WHITEPAPER

Blaze Advisor

Technical White Paper

*updated for version 2.5

Blaze Software 1310 Villa Street Mountain View CA 94041

Tel 800-876-4900 650-528-3450 Fax 650-943-2752 info@blazesoft.com www.blazesoft.com

TABLE OF CONTENTS

TABLE OF CONTENTS cont'd

INTRODUCTION

In today's changing world, the success of a business depends heavily on its ability to quickly adapt itself to its market and its competition. As more and more business processes are being automated, there is a growing need for Information Technologies that can cope with change.

Most of today's information systems have been developed with procedural programming languages. Over time, the languages have improved. Object orientation has made programming safer and has promoted the reuse of components. SQL and scripting have made programming and interaction with data more accessible. These innovations have supported the development of more powerful and more sophisticated information systems but they have only provided partial answers to the issues raised by rapid changes to the business practices. The business policies are spread through the system and often expressed in different procedural languages at different levels (SQL, 3GL, VB).

Thus, changing a business policy requires several steps:

- Find where the policy is expressed (it may be replicated).
- Analyze the impact of the change that needs to be done.
- Modify the procedure that implements it (you need to know the language).
- Rebuild the system (you need to know the tools).
- Test and validate.

This process is tedious and rather inefficient, which explains why companies have difficulties adapting their information systems to follow the pace of their business.

During the last few years, technology analysts have been advocating a new approach based on Business Rules Automation. In this vision, the business policies are expressed in the form of rules and managed separately from the rest of the IT infrastructure. This brings several major advantages:

- Finding how a policy is implemented becomes easy because the rules are managed separately.
- Changing rules is easier than changing procedures because policies are naturally expressed in the form of declarative rules, and because rules are more independent from each other than procedures.
- The system can be quickly updated to take the new rules into account.
- The rules can be tested and validated with a single set of tools. This is much easier than testing and validating logic that is spread through out the system and possibly expressed in several languages.

BLAZE ADVISOR

Blaze Software shares this vision and bring it to reality with Blaze Advisor, its new generation of business rules engine. Blaze Software is the dominant company in the rules industry, delivering powerful rule engines that have been integrated at the heart of "mission critical" applications in many domains (scoring, configuration, diagnostic). With Blaze Advisor, Blaze Software brings to the market a new generation of Rules technology specifically designed to

capture, manage and execute business rules. Advisor is a complete product that supports the whole cycle of business rules applications, from development to deployment and maintenance.

Advisor has two components:

- **Advisor Builder** is a sophisticated development tool with visual editors, powerful debugging facilities and wizards to assist you in the integration of your rule-based applications with databases, Java objects, CORBA objects, ActiveX objects, etc.
- **Advisor Engine** is a versatile high performance rule engine that can either be deployed on an application server or executed directly on a client platform.

Advisor benefits from all the experience that Blaze Software has gathered around rulebased applications for over 10 years but it also incorporates a number of innovations that make it unique among today's business rules products. Blaze Software designed a brand new product with special emphasis on the following subjects:

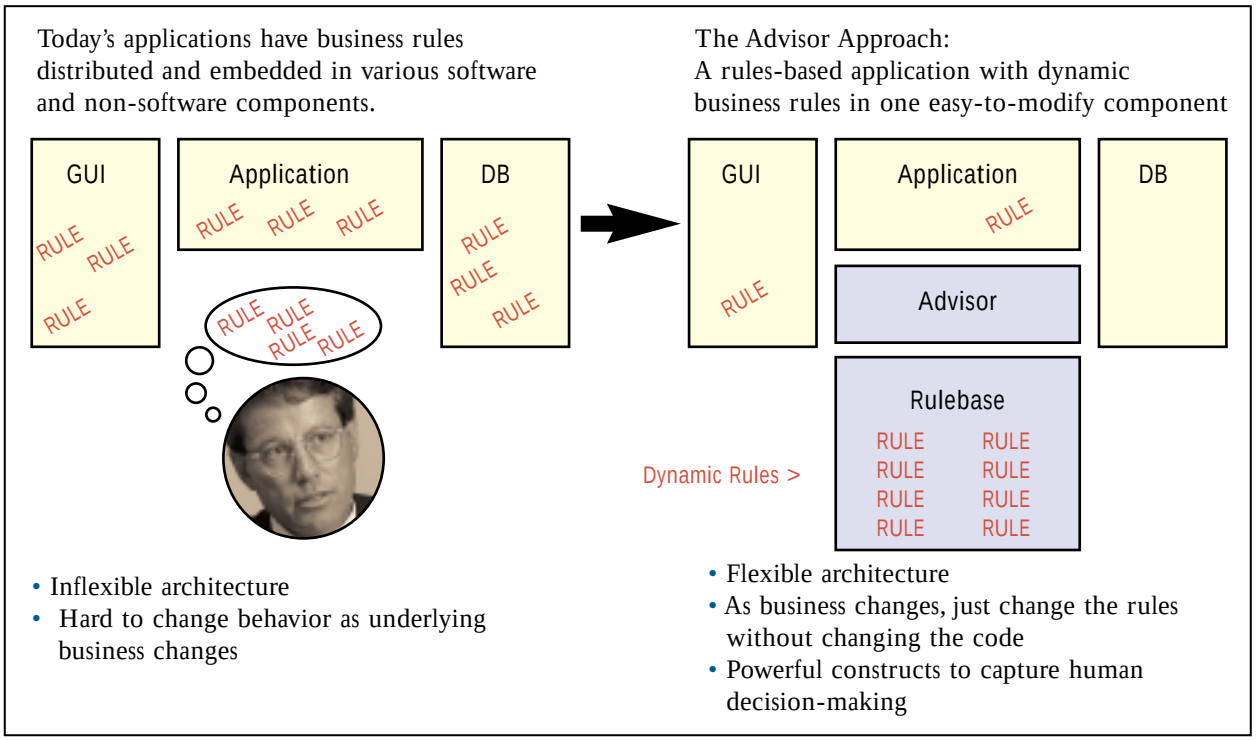
- **Ease of use**. The rules are expressed in a natural language. They are easy to write and easy to understand. A very intuitive visual tool assists the development process.
- **Integration**. Business rules do not exist in a vacuum; rules are written in relation to existing data and objects in the business environment. Because of this, ADVISOR allows developers to work with Java objects, CORBA objects, ActiveX (COM) objects, or on objects that have been retrieved from SQL databases. The development tool includes some intuitive wizards that assist you in connecting the rules to your world. Also, the rule engine can be controlled from Java, to be run as part of a desktop application, or on an application server of various flavors:

  - Web servers
  - CORBA servers
  - Publish/subscribe messaging servers, such as IBM MQSeries
  - COM/ActiveX containers such as Microsoft Transaction Server
  - Custom Java application servers

- **Performance**. The rules are indexed in a very efficient way and the engine can quickly find which rules apply to which objects, even when monitoring a large number of complex rules. In most practical cases, the rules approach compares favorably to conventional means of expressing business rules.

Altogether, the expressive power, ease of change and performance of Advisor makes for a very compelling application architecture. This is illustrated in the figure below.

BUSINESS RULES-BASED APPLICATIONS: THE NEW ARCHITECTURE



An Advisor-based application architecture compares favorably with traditional architectures in several ways:

- The dynamic rules which would otherwise be hard-coded into procedural code are placed in a rulebase which can be modified by business analysts, allowing for the behavior of the application to change without recourse to information technology professionals.
- Rules that exist in the heads of end users such as salespeople, customer service agents, underwriters, lending agents, and administrative clerks can be also described as rules in the Advisor rulebase. Thus, when a company wishes to offer its services directly to end customers through the Internet, there is no company human "agent" acting between the customer and the company. Instead, the Advisor rulebase acts as that company agent and the company's customers benefit as if there were a human interceding on their behalf. Rules can even provide for assisted service, thereby closing gaps in an employee's knowledge.
- Advisor supports the introduction of new "knobs" to control an application behavior. For example, in conventional systems it would require an MIS professional to implement new procedural code to introduce a notion of favored customers (Platinum vs. Gold vs. Silver). In effect, a new knob must be created that has settings which determine when a customer becomes platinum. Contrast this labor intensive approach to an Advisor–based architecture where the business analysts can simply define rules which determine a customer's tier without having to change anything in the overall application code. Thus, a new knob can be introduced and furthermore, its settings can be redefined at will as the company's sales team dreams up new ways to determine customer tier levels.

BUSINESS RULE PROBLEMS WELL-SERVED BY ADVISOR

Advisor provides an effective solution for many different types of business problems:

Selection
Choosing among alternatives. For example, a rule-based application to choose between various home loan packages (comparing interest rates, term, prepayment penalties, fees) across one or many lenders.

Assessment, Scoring, and Prediction
Fact and judgmental-based reasoning to determine the likelihood of some positive or negative outcome occurring. For example, insurance underwriting, credit scoring, propensity to default, propensity to take up a consumer marketing initiative.

Classification
Assignment of a type or tier to an otherwise undifferentiated transaction or entity. For example, sorting customers into high, medium, and low value. Another example is fraud detection or suspicion. Somewhat differently, validating a transaction for legitimate values is another example of classification. (Are all the fields on a customer purchase order correct? Has the expense report been completed per company guidelines?.

Monitoring
Listening to changes in the outside world and alerting or otherwise acting when some threshold is reached. Examples include monitoring the backlog of incoming customer service work to alert management to schedule overtime, monitoring the risk level of a trader's portfolio and suggesting hedging strategies, monitoring inventory levels to automatically submit re-orders to suppliers.

Configuration Verification
Checking that all pieces actually work with each other, recommending additional pieces based on pieces already specified. Good examples here are complex product order validation such as a truck, machine tool, or medical diagnostic equipment where there are tens to hundreds of options, not all of which are internally compatible and some options suggest other options for customer satisfaction.

Diagnostic and Prescription
Suggesting why some problem is occurring, recommendation of a course of action. Some examples include: help desk systems, Internet-based automotive problem assistant, and health problem detection systems.

Planning
Recommendation of a course of action based on the collection, analysis, and synthesis of facts. Examples here include financial planning tools, action plans to convert one's business to the euro, and logistics plans for large scale building construction projects.

EXAMPLES OF ADVISOR-ENABLED APPLICATIONS

There is a large range of possible applications that are well-suited to a rules-based architecture. Using Advisor, high-value business applications can be produced that provide:

Consistent delivery of customer relationship management across multiple delivery channels.
In modern financial institutions, the customer is served across a variety of delivery channels—traditional customer service representatives within a call center, branch office staff, automated teller machines, voice response systems, the Internet, and direct mail. In every transaction, the financial institution wants to maximize the customer's value to the institution. This customer value strategy is expressed through rules such as:

- Platinum customers receive low fee offers on new products;
- Tin customers are offered special debt consolidation packages with high interest rates but long repayment terms.

With Advisor, these rules (and the hundreds of others needed to implement the institution's customer value strategy) are placed in an Advisor rulebase and deployed in a server environment. This "Rules Service" can be invoked by the various client systems such as the Voice Response Unit, customer service representative desktop, Web page, and so on. Using such a rule service means that the policy is delivered consistently regardless of how the customer "touches" the financial institution.

Self-service, computer-aided selling
Today's e-commerce businesses use rules-driven applications to offer the Internet-connected customer access to the company's products and services without the intercession of a customer service agent or salesperson. In order to make this work, all of the "help" that a customer service agent or salesperson traditionally gives to a customer must now be made available to the end customer (or else the customer feels that the provided service level is worse than before and is likely to shop elsewhere).

For example, a retail catalog merchant which sets up an effective E-commerce site wants to use rules to aid the Web-ordering customer in dealing with situations normally handled by their telephone sales reps. Several areas of rules applications apply:

- Helping the customer by offering alternative goods when the selected good is out of stock. The customer may accept an alternate color or other substitute.
- Suggesting additional goods based on buying patterns and /or the current order. These additional goods may be items the merchant is trying to sell (off-season, overstock, or simply high margin).
- Suggesting alternative shipping conveyances around times when the standard shippers are exceptionally busy (like holidays).

Multiple experiments to increase customer loyalty
Traditionally, rules-based systems are used to implement a specific policy. For example, a rules system might capture regulatory rules that determine whether a person qualifies to participate

in a tax-exempt deferred savings plan. However, the rules for a customer loyalty program cannot be known in advance to be "best." It is not until after the enterprise has tried various product and service inducements and waited several months to see whether the customers are tending to remain loyal or are defecting to competitors in increasing numbers that it can evaluate whether or not the "inducement" rules were correct. To address this situation, sophisticated enterprises create parallel Advisor rulebases – one rulebase becomes the "control" rulebase and the other (could be more than one) rulebase becomes the "test" rulebase.

