Return Early Pattern

Leonel Menaia Dev · Follow
Published in The Startup · 6 min read · Aug 5, 2020

When I learned about programming, the basic thought process that I had when creating a function was to validate the corresponding requirements until reaching the result.

```java
public String returnStuff(SomeObject argument1, SomeObject argument2) {
        if (argument1.isValid()) {
                if (argument2.isValid()) {
                        SomeObject otherVal1 = doSomeStuff(argument1, argument2)

                        if (otherVal1.isValid()) {
                                SomeObject otherVal2 = doAnotherStuff(otherVal1)

                                if (otherVal2.isValid()) {
                                        return "Stuff";
                                } else {
                                        throw new Exception();
                                }
                        } else {
                                throw new Exception();
                        }
                } else {
                        throw new Exception();
                }
        } else {
                throw new Exception();
        }
}
```

ConfusingCode.java hosted with ❤ by GitHub view raw

What can be observed in this approach?

- The nonlinear flow of the code is hard to follow because of the nested conditions.
- It is difficult to figure out the corresponding “else” for each “if” which makes the error handling confusing to read, especially when the “if” block is big.
- It is required to follow the flow of the code, navigating through nested “ifs”, to find the expected positive result.
- For the sake of this example, an exception is being thrown on the “else”. If the “else” didn’t terminate the execution, it would execute the rest of the code. This can lead to unnecessary errors.

It also includes a couple of anti-patterns:

- Else is considered smelly. When the condition is complicated, the “else” is twice, because the reader has to invert it; When the “if “ block is big, it is easy to forget the condition; Nested “If” and “else” confuse the reader.
- Arrow anti-pattern is when the code starts becoming shaped like an arrow because of nested conditions and loops.

Return Early

Let’s attempt to refactor the code with a different mindset.

> *Return early is the way of writing functions or methods so that the* ***expected positive result*** *is returned at the end of the function and the rest of the code terminates the execution (by returning or throwing an exception) when conditions are not met.*

This is accomplished by reverting the "if" conditions, doing the necessary error handling, and returning or throwing an adequate exception, finishing the execution of the function.

```
public String returnStuff(SomeObject argument1, SomeObject argument2){
        if (!argument1.isValid()) {
                throw new Exception();
        }

        if (!argument2.isValid()) {
                throw new Exception();
        }

        SomeObject otherVal1 = doSomeStuff(argument1, argument2);

        if (!otherVal1.isValid()) {
                throw new Exception();
        }

        SomeObject otherVal2 = doAnotherStuff(otherVal1);

        if (!otherVal2.isValid()) {
                throw new Exception();
        }

        return "Stuff";
}
```

Refactor.kt hosted with ❤ by GitHub view raw

There are a few things that are observable here:

- The code only has one level of indentation. It is possible to read it linearly.
- The expected positive result is quickly findable at the end of the function.
- Using this thought process, there is a bigger focus on finding the errors first and safely implementing business logic later, which avoids unnecessary bugs.
- The fail-fast mindset used is similar to Test-Driven Development, which makes the code easier to test.
- The function ends immediately on errors, avoiding the possibility of more code being executed without intention.

Design Patterns

While using the "return early" mindset the succeeding design patterns are followed.

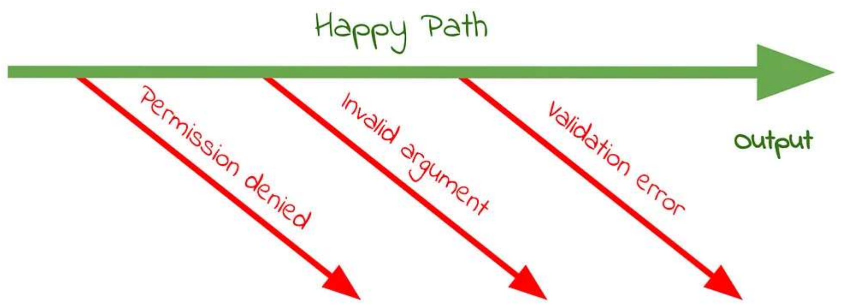
Fail Fast

Jim Shore and Martin Fowler created the concept of Fail Fast in 2004. This concept is the basis for the "return early" rule. While failing fast, the code is more robust because of the initial focus in finding the conditions where the code execution can terminate. With this approach, bugs are easier to find and fix.

Guard Clause

A guard clause is simply a check (the inverted "if") that immediately exits the function, either with a "return" statement or an exception. Using guard clauses, the possible error cases are identified and there is the respective handling by returning or throwing an adequate exception.

Happy Path



> *The happy path for a function would be where none of the validation rules raise an error, thus letting execution continue successfully to the end, generating a positive response.*

Using the "return early" approach, the code is read linearly, thus exposing the happy path. Using this pattern, it is not needed to lose time following a code flow to get to the goal. It is possible to use muscle memory to find it.

