# REFATORANDO O PARADIGMA DE PROGRAMAÇÃO APLICADO EM UM SOFTWARE DE CAPTURA DE DADOS

Maxranderson Rodrigues Araújo
Alana Marques de Morais
Aline Marques de Morais

ISBN: 978-85-5597-032-0

# Refatorando o Paradigma de Programação Aplicado em um Software de Captura de Dados

**Maxranderson Rodrigues Araújo**
**Alana Marques de Morais**
**Aline Marques de Morais**
(Autores)

Instituto de Educação Superior da Paraíba - IESP

Cabedelo
2018

**Dados Internacionais de Catalogação na Publicação (CIP)**
**Biblioteca Padre Joaquim Colaço Dourado (IESP)**

R281 Refatorando o paradigma de programação aplicado em um software de captura de dados [recurso eletrônico] / organizadores, Maxranderson Rodrigues Araújo, Alana Marques de Morais, Aline Marques de Morais. - Cabedelo, PB : Editora IESP, 2018.
51 p.

Formato: E-book
Modo de Acesso: World Wide Web
ISBN 978-85-5597-032-0

1. Computação. 2. Programação - Software. 3. Captura de dados - Computação. I. Araújo, Maxranderson Rodrigues. II. Morais, Alana Marques de. III. Morais, Aline Marques de.

CDU 004.4

Bibliotecária: Elaine Cristina de Brito Moreira – CRB-15/053

**Editora IESP**
Rodovia BR 230, Km 14, s/n,
Bloco E - 3 andar - COOPERE
Morada Nova. Cabedelo - PB.
CEP 58109-303

# Prefácio

Este livro se refere ao segundo exemplar de uma série organizada pela professora e pesquisadora Drª. Alana Marques de Morais – membro do corpo docente dos cursos de Sistemas de Informação e Sistemas para Internet do Instituto de Ensino Superior da Paraíba -IESP.

Com o auxílio da docente Drª Aline Marques de Morais da instituição referida e de alunos vinculados ao IESP, alguns trabalhos de destaque apresentados como trabalhos de conclusão de curso foram convidados a serem publicados no formato de livro eletrônico.

# Agradecimentos

Nossa gratidão a todos os envolvidos neste projeto, que dedicaram noites e muito trabalho, especificamente ao bacharel de Sistemas de Informação Maxranderson Araújo. Agradecemos ainda a Professora Doutora Erika Marques, diretora da Instituição de Ensino Superior da Paraíba- IESP, pelo apoio incondicional para a concretização desta obra. Ao Professor Doutor Marcelo Fernandes, Coordenador de Sistemas, pelo suporte técnico, confiança e disponibilidade que permitiram a construção deste livro. Por fim, um último agradecimento ao professor Mestre Hercílio pelo apoio e suporte na edição e publicação deste trabalho.

# Lista de Figuras

# Lista de Quadros

# Lista de Siglas

API - Application Programming Interface (Interface de Programação de Aplicações)

FP - Functional Programming (Programação Funcional)

JVM - Java Virtual Machine (Máquina Virtual Java)

LIFO - Last-in-first-out

OO - Object Oriented Programming (Orientação a Objetos)

RP - Reflection Programming (Programação reflexiva)

RT - Referential Transparency (Transparência Referencial)

SAGRES - Sistema de Acompanhamento da Gestão dos Recursos da Sociedade

TCE-PB - Tribunal de Contas do Estado da Paraíba

UG - Unidade Gestora

# Sumário

# Capítulo 1. INTRODUÇÃO

Os paradigmas de programação surgiram como uma forma de padronizar o desenvolvimento, ajudam o desenvolvedor como forma de abstração e modelagem dos problemas, auxiliando na estruturação do *software* (SCOTT, 2009). A maioria das linguagens provê suporte para um ou mais paradigmas e tendem a otimizar o uso de um. Um exemplo disto é a linguagem Java que usa o paradigma Orientado a Objetos (OO) e a linguagem Scala, que utiliza o paradigma OO e otimiza e aconselha o uso do paradigma de Programação Funcional (FP). Além de OO e FP, temos outros paradigmas bem populares, como o imperativo, o lógico, o estruturado e o declarativo (SEBESTA, 2016).

De acordo com a *Computer Hope* (2017), antes do ano 2005, o desempenho dos processadores crescia com o aumento da frequência do seu *clock.* Contudo, aumentar a frequência começava a não ser viável por conta das limitações dos materiais utilizados na fabricação. Por conseguinte, a partir de 2005, os processadores começaram a ter mais de um núcleo e resultou em aplicações que não estavam preparadas para tal arquitetura.

Para resolver esse desafio, foi desenvolvido o conceito de programação paralela, que visa utilizar todos os núcleos do processador. O problema da programação paralela surge quando ocorre uma mudança de estado do dado em memória, pois um núcleo pode estar realizando a leitura desse dado e outro pode estar sobrescrevendo-o. Por conta desse contexto, alguns paradigmas não estão prontos para trabalhar em paralelo.

A linguagem Java é uma das que não estavam preparadas para o paralelismo. Esta linguagem que teve sua versão 1.0 lançada em 1996, onde utiliza o paradigma OO. A OO é muito popular entre as comunidades de desenvolvedores e um dos mais utilizados. permitindo abstrair modelos que refletem objetos no mundo real, suas características, interações e comportamentos. Este paradigma tem como base quatro princípios, encapsulamento, polimorfismo, herança e abstração de dados. Com o tempo a linguagem obteve um novo pacote chamado *java.util.concurrent* e juntamente com bibliotecas externas, possibilitaram os

desenvolvedores a trabalhar com essas novas CPUs com múltiplos núcleos, mas infelizmente, por conta da limitação do paradigma, eles não puderam ir muito longe (WARBURTON, 2014).

Para seguir em frente, é necessário mais uma mudança na linguagem, a introdução das *Lambdas expressions*, ou expressões Lambdas, essa nova sintaxe introduz a FP na linguagem Java, esse novo paradigma permite executar operações paralelas em grandes coleções de dados (WARBURTON, 2014).

Para realizar essas operações em paralelo, a FP se molda em alguns conceitos, como a imutabilidade, no qual não se pode alterar o estado do dado, quando duas CPUs precisam ler dados memória. Este conceito garante que o dado não seja alterado por outro processamento realizado, além destes outros conceitos pertencentes à FP são mostrados ao longo deste trabalho.

Até uma linguagem de programação popular como Java precisou de uma mudança para usufruir da nova capacidade dos *hardwares*, principalmente nessa nova fase, no qual se gera uma grande quantidade de dados. Algumas aplicações precisam mudar para se adequar a essa nova demanda de dados, com isso, este trabalho visa estudar, planejar e investigar a necessidade dessa mudança de paradigma.

No decorrer do projeto, o presente estudo analisou o Sistema de Acompanhamento da Gestão dos Recursos da Sociedade (SAGRES) na versão Captura, que é um módulo que permite a captura dos dados da execução orçamentária, licitações, obras e folha de pessoal dos municípios. Este sistema está sendo refatorado para a linguagem Scala, que possui uma arquitetura Web com o *framework* Play. Neste módulo, os 223 municípios da Paraíba informam os seus lançamentos contábeis, no qual a aplicação realiza uma série de análises nesses, verificando se estão de acordo com as leis da contabilidade pública brasileira e com a estrutura definida pelo Tribunal de Contas do Estado da Paraíba (TCE - PB).

Com o crescimento desta aplicação, a quantidade de dados informados está aumentando, mas devido aos prazos curtos e pressa na implementação do sistema, algumas rotinas não estão bem otimizadas, além de não utilizar corretamente os recursos propostos pela linguagem. Este trabalho se propõe realizar um estudo

sobre os paradigmas disponíveis nas tecnologias usadas por este sistema, planejar e desenvolver uma nova rotina que se adeque às necessidades futuras e ao processamento em múltiplos CPUs.

Além do desenvolvimento dessa nova rotina, é necessária uma análise crítica entre os paradigmas escolhidos devido à carência de estudos que comparem os paradigmas em um contexto governamental, serão apresentadas algumas vantagens e desvantagens, serão geradas métricas de desempenho para serem usadas na comparação dessa nova estrutura com a anterior, verificando assim, a viabilidade da implementação dessa nova rotina.

## 1.1 OBJETIVOS GERAIS

Analisar comparativamente os paradigmas de programação disponíveis na linguagem Scala.

## 1.2 OBJETIVOS ESPECÍFICOS

Para alcançar tais metas, são adotados os seguintes objetivos específicos:

- Pesquisar sobre os paradigmas de programação disponíveis na linguagem Scala.
- Planejar e desenvolver a nova rotina com o paradigma definido.
- Comparar os resultados obtidos.

## 1.3 ESTRUTURA DO LIVRO

Este livro foi organizado em 6 capítulos, incluindo o introdutório, que discorreu sobre a apresentação do trabalho, a descrição do problema, os objetivos do trabalho, contribuição e a organização do mesmo. O capítulo 2 apresentou os conceitos relativos aos principais paradigmas utilizados na presente análise. O capítulo 3 apresenta a estrutura do sistema SAGRES, utilizado como estudo de caso no comparativo de paradigmas. O capítulo 4 apresentou os resultados obtidos até o momento, além de descreveu uma rotina implementada no estudo comparativo. Por

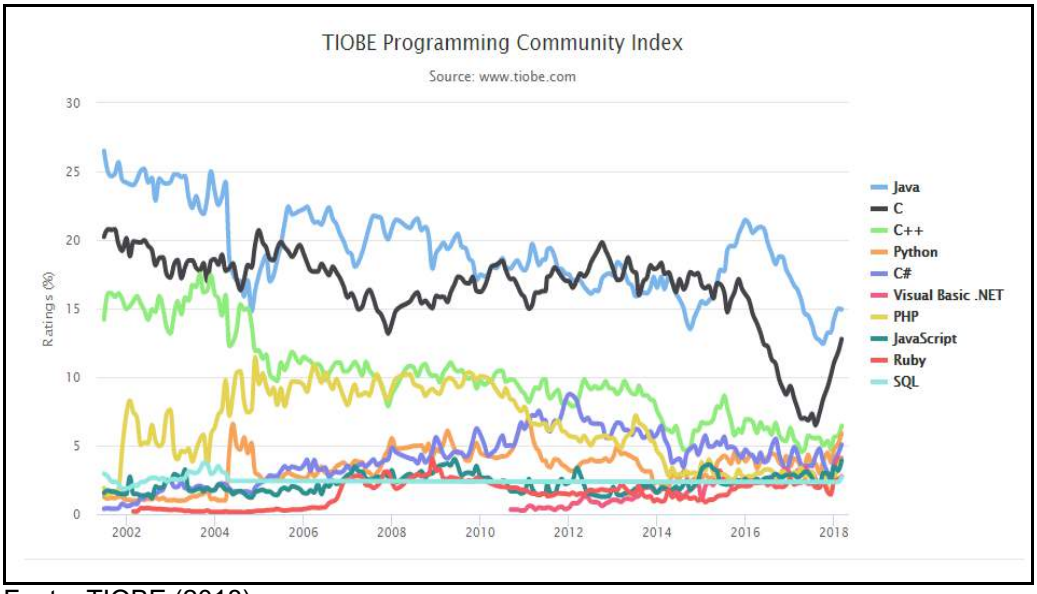
fim, o capítulo 5 apresentou as considerações finais do trabalho e o capítulo 6 as referências utilizadas neste livro.

