Functional vs. Object-Oriented: Comparing How Programming Paradigms Affect the Architectural Characteristics of Systems

Briza Mel Dias
University of São Paulo
São Paulo, Brazil
brizamel.dias@alumni.usp.br

Renato Cordeiro Ferreira
Jheronimus Academy of Data Science
's-Hertogenbosch, The Netherlands
University of São Paulo
São Paulo, Brazil
renatocf@ime.usp.br

Alfredo Goldman
University of São Paulo
São Paulo, Brazil
gold@ime.usp.br

ABSTRACT

This study compares the impact of adopting object-oriented programming (OOP) or functional programming (FP) on the architectural characteristics of software systems. For that, it examines the design and implementation of a Digital Wallet system developed in Kotlin (for OOP) and Scala (for FP). The comparison is made through a mixed-method approach. The self-ethnographic qualitative analysis provides a side-by-side comparison of both implementations, revealing the perspective of those writing such code. The survey-based quantitative analysis gathers feedback from developers with diverse backgrounds, showing their impressions of those reading this code. Hopefully, these results may be useful for developers seeking to decide which paradigm is best suited for their next project.

KEYWORDS

Software Architecture, Programming Paradigms, Object-Oriented Programming, Functional Programming, Empirical Software Engineering

1 Introduction

After decades of dominance of Object-Oriented Programming (OOP), the Functional Programming (FP) paradigm has gained significant attention in the software industry [4]. 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 the IEEE Spectrum magazine in 2022 [12], one of the clearest indications of this shift is the increasing incorporation of functional features into different mainstream programming languages. Evaluating how FP compares to OOP is essential to understanding the future of software development.

The main research question of this paper can be summarized as:

> **RQ:** How do the functional and object-oriented paradigms impact different architectural characteristics of a system?

According to Richards and Ford [10], an *architectural characteristic* describes a concern that follows three criteria:

(1) it specifies a non-domain design consideration,
(2) it influences structural aspects of the design, and
(3) it is critical or important to the application's success.

This research focuses on six architectural characteristics, summarized in table 1. They encompass concerns that highlight important differences between the paradigms, making it easier to compare their strengths and limitations.

Table 1: Evaluation Criteria for Qualitative Analysis

Architectural Characteristic	From the code, how easy it is to...
Extensibility	...add new behavior
Reusability	...apply it in a new use case
Error Handling	...understand errors handled by it
Error Propagation	...understand errors propagated by it
Testability	...test its intended behavior
Readability	...understand its intended behavior

Due to space constraints, this paper will address only four of the six architectural characteristics from table 1, whose analyses brought the most interesting results. For a full discussion, please refer to the bachelor thesis that originated this paper [2].

> **sRQ1:** How do the functional and object-oriented paradigms impact the extensibility of a system?

sRQ1 aims to explore how each paradigm influences the ability to add new components to the system. OOP typically relies on object composition and class inheritance, while FP uses high-order functions and generic data structures.

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

sRQ2 aims to explore how each paradigm influences the ability to reuse components in the system. OOP typically relies on design patterns that take advantage of polymorphism, while FP combines list-based immutable data types with closures and recursion.

> **sRQ3:** How do the functional and object-oriented paradigms impact error handling and propagation of a system?

sRQ3 aims to explore how each paradigm embodies distinct philosophies in error management. OOP typically uses exceptions, while FP applies monadic types.

2 Programming Paradigms

Understanding the principles of OOP and FP is essential for a meaningful comparison between them. Their distinct philosophies, strengths, and limitations significantly influence how software systems approach architectural characteristics.

2.1 Object-Oriented Programming

According to Martin [7], any OOP language must support these three characteristics: *encapsulation*, *inheritance*, and *polymorphism*.

Encapsulation. It ensures that a clear boundary is established around a cohesive set of data and behaviors [7]. 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. It is the process of reusing and extending a group of variables and functions defined in one scope within another [7]. In OOP, this often means that a new class (called a subclass) can inherit properties and behaviors from an existing class (called a superclass). This allows the new class to reuse, override, or expand existing functionality without redefining it entirely.

