UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
(UNIRIO)

MATEUS TEIXEIRA BANDEIRA DE ALBERTIN

FLUENTASYNC: UMA API FLUENTE PARA IMPLEMENTAÇÃO DE OPERAÇÕES ASSÍNCRONAS

RIO DE JANEIRO
2023

MATEUS TEIXEIRA BANDEIRA DE ALBERTIN

FLUENTASYNC: UMA API FLUENTE PARA IMPLEMENTAÇÃO DE OPERAÇÕES ASSÍNCRONAS

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação

Orientador: Prof. Jobson Luiz M. da Silva

RIO DE JANEIRO

2023

Catalogação informatizada pelo(a) autor(a)

A334 Albertin, Mateus Teixeira Bandeira de
FluentAsync: Uma API Fluente para Implementação de Operações Assíncronas / Mateus Teixeira Bandeira de Albertin. -- Rio de Janeiro, 2023.
84

Orientador: Jobson Luiz Massollar da Silva.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Estado do Rio de Janeiro, Graduação em Sistemas de Informação, 2023.

1. Programação Assíncrona. 2. APIs Fluentes. 3. Programação Declarativa. 4. HTTP. I. Silva, Jobson Luiz Massollar da, orient. II. Título.

MATEUS TEIXEIRA BANDEIRA DE ALBERTIN

FLUENTASYNC: UMA API FLUENTE PARA IMPLEMENTAÇÃO DE OPERAÇÕES ASSÍNCRONAS

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação

Aprovado em: ____/_____/____

Banca examinadora:

_____

Prof. Jobson Luiz Massollar da Silva (Orientador)
Universidade Federal do Estado do Rio de Janeiro — UNIRIO

_____

Prof. Márcio de Oliveira Barros
Universidade Federal do Estado do Rio de Janeiro — UNIRIO

_____

Prof. Paulo Sérgio Medeiros dos Santos
Universidade Federal do Estado do Rio de Janeiro — UNIRIO

RESUMO

Com a popularização de estilos arquiteturais de software que dependem fortemente da comunicação entre aplicações via APIs HTTP, tornou-se comum que desenvolvedores precisem adotar estratégias para reduzir o efeito da latência introduzida pela natureza da comunicação. A programação assíncrona é uma dessas estratégias, pois permite a realização de múltiplas operações em paralelo, reduzindo potencialmente o tempo total de processamento. Neste contexto, este trabalho apresenta o desenvolvimento e a avaliação da API FluentAsync, que tem como objetivo facilitar o uso de técnicas de programação assíncrona. A metodologia de desenvolvimento da API foi inspirada na DSR (Design Science Research), e a avaliação da API foi realizada a partir de métricas tradicionais no campo da Engenharia de Software. Os resultados dessa avaliação mostraram que o uso da FluentAsync pode trazer benefícios, como a redução do esforço necessário para desenvolver programas que fazem uso de rotinas assíncronas, assim como a diminuição da complexidade destes códigos, resultando numa maior legibilidade e manutenibilidade do código-fonte resultante.

Palavras-chave: Programação Assíncrona, APIs Fluentes, programação declarativa, HTTP

ABSTRACT

With the rising popularity of software architectures that heavily rely on HTTP API calls for communication, it has become usual that developers need to adopt strategies to reduce the effects of the latency added by the nature of the communication process. Asynchronous programming is one of these strategies as it allows multiple operations to be performed in parallel, potentially reducing the overall processing time. In this context, this work presents the development and evaluation of the FluentAsync API, which aims to ease the use of asynchronous programming techniques. The API development methodology was inspired by DSR (Design Science Research), and the API evaluation was carried out using traditional metrics in the field of Software Engineering. The results of this evaluation showed that the use of FluentAsync can bring benefits such as reducing the effort required to develop programs that make use of asynchronous routines, as well as reducing the complexity of these programs, resulting in greater readability and maintainability of the resulting source code.

Keywords: Asynchronous Programming, Fluent APIs, declarative programming, HTTP

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Os sistemas modernos estão se tornando cada vez mais interconectados e os usuários acostumaram-se a esperar uma integração profunda entre os diversos serviços ao seu dispor. Nesse cenário, as empresas perceberam que a troca de informações entre seus sistemas é uma questão de sobrevivência e, também, um negócio lucrativo. Quando se trata de dados, podemos dizer que *o todo é maior que a soma de suas partes*.

Para desenvolver sistemas que dependem de informações que estão em outros sistemas é necessário que haja algum mecanismo de comunicação entre eles, comumente denominado *chamada externa*, pois o sistema faz uma solicitação a outro sistema fora do seu contexto. Esse tipo de comunicação traz à tona um problema conhecido há décadas: chamadas externas, por sua natureza, são mais suscetíveis a altas latências e erros de comunicação. Nesse contexto, uma questão relevante é: como realizar chamadas para outros sistemas da maneira mais eficiente possível?

As técnicas de programação assíncrona são uma ferramenta efetiva para mitigar os problemas causados pela alta latência das chamadas externas. Por meio dela, é possível que o sistema realize outras tarefas em paralelo enquanto o resultado de uma chamada externa (ou até mesmo várias) não é retornado. Portanto, apesar de não reduzir o tempo de execução das chamadas individualmente, executá-las em paralelo pode resultar em uma economia significativa de tempo.

Tradicionalmente, programadores utilizavam *callbacks* para realizar operações assincronamente em suas aplicações: em suma, um *callback* é um mecanismo que permite o programador definir um código, geralmente uma função, que deverá ser executado no futuro, após o término de uma atividade assíncrona. Uma das desvantagens desta abordagem é o desmembramento da lógica da aplicação, pois a ordem de execução dos comandos deixa de ser definida pela sua disposição no código fonte, passando a depender de fatores externos como a latência de rede e os níveis de utilização dos recursos computacionais do dispositivo. Consequentemente, à medida que o fluxo do código se fragmenta, a manutenção do sistema torna-se mais complicada.

Existem outras iniciativas com objetivos similares, como o *Project Reactor*[1] e o *RxJS*[2], entretanto, este trabalho visa explorar novas abordagens acerca de técnicas de programação assíncrona voltadas para o melhoramento das chamadas a sistemas externos, de maneira a minimizar os problemas introduzidos por técnicas anteriores.

1.2 OBJETIVOS

Este trabalho visa desenvolver e avaliar uma API (*Application Programming Interface*) voltada para a execução e coordenação de tarefas assíncronas, com foco em consumo assíncrono de serviços externos. Diferentemente das APIs disponíveis, como as definidas no pacote *java.util.concurrent* do Java, e dos mecanismos de programação assíncrona embutidos em linguagens de programação como Java, C# e JavaScript, essa API, nomeada FluentAsync fornece um conjunto de abstrações de alto nível com o intuito de reduzir a complexidade e a quantidade do código necessário para realizar e coordenar operações assíncronas. Como consequência direta, espera-se um aumento na produtividade do desenvolvedor, uma vez que as abstrações permitirão que ele defina quais operações assíncronas serão executadas e como elas serão encadeadas, sem entrar nos detalhes de como elas serão implementadas na API da linguagem de programação. Entretanto, não é parte do escopo deste trabalho avaliar diretamente um aumento na produtividade do desenvolvimento, apenas sendo uma extrapolação plausível a partir das avaliações realizadas. Ainda, é importante ressaltar que não é um objetivo desse trabalho tratar da melhoria de performance agregada ao uso de técnicas de programação assíncrona. Neste sentido, consideramos estabelecido o fato de que paralelizar chamadas externas é uma maneira eficiente de reduzir tempos de processamento.

1.3 METODOLOGIA

As fases desse trabalho foram inspiradas na metodologia DSR (*Design Science Research*). O DSR é um conjunto de técnicas de pesquisa voltadas para a criação de novas técnicas e o melhoramento do conhecimento já existente (VAISHNAVI e KUECHLER, 2004).

Diferentemente de métodos de pesquisa mais tradicionais, pelos quais se investiga a ocorrência de um fenômeno, a DSR permite o desenvolvimento ou aprimoramento de soluções para um problema proposto.

[1] https://projectreactor.io
[2] https://rxjs.dev

Segundo Vaishnavi e Kuechler, o processo da DSR possui 5 etapas:

1. **Reconhecimento do Problema:** o problema é identificado e analisado, e, ao final desta etapa, é elaborada uma proposta de pesquisa;
2. **Sugestão:** após o reconhecimento do problema, são propostas novas técnicas para abordá-lo. A sugestão compreende novas formas de utilizar artefatos existentes ou não para solucionar o problema;
3. **Desenvolvimento:** nesta etapa, a solução elaborada no passo anterior é desenvolvida;
4. **Avaliação:** o artefato produto da fase de desenvolvimento é colocado à prova. Quando ocorrerem, expectativas não alcançadas devem ser citadas e explicadas;
5. **Conclusão:** é o encerramento do processo de pesquisa. Os resultados da pesquisa são descritos, e o conhecimento adquirido é classificado como "sólido" caso os fatos observados sejam consistentes e replicáveis, ou, caso contrário, é necessário que outros pontos sejam esclarecidos em outra pesquisa.

1.4 ORGANIZAÇÃO DO TEXTO

Neste trabalho, os conceitos fundamentais são abordados na Seção 2. A Seção 3 introduz as operações básicas consideradas durante o desenvolvimento da API, assim como suas combinações. A API é apresentada na Seção 4, onde a coleção de métodos disponíveis é descrita, e são dados exemplos de sua utilização. A Seção 5 expõe e interpreta as avaliações realizadas com o objetivo de comparar as implementações utilizando a API e as implementações *Vanilla*. A conclusão na Seção 6 traz um breve resumo sobre as contribuições deste trabalho, assim como sugestões de possíveis trabalhos futuros nesta área.

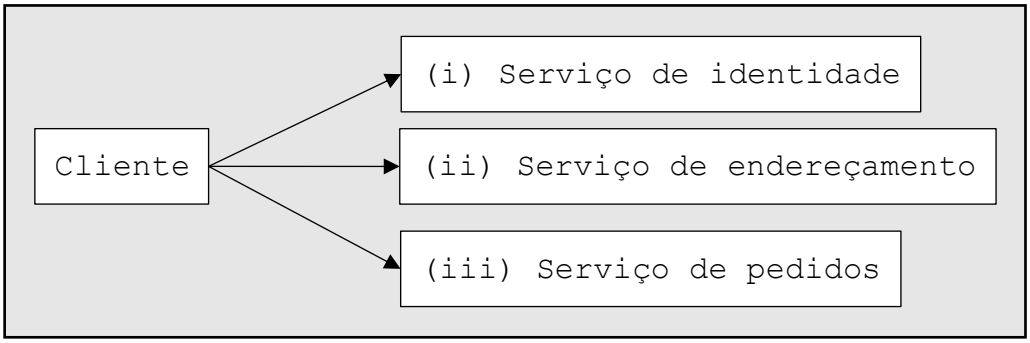
O APÊNDICE A define a especificação da API da FluentAsync. O APÊNDICE B apresenta uma implementação de referência da FluentAsync em Java. O APÊNDICE C inclui as classes de apoio utilizadas nos exemplos contidos na Seção 4.2. Uma implementação alternativa da FluentAsync em TypeScript está disponível no APÊNDICE D. Por fim, como citado na Seção 5.1.2, as Métricas de Halstead completas estão no APÊNDICE E.

2 CONCEITOS FUNDAMENTAIS

Do ponto de vista arquitetural, os sistemas e aplicações que adotam os padrões cliente-servidor e as arquiteturas orientadas à serviço, como Microsserviços, são o foco central deste trabalho. Dessa forma, esse capítulo tem como objetivo apresentar uma série de conceitos relacionados a esses tópicos.

Por exemplo, considerando um hipotético sistema gerenciador de vendas on-line, que possui a funcionalidade de imprimir etiquetas para os pedidos enviados pela transportadora. Cada etiqueta deve conter: (i) o nome completo do destinatário; (ii) o endereço do destinatário; (iii) a declaração de conteúdo do pacote contendo a lista dos produtos enviados. Em uma arquitetura de orientada a serviços, cada uma dessas informações poderá ser obtida a partir de um serviço diferente. A aplicação *Cliente* deverá interagir com cada um desses serviços a fim de realizar sua tarefa.

Figura 1 — Arquitetura orientada a serviços para a funcionalidade de etiquetagem



Fonte: Elaborado pelo autor.

Nesse contexto, o padrão HTTP desempenha um papel fundamental, pois sobre ele são definidas diversas tecnologias que permitem a comunicação entre os componentes de software que aderem a esse tipo de padrão arquitetural. Dois padrões comumente adotados para esse cenário são “Cliente-Servidor” e Microsserviços.

Independentemente de qual padrão arquitetural seja adotado, é comum existir a necessidade de obter múltiplos recursos remotos sem que haja relação de dependência entre eles. A Figura 1 ilustra o cenário em que três requisições são realizadas isoladamente; neste caso, as três requisições poderiam ser realizadas em paralelo utilizando técnicas de programação assíncrona. Ao paralelizar as requisições dos serviços é possível otimizar o uso dos recursos da aplicação Cliente e reduzir sensivelmente os tempos de carregamento vivenciados pelo usuário final.

Desenvolver uma aplicação altamente dependente de programação assíncrona não é uma tarefa trivial. Linguagens de programação distintas oferecem mecanismos diferentes para abordar o problema de coordenação e controle dos fluxos paralelos de processamento. Ainda assim, a atividade de desenvolver rotinas assíncronas permanece uma atividade cognitivamente onerosa ao programador, aumentando o potencial de surgimento de falhas no software resultante.

Nessa mesma linha de raciocínio, a adoção de uma linguagem específica de domínio para programação assíncrona pode ser uma abordagem útil para reduzir a complexidade dos códigos que utilizam esse tipo de programação. Por meio de uma nova camada de abstração, é possível criar significados semânticos para as operações assíncronas mais comuns, simplificando o código-fonte final. Por consequência, a susceptibilidade para o surgimento de problemas é reduzida.

2.1 ARQUITETURA CLIENTE-SERVIDOR

Segundo Bass, Clements e Kazman (2021, p. 165),

> O padrão cliente-servidor consiste em um servidor disponibilizando serviços simultaneamente a múltiplos clientes distribuídos. O exemplo mais comum é um servidor web disponibilizando informações a múltiplos usuários simultâneos de um website. (tradução nossa).

Este padrão de arquitetura permite que seja realizada uma requisição ao Servidor quando houver necessidade de algum processamento que o Cliente não seja capaz de realizar por falta de algum recurso, como dados ou software/hardware adequados. Por sua vez, o Servidor irá realizar o trabalho solicitado e enviar o resultado do processamento como resposta ao Cliente. Uma característica relevante desse padrão é que o Cliente precisa conhecer o Servidor para o qual ele irá enviar a requisição, por outro lado, o Servidor não precisa conhecer quais Clientes ele irá atender. Atualmente, é bastante comum que a comunicação entre o Cliente e o Servidor utilize o protocolo HTTP.

