

**МОДЕЛИ ПРЕДСТАВЛЕНИЯ И ОБРАБОТКИ  
ДАННЫХ И ЗНАНИЙ.  
ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

Под редакцией В. В. Голенкова

В 3-х частях

Часть 2

**Т. Л. ЛЕМЕШЕВА, Н. А. ГУЛЯКИНА, А. И. ТОЛКАЧЕВ**

*Рекомендовано Учебно-методическим объединением  
вузов Республики Беларусь по образованию  
в области информатики и радиоэлектроники  
в качестве учебно-методического пособия  
для студентов учреждений, обеспечивающих получение  
высшего образования по специальности «Искусственный интеллект»*

Минск БГУИР 2008

УДК 004.65+004.82 (075.8)

ББК 32.973 – 018.2 я 73

М 74

**Р е ц е н з е н т ы:**

заведующий кафедрой информатики и вычислительной техники  
Учреждения образования «Высший государственный колледж связи»,  
канд. техн. наук, доц. Е. В. Новиков;

доцент кафедры педагогики и проблем развития образования  
Учреждения образования «Белорусский государственный университет»,  
канд. пед. наук М. Ф. Поснова

**Модели представления и обработки данных и знаний. Лабораторный**  
М 74 **практикум : учеб.-метод. пособие ; под ред. В. В. Голенкова : В 3 ч. Ч. 2 /**  
**Т. Л. Лемешева, Н. А. Гулякина, А. И. Толкачев. – Минск : БГУИР, 2008. –**  
**46 с. : ил.**

ISBN 978-985-488-197-3 (ч. 2)

Во второй части лабораторного практикума содержатся теоретические сведения и практические задания, касающиеся трех наиболее распространенных на сегодняшний день моделей представления и обработки знаний, используемых при автоматизации бизнес-процессов различных организаций: документо-ориентированная иерархическая модель, агентно-ориентированная модель, графодинамическая ассоциативная модель. В теоретической части приводятся описания моделей, языки и программные средства для их реализации, а также примеры их использования. В практической части даны контрольные вопросы и варианты индивидуальных компьютерных заданий, которые касаются различных аспектов реализации приведенных моделей.

**УДК 004.65+004.82 (075.8)**

**ББК 32.973 – 018.2 я 73**

Часть 1 издана в БГУИР в 2007 г. Модели представления и обработки данных и знаний. Лабораторный практикум : учеб.-метод. пособие; под ред. В. В. Голенкова : В 3 ч. Ч. 1 / Н. А. Гулякина, Т. Л. Лемешева, В. В. Агашков. – Минск : БГУИР, 2007. – 44 с. : ил.

**ISBN 978-985-488-197-3 (ч. 2.)**

**ISBN 978-985-488-178-2**

© Лемешева Т. Л., Гулякина Н. А.,  
Толкачев А. И., 2008

© УО «Белорусский государственный  
университет информатики  
и радиоэлектроники», 2008

## Содержание

|   |    |
|---|----|
| Предисловие.....  | 4  |
| Введение.....   | 6  |
| Лабораторная работа №1. Реализация систем обработки онтологий на базе инструментального средства Java Agent Development Framework ..... | 7  |
| Лабораторная работа №2. Моделирование документов в среде Lotus Notes и реализация функций их обработки.....                             | 24 |
| Лабораторная работа №3. Реализация операций обработки семантических сетей на базе языка Semantic Code Programming .....                 | 35 |
| Литература .....  | 45 |

Библиотека БГУИР

## Предисловие

Лабораторный практикум создан по материалам лекций и лабораторных работ, проводимых в рамках учебного курса «Модели представления знаний, базы данных и интеллектуализация информационно-поисковых систем» для студентов БГУИР, с необходимыми поправками и дополнениями. В данном практикуме приводятся теоретические сведения и варианты индивидуальных заданий для трех лабораторных работ по курсу, а также затрагивается тема представления и обработки знаний в виде семантических сетей и реализации систем обработки знаний с использованием специализированных программных средств. Приведенный материал может быть использован для проведения лабораторных работ и тренингов по соответствующему учебному курсу, а также при выполнении курсовых и дипломных проектов и работ в рамках учебного плана специальности «Искусственный интеллект» и смежных специальностей.

Лабораторный практикум содержит теоретический материал и задачи для реализации на компьютере, касающиеся трех наиболее распространенных на сегодняшний день моделей представления и обработки знаний, используемых при автоматизации бизнес-процессов различных организаций:

- документо-ориентированная иерархическая модель представления корпоративных знаний, которая позволяет представить [9] множество знаний организации в виде взаимосвязанных структурированных текстов на естественном языке, называемых документами;
- агентно-ориентированная модель FIPA [2], позволяющая описать множество корпоративных знаний в виде набора взаимосвязанных классов и используемая в многоагентных системах различного назначения;
- графодинамическая ассоциативная модель представления и обработки знаний [6], базирующаяся на однородных семантических сетях и используемая для представления и обработки сложноструктурированных знаний различных предметных областей.

Задачами практикума являются изучение и приобретение практических навыков использования языков представления и обработки знаний на примерах таких инструментальных программных средств, как JADE [5], Lotus Notes [10; 11], SCP [7].

Выполнение практикума базируется на знаниях и навыках студентов, приобретенных ими при изучении курсов «Введение в специальность», «Математические основы искусственного интеллекта», «Общая теория систем», «Аппаратное и программное обеспечение сетей», «Ассоциативная память и ассоциативные процессоры в интеллектуальных компьютерах», «Организация и функционирование традиционных и интеллектуальных компьютеров».

Объектами изучения лабораторного практикума являются:

- архитектуры современных инструментальных программных средств обработки знаний;
- модели представления знаний, ориентированные на сетевое и иерархическое представление информации;
- языковые и инструментальные средства управления/обработки знаний, а также обеспечение их корректности, целостности и актуальности.

Задача лабораторного практикума заключается в формировании у студентов:

- знаний об архитектурах современных инструментальных программных средств обработки знаний;
- знаний о моделях представления знаний, ориентированных на сетевое и иерархическое представление информации, а также средствах их переработки;
- знаний основных подходов формализации различных проблемных областей с использованием технологий семантических сетей;
- умений и навыков разработки и обслуживания баз знаний в современных инструментальных средах.

Данный лабораторный практикум является дополнением учебно-методического материала, приведенного в учебном пособии «Семантическая модель сложноструктурированных баз данных и баз знаний» / В. В. Голенков [и др.]. – Минск : БГУИР, 2004. – 263 с.

В лабораторном практикуме для улучшения восприятия учебного материала принята следующая структура: текст разбивается на нумеруемые темы; в рамках темы – на лабораторные работы; их наименования выделяются жирным шрифтом разного размера; каждая лабораторная работа содержит теоретические сведения по изучаемой теме, пример решения задачи, контрольные вопросы, формулировки задач и варианты индивидуальных заданий.

## Введение

Основными проблемами искусственного интеллекта являются представление и обработка знаний. Решение этих проблем состоит как в разработке эффективных моделей представления знаний, методов получения новых знаний, так и в создании программ, устройств, реализующих эти модели и методы. Всякая интеллектуальная деятельность опирается на знания. В эти знания включаются характеристики текущей ситуации, оценки возможности выполнения действий, законы и закономерности предметной области. База знаний является неотъемлемым компонентом любой интеллектуальной системы. Знания в базе знаний хранятся в явном формализованном виде, в отличие от знаний, которыми владеет человек. Для представления и обработки знаний необходимы специальные модели и формальные языки, а также программные средства, которые автоматизируют процессы представления и обработки знаний.

Современными тенденциями развития прикладных интеллектуальных систем различного назначения являются увеличение объемов обрабатываемых знаний, усложнение структуры баз знаний, увеличение количества решаемых задач. Поэтому ориентация на сетевые модели представления знаний и параллельную асинхронную их обработку является обоснованной.

Во второй части лабораторного практикума рассматриваются сетевые иерархические модели представления и обработки знаний, а также их реализация в современных инструментальных программных средствах. Наиболее эффективной и распространенной реализацией документо-ориентированной иерархической модели представления корпоративных знаний является продукт ©Lotus Domino/Notes компании Lotus Development IBM. Для изучения агентно-ориентированной модели представления и обработки корпоративных знаний выбран свободно распространяемый пакет JADE (Java Agent Development Framework). Для изучения графодинамической ассоциативной модели представления и обработки знаний различных предметных областей выбраны инструментальные средства на базе языка SC (Semantic Code), ориентированные на параллельную асинхронную переработку графовых структур специального вида.

# **Лабораторная работа №1**

## **Реализация систем обработки онтологий на базе инструментального средства Java Agent Development Framework**

**Цель работы:** приобрести навыки формализации знаний предметной области и построения онтологии, а также навыки использования агентно-ориентированных инструментальных средств создания многоагентных систем и обработки построенной онтологии.

### **Теоретические сведения**

Процесс формализации знаний предметной области заключается в структуризации информационных ресурсов, построении понятийной системы, системы синтаксических и семантических правил и аксиом. Онтология – это формальное явное описание терминов предметной области и отношений между ними [3], а также построение системы утверждений, описывающей свойства этих понятий и закономерности предметной области. Существует большое количество онтологий, используемых в сети Internet. Онтологии, встречаемые в Internet, варьируются от больших таксономий, категоризирующих веб-сайты (например Yahoo!), до категоризаций продаваемых товаров и их характеристик (например Amazon.com). Консорциумом WWW (W3C) предложен язык RDF (Resource Description Framework) [1] для кодирования знаний на веб-страницах, для того чтобы сделать их понятными для электронных агентов, которые осуществляют поиск информации. Управление перспективных исследований и разработок министерства обороны США (The Defense Advanced Research Projects Agency, DARPA) в сотрудничестве с W3C разрабатывает язык разметки для агентов DARPA (DARPA Agent Markup Language, DAML), расширяя RDF более выразительными конструкциями, предназначенными для облегчения взаимодействия агентов в сети [4]. Во многих дисциплинах сейчас разрабатываются стандартные онтологии, которые могут использоваться экспертами по предметным областям для совместного использования и аннотирования информации в своей области. Например, в области медицины созданы большие стандартные структурированные словари, такие как SNOMED (Price and Spackman 2000), и семантическая сеть «Системы Унифицированного Медицинского Языка» (The Unified Medical Language System) (Humphreys and Lindberg 1993). Также появляются обширные общецелевые онтологии.

Например, Программа ООН по развитию (The United Nations Development Program) и компания Dun & Bradstreet объединили усилия для разработки онтологии UNSPSC, которая предоставляет терминологию товаров и услуг.

