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

Briza Mel Dias de Sousa

Capstone project
presented to MAC0499

Supervisors:
Prof. Dr. Alfredo Goldman
M.Sc. Renato Cordeiro Ferreira

São Paulo, December 2024

Contents

1 Introduction **2**

1.1 Objective . . . 2

1.2 Research Questions . . . 2

1.3 Proposal Structure . . . 4

2 Methodology **5**

2.1 Proof of Concept . . . 5

2.2 Qualitative Analysis . . . 6

2.3 Quantitative Analysis . . . 7

2.4 Analysis Discussion . . . 7

2.5 Threats to Validity . . . 7

2.5.1 Programming languages . . . 7

2.5.2 Qualitative analysis . . . 7

3 Programming Paradigms Concepts **8**

3.1 Object-Oriented Programming . . . 8

3.2 Functional Programming . . . 9

4 Requirements **10**

4.1 Wallet Management . . . 10

4.2 Investment Customization . . . 11

4.3 Transaction Lifecycle Management . . . 11

4.4 Other minor requirements . . . 12

5 System **13**

5.1 Models . . . 13

5.1.1 Wallet . . . 13

5.1.2 Subwallet . . . 14

5.1.3 Transaction . . . 15

5.1.4 Journal Entry . . . 16

5.1.5 Investment Policy . . . 17

5.2 Services . . . 18

6 Qualitative Analysis **20**

6.1 Extensibility . . . 20

6.2 Reusability . . . 24

6.3 Error handling and propagation 29
6.4 The Batch Processing With Atomicity Requirement 31
6.5 Testability 50
6.6 Readability 57

7 Quantitative Analysis **63**
7.1 Results 63
7.1.1 Question #1 63
7.1.2 Question #2 65
7.1.3 Question #3 66
7.1.4 Question #4 67

8 Discussion **69**
8.1 Extensibility 69
8.2 Reusability 69
8.2.1 Error Handling and Propagation 70
8.2.2 Testability 70
8.2.3 Readability 70

9 Conclusion **72**
9.1 Main results 72
9.2 Future work 73

A Survey **74**
A.1 Questionnaire 74
A.1.1 Question #1 74
A.1.2 Question #2 76
A.1.3 Question #3 79
A.1.4 Question #4 81
A.2 Background Questions 85

Bibliography **86**

Abstract

After decades of dominance by object-oriented programming (OOP), functional programming (FP) is gaining increasing attention in the software industry (Finley). This study compares the impact of OOP and FP on the architectural characteristics of software systems. Specifically, it examines the design and implementation of a Digital Wallet system, developed in Kotlin (representing OOP) and Scala (representing FP). The comparison is made through both qualitative and quantitative analyses to explore how each paradigm influences the system's architectural characteristics. The qualitative analysis examined the implementation of specific functionalities within the paradigms. Each architectural characteristic was meticulously compared, revealing the strengths and weaknesses of each paradigm for a given functionality. The quantitative analysis applied a survey to gather feedback from developers with diverse backgrounds. Together, these approaches provided a comprehensive understanding of how OOP and FP shaped the architectural characteristics of the Digital Wallet system.

Keywords: Object-oriented Programming, Functional Programming, Software Architecture, Programming Paradigms.

Chapter 1

Introduction

After decades of dominance of object-oriented programming (OOP), the functional programming (FP) paradigm is gaining significant attention in the software industry (Finley). This shift is a response to growing complexities in software systems, the demand for better scalability, and the need for more predictable, robust, and maintainable codebases. According to Scalfani in 2022, one of the clearest indications of this shift is the increasing incorporation of functional features into mainstream programming languages.

As the software industry embraces FP concepts to address these challenges, evaluating how FP affects critical aspects such as extensibility, reusability, error handling, error propagation, testability and readability compared to OOP is essential to understand the future of software development. By examining this transition and its implications, this research aims to contribute with valuable insights into how programming paradigms influence architectural characteristics in software systems. This understanding will help developers and organizations make informed decisions about the paradigms best suited for their projects.

1.1 Objective

Compare how object-oriented programming and functional programming paradigms impact the architectural characteristics of systems.

1.2 Research Questions

RQ1: What are the architectural characteristics for a system that can be used to compare the functional and object-oriented paradigms?

According to Mark Richards (2020), architectural characteristics are all the things the software must do that is not directly related to the domain functionality. The first question aims to establish a well-defined set of architectural characteristics to guide the comparison between object-oriented and functional programming paradigms. These architectural characteristics should encompass functionalities that highlight the differences between the paradigms, facilitating the evaluation of their strengths and limitations in various aspects of software development. This approach will enable a focused comparison of how each paradigm approaches key areas of development and emphasize their unique characteristics.

RQ2: How do functional and object-oriented paradigms impact different architectural characteristics of a system?

The second question aims to understand the differences that arise when a system is developed using a functional paradigm compared to an object-oriented paradigm. This analysis focuses on observing how each paradigm influences critical aspects of software development, providing insights into which approach better supports it.

sRQ1: How do functional and object-oriented paradigms impact error handling, error propagation, and the overall robustness of a system?

The first subquestion explores how each paradigm handles and propagates errors, impacting the overall robustness of the system. Object-oriented and functional paradigms embody distinct philosophies in error representation and management, with each approach offering unique mechanisms that may enhance or challenge system reliability. This analysis aims to determine how these differences influence error-handling effectiveness and whether one paradigm naturally contributes to building a more reliable system.

sRQ2: How do functional and object-oriented paradigms impact the testability of a system?

The second subquestion aims to understand how functional and object-oriented paradigms might facilitate the testing process, examining whether one paradigm results in more comprehensive test coverage or impacts the complexity of the tests themselves.

sRQ3: How do functional and object-oriented paradigms impact the legibility of a system?

The third subquestion seeks to answer how functional and object-oriented paradigms influence the readability and clarity of the system. The distinct structures and conventions of each paradigm may enhance or hinder the ease with which developers can understand and reason about the code.

sRQ4: How do functional and object-oriented paradigms impact the maintainability of a system?

The fourth subquestion explores how functional and object-oriented paradigms support the long-term maintainability of a system by enabling the development of modular, reusable, and easily extendable code.

1.3 Proposal Structure

The remaining of this proposal is structured as follows. Chapter 2 outlines the methodology employed to conduct this research. Chapter 3 introduces essential paradigm concepts used throughout the work. Chapter 4 defines the requirements that the study object must fulfill. Chapter 5 describes a language- and paradigm-agnostic architecture for the study object. Chapter 6 presents the author's comparative analysis of the paradigms. Chapter 7 supplements the qualitative analysis with a quantitative analysis. Finally, Chapter 8 discusses the outcomes of the research and their implications.

Chapter 2

Methodology

This chapter details the methodology employed to bring the proposed work to reality, outlining how the research questions are going to be answered. A Digital Wallet System will serve as the primary study object, enabling detailed comparison of object-oriented and functional paradigms. Through this approach, the research will examine how each paradigm approaches key architectural characteristics, providing insight into their benefits and drawbacks.

2.1 Proof of Concept

The initial step in addressing the research questions is to build a system that will serve as the foundation of this research. A Digital Wallet System was chosen as the domain for this system due to its multifaceted functionality, which includes transaction management, balance tracking, batch processing, and support for multiple wallet and transaction types. These core operations demand intricate data handling, state management, and consistency, each of which can be approached differently within object-oriented and functional programming paradigms. By implementing a system with these features, the research allows for a meticulous exploration of how each paradigm addresses different types of architectural characteristics, including:

- **Extensibility**: The Digital Wallet System could be incremented with several new functionalities, which might require the extension of the existing models, such as wallets and transactions. The research also explores how object-oriented and functional paradigms contribute to building a system that is easy to expand.
- **Reusability**: Reusability is a fundamental goal in software design, offering flexibility and reducing the need for further redesign. However, as Erich Gamma observes in Design Patterns: Elements of Reusable Object-Oriented Software, achieving reusability is challenging as it requires the definition of relationships that are both specific to current needs and adaptable for future requirements. This characteristic of software architecture will also be examined in the context of the Digital Wallet System, exploring how object-oriented and functional paradigms promote code reusability.
- **Error handling and propagation**: The Digital Wallet System supports end-to-end transaction processing, from creation to settlement. Throughout this lifecycle, transactions may

encounter validation or unexpected errors that need to be identified and managed to maintain system consistency. This aspect of the system presents an opportunity to explore how the object-oriented and functional paradigms handle error detection, management, and propagation, demonstrating how they influence the robustness and reliability of the Digital Wallet System.

- **Testability**: The Digital Wallet System involves several components that must be tested to ensure the correctness of its functionalities, such as transaction processing, wallet management, and record-keeping. This provides an opportunity to explore the ease of testing code written in object-oriented versus functional paradigms.
- **Readability**: he Digital Wallet System supports multiple wallet and transaction types, each governed by different rules. Understanding how these components interact can be challenging for the reader. This offers an opportunity to explore which paradigm better expresses the business logic, making the system's flow and interactions more transparent and easier to understand.

To make the comparison of paradigms possible, the Digital Wallet System will be developed twice with the aim of producing a study object for the object-oriented paradigm and another study object for the functional paradigm.

The object-oriented version of the Digital Wallet System will be based on Kotlin programming language. Kotlin is a modern language that offers strong support for classic object-oriented principles such as inheritance, encapsulation, polymorphism, and abstraction. The developer can define classes in a straightforward manner, enabling the structuring of data and behavior through objects.

The functional version of the Digital Wallet System will be based on Scala programming language. Scala is a modern language that supports many functional programming constructs such as higher-order functions, immutable data structures, pattern matching, and monads. It provides the ability of developing with emphasis on immutability and pure functional constructs, aligning with the core principles of functional programming.

The initial step in creating both study objects is to establish a set of functional requirements that will guide the development of both the functional and object-oriented versions of the Digital Wallet System. These functional requirements should be paradigm and language agnostic, ensuring they are defined independently of any specific characteristics or limitations of the chosen paradigms or languages.

2.2 Qualitative Analysis

Once the Digital Wallet System is implemented in both Kotlin and Scala, a qualitative comparison will be done to assess how each version addresses the challenges a Digital Wallet System is designed to solve.

To conduct this qualitative analysis, code snippets from both the Kotlin and Scala implementations that perform the same tasks will be carefully selected to form the basis of the qualitative analysis. The analysis will evaluate the extensibility, reusability, error handling, error propagation, testability and readability of the Digital Wallet System.

2.3 Quantitative Analysis

The quantitative analysis will be conducted through survey research. This survey will consist of five questions, each presenting code snippets from the Kotlin and Scala implementations of the Digital Wallet System that perform the same task. Respondents will be asked to evaluate each snippet based on the following criteria: extensibility, reusability, error handling, error propagation, testability and readability. A Likert scale format will be employed for these evaluations, providing a standardized framework for assessing each criterion.

Additionally, the survey will collect information about the participants' backgrounds to provide context for their evaluations. These background questions can be found in Appendix A.

The background questions provide essential context for interpreting the survey responses. Participants' experiences, familiarity with programming paradigms, and expertise in Kotlin or Scala can significantly influence how they understand and evaluate the code snippets.

2.4 Analysis Discussion

The analysis discussion step involves combining the results of both the qualitative and quantitative analyses to discuss their findings. This discussion will identify whether one paradigm performed better in specific criteria, such as readability or testability, and highlight any notable differences between the Kotlin and Scala implementations. The goal is to determine if one paradigm offered significant advantages over the other, answering the research question RQ2.

2.5 Threats to Validity

Assessing threats to validity is essential to ensure the solidity of this research and to acknowledge potential limitations that might influence the interpretation of the results.

2.5.1 Programming languages

The evaluation of programming paradigms in this research may be influenced by the choice of programming languages representing each paradigm. Kotlin and Scala were chosen as modern, multiplatform languages that run on the Java Virtual Machine (JVM), ensuring a level of comparability.

Despite Kotlin supports functional programming features and Scala supports object-oriented design, the research intentionally limits Kotlin to object-oriented programming principles and Scala to functional programming principles. This restriction aims to focus the comparison on the paradigms themselves rather than on language-specific capabilities.

2.5.2 Qualitative analysis

The qualitative analysis conducted in this research may reflect biases influenced by the author's background and expertise. To address this, a mixed-methods approach will be employed, incorporating quantitative analysis to complement the qualitative findings. Feedback from professionals with diverse backgrounds will be collected, ensuring a wider range of perspectives and reducing the potential impact of individual biases.

Chapter 3

Programming Paradigms Concepts

Understanding the fundamental principles of Object-Oriented Programming (OOP) and Functional Programming (FP) is essential for any meaningful comparison between these paradigms. Both paradigms have distinct philosophies, strengths, and limitations that significantly influence how software systems approach architectural characteristics.

This chapter introduces OOP and FP concepts that are essential to understand the comparisons discussed in this study.

3.1 Object-Oriented Programming

According to Martin (2017), object-oriented programming is the proper admixture of encapsulation, inheritance and polymorphism. Any OOP programming language must support these three characteristics.

Encapsulation

Effective encapsulation of data and functions ensures that a clear boundary is established around a cohesive set of data and behaviors. Within this boundary, data is hidden and protected from direct external access, while only specific functions are exposed to interact with the data. This approach is commonly seen in object-oriented programming through private data members and public methods in classes.

Inheritance

Inheritance is the process of reusing and extending a group of variables and functions defined in one scope within another. In object-oriented programming, this often means that a new class (called a subclass) can inherit properties and behaviors (variables and methods) from an existing class (called a superclass). This allows the subclass to reuse, override, or expand existing functionality without redefining it entirely.

Polymorphism

Polymorphism is the ability of a method to behave differently based on the context in which it is used. Polymorphism unblocks **dependency inversion** by allowing high-level modules to depend on abstractions rather than concrete implementations. It decouples implementation details from the modules that depend on them through interfaces, allowing dependencies to be reversed.

3.2 Functional Programming

According to Paul Chiusano (2014), functional programming is based on the premise of writing programs using only **pure functions**.

Pure functions

Pure functions are functions without **side effects**. A function has side effects if it does something other than returning a result, such as modifying a variable, throwing an exception or printing to the console.

High-order functions

In functional programming, functions are values. They can be assigned to variables, stored in data structures, and passed as arguments to functions.

When writing purely functional programs, it is often useful to write a function that accepts other functions as arguments. This is called a higher-order function (HOF).

Chapter 4

Requirements

The Digital Wallet System is the object of study used in this work to establish the comparison between the object-oriented and the functional programming paradigms. Although it models a real-world problem, the Digital Wallet System was not developed for production, but rather was designed to provide examples of functionalities that highlight the differences between the paradigms. Therefore, the project ignores some typical concerns for this type of system, such as data security and auditability.

The functional requirements outlined for the Digital Wallet System are listed below.

4.1 Wallet Management

The system should support wallet management by providing each user with three types of wallets:

- Real Money Wallet, representing a real-world checking account owned by the customer. This wallet is the entry and exit point of the Digital Wallet System, where the customer can deposit from or withdraw to an external bank account. It is modeled to be a temporary allocation of the customer's funds until they are moved to another wallet;

- Investment wallet, representing a portfolio of investment options where customers can allocate their funds. Funds are invested by transferring funds from the Real Money Wallet to the Investment Wallet, and are liquidated by transferring from the Investment Wallet to the Real Money Wallet. Investments and liquidations can be initiated at any time, but will only settle on the next business day;

- Emergency Funds Wallet, representing a deposit insurance where customers can allocate funds without taking the risks of the Investment Wallet. Funds are deposited into and withdrawn from the Emergency Funds Wallet through instant transfers to and from the Real Money Wallet.

4.2 Investment Customization

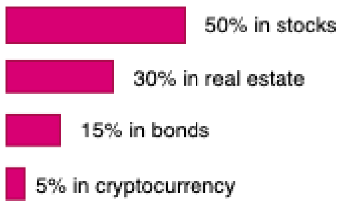


Figure 4.1: *The figure illustrates an* INVESTMENTPOLICY *that allocates a given amount as follows: 50% to stocks, 30% to real estate, 15% to bonds, and 5% to cryptocurrency.*

The system should support investment and liquidation customization by letting customers define the participation of each investment option under an INVESTMENT WALLET. The participation should be measured as a percentage of the total invested, and the customer should have real-time control over it. This requirement includes the capacity of setting the percentage to zero for undesired investments, or even selecting a single investment.

4.3 Transaction Lifecycle Management

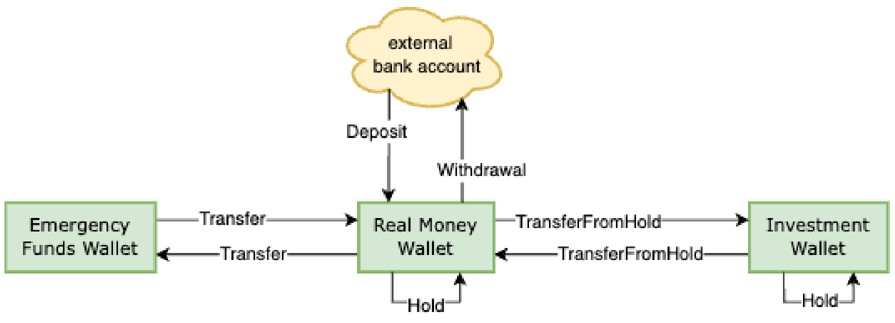


Figure 4.2: *DEPOSIT and WITHDRAWAL represent an interaction of the REAL MONEY WALLET with the external world. TRANSFER is intended to instantly transfer funds between REAL MONEY WALLET and EMERGENCY FUNDS WALLET. HOLD reserves funds within REAL MONEY WALLET or INVESTMENT WALLET for future use. TRANSFERFROMHOLD instantly transfer the reserved funds.*

The system should support the creation, validation and execution of transactions between wallets. Transactions represent a money movement inside the Digital Wallet System which might succeed or fail. Providing customers with all the functionalities they need embraces the following types of transactions:

- DEPOSIT, which represents a deposit to the REAL MONEY WALLET, modeling the inflow of funds into the system;
- WITHDRAWAL, which represents a withdrawal from the REAL MONEY WALLET, modeling the outflow of funds from the system;
- TRANSFER, representing an instant transfer to or from the REAL MONEY WALLET. While the Digital Wallet System is not required to handle transfers directly with real-world bank accounts, it should integrate with a third-party API to enable this functionality;

- HOLD, representing a reserved amount within the wallet, crucial for non-instant money movements. This transaction blocks a specified portion of funds for a future purpose, ensuring availability when the associated movement settles;
- TRANSFER FROM HOLD, representing an instant release and transfer of funds previously reserved in a HOLD transaction. This transaction finalizes the settlement of a HOLD transaction, completing the intended fund movement.

Every transaction should be validated prior to execution. DEPOSIT and WITHDRAWAL transactions are restricted to REAL MONEY WALLET. TRANSFER transactions are permissible only between REAL MONEY WALLET and EMERGENCY FUNDS WALLET. A HOLD can be placed exclusively on REAL MONEY WALLET and INVESTMENT WALLET, while TRANSFERFROMHOLD transactions are limited to be executed between REAL MONEY WALLET and INVESTMENT WALLET. Additionally, all transactions — except for DEPOSIT — require a balance check to ensure sufficient funds are available to complete the operation.

The system is also expected to manage transaction failures as part of the transaction lifecycle. Transactions failing validation should be marked as permanently failed, while those that fail during execution should be eligible for retry. In general, validation should capture expected, unrecoverable errors, such as insufficient funds. Execution errors, though unlikely after successful validation, may still occur due to internal issues or third-party API instabilities. In such cases, the system should provide a mechanism to retry the transaction.

4.4 Other minor requirements

Financial Tracking

The system must provide means of tracking all financial transactions to maintain accurate account balances and ensure data integrity.

Investment and Liquidation settlement

The system should support a mechanism to settle all investment and liquidation requests initiated on the previous business day.

Batch Processing of Transactions

The system should support batch processing of transactions with atomicity. This means that in any given batch, all transactions must either complete successfully or, if any transaction fails, the entire batch must be rolled back. This ensures data integrity and prevents partial updates that could disrupt the financial record-keeping system.

Chapter 5

System

The architecture of the Digital Wallet System was designed to embrace all proposed requirements, providing a high-level overview of the system's components and their interactions.

This chapter presents a description that focuses on the structure and communication flows between components, independent of specific programming languages or paradigms.

5.1 Models

The requirements introduced some key concepts of the Digital Wallet System, such as wallets and the ability of performing money movements through transactions. This section presents core data structures representing these business concepts within the Digital Wallet System. They form the foundation of the system's functionality, modeling entities such as Wallet, Transaction, InvestmentPolicy and JournalEntry.

