

Give your Business Logic a Framework with JBoss Rules

Author Biography

Based in Dublin, Ireland, Paul Browne has been consulting in Enterprise Java with FirstPartners.net for almost 7 years. When not promoting the [Red Piranha](#) (Search and Knowledge Management) and [NoUnit](#) (Java Development Process) projects , he can be found [blogging online](#). This article was originally published by [O'Reilly](#) books.

Most Web and Enterprise Java applications can be split into three parts ; a front end to talk to the user , a service layer to talk to back end systems such as databases , and business logic in between. While it is now common practice to use frameworks for both front and back end requirements (e.g. Struts, Cocoon , Spring, Hibernate, JDO and Entity Beans) , there is no standard way of structuring business logic. Frameworks like EJB and Spring do this at a high level , but don't help us in organizing our code. Wouldn't it would be great if would could replace messy , tangled if...then statements with a framework that gave us the same benefits of configurability , readability and reuse that we already enjoy in other areas? This article suggests using the JBoss Rules Engine (Drools) as a framework to solve the problem.

Whitepapers, Consulting and
Products at

 **firstpartners.net**

Enterprise Project Rescue

**Business Knowledge
Management**

Development Process

**Security, Performance and
Architecture**

Dynamic Web 2.0 Solutions

©Firstpartners.net 2005. Article may be
renroduced in full with this notice.

The sample code below gives a sample of the problem we're trying to avoid. It shows some business logic in a typical Java application.

```
if ((user.isMemberOf(AdministratorGroup)
    && user.isMemberOf(teleworkerGroup))
    || user.isSuperUser()) {
```

```
// more checks for specific cases
if ((expenseRequest.code().equals("B203")
    || (expenseRequest.code().equals("A903")
        &&(totalExpenses<200)
        &&(bossSignOff> totalExpenses))
    &&(deptBudget.notExceeded)) {
    //issue payments
} elseif {
    //check lots of other conditions
}
} else {
    // even more business logic
}
```

We've all come across similar or more complex business logic. While this has been the standard way of implementing business logic in Java , there are many problems with it.

- What if the business users come up with another form ('C987') that needs to be added to the already hard to understand code? Would you want to be the person to maintain it once all the original programmers had moved on?
- How do we check that these rules are correct? It's hard enough for technical never mind commercial people to review. Do we have any methodical way of testing this business logic?
- Many applications have similar business rules – if one of the rules change , can we be sure that it is changed consistently across all systems? If a new application uses some of these rules , but also adds some new ones , do we need to rewrite all the logic from scratch?
- Is the business logic easily configurable , not so firmly tied to Java code that we need to recompile / redeploy every time that a small change is made?
- What if other (scripting) languages want to leverage the existing investment in business rule logic?

J2EE / EJB and other 'inversion of control' frameworks (such as Spring, Pico and Avalon) give us the ability to organize our code at a high level. While they are very good at providing reusability , configuration , and security , none of them would replace the 'spaghetti' code in

the above example. Ideally whatever framework we choose will be compatible with not only J2EE applications ,but also 'normal' Java programs and most of the widely used presentation and persistence frameworks. Such a framework should allow us to do the following:

- Business users should be able to easily read and verify the business logic.
- Business rules should be reusable and configurable across applications.
- The framework should be scalable and performant under heavy load.
- It should be as easy to use for Java programmers as existing Front End (Struts , Spring) and Back End (Object Relational mapping) frameworks.

An additional problem is that while there are only so many ways to organize web pages and database access, business logic tends to differ widely between applications. Our framework should be able to cope with this and still promote code reuse. Ideally , our application would be 'frameworks all the way down'. By using frameworks in this way , we can a large amount of our application 'out of the box' , allowing us to write only the parts that add value for the customer.

Rule Engines to the rescue

So , how are we going to solve this problem? One solution that is gaining traction is to use a Rule-Engine. Rule Engines are frameworks for organizing our business logic that allow the developer to concentrate on things that are known to be true , rather than the low level mechanics of making decisions.

Often , Business users are more comfortable with expressing things that they know to be true, than to express things in an if...then format. Examples of things that you might hear from a business expert are:

- 'FORM 10A is used for expense claims over 200 Euro'.

- 'We only trade shares in quantities of 10,000 or more'.
- 'Purchases over Eur10m need the approval of a company director.'

