Um Estudo Sobre a Utilização de Operadores de Bibliotecas Reativas em Projetos de Código Aberto

José Murilo Sodré da Mota Filho[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brazil

jmsmf@cin.ufpe.br

Abstract. *Programação Reativa é um paradigma de programação que dá ênfase ao fluxo de dados e a como esses dados mudam e são propagados. Visando permitir que linguagens de programação procedurais tradicionais consigam operar em cima de fluxos de dados, diversas ferramentas foram desenvolvidas e mantidas ao longo dos anos, algumas dessas através do projeto ReactiveX, desenvolvido inicialmente pela Microsoft, que administra ferramentas de código-aberto com intuito de prover as abstrações necessárias para variadas linguagens de programação.*

Este trabalho procura analisar alguns projetos de código aberto, utilizando técnicas de mineração de repositórios para extrair dados de utilização de projetos do ReactiveX, tais como RxJava, RxJS, RxKotlin e RxSwift, com o objetivo de tentar entender se, e como, seus operadores estão sendo utilizados na prática, bem como seus padrões de utilização. Para isso, são utilizadas e analisadas métricas como frequência de usos, médias, mediana, entre outros, disponibilizados posteriormente para auxiliar trabalhos futuros.

Keywords— Programação Reativa, ReactiveX, Mineração de Repositórios

1. Introdução

Diversas aplicações modernas são criadas de forma reativa, isto é, dependem de modificações de dados e valores, eventos que estão sendo observados para realizar suas operações, desde interfaces gráficas de aplicações *mobile*, que devem reagir aos toques de seus usuários, até sistemas bancários robustos, que, ao tentar transferir uma quantia, vão rodar as checagens necessárias que foram engatilhadas pelos seus criadores.

Com o crescimento do interesse em Linguagens de Programação Reativas, várias soluções foram desenvolvidas pelo mercado a fim de auxiliar linguagens procedurais a operarem de forma reativa, e dessa forma aumentar a adoção desse paradigma. São soluções como *ReactiveX, Akka, EventBus, Sodium, Flow*. Essas soluções visam dar ao programador controle sobre o fluxo de dados, já que o fluxo de controle em si se inverte em programas reativos [1]. Um exemplo do interesse do mercado em soluções que auxiliassem esse paradigma é a *ReactiveX*. Criada originalmente em 2011 por um time da *Microsoft* para trazer certas funcionalidades baseadas no paradigma reativo a soluções desenvolvidas através de seu *.NET Framework*[26].

Porém a adoção dessas soluções não é indolor. Segundo Salvaneschi et al. [32], devido à forma como eventos são tratados, pode ser difícil de se entender o fluxo de sistemas por completo. Além disso cada uma delas apresenta seus próprios obstáculos, tendo diferentes regras de implementação e de arquitetura, além de diferenças em suas comunidades de apoio.

Apesar de programação reativa aumentar consideravelmente o entendimento e compreensão do funcionamento de uma aplicação reativa quando comparado ao padrão *Observer*, como colocado por Salvaneschi et al. [33], também deve-se atentar a outro fator importante para a adoção de uma nova tecnologia: A sua facilidade de uso. Ainda segundo Salvaneschi et al. [33], para facilitar o uso e aprendizado desse paradigma seria recomendado manter essas ferramentas o menos especializadas possível, focando em funcionalidades fundamentais.

1.1. Motivação

Apesar desse crescimento de interesse e ofertas de ferramentas para programação reativa, poucos estudos têm sido executados em busca de dados reais sobre as suas vantagens [30]. Contudo, alguns estudos já constataram que a programação reativa ajuda na produção de programas mais intuitivos [31], ou com maior facilidade de leitura [33, 17].

Uma visita à documentação de uma dessas iniciativas de programação reativa mencionada previamente, o *ReactiveX*, mostra 74 operadores *core* e mais 380 operadores *specialty*, versões especializadas dos outros 74 operadores [19]. Informações como esta ajudaram a elaborar a pergunta: A programação reativa oferece uma solução que consegue ser simples e completa, evitando a proliferação de operadores sobre-especializados [33, 30]?

A resposta para tal pergunta poderia ajudar a entender como tem sido o uso dos operadores em uma das bibliotecas mais utilizadas pela comunidade reativa. Entender esse problema também ajudaria a estimular pesquisas futuras na área, as quais poderiam averiguar as dificuldades principais em relação ao uso dos diferentes operadores dessas linguagens.

1.2. Objetivos

Esse trabalho tem como objetivo estudar quantitativamente a utilização de operadores de bibliotecas ReactiveX, para assim tentar avançar estudos na área e prover dados que possam ser utilizados em trabalhos futuros. Essa questão sobre utilização de bibliotecas e soluções reativas foi por sua vez também levantada por Salvaneschi et al. [33, 30], o que auxiliou nesta decisão. Com isso, este estudo se propõe a responder o primeiro ponto levantado na Motivação, verificando o uso de operadores de linguagens reativas e suas variações em linguagens diferentes.

Esse estudo se dará por meio de extração de dados de alguns dos repositórios *opensource* com mais contribuições na plataforma de controle de versão colaborativa *GitHub* que se utilizam de bibliotecas reativas promovidas pelo projeto ReactiveX [18].

2. Fundamentação Teórica

Nesse capítulo, o objetivo será contextualizar as áreas de interesse que serão tocadas, que tipos de tecnologias foram utilizados, as bibliotecas reativas exploradas e todos os outros temas abordados por esse trabalho. A primeira parte fala sobre programação reativa, seu surgimento, sua importância, suas abstrações. A segunda parte fala rapidamente sobre *Mining Software Repository*, seus usos e exemplos de outros trabalhos da área.

2.1. Programação reativa

2.1.1. Contextualização

Vários sistemas modernos são reativos, isto é, precisam responder a eventos de interesse, computá-los, e reagir, ou não, a eles, possivelmente gerando novos eventos em si [32]. Um jeito de facilitar o desenvolvimento de *software* que funciona dessa forma é com a Programação Reativa. Sistemas desenvolvidos nesse paradigma conseguem reduzir o custo de implementação, tempo de

resposta e manutenção quando se comparado a paradigmais imperativos, além de conseguirem atingir tempos de resposta ordens de magnitude menores [4].

Uma tentativa de coordenar uma definição mais concreta do que são sistemas reativos em si foi o desenvolvimento do *The Reactive Manifesto*, O Manifesto Reativo. Esse manifesto define que as necessidades que precisam ser sanadas por um sistema reativo podem ser definidas através de quatro propriedades: Responsividade, Resiliência, Elasticidade e Orientação a Mensagens [4]. Mais especificamente:

- **Responsividade** – define que sistemas devem responder a eventos de forma ágil. É considerado pelo manifesto como o pilar da usabilidade e utilidade de um sistema reativo. Além disso, o manifesto define que responsividade é dependente da eficácia ao responder esses eventos. Definido dessa forma, o comportamento do sistema seria consistente o suficiente a ponto de simplificar tratamento de erros e de encorajar o usuário a interagir novamente com o sistema, desenvolvendo a confiança no mesmo.

- **Resiliência** – seria a capacidade do sistema de se manter responsivo ao enfrentar falhas. Isso seria alcançado através da replicabilidade do código, conceito assim definido pelo manifesto como a capacidade desse código ser executado concomitantemente, ou seja, o quanto auto-contido e isolado é esse código. Isso resulta em métodos bem definidos e desacoplados. Com essa definição, falhas conseguem ser contidas dentro de cada componente, evitando comprometer outras partes do sistema em caso de falha. A própria recuperação dessas falhas seriam delegadas a outros componentes, garantindo uma alta disponibilidade do sistema.

- **Elasticidade** – requer que os sistemas se mantenham responsivos independentemente de demanda e a carga imposta pela mesma. Dessa forma, sistemas reativos devem ter a capacidade de reagir a mudanças na demanda aumentando ou diminuindo recursos alocados aos serviços necessários. Essa elasticidade evitaria a criação de gargalos ou pontos de contenção, remetendo de novo à responsabilidade distribuída mencionada nas propriedades anteriores.

- **Orientação a Mensagens** – é a ideia de que sistemas reativos dependem da comunicação assíncrona entre componentes, garantindo baixo índice de acoplamento. Isso prevê a capacidade de delegar as falhas como um tipo de mensagem em si. Essa orientação a mensagem permite que as outras propriedades, como elasticidade e resiliência, possam ser implementadas com sucesso.

2.1.2. ReactiveX

ReactiveX, ou *Reactive Extensions* como também é conhecido, se auto-intitula como uma biblioteca destinada à composição de programas assíncronos baseados em eventos, que se utiliza de sequências de *observables* seguindo e extendendo sobre o padrão de programação de *Observer* [19], podendo assim ser considerado uma implementação de Programação Reativa.

Esses objetos *observables* terão seus valores observados ao longo da execução do programa, permitindo que o programador trate eventos assíncronos com a mesma simplicidades, facilidade e composição operacional com a qual tradicionalmente se trata coleções de dados como *arrays*.

Essa biblioteca foi implementada em diversas linguagens, como em *Scala*, o RxScala, e em *Java*, o RxJava. Suas implementações são baseadas em coleções de operadores pré-definidos na documentação encontrada no site do ReactiveX, sendo assim uma implementação "poliglota"[19]. Devido a essa implementação padronizada em várias linguagens de programação diferentes, essas ferramentas do ReactiveX são perfeitas para o estudo quantitativo proposto por esse trabalho.

2.1.3. Operadores do ReactiveX

Operadores, ou *Operators*, é o mecanismo principal para definir como sequências de eventos devem acontecer durante a execução do código criado com uma biblioteca *ReactiveX*. A adição de um operador sobre um *Observable* faz com que alguma operação seja executada sobre este *Observable*, geralmente retornando um novo *Observable* com as modificações desejadas de acordo com o operador e os parâmetros utilizados.

Um exemplo de utilização de um operador comumente utilizado na criação de aplicações reativas, *filter*, pode ser encontrado na Figura 1; nessa figura, *filter* é usado para filtrar elementos da *stream* (*Observable*) permitindo que "Apenas elementos maiores que 10"sejam emitidos no novo observable. Outro exemplo de operador, *map*, é demonstrado no diagrama da Figura 2; nesse diagrama, elementos do *Observable* são mapeados para valores "10 vezes os valores emitidos pela stream".

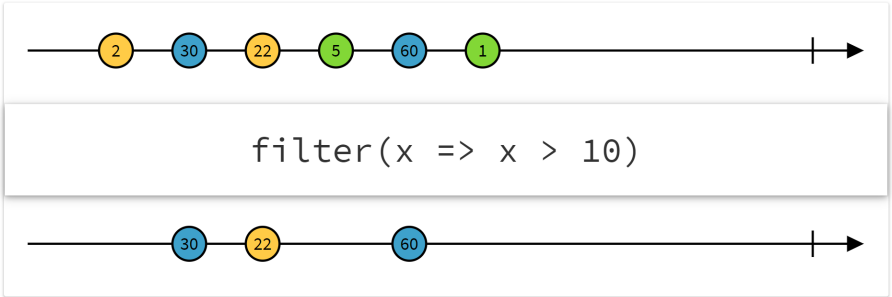

Figura 1. Diagrama representando a Execução do Operador *filter*.

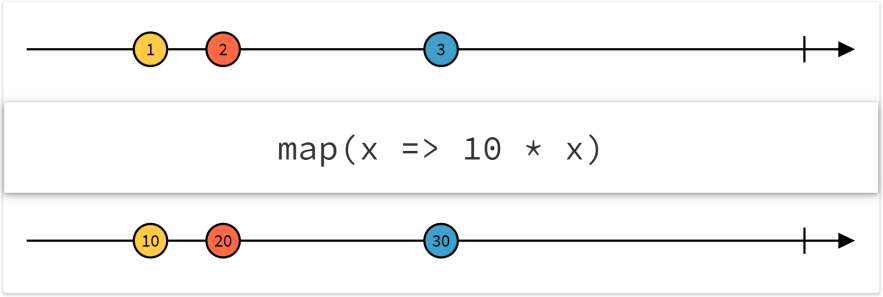

Figura 2. Diagrama representando a Execução do Operador *map*.

2.2. Mineração de Repositórios de Software

2.2.1. Contextualização

Mining Software Repository, ou MSR, é o campo de pesquisa que envolve analisar os dados que são gerados pela própria produção de código de programação, minerando esses dados diretamente dos repositórios onde esses códigos são compartilhados [25]. Esses dados recuperados de repositórios estão em condição de serem utilizados para descobrir padrões e informações interessantes, que podem vir a ser estudadas através de investigações empíricas, trazendo conclusões sobre o processo de desenvolvimento de software [14]. De acordo com Herzig e Zeller [13], esses dados minerados seriam inerentemente sem viés.

Várias ferramentas foram desenvolvidas nessa área para auxiliar a parte mais mecânica da tarefa, abstraindo a complexidade que pode surgir em projetos que analisam a mudança de código de acordo com o tempo. Uma dessas ferramentas é a *PyDriller* [35], por exemplo, que possui funções dedicadas a análise de projetos desenvolvidos através da ferramente de controle de versão *Git*.

2.2.2. Mineração no Github

O Github, por ser uma plataforma de colaboração de códigos abertos e que inclui aspectos sociais em suas interações, se vê como uma fonte de dados atrativa para diversos projetos de MSR [15]. Projetos passados já se utilizaram de dados dessa plataforma para promover pesquisas, que vão desde mineração de dados relacionados ao uso de funções de alta ordem em programas construídos usando *Scala* [38], até estudos qualitativos sobre as interações entre programadores diferentes ao colaborarem na plataforma [15].

O GitHub é uma plataforma de enormes proporções quando se trata do número de projetos hospedados, se aproximando de 38 milhões no começo de agosto de 2021, e igualmente enorme número de usuários, 63 milhões [9]. Por causa dessa magnitude de projetos existentes na plataforma, costuma ser necessário definir algum tipo de filtro a ponto de viabilizar essa mineração. Um filtro comum costuma ser o número de ”estrelas”que um projeto possui [38], métrica geralmente vista como popularidade de um repositório na plataforma. Entretanto, alguns trabalhos chamam atenção para como essa métrica pode ser mal interpretada e enviesar os dados caso mal utilizada [5, 15].