Entre as vantagens deste tipo de arquitetura estão:

- **Especialização do servidor:** Algumas tarefas, como a execução de algoritmos de reconhecimento de linguagem natural, podem não ser executadas em qualquer dispositivo com a mesma eficiência. Por exemplo, ao receber um comando de voz, um software assistente virtual pode realizar o upload do arquivo de áudio para um serviço especializado, e receber como resposta a transcrição do comando realizado. Com isso, é possível reduzir o tempo de

processamento, a depender do poder computacional do servidor, e também o consumo de bateria em dispositivos móveis, uma vez que a tarefa de processamento é delegada para um servidor que não possui a mesma limitação de consumo de energia.

- **Centralização do processamento:** Uma vez que a aplicação Cliente muitas vezes é executada em dispositivos em controle de terceiros, como um computador de um usuário final, não é possível confiar que o software não foi modificado de modo indesejável. Em casos em que é imperativo que operações sejam feitas de maneiras específicas, esse processamento deve ser realizado em um Servidor sobre o qual há o controle e a segurança de que o software correto está sendo executado. Por exemplo, durante uma transferência bancária em um portal de *Internet Banking*, diversas checagens são realizadas antes da efetuação da operação. Entre elas, o sistema do banco deve verificar se a conta de onde sairão os fundos possui saldo suficiente. Estas checagens devem ser realizadas em um servidor de propriedade da instituição bancária. Caso contrário, um agente mal-intencionado poderia modificar o software Cliente de modo a aprovar transações mesmo no caso de não haver saldo disponível.

Por outro lado, esse padrão arquitetural introduz problemas em potencial, como os listados a seguir:

- **Dependência da rede:** É esperado que a comunicação entre Cliente e Servidor apresentem latências introduzidas pela rede, algo que não ocorreria caso todo o processamento fosse realizado no dispositivo local. Ainda, a aplicação Cliente pode tornar-se completamente inoperante caso falhas na rede impeçam a comunicação com o Servidor.
- **Falhas no servidor comprometem todo o sistema:** Como citado anteriormente, caso o servidor esteja inacessível, as aplicações Clientes podem tornar-se completamente inoperantes. Caso o problema resida no servidor em si, é possível que muitos Clientes sejam impactados devido a este único ponto de falha.

Não existe uma regra clara para definir quais funcionalidades serão implementadas localmente (no Cliente) e quais serão implementadas remotamente (no Servidor). Em geral, uma boa heurística é implementar todas as validações e regras de negócio no Servidor para garantir a segurança e a consistência dos dados. Idealmente, validações também devem ser

implementadas na aplicação Cliente de modo a melhorar a experiência de usuário, uma vez que as requisições para o servidor introduzem latências, que podem ser evitadas ao validar os dados localmente. No entanto, as validações realizadas pelo Cliente são úteis somente para oferecer ao usuário um fluxo de trabalho mais fluido, e nunca para assegurar a validade dos dados recebidos pelo Servidor; todos os dados enviados pelo Cliente só devem ser considerados confiáveis após realizadas as validações pelo Servidor.

Aplicações Clientes podem ser classificadas num espectro entre *thin clients* e *thick clients* (ou *fat clients*). *Thin clients* são Clientes dos quais a grande a maioria das funcionalidades é dependente do Servidor. Normalmente por esse motivo, *thin clients* utilizam poucos recursos computacionais dos dispositivos que os executam, delegando a maior parte do trabalho para o Servidor. Em contrapartida, *thin clients* estão mais suscetíveis a pararem de funcionar por completo caso ocorram problemas de comunicação ou com o próprio servidor. Por sua vez, *thick clients* possuem funcionalidades projetadas para serem executadas localmente, com menor dependência do Servidor remoto (ainda que ela exista). Estes Clientes são capazes de funcionar ao menos parcialmente mesmo que ocorram falhas no Servidor. *Thick clients* costumam consumir mais recursos do dispositivo que os executam, devido à sua maior complexidade.

2.2 MICROSSERVIÇOS

A arquitetura de Microsserviços se caracteriza por dividir um sistema complexo em múltiplas partes menores (os chamados Microsserviços), onde cada uma delas trabalha independentemente das outras, com seus próprios meios de processamento, e, em geral, utilizando-se do HTTP para comunicação com outros Microsserviços (FOWLER e LEWIS, 2014).

Cada Microsserviço existe como uma aplicação completamente independente das outras no sentido da infraestrutura, ou seja, não compartilham recursos computacionais, bancos de dados, configurações, etc. No entanto, um Microsserviço pode depender de outros componentes para realizar o seu trabalho (evidentemente, as dependências não podem ser cíclicas). A comunicação entre os Microsserviços é o aspecto central deste modelo, e geralmente é o único ponto onde há uma severa restrição sobre como os componentes devem se comportar: é necessário que exista uma API acordada previamente, especificando os formatos de requisições e de respostas, possíveis fluxos de erros e suas implicações. Somente assim é possível que os diversos Microsserviços funcionem harmonicamente.

Os principais benefícios deste padrão arquitetural são:

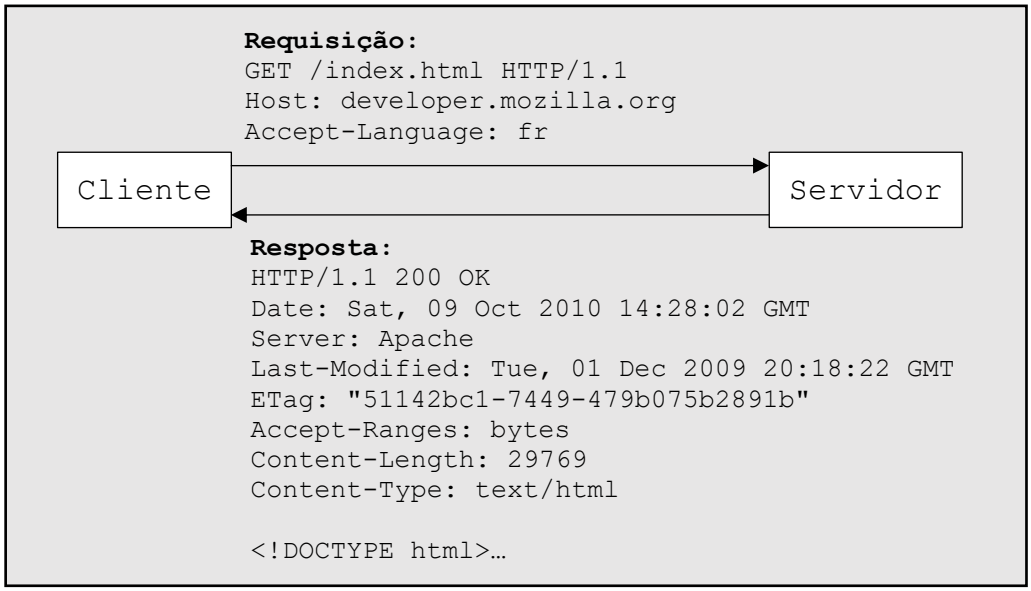
- **Escalabilidade eficiente:** Esta arquitetura permite um controle granular sobre os recursos computacionais à disposição dos diferentes serviços. Em ambientes em que as cargas de trabalho são heterogêneas ou imprevisíveis, a possibilidade de ajustar os recursos apenas dos serviços sobrecarregados é atrativa sob diversos pontos de vista, entre eles: o financeiro (com a redução da capacidade ociosa), o operacional (com o uso mais eficiente dos recursos disponíveis) e o ambiental (com a redução do uso de energia elétrica).
- **Menor complexidade:** Sendo cada Microsserviço um componente independente responsável por realizar apenas um fragmento da lógica de negócio, os softwares resultantes tendem a ser consideravelmente menos complexos do que seria um monolito desenvolvido para desempenhar o mesmo trabalho. Em contrapartida, a comunicação entre serviços diferentes adiciona uma nova camada de complexidade, que deve ser controlada por meio de um conjunto de APIs bem desenhadas. Na prática, esta é uma tarefa desafiadora para os arquitetos de sistemas.

2.3 HTTP

HTTP (*HyperText Transfer Protocol*) é um protocolo que permite a transferência de dados pela rede por meio de uma arquitetura Cliente-Servidor. A comunicação ocorre por meio de mensagens entre as duas partes; as mensagens originadas pelo Cliente são chamadas de Requisições (Requests), e as mensagens originadas do Servidor são chamadas de Respostas (Responses).

O HTTP é um protocolo *stateless*, o que significa que as mensagens HTTP não compartilham estados entre si. Isso significa que cada Requisição é tratada de maneira independente e não tem conhecimento das Requisições anteriores ou posteriores. Isso pode ser visto como uma vantagem, pois permite que o Servidor responda a Requisições de maneira mais eficiente e escalável, sem a complicação adicional de manter o estado de conexão entre várias Requisições.

Figura 2 — O fluxo de uma transação HTTP



Fonte: MDN Web Docs (adaptado pelo autor).

A Figura 2 ilustra uma transação HTTP. O Cliente inicia a transação ao enviar uma Requisição para o servidor *developer.mozilla.org*, solicitando que o envie o arquivo *index.html*. Por sua vez, o Servidor responde à requisição enviando o recurso solicitado, com a inclusão de metadados como o tipo do recurso, tamanho em bytes, e data de modificação.

Por ser um protocolo baseado estritamente em texto, outros tipos de recursos, como áudio e vídeo, devem ser codificados em um formato textual antes de serem enviados pelo remetente, e posteriormente decodificados pelo destinatário.

Um Cliente pode realizar várias Requisições simultaneamente, para um ou mais Servidores diferentes. Idealmente, o Cliente deve estar preparado caso ocorram problemas e o Servidor não responda à sua Requisição. Também é possível que um Cliente abandone uma Requisição antes que ela seja respondida pelo Servidor, por qualquer motivo.

A versão original do HTTP transmite dados em texto simples pela rede. Para introduzir uma camada de segurança criptográfica ao protocolo, utiliza-se o TLS (*Transport Layer Security*). A versão do protocolo que emprega a criptografia das mensagens chama-se HTTPS (*HyperText Transfer Protocol Secure*).

2.3.1 APIs REST

É possível utilizar o HTTP para disponibilizar recursos que não são arquivos estáticos, ou que até mesmo necessitam de algum processamento antes de serem enviados na Resposta do Servidor. Uma API HTTP pode ser caracterizada como um conjunto de operações

predefinidas, disponibilizadas por um Servidor, que podem ser executadas por meio de uma Requisição do Cliente. APIs REST são uma forma comum de projetar tais operações (embora não seja a única forma).

Figura 3 — Exemplo de uma Requisição e uma Resposta envolvendo uma API REST

```
Requisição:
GET /api/planets/3 HTTP/1.1
Host: swapi.dev

Resposta:
HTTP/1.1 200 OK
Server: nginx/1.16.1
Date: Sat, 14 Jan 2023 21:40:35 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
ETag: "ccbca9ad5dbcc6c73413df0765660c26"
Allow: GET, HEAD, OPTIONS
Strict-Transport-Security: max-age=15768000

{"name":"Yavin IV","rotation_period":"24","orbital_period":"48
18","diameter":"10200","climate":"temperate, tropical","gravit
y":"1 standard","terrain":"jungle, rainforests","surface_water
":"8","population":"1000","residents":[],"films":["https://swa
pi.dev/api/films/1/"],"created":"2014-12-10T11:37:19.144000Z",
"edited":"2014-12-20T20:58:18.421000Z","url":"https://swapi.de
v/api/planets/3/"}
```

Fonte: swapi.dev (adaptado).

A Figura 3 ilustra o funcionamento de uma operação em uma API REST. No cenário demonstrado, para obter informações relacionadas a um planeta, o Cliente deve requisitar o recurso em *swapi.dev/api/planets/{id}*, onde *{id}* representa o número identificador do planeta desejado. No exemplo, é solicitado que o Servidor responda com as informações do planeta com identificador número 3. A Resposta do Servidor é codificada em formato JSON (*JavaScript Object Notation*), uma notação comum para o transporte de dados estruturados. No passado, outra notação comum era o XML (*Extensible Markup Language*).

O protocolo HTTP disponibiliza diversos Métodos de Requisição (ou “verbos”) que são utilizados para descrever as operações disponíveis em uma API REST. Na Figura 3, o Método utilizado é GET (obter, em inglês), que realiza a leitura de um recurso. Os Métodos HTTP mais comuns em APIs REST são listados resumidamente na Tabela 1.

Tabela 1 — Métodos de Requisições HTTP mais comuns

Método	Tradução	Operação
GET	Obter	Solicitar um recurso existente.
POST	Publicar	Criar um novo recurso a partir dos dados contidos na Requisição.
PUT	Aplicar	Substituir um recurso existente pelos dados contidos na Requisição.
PATCH	Remendar	Alterar um recurso existente, mesclando-o aos dados contidos na Requisição.
DELETE	Apagar	Remover um recurso existente.

Fonte: Elaborado pelo autor.

2.4 PROGRAMAÇÃO ASSÍNCRONA

A programação assíncrona é caracterizada pela execução de múltiplas rotinas “simultaneamente”. Na realidade, o que ocorre é uma rápida troca de contexto entre as múltiplas rotinas, que é abstraída com o conceito de paralelismo em linguagens de alto nível.

Durante o consumo de um serviço disponibilizado por um Servidor é comum que o Cliente permaneça ocioso enquanto a resposta de sua requisição não é disponibilizada, em outras palavras, o Cliente aguarda o Servidor terminar o processamento da sua requisição. A programação assíncrona se faz especialmente útil nestes casos, pois torna possível que outras rotinas sejam processadas no Cliente durante o período de espera.

Códigos que fazem uso dessas técnicas são frequentemente mais difíceis de escrever, entender e manter. É comum que partes de código relacionadas a uma mesma rotina fiquem separadas no código fonte, apenas por uma particularidade da natureza assíncrona (BIERMAN, RUSSO, *et al.*, 2012).

Uma solução para o problema da inteligibilidade de códigos assíncronos foi o conceito de “Promises”, implementado por linguagens de alto-nível, como C#, JavaScript, e Python. Uma Promise representa uma sub-rotina que é executada paralelamente ao fluxo principal do programa. O programador pode utilizar uma palavra reservada “`await`” para aguardar o fim da sub-rotina e obter seu resultado. Desta forma, é possível escrever códigos assíncronos de maneira mais parecida com a alternativa síncrona (BELSON, HOLDSWORTH, *et al.*, 2019).

2.4.1 Callbacks, Promises e Completable Futures

De modo a destacar as diferenças entre as diferentes abordagens para programação assíncrona, as seções a seguir exemplificarão a implementação de um programa que deve realizar as tarefas descritas na Tabela 2.

