ILOG JRules

Technical White Paper

ILOG

Changing the rules of business™

ILOG JRules

Technical ILOG JRules White Paper

Contents

Preface 5
- Intended Document Audience 5
- Applicable ILOG JRules Versions 5

1 Introduction 6
- Defining Business Rules 6

2 ILOG Business Rule Application 7
- A Quick Tour 7
- Application Objects 7
- Working Memory and Context 8
- ILOG Rule Engine 9
- Writing Rules 10
- Rule Structure 10
- Rule Conditions 10
- Actions 11
- Execution of Example Application 12
- Tour Summary 17

3 ILOG JRules Overview 18
- Rule Engine 18
- Integration 18
- Scalability and Performance 19
- Rule Language 20
- Comprehensive Rule Language 20
- XML Rule Representation 20
- Rule Kit 20
- Rule Builder 21
- Rule Editors 21
- Business Rule Language 21
- Rule Repository 22
- Implementing ILOG JRules 23

4 ILOG JRules Rule Engine 24
- ILOG JRules API 24
- ILOG JRules Deployment 24
- Supports Common Architectures 24
- Seamless Java Integration 25
- XML Binding 25
- EJB Integration 26
- Performance 27
- Compile Rules Directly into Java 27
- Automatic and Developer Specified Optimizations 28

5 ILOG JRules Rule Language 31
- Java-like Syntax 31
- Rule Structure 31

XML and Exchange of Business Rules 32
Advanced Language Features 33
Collections 33
Relations 34
Temporal Reasoning 36

6 ILOG Business Rules Repository 39

7 ILOG JRules Development Tools 41
Rule Builder 41
Debugger 43
Profiler 44
DB Tool and Database Connectivity 45
Connecting to a Database 45
Architecture at Runtime 46
Integrity Management 47

8 ILOG JRules Business Rule Tools 48
Business Action Language 48
Business Object Model 50
Token Model 52
BRLDF (Business Rule Language Definition Framework) 52
Business Rule Editors 53
JavaBean Editor 53
Web Editor 53
Rule Templates 53
Rule Properties 54
Versioning 56
History 56
Query 56
Permission Management 57
Locking 58
Ruleset Extraction 58

Additional Resources 60

Preface

ILOG business rule technology is based on the philosophy of providing fast and flexible software components and libraries to aid in managing the complexity of today's computer applications. Competitive pressure, requirements changes, new product offerings, regulatory mandates, mergers and acquisitions all contribute to making applications obsolete shortly after being deployed. This requires costly code maintenance or redesign of the complete application. And object oriented design alone doesn't help. Why? Because often times the business rules — the logic that has to adapt — is buried in multiple objects and in multiple tiers. But it doesn't have to be that way. Business rule technology, in the form of tools that help architect and implement business rule applications, can make a difference. ILOG technology can help by enabling you to *Build Applications That Last*.

The rule engines in ILOG Rules (C++) and ILOG JRules (Java) have the ability to fire thousands of rules per second and accommodate rule sets of up to 50,000 rules. ILOG rule engines are the fastest, most compact and robust rule engines on the market. This white paper describes ILOG JRules in detail. We begin by briefly summarizing what business rules are and why they are important. A more in depth discussion of business rules can be found in a companion paper, *Business Rules – Powering Business and e-Business*. This is followed by a tour of what an ILOG rule engine is all about, how it operates within an application and how its addition can turn an ordinary application into one that uses business rules. The tour walks you through some of the key concepts associated with rule engines. Next is an overview of the ILOG JRules product, describing at a high level the features and functionality that make ILOG JRules unique in the landscape of rule engine vendors. These **first three sections provide a quick rundown of what ILOG JRules is all about**, providing concise reading for IT directors, managers, and project managers. Software developers will be more interested in the remaining sections, which describe in detail the rule engine, API, application deployment, rule language and tools that are provided in the ILOG JRules product.

After reading this paper, we hope you will become as excited about business rule technology as we are, and ready to *Build Applications That Last*!

Intended Document Audience

This document is directed towards Java developers and project managers. The aim is to define the ILOG JRules product in the context of business rules and the relationship ILOG JRules holds with the Java language.

Applicable ILOG JRules Versions

This document covers ILOG JRules 4.0.

- ILOG Business Rules Team

1 Introduction

Defining Business Rules

Business rules describe and control the structure, operation and strategy of an organization. Often they are captured in policy and procedure manuals, customer contracts, supplier agreements, marketing strategies, and as the expertise embodied in employees. They are dynamic and likely to change with time, found in all manner of applications — from the click-stream of a website to the numerous business rules codified as legalese in B2B trading partner contracts. Finance and insurance companies must enforce numerous internal policies and external regulatory policies. Telecommunications companies introduce new servicing and pricing schemes at a dizzying pace. E-business, self-service and personalization on the web demand real time order tracking, sales histories, and customer preferences. Each of these business domains relies heavily on being able to convey the policies, regulations, and strategies of their industry to IT departments for integration into software applications.

However, because of the dynamic nature of business rules, conventional application development fails to adequately support their unique demands. Business logic, encapsulated in application objects, database structures or stored procedures, is difficult and costly to maintain. When policy changes, application code must change to keep up. In today's fast paced business and e-business environment, competition has driven organizations to adapt to changing markets in ever shorter cycles. Often, current applications cannot adapt fast enough to keep pace and allow organizations to remain competitive. Business rule engines offer a solution.

Separating the business rules from application code and implementing them using a rule engine can make applications adaptable *and* maintainable. Since business rules are externalized from the application code, they can be changed independently without recompiling the application. When represented in a natural business language that business users understand, business users and analysts can actually write and maintain rules.

With ILOG JRules, developing new or adapting existing Java applications to utilize business rule technology is simple and elegant. ILOG JRules is the most flexible, lightweight and high performance rule engine available for the Java platform. This paper discusses the unique features of ILOG JRules, including its powerful rule engine, its support for business rules, and integrated development environment. A companion paper, "Business Rules, Powering Business and e-Business," presents in more detail the role of business rules within organizations and their relationship to current programming paradigms.

2 ILOG Business Rule Application

A Quick Tour

Embedding an ILOG Rules (C++) or ILOG JRules (Java) engine into an application is just like including any other Java classes. Because the rule engine is part of an extensive Java library that ILOG JRules provides for business rule application development, adding an ILOG JRules rule engine is the same whether it is to be used in a new or an existing application — requiring only two classes to implement.

In the following example, the process of integrating an ILOG rule engine into a simple application will be illustrated. This quick tour is included to introduce key concepts in ILOG rule engines.

Application Objects

Consider a simple pet store shopping cart application where a customer has the ability to fill their shopping cart with various pets and related items. A set of objects for this type of application might be: a ShoppingCart, Customer, Item, and ItemType. These can be represented in an object model as shown in Figure 1.

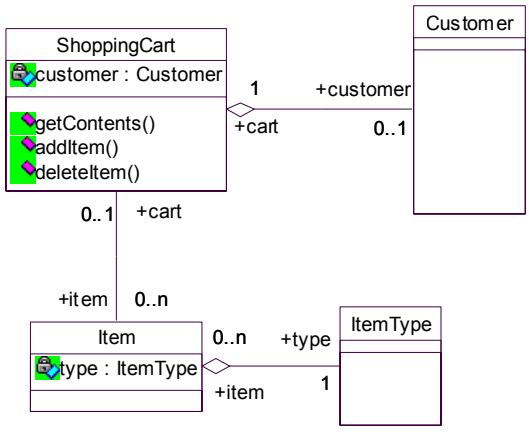
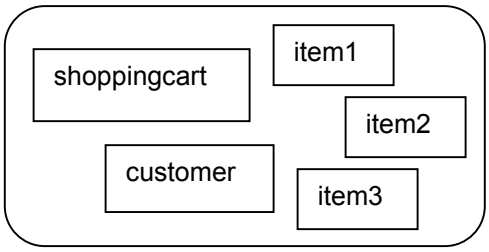


Figure 1 Object Model for simple shopping cart application.

Based upon the quantity and contents of the cart, various business rules will be applied to make decisions about customers and their purchases, informing them of special discounts or make suggestions for future purchases. But before introducing the rule engine or rules to the application, the concepts of *working memory* and *context* need to be discussed.

Working Memory and Context

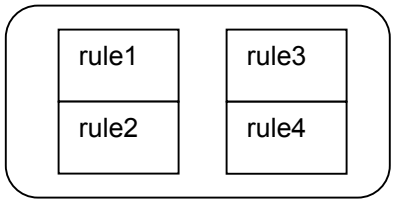
When an ILOG Rule Engine is integrated into an application, part of its function is to monitor the state of various application objects. This is necessary, as rules will typically reference application objects in the condition part of their logic (*e.g.*, if the shopping cart contains 10 or more fish, suggest a new aquarium to the customer). *Working memory* is where ILOG JRules stores references to all of the objects with which it is currently working. For example, working memory for the shopping cart application might contain references to instances of a shopping cart, a customer, and several items.



Working Memory

It is important to realize that application objects are not duplicated within working memory. The ILOG JRules rule engine infers directly from application objects, so no proxy layer is required to connect rules to objects.

In ILOG JRules, rules are grouped into sets. For the shopping cart example, assume the rule set contains four rules related to a promotional campaign for tropical fish.



Rule Set

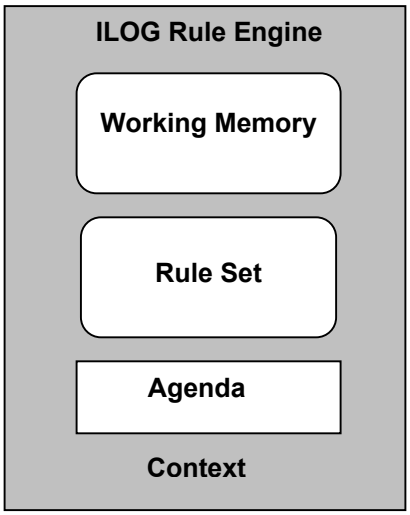
Rules in a rule set and the application objects that they reference are associated by what is called a *context*. That is, a context associates rules with working memory and implements the rule engine that controls the relationship between the two. The next section discusses how this context is created and rules are added to it.

ILOG Rule Engine

An ILOG Rule Engine is integrated into an application by simply having the application instantiate an `IlrContext` object, which creates a rule engine. Formally, this instantiation creates a context:

```
// Create an ILOG rule engine
IlrContext myContext = new IlrContext();
```

The context associates a rule set with working memory and implements a rule engine that controls the relationship between the two.



That's all there is to it. The `IlrContext` constructor creates a rule set container (yet to be filled with rules), working memory, a rule engine, and a context to control it all.

The final step in setting up the rule engine is adding rules to the rule set container. Rules in the form of XML, a text file, stream or URL, can be added or removed from the engine — on the fly — by using any of the available parsing methods on the rule set class:

```
// get the IlrRuleset associated with myContext
IlrRuleset myRuleset = myContext.getRuleset();


// Add rules to myRuleset
// (fish-promo-rules is a text file
// containing the business rules)
myRuleset.parseFileName("fish-promo-rules");
```

In this case, the rules in a text file called fish-promo-rules are read into an `IlrRuleset` container called `myRuleset`, and associated with `myContext`. The rule engine is now ready to examine working memory, matching objects referenced in working memory against patterns found in rule conditions and placing rule instances on the *agenda*. The agenda, shown in the above diagram of the context, is a container that stores rule instances that are ready to be executed.

At this point, the shopping cart application has become a full business rule application, ready to take on customers. However, before stepping through a typical scenario, let's examine how business rules are written.

Writing Rules

The basic structure of a rule and several key concepts needed for demonstrating the shopping cart example will be examined. These can be broken down into the following basic units: rule structure, patterns (conditions), and actions.

Rule Structure

The basic structure of a rule is IF, THEN. Typically, these are referred to as the Left Hand Side (LHS) and Right Hand Side (RHS) of a rule. The LHS contains conditions in the form of patterns and the RHS contains actions — things the rule should do if all the conditions on the LHS have been met.

Rule Conditions

The LHS of a rule is composed of a set of conditions, or *patterns*, that refer to Java objects. Each pattern is matched, if possible, with one or more objects in working memory. More precisely, a pattern comprises tests that are applied to each object in working memory, and an object is said to match the pattern when it passes these tests successfully. Here is a typical pattern that might be useful in the shopping cart example:

```
the item of type fish is in stock
```

This pattern would match Item objects in working memory that are in stock and of type fish. Notice that this pattern references Item objects, followed by specifications for attributes (type fish and in stock). Another pattern could be:

```
the shopping cart value is greater than 100
```

a pattern for ShoppingCart objects that contain more than $100 of merchandise.

Actions

The RHS of a rule is said to execute, or *fire* when all of the conditions on the LHS have been met. There are many actions that a rule might perform. Depending upon the requirements of the application, a rule may add an object to or remove an object from working memory, modify an object, or execute a method on one of the objects. In the example below, a 15% discount is being applied to a customer's order:

```
apply a 15% discount
```

Such an action might execute a method on the shopping cart object, modifying it by reducing the total purchase price by 15%.

Let's construct a set of four complete rules. The rules are written in a natural language syntax, available in ILOG JRules and ILOG Rules, and are readable enough to be self-explanatory.

