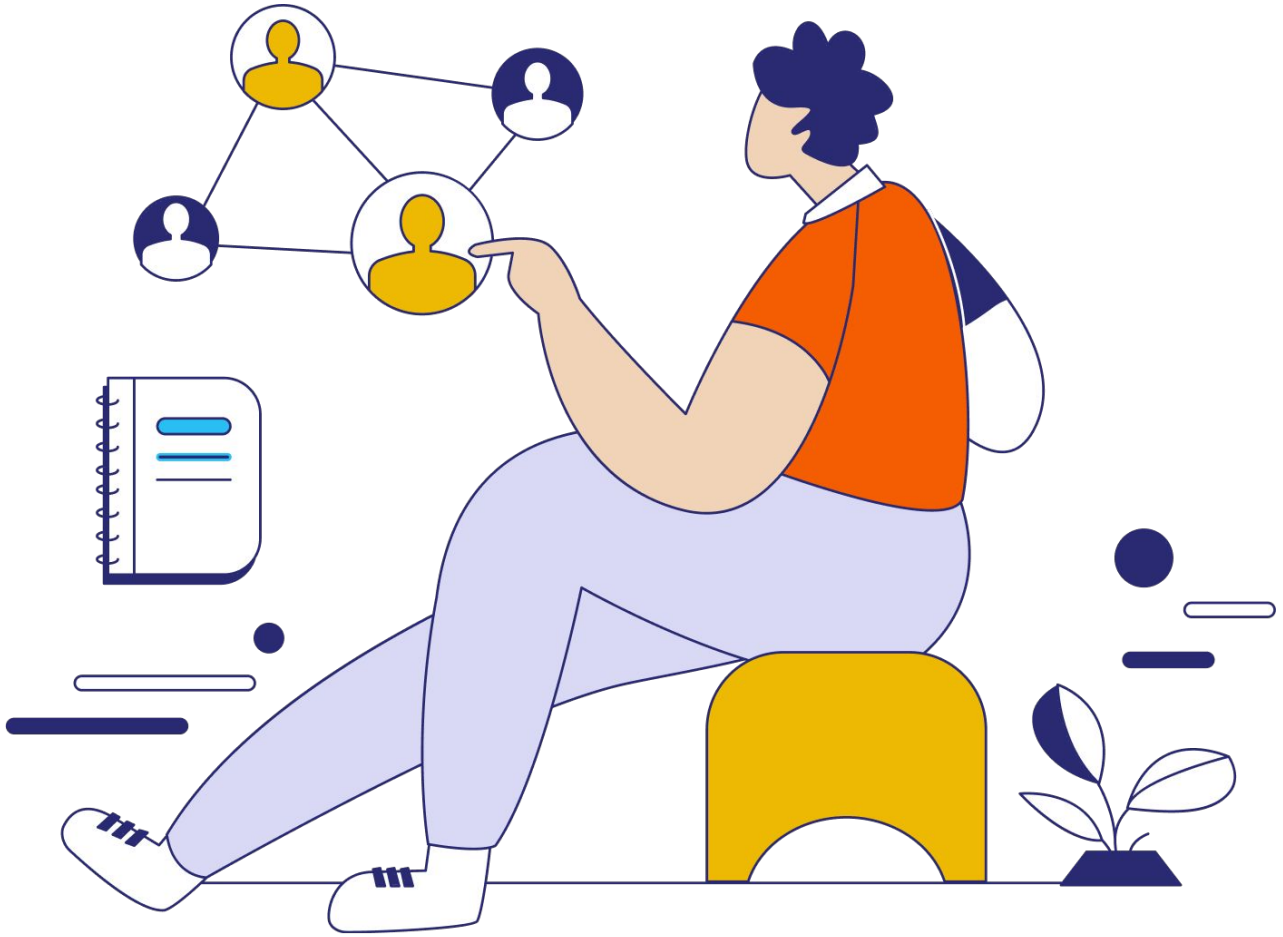
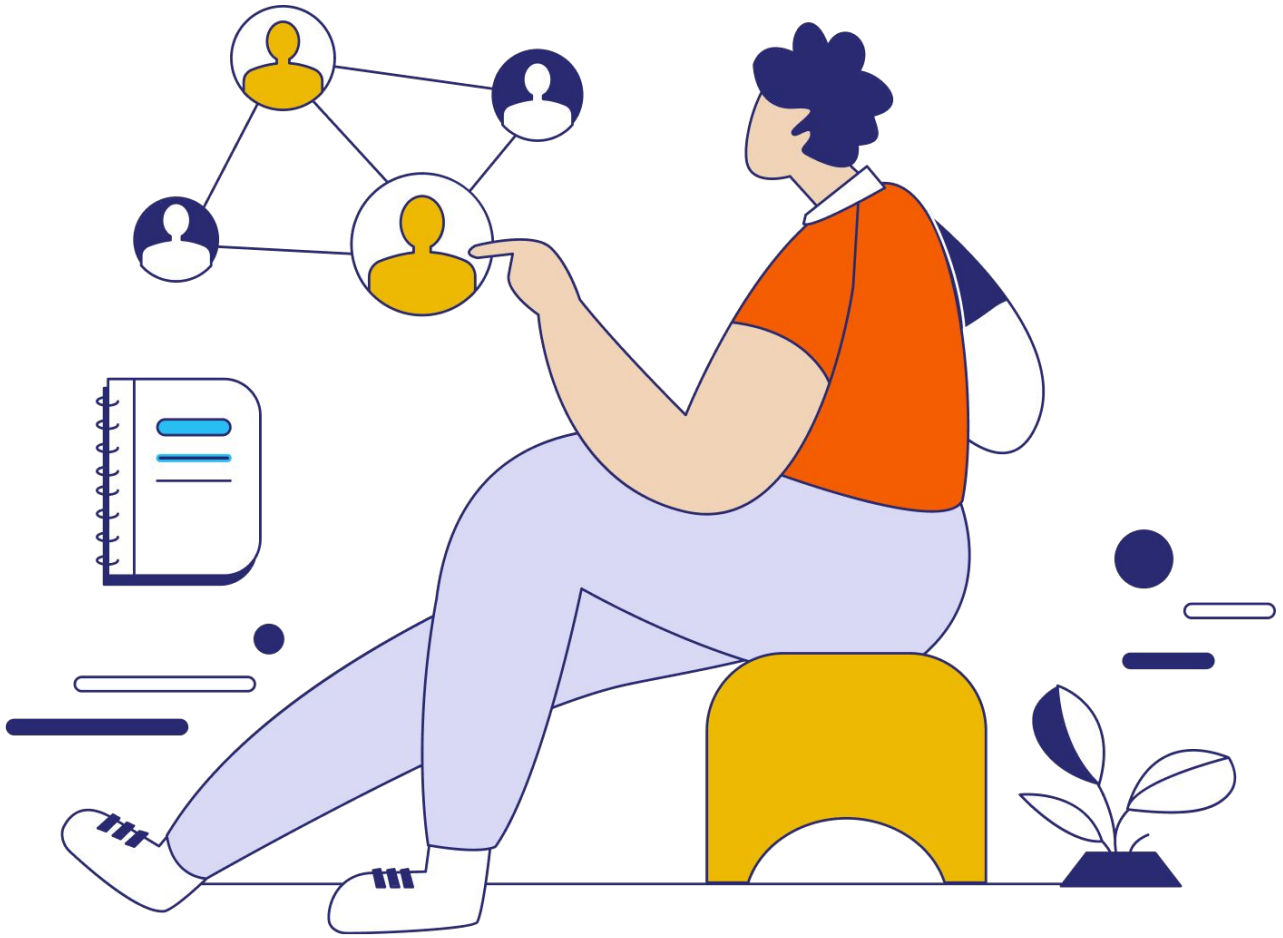
Rimantas Zukaitis,

