

Hands on Complex Event Processing: Lab Guide

Hands on Complex Event Processing Information	
Technology/Product	Red Hat JBoss Business Rules Management Platform
Difficulty	3
Time	90 minutes
Prerequisites	Knowledge of Java, Eclipse, Junit, Maven

In this lab you will...

- Learn about Complex Event Processing
- Modify and create Complex Event Processing business rules

This is any basic information someone should know before starting your lab. It could be some background information on the subject or something that is covered by your presentation, but is not necessarily provided in the steps of the lab below. It could also just be a further, more detailed, explanation of why you're doing this lab, or how this might help someone in their real lives.

BEFORE YOU BEGIN...

Raise your hand if you have any questions! We are here to help.

Built on Eclipse JBoss® Developer Studio provides superior support for your entire development lifecycle. It includes a broad set of tooling capabilities and support for multiple programming models and frameworks, including Java™ Enterprise Edition 6, RichFaces, JavaServer Faces (JSF), Enterprise JavaBeans (EJB), Java Persistence API (JPA), and Hibernate®, JAX-RS with RESTEasy, Contexts Dependency Injection (CDI), HTML5, and many other popular technologies. It provides developer choice in supporting multiple JVMs, productivity with Maven, and in testing with Arquillian. It is fully tested and certified to ensure that all its plug-ins, runtime components, and their dependencies are compatible with each other.

The lab will use JBoss Developer Studio as the IDE and will run all code inside of Junit Tests. JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks. Familiarity with JUnit is helpful but not required.

Have fun!

KEY TERMS

CEP is short for Complex Event Processing. The terms will be used interchangeably throughout the labs.

A **Fact** is a Java POJO. The rules engine analyzes Facts.

An **Event** is both a significant change in state at a particular point in time a Fact that has temporal metadata associated with it. In CEP the rules engine analyzed Events.

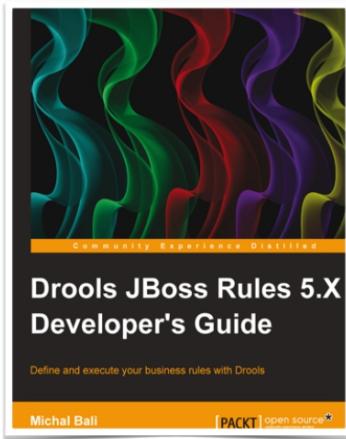
Kie stands for “Knowledge Is Everything.” It is an upstream, community umbrella for projects in the Drools family. Drools Expert, Fusion, OptaPlanner, UberFire and jBPM are included.

Drools Fusion is the upstream project that contains the CEP functionality.

A **.drl** is a file that holds Drools rules.

LAB INSTRUCTIONS

The examples in this lab are from a financial services domain and borrow heavily from **Drools JBoss Rules 5.x** by Michl Bali.



The labs consist of the following Maven projects :

- summit2015-model
- summit2015-tests
- lab1
- lab2
- lab3
- lab4
- lab5

You will import each project into Eclipse and work through the corresponding text. If you encounter any problems along the way ask for help from one of the proctors! Complex Event Processing is a “complex” topic.

PROJECT OVERVIEW

summit2015-model

This project contains the domain model for the labs

summit2015-tests

Contains a base JUnit test handles the boilerplate code for running rules applications inside of Junit

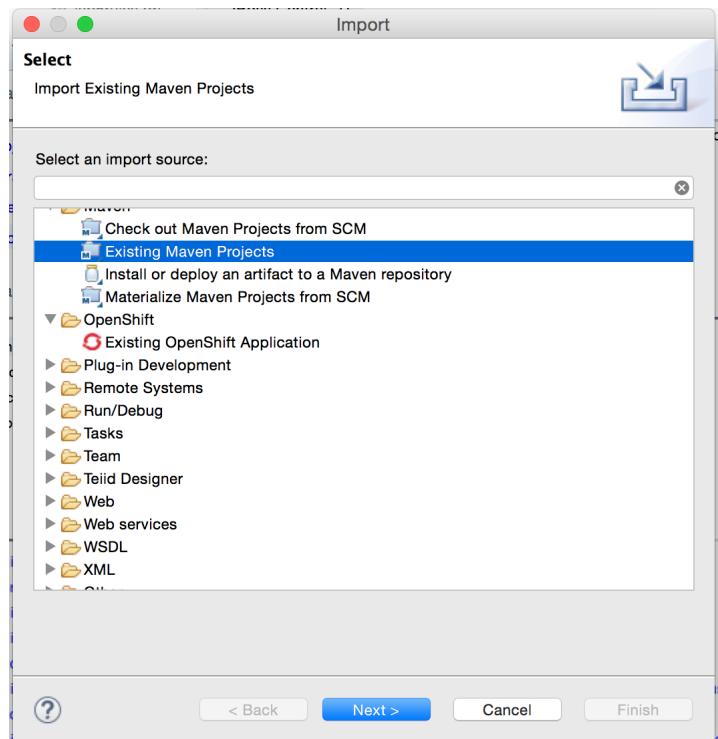
lab1

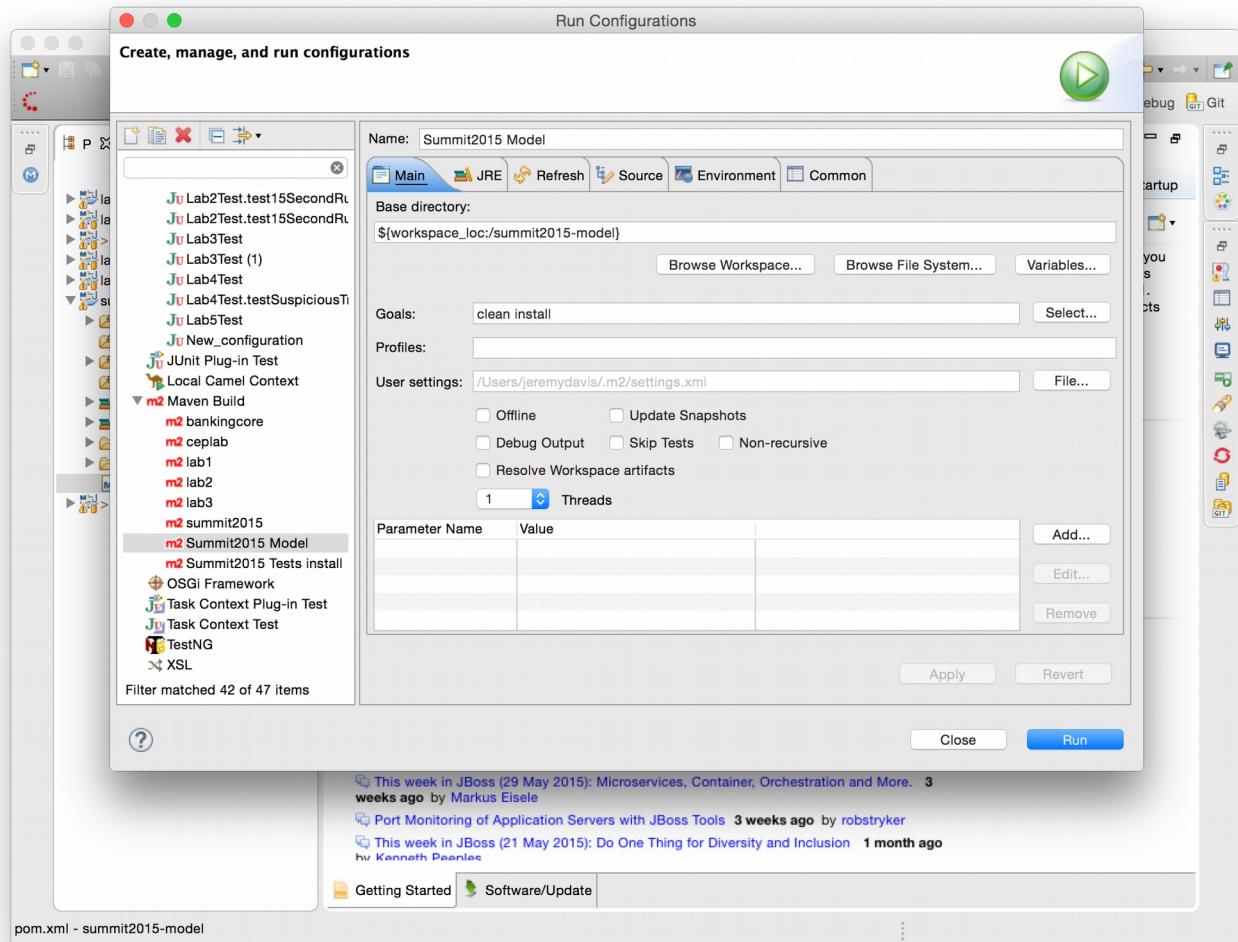
Introduces the basics of authoring and running rules

LAB 1

Summary

1. Open JBoss Developer Studio
2. Import summit2015-model and summit2015-tests from /home/student as "Existing Maven Projects"
3. Build and install both projects
4. Browse the domain classes and BaseCEPTestCase





DIRECTIONS

1. Open JBoss Developer Studio
2. Import summit2015-model and summit2015-tests from /home/student as "Existing Maven Projects"
 1. Choose File → Import → Existing Maven Projects
 2. Navigate to the "student2015" directory
 3. Select the folder, "summit2015-model"
 4. Click "Finish"
 5. Repeat for the "summit2015-tests" project
3. Build and install both projects
 1. You can build by choosing "Run Configurations" from the top menu or by right clicking on the pom.xml for the summit2015-model project
 2. Be sure to assign the targets, "Clean, Install," in the dialog
4. Browse the domain classes and the BaseCEPTestCase class

Domain Classes

NOTE : SCREENSHOTS OF THE DOMAIN MODEL ARE IN THE FOLLOWING PAGES.

The domain consists of 4 classes :

- Account
- AccountStatus
- Transaction
- TransactionStatus

This is obviously an extremely simplistic model; however, it will serve the examples. AccountStatus and TransactionStatus are Enums that we will manipulate as the rules change.

JBoss - summit2015-model/src/main/java/com/redhat/summit2015/ceplab/model/Account.java - JBoss Developer Studio - /Users/jeremydavis/Wor...