1) Rule: **GoldCategory**

```
IF
  the purchase value is greater than the customer
previous purchase amount
  and the purchase value is greater than or equal to
$100

THEN
  change the customer category to Gold
  and display the message 'You're now a Gold
customer!"
```

2) Rule: **GoldDiscount**

```
IF
  the shopping cart contains between 2 and 4 items
  and the purchase value is greater than $100
  and the customer category is Gold

THEN
  apply a 15% discount
  and display the message 'We're giving you a Gold
discount!'
```

3) Rule: **SuggestFish**

```
IF
  the customer has previously bought fish
```

```
  and the shopping cart contains fish

THEN
  suggest items related to fish
```

4) Rule: **JiffyFishFood**

```
IF
  the item is fish

THEN
  add free food sample to shopping cart
  and display the message 'A free sample of Jiffy
fish food!'
```

Application objects must be asserted into, retracted from, or updated in working memory before pattern matching in rules can begin. This can be done either by using keywords within rules or, using the API, in Java code from within application objects themselves.

Summarizing the tour at this point, an ILOG rule engine, context, working memory and rule set have been created, and the rule set container loaded with fish promo rules:

```
// Create an ILOG rule engine
IlrContext myContext = new IlrContext();

// get the IlrRuleset associated with myContext
IlrRuleset myRuleset = myContext.getRuleset();

// Add rules to myRuleset
myRuleset.parseFileName("fish-promo-rules");
```

The application is now ready to be run. Once a shopping session begins and objects are inserted into working memory, the rule engine will match objects in working memory against the LHS of rules, placing onto the agenda rules that have been successfully matched. When a rule or application object modifies working memory, additional rules may be put on or retracted from the agenda.

Execution of Example Application

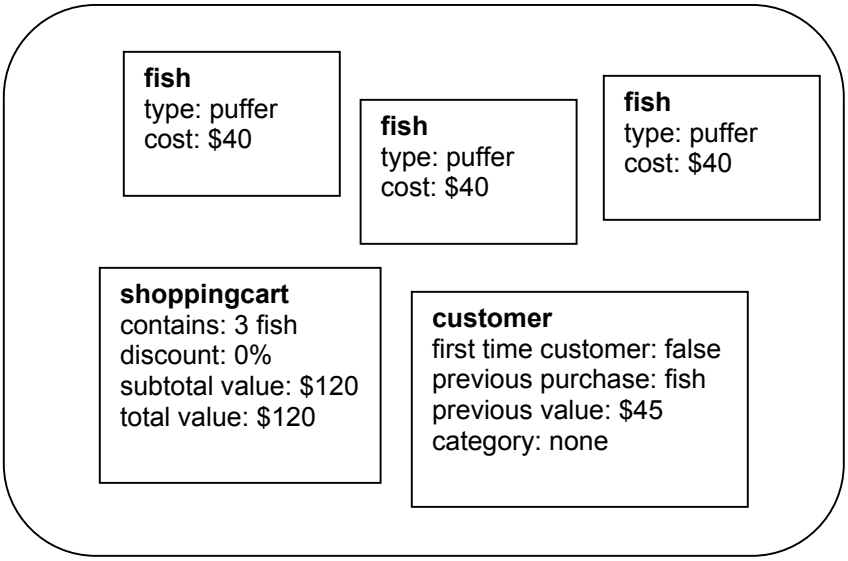
With the concepts defined and the rules in place, all that needs to be done before stepping through the shopping cart example is to create a customer and load their shopping cart. Let's give the customer and their shopping cart the following characteristics:

- Customer: previous shopper, has previously purchased fish
- Shopping cart: currently containing 3 Puffer fish at a cost of $40 each

A shopping scenario could be the following: the customer has navigated through the pet store making purchases, and is now ready to check out. At checkout, the customer, shopping cart and item objects are inserted into working memory by the application:

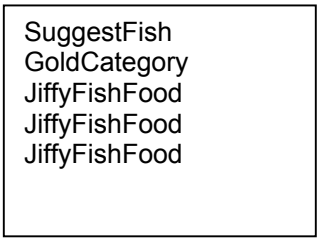
```
mycontext.assert(shoppingcart);
mycontext.assert(customer);
mycontext.assert(item);
```

A snapshot of working memory at this point is shown below.



Working Memory

Once the objects are in working memory, the rule engine will determine which rules are eligible for execution, and place an instance of those rules on the agenda. Comparing the contents of working memory with patterns found on the LHS of each of the four rules, it can be seen that the following rules will be added to the agenda, where the order is determined by the engine, and may be



Agenda

different each time the application is run (it is desirable to design a business rule application so that the order in which the rules execute is not important). A moment of study will show the agenda to be correct. It should be noted that three instances of the `JiffyFishFood` rule will be placed on the agenda since there are three instances of fish items in working memory.

To understand how the rule engine evaluates and executes rules, let's step through the firing of each rule, one at a time, and examine the contents of working memory and the agenda. Rules are fired once the `fireAllRules` method is called on the `IlrContext` object,

```
// execute the rules on the agenda
myContext.fireAllRules();


SuggestFish rule is executed.
Rule:  SuggestFish

IF
  the customer has previously bought fish
  and the shopping cart contains fish

THEN
  suggest items related to fish
```

Depending upon the design, this rule might execute an application method that pops up a banner on the customer's web page, suggesting other fish or fish related products in which the customer might be interested. As a result of firing this rule, there will be no change to working memory.

It is important to note that once this rule fires, it will be removed from the agenda and not fire again (even though the rule conditions are still true, i.e. the customer has still previously bought fish and the current shopping cart contains fish). This maintenance is inherent in the rule engine. The next rule on the agenda is `GoldCategory`.

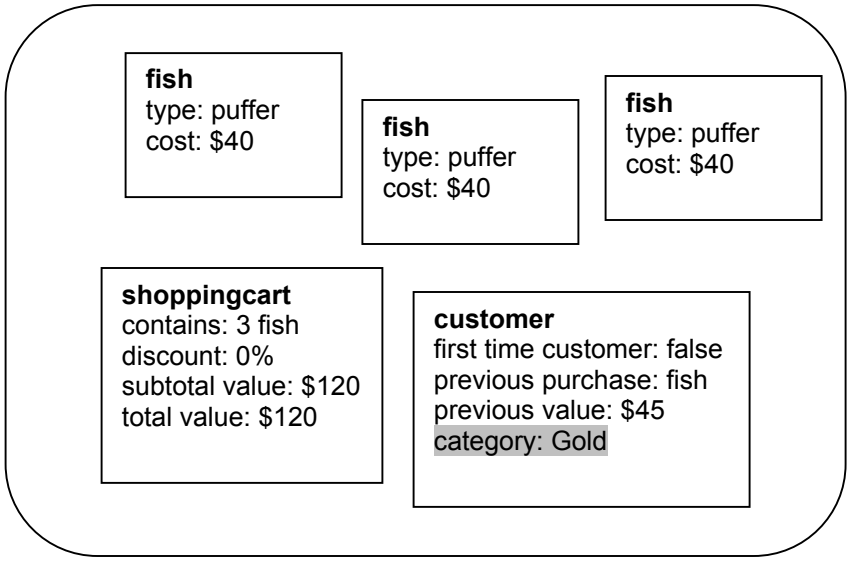
GoldCategory rule is executed.

Rule: **GoldCategory**

```
IF
  the purchase value is greater than the customer
previous purchase amount
  and the purchase value is greater than or equal to
$100

THEN
  change the customer category to Gold
  and display the message 'You're now a Gold
customer!"
```

Executing the GoldCategory rule will update the customer object in working memory, changing the category to Gold. Once fired, the GoldCategory rule is removed from the agenda.



Working Memory

Changing the customer's category to Gold will result in activating the GoldDiscount rule, which is directly added to the agenda. The agenda will now appear as:

Agenda

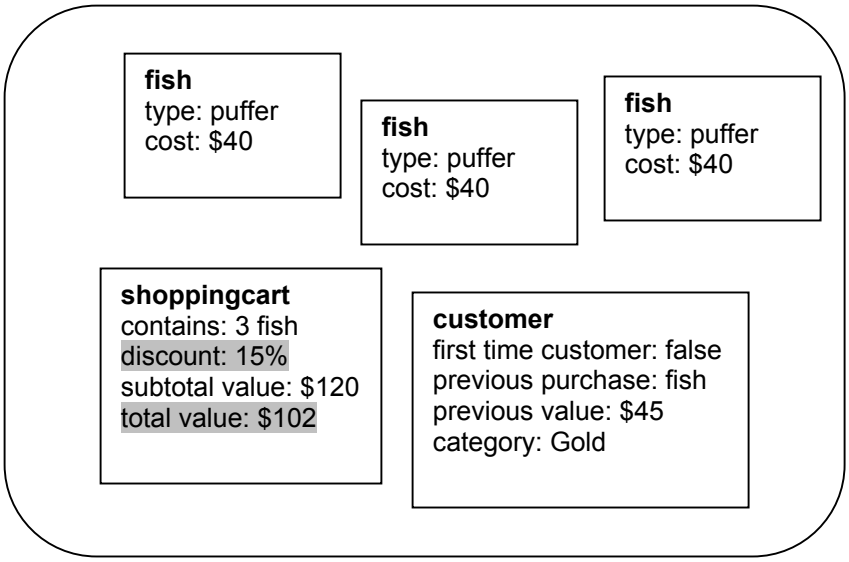
The GoldDiscount rule fires next, there will be a further update to working memory as a 15% discount is given to the customer, reducing the total value of the cart to $102.

GoldDiscount rule is executed.

Rule: **GoldDiscount**

```
IF
  the shopping cart contains between 2 and 4 items
  and the purchase value is greater than $100
  and the customer category is Gold

THEN
  apply a 15% discount
  and display the message 'We're giving you a Gold
discount!'
```



Working Memory

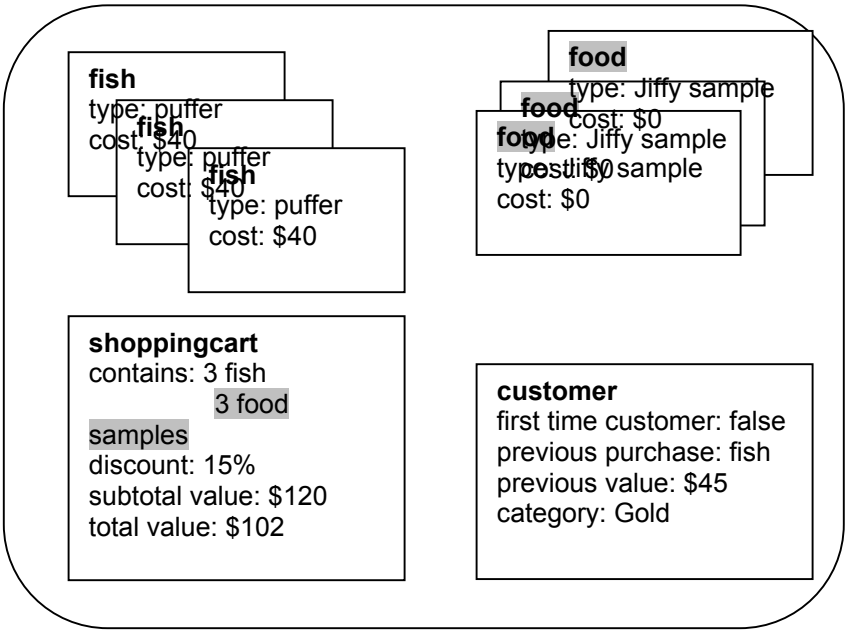
After the GoldDiscount rule has executed and is removed from the agenda, all that remains are the three JiffyFishFood rules. When the final three rules execute, working memory is again modified as new item objects are introduced and free fish food samples are added to the shopping cart.

JiffyFishFood rule is executed.

Rule: **JiffyFishFood**

```
IF
  the item is fish

THEN
  add free food sample to shopping cart
        and display the message 'A free sample of
  Jiffy fish food!'
```



Working Memory

At this point, the agenda is be empty (i.e., no other rules are eligible to execute), and the customer could either continue shopping, or proceed to billing and shipping, feeling satisfied that they not only received a discount, but free fish food as well.

Tour Summary

This tour demonstrates how easy it is to integrate an ILOG rule engine into an application, and how a simple application might behave in using business rules. It represents the simplest use of a rule engine: few rules, single threaded, one context.

The rest of this paper will build upon and detail the richness of ILOG JRules, and the simplicity with which that power may be used.

3 ILOG JRules Overview

ILOG JRules is a rich and flexible product aimed at enabling software developers to create applications that can be maintained with minimal effort. This section presents an overview of the distinguishing features that set ILOG JRules apart as the technological leader in business rule engines.
ILOG JRules leverages all of the unique strengths of Java, such as platform independence, multithreading, interfaces and introspection. It allows developers to combine rule-based and object-oriented programming to add business rules to new *and* existing applications.

ILOG JRules includes a high performance Java **rule engine**, a full featured **rule language** with support for business rules, and the **Rule Kit** — a comprehensive set of tools supporting the development of business rule applications.

Rule Engine

ILOG JRules is packaged as a set of Java class libraries. As a Java class, the rule engine can be implemented directly, or derived from to add application specific data members and methods. The rule engine is thread safe and optimized for Java, and provides a rich and flexible API that gives developers a high degree of control over the engine. It easily integrates into any Java application, including EJBs in the J2EE architecture, and has been successfully deployed into applications using CORBA, COM, IBM MQSeries, RMI, and Java servlets. Rules can be compiled directly to Java, or optimized automatically by the rule engine to yield the best performance for any business application.