With large enough sample sizes, the enterprise creates a definition of a population test cell. For example, all married couples with homeowner's insurance with household income between US $40,000 and 100,000. Whenever it is time for insurance renewal, a random number is drawn. If between 0 and 0.9, the couple's recommended insurance products/services are determined using the "control" Advisor rulebase (which represents best-practice loyalty rules). If the random number is between 0.90001 and 1.00, the couple's recommended insurance products/services are determined by the "test" Advisor rulebase (which represents a hypothesized better best-practice loyalty policy). Six months later, the insurance company calculates the benefit to the insurer (on average) of all couples that went through the "control" rulebase versus the benefit to the insurer (on average) of the couples which were evaluated using the "test" insurance loyalty rulebase. If the insurer benefited more from the couples which went through the "test" rulebase, the "test" rulebase becomes the new "control" rulebase and a new "test" rulebase is established followed by a new observation period. When an insurer has millions of customers, a small percentage improvement in results from doing this experimentation can translate to millions of dollars in bottom line contribution.

As you can see, it is considerably easier to implement a test and control strategy if the rules are externalized in a rulebase rather than inside transaction processing systems. Externalized rulebases facilitate the creation of multiple-hypothesis test rules.

Order configuration and validation

In many industries, the ordering of a finished good involves the specification of many different options. In a customer-oriented manufacturer, each customer's order could conceivably be different from any other customer's order. The manufacturer wants to ensure before starting manufacturing that the customer's order is valid.

For example, in ordering a luxury automobile, there may be tens of different options ranging from air conditioning and traction control to trailer hitches and sun roofs. Not all options go together. For example, if customers order trailer hitches, they had better also order the 4.2 liter engine. But if they order the 4.2 liter engine, there is not enough room in the engine compartment for the turbo charger. Similarly, the customer may want delivery in three weeks but the desired paint color is out of stock. It would be nice to let the customer know before manufacturing that the color is unavailable and give the customer the choice between waiting and choosing another color.

Advisor rulebases can be used to verify the configurations of complex orders with multiple options and supplier delivery constraints. The rulebase can be run every day against all outstanding orders to check for alerts (caused by either the supplier situation changing or because the manufacturer's engineering department defines additional relationships between options).

Customized and personalized content delivery

Most people today are familiar with the customized content Web pages they can design using services like My Yahoo!. Using Advisor, this concept can be extended to create rules-driven dynamic content delivery for sophisticated end user applications.

For example, a securities broker/dealer has many institutional investors, each of which has a very unique portfolio. The broker's salespersons want to deliver the right information to their customer's desktop and so encourage the customer to perform new trades with the broker. The Advisor rules engine can examine each customer's portfolio and preferences and assemble investment information elements onto an HTML page. This could include notification to the customer of relevant research reports, notification of upcoming corporate actions such as bond redemptions or tenders, suggestions for hedging strategies, tracking price changes and notifying the customer of thresholds being reached.

Through the Advisor rules engine, as new ideas for delivering value are hatched by the broker's sales force, these ideas can be simply delivered by adding or changing the Advisor rules—again, without having to go in and change or invent traditional application programs.

Another example comes from the mutual fund industry. Customers who call in to the service center can be presented with a tailored voice response unit based on the customer's particular portfolio. The Advisor rulebase can decide to offer new available emerging market fund voice menu options for the active trader versus quotation menu options for the buy-and-hold investor.

DEFINING WHAT BUSINESS RULES ARE

We have spent some time in this document describing business scenarios where rule technology is appropriate. To understand how Advisor solves these problems, we must first clearly define what is meant by the term "business rule".

A business rule is a statement that describes how a business is organized, how a business decision is made or how a business process works. Some typical examples are:

```
"if the order exceeds $10,000 and if the customer has been timely in his last
 3 payments, discount the order by 10%"
"any purchase of equipment above $3,000 must be approved by a manager in the
 finance department"
"all products that include cryptographic modules cannot be exported".
"if any manager meets all his quarterly objectives, they will get a 10%
 bonus"
```

Most business rules associate a condition to an action and naturally take the form of an "if … then …" statement. Sometime the "if … then …" form is not the most natural one but the rule can be easily rephrased in this form. For example, some of the rules above can be rephrased as:

```
"if a purchase of equipment exceeds $3,000 then it must be approved by the
 finance department"
"if a product includes cryptographic modules then it is not exportable"
```

STATIC VS. DYNAMIC RULES

The rules that we just gave are "dynamic" rules, also called "production" rules. They apply to business objects (orders, products, employees, etc.). They link conditions on the state of these objects (price, presence of crypto modules, etc.) to actions that modify the objects (set the discount, mark as non exportable).

But rules may also be used to describe the structure of the objects. For example:

"an employee has one manager"
"an order contains one or more order items"

These rules are rather static by nature. They express structural constraints that must hold at all times, whereas the rules given previously associate conditions to actions and define how the system will evolve over time.

Advisor can also capture these structural rules. Actually, the first thing that you do when creating an Advisor application is describe your objects, and Advisor lets you do this in a very natural fashion. Advisor makes a strong distinction between static and dynamic rules. The static rules are verified at compile time, whereas the dynamic rules are processed at run-time.

Sometimes, it is difficult to classify a rule as static or dynamic. For example, a rule like:

"a product team cannot have more than 10 engineers"

looks very similar to the other static rules that we gave.

On the other hand, it is usually impossible to verify whether this rule applies or not without actually running the system. In this case, the verification is impossible if we have rules that modify the assignments of engineers to product teams, or if the data (engineers, teams) is retrieved from a database at runtime.

Then, this rule must be handled as a dynamic rule, and rephrased as:

```
"if a product team has more than 10 engineers then signal this as an anomaly"
```

EXPRESSING RULES

Ideally, you would probably like to write rules in plain English and pass them to a system that can "understand" them and use them to drive automated business processes.

Unfortunately, today's technology does not yet permit this, and English allows for ambiguous statements. Business rules must be translated into "software rules" that the rule processor can compile and execute. If the translation is straightforward, the business rules approach will bring you great benefits because the business process will be easy to change. On the other hand, if the translation is too complex, the advantage of business rules over more traditional approaches (3GL, 4GL) will not be obvious.

This "translation" issue has been central in the design of Advisor. Every effort has been made to "close the gap" between the business rules as they would be naturally expressed and their representation in Advisor:

- The rules are expressed in a natural, English-like language. This language makes it easy for rule developers to write the rules, and for business analysts to read the rules and validate them.
- The Advisor Structured Rule Language (SRL) incorporates powerful operators. In general, one business rule becomes one software rule, even if it implicitly contains iterations or complex tests.
- The rules are relatively independent from each other and controlled by simple mechanisms.

A SIMPLE RULE

To illustrate the "translation" issue, let us consider a simple typical business rule and compare how it would be translated to various programming or rules languages. In plain English, the rule will read:

If the salary of an employee exceeds the salary of his manager then mark this employee as having a special status.

In Advisor, this rule can be expressed in a very concise and simple way, with rather classical operators (except the 'any' keyword).

```
emp is any employee.
if emp.salary > emp.manager.salary then emp.status = special.
```

Advisor will also accept a more verbose, English-like formulation and the rule could be rewritten as:

```
if the salary of emp > the salary of the manager of emp
then the status of emp is special.
```

A business analyst should be able to understand and criticize this, even write it with a minimal amount of training.

INVOKING RULES

The business rule should be fired whenever any of the following events occur:

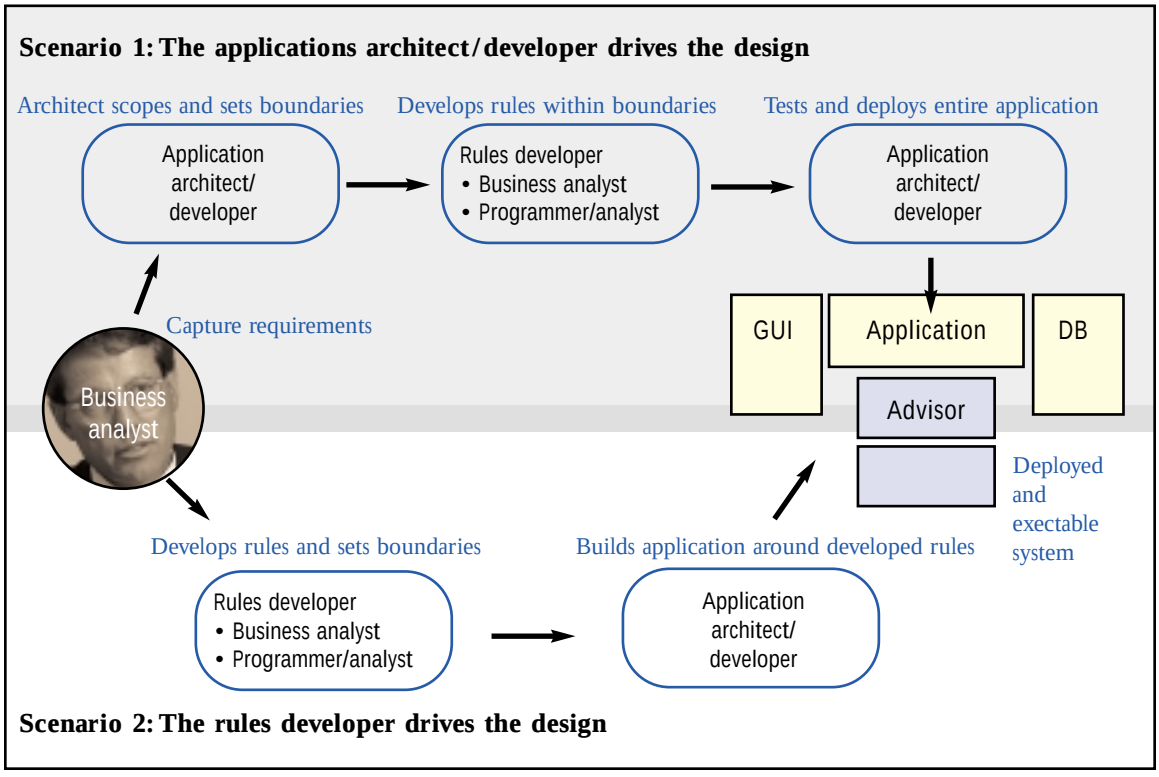
- The employee's salary is raised.
- The manager's salary is cut down.
- The employee changes manager and his new manager has a lower salary than his old manager.

And in fact, the Advisor rule engine reacts to any of the above three events. The rules developer does not have to worry about when the rule is fired or special control logic to cause the rule to fire. This behavior is one of the key value points of Advisor because in traditional procedural programming, explicit control must be introduced to cause the business rule to execute.

TWO DEVELOPMENT SCENARIOS

Rule-based applications are generally constructed by following one of two development scenarios as illustrated by the figure below.

SCENARIOS FOR BUILDING A RULES-BASED APPLICATION



In scenario one (1), an application architect owns the entire application design and defines at which points the rules service should be invoked. This architect defines all the objects, the persistence mechanisms, and the master control flow. The rules developer is then given a set of objects to write rules against and the specification of when the rules service will get those objects. Once the rulebase is debugged, the rulebase (and rule service) is handed back to the application architect for systems integration and overall application testing.

In scenario two (2) the rules developer defines the application's objects and the rules that are written against those objects. The rulebase is written and tested against these "for testing only" objects. Then, the rules developer defines how the overall application architecture should invoke the rules. At this point, it becomes the job of the application architect to implement the "real" objects which were simulated by the rules developer.

For simplicity, we'll start with the scenario in which the rulebase is built based on an existing object model.

IMPORTING YOUR CLASSES, OBJECTS OR DATA

Rules do not exist in a vacuum; they apply to objects. Rule based products have been on the market for many years, but they did not reach mainstream acceptance and they did not always deliver their promises. This mild success can be partially explained by the difficulty of integrating these products with mainstream data and especially with standard objects.