5.1.1 Wallet

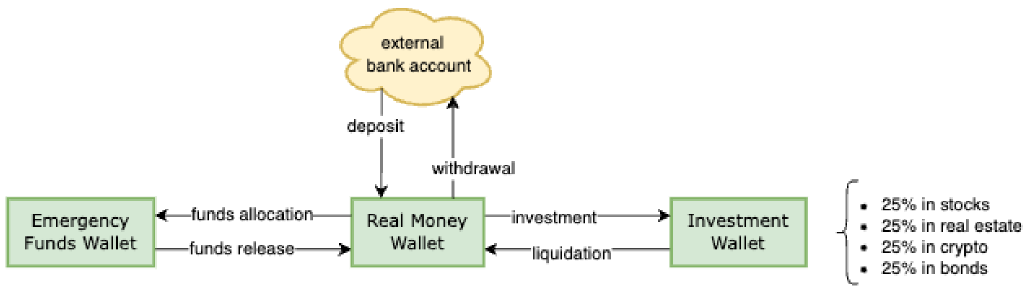


Figure 5.1: *The Real Money Wallet serves as the primary interface with the external world, enabling deposits and withdrawals of funds. It can allocate funds to the Emergency Funds Wallet for insurance purposes or to the Investment Wallet for investment opportunities. The Emergency Funds Wallet can transfer funds back to the Real Money Wallet when needed. Similarly, the Investment Wallet may liquidate investments, transferring the funds back to the Real Money Wallet.*

The Wallet model serves as an account structure where customer funds are held and managed. It functions as the starting or ending point for all financial transactions, allowing money to be deposited, withdrawn, or transferred to other destinations. The system provides three types of wallets for each customer:

- Real Money Wallet: This wallet represents a real-world checking account which can be deposited or withdrawn. It is the only wallet type that directly interacts with external bank accounts, serving as a channel for funds entering and leaving the system. Additionally, the Real Money Wallet can transfer funds to the Investment Wallet for investment purposes and to the Emergency Funds Wallet to allocate funds intended for preservation;

- Investment Wallet: This wallet is an abstraction of the customer's investment portfolio. The Investment Wallet holds funds allocated in up to four types of investments: stocks, bonds, real estate, and cryptocurrency. The total balance of the Investment Wallet is calculated as the sum of the funds distributed among these investment options, providing an abstracted view of the customer's investment assets. For liquidation purposes, the Investment Wallet can transfer funds back to the Real Money Wallet, enabling access to liquidity when needed. However, no interactions with the Emergency Funds Wallet are allowed, ensuring that designated emergency reserves remain separate from investment assets;

- `Emergency Funds Wallet`: This wallet is designed to hold funds intended to remain secure and free from investment risks. The Emergency Funds Wallet can transfer to the Real Money Wallet as needed, enabling quick access to liquidity in case of urgent situations.

While transactions dictated the flow of funds between wallets, the wallets themselves are unaware of the transactions' existence. Their role is solely to model and hold customer funds within the system; they are not responsible for determining the validity of incoming or outgoing transactions. This separation of concern ensures that wallets focus on balance management without handling transaction logic.

5.1.2 Subwallet

The Subwallet concept is introduced to represent the distribution of funds within the same wallet.

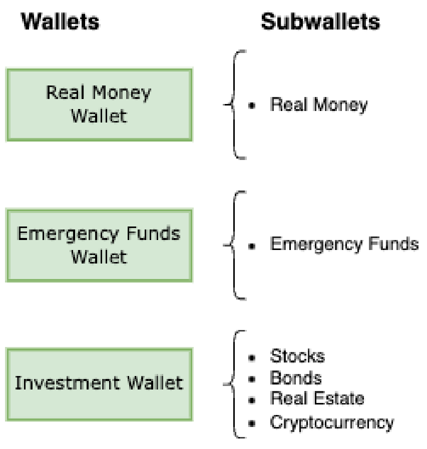


Figure 5.2: *Wallet types and corresponding Subwallet types.*

The Real Money Wallet has only one Subwallet where funds can be allocated: the Real Money Subwallet. In practice, having a single Subwallet means that, for record-keeping purposes, funds in the Real Money Wallet are not divided in any way. Similarly, the Emergency

Funds Wallet consists solely of the Emergency Funds Subwallet. The relevance of the Subwallet becomes evident in the Investment Wallet, where funds are allocated across various investment options. Introducing a new investment option involves adding a corresponding Subwallet within the Investment Wallet.

5.1.3 Transaction

A Transaction represents an attempt to change the state of a Wallet. Each transaction specifies an **amount**, as well as **originator** and **beneficiary** entities, which can either be a Wallet (and its designated Subwallet) or an external entity. Each transaction has also a **status** representing its current phase in processing. Possible statuses include Processing, Failed, Transient Error, and Completed.

Transaction status

Transactions are created with the **Processing** status, indicating that they have not yet gone through validation or execution. The **Failed** status is a terminal state, representing a permanent failure with no possibility for retry. Similarly, **Completed** is a terminal status, meaning that all validations and execution steps succeeded, the transaction has settled and the involved Wallet(s) was updated. The **Transient Error** status indicates a recoverable error occurred during transaction processing, such as a failure in a third-party API call. Transactions with this status are eligible for retry.

Transaction validation

Transactions must be validated to ensure they adhere to the internal rules of the Digital Wallet System, as some transactions may be faulty. For instance, specific transaction types can only be initiated on a Real Money Wallet, but not on an Investment Wallet. Additionally, transactions must respect the restriction that prevents any interaction between the Investment Wallet and the Emergency Funds Wallet, so these wallets should not interact or even be aware of each other's existence. Some transactions are also subject to a balance validation to verify that the initiating wallet has sufficient funds for the transfer, ensuring it will not be overdrafted.

Transaction types

A Transaction can be classified as a Deposit, Withdrawal, Hold, Transfer, or TransferFromHold. Each transaction type represents a standard operation that modifies the state of a Wallet.

A Deposit transaction represents an increase in a Real Money Wallet's balance, resulting from an external source adding funds to the system. Deposits do not require balance validation, as they only involve incoming funds and do not reduce the wallet's balance. Conversely, a Withdrawal transaction decreases the Real Money Wallet balance by transferring funds out of the system to an external recipient. Withdrawals require balance validation to ensure there are sufficient funds to cover the transaction; if the balance is less than the requested amount, the transaction will fail permanently.

Completing a Deposit or Withdrawal transaction authorizes funds to enter or leave the system. These transaction types are exclusive to the Real Money Wallet, as it is the only wallet that interacts with the external world.

A Transfer transaction moves funds from a source wallet to a target wallet. Transfers are allowed between the customer's Real Money Wallet and Emergency Funds Wallet, and balance validation is required to confirm that the source wallet has sufficient funds to continue processing the transfer. Once a Transfer transaction passes all validations, it proceeds to execution, initiating the actual transfer of funds within the system. The Digital Wallet System delegates this transfer operation to a third-party API, which instantly moves funds between the Real Money Wallet and the Emergency Funds Wallet as needed. This transaction type is designed to provide customers with quick access to their insured funds when necessary.

While transfers are permanently marked as failed if they do not pass validation, they are eligible for retry if an error occurs after successfully completing validation. This flexibility allows the system to recover from issues such as third-party API instabilities, preventing customers from experiencing errors directly or even needing to initiate a new transfer.

Unlike Deposit, Withdrawal, and Transfer transactions, a Hold transaction does not involve moving money from one place to another; it simply reserves a specified amount of funds within the same Wallet. Reserved funds can either be released, making them available again, or transferred to another Wallet through a TransferFromHold transaction. A TransferFromHold transaction allows movement of reserved funds, which do not appear in the available balance and, therefore, are not accessible by standard Transfer transactions.

Hold transactions require balance validation to ensure that the amount reserved does not exceed the current available balance. Although this validation occurs when placing the Hold, TransferFromHold transactions also check that enough reserved funds are available to cover the transfer amount. This is necessary because there is not a strict one-to-one relationship between Hold transactions and TransferFromHold transactions; any portion of the reserved funds can be transferred as long as it does not exceed the total amount held in reserve.

5.1.4 Journal Entry

A JournalEntry serve to document the effects of a Transaction on a Wallet. They track changes in the Wallet's balance and provide a detailed history of all financial activities. By listing all journal entries linked to a Wallet, it is possible to ascertain its current state accurately.

Journal entries are always created in pairs to represent the origin and destination of funds being moved. To accurately reflect changes in a Wallet's state, a JournalEntry must include the following:

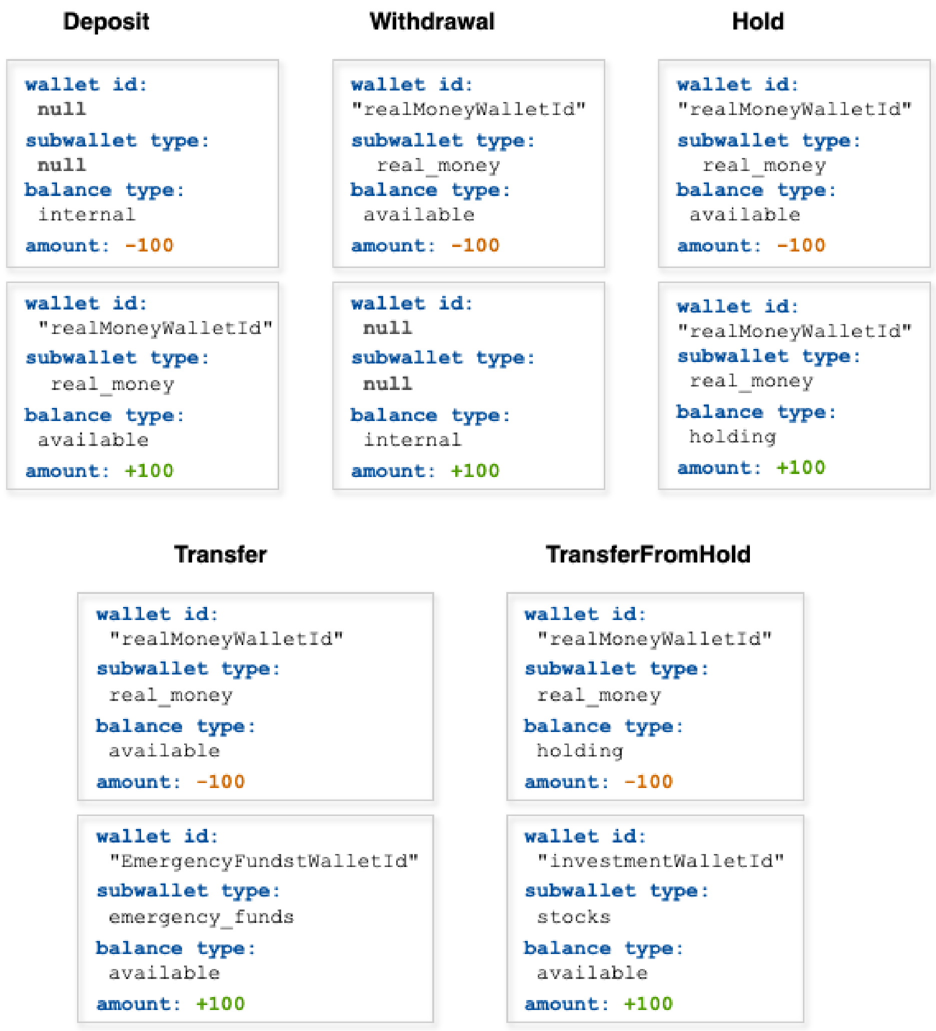
- **Source or Target Wallet ID**: This identifies the Wallet involved in the transaction. It can be absent to indicate interactions with the external world for Deposit and Withdrawal transactions.

- **Subwallet ID**: This specifies the Subwallet under the main Wallet. It may also be absent to represent the external world.

- **Amount**: A positive amount indicates that money is being added to the Subwallet, while a negative amount signifies a deduction from the Subwallet.

- **Balance Type**: This can be Internal, Available, or Holding. **Internal** represents funds being sent to or received from the external world. **Available** indicates funds being added to or deducted from the available balance of a Subwallet. **Holding** indicates funds being added to or deducted from the reserved balance of a Subwallet.

The ledger gathers all journal entries created by the Digital Wallet System. Each completed transaction results in a pair of journal entries being recorded in the ledger. The figure below displays the corresponding journal entries for each transaction type.



Deposit

wallet id: null
subwallet type: null
balance type: internal
amount: -100

wallet id: "realMoneyWalletId"
subwallet type: real_money
balance type: available
amount: +100

Withdrawal

wallet id: "realMoneyWalletId"
subwallet type: real_money
balance type: available
amount: -100

wallet id: null
subwallet type: null
balance type: internal
amount: +100

Hold

wallet id: "realMoneyWalletId"
subwallet type: real_money
balance type: available
amount: -100

wallet id: "realMoneyWalletId"
subwallet type: real_money
balance type: holding
amount: +100

Transfer

wallet id: "realMoneyWalletId"
subwallet type: real_money
balance type: available
amount: -100

wallet id: "EmergencyFundstWalletId"
subwallet type: emergency_funds
balance type: available
amount: +100

TransferFromHold

wallet id: "realMoneyWalletId"
subwallet type: real_money
balance type: holding
amount: -100

wallet id: "investmentWalletId"
subwallet type: stocks
balance type: available
amount: +100

Figure 5.3: *Journal entries posted for each transaction type.*

5.1.5 Investment Policy

An InvestmentPolicy defines the allocation of an investment portfolio by associating each Subwallet with a specific percentage, representing its share of the total portfolio. Only subwallets within the Investment Wallet are permitted to be included in an InvestmentPolicy.

5.2 Services

The Digital Wallet System relies on a set of core services that operate collectively to serve the functionalities described by the requirements. Each service plays a distinct role, supporting processes such as wallets management, transactions processing, record-keeping, integrations with third-party APIs and investments and liquidations management.

The LedgerService is responsible for intermediating the interactions between other services and the Digital Wallet System's ledger. It essentially provides two functionalities:

- **Posting journal entries**
- **Querying the ledger for balances**: Aggregates JournalEntries to compute the balance of a specific Wallet.

The TransactionsService provides functionalities to fulfill each step of a transaction lifecycle. It creates, validates and processes transactions, encapsulating the business logic for each transaction type. This service also guarantees that the ledger is accurately updated upon transaction completion, maintaining consistency across the system.

Although the TransactionsService is responsible for managing different transaction types, it remains agnostic to their specific purposes. Its role is to ensure that all transactions are processed correctly and that the ledger remains consistent.

The WalletsService is responsible for managing and initiating transactions within wallets. It encapsulates the business logic needed to translate money movements initiated by the customer into a series of transactions that fulfill these requests. The types of requests handled by the WalletsService include:

- **Deposit Request**: Deposits funds into the Real Money Wallet.
- **Withdraw Request**: Withdraw funds from the Real Money Wallet.
- **Emergency Allocation Request**: Withdraw funds from the Real Money Wallet.
- **Emergency Release Request**: Transfers funds from the Emergency Funds Wallet back to the Real Money Wallet.
- **Investment Request**: Allocates funds from the Real Money Wallet to the Investment Wallet, distributing the amount across investment options based on the InvestmentPolicy.
- **Liquidation Request**: Withdraws funds from the Investment Wallet to the Real Money Wallet, proportionally liquidating investments according to the InvestmentPolicy.

The InvestmentService is dedicated to handling investment and liquidation requests, which involve multiple steps and do not settle instantly like other transaction types. This service knows how to effectively use an InvestmentPolicy to initiate and manage the transactions required to complete an investment or liquidation.

The diagram below illustrates the communication flow between services:

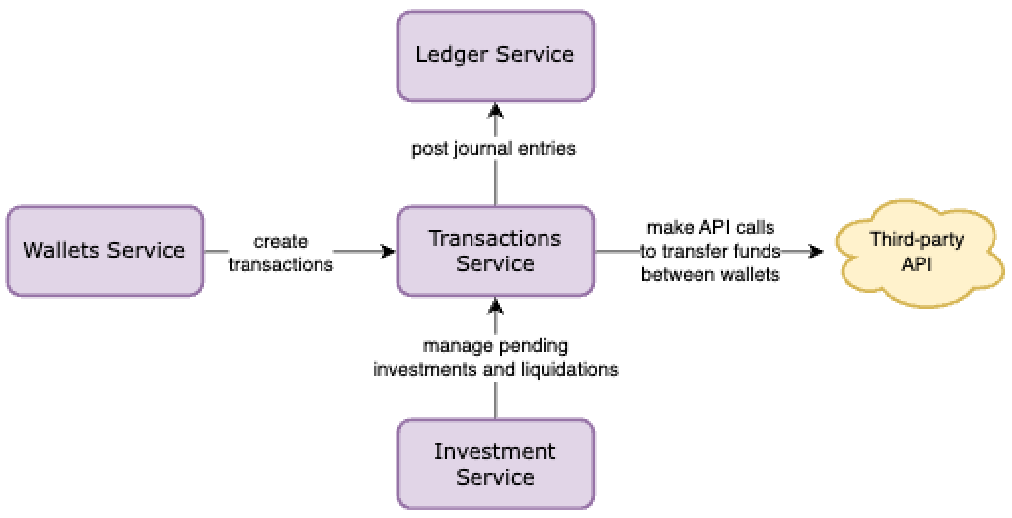


Figure 5.4: *The services that compose the Digital Wallet System.*

Chapter 6

Qualitative Analysis

This chapter presents the qualitative analysis of the Digital Wallet System for both Kotlin and Scala versions. The qualitative analysis compares code snippets from each version of the Digital Wallet System that perform the same task, evaluating them according to the following criteria:

- **Readability:** How easy it is to understand its intended behavior.
- **Maintainability:** How easy it is to change its current behavior.
- **Extensibility:** How easy it is to add new behavior.
- **Testability:** How easy it is to test it.
- **Reusability:** How easy it is to apply it in a new use case.
- **Error Handling:** How easy it is to understand errors handled by it.
- **Error Propagation:** How easy it is to understand errors propagated by it.

This chapter seeks to reveal how the functional and object-oriented paradigms perform with respect to the criteria listed above.

6.1 Extensibility

To analyze the extensibility, this section examines how the Kotlin and Scala versions of the Digital Wallet System model the concept of a Wallet.

```

// Wallet superclass
abstract class Wallet(
    val id: String,
    val customerId: String,
    val policyId: String,
) {
    abstract fun getAvailableBalance(ledgerService: LedgerService) : BigDecimal
}

// RealMoneyWallet subclass
```

```kotlin
class RealMoneyWallet(
    id: String,
    customerId: String,
    policyId: String,
) : Wallet(id, customerId, policyId) {
    override fun getAvailableBalance(ledgerService: LedgerService): BigDecimal {
        val ledgerQuery = listOf(
            LedgerQuery(
            subwalletType = SubwalletType.REAL_MONEY,
            balanceType = BalanceType.AVAILABLE,
            )
        )

        return ledgerService.getBalance(this.id, ledgerQuery)
    }
}

// EmergencyFundsWallet subclass
class EmergencyFundsWallet(
    id: String,
    customerId: String,
    policyId: String,
    ) : Wallet(id, customerId, policyId) {
    override fun getAvailableBalance(ledgerService: LedgerService): BigDecimal {
        val ledgerQuery =  listOf(
            LedgerQuery(
            subwalletType = SubwalletType.EMERGENCY_FUND,
            balanceType = BalanceType.AVAILABLE,
            )
        )

        return ledgerService.getBalance(this.id, ledgerQuery)
    }
}

// InvestmentWallet subclass
class InvestmentWallet(
    id: String,
    customerId: String,
    policyId: String,
    ) : Wallet(id, customerId, policyId) {
    override fun getAvailableBalance(ledgerService: LedgerService): BigDecimal {
        val ledgerQueries =  listOf(
            LedgerQuery(
                subwalletType = SubwalletType.BONDS,
                balanceType = BalanceType.AVAILABLE,
            ),
            LedgerQuery(
                subwalletType = SubwalletType.STOCK,
                balanceType = BalanceType.AVAILABLE,
            ),
            LedgerQuery(
                subwalletType = SubwalletType.REAL_ESTATE,
                balanceType = BalanceType.AVAILABLE,
            ),
            LedgerQuery(
                subwalletType = SubwalletType.CRYPTOCURRENCY,
                balanceType = BalanceType.AVAILABLE,
            )
        )

        return ledgerService.getBalance(this.id, ledgerQueries)
    }
}
```

In the Kotlin implementation of the Digital Wallet System, the WALLET model is structured using object-oriented programming principles, including inheritance and polymorphism. The abstract class `Wallet` serves as a blueprint for all specific wallet types: `Real Money Wallet`, `Investment Wallet` and `Emergency Funds Wallet`.

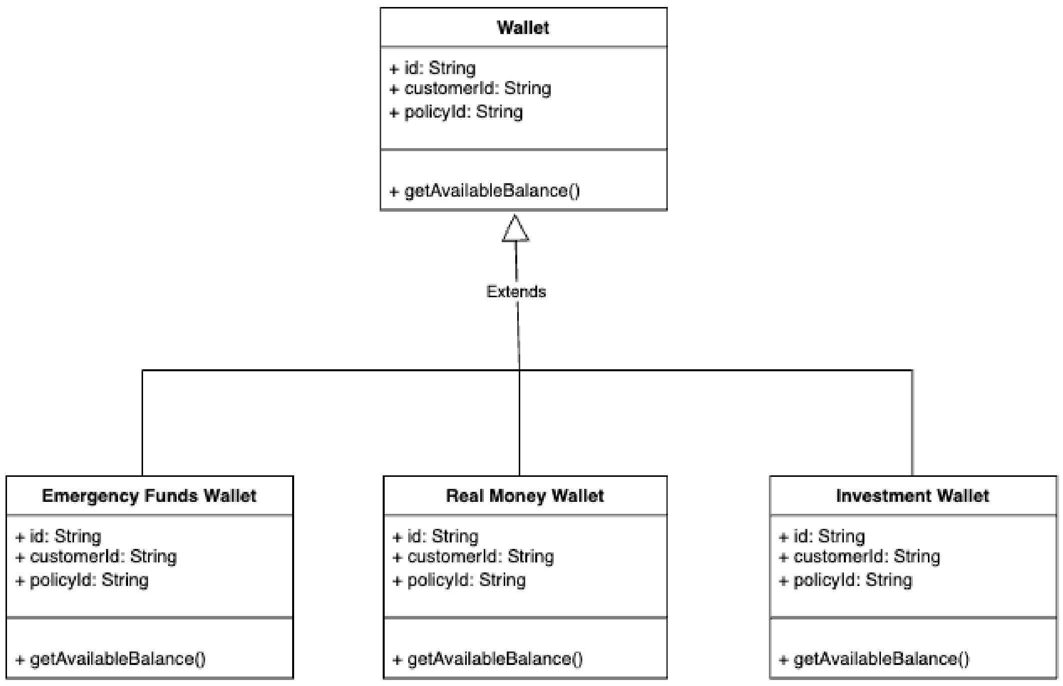


Figure 6.1: *The WALLET model depicted using Unified Modeling Language (UML).*

The `Wallet` abstract class defines the shared attributes and behavior common to all wallet types. These attributes include:

- A unique wallet identifier.
- The customer identifier to associate the wallet with its owner.
- An optional policy identifier governing the rules for money movement within the wallet.