Em Se tratando do método de mineração dos dados, vários caminhos podem ser tomados, desde a própria API do GitHub [8], a qual, com limitações, permite buscar sobre todos os repositórios, bem como pesquisar dados sobre repositórios específicos, tendo como sua principal limitação o número baixo de requisições permitidas: apenas 5000 requisições por hora na API REST e apenas 30 na API de buscas dentro da plataforma, e essas requisições retornam no máximo 1.000 resultados, cortando os resultados usando um algoritmo de ranqueamento próprio do GitHub se nenhuma ordenação for especificada. Outras opções populares de se obter dados de repositórios hospedados no GitHub são o GHTorrent e o GH Archive (originalmente GitHub Archive), iniciativas não oficiais que oferecem acesso alternativo a dados de do GitHub [22].

O GHTorrent é um projeto que tenta funcionar como um espelho offline dos dados obtidos através da API REST padrão do GitHub [11], se utilizando de chaves de API que são ”doadas”por usuários que queiram contribuir com a causa, e monitorando eventos de repositórios públicos que são anunciados pelo próprio GitHub. Esses dados são armazenados em um banco de dados não-estruturados MongoDB, e depois as estruturas são extraídas para um banco de dados estrutu-

rado em MySQL. Apesar de não oficial, esse projeto é patrocinado pela Microsoft Corporation, empresa-mãe do GitHub após aquisição em 2018.

Já o GitHub Archive, recentemente renomeado como GH Archive pra evitar confusões com o projeto de arquivamento oficial do GitHub, segue uma via similar ao do GHTorrent, minerando e armazenando eventos públicos através da própria API do GitHub [12]. Esses dados são disponibilizados por versões que são atualizadas a cada hora do dia. Uma possível vantagem é que os dados são disponibilizados também através de uma base de dados hospedada no Google Big Query, permitindo acesso aos dados através de linguagem baseada em SQL diretamente online, sem necessidade de se baixar nada, tendo como única limitação as tradicionais da própria plataforma da Google.

Essas iniciativas permitem acessar vários dados do GitHub, mas sem se preocupar com a limitação do número de requisições existente na API oficial, facilitando pesquisas mais complexas que necessitam de dados históricos da interação de usuários dentro da plataforma.

2.3. APIs Robustas

2.3.1. Contextualização

APIs, ou *Application Programming Interfaces*, são interfaces para funcionalidades previamente implementadas, provendo abstrações de alto nível e tornando mais acessível o reuso de funcionalidades entre sistemas [28].

Segundo Reddy [27] APIs podem ser definidas como a melhor forma de conseguir desenvolver sistemas robustos, estáveis e duráveis. Isso acontece porque APIs escondem complexidades de funcionalidades que já foram resolvidas evitando retrabalho por parte de desenvolvedores em geral [34]. Vários termos utilizados em estudos de Engenharia de Software implicam em APIs bem desenhadas, como Modularidade, Reusabilidade, Computação Distribuída, dentre outros [27].

2.3.2. Necessidade de APIs Robustas

APIs robustas, elegantes e bem desenvolvidas podem ser consideradas parte crítica de grandes projetos. Bloch definiu em [2] a importância de APIs bem desenhadas, explicitando como elas podem ser uma das propriedades mais importantes de uma empresa, devido a quanto tempo e recurso acabam sendo investidos em seu aprendizado e sua aplicação por outras empresas e desenvolvedores, mas ao mesmo tempo pode ser um dos maiores pesadelos de uma empresa, resultando em custos elevados de suporte técnico e manutenção.

Outro problema explicitado por Scaffidi [34] é que APIs, especialmente quando muito especializadas, podem facilmente ficar obsoletas. O *status quo* em desenvolvimento de software vive mudando e evoluindo, alterando as necessidades possíveis de desenvolvedores que fazem utilização de uma API.

Segundo Bloch [2], as características de uma boa API são, dentre outras: facilidade de uso, facilidade de aprendizado, apropriado à audiência que vai utilizá-la [2, 34]. É levantado também por Robillard [28] como as dificuldades de se utilizar uma API podem acabar detraindo, ou até anulando, as vantagens normalmente associadas ao uso de APIs.

Alguns dos jeitos de se obter uma API apropriada, como citado previamente, seria fazer APIs com funcionalidade fáceis de se explicar, o que incentivaria o usuário a continuar seu uso [2], e fazer com que a API seja a menor possível, isto é, com a menor quantidade de classes e operadores, mas que que ainda atenda aos seus requerimentos [2, 27]. Uma vez que uma classe ou

método é inserido na API, sua remoção pode causar grandes dores aos seus usuários [2], além de gerar custos adicionais de manutenção futuros [27].

3. Metodologia

Neste capítulo será descrito o processo de coleta dos dados, bem como as decisões tomadas em torno desse processo. A plataforma de colaboração de repositórios com código aberto GitHub foi escolhida como a principal fonte dos dados a serem recuperados e explorados posteriormente nesse trabalho, devido à ampla disponibilidade de dados, além de já ter sido alvo de vários estudos diferentes [6]. A linguagem utilizada para processar os dados foi *Python* devido a familiaridade prévia com tal linguagem. Todo o código utilizado para realizar a mineração dos repositórios encontra-se disponível publicamente através do GitHub [23].

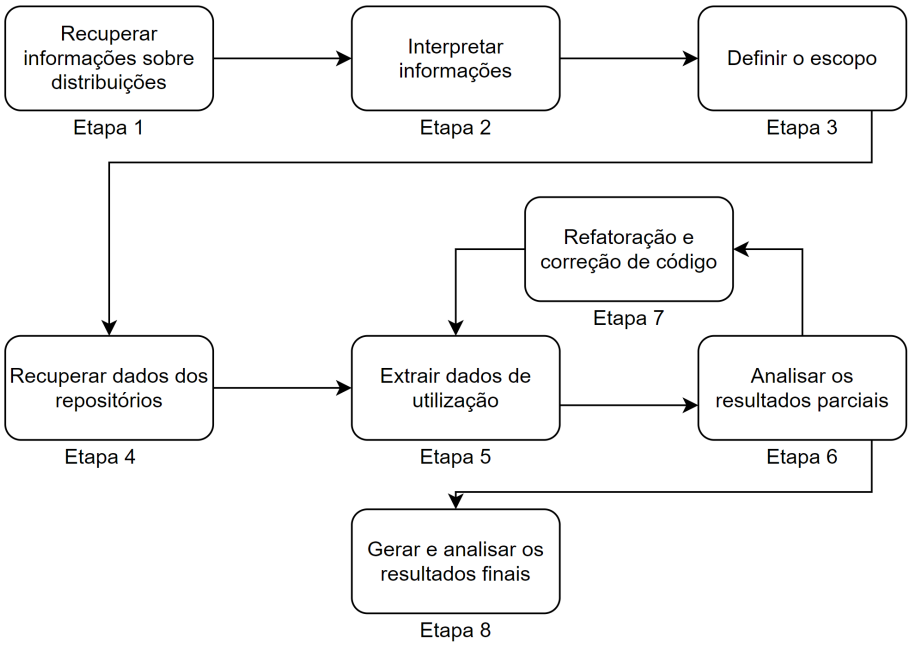

Figura 3. Diagrama representando o fluxo deste trabalho.

Apesar de diferir no objeto de estudo e em escopo, parte da metodologia foi baseada no estudo exploratório feito por Xu et al. [38], onde informações sobre repositórios foram recuperadas no GitHub, e depois analisadas. Essas informações foram usadas para definir o escopo do projeto e então os repositórios foram de fato minerados, primeiro se recuperando o código do repositório em si e depois analisando os repositórios e extraindo seus dados de utilização. Conclusões foram tiradas a partir da análise destes dados. Baseado nessa metodologia foi gerado o diagrama da Figura 3, o qual descreve o fluxo que esse trabalho seguiu.

3.1. Recuperando informações do Github

3.1.1. Escolha de ferramenta

Como discutido na **Seção 2.2.2**, várias ferramentas são disponibilizadas, cada uma com suas vantagens e desvantagens. Como a proposta desse trabalho gira em torno de projetos que ainda existem, e apenas na sua forma mais recente, especialmente devido a uma limitação de escopo e tempo baseada na pesquisa de Mombach et al. [22], foi definido que a API oficial do GitHub seria o suficiente para permitir esse estudo, visto que sua limitação no número de requisições se torna um fator menor quando dados históricos de um repositório não serão recuperados.

	RxJava	RxJS	RxSwift	RxKotlin	RxDart	UniRx[1]
Total	16.090	15.142	5.192	596	465	341
$\geq$ 15 Estrelas	1.179	561	297	60	56	27
0 Estrelas	10.669	11.701	3.595	343	257	190

	Rx.NET	RxPY	RxCpp	RxScala	RxGo	RxPHP
Total	130	96	78	54	52	39
$\geq$ 15 Estrelas	15	13	6	3	8	4
0 Estrelas	64	52	42	31	29	19

	RxGroovy	RxLua	RxClojure	RxRuby	reaxive[2]	RxJRuby
Total	10	6	5	4	1	1
$\geq$ 15 Estrelas	1	1	1	2	1	1
0 Estrelas	6	3	4	0	0	0

Tabela 1. Total de Repositórios no GitHub por Distribuição[3]

3.1.2. Informações Sobre as Distribuições

Após as primeiras tentativas de buscar repositórios hospedados no GitHub, como descrito na **Etapa 1** da Figura 3, foi possível ter a primeira visão concreta da dimensão do escopo. Das 18 bibliotecas oficialmente reconhecidas pelo ReactiveX, 2 não tinham presença em nenhum repositório público que não o da própria distribuição na plataforma: O *reaxive* e o *RxJRuby*. As distribuições com maior adoção em projetos públicos foram o *RxJava*, *RxJS*, *RxSwift*, *RxKotlin* e *RxDart*, todas linguagens de programação muito utilizadas para criação de clientes, tanto para desktop quanto para web ou mobile; isto é, aplicações onde o fluxo de controle costuma se inverter.

Os repositórios foram então recuperados através da API de busca do GitHub. Um problema inicial foi encontrado ao tentar recuperar os repositórios com apenas 0 estrelas nas distribuições mais populares. Estes tiveram mais de 1000 resultados, o que é uma das limitações da API oficial referenciadas na API oficial e no trabalho de Mombach et al. [8, 22].

3.1.3. Definição do Escopo

Observando as informações sobre o total número de repositórios pra cada distribuição, que podem ser encontrados na Tabela 1, foram percebido algumas limitações, como, por exemplo, a diferença da ordem de magnitude entre os totais de repositórios encontrados por distribuição. Dessa forma, fica evidente a necessidade de definir limites, visto que como demonstrado por Borges et al. [5], existe uma correlação moderada entre o número de estrelas de um repositório e o seu número de contribuintes ou número de *forks*, e uma correlação leve com o número de *commits*.

Logo, na **Etapa 2**, foi definido então um limite mínimo de 15 estrelas em um repositório, para assim tentar criar uma base de comparação entre eles. Para evitar a sobre-representação das distribuições mais utilizadas, como *RxJava* e *RxJS*, foi definido que os 50 repositórios com mais estrelas seriam avaliados por distribuição, sendo esse número definido pelo menor total de repositórios com mais de 15 estrelas nas distribuições mais utilizadas.

[1]Extensão reativa Rx para o motor de jogo Unity
[2]Extensão de Rx para a linguagem de programação Elixir
[3]Atualizado em 15 de Agosto de 2021 de acordo com dados consultados em https://github.com/search

Esses filtros de amostragem de repositórios ajudam a evitar problemas mencionados por Cosentino et al. [6], em cujo estudo foram analisados 231 trabalhos que se utilizaram do GitHub como fonte de repositórios. Problemas como a escolha arbitrária de projetos, combinado à falta de transparência quanto a esses repositórios escolhidos pra ser estudados, são alguns dos principais problemas apontados.

Outro problema presente em estudos baseados no GitHub, como apontado por Cosentino et al. [6], é o baixo número de repositórios analisados, isto é, o número de pesquisas com amostragem menor que 100 repositórios. No caso, o ponto de corte de 15 estrelas estabelecido para este trabalho, combinado ao limite de 50 repositórios por distribuição, totalizaria em 250 repositórios, o que permite não inflar o escopo deste trabalho e ao mesmo tempo exceder linhas definidas por Xu et al. [38], finalizando assim as **Etapas 2** e **3**.

3.2. Extração dos dados

Na **Etapa 4**, para facilitar o processo de análise dos dados e para evitar vieses e limitações dos algoritmos de busca da API do GitHub descritos por Cosentino et al. [6] e por Kalliamvakou et al. [15], no lugar de se utilizar a API de busca do GitHub para extrair informações sobre o uso de expressões dentro de um repositório, os repositórios foram copiados localmente para análise. Esses repositórios foram separados em pastas de acordo com a distribuição da *ReactiveX* utilizada.

Além de ter um controle maior e mais direto sobre os dados, essa decisão também evitou problemas com o limite de requisições mencionado na Seção 2.2.2, limite este que seria facilmente exaurido, aumentando em muito o tempo de cada execução e dificultado a análise dos dados gerados. Dessa forma a **Etapa 4** foi finalizada.

3.2.1. Extração de Uso dos Operadores Repositórios

Por ser a distribuição de maior utilização, a **Etapa 5** dos testes de extração de uso dos operadores começou focando apenas em repositórios que se utilizavam de *RxJava*. Uma estratégia de análise textual foi utilizada, de forma a conseguir gerar um código que fosse agnóstico à linguagem de programação, ou no mínimo reutilizável o suficiente para ser usado nas extrações de utilização posteriores, as quais incluíram as demais distribuições.

Devido ao fato de *Java* ser estaticamente tipado, garantindo uma padronização maior nas chamadas de funções dos repositórios testados, e de ter uma das distribuições mais antigas e utilizadas do *ReactiveX*, tendo inclusive passado por revisões, não foram encontradas muitas dificuldades para varrer os arquivos e encontrar as utilizações dos operadores listadas na documentação oficial do *RxJava* e no site do *ReactiveX* [19].

3.2.2. Dificuldades Iniciais

Na **Etapa 6**, descrita na Figura 3, foi feita uma análise inicial dos resultados. Devido à ausência de estudos feitos nessa exata área, esse processo se deu de forma mais manual. Foi encontrado um problema inicial na busca por arquivos de código de programação dentro dos repositórios, e devido a isto, a contagem inicial de utilização de operadores de *RxJava* estava inflacionada.

