Better Validation Rules with ValidationResults

version 0.2

Sven Gorts

https://www.linkedin.com/in/svengorts/

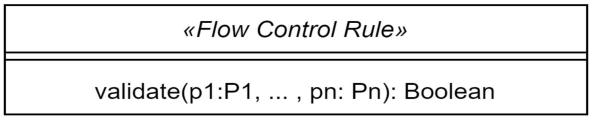
Introduction

In software development, we often write validations and checks to ensure adherence to a set of business rules. These rules, typically expressed in plain language, translate into validation code. Validation rules determine what is allowed in the application. They define the valid inputs, specify the allowed actions, and determine the available flows. We can define a Validation Rule as a class that represents the implementation of a business rule.

Our exploration begins by delving into the strengths and limitations of three frequently encountered rule implementations: the Flow Control Rule, the Guarding Rule, and the Feedback Rule. Studying these rules we'll find that each of these rules comes with its own set of advantages and disadvantages. We then explore how returning a ValidationResult allows creating a more general ValidationResult Rule which combines the strengths of the other rules while addressing their weaknesses.

The code examples in this article are in Kotlin because it allows for more compact examples compared to Java. Throughout the article, we will refer to Kotlin properties as methods to make the content accessible to a broader audience.

Flow Control Rule



A Flow Control Rule is a validator that evaluates its input parameters to a boolean result. As implied by its name, a Flow Control Rule facilitates decision-making and contributes to flow logic.

In practice, a Flow Control Rule typically begins by extracting the boolean expression of an if or while condition into a method. This offers two advantages: First, it allows for method reuse, reducing duplicated logic. Second, it encapsulates the conditional logic into a concept with a meaningful name. As the code matures, many of these flow control methods evolve into Flow Control Rule classes with a single validate() method. Expressed as such, our rules have now become a fully-fledged concept.

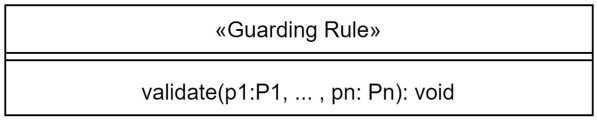
A Flow Control Rule is characterized by its validate() method, which returns a boolean value.

For example:

```
class PdfExtensionFileRule {
    fun validate(file: File): Boolean {
        return file.extension == "pdf"
    }
}
```

A Flow Control Rule has the advantage that it's easy and straightforward to implement. However, the price for its simplicity is a rule that can't provide any details about why a given validation fails. In particular, when rules are integrated into composite structures, understanding and debugging why a combination of validate() methods evaluates to false can become challenging. Encountering such complexity often indicates the necessity for detailed feedback messages, as users might be unaware of what goes wrong.

Guarding Rule



A Guarding Rule is a fail-fast validator with behavior closely related to a throwing guard clause.

During validation a Guarding Rule evaluates a required condition. When this condition is evaluated as a success, validation terminates quietly. Otherwise validation immediately throws a ValidationException to stop the program flow and report the failure.

Guarding Rules help safeguard your business logic, eliminating the need for repetitive checks throughout the codebase. An advantage of Guarding Rules is that the code to check the required condition and the code to generate a detailed failure message are closely together.

A Guarding Rule is characterized by a validate() method that returns void.

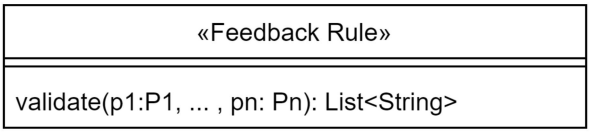
For example:

```
class PdfExtensionFileRule {
    fun validate(file: File) {
        if (file.extension != "pdf") {
            throw ValidationException(
                "The file ${file} is not a .pdf file"
            )
        }
    }
}
```

Guarding Rules are particularly well-suited when a ValidationException upon failure is desired. However, when the required condition expresses some kind of business rule the domain logic often also demands a boolean validation that can be used for control flow.