Polymorphism. It is the ability of a method to behave differently based on the context in which it is used [7]. 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.

2.2 Functional Programming

According to Pilquist et al. [9], functional programming is based on the premise of writing programs using only *pure functions*, which leads to other patterns like *high-order functions* and *monadic types*.

Pure Functions. They are functions without *side effects* [7]. 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 FP, functions are values [9]. They can be assigned to variables, stored in data structures, and passed as arguments to functions. A function that accepts other functions as arguments is called a higher-order function (HOF).

Monadic Types. Most FP languages support *pattern matching* [9] as a way to execute functions depending on the value passed to them. Constructs such as `Option` and `Either` have the properties of Monads [9], types that leverage pattern matching to provide clean composition and error handling.

3 Related Literature

Few works in the literature focus on the same research questions proposed in section 1. A search with the keywords "Functional versus Object-Oriented Programming" on on Google Scholar results in papers that propose to mix the paradigms in the context of a single programming language or application.

In the late 1990s, Harrison et al. [5, 11] compared the code of 12 programs produced in SML (FP) and C++ (OOP), measuring their difference on development and code metrics. In 2016, Alic et al. [1] compared algorithms in Java and C# (OOP) versus Haskell and F# (FP), focusing on lines of code and efficiency. In 2025, Khan et al. [6] compared the impact of choosing OOP, FP or Procedural Programming for implementing microservices architectures [8], analyzing their impact on application design.

The current gap in the literature is an opportunity to explore the differences between paradigms once again, using modern programming languages and focusing on both code and design.

4 Methodology

This section details the research methodology employed in this project, whose goal is to answer RQ.

4.1 Proof of Concept

The first phase of the research consists of the specification and development of a proof-of-concept system that can be used as an object of study. This provides a detailed example for comparing the object-oriented and functional paradigms.

A Digital Wallet was chosen as the proof-of-concept system for this research. It contains many functionalities, such as wallet management, transaction management, and calculation of balances. These operations demand intricate data handling to keep consistency, whose implementation can be approached differently within the object-oriented and functional paradigms.

To compare the paradigms, the same Digital Wallet system was developed *twice*, following the good practices of each paradigm.

Object-Oriented PoC. The object-oriented version of the Digital Wallet system was based on the Kotlin programming language, which supports the OOP principles from section 2.1. The developer can define classes in a straightforward manner, enabling the structuring of data and behavior through objects.

Functional PoC. The functional version of the Digital Wallet system was based on the Scala programming language, which supports classic FP principles from section 2.2. The developer can focus on immutability and pure functional constructs, aligning with the core principles of functional programming.

Section 5 describes the set of functional requirements that guided the development of both versions of the Digital Wallet system. Sections 6 and 7 describe the main business logic that was implemented in both proofs of concept. The code developed during the research is available in the paper's reproduction package.

4.2 Analysis

The second phase of the research consists of an analysis of the paradigms using a mixed-method approach. By triangulating benefits and drawbacks of each paradigm over the architectural characteristics, it is possible to provide an answer to RQ.

The analysis will be based on the implementation of the Digital Wallet system. In particular, the focus is on code snippets that perform the same task from both proofs of concept.

To provide a good perspective over the paradigms, this research proposes to combine two research techniques:

Self-Ethnographic Qualitative Analysis. It represents the perspective of *writing code* in the different paradigms. In particular, the self-ethnographic methodology means that it presents the assessments of the authors of the paper, who developed the proofs of concept. These results are further explored in section 8.

Survey-Based Quantitative Analysis. It represents the perspective of *reading code* in the different paradigms. In particular, the survey-based methodology means that it summarizes the assessments of different developers with varying levels of experience with the paradigms. These results are further explored in section 9.

After presenting these results, section 10 combines these perspectives to answer the research subquestions.