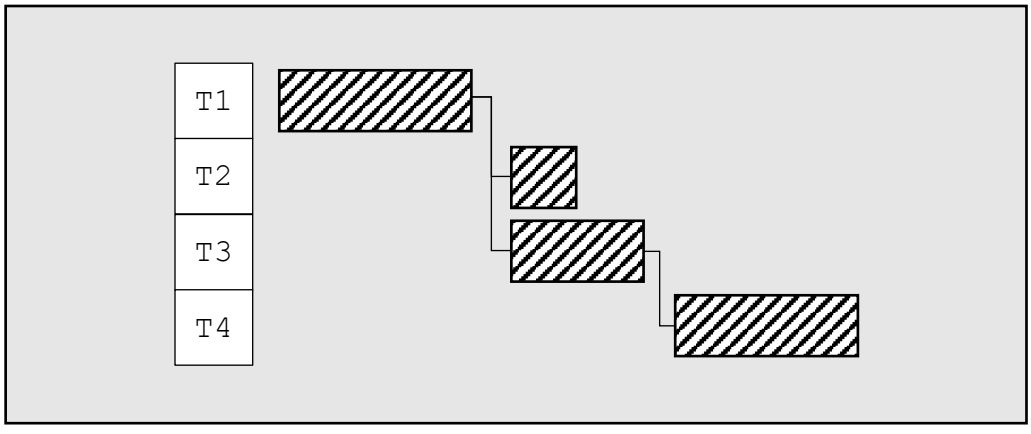
Tabela 2 — Descrição das tarefas

Tarefa	Descrição	Pré-requisito
T1	Obter uma lista de clientes de um banco de dados.	–
T2	Exibir na tela o número total de clientes obtidos.	Tarefa 1
T3	Calcular o total devido por cada cliente.	Tarefa 1
T4	Imprimir um relatório de valores a receber de cada cliente.	Tarefa 3

Fonte: Elaborado pelo autor.

A Figura 4 apresenta uma linha do tempo ilustrando como a relação entre pré-requisitos tem impacto na ordem de execução das tarefas. Cada barra horizontal listrada representa a execução de uma tarefa, identificada pelos títulos na margem esquerda. O eixo horizontal representa a passagem do tempo, ou seja, barras listradas empilhadas verticalmente indicam tarefas sendo executadas em paralelo. As linhas conectando duas barras diferentes indicam que uma é pré-requisito da outra, ou seja, a tarefa à direita da linha só pode ser iniciada após o término da tarefa à esquerda. Todas as outras ilustrações de linha do tempo utilizadas neste trabalho seguem este mesmo padrão.

Figura 4 — Linha do tempo das tarefas descritas na Tabela 2



Fonte: Elaborado pelo autor.

2.4.1.1 Callbacks

O uso de Callbacks foi uma das primeiras técnicas de programação assíncronas largamente utilizadas pelos desenvolvedores de software. Um Callback é uma função passada como parâmetro para outra função (geralmente assíncrona). A Figura 5 ilustra uma implementação onde `imprimirClientes` é passado como Callback para a função `obterClientes`.

Figura 5 — Encadeando duas tarefas utilizando Callbacks (primeira implementação)

```
const obterClientes = function (callback) {
  const listaDeClientes = ... // Omitido por simplicidade
  callback(listaDeClientes);
};

const imprimirClientes = function (listaDeClientes) {
  // Omitido por simplicidade
};

obterClientes(imprimirClientes);
```

Fonte: Elaborado pelo autor.

É possível adaptar o código da Figura 5 para que a função `obterClientes` suporte zero ou mais Callbacks, como ilustrado na Figura 6.

Figura 6 — Encadeando duas tarefas utilizando Callbacks (segunda implementação)

```
const obterClientes = function (callbacks) {
  const listaDeClientes = ... // Omitido por simplicidade
  for (callback of callbacks) {
    callback(listaDeClientes);
  }
};

const imprimirClientes = function (listaDeClientes) {
  // Omitido por simplicidade
};

obterClientes(imprimirClientes);
```

Fonte: Elaborado pelo autor.

A Figura 7 ilustra uma implementação das tarefas descritas na Tabela 2 utilizando Callbacks.

Figura 7 — Encadeando quatro tarefas utilizando Callbacks

```
const obterClientes = function (callbacks) {
  const listaDeClientes = ... // Omitido por simplicidade
  for (callback of callbacks) {
    callback(listaDeClientes);
  }
};
const exibirNumeroTotal =
  function (listaDeClientes, callbacks) {
    // Omitido por simplicidade
    for (callback of callbacks) {
      callback(listaDeClientes);
    }
  };
const calcularTotalDevidoPorCliente =
  function (listaDeClientes, callbacks) {
    const totais = ... // Omitido por simplicidade
    for (callback of callbacks) {
      callback(totais);
    }
  };
const imprimirRelatorio = function (totais, callbacks) {
    // Omitido por simplicidade
    for (callback of callbacks) {
      callback(totais);
    }
  };
obterClientes([
  exibirNumeroTotal,
  (listaDeClientes) => {
    calcularTotalDevidoPorCliente(
      listaDeClientes,
      imprimirRelatorio);
  },
]);
```

Fonte: Elaborado pelo autor.

Fica evidente que uma aplicação assíncrona baseada apenas em Callbacks corre o risco de tornar-se demasiadamente complexa em pouco tempo, uma vez que a lógica deve ser transportada via parâmetros de funções, o que torna difícil o entendimento do fluxo do programa.

Problemas ocasionados pelo uso não otimizado de Callbacks são tão comuns que o fenômeno é chamado de "Callback Hell" na comunidade de desenvolvedores. A Figura 8 ilustra um cenário onde múltiplos callbacks são criados de forma aninhada, criando um aspecto de "pirâmide" na indentação, o que piora a legibilidade do código.

Figura 8 — Exemplo de um Callback Hell

```
fs.readdir(source, function (err, files) {
 if (err) {
  console.log('Error finding files: ' + err)
 } else {
  files.forEach(function (filename, fileIndex) {
   console.log(filename)
   gm(source + filename).size(function (err, values) {
    if (err) {
     console.log('Error identifying file size: ' + err)
    } else {
     console.log(filename + ' : ' + values)
     aspect = (values.width / values.height)
     widths.forEach(function (width, widthIndex) {
      height = Math.round(width / aspect)
      console.log('resizing ' + filename + 'to ' + height
                  + 'x' + height)
      this.resize(width, height).write(dest + 'w' + width
                  + '_' + filename, function(err) {
       if (err) console.log('Error writing file: ' + err)
      })
     }.bind(this))
    }
   })
  })
 }
})
```

Fonte: callbackhell.com.

2.4.1.2 Promises

Introduzidas no JavaScript em 2015, as Promises abordam o problema da legibilidade que surge durante o uso de Callbacks. Uma Promise pode ser compreendida como um objeto que representa a execução de uma função, no qual é possível inserir Callbacks mais elegantemente, e a qualquer momento, mesmo após o fim da execução da função.

Figura 9 — Encadeando quatro tarefas utilizando Promises

```
const obterClientes = function () {
  const listaDeClientes = ... // Omitido por simplicidade
  return listaDeClientes; // listaDeClientes é uma Promise.
};
const exibirNumeroTotal = function (listaDeClientes) {
    // Omitido por simplicidade
};
const calcularTotalDevidoPorCliente =
  function (listaDeClientes) {
    const totais = ... // Omitido por simplicidade
    return totais; // totais é uma Promise.
};
const imprimirRelatorio = function (totais) {
    // Omitido por simplicidade
};

const promiseClientes = obterClientes();

promiseClientes.then(exibirNumeroTotal);
promiseClientes.then(calcularTotalDevidoPorCliente)
               .then(imprimirRelatorio);
```

Fonte: Elaborado pelo autor.

Utilizando o método `then` é possível vincular uma função a ser executada uma vez que o resultado da Promise estiver disponível. O retorno de uma chamada para este método é uma nova Promise, que representa a execução das duas funções, uma após a outra. Isso permite que várias chamadas ao método `then` sejam encadeadas, criando uma corrente de operações assíncronas que devem ser executadas uma após a outra.

2.4.1.2.1 Async e Await

Em JavaScript, é possível empregar Promises de forma menos verborrágica, utilizando as palavras reservadas `async` e `await`, como ilustrado na Figura 10.

Figura 10 — Encadeando quatro tarefas utilizando Promises, async e await

```
const obterClientes = function () {
  const listaDeClientes = ... // Omitido por simplicidade
  return listaDeClientes; // listaDeClientes é uma Promise.
};
const exibirNumeroTotal = function (listaDeClientes) {
    // Omitido por simplicidade
};
const calcularTotalDevidoPorCliente =
  function (listaDeClientes) {
    const totais = ... // Omitido por simplicidade
    return totais; // totais é uma Promise.
};
const imprimirRelatorio = function (totais) {
    // Omitido por simplicidade
};

const listaDeClientes = await obterClientes();
exibirNumeroTotal(listaDeClientes);
const totais =
  await calcularTotalDevidoPorCliente(listaDeClientes);
imprimirRelatorio(totais);
```

Fonte: Elaborado pelo autor.

A palavra reservada `await` suspende a execução do código até que a tarefa correspondente seja completada. O uso do `await` pode tornar o código fonte mais fácil de ser entendido, uma vez que existe maior semelhança ao seu equivalente síncrono.

2.4.1.3 Completable Futures

O Java trouxe em sua versão 1.8 a classe *CompletableFuture*, introduzida como parte de uma melhoria extensa na API de concorrência *java.util.concurrent*. O Completable Future

reduz a quantidade de código necessária para realizar operações assíncronas, sendo possível, inclusive, notar padrões similares de uso em comparação com as Promises do JavaScript.

Completable Futures podem ser classificados como Monads, um conceito emprestado da programação funcional que consiste em encapsular um valor em um tipo com comportamentos adicionais (O'SULLIVAN, STEWART e GOERZEN, 2009). Neste caso, objetos comuns da linguagem Java são encapsulados de modo a poderem ser trabalhados assincronamente.

Suponha uma classe `ClienteWeb`, que possui o método `baixarVideo`, cuja função é realizar o download de um vídeo disponível em um servidor remoto. Essa função é síncrona, ou seja, a execução do programa é suspensa enquanto o download não for completado. Para tornar assíncrona esta tarefa, instancia-se um `CompletableFuture` apontando a função síncrona que se deve executar assincronamente. A Figura 11 exemplifica o uso da função `suppplyAsync`, que retorna um `CompletableFuture` encapsulando o método `baixarVideo`, e posteriormente, o valor retornado por ele.

Figura 11 — Instanciando um CompletableFuture

```
CompletableFuture<Byte[]> futureVideo =
  CompletableFuture.supplyAsync(ClienteWeb::baixarVideo);
```

Fonte: Elaborado pelo autor.

Uma vez instanciado, `futureVideo` imediatamente iniciará a execução da função `baixarVideo` em "segundo plano", e a execução do programa seguirá para as instruções seguintes no código-fonte.

Para obter os dados retornados pela função `baixarVideo`, o `CompletableFuture` dispõe dois métodos: `join` e `get`. Os dois métodos retornam o mesmo valor, porém a primeira opção permite a omissão do bloco de tratamento de erro. Ao invocar um dos métodos, o valor encapsulado pelo `CompletableFuture` é retornado caso ele já esteja disponível; caso contrário, a execução do código é suspensa até que o processamento da função passada anteriormente no método `supplyAsync` (Figura 11) termine.

Figura 12 — Obtendo o resultado de um CompletableFuture

```
// Usando join
Byte[] videoBaixado = futureVideo.join();
```

```
// Usando get
try {
   Byte[] videoBaixado = futureVideo.get();
} catch (InterruptedException | ExecutionException e) {
  // Tratamento de erro
}
```

Fonte: Elaborado pelo autor.

Também é possível encadear tarefas facilmente: como ilustrado na Figura 13, o método `thenAccept` retorna um novo `CompletableFuture` que executa uma segunda função logo após o término da primeira (no exemplo, a segunda função grava o vídeo em um CD). Tanto o `CompletableFuture` resultante quanto o original podem ser utilizados para criar cadeias complexas de processamento.

Figura 13 — Encadeando operações usando CompletableFuture

```
CompletableFuture<Void> futureGravacao =
  CompletableFuture.supplyAsync(ClienteWeb::baixarVideo)
    .thenAccept(LeitorDisco::gravarDisco);
```

Fonte: Elaborado pelo autor.

A Figura 14 ilustra uma implementação das tarefas descritas na Tabela 2 utilizando Completable Futures.

Figura 14 — Encadeando quatro tarefas utilizando Completable Futures

```
public static class Cliente {
  // Omitido por simplicidade
}
public List<Cliente> obterClientes() {
  return ...; // Omitido por simplicidade
}
public void exibirNumeroTotal(List<Cliente> listaDeClientes) {
  // Omitido por simplicidade
}
public Map<Cliente, BigDecimal> calcularTotalDevidoPorCliente
(List<Cliente> listaDeClientes) {
  return ...; // Omitido por simplicidade
}
public void imprimirRelatorio(Map<Cliente, BigDecimal> totais)
{
  // Omitido por simplicidade
}
public void executar() {
  CompletableFuture<List<Cliente>> futureClientes =
    CompletableFuture.supplyAsync(this::obterClientes);

  futureClientes.thenAccept(this::exibirNumeroTotal);

  CompletableFuture<Map<Cliente, BigDecimal>> futureTotais =
    futureClientes
    .thenApply(this::calcularTotalDevidoPorCliente);

  Map<Cliente, BigDecimal> totais = futureTotais.join();

  imprimirRelatorio(totais);
}
```

Fonte: Elaborado pelo autor.

2.5 PROGRAMAÇÃO IMPERATIVA E DECLARATIVA

Comumente utilizada, a programação Imperativa consiste em efetuar chamadas de funções ou métodos explicitamente para realizar operações. Em contrapartida, a programação Declarativa introduz abstrações em que o programador é capaz de "declarar" o resultado desejado, e o programa efetuará as operações necessárias para alcançar o objetivo.

A Figura 15 e a Figura 16 ilustram a realização da mesma tarefa, utilizando a programação imperativa e declarativa, respectivamente.

Figura 15 — Filtragem de uma lista utilizando a abordagem Imperativa

```
List<Employee> f(List<Long> ids, List<Employee> employees) {
  List<Employee> l = new ArrayList<>();
  for (Employee employee : employees) {
    if (ids.contains(employee.getId())) {
      l.add(employee);
    }
  }
  return l;
}
```

Fonte: MEHLHORN e HANENBERG, 2022, p. 1160 (adaptado para corrigir um erro de digitação no material original).

Figura 16 — Filtragem de uma lista utilizando a abordagem Declarativa

```
List<Employee> f(List<Long> ids, List<Employee> employees) {
  return employees.stream()
    .filter(employee -> ids.contains(employee.getId()))
    .collect(toList());
}
```

Fonte: MEHLHORN e HANENBERG, 2022, p. 1160.

Segundo o estudo conduzido por Mehlhorn e Hanenberg (2022), funções na forma Declarativa que realizam operações em Coleções em Java são mais facilmente compreendidas por programadores, quando comparadas às suas equivalentes Imperativas. Ainda, observou-se que o efeito se torna mais perceptível à medida que as funções se tornam mais complexas.

2.6 LINGUAGENS ESPECÍFICAS DE DOMÍNIO

Linguagens Específicas de Domínio, do inglês Domain-Specific Languages (DSLs) podem ser explicadas como:

> Uma DSL é uma linguagem de programação voltada para um problema específico; as outras linguagens de programação existentes são de uso mais geral. Ela contém a sintaxe e a semântica que modelam os conceitos no mesmo nível de abstração que o domínio do problema oferece. (GHOSH, 2011, p. 10) tradução nossa.