Integration

Extensive Java API

ILOG JRules features the most extensive API available for business rule engines, supporting flexible integration options, and allowing the developer a detailed degree of control over the runtime behavior of the rule engine. ILOG JRules uniquely separates rules and the engine at the object level. This separation allows a rule engine to be connected to multiple rule sets, or a rule set to be used by any number of rule engines. ILOG JRules supports quick and maintainable development of the simplest to the most complex design strategies — from a single rule engine embedded in a Java application to multiple rule engines monitoring a common set of Java or XML application objects. Rules can be dynamically added on the fly, modified, or removed from the engine — without shutting down or recompiling the application. Finally, rules can be manipulated as objects and therefore constructed directly from application code.

The ILOG JRules API supports EJB and J2EE integration, further expanding the role of rule engines in enterprise business applications. Using ILOG JRules, engineers can design with confidence in one of the leading architectures available for distributed computing.

EJB and J2EE Ready

ILOG JRules is the only rule engine to provide full support for EJB integration. Since the ILOG JRules rule engine is thread-safe and serializable, it can be embedded directly into Enterprise JavaBeans, both session and entity beans. In addition, rules can directly reference EJBs. This level of integration gives developers tremendous power and flexibility in writing J2EE applications.

ILOG JRules includes a scalable session bean rule engine server that transparently manages a shared pool of rule engines containing pre-parsed rule sets. This saves the time required for rule set parsing and rule engine creation, and leads to increased efficiency in applications with demanding performance requirements. Since the rule engine server is a session bean, developers have full benefit of all the middleware services provided by the application server — load balancing, persistence, and fail over.

Scalability and Performance

Highly Optimized Rule Engine

ILOG JRules optimizes the Rete algorithm for the Java language, providing unsurpassed execution speed for large numbers of rules. The Rete algorithm is widely used because of its ability to handle large numbers of rules within an application and its unsurpassed performance in handling rules that reference dynamically changing data.

Multi-threaded

Multiple instances of the rule engine can be executed in parallel — because it is thread safe — making ILOG JRules scalable as well as suitable for high-speed parallel processing architectures. Using Java's synchronization capability, ILOG JRules is *thread-hot*, allowing multithreaded applications to share objects and rules, and multiple rule engines to operate on the same set of application objects.

Compiled Rules

ILOG JRules is the only rule engine supporting compiled rules for applications that require high performance. Rules can be translated directly into Java classes and integrated into the application to improve performance of the rules by a factor ranging from 4 to more than 10.

Automatic Rule Optimization

Not all applications have the same characteristics. Some applications have large numbers of rules applied against very few application objects, others have few rules applied against many application objects that change dynamically. ILOG JRules can be tuned to the unique demands of an applications performance requirement. Rules are automatically optimized to maximize performance and make better use of application resources. Further tuning can be performed through the use of a configuration file that puts the power of optimization specific parameters in the hands of developers.

Rule Language

ILOG has a comprehensive rule language that integrates seamlessly with Java and can be used with any existing Java or C++ application, simply by adding an ILOG rule engine — ILOG JRules or ILOG Rules. Rules may be represented in the Java-like syntax of the ILOG Rule Language, in XML, or in a readable syntax called a business rule language. So everyone from developers to end business users may write and manage rules, placing the power of rules in the hands of those who need it most.

Comprehensive Rule Language

The sophisticated ILOG Rule Language can be used by either the ILOG ILOG JRules or ILOG Rules engine. Rules can reference any application object and invoke methods on the applications objects without the overhead of a proxy layer.

Developers have at their disposal full support of Java operators in expressions and tests, Java-like syntax for interfaces, Java arrays, and variable scope management. Powerful constructs in the language support rules that reference objects that might have multiple instances in working memory, enhancing performance by allowing a single rule to reference the instances as a collected entity. There is support for relations between objects so that a rule may reference not only objects in working memory, but objects that are linked to those in working memory by data members or method invocations. Temporal reasoning is available for applications that need to operate in real time, and is supported by extended features of the language.

XML Rule Representation

The ILOG JRules rule engine can directly parse and output rules in an XML representation, allowing the management of rules by standard XML tools, and the sharing of rules among ILOG JRules applications. This is made possible through XRL (eXecutable Rule Language), which can be used to define business rules in XML. XRL is an ILOG JRules specific XML rule representation that includes the full ILOG rule language; any rule that can be written using the ILOG rule language can be represented in XRL.

Rule Kit

The Rule Kit contains a common set of tools delivered with both ILOG JRules and ILOG Rules. It features an integrated development environment, rule editing capabilities, and business rule language support for business level representation of rules. This common environment is perfect for developers who use both Java and C++, enabling them to use the same tools no matter what language the application is written in. Because of the mutual interface, rules can be shared between Java or C++ applications — meaning easy conversion of applications from one language to the other. The rule editor can be used to

design rule editing capabilities directly into an application so rules may be maintained by business users or analysts, without the need of a developer.

The Rule Kit, combined with other business rule features of ILOG JRules (described later), allows developers to create custom rule management environments. Rule management goes beyond simple rule editing. A proper business rules application should provide some mechanism to create, edit, maintain, debug, and deploy business rules and all associated data. When used with the JRules Repository, the Rule Kit provides all of the features necessary to create a rules management environment custom to the needs of the application. Business rules management and its appropriate features are described in detail in the section *ILOG JRules Business Rule Tools.*

Rule Builder

The Builder is a graphical environment for developing and debugging rule sets. It manages projects, provides a set of sophisticated editors to create and modify rules, and has integrated debugging, inspection and tracing tools to simplify the process of developing and debugging a business rule application. It's flexible, with the ability to monitor, manage and debug multiple rule engines and rule sets. Applications can remotely connect to it, or it can connect to multiple rule engines running in a remote application.

A challenge in debugging any business rule application is monitoring the trace of rules as they are placed on the agenda and executed. ILOG JRules tackles this challenge by providing the only integrated development environment for rules that includes a tool for the graphical representation of run time relationships between rules — the Profiler. See the section titled "ILOG JRules Development Tools" for further details.

Rule Editors

The Rule Kit enables developers as well as end users to modify rules by providing a flexible rule editor component that can add business rule editing capabilities to any Java application. The rule editor is available as a JavaBean for Java applications and as a web-based component for web applications. The rule editor interface is a point and click interface that can be used to create or edit rules in a business level syntax that business people can use and understand.

Business Rule Language

ILOG JRules is the only rule engine to provide a flexible and extensible business rule language — a rule language with a readable business level syntax. A business rule language uses a business rather than technical vocabulary, and allows reasoning on an object model that reflects the structure of the business domain, rather than the underlying Java implementation. This places the command of business rules in the hands of

business users. The language can be made as simple or complex as necessary, providing business users with the freedom to create complex rules or constraining them to work within a predefined scope of the business domain. The combination of the rule editor and customizable business rule language creates one of the most powerful features available in today's rule engines. Maintaining business rules is no longer quarantined to the realm of application developers, but opened up to the domain of business users themselves. Instead of typing in rules, they are constructed point and click, customized to the terms and vocabulary of the business world.

Rule Repository

ILOG JRules products persist business rules and all relevant business data in a in a structure called a "repository". The repository is the central place for an application's business rules data, and is much more efficient that traditional file-based persistence schemes. It contains the application's projects, business rules, Business Object Models, rule templates, and other relevant information. Business rules are organized in packages that can be nested at an arbitrary depth, much like files in a standard file hierarchy. This allows you to structure your project in a way that is closer to the logic of your application.

The diagram below gives an overview of the structure of a business rules repository:

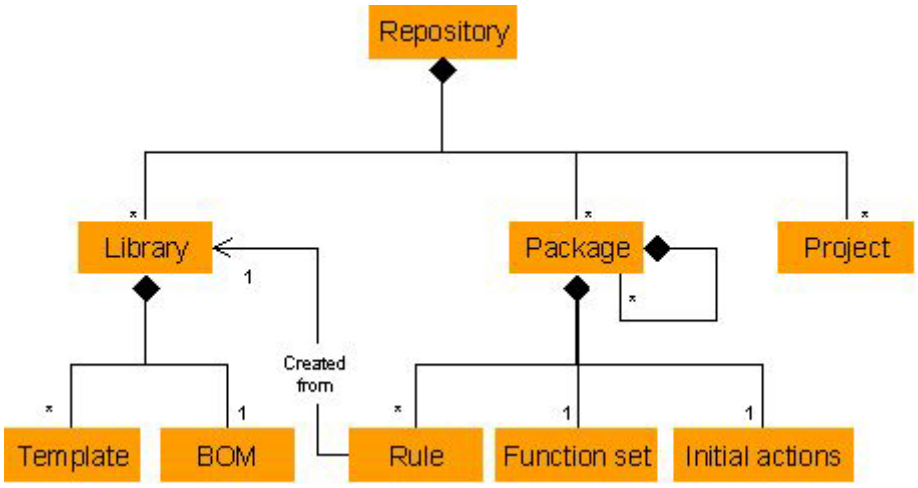


Figure 2 - The ILOG Business Rule Repository structure.

A project is a collection of references to packages and libraries of the repository. The libraries include all of the relevant data used throughout the business rules, including Business Object Models, rule languages, and rule templates. These components are described in detail later in this document.

The Rule Kit contains a common set of tools delivered with both ILOG JRules and ILOG Rules. It features an integrated development environment, rule editing capabilities, and business rule language support for business level representation of rules. This common environment is perfect for developers who use both Java and C++, enabling them to use the same tools no matter what language the application is written in. Because of the mutual interface, rules can be shared between Java or C++ applications — meaning easy conversion of applications from one language to the other. The rule editor can be used to design rule editing capabilities directly into an application so rules may be maintained by business users or analysts, without the need of a developer.

Implementing ILOG JRules

ILOG JRules is the most feature rich rule engine available. Just like the Java class libraries, ILOG JRules is packaged to allow you to use only those features required for your application. ILOG JRules does not impose upon the application or the technical architecture of your application. ILOG rule engines are designed to be fast, flexible and lightweight. Integration of ILOG JRules into a new or existing application does not mean a heavy, intrusive server loaded with every available feature — whether you need them or not. With ILOG JRules, only what is needed is deployed in your application. In fact, you can have a rule engine up and running with only two classes. There is no need to use the database tool if you have your own way to access a database, no need to include the EJB package if you're only writing Java classes. *No architecture is imposed*. Use ILOG JRules in the best way that fits your application and its architecture — this is what makes ILOG JRules unique to business rule application development.

4 ILOG JRules Rule Engine

The value of any business application depends upon how well it can adapt to future requirements, operate under high demand, and integrate into existing enterprise systems. Using third party software packages or libraries can often complicate deployment; restrictions may be required because of proprietary technology, inflexible design or weak support for existing platforms. With ILOG JRules there are no restrictions. Anywhere Java is used, ILOG JRules can be used. Integration into any Java, EJB or J2EE environment is assured. The libraries that come with ILOG JRules extend business rule technology to the world of Java, providing a high performance rule engine flexible enough to deploy anywhere.

ILOG JRules API

Java provides flexibility, reuse and stability with platform independence. ILOG provides flexibility and adaptability at the level of business rules by providing the most extensive set of Java packages available for deploying a business rule application. ILOG JRules is implemented as 100% pure Java code and controlled by a set of APIs. The ILOG JRules APIs provide full access to nearly all parts of the product, including the following:

- Rule engines
- Repository
- Rule Builder components
- Web Builder components
- Rule construction
- Business Rules Languages
- Rule debugging
- Java object introspection
- Rule query
- Rule profiler
- Database access
- EJB deployment of the rule engine and the rulesets

ILOG JRules Deployment

Standards based and platform independent; the Java solution for business rules should provide highly optimized and scalable rule engines; engines that can be seamlessly integrated into Java applications, Enterprise Java Beans and relational databases.

Supports Common Architectures

ILOG JRules easily integrates into any Java application, including EJBs in the J2EE architecture. ILOG JRules has been successfully used in applications using CORBA, COM, IBM MQSeries, RMI, and Java servlets.

As described in the previous sections, ILOG JRules is a complete programming environment. Application developers may write rule-based programs as either stand-alone applications or as a rule engine integrated into new or existing Java applications. ILOG JRules is thread-safe and can be used in both client-server and multi-tier environments.

Seamless Java Integration

While traditional rules-based systems often require a proprietary language to define the objects used by the inference engine, the ILOG JRules engine directly infers from Java objects in the application - *without any duplication*. There is no restriction on the types of classes that are usable in the ILOG Rule Language. Any application classes can be used, and there is no need to derive the application classes from the ILOG JRules classes.

Utilizing Java introspection, rules are directly applied to the application's objects, meaning ***ILOG JRules rules only carry a reference to the object***. Consequently, no intermediate layer between the application classes and rules is required. This saves memory and increases performance, as no synchronization between 'shadow' objects and application objects is necessary. Application developers are free from any burdens or restrictions about the way application classes are developed.

XML Binding

ILOG JRules is an **XML-centric rule engine**, meaning that an XML binding mechanism is available that allows rules to be written that **reason directly on XML objects**. This functionality has been integrated into the core of the rule engine, enabling the rule engine to treat XML objects just as it does Java objects. Since the rule engine parses the XML into its own representation, there is no need for adapters or translation code (such as DOM, SAX, or JDOM). XML objects can be directly referenced in rules with the same syntax as for Java objects and can be asserted, retracted, and updated, just like Java objects. Java and XML objects can be freely mixed in rules and in working memory.

ILOG JRules XML support is based on the W3C XML Schema, is fully dynamic, and involves no Java code generation. Accessing XML objects is as efficient as accessing Java objects.

While the XML binding capability of ILOG JRules is a core data representation as opposed to an ad hoc approach found in many XML-enabled applications, the real power lies in coupling this centricity with an XML version of the ILOG Rule Language. These two pieces set the stage towards an open exchange of business rules, which will be discussed in section 5, titled "ILOG JRules Rule Language."