# Capítulo 2. PARADIGMAS DE PROGRAMAÇÃO

Neste capítulo, foram analisados alguns paradigmas de programação disponíveis na linguagem Scala e utilizados no presente estudo. Na Figura 1, é apresentado uma pesquisa feita pela TIOBE, realizada por meio de consultas utilizando o nome da linguagem nos principais mecanismos de buscas da Internet, como o Google. Nesta figura, é possível identificar as linguagens mais populares do mundo (TIOBE, 2018).

De acordo com a Figura 1, pode-se concluir que a linguagem mais popular é a Java. De acordo Ritter (2016), os principais motivos dessa popularidade são: (i) a praticidade, devido as suas convenções e legibilidade do código; (ii) a compatibilidade com versões anteriores; (iii) a escalabilidade, performance e confiabilidade, devido a Máquina Virtual Java; e (iv) as atualizações nas novas versões, como observado na Figura 1, no fim do ano 2014, a linguagem teve um grande aumento por conta do lançamento da *Java Development Kit* (JDK) 8, onde foi incluído o paradigma funcional.

**Figura 1 - *Programming Community Index.***




Fonte: TIOBE (2018).

Diante da relevância da linguagem Java, é interessante para este trabalho discutir em detalhes os paradigmas OO e FP, pois ambos foram importantes para o entendimento da nova rotina proposta por este trabalho.

## 2.1 ORIENTAÇÃO A OBJETOS

De acordo com Baesens, Backiel e Broucke (2015) “o paradigma OO é um dos mais populares”. Alguns exemplos de linguagem que utilizam esse paradigma são Eiffel, Smalltalk, C++ e Java. Também pode-se verificar na Figura 1, que a linguagem Java é uma das mais populares linguagens do mundo e utiliza esse paradigma.

A OO consiste em uma abstração dos dados em Objetos, no qual cada elemento é uma instância de uma classe e representa um modelo dos atributos e comportamentos do objeto (BAESENS, BACKIEL, BROUCKE, 2015).

Para manter essa forma de abstração, é necessário fazer uso de dois conceitos, o encapsulamento, que representa quando os dados são agrupados e armazenados em um objeto, e a *information hiding* onde é necessário a criação de métodos no objeto dando acesso à leitura ou alterações no dado, para isolá-los do “mundo externo” (BAESENS, BACKIEL, BROUCKE, 2015).

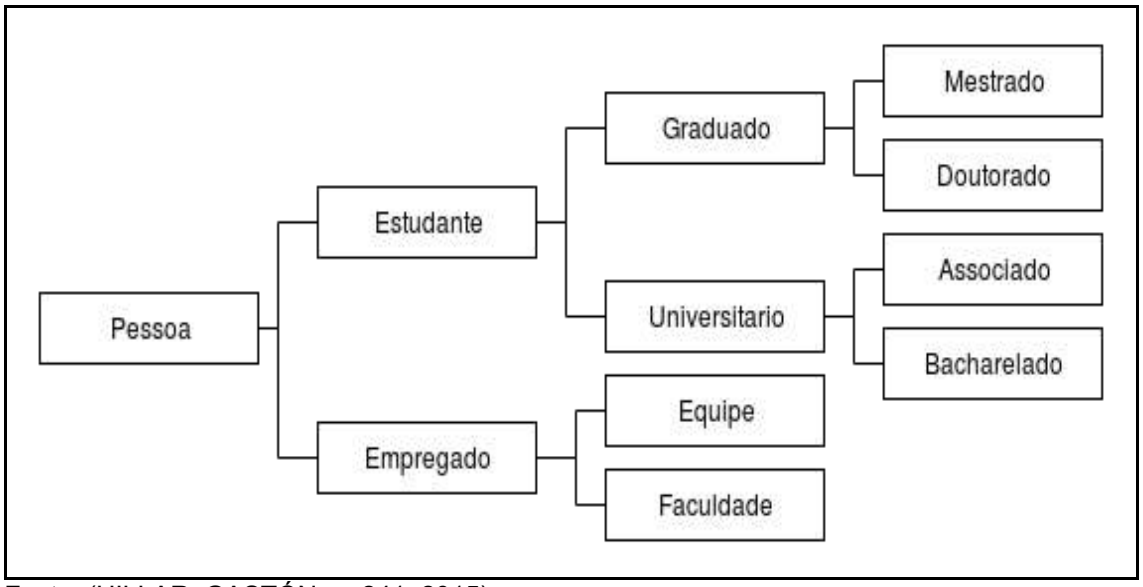
A OO utiliza outros conceitos para garantir que o programa cresça e se desenvolva com facilidade, como: (i) o polimorfismo, que permite que objetos diferentes tratem outros de maneira similar, e (ii) a herança, que garante o reuso do código fazendo com que objetos herdem características de outros.

### 2.1.1 Herança

Um conceito muito importante na OO, representa uma forma de generalização de classes, analisando duas classes com características idênticas. Para utilizar a herança, extraímos o que elas têm em comum e criamos uma classe superior que as represente de uma forma mais genérica. Em seguida, fazemos com que essas classes herdem essas características, desta forma declaramos essas características em um único local (HILLAR, GASTÓN 2015).

Na Figura 2, está representado um exemplo de herança, no qual o estudante de uma faculdade pode ser um objeto da classe *Graduado* ou *Universitario*, e uma *Pessoa*.

**Figura 2 - Exemplo de Herança.**




Fonte: (HILLAR, GASTÓN, p. 241, 2015).

### 2.1.2 Polimorfismo

Outro conceito importantíssimo na OO é o polimorfismo que está diretamente relacionado à herança. De acordo com a Figura 3, há duas instâncias de objetos no código exemplificado, uma da classe *Master* e outra da classe *Associate*, representadas no código abaixo, utilizando a linguagem Java.

Na Figura 3, os métodos *getName()* não pertencem a *anne* e *john*, ou seja, a classe *Master* ou a classe *Associate* não possuem esses métodos. Sendo assim, a *Java Virtual Machine* (JVM) ou Máquina Virtual Java irá percorrer suas superclasses até encontrar quem possui esse método, que no caso é a classe *Person*. Apesar das classes *Master* ou *Associate* herdarem os mesmos métodos, os resultados obtidos após a execução são diferentes, pois *anne* e *john* implementam esse método de forma diferente. Dependendo do tipo da instância, percebe-se um resultado diferente para a chamada do mesmo método e com os mesmos argumentos (HILLAR, 2015).

**Figura 3 - Exemplo de Polimorfismo.**

```
public class PersonProgram{
    public static void main(String[] args){
        Student john =  new Master("John Adams");
        john.setGrades(0.75, 0.82, 0.91, 0.69, 0.79);
        Student anne = new Associate("Anne Philips");
        anne.setGrades(0.75, 0.82, 0.91, 0.69, 0.79);

        System.out.println(john.getName() + ": " +
john.calculateGPA() );
        System.out.println(anne.getName() + ": " +
anne.calculateGPA() );
    }
}
```

Fonte: (HILLAR, GASTÓN, p. 244, 2015).

## 2.2 PROGRAMAÇÃO FUNCIONAL

A FP é bem desafiadora e para entender os seus conceitos é necessário o uso de alguns trechos de código. Neste capítulo, é discutido o que é a FP, seus principais conceitos e benefícios.

> "*O que é programação funcional? Para mim, é simplesmente um apelido para programar com funções, só isso, um estilo de programação que põe em foco as funções em um programa. O que são funções? Aqui, nós achamos um grande espectro de definições. Enquanto uma definição frequentemente admite funções que podem ter efeitos colaterais além de retornar um resultado, programação funcional pura restringe funções a como elas são na matemática: operações binárias que mapeiam os argumentos aos seus resultados.*" (CHIUSANO, BJARNASON. p. 14, 2015)

A FP é baseada nos conceitos de funções puras, no qual se desenvolve aplicações com funções que não possuem efeitos colaterais, por exemplo: modificar uma variável e ler dados de um arquivo.

No trecho de código, apresentado na Figura 4, está um exemplo de efeito colateral, onde é descrito um método de um programa para comprar cafézinhos. No trecho *cc.adicionar(xicara.preco)*, gerou um efeito colateral, cujo o cartão de crédito está armazenando um valor internamente, entrando em contato com a empresa do cartão e autorizando a compra, todas esses efeitos em um único método.

Para transformar esse código em uma função pura, é necessário que todos os resultados do processamento sejam retornados pela função, como no trecho apresentado na Figura 5.

Torna-se, assim, o trecho completamente funcional, no qual se retorna a *xicara* e a representação da compra feita com cartão de crédito. De uma maneira mais formal, Chiusano e Bjarnason (2015, p. 6) definem “que uma função pura F, com uma entrada do tipo A e uma saída do tipo B, é um processamento que relaciona todo valor do tipo A com exatamente um valor do tipo B, qualquer mudança de estado interna ou externa é irrelevante para o processamento do resultado de F(A)”.

**Figura 4 - Código com efeito colateral.**

```
class Cafe {

   def comprarCafe(cc: CartaoCredito): Cafezinho = {
      val xicara  = new Cafezinho()
      cc.adicionar(xicara.preco)
      xicara
   }

}
```

Fonte: Próprio Autor.

**Figura 5 - Código funcional.**

```
class Cafe {

 def comprarCafe(cc: CartaoCredito):(Cafezinho, Compra) = {
      val xicara  = new Cafezinho()
      ( xicara, Compra(cc, xicara.preco) )
 }

}
```

Fonte: Próprio Autor.

Somente remover os efeitos colaterais não são suficientes, é necessário isolar a função do mundo externo, ou seja, as funções não podem fazer leituras de dados externos, ela só deve depender dos seus parâmetros de entrada (SAUMONT e PIERRE-YVES, 2017).

Pode-se formalizar o conceito de funções puras de uma outra forma, fazendo uso do conceito de Transparência Referencial (RT).

### 2.2.1 Transparência Referencial

De acordo com Chiusano e Bjarnason (2015, p. 10), uma expressão *e*, é uma RT, se para cada programa *p*, todas as ocorrências de *e* em *p* podem ser substituídos pelo resultado processado de *e* sem afetar o sentido de *p*. Uma função *f* é pura quando a expressão *f(x)* é RT para toda RT de *x*.

Para um melhor entendimento da RT, há o código apresentado na Figura 6, no qual *x* recebe uma nova instância de StringBuilder, *r1* recebe a *String* resultada da chamada da função *x.append(“, World)*, após isso, *r2* recebe o resultado da chamada dessa mesma função.

**Figura 6 - Exemplo de código sem RT.**

```
val x = new StringBuilder("Hello")

/* x é Hello: scala.collection.mutable.StringBuilder */

val r1 = x.append(", World").toString

/* r1 é Hello, World: java.lang.String */

val r2 = x.append(", World").toString

/* r2 é Hello, World, World: java.lang.String */
```

Fonte: Próprio Autor.