Um benefício trazido pelo uso de uma DSL é a sua reduzida complexidade, em comparação com outras linguagens de programação de uso geral (GHOSH, 2011, p. 11).

DSLs podem ser classificadas como *internas* (ou *embutidas*) ou *externas*. As linguagens de domínio internas são construídas sobre uma linguagem de programação pré-existente de modo a adicionar a semântica específica do domínio. Por sua vez, linguagens externas são construídas do zero (GHOSH, 2011, p. 18).

Figura 17 — Exemplo de uso de uma DSL interna pela biblioteca Spring Security

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
throws Exception {
  return http.authorizeRequests()
    .antMatchers("/auth") // Requests que iniciam em /auth ...
    .permitAll() // ... são todos permitidos
    .anyRequest() // Todos os outros Requests ...
    .authenticated() // ... devem estar autenticados
    .and()
    .httpBasic() // O tipo de autenticação é Basic
    .build();
}
```

Fonte: Elaborado pelo autor.

A Figura 17 demonstra como uma DSL interna é utilizada pela biblioteca Spring Security para definir as configurações de segurança de uma aplicação.

2.6.1 Fluent Interfaces

Fowler (2005) chama de *Fluent Interfaces* as linguagens específicas de domínio internas que são construídas de modo a facilitar a legibilidade de forma "fluente".

A Figura 18 demonstra a diferença entre uma interface comum e uma Interface Fluente. A abordagem comum necessita que os objetos sejam construídos manualmente (`new OrderLine`), e suas relações estabelecidas separadamente (`o1.addLine`). Já a Interface Fluente permite que os objetos sejam instanciados e vinculados a partir de uma única invocação de método (`with`) para cada item. A semântica adicionada pelas Interfaces Fluentes e a redução da quantidade de instruções necessárias no código as tornam mais legíveis que suas alternativas.

Figura 18 — Comparação entre uma interface comum e uma Interface Fluente em Java

```
private void makeNormal(Customer customer) {
  Order o1 = new Order();
  customer.addOrder(o1);
  OrderLine line1 = new OrderLine(6, Product.find("TAL"));
  o1.addLine(line1);
  OrderLine line2 = new OrderLine(5, Product.find("HPK"));
  o1.addLine(line2);
  OrderLine line3 = new OrderLine(3, Product.find("LGV"));
  o1.addLine(line3);
   line2.setSkippable(true);
   o1.setRush(true);
}
private void makeFluent(Customer customer) {
  customer.newOrder()
          .with(6, "TAL")
          .with(5, "HPK").skippable()
          .with(3, "LGV")
          .priorityRush();
}
```

Fonte: FOWLER, 2005.

3 DESENVOLVIMENTO DA API

Esse capítulo apresenta o processo de desenvolvimento da FluentAsync e os blocos básicos de execução de tarefas nos quais ela se baseia.

3.1 BLOCOS BÁSICOS DE EXECUÇÃO DE TAREFAS

Esta seção apresenta os quatro blocos básicos nos quais a API proposta se baseia: (i) Encadeamento; (ii) Paralelização; (iii) Ramificação; (iv) Unificação. A partir destes é possível implementar outros arranjos complexos.

3.1.1 Encadeamento

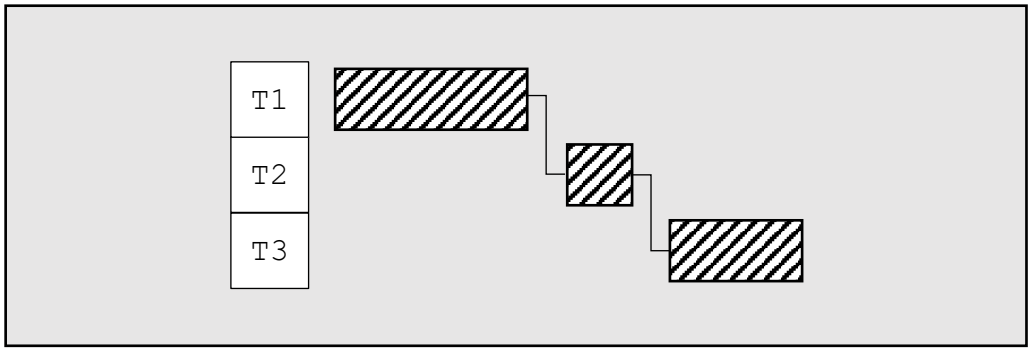
Quando há relação de dependência entre duas ou mais tarefas elas devem ser executadas em sequência, caracterizando o Encadeamento. A Tabela 3 e a Figura 19 ilustram o encadeamento de três tarefas.

Tabela 3 — Relação de pré-requisitos entre três tarefas em um Encadeamento

Tarefa	Pré-requisito
T1	–
T2	Tarefa 1
T3	Tarefa 2

Fonte: Elaborado pelo autor.

Figura 19 — Linha do tempo do Encadeamento de 3 tarefas



Fonte: Elaborado pelo autor.

A Figura 20 exemplifica a implementação em JavaScript do encadeamento de três tarefas utilizando Promises.

Figura 20 — Encadeando três tarefas utilizando Promises

```
const resultadoT3 =
  t1().then((resultadoT1) => t2(resultadoT1))
      .then((resultadoT2) => t3(resultadoT2));
```

Fonte: Elaborado pelo autor.

3.1.2 Paralelização

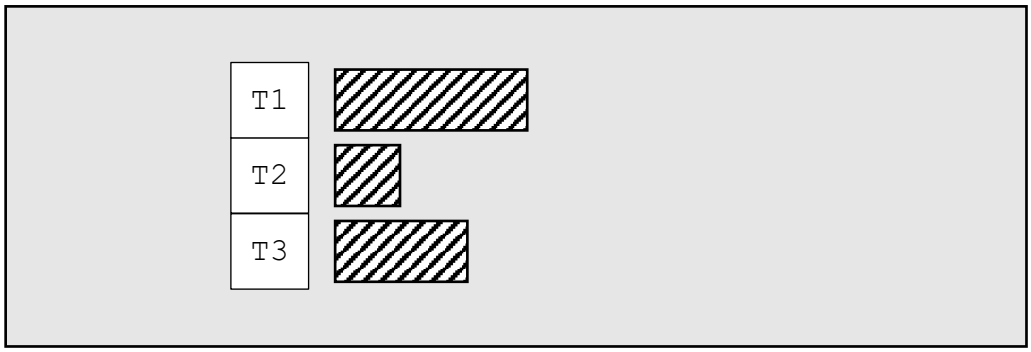
Quando duas ou mais tarefas não são dependentes entre si, elas podem ser executadas de forma concorrente, caracterizando a Paralelização. A Tabela 4 e a Figura 21 ilustram a paralelização de três tarefas.

Tabela 4 — Relação de pré-requisitos entre três tarefas em uma Paralelização

Tarefa	Pré-requisito
T1	_
T2	_
T3	_

Fonte: Elaborado pelo autor.

Figura 21 — Linha do tempo da Paralelização de 3 tarefas



Fonte: Elaborado pelo autor.

A Figura 22 exemplifica a implementação em JavaScript da paralelização de três tarefas utilizando Promises. Uma característica importante a se destacar é a independência das tarefas, uma vez que os resultados de cada uma não são pré-requisitos para a execução de outras.

Figura 22 — Paralelizando três tarefas utilizando Promises

```
const resultadoT1 = t1();
const resultadoT2 = t2();
const resultadoT3 = t3();
```

Fonte: Elaborado pelo autor.

3.1.3 Ramificação

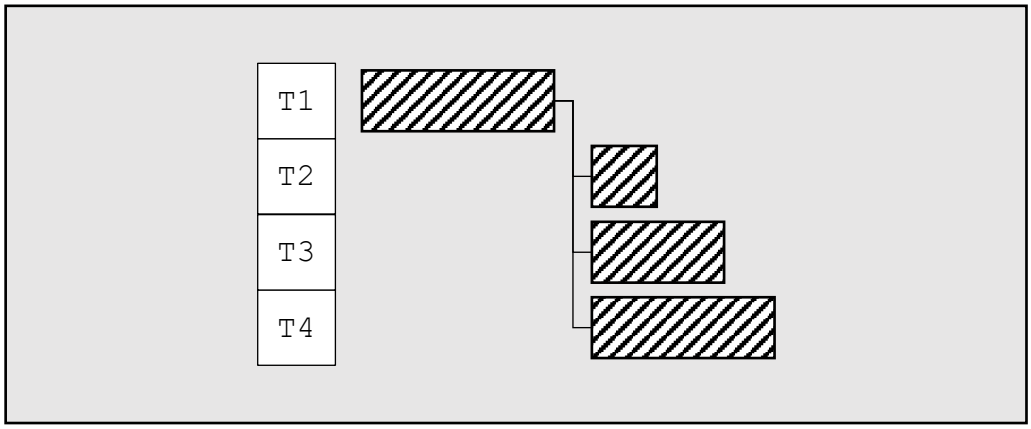
A Ramificação se caracteriza por um Encadeamento no qual duas ou mais tarefas na sequência podem ser executadas em paralelo, levando a uma Paralelização. A Tabela 5 e a Figura 23 ilustram um exemplo de Ramificação.

Tabela 5 — Relação de pré-requisitos entre quatro tarefas em uma Ramificação

Tarefa	Pré-requisito
T1	–
T2	Tarefa 1
T3	Tarefa 1
T4	Tarefa 1

Fonte: Elaborado pelo autor.

Figura 23 — Linha do tempo de uma tarefa ramificando-se em três tarefas



Fonte: Elaborado pelo autor.

A Figura 24 exemplifica a implementação em JavaScript da Ramificação de uma tarefa em três utilizando Promises.

Figura 24 — Ramificando uma tarefa em três tarefas utilizando Promises

```
t1().then((resultadoT1) => {
    const resultadoT2 = t2();
    const resultadoT3 = t3();
    const resultadoT4 = t4();
    return Promise.all(
      [resultadoT2, resultadoT3, resultadoT4]
    );
  })
```

Fonte: Elaborado pelo autor.

3.1.4 Unificação

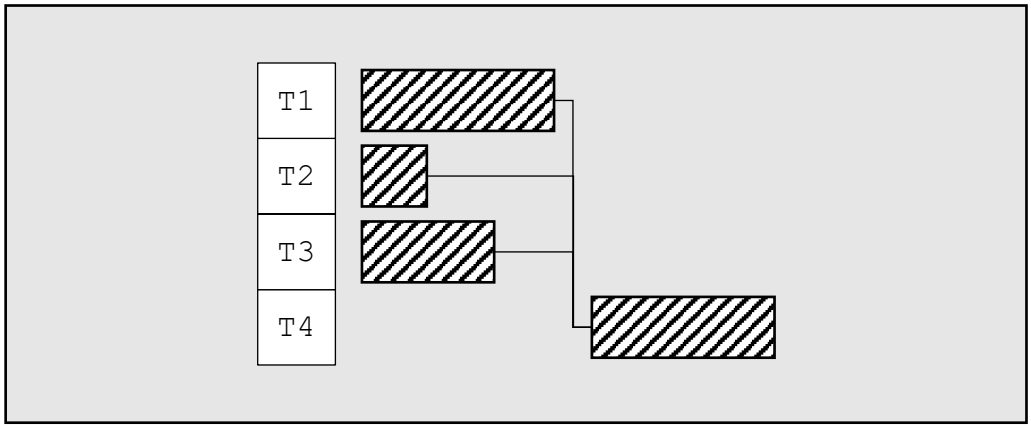
A Unificação se caracteriza pela Paralelização de duas ou mais tarefas, com o encadeamento de uma outra tarefa, levando a uma Unificação. A Tabela 6 e a Figura 25 ilustram um exemplo de Unificação.

Tabela 6 — Relação de pré-requisitos entre quatro tarefas em uma Unificação

Tarefa	Pré-requisito
T1	–
T2	–
T3	–
T4	Tarefa 1; Tarefa 2; Tarefa 3

Fonte: Elaborado pelo autor.

Figura 25 — Linha do tempo de três tarefas unificando-se em uma tarefa



Fonte: Elaborado pelo autor.

A Figura 26 exemplifica a implementação em JavaScript da Unificação de três tarefas em uma utilizando Promises.

Figura 26 — Unificando três tarefas em uma tarefa utilizando Promises

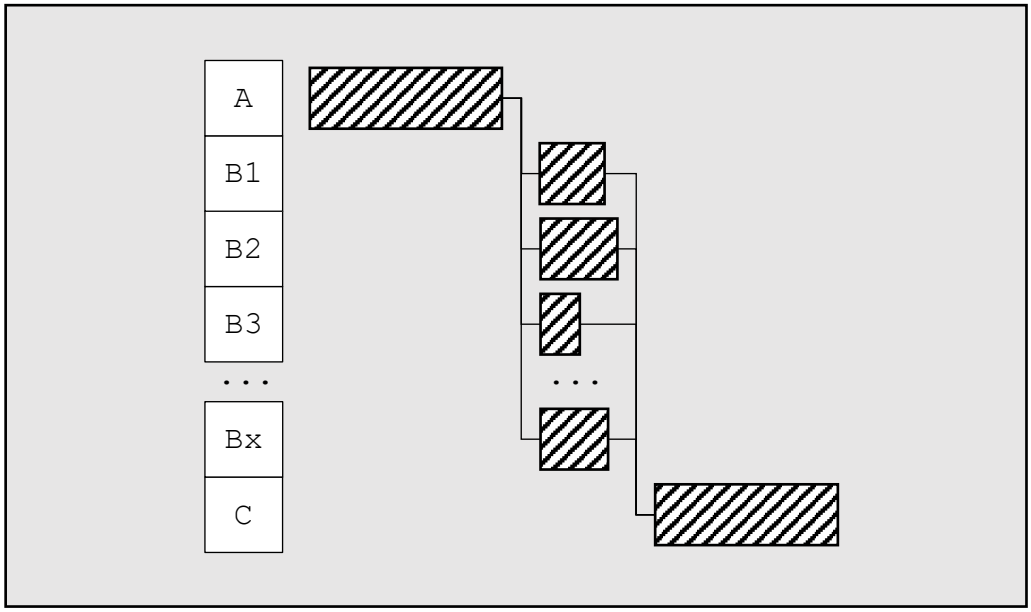
```
const resultadoT4 =
  Promise.all([t1(),  t2(),  t3()])
    .then(([resultadoT1, resultadoT2, resultadoT3]) => {
      return t4(resultadoT1, resultadoT2, resultadoT3)
    });
```

Fonte: Elaborado pelo autor.

3.2 ARRANJOS COMPLEXOS

A combinação dos blocos básicos definidos na seção 3.1 permite a execução de tarefas complexas, comuns em aplicações modernas. A Figura 27 ilustra o uso combinado da Ramificação e Unificação em um aplicativo de rastreamento de encomendas que deve: (a) obter uma lista de itens disponíveis para rastreio, (b) para cada item da lista, obter a última localização do pacote; em seguida, (c) ordenar os resultados em relação à data da última atualização.

Figura 27 — Linha do tempo das tarefas executadas pelo aplicativo de rastreamento



Fonte: Elaborado pelo autor.

Neste cenário ocorre a combinação de dois blocos básicos: após a obtenção da lista dos itens disponíveis no serviço de rastreio (Tarefa A), o programa utiliza a Ramificação para obter, paralelamente, informações sobre cada pacote (Tarefas B1, B2, B3, ..., Bx). Uma vez executadas, as instâncias da Tarefa B são unificadas para serem ordenadas pela data da última atualização (Tarefa C). A Figura 28 e a Figura 29 apresentam um exemplo de implementação do cenário anteriormente descrito em JavaScript e Java, respectivamente.

Figura 28 — Implementação do fluxo do aplicativo de rastreamento em JavaScript