5 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 as a proof of concept whose functionalities highlight the differences between the paradigms. Therefore, the Digital Wallet system ignores some important non-functional requirements for this type of system, such as security and auditability.

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

5.1 Wallet Management

The Digital Wallet system should provide each customer with three different types of wallets:

Real Money Wallet. It represents a real-world checking account owned by the customer. This wallet is the entry 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. It represents 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 the other way around. Investments and liquidations can be initiated at any time but will only settle on the next business day.

Emergency Funds Wallet. It represents 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.

5.2 Transaction Lifecycle Management

The Digital Wallet system should support the creation, validation, and execution of transactions between wallets. Transactions represent the flow of money inside the Digital Wallet system, which might succeed or fail.

To provide customers with all the functionalities, the Digital Wallet system needs to support the following transactions:

- Deposit, which represents a deposit to the Real Money Wallet, modeling the entry of funds into the Digital Wallet system.
- Withdrawal, which represents a withdrawal from the Real Money Wallet, modeling the exit of funds from the Digital Wallet system.
- Transfer, which represents 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, which represents an amount of money reserved for future use. This transaction ensures funds until an application settles.
- Transfer From Hold, which represents an instant release of an amount of money previously reserved in a Hold. This transaction settles an application, using the available funds.

Figure 1 summarizes the compatibility between transactions and different wallets. Every transaction must be validated before execution. Deposit and Withdrawal transactions are restricted to the Real Money Wallet. Transfer transactions are allowed only between Real Money Wallet and Emergency Funds Wallet. Hold transactions can be placed only on Real Money Wallet and Investment Wallet, whereas Transfer From Hold transactions are limited to being executed between them. Finally, all transactions – except for Deposit – require a balance check to ensure sufficient funds are available to complete the operation.

The Digital Wallet system must 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 allow to retry the transaction.

5.3 Investment Customization

The Digital Wallet system should allow customizing investments and liquidations. The participation should be a percentage of the total invested under an Investment Wallet, 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.

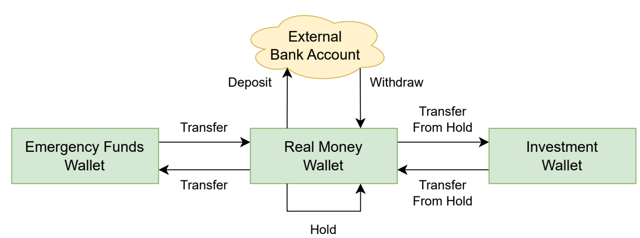

Figure 1: Transaction Lifecycle.

5.4 Minor Requirements

Financial Tracking. The Digital Wallet system must track all financial transactions to maintain accurate account balances and ensure data integrity.

Investment and Liquidation Settlement. The Digital Wallet system must settle all investment and liquidation requests initiated on the previous business day.

Batch Processing of Transactions. The Digital Wallet system must support batch processing of transactions with atomicity. 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.

6 Digital Wallet System: Entities

This section presents entities and value objects [3] representing the core business concepts of the Digital Wallet system.

6.1 Wallet

The `Wallet` entity models an account where customer funds are held and managed. It functions as the hub for all financial transactions, allowing money to be deposited or withdrawn to other destinations. The Digital Wallet system provides three types of wallets, as specified in section 5.1: `Real Money Wallet`, `Investment Wallet`, and `Emergency Fund Wallet`.

While `Transaction` entities dictate the flow of funds between `Wallets` entities, the latter are unaware of the former's existence. Their role is to model and hold customer funds within the system. They are not responsible for determining the validity of any incoming or outgoing `Transaction`. This separation of concerns ensures that `Wallet` entities focus on balance management without handling `Transaction` logic.

6.2 Subwallet

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

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

The relevance of the `Subwallet` entity becomes evident in the `Investment Wallet`, where funds are allocated across various investment options, as illustrated by figure 2. Introducing a new investment option involves composing new `Subwallet` entities within the `Investment Wallet`.

