OOP vs FP - Architectural Characteristics

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

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

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

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

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

* Indica uma pergunta obrigatória

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.

Sua resposta

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-Oriented Programming	○	○	○	○	○	○
Functional Programming	○	○	○	○	○	○
Java	○	○	○	○	○	○
Kotlin	○	○	○	○	○	○
Scala	○	○	○	○	○	○

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

Sua resposta

Avançar

Página 1 de 2

Limpar formulário

OOP vs FP - Architectural Characteristics

renatocf@ime.usp.br Mudar de conta

Não compartilhado

* Indica uma pergunta obrigatória

Code Snippet Evaluation

In this section, we show implementations of **five features** of the same system. For each one of them, we provide a code snippet **first in Kotlin**, **then in Scala**.

After reading a code snippet, please evaluate it on the following architectural characteristics / non-functional requirements:

- **readability**, how easy is to *understand its intended behavior*
- **maintainability**, how easy is to *change its current behavior*
- **extensibility**, how easy is to *add new behavior*
- **testability**, how easy is to *test it*
- **reusability**, how easy is to *apply it in a new use case*
- **error handling**, how easy is to *understand errors handled by it*
- **error propagation**, how easy is to *understand errors created / propagated by it*

#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.

retryBatch - **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
            )
        }
    }
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

retryBatch - **Scala** *

```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)
          }
        }
      })
  }
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

#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.

Validation of Withdraw transaction - **Kotlin** *

```kotlin
class Withdraw(
    id: String,
    amount: BigDecimal,
    ...
) : Transaction(
        id,
        amount,
        ...
    ) {
    override fun validate(
        walletsRepo: WalletsDatabase,
        ledgerService: LedgerService,
    ) {
        validateExternalTransaction()
        validateBalance(walletsRepo, ledgerService)
    }
}

// ------------------------------------------------------------------------------- //

abstract class Transaction(
    val id: String,
    val amount: BigDecimal,
    ...
) {
  abstract fun validate(
    walletsRepo: WalletsDatabase,
    ledgerService: LedgerService,
  )

  fun validateExternalTransaction() {
    if (this.originatorSubwalletType != SubwalletType.REAL_MONEY) {
        throw ExternalTransactionValidationException("External 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")
    }
  }
}
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

Validation of Withdraw transaction - **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 ...
    }
  }

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

  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 ()
  }
}
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
reusability	○	○	○	○	○
testability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

#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.

invest - **Kotlin** *

```kotlin
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
          )
  }
}
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

invest - **Scala** *

```scala
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"))
  }
}
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

#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.

Transfer - **Kotlin** *

```kotlin
class Transfer(
    id: String,
    amount: BigDecimal,
    ...
) : Transaction(
        id,
        amount,
        ...
    ) {
    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)
    }
}
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○

Transfer - **Scala** *

```scala
private def processTransfer(transaction: Transaction):
  Either[TransactionServiceError, ProcessTransactionTuple] = {
    for {
      beneficiaryWalletId <- transaction.beneficiaryWalletId.toRight(
        ProcessError(s"Transfer ${transaction.id} must contain beneficiaryWalletId")
      )
      beneficiarySubwalletType <- transaction.beneficiarySubwalletType.toRight(
        ProcessError(s"Wallet ${transaction.beneficiaryWalletId} 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))
    }
}
```

	Very easy	Easy	Neutral	Hard	Very hard
readability	○	○	○	○	○
maintainability	○	○	○	○	○
extensibility	○	○	○	○	○
testability	○	○	○	○	○
reusability	○	○	○	○	○
error handling	○	○	○	○	○
error propagation	○	○	○	○	○