№ 1(6) 2012 ISSN 2309-8856

**Шахтинский институт (филиал)
государственного образовательного учреждения
высшего профессионального образования
«Южно-Российский государственный технический
университет
(Новочеркасский политехнический институт)»**

**Южно-Российский государственный университет
экономики и сервиса**

ОБЪЕКТНЫЕ СИСТЕМЫ – 2012

Материалы VI Международной научно-практической конференции

**Россия, Ростов-на-Дону
10-12 мая 2012 г.**

№ 1(6) 2012 ISSN 2309-8856

**Шахтинский институт (филиал)
государственного образовательного учреждения
высшего профессионального образования
«Южно-Российский государственный технический
университет
(Новочеркасский политехнический институт)»**

**Южно-Российский государственный университет
экономики и сервиса**

ОБЪЕКТНЫЕ СИСТЕМЫ – 2012

Материалы VI Международной научно-практической конференции

**Россия, Ростов-на-Дону
10-12 мая 2012 г.**

УДК 004 (05) **№ 1(6) 2012**
ББК 32.97-018 **ISSN 2309-8856**
О-29

Объектные системы – 2012: материалы VI Международной научно-практической конференции (Ростов-на-Дону, 10-12 мая 2012 г.) / Под общ. ред. П.П. Олейника. – Ростов-на-Дону: ШИ ЮРГТУ (НПИ), 2012. – 110 с.

Материалы конференции посвящены принципам проектирования, реализации и сопровождения объектных систем и включают в себя освещение широкого круга проблем. Каждая статья, включённая в сборник, принята к печати на основании положительной рецензии членов организационного и программного комитетов.

Организаторы конференции

Шахтинский институт (филиал) государственного образовательного учреждения высшего профессионального образования «Южно-Российский государственный технический университет (Новочеркасский политехнический институт)»

Южно-Российский государственный университет экономики и сервиса

Информационные партнёры

Сообщество системных аналитиков

Теоретический и прикладной научно-технический журнал "**Информационные технологии**" с ежемесячным приложением

Ежемесячный научно-технический и производственный журнал "**Автоматизация в промышленности**"

Журнал "**Информационные технологии и вычислительные системы**"

Оргкомитет конференции

1. Прокопенко Николай Николаевич, д.т.н., проф., Ректор Южно-Российского государственного университета экономики и сервиса, Россия, Шахты ***(председатель конференции)***
2. Олейник Павел Петрович, к.т.н., Системный архитектор программного обеспечения, Астон, Россия, Ростов-на-Дону ***(сопредседатель конференции)***
3. Божич Владимир Иванович, д.т.н., проф., Кафедра "Информационные системы и радиотехника", Южно-Российский государственный университет экономики и сервиса, Россия, Шахты
4. Сидельников Владимир Иванович, д.т.н., проф., Заведующий кафедрой "Экономика и прикладная математика", Педагогический Институт Южного Федерального университета, Россия, Ростов-на-Дону
5. Черкесова Эльвира Юрьевна, д.э.н., проф., Заведующая кафедрой "Информационные технологии и управление", Шахтинский институт (филиал), Южно-Российский государственный технический университет (Новочеркасский политехнический институт), Россия, Шахты
6. Михайлов Анатолий Александрович, д.т.н., проф., Кафедра "Автоматизированные системы управления", Южно-Российский государственный технический университет (Новочеркасский политехнический институт), Россия, Новочеркасск
7. Кравчик Владислав Георгиевич, к.т.н., доц., Кафедра "Энергетика и БЖД", Южно-Российский государственный университет экономики и сервиса, Россия, Шахты

Международный программный комитет конференции

1. Euclid Keramopoulos, Ph.D., Lecturer, Alexander Technological Educational Institute of Thessaloniki, Греция, Салоники
2. Piotr Habela, Ph.D., Assistant Professor, Polish-Japanese Institute of Information Technology, Польша, Варшава
3. Erki Eessaar, Ph.D., Associate Professor, Acting Head of the Chair, Faculty of Information Technologies: Department of Informatics, Tallinn University of Technology, Эстония, Таллин
4. German Viscuso, MSc in Computer Science, Marketing, Versant Corp., Испания, Мадрид
5. Кузнецов Сергей Дмитриевич, д.т.н., проф., Факультет вычислительной математики и кибернетики, МГУ имени М. В. Ломоносова, Главный научный сотрудник Института системного программирования РАН, член ACM, ACM SIGMOD и IEEE Computer Society, Россия, Москва
6. Шалыто Анатолий Абрамович, д.т.н., проф., лауреат премии Правительства РФ в области образования, Заведующий кафедрой "Технологии программирования", Санкт-Петербургский государственный университет информационных технологий механики и оптики, Россия, Санкт-Петербург
7. Кирютенко Александр Юрьевич, к.ф.-м.н., Директор по ИТ, Астон, Россия, Ростов-на-Дону
8. Галиаскаров Эдуард Геннадьевич, к.х.н, доц., Ивановский государственный химико-технологический университет, Россия, Иваново
9. Чекирис Александр Владимирович, Начальник отдела технического проектирования и НСИ, НИИЭВМсервис, Беларусь, Минск
10. Векленко Ирина Юрьевна, к.э.н., Системный аналитик, Россия, Черноголовка
11. Малышко Виктор Васильевич, к.ф.-м.н., доц., Факультет вычислительной математики и кибернетики, МГУ имени М. В. Ломоносова, Россия, Москва
12. Жилякова Людмила Юрьевна, к.ф.-м.н., доц., с.н.с., Институт проблем управления им. В.А. Трапезникова РАН, Россия, Москва
13. Шахгельдян Карина Иосифовна, д.т.н., Начальник информационно-технического обеспечения, Владивостокский государственный университет экономики и сервиса, Россия, Владивосток
14. Добряк Павел Вадимович, к.т.н., доц., Уральский государственный технический университет, Россия, Екатеринбург
15. Байкин Александр Сергеевич, Ведущий системный аналитик, Автомир, Россия, Москва
16. Аверин Алексей Иванович, Системный аналитик, Астон, Россия, Ростов-на-Дону
17. Лаптев Валерий Викторович, к.т.н., доц., Кафедра "Автоматизированные системы обработки информации и управления", Астраханский государственный технический университет, Россия, Астрахань
18. Ермаков Илья Евгеньевич, Заместитель директора, казенное учреждение Орловской области "Региональный центр оценки качества образования", Технический директор, НПО "Тесла", Россия, Орёл
19. Иванов Денис Юрьевич, Консультант, Ай Ти Консалтинг, Россия, Санкт-Петербург

Рецензенты

Божич Владимир Иванович, Михайлов Анатолий Александрович, Шалыто Анатолий Абрамович, Векленко Ирина Юрьевна, Малышко Виктор Васильевич, Добряк Павел Вадимович

Редакторы

Олейник Павел Петрович (***главный редактор***), Лаптев Валерий Викторович, Галиаскаров Эдуард Геннадьевич, Ермаков Илья Евгеньевич

ISBN 978-5-9903782-1-6
Статьи сборников конференции индексируются в РИНЦ
http://elibrary.ru/title_about.asp?id=48777

№ 1(6) 2012 ISSN 2309-8856

Shakhty Institute (Branch) of
South Russian State Technical University
(Novocherkassk Polytechnic Institute)

South Russian State University of
Economics and Service

Object Systems – 2012

Proceedings of the Sixth International Theoretical and Practical Conference

Edited by Pavel P. Oleynik

Rostov-on-Don, Russia
10-12 May 2012

№ 1(6) 2012 **ISSN 2309-8856**

Object Systems – 2012: Proceedings of the Sixth International Theoretical and Practical Conference. Rostov-on-Don, Russia, 10-12 May, 2012. Edited by Pavel P. Oleynik

The proceedings consist of papers which cover the topics of design work, implementation and maintenance of object systems, considering a broad range of problems. These papers were accepted to publish on the basis of the organization and program committee members reviews.

Conference Organizers

Shakhty Institute (Branch) of South Russian State Technical University (Novocherkassk Polytechnic Institute)

South Russian State University of Economics and Service

Information Partners

Community of System Analysts

Journal "**Information Technologies**" with monthly supplement

Journal "**AUTOMATION IN INDUSTRY**"

Journal "**Information technologies and computer systems**"

ISBN 978-5-9903782-2-3

Содержание / Contents

---

[1] Лауреат номинации "**Лучший доклад по UML-моделированию**". Авторы доклада награждаются книгой Иванова Д.Ю. и Новикова Ф.А. "**Моделирование на UML. Теория, практика, видеокурс**" (www.umlmanual.ru) с автографами авторов

[2] Статья рекомендована к опубликованию в журнале "**Информационные технологии**"

[3] Статья рекомендована к опубликованию в журнале "**Информационные технологии**"

[4] Статья рекомендована к опубликованию в журнале "**Информационные технологии**"

[5] Лауреат номинации "**Лучший доклад о методах преподавания объектных технологий в ВУЗе**". Автор доклада награждается правом бесплатной публикации одного доклада по данной тематике на следующей конференции

Содержание / Contents

УДК 004.4'2, 519.683.2

ГЕНЕРАЦИЯ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПО ОБЪЕКТНЫМ МОДЕЛЯМ[1]

Манжосов Андрей Николаевич, аспирант, Московский Государственный Университет имени М.В.Ломоносова, Россия, Москва, amanzhosov@gmail.com

При помощи языка UML [1] есть возможность создавать подробные и точные объектные модели, на элементы которых можно накладывать строгие формальные ограничения. В составе модели в виде выражений языка OCL [2] могут быть специфицированы инварианты классов, пред- и постусловия операций, тела операций-запросов, начальные значения, правила вывода производных элементов модели и т. д.

Современные инструменты объектного моделирования поддерживают OCL, но возможности этих инструментов в плане исследования свойств объектных моделей невелики. Практика использования программных инструментов верификации и валидации UML-моделей на сегодняшний день распространена нешироко. Традиционные среды верификации и валидации, такие как SPIN, PVS, Frama-C [3, 4], применяемые достаточно часто, базируются на текстовых спецификациях и не могут напрямую исследовать UML-модели.

В такой ситуации можно применить следующий способ: по объектной модели сгенерировать текстовую спецификацию, а затем исследовать свойства модели по полученной спецификации. Схема реализации такого способа изображена на рисунке 1. Согласно схеме, построенная объектная модель подается на вход генератору, который создает по ней спецификацию. Далее полученная спецификация исследуется при помощи средства верификации и валидации.

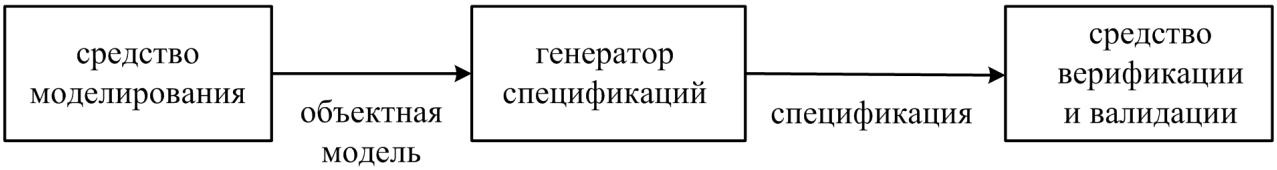


Рис. 1 – Схема исследования свойств модели с помощью генерации спецификаций

Для реализации способа на практике требуется создать средство генерации спецификаций, соответствующих исходным объектным моделям. Во время генерации необходимо обеспечить соответствие между сгенерированной спецификацией и исходной моделью, чтобы результаты проверок и/или вычисленных по спецификации запросов сохраняли силу для исходной модели. При этом спецификация должна иметь такую же степень подробности, что и исходная модель.

Для генерации текстов по объектным моделям концерном OMG разработан стандарт Mof2Text [5]. Стандарт не ориентирован на какой-либо специальный тип генерируемых документов, например, исходных кодов на языках программирования и может использоваться для генерации спецификаций, а также документов на естественном языке. В основе Mof2Text лежит подход, согласно которому генерируемый текст формируется на основе шаблонов. В качестве параметров шаблонов используются элементы исходной модели.

Существует инструментальное средство, базирующееся на стандарте Mof2Text, – Acceleo [6]. Это плагин для Eclipse [7] – среды с большим набором инструментов объектного моделирования. Применяя Acceleo для генерации спецификаций по объектным моделям, созданным в Eclipse Modeling Framework [7], можно реализовать предложенную выше схему. При этом средство моделирования и генератор спецификаций будут иметь общую базу.

[1] Лауреат номинации "**Лучший доклад по UML-моделированию**". Авторы доклада награждаются книгой Иванова Д.Ю. и Новикова Ф.А. "**Моделирование на UML. Теория, практика, видеокурс**" (www.umlmanual.ru) с автографами авторов

Генератор спецификаций будет реализован в виде набора шаблонов Acceleo.

Другим вопросом при реализации описанной ранее схемы является выбор средства верификации и валидации и используемого языка спецификации. Традиционные текстовые языки спецификаций, такие как RSL, VDM-SL, ASCL, Z [4] основаны не на объектном подходе. Составленные с их помощью спецификации сформулированы в терминах, существенно отличающихся от терминов, которые используются в объектных моделях. Генерация спецификаций на традиционных языках осложняется из-за необходимости перехода к принципиально другой терминологии. Использование объектно-ориентированного языка спецификаций существенно облегчает задачу генерации. Примером языка такого рода является язык среды USE (UML-based Specification Environment – среда спецификаций, базирующаяся на UML).

Рассмотрим язык спецификации USE на примере. Предположим, что мы создали модель, описывающую семью. В ней имеется базовый класс *Person*, два производных от него класса *Man* и *Woman* и две ассоциации*: offspring* – между родителями и детьми, и *spouse* – между мужьями и женами. Базовый класс является абстрактным, содержит атрибуты *year* (год рождения), *name*, *surname* и операцию *setYear*, устанавливающую значение соответствующего атрибута. Класс *Woman* содержит операцию *setHusband*, задающую соединение *spouse* между соответствующими объектами. Эта операция имеет предусловие, проверяющее, что участвующие в операции персоны уже не состоят в каком-либо браке, и постусловие, проверяющее, что добавилось соответствующее соединение между объектами. Класс *Man* содержит операцию *setWife*, выполняющую такие же действия, что и *setHusband*, и имеющую аналогичные пред- и постусловие. Также в модели присутствует ограничение на инварианты класса *Person*, контролирующее, что возраст ребенка меньше, чем возраст любого из его родителей. Данная модель проиллюстрирована на следующих диаграммах:

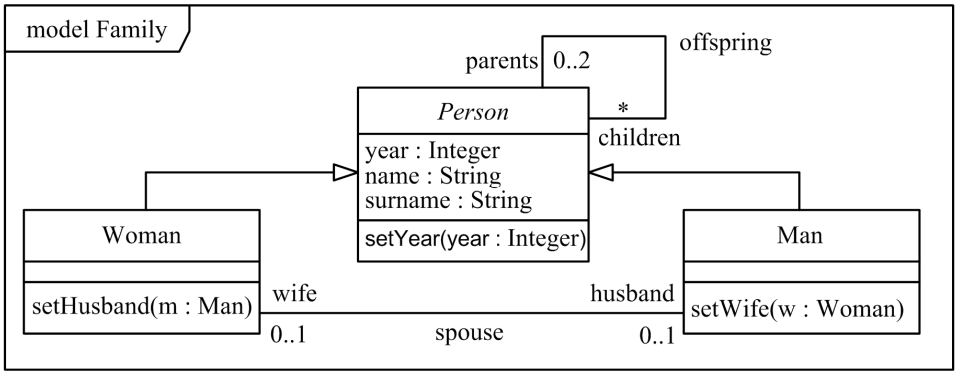


Рис. 2 – Диаграмма классов для описанной модели

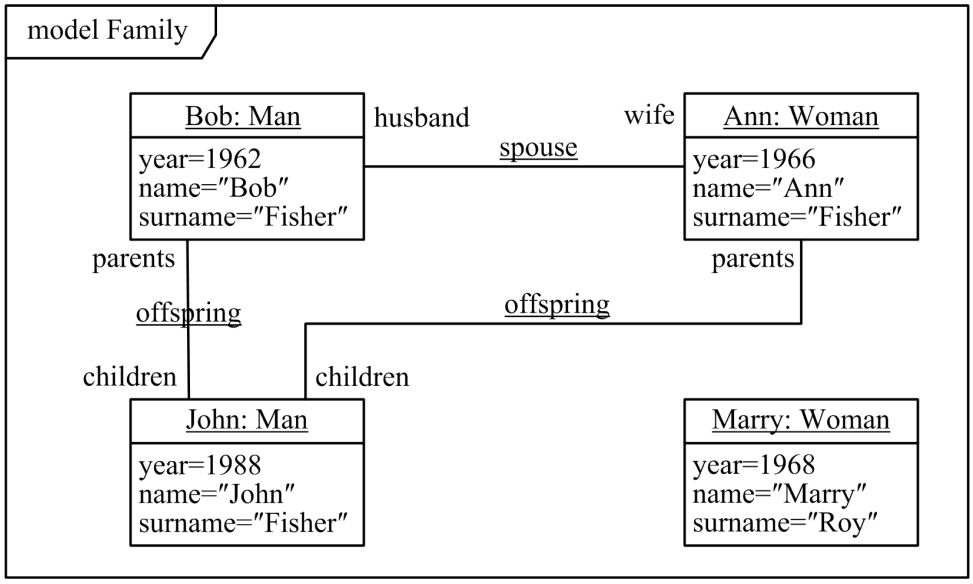


Рис. 3 – Диаграмма объектов для описанной модели

Спецификации USE формулируется в терминах объектной модели, таких как: абстрактные и конкретные классы, классы ассоциаций, бинарные и n-арные ассоциации, наследование. В ней можно описать структуру класса – атрибуты, операции. Например, для класса *Person* описание будет выглядеть следующим образом:

```
abstract class Person
attributes
   year : Integer
   name : String
   surname : String
operations
   setYear (year : Integer)
end
```

Вначале следует имя класса, затем раздел, содержащий его атрибуты, и раздел, описывающий операции класса.

В спецификацию могут входить описания экземпляров классов и соединений между ними. Они могут включать в себя следующие команды:

1. Создание экземпляра класса:
   create <obj_name> : <class_name>
2. Присваивание значения свойству объекта:
   set <obj_name>.<attribute_name> := <attribute_value>
3. Создание соединения между объектами:
   insert(<obj>, <obj>{, <obj>}) **into** <association_name>
4. Создание экземпляра класса ассоциации:
   create <obj> : <ass_class_name> **between** (<obj>, <obj>{, <obj>})

Например, описание объекта *Bob* будет выглядеть так:

```
create Bob : Man
set Bob.year := 1962
set Bob.name := 'Bob'
set Bob.surname := 'Fisher'
```

А описание соединения между *Bob* и *Ann* так:

```
insert(Bob, Ann) into spouse
```

Спецификации могут содержать в себе ограничения на OCL: инварианты классов, пред- и постусловия для операций и др. Ограничение, контролирующее, что возраст ребенка меньше, чем возраст любого из его родителей, будет выглядеть следующим образом:

```
context Person
inv YoungerThanParents: parents->forAll(p | p.year < self.year)
```

Предусловие и постусловие для операции *setWife(w : Woman)* класса *Man* описываются так:

```
pre: w.husband->isEmpty()
post: self.wife->includes(w)
```

Аналогично можно задать предусловие и постусловие для операции *setHusband* класса *Woman*. Можно видеть, что спецификация USE может быть столь же подробной и точной, как UML-модель.

В отличие от традиционных UML-моделей спецификации USE могут быть исполняемыми. В среде USE можно задавать запросы на языке OCL к загруженной спецификации, проверять выполнение инвариантов и других ограничений в составе спецификации. Также можно менять текущее состояние путем изменения экземпляров классов и вызова операций, тела которых можно задавать при помощи языка SOIL[8]. Команды, описанные на SOIL, позволяют изменять значение атрибутов объекта, создавать и удалять объекты и связи между ними. Например, для рассмотренной ранее операции *setWife* мы можем задать тело следующим образом:

```
setWife(w : Woman)
begin
   insert (self, w) into spouse
end
```

Аналогично можно задать тело для операции *setHusband* класса *Woman*.

Рассмотрев возможности USE, можно сделать вывод, что эта среда подходит для реализации предлагаемого способа к исследованию свойств объектных моделей по сгенерированным спецификациям.

Ранее было показано, какие элементы спецификации USE соответствуют каким элементам объектной модели. Для того, чтобы автоматизировать процесс генерации спецификации был создан набор шаблонов Acceleo. Набор составлен из двух групп шаблонов. Первая отвечает за генерацию описания классов и связей между ними. В нее входят 10 шаблонов для генерации спецификаций классов, операций, OCL-ограничений, ассоциаций, классов ассоциаций и перечислимых типов. Вторая группа шаблонов определяет генерацию описаний экземпляров классов и соединений. В нее входят 5 шаблонов для описаний экземпляров классов, экземпляров классов ассоциаций, а также соединений между объектами.

В качестве примера рассмотрим шаблон для описания параметров операции:

```
[template public setInParameters(o : Operation)]
[for(p : Parameter | o.ownedParameter)separator(', ')][p.name/]:
    [p.type.name/][/for]
[/template]
```

Как и любой шаблон Acceleo, этот шаблон имеет название (*setInParameters*) и список параметров (*o*). Во время генерации текста, для каждой операции исходной объектной модели будет вызываться этот шаблон, и вместо параметра *o* будет подставлена конкретная операция модели. Тело шаблона заключается между тегами **template**. Оно включает в себя текст и теги Acceleo, заключенные в квадратные скобки. В данном шаблоне в цикле выбираются параметры операции (за это отвечает элемент модели UML *ownedParameter*). Для каждого из них генерируется текст в виде пары <имя> : <тип>. Эти пары разделяются запятыми (за это отвечает **separator**(', ')). Например, для списка параметров операции *setYear* класса *Person* будет сгенерирован текст: year : Integer.

Шаблон *setInParameters* является вспомогательным, он применяется при работе шаблона *createOperation*, отвечающего за генерации полного описания операции. Рассмотрим этот шаблон:

```
[template public createOperation(o : Operation)]
[o.name/]([o.setInParameters()/])[if (o.type.oclIsUndefined())][else]:
[o.type.name/][/if]
[if (o.bodyCondition->size() > 0)]
begin
[o.bodyCondition.specification.stringValue() /]
end
[/if]
[for (constr: Constraint | o.ownedRule->asSet() - o.bodyCondition-
>asSet())] [constr.specification.stringValue() /]
[/for]
[/template]
```

Вначале генерируется сигнатура операции – ее имя, список параметров (за это отвечает шаблон *getInParameter*) и возвращаемый тип (в случае, если он определен – *oclIsUndefined()* возвращает false*)*. Далее создается тело операции (если оно задано в модели). В конце генерируется набор ограничений (предусловия и постусловия). В качестве примера возьмем операцию *setHusband* класса *Woman*. Для неё будет сгенерировано следующее описание:

```
setHusband(m : Man)
begin
   insert (m, self) into spouse
```

```
end
pre: m.wife->isEmpty()
post: self.husband->includes(m)
```

Остальные шаблоны Acceleo, реализующие генерацию спецификаций, описываются аналогично.

Предположим, что для рассматриваемого выше примера объектной модели сгенерирована спецификация USE. Рассмотрим исследование свойств модели в среде USE. Для проверки операции *setHusband* в среде USE можно ввести команду вызова операции: Marry.setHusband(Bob). Получив команду, среда выдаст сообщение о том, что не выполнено предусловие операции. Это связано с тем, что уже существует соединение *spouse* между *Bob* и *Ann* (*Bob.wife* не пусто).

Для проверки операции *setWife* введем команду вызова операции: Bob.setWife(Marry). Среда выдаст сообщение о том, что не выполнено постусловие операции. Это связано с тем, что теперь *Bob* соединен с двумя объектами (*Ann* и *Marry*), а это противоречит описанию ассоциации *spouse*, так как любой экземпляр класса *Man* может быть соединен не более чем с одним экземпляром класса *Woman* и наоборот.

Проведённое исследование свойств модели показывает, что предусловие для операции *setWife* записано некорректно. На отсутствие соединений следует проверять оба соединяемых объекта, а не только один объект, передаваемый в качестве параметра операции. Корректное предусловие для операции *setWife* класса *Man* выглядит следующим образом: w.husband->isEmpty() and self.wife->isEmpty(). Схожие изменения нужно внести в предусловие операции *setHusband* класса *Woman*. Новое предусловие для него должно выглядеть так: m.wife->isEmpty() and self.husband->isEmpty().