Система Java Agent Development Framework (JADE) представляет собой микропрограммное средство, предназначенное для разработки многоагентных систем, обрабатывающих различные онтологии, и включает [5]:

- среду запуска агентов на выполнение;
- библиотеку классов для программирования агентов;
- пакет графических средств для администрирования и мониторинга работы агентов.

Каждый исполняемый модуль JADE называется контейнером, который может содержать несколько работающих агентов. Множество работающих контейнеров называется платформой. Главный контейнер должен всегда быть активным, и все другие контейнеры должны быть зарегистрированы у него. При запуске первого контейнера он автоматически становится главным. Агенты JADE имеют уникальные имена. Главный контейнер отличается от других контейнеров наличием двух специальных агентов, которые автоматически запускаются при запуске JADE. Первый агент – AMS (Agent Management System). Данный агент предоставляет услуги именования агентов и права в рамках данной платформы. Вторым агентом – DF (Directory Facilitator). Данный агент предоставляет услуги доски объявлений, при помощи которой одни агенты находят услуги, предоставляемые другими агентами, и используют их для достижения своих целей.

Рассмотрим пример «Электронный книжный магазин», иллюстрирующий шаги создания приложения с использованием JADE. В электронном книжном магазине существует две группы агентов. Одни агенты продают книги, другие – покупают книги от имени своих пользователей. Каждый агент-покупатель получает от своего пользователя название книги через командную строку и периодически запрашивает всех известных ему агентов-продавцов о наличии этой книги в магазине. Как только один из агентов-продавцов отвечает положительно, агент-покупатель заполняет бланк покупки. Если ответ прислали несколько агентов-продавцов, то агент-покупатель принимает наилучшее



предложение (минимальная стоимость книги). Купив книгу, агент-покупатель прекращает свою работу.

Каждый агент-продавец предоставляет минимальный GUI, при помощи которого пользователь может добавлять новые книги и соответствующие им цены в каталог книг для продажи. Агенты-продавцы постоянно ожидают запрос от агентов-покупателей. Как только запрос поступил, агент-продавец проверяет наличие запрошенной книги в каталоге. Если такая книга присутствует, то агент-продавец отправляет стоимость книги, иначе отправляет отказ. Если агент-покупатель прислал бланк покупки, то агент-продавец сохраняет его и удаляет запрошенную книгу из каталога.

Данный пример можно также увидеть в пакете `examples.bookTrading`, включенном в программный комплекс JADE.

### ***Создание и идентификация агента***

Агент представляет собой класс, расширяющий класс `jade.core.Agent` и снабженный методом `setup()`:

```
import jade.core.Agent;
public class BookBuyerAgent extends Agent {
    protected void setup() {
        System.out.println("Hello!Buyer-agent"+getAID().getName()+" is ready.");
    }
}
```

Метод `setup()` предназначен только для инициализации агента. Основная работа агента реализуется при помощи так называемого «поведения».

Каждый агент идентифицируется при помощи объекта класса `jade.core.AID`. Для того чтобы получить идентификатор агента, нужно вызвать метод `getAID()` класса `Agent`. Объект класса `AID` содержит уникальное имя и номера адресов. Имя JADE имеет следующий формат `<nickname>@<platform-name>`. Например, если имя агента `Peter`, размещенный на платформе с именем `P1`, то глобальное уникальное имя агента будет `Peter@P1`:

```
String nickname = "Peter";
AID id = new AID(nickname, AID.ISLOCALNAME);
```

Адреса, включенные в `AID`, являются адресами платформ, на которых проживает агент. Эти адреса используются для коммуникации с другими агентами, проживающими на других платформах. Константа `ISLOCALNAME` указывает на то, что первый параметр представляет собой локальное для данной платформы имя, а не глобальное уникальное имя агента.

Откомпилировать созданного агента можно следующим образом:

```
javac -classpath <JADE-classes> BookBuyerAgent.java
```

Для того чтобы запустить агента, необходимо передать имя агента контейнеру JADE. Результат выполнения команды:

```
C:\jade>java -classpath <JADE-classes> jade.Boot Peter:BookBuyerAgent
This is JADE 3.3 - 2005/03/02 16:11:05
downloaded in Open Source, under LGPL restrictions,
at http://jade.cselt.it/
```

.....

```
Agent container Main-Container@JADE-IMTP://608ws02 is ready.
```

```
-----
```

```
Hello! Buyer-agent Peter@608ws02:1099/JADE is ready.
```

В консоли появится сообщение о версии среды, инициализированных сервисах, а также имя контейнера JADE, готового для работы, и сообщение агента, который запустился после запуска среды. Имя агента «Peter», имя платформы — «608ws02:1099/JADE».

Агент после запуска продолжает работать до тех пор, пока не будет вызван метод `doDelete()`. Аналогично методу `setup()`, который включает инициализацию агента, существует метод `takeDown()`, который вызывается сразу перед завершением работы агента и выполняет сборку мусора, возникшего в процессе работы агента.

Существует возможность передать агенту аргументы при помощи командной строки. Доступ к аргументам осуществляется при помощи метода `getArgument()`, размещенного в классе `Agent`. Данный метод возвращает аргументы, представленные в виде массива объектов `Object`. Например, агент `BookBuyerAgent` должен получить название книги, которую необходимо приобрести, как аргумент командной строки:

```
import jade.core.Agent;
import jade.core.AID;

public class BookBuyerAgent extends Agent {
    // Название книги, которую необходимо купить
    private String targetBookTitle;
    // Список известных агентов-продавцов
    private AID[] sellerAgents = { new AID("seller1", AID.ISLOCALNAME),
                                   new AID("seller2", AID.ISLOCALNAME) };

    // Инициализация агента
    protected void setup() {
        // Вывод приветственного сообщения
        System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
        // Получение названия книги как аргумента командной строки
        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            targetBookTitle = (String) args[0];
            System.out.println("Trying to buy "+targetBookTitle);
        }
    }
}
```

```

    }
    else {
        // Немедленное прекращение работы агента
        System.out.println("No book title specified");
        doDelete();
    }
}
// Здесь размещают операции очистки памяти
protected void takeDown() {
    System.out.println("Buyer-agent "+getAID().getName()+" terminating.");
}
}

```

В данном примере количество агентов-продавцов фиксировано. Однако существует возможность поиска и динамического добавления агентов-продавцов в список уже известных продавцов. В следующем примере передается название книги «Властелин колец»:

```
C:\jade>java jade.Boot buyer:BookBuyerAgent(The_Lord_of_the_rings)
```

### ***Поведение агента***

Поведение агента – это последовательность действий при решении агентом задачи. Для реализации поведения агента используется класс `jade.core.behaviours.Behaviour`. При помощи метода `addBehaviour()` класса `Agent` разработанному агенту можно добавить объект поведенческого класса. В каждый поведенческий класс необходимо включать методы `action()` и метод `done()`. Первый – позволяет описать действия, выполняемые агентом. Вторым – возвращает признак завершенности действий, составляющих поведение агента. Если агент все успешно выполнил, то объект класса `Behaviour` удаляется из списка всех подобных объектов. В общем случае у агента может существовать несколько линий поведения, которым будут соответствовать разные поведенческие объекты.

Агент может демонстрировать разные линии поведения параллельно. При этом различным линиям не устанавливаются различные приоритеты, как для `java`-потоков, они выполняются кооперативно. Метод `action()` всегда выполняется от начала до конца. Выбор агентом следующей линии поведения определяется программистом. Обобщенный алгоритм работы агента показан на рис. 1.1.

Метод `action()` никогда не завершит свою работу самостоятельно. Если в текущий момент список поведенческих объектов пуст, то агент выгружается из памяти («засыпает») и не занимает ресурсы. Агент «пробуждается», если в списке появляется новый объект.

Различают три вида поведения агентов:

1. Действие выполняется один раз, и агент немедленно прекращает работу после выполнения метода `action()`. Для реализации этого вида поведения используется класс `jade.core.behaviours.OneShotBehaviour`, в котором уже присутствует метод `done()`.

```

public class MyOneShotBehaviour extends OneShotBehaviour {
    public void action() {
        // выполнение действий X
    }
}

```

Действия X будут выполнены только один раз.

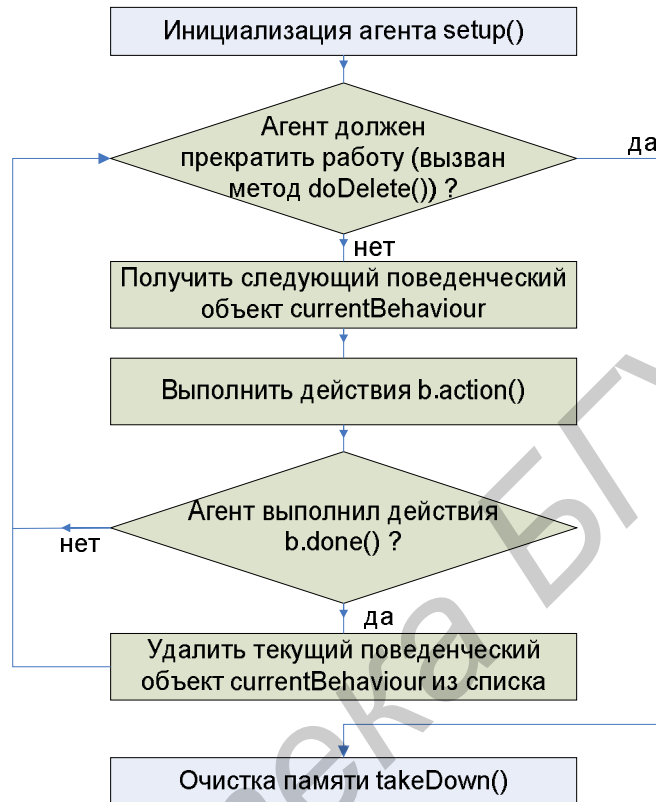


Рис. 1.1. Обобщенный алгоритм работы агента

2. Циклическое поведение. Метод `action()` будет выполнять одни и те же действия столько раз, сколько он будет вызван. Для реализации этого вида поведения используется класс `jade.core.behaviours.CyclicBehaviour`.

```

public class MyCyclicBehaviour extends CyclicBehaviour {
    public void action() {
        // выполнение действий Y
    }
}

```

Действия Y будут выполняться бесконечно. Агент может прекратить выполнение действий Y при помощи внешнего поведения.

3. Общее поведение. Выполнение тех или иных действий зависит от некоторого статуса и прекращается, если достигнуто заданное условие.

```

public class MyThreeStepBehaviour extends Behaviour {
    private int step = 0;
    public void action() {
        switch (step) {
            case 0:
                // выполнение действий X
                step++;
                break;
            case 1:

```

```

        // выполнение действий Y
        step++;
        break;
    case 2:
        // выполнение действий Z
        step++;
        break;
    }
}
public boolean done() {
    return step == 3;
}
}

```

Группы действий X, Y и Z будут выполнены последовательно, после чего работа агента будет прекращена.

