Spring 5-18-2021

Asynchronous Validations using Programming Contracts in Java

Rahul Shukla

Asynchronous Validations using Programming Contracts in Java

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Rahul Shukla

May 2021

The Designated Project Committee Approves the Project Titled

Asynchronous Validations using Programming Contracts in Java

by

Rahul Shukla

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2021

Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Mike Wu	Department of Computer Science

ABSTRACT

Asynchronous Validations using Programming Contracts in Java

by Rahul Shukla

Design by Contract is a software development methodology based on the idea of having contracts between two software components. Programming contracts are invariants specified as pre-conditions and post-conditions. The client component must ensure that all the pre-conditions are satisfied before calling the server component. The server component must guarantee the post-conditions are met before the call returns to the client component. Current work in Design by Contract in Java focuses on writing shorthand contracts using annotations that are processed serially.

Modern software systems require a lot of business rules validations on complicated domain objects. Often, such validations are in the form of a chain of independent tasks that need to be validated one after another. These tasks are computation-intensive and often involve numerous database calls and API calls over the web.

This paper presents a validation rule engine framework, *Rule4j* to facilitate writing such business rules with the help of programming contracts in Java. The contracts are organized in a hierarchy similar to the Racket programming language. The programmer can specify the business rules in the form of a series of higher-order contracts that form a chain. These chains of contracts are validated concurrently and asynchronously to present a final validation result to the programmer. A sample scenario of trade execution is used to demonstrate the performance gain and maintainability of the framework. The experiments conducted show that validations executed using Rule4J run four times faster than the traditional approach. A clear separation of business logic and business validations for the trade execution scenario was achieved using Rule4J.

ACKNOWLEDGMENTS

I want to thank my advisor Dr. Thomas Austin, for his continuous guidance and motivation throughout this project. Dr. Thomas Austin taught me CS 252, which introduced me to Design by Contracts and functional programming. This project builds on the concepts taught in the course.

I would also like to thank the committee members Dr.Chris Pollett and Dr. Mike Wu for their support.

Lastly, I would like to thank my friends and family, for their endless support in the last two years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

The software construction process is getting complicated by the day and has drastically matured over the last two decades. The scale on which modern software operates is enormous. Building the highest quality software applications which are scalable and maintainable is increasingly difficult. According to one of Lehman's laws of software evolution, a software system will continue to decline in quality, if its design is not maintained and adapted to new operational needs.[1]

Software must cater to the dynamic requirements of the business. The cost of requirements grows at an average of 2 percent, from a software's inception to its completion. The total cost of new or modified requirements can go up to 50 percent of the original estimate.[2] The National Health System (NHS) Connecting for Health system is described as one of the biggest software failures ever. The NHS system was supposed to be one of the largest civilian software whose development spanned over 9 years. Changing requirements due to complex regulations governing the health care system was one of the major impediments to NHS system's development. The project cost offshoot from an initial estimate of £6.4 billion to £11 billion and ultimately abandoned after 9 years from its inception.[3, 4]

Subject Matter Experts (SMEs), Product Owners, and Business Analysts typically will be involved in the design stage of the requirements as they have been working in the project domain for a long time. They have the domain knowledge and the business logic that will form the backbone of the software. The programmer needs to have a good understanding of the domain and business logic from the SMEs, to construct an efficient solution. The business rules in most of the domains change frequently, and due to the constant pressure of frequent releases, these business validations are poorly translated into code. The business validations are scattered throughout the code-base. The SMEs are at the behest of the software developers, to understand the current business logic.

Therefore, there is an increasing need for frameworks that expedite the organization and maintenance of business validations. In [5] the authors state that 70% of the total time of Software Development Life Cycle (SDLC) is consumed in maintenance. Having a framework to organize the validations would greatly increase the development speed and decrease the time required to debug and enhance the existing system.

1.2 Problem Statement

One of the main problems tormenting modern software construction is separating the business rules and validations from the core business logic and underlying entities. Highly effective software is the one that caters to the changes effectively without compromising on the code quality, readability. Decoupling software validations from the core business logic promote re-usability and readability. Decoupling also facilitates the addition of new validations and modification of existing validations.

Design by Contract [6] is a software development methodology that was introduced by Bertrand Meyer in his design of the Eiffel programming language. The components interacting within the software system adhere to contracts. The contracts specify what the server component expects from the client components and vice-versa. Design by Contract help to construct bug-free software and also provide an alternative to defensive programming.

The goal of this project is to create a validation rule engine *Rule4j* in Java based on the principles of Design by Contract. Rule4j will facilitate the decoupling of business validations and business logic by the specification of validations as programming contracts. The programming contracts are processed asynchronously. The library allows the programmer to define a series of dependent validations as a contract chain, which is validated in parallel. The programming contracts are organized in a hierarchy similar to contracts in Racket programming language. The hierarchy enables the construction of higher-order contracts from existing contracts.

The Rule4j framework is a Maven [7] project that is built using the Guice [8] dependency injection framework. Rule4j allows better organization of business validations, as contracts that are organized in factory classes. These contracts can be reused to write higher-order contracts. The preconditions and postconditions are specified using contract chain definitions using custom annotations on the method under validation.

Prevailing Design by Contract libraries in Java only cater to primitive checks and are not fit for complex computations. Rule4j follows a different approach to contract definition and execution compared to the other programming contract libraries in Java. Complex business validations that are compute-intensive and memory-intensive are written as contracts in Rule4j. A series of contracts can be specified as contract chains, which are then executed in parallel for maximum performance.

Rule4j enables the programmer to wholly concentrate on the business logic without the concern of the execution of contracts. Rule4j executes the chains of validations in parallel and asynchronously. Chapter 5 demonstrates the performance benefits that arise by using Rule4j to perform business validations over the conventional methods.

CHAPTER 2

Literature Review

This chapter introduces some of the attributes of software maintainability in the first section. The second section explains maintainability metrics, coupling, and cohesion in software construction. The last section lists down and briefly describes some of the existing programming contract libraries in Java.

2.1 Software Maintainability

The quality standard ISO/IEC 25010:2011 [9] defines software maintainability as "the degree of effectiveness and efficiency by which a product or system can be modified by the intended maintainers". Therefore it is important to understand the impact of software maintainability while designing the Validation Rule Engine. Below, is the list of all the characteristics of maintainability as described in [10] and how the Validation Rule Engine adheres to these characteristics.

- *Flexibility* - Flexibility is defined as the ease with which the architecture of the system can be changed to accommodate new requirements and environments. The Rule4j framework is independent of the business domain and can be easily plugged into existing frameworks. Programming to interfaces along with Dependency Injection is used as much as possible to keep the components loosely coupled.
- *Modifiability* - Modifiability is defined as the ease with which changes can be made to the existing system without adding errors and diminishing the existing quality. A highly modifiable software system is associated with low costs to accommodate changes. Rule4j can be easily modified to support new annotations, contract types, and other cross-concerns.
- *Extensibility* - Extensibility is defined as the system's ability to add new features without affecting the existing features. A highly extensible software comprises

various components, which high coupling. Any changes to one of the components do not affect the other components. Rule4j promotes writing validations as contracts that can be organized in factory classes, the addition of any new validations does not impact existing validations.

- *Portability* - Portability is the ability of the software to run on diverse systems. Highly portable software is not tightly coupled to the underlying hardware. Rule4j is a framework built in Java, therefore it can run different platforms. The thread pool configurations, powering the rule engine can be changed on the startup for optimum performance.
- *Reusability* - Reusability is the degree to which the underlying software modules can be reused in building other components. This is the core focus of in the deisgn of Rule4j. The validations are decoupled from the core business logic and organized separately in factory classes. This enables high reusability, as a same contract could be used in multiple contract chains to validate different use-cases, avoiding code duplication.
- *Integrability* - Integrability is the ability of the software to enable different components of the system to work accurately together as a single component. The contracts written using Rule4j can be combined together easily to form different kinds of contract chains. Rule4j also promotes the wrapping of existing contracts to create higher-order contracts hence demonstrating high integrability.
- *Testability* - Testability is the degree to which test criteria can be established for the given system and the ease at which tests can be written to verify the criteria. Since the contracts in Rule4j are written independently, it is very easy to write Unit Tests to check the correctness of the validations promoting Test Driven Development (TDD) [11].

These attributes are summarized in the Figure 1.

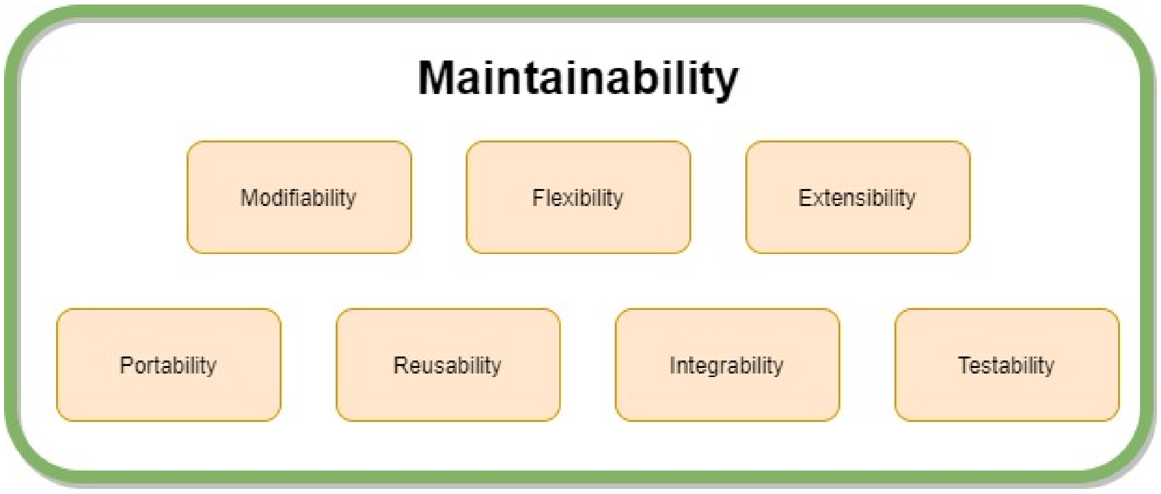


Figure 1: Attributes of Maintainability

2.2 Maintainability Measures

Software maintainability is challenging to measure as it depends on various factors, which are not technical. Razina and Janzen [5] present two maintainability measures while demonstrating the effectiveness of dependency injection to make the software more maintainable. This section review the two maintainability measures cohesion and coupling.

2.2.1 Coupling

Coupling is the interdependence of modules in a system. Two objects are said to be tightly coupled if they interact with each other via instance variables or methods. [12] Increased coupling is associated with poor readability and maintainability. The parts of the system that are coupled become more sensitive to changes made in other parts of the system. One such measure of coupling for a class can be the number of classes it interacts with. Razina and Janzen [5] demonstrate that using dependency injection reduces the coupling between interconnected systems. The authors also point out that lower coupling leads to testable components and increased flexibility, thereby increasing maintainability.

Cerny and Donahoo [13] summarize their four years of experience in building large enterprise software and present a framework to handle business rules. The authors give multiple examples of how Aspect Oriented Programming (AOP) [14] helps to reduce the coupling between involved components. The authors also presented a framework that decouples the business rules by centralizing them in a separate component. This component stores all the business rules connected to the domain object. They also decouple and centralize the exception handling to a separate component. The authors contrast this approach to business logic integration tool Drools [15] which has a significant performance overhead.

2.2.2 Cohesion

Chidamber and Kemerer [12] define Cohesion is the degree of similarity between components in software. There should be high cohesion between methods of a class. If there is a lack of cohesion between methods of a class, it needs to be refactored into separate classes. Low cohesion in a class would considerably increase the complexity and decrease the maintainability of the system. The programmer will have to take care of interactions between various other components before making changes to a particular feature, thereby increasing the chances of errors and increasing the development costs. Low cohesion also decreases the readability, as similar modules are spread across the codebase. Therefore it is highly desirable to have a system with high cohesion.

Chidamber and Kemerer [12] present a widely used metric, Lack of Cohesion in Methods (LCOM), that is defined as follows:

For a class $\mathtt{C}$ with methods $\mathtt{M_1}, \mathtt{M_2}, \mathtt{M_3}, ..., \mathtt{M_n}$

Let $\{\mathtt{I_i}\}$ be the instance variables of class

$\mathtt{C}$ that are being used by $\mathtt{M_i}$

$\mathtt{LCOM(C)} = \{\mathtt{I_1}\} \cup \{\mathtt{I_2}\} \cup \{\mathtt{I_3}\} \cup ... \cup \{\mathtt{I_n}\}$

Goswami [16] presents a Rule Engine for automation of validations of a complex System Configuration File to configure a Real-Time Measurement System. The framework successfully separated the business rules from the domain objects and enabled the business users to directly specify the rules. The author created a Rule Engine to process the rules serially. The Rule Library contains all the rules, that need to be validated. This leads to a higher cohesion and a more maintainable system. The rules are defined as a grouping of an object, attribute, and constraint. The framework also enables the composition of rules using OrRule, AndRule, NotRule, and IfThenRule. This enables high cohesion, as the existing rules can be reused and organized together.

2.3 Design By Contract

Bertrand Meyer [6], presents a set of methodological guidelines known as Design by Contract for constructing reliable software. The approach is different from defensive programming, which handles each and every edge case and hence is redundant in nature. The code consequently is verbose and the actual business logic is obfuscated with error checking code. An approach based on the notion of a contract between the client and the supplier in software construction is presented.

If the execution of a function depends on another function call, it is necessary to establish the relationship between the client and the supplier(contract). Such conditions are expressed using assertions. Assertions can be preconditions and postconditions, that apply to individual routines. Assertions are boolean conditions separated by semicolons. Any failure(runtime violation) in these assertions would indicate the presence of a software bug. If a precondition has failed, then the caller did not observe the contract and if a postcondition fails then the current method did not work as expected. The author then emphasized that assertions should not be then handled separately in the method code similar to defensive programming. Assertions, on the contrary, specify all such cases beforehand, which should be met to avoid any runtime violations.