Prevendo tal possibilidade, a **Etapa 7** foi adicionada ao diagrama de fluxo do trabalho, pois apesar de não ter sido citada em Xu et al. [38], esta etapa se mostrou essencial e foi revisitada mais vezes de formas menores posteriormente.

Distribuições	RxJava	RxJS	RxKotlin	RxSwift	RxDart
Nº de Operadores	153	179	144	66	59

Tabela 2. Total Operadores Distribuição

Após correções iniciais, novas etapas de extração e análise de resultados parciais foram executadas de forma a encontrar *bugs* e exceções aos casos comuns, tentando aumentar a robustez do código.

Uma vez que a análise inicial com *RxJava* se mostrou coerente, foi criado então um índice apontando o quanto um operador foi utilizado em cada repositório. A partir deste índice foram combinadas as informações extraídas gerando as seguintes métricas: frequência de usos, média de usos, mediana de usos e presença de operadores, este último conceito definido na Seção 4.3.

Após verificação de que a estrutura estava sendo gerada corretamente, a **Etapa 5** foi retomada e a extração foi expandida a todos os repositórios de *RxJava*, e aos repositórios de *RxJS*, *RxKotlin*, *RxSwift* e *RxDart*.

Uma nova **Etapa 6** foi executada, e as distribuições que fazem uso de *RxDart* apresentaram indícios de novos problemas. Uma análise manual mostrou que a distribuição de *RxDart* tem um número reduzido de operadores e que seus operadores mais utilizados diferem dos mais comuns encontrados em *RxJava*, como pode ser visto na Tabela 2.

3.2.3. Dificuldades Posteriores Com RxJS

Em novas etapas de análise de resultados parciais **Etapa 6**, foi dada atenção a um problema de contagem dos operadores utilizados em *RxJS*.

Nas outras distribuições, os operadores, funções membro da classe *Observable* que retornam um *Observable* em si, costumam ser encadeados de uma forma mais tradicional, em sequência, como pode ser visto na Figura 4. Já em versões mais novas de *RxJS* foi introduzido um novo padrão de utilização dos operadores. Eles seriam utilizados através da função *pipe*.

Essa mudança foi introduzida como o novo padrão a partir da versão 5.5 dessa distribuição [21], logo alguns projetos mais recentes fazem uso exclusivo da biblioteca *RxJS* através desse novo padrão. Com isso a extração de utilização de operadores de *RxJS* precisou ser alterada para levar em consideração projetos que se utilizaram de versões mais recentes.

A introdução da função *pipe* se deu através do argumento de que esse novo método garante uma melhor legibilidade do código, assim como uma maior composibilidade funcional [21], de forma que operadores possam ser customizados com maior facilidade. Este método também ajuda a evitar poluir a classe Observable, separando operadores que não são funções essenciais obrigatórias da classe para outros módulos, o que facilita o trabalho de ferramentas de produção de código de *JavaScript* que removem partes de código importado que não foram utilizadas, removendo dependências ”cegas”. Tal prática é conhecida como ”Tree shaking”[24].

Um exemplo de como operadores são utilizados através da função *pipe* pode ser observado na Figura 5. Dessa forma, os usos de operadores da distribuição *RxJS* não estavam sendo contabilizados devidamente.

Para acomodar essa diferença em como os operadores da distribuição *RxJS* são utilizados, foi necessária uma abordagem mista. Uma nova **Etapa 5** foi executada, com subsequentes **Etapas 6** e **7** até que eventualmente em uma análise reduzida os resultados fossem fidedignos a ponto de

```
public Observable<FakeToken> getFakeToken(@NonNull String fakeAuth) {
    return Observable.just(fakeAuth)
            .map(new Function<String, FakeToken>() {
                @Override
                public FakeToken apply(String fakeAuth) {
                    // Add some random delay to mock the network delay
                    int fakeNetworkTimeCost = random.nextInt(500) + 500;
                    try {
                        Thread.sleep(fakeNetworkTimeCost);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    FakeToken fakeToken = new FakeToken();
                    fakeToken.token = createToken();
                    return fakeToken;
                }
            });
}
```

Figura 4. Um exemplo do uso do operador *map* em RxJava

dar seguimento para a **Etapa 8**.

```
import { ajax } from 'rxjs/ajax';

const fetchUserEpic = (action$, state$) => action$.pipe(
  ofType('FETCH_USER'),
  mergeMap(({ payload }) => ajax.getJSON(`/api/users/${payload}`).pipe(
    map(response => ({
      type: 'FETCH_USER_FULFILLED',
      payload: response
    }))
  )
);
```

Figura 5. Um exemplo do uso do operador *map* em RxJS

3.2.4. Dificuldades Posteriores com RxDart

Durante a primeira revisão a fim de gerar resultados finais, foi percebido um problema com a biblioteca *RxDart*. Pelo menos 29 dos 56 repositórios com mais de 15 estrelas referenciados na Tabela 1 são repositórios que podem ser separados em grupos com utilização bastante similar de operadores, tanto na frequência quanto na utilização de cada operador individualmente.

Após uma análise manual, foi descoberto um fio em comum nesses repositórios menci-

Distribuições	RxJava	RxJS	RxKotlin	RxSwift	Total	Distintos
Nº de Operadores	155	221	144	66	586	342

Tabela 3. Total Operadores Distribuição Atualizado

onados. Eles são frutos de alguns tutoriais para *Flutter*, um framework de *RxDart*[3]. Todos apresentam código similar e compartilham alguns arquivos de configuração.

Esses repositórios foram subsequentemente removidos da pesquisa, visto que a sua presença representaria de forma não fidedigna o uso da biblioteca *Rx* no mundo real. Essa remoção causou a distribuição *RxDart* a ter seu número de repositórios abaixo dos limites definidos para a pesquisa, o que levaria a uma amostragem pequena de repositórios, um dos erros comuns apontados por Cosentino et al. [6]. Devido a este baixo número de repositórios restantes, optou-se por remover *RxDart* das distribuições analisadas.

Problemas similares também foram observados em alguns projetos que fazem uso de *RxKotlin*, mas em escala menor. Com o corte de *RxDart*, a decisão de se aumentar o número de repositórios analisados nas outras distribuições para 60 repositórios por distribuição foi tomada. Isso totalizaria em 240 repositórios de código aberto analisados por este trabalho e o manteria com um nível de amostragem considerada relevante, baseado na análise de Kalliamvakou et al. [15].

Com essa remoção, o número de operadores também foi alterado. Uma análise manual durante a remoção dos operadores de *RxDart* levantou suspeitas de inconsistências entre os operadores de *RxJS*. Ao ser investigado, foram percebidas diferenças na documentação do site oficial da *ReactiveX*[19], na documentação do site da distribuição *RxJS*[20], e também nos operadores que podem ser encontrados no repositório oficial da *RxJS*[29].

Todos os operadores distintos encontrados das 3 formas foram contabilizados e o mesmo processo de análise foi feito nas outras distribuições. Uma lista atualizada destes operadores pode ser vista na Tabela 3

4. Resultados

Neste capítulo serão apresentado os resultados obtidos através da coleta de dados descrita no capítulo anterior, junto com as estatísticas geradas através dos mesmos. Um resumo dos parâmetros usados para esta pesquisa pode ser encontrado na Tabela 7 no Apêndice A.

4.1. Operadores

Do total absoluto de 584 operadores encontrados nas diversas documentações e utilizados na construção deste trabalho, são 342 operadores distintos. Como pode ser observado na Figura 6, destes operadores distintos, 119 não foram utilizados em nenhum dos 60 repositórios com mais estrelas de nenhuma das distribuições de *Rx* abordadas.

Este foi o primeiro indício real de operadores sobre-especializados encontrado ao longo deste trabalho. Na Figura 7 pode ser observado como esses dados são distribuídos nas diferentes bibliotecas *Rx*, somando 125 operadores não utilizados do total de 586 operadores contabilizados quando se inclui redundâncias entre distribuições.

4.2. Frequência de Utilização

A frequência de utilização é o número absoluto de utilizações em um dado repositório ou grupo de repositórios. O gráfico de frequência acumulada de utilização pode ser encontrado na

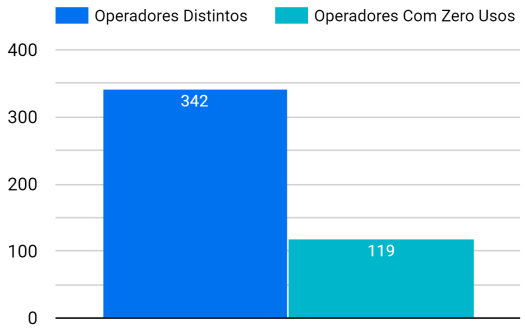

Figura 6. Operadores Distintos e Operadores Não Utilizados

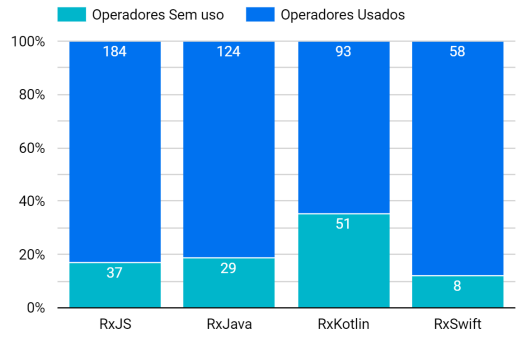

Figura 7. Operadores Utilizados e Não Utilizados

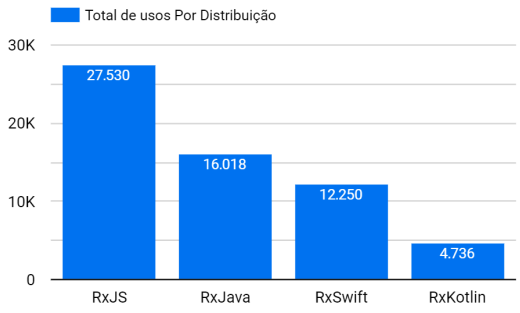

Figura 8. Frequência Acumulada de Utilização Por Distribuição

Figura 8. Ele mostra o quanto os operadores foram usados nos repositórios recuperados, separados por distribuição *Rx*, que juntos somam 60.534 utilizações de operadores.

Como pode ser observado, esse gráfico indica que *RxJS* tem um número de frequência de utilização de operadores nos repositórios recuperados muito maior do que as outras distribuição, chegando a quase 6 vezes o número de utilizações de operadores do que *RxKotlin* (27.530 usos em *RxJS* comparado a 4.736 usos de *RxKotlin*).

Isto provavelmente deve-se ao fato dos projetos de *JavaScript* possuírem uma tendência maior a tratarem de eventos, tanto em clientes quanto em servidores, se comparados a linguagens como Java. Existe também o fato de *RxJS* ser uma distribuição mais antiga, se comparada aos projetos de *RxKotlin* ou *RxSwift*. Uma investigação mais profunda dedicada a esse fenômeno poderia ser interessante para averiguar a real razão.

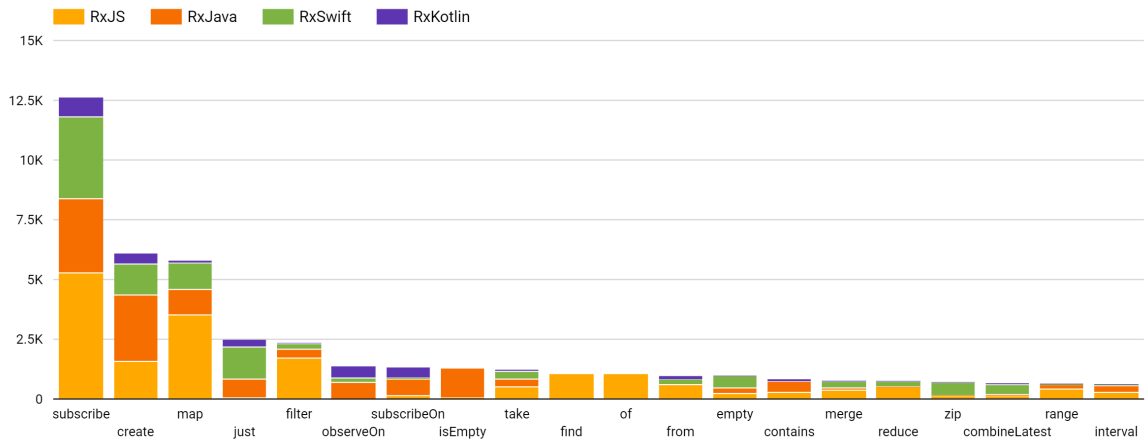

Figura 9. Frequência de utilização Por Distribuição, Por Operador

4.2.1. Frequência de Utilização Por Operador

Quando as utilizações são separadas por operadores (Figura 9), é possível perceber que certos operadores possuem uso consideravelmente maior em certas distribuições. O operador *map* teve mais de 50% de sua utilização na distribuição *RxJS* por exemplo. Já o operador *just* teve mais 50% de seus usos na distribuição *RxSwift*.

Se visualizarmos esses mesmos dados num gráfico de setores como na Figura 10, é possível ver que apenas 6 operadores compreendem mais de 50% da utilização de todos os operadores.

Quando expandimos a visualização de setores para todas as distribuições, é possível ver que 6 é na verdade o número máximo de operadores necessário para compreender pelo menos 50% de todos os usos das 4 distribuições *Rx* estudadas, como pode ser visto mais detalhadamente na Figura 44 no Apêndice G.