Additionally, the class declares an abstract method, `getAvailableBalance`, which is intended to encapsulate the logic for querying balances from the ledger. By declaring `getAvailableBalance` as an abstract method, the design enforces that all subclasses must implement their specific logic for computing balances. This ensures consistency across wallet types while allowing each subclass to customize its behavior to meet its unique requirements. This approach leverages polymorphism, where the behavior of the `getAvailableBalance` method depends on the subclass being used.

The Kotlin version of the Digital Wallet System structured the WALLET model by using concepts of object-oriented programming such as inheritance and polymorphism. A future requirement demanding a new wallet type could be achieved by simply adding a new subclass that inherits from the `Wallet` abstract class. This subclass automatically gains access to the shared attributes defined in the `Wallet` class while also being required to implement the `getAvailableBalance` method, enabling it to define its specific balance-query behavior.

```scala
case class Wallet(
  val id: String,
  val customerId: String,
  val policyId: Option[String],
  val walletType: WalletType
)
```

In the Scala implementation of the Digital Wallet System, the WALLET is represented as a case class, emphasizing immutability and ease of use. Unlike the Kotlin implementation, the Scala model includes an additional field: `walletType`. This field is an enumeration (`WalletType`) that distinguishes the wallet derivations within the system.

```scala
object WalletType extends Enumeration {
  type WalletType = Value
  val RealMoney, Investment, EmergencyFunds = Value
}
```

Unlike Kotlin, which employs inheritance to represent different wallet derivations, Scala relies on an unified approach: all other fields, regardless of the `walletType`, are consolidated within a single WALLET model.

Instead of overriding the `getAvailableBalance` method in subclasses as Kotlin does, Scala defines a single `getAvailableBalance` function. This function uses pattern matching on the `walletType` field to differentiate the balance computation logic for each wallet type. For every variant of the `WalletType` enumeration, the function specifies a corresponding list of ledger queries to compute the available balance, providing behavior tailored to each wallet type without requiring separate models.

```scala
def getAvailableBalance(wallet: Wallet): BigDecimal = {
    val ledgerQuery = wallet.walletType match {
      case WalletType.RealMoney =>
        List(
          LedgerQuery(
            subwalletType = SubwalletType.RealMoney,
            balanceType = BalanceType.Available
          )
        )
      case WalletType.Investment =>
        List(
          LedgerQuery(
            subwalletType = SubwalletType.Bonds,
            balanceType = BalanceType.Available,
          ),
          LedgerQuery(
            subwalletType = SubwalletType.Stock,
            balanceType = BalanceType.Available,
          ),
          LedgerQuery(
            subwalletType = SubwalletType.RealEstate,
            balanceType = BalanceType.Available,
          ),
          LedgerQuery(
            subwalletType = SubwalletType.Cryptocurrency,
            balanceType = BalanceType.Available,
          )
        )
      case WalletType.EmergencyFunds =>
        List(
          LedgerQuery(
            subwalletType = SubwalletType.EmergencyFunds,
            balanceType = BalanceType.Available
          )
        )
    }

    ledgerService.getBalance(wallet.id, ledgerQuery)
  }
```

In Scala, pattern matching is exhaustive, meaning that any unhandled enumeration variant results in a compilation error. This feature ensures that all logic associated with `WalletType` is revisited whenever a new variant is added to the enumeration to introduce a new wallet type. Consequently, pattern matching enforces consistency across the system by guaranteeing that the behavior for all wallet types is explicitly defined, minimizing the risk of missing cases.

Both Kotlin and Scala implementations promote extensibility, enabling the system to adapt through localized code updates. In Kotlin, this is achieved by adding a new subclass to represent a wallet type and implementing the specialized `getAvailableBalance` method for that class. In Scala, extensibility is achieved by introducing a new variant to the `WalletType` enumeration.

These features also enforce the extension of existing specialized logic whenever a new wallet type is introduced. While Kotlin requires the creation of a dedicated model and method implementation for each new type, in Scala this can be achieved by simply adding a new enumeration variant and updating the existing functions that centralize the behavior of all wallet types.

6.2 Reusability

To analyze the reusability, this section examines how the Kotlin and Scala versions implement utilities commonly used throughout the Digital Wallet System.

Logging

The system leverages logs to track errors occurring at various points in the workflow. In the Kotlin version, any service can declare a `Logger` to trace its errors:

```

private val logger = Logger()
```

```
fun handleException(
    e: Exception,
    status: TransactionStatus,
    idempotencyKey: String,
): Transaction? {
    val message = e.message.toString()
    logger.error(message)

    val filter = TransactionFilter(idempotencyKey = idempotencyKey)
    val transaction = transactionsRepo.find(filter).firstOrNull()
    transaction?.updateStatus(transactionsRepo, status)

    return transaction
}
```

In the Kotlin implementation, the logging mechanism is utilized to ensure that errors are tracked on transaction failures. For instance, in the `handleException` method, the error message from the provided exception is extracted and logged using `logger.error(message)`. This guarantees that every invocation of `handleException` automatically logs the error, facilitating debugging and monitoring.

However, if exceptions unrelated to transactions need to be logged, a separate method must be created to incorporate the logging mechanism into the error-handling logic.

```

private val logger = Logger()
```

```
def maybeLogError[A, B](f: () => Either[A, B]): Either[A, B] = {
    f() match {
      case Left(e) =>
        logger.error(e.toString)
        Left(e)
      case Right(result) => Right(result)
    }
  }
```

In the Scala implementation, a higher-order function, `maybeLogError`, is introduced to encapsulate error logging. This function takes a parameter `f: () => Either[A, B]`, which represents a function that returns an `Either[A, B]`. It behaviors as follows:

- If `f` returns a `Right`, the result is directly returned.

- If `f` returns a `Left`, the error message is logged, and the error is returned.

The generality of the `maybeLogError` function dispenses the need for additional utility methods to handle errors. Any function returning an `Either[A, B]` can integrate with this higher-order function, making it versatile and reusable.

```scala
def process(
  transaction: Transaction
): Either[ProcessError, ProcessTransactionTuple] = {
  lib.maybeLogError(() => {
    for {
      validationResult <- validationService
        .validateTransaction(transaction)
        .left.map(e => ProcessError(e.message))
      processTransactionTuple <- processTransaction(transaction)
        .left.map(e => ProcessError(e.message))
    } yield processTransactionTuple
  })
}
```

In the example above, the `process` function, which handles transaction processing, uses a `for`-comprehension to chain two operations:

- `validateTransaction`, which checks the validity of the transaction.
- `processTransaction`, which performs the transaction.

Both operations return an `Either`, and their errors are mapped to a `ProcessError`. By wrapping the entire block inside `maybeLogError`, the error logging is applied automatically. If an error occurs during validation or processing, the error is logged, and the `Left` value is returned. Otherwise, the result is returned as `Right`. This approach promotes reusable error logging logic while reducing boilerplate code.

Retrying a batch of transactions

Not all transaction failures within the Digital Wallet System are considered permanent. For example, transactions that pass validation but fail during execution are eligible for retry.

```kotlin
suspend fun retryBatch(
    batchId: String,
    n: Int
) {
    val transactions = transactionsRepo.find(
        TransactionFilter(
            batchId = batchId,
            status = TransactionStatus.TRANSIENT_ERROR
        )
    )
    var allCompleted = true

    for (transaction in transactions) {
        var attempts = 0
        var success = false

        while (attempts < n && !success) {
            attempts++
            try {
                transaction.process(
                    transactionsRepo,
                    ledgerService,
                    partnerService
                )
                transaction.updateStatus(
                    transactionsRepo,
                    TransactionStatus.COMPLETED
                )
                success = true
            } catch (e: PartnerException) {
                break
            }
        }

        if (!success) {
            allCompleted = false
        }
    }

    if (allCompleted) {
        val originalTransaction = transactionsRepo.find(
            TransactionFilter(idempotencyKey = batchId)
        ).firstOrNull()

        originalTransaction?.updateStatus(
            transactionsRepo,
            TransactionStatus.COMPLETED
        )
    }
}
```

The `retryBatch` method attempts to reprocess each transaction within a batch up to `n` times. If any transaction fails while interacting with the partner API, the entire `retryBatch` operation fails. Conversely, if all transactions are successfully processed, the batch is marked as successful. The core of the retry mechanism is built around a `while` loop, leveraging a counter variable (`attempts`) to track the number of retries for each transaction and a flag variable (`success`) to indicate whether the transaction processing succeeded.

Although this implementation provides precise control over the transaction retry process, it lacks the flexibility to be reused for other operations that could benefit from a generic retry mechanism. Such a mechanism would be particularly valuable for handling calls to third-party APIs or managing asynchronous processes in general.

```
def retryBatch(batchId: String): Either[TransactionServiceError, Unit] = {
  transactionsRepo
```

```scala
    .find(
      TransactionFilter(
        batchId = Some(batchId),
        status = Some(TransactionStatus.TransientError)
      )
    )
    .traverse { t => process(t).left.map { e => ProcessError(e.message) }}
    .flatMap { tuples =>
      val failures = tuples
        .map { tuple => lib.retry(() => execute(tuple), 3) }
        .collect { case Left(error) => error }

      if (failures.nonEmpty) {
        Left(ExecutionError(s"Could not execute batch successfully."))
      } else {
        for {
          originatingTransaction <- transactionsRepo
            .find(TransactionFilter(idempotencyKey = Some(batchId)))
            .headOption
            .toRight(
              TransactionServiceInternalError(
                s"Could not find transaction with idempotency key $batchId"
              )
            )
        } yield {
          updateStatus(originatingTransaction.id, TransactionStatus.Completed)
        }
      }
    }
}
```

The Scala implementation of the `retryBatch` function delegates the retry logic to a higher-order function named `retry`:

```scala
def retry[A, B](f: () => Either[A, B], n: Int): Either[Unit, B] = {
    require(n > 0, "n must be greater than 0")

    @tailrec
    def attempt(attemptsLeft: Int): Either[Unit, B] = {
      f() match {
        case Right(result) => Right(result)
        case Left(_) if attemptsLeft > 1 => attempt(attemptsLeft - 1)
        case Left(_) => Left(())
      }
    }

    attempt(n)
}
```

The `retry` function takes a function `f: () => Either[A, B]` and an integer `n` as parameters. It attempts to execute `f` up to `n` times, early returning a `Right` containing the result if any attempt succeeds. If all `n` attempts fail, it returns a `Left`. This design encapsulates retry logic into a reusable function, providing this functionality to any function that needs to be retried.

Conclusion

These two examples demonstrate how the concept of higher-order functions enables the reuse of the same logic across multiple operations in the Scala implementation, eliminating the need to replicate similar logic for different functions. By leveraging pattern matching and recursion, the Scala implementation also achieves a more concise and expressive code structure compared to the Kotlin version, which relies on loops and mutable variables to manage the retry flow.

6.3 Error handling and propagation

To analyze error handling and propagation, this section examines how Kotlin and Scala implement the logic to create investments.

An investment allocates funds from the Real Money Wallet to the Investment Wallet. This operation consists essentially of a hold transaction that does not settle instantly.

```kotlin
class WalletsService(
    private val walletsRepo: WalletsDatabase,
    private val investmentPolicyRepo: InvestmentPolicyDatabase,
    private val transactionsService: TransactionsService,
    private val investmentService: InvestmentService,
) {
    suspend fun invest(request: InvestmentRequest) {
        val wallet =
            walletsRepo
                .find(
                    WalletFilter(
                        customerId = request.customerId,
                        type = WalletType.REAL_MONEY,
                    ),
                ).firstOrNull() ?: throw NoSuchElementException("Wallet not found")

        val processTransactionRequest =
            ProcessTransactionRequest(
                amount = request.amount,
                idempotencyKey = request.idempotencyKey,
                originatorWalletId = wallet.id,
                originatorSubwalletType = SubwalletType.REAL_MONEY,
                type = TransactionType.HOLD,
            )

        try {
            transactionsService.processTransaction(processTransactionRequest)
        } catch (e: ValidationException) {
            transactionsService
                .handleException(e, TransactionStatus.FAILED, request.idempotencyKey)

            throw InvestmentFailed(e.message.toString())
        } catch (e: PartnerException) {
            transactionsService
                .handleException(e,
                    TransactionStatus.TRANSIENT_ERROR,
                    request.idempotencyKey
                )
        }
    }
}
```

In the Kotlin implementation, the hold transaction is created through the `processTransaction` method, which can throw exceptions. These exceptions must be explicitly caught in the `invest` method to map them to a corresponding `WalletService` exception. Additionally, within the `catch` blocks, it is necessary to ensure the transaction transitions to the appropriate state.

However, if a new exception type arises in the future, developers must explicitly update the `try-catch` block adding a new catch case, risking unhandled runtime exceptions if they forget.

Additionally, as a `try-catch` block might or not handle all the error cases, reasoning about the program becomes more challenging since the code does not clearly express the potential failure paths.

```
def invest(
  request: InvestmentRequest
): Either[InvestmentFailedError, Transaction] = {
  val wallets = walletsRepo.find(
    WalletFilter(
      customerId = Some(request.customerId),
      walletType = Some(WalletType.RealMoney)
    )
  )

  wallets match {
    case List(wallet) =>
      for {
        transaction <- transactionsService
          .create(
            CreateTransactionRequest(
              amount = request.amount,
              idempotencyKey = request.idempotencyKey,
              originatorWalletId = wallet.id,
              originatorSubwalletType = SubwalletType.RealMoney,
              transactionType = TransactionType.Hold
            )
          )
          .left.map { e =>
            InvestmentFailedError(e.message)
          }

        processTransactionTuple <- transactionsService
          .process(transaction)
          .left.map { e =>
            transactionsService.updateStatus(
              transaction.id,
              TransactionStatus.Failed
            )
            InvestmentFailedError(e.message)
          }

        executedTransaction <- transactionsService
          .execute(processTransactionTuple)
          .left.map { e =>
            InvestmentFailedError(e.message)
          }
      } yield executedTransaction

    case _ =>
      Left(
        InvestmentFailedError(
          s"None or multiple wallets found for customer ${request.customerId}"
        )
      )
  }
}
```

Instead of using exceptions, Scala handles and propagates errors by treating them as values with `Either`. This approach allows errors to be explicitly modeled as part of the method's return type, making it clear to developers that a method can fail. The use of `Either` enhances type safety by enforcing that the caller must handle both success and failure cases, ensuring that no error is missed during development. This type-safe nature of `Either` eliminates the risk of unhandled runtime errors, promoting more predictable and reliable error handling within the system.

6.4 The Batch Processing With Atomicity Requirement

This section aims at examining how Kotlin and Scala implementations handled the batch processing with atomicity requirement. This requirement must be upheld during both the investment and liquidation of funds to ensure the wallets remain consistency. This section analyzes the investment operation.

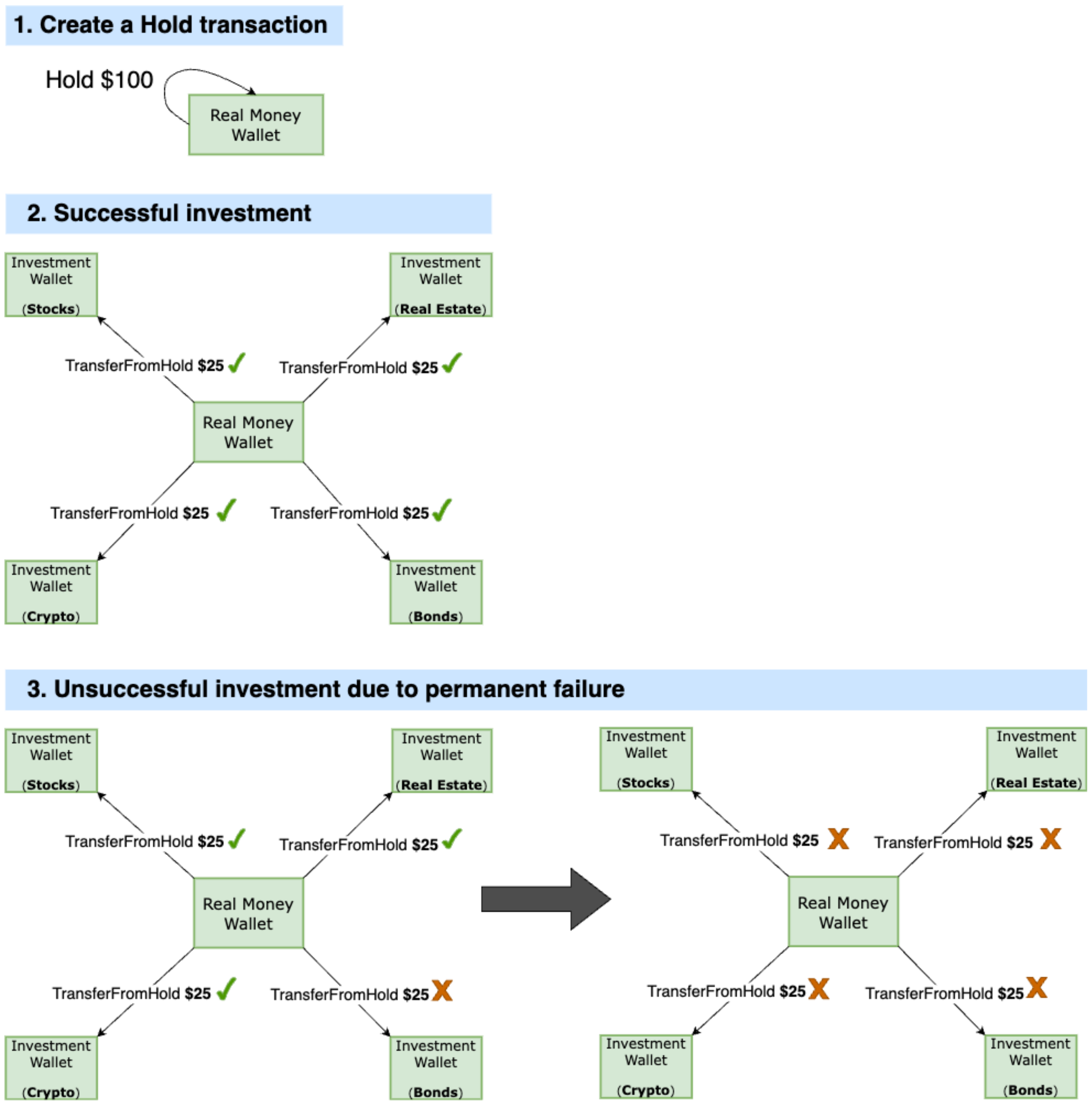

1. Create a Hold transaction
Hold $100
Real Money Wallet
2. Successful investment
Investment Wallet (Stocks)
Investment Wallet (Real Estate)
Investment Wallet (Crypto)
Investment Wallet (Bonds)
TransferFromHold $25 ✓
Real Money Wallet
3. Unsuccessful investment due to permanent failure
TransferFromHold $25 ✓
TransferFromHold $25 X

Figure 6.2: *The Digital Wallet System initiates an investment request by creating a* Hold *transaction (1). In a subsequent step, the system creates* TransferFromHold *transactions to move the funds to the* Investment Wallet *(2). If at least one transaction in the batch fails permanently, the entire batch must fail permanently (3).*

An investment request allocates funds from the Real Money Wallet to the Investment Wallet, distributing the amount across investment options based on the customer's Investment-Policy.

```kotlin
class WalletsService(
    private val walletsRepo: WalletsDatabase,
    private val investmentPolicyRepo: InvestmentPolicyDatabase,
    private val transactionsService: TransactionsService,
    private val investmentService: InvestmentService,
) {
    suspend fun invest(request: InvestmentRequest) {
        val wallet =
            walletsRepo
                .find(
                    WalletFilter(
                        customerId = request.customerId,
                        type = WalletType.REAL_MONEY,
                    ),
                ).firstOrNull() ?: throw NoSuchElementException("Wallet not found")

        val processTransactionRequest =
            ProcessTransactionRequest(
                amount = request.amount,
                idempotencyKey = request.idempotencyKey,
                originatorWalletId = wallet.id,
                originatorSubwalletType = SubwalletType.REAL_MONEY,
                type = TransactionType.HOLD,
            )

        try {
            transactionsService.processTransaction(processTransactionRequest)
        } catch (e: ValidationException) {
            transactionsService
                .handleException(e, TransactionStatus.FAILED, request.idempotencyKey)

            throw InvestmentFailed(e.message.toString())
        } catch (e: PartnerException) {
            transactionsService
                .handleException(e,
                    TransactionStatus.TRANSIENT_ERROR,
                    request.idempotencyKey
                )
        }
    }
}
```

In the Kotlin implementation, the investment request is handled by the `invest` method in the `WalletService`. This method begins by locating the customer's REAL MONEY WALLET, which serves as the funding source for the investment operation. Once the wallet is identified, the method invokes `processTransaction` on the `transactionService` to create a `Hold` transaction on the REAL MONEY WALLET, reserving the specified funds for the investment.

```kotlin
class TransactionsService(
    private val transactionsRepo: TransactionsDatabase,
    private val walletsRepo: WalletsDatabase,
    private val ledgerService: LedgerService,
    private val partnerService: PartnerService,
) {
    suspend fun processTransaction(request: ProcessTransactionRequest): Transaction {
        val transaction = transactionsRepo.insert(request)
        transaction.validate(walletsRepo, ledgerService)
        transaction.process(transactionsRepo, ledgerService, partnerService)

        return transaction
    }

  // [...]
}
```

In the Kotlin version, processing a transaction encompasses all three stages of the transaction lifecycle: creation, validation, and processing. For investment requests, creating a `Hold` transaction is necessary to reserve the specified amount in the Real Money Wallet, ensuring the funds remain allocated until they are transferred to the Investment Wallet, preventing them from being used for other purposes.

