Hammurapi Group

Hammurapi rules
user guide

Legal Notices

Table Of Contents

Introduction

Hammurapi rules is a [JSR-94](#) compatible [forward chaining](#) [Rule engine](#).

Feature highlights

- Java is the primary language for rules authoring. **Hammurapi rules** leverages Java type system and several naming conventions to build its Rete network.

    o Java developers quickly get familiar with rules authoring.

    o No need for a Java developer to learn a specialized rules language and development tools and switch between languages. Developers keep "thinking in Java", which strengthens their Java skills instead of diluting them shall they have to switch between languages (see [Occam's Razor overarching principle](#)).

    o Debug rules in your Java IDE.

    o Compile time checks vs. runtime exceptions. There is no interpretation steps once rule set is instantiated from XML definition (rule sets are defined in XML) but only fast, robust, type-safe compiled Java.

    o Rules can natively access underlying Java application.

- Flexibility

    o Rule sets are defined in XML.

    o Rules can be parameterized at assembly time and registration time.

    o Specialized rules languages can be added if needed.

- Scalability

    o **Hammurapi rules** supports multithreaded inference.

    o Working memory can be stored to disk or to database and as such survive JVM shutdown.

    o Inference can be distributed across multiple machines.

- Rules are assembled into rule sets and parameterized at runtime by the means of XML. **Hammurapi rules** can load rule sets from different sources including file, url and classloader resources.

- Negations support.

- Logical loops detection.

- Autodetection of updates.

- Rule engine builds derivation trees to help in debugging rules logic. Derivation trees can be dumped to XML or visualized in Swing GUI.

Installation

- Download **Hammurapi rules** from the **Hammurapi rules** download page.

- Unzip it to location of your choice, e.g. C:\Tools\HammurapiRules.

- Set HAMMURAPI_RULES_HOME environment variable to the location where you unzipped the product, e.g. C:\Tools\HammurapiRules.

- Add the product directory to the system path.

- If you use Java 1.4, download JAXP-1.3 from https://jaxp.dev.java.net/1.3/index.html. Install it and add jar files from the JAXP installation directory to the Hammurapi Rules `lib` directory.

Quick start guide

This sections provides a quick overview of rule engines, JSR-94 and **Hammurapi rules**.

Terminology

- **Fact** is a piece of knowledge. E.g. "*John is a 45 years old male*" or "*Jim is a child of John*". Facts are inputs to **rules**.

- **Conclusion** is a fact derived/inferred from other facts. E.g. from the two facts above and the fact that "*Mary is a child of Jim*" we can conclude that "*John is a grandfather of Mary*".

- **Knowledge base** in an internal collection of facts including conclusions which rule engine operates with. In some engines knowledge base is called *working memory*. **Hammurapi rules**' knowledge base can be held in memory or in persistent storage.

- **Rule** is a piece of logic which can make conclusions based on provided facts and context (e.g. configuration parameters). Rules can also execute actions. E.g. "*If the credit limit of the customer is greater than the amount of the invoice and the status of the invoice is unpaid, then **decrement** the credit limit with the amount of the invoice and **set** the status of the invoice to paid.*

- **Rule set** is a collection of rules.

- **Rule engine** (a forward chaining one) is a piece of software which hosts rule sets, passes input facts to rules, collects conclusions, and passes the conclusions to rules. By doing so it chains conclusions and allows to infer non-trivial ones from simple facts and with simple rules. Please note that in **Hammurapi rules** chaining conditions and

chaining conclusions is the same thing, because condition evaluated to true is a form of conclusion. See Rete algorithm section for more details.

- **JSR-94** is a Java specification for interfacing with rule engines. For rule engine implementations it serves the same purpose as JDBC for databases or JSR-168 for portlets.

Using rule engine in a Java program

Using a JSR-94 compatible rule engine in a Java application includes the following steps:

- Obtain rule service provider.

- Obtain rule runtime.

- Create rule session.

- Give input facts to the session.

- Collect conclusions.

- Release the session.

There are two types rule sessions in JSR-94: stateful and stateless. A stateless session gets all inputs and returns conclusions in one method call. A stateful session allows to add facts and draw conclusions in a series of calls over the course of program execution.

The example below shows how to use a stateful rule session.

```
String ruleServiceProviderClassName = "biz.hammurapi.rules.jsr94.FileRuleServiceProvider";
Class.forName(ruleServiceProviderClassName);
RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider(ruleServiceProviderClassName);
RuleRuntime runtime = serviceProvider.getRuleRuntime();
StatefulRuleSession session = (StatefulRuleSession) runtime.createRuleSession(ruleSetUri, null,
RuleRuntime.STATEFUL_SESSION_TYPE);

Person kate = new Person("Kate", 58, false);
Person victor = new Person("Victor", 63, true);
session.addObject(new Spouse(kate, victor));

Person peter = new Person("Peter", 37, true);
session.addObject(new Child(peter, kate));
session.addObject(new Child(peter, victor));

... Add more facts ...

session.executeRules();

Iterator it=session.getObjects().iterator();
while (it.hasNext()) {
        System.out.println(it.next());
}

session.release();
```

Writing rules

In **Hammurapi rules** rules are typicaly written in Java. There is no need for a Java programmer to learn a new language and new IDE to write and troubleshoot rules. Also there are compile-time checks, which reduce probability of runtime errors.

NOTE: *Specialized rules language can be added to the engine if it is justified by the problem domain and rule authors' skill set.*

NOTE: Most code snippets in this document are based either on **Hammurapi rules tutorial** or **Hammurapi rules Technology Compatibility Kit (TCK)**, which can be downloaded from the **Hammurapi rules** [download page](#).

Java rules should extend `biz.hammurapi.rules.Rule` and implement one or more `infer()` methods. Infer methods shall take one or more parameters.

Example:
```
public class DaughterRule extends Rule {

        public void infer(Child child) {
                if (!(child instanceof Daughter) && !child.getSubject().isMale()) {
                        post(new Daughter(child.getSubject(), child.getObject()));
                }
        }
}
```

New facts can be posted to other rules either by `post()` method or by simply returning a value from `infer()` method as shown below. These two methods are equivalent and can be combined. Null values are not posted to rules.
```
public GrandDaughter infer(Daughter daughter, Parent parent) {
        return daughter.getObject().equals(parent.getObject())
                              ? new GrandDaughter(daughter.getSubject(), parent.getSubject())
                              : null;

}
```

In addition to `infer()` methods rules can implement `accept()` methods. These methods are used to filter objects which are passed to multi-fact (multi-parameter) `infer()` methods.

Example:
```
public class TckRule extends Rule {

        /**
         * This method filters invoices so only unpaid ones are passed to <code>infer(customer,
invoice)</code>
         * @param invoice
         * @param acceptInfo
         */
        public boolean accept(Invoice invoice, AcceptInfo acceptInfo) {
                return !"paid".equals(invoice.getStatus());
        }

        /**
         * If the credit limit of the customer is greater than the amount of the invoice
         * and the status of the invoice is unpaid then decrement
         * the credit limit with the amount of invoice and set the status of the
         * invoice to paid.
         * @param customer
         * @param invoice
         */
```

```
        public void infer(Customer customer, Invoice invoice) {
            if (customer.getCreditLimit()>invoice.getAmount()) {
                customer.setCreditLimit(customer.getCreditLimit()-invoice.getAmount());
                invoice.setStatus("paid");
            }
        }
}
```

If a rule implements single-argument `remove()` method(s) then this method(s) will be invoked when objects are removed from knowledge base. Collection manager and handle manager, which will be described later, take care of removing objects from internal collections. Therefore rules rarely need to implement `remove()` methods themselves.

Negation

Negation is a way to say "this fact is not true". In **Hammurapi rules** object which negates other object shall implement `biz.hammurapi.rules.Negator`. When a negator is posted to the knowledge base, all facts/conclusions which are directly negated by the negator are removed from the knowledge base. Also all conclusions based on facts/conclusions directly negated by the negator are removed from the knowledge base.

Example: From the facts that "*John is a 45 years old male*", "*Jim is a child of John*", and "*Mary is a child of Jim*" we came to a conclusion that "*John is a grandfather of Mary*". If we post a negator that negates "*Mary is a child of Jim*", it will also negate "*John is a grandfather of Mary*" because it is inferred from "*Mary is a child of Jim*". Both these facts will be removed from the knowledge base.

Composite classes may choose to implement `biz.hammurapi.rules.Negatable` interface. `biz.hammurapi.rules.Conclusion` class implements this interface. It is recommended to use this class as a base class for conclusions. Conclusion class also has `object2Negator()` convenience method.

A fact can be a negator at the same time. For example, because a person can have only one mother, the fact that "***Nancy** is Jim's mother*" would negate "***Margaret** is Jim's mother*".

Negations is a powerfult feature and shall be used with care because negated facts and conclusion are removed from the knowledge base and disappear. As such bugs in negation logic it can be difficult to debug. A word of advice is to add logging statements to `negates()` and `isNegatedBy()` methods.

Updating and removing facts

When `removeObject()` is invoked, **Hammurapi rules** posts a negator of this object to the knowledge base. The object being removed and also all conclusions based on this object get removed from the knowledge base.

As stated in the specification update is equivalent to removal of object from the knowledge base and then adding new/modified object the knowledge base.

Hammurapi rules can automatically detect changes in some types of facts and invoke <u>Rule's</u>
`update()` method for changed objects. This feature is described in more detail in the
<u>Archtecture</u> section.

Parameterization

Rules can be parameterized. To be parameterizable, a rule shall implement proper setters and
the XML rule descriptor shall contain corresponding elements.

The code snippets below show fragments of the <u>Validator</u> class source and its XML definition
in the rule set.

```
public class Validator extends Rule {

        ...

        private int legalAge;

        /**
         * Age when people are allowed to marry
         * @param legalAge
         */
        public void setLegalAge(int legalAge) {
                this.legalAge = legalAge;
        }

        /**
         * Validates that difference between parent and child age is not less than legal age.
         * @param mother
         */
        public void validate(Parent parent) {
                if (parent.getSubject().getAge()-parent.getObject().getAge()<legalAge) {
                        post(new SuspectParentRelationship(parent, parent.getSubject() + " was under
legal age when "+parent.getObject()+" was born. Please verify input data."));
                }
        }
}
...
        <rule type="biz.hammurapi.rules.tutorial.rules.Validator">
            <name>Validates conclusions</name>
            <description>Contains a number of different validations.</description>

            <legalAge>18</legalAge>
        </rule>
...
```

Parameters are instantiated and injected into rules by the means of <u>DomConfigFactory</u>, which
allows to instantiate and initialize a wide variety of Java classes including collections and
maps. Parameterization with maps is one of ways of implementing decision tables.

Rules can also access other components and rule engine properties using the naming bus. For
example to access property "foo", which was set when rule set was registered or when the rule
session was created, a rule can use `get("/@foo")` method call.

Creating a rule set

Once rules are written, they shall be assembled into a rule set. **Hammurapi rules** rule sets are
defined in XML. All aspects of the rule set XML definition will be described below. Here we'll
just mention that typically you'd need to take a template definition and then change its name,

description, and rule definitions under `rules` element. The sample below shows the rule set for **Hammurapi rules** Technology Compatibility Kit.

```
<ruleset type="com.pavelvlasov.config.ElementNameDomConfigurableContainer">

    <name>Hammurapi Rules TCK for JSR-94</name>
    <description>This is rule set used in JSR-94 TCK (Technology Compatibility Kit)</description>

    <handle-manager type="biz.hammurapi.rules.IdentityHandleManager"/>

    <collection-manager type="biz.hammurapi.rules.PojoCollectionManager"/>

    <rules type="biz.hammurapi.rules.QueueingRulesContainer">
        <rule type="biz.hammurapi.rules.jsr94.tck.TckRule">
            <name>TCK Rule</name>
            <description>If the credit limit of the customer is greater than the amount of the invoice
 and the status of the invoice is unpaid then decrement
 the credit limit with the amount of invoice and set the status of the
 invoice to paid.</description>
        </rule>
    </rules>
</ruleset>
```

Registering a rule set

The JSR-94 API requires that a rule set shall be registered before it can be used. This approach decouples rule set providers from rule set consumers. It is similar to Oracle's TNS names and thick JDBC client, and to Microsoft's ODBC DSN's.

There are two ways to register a rule set in **Hammurapi rules**. They are described below.

With the command line utility

```
hradmin register <uri> -f <file>
```

Programmatically

```
String ruleServiceProviderClassName = "biz.hammurapi.rules.jsr94.FileRuleServiceProvider";
Class.forName(ruleServiceProviderClassName);
RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider(ruleServiceProviderClassName);
RuleAdministrator administrator = serviceProvider.getRuleAdministrator();
RuleExecutionSetProvider resp = administrator.getRuleExecutionSetProvider(null);
RuleExecutionSet res = resp.createRuleExecutionSet(DOMUtils.parse(new
File(ruleSetFile)).getDocumentElement(), properties);
administrator.registerRuleExecutionSet(uri, res, null);
```

NOTE: While being a valuable feature, registration of rule sets sometimes becomes a hindrance. For example, when a rule engine is used from an applet and needs to load rule set definition from URL. **Hammurapi rules** solves this problem by using a special form of rule set URI. If URI starts with "`direct:`", then the rest of the URI is treated as a rule set URL and the rule set is loaded from that URL instead of the registrations file. **Hammurapi rules** can also load rule sets from class loader resources.