The screenshot shows the JBoss Developer Studio interface with the Account.java file open in the central editor window. The code defines a simple Account class with fields for id, status, and balance, and methods for constructor and toString(). The editor includes syntax highlighting, code completion, and various toolbars and panels typical of an IDE.

```
1 package com.redhat.summit2015.ceplab.model;
2
3 import java.math.BigDecimal;
4
5 public class Account {
6
7     private String id;
8
9     private AccountStatus status;
10
11    private BigDecimal balance = BigDecimal.ZERO;
12
13    public Account(AccountStatus status, BigDecimal balance) {
14        super();
15        this.status = status;
16        this.balance = balance;
17        this.id = String.valueOf(System.nanoTime());
18    }
19
20    public Account() {
21        super();
22    }
23
24    @Override
25    public String toString() {
26        return new ToStringBuilder(this).append("id", id)
27            .append("status", status)
28            .append("balance", balance).toString();
29    }
30
31    // -----
32    // generated methods
33    // -----
34
35 }
```

The screenshot shows a Java code editor window in JBoss Developer Studio. The file being edited is `AccountStatus.java`, located at `JBoss - summit2015-model/src/main/java/com/redhat/summit2015/ceplab/model/AccountStatus.java`. The code defines a public enum `AccountStatus` with four values: `PENDING`, `ACTIVE`, `TERMINATED`, and `BLOCKED`.

```
1 package com.redhat.summit2015.ceplab.model;
2
3 public enum AccountStatus {
4     PENDING, ACTIVE, TERMINATED, BLOCKED
5 }
6
7
8
```

JBoss - summit2015-model/src/main/java/com/redhat/summit2015/ceplab/model/Transaction.java - JBoss Developer Studio - /Users/jeremydavis/...

The screenshot shows the JBoss Developer Studio interface with the Transaction.java file open in the central editor window. The code defines a Transaction class with fields for id, status, fromAccount, toAccount, and amount, along with constructors for different parameter sets. The code editor includes standard Java syntax highlighting and a status bar at the bottom.

```
1 package com.redhat.summit2015.ceplab.model;
2
3 import java.math.BigDecimal;
4
5 public class Transaction {
6
7     private String id;
8
9     private TransactionStatus status;
10
11    private Account fromAccount;
12    private Account toAccount;
13
14    private BigDecimal amount;
15
16    public Transaction() {
17        super();
18    }
19
20    public Transaction(Account fromAccount, Account toAccount, BigDecimal amount) {
21        super();
22        this.fromAccount = fromAccount;
23        this.toAccount = toAccount;
24        this.amount = amount;
25        this.status = TransactionStatus.SCHEDULED;
26        this.id = String.valueOf(System.nanoTime());
27    }
28
29
30    public Transaction(Account fromAccount, BigDecimal amount){
31        super();
32        this.fromAccount = fromAccount;
33        this.amount = amount;
34        this.status = TransactionStatus.SCHEDULED;
35    }
}
```

The screenshot shows a Java code editor window in JBoss Developer Studio. The title bar indicates the file is `JBoss - summit2015-model/src/main/java/com/redhat/summit2015/ceplab/model/TransactionStatus.java`. The code editor displays the following Java enum definition:

```
1 package com.redhat.summit2015.ceplab.model;
2
3 public enum TransactionStatus {
4     SCHEDULED, ACTIVE, TERMINATED, COMPLETED, SUSPICIOUS, DUPLICATE, DENIED;
5 }
6
7
8
```

The interface includes standard Java editor features like code completion, syntax highlighting, and navigation tools. The status bar at the bottom shows "Writable", "Smart Insert", and "1 : 1".

BaseCEPTTestCase

NOTE : A SCREENSHOT OF BASECEPTTESTCASE IS IN THE FOLLOWING PAGE

BaseCEPTTestCase is a base class that handles the boilerplate required for interacting with the rules engine.

The important objects to note are :

KieSession

SessionPseudoClock

EntryPoint

Also of note is the **setUp()** method of the class.

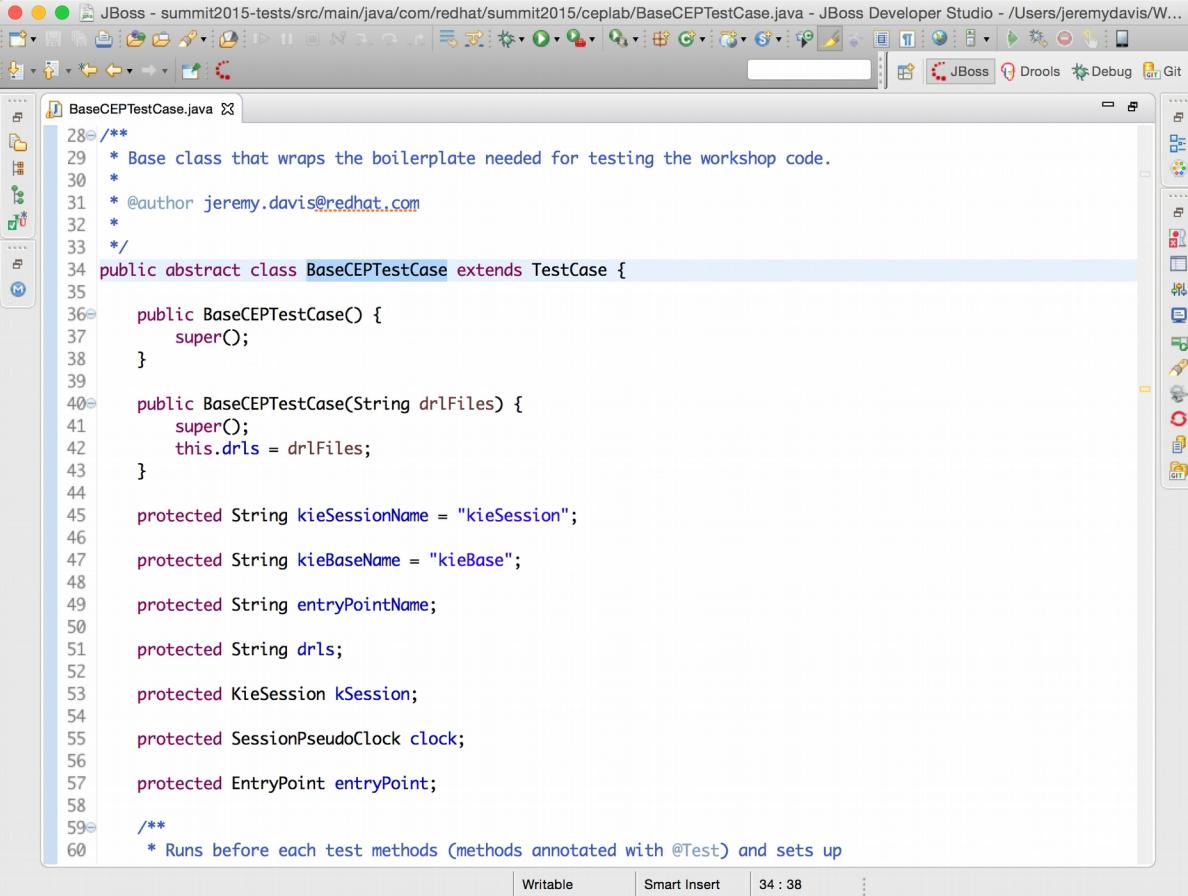
The **KieSession** is the most common way to programmatically interact with the rules engine. A KieSession allows for iterative conversations with the rules engine and keeps state across invocations. Events are inserted into the KieSession. The KieSession then executes the rules. KieSessions can be stateful or stateless. Complex Event Processing uses stateful KieSessions.

The **SessionPseudoClock** is a clock that is controlled programmatically. The SessionPseudoClock contains methods that allow our programs to manipulate the date and time of the KieSession's clock. It is used during testing and development so that we do not have to wait in real time for our rules to fire. It can also be used in production. The reasons are outside the scope of this lab, but feel free to ask the proctors of you would like to discuss CEP deployments in detail!

An **EntryPoint** represents a stream through which events enter the engine. Most CEP scenarios use streams. A running engine can contain multiple streams and corresponding EntryPoints.

The **setUp()** method performs the work of initializing the KieSession. The KieSession can discover relevant files from the classpath and is typically configured using an XML file, kmodule.xml, that is included in the META-INF folder of the jar. In this lab we programmatically create the KieSession, lines 81-84, and add the rules files individually.