Similarly to the Wallet model, the Kotlin implementation of the Digital Wallet System leverages inheritance and polymorphism to structure a Transaction handling, enabling flexibility and reusability across different transaction types.

```kotlin
abstract class Transaction(
    val id: String,
    val batchId: String? = null,
    val amount: BigDecimal,
    val idempotencyKey: String,
    val originatorWalletId: String,
    val originatorSubwalletType: SubwalletType,
    var status: TransactionStatus,
) {
    abstract fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService
    )

    fun validateExternalTransaction() {
        if (originatorSubwalletType != SubwalletType.REAL_MONEY) {
            throw ExternalTransactionValidationException("Invalid transaction")
        }
    }

    fun validateBalance(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService
    ) {
        val wallet = walletsRepo.findById(originatorWalletId)
            ?: throw NoSuchElementException("Wallet not found")
        if (amount > wallet.getAvailableBalance(ledgerService)) {
            throw InsufficientFundsException("Insufficient funds")
        }
    }

    fun updateStatus(
        transactionsRepo: TransactionsDatabase,
        newStatus: TransactionStatus,
    ) {
        transactionsRepo.update(id, status = newStatus)
        status = newStatus
    }

    abstract suspend fun process(
        transactionsRepo: TransactionsDatabase,
        ledgerService: LedgerService,
        partnerService: PartnerService,
    )
}
```

The Hold class extends the Transaction class, implementing the validate and process methods. The validate method consolidates all the necessary validations for a Hold transaction, while the process method encapsulates the logic required to execute it.

```kotlin
class Hold(
    id: String,
    batchId: String? = null,
    amount: BigDecimal,
    idempotencyKey: String,
    originatorWalletId: String,
    originatorSubwalletType: SubwalletType,
    status: TransactionStatus,
) : Transaction(
        id,
        batchId,
        amount,
        idempotencyKey,
        originatorWalletId,
        originatorSubwalletType,
        status,
    ) {
    override fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        if (this.originatorSubwalletType !in listOf(
        SubwalletType.REAL_MONEY, SubwalletType.STOCKS,
        SubwalletType.REAL_ESTATE, SubwalletType.BONDS,
        SubwalletType.CRYPTOCURRENCY)) {
            throw HoldNotAllowed("Hold not allowed")
        }

        validateBalance(walletsRepo, ledgerService)
    }

    override suspend fun process(
        transactionsRepo: TransactionsDatabase,
        ledgerService: LedgerService,
        partnerService: PartnerService,
    ) {
        val journalEntries =
            listOf(
                CreateJournalEntry(
                    walletId = this.originatorWalletId,
                    subwalletType = this.originatorSubwalletType,
                    balanceType = BalanceType.AVAILABLE,
                    amount = -this.amount,
                ),
                CreateJournalEntry(
                    walletId = this.originatorWalletId,
                    subwalletType = this.originatorSubwalletType,
                    balanceType = BalanceType.HOLDING,
                    amount = this.amount,
                ),
            )

        ledgerService.postJournalEntries(journalEntries)

        this.updateStatus(transactionsRepo, TransactionStatus.PROCESSING)
    }
}
```

The validation of a `Hold` transaction includes the following steps:

- Verifying that the SUBWALLET is one of the following: `REAL_MONEY`, `STOCKS`, `REAL_ESTATE`, `BONDS`, or `CRYPTOCURRENCY`.
- Ensuring that the WALLET has sufficient available funds to cover the requested hold amount.

Executing a `Hold` transaction does not involve interacting with third-party APIs since no funds are actually moved. Instead, the funds are reserved within the WALLET to indicate their future use. This is achieved by posting two journal entries to the ledger: one decreases the funds under `BalanceType.AVAILABLE`, while the other increases the funds under `BalanceType.HOLDING`. After these entries are recorded, the status of the `Hold` transaction transitions to `PROCESSING`, indicating that the hold is in progress but not yet completed. The hold must be explicitly completed at a later stage. In the example scenario, it will be completed once the reserved funds are successfully transferred to the INVESTMENT WALLET, fulfilling the investment request.

Once the `Hold` is created for the investment request, the next step is to create `TransferFromHold` transaction(s), which instantly move(s) the reserved funds to another wallet.

```
class TransferFromHold(
    id: String,
    batchId: String? = null,
    amount: BigDecimal,
    idempotencyKey: String,
    originatorWalletId: String,
    originatorSubwalletType: SubwalletType,
    status: TransactionStatus,
    private val beneficiaryWalletId: String,
    private val beneficiarySubwalletType: SubwalletType,
) : Transaction(
        id,
        batchId,
        amount,
        idempotencyKey,
        originatorWalletId,
        originatorSubwalletType,
        status,
    ) {
    override fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        val originatorWallet =
            walletsRepo.findById(this.originatorWalletId)
                ?: throw NoSuchElementException("Wallet not found")
        val beneficiaryWallet =
            walletsRepo.findById(this.beneficiaryWalletId)
                ?: throw NoSuchElementException("Wallet not found")

        val validTransferFromHold =
            (originatorWallet is RealMoneyWallet &&
                beneficiaryWallet is InvestmentWallet) ||
            (originatorWallet is InvestmentWallet &&
                beneficiaryWallet is RealMoneyWallet)

        if (!validTransferFromHold) {
            throw TransferFromHoldNotAllowed("Transfer not allowed between wallets")
        }

        val originatorSubwalletPendingBalance =
            ledgerService.getBalance(
                originatorWalletId,
                listOf(
                    LedgerQuery(
                        subwalletType = originatorSubwalletType,
                        balanceType = BalanceType.HOLDING,
                    ),
                ),
            )

        if (this.amount > originatorSubwalletPendingBalance) {
            throw InsufficientFundsException("Wallet has no sufficient funds")
        }
    }

    override suspend fun process(
        transactionsRepo: TransactionsDatabase,
        ledgerService: LedgerService,
        partnerService: PartnerService,
    ) {
        partnerService.executeInternalTransfer(this)

        val journalEntries =
            listOf(
                CreateJournalEntry(
                    walletId = this.originatorWalletId,
                    subwalletType = this.originatorSubwalletType,
                    balanceType = BalanceType.HOLDING,
```

```kotlin
                        amount = -this.amount,
                    ),
                    CreateJournalEntry(
                        walletId = this.beneficiaryWalletId,
                        subwalletType = this.beneficiarySubwalletType,
                        balanceType = BalanceType.AVAILABLE,
                        amount = this.amount,
                    ),
                )

        ledgerService.postJournalEntries(journalEntries)

        this.updateStatus(transactionsRepo, newStatus = TransactionStatus.COMPLETED)
    }
}
```

A `TransferFromHold` it goes through the same lifecycle every transaction does: creation, validation and processing. The validation of a `TransferFromHold` transaction consists of:

- Verifying that the `TransferFromHold` is being executed between Real Money Wallet and Investment Wallet;
- Ensuring that the Wallet has sufficient reserved funds to cover the requested transfer amount.

Unlike a `Hold` transaction, a `TransferFromHold` involves the actual movement of funds between wallets, requiring a call to the `executeInternalTransfer` method, which interacts with a third-party API. Once the partner fulfills the transfer, the ledger must be updated to reflect the new wallet state. This is achieved by posting two journal entries: one reduces the funds under `BalanceType.HOLDING`, and the other increases the funds under `BalanceType.AVAILABLE` in the receiving wallet. Since a `TransferFromHold` is processed instantly, its status transitions to `COMPLETED` at the end of the process method, indicating the successful completion of the transaction.

`TransferFromHold` transactions are created to complete investment requests after a `Hold` transaction has been placed. They are created by a method called `buyFunds`, designed to be run periodically by a cronjob.

```kotlin
class InvestmentService(
    private val transactionsRepo: TransactionsDatabase,
    private val walletRepo: WalletsDatabase,
    private val investmentPolicyRepo: InvestmentPolicyDatabase,
    private val transactionsService: TransactionsService,
    private val ledgerService: LedgerService,
) {
    // [...]

    suspend fun buyFunds() {
        val transactions =
            transactionsRepo.find(
                TransactionFilter(
                    status = TransactionStatus.PROCESSING,
                    subwalletType = listOf(SubwalletType.REAL_MONEY),
                ),
            )

        for (investmentTransaction in transactions) {
            val wallet =
                walletRepo.findById(investmentTransaction.originatorWalletId)
                    ?: throw NoSuchElementException("Wallet not found")
            val investmentPolicy =
                investmentPolicyRepo.findById(wallet.policyId)
                    ?: throw NoSuchElementException("Policy not found")
            val investmentWallet =
                walletRepo
                    .find(
                        WalletFilter(
                            customerId = wallet.customerId,
                            type = WalletType.INVESTMENT,
                        ),
                    ).firstOrNull()
                        ?: throw NoSuchElementException("Wallet not found")

            try {
                executeMovementWithInvestmentPolicy(
                    InvestmentMovementRequest(
                        amount = investmentTransaction.amount,
                        idempotencyKey = investmentTransaction.id,
                        walletId = investmentTransaction.originatorWalletId,
                        targetWalletId = investmentWallet.id,
                        investmentPolicy = investmentPolicy,
                        transactionType = TransactionType.TRANSFER_FROM_HOLD,
                    ),
                    SubwalletType.REAL_MONEY,
                )

                val transientErrorTransactions =
                    transactionsRepo.find(
                        TransactionFilter(
                            batchId = investmentTransaction.id,
                            status = TransactionStatus.TRANSIENT_ERROR,
                        ),
                    )

                if (transientErrorTransactions.isEmpty()) {
                    investmentTransaction
                        .updateStatus(transactionsRepo, TransactionStatus.COMPLETED)
                }
            } catch (e: TransactionFailed) {
                val message = e.message.toString()
                logger.error(message)
                // validations failed for some transaction in the batch, but other
                // transactions may have succeeded before and their funds are invested
                // now! To solve this, we'd have to reverse the investment transaction
                // partially or use manual remediation...
            }
        }
    }

    // [...]
}
```

The buyFunds method starts by finding all Hold transactions on Real Money Wallet's currently in progress within the Digital Wallet System. These Hold transactions were created by the WalletsService to reserve funds for the investment operation. For each Hold transaction found, buyFunds will:

- Find the Real Money Wallet that originated the investment request;
- Find the customer's InvestmentPolicy;
- Execute a money movement based on the InvestmentPolicy by calling executeMovementWithInvestmentPolicy;
- Complete the investment request, depending on the outcome of executeMovementWithInvestmentPolicy, by updating the wallet status.

```kotlin
class InvestmentService(
    private val transactionsRepo: TransactionsDatabase,
    private val walletRepo: WalletsDatabase,
    private val investmentPolicyRepo: InvestmentPolicyDatabase,
    private val transactionsService: TransactionsService,
    private val ledgerService: LedgerService,
) {
    // [...]

    suspend fun executeMovementWithInvestmentPolicy(
        request: InvestmentMovementRequest,
        originatorSubwalletType: SubwalletType? = null,
    ) {
        for ((subwalletType, percentage) in
          request.investmentPolicy.allocationStrategy) {
            val processTransactionRequest: ProcessTransactionRequest =
                when (request.transactionType) {
                    TransactionType.HOLD ->
                        buildHoldRequest(request, subwalletType, percentage)
                    TransactionType.TRANSFER_FROM_HOLD ->
                        buildTransferFromHoldRequest(
                            request,
                            originatorSubwalletType,
                            subwalletType,
                            percentage,
                        )
                    else ->
                        throw IllegalArgumentException("Unsupported transaction type")
                }

            try {
                transactionsService.processTransaction(processTransactionRequest)
            } catch (e: ValidationException) {
                transactionsService
                    .handleException(e,
                        TransactionStatus.FAILED,
                        request.idempotencyKey
                    )

                // problem: we cannot ensure atomicity as we might have executed
                // transfers with the partner at this point. We'd have to call the
                // partner again to reverse the transfers already executed
                // in this batch and then reverse the journal entries posted to
                // ledger, which sounds not the best design...

                throw TransactionFailed("holdWithPolicy failed")
            } catch (e: PartnerException) {
                // this error can be retried; let's just ignore it

                transactionsService.handleException(e,
                    TransactionStatus.TRANSIENT_ERROR,
                    request.idempotencyKey
                )
            }
        }
    }

    private suspend fun buildHoldRequest(
        request: InvestmentMovementRequest,
        subwalletType: SubwalletType,
        percentage: BigDecimal,
    ): ProcessTransactionRequest =
        ProcessTransactionRequest(
            amount = request.amount * percentage,
            batchId = request.idempotencyKey,
            idempotencyKey = "${request.idempotencyKey}_$subwalletType}",
            originatorWalletId = request.walletId,
            originatorSubwalletType = subwalletType,
            type = TransactionType.HOLD,
        )

    private suspend fun buildTransferFromHoldRequest(
        request: InvestmentMovementRequest,
        originatorSubwalletType: SubwalletType?,
        beneficiarySubwalletType: SubwalletType,
        percentage: BigDecimal,
    ): ProcessTransactionRequest {
        val subwalletType = originatorSubwalletType
            ?: throw IllegalArgumentException("originatorSubwalletType not provided.")

        return ProcessTransactionRequest(
            amount = request.amount.multiply(percentage),
            batchId = request.idempotencyKey,
            idempotencyKey = "${request.idempotencyKey}_$beneficiarySubwalletType",
            originatorWalletId = request.walletId,
            originatorSubwalletType = subwalletType,
            beneficiaryWalletId = request.targetWalletId,
            beneficiarySubwalletType = beneficiarySubwalletType,
            type = TransactionType.TRANSFER_FROM_HOLD,
        )
    }

    // [...]
}
```

For a `Hold` transaction reserving $100 in the Real Money Wallet and an InvestmentPolicy that splits this amount equally among all investment options (25% each for stocks, real estate, bonds, and cryptocurrency), the method `executeMovementWithInvestmentPolicy` will create a transaction batch comprising four `TransferFromHold` transactions. These transactions are defined as follows:

- $25 is transferred to the Investment Wallet under the `SubwalletType.Stocks`;
- $25 is transferred to the Investment Wallet under the `SubwalletType.RealEstate`;
- $25 is transferred to the Investment Wallet under the `SubwalletType.Bonds`;
- $25 is transferred to the Investment Wallet under the `SubwalletType.Cryptocurrency`.

The investment request of $100 will be fulfilled only if all four `TransferFromHold` transactions complete successfully, ensuring the full amount reserved in the Real Money Wallet is invested.

A key requirement of the Digital Wallet System is batch processing with atomicity. For this scenario, atomicity ensures that either all `TransferFromHold` transactions complete successfully — finalizing the $100 investment — or all transactions fail permanently, resulting in the failure of the entire investment request. This guarantees that partial investments do not occur, maintaining system consistency.

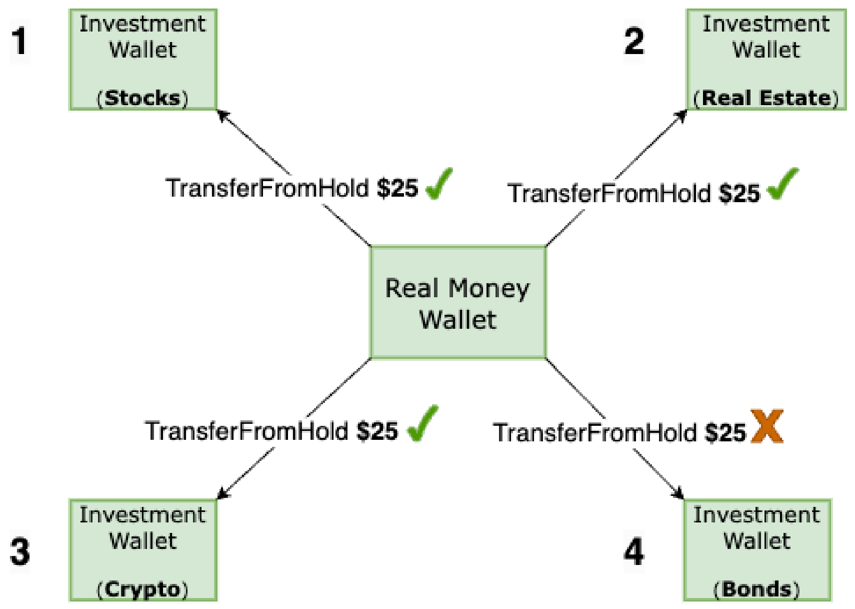


Figure 6.3: *The figure represents a transaction batch containing four* `TransferFromHold` *transactions. Transactions 1, 2 and 3 succeed while transaction 4 fails permanently.*

In the scenario illustrated in Figure 6.3 , transactions 1, 2, and 3 were successfully executed, while transaction 4 failed permanently due to a validation error. Since transactions failing due to validation errors are ineligible for retries, the status of transaction 4 must transition to the terminal status `FAILED`.

However, transaction 4 is part of a batch that includes three other transactions, all of which must also fail permanently to meet the atomicity requirement. In this case, the expected behavior is that failing the entire batch will prevent the investment request from being fulfilled and ensure the release of the funds initially held in the REAL MONEY WALLET.

Enforcing atomicity in the Kotlin implementation of the Digital Wallet System is challenging. By the time transaction 4 encountered a validation error, transactions 1, 2, and 3 had already been completed, transferring $75 to the INVESTMENT WALLET. To reverse these successfully executed transactions, the system would need to:

- Move the $75 back to the Real Money Wallet and reverse the corresponding journal entries in the ledger to maintain consistency, or
- Resort to manual remediation.

Reverting the $75 to the Real Money Wallet involves recalling the third-party API, which may incur additional costs if the partner charges per transfer. Ideally, the Digital Wallet System should avoid invoking the third-party API unless it can ensure that permanent failures, such as validation errors, are unlikely for the entire batch.

```
type Action = Transaction => Either[PartnerServiceError, Unit]
type ProcessTransactionTuple =
    (Transaction, List[CreateJournalEntry], TransactionStatus, Option[Action])

class TransactionsService(
  transactionsRepo: TransactionDatabase,
```

```scala
  validationService: TransactionValidationService,
  partnerService: PartnerService,
  ledgerService: LedgerService,
) {
  // [...]

  def process(
  transaction: Transaction
  ): Either[ProcessError, ProcessTransactionTuple] = {
    lib.maybeLogError(() => {
      for {
        _ <- validationService
                .validateTransaction(transaction)
                .left.map(e => ProcessError(e.message))
        processTransactionTuple <-
                processTransaction(transaction)
                .left.map(e => ProcessError(e.message))
      } yield processTransactionTuple
    })
  }

  private def processTransaction(
    transaction: Transaction
  ): Either[TransactionServiceError, ProcessTransactionTuple] = {
    transaction.transactionType match {
      case TransactionType.Deposit => processDeposit(transaction)
      case TransactionType.Withdraw => processWithdraw(transaction)
      case TransactionType.Hold => processHold(transaction)
      case TransactionType.Transfer => processTransfer(transaction)
      case TransactionType.TransferFromHold => processTransferFromHold(transaction)
    }
  }

  private def processTransferFromHold(
    transaction: Transaction
  ): Either[TransactionServiceError, ProcessTransactionTuple] = {
    for {
      beneficiaryWalletId <-
        transaction
        .beneficiaryWalletId
        .toRight(ProcessError(s"TransferFromHold must contain beneficiaryWalletId"))
      beneficiarySubwalletType <-
        transaction
        .beneficiarySubwalletType
        .toRight(ProcessError(s"Wallet not found"))
    } yield {
      val journalEntries = List(
        CreateJournalEntry(
          walletId = Some(transaction.originatorWalletId),
          subwalletType = transaction.originatorSubwalletType,
          balanceType = BalanceType.Holding,
          amount = -transaction.amount
        ),
        CreateJournalEntry(
          walletId = Some(beneficiaryWalletId),
          subwalletType = beneficiarySubwalletType,
          balanceType = BalanceType.Available,
          amount = transaction.amount
        )
      )

      val partnerAction: Action = partnerService.executeInternalTransfer
      (transaction, journalEntries, TransactionStatus.Completed, Some(partnerAction))
    }
  }

  // [...]
}
```

In the Scala implementation, the function handling the processing logic for a `TransferFromHold` transaction is a pure function. Instead of directly performing side effects like calling a third-party API, posting journal entries, or updating the transaction status, the function returns a tuple containing all the necessary details for the caller to decide whether to execute the side effects. The returned information includes:

- Journal entries: Entries to be posted to the ledger.
- Action: A function that interacts with the third-party API.