Troubleshooting

In **Hammurapi rules** conclusions and derivations are XML-serializable. If you wonder how some particular conclusion was derived, you can dump it to XML and then browse derivation tree in a browser. The sample below shows how to dump conclusions to an XML file.

```
DOMUtils.serialize(objects, "conclusions", new File("conclusions.xml"));
```

Here is the XML output.
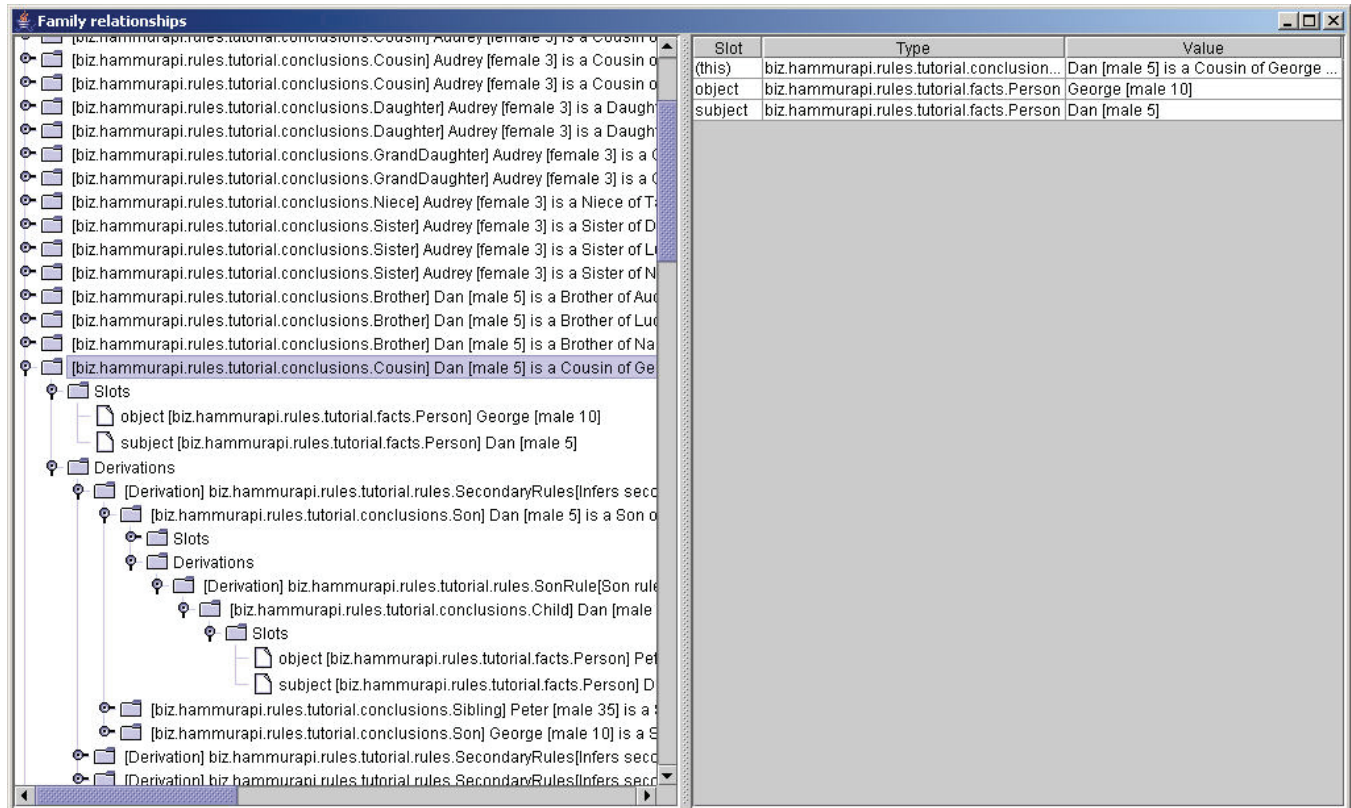
Figure 1 Conclusion browser

You can also use `Browser` class to visualize consclusions and derivations in the Swing GUI as shown in the code snippet below and on Figure 1.

```
Browser.show(objects, "Family relationships");
```

Persisting knowledge base

Out of the box handle manager and collection manager implementations hold knowledge base in memory. To make knowledge base persistent it is generally recommended to provide application-specific implementations of the managers.

Another option is to use the out-of-the-box managers and provide them a reference to ObjectStorage implementation. **Hammurapi rules** comes with FileObjectStorage class which implements ObjectStorage and uses Java serialization to read/write object from/to a file.

Distributed inference

With **Hammurapi rules** it is possible to build a distributed rule engine. This can be achieved by adding "remote rules" to the rule set. Remote rule will send objects dispatched to it to remote engines and post objects received from remote engines to the object bus. A number of communication mechanizms can be used for rules communications e.g. JMS or JGroups

Architecture

Administrative interface
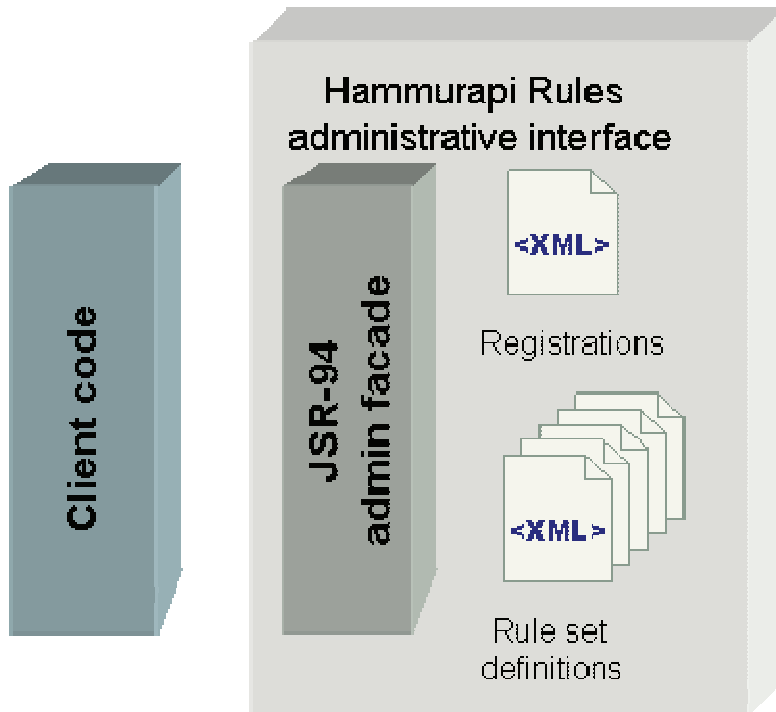
Figure 2. Administrative interface

Administrative part of the **Hammurapi rules'** JSR-94 implementation deals with management of rule set definitions in `.hammurapi-rules/registrations.xml` file in the user home directory. Also **Hammurapi rules** provides a command line administration utility.

NOTE: Rule set registration URI's shall not start with `direct:` because it is a reserved URI prefix for direct loading of unregistered rulesets at runtime. Direct loading might be needed if you distribute rule sets bundled with your product. In such a case you can store them as a classloader resource and use `direct:resource:<rule set resource path>` URI to load the rule set at runtime. Another scenario is to host rule sets on your web site and use them directly from your product without requiring user to explicitly register rule sets in order to make the product operational.

Registrations file XML format

- The root element is `registrations.`

- The root element contains multiple `registration` elements.

- `registration` element has a mandatory `uri` attribute.

- It also must contain either `ref` attribute which contains a reference (URL) to an external rule set definition or `definition` element which holds a rule set definition.

- Optionally, there can be `properties` element which holds XML-ized property map passed to `createRuleExecutionSet()` methods.

This is an outline of the registrations file structure:

```
<registrations>
    <registration uri="RuleSet1" ref="http://www.mysite.com/myrules.xml"/>

    <registration uri="RuleSet2">
        <definition>
            ...
        </definition>

        <properties>
            ...
        </properties>
    <registration>
<registrations>
```

Rule set XML format

Rule set XML definitions are used by both runtime and adminitrator API's. At runtime `DomConfigFactory` is used to instantiate the engine class specified in the mandatory `type` attribute.

The name of the root element doesn't matter. The root element's class should implement `Context` interface. It is recommended to use subclasses of `DomConfigurableContainer`, `ElementNameDomConfigurableContainer` in particular as the root object. We recommend to use `ruleset` as the root element name.

The root element shall have the following subelements:

- `name` - Ruleset name.

- `description` - Ruleset description.

- `rules` - Container of rules with `rule` subelements.

- `handle-manager` - Handle manager.

- `collection-manager` - Collection manager.

- `knowledge-compactor` - Optional knowledge compactor.

- Default object filter can be optionally defined in `object-filter` element's `type` attribute.

Example of rule set definition

```
<ruleset type="com.pavelvlasov.config.ElementNameDomConfigurableContainer">
```

```
    <name>Family ties</name>
    <description>Infers familty relationships from gender and parent/child relationship</description>

    <handle-manager type="..."/>

    <collection-manager type="..."/>

    <rules type="...">
        <rule type="...">
            <name>Grandmother</name>
            <description>Infers 'grandmother' relationship from 'mother' and 'child'
relationships.</description>
            ...
        </rule>

        ...
    </rules>
</ruleset>
```

There can be additional elements supporting work of the mandatory components. For example, a thread pool declaration. Rule set XML structure reflects rule engine runtime structure which is shown on Figure 3. Its components are described below.

Runtime components

19
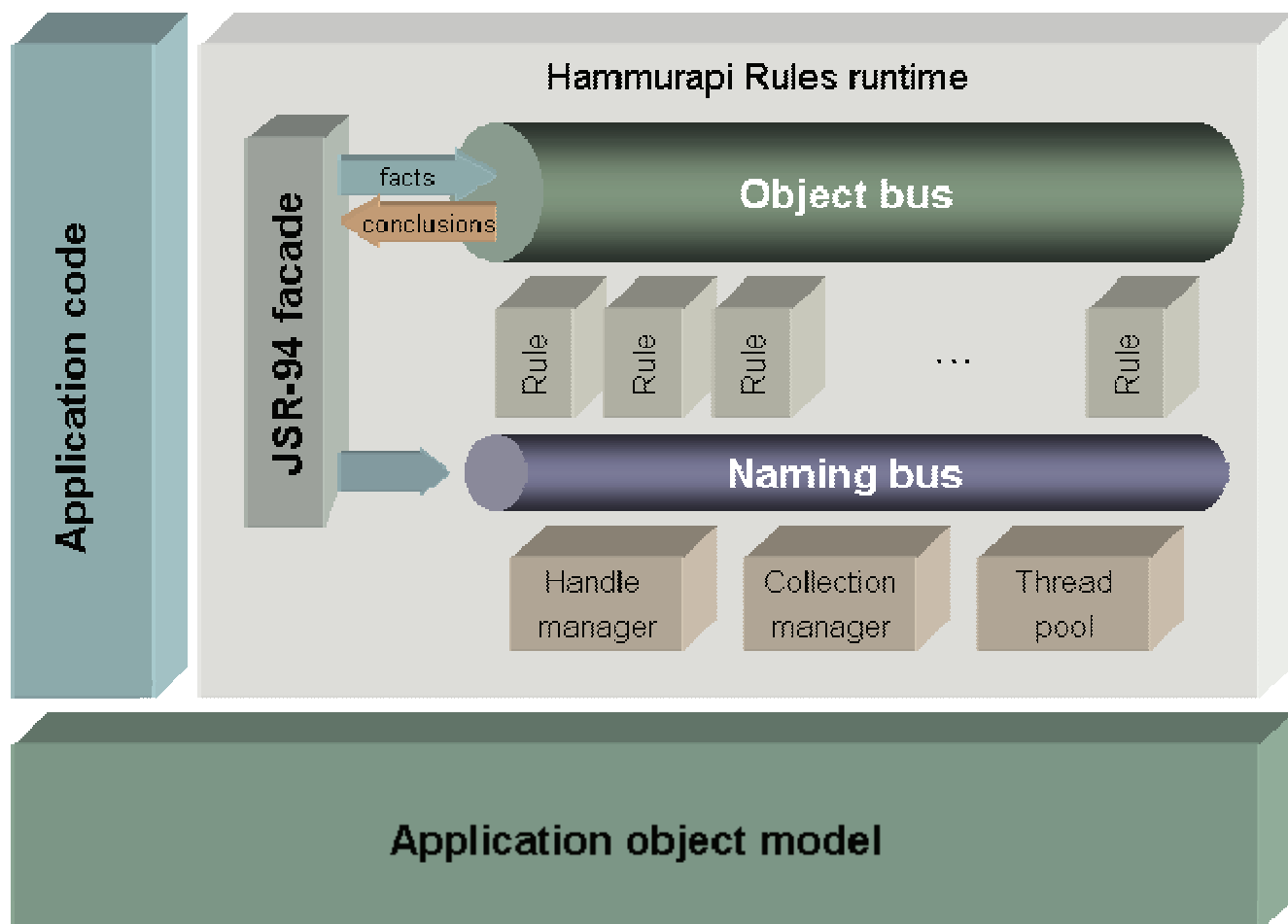
Figure 3. Runtime architecture

- **Rules** are Java objects which are "plugged" into the **Object bus** and the **Naming bus**.

- **Object bus** dispatches objects (facts) to matching rules' `infer()` methods.

- **Naming bus** allows engine components to locate each other.

- **Handle manager** keeps references to knowledge base objects (posted facts and inferred conclusions).

- **Collection manager** manages collections for multi-parameter `infer()` methods.

- **JSR-94 facade** (implementation of `RuleSession`) coordinates work of the engine components.

- **Thread pool** can be used for multi-threaded inference. It is shown as an example of an optional component, which can be located through the naming bus and used by other components.

Object bus internals

The Object bus is built on classes in `biz.hammurapi.dispatch` package. The job of the Object bus is to dispatch objects posted to the bus to rules' methods which can take those objects as arguments i.e. `object` **`instanceof`** `argument type` evaluates to `true`. Needless to say that the Object bus shall do its job in the most efficient manner; as such a simple iteration over all rules' methods is not an option. Figure 3. shows the internal structure of the Object bus.

[edit]

Invocation handlers

Rules are "plugged" into the Object bus by "invocation handlers". Definition of the invocation handler interface is shown below

```
public interface InvocationHandler {

        /**
         * Invokes target method. Passes returned value(s) to result consumer.
         * The target "method" might be invoked multiple times, e.g. in composite
         * invocation handler scenario.
         * @param parameter
         * @param resultConsumer
         * @throws Throwable
         */
        void invoke(Object arg, ResultConsumer resultConsumer) throws Throwable;

        /**
         * This method is used to build invocation network.
         * @return Invocation parameter type. Null if parameter type is unknown.
         */
        Class getParameterType();
}
```

. One rule can expose one or more invocation handlers to the bus.