После внесения изменений в модель следует заново сгенерировать спецификацию. Можно убедиться, что исследование вызова операции Bob.marry(Marry) теперь приводит к желаемому результату. Среда USE выдаёт сообщение о том, что для данного вызова операции предусловие ложно. Очевидно, что после внесенных изменений предусловие стало более строгим, мешающим создавать некорректные соединения между объектами. Таким образом, поиск ошибки в модели и её исправление стали возможными благодаря предложенному генератору спецификаций и применению среды USE.

Подведём итоги. Для компенсации недостаточно развитых возможностей инструментов объектного моделирования по исследованию свойств моделей предложен способ, предусматривающий генерацию спецификаций по исходным моделям и дальнейшее их исследование в среде спецификации. Способ реализован при помощи Eclipse, Acceleo и USE. Предложен набор шаблонов Acceleo для генерации спецификаций USE по объектным моделям, созданным средствами EMF UML. На примере рассмотрена генерация спецификации, её исследование, обнаружение ошибки и исправление неточности в исходной модели.

Литература

1. Арлоу Д., Нейштадт И. UML 2 и унифицированный процесс. Практический объектно-ориентированный анализ и проектирование, 2-е издание. СПб.: Символ-Плюс, 2007. 624 с.
2. Warmer J., Kleppe A. The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition. Boston: Addison-Wesley, 2003. 240 p.
3. Holzmann G. The Spin Model Checker: Primer and Reference Manual. Boston: Addison Wesley, 2003. 596 p.
4. Кузьменкова Е. А., Петренко А. К. Формальная спецификация программ на языке RSL: Конспект лекций. М.: Издательский отдел факультета ВМК МГУ, 2001. 107 с.
5. Object Management Group MOF Model to Text Transformation Language, v 1.0 2008 (http://www.omg.org/spec/MOFM2T/1.0/PDF)
6. Begaudeau S. And you thought you knew Template Based Generators? – «Code generation 2011» conference. Cambridge, 2011.

7. Steinberg D., Budinsky F., Paternostro M., Merks E. EMF: Eclipse Modeling Framework Second Edition. Boston: Addison-Wesley, 2008. 744 p.
8. Büttner F., Gogolla M. Modular Embedding of the Object Constraint Language into a Programming Language // 14th Brazilian Symposium on Formal Methods (SBMF'2011). Berlin: Springer, 2011. 124-139 p.

УДК 004.4'22

ПРЕОБРАЗОВАНИЕ ПРОГРАММ НА VISUAL PROLOG В МОДЕЛИ UML[1]

Андрианов Иван Алексеевич, стажер-исследователь, Институт системного программирования РАН, аспирант, Факультет вычислительной математики и кибернетики, МГУ имени М. В. Ломоносова, Россия, Москва, andrianov.sp.cmc@gmail.com

Поддержка и модификация систем, написанных на языках логического программирования, затруднена в связи с недостатком средств программной инженерии логических программ. В то же время существует широкий спектр средств программной инженерии объектно-ориентированных программ. В связи с этим актуальна задача преобразования логической программы в объектную модель. Решение данной задачи позволит получать объектные модели, соответствующие логическим программам. Для полученных объектных моделей станет возможным в полной мере использовать средства объектно-ориентированной программной инженерии, например, модифицировать полученные объектные модели и автоматически генерировать по ним код на одном из объектно-ориентированных языков.

```
domains:                                                          (1)
  department_type = financial, logistics, technical.              (2)
  date = d(integer, integer, integer).                            (3)
  person = p(string, date).                                       (4)
  department = dep(string, department_type).                      (5)
...
predicates:                                                       (6)
  male(?person).                                                  (7)
  age(+person, ?integer).                                         (8)
  works(?person, ?department).                                    (9)
...
clauses:                                                         (10)
  male(p("Andrey", d(21, 02, 1986))).                            (11)
  age(p(_, d(_, _, Year), Age) :- Age is 2012 - Year.            (12)
  works(p("Andrey", d(21, 02, 1986)), dep("ISP RAS", technical)). (13)
  works(p("Elena", d(13, 01, 1987)), dep("ISP RAS", technical)).  (14)
```

Листинг 1 — Фрагмент логической программы, рассматриваемый в качестве примера

Исследования в области анализа и преобразования логических программ активно проводились в 1990-е годы. В то время были разработаны алгоритмы выявления типов аргументов предикатов (например, GAIA [1]) и профилей вызова предикатов (например, MDDAA [2]) программ на стандартном Прологе. Созданные алгоритмы применялись для повышения эффективности интерпретаторов и компиляторов Пролога. В данной статье

[1] Статья рекомендована к опубликованию в журнале "**Информационные технологии**"

логические программы подвергаются анализу с другими целями. Задача состоит в переходе от логических терминов исходной программы к терминам объектной модели. Непосредственно для её решения наработки 1990-х годов не подходят.

Будем рассматривать преобразование программ, написанных на языке Visual Prolog [3]. Особенности Visual Prolog проиллюстрируем на примере.

Программа в листинге 1 представляет собой модельную базу данных о сотрудниках и отделах, в которых они работают. К данной базе можно сделать запрос, например, на получение всех сотрудников заданного отдела мужского пола: «works(P, dep("ISP RAS", technical)), male(P).». Ответом на данный запрос будет «P = p("Andrey", d(21, 02, 1986)).».

Программа разбита на 3 раздела: «domains», «predicates», «clauses». В разделе «domains» объявляются типы программы. Тип может представлять собой набор термов-констант (например, «department_type» в строке 2), функтор с типами аргументов (например, person в строке 4) или списочный тип (имеет вид «тип_элементов*»). В разделе «predicates» объявляются предикаты с типами аргументов. Аргументам предикатов приписаны профили вызова: «+» для входных аргументов, «-» для выходных аргументов и «?» для аргументов, являющихся и входными, и выходными. В разделе «clauses» задаются факты и правила для предикатов.

Для осуществления преобразования программ в модели составлен набор правил преобразования и реализовано программное средство, осуществляющее преобразование в соответствии с этими правилами.

Правила преобразования программ на Visual Prolog в модели UML [4] составлены специальным образом, обеспечивающим сохранение в результирующей модели семантики исходной программы. Это означает, что предусмотрено такое преобразование запросов к программам в запросы к моделям, что ответ на произвольный к входной программе запрос совпадает с ответом на соответствующий ему запрос к результирующей модели.

Правила преобразования сгруппированы в зависимости от их назначения. К первой группе относятся правила преобразования типов, ко второй – правила преобразования предикатов, к третьей – правила преобразования запросов.

Строковый и числовой простой тип следует преобразовывать в соответствующий тип UML.

Каждый простой тип, представляющий собой набор термов-констант, следует преобразовывать в перечислимый тип с тем же набором констант. Например, «department_type» в строке 2 преобразуется в перечислимый тип «DepartmentType».

Каждый тип, представляющий собой множество термов с общим функтором и арностью, следует преобразовывать в класс. При этом каждый аргумент простого типа следует преобразовывать в атрибут класса, а каждый аргумент пользовательского типа — в направленную ассоциацию между соответствующими классами. Например, «person» в строке 4 преобразуется в класс «Person», строковый аргумент в атрибут «arg1», а аргумент типа «date» в направленную ассоциацию между классами «Person» и «Date».

Каждый списочный тип с элементами простого типа следует преобразовывать в класс с атрибутом соответствующего типа мощности «*».

Каждый списочный тип с элементами пользовательского типа следует преобразовывать в класс, связанный с классом-элементом ассоциацией мощностью «*» с обоих концов.

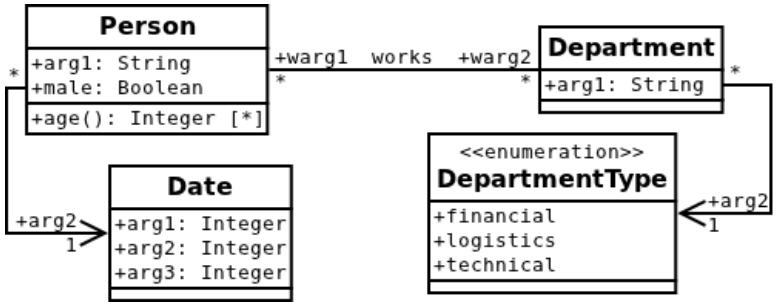


Рис. 1 – Классы модели, построенной по логической программе

Преобразования предикатов определены следующими правилами:

1. Каждый предикат, описываемый фактами без переменных, имеющий не менее двух аргументов пользовательских типов и ни одного аргумента простых типов, следует преобразовывать в ассоциацию мощностью * с обоих концов между классами-аргументами. При этом между объектами, соответствующими термам-аргументам из фактов, описывающих предикат, устанавливается связь. Например, «works» в строке 9 преобразуется в ассоциацию «works» между классами «Person» и «Department». При этом связь установлена между объектами «p1» – «dep1» (строка 13), «p2» – «dep1» (строка 14).
2. Каждый предикат, описываемый фактами без переменных, имеющий один аргумент пользовательского типа и ни одного аргумента простого типа, следует преобразовывать в булев атрибут класса-аргумента. При этом атрибут получает значение «true» для объектов, соответствующих термам-аргументам из фактов, описывающих предикат, и «false» для остальных. Например, «male» в строке 7 преобразуется в булев атрибут «male» класса «Person». Атрибут получает значение «true» для объекта «p1» (строка 11) и «false» для объекта «p2».
3. Каждый предикат, описываемый хотя бы одним правилом, имеющий по одному аргументу пользовательского и простого типа, следует преобразовывать в операцию класса-аргумента, возвращающую список элементов соответствующего простого типа. При этом тело операции на языке OCL [5] следует генерировать по аналогии с генерацией соответствующего запроса для правых частей правил (см. преобразование запросов). Например, «age» в строке 8 преобразуется в операцию «age» класса «Person» с OCL-описанием «context Person::age(): Bag(Integer) body: Bag {2012 – self.arg2.arg3}» (строка 12).

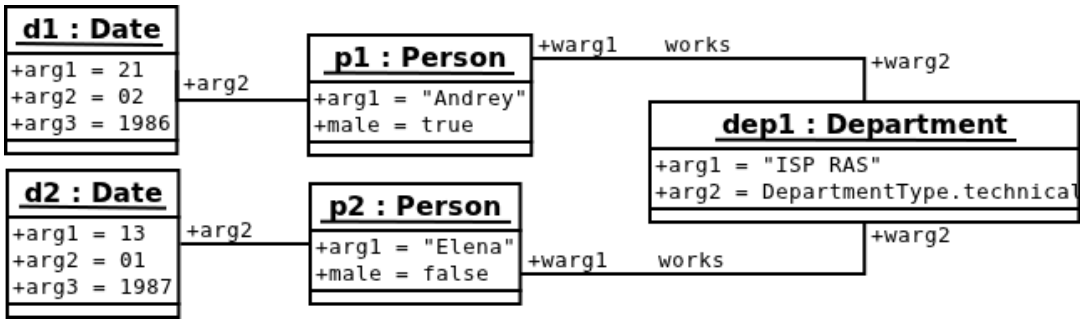


Рис. 2 – Объекты модели, построенной по логической программе

Модельная программа не охватывает все возможные ситуации. Перечислим правила преобразования предикатов в остальных случаях:

1. Каждый предикат, описываемый фактами без переменных, имеющий не менее двух аргументов пользовательских типов и хотя бы один аргумент простого типа, следует преобразовывать в класс ассоциации мощностью «*» с обоих концов между классами-аргументами. Каждый аргумент простого типа следует преобразовывать в атрибут класса ассоциации с соответствующим типом. Между объектами, соответствующими термам-аргументам из фактов, описывающих предикат, устанавливается связь.
2. Каждый предикат, описываемый фактами без переменных, имеющий по одному аргументу пользовательского и простого типа, следует преобразовывать в атрибут класса-аргумента соответствующего простого типа мощности «*».
3. Каждый предикат, описываемый фактами без переменных, имеющий один аргумент пользовательского типа и 2 или более аргументов простых типов, следует преобразовывать в ассоциацию мощностью * с обоих концов между классом-аргументом и вспомогательным классом. Каждый аргумент простого типа следует преобразовывать в атрибут данного класса, имеющий соответствующий тип. Между объектами, соответствующими термам-аргументам из фактов, описывающих предикат, устанавливается связь.

4. Каждый предикат, описываемый фактами без переменных, не имеющий ни одного аргумента пользовательского типа и произвольное количество аргументов простых типов, следует преобразовывать в класс. При этом каждый аргумент следует преобразовывать в атрибут класса, имеющий соответствующий тип.
5. Каждый предикат, описываемый хотя бы одним правилом, имеющий один аргумент пользовательского типа и ни одного аргумента простого типа, следует преобразовывать в булев метод класса-аргумента. Тело метода на языке OCL следует генерировать по аналогии с генерацией соответствующего запроса для правых частей правил (см. преобразование запросов).
6. Каждый предикат прочего вида следует преобразовывать в операцию. Типом возвращаемого значения операции будет булев тип в случае, если у предиката имеются только входные аргументы, и мультимножество кортежей, элементы которых имеют типы, соответствующие типам выходных и входных-выходных аргументов предикатов, в противном случае. В случае, если у предиката имеется хотя бы один входной аргумент пользовательского типа, операция добавляется в класс, соответствующий типу одного из таких аргументов. При этом соответствующий аргумент удаляется из операции (становится неявным аргументом self). В случае, если у предиката имеется хотя бы один выходной или входной-выходной аргумент пользовательского типа, операция добавляется как статическая в класс, соответствующий типу одного из таких аргументов. Если у предиката нет аргументов пользовательских типов, метод добавляется как статический в служебный класс «Predicates». Недетерминированность выбора класса разрешается с помощью метрики «класс с наименьшим числом операций». Тело метода на языке OCL следует генерировать по аналогии с генерацией соответствующего запроса для правых частей правил (см. преобразование запросов).

Перейдем к преобразованию запросов. Сначала рассмотрим пример преобразования. Вернемся к рассмотренному ранее запросу на получение всех сотрудников заданного отдела мужского пола «works(P, dep(“ISP RAS”, technical)), male(P).». Напомним, что ответом на данный запрос будет «P = p(“Andrey”, d(21, 02, 1986)).». Данному запросу будет соответствовать запрос из листинга 2.

Рассмотрим структуру данного запроса к модели. В нем осуществляется итерация по всем экземплярам ассоциации «works» и класса «Person». Выбор данных коллекций обусловлен предикатами, входящими в исходный запрос. В результирующее мультимножество добавляются те и только те объекты класса «Person», которые входят в качестве первого аргумента в экземпляр «works», вторым аргументом которого является объект «dep1» класса «Department», и имеют «true» в качестве значения атрибута «male». Выбор таких условий отбора обусловлен константой, переменной и вторым предикатом запроса. Ответом на запрос будет являться «Bag {Tuple {P = @p1}}». Нетрудно заметить, что ответы на запрос к программе и к модели совпадают с точностью до различий в терминологии логических программ и объектных моделей. Действительно, объект «p1» класса «Person» соответствует терму «p(“Andrey”, d(21, 02, 1986))».

```
works::allInstances()->iterate(e1: works; res1: Bag(Tuple(P: Person)) = Bag {} |
Person::allInstances()->iterate(e2: Person; res2: Bag(Tuple(P: Person)) = res1 |
    if e1.warg2 = dep1 and e1.warg1 = e2 and e2.male
      then res2->including(Tuple {P = e2})
      else res2
```

Листинг 2 – Запрос к полученной модели

Заметим, что представляется возможным построить более простой, но эквивалентный рассмотренному запрос. Он имеет следующий вид «dep1.warg1->select(male)». Однако целью являлось создание универсального алгоритма преобразования запросов, работающего для произвольного запроса и обладающего свойством равенства ответов и, как следствие, подтверждающего, что рассмотренное преобразование программ в модели сохраняет семантику. На данном этапе исследования преобразования программ на Visual Prolog в

модели UML вопросы сложности и скорости вычисления запроса не рассматривались. Однако в ходе дальнейших исследований в этой области на них следует обратить внимание.

Рассмотрим алгоритм преобразования запросов. Вычисление ответа на запрос к логической программе осуществляется посредством полного перебора с возвратами. В связи с этим соответствующий запрос к модели можно строить в виде итерации по набору коллекций с последующим отбором элементов. Для каждого предиката следует по очереди рассматривать предикаты, входящие в исходный запрос, и в зависимости от того, по какому правилу они преобразованы, добавлять к набору коллекций перебора и условию отбора новые элементы.

Если предикат «p» преобразован в класс, класс ассоциации, ассоциацию, то в набор добавляется коллекция «p::allInstances()». В условие отбора добавляется равенство атрибутов элемента коллекции константам или элементу, соответствующему той же переменной.

Если предикат «p» преобразован в булев атрибут или операцию без аргументов (предикат с единственным аргументом пользовательского типа, преобразованного в класс «T»), аргумент содержит переменные, то в набор добавляется коллекция «T::allInstances()». В условие отбора добавляется значение булева атрибута или операции элемента коллекции. В случае, если аргумент переменных не содержит, коллекция остается без изменений. В условие отбора добавляется значение булева атрибута или операции объекта, соответствующего аргументу предиката.

Если предикат «p» преобразован в атрибут (операцию) простого типа мощностью * (предикат с двумя аргументами: одним пользовательского типа, преобразованного в класс «T», другим простого типа), аргумент пользовательского типа содержит переменные, то в набор добавляются коллекции «T::allInstances()» и «p» («p()»). Если же аргумент пользовательского типа переменных не содержит, в набор добавляются коллекции «o.p» («o.p()»), где «o» – объект, соответствующий терму-пользовательскому аргументу. В обоих случаях в условие отбора добавляется равенство элемента коллекции константе или элементу, соответствующему той же переменной.

В остальных случаях, если у предиката «p» имеется хотя бы один выходной или входной-выходной аргумент, то в набор добавляется коллекция «o_1.p(o_2, .., o_n)» или «T::p(o_1, .., o_n)», где «o_1», …, «o_n» – термы, соответствующие входным аргументам предиката, а «T» – класс, в который добавлена операция, соответствующая «p». В условие отбора добавляется равенство атрибутов элемента коллекции константам или элементу, соответствующему той же переменной. Если же у предиката имеются лишь входные аргументы (следовательно, построенная по нему операция возвращает значение булева типа), список остается без изменений. В условие отбора добавляется вызов операции с аргументами, соответствующими аргументам предиката.

Разработанное программное средство, осуществляющее преобразование программ на языке Visual Prolog в модели UML, основывается на сформулированных выше правилах.

В качестве средства для создания анализатора входной программы был использован ANTLR [6]. Это средство принимает на вход LL(k) грамматики, а на выходе позволяет получить исходный код анализатора на одном из языков программирования, в частности на языке Java. В ходе разработки была создана LL(1) грамматика языка Visual Prolog, включающая в себя также описание структуры дерева программы. По разработанной грамматике с помощью ANTLR был получен исходный код синтаксического анализатора на языке Java.

ANTLR позволяет также описать так называемую грамматику дерева, в которой для каждой части полученного дерева можно задать код, который будет выполняться. Такой код, как правило, создает объекты классов, с помощью которых описывается внутреннее представление программы, и налаживает связи между объектами этих классов. В ходе разработки был разработан набор классов внутреннего представления программ на языке Visual Prolog, а также создана грамматика дерева, использующая данные классы. По данной

грамматике с помощью ANTLR был получен исходный код семантического анализатора на языке Java.

В качестве средства для генерации моделей UML использовалась система Eclipse MDT [7]. Это средство содержит одну из наиболее полных реализаций метамодели UML 2 и является дополнением к известной среде разработки Eclipse. В ходе разработки был реализован класс, анализирующий полученное внутреннее представление программы. Сначала анализируется информация о типах, затем о предикатах. В случае нахождения соответствия внутреннего представления одному из правил преобразования данный класс обращается к Eclipse MDT для записи соответствующей части модели.

Рис. 3 – Компоненты средства и взаимодействие между ними

Итак, взаимодействие между компонентами средства можно представить схемой, показанной на рисунке 3. Необходимо отметить, что такая схема позволяет без особого труда изменять правила построения моделей в ходе эксплуатации средства, т.к. от них зависит лишь соответствующий компонент-преобразователь.

Результатами исследования являются составленные правила преобразования программ на языке Visual Prolog в модели UML, которые обеспечивают сохранение семантики исходной программы, и разработанное программное средство, которое осуществляет преобразование по этим правилам. Набор правил построения моделей включает в себя правила преобразования запросов к исходной программе в запросы к построенной модели такие, что выполняется условие равенства ответов. Полученные результаты могут быть использованы для реинженерии программ, написанных на объектно-ориентированных диалектах языка Пролог.

Литература

1. Van Hentenryck P., Cortesi A., Le Charlier B. Type analysis of Prolog using type graphs // Journal of logic programming. 1993. N 22. P. 179-209.
2. Debray S. K. Static inference of modes and data dependencies in logic programs // ACM transactions on programming languages and systems. 1989. N 11. P. 418-450.
3. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. СПб.: БХВ-Петербург, 2003. 992 с.
4. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. 2е изд. М.: ДМК Пресс, 2007. 496 с.
5. Warmer J., Kleppe A. Object Constraint Language, The: Getting Your Models Ready For MDA, Second Edition. Addison Wesley, 2003. 240 p.
6. Parr T. The definitive ANTLR Reference. Raleigh, USA: The Pragmatic Bookshelf, 2007. 369 p.
7. Gronback R. Eclipse Modeling Project: A Domain-Specific Language Toolkit. Addison Wesley, 2009. 734 p.

УДК 681.3

ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ОНТОЛОГИИ НАВИГАЦИОННОЙ СИСТЕМЫ САЙТА

Грегер Сергей Эдуардович, доцент, Уральский федеральный университет имени первого Президента России Б.Н.Ельцина, Нижнетагильский технологический институт (филиал), Россия, Нижний Тагил, segreger@gmail.com

Использование технологий управления знаниями становится в последние годы все более востребовано при разработке сложных программных систем. Не исключением являются и веб-системы, такие как, например, порталы. [1, 2] Навигационная модель является одним из важнейших аспектов, характеризующих портал [3]. Навигационная модель задает структуры для представления реальных объектов и связей между ними. Целью построения модели является стремление выявить связи между информационными объектами для решения конкретных задач пользователя.

Данные на портале представлены как множество взаимосвязанных информационных объектов (ИО). В общем случае каждый ИО соответствует некоторому понятию онтологии (является его экземпляром) и имеет заданную онтологией структуру. Между конкретными ИО существуют связи, семантика которых определяется отношениями, заданными между соответствующими понятиями онтологии. Совокупность таких ИО и их связей образует информационное содержание (контент) портала. Одновременно с такими связями между конкретными ИО могут существовать дополнительные связи, семантика которых определяется отношениями, определяемыми при анализе требований пользователя к проектируемой системе и обеспечивающими удобную навигацию между ИО портала. Набор таких связей совместно с дополнительно введенными концептами, позволяющими строить эффективную навигацию по порталу, образуют навигационную модель, представленную онтологией навигации [4].

Вся информация о конкретном объекте и его связях отображается в виде HTML-страницы, формат и наполнение которой зависят от класса данного объекта и заданного для него шаблона визуализации. При этом объекты, связанные с данным объектом, представляются на его странице в виде гиперссылок, по которым можно перейти к их детальному описанию. Для больших списков формируется составная страница, включающая список страниц с элементами навигации по этому списку.

Навигация по данным портала представляет собой процесс перехода от одних информационных объектов к другим по заданным между ними связям. Например, при просмотре информации о конкретном проекте мы можем видеть значения его атрибутов и его связи с другими объектами. Используя представленные связи в качестве элементов навигации, можно перейти к просмотру подробной информации, как по прямым связям, так и по обратным

Существуют различные типы навигации, которые разработчик может использовать при построении навигационной модели. Каждый из этих типов обеспечивается специальными визуальными компонентами, размещаемыми на страницах портала, например такими, как внедренные ссылки, навигационные панели, закладки, поисковые формы, карты сайтов и др. [3].

Глобальная навигация портала предоставляет ссылки на основные части портала. Эти ссылки доступны с любой страницы портала и позволяют пользователю получить доступ к группам ресурсов, организованным на основе достаточно общих критериев [5].

Локальная навигация позволяет пользователю перемещаться между ресурсами внутри групп, определяемых глобальной навигацией [6].

Контекстуальная навигация позволяет пользователю перемещаться на ресурсы, связанные с текущим ресурсом единым контекстом. Примером такой навигации являются ссылки на сопутствующие товары, предлагаемые сайтами интернет-магазинов.