This design gives the caller the flexibility to determine whether and when to execute these side effects. If the caller chooses to proceed, it can invoke the execute method to trigger the necessary operations.

```scala
type Action = Transaction => Either[PartnerServiceError, Unit]
type ProcessTransactionTuple =
    (Transaction, List[CreateJournalEntry], TransactionStatus, Option[Action])

class TransactionsService(
  transactionsRepo: TransactionDatabase,
  validationService: TransactionValidationService,
  partnerService: PartnerService,
  ledgerService: LedgerService,
) {
  // [...]

  def execute(
    tuple: ProcessTransactionTuple
  ): Either[ExecutionError, Transaction] = {
    val (transaction, journalEntries, statusOnSuccess, maybeExecuteAction) = tuple

    val actionResult: Either[PartnerServiceError, Unit] = maybeExecuteAction match {
      case Some(action) => lib.maybeLogError(() => action(transaction))
      case None => Right(())
    }

    actionResult match {
      case Left(e) =>
        updateStatus(transaction.id, TransactionStatus.TransientError)
        Left(ExecutionError(e.message))
      case Right(_) =>
        ledgerService.postJournalEntries(journalEntries)
        Right(updateStatus(transaction.id, statusOnSuccess))
    }
  }

  def updateStatus(
    transactionId: String,
    status: TransactionStatus
  ): Transaction = {
    transactionsRepo.update(transactionId, status)
  }

  // [...]
}
```

In the Scala implementation of `executeMovementWithInvestmentPolicy`, all `TransferFromHold` transactions are processed before execution. If transactions 1, 2, and 3 succeed but transaction 4 encounters a permanent failure, the method finishes without attempting to execute any of the transactions.

```
class InvestmentService(
    transactionsRepo: TransactionDatabase,
    walletsRepo: WalletsDatabase,
    investmentPolicyRepo: InvestmentPolicyDatabase,
    transactionsService: TransactionsService) {
  // [...]

  def executeMovementWithInvestmentPolicy(
    request: MovementRequest
  ): Either[InvestmentServiceError, Unit] = {
    val allocationStrategy = request.investmentPolicy.allocationStrategy.toList

    allocationStrategy
      .filter((_, percentage) => percentage != BigDecimal(0))
      .traverse { case (subwalletType, percentage) =>
        for {
          (originatorSubwalletType,
            beneficiaryWalletId,
            beneficiarySubwalletType
          ) <- getTransactionDetails(request, subwalletType)

          createTransactionRequest = CreateTransactionRequest(
            amount = request.amount * percentage,
            batchId = Some(request.idempotencyKey),
            idempotencyKey = s"${request.idempotencyKey}_$subwalletType",
            originatorWalletId = request.walletId,
            originatorSubwalletType = originatorSubwalletType,
            beneficiaryWalletId = beneficiaryWalletId,
            beneficiarySubwalletType = beneficiarySubwalletType,
            transactionType = request.transactionType
          )

          transaction <-
            transactionsService
                .create(createTransactionRequest)
                .left.map { e =>
                    CreateTransactionFailed(s"Failed to create transaction")
                }

          processTransactionTuple <-
            transactionsService
            .process(transaction)
            .left.map { e =>
                transactionsService.failBatch(request.idempotencyKey)
                ProcessTransactionFailed(s"Failed to process transaction")
          }
        } yield processTransactionTuple
      }.flatMap { processTransactionTuples =>
        lib.maybeLogError(() => {
            val failures =
              processTransactionTuples
                .map { tuple => transactionsService.execute(tuple) }
                .collect { case Left(error) => error }

            if (failures.nonEmpty) {
              Left(
                ExecuteTransactionFailed(s"${failures.size} transaction(s) failed")
              )
            } else {
              Right(())
            }
          }
        )
      }
  }

  private def getTransactionDetails(
    request: MovementRequest,
    subwalletType: SubwalletType
  ): Either[
        InvestmentServiceError, (SubwalletType, Option[String], Option[SubwalletType])
    ] = {
    request.transactionType match {
      case TransactionType.Hold =>
        Right((subwalletType, None, None))

      case TransactionType.TransferFromHold =>
        request.walletSubwalletType
          .toRight(InvestmentServiceInternalError(s"Missing wallet subwallet type"))
          .map(walletSubwalletType => (
            walletSubwalletType,
            request.targetWalletId,
            Some(subwalletType)
          ))

      case _ =>
        Left(InvestmentServiceInternalError(s"Unsupported transaction type"))
    }
  }

  // [...]
}
```

Since `process` is a pure function, no side effects are performed during this stage. This ensures that the entire transaction batch can be marked as failed without the need for complex rollback operations to reverse prior actions, such as recalling the third-party API.

```
class InvestmentService(
    transactionsRepo: TransactionDatabase,
    walletsRepo: WalletsDatabase,
    investmentPolicyRepo: InvestmentPolicyDatabase,
    transactionsService: TransactionsService) {
  // [...]

  def buyFunds(): Unit = {
    transactionsRepo
      .find(
        TransactionFilter(
          status = Some(TransactionStatus.Processing),
          subwalletType = Some(List(SubwalletType.RealMoney))
        )
      )
      .foreach(investmentTransaction => {
          for {
            wallet <-
              walletsRepo
                .findById(investmentTransaction.originatorWalletId)
                .toRight(InvestmentServiceInternalError(s"Wallet not found"))
            policyId <-
              wallet
                .policyId
                .toRight(InvestmentServiceInternalError(s"Wallet has no policyId"))
            investmentPolicy <-
              investmentPolicyRepo
                .findById(policyId)
                .toRight(
                    InvestmentServiceInternalError(s"Investment policy not found")
                )
            investmentWallet <-
              walletsRepo
                .find(WalletFilter(
                  customerId = Some(wallet.customerId),
                  walletType = Some(WalletType.Investment)
                ))
                .headOption
                .toRight(
                    InvestmentServiceInternalError(s"Investment wallet not found")
                )
          } yield {
            executeMovementWithInvestmentPolicy(MovementRequest(
              amount = investmentTransaction.amount,
              idempotencyKey = investmentTransaction.idempotencyKey,
              walletId = investmentTransaction.originatorWalletId,
              walletSubwalletType = Some(SubwalletType.RealMoney),
              targetWalletId = Some(investmentWallet.id),
              investmentPolicy = investmentPolicy,
              transactionType = TransactionType.TransferFromHold
            ))
            .fold(
              error => {
                error match {
                  case ProcessTransactionFailed(_) =>
                    // Fail the investment transaction gracefully
                    transactionsService.releaseHold(investmentTransaction)
                    transactionsService
                        .updateStatus(
                            investmentTransaction.id,
                            TransactionStatus.Failed
                        )
                  case _ => ()
                }
              },
              _ => {
                // Update the transaction status to completed
                transactionsService
                    .updateStatus(
                        investmentTransaction.id,
                        TransactionStatus.Completed
                    )
              }
            )
          }
      })
  }

  // [...]
}
```

In Scala, if a permanent failure occurs, the system releases and fails the `Hold` transaction that initiated the investment. If any `TransferFromHold` transaction encounters a failure during execution, it becomes eligible for a retry, and no further actions are taken.

While the Scala implementation successfully meets the batch processing with atomicity requirement, the Kotlin implementation faces challenges due to the presence of side effects within the `TransferFromHold` processing logic.

6.5 Testability

To analyze testability, this section examines how the Kotlin and Scala versions process transactions.

Transfer implementation

In the previous section, the Kotlin implementation of the `TransferFromHold` processing logic directly executes side effects such as calling a third-party API, posting journal entries, and updating the transaction status. This pattern is consistently applied across other transaction types. The code below shows the processing logic for `Transfer` transactions:

```kotlin
class TransactionsService(
    private val transactionsRepo: TransactionsDatabase,
    private val walletsRepo: WalletsDatabase,
    private val ledgerService: LedgerService,
    private val partnerService: PartnerService,
) {
    suspend fun processTransaction(request: ProcessTransactionRequest): Transaction {
        val transaction = transactionsRepo.insert(request)
        transaction.validate(walletsRepo, ledgerService)
        transaction.process(transactionsRepo, ledgerService, partnerService)

        return transaction
    }

  // [...]
}
```

```kotlin
class Transfer(
    id: String,
    batchId: String? = null,
    amount: BigDecimal,
    idempotencyKey: String,
    originatorWalletId: String,
    originatorSubwalletType: SubwalletType,
    status: TransactionStatus,
    val beneficiaryWalletId: String,
    val beneficiarySubwalletType: SubwalletType,
) : Transaction(
        id,
        batchId,
        amount,
        idempotencyKey,
        originatorWalletId,
        originatorSubwalletType,
        status,
    ) {
    override fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        val originatorSubwalletType = this.originatorSubwalletType
        val beneficiarySubwalletType = this.beneficiarySubwalletType

        val validTransferPairs =
            setOf(
                SubwalletType.REAL_MONEY to SubwalletType.EMERGENCY_FUND,
                SubwalletType.EMERGENCY_FUND to SubwalletType.REAL_MONEY,
            )

        if ((originatorSubwalletType to beneficiarySubwalletType)
                !in validTransferPairs) {
            throw TransferNotAllowed("Transfer not allowed")
        }

        validateBalance(walletsRepo, ledgerService)
    }

    override suspend fun process(
        transactionsRepo: TransactionsDatabase,
        ledgerService: LedgerService,
        partnerService: PartnerService,
    ) {
        partnerService.executeInternalTransfer(this)

        val journalEntries =
            listOf(
                CreateJournalEntry(
                    walletId = this.originatorWalletId,
                    subwalletType = this.originatorSubwalletType,
                    balanceType = BalanceType.AVAILABLE,
                    amount = -this.amount,
                ),
                CreateJournalEntry(
                    walletId = this.beneficiaryWalletId,
                    subwalletType = this.beneficiarySubwalletType,
                    balanceType = BalanceType.AVAILABLE,
                    amount = this.amount,
                ),
            )

        ledgerService.postJournalEntries(journalEntries)

        this.updateStatus(transactionsRepo, newStatus = TransactionStatus.COMPLETED)
    }

  // [...]
}
```

By contrast, the Scala implementation separates concerns by providing a pure function to process `Transfer` transactions, postponing side effects through a distinct `execute` function.

```scala
type Action = Transaction => Either[PartnerServiceError, Unit]
type ProcessTransactionTuple =
    (Transaction, List[CreateJournalEntry], TransactionStatus, Option[Action])

class TransactionsService(
  transactionsRepo: TransactionDatabase,
  validationService: TransactionValidationService,
  partnerService: PartnerService,
  ledgerService: LedgerService,
) {
  private val lib = Library()

  def process(transaction: Transaction): Either[ProcessError, ProcessTransactionTuple] = {
    lib.maybeLogError(() => {
      for {
        _ <-
            validationService
            .validateTransaction(transaction)
            .left.map(e => ProcessError(e.message))
        processTransactionTuple <-
            processTransaction(transaction)
            .left.map(e => ProcessError(e.message))
      } yield processTransactionTuple
    })
  }

  def execute(
    tuple: ProcessTransactionTuple
): Either[ExecutionError, Transaction] = {
    val (transaction, journalEntries, statusOnSuccess, maybeExecuteAction) = tuple

    val actionResult: Either[PartnerServiceError, Unit] = maybeExecuteAction match {
      case Some(action) => lib.maybeLogError(() => action(transaction))
      case None => Right(())
    }

    actionResult match {
      case Left(e) =>
        updateStatus(transaction.id, TransactionStatus.TransientError)
        Left(ExecutionError(e.message))
      case Right(_) =>
        ledgerService.postJournalEntries(journalEntries)
        Right(updateStatus(transaction.id, statusOnSuccess))
    }
  }

  def updateStatus(
    transactionId: String,
    status: TransactionStatus
  ): Transaction = {
    transactionsRepo.update(transactionId, status)
  }

  private def processTransaction(
    transaction: Transaction
  ): Either[TransactionServiceError, ProcessTransactionTuple] = {
    transaction.transactionType match {
      case TransactionType.Deposit => processDeposit(transaction)
      case TransactionType.Withdraw => processWithdraw(transaction)
      case TransactionType.Hold => processHold(transaction)
      case TransactionType.Transfer => processTransfer(transaction)
      case TransactionType.TransferFromHold => processTransferFromHold(transaction)
    }
  }

  private def processTransfer(
    transaction: Transaction
  ): Either[TransactionServiceError, ProcessTransactionTuple] = {
    for {
      beneficiaryWalletId <
        transaction
        .beneficiaryWalletId
        .toRight(ProcessError(s"Transfer must contain beneficiaryWalletId"))
      beneficiarySubwalletType <-
        transaction
        .beneficiarySubwalletType
        .toRight(ProcessError(s"Wallet not found"))
    } yield {
      val journalEntries = List(
        CreateJournalEntry(
          walletId = Some(transaction.originatorWalletId),
          subwalletType = transaction.originatorSubwalletType,
          balanceType = BalanceType.Available,
          amount = -transaction.amount
        ),
        CreateJournalEntry(
          walletId = Some(beneficiaryWalletId),
          subwalletType = beneficiarySubwalletType,
          balanceType = BalanceType.Available,
          amount = transaction.amount
        )
      )

      val partnerAction: Action = partnerService.executeInternalTransfer
      (transaction, journalEntries, TransactionStatus.Completed, Some(partnerAction))
    }
  }

  // [...]
}
```

Testing the transfer implementation

Testing the Kotlin version of the `Transfer` logic involved mocking the third-party API and the LedgerService.

```
it("works") {
    val request =
        ProcessTransactionRequest(
            amount = BigDecimal(100),
            idempotencyKey = "idempotencyKey",
            originatorWalletId = "realMoneyWalletId",
            originatorSubwalletType = SubwalletType.REAL_MONEY,
            beneficiaryWalletId = "emergencyFundsWalletId",
            beneficiarySubwalletType = SubwalletType.EMERGENCY_FUNDS,
            type = TransactionType.TRANSFER,
        )

    val transaction = transactionsService.processTransaction(request)

    transaction.status shouldBe TransactionStatus.COMPLETED

    coVerify(exactly = 1) { partnerServiceMock.executeInternalTransfer(any()) }
    coVerify(exactly = 1) {
        ledgerServiceMock.postJournalEntries(
            journalEntries =
                listOf(
                    CreateJournalEntry(
                        walletId = "realMoneyWalletId",
                        subwalletType = SubwalletType.REAL_MONEY,
                        amount = BigDecimal(-100),
                        balanceType = BalanceType.AVAILABLE,
                    ),
                    CreateJournalEntry(
                        walletId = "emergencyFundsWalletId",
                        subwalletType = SubwalletType.EMERGENCY_FUNDS,
                        amount = BigDecimal(100),
                        balanceType = BalanceType.AVAILABLE,
                    ),
                ),
        )
    }
}

```

Although the Scala implementation avoids mocking the third-party API and LedgerService for transaction processing, it requires mocking the TransactionValidationService. The Kotlin implementation encapsulates validation logic within subclasses using polymorphic methods, which eliminates the need for mocking the validation process entirely.

```
it should "process transfer" in {
    val transaction = Transaction(
      id = UUID.randomUUID().toString,
      transactionType = TransactionType.Transfer,
      originatorSubwalletType = SubwalletType.RealMoney,
      amount = BigDecimal(100),
      originatorWalletId = "realMoneyWalletId",
      beneficiarySubwalletType = Some(SubwalletType.EmergencyFunds),
      beneficiaryWalletId = Some("emergencyFundsWalletId"),
      idempotencyKey = "idempotencyKey",
      insertedAt = LocalDateTime.now(),
      status = TransactionStatus.Creating
    )

    when(
      transactionValidationServiceMock
        .validateTransaction(any()))
        .thenReturn(Right(()))
    )

    val result = transactionService.process(transaction)

    val expectedJournalEntries = List(
      CreateJournalEntry(
        walletId = Some("realMoneyWalletId"),
        subwalletType = SubwalletType.RealMoney,
        balanceType = BalanceType.Available,
        amount = -BigDecimal(100)
      ),
      CreateJournalEntry(
        walletId = Some("emergencyFundsWalletId"),
        subwalletType = SubwalletType.EmergencyFunds,
        balanceType = BalanceType.Available,
        amount = BigDecimal(100)
      )
    )

    result match {
      case Right((transactionResult, journalEntries, transactionStatus, action)) =>
        transactionResult shouldBe transaction
        journalEntries shouldBe expectedJournalEntries
        transactionStatus shouldBe TransactionStatus.Completed
        action.isDefined shouldBe true
      case Left(error) =>
        fail(s"Expected Right but got Left with error: $error")
    }
}
```

Moreover, Scala introduces an additional function, `execute`, to handle side effects such as calling the third-party API and posting journal entries. While the `process` function remains pure and avoids side effects, this separation means that `execute` requires its own set of tests. These tests need to mock dependencies like the third-party API and LEDGERSERVICE to ensure the side effects are correctly executed and verified. This adds an extra layer of testing compared to Kotlin, where the side effects are integrated into the main processing logic and tested together.

```
it should "execute transfer" in {
    val id = UUID.randomUUID().toString
    val transaction = utils.insertTransactionInMemory(
      db = transactionsRepo,
      id = id,
```

```scala
      transactionType = TransactionType.Transfer,
      originatorSubwalletType = SubwalletType.RealMoney,
      amount = BigDecimal(100),
      originatorWalletId = "realMoneyWalletId",
      beneficiarySubwalletType = Some(SubwalletType.EmergencyFunds),
      beneficiaryWalletId = Some("emergencyFundsWalletId"),
      insertedAt = LocalDateTime.now(),
      status = TransactionStatus.Creating
    )

    val journalEntries = List(
      CreateJournalEntry(
        walletId = Some("realMoneyWalletId"),
        subwalletType = SubwalletType.RealMoney,
        balanceType = BalanceType.Available,
        amount = -BigDecimal(100)
      ),
      CreateJournalEntry(
        walletId = Some("emergencyFundsWalletId"),
        subwalletType = SubwalletType.EmergencyFunds,
        balanceType = BalanceType.Available,
        amount = BigDecimal(100)
      )
    )

    val action: Action = partnerServiceMock.executeInternalTransfer

    val tuple: ProcessTransactionTuple =
        (transaction, journalEntries, TransactionStatus.Completed, Some(action))

    when(partnerServiceMock.executeInternalTransfer(any())).thenReturn(Right(()))
    when(ledgerServiceMock.postJournalEntries(any())).thenReturn(LocalDateTime.now())

    val result = transactionService.execute(tuple)

    result match {
      case Right(transactionResult) =>
        transactionResult.id shouldBe id
        transactionResult.status shouldBe TransactionStatus.Completed
      case Left(error) =>
        fail(s"Expected Right but got Left with error: $error")
    }

    verify(ledgerServiceMock).postJournalEntries(ArgumentMatchers.eq(journalEntries))
}
```

Conclusion

The Scala implementation, by keeping the `processTransfer` function pure and free of side effects, allows for straightforward and isolated testing of the core logic. However, the separation of side effects into the `execute` function introduces additional testing requirements. Testing the `execute` function necessitates mocking dependencies to verify that side effects are performed correctly. While this adds an extra testing layer, it ensures a clear distinction between testing transaction processing logic and side-effect execution.

On the other hand, Kotlin integrates side effects directly into the processing logic, requiring mocks for both the third-party API and LedgerService when testing the transaction flow. While this approach simplifies the overall test structure, it increases the coupling between the processing logic and dependencies, making it harder to isolate the core logic for testing.

6.6 Readability

To analyze readability, this section examines how the Kotlin and Scala versions implements the logic that fulfills liquidation requests.

Similar to investment requests, a liquidation request begins in the WalletsService with the creation of a Hold transaction. However, in this case, the Hold is created on the Investment Wallet to reserve funds that will later be transferred to the Real Money Wallet. A batch of Hold transactions is generated according to the customer's InvestmentPolicy:

```kotlin
suspend fun liquidate(request: LiquidationRequest) {
    val wallet =
        walletsRepo
            .find(
                WalletFilter(
                    customerId = request.customerId,
                    type = WalletType.INVESTMENT,
                ),
            ).firstOrNull() ?: throw NoSuchElementException("Wallet not found")
    val investmentPolicy =
        investmentPolicyRepo.findById(wallet.policyId)
            ?: throw NoSuchElementException("Policy not found")

    try {
        investmentService.executeMovementWithInvestmentPolicy(
            InvestmentMovementRequest(
                amount = request.amount,
                idempotencyKey = request.idempotencyKey,
                walletId = wallet.id,
                investmentPolicy = investmentPolicy,
                transactionType = TransactionType.HOLD,
            ),
        )
    } catch (e: TransactionFailed) {
        throw LiquidationFailed(e.message.toString())
    }
}
```

The Kotlin implementation places the logic for fulfilling liquidation requests in the `sellFunds` method within the InvestmentService:

```kotlin
suspend fun sellFunds() {
    val transactions =
        transactionsRepo.find(
            TransactionFilter(
                status = TransactionStatus.PROCESSING,
                subwalletType =
                    listOf(
                        SubwalletType.STOCK,
                        SubwalletType.BONDS,
                        SubwalletType.REAL_ESTATE,
                        SubwalletType.CRYPTOCURRENCY,
                    ),
            ),
        )

    for (liquidationTransaction in transactions) {
        val wallet =
            walletRepo.findById(liquidationTransaction.originatorWalletId)
                ?: throw NoSuchElementException("Wallet not found")
        val realMoneyWallet =
            walletRepo
                .find(
                    WalletFilter(
                        customerId = wallet.customerId,
                        type = WalletType.REAL_MONEY,
                    ),
                ).firstOrNull() ?: throw NoSuchElementException("Wallet not found")

        try{
            transactionsService.processTransaction(
                ProcessTransactionRequest(
                    amount = liquidationTransaction.amount,
                    idempotencyKey = liquidationTransaction.id,
                    originatorWalletId = liquidationTransaction.originatorWalletId,
                    originatorSubwalletType =
                        liquidationTransaction.originatorSubwalletType,
                    beneficiaryWalletId = realMoneyWallet.id,
                    beneficiarySubwalletType = SubwalletType.REAL_MONEY,
                    type = TransactionType.TRANSFER_FROM_HOLD,
                )
            )

            liquidationTransaction.updateStatus(
              transactionsRepo,
              newStatus = TransactionStatus.COMPLETED,
             )
        } catch (e: ValidationException) {
            logger.error("Transaction ${liquidationTransaction.id} failed")
            liquidationTransaction.reverse(ledgerService)
            liquidationTransaction.updateStatus(
                transactionsRepo,
                newStatus = TransactionStatus.FAILED
            )
        } catch (e: PartnerException) {
            // this error can be retried; let's just ignore it
            transactionsService.handleException(e,
                TransactionStatus.TRANSIENT_ERROR,
                request.idempotencyKey
            )
        }
    }
}
```

The Kotlin implementation starts by finding all HOLD transactions in the INVESTMENT WALLET. For each HOLD, it identifies the corresponding REAL MONEY WALLET to which the reserved funds should be transferred.