```
obterListaDeItens()
  .then((itens) => {
    const movimentacoes = [];
    for(item of itens) {
      const movimentacao = obterUltimaMovimentacao(item);
      movimentacoes.push(movimentacao);
    }
    return Promise.all(movimentacoes);
  })
  .then(
    (movimentacoes) => ordenarMovimentacoes(movimentacoes)
  );
```

Fonte: Elaborado pelo autor.

Figura 29 — Implementação do fluxo do aplicativo de rastreamento em Java

```
ArrayList<ItemRastreado> itens = obterListaDeItens().join();

List<CompletableFuture<Movimentacao>> futuresMovimentacoes =
  new ArrayList<>();
for (ItemRastreado itemRastreado : itens) {
  CompletableFuture<Movimentacao> movimentacao =
    obterUltimaMovimentacao(itemRastreado);
  futuresMovimentacoes.add(movimentacao);
}

CompletableFuture.allOf(
  futuresMovimentacoes.toArray(CompletableFuture[]::new)
).join();

List<Movimentacao> movimentacoes =
  futuresMovimentacoes.stream()
    .map(CompletableFuture::join)
    .toList();

List<Movimentacao> movimentacoesOrdenadas =
  ordenarMovimentacoes(movimentacoes);
```

Fonte: Elaborado pelo autor.

Nas duas implementações, foi necessário incluir uma lógica com o intuito de ramificar, e unificar as múltiplas tarefas assíncronas (*Promises* em JavaScript e *CompletableFutures* em Java). Em ambas, foi utilizado um loop para iniciar as tarefas `obterUltimaMovimentacao`, e em seguida armazená-las em uma coleção. Após iniciadas todas as tarefas, foi utilizado o meio adequado de cada linguagem para aguardar o término de cada uma, e posteriormente obter o valor retornado por elas antes de iniciar a tarefa `ordenarMovimentacoes`.

Esses exemplos indicam que, à medida que a relação entre as tarefas se torna mais complexa, o código necessário para coordená-las se torna o principal obstáculo para o seu desenvolvimento e, posteriormente, o seu entendimento. A severidade deste problema varia de acordo com a linguagem, afetando menos as linguagens como o JavaScript, e mais as linguagens como o Java. Ainda assim, ambas as linguagens podem se beneficiar de um modelo de programação assíncrona declarativo capaz de abstrair a complexidade do controle de fluxo e de operações assíncronas.

4 APRESENTAÇÃO DA API

O FluentAsync permite implementar declarativamente as operações apresentadas na Seção 3.1. O modelo também disponibiliza outros métodos utilitários.

Os exemplos desta seção e a implementação de referência da API foram escritos em Java, mas o modelo proposto pode ser implementado em outras linguagens, como JavaScript, TypeScript e Python. O APÊNDICE D exemplifica uma implementação do FluentAsync em TypeScript.

Assim como o `CompletableFuture`, o modelo FluentAsync implementa um Monad chamado `TarefaAssincrona`. O ponto de entrada no Monad é o próprio construtor de `TarefaAssincrona` (Figura 30), que recebe uma função que será executada assincronamente e, uma vez terminada, terá seu retorno armazenado.

Figura 30 — Instanciando uma *TarefaAssincrona*

```
new TarefaAssincrona<>(() -> {
 var resultadoDemorado = ClasseDemorada.operacaoDemorada();
 return resultadoDemorado;
});
```

Fonte: Elaborado pelo autor.

4.1 OPERAÇÕES DISPONÍVEIS

É possível realizar uma ilimitada gama de operações com os métodos disponibilizados por uma instância de `TarefaAssincrona`.

4.1.1 Aguardar

Ao ser invocado, o método `TarefaAssincrona#aguardar` retorna o valor armazenado pela `TarefaAssincrona`, caso a atividade tenha sido completada. Caso a tarefa ainda esteja sendo executada, suspende a execução do código até o seu término.

Figura 31 — Utilizando o método *TarefaAssincrona#aguardar*

```
var tarefa = new TarefaAssincrona<>(() -> {
 try {
  Thread.sleep(1000);
 } catch (InterruptedException e) {}
 return "terminado";
});
System.out.println("Tarefa criada.");
var resultado = tarefa.aguardar();
System.out.println(resultado);
```

Fonte: Elaborado pelo autor.

Ao executar o código da Figura 31, o texto "*Tarefa criada.*" será imediatamente impresso no console. Após 1000 milissegundos, o texto "*terminado*" será impresso no console.

4.1.2 Transformar

O método `TarefaAssincrona#transformar` recebe como um parâmetro uma função transformadora, que será encadeada após a execução de uma tarefa. A função transformadora recebe como parâmetro o retorno da primeira tarefa, e o transforma em qualquer outro objeto.

Figura 32 — Utilizando o método *TarefaAssincrona#transformar*

```
var tarefa = new TarefaAssincrona<>(() -> {
 return "terminado";
}).transformar(String::toUpperCase);
var resultado = tarefa.aguardar();
System.out.println(resultado);
```

Fonte: Elaborado pelo autor.

A execução do código da Figura 32 termina com a impressão do texto "*TERMINADO*" no console.

4.1.3 Consumir

Similar ao método `transformar` descrito na seção 4.1.2, o método `TarefaAssincrona#consumir` também encadeia uma função (chamada "função consumidora") após a execução da primeira tarefa. A função consumidora, porém, não possui

retorno, e a instância de `TarefaAssincrona` retornada pela chamada do método `transformar` deixa de ter um valor.

Figura 33 — Utilizando o método *TarefaAssincrona#consumir*

```
var retorno = new TarefaAssincrona<>(() -> {
 return "terminado";
}).consumir((texto) -> {
 System.out.println(texto.toUpperCase());
}).aguardar();
System.out.println(retorno);
```

Fonte: Elaborado pelo autor.

Ao executar o código da Figura 32, o console exibirá o texto "*TERMINADO*", e, na linha seguinte, "*null*".

4.1.4 Ramificar

O método `TarefaAssincrona#ramificar` recebe como parâmetro uma função (chamada de "função ramificadora") cujo objetivo é receber o resultado desta `TarefaAssincrona` e retornar um objeto do tipo *List*, onde cada elemento da lista dará origem a uma nova tarefa assíncrona. O objeto retornado por este método chama-se `TarefaRamificada`.

Figura 34 — Utilizando o método *TarefaAssincrona#ramificar*

```
new TarefaAssincrona<>(() -> {
 return "Luke;Leia;Han";
}).ramificar(nomes -> {
 String[] arrNomes = nomes.split(";");
 return Arrays.asList(arrNomes);
});
```

Fonte: Elaborado pelo autor.

O código ilustrado na Figura 34 inicia-se com uma `TarefaAssincrona` com resultado "*Luke;Leia;Han*". A função ramificadora divide esta `String` em uma lista de três elementos: "*Luke*", "*Leia*", e "*Han*". O objeto retornado é do tipo `TarefaRamificada<String>`, onde são encapsuladas 3 tarefas assíncronas, uma para cada elemento da lista.

4.1.5 Filtrar

Invocar o método `filtrar` em uma `TarefaRamificada` permite o descarte de tarefas baseado em um predicado passado como parâmetro.

Figura 35 — Utilizando o método *TarefaRamificada#filtrar*

```
new TarefaAssincrona<>(() -> {
 return "Luke;Leia;Han";
}).ramificar(nomes -> {
 String[] arrNomes = nomes.split(";");
 return Arrays.asList(arrNomes);
})
  .filtrar(nome -> nome.startsWith("L"))
  .consumir(System.out::println)
  .aguardar();
```

Fonte: Elaborado pelo autor.

Como ilustrado no código da Figura 35, após a ramificação das três tarefas (“*Luke*”, “*Leia*”, e “*Han*”), a chamada do método `ramificar` retorna uma nova `TarefaRamificada<String>` contendo apenas as tarefas com valor que iniciam com a letra “L”. Após a execução, as linhas “*Leia*” e “*Luke*” serão impressas no console.

4.1.6 Unificar

O método `TarefaRamificada#unificar` retorna uma instância de `TarefaAssincrona` da qual o valor é gerado a partir de uma função unificadora.

Figura 36 — Utilizando o método *TarefaRamificada#unificar*

```
new TarefaAssincrona<>(() -> {
 return "Luke;Leia;Han";
}).ramificar(nomes -> {
 String[] arrNomes = nomes.split(";");
 return Arrays.asList(arrNomes);
})
  .filtrar(nome -> nome.startsWith("L"))
  .unificar(listaNomes -> String.join(" e ", listaNomes))
  .consumir(System.out::println)
  .aguardar();
```

Fonte: Elaborado pelo autor.

O código da Figura 36, após ramificar e filtrar, unifica as tarefas "*Luke*" e "*Leia*" em uma única *String*, resultando em "*Luke e Leia*", que é impressa no console.

4.2 EXEMPLOS DE UTILIZAÇÃO

Esta seção demonstra a utilização do modelo declarativo de programação assíncrona oferecido pela *FluentAsync* a partir de exemplos ilustrativos de cenários comuns no dia a dia do desenvolvimento de software. Os exemplos são constituídos de uma requisição, uma implementação em Java sem a utilização do *FluentAsync* (ou Java *Vanilla*), e outra implementação com o uso do *FluentAsync*.

4.2.1 A API utilizada

Todos os exemplos ilustram uma aplicação Cliente, implementando um código que realizará requisições para o Servidor disponível em *https://swapi.dev*. O serviço *swapi.dev* é disponibilizado gratuitamente na internet e é comumente encontrado em tutoriais de programação por ser uma API simples e fácil de utilizar. A API, desenvolvida nos padrões REST, disponibiliza informações sobre os filmes da saga *Star Wars* em formato JSON. Todos os *endpoints* estão disponíveis a partir da raiz *https://swapi.dev/api/*.

4.2.1.1 People

A partir do *endpoint* "`people`" é possível obter uma lista de personagens. Os atributos disponibilizados para cada personagem estão ilustrados na Figura 37.

Figura 37 — Resposta HTTP para a requisição *GET https://swapi.dev/api/people/1/*


```
{
  "birth_year": "19 BBY",
  "eye_color": "Blue",
  "films": ["https://swapi.dev/api/films/1/", ...],
  "gender": "Male",
  "hair_color": "Blond",
  "height": "172",
  "homeworld": "https://swapi.dev/api/planets/1/",
  "mass": "77",
  "name": "Luke Skywalker",
  "skin_color": "Fair",
  "created": "2014-12-09T13:50:51.644000Z",
  "edited": "2014-12-10T13:52:43.172000Z",
  "species": ["https://swapi.dev/api/species/1/"],
  "starships": ["https://swapi.dev/api/starships/12/", ...],
  "url": "https://swapi.dev/api/people/1/",
  "vehicles": ["https://swapi.dev/api/vehicles/14/", ...]
}
```


Fonte: swapi.dev. Adaptado pelo autor.

4.2.1.2 Films

O *endpoint* "`films`" disponibiliza informações relacionadas aos filmes da saga. Os atributos presentes na resposta das requisições estão ilustrados na Figura 38.

Figura 38 — Resposta HTTP para a requisição *GET https://swapi.dev/api/films/1/*


```
{
  "characters": ["https://swapi.dev/api/people/1/", ...],
  "created": "2014-12-10T14:23:31.880000Z",
  "director": "George Lucas",
  "edited": "2014-12-12T11:24:39.858000Z",
  "episode_id": 4,
  "opening_crawl": "It is a period of civil war.\n\nRebel
spaceships, striking\n\nfrom a hidden base, have won\n\ntheir
first victory against\n\nthe evil Galactic
Empire.\n\n\n\nDuring the battle, Rebel\n\nspies managed to
steal secret\r\nplans to the Empire's\n\nultimate weapon, the
DEATH\n\nSTAR, an armored space\n\nstation with enough
power\n\nto destroy an entire planet.\n\n\n\nPursued by the
Empire's\n\nsinister agents, Princess\n\nLeia races home
aboard her\n\nstarship, custodian of the\n\nstolen plans that
can save her\n\npeople and restore\n\nfreedom to the
galaxy....",
  "planets": ["https://swapi.dev/api/planets/1/", ...],
  "producer": "Gary Kurtz, Rick McCallum",
  "release_date": "1977-05-25",
  "species": ["https://swapi.dev/api/species/1/", ...],
  "starships": ["https://swapi.dev/api/starships/2/", ...],
  "title": "A New Hope",
  "url": "https://swapi.dev/api/films/1/",
  "vehicles": ["https://swapi.dev/api/vehicles/4/", ...]
}
```


Fonte: swapi.dev. Adaptado pelo autor.

4.2.2 A classe Utils

De modo a tornar os exemplos mais objetivos e reduzir a duplicação de código, foi implementada uma classe de utilidades, que disponibiliza uma série de métodos necessários, mas que não fazem parte especificamente dos cenários estudados. Para realizar as requisições HTTP foi escolhida a biblioteca Unirest[3] devido à sua simplicidade de utilização. A

[3] http://kong.github.io/unirest-java/

implementação da classe `Utils` e de outras classes de apoio estão presentes no APÊNDICE C.

4.2.3 Cenário 1: Ordenar Personagens

A partir do *endpoint* `people`, construir uma tabela relacionando os personagens ao número de aparições em filmes, exibindo-os em ordem decrescente em número de aparições. A Tabela 7 ilustra o resultado esperado deste cenário.

Tabela 7 — Resultado esperado do cenário 1

Nome	Número de aparições
C-3PO	6
R2-D2	6
Obi-Wan Kenobi	6
Luke Skywalker	4
Darth Vader	4
Leia Organa	4
Owen Lars	3
Beru Whitesun lars	3
R5-D4	1
Biggs Darklighter	1

Fonte: Elaborado pelo autor com dados da swapi.dev.

A Figura 39 ilustra as implementações do cenário proposto anteriormente.

Figura 39 — Implementação do cenário 1