Построение навигационной структуры проводится в определенной последовательности:

1. На основании анализа задач пользователя строится навигационная структура (онтология) виртуальных сервисов, каждый из которых определяет запрос к онтологии или трансформацию такого запроса.
2. Множество виртуальных сервисов классифицируется по одинаковым типам запросов к онтологии, по трансформациям запросов, по способам представления запросов.

3. С каждым запросом связывается какой-либо контент-тип. Если такая связь возможна, то узел навигационной структуры отмечается принадлежностью к этому типу, если нет — отмечается как виртуальный сервис.
4. Навигационная структура преобразуется в иерархическое дерево компонентов и сервисов.
5. Производится генерация портала.
6. После выявления набора виртуальных сервисов, не связанных с компонентами системы, принимается решение об их реализации каждого либо в виде специального компонента с хранением его экземпляров в базе данных, либо в виде программного компонента, выполняющего запрос к системе хранения данных и отображающего результат запроса в некотором шаблоне представления.
7. Для всех компонентов принимается решение о разработке дополнительных шаблонов представления (видов), соответствующим требованиям представления пользовательской задачи.

Для обеспечения такого процесса необходима спецификация модели и соответствующие инструментальные средства. Формализация навигационной модели обеспечена специально созданной онтологией навигации, схема которой представлена на рисунке 1.

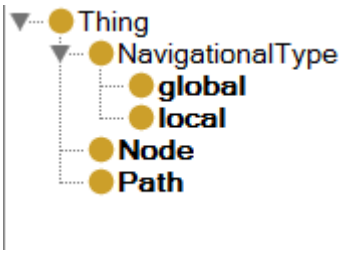


Рис. 1 – Схема онтологии навигации

В представленной онтологии концепция «Узел» (Node) представляет информационные объекты, которые размещены на портале. Для каждого узла с помощью соответствующих связей определены следующие атрибуты:

- Unique ID — уникальный идентификатор узла;
- Title — наименование узла;
- AssociationWithType — определяет тип информационного объекта, с которым связан узел;
- AssociationWithTask — определяет связь узла с пользовательскими задачами, определенными в онтологии пользовательских задач.

Связи между узлами, устанавливающими навигационную схему, представлены в онтологии концепцией Path. Каждый индивидуал этого класса определяет возможность навигационного переход с одного узла на другой, а также определяет тип узла (local or global) по наличию соответствующей связи с концепцией NavigationalType.

Особое внимание при разработке онтологии навигации следует уделить навигации между индивидуалами онтологии, поскольку их значения являются основным источником знаний для пользователей. С этой точки зрения важно поддерживать не только навигацию через иерархию наследования, определяемую отношениями is-A, subClassOf и т.п., но и другими видами отношений.

Учитывая факт, что разработчики веб-приложений часто не имеют достаточной подготовки в области управления знаниями, актуальной является задача создания инструментальных средств, позволяющих заполнять навигационную онтологию, используя семантику, характерную для таких пользователей.

Редактор навигационной модели предназначен для заполнения базы данных онтологии навигации путем создания объектов онтологических классов, входящих в ее базу знаний. Результатом работы редактора является навигационная модель, дающая представление о структуре навигационного графа, о связи узлов графа с задачами пользователя и с

информационными объектами, представляющими навигационные узлы. Выделим задачи, решаемые с помощью редактора:

- создание объектов, представляющих в модели понятие «узел»;
- редактирование свойств узлов навигационной модели;
- определение связей между узлами с возможностью спецификации вида связи — связь «целое-часть» или связь типа «внешняя ссылка»
- визуальное отображение построенной навигационной структуры;
- возможность связывания узла с задачами пользователя, решение которых связано с посещением данного узла;
- возможность связывания узла с информационным объектом, поставляющим данные для отображения содержимого узла.

Дополнительным требованием является реализация редактора в виде веб-приложения, позволяющего производить совместную разработку навигационной модели.

Учет специфической семантики пользователя производится предоставлением ему специальных компонент и операций над ними, отражающими привычные ему понятия. Первоначально реализации редактора навигационной модели основывалась на использовании объектной модели, представленной на рисунке 2.

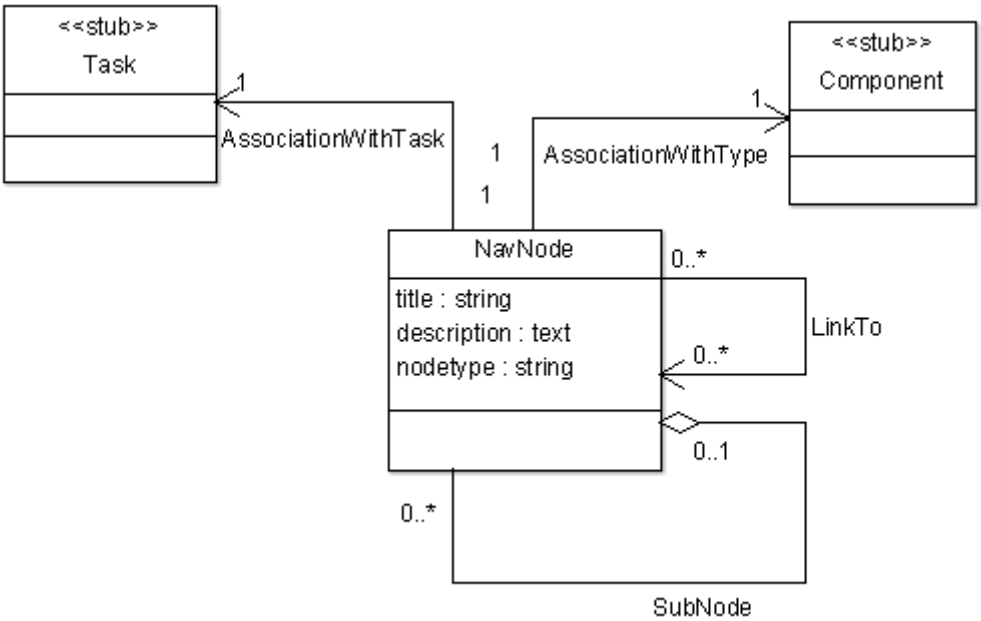


Рис. 2 – Объектная модель реализации редактора

Класс «NavNode» предназначен для представления в программе концепта онтологии «Node». Для определения глобального или локального типа узла используется свойство nodetype.

Классы «Task» и «Component» являются внешними классами по отношению к рассматриваемой объектной модели и предназначены для представления пользовательских задач и информационных объектов соответственно.

Свойство SubNode позволяет строить иерархическую структуру узлов, обычную для структуры сайтов, а свойство LinkTo — контекстуальную навигацию.

Использование таких объектов позволяет строить онтологию как объектно-ориентированную семантическую сеть, которая затем может быть трансформирована в описание на языке OWL.

Системой реализации редактора выбрана система управления содержимым CMS Plone, позволяющая реализовать требование об обеспечении совместной разработки навигационной модели, и предоставляющая возможность хранения онтологии в виде объектно-ориентированной сети в объектной базе данных.

Хранение разработанной онтологии обеспечивается специальными адаптерами хранения, определяющими способ хранения — объектная или реляционная база данных, текстовый файл в формате OWL. Хотя создание индивидуалов обеспечивается соответствующими компонентами, по умолчанию хранимыми в объектной базе, такая возможность необходима для интеграции с внешними программными системами.

В режиме просмотра навигационной модели отображаются объекты навигационной модели выбранного уровня вложенности, в данном случае — все узлы, имеющие статус глобальных. Левая часть окна редактора отображает дерево вложенности узлов навигационной модели, формируемое по связи «SubNode». В центральной части редактора отображается окно просмотра свойств текущего узла.

При переходе на другой уровень в окне просмотра представлен список узлов, связанных с текущим узлом связью «SubNode», список ссылок на внешние узлы, а также представлены ссылки на объекты, представляющие связанные с данным узлом задачу пользователя и на тип объекта, реализующего текущий узел (рис. 3).

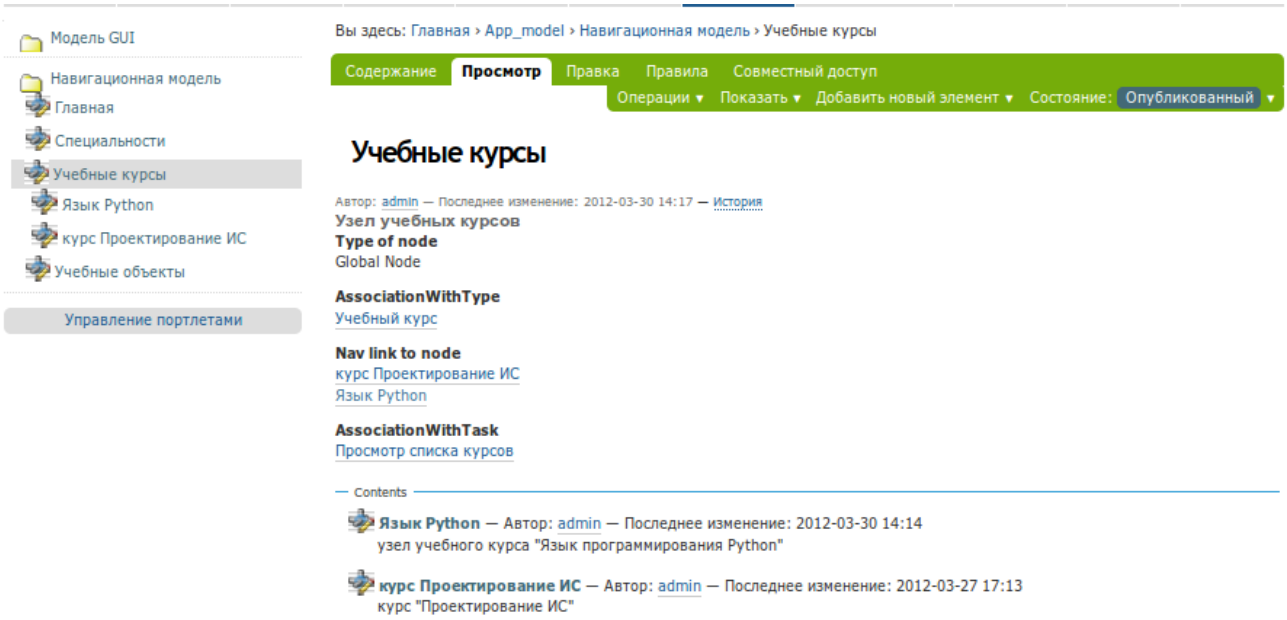


Рис. 3 – Окно просмотра навигационной модели

При выборе режима «правка» в меню редактора происходит переход в режим редактирования текущего узла с возможностью изменения свойств узла и его связей. Редактирование иерархического дерева производится посредством создания или удаления объектов внутри объекта, представляющего родительский узел. При выборе любой ссылки в окне редактора происходит переход в окно управления соответствующим объектом, где доступны режимы просмотра и редактирования объекта.

Представленный редактор функционирует во взаимодействии со специальными редакторами онтологий, такими как редактор онтологии информационных компонент, редактор онтологии пользовательского интерфейса, редакторы онтологии задач и т.п. Все эти редакторы используются в составе онтологического портала дистанционного обучения [7], и их объединяет единый способ хранения онтологий, что позволяет им функционировать в рамках единой онтологической модели сайта, выполняя специфические задачи. Преобразование в формат OWL производится специальным сервисом, общим для всех редакторов. Еще одним общим сервисом является сервис генерации классов, навигации и визуальных компонент. Рассмотрение этих сервисов выходит за рамки обсуждаемой работы.

Разработанный редактор позволяет заполнять онтологию навигации в режиме моделирования, представлять ее с использованием языка OWL и генерировать структуру сайта при выполнении некоторых дополнительных условий.

Литература

1. Загорулько Ю.А. Технология разработки порталов научных знаний // Программные продукты и системы, № 4, 2009.
2. Грегер С.Э., Сковородин Е.Ю. Построение онтологического портала с использованием объектной базы // Объектные системы – 2010: Материалы I Международной научно-практической конференции. Россия, Ростов-на-Дону, 10-12 мая 2010 г / под общ. ред. П.П. Олейника. – Ростов-на-Дону, 2010. – с. 74-78.
3. Doss, G. (2002a). Designing Effective Web Navigation. Retrieved on June 20, 2006 from url:http://www.gdoss.com/web_info/web_navigation.pdf
4. Creative Commons. Navigation models. Retrieved on June 20, 2006 from url:http://www.webdesignfromscratch.com/navigation_models.cfm
5. Timberlake, S. The basics of navigation. Retrieved on June 15, 2006 from <http://www.efuse.com/Design/navigation.html>.
6. Doss, G. (2002a). Designing Effective Web Navigation. Retrieved on June 20, 2006 from <http://www.gdoss.com/web_info/web_navigation.pdf>
7. Грегер С.Э. Реализация задач дистанционного обучения средствами CMS Plone. Актуальные вопросы использования инновационных технологий в образовательном процессе: Материал всероссийской научно-практической конференции, Нижний Тагил, Россия, 2010г. / НТГСПА — Нижний Тагил, 2010. – с. 166-169.

УДК 681.3.07

ПРИМЕНЕНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ C# И ПЛАТФОРМЫ .NET 4.0 ДЛЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО МОДЕЛИРОВАНИЯ

Сарсимбаева Сауле Мусаевна, к.ф.-м.н., доцент, заведующая кафедрой информатики и ВТ, Актюбинский государственный университет имени К. Жубанова, Республика Казахстан, Актобе, sarsi@mail.ru

Саймагамбетова Азиза Жанболатовна, студент, Актюбинский государственный университет имени К.Жубанова, Республика Казахстан, Актобе, sunrise-japan@list.ru

На сегодняшний день объектно-ориентированное программирование достигло в своем развитии того уровня, когда разработчики смогли увидеть потенциальные возможности этой технологии. Многие компании, предприятия, имеют возможность организовывать свою деятельность, используя инструменты компьютерного моделирования, тем самым повышая свою производительность. Главные достоинства объектно-ориентированного подхода позволяют создавать проекты и программы с гибким кодом, с возможностью модификации, усовершенствования и повторного использования готового кода. Так как этот подход сложился на основе многолетней практики, он вобрал в себя все лучшие достижения в технологии программирования и именно он является одним из интенсивно развивающихся направлений теоретического и прикладного программирования. Мы являемся свидетелями бурного роста применения объектно-ориентированных языков в производстве программного обеспечения за последние десять лет. Компания Microsoft предлагает наиболее развитое и комплексное решение для проектирования и реализации программного обеспечения на основе объектно-ориентированного подхода: платформу .NET. Платформа .NET включает следующие основные аспекты:

- идеологию проектирования и реализации программного обеспечения;
- модель эффективной поддержки жизненного цикла прикладных систем;
- унифицированную, интегрированную технологическую платформу;
- современный, удобный в использовании, безопасный инструментарий для создания, размещения и поддержки программного обеспечения.

Платформу .NET удалось реализовать на качественно новом уровне, обеспечив существенное продвижение вперед в направлении гибкости интеграции с программно-

аппаратными ресурсами, безопасности и удобстве использования кода, а также снижении затрат на производство программного обеспечения. Платформа .NET имеет немало достоинств:

- вся платформа .NET основана на единой объектно-ориентированной модели. Все сервисы, интерфейсы и объекты, которые платформа предоставляет разработчику, объединены в единую иерархию классов;
- приложение, написанное на любом .NET-совместимом языке, является межплатформенным;
- в состав платформы .NET входит так называемый «сборщик мусора», который освобождает ресурсы; приложения защищены от утечки памяти и от необходимости освобождать ресурсы, что делает программирование более легким и безопасным.
- приложения .NET используют метаданные, что позволяет не пользоваться системным реестром ОС;
- приложения .NET используют безопасные типы – это повышает надежность, совместимость и межплатформенность программы;
- абсолютно все ошибки обрабатываются механизмом исключительных ситуаций;
- повторное использование кода стало удобнее. Это связано с тем, что промежуточный язык MSIL не зависит от языка программирования;
- универсальный интерфейс .NET Framework обеспечивает интегрированное проектирование и реализацию компонент приложений, разработанных согласно различным подходам к программированию;
- среда выполнения программ Common Language Runtime (CLR) реализует управление памятью, типами данных, межъязыковым взаимодействием, разворачиванием приложений;
- существенным позитивным отличием Microsoft .NET от существующих аналогов на современном рынке программного обеспечения является универсальная система типизации (UTS);
- веб-сервисы являются одной из важнейших составляющих идеологии .NET и центральной частью данной архитектуры, поскольку предназначены для реализации декларируемого Microsoft основополагающего принципа «программное обеспечение как сервис».

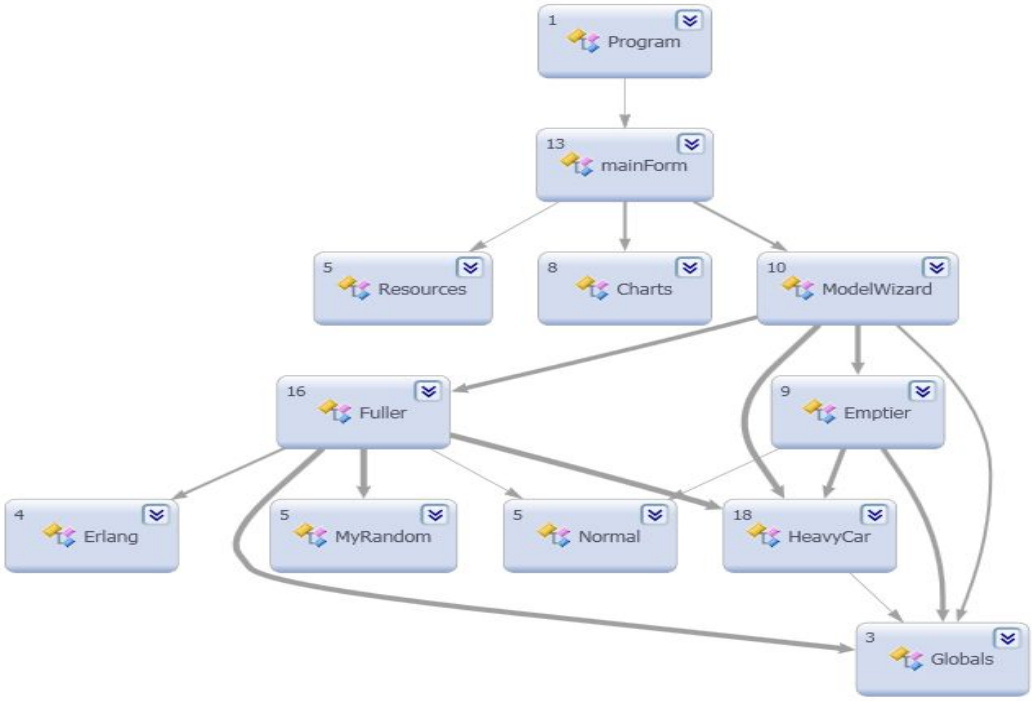


Рис. 1 – UML-диаграмма «Общая структура имитационной модели работы грузовых перевозок»

Несмотря на перечисленные выше инновации в области теории, технологии и практической реализации, в силу масштабности идеологии и новизны исследуемой проблематики, подход .NET не лишен отдельных недостатков.

Из-за того, что платформа .NET так радикально отличается от предыдущих технологий, Microsoft разработала под нее новый язык программирования C#. Благодаря тому, что C# представляет собой собранный из нескольких языков гибрид, он является простым (с синтаксической точки зрения), мощным и гибким. В него входит много полезных особенностей — простота, объектная ориентированность, типовая защищенность, «сборка мусора», поддержка совместимости версий и много других не менее важных свойств. Все имеющиеся возможности языка C# позволяют быстро и легко разрабатывать приложения, особенно приложения COM+ и Web-сервисы.

Из средств разработки для платформы .NET нами было использовано Microsoft Visual Studio.NET 2010, хотя существуют более двух десятков компиляторов независимых производителей, большая часть из которых может быть использована совместно с Visual Studio .NET.

С использованием данных технологий было разработано программное обеспечение, моделирующее работу грузовых перевозок. Модель заимствована из [2], и для разработки приложения использованы язык C# и платформа .NET. Данное приложение построено по всем принципам объектно-ориентированного подхода. Использовано множество библиотек платформы .NET, в том числе и для вывода графиков (рис.1).

Приложение функционирует во всех операционных системах семейства Windows.
Программа демонстрирует следующее:

1. ввод необходимых данных;
2. выбор необходимых режимов работы;
3. обработку данных;
4. вывод полученных результатов моделируемой системы в виде таблицы;
5. выбор зависимости для показа графиков;
6. вывод результатов в виде графиков (рис. 2);
7. сравнительный анализ полученных результатов.

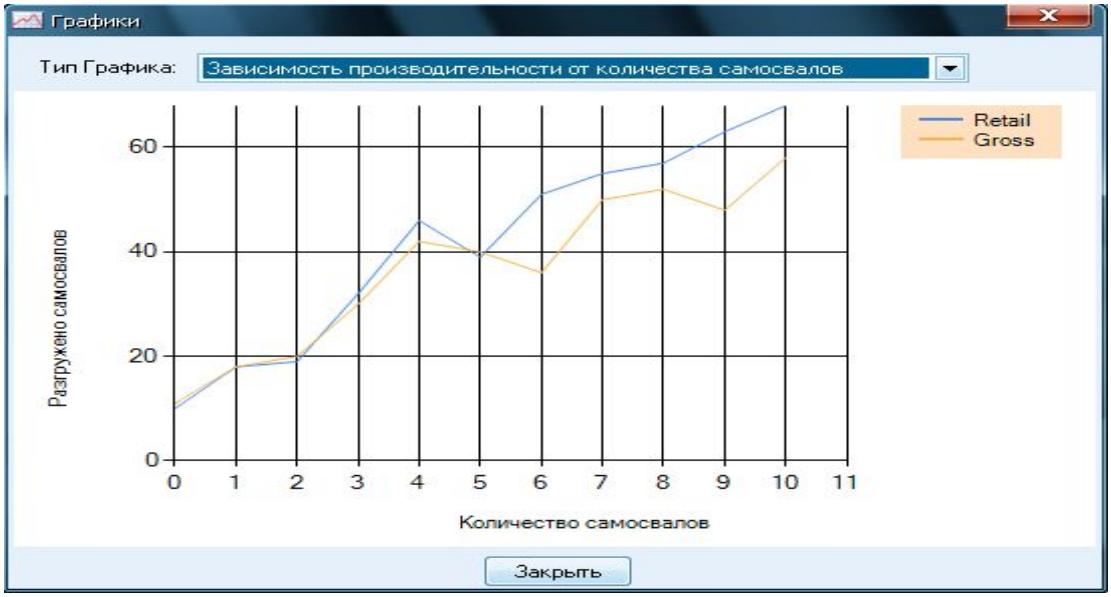


Рис. 2 – Форма приложения «Графики»

В составе среды Visual Studio 2010 имеется система WPF (Windows Presentation Foundation) – это графическая система, предназначенная для создания интерфейса программ. Данная система использует вспомогательный расширенный язык разметки приложений-XAML. Данная система была создана для облегчения и ускорения создания приложений. Преимущество системы WPF заключается в том, что дизайнеры занимаются интерфейсом программы, а разработчики непосредственно самим исполнительным кодом.

С использованием системы WPF и технологии объектно-ориентированного программирования было разработано приложение, имитирующее работу микроволновой печи [2]. Система классов была дополнена классами Приготовление, Разогрев и Разморозка, которые являются наследными от базового класса, представляющего типы операций, которые могут быть выполнены с пищей. Реализация наследования на C# отличается от C++. Данное приложение демонстрирует основные функции работы микроволновой печи (Рис.3)

Язык программирования C# и платформа .NET дают большие возможности для разработки объектно-ориентированного программного обеспечения.