A single TRANSFERFROMHOLD transaction is created for each HOLD to transfer the reserved funds to the REAL MONEY WALLET. The implementation uses a `try-catch` block to handle two types of failures: permanent failures and retriable failures. If no failures occur, the originating HOLD transaction is marked as **Completed**.

The structure of the `sellFunds` method is straightforward and intuitive. Its purpose is easy to infer, as it starts by finding ongoing transactions under the STOCK, BONDS, REALESTATE, and CRYPTOCURRENCY subwallets, which clearly indicates the goal of fulfilling liquidation requests.

The `for` loop explicitly iterates over each transaction, performing a series of operations that may fail. These failures are addressed using `Exceptions` and `try-catch` blocks. The use of `try-catch` blocks, which visually stand out from the rest of the code, enhances readability by distinguishing the error-handling logic from the main operation flow.

Similarly, in the Scala implementation, a `liquidate` function in the WALLETSSERVICE initiates liquidation requests:

```
def liquidate(request: LiquidationRequest): Either[LiquidationFailedError, Unit] = {
    val wallets = walletsRepo.find(
        WalletFilter(
            customerId = Some(request.customerId),
            walletType = Some(WalletType.Investment)
            )
        )

    wallets match {
      case List(wallet) =>
        for {
          policyId <-
              wallet
              .policyId
              .toRight(LiquidationFailedError(s"Wallet has no policyId"))

          investmentPolicy <-
            investmentPolicyRepo
              .findById(policyId)
              .toRight(LiquidationFailedError(s"Investment policy found"))

          _ <-
            investmentService.executeMovementWithInvestmentPolicy(MovementRequest(
                amount = request.amount,
                idempotencyKey = request.idempotencyKey,
```

```scala
                  walletId = wallet.id,
                  investmentPolicy = investmentPolicy,
                  transactionType = TransactionType.Hold
                )).left.map(e => LiquidationFailedError(e.message))
          } yield ()
        case _ =>
          Left(LiquidationFailedError(s"None or multiple wallets found"))
      }
    }
```

The core logic for fulfilling liquidation requests resides in the `sellFunds` function within the InvestmentService:

```scala
def sellFunds(): Unit = {
    transactionsRepo
      .find(
        TransactionFilter(
          status = Some(TransactionStatus.Processing),
          subwalletType =
            Some(
              List(
                SubwalletType.Stock,
                SubwalletType.Bonds,
                SubwalletType.RealEstate,
                SubwalletType.Cryptocurrency,
              )
            )
          )
      )
      .map(liquidationTransaction => {
        for {
          wallet <-
            walletsRepo
              .findById(liquidationTransaction.originatorWalletId)
              .toRight(InvestmentServiceInternalError(s"Wallet not found"))
          realMoneyWallet <-
            walletsRepo
              .find(WalletFilter(
                customerId = Some(wallet.customerId),
                walletType = Some(WalletType.RealMoney)
              ))
              .headOption
              .toRight(InvestmentServiceInternalError(s"Real money wallet not"))
          transaction <- transactionsService.create(
              CreateTransactionRequest(
                amount = liquidationTransaction.amount,
                idempotencyKey = liquidationTransaction.id,
                originatorWalletId = liquidationTransaction.originatorWalletId,
                originatorSubwalletType =
                    liquidationTransaction.originatorSubwalletType,
                beneficiaryWalletId = Some(realMoneyWallet.id),
                beneficiarySubwalletType = Some(SubwalletType.RealMoney),
                transactionType = TransactionType.TransferFromHold,
              )
            ).left.map { e =>
              CreateTransactionFailed(e.message)
            }

          processTransactionTuple <-
            transactionsService
              .process(transaction)
              .left
              .map { e =>
                // fail transfer from hold
                transactionsService.updateStatus(
                    transaction.id,
                    TransactionStatus.Failed
                )
                // fail liquidation transaction
                transactionsService.releaseHold(liquidationTransaction)
                transactionsService.updateStatus(
                    liquidationTransaction.id,
                    TransactionStatus.Failed
                )
                ProcessTransactionFailed("Failed to process transaction")
              }

          _ <-
            transactionsService
              .execute(processTransactionTuple)
              .fold(
                error => {
                  Left(ExecuteTransactionFailed(error.message))
                },
                _ => {
                  Right(transactionsService.updateStatus(
                    liquidationTransaction.id,
                    TransactionStatus.Completed
                   ))
                }
              )
        } yield ()
      })
  }
```

The Scala implementation also begins by finding ongoing transactions in the Stock, Bonds, RealEstate, and Cryptocurrency subwallets. It uses the higher-order function `map` to apply a series of operations to each Hold transaction.

For error handling, the Scala implementation employs a `for`-comprehension to manage each potentially failable operation. The number of operations in Scala is greater because the creation, processing, and execution logic of transactions are separated into distinct methods.

Errors in Scala are treated as values, composing part of the return type. This approach enforces localized error handling and minimizes the risk of unhandled cases. However, treating errors as regular values reduces their prominence compared to Kotlin's `Exceptions` and `try-catch` blocks, which stand out more clearly in the code. While the functional approach ensures safety and composability through high-order functions, it may lack the visual distinction that facilitates understanding of error-handling and propagation flows.

Chapter 7

Quantitative Analysis

The quantitative analysis aims to gather feedback from developers with diverse backgrounds through a survey, complementing the qualitative analysis.

At the time of this research, the survey has received eight responses. However, it remains open, and additional responses will be considered for future work.

The survey was initially beta-tested with a developer experienced in both paradigms. It was then tested with two additional developers before being made available to a wider public.

The survey consists of two sections: **background questions** and a **questionnaire**. The background questions are designed to map the developers' experience with object-oriented and functional programming languages, as this experience might influence their reasoning when evaluating the code snippets. The questionnaire comprises four questions, each of which includes a brief explanation of a functionality from the Digital Wallet System and two code snippets—one from the Kotlin implementation and the other from the Scala implementation—that implement this functionality.

Developers are asked to evaluate various architectural characteristics, including extensibility, reusability, error handling, error propagation, testability, readability, and maintainability, for each implementation using a Likert scale. Detailed background questions and the descriptions of the questionnaire items can be found in Appendix A.

While the survey collects feedback on maintainability, this characteristic will not be analyzed in this work.

As the survey gathers more responses, it will become possible to correlate the results of the background questions with the responses to the main questionnaire. This will enable the enable the understanding of how developers' experiences influence their perceptions of architectural characteristics.

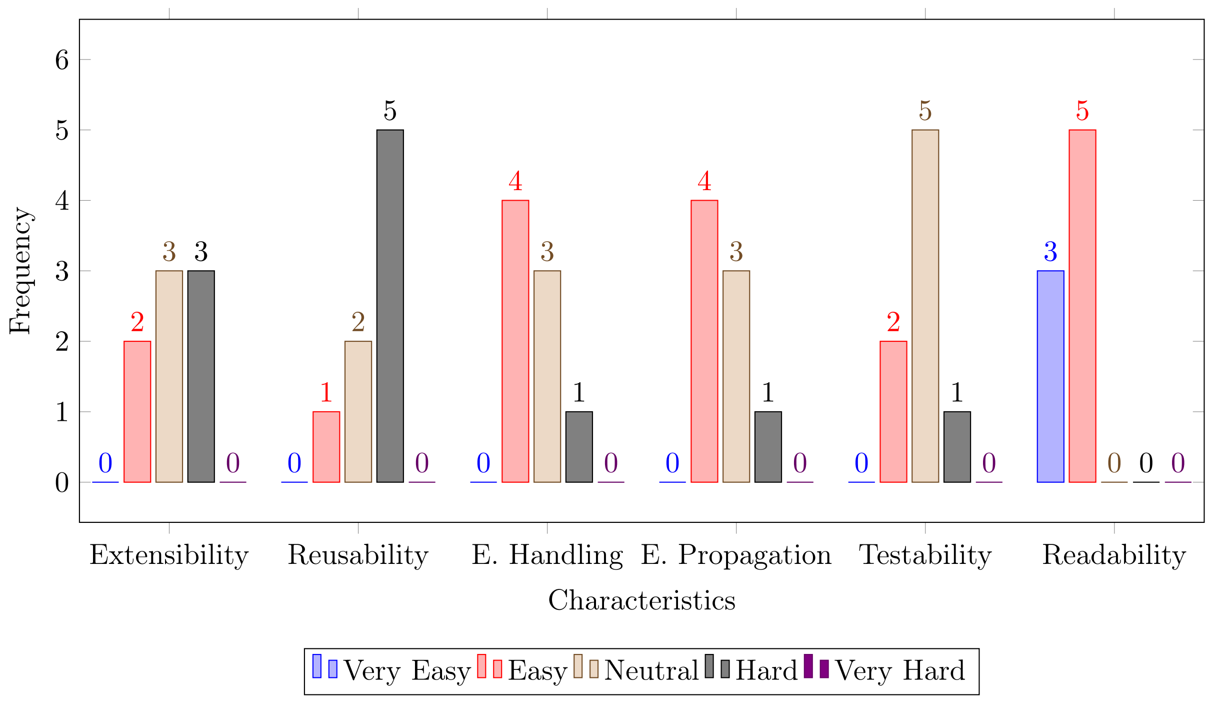
7.1 Results

This section presents the results collected from the survey to date.

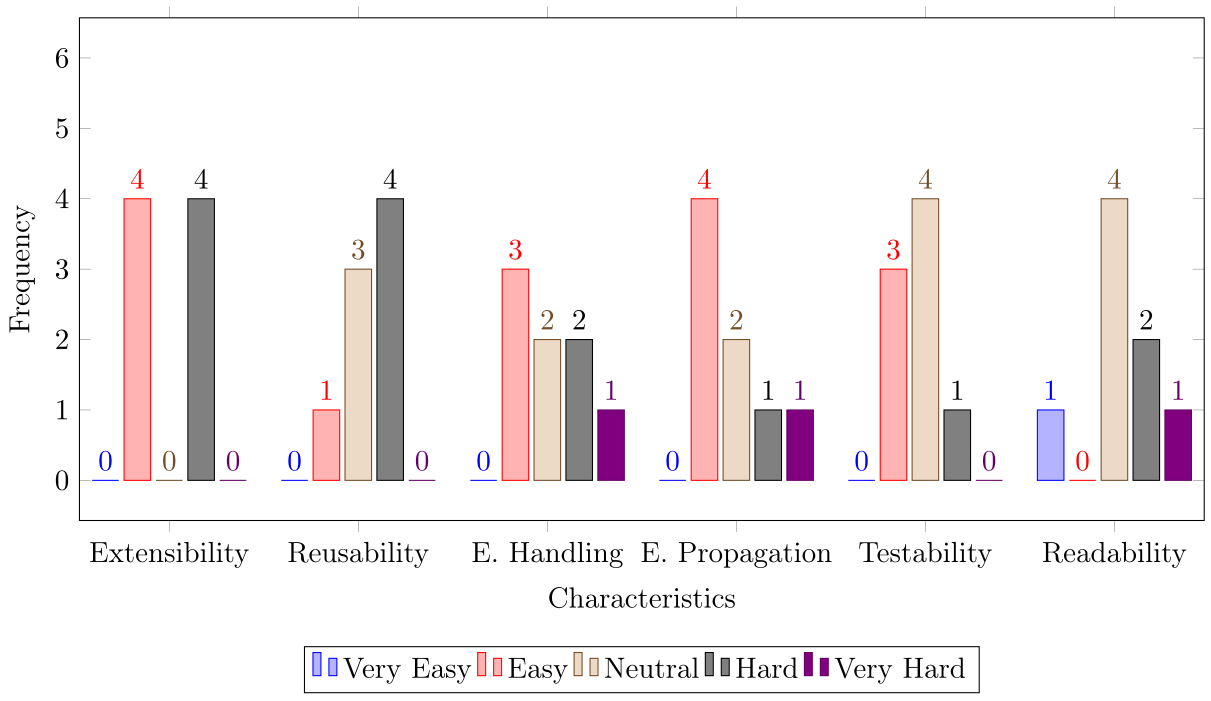
7.1.1 Question #1

Question 1 collects feedback on Kotlin and Scala implementation of the logic that retries a transaction batch.

Kotlin



Scala

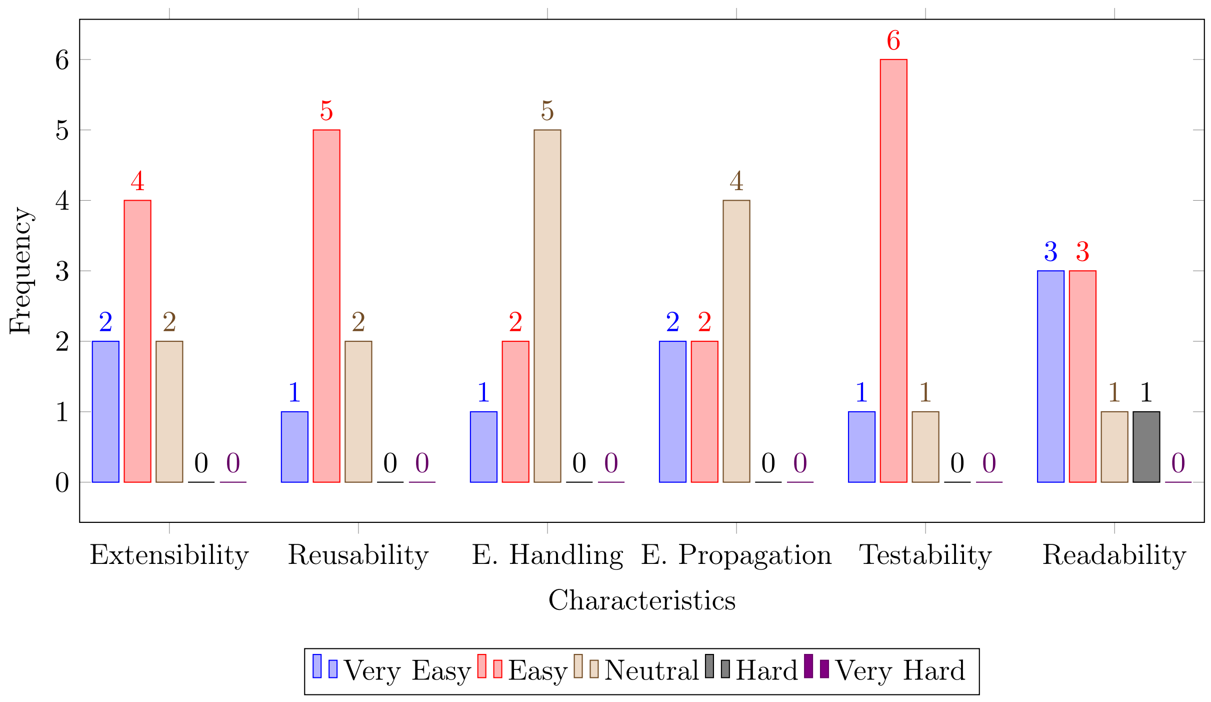


The results of Question 1 reveal a notable difference in the characteristics of error handling, error propagation, and readability. Many developers found the Scala implementation of the retry logic to be more challenging to handle, propagate errors, and read when compared to the Kotlin implementation. This indicates that, for these aspects, Kotlin may offer a clearer and more accessible approach.

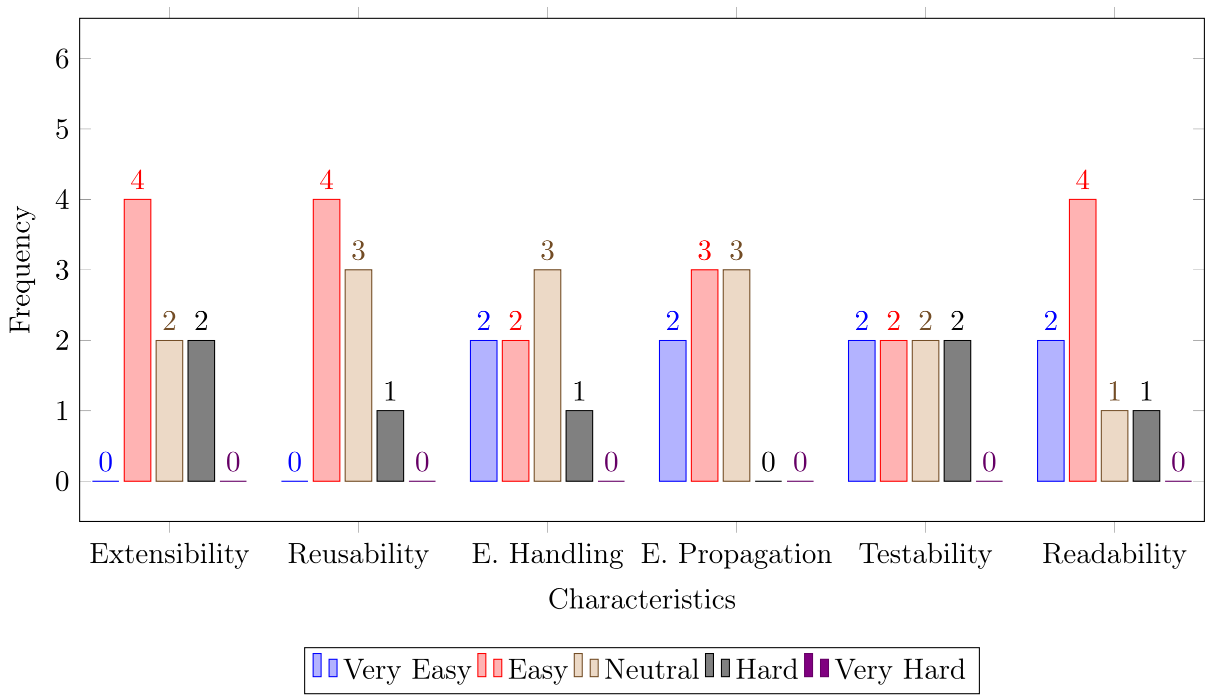
7.1.2 Question #2

Question 2 gathers feedback on the Kotlin and Scala implementations of the Withdrawal validation logic.

Kotlin



Scala

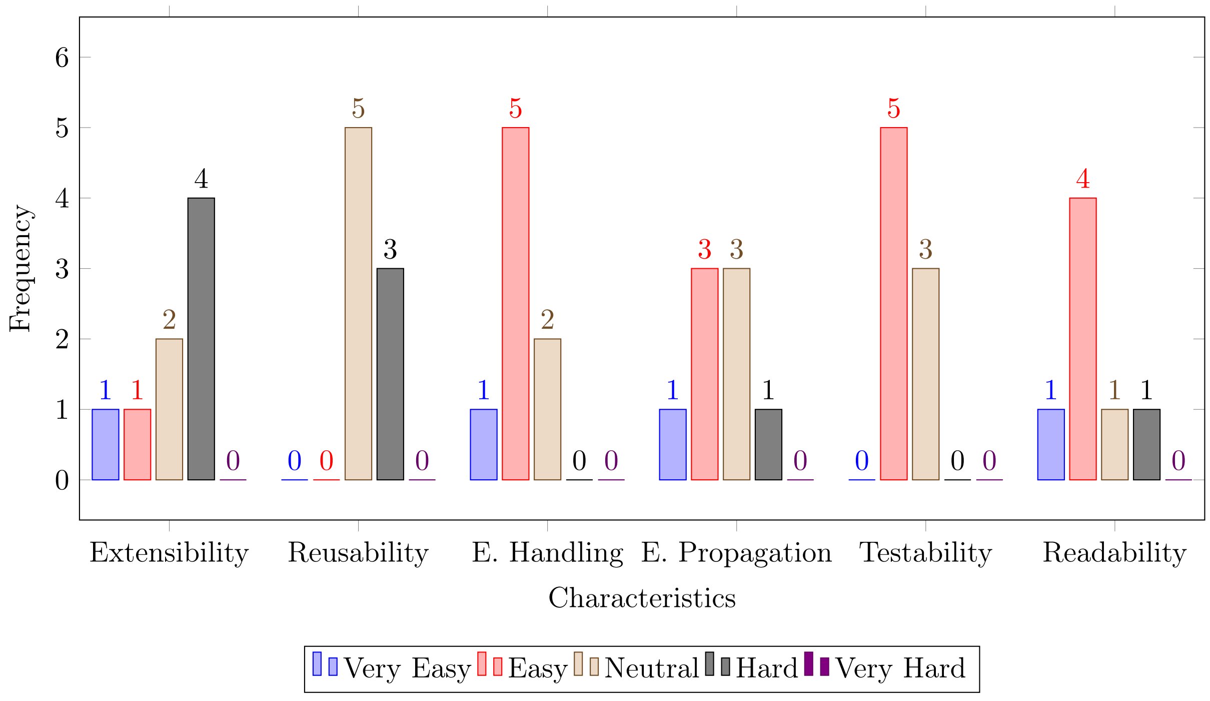


The results of Question 2 reveal a notable difference in several characteristics. Extensibility and reusability were slightly favored in the Kotlin implementation. Error handling, error propagation, and readability showed comparable results between the two implementations. While responses for testability in the Scala implementation were inconclusive, the Kotlin implementation stood out positively in this area, reflecting a stronger perceived capability for effective testing in Kotlin.

