**Lab Exercise 20- Debugging in Drools**

Debugging Drools rules can be challenging because Drools is declarative and the engine works on the rules and facts in the working memory. However, there are various techniques and tools available to make this process easier. In this lab exercise, we will focus on debugging techniques in Drools and explore how to use logging, breakpoints, and the Drools Audit Log to trace the flow of rule execution and fact manipulation.

**Objective:**

* Learn how to debug Drools rules using **logging**, **audit logs**, and **breakpoints**.
* Understand how to trace rule execution, fact insertion, and rule firing.
* Analyze rule conflicts, infinite loops, and unintended rule firing.

**Scenario:**

We will simulate an e-commerce system where different rules apply discounts based on the order total, and rules manage inventory. We will intentionally create a situation where a rule does not fire as expected, and we will use debugging techniques to diagnose and fix the problem.

**Step 1: Define Java Model Class (Order.java)**

package com.example.model;

public class Order {

private String id;

private double totalAmount;

private boolean processed;

private double discount;

public Order(String id, double totalAmount) {

this.id = id;

this.totalAmount = totalAmount;

this.processed = false;

this.discount = 0.0;

}

public String getId() {

return id;

}

public double getTotalAmount() {

return totalAmount;

}

public void setTotalAmount(double totalAmount) {

this.totalAmount = totalAmount;

}

public boolean isProcessed() {

return processed;

}

public void setProcessed(boolean processed) {

this.processed = processed;

}

public double getDiscount() {

return discount;

}

public void setDiscount(double discount) {

this.discount = discount;

}

@Override

public String toString() {

return "Order{id='" + id + "', totalAmount=" + totalAmount + ", processed=" + processed + ", discount=" + discount + '}';

}

}

**Step 2: Define Drools Rules (debuggingRules.drl)**

We will intentionally create two rules, one that applies a discount and another that checks if the order is processed. We'll introduce an issue where one rule doesn’t fire as expected, and use debugging techniques to resolve it.

package com.example.rules;

import com.example.model.Order;

// Rule to apply a discount to orders with a total amount >= 100

rule "Apply Discount to High Value Orders"

when

$order : Order(totalAmount >= 100, processed == false)

then

$order.setDiscount(10.0); // Apply a discount

System.out.println("[INFO] Discount applied to order " + $order.getId());

update($order); // Make sure Drools knows the fact has changed

end

// Rule to process the order after applying the discount

rule "Process Order"

when

$order : Order(discount > 0, processed == false)

then

$order.setProcessed(true); // Mark the order as processed

System.out.println("[INFO] Order " + $order.getId() + " is now processed.");

update($order); // Update the order fact in the session

end

// Rule that should not fire (for debugging purposes)

rule "Debug Rule - Unexpected"

when

$order : Order(discount < 0)

then

System.out.println("[DEBUG] This rule should not fire.");

end

**Step 3: Define kmodule.xml**

<?xml version="1.0" encoding="UTF-8"?>

<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">

<kbase name="debuggingKBase" packages="com.example.rules">

<ksession name="debuggingKSession" type="stateful"/>

</kbase>

</kmodule>

**Step 4: Implement the Test Class (DebuggingTest.java)**

We’ll introduce **logging** to trace the rule execution flow and use the Drools **audit log** to debug rules.

package com.example.model;

import org.kie.api.KieServices;

import org.kie.api.runtime.KieContainer;

import org.kie.api.runtime.KieSession;

import org.kie.api.logger.KieRuntimeLogger;

public class DebuggingTest {

public static void main(String[] args) {

// Load KieServices and KieContainer

KieServices ks = KieServices.Factory.get();

KieContainer kContainer = ks.getKieClasspathContainer();

// Create a stateful Kie session

KieSession kSession = kContainer.newKieSession("debuggingKSession");

// Create a runtime logger to generate an audit log file

KieRuntimeLogger logger = KieServices.Factory.get().getLoggers().newFileLogger(kSession, "audit");

// Create a new order with a total amount of 120

Order order = new Order("Order-001", 120);

kSession.insert(order); // Insert the order into the session

// Fire all rules

kSession.fireAllRules();

// Dispose of the session

kSession.dispose();

// Close the logger to ensure the log file is saved

logger.close();

}

}

**Step 5: Debugging Techniques**

**. Console Logging:**

* Use **System.out.println** statements in your rules to trace rule execution.
* For example, each rule in the debuggingRules.drl file contains a logging statement like System.out.println("[INFO] Discount applied to order ...");. This allows you to track when a rule is triggered and executed.