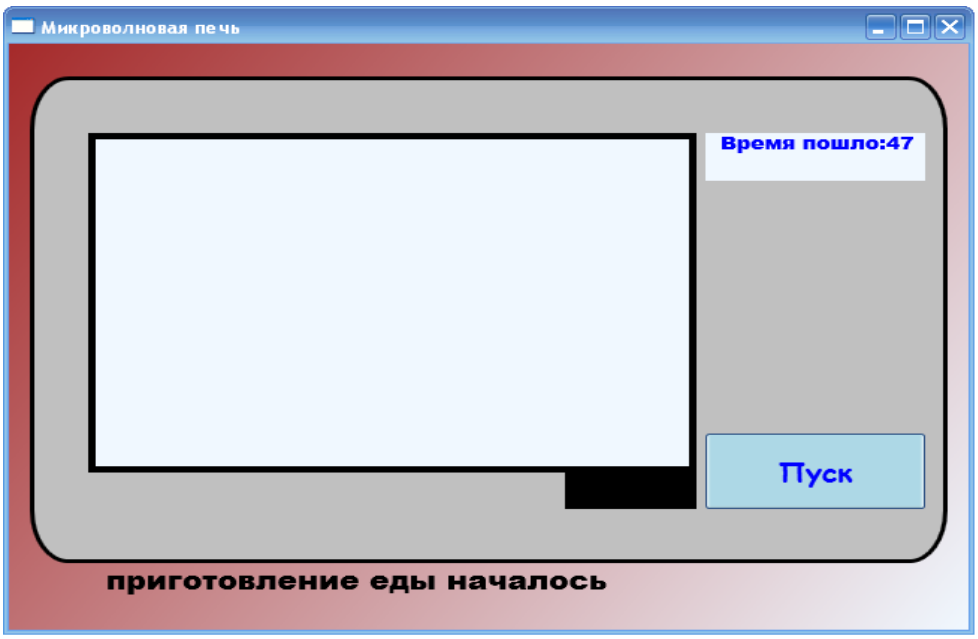


Рис. 3 – Имитационная модель работы микроволновой печи

Литература

1. Троелсон Э. Язык программирования C# 2010 и платформа .Net 4.0., 5-е изд.: Пер. с англ. – М: ООО «И.Д.Вильямс», 2011, 1392 с.
2. Труб И.И. Объектно-ориентированное моделирование на C++: Учебный курс. – СПб.: Питер, 2006. – 411 с.

УДК 004.4

РАЗРАБОТКА КОНЦЕПТУАЛЬНОЙ МОДЕЛИ ИНТЕЛЛЕКТУАЛЬНОЙ СИСТЕМЫ ПРОГНОЗИРОВАНИЯ ЖИЗНЕННОГО ЦИКЛА ЭЛЕМЕНТОВ МАГИСТРАЛЬНОГО НЕФТЕПРОВОДА

Гришина Александра Сергеевна, аспирант, Россия, Комсомольск-на-Амуре
Ale-solnishko@yandex.ru
Горькавый Михаил Александрович, к.т.н., Россия, Комсомольск-на-Амуре idpo@knastu.ru

В России очень остро стоит проблема повышения энергоэффективности и экологической безопасности нефтегазодобывающих предприятий. В настоящее время на таких предприятиях используется достаточно большое количество АСУ ТП [1], как правило, они установлены на локальных объектах и обслуживают только основной производственный процесс, а именно – добычу углеводородного сырья, не затрагивая при этом, например, транспортное обслуживание и проведение плановых и экстренных ремонтов. К тому же эти системы, действуя локально, обеспечивают управление и мониторинг только отдельно взятого объекта или процесса, в то время как для принятия эффективных решений требуется иметь интегральную информацию обо всех объектах и процессах технологической инфраструктуры нефтегазодобывающих предприятий. В связи с этим такие системы не могут обеспечить ЛПР

(лицо, принимающее решение) необходимой информацией обо всех процессах, влияющих на эффективность и экологическую безопасность работы предприятия [2].

В настоящее время задачи поддержания технического состояния магистральных нефтепроводов решаются с использованием различных методов диагностического мониторинга. Наличие и анализ только оперативной информации не позволяет осуществить прогноз будущего состояния узлов нефтепровода и предвидеть возникновение нештатных или аварийных ситуаций в сети.

Таким образом, одной из важнейших задач в рамках стратегии инновационного развития системы магистральных нефтепроводов является разработка новых, более эффективных механизмов обеспечения надежности основных узлов транспортирующих сетей.

В рамках решения представленной задачи в Комсомольском-на-Амуре государственном техническом университете ведутся исследования, направленные на разработку интеллектуальной системы мониторинга и прогнозирования жизненного цикла основных узлов магистральных трубопроводов. Основной особенностью системы является разработка решения не только на основе количественной оперативной информации текущего мониторинга, но и на основе результатов интеллектуального анализа большого массива качественной информации.

Важным компонентом такой системы является подсистема, обеспечивающая поддержку принятия решений для ЛПР. Такая система поддержки принятия решений (СППР) должна на основе анализа статической и динамической информации об объектах технологической инфраструктуры магистрального нефтепровода вырабатывать для ЛПР рекомендации по предотвращению аварийных ситуаций на магистральном нефтепроводе, улучшению показателей его работы, о проведении планового технического обслуживания и экстренного ремонта объектов нефтепровода, о списании объектов [2] нефтепроводов и замене их новыми.

Диаграмма классов исследуемого объекта «магистральный нефтепровод» (МНП), представлена на рисунке 1. Она является центральным звеном методологии объектно-ориентированных анализа и проектирования.

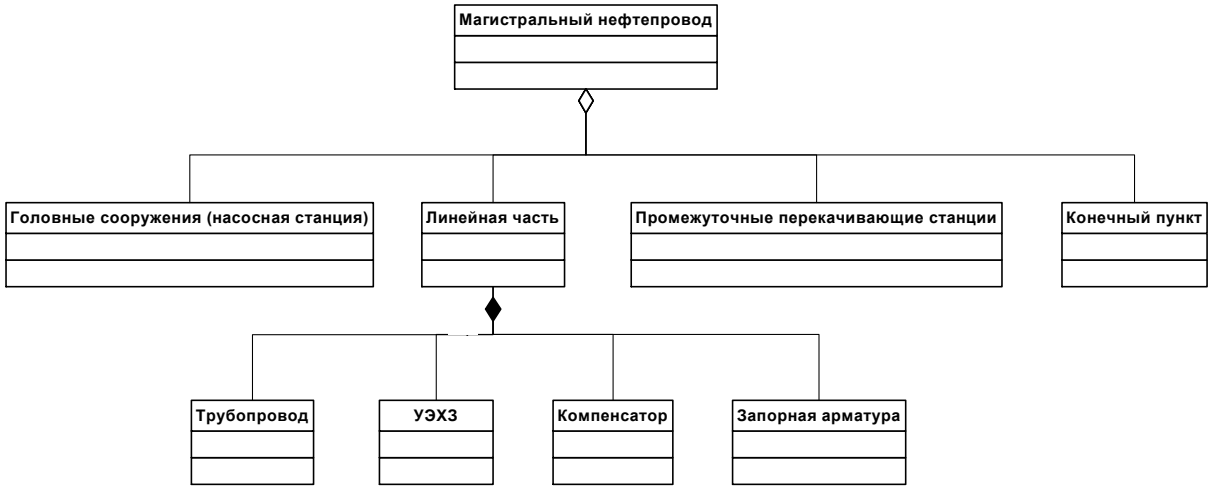


Рис. 1 – Диаграмма классов магистрального нефтепровода

Диаграмма классов показывает классы и их отношения, тем самым представляя логический аспект проекта.

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов (магистральный нефтепровод) представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности (насосные станции, линейная часть, промежуточные перекачивающие станции, конечный пункт). Данное отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа "часть-целое". Раскрывая внутреннюю структуру системы, отношение агрегации показывает, из каких компонентов состоит система и как они связаны между собой.

Отношение композиции является частным случаем отношения агрегации. Это отношение служит для выделения специальной формы отношения "часть-целое", при которой составляющие части в некотором смысле находятся внутри целого. Специфика взаимосвязи между ними заключается в том, что части (трубопровод, УЭХЗ, компенсатор, запорная арматура) не могут выступать в отрыве от целого (линейная часть), т. е. с уничтожением целого уничтожаются и все его составные части.

Диаграмма вариантов использования интеллектуальной системы мониторинга и прогнозирования узлов магистрального нефтепровода (МНП) представлена на рисунке 2.

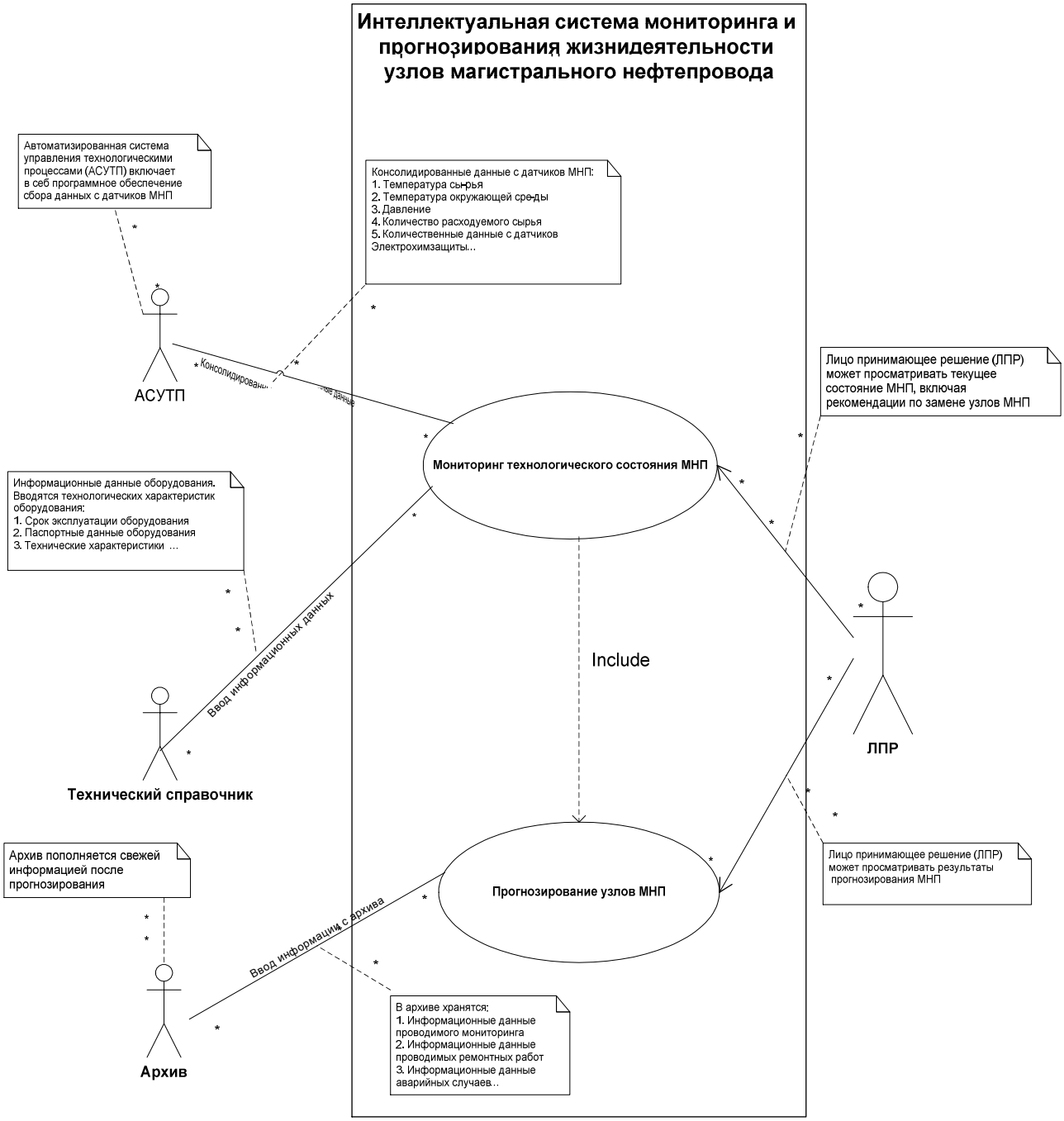


Рис. 2 – Диаграмма вариантов использования интеллектуальной системы мониторинга и прогнозирования узлов магистрального нефтепровода (МНП)

Диаграмма вариантов использования (Use-Cases Diagram) – это UML-диаграмма, с помощью которой в графическом виде изображаются требования к разрабатываемой системе. Use-Cases Diagram представляет собой исходную концептуальную модель проектируемой интеллектуальной системы мониторинга и прогнозирования узлов МНП.

Модель по поступающим данным проводит интеллектуальный анализ и выдает результат лицу, принимающему решения (ЛПР).

Диаграмма вариантов использования состоит из актеров (АСУТП, технический справочник, архив, ЛПР), для которых система производит действие и собственно действия Use Case (мониторинг технологического состояния МНП и прогнозирование состояния узлов МНП), которое описывает то, что актер хочет получить от системы. Актер обозначен значком человечка, а Use Case (вариант использования) – эллипсом.

«АСУТП» включает в себя программное обеспечение сбора данных с датчиков МНП.

«Технологический справочник» предназначен для сбора информационных данных оборудования транспортирующей системы. К информационным данным можно отнести информацию с паспортов оборудования (номинальное давление, температура при котором полноценно работает оборудование, материал изготовления оборудования, срок эксплуатационной службы и т.д.), характеристики при которых может произойти поломка того или иного оборудования и т.д. Ввод технологических параметров оборудования производится вручную рабочим персоналом.

«Архив» содержит информацию за все предыдущие периоды: данные проводимого мониторинга (внутренняя диагностика труб, пеший обход и т. д.), ремонтных работ (ликвидация трещин, замена линейной части трубопровода, замена арматуры и т.д.), данные аварийных случаев (когда, как, происходила авария, результатом чего произошла авария и т.д.).

Варианты использования применяются для спецификации внешних требований к проектируемой системе. Множество вариантов использования в целом определяют все возможные стороны ожидаемого поведения системы, а также устанавливают требования, определяющие, как актеры должны взаимодействовать с системой, чтобы иметь возможность корректно работать с предоставляемыми сервисами. Дополнительно в диаграмме добавлены комментарии для более подробного представления взаимодействия актеров и вариантов использования.

Между актерами и вариантами использования имеются различные виды взаимодействия. Так, например, актеры (АСУТП, архив, технический справочник) взаимодействуют с вариантами использования с помощью простой ассоциации. Простая ассоциация отражается линией между актером и вариантом использования (без стрелки).

Актер ЛПР взаимодействует с Use Case с помощью направленной ассоциации. Направленная ассоциация показывает, что вариант использования инициализируется актером. Обозначается стрелкой. Направленная ассоциация позволяет ввести понятие основного актера (он является инициатором ассоциации) и второстепенного актера.

Включение – показывает, что вариант использования «мониторинг технологического состояния МНП» включается в вариант использования прогнозирование состояния узлов МНП и выполняется всегда.

Система позволит повысить уровень прогнозируемости текущего состояния технологического оборудования и вести аппаратный контроль технического состояния объектов и работы всей системы с выводом результатов непосредственно лицу, принимающему решения. Более, того система позволит снизить затраты на проведение периодических обследований, позволит осуществить непрерывный контроль и прогнозирование технического состояния объектов в процессе их эксплуатации, основываясь на различных методах неразрушающего контроля, методах измерения напряженно-деформированного состояния, средствах измерения рабочих параметров технологического процесса и методах слежения за факторами, влияющими на износ оборудования.

При помощи данной системы представляется возможным прослеживать износ оборудования, арматуры, линейной части трубопроводов, выявлять потенциально аварийные элементы системы магистральных трубопроводов, что обеспечит своевременную замену изношенных узлов и предотвратит появление аварий.

Таким образом, разработка и внедрение системы мониторинга и прогнозирования

жизненного цикла основных узлов магистральных нефтепроводов позволит обеспечить устойчивость реализации инновационной стратегии развития нефтетранспортирующих сетей.

Литература

1. Байков И.Р., Самородов Е.А., Ахмадуллин К.Р. Методы анализа надежности и эффективности систем добычи и транспорта углеводородного сырья. – М.: ООО «Недра-Бизнесцентр», 2003.
2. Загорулько Ю.А., Загорулько Г.Б., Кравченко А.Ю., Сидорова Е.А. Разработка системы поддержки принятия решений для нефтегазодобывающего предприятия // Труды 12-й национальной конференции по искусственному интеллекту с международным участием – КИИ-2010. – Москва: Физматлит, 2010. –Т.3. –С.137-145.

УДК 004.272.44, 004.4.414, 004.81

АНАЛИЗ ЯЗЫКА С ПОМОЩЬЮ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ ОБЪЕКТНО-АТРИБУТНОЙ АРХИТЕКТУРЫ[1]

Салибекян Сергей Михайлович, ст. преподаватель, Московский институт электроники и математики (технический университет), Россия, Москва, salibek@yandex.ru

Панфилов Пётр Борисович, к.т.н., доцент, Московский институт электроники и математики (технический университет), Россия, Москва, panfilov@miem.edu.ru

Распознание текста – проблема в современной вычислительной технике весьма актуальная: компиляция, трансляция, поисковые запросы, смысловое распознание текста, автоматический перевод текста с одного языка на другой и т.д. В настоящее время для подобных целей используется несколько подходов: теория формальных языков (формальные грамматики, конечные автоматы) [1], нейронные сети, теория фреймов [2], концепция «Смысл-текст» [3]. В данной же статье описывается новый способ распознания языка – анализ с помощью объектно-атрибутного подхода (данный подход разработан в Московском институте электроники и математики).

Задача создания нового языка программирования и подходящего метода компиляции для него встала после разработки концепции объектно-атрибутной (ОА) архитектуры вычислительной системы с управлением потоком данных (dataflow) – старые парадигмы программирования для описания работы такой системы были неудобны: классическая императивная парадигма оказалась неприменимой, т.к. она несовместима с парадигмой dataflow; функциональные языки и акторные системы (в основном используемые в парадигме dataflow) имеют весьма скудные возможности абстракции данных и программы, объектно-ориентированная и агентная парадигмы весьма затруднительно реализовывать на распределенных вычислительных системах. В результате, мы пришли к выводу, что и язык программирования и компилятор, используемый для его распознания, также должны быть выполнены по принципам ОА-архитектуры (ОА-архитектура оказалась применимой не только к аппаратной, но и к программной части вычислительной системы) [4]. И, конечно, самым весомым доводом в пользу использования ОА-подхода для создания компилятора стало наше стремление к унификации разрабатываемой вычислительной системы: аппаратура, язык программирования и компилятор, работающие по одним и тем же принципам, делают вычислительную систему целостной и устраняют так называемый семантический разрыв (например, различие между организацией классического языка высокого уровня и машинным языком фон неймановского компьютера требует больших вычислительных затрат на компиляцию и делает скомпилированные программы неоптимальными).

В настоящее время в рамках НИР «Исследование и разработка архитектуры и среды программирования перспективной суперкомпьютерной системы на основе динамической

[1] Статья рекомендована к опубликованию в журнале "**Информационные технологии**"

модели вычислений с управлением потоком данных», выполняемой по федеральной целевой программе «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы» ведется разработка среды программирования и модели суперкомпьютера, функционирующих по правилам ОА-архитектуры. В рамках данного проекта осуществлено создание новой версии компилятора ОА-языка, обладающего следующими качествами: быстрый запуск среды программирования, распределенный режим работы компилятора, б**о**льшее разнообразие конструкций ОА-языка (по сравнению со старой версией), возможность добавления в ОА-язык конструкций языка Си для облегчения написания фрагментов программы, работающих в последовательном стиле.

В процессе выполнения НИР был отработан синтаксис ОА-языка программирования и выработаны основные приемы создания систем распознания языка. Однако ОА-концепция компиляции применима не только для работы с машинными языками высокого уровня (что было сделано в рамках НИР), но и для распознания естественного языка, где требуется высокая степень абстракции данных. Далее необходимо привести небольшой пример реализации ОА-системы для анализа языка.

ОА-вычислительная система представляет собой набор функциональных устройств, обменивающихся информационными парами (ИП), представляющими собой совокупность нагрузки (данные или указатель на другую информационную конструкцию) и ярлыка (уникальный идентификатор нагрузки), т.е. ОА-система работает по принципу dataflow (управление вычислительным процессом с помощью потока данных). ИП могут быть двух видов:

1. Милликоманда – предназначена для управления ФУ-ом;
2. Информационная ИП – служит для описания признака какого-либо объекта.

ИП объединяются в капсулы [4], которые используются либо для описания признака какого-то объекта, либо хранят последовательность милликоманд (миллипрограмма). Форматы милликоманды и информационной ИП совпадают, что позволяет объединять данные и миллипрограмму в одной информационной конструкции.

Для распознания текста в ОА-системе в основном применяются следующие ФУ:

1. ФУ Find (поиск) предназначен для поиска заданных ИП в одной капсуле и запуска миллипрограмм, исходя из результата поиска: «удача» (Success), когда выполняется условие поиска, и «неудача» (Fail), когда условие поиска не выполняется. ФУ реализует следующие милликоманды (в скобках приведена мнемоника милликоманды):

- установить указатель на миллипрограмму, запускаемую при выполнении условия поиска (**SuccessProgSet**);
- установить указатель на миллипрограмму, запускаемую при невыполнении условия поиска (**FailProgSet**);
- установить указатель на капсулу, по которой будет осуществляться поиск (**Set**);
- поиск по капсуле одной информационной пары (**FindIc**), которая пришла в качестве нагрузки к милликоманде.

Получается, что миллипрограмма, на которую указывает SuccessProg, выполняет роль блока программы условного оператора then, а FailProg – блока else, в классической конструкции языка программирования высокого уровня if-then-else.

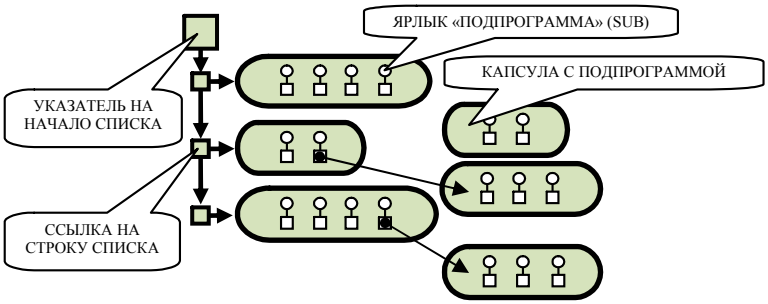


Рис. 1 – Информационная конструкция «Список»

Для реализации же множественного выбора (конструкция switch в языке Си или case в Pascal) предназначено ФУ «Список». В контексте данного ФУ находится ссылка на список капсул, где хранятся данные для осуществления поиска (рис. 1). В ОА-список кроме данных для поиска могут быть добавлены и миллипрограммы обработки данных и осуществления перенаправления информационного потока. ОА-список также может быть использован и в качестве таблицы переменных, где хранятся сведения обо всех мнемониках, что, например, встречаются в распознаваемой программе высокого уровня (переменные, константы, указатели и т.д.).

Источником же данных для ОА-системы распознания языка является ФУ лексического разбора, которое разбивает поступающий к нему текст на лексемы. В состав данного ФУ входит регистр, где хранится милликоманда, которая должна быть прикреплена к генерируемой лексеме. На начальной стадии разбора с помощью специальной милликоманды в регистр заносится милликоманда для первого ФУ, которое будет заниматься разбором лексемы в самом начале процесса компиляции. ФУ лексического разбора строится на основе конечного автомата, т.к. лексический анализ довольно прост и здесь не требуется особых «изысков». На выходе с ФУ лексического разбора язык будет представлен в виде потока милликоманд с ИП, описывающими лексемы: <Mk,<a,M>>, где Mk э N – милликоманда для ФУ, a э N – атрибут (индекс) нагрузки, M – нагрузка, представляющая собой указатель на ячейку памяти, где храниться информационная конструкция, передаваемая в качестве операнда, <…> – обозначение ИП, N – множество натуральных чисел.

В процессе распознания могут быть использованы не только вышеперечисленные типы ФУ, но и другие ФУ, которые могут выполнять различные вычислительные операции, операции ввода-вывода, хранение информации и т.д., что существенно повышает возможности ОА-системы по сравнению с классическим конечным автоматом. Теперь перейдем к описанию принципа функционирования ОА-системы в целом.

Для начала создания ОА-компилятора необходимо составить синтаксическую диаграмму для него. Затем для каждого узла этой диаграммы в ОА-системе создается свое ФУ: для тех вершин, из которых выходит только одна дуга, применяется ФУ «Поиск», для остальных – ФУ «Список». Также создается и ФУ лексического разбора, которое будет являться источником данных в ОА-системе: оно разбивает исходный текст на лексемы и выдает их для дальнейшего анализа на другие ФУ. В состав ФУ лексического анализа входит регистр, где хранится милликоманда, которая «прикрепляется» к генерируемой лексеме.