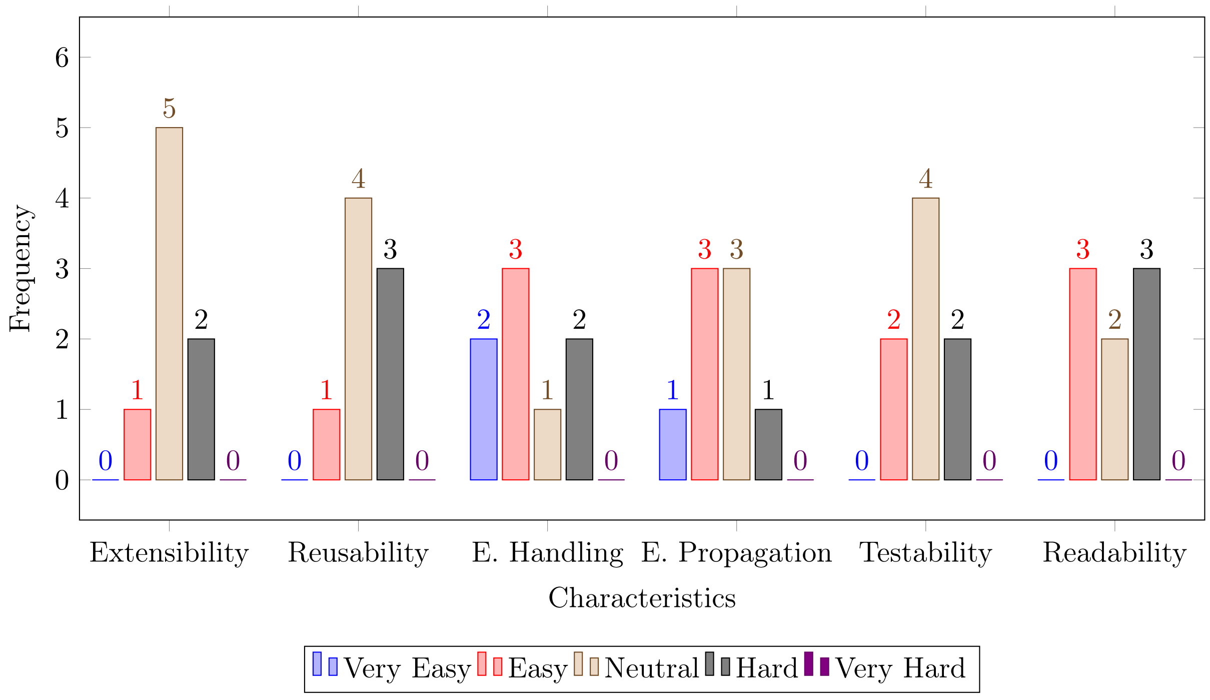
7.1.3 Question #3

Question 3 gathers feedback on the Kotlin and Scala implementations of the logic responsible for initiating an investment.

Kotlin



Scala

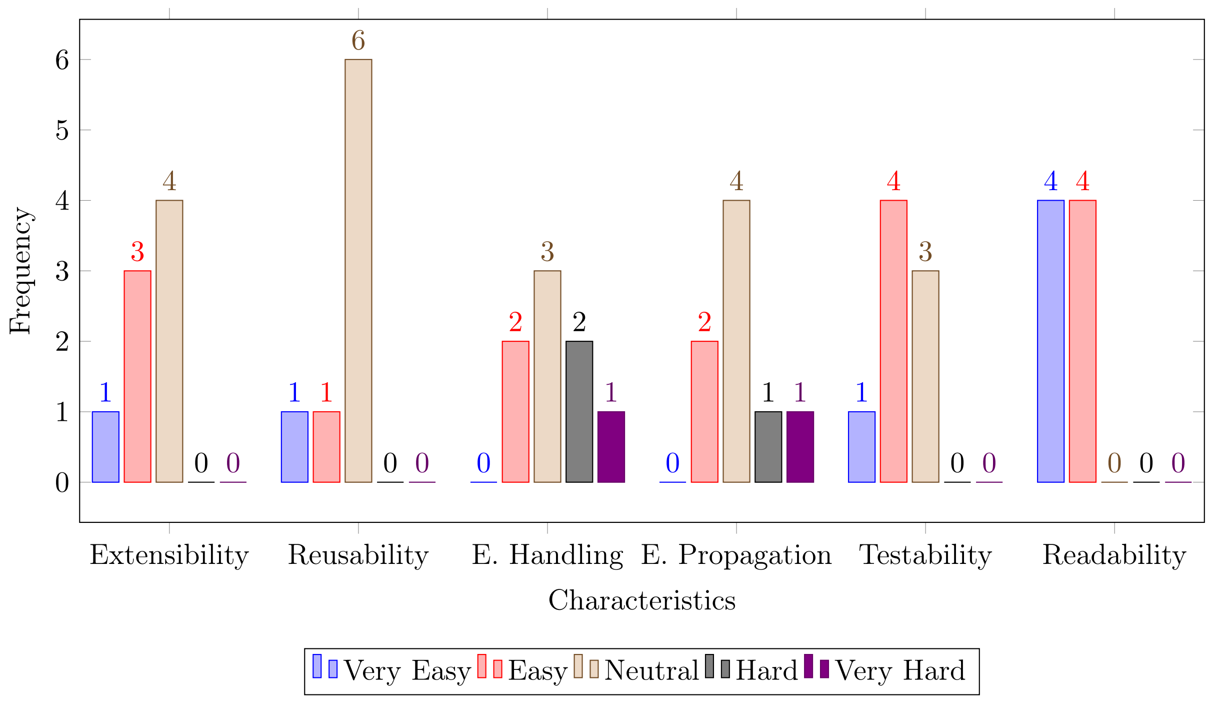


The results of Question 3 indicate that extensibility is slightly favored in the Scala implementation compared to the Kotlin implementation. Error handling and error propagation are comparable between the two, though some responses suggest these aspects may be more challenging to achieve in Scala. Reusability is also quite similar for both implementations, with a slight preference for Scala. Testability demonstrates a significant advantage in favor of the Kotlin implementation. Readability, while inconclusive for the Scala implementation, received more favorable feedback for the Kotlin version.

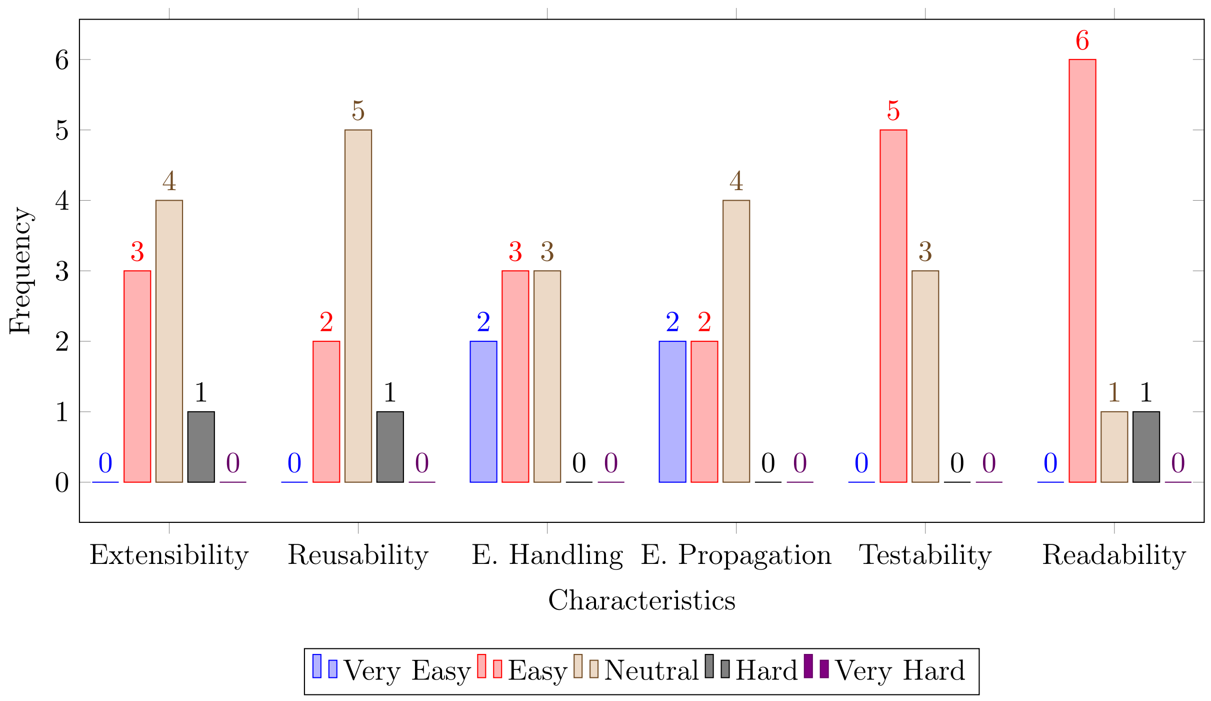
7.1.4 Question #4

Question 4 gathers feedback on the Kotlin and Scala implementations represent a Transfer.

Kotlin



Scala



The results of Question 4 suggest that extensibility is slightly preferred in the Kotlin implementation over the Scala implementation. Error handling and error propagation, on the other hand, were perceived more favorably in Scala. Reusability and testability were largely comparable between the two, with a slight preference for Kotlin. Both implementations were considered easy to read for this specific example, though readability was slightly more favored in Kotlin.

Chapter 8

Discussion

This chapter discusses the findings derived from both the qualitative and quantitative analyses conducted for each architectural characteristic outlined in the methodology (chapter 2).

8.1 Extensibility

This research assessed how object-oriented and functional paradigms contribute to the development of a Digital Wallet System designed for ease of expansion.

The qualitative analysis revealed that both Kotlin and Scala offer features that promote extensibility. Kotlin leveraged object-oriented principles such as inheritance and polymorphism to achieve extensibility, while Scala employed enumerations and pattern matching to accomplish the same objective. Although the Scala implementation was more concise, requiring less boilerplate code to extend a model, both approaches were effective and provided comparable benefits to the extensibility characteristic.

The quantitative analysis corroborated these findings. The perceptions of Scala and Kotlin implementations were similar, with only minor differences observed. Notably, these differences mostly favored the Kotlin implementation of the Digital Wallet System.

8.2 Reusability

This research assessed how object-oriented and functional paradigms promote flexibility and reduce the need for further redesign by enabling code reusability.

The qualitative analysis examined the object-oriented and functional paradigms by evaluating two distinct examples. The findings for both were consistent: Scala demonstrated stronger support for code reusability through higher-order functions, enabling generic implementations of various functionalities. In contrast, the Kotlin implementations in both examples lacked sufficient flexibility to be effectively adapted for different domains.

However, the quantitative analysis did not reveal significant differences between Kotlin and Scala in terms of reusability. Responses indicated that both implementations were perceived as similar, with only minor variations.

The discrepancy between the qualitative and quantitative analysis findings suggests that the reusability characteristic may considerably depend on the specific functionality being evaluated.

8.2.1 Error Handling and Propagation

This research assessed how object-oriented and functional paradigms handle error detection, management, and propagation.

The qualitative analysis demonstrated that the Scala implementation contributed more effectively to the robustness of the Digital Wallet System than the Kotlin implementation for the example under consideration. Kotlin propagates errors using exceptions and handles them with traditional `try-catch` blocks. In contrast, Scala utilizes the `Either[A, B]` type, representing a disjoint union of two types, to propagate errors. Error handling in Scala is performed through higher-order functions, allowing mapping to new error types as needed.

By incorporating errors directly into return types, the Scala implementation enforces comprehensive error handling through type checking, ensuring that new error cases cannot be missed.

Nevertheless, the quantitative analysis did not reflect the same dominance of the Scala implementation. Most responses indicated that Kotlin is either comparable or slightly superior to Scala in terms of error handling and propagation.

This divergence may suggest the influence of readability characteristic. In the Scala implementation, treating errors as values can cause them to blend with other elements of the code, potentially impacting clarity and comprehension. Conversely, Kotlin utilizes exceptions and `try-catch` blocks, which stand out from the rest of the code, effectively highlighting error management within the Digital Wallet System.

8.2.2 Testability

This research assessed how object-oriented and functional paradigms contribute to the correctness of the Digital Wallet System by facilitating the development of tests.

The qualitative analysis showed that Scala enabled more isolated testing by implementing transaction processing logic as pure functions, while side effects were implemented in a separate method. This approach enhanced test isolation but introduced an additional layer of testing compared to the Kotlin implementation.

The quantitative analysis, on the other hand, revealed that Kotlin was preferred in several examples, with developers perceiving it as more intuitive to test in practice.

Overall, these findings suggest that while Scala's functional approach offers advantages in test isolation, Kotlin promotes more integration with side effects and aligns better with developers' expectations.

8.2.3 Readability

This research assessed how object-oriented and functional paradigms contribute to the legibility of the Digital Wallet System.

The qualitative analysis revealed that Scala effectively utilizes constructs like `for`-comprehension to compose the various steps required to fulfill a liquidation request. However, it falls in providing clarity in its error handling and propagation flow. In contrast, Kotlin's use of `Exceptions` and `try-catch` blocks makes error handling more distinct from the main logic, enhancing readability and making the flow of error handling and propagation easier to understand.

The quantitative analysis relatively corroborated the findings of the qualitative analysis. With a few exceptions, the Kotlin implementation was consistently perceived as easier to read and reason about by the developers who answered the survey.

Chapter 9

Conclusion

As the functional programming paradigm has gained relevance after decades of dominance by object-oriented programming, this research aims to comparing how object-oriented programming and functional programming paradigms impact the architectural characteristics of systems, contributing with valuable insights into how programming paradigms influence architectural characteristics in software systems.

9.1 Main results

This research facilitated the comparison of object-oriented programming and functional programming paradigms, addressing the research questions outlined in Chapter 1.

The first research question, **RQ1**, aimed to establish a well-defined set of architectural characteristics to serve as the foundation for comparing the two paradigms. These architectural characteristics, detailed in Chapter 2, include extensibility, reusability, error handling, error propagation, testability, and readability.

The second research question, **RQ2**, aimed to determine how object-oriented and functional programming paradigms impact the architectural characteristics defined in **RQ1**. To answer this, the research employed a mixed-methods approach, involving both qualitative and quantitative analyses, followed by a synthesis of findings in a final discussion.

The qualitative analysis, detailed in Chapter 6, examined the implementation of specific functionalities within the paradigms. Each architectural characteristic was meticulously compared, revealing the strengths and weaknesses of each paradigm for a given functionality. This analysis provided valuable insights into which paradigm better satisfies a particular architectural characteristic or if they are relatively equivalent.

The quantitative analysis, presented in Chapter 7, involved a survey to gather feedback from developers with diverse backgrounds. This analysis complemented the qualitative findings by adding perspectives from real-world experience.

Finally, the combined results of the qualitative and quantitative analyses were discussed in Chapter 8. This discussion explored the similarities and differences between the two analyses, exploring potential factors contributing to significant discrepancies and offering deeper insights into how each paradigm addresses the defined architectural characteristics.

9.2 Future work

This work offers opportunities for further expansion to yield even more comprehensive results. For instance, future iterations could increase the scope of the current research questions. The list of architectural characteristics could be expanded to include non-functional requirements, such as scalability and security. Introducing new requirements could uncover the analysis of object-oriented or functional concepts that were not addressed in the current research, such as dependency inversion and laziness.

Additionally, the research could be enhanced by gathering more responses through the survey. This would strengthen the validity of the findings by incorporating more practical experiences from the real-world.

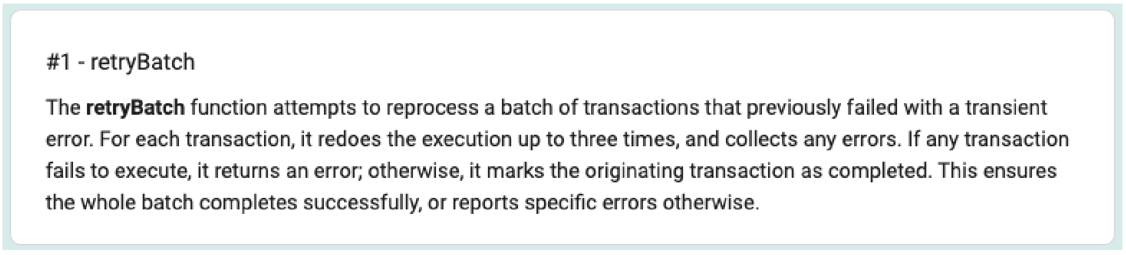
The research could also be enhanced by developing additional study objects that explore domains beyond the Digital Wallet System. This would provide a broader perspective on how object-oriented and functional programming paradigms perform across different contexts, enriching the qualitative and quantitative analysis.

Appendix A

Survey

A.1 Questionnaire

A.1.1 Question #1



#1 - retryBatch

The **retryBatch** function attempts to reprocess a batch of transactions that previously failed with a transient error. For each transaction, it redoes the execution up to three times, and collects any errors. If any transaction fails to execute, it returns an error; otherwise, it marks the originating transaction as completed. This ensures the whole batch completes successfully, or reports specific errors otherwise.

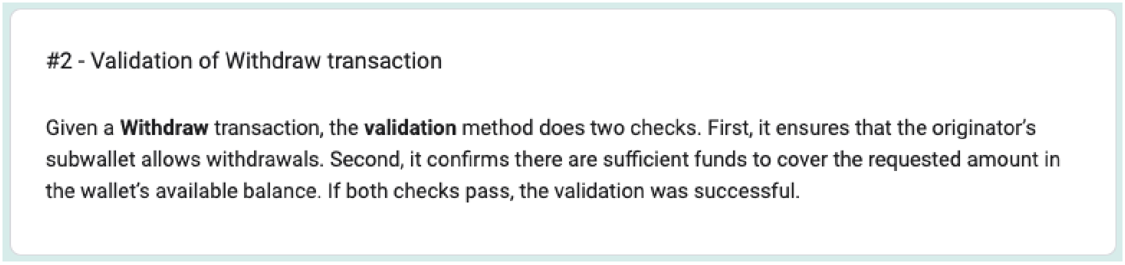
Kotlin

```kotlin
suspend fun retryBatch(
        batchId: String,
        n: Int,
    ) {
        val transactions = transactionsRepo.find(
            TransactionFilter(
                batchId = batchId,
                status = TransactionStatus.TRANSIENT_ERROR
            ))
        var allCompleted = true

        for (transaction in transactions) {
            var attempts = 0
            var success = false

            while (attempts < n && !success) {
                attempts++
                try {
                    transaction.process(
                        transactionsRepo,
                        ledgerService,
                        partnerService
                    )
                    transaction.updateStatus(
                        transactionsRepo, TransactionStatus.COMPLETED
                    )
                    success = true
                } catch (e: PartnerException) {
                    break
                }
            }

            if (!success) {
                allCompleted = false
            }
        }

        if (allCompleted) {
            val originalTransaction = transactionsRepo.find(
                TransactionFilter(
                    idempotencyKey = batchId
                )).firstOrNull()
            originalTransaction?.updateStatus(
                transactionsRepo,
                TransactionStatus.COMPLETED
            )
        }
    }
```

Scala

```

def retryBatch(batchId: String): Either[TransactionServiceError, Unit] = {
  transactionsRepo
    .find(TransactionFilter(
      batchId = Some(batchId),
      status = Some(TransactionStatus.TransientError)
    ))
    .traverse { t =>
      process(t).left.map { e =>
        // We are not really expecting a process error
        // as we know it worked the first time
        ProcessError(e.message)
      }
    }
    .flatMap(tuples => {
      val failures =
        tuples
          .map { tuple => lib.retry(() => execute(tuple), 3) }
          .collect { case Left(error) => error }

      if (failures.nonEmpty) {
        Left(ExecutionError(s"Could not execute batch successfully."))
      }
      else {
        for {
          originatingTransaction <-
            transactionsRepo
              .find(TransactionFilter(idempotencyKey = Some(batchId)))
              .headOption
              .toRight(
                TransactionServiceInternalError(s"Could not find transaction")
              )
        } yield {
          updateStatus(originatingTransaction.id, TransactionStatus.Completed)
        }
      }
    })
}
```

A.1.2 Question #2


#2 - Validation of Withdraw transaction

Given a **Withdraw** transaction, the **validation** method does two checks. First, it ensures that the originator's subwallet allows withdrawals. Second, it confirms there are sufficient funds to cover the requested amount in the wallet's available balance. If both checks pass, the validation was successful.