Advisor takes a very innovative approach for using objects. It is the first system that lets you write rules that apply directly to your objects. This is made possible due to:

- **A Classical Object Model**. You can write rules that apply to individual objects as well as rules that apply generically to all the instances of a class. Advisor has a very classical interpretation of what a class or an object is. It supports simple, standard OO concepts like properties and single inheritance. So, there is a very direct and clear mapping between the Advisor model and modern OO component models like Java Beans.
- **Powerful mapping technology**. Advisor takes full advantage of the Java Beans component Object Model and of Java's introspection capabilities. The development tool includes a Java object model import wizard that lets you select the Java classes/interfaces and Beans components that the rules will access. Once the components have been selected through the Wizard, the Advisor compiler analyzes their structure (their properties, their methods) and is ready to compile rules that deal with these components (test or modify their properties, invoke their methods).

Advisor is designed to work on the objects already defined in the application which Advisor is "plugged into." For example, if an application is structured as a set of CORBA services, Advisor can interact with those service's exposed objects, or if the application is structured as a set of cooperating COM/ActiveX components, Advisor interacts with the COM/ActiveX

exposed objects. The most natural way to interact is when the application is structured as a set of Java objects. In all cases, Advisor sees the world through Java classes and Advisor has built-in adapters to map COM/ActiveX and CORBA components as Java classes.

This tight integration with Java objects opens the door to a wealth of services, including access to remote objects through third party middleware like Microsoft Transaction Server, IBM MQSeries, Iona's Orbix or Inprise's VisiBroker.

Not only can Advisor rules be written against true objects such as Java, CORBA or COM/ActiveX but Advisor can have rules written against database rows mapped as "data-only" objects. This is particularly handy for many applications where the information needed to use in a rule is contained directly in an SQL database

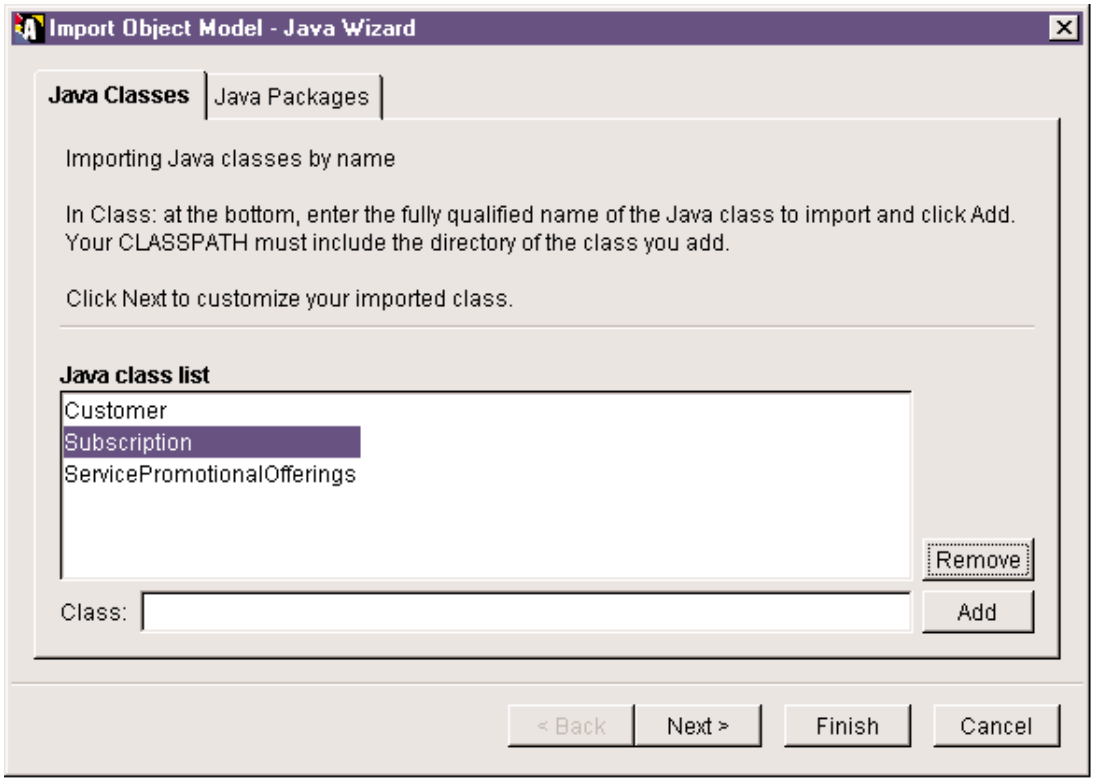
Although Advisor has been designed to work directly on Java objects, it does not require that you define your objects as Java objects coming from application into which Advisor is "plugged." Advisor also gives you the possibility to define "pure Advisor" classes and objects that do not map to any existing Java counterpart. This feature is very useful for prototyping but also for designing applications that do not need to be tightly integrated to an existing Java infrastructure. This feature also allows for doing symbolic reasoning or rules upon derived data or concepts that do not exist in the over-arching application. Thus, applications don't need to be changed (such as adding a new member variable) in order to support some intermediate rule result of the business rule problem.

The Advisor Builder lets you define classes and objects through some simple editors. Then, you can write rules that apply to these classes and objects. In this mode, you do not need any Java expertise to develop a rulebased application.

Here, Advisor gives you maximum flexibility. You can mix and match between external (Java, COM, CORBA) and Advisor classes and objects. For example, you can prototype your application with "pure Advisor" classes, and replace these classes by Java classes later in your development process. This is a simple matter of deleting the Advisor class definitions and importing the equivalent Java classes with the wizard. You do not need to change the rules.

IMPORTING JAVA CLASSES/INTERFACES: MAPPING JAVA OBJECTS

When you are ready to attach your Advisor rule project to an external Java application you just need to launch the "Import Java Class" wizard from Advisor Builder. This wizard lets you enter the names of the imported Java classes. Once you have specified the classes, you can customize the way the class will be "viewed" by the rules. You can select which properties and methods will be accessible from the rules and you can also give them a different name on the rules side. All this is done through a very simple point and click interface:



Advisor analyzes the class API according to the Java Beans conventions:

- A couple of setXxx/getXxx accessors is mapped to an Advisor property called xxx (that you can rename through the Wizard).
- A getXxx accessor without any setXxx counterpart is mapped to a read-only xxx property.
- If the class defines an "addPropertyChangeListener" method, Advisor will flag the properties as being bound properties.
- "Indexed" properties are fully supported. If the Java accessors take an additional index argument, the Advisor property will be "indexed" and you will be able to access the indexed values with the usual prop[index] notation.

From the rules, you can manipulate these properties like normal Advisor properties. Moreover, if the properties are bound (they should be in well designed Bean components), the Advisor engine will reevaluate its rules whenever the property is changed, whether the change originates from Advisor or from another component.

From the rules, you can also:

- Invoke public methods on the Java objects (with or without arguments).
- Invoke static methods of the Java class.
- Create new instances of the Java class (eventually passing arguments to the constructor).
- Use the constants defined by the Java class (public static final fields).

In applications that embed the Advisor rule engine, you will be writing rules that deal with Java objects that have been created from Java rather than from Advisor. In this case, you need to inform (called "mapping") the Advisor engine that the Java objects exist and need to be

processed by the rules. The NdRuleAgent API provides two calls to control which Java objects will be seen by the rule engine:

```
NdObject mapExternalObject(Object obj)
void unmapExternalObject(Object obj)
```

But often, you do not even need to introduce these calls in your Java code. If you get the objects through a static method of a Java class, the mapping will be automatically done by Advisor. If you get the objects by applying methods to an "object factory," you just need to map the "object factory," the related objects will be mapped automatically by Advisor.

This powerful yet simple mapping scheme allows you to apply rules to your Java objects without having to modify your existing Java source code. It promotes the Component approach supported by Java Beans. You can package the low level logic as Java Bean components and write the high level logic as rules.

Then you get the best of both worlds. You can use the full power of Java at the low level and still write your business logic as rules that are easy to understand and easy to change.

IMPORTING CORBA INTERFACES; MAPPING CORBA OBJECTS

Enterprise systems are moving towards multi-tier architectures in which the objects are distributed over several servers which are accessed through an Object-Oriented middleware. In this context, it seems natural to introduce distributed rule servers that operate on remote objects and eventually interact with each other.

Advisor allows you to implement distributed rules servers by allowing you to write rules that operate on remote objects. This opens the door to very interesting architectures where distributed rule servers can cooperate with traditional CORBA servers but also cooperate with other rule servers by acting on shared remote objects.

The access to remote objects comes as a natural extension of the Java mapping scheme. To go beyond "local" objects, Advisor allows you to import Java "interfaces" as Advisor "classes." Then, you can map the CORBA interfaces generated by CORBA/Java middleware products (OrbixWeb, VisiBroker) and write rules about remote objects the same way you would write rules about local objects.

The get/set accessors of the CORBA interfaces are imported as attributes of the Advisor class. You can manipulate these attributes like normal attributes from the rules. You can also invoke the other methods (non accessor) exposed by the CORBA interfaces.

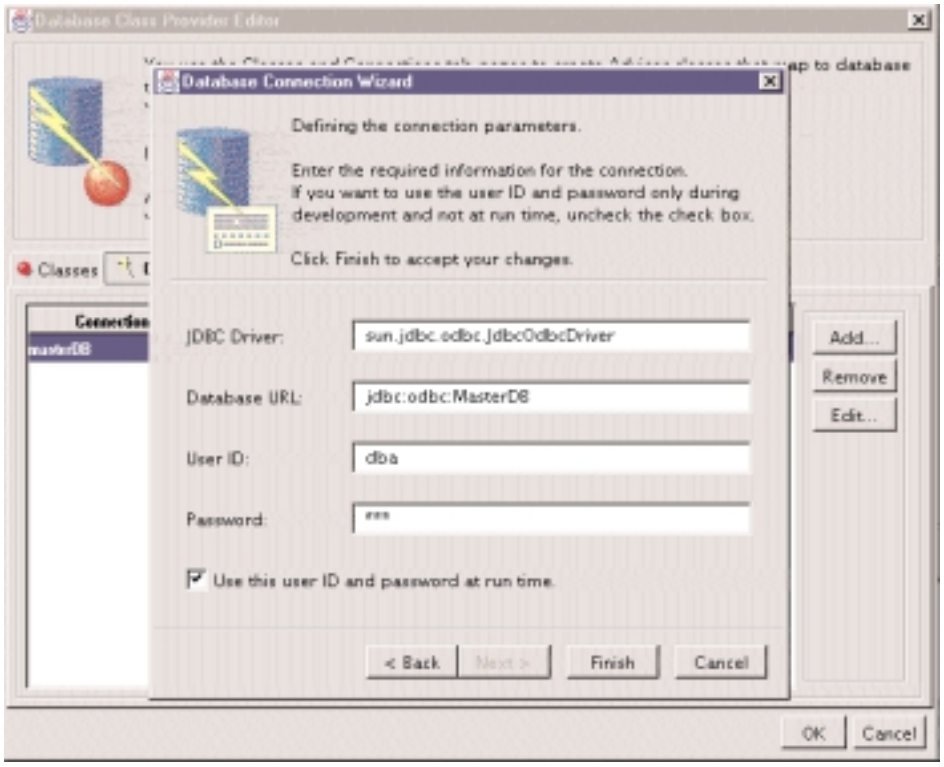
Advisor keeps a memory cache of the values of the attributes. It only invokes the get accessor the first time it needs the attribute or after it has been notified (through a `PropertyChange` listener) that the attribute changed. Thanks to this caching mechanism, the engine can efficiently compute the state of the rules, without triggering too many roundtrip calls to the remote server.