BaseCEPTestCase

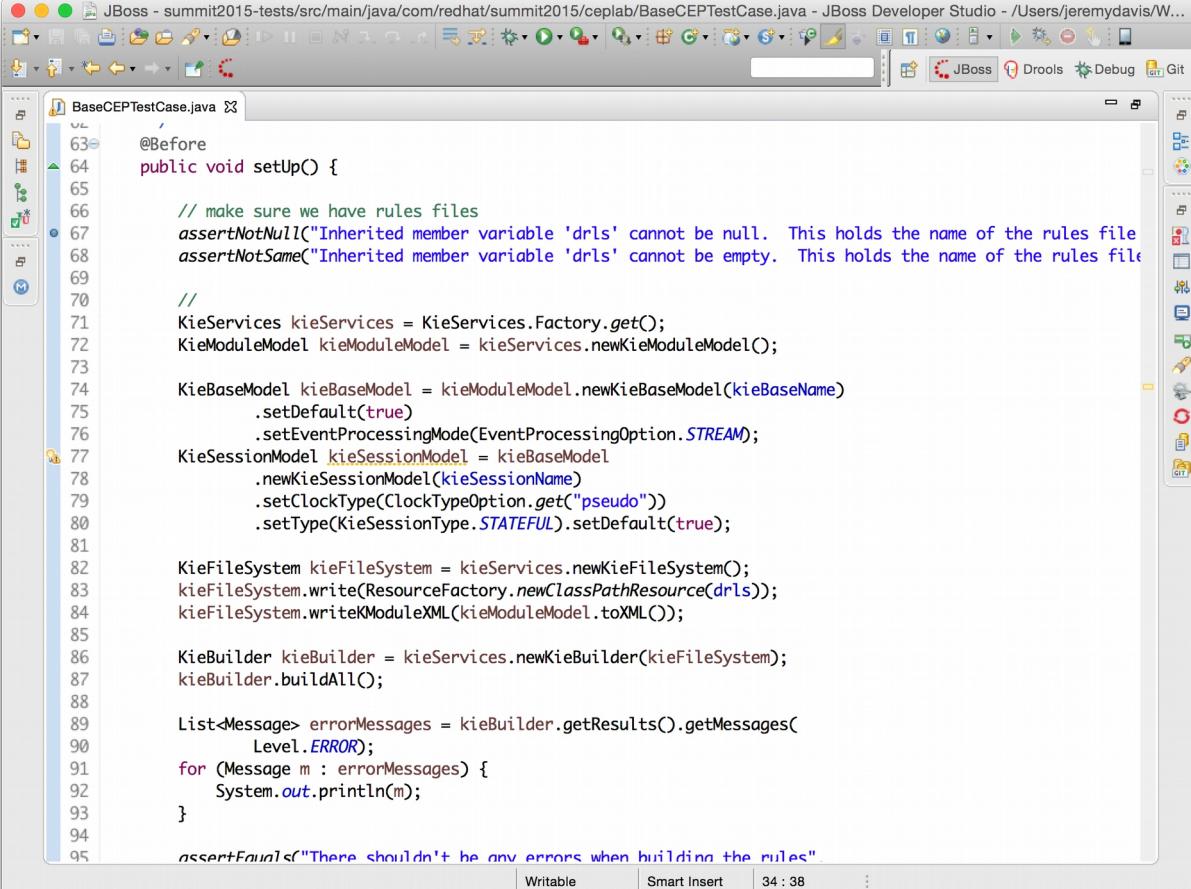


The screenshot shows the JBoss Developer Studio interface with the 'BaseCEPTestCase.java' file open in the central editor window. The code defines an abstract base class for CEP tests, extending TestCase. It includes fields for session names, base names, entry points, and drl files, along with methods for creating instances and setting up test environments.

```
28  /**
29  * Base class that wraps the boilerplate needed for testing the workshop code.
30  *
31  * @author jeremy.davis@redhat.com
32  *
33  */
34 public abstract class BaseCEPTestCase extends TestCase {
35
36     public BaseCEPTestCase() {
37         super();
38     }
39
40     public BaseCEPTestCase(String drlFiles) {
41         super();
42         this.drls = drlFiles;
43     }
44
45     protected String kieSessionName = "kieSession";
46
47     protected String kieBaseName = "kieBase";
48
49     protected String entryPointName;
50
51     protected String drls;
52
53     protected KieSession kSession;
54
55     protected SessionPseudoClock clock;
56
57     protected EntryPoint entryPoint;
58
59     /**
60      * Runs before each test methods (methods annotated with @Test) and sets up

```

BaseCEPTestCase.setUp()



The screenshot shows the JBoss Developer Studio interface with the file `BaseCEPTestCase.java` open in the central editor window. The code defines a `@Before` annotated method `setUp()`. This method performs several steps to set up a KieSessionModel:

```
63  '
64  @Before
65  public void setUp() {
66
67      // make sure we have rules files
68      assertNotNull("Inherited member variable 'drls' cannot be null. This holds the name of the rules file",
69      assertEquals("Inherited member variable 'drls' cannot be empty. This holds the name of the rules file",
70
71      KieServices kieServices = KieServices.Factory.get();
72      KieModuleModel kieModuleModel = kieServices.newKieModuleModel();
73
74      KieBaseModel kieBaseModel = kieModuleModel.newKieBaseModel(kieBaseName)
75          .setDefault(true)
76          .setEventProcessingMode(EventProcessingOption.STREAM);
77      KieSessionModel kieSessionModel = kieBaseModel
78          .newKieSessionModel(kieSessionName)
79          .setClockType(ClockTypeOption.get("pseudo"))
80          .setType(KieSessionType.STATEFUL).setDefault(true);
81
82      KieFileSystem kieFileSystem = kieServices.newKieFileSystem();
83      kieFileSystem.write(ResourceFactory.newClassPathResource(drls));
84      kieFileSystem.writeModuleXML(kieModuleModel.toXML());
85
86      KieBuilder kieBuilder = kieServices.newKieBuilder(kieFileSystem);
87      kieBuilder.buildAll();
88
89      List<Message> errorMessages = kieBuilder.getResults().getMessages(
90          Level.ERROR);
91      for (Message m : errorMessages) {
92          System.out.println(m);
93      }
94
95      assertEquals("There shouldn't be any errors when building the rules".
```

The code uses annotations from the `org.junit` and `org.drools` packages. It also imports various classes from the `org.kie` and `org.kie.builder` packages.

LAB 1

Summary

1. Import the project lab1
2. Read through the TestCase and the rules file
3. Create a new test and a new rule

NOTE : SCREENSHOTS OF THESE CLASSES ARE IN THE FOLLOWING PAGES.

Import lab1

1. Choose File → Import → Existing Maven Projects
2. Navigate to the “lab1” directory
3. Select the folder, “lab1”
4. Click “Finish”

Lab1Test and lab1rules.drl

- src/main/resources/rules/lab1rules.drl
- src/main/resources/rules/lab1solution.drl.activity
- src/test/java/com/redhat/summit2015/ceplab/Lab1Test.java
- src/test/java/com/redhat/summit2015/ceplab/Lab1TestSolution.java

We won't be doing any Complex Event Processing in this lab. This lab is designed to show the basics of the rules engine. The rules file and unit test ending in “solution” contain solutions to the activity. Ignore those files for now.

Lab1Test

Important things to note :

2 Account objects are inserted into the KieSession, line 41

The KieSession.fireAllRules() method is called to perform analysis on the objects

This test class uses a KieClasspathContainer which loads all rules on the classpath into the KieSession. This approach is different from subsequent tests which will specify the rules files by name.

lab1rules.drl

The local variable \$account is assigned to each Fact in memory, line 13.

The logic that matches Facts to this rule are in the parens, line 13.

Activity

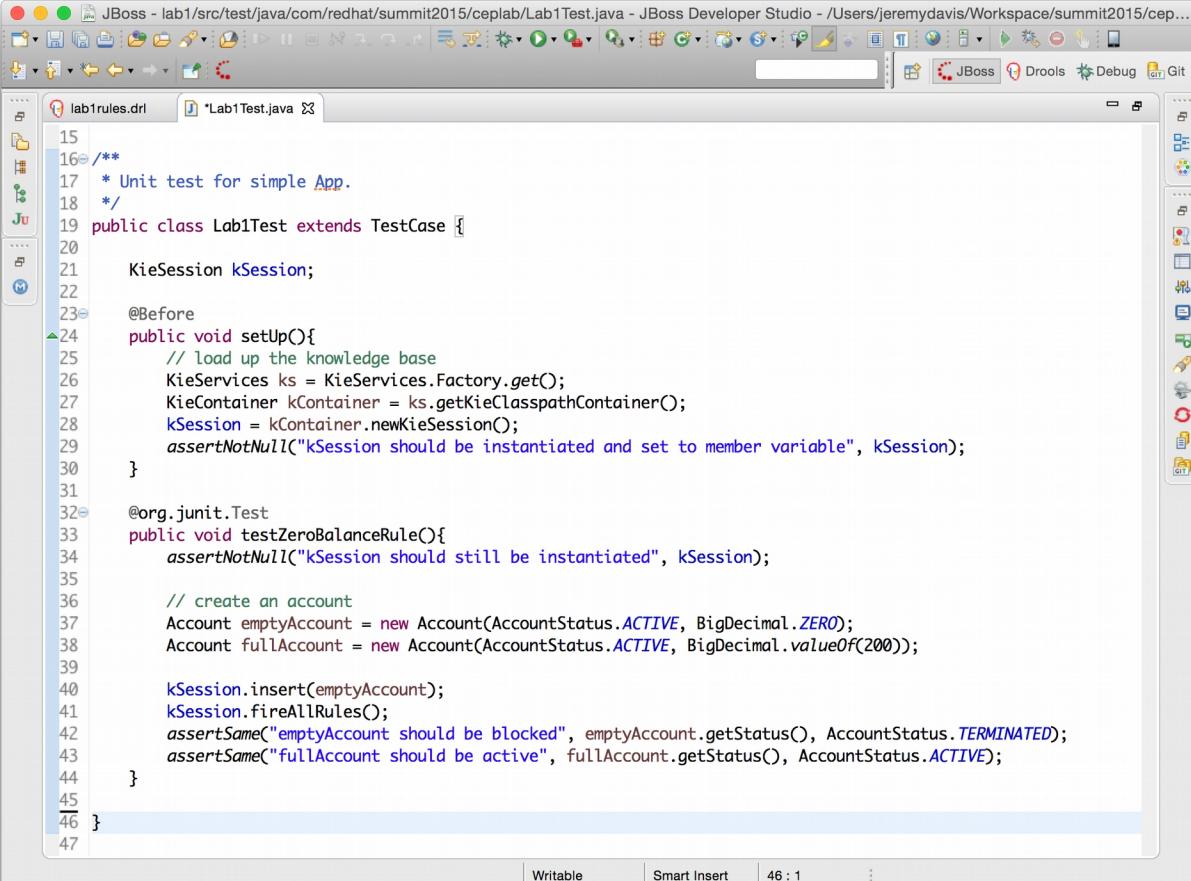
Create a test method that blocks accounts with a balance under \$100. Add a rule that passes this test.

HINT : Be careful not to conflict with the existing test that terminates accounts with a zero balance.

BEST PRACTICE : START YOUR RULE BY WRITING THE RULE IN NATURAL LANGUAGE IN A COMMENT

The TestCase, Lab1TestSolution, contains a possible solution. This test should fail initially. The drl file, "lab1solution.drl.activity" contains a solution for this activity. To use this rule rename it, removing "activity." In this example the rules engine is picking up all rules files on the classpath so the extension ".activity" prevents the solution from being included.