System Architect

DSL-Based Approach on Business Rules Unit Testing



Testing is essential



Testing is essential, but really hard

Testing is a Mature Discipline in Software Engineering

EIS

- Different types of tests:
  - Unit Testing
  - Integration Testing
  - E2E Testing
  - Manual Testing
- Plenty of various libraries and frameworks for almost any ecosystem
- Popular methodologies and paradigms:
  - TDD, Automated testing, Commit Gating, CI/CD, Coverage KPIs
- Developers are well armed for day to day tasks

Challenges When Testing Business Rules

EISA

- High complexity
  - variability of possible test cases
  - dependencies between rules and their outcomes
- Enablement of non-technical users is double edged sword
  - lower access barrier means more thorough testing is needed
- Management and maintenance of tests and test data
- Complex troubleshooting process:
  - is detected issue in rule implementation or application layer?

Our Use Case

EIS

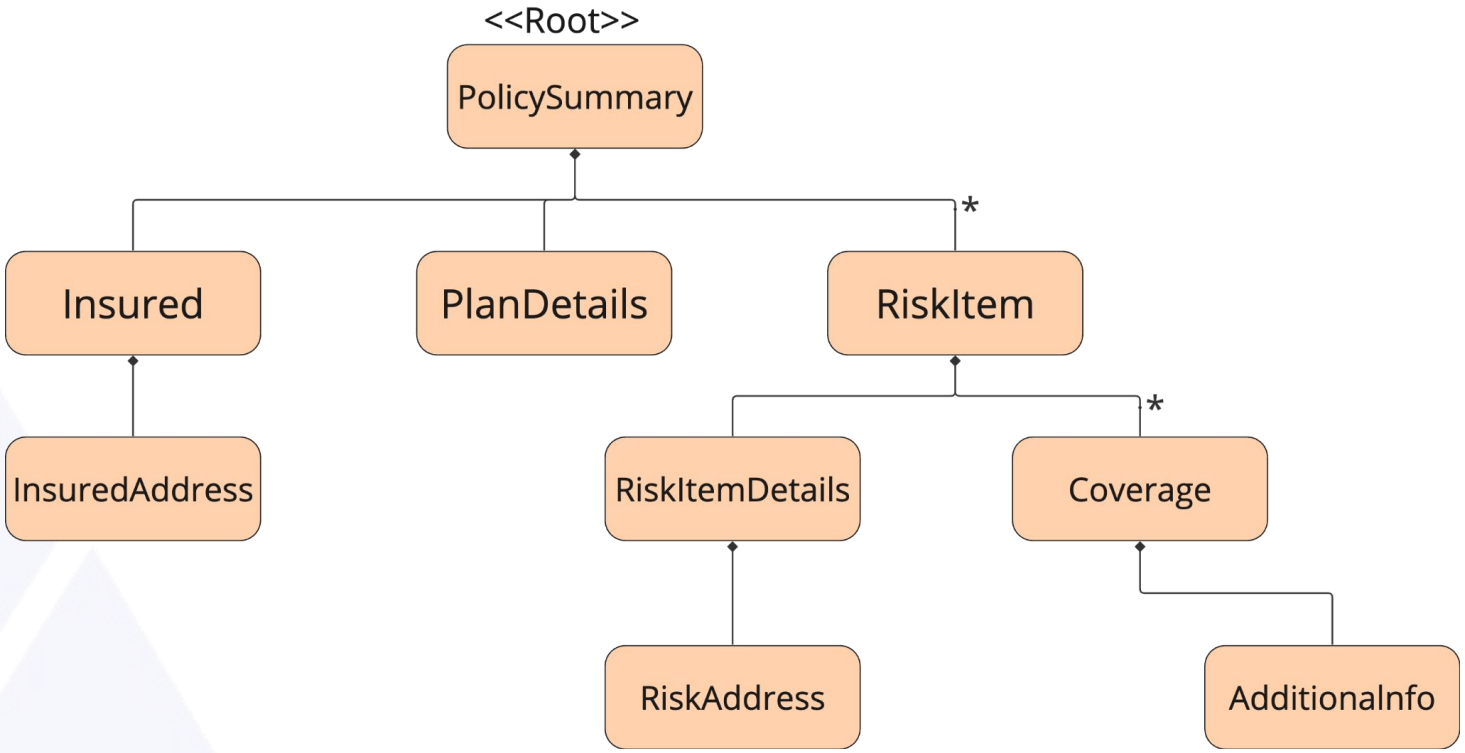
- Business Rule for Data Validation
  - Kraken validation rules engine
  - “middle ground” between constraints and complex business rules
- Rules are defined directly on the Domain Entity Model
  - domain model is defined using modelling DSL
  - there can be multiple rules on same attribute
- Up to 1000’s of rules of varying complexity
  - majority are simple, but there can be really complex ones
- The same rules are used for validation in UI and backend
- Elaborate integration layer interpreting rule results
  - Pass/fail for backend use cases
  - Interactive evaluation and presentation metadata update in UI

Anatomy of Kraken Rule

```
Rule “MinDriverAge” on Driver.age {
    When RiskItem.riskStateCd = “CA”
    Assert age > 20
    Error “err01” : “Driver age must be above 20”
}
```