Перечисленные поведения агентов не зависят от событий, происходящих в системе и которые могут повлиять на время ожидания агента. В JADE предусмотрено два класса `WakerBehaviour` и `TickerBehaviour`, которые позволяют определить время выполнения агентом действия.

1. Класс `WakerBehaviour` содержит абстрактный метод `handleElapsedTimeout()`, который начинает выполняться через указанный в конструкторе промежуток времени.

```

public class MyAgent extends Agent {
    protected void setup() {
        System.out.println("Adding waker behaviour");
        addBehaviour(new WakerBehaviour(this, 10000) {
            protected void handleElapsedTimeout() {
                // выполнение действий X
            }
        });
    }
}

```

Действия X будут выполнены через 10 с, после того как новая линия поведения агента будет добавлена. После завершения работы метода `handleElapsedTimeout()` работа агента прекращается.

2. Класс `TickerBehaviour` содержит абстрактный метод `onTick()`, который повторяет описанные в нем действия через заданный в конструкторе промежуток времени.

```

public class MyAgent extends Agent {
    protected void setup() {
        addBehaviour(new TickerBehaviour(this, 10000) {
            protected void onTick() {
                // выполнение действий Y
            }
        });
    }
}

```

Действия Y будут выполняться периодически каждые 10 с. Работа метода `onTick()` никогда не прекратится.

Рассмотрим поведение агента-продавца и агента-покупателя на примере электронного книжного магазина. Агент-покупатель периодически запрашивает агента-продавца о наличии книги, которую необходимо купить. Указанное поведение агента-покупателя можно реализовать при помощи класса

TickerBehaviour, в котором будет генерироваться запрос продавцу. Необходимо модифицировать класс BookBuyerAgent следующим образом.

```
protected void setup() {
    // Вывод приветственного сообщения
    System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
    // Получение названия книги из командной строки
    Object[] args = getArguments();
    if (args != null && args.length > 0) {
        targetBookTitle = (String) args[0];
        System.out.println("Trying to buy "+targetBookTitle);
        // Добавление TickerBehaviour, который отправляет запрос продавцу
        // каждую минуту
        addBehaviour(new TickerBehaviour(this, 60000) {
            protected void onTick() {
                myAgent.addBehaviour(new RequestPerformer());
            }
        });
    }
    else {
        // Прекратить работу агента
        System.out.println("No target book title specified");
        doDelete();
    }
}
```

В приведенном классе агента-покупателя использовано поведение RequestPerformer, которое позволяет генерировать сообщения нескольким агентам-продавцам и получать от них ответы. Как только будет получен ответ о наличии книги от одного из агентов-продавцов, то на этот ответ генерируется сообщение о принятии предложения. Агент-продавец ожидает запрос от агента-покупателя. В качестве запроса может быть либо «какова стоимость книги», либо «сделать заказ». Для реализации указанного поведения необходимо использовать два вида циклического поведения: согласно первому – обслуживаются запросы на цены, согласно второму – обслуживаются конкретные заказы. Также необходимо реализовать функцию обновления книжного каталога после добавления новой книги для продажи.

```
import jade.core.Agent;
import jade.core.behaviours.*;
import java.util.*;

public class BookSellerAgent extends Agent {
    // Каталог книг для продажи, в котором содержатся названия книг и цены
    private Hashtable catalogue;
    // GUI для добавления пользователем новой книги в каталог
    private BookSellerGui myGui;
    // Инициализация агента
    protected void setup() {
        // Создание каталога
        catalogue = new Hashtable();
        // Создание и отображение GUI
        myGui = new BookSellerGui(this);
        myGui.show();
        // Реализация поведения агента в ответ на запрос по ценам
        addBehaviour(new OfferRequestsServer());
        // Реализация поведения агента в ответ на заказ конкретной книги
        addBehaviour(new PurchaseOrdersServer());
    }
    // Очистка памяти
```

```

protected void takeDown() {
    myGui.dispose();
    System.out.println("Seller-agent "+getAID().getName()+"
terminating.");
}
/**
    Вызывается из GUI, когда пользователь добавил новую книгу для продажи
*/
public void updateCatalogue(final String title, final int price) {
    addBehaviour(new OneShotBehaviour() {
        public void action() {
            catalogue.put(title, new Integer(price));
        }
    });
}
}

```

Использованный здесь класс `BookSellerGui` представляет собой простой Swing GUI. Его описание доступно в документации к пакету `javax.swing.*`.

### ***Коммуникация агентов***

Наиболее важной особенностью JADE является предоставление агентам возможности коммуникации. В JADE использована идея коммуникации путем асинхронной передачи сообщений. Каждый агент владеет почтовым ящиком, где хранятся в виде очереди сообщения, полученные от других агентов. Как только в очереди появляется новое сообщение, агент получает от среды извещение, далее он выбирает из очереди новое сообщение для обработки.

Сообщения, которыми обмениваются агенты JADE, специфицированы в языке ACL (Agent Communication Language), который является международным стандартом для обеспечения интероперабельности агентов [2]. Согласно указанному стандарту сообщение состоит из следующих полей:

- отправитель сообщения;
- список получателей сообщения;
- коммуникативная интенция (также называемая «перформатив»), указывающая на то, что необходимо достичь отправителю сообщения. Например, перформатив `REQUEST` означает, что отправитель ожидает выполнения действия; перформатив `INFORM` означает, что отправитель информирует получателя о некотором факте; перформатив `QUERY_IF` означает, что отправитель ожидает выполнения заданного условия; `CFP` (Call For Proposal), `PROPOSE`, `ACCEPT_PROPOSAL`, `REJECT_PROPOSAL` означает, что отправитель и получатель участвуют в переговорах;
- содержание сообщения. Например, описание действия, которое должно быть выполнено в `REQUEST`-сообщении, или факт, который необходимо сообщить в `INFORM`-сообщении;
- язык содержимого. Агент-отправитель и агент-получатель сообщения должны распознавать и поддерживать синтаксис языка, на котором формулируется содержание сообщения;
- онтология – словарь термов, используемых в содержимом сообщения, и их семантика. Агент-отправитель и агент-получатель сообщения должны одинаково понимать смысл одного сообщения;

- специальные поля, которые используются для управления несколькими параллельными диалогами и специфицируют время ожидания ответа.

Для реализации сообщения в JADE поддерживается класс `jade.lang.acl.ACLMessage`, в котором устанавливаются значения указанных полей сообщения. Для отправки сообщения необходимо заполнить поля класса `ACLMessage` и вызвать метод `send()` класса `Agent`. Например, в следующем фрагменте программы агент «Peter» получает информацию о том, что сегодня дождливая погода:

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-forecast-ontology");
msg.setContent("Today it's raining");
send(msg);
```

Для реализации запроса агента-покупателя агенту-продавцу о стоимости книги необходимо использовать перформатив `CFP`. Для реализации ответа агента-продавца о стоимости книги необходимо использовать перформатив `PROPOSE`. Для реализации ответа агента-покупателя о том, что он принимает предложение агента-продавца и готов оформить заказ, необходимо использовать перформатив `ACCEPT_PROPOSAL`. Если требуемая книга отсутствует в каталоге, то необходимо использовать перформатив `REFUSE`. В содержании сообщений, отправляемых агентом-покупателем, будет храниться название запрашиваемой книги. В содержании сообщения, использующего перформатив `PROPOSE`, будет храниться стоимость книги. Сообщение, использующее перформатив `CFP`, формируется следующим образом:

```
// Запрос агенту-продавцу о стоимости книги
ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.length; ++i) {
    cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
myAgent.send(cfp);
```

Агент, получивший сообщение, может прочитать его при помощи метода `receive()`, который выбирает первое сообщение из очереди сообщений либо возвращает `null`, если в очереди нет ни одного сообщения.

```
ACLMessage msg = receive();
if (msg != null) { // Обработка сообщения }
```

Для обработки сообщений от агента-покупателя необходимо реализовать поведение типа `OfferRequestsServer` и `PurchaseOrdersServer`. Это циклическое поведение, при котором в каждом вызове метода `action()` необходимо проверять и обрабатывать полученное сообщение.

```
/**
```

```
    Внутренний class OfferRequestsServer.
```

```
    Это поведение используется агентом-продавцом для обслуживания запросов от агентов-покупателей. Если запрашиваемая книга присутствует в каталоге, то агент-продавец отправляет сообщение с предложением купить книгу указанной стоимости. Иначе агент-продавец отправляет отказ.
```

```
*/
```

```
private class OfferRequestsServer extends CyclicBehaviour {
```



```

public void action() {
    MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage.CFP);
    ACLMessage msg = myAgent.receive(mt);
    if (msg != null) {
        // Получение сообщения и его обработка
        String title = msg.getContent();
        ACLMessage reply = msg.createReply();
        Integer price = (Integer) catalogue.get(title);
        if (price != null) {
            // Требуемая книга находится в продаже. Отправляется цена
            КНИГИ
            reply.setPerformative(ACLMessage.PROPOSE);
            reply.setContent(String.valueOf(price.intValue()));
        }
        else {
            // Требуемая книга в продаже отсутствует.
            reply.setPerformative(ACLMessage.REFUSE);
            reply.setContent("not-available");
        }
        myAgent.send(reply);
    }
}
}
}

```

Описанное поведение агента должно быть реализовано, только если будет получено новое сообщение с перформативом CFP. В классе Behaviour предусмотрен метод block(), который блокирует действия агента. Как только новое сообщение появляется в очереди, все заблокированные действия агента становятся доступными для выполнения и обработки нового сообщения.

```

public void action() {
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
        // Сообщение получено. Обработка
        ...
    }
    else {
        block();
    }
}
}

```

Класс RequestPerformer позволяет отправить сообщение нескольким агентам-продавцам и дождаться ответа. Если полученный ответ содержит перформатив PROPOSE, то агент-покупатель отправляет сообщение, содержащее перформатив ACCEPT\_PROPOSAL.