6.3 Transaction

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

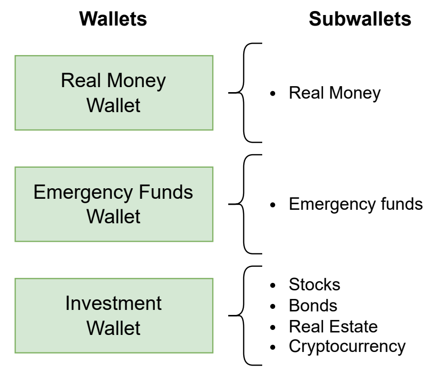

Figure 2: `Wallets` and corresponding `Subwallets`.

Transaction Types. A `Transaction` can be classified as a *Deposit*, *Withdrawal*, *Hold*, *Transfer*, or *Transfer From Hold*. Each transaction type represents a standard operation that modifies the state of a `Wallet` as specified in section 5.2.

Transaction validation. A `Transaction` must be validated to ensure they adhere to the internal rules of the Digital Wallet system. For example, a *Deposit* and *Withdrawal* can only be initiated on a `Real Money Wallet`, but not on an `Investment Wallet`. These validations follow the restrictions specified in section 5.2.

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

6.4 Journal Entry

A `Journal Entry` entity documents the effects of a `Transaction` on a `Wallet`. It tracks changes in the `Wallet` balance. By listing all `Journal Entry` entities linked to a `Wallet`, it is possible to calculate its current state accurately.

`Journal Entry` entities are always created in pairs to represent the source and destination of funds. To accurately reflect changes in a `Wallet`, a `Journal Entry` includes the following:

- `wallet_id`: It identifies the source or target `Wallet`. It may be absent to represent an external bank account.
- `subwallet_id`: It identifies the `Subwallet` under a Wallet. It may also be absent to represent an external bank account.
- `amount`: It accounts for the money being added (if positive) or deducted (if negative) from the `Subwallet`.
- `balance_type`: It can have three values: *Internal*, for funds sent to or received from an external bank account; *Available*, for funds added to or deducted from the available balance of a `Subwallet`; and *Holding*, for funds added to or deducted from the reserved balance of a `Subwallet`.

The `Ledger` entity collects all `Journal Entry` entity created by the Digital Wallet system. Each `Transaction` with *Completed* status results in a pair of `Journal Entry` being recorded in the `Ledger`. Figure 3 illustrates how these journal entries look like for each transaction type.

6.5 Investment Policy

An `Investment Policy` entity defines the allocation of an investment portfolio by associating each `Subwallet` with a specific percentage representing its share of the total portfolio, as specified in section 5.3.

Only a `Subwallet` within the `Investment Wallet` is permitted to be included in an `Investment Policy`.

7 Digital Wallet System: Services

The Digital Wallet system relies on a set of services that collectively operate the functionalities described in section 5.

Figure 4 summarizes the existing services and illustrates the communication flow between them.

7.1 Ledger Service

The `Ledger` service is responsible for orchestrating between other services and the Digital Wallet system `Ledger` entity. It provides two functionalities: (1) posting new `Journal Entry`, and (2) querying the `Ledger` for its `balance`. For the latter, it aggregates `Journal Entry` to compute the balance of a specific `Wallet`.

7.2 Investment Service

The `Investment` service 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 `Investment Policy` to initiate and manage the transactions required to complete an investment or liquidation.

Figure 3: Pairs of `Journal Entry` for each `Transaction` type.

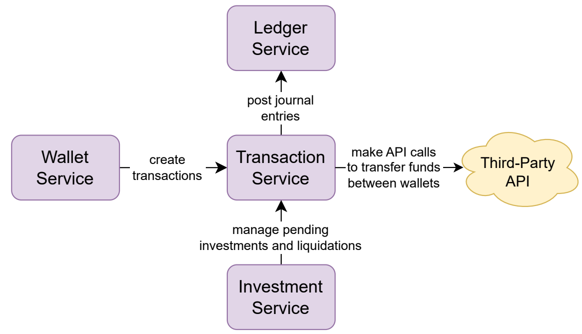

Figure 4: Services that compose the Digital Wallet system.

