por Briza Mel Dias de Sousa

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

orientador: Alfredo Goldman Vel Lebjman
coorientador: Renato Cordeiro Ferreira

São Paulo, Dezembro 2024

MOTIVAÇÃO

Após décadas de dominância da **programação orientada a objetos** (POO), a **programação funcional** (PF) está ganhando popularidade na indústria.

Um dos indícios de que a indústria está se preparando para uma migração de paradigmas é a incorporação de características funcionais em linguagens de programação amplamente utilizadas.

OBJETIVO

Comparar como os paradigmas programação orientada a objetos e programação funcional afetam características arquiteturais de sistemas.

CARACTERÍSTICAS ARQUITETURAIS

1. Legibilidade
2. Manutenabilidade
3. Extensibilidade
4. Testabilidade
5. Reusabilidade
6. Propagação e tratamento de erros

METODOLOGIA

- Desenvolver um sistema de **Carteira Digital** em **Kotlin** (representando POO) e em **Scala** (representando PF)
- Análise Qualitativa
- Análise Quantitativa

FORMULÁRIO

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.

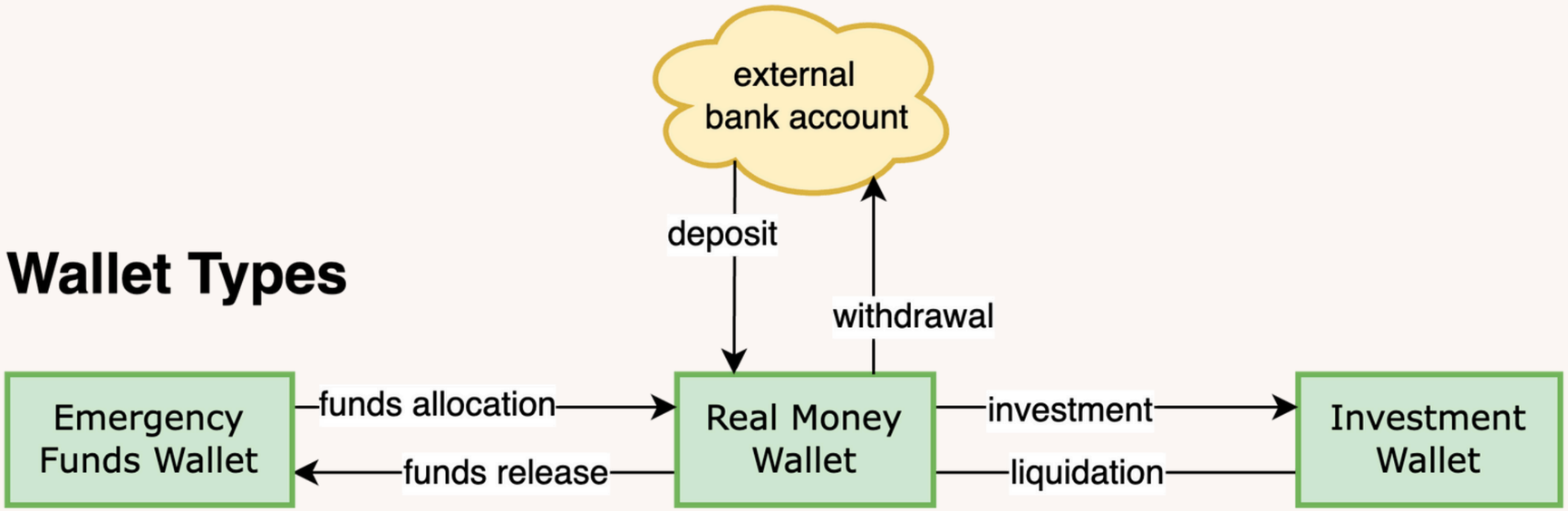
It aims to compare how the use of Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms affects the **understanding** of different architectural characteristics / non-functional requirements by developers. To conduct this comparison, the authors developed a Digital Wallet system 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]