EJB Integration

ILOG JRules is EJB and J2EE ready, benefiting from all of the middleware services provided by the application server – load balancing, persistence, and fail over. *ILOG JRules is the only rule engine that can be embedded directly into Enterprise JavaBeans, both entity and session beans.* ILOG JRules provides the most flexible rule engine integration available for EJBs, including a **rule engine server** packaged as an Enterprise Bean.

ILOG JRules includes a package, `ilog.rules.ejb`, containing the classes required to integrate an ILOG JRules rule engine into an entity or session bean. The `IlrEjbContext` class, the class used to implement the rule engine within an EJB, is serializable. The IlrRuleset class is also serializable so the state of the rule engine can be persisted in an entity bean or a stateful session bean. In addition, business rules can directly reference EJBs, evaluating data stored in entity beans or invoking methods on session beans. This level of integration enables application design and development without restriction — the ILOG JRules rule engine, when embedded in an EJB, acts as a good citizen on the J2EE platform and is managed by the application server.

Entity Beans

Entity beans usually represent business domain objects. They persist beyond a client-server interaction and are shared by many users. ILOG JRules can be embedded in an entity bean to perform data validation or add business logic, such as implementing an entity bean called OrderBean, in an e-commerce application. This bean could represent the placement of an order for goods. ILOG JRules could be embedded in this bean to compute taxes or shipping charges.

Session Beans

Session beans are transient and implement business processes. There are two types of session beans: stateless beans, which contain no memory of previous transactions, and stateful beans for more complex transactions that last for multiple method calls. The ILOG JRules rule engine can be embedded in either type of session bean to implement business logic, such as an e-commerce application that has a stateful session bean, ShoppingCartBean, that represents the products in which a customer is interested. ILOG JRules could be embedded in this bean to compute the price using pricing rules for discounts and special promotions.

EJB Rule Engine Server

To simplify the development of scalable EJB applications, ILOG JRules includes a rule engine server enterprise bean, and rule set enterprise bean in the `ilog.rules.ejb.beans` package. The rule engine server does not need to be purchased separately. It is included in the ILOG JRules product!

The rule engine server is a stateless session bean that runs within — *not outside of* — the application server environment. This allows the application to take full advantage of all of the middleware services offered by the

application server. Furthermore, the ILOG JRules rule engine server can be managed and administered consistently — in the same manner as any other EJB.

The interface to the rule engine server provides a simple invokeRules method that takes two arguments: a primary key that identifies a rule set entity bean containing business rules, and a session data class containing the data on which the business rules are to be evaluated and the result of rule set execution. This means EJB applications can invoke the rule engine simply with a single API call.

For scalability and performance, the rule engine server session bean transparently manages a shared pool of rule engines containing pre-parsed rule sets. This avoids rule set parsing and rule engine creation time when business rules are evaluated. When business rules are modified at execution time, the rule engine server rebuilds the rule engine pool associated with the modified rule set, so business rule changes are instantly available to subsequent transactions and in-progress transactions are not affected.

Performance

The pattern matching that must be done by a rule engine takes most of the execution time. Although the Rete algorithm in ILOG JRules is highly efficient and optimized for the Java language, there are programmatic ways of improving the performance by modifying the condition part of certain rules. Some performance tuning techniques are general and apply to any rule engine, such as placing the most discriminate tests, or those with the most selective conditions or groups of conditions, at the beginning of a rule. While these are mostly a matter of programming style, ILOG JRules provides real support in the form of product features for maximizing performance; rules can be compiled directly into Java code, and *automatic* optimization is available for certain types of rules. Two other developer-implemented methods are also available: finders and auto-hashing.

Compile Rules Directly into Java

A unique feature contained in the tools package is the ability to compile rules directly into Java source code. The resulting file can be compiled with the application, just as any other application code. Depending upon the rules, compilation can improve processing performance of the engine by a factor ranging from 4 to more than 10. The more complex the rules and the more objects there are in working memory, the bigger the gain. Compiling rules also reduces the activity of the garbage collector at runtime, thereby enhancing performance.

Rules can be compiled into a Java class either from within the application code, or using the IlrCodeGenerator tool provided in ilog.rules.tools. The

generated classes are included in the application in a way similar to non-compiled rules:

```
// Create an IlrRuleset
IlrRuleset myRuleset = new IlrRuleset();

// Add rules to myRuleset
myRuleset.parseCompiledRules(new CompiledRules());
```

Here, `CompiledRules`() is the generated class that contains the compiled rules.

If the application generates or modifies rules dynamically, code generation can still be implemented using methods contained in the rule engine API.

In addition to enhanced performance, compiling rules has the benefit of rendering the rules unreadable. That is, if the contents of a rule must remain confidential, rule compilation obfuscates the rule.

Automatic and Developer Specified Optimizations

Automatic rule optimizations occur transparently to the developer. Other optimization features are requested through the use of an optimization file, read by the rule engine. Developers can use these optimization features when appropriate.

Automatic Rule Optimization

Automatic optimization occurs with rules being internally rewritten by the engine to use the **from** and **in** keywords. The use of these keywords reduces the number of objects on which pattern matching must be done. ILOG JRules will automatically rewrite rules to use these keywords whenever possible. See the section on “Relations” under “ILOG JRules Rule Language” for a more detailed description.

Auto-hashing

Auto-hashing is a developer controlled tuning that allows the building of a hash table to improve the performance of equality tests in rules. Often, a test involves bound variables from a *number of different rule conditions*. In this circumstance, auto-hashing dramatically reduces the number of combinations that must be evaluated. For example, if there is a rule that matches a Mayor object with a City object (i.e., find the mayor of a city), auto-hashing organizes the Mayor and City objects in working memory using hash tables. That is, the cities may be classified by zip code in a hash table. The index used to classify a city could simply be the first 2 digits (e.g., the 94 of 94260) of the zip code. Auto-hashing gives the Mayor objects this same kind of organization, because the rule engine knows that there is an equality test between City and Mayor objects in working memory.

Given a City, a hash table index is computed. This index is then used to seek the hash table created for Mayor objects. Auto hashing brings the speed of hash tables to rule evaluations, and can significantly reduce the number of equality tests required by a rule.

Finders

Finders allow the programmer to speed up the execution of rules by providing application-specific code as a substitute for certain rule conditions. These substitutions are in the form of class methods, which enable more efficient pattern matching on specific classes.

For example, suppose the application defines a Customer class with two fields, `zipcode` and `income`:

```
class Customer {
    public int zipcode;
    public int income;
};
```

The simplest way of writing a rule condition in the ILOG Rule Language syntax that matches a Customer with a given `zipcode` and `income`, is:

```
Customer(zipcode == 20886; income == 120000);
```

This can be very inefficient if there are many customers, because all of the customers need to be asserted into working memory and the condition (zipcode == 20886; income == 120000) tested for each one of them.

There is a more efficient way of writing this condition in Java code. The following method retrieves a list of customers from their zip code — using a hash table, for example:

```
class Customer {

   public static Customer[]
   findByZipCode(int zipcode) {

   //Java retrieval code here...
   }
};
```

The previous rule condition can be rewritten so this method is used instead of the explicit pattern matching syntax shown above:

```
Customer(income == 120000) in
Customer.findByZipCode(20886);
```

This makes the rule much more efficient because it restricts the number of objects on which the pattern matching occurs. Moreover, not all of the customers need to be asserted into working memory (because of the use of

the **in** statement — see the section on “Relations” under “ILOG JRules Rule Language”).

Finders allow developers to achieve this optimization without explicitly rewriting each rule to include the Java method. A developer simply declares finders methods, such as `findByZipCode`, in the optimization file. The engine detects these methods and automatically uses them in rules, as if the rule condition contained them explicitly.

5 ILOG JRules Rule Language

ILOG JRules uses a sophisticated language, the ILOG Rule Language, which features Java-like syntax, XML support, and powerful language extensions. There are forty keywords in the ILOG Rule Language. Combining these with the features of Java provides the flexibility to implement any business logic quickly and easily. This section describes the ILOG Rule Language, a language ideally suited for programmers. Business rule language features, enabling business users with no programming experience to write rules, are described in the section titled "ILOG JRules Business Rule Tools".

Java-like Syntax

ILOG JRules uses Java as the basis for its rule language and exploits all Java's object-oriented features, including common expressions and tests, interfaces, arrays, loops, and scope management. Literals are the same syntax as Java, and can be values that are Boolean, integer, floating point, character and string. The syntax of ILOG JRules identifiers is identical to that of Java, except for the reserved keywords.

Rule Structure

An ILOG JRules rule is composed of the following three parts: a header, condition part and action part.

```
rule ruleName { [ priority = value; ]
 [ packet = packetName; ]

when { conditions ...  }

then { [ actions  ... ]   }

};
```

The header defines the name of the rule, its priority and packet name. The condition part begins with the keyword **when**, and is also referred to as the left-hand side (LHS) of the rule. It defines the conditions that must be met in order for the rule to be eligible for execution. The action part, which begins with the keyword **then**, is referred to as the right-hand side (RHS) of the rule. When all of the LHS conditions are met, the RHS of the rule is executed (or 'fires').

A rule's priority controls the sequence of rule execution. The packet name associates a rule with a rule packet. Packets provide a way of organizing rules into groups of related rules, such as those that pertain to a common domain of expertise. Packets can be activated or inactivated through the rule engine API or the ILOG Rule Language itself.

For instance, implementation of the following rule is shown below:

If the loan application is for a first time homebuyer, and the lien type is first mortgage, then the occupancy status must be principal residence.

Left Hand Side - Condition

```
rule OccupancyStatus
{
   packet = mortgage
   when
     {
       ?loanApp: Loan( firstHome == true;
          lienType == FIRSTMORTGAGE );
       Property( occupancyStatus !=
          PRINCIPALRESIDENCE );
     }
```

Right Hand Side - Action

```
then
   {
     execute ?loanApp.GenerateMessage(
        "Occupancy status must be principal
        residence.");
   }
   };
```

The ILOG Rule Language is Java-like in structure, and easily references application objects directly. The name of this rule is `OccupancyStatus`. It is part of a larger group of rules called `mortgage`, which likely pertain to the borrower's eligibility for a mortgage loan. `firstHome`, `lienType`, and `occupancyStatus` are attributes of the objects `Loan` and `Property`. `?loanApp` is a variable, bound using the colon operator, to the `Loan` object. `execute` calls the Java method `generateMessage(String message)` on the `Loan` object, which displays a message to the loan applicant.

XML and Exchange of Business Rules

The XML representation of rules alone doesn't promote their sharing between applications. Because rules must operate on objects associated with the application, having the rules but not the business objects is like providing a car without wheels. However, with ILOG JRules and the XML-centric approach taken in designing the rule engine, using XML the business objects can travel along with the rules.

This is possible because XML is now a central data type for the rule engine, which can interact with XML objects, defined in schemata, the same way as it does with Java objects. Business objects, whether they are Java or XML interact with ILOG JRules rules in the same way, the same rule will work identically on Java or XML objects, with no changes in syntax or special keywords. The power of XML becomes evident when rules are exported as XRL (through an api in the ilog.rules.rulefactory). Rules expressed in XRL, combined with business objects defined as XML schemas, enables the true exchange of business rules, where rules and data travel together and ILOG JRules applications can communicate freely with one another, sharing rules and data.

Advanced Language Features

One of the main advantages of using a rule engine is the ability to express business rules declaratively — that is, without procedural instructions related to processing. Business rules that demand the use of time as a factor, or rules that reference large numbers of objects, would require a considerable amount of procedural code. The ILOG Rule Language provides the following extensions to address some of these issues in working with rules, objects, and time.

Collections

When a rule must process a set of related objects, the rule's conditions will apply to each *relevant* object present in working memory. This causes the rule to execute once for each object that matches the condition. If this is not the desired behavior, two or more rules are required to prevent the rule from firing multiple times: one rule that behaves as desired, and another rule to retract all of the 'extra' objects from working memory once the rule has executed, thus preventing it from firing again. To avoid this clumsy approach, ILOG JRules provides a collection feature that enables the developer to handle such requirements in a single rule.

The collect feature is used to collect objects that satisfy a given condition, and allows them to act all within the same rule firing. For example, the following rule finds groups of 10 or more employees who are under the age of 35, belong to a technology company, and whose average salary (as a group) is more than $100,000:

```
when {
  ?s: Company(domain == IT);
  ?c: collect(new EmployeeCollector(?s))
        Employee(company == ?s; age < 35)

      where (size() > 10 &&
         EmployeeCollector.averageSalary()
   > 100000);
}
```

```
then {

  //do something with the collection of
employees...

}
```

The first condition binds objects of type `Company` that are in the IT domain, to the variable `?s`. The variable `?c` refers to the collection object, `EmployeeCollector`, which collects `Employee` objects that have data members `age < 35` and `company` equal to the result in `?s`. For the **collect** statement to be true, there are two tests that must be evaluated in the **where** statement. The default collector method `size()` is used to test that there are more than 10 employees collected in the `EmployeeCollector` container, and the user defined method `averageSalary()` is used to test that the average salary of that collection of employees is > $100,000.

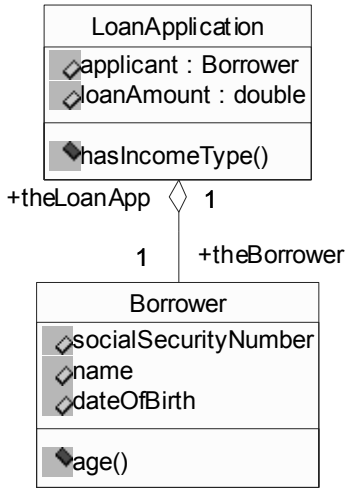
Without the collection feature, multiple rules would have to be written to accomplish this same task. Not only does this simplify the amount of code that has to be written, the use of collect reduces unnecessary rule executions, freeing up system resources.