7.3 Transaction Service

The `Transactions` service provides functionalities to fulfill each step of a transaction lifecycle specified in section 5.2. 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 `Transactions` service 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.

7.4 Wallet Service

The `Wallets` service is responsible for managing and initiating `Transaction` s within `Wallet` s. It encapsulates the business logic needed to translate money movements initiated by the customer into a series of `Transaction` s that fulfill these requests.

The types of requests handled by the `Wallets` service include:

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

8 Qualitative Analysis

The qualitative analysis compares functionally equivalent code snippets from the Kotlin and Scala proofs of concept, evaluating them according to the architectural characteristics focused on by the research sub-questions. This methodology is self-ethnographic, since it presents the assessments of the authors who led the development process.

8.1 Extensibility [sRQ1]

To analyze the *Extensibility* quality attribute, this section examines the Kotlin and Scala implementations of a `Wallet` in the Digital Wallet system.

Kotlin. In Kotlin, the `Wallet` entity is structured using OOP principles like 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`. This is illustrated in listing 1.

In Kotlin, `getAvailableBalance` is an abstract method of the `Wallet` class, which encapsulates the logic for querying balances from the ledger. All subclasses must implement their specific logic for computing balances. This ensures consistency across wallet types while allowing each subclass to customize its behavior. Such an approach leverages polymorphism: the behavior of a method depends on the subclass of the object being called.

A future requirement demanding a new `Wallet` 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 superclass while also being required to implement its abstract method, enabling it to define specific balance-query behavior.

Scala. In Scala, the `Wallet` entity is structured using FP principles like immutable data structures. The *case class* `Wallet` consolidates all possible attributes and contains an additional field `WalletType` that distinguishes the wallet derivations within the system. This is illustrated in listing 2.

In Scala, `getAvailableBalance` is a standalone function in the `WalletService` class, which encapsulates the logic for querying balances from the ledger. For every variant of the `WalletType` enumeration, the function specifies a corresponding list of ledger queries to compute the available balance. This provides behavior tailored to each wallet type without requiring separate case classes. Such an approach leverages pattern matching: the `walletType` field maps to different balance computation logic.

A future requirement demanding a new `Wallet` could be achieved by simply adding a new `WalletType` enumeration. In Scala, pattern matching is exhaustive, meaning that any unhandled enumeration variant results in a compilation error. Therefore, pattern matching ensures that all standalone functions related to `WalletType` need to be revisited, ensuring it to define all balance-query behavior.

Kotlin vs. Scala. 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 class 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.

(Kotlin – Wallet)

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

(Kotlin – Real Money Wallet)

```kotlin
class RealMoneyWallet(  // subclass
  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)
  }
}
```

(Kotlin – Emergency Funds Wallet)

```kotlin
class EmergencyFundsWallet(  // subclass
  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)
  }
}
```

(Kotlin – Investment Wallet)

```kotlin
class InvestmentWallet(  // subclass
  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)
  }
}
```

Listing 1: `Wallet` class *inheritance* hierarchy in Kotlin

(Scala − Wallet)

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

(Scala − WalletType)

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

(Scala − getAvailableBalance)

```
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)
}
```

Listing 2: `Wallet` type and factory *pure function* in Scala

8.2 Reusability [sRQ2]

To analyze the *Reusability* quality attribute, this section examines the Kotlin and Scala implementations of retrying a batch of transactions. Not all transaction failures within the Digital Wallet system are permanent. For example, transactions that pass validation but fail during execution are eligible for retry.

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 should fail. Conversely, if all transactions are successfully processed, the batch should be marked as successful.

Kotlin. In Kotlin, the implementation of the retry logic is built around a `while` loop. This is illustrated in listing 3. It leverages the counter variable `attempts` to track the number of retries for each transaction and the flag variable `success` to indicate whether the transaction processing succeeded.