4.3. Presença

O conceito ”Presença”foi definido neste trabalho como o número de repositórios em que um devido operador se encontra presente, independente do número de utilizações no mesmo repositório. O gráfico de presença de operadores em repositórios por distribuição, encontrado na Figura 11, traz uma informação interessante: Nenhum operador aparece em todos os repositórios

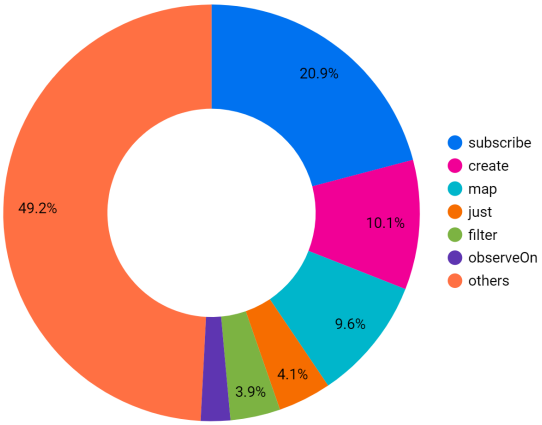

Figura 10. Operadores Mais Usados em Todas as Distribuições

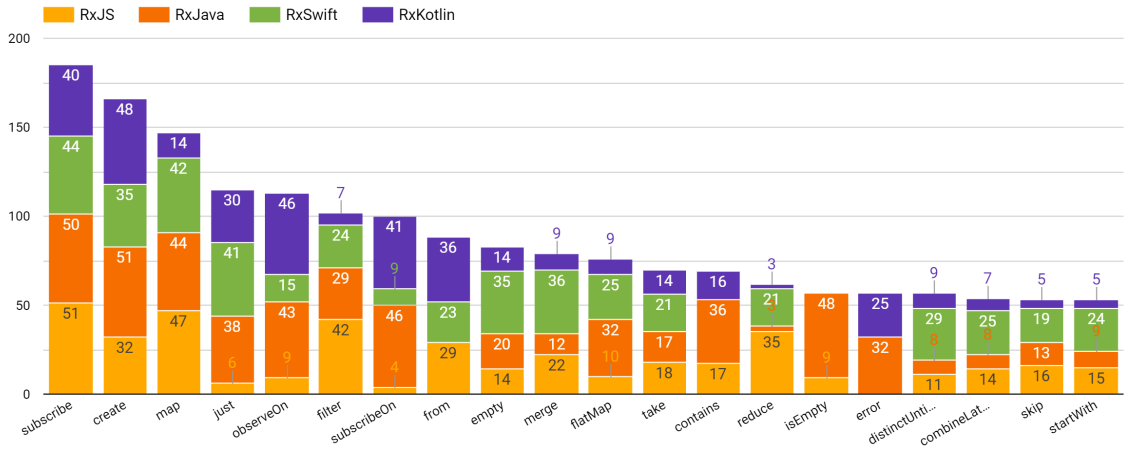

Figura 11. Presença de Operadores em Repositório Por Distribuição

de nenhuma distribuição. Os operadores com maior presença por distribuição foram: *subscribe* em *RxJS* e *RxSwift*, e *create* em *RxJava* e *RxKotlin*, totalizando 85%, 73%, 85% e 80% respectivamente.

Apesar de nenhum operador ter demonstrado cobertura de 100% nos repositórios em nenhuma distribuição, ainda mostra-se possível afirmar que existe uma boa cobertura da API para casos comuns, devido à tamanha presença nos operadores mais comumente utilizados nos repositórios estudados. Uma boa cobertura é algo recomendado como boa prática no desenho e planejamento de APIs, como mencionado na Seção 2.3.

Mais informações sobre presença, incluindo separação por distribuição *Rx*, podem ser encontradas no Apêndice C.

4.4. Média da Frequência de Utilização

A visualização gerada através dos dados de utilização média pode ser vista na Figura 12. Nela é possível ver uma forte semelhança com o gráfico de uso total, não trazendo muitas informações novas além de reafirmar as observações feitas previamente.

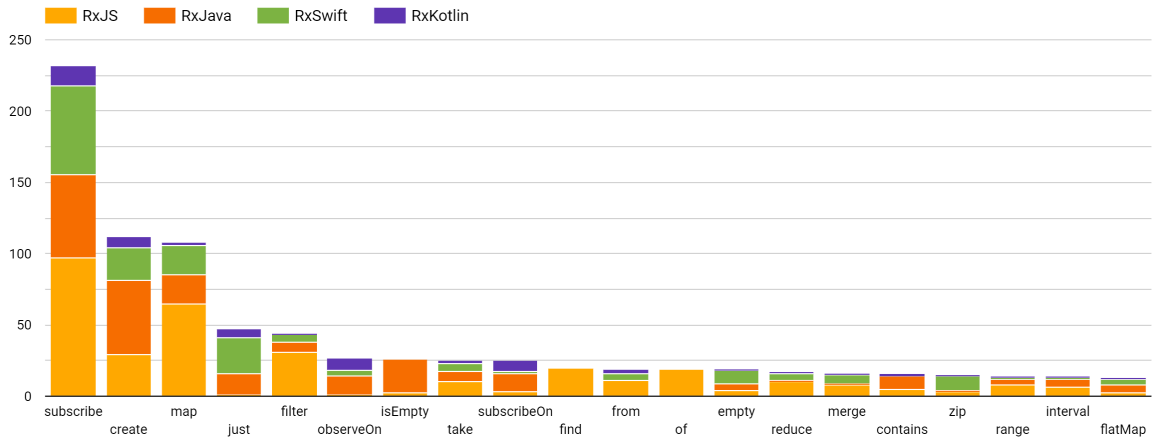

Figura 12. Média de Frequência de Utilização de Operadores Por Distribuição

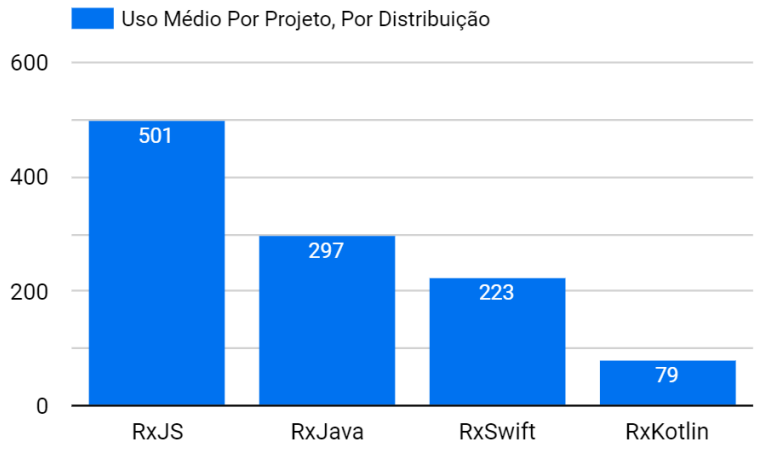

Figura 13. Média de Frequência de Utilização de Operadores Por Repositório, Por Distribuição

Na Figura 13 podemos ver o utilização média de operadores por repositório, por distribuição *Rx*. No gráfico é possível ver que mesmo com um número duas vezes maior de operadores, *RxJava* possui um uso médio de operadores por projeto similar a *RxSwift*.

Gráficos separados por distribuição contendo mais informações sobre média da frequência de utilização de operadores pode ser encontrado no Apêndice D.

4.5. Mediana

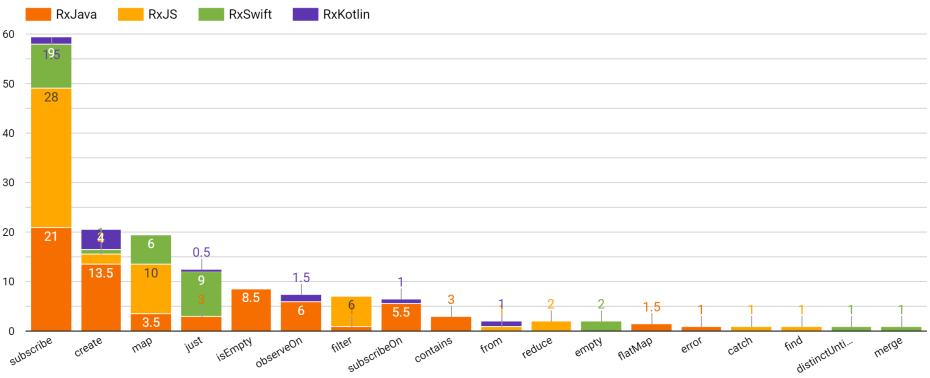

Figura 14. Mediana do Uso de Operadores, Por Distribuição

O conceito ”Mediana”consiste no número que separa as amostras pela metade, sendo uma metade composta de amostras com valores superiores ou iguais a mediana, e a outra com valores inferiores ou iguais à mediana. Na Figura 14 pode ser encontrado o gráfico com as medianas de uso de operadores por distribuição. Para esse gráfico foram selecionados apenas os operadores que tiveram mediana maior que 1 em pelo menos uma distribuição. O fato de só existirem 18 operadores entre todos os 342 operadores distintos investigados neste trabalho que possuem uma mediana maior que 0 mostra que há indícios de sobre-especialização na maior parte dos operadores das distribuições *Rx*.

Mais gráficos sobre medianas específicas dos operadores em cada uma de suas respectivas distribuições podem ser encontrados no Apêndice E.

5. Conclusão

Este trabalho teve como objetivo analisar e extrair dados sobre a utilização de operadores de linguagem reativas, mais especificamente das bibliotecas *Rx*. Isso foi alcançado através da extração desses dados em 240 repositórios de código aberto na plataforma GitHub. Com esses dados é possível afirmar que 119 operadores não foram utilizados por nenhum dos repositórios estudados, um total de 34.80% dos 342 operadores distintos extraídos das documentações das distribuições estudadas. É também possível afirmar através desses dados que uma parcela pequena dos operadores é responsável por mais de 50% da utilização das bibliotecas estudadas.

Com essas duas observações e revisitando a pergunta de pesquisa “A programação reativa oferece uma solução que consegue ser simples e completa, evitando a proliferação de operadores sobre-especializados?”, é possível concluir que, apesar das bibliotecas *Rx* possuírem indícios de boa generalização dos casos comuns, são fortes os indícios de que as mesmas possuem operadores altamente especializados com baixo, ou zero, nível de utilização, o que atua como um dificultador à adoção da tecnologia.

Esses fortes indícios de sobre-especialização presente nas distribuições estudadas são baseados nas métricas observadas neste estudo, como por exemplo o fato de que apenas 18 operadores tiveram mediana de utilização maior que 0, mostrando que os outros 205 operadores que tiveram algum uso não foram utilizados nenhuma vez em pelo menos 50% dos repositórios, o que, como foi apresentado neste trabalho, pode ser um obstáculo na adoção pelos usuários.

Os dados obtidos através desse estudo podem ser utilizados para guiar futuras implementações de bibliotecas reativas similares, inclusive novas bibliotecas *Rx* para novas linguagens de programação, ajudando a focar o esforço de implementação nos operadores mais utilizados. Além disso, este estudo também pode ajudar a nortear usuários que queiram aprender a utilizar essas bibliotecas, pois os mesmos saberiam quais operadores são essenciais para se entender e implementar a maior parte dos projetos reativos.

5.1. Trabalhos Futuros

Devido à restrição de tempo e escopo, não foi possível investigar um dos pontos levantados na Seção 1.1 sobre as principais dificuldades existentes quanto à adoção de linguagens e bibliotecas reativas. Esse estudo seria importante pois ajudaria a corroborar pontos levantados por esse trabalho sobre facilitadores e dificultadores de adoção de tecnologias reativas.

Se mostra relevante confrontar os dados obtidos neste trabalho com dados provenientes de outras fontes de projetos de códigos abertos, para assim confirmar ou não as conclusões obtidas. Se mostra relevante também realizar pesquisas similares em outras bibliotecas, APIs e linguagens de programação, para averiguar se os dados obtidos por esse trabalho são reproduzidos nessas outras ferramentas.

Referências

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4), August 2013. ISSN 0360-0300. doi: 10.1145/2501654.2501666. URL https://doi.org/10.1145/2501654.2501666. 1

[2] Joshua Bloch. How to design a good api and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 506–507, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593491X. doi: 10.1145/1176617.1176622. URL https://doi.org/10.1145/1176617.1176622. 2.3.2

[3] Didier Boelens. Reactive Programming - Streams - BLoC. https://www.didierboelens.com/2018/08/reactive-programming-streams-bloc/, August 2021. 3.2.4

[4] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The reactive manifesto. https://www.reactivemanifesto.org/, September 2014. 2.1.1

[5] Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, Dec 2018. ISSN 0164-1212. doi: 10.1016/j.jss.2018.09.016. URL http://dx.doi.org/10.1016/j.jss.2018.09.016. 2.2.2, 3.1.3

[6] Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from github: Methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 137–141, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901776. URL https://doi.org/10.1145/2901739.2901776. 3, 3.1.3, 3.2, 3.2.4

[7] C. J. Date and E. F. Codd. The relational and network approaches: Comparison of the application programming interfaces. SIGFIDET '74, page 83–113, New York, NY, USA, 1975. Association for Computing Machinery. ISBN 9781450374187. doi: 10.1145/800297.811532. URL https://doi.org/10.1145/800297.811532.

[8] GitHub, Inc. Github api. https://docs.github.com/en/rest, 2021. 2.2.2, 3.1.2

[9] GitHub, Inc. Github public repos and users. https://github.com/search, August 2021. 2.2.2

[10] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

[11] Georgios Gousios. The ghtorrent project. https://ghtorrent.org/, 2021. 2.2.2

[12] Ilya Grigorik. Gh archive. https://www.gharchive.org/, 2021. 2.2.2

[13] Kim Herzig and Andreas Zeller. *Mining Your Own Evidence*, chapter 27. O'Reilly Media, Inc., October 2010. ISBN 9780596808327. 2.2.1

[14] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, March 2007. ISSN 1532-060X. doi: 10.1002/smr.344. URL https://doi.org/10.1002/smr.344. 2.2.1

[15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 92–101, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597074. URL https://doi.org/10.1145/2597073.2597074. 2.2.2, 3.2, 3.2.4

[16] S.G. McLellan, A.W. Roesler, J.T. Tempest, and C.I. Spinuzzi. Building more usable apis. *IEEE Software*, 15(3):78–86, 1998. doi: 10.1109/52.676963.

