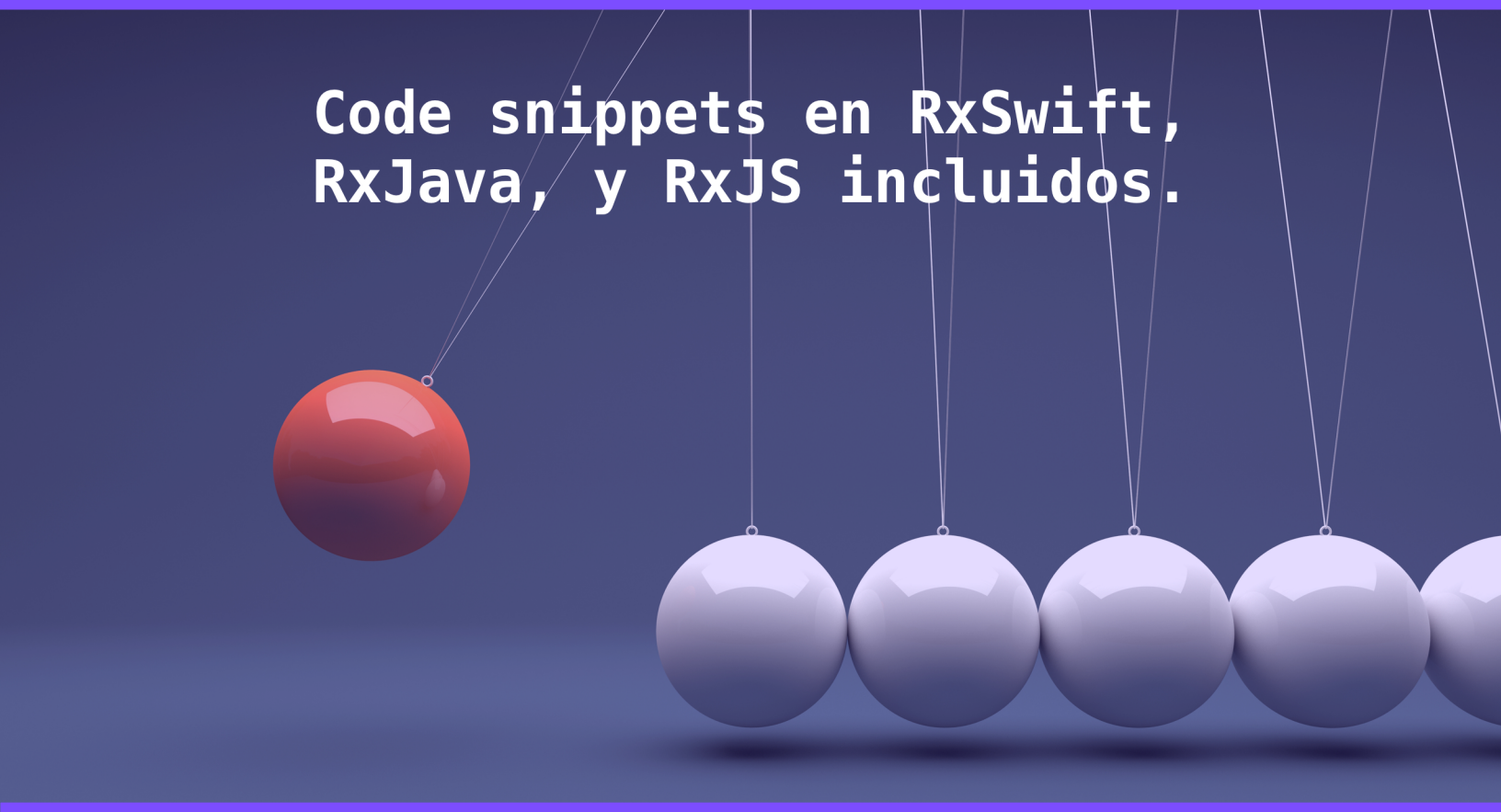
The Clean Way to Use Rx

Manual de guía y consejos para implementar con extensiones Rx

Code snippets en RxSwift, RxJava, y RxJS incluidos.



Yair Carreno

The Clean Way to Use Rx

Manual de guía y consejos para implementar con extensiones Rx.

Yair Carreno

Este libro está a la venta en http://leanpub.com/the-clean-way-to-use-rx-spanish

Esta versión se publicó en 2020-07-15

Éste es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

Índice general

Prefacio

Acerca del libro

El presente trabajo agrupa las diferentes prácticas usadas en la implementación de componentes de software a través de extensiones *Rx*. Dichas prácticas son organizadas en 35 ítems de aplicación y por cada ítem se analiza cuales son prácticas de código recomendadas y cuales son aquellas prácticas que se deben evitar.

También podría decirse que el presente documento es la recopilación de buenas prácticas aprendidas a través de la experiencia propia adquirida en proyectos empresariales, recomendaciones recibidas de foros, blogs, workshops y en general de análisis dados por expertos en el área, incluyendo las recomendaciones dadas en los sitios oficiales de cada extensión[1].

Este libro también pretende ser un manual de referencia práctico a la hora de implementar código *Rx*, es por ello que gran parte del material del libro lo compone ejemplos de *code snippet* escrito en los principales lenguajes de programación para clientes tanto Frontend (móviles, web) como Backend, es decir, JavaScript, Swift, Java. Para la edición siguiente se tiene planeado incluir *Kotlin*.

Los conceptos estudiados en cada uno de los ítems son agnósticos al lenguaje de programación, son pocos los casos en donde cierta capacidad no se encuentra disponible en una extensión y en cuyo caso se hace la respectiva anotación en el ítem.

Se pretende que una vez estudiados los conceptos de cada ítem, el lector se beneficie con el mayor entendimiento sobre como opera *Rx* y dote al lector de herramientas para aplicar buenas prácticas que al final se vean reflejadas en las soluciones en el ámbito de:

- Código limpio (Clean Code).
- Mejor *performance*:
- Mitigar escenarios de *bugs*.