Percebe-se que os resultados da chamada da função "*x.append(", World")*" (responsável por concatenar o parâmetro passado com a *String* interna) são diferentes, a função mudou o estado do objeto referenciado pela variável x, quebrando o conceito de imutabilidade da RT. A RT garante alguns benefícios como:

- Modularidade:

  "*Programação Funcional força problemas grandes a serem quebrados em pequenos problemas para serem resolvidos. Isto significa que o código fica mais modular*." (Ved Antani, Simon Timms, Dan Mantyla, 2016, p. 482)

- Testabilidade:

  "*Programação Funcional é mais fácil de testar. Por não ter efeitos colaterais, não precisa de mocks, o que geralmente é necessário para isolar o programa em teste, do ambiente externo.*" (SAUMONT, PIERRE-YVES, 2017, p. 7)

- Reusabilidade:

> "*Para escrever um programa funcional, deve-se começar escrevendo várias funções base e combiná-las em funções maiores, repetindo esse processo até que obtenha uma única função que corresponde ao programa que se deseja construir. Como todas essas funções são transparentes referenciais, elas podem ser reaproveitadas para construir outros programas sem modificações.*" (SAUMONT, PIERRE-YVES, 2017, p. 8)

- Matematicamente Correto:

> "*Essa está mais em um nível teórico. Devido às raízes em Lambda Cauculus, programas funcionais podem ser provados matematicamente como corretos. Esse é grande vantagem para os pesquisadores que precisam provar a taxa de crescimento, complexidade do tempo, e a exatidão matemática do programa.*" (Ved Antani, Simon Timms, Dan Mantyla, 2016, p. 482)

## 2.3 REFLEXIVO

O paradigma Reflexivo vem da necessidade de postergar a implementação do programa para o tempo de execução. De acordo com Malefant (1996), a programação reflexiva é *"a capacidade do programa de manipular os dados como algo que represente seu estado durante a sua execução"*.

A Programação Reflexiva ou *Reflection Programming* (RP), ganhou destaque no fim dos anos 80, por conta da família de linguagens de programação Lisp, devido ao seu grande poder de metalinguística, foi possível manipular e executar fragmentos dos programas em tempo de execução (DEMERS, 1995).

No começo dos anos 90, foi identificado uma dependência de outras estruturas, que para uma completa utilização do paradigma reflexivo é necessário a utilização de outro paradigma, ou seja, o paradigma reflexivo precisa ser implementado juntamente de outro paradigma, atingindo uma implementação mais avançada (DEMERS, 1995).

Devido a essa dependência de outros paradigmas, cada comunidade desenvolveu sua própria solução utilizando o paradigma reflexivo, dificultando a

identificação dos conceitos similares implementados por cada comunidade. Dessas implementações, foram notadas o uso de muitas técnicas e mecanismos, como a habilidade de manipular programas em tempo de execução, inspecionar estruturas de dados em tempo de execução, entidades de primeira classe, *metaprogramming*, etc.

Alguns tópicos definidos por Maes (MAES, 1987 apud DEMERS, 1995, p. 2) sintetizam o que é reflexão nos sistemas computacionais:

- Um sistema computacional é algo que dá significado e age em alguma parte do mundo, chamado de o domínio do sistema.
- Um sistema computacional pode estar conectado ao seu domínio. Onde o domínio e o sistema podem estar ligados de forma que uma mudança em um, pode afetar o outro.
- Um *meta-system* é um sistema computacional possui as informações de outros sistemas nos seus dados, onde os manipula e os gerencia, esses outros sistemas são chamados de sistemas objeto.
- Reflexão é o processo de dar razão a algo e/ou agir sobre si mesmo.
- Um sistema reflexivo é ocorre quando, o sistema possui informações de si, a capacidade de dar razão e agir sobre si mesmo.

Para um melhor entendimento do paradigma, Demers (1995) distinguiu dois conceitos relevantes: reflexão estrutural e reflexão comportamental. A reflexão estrutural implica na habilidade da linguagem de prover uma representação e permitir alterar as definições do programa executado, incluindo os tipos de dados abstraídos. O segundo conceito, é a reflexão comportamental implica na habilidade da linguagem de prover uma representação e permitir alterar a sua semântica, além dos dados utilizados para execução do programa atual.

### 2.3.1 Reflexão Estrutural

A linguagem Java possui uma Interface de Programação de Aplicações ou *Application Programming Interface* (API), para desenvolvimento no paradigma Reflexivo, mas a linguagem não possui a capacidade completa desse paradigma.

Para exemplificar a utilização desses conceitos nessa linguagem, é necessária uma extensão que adicione esses comportamentos.

Para demonstrar esse conceito em Java, será utilizada a extensão *Javassist*. Na Figura 7, tem-se um exemplo da reflexão estrutural, criamos uma instância de *CtClass*, que representa o *bytecode* da classe *Pai*, essa classe não possui nenhum atributo ou método, mas possui uma herança, a classe *Filho* é subclasse de Pai. Criou-se um atributo representado pela instância de *CtField*, adicionou-se esse novo campo a classe *Pai*, e em seguida, salvou-se a classe, tudo isto realizado em tempo de execução.

**Figura 7 - Exemplo de Reflexão.**

```
/* … Código anterior */
CtClass c = new CtClass("Pai");
CtField novo = new CtField(CtClass.intType, "nome", c);
c.addField(novo);
c.writeFile(".");
/* Código posterior */
```

Fonte: Próprio Autor.

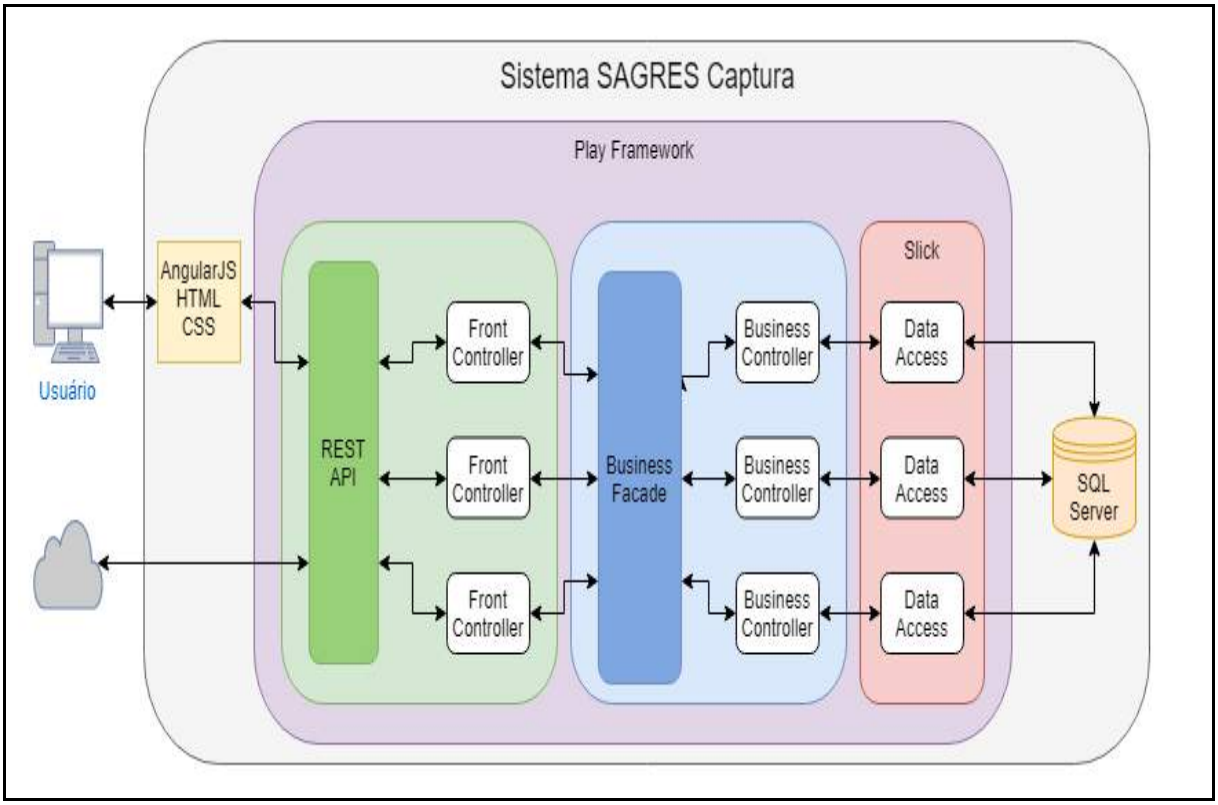
### 2.3.2 Reflexão Comportamental

Este conceito é um pouco mais desafiador do que a estrutural, é possível implementá-lo usando a mesma extensão *Javassist*, mas exige uma quantidade muito maior de linhas de código. A mudança de comportamento é feita por meio de ganchos, cria-se um *MetaObject*, que copia e armazena todas as informações relacionadas ao objeto a ser modificado. Quando alguma informação for requisitada desse objeto, o *MetaObject* realizará uma interceptação, substituindo o método por outro definido por ele. Desta forma, consegue-se modificar o comportamento de uma classe em tempo de execução.

# Capítulo 3. SISTEMA SAGRES

O SAGRES foi desenvolvido pelo Tribunal de Contas do Estado da Paraíba (TCE-PB). A versão CAPTURA é um dos módulos que permite o envio de dados gerados pelos municípios, como os dados de execução orçamentária, licitações, obras, etc. No momento, o seu módulo CAPTURA está sendo refatorado usando novas tecnologias Web, são elas: (i) a linguagem Scala, que compila para JVM; (ii) um *framework* Web chamado *Play*; (iii) um *framework* Javascript chamado AngularJS na camada de visualização do usuário ; (iv)para abstrair e manipular o banco optou-se pela ferramenta *Slick; e (v)* o SQL Server como banco de dados. Para representar melhor esse acervo de tecnologias pode-se observar na Figura 8 representando a arquitetura do sistema.

**Figura 8 - Arquitetura do SAGRES Captura.**




Fonte: Próprio Autor.

O processo de informar os dados municipais ocorre em algumas etapas, como, por exemplo, o município deve informar os dados gerados diariamente. Nesta etapa são informados oito arquivos, cada arquivo pode chegar a ter mais de mil linhas de informações sobre municípios. Assim que os arquivos são informados pela aplicação, é realizada uma série de validações, verificando se os dados informados estão de acordo com a estrutura definida pelo TCE-PB e pelas Normas Brasileiras de Contabilidade aplicadas ao setor público.

O sistema possui uma rotina específica responsável por fazer todas validações. Neste processo de validação, são feitas consultas a banco de dados e outros sistemas *Web*. Todo a etapa parece simples, mas este processo de validação é feito para cada linha do arquivo, que como foi dito, pode conter mais de mil linhas, feito em cada arquivo, que no momento são oito, e informados diariamente por mais de 600 usuários. Este processo é uma das principais funcionalidades, um tempo de resposta alto provoca muitos problemas, como *timeouts* nas requisições, clientes reenviando requisição, entre outros.

A refatoração executada no sistema do SAGRES exigiu que uma nova rotina fosse definida a fim de avaliar um novo paradigma proposto. Os fatores para o desenvolvimento desta etapa foram: melhorar o desempenho da rotina com o uso da paralelização, facilitar a implementação de novas regras e novas estruturas de dados, melhor modularização das regras e permitir a criação de testes com menos dependências com o uso da FP.

