ТЕХНОЛОГИЧЕСКОЕ ОБРАЗОВАНИЕ

*Сафонов Владимир Олегович,*
*Новиков Антон Владимирович,*
*Сигалин Максим Владимирович,*
*Смоляков Алексей Леонидович,*
*Черепанов Дмитрий Геннадьевич*

# ИНТЕГРАЦИЯ МЕТОДОВ ИНЖЕНЕРИИ ЗНАНИЙ И ИНЖЕНЕРИИ ПРОГРАММ: СИСТЕМА УПРАВЛЕНИЯ ЗНАНИЯМИ KNOWLEDGE.NET

## ВВЕДЕНИЕ

*Инженерия знаний* (knowledge engineering) – одно из важных направлений современного программирования, занимающееся развитием языков, методов и систем представления и обработки знаний на компьютерах. Оно возникло в 1950-х гг. как один из разделов *искусственного интеллекта*, в связи с тем, что при решении задач творческого характера с помощью компьютеров оказалось необходимым не только разрабатывать алгоритмы и программы, реализующие те или иные методы (например, стратегию поиска решения сначала в глубину), но и явным образом (отдельно от программ) хранить, изменять и пополнять *базы знаний*, содержащие в обобщенном виде необходимую для решения задач информацию о предметной области.

Как выяснилось, знания, необходимые для решения многих нетривиальных практических задач с использованием компьютеров, носят *гибридный* характер, то есть требуются не только *процедурные знания* (алгоритмы, их программные реализации и типовые процедуры решения задач – то есть те формы знаний, которые фактически всегда неявно используются для компьютерного решения любой задачи, но в виде, жестко «зашитом» в исходный текст программы), но также *концептуальные знания* (определения концепций проблемной области и отношений между ними), *фактуальные знания* (конкретные факты и их связи между собой) и *эвристические знания* (неформальные правила рассуждений, отражающие практический опыт решения задач в проблемной области).

Соответственно, основными методами представления знаний на компьютерах, сложившимися в течение нескольких десятилетий, являются:

– *продукции,* или *правила (rules)* – конструкции вида *ЕСЛИ условие ТО действие*, используемые для *продукционного вывода*, при котором на каждом шаге при истинности условия некоторого выбранного правила активизируется его действие, которое может заключаться в добавлении в *рабочее множество* гипотез некоторого утверждения. Правила удобны для представления эвристических знаний и структурируются в виде *наборов правил (rule sets)*, каждый из которых применяется для вывода некоторого *целевого утверждения (goal)*;

– *фреймы (frames)* – иерархические структуры, удобные для представления зна-

ний о концепциях и их взаимосвязи; фрейм состоит из *имени* и *слотов (slots)*, каждый из которых имеет свое *имя* и *значение* (как правило, ссылку на другой фрейм); в соответствии с оригинальным подходом автора данной концепции M.Minsky (США), фреймы могут использоваться не только для представления структур знаний, но и для *сопоставления* с объектами исследуемой проблемной области с целью их анализа и классификации;

– *семантические сети (semantic nets)* – нагруженные ориентированные мультиграфы, которые, как и фреймы, удобны для представления концепций и их взаимоотношений и являются, по-видимому, наиболее общей формой представления знаний.

Методы представления знаний получили особенно широкое распространение, начиная с 1970-х гг., когда весьма популярным направлением искусственного интеллекта стали *экспертные системы (expert systems)* [1] – интеллектуальные программы, основанные на использовании отдельно хранимой пополняемой *базы знаний* о предметной области и играющие в данной области роль *эксперта*, умеющего решать в ней, как правило, достаточно узкий круг задач диагностики, планирования, прогнозирования и т. д.

В настоящее время развитие методов и инструментов представления знаний приобрело особую актуальность, в связи с необходимостью улучшения качества информационного поиска в Интернете.

## 1. ПРОБЛЕМА ИНТЕГРАЦИИ МЕТОДОВ ИНЖЕНЕРИИ ЗНАНИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

По мере развития методов инженерии знаний их теоретические, методологические и технологические основы все более отдалялись от основ *инженерии программ (software engineering)* – методов спецификации, проектирования, реализации, тестирования и сопровождения программ и программных продуктов.

Исторически сложилось так, что инженерия знаний, начиная с 1960-х гг. (создание языка ЛИСП для символьной обработки) и с 1970-х гг. (создание языка Пролог, основанного на идее использования метода резолюций для эффективного логического вывода в исчислении предикатов первого порядка, в качестве основы для систем искусственного интеллекта), развивалась на основе совершенно иных вычислительных моделей и парадигм, нежели традиционная программная инженерия (ФОРТРАН, АЛГОЛ, Паскаль, структурное и модульное программирование, абстрактные типы данных, объектно-ориентированное программирование и т. д.).

Для инженерии знаний использовались методы математической логики, теории продукций, *системы преобразований термов (term-rewriting systems)*, а опыт и результаты в данной области накапливались, главным образом, усилиями специалистов в указанных теоретических областях, программирующих на языках ЛИСП, Пролог и их диалектах, применение которых требует иных методов мышления, принципиально отличных от тех, которые используются при программировании на процедурных или объектно-ориентированных языках.

Напротив, в инженерии программ использовались и развивались языки, методы и инструменты, основанные на более традиционных вычислительных моделях, с точки зрения которых использование языков ЛИСП или Пролог выглядело «экзотикой».

Таким образом, за десятилетия развития обеих дисциплин сложился серьезный разрыв между ними.

Специалисты по инженерии знаний мыслят в терминах, подобных следующим: *«концепт», «факт», «гипотеза», «атом», «список», «условие», «заключение», «целевое утверждение», «граф», «логическая связка», «фрейм», «демон», «противоречие», «возврат (back-tracking)», «нечеткость», «коэффициент уверенности», «степень истинности».*

Характерный перечень терминов, используемый специалистом по инженерии программ, следующий: *«программа», «описание», «оператор», «переменная», «ячей-*

*ка памяти», «значение», «присваивание», «массив», «структура», «процедура», «параметр», «программный модуль», «класс», «объект», «метод», «исключительная ситуация», «стек», «куча».*

Налицо, к сожалению, практически полное взаимное непонимание между специалистами обеих категорий; пересечение множеств самих таких специалистов, увы, практически пусто.

Этот феномен нашел отражение и в языках и инструментах программирования, используемых в обеих областях. Не встретишь, например, специалиста по объектно-ориентированному программированию, использующего современные платформы Java и .NET, который в то же время был бы фанатиком логического программирования и разрабатывал бы многие свои программы в системах Turbo Prolog или Visual Prolog.