Relations

Rules can access objects that have been asserted into working memory. With relations, rules can access objects that are *related* to objects that have been asserted into working memory, but have not themselves been asserted. That is, relations allow a programmer to navigate an application's object model in rules, using object references that are either data members or method invocations. Prior to this feature, all related objects had to be asserted to working memory and objects had to be related through bound variables in rule conditions.

The **from** statement is used in the condition part of a rule to express 1 to1 *relations between objects*. Using the from statement enables objects that are linked to other objects by fields or methods, to be accessed, even if they do not reside in working memory. This enhances performance since the pattern matching is done on a reduced number of objects, and not on all of the relevant objects that exist in working memory.

For example, in the model shown below, assume the relationship between `LoanApplication` and `Borrower` is implemented with a data member on `LoanApplication` of type `Borrower`, called `applicant`.



In this case, once the `LoanApplication` object is asserted to the rule engine, rules can reference associated `Borrower` objects in rule conditions using the **`from`** keyword, even if the `Borrower` object has not been asserted into working memory:

```
when {
   ?l: LoanApplication(loanAmount > 100000);
   ?b: Borrower(age() < 25) from ?l.applicant);
}
then {

   //do something...

}
```

The first condition in this rule binds all of the loan applications in working memory that have a `loanAmount` > $100000, to the variable ?l. The second condition uses this reduced set of loan applications to find those that have borrowers under the age of 25 years. This is possible, using the **from** keyword, even though no `Borrower` objects are in working memory.

The **in** statement is similar in function to the **from** statement (used for 1:1 relations), except that it causes conditions to be evaluated against all associated objects:

```
when {
   ?l: LoanApplication();
   ?i: IncomeType() in ?l.hasIncomeType();
}
then {

   //do something...

}
```

In the above example, again only `LoanApplication` objects are in working memory. The method `hasIncomeType()` may return a vector, array, etc. of `IncomeType` objects contained in a given loan application object. The variable `?i` will return the `IncomeType` objects used in each loan application, even though `IncomeType` objects have not been asserted into working memory.

Since the pattern matching is done on a reduced number of objects and not on all the objects in the working memory, using relations can actually enhance performance tremendously, depending upon the number of objects. When used with the collections feature, relations enable the expression of logical conditions such as “all”, “at least” and “none”.

Temporal Reasoning

Temporal reasoning is the ability to reason about events that depend on time. Often, real world applications operate in real time and therefore need to incorporate time as a parameter in their reasoning. Without a mechanism for temporal reasoning, rules that require a representation of time would have to obtain it as an attribute on objects in working memory, and complex rules that controlled and synchronized timing between the application and rule engine would have to be written.

To avoid these complications, ILOG JRules has incorporated temporal reasoning. Through the use of a waiting period, rule actions can be prevented from executing until a prescribed time period has been reached, or can fire immediately if the rules conditions become satisfied during the waiting period. A separate (different) action — also known as a timeout action — can be programmed within the same rule to execute if the waited condition does not occur within that specified time period.

The keywords used in the ILOG Rule Language to implement temporal reasoning are: **wait, timeout, logical,** and **until.** The wait statement is used in the condition part of a rule and operates by creating a timer that delays the execution of the rule, allowing the validity of the condition to be tested during the designated waiting period. If the time period elapses, a timeout action can be executed. This is shown in the rule fragment below:

```
when {
   index: StockIndex(name == DJIA)
   ?e1: wait 5 {
      Stock(symbol == MSFT && ask < 50)
   }
}
then {

   //do something if wait succeeds...

   timeout ?e1 {

      //do something else if wait fails...

   }
}
```

The rule contains two conditions. The first condition matches on `StockIndex` objects that are of type DJIA. The second condition, **wait 5**, waits 5 time units (defined by the application through methods on `IlrContext)` for a `Stock` object to be inserted into working memory that is of type MSFT and has an asking price of less that $50.

If such an object is inserted into working memory during the wait period, the rule conditions are satisfied and the rule is put on the agenda with the appropriate actions programmed for a successful wait condition. If no `Stock` object is found that satisfies the criteria, the rule is placed on the agenda with the appropriate timeout actions.

The keyword logical and **until** provide additional flexibility. The behavior for the wait logical statement is to require that all previously realized conditions remain true *during* the wait.

The keyword **until** is used to designate that the wait expression specifies an absolute time. Without until, the wait statement is evaluated relative to the time that the preceding conditions were evaluated.

```
when {
   index: StockIndex(name == DJIA; current > 9000)
   ?e1: wait logical until 20 {
   Stock(symbol == MSFT && ask < 50)
   }
}
then {

   //do something if wait succeeds...

   timeout ?e1 {

      //do something else if wait fails...

   }
}
```

The behavior of this rule is similar to its predecessor above it. The difference is that the conditions on the `StockIndex` object must remain true for the entire wait period, until absolute time 20 (i.e., the DJIA must remain above 9000). Also, since the keyword until is used, the wait period is not a relative time, but an absolute time, defined by the application. Absolute time 20 could represent 3 in the afternoon.

6 ILOG Business Rules Repository

A repository is a central place where structured data is kept and maintained. ILOG JRules uses a business rule repository, designed to store business rules and other elements required for their definition and deployment. The Business Rule Repository can be persisted to either a file structure or relational database and supports various business rule management features such as versioning, history, permission management, locking, and queries.

In the JRules repository, business rules are organized into packages. Packages allow a flexible organization scheme for business rules where business rules can be organized into groups of related business rules based on a business categorization scheme. Packages can be nested so that packages can contain not only business rules, but also other packages.

For example, in the figure below, the policy validation rules for an insurance application are stored in the Policy Validation package. The Policy Validation package contains a core set of rules common to all insurance claims in addition to the vehicle and building specific policy validation rules that are organized in the Vehicle and Building packages, respectively.

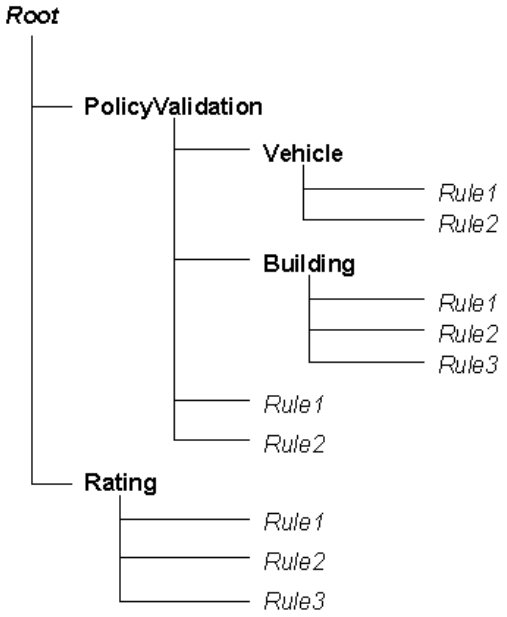


Figure 3 - JRules Repository Structure

The package structure is intended to support a flexible organization of business rules independent of how rules are deployed.

The JRules Business Rule Repository is open and extensible and can be customized to the unique needs of a particular organization. It is based on industry standards, including:

- Meta Object Facility (MOF) – OMG standard model for metadata representation and interchange. The MOF is the foundation of the JRules business rule model.
- XML Metadata Interchange (XMI) – OMG standard format for metadata interchange. XMI is used for the JRules Business Rule Repository file persistence.

Java Metadata Interface (JMI) – Java programming model for accessing metadata. JMI is defined by JSR-40 and is implemented by the JRules Business Rule Model API.

7 ILOG JRules Development Tools

Business rule applications that use an ILOG rule engine have access to a complete set of tools that simplify the process of rule development and debugging. These tools are known as the Rule Kit, which contains debugging, trace and inspection, and project management utilities, as well as a customizable business rule language and sophisticated editors for creating and modifying rules.

This section will discuss the features available in the Rule Kit that are valuable in the process of integrating an ILOG rule engine into an application and debugging the rules in the application.

Rule Builder

One of the main features of the Rule Kit is the Rule Builder. It is the main venue for creating business rule applications that include ILOG JRules, and is a fully integrated graphical environment for developing, maintaining, and debugging ILOG business rule engines. The Builder is highly customizable. The look and feel of the interface can be customized through property files, read when the Builder is launched. For instance, the content of menus, toolbars or the colors and fonts used in the GUI can be modified. Graphical styles used in the rule editors, colors and fonts, highlighting and graphics used in debugging, error, and breakpoint marking are also adjustable.

Figure *4* shows a screenshot of the Builder. Dictionary tabs for rule sets, class, and engine pages of the browser can be seen in the panel on the left-hand side. These display rules, classes referenced by the rules and available rule engine contexts. The rule editor is to the right, displaying the selected rule (OccupancyStatus) in the syntax of the ILOG Rule Language. Documentation notes and messages about the session are displayed in the two panels below the rule editor.

The Rule Builder and the engine being used in an application run in different processes, possibly on different machines, and communicate via the network. To connect the builder to an application embedded with an ILOG rule engine, the application must be started referencing the builder process (using the –D property switch in the Java virtual machine). This connects the business rule application with the Rule Builder. Once connected, the Builder and the applications execution context synchronize. The Builder can then take control of the applications execution context, receiving notification messages from the context and transmitting any debugging commands to the application. The same builder can be connected to several engines at the same time.

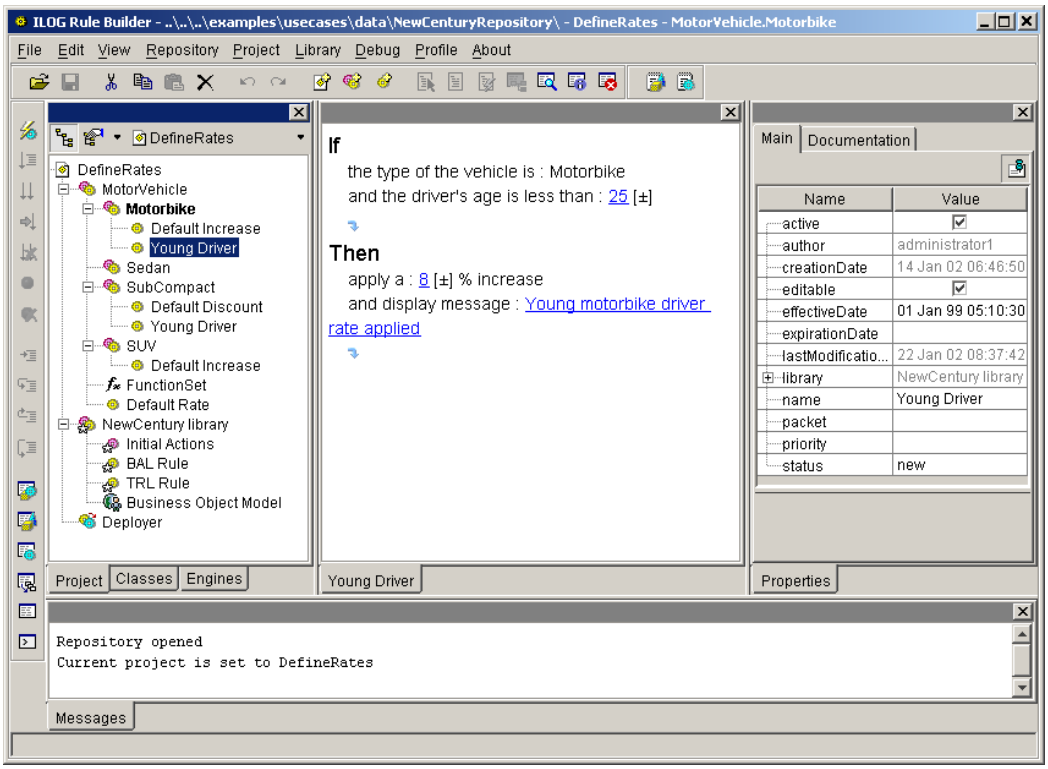


Figure 4 - The Rule Builder being used to define business rules for an auto insurance application. The project loaded is called DefineRates, with rule packages Motorbike, Sedan, SubCompact, and SUV. There are two rules in the Motorbike package: "Default Increase" and "Young Driver", the latter of which is shown in the rule editor is the center panel.

Because of this flexibility between the Builder and the application, the Builder has three modes of operation, with the ability to either take control of the rule engine, or leave it under the control of the application. In the *edit only mode*, the Rule Builder only provides editing capabilities. All debugging features are disabled. This is useful for using the editor to create rules without the overhead of debugging. In the *synchronized mode*, debugging capabilities are enabled and the Builder makes the assumption that the rule engine doesn't generate or modify rules dynamically. Under this assumption, the rules running in the rule engine are identical to the rules edited in the current Builder's project, and are displayed using the same editor. In the *advanced mode*, the Builder assumes the rules being edited in the current project, and the rules running in the applications rule engine, can be different to allow the editing and addition of rules during debugging.

Debugger

The Debugger is useful for monitoring many of the events which occur during rule execution: checking the rule set, connecting to an engine, sending the rules to the engine, resetting the engine, and firing all of the rules. It provides panels for inspecting working memory, looking at the rules that are on the agenda, inspecting the values of objects that are bound to a particular instantiation of a rule and for viewing the trace output from a given session.