Subclasses of [Rule](Java rules) introspect themselves in the constructor and create an invocation handler for each parameter of `infer()` methods and a remove handler for each `remove()` method. `accept()` methods are connected to matching multi-parameter `infer()` methods' handlers as shown on Figure 5.

Rule class for a specialized rules language (see below) would need to parse rule's context definition and create an invocation handler for each declared context object.
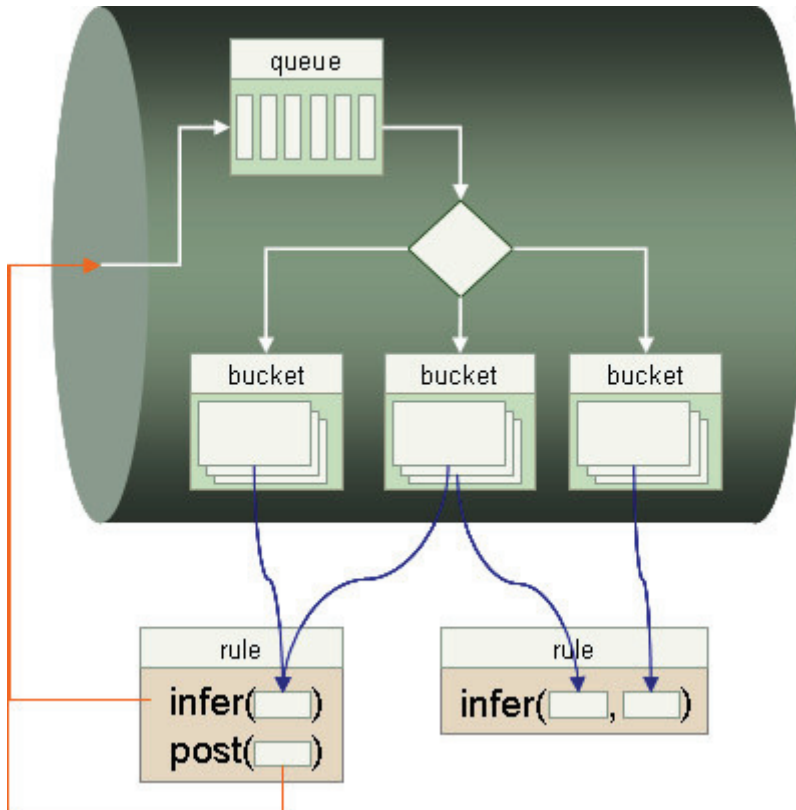
Figure 4. Object bus

The bus collects invocation handlers from rules and organizes then into buckets. There is an internal map which maps classes to buckets. One handler can be put to multiple buckets. E.g. a handler which takes `java.util.Collection` will be put to `java.util.LinkedList`, `java.util.ArrayList` buckets and to buckets of other classes which implement `java.util.Collection`.

Dispatching

When an object is posted to the bus it is put to the internal queue. The queue is needed for multithreaded inference and also to prevent reentrancy because reentrancy would be a real headache collection management wise. There is a queue processing thread which retrieves objects from the queue and either posts them to a thread pool attached to the engine for processing or processes them by itself if there is no thread pool to delegate this job to.

During object processing a bucket for object's types is retrieved from the map and all handlers in the bucket are invoked. If there is no bucket for a given type, then it gets created.

Example: *The object's class implements multiple interfaces. Rules don't use the object's class per se in `infer()` methods' parameters, but only some interfaces implemented by this class. In this case a bucket for the object's class will not be created at rule engine start because the rule engine has no knowledge about this class. A bucket will be created only when an instance of the class is posted to the bus.*

In Java rules return values from `infer()` methods and arguments of `post()` methods are posted to the bus. Null values are not posted.
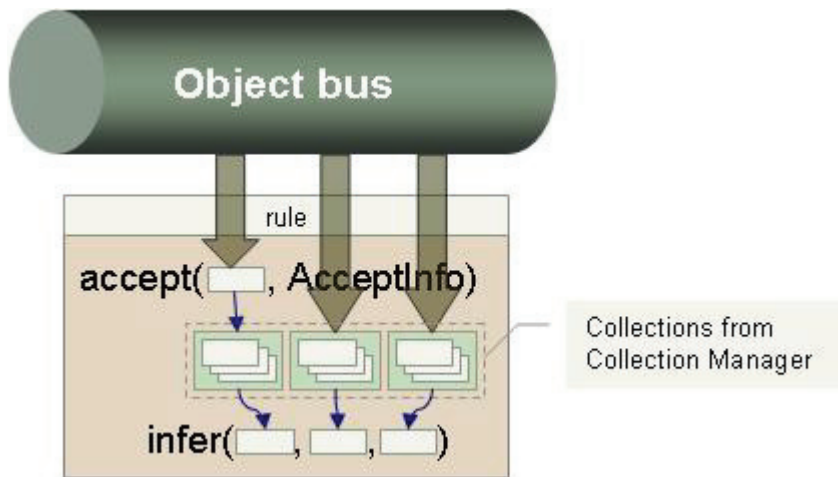
Figure 5. Multi-fact inference

Multi-parameter infer() methods and collection manager

If you were carefully reading the document, you should ask by this point: "How is it possible to invoke multi-parameter `infer()` methods if objects are posted to the bus one by one?" This is a very good question. And the answer is: "This is where the collection manager comes into play". For each parameter in a multi-parameter `infer()` method an invocation handler is created and is put to the appropriate Object bus' buckets. This invocation handler retrieves a collection from the collection manager. The collection name consists of rule name, method signature and parameter index. When an object is dispatched to the handler, the handler adds the object to the parameter's collection. If collection's `add()` method returns `true`, then other method's parameters' collections are iterated and all permutations of the dispatched object and elements in other method's collections are passed to the method. Figure 5. demonstrates the concept.

If the rule's condition includes predicates which involve only one parameter, then it is recommended to implement `accept()` method for this parameter. The `accept()` method must take two parameters and return `boolean`. The first parameter must be of the same type as the matching `infer()` method parameter. The second parameter must be of type `AcceptInfo`. Accept methods filter input objects. By doing so they reduce collections size and improve performance. The code snippet below demonstrates how to use `accept()` methods.

```
public class TckRule extends Rule {

        /**
```

```
        * This method filters invoices so only unpaid ones are passed to <code>infer(customer,
invoice)</code>
        * @param invoice
        * @param acceptInfo
        */
       public boolean accept(Invoice invoice, AcceptInfo acceptInfo) {
               return !"paid".equals(invoice.getStatus());
       }

       /**
        * If the credit limit of the customer is greater than the amount of the invoice
        * and the status of the invoice is unpaid then decrement
        * the credit limit with the amount of invoice and set the status of the
        * invoice to paid.
        * @param customer
        * @param invoice
        */
       public void infer(Customer customer, Invoice invoice) {
               if (customer.getCreditLimit()>invoice.getAmount()) {
                       customer.setCreditLimit(customer.getCreditLimit()-invoice.getAmount());
                       invoice.setStatus("paid");
               }
       }
}
```

The `acceptInfo` parameter is needed to differentiate `infer()` methods and parameters. For example, a rule may have multiple `infer()` methods which take parameters of the same type; some rules may have one `infer()` method taking several parameters of the same type. In such situation one `accept()` method will be bound to all parameters of the same type and `acceptInfo` will be a discriminator allowing to switch filtering logic.

New invocation semantics

Probably the best way for a Java developer to grasp **Hammurapi rules** concepts is to think about the object bus and dispatching as one more way to invoke a method.

Java language itself provides several semantics of invocation, i.e. binding a method call (e.g. `println(a)`) with the body of code (e.g. `println(java.lang.String)` method body if `a` is of type `String`). Java built-in invocation semantics include method overloading, method overriding, implementation of abstract methods, ...

Web developers are familiar with the concept that when servlet container receives HTTP request it eventually translates into `Servlet.service()` invocation, but before that several `Filter.doFilter()` methods might be invoked.

J2EE developers know that call of `create()` method in EJB home interface translates into invocation of `ejbCreate()` method of the enterprise bean.

Hammurapi rules continues the tradition and **Hammurapi rules** developers shall remember that call of `addObject()` eventually translates into invocation of `infer()` rule methods with formal parameters compatible with object's type. The trick, though, is that in the case of servlet or EJB there is one-to-one relationship between the stimulus (HTTP request or `create()` call) and response (invocation of `service()` or `ejbCreate()`) and the response is synchronous. In **Hammurapi rules** it is valid only for single-parameter `infer()` methods. In the case of

multi-parameter `infer()` methods the relationship between stimulus and response is one-to-many and the response may be deferred.

Autodetection of updates

Hammurapi rules can detect updates in objects of sertain types passed to `infer()` methods as parameters. When it is detected that an object passed to rule's `infer()` method as a parameter has changed inside the method, the rule's `update()` method is invoked with the changed object as parameter. Changes can be detected in instances of `Versioned` and `Observable`. If some class implements both the interfaces, then `Versioned` is used for change detection because it is more efficient.

The `update()` method is invoked after `infer()` method returns. Therefore, it is invoked only once per changed object even if there have been multiple modifications of the object inside `infer()` method.

Notes

- [SQLC](#)-generated classes implement both Versionable and Observable interfaces which makes them a good choice for the application object model.

- [Aspect Oriented Programming](#) can "introduce" Versioned or Observable interfaces to existing classes either at compile time or at class loading time.

Lifecycle of rule engine components

Lifecycle of rule engine components is managed by `DomConfigurableContainer`. Here is a synopsis of the component lifecycle

- Component is instantiated from XML definition by `DomConfigFactory`.

- Component is configured by `DomConfigFactory`.

- If component's class implements `com.pavelvlasov.config.Component`, then

   o Component's `setOwner()` method is invoked to give the component access to the naming bus.

   o When engine's `start()` method is invoked as part of rule session initialization, the engine invokes component's `start()` method.

   o When engine's `stop()` method is invoked from rule session's `release()` method, the engine invokes component's `stop()` method.

Tutorial

Hammurapi rules tutorial demonstrates a rule set which infers multiple family relationships from `Child` and `Spouse` relationships.

Tutorial files can be downloaded from the **Hammurapi rules** [download page](#).

In this section we'll take a look at several code snippets and also explore how to register the tutorial rule set and run it.

Installation

- Download **Hammurapi rules** and tutorial files from the **Hammurapi rules** [download page](#).

- Install **Hammurapi rules** as described in [Installation](#) section.

- Unzip tutorial files.

Looking inside

In Tarantino's fashion we'll start with the final scene, and then we'll unwind the mystery of getting there. The listing below is the `main()` method of the tutorial.

*All characters in this tutorial and their relationships are fictitious and are used solely to demonstrate features of **Hammurapi rules**; and any resemblance to actual persons, living or dead, is entirely coincidental.*

```
1  System.out.println("Hammurapi rules tutorial");
2
3  if (args.length!=1) {
4       System.out.println("Usage: java <options> biz.hammurapi.rules.tutorial.Tutorial <rule set
uri>");
5       System.exit(1);
6  }
7
8  String ruleServiceProviderClassName = "biz.hammurapi.rules.jsr94.FileRuleServiceProvider";
9  Class.forName(ruleServiceProviderClassName);
10 RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider(ruleServiceProviderClassName);
11 RuleRuntime runtime = serviceProvider.getRuleRuntime();
12 System.out.println("Loading rule set from "+args[0]);
13 StatefulRuleSession session = (StatefulRuleSession) runtime.createRuleSession(args[0], null,
RuleRuntime.STATEFUL SESSION TYPE);
14
15 Person kate = new Person("Kate", 58, false);
16 Person victor = new Person("Victor", 63, true);
17 session.addObject(new Spouse(kate, victor));
18
19 Person peter = new Person("Peter", 37, true);
20 session.addObject(new Child(peter, kate));
21 session.addObject(new Child(peter, victor));
22
23 Person alison = new Person("Alison", 36, false);
24 session.addObject(new Spouse(peter, alison));
25
26 Person lucy = new Person("Lucy", 17, false);
27 session.addObject(new Child(lucy, alison));
```

```
28
29 Person nancy = new Person("Nancy", 14, false);
30 session.addObject(new Child(nancy, peter));
31
32 Person dan = new Person("Dan", 7, true);
33 session.addObject(new Child(dan, peter));
34 session.addObject(new Child(dan, alison));
35
36 Person audrey = new Person("Audrey", 4, false);
37 session.addObject(new Child(audrey, peter));
38 session.addObject(new Child(audrey, alison));
39
40 Person tanya = new Person("Tanya", 31, false);
41 Person max = new Person("Max", 32, true);
42 session.addObject(new Spouse(tanya, max));
43 session.addObject(new Child(tanya, kate));
44 session.addObject(new Child(tanya, victor));
45
46 Person vilma = new Person("Vilma", 14, false);
47 session.addObject(new Child(vilma, tanya));
48
49 Person george = new Person("George", 10, true);
50 session.addObject(new Child(george, tanya));
51
53 Person lisa = new Person("Lisa", 5, false);
53 session.addObject(new Child(lisa, tanya));
54 session.addObject(new Child(lisa, max));
55
56 session.executeRules();
57
58 // Exclude already known.
59 ObjectFilter of = new ObjectFilter() {
60
61         public Object filter(Object arg) {
62                 return arg instanceof Conclusion && ((Conclusion) arg).getDepth()==0 ? null : arg;
63         }
64
65         public void reset() { }
66
67 };
68
69 List objects = new ArrayList(session.getObjects(of));
70 Collections.sort(
71                 objects,
72                 new Comparator() {
73
74                         public int compare(Object o1, Object o2) {
75                                 return o1.toString().compareTo(o2.toString());
76                         }
77
78                 });
79
80 session.release();
81
82 Iterator it=objects.iterator();
83 for (int i=1; it.hasNext(); i++) {
84      System.out.println(i + ": "+it.next());
85 }
86
87 DOMUtils.serialize(objects, "conclusions", new File("conclusions.xml"));
88
89 Browser.showAndExitOnClose(objects, "Family relationships");
```

Description

- Lines 3-6 - check number of command line arguments.