Однако на практике для решения очень многих реальных задач должно использоваться именно *сочетание методов программной инженерии и методов инженерии знаний*. Например, для решения задачи планирования развития сложных технических систем, которая была поставлена перед нами одним из заказчиков уже почти 20 лет назад, характерно сочетание *алгоритмических методов* дискретного программирования (для решения крупных частей задачи) с *методами искуственного интеллекта* (например, основанными на наборах эвристических правил), которые необходимо использовать для оценки результатов применения этих алгоритмических методов решения.

Увы, ни одна из известных, широко используемых платформ «традиционного» программирования (Java, .NET, Eiffel, Active Oberon и др.) не содержит адекватных средств представления знаний, которые были бы включены в эту систему в качестве основных понятий и конструкций базовых языков.

И наоборот, ни одна из известных систем инженерии знаний (Common LISP, Turbo Prolog, Visual Prolog, KEE, Ontolingua, Protégé и др.) не содержит адекватных, современных, удобных и эффективно реализованных средств «традиционного» программирования. Семантика конструкций подобных систем, как правило, формулируется в терминах их реализации на одном из диалектов ЛИСПа. Ни один специалист не станет, например, на ЛИСПе или Прологе реализовывать алгоритмы численного решения задач. Возможности же использования в системах инженерии знаний программных модулей на традиционных языках либо отсутствуют, либо ограничены, например, возможностью запустить из набора правил целое приложение для Windows (.exe) и использовать его результат, записанный в некоторый файл.

Описанный семантический разрыв между методами инженерии программ и инженерии знаний все более препятствует успешному решению многих реальных практических задач с помощью компьютеров и, по нашему мнению, должен быть преодолен в будущих системах.

В связи с этим в Санкт-Петербургском университете в лаборатории проф. В.О. Сафонова (до 2001 г. – лаборатории технологии программирования и экспертных систем, с 2001 г. – лаборатории Java-технологии НИИ математики и механики), начиная с 1980–1990 гг., разрабатываются методы *интеграции инженерии знаний и инженерии программ*. Они уже реализованы и в языке представления знаний Турбо-Эксперт (1991 г.) [3] – расширении Турбо-Паскаля средствами представления гибридных знаний, и в экспериментальных версиях языков и систем Java Expert [4] (расширения Java средствами представления знаний), и в C# Expert [5] – расширения языка C# средствами представления фреймовых и продукционных знаний.

В основе всех этих языков и систем, как и в основе описанных в данной работе языка и системы Knowledge.NET, лежит принцип *расширения современных языков и систем программирования средствами*

*(концепциями, языковыми конструкциями, библиотеками и др.) инженерии знаний.*

На наш взгляд, только такое сочетание методов и конструкций «традиционного» программирования (то есть инженерии программ) с методами и конструкциями инженерии знаний в одной системе программирования (платформе для разработки программ) позволит, наконец, создавать с их помощью гибкие, удобные, эффективные, понятные пользователю и адекватные для сопровождения и развития *интеллектуальные решения (intelligent solutions*) для реальных практических задач.

Как это выглядит на практике с точки зрения пользователя такой системы программирования?

Инженер знаний и программ, проектировщик и разработчик такого интеллектуального решения, использует весь арсенал методов представления гибридных знаний (онтологии, фреймы, наборы правил и др.) для формализованного описания предметной области. Для реализации же более традиционных «алгоритмических» компонент он использует конструкции современного, эффективно реализуемого, базового языка программирования (Java, C# и др.), расширением которого является инструментальный язык данной гибридной системы.

Для реализации расширений, предназначенных для представления знаний, используется *конвертирование* – перевод этих расширений в программу на *базовом языке* (Java, C# и др.), содержащую вызовы методов специализированной библиотеки (API), также являющейся частью данной системы и обеспечивающей поддержку реализации наиболее сложных по семантике конструкций для представления и обработки знаний.

Для компиляции в исполняемый объектный код полученных в результате конвертирования исходных текстов на базовом языке используется его «штатный» компилятор; например, для Java – компилятор *javac*, входящий в состав Java Developer's Kit (JDK) фирмы Sun Microsystems; для C# – компилятор фирмы Microsoft, входящий в состав Microsoft.NET Framework.

Отметим, что идеи и принципы интеграции методов инженерии знаний и инженерии программ получили в последние годы распространение и среди западных специалистов – например, в работе [2] и в докладах на крупных международных конференциях их развивает профессор Jeff Zhuk (University of Phoenix, Arizona, USA), с которым наш коллектив поддерживает творческое содружество.

## 2. АРХИТЕКТУРА СИСТЕМЫ KNOWLEDGE.NET

Как уже отмечалось в одной из наших предыдущих работ, опубликованных в журнале [6], одной из наиболее перспективных платформ для разработки программного обеспечения в настоящее время является платформа Microsoft.NET. Естественно, в нашем коллективе возникла идея разработки современной системы представления знаний на базе данной платформы – *системы Knowledge.NET*. При ее разработке использован опыт другого нашего проекта для Microsoft.NET – *системы аспектно-ориентированного программирования Aspect.NET* [7, 8].

Система Knowledge.NET [9] базируется на одноименном языке, который является расширением языка C# – наиболее современного и развитого языка программирования в настоящее время – концепциями и конструкциями для инженерии знаний.

Система предназначена для разработки и использования библиотек (баз) знаний из разнообразных предметных областей, которые могут проектироваться и реализовываться непосредственно на языке Knowledge.NET (средствами специализированного редактора и визуализатора знаний), а также могут автоматически извлекаться из WWW. В системе предусмотрено конвертирование полученных, сформулированных на языке Knowledge.NET знаний в общеупотребительный формат представления знаний – *KIF (Knowledge Interchange Format)* [10].

В соответствии с современными идеями и тенденциями, на основе успешного опыта разработки системы Aspect.NET система Knowledge.NET реализована как *расширение (add-in)* одной из наиболее со-

временных интегрированных сред современного программирования – *Microsoft Visual Studio.NET 2005 (Whidbey)*. Этот важнейший принцип архитектуры Knowledge.NET обеспечивает возможность применения при разработке баз знаний в системе Knowledge.NET всего широкого спектра возможностей проектирования, разработки и отладки программ, предоставляемых Microsoft Visual Studio.NET.

Система Knowledge.NET состоит из следующих основных программных компонент:

– *конвертора с языка Knowledge.NET в базовый язык C#* платформы Microsoft.NET (последующая компиляция программ на C#, полученных в результате конвертирования, обеспечивается штатными средствами интегрированной среды Visual Studio.NET);

– *редактора и визуализатора знаний Knowledge Editor*, обеспечивающего визуализацию, проектирование, реализацию и модификацию в интерактивном режиме исходных текстов баз знаний на языке Knowledge.NET;

– *системы Knowledge Prospector извлечения знаний в формате Knowledge.NET из текстов на естественном языке* (ориентированную на извлечение знаний из Интернета);

– *конвертора KIF Converter знаний из формата Knowledge.NET в общеупотребительный формат KIF* [10].