Acerca del código fuente

En el libro, los ejemplos de *code snippet* se presentan en *Swift*. El lector también podrá encontrar el código equivalente en *JavaScript* y *Java* en los capítulos:

- RxJS code snippet
- RxJava code snippet

[1] ReactiveX

El lector también podrá acceder a los repositorios públicos escritos para cada tecnología en:

- Clean Way Rx in JavaScript[2]
- Clean Way Rx in Swift[3]
- Clean Way Rx in Java[4]

Desarrollo del contenido

El desarrollo del contenido inicia desde el análisis de los conceptos fundamentales tanto de *Reactive programming* como de *Functional programming*. La sección *Foundation*, presenta la forma en que dichos conceptos son usados en *Rx* de forma práctica. Los ítems descritos en esta sección son la base para los posteriores ítems.

En la sección siguiente nombrada *Be Clean*, se hace énfasis en el código limpio, bien escrito y de fácil lectura. *Rx* simplifica sustancialmente el código escrito necesario para cumplir con una tarea, sin embargo se puede fácilmente perder la ventaja de la simplicidad y pasar al otro extremo en donde se hace complejo el entendimiento del código por estar escrito de forma no limpia.

Uno de los actores principales en *Rx* es el flujo de datos (*data streams*). Dicho flujo de datos también se puede relacionar con otras fuentes generadoras de flujos y por ende establecer dependencias entre ellos. La sección *Flow dependencies* analiza las consideraciones a tener en cuenta cuando se establecen dichas dependencias entre flujos.

Los operadores son otro de los actores claves en *Rx*, aprender a distinguirlos y saber cuando usar uno u otro es una de las habilidades claves que se adquieren durante el desarrollo y diseño con *Rx*. En la sección *Operators*, se involucran los ítems con las recomendaciones del uso de los operadores.

El tiempo entra a ser una variable a tener en cuenta cuando se orquesta flujos de datos, esto es analizado en la sección *Timing*. Una buena administración de los momentos y tiempos de ejecución generan los resultados esperados.

La siguiente sección trata un tema relevante para cualquier aplicación y tiene que ver con la gestión y administración de los escenarios de error. *Rx* cuenta con un conjunto importante de operadores y funciones que permiten diseñar un control efectivo de los errores. En la sección *Error Handling* se analiza dichas opciones.

Las dos secciones siguiente *Multicasting* y *Multi-threading* podrían lucir parecidas pero en realidad son muy distintas. En *Multicasting* se analiza las prácticas a la hora de vincular múltiples fuentes emisoras de datos y múltiples receptores de dichos datos. En cambio, en la sección *Multi-threading* se analiza la aplicación de concurrencia y paralelización de tareas en múltiples contextos de ejecución en el ámbito de procesamiento.

[2] https://github.com/yaircarreno/Clean-Way-Rx-JavaScript
[3] https://github.com/yaircarreno/Clean-Way-Rx-Swift
[4] https://github.com/yaircarreno/Clean-Way-Rx-Java

Y finalizando, se tiene la sección *Optimizing*, la cuál contiene un par de ítems referentes a buenas prácticas que evitan generación de fugas de memoria y buen uso de los recursos de memoria usados por los componentes diseñados con *Rx*.

Como el lector podrá apreciar, el contenido no se desarrolla siguiendo una línea de temas básicos, medios y avanzados, sino que al contrario todo se encuentra relacionado, a tal punto que existen ítems que se encuentran relacionados con otros y por tanto se complementan.

Glosario

Durante el desarrollo de los temas en el libro, el lector podrá encontrar algunos términos los cuales son descritos a continuación para dar mayor claridad.

Contrato: Hace referencia a los eventos que podrían presentarse en una emisión y los cuales corresponden a *onNext*, *onComplete* o *onError*.

Imperativo: Se refiere a la forma tradicional de programación basada en la ejecución de tareas de forma secuencial.

Reactivo: Hace referencia al estilo de programación basado en la ejecución de tareas de forma sincrónica o asincrónicas y para cuyos propósitos se emplea las extensiones *Rx*.

Rx: Este término será usado para hacer referencia a *Reactive programming*.

Audiencia

Recomiendo esta lectura para todo actor que participa en procesos de diseño, implementación y validación de componentes de software a través de *Reactive programming* y *Functional programming*, apoyados en el uso de extensiones *Rx*.

Ya sea que se trate de un componente en Frontend o Backend, el lector podrá aplicar los conocimientos de este libro al momento de implementar código con extensiones *Rx*. Por ejemplo, permite aplicar diseños de patrones tales como Uniderectional State Flow, MVVM, Optimistic UI y otros patrones que podría requerir la arquitectura de la solución[5].

Consultas y/o contacto

Este trabajo pretender guiar al lector en la búsqueda de buenas prácticas de diseño en soluciones, ha sido desarrollado desde mi punto de vista y teniendo en cuenta las recomendaciones de fuentes expertas citadas en la bibliografía, muy seguramente el lector podría encontrar alternativas a las técnicas expuestas que pueden variar un poco a las que aquí se encuentren consignadas y esa es precisamente la idea del presente trabajo.

[5] Clean Architecture in iOS

Si el lector encuentra en el libro algún aspecto que merezca ser revisado son bienvenidos los comentarios y la retroalimentación que se dé al respecto. Para ello y cualquier duda o inquietud se encuentra disponible los siguientes canales:

- Email: yaircarreno@gmail.com
- Twitter: @yaircarreno[6]
- Blog: yaircarreno.com[7]

Idioma de textos en las imágenes

Por estándar, las imágenes presentadas a lo largo del presente libro contienen textos en inglés.

Idioma en Code Snippets

Por estándar, los ejemplos de código que acompañan el presente libro se encuentran en inglés. También algunos textos principales se encuentran en Inglés para mantener consistencia con la documentación oficial.