IMPORTING SQL TABLES; MAPPING SQL TABLE ROWS

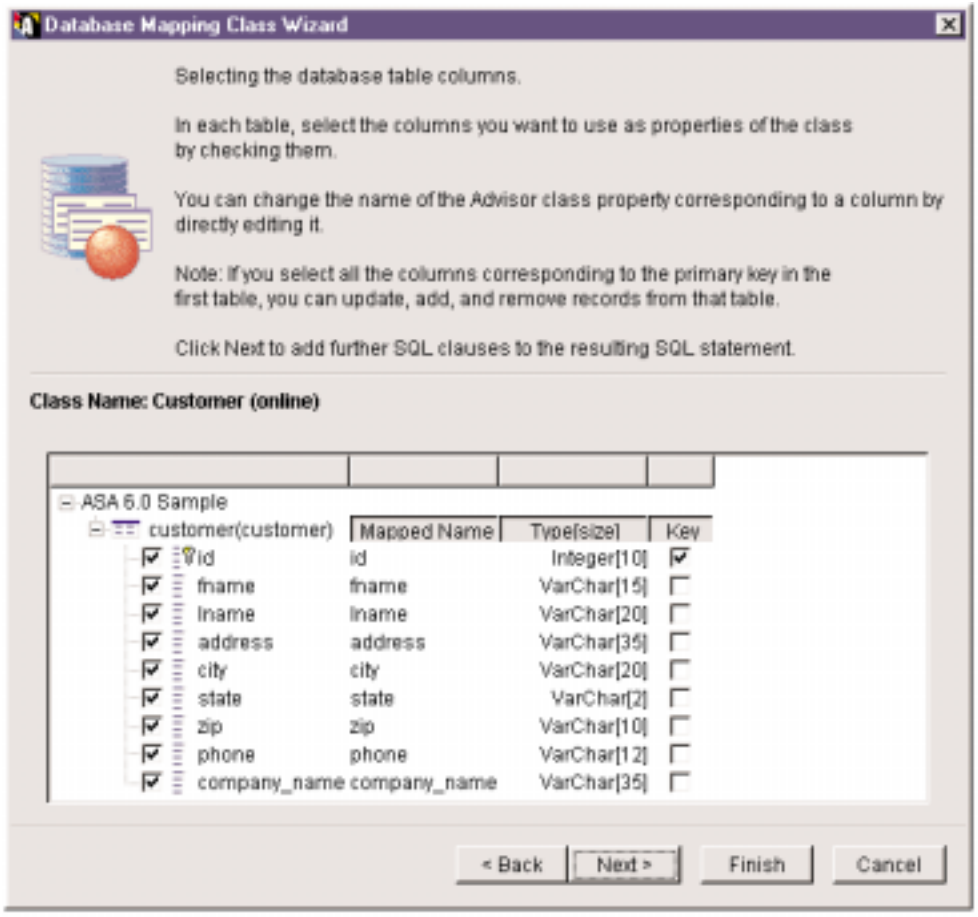
But most of today's corporate data has not yet been packaged as object servers and resides in databases, usually relational ones that can be accessed with the SQL language.

Advisor allows you to define classes that map to SQL queries result sets. The mapping process is entirely guided by a point and click wizard and you do not need to write any intermediate Java code to access the database.

First, the "Import Database Objects" wizard lets you define the parameters that will allow Advisor to connect itself to the database server. You specify these parameters through the following wizard page:



Once the connection has been defined, the wizard connects itself to the database and reads the database schema. Then, you can select which tables you want to map as Advisor classes and you can refine the association between database columns and Advisor properties in the following wizard page:



Advisor gives you access to the full power of the SQL language and for example, allows you to:

- Join two or more tables to define a single class on the Advisor side.
- Select a subset of the table's record by specifying a possibly parameterized WHERE clause in the mapping.
- Control how null values (or any specific value) will be mapped to Advisors special values (unknown, unavailable, null).

Once the association has been defined, you can write rules about the Advisor class that maps to your table's rows. Advisor does not place any restrictions on these rules and you can use the full power of Advisor's pattern matcher on your database objects.

At runtime, your application will need to retrieve the database objects from the database, and may eventually write back the objects that have been modified. It may also need to insert or delete records to reflect the objects that have been created or deleted during the session.

All these operations can be triggered directly from the rule without any Java programming. The class that you defined through the wizard contains methods like `fetchAllInstances()` or `insert()` that allow you to perform all these operations.

Typically, the retrieval of objects will be performed in a "whenever … is needed" event rule, for example:

```
event rule retrieve_products is
whenever the selectedProduct of an order is needed
do {
     prodQueryParams.product_type = product_type.
     ProdQuery.setSelectParams(prodQueryParams).
     ProdQuery.fetchAllInstances().
}
```

where `ProdQuery` is the class that has been mapped to a database table.

CREATING CLASSES AND OBJECTS

Advisor also supports the creation of "pure Advisor" objects and classes which do not map to external objects. In some instances, the rulebase will require object definitions or extensions to external objects solely for rule processing. For this requirement, Advisor supports the creation of classes and objects. This section will give us an overview of the Advisor Object model.

SUPPORT FOR NOVICE AND POWER USERS

Advisor Builder, the development environment of Advisor, supports both users with limited technical expertise as well as power users. The editing environment can be configured in two modes:

- **The "editors" mode**. In this mode, the classes, objects, and rules are edited through specialized editors. The editor guides the user and, for example, allows him to choose the parent of a class or the type of an attribute by picking the name of an existing class (or a basic type) in a choice box. This mode requires very little knowledge of the underlying rule language. You do not need to know the high level constructs for class declarations or the punctuation rules. You just need to know about simple things like reserved words or basic expression syntax.
- **The "source" mode**. In this mode, you edit the source of your rulebases directly through a standard text editor. This mode requires a good knowledge of the rule language but allows a power user to make important modifications very quickly. The rule language is quite simple and easy to learn. So, this mode is not really reserved to an elite of programmers.

You can toggle between the two modes by selecting the appropriate command ("Show Editors" or "Show Source") in the "View" menu.

CREATING A PROJECT

Before creating classes and objects, you must create a "project." This must also be done before mapping in external objects. The project will keep track of all the elements that contribute to your rulebased application:

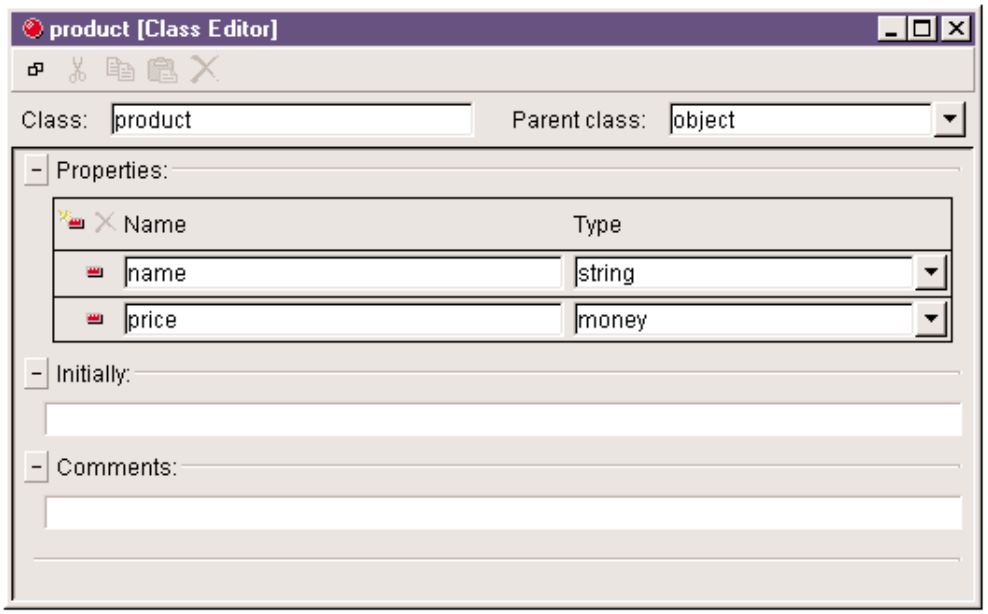
- **Rulebases**. A rulebase is a source file containing class definitions, object definitions, rule sets, rules,…, expressed in the ADVISOR rule language.
- **Mapping files**. These files describe mappings to Java classes/interfaces, COM types, CORBA interfaces, SQL tables, etc. These files are maintained by the Builder and normally, you should not edit them manually.
- **Rule flow file**. This file manages the layout definition of the ruleflow that you create

To create a project, you just need to invoke the "New Project" command from the File menu. This will create a new project with an empty rulebase called "untitled.rb". All the project related operations (loading, saving, adding and removing components) are done through very simple menu functions in the builder.

CREATING AN ADVISOR CLASS

In most cases, you import your class definitions from external object models. However, if you are developing an Advisor project before you have an external model ready or you need to subclass an imported external class you use the Advisor Class Editor.

If you chose the "editors" mode, you can create a class by selecting the "New Class" option in the "Project" menu or the corresponding toolbar icon. Then you get the following editor:



To define the class, you need to enter its name ("product"), choose its parent class and define a list of properties. Advisor supports a single inheritance model, like Java. The properties are inherited by subclasses.

The "initially" box allows you to specify initialization statements that will be executed every

time an instance is created. This initializer plays more or less the same role as a constructor in Java (and the initializers are chained like Java constructors).

The "comments" box lets you enter descriptive comments about your class.
If you choose the "source" mode, you will edit the following source code fragment:

```
a product is an object with {
    a name: a string,
    a price: a real
}
initially {
    // some statements ...
}
```

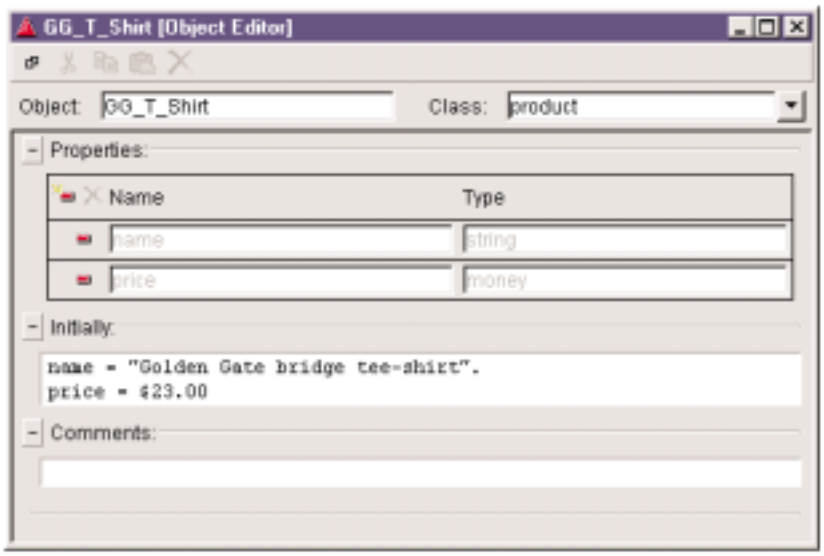
The syntax is deliberately English-like. The punctuation rules are easy to learn and flexible (e.g. the compiler will not complain if the last property declaration is followed by a comma).

Advisor classes are meant to represent high level classes of "business" objects, not low level "implementation" classes. To keep the description simple and accessible to business analysts, a number of "programmer oriented" features supported by Java have been deliberately omitted here (methods, qualifier keywords like public, private, static, abstract, etc.). This is not really a limiting factor because you can implement your classes as Java classes and import their definition into Advisor if you need the full power of a complete OO programming language.

CREATING A NAMED ADVISOR OBJECT

In most cases, you will be mapping in objects from your application at runtime in order to provide the Advisor rule engine with object(s) to work on. However, if you need objects local only to Advisor or are developing your rulebase in advance of attaching it to an external application, use the Advisor Object Editor.

Once you have defined classes (either through importing their definition or explicitly creating in Advisor), you can define objects that belong to these classes. This can be done through an object editor that resembles the class editor:



The source code equivalent is:

```
GG_T_Shirt is a product
initially {
    name = “Golden Gate bridge tee-shirt”.
    Price = 23.
}
```

The editor displays the list of properties inherited from the “product” class. It also allows you to add properties that will be specific to the object. This “object-level specialization” is very practical as it allows you to define specialized global objects without forcing you to introduce a new class.

OBJECTS AND VALUES

Advisor provides a set of built-in types:

boolean
integer
real
string
date
time
timestamp
duration
money

And you can define your own enumerated types, for example:

```
a size is one of { small, medium, large, extra_large }
```

Advisor makes a strong distinction between “types” and “classes.” Classes describe mutable objects whereas types describe immutable “values.” This distinction is deeply related to the distinction between “objects” and “non objects” in the OMG model. It makes the language simpler and avoids some programming pitfalls that even a modern OO language like Java did not eliminate, for example:

- You can compare all values, including string and dates with the built-in comparison operators (<, <= , >, >=, =, <>). There is no need for a special “equals” method like in Java.
- There is no need to introduce “convenience classes” for built-in types (like the Boolean, Integer, Long, … classes in Java).

Once the classes and objects have been defined, you can start writing rules.