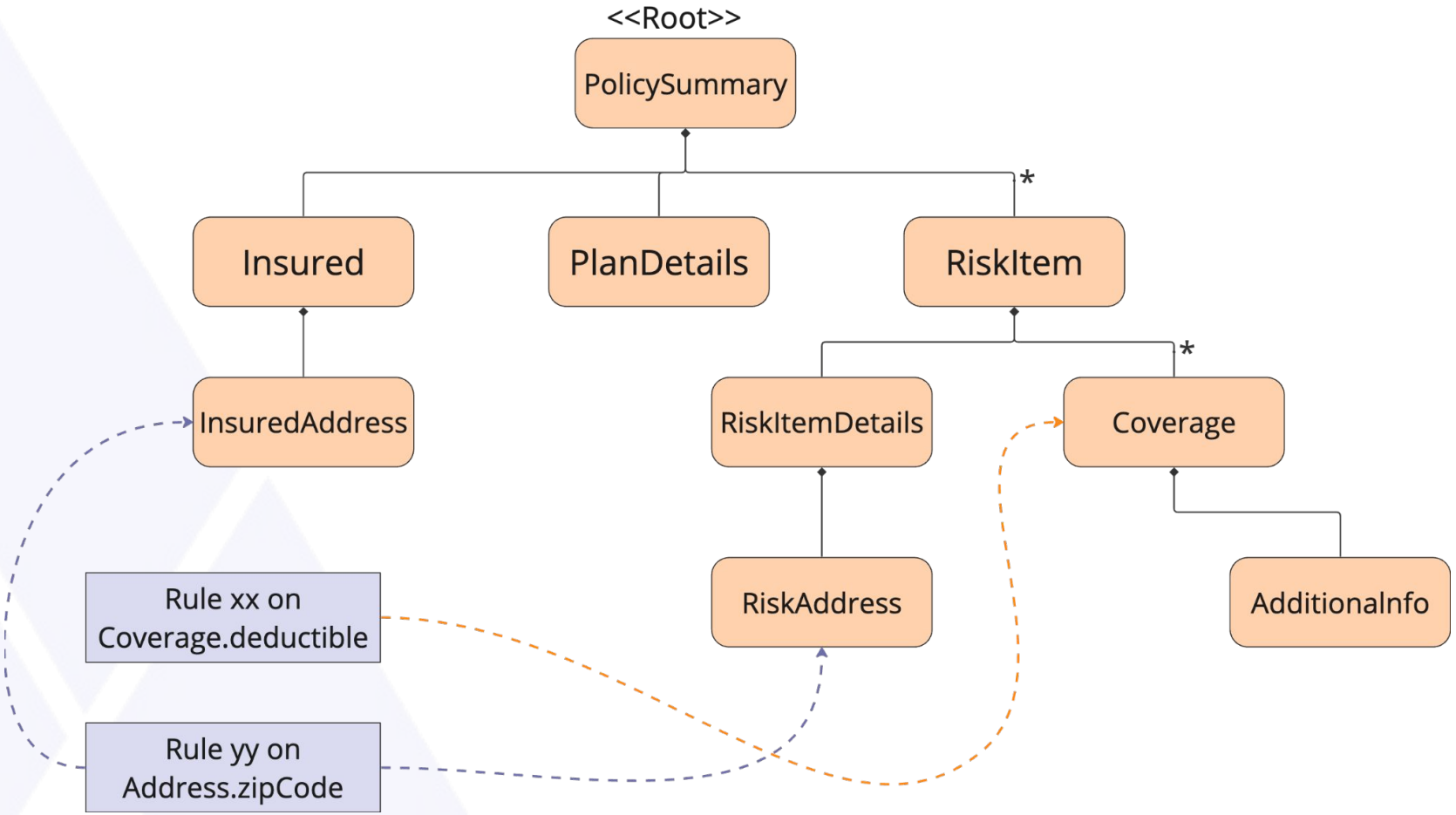
- Bound to an entity attribute in domain model
- Different rule types:
    - Validation rules
    - Default Value rules
    - Presentation rules
- Optional condition expression
- Rule payload (rule type)
- References to other entities
- Hierarchy and dependency resolution

How Kraken Rules Are Evaluated



- Rules are included in entry points (up to 100s of rules)
- Root entity instance is passed as input parameter
- Rule engine navigates to each entity and resolves required references
- Dependencies taken into account for evaluation order
- Multiple occurrences and hierarchy taken into account

How Kraken Rules Are Evaluated



- Rules are included in entry points (up to 100s of rules)
- Root entity instance is passed as input parameter
- Rule engine navigates to each entity and resolves required references
- Dependencies taken into account for evaluation order
- Multiple occurrences and hierarchy taken into account

Rule Variability by Dimensions

```
@Dimension("planCode", "Gold")
Rule "MinDriverAge" on Coverage.deductible {
    Assert deductible > 100
}

@Dimension("planCode", "Silver")
Rule "MinDriverAge" on Coverage.deductible {
    Assert deductible > 250
}

@Dimension("planCode", "Bronze")
Rule "MinDriverAge" on Coverage.deductible {
    Assert deductible > 500
}
```

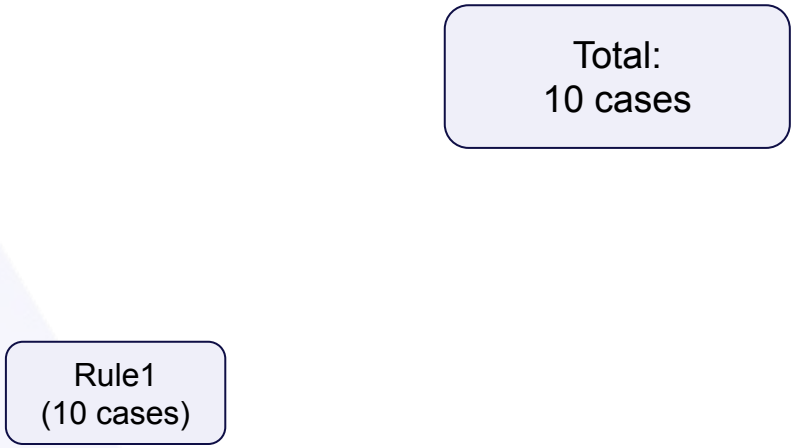
- Each rule can have multiple variations based on dimensions
- Actual variation of rule determined in runtime, based on data
- Different rules can have different variation patterns
- Additional layer of complexity from testing perspective

Compound Complexity from Testing Perspective

Total:
0 cases

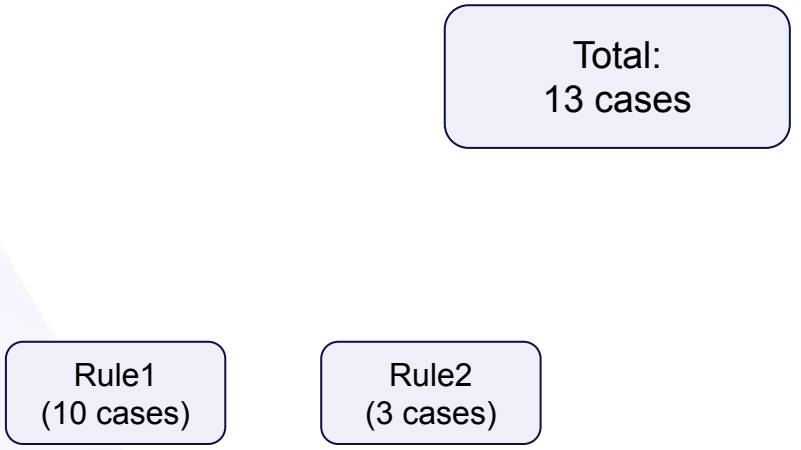
- Each rule requires N test cases to cover

Compound Complexity from Testing Perspective



- Each rule requires N test cases to cover

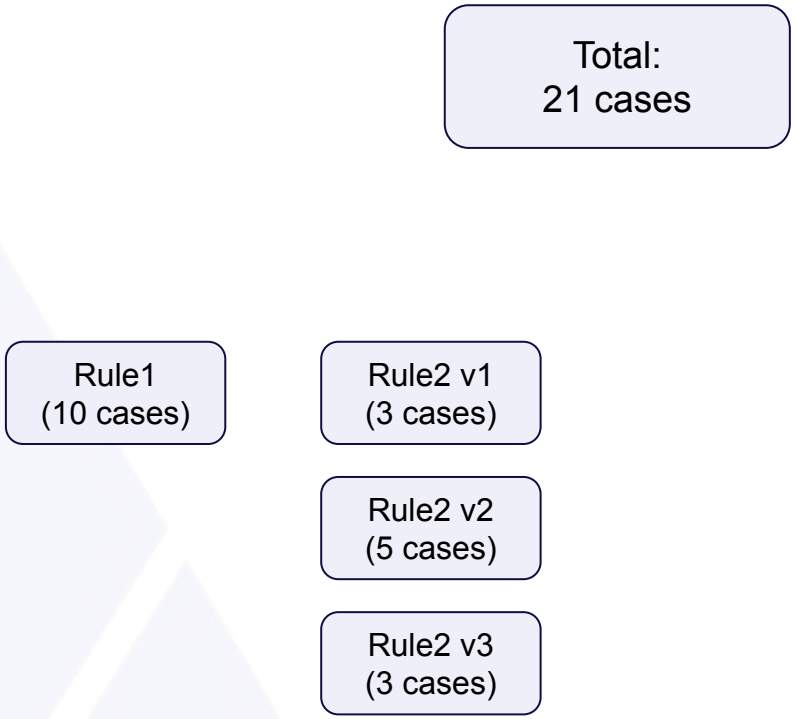
Compound Complexity from Testing Perspective