By focusing on what we know to be true, rather than the mechanics of how to express it in Java code , the above statements are clearer than our previous code sample. As clear they may be, we still need a mechanism to apply these rules to the facts that we know and get a decision. Such a mechanism is a Rule Engine.

Whitepapers, Consulting and
Products at

 **firstpartners.net**

Enterprise Project Rescue

**Business Knowledge
Management**

Development Process

**Security, Performance and
Architecture**

Dynamic Web 2.0 Solutions

©Firstpartners.net 2005. Article may be
reproduced in full with this notice.

Rule Engines in Java

JSR 94 (the javax.rules api) sets a common standard for interacting with Rule Engines , much as JDBC allows us to interact with varying databases. What JSR-94 does not specify is how the actual rules are written , leaving plenty of choice between the most widely used java rule engines:

- Jess is perhaps the most mature Java rule engine with good tool support (including Eclipse plugins) and documentation. However it is a commercial product , and it

writes it's rules in a Prolog-style notation , which can be intimidating for many Java programmers.

- Jena is an open source framework , originally from HP. While it has a Rules engine , and is especially strong for those interested in the Semantic web , it is not fully JSR-94 compliant.
- Drools (The JBoss Rules Engine) is a JSR-94 complaint rules engine , and is fully open source under an 'Apache-Style' License. Not only does it express rules in a familiar Java and XML syntax , it has a strong user and developer community. For

the examples in this article , we'll be using Drools as it has the easiest to use Java -like syntax and it has the most open license.

Starting a Java application using JBoss Rules

Minutes after reading this article your boss asks your to prototype a stock trading application. As the business users still haven't fully defined the business logic , you think it a good idea to implement it using a rules engine. The final system will be accessible over an intranet and need to communicate with back end database and messaging systems. To get started , download the drools framework (with dependencies) from www.drools.org. Create a new project in your favorite IDE and make sure all the jars are referenced in it as per Figure 1. This screenshot is Eclipse Based , but the setup will be similar for other IDE's.

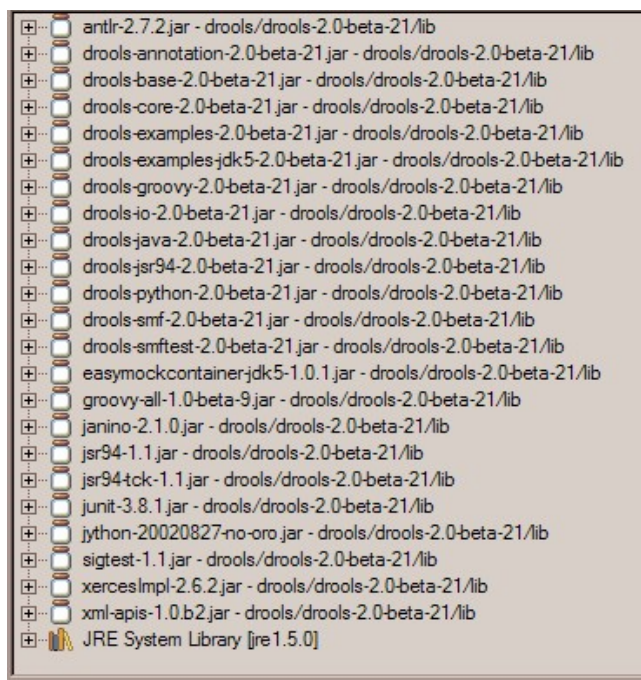


Figure 1. Libraries needed to Run Drools

Due to the huge potential losses if our stock-trading system went amok , it's vital that we have some sort of simulator to put our system through it's paces. Such a simulator also gives you confidence that the decisions made by the system are those that are intended , even after rule changes are made. We'll borrow some tools from the Agile toolbox and use Junit as a framework for our simulations.

The first code we write is the Junit Test / simulator as per listing 1. Even if we can't test every combination of values likely to be input into our application , some tests are better than none at all.

In this example all our files and classes (including unit tests) are in one folder / package, but in reality you would implement a proper package and folder structure. We'd also use Log4j instead of the `System.out` calls in the sample code.