[17] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1–20, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587660. doi: 10.1145/1640089.1640091. URL https://doi.org/10.1145/1640089.1640091. 1.1

[18] Microsoft Corporation. Reactivex languages. http://reactivex.io/languages.html, . visited on 2021-08-07. 1.2

[19] Microsoft Corporation. Reactivex operators. http://reactivex.io/documentation/operators.html, . 1.1, 2.1.2, 3.2.1, 3.2.4, A

[20] Microsoft Corporation. Rxjs - operators. https://rxjs.dev/guide/operators, . 3.2.4

[21] Microsoft Corporation. Pipeable operators. https://v6.rxjs.dev/guide/v6/pipeable-operators, August 2021. 3.2.3

[22] Thaís Mombach and Marco Tulio Valente. Github rest api vs ghtorrent vs github archive: A comparative study. 2018. 2.2.2, 3.1.1, 3.1.2

[23] José Mota Filho. Operators mining on github. https://github.com/jmsmf/github-mining-python. 3

[24] Mozilla Developer Network. Tree shaking. https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking, 2021. 3.2.3

[25] MSR Steering Committee. International conference on mining software repositories. http://www.msrconf.org/, 2020. 2.2.1

[26] Thomas Nield. *Learning RxJava*, chapter 2. Packt, June 2017. ISBN 9781787120426. 1

[27] Martin Reddy. *API Design for C++*. Elsevier Science, 2011. ISBN 9780123850041. URL https://books.google.com.br/books?id=IY29LylT85wC. 2.3.1, 2.3.2

[28] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009. doi: 10.1109/MS.2009.193. 2.3.1, 2.3.2

[29] RxJS. Rxjs on github. https://github.com/ReactiveX/rxjs. 3.2.4

[30] Guido Salvaneschi. What do we really know about data flow languages? PLATEAU 2016, page 30–31, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450346382. doi: 10.1145/3001878.3001884. URL https://doi.org/10.1145/3001878.3001884. 1.1, 1.2

[31] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 564–575, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635895. URL https://doi.org/10.1145/2635868.2635895. 1.1

[32] Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. Reactive programming: A walkthrough. 2:953–954, 2015. doi: 10.1109/ICSE.2015.303. 1, 2.1.1

[33] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, 2017. doi: 10.1109/TSE.2017.2655524. 1, 1.1, 1.2

[34] Christopher Scaffidi. Why are apis difficult to learn and use? *Crossroads*, 12:4, 08 2006. doi: 10.1145/1144359.1144363. 2.3.1, 2.3.2

[35] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3264598. URL http://dl.acm.org/citation.cfm?doid=3236024.3264598. 2.2.1

[36] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving api documentation using api usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126, 2009. doi: 10.1109/VLHCC.2009.5295283.

[37] Joshua Sushine, James D. Herbsleb, and Jonathan Aldrich. Searching the state space: A qualitative study of api protocol usability. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 82–93, 2015. doi: 10.1109/ICPC.2015.17.

[38] Yisen Xu, Fan Wu, Xiangyang Jia, Lingbo Li, and Jifeng Xuan. Mining the use of higher-order functions: An exploratory study on scala programs. *Empirical Software Engineering*, 25:1–38, 11 2020. doi: 10.1007/s10664-020-09842-7. 2.2.2, 3, 3.1.3, 3.2.2

A. Tabelas

	RxJava	RxJS	RxSwift	RxKotlin	RxDart	UniRx
Total	16.090	15.142	5.192	596	465	341
$\geq$ 15 Estrelas	1.179	561	297	60	56	27
0 Estrelas	10.669	11.701	3.595	343	257	190

	Rx.NET	RxPY	RxCpp	RxScala	RxGo	RxPHP
Total	130	96	78	54	52	39
$\geq$ 15 Estrelas	15	13	6	3	8	4
0 Estrelas	64	52	42	31	29	19

	RxGroovy	RxLua	RxClojure	RxRuby	reaxive	RxJRuby
Total	10	6	5	4	1	1
$\geq$ 15 Estrelas	1	1	1	2	1	1
0 Estrelas	6	3	4	0	0	0

Tabela 4. Total de Repositórios no GitHub por Distribuição[1]

Distribuições	RxJava	RxJS	RxKotlin	RxSwift	RxDart
Nº de Operadores	153	179	144	66	59

Tabela 5. Total Operadores Distribuição

Distribuições	RxJava	RxJS	RxKotlin	RxSwift	Total	Distintos
Nº de Operadores	155	221	144	66	586	342

Tabela 6. Total Operadores Distribuição Atualizado

[1] Atualizado em 15 de Agosto de 2021 de acordo com dados consultados em https://github.com/search

Parâmetro	Valor
Distribuições	RxJS, RxJava, RxSwift, RxKotlin
Mínimo de Estrelas por Repositório	15
Número de Repositórios por Distribuição	60
Data de Recuperação dos Repositórios	08/08/2021
Informações Sobre Operadores	Site oficial da *ReactiveX*[19]