With the debugger, the process of rule execution can be controlled:

- Stepping into, out of and over any given block of rule code
- Continue, Continue without Stop, Continue to Cursor, Stop, and End
- Set breakpoints on rules, classes and objects.

The working memory panel of the Debugger is shown in Figure *5*. This panel displays the list of all the objects in working memory. Double-clicking on the name of an object will expand and display the value and type of the objects various fields. This is important in determining whether a rule is behaving correctly by allowing direct inspection of the objects that it uses. For example, in the WMemory panel, an instance of the ShoppingCart object is shown with a content valuing $119.5. Rules currently on the agenda that use this value can easily be checked if there is some suspicion about their behavior. This is a great feature — to dynamically inspect objects as they affect rules.

The Bindings panel displays the name of variables currently bound in a rule. That is, if the ShoppingCart object is bound in one of the rules to a variable called ?shoppingCart, the Bindings panel would show this relationship.

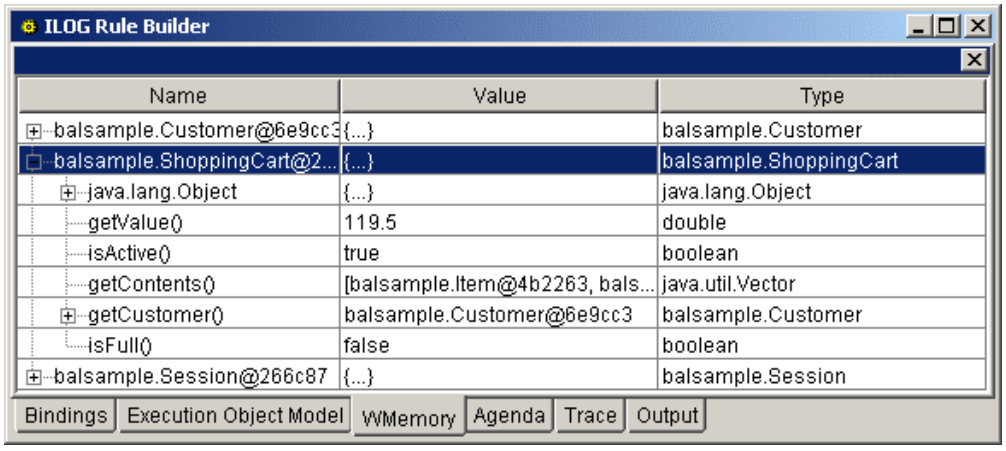


Figure 5 - View of the Debugger panel, featuring inspection of variable bindings, execution object model, working memory, the agenda, rule trace, and application standard out.

The execution object model is a view that displays the object model that the rule engine is currently using. This could be an applications objects defined by Java classes or XML data.

The current state of the agenda is also available, displaying any rule instances scheduled to be executed. Double-clicking on the rule name instance expands the rule, displaying its priority in the agenda and the objects used by the rule. This, in conjunction with the WMemory panel, provides a powerful way to quickly debug an application.

The history of all the events occurring in the engine is accessible in the Trace panel. Right-clicking the panel opens a contextual menu, which allows the user to pick the types of notification to trace. Objects, rules, or the agenda may all or individually be selected for tracing. This helps reduce the amount of data that must be filtered through by the developer during a debugging session. The Output panel displays messages that have been sent to the output stream associated with the engine.

Even with these great features, debugging an application that implements an ILOG rule engine gets even easier — with the visualization of rule executions. Such a graphical feature is available in ILOG JRules through the Profiler.

Profiler

The Profiler tool allows the evaluation of a rule set's runtime performance, enabling the visualization of rule executions, and providing summary statistics about all of the rules placed on the agenda during a debugging session. The Profiler Rules tab displays the number of times that each rule was instantiated, how often it fired, and the total and average time which was spent executing it. This information is shown in the left-hand side panel of
Figure *6*, below.

In the right-hand panel, the Flow tab visually graphs the flow of executed rules (as indicated by arrows). The number next to the arrow is the number of times a given rule caused another to fire. In the figure, the rule ClimbOntoObjectBis has caused GetObject to fire twice. If the rule has been fired fewer times than it has been instantiated, the label of the link consists of the number of times the rule has been fired followed by a hyphen and the number of times it has been instantiated. In such a case, the rule has been retracted by some other rule. The lower right panel displays which rules have retracted the selected rule. In this example, the rule PutChestOnFloor has twice retracted GetObject.

The Profiler Objects and Objects tab are two alternate displays of object level information. Here, the objects each rule has asserted, retracted, or updated are listed. Also, this view provides the number of times these actions have occurred for each class. The *Source* tab displays the source code of the selected rule.

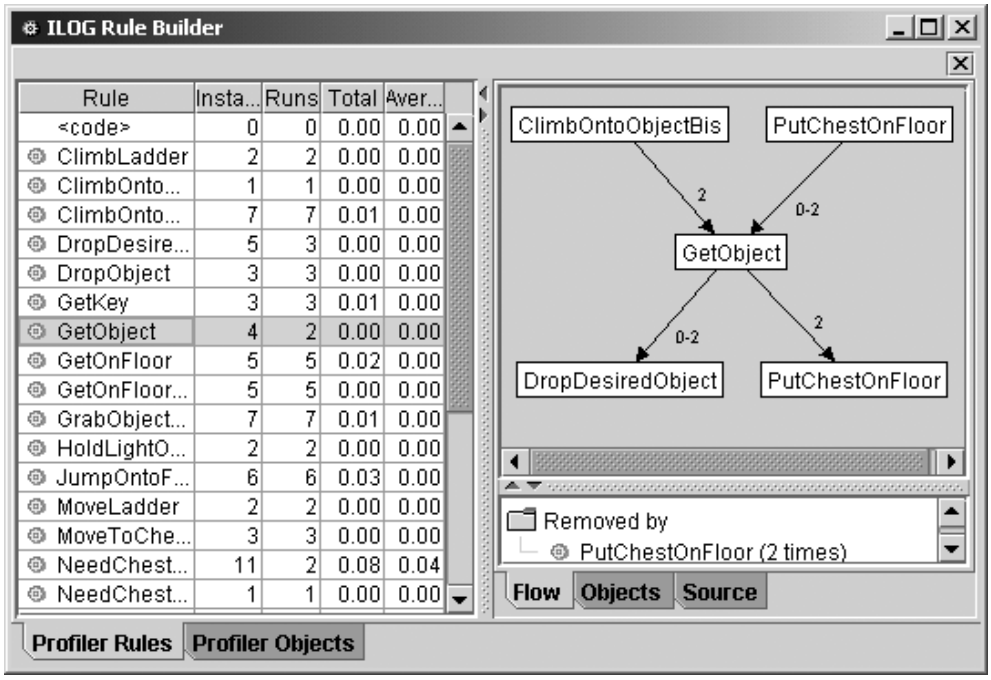


Figure 6 - A screenshot of the Profiler, showing the summary statistics and graphical display of rule instantiations and firings in the left and right panels, respectively.

DB Tool and Database Connectivity

The ILOG JRules DB Tool is a Java package that provides access to a relational database. Using DB Tool, access from an ILOG JRules application to any database compliant with the JDBC™ API is easily accomplished. With this tool and a few simple steps, relational database entities can be made available as Java objects to any business rule application.

Business rule applications built with ILOG JRules and using DB Tool can use database table rows in rules as if they were application objects, make updates into the database directly from rule actions, and notify all applications that connect to the database of changes to the table data.

Connecting to a Database

DB Tool is based on the JDBC library provided by the JDK. It also uses the ILOG JRules Rule Factory API. Three steps are involved in creating accesses to a database from within ILOG JRules:

- Generation of mapping definitions between database entities and a Java object model, including the Java methods needed to access the database.
- Integration of the Java classes into the application code, as well as various DBMS runtime environment API methods.
- Compilation of the application.

The first step is to generate the appropriate mappings from the relational database model to a Java object model. Using a graphical interface, DB Tool provides an automatic way of carrying out such a mapping. The mapping between the database and Java object model can also be defined separately in an XML file. This provides additional flexibility, as a developer may customize the mapping, rather than using DB Tool.

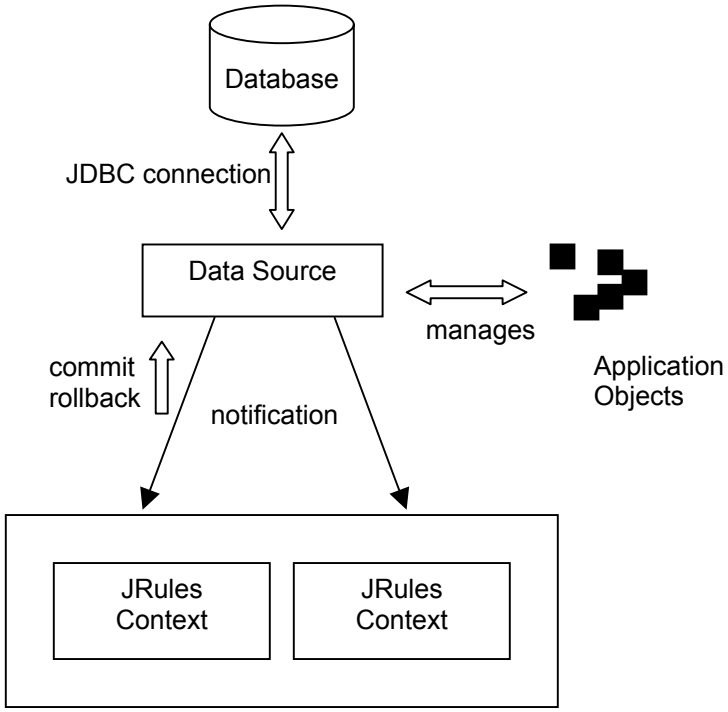
Either through DB Tool (automatically, as part of the mapping process) or a command line utility, the Java classes that access the database must be generated once the mapping is defined. The generated methods permit loading, creating, saving, and modifying objects in the database. Attributes of a Java class are used to represent the columns of a database table. There are two versions of Java classes that may be generated: cache or proxy. With the cache version, class attributes are stored in a local cache memory. In the proxy version, they are accessed from the database for every method that reads or writes the attribute.

The second step involves integrating the generated Java classes into the application. For this step, various runtime environment API methods are provided to manage the database access. The final step involves compiling the entire application to generate the executable.

Architecture at Runtime

In order to establish a connection to a database, a data source object must be created. The data source API, found in the ilog.rules.dbms package, is used to manage the database connection. This class manages the communication with a relational database and its object representations in memory, based on the JDBC library. A schematic of this architecture is shown below. Rule contexts can be notified of DB events, such as object modifications. In the same way, an integrity manager (discussed below) can be declared in order to re-evaluate relations whenever a change is made to a ILOG JRules context.

The data source has two modes: the auto-commit mode, and the manual-commit mode (default). In auto-commit mode, modifications on objects in memory are automatically and synchronously sent to the database. Manual-commit mode permits the modifications to be committed by the user before becoming effective in the database. A rollback operation permits erasure of the memory modifications made since the last commit trigger. This forces a return of the data source memory and the context working memory to their old state. In case of error during the commit operation, the rollback operation is automatically triggered before an exception is raised.



Run Time Architecture

Integrity Management

Java classes are used to represent the tables of a relational database. In a RDBMS, an association specified by primary and foreign keys between columns in two tables defines a relation. In ILOG JRules, attributes in the generated Java classes are used to define a relation. This allows navigation from an object of a source class to an object of a destination class. For example, one may navigate from a customer object to an orders object. In order to limit consumption of memory and reduce processing time, ILOG JRules defers, until accessed, pulling all of the database tables into working memory. Only when accessed are relations evaluated.

Integrity management plays a role in reducing the potential incoherence between working memory and the database. As a database evolves, a relation may become out-of-date and should be re-evaluated. The use of integrity management can force the re-evaluation of all relations after a modification is processed on database objects or re-evaluation when an object is accessed.