Listing 1: BusinessRuleTest.java

```
import junit.framework.TestCase;
/*
 * JUnit test for the business rules in the
 * application.
 *
 * This also acts a 'simulator' for the business
 * rules - allowing us to specify the inputs ,
 * examine the outputs and see if they match our
 * expectations before letting the code loose in
 * the real world.
 */
public class BusinessRuleTest extends TestCase {
    /**
     * Tests the purchase of a stock
     */
    public void testStockBuy() throws Exception{

        //Create a Stock with simulated values
        StockOffer testOffer = new StockOffer();
        testOffer.setStockName("MEGACORP");
        testOffer.setStockPrice(22);
        testOffer.setStockQuantity(1000);

        //Run the rules on it
    }
}
```

```
BusinessLayer.evaluateStockPurchase(testOffer);

//Is it what we expected?
assertTrue(
    testOffer.getRecommendPurchase() != null);

assertTrue("YES".equals(
    testOffer.getRecommendPurchase()));
}
}
```

This is a basic Junit test , as we know that our (very simple!) system should buy all stocks with a price of less than 100 Euro. Obviously this won't compile without our Data holding class (StockOffer.java) and our Business Layer class (BusinessLayer.java). These are provided in listings 2 and 3.

Listing 2: BusinessLayer.java

```
/**
 * Facade for the Business Logic in our example.
 *
 * In this simple example , all our business logic
 * is contained in this class but in reality it
 * would delegate to other classes as required.
 */
public class BusinessLayer {
    /**
     * Evaluate whether or not it is a good idea
     * to purchase this stock.
     * @param stockToBuy
     * @return true if the recommendation is to buy
     * the stock, false if otherwise
     */
    public static void evaluateStockPurchase
        (StockOffer stockToBuy){
        return false;
    }
}
```

**Whitepapers, Consulting and
Products at**

 **firstpartners.net**

Enterprise Project Rescue

**Business Knowledge
Management**

Development Process

**Security, Performance and
Architecture**

Dynamic Web 2.0 Solutions

©Firstpartners.net 2005. Article may be
reproduced in full with this notice.

Listing 3: StockOffer.java

```
/**
 * Simple JavaBean to hold StockOffer values.
 * A 'Stock offer' is an offer (from somebody else)
 * to sell us a Stock (or Company share).
 */
public class StockOffer {

    //constants
    public final static String YES="YES";
    public final static String NO="NO";

    //Internal Variables
    private String stockName =null;
    private int stockPrice=0;
    private int stockQuantity=0;
    private String recommendPurchase = null;

    /**
     * @return Returns the stockName.
     */
    public String getStockName() {
        return stockName;
    }
    /**
     * @param stockName The stockName to set.
     */
    public void setStockName(String stockName) {
        this.stockName = stockName;
    }
    /**
     * @return Returns the stockPrice.
     */
    public int getStockPrice() {
        return stockPrice;
    }
    /**
     * @param stockPrice The stockPrice to set.
     */
    public void setStockPrice(int stockPrice) {
        this.stockPrice = stockPrice;
    }
    /**
     * @return Returns the stockQuantity.
     */
    public int getStockQuantity() {
        return stockQuantity;
    }
    /**
     * @param stockQuantity to set.
     */
}
```

```
public void setStockQuantity(int stockQuantity) {
    this.stockQuantity = stockQuantity;
}
/**
 * @return Returns the recommendPurchase.
 */
public String getRecommendPurchase() {
    return recommendPurchase;
}
}
```

We run `BusinessRuleTest` through the JUnit extension of our favorite IDE. If you're not familiar with JUnit, more information can be found at junit.org. Not surprisingly, our test fails at the 2nd assertion, as we don't (yet) have the appropriate business logic in place. This is reassuring, as it shows that our simulator / unit tests are highlighting the problems that they should.