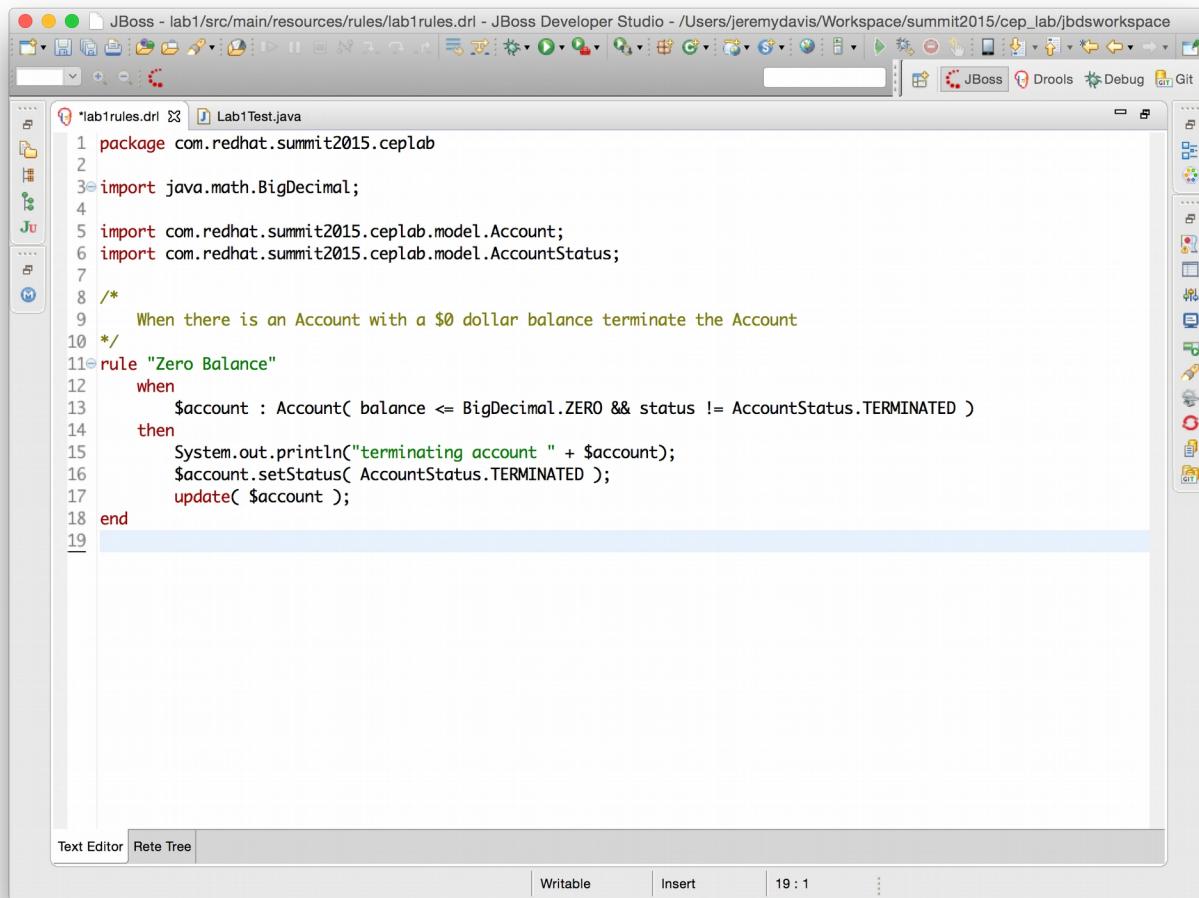
Lab1Test.java



The screenshot shows the JBoss Developer Studio interface with the 'Lab1Test.java' file open in the central editor area. The code implements a unit test for a simple application using JBoss Drools. It includes a setup method to load the knowledge base and a test method to verify account rules.

```
15 /**
16  * Unit test for simple App.
17 */
18
19 public class Lab1Test extends TestCase {
20
21     KieSession kSession;
22
23     @Before
24     public void setUp(){
25         // load up the knowledge base
26         KieServices ks = KieServices.Factory.get();
27         KieContainer kContainer = ks.getKieClasspathContainer();
28         kSession = kContainer.newKieSession();
29         assertNotNull("kSession should be instantiated and set to member variable", kSession);
30     }
31
32     @org.junit.Test
33     public void testZeroBalanceRule(){
34         assertNotNull("kSession should still be instantiated", kSession);
35
36         // create an account
37         Account emptyAccount = new Account(AccountStatus.ACTIVE, BigDecimal.ZERO);
38         Account fullAccount = new Account(AccountStatus.ACTIVE, BigDecimal.valueOf(200));
39
40         kSession.insert(emptyAccount);
41         kSession.fireAllRules();
42         assertEquals("emptyAccount should be blocked", emptyAccount.getStatus(), AccountStatus.TERMINATED);
43         assertEquals("fullAccount should be active", fullAccount.getStatus(), AccountStatus.ACTIVE);
44     }
45
46 }
47
```

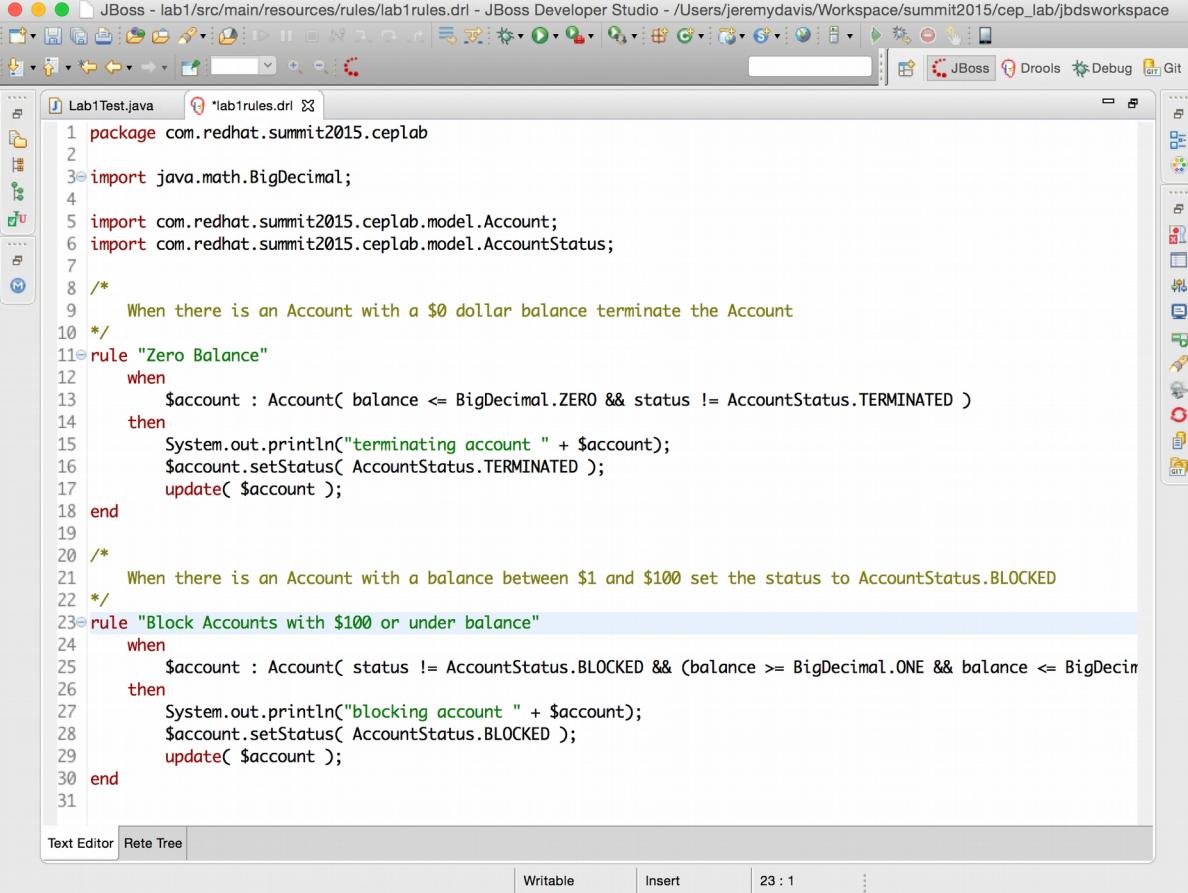
lab1rules.drl



The screenshot shows the JBoss Developer Studio interface with the file `lab1rules.drl` open in the central editor area. The code defines a rule named "Zero Balance" that triggers when an account has a balance less than or equal to zero and is not terminated. It prints a message to the console, sets the account's status to terminated, and updates the account.

```
1 package com.redhat.summit2015.ceplab
2
3 import java.math.BigDecimal;
4
5 import com.redhat.summit2015.ceplab.model.Account;
6 import com.redhat.summit2015.ceplab.model.AccountStatus;
7
8 /*
9  * When there is an Account with a $0 dollar balance terminate the Account
10 */
11@rule "Zero Balance"
12    when
13        $account : Account( balance <= BigDecimal.ZERO && status != AccountStatus.TERMINATED )
14    then
15        System.out.println("terminating account " + $account);
16        $account.setStatus( AccountStatus.TERMINATED );
17        update( $account );
18    end
19
```