Kotlin

```kotlin
abstract class Transaction(
    val id: String,
    val batchId: String? = null,
    val amount: BigDecimal,
    val idempotencyKey: String,
    val originatorWalletId: String,
    val originatorSubwalletType: SubwalletType,
    var status: TransactionStatus,
) {
    abstract fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    )

    fun validateExternalTransaction() {
        if (this.originatorSubwalletType != SubwalletType.REAL_MONEY) {
            throw ExternalTransactionValidationException("Transaction not allowed")
        }
    }

    fun validateBalance(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        val walletId = this.originatorWalletId
        val wallet = walletsRepo.findById(walletId)
            ?: throw NoSuchElementException("Wallet not found")

        val balance = wallet.getAvailableBalance(ledgerService)

        if (this.amount > balance) {
            throw InsufficientFundsException("Wallet has no sufficient funds")
        }
    }

    // [...]
}

//////////

class Withdraw(
    id: String,
    batchId: String? = null,
    amount: BigDecimal,
    idempotencyKey: String,
    originatorWalletId: String,
    originatorSubwalletType: SubwalletType,
    status: TransactionStatus,
) : Transaction(
        id,
        batchId,
        amount,
        idempotencyKey,
        originatorWalletId,
        originatorSubwalletType,
        status,
    ) {
    override fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        validateExternalTransaction()
        validateBalance(walletsRepo, ledgerService)
    }

    // [...]
}
```

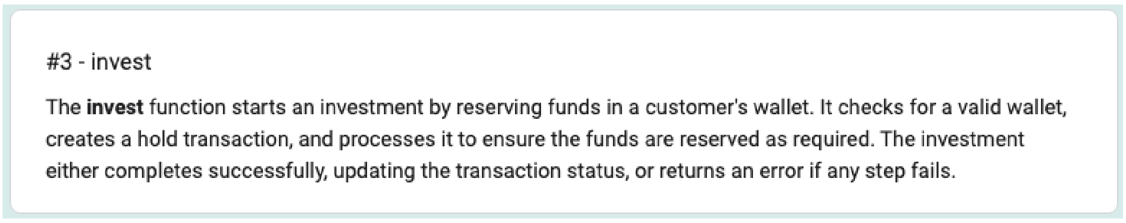
Scala

```
class TransactionValidationService(
    walletsRepo: WalletsDatabase,
    walletsService: WalletsService,
    ledgerService: LedgerService
  ) {
```

```scala
def validateTransaction(
  transaction: Transaction
): Either[TransactionValidationError, Unit] = {
  transaction.transactionType match {
    case TransactionType.Deposit => validateDeposit(transaction)
    case TransactionType.Withdraw => validateWithdraw(transaction)
    case TransactionType.Hold => validateHold(transaction)
    case TransactionType.Transfer => validateTransfer(transaction)
    case TransactionType.TransferFromHold => validateTransferFromHold(transaction)
  }
}

private def validateWithdraw(
  transaction: Transaction
): Either[TransactionValidationError, Unit] = {
  for {
    _ <- validateOriginatorSubwalletType(
          transaction.originatorSubwalletType,
          List(SubwalletType.RealMoney)
         )
    _ <- validateBalance(
          transaction.originatorWalletId,
          transaction.amount
        )
  } yield ()
}

private def validateOriginatorSubwalletType(
  originatorSubwalletType: SubwalletType,
  validSubwalletTypes: List[SubwalletType]
  ): Either[TransactionValidationError, Unit] =
  if (!validSubwalletTypes.contains(originatorSubwalletType)) {
    Left(OriginatorSubwalletTypeValidationError(s"Transaction not allowed"))
  } else {
    Right(())
  }

private def validateBalance(
  originatorWalletId: String,
  amount: BigDecimal
): Either[TransactionValidationError, Unit] =
  for {
    wallet <- walletsRepo.findById(originatorWalletId)
      .toRight(TransactionValidationFailed(s"Wallet not found"))

    balance = walletsService.getAvailableBalance(wallet)

    result <- if (amount > balance) {
      Left(InsufficientFundsValidationError(s"Wallet has no sufficient funds"))
    } else {
      Right(())
    }
  } yield ()
```

```
}
```

A.1.3 Question #3



#3 - invest

The **invest** function starts an investment by reserving funds in a customer's wallet. It checks for a valid wallet, creates a hold transaction, and processes it to ensure the funds are reserved as required. The investment either completes successfully, updating the transaction status, or returns an error if any step fails.

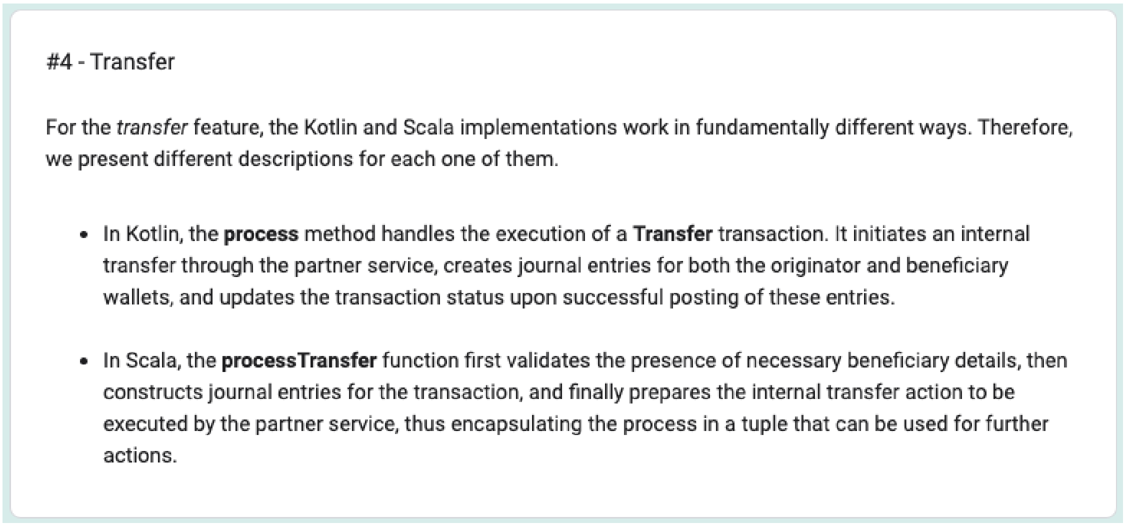
Kotlin

```
def invest(
        request: InvestmentRequest
    ): Either[InvestmentFailedError, Transaction] = {
    val wallets =
      walletsRepo.find(
        WalletFilter(
          customerId = Some(request.customerId),
          walletType = Some(WalletType.RealMoney)
        )
      )

    wallets match {
      case List(wallet) =>
        for {
          transaction <- transactionsService.create(
            CreateTransactionRequest(
              amount = request.amount,
              idempotencyKey = request.idempotencyKey,
              originatorWalletId = wallet.id,
              originatorSubwalletType = SubwalletType.RealMoney,
              transactionType = TransactionType.Hold
            )
          ).left.map { e =>
            InvestmentFailedError(e.message)
          }

          processTransactionTuple <-
            transactionsService
                .process(transaction)
                .left
                .map { e =>
                  transactionsService.updateStatus(
                    transaction.id,
                    TransactionStatus.Failed
                  )
                  InvestmentFailedError(e.message)
                }

          executedTransaction <-
            transactionsService
                .execute(processTransactionTuple)
                .left
                .map(e =>
                  InvestmentFailedError(e.message)
                 )
        } yield executedTransaction

      case _ =>
        Left(InvestmentFailedError(s"None or multiple wallets found"))
    }
  }
```

Scala

```scala
class TransactionValidationService(
    walletsRepo: WalletsDatabase,
    walletsService: WalletsService,
    ledgerService: LedgerService
  ) {
  def validateTransaction(
    transaction: Transaction
  ): Either[TransactionValidationError, Unit] = {
    transaction.transactionType match {
      case TransactionType.Deposit => validateDeposit(transaction)
      case TransactionType.Withdraw => validateWithdraw(transaction)
      case TransactionType.Hold => validateHold(transaction)
      case TransactionType.Transfer => validateTransfer(transaction)
      case TransactionType.TransferFromHold => validateTransferFromHold(transaction)
    }
  }

  private def validateWithdraw(
    transaction: Transaction
  ): Either[TransactionValidationError, Unit] = {
    for {
      _ <- validateOriginatorSubwalletType(
            transaction.originatorSubwalletType,
            List(SubwalletType.RealMoney)
          )
      _ <- validateBalance(
            transaction.originatorWalletId,
            transaction.amount
          )
    } yield ()
  }

  private def validateOriginatorSubwalletType(
    originatorSubwalletType: SubwalletType,
    validSubwalletTypes: List[SubwalletType]
    ): Either[TransactionValidationError, Unit] =
    if (!validSubwalletTypes.contains(originatorSubwalletType)) {
      Left(OriginatorSubwalletTypeValidationError(s"Transaction not allowed"))
    } else {
      Right(())
    }

  private def validateBalance(
    originatorWalletId: String,
    amount: BigDecimal
  ): Either[TransactionValidationError, Unit] =
    for {
      wallet <- walletsRepo.findById(originatorWalletId)
        .toRight(TransactionValidationFailed(s"Wallet not found"))

      balance = walletsService.getAvailableBalance(wallet)

      result <- if (amount > balance) {
        Left(InsufficientFundsValidationError(s"Wallet has no sufficient funds"))
      } else {
        Right(())
      }
    } yield ()
}
```

A.1.4 Question #4



#4 - Transfer

For the *transfer* feature, the Kotlin and Scala implementations work in fundamentally different ways. Therefore, we present different descriptions for each one of them.

- In Kotlin, the **process** method handles the execution of a **Transfer** transaction. It initiates an internal transfer through the partner service, creates journal entries for both the originator and beneficiary wallets, and updates the transaction status upon successful posting of these entries.
- In Scala, the **processTransfer** function first validates the presence of necessary beneficiary details, then constructs journal entries for the transaction, and finally prepares the internal transfer action to be executed by the partner service, thus encapsulating the process in a tuple that can be used for further actions.

Kotlin

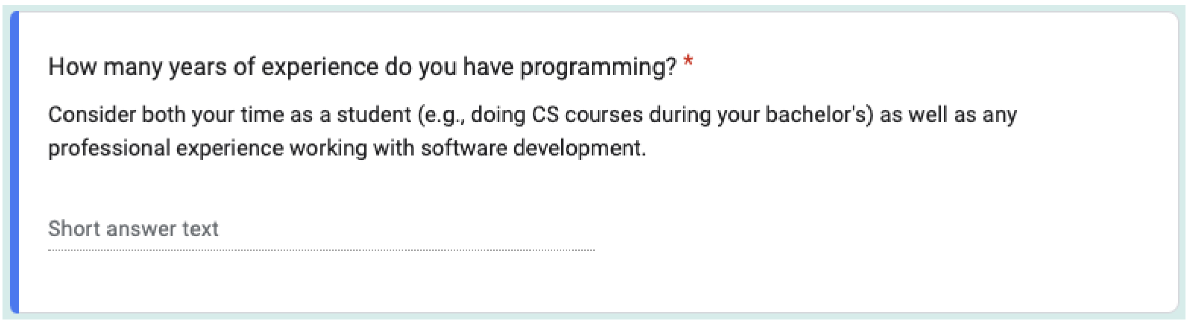
```
class TransactionsService(
    private val transactionsRepo: TransactionsDatabase,
    private val walletsRepo: WalletsDatabase,
    private val ledgerService: LedgerService,
    private val partnerService: PartnerService,
) {
    private val logger = Logger()

    suspend fun processTransaction(
        request: ProcessTransactionRequest
    ): Transaction {
        val transaction = transactionsRepo.insert(request)
        transaction.validate(walletsRepo, ledgerService)
        transaction.process(transactionsRepo, ledgerService, partnerService)

        return transaction
    }

    // [...]
}

/////////

class Transfer(
    id: String,
    batchId: String? = null,
    amount: BigDecimal,
    idempotencyKey: String,
    originatorWalletId: String,
    originatorSubwalletType: SubwalletType,
    status: TransactionStatus,
    val beneficiaryWalletId: String,
    val beneficiarySubwalletType: SubwalletType,
) : Transaction(
        id,
        batchId,
        amount,
        idempotencyKey,
        originatorWalletId,
        originatorSubwalletType,
        status,
    ) {
    override fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        val originatorSubwalletType = this.originatorSubwalletType
        val beneficiarySubwalletType = this.beneficiarySubwalletType

        val validTransferPairs =
            setOf(
                SubwalletType.REAL_MONEY to SubwalletType.EMERGENCY_FUND,
                SubwalletType.EMERGENCY_FUND to SubwalletType.REAL_MONEY,
            )

        if ((originatorSubwalletType to beneficiarySubwalletType)
            !in validTransferPairs) {
            throw TransferNotAllowed("Transfer not allowed")
        }

        validateBalance(walletsRepo, ledgerService)
```

```kotlin
    }

    override suspend fun process(
        transactionsRepo: TransactionsDatabase,
        ledgerService: LedgerService,
        partnerService: PartnerService,
    ) {
        partnerService.executeInternalTransfer(this)

        val journalEntries =
            listOf(
                CreateJournalEntry(
                    walletId = this.originatorWalletId,
                    subwalletType = this.originatorSubwalletType,
                    balanceType = BalanceType.AVAILABLE,
                    amount = -this.amount,
                ),
                CreateJournalEntry(
                    walletId = this.beneficiaryWalletId,
                    subwalletType = this.beneficiarySubwalletType,
                    balanceType = BalanceType.AVAILABLE,
                    amount = this.amount,
                ),
            )

        val postedAt = ledgerService.postJournalEntries(journalEntries)

        this.updateStatus(
            transactionsRepo,
            newStatus = TransactionStatus.COMPLETED,
            at = postedAt
        )
    }
}

```

Scala

```scala
class TransactionsService(
  transactionsRepo: TransactionDatabase,
  validationService: TransactionValidationService,
  partnerService: PartnerService,
  ledgerService: LedgerService,
) {
  private val lib = Library()

  // [...]

  def process(
    transaction: Transaction
  ): Either[ProcessError, ProcessTransactionTuple] = {
    lib.maybeLogError(() => {
      for {
        _ <- validationService
                .validateTransaction(transaction)
                .left.map(e => ProcessError(e.message))
        processTransactionTuple <-
            processTransaction(transaction)
            .left.map(e => ProcessError(e.message))
      } yield processTransactionTuple
    })
  }

  private def processTransaction(
    transaction: Transaction
   ): Either[TransactionServiceError, ProcessTransactionTuple] = {
    transaction.transactionType match {
      case TransactionType.Deposit => processDeposit(transaction)
      case TransactionType.Withdraw => processWithdraw(transaction)
      case TransactionType.Hold => processHold(transaction)
      case TransactionType.Transfer => processTransfer(transaction)
      case TransactionType.TransferFromHold => processTransferFromHold(transaction)
    }
  }


  private def processTransfer(
    transaction: Transaction
  ): Either[TransactionServiceError, ProcessTransactionTuple] = {
    for {
      beneficiaryWalletId <-
        transaction
        .beneficiaryWalletId
        .toRight(ProcessError(s"Transfer must contain beneficiaryWalletId"))
      beneficiarySubwalletType <-
        transaction
        .beneficiarySubwalletType
        .toRight(ProcessError(s"Wallet not found"))
    } yield {
      val journalEntries = List(
        CreateJournalEntry(
          walletId = Some(transaction.originatorWalletId),
          subwalletType = transaction.originatorSubwalletType,
          balanceType = BalanceType.Available,
          amount = -transaction.amount
        ),
        CreateJournalEntry(
          walletId = Some(beneficiaryWalletId),
          subwalletType = beneficiarySubwalletType,
          balanceType = BalanceType.Available,
          amount = transaction.amount
        )
      )

      val partnerAction: Action = partnerService.executeInternalTransfer
      (transaction, journalEntries, TransactionStatus.Completed, Some(partnerAction))
    }
  }

}
```

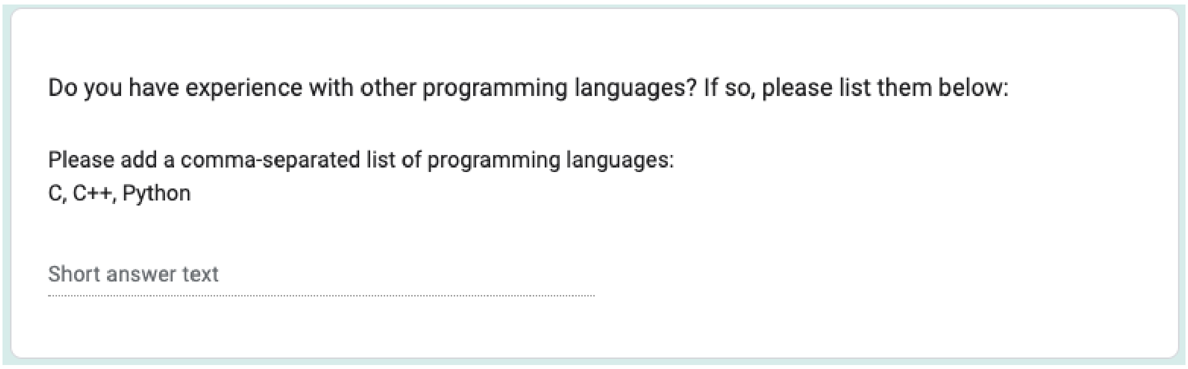
A.2 Background Questions



How many years of experience do you have programming? *

Consider both your time as a student (e.g., doing CS courses during your bachelor's) as well as any professional experience working with software development.

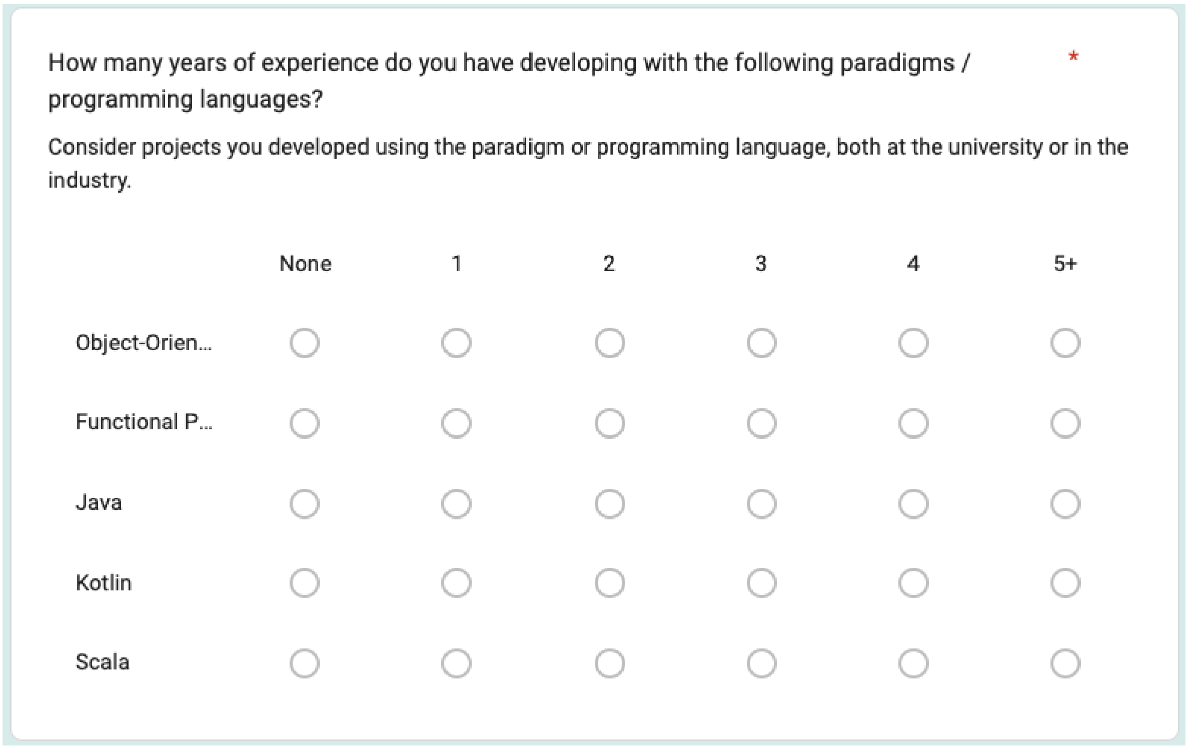
Short answer text



Do you have experience with other programming languages? If so, please list them below:

Please add a comma-separated list of programming languages:
C, C++, Python

Short answer text



How many years of experience do you have developing with the following paradigms / programming languages? *

Consider projects you developed using the paradigm or programming language, both at the university or in the industry.

	None	1	2	3	4	5+
Object-Orien...	○	○	○	○	○	○
Functional P...	○	○	○	○	○	○
Java	○	○	○	○	○	○
Kotlin	○	○	○	○	○	○
Scala	○	○	○	○	○	○

Bibliography

Ralph Johnson John Vlissides Erich Gamma, Richard Helm. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional, 1st edition, 1994. 5

Finley. Functional programming is finally going mainstream. URL https://github.com/readme/featured/functional-programming. 1, 2

Neal Ford Mark Richards. **Fundamentals of Software Architecture: An Engineering Approach**. O'Reilly Media, 1st edition, 2020. 2

Robert Martin. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Pearson, illustrated edition edition, 2017. 8

Rúnar Bjarnason Paul Chiusano. **Functional Programming in Scala**. Manning Publications, 1st edition, 2014. 9

Charles Scalfani. Why functional programming should be the future of software development. URL https://spectrum.ieee.org/functional-programming. 2