## 3. ЯЗЫК ПРЕДСТАВЛЕНИЯ ЗНАНИЙ KNOWLEDGE.NET

Язык Knowledge.NET является расширением языка C# средствами представления гибридных знаний, основанными на концепции *онтологии*, предложенной в начале 1990-х гг. в работах Стэнфордского университета (США). Онтология – это разновидность спецификации предметной области, выраженной в терминах ее *концептов* (понятий) и их взаимосвязей В качестве основы языка представления знаний Knowledge.NET используется фреймовые языки C# Expert [5] и Turbo Expert [3]. Однако, в отличие от обычных фреймовых языков, Knowledge.NET расширен языковыми конструкциями для представления онтологических знаний. Семантика онтологии Knowledge.NET схожа с концепцией языка представления знаний OWL, разработанного W3C консорциумом [11]. Также следует отметить, что в Knowledge.NET используется терминология, принятая в онтологических системах (Concept-Property-Individual), вместо терминологии фреймовых систем (Frame-Slot-Instance).

### 3.1. КОНЦЕПТЫ

Для представления множества экземпляров одного типа в онтологической модели используется понятие концепта (Concept). Аналогично, в объектно-ориентированной модели объекты описываются с помощью классов.

В качестве примера рассмотрим онтологию *Транспортные Средства (Vehicles)*. В рамках данной онтологии можно предположить, что все автомобили являются экземплярами некоторого концепта *Автомобиль (Automobile)*. При описании сложных онтологий концепты организованы в виде иерархической структуры (концепт-подконцепт), или *таксономии*. В вершине дерева концептов находится концепт *Thing*, от которого унаследованы все остальные концепты. Также введен концепт *Nothing*. Последний является подконцептом любого концепта (то есть унаследован от любого концепта).

### 3.2. ПОДКОНЦЕПТЫ

Концепт может быть конкретизацией (унаследован от) одного или более других концептов. Если концепт **A** унаследован от концептов **B** и **C**, в программе на Knowledge.NET это записывается с помощью одной из следующих конструкций:

1. **A** is_subconcept_of **B,C;**

Данная, наиболее короткая, форма записи может быть использована, если определяемая сущность (в данном случае концепт) задается с помощью только одного параметра (*is_subconcept_of*).

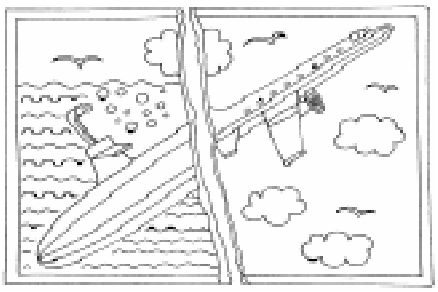
2. **A**
```
{
    is_subconcept_of B,C;
}
```

Данная форма записи используется, если сущность определяется несколькими параметрами.

3. **A**
```
{
   is_subconcept_of B;
   is_subconcept_of C;
}
```

Если параметр (в данном примере *is_subconcept_of*) в качестве значения получает несколько сущностей (в данном примере **B** и **C**), то этот параметр можно продублировать для каждой полученной сущности. Дублирование параметра особенно удобно, если одна из сущностей является анонимным концептом.

### 3.3. РАЗЪЕДИНЕННЫЕ КОНЦЕПТЫ



Иногда онтология подразумевает, что некоторый экземпляр не может быть одновременно реализацией некоторого множества концептов. Такие концепты называются *разъединенными*, или *дизъюнктными* (*disjoint*). Например, оба концепта *Самолет (Plane)* и *Подводная Лодка (Submarine)* являются подконцептами концепта *Транспортное Средство.* Тем не менее, конкретный экземпляр *Транспортного средства* (индивид) не может одновременно являться *Самолетом* и *Подводной Лодкой*. Разъединенные концепты определяются с помощью слова *disjoint*.

```
Vehicle is_subconcept_of Thing;
Plane is_subconcept_of Vehicle;
Submarine is_subconcept_of Vehicle;
disjoint Plane, Submarine;
```

Разъединенные концепты можно также определять с помощью слова *disjoint_with,* используемого внутри определения концепта. Семантика *disjoint_with* слегка отличается от семантики *disjoint* и может быть удобна в некоторых ситуациях. Например, предположим, что онтология *Транспортные Средства* содержит также концепт *Корабль (Ship)*. Концепты *Корабль* и *Самолет* являются разъединенными концептами, а концепты *Корабль* и *Подводная Лодка* будем считать не разъединенными концептами. В таком случае разъединенные концепты удобно определить следующим образом:

```
Vehicle is_subconcept_of Thing;
Submarine is_subconcept_of Vehicle;
Ship is_subconcept_of Vehicle;
Plane
{
is_subconcept_of Vehicle;
      disjoint_with Ship, Submarine;
}
```

### 3.4. НАБОР ЭКЗЕМПЛЯРОВ

Концепт может быть определен с помощью перечисления всех его экземпляров. Такие концепты называются *Наборами*. Если перечисляемый в наборе экземпляр не определен в системе, Knowledge.NET создает экземпляр с данным именем на основе концепта *Thing*.

```
enumerated concept CONCEPT_IDEN is_one _of
                INDIVIDUAL1, INDIVIDUAL2, ...;
```

где **CONCEPT_IDEN** – идентификатор концепта; **INDIVIDUAL1**, **INDIVIDUAL2** – идентификаторы экземпляров.

### 3.5. ОБЪЕДИНЕНИЕ, ПЕРЕСЕЧЕНИЕ И ДОПОЛНЕНИЕ КОНЦЕПТОВ

Концепт может быть определен как объединение (*union*), пересечение (*intersection*) или дополнение (*complement*) множеств экземпляров некоторого числа других концептов:

```
concept A is_intersection_of B, C, D...
concept A is_union_of B, C, D...
concept A is_complement_of B, C, D...
```

где **A**, **B**, **C**, **D** – концепты.

Концепт **A** – дополнение некоторого числа других концептов – является множеством всех экземпляров, которые не принадлежат перечисленным концептам (**B**, **C**, **D**...).

Рассмотрим следующий пример. Пусть определена некоторая онтология, состоящая из следующих концептов и экземпляров:

```
концепт A и его экземпляры a1, a2
концепт B и его экземпляры b1, b2, b3
концепт C и его экземпляр  c1
концепт D и его экземпляры d1, d2, d3, d4, d5
```

Тогда выражение «concept **E** is complement of **B**, **C**» определяет концепт, состоящий из экземпляров концептов **A**, **D** (**a1**, **a2**, **d1–d5**).

