perform key operations – such as those in PasswordManager or other utility classes – developers can focus on validating the core business logic without the complexity of external dependencies. This direct testing allows for faster and more targeted verification of functionality, ensuring that methods perform as expected. For example, testing the encryption and decryption logic in the PasswordManager class helps ensure that these processes work correctly, without requiring UI interaction. This approach streamlines testing, improves test coverage, and allows developers to catch potential issues early in the development process.

Challenges and Areas for Future Enhancement

Handling Rare Tokens and Missing Data

Being able to handle new or unpredictable categories during the tokenization process was one of the reasons Naive Bayes worked well for SmartFin's auto-categorization functionality. However, it suffers from the "Zero Frequency Problem," where a new token that is not present in the training data can result in a zero probability for a prediction, making it impossible to classify certain expenses accurately.

Currently, SmartFin does not have a built-in solution for this issue. One common approach to address this is Laplace Smoothing (Additive Smoothing), which assigns small default probability to unseen tokens, preventing probabilities from reaching zero. Integrating Laplace Smoothing or similar smoothing techniques could improve SmartFin's ability to handle rare or new expense descriptions without breaking the prediction model.

Additionally, SmartFin could enhance its prediction accuracy by implementing real-time model updates based on user conditions. By incorporating a feedback loop where user-adjusted categories are fed back into the model, SmartFin can dynamically learn and adapt to new spending patterns. Future implementations may also explore hybrid models, combining Naive Bayes with more advanced machine learning techniques like decision trees or neural networks to improve categorization accuracy further.

By addressing the Zero Frequency Problem and refining how SmartFin handles missing or rare data, the system can become more resilient and capable of adapting to evolving expense trends.

Conclusion

Takeaways

The purpose of this project was to practice developing modular, iterative, user-centered systems while maintaining scalability and adaptability. Through the building of this project there was a strong emphasis on flexibility and robustness. I was able to focus and take both short and long term challenges, all without any compromise to the overall project's design. This was achieved in learning and implementing new design techniques like dependency injection and unit testing each implementation at every step. Building modular validation checks and the usage of static methods added to the robustness while also minimizing errors during the operation and testing with the removal of overhead.

Lessons Learned/Informal

This project may not be groundbreaking or especially creative in the conventional sense, aside from a few additional features. However, that was never the goal. The purpose was to take an idea for a well-designed system and build it start to finish. From planning out design choice, functionality, and classes, to sticking to core principles throughout the process, the focus was on maintaining a structured approach. Along the way, I learned new techniques like static testing and using Maven for unit testing. Debugging also presented its own challenges; for instance, it took me three hours of trial and error before I realized that converting to a Unix timestamp could be done simply by changing a value in the POM file. Getting to the point where I could write a paper about the project brought everything full circle. Summarizing the classes and their interactions, explaining the reasoning behind certain design choices, and reflecting on how each class works independently while still collaborating effectively helped me see how everything connected. Ultimately, this project has shown me just how much learning can be gained by simply building stuff.

References

GeeksforGeeks. (n.d.). *Naïve Bayes classifiers*. GeeksforGeeks. Retrieved March 9, 2025, from https://www.geeksforgeeks.org/naive-bayes-classifiers/

GeeksforGeeks. (n.d.). *Unit testing – Software testing*. GeeksforGeeks. Retrieved March 9, 2025, from https://www.geeksforgeeks.org/unit-testing-software-testing/

TinyMCE. (n.d.). *Modular programming principle: What it is and why it matters*. TinyMCE. Retrieved March 9, 2025, from https://www.tiny.cloud/blog/modular-programming-principle/

Stack Overflow. (n.d.). *What is dependency injection?* Stack Overflow. Retrieved March 9, 2025, from https://stackoverflow.com/questions/130794/what-is-dependency-injection