```
public static void executarVanilla() {
  CompletableFuture.supplyAsync(Utils::getAllPeople)
    .thenApply(ResultadosBusca::results)
    .thenApply(personagens -> {
     personagens.sort(
      (p1, p2) -> p2.films().size() - p1.films().size());
     return personagens;
    })
    .thenAccept(Utils::imprimirPersonagensENumeroFilmes)
    .join();
}

public static void executarFluentAsync() {
  new TarefaAssincrona<>(Utils::getAllPeople)
    .transformar(ResultadosBusca::results)
    .transformar(personagens -> {
     personagens.sort(
      (p1, p2) -> p2.films().size() - p1.films().size());
     return personagens;
    })
    .consumir(Utils::imprimirPersonagensENumeroFilmes)
    .aguardar();
}
```

Fonte: Elaborado pelo autor.

4.2.4 Cenário 2: Filtrar Personagens

A partir do *endpoint* `people`, construir uma tabela relacionando os personagens ao número de aparições em filmes, exibindo apenas os personagens aparições em três ou mais filmes. A Tabela 7 ilustra o resultado esperado deste cenário.

Tabela 8 — Resultado esperado do cenário 2

Nome	Número de aparições
Luke Skywalker	4
C-3PO	6
R2-D2	6
Darth Vader	4
Leia Organa	4
Owen Lars	3
Beru Whitesun lars	3
Obi-Wan Kenobi	6

Fonte: Elaborado pelo autor com dados da swapi.dev.

A Figura 40 ilustra as implementações do cenário proposto anteriormente.

Figura 40 — Implementação do cenário 2

```
public static void executarVanilla() {
  CompletableFuture.supplyAsync(Utils::getAllPeople)
    .thenApply(ResultadosBusca::results)
    .thenApply(personagens -> {
     personagens.removeIf(
      personagem -> personagem.films().size() < 3);
     return personagens;
    })
    .thenAccept(Utils::imprimirPersonagensENumeroFilmes)
    .join();
}

public static void executarFluentAsync() {
  new TarefaAssincrona<>(Utils::getAllPeople)
    .ramificar(ResultadosBusca::results)
    .filtrar(personagem -> personagem.films().size() >= 3)
    .unificar()
    .consumir(Utils::imprimirPersonagensENumeroFilmes)
    .aguardar();
}
```

Fonte: Elaborado pelo autor.

4.2.5 Cenário 3: Personagens e Filmes

A partir dos *endpoints* `people` e `films`, construir uma tabela relacionando personagens aos títulos dos filmes dos quais eles participam. A Tabela 9 ilustra o resultado esperado deste cenário.

Tabela 9 — Resultado esperado do cenário 3

Nome	Filmes
Luke Skywalker	A New Hope; The Empire Strikes Back; Return of the Jedi; Revenge of the Sith
C-3PO	A New Hope; The Empire Strikes Back; Return of the Jedi; The Phantom Menace; Attack of the Clones; Revenge of the Sith
R2-D2	A New Hope; The Empire Strikes Back; Return of the Jedi; The Phantom Menace; Attack of the Clones; Revenge of the Sith
Darth Vader	A New Hope; The Empire Strikes Back; Return of the Jedi; Revenge of the Sith
Leia Organa	A New Hope; The Empire Strikes Back; Return of the Jedi; Revenge of the Sith
Owen Lars	A New Hope; Attack of the Clones; Revenge of the Sith
Beru Whitesun lars	A New Hope; Attack of the Clones; Revenge of the Sith
R5-D4	A New Hope
Biggs Darklighter	A New Hope
Obi-Wan Kenobi	A New Hope; The Empire Strikes Back; Return of the Jedi; The Phantom Menace; Attack of the Clones; Revenge of the Sith

Fonte: Elaborado pelo autor com dados da swapi.dev.

A Figura 41 e a Figura 42 ilustram as implementações do cenário proposto anteriormente.

Figura 41 — Implementação do cenário 3 em Java puro

```
public static void executarVanilla() {
  CompletableFuture.supplyAsync(Utils::getAllPeople)
    .thenApply(resultadosBusca -> {
     var futurePersonagens =
       new ArrayList<CompletableFuture<Personagem>>();

     for (Personagem personagem : resultadosBusca.results())
     {
      var futurePersonagem =
        CompletableFuture.supplyAsync(() -> {
         var futuresFilmes =
           new ArrayList<CompletableFuture<Filme>>();

         for (Filme filme : personagem.films()) {
          var futureFilme =
            CompletableFuture.supplyAsync(() -> {
             Integer idFilme = filme.id();
             return Utils.getFilmById(idFilme);
            });
          futuresFilmes.add(futureFilme);
         }

         return personagem.comFilms( //
           futuresFilmes.stream()
             .map(CompletableFuture::join)
             .toList());
        });
      futurePersonagens.add(futurePersonagem);
     }
     return futurePersonagens.stream()
       .map(CompletableFuture::join)
       .toList();
    })
    .thenAccept(Utils::imprimirPersonagensETitulosFilmes)
    .join();
 }
```

Fonte: Elaborado pelo autor.

Figura 42 — Implementação do cenário 3 em Java utilizando FluentAsync

```
public static void executarFluentAsync() {
 new TarefaAssincrona<>(Utils::getAllPeople)
   .ramificar(ResultadosBusca::results)
   .transformar(personagem -> {
    var filmes =
      new TarefaAssincrona<Personagem>(() -> personagem)
        .ramificar(Personagem::films)
        .transformar(Filme::id)
        .transformar(Utils::getFilmById)
        .aguardar();