CARTEIRA DIGITAL



Wallet Types

Investment Policy

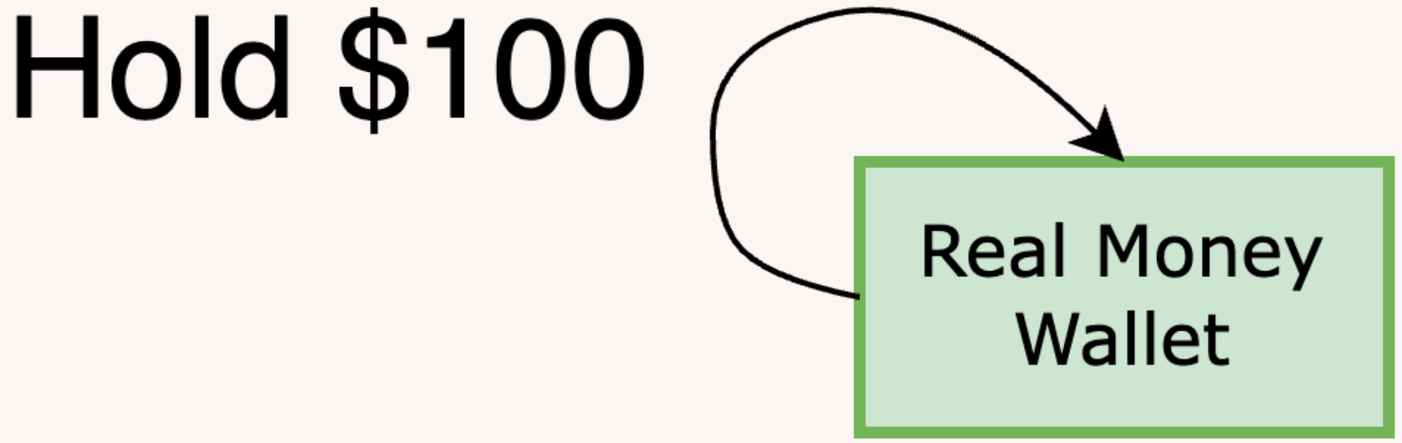
- 25% in stocks
- 25% in real estate
- 25% in crypto
- 25% in bonds

CARTEIRA DIGITAL

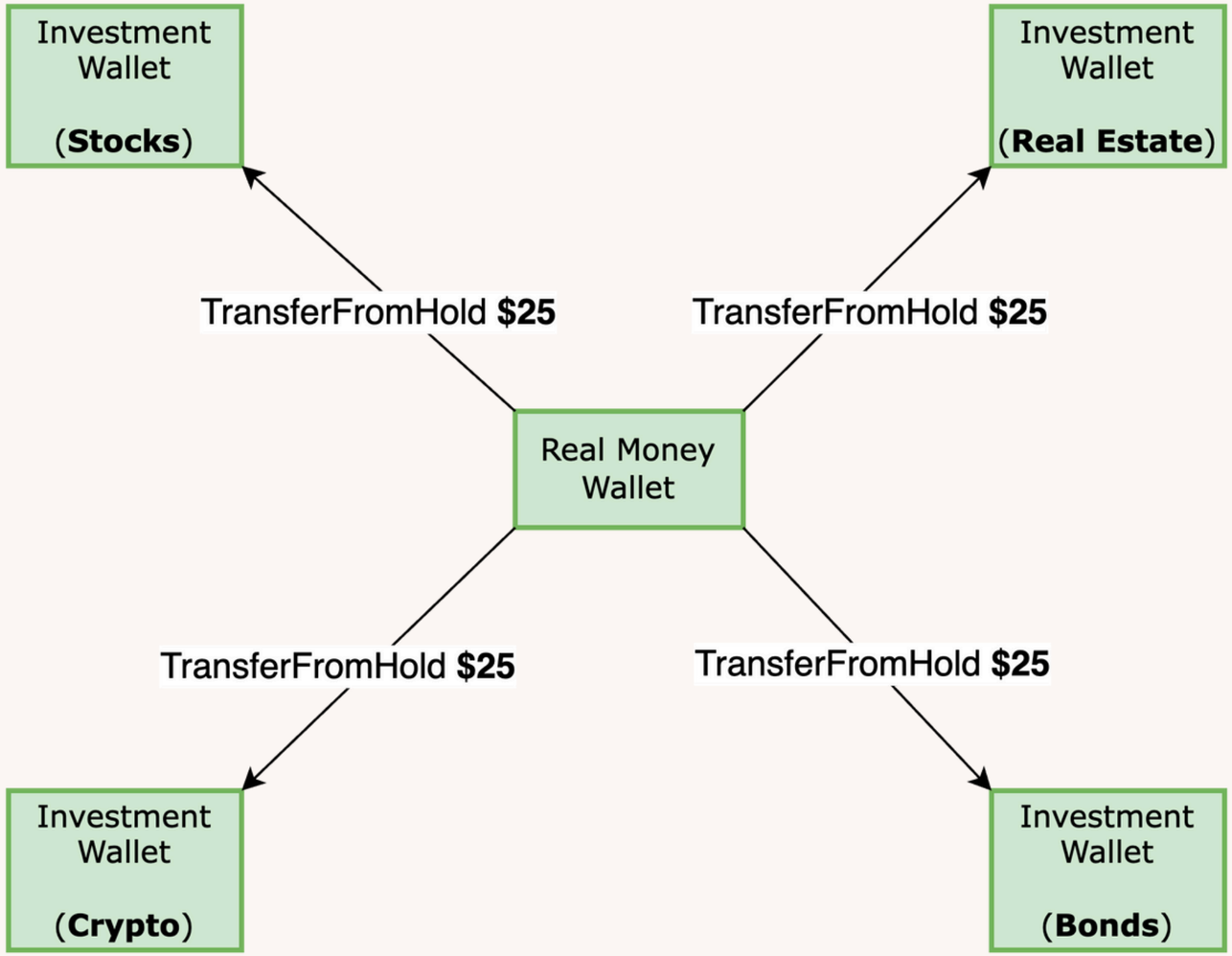
Uma **Transaction** representa uma tentativa de mudar o estado de uma ou mais carteiras. Para uma operação de investimento, dois tipos de Transactions são relevantes:

- **Hold**: reserva dinheiro em uma carteira.
- **Transfer From Hold**: transfere o dinheiro reservado para outra carteira. A transferência requer uma chamada para uma **API externa**.

INVESTIMENTO DE $100



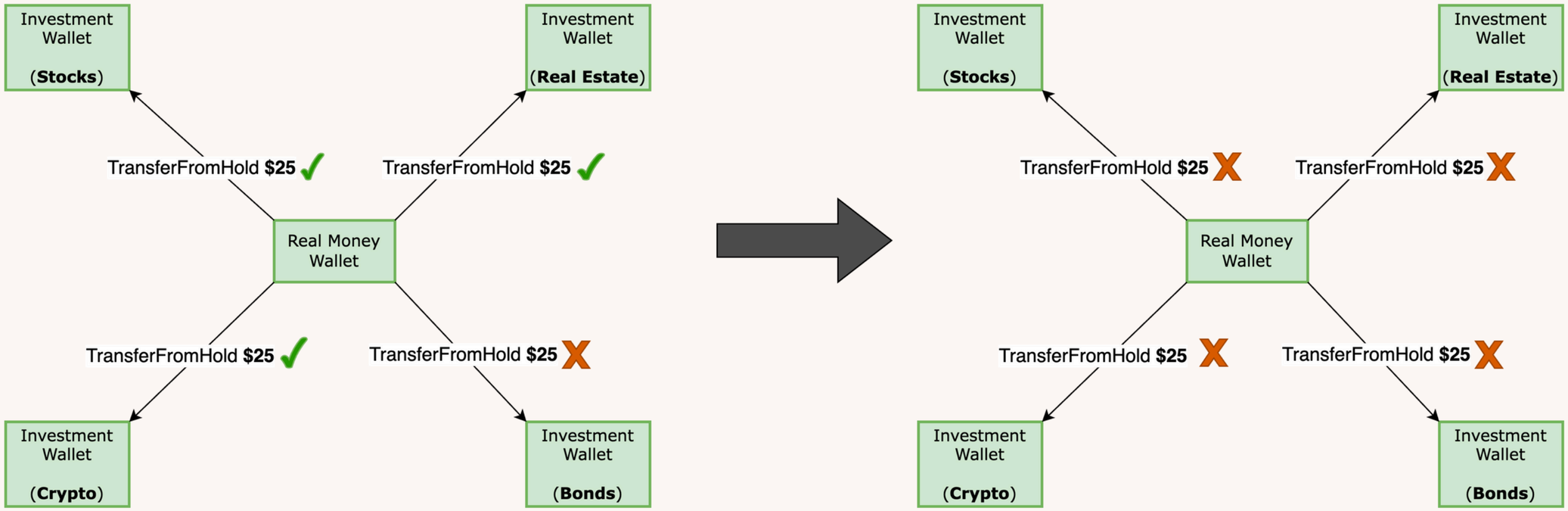
INVESTIMENTO DE $100



Investment Policy

- 25% in stocks
- 25% in real estate
- 25% in crypto
- 25% in bonds

ATOMICIDADE



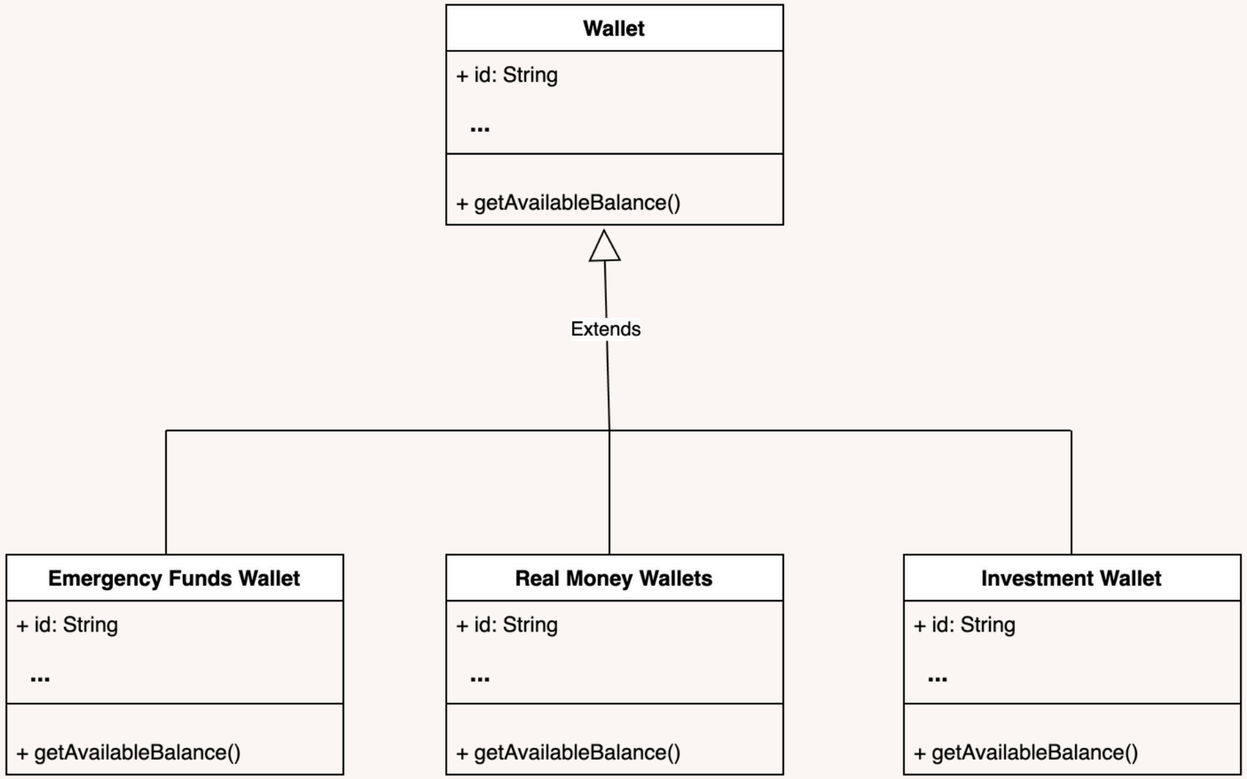
ANÁLISE QUALITATIVA

1. Extensibilidade
2. Reusabilidade
3. Propagação e tratamento de erros
4. Testabilidade
5. Mais observações

EXTENSIBILIDADE

Estudo de caso: representação de uma **Wallet**

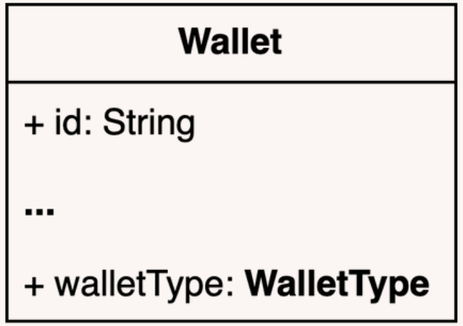
Na implementação em Kotlin, os diferentes tipos de Wallet são representados usando conceitos de POO como **herança** e **polimorfismo**



EXTENSIBILIDADE

Estudo de caso: representação de uma **Wallet**

Na implementação em Scala, os diferentes tipos de Wallet são representados usando um **enumaration** e **pattern matching**




```
match {
   case WalletType.RealMoney =>
        // logic for real money
   case WalletType.Investment =>
       // logic for investment
   case WalletType.EmergencyFunds =>
       // logic for emergency funds
}
```

REUSABILIDADE

Estudo de caso: *retry* de Transactions

A versão em Kotlin da Carteira Digital possui um método que tenta reprocessar uma Transaction em até ***n*** vezes, mas esse método não é genérico o suficiente para ser reutilizado em outros domínios.