Bouncer Pattern

The bouncer pattern is a method to validate certain conditions by either returning or throwing an exception. It is particularly useful when the validation code is complex and can be used in multiple scenarios. It complements the "return early" pattern.

```
private void validateArgument1(SomeObject argument1){
        if(!argument1.isValid()) {
                throw new Exception();
        }

        if(!argument2.isValid()) {
                throw new Exception();
        }
}

public void doStuff(String argument1) {
        validateArgument1(argument1);
```

Open in app ↗

Search Write

Disadvantages

While the "return early" approach has positive points, it also has some fair criticisms which will now be brought to light.

Functions should only have one exit point

This coding rule dates back to Dijkstra's structured programming. This notion of Single Entry, Single Exit (SESE) comes from languages with explicit resource management, like C and Assembly.

In languages where resources are not or should not be managed manually, there is little to no value in adhering to the old convention. SESE often makes code more complex. It is a dinosaur that (except for C) does not fit well into most of today's languages. Instead of helping the understandability of code, it hinders it.

Resource Cleaning

High-level languages like Java and C# have garbage collection, but sometimes it is still needed to manage some resources manually. Thankfully, newer languages have the following concepts:

- "Try, catch, and finally" statements enable the use of a resource, catching any possible exceptions and then releasing the resource in the final block, making sure that there are no memory leaks.
- The "using" statement allows the use of a resource inside a block and automatically disposes of it after its usage, even if the termination is caused prematurely.

These concepts allow the use of the "return early" rule while also disposing of the resources used when terminating the execution of the function.

Logging and Debugging

One argument is that one single "return" is easier to debug since it is only needed to add one breakpoint to catch all exits from a function, and easier to log because it only requires a log at the end. This is not necessarily true.

Using the "return early" approach it is possible to throw exceptions right way. Debugging is much easier if the reason why the code is failing is obvious. A log can also be added before each termination to better inform the developer. If it is required to log all exits, in case of multiple "return" statements, it is possible to log after getting the value from the respective function.

Multiple exit points affect readability

200 lines of code function with various "return" statements sprinkled randomly over it is not a great programming style and it is not readable. But such a function wouldn't be easy to understand without those returns either. The bouncer pattern and the extract method pattern should be used to keep the size of the function within reasonable limits.

Code style is subjective

A **design pattern** is a general repeatable solution to a commonly occurring problem in software design. These are **conventions** that developers found over time that help to ease their work and should be used in the appropriate cases. However, some aspects of programming are subjective. Let's take a look at the following example:

```java
public String returnStuff(SomeObject argument) {
  if(!argument.isValid()) {
    return;
  }

  return "Stuff";
}

public String doStuff(SomeObject argument) {
  if(argument.isValid()) {
    return "Stuff";
  }
}
```

SometimesReturnEarlyIsSubjective.java hosted with ❤ by GitHub view raw

- On the first approach, there is more written code and complexity compared to the second approach. However, the code is prepared, using the "return early" mindset, for future improvements to the function. However, this way of thinking infringes on the KISS and YAGNI rules. **K**eep **I**t **S**imple **S**tupid and **Y**ou **A**ren't **G**onna **N**eed **I**t. It is easy to change the code to the "return early" pattern if it is needed in the future.

- The second approach is much more simple and readable. It goes straight to point and it would be my personal choice in this case.

Nevertheless, it is understandable why someone would use the first approach. In this case, arguing over the "correct way" to do it is losing important time.

Conclusion

The "return early" pattern is an excellent way to prevent functions from becoming confusing. However, this doesn't mean that it can be applied every time. Occasionally, during the complex business logic, it is inevitable to have some nested "ifs", even with the option of extracting the code to other functions.

The better approach is to align with the respective team, to share knowledge with them, to decide which patterns to use in each case, making sure that everyone has a similar mindset when programming. Developers spend more time reading code than writing it so it is essential to provide a positive experience for everyone.

Sources

Why should you return early?

When I started my journey as a computer programmer, I had written my code in various way. In most cases, I translated...

szymonkrajewski.pl

Where did the notion of "one return only" come from?

This notion of Single Entry, Single Exit (SESE) comes from languages with explicit resource management , like C and...

softwareengineering.stackexchange.com

The Single Return Law

I wrote this a few years ago (before 2012), about the idea that a method or function should have only one "exit point"...

www.anthonysteele.co.uk

Programming Software Engineering Software Development Coding

Learning To Code

Written by Leonel Menaia Dev

61 Followers · Writer for The Startup

Follow

Android Developer at Critical Techworks / BMW Group

More from Leonel Menaia Dev and The Startup

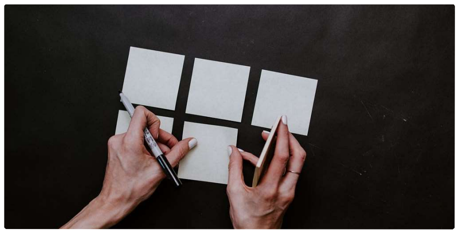
Leonel Menaia Dev in Level Up Coding

Unit Tests — Argument Captor with Mockito

Argument Captor is a Mockito feature used to capture arguments from functions called...