STRUCTURING YOUR APPLICATION LOGIC WITH RULESETS AND RULE FLOWS

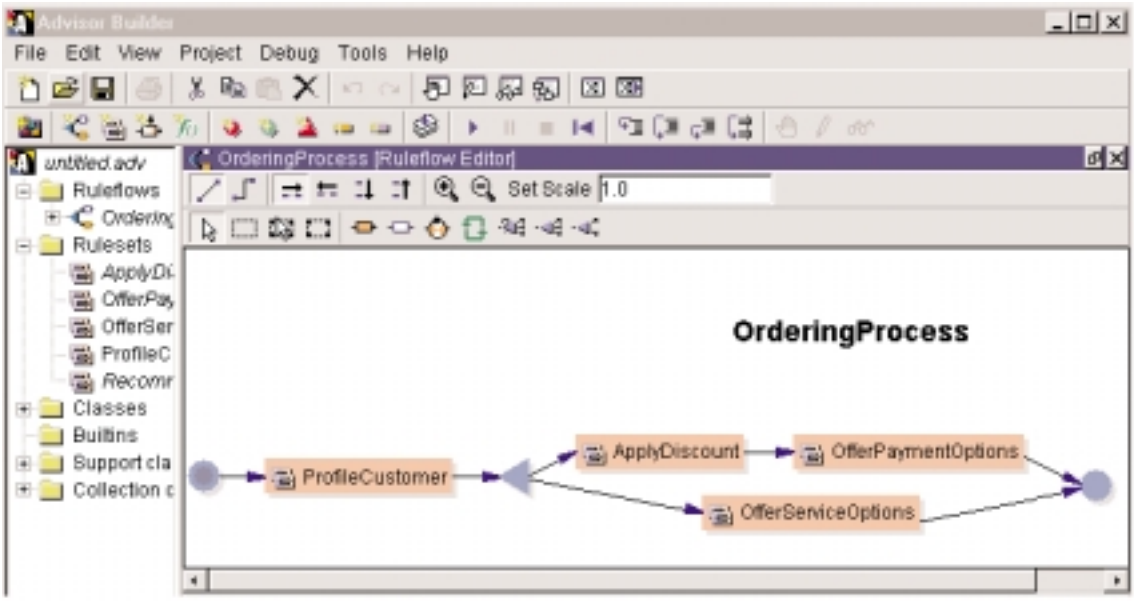
Most processes can be decomposed into tasks with different sets of rules for each task. For example, a typical e-commerce process will have tasks like:

- Customer profiling.
- Product selection.
- Pricing and discount computation.
- Payment conditions, etc.

Advisor provides two powerful mechanisms to let you structure your application logic in a way that closely matches this high level decomposition.

First, you can define rulesets to group the rules by task. For example, you would group the rules that analyze customer profiles into a “customer profiling” ruleset, the rules that select products based on the needs expressed by the customer into a “product selection” ruleset, etc.

Then, you can set up the high level flow between tasks with a powerful graphical “flow editor”:



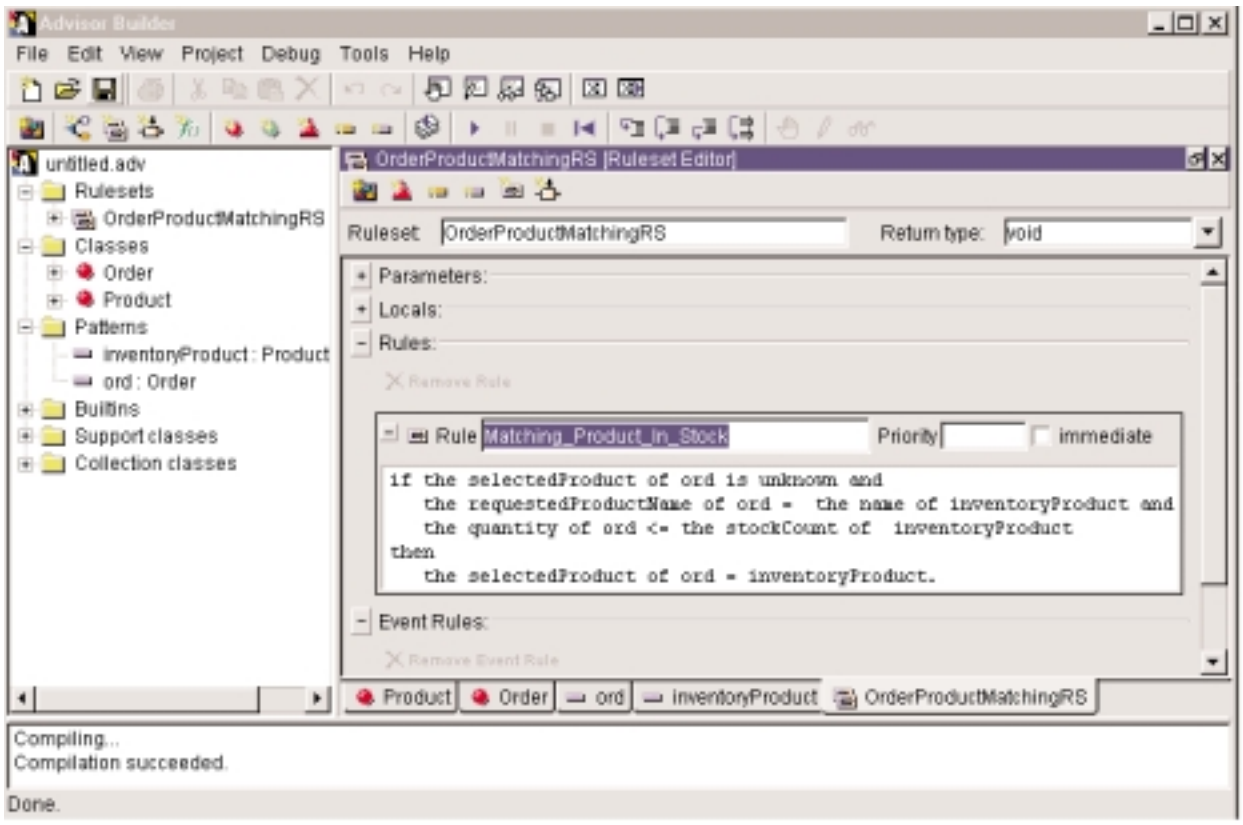
Through simple point and click manipulations, you can chain several tasks, insert a decision node, set up an iteration, etc. Also, a double click on a task node brings you directly to the ruleset editor where you can review and modify the rules that implement the task.

At run time, the process will follow the flow diagram. To execute a task node, Advisor will evaluate the rules of the corresponding ruleset. Once all the eligible rules of the ruleset are evaluated, Advisor will move to the following task in the graph.

The rule flow diagram may have several "input" nodes, and you can connect these input nodes to "external events" in the Java/COM/CORBA world. This way, you can very easily tie your rulebased business processes to an event driven application infrastructure.

CREATING A SIMPLE RULE

The easiest way to create a rule is to use the Advisor ruleset editor. This ruleset editor has structured fields for entering the rule name, priority (if any) and rule body.



The English version of the rule is:

```
"If the quantity of product of the order is available in stock, then accept
the order".
```

The rule can also be entered in source form as:

```
rule matching_product_in_stock is
if the selectedProduct of ord is unknown and
   the requestedProductName of ord =  the name of inventoryProduct and
   the quantity of ord <= the stockCount of  inventoryProduct
then
   the selectedProduct of ord = inventoryProduct.
```

The ‘ord’ and ‘prod’ variables that appear in the rule are special variables called “patterns” they have been defined as:

```
ord              is any order.
inventoryProduct is any product.
```

As these definitions suggest, the ‘matching_product_in_stock’ rule applies generically to all the order objects and product objects in the system, not just to a specific order or a specific product. In English, we could phrase this rule as:

```
“If the quantity of product of the order is available in stock, then accept
the order”.
```

The Advisor engine will monitor the order and product objects in the system. When an order/product combination will satisfy the rule’s condition, the rule engine will “schedule” the rule for the combination. Then, the rule engine will fire the rule and the product will be selected.

With a single rule, the benefits of this approach are not very obvious. But we can also write rules that will select replacement products based on various “good match” criteria when the product is not in stock. Suppose we had an additional rule:

```
“If the quantity of product of the order is not in stock, then fill the
order with a product of the same type which is in stock. The substituted
product’s price must be within 10% of the original product ordered”.
```

This rule could be written as follows:

```
rule matching_product_out_of_stock is
if the selectedProduct of ord is unknown and
   the productType of ord = the type of inventoryProduct and
   the quantity of ord <= stockCount of inventoryProduct and
   1.1 * ord.productPrice > the price of inventoryProduct
then
   the selectedProduct of ord = inventoryProduct.
```

Then, you just need to add rules or modify rules if your order/product matching rules change.

PATTERNS AND PATTERN CONSTRAINTS

If we look at our two product matching rules, we will notice that they share some tests:

```
the selectedProduct of ord is unknown
the quantity of ord <= the stockCount of inventoryProduct
```

These conditions are not really specific to these two rules; they are likely to appear in all the rules that will drive the product selection process. In all these rules:

- We are only interested in orders that have an unknown selected product.
- We will only consider products for which we have sufficient stock.

Advisor lets you associate these conditions with the patterns so that you do not have to repeat them in every rule. Then, the product selection patterns and rules can be rewritten as:

```
ordWithoutAssignedProduct is any order
     such that selectedProduct is unknown.

inStockProduct is any product such that
     ordWithoutAssignedProduct.quantity <= stockCount.

rule matching_product_in_stock is
if the requestedProductName of ordWithoutAssignedProduct =
   the name of inStockProduct
then
   the selectedProduct of ordWithoutAssignedProduct = inStockProduct.

rule matching_product_out_of_stock is
if the productType of ordWithoutAssignedProduct =
        the type of inStockProduct and
   1.1 * ordWithoutAssignedproduct.product_price >
        the price of inStockProduct
then
   selectedProduct of ordWithoutAssignedProduct = inStockProduct.
```

Here, the patterns have been "constrained." The rules are more concise, more readable and easier to modify.

OBTAINING DATA

The Advisor engine works by matching objects with rules. If some objects or some combinations of objects satisfy the conditions of a rule, the rule will be scheduled. Then, the engine will execute the rule (unless its conditions are no longer satisfied).

This mode of operation assumes that the objects tested by the rules are present in the system and that the values of their attributes have been set. If the information is insufficient, none of the rules will apply and the system will not do anything.

This approach works well when the information is "pushed" through the system, for example in an alarm detection system, or a trading system. It does not work as well when the information needs to be obtained from the outside (a database, a user, another process).

In a typical order processing system, the orders will be pushed through the system but the product information will be retrieved from a database. The product database can be huge and we cannot usually afford to retrieve the entire database, create one object per product and push these objects through the rules. We need to selectively retrieve the product information that is relevant to the orders we are processing.

Advisor lets you write rules that will be triggered when some piece of information is missing. These rules are called "event rules" and are introduced by a special "whenever…is needed" syntax. For example, we could direct Advisor to retrieve product objects from the database when an order comes in by writing:

```
event rule retrieve_products is
whenever the selectedProduct of an order is needed
do {
     prodQueryParams.product_type = product_type.
     ProdQuery.setSelectParams(prodQueryParams).
     ProdQuery.fetchAllInstances().
}
```

The data types of Advisor are slightly more sophisticated that the data types of classical programming languages. Any variable or object attribute can either take a "normal" value or one of the following "special" values:

- **unknown**. This is the default value for any variable or attribute. It usually means that the value has not been set and that the engine did not try to obtain it.
- **unavailable**. This value usually means that the engine tried to obtain a value but that it could not get it, either because of an error occurred or because the user did not have the answer to a question, or any other reason.
- **null**. This value is used on object relationships, to indicate a void reference.

When a condition of a rule cannot be evaluated because a variable or an object property is "unknown," The Advisor engine considers the variable or property to be "needed." Then, it activates the corresponding "whenever … is needed" rules.

ASKING QUESTIONS

The "needed" keyword is especially useful for interactive applications like diagnostic systems or intelligent questionnaires. Advisor comes with a set of built-in "prompt" methods that you can use to obtain data from the user. For example:

```
whenever the running of an engine is needed
do running =
   promptBoolean("Do you hear the engine running?").
whenever the abnormal_noise of an engine is needed
do abnormal_noise =
   promptBoolean("Does the engine make an abnormal noise?").
```

If you have a diagnostic rule like:

```
if eng.running and eng.abnormal_noise then …
```

Advisor will first ask the user whether the engine runs. If the answer is positive, the "abnormal_noise" property will become "needed" and the second question will be asked, but if the answer to the first question is negative, the Advisor engine does not need the value of "abnormal_noise" to conclude that the condition is false. Then it will not ask the second question.