When used in composite structures the fail-fast nature of Guarding Rules quickly becomes an annoyance for end users. Consider an action that violates multiple business rules. Validation will fail providing feedback about the first failure only. When that issue is corrected and the user retries the action will fail again, only to reveal the next issue and so on and on. From a user perspective such a feedback mechanism is undesirable, which brings us to our next rule.

Feedback Rule



A Feedback Rule is a validator intended to provide clear and sufficiently detailed feedback messages to the user. Interpreting these feedback messages, the user, who lacks access to the source code, must be able to understand what business rule is violated, and even more importantly, why it is being violated.

When validating its input data a Feedback Rule can return one or more failure messages. Some implementations, out of scope for this article, also provide warning messages. As such a Feedback Rule is characterized by a validate() method that returns a List of messages.

The main focus of a Feedback Rule is to detect and report validation failures. When the validated data doesn't meet its required criteria a Feedback Rule must report all validation failures, in a format that provides the most helpful feedback to the user.

Helpful feedback involves a message that:

- Clearly expresses why a business rule is violated
- Includes relevant domain data like business codes and values
- Excludes pure technical data like internal ids

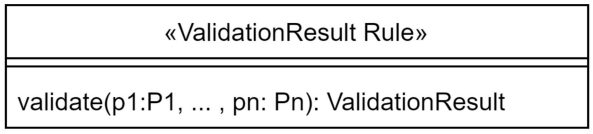
The Feedback Rule and Guarding Rule are behavioral opposites: A Guarding Rule immediately stops validation upon encountering the first failure and conveys information about that failure through an exception. In contrast, a Feedback Rule patiently collects all failures before providing a detailed report to the user.

For example:

```
class PdfExtensionFileRule {
    fun validate(file: File): List<String> {
        val messages = mutableListOf<String>()
        if (file.extension != "pdf") {
            messages += "The file ${file} is not a .pdf file"
        }
        return messages
    }
}
```

While an empty list intuitively signals a successful outcome of an operation, and a non-empty list suggests a failure, relying on these lists for control logic often causes confusion which may introduce bugs into the system. For example, adding a warning message leaves the message list nonempty.

ValidationResult Rule



A ValidationResult Rule is a validator that produces a ValidationResult which represents either success or failure. The idea is to separate the outcome of a validation from its immediate interpretation such as a boolean value or a ValidationException. While the ValidationResult Rule is responsible for the creation of a ValidationResult we leave the interpretation of the result to the ValidationResult itself.

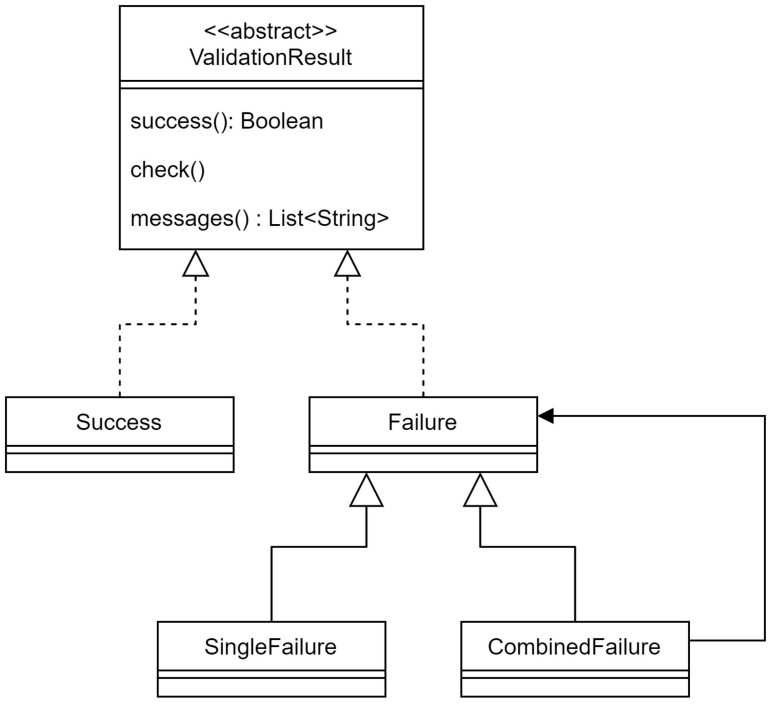
For example:

```
class PdfExtensionFileRule {
    fun validate(file: File):ValidationResult {
        if (file.extension != "pdf") {
            return ValidationResult
                .failure("The file ${file} is not a .pdf file")
        }

        return ValidationResult.success()
    }
}
```