8 ILOG JRules Business Rule Tools

Business Action Language

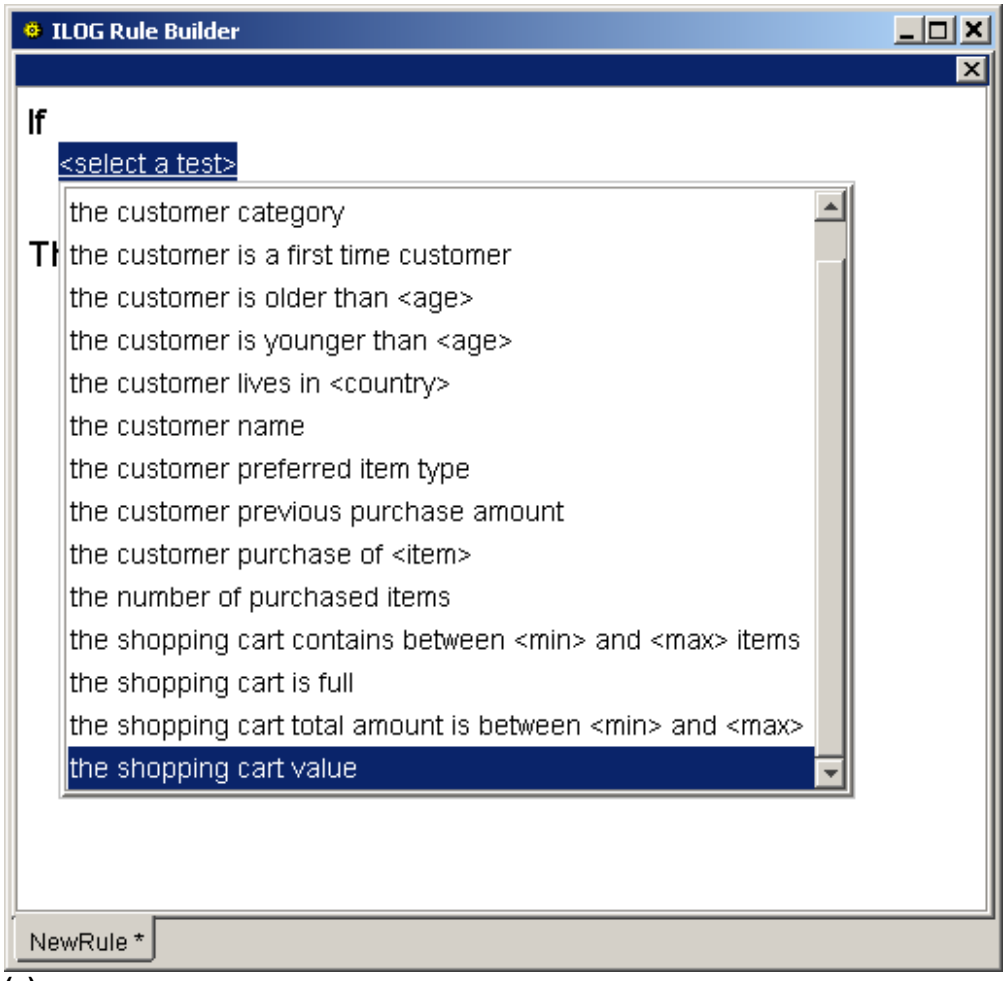
The separation of business logic from application logic is a key feature of any business rule engine. ILOG JRules and ILOG Rules further develop this separation by providing the components to put the power of rules based programming into the hands of the business user. This is achieved through a business rule language (BRL) that enables the language of business to be expressed in a rule form that can be understood by non-technical users. The business rule language is based on business terms and uses an intuitive and natural language like syntax. ILOG JRules and ILOG Rules provide a functional business rule language with the product called the business action language (BAL). The business action language provides the functionality to begin using natural language like syntax in rules, right from the start. However, *instead of forcing a single business rule language onto all users,* the business action language can be extended to accommodate domain-specific requirements. Additionally, customers can create their own business rule language from scratch, using ILOG Rule Language technology. The rule samples below show the dramatic difference between the Java like syntax of the ILOG Rule Language and the natural language like syntax available in the business action language.

```
when {
     ?s: ShoppingCart(value > 100);
     ?c: Customer(category == Category.Gold);
  } then {
        ?s.setDiscount(15);
  }
```

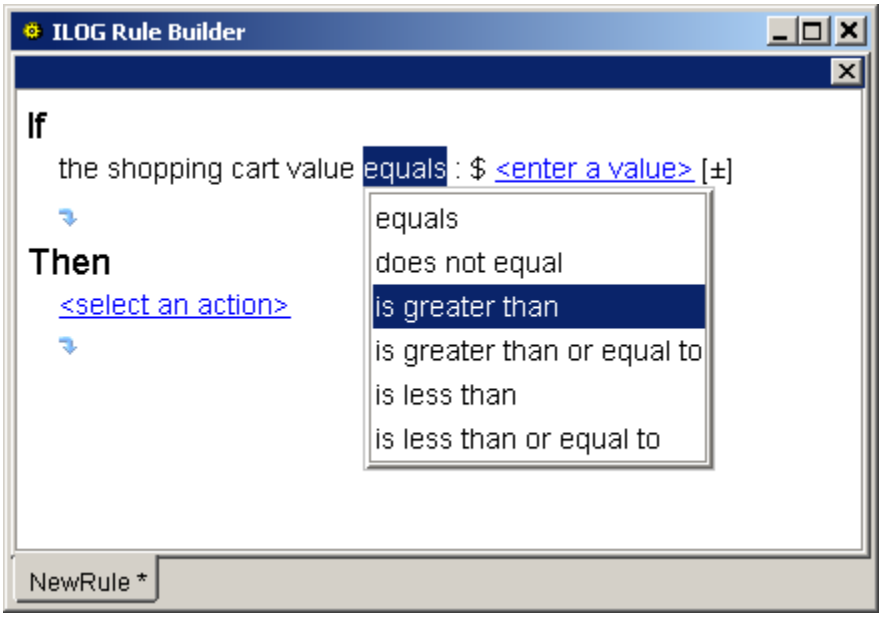
The ILOG Rule Language requires dealing with programming concepts that are not related to the business domain: Java classes, attributes, method calls, comparison operators with a non-intuitive syntax such as ==, syntactic delimiters such as semicolons and braces, and so forth — a syntax not very intuitive for most business users. Here is the same rule written in the business action language designed for e-commerce customization:

```
If
     the shopping cart value is greater than $100
     and the customer category is Gold
     Then
        apply a 15% discount
```

A striking difference in syntax! The rule is actually *constructed* in the syntactic editor, using a point-and-click interface to select terms (from drop down lists) that are used in the e-commerce domain, to build the logic of the rule. Figure 7, below, shows a series of snapshots building the rule in the editor.


ILOG Rule Builder
If
<select a test>
the customer category
the customer is a first time customer
the customer is older than <age>
the customer is younger than <age>
the customer lives in <country>
the customer name
the customer preferred item type
the customer previous purchase amount
the customer purchase of <item>
the number of purchased items
the shopping cart contains between <min> and <max> items
the shopping cart is full
the shopping cart total amount is between <min> and <max>
the shopping cart value
NewRule *

(a)



(b)

Figure 7 - Snapshots of building a rule in the syntactic editor (a point and click business rule editor) using the default business action language that comes with ILOG JRules and ILOG Rules . (a) Selecting a test based upon terms used in the business model. Here, the value of the shopping cart's contents is being selected for use in a rule, (b) Adding a conditional value to the rule.

The business action language is more than just an easy way to read rules; it can actually put the power of rule management into the hands of business users. And since the syntactic editor is available as a JavaBean or web component, developers can actually build rule management functionality directly into the application.

The components that make this possible are: the business object model, the token model, and the business rule editors. Together, they extend to the business user the ability to write and maintain rules.

Business Object Model

A business object model (BOM) is an abstract model that contains business objects that are used for business rule processing. It is a model that represents application objects, which can be Java or XML objects. Because both Java and XML are core data representations to the rule engine, the business object model can be derived from Java application classes or XML schemata; whatever implementation the application architecture uses, the business object model is an abstract representation of it. When Java classes represent the data, the business object model is derived from the classes and class members of the application. In the case of XML data, the business object model is derived from an XML file. No matter what data is presented to the builder (Java objects or

XML schemas) building rules with the business action language is the same. Building a rule is as simple as pointing and clicking.

The BOM adopts most of the concepts and keywords of Java: basic types, classes, interfaces, etc. It also includes entities that are not defined in Java, but are useful for business rule languages, like enumerated values.

When the Builder is opened, a new project is created by default, which is the root container for all objects brought into the Builder. A project contains one or more rule sets and is associated with a business object model, which is automatically created when a rule set is imported into the project. While the BOM is not always needed to write rules in the ILOG Rule Language, it is one of the key components of the business rule language, providing the foundation for translating between the domain specific language used in a business rule language and the actual implementation of the application in Java or the data represented by XML schemas. The names shown in the list of available tests in the drop down menu of Figure 5a is just such an example. This mapping can be edited in the Class Editor. Figure 6 shows the Class Editor panel, where classes in the business object model may be hidden, removed, displayed, or derived. For example, alternate names for various application object methods are shown in the DisplayName column (e.g., 'the shopping cart is full,' is used as an alternate name of the actual method, isFull( )).

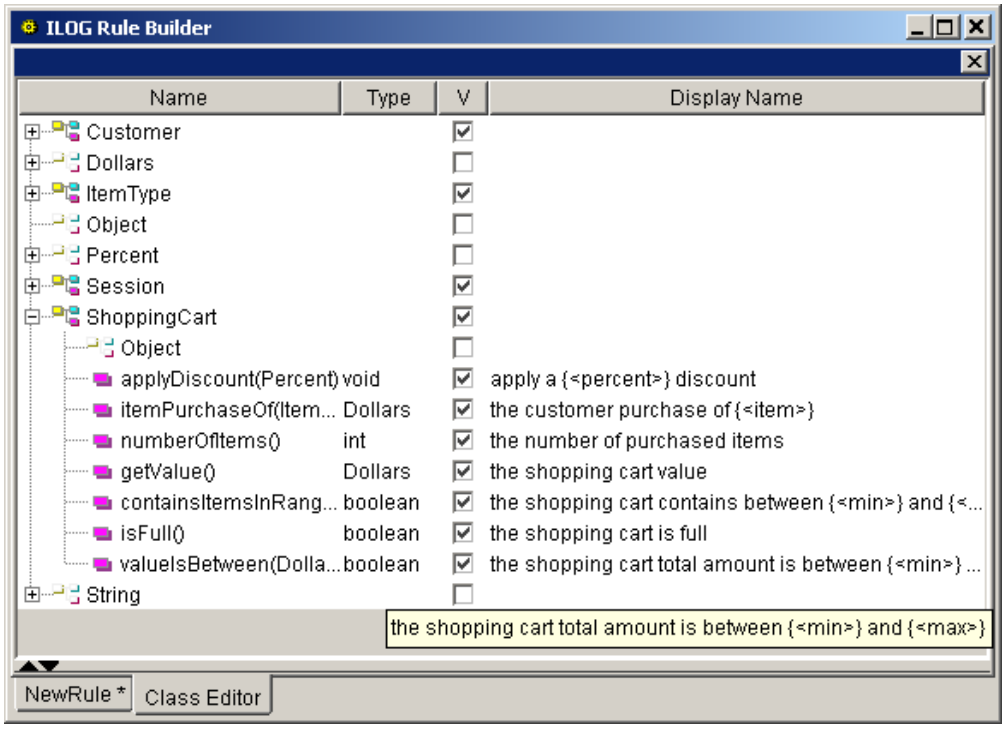


Figure 8 - View of the Class Editor panel, showing the expanded view of the ShoppingCart class in the Business Object Model.

As rules are created, additional Java classes or XML schemata may be included in the business object model simply by importing them into the Builder.

User-defined properties are stored in the Java and XML objects representing the BOM entities, which can be accessed by BOM aware tokens. This is how the business rule editors know to use these business terms instead of the underlying Java class member names or XML schema field definitions. Either extracting class definitions from actual Java classes (which is done automatically when a rule set is imported into the Builder), or explicitly defining them from scratch, can define the business object model.

A command line utility is available to create and manipulate a business object model, saving the model either in Java or XML. A separate business object model API is available, which contains a set of interfaces that can be used to define and explore the model.

Token Model

The token model is a framework for specifying the syntax of a business rule language. It describes the syntax of the language and is defined by the terms and idioms of a given business domain. In the token model, the syntax of a business rule language is expressed as a sequence of *tokens* that represent the different parts of the rule.

The various types of tokens are represented as Java classes with flags and parameters that determine their behavior. In Figure 5, these tokens are represented by *If*, <*select a test*>, <*enter a value*>, etc., and their available values as shown in drop down lists. The tokens can be made aware of the business object model, meaning that application objects modeled by the BOM are directly accessible, such as the 'shopping cart value' of the ShoppingCart object, as shown in Figure 5a.

The business rule language APIs allow developers to build such a customized syntax for any business domain. The token model is defined through a Java program that instantiates token classes, defining the language syntax. The model may then be compiled and incorporated into the Builder by way of a property file, read when the Builder is launched.

The token model framework provides several predefined business object model aware tokens, although a business rule language does not necessarily need to use them. In fact, a business rule language can be completely independent of the business object model. This might be desirable for implementing very simple business rule languages, or when using an object model that is already defined by the structure of the business application.

BRLDF (Business Rule Language Definition Framework)

JRules also includes the Business Rule Language Definition Framework (BRLDF). This framework provides a higher level and simpler way of defining

business rule languages than the JRules Token Model. Business rule languages are expressed in a BNF-like format using XML. Using this mechanism, business rules can easily be translated into executable rules using XSLT or Java.

Business Rule Editors

Although the syntactic editor (a point and click business rule editor) is integrated into the ILOG JRules Builder, two other options exist for creating rules: the syntactic editor as a JavaBean for Java applications, and the web-based component that extends the syntactic editor to web based applications. Each of these alternatives introduces the ability to integrate customized rule editing into an application.

JavaBean Editor

The syntactic editor as a Java Bean can be integrated into any Java application. Using this editor, a developer may incorporate a customized rule editor right into an application, enabling a business user or analyst to create, edit, and manage the rules of the system — all using the point-n-click ease made possible by the token model and business object model. The web enabled editor component provides an alternative to the JavaBean version of the syntactic editor.

Web Editor

The web enables rule editor can be deployed in a Servlet (Servlet 2.2 compliant) or a JSP page where web users can use it to edit business rules across the web. The editor is based on an advanced framework for server-side web-based components with a Swing-like API. The web editor works with any browser that supports JavaScript (Netscape Navigator 3 to Internet Explorer 5.5). In addition to the rule editing capabilities, the web rule browser allows for quick navigation of rules in the application by providing a list of available rules in the form of a tree.

Rule Templates

While the rule editor is a very effective tool for creating and editing rules from scratch, there may be situations where it is more convenient to start with a partially written rule. Partially written rules, or rule templates, are useful for:

- Creating simple, form-like rules that are suitable for untrained users
- Developing libraries of rules that embed domain knowledge
- Creating many rules with the same form
- Restricting the type of rules that can be written by the final user

Rule templates are created using the template editor. The template editor is similar to the rule editor, except that it has additional operations for marking parts of the rule as *frozen*. When a part of a rule template is marked as frozen, it can no longer be changed in rules created from the template. When the user

instantiates a new rule from a template, they simply fill in the unfrozen parts of the template with rule-specific values.