Versiones de IDEs y tecnologías usadas

- Xcode 11.5 - Swift 5
- Android Studio 4.0 - Java 1.8.0
- Visual Studio Code 1.47.1 - TypeScript 3.8.3 - Angular 9.1.0

Change log

Para mantener al lector informado sobre las actualizaciones y cambios en el presente libro, se proporciona el capítulo Changelog.

[6] https://twitter.com/yaircarreno
[7] https://www.yaircarreno.com/

Resumen de reglas

Foundation

Ítem 1: De Imperativo a Reactive

Ítem 2: Focus en funciones puras

Ítem 3: Verifique la naturaleza de los streams cuando sean vinculados

Ítem 4: Decida cuando utilizar un Observable tipo empty o never

Be Clean

Ítem 5: Mantenga limpia la cadena de operadores

Ítem 6: Defina cadenas funcionales de operadores

Ítem 7: No anide las suscripciones, utilice la cadena de operadores

Ítem 8: Utilice Rx cuando existan eventos involucrados

Ítem 9: No orqueste operaciones Rx con programación imperativa

Ítem 10: No incluya código Rx en operadores Do

Ítem 11: Propague envolventes (wrappers), no unidades

Ítem 12: Evite propagar valores nulos, recurra a Optionals

Ítem 13: Recuerde que puede usar filter, en lugar de if

Ítem 14: Es posible dividir la emisión cuando se presente if-else

Ítem 15: Compartir código no es lo mismo que compartir emisiones

Ítem 16: No todo se delega al Observer

Ítem 17: Retrase lo que más pueda la definición del Schedule

Flow dependencies

Ítem 18: Verifique eventos de finalización o error cuando utilice los filtros

Ítem 19: Cuidado con los operadores de reducción en streams infinitos

Ítem 20: Mantenga consistencia en la naturaleza de la emisión

Operators

Timing

Error handling

Multicasting

Multi-threading

Optimizing

Foundation

Ítem 1: De Imperativo a Reactive

Relacionar *código imperativo* con *código reactive* es posible siempre y cuando se tengan en cuenta las capacidades de cada uno de estos estilos.

Es normal que la solución contenga bloques de código tanto imperativo como reactive. En realidad el origen de código reactive es código imperativo al cual se le agrega capacidades heredadas de *programación funcional* y capacidades *multithreaded*.

Las reglas básicas para relacionar código imperativo y código reactive podrían resumirse en los siguientes puntos:

1. La orquestación debe ocurrir solo entre código *Rx*, es decir, no se debería orquestar tareas *Rx* usando código *imperativo*.
2. Código *imperativo* puede ser orquestado con tareas *Rx* siempre y cuando el código *imperativo* sea envuelto en código *Rx*.

El lector encontrará en ítem 8, ítem 9 y ítem 10, los efectos de no cumplir las dos recomendaciones anteriores.

Ahora, ¿Cómo se envuelve código imperativo en código *Rx*?

Mediante dos mecanismos disponibles:

- A través de operadores de creación.
- A través de las funciones de creación.

Pero antes de diseñar la envolvente tenga en cuenta el siguiente análisis planteado en la figura 1.1.