Для примера возьмем синтаксическую диаграмму простейшего оператора присваивания языка Си: [Var | * Var] = [Const | Var | *VarPointer], где Var – переменная, Const – константа, VarPointer – указатель на переменную, «|» – знак перечисления вариантов языковых конструкций. На рис. 2 представлена диаграмма синтаксического разбора, т.е. на данную стадию разбора приходят не отдельные символы, а информационные пары, обозначающие лексемы (лексический анализ проводит ФУ лексического разбора). К дугам графа приписываются лексемы, по которым ОА-система переходит из одного состояния в другое. Но, в отличие от синтаксической диаграммы распознающего автомата, к дугам диаграммы также приписываются и действия, которые должна выполнять ОА-система, реагируя на приходящую лексему (а не символы, выдаваемые на выходную ленту, как в классической теории автоматов-трансляторов). Такое решение дает программисту намного больше возможностей при создании распознающей системы, чем конечный автомат. Как уже говорилось, каждому узлу диаграммы соответствует свое ФУ – и обозначение (мнемоника) этого ФУ приписано к вершине графа. Основной алгоритм работы ОА-системы распознавания следующий: перед началом осуществляется настройка всех ФУ, и в том числе производиться и настройка ФУ лексического анализа, чтобы оно выдавало первую лексему на ФУ, отвечающее за начало анализа (в нашем случае Root). Далее ФУ, которому от ФУ лексического разбора поступила ИП, анализирует пришедшую лексему и в зависимости от результата анализа перенастраивает лексический анализатор таким образом, чтобы тот

«выбрасывал» очередную лексему для ФУ, ответственного за следующую стадию синтаксического анализа.

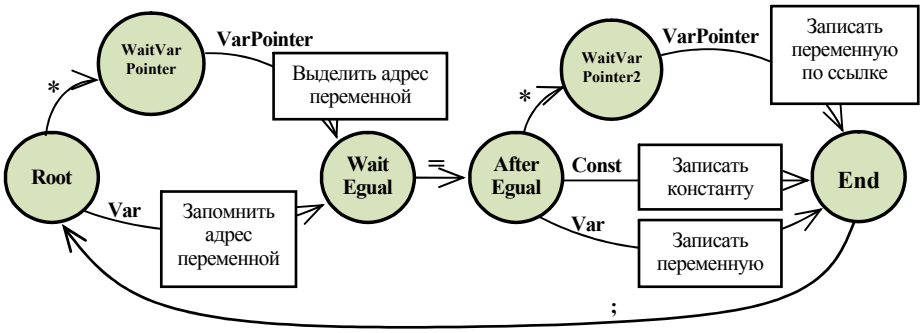


Рис. – 2 Синтаксическая диаграмма разбора оператора присваивания языка Си

Приведем фрагмент ОА-программы (Листинг 1), соответствующий вышеприведенной синтаксической диаграмме (синтаксис ОА-языка описан в [5]): здесь приводится описание инициализации первых трех ФУ ОА-системы.

```
    Root.Set=
    >{Separator="*" Lex.ReceiverMkSet=WaitVarPointer.FindIc}
    >{Var=nil Root.InObjPopMk=VarManager.Set
Lex.ReceiverMkSet=WaitEqual.FindIc}
    >{0=nil Console.Out="Wrong variable describeing" Lex.Stop}

    WaitVarPointer.Set={VarPointer=nil}
    WaitVarPointer.SuccessProgSet=
            {WaitVarPointer.InObjPointPop=VarManager.Read
Lex.ReceiverMkSet=WaitEqual.FindIc}
    WaitVarPointer.FailProgSet={Console.Out="Pointer not finded" Lex.Stop}

    WaitEqual.Set={Separator="="}
    WaitEqual.SuccessProgSet={Lex.ReceiverMkSet=AfterEgual.FindIc}
    WaitEqual.FailProgSet={Console.Out=" '=' not finded"   Lex.Stop}
    ……————
    Lex.Lexing="x=10; y=*Uk" \*Запуск генератора лексем*\
```

Листинг 1 – ОА-программа синтаксического разбора

На листинге присутствуют следующие обозначения:

- Root.Set – милликоманда установки списка для ФУ Список (Root);
- ➢ - знак начала новой капсулы, входящей в ОА-список;
- Separator – атрибут символа-разделителя;
- nil – обозначает, что в качестве нагрузки ИП могут выступать любая константа или указатель;
- Var – атрибут переменной;
- FindIc – милликоманда поиска одной ИП в капсуле или списке;
- VarManager – ФУ, ответственное за работу с переменными;
- VarManager.Read – милликоманда чтения переменной из указанного в нагрузке адреса;
- Console.Out – милликоманда вывода сообщения на выводную консоль;
- Lex – обозначение ФУ лексического разбора;
- Lex.ReceiverMkSet – установка милликоманды, прикрепляемой к генерируемой лексеме;
- Lex.Lexing – запуск лексического разбора (в нагрузке символьная строка для разбора).
- Lex.Stop –милликоманда остановки ФУ лексического разбора.

Как видно из листинга 1, при выполнении условия поиска ФУ с помощью милликоманды Lex.ReceiverMkSet перенастраивает ФУ лексического разбора таким образом, чтобы оно выдавало очередную лексему на следующую стадию синтаксического разбора.

Т.е. ФУ работают асинхронно (параллельно), и вычисления активизируются с помощью милликоманд, которыми ФУ обмениваются между собой. Таким образом, ОА-распознающая лексическая система работает по принципу dataflow, который открывает огромные возможности:

1. распараллеливание вычислений;
2. реализация анализа на распределенной и гетерогенной (состоящий из вычислительных узлов различной архитектуры) вычислительной системе.

Напомним, что конечный автомат, активно используемый сейчас для анализа языка, по определению является последовательным, т.к. он в один момент времени должен находится в только одном состоянии q э Q, где Q – множество всех возможных состояний автомата (недетерминированный автомат в плане параллелизма особых преимуществ не дает, т.к. он также работает в последовательном режиме, обрабатывая один символ со входной ленты в один момент времени). Параллелизм в классической системе распознания языка может быть обеспечен лишь благодаря использованию нескольких уровней (проходов) компиляции (например, лексический и синтаксический уровни), где для каждого уровня применяется свой распознающий автомат. ОА же система состоит из совокупности параллельно работающих ФУ. А для более сложного распознания (далее синтаксического уровня языка) конечный автомат неприменим, т.к. он не может работать с глубокой абстракцией данных. ОА же архитектура подходит и для смыслового анализа языка, т.к. обладает следующими преимуществами, связанными с удобством абстракции данных:

1.Высокий уровень абстракции данных и программы [6].

2.Возможность совмещать в одной информационной структуре как данные, так и программу их обработки.

3.Возможность динамического синтеза и перестройки абстрактных данных. Это свойство самое важное для смыслового анализа текста: смыслы (объекты), заложенные в тексте, в подавляющем числе случаев не вписываются в рамки стандартных шаблонов (фреймов). И именно динамический синтез абстракций позволит по заложенным в ОА-систему правилам создавать описания объектов, заранее неизвестных программисту.

Наиболее близкой к предложенному способу смыслового распознания текста является концепция советского лингвиста Игоря Мельчука «Смысл-текст» [3], где синтез смысловых конструкций осуществляется не напрямую, а через несколько этапов (уровней) – от простого к сложному (рис. 3).

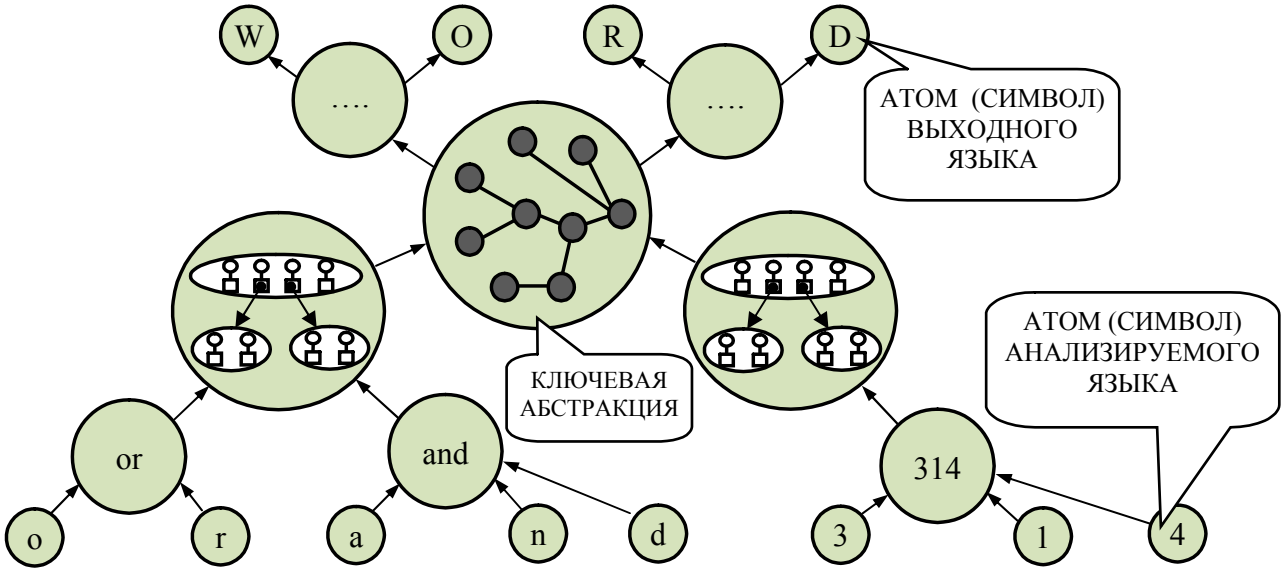


Рис. 3 – ОА-конус абстракций

Мельчук предлагает пять уровней: семантический (уровень смысла), глубинно-морфологический, поверхностно-морфологический, глубинно-синтаксический, поверхностно-синтаксический, фонологический, фонетический (уровень текста) – хотя и

делает оговорку, что число уровней может быть и другим (на усмотрение разработчика модели). Смысл же, заложенный в тексте, Мельчук представляет в виде графа, вершинами которого являются так называемые семы – элементарные (неразложимые) смысловые единицы. В виде графов представляются и отдельные элементы на глубинно-морфологическом и поверхностно-морфологическом уровнях. Однако концепция «Смысл-текст» не получила широкого распространения ввиду своей сложности и, самое главное, негибкости (так, семы в концепции Мельчука представляли собой несколько видов предикатов с фиксированным числом аргументов), что существенно затрудняло синтез абстракций.

В предлагаемой концепции смыслового анализа текста синтез смысловых абстракций также осуществляется от простого к сложному (рис. 3): процесс синтеза начинается со смысловых атомов (элементарных смыслов), в случае анализа текста смысловыми атомами являются символы. Далее осуществляется лексический анализ: выделение из последовательности символов лексем – данной работой в ОА-системе занимается ФУ лексического разбора. Как правило, синтезированные на данном этапе абстракции представляют собой отдельные ИП, в которых атрибутом является идентификатор типа лексемы, а в нагрузке хранится сама лексема (символ-разделитель, число, строка и т.д.). Далее лексемы поступают на этап синтаксического анализа, где происходит анализ взаимосвязи лексем. На этом этапе формируемые абстракции довольно сложны и представляют собой информационные капсулы или ОА-информационные деревья (смысловой граф) [4, 6]. Абстракции эти передаются для дальнейшего анализа на следующий уровень анализа, где ФУ синтезируют из них еще более сложные абстракции. Основная нагрузка по синтезу абстракций на уровнях от синтаксического и выше ложится на ФУ «Поиск» и «Список», которые производят анализ поступивших абстракций и вызов соответствующих миллипрограмм для их обработки. Процесс анализа текста представляет собой так называемый конус абстракций: в его основании располагаются смысловые атомы (символы текста), в вершине – ключевая абстракция, описывающая смысл всего распознаваемого текста. Если же необходимо осуществить перевод с одного языка на другой, то прямой конус абстракций дополняется конусом обратным, который осуществляет синтез текста на другом языке из ключевой абстракции (рис. 3).

ОА-подход, обладающий большой гибкостью при построении смысловых конструкций, позволит в будущем осуществлять не только анализ машинных языков высокого уровня, относящихся к классу контекстно-независимых по иерархии Хомского, но и живых языков, относящихся к контексто-зависимым и свободным. Так, в ОА-модель можно, например, добавлять признаки стилистической окраски текста, ассоциативные связи и т.д; т.е. ОА-ситема будет представлять собой мощную базу знаний для анализа текста.

В заключение следует сказать о том, какая работа была проделана в области распознания текста с применением ОА-архитектуры:

- разработана концепция разбора языка с помощью ОА-системы;
- отработаны основные типы ФУ, используемые для языкового анализа;
- создана среда программирования для написания ОА-программ [6], которая может быть использована в том числе и для анализа текста;
- с помощью ОА-среды программирования создан ОА-компилятор, который осуществляет перевод программы, написанной на специальной ОА-языке в ОА-образ (совокупность информационных капсул и миллипрограмм, описывающих алгоритм решения какой-либо задачи).

Литература

1. Ахо, Ульман Теория Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978
2. Минский М. Фреймы для представления знаний: Пер. с англ.- М.: Энергия, 1979.
3. Мельчук И.А. Опыт теории лингвистических моделей «СМЫСЛ <–>ТЕКСТ» – М.: Школа «Языки русской литературы», 1999.

4. Салибекян С.М., Панфилов П.Б. ОА-архитектура – новый подход к созданию объектных систем // Объектные системы – 2011: материалы III Международной научно-практической конференции (Ростов-на-Дону 10-12 мая 2011 г.) / Под общ. ред. П.П. Олейника. – Ростов-на-Дону, 2011. – С. 73-79 URL: http://objectsystems.ru/files/Object_Systems_2011_Proceedings.pdf
5. Салибекян С.М., Панфилов П.Б. Перспективная суперкомпьютерная система на основе объектно-атрибутной модели вычислений с управлением потоком данных / Международная конференция «Развитие суперкомпьютерных и грид-технологий в России» в рамках «Второго Московский Суперкомпьютерного форума» Россия, Москва, ВВЦ 26–27 октября 2011 года URL: http://www.hpc-platform.ru/tiki-download_file.php?fileId=82
6. Салибекян С.М., Панфилов П.Б. Объектно-атрибутный подход к построению интеллектуальных систем // Нейрокомпьютеры: разработки и применение. 2

УДК 004.4

ИЕРАРХИЯ КЛАССОВ МЕТАМОДЕЛИ ОБЪЕКТНОЙ СИСТЕМЫ

Олейник Павел Петрович, к.т.н., Системный архитектор программного обеспечения, ОАО «Астон», Россия, Ростов-на-Дону, xsl@list.ru

В настоящее время наиболее часто используемыми являются объектно-ориентированные языки программирования, популярность которых обусловлена наличием ряда хорошо известных преимуществ [1].

В данной статье подробно рассматривается иерархия классов, используемая для представления метамодели объектной системы, которая была апробирована и успешно применяется в корпоративном решении, ключевой особенностью которого является предоставление возможности добавления новых классов, описывающих сущности предметной области, без перекомпиляции приложения. Данная задача возникла из-за необходимости самостоятельного расширения функционала (путём добавления как новых атрибутов существующих классов, так и добавлением производных классов) опытными пользователями системы. Для решения задачи потребовалось разработать ряд структур, которые бы позволили сохранять информацию об имеющихся классах предметной области и о наличии в них атрибутов. Т.е. необходим механизм представления метамодели объектной системы для создаваемой среды разработки приложений.

Решение любой задачи начинается с формирования критериев оптимальности (КО), которым должна соответствовать полученная реализация. Для рассматриваемой задачи были сформулированы следующие критерии:

1. Разработать единую иерархию, позволяющую представить ключевые элементы объектно-ориентированной парадигмы, такие как классы, атрибуты, наследование (одиночное и множественное), ассоциации и т.п.
2. Предусмотреть возможность предоставления базовых (системных) классов, в которых реализован наиболее общий функционал
3. Разработать иерархию атомарных литеральных типов, которые представляют наиболее распространенные типы данных современных объектно-ориентированных языков программирования.

Рассмотрим требования каждого критерия более подробно. Реализация требования КО1 позволит унифицировано обрабатывать все элементы метамодели (которые представляются в виде экземпляров выделенных классов) с помощью различных структур данных, например с помощью итераторов при формировании цикла. При этом пользователю предоставляется возможность применять хорошо зарекомендовавшие себя объектно-ориентированные подходы, как например, наследование и организация ассоциаций.

КО2 требует наличия системных (встроенных) классов. В общем случае пользователи не могут модифицировать поведение этих классов, но имеют возможность наследоваться от

них. При этом наличие множественного наследования предоставляет широкие возможности моделирования задач прикладной предметной области.

Современные объектно-ориентированные языки программирования имеют в своем арсенале богатый набор встроенных атомарных литеральных типов. Поэтому необходимо предоставить подобный функционал в разрабатываемой системе, что соответствует требованию КО3. Так как добавлять атрибуты классов будут опытные пользователи, которые имеют слабые представления о принципах проектирования информационных систем, имеет смысл сократить количество атомарных встроенных типов данных, что позволит упростить процесс проектирования и уменьшить количество проблем, возникающих при рефакторинге [6], когда пользователь изменяет тип данных атрибута.

Сама идея представления метамодели объектной системы не нова. Так, в работах, посвященных стандарту объектно-ориентированных баз данных ODMG 3.0 [2,3], и в работах, описывающих язык запросов SQL:2003 [4] имеются определённые решения. Но первый стандарт (ODMG 3.0) в значительной степени ориентирован лишь на удобство обработки метамодели с помощью встроенных коллекций и итераторов, а второй (SQL:2003) ориентирован на представление сложных типов данных в ячейках таблицы, что обусловлено его направленность на реляционные БД.

По выше описанным причинам было принято решение самостоятельно разработать иерархию классов, полностью удовлетворяющую поставленной задаче. На рисунке 1 в виде диаграммы классов унифицированного языка UML [5] представлена разработанная иерархия.

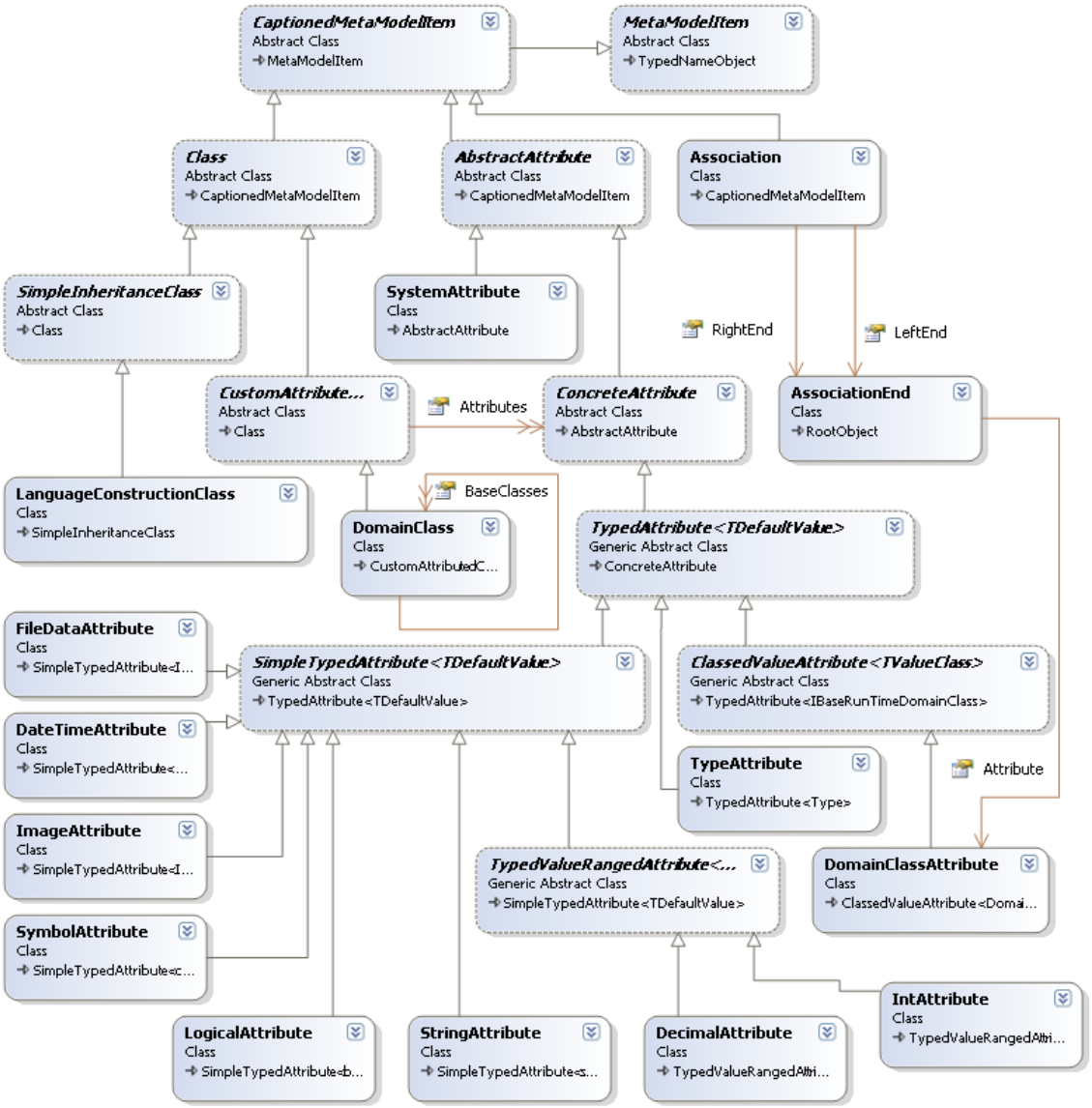


Рис. 1 – Иерархия классов метамодели объектной системы

Базовым классом рассматриваемой иерархии является MetaModelItem, который содержит общие свойства и методы, характерные для всех элементов метамодели. Основное свойство описываемого класса содержит тип самого элемента и используется многократно для получения метаинформации. Элементы метамодели, которые должны иметь название и заголовок (класс, атрибут класса, ассоциация и т.п.), должны быть унаследованы от CaptionedMetaModelItem. Для представления различных классов системы используется корневой абстрактный класс Class.

В системе предусмотрено два вида наследования. Первое предполагает простое наследование классов без возможности добавления атрибутов пользователем. Данный тип наследования может использоваться классами лингвистического транслятора, который предполагает создание иерархии классов, представляющей элементы русского языка (например, морфемы). Для решения описанной задачи используется абстрактный класс SimpleInheritanceClass.

Для реализации классического наследования, предполагающего добавление, как унаследованных классов, так и определения имеющихся атрибутов, используются производные от CustomAttributedClass классы. В частности, реализованный (неабстрактный) DomainClass класс используется для реализации сущностей предметной области, каждая из которых может быть унаследована от нескольких базовых. Множественное наследование позволяет проектировать более гибкие системы, автоматизирующие широкий спектр прикладных предметных областей. Например, при проектировании классов, представляющих некоторые товары, может потребоваться представить их структуру в виде дерева. Т.е. необходимо одновременно наследоваться от базового класса, представляющего товар, и от системного класса древовидной структуры.

При проектировании иерархии атомарных литеральных типов (целых чисел, строк, дробных чисел и т.п.) использовался общий подход, подробно описанный в работе [7]. Однако с целью упрощения процесса рефакторинга было принято решение о сокращении набора классов и применение параметризованных типов (Generic в языке C#). Корневым является класс AbstractAttribute, от которого унаследованы SystemAttribute и ConcreteAttribute. Первый используется для описания атрибутов системных классов, которые не могут редактироваться или добавляться пользователем. Второй представляет непосредственно атрибуты, добавляемые пользователем в режиме выполнения приложения.

Базовый параметризованный класс TypedAttribute является корневым для всех атрибутов, у которых параметр типа выступает в качестве типа данных для значения атрибута. Абстрактный класс SimpleTypedAttribute является корнем иерархии атомарных литеральных типов. В свою очередь, ClassedValueAttribute является корнем для всех атрибутов, значениями которых выступает объект определенного класса.

Для представления связей между классами используется механизм ассоциаций языка UML[5], для реализации которого выделен класс Association. В системе предусмотрена возможность описания только бинарных связей, т.к. для них проще всего определить кратность каждого участника. Поэтому для представления отдельного края ассоциации используется экземпляр класса AssociationEnd.

Рассмотрим соответствие данной иерархии выделенным ранее критериям оптимальности. Иерархия соответствует требованиям КО1, т.к. она содержит все ключевые элементы объектно-ориентированной парадигмы, такие как классы, атрибуты, ассоциации и предоставляет возможность реализации наследования (как одиночного, так и множественного). Также выполнены требования КО2, т.к. имеется возможность предоставления базовых (системных) классов, в которых реализован наиболее общий функционал. Разработанная иерархия классов, представляющая атомарные литеральные типы, которые представляют наиболее распространенные типы данных современных объектно-ориентированных языков программирования. Поэтому можно утверждать, что выполнены требования КО3.

В заключение отметим, что дальнейшим развитием описанной метамодели может являться добавление классов, предоставляющих механизмы фильтрации данных и элементов, позволяющих описать элементы поведения (например, правила валидации) и принципы отображения (подходы формирования графических форм приложения). Также необходимо предусмотреть возможность добавления пользователем программного кода, реализующего бизнес-логику предметной области, т.к. невозможно предусмотреть заранее весь функционал, который потребуется в прикладной предметной области.