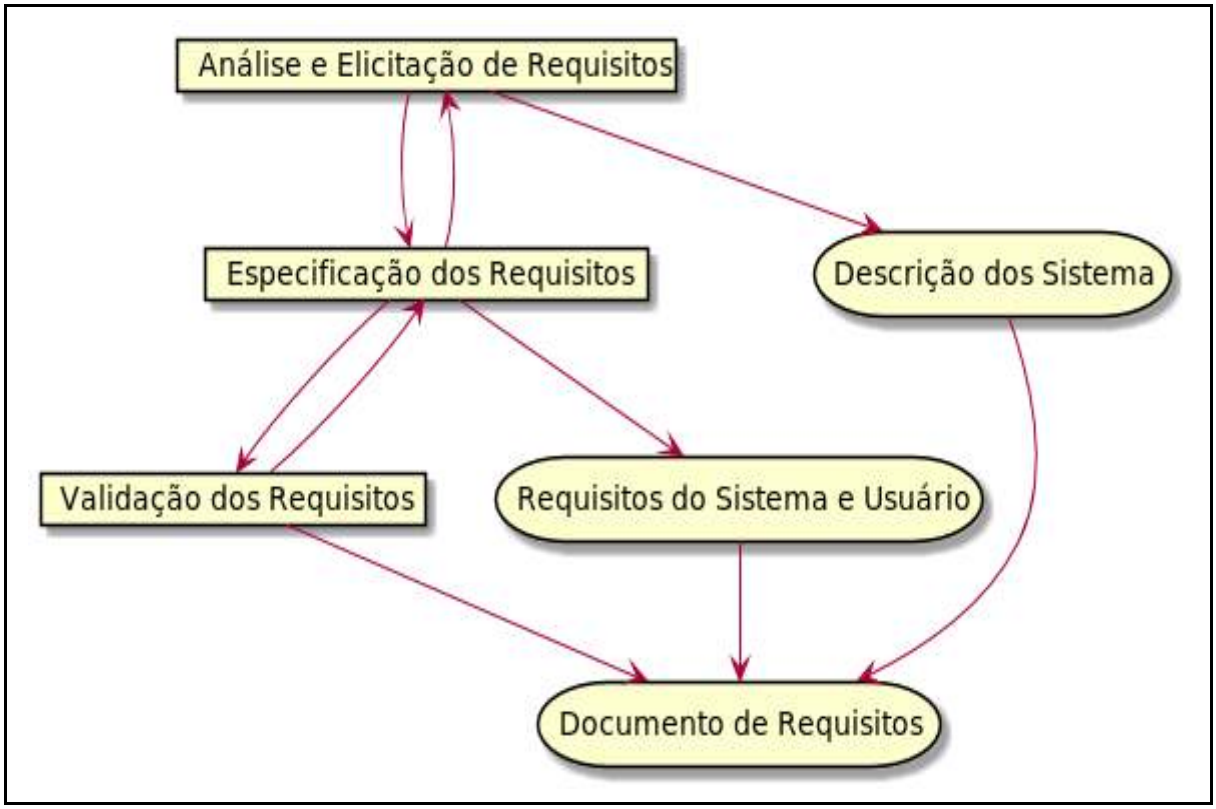
Para que tal implementação fosse executada e testada, algumas etapas foram executadas de acordo com um planejamento prévio realizado. No processo de desenvolvimento da nova rotina foi utilizado um processo baseado no modelo incremental definido por Sommerville (p. 49, 2016), no qual “este modelo é baseado na ideia desenvolvimento de uma implementação inicial, coletar *feedback* dos usuários e outros, e evoluir o *software* por diversas versões até que a versão final esteja pronta”. Para este processo, foi observado que a rotina não é um *software* completo, mas uma *feature*. O prazo é longo, possibilitando a definição de atividades concretas e não sendo necessário uma implementação contínua.

## 3.1 ROTINA VALIDAÇÃO

Nesta seção, é apresentado as etapas percorridas para a construção da nova rotina desenvolvida. Tomando como base as etapas do processo de *software* definido por Sommerville (p. 23, 2016), que diz “as quatro atividades básicas do processo são, a especificação, o desenvolvimento, validação e evolução”. Este projeto desenvolveu duas das quatro atividades, a especificação e o desenvolvimento.

De acordo com Sommerville (p. 55, 2016), a atividade de especificação possui três grandes atividades. Análise e Elicitação, etapa em que ocorre a coleta de toda as informações necessárias para a elaboração dos requisitos. Especificação, etapa no qual são gerados os documentos especificando os requisitos do usuário e os requisitos do sistema. Validação, etapa no qual os requisitos gerados pela etapa anterior são validados. Todo esse processo está demonstrado na Figura 9.

**Figura 9 - Processos de engenharia dos requisitos.**



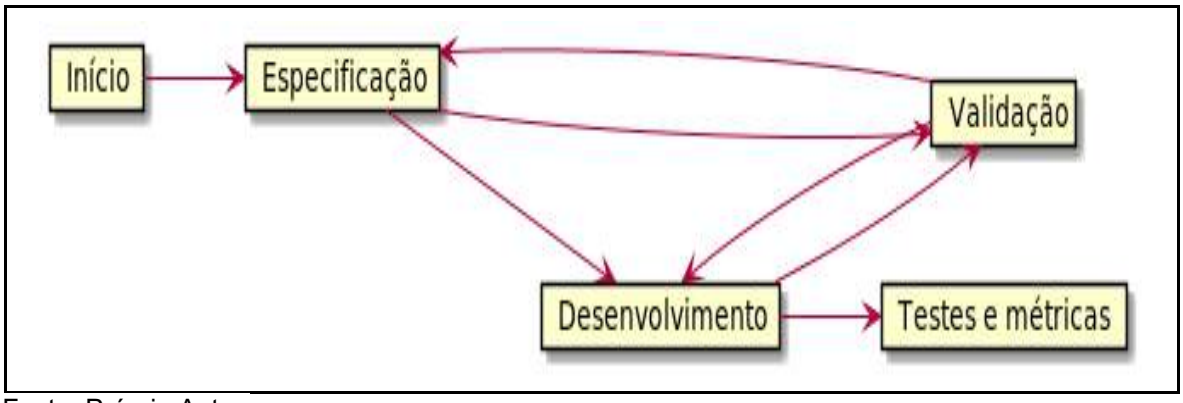

Fonte: Sommerville (p. 55, 2016).

Neste projeto, como os requisitos são para uma única *feature*, assim que elaborados são validados pelos desenvolvedores do sistema, por isso a atividade de Validação foi realizada juntamente com a Análise e Elicitação. O processo de Especificação foi detalhado na subseção 3.1.1.

Na etapa de Desenvolvimento, é realizado o *design* do código e a implementação do mesmo, em um processo de um *software* completo esta etapa possui quatro atividades no processo de *design*. O *design* arquitetural, onde é definido a estrutura do sistema e seus principais componentes. O *design* do banco de dados, em que é definido a estrutura de dados que será representada no banco. O *design* da interface, onde é definida a interface entre os sistemas. Por fim, o *design* e seleção dos componentes, no qual é realizado uma busca por componentes reutilizáveis. Se não encontrados, será feito um *design* para eles. Após essas atividades, o código começa a ser implementado (SOMMERVILLE, 2016).

Em relação ao processo anteriormente mencionado, para o desenvolvimento da nova rotina do sistema SAGRES o processo percorrido percorreu as etapas ilustradas pela Figura 10.

**Figura 10 - Processo de desenvolvimento.**




Fonte: Próprio Autor.

Como observado da Figura 10, os processos de especificação e desenvolvimento foram serão intercalados com ao processo de validação, ou seja, tudo que for produzido é será validado com os envolvidos com o sistema. Desta

forma, é possível verificar se a *feature* está de acordo com os requisitos definidos na especificação do projeto.

Na etapa de especificação, foram definidas as seguintes atividades:

- Análise da rotina anterior,
- Coleta de informações adicionais,
- Geração e documentação dos requisitos,
- Elaboração dos diagramas de classe,
- Elaboração dos diagramas de sequência e
- Escolha dos testes e métricas.

Na etapa de desenvolvimento, foram definidas as seguintes atividades:

- Elaborar os diagramas de classe
- Elaborar os diagramas de sequência
- Implementação da nova rotina,
- Correção de diagramas de classe e sequência, quando necessário

Na etapa de testes e métricas, foram definidas as seguintes atividades:

- Refatoração da rotina anterior;
- Escolha dos testes e métricas
- Implementação e execução dos testes e
- Documentação dos resultados.

### 3.1.1 Especificação

Nesta etapa, realizando uma análise na rotina atual, foi identificada as seguintes características: a rotina recebe como dados de entrada, a linha do arquivo, o código da unidade gestora, o número da linha, data de competência desses dados, um objeto da classe *ControleArquivo* e um objeto da classe *ImportacaoException*.

A classe *ControleArquivo* contém os seguintes dados:

- *codigoArquivo*: Código que representa o tipo de entidade que está sendo recebida.

- *ano*: O ano de competência da entidade a ser recebida.
- *tamanhoLinha*: O tamanho da linha que o arquivo deve ter quando recebido.
- *ativo*: Status que representa se esse controle está ativo.
- *tipoSistema*: O código do sistema a que este arquivo pertence.
- *ordemImportacao*: Representa a posição da ordem em que o arquivo deve ser validado.
- *layout*: Contém um objeto *JsonValue* onde estão as informações das posições dos campos na linha do arquivo.

A classe *ImportacaoException* encapsula todos os erros gerados na validação dos dados, provenientes do arquivo recebido dos municípios. Os erros podem ter dois tipos, o tipo *warning* e o tipo *error*. O tipo *warning* são apenas avisos e não impede a construção da entidade, o tipo *error* são gerados quando o dado não passa na validação e impede a construção da entidade.

Analisando as rotinas atuais de validação das entidades foram identificadas as seguintes semelhanças: (i) valida o tamanho da linha do arquivo verificando se o tamanho é igual ao definido no objeto da classe *ControleArquivo; (ii)*extrai o campo que representa a Unidade Gestora (UG) a qual esta linha pertence, verifica se está vazio e se pertence a UG informada no arquivo.

No restante dos campos da linha são realizadas as seguintes operações: (i) extração do dado da linha; (ii) se o campo for obrigatório, é realizado uma verificação se o dado está vazio, (iii) São aplicadas várias regras para verificar a integridade do dado, como por exemplo, verificar se o dado é do tipo número ou não.

Com a integridade de todos os campos validada, é gerada uma entidade parcial a partir desses dados. Na entidade, são aplicadas várias regras que dependem de serviços externos e consultas ao banco.

Outro aspecto que precisa ser mencionado, é que a saída gerada pela rotina são mensagens de erro na validação dos dados ou a entidade gerada a partir desses dados.

Foi consultado o desenvolvedor Sênior do sistema, ele definiu dois pontos que a rotina deve ter. O primeiro, a rotina deve parar o processamento se uma exceção ocorrer no sistema e retorná-la como resposta. O segundo, a rotina deve ter uma forma simplificada para definir as regras a serem aplicadas.

Com base na análise feita e nas futuras funcionalidades do sistema, foram definidos os seguintes requisitos para a nova rotina são:

- Regras de validações são anuais, portanto, os dados do ano devem ser validados com as regras do mesmo ano.
- Além das regras serem por ano, elas podem não ser aplicadas para algumas UGs em períodos diferentes dentro do ano.
- Deverá gerar dois tipos de erros, o tipo aviso e o tipo erro.
- Se ocorrer exceção, o processamento deve parar imediatamente e retornar essa exceção.
- A rotina deve ser otimizada e paralela.
- A rotina deverá receber os dados de duas formas, por meio do tipo *File* e por meio do tipo *JsonValue*.
- O resultado dessa rotina deve ser as entidades geradas e/ou os erros gerados das validações.