3 min read · Aug 12, 2020

55 1



Kurtis Pykes in The Startup

Don't Just Set Goals. Build Systems

The Secret To Happiness And Achieving More

· 9 min read · Dec 21, 2022

21K 341

Martina D. in The Startup

11 Companies Making $1M+ That Are [Practically] Bedroom Businesses

Leonel Menaia Dev in ITNEXT

Barista — Enjoyable Espresso Android UI Tests

Time to feed the ideation part of your brain with some real treats.

· 7 min read · Dec 31, 2023

2.4K 36

Barista makes developing UI tests faster, easier, and more predictable. Built on top of...

3 min read · Mar 11, 2022

103

See all from Leonel Menaia Dev

See all from The Startup

Recommended from Medium

```
{
  "name": "JSON is slow!",
  "blog": true,
  "writtenAt": 1695884403,
  "topics": ["JSON", "Javascript"]
}
```

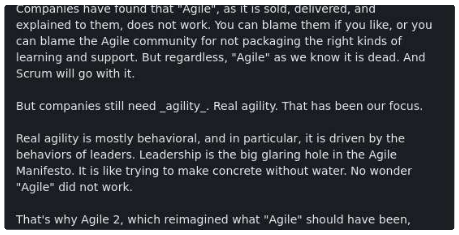
Vaishnav Manoj in DataX Journal

JSON is incredibly slow: Here's What's Faster!

Unlocking the Need for Speed: Optimizing JSON Performance for Lightning-Fast Apps...

16 min read · Sep 28, 2023

12.4K 144



Companies have found that "Agile", as it is sold, delivered, and explained to them, does not work. You can blame them if you like, or you can blame the Agile community for not packaging the right kinds of learning and support. But regardless, "Agile" as we know it is dead. And Scrum will go with it.

But companies still need _agility_. Real agility. That has been our focus.

Real agility is mostly behavioral, and in particular, it is driven by the behaviors of leaders. Leadership is the big glaring hole in the Agile Manifesto. It is like trying to make concrete without water. No wonder "Agile" did not work.

That's why Agile 2, which reimagined what "Agile" should have been,

Tamás Polgár in Developer rants

Agile has failed. Officially.

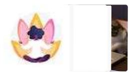
Either I'm a gifted oracle, and all of my friends are, or Agile really was just a stupid idea to...

2 min read · Dec 2, 2023

13.5K 422

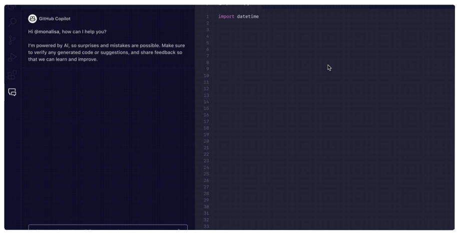
Lists

General Coding Knowledge
20 stories · 828 saves


Stories to Help You Grow as a Software Developer
19 stories · 739 saves

Coding & Development
11 stories · 400 saves

Leadership
42 stories · 204 saves



Jacob Bennett in Level Up Coding

The 5 paid subscriptions I actually use in 2024 as a software engineer

Tools I use that are cheaper than Netflix

· 5 min read · Jan 4

4.4K 55

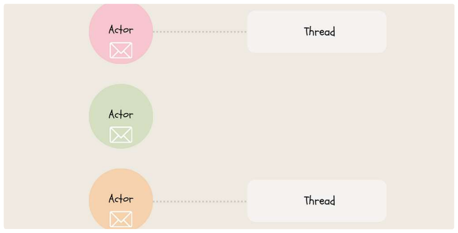
fatfish in JavaScript in Plain English

Interview: Can You Stop "forEach" in JavaScript?

there are 3 ways to stop forEach in JavaScript

· 5 min read · Aug 3, 2023

3.9K 80



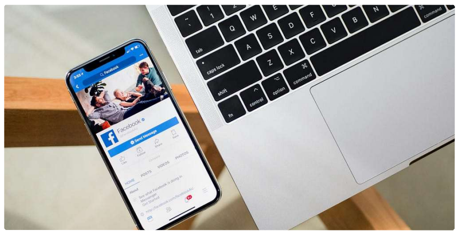
Nidhey Indurkar

How did PayPal handle a billion daily transactions with eight virtual...

I recently came across a reddit post that caught my attention: 'How PayPal Scaled to...

7 min read · Jan 1

3.4K 35



Andrew Zuo

Native Apps Are Dead — Get Over It

There is a shift happening in programming. I first noticed it in my Flutter application. I was...

· 5 min read · Nov 18, 2023

1.7K 62

See more recommendations