Литература

1. Грэхем И., Объектно-ориентированные методы. Принципы и практика. 3-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. – 880 с.: ил. – Парал. тит. англ.
2. Cattell R.G., Barry D.K. The Object Data Standard:ODMG 3.0, Morgan Kaufmann Publishers, 2000, 288p.
3. Джордан Д. Обработка объектных баз данных на C++. Программирование по стандарту ODMG. : Пер. с англ. : Уч. пос. – М.:Издательский дом "Вильямс", 2001.-384с.:ил.-Парал. тит. англ.
4. Calero C., Ruiz F. and oth. An ontological approach to describe the SQL:2003 ob-ject-relational features, Ontologies for Software Engineering and Software Technology, Springer Berlin Heidelberg, 2006, 197-215pp.
5. Новиков Ф.А., Иванов Д.Ю. Моделирование на UML. Теория, практика, видеокурс. – СПб.: Профессиональная литература, Наука и Техника, 2010. – 640 с.: ил. + цв. Вклейки (+ 2 DVD).
6. Фаулер М., Бек К., Брант Д., Робертс Д., Апдайк У. Рефакторинг: улучшение существующего кода. – М: Символ-Плюс, 2008. – 432 с.: ил. – Парал. тит. англ.
7. Олейник П.П. Принципы организации иерархии атомарных литеральных типов объектной системы на основе РСУБД Microsoft SQL Server 2005, http://citforum.ru/database/articles/oleynick/

УДК 519.711

АНАЛИЗАТОР ДИАГРАММНЫХ ЯЗЫКОВ ДЛЯ MICROSOFT VISIO[1]

Гайнуллин Ринат Фаязович, аспирант, Ульяновский государственный технический университет, Россия, Ульяновск, r.gainullin@gmail.com
Брагин Дмитрий Геннадьевич, аспирант, Ульяновский государственный технический университет, Россия, Ульяновск, dmbragin@gmail.com

Введение

При концептуальном проектировании автоматизированных систем широкое применение находят различные графические спецификации: сети Петри, диаграммы переходов состояний, блок-схемы и другие. В сфере разработки программного обеспечения особое место заняли методологии Rational Unified Process (RUP) и Structured Analysis and Design Technique (SADT). При проектировании в соответствии с данными методологиями активно используются графические спецификации UML и IDEF.

В проектном моделировании применяются различные графические редакторы, которые можно разделить на универсальные и специализированные. Применение универсальных редакторов сводится к созданию неформализованных схем и диаграмм, чаще всего в демонстрационных целях. К таковым можно отнести Microsoft Visio и Dia. Специализированные редакторы входят в состав инструментальных средств реализации той или иной технологии. Так, Rational Suite, Visual Paradigm for UML, ArgoUML представляют собой специализированные графические редакторы для методологии UML. Примерами редакторов семейства графических спецификаций IDEF являются BPWin, ERWin, Business Studio. Общим недостатком как универсальных, так и специализированных редакторов

[1] Статья рекомендована к опубликованию в журнале "**Информационные технологии**"

является либо полное отсутствие, либо ограниченность инструментов анализа и контроля построенных диаграмм.

1. RV-грамматика

В основу разработанного анализатора положен аппарат RV – грамматик [1,3].

RV – грамматикой языка L (G) называется упорядоченная пятерка непустых множеств $G = (V, \Sigma, \Sigma, R, r_0)$, где

$V = \{v_e, e = \overline{1, L}\}$ – вспомогательный алфавит (алфавит операций над внутренней памятью);

$\Sigma = \{a_t, t = \overline{1, T}\}$ – терминальный алфавит графического языка, являющийся объединением множеств его графических объектов и связей (множество примитивов графического языка);

$\Sigma = \{a_t, t = \overline{1, T}\}$ – квазитерминальный алфавит, являющийся расширением терминального алфавита.

$R = \{r_i, i = \overline{0, I}\}$ – схема грамматики G (множество имен комплексов продукций, причем каждый комплекс r_i состоит из подмножества P_{ij} продукций $r_i = \{P_{ij}, j = \overline{1, J}\}$);

$r_0 \in R$ – аксиома RV – грамматики (имя начального комплекса продукций), $r_k \in R$ – заключительный комплекс продукций.

Продукция $P_{ij} \in r_i$ имеет вид $\widetilde{a_t} \xrightarrow{\Omega_\mu [W_\nu(\gamma_1, \ldots, \gamma_n)]} r_m$

Таблица 1. RV- грамматики диаграмм Use Case

Комплекс	Квазитерм	Комплекс–преемник	RV – отношение
r_0	R_I	r_3	$W_1(e^{t(1)})/W_3(e^{t(1)} == 0 \parallel e^{t(2)} == 0)$
	R_E	r_4	$W_1(e^{t(1)})/W_3(e^{t(1)} == 0 \parallel e^{t(2)} == 0)$
	R_G	r_5	$W_1(e^{t(1)})/W_3(e^{t(1)} == 0 \parallel e^{t(2)} == 0)$
	R_I	r_6	-
	R_E	r_7	-
	R_G	r_8	-
r_1	R_A	r_2	-
	R_G	r_5	-
r_2	C	r_0	-
r_3	C	r_0	-
r_4	C	r_0	-
r_5	C	r_0	-
	A	r_1	-
r_6	C	r_0	$W_1(e^{t(1)})/W_3(e^{t(1)} == 0 \parallel e^{t(2)} == 0)$
r_7	C	r_0	$W_1(e^{t(1)})/W_3(i^{t(1)} == 0 \parallel e^{t(2)} == 0)$
r_8	C	r_0	W1(et(2))/W3(et(1) == 0 ‖ et(2) == 0)
	A	r_1	-

В качестве внутренней памяти предлагается использовать стеки для обработки графических объектов, имеющих более одного выхода (чтобы хранить информации о связях

– метках), и эластичные ленты для обработки графических объектов, имеющих более одного входа (чтобы отмечать количество возвратов к данной вершине, а, следовательно, количество входящих дуг). Ленты позволяют считывать данные из ячеек без уничтожения их содержимого, а ячейки лент могут работать в режиме счетчика целых положительных чисел.

Приведем примеры графических грамматик спецификаций UML и IDEF.

2. Грамматика языка Use Case

Use Case-диаграммы предназначены для описания проектируемой системы (или ее части) с точки зрения взаимодействия пользователя (актора) с вариантом использования системы (прецедентом). Этот вид диаграмм решает задачу представления системы в виде, удобном для обсуждения на общем языке функциональности между заказчиком, пользователем и разработчиком. В табл. 1 представлена RV-грамматика диаграммы Use Case языка UML после минимизации.

После окончания разбора необходимо провести операцию контроля:

$* = W_2(e^{t(1)}, e^{t(2)})/W_3(e^{t(1)} <> 0 \;\&\&\; e^{t(2)} <> 0)$

В результате должны остаться пустые ленты.

Таблица 2. RV- грамматики диаграмм IDEF3

Комплекс	Квазитерм	Комплекс – преемник	RV – отношение
r_0	U	r_1	–
	L	r_2	$w1\left(inc\left(m^{i(7)}\right)\right)$
r_1	Relp	r_5	
	rel_{OF}	r_6	–
r_2	rel_p	r_5	
r_3	S	r_0	$w2(b^{4m})$
	l_{AND}	r_0	$\frac{w2(b^{1m})}{w3(m^{t(1)} == k^{t(2)})}$
	l_{OR}	r_0	$w2(b^{2m})$
	l_{XOR}	r_0	$\frac{w2(b^{3m})}{w3(m^{t(6)} == k^{t(6)})}$
	no_label	r_k	–
r_4	l_{OR}	r_0	$w2(b^{2m})$
r_5	U	r_1	–
	L	r_2	$w1\left(inc\left(m^{i(7)}\right)\right)$
	AND	r_3	[illegible]
	<u>AND</u>	r_3	$\frac{w1\left(inc\left(m^{t(1)}\right)\right)}{w3\left(k^{t(2)} > 1\right)}$
	OR	r_4	[illegible]
	<u>OR</u>	r_3	$\frac{w1\left(inc\left(m^{t(3)}\right)\right)}{w3\left(k^{t(4)} > 1\right)}$
	X O R	r_3	[illegible]
	<u>X O R</u>	r_3	$\frac{w1\left(inc\left(m^{t(5)}\right)\right)}{w3\left(k^{t(6)} > 1\right)}$
r_6	U	r_1	–
r_k			–

3. Грамматика языка IDEF3

Методология IDEF3 – это методология графического моделирования, предназначенная для описания и документирования информационных потоков в системе, взаимоотношений между процессами обработки информации и объектами, являющимися частью этих процессов. В табл. 2 представлена RV-грамматика языка IDEF3 после минимизации [2].

4. Реализация анализатора диаграммных языков

В качестве практического применения предлагаемого подхода к анализу диаграммных языков было реализовано расширение для программного продукта Microsoft Visio. Общая структура системы представлена на рисунке 1.

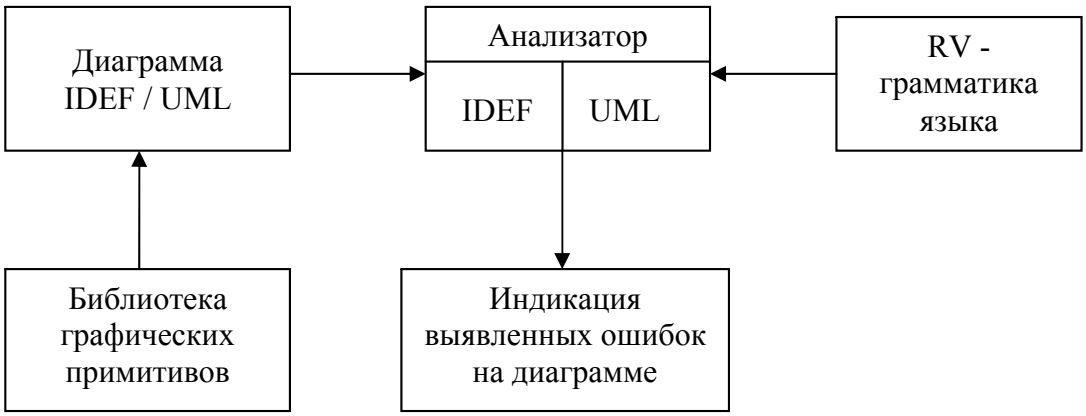


Рис. 1 – Общая структура системы анализа и контроля

Для анализа языков IDEF и UML было реализовано два отдельных анализатора, что обусловлено спецификой данных графических нотаций.

На практике используются два варианта создания расширения для Microsoft Visio:

1. Создание компонента согласно технологии Component Object Model (COM);
2. Создание встраиваемого в документ макроса на языке Visual Basic for Application (VBA).

Достоинствами первого варианта являются недостатки второго и наоборот. Так, разработка макроса на языке VBA занимает меньшее время, но ограничена по функциональности. Поэтому для разработки анализатора был выбран первый вариант, так как он обеспечивает более широкие возможности по работе как с объектами самого Visio, так и со сторонними библиотеками и самой операционной системой в целом. Компонент COM представляет собой обычную DLL библиотеку, которая регистрируется в системе и имеет свой уникальный номер. Непосредственно в компоненте согласно стандарту COM необходимо реализовать интерфейс IUnknown [3].

Информация по этому компоненту указывается в ключе HKEY_CURRENT_USER/ Software/Mirosoft/Visio/Addins/ реестра системы для того, чтобы Visio мог загрузить данные компонента. Подключение компонента в самом Microsoft Visio осуществляется из пункта меню Сервис – Настройки – Панели Инструментов.

```
<memory>
<storage id="1" type="0" />
<storage id="2" type="1" />
<storage id="3" type="1" />
<storage id="4" type="0" />
</memory>
```

Листинг 1 – XML-описание внутренней памяти

RV-грамматика графического языка хранится в формате XML. Первая секция файла грамматики описывает структуру внутренней памяти, используемой данной грамматикой. Параметры для каждого элемента памяти это уникальный идентификатор и тип элемента

памяти (0 – стек, 1 – эластичная лента). В листинге 1 приведена внутренняя память автомата, состоящая из двух стеков с идентификаторами 1, 4 и двух эластичных лент 2, 3.

Далее идёт секция с описанием продукций грамматики, каждая составляющая которой отражает строку табличной формы RV-грамматики. Параметры продукции currentState и nextState отражают соответственно текущее и следующее состояния автомата, term – соответствующий графический примитив. Далее идет список операций записи значений во внутреннюю память. Каждая операция записи содержит информацию об идентификаторе элемента памяти storageId и заносимом значении value. Для элемента памяти «эластичная лента» необходим номер ячейки key, в которую производится запись. После списка операций следует условие выполнения этих операций, параметры у которого те же, что и параметры операций записи. Для примера приведем типичную продукцию, xml описание которой представлено в листинге 2.

```
<production>
  <currentState>1</currentState>
  <nextState>2</nextState>
  <term>A</term>
 <operations>
   <operation storageId="2" key="t" value="1"/>
    <operation storageId="3" key="t" value="k"/>
    <operation storageId="4" value="t"/>
 </operations>
 <conditions>
   <condition storageId="2" key="t" value="0" />
  </conditions>
</production>
```

Листинг 2 – XML-описание продукции RV-грамматики

Алгоритм работы анализатора состоит из следующих шагов:

1. Проектировщик строит диаграмму в среде проектирования Visio.
2. С помощью разработанного расширения диаграмма преобразуется в XML-описание, которое содержит все элементы диаграммы и связи между ними. Описание не содержит информации о расположении элементов, т.к. данная информация не используется при разборе.
3. Анализатор принимает на вход XML-описание построенной диаграммы.
4. XML-описание преобразуется во внутреннее представление для работы анализатора. Внутреннее представление содержит описание диаграммы, аналогичное входному XML-файлу. Происходит дополнительная обработка входной информации необходимой для работы с RV-анализатором.
5. Последовательно, считывая элемент за элементом, анализатор производит анализ и контроль диаграммы.
6. По результатам анализа и контроля формируется список ошибок.
7. Список преобразуется в XML и возвращается в среду проектирования.
8. По полученному списку в Visio отмечаются типы и местоположения синтаксических и семантических ошибок, обнаруженные анализатором.

Выводы

Развитием предлагаемой системы является создание единого универсального анализатора диаграммных языков, предоставляющего возможность добавления определенной графической нотации, её грамматики и последующего анализа. Также ведутся работы по расширению предлагаемого подхода на другие программные средства и платформы.

Литература

1. Афанасьев А.Н. и др. Контроль информации в системах автоматизации проектирования. // Саратов: Изд-во Саратовского университета, 1985.- 136 с

2. Брагин Д.Г. Анализ IDEF3 диаграмм на основе автоматных графических грамматик // Сборник научных трудов «Информатика, моделирование, автоматизация проектирования». – Ульяновск: УлГТУ, 2011.– С. 67-74.
3. Шаров О.Г., Афанасьев А.Н. Синтаксически-ориентированная реализации графических языков на основе автоматных графических грамматик // Программирование. – 2005. – № 5. – С. 56-66.

УДК 004.04

ИНСТРУМЕНТАРИЙ ОПТИМИЗАЦИИ РАБОТЫ СИСТЕМЫ УПРАВЛЕНИЯ ОБЪЕКТНО-РЕЛЯЦИОННОЙ БАЗЫ ДАННЫХ

Микляев Иван Александрович, к.ф.-м. н., доцент, Филиал «СЕВМАШВТУЗ», Санкт-Петербургский государственный морской технический университет, Россия, Северодвинск, ivanmia1@rambler.ru
Черткова Ольга Владимировна, старший преподаватель, Филиал «СЕВМАШВТУЗ», Санкт-Петербургский государственный морской технический университет, Россия, Северодвинск, chertkova-o@mail.ru

Введение

Испокон веков человечество занято познанием окружающего мира и себя как его части. Парадигмы сменяют друг друга в попытках смоделировать окружающую реальность. Однако идея о том, что мир пронизан синергией – энергией взаимодействия, нашла удивительный отклик во всех сферах современного мира. Феноменом этого отклика стало представление о нашей жизни в целом в виде бесконечной совокупности взаимосвязанных и взаимозависимых открытых и закрытых систем. Элементы систем носят голографический или фрактальный характер, обладая вместе с тем уникальными свойствами, приводящими сами элементы и системы в целом к процессу постоянных изменений [1] .

И вся эта "реальность, данная нам в ощущениях" требует создания соответствующего информационного пространства – синергетического. Описываемые базами данных (БД) предметные области и бизнес-процессы, как и все живые системы, обладают свойствами жизнестойкости, функциональности и адаптивности. Информационный образ каждого объекта, занесённого в базу, должен быть синергетическим – отражающим зависимость значений характеристик объекта от среды взаимодействия. Естественно, программные средства и структуры представления данных в БД должны обладать теми же свойствами, чтобы адекватно отображать многогранный окружающий мир.

Стандартный реляционный подход к организации данных делает ставку на максимально эффективное манипулирование данными, для этого применяет дезинтеграцию данных до их полной однородности. А поскольку реальные бизнес-среды практически никогда не обладают однородностью, это приводит к чрезмерному абстрагированию от предметной области [2], и сложностям с описанием синергетических характеристик [3].

Напротив, стандартный объектно-ориентированный подход к организации данных предполагает каждый объект реального мира описывать индивидуально, всей совокупностью его параметров, что является более естественным для сложных систем. Безусловно, это позволяет адекватно отобразить структуру и состав предметной области, многообразие синергитических характеристик и облегчает проектирование.

1. Примеры решения задач синергетической природы

Организовать объектно-ориентированный подход в условиях реляционной базы данных можно с помощью выявления фрагментов баз данных, содержащих объекты с варьируемым числом параметров, и вынесения самих объектов и параметров-атрибутов в отдельные справочники, что сделает информационную систему более функционально и

устойчивой к изменениями бизнес-среды. Отношение «объект – параметр объекта» позволяет корректировать список параметров даже на поздних стадиях проектирования.

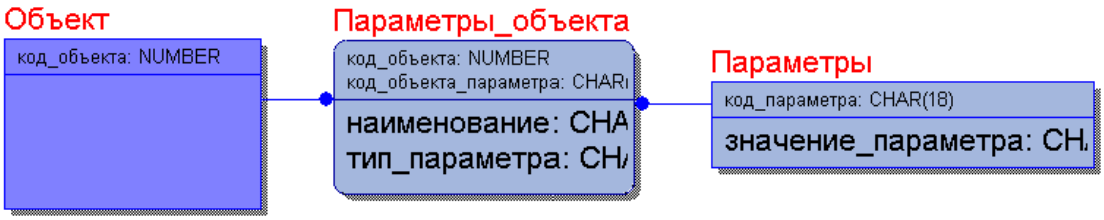


Рис. 1 – Отношение объект – параметр объекта.

Такие фрагменты можно выделить во многих задачах. Например, при разработке информационной системы учета лекарственного обеспечения пациентов с хроническими заболеваниями в [4], каждое заболевание характеризуется обширным перечнем параметров. Некоторые параметры уникальны и характерны только для одного заболевания, другие присущи нескольким заболеваниям. И у конкретного пациента эти параметры – симптомы – могут присутствовать, отсутствовать и быть выражены в различных степенях. Аналогичная ситуация возникает и при проектировании информационной системы учета компьютерной техники, так как компьютеры могут быть представлены как объекты с иерархической структурой, имеющие различное количество комплектующих. А также в риэлторских системах и системах учета наследственных дел, в системах расчета заработной платы [4].

2. Модель универсальной базы данных

Спектр подобных задач привёл к идее универсальной базы данных (УБД), которая способна без изменения структуры содержать в себе информацию любой предметной области [5].

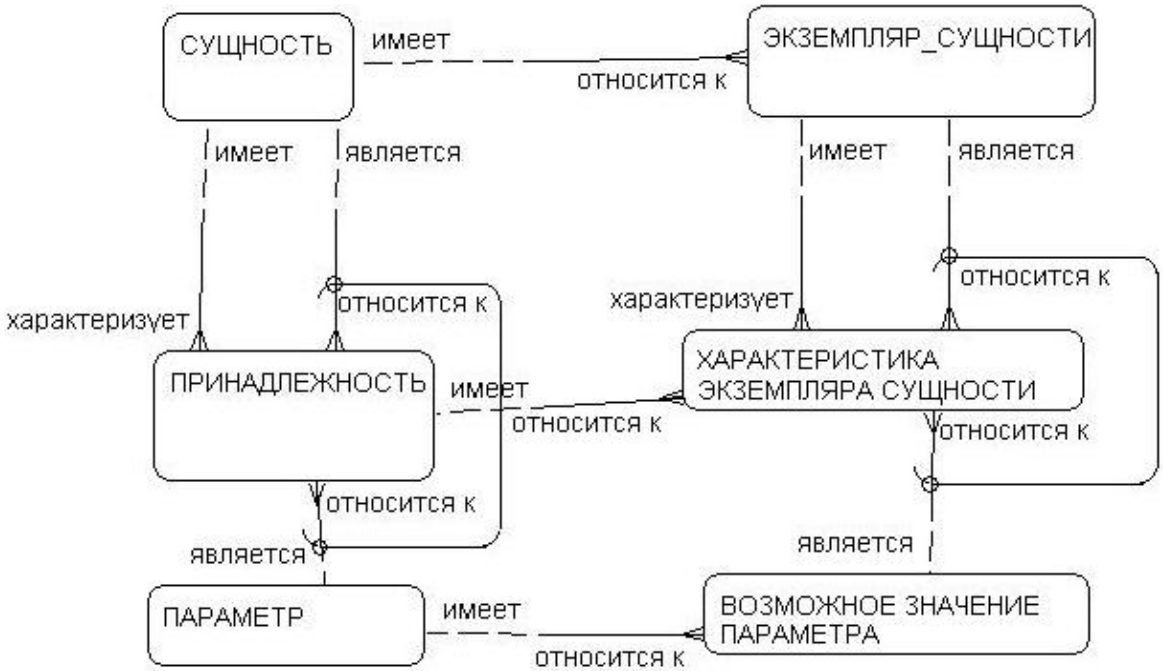


Рис. 2 – Логическая модель универсальной базы данных

3. Проблемы объектно-ориентированного подхода в базах данных

Недостатками объектно-ориентированного подхода являются отсутствие универсальных моделей данных и стандартов, отсутствие поддержки представлений, неудобство обработки данных и низкая скорость выполнения запросов [2]. То есть сложность оперирования данными в виде объектов сводит на нет неоспоримые преимущества их использования.

Центральными проблемами, вызывающими эту сложность, являются:

1. Значения параметров объектов одной и той же природы могут быть совершенно разнотипными;
2. Хаотичное расположение параметров затрудняет оперирование информацией.

4. Варианты решения проблемы разнотипности

В рамках реляционной БД предложено два решения первой проблемы. Одно из решений изложено в статье Банникова А.Н. [6] и заключается в методике создания справочников типов. Разложить данные в таблицы по типам, создать справочники типов и свойств, и в поле значений таблицы параметров объекта указывать ссылку на соответствующую таблицу справочника определённого типа и на нужное значение в ней. Параллельно оптимизировать структуру SQL запросов под такое строение справочников.

Другим решением может служить использование унифицированного и единого типа данных [7]. Логично выбрать единицей измерения унифицированного типа – массив байт, так как он является естественной размерностью для виртуального информационного пространства. Такой универсальный тип создан и применяется в матричной универсальной объектно-реляционной базе данных (МУОРБД).

5. МУОРБД

МУОРБД является матричным представлением УБД[8]. Матричная структура сохраняет все преимущества реляционных баз данных и позволяет работать с информацией, используя понятия наследования, агрегации, возможен полиморфизм и инкапсуляция. Структура ядра МУОРБД представляет собой семимерный массив, и имеет вид [8]:

```
ARRUDB : {массив УБД}
array of {БД}
array of {Сущность}
array of {Экземпляр сущности}
array of {Информационная единица -группа}
array of {№ п\п элемента группы - параметра экземпляра сущности}
array of {Атрибуты параметра - ссылка, значение, дата с, дата по,
родительский элемент }
array of byte {Массив параметра}
```

Первое измерение – для работы распределённой базы данных. Второе измерение «Сущность» – это таблицы объектов БД, в том числе таблицы метаданных МУОРБД. Третье измерение Экземпляр сущности – непосредственно сам объект, четвёртое измерение является измерением групп, в которые объединены значения параметров по смысловому содержанию (многострочная единица информации), пятый индекс – элемент группы параметра экземпляра сущности, шестой – статический массив атрибутов значения параметра, содержащий помимо самого параметра и его значения временные границы актуальности значения и древовидность структуры единицы информации. Седьмое измерение – массив параметра – байты значений атрибутов параметра (универсальный тип данных).