Tabela 7. Parâmetros Usados Neste Trabalho

B. Gráficos de Utilização de Operadores Por Distribuição

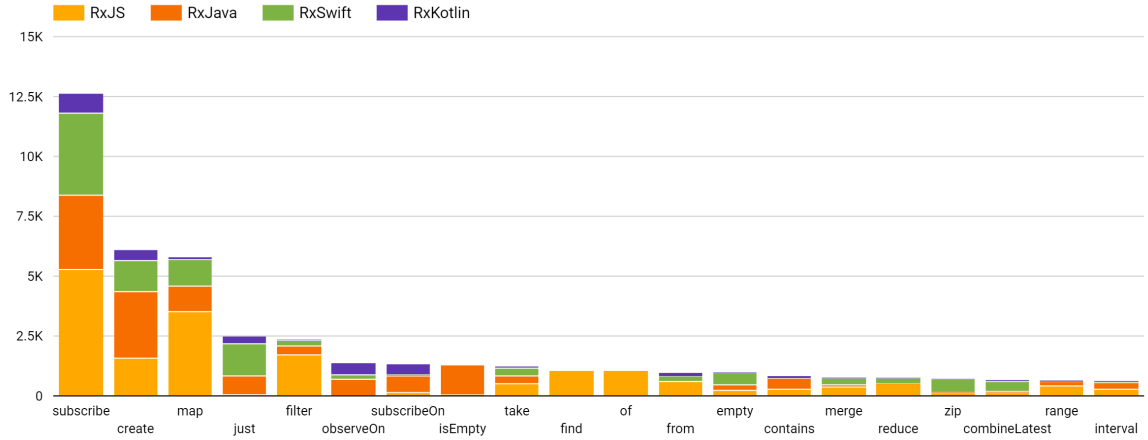

Figura 15. Utilização de Operadores por Distribuição

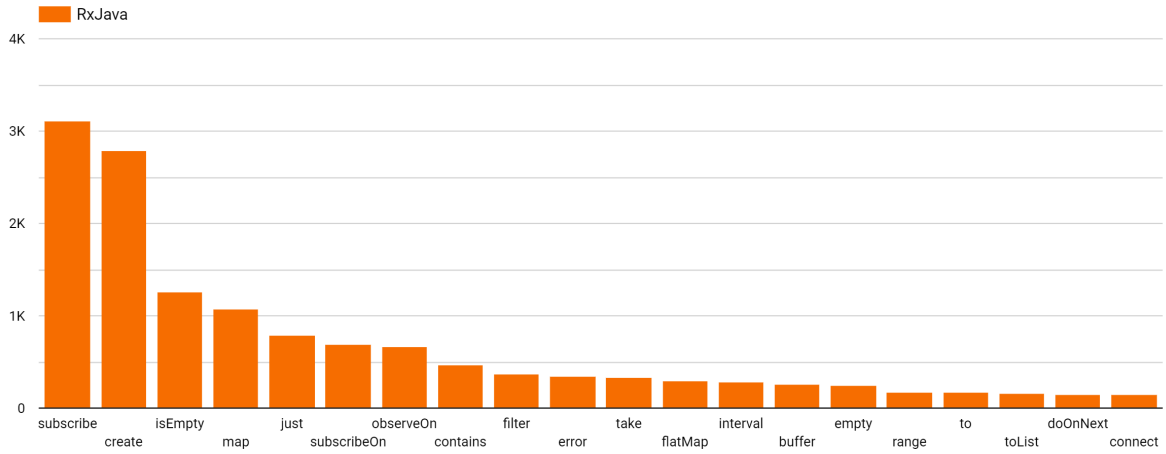

Figura 16. Utilização de Operadores em RxJava

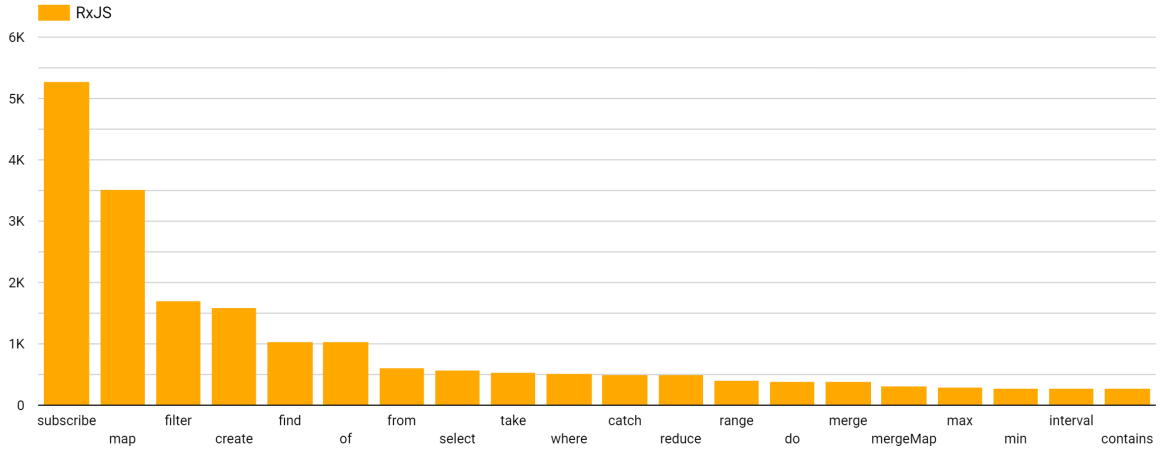

Figura 17. Utilização de Operadores em RxJS

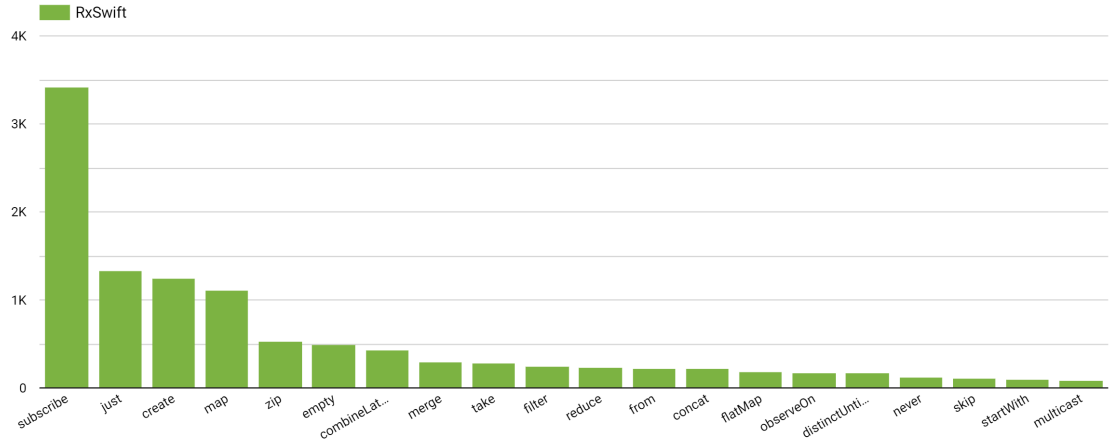

Figura 18. Utilização de Operadores em RxSwift

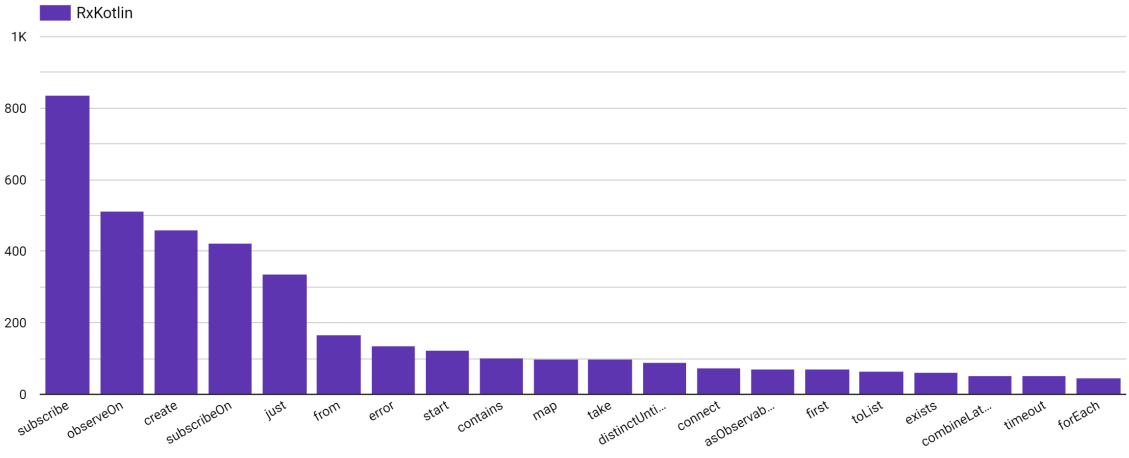

Figura 19. Utilização de Operadores em RxKotlin

C. Gráficos de Presença de Operadores Por Distribuição

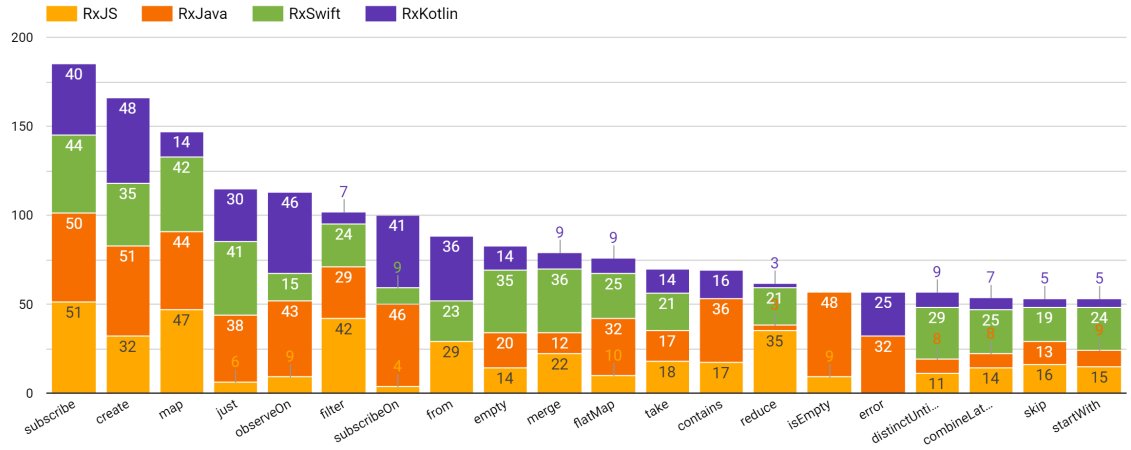

Figura 20. Presença de Operadores por Distribuição

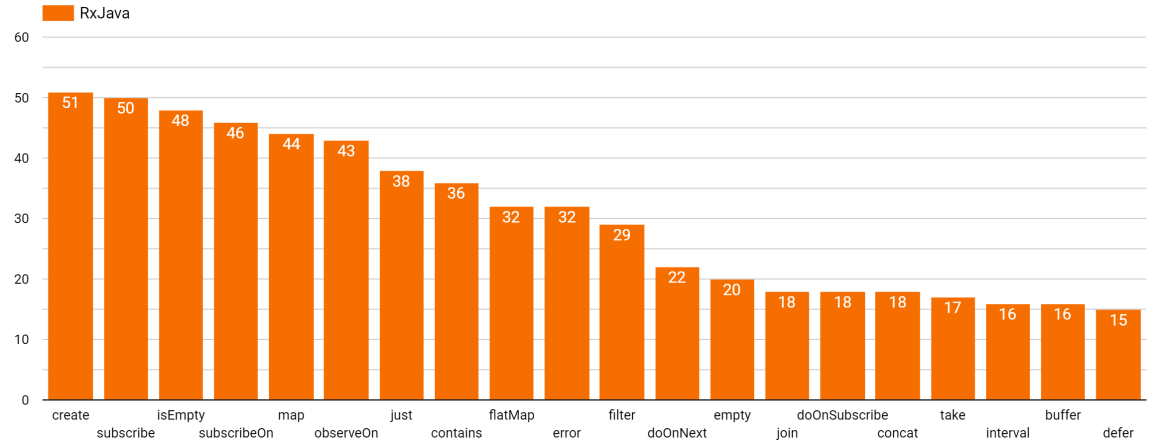

Figura 21. Presença de Operadores em RxJava

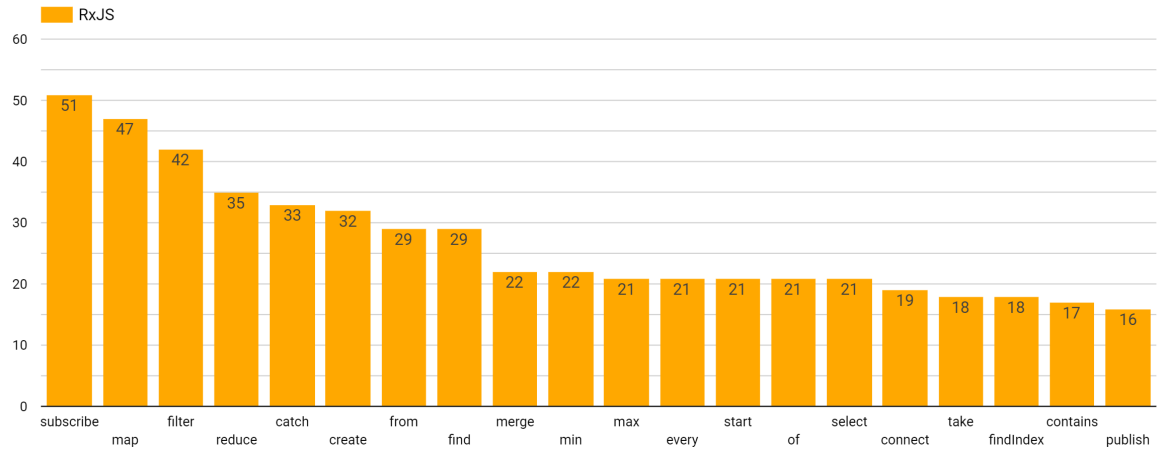

Figura 22. Presença de Operadores em RxJS

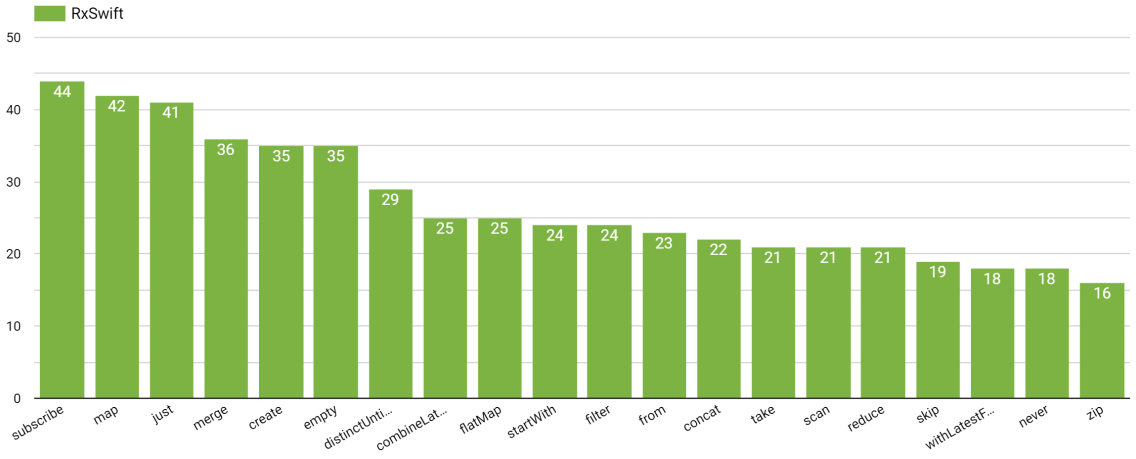

Figura 23. Presença de Operadores em RxSwift

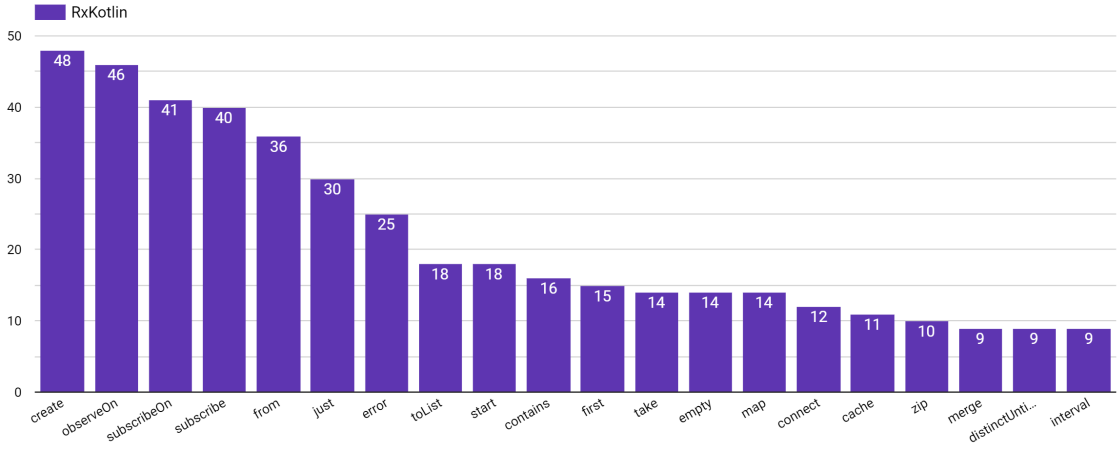

Figura 24. Presença de Operadores em RxKotlin

D. Gráficos de Uso Médio de Operadores Por Distribuição

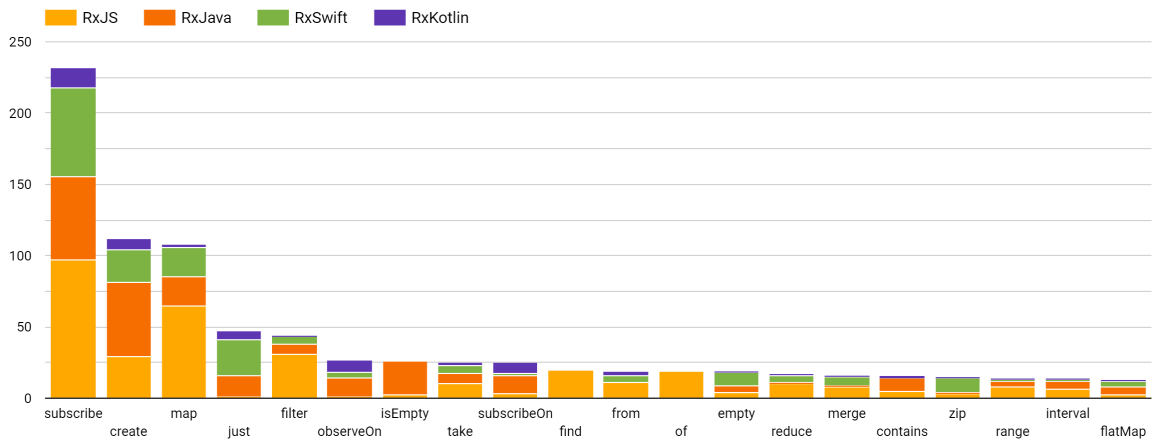

Figura 25. Uso Médio de Operadores por Distribuição