This design lacks the flexibility to be reused for other operations, even though it provides the necessary control for transactions.

Scala. In Scala, the implementation of the `retry` logic is built around a higher-order function named `retry`. This is illustrated in listing 4. It takes a function `f: => Either[A, B]` and an integer `n` as parameters to execute `f` up to `n` times, early returning `Right` if any attempt succeeds, or returning `Left` if all `n` attempts fail.

This design encapsulates the retry logic into a reusable function, providing this functionality to any function that needs to be retried.

Kotlin vs. Scala. This example demonstrates how the concept of higher-order functions enables the reuse of the same logic across multiple operations in the Scala PoC, eliminating the need to replicate similar logic for different functions. By leveraging pattern matching and recursion, the Scala PoC also achieves a more concise and expressive code structure compared to the Kotlin PoC.

(Kotlin − retryBatch)

```
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
    )
  }
}
```

Listing 3: `retryBatch` built with a `for` loop in Kotlin

(Scala – retryBatch)

```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 => 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)
        }
      }
    }
}
```

(Scala – 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)
}
```

Listing 4: `retryBatch` built over the HOF `retry` in Scala

8.3 Error Handling and Propagation [sRQ3]

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` entity to the `Investment Wallet` entity. This operation consists essentially of a hold `Transaction` that does not settle instantly.

Kotlin. In Kotlin, 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. This is illustrated in listing 5.

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

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

Scala. In Scala, errors are emitted via the monadic type `Either`. This approach allows them to be explicitly modeled as part of the method's return type, making it clear to developers that a method can fail. This is illustrated in listing 6.

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 promotes more predictable and reliable error handling within the system.

9 Quantitative Analysis

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

The draft survey was initially reviewed by a developer with experience in both paradigms. Then, it was beta-tested with two additional developers knowledgeable in both paradigms. After incorporating all feedback, the final version was made publicly available to a wider public.

The survey was divided into two sections:

Background. This section was designed to map the developers' experience with object-oriented and functional programming languages, since such experience might influence participants' reasoning when evaluating the code snippets.

(Kotlin – invest)

```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
        )
    }
  }
}
```

Listing 5: `invest` method using *exceptions* in Kotlin