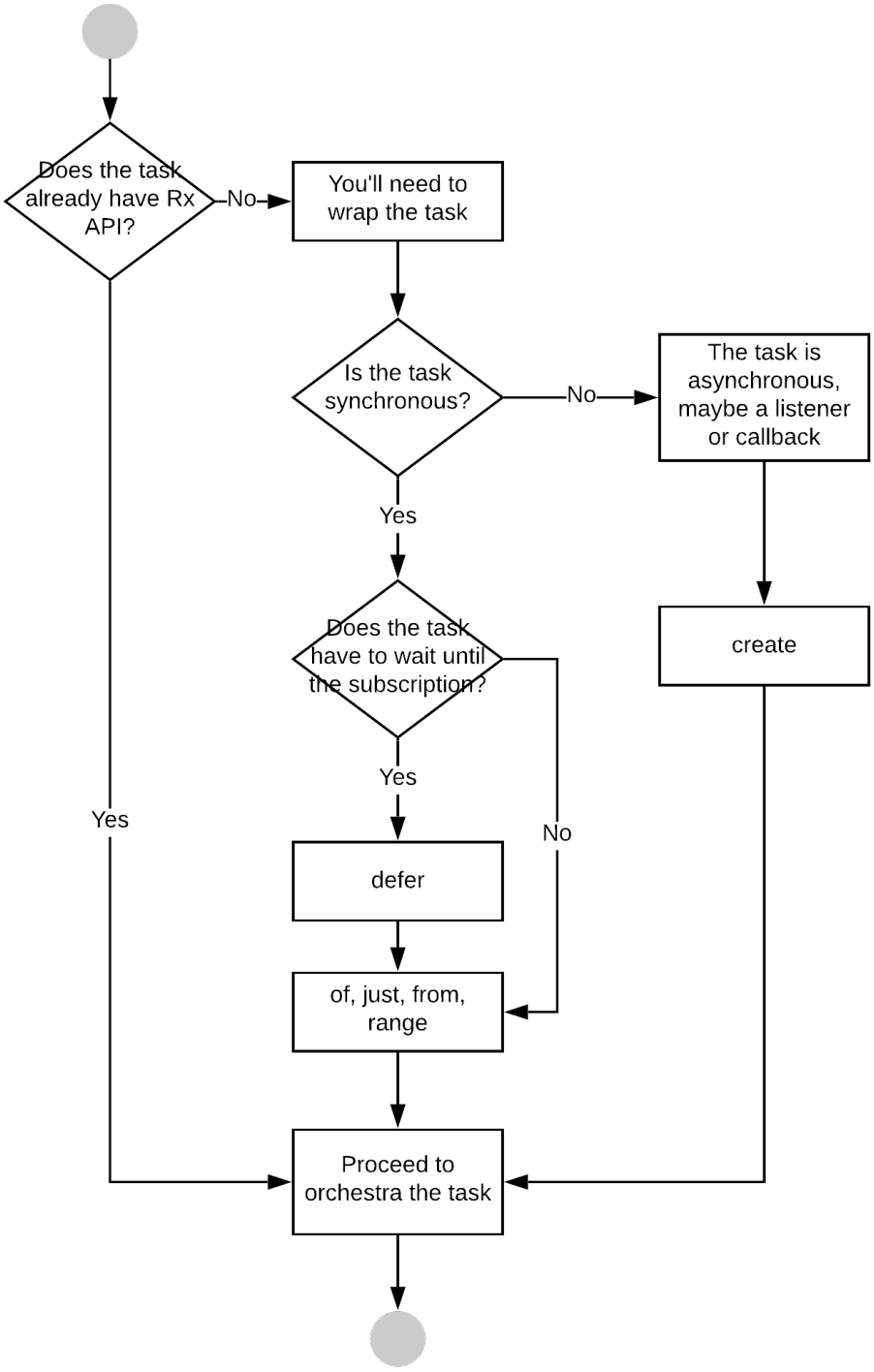


Figure 1.1 Wrapping tasks to Rx

- Valide primero si la tarea ya cuenta con una envolvente. Las librerías de terceros, por ejemplo, proporcionan estas implementaciones cuando se trata de integrar SDK de terceros. Incluso en el propio proyecto podría ya existir una utilidad con dicha envolvente.

- En caso de requerir la envolvente, tenga en cuenta si la tarea realiza la operación de forma *sincrónica* o *asincrónica*. Identifique esto verificando si la tarea la ejecuta un *listener*, *callback* o cualquier componente de naturaleza asincrónica.

  A veces se cree por error que toda operación realizada con *Rx* es *multithreaded* por defecto. La verdad es que Rx es *single-threaded* por defecto[^8].

  A menos que se indique lo contrario, toda operación correrá sobre el mismo hilo asignado por defecto. En el Ítem 30: Rx es single-threaded por defecto se analiza esta característica.
- Si la tarea es *sincrónica*, podría crearse a través de los operadores de creación un *Observable* que envuelva la tarea.
- Si la tarea es *asincrónica*, podría ser envuelta a través de las funciones de creación y crear *Traits*, es decir, *Observables* de tipo *Single*, *Complete* o *Maybe* de acuerdo al requerimiento y teniendo pleno conocimiento de los contratos a cumplir por parte del *Observer*.

A continuación se presenta algunas de las opciones para envolver tareas a través de operadores.

Nombre de operadores

Tenga en cuenta que algunos operadores se nombran de forma distinta o no se encuentran disponibles en todos los lenguajes.

Operadores de creación

Rx cuenta con los siguientes mecanismos para envolver tareas a través de operadores:

Code Snippet: RxJS - RxJava

Example 1.1: of operator - CleanWayRx/Items/Item1.swift

```
Observable.of(["h", "e"], ["l", "l", "o"])
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)


Console output:
---------------
["h", "e"]
["l", "l", "o"]
```

Example 1.2: just operator - CleanWayRx/Items/Item1.swift

```
Observable.just(["h", "e", "l", "l", "o"])
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)

Console output:
---------------
["h", "e", "l", "l", "o"]
```

Example 1.3: from operator - CleanWayRx/Items/Item1.swift

```
Observable.from(["h", "e", "l", "l", "o"])
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)

Console output:
---------------
h
e
l
l
o
```

Example 1.4: range operator - CleanWayRx/Items/Item1.swift

```
Observable.range(start: 1, count: 5)
    .subscribe(onNext: { element in
        print(element)
    })
    .disposed(by: disposeBag)


Console output:
---------------
1
2
3
4
5
```

Example 1.5: defer operator - CleanWayRx/Items/Item1.swift

```
Observable.deferred {
    return Observable.just(["h", "e", "l", "l", "o"])
}
.subscribe(onNext: { element in print(element) })
.disposed(by: disposeBag)


Console output:
---------------
["h", "e", "l", "l", "o"]
```

Operadores de apoyo

Estos son también operadores de creación no destinados a envolver sino a apoyar procesos testing, depuración y del manejo de errores:

Code Snippet: RxJS - RxJava

Example 1.6: interval operator - CleanWayRx/Items/Item1.swift

```
let subscription = Observable<Int>
    .interval(.seconds(1), scheduler: scheduler)
    .subscribe { event in print(event) }

Thread.sleep(forTimeInterval: 5.0)
subscription.dispose()


Console output:
---------------
next(0)
next(1)
next(2)
next(3)
next(4)
```

Example 1.7: timer operator - CleanWayRx/Items/Item1.swift

```
let subscription = Observable<Int>.timer(.seconds(2), scheduler: scheduler)
    .subscribe { event in print(event) }

Thread.sleep(forTimeInterval: 5)
subscription.dispose()


Console output:
---------------
next(0)
completed
```

Example 1.8: empty operator - CleanWayRx/Items/Item1.swift

```
Observable<String>.empty()
    .subscribe { event in print(event) }
    .disposed(by: disposeBag)

Console output:
---------------
completed
```

Example 1.9: never operator - CleanWayRx/Items/Item1.swift

```
Observable<String>.never()
    .subscribe { event in print(event) }
    .disposed(by: disposeBag)

Console output:
---------------
```

Example 1.10: error operator - CleanWayRx/Items/Item1.swift

```
Observable<String>.error(SampleError())
    .subscribe { event in print(event) }
    .disposed(by: disposeBag)


Console output:
---------------
error(SampleError())
```

Funciones de creación

Cuando se trata de envolver una tarea asincrónica que forma parte de una API externo y en cuyo caso no es suficiente con un operador de creación, se recurre a la creación del *Observable* a través de las funciones de creación.