- Each rule requires N test cases to cover
- Combining them increases the complexity

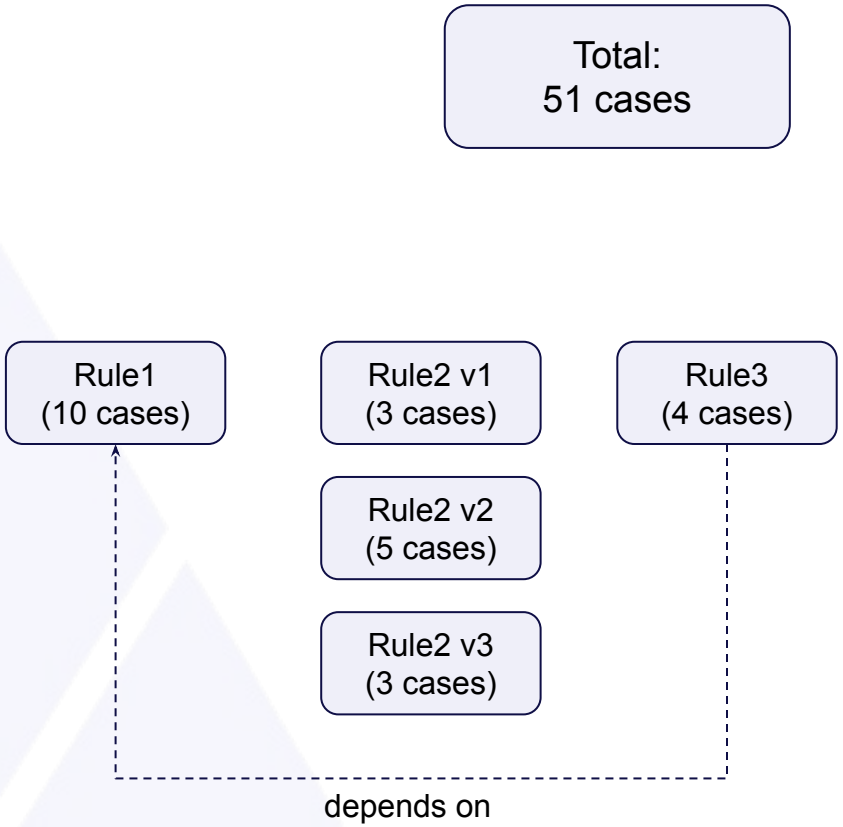
I

Compound Complexity from Testing Perspective



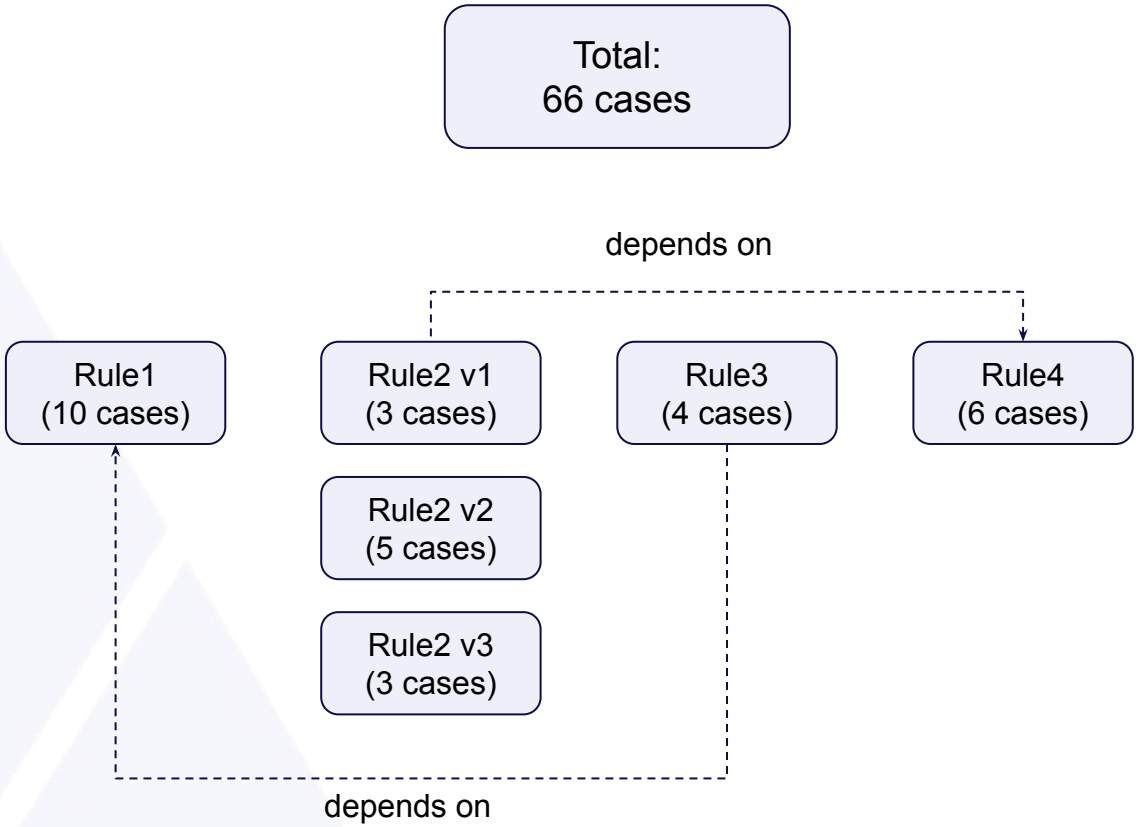
- Each rule requires N test cases to cover
- Combining them increases the complexity

Compound Complexity from Testing Perspective



- Each rule requires N test cases to cover
- Combining them increases the complexity
- Dependencies further complicate the picture

Compound Complexity from Testing Perspective



- Each rule requires N test cases to cover
- Combining them increases the complexity
- Dependencies further complicate the picture
- Full entry point coverage using typical testing means is impractical