```
( Scala - invest )
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}"
        )
      )
  }
}
```

Listing 6: invest function using *monadic types* in Scala

Code Analysis. This section was designed to measure the developer's impressions about the two Digital Wallet system proof of concepts. Each question includes a brief explanation of a functionality from the Digital Wallet system, followed by two code snippets: the first from the Kotlin PoC and the second from the Scala PoC. For each code snippet, participants were asked to evaluate the six architectural characteristics summarized in table 1. They answered the question: *"From the code, how easy is it to..."* using a Likert scale with five steps, from *Very Easy* to *Very Hard.*

At the publication of the bachelor thesis that inspired this paper [2], the survey received *eight responses*. Considering space constraints, this section will only address two of four questions proposed in the questionnaire, which intersect with some of the code previously analyzed in section 8. For a full analysis, please refer to the bachelor thesis [2].

The full questionnaire and corresponding responses can be found in the paper's reproduction package.

9.1 Question #1 — `retryBatch`

Question #1 collected feedback on the implementation of the `retryBatch`, whose code was explored in listings 3 and 4.

The results of Question #1 are shown in figure 5. They reveal a notable difference in the architectural characteristics of *error handling*, *error propagation*, and *readability*.

Many developers found the Scala implementation of the `retry` logic to be more challenging to handle, to propagate errors, and to read when compared to the Kotlin implementation. This indicates that, for these aspects, Kotlin may offer a clearer approach.

9.2 Question #3 — `invest`

Question #3 collected feedback on the implementation of the `invest` logic, whose code was explored in listings 5 and 6.

The results of the Question #3 are shown in figure 6. They reveal that *extensibility* is slightly favored in the Scala PoC compared to the Kotlin PoC. On the other hand, *testability* showed a significant advantage in favor of Kotlin.

Error handling and *error propagation* were comparable between both implementations, though some responses suggested these aspects might be more challenging to achieve in Scala. *Reusability* was also quite similar for both implementations, with a slight preference for Scala. *Readability*, while inconclusive for Scala, received more favorable feedback for Kotlin.

10 Discussion

Based on the findings from the qualitative and quantitative analyses, this section discusses the research subquestions.

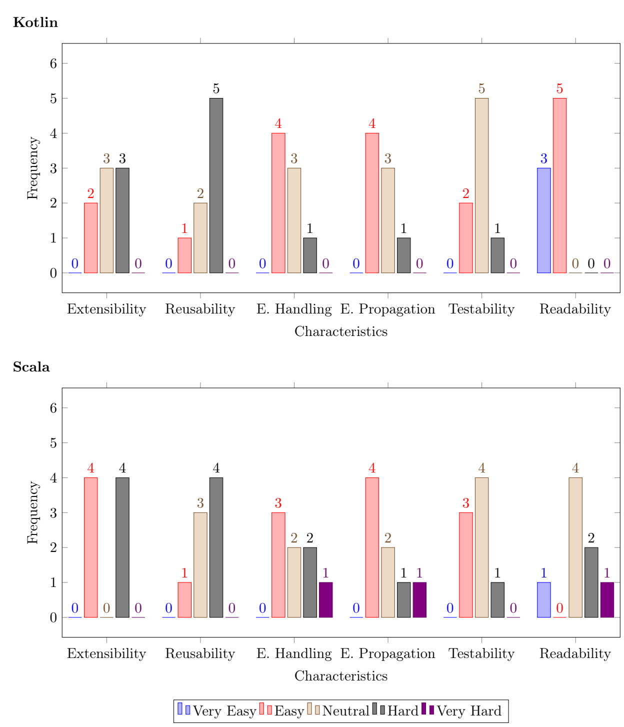

Figure 5: Kotlin vs. Scala — `retryBatch`

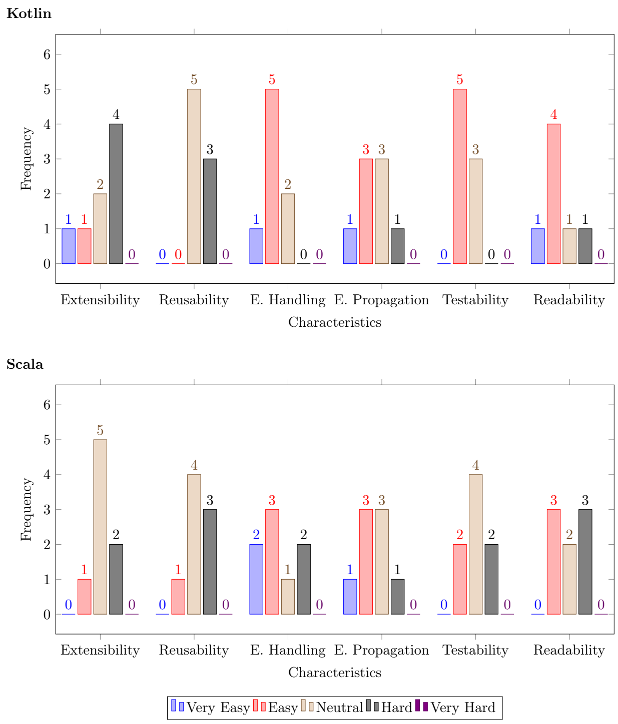

Figure 6: Kotlin vs. Scala — `invest`

> **sRQ1:** How do the functional and object-oriented paradigms impact the extensibility of a system?

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.

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

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.

> **sRQ3:** How do the functional and object-oriented paradigms impact error handling and propagation of a system?

Kotlin propagates errors using exceptions and handles them with traditional `try-catch` blocks. In contrast, Scala utilizes the monadic type `Either[A, B]` 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.

11 Threats to Validity

The threats to validity acknowledge the limitations of this research.