- Lines 8-13 - a stateful rule session is created.

- Lines 15-54 - objects are added to the rule session.

- Lines 56-69 - we retrieve only inferred conclusions from the knowledge base.

- Lines 70-78 - conclusions are sorted alphabetically

- Line 80 - session is released, object bus' queue processing thread is stopped

- Lines 82-85 - conclusions are printed to console.

- Line 87 - conclusions are stored to a file in XML format.

- Line 89 - conclusions are displayed in a GUI browser.

Here is a tail of the output produced by the code above. The output shows inferred Victor's family relationships.

```
90: Victor [male 63] is a Father of Peter [male 37] (2/1)
91: Victor [male 63] is a Father of Tanya [female 31] (2/1)
92: Victor [male 63] is a GrandFather of Audrey [female 4] (3/1)
93: Victor [male 63] is a GrandFather of Dan [male 7] (3/1)
94: Victor [male 63] is a GrandFather of George [male 10] (3/1)
95: Victor [male 63] is a GrandFather of Lisa [female 5] (3/1)
96: Victor [male 63] is a GrandFather of Nancy [female 14] (3/1)
97: Victor [male 63] is a Husband of Kate [female 58] (2/1)
```
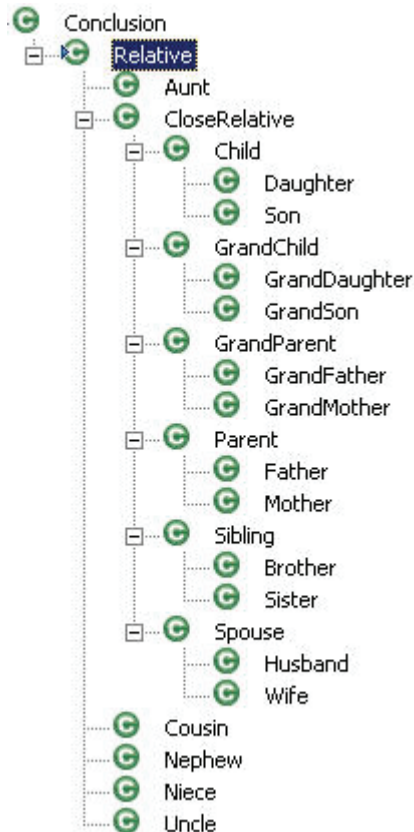

Figure 6. Conclusions class hierarchy

Figure 6. shows the hierarchy of conclusions. All conclusions in the tutorial have a constructor which takes two arguments of type `Person`. The first argument is the **subject** of the relationship

and the second is the **object**. `Relative` is the superclass for all conclusions in the tutorial. It's `toString()` method outputs conclusion as follows:

```
<subject> is a <conclusion name> of <object>
```

For example, "**Peter** is a **Husband** of **Alison**". In this example **Peter** is the subject, **Alison** is the ojbect and **Husband** is the name of relationship.

```
public class Husband extends Spouse {
        public Husband(Person subject, Person object) {
                super(subject, object);
        }

}
```

The code snippet above is an example of a conclusion class.

```
biz.hammurapi.rules.tutorial.rules
├── DaughterRule.java
│   └── DaughterRule
│       └── infer(Child)
├── GrandRules.java
│   └── GrandRules
│       ├── infer(Child, Father)
│       ├── infer(Child, Mother)
│       ├── infer(Daughter, Parent)
│       └── infer(Son, Parent)
├── ParentChildRules.java
│   └── ParentChildRules
│       ├── infer(Child)
│       └── infer(Parent)
├── ParentRules.java
│   └── ParentRules
│       └── infer(Parent)
├── SecondaryRules.java
│   └── SecondaryRules
│       ├── infer(Aunt, Parent)
│       ├── infer(Brother, Parent)
│       ├── infer(Child, Child, Sibling)
│       ├── infer(Cousin)
│       ├── infer(Daughter, Sibling)
│       ├── infer(Husband, Aunt)
│       ├── infer(Sister, Parent)
│       ├── infer(Son, Sibling)
│       ├── infer(Uncle, Parent)
│       └── infer(Wife, Uncle)
├── SelfRelativeValidator.java
│   └── SelfRelativeValidator
│       └── infer(Relative)
├── SiblingRules.java
│   └── SiblingRules
│       ├── infer(Child, Child)
│       └── infer(Sibling)
├── SonRule.java
│   └── SonRule
│       └── infer(Child)
├── SpouseRules.java
│   └── SpouseRules
│       └── infer(Spouse)
└── Validator.java
    └── Validator
        ├── minFemaleReproductiveAge
        ├── Validator()
        ├── setMinFemaleReproductiveAge(int)
        ├── validate(CloseRelative, CloseRelative)
        ├── validate(Mother)
        └── validate(Relative)
```

Figure 7. Rule classes

Figure 7. shows rules classes with their `infer()` methods. The listing below shows source code of one of rules

```java
public class SpouseRules extends Rule {

        /**
         * Male spouse is husband, female spouse is wife.
         * If A is a spouse of B then B is a spouse of A.
         * @param spouse
         * @return Brother or Sister.
         */
        public Spouse infer(Spouse spouse) {
                post(new Spouse(spouse.getObject(), spouse.getSubject()));

                if (spouse.getSubject().isMale()) {
                        if (!(spouse instanceof Husband)) {
                                return new  Husband(spouse.getSubject(), spouse.getObject());
                        }
                } else {
                        if (!(spouse instanceof Wife)) {
                                return new Wife(spouse.getSubject(), spouse.getObject());
                        }
                }
                return null;
        }
}
```

Notice the first line in the `infer()` method. This line posts a new Spouse to the bus with subject and object swapped. If that new spouse were directly dispatched to the rules it would come to this method again and it would cause an infinite loop. **Hammurapi rules** keeps a collection of weak references to objects recently posted to the object bus and detects loops of this type for subclasses of [Conclusion](). Derivation trees of equal conclusions are merged. I.e. if a conclusion is posted on the bus and there is a recent conclusion which is equals to the new one, then it means that there are several ways to come to the same conclusi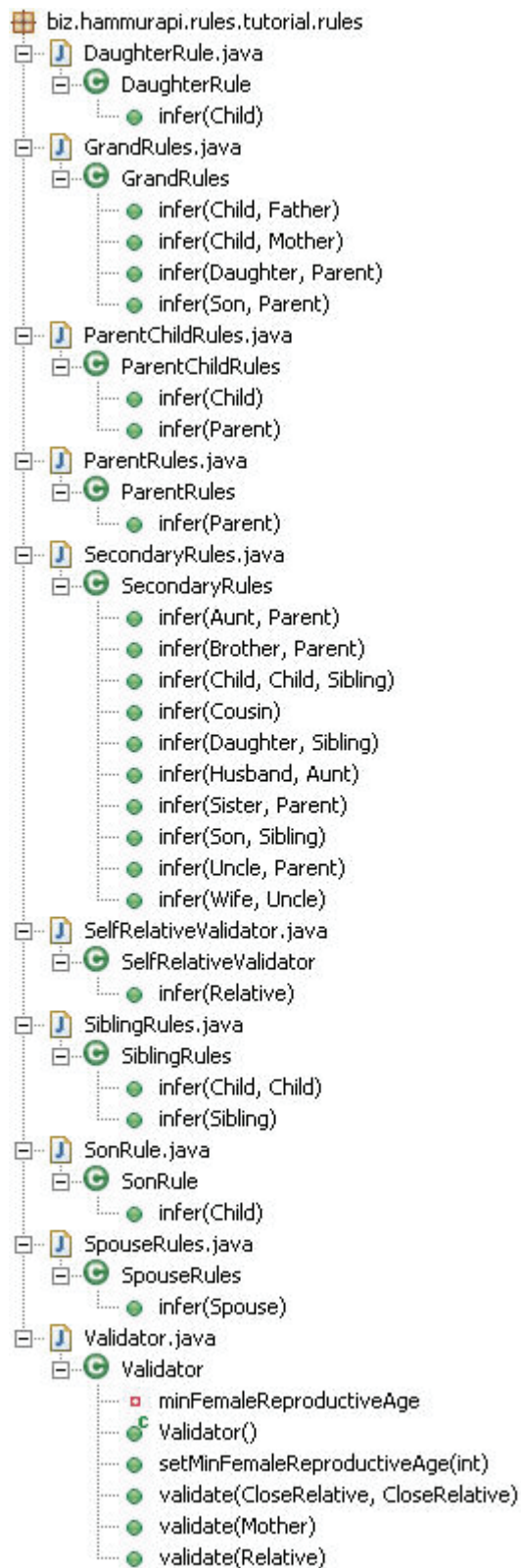on, and derivations are merged. This is important for negation because it is possible that one of conclusion's derivation is negated, but some other is not. A conclusion is negated if all of its derivations are negated.

```xml
<ruleset type="com.pavelvlasov.config.ElementNameDomConfigurableContainer">

    <name>Family ties</name>
    <description>Infers familty relationships from gender and parent/child relationship</description>

    <handle-manager type="biz.hammurapi.rules.KnowledgeMaximizingHandleManager"/>

    <collection-manager type="biz.hammurapi.rules.PojoCollectionManager">
        <collectionType>biz.hammurapi.rules.KnowledgeMaximizingSet</collectionType>
    </collection-manager>

    <rules type="biz.hammurapi.rules.QueueingRulesContainer">
        <rule type="biz.hammurapi.rules.tutorial.rules.SiblingRules">
            <name>Sibling rules</name>
            <description>Infers sibling, brother and sister relationships.</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.SpouseRules">
            <name>Spouse rules</name>
            <description>Infers wife and husband relationships.</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.ParentChildRules">
            <name>Parent and Child rules</name>
            <description>Infers parent from child and child from parent</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.ParentRules">
```

```
            <name>Parent rules</name>
            <description>Infers father and mother from parent</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.SonRule">
            <name>Son rule</name>
            <description>Infers son from child</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.DaughterRule">
            <name>Daughter rule</name>
            <description>Infers daughter from child</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.GrandRules">
            <name>Grand... rules</name>
            <description>Infers grandmother, grandfather, grandson and granddaughter</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.SecondaryRules">
            <name>Infers secondary relatives</name>
            <description>Infers ant, uncle, niece, nephew, and cousin relationships.</description>
        </rule>

        <rule type="biz.hammurapi.rules.tutorial.rules.Validator">
            <name>Validator</name>
            <description>Contains a number of different validations.</description>
            <legalAge>18</legalAge>
        </rule>

    </rules>
</ruleset>
```

Above you see the rule set definition. There are two lines in this file which requre elucidation. These are ones with `KnowledgeMaximizingHandleManager` and `KnowledgeMaximizingSet`.

Knowledge maximization (supersession) is the act of replacing less specific facts or conclusion with more specific. There is `Supercedable` interface with `supercedes()` method. This method returns true it its instance is more specific than the argument of the method.

`Conclusion` class implements `Supercedable` interface. In the case of `Conclusion`, `supercedes()` returns `true` if this conclusion class is a subclass of the argument's class and all slots are equal. For example, conclusion that *Mary is a mother of Joe* is more specific than that *Mary is a parent of Joe* and as such the first one supercedes the second.

In our case as soon as we know that "A is a father of B" we don't need to know that "A is a parent of B" because `Father` class is a subclass of `Parent` and can be used in all places where `Parent` is used.

`KnowledgeMaximizingSet` doesn't add supercedable instances to self if there are instances superceding the instance being added in the set already. In the reverese situation, if an object being added supercedes any of set elements those elementsare removed from the set. For example, if a set contains "A is a parent of B" and we add "A is a father of B" then "A is a father of B" will be added, and "A is a parent of B" will be removed from the set. On the other hand if the set contains "A is a father of B", then "A is a parent of B" will not be added to the set.

`KnowledgeMaximizingHandleManager` operates in a similar fashion, but superceded objects are not removed but "rebound". E.g. we add "A is a parent of B" to the rule engine and obtain a handle. Then the engine comes to a conclusion that "A is a father of B". The handle manager

will rebind the handle to the new conclusion. When we invoke `getObject(Handle)` for the handle obtained from adding "A is a parent of B", we'll get "A is a father of B".

Running tutorial

To run the tutorial code you need to execute `hrtutorial` command file. It takes a single argument, a rule set URI.

Direct loading of the rule set

Execute
```
hrtutorial direct:file:familyties.xml
```

to load the rule set directly from a file.

Registering the rule set

Execute
```
hradmin register familyties -f familyties.xml
```

to register the rule set and then
```
hrtutorial familyties
```

to run it. You can also try
```
hradmin dump familyties -f report.html -H
```

to produce a rule set HTML report. Then try to parameterize `Validator` by executing
```
hradmin register familyties-customized -f familyties.xml -R Validator:legalAge=16
```

Notice the difference in output when you run
```
hrtutorial familyties-customized
```

List registered rule sets

As the last exercise print a list of registered rule sets

```
hradmin list
Hammurapi Rules administrator. Copyright (C) 2006 Hammurapi Group

Rule sets registrations

URI:         familyties
Name:        Family ties
Description: Infers familty relationships from gender and parent/child relationship

URI:         familyties-customized
Name:        Family ties
Description: Infers familty relationships from gender and parent/child relationship
```

TO-DO

There are `TODO` comments in the `SecondaryRules` class. Inference of `Niece` and `Nephew` is not fully implemented. This is an exercise left for you, dear reader.

Validator

You probably noticed `Validator` class on Figure 7. This class doesn't have `infer()` methods, but rather `validate()` methods.

Here is a fragment of `Validator` code.

```
public class Validator extends Rule {

        public Validator() {
                super("validate", "remove", "accept");
        }

        /**
         * A person cannot be a relative of self.
         * @param relative
         */
        public void validate(Relative relative) {
                if (relative.getSubject().equals(relative.getObject())) {
                        post(new BadFact(relative, "A person cannot be a relative of self"));
                }
        }

...
```

The first thing to notice is that it uses non-default superclass constructor. By doing so it changes the default name for inference methods from `infer` to `validate`, which makes the rest of the code more understandable. The other thing to notice that `validate()` method posts [BadFact](#) instance to the knowledge base. This class implements [Negator](#) interface and as such negated fact and all conclusions based on this fact are removed from the knowledge base.