    return personagem.comFilms(filmes);
   })
   .unificar()
   .consumir(Utils::imprimirPersonagensETitulosFilmes)
   .aguardar();
}
```

Fonte: Elaborado pelo autor.

Os três cenários de exemplo apresentados anteriormente buscaram mostrar a mesma tarefa implementada com o uso da *CompletableFuture* e da FluentAsync. No próximo capítulo será feita uma avaliação da API FluentAsync, utilizando quatro métricas quantitativas existentes no campo da Engenharia de Software. Em seguida, os resultados são interpretados de modo a investigar um possível benefício no uso da FluentAsync em detrimento de técnicas tradicionais.

5 AVALIAÇÃO DA API

A avaliação da FluentAsync tem como objetivo observar se há indícios de que sua adoção pode trazer algum benefício para o desenvolvimento de programas assíncronos. Para tal, foram comparados os códigos construídos usando o modelo de programação assíncrona tradicional (*Vanilla*) com o modelo proposto pela FluentAsync (ambos implementados em Java), para os cenários apresentados nas seções 4.2.3, 4.2.4 e 4.2.5.

5.1 AVALIAÇÃO QUANTITATIVA

As comparações dos códigos foram feitas usando três métricas. O principal objetivo da adoção dessas métricas está em observar se as indicações obtidas com uma das métricas são fortalecidas ou enfraquecidas usando as demais, já que essas métricas não têm relação umas com as outras. As três métricas escolhidas para a avaliação foram:

- **Linhas de Código (LoC)**: é uma métrica usada desde meados da década de 1960 (FENTON e MARTIN, 2000) e é usada para medir o tamanho de um programa. Apesar de ser antiga, ainda é uma das métricas mais frequentemente usadas atualmente (NUÑEZ-VARELA, PÉREZ-GONZALEZ, *et al.*, 2017). Uma das críticas a essa métrica é que não faz sentido comparar a quantidade de linhas de dois códigos quanto eles são criados usando duas linguagens de programação diferentes. Entretanto, neste trabalho, essa medição será realizada em dois códigos escritos na mesma linguagem de programação e usando o mesmo estilo de formatação.
- **Métricas de Complexidade de Halstead:** as métricas de Halstead são obtidas a partir da análise estática do código-fonte. Foram desenvolvidas por Maurice Howard Halstead em 1977 como um método de avaliar a complexidade computacional de um programa a partir dos seus "Operadores" e "Operandos" (BRAY, BRUNE, *et al.*, 1997, p. 209-212). As métricas de Halstead completas estão disponíveis no APÊNDICE E.
- **Complexidade Ciclomática (V(G)):** proposta por Thomas J. McCabe em 1976 (MCCABE, 1976), a Complexidade Ciclomática fornece uma indicação da complexidade de um programa através da análise dos diferentes "caminhos" que a execução desse programa pode tomar. Apesar de também ter sido proposta há

bastante tempo, ela ainda é uma das métricas de complexidade mais usadas na academia e na indústria (NUÑEZ-VARELA, PÉREZ-GONZALEZ, *et al.*, 2017).

- **Modified Cognitive Complexity Measure (MCCM):** proposta por Misra (2006), essa métrica faz parte de um conjunto de métricas denominadas métricas de complexidade cognitiva, introduzidas por Shao e Wang (2003). Essas métricas adotam o conceito de peso cognitivo associados às estruturas básicas de controle utilizadas no código. Tais pesos "definem o esforço necessário, tempo relativo ou grau de dificuldade para compreender o software" (SHAO e WANG, 2003).

5.1.1 Linhas de Código (LoC)

Para evitar viés na contagem, as quantidades de linhas de código foram obtidas a partir do código-fonte dos exemplos, formatados automaticamente pelo *Eclipse IDE for Enterprise Java and Web Developers 2022-09*[4], utilizando o perfil de formatação *Eclipse 2.1* e sem modificações. Um perfil de formatação diferente foi utilizado para a exibição dos exemplos neste trabalho, de modo a facilitar a legibilidade fora de um ambiente de desenvolvimento integrado. Esta mudança justifica a diferença entre o número de linhas de código dos exemplos deste trabalho e as medidas obtidas nesta seção.

Tabela 10 — Comparação de número de linhas de código entre as implementações

Cenário	Java Vanilla	FluentAsync	%
Cenário 1: Ordenar Personagens	6	7	116,6
Cenário 2: Filtrar Personagens	6	5	83,3
Cenário 3: Personagens e Filmes	27	9	33,3

Fonte: Elaborado pelo autor.

Com os dados da Tabela 10, é possível observar que a quantidade de linhas de código necessária para realizar a mesma tarefa utilizando Java *Vanilla* e FluentAsync é similar para problemas mais simples (cenários 1 e 2). Entretanto, a quantidade de linhas necessárias para a versão Java *Vanilla* torna-se maior à medida que os requisitos se tornam mais complexos (cenário 3 comparado aos cenários 1 e 2). Por outro lado, a versão FluentAsync teve um aumento bastante discreto, comparando os cenários mais simples (cenários 1 e 2) com o mais complexo (cenário 3). Esse comportamento levou a versão FluentAsync do cenário 3 a ter

[4] https://www.eclipse.org/downloads/packages/release/2022-09/r/eclipse-ide-enterprise-java-and-web-developers

somente 33,3% das linhas da versão correspondente em Java *Vanilla*. Esses valores nos fornecem indicações de que tarefas assíncronas implementadas com o FluentAsync podem ter uma quantidade de linhas consideravelmente menor, se comparadas com a versão Java *Vanilla* usando CompletableFuture.

5.1.2 Métricas de Complexidade de Halstead

As regras para a determinação de Operadores e Operandos usados nas métricas de Halstead não são estritamente definidas e variam de acordo com a linguagem. Para a avaliação dos três cenários, foram definidas as seguintes regras para a contagem de Operadores e Operandos.

Os Operadores são:

- Chamadas de função;
- Palavras de controle de fluxo, como `if`, `else`, `switch`, `case`, `default`, `do`, `while`, `for`, `break` e `continue`;
- Palavras reservadas, como `return`, `var` e `new`;
- Operadores funcionais `->` e `::`;
- Pares de chaves delimitadoras de blocos `{}`;
- Operadores de comparação ==, `!=`, >, <, >=, <=;
- Operadores aritméticos +, −, `*`, `/`, `%`, ++, −−;
- Operadores de atribuição =, +=, −=, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, >>=, <<=;
- Operadores lógicos `&&`, `||`, `!`.

Os Operandos são:

- Nomes de classes;
- Referências de variáveis;
- Nomes de métodos, quando referenciados utilizando o operador `::` (por exemplo, `Utils::getAllPeople`).

A Tabela 11 compara as métricas de Halstead das implementações dos três cenários. Em todas as comparações, a implementação usando FluentAsync apresentam métricas melhores que suas alternativas *Vanilla*, com exceção das métricas $n2$ e N do cenário 1, onde os valores foram iguais para ambas as implementações.

Tabela 11 — Comparação das medidas de complexidade de Halstead entre as implementações

Métrica	Cenário 1			Cenário 2			Cenário 3		
	Vanilla	F.A.	%	Vanilla	F.A.	%	Vanilla	F.A.	%
Operadores distintos ($n1$)	12	**11**	91,6	12	**11**	91,6	21	**13**	61,9
Ocorrências de operadores ($N1$)	18	**17**	94,4	18	**13**	72,2	47	**32**	68,0
Operandos distintos ($n2$)	9	9	100,0	9	**7**	77,7	16	**13**	81,2
Ocorrências de operandos ($N2$)	**13**	14	107,6	12	**9**	75,0	34	**26**	76,4
Vocabulário ($n = n1 + n2$)	21	**20**	95,2	21	**18**	85,7	37	**26**	70,2
Tamanho ($N = N1 + N2$)	31	31	100,0	30	**22**	73,3	81	**58**	71,6
Volume ($V = N \log_2 n$)	136,1	**133**	97,7	131,7	**91,7**	69,6	421,9	**272,6**	64,6
Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)	8,6	**8,5**	98,8	8	**7**	87,5	22,3	**13**	58,3
Esforço ($E = D \times V$)	1180	**1146,2**	97,1	1054,1	**648,7**	61,5	9415,1	**3544,1**	37,6

Fonte: Elaborado pelo autor.

O mesmo fenômeno observado na quantidade de Linhas de Código se aplica às métricas de Halstead: à medida que a complexidade dos requisitos aumenta, a implementação com uso do FluentAsync torna-se sensivelmente mais simples quando comparada com a sua versão em Java *Vanilla*. Tendo em vista os valores da métrica de Esforço de Halstead, que é calculada a partir do resultado de todas as métricas anteriores, é possível observar que o Esforço calculado para as implementações utilizando o FluentAsync se tornam proporcionalmente cada vez menores em relação às implementações *Vanilla*. No cenário 3, a implementação utilizando FluentAsync representou apenas 37,6% do esforço em relação à implementação *Vanilla*. Ainda, diferentemente da métrica de Linhas de Código, foi possível detectar os benefícios do uso do FluentAsync em requisitos mais simples, como no Cenário 2, onde a implementação com FluentAsync representou 61,5% do esforço em comparação com o Java *Vanilla*. Essas métricas indicam que o uso do FluentAsync pode reduzir o esforço (e consequentemente o tempo) necessário para desenvolver programas assíncronos, quando comparado com o Java *Vanilla*.

5.1.3 Complexidade Ciclomática de McCabe

Tabela 12 — Comparação de Complexidade Ciclomática entre as implementações

Cenário	Java Vanilla	FluentAsync	%
Cenário 1: Ordenar Personagens	1	1	100
Cenário 2: Filtrar Personagens	1	1	100
Cenário 3: Personagens e Filmes	3	1	33,3

Fonte: Elaborado pelo autor.

A Tabela 12 relaciona as Complexidades Ciclomáticas aferidas pelas implementações dos cenários propostos. O terceiro é o único cenário que apresenta uma diferença entre as implementações, sendo a FluentAsync a menos complexa. Isto se dá pelo fato de que a Ramificação implementada em Java *Vanilla* requer o uso de loops para a criação de instâncias de novas tarefas, algo que é abstraído pela FluentAsync. Consequentemente, as operações de Ramificação da FluentAsync são menos complexas sob o ponto de vista da Complexidade Ciclomática de McCabe, e é esperado que este resultado se repita em outros cenários onde Ramificações e Unificações sejam necessárias. Dessa forma, o mesmo comportamento observado nas métricas anteriores se repete aqui: para os cenários menos complexos (cenários 1 e 2) não foi observada nenhuma diferença, porém, para o cenário mais complexo (cenário 3) houve uma redução não desprezível da Complexidade Ciclomática.

5.1.4 Modified Cognitive Complexity Measure (MCCM)

A Tabela 13 compara os valores calculados métrica de complexidade cognitiva proposta por Shao e Wang, uma vez que estes valores são necessários para o cálculo da MCCM introduzida por Misra.

Tabela 13 — Comparação de Cognitive Weight Complexity (W_c) entre as implementações

Cenário	Java Vanilla	FluentAsync	%
Cenário 1: Ordenar Personagens	14	14	100
Cenário 2: Filtrar Personagens	14	15	107,1
Cenário 3: Personagens e Filmes	328	206	62,8

Fonte: Elaborado pelo autor.

A MCCM de Misra é calculada como $MCCM = (N1 + N2) \times W_c$, onde $N1$ e $N2$ são as métricas de ocorrências de operadores e de ocorrências de operandos de Halstead,

respectivamente. A Tabela 14 apresenta os valores de MCCM calculados para cada implementação.

Tabela 14 — Comparação de Modified Cognitive Complexity Measure (MCCM) entre as implementações

Cenário	Java Vanilla	FluentAsync	%
Cenário 1: Ordenar Personagens	434	434	100
Cenário 2: Filtrar Personagens	420	330	78,5
Cenário 3: Personagens e Filmes	26.568	11.948	44,9

Fonte: Elaborado pelo autor.

O mesmo efeito encontrado nas avaliações anteriores foi observado nos valores de MCCM. A FluentAsync se mostrou drasticamente menos complexa no cenário 3, onde a métrica de complexidade cognitiva modificada de Misra foi de 44,9% do valor da alternativa *Vanilla*. Este efeito também foi observado no cenário 2, porém com menor intensidade (78,5% em relação à implementação *Vanilla*).

5.2 CONCLUSÃO DAS AVALIAÇÕES

As diversas métricas usadas na avaliação oferecem perspectivas distintas de análise do código. Todas elas foram consistentes em indicar que, para cenários simples, não há diferença entre os códigos que usam o Java *Vanilla* (*CompletableFuture*) e a FluentAsync ou ela é bastante pequena. Entretanto, quando a complexidade do cenário aumenta, essas mesmas métricas indicaram um distanciamento consistente em favor da FluentAsync, que devem ser confirmados com estudos mais elaborados.

Um mapeamento sistemático elaborado por Nuñez-Varela *et al.* (2017) lista mais de 200 artigos, publicados no período de 2010 a 2015, que usam métricas de código para diversos tipos de pesquisa na área de medição. Dentre esses, alguns trabalhos buscam relacionar características como legibilidade, compreensibilidade e manutenibilidade com diversas métricas, incluindo aquelas aqui adotadas.

Um ponto a destacar é que o tempo gasto para o desenvolvimento das implementações não foi medido, por isso essa medida não foi levada em consideração nas comparações realizadas. Além disso, é necessário frisar que as implementações *Vanilla*, em especial no Cenário 3, poderiam ser desmembradas em múltiplas funções, com objetivo de reduzir a quantidade de aninhamentos no código-fonte. Isto não foi realizado de modo a não impactar negativamente na métrica de Linhas de Código, uma vez que a declaração de funções resulta no uso de mais linhas, e na métrica MCCM, que atribui peso 2 para invocações de funções.

Por fim, todas as implementações aqui apresentadas foram codificadas pelo autor deste trabalho. Portanto, há de se considerar a existência de vieses inconscientes que possam ter impactado a produção do código, e consequentemente, os resultados obtidos. Assim, os resultados da avaliação devem ser encarados somente como indicadores preliminares das vantagens oferecidas pelo uso da FluentAsync, e não como uma demonstração da sua proeminência.

6 CONCLUSÃO

6.1 CONSIDERAÇÕES FINAIS

O objetivo deste trabalho foi desenvolver uma nova API, chamada FluentAsync, voltada para o desenvolvimento de programas assíncronos que não estivesse limitada apenas a uma linguagem específica. Assim, inspirado nas etapas previstas na DSR, alguns padrões comumente observados na programação assíncrona foram definidos. Em seguida, foi desenvolvida uma implementação de referência da API em Java, e, de modo a comprovar a característica "agnóstica de linguagem" do modelo, uma versão em TypeScript também foi escrita. O repositório contendo as duas implementações pode ser acessado a partir do endereço *https://mateusbandeira.dev/projects/fluentasync*.

Para que os recursos da API fossem efetivamente usados e avaliados, foram escolhidos 3 cenários com requisitos comuns em programas reais, e soluções foram implementadas para satisfazer estes requisitos, tanto utilizando a API FluentAsync em Java, quanto utilizando o Java puro com o *CompletableFuture*. As implementações foram avaliadas quantitativamente através de quatro métricas e foram obtidas indicações de que o uso da FluentAsync tornou as implementações menores e menos complexas. Além disso, as avaliações dos 3 cenários indicaram que os benefícios do uso da FluentAsync em Java tendem a ser maiores quando os requisitos de processamento assíncrono se tornam mais complexos.

6.2 CONTRIBUIÇÕES

A FluentAsync introduz uma nova camada de abstração para a realização de tarefas assíncronas, permitindo que o código-fonte resultante seja mais próximo dos problemas de negócio sendo resolvidos do que os problemas técnicos associados à coordenação de tarefas em paralelo. Os métodos fornecidos se mostraram capazes de reduzir a complexidade das implementações resultantes, e o significados sintáticos introduzidos por eles tornam a abstração mais tangível aos implementadores.

Por ser "agnóstica de linguagem", a FluentAsync pode ser implementada em outras linguagens além das vislumbradas neste trabalho. Assim, a experiência adquirida por um programador utilizando a API em uma linguagem pode ser aproveitada ao utilizá-la em outras linguagens, uma vez que as operações e comportamentos são equivalentes.

Além disso, a FluentAsync, por ser versátil e flexível, pode ser adaptada às necessidades específicas de um projeto ou combinada com outras bibliotecas e tecnologias, permitindo que os programadores possam criar soluções ainda mais robustas. Em resumo, a FluentAsync oferece uma opção atraente para programadores que procuram uma maneira mais clara e intuitiva de lidar com tarefas assíncronas em seu código.

6.3 TRABALHOS FUTUROS

A FluentAsync demonstrou uma versatilidade considerável, principalmente em face da sua reduzida coleção de métodos. Futuras versões da FluentAsync poderiam incluir novas operações que não foram propostas no modelo original, mas que podem adicionar ainda mais potencialidades à API. Portanto, um trabalho voltado a identificar possíveis lacunas no design da FluentAsync seriam úteis neste sentido.

A implementação e avaliação de outros cenários também são necessárias para aprofundar o entendimento da real contribuição do uso da FluentAsync e verificar se os indícios aqui obtidos permanecem válidos. Idealmente, tais cenários deveriam ser desenvolvidos por programadores que não tivessem envolvidos com o desenvolvimento da API, de forma a diminuir o viés dos resultados das avaliações e detectar eventuais dificuldades no seu uso. Por fim, características como legibilidade, compreensibilidade e manutenibilidade também poderiam ser avaliadas nesses estudos.

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 4. ed. [S.l.]: Addison-Wesley, 2021.

BELSON, B. et al. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. **ACM Transactions on Embedded Computing Systems**, 18, n. 3, maio 2019.

BIERMAN, G. et al. Pause 'n' play: Formalizing asynchronous C sharp. **ECOOP 2012 - Object-Oriented Programming**, junho 2012. 233-257.

BRAY, M. W. et al. **C4 Software Technology Reference Guide:** A Prototype. 1. ed. Pittsburgh: [s.n.], 1997.

FENTON, N. E.; MARTIN, N. Software Metrics: Roadmap. **Proceedings of the Conference on the Future of Software Engineering**, Nova Iorque, set. 2000. 357-370.

FOWLER, M. FluentInterface. **martinFowler.com**, 2005. Disponivel em: <https://martinfowler.com/bliki/FluentInterface.html>. Acesso em: 30 janeiro 2023.

FOWLER, M.; LEWIS, J. Microservices. **martinFowler.com**, 2014. Disponivel em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 3 agosto 2022.

GHOSH, D. **DSLs in Action**. Stamford: Manning Publications Co., 2011.

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, 1976. 308-320.

MEHLHORN, N.; HANENBERG, S. **Imperative versus Declarative Collection Processing:** An RCT on the Understandability of Traditional Loops versus the Stream API in Java. 44th International Conference on Software Engineering (ICSE '22). Nova Iorque: Association for Computing Machinery. 2022. p. 1157-1168.

MISRA, S. **Modified Cognitive Complexity Measure**. International Symposium on Computer and Information Sciences. [S.l.]: [s.n.]. 2006.

NUÑEZ-VARELA, A. S. et al. Source code metrics: A systematic mapping study. **Journal of Systems and Software**, 2017. 164-197.

O'SULLIVAN, B.; STEWART, D.; GOERZEN, J. **Real World Haskell**. [S.l.]: O'Reilly, 2009.

SHAO, J.; WANG, Y. **A new measure of software complexity based on cognitive weights**. CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology. [S.l.]: [s.n.]. 2003. p. 1333-1338 vol.2.

VAISHNAVI, V.; KUECHLER, W. Design Science Research in Information Systems. **Association for Information Systems**, janeiro 2004.

APÊNDICE A — Especificação do FluentAsync

TarefaAssincrona

TarefaAssincrona(Supplier<T> tarefa) (construtor)

Constrói uma instância de `TarefaAssincrona<T>`. O argumento `Supplier<T> tarefa` é a função a ser executada assincronamente, eventualmente retornando um objeto do tipo `T`.

T aguardar()

Retorna o valor desta `TarefaAssincrona` caso esta esteja terminada. Caso contrário, suspende a execução do código até o seu término.

TarefaAssincrona<Void> consumir(Consumer<T> consumidora)

Retorna uma nova `TarefaAssincrona<Void>`, cuja tarefa é o encadeamento do argumento `Consumer<T> consumidora` após o término desta `TarefaAssincrona`. A função `consumidora` deve receber como argumento o resultado desta tarefa, e não deve retornar nenhum valor.

<U> TarefaAssincrona<U> transformar(Function<T, U> transformadora)

Retorna uma nova `TarefaAssincrona<U>`, cuja tarefa é o encadeamento do argumento `Function<T, U> transformadora` após o término desta `TarefaAssincrona`. A função `transformadora` deve receber como argumento o resultado desta tarefa e deve retornar um objeto do tipo `U`, que será o resultado da nova instância de `TarefaAssincrona<U>` retornada.

<U> TarefaRamificada<U> ramificar(
Function<T, List<U>> funcaoRamificadora)

Retorna uma nova `TarefaRamificada<U>`, a partir do argumento `Function<T, List<U>> funcaoRamificadora`, uma função que recebe o resultado desta `TarefaAssincrona` e retorna uma lista de objetos do tipo `U`. Cada elemento da lista retornada será convertido em uma nova tarefa paralelizada dentro de `TarefaRamificada`.

TarefaRamificada

<T> TarefaRamificada(
TarefaAssincrona<T> tarefaOriginal,
Function<T, List<R>> funcaoRamificadora) (construtor)

Constrói uma nova `TarefaRamificada<T>`, cujo valor da tarefa passada como argumento `TarefaAssincrona<T> tarefaOriginal` será processado pela função

passada como argumento `Function<T, List<R>> funcaoRamificadora`, e dará origem às tarefas paralelizadas por esta `TarefaRamificada`.

<U> TarefaAssincrona<U> unificar(Function<List<R>, U> funcaoUnificadora)

Equivalente a invocar o método `TarefaAssincrona<List<R>> unificar()` desta `TarefaRamificada`, com a posterior execução do argumento `Function<List<R>, U> funcaoUnificadora` para transformar a lista de resultados em um objeto arbitrário.

TarefaAssincrona<List<R>> unificar()

Equivalente a invocar o método `T aguardar()` de cada uma das tarefas paralelizadas por esta `TarefaRamificada`, retornando uma nova instância de `TarefaAssincrona` cujo resultado é a lista de resultados das tarefas paralelizadas por esta `TarefaRamificada`.

<U> TarefaRamificada<U> transformar(Function<R, U> transformadora)

Equivalente a invocar o método `TarefaAssincrona<U> transformar(Function<T, U> transformadora)` de cada uma das tarefas paralelizadas por esta `TarefaRamificada`.

TarefaRamificada<Void> consumir(Consumer<R> consumidora)

Equivalente a invocar o método `TarefaAssincrona<Void> consumir(Consumer<T> consumidora)` de cada uma das tarefas paralelizadas por esta `TarefaRamificada`.

TarefaRamificada<R> filtrar(Predicate<R> filtro)

Retorna uma nova `TarefaRamificada`, cujas tarefas paralelizadas são um subconjunto das tarefas paralelizadas desta `TarefaRamificada`. O argumento `Predicate<R> filtro` define o critério para inclusão das tarefas, onde serão incluídas apenas as tarefas cuja chamada retorne `true`.

List<R> aguardar()

Equivalente a invocar o método `unificar()` desta `TarefaRamificada`, e em seguida, invocar o método `aguardar()` da `TarefaAssincrona` retornada por ele.

<U> U aguardar(Function<List<R>, U> funcaoUnificadora)

Equivalente a invocar o método `unificar(Function<List<R>, U> funcaoUnificadora)` desta `TarefaRamificada`, e em seguida, invocar o método `aguardar()` da `TarefaAssincrona` retornada por ele.

APÊNDICE B — Implementação do FluentAsync em Java

TarefaAssincrona.java

```
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class TarefaAssincrona<T> {
 protected final CompletableFuture<T> tarefa;

 private TarefaAssincrona(CompletableFuture<T> tarefa) {
  this.tarefa = tarefa;
 }

 public TarefaAssincrona(Supplier<T> tarefa) {
  this(CompletableFuture.supplyAsync(tarefa));
 }

 public T aguardar() {
  return tarefa.join();
 }

 public TarefaAssincrona<Void> consumir(
   Consumer<T> consumidora) {
  return new TarefaAssincrona<>(
    tarefa.thenAcceptAsync(consumidora));
 }

 public <U> TarefaAssincrona<U> transformar(
   Function<T, U> transformadora) {
  return new TarefaAssincrona<>(
    tarefa.thenApplyAsync(transformadora));
 }

 public <U> TarefaRamificada<U> ramificar(
   Function<T, List<U>> funcaoRamificadora) {
  return new TarefaRamificada<>(this, funcaoRamificadora);
 }
```

```
 public TarefaAssincrona<T> casoErro(
   Function<Throwable, ? extends T> casoErro) {
  return new TarefaAssincrona<>(
    tarefa.exceptionally(casoErro));
 }