Here, Advisor gives you a lot of flexibility. If you want to bypass the needed mechanism and ask a question systematically, you can use the following rule:

```
if eng.abnormal_noise is unknown
then eng.abnormal_noise =
     promptBoolean("Does the engine make an abnormal noise?").
```

You can also ask the question from an initializer:

```
a FordExplorerEngine is an object with {...}
initially {
   abnormal_noise = promptBoolean(...).
}
```

You can use the prompt mechanism not just for asking questions but to control the retrieval of relatively expensive information only when such information is "needed." The prompt mechanism can be intercepted by the external application that invokes Advisor (in effect, the external application provides a prompt "handler" that Advisor calls whenever the prompt statement is executed). This allows the external application to query a user for the needed information or query a remote service or database.

CONTROLLING THE REEVALUATION OF RULES

Advisor provides you with powerful, yet simple mechanisms to control when rules will be executed. You can assign priorities to rules but you can also finely control the reevaluation of the rules. This is very important for rules that monitor fluctuating data. Let us consider the following example:

```
rule check_overheat is
if the temperature of a sensor > 200 then create an alarm ...
```

This rule is rather straightforward but it could be processed by the engine in two different ways. It could be executed:

- **only when** the temperature crosses the 200 degrees threshold.
- **every time** the temperature changes and remains over the threshold.

By default, Advisor uses the first mode. It schedules or reschedules a rule when the condition "as a whole" becomes true, not every time the individual pieces of data tested by the rule change. But you can also set up the rule as an event rule so that it gets reevaluated every time a particular piece of data changes. To achieve this, you just need to introduce a special "whenever … is changed" clause into the rule. For example:

```
event rule continuously_monitor_overheat is
whenever the temperature of a sensor is changed
do
     if the temperature of a sensor > 200 then …
```

MORE COMPLEX RULES

Advisor can handle complex rules. In the condition section of a rule, you can:

- Test variables.
- Test properties, either on a specific object, or generically on all the instances of a class (thanks to patterns).
- Use the standard comparison and arithmetic operators.
- Invoke Java methods to do arbitrary computations.
- Write expressions that involve any number of objects or patterns, from the same class or from different classes.
- Test whether a condition is satisfied by a certain number of objects, by all the objects in a class, etc.
- Combine conditions with logical operators.

In the action section of a rule, you can:

- Modify variables or properties of existing objects.
- Create new objects or delete existing objects.
- Invoke Java/COM/CORBA methods on objects.
- Invoke entire rulesets.
- Invoke Advisor functions.
- Execute the prompt statement to query for "needed" information.

You can also trigger actions on objects that do not satisfy the condition of the rule by adding an 'else' clause to the rule.

Every effort has been made to hide the sophistication behind simple, natural notations. For example, you can write rules like:

```
rule discontinued_product_type is
if every InventoryProduct such that type = ord.prod_type
   satisfies discontinued = true
then
   the selectedProduct of ord = unavailable.

rule refine_with_preferences is
if the preferredProductCharacteristics of ord is unknown and
   at least 3 product satisfy
      (type = ord.prod_type and
       Math.abs(price – ord.requestedProductPrice) < 10)
then {
   the preferredProductCharacteristics of ord = a preference initially {
      price_sensitivity = promptEnumerationItem(...).
      strict_size_match = promptBoolean(...).
   }
}
```

Here, the natural syntax of Advisor (i.e. "every … such that .. satisfies …") starts to show its real power.

MIXING RULES, FUNCTIONS, AND PROCEDURES

Although rules are usually more flexible and more intelligible than functions, there are cases where a computation or a sequence of operations fits the procedural model better than the declarative model of rules. This is particularly true when the business rules are best described as an algorithm or require support from algorithms such as bond yield calculations, mortgage payment calculations, or ITALK tax calculations.

Advisor also provides a rich set of procedural constructs so that you can create functions and procedures to capture the non-declarative part of your business logic. These constructs encompass:

- Functions with parameters and return values. Optional parameters are supported.
- Parameter overloading.
- Loops, conditional statements (if … then … else …, select … case …)
- Exception handling, etc.

Advisor gives you a high degree of flexibility in the way you can mix the declarative and procedural schemes. For example:

- A task of the rule flow diagram can be implemented either as a ruleset or as a function.
- A function may call a ruleset to perform part of its task.
- The actions of a rule may trigger the evaluation of a function, a procedural set of statements or another ruleset..
- The conditions of a rule may call a function.

Actually, Advisor treats rule sets, functions and procedures on an equal footing. Like functions, rule sets may take parameters and may return values, and the invocation syntax is the same. So, you can easily switch from a procedural implementation to a declarative one or vice versa.

DATE, TIME AND MONEY SUPPORT

Advisor provides a powerful and friendly support for date, time and money operations. You can express date and time values in localized formats and you can combine them and test them with standard operators. For example, you can write rules like:

```
rule qualifyAccount is
if the openingDate of an acct < today – 5 years and
   the balance on an acct > $10,000
then
   the qualityRating on an acct = good.

rule checkPersonnelReview is
if today > emp.lastReviewDate + 1 year
then {
   emp.needsReview = true.
   // schedule it for Monday next week.
   emp.nextReviewDate =
      yearWeekDay(today.year, today.yearWeek + 1, date.Monday)
```

Advisor's money type is designed to handle multiple currencies. The conversions between the currencies of the Euro area are built-in and they conform to the EC regulations (triangulation, rounding). For example, you can convert a Deutsche Mark amount into a Euro amount with:

```
euroAmount = 0.00 EUR + demAmount.
```

You can also set conversion rates for non Euro currencies. Then Advisor will automatically apply your conversion rates in the computations that mix currencies.

The money amounts are internally represented as Java BigDecimal values. So, there is no practical limitation on the ranges and decimal precision, and you can precisely control the rounding behavior.

LOCALIZATION

Advisor allows you to localize your projects. Date, time, money and numeric quantities will be parsed and formatted according to the locale's conventions (month names, decimal separator, etc.). The locale setting is recorded in the project file and will be preserved if you run the project on a platform that has been configured with a different default locale.

COLLECTIONS

Advisor provides basic collections classes like arrays and associations, as well as specialized constructs to iterate on collections. So, you can write rules like:

```
rule checkCreditHistory is
if the status of currentCreditAppl is unknown and
   at least 2 CreditApplication in cust.history satisfy status = rejected
then {
          the status of currentCreditAppl = rejected,
          cust.history.append(currentCreditAppl)
}

rule checkExpenses is
if every Expense in emp.expenses such that category = hotel satisfies
          amount / days < $200
then {
          for each Expense in emp.expenses such that category = hotel
          do approved = true.
}
```

COMPILING, TESTING AND DEBUGGING YOUR RULE-BASED APPLICATION

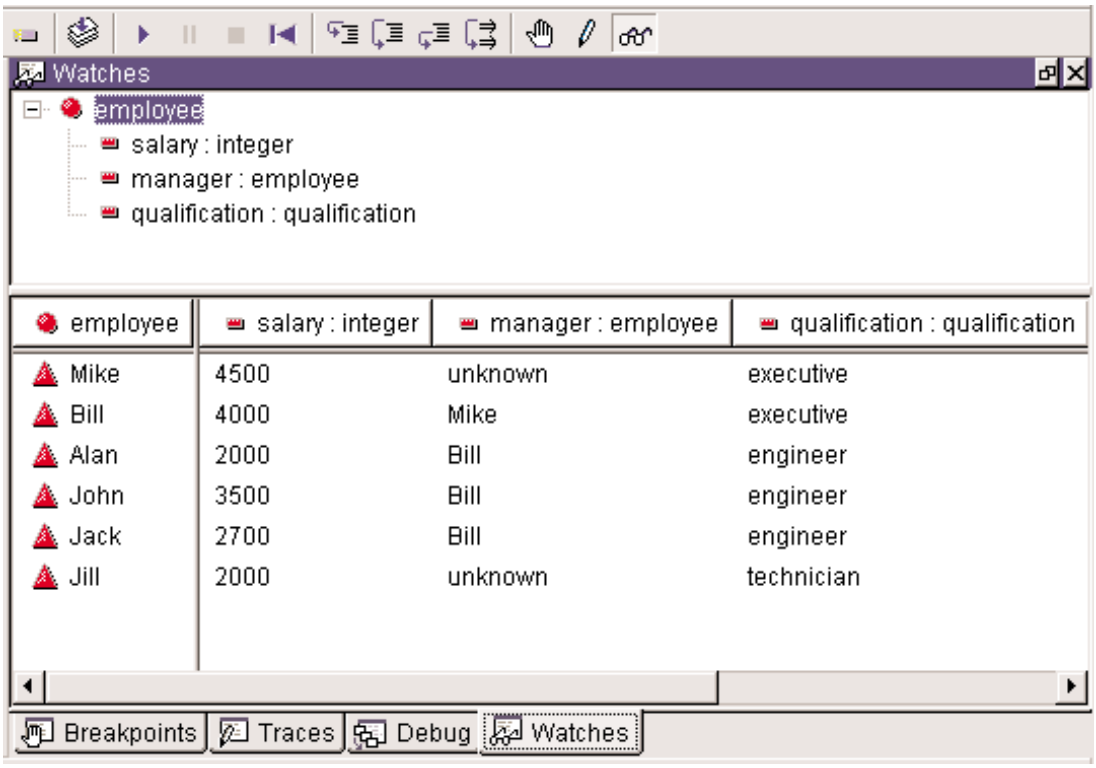
The editors and wizards of Advisor Builder are tightly integrated with the rule compiler. Advisor incrementally verifies the syntax of the conditions and actions of your rules.

When all your definitions and rules are in place, you can perform a global compilation. The rule compiler will perform all the necessary type verifications (the rule language is strongly typed) and will issue precise diagnostics in the "output" panel of the Builder. By clicking on a diagnostic line, you immediately bring into view the editor or the source section where a correction is needed.

Once you have successfully compiled your project, you can test your application. Advisor Builder incorporates a very powerful visual debugging environment featuring:

- **Step by step execution**. You can execute a session rule by rule or statement by statement. You can also interrupt a session at any time.
- **Breakpoints** You can set breakpoints on various conditions: execution of a rule, modification of an attribute, creation or deletion of an object. The object and attribute break points can be set either on a specific instance or globally for all the instances of a class.
- **Traces**. The events that occur during a session (questions, breakpoints, print statements) are logged into an "output" panel that you can save to a file or print. You can set additional traces on rules, properties or objects to get more debugging information without disturbing the execution of the session.
- **Visual trace**. You can observe a visual tree network of the execution-time relationships between rulesets, rules, functions, and ruleflow tasks. As execution procedes, the nodes on the tree graph change color to show an Advisor execution entity starting, leaving, or for rules, scheduled and unscheduled.

- **Watchpoints**. You can monitor the values of selected variables or objects, and explore object graphs by following the attributes that reference other objects. You can also monitor the attributes of all the instances of a class in a table.
- **Agenda**. You can view the set of rules that have been scheduled by the rule engine, as well as the list of "needed" properties.



DEPLOYING A RULE PROJECT

DEPLOYMENT SCENARIOS

The Advisor rule engine is a very versatile component. It is a 100% Pure Java component based on JavaSoft's JDK and Microsoft's SDK, which make it suitable for a variety of deployment scenarios. The term "rule agent" is used to denote a runtime context between a client of Advisor and a running instance of the rule engine. An Advisor rulebase can be executed:

- **On a server**. The rule engine and the rule project are loaded on a server. The Advisor run-time is thread-safe and you can serve several clients with a single instance of the rule engine on a server. Advisor servers can be configured for various server platforms—CORBA, ActiveX Containers such as Microsoft Transaction Server, messaging servers such as IBM MQSeries, TibCo or Active Software, and Web Servers through servlets or Microsoft Active Server pages.

- **As a standalone application**. The Advisor engine can be embedded in a standalone Java application or in a standalone ActiveX application (such as Visual Basic). In this configuration, the Advisor runtime must be installed on the client platform.
- **As a distributed application** that uses two or more of the above scenarios.

Your choice will depend on the characteristics and requirements of your application.

The standalone (or desktop) approach is useful if the application primarily accesses local data stored on the client workstation and/or a remote database server. The "standalone" solution requires a simple setup on the client workstation (which could be handled automatically through "push" technologies). The classical 2-tier client/server architecture is fully supported by Advisor applications through JDBC.