### 3.1.2 Desenvolvimento

Para o desenvolvimento dessa nova rotina, era necessário um paradigma que provesse uma forma de escolher quais regras seriam aplicadas na validação dos dados. O primeiro paradigma que foi selecionado para ser utilizado foi o Paradigma Reflexivo, mas como visto na seção 2.3, esse paradigma tem uma desvantagem crucial, postergar as decisões para o tempo de execução aumentariam a quantidade de falhas, além de que deixaria de usar o sistema de tipos da linguagem.

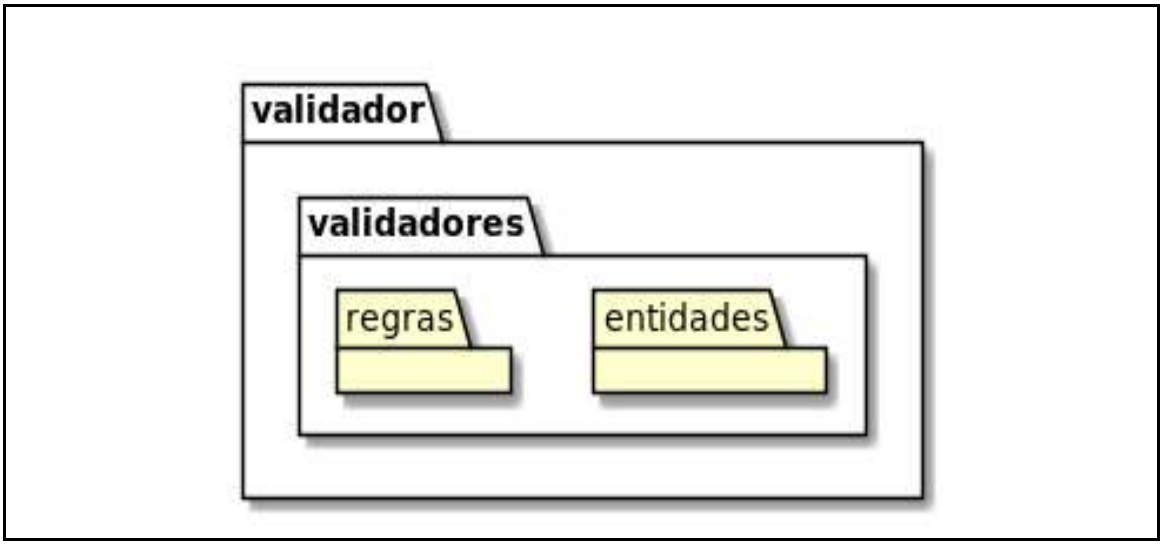
Como a primeira opção não se fez vantajosa, foi necessária uma nova escolha. A segunda escolha foi mesclar o Paradigma Funcional com o OO, todos os dois são suportados pela linguagem Scala.

*a) Designs da Rotina*

Os pacotes são *namespaces* e podem ser entendidas como pastas de um sistema operacional. Agrupando classes, interfaces e *traits* com contextos parecidos dentro de um pacote mantém uma aplicação mais organizada. (JAVA. 2017)

A rotina anterior não era muito elaborada, portanto não foi necessário a separação por pacotes. Na nova rotina se fez necessário essa separação. Como representada na Figura 11, há o pacote validador, no qual ficará as classes necessárias para o funcionamento da rotina, como as classes que representarão os resultados desta rotina. No pacote validadores, ficarão dois pacotes, o pacote regras, que será responsável por conter todas as regras de validação, o pacote entidades será responsável por conter todas as estruturas de dados que representam os arquivos a serem validados.

**Figura 11 - Diagrama dos pacotes.**

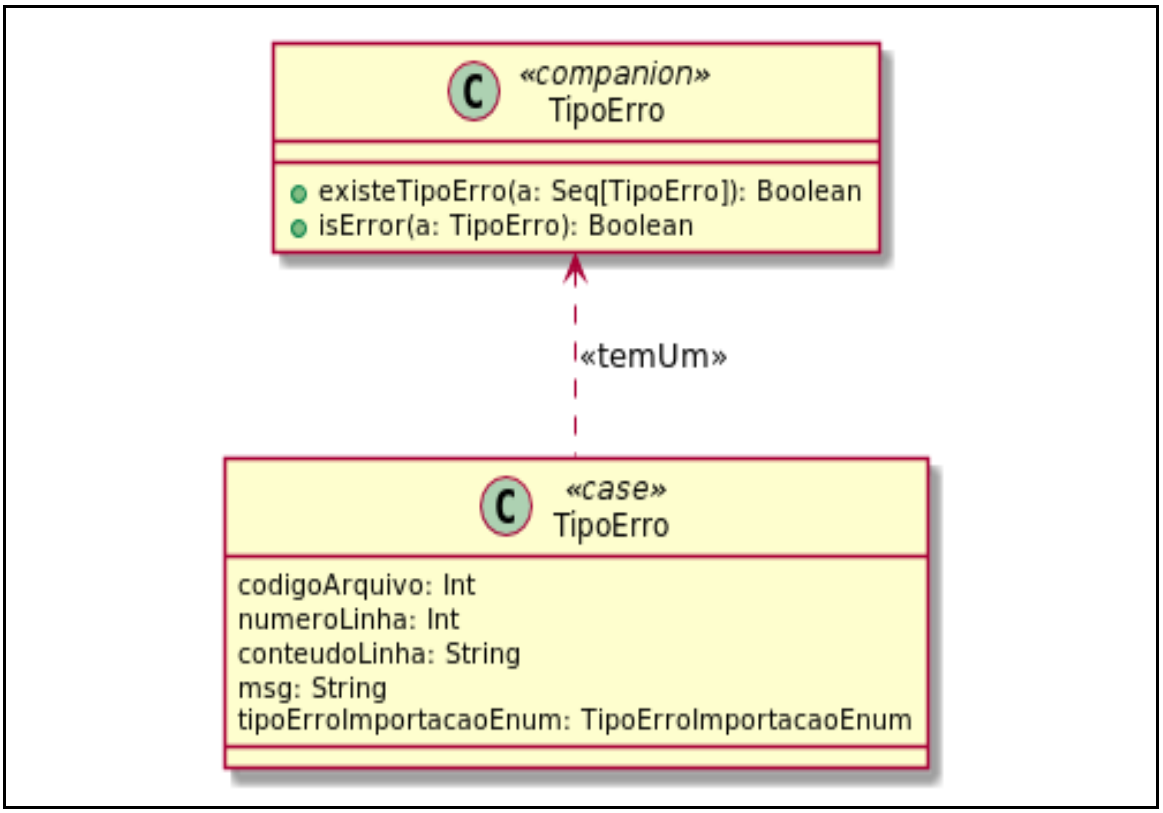



Fonte: Próprio Autor.

Na rotina anterior, os erros eram adicionados a uma exceção para serem lançados ao final da rotina. Devido a FP não aceitar efeitos colaterais, foram desenvolvidas classes para representar os erros gerados das validações, como demonstrada na Figura 12. Como observado na análise, existem dois tipos de erro, o tipo *error* e o *waning*. Para abstrair todos os dados necessários do erro, foi criada

uma classe TipoErro, com ela temos um *companion object* para manipular um objeto TipoErro.

**Figura 12 - Diagrama do TipoErro.**



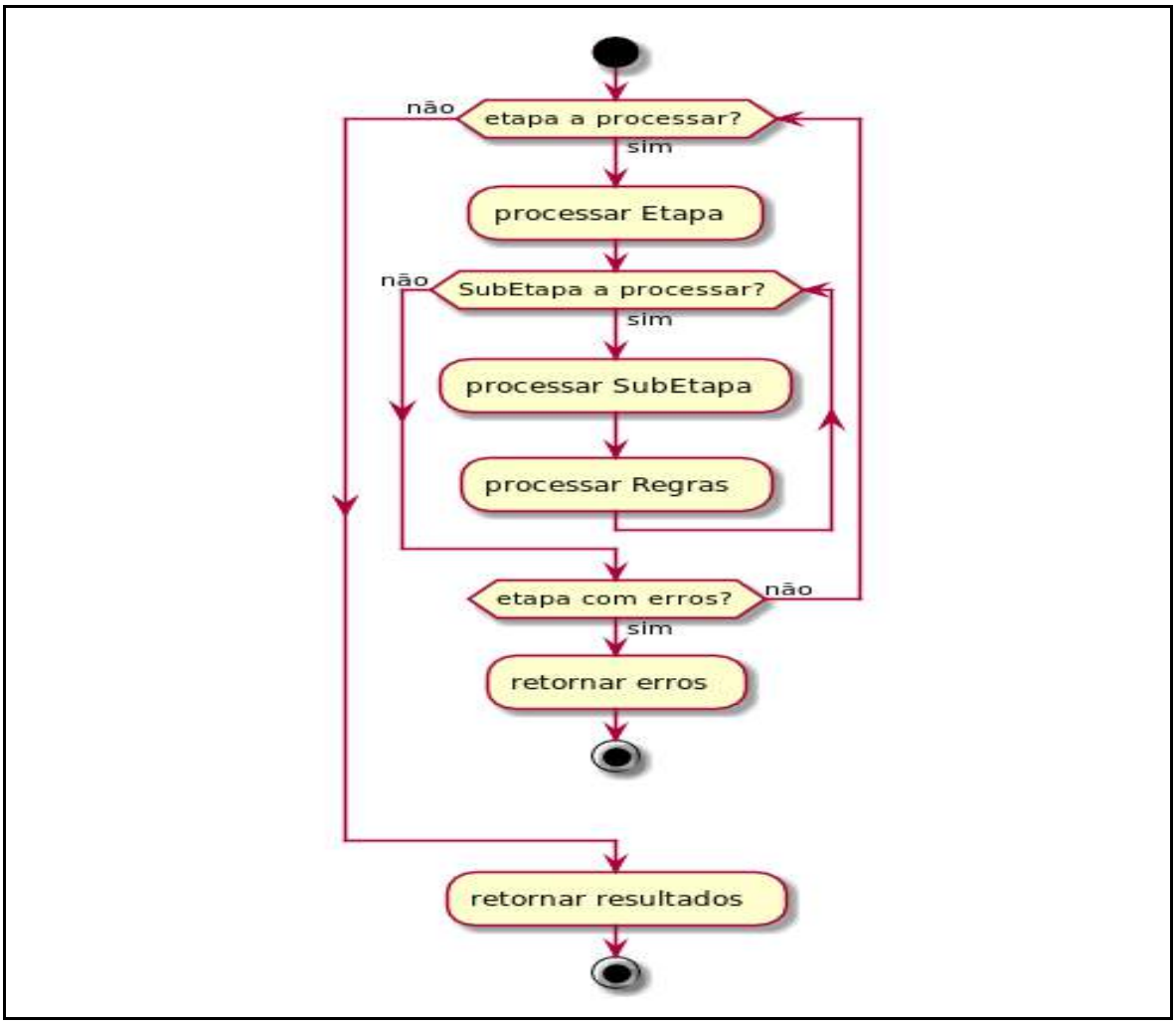

Fonte: Próprio Autor

No processamento das linhas, foi desenvolvido duas classes para identificar as etapas desse processo. A primeira classe é a *Etapa*, essa classe representa a etapa da validação, quando um dado depende da validação de outro, cria-se uma *Etapa*, por exemplo, criamos duas etapas, a primeira irá se certificar se o dado é íntegro, ou seja, se ele pode ser convertido para o tipo necessário. A segunda etapa irá utilizar os dados para realizar consultas no banco de dados. Com isso, a classe *Etapa* foi desenvolvida para representar essa dependência.