Activity Solution : lab1rules.drl

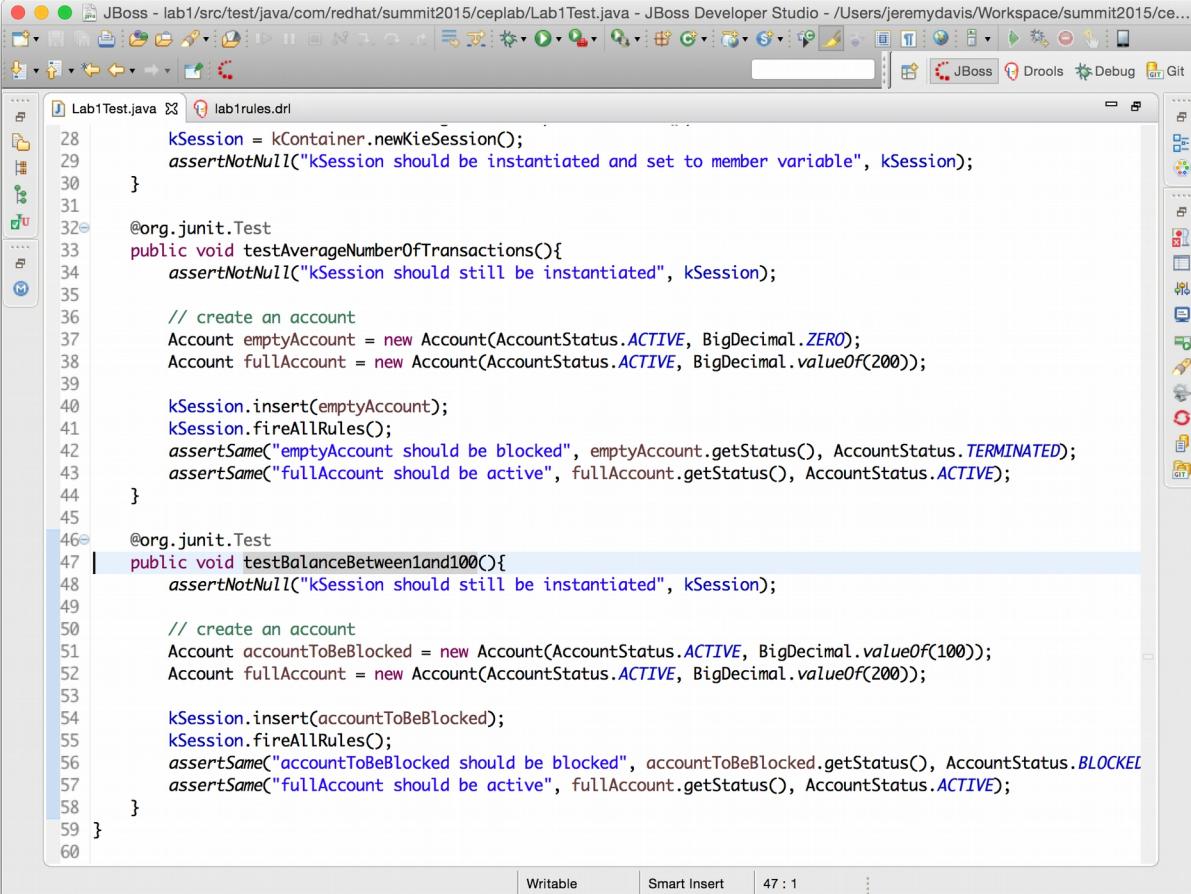


The screenshot shows the JBoss Developer Studio interface with the 'lab1rules.drl' file open in the central editor area. The code defines two Drools rules for managing account balances.

```
1 package com.redhat.summit2015.ceplab
2
3 import java.math.BigDecimal;
4
5 import com.redhat.summit2015.ceplab.model.Account;
6 import com.redhat.summit2015.ceplab.model.AccountStatus;
7
8 /*
9      When there is an Account with a $0 dollar balance terminate the Account
10 */
11 rule "Zero Balance"
12   when
13     $account : Account( balance <= BigDecimal.ZERO && status != AccountStatus.TERMINATED )
14   then
15     System.out.println("terminating account " + $account);
16     $account.setStatus( AccountStatus.TERMINATED );
17     update( $account );
18 end
19
20 /*
21      When there is an Account with a balance between $1 and $100 set the status to AccountStatus.BLOCKED
22 */
23 rule "Block Accounts with $100 or under balance"
24   when
25     $account : Account( status != AccountStatus.BLOCKED && (balance >= BigDecimal.ONE && balance <= BigDecir
26   then
27     System.out.println("blocking account " + $account);
28     $account.setStatus( AccountStatus.BLOCKED );
29     update( $account );
30 end
31
```

The code uses Java imports and Drools-specific syntax like rules, when clauses, and then clauses. It also includes comments explaining the logic for terminating accounts with zero balance and blocking accounts with a balance between \$1 and \$100.

Lab 1 Activity Solution : Lab1Test



The screenshot shows the JBoss Developer Studio interface with the file `Lab1Test.java` open in the editor. The code implements unit tests for a KIE session using JUnit. It includes two test methods: `testAverageNumberOfTransactions()` and `testBalanceBetween1and100()`. Both tests verify the status of accounts after rules are fired. The code uses AccountStatus enum values like `ACTIVE`, `ZERO`, `TERMINATED`, `BLOCKED`, and `valueOf(200)`.

```
28     kSession = kContainer.newKieSession();
29     assertNotNull("kSession should be instantiated and set to member variable", kSession);
30 }
31
32 @org.junit.Test
33 public void testAverageNumberOfTransactions(){
34     assertNotNull("kSession should still be instantiated", kSession);
35
36     // create an account
37     Account emptyAccount = new Account(AccountStatus.ACTIVE, BigDecimal.ZERO);
38     Account fullAccount = new Account(AccountStatus.ACTIVE, BigDecimal.valueOf(200));
39
40     kSession.insert(emptyAccount);
41     kSession.fireAllRules();
42     assertEquals("emptyAccount should be blocked", emptyAccount.getStatus(), AccountStatus.TERMINATED);
43     assertEquals("fullAccount should be active", fullAccount.getStatus(), AccountStatus.ACTIVE);
44 }
45
46 @org.junit.Test
47 public void testBalanceBetween1and100(){
48     assertNotNull("kSession should still be instantiated", kSession);
49
50     // create an account
51     Account accountToBeBlocked = new Account(AccountStatus.ACTIVE, BigDecimal.valueOf(100));
52     Account fullAccount = new Account(AccountStatus.ACTIVE, BigDecimal.valueOf(200));
53
54     kSession.insert(accountToBeBlocked);
55     kSession.fireAllRules();
56     assertEquals("accountToBeBlocked should be blocked", accountToBeBlocked.getStatus(), AccountStatus.BLOCKED);
57     assertEquals("fullAccount should be active", fullAccount.getStatus(), AccountStatus.ACTIVE);
58 }
59 }
60 }
```

LAB 2 : INTRODUCING TEMPORAL CORRELATION

Summary

1. Import the project lab2
2. Read through the TestCase and the rules file
3. Create a new test and a new rule

NOTE : SCREENSHOTS OF THESE CLASSES ARE IN THE FOLLOWING PAGES.

Import lab2

1. Choose File → Import → Existing Maven Projects
2. Navigate to the “lab2” directory
3. Select the folder, “lab2”
4. Click “Finish”

Lab2Test and lab2rules.drl

- src/main/resources/rules/lab2rules.drl
- src/main/resources/rules/lab2solution.drl
- src/test/java/com/redhat/summit2015/ceplab/Lab2Test.java
- src/test/java/com/redhat/summit2015/ceplab/Lab2TestSolution.java

In this lab we will begin Complex Event Processing.

Lab2Test

Important things to note :

We extend BaseCEPTTestCase

lab2rules.drl

The Transaction Fact is declared as an Event, line 8

```
7  
8 declare Transaction  
9     @role( event )  
10 end  
11
```

The Transaction Events are inserted from the “Transfers” stream. This prevents unrelated Transaction Events from being evaluated by this rule.

```
19     $t1 : Transaction( $fromAccount : fromAccount ) from entry-point Transfers
20     $t2 : Transaction( this != $t1, fromAccount == $fromAccount, this after [0s, 15s] $t1 )
21         from entry-point Transfers
```

The second Transaction Event must occur within 15 seconds of the first.

```
19     $t1 : Transaction( $fromAccount : fromAccount ) from entry-point Transfers
20     $t2 : Transaction( this != $t1, fromAccount == $fromAccount, this after [0s, 15s] $t1 )
21         from entry-point Transfers
```

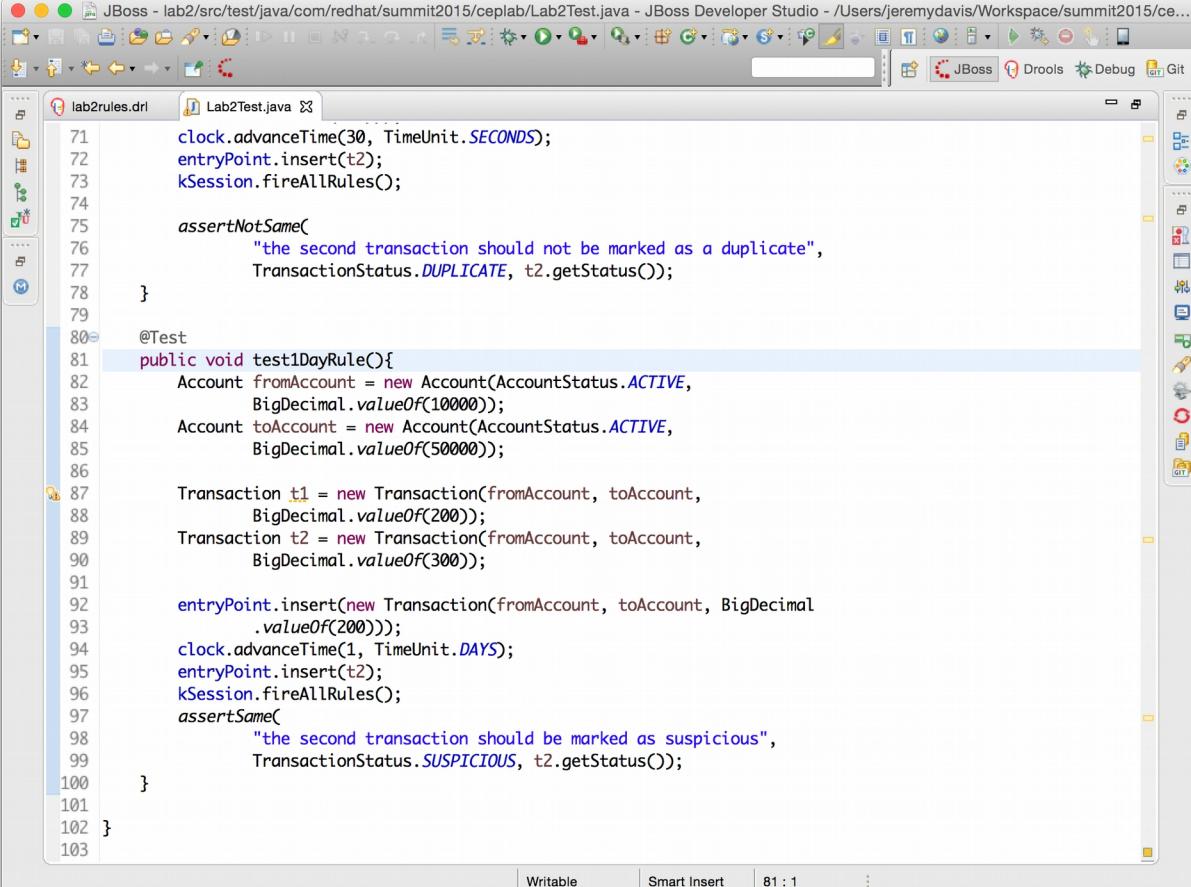
Activity

Create a test method and a rule that flags 2 transfers from the same account within 1 day as suspicious.

HINT : Be careful not to conflict with the existing test.