In the context of object-oriented languages, the author introduces "class invariants". Class invariants are properties/assertions that apply to all instances of the class. Every constructor/creational-method of the class. Class invariants must be observed by all the methods of the class. Here the invariants apply to the entire class instead of one particular routine. We can use various levels of assertion monitoring in classes. It is expected to turn on assertion monitoring on all levels, as assertion violations are manifestations of software bugs. In the case of inheritance, the absence of precondition and postcondition means that the overridden(re declared) method retains the parent class assertions. We don't use the forms require and ensure in case of redeclaration instead we use `orElse` and `then`. Invariants are always passed to subclasses(descendants).

A more disciplined error handling approach, by including a `rescue` clause is introduced. Any exception in the method will start the execution of the rescue clause, which can contain a retry instruction that will attempt a resumption. If an explicit rescue clause is not present, then a default rescue is assumed.

2.4 Existing Programming Contracts Libraries in Java

This section explores some of the existing programming contract libraries in Java.

2.4.1 jContractor: A Reflective Java Library to Support Design By Contract

Karaorman *et al.* [17] present a library to support Design by Contract specifications. The library uses Java instrumentation to enforce the preconditions, postconditions, and class level invariants. The library specifies a set of naming conventions to specify the preconditions, postconditions, and class invariants. The jContracter library uses a custom class loader to instrument the concerned classes. The class loader searches for patterns in the loaded classes, and the with the help of Java Reflection. jContracter is purely a library-based approach without any modification to JVM or pre-processor. If the class contains contract methods, it will instrument the class bytecode, and add the wrapper contract methods to public methods to check violations. Otherwise, the original bytecode is left unmodified. This is illustrated in figure [17, Figure 2].

2.4.2 Design by Contract with JML

Leavens and Cheon [18] introduce the Java Modelling Language(JML) to write programming contracts in Java. JML is a specification language for Java that requires a JML compiler (`jmlc`). JML compiler (`jmlc`) is an extension of the Java compiler, which can compile Java code with JML specifications into bytecode. This is unlike the approach in [17] which is purely a library-based approach that relies on Java Reflection to instrument the bytecode. JML specifications are written with the help of special annotation comments, which start with `@`. JML has various clauses like `requires`, `ensures`, `\result` and `signals`, that can be used to write contracts. The author relates DBC contract specification to informal documentation, as it does not have to be detailed, but specify the assumptions and expectations. Writing the specifications with JML rather than comments is better as they can be checked with the help of `jmlc`.

2.4.3 Contract4J

Wampler [19] introduces Contract4j, a Design by Contract library written using AspectJ [20] which is an AOP extension for Java Programming Language. The library supports programming contracts using JSR annotations as well as using a JavaBeans-like method naming convention. Unlike [17, 18] this approach uses AOP to intercept and execute the contracts. The invariants are specified using annotations on the methods, and fields. Contract4j annotations like `@Contract`, `@Post`, and `@Pre` are used to define invariants that are strings. The contracts are specified on the interfaces. The invariants are evaluated at runtime as Java expressions using a runtime evaluator, Jakarta Jexl interpreter [21]. The author also provides another implementation of Contract4j which is called ContractBeans. In this approach instead of the annotations, we use the JavaBeans-like naming convention to specify the preconditions and postconditions. Instead of an interface, an abstract class is used where the preconditions and postconditions are defined. However, the author points to the drawback of the ContractBeans approach, as it has more verbosity and has a significant runtime overhead compared to the annotation-based approach.

2.4.4 Other Libraries

Rajkumar [22] presents a programming contract library in Java following a similar approach as [19]. The contracts are written with the help of custom annotations which are then intercepted and executed using AspectJ advice. The invariants are specified as strings in the custom annotations similar to the approach in [19]. However, the author also supported Java 8 lambda expressions to specify the invariants. The lambda expression is provided as a string expression, which is converted to a lambda object using the LambdaFromString library [23]. The author points to the limitation of the library in parsing complicated objects.

Dixit [24] follows a similar approach to [19, 22] to define contracts using custom annotations and intercept them using AspectJ join points. However, the author uses Prolog to write the contracts using facts and rules. The author credited this to Prolog's declarative syntax, which made it easier to write contracts. The library queries the Prolog files to validate the contracts which are specified in the annotations.

CHAPTER 3
Contract Hierarchy in Rule4j

The business validations are written with the help of programming contracts in Rule4j. These contracts are organized in a hierarchy similar to the Racket programming language. This chapter briefly explores Racket's contract system and followed by the design of the contract system in Rule4j. I also present examples of some higher-order contracts that can be constructed using the framework.

3.1 Contract Hierarchy in Racket

Racket helps to establish boundaries between modules with the help of contracts. Whenever a value crosses the boundary, a contract check is triggered to ensure that the modules adhere to the contract. [25] The contract check can be performed immediately when a value crosses the boundary, or it can be delayed in the case of function contracts. Contracts can be thought of as predicate functions that check the input arguments and return a Boolean value depending on an criteria. Contracts in Racket ensure only the values that meet the requirements cross the module boundary. There are two classes of contracts in Racket [26] -

- *Flat Contracts*

  These contracts can be thought of as predicate functions that are checked immediately at the boundaries. Below is an example of flat contract in Racket, the function `num-between-1-100` checks whether a value is between 1 and 100. This contract is used to guard `fun-sqr` which squres the input number. We can see this in action in the Listing 3.1. The function `flat-contract?` is used to identify a flat contract.

Listing 3.1: Flat Contract in Racket

```
(define (num-between-1-100? i) (<= 0 i 100))
(print (num-between-1-100? 50))
#t
(print (num-between-1-100? 101))
#f
(print (flat-contract? num-between-1-100?))
#t
(define/contract (fun_sqr i)
    (-> num-between-1-100? any)
    (* i i)
    )

(print (fun_sqr 50))
2500
(print (fun_sqr 150))
fun_sqr: contract violation}
  expected: num-between-1-100?
  given: 150
  in: the 1st argument of
      (-> num-between-1-100? any)
```

- *Higher-Order Contracts*

  These contracts have wrappers that delay check. These wrap flat contracts with wrapper functions and delay the contract checking. Higher Order contracts are further classified into two classes.

  - *Chaperone Contracts*

    Chaperone contracts wrap the given flat contract to delay check. The wrapped values must behave the same as they would without the chaperoning. They can be used for checking the inputs to the underlying flat contract, to perform logging operations, or any other actions that are not guaranteed to change the underlying values. A chaperone can only restrict the behavior of the objects they wrap, it must raise an exception or return the same value as the underlying flat contract.[27] The example below showcases a chaperone contract `num-between-1-100?+log` which wraps or chaperones the flat-contract `num-between-1-100?` in Listing 3.1. The chaperone adds delay by printing the input and output to the underlying contract, but return the same value as the underlying contract.

Listing 3.2: Chaperone Contract in Racket

```
(define num-between-1-100?+log
(chaperone-procedure num-between-1-100?
       (λ(x)
           (printf "called with input - ~s\n" x)
           (values (λ(res)
                     (printf "returned - ~s\n" res)
                     res)
                 x))))
(define/contract (fun_sqr i)
    (-> num-between-1-100?+log any)
    (* i i)
    )
(print (fun_sqr 50))
called with input - 50
returned - #t
2500
(print (fun_sqr 150))

called with input - 150
returned - #f

fun_sqr: contract violation
  expected: num-between-1-100?
  given: 150
  in: the 1st argument of
      (-> num-between-1-100? any)

 (print (chaperone-of? num-between-1-100?+log num-between-1-100?))
 #t
```

– *Impersonator Contracts* -

Impersonator Contracts can change the values which they wrap. They can change the values to the underlying contract with a completely new value. In the example below the flat-contract in example 3.1 is wrapped with the impersonator contract `num-between-1-150?`. The impersonator completely changes the underlying behaviour by modifying the input argument. The Listing 3.3 demonstrates this.

Listing 3.3: Impersonator Contract in Racket

```
(define num-between-1-150?
(impersonate-procedure num-between-1-100?
       (λ(x)
           (printf "called with input - ~s\n" x)
           (values (λ(res)
                   (printf "returned - ~s\n" res)
                   res)
               (- x 50)))))



(define/contract (fun_sqr i)
   (-> num-between-1-150? any)
   (* i i)
   )
(print (fun_sqr 150))
called with input - 150
returned - #t
22500
(print (impersonator-of? num-between-1-150? num-between-1-100?))
#t
```

With the help of Chaperones and Impersonators API in Racket, higher-order contracts can be implemented in the Racket programming language by using proxy patterns. [27]

3.2 Design of Contract Hierarchy in the Rule4j

The contract hierarchy in the Rule4j Rule Engine is influenced by the contract hierarchy of the Racket programming language. Since Java is a strongly typed language, this served as a limitation while designing the contract hierarchy in Rule4j. Due to this, the contracts in Rule4j have been capped to have a maximum arity of 2. A validation written in Rule4j can operate on two arguments at max.

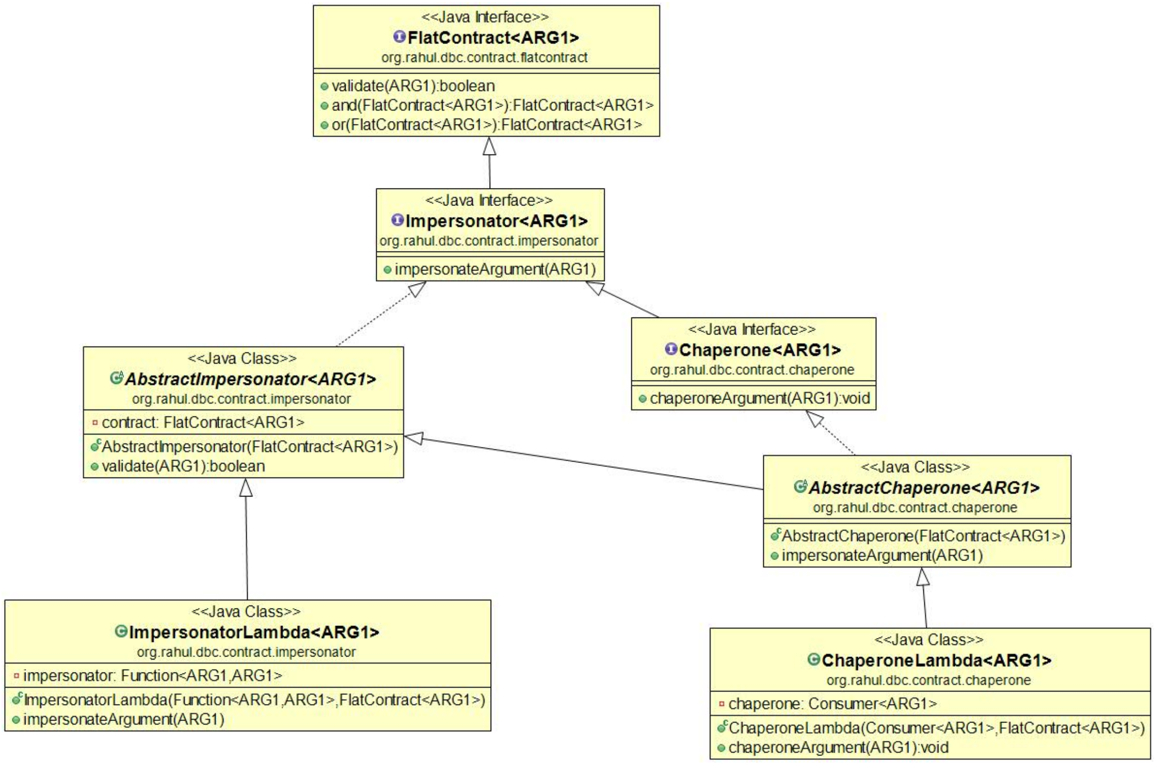
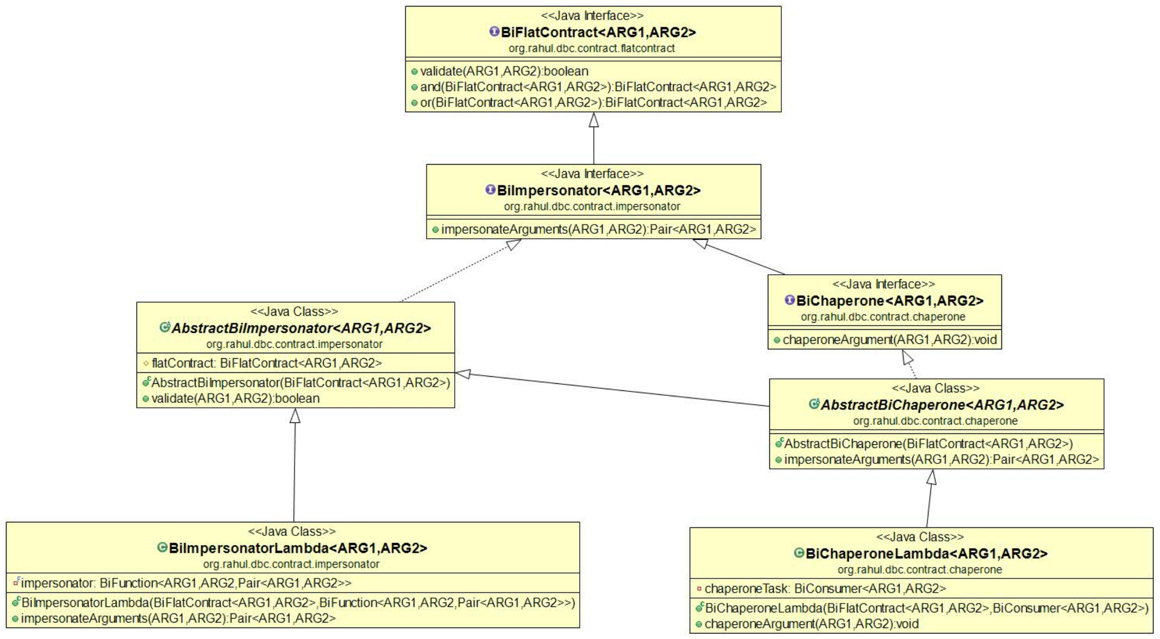