```

/**
    Внутренний класс RequestPerformer.
    Класс используется агентом-покупателем для формирования и отправки запроса
    агенту-продавцу на покупку книги.
*/
private class RequestPerformer extends Behaviour {
    private AID bestSeller; // Агент, сделавший самое выгодное предложение
    private int bestPrice; // Самая выгодная цена
    private int repliesCnt = 0; // Счетчик ответов агента-продавца
    private MessageTemplate mt; // Шаблон ожидаемого сообщения
    private int step = 0;
    public void action() {
        switch (step) {
            case 0:
                // Отправка cfp всем продавцам

```

```

ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.length; ++i) {
    cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
cfp.setConversationId("book-trade");
cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Уникальное значение
myAgent.send(cfp);
// Подготовка шаблона для ответа
mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-
trade"),
                        MessageTemplate.MatchInReplyTo(cfp.getReplyWith()))
;

step = 1;
break;
case 1:
// Прием всех предложений/отказов от агентов-продавцов
ACLMessage reply = myAgent.receive(mt);
if (reply != null) {
    // Ответ получен
    if (reply.getPerformative() == ACLMessage.PROPOSE) {
        // Получена стоимость книги
        int price = Integer.parseInt(reply.getContent());
        if (bestSeller == null || price < bestPrice) {
            // Предложена наилучшая стоимость
            bestPrice = price;
            bestSeller = reply.getSender();
        }
    }
    repliesCnt++;
    if (repliesCnt >= sellerAgents.length) {
        // Получены все ответы
        step = 2;
    }
}
else {
    block();
}
break;
case 2:
// Отправка запроса на покупку книги продавцу, предложившему наилучшую
цену
ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
order.addReceiver(bestSeller);
order.setContent(targetBookTitle);
order.setConversationId("book-trade");
order.setReplyWith("order"+System.currentTimeMillis());
myAgent.send(order);
// Подготовка шаблона для ответного сообщения
mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-
trade"),
                        MessageTemplate.MatchInReplyTo(order.getReplyWith()))
;

step = 3;
break;
case 3:
// Получение ответа
reply = myAgent.receive(mt);
if (reply != null) {
    // Ответ получен
    if (reply.getPerformative() == ACLMessage.INFORM) {
        // Успешная покупка

```

```

        System.out.println(targetBookTitle+" successfully purchased.");
        System.out.println("Price = "+bestPrice);
        myAgent.doDelete();
    }
    step = 4;
}
else {
    block();
}
break;
}
}
public boolean done() {
    return ((step == 2 && bestSeller == null) || step == 4);
}
} // class RequestPerformer

```

### **Размещение и поиск сервисов на доске объявлений**

Рассматриваемые агенты-покупатели могут взаимодействовать только с теми агентами-продавцами, которых они знают. Устранить это ограничение позволяет модуль управления общим каталогом, который позволяет одним агентам размещать на доске объявлений информацию о предоставляемых ими сервисах, а другим – находить и использовать нужные им сервисы (рис. 1.2). Количество агентов, предоставляющих сервисы, и количество агентов, использующих сервисы, не фиксировано. Услуги доски объявлений предоставляет специальный агент JADE, называемый «df» (directory facilitator). С каждым df-агентом можно взаимодействовать при помощи ACL-сообщений и языка, при помощи которого формируется текстовое содержимое сообщения [2]. Для управления доской объявлений JADE предоставляет класс `jade.domain.DFService`.

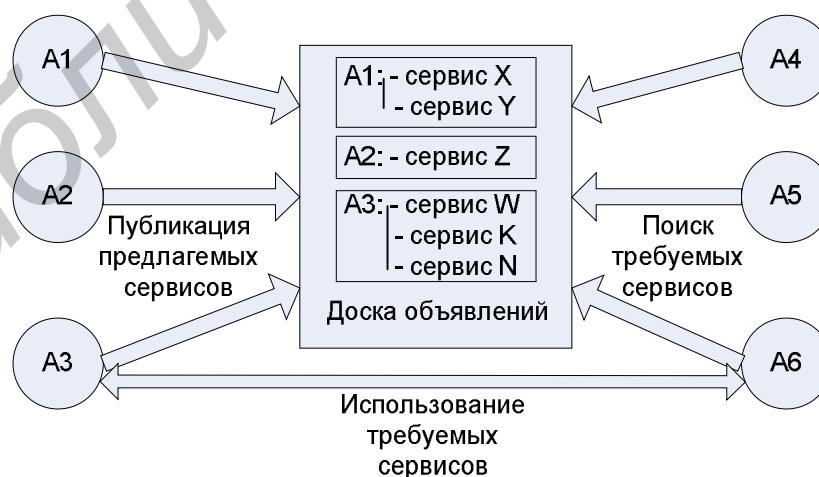


Рис. 1.2. Взаимодействие агентов с доской объявлений

Если агенту необходимо опубликовать один или более сервисов, то он должен предоставить df-агенту описание, включающее его AID, язык, на котором сформулировано сообщение, онтологию и публикуемые сервисы. Язык и онтология необходимы для того, чтобы другие агенты могли взаимодействовать с агентом, который публикует свои сервисы. Для каждого публикуемого сервиса необходимо предоставлять тип сервиса, название сервиса, язык и онтологию, которые необходимы для использования этого сервиса. Для реализации описания публикуемого сервиса необходимо использовать классы `DFAgentDescription`, `ServiceDescription`, `Property`, размещенные в пакете `jade.domain.FIPAAgentManagement`.

Рассмотрим фрагмент программы, реализующей агента-продавца, в котором агент-продавец публикует информацию о своих услугах.

```
protected void setup() {
    ...
    // Регистрация сервиса по продаже книг
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType("book-selling");
    sd.setName("JADE-book-trading");
    dfd.addServices(sd);
    try {
        DFService.register(this, dfd);
    }
    catch (FIPAException fe) {
        fe.printStackTrace();
    }
    ...
}
```

В этом примере не специфицированы язык, онтология, а также специфические свойства сервиса [5]. Перед завершением работы агента необходимо отменить регистрацию сервисов на доске объявлений.

```
protected void takeDown() {
    // Отмена регистрации
    try {
        DFService.deregister(this);
    }
    catch (FIPAException fe) {
        fe.printStackTrace();
    }
    // Закрыть пользовательский интерфейс
    myGui.dispose();
    System.out.println("Seller-agent "+getAID().getName()+" terminating.");
}
```

При поиске требуемого сервиса агент должен предоставить df-агенту шаблон поиска. Результатом поиска будет являться список описаний сервисов, подходящих для указанного шаблона. Описание сервиса будет считаться подходящим под указанный шаблон, если все поля, указанные в шаблоне,

присутствуют и в описании сервиса с теми же значениями. Рассмотрим фрагмент реализации агента-покупателя, который динамически находит всех агентов-продавцов, предоставляющих услуги по продаже книг.