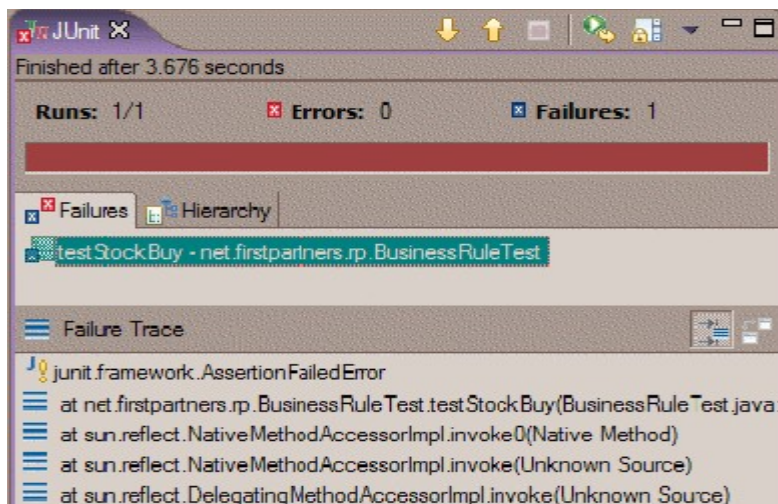


Figure 2. Junit Test Results

Writing the business logic using rules

At this point, we need to write some business logic which says 'if the stock price is less than 100 Euro, then we should buy it'. To do this we will modify `BusinessLayer.java` to read:

Listing 4: BusinessLayer.java (Updated)

```
package net.firstpartners.rp;
import java.io.IOException;
import org.drools.DroolsException;
import org.drools.RuleBase;
import org.drools.WorkingMemory;
import org.drools.event.DebugWorkingMemoryEventListener;
import org.drools.io.RuleBaseLoader;
import org.xml.sax.SAXException;
/**
 * Facade for the Business Logic in our example.
 *
 * In this simple example, all our business logic
 * is contained in this class but in reality it
 * would delegate to other classes as required.
 * @author default
 */
public class BusinessLayer {
    //Name of the file containing the rules
    private static final String BUSINESS_RULE_FILE=
        "BusinessRules.drl";

    //Internal handle to rule base
    private static RuleBase businessRules = null;
    /**
     * Load the business rules if we have not
     * already done so.
     * @throws Exception - normally we try to
     *     recover from these
     */
    private static void loadRules()
        throws Exception{
        if (businessRules==null){
            businessRules = RuleBaseLoader.loadFromUrl(
                BusinessLayer.class.getResource(
                    BUSINESS_RULE_FILE ) );
        }
    }

    /**
     * Evaluate whether or not to purchase stock.
     * @param stockToBuy
     * @return true if the recommendation is to buy
     * @throws Exception
     */
    public static void evaluateStockPurchase
        (StockOffer stockToBuy) throws Exception{

        //Ensure that the business rules are loaded
        loadRules();
        //Some logging of what is going on
    }
}
```

**Whitepapers, Consulting and
Products at**

 **firstpartners.net**

Enterprise Project Rescue

**Business Knowledge
Management**

Development Process

**Security, Performance and
Architecture**

Dynamic Web 2.0 Solutions

©Firstpartners.net 2005. Article may be
reproduced in full with this notice.

```
System.out.println( "FIRE RULES" );
System.out.println( "-----" );

//Clear any state from previous runs
WorkingMemory workingMemory
    = businessRules.newWorkingMemory();
//Small ruleset , OK to add a debug listener
workingMemory.addEventListener(
    new DebugWorkingMemoryEventListener());

//Let the rule engine know about the facts
workingMemory.assertObject( stockToBuy );

//Let the rule engine do it's stuff!!
workingMemory.fireAllRules();
}
}
```

This class now has some important methods:

- `loadRules()` which loads the rules from the `BusinessRules.drl` file.
- An updated `evaluateStockPurchase()` which evaluates these business rules. Some points to note about this method are:
 - We can reuse the same `RuleSet` over and over (as business rules in memory are stateless).
 - We use a new `WorkingMemory` for every evaluation, as this is our knowledge of what we know to be true at this time. We use `assertObject()` to place known facts (as Java Objects) into this memory.
 - Drools has an `EventListener` model , to allow us to 'see' what is going on within the event model. Here we use it to print debug information.
 - The `fireAllRules()` method on the working memory class causes the rules to be evaluated and updated (in this case stock offer).

Before we can run the example again , we need to create our `Business Rules (drl)` file , as per listing 5.