Typical Testing Approaches

EISA

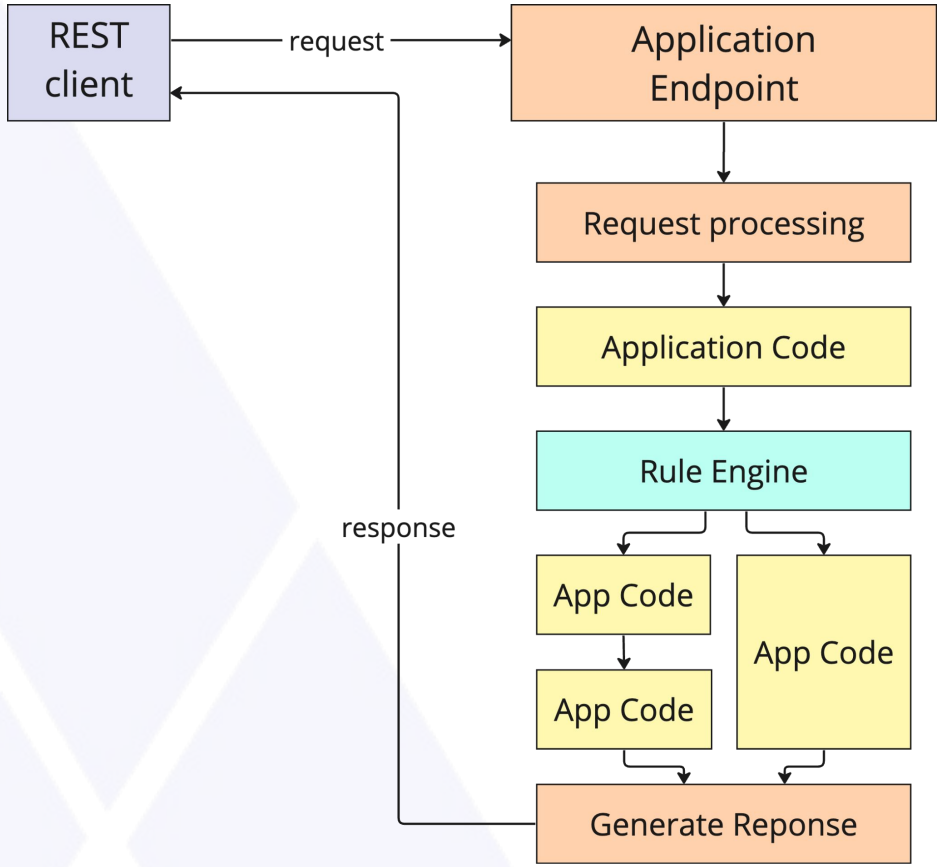
- Manual Application Testing
- Testing Using Application REST Endpoints
- Implementing JUnit Tests in Java

Manual Application Testing

EISA

- Testing the Application itself
  - reverse engineer rule behavior by observing application behavior
- Labor intensive
  - involves multiple steps not related to rule testing
  - especially costly for regression testing scenarios
- Test scenarios need to be aligned with rule implementation
- Typically focus on UI behavior
- Difficult troubleshooting
  - is this rule or not rule related?
  - test scenarios typically not aimed at particular rules
- Usually limited to happy-path scenarios
  - “rules are fine if application works” assumption
  - essentially, no specific rules testing

Testing using Application REST Endpoints



- Invoking REST endpoints with pre-build test data (JSON)
- Testing all rules in rule set, no Isolation
- Reverse engineer rule behavior from REST response
- Focused on backend behavior

PROs:

- Easy to automate
- No developer involvement
- Reuse of existing infrastructure

CONs:

- Hard to achieve coverage
- Huge amount of test data
- Fragile - sensitive to domain model changes
- Difficult to troubleshoot

Implementing JUnit Tests in Java

EIS

- Prepare test data and invoke rule engine from unit test code
- Assert on rule evaluation results
- Declarative test definition - but readable for developers only (its code)
- Offers very good coverage, but at significant cost

PROs:

- Promotes good practices (TDD)
- Allows to test rules in isolation
- Great choice of assertions
- Somewhat easier to maintain test data

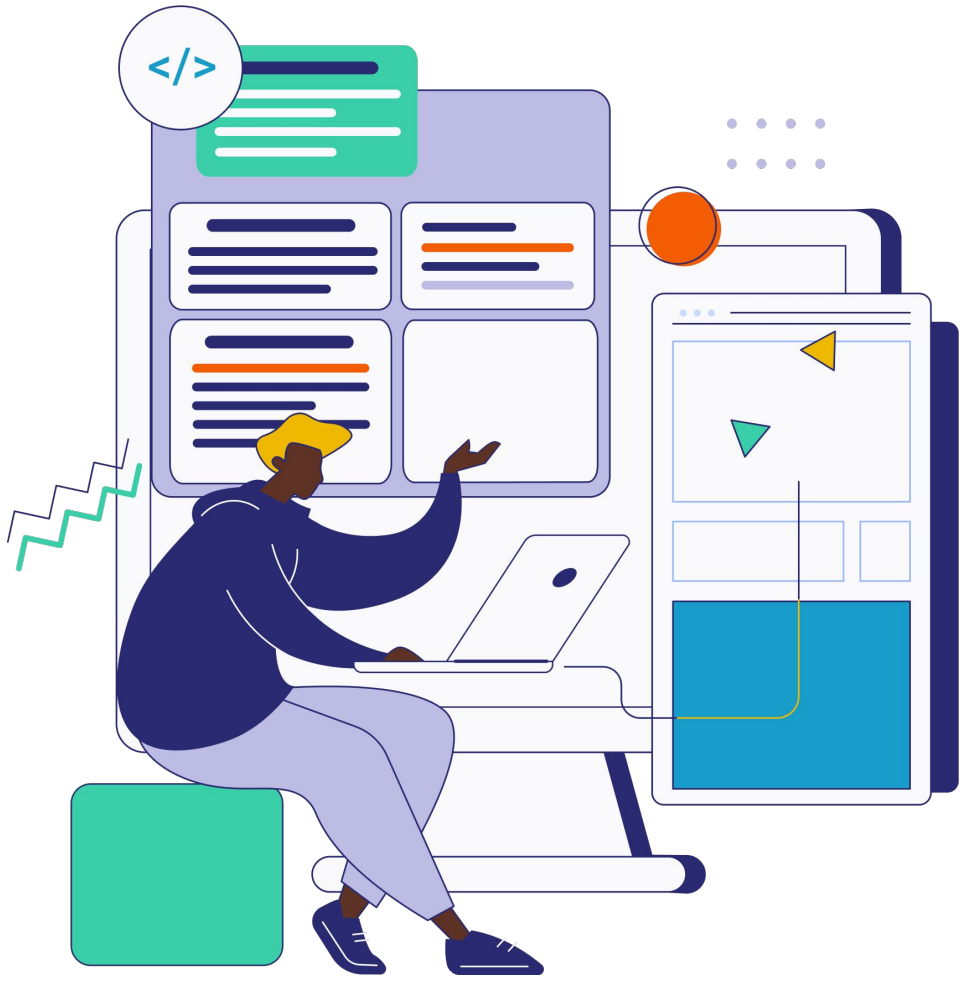
CONs:

- Requires developer effort
- Effort intensive
- Inefficient - preparatory vs test code ratio can be 50:1 or more
- Antipattern - code (tests) depending on configuration (rule definitions)

Building a Rules Testing Framework

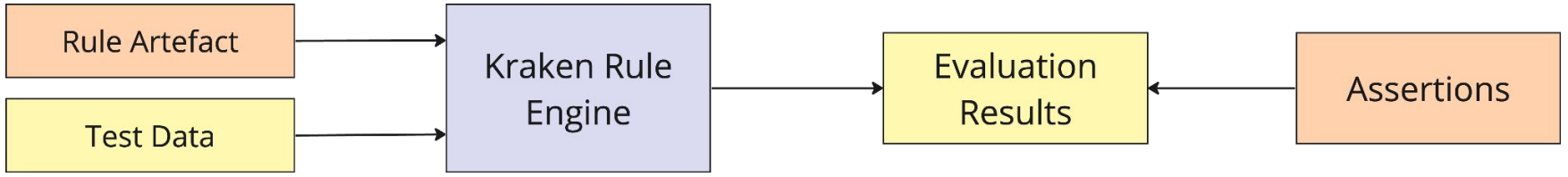
Target goals:

- Declarative test definitions
- Same abstraction level as rules development
- Support different levels of granularity testing
- Focus on automation and ease of use
- Simplify test data management
- No code approach - no programming skills are necessary

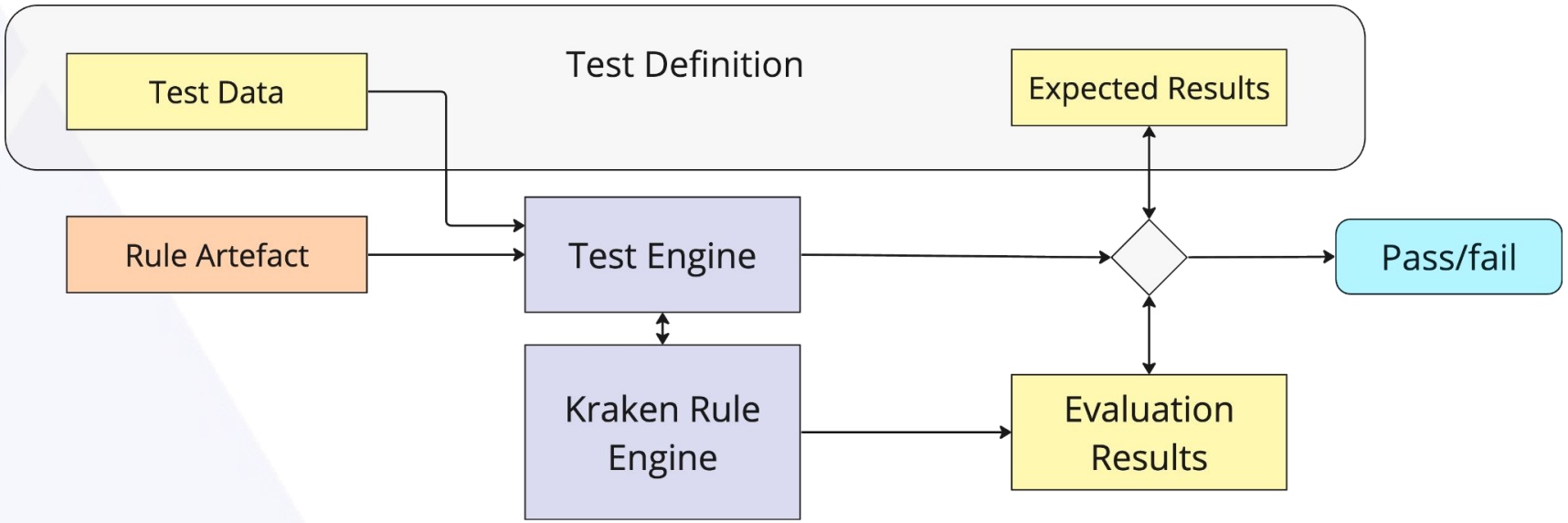


Modelling a Unit Test for Rule

We want to achieve same capabilities as in JUnit, but avoid need to write code.



Moving testing logic from code to configuration (model):



Modelling a Unit Test for Rule

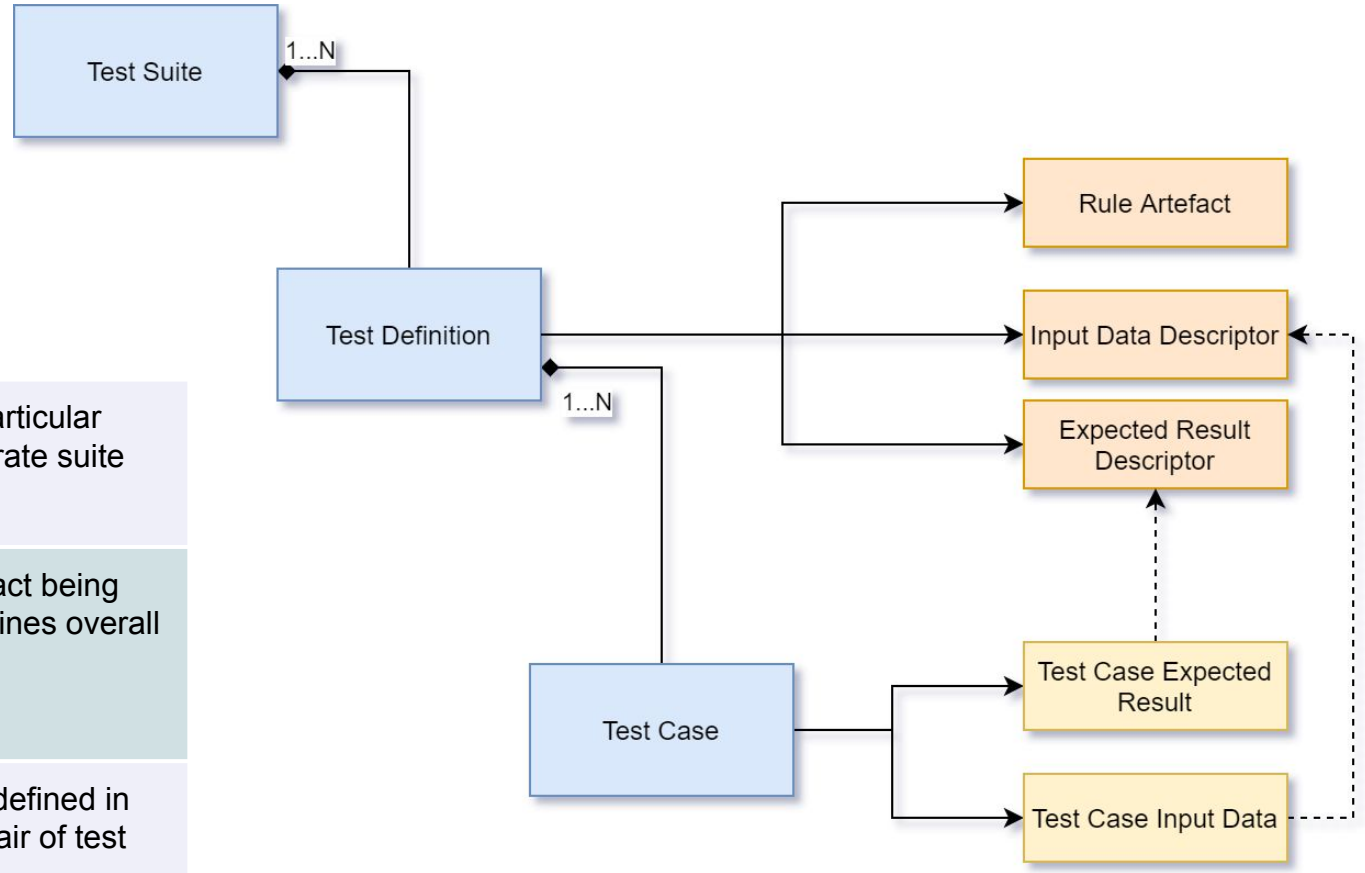
- Each test is defined on one rule artifact:
  - Single Rule
  - Subset of rules
  - Whole entry point
- Each test defines input data template and expected assertions
- Multiple cases for same test
  - Variability on input data
  - Variability on asserted outcomes