Using Advisor in a server architecture is the best choice for heavy-duty applications that access your corporate databases, message queues or object services. Advisor rules services are fully compatible with advanced 3 tier or N-tier architectures. The Advisor engine will be a component of an application server that the client workstations (or other servers) will access through your corporate middleware (COM-based, CORBA-based, message-based or other). The Advisor runtime package is thread-safe and you can implement multi-threaded servers. Here, you have two options:

- If the rules and most objects are shared by all the clients, the server can be set up with a single rule agent that serves multiple clients.
- If the clients need to load specific rules, or if the objects cannot be shared across clients, the server can create a separate rule agent for each client. A single server process can manage several rule agents connected to several client. The Advisor engine is thread-safe and makes it possible to implement the classical solution in which each rule agent is assigned to a particular thread within the server process.

SERVER ARCHITECTURES

The Advisor engine can be deployed in any of a large number of possible server architectures. The techniques used to deploy Advisor engines in a server depend on both the type of service that is requested, as well as the characteristics of the chosen server architecture.

There are three main styles of rule services:

Monitoring services: Your service will be started when the server is started. It will be told to monitor a subset of your environment for changes, and to apply the rules that are relevant whenever changes are detected. In general, there will be no expected end-user interaction for the purpose of the execution of the business rules. An example of this style of rules service would be a rule project that monitors changes in a trading portfolio, calculating the current risk level, and then when the risk level exceeds a threshold, notifies the trader of some appropriate hedging strategy. Another example is a system monitoring service which alerts a management console whenever devices attached to the network are experiencing conditions worthy of notice (e.g. excessive retries, long queues, etc.)

Non-interactive request-response: Your service will be waiting for requests to be submitted, will take requests as they come, apply the rules that are relevant to them, and then go back waiting for additional requests. No end-user interaction within the execution of the business rules is expected. This style is also known as evaluate and return. A common example would be the processing of insurance policies during the renewal cycle. The application hands to the rules service a policy up for renewal. The rules service evaluates the desirability of renewing the policy (or perhaps increasing the premium). Then, the rules service returns to the application the result and goes into a wait state for the next policy up for renewal.

Interactive request response: Your service will be waiting for requests to be submitted, will take the requests as they come, apply the rules that are relevant, possibly interacting with the service requester, and then go back waiting for additional requests. Common examples of this style are intelligent questionnaire applications. The rule service determines that based on information provided so far, additional questions must be posed to the end user and the answers are needed before the rules service can continue.

There are of course many different specific server platforms. However, these platforms support many different styles of server architecture, each of which has an impact on how one goes about integrating Advisor. These architectures differ in the amount of work one has to do oneself versus the amount of support provided as a feature of the platform. This is particularly true if one needs to manage some "state" or context between multiple invocations of a rule agent.

For example, a Web client that presents many different rules-controlled promotional messages needs some mechanism to retain the state of the user "engagement" with the web server (e.g., fields filled in on the web page, promotions already recommended, shown and/or selected).

Lightweight servlet

Your service will be invoked in a new externally managed context whenever a client requests its execution. The servlet environment provides

- *the name services* (the translation of the request as identified by the user into the execution of your Advisor rule service)
- *the execution services* (in general, the creation of a new thread to set the invocation context and invoke your Advisor rule service)
- (possibly) *the session services* (if a context needs to be preserved while the rule service is executing, in particular in the case of interactive sessions). If no session services are provided by the Web Server servlet platform, you will have to maintain state (if state is needed) either in persistent storage or as hidden fields on the HTML page.

Basic server components

Your Advisor rule service will be invoked in a new externally-managed context whenever a client requests its execution. The environment is similar to a servlet except that session services are naturally provided by the platform (such as CORBA service).

Extended server components

Your Advisor rule service will be invoked in a new externally managed context whenever a client requests its execution. The environment provides the same services as for the Basic Server components, but adds:

- The transaction services (often transactions are managed by the server container, and additional services are provided to let your service implement the proper transaction semantics).
- The resource dispensing services (the pooling of server components).

Middleware

Your Advisor rule service will be invoked as a result of a low-level event in the middleware. The environment provides in general very little and almost everything needs to be implemented manually. Examples of this are services which would be invoked via an event from MQ Series, receipt of an FTP file, or any other simple message queuing system.

Monitoring, non-interactive, and interactive services can be deployed in any of the above four server architectures. The following two main tasks need to be carried out:

- Embedding the Advisor engine in an execution platform
- Providing access to the objects and services available in the execution platform.

Embedding the engine in the execution platform boils down, in general, to using the Advisor callable API. In other words, the Advisor rules engine is not in and of itself a complete rules server. It is the responsibility of the application architect to build the surrounding infrastructure that provides the connection between the clients and the Advisor rules engine. This is done through exploiting the published Advisor API. The capabilities of Advisor in that sense are described in the next section.

Providing access to the objects available in the execution platform means, in general, providing access to the entities in your application's object model supported by the execution platform. In certain cases, multiple such object models may be available. The capabilities of Advisor in that sense were described in a previous section.

EMBEDDING THE ADVISOR ENGINE

The Advisor rule engine has been designed so that you can easily embed it in an execution environment you provide to create rulebased servers or standalone applications. The Advisor runtime is a Java package (COM.neurondata.engines.rules) with APIs to control the rule engine, customize its inputs and outputs, map and unmap objects, post events, query and set values, etc. The runtime API revolves mostly around the NdRuleAgent class. This class encapsulates all the high level services that the rule engine provides.

In the case of a monitoring type of service (where Advisor monitors the changed values of the application objects), the embedding follows these steps:

1. Create a rule agent from the execution environment at the point in which the monitoring is to be launched.
2. Start the rule agent with an initial object.
3. Run the rule agent in a separate thread (or possibly process) in such a way that it goes back to wait for more changes when no rule needs to be applied.
4. Use bound properties within your application's objects so as they change value, they are automatically posted into the Advisor rule agent where they will be recognized and process by "whenever xxx is changed" event rules.

The results of the monitoring rules are communicated through the execution of the rules. The "then" and "else" consequences of rules cause changes in objects, sending of messages, or updating of databases. The specification of what these results are is your responsibility.

In the case of a request-response (interactive or non-interactive) service in an application, the embedding follows these steps:

1. Create a rule agent in the execution environment at the point the request is made.
2. Map the external objects on which the rules need to operate.
3. Run the rule agent.
4. Inspect properties within the external objects of step 2 to ascertain results.

In this approach, the results could be communicated as above or could be stored within Advisor objects and variables and explicitly pulled from Advisor via an API call.

In the case of a request-response service in a server, the processing described above is implemented on the server side and the client simply invokes it. Note that the rule agent needs to run in the server in such a way that it does not block the server (i.e. it runs in its own thread) and that in the case of interactive services, it possibly needs to take advantage of session services.

USING THE ADVISOR ENGINE API

To trigger the execution of a rule agent from the execution environment, first create a rule agent (associated to an existing Advisor rules project). You only need a few method calls. Let's assume that the project is stored in a file myproj.adv which is locally accessible (URLs can also be used to trigger the loading of a project from another host).

```
NdProjectResource project = new NdFileProjectResource("myproj.adv");
NdRuleAgent agent = NdRuleAgent.createRuleAgent(project);
agent.loadProject();
agent.compile();
agent.initialize();
```

The loadProject, compile, and initialize calls will load the project from files or URLs, compile it, initialize the objects and variables declared in the rules. In the case no external object needs to be mapped into the rule agent, the runProject call can be used instead of these three. To map external objects into the rule agent, assuming that the corresponding classes have been imported through the relevant wizards, you simply need to use a couple of calls. Let's assume that the external objects to map are stored in a Java vector.

```
NdAdvisorExternalObjectMapper
    mapper =   agent.getExternalObjectMapper();
for (int i = 0; i < objects.size(); i++) {
    mapper.mapInstance(objects.elementAt(i));
}
```

The mapInstance call will analyze the class of the external object, find out which is the Advisor class it needs to map the object into, and then map it. By mapping the object, it becomes accessible from the rules. The rules that operate on object patterns can directly be applied to the object. Using the Advisor callable API, it is also possible to "bind" the mapped external object to a property of an Advisor object or to an Advisor variable.

To execute the rule agent, you simply use the run call.

```
agent.run();
```

If your design calls for Advisor to place its results into properties of mapped objects all you need to do when Advisor returns is inspect and act on those properties. On the other hand, to retrieve the results stored in objects or variables in the rule agent, you need to use the Advisor callable API again. Let's assume the result is stored in a boolean called "result" in the rule agent.

The following piece of code retrieves the result out of the rule agent:

```
boolean result = agent.getVariableBooleanValue("result");
```

Once the results are read, the rule agent can be reused (if wished) but resetting its contents to what it was just after initialization. You can do that by simply invoking the reset method,

```
agent.reset()
```

The typical cycle for a server request response service would then be something like the following (using the assumptions listed above):

```
// Prepare the service
agent.loadProject();
agent.compile();
agent.initialize();
while (true) {
// Start with a clean rule agent
       agent.reset();
       // Map the external objects
       NdADVISORExternalObjectMapper
           mapper = agent.getExternalObjectMapper();
       for (int i = 0; i < objects.size(); i++) {
           mapper.mapInstance(objects.elementAt(i));
       }
       // Execute the agent     agent.run();
       // Read the results (if necessary)
       boolean result = agent.getVariableBooleanValue("result");
       // Report the result
       ../..
       // Wait for a new service invocation
waitForNewRequest();
}
```

where waitForNewRequest calls java.lang.Object.wait() to wait for the arrival of a new request.

You can create as many rule agents as the available memory will allow. Then, you can run all the agents concurrently by calling NdRuleAgent.run() (or runProject()) from separate threads.

The rule engine may need to perform some input or output operations while processing the rules. For example, rules may ask questions or print messages. All the input/output operations are packaged as a Java interface (NdExecutionHandlers). So, you can override the default input/output methods by registering an instance of an object that implements your version of this interface:

```
Class MyExecutionHandlers implements NdExecutionHandlers {
     public boolean promptBoolean(String s) { ... }
     public int promptInteger(String s) { ... }
     ...
     public void print(String s) { ... }
}

// after creating the rule agent:
agent.setExecutionHandlers(new MyExecutionHandlers());
```

You can also monitor what the rule engine does by installing "listeners" on events like rule firing, object creation and deletion, modification of properties, etc.

The NdRuleAgent API also provides methods to investigate the classes, objects and variables during a session. You can:

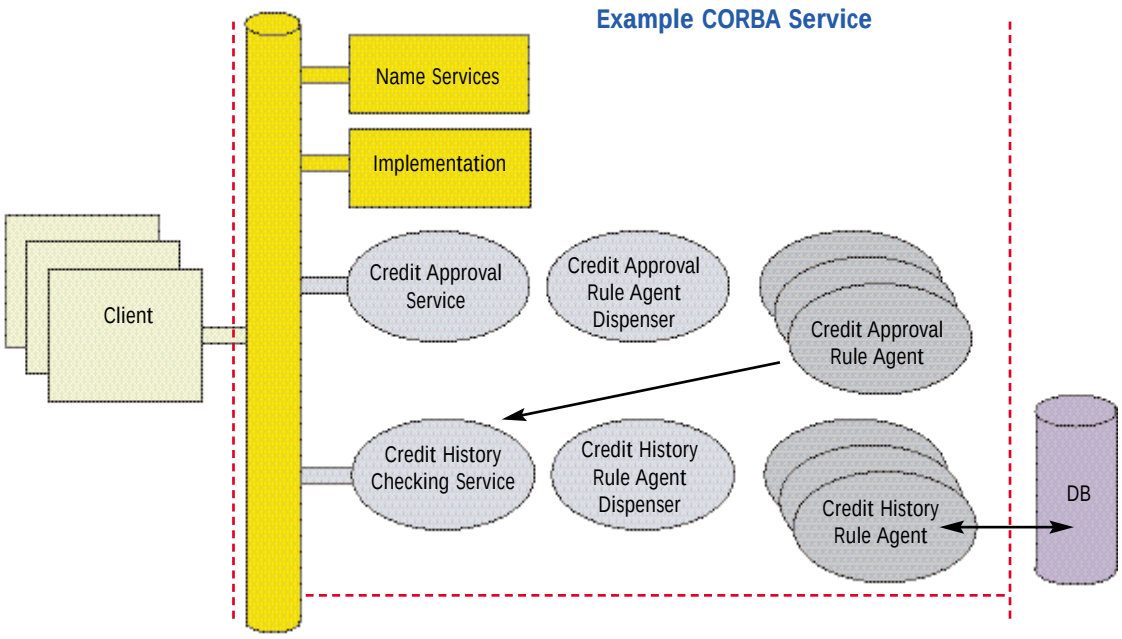
- Query the list of classes and static objects.
- Query the instances of a class.
- Create new objects, or map/unmap existing objects.
- Query and set the values of properties or variables.