Listing 5 : BusinessRules.drl

```
<?xml version="1.0"?>
<rule-set name="BusinessRulesSample"
  xmlns="http://drools.org/rules"
  xmlns:java="http://drools.org/semantics/java"
  xmlns:xs
    ="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation
    ="http://drools.org/rules rules.xsd
  http://drools.org/semantics/java java.xsd">
  <!-- Import the Java Objects that we refer
        to in our rules -->

  <java:import>
    java.lang.Object
  </java:import>
  <java:import>
    java.lang.String
  </java:import>
  <java:import>
    net.firstpartners.rp.StockOffer
  </java:import>
  <!-- A Java (Utility) function we reference
        in our rules-->
  <java:functions>
    public void printStock(
      net.firstpartners.rp.StockOffer stock)
    {
      System.out.println("Name:"
        +stock.getStockName()
        +" Price: "+stock.getStockPrice()
        +" BUY:"
        +stock.getRecommendPurchase());
    }
  </java:functions>
  <rule-set>
    <!-- Ensure stock price is not too high-->
    <rule name="Stock Price Low Enough">
      <!-- Params to pass to business rule -->
      <parameter identifier="stockOffer">
        <class>StockOffer</class>
      </parameter>
      <!-- Conditions or 'Left Hand Side'
            (LHS) that must be met for
            business rule to fire -->
      <!-- note markup -->
      <java:condition>
        stockOffer.getRecommendPurchase() == null
      </java:condition>
      <java:condition>
        stockOffer.getStockPrice() < 100
      </java:condition>
    </rule>
  </rule-set>
</rule-set>
```

```
<!-- What happens when the business
      rule is activated -->
<java:consequence>
    stockOffer.setRecommendPurchase (
        StockOffer.YES);
    printStock(stockOffer);
</java:consequence>
</rule>
</rule-set>
```

This rules file has several interesting parts:

- Just after the XML-Schema definitions come the Java Objects we reference in our rules. These objects can come from any Java Library as required.
- Next comes our functions , which can incorporate standard Java code. In this case , we incorporate a logging function to help us see what is going on.
- After that comes our rule-set , consisting of one or more rules.
- Each rule can take parameters (the `StockOffer` class) , one or more conditions that need to be fulfilled and a consequence that is carried out if and when the conditions are met.

Having modified and compiled our code , we run the Junit test simulations again. This time the business rules are called, our logic evaluates correctly and our tests pass. Congratulations – you've just built your first rule based application!

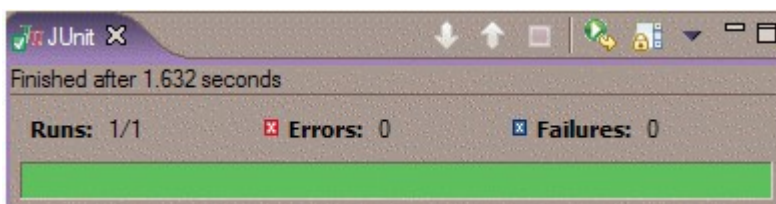


Figure 3. Successful Junit Test

Smarter Rules

Fresh from building the application , you demonstrate the prototype above to the business users and they remember a few more rules that they forgot to mention before. One of the new rules is that we shouldn't trade stocks where the Quantity is a negative number (<0). 'No problem' you say and return to your desk secure in the knowledge that you can quickly evolve your system.

The first thing you do is to update your simulator , and add the the code in listing 6 to `BusinessRuleTest.java`

Listing 6 : new method added to BusinessRuleTest.java

```
/**
 * Tests the purchase of a stock
 * makes sure the system will not accept
 * negative numbers.
 */
public void testNegativeStockBuy()
    throws Exception{

    //Create a Stock with our simulated values
    StockOffer testOffer = new StockOffer();
    testOffer.setStockName("MEGACORP");
    testOffer.setStockPrice(-22);
    testOffer.setStockQuantity(1000);

    //Run the rules on it
    BusinessLayer
        .evaluateStockPurchase(testOffer);

    //Is it what we expected?
    assertTrue("NO".equals(
        testOffer.getRecommendPurchase()));
}
```

This tests for the new rule described by the business users. If we run this Junit test , as expected , our new test fails. We we need to add a new rule to our drl file , as per listing 7 below.

Whitepapers, Consulting and
Products at

 **firstpartners.net**

Enterprise Project Rescue

**Business Knowledge
Management**

Development Process

**Security, Performance and
Architecture**

Dynamic Web 2.0 Solutions

©Firstpartners.net 2005. Article may be
reproduced in full with this notice.