<<Java Interface>>
FlatContract<ARG1>
org.rahul.dbc.contract.flatcontract
validate(ARG1):boolean
and(FlatContract<ARG1>):FlatContract<ARG1>
or(FlatContract<ARG1>):FlatContract<ARG1>
<<Java Interface>>
Impersonator<ARG1>
org.rahul.dbc.contract.impersonator
impersonateArgument(ARG1)
<<Java Class>>
AbstractImpersonator<ARG1>
org.rahul.dbc.contract.impersonator
contract: FlatContract<ARG1>
AbstractImpersonator(FlatContract<ARG1>)
validate(ARG1):boolean
<<Java Interface>>
Chaperone<ARG1>
org.rahul.dbc.contract.chaperone
chaperoneArgument(ARG1):void
<<Java Class>>
AbstractChaperone<ARG1>
org.rahul.dbc.contract.chaperone
AbstractChaperone(FlatContract<ARG1>)
impersonateArgument(ARG1)
<<Java Class>>
ImpersonatorLambda<ARG1>
org.rahul.dbc.contract.impersonator
impersonator: Function<ARG1,ARG1>
ImpersonatorLambda(Function<ARG1,ARG1>,FlatContract<ARG1>)
impersonateArgument(ARG1)
<<Java Class>>
ChaperoneLambda<ARG1>
org.rahul.dbc.contract.chaperone
chaperone: Consumer<ARG1>
ChaperoneLambda(Consumer<ARG1>,FlatContract<ARG1>)
chaperoneArgument(ARG1):void

Figure 2: Single Argument Contract Hierarchy

Figure 2 and 3 shows the hierarchy of contracts that can be used to write programming contracts in Rule4j. At the base of the contract hierarchy, we have a `FlatContract<T>` interface, which has a method `validate`, that takes input arguments and returns a `boolean`. All the other contracts extend from the `FlatContract<T>` interface. Here `T` represents the generic type argument in Java. `Impersonator<T>` interface extends the `FlatContract<T>` interface and exposes an-



<<Java Interface>>
BiFlatContract<ARG1,ARG2>
org.rahul.dbc.contract.flatcontract
validate(ARG1,ARG2):boolean
and(BiFlatContract<ARG1,ARG2>):BiFlatContract<ARG1,ARG2>
or(BiFlatContract<ARG1,ARG2>):BiFlatContract<ARG1,ARG2>
<<Java Interface>>
BiImpersonator<ARG1,ARG2>
org.rahul.dbc.contract.impersonator
impersonateArguments(ARG1,ARG2):Pair<ARG1,ARG2>
<<Java Interface>>
BiChaperone<ARG1,ARG2>
org.rahul.dbc.contract.chaperone
chaperoneArgument(ARG1,ARG2):void
<<Java Class>>
AbstractBiImpersonator<ARG1,ARG2>
org.rahul.dbc.contract.impersonator
flatContract: BiFlatContract<ARG1,ARG2>
AbstractBiImpersonator(BiFlatContract<ARG1,ARG2>)
validate(ARG1,ARG2):boolean
<<Java Class>>
AbstractBiChaperone<ARG1,ARG2>
org.rahul.dbc.contract.chaperone
AbstractBiChaperone(BiFlatContract<ARG1,ARG2>)
impersonateArguments(ARG1,ARG2):Pair<ARG1,ARG2>
<<Java Class>>
BiImpersonatorLambda<ARG1,ARG2>
org.rahul.dbc.contract.impersonator
impersonator: BiFunction<ARG1,ARG2,Pair<ARG1,ARG2>>
BiImpersonatorLambda(BiFlatContract<ARG1,ARG2>,BiFunction<ARG1,ARG2,Pair<ARG1,ARG2>>)
impersonateArguments(ARG1,ARG2):Pair<ARG1,ARG2>
<<Java Class>>
BiChaperoneLambda<ARG1,ARG2>
org.rahul.dbc.contract.chaperone
chaperoneTask: BiConsumer<ARG1,ARG2>
BiChaperoneLambda(BiFlatContract<ARG1,ARG2>,BiConsumer<ARG1,ARG2>)
chaperoneArgument(ARG1,ARG2):void

Figure 3: Bi Argument Contract Hierarchy

other method `impersonateArguments` which can be used to modify the input arguments to the underlying contract. This facilitates the construction of impersonators that wrap the underlying flat contracts and change the behavior of the values during run-time.

The `Chaperone<T>` interface extends `Impersonator<T>` interface and exposes method `chperoneArguments` which can be used to perform checks and other operations on the input arguments. Chaperones wrap underlying contracts without changing the input values to the underlying contract, hence they do not change the behavior of the underlying contract like an impersonator contract.

The contract library facilitates writing contracts using functional programming. All the interfaces are functional interfaces and a lambda expression be used to represent a contract. `ImpersonatorLambda<T>` and `ChaperoneLambda<T>` facilitate wrapping up contracts with the help of the `Consumer` and `Function` functional interface introduced in Java 8. In the next section, we will explore how to build contracts using the contract hierarchy.

3.3 Higher-Order Contracts

The previous section explored the contract hierarchy of the Rule4j framework. In this section we showcase examples of higher-order contracts that can be written using Rule4j. The contracts can be categorized into 3 types, similar to the Racket Programming Language -

- *Flat Contract* -

  A flat contract can be written by implementing the `FlatContract<T>` interface. Listing 3.4 below showcases how the flat contract in Listing 3.1 can be written in Java. The first approach uses the traditional approach by implementing the `FlatContract` interface using an anonymous inner class, whereas the the second approach takes advantage of Functional Interface in Java. The second approach clearly is less verbose and more intuitive to write.

  Listing 3.4: Flat Contract in Java

  ```
  FlatContract<Integer> traditionalContract = new FlatContract<Integer>() {
      @Override
      public boolean validate(Integer data) {
          return data >= 0 && data <= 100;
      }
  };
  FlatContract<Integer> numBetween1And100 = i -> i >= 0 && i <= 100;
  System.out.println(numBetween1And100.validate(50)); // true
  System.out.println(numBetween1And100.validate(150)); // false
  ```

- *Chaperone Contract* -

  Listing 3.5 showcases a chaperone contract similar to Listing 3.2. The flat contract in Listing 3.4 is wrapped in the chaperone contract `numBetween1And100AndLog`. The `LambdaChaperone` constructor will only accept a `Consumer`, hence not allowing the programmer to change the input values to the underlying flat contract.

Listing 3.5: Chaperone Contract in Java

```
Chaperone<Integer> numBetween1And100AndLog = new ChaperoneLambda<>(i ->
    System.out.println(i),
        numBetween1And100);

System.out.println(numBetween1And100AndLog.validate(50)); //50 true
```

- *Impersonator Contract* -

  Listing 3.6 below showcases how the impersonator contract in Listing 3.3 contract can be implemented in Java. The flat contract in Listing 3.4 is wrapped in `numBetween1And150`. The `ImpersonatorLambda` accepts 2 arguments, the first is a lambda function that can modify the input values and the second argument is the underlying flat contract.

Listing 3.6: Impersonator Contract in Java.

```
Impersonator<Integer> numBetween1And150 =
    new ImpersonatorLambda<>(i -> (i > 50) ? (i - 50) : i
    , numBetween1And100);

System.out.println(numBetween1And150.validate(150)); //true
System.out.println(numBetween1And150.validate(200)); // false
```

Listing 3.7 demonstrates how a cached contract can be built using the contract library. Rule4j will cater to business validations that will involve a Database call or an API call and the results of some such validations can be cached. Since the business validations in the rule engine is written using contracts, there is a requirement to cache the result of the contract, to optimize performance. Listing 3.7 demonstrates how a requirement to cache can be solved by building a higher order contract using the library.

Listing 3.7: Implementation of a Single Argument Cached Contract

```java
public class SingleArgCachedContract<ARG1> implements FlatContract<ARG1> {
    private FlatContract<ARG1> underlyingContract;
    private Map<ARG1, Boolean> cache;
    private Lock lock;
    public SingleArgCachedContract(final FlatContract<ARG1> underlyingContract) {
        this.underlyingContract = underlyingContract;
        this.lock = new ReentrantLock();
        this.cache = new HashMap<>();
    }
    @Override
    public boolean validate(ARG1 data) {

        if (this.cache.containsKey(data)) {
            return this.cache.get(data);
        }
        boolean result = this.underlyingContract.validate(data);
        this.lock.lock();
        try {
            this.cache.put(data, result);
        } finally {
            this.lock.unlock();
        }
        return result;
    }
}
```

`SingleArgCachedContract` is a higher order contract that can be used to used to cache a `FlatContract`. The underlying `FlatContract` is taken as a constructor argument. Single Argument Cached Contract can help to gain performance benefits by wrapping other contracts, which is demonstrated in Listing 3.8 and Listing 3.9.

Listing 3.8: Caching a Single Argument Flat Contract.

```
FlatContract<Integer> computeIntensiveContract = i -> {
           try {
               TimeUnit.SECONDS.sleep(5);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
           return true;
       };

SingleArgCachedContract<Integer> cachedContract = new SingleArgCachedContract<>(
    computeIntensiveContract);

long startTime = System.currentTimeMillis();
System.out.println("First Call");
cachedContract.validate(1);
System.out.println("Time Required :: " + (System.currentTimeMillis() - startTime)
    + " ms");
startTime = System.currentTimeMillis();
System.out.println("Second Call");
cachedContract.validate(1);
System.out.println("Time Required :: " + (System.currentTimeMillis() - startTime)
    + " ms");
```

Listing 3.9: Result of caching a contract with delay 5 seconds

```
First Call
Time Required :: 5016 ms
Second Call
Time Required :: 0 ms
```

The contract hierarchy in Rule4j empowers the programmer to build higher-order contracts by wrapping a flat contract inside another contract. Listing 3.10 demonstrates a how higher order contract by logically combining multiple flat contracts. Rule4j promotes functional programming and Listing 3.10 showcases how the contracts are combined in a declarative fashion.

Listing 3.10: Combining Flat Contracts

```
FlatContract<Person> flatContract1 = person -> true;

FlatContract<Person> flatContract2 = person -> false;

FlatContract<Person> flatContract3 = person -> true;

System.out.print(flatContract1.or(flatContract2).or(flatContract3).validate(new
    Person("R", "S"))); // true

System.out.print(flatContract1.and(flatContract2).and(flatContract3).validate(new
    Person("R", "S"))); // false
```

These are some examples of how complicated contracts can be build using Rule4j. The framework facilitates building new contracts using the Open-Closed principle [28] by using the existing contracts. This promotes the reusability and maintainability of business validations which can be organized in contracts. The framework promotes Java functional programming constructs so that the contracts can be built in a declarative fashion and are readable.

CHAPTER 4

Contract Processing in Rule4j

This chapter gives a brief overview of how the contract chains are intercepted and executed in parallel by Rule4j. The first section describes how contract chains are defined and attached to the method signature. The second section gives a brief overview how contract organization. The third section outlines the design and architecture of the core contract execution engine and describes the approach used for asynchronous execution of contracts.

4.1 Defining Contract Chains

A contract chain consists of a list of contracts that need to be validated one after the other sequentially. The order of execution of contracts is the same as the order in which the contracts are specified in the chain. When a contract fails in the chain, the chain execution is aborted and all the subsequent contracts are not executed. A contract chain is composed of different types of contracts which is illustrated in the Figure 4.

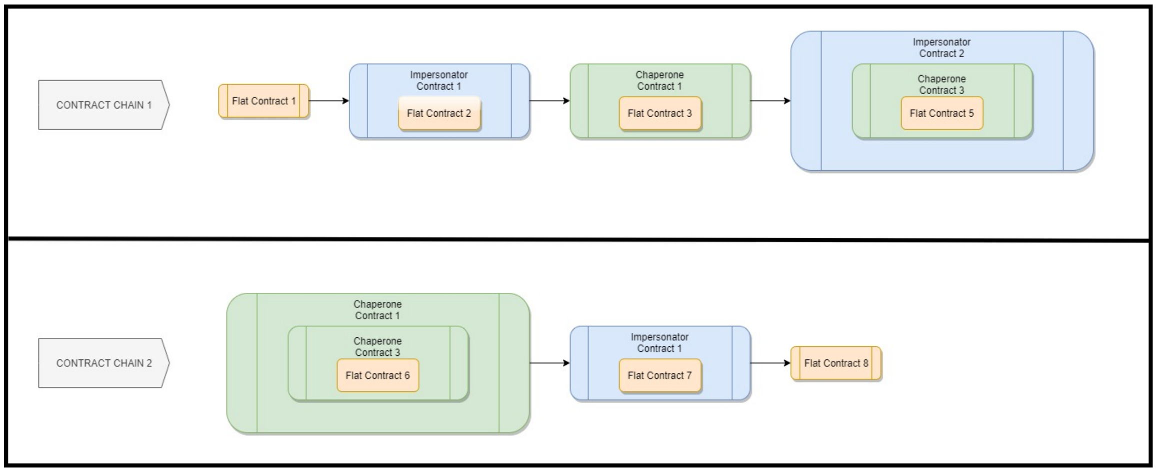


Figure 4: Contract Factory

The methods which are bound by contracts are annotated using the marker annotation `@UnderValidation`. This annotation tells the framework that the current method is bound by contract chains and is under validation and the execution is intercepted to process the contract chains. Table 1 lists the custom annotations created in Java to define the contract chains. Guice [29] method interceptors are injected to intercept the arguments and begin contract execution when the following annotations are used.

Annotation	Type	Description
`@Validate`	Precondition	This annotation specifies all the contract chains containing single argument contracts.
`@BiValidate`	Precondition	This annotation lists contract chains comprising double argument contracts.
`@PostValidate`	Postcondition	This annotation is used to specify single argument contract chains to validate function result.

Table 1: Annotations to declare contract chains

`@Validate, @BiValidate` and `@PostValidate` are single value annotations and take in a list of strings that specify the contract chains. Each chain is a string containing the chain name followed by contracts constituting the chain separated by `->`. Each contract is followed by the argument enclosed in parenthesis on which it operates. The name of the argument must match with the parameter name in the method signature. The method return value is denoted as `*`, and the postvalidation contracts should specify `*` as the argument. Listing 4.1 showcases the syntax to define contract chains.