So, you have a very rich call-in API to control the engine and query or modify the objects and variables processed by the rules.

The call out API is even richer. You can install your own "execution handlers" and "listeners" but you can also call any public method of any Java class that has been mapped with the "Java Import" wizard. This gives you access to any service implemented in Java (even remote services) directly from the rules. You do not need to learn a special API to access existing Java services.

Finally, you can use the NdRuleAgent API to post events to Advisor that are recognized by any "whenever xxx occurs" event rules.

EXAMPLE



The picture above is an example CORBA architecture involving two cooperating rules services. The scenario is this:

A credit approval rules service is established that is made accessible to clients of various forms. For maximum interoperability, the company has elected to make the Credit Approval Service a CORBA service using one of the leading CORBA vendors such as Iona or Inprise. The way

in which credit approval is done is that the credit approval rules base analyzes and assesses the incoming credit request data. In some cases, the rules may need to acquire the customer's credit history that has also been made a separate CORBA service. The actual credit history exists in a database (in real life, this would probably be done through accessing an external credit bureau).

The Credit Approval and Credit History Checking services are true CORBA services with defined IDL (which must be provided by the application architect). Java bindings for the IDL interfaces are created and imported during the Advisor building phase into the respective rulebases. It is assumed that both services are written in Java.

Because these are true services, they need to be able to have multiple concurrent rule agent threads executing. Otherwise, the services can not have multiple clients connected and processing in parallel. One way to do this is to have a separate thread (or process) running which is called here a "rule agent dispenser." The dispenser's job is to have several Advisor rules agents "warm" loaded (rulebases already retrieved and compiled) and as client requests come into the two services, provide a ready rule agent when asked. The dispenser could be started at the same time the service is started or could be started upon the first client request.

CORBA Name Services are used by the client to locate the Credit Approval service. In the example, only one Credit Approval service is running but there is no reason why more than one could not be started. Then, using some sort of round robin, randomized, or CORBA Trader Service approach, greater scalability and load leveling could be obtained.

In the example, if the Credit Approval rules agent needs to get the credit history, a request is made to get an object value from the Credit History Checking Service. Through CORBA Name Services, the Credit History Checking Service is located, the Credit History Checking Service finds a ready rules agent from its associated dispenser, and the rules for credit history checking are started. These rules retrieve the credit history from the database, analyze the history, and return some sort of history "rating." Upon this return, the Credit Approval rules continue and ultimately decide whether or not to grant the credit approval.

What this example illustrates is the flexibility of architecture provided through Advisor. An Advisor rule is one rulebase can reference an object that could (as the example shows with the Credit History Checking service) be implemented as a remote CORBA service. Of course, the remote service need not be CORBA. It could just as easily be a COM service, a destination on a publish/subscribe messaging system, a Java RMI class or an Enterprise Java Bean.

OTHER EXAMPLES AND CONSIDERATIONS

Pictures for Advisor running as a servlet or within an ActiveX container are similar to the CORBA picture above. Web Servers that support servlets have their own methods for name services and session handling. Microsoft's ActiveX containers also have their own ways of locating services and providing for session management. Optimal performance is achieved by implementing some form of rule agent dispenser of "warm" rule agents. The key in any server implementation is a well-defined interface between the client and server. At a minimum, such an interface should define the object(s) passed to the server, the objects passed back from the

server, an, if needed, objects to use to ask questions of the client from the rulebase. The implementation of these interface objects should be done with performance in mind. They may need to act as a bridge between a "pull" model from the rules agent ("give me an object value when I need it") and the "push" model supported by a client ("here are my objects, do something with them".) The interface may do some caching of values for performance.

PERFORMANCE CONSIDERATIONS

Advisor compiles the rules into an optimized RETE network.

The RETE algorithm is a very efficient algorithm to incrementally recompute the state of a large number of rule conditions when the objects involved in these conditions change. The conditions that appear in more than one rule are "unified" and tested only once. Also, the engine memorizes the objects or combinations of objects that satisfy conditions or partial conditions to reduce the number of tests to be performed when an object changes. All of the most popular rule engine and Expert System tools (particularly all OPS/5 based tools) are based on this algorithm, which has been validated in the industry over the past ten years.

Advisor incorporates some innovative optimizations and enhancements into the original algorithm. For example, it handles inheritance between classes in a much more elegant and efficient way than most OPS/5 based tools.

The use of rulesets to organize rules is another way where Advisor optimizes performance. When the rule engine is within a ruleset, the rule engine only considers those rules and those objects known to that ruleset. This saves time, as the rule engine need not scan rules and objects that simply will not apply.

THE ADVISOR ADVANTAGE

This white paper illustrates some important benefits of the Advisor approach:

- Rules are expressed in a concise and very natural declarative form. Rules can be organized into visual maps and rulesets to localize understanding by the business analyst teams.
- Rules are written against the actual application objects. No separate rules engine / rules language object model is required.
- You do not need to "index" the rules by associating them to specific events. You do not even need to find which combinations of events could cause the rule to apply. The rules are automatically monitored by the rule engine.
- You can concentrate on the high level business logic. The low-level chores (i.e., iterating through the employees that report to a given manager) are handled by the rule engine.
- You get better performance. The rule engine uses a very powerful scheme to index the rules. An ad hoc implementation that would monitor the events one by one is likely to be less efficient.

This ultimately translates into more flexibility and adaptability. As you do not need to "index" the business logic, and as the business logic is expressed in natural terms, you can very easily modify it. You just need to add or modify the business rules.

BLAZE SOFTWARE AND BUSINESS RULES TECHNOLOGY

Blaze Software, Inc., provides the essential products and services to support the vision of consistent customer relationship management across multiple customer touchpoints and customer-facing systems. This is more than ever for self-service access to mission-critical applications via the Web.

Blaze Software is also a leader in providing embeddable Java-based rules engines for independent software companies (ISVs) who need to get to market quickly with the most flexible total product. The number one company in business-rules automation, Blaze Software is a privately held corporation that designs, develops, and markets products used around the world.

A rules-driven approach to application development keeps the business rules and policies separate from the procedural logic, enabling applications to adapt and change easily when the business changes. Using a rulebase that can be shared between applications, intelligent application components can be developed that easily integrate within existing application architectures. This approach is essential to self-service Web applications, which demand constant change. Blaze Software offers both products to create these "adaptive" applications and professional services to support the customer throughout the development process. Blaze Software is a partner with Iona, Inprise, Microsoft, IBM, Active Software and all the relational database vendors.

Customers choose Blaze Software products because they:

- Are proven in the toughest customer environments
- Are linked to a powerful business rule engine
- Have superb business rule visualization for ease of understanding and change
- Are totally open
- Offer superior scalability and performance
- Allow a single application to be written once and deployed on a vast number of platforms in many national languages

Blaze Software's customers have used Advisor to develop industry-leading, mission-critical applications in the areas of customer service, electronic commerce, and automation of financial-services processes such as insurance underwriting.

This paper has been able to discuss only a few of the capabilities of Advisor business rules and their potential business benefits. **For more information, visit our Web site at http://www.blazesoft.com or call us at 800.876.4900 or 650.528.3450.**

APPENDIX: COMPARISON OF ADVISOR TO ALTERNATE APPROACHES

To help compare Advisor to alternate technologies used to create business rules, let us re-introduce the simple business rule from earlier in the paper. In plain English, the rule will read:

```
If the salary of an employee exceeds the salary of his manager then mark
this employee as having a special status.
```

THE ADVISOR APPROACH

In Advisor, this rule can be expressed in a very concise and simple way, with rather classical operators (except the 'any' keyword).

```
emp is any employee.
if emp.salary > emp.manager.salary then emp.status = special.
```

Advisor will also accept a more verbose, English-like formulation and the rule could be rewritten as:

```
if the salary of emp > the salary of the manager of emp
then the status of emp is special.
```

THE SQL TRIGGER APPROACH

SQL triggers are extensions of the SQL query language that let you associate actions to operations on tables. They are at the heart of the products such as Vision Jade. Different vendors have proposed several trigger extensions. In the example below, we chose the PL/SQL language (DB/2, Oracle) but the result would be very similar with other SQL extensions:

```
CREATE TRIGGER checkManagerSalary
AFTER UPDATE OF salary ON employees
DECLARE
    mgrSalary INTEGER;
BEGIN
    SELECT salary INTO mgrSalary FROM employee
    WHERE id = :new.manager_id;
    IF (:new.salary > mgrSalary)
    THEN UPDATE employee
         SET status = 'special' WHERE id = :new.id;
    END IF;
END
```

Here, the result is somewhat understandable because of the "natural language" orientation of SQL. But still, we are far from the original English formulation. SQL lacks some fundamental Object-Oriented features like object references (or pointers). Then, the salary of the manager must be retrieved through a separate SELECT statement, which leads to a very heavy formulation.

The rule systems that have emerged around data bases (rules written as triggers) are tightly related to the SQL language. This language is a powerful and natural query language but is very "data" centric. Here, the rules are applied to "data" rather than to "objects" and all the manipulations need to be programmed at the "data" level. The lack of object "references" is particularly critical.

THE 3GL APPROACH

The rule can also be expressed in an third generation programming language like Visual Basic, Java, JavaScript, C or C++. For example, the rule can be captured in Java as:

```
Class Employee {
     ...
     void setSalary(double newSalary)
     {
          salary = newSalary;
          if (manager != null && salary > manager.salary)
               status = SPECIAL;
     }
}
```

This formulation is actually relatively easy to understand and fairly similar to the Advisor formulation. We need an additional test on the manager field to handle the top-level manager correctly (the CEO is likely to have his manager field set to null and it would be a bad idea to have the application crash when the CEO's salary is increased).

The main difference with the Advisor approach is the fact that the rule is embedded in a method (setSalary). The business logic (the if test) is intertwined with the application infrastructure (the assignment of the salary field). Then the business rules are much more difficult to find and maintain.

THE EXPERT SYSTEM APPROACH

The expert system tools that have emerged from artificial intelligence research can capture complex rules. But the rules are usually expressed in esoteric languages. For example, OPS/5, the most popular rule language among the AI tool vendors would let you express the previous rule as:

```
(defrule check-manager-salary ""
  ?e1 <- (employee (manager ?m1) (salary ?s1))
  (employee (id ?m1) (salary ?s2: ?s1 > ?s2))
=>
  (modify ?e1 (status special)))
```

The systems that have their roots in AI research are usually based on an object model but the underlying model is either very simplistic (e.g., OPS/5) or on the contrary much more sophisticated and flexible than classical models (e.g., Common Lisp Object System). In both cases, the rules cannot directly operate on existing objects. They have to go through a more or less complex mapping layer or through a native code generation scheme, with all sorts of negative consequences (duplication of class definitions, copy of data, etc.).

EXPLICIT VS. IMPLICIT TRIGGERING

Although the different versions of the rule express the same "logical" connection between a condition and an action, they are not completely equivalent because they are not be triggered by the same events. As we stated it initially, the rule may be triggered when one of the following events occur:

- The employee's salary is raised.
- The manager's salary is cut down.
- The employee changes manager and his new manager has a lower salary than his old manager.

The implementations that we are comparing will react differently to these events:

- The Advisor rule engine will respond to all three events.
- Most expert system tools will also respond to the three events.
- The database trigger is associated to an update on the employee's salary. As written above, it only responds to the first event. To handle the three events, we would have to create two additional triggers.
- The Java version will also respond only to the first event. The third event can be handled by isolating the rule in a separate method that gets called from the `setSalary` and the `setManager` methods. Handling the second event is more involved because it requires iterating through the employees that report to the manager, every time the manager's salary changes.

In Advisor, the rules are triggered implicitly. They are not associated to specific events (although the language also allows you to do this). The engine monitors the state of the objects and is able to find out when the conditions of a rule become true, regardless of why this condition became true.

On the other hand, database triggers and 3GL languages require that you explicitly associate the rule with one or several events. In dynamic objects systems where the relationships between the objects change at run-time, if becomes very difficult to guarantee that the rules will be checked in a systematic way. This requires a careful analysis of the events that may cause the rule to hold and some rather tedious programming afterwards.