REUSABILIDADE

Estudo de caso: *retry* de Transactions

A versão em Scala usa o conceito de **high-order function** (HOF) para implementar um método de *retry* genérico

Uma **high-order function** é uma função que (a) recebe outras funções como parâmetro, ou (b) retorna funções como resultado

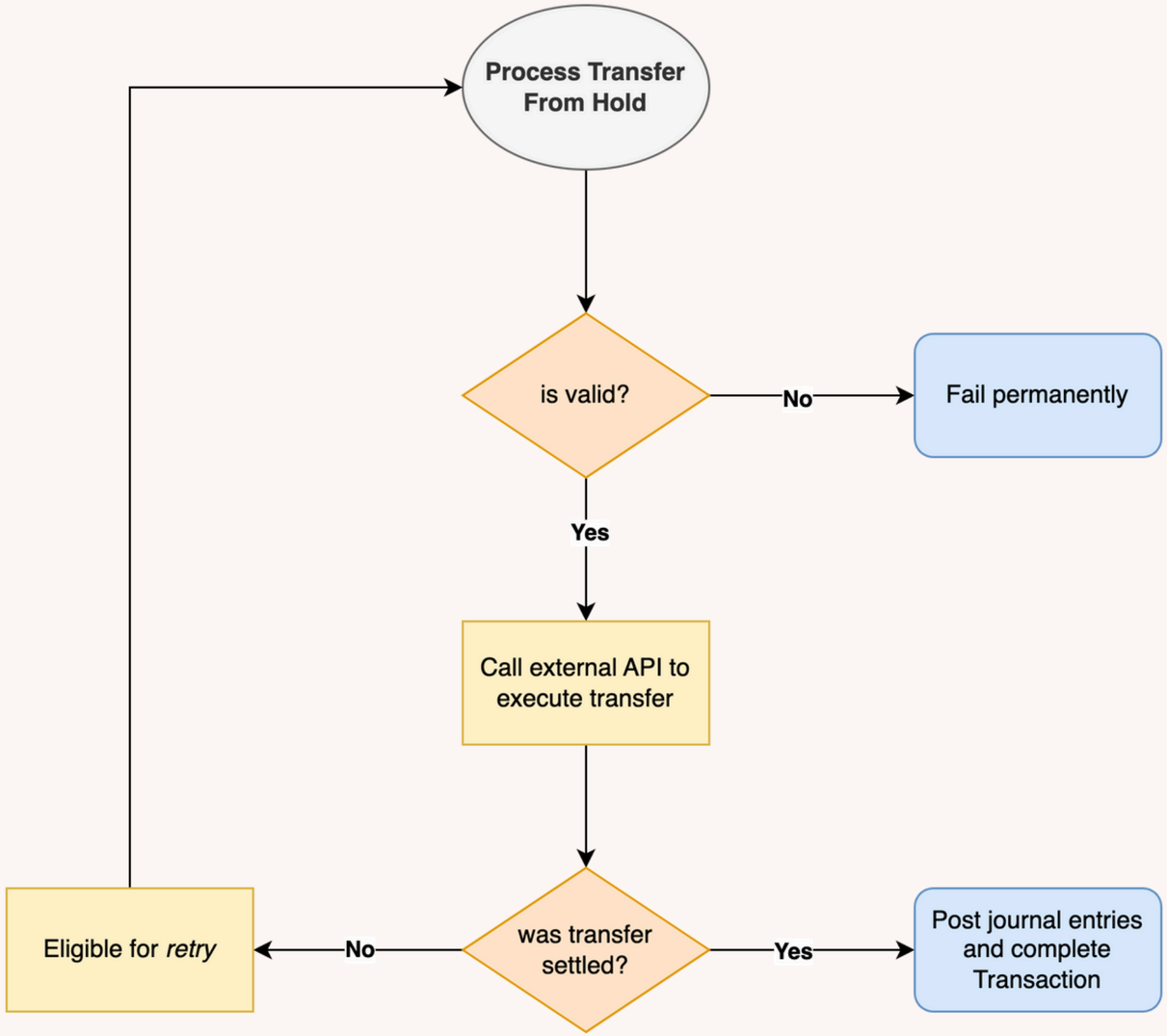
```
def retry[A, B](f: () => Either[A, B], n: Int): Either[Unit, B]

retry(() => processTransaction(t), 3)
```

TESTABILIDADE

Estudo de caso: Transfer From Hold

Na versão em Kotlin, o método responsável por processar um TransferFromHold tem **efeitos colaterais** por toda parte.