Listing 4.1: Contract chain definition

```
Single Argument Contract chain
"chain-name = contract_1(arg) -> contract_2(arg) -> contract_3(arg) -> ... ->
    contract_n(arg)"
Double/Bi Argument Contract chain
"chain-name = contract_1(arg1, arg2) -> contract_2(arg1, arg2) -> contract_3(arg1,
     arg2) -> ... -> contract_n(arg1, arg2)"
```

Listing 4.2 demonstrates an example, how the contract chains can be defined using the annotations. The preconditions consist of 2 single argument contract chains and 1 bi argument contract chains which validate the input arguments. The post condition consist of 1 single argument contract chain that validates the result of the method `generate` There are four contract chains in the example, and the contracts are specified in the order which they will be executed.

Listing 4.2: An Example Demonstrating Contract Chains Definition

```
@UnderValidation
@Validate(value = {"single-arg-chain1 = personValidator1(person) ->
    personValidator2(person)",
       "single-arg-chain2 = portfolioValidator(portfolio) -> portfolioValidator2(
           portfolio)"})
@BiValidate(value = {"bi-arg-chain1 = biValidator1(person, portfolio) ->
    biValidator2(portfolio, person)"})
@PostValidate(value = {"post-arg-chain1 = resultValidator1(*)", "post-arg-chain2 =
     resultValidator2(*)"})
public int generate(Person person, Portfolio portfolio) {
    System.out.println("Portfolio Generated ");
    return 1;
}
```

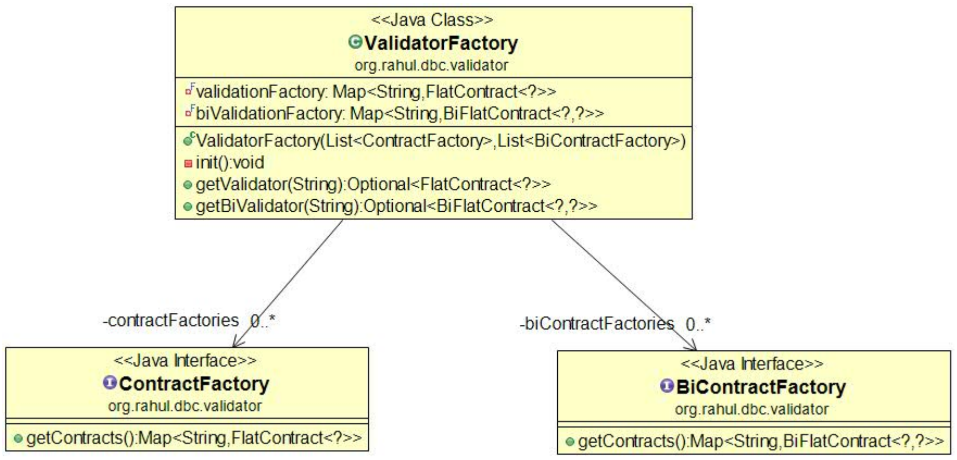
Table 2 lists all the contracts involved in the contract chain definition in the Listing 4.2.

Chain Name	Chain Type	Underlying Contracts
`single-arg-chain1`	Single Argument	personValidator1 personValidator2
`single-arg-chain2`	Single Argument	portfolioValidator portfolioValidator2
`bi-arg-chain1`	Bi Argument	biValidator1 biValidator2
`post-arg-chain1`	Postcondition	resultValidator1 resultValidator2

Table 2: Contract Chain Definition Details

4.2 Contract Definition

The previous chapter addressed how the different types of contracts are constructed. This section will showcase how these contracts can be injected into the framework and can be used to build chains.The contracts are organized in factory classes. The hierarchy of factory classes



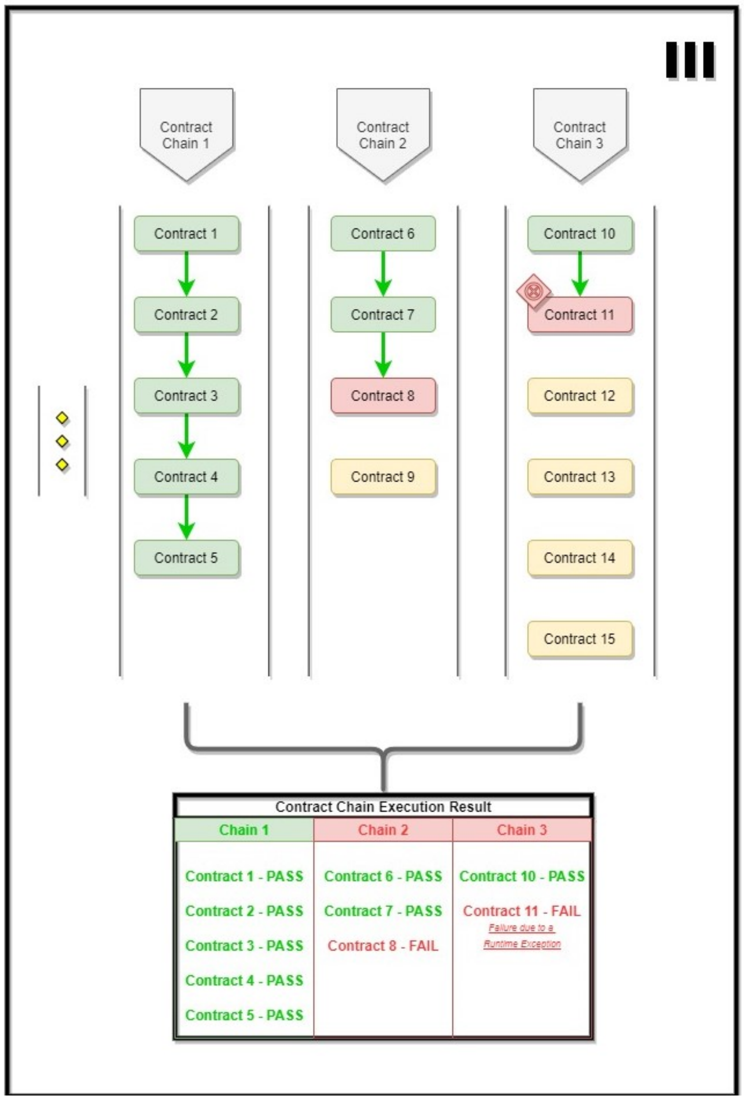
<<Java Class>>
ValidatorFactory
org.rahul.dbc.validator
validationFactory: Map<String,FlatContract<?>>
biValidationFactory: Map<String,BiFlatContract<?,?>>
ValidatorFactory(List<ContractFactory>,List<BiContractFactory>)
init():void
getValidator(String):Optional<FlatContract<?>>
getBiValidator(String):Optional<BiFlatContract<?,?>>
-contractFactories 0..*
-biContractFactories 0..*
<<Java Interface>>
ContractFactory
org.rahul.dbc.validator
getContracts():Map<String,FlatContract<?>>
<<Java Interface>>
BiContractFactory
org.rahul.dbc.validator
getContracts():Map<String,BiFlatContract<?,?>>

Figure 5: Contract Factory

The factory classes must implement the `ContractFactory` interface or `BiContractFactory` interface. They contain the mapping of contract names to the actual contract object instance. These factory classes are then injected into the `ValidatorFactory` class. When the method under validation is intercepted by the rule engine, will then query the `ValidatoryFactory` to get the contracts specified in the contract chains. The class diagram in Figure 5 depicts this relationship.

4.3 Design of Contract Execution Engine

The previous two sections explained how to define contracts and contract chains. This section will describe the architecture of the Rule4j contract execution engine. The core feature of Rule4j is that it executes the contract chains in parallel and all the contracts are executed asynchronously.



Contract Chain Execution Result

Chain 1	Chain 2	Chain 3
Contract 1 - PASS	Contract 6 - PASS	Contract 10 - PASS
Contract 2 - PASS	Contract 7 - PASS	Contract 11 - FAIL Failure due to a Runtime Exception
Contract 3 - PASS	Contract 8 - FAIL	
Contract 4 - PASS		
Contract 5 - PASS		

Figure 6: Contract Chain Execution

Each contract chain is validated independently, and their result is combined to create a final validation result. This has a performance advantage over the serial execution of contracts. Another benefit of this approach is that a contract failure will not result in the blocking of other independent contracts. Figure 6 gives an overview of contract chain execution in the Validation Rule Engine.

4.3.1 Method Interception

Guice [29] conforms to AOP Alliance [30] and provides support for Aspect-oriented programming [14] using method interceptors.Rule4j intercepts the contract chain definition provided in the annotations. After the interception, a list of contract wrappers is constructed depending on the type of contract chain. As the name suggests a contract wrapper is a wrapper around the actual contract instance pared with the argument values that it operates on. Figure 7 and Figure 8 illustrate the design of contract wrappers.

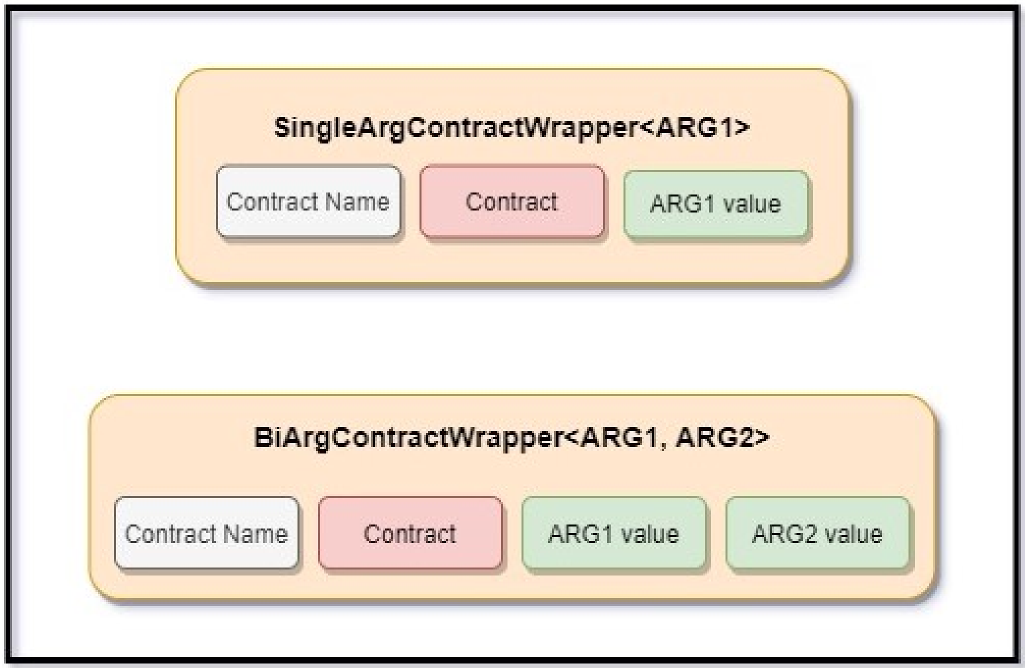
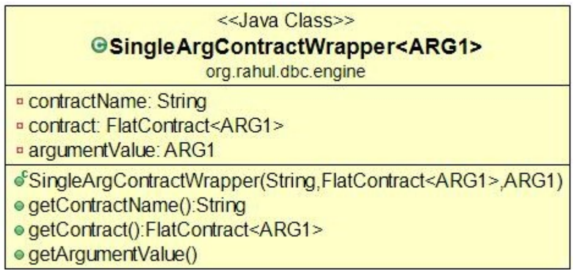
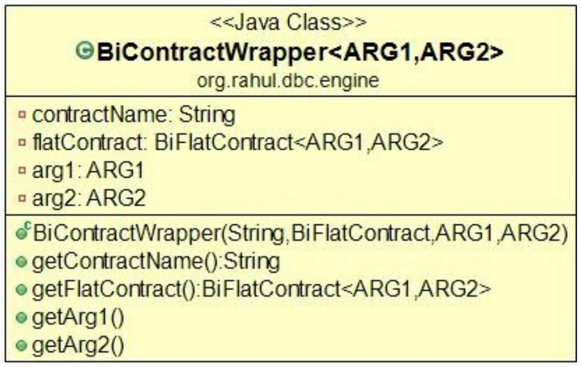


Figure 7: Contract Wrappers


<<Java Class>>
SingleArgContractWrapper<ARG1>
org.rahul.dbc.engine
contractName: String
contract: FlatContract<ARG1>
argumentValue: ARG1
SingleArgContractWrapper(String,FlatContract<ARG1>,ARG1)
getContractName():String
getContract():FlatContract<ARG1>
getArgumentValue()

(a) Single Argument Contract Wrapper


<<Java Class>>
BiContractWrapper<ARG1,ARG2>
org.rahul.dbc.engine
contractName: String
flatContract: BiFlatContract<ARG1,ARG2>
arg1: ARG1
arg2: ARG2
BiContractWrapper(String,BiFlatContract,ARG1,ARG2)
getContractName():String
getFlatContract():BiFlatContract<ARG1,ARG2>
getArg1()
getArg2()

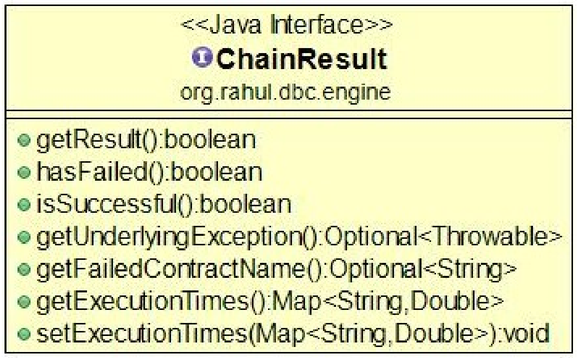
(b) Bi Argument Contract Wrapper

Figure 8: Contract Wrapper Class Diagrams

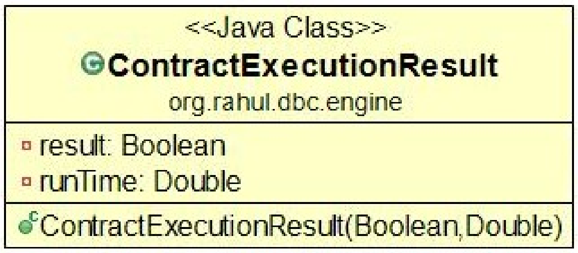
After the interception, a list of contract chain wrappers is constructed from a contract chain definition. These lists are then executed in parallel by the `ContractChainExecutor`.

4.3.2 Contract Execution

The wrapped contract chains are executed in parallel by forking the execution. The chains are submitted to the `ContractChainExecutor`, which returns a promise of `ChainResult` that contains the detailed execution result of the chain. The promises of `ChainResult` are joined together to compute the final execution result of the chains.