Validator vs. negators

In this tutorial we chose to implement validations with `Validator` class. Another option would be to have some conclusions implement `Negator` interface. For example,

- `CloseRelative` would negate any other close relative with equal slots.

- `Mother` would negate other mothers with the same subject.

- `Spouse` would negate other spouses with the same subject (in monogamic countries).

In the case of `Validator` we retained the fact which was posted earlier than the other conflicting fact. The other fact, which was posted later, was discarded. Though, in the case of negators, a fact posted later to the bus will be retained and facts which it negates will be removed from the knowledge base (unless `retain-negators` attribute of the rule set is set to "yes").

Also, in the first case all logic is located in rules, but the negators approach spreads the logic between conclusions and rules.

The rule of thumb is to put only "hard-wired" logic to fact and conclusion classes. Volatile and parameterizable logic shall reside in rules. E.g. "self-relative" validation could be implemented in `Relative` constructor so it would throw `IllegalArgumentException`. A person can have

only one biological mother and as such `Mother` shall negate other mothers with the same object.

On the other hand, beacuse legal age may be different in different countries, this validation shall be in rules, not in the `Child` or `Parent` conclusion classes.

Administrative command line utility

Hammurapi rules distribution contains a command line utility to manage rule set registrations. The name of the utility is `hradmin`. The utility provides four commands:

- **register** - Adds a rule set to the registrations file.

- **deregister** - Removes a rule set with a given URI from the registrations file.

- **list** - produces a report about registered rule sets in text, XML or HTML format. Outputs the report to console or to a file.

- **dump** - produces a report about a registered rule set with a given URI in text, XML or HTML format. Outputs the report to console or to a file. .

If you run `hradmin` with no arguments or wrong arguments it outputs the following information

```
Hammurapi Rules administrator. Copyright (C) 2006 Hammurapi Group
Unrecognized command: -h
Usage: hradmin command [options]
        Commands:
                register <uri> {-f <file> | -u <url> | -r <url>} [-p <name>=<value>] [-R <rule
name>:<property name>=<property value>] [-o <object filter>]- Registers rule set.
                deregister <uri> - Unregisters rule set.
                list [-f <file>] {-x|-H} - Lists names of registered rule sets to console or file.
                dump <uri>  [-f <file>] {-x|-H} - Outputs ruleset definition to console or file.

        Execute 'hradmin command -h' for command-specific options
```

register

This command registers a rule set. It can read rule set from a file or a URL. Rule sets provided as URL's can be stored by value, i.e. The XML definition will be copied to the registrations file, or by reference i.e. the registrations file will contain the URL per se but not the XML definition. This allows to maintain centralized rulesets without the need to update registrations every time the XML definition changes.

`hradmin register -h` outputs

```
Hammurapi Rules administrator. Copyright (C) 2006 Hammurapi Group
usage: hradmin register <uri> [options]
 -R <rule name>:<key>=<value>   Rule property
 -f <file>                      Ruleset file
 -h                             Print this message
 -o <class name>                Default object filter
 -p <key=value>                 Rule set property
 -r <URL>                       Ruleset URL to be registered by reference
 -u <URL>                       Ruleset URL
```

Options `f`, `r` and `u` are mutually exclusive. At least one of them must be provided. Administrator can specify multiple rule set and rule properties. These properties are passed to the rule set at runtime. It allows to provide user-specific parameterization of rule sets and rules at registration time. When provided from the command prompt both key and value are treated as instances of `java.lang.String`. If you need to provide properties of other types you would need to use the API or directly edit the registrations file. Object filter and rule properties

cannot be set for rule sets registered by reference. Rule set properties can be set for any registration type.

deregister

Removes rule set registration.

`hradmin deregister -h` outputs

```
Hammurapi Rules administrator. Copyright (C) 2006 Hammurapi Group
usage: hradmin deregister <uri> [options]
 -h   Print this message
```

list

This command outputs a list of registered rule sets in text, html or xml format.

`hradmin list -h` outputs

```
Hammurapi Rules administrator. Copyright (C) 2006 Hammurapi Group
usage: hradmin list [options]
 -H          Output in HTML format
 -f <file>   Output file
 -h          Print this message
 -x          Output in XML format
```

If no options are provided then the command outputs formatted text to the console.

dump

The `dump` command outputs registered rule set details in different formats. If you run `hradmin dump -h` a help message appears

```
Hammurapi Rules administrator. Copyright (C) 2006 Hammurapi Group
usage: hradmin dump <uri> [options]
 -H          Output in HTML format
 -f <file>   Output file
 -h          Print this message
 -x          Output in XML format
```

If no options are provided then the command outputs formatted text to the console.

JSR-94 implementation notes

Because **Hammurapi rules** is an implementation of JSR-94, it is recommended to get familiar with the JSR-94 Specification. Further material assumes reader's familiarity with the specification and concentrates on describing the most important details of **Hammurapi rules'** implementation of JSR-94. See Hammurapi Rules JavaDoc for more information.

Technology compatibility kit

JSR-94 Technology compatibility kit (TCK) implementation for **Hammurapi rules** can be downloaded from the **Hammurapi rules** download page.

The JSR-94 TCK itself is part of the JSR-94 distribution and can be downloaded from the JSR-94 home page.

Download and unzip the JSR-94 package and then download and unzip the TCK for Hammurapi Rules. Copy over directories ant and lib from the Hammurapi Rules TCK to the JSR-94 directory. This operation will overwrite three files in the lib folder: tck.conf, tck_res_1.xml, and tck_res_2.xml.

After the download and copying files run the TCK as described in the JSR-94 documentation (jsr94_tck.pdf).

The example below shows how to do it on Windows.

```
C:\jsr94-1.0>set ANT HOME=ant

C:\jsr94-1.0>ant\bin\ant -f run_tck.xml
```

We executed TCK tests in our environment using JDK 1.5. With JDK 1.5 tests report signature analysis failure because of new methods added to `java.lang.Exception`.

TCK tests failed on JDK 1.4 because of conflicts in XML libraries between Hammurapi Rules (it depends on `javax.xml.xpath` classes) and Ant 1.4.1 libraries shipped with the TCK.

Rule Service Provider

The `javax.rules.RuleServiceProvider` class implements a single point of access to both runtime and administration interfaces. The rule service provider implementation class for **Hammurapi rules** is `biz.hammurapi.rules.jsr94.FileRuleServiceProvider` The code snippet below shows how to obtain rule service provider.

```
Class.forName("biz.hammurapi.rules.jsr94.FileRuleServiceProvider");
RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider("biz.hammurapi.rules.jsr94.FileRuleServiceProvider");
```

This provider stores registrations in `.hammurapi-rules/rules.xml` file in user home directory.

RuleRuntime

createRuleSession(String uri, Map properties, int type)

This method creates a rule session. If `uri` starts with `direct:` then the rest of the uri is treated as rule set source URL and the registrations file is bypassed. Direct loading might be needed if you distribute rule sets bundled with your product. In this case you can store them as classloader resource and use `direct:resource:<rule set path>` URI to load the ruleset at runtime. Another scenario is to host rule sets on your web site and use them directly from your product without having users to explicitly register rule sets in order to make your product operational. Examples:

- `direct:`http://www.hammurapi.biz/someruleset.xml will load rule set directly from the specified URL.

- `direct:resource:biz/hammurapi/somepackage/someruleset.xml` will load rule set from a classloader resource.

XML definition of rule set is instantiated by the means of DomConfigFactory. The instantiated object must implement `com.pavelvlasov.config.Context` to provide naming services. It is recommended to use ElementNameDomConfigurableContainer, a subclass of DomConfigurableContainer as the type of the root rule set element.

Example:
```
<ruleset type="com.pavelvlasov.config.ElementNameDomConfigurableContainer">
    <name>Family ties</name>
    <description>Infers familty relationships from gender and parent/child relationship</description>

    <rules type="...">
        <handle-manager type="..."/>

        <collection-manager type="..."/>

        <rule type="...">
            <name>Grandmother</name>
            <description>Infers 'grandmother' relationship from 'mother' and 'child'
relationships.</description>
            ...
        </rule>

        ...
    </rules>
</ruleset>
```

Properties set at registration time and properties passed in `properties` parameter are set as attributes in the instantiated object. For this the instantiated definition must implement com.pavelvlasov.util.Attributable. Values of properties provided in `properties` parameter override those set at the registration time with matching keys. Rule set components can access properties using naming bus. E.g. property `foo` can be accessed using `get("/@foo")` method invocation.

NOTE: Though it is possible to directly instantiate `biz.hammurapi.rules.jsr94.RuleSession` this approach is not JSR-94 compliant.

RuleSession

Hammurapi rules' rule session implements both [stateful](#) and [stateless](#) session interfaces in the same class. Stateless' session `executeRules()` methods are wrappers around stateful session methods. This class requires the following components be present on the naming bus:

- `/handle-manager` of type [`biz.hammurapi.rules.HandleManager`](#).

- `/collection-manager` of type [`biz.hammurapi.rules.CollectionManager`](#).

- `/rules` of type [`biz.hammurapi.rules.KnowledgeBase`](#) KnowledgeBase is a facade for the Object bus.

- `/name` and `/description` of type String.

Optional components:

- `/object-filter` of type [`javax.rules.ObjectFilter`](#).

- `/knowledge-compactor` This entry shall be an implementation of [KnowledgeCompactor](#). It is used to pre-process object collections before returning them from `RuleSession.getObjects()`. Unlike ObjectFilter this interface works on entire collection of facts and conclusions.

These requirements essentially translate into the following XML structure of the rule set:

```xml
<ruleset type="com.pavelvlasov.config.ElementNameDomConfigurableContainer">
    <name>Family ties</name>
    <description>Infers familty relationships from gender and parent/child relationship</description>

    <handle-manager type="class which implements HandleManager"/>

    <collection-manager type="class which implements CollectionManager"/>

    <object-filter type="class which implements object filter"/>

    <rules type="class which implements KnowledgeBase">

        <rule type="...">
            <name>Grandmother</name>
            <description>Infers 'grandmother' relationship from 'mother' and 'child'
relationships.</description>
            ...
        </rule>

        ...
    </rules>
</ruleset>
```

There can be other components on the naming bus. E.g. one might want to add a thread pool for multithreaded inference. If rule engine components need to reach application objects outside of the rule engine container it can be done by injecting those object as container attributes by passing them in the `properties` parameter to the `createSession()` method.

Methods

This is an excerpt from JavaDoc describing **Hammurapi rules'** implementation details regarding rule session. For information about other classes see [Hammurapi rules JavaDoc](#).

executeRules()

This method delegates to `KnowledgeBase`.`executeRules()`. Normally rules are executed automatically when `addObject()` or `updateObject()` methods are invoked. Invocation of this method is required to make sure that all inference threads finished inference process and the internal object bus queue is empty.

executeRules(List objects)

Calls executeRules(object, null)

executeRules(List objects, ObjectFilter objectFilter)

This method is a wrapper around stateful method. It adds all objects to the session, executes rules, returns objects and finally resets the session.

Admin

RuleAdministrator

getRuleExecutionSetProvider(Map properties)

Creates new RuleExecutionSetProvider. Properties are ignored.

getLocalRuleExecutionSetProvider(Map properties)

Creates new LocalRuleExecutionSetProvider. Properties are ignored.

registerRuleExecutionSet(String uri, RuleExecutionSet res, Map properties)

Registers a rule execution set.

Rule sets can be registered by value or by reference. When registered by value XML definition of the rule set is loaded into `registrations.xml` file. When a rule set is registered by reference then its URI is stored in the registrations file. To store a rule set by reference `properties` argument of `registerRuleExecutionSet(String uri, RuleExecutionSet res, Map properties)` shall contain an entry with the key `by-reference` and the value `Boolean.TRUE`. In order to be storable by reference a rule set must be created by `RuleExecutionSetProvider.createRuleExecutionSet(String uri, Map properties)` method. The code fragment below shows how to store a rule set by reference.

```
String ruleServiceProviderClassName = "biz.hammurapi.rules.jsr94.FileRuleServiceProvider";
Class.forName(ruleServiceProviderClassName);
```

```
RuleServiceProvider serviceProvider =
RuleServiceProviderManager.getRuleServiceProvider(ruleServiceProviderClassName);
RuleAdministrator administrator = serviceProvider.getRuleAdministrator();
RuleExecutionSetProvider resp = administrator.getRuleExecutionSetProvider(null);
RuleExecutionSet res = resp.createRuleExecutionSet("http://www.mysite.com/myrules.xml", null);
Map properties=new HashMap();
properties.put("by-reference", Boolean.TRUE);
administrator.registerRuleExecutionSet("family ties", res, properties);
```

deregisterRuleExecutionSet(String uri, Map properties)

Removes a rule set with a given uri from the registration file. Properties are ignored.

LocalRuleExecutionSetProvider

`properties` map passed to `createRuleExecutionSet()` methods gets XML-ized to the registration file and then is reinstantiated and injected into the runtime rule set. It allows to provide registation time parameterization of rule sets. It is recommended to use Java primitive types or collections/maps from `java.util` package for map's keys and values. Rule execution sets created by `LocalRuleExecutionSetProvider` cannot be registered by reference.

createRuleExecutionSet(Object holder, Map properties)

`holder` must be of type `org.w3c.Element`.

RuleExecutionSet

Only rule execution sets created by `RuleExecutionSetProvider.createRuleExecutionSet(String uri, Map properties)` can be registered by reference. If `setDefaultObjectFilter()` is invoked of such a rule execution set or `setProperty()` is invoked on one of rule set rules before the invocation of `registerRuleExecutionSet()` then such a rule execution set cannot be registered by reference.