Managing Test Data

- Rules are evaluated on root object
- Most of the data is the same for individual test cases
- Need a mechanism to:
  - create variations of input data
  - parametrise assertions for each input data variation

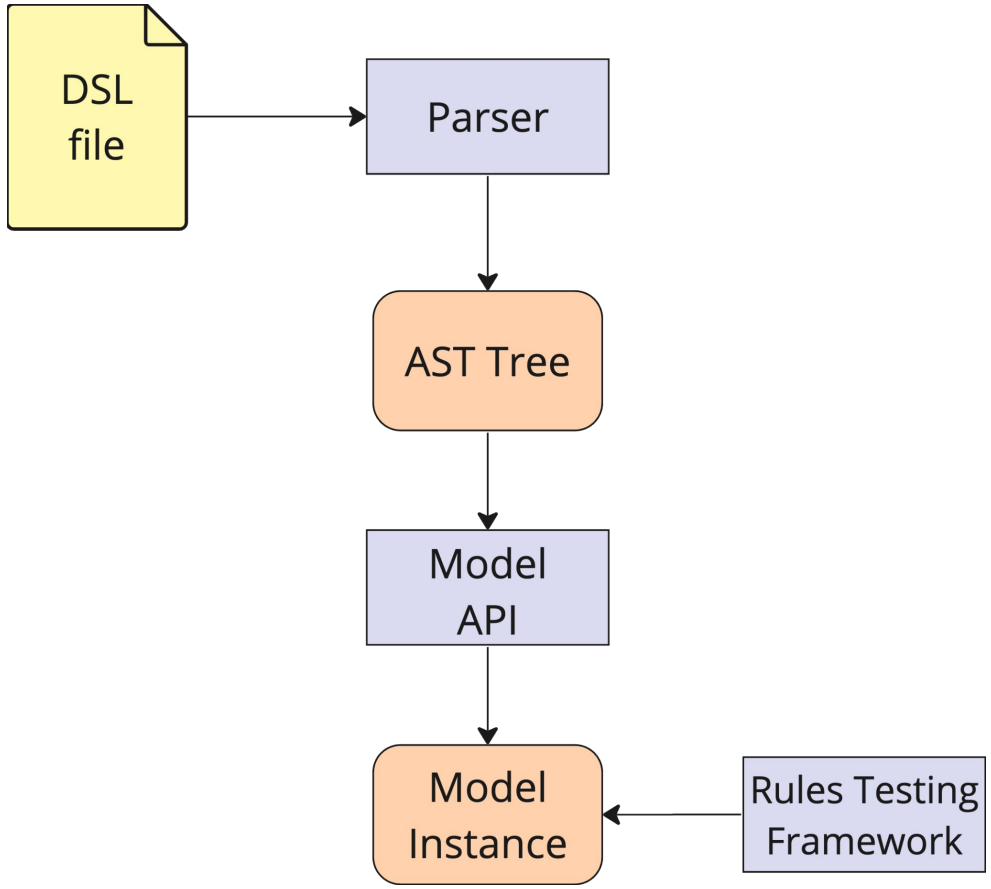
Main Model Elements

Test Suite	Defines a list of tests to be evaluated for particular scenario. User has ability to evaluate separate suite only.
Test Definition	Defines actual rule test. Specifies rule artifact being tested, input data format and variables, defines overall structure of the test
Test Case	Defines a separate case of particular test, defined in test definition. Represents a set of single pair of test input data and expected results. For example if rule is applicable in 10 states, it could have one test with 10 test cases.



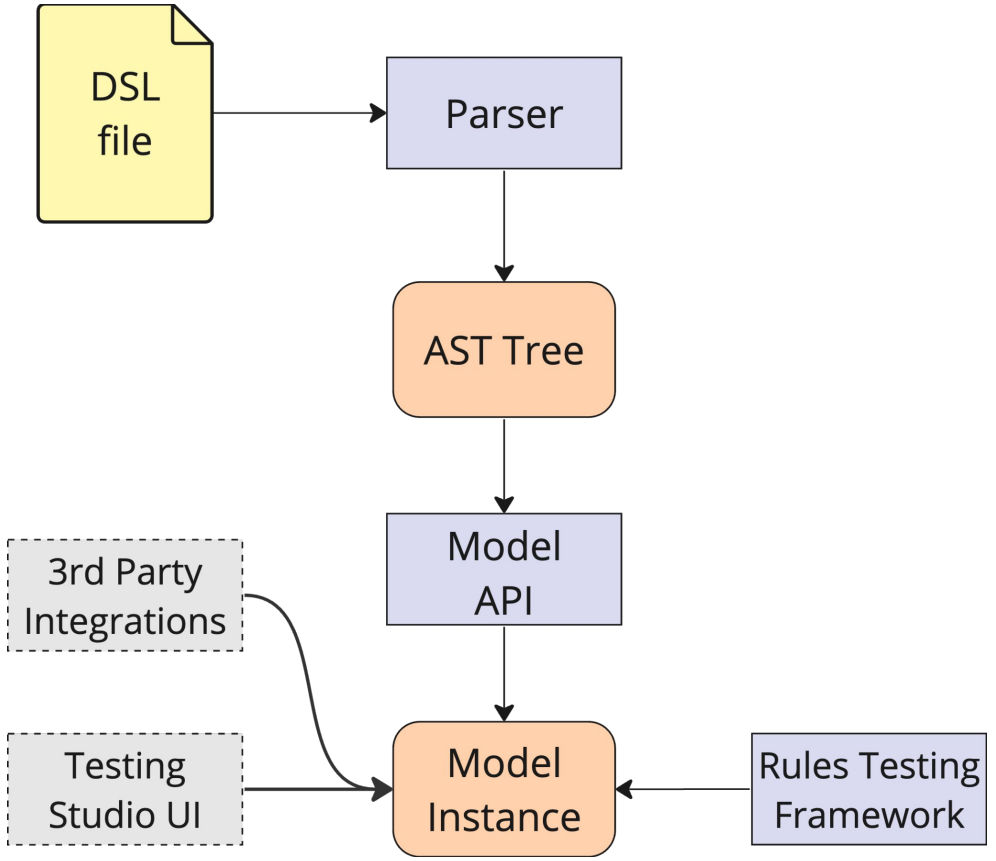
Embracing DSL Approach

- Using ANTLR library
- DSL files translated model in runtime
  - Better control than depending on AST
  - Possibility for extension
  - Model can be created without DSL
- Efficient and cheap approach to represent models
  - More readable than XML or JSON
  - Cheaper than building dedicated UI
- Especially useful in prototyping phase
- Read-write capability



Embracing DSL Approach

- Using ANTLR library
- DSL files translated model in runtime
  - Better control than depending on AST
  - Possibility for extension
  - Model can be created without DSL
- Efficient and cheap approach to represent models
  - More readable than XML or JSON
  - Cheaper than building dedicated UI
- Especially useful in prototyping phase
- Read-write capability



Test Definition Structure

- Test Definition Name
- Rule Artifact
- Input Data Description
- Variables
- Parametrized Assertions