BEST PRACTICE : START YOUR RULE BY WRITING THE RULE IN NATURAL LANGUAGE IN A COMMENT

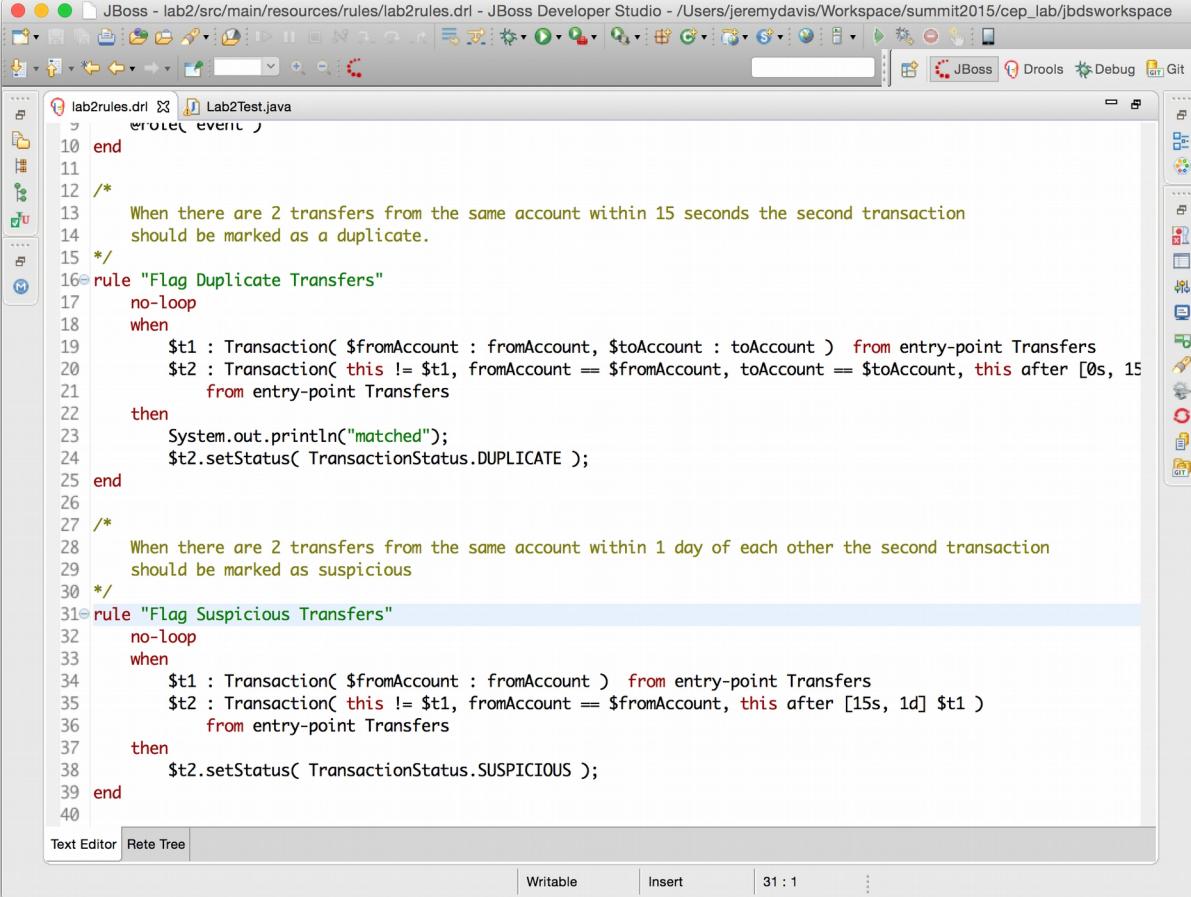
Lab 2 Activity Solution : Lab2Test.java



The screenshot shows the JBoss Developer Studio interface with the file `Lab2Test.java` open. The code implements unit tests for a banking system's transaction rules. It uses the Drools library for rule execution.

```
1  package com.redhat.summit2015.ceplab;
2
3  import org.junit.Test;
4  import org.kie.api.runtime.KieSession;
5  import org.kie.api.runtime.rule.EntryPoint;
6  import org.kie.api.runtime.rule.Transaction;
7  import org.kie.api.runtime.rule.TransactionStatus;
8  import org.kie.internal.time.Clock;
9
10 import java.math.BigDecimal;
11 import java.util.Date;
12
13 public class Lab2Test {
14
15     @Test
16     public void test1DayRule() {
17         Account fromAccount = new Account(AccountStatus.ACTIVE,
18             BigDecimal.valueOf(10000));
19         Account toAccount = new Account(AccountStatus.ACTIVE,
20             BigDecimal.valueOf(5000));
21
22         Transaction t1 = new Transaction(fromAccount, toAccount,
23             BigDecimal.valueOf(200));
24         Transaction t2 = new Transaction(fromAccount, toAccount,
25             BigDecimal.valueOf(300));
26
27         EntryPoint.insert(new Transaction(fromAccount, toAccount, BigDecimal
28             .valueOf(200)));
29         clock.advanceTime(1, TimeUnit.DAYS);
30         EntryPoint.insert(t2);
31         kSession.fireAllRules();
32         assertEquals(
33             "the second transaction should be marked as suspicious",
34             TransactionStatus.SUSPICIOUS, t2.getStatus());
35     }
36
37     @Test
38     public void test1DayRule2() {
39         Account fromAccount = new Account(AccountStatus.ACTIVE,
40             BigDecimal.valueOf(10000));
41         Account toAccount = new Account(AccountStatus.ACTIVE,
42             BigDecimal.valueOf(5000));
43
44         Transaction t1 = new Transaction(fromAccount, toAccount,
45             BigDecimal.valueOf(200));
46         Transaction t2 = new Transaction(fromAccount, toAccount,
47             BigDecimal.valueOf(300));
48
49         EntryPoint.insert(new Transaction(fromAccount, toAccount, BigDecimal
50             .valueOf(200)));
51         clock.advanceTime(30, TimeUnit.SECONDS);
52         EntryPoint.insert(t2);
53         kSession.fireAllRules();
54         assertEquals(
55             "the second transaction should not be marked as a duplicate",
56             TransactionStatus.DUPLICATE, t2.getStatus());
57     }
58 }
```

Lab 2 Activity Solution : lab2rules.drl



The screenshot shows the JBoss Developer Studio interface with the 'lab2rules.drl' file open in the central editor area. The code defines two rules: 'Flag Duplicate Transfers' and 'Flag Suspicious Transfers'. The 'Flag Duplicate Transfers' rule triggers when two transactions from the same account occur within 15 seconds. It prints 'matched' to the console and sets the status of the second transaction to DUPLICATE. The 'Flag Suspicious Transfers' rule triggers when two transactions from the same account occur within 1 day. It sets the status of the second transaction to SUSPICIOUS.

```
10 end
11
12 /*
13     When there are 2 transfers from the same account within 15 seconds the second transaction
14     should be marked as a duplicate.
15 */
16 rule "Flag Duplicate Transfers"
17     no-loop
18     when
19         $t1 : Transaction( $fromAccount : fromAccount, $toAccount : toAccount ) from entry-point Transfers
20         $t2 : Transaction( this != $t1, fromAccount == $fromAccount, toAccount == $toAccount, this after [0s, 15
21             from entry-point Transfers
22     then
23         System.out.println("matched");
24         $t2.setStatus( TransactionStatus.DUPLICATE );
25     end
26
27 /*
28     When there are 2 transfers from the same account within 1 day of each other the second transaction
29     should be marked as suspicious
30 */
31 rule "Flag Suspicious Transfers"
32     no-loop
33     when
34         $t1 : Transaction( $fromAccount : fromAccount ) from entry-point Transfers
35         $t2 : Transaction( this != $t1, fromAccount == $fromAccount, this after [15s, 1d] $t1 )
36             from entry-point Transfers
37     then
38         $t2.setStatus( TransactionStatus.SUSPICIOUS );
39     end
40
```

LAB 3 : DECLARING FACTS AND @PROPERTYREACTIVE

Summary

1. Import the project lab3
2. Read through the TestCase and the rules file
3. Create a new test and a new rule

NOTE : SCREENSHOTS OF THESE CLASSES ARE IN THE FOLLOWING PAGES.

Import lab3

1. Choose File → Import → Existing Maven Projects
2. Navigate to the “lab3” directory
3. Select the folder, “lab3”
4. Click “Finish”

Lab3Test and lab3rules.drl

- src/main/resources/rules/lab3rules.drl
- src/main/resources/rules/lab3solution.drl
- src/test/java/com/redhat/summit2015/ceplab/Lab3Test.java
- src/test/java/com/redhat/summit2015/ceplab/Lab3TestSolution.java

In this lab we will see some new rule techniques.

Lab3

Important things to note :

Declaring Facts and Events within a drl file

We have a Fact (Java POJO) , AccountInfo, declared inside the rule file. This is useful when there are objects that are not part of the domain but are necessary for executing the rules. The AccountInfo stores average monthly information about an Account. This information is necessary for executing the rules, but is not an important part of the domain. By declaring the AccountInfo Fact in the rules file it can exist in the KieSession longer than individual Transaction Events.

```
12
13 declare AccountInfo
14     @propertyReactive
15     id : String
16     averageBalance : BigDecimal
17     averageAmount : BigDecimal
18     averageNumberOfTransactions : BigDecimal
19     numberOfTransactions1Day : Long
20 end
21
```

FactTypes

A FactType allows us to create an instance of the AccountInfo. In Lab3Test.java we will create a local instance of AccountInfo in order to set values that would otherwise have to be calculated over time.

```
40 FactType accountInfoFactType = kSession.getKieBase().getFactType("com.redhat.summit2015.ceplab", "Accour
41 Object accountInfo = accountInfoFactType.newInstance();
42 accountInfoFactType.set(accountInfo, "averageBalance",
43     BigDecimal.valueOf(1000));
44 accountInfoFactType.set(accountInfo, "id", account.getId());
45 FactHandle accountInfoHandle = kSession.insert(accountInfoFactType);
46 kSession.update(accountInfoHandle, accountInfo);
```

Calling newInstance() on the FactType creates a local instance that we can insert into the KieSession's working memory.

```
40 FactType accountInfoFactType = kSession.getKieBase().getFactType("com.redhat.summit2015.ceplab", "Accour
41 Object accountInfo = accountInfoFactType.newInstance();
42 accountInfoFactType.set(accountInfo, "averageBalance",
43     BigDecimal.valueOf(1000));
44 accountInfoFactType.set(accountInfo, "id", account.getId());
45 FactHandle accountInfoHandle = kSession.insert(accountInfoFactType);
46 kSession.update(accountInfoHandle, accountInfo);
```