Construct Validity. 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, multi-platform languages that run on the Java Virtual Machine (JVM), ensuring a level of syntactic compatibility between the programming languages. Despite Kotlin supporting FP features and Scala supporting OOP features, this research intentionally limited both to their selected domains.

External Validity. 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 was employed, incorporating quantitative analysis to complement the qualitative findings. Nevertheless, due to time constraints, the number of participants in the survey was limited. To strengthen the results, more participants should be included to triangulate the conclusions from this research.

12 Conclusion

> **RQ:** How do the functional and object-oriented paradigms impact different architectural characteristics of a system?

This paper compared the impact of adopting OOP versus FP over six architectural characteristics of software systems. It introduced the Digital Wallet system, implemented twice: once in Kotlin (OOP), once in Scala (FP). The use of these modern, popular programming languages for the research will hopefully reach the interest of practitioners, who may understand the results and then be better informed when choosing a certain paradigm.

For future work, this research can be extended to target other architectural characteristics, such as scalability and security. Additionally, the current results can be enhanced by expanding the survey, thus addressing some of the threats to validity.

REFERENCES

[1] Dino Alic, Samir Omanovic, and Vaidas Giedrimas. 2016. Comparative Analysis of Functional and Object-Oriented Programming. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, Piscataway, NJ, USA, 667–672. doi:10.1109/MIPRO.2016.7522224

[2] Briza Mel Dias, Alfredo Goldman, and Renato Cordeiro Ferreira. 2024. *Functional vs. Object-Oriented: Comparing how Programming Paradigms affect the architectural characteristics of systems*. Bachelor's Thesis. University of São Paulo (USP), São Paulo. https://web.archive.org/*/https://linux.ime.usp.br/~brizamel/mac0499/

[3] Eric Evans. 2003. *Domain-Driven Design* (1st ed.). Addison-Wesley Professional, Boston, MA. 1–529 pages.

[4] Finley. 2022. *Functional programming is finally going mainstream*. https://github.com/readme/featured/functional-programming

[5] R. Harrison, L.G. Samaraweera, M.R. Dobie, and P.H. Lewis. 1996. Comparing Programming Paradigms: An Evaluation of Functional and Object-Oriented Programs. *Software Engineering Journal* 11, 4 (1996), 247. doi:10.1049/sej.1996.0030

[6] Mustafa Ahmed Khan, Syed Shabeeb Raza, Khalid Mahboob, Sana Alam, Muhammad Asghar Khan, and Muhammad Noman Hasany. 2025. A Comparative Study of Object-Oriented, Procedural, and Functional Programming Paradigms in Microservice Architecture. *VFAST Transactions on Software Engineering* 13, 3 (9 2025), 176–186. doi:10.21015/vtse.v13i3.2216

[7] Robert C. Martin. 2017. *Clean Architecture* (1st ed.). Pearson, Boston, MA. 1–404 pages.

[8] Sam Newman. 2021. *Building Microservices* (2nd ed.). O'Reilly Media. 1–612 pages.

[9] Michael Pilquist, Rúnar Bjarnason, Paul Chiusano, Martin Odersky, and Daniel Spiewak. 2023. *Functional programming in Scala* (2nd ed.). Manning Publications, Shelter Island, NY. 1–488 pages.

[10] Mark Richards and Neal Ford. 2020. *Fundamentals of Software Architecture* (1st ed.). O'Reilly Media, Sebastopol, CA. 1–419 pages.

[11] L. G. Samaraweera and R. Harrison. 1998. Evaluation of the Functional and Object-Oriented Programming Paradigms. *ACM SIGSOFT Software Engineering Notes* 23, 4 (7 1998), 38–43. doi:10.1145/286366.286374

[12] Charles Scalfani. 2022. *Why Functional Programming Should Be The Future of Software Development*. https://spectrum.ieee.org/functional-programming