Test Definition Syntax - Header

```
TestDefinition "AssertDriveAgeValidation" {

    Rule "MinDriverAge"

    ...
```

Test definition name identifies test, must be unique

Rule artifact can be:

- single rule
- set of rules
- entry point

Specifying Input Data

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age
        }
    }
```

Entity specifies data for root entity

Data from specified JSON file will be used

Specifying Input Data

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity generated {
            $driverAge -> /vehicles/0/driver/age
        }
    }
```

generated keyword will generate empty entity instance using domain model metadata

Parametrizing Input Data

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age,
            “John” -> /vehicles/0/driver/name
        }
    }
```

Following block contains data overrides

Values in entity will be overwritten with specified ones

Parameterizing Input Data

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age,
            “John” -> /vehicles/0/driver/name
        }
    }
```

Following block contains data overrides

Values in entity will be overwritten with specified ones

In case of variable, its value will be resolved from test case

Specifying Dimension Data

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age,
            “John” -> /vehicles/0/driver/name
        }
        Dimensions “data/dimensions.json”
    }
```

Dimensions allows to specify map with dimension values

Specifying Dimension Data

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age,
            “John” -> /vehicles/0/driver/name
        }
        Dimensions from entity
    }
```

from entity keyword will extract dimensions from root entity, instead of loading them from map

Defining variables

```
TestDefinition "AssertTemplateDefaultsTesting" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age
        }
    }
    Variables {
        driverAge : Integer
        isValid : Boolean
    }
```

Variables are used to parametrize test input data with values, specific for each test case

Variables must be defined and used in test definition, and values will be supplied in test cases

Using Parameterized Asserts

```
TestDefinition "AssertDriveAgeValidation" {
    Rule "MinDriverAge"
    Input {
        Entity "data/entity.json" {
            $driverAge -> /vehicles/0/driver/age
        }
    }
    Variables {
        driverAge : Integer
        isValid : Boolean
    }
    Assert Driver.age $isValid as validity
 }
```

Parameterized asserts will be asserted on each test case

Parameter values will be taken from test case

Test Case Structure

- Reference to Test Definition
- Variable values
- Test Case specific assertions

Specifying Variable Values

```
For TestDefinition "AssertDriveAgeValidation" {
   Test Case “AgeValid” {
      Variables {
          age : 21
          isValid : true
      }
   }
   Test Case “AgeInvalid” {
      Variables {
          age : 19
          isValid : false
      }
   }
```

Test cases defined in scope of particular test definition

Each test case is identified by name

Variable values will be used to build final data image used for testing

Test Case Specific Asserts

```
For TestDefinition "AssertDriveAgeValidation" {
   Test Case “AgeValid” {
      Variables {
          age : 21
          isValid : true
      }
      Assert Driver.age has no error on /vehicles/0/driver
   }
   Test Case “AgeInvalid” {
      Variables {
          age : 19
          isValid : false
      }
      Assert Driver.age has no error "err01" on
                f88889d8-e1d3-4b4f-850d-383a9e20ae31
   }
```

Each test case can define additional assert statements

They will be asserted for this test case only

In case there are multiple instances of same entity, it needs to be specified by ID of path

Multiple assertion types:

- default value,
- validity,
- visibility,
- accessibility,
- has error/warning/info message
- has not error/warning/info msg

Test Suite Definition

```
TestSuite “Driver Tests” [
    “AssertDriveAgeValidation”,
    “AssertDriverNameValidation”,
    “AssertDriverSSNVisibility”
]
```

Test Suite is a list of test definitions, referenced by name

All included tests will be executed, when executing test suite

It’s offered as a way to organize semantically related tests

Sample Case - Writing a Rule Unit Test

```
Rule "Insured.licenseAcquired-validate" on Insured.licenseAcquired
{
  Assert birthDate < licenseAcquired
         and NumberOfYearsBetween(birthDate, licenseAcquired) >= 16
}
```

Test Definition

```
TestDefinition "rules-Insured" {
    Rule "Insured.licenseAcquired-validate"
    Input {
        Entity generated {
            $birthDate        -> /insured/birthDate
            $licenseAcquired  -> /insured/licenseAcquired
        }
    }
    Variables {
        birthDate: Date
        licenseAcquired: Date
    }
}
```

Test Definition

```
TestDefinition "rules-Insured" {
    Rule "Insured.licenseAcquired-validate"
    Input {
        Entity generated {
            $birthDate        -> /insured/birthDate
            $licenseAcquired  -> /insured/licenseAcquired
        }
    }
    Variables {
        birthDate: Date
        licenseAcquired: Date
    }
}
```

Test Definition

```
TestDefinition "rules-Insured" {
    Rule "Insured.licenseAcquired-validate"
    Input {
        Entity generated {
            $birthDate        -> /insured/birthDate
            $licenseAcquired  -> /insured/licenseAcquired
        }
    }
    Variables {
        birthDate: Date
        licenseAcquired: Date
    }
}
```

Modelling Test Cases

```
For TestDefinition "rules-Insured" {


}
```

Modelling Test Cases

```
For TestDefinition "rules-Insured" {
    TestCase "Should be valid if insured acquired license when he was at least 16"
{
        Variables {
            birthDate: 1970-05-21
            licenseAcquired: 1986-05-21
        }
        Assert Insured.licenseAcquired is valid
    }




}
```

Modelling Test Cases

EISA

```
For TestDefinition "rules-Insured" {
    TestCase "Should be valid if insured acquired license when he was at least 16" {
        Variables {
            birthDate: 1970-05-21
            licenseAcquired: 1986-05-21
        }
        Assert Insured.licenseAcquired is valid
    }
    TestCase "Should not be valid if insured acquired license before he was 16" {
        Variables {
            birthDate: 1970-05-21
            licenseAcquired: 1986-05-20
        }
        Assert Insured.licenseAcquired is not valid
    }
}
```

Executing Tests

- During build cycle
  - integration through maven plugin
  - generates JUnit tests, runs during test phase
  - free integration in already existing pipelines (commit gating, release, etc)
- Through programmatic Java API
- Remotely through REST endpoint
- Interactively as part of Rules Studio web application

Test Granularity and Coverage

EISA

Rules Unit Tests

- Exhaustive coverage of cases
- Can test combinations of rules
- Applicability based on complexity

Tackle complexity

Integration Testing

- Rule Entry Points per Use Case
- Ensure that rules are included
- Rely on Unit tests for individual rules

Bringing it together

E2E and Manual Testing

- Test General Application Behavior
- Ensure Entry Points are triggered
- Rely on lower level tests for details

“Happy Paths”

Further Work

- Methodology to calculate coverage
  - what percentage of rules are covered?
  - what percentage of corner test cases covered?
  - what percentage of rules are actually triggered in entry points?
- More user friendly formats to represent test cases (e.g. excel table)
- Auto generate test definition stubs from rule definitions

Lessons Learned

- Sometimes custom solution is a way to go, especially in utility domains, like testing
  - good fit when applied to custom cases
  - simplifies user effort by order of magnitude
  - ability to tailor and address pain-points, e.g. integration
- Model first approach best for prototyping
  - Avoid “visual thinking”
- DSL approach shines here, due to efficiency and speed of implementation
- UIs and related tooling can be implemented later in backward compatible fashion, as its based on model

Thank You

Rimantas Zukaitis

rzukaitis@eisgroup.com

@RimasZuk