Listing 7: new <rule> added to BusinessRules.drl

```
<!-- Ensure that negative prices
        are not accepted-->
<rule name="Stock Price Not Negative">
  <!-- Parameters we can pass into
        the business rule -->
  <parameter identifier="stockOffer">
    <class>StockOffer</class>
  </parameter>
  <!-- Conditions or 'Left Hand Side' (LHS)
        that must be met for rule to fire -->
  <java:condition>
    stockOffer.getStockPrice() < 0
  </java:condition>
  <!-- What happens when the business rule
        is activated -->
  <java:consequence>
    stockOffer.setRecommendPurchase (
                                StockOffer.NO);
    printStock(stockOffer);
  </java:consequence>
</rule>
```

This rule is similar in format to the previous one , expect that our <java:condition> is different (testing for negative numbers) and the <java:consequence> sets the recommend purchase to No. We run our Unit Tests / Simulator again , and this time the test passes.

At this point , if you're used to procedural programming (like most Java programmers) , you may be scratching your head: here we have a file containing 2 separate business rules , yet we haven't told the rule engine which is more important. However , our stock price (of -22) satisfies both rules (i.e. it is less than 0 and it is less than 100. Despite this , we get the correct result, even if we swap the order of the rules around. How does this work?

The extract of the console output below helps us to see what is going on. We see that both rules are firing (the [activationfired] line) , and that the 'Recommend Buy' is first set to Yes and then to No. How does Drools know to fire these rules in the correct order? If you look at the 'Stock Price Low Enough' rule you will see that one of the conditions is that the recommendPurchase() is null. This is enough for the Drools rule engine to decide that the 'Stock Price Low Enough' rule should be fired before the 'Stock Price Not Negative' rule. This process is called conflict resolution.

If you're a procedural programmer, not matter how clever you think this is , you still may not trust it completely. That is why we have our Unit tests / simulator : 'Hard' Junit tests (using normal Java code) ensure that the Rule Engine makes it's decisions along the lines we want it to (and not spend billions on worthless stock!). At the same time , the power and the flexibility of our rule engine allows us to quickly develop the business logic.

Later on we will see more sophisticated forms of conflict resolution.

Listing 8: extract of console output.

```
FIRE RULES
-----
[ConditionTested: rule=Stock Price Not Negative;
  condition=[Condition: stockOffer.getStockPrice()
  < 0]; passed=true; tuple={[]}]
[ActivationCreated: rule=Stock Price Not Negative;
  tuple={[]}]
[ObjectAsserted: handle=[fid:2];
  object=net.firstpartners rp.StockOffer@16546ef]
[ActivationFired: rule=Stock Price Low Enough;
  tuple={[]}]
[ActivationFired: rule=Stock Price Not Negative;
  tuple={[]}]
Name:MEGACORP Price: -22 BUY:YES
Name:MEGACORP Price: -22 BUY:NO
```

Conflict Resolution

Now the folks on the business side are really impressed and are starting to think through the

possible options. They've come across a problem with stocks of XYZ corp and have decided to implement a new rule : Only buy stocks of XYZ corp if they are less than 10 Euro.

As before , you add the test to our simulator , and include the new business rule in our rules file as per listings 9 and 10.

Listing 9 : new method added to `BusinessRuleTest.java`

```
/**
 * Makes sure the system will buy stocks
 * of XYZ corp only if it really cheap
 */
public void testXYZStockBuy() throws Exception{

    //Create a Stock with our simulated values
    StockOffer testOfferLow = new StockOffer();
    StockOffer testOfferHigh = new StockOffer();

    testOfferLow.setStockName("XYZ");
    testOfferLow.setStockPrice(9);
    testOfferLow.setStockQuantity(1000);

    testOfferHigh.setStockName("XYZ");
    testOfferHigh.setStockPrice(11);
    testOfferHigh.setStockQuantity(1000);

    //Run the rules on it and test
    BusinessLayer.evaluateStockPurchase(
        testOfferLow);
    assertTrue("YES".equals(
        testOfferLow.getRecommendPurchase()));

    BusinessLayer.evaluateStockPurchase(
        testOfferHigh);
    assertTrue("NO".equals(
        testOfferHigh.getRecommendPurchase()));
}
```