Nos testes, foi necessário criar um **mock** para a API externa e estabelecer uma conexão com o banco de dados para verificar a criação dos journal entries.



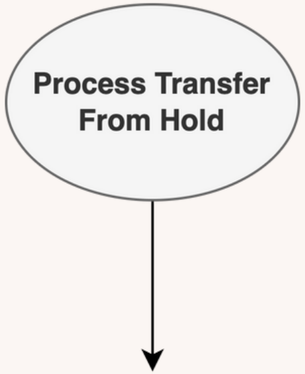
TESTABILIDADE

Estudo de caso: Transfer From Hold

Na versão em Scala, a função responsável por processar um TransferFromHold é uma **função pura**.

Funções puras são funções sem efeitos colaterais, e o seu *output* depende unicamente do seu *input*.

Como não há efeitos colaterais, não foi necessário fazer o mock da API externa ou estabelecer uma conexão com o banco de dados.

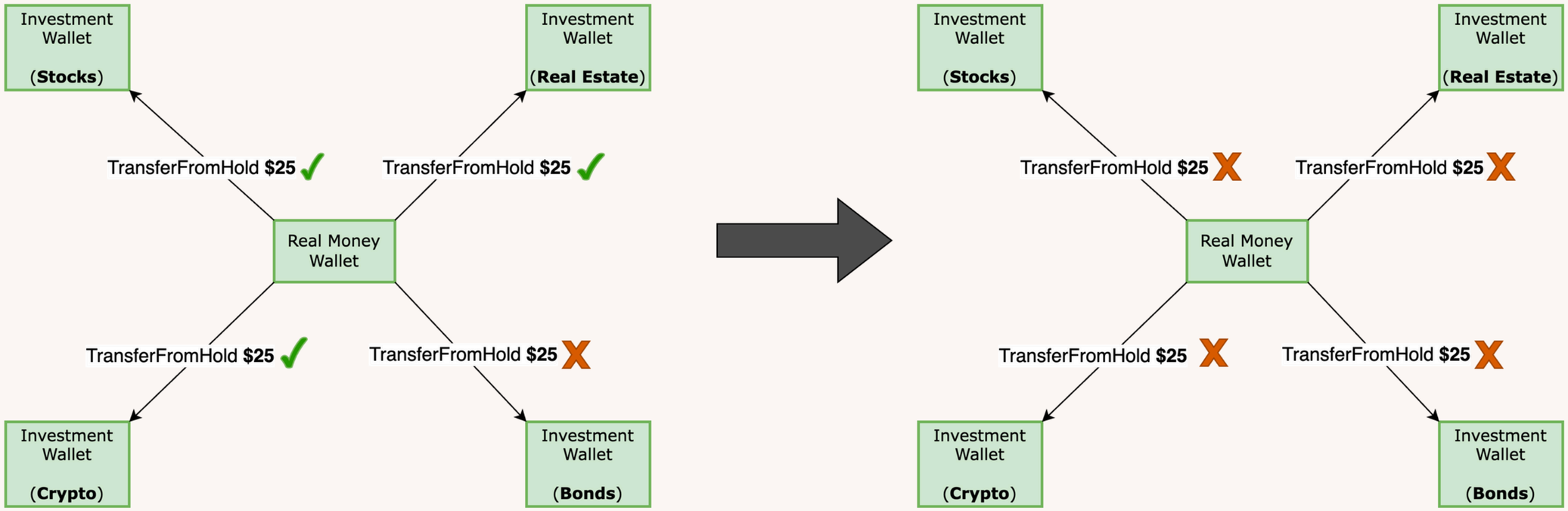


(transaction, Some(action), journalEntries, TransactionStatus.Completed)

MAIS OBSERVAÇÕES

A pureza da função que processa um TransferFromHold garantiu que Scala conseguisse atender ao requisito de **atomicidade**, ao passo que em Kotlin isso é bem desafiador.

ATOMICIDADE



REFERÊNCIAS

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

- Chiusano, Bjarnasson and Pilquist. **Functional Programming in Scala**. Manning, 2023.