6. Решение проблемы разнотипности в МУОРБД

Универсальный тип данных реализован на основе понижения кратности системы счисления и обеспечивает упрощение и оптимизацию размещения информации МУОРБД в оперативной памяти [6]. Это позволяет отказаться от определения типов данных атрибутов на физическом уровне, и выводит задачу определения типа информации на уровень программирования, где тип задаётся программистом для пользователя, и ограничения выставляет из условий поставленной задачи, а не из требований СУБД. Этот механизм снимает ограничения на ввод информации в поля атрибутов и даёт разработчикам информационных систем столь необходимую жизнестойкость и адаптивность БД к изменяющимся требованиям.

В седьмом измерении МУОРБД хранятся в основном ссылки на справочники внесённых единиц информации, так как вся тяжёлая информация из базы убрана и разделена на справочники по типу символов, содержащихся в значениях, исходя из специфики работы алгоритма на основе аппарата понижения кратности системы счисления.

7. Решение проблемы хаоса

Решение второй проблемы в рамках МУОРБД потребовало создания дополнительных описательных средств: инструмента поддержки табличного вида сущности (ИПТВС), индексного пространства основного массива данных (ИПОМД), индексного пространств справочников (ИПС) и пространства целостности ссылок (ПЦС).

ИПТВС – решает главную проблему хаотичности расположения атрибутов и представляет собой привычную реляционную таблицу, в каждой ячейке которой массив ссылкок на строку таблицы параметров. Атрибутам – синергетическим, в частности, имеющим несколько параллельных значений, – соответсвуют кратные ссылки. По запросу пользователя выводится все значения параметра через разделитель, либо значения соответвтвующие запрашиваемому внешнему условию. Например, для индивида может выводиться фамилия, действущая на заданный период времени, или весь их список.

ИПОМД представляет собой семимерный массив, в котором размещены отсортированные указатели на строки таблиц в нужном порядке по каждому параметру. Структура массива ИПОМД имеет вид:

```
IASTSpr : {массив ИПОМД }
array of {БД}
array of { № Сущности}
array of { № Параметра }
array of { № Префикса }
array of {№  Группы первого символа}
array of {№ п/п в группе первого символа}
array of byte { адрес }
```

Первое измерение – номер базы даных, второе измерение – номер таблицы отражающей сущность, третье – номер параметра. Дальнейшие измерения связаны с реализацией группового индексирования.

Значения параметра разделены на 256 групп – соответственно количеству возможных значений у байта. Такое разделение даёт однозначный положительный эффект для поиска, только при равномерном распределении первых символов в значениях параметра, что редко соответсвует действительности. Часто встречающаяся ситуация, когда существенное количество значений параметра начинаются с одного или нескольких одинаковых символов и различаются только несколькими последними (наименование документа, чертежа, множество однокоренных фамилий), смоделирована с помощью системы префексов. Начиная с определённого количества строк, соответствующего группе первого символа, этот символ выносится в отдельный массив из префиксов всех параметров, затем строки распределются в группы соответсвенно второму символу и так далее. Структура массива префиксов имеет вид:

```
PrASTSpr : {массив префиксов }
array of {БД}
array of { Сущность}
array of { Параметр }
array of { № Префикса }
array of byte { массив префикса }
```

Строки, имеющие несколько значений одного параметра, упоминаются в различных группах соответствующее число раз. Поэтому четвёртое измерение ИПОМД – номер префикса, – если он не выделен, то это 0. Пятое – номер кода первого символа (от 0 до 255), наконец, шестое – порядковай номер в группе первого символа.

ИПС – представляет собой шестимерный массив, в котором размещены отсортированные указатели на вынесенные в справочники значения. Структура массива ИПС имеет вид:

```
IAD: {массив ИПС}
array of {БД}
array of {№ Справочника}
array of {№ Префикса}
array of {№  Группы первого символа}
array of {№ п/п в группе первого символа}
array of byte { Массив значения }
```

Для построения системы группового индексирования информации аналогично введён массив префиксов для справочников:

```
PrAD: {массив ИПС}
array of {БД}
array of {№ Справочника}
array of {№ Префикса}
array of {№  Группы первого символа}
array of {№ п/п в группе первого символа}
array of byte { Массив значения }
```

ПЦС воспроизводит структуру основного массива данных с тем отличием что на шестом уровне не атрибуты значения параметра, а список ссылающихся на это значение элементов базы данных, на последнем уровне сами адреса ссылки.

Структура массива ПЦС имеет вид:

```
RIASTSpr: {массив ПЦС }
array of {БД}
array of {Сущность}
array of {Экземпляр сущности}
array of {Информационная единица -группа}
array of {Элемент группы – параметр экземпляра сущности}
array of {№  п\п в списке ссылок }
array of byte {ссылка}
```

Все индексные пространства используют систему группового индексирования информации.

Выводы

Применение универсального типа данных, ИПТВС, ИПОМД, ИПС и ПЦС компенсируют основные отрицательные отличия объектного подхода от реляционного. А возможность использования группового индексирования и вовсе даёт превосходство перед ним.

В дальнейшем следует задуматься над формализацией объектно-реляционных моделей. Расширение структуры, появление новых возможностей взаимосвязи объектов, объектно-ориентированные свойства, синергетичность атрибутов объектов – всё это делает недостаточным стандартные нотации для реляционных СУБД, и требует расширения их либо замены.

Литература

1. Хакен Г., Информация и самоорганизация: Макроскопический подход к сложным системам. М.: Мир, 1991.-240 с.
2. Конноли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е издание.:Пер. с англ. / Т. Конноли, К. Бегг. М.: Издательский дом «Вильямс», 2003. С.1440
4. Микляев И.А., Черткова О.В. Синергетическое информационное пространство МУОРБД // // Объектные системы – 2011 (Зимняя сессия): материалы V Международной научно-практической конференции (Ростов-на-Дону, 10-12 декабря 2011 г.) / Под общ. ред. П.П. Олейника. – Ростов-на-Дону: ШИ ЮРГТУ (НПИ), 2011. – С. 67-72.

5. И.А. Микляев, А.Н. Ундозерова, М.В. Кудаева Типовой подход к проектированию объектов баз данных с варьируемым числом параметров Вестники СПбО АИН Выпуск №7, 2010, с.303-312.
1. 5. Микляев И.А., Ундозерова А.Н., Кудаева М.В., Свидетельство ОФЕРНиО (Объединённого фонда электронных ресурсов «Наука и образование») №15041 "Универсальная база данных", 2009.
6. Банников Н.А. Объектно-ориентированные базы данных // http://www.stikriz.narod.ru/art/oobd.htm
7. Микляев И.А., Свидетельство ОФЕРНиО №15405 (Объединённого фонда электронных ресурсов «Наука и образование») "Универсальный тип данных баз данных", 2010.
8. Микляев И.А, Свидетельство ОФЕРНиО (Объединённого фонда электронных ресурсов «Наука и образование») №15175 "Матричное представление универсальной базы данных", 2009.

УДК 004.89

ОБ АГЕНТНО-ОРИЕНТИРОВАННЫХ СИСТЕМАХ И МНОГОАГЕНТНЫХ БАНКАХ ЗНАНИЙ

Зайцев Евгений Игоревич, к.т.н., доцент, Московский государственный университет приборостроения и информатики, Россия, Москва, zei@tsinet.ru

Агентно-ориентированная технология позволяет создавать распределенные интеллектуальные системы, в которых для решения сложных плохоформализуемых задач вместо одного интеллектуального решателя используется самоорганизующаяся сеть программных агентов, формирующих частные решения, взаимодействующих друг с другом, а также координирующих коллективное поведение. Понятие программного агента (software agent) трактуется по-разному. Иногда программные агенты определяют просто как автономные процессы, которые действуют от имени пользователя в среде, в которой выполняются другие процессы. При этом агенты рассматриваются как сущности, которые могут воспринимать окружающую среду посредством рецепторов и взаимодействовать с ней [1]. Выделяют такие свойства программных агентов, как интерактивность или общественное поведение (social ability) (т.е. способность функционировать в сообществе агентов, инициируя взаимодействия и обмениваясь сообщениями с помощью некоторого языка коммуникаций); реактивность (способность воспринимать окружающую среду и своевременно реагировать на события недетеминированным образом); проактивность и целеустремленность (способность действовать в упреждающей манере, в частности, генерировать новые цели и действовать рационально для их достижения).

В определении агента, которое дает Фонд интеллектуальных физических агентов (Foundation for Intelligent Physical Agents), являющийся комитетом IEEE Computer Society (http://fipa.org), агент – это главный исполнитель в домене, обладающий одной или несколькими сервисными возможностями, образующими единую и комплексную модель выполнения, которая может включать доступ к внешнему программному обеспечению, пользователям и средствам связи. Отмечается, что главной отличительной чертой программных агентов являются их интеллектуальность (intelligent) и мобильность (mobility). С общепринятой точки зрения объектно-ориентированного программирования первое свойство проявляется в том, что агент содержит не только типичные для объектных классов методы и атрибуты (члены-данные класса), но также когнитивные структуры данных (CDS, Cognitive Data Structures) и методы, реализующие логические выводы, а также механизмы самообучения и адаптации.

Второе свойство (мобильность) агентов, связанное с их способностью мигрировать по сети в поисках необходимой информации или сервисов, позволяет динамически перераспределять вычислительную нагрузку в зависимости от состояния сети, а также

обеспечивает интероперабельность между различными существующими и разрабатываемыми системами. Интероперабельность достигается при стандартизации таких аспектов, как передача агентов и служебных (используемых агентом) классов между агентными системами, а также управление агентами. Технология мобильных агентов может поддерживать как слабую (weak mobility), так и сильную (strong mobility) модели мобильности. В случае слабой мобильности на удаленный узел переносится только сегмент кода (набор инструкций), возможно, вместе с некоторыми данными инициализации. Характерной чертой слабой мобильности является то, что перенесённый код всегда запускается из своего исходного состояния, как это происходит, например, с Java-апплетами (applets). В случае сильной мобильности вместе с сегментом кода переносится также сегмент исполнения, что позволяет работающему процессу после приостановки и перенесения на другую машину продолжить его выполнение с того места, на котором этот процесс был приостановлен. Примером агентов с сильной мобильностью являются Java-аглеты (aglets), которые пересылаются из одного контекста в другой по протоколу ATP (Agent Transfer Protocol) прикладного уровня, не зависящему от платформы и использующему унифицированный указатель ресурса (URL, Uniform Resource Locator) для определения местоположения агентов и серверов.

Фонд FIPA унифицирует архитектуру платформ агентов, а также необходимые для управления агентами операции и способы взаимодействия агентов с программным обеспечением, которое не использует агентную технологию. Согласно FIPA система агентов содержит подсистему управления агентами (Agent Management System), службу каталога (Directory Facilitator), канал связи агентов (Agent Communication Channel) и менеджер безопасности платформы агентов (Agent Platform Security Manager). Данные компоненты включают сервисы по созданию, удалению, деактивации, возобновлению работы и миграции агентов, службы маршрутизации сообщений и управления жизненным циклом, службы белых страниц (с информацией о соответствии между глобально уникальными именами агентов и адресами локального транспорта, используемого платформой) и желтых страниц службы каталога (с описанием агентов и услуг, которые они предоставляют), а также сервисы по осуществлению политики безопасности на транспортном уровне.

Другой стандарт для реализации систем мобильных агентов и обеспечения интероперабельности между различными архитектурами – MASIF (Mobile Agent System Interoperability Facilities) ассоциации Object Management Group (http://www.omg.org), также унифицирует синтаксис и правила выполнения операций, связанных с созданием, удалением, перемещением и идентификацией агентов, приостановкой и возобновлением их работы, получением агентом информации о типе платформы. Представляя собой надстройку над стандартом CORBA (Common Object Request Broker Architecture), стандарт MASIF позволяет объединить традиционную клиент-серверную парадигму и технологию мобильных агентов. MASIF поддерживает некоторые сервисы CORBA, но не требует обязательного их использования.

В большинстве случаев вместо модели обмена сообщениями транспортного уровня в агентно-ориентированных системах используются разработанные FIPA коммуникационные протоколы прикладного уровня, называемые языком взаимодействия агентов ACL (Agent Communication Language), которые могут применяться для взаимодействия агентов разных систем поверх брокеров объектных запросов ORB (Object Request Broker) стандарта CORBA. В ACL присутствует жесткое разделение между целью сообщения и его содержанием: посылка сообщения обычно предполагает конкретную реакцию получателя на основании исключительно цели сообщения. Часть языка FIPA ACL, предназначенная для представления содержания посланий интеллектуальных агентов, называется семантическим языком FIPA SL (Semantic Language).

Отличительной особенностью агентов является то, что в их модели присутствуют ментальные свойства (интенциональные характеристики), такие как убеждения, желания и намерения (модель BDI – Belief, Desires, Intentions – минимальное интенциональное

описание агента), которые направляют деятельность агентов и могут меняться в процессе функционирования. В архитектуре BDI, которая во многом строится по аналогии с практическими рассуждениями человека, процесс принятия решений выполняется с использованием механизмов вывода на основе ментальных понятий агента, представленных некоторыми структурами знаний. Поскольку эти ментальные понятия связывают мотивационные и поведенческие понятия, то эта архитектура называется также архитектурой, управляемой целями (goal – oriented). Формальные модели знаний, убеждений, а также соответствующие механизмы рассуждений строятся на основе логических языков, которые включают в себя расширения модальных и темпоральных логик с семантикой возможных миров [2, 3, 4].

Желания агента представляют собой состояния (ситуации) в пространстве состояний или пространстве задач, достижение которых является для агента желательным. В противоположность убеждениям (изменяющимся во времени знаниям агента о внешней среде и о других агентах, которые могут оказаться ложными), желания рассматриваются не с позиций их истинностного значения, а в контексте удовлетворения или выполнения. К исполнению желания агента ведёт реализация его намерений – того, что агент должен (собирается) сделать в силу своих обязательств перед другими агентами и/или своих желаний. Если желания выполнимы, значит, агент в состоянии реализовать свои намерения и выполнить необходимые для достижения цели действия. Цели представляются конкретным множеством конечных и промежуточных состояний, достижение которых агент считает реализацией своих намерений. Для удовлетворения желаний агент может, не меняя своих намерений и убеждений в том, что действия, которые он намерен совершить, возможны, делать попытки соответствующим образом воздействовать на внешнее окружение.

Не обладая полным знанием о внешнем окружении и имея лишь частичное представление о проблеме, агенты осуществляют неточные, правдоподобные рассуждения, которые необходимо подвергать пересмотру (belief revision) при получении агентом дополнительной информации, несовместимой с полученными ранее заключениями, а также при изменении модели мира в результате обновления убеждений агентов (belief update). Убеждения становятся знаниями только при подтверждении (обосновании) новыми фактами или объяснениями. Знания программных агентов о моделируемых в их базах предметных областях могут быть представлены с помощью традиционной для систем искусственного интеллекта модели – с помощью специальных фреймов-прототипов, которые используются для описания объектов предметной области и их состояний, действий и событий, а также процессов, под которыми понимаются упорядоченные совокупности событий и/или других процессов, реализуемых в целях решения тех или иных задач.

В отличие от монотонной логики, в которой добавление в систему новой информации не приводит к уменьшению множества истинных утверждений и все высказывания, которые следуют из базы знаний K, по-прежнему следуют из неё после добавления множества новых посылок F (то есть, если $K \vdash B$, то $(K \cup F) \vdash B$), немонотонные рассуждения не являются в классическом смысле общезначимыми и если утверждение B выводимо из K, то существует модель для $K \cup F$, не подтверждающая B. Таким образом, добавление в базу знаний новой информации может потребовать пересмотра полученных ранее результатов. Немонотонные логики, такие как логика умолчания (default logic), автоэпистемическая (autoepistemic logic), немонотонная модальная логика (modal nonmonotonic logic), косвенное описание (circumscription), позволяют решить проблему отсутствия у агентов определенных знаний, при этом нет необходимости придерживаться предположения о замкнутом мире (closed-world assumption), согласно которому высказывания, истинность которых не может быть доказана, считаются ложными, а имеющаяся в базе знаний информация полной.

Формализовать интенциональные характеристики агента, а также ввести в механизм его рассуждений элемент немонотонности можно, используя модальные логики, которые могут интерпретироваться в различных мирах, тогда как классическая логика интерпретируема только в одном мире [5]. Однако модальные логики не позволяют

адекватно представить неточные (по содержанию) и неопределенные (в смысле уверенности в истинности тех или иных высказываний) знания. Поэтому при моделировании агентов важное значение придаётся нечётким (fuzzy) немонотонным логикам, в которых используются специальные механизмы, позволяющие когнитивным агентам оперировать нечеткими понятиями и реализовывать прямые (fuzzy forward-chaining reasoning) либо обратные (fuzzy backward-chaining reasoning) нечеткие выводы. При прямом нечетком выводе отдельные факты базы знаний когнитивных агентов преобразуются в конкретные значения функций принадлежности антецедентов нечетких продукций и находятся значения функций принадлежности заключений по каждому из нечетких правил. Процесс обратного нечеткого вывода заключается в подстановке отдельных значений функций принадлежности заключений и нахождении функций принадлежности условий, которые принимаются в качестве очередных подцелей и далее могут использоваться как функции принадлежности новых заключений.

Исходная система нечетких правил, используемая когнитивным агентом в процессе логического вывода, может оказаться неполной или противоречивой, а субъективно выбранные экспертом вид и параметры функций принадлежности, описывающих входные и выходные переменные системы, могут не вполне адекватно отражать действительность. Для устранения этих недостатков в модель когнитивного агента может дополнительно включаться адаптивная нечеткая продукционная сеть, в которой нечеткая продукционная модель интегрирована с технологией нейронных сетей. В агентно-ориентированных интеллектуальных системах необходима также возможность проведения адекватных рассуждений о времени в условиях, когда имеющееся для этого время само является значимым ресурсом, расход которого должен учитываться рассуждающим агентом. Для формализации немонотонных рассуждений о времени может быть использован темпоральный (временной) вариант логики минимальной веры и отрицания по умолчанию (MBNF, Minimal Belief and Negation as Failure) [6].

В агентно-ориентированных интеллектуальных системах знания агентов распределены и независимы друг от друга, поэтому проблема обеспечения непротиворечивости базы знаний уступает место задачам обеспечения кооперации и коммуникации агентов. Кооперация предполагает не только сотрудничество агентов при решении задач, но также координацию их совместных действий и возможность разрешения возникающих в процессе работы конфликтов. Коммуникация агентов является не столько процессом передачи информации от одного агента к другому, сколько процессом, в котором агенты преследуют свои цели.

Адекватные реакции агентов на возможные в окружающей среде ситуации реализуются благодаря обучению входящей в модель агента нейронной сети, которая поддерживает механизм конкуренции между желаниями агента и позволяет при предъявлении агенту входного вектора (ситуации) возбуждать наиболее точно соответствующие ему действия. Реактивный агент также может интегрировать методы нечетких и нейросетевых технологий, вводя нечеткость в структуру нейронной сети путем “размывания” значений обучающих примеров, входных и выходных данных, использования нечетких весов связей, замены выполняемых нейронами стандартных функций на операции, применяемые в теории нечетких множеств. Таким образом, агентно-ориентированный подход комбинирует в себе подходы, используемые в реактивных и содержащих символьную модель мира BDI-архитектурах, что предполагает использование двух основных типов программных агентов (когнитивных и реактивных), интегрирующих коннекционистский и семиотический подходы в искусственном интеллекте [7,8].

Исследования в области агентно-ориентированных систем делят на такие основные направления, как теория агентов, методы кооперации агентов, архитектура агентов и многоагентных систем, технология и программные средства их разработки. В теории агентов рассматриваются формализмы и математические методы для описания рассуждений об агентах и для выражения желаемых свойств агентов. Исследуются методы кооперации

агентов, которые определяют организацию кооперативного поведения агентов в процессе совместного решения задач. В исследованиях по архитектуре агентов и многоагентных систем рассматривается, как построить вычислительную систему, удовлетворяющую тем или иным свойствам, которые выражены средствами теории агентов. Особое место занимают исследования, связанные с разработкой инструментальных средств поддержки технологии разработки агентно-ориентированных систем.

Для реализации многоагентных систем (МАС, Multi-agent systems – MAS) на основе спецификаций стандартов FIPA используют такие инструментальные средства, как JADE (Telecom Italia Lab), Agent Factory (PRISM Laboratory), FIPA-OS (Emorphia), JACK Intelligent Agents (AOS Group), Zeus (British Telecommunication), Agent Development Kit (Tryllian BV), April Agent Platform (Agent Research Group), Comtec Agent Platform (FIPA), Java Agent Services API, Grasshopper (IKV technologies AG). Для реализации МАС на основе спецификаций стандарта MASIF используют такие среды, как Aglets SDK (IBM), Odyssey (GenMagic), D'Agents (Dartmouth college), Grasshopper (IKV++). Многие из этих систем позиционируются, как проекты с открытым исходным кодом (JADE, ZEUS, FIPA-OS, AgentFactory, Tryllian ADK).

Разрабаботка многоагентных систем возможна с использованием современных универсальных интегрированных сред разработки (Integrated Development Environment – IDE), позволяющих подключать модули специального назначения. Например, такие IDE, как мультиплатформенные Eclipse (Eclipse Foundation) или NetBeans (NetBeans Community) позволяют реализовывать как прикладные предметно-ориентированные МАС, так и специализированные инструментальные средства их разработки. Поскольку Eclipse и NetBeans состоят из плагинов (PDE), у разработчиков инструментальных средств имеется возможность предложить свои расширения к этим IDE и предоставить пользователям последовательную и цельную интегрированную среду разработки. Создаваемая агентная платформа может непосредственно опираться на операционную систему, либо использовать промежуточное программное обеспечение, типа Java Virtual Machine (JVM) исполняющей системы Java Runtime Environment (JRE) или Common Language Runtime (CLR) программной платформы .NET Framework.

Высокий уровень абстракции, используемый при агентно-ориентированном подходе, поддерживаемый агентными кросс-платформенными технологиями, позволяет использовать концепцию программных агентов при разработке таких прикладных МАС, как многоагентные банки знаний (МБЗ) [9]. Многоагентные банки знаний представляют собой распределенные интеллектуальные информационные системы учебного назначения, которые интегрируют функции интеллектуальных учебных сред (ILE, Intelligent Learning Environments) и интеллектуальных обучающих систем (ITS, Intelligent Tutoring System). МБЗ включают общие и специальные знания о предметной области, о процессе обучения и модели обучаемого, ассоциируя их с реактивными и когнитивными программными агентами, которые реализуют процедуры обработки этих знаний, формируют и выдают ответы на запросы пользователей, осуществляют адаптивное обучение.

Ответы на запросы пользователей формируются когнитивными агентами МБЗ в результате спецификации свойств сущностей (событий и их субъектов), вычисления каузальных, временных и других отношений на множестве сущностей, а также в результате планирования решения задач. При этом вычисление отношений и синтез плана действий для решения некоторой задачи происходят не только в процессе выполнения когнитивным агентом продукционных, редукционных или трансформационных правил, но также в процессе его переговоров с другими агентами. Адекватные реакции агентов на возможные в окружающей среде ситуации реализуются благодаря обучению искусственных нейронных сетей реактивных агентов, которые при предъявлении агентам входного вектора (ситуации) активизируют наиболее точно соответствующие ему действия.

Программные агенты МБЗ создаются как компоненты (Component-Based Design) в виде специализированных категорий классов (включающих методы, свойства, события), в

которых интегрированы механизмы логического вывода с технологией искусственных нейронных сетей. Реактивные и когнитивные агенты многоагентного банка знаний сочетают четкие методы и модели поиска решений и адаптации с нечеткими. Нечеткость включается как в элементы структуры нейронных сетей реактивных агентов, так и в механизмы логического вывода когнитивных агентов. Программные агенты могут осуществлять немонотонный и нечеткий вывод, а также использовать нечеткие запросы (fuzzy queries) к базе знаний. Нечеткие запросы позволяют расширить область поиска в соответствии с изначально заданными ограничениями. Применяя нечеткие модификаторы и задавая соответствующие диапазоны числовых величин, можно получить информацию, в соответствующем приближении удовлетворяющую заданным критериям. Для реализации нечетких запросов строятся нечеткие отношения с использованием соответствующих функций принадлежности, на основе которых формулируются и реализуются запросы в нечетком виде.