La anatomía para envolver una tarea es la siguiente:

Example 1.11: CleanWayRx/Items/Item1.swift

```swift
private func taskWrapped() -> Observable<Any> {

    return Observable.create { observer in

        // Here the imperative code is embedded
        return Disposables.create()
    }
}
```

Code Snippet: RxJs - RxJava

Los puntos importantes para diseñar el *Observable* a partir de *Observable.create* y código imperativo son:

1. Tener en cuenta los contratos que podría necesitar el *Observer*, es decir, *onNext*, *onComplete* y *onError*.
2. De acuerdo a lo definido en el punto 1, analizar si se usa un *Observable* general o uno de los tipos: *Single*, *Complete* o *Maybe*.
3. Manejar los posibles escenarios que generan error y emitir a través del contrato *onError*.
4. Asegurarse de liberar las respectivas referencias fuertes y cualquier listener vinculado por el código imperativo.
5. Considerar si es necesario manejar escenarios de *Backpressure*[^9].

Considere el siguiente ejemplo de implementación de un envolvente:

Example 1.12: CleanWayRx/Items/Item1.swift

```swift
private func imperativeTask() -> Any? {

    let data = "any data"
    print("Do any imperative task or process")

    return data
}
```

La tarea imperativa llamada *imperativeTask* podría generar un resultado a partir de un cálculo o simplemente realizar una tarea que no genera un resultado pero si una acción finalizada.

Cualquiera que sea el caso, se crea un *Observable* con el resultado con datos o si ellos. Se definen las condiciones para emitir un evento de error, un evento de completado o un evento de emisión del ítem:

Example 1.13: CleanWayRx/Items/Item1.swift

```
private func taskWrapped(task: Any?) -> Observable<Any> {

    return Observable.create { observer in

        guard let data = task else {
            observer.onError(SampleError())
            return Disposables.create {}
        }

        observer.onNext(data)
        observer.onCompleted()
        return Disposables.create()
    }
}
```

Code Snippet: RxJs - RxJava

A través de la envolvente, la tarea queda disponible para ser orquestada y controlada de forma *Rx*:

Example 1.14: CleanWayRx/Items/Item1.swift

```
taskWrapped(task: self.imperativeTask())
    .subscribe(onNext: { data in print("next:", data) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)

Console output:
---------------
Do any imperative task or process
next: any data
completed
```

Ahora se muestra un ejemplo de una envolvente diseñada para una API bastante conocida, *Firebase Database Realtime*:

Example 1.15: Wrapping Firebase login task

```
public func rx_loginUser(user: User) -> Single<Bool> {

    return Single<Bool>.create { single in
        self.authReference.signIn(withEmail: user.email, password: user.password in
            if error == nil {
                single(.success(true))
            }
            else{
                single(.success(false))
            }
        }
        return Disposables.create()
    }
}
```

Otra posible envolvente para consultas:

Example 1.16: Wrapping Firebase single event task

```
func rx_observeSingleEvent(of event: DataEventType) -> Observable<DataSnapshot> {
    return Observable.create({ (observer) -> Disposable in
        self.observeSingleEvent(of: event, with: { (snapshot) in
            observer.onNext(snapshot)
            observer.onCompleted()
        }, withCancel: { (error) in
            observer.onError(error)
        })
        return Disposables.create()
    })
}
```

Be Clean

Ítem 5: Mantenga limpia la cadena de operadores

La *cadena de operadores* es una herramienta poderosa para orquestar tareas. Mantener la definición de la cadena de operadores de forma clara y limpia permite el mejor entendimiento de las tareas orquestadas.

Considere el siguiente ejemplo:

Example 5.1: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token)
        .concatMap { saved in self.network.getUser(username) }
}
.subscribe(onNext: { user in print(user) })
.disposed(by: disposeBag)
```

El objetivo del *code snippet* anterior es orquestar una tarea que obtiene un *token*, luego lo almacena en caché y posteriormente procede a realizar la consulta del usuario.

Ese código es válido, sin embargo podría ser mejorado de la siguiente forma:

Example 5.2: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token) }
    .concatMap { saved in self.network.getUser(username) }
    .subscribe(onNext: { user in print(user) })
    .disposed(by: disposeBag)
```

Code Snippet: RxJS - RxJava

¿Cuál es la diferencia entre el ejemplo de código 5.1 y el ejemplo de código 5.2?

El código del ejemplo 5.1 se suele utilizar para orquestar tareas con dependencias de resultados, es decir cuando la siguiente tarea (*downstream*) requiere de los resultados de la tarea anterior (*upstream*).

En el ejemplo 5.1, la operación *getUser* no necesita de los resultados de las operaciones anteriores, es decir los resultados de *getToken* o *storeToken*. Por esa razón el código del ejemplo 5.1 puede ser reemplazado por el código del ejemplo 5.2 y hacer más claro el código.

Ahora se muestra un caso en donde la dependencia de resultados es necesaria:

Example 5.3: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token)
        .concatMap { saved in self.network.getUser(username, token) }
}
.subscribe(onNext: { user in print(user) })
.disposed(by: disposeBag)
```

En este caso, la tarea *getUser* necesita el resultado de una de las tareas anteriores, es decir, el *token* en este caso. Por lo tanto se emplea una concatenación interna para lograr acceder al valor, ya que de lo contrario la variable *token* sería desconocida y generaría error.

Sin embargo en estos casos de dependencia de tareas también es posible *aplanar* las operaciones. Esto es posible a través de *Tuplas*[^11] así:

Example 5.4: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token).map{saved in (token, saved)} }
    .concatMap { pair in self.network.getUser(username, pair.0) }
    .subscribe(onNext: { user in print(user) })
    .disposed(by: disposeBag)
```

Se recurre a una transformación con el operador *map* para generar y propagar una *tupla* con los valores necesarios a la siguiente tarea.