RuleExecutionSetProvider

createRuleExecutionSet(String uri, Map properties)

If `uri` starts with `resource:` then the rest of the `uri` is treaded as a resource path in classloader. Otherwise the `uri` is treated as URL. Only rule execution sets created by this method can be registered by reference unless `setDefaultObjectFilter()` method or `setProperty()` methods of one of rule has been invoked before the invocation of `registerRuleExecutionSet()`.

createRuleExecutionSet(Serializable xmlString, Map properties)

The first parameter must be of type `java.lang.String` and contain XML defition of a rule set.

`properties` map passed to `createRuleExecutionSet()` methods gets XML-ized to the registration file and then is reinstantiated and injected into the runtime rule set. It allows to provide registation time parameterization of rule sets. It is recommended to use Java primitive types or collections/maps from `java.util` package for map's keys and values.

Methodology

This section outlines steps and roles involve in development and usage of **Hammurapi rules** rule sets. Please note that one person can play multiple roles and one role can be played by multiple people.

Roles

- **Object model developer** - develops application object model. If the object model is generated or already exists then this role is not required.

- **Application developer** - develops application code. This person(s) should know JSR-94 runtime API and the application object model which is used by rules.

- **Rules developer** - people in this role develop rules and they should know the application object model which will be used by rules and **Hammurapi rules** rule authoring.

- **Rule set assembler** - creates rule set XML definitions, provides rule parameters, adjusts QoS by selecting proper implementations of the engine components. This role requires knowledge or rule parameters (e.g. from JavaDoc generated from rule classes) and **Hammurapi rules**' components.

- **Rule administrator** - registers rules on local machines and provides registration time parameters. Names and possible values of supported registration time parameters shall be supplied by the rule set assembler.

Steps

- Develop application object model. When the object model is ready rule developers can start working on rules and application developers can work on the application code. These two activities are independent because the application code is decoupled from rules by JSR-94 interfaces.

- Rule developers develop rules.

- Rule assemblers assemble rules into rule sets.

- Application developer develop application code.

- Rule sets are registered and parameterized by the rules administrator (or automated script) as part of the application installation procedure.

Support of specialized rule languages

Although there are many reasons to use Java as a language for rule authoring (see Occam's Razor overarching principle), there are situations where it is justified to use a specialized rule engine. Such situations include:

- Rules use a limited set of domain objects and operations. Rules don't need to access underlying Java application resources such as making SQL queries or sending JMS messages. There is (going to be) a lot of rules and therefore benefits of concisness of specialized rule engine overweight expenses associated with the introduction of the new rules language into development enviorinment.

- You are migrating rules from existing rule engine solution to **Hammurapi rules**. There are already people familiar with the specialized rules language and there is a significant number of rules written in that language. In this case it maybe cheaper/less risky to develop a language module for **Hammurapi rules** than to retrain people and rewrite rules.

A specialized rule engine can be introduced into **Hammurapi rules** in two ways:

- Rules defined in a specialized language are evaluated at runtime by a subclass of AbstractRule

- JSP-like approach. Rules defined in a specialized language are compiled to Java source files, which in turn get compiled to class files.

One rule set can contain rules defined in different languages.

Custom rule example

Code snippets below show how to implement the first approach of introducing a specialized rules language to **Hammurapi rules**.

```
public class MyRule extends AbstractRule {
        private String definition;
        private Collection invocationHandlers;
        private Collection removeHandlers;

        public void setDefinition(String definition) {
                this.definition=definition;
        }

        public Collection getRemoveHandlers() {
                return removeHandlers;
        }

        public Collection getInvocationHandlers() {
                return invocationHandlers;
        }

        public void start() throws ConfigurationException {
                super.start();
                ... Parse/compile definition and create invocation handlers ...
        }
}
```

Rule class.

```
<rule type="mypackage.MyRule">
    <name>...</name>
    <description>...</description>
    <definition>
        ... Rule definition in a specialized rule language ...
    </definition>
</rule>
```

Rule XML definition.

Specialized rules language for the tutorial

Here we'll show an example of a specialized rules language which could be used in the tutorial. Tutorial rules use a small set of classes and operations. It would make sense to create this specialized language if we planned to develop hundreds or thousands of rules and our rule developers weren's Java-savvy.

Language constructs

All types in the language belong to a predefined package. Type names start with upper case leter, variable names start with lower case letter.

Name	Defintion	Example	Remarks
Rule definition	**with** *context* [**if** *condition*] **then** *conclusions*.	with Child childOne and Child childTwo if object of childOne equals to object of childTwo then childOne is Sibling of childTwo and childTwo is Sibling of childOne	
Context	variable definition [**and** variable definition]*	Child child and Parent parent	
Variable definition	<Type> <name>	Child child	
Condition	and, or, equals [to], is	object of childOne equals to object of childTwo	**equals [to]** is equivalent of Java **equals()**, **is** is equivalent to Java **instanceof**
Conclusions	conclusion [**and** conclusion]*	childOne is Sibling of childTwo and childTwo is Sibling of childOne	
Conclusion	<person 1> is	subject of childOne is	This construct is equivalent to Java

	<Relationship type> of <person 2>	Sibling of subject of childTwo	new <Relationship type>(<person 1>, <person 2>)
Member access	<member> of <instance>	subject of childOne	Conclusion can be implicitly converted to its subject for new conclusion construction. `childTwo is Sibling of childOne` is the same as `subject of childTwo is Sibling of subject of childOne`

Example

Compare a ternary Cousin rule defined in Java and the specialized language

```
public void infer(Child child1, Child child2, Sibling sibling) {
        if (child1.getObject().equals(sibling.getSubject()) &&
child2.getObject().equals(sibling.getObject())) {
                post(new Cousin(child1.getSubject(), child2.getSubject()));
        }
}
with
    Child child1 and Child child2 and Sibling sibling
if
    object of child1 equals to subject of sibling and object of child2 equals to object of sibling
then
    child1 is Cousin of child2
```

Implementation notes

If you decide to implement a parser for this language keep in mind that the language grammar is not context-free. Context, condition, and conclusions clauses' grammars are different. In case of ANTLR you'd need one lexer, three parsers and a multiplexor to switch parsers on **with**, **if** and **then**.

If you successfullty implement the parser and decide to share it with the rest of the world we'll gladly host it on our web site or provide a link from our web site to the implementation.

Rete algorithm

Rete algorithm is an efficient pattern matching algorithm for implementing rule-based ("expert") systems. The Rete algorithm was designed by Dr. Charles L. Forgy of Carnegie Mellon University in 1979. Rete has become the basis for many popular expert systems.

This section describes **Hammurapi rules** in terms of the Rete algorithm.

This section is based on the description of the Rete algorithm in Charles Forgy's article "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem" published in the Artificial Intelligence magazine in 1982, pp 17-37. This article is referenced in the text as [RETE].

As an example we will use a rule engine which comes to dinner ideas based on items in a grocery store. The engine will contain the following three rules:

- If food is fish and wine is white and wine age is above 2 years then it is a dinner idea.

- If food is meat and wine is red and wine age is above 3 years then it is a dinner idea.

- If food is meat and wine is merlot and wine age is above 2 years then it is a dinner idea.

We'll build a Rete network for each of rules and describe how the network is "physically" implemented in **Hammurapi rules** Java constructs. Then we'll build a combined network for all three rules and explore its physical implementation.

Disclaimer: *The rules were created for demonstration purposes only and do not constitute any culinary advice.*

Figure 8 depicts inference network for a dinner with fish and white wine. Input objects are passed to the root node, which discriminates food and wine. There are five intra-nodes `Food, Fish, Wine, WhiteWine, Age > 2 years`, and one inter-node `Dinner`, which is also a terminal node.
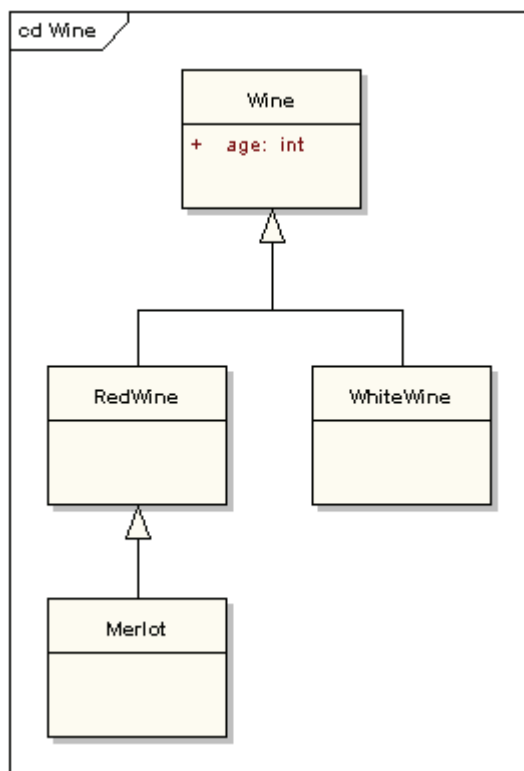
The original Rete description has no notion of inheritance. Each object has a type and the root node is an object type discriminator. In Java we have a luxury of inheritance and **Hammurapi rules** takes advantage of it as will be shown below. Figures 9 and 10 show wine and food class hierarchies which will be used in the example. Our conclusion class is `Dinner`. It is shown on Figure 11.
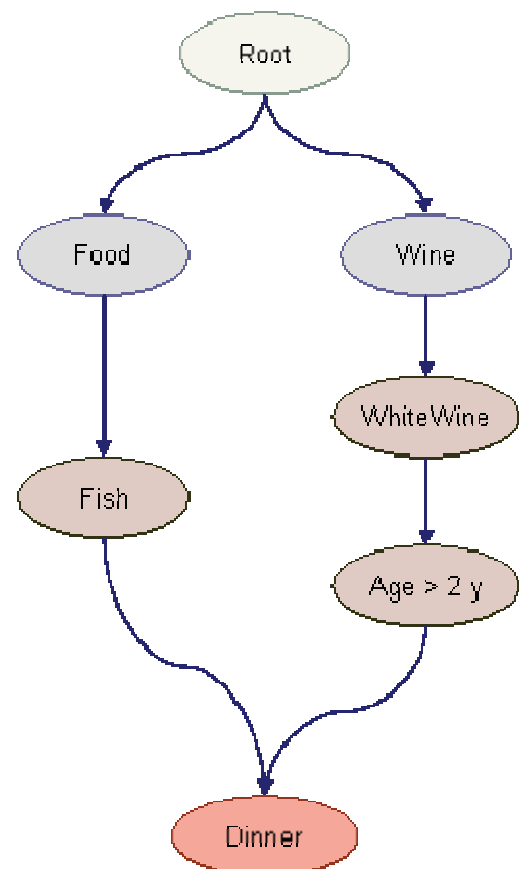
Figure 8. Inference network for a dinner with fish and white wine

Figure 9. Wine class hierarchy
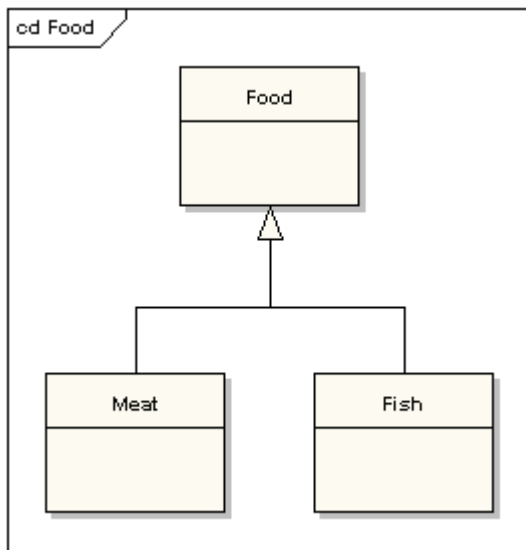
Figure 10. Food class hierarchy
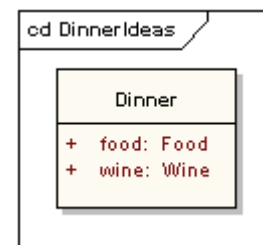
Figure 11. Dinner class
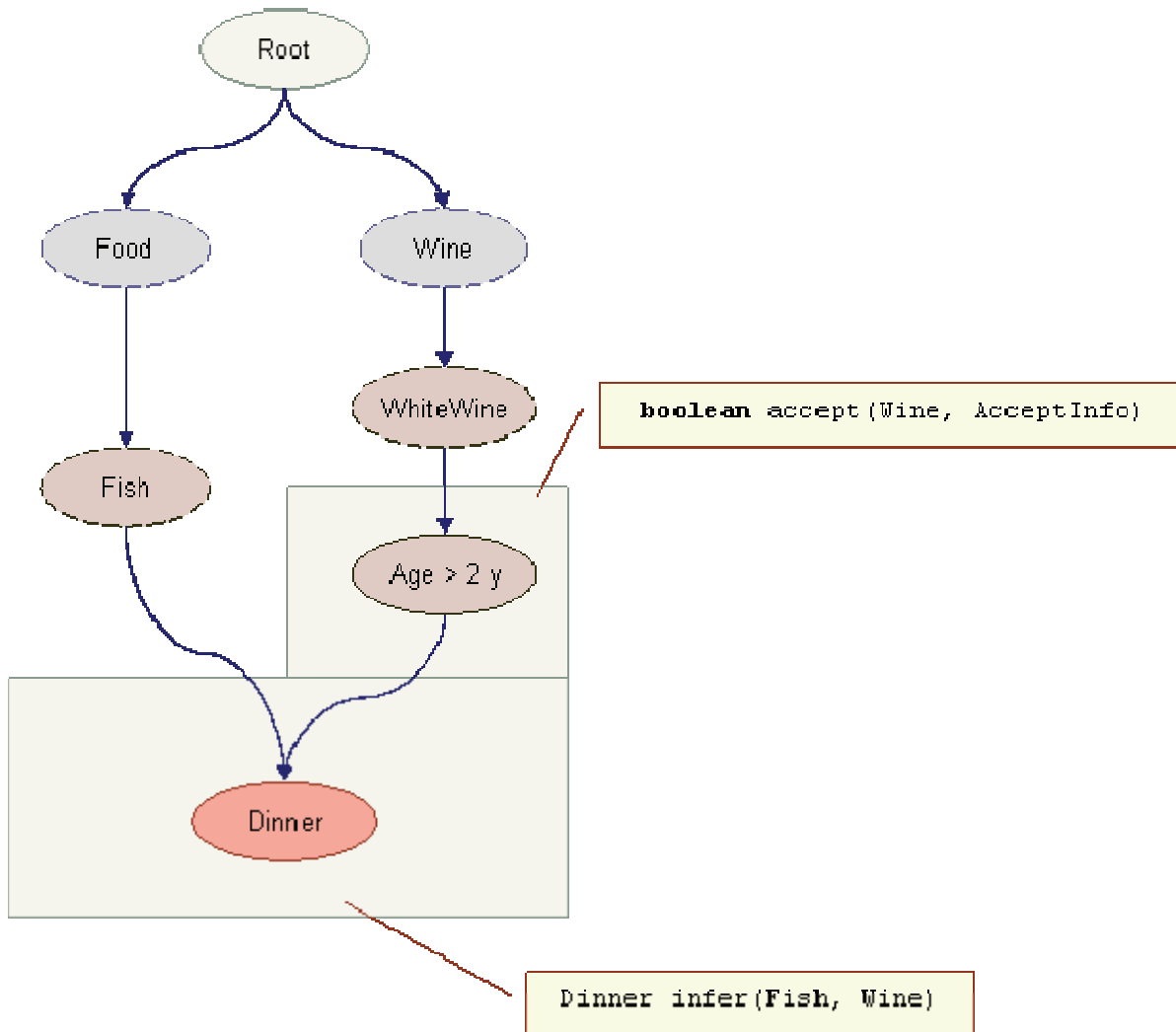
Figure 12. Mapping of fish dinner inference network to Java constructs