```
public class BookBuyerAgent extends Agent {
    // Название требуемой книги
    private String targetBookTitle;
    // Список доступных агентов-продавцов
    private AID[] sellerAgents;
    // Инициализация агента
    protected void setup() {
        // Печать приветственного сообщения
        System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
        // Получение названия книги из командной строки
        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            targetBookTitle = (String) args[0];
            System.out.println("Trying to buy "+targetBookTitle);
            // Добавление TickerBehaviour для отправления запроса агенту-продавцу
            // каждую минуту
            addBehaviour(new TickerBehaviour(this, 60000) {
                protected void onTick() {
                    // Обновление списка агентов-продавцов
                    DFAgentDescription template = new DFAgentDescription();
                    ServiceDescription sd = new ServiceDescription();
                    sd.setType("book-selling");
                    template.addServices(sd);
                    try {
                        DFAgentDescription[] result = DFService.search(myAgent, template);
                        sellerAgents = new AID[result.length];
                        for (int i = 0; i < result.length; ++i) {
                            sellerAgents[i] = result[i].getName();
                        }
                    }
                    catch (FIPAException fe) {
                        fe.printStackTrace();
                    }
                    // Выполнение запроса
                    myAgent.addBehaviour(new RequestPerformer());
                }
            });
        }
    }
    ...
}
```

При отправке запроса на покупку книги необходимо каждый раз обновлять список доступных агентов-продавцов, т.к. они могут динамически появляться и исчезать в системе. Класс `DFService` позволяет осуществлять подписку на рассылку сообщений при появлении нового сервиса при помощи методов `searchUntilFound()` и `searchUntilFound()`.

### Контрольные вопросы

1. Поясните понятие контейнера JADE.
2. Какой из контейнеров JADE становится главным при их запуске?
3. Какая парадигма коммуникации реализована в JADE?
4. Существует ли возможность отправить сообщение агентом самому себе?

## **Задание**

Реализовать многоагентную систему для предоставления товаров или услуг клиентам. Многоагентная система должна содержать несколько видов агентов. Например, агенты, предоставляющие товар или услуги за определенную стоимость; агенты, покупающие указанные товары или услуги; агенты-посредники между агентом-продавцом и агентом – конечным покупателем. Каждый агент получает начальную информацию от пользователя. Общение пользователя и агента происходит при помощи графического пользовательского интерфейса. Один пользователь вводит описания товаров и услуг, а другой пользователь вводит информацию о том, какой товар или услугу необходимо приобрести.

### **Варианты индивидуальных заданий**

1. Бронирование авиа-, автобусных или железнодорожных билетов. В решении участвуют два агента. Агент-кассир предоставляет информацию о свободных билетах и продает билеты. Агент-покупатель осуществляет поиск необходимых рейсов и бронирует билеты. Расписание вводится диспетчером. Информация о необходимом билете вводится пользователем-покупателем. Первоначальное состояние билетов – все свободны.

2. Бронирование путевки у туристических операторов. Агент-продавец является представителем туроператора и предоставляет информацию о туристических путевках. Агент-покупатель является представителем покупателя туристической путевки. Информация о путевках вводится туроператором. Информация о необходимой путевке вводится пользователем-покупателем.

3. Сбор информации для покупки сотового телефона. Агент-продавец является представителем продавца сотовых телефонов и представляет информацию о моделях, ценах и других свойствах телефонов. Агент-покупатель – представителем покупателя и осуществляет поиск и покупку телефона требуемых свойств. Информация о товарах размещается в каталоге товаров и изменяется пользователем-продавцом. Информация о требуемой покупке вводится пользователем-покупателем.

4. Сбор информации для покупки мебели. Агент-продавец является представителем продавца мебельного магазина и предоставляет информацию о товаре, присутствующем в магазине. Пользователь-продавец вводит информацию о предлагаемом товаре. Агент-покупатель является представителем покупа-

теля мебели. Пользователь-покупатель сообщает своему агенту информацию о требуемой мебели.

5. Сбор информации для покупки недвижимости. Агент-риэлтер-продавец является представителем риэлтерской фирмы и предоставляет информацию о продаваемых объектах недвижимости, ценах и свойствах. Пользователь-риэлтер вводит информацию о предлагаемых объектах недвижимости: квартиры, коттеджи, гаражи, офисы и т.д. Агент-риэлтер-покупатель является представителем риэлтерской фирмы, осуществляющей поиск объектов недвижимости для своих клиентов. Пользователь-риэлтер в данном случае передает информацию своему агенту о требуемых объектах недвижимости для своих клиентов. Агент-покупатель является представителем покупателя недвижимости. Конечный покупатель передает информацию своему агенту о требуемом объекте недвижимости.

6. Сбор информации для покупки автомобиля. Агент-продавец является представителем салона по продаже автомобилей и предоставляет информацию об автомобилях. Пользователь-продавец вводит информацию о продаваемых в салоне автомобилях. Агент-покупатель является представителем покупателя автомобиля. Пользователь-покупатель передает информацию своему агенту о требуемом автомобиле.

7. Поиск места работы. Агент-наниматель является представителем кадрового агентства и предоставляет информацию о вакантных местах. Сотрудник агентства вводит информацию о существующих вакансиях на рынке труда. Агент-работник является представителем работника, нанимающегося на работу. Пользователь-работник передает информацию своему агенту о требуемой должности, уровне заработной платы, расположении, условиях труда и т.д.

8. Распределение студентов на предприятия. В решении задачи участвуют три вида агентов: агент-наниматель, агент – ответственный за распределение, агент-студент. Агент-наниматель предоставляет информацию о предприятии, должностях и условиях труда. Агент – ответственный за распределение осуществляет поиск мест распределения студентов. Агент-студент предоставляет информацию о прописке, среднем балле студента, его предпочтениях и другую информацию, необходимую для выбора места распределения.

## **Лабораторная работа №2**

### **Моделирование документов в среде Lotus Notes и реализация функций их обработки**

**Цель работы:** приобрести навыки представления информации с использованием иерархической документо-ориентированной модели данных, а также навыки реализации этой модели в среде Lotus Notes.

#### **Теоретические сведения**

Документо-ориентированные офисные системы на сегодняшний день являются важным классом программного обеспечения организации, т.к. являются фактором их конкурентоспособности. Они аккумулируют знания организации в форме документов. Появившись в системе, знания становятся доступными сотрудникам, которые могут их использовать в своей деятельности. Среди наиболее распространенных следует отметить Lotus Notes (Lotus Development), Visual Document Library (IBM), Relation Document Manager (Interleaf), Visual Recall (Xsoft), GroupWise (Novell), DOCS Open (PC DOCS) [11].

Среда Lotus Notes является системой управления документо-ориентированными базами данными и состоит из следующих компонент [10;11]:

- Lotus Notes – клиент для навигации и поиска документов в документо-ориентированных базах данных;
- Lotus Domino – сервер для управления документо-ориентированными базами данных;
- Lotus Designer – для создания документо-ориентированных баз данных, содержащих графические формы, виды и навигаторы, а также функции обработки документов.

К средствам навигации по базе данных в Lotus Notes относятся [10; 11]:

- view – вид для навигации по различным выборкам документов;
- outline – иерархическое отображение видов, где каждому виду можно присвоить пользовательское имя;
- page – страница, подобна форме, на ней можно размещать статический текст, таблицы, секции, иерархии, кнопки и другие элементы дизайна;
- navigator – навигатор для создания графического (кнопочного) меню, при помощи которого можно быстро загружать различные виды;



- `frameset` – множество фреймов, аналогично фреймам web-сайта, которые позволяют отображать различного рода информацию из базы данных (документ, вид, навигатор, графический объект и т.д.) в любом месте экрана, обозначенном как фрейм.

### ***Создание вида***

Вид позволяет организовать выборку документов в списке, где каждый документ представлен в отдельном ряду (до 9 строк на документ), ряды разделены на колонки, каждая колонка содержит определенную информацию о документе (содержимое поля, результат вычислений, простое действие: вычислить дату создания, порядковый номер и др.). В каждой базе данных должен существовать хотя бы один вид. Подмножество документов в виде определяется критерием отбора вида. Самый простой случай `SELECT @ALL` – отображает все документы, хранящиеся в базе. Представление каждого документа зависит от формул и параметров колонок. Сортировка документов в виде осуществляется при помощи признака сортировки в колонке. Категоризация документов осуществляется при помощи категоризированных колонок. Возможно до 32 уровней вложенности, что связано с количеством колонок в виде. Поддерживается также иерархия ответных документов. Возможно до 32 уровней вложенности с учетом их категорий. По всем документам и/или по категориям могут быть показаны обобщающие результаты, для чего используются простые функции: `total`, `max`, `avg` и др. Кроме перечисленных атрибутов в виде могут быть акции, автоматизирующие некоторые функции. Набор акций для каждого вида индивидуален. Таким образом, вид отображает краткое содержание множества документов в базе данных.

Рассмотрим различные пути создания видов.

1. `Personal` – это личный вид, создается по умолчанию и доступен только своему создателю. Каждый депозитор может создавать личные виды. Личный вид сохраняется в файле «`desktop.dsk`» рабочей станции пользователя, который его создал, или в самом файле базы данных. Место сохранения вида зависит от уровня доступа пользователя, определенного в ACL (Access Control List) базы данных:

- дизайнер имеет возможность сохранить вид в файле базы данных (\*.nsf);

- редактор, автор или читатель, если установлена опция «Create personal folder/views», имеет возможность сохранить вид в базе данных, иначе – в рабочем пространстве;

- депозитор имеет возможность сохранить вид только в рабочем пространстве.

2. Shared – это общедоступный вид, который может открыть любой пользователь, начиная с депозитора, если доступ к нему не ограничен специально. Создать новый или модифицировать существующий вид может только менеджер, дизайнер или редактор с установленной в ACL опцией «Create shared folder/views».

Использование опции «Personal on first use» в общедоступном виде означает, что вид создается разработчиком как общедоступный, но после первого обращения к нему каждого пользователя создается его личная копия. Разработчик может изменить исходный вид, а личная копия пользователя не изменится. Пользователь может только удалить свой личный вид. С точки зрения защиты информации такие виды ничего не дают, пользователь может создать вид со всеми документами, к которым ему запрещен доступ. Защита базы данных обеспечена на уровне документа. Используя формулу @UserName, можно подстраивать вид под конкретного пользователя.

Тип вида (shared, personal, shared – personal on first use) может быть выбран только при создании вида, позже его нельзя будет изменить.

При создании вида можно определить родительский вид «select a location for the new view». Если это не сделано, то после создания можно включить такой вид в иерархию. Для этого необходимо изменить имя вида. Например, множество «словари» включается в множество «универсальные издания», а те в свою очередь – в множество «литература».

Поле «Inherit design from» означает вид (или папку), у которого будет скопирован дизайн для нового вида. По умолчанию используется тот вид, у которого эта опция уже установлена.

Поле «Selection conditions» означает критерий отбора документов в виде. Например, form = «Student» AND learnform = «бесплатная форма обучения», в данном случае произойдет выборка документов, созданных по форме с именем «Student», а в поле «форма обучения» должно быть указано «бесплатная форма обучения».

Рассмотрим акции по умолчанию, присутствующие в окне проектирования вида:

- edit document – переключение из режима чтения документа в режим редактирования и наоборот;

- categorize – категоризация документов, изменение значения поля категоризации;

- forward – отправка почтой изображения документа, т.е. документа без формы;

- move to folder – перемещение документа в папку;

- remove from folder – удаление документа из папки;

- send document – отправка документа почтой.

Каждый вид имеет следующие свойства: базовые, опциональные, свойства стиля, особые, безопасность.

К базовым свойствам относятся:

- name, alias и comment – эти поля нужны только разработчику.

Ограничения на Name и Alias – по 64 байта на один уровень. Разделитель уровней вложенности – «\». На все уровни отводится 256 байт;

- если название вида в заключено в круглые скобки, то вид будет невидим конечному пользователю. Такие виды могут использоваться для получения выборок при помощи функций @DbColumn и @DbLookup;

- существуют predefined имена видов: (\$Drafts) – черновики документов, (\$Sent) – отправленные, (\$All) – все, (\$Calendar) – календарь, (\$ToDo) – необходимо сделать.

К опциональным свойствам относятся:

- default when database is first opened – вид по умолчанию. Каждая база данных должна иметь вид по умолчанию, который загружается при первом ее открытии пользователем;

- default design for new folders and views – вид по умолчанию, дизайн которого используется при создании нового вида, если не указан иной;

- collapse all when database is first opened – свернуть все категории при загрузке вида в первый раз. При последующей загрузке этого вида категории будут выглядеть так, как при последней работе с ними конечного пользователя;

- show response documents in a hierarchy – отображать ответные документы в виде иерархии. Используется по умолчанию. Если родительский документ не подходит по критерию отбора документов данного вида, то все ответные документы не отображаются. Ответные документы не участвуют в общем

порядке сортировки, а сортируются в пределах группы одного уровня, принадлежащей одному документу-родителю. Если эта опция не установлена, то все документы в виде отображаются независимо друг от друга, т.е. родственные связи не учитываются;

- `show in View menu` – отображать документы в меню. По умолчанию эта опция не включена и загрузить вид можно только через меню «`view->go to->имя вида`».

К опциональным свойствам вида также относятся следующие группы свойств: `on open` и `on refresh`.

Опциональные свойства `on open` определяют события, которые происходят при открытии вида:

- `go to last opened document` – перейти к документу, который был открыт при последнем сеансе работы с базой данных;
- `go to top row` – перейти к первому документу в виде;
- `go to bottom row` – перейти к последнему документу в виде.

Опциональные свойства `on refresh` определяют, как быстро обновляется вид при изменениях. Вид содержит индекс, который представляет собой скрытый от пользователя объект в базы данных и содержащий всю информацию из документов, необходимую для отображения вида на экране:

- `display indicator` – пользователь сам должен обновлять вид при помощи отображаемого индикатора;
- `refresh display` – вид обновляется при любом изменении. Если с базой одновременно работают несколько пользователей, то изменения отображаются у всех на экранах, однако при слишком частом обновлении замедляется работа;
- `refresh display from top row` – обновления происходят начиная с документов верхнего уровня. Эта опция эффективно используется, если документы хронологически упорядочены и изменения происходят только в верхних документах;
- `refresh display from bottom row` – обновления происходят начиная с документов верхнего уровня. Эта опция эффективно используется, если документы хронологически упорядочены и изменения происходят только в документах нижнего уровня.

К свойствам стиля вида относятся:

- `colors` – цвета для фона, итоговых значений, неп прочитанных строк, альтернативных строк;

- `show selection margin` – отображение крайней левой колонки для выбора документов, если ее нет, то она появляется при выборе документов пробелом;
- `extend last column to window width` – расширение последней колонки до границы экрана. Это позволяет максимально использовать пространство экрана. В последней колонке может быть текст неопределенного размера;
- `show column headings` – отображение заголовков колонок. При этом экономится место для показа документов, т.к. строка заголовков колонок не отображается;
  - `line per heading` – количество строк, зарезервированных под заголовки;
  - `line per row` – количество строк, зарезервированных под документ. Если при этом указано «`shrink rows to content`» (сокращать по содержанию), то лишние строки не будут использоваться;
  - `row spacing` – интервал между строками;
  - `refresh index: auto, after first use` – обновлять индекс автоматически в момент первого открытия базы данных на сервере и далее во время работы; `automatic` – даже если базу никто не открывал, индекс поддерживается в актуальном состоянии, открытие в этом случае происходит гораздо быстрее; `manual` – если база данных размещена локально, то обновление индекса выполняется только вручную, если на сервере – запускаются процессы `UPDATE` и `UPDALL`; `automatic, at most n hour` – указывается период обновления индекса, который целесообразно устанавливать равным периоду изменения результатов формул (например, значения кардиограммы, почтовый трафик);
  - `discard index` (удаление индекса) – если вид используется редко, удаление индекса позволит сэкономить дисковое пространство, но при открытии вида потребуются время для создания индекса: `never` – индекс будет обновляться при каждом открытии базы данных и в зависимости от `refresh index` при частом изменении данных; `after each use` – если база данных изменяется регулярно, но редко, например, раз в неделю, то индекс при открытии создается, а при закрытии удаляется; `if inactive for n days` – индекс обновляется в течение `n` дней, когда изменения редки и непредсказуемы;
  - `UPDALL` на сервере удалит индекс в любом случае, т.к. администратор может установить время жизни индексов для всех баз данных или по умолчанию – 45 дней;

- unread marks: none, unread documents only – если неп прочитанный документ скрыт внутри категории, то категория не отмечается, если развернуть категорию, тогда отобразятся отмеченные документы; standard [compute in hierarchy] – если неп прочитанный документ скрыт внутри категории, то отмечается и вся категория;

- from formula – определяет имя формы, с помощью которой будут открываться документы в виде.

К особым свойствам вида относятся свойства, связанные с существованием индексов видов, а также способ отслеживания неп прочитанных пользователем документов и формула формы. Для отображения вида на экране кроме описания его дизайна необходим индекс. Индекс перестраивается каждый раз при изменении данных или при изменении дизайна вида. Пользователь может обновить сам при помощи кнопки «Обновить».

В свойствах защиты вида указываются пользователи, которые имеют доступ к виду. Самый бесправный пользователь может создать свой вид, где будут представлены документы, к которым ему разрешен доступ. Права ограничиваются на уровне документов.

Критерий отбора документов определяет, какие документы базы данных отображаются в этом виде, и задается в панели дизайна вида. Существуют следующие способы определения критерия:

- easy: by author, by date, by field, by form, by form used. Если ничего не указано, то в виде будут отображаться все документы базы;

- formula – «SELECT @формула критерия» – задает формулу критерия. Документ удовлетворяет критерию, если формула истинна. Для связывания различных критериев используются логические операторы: (|, &, !). По умолчанию устанавливается формула для выбора всех документов базы: «SELECT @ALL».

Каждый вид состоит из колонок. Количество колонок ограничено только их суммарной шириной – 22.75” (57.8 см). У колонки аналогично виду существует ряд свойств: column info, sorting, font, numbers, date and time format, title, advanced.

Группа свойств column info:

- Title – текст усекается и переносится (либо нет) на следующую строку;

- Width – ширина колонки в символах;
- Resizable – форматирование по правой границе;
- Hide column – скрытие колонки. Необходима для сортировки, получения данных с помощью функций @DbColumn, @DbLookup или из LotusScript-программ;
- Show twisty when row is expandable – установка знака иерархии. Устанавливается в классифицированных колонках, колонках для ответных документов и основной колонке главных документов;
- Multi-value separator – “ ”, “;”, “;”, New Line, Blank Line. New Line – совместно с использованием нескольких строк для представления одного документа в виде;
- Show responses only – отображение только ответных документов. Может быть использовано только для одной колонки вида;
- Display Values as icons – отображение значения колонки в виде пиктограммы. Формула для вычисления вида пиктограммы должна возвращать целое число, являющееся номером пиктограммы. Например: @If(@Attachments;5;0) – возвращает 5 («скрепка») для документов, в которых присоединены файлы, иначе – 0.

Группа свойств sorting позволяет отсортировать документы по содержанию одной колонки:

- totals – представление в числовой колонке обобщающих результатов по всем главным документам в виде (ответные документы не учитываются). Например: колонка с именем Size отображает размер документа в байтах;
- hide detail rows – отображает значения только для категории, а не для каждого документа.

Группа свойств numbers позволяет использовать функции @Abs, @Month, @Year, @TextToNumber, @DocLength, @Member для определения формата возвращаемых чисел.

В группе свойств advanced можно, например, определить поле Name, в котором указывается имя колонки, используемое в формулах других колонок. Например, первая колонка имеет формулу «@Modified» и имя «\$01», а в формуле другой колонки указано «@Now-\$01», это означает, что значения второй колонки будут равняться разнице между временем модификации документа, указанным в первой колонке, и текущим временем в секундах.

## ***Автоматизация действий пользователя***

Для автоматизации действий пользователей в Lotus Notes используются язык @формул и язык Lotus Script. Для автоматизации формирования поиска значений в полях типа Listbox и Dialog list на основе данных вида используются функции @DbColumn и @DbLookup. Некоторые параграфы документов, содержащие нередактируемые поля, скрываются в режиме чтения документа при помощи поля hide. Параграфом документа называется набор смежных строк документа. Для вызова дополнительной формы при вводе данных используются кнопки типа hotspot. Дополнительные формы могут использоваться в том случае, если необходимо разделить вводимые данные на несколько полей, а потом отображать их в виде единой строки, но с некоторыми сокращениями и/или дополнениями. Например, адрес должен отображаться в виде строки «Иванов И. И., г. Минск, ул. П. Бровки». При этом используется функция @DialogBox. Контроль ввода осуществляется при помощи функции @If.

Рассмотрим пример создания кнопки для добавления встроенных файлов (Attachments):