<<Java Interface>>
ChainResult
org.rahul.dbc.engine
getResult():boolean
hasFailed():boolean
isSuccessful():boolean
getUnderlyingException():Optional<Throwable>
getFailedContractName():Optional<String>
getExecutionTimes():Map<String,Double>
setExecutionTimes(Map<String,Double>):void

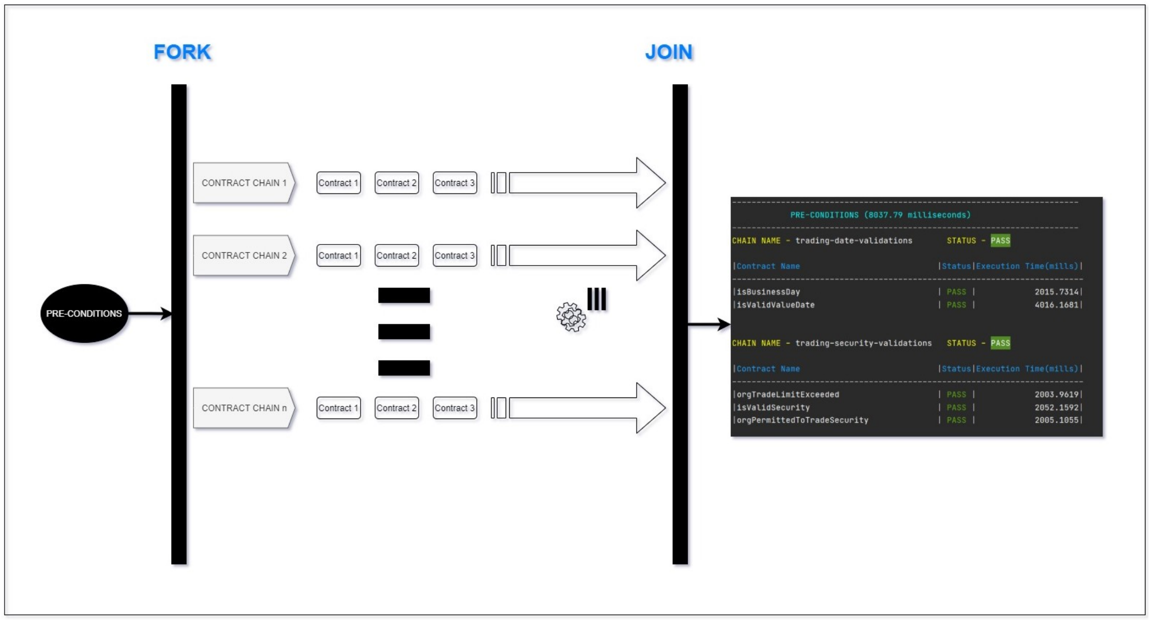
(a) Contract chain Result



(b) Contract Execution Result

Figure 9: Result Object Class Diagrams

Figure 10 illustrates the execution of precondition contract chains. The method under validation is only executed when the preconditions pass successfully, otherwise an `RuntimeException` is thrown with the details of the chain results.



```
------------------------------------------------------------------
          PRE-CONDITIONS (8037.79 milliseconds)
------------------------------------------------------------------
CHAIN NAME - trading-date-validations       STATUS - PASS

|Contract Name                               |Status|Execution Time(mills)|
---------------------------------------------------------------------
|isBusinessDay                               | PASS |            2015.7314|
|isValidValueDate                            | PASS |            4016.1681|


CHAIN NAME - trading-security-validations   STATUS - PASS

|Contract Name                               |Status|Execution Time(mills)|
---------------------------------------------------------------------
|orgTradeLimitExceeded                       | PASS |            2003.9619|
|isValidSecurity                             | PASS |            2052.1592|
|orgPermittedToTradeSecurity                 | PASS |            2005.1055|
```

Figure 10: Preconditions Contract Chain Execution

The `ContractChainExecutor` queues the contract wrappers in each chain in the execution order and submits the contract wrapper to `ContractExecutionEngine`, which returns a promise of `ContractExecutionResult`. The `ContractChainExecutor` will submit the subsequent task if the `ContractExecutionResult` is successful. The `ContractExecutionEngine` contains a `ThreadPool` that is configurable during startup. This non-blocking execution of contracts is achieved using asynchronous programming features like `FutureTask` ,`CompletableFuture` and `CompletionStage` in Java. The working of core engine and the relationship between `ContractChainExecutor` and `ContractExecutionEngine` is illustrated in the Figure 11.

The `ContractHierarchyInterceptor` intercepts the execution when it finds a method under validation. It then constructs a list of contract wrappers corresponding to the contract chain definitions. The execution is then forked to and the list of wrappers is passed to the `ContractChainExecutor` which returns a promise containing `ChainResult` for each chain. The `ContractChainExecutor` queues all the contracts in the chain submits them to the `ContractExecutionEngine`. The `ContractExecutionEngine` returns a promise containing `ContractExecutionResult` to the `ContractChainExecutor` and executes the contracts asynchronously by submitting them to a `ThreadPool`. Finally, the promises of `ChainResult` are joined to create the final result. Figure 12 illustrates the Class Diagram of the core engine.

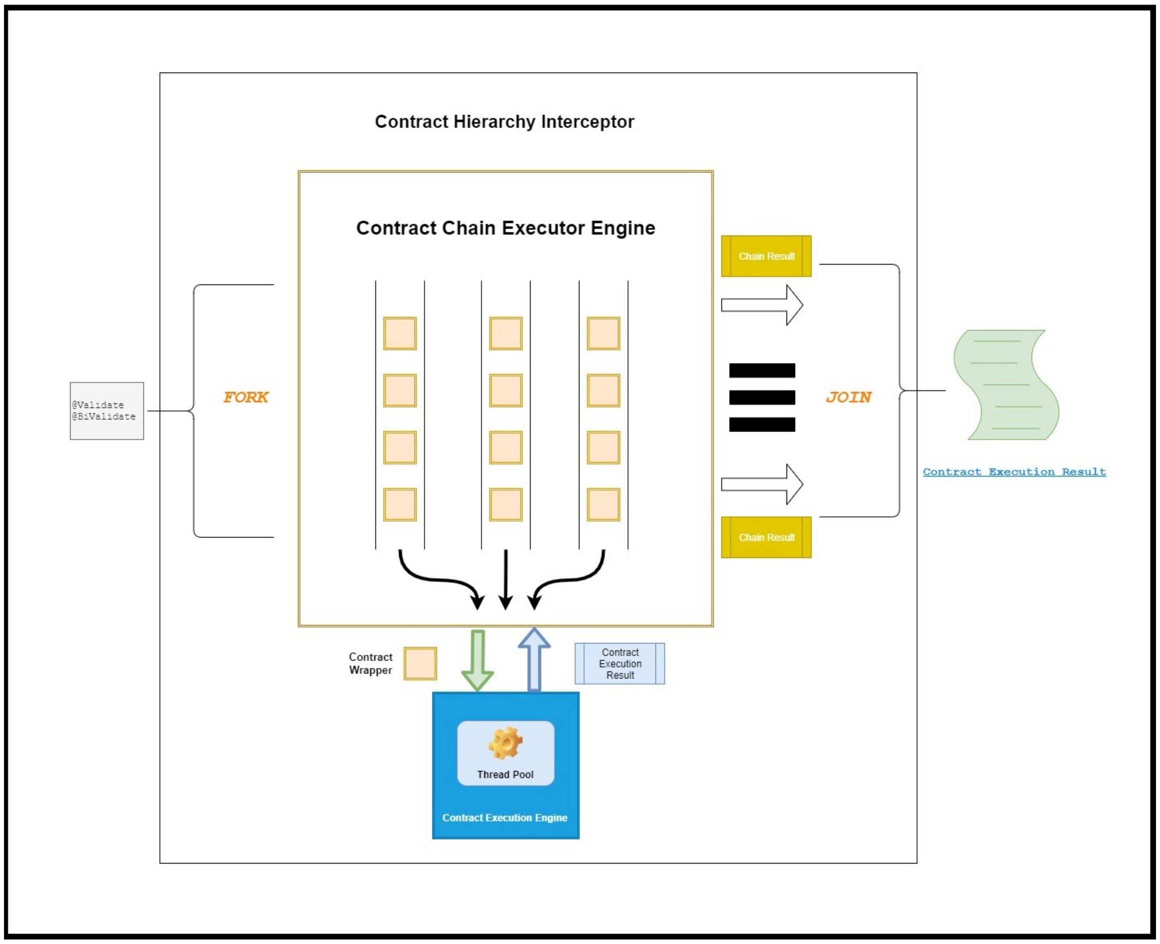
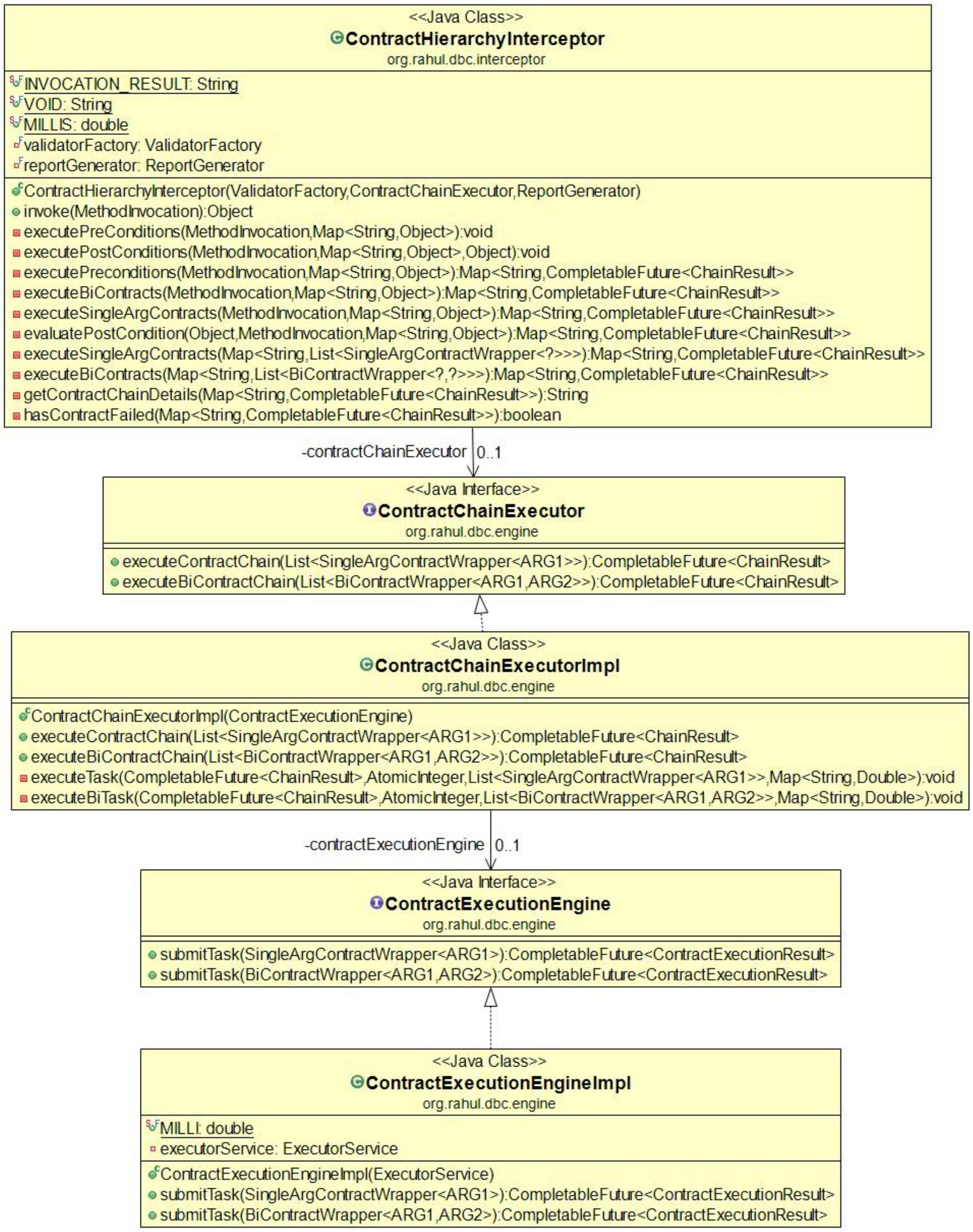

Contract Hierarchy Interceptor
Contract Chain Executor Engine
@Validate
@BiValidate
FORK
JOIN
Chain Result
Chain Result
Contract Execution Result
Contract Wrapper
Contract Execution Result
Thread Pool
Contract Execution Engine