Особенно интересно использовать дополнение концептов для *анонимных (inline)* концептов. Например:

```
concept E
      {
            is_complement_of
            {
IsLivingAt has_value Russia
}
}
```

Здесь концепт **E** описывает множество всех людей, живущих не в России.

Конструкции *is_complement_of*, *is_union_of*, *is_intersection_of* можно вкладывать друг в друга.

### 3.6. СВОЙСТВА

Свойство представляет связь между двумя сущностями (во фреймовых системах свойства называются слотами). Поддерживаются следующие основные типы свойств:

• *Простые свойства.* Простое свойство связывает экземпляр со значением некоторого простого типа данных, например, со строкой (*string*) или целым числом (*int*).

• *Объектные свойства.* Объектное свойство связывает между собой два экземпляра.

• *Аннотации.* Аннотации также относятся к свойствам. С помощью свойства *Аннотация* пользователь может добавить описание (метаданные) к концептам, экземплярам, объектным и простым свойствам.

### 3.7. ДОМЕН И ОБЛАСТЬ ЗНАЧЕНИЯ

Для каждого свойства должен быть определен *домен (domain)* и *область значений (range)*. Например, в онтологии *Транспортные Средства* может быть определено свойство *ИмеетМаксимальнуюСкорость (HasMaxSpeed),* у которого *домен* содержит концепт *Транспортное Средство,* а *область значений* есть простой тип *uint*. На языке Knowledge.NET свойство *ИмеетМаксимальнуюСкорость* определяется следующим образом:

```
datatype property HasMaxSpeed
{
      domain Vehicle;
      range uint;
}
```

Отметим, что *домен* и *область значений* может содержать более одного идентификатора. В этом случае идентификаторы перечисляются через запятую.

### 3.8. ПРОСТЫЕ СВОЙСТВА

Простое свойство связывает экземпляр с не более чем одним значением некоторого простого типа данных. Например, свойство *ИмеетМаксимальнуюСкорость (HasMaxSpeed)* связывает экземпляры концепта *Транспортное Средство* со значением простого типа данных *uint.* Синтаксис определения простого свойства:

```
datatype property PROP_NAME
{
      [domain A, B, C, ...;]
      [range X, Y, Z;]
}
```

где

– **PROP_NAME** – название свойства. Хотя в Knowledge.NET не предусмотрено ограничений на наименования свойств, рекомендуется, чтобы имя начиналось со слова *Has (имеет...)* или *Is (является...)*. Например: *HasParent*, *IsParentOf*;

– **A**, **B**, **C** – идентификаторы концептов. Если *домен* не определен, то в качестве домена по умолчанию используется концепт *Thing*;

– **X**, **Y**, **Z** – идентификаторы простых типов. Если *область значений* не определена, то значение может быть любым простым типом.

### 3.9. ОБЪЕКТНЫЕ СВОЙСТВА

#### 3.9.1. Общее описание

*Объектное свойство* связывает между собой два экземпляра. Например, в рассматриваемой нами онтологии *Транспортные Средства* концепт *Транспортное Средство* мы можем связать с концептом *Цвет (Color)* с помощью свойства *ИмеетЦвет (HasColor)*:

```
#ontology "Vehicles"
#concepts
Color is_subconcept_of Thing;
Vehicle is_subconcept_of Thing;
Plane is_subconcept_of Vehicle;
Submarine is_subconcept_of Vehicle;
disjoint Plane, Submarine;

disjoint Color, Vehicle;

#properties

object property HasColor
{
      domain Vehicle;
      range Color;
}
#end_of_ontology "Vehicles"
```

#### 3.9.2. Подсвойства

В Knowledge.NET объектные свойства так же, как и концепты, образуют иерархическую структуру. Другими словами, допускается наследование свойств. Подсвойство имеет в качестве *домена* и *области значений* подмножества *домена* и *области значений* родителя. Определяется подсвойство с помощью ключевого слова *is_subproperty_of*. Например, если свойство **X** является подсвойством свойства **Y**, это должно быть записано следующим образом:

```
object property X is_subproperty_of Y
{
      domain A, B, C, ...;
      range K, L, M, ...;
}
```

Множественное наследование для свойств не допускается.

#### 3.9.3. Обратные свойства

Объектное свойство может иметь соответствующее обратное (инвертированное) свойство. Если некоторое свойство соединяет экземпляр **A** с экземпляром **B**, то обратное свойство соединяет экземпляр **B** с экземпляром **A**. Например, если машина Иванова А.К. *ИмеетЦвет (HasColor)* Красный, то Красный *ЯвляетсяЦветом (isColorOf)* машины Иванова А.К. Таким образом, в данном примере свойство *ЯвляетсяЦветом* – это обратное свойство по отношению к свойству *ИмеетЦвет*. Обратные свойства определяются с помощью ключевого слова *inverse*.

```
object property HasColor
{
      domain Vehicle;
      range Color;
      inverse IsColorOf;
}
```

#### 3.9.4. Функциональные свойства

Функциональные свойства связывают экземпляр не более чем с одним экземпляром из области значений. Например, свойство *ИмеетЦвет* из примера, описанного в разделе 3.9.3, является функциональным,

потому что экземпляр концепта *Транспортное Средство* не может быть покрашен сразу в два цвета. В то же время обратное свойство *ЯвляетсяЦветом* – не функциональное, так как несколько экземпляров могут иметь один и тот же цвет. Функциональные свойства определяются с помощью модификатора *functional*.

```
functional object property HasColor
{
domain Vehicle;
range Color;
inverse IsColorOf;
}
```

### 3.9.5. Обратные функциональные свойства

Так же, как и прямые свойства, обратные свойства могут быть функциональными. Для определения обратного функционального свойства используется модификатор *functional* перед ключевым словом *inverse*.

```
object property IsColorOf
{
domain Color;
range Vehicle;
functional inverse HasColor;
}
```

### 3.9.6. Транзитивные свойства

Свойства, удовлетворяющие следующему правилу, называются транзитивными: если экземпляр **A** связан с экземпляром **B** через свойство **P** и экземпляр **B** связан с экземпляром **C** через свойство **P**, то экземпляр **A** связан с экземпляром **C** через свойство **P**. Следует отметить две особенности транзитивных свойств:

1. Будем полагать, что свойство не может быть транзитивным и функциональным одновременно.

2. Если свойство транзитивно, то и обратное свойство транзитивно.