```
@If(@AttachmentNames = "" ;  
  @Do(  
    @Command([EditGotoField]; "Annotation");  
    @Command([FileImport]);  
    @Command([ViewRefreshFields])  
  );  
  @Prompt([Ok]; "Ошибка"; "В документ может быть вложен только один файл."))
```

### **Контрольные вопросы**

1. Чем отличается иерархия ответных документов от иерархии категорий?
2. Назовите ограничение, накладываемое на количество строк в виде, соответствующих одному документу.
3. Объясните ограничение на количество категорий вида.
4. Для одной из колонок (например «Зарплата») выведите обобщающий результат (например средняя зарплата по отделам).
5. Знает ли депозитор названия полей какой бы то ни было формы, ведь ему это надо для создания вида?
6. Будет ли пользователь с правами менеджера или дизайнера иметь доступ (и какой) к личным видам других пользователей?
7. Укажите в среде Lotus Notes, где устанавливается значение «Create personal folder/views»?
8. Что устанавливается в свойствах вида Security?



9. Может ли пользователь, который работает с видом типа Personal on first use, сделать его общедоступным?

### **Задание**

Выбрать предметную область из списка индивидуальных заданий. Провести исследование предметной области. Выделить сущности, атрибуты сущностей и связи между сущностями. Описать предметную область в виде диаграммы «сущность – связь». Изучить такие компоненты среды Lotus Notes, как форма, документ, вид. Реализовать созданную модель предметной области в виде приложения в среде Lotus Notes. Количество форм должно соответствовать количеству сущностей. Каждой форме должен соответствовать вид. Для созданного приложения разработать все виды навигаторов по документам: outline, page, navigator, frameset. Реализовать указанные в индивидуальном задании функции обработки документов.

### **Варианты индивидуальных заданий**

1. База данных студентов университета. Реализовать функцию вычисления количества студентов в группе и в подгруппе; функцию вычисления количества групп на специальности и на потоке; функцию генерации номера для новой группы; функцию отчисления студента, т.е. перевод его из вида учащихся студентов в вид отчисленных студентов.

2. База данных преподавателей университета. Реализовать функцию вычисления стажа работы преподавателя; функцию вычисления количества лет, оставшихся до пенсионного возраста (для мужчин – 60 лет, для женщин – 55 лет); функцию формирования в начале каждого месяца вида/папки по преподавателям, у которых день рождения в текущем месяце, а также функцию рассылки поздравлений.

3. База данных подразделений университета. Реализовать функцию вычисления количества подразделений, находящихся в отношении подчинения; функцию перевода подразделения в подчинение к другому подразделению, создания автономного подразделения из подчиненного подразделения.

4. База данных библиотеки. Реализовать функцию вычисления объема библиотечного фонда, т.е. общего количества экземпляров всех книг в библиотеке. Реализовать вид, в котором отображаются читатели с просроченными книгами на заданный промежуток времени.

5. База данных успеваемости студентов. Реализовать функцию вычисления среднего балла студента; функцию вычисления количества экзаменов и зачетов, которые студент уже сдал и которые еще нужно сдать.

6. База данных материально-технического оборудования кафедры. Реализовать функцию вычисления общей стоимости сборного оборудования по стоимости комплектующих его материалов.

7. База данных студентов, проживающих в общежитии. Реализовать функцию вычисления количества заселенных комнат, количества свободных комнат; функцию заселения и выселения студента из общежития.

8. База данных материально-технического обеспечения общежития. Реализовать функцию вычисления количества электроприборов в сети общежития. Сигнализировать о превышении приборов в сети в зависимости от нагрузки, на которую рассчитана сеть. Реализовать возможность регистрации заявок и жалоб.

9. База данных расписания занятий. Реализовать функцию вычисления количества «форточек»; функцию поиска замен и переносов занятий в текущем расписании.

10. База данных программного обеспечения. Реализовать функцию вычисления общей стоимости установленного программного обеспечения в подразделении университета с учетом количества рабочих мест. Стоимость рабочего места задается пользователем.

11. База данных курсовых проектов. Реализовать функцию вычисления количества курсовых проектов на одного преподавателя; функцию вычисления среднего балла по курсовым проектам у конкретного преподавателя.

12. База данных лабораторных работ по некоторой дисциплине. Реализовать функцию вычисления количества часов по всем лабораторным работам. Сигнализировать о превышении или недоборе часов по сравнению с запланированным количеством.

13. База данных мероприятий (конференции, семинары, выставки). Реализовать функцию ежедневной проверки сроков; функцию рассылки напоминаний и информационных писем участникам.

14. База данных абитуриентов. Реализовать функцию вычисления общего балла, полученного абитуриентом на вступительных экзаменах; функцию зачисления абитуриента на факультет в зависимости от полученных баллов и с учетом приоритетов в зачислении.

## **Лабораторная работа №3**

### **Реализация операций обработки семантических сетей на базе языка Semantic Code Programming**

**Цель работы:** приобрести навыки использования графового процедурного языка программирования Semantic Code Programming для создания операций обработки знаний, представленных в виде семантических сетей.

#### **Теоретические сведения**

Современными тенденциями развития интеллектуальных систем являются увеличение объемов и усложнение структуры перерабатываемых знаний, усложнение семантики и расширение типологии задач, решаемых интеллектуальной системой, интеграция различных моделей обработки знаний. Таким образом, наиболее адекватными средствами создания прикладных интеллектуальных систем различного назначения являются параллельные графовые языки представления и обработки знаний, базирующиеся на универсальных графовых моделях знаний. Одним из таких универсальных средств создания интеллектуальных систем является семейство графовых семантических языков Semantic Code (SC) для представления и обработки знаний сложноструктурированных предметных областей [6], а также среда разработки прикладных интеллектуальных систем, включающая средства ввода и редактирования баз знаний, средства разработки и отладки программ. Указанное семейство языков включает язык-ядро SC, логический язык Semantic Code Logic (SCL) и язык программирования Semantic Code Programming (SCP). Языки семейства SC имеют две эквивалентные модификации: графовую и строковую.

Язык Semantic Code Programming (SCP) является графовым процедурным языком программирования, ориентированным на обработку знаний, представленных в виде семантических сетей [7]. В языке реализованы базовые операции работы со специальными графовыми конструкциями: узлами, дугами, трехэлементными конструкциями и пятиэлементными конструкциями (рис. 3.1). Кроме того, некоторые типы операторов языка SCP (scp-операторов) работают либо с одиночной конструкцией, либо с множеством конструкций. На месте любого узла может стоять или узел, или дуга, или элемент неопределённого типа. Рассмотрим примеры графовых конструкций.

Наиболее распространенной в языке SC является трёхэлементная конструкция, которая состоит из узла, некоторого элемента и дуги, соединяющей их. Элементом может быть либо узел, либо дуга. Так же как и в случае трёхэлементной конструкции, в пятиэлементной конструкции на месте узла  $v4$  (рис.3.1) может быть либо узел, либо дуга.

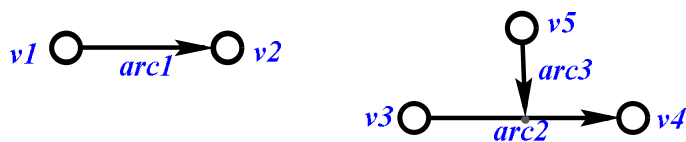


Рис. 3.1. Изображение трехэлементной (слева) и пятиэлементной (справа) конструкций

Рассмотрим взаимосвязь графовых конструкций и операторов языка SCP: большинство scp-операторов представляют собой классы, объединяющие операторы с определёнными действиями над графовыми конструкциями. В свою очередь классы состоят из подклассов, разделяющих операторы на две группы: операторов, использующих одну конструкцию и операторов, использующих множество конструкций. Каждый подкласс содержит операторы, различающиеся между собой только типом конструкций, с которыми они работают. В качестве примера рассмотрим класс операторов *search*, который включает два подкласса операторов *searchEl* и *searchSet*. Первый подкласс состоит из операторов *searchElStr3*, *searchElStr5*. Второй подкласс состоит из операторов *searchSetStr3*, *searchSetStr5*. По такому принципу строятся названия многих scp-операторов:

- название начинается с типа оператора, определяющего действие над графом (например *search* – «поиск»);
- затем следует название семейства оператора, определяющее количество конструкций, над которыми производится действие (*El* (element) – «элемент» и *Set* (set) – «набор»);
- в конце определяется структура конструкций, над которыми производится действие (*Str3*, *Str5* «structure» – «структура»; соответственно с тремя и пятью элементами).

Важным понятием языка SCP является понятие атрибута. Атрибутом называется узел графа, уточняющий дугу – «пару принадлежности» (пара принадлежности – два элемента графа, связанные между собой дугой, причём

элемент графа, в который дуга входит, может являться либо узлом, либо дугой). Количество уточняющих атрибутов для одной пары принадлежности не ограничено.

### **Структура scp-программы**

Рассмотрим пример scp-программы:

```
                Исходный текст scp-программы
1. #include "scp_keynodes.scsy"
2. procedure(hello_world,
3. [[
           message =c= /" *** Hello, world! *** "/;
       ]],
4. [{
       }],
5. {[}] )
6. printNl([l_: fixed_: message])
7. printEl([l_: fixed_: message])
8. return()
   end
```

В строке 1 подключается файл scp-синонимов (аналог прототипов в заголовочных файлах \*.h языка C), который содержит пути к ключевым узлам.

В строках 2–5 описывается заголовок программы, который содержит:

- имя программы `hello_world`;
- множество константных узлов, ограниченное двойными квадратными скобками и включающее только один узел `message` с текстовым содержимым «Hello, world!»;
- множество переменных узлов, ограниченное двойными скобками, причем внутренние скобки фигурные. В примере это множество является пустым;
- множество входных и выходных параметров программы. В примере это множество является пустым.

Далее в строках 6–9 идёт блок операторов scp-программы, состоящий из оператора `printNl` (выводит текстовое содержимое своего параметра в файл протоколирования или в консоль), оператора `printEl` (выводит входящие и выходящие дуги для указанного элемента в файл протоколирования или в консоль), оператора `return` и признака конца программы – ключевого слова `end`.

Далее необходимо запустить программу на исполнение и проверить результат в файле протоколирования `_out_log` (лог):