Figure 12 shows mapping of the logical inference network shown on Figure 8 to "physical" implementation in **Hammurapi rules**. The first thing to note is that **Hammurapi rules** takes care of nodes `Food`, `Fish`, `Wine`, and `WhiteWine` by using Java type system and method signatures. Implementation of nodes `Age > 2 years` and `Dinner` is shown below.

```
/**
* Age > 2 years node
*/
public boolean accept(WhiteWine wine, AcceptInfo acceptInfo) {
    return wine.age > 2;
}

/**
* Dinner node
*/
public Dinner(Fish fish, WhiteWine whiteWine) {
    return new Dinner(fish, whiteWine);
}
```
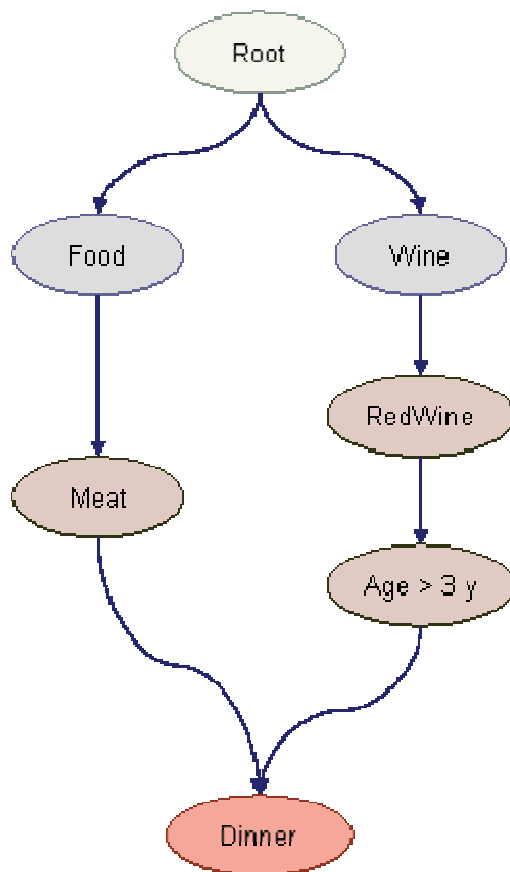
Figure 13. Inference network for a dinner with meat and red wine
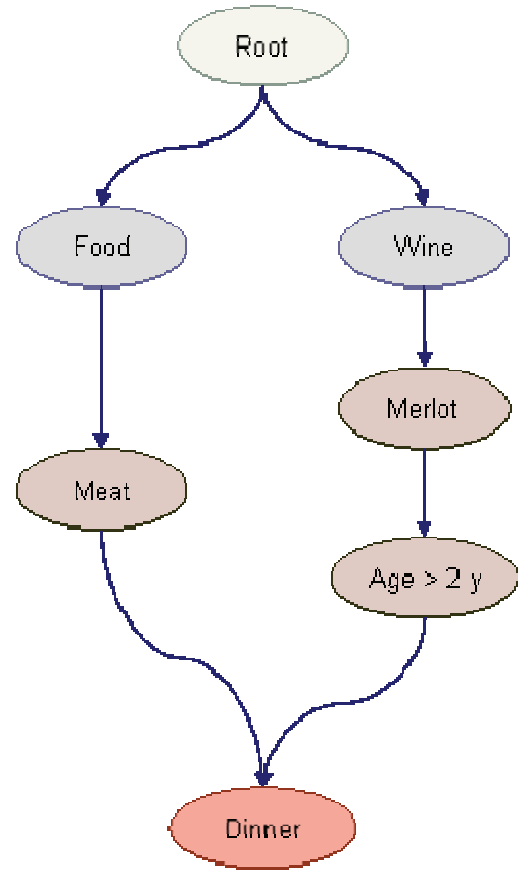
Figure 14. Inference network for a dinner with meat and red wine

Figures 13 and 14 show inference networks for the two remaining rules
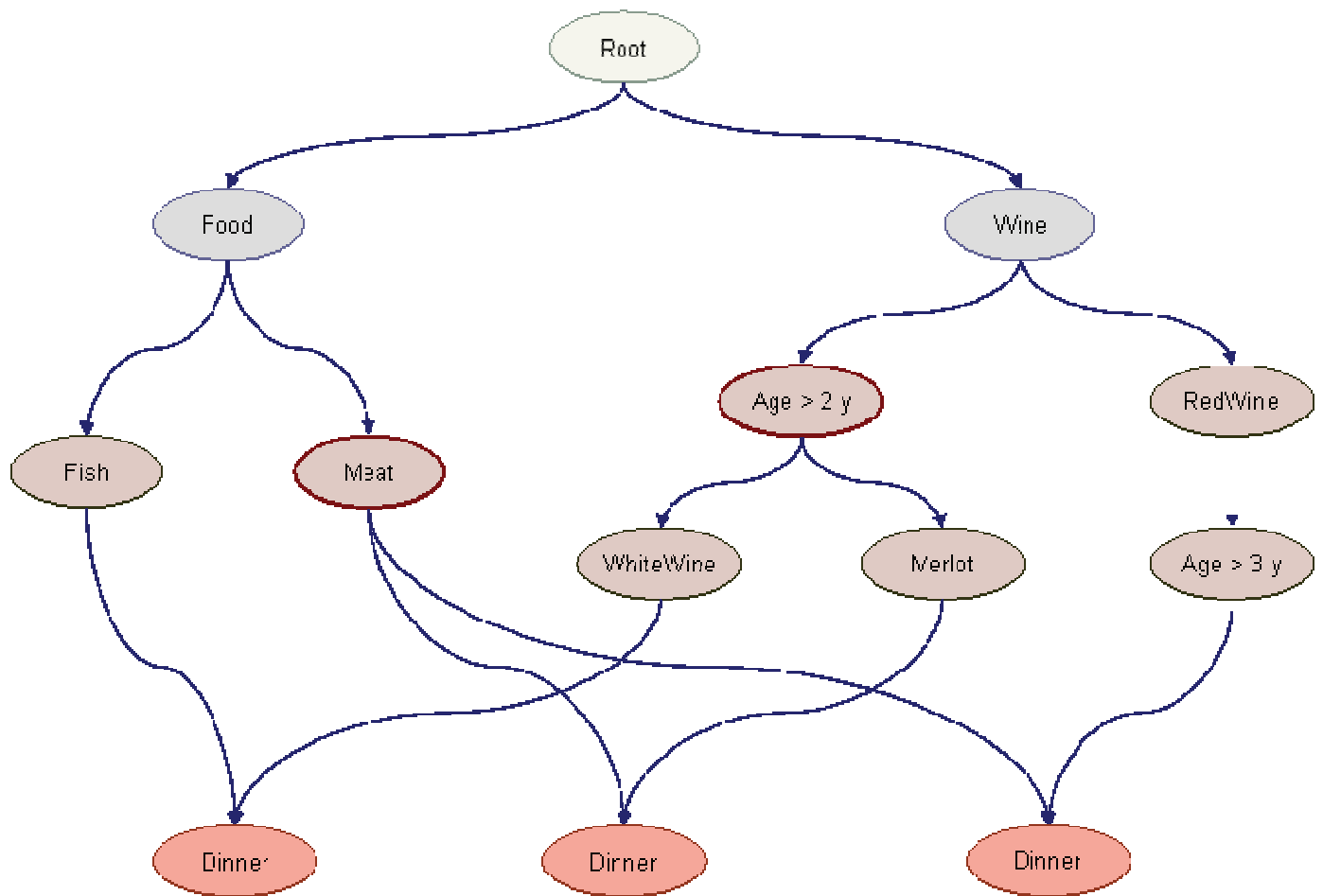
Figure 15. Combined inference network

and Figure 15 shows the combined network for all three rules. `Meat` and `Age > 2 y` are shared nodes. As you can see, we swapped `Age > 2 y` with preceding nodes to make `Age > 2 y` a shared node. We did it assuming that computational cost of determining wine age is about the same as the cost of determining wine type, and, therefore, we can leverage commutativeness of the `and` operation in order to create a shared node. In Java it is often not true. Also `and` and `or` operations are commutative in logic, but "not exactly" in Java. For example, in Java the right operand of `&&` expression is not evaluated if the left operand evaluates to `false`. As such swapping operands can lead to performance degradation or to an exception.
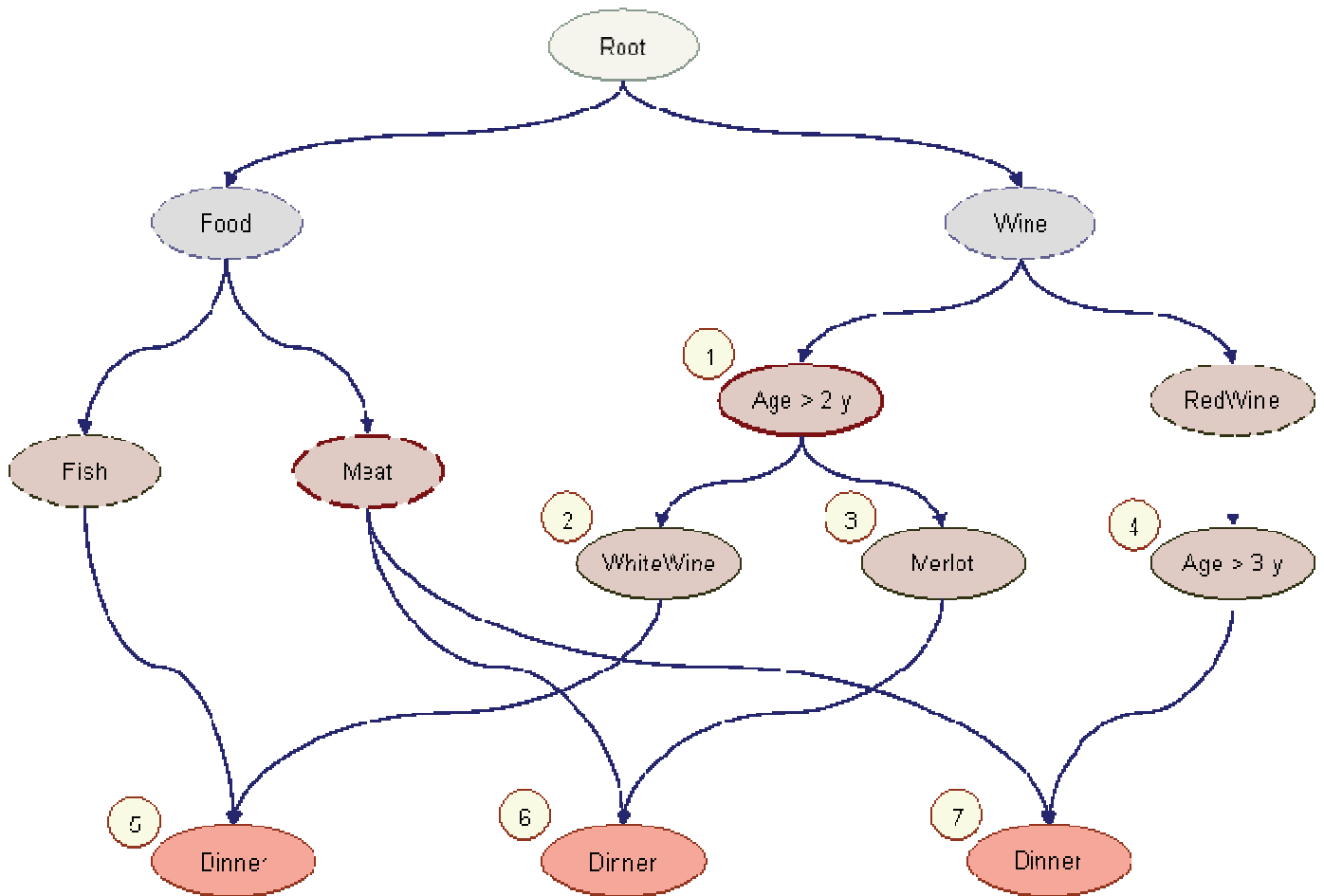
Figure 16. Mapping of combined inference network to Java constructs

Figure 16 shows mapping of the combined infernce network to implementation in **Hammurapi rules**. Dashed nodes don't need any implementation, **Hammurapi rules** takes care of these nodes by dispatching input objects only to `accept()` and `infer()` methods with compatible parameters.