С теоретической точки зрения, свойство может быть транзитивным и функциональным одновременно только в случае, когда свойство соединяет экземпляр концепта сам с собой, что нетрудно доказать. Однако подобная ситуация представляется нам вырожденной и не поддерживается на уровне конструкций языка.

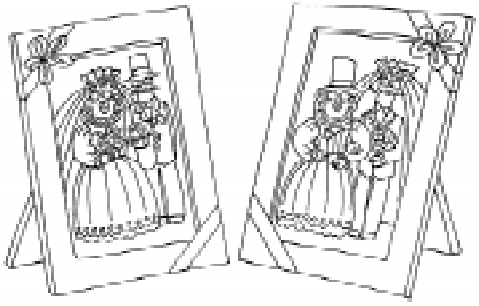
В качестве примера рассмотрим онтологию *Генеалогического Древа.* В данной онтологии определен концепт *Человек (Human)* и свойство *ЯвляетсяПредком (IsAncestorOf),* у которого *домен* совпадает с *областью значений* и состоит из единственного концепта *Человек.* Очевидно, что свойство *ЯвляетсяПредком* транзитивно. Действительно, предположим что Алексей *ЯвляетсяПредком* Сергея, а Сергей *ЯвляетсяПредком* Ирины, тогда Алексей *ЯвляетсяПредком* Ирины. Такую онтологию можно записать с использованием модификатора *transitive*.

```
#ontology "family-tree"

#concepts
Human is_subconcept_of Thing;

#properties
transitive object property IsAncestorOf
{
      domain Human;
      range Human;
      inverse HasAncestor;
}
#end_of_ontology "family-tree"
```

#### 3.9.7. Симметричные свойства



Свойство **P** называется симметричным, если из условия, что свойство **P** связывает концепт **A** с концептом **B**, следует связь концепта **B** с концептом **A** также через свойство **P**. Для симметричных свойств *область значений* совпадает с *доменом*, поэтому при записи область значений для симметричных свойств будем опускать. Также отметим, что, по определению симметричного свойства, оно обратно самому себе. В качестве примера рассмотрим свойство *ЯвляетсяБратомИлиСестрой (HasSibling)*. Симметричность данного свойства очевидна из его семантики. Действительно, если Алексей *ЯвляетсяБратомИлиСестрой* Оксаны, тогда Оксана *ЯвляетсяБратомИлиСестрой* Алексея. В Knowledge.NET симметричные свойства определяются с помощью модификатора *symmetric*.

Свойство *ЯвляетсяБратомИлиСестрой* в онтологии *Генеалогическое Древо* может быть определено следующим образом

```
symmetric object property HasSibling
{
      domain Human;
}
```

### 3.10. АННОТАЦИИ

Важным типом свойств являются Аннотации. С помощью свойства *Аннотация* пользователь может добавить описание (метаданные) к концептам, экземплярам, объектным и простым свойствам, правилам. Аннотация может содержать следующие поля:

1. Информация о Версии (version_info) Описание версии сущности (концепта, экземпляра, свойства);
2. Название (label) Развернутое название сущности на естественном языке.
3. Комментарий (comment)
   Автор (created_by)
4. Дата Создания (creation_date)

Все поля являются строками в формате UNICODE. Ниже приведен синтаксис определения аннотации.

```
annotation
{
      [version_info "SOME_TEXT";]
      [label "SOME_TEXT";]
      [comment "SOME_TEXT";]
      [created_by "SOME_TEXT";]
      [creation_date "SOME_TEXT";]
}
```

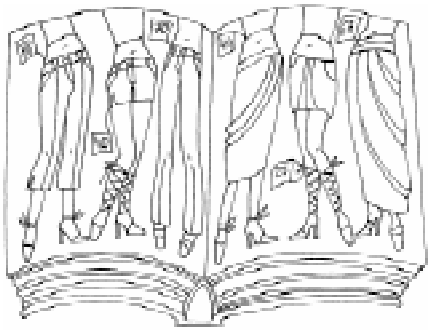
где **SOME_TEXT** – любой текст в формате UNICODE.

### 3.11. ОГРАНИЧЕНИЯ

В Knowledge.NET свойства могут быть использованы для задания *ограничений*. В онтологии каждому ограничению удовлетворяет ноль или более экземпляров. Таким образом, можно рассматривать ограничения как концепты. Поддерживаются три основных типа ограничений:

1. Ограничения по квантору.
2. Ограничения по мощности.
3. Ограничения по значению.

#### 3.11.1. Ограничения по квантору



Ограничения по квантору состоят из трех компонентов:

1. Свойство. Любое простое или объектное свойство, определенное в онтологии.
2. Квантор. Допускается квантор существования ($\exists$, *some_values_from*) и квантор всеобщности ($\forall$, *all_values_from*).

3. Наполнитель. Концепт или простой тип, который может быть *анонимным* концептом, то есть концептом, не имеющим имени и заданным явно в рассматриваемом фрагменте исходного кода.

В качестве примера рассмотрим онтологию *Генеалогического Древа,* введенную выше. Так как у каждого человека есть предки, мы можем ввести следующее ограничение в данной онтологии:

```
HasAncestor some_values_from Human
```

Данное ограничение определяет некоторое множество экземпляров, которые обладают следующими свойствами:

1. Унаследованы от концепта *Человек*.
2. Есть хотя бы один предок.

Другими словами, введенное нами отношение само по себе определяет некоторый новый концепт (множество экземпляров).

Ранее мы рассмотрели свойство *ЯвляетсяБратомИлиСестрой (HasSibling)*. Теперь с помощью ограничений определим концепт *БратИлиСестра (Sibling)*.

```
concept Sibling
{
   is_subconcept_of
   {
      Human;
      HasSibling some_values_from Sibling;
   }
}
```

### 3.11.2. Ограничения по мощности

В Knowledge.NET можно определить для свойства ограничение на количество связей с другими экземплярами или простыми типами.

Поддерживаются следующие ограничения по мощности:

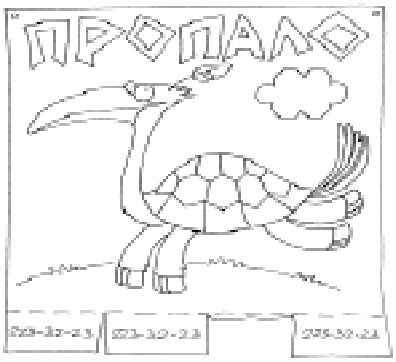
1. Не более чем (*max_cardinality*).
2. Не менее чем (*min_cardinality*).
3. Точное количество (*cardinality*).