```
...
Executing hello_world_op1(printNl) of prg hello_world(023B5C80)
  *** Hello, world! ***
...
```

Среди множества системных сообщений (в местах многоточия) можно найти сообщение о запуске первого оператора рассмотренной scp-программы «Executing hello\_world\_op1(printNl)», а в следующей строке лога можно увидеть

пользовательское сообщение, а именно строку «\*\*\*Hello, world!\*\*\*» , что является подтверждением корректной работы программы.

### ***Система операций обработки знаний***

Каждая операция обработки знаний представляет собой независимого целенаправленного агента, условием инициирования которого является появление в базе знаний специальной структуры. Результатом работы агента также является появление в базе знаний некоторой структуры, которая может служить условием инициирования того же или другого агента. Условие инициирования операции – это информационная конструкция, записанная на языке представления знаний, а также признак отправки задания на выполнение [8].

Рассмотрим процесс погружения новой введенной конструкции в базу знаний. Процесс погружения в базу знаний включает несколько этапов проверки корректности вводимой конструкции, а также интеграции новой конструкции с базой знаний.

**Этап 1.** Поиск в базе знаний узлов, имеющих идентификаторы, совпадающие с идентификаторами узлов введенной конструкции. Найденные идентичные узлы должны быть склеены.

**Этап 2.** Верификация синтаксиса отношений вводимой конструкции. Рассматривается множество всех используемых отношений. Все кортежи всех отношений должны соответствовать схемам этих отношений. Элементы кортежей отношений должны принадлежать соответствующим доменам. Если встречается ошибка, то дальнейшее погружение в базу знаний не выполняется.

**Этап 3.** Склейка конструкций по идентифицирующему их контексту. Например, склейка равных кортежей отношения, склейка равных полностью заданных множеств, склейка кратных элементов канторовских множеств.

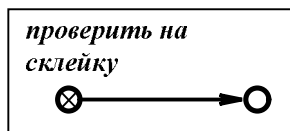
**Этап 4.** Склейка конструкций на основании утверждений о единственности существования.

Рассмотрим операцию поиска конструкций, подлежащих склеиванию. Назначение операции – проверка вхождения заданного sc-элемента в какую-либо sc-конструкцию, в которой возможно склеить некоторые элементы. Для заданного sc-элемента операция ищет некоторую окрестность, элементы которой подлежат склейке. Склейка возможна при наличии окрестности элемента, совпадающей с одним из известных шаблонов. Каждый шаблон соответствует

одному случаю склейки. Заданный SC-элемент должен обязательно входить в конструкцию, подлежащую склейке. Если такая окрестность найдена, т.е. некоторые элементы данной окрестности можно склеить, тогда она помечается специальным образом, после чего она будет обработана отдельной операцией склейки.

### ***Условие применения операции обработки знаний***

Условием применения операции подготовки к склейке является появление в SC-памяти конструкции вида:



Данная конструкция означает, что во множество элементов, окрестность которых необходимо проверить на возможность склеивания, был добавлен новый sc-элемент. Возможные пути добавления элемента в это множество:

1. При вводе новых sc-конструкций в базу знаний все вводимые элементы можно помечать как «кандидатов» на склейку с существующими элементами в базе знаний.

2. Непосредственно при склейке любой конструкции меняется ее контекст, вследствие чего все склеенные элементы необходимо снова проверить на возможность дальнейшей склейки.

3. При склейке элементов также могут образовываться конфликтные ситуации. Например, при склейке двух узлов, в которые проведены дуги из одного общего узла, образуются 2 кратные дуги. Поэтому «кандидатами» на дальнейшую склейку являются и все элементы, непосредственно связанные с только что склеенными.

### ***Результат операции обработки знаний***

Операция подготовки конструкций к склейке перебирает все конструкции, в которых присутствует заданный sc-элемент, и которые можно подвергнуть операции склейки. Каждая найденная конструкция помечается таким образом, чтобы для нее запустилась операция склейки. Для этого указывается вид найденной конструкции и проводится дуга из узла «склеить». Проведение этой дуги является условием срабатывания операции склейки.

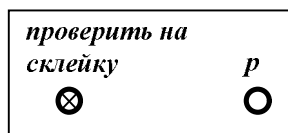
В зависимости от вида найденной конструкции возможны 3 варианта результирующей конструкции:

|  |   |
|--|---|
|  | <p>Найдено два совпадающих кортежа. Соединяем их отношением «совпадающие кортежи» и проводим дугу из узла «склеить». Проведение этой дуги является условием применения операции склейки, которая склеит 2 указанных кортежа. Благодаря явному указанию на то, что склеить необходимо именно кортежи, операция склейки сразу применит к данной конструкции нужный алгоритм</p> |
|  | <p>Найдено два совпадающих множества. Соединяем их отношением «совпадающие множества» и проводим дугу из узла «склеить». Проведение этой дуги является условием применения операции склейки, которая склеит 2 указанных множества</p>   |
|  | <p>Найдены кратные дуги, выходящие из узла, который не может иметь кратных дуг. Соединяем их отношением «совпадающие дуги» и проводим дугу из узла «склеить». Проведение этой дуги является условием применения операции склейки, которая склеит 2 указанные дуги</p>   |

Если искомым конструкций не найдено, операция заканчивает свою работу. Используемые в операции ключевые узлы: «*совпадающие кортежи\**», «*совпадающие множества\**», «*совпадающие дуги\**», «*склеить*», «*rel без совпадающих кортежей*», «*проверить на склейку*».

Алгоритм операции подготовки вводимых конструкций к склейке состоит из следующих шагов:

1. Удаляем дугу, указывающую, что окрестность данного sc-элемента подлежит проверке на возможность склейки.



Дальнейшая работа производится с sc-элементом p.

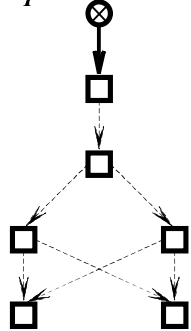
2. Проверяем, входит ли sc-элемент p в какие-либо конструкции, подходящие для склейки.

Такие типичные конструкции прописаны в самой операции. Например, склейке подлежат sc-конструкции вида:



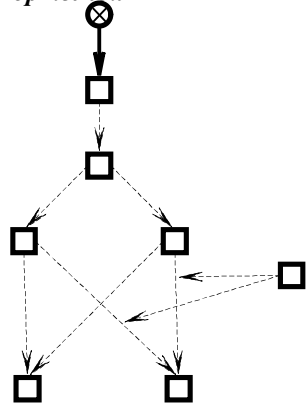
sc-конструкции, в которых склеиваются кортежи

*rel без совпадающих кортежей*



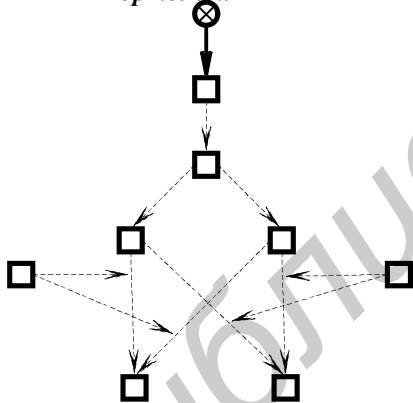
В отношении, которое не может содержать одинаковых кортежей, входят два кортежа, элементы которых совпадают, причем оба элемента не имеют атрибутов, т.е. два указанных кортежа совпадают

*rel без совпадающих кортежей*



В отношении, которое не может содержать одинаковых кортежей, входят два кортежа, элементы которых совпадают, причем один из элементов входит в кортежи под одинаковым атрибутом, а другой элемент не имеет атрибута, т.е. два указанных кортежа совпадают

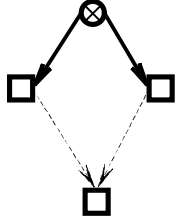
*rel без совпадающих кортежей*



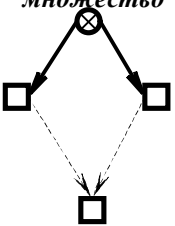
В отношении, которое не может содержать одинаковых кортежей, входят два кортежа, элементы которых совпадают, причем оба элемента входят в кортежи под одинаковыми атрибутами, т.е. два указанных кортежа совпадают

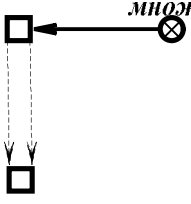
sc-конструкции, в которых склеиваются множества

*канторовское множество*



Имеются два множества, элементы которых полностью совпадают. sc-элементы, соответствующие таким множествам, подлежат склейке. Если данные множества являются канторовскими, то дуги, входящие в каждый элемент из обоих множеств, также склеиваются

|  |   |
|--|---|
| <p style="text-align: center;"><i>канторовское множество</i></p>  | <p>Если элементы обоих множеств совпадают, но множества не канторовские, то склеиваются только sc-элементы, соответствующие множествам. При этом образуются кратные дуги, ведущие в элементы множеств</p> |
|--|---|

| sc-конструкции, в которых склеиваются кратные дуги   |  |
|--|--|
| <p style="text-align: center;"><i>канторовское множество</i></p>  | <p>Эта sc-конструкция означает, что в канторовское множество один и тот же элемент входит дважды. Это противоречит определению канторовского множества, следовательно, мы должны склеить кратные дуги, входящие в элементы канторовского множества</p> |

3. Последовательно рассматриваем каждую из описанных sc-конструкций. Для каждой конструкции необходимо проверить, существует ли она в sc-памяти, причем одним из sc-элементов, входящих в конструкцию, обязательно должен быть заданный sc-элемент  $p$ .

Если рассмотрены все конструкции, то операция заканчивает работу (шаг 7).

4. В текущей конструкции по очереди заменяем все переменные элементы на sc-элемент  $p$ . При этом тип переменного элемента должен совпадать с типом элемента  $p$ .

Таким образом, получаем фиксированный шаблон поиска, в котором один из элементов известен (sc-элемент  $p$ ), а другие требуется найти.

Если последовательно заменили все переменные узлы текущей sc-конструкции, то переходим к рассмотрению следующей конструкции (шаг 3).

5. Осуществляем поиск по полученному шаблону.

6. Если искомая конструкция не найдена, то генерируем следующий шаблон поиска (переходим на шаг 4).

Если такая конструкция найдена, то необходимо проверить, не была ли она уже помечена, как подлежащая склейке, на предыдущих шагах. Для этого проверяем, не достроены ли к этой конструкции результирующие элементы. Если определенные элементы конструкции соединены отношением «совпадающие кортежи», «совпадающие множества» или «совпадающие дуги», то это значит, что операция уже обработала данную конструкцию и пометила ее как подлежащую склейке (переходим на шаг 4).

Если данная конструкция обрабатывается впервые, то помечаем, что она подлежит склейке. На этом шаге формируется результирующая конструкция. В зависимости от типа обрабатываемой конструкции, возможны следующие варианты:

|   |  |
|---|--|
|  | $m$ – множество найденных совпадающих кортежей |
|  | $m$ – множество найденных совпадающих множеств |
|  | $m$ – множество найденных совпадающих дуг      |

Переходим на шаг 4.

7. Когда все sc-конструкции проверены, операция завершается.

### Контрольные вопросы

1. Как изображается 7-элементная графовая конструкция в языке SC?
2. На какие группы делятся операторы языка SCP?
3. По какому принципу построено название scp-оператора genElStr5?
4. Дайте определение процесса погружения sc-конструкции в базу знаний.
5. Поясните семантику операции склейки sc-элементов в графовой памяти.
6. Из каких компонент состоит автономная целенаправленная sc-операция обработки базы знаний?
7. Поясните семантику операции склейки кортежей отношения.
8. Поясните семантику операции склейки множеств.
9. Дайте определение понятию «условие применения операции над графовой памятью».

### Задание

Разработать спецификацию операции обработки базы знаний, представленной в виде семантической сети, которая включает условие применения операции, результат выполнения операции, алгоритм, используемые ключевые узлы. Реализовать специфицированную операцию на языке SCP. Разработать методику испытания реализованной операции.

### Варианты индивидуальных заданий

1. Разработать программу проверки двух множеств на равенство и генерации кортежа отношения «быть равными множествами».

2. Разработать программу проверки двух кортежей на равенство и генерации кортежа отношения «быть равными кортежами».

3. Разработать программу склейки двух графовых структур по совпадению их внешних идентификаторов. В результате работы программы создается кортеж отношения «синонимичные термы», в котором объединяются два множества идентификаторов, принадлежащих соответствующим графовым структурам.

4. Разработать программу склейки двух равных кортежей, имеющих одинаковые элементы, входящие в кортежи под одними и теми же атрибутами, и связанных отношением «быть равными кортежами».

5. Разработать программу склейки двух равных множеств, имеющих одинаковые элементы и связанных отношением «быть равными множествами».

6. Разработать программу склейки двух равных прямых, если они проходят через одну общую точку и существует третья прямая, которая параллельна этим двум.

7. Разработать программу склейки двух точек, если они принадлежат двум различным прямым.

8. Разработать программу проверки расположения граничной точки отрезка по одну сторону относительно всех других точек отрезка.

9. Разработать программу проверки склейки двух различных прямых, проходящих через общую точку и параллельных третьей прямой.

## Литература

1. Brickley, D. Resource Description Framework (RDF) Schema Specification. Proposed Recommendation // D. Brickley, R. V. Guha. – World Wide Web Consortium [Электронный ресурс]. – 1999. – Режим доступа: <http://www.w3.org/TR/PR-rdf-schema>.
2. Foundation of intelligent physical agents [Электронный ресурс]. – 2006. – Режим доступа: <http://www.fipa.org>.
3. Gruber, T. R. A Translation Approach to Portable Ontology Specification / T. R. Gruber // Knowledge Acquisition. – 1993. – №5. – P. 199–220.
4. Hendler, J. The DARPA Agent Markup Language / J. Hendler, D. L. McGuinness // IEEE Intelligent Systems. – 2000. – 16(6). – P. 67–73.
5. Java agent development framework [Электронный ресурс]. – 2005. – Режим доступа: <http://jade.cselt.it>
6. Представление и обработка знаний в графодинамических ассоциативных машинах / В. В. Голенков [и др.]; под ред. В. В. Голенкова. – Минск : БГУИР, 2001. – 412 с.
7. Программирование в ассоциативных машинах / В. В. Голенков [и др.]. – Минск : БГУИР, 2001. – 276 с.
8. Интеллектуальные обучающие системы и виртуальные учебные организации / В. В. Голенков [и др.]; под ред. В. В. Голенкова и В. Б. Тарасова. – Минск : БГУИР, 2001. – 488 с.
9. Курбацкий, А. Н. Автоматизация обработки документов / А. Н. Курбацкий. – Минск : БГУ, 1999. – 220 с.
10. Ресурсы IBM для разработчиков [Электронный ресурс]. – 2007. – Режим доступа: <http://www.ibm.com/developerworks/ru/lotus/>.
11. Системы управления документами [Электронный ресурс]. – 2004. – Режим доступа: <http://www.lotusnotes.ru/>.

Учебное издание

**МОДЕЛИ ПРЕДСТАВЛЕНИЯ И ОБРАБОТКИ  
ДАННЫХ И ЗНАНИЙ.  
ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

В 3-х частях

Часть 2

**Лемешева Татьяна Леонидовна  
Гулякина Наталья Анатольевна  
Толкачев Антон Иванович**

Редактор *Е. Н. Батурчик*

Корректор *М. В. Тезина*

Компьютерная верстка и дизайн обложки *Е. Н. Мирошниченко*

Подписано в печать 01.02.2008. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».  
Печать ризографическая. Усл. печ. л. 2,91. Уч.-изд. л. 2,3. Тираж 150 экз. Заказ 217.

Издатель и полиграфическое исполнение: Учреждение образования  
«Белорусский государственный университет информатики и радиоэлектроники»  
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.  
220013, Минск, П. Бровки, 6