Functional vs. Object-Oriented: Comparing how Programming Paradigms affect the architectural characteristics of systems

Briza Mel Dias de Sousa[1], Alfredo Goldman Vel Lebjman[1], and Renato Cordeiro Ferreira[1]

[1] Institute of Mathematics and Statistics - University of São Paulo

Introduction

After decades of dominance of object-oriented programming, the functional paradigm is becoming widespread in the industry. According to the IEEE Spectrum magazine in 2022 [1], one sign that the software industry is preparing for a paradigm shift is that functional features are showing up in more and more mainstream languages.

The present project aims to compare the impact of Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms on developers' understanding of various architectural characteristics and non-functional requirements.

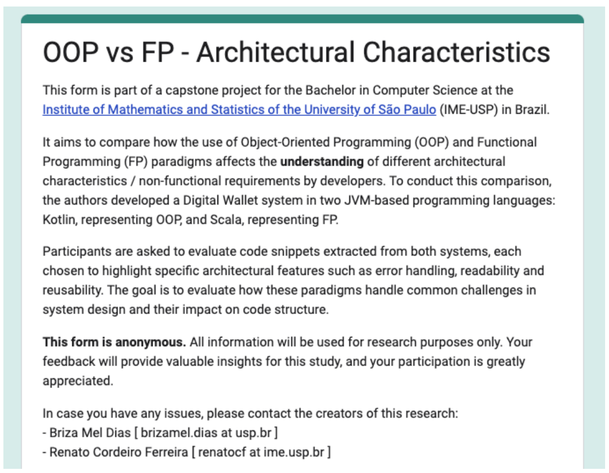
Methodology

A Digital Wallet system has been developed using two JVM-based programming languages: Kotlin (representing OOP) and Scala (representing FP).

Both versions of the Digital Wallet system will be evaluated based on the following key characteristics: readability, maintainability, extensibility, testability, reusability, error handling and error propagation.

The evaluation of both system implementations will be conducted through two approaches: a qualitative analysis by the author, and a quantitative analysis based on feedback from other developers via a survey.

Survey



OOP vs FP - Architectural Characteristics

This form is part of a capstone project for the Bachelor in Computer Science at the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP) in Brazil.

It aims to compare how the use of Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms affects the **understanding** of different architectural characteristics / non-functional requirements by developers. To conduct this comparison, the authors developed a Digital Wallet system in two JVM-based programming languages: Kotlin, representing OOP, and Scala, representing FP.

Participants are asked to evaluate code snippets extracted from both systems, each chosen to highlight specific architectural features such as error handling, readability and reusability. The goal is to evaluate how these paradigms handle common challenges in system design and their impact on code structure.

This form is anonymous. All information will be used for research purposes only. Your feedback will provide valuable insights for this study, and your participation is greatly appreciated.

In case you have any issues, please contact the creators of this research:
- Briza Mel Dias [brizamel.dias at usp.br]
- Renato Cordeiro Ferreira [renatocf at ime.usp.br]

Figura 1: Survey: The survey contains four questions with code snippets extracted from the Digital Wallet System. It also contains background questions.