В качестве примера введем отношение *ЯвляетсяРебенком (IsChildOf)* в онтологии *Генеалогическое Древо*. Поскольку *Человек* всегда имеет ровно двух родителей, определение концепта *Человек* имеет следующий вид:

```
#concepts
concept Human
{
  is_subconcept_of
  {
      Thing;
      HasSibling some_values_from Sibling;
      IsChildOf cardinality 2;
  }
}

#properties
object property IsChildOf
{
  domain Human;
  range Human;
  inverse HasChild;
}
```

### 3.11.3. Ограничения по значению




Ограничение по значению (*has_value*) позволяет определять концепты, основанные на существовании определенных экземпляров. Таким образом, экземпляр будет принадлежать данному концепту, только если одно из значений его свойств совпадает со значением, заданным в концепте.

Введем в онтологию из раздела 3.11.2 свойство *ЖивётВ* (*IsLivingAt*) и концепт (набор экземпляров) *Country*, тогда концепт *ЖительРоссии (ResidentOfRussia)* будет определяться следующим образом:

```
#concepts
enumerated concept Country is_one_of Russia,
USA, UK, Germany, France, Italy, China, Egypt;
concept ResidentOfRussia
{
      is_subconcept_of
      {
            Human;
            IsLivingAt cardinality 1;
            IsLivingAt has_value Russia;
      }
}
#properties
object property IsLivingAt
{
      domain Human;
      range Country;
}
```

### 3.12. ДОПОЛНИТЕЛЬНЫЕ ИМЕНА

Для концептов и экземпляров можно определить несколько имен (*aliases*). Например, чтобы иметь возможность использовать идентификатор *Машина (Car)* для концепта *Автомобиль,* необходимо ввести следующее определение:

```
concept Automobile
{
      alias "Car";
      is_subconcept_of Vehicle;
}
```

### 3.13. ЭКВИВАЛЕНТНЫЕ КОНЦЕПТЫ

Концепт **A** и концепт **B** называются эквивалентными, если это явно указано с помощью ключевого слова *equivalent_concepts* в конструкции следующего вида.

```
equivalent_concepts A, B, C, ...;
```

**A**, **B**, **C** – концепты.

### 3.14. ЭКЗЕМПЛЯРЫ

*Экземпляры* (или *Индивиды*) представляют объекты в определяемой онтологии. Важное отличие экземпляров от объектов в том, что один и тот же экземпляр может иметь несколько разных имен. Например, *ВАЗ 21010 номерной знак Y123AK* и *машина Иванова И.Л.* могут являться идентификаторами одного и того же экземпляра.

В программе экземпляр задается с помощью ключевого слова *individual*. Концепты, реализуемые данным экземпляром, определяются после ключевого слова *is_a*. Значения свойств задаются с помощью одной из ниже перечисленных конструкций:

1. Если присваивается только одно значение:

```
NAME_OF_PROPERTY  = VALUE_OF_PROPERTY;
```

2. Если присваивается два и более значений:

```
NAME_OF_PROPERTY  = {VALUE_OF_PROPERTY1,
                    VALUE_OF_PROPERTY2, ...}
```

Пример создания экземпляра концепта *Человек*:

```
individual Dmitry
{
   alias "Dima";
   is_a Human;
   HasSibling = Nina;
   HasAncestor = {Victor, Ludmila, Grigory,
                                    Alena};
   HasChild = {Egor, Maria};
   annotation
   {
      version_info "1.0";
      created_by "Alexander Fedorov";
      creation_date "01 July 2005";
   }
}
```

### 3.15. ОДИНАКОВЫЕ ЭКЗЕМПЛЯРЫ

Если экземпляр **A** совпадает с экземпляром **B**, то данный факт может быть определен в онтологии с помощью ключевого слова *same:*

```
same A, B, C, ...;
```

где **A**, **B**, **C** – экземпляры.

Необходимо отметить, что ключевые слова *alias* и *same* имеют различную семантику.

Слово *same* используется для объединения различных экземпляров в одну сущность. Например, инженер знаний создает новую онтологию, объединяя существующие онтологии. В исходных онтологиях, одни и те же экземпляры могут быть описаны по-разному, но в новой онтологии данные экземпляры должны трактоваться как одна сущность (экземпляр). Чтобы объединить эти экземпляры, инженер знаний использует слово *same*.

*Alias*, в свою очередь, – это способ ассоциировать один и тот же физический объект с разными именами.

Таким образом, можно сказать, что с помощью оператора *same* объединяются разные физические представления, а с помощью оператора *alias* одному и тому же физическому представлению присваиваются разные логические имена.

### 3.16. РАЗЛИЧНЫЕ ЭКЗЕМПЛЯРЫ

Ключевые слова *different_from*, *all_different* позволяют явно определить в онтологии, что данные экземпляры концептов различны.

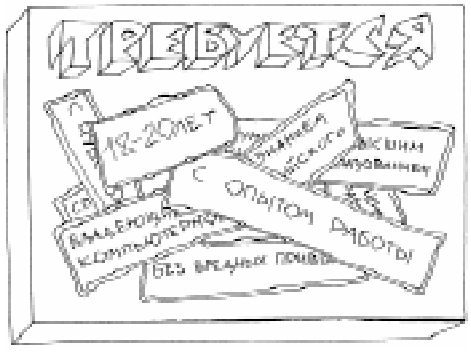
Например, если необходимо явно задать экземпляры *Билл Смит, Джефф Крамак* и *Стивен Фриман* концепта *Человек* как отличные друг от друга, это можно сделать с помощью следующей конструкции:

```
all_different Bill_Smith, Jeff_Kramak,
                          Stephen_Freeman;
```

Если необходимо определить, что Джефф_Крамак отличен от экземпляров Билл_Смит и Вильям_Смит, а о том, различны ли экземпляры Билл_Смит и Вильям_Смит, нам ничего не известно, тогда удобнее воспользоваться конструкцией:

```
individual Jeff_Kramak
{
is_a Human;
different_from Bill_Smith, William_Smith;
}
```

### 3.17. ЯЗЫК ЗАПРОСОВ



Язык запросов позволяет выбирать из онтологии экземпляры по некоторому заданному критерию (запросу).

Следует отметить, что сами по себе запросы не являются онтологическими знаниями. Язык запросов является инструментом, который позволяет осуществлять экспертную деятельность на заданной онтологии.

В качестве примера рассмотрим следующий запрос в онтологии *Транспортные Средства*: «Все транспортные средства, максимальная скорость которых больше чем 100 км/ч». Результатом выполнения данного запроса будет набор экземпляров концепта *Транспортное Средство*, у которых значение свойства *ИмеетМаксимальную-Скорость* превышает 100 км/ч.

Язык запросов поддерживается с помощью класса *QueryEngine,* определенного в библиотеке классов Knowledge.NET.