Figure 11: Visualization of Contract Processing in Rule4j



<<Java Class>>
ContractHierarchyInterceptor
org.rahul.dbc.interceptor
INVOCATION_RESULT: String
VOID: String
MILLIS: double
validatorFactory: ValidatorFactory
reportGenerator: ReportGenerator
ContractHierarchyInterceptor(ValidatorFactory,ContractChainExecutor,ReportGenerator)
invoke(MethodInvocation):Object
executePreConditions(MethodInvocation,Map<String,Object>):void
executePostConditions(MethodInvocation,Map<String,Object>,Object):void
executePreconditions(MethodInvocation,Map<String,Object>):Map<String,CompletableFuture<ChainResult>>
executeBiContracts(MethodInvocation,Map<String,Object>):Map<String,CompletableFuture<ChainResult>>
executeSingleArgContracts(MethodInvocation,Map<String,Object>):Map<String,CompletableFuture<ChainResult>>
evaluatePostCondition(Object,MethodInvocation,Map<String,Object>):Map<String,CompletableFuture<ChainResult>>
executeSingleArgContracts(Map<String,List<SingleArgContractWrapper<?>>>):Map<String,CompletableFuture<ChainResult>>
executeBiContracts(Map<String,List<BiContractWrapper<?,?>>>):Map<String,CompletableFuture<ChainResult>>
getContractChainDetails(Map<String,CompletableFuture<ChainResult>>):String
hasContractFailed(Map<String,CompletableFuture<ChainResult>>):boolean
-contractChainExecutor
0..1
<<Java Interface>>
ContractChainExecutor
org.rahul.dbc.engine
executeContractChain(List<SingleArgContractWrapper<ARG1>>):CompletableFuture<ChainResult>
executeBiContractChain(List<BiContractWrapper<ARG1,ARG2>>):CompletableFuture<ChainResult>
<<Java Class>>
ContractChainExecutorImpl
org.rahul.dbc.engine
ContractChainExecutorImpl(ContractExecutionEngine)
executeContractChain(List<SingleArgContractWrapper<ARG1>>):CompletableFuture<ChainResult>
executeBiContractChain(List<BiContractWrapper<ARG1,ARG2>>):CompletableFuture<ChainResult>
executeTask(CompletableFuture<ChainResult>,AtomicInteger,List<SingleArgContractWrapper<ARG1>>,Map<String,Double>):void
executeBiTask(CompletableFuture<ChainResult>,AtomicInteger,List<BiContractWrapper<ARG1,ARG2>>,Map<String,Double>):void
-contractExecutionEngine
0..1
<<Java Interface>>
ContractExecutionEngine
org.rahul.dbc.engine
submitTask(SingleArgContractWrapper<ARG1>):CompletableFuture<ContractExecutionResult>
submitTask(BiContractWrapper<ARG1,ARG2>):CompletableFuture<ContractExecutionResult>
<<Java Class>>
ContractExecutionEngineImpl
org.rahul.dbc.engine
MILLI: double
executorService: ExecutorService
ContractExecutionEngineImpl(ExecutorService)
submitTask(SingleArgContractWrapper<ARG1>):CompletableFuture<ContractExecutionResult>
submitTask(BiContractWrapper<ARG1,ARG2>):CompletableFuture<ContractExecutionResult>

Figure 12: Class Diagram of Rule4j Engine

CHAPTER 5
Experiments and Performance Results

This chapter highlights the advantages of using Rule4j over the traditional approach to write business validations. Business validations have plagued large enterprise software, as they tend to scatter across the entire codebase over time. This has a detrimental impact on software maintainability and extensibility. Due to the proliferation of the validations throughout the code, even the business validations that are independent of each other tend to be executed sequentially. There is a significant scope of performance gain here, which is demonstrated by the application of Rule4j in this chapter.

A sample use case of equity trade execution is used to demonstrate the performance benefits. Experiments conducted in Section 5.3 show that a successful validation execution using Rule4j run four times faster than validations executed without using Rule4j. Section 5.2.2 demonstrates how separation of business validations and business logic is achieved using Rule4j in the trade execution scenario.

5.1 System Configuration

The experiments were conducted on a system with following configurations -

- **Operating System -** Windows 10 Pro 64-bit(10.0, Build 19042)
- **Processor -** Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz(8 CPUs), 2.0GHz
- **System Memory -** 16 GB
- **Java Version -** 12.0.2
- **Executor Service -** Fixed Thread Pool of size 3
- **Initial JVM Memory Allocation (Xms) -** 256 MB
- **Maximum JVM Memory Allocation (Xmx) -** 2 GB

5.2 Use Case - Equity Trade Execution

Consider a software module to execute an equity trade. Since day trading is a highly regulated, it requires various restrictions and defensive checks before placing an order. These restrictions are modified regularly depending on external factors like market volatility and investor behavior. There are additional validations that are in place to restrict the day trader's activity in the market. These restrictions are based on the seniority of the trader and the diversity of the trader's portfolio. These validations in the trade execution scenario mimic the validations that programmers encounter while programming various use-cases.

A sample Java module is built around this scenario, by implementing such checks and mocking the required APIs. A noticeable delay is introduced in the mock validations to simulate a real-time scenario. Table 3 lists down all the validations, grouping them by their dependencies. The validations listed in Table 3 are exposed as API by creating a mocked service. These services have been given a delay of 2 seconds using a thread sleep. The contracts make calls to these APIs to perform validations.

Group	Name	Details
PRECONDITIONS		
Trader Credentials	Trader should not be empty	Trader object in the Trade Object should not be null.
	Id should be valid	The Trader Id in the Trader Object should be valid identifier.
	Trader is authorized	The Trader should be authorized to make a trade.
Trader Margins	VaR exposure permissible	The Trader's Value-at-Risk exposure is permissible.
	VoE exposure permissible	The Trader Equity exposure is permissible.
	Daily limit not exceeded	The Trader's daily trading limit is not exceeded.
Trading Date	Should be a Business Day	The Trade date should be a valid business day.
	Should be a valid Value Date	The Trade can be valued on the date.
Trading Security	Trade security should be valid	The Trade security is publicly traded on the exchange.
	Permitted to trade security	The organization is permitted to trade the security.
	Security trading under limit	The organization has not exceeded the daily trading limit of the security.
Trader Consistency	Trader is owner of the Trade	The Trader executing the trade should own the trade.
	Trader authorized to trade on date.	The Trader is authorized to trade on the date.
	Trader margin is available for trade.	The Trader's margin balance is available for trade.
POSTCONDITIONS		
Trade Confirmation	Trade not overpriced	The trade price and commission should not exceed the daily limit.

Table 3: Business Validations for an Equity Trade Execution

5.2.1 Equity Trade Execution using Traditional Approach

Listing A.1 shows how these validations would be implemented traditionally without Rule4j. The validations are organized in a large sequence of `if-else` statement blocks. One can easily imagine a scenario where there is a complicated nesting of `if-else` statements. This will degrade the readability and maintainability of the code.

5.2.2 Equity Trade Execution using Rule4j

Listing 5.1 shows how the equity trade processor can be implemented with the help of the Rule4j. The validations listed in the Table 3. are organized in factory classes as explained in section 4.2. The contract chain definition lists all the validations that need to be taken care of as preconditions and postconditions.

Listing 5.1: Equity Trade Processing using Rule4j

```
public class NewEquityBuyOrderExecutor implements OrderExecutor {
    private TradeProcessor tradeProcessor;
    @Inject
    public NewEquityBuyOrderExecutor(final TradeProcessor tradeProcessor) {
        this.tradeProcessor = tradeProcessor;
    }
    @UnderValidation
    @Validate(value = {"trader-credentials = TraderShouldNotBeEmpty(trade) ->
        TraderIdShouldBeValid(trader) -> isAuthorizedTrader(trade)",
            "trader-margins = tradersVaRExposurePermissible(trader) ->
                tradersVoEExposurePermissible(trader) ->
                tradersDailyLimitNotExceeded(trade)",
            "trading-date-validations = isBusinessDay(trade) -> isValidValueDate(
                trade)",
            "trading-security-validations = isValidSecurity(trade) ->
                orgPermittedToTradeSecurity(trade) -> orgTradeLimitExceeded(trade)"
                })
    @BiValidate(value = {"trader-consistency = traderExecutingTradeShouldOwnTrade(
        trade, trader) -> traderAuthorizedToTradeOnDate(trade, trader) ->
        traderMarginAvailableForTrade(trade, trader)"})
    @PostValidate(value = {"trade-confirmation = tradeConfirmationNotOverpriced(*)"
        })
    public TradeConfirmation executeOrder(Trade trade, Trader trader) throws
        Exception {
        //only business logic
        return this.tradeProcessor.executeTrade(trade);
    }
}
```

Listing A.6 demonstrates an alternative approach approach by combining contracts in chain `trading-date-validations` and replacing them with a single higher-order contract. This approach can be used to shorten the contract chains by constructing higher-order contracts. The programmers can use this approach to make their contract chains more concise without degrading the performance.

5.3 Experiments

This section shows the output along with the total execution time of both approaches to execute the trading validations. This is done by writing unit test cases that simulate the various use-cases. The unit tests are written using JUnit4 [31].

5.3.1 Successful Equity Trade Execution

In this scenario all the preconditions and postconditions are satisfied and this results in successful execution of the trade. Listing 5.2 shows the output and execution time of successful equity trade execution without using Rule4j. Listing 5.3 shows the output when Rule4j is used to execute the trading validations. The output shows in detail the name of the Chain and its status. The output also presents the time taken to execute each contract. Table 4 lists the execution time for both the approaches.

Listing 5.2: Output after executing the trade using traditional approach

```
Trader is not empty.
Trader Id Valid.
Trader is authorized to Trade.
The Trader owns the trade
Trade is authorized to trade today.
Trader has minimum margin balance to trade the security.
Trader's VaR exposure is permissible.
Trader's VoE permissible.
Trader's daily limit not exceeded.
Trading date is a valid business day.
Trade has valid Value date.
The Trade security is valid and publicly traded.
The Trade security authorized by the firm.
Trade limit not exceeded.
Trade commission not over priced.
Trade Executed Successfully
Time required for execution - 24136.7762 ms
```

Listing 5.3: Output after executing the trade using Rule4j

```


-----------------------------------------------------------------------
          PRE-CONDITIONS (6079.87 milliseconds)
-----------------------------------------------------------------------
CHAIN NAME - trading-date-validations   STATUS - PASS

|Contract Name                      |Status|Execution Time(mills)|
------------------------------------------------------------------------
|isBusinessDay                      | PASS |          2002.7476|
|isValidValueDate                   | PASS |          4007.6596|

CHAIN NAME - trading-security-validations STATUS - PASS

|Contract Name                      |Status|Execution Time(mills)|
------------------------------------------------------------------------
|orgTradeLimitExceeded              | PASS |          2020.0318|
|isValidSecurity                    | PASS |          2039.5496|
|orgPermittedToTradeSecurity        | PASS |          2005.6059|

CHAIN NAME - trader-consistency         STATUS - PASS

|Contract Name                      |Status|Execution Time(mills)|
------------------------------------------------------------------------
|traderAuthorizedToTradeOnDate        | PASS |          2008.1244|
|traderMarginAvailableForTrade        | PASS |          2012.0686|
|traderExecutingTradeShouldOwnTrade   | PASS |              0.032|

CHAIN NAME - trader-credentials         STATUS - PASS

|Contract Name                        |Status|Execution Time(mills)|
------------------------------------------------------------------------
|TraderIdShouldBeValid                | PASS |             0.0173|
|TraderShouldNotBeEmpty               | PASS |             0.8664|
|isAuthorizedTrader                   | PASS |          2011.3714|

CHAIN NAME - trader-margins             STATUS - PASS

|Contract Name                        |Status|Execution Time(mills)|
------------------------------------------------------------------------
|tradersDailyLimitNotExceeded         | PASS |          2001.6636|
|tradersVaRExposurePermissible        | PASS |          2013.1303|
|tradersVoEExposurePermissible        | PASS |          2012.5087|
-----------------------------------------------------------------------
            POST-CONDITIONS (2.14 milliseconds)
-----------------------------------------------------------------------
CHAIN NAME - trade-confirmation         STATUS - PASS

|Contract Name                        |Status|Execution Time(mills)|
------------------------------------------------------------------------
|tradeConfirmationNotOverpriced       | PASS |             0.0509|
```

Traditional Approach	Rule Engine Approach
24.136	6.079

Table 4: Execution Time for a Successful Validation (in seconds)

5.3.2 Failure in Equity Trade Execution

In this scenario, there is a failure in one of the validations, which results in a failure to execute the trade. Listing 5.4 shows the output and execution time without the use of Rule4j. Listing 5.5 shows the final output when Rule4j is used to manage the trading validations.

Listing 5.4: Failure Details when executing the trade using traditional approach

```
Trader is not empty.
Trader Id Valid.
Trader is authorized to Trade.
The Trader owns the trade
Trade is authorized to trade today.
Trader has minimum margin balance to trade the security.
Trader's VaR exposure is permissible.
Trader's VoE permissible.
Trader's daily limit not exceeded.
java.lang.Exception:  Trade Date is not a valid date.
  at org.rahul.dbc.use_case.trade_processing.TraditionalEquityBuyOrderExecutor.
      executeOrder(TraditionalEquityBuyOrderExecutor.java:76)
  at org.rahul.dbc.UseCaseDemoTest.executeTradeWithError_TraditionalMethod(
      UseCaseDemoTest.java:55)}
Time required for execution - 14044.8799 ms
```

Listing 5.5: Output after executing the trade using Rule4j

```
java.lang.RuntimeException:
----------------------------------------------------------------------
           PRE-CONDITIONS (6080.01 milliseconds)
----------------------------------------------------------------------
CHAIN NAME - trading-date-validations   STATUS - FAIL
|Contract Name                       |Status|Execution Time(mills)|
-----------------------------------------------------------------------
|isBusinessDay                       | FAIL |                0.0|
Failed Due to Underlying Exception -
java.lang.RuntimeException:  Trade Date is not a valid date
  at org.rahul.dbc.use_case.trading_validations.TradeContracts.lambda$init$0(
      TradeContracts.java:36)
  at org.rahul.dbc.engine.ContractExecutionEngineImpl.lambda$submitTask$0(
      ContractExecutionEngineImpl.java:22)
  at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java
      :515)
  at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
  at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
      ThreadPoolExecutor.java:1128)
  at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(
      ThreadPoolExecutor.java:628)
  at java.base/java.lang.Thread.run(Thread.java:835)

CHAIN NAME - trading-security-validations STATUS - PASS
```

```
|Contract Name                      |Status|Execution Time(mills)|
-------------------------------------------------------------------
|orgTradeLimitExceeded              | PASS |           2009.659|
|isValidSecurity                    | PASS |          2047.4333|
|orgPermittedToTradeSecurity        | PASS |          2009.3355|

CHAIN NAME - trader-consistency         STATUS - PASS
|Contract Name                      |Status|Execution Time(mills)|
-------------------------------------------------------------------
|traderAuthorizedToTradeOnDate      | PASS |          2000.4417|
|traderMarginAvailableForTrade      | PASS |          2010.2296|
|traderExecutingTradeShouldOwnTrade | PASS |             0.0333|

CHAIN NAME - trader-credentials         STATUS - PASS
|Contract Name                      |Status|Execution Time(mills)|
-------------------------------------------------------------------
|TraderIdShouldBeValid              | PASS |             0.0194|
|TraderShouldNotBeEmpty             | PASS |             1.0732|
|isAuthorizedTrader                 | PASS |          2003.2864|

CHAIN NAME - trader-margins             STATUS - PASS
|Contract Name                      |Status|Execution Time(mills)|
-------------------------------------------------------------------
|tradersDailyLimitNotExceeded       | PASS |           2009.475|
|tradersVaRExposurePermissible      | PASS |          2006.0442|
|tradersVoEExposurePermissible      | PASS |          2010.2055|
  at org.rahul.dbc.interceptor.ContractHierarchyInterceptor.executePreConditions(
      ContractHierarchyInterceptor.java:63)
  at org.rahul.dbc.interceptor.ContractHierarchyInterceptor.invoke(
      ContractHierarchyInterceptor.java:46)
  at org.rahul.dbc.UseCaseDemoTest.executeTradeWithError_WithContracts(
      UseCaseDemoTest.java:69)
```

Traditional Approach	Rule Engine Approach
14.04	6.08

Table 5: Execution Time to Report a Failed Validation (in seconds)

In the traditional approach, a short circuit evaluation is carried out and an Exception is thrown as soon as there is a validation failure. All the other validations which are independent of the failed validation do not execute. This is not the case when Rule4j is used to execute the validations, only the contract chain which has the failing validation is stopped from executing, while all the other chains continue to execute normally. Due to a failure in one of the contract chain, a `RuntimeException` is thrown with the combined result of all the contract chains and their details. Table 5 lists the time taken for both the approaches.