Создание многоагентных банков знаний является сложной задачей, которая требует от разработчиков опыта проектирования и реализации как концептуальных, так и технических решений в таких областях, как представление и обработка знаний, сетевые коммуникации и протоколы взаимодействия, нейронные сети и нечеткая логика. Проектирование и реализация МБЗ значительно упрощается при использовании специальных инструментальных средств, поддерживающих процесс разработки многоагентных банков знаний. К таким средствам относится проблемно-ориентированная инструментальная среда AgentITS (МГУПИ), включающая такие группы программных средств, как среда выполнения МАС (агентная платформа) и инструментальная среда разработки МБЗ. С использованием AgentITS возможно осуществить разработку предметно-ориентированного многоагентного статического банка знаний (МСБЗ) [10], предназначенного для параллельного изучения императивных, функциональных, логических и объектно-ориентированных языков программирования. В МСБЗ группы агентов, реализуя запросно-ответное отношение между языком запросов и языком ответов, предоставляют пользователю возможность систематически получать сведения не только о конструкциях конкретных языков, но и связанных с ними стилях и методах программирования.

Литература

1. Рассел С., Норвиг П. Искусственный интеллект: современный подход. –М.: Вильямс, 2006. - 1408с.
2. Тарасов В.Б. От многоагентных систем к интеллектуальным организациям. –М.: Эдиториал УРСС, 2002. -352с.
3. Городецкий В.И., Грушинский М.С., Хабалов А.В. Многоагентные системы / Новости искусственного интеллекта, №2, 1998.
4. Городецкий В.И., Карсаев О.В., Конюший В.Г., Самойлов В.В., Хабалов А.В. Среда разработки многоагентных приложений MASDK // Информационные технологии и вычислительные системы, 2003, №12. С.26-41.
5. Люггер Д.Ф. Искусственный интеллект: стратегии и методы решения сложных проблем. –М.: Вильямс, 2003., 864с.
6. Виньков М.М., Фоминых И.Б. Немонотонные рассуждения в динамических интеллектуальных системах // Новости искусственного интеллекта, №4, 2005.
7. Зайцев Е.И. Методология представления и обработки знаний в распределенных интеллектуальных информационных системах. // Автоматизация и современные технологии. 2008, №1, С.29-34.
8. Зайцев Е.И. Агентно-ориентированная технология разработки распределенных интеллектуальных систем. // Материалы III Международной научно-практической конференции “Объектные системы – 2011”. – Ростов-на-Дону, 2011.
9. Зайцев Е.И. О концепции многоагентных банков знаний, как интеллектуальных обучающих системах // Материалы 16-й международной конференции “Современное образование: содержание, технологии, качество”. В 2-х.т. – СПпб.:Изд-во СПбГЭТУ “ЛЭТИ”. 2010. Т.2. С. 36 – 38.

10. Миронов А.С., Зайцев Е.И. О концепции многоагентных учебных сред, называемых многоагентными статическими банками знаний. // Материалы XVII Международной конференции “Современное образование: содержание, технологии, качество”. В 2-х.т. – СПпб.:Изд-во СПбГЭТУ “ЛЭТИ”. 2011. Т.2. С. 155 – 156.

УДК 368.914

ОБЪЕКТНОЕ МОДЕЛИРОВАНИЕ ПЕНСИОННОГО ОБЕСПЕЧЕНИЯ

Базилевич Ксения Алексеевна, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Украина, Харьков, ksenia.bazilevich@gmail.com
Мазорчук Мария Сергеевна, к.т.н., доцент, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Украина, Харьков, mazorchuk_mary@inbox.ru

Введение

Тема пенсионного страхования для нашей страны является актуальной, так как существующая пенсионная система не в состоянии обеспечить достойный уровень жизни граждан после окончания трудовой деятельности. В отличие от банковских пенсионных программ и негосударственных пенсионных фондов, страховые компании обеспечивают защиту от целого ряда рисков, а не только накопление средств [1,2].

Основной чертой современного состояния этой области является отсутствие инструментальных средств, позволяющий спроектировать информационную систему, способную на достойном уровне решить множество задач пенсионного страхования и максимально автоматизировать данный процесс.

Поэтому основной задачей данной работы является разработка инструментальных средств моделирования пенсионных тарифов. Применение данной разработки позволит проанализировать основные модели пенсионного фонда и получить на практике конкретные результаты. Преимущество такого подхода заключается в структуризации различных пенсионных схем и создании единой информационной модели, которая будет отображать все характерные особенности этого вида страхования.

Рассмотрим более подробно архитектуру проекта, подходы к решению поставленных задач и механизмы их реализации.

1. Постановка задачи исследования

Целью данной работы является проектирование системы моделирования пенсионной схемы.

Предмет исследования – инструментальные средства для моделирования информационных систем.

Объектом данного исследования является процесс формирования пенсионного фонда.

Основные задачи исследования:

- Провести анализ проблемы автоматизации актуарных расчетов в пенсионном страховании.
- Разработать концептуальную и физическую модели базы данных хранения как личной информации о клиентах, так и информации о договорах страховой компании.
- Провести анализ и обосновать выбор существующих программных инструментариев для разработки систем хранения информации и осуществления эффективной работы с этой информацией.

Специфика развития страхования в Украине, отсутствие программного обеспечения в данной сфере, высокая стоимость зарубежных аналогов, наличие весьма трудоемких расчетов, которых требует пенсионное страхование, большие потоки информации – все это делает автоматизацию страхового процесса необходимой. Существует множество проблем, которые возникают в процессе страхования.

К таким проблемам относятся:

- большие объемы и сложность составления отчетности для контролирующих органов;
- накопление и обработка собственных статистических данных, необходимых для последующего анализа;
- сложность расчета страховых тарифов и разработки соответствующих методик;
- необходимость оперативного анализа эффективности различных видов страхования;
- необходимость оперативной оценки финансового состояния компании для рационального инвестирования средств;
- невозможность развития некоторых видов страхования без соответствующего программного обеспечения [2].

Инструментальные средства для автоматизации страховой деятельности должны отвечать требованиям, которые не только бы отражали отраслевую специфику и решали эти проблемы, но и обеспечивали повышение эффективности работы страховой компании.

2. Проектирование системы для моделирования пенсионного фонда

Наиболее важным этапом создания любой информационной системы является проектирование инструмента, способного реализовать все поставленные в начале проекта задачи. Для проектирования предложенной системы был использован инструмент для моделирования, анализа, документирования и оптимизации бизнес-процессов Rational Rose. Графически представленная схема выполнения работ, обмена информацией, документооборота визуализирует модель бизнес-процесса. Графическое изложение этой информации позволяет перевести задачи управления организацией из области сложного ремесла в сферу инженерных технологий.

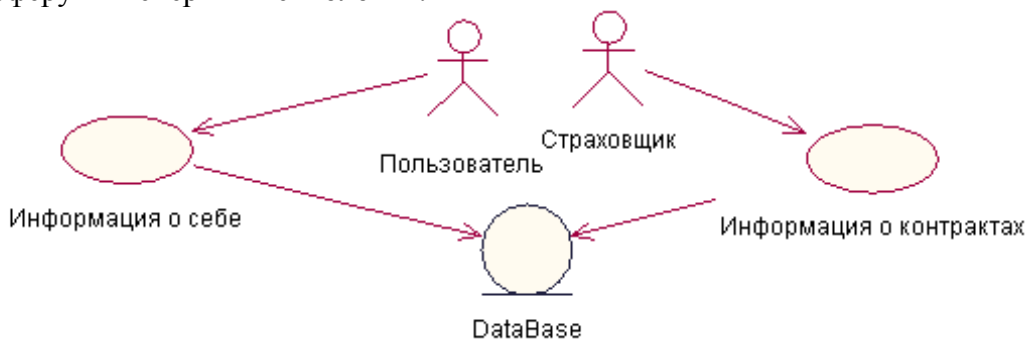


Рис. 1 – Диаграмма прецедентов

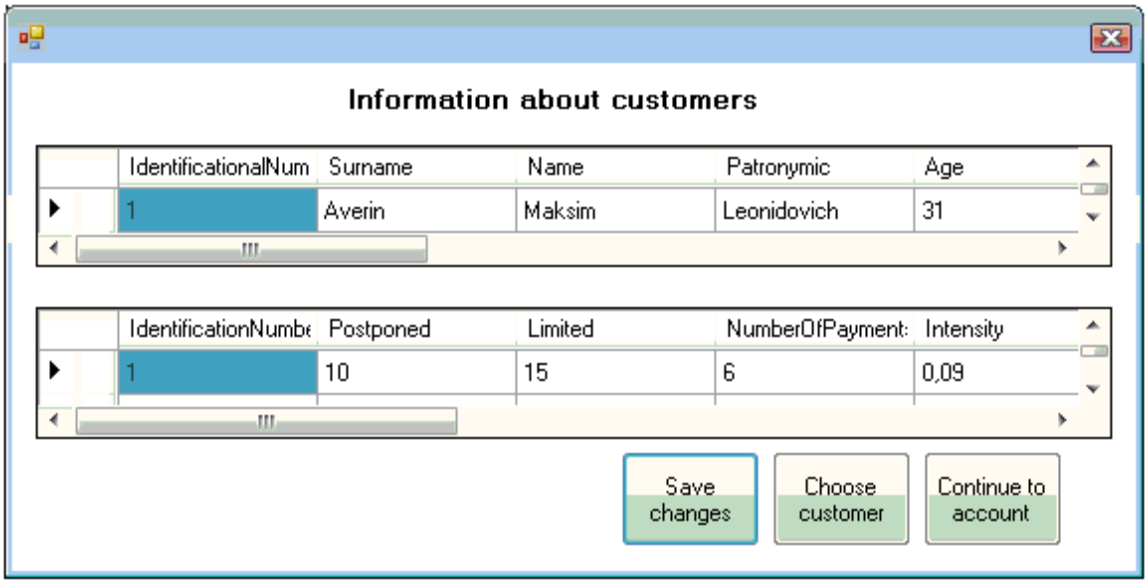


Рис. 2 – Форма для работы с базой данных проекта

На рис. 1 представлена диаграмма прецедентов для страхователя и страховщика. В

данном случае системы должна содержать базу данных, т.к. необходимо удерживать личную информацию о клиентах и договорах страховой компании, а также о выплатах и взносах клиентов страховой компании.

Использование диаграмм проектирования значительно упростило процесс создания самой базы данных для проекта, которая была реализована на mysql. Приложение для работы с базой данных в проекте имеет следующий вид (рис.2).

Использование UML-диаграмм позволяет структурировать и формализовать сущности и классы, которые будут необходимы для создания информационной системы. Очень удобно разделить весь проект на основные части, как это позволяет сделать диаграмма компонентов (рис. 3), и хранить все данные отдельно.

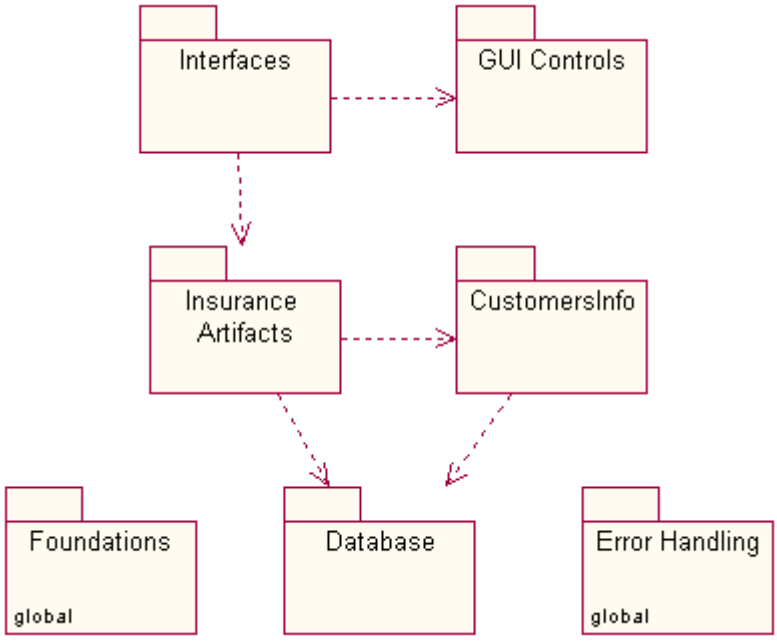


Рис. 3 – Диаграмма компонентов

На рисунке 4 изображена функциональная модель системы. Входными потоками для данной системы являются:

- возраст;
- время в обороте;
- количество выплат в год;
- страховая выплата;
- вероятность неразорения компании, которая устраивает пользователя;
- процентная ставка;
- возможность рассрочки.

Выходными потоками являются:

- численные значения, определяющие аналитический закон;
- численные значения таблицы смертности;
- нагрузка;
- моделирование пенсионного фонда;
- расчет величины пенсий или выплат по заложенным математическим методам или по стандартной схеме.

Управляющий механизм – пользователь, реализующий механизм – программист.

Спроектированная система решает такие задачи:

1. Автоматизация учета информации о клиентах и дальнейший расчет с использованием информации непосредственно из базы данных, т.е. пользователю не нужно самостоятельно вводить данные о себе во всевозможные поля в программе.

2. Моделирование продолжительности жизни популяции в возрасте от 0 до 99 лет путем аппроксимации рассматриваемыми аналитическими законами смертности: де Муавра, Гомперца и Мэйкхама или использование статистических данных за прошлые годы для расчета.
3. Построение таблицы смертности и коммутационных чисел.
4. Формирование и оценка параметров нагрузки для страхования жизни.
5. Расчет пенсий или выплат по стандартной схеме с дальнейшим выводом графиков на экран.
6. Расчет пенсий или выплат с использованием индивидуальных методов, сравнение полученных результатов с дальнейшим выводом графиков на экран.
7. Расчет вероятностей дожития пользователя до определенного возраста.
8. Расчет абсолютной ставки, которую должна будет выплатить компания в случае наступления всех страховых случаев.
9. Расчет баланса между выплатами и взносами для страховой компании.
10. Моделирование пенсионного фонда с дальнейшим выводом графиков на экран.
11. Формирование выводов о полученных результатах, импорт отчетов в MS Excel.

Основной задачей проектирования было дальнейшее создание программного продукта, способного автоматизировать сложные и трудоемкие расчеты в сфере актуарной математики, вероятностные модели которой лежат в основе любых страховых расчетов.

Интерфейс разработанной информационной системы интуитивно понятен пользователю, так как при разработке использовались стандартные компоненты. Разработка проводилась с использованием объектно-ориентированного инструментария C#.

На рисунке 4 изображена диаграмма классов, использованных для создания информационной системы моделирования пенсионного фонда.

Рис. 4 – Диаграмма классов

В данной работе были рассмотрены UML-диаграммы прецедентов и компонентов, также была построена функциональная диаграмма системы. Визуальное моделирование является очень эффективным средством при создании баз данных, а также сложных информационных систем. При помощи созданной архитектуры была разработана информационная система для моделирования пенсионного фонда.

Литература

1. Баскаков В.Н. Актуарные проблемы системы социального страхования / В.Н. Баскаков, А.В Мельников. 1999. – 130 с.
2. Федорова Т.А. Основы страховой деятельности – М.: Издательство БЕК, 2002. – 776 с.

УДК 004.04

ОПЫТ ОБЪЕКТНОГО ПРОЕКТИРОВАНИЯ СТРУКТУРЫ БАЗЫ ДАННЫХ ИНФОРМАЦИОННОЙ СИСТЕМЫ РЕКЛАМНО-ИЗДАТЕЛЬСКОГО ЦЕНТРА

Герасимова Ольга Игоревна, Шахтинский институт (филиал) Южно-Российского государственного технического университета (Новочеркасского политехнического института), Россия, Шахты, itblack@rambler.ru

Олейник Павел Петрович, к.т.н., Системный архитектор программного обеспечения, ОАО «Астон», Россия, Ростов-на-Дону, xsl@list.ru

В статье рассматривается процесс проектирования и реализации структуры БД для информационной системы рекламно-издательского цента с подробным описанием ключевых этапов (концептуального и логического проектирования). Актуальность работы обуславливается тем, что в настоящее время все большее количество организаций переходит от ведения учета на бумажных носителях к использованию единых информационных систем (ИС). Корректно спроектированная и реализованная ИС позволяет сделать процесс учета более быстрым и удобным, одновременно способствуя повышению конкурентоспособности организации на рынке.

Проектирование структуры базы данных (БД) – одна из наиболее ответственных фаз реализации программного обеспечения, успех которой во многом зависит от выбранной методологии. В данной работе использована методология сущность-связь, подробно описанная в [1-2]. Этап проектирования БД начинается с установления границ будущей разрабатываемой системы и определения функционала, который необходимо реализовать в приложении. Концептуальное проектирование БД позволяет описать высокоуровневую модель и начинается с создания концептуальной модели данных организации, полностью независимой от любых деталей реализации. К последним относятся выбранный тип СУБД, состав программ приложения, используемый язык программирования, конкретная аппаратная платформа, вопросы производительности и любые другие физические особенности реализации.

Следующим этапом является логическое проектирование БД, под которым понимается конструирование информационной модели предприятия на основе существующих конкретных моделей данных, но без учёта специфики используемой СУБД и прочих условий реализации [1].

В общем случае, логическое проектирование БД заключается в преобразовании концептуальной модели данных в логическую модель данных предприятия с учетом выбранного типа СУБД. Логическая модель данных является источником информации для этапа физического проектирования. Она предоставляет разработчику физической модели данных средства проведения всестороннего анализа различных аспектов работы с данными, что имеет важное значение для выбора эффективного проектного решения.

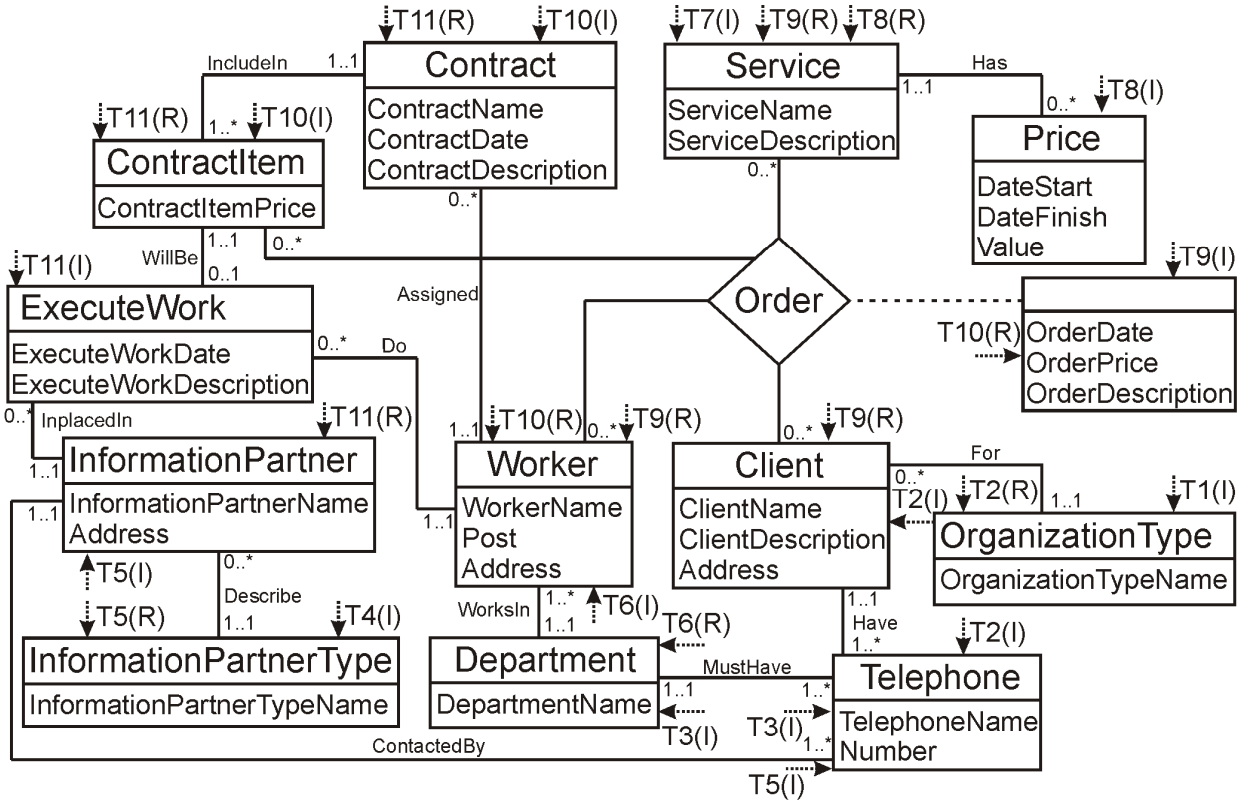
Последним этапом является физическое проектирование БД, под которым понимается описание конкретной реализации БД, размещаемой во внешней памяти. Физический проект реализует базовые отношения, определяет организацию файлов и состав индексов, применяемых для обеспечения эффективного доступа к данным, а также регламентирует все соответствующие ограничения целостности и меры защиты [1].

Физическое проектирование БД предусматривает принятие разработчиком окончательного решения о способах реализации создаваемой базы. Поэтому физическое проектирование должно производиться с учетом всех особенностей используемой СУБД. Т.к. объектно-ориентированный подход является одним из часто используемых, применяемых при разработке программного обеспечения, то принято решение использовать именно объектно-ориентированную систему управления базами данных, допускающую создание новых типов данных на основе существующих, поддерживающую продолжительные транзакции и наследование, что способствует ускорению разработки и

упрощению сопровождения БД. Между этапами физического и логического проектирования всегда имеется определенная обратная связь, поскольку решения, принятые на этапе физического проектирования с целью повышения производительности разрабатываемой системы могут потребовать определенной корректировки логической модели данных.

В ходе анализа предметной области было выявлено, что необходимо разработать структуру БД, позволяющую сохранять следующую информацию:

- о сотрудниках организации;
- об отделах, в которых работают сотрудники;
- о клиентах, с которыми работает организация;
- о заявках на оказание услуг, подаваемых клиентами;
- о статусе обработки заявки;
- об оказываемой услуге;
- об истории изменения цен на услуги;
- об информационных партнерах, у которых размещается рекламная продукция;
- о заключенных контрактах и конкретных услугах, предоставляемых в его рамках;
- об уже выполненных контрактах и конкретных услугах, предоставленных в рамках контракта.



T1. Добавление нового типа организации (клиента)
T2. Добавление нового клиента
T3. Добавление нового отдела рекламного агентства
T4. Добавление нового типа информационного партнёра
T5. Добавление нового информационного партнёра
T6. Добавление нового работника организации
T7. Добавление новой услуги
T8. Ввод цены на предоставляемую услугу
T9. Регистрация заявки клиента
T10. Оформление нового контракта на услуги
T11. Занесение информации о выполненной услуги

Рис.1 – Карта выполнения транзакций

На этапе концептуального проектирования информационной модели предприятия, независящей от каких-либо физических условий реализации, необходимо определить типы сущностей, т.е. объекты, информация о которых будет сохраняться в БД.

Список типов сущностей предметной области:

- Service – услуга, оказываемая клиентам;
- Price – история изменения цен на услуги;
- Client – клиент, подающий заявку на оказание услуг;

- OrganizationType – тип организации клиента (ООО, ОАО и тп)
- Worker – сотрудник организации, оказывающий услуги и регистрирующий заявки;
- Department – отдел, в котором работает сотрудник;
- Telephon – контактный телефон;
- Contract – договор об оказании услуг клиенту;
- ContractItem – строка контракта, в которой указана оказываемая в рамках контракта услуга;
- ExecuteWork – выполненная услуга, входящая в контракт;
- InformationPartner – средство массовой информации, выступающее в качестве информационного партнёра;
- InformationPartnerType – тип информационного партнёра (газета, журнал и т.п.).

Дальнейшим этапом является выделение всех связей, имеющихся между сущностями. Все выделенные связи наглядно представлены на рисунке 1, на котором также изображены все основные атрибуты, определённые для сущностей и связей.

Для графического представления сущностей и связей создаваемой БД (см. рис.1) разработана ER-диаграмма предметной области, для отображения которой использована нотация унифицированного языка моделирования UML [3]. После того, как построена концептуальная модель БД, необходимо проверить ее на достоверность с помощью определения соответствия транзакциям пользователя. Проверим выполнимость транзакций (T1..T11) при помощи графического представления путей её реализации (карты транзакций), изображённой на рис.1.

Для каждой транзакции в круглых скобках указаны операции, выполняемые транзакцией над данной сущностью: I (Insert) – вставка нового экземпляра сущности, U (Update) – обновление значений атрибутов, R (Read) – чтение значений атрибутов, D (Delete) – удаление сущности из БД.