The ValidationResult needs to support three kinds of result interpretation: flow control logic, guarding support and reporting of feedback messages. Given that a ValidationResult can be

either a Success or a Failure, and considering that a Failure may arise as a single failure or as a combination of failures, the class diagram of a possible implementation could look like this:



Let's start with flow logic. The availability of a success() method and, optionally a failure() method, allows for the result to be used in flow control. Alternatively, in certain implementations, the Success and Failure class types can serve this purpose as well. Implementing control flow now looks like this:

```
if (pdfRule.validate(file).success) {
   println("... processing the .pdf file")
}
```

For guarding we have the check() method which asserts that the result is a Success; otherwise, it throws a ValidationException. Given that exceptions have only a single message, we will need to do some aggregation of the messages before throwing. Adding a guard clause now becomes a single line:

```
pdfRule.validate(file).check()
```

Reporting feedback is supported by means of an messages() method, which returns a list of failure messages. For the most basic implementations, a list of strings suffices. However in an international context the returned list will be a list of translatable messages. Where needed messages can be extracted:

```
val messages = pdfRule.validate(file).messages
```

Let's examine how the example ValidationResult implements the desired behavior. For example:

```
sealed class ValidationResult {
   abstract val success: Boolean
   abstract val messages: List<String>
   abstract fun check()

   object Success : ValidationResult() {
       override val success: Boolean = true
       override fun check() = Unit
       override val messages: List<String> = listOf()
   }

   abstract class Failure : ValidationResult() {
       override val success: Boolean = false
       override fun check() {
           throw ValidationException(messages.joinToString(", "))
       }
   }

   data class SingleFailure(private val message: String) : Failure() {
       override val messages: List<String>
           get() {
               return listOf(message)
           }
   }

   data class CombinedFailure(val failures: List<Failure>) : Failure() {
       override val messages: List<String>
           get() {
               return failures.flatMap { it.messages }
           }
   }

   companion object {
       fun success() = Success
       fun failure(message: String) = SingleFailure(message)
       fun combine(vararg validationResults: ValidationResult)
           : ValidationResult {
           val failures = validationResults
               .filterIsInstance<Failure>()

           return if (failures.isNotEmpty()) {
               CombinedFailure(failures)
           } else {
               Success
           }
```

```
        }

        fun combine(validationResults: Collection<ValidationResult>)
            : ValidationResult {
            return combine(*validationResults.toTypedArray())
        }
    }
}
```

In addition to the result interpretation methods the ValidationResult class provides factory methods for the creation of ValidationResults in our rule implementations. The success() method creates a Success which is always without a message. The failure() method creates a single Failure instance. The combine() method aggregates multiple ValidationResults into a single ValidationResult instance.

Conclusion

In this article, we explored three common validation rule types: Flow Control, Guarding, and Feedback Rules. Examining the limitations of each, we discovered that by adopting a ValidationResult, which distinguishes the outcome from its interpretation, we can create a more general rule type—the ValidationResult Rule. The ValidationResult Rule supports each of the use cases for flow control, guarding, and detailed feedback. In a validation-rich codebase, implementing the business rules as Validation Result Rules is definitely worth considering.