Cuando los resultados de una tarea no influye en la ejecución de las siguientes tareas, se podría ignorar el resultado. Por ejemplo en el siguiente código:

Example 5.5: CleanWayRx/Items/Item5.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeToken(token) }
    .ignoreElements()
    .andThen(self.network.getUser(username))
    .subscribe(onNext: { user in print(user) })
    .disposed(by: disposeBag)
```

La tarea *getUser* al no depender de los resultados de las tareas anteriores, podría ignorar los elementos.

Error handling

Ítem 28: Gestionando los errores

La clave de una efectiva gestión de errores consiste en identificar plenamente la estrategia a utilizar para reaccionar ante dichos escenarios.

Cuando se presenta un error en el stream, se podría reaccionar de las siguientes formas:

Escenario 1: No hacer nada, no controlar el error y dejar que se propague la excepción hasta el Observer con la finalización inmediata de la ejecución de la cadena de operadores y apagado del *stream*. Esta es la que nunca se debería seguir.

Escenario 2: Capturar el error, controlarlo y emitir un valor personalizado sin apagar el *stream*, aunque con la finalización inmediata de la ejecución de la cadena de operadores.

Escenario 3: Capturar el error, registrarlo en log, e ignorarlo para suspender la ejecución de la cadena de operadores sin apagar el stream.

Escenario 4: Se presenta el error, se reintenta la ejecución de la tarea un número limitado de veces (*n times*). Si después de los reintentos persiste el error, se captura, controla y se suspende la ejecución de la cadena de operadores sin apagar el stream.

Escenario 5: Se presenta el error, se reintenta la ejecución de la tarea un número limitado de veces (*n times*) cada cierta ventana de tiempo (*period*). Si después de los reintentos persiste el error, se captura, controla y se suspende la ejecución de la cadena de operadores sin apagar el stream.

Rx proporciona operadores que permiten diseñar diferentes estrategias para aplicar los escenarios en mención.

A continuación se muestran los ejemplos con las estrategias recomendadas. Se hace énfasis en que el escenario 1 se debe evitar.

Escenario 1

Example 28.1: CleanWayRx/Items/Item28.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
error: SampleError()
```

En este caso, la tarea *storeTokenWithError* generará un error. El resultado es la suspensión de la emisión y sólo se ejecuta el contrato *onError*, ni siquiera *onComplete*. Recordar que *onError* y *onComplete* son excluyentes, o se ejecuta uno, o se ejecuta el otro pero no ambos.

Escenario 2

Example 28.2: CleanWayRx/Items/Item28.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .catchError({ error in Observable.just(User()) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
next: User(name: "", email: "", posts: [])
completed
```

En este caso, el error es controlado y personalizado (se propaga *User* vacío o con valores por defecto). Además los contratos *onNext* y *onComplete* son alcanzados.

También hay que anotar que la tarea *getUser* no alcanza a ser ejecutada, ya que una vez se genera el error en *storeTokenWithError*, se suspende la ejecución de la cadena de operadores.

¿Importa la ubicación del operador de error?

Sí. La captura del error se hace en el punto en donde se defina el operador. En una cadena de operadores, lo recomendable es ubicarlo justo antes de la subscripción.

Escenario 3

Example 28.3: CleanWayRx/Items/Item28.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .catchError({ error in Observable.empty()
        .do(onCompleted: { print(error)  }) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
SampleError()
completed
```

En este caso, cuando se presenta el error se procede a dejar el rastro en logs y se continúa con la emisión ignorando el valor.

Escenario 4

Example 28.4: CleanWayRx/Items/Item28.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .retry(2)
    .catchError({ error in Observable.empty()
        .do(onCompleted: { print(error)  }) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
SampleError()
completed
```

Similar al escenario anterior, sólo que se adiciona el operador *retry*. Importante siempre especificar el número de intentos, ya que de no especificarse dicho parámetro el reintento podría ser permanente y contraproducente para el *performance* de la aplicación.

Escenario 5

Example 28.5: CleanWayRx/Items/Item28.swift

```
network.getToken("api-key")
    .concatMap { token in self.cache.storeTokenWithError(token) }
    .concatMap { saved in self.network.getUser(username) }
    .retryWhen{ errors in errors
        .do(onNext: { ignored in print("retrying...") })
        .delay(.seconds(2), scheduler: MainScheduler.instance)
        .take(4)
        .concat(Observable.error(SampleError()))}
    .catchError({ error in Observable.empty()
        .do(onCompleted: { print(error)  }) })
    .subscribe(onNext: { user in print("next:", user) },
               onError: { error in print("error:", error) },
               onCompleted: { print("completed") } )
    .disposed(by: disposeBag)
```

```
Console output:
---------------
retrying...
retrying...
retrying...
retrying...
retrying...
SampleError()
completed
```

Code Snippet: RxJS - RxJava

RxJS code snippet

Ítem 1: De Imperativo a Reactive

Code Snippet: RxSwift - RxJava

Example 1.1: of operator - clean-way-rx/src/app/items/item1.ts

```
of(["h", "e"], ["l", "l", "o"])
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: (2) ["h", "e"]
next: (3) ["l", "l", "o"]
```

Example 1.2: just (of with single emision) operator - clean-way-rx/src/app/items/item1.ts

```
of(["h", "e", "l", "l", "o"])
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: (5) ["h", "e", "l", "l", "o"]
```

Example 1.3: from operator - clean-way-rx/src/app/items/item1.ts

```
from(["h", "e", "l", "l", "o"])
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: h
next: e
next: l
next: l
next: o
```

Example 1.4: range operator - clean-way-rx/src/app/items/item1.ts

```
range(1, 5)
    .subscribe(
        next => console.log('next:', next));


Console output:
---------------
next: 1
next: 2
next: 3
next: 4
next: 5
```

Example 1.5: defer operator - clean-way-rx/src/app/items/item1.ts

```
defer(() =>
    of(["h", "e", "l", "l", "o"]))
    .subscribe(
        next => console.log('next:', next));

Console output:
---------------
next: (5) ["h", "e", "l", "l", "o"]
```

Operadores de apoyo

Example 1.6: interval operator - clean-way-rx/src/app/items/item1.ts