### 3.18. СИНТАКСИС ЗАПРОСА

Запрос позволяет инженеру знаний получить коллекцию экземпляров онтологий, которые удовлетворяют определенному критерию. Все запросы можно разделить на три основных типа:

1. Запросы, критерий которых основан на некоторой коллекции концептов и значениях их свойств;

2. Запросы, критерий которых основан только на значениях свойств, без указания концептов;

3. Запросы, критерий которых содержит только перечисление концептов, без указания свойств и их значений.

#### 3.18.1. Общее описание

С точки зрения синтаксиса, запрос представляет собой текстовую строку (*string*) в следующем формате:

```
individuals of concepts CONCEPT1, CONCEPT2,
                                  CONCEPT3...
where EXP1
```

Фрагмент «of concepts **CONCEPT1**, **CONCEPT2**, **CONCEPT3**...» задает множество концептов, из которых будут выбираться экземпляры. Данная часть запроса не является обязательной (например, она не определяется в запросах второго типа).

Некоторые из концептов могут быть *Набором Экземпляров,* который определен с помощью подзапроса. Например:

```
individuals of concepts CONCEPT1, CONCEPT2,
    individuals of  CONCEPT3, CONCEPT4...
    where EXP2
    where EXP1
```

Фрагмент «where **EXP1**» задает ограничения на свойства выбираемых концептов (критерий). Описание синтаксиса выражения **EXP1** приведено ниже. Данная часть запроса не является обязательной (например, она не определяется в запросах третьего типа). Тем не менее, одна из частей запроса «*of concepts*» или «*where*» должна быть обязательно задана в запросе.

Пример запроса без критерия:

```
individuals of Vehicle
```

Данный запрос в онтологии *Транспортные Средства* возвращает все экземпляры концепта *Транспортное Средство* и его подконцептов (то есть экземпляры концептов *Автомобиль, Корабль, Подводная лодка* и *Самолет*). Данный запрос относится к третьему типу.

### 3.18.2. Критерий запроса

Критерий запроса задает ограничения на свойства выбираемых концептов. Например, запрос в онтологии *Транспортные Средства*: «Все транспортные средства, максимальная скорость которых больше чем 100 км/ч» на языке Knowledge.NET записывается следующем образом:

```
individuals of concepts Vehicle where
HasMaxSpeed > 100
```

Формальное описание грамматики критерия запроса в форме Бэкуса-Наура представлено ниже:

```
EXP ::= OBJ_PROP  OBJ_OP  ID
OBJ_OP ::= contains | does_not_contain
EXP ::= DATA_PROP  DATA_OP  VALUE
EXP ::= EXP  OP  EXP
EXP ::= (EXP)
OP ::= or | and
```

Пояснения:

**EXP** – критерий запроса. Критерий запроса задается после ключевого слова «where». Выражение может состоять из нескольких подвыражений, соединенных логическими связками «И» и «ИЛИ». Логическая связка «И» задается с помощью ключевого слова *and*. Логическая связка «ИЛИ» определяется с помощью ключевого слова *or*. Для определения приоритета логических операций используются круглые скобки. Например:

```
(EXP1 or EXP2) and EXP3
```

**OBJ_PROP** – идентификатор объектного свойства. Если идентификатор совпадает с каким-либо ключевым словом запроса, идентификатор пишется в квадратных скобках. Например:

```
[individuals]
```

**ID** – идентификатор некоторого экземпляра. Если идентификатор совпадает с каким-либо ключевым словом запроса, идентификатор пишется в квадратных скобках.

**OBJ_OP** – двуместная связка между экземпляром и объектным свойством. Поддерживаются следующие типы связок: *contains* и *does_not_contain.*

Рассмотрим пример запроса с использованием объектного свойства. Запрос составлен на онтологии *Генеалогическое Дерево*, определенной выше:

```
individuals of Human where HasChild contains
                                        Nina
```

Поскольку домен свойства **HasChild** содержит в себе единственный концепт Human, на данной онтологии вышеприведенный запрос аналогичен следующему:

```
individuals where HasChild contains Nina
```

**DATA_PROP** – идентификатор простого свойства. Если идентификатор совпадает с каким-либо ключевым словом запроса, идентификатор пишется в квадратных скобках.

**VALUE** – значение свойства. Числовые значения указываются без кавычек. В качестве разделителя между целой и дробной частью используется точка (например: 3.1415926). Числа в шестнадцатиричной системе исчисления записываются в формате: 0×1234. Текстовые строки берутся в

одиночные кавычки (например: 'Hello World!').

**DATA_OP** – двуместная связка между простым свойством и некоторым значением.

Поддерживаются следующие типы связок:

• Логические свойства (bool):
'=' (равно)
'!=' (отлично от)

• Числовые свойства (int, float, long и другие):
'=' (равно)
'!=' (отлично от)
'<' (больше)
'>' (меньше)
'<=' (больше или равно)
'>=' (меньше или равно)

• Строковые свойства (string):
'=' (равно)
'!='(отлично от)
'<'(больше в лексикографическом порядке)
'>' (меньше в лексикографическом порядке)
'<=' (больше или равно в лексикографическом порядке)
'>=' (меньше или равно в лексикографическом порядке)
'contains' (содержит хотя бы одно вхождение текста)
'does_not_contain' (не содержит ни одного вхождения текста)
'begins_with' (начинается с текста)
'ends_with' (заканчивается текстом)

Рассмотрим пример запроса, содержащего простые и объектные свойства, а также логические связки между ними. Данный запрос составлен для онтологии *Транспортные Средства*:

```
individuals of Automobile
where (Color contains Red) or (HasMaxSpeed > 100
                    and HasMaxSpeed <= 250)
```

Данный запрос возвращает все экземпляры концепта *Автомобиль,* которые либо красного цвета, либо у которых максимальная скорость больше 100 км/ч и не превышает 250 км/ч.

Отметим, что в вышеприведенном примере конструкция «individuals of **Automobile**» имеет существенное значение. Результат запроса, приведенного ниже, может быть отличен от результата предыдущего запроса, поскольку выборка будет проводиться по всем экземплярам концепта **Vehicle**, а не только по экземплярам концепта **Automobile**:

```
individuals
where (Color contains Red) or (HasMaxSpeed > 100
                    and HasMaxSpeed <= 250)
```

### 3.19. ВЫПОЛНЕНИЕ ЗАПРОСА

Выполнение запросов происходит с помощью класса *QueryEngine,* определенного в библиотеке классов Knowledge.NET. Для выполнения запроса используется статический метод:

```
ICollection<Individual>
        QueryEngine.execute(string query)
```

Данный метод в качестве параметра получает запрос и возвращает коллекцию экземпляров, полученную в результате выполнения запроса. В случае, если запрос содержит ошибку, метод execute() генерирует исключение *QueryValidationException.*

### 3.20. НАБОРЫ ПРАВИЛ

Набор правил задает совокупность продукций, используемых для вывода. Основные компоненты набора правил:

1. Локальные переменные, объявленные в наборе правил в C# нотации.
2. Целевая переменная.
3. Правила, составляющие набор.