Because we have a shared node `Age > 2 years`, we need a class to represent output from this node. This class is shown on figure 17.

`WineOlderThanTwoYears` class implements `Fact` interface and its `isPrivate()` method returns true. This tells **Hammurapi rules** that instances of this class shall be internal to the engine. I.e. they shall not be added to the handle manager and made available to engine's clients. Here is the implementation of `Age > 2 years` node (#1)

```
public WineOlderThanTwoYears infer(Wine wine) {
    return wine.age > 2 ? new WineOlderThanTwoYears(wine) : null;
}
```


Figure 17. Private conclusion class

In many problem domains number of shared condition (internal conclusion) classes will grow as number of rules grows. Therefore, manual discovery and implementation of shared condition classes may

become cumbersome. If you reach this point it is an indicator that it is time to switch to a specialized, domain-specific, rules language. The language compiler shall automatically discover shared conditions and generate classes for them. Generation can be done either at build time, startup time or runtime. Bytecode generation article describes how to generate Java bytecode and inject it into JVM at runtime.

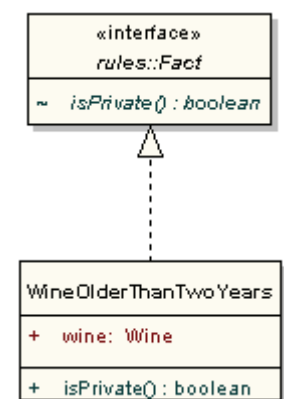You will still be able to use existing rules written in Java after you switch to a specialized rules language. You will also be able to combine rules written in different languages. This feature allows to start development of rules in Java leveraging organization's existing Java skill pool. Initial implementation of rules will allow to better understand the problem domain and come up with an effective grammar and implementation for a specialized language if there is a need in such a language.

Implementation of nodes 2 - 7 is shown below.

```java
/**
* Node # 2
*/
public boolean accept(WineOlderThanTwoYears wotty, AcceptInfo acceptInfo) {
    return wotty.wine instanceof WhiteWine;
}

/**
* Node # 5
*/
public Dinner(Fish fish, WineOlderThanTwoYears wotty) {
    return new Dinner(fish, wotty.wine);
}
```

```java
/**
* Node # 3
*/
public boolean accept(WineOlderThanTwoYears wotty, AcceptInfo acceptInfo) {
    return wotty.wine instanceof Merlot;
}

/**
* Node # 6
*/
public Dinner(Meat meat, WineOlderThanTwoYears wotty) {
    return new Dinner(fish, wotty.wine);
}
```

```java
/**
* Node # 4
*/
public boolean accept(RedWine wine, AcceptInfo acceptInfo) {
    return wine.age > 3;
}

/**
* Node # 7
*/
public Dinner(Meat meat, RedWine redWine) {
    return new Dinner(meat, redWine);
}
```

Negations and retractions

In the Rete algorithm when an object is retracted (removed) from the working memory, it traverses the network as a "minus" token (see Section 2.2.4 in [RETE]) and when it matches a pattern instantiation is removed from the conflict set instead of being added as in the case of "plus" token. Object modification is handles are removal and addition of the object from/to the working memory. The problem with this approach is that it stipulates immutability of the object (see Section 6 in [RETE]). In other words when we change an object (say change wine age from 4 to 1) then the object will not match patterns it matched before and will not be properly removed from the conflict set. Taking into account that in Java conditions can involve not only object's attributes but also related objects (e.g. Account condition may use account transactions), implementation of retraction as described in [RETE] would require the whole object model be immutable.

Hammurapi rules uses negators to retract objects from conflict sets and working memory. Conclusion instances are linked to the input objects they were inferred from. When an object is removed from the working memory it doesn't have to match patterns again (because it might not match any or match wrong ones because object state was changed).

There is one more method to handle retractions - action tracing. It is similar to "Undo" functionality in text editors - on each `infer()` method invocation **Hammurapi rules** records remove and post actions resulted from the method and builds a map Object -> Actions. When the object is removed from the working memory, `undo()` methods of associated actions are invoked.

Summary

- The way how **Hammurapi rules** leverages Java type sytem obviates impelmentation of a good deal of logical nodes.

- Shared and terminal intra-object nodes are implemented by single-argument `infer()` methods.

- Intra-object nodes which are inputs to inter-object nodes are implemented as `accept()` methods.

- Inter-object nodes are implemented as multi-parameter `infer()` methods.

- Linear network fragments can be collapsed during implementation into one node (i.e. `infer()` or `accept()` method). See Section 3.2.2 in [RETE].

- Rete algorithm poses a number of constraints on the object model (see Sections 1, 3.1, and 6 in [RETE]). These constraints are difficult to satisfy in real-life object models. **Hammurapi rules** poses no constraints on the object model.

- Assertion part of **Hammurapi rules** is exact implementation of Rete algorithm, but retraction part is not in order to remove immutability of the object model requirement.

- Hangle manager corresponds to the working memory.

- Collection manager corresponds to the conflict set.

- `post()` method corresponds to MAKE action.
- `update()` method corresponds to MODIFY action.
- `remove()` method corresponds to REMOVE action.

Backward chaining

So far we talked about *forward chaining,* a.k.a. *forward reasoning*. Starting from version 3.x **Hammurapi Rules** supports *backward chaining*, a.k.a. *backward reasoning*.

Forward chaining starts with a set of source facts and produces conclusions using rules a rule set. Backward chainng starts with a set of conclusion types it needs and then works backwards to find rules and facts to produce conclusions of required type.

Example

Hammurapi Rules tutorial shows how to construct family tree from two source fact types – `Child(Person, Person)` and `Spouse(Person, Person)`. The rule set has 25 rules and 27 conclusion types. The forward chaining part of the tutorial takes input of 17 source facts (3 spouses, 14 children) and infers 98 family relationships.

What if all we needed was `GrandMother` relationships (6 out of 98)? In this case we would have wasted a lot of computing resources by

- Inferring 92 conclusions only to discard them.

- Firing 23 rules a number of times unnecessarily, as only two rules are needed to infer `GrandMother` relationship – `ParentRules.infer(Child)` and `GrandRules.infer(Child, Mother)`.

- Adding `Spouse` to input, as it is not used in `GrandMother` inference.

In this situation backward chaining is much more effective. It works in the following way:

- Client code requests conclusions of type `GrandMother` from a rule session.

- The rule session finds rules and facts sources attached to the session which produce facts of requested type. In our case it is `GrandRules.infer(Child, Mother)`

- Then the rules session finds fact source and rules which produce `Child` and `Mother`.

- There is a facts source which produces `Child` and there is `Parent` rule which infers `Mother` from `Child` input.

- As a result, the backward chaining rule session builds an inference chain to infer only `GrandMother` conclusions as shown on the picture below.
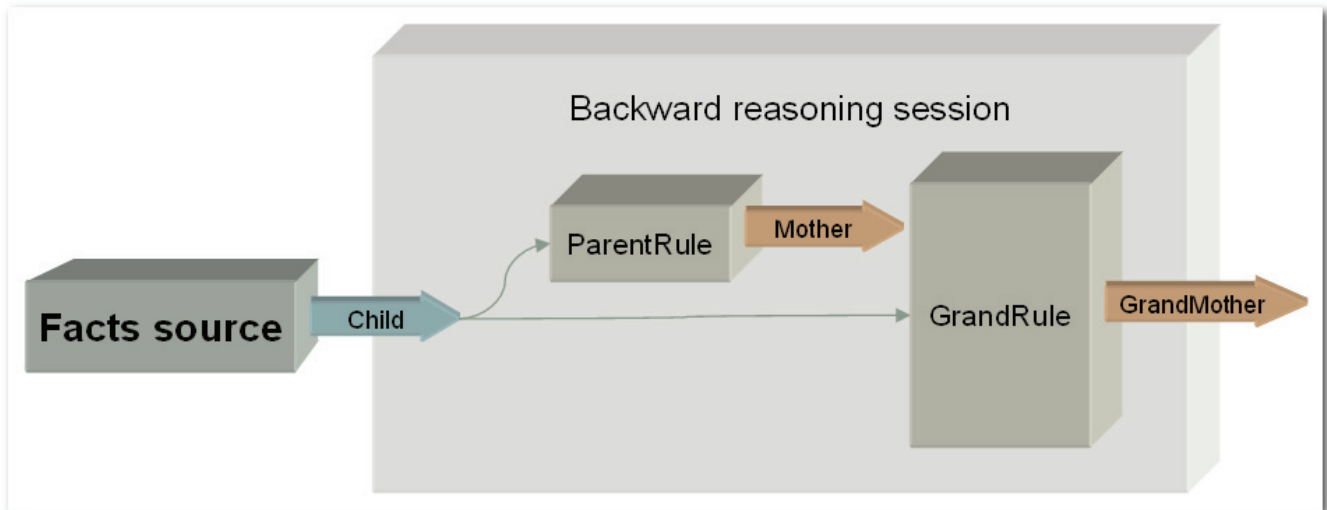
Figure 2 Backward chaining.

Implementation

There is no standard Java API for backward chaining rules engines at the moment. Therefore, Hammurapi rules employs a simple proprietary API.

Backward chaining classes reside in `biz.hammurapi.rules.backwardreasoning` package. Backward chaining leverages forward chaining and dispatching classes. There are only three classes in the backward reasoning package.

The same rules and rulesets can be used for forward and backward chaining. If rules use `post()` method, then they might need to invoke setMethodFactTypes() method from constructor. Below is JavaDoc for the method.

> Rule methods can return facts and post facts. Backward reasoning engine needs to know types of rule outputs. From rule class introspection the rule system knows about rule method return type, but it doesn't know about types posted through `post()`. Also, return type might not be enough, as returned instances may implement interfaces which other rules are interested in, but which are not declared in the rule return type. This method allows rules to inform the inference system about posted types.
>
> This method shall be invoked before rule is started. Place invocations of this method in rule constructors.
>
> Only rules used in backward reasoning need/must call this method to ensure proper reasoning. For methods without fact types information set through this method, the rule system uses method return type.

Ruleset definition XML file format is identical to the `<rules>` element in forward chaining rule set definition. The only difference is the `type` attribute of the root element. See `familyties-backwardreasoning.xml` ruleset definition file in Hammurapi Rules Tutorial for more details.

If Backward reasoning rules container is configured to use a thread pool (worker), then it performs inference in multiple threads. It allows to produce conclusions time efficiently in situations where facts source are slow. E.g. facts source may be a database query, Web service, ESB service, or remote file on a network. Rules session may use multiple such

sources. Reading source facts from them sequentially will result in much slower reasoning as opposed to doing it in parallel.

Backward reasoning rules container implements `FactSource` interface and returns `Iterator`. The container performs inference immediately as new facts are read from input fact sources. If reasoning is performed faster than new facts are read from sources, the output iterator blocks waiting for additional facts. It unblocks when new conclusion is available or when all input sources are exhausted. In the latter case it returns null if there is no new conclusion. Client code shall check value returned by it.next() for nullability and discard null values.

If a rule in a ruleset produces multiple conclusion types and current query needs only one of them (e.g. `Parent` rule infers `Mother` and `Father`), other types are accumulated in the rule session.

Multiple queries for different conclusion types can be run against the same rule session. Rules are not executed repeatedly for every query. For every subsequent query only rules in the inference chain which haven't been executed yet are invoked. For example, if after inferring `GrandMother` we decided to run a query for `GrandFather`, then the `Parent` rule wouldn't be executed during the second query, because it would have inferred `Father` conclusions during the first run. Needless to say, if the same query is run twice, no inference takes place during the second run – client code receives accumulated results immediately.

The reasoning engine features infinite reasoning loop detection and means to avoid processing of duplicate facts/conclusions.

Interactive Rules Parameterization

One interesting aspect of using a backward chaining rules engine is implementation of complex wizards, e.g. mortgage or insurance policy application. In such a wizard a user iteraction call-back interface is provided to rules so they can ask user for information they need. In this case user is asked only information which is really needed to come to required conclusion.

In the case of forward-chaining decisioning, user might be asked to provide information which is not used in decisioning process. On the other hand, some required information might be omitted and decisioning will either fail or will yield incorrect results.

Applicability

Forward chaining is applicable when most of output conclusions are going to be used by the application. For example, if a rule session is used for validation of web page inputs, then all validation errors inferred by the session are used by the application.

Backward chaining is applicable when

- There are many fact sources,

- There are many facts in the sources, e.g. millions of records in a database.

- Retrieval of such facts is costly computationally (long-running database queries or mainframe transactions) or monetarily (e.g. in SOA there might be a charge associated with invocation of a particular service).

- There are many conclusion types, but only few of them used for each particular query.

Data flow package

Classes in the data flow package (`biz.hammurapi.dataflow`) are intended for building multi-threaded process execution engines, e.g. workflow processors.

These classes will be used in the future products of Hammurapi Group.

They are placed in Hammurapi Rules because of their conceptual affinity to dispatchin and inference classes of Hammurapi Rules.

Appendix 1 Engagement model

Error! Not a valid filename.

Appendix 2 GNU Free Documentation License

Version 1.2, November 2002

```
Copyright (C) 2000,2001,2002  Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution

and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.

- **D.** Preserve all the copyright notices of the Document.

- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- **H.** Include an unaltered copy of this License.

- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

References

1. Hammurapi Rules home page - http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/products/hammurapi-rules/index.html

2. Hammurapi Rules JavaDoc - http://www.hammurapi.biz/hammurapi-biz/doc/products/hammurapi-rules/api/index.html

3. JSR-94 Specification - http://www.hammurapi.biz/hammurapi-biz/system/FileActions/get/81/jsr94_spec.pdf

4. JSR-94 API - http://www.hammurapi.biz/hammurapi-biz/doc/products/hammurapi-rules/jsr94/api/index.html

5. JSR-94 Home page - http://www.jcp.org/en/jsr/detail?id=94

6. Getting started with Java Rule Engine API - http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html

7. The Java Business Rules Community - http://www.javarules.org/

8. Open source rule engines written in Java - http://www.manageability.org/blog/stuff/rule_engines/view