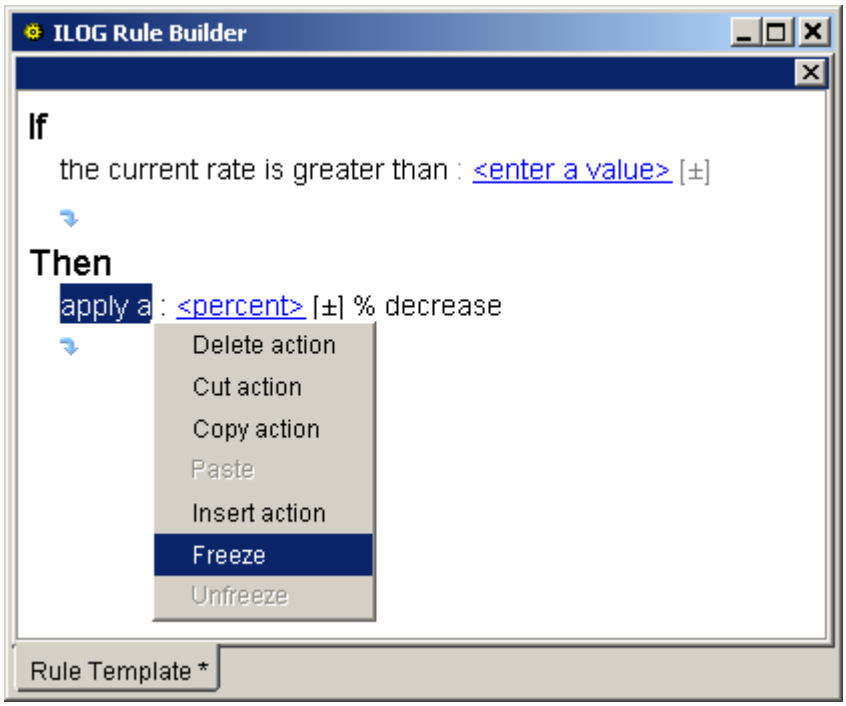


Figure 9 - The rule template editor. Here the user has decided to freeze the 'apply a <percent>% decrease' action on the right hand side of the rule. This template will be convenient when creating many rules that follow the same format and only differ on the 'current rate' and 'percent decrease' values.

Rule Properties

In addition to the information defined within the rule, users can specify additional data values, called 'properties', for each rule. Rule properties include information such as the author of the rule, the rule expiration date, and the rule priority. Properties represent rule-specific data that is necessary in the rule management process, but that does not conveniently fit within the rules themselves. The Rule Builder is delivered with a predefined set of properties.

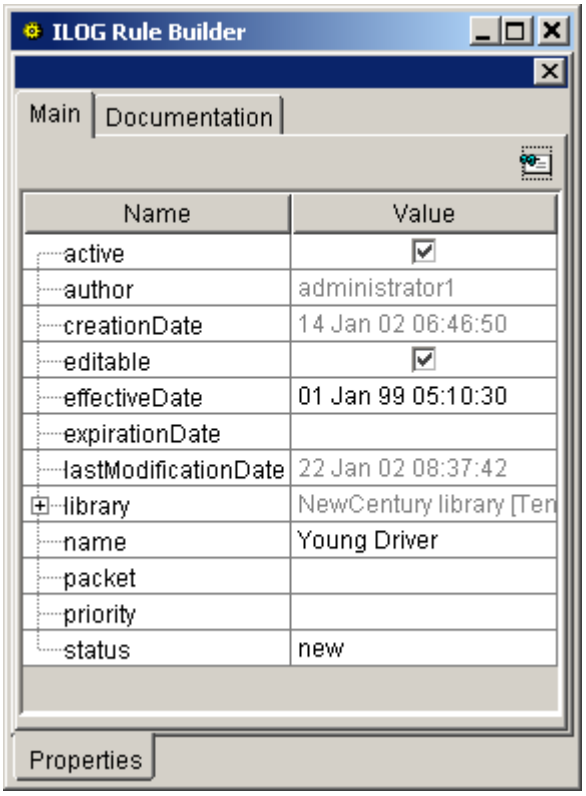


Figure 10 - The Properties panel of the Rule Builder.

In addition to the standard rule properties, users can define custom rule properties associated with a business rule, such as rule effective/expiration date and rule status. Custom rule properties are provided to accommodate the unique needs of individual organizations.

For example, effective/expiration date seems to be a well-defined, easily understood property that defines the date range for which a business rule should be enforced. However, different organizations have different meanings for these terms. An organization might wish to set the effective date to the rule creation date or some future date. Requirements may dictate that the expiration date for a business rule be compared at execution time to the system date/time, the beginning of transaction date/time, or some incoming data element such as loan origination date/time.

The JRules custom rule properties feature is designed to allow organizations to customize the repository to meet their unique requirements. Developers can add any number of custom business rule properties to the repository. These properties will be available for each business rule stored in the repository and will be accessible through business rule management functions such as query, history, and permission management.

Versioning

JRules supports versioning for business rules and the Business Object Model (BOM). Each business rule has a version property that indicates the version. When a rule is created, it is assigned the version number 1. Creating a new version of a rule makes a "backup" copy of the rule and increments the version number. A new version is always created from the last rule of a version sequence, i.e. the rule with the highest version number.

The last version of a rule (i.e., the version with the highest version number) is by default the *working version*. The working version of a rule can be edited, deleted, or moved. All the other versions are immutable. When a new rule is created, it is automatically set as the working version. Similarly, when a new version is created, it becomes the working version.

All rule versions are simultaneously available in the repository. This supports business rule management requirements, allowing a users to track business policy changes, as implemented in business rules, over time.

The Business Object Model (BOM) is versioned as a whole. A business rule always uses the latest version of the BOM of the library from which it was created. When a new version of a BOM is created in a library, the rules that have been created previously with this library automatically use the new version of the BOM and benefit from its extensions.

History

Each operation performed on a repository element is logged and stored in the repository as an event. Operations for which an event is logged include:

- Creating or deleting an element
- Changing the value of a property
- Changing the definition of an element
- Creating an element version

The logging of events is customizable so that the logging can be tailored to meet specific requirements. The JRules History feature allows users to track changes to business rules and other repository elements over time.

Query

JRules supports a query feature through the Builder that allows users to query rules by any rule property, including user-defined properties such as rule author, effective/expiration date and rule status. Users can also query by classes, data members and methods referenced in rules.

The JRules query feature supports:

- Development Impact Analysis – for example, to determine rules impacted by a change in the Business Object Model
- Business Impact Analysis – to determine the business impact of rule changes

Developers require the query/sort feature to assess the impact of a change in a JRules project on other components in a project. Business analysts/users require the query/sort feature to perform impact analysis on business policy changes.

JRules queries are written using the Business Query Language (BQL). The BQL is a business rule language derived from the Business Action Language (BAL) that is specifically tailored for querying rules. Like the BAL, the BQL is a structured-English language designed for business users.

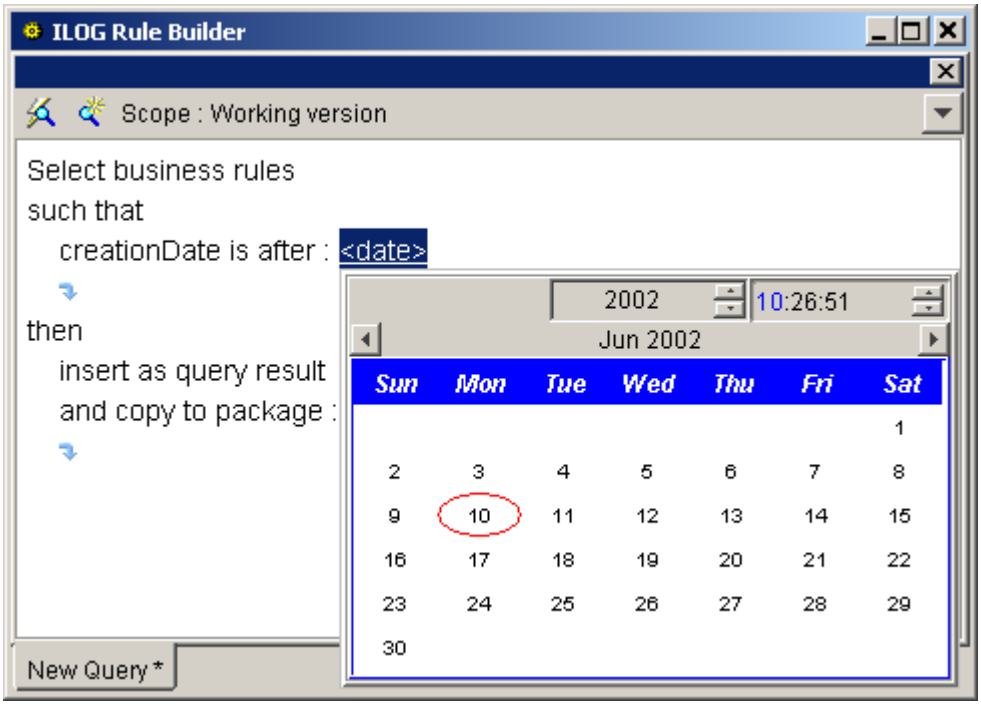


Figure 11 - The query editor. Here the user is identifying all rules that have been created after a certain date, and copying them to a specific package.

Permission Management

In JRules, a Permission Manager controls access to the elements of the repository. The Permission Manager checks operations performed on repository elements, and rejects those that are not authorized based on a user-defined authorization scheme.

Operations checked by the permission manager include all operations on the repository that retrieve a property, set or change the value of a property, remove the value of a property, or create a new version of a repository element. For example, creating, editing or deleting a rule, changing the status of a rule, or creating a new version of a rule.

All the methods of the BRM API that perform operations on the elements of a repository first call the corresponding methods of the repository's permission manager to check the validity of these operations. This is similar to the way the Java security manager checks Java API calls.

Since permissions are checked at BRM API level, and the JRules Builder and JRules Web-Editor use this API to access the repository, the Builder and Web-Editor automatically enforce permission policies defined by the permission manager. This means that a user cannot use the Builder or Web-Editor to perform operations that are not allowed by the permission manager. In addition to reactively checking that an operation is allowed when the user performs it, the Builder also prevents the user from accessing this operation at all whenever possible. For example, Builder operations that are not allowed will be disabled in Builder menus and properties that cannot be edited will be displayed as read-only.

The permission manager can compute permissions to enforce organization or application specific permission policy. For example, the permission manager could retrieve the user's name from a property file, and use a permission policy specifically associated with the user. Or, a permission manager could implement a policy that dictates that a rule with a *status* property with the status value *deployed* cannot be deleted or edited.

Locking

JRules provides basic *locking* functionality to prevent several users from simultaneously modifying the same repository element. When a project is loaded in the Builder, all the packages of the project are automatically locked. If another user loads the same project in another Builder at the same time, the user won't be allowed to modify the packages and their contents since the project is in use.

Unlike packages, the libraries of a project are not automatically locked when the project is loaded in the Builder. When a project is loaded, its associated libraries are read-only and cannot be modified. To modify a library, the user must explicitly lock the library. As long as the library is locked, other users cannot edit it.

Ruleset Extraction

In JRules projects, rules are organized into packages. Packages are intended to reflect the logical organization of the business rules, and not necessarily the deployment structure. This means that there's not necessarily a one-to-one

correspondence between packages and execution rulesets. The mapping of a repository to an execution ruleset must be explicitly specified in order to deploy the business rules. JRules includes the concept of ruleset extraction where a ruleset extractor defines how an execution ruleset is built from the repository.

Ruleset extraction is an application-specific task typically based on the value of user-defined rule properties, like the expiration date or the status. For example, rules with an expiration date that had passed or an inactive status would not be deployed. For each JRules project stored in the Repository, the developer can define a custom ruleset extractor in Java. JRules provides a default rule extractor that directly maps the current top-level package in a project to an execution ruleset.

In order to deploy business rules in an application, the required rules must be extracted from the repository, translated into the ILOG Rule Language (IRL), and made available to the application in a data source (file, database or entity bean database) from which they can be retrieved and passed to the rule engine embedded in the application for execution.

JRules supports two methods of deployment, builder-driven and application driven:

In builder-driven deployment, rules are extracted from the repository from within the Builder environment (through Builder menus), translated into the IRL and published in a data source. The runtime application then reads the IRL rules from the data source and executes the rules. In application-driven deployment, the Builder is only used to edit and save rules in the repository. The runtime application then extracts the rules from the repository, and translates them to the IRL using the BRM API.

Application-driven deployment supports business rule extraction at execution time. This is useful for applications with large numbers of rules where runtime extraction provides a performance or application design benefit. For example, assume an insurance company is developing a claims processing application where there is a common set of rules required to evaluate each claim and in addition there are state specific and vehicle specific rules. Since total number of rules in the JRules project for this application is very large, and the set of applicable rules for a given transaction is only a small subset of those rules, it may be more efficient to retrieve the state and vehicle specific rules for a given transaction at execution time rather than loading all the project's rules into the application.

Additional Resources

Product information such as data sheets, specifications, and white papers are available on the web site at *www.ilog.co*m.

For additional information on ILOG products and technologies, please visit the following web sites:

- www.ilog.com
- www.ilog.[fr/co.uk/de/es]
- www.ilog.com.sg
- www.ilog.co.jp