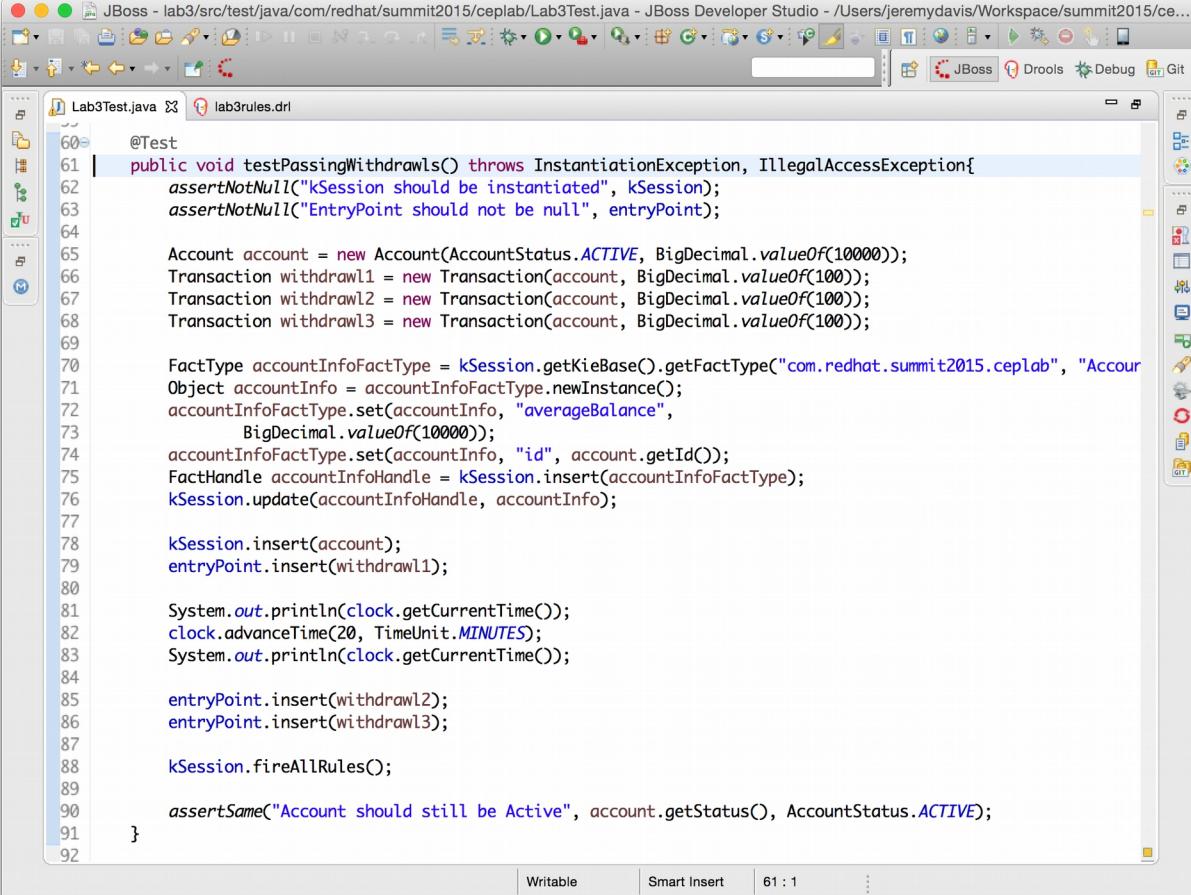
A FactHandle will allow us to keep track of the FactType. It is the return type when inserting a FactType into the KieSession.

```
40 FactType accountInfoFactType = kSession.getKieBase().getFactType("com.redhat.summit2015.ceplab", "Accour
41 Object accountInfo = accountInfoFactType.newInstance();
42 accountInfoFactType.set(accountInfo, "averageBalance",
43     BigDecimal.valueOf(1000));
44 accountInfoFactType.set(accountInfo, "id", account.getId());
45 FactHandle accountInfoHandle = kSession.insert(accountInfoFactType);
46 kSession.update(accountInfoHandle, accountInfo);
```

Activity

Using a FactType and a FactHandle create a test that verifies 3 valid transactions will pass the existing rule. One possible solution is on the next page.

Lab 3 Activity Solution



The screenshot shows the JBoss Developer Studio interface with the following details:

- Title Bar:** JBoss - lab3/src/test/java/com/redhat/summit2015/ceplab/Lab3Test.java - JBoss Developer Studio - /Users/jeremydavis/Workspace/summit2015/ce...
- Toolbar:** Standard Java development toolbar with icons for file operations, navigation, and debugging.
- Left Sidebar:** Project Explorer showing files Lab3Test.java and lab3rules.drl.
- Central Editor:** The code editor displays the `Lab3Test.java` file, which contains Java code for testing a KIE session. The code includes assertions for session instantiation, entry points, and account status, along with logic for inserting transactions and account info facts.
- Right Sidebar:** A panel containing icons for JBoss, Drools, Debug, and Git.
- Status Bar:** Shows "Writable", "Smart Insert", and "61 : 1".

```
60 @Test
61 public void testPassingWithdrawls() throws InstantiationException, IllegalAccessException{
62     assertNotNull("kSession should be instantiated", kSession);
63     assertNotNull("EntryPoint should not be null", entryPoint);
64
65     Account account = new Account(AccountStatus.ACTIVE, BigDecimal.valueOf(10000));
66     Transaction withdrawl1 = new Transaction(account, BigDecimal.valueOf(100));
67     Transaction withdrawl2 = new Transaction(account, BigDecimal.valueOf(100));
68     Transaction withdrawl3 = new Transaction(account, BigDecimal.valueOf(100));
69
70     FactType accountInfoFactType = kSession.getKieBase().getFactType("com.redhat.summit2015.ceplab", "Accour
71 Object accountInfo = accountInfoFactType.newInstance();
72 accountInfoFactType.set(accountInfo, "averageBalance",
73     BigDecimal.valueOf(10000));
74 accountInfoFactType.set(accountInfo, "id", account.getId());
75 FactHandle accountInfoHandle = kSession.insert(accountInfoFactType);
76 kSession.update(accountInfoHandle, accountInfo);
77
78 kSession.insert(account);
79 entryPoint.insert(withdrawl1);
80
81 System.out.println(clock.getCurrentTime());
82 clock.advanceTime(20, TimeUnit.MINUTES);
83 System.out.println(clock.getCurrentTime());
84
85 entryPoint.insert(withdrawl2);
86 entryPoint.insert(withdrawl3);
87
88 kSession.fireAllRules();
89
90 assertEquals("Account should still be Active", account.getStatus(), AccountStatus.ACTIVE);
91 }
92 }
```

LAB 4 : SLIDING WINDOWS

Summary

1. Import the project lab4
2. Read through the TestCase and the rules file
3. Create a new test and a new rule

NOTE : SCREENSHOTS OF THESE CLASSES ARE IN THE FOLLOWING PAGES.

Import lab4

1. Choose File → Import → Existing Maven Projects
2. Navigate to the “lab4” directory
3. Select the folder, “lab4”
4. Click “Finish”

Lab4Test and lab4rules.drl

- src/main/resources/rules/lab4rules.drl
- src/main/resources/rules/lab4solution.drl
- src/test/java/com/redhat/summit2015/ceplab/Lab4Test.java
- src/test/java/com/redhat/summit2015/ceplab/Lab4TestSolution.java

This lab introduces sliding windows. Sliding windows allow scoping Events of interest to a constantly moving group. There are two types of sliding windows: length and time. Lab 4 introduces length and Lab 5 contains a temporal example.

Lab4

lab4rules.drl note:

The four most recent transactions are being evaluated regardless of when they occurred.

```
21 when
22     $average: Number() from accumulate (
23         Transaction( $amount: amount ) over window:length ( 4 ) from entry-point CreditCard,
24         average($amount))
25     $transaction: Transaction( amount > ($average * 2) ) from entry-point CreditCard
26 then
27     $transaction.setStatus(TransactionStatus.DENIED);
```

The Drools method “from accumulate” is being called to collect all matching Events from the KieSession.

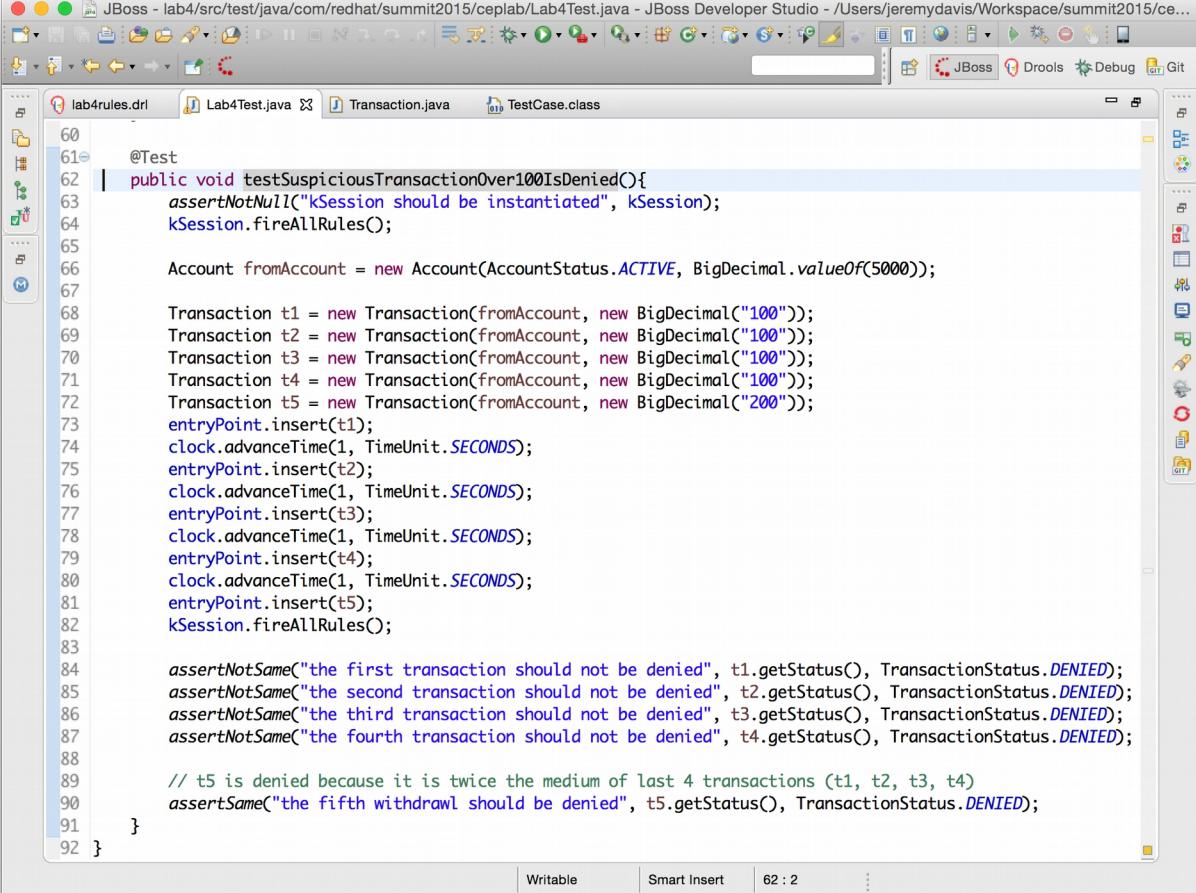
```
21 when
22     $average: Number() from accumulate (
23         Transaction( $amount: amount ) over window:length ( 4 ) from entry-point CreditCard,
24         average($amount))
25     $transaction: Transaction( amount > ($average * 2) ) from entry-point CreditCard
26 then
27     $transaction.setStatus(TransactionStatus.DENIED);
```