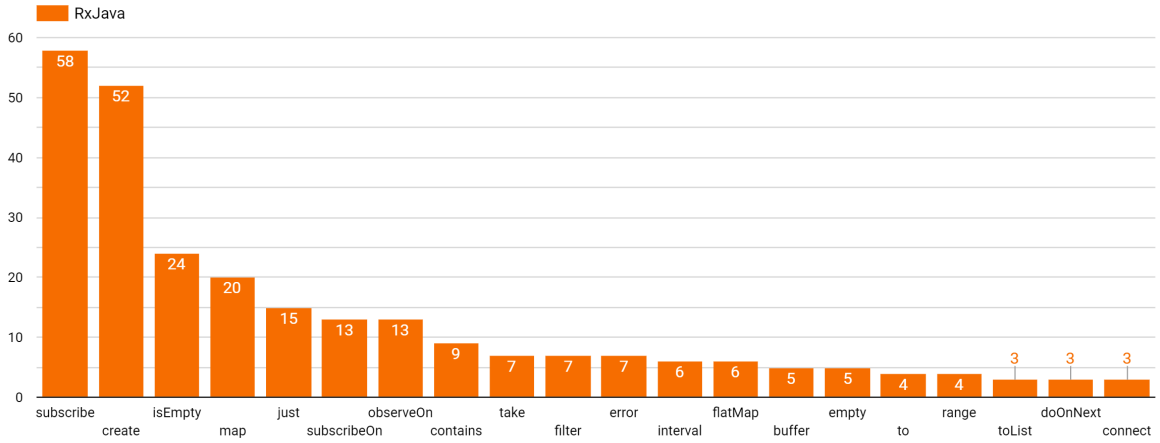

Figura 26. Uso Médio de Operadores em RxJava

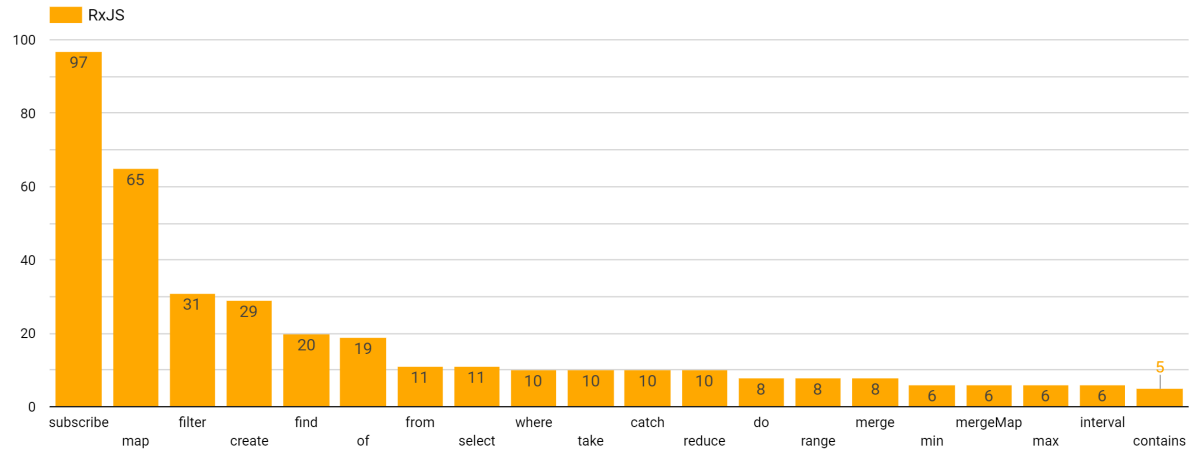

Figura 27. Uso Médio de Operadores em RxJS

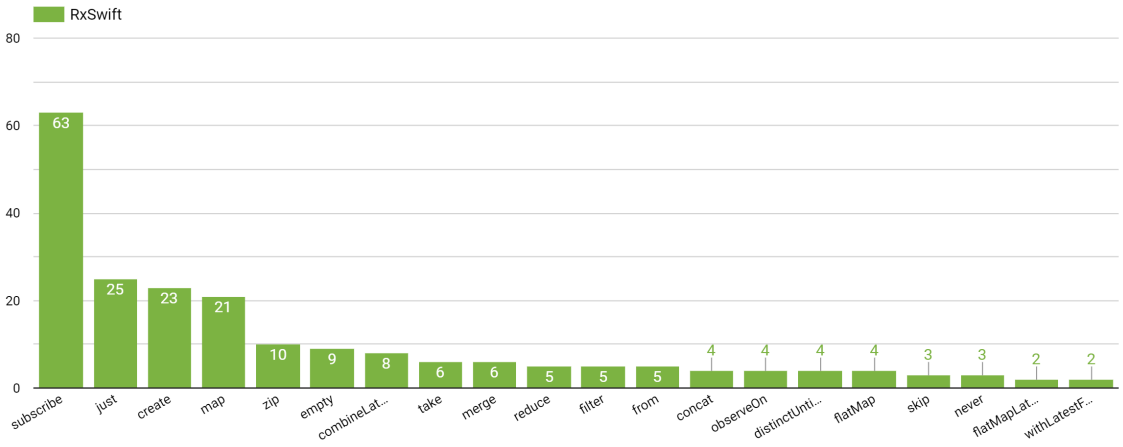

Figura 28. Uso Médio de Operadores em RxSwift

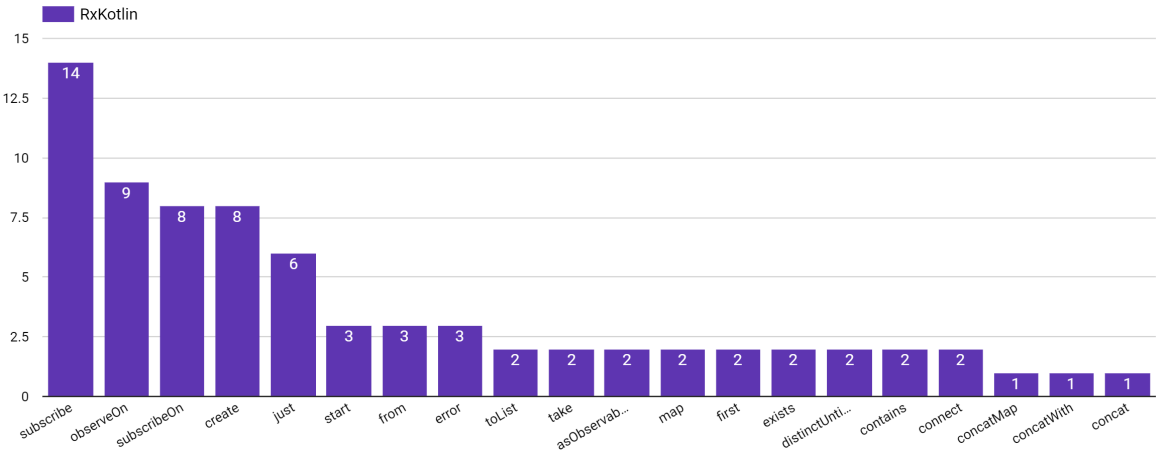

Figura 29. Uso Médio de Operadores em RxKotlin

E. Gráficos de Mediana do Uso de Operadores Por Distribuição

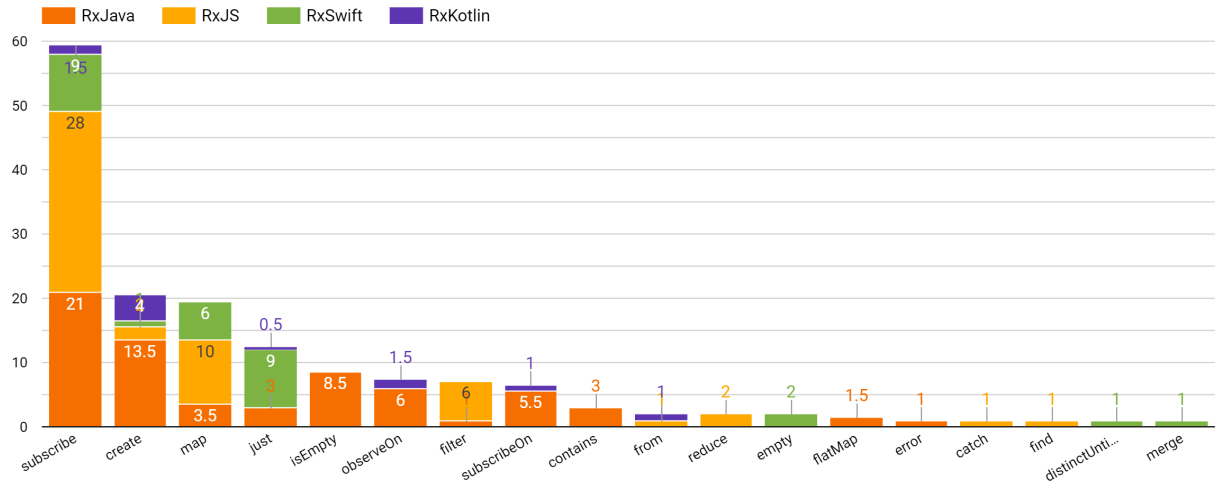

Figura 30. Mediana do Uso de Operadores por Distribuição

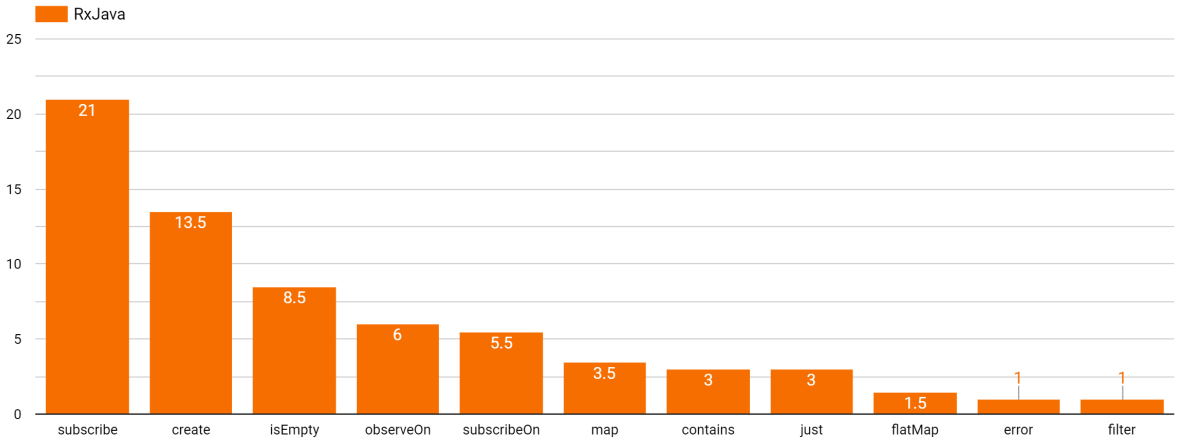

Figura 31. Mediana do Uso de Operadores em RxJava

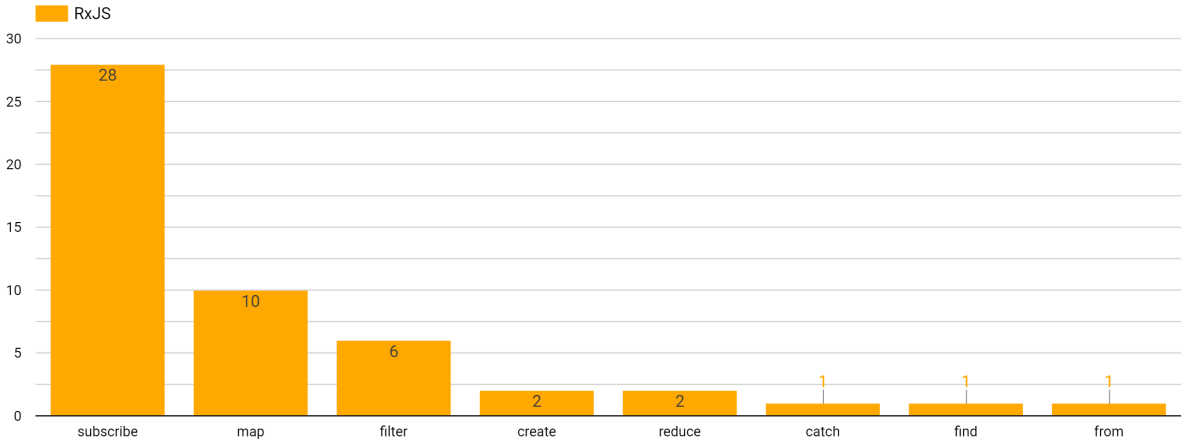

Figura 32. Mediana do Uso de Operadores em RxJS

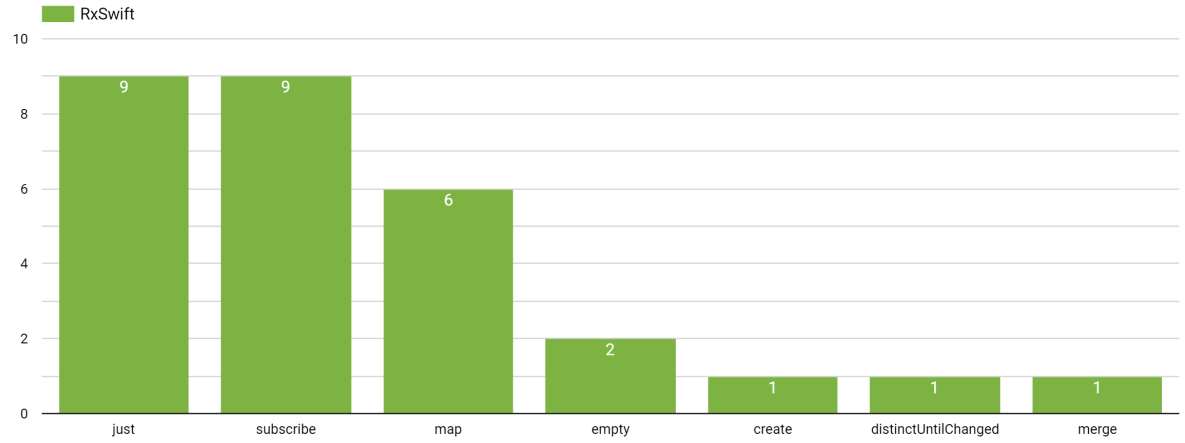

Figura 33. Mediana do Uso de Operadores em RxSwift

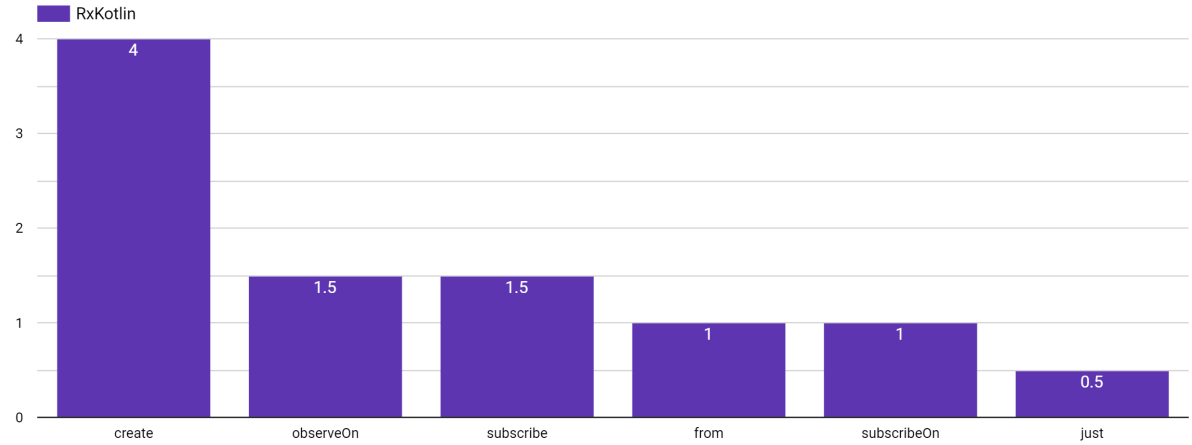

Figura 34. Mediana do Uso de Operadores em RxKotlin

F. Histograma do Total de Operadores Por Frequência de Utilização

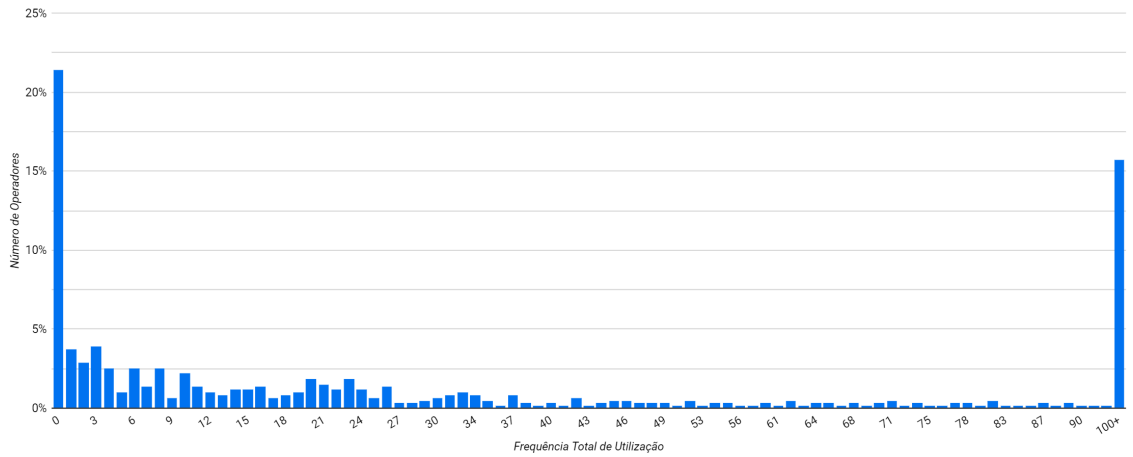

Figura 35. Histograma do Total de Operadores Por Frequência de Utilização

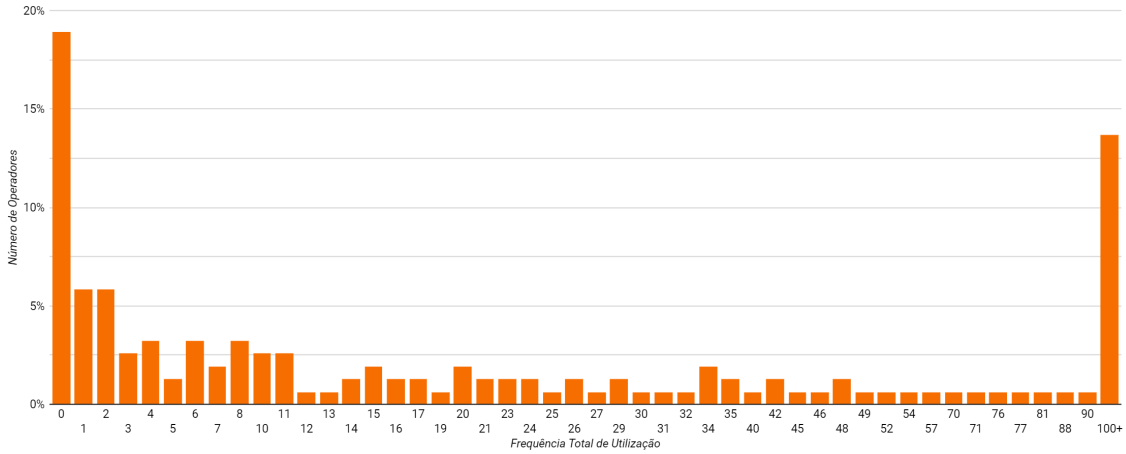

Figura 36. Histograma do Total de Operadores Por Frequência de Utilização em RxJava