Форма представления набора правил близка к принятым в языках KEE и Турбо-Эксперт.

Контекстом набора правил является онтология, в которой объявлен данный набор.

### 3.21. ОПИСАНИЕ СТРУКТУРЫ НАБОРА ПРАВИЛ

Набор правил имеет следующую структуру:

```
ruleset IDEN
{
      C#_CODE
      goal V;
      [rule RULE_IDEN
```

```
    {
        if (B1)
        then S2;
        [else S3;]
        [priority INT_VALUE;]
    }]+
}
```

Пояснения:

**IDEN** – идентификатор набора правил;

**C#_CODE** – некоторый код на C#, содержащий определения локальных переменных и констант, используемых в правилах вывода. Одна из этих переменных может быть помечена как целевая (goal) переменная. Целевая переменная может быть переопределена при вызове метода consult();

**RULE_IDEN** – имя правила;

**B1** – условие правила; представляется в виде выражения, принимающего значения истина (true) или ложь (false);

**S2** – заключение (код C#, исполняемый в случае истинности условия);

**S3** – код C#, исполняемый в случае, если выражение **B1** ложно;

priority **INT_VALUE** – приоритет правила; данный параметр может использоваться машиной вывода для разрешения конфликтов (в случае, если на каком-либо шаге вывода несколько правил имеют истинное значение условия). Отметим, что машина вывода *ProductionSystem*, поставляемая вместе с библиотекой Knowledge.NET, не использует приоритеты.

### 3.22. МАШИНА ВЫВОДА

Библиотека классов Knowledge.NET предоставляет доступ к наборам правил через специальные интерфейсы, что позволяет разработчику реализовать свою собственную машину вывода.

Также библиотека содержит простую машину вывода, реализованную в виде класса *ProductionSystem*. Данный класс предоставляет следующие статические свойства и методы:

*public static bool showAnnotations* – данное свойство задает, будут ли показываться аннотации, определенные в правилах в ходе консультации, или нет;

*public static void consult (string ruleset);*

*public static void consult (string ruleset, string goal);*

*public static void consult (string ruleset, string goal, ChainingMethod method);*

Данные методы начинают консультацию, используя заданный набор правил.

*Параметры:*

*ruleset* – имя набора правил, с использованием которого будет осуществляться консультация;

*goal* – некоторая локальная переменная из контекста набора правил. Данный параметр не обязателен: если цель не определена, выводится цель, определенная в наборе правил;

*method* – стратегия логического вывода, может быть прямой или обратной (по умолчанию используется прямой вывод).

Особенности текущей реализации машины вывода:

- В настоящее время машина вывода поддерживает только прямую стратегию вывода.
- Метод разрешения конфликтов, используемый машиной, заключается в выборе правила, имя которого будет первым, исходя из лексикографического порядка.

### 3.23. ОБЩАЯ СТРУКТУРА ПРОГРАММЫ НА KNOWLEDGE.NET

Программа на Knowledge.NET состоит из двух частей:

1. Исходный код C#;
2. Исходный код, специфичный для Knowledge.NET (Концепты, Свойства, Экземпляры)

Исходный код Knowledge.NET отделяется от (обычного) исходного кода C# с помощью ключевого слова ***«#ontology»***.

Из исходного кода на C# имеется возможность использовать концепты и их свойства, применяя стандартный способ адресации: *имя_концепта.свойство*. Также вместе с Knowledge.NET поставляется библиотека, которая позволяет делать удобные запросы к онтологическим знаниям.

Пример программы на языке Knowledge.NET:

```
using System;
// C# native code
namespace HelloWorld
{
  class Hello
    {
      [STAThread]
      static void Main(string[] args)
      {
       Console.out.WriteLine ("Vehicle: " +
                              Lada.HasName);
      }
    }
}

  // Knowledge.NET specific code
#ontology "Vehicles"
#concepts
Color is_subconcept_of Thing;
Vehicle
{
    is_subconcept_of Thing;
    some_values_from HasName string;
   cardinality HasName 1;
}
Plane is_subconcept_of Vehicle;
Submarine is_subconcept_of Vehicle;
disjoint Plane, Submarine;
disjoint Color, Vehicle;
#properties
object property HasColor
{
   domain Vehicle;
   range Color;
}
functional datatype property HasName
{
   domain Vehicle;
   range string;
}
#individuals
individual Lada
{
    is_a Vehicle;
    HasName = "Lada";
}
#end_of_ontology "Vehicles"
```

### Литература


1. *Сафонов В.О.* Экспертные системы – интеллектуальные помощники специалистов. СПб.: «Знание», 1992.
2. *J. Zhuk.* Integration-ready architecture and design. Cambridge University Press, 2004.
3. *Сафонов В.О. и др.* Язык представления знаний Турбо-Эксперт // Кибернетика, 1991, № 5.
4. *Safonov V.O., Cherepanov D.V.* Java extension by production knowledge representation constructs and its implementation // Proceedings of International Conference «110th Anniversary of Radio Invention» and Regional IEEE Conference, St. Petersburg, 2005.
5. *Новиков А.В.* C# Expert – расширение языка C# средствами представления знаний // Дипломная работа. СПбГУ, 2004.
6. *Сафонов В.О.* Платформа Microsoft.NET: принципы, возможности, перспективы // Компьютерные инструменты в образовании, 2004, № 5.
7. *Safonov V.O., Grigoryev D.A.* Aspect.NET: aspect-oriented programming for Microsoft.NET in practice // .NET Developer's Journal, 2005, № 7.
8. Web-страницы проекта Aspect.NET: http://www.msdnaa.net/curriculum/?id=6219
9. Web-страницы проекта Knowledge.NET: http://www.knowledge-net.ru
10. Спецификация языка KIF: http://logic.stanford.edu/kif/dpans.html
11. OWL Web Ontology Overview. http://www.w3.org/TR/owl-features/



***Сафонов Владимир Олегович, доктор технических наук, профессор кафедры информатики СПбГУ, руководитель лаборатории Java-технологии.***

***Новиков Антон Владимирович, Сигалин Максим Владимирович, Смоляков Алексей Леонидович, Черепанов Дмитрий Геннадьевич – аспиранты кафедры математико-механического факультета СПбГУ.***