Listing 10: new <rule> added to `BusinessRules.drl`

```
<rule name="XYZCorp" salience="-1">
  <!-- Parameters we pass to rule -->
  <parameter identifier="stockOffer">
    <class>StockOffer</class>
  </parameter>
```

```

<java:condition>
  stockOffer.getStockName().equals("XYZ")
</java:condition>
<java:condition>
  stockOffer.getRecommendPurchase() == null
</java:condition>
<java:condition>
  stockOffer.getStockPrice() > 10
</java:condition>

<!-- What happens when the business
                                rule is activated -->
<java:consequence>
  stockOffer.setRecommendPurchase(
    StockOffer.NO);
  printStock(stockOffer);
</java:consequence>
</rule>

```

Note that in the Business Rules File , after the rule name , we set our Saliency =-1 (ie the lowest priority of all the rules we have specified so far). Most of the rules in our system 'conflict' i.e. Drools must make some decision on the order in which to fire rules , given that the conditions for all the rules will be met. The default way of deciding is :

**Whitepapers, Consulting and
Products at**

 **firstpartners.net**

Enterprise Project Rescue

**Business Knowledge
Management**

Development Process

**Security, Performance and
Architecture**

Dynamic Web 2.0 Solutions

©Firstpartners.net 2005. Article may be
reproduced in full with this notice.

- Saliency (a value we assign , as per the above listing)
- Recency (how many times we have used a rule)
- Complexity (specific rules with more complicated values fire first)
- LoadOrder (the order in which rules are loaded)

If we did not specify the saliency of our rule in this example , what would happen is :

- XYZ Corp rule would fire first (don't buy xyz if the price is more than 10 Euro) -Buy

=No.

- Then the more general rule fires (buy all stock under 100) fires , setting the recommend Buy="yes'

This would give a result that we don't want. However , since our example does set the saliency factor , the test and our business rules work as expected.

While most of the time writing clear rules , and setting the saliency will give enough information to Drools to decide which order to fire rules, sometimes we want to change entire manner in which Rule Conflicts are resolved. An example of how to change this is given below , where we tell the Rule Engine to fire the most simple rules first. A word of warning : be careful when changing conflict resolution , as it can fundamentally change the behavior of the rule engine – a lot of problems can be solved first with clear and well written rules.

Code Snippet 1: Changing the order of Conflict Resolution

```
//Generate our list of conflict resolvers
ConflictResolver[] conflictResolvers =
    new ConflictResolver[] {
        SaliencyConflictResolver.getInstance(),
        RecencyConflictResolver.getInstance(),
        SimplicityConflictResolver.getInstance(),
        LoadOrderConflictResolver.getInstance()
    };

//Wrap this up into one composite resolver
CompositeConflictResolver resolver =
    new CompositeConflictResolver(
        conflictResolvers);

//Specify this resolver when we load the rules
businessRules = RuleBaseLoader.loadFromUrl(
    BusinessLayer.class.getResource(
        BUSINESS_RULE_FILE), resolver);
```

For our simple application , driven by JUnit tests, we don't need to alter the way the Drools resolves rule conflicts. It is useful to know how conflict resolution works , especially when your application grows to meet more complex and demanding requirements.

Conclusion

This article demonstrated a problem that most programmers have had to face : how to put some order on the complexity of business logic. We demonstrated a simple application using Drools as a solution and introduced the notion of rule based programming, including how these rules are resolved at runtime. Later on , a follow up article [Using Drools in your Enterprise Java application](#) takes these foundations and shows how to use them in an Enterprise Java application.

Resources

- [Sample code for this article](#)
- [Drools Project home page](#)
- [Info on Drools rules](#)
- [Introduction to Drools and Rule Engines , by the Drools project lead.](#)
- [Drools Rules Schema files](#)
- [JSR-94 , Java Rule Engines , Overview](#)
- [Struts Framework web site](#)
- [Spring Framework web site](#)
- [Hibernate Website](#)
- [JUnit ,Test Framework](#)
- [Jess Java Rule Engine](#)
- [Jena Semantic and Rule Engine](#)
- [JSR-94 Homepage](#)
- [Jess in Action Homepage](#)
- [Business Rule Thinking \(Jess Based\)](#)
- [General introduction to Rule Systems](#)
- [Jess Implementation of the Rete Algorithm](#)