Em algumas etapas do processamento, será necessário de realizar uma consulta no banco para realizar o processamento de algumas regras, por exemplo, tenho três regras que dependem de dados do banco, pode-se fazer três consultas e validar o dado, mas estamos fazendo consultas desnecessárias. Para evitar esta

situação, foi desenvolvido uma segunda classe, a *SubEtapa*, com esta classe, pode-se agrupar as regras que precisam do mesmo dado e que esteja no banco ou em outros serviços. Esta forma de processamento está representada na Figura 13.

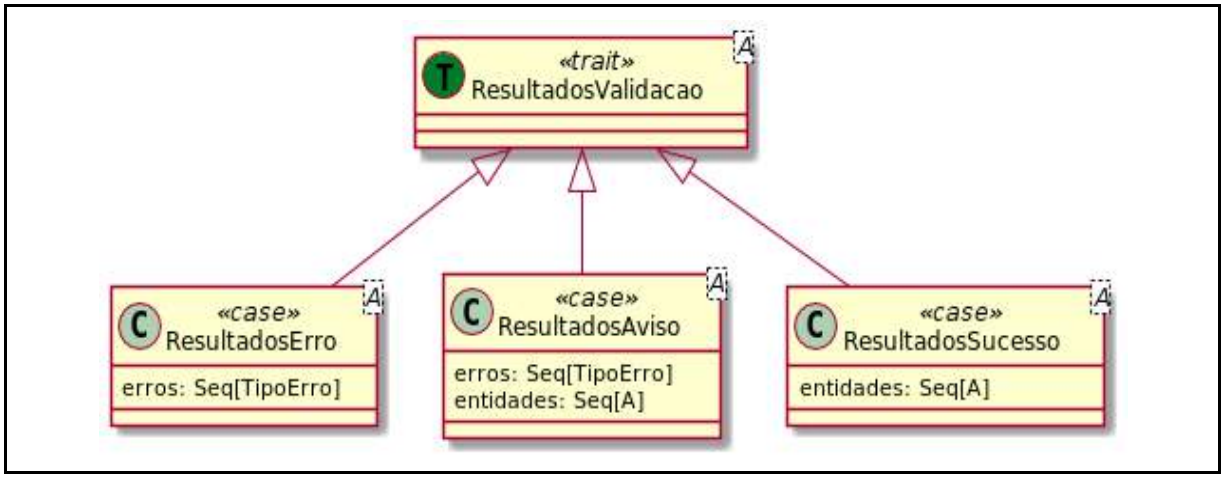
**Figura 13 - Diagrama de atividade das Etapas.**




Fonte: Próprio Autor.

As possíveis respostas da rotina são *ResultadosErro*, *ResultadosAviso*, *ResultadosSucesso*, como demonstrado na Figura 14. Todas essas classes encapsulam os resultados, para que possam ser obtidos por meio de uma funcionalidade da linguagem Scala, chamada *pattern matching*.

**Figura 14 - Diagrama dos tipos de resultados.**



Fonte: Próprio Autor.

Algumas classes que serão desenvolvidas não precisam de UML, pois são classes independentes e é necessário somente das suas responsabilidades. Essas classes são: (i) Validador é um *object* de Scala responsável pela interface com a rotina, onde receberá as requisições e retornará os resultados; (ii) Rotina é uma *trait* responsável por chamar os métodos de processamento; (iii) Regra é uma classe que representa a regra a ser aplicada; (iv) MetaDados é uma classe que encapsula todos os dados necessários para o processo de validação; (v) Validadores é um *object* responsável por retornar às abstrações dos dados e as regras a serem aplicadas; (vi) Processadores é um *object* responsável por conter todos os métodos de processamento.

### 3.1.3 Testes e métricas

Testes são muito importantes no desenvolvimento de um *software*. Com eles se pode descobrir defeitos, verificar anomalias, informações sobre os atributos não funcionais e se o *software* foi desenvolvido de acordo com os requisitos (SOMMERVILLE, 2016).

Inicialmente é importante comentar sobre o ambiente em que os testes foram realizados, este possui as seguintes configurações:

- **CPU:** Intel® Core™ i3-2330M CPU @ 2.20GHz × 4
- **Memória:** 4GB DDR3
- **Sistema:** Ubuntu 16.04 LTS 64-bit

- **HD:** 500GB 5400 RPM

Para a realização dos testes foi desenvolvido uma *microbenchmark* para avaliar o tempo de execução da rotina atual e da rotina desenvolvida neste trabalho.

> *"microbenchmark é um teste desenhado para medir uma pequena unidade de desempenho: o tempo de chamada de um método síncrono contra um método assíncrono; a sobrecarga em criar uma thread contra usar uma thread pool; o tempo de execução de um algoritmo aritmético contra uma implementação alternativa; e por assim em diante."* (OAKS, p. 11, 2014)

a) MICROBENCHMARK

Para testar as rotinas foram criados quinze arquivos numerados de 1 a 15, cada um contendo linhas utilizando a seguinte fórmula: número do arquivo * 400. O número 400 dessa fórmula são linhas contendo dados reais, mas como a rotina têm uma verificação de duplicidade das linhas, foi necessário gerar novas linhas com números sequenciais de 1 até N, no qual N será o número total de linhas do arquivo. Com isto, cada linha será única, possibilitando a rotina validar normalmente os arquivos.

Com base nestes arquivos foi desenvolvido o *microbenchmark* demonstrado na Figura 15.

Para a realização dos testes, é necessário refatorar todas as dependências das rotinas, retirando todas as utilizações de serviços externos como banco ou servidores *Web*, ou seja, as regras que utilizam esses serviços estão consultando dados em memória.

Para realizar os testes, todos os dados, exceto os dos arquivos, estarão na memória. Com isso, as rotinas mostrarão seu real desempenho. Foi pensando em um outro cenário como uma consulta a outro serviço ou banco de dados, quando este tipo de consulta é realizado o sistema fica esperando o resultado retornar para continuar o processamento.

**Figura 15 - *Microbenchmark* de tempo de execução.**

```
def tempoExecucaoPorArquivoEhQuantidadeLinha[R](bloco:
(File) => R): String = {
 val quantidade = 1 to 15
 val arquivos = quantidade.map(vez => (vez, new
File(s"src/main/resources/arquivos-testes/vezes$vez.txt")))
 val resultados = arquivos.map {
   tuplaVezArquivo =>
     val (vezes, arquivo) = tuplaVezArquivo
     (Metricas.tempoExecucaoComResultados {
       bloco(arquivo)
     }, vezes)
 }
 val resultadosPorLinha = resultados.foldLeft("") {
   (acumulador, proximo) =>
     val ((res, tempo), vezes) = proximo
     acumulador + "\n" + s"Resultado de ${vezes * 400}
linhas: $tempo ms"
 }
 resultadosPorLinha + "\n" + s"O tempo total foi:
${resultados.foldLeft(0.toLong)(_ + _._1._2)} ms"
}
```

Fonte: Próprio Autor.

Como todos os dados estão em memória e para não afetar os resultados dos testes, foi utilizado uma função da linguagem que faz a t*hread* fique em estado ocioso por um determinado período, com isto se consegue simular o tempo de espera.

As rotinas serão testadas em dois cenários principais:

- Cenário A: Como demonstrado na Figura 15, serão entregues 15 arquivos, cada um contendo quantidades de linhas diferentes, no qual o próximo arquivo terá 400 linhas a mais que o anterior, e os resultados serão retornados pela rotina.
- Cenário B: Será realizado da mesma forma que o cenário A, mas em uma das regras será utilizada a função que deixa a *thread* ociosa, o tempo de espera foi definido em 500.000 nanosegundos.

Os resultados do tempo de execução em milissegundos serão retornados pela função demonstrada na Figura 16, que será obtido por meio da seguinte fórmula **P - A**, onde P é o tempo atual do sistema depois que a rotina retornar os resultados e A é o tempo atual do sistema antes de executar a rotina.

**Figura 16 - Função que calcula o tempo de execução.**

```
def tempoExecucaoComResultados[R](bloco: => R):(R, Long)= {
 val t0 = System.currentTimeMillis()
 val result = bloco
 val t1 = System.currentTimeMillis()
 (result, t1-t0)
}
```

Fonte: Próprio Autor.

Em *microbenchmarks*, alguns fatores como processos em segundo plano podem afetar o resultado dos testes (OAKS, 2014). Para evitar divergências nos resultados, cada cenário e rotina serão realizadas 10 tentativas, e, ao final, será tirada a média aritmética destas tentativas.

# Capítulo 4. IMPLEMENTAÇÃO DA NOVA ROTINA

Para implementar a nova rotina e exemplificar como será utilizada, foi necessário extrair uma entidade do sistema SAGRES e todas as suas dependências. Começando pela interface da rotina, que é o objeto *Validador*. Os desenvolvedores criam métodos neste objeto, para validar cada entidade e os métodos chamam os processos de validação predefinidos pela *trait* Rotina.

Na *trait* Rotina, são desenvolvidas duas formas de processamento, o método *validarFromFile que* faz o processamento sequencial em uma única *thread*, o método *validarFromFileParalelo que* faz o processamento de forma paralela em múltiplas *threads*. Um detalhe importante está na assinatura do mesmo, como demonstrado na Figura 17, o parâmetro *gerarTuplaProcessadorEtapas* é uma função que retorna uma tupla contendo um conversor e as etapas que são processadas.

**Figura 17 - Assinatura do método de Rotina.**

```
def validarFromFile[A](file: File,
                       dataCompetencia: Date,
                       unidadeGestoraArquivo: String,
                       controleArquivo: ControleArquivo,
                       gerarTuplaProcessadorEtapas:
(MetaDados) => Try[(Conversor[A], Seq[Etapa])]
                       ): Try[ResultadosValidacao[A]] = {
     /*Código omitido*/

}
```

Fonte: Próprio Autor.

Na rotina atual do SAGRES, baseado no paradigma OO, é utilizado uma iteração utilizando o laço *for*, obtendo a linha pelo índice e passando-a para o validador processar, como demonstrado na Figura 18. Também é possível observar que é adicionado às entidades na lista *acoes*, mudando o estado da lista.

Como mencionado, no processamento das linhas são desenvolvidas duas formas. A primeira forma é utilizada uma estrutura de dados chama *List*, na linguagem Scala, essa estrutura é do tipo *last-in-first-out* (LIFO), imutável e simplesmente encadeada. Devido a estas características, estes tipos de estruturas são excelentes para aplicar transformações em todos os elementos. Utilizando a cabeça que é o último elemento e a cauda que é o restante da lista, se consegue manter um custo constante de processamento (ODERSKY, SPOON, VENNERS, 2016).

**Figura 18 - Processamento de forma OO.**

```
val acoes = ListBuffer[Option[Acao]]()
val linhas = arquivo.getLines().toList
/* Código omitido */
 for (i <- linhas.indices.toList) {
   val line = linhas(i)
   val acao = new AcaoFileLineValidator(/* Código omitido
*/).getEntidade(/* Código omitido */)
   if(acao.isDefined) acoes += acao
 }
/* Código omitido */
```

Fonte: Próprio Autor.