One of the major advantages of using Rule4j is that it can report multiple validation failures that take place in different chains. This allows extensive feedback in the first attempt of execution of the method under validation. Listing A.5 demonstrates this for the equity trade execution scenario.

5.3.3 Result Analysis

The experiments performed in Section 5.3.1 and Section 5.3.2, conclude the superiority of Rule4j over the traditional approach to perform business validations. Experiments conducted in Section 5.3 show a performance improvement by a factor of 4, for successful validations as illustrated in Table 4. Rule4j reports a failed validation much faster as illustrated in Table 5. Rule4j reports multiple failures without increasing the processing time of execution, which is demonstrated in Section 5.3.2. The execution time of validations performed using Rule4j is similar in the case of success and failure scenarios. This is attributed to the parallel processing of contract chains in Rule4j. The total execution time is approximately equal to the execution time of the longest successful contract chain.

Table 6 lists the execution time of all the validations that are performed before executing an equity trade. The execution time of the traditional method in Table 4 is approximately equal to the sum of all the validation execution times in Table A since the validations are performed sequentially. The execution time of validations using Rule4j in Table 4 and Table 5 is approximately equal to the execution time of contract chain Trader Margins and Trading Date.

Chain Name	**Validation Name**	**Execution Time in Milliseconds**	
		Traditional	**Rule4j**
Trader Credentials	Trader not be empty	0.315	0.013
	Trader Id valid	0.038	0.592
	Authorized trader	2005.789	2015.042
Trader Margins	VaR exposure permissible	2013.012	2010.823
	VoE exposure permissible	2005.715	2000.777
	Trader's daily limit	2000.609	2012.534
Trading Date	Business day valid	2014.259	2005.791
	Value date valid	4021.293	4022.530
Trading Security	Valid Security	2022.772	2012.529
	Security permission	2010.349	2011.026
	Security limit	2013.396	2001.299
Trader Consistency	Trade ownership	0.047	0.0264
	Trader authorized on date	2000.606	2012.5682
	Trader Margin Balance	2002.1092	2011.0552
Trade Confirmation	Confirmation overpriced	0.059	0.018

Table 6: Execution Time of Validations

Since all the chains are validated in parallel, the run time of validations would be approximately equal to the execution time of the longest chain. The Trader Margin contract has 3 validations, each taking 2 seconds to complete the validation. Therefore the total execution time of the chain is approximately equal to 6 seconds. The average performance overhead due to Rule4j was found to be 38 milliseconds. This additional time can be attributed to the forking and joining of contract chain promises, creation of contract wrappers, instantiating contract chains, and the performance overhead of the underlying thread pool of execution.

Listing 5.2.2 describes how the trading validations are defined using contract chains in Rule4j. Rule4j enforces a clear separation of business validations from the business rules. Section A.1 demonstrates the traditional approach to write business validations. Section A.2 describes how these trading validations are organized using Rule4j, promoting reusability and maintainability.

CHAPTER 6
Conclusion

The current implementations of Design by Contract libraries in Java focus on writing short-hand contracts. The invariant specifications only perform rudimentary checks and are not capable of handling most of the business validation scenarios that large enterprise software encounter. The libraries run the contracts serially, as they intended to perform simple checks on the method arguments and results. The Rule4j library presented in this paper enables the programmer to write complicated and nested contracts using a contract hierarchy similar to the Racket programming language. The programmer can use the library to build validation chains that can be easily maintained and extended as the business requirements grow. Rule4j aims to achieve high cohesion and low coupling while organizing business validation for a system.

Rule4j executes the contract chains in parallel in a Fork-Join manner to achieve maximum performance. This offers greater flexibility to the programmer to organize validations in chains to achieve optimal performance. The contract hierarchy in Rule4j enables the programmer to write higher-order contracts and promote the reusability of existing contracts. The results of using Rule4j to write validations for equity trade execution scenarios demonstrate significant performance gain over the traditional approach. The experiments conducted show that validations performed using Rule4j run 4 times faster than the traditional approach.

The scope of future work to enhance Rule4j is to include marshaling and serialization of contracts, enabling sharing of existing contracts over the network. The current implementation of Rule4j supports contracts with a maximum arity of 2. The Rule4j framework can further be enhanced to support contracts with arity greater than 2.

LIST OF REFERENCES

[1] M. W. Godfrey and D. M. German, ‘‘On the evolution of lehman’s laws,’’ *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 613--619, 2014.

[2] C. Jones, ‘‘Social and technical reasons for software project failures,’’ *CrossTalk*, vol. 19, no. 6, pp. 4--9, 2006.

[3] ‘‘Nhs connecting for health,’’ Jan 2021. [Online]. Available: https://en.wikipedia.org/wiki/NHS_Connecting_for_Health

[4] ‘‘Abandoned nhs it system has cost £10bn so far,’’ Sep 2013. [Online]. Available: https://www.theguardian.com/society/2013/sep/18/nhs-records-system-10bn

[5] E. Razina and D. S. Janzen, ‘‘Effects of dependency injection on maintainability,’’ in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 2007, p. 7.

[6] B. Meyer, ‘‘Applying ’design by contract’,’’ *Computer*, vol. 25, no. 10, pp. 40--51, 1992.

[7] B. Porter, J. v. Zyl, and O. Lamy, ‘‘Welcome to apache maven.’’ [Online]. Available: https://maven.apache.org/

[8] Google, ‘‘google/guice.’’ [Online]. Available: https://github.com/google/guice

[9] ‘‘iso/iec 25010:2011 - systems and software.’’ [Online]. Available: https://www.iso.org/standard/35733.html

[10] M. Mari and E. Niemelä, ‘‘The impact of maintainability on component-based software systems,’’ in *Euromicro Conference.* IEEE Computer Society, 2003, pp. 25--25.

[11] D. Astels, *Test driven development: A practical guide.* Prentice Hall Professional Technical Reference, 2003.

[12] S. R. Chidamber and C. F. Kemerer, ‘‘A metrics suite for object oriented design,’’ *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476--493, 1994.

[13] T. Cerny and M. J. Donahoo, ‘‘How to reduce costs of business logic maintenance,’’ in *2011 IEEE International Conference on Computer Science and Automation Engineering*, vol. 1. IEEE, 2011, pp. 77--82.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, ‘‘Aspect-oriented programming,’’ in *European conference on object-oriented programming.* Springer, 1997, pp. 220--242.

[15] ‘‘Drools - business rules management system (java™, open source).’’ [Online]. Available: https://www.drools.org/

[16] D. Goswami, ‘‘Rule engine for validating complex business objects,’’ in *Proceedings of the 20th Conference on Pattern Languages of Programs*, 2013, pp. 1--18.

[17] M. Karaorman, U. Hölzle, and J. Bruno, ‘‘jcontractor: A reflective java library to support design by contract,’’ in *International Conference on Metalevel Architectures and Reflection.* Springer, 1999, pp. 175--196.

[18] G. T. Leavens and Y. Cheon, ‘‘Design by contract with jml,’’ 2006.

[19] D. Wampler, ‘‘Contract4j for design by contract in java: Design pattern-like protocols and aspect interfaces,’’ in *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2006, pp. 27--30.

[20] T. y. n. here, ‘‘The aspectj project: The eclipse foundation.’’ [Online]. Available: https://www.eclipse.org/aspectj/

[21] ‘‘apache commons jexl overview.’’ [Online]. Available: http://commons.apache.org/jexl/

[22] N. Rajkumar, ‘‘Designing a programming contract library for java,’’ 2015.

[23] Greenjoe, ‘‘greenjoe/lambdafromstring.’’ [Online]. Available: https://github.com/greenjoe/lambdaFromString

[24] Y. Dixit, ‘‘Library for writing contracts for java programs using prolog,’’ 2017.

[25] [Online]. Available: https://docs.racket-lang.org/guide/contract-boundaries.html

[26] R. B. Findler and M. Felleisen, ‘‘Contracts for higher-order functions,’’ in *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, 2002, pp. 48--59.

[27] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt, ‘‘Chaperones and impersonators: run-time support for reasonable interposition,’’ *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 943--962, 2012.

[28] R. C. Martin, ‘‘Getting a solid start,’’ *Robert C. Martin-objectmentor. com*, 2013.

[29] R. Vanbrabant, *Google Guice: agile lightweight dependency injection framework.* APress, 2008.

[30] R. Pawlak, “The aop alliance: Why did we get in,” *White Paper draft*, 2003.

[31] “About.” [Online]. Available: https://junit.org/junit4/index.html

APPENDIX

Code Snippets

A.1 Traditional Approach to Equity processing

Listing A.1: Equity Trade Processing without Rule4j

```java
package org.rahul.dbc.use_case.trade_processing;

import org.rahul.dbc.use_case.services.ValidationServices;
import org.rahul.dbc.use_case.trade.Trade;
import org.rahul.dbc.use_case.trade.TradeConfirmation;
import org.rahul.dbc.use_case.trader.Trader;

import javax.inject.Inject;
import java.util.Optional;
import java.util.function.BiConsumer;

public class TraditionalEquityBuyOrderExecutor {

   private ValidationServices validationService;
   private TradeProcessor tradeProcessor;

   @Inject
   TraditionalEquityBuyOrderExecutor(ValidationServices validationService,
       TradeProcessor tradeProcessor) {
       this.validationService = validationService;
       this.tradeProcessor = tradeProcessor;
   }

   public TradeConfirmation executeOrder(Trade trade, Trader trader) throws
       Exception {


       // Trader credentials

       this.logTime((t, tr) -> {
          if (Optional.ofNullable(trade.getTrader()).map(Trader::getTraderId).
              isEmpty()) {
              throw new RuntimeException("Trader cannot be empty in a Trade.");
          }
          System.out.println("Trader is not empty.");
       }, trade, trader);
```

```
        this.logTime((t, tr) -> {
            if (trader.getTraderId() == null) {
                throw new RuntimeException("Invalid trader executing the trade.");
            }
            System.out.println("Trader Id Valid.");
        }, trade, trader);

        this.logTime((t, tr) -> {
            if (!this.validationService.isAuthorizedTrader(trade.getTrader())) {
                throw new RuntimeException("The trader is not authorized to trade");
            }
            System.out.println("Trader is authorized to Trade.");
        }, trade, trader);


        this.logTime((t, tr) -> {
            if (!trader.getTraderId().equals(trade.getTrader().getTraderId())) {
                throw new RuntimeException("Invalid trader executing the trade.");
            }
            System.out.println("The Trader owns the trade");
        }, trade, trader);


        this.logTime((t, tr) -> {

            if (!this.validationService.isAuthorizedToTradeOnDate(trade.getTrader(),
                 trade.getTradeDate())) {
                throw new RuntimeException("The trader id not authorized to trade on
                    date " + trade.getTradeDate());
            }
            System.out.println("Trade is authorized to trade today.");

        }, trade, trader);

        // Trader exposure


        this.logTime((t, tr) -> {

            if (!this.validationService.traderMarginBalanceAvailableForTrade(trader,
                 trade.getQuantity(), trade.getSecurity())) {
                throw new RuntimeException(("The trader's margin is not sufficient
                   to make the trade"));
            }
        System.out.println("Trader has minimum margin balance to trade the
            security.");

    }, trade, trader);

    this.logTime((t, tr) -> {

        if (!this.validationService.tradersVaRExposurePermissible(trader)) {
            throw new RuntimeException("The trader's VAR exposure is above the
                permissible limits to execute trade");
        }
        System.out.println("Trader's VaR exposure is permissible.");

    }, trade, trader);


    this.logTime((t, tr) -> {

        if (!this.validationService.tradersVoEExposurePermissible(trader)) {
            throw new RuntimeException("The trader's Value of Equity is above
                the permissible limits");
        }
        System.out.println("Trader's VoE permissible.");

    }, trade, trader);


    this.logTime((t, tr) -> {

        if (!this.validationService.tradersDailyLimitNotExceeded(trader, trade.
            getQuantity(), trade.getSecurity())) {
            throw new RuntimeException(("The traders daily limit exceeded"));
        }
        System.out.println("Trader's daily limit not exceeded.");

    }, trade, trader);


    // Date


    this.logTime((t, tr) -> {
        if (!this.validationService.isBusinessDay(trade.getTradeDate())) {
            throw new RuntimeException("Trade Date is not a valid date.");
        }
        System.out.println("Trading date is a valid business day.");
    }, trade, trader);

    this.logTime((t, tr) -> {

        if (!this.validationService.isValidValueDate(trade.getValueDate(),
            trade.getTradeDate())) {
            throw new RuntimeException("Trade Value date is not a valid date");
        }

        System.out.println("Trade has valid Value date.");
    }, trade, trader);


    // Organization Permission


    this.logTime((t, tr) -> {

        if (!this.validationService.isValidSecurity(trade.getSecurity())) {
            throw new RuntimeException(("Security Invalid and not Traded"));
        }
        System.out.println("The Trade security is valid and publicly traded.");

    }, trade, trader);


    this.logTime((t, tr) -> {

        if (!this.validationService.orgPermittedToTradeSecurity(trade.
            getSecurity())) {
            throw new RuntimeException("Security cannot be traded by the firm");
        }
        System.out.println("The Trade security authorized by the firm.");

    }, trade, trader);


    this.logTime((t, tr) -> {
        if (!this.validationService.orgTradeLimitExceeded(trade.getSecurity()))
             {
            throw new RuntimeException("Trade Limit exceeded");
        }
        System.out.println("Trade limit not exceeded.");

    }, trade, trader);




    /*
     * ==> Trade Execution Logic goes in here
     * ==> Serial execution of validations/contracts
     * ==> Clean code due to the extracting the validation to a different
         services
     * ==> if statements
     * ==> will get complicated pretty soon as the business logic expands
     * ==> The business rules are not organized and hence the product owner
     * and developer will waste a lot of time going over the logic
     * ==> No asynchronous execution, special efforts required to parallelize
         the validations
     * ==> SHORT CIRCUIT VALIDATION - lot of changes in the trade.
     *
     */

    TradeConfirmation tradeConfirmation = this.tradeProcessor.executeTrade(
        trade);


    this.logTime((t, tr) -> {

        if (tradeConfirmation.getCommissionPaid() > 1000d) {
            throw new RuntimeException("Trade Overpriced");
        }

        System.out.println("Trade commission not over priced.");

    }, trade, trader);


    return tradeConfirmation;
}
```

```
    public void logTime(BiConsumer<Trade, Trader> validation, Trade trade, Trader
        trader) {

        try {
            long startTime = System.nanoTime();

            validation.accept(trade, trader);

            long endTime = System.nanoTime();

            System.out.println("Total time for validation - " + (endTime -
                startTime) / 1000000);
        } catch (RuntimeException e) {
            e.printStackTrace(System.out);
        }
    }
}
```

A.2 Organizing equity trade execution validations in Rule4j

This section showcases how the validations are organized using Factory Classes in Rule4j. The equity trade validations are distributed amongst Factory Classes, i.e, `TradeContracts`, `TraderContracts` and `TradeExecutionContracts`. This enables the reuse of existing validations in different contract definitions.