**2. Audit Log:**

* The **audit log** provides a detailed trace of all rule activations, fact insertions, modifications, and deletions.
* In the DebuggingTest.java class, the line KieRuntimeLogger logger = KieServices.Factory.get().getLoggers().newFileLogger(kSession, "audit"); creates an audit log file named audit.log. The log will contain information about the rule-firing sequence and fact changes.
* After running the program, you can open the audit.log file to analyze the rule execution flow.

**3. Breakpoints in IDE:**

* You can also set breakpoints inside your **Drools code** (e.g., DebuggingTest.java) or in the **rules** (if your IDE supports it, like IntelliJ IDEA with Drools support) to see when facts are inserted or modified.

**4. KieScanner for Rule Monitoring:**

* **KieScanner** can be used for continuous scanning of changes in the knowledge base. This can help if you need to re-load rules dynamically and debug issues with dynamic rule sets.

**Step 6: Run the Application**

Run the DebuggingTest.java class. The audit log will be generated as audit.log, and the following output should be logged to the console if the rules work correctly:

[INFO] Discount applied to order Order-001

[INFO] Order Order-001 is now processed.

**Step 7: Debugging and Fixing Issues**

Let’s assume that one of the rules doesn’t fire as expected (for instance, the Process Order rule). Follow these steps to debug and fix the issue:

1. **Check Logging Output**:
   * Look at the console output to see if the Process Order rule is firing. If it’s not firing, ensure that the condition (discount > 0) is being met.
2. **Review the Audit Log**:
   * Open the audit.log file. It will show you when the Order fact was inserted and which rules were activated. This can help identify why the Process Order rule didn’t fire.
3. **Fix the Rule Condition**:
   * If you realize that the rule condition is too strict or not being met, modify the rule condition and re-run the test. For example, if the discount > 0 condition is incorrect, you may want to adjust it to discount >= 0.

**Step 8: Analyze Audit Log**

Look for entries similar to the following in the audit.log:

<org.drools.core.audit.WorkingMemoryLog>

<version>6.1</version>

<events>

<org.drools.core.audit.event.ObjectLogEvent>

<type>1</type>

<factId>1</factId>

<objectToString>Order{id=&apos;Order-001&apos;, totalAmount=120.0, processed=false, discount=0.0}</objectToString>

</org.drools.core.audit.event.ObjectLogEvent>

<org.drools.core.audit.event.ActivationLogEvent>

<type>4</type>

<activationId>Apply Discount to High Value Orders [1]</activationId>

<rule>Apply Discount to High Value Orders</rule>

<declarations>$order=Order{id=&apos;Order-001&apos;, totalAmount=120.0, processed=false, discount=0.0}</declarations>

<factHandleIds>1</factHandleIds>

</org.drools.core.audit.event.ActivationLogEvent>

This shows the sequence of fact insertions, rule activations, and fact updates, allowing you to trace how facts are manipulated and why certain rules were (or were not) triggered.

**Key Points:**

* **Console Logging**: Use System.out.println in rules to trace their execution.
* **Audit Logging**: Use KieRuntimeLogger to generate audit logs to trace rule activations and fact manipulation.
* **Breakpoints in IDE**: Use breakpoints to inspect the state of the working memory at any point during rule execution.
* **Fact Inspection**: Use the audit log or breakpoints to inspect facts in the working memory and analyze why a rule is (or isn’t) firing.

This lab exercise demonstrates how to debug Drools rules using logging, breakpoints, and audit logs to identify issues like rule conflicts, fact manipulation errors, or unexpected rule behavior.