Na Figura 19, está o processamento utilizando a *List*, no trecho *tuplaLinhaIndice :: proximas*, o que está a esquerda do "::" é a cabeça da lista e a direita está a cauda, assim que é finalizado o processamento da cabeça, a função *loop* é chamada de forma recursiva passando a cauda e o resultado como parâmetros.

**Figura 19 - Processamento das linhas de uma Lista.**

```
@tailrec
def loop(linhasAhProcessar: Seq[(String, Int)],
          acumulador: (Seq[TipoErro], Option[Seq[A]]) =
(Seq(), Option(Seq()))
       ): Try[ResultadosValidacao[A]] = {
 linhasAhProcessar match {
   case Nil => /* Código omitido */
   case tuplaLinhaIndice :: proximas =>
     /* Codigo omitido */
 }
}
```

Fonte: Próprio Autor.

A segunda forma de processamento desenvolvido foi o processamento em paralelo. Para esta forma de processamento, é utilizada uma estrutura de dados chamada *Vector*, essa estrutura é uma árvore binária e imutável (WAMPLER, PAYNE, 2015).

A estrutura *Vector* foi escolhida por possuir uma forma paralela chamada *ParVector*, sendo transformada quando o método *par* é chamado. As operações em paralelo, de acordo com Prokopec e Miller (2018), “são realizadas recursivamente, dividindo a coleção, aplicando as operações nas partes divididas e recombinando todos os resultados quando finalizado”.

A Figura 20 demonstra a aplicação deste processamento paralelo, no qual o *Vector tuplasLinhaIndice* é convertida em uma estrutura paralela, aplicando uma transformação com o método *map* e reduzindo a um único valor com o método *foldLeft*.

**Figura 20 - Processamento das linhas em paralelo**

```
def processarLinhasParalelo[A](/* Código omitido */):
Try[ResultadosValidacao[A]] = Try {
  /* Código omitido */
   val result: (Seq[TipoErro], Option[Seq[A]]) =
tuplasLinhaIndice.par.map(linhaIndice => {
     /* Código omitido */
   }).foldLeft(/* Variável acumuladora */)(/*Função de
iteração*/)
   /* Código omitido */
}
```

Fonte: Próprio Autor

No SAGRES, as regras e os arquivos podem mudar de um ano para outro. Pensando nessa característica, foi desenvolvido o objeto *Validadores* que irá conter os métodos que retornam os dados e as regras a serem utilizadas. Os métodos utilizam funções parciais para verificar quais dados e regras são aplicados para o período específico, como demonstrado na Figura 21.

Na Figura 22, tem-se um exemplo de como as regras eram aplicadas, com a utilização de um *if* e uma condição, o erro é adicionado a uma exceção. Desta forma, todo os erros são guardados dentro da exceção para posteriormente ser lançada. O problema da estratégia é que ela é utilizada para guardar todos os erros gerados das linhas e ao adicionar o erro na exceção, ocorre a mudança de estado, sendo assim a exceção é mutável. Devido a mutabilidade da exceção, fica difícil a paralelização, pois os efeitos colaterais se tornam não determinísticos, ou seja, os resultados podem não estar corretos (PROKOPEC, MILLER, 2018).

**Figura 21 - Exemplo do método que retorna o dado e as Etapas.**

```
protected[validador] object Validadores {
 def validarAcao(metaDados: MetaDados): Try[(Conversor[Acao],
Seq[Etapa])] = {
   val default: PartialFunction[MetaDados, (Conversor[Acao],
Seq[Etapa])] = {
     case meta if meta.dataCompetencia.toLocalDate.getYear ==
2018
  =>
       (
         ArquivoAcao,
         Seq(
           Etapa(Seq(SubEtapa(/* Regras */))),
           Etapa(Seq(SubEtapa(/* Regras */)))
         )
       )
   }
   Try(default(metaDados))
 }
}
```

Fonte: Próprio Autor.

**Figura 22 - Exemplo de Regra em OO.**

```
private lazy val getDenominacaoAcao = {
 val minDescricao = 10
 if(denominacaoAcaoStr.get.length < minDescricao) {
   erros.adicionarErro(controle.codigoArquivo, lineNumber, line,
   s"Descrição da ação não deve ser inferior a $minDescricao
   caracteres. $msgIdentificadora", TipoErroImportacaoEnum.ERROR)
 }
 denominacaoAcaoStr
}
```

Fonte: Próprio Autor.

Para tornar possível a paralelização, foi desenvolvido a classe *Regra* que representa a regra a ser aplicada no dado, a classe Regra possui um único atributo que é essa função literal *(EntidadeArquivo, MetaDados) => Try[Option[TipoErro]]*.

Para ajudar o desenvolvedor, foram identificados os pontos em comum na criação da regra e desenvolvido um *Factory* para *Regra*, de acordo com Wampler e Payne (2015) “os métodos apply dos objetos são métodos construtores de uma instância”. O método *Factory* da *Regra* está demonstrado na Figura 23, omitindo o terceiro grupo de parâmetros da função *fazerRegra*, passa-se uma função com estes parâmetros omitidos, com isso, faz-se uso do *currying* disponível no Scala.

Fazendo uso da modularidade do *Factory* de *Regra* se consegue um fácil reaproveitamento das declarações das regras. Como demonstrado na Figura 24, a função de decisão pode ser declarada de forma isolada e fácil de ser testada.

**Figura 23 - *Factory* de Regra.**

```
def apply[T](/* Código omitido */)(/* Código omitido */):
Regra = new Regra(fazerRegra(tipo, mensagem)(decisao))

def fazerRegra[T <: EntidadeArquivo](/* Código omitido
*/)(/* Código omitido */)(/* Código omitido */):
Try[Option[TipoErro]] = {
 decisao(entidadeArquivo.asInstanceOf[T],
dadosValidacao).map {
   decisao => {
     if(decisao){
       Option(
         TipoErro(/* Código omitido */)
       ) } else None
   }
 }
}
```

Fonte: Próprio Autor.

**Figura 24 - Exemplo de como criar Regra.**

```
/* Código omitido */
def naoContemCodigoAcao(arquivoAcao: ArquivoAcao,
metaDados: MetaDados) = Try(arquivoAcao.codigoAcao.isEmpty)


def regraNaoContemCodigoAcao: Regra =
Regra(TipoErroImportacaoEnum.ERROR, "Código da ação deve
ser informado.")(naoContemCodigoAcao)
/* Código omitido */
```

Fonte: Próprio Autor.

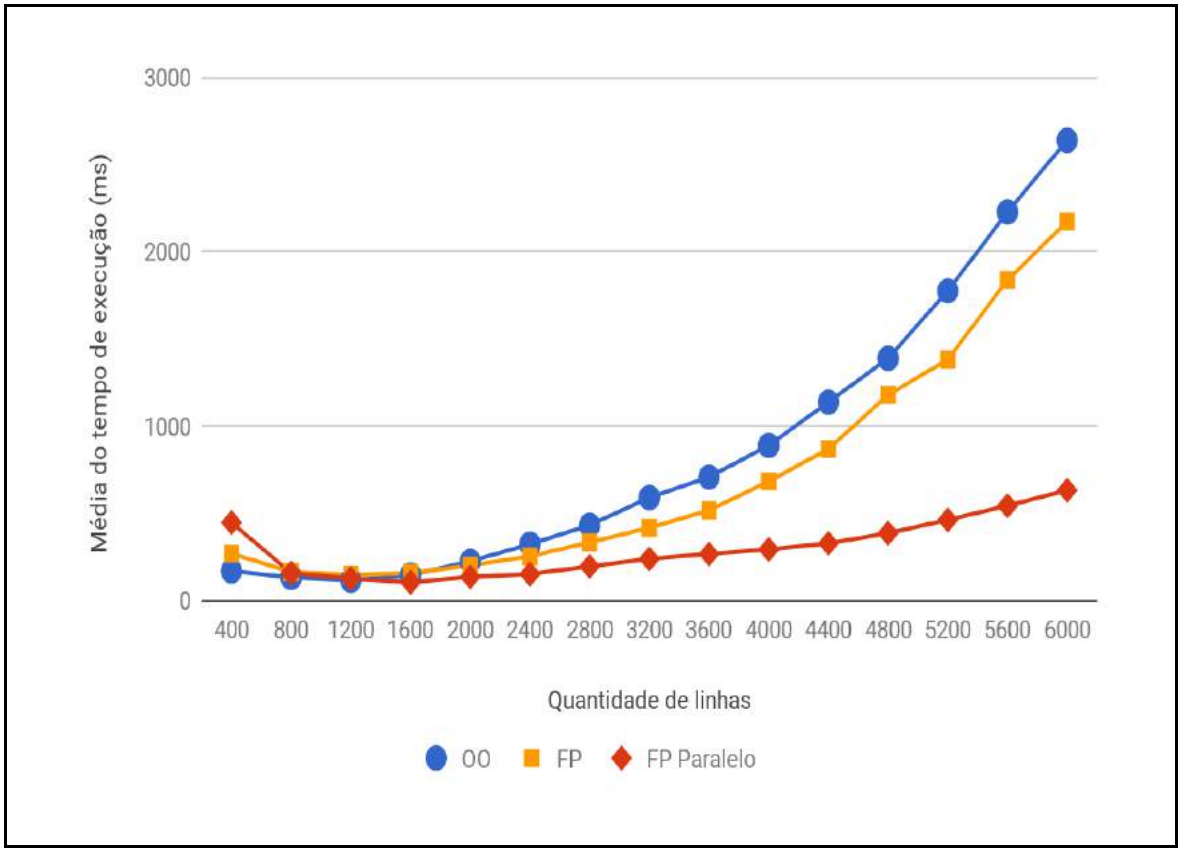
## 4.1 RESULTADO DOS TESTES

Realizado os testes descritos na seção 3.1.3, o presente estudo obteve os seguintes resultados. No Gráfico 1, tem-se o cenário A, no qual os testes foram realizados com todos os dados em memória. No Gráfico 2, há o cenário B, no qual possui as mesmas características do cenário A, mas com uma única regra com tempo de espera.

Na Figura 25, pode-se observar três pontos importantes. O primeiro ponto é o início, neste ponto há um desempenho diferenciado entre OO, FP e FP Paralelo. No OO, há menos código para interpretar/compilar, ou seja, é uma estrutura menor do que os outros dois. A diferença entre o FP e FP Paralelo, apesar de usar quase as mesmas estruturas, é devido ao FP Paralelo utilizar das ferramentas de paralelização.

Após o tempo de aquecimento da JVM, que ocorre durante o teste das 400 linhas, há o segundo ponto destacado que ocorre entre a quantidade de linhas 800 e 1200. Neste ponto, é possível observar resultados parecidos e que as estruturas de dados escolhidas não fizeram diferença devido à pequena quantidade de linhas.