}
```

TarefaRamificada.java

```
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class TarefaRamificada<R> {
 protected final TarefaAssincrona<List<TarefaAssincrona<R>>>
   ramos;

 public <T> TarefaRamificada(
   TarefaAssincrona<T> tarefaOriginal,
   Function<T, List<R>> funcaoRamificadora) {
  this.ramos =
    tarefaOriginal.transformar(funcaoRamificadora)
      .transformar(listaRamos -> listaRamos.stream()
        .map(ramo -> new TarefaAssincrona<>(() -> ramo))
        .toList());
 }

 private TarefaRamificada(
   TarefaAssincrona<List<TarefaAssincrona<R>>> ramos) {
  this.ramos = ramos;
 }

 public <U> TarefaAssincrona<U> unificar(
   Function<List<R>, U> funcaoUnificadora) {
  return this.unificar().transformar(funcaoUnificadora);
 }
```

```
public TarefaAssincrona<List<R>> unificar() {
 return this.ramos
   .transformar(listaRamos -> listaRamos.parallelStream()
     .map(TarefaAssincrona::aguardar)
     .toList());
}

public <U> TarefaRamificada<U> transformar(
  Function<R, U> transformadora) {
 TarefaAssincrona<List<TarefaAssincrona<U>>> transformar =
   ramos.transformar(r -> r.stream()
     .map(ramo -> ramo.transformar(transformadora))
     .toList());
 return new TarefaRamificada<>(transformar);
}

public TarefaRamificada<Void> consumir(
  Consumer<R> consumidora) {
 TarefaAssincrona<List<TarefaAssincrona<Void>>> consumir =
   ramos.transformar(r -> r.stream()
     .map(ramo -> ramo.consumir(consumidora))
     .toList());
 return new TarefaRamificada<>(consumir);
}

public TarefaRamificada<R> filtrar(Predicate<R> filtro) {
 TarefaAssincrona<List<TarefaAssincrona<R>>> ramosFiltrados =
   ramos.transformar(r -> r.stream()
     .filter(ramo -> filtro.test(ramo.aguardar()))
     .toList());
 return new TarefaRamificada<>(ramosFiltrados);
}

public List<R> aguardar() {
 return this.unificar().aguardar();
}
```

```
 public <U> U aguardar(
   Function<List<R>, U> funcaoUnificadora) {
  return this.unificar(funcaoUnificadora).aguardar();
 }
}
```

APÊNDICE C — Implementações de outras classes utilizadas nos exemplos

Utils.java

```
import java.util.List;
import java.util.stream.Collectors;
import kong.unirest.GenericType;
import kong.unirest.Unirest;
import kong.unirest.UnirestInstance;
import kong.unirest.jackson.JacksonObjectMapper;

public class Utils {

 private static final String URI_BASE =
   "https://swapi.dev/api";

 private static UnirestInstance http;

 static {
  http = Unirest.spawnInstance();
  http.config().defaultBaseUrl(URI_BASE);
  http.config().setObjectMapper(new JacksonObjectMapper());
 }

 private Utils() {
 }

 public static ResultadosBusca<Personagem> getAllPeople() {
  var httpResponse = http.get("/people")
    .asObject(
      new GenericType<ResultadosBusca<Personagem>>() {
      });
  if (httpResponse.getParsingError().isPresent()) {
   throw httpResponse.getParsingError().get();
  }
  return httpResponse.getBody();
 }
```

```
 public static Filme getFilmById(Integer idFilme) {
  var httpResponse = http.get("/films/{id}")
    .routeParam("id", String.valueOf(idFilme))
    .asObject(Filme.class);
  if (httpResponse.getParsingError().isPresent()) {
   throw httpResponse.getParsingError().get();
  }
  return httpResponse.getBody();
 }

 public static void imprimirPersonagensENumeroFilmes(
   List<Personagem> resultados) {

  var formato = "| %25s | %10s |\n";

  System.out.printf(formato, "Nome", "# filmes");

  for (Personagem personagem : resultados) {
   System.out.printf(formato, personagem.name(),
     personagem.films().size());
  }
 }

 public static void imprimirPersonagensETitulosFilmes(
   List<Personagem> resultados) {

  var formato = "| %25s | %-120s |\n";

  System.out.printf(formato, "Nome", "Filmes");

  for (Personagem personagem : resultados) {
   var filmes = personagem.films()
     .stream()
     .map(Filme::title)
     .collect(Collectors.joining("; "));
   System.out.printf(formato, personagem.name(), filmes);
  }
 }
}
```

Personagem.java

```
import java.util.List;

public record Personagem(Integer id, String name,
  Integer height, Integer mass, String hairColor,
  String eyeColor, String birthYear, String gender,
  List<Filme> films) {
 public Personagem comFilms(List<Filme> films) {
  return new Personagem(id, name, height, mass, hairColor,
    eyeColor, birthYear, gender, films);
 }
}
```

Filme.java

```
import java.time.LocalDate;

import
com.fasterxml.jackson.databind.annotation.JsonDeserialize;

@JsonDeserialize(using = FilmDeserializer.class)
public record Filme(Integer id, String title,
  String director, LocalDate releaseDate) {
}
```

ResultadosBusca.java

```
import java.util.List;

public record ResultadosBusca<T> (Integer count,
  String next, String previous, List<T> results) {

}
```

FilmDeserializer.java

```
import java.io.IOException;
import java.time.LocalDate;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.JsonNode;
import
com.fasterxml.jackson.databind.deser.std.StdDeserializer;
import com.fasterxml.jackson.databind.node.JsonNodeType;

public class FilmDeserializer
  extends StdDeserializer<Filme> {

 private static final Pattern FILM_ID_PATTERN =
   Pattern.compile("\\d+(?=\\/$)");

 public FilmDeserializer() {
  this(null);
 }

 public FilmDeserializer(Class<?> vc) {
  super(vc);
 }

 private static final long serialVersionUID =
   -6209245788472960377L;
```

```
@Override
public Filme deserialize(JsonParser p,
  DeserializationContext ctxt) throws IOException {
 JsonNode node = p.getCodec().readTree(p);
 if (node.getNodeType().equals(JsonNodeType.STRING)) {
  String filmUri = node.asText();

  Matcher matcher = FILM_ID_PATTERN.matcher(filmUri);
  if (matcher.find()) {
   String filmId = matcher.group();
   return new Filme(Integer.valueOf(filmId), null, null,
     null);
  }
 }
 String title = node.get("title").asText();
 String director = node.get("director").asText();
 LocalDate releaseDate =
   LocalDate.parse(node.get("release_date").asText());

 return new Filme(null, title, director, releaseDate);
}

}
```

APÊNDICE D — Implementação do FluentAsync em TypeScript

TarefaAssincrona.ts

```
import TarefaRamificada from "./TarefaRamificada.js";

export default class TarefaAssincrona<T> {
  tarefa: Promise<T>;
  constructor(
    tarefa: Promise<T> | (() => Promise<T>) | (() => T) | T
  ) {
    if (tarefa instanceof Promise<T>) {
      this.tarefa = tarefa;
    } else if (tarefa instanceof Function) {
      const retorno = tarefa();
      if (retorno instanceof Promise<T>) {
        this.tarefa = retorno;
      } else {
        this.tarefa = new Promise((resolve) =>
          resolve(retorno)
        );
      }
    } else {
      this.tarefa = new Promise((resolve) =>
        resolve(tarefa)
      );
    }
  }

  obterPromise(): Promise<T> {
    return this.tarefa;
  }
```

```
  transformar<U>(
    transformadora: (value: T) => U
  ): TarefaAssincrona<U> {
    return new TarefaAssincrona(
      this.tarefa.then(transformadora)
    );
  }

  consumir(
    consumidora: (value: T) => void
  ): TarefaAssincrona<T> {
    return new TarefaAssincrona(
      this.tarefa.then((val) => {
        consumidora(val);
        return val;
      })
    );
  }

  ramificar<U>(
    funcaoRamificadora: (a: T) => Array<U>
  ): TarefaRamificada<U> {
    return TarefaRamificada.instanciar(
      this,
      funcaoRamificadora
    );
  }
}
```

TarefaRamificada.ts

```
import TarefaAssincrona from "./TarefaAssincrona.js";

export default class TarefaRamificada<R> {
  ramos: TarefaAssincrona<TarefaAssincrona<R>[]>;

  private constructor(
    ramos: TarefaAssincrona<TarefaAssincrona<R>[]>
  ) {
    this.ramos = ramos;
  }

  static instanciar<U, R>(
    tarefaOriginal: TarefaAssincrona<U>,
    funcaoRamificadora: (a: U) => Array<R>
  ): TarefaRamificada<R> {
    const futurosRamos = tarefaOriginal
      .transformar(funcaoRamificadora)
      .transformar((valores) =>
        valores.map((valor) => new TarefaAssincrona(valor))
      );
    return new TarefaRamificada(futurosRamos);
  }

  obterPromise() {
    return this.unificar().obterPromise();
  }

  transformar<U>(
    transformadora: (value: R) => U
  ): TarefaRamificada<U> {
    return new TarefaRamificada(
      this.ramos.transformar((ramos) => {
        return ramos.map((ramo) =>
          ramo.transformar(transformadora)
        );
      })
    );
  }
```

```
  consumir<U>(
    consumidora: (value: R) => U
  ): TarefaRamificada<R> {
    return new TarefaRamificada(
      this.ramos.transformar((ramos) => {
        return ramos.map((ramo) =>
          ramo.consumir(consumidora)
        );
      })
    );
  }

  unificar<U>(
    funcaoUnificadora?: (valores: R[]) => U
  ): TarefaAssincrona<R[]> | TarefaAssincrona<U> {
    const tarefaUnificada = new TarefaAssincrona(
      async () => {
        return await this.ramos
          .transformar((ramos) =>
            ramos.map((ramo) => ramo.obterPromise())
          )
          .transformar((promises) => Promise.all(promises))
          .obterPromise();
      }
    );
    if (!funcaoUnificadora) {
      return tarefaUnificada;
    } else {
      return tarefaUnificada.transformar(funcaoUnificadora);
    }
  }
}
```

APÊNDICE E — Métricas de Halstead

Cenário 1							
Vanilla				**FluentAsync**			
Operadores ($n1$)	**Ocorrências ($N1$)**	**Operandos ($n2$)**	**Ocorrências ($N2$)**	**Operadores ($n1$)**	**Ocorrências ($N1$)**	**Operandos ($n2$)**	**Ocorrências ($N2$)**
supplyAsync	1	CompletableFuture	1	new	1	TarefaAssincrona	1
::	3	Utils	2	::	3	Utils	2
thenApply	2	getAllPeople	1	transformar	2	getAllPeople	1
->	2	ResultadosBusca	1	->	2	ResultadosBusca	1
{}	1	results	1	{}	1	results	1
sort	1	personagens	3	sort	1	personagens	3
films	2	p1	2	films	2	p1	2
size	2	p2	2	size	2	p2	2
-	1	Imprimir Personagens ENumeroFilmes	1	return	1	Imprimir Personagens ENumeroFilmes	1
return	1			consumir	1		
thenAccept	1			aguardar	1		
join	1						
12	**18**	**9**	**13**	**11**	**17**	**9**	**14**

Vocabulário ($n = n1 + n2$)	21
Tamanho ($N = N1 + N2$)	31
Volume ($V = N \log_2 n$)	136,1618401
Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)	8,666666667
Esforço ($E = D \times V$)	1180,069281

Vocabulário ($n = n1 + n2$)	20
Tamanho ($N = N1 + N2$)	31
Volume ($V = N \log_2 n$)	133,9797709
Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)	8,555555556
Esforço ($E = D \times V$)	1146,271374

Cenário 2							
Vanilla				**FluentAsync**			
Operadores ($n1$)	**Ocorrências ($N1$)**	**Operandos ($n2$)**	**Ocorrências ($N2$)**	**Operadores ($n1$)**	**Ocorrências ($N1$)**	**Operandos ($n2$)**	**Ocorrências ($N2$)**
supplyAsync	1	CompletableFuture	1	new	1	TarefaAssincrona	1
::	3	Utils	2	::	3	Utils	2
thenApply	3	getAllPeople	1	ramificar	1	getAllPeople	1
->	3	ResultadosBusca	1	filtrar	1	ResultadosBusca	1
{}	1	results	1	->	1	results	1
removeIf	1	personagens	3	films	1	personagem	2
films	1	personagem	2	size	1	Imprimir Personagens ENumeroFilmes	1
size	1	3	1	>=	1		
<	1	Imprimir Personagens ENumeroFilmes	1	unificar	1		
return	1			consumir	1		
thenAccept	1			aguardar	1		
join	1						
12	**18**	**9**	**13**	**11**	**13**	**7**	**9**

Vocabulário ($n = n1 + n2$)	21
Tamanho ($N = N1 + N2$)	30
Volume ($V = N \log_2 n$)	131,7695227
Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)	8
Esforço ($E = D \times V$)	1054,156181

Vocabulário ($n = n1 + n2$)	18
Tamanho ($N = N1 + N2$)	22
Volume ($V = N \log_2 n$)	91,73835003
Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)	7,071428571
Esforço ($E = D \times V$)	648,7211895

Cenário 3							
Vanilla				FluentAsync			
Operadores ($n1$)	**Ocorrências ($N1$)**	**Operandos ($n2$)**	**Ocorrências ($N2$)**	**Operadores ($n1$)**	**Ocorrências ($N1$)**	**Operandos ($n2$)**	**Ocorrências ($N2$)**
supplyAsync	3	CompletableFuture	7	new	2	TarefaAssincrona	2
::	4	Utils	3	::	6	Utils	3
thenApply	1	getAllPeople	1	ramificar	2	getAllPeople	2
->	3	resultadosBusca	2	transformar	3	ResultadosBusca	2
{}	5	futurePersonagens	3	->	3	results	1
var	4	ArrayList	3	{}	2	personagem	3
=	5	Personagem	2	var	2	filmes	3
new	2	personagem	3	=	2	Personagem	2
for	2	futurePersonagem	2	aguardar	3	films	2
results	1	futuresFilmes	3	return	2	Filme	2
return	3	futureFilme	2	comFilms	2	id	2
films	1	Filme	2	unificar	2	getFilmById	1
id	1	Integer	2	consumir	1	Imprimir Personagens ETitulosFilmes	1
getFilmById	1	idFilme	3				
add	2	join	2				
comFilms	1	Imprimir Personagens ETitulosFilmes	1				
stream	2						
map	2						
toList	2						
thenAccept	1						
join	1						
21	**47**	**16**	**34**	**13**	**32**	**13**	**26**

Cenário 3			
Vanilla		**FluentAsync**	
Vocabulário ($n = n1 + n2$)	37	**Vocabulário ($n = n1 + n2$)**	26
Tamanho ($N = N1 + N2$)	81	**Tamanho ($N = N1 + N2$)**	58
Volume ($V = N \log_2 n$)	421,9657226	**Volume ($V = N \log_2 n$)**	272,6255037
Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)	22,3125	**Dificuldade ($D = {n1}/{2} \times {N2}/{n2}$)**	13
Esforço ($E = D \times V$)	9415,110186	**Esforço ($E = D \times V$)**	3544,131547