Architecture

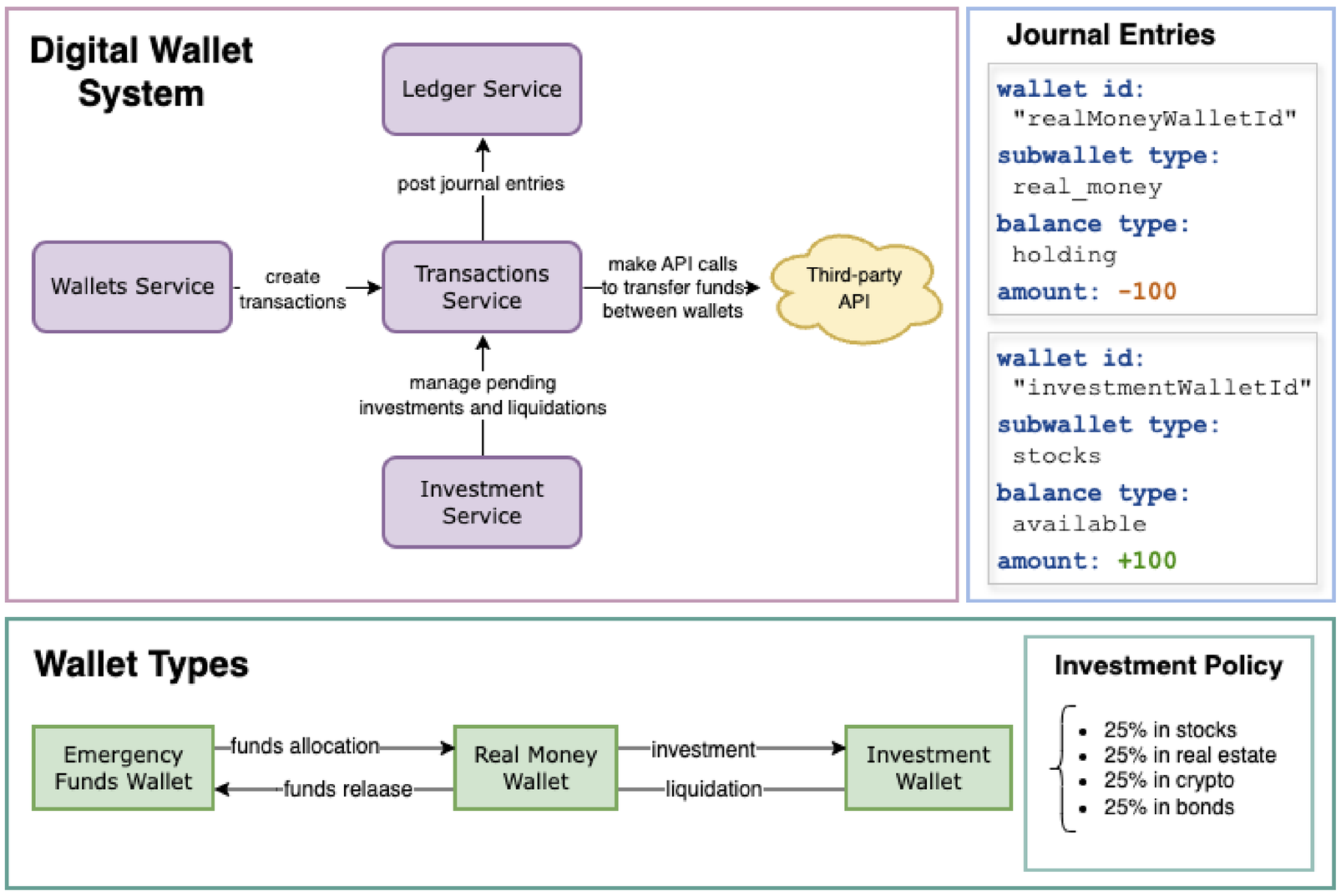


Figura 2: Architecture: The figure illustrates the architecture of the Digital Wallet system, highlighting the main services, one of which interacts with a third-party API. It also depicts essential system models, such as Wallet, Investment Policy, and Journal Entry.

Qualitative Analysis

- Kotlin represents different Wallet types using OOP concepts like inheritance and polymorphism, while Scala uses enumerations and pattern matching. Both approaches ensure code updates are enforced via compilation errors when adding new Wallet types;
- Scala adopts more generic solutions through higher-order functions. While Kotlin's `retryBatch` method is tightly coupled to the transactions domain, Scala implements a reusable `retry` function that handles any operation returning `Either[A, B]`, enabling its application across multiple domains;
- Kotlin handles errors using exceptions, while Scala uses typed monads like `Option` and `Either` that can be part of function return types. These monads enforce localized and explicit error handling, enhancing robustness and reducing the risk of missing errors;
- Scala uses pure functions for transaction processing logic, postponing side effects, which allows seamless support for atomic transaction batch processing. On the other hand, Kotlin integrates side effects directly into the transaction logic, making it more challenging to achieve the same level of atomicity in transaction batch processing.

References

[1] Charles Scalfani. *Why Functional Programming Should Be The Future of Software Development*. url: https://spectrum.ieee.org/functional-programming.

For further information, see https://linux.ime.usp.br/~brizamel/mac0499/ or submit an email to brizamel.dias@usp.br