**Figura 25 - Avaliação de desempenho no Cenário A.**

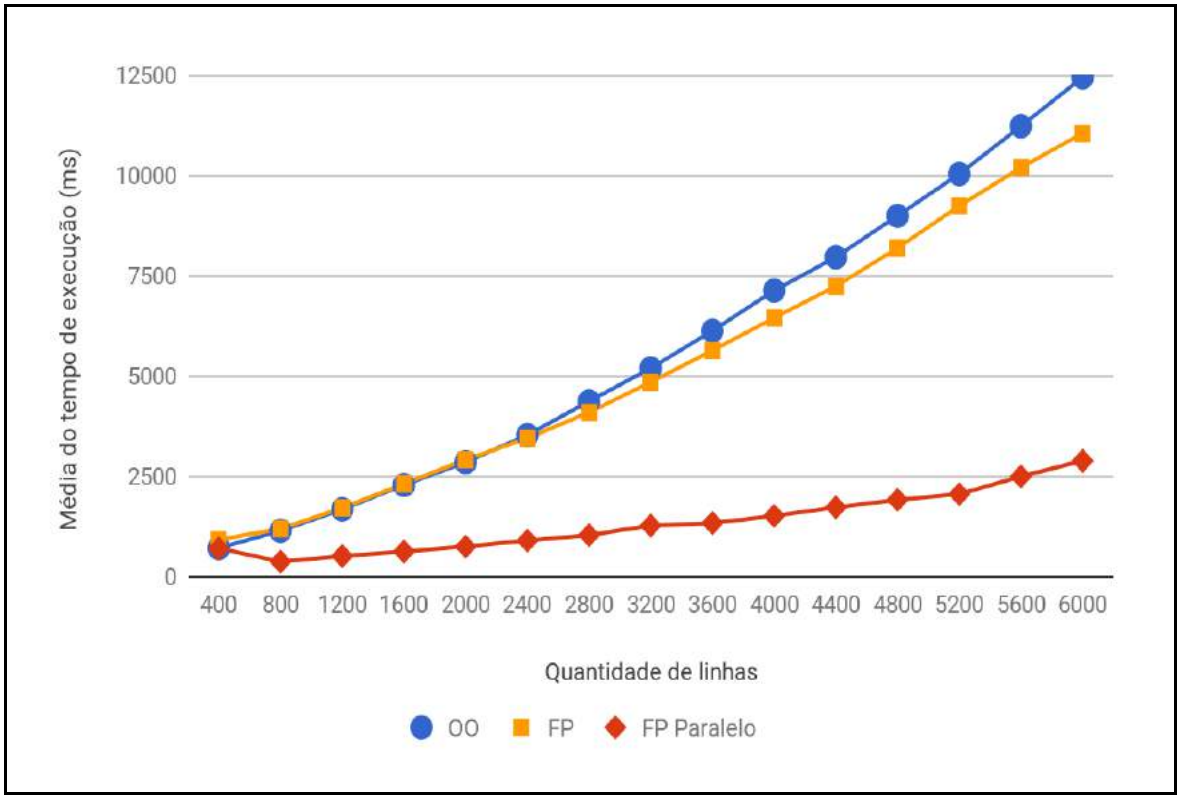



Fonte: Próprio Autor.

O terceiro ponto observado inicia na quantidade de linhas 2000. A diferença entre OO e FP é observado na Figura 18, no trecho *arquivo.getLines().toList* está sendo utilizada um *List*, sendo que o acesso ao índice dessa estrutura de dados têm um custo linear, ou seja, vai aumentando de acordo com o tamanho da lista. A diferença entre FP e FP Paralelo ocorre justamente no uso da paralelização do processamento e no uso da estrutura de dados *Vector*.

No Figura 26, é observado que no início, no ponto onde ocorre o aquecimento da JVM, as diferenças observadas no Figura 25 foram compensadas pelo tempo de espera da *thread*. O segundo e o terceiro ponto observados no Figura 25, ocorrem somente entre OO e FP, a diferença que ocorre entre esses dois e FP Paralelo ocorre de 800 linhas em diante. Esta diferença se refere ao processamento em paralelo ocorre em múltiplas *threads*, por conta disso, o tempo de espera total deste processamento é bem menor.

**Figura 26 - Avaliação de desempenho no Cenário B.**




Fonte: Próprio Autor.

Baseado nos dois cenários, pode-se concluir que a diferença do tempo de execução entre OO e FP ocorreu devido ao uso de diferentes tipos de estruturas de dados da linguagem Scala e não do paradigma em si. Sendo que, o uso dos conceitos da FP nos garante um bom uso da paralelização e concorrência, como demonstrado nos gráficos.

Além dos resultados demonstrados anteriormente, também foram gerados alguns pontos baseados nas experiências obtidas no desenvolvimento deste projeto. Esses pontos estão definidos no Quadro 1.

No Quadro 1, o quesito de desempenho já foi discutido anteriormente, onde na análise dos resultados foi observado que os paradigmas não afetaram o tempo de execução, com o aumento da quantidade de linhas, o desempenho das estruturas de dados começou a fazer a diferença.

Na possibilidade de paralelização, na rotina OO, os erros eram adicionados na exceção, mudando seu estado. Paralelizar essa rotina fariam esses erros serem

não determinísticos. Na rotina FP, utilizando os conceitos de imutabilidade e de transparência referencial, foi possível paralelizar o processamento das linhas.

**Quadro 1 - Quadro comparativo entre os paradigmas.**

| | **FP** | **OO** |
|---|---|---|
| **Desempenho** | Depende das estruturas de dados. | Depende das estruturas de dados. |
| **Possibilidade de Paralelização?** | Sim, devido a imutabilidade. | Não, devido a mutabilidade. |
| **Dificuldades** | Trabalha com o retorno das exceções de cada função.<br><br>Atomizar métodos em funções. | Efeitos colaterais. |
| **Vantagens** | Retorna tudo que foi processado.<br><br>Modularidade das funções. | Facilidade no uso de exceções. |
| **Curva de aprendizagem** | Alta. | Média. |

Fonte: Próprio Autor.

Na FP, retornar todas as exceções é algo trabalhoso de se lidar, principalmente devido à falta de experiência com o paradigma. Outra desvantagem é a dificuldade de reduzir as funções para funções menores, com o intuito que façam uma única operação lógica. No OO, alguns efeitos colaterais, como adicionar os erros na exceção, deixam o código obscuro por não saber o que a rotina irá fazer com o objeto da exceção.

Apesar das desvantagens dos efeitos colaterais do OO, lançar exceções é bem mais simples de se lidar, um único *try catch* envolvendo o código resolveria o tratamento das exceções. Na FP é diferente, todas as funções retornam o que foi processado por ela. Além disto, utilizando-as com outras funções que as aceitam como parâmetro, deixam o código mais modular.

Por fim, a curva de aprendizagem, a proximidade dos conceitos de OO com o mundo real deixam a curva menor do que a FP.

# Capítulo 5. CONSIDERAÇÕES FINAIS

Os resultados obtidos no presente trabalho, mostram que o uso dos paradigmas não influenciou no desempenho da rotina, mas com o uso dos conceitos da FP se tornou possível o uso da programação paralela, garantindo resultados determinísticos.

Além dos resultados de desempenho observados, o uso de funções para representar as regras possibilitaram uma melhor modularização e visibilidade de quais dados e regras estão sendo utilizadas.

Algumas dificuldades foram encontradas na pesquisa dos testes e métricas para serem utilizadas neste trabalho. Os testes encontrados para comparar os paradigmas dependiam muito de como o código foi desenvolvido, além de que os códigos de cada paradigma eram desenvolvidos pela mesma equipe e em certas condições, devido a estes fatores, esses testes não poderiam ser implementados neste trabalho em razão da rotina OO não foi desenvolvida neste trabalho.

O aprendizado da FP aconteceu durante o desenvolvimento da rotina FP, por conta disso, ocorreram várias mudanças se adequando aos conhecimentos adquiridos, outro fator que contribuiu para as mudanças foram a quantidade de ferramentas que a linguagem Scala possui. Um bom exemplo é a quantidade de coleções, possui estruturas imutáveis e mutáveis, cada uma com características de desempenho diferentes para cada operação diferente.

O uso da FP se mostrou vantajosa para lidar com grande quantidade de dados, com a possibilidade de paralelização é possível processar uma grande quantidade de dados com melhor desempenho. O uso do OO foi essencial para encapsular dados e funções com o mesmo contexto.

## 5.1 Trabalhos futuros

A rotina desenvolvida trouxe um comparativo relevante para utilização de outros paradigmas no contexto estudado. Contudo, outra consequência pode ser

destacada: o conhecimento adquirido durante o desenvolvimento, mas com o uso diário na linguagem Scala alguns pontos podem ser melhorados:

- No código desenvolvido, foi utilizado os dois paradigmas o OO e o FP, só que alguns pontos foram chamados métodos diretamente e algumas consultas a dados fora da função. Para evitar isso, poderia ser utilizado o *currying* para deixar o código mais funcional.
- Algumas classes como Etapa e SubEtapa foram declaradas como *case class*, esse tipo de classe ganha *companion objects* com métodos implementados, sendo que essa implementação não era necessária, desenvolvendo uma classe normal e um *companion object* com os métodos *apply* seriam mais performáticos.

# Capítulo 6. REFERÊNCIAS


ANTANI, Ved; TIMMS, Simon; MANTYLA, Dan. **Javascript**: Functional Programming for Javascript Developers. Birmingham: Packt, 2016.

BAESENS, Bart; BACKIEL, Aimee; BROUCKE, Seppe vanden. **Beginning Java Programming:** The Object-Oriented Approach. Birmingham: Packt, 2015.

CHIUSANO, Paul; BJARNASON, Runar. **Functional Programming in Scala**. New York: Manning, 2015.

COBB, Charles G. **The project manager's guide to mastering agile**: Principles and practices for an adaptive approach. New Jersey: Wiley, 2015.

DEMERS, François-Nicola; MALENFANT, Jacques. **Reflection in logic, functional and object-oriented programming**: a Short Comparative Study. In: Workshop on Reflection and Metalevel Architectures and their Applications in AI, 1995, Québec. **Anais eletrônicos...** Montreal: Universidade de Montreal. Disponível em: <http://www-master.ufr-info-p6.jussieu.fr/2006/Ajouts/Master_esj_2006_2007/IMG/pdf/malenfant-ijcai95.pdf>. Acesso em: 14 de abril de 2018.

HILLAR, Gaston C. **Learning Object-Oriented Programming**. Birmingham, 2015.

JAVA. **What is a Package?** 2017. Disponível em: <https://docs.oracle.com/javase/tutorial/java/concepts/package.html>. Acesso em: 15 de abril de 2018.

MALENFANT, J.; JACQUES, M.; DEMERS, F.-N. **A Tutorial on Behavioral Reflection and its Implementation**. 1996. Disponível em: <http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf> Acesso em: 14 de abril de 2018.

OAKS, Scott. **Java Performace**: The Definitive Guide. Sebastopol: O'Reilly, 2014.

ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. **Programming in Scala**. 3 ed. California: Artima, 2016.

PROKOPEC, Aleksandar; MILLER, Heather. **Parallel Collections**: Overview. Disponível em: <https://docs.scala-lang.org/overviews/parallel-collections/overview.html>. Acesso em: 29 de abril de 2018.

SAUMONT, Pierre-Yves. **Functional Programming in Java**: How functional techniques improve your Java programs. New York: Manning, 2017.

SCOTT, Michael L. **Programming Languages Pragmatics**. 3 ed. Burlington: Morgan Kaufmann, 2009.

SEBESTA, Robert W. **Concepts of programming languages**. 11 ed. Harlow: Pearson, 2016.

SOMMERVILLE, Ian. **Software Enginnering**. 10 ed. Harlow: Pearson, 2016.

TIOBE. **TIOBE Programming Community Index**. 2018. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 10 de março de 2018.

WAMPLER, Dean; PAYNE, Alex. **Programing Scala**. 2 ed. Sebastopol: O'Reilly, 2015.

WARBURTON, Richard. **Java 8 Lambdas**. Sebastopol: O'Reilly, 2014.