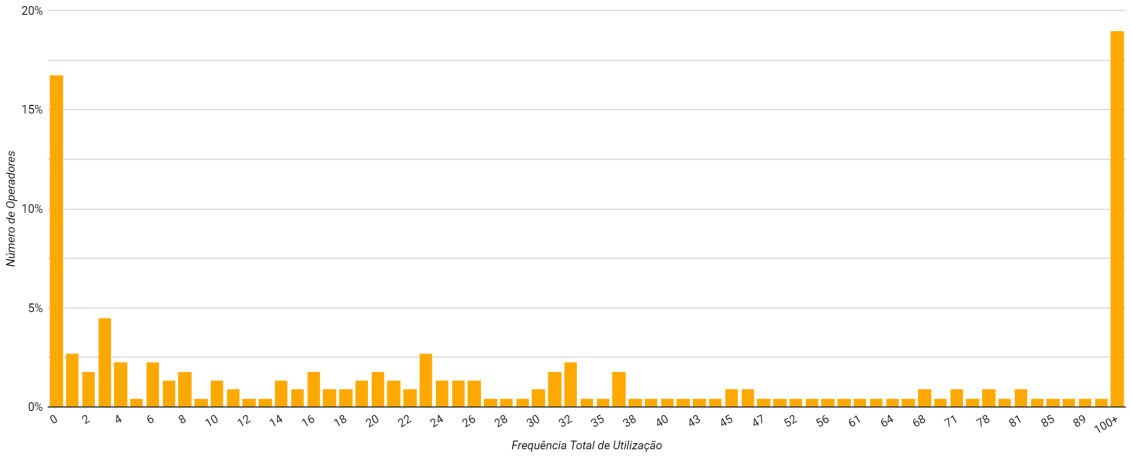

Figura 37. Histograma do Total de Operadores Por Frequência de Utilização em RxJS

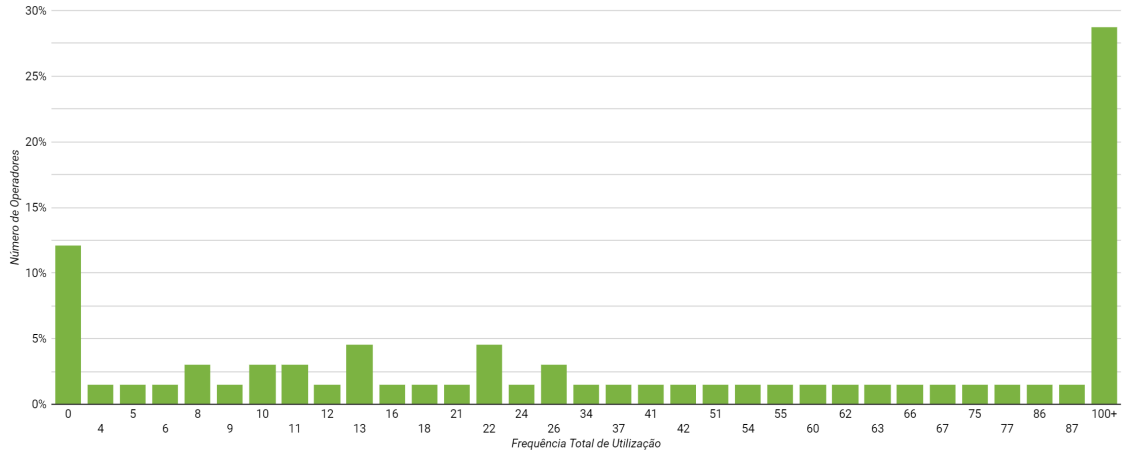

Figura 38. Histograma do Total de Operadores Por Frequência de Utilização em RxSwift

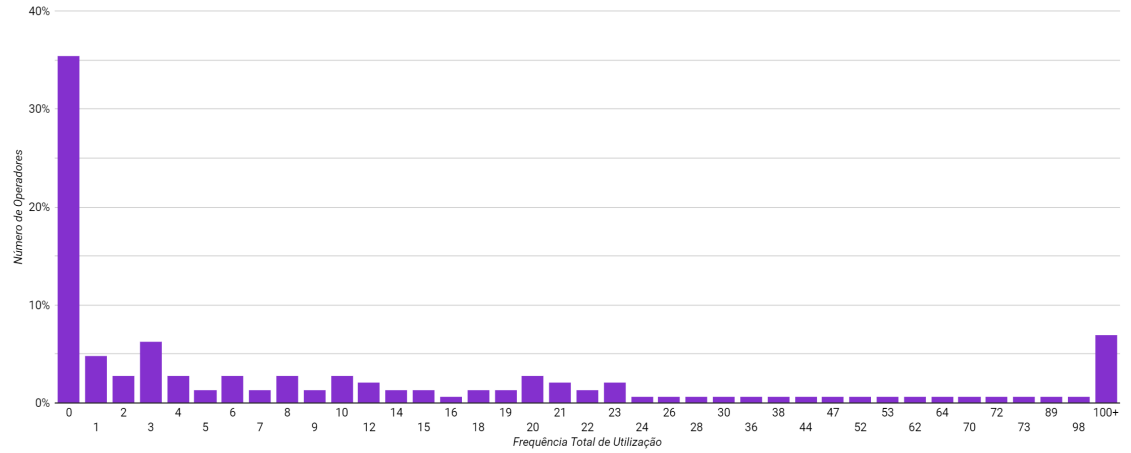

Figura 39. Histograma do Total de Operadores Por Frequência de Utilização em RxKotlin

G. Outros Gráficos

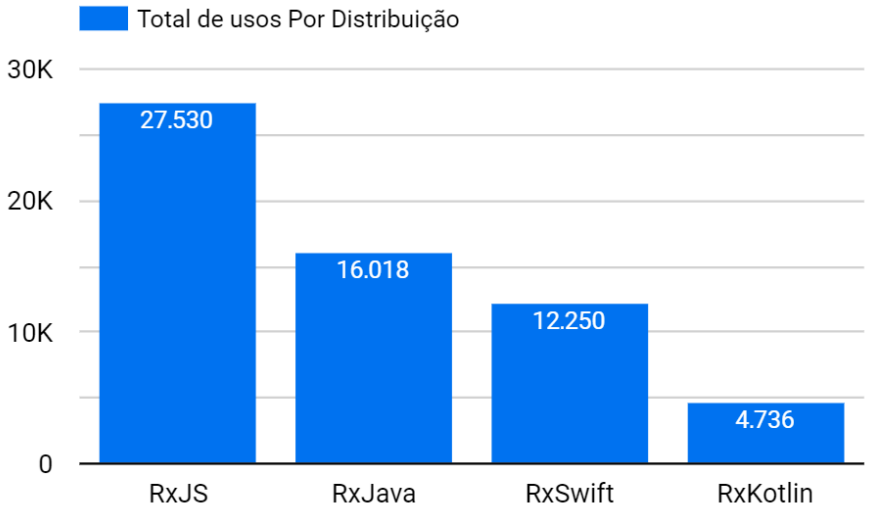

Figura 40. Total de Usos Por Distribuição

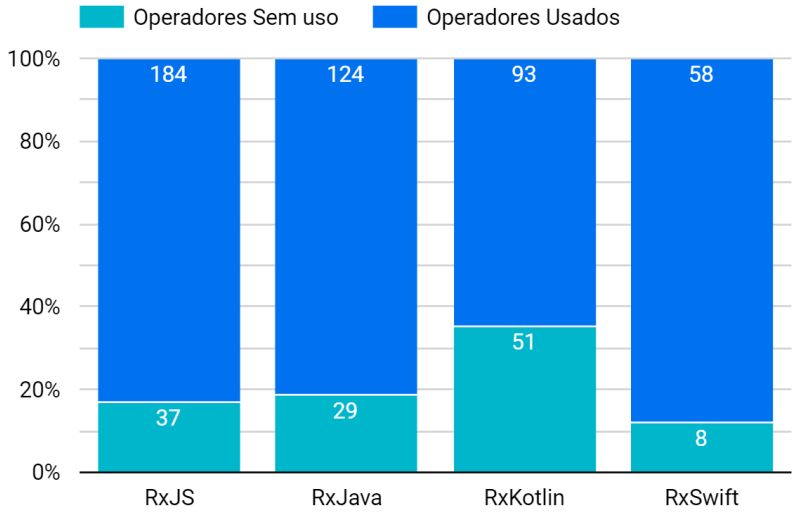

Figura 41. Número de Operadores Usados Não Usados

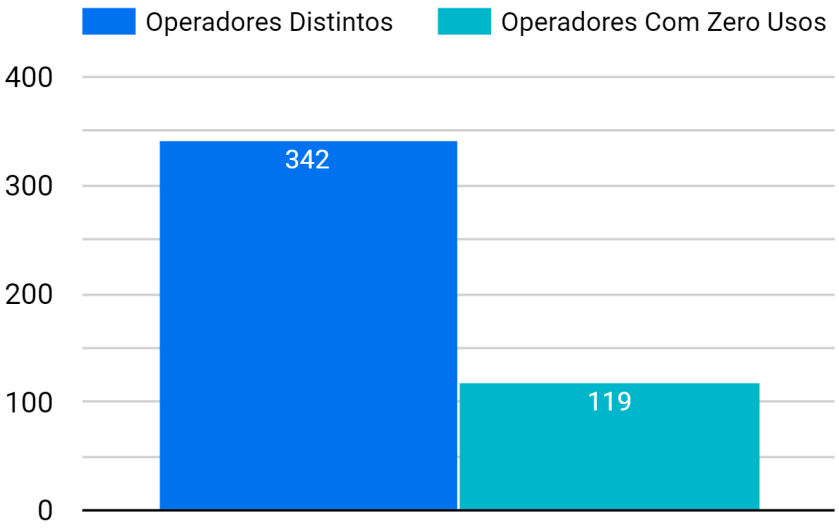

Figura 42. Operadores Distintos Somando as 4 Distribuições

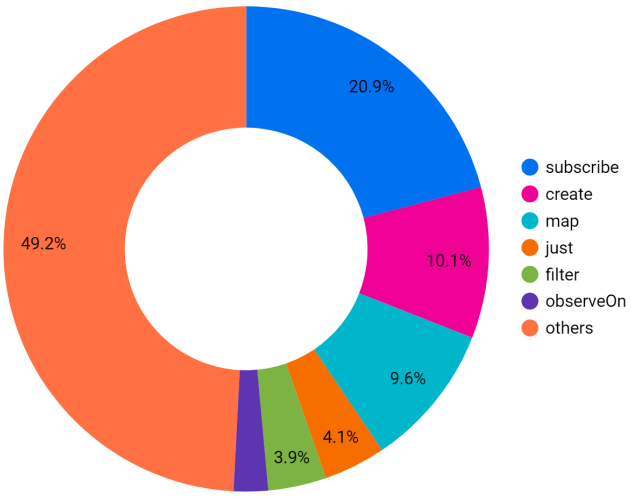

Figura 43. Operadores Que Somam 50% ou Mais das Utilizações

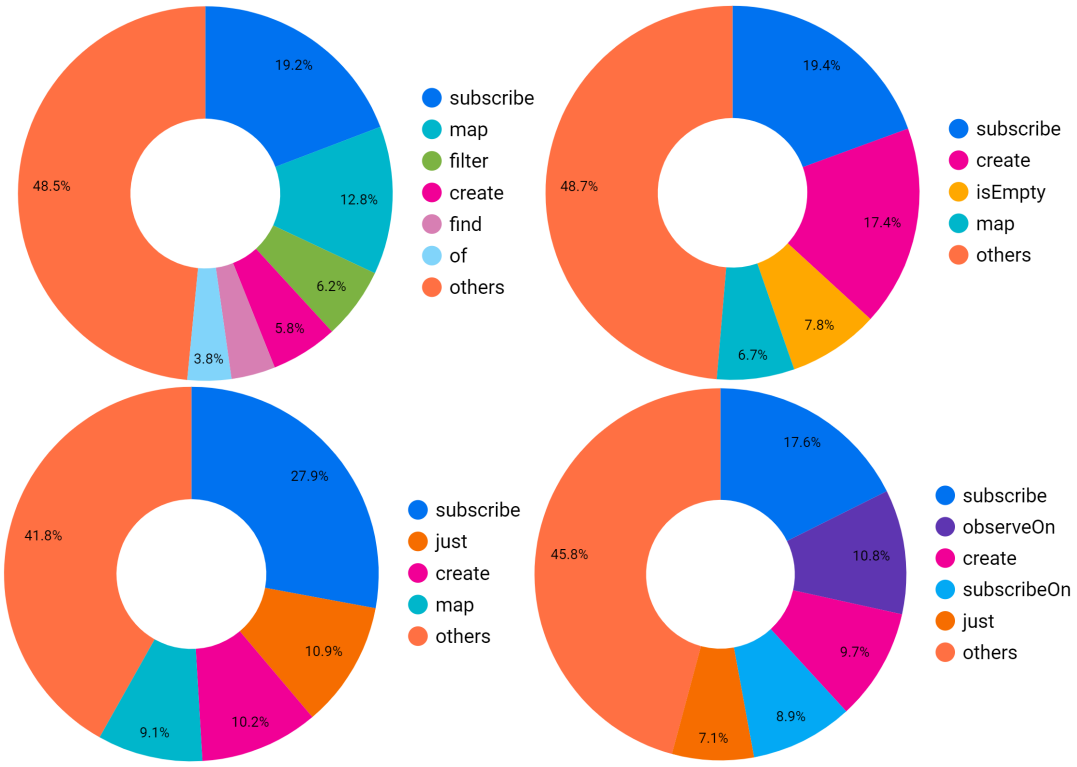

Figura 44. Operadores Que Somam 50% ou Mais das Utilizações em RxJS, RxJava, RxSwift, RxKotlin, respectivamente.