```
const subscribe = interval(1000)
    .subscribe(
        next => console.log('next:', next));

setTimeout(() => {
    subscribe.unsubscribe();
}, 5000);

Console output:
---------------
next: 0
next: 1
next: 2
next: 3
next: 4
```

Example 1.7: timer operator - clean-way-rx/src/app/items/item1.ts

```
const subscribe = timer(1000, 1000)
    .subscribe(
        next => console.log('next:', next));

setTimeout(() => {
    subscribe.unsubscribe();
}, 5000);


Console output:
---------------
next: 0
next: 1
next: 2
next: 3
next: 4
```

Example 1.8: empty operator - clean-way-rx/src/app/items/item1.ts

```
empty()
    .subscribe({
        next: () => console.log('Next'),
        complete: () => console.log('Complete!')
    });


Console output:
---------------
Complete!
```

Example 1.9: never operator - clean-way-rx/src/app/items/item1.ts

```
never()
    .subscribe({
        next: () => console.log('Next'),
        complete: () => console.log('Complete!')
    });
```

Example 1.10: error operator - clean-way-rx/src/app/items/item1.ts

```
throwError('SampleError!')
    .subscribe({
        next: val => console.log(val),
        complete: () => console.log('Complete!'),
        error: val => console.log(`Error: ${val}`)
    });


Console output:
---------------
Error: SampleError!
```

Example 1.11: clean-way-rx/src/app/items/item1.ts

```
taskWrapped(): Observable<any> {

    return Observable.create(function (observer: any) {
        // Here the imperative code is embedded
    });
}
```

Code Snippet: RxSwift - RxJava

Example 1.12: clean-way-rx/src/app/items/item1.ts

```
imperativeTask() {
    const data = "any data";
    console.log("Do any imperative task or process");
    return data;
}
```

Example 1.13: clean-way-rx/src/app/items/item1.ts

```
taskWrapped(task: any): Observable<any> {

    return Observable.create(function (observer: any) {

        let data = task;
        if (data) {
            observer.next(data);
            observer.complete();
        } else {
            observer.error("SampleError");
        }
    });
}
```

Code Snippet: RxSwift - RxJava

Example 1.14: clean-way-rx/src/app/items/item1.ts

```
this.taskWrapped(this.imperativeTask())
    .subscribe({
        next: val => console.log(val),
        complete: () => console.log('Complete!'),
        error: val => console.log(`Error: ${val}`)
    });

Console output:
---------------
Do any imperative task or process
any data
Complete!
```

Ítem 5: Mantenga limpia la cadena de operadores

Code Snippet: RxSwift - RxJava

Example 5.1: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)
            .pipe(saved => this.network.getUser(username)))
    )
    .subscribe(user => console.log(user));
```

Example 5.2: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)),
        concatMap(saved => this.network.getUser(username))
    )
    .subscribe(user => console.log(user));
```

Example 5.3: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)
            .pipe(saved => this.network.getUserWithToken(username, token)))
    )
    .subscribe(user => console.log(user));
```

Example 5.4: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
  .pipe(
    concatMap(token => this.cache.storeToken(token).pipe(map(saved => [token, saved]))),
    concatMap(pair => this.network.getUserWithToken(username, pair[0] as Token))
  )
  .subscribe(user => console.log(user));
```

Example 5.5: clean-way-rx/src/app/items/item5.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeToken(token)),
        ignoreElements(),
        concat(this.network.getUser(username))
    )
    .subscribe(user => console.log(user));
```

Ítem 28: Gestionando los errores

Code Snippet: RxSwift - RxJava

Example 28.1: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username))
    )
    .subscribe(user => console.log(user));
```

Example 28.2: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        catchError(error => of(new User())))
    .subscribe(
        user => console.log('next:', user),
        error => console.log('error:', error),
        () => console.log('completed'));
```

Example 28.3: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        catchError(error =>
            empty()
                .pipe(tap({ complete: () => console.log(error) }))
        ))
    .subscribe(
        user => console.log('next:', user),
        err => console.log('error:', err),
        () => console.log('completed'));
```

Example 28.4: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        retry(2),
        catchError(error =>
            empty()
                .pipe(tap({ complete: () => console.log(error) }))
        ))
    .subscribe(
        user => console.log('next:', user),
        err => console.log('error:', err),
        () => console.log('completed'));
```

Example 28.5: clean-way-rx/src/app/items/item28.ts

```
this.network.getToken("api-key")
    .pipe(
        concatMap(token => this.cache.storeTokenWithError(token)),
        concatMap(saved => this.network.getUser(username)),
        retryWhen(errors => errors.pipe(
            tap(() => console.log('retrying...')),
            delay(2000),
            take(4),
            concat(throwError('SampleError!'))
        )),
        catchError(error =>
            empty()
                .pipe(tap({ complete: () => console.log(error) }))
        ))
    .subscribe(
        user => console.log('next:', user),
        err => console.log('error:', err),
        () => console.log('completed'));
```

RxJava code snippet

Ítem 1: De Imperativo a Reactive

Code Snippet: RxSwift - RxJS

Example 1.1: of (fromArray) operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
    Observable.fromArray(Arrays.asList("h", "e"), Arrays.asList("l", "l", "o"))
            .subscribe(element -> Log.d(TAG, "" + element)));

Console output:
---------------
D/Item1: [h, e]
D/Item1: [l, l, o]
```

Example 1.2: just operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.just(Arrays.asList("h", "e", "l", "l", "o"))
                .subscribe(element -> Log.d(TAG, "" + element)));


Console output:
---------------
D/Item1: [h, e, l, l, o]
```

Example 1.3: from (fromIterable) operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.fromIterable(Arrays.asList("h", "e", "l", "l", "o"))
                .subscribe(element -> Log.d(TAG, "" + element)));


Console output:
---------------
D/Item1: h
D/Item1: e
D/Item1: l
D/Item1: l
D/Item1: o
```

Example 1.4: range operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.range(1, 5)
                .subscribe(element -> Log.d(TAG, "" + element)));


Console output:
---------------
D/Item1: 1
D/Item1: 2
D/Item1: 3
D/Item1: 4
D/Item1: 5
```