Activity

Create a test method that checks for a Transaction that is greater than the average of the previous 4 transactions when all Transactions are over \$100.

Create a rule that passes the test.

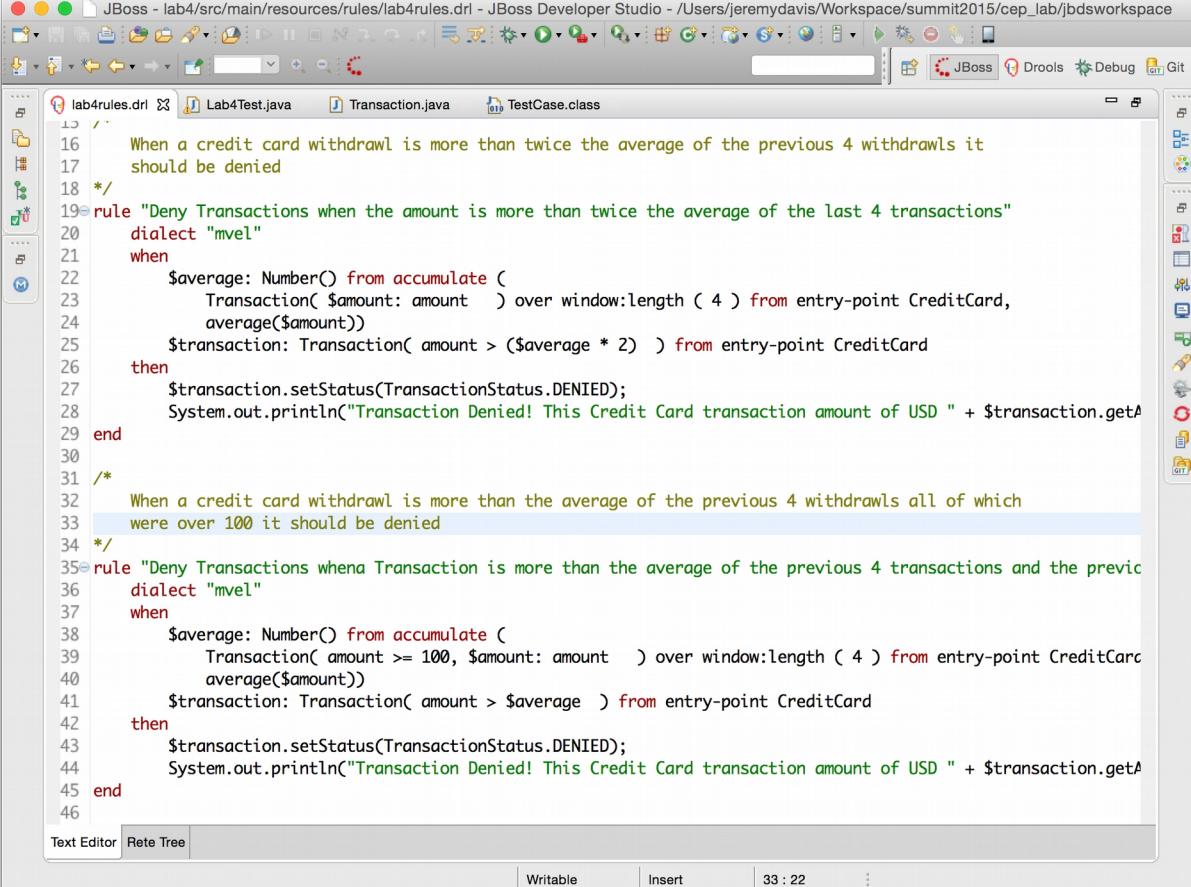
Lab 4 Activity Solution: Lab4Test



The screenshot shows the JBoss Developer Studio interface with the file `Lab4Test.java` open in the editor. The code implements a test for a banking system's transaction rules. It creates an account with \$5000, performs four transactions of \$100 each, and then a fifth transaction of \$200. The fifth transaction is denied because it exceeds the sum of the previous four. Assertions check that the first four transactions are not denied and that the fifth one is denied.

```
60
61 @Test
62 public void testSuspiciousTransactionOver100IsDenied(){
63     assertNotNull("kSession should be instantiated", kSession);
64     kSession.fireAllRules();
65
66     Account fromAccount = new Account(AccountStatus.ACTIVE, BigDecimal.valueOf(5000));
67
68     Transaction t1 = new Transaction(fromAccount, new BigDecimal("100"));
69     Transaction t2 = new Transaction(fromAccount, new BigDecimal("100"));
70     Transaction t3 = new Transaction(fromAccount, new BigDecimal("100"));
71     Transaction t4 = new Transaction(fromAccount, new BigDecimal("100"));
72     Transaction t5 = new Transaction(fromAccount, new BigDecimal("200"));
73     entryPoint.insert(t1);
74     clock.advanceTime(1, TimeUnit.SECONDS);
75     entryPoint.insert(t2);
76     clock.advanceTime(1, TimeUnit.SECONDS);
77     entryPoint.insert(t3);
78     clock.advanceTime(1, TimeUnit.SECONDS);
79     entryPoint.insert(t4);
80     clock.advanceTime(1, TimeUnit.SECONDS);
81     entryPoint.insert(t5);
82     kSession.fireAllRules();
83
84     assertNotSame("the first transaction should not be denied", t1.getStatus(), TransactionStatus.DENIED);
85     assertNotSame("the second transaction should not be denied", t2.getStatus(), TransactionStatus.DENIED);
86     assertNotSame("the third transaction should not be denied", t3.getStatus(), TransactionStatus.DENIED);
87     assertNotSame("the fourth transaction should not be denied", t4.getStatus(), TransactionStatus.DENIED);
88
89     // t5 is denied because it is twice the medium of last 4 transactions (t1, t2, t3, t4)
90     assertEquals("the fifth withdraw should be denied", t5.getStatus(), TransactionStatus.DENIED);
91 }
92 }
```

Lab 4 Activity Solution: lab4rules.drl



The screenshot shows the JBoss Developer Studio interface with the 'lab4rules.drl' file open in the central editor area. The code defines two rules for denying credit card transactions based on their amount relative to the average of the previous four transactions.

```
16 When a credit card withdrawl is more than twice the average of the previous 4 withdrawls it
17 should be denied
18 */
19 rule "Deny Transactions when the amount is more than twice the average of the last 4 transactions"
20 dialect "mvel"
21 when
22     $average: Number() from accumulate (
23         Transaction( $amount: amount ) over window:length ( 4 ) from entry-point CreditCard,
24         average($amount))
25     $transaction: Transaction( amount > ($average * 2) ) from entry-point CreditCard
26 then
27     $transaction.setStatus(TransactionStatus.DENIED);
28     System.out.println("Transaction Denied! This Credit Card transaction amount of USD " + $transaction.getAmount())
29 end
30
31 /*
32 When a credit card withdrawl is more than the average of the previous 4 withdrawls all of which
33 were over 100 it should be denied
34 */
35 rule "Deny Transactions whena Transaction is more than the average of the previous 4 transactions and the previc
36 dialect "mvel"
37 when
38     $average: Number() from accumulate (
39         Transaction( amount >= 100, $amount: amount ) over window:length ( 4 ) from entry-point CreditCard,
40         average($amount))
41     $transaction: Transaction( amount > $average ) from entry-point CreditCard
42 then
43     $transaction.setStatus(TransactionStatus.DENIED);
44     System.out.println("Transaction Denied! This Credit Card transaction amount of USD " + $transaction.getAmount())
45 end
46
```

The code uses the mvel dialect and defines two rules. The first rule denies a transaction if its amount is more than twice the average of the last four transactions. The second rule denies a transaction if its amount is more than the average of the last four transactions, provided that all four transactions were over 100. Both rules output a message to the console indicating the denial.

LAB 5

Summary

1. Import the project lab
2. Read through the TestCase and the rules file
3. Experiment!

NOTE : SCREENSHOTS OF THESE CLASSES ARE IN THE FOLLOWING PAGES.

Import lab5

1. Choose File → Import → Existing Maven Projects
2. Navigate to the “lab5” directory
3. Select the folder, “lab5”
4. Click “Finish”

Lab5Test and lab5rules.drl

- src/main/resources/rules/lab5rules.drl
- src/test/java/com/redhat/summit2015/ceplab/Lab5Test

This lab introduces temporal sliding windows.

lab5rules.drl

The sliding windows in this lab are temporal

```
20  $average: Number() from accumulate( 
21    Transaction( $amount: amount ) over window:time ( 30d ) from entry-point CreditCard,
22    average($amount))
23  $transaction: Transaction( amount > $average ) from entry-point CreditCard
```

Activity

Experiment! Create some tests, change rules around, and ask for help if you get stuck.

5. **[student@serverX ~]\$ sudo run-some-command**

Show any expected output

...

...

6. Instruction two

[student@serverX ~]\$ echo huzzah

huzzah

More detail on output or command that was run to help it make sense to people.

7. MOAR INSTRUCTIONS!!!