Listing A.2: Trade Contracts for Equity Trade Execution

```java
public class TradeContracts implements ContractFactory {

    public static final String TRADE_DATE_IS_VALID_BUSINESS_DAY = "isBusinessDay";
    public static final String VALUE_DATE_IS_VALID = "isValidValueDate";
    public static final String VALID_SECURITY = "isValidSecurity";
    public static final String SECURITY_PERMITTED = "orgPermittedToTradeSecurity";
    public static final String SECURITY_UNDER_TRADING_LIMIT = "
        orgTradeLimitExceeded";

    public static final String TRADE_CONFIRMATION_NOT_OVERPRICED = "
        tradeConfirmationNotOverpriced";

    private ValidationServices validationService;

    private Map<String, FlatContract<?>> contracts;

    public TradeContracts(final ValidationServices validationService) {
        this.validationService = validationService;
        this.init();
    }

    private void init() {

        this.contracts = new HashMap<>();
        this.contracts.put(TRADE_DATE_IS_VALID_BUSINESS_DAY, (Trade trade) -> {
            if (!this.validationService.isBusinessDay(trade.getTradeDate())) {
                throw new RuntimeException("Trade Date is not a valid date");
            }

            return true;
        });

        this.contracts.put(VALUE_DATE_IS_VALID, (Trade trade) -> {
```

```java
        if (!this.validationService.isValidValueDate(trade.getValueDate(),
            trade.getTradeDate())) {
            throw new RuntimeException("Trade Value date is not a valid date");
        }

        return true;
    });

    this.contracts.put(VALID_SECURITY, (Trade trade) -> {
        if (!this.validationService.isValidSecurity(trade.getSecurity())) {
            throw new RuntimeException(("Security Invalid and not Traded"));
        }

        return true;
    });

    this.contracts.put(SECURITY_PERMITTED, (Trade trade) -> {
        if (!this.validationService.orgPermittedToTradeSecurity(trade.
            getSecurity())) {
            throw new RuntimeException("Security cannot be traded by the firm");
        }

        return true;
    });

    this.contracts.put(SECURITY_UNDER_TRADING_LIMIT, (Trade trade) -> {
        if (!this.validationService.orgTradeLimitExceeded(trade.getSecurity()))
             {
            throw new RuntimeException("Trade Limit exceeded");
        }

        return true;
    });

    this.contracts.put(TRADE_CONFIRMATION_NOT_OVERPRICED, (TradeConfirmation
        tradeConfirmation) -> {
        if (tradeConfirmation.getCommissionPaid() > 1001d) {
            throw new RuntimeException("Trade Overpriced");
        }

        return true;
    });
}
```

```
    @Override
    public Map<String, FlatContract<?>> getContracts() {
        return this.contracts;
    }
}
```

Listing A.3: Trader Contracts for Equity Trade Execution

```java
public class TraderContracts implements ContractFactory {

    public static final String TRADER_SHOULD_NOT_BE_EMPTY = "TraderShouldNotBeEmpty
        ";
    public static final String TRADER_ID_SHOULD_BE_VALID = "TraderIdShouldBeValid";
    public static final String TRADER_SHOULD_BE_AUTHORIZED = "isAuthorizedTrader";
    public static final String TRADER_HAS_ENOUGH_MARGIN_BALANCE = "
        traderMarginBalanceAvailableForTrade";
    public static final String TRADER_VAR_EXPOSURE_WITHIN_LIMIT = "
        tradersVaRExposurePermissible";
    public static final String TRADER_VOE_EXPOSURE_WITHIN_LIMIT = "
        tradersVoEExposurePermissible";
    public static final String TRADER_DAILY_LIMIT_NOT_EXCEEDED = "
        tradersDailyLimitNotExceeded";
    private ValidationServices tradeValidationServices;
    private Map<String, FlatContract<?>> contracts;


    public TraderContracts(ValidationServices tradeValidationServices) {
        this.tradeValidationServices = tradeValidationServices;
        this.initContracts();
    }

    private void initContracts() {

        this.contracts = new HashMap<>();

        this.contracts.put(TRADER_SHOULD_NOT_BE_EMPTY, (Trade trade) -> {
            if (Optional.ofNullable(trade.getTrader()).map(Trader::getTraderId).
                isEmpty()) {
                throw new RuntimeException("Trader cannot be empty in a Trade.");
            }
            return true;
        });

        this.contracts.put(TRADER_ID_SHOULD_BE_VALID, (Trader trader) -> {
            return trader.getTraderId() != null;
        });

        this.contracts.put(TRADER_SHOULD_BE_AUTHORIZED, (Trader trader) -> {
            if (!this.tradeValidationServices.isAuthorizedTrader(trader)) {
                throw new RuntimeException("The trader is not authorized to trade");
            }
            return true;
        });


        this.contracts.put(TRADER_HAS_ENOUGH_MARGIN_BALANCE, (Trade trade) -> {
            if (!this.tradeValidationServices.traderMarginBalanceAvailableForTrade(
                trade.getTrader(), trade.getQuantity(), trade.getSecurity())) {
                throw new RuntimeException(("The trader's margin is not sufficient
                    to make the trade"));
            }
            return true;
        });

        this.contracts.put(TRADER_VAR_EXPOSURE_WITHIN_LIMIT, (Trader trader) -> {
            if (!this.tradeValidationServices.tradersVaRExposurePermissible(trader))
                 {
                throw new RuntimeException("The trader's VAR exposure is above the
                    permissible limits to execute trade");
            }

            return true;
        });

        this.contracts.put(TRADER_VOE_EXPOSURE_WITHIN_LIMIT, (Trader trader) -> {
            if (!this.tradeValidationServices.tradersVoEExposurePermissible(trader))
                 {
                throw new RuntimeException("The trader's Value of Equity is above
                    the permissible limits");
            }

            return true;
        });

        this.contracts.put(TRADER_DAILY_LIMIT_NOT_EXCEEDED, (Trade trade) -> {
            if (!this.tradeValidationServices.tradersDailyLimitNotExceeded(trade.
                getTrader(), trade.getQuantity(), trade.getSecurity())) {
                throw new RuntimeException(("The traders daily limit exceeded"));
            }

            return true;
        });

    }
    @Override
    public Map<String, FlatContract<?>> getContracts() {
        return this.contracts;
    }
}
```

Listing A.4: BiContracts for Equity Trade Execution

```
public class TradeExecutionContracts implements BiContractFactory {

    public static final String TRADER_EXECUTING_TRADE_SHOULD_OWN_TRADE = "
        traderExecutingTradeShouldOwnTrade";
    public static final String TRADER_AUTHORIZED_TO_TRADE_ON_DATE = "
        traderAuthorizedToTradeOnDate";
    public static final String TRADER_MARGIN_AVAILABLE_FOR_TRADE = "
        traderMarginAvailableForTrade";

    private ValidationServices validationService;

    private Map<String, BiFlatContract<?, ?>> contracts;

    public TradeExecutionContracts(final ValidationServices validationService) {
        this.validationService = validationService;
        this.init();
    }

    private void init() {
        this.contracts = new HashMap<>();

        this.contracts.put(TRADER_EXECUTING_TRADE_SHOULD_OWN_TRADE, (Trade trade,
            Trader trader) -> {
            if (!trader.getTraderId().equals(trade.getTrader().getTraderId())) {
                throw new RuntimeException("Invalid trader executing the trade");
            }

            return true;
        });

        this.contracts.put(TRADER_AUTHORIZED_TO_TRADE_ON_DATE, (Trade trade, Trader
             trader) -> {
            if (!this.validationService.isAuthorizedTrader(trade.getTrader())) {
                throw new RuntimeException("The trader is not authorized to trade");
            }

            return true;
        });

        this.contracts.put(TRADER_MARGIN_AVAILABLE_FOR_TRADE, (Trade trade, Trader
            trader) -> {
            if (!this.validationService.tradersDailyLimitNotExceeded(trader, trade.
                getQuantity(), trade.getSecurity())) {
                throw new RuntimeException(("The traders daily limit exceeded"));
            }

            return true;
        });


    }

    @Override
    public Map<String, BiFlatContract<?, ?>> getContracts() {
        return this.contracts;
    }
}
```

A.3 Multiple Failures Reporting using Rule4j

Listing A.5: Output listing multiple failures while executing the trade

```
java.lang.RuntimeException:
-----------------------------------------------------------------------
           PRE-CONDITIONS (6026.18 milliseconds)
-----------------------------------------------------------------------
CHAIN NAME - trading-date-validations   STATUS - FAIL

|Contract Name                      |Status|Execution Time(mills)|
------------------------------------------------------------------------
|isBusinessDay                      | FAIL |                0.0|
Failed Due to Underlying Exception -
java.lang.RuntimeException:  Trade Date is not a valid date
  at org.rahul.dbc.use_case.trading_validations.TradeContracts.lambda$init$0(
      TradeContracts.java:36)
  at org.rahul.dbc.engine.ContractExecutionEngineImpl.lambda$submitTask$0(
      ContractExecutionEngineImpl.java:22)
  at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java
      :515)
  at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
  at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
      ThreadPoolExecutor.java:1128)
  at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(
      ThreadPoolExecutor.java:628)
  at java.base/java.lang.Thread.run(Thread.java:835)

CHAIN NAME - trading-security-validations STATUS - FAIL
|Contract Name                      |Status|Execution Time(mills)|
------------------------------------------------------------------------
|isValidSecurity                    | PASS |           2033.413|
|orgPermittedToTradeSecurity        | FAIL |                0.0|
Failed Due to Underlying Exception -
java.lang.RuntimeException:  Security cannot be traded by the firm
  at org.rahul.dbc.use_case.trading_validations.TradeContracts.lambda$init$3(
      TradeContracts.java:60)
  at org.rahul.dbc.engine.ContractExecutionEngineImpl.lambda$submitTask$0(
      ContractExecutionEngineImpl.java:22)
  at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java
      :515)
  at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
  at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
      ThreadPoolExecutor.java:1128)
  at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(
      ThreadPoolExecutor.java:628)
  at java.base/java.lang.Thread.run(Thread.java:835)

CHAIN NAME - trader-consistency        STATUS - PASS
|Contract Name                        |Status|Execution Time(mills)|
-----------------------------------------------------------------------
|traderAuthorizedToTradeOnDate        | PASS |          2009.1499|
|traderMarginAvailableForTrade        | PASS |          2000.5315|
|traderExecutingTradeShouldOwnTrade   | PASS |             0.0257|

CHAIN NAME - trader-credentials        STATUS - PASS
|Contract Name                        |Status|Execution Time(mills)|
-----------------------------------------------------------------------
|TraderIdShouldBeValid                | PASS |             0.0224|
|TraderShouldNotBeEmpty               | PASS |             1.8709|
|isAuthorizedTrader                   | PASS |          2012.1312|

CHAIN NAME - trader-margins            STATUS - PASS
|Contract Name                        |Status|Execution Time(mills)|
-----------------------------------------------------------------------
|tradersDailyLimitNotExceeded         | PASS |           2001.405|
|tradersVaRExposurePermissible        | PASS |          2002.4082|
|tradersVoEExposurePermissible        | PASS |          2000.8661|
  at org.rahul.dbc.interceptor.ContractHierarchyInterceptor.executePreConditions(
      ContractHierarchyInterceptor.java:63)
  at org.rahul.dbc.interceptor.ContractHierarchyInterceptor.invoke(
      ContractHierarchyInterceptor.java:46)
  at org.rahul.dbc.UseCaseDemoTest.executeTradeWithMultipleError_WithContracts(
      UseCaseDemoTest.java:78)
```

A.4 Construction of a Higher-Order Trading Date Validation

Listing A.6: Higher Order Contract for validating Trading Date

```java
FlatContract<Trade> validBusinessDay = (Trade trade) -> {
    if (!this.validationService.isBusinessDay(trade.getTradeDate())) {
        throw new RuntimeException("Trade Date is not a valid date");
    }

    return true;
};

FlatContract<Trade> validValueDay = (Trade trade) -> {
    if (!this.validationService.isValidValueDate(trade.getValueDate(), trade.
        getTradeDate())) {
        throw new RuntimeException("Trade Value date is not a valid date");
    }

    return true;
};

FlatContract<Trade> validTradingDay = validBusinessDay.and(validValueDay);
```