Example 1.5: defer operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.defer(() ->
                Observable.just(Arrays.asList("h", "e", "l", "l", "o")))
                .subscribe(element -> Log.d(TAG, "" + element)));

Console output:
---------------
D/Item1: [h, e, l, l, o]
```

Operadores de apoyo

Example 1.6: interval operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.interval(1, TimeUnit.SECONDS)
                .subscribe(element -> Log.d(TAG, "" + element)));

sleep(5000);

Console output:
---------------
D/Item1: 0
D/Item1: 1
D/Item1: 2
D/Item1: 3
D/Item1: 4
```

Example 1.7: timer operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.timer(1, TimeUnit.SECONDS)
                .subscribe(element -> Log.d(TAG, "" + element),
                        throwable -> Log.e(TAG, "error:" + throwable),
                        () -> Log.d(TAG, "completed")));

sleep(5000);


Console output:
---------------
D/Item1: 0
D/Item1: completed
```

Example 1.8: empty operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.empty()
                .subscribe(element -> Log.d(TAG, "" + element),
                        throwable -> Log.e(TAG, "error:" + throwable),
                        () -> Log.d(TAG, "completed")));


Console output:
---------------
D/Item1: completed
```

Example 1.9: never operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.never()
                .subscribe(element -> Log.d(TAG, "" + element),
                        throwable -> Log.e(TAG, "error:" + throwable),
                        () -> Log.d(TAG, "completed")));


Console output:
---------------
```

Example 1.10: error operator - CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        Observable.error(() -> {
            throw new Exception("SampleError!");
        })
        .subscribe(element -> Log.d(TAG, "" + element),
                   throwable -> Log.e(TAG, "error:" + throwable),
                   () -> Log.d(TAG, "completed")));


Console output:
---------------
E/Item1: error:java.lang.Exception: SampleError!
```

Example 1.11: CleanWayRx/app/src/main/java/../items/Item1.java

```
private Observable<String> taskWrapped() {

    return Observable.create(emitter -> {
        // Here the imperative code is embedded
    });
}
```

Code Snippet: RxSwift - RxJS

Example 1.12: CleanWayRx/app/src/main/java/../items/Item1.java

```
private String imperativeTask() {

    String data = "Any data";
    Log.d(TAG, "Do any imperative task or process");
    return data;
}
```

Example 1.13: CleanWayRx/app/src/main/java/../items/Item1.java

```
private Observable<String> taskWrapped(final Object task) {

    return Observable.create(emitter -> {
        try {
            String data = (String) task;
            emitter.onNext(data);
            emitter.onComplete();
        } catch (Throwable e) {
            emitter.onError(e);
        }
    });
}
```

Code Snippet: RxSwift - RxJS

Example 1.14: CleanWayRx/app/src/main/java/../items/Item1.java

```
compositeDisposable.add(
        taskWrapped(this.imperativeTask())
                .subscribe(data -> Log.d(TAG, "next: " + data),
                           throwable -> Log.e(TAG, "error: " + throwable),
                           () -> Log.d(TAG, "completed")));

Console output:
---------------
D/Item1: Do any imperative task or process
D/Item1: next: Any data
D/Item1: completed
```

Ítem 5: Mantenga limpia la cadena de operadores

Code Snippet: RxSwift - RxJS

Example 5.1: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token)
                        .concatMap(saved -> network.getUser(username)))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.2: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token))
                .concatMap(saved -> network.getUser(username))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.3: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token)
                        .concatMap(saved -> network.getUser(username, token)))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.4: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token)
                        .map(saved -> new Pair<>(token, saved)))
                .concatMap(pair -> network.getUser(username, pair.first))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Example 5.5: CleanWayRx/app/src/main/java/../items/Item5.java

```
compositeDisposable.add(
        this.network.getToken("api-key")
                .concatMap(token -> cache.storeToken(token))
                .ignoreElements()
                .andThen(network.getUser(username))
                .subscribe(user -> Log.d(TAG, "User: " + user)));
```

Ítem 28: Gestionando los errores

Code Snippet: RxSwift - RxJS

Example 28.1: CleanWayRx/app/src/main/java/../items/Item28.java

```
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .subscribe(element -> Log.d(TAG, "" + element),
                       throwable -> Log.e(TAG, "error:" + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.2: CleanWayRx/app/src/main/java/../items/Item28.java

```
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .onErrorResumeNext(throwable -> Observable.just(new User()))
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.3: CleanWayRx/app/src/main/java/../items/Item28.java

```
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .onErrorResumeNext(throwable -> {
                    Log.d(TAG, Objects.requireNonNull(throwable.getMessage()));
                    return Observable.empty();
            })
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.4: CleanWayRx/app/src/main/java/../items/Item28.java

```
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .retry(2)
            .onErrorResumeNext(throwable -> {
                    Log.d(TAG, Objects.requireNonNull(throwable.getMessage()));
                    return Observable.empty();
            })
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```

Example 28.5: CleanWayRx/app/src/main/java/../items/Item28.java

```
compositeDisposable.add(
    this.network.getToken("api-key")
            .concatMap(token -> cache.storeTokenWithError(token))
            .concatMap(saved -> network.getUser(username))
            .retryWhen(errors -> errors
                    .doOnNext(ignored -> Log.d(TAG, "retrying..."))
                    .delay(2, TimeUnit.SECONDS)
                    .take(4)
                    .concatWith(Observable.error(new Throwable())))
            .onErrorResumeNext(throwable -> {
                    Log.d(TAG, Objects.requireNonNull(throwable.getMessage()));
                    return Observable.empty();
            })
            .subscribe(user -> Log.d(TAG, "next: " + user),
                       throwable -> Log.e(TAG, "error: " + throwable),
                       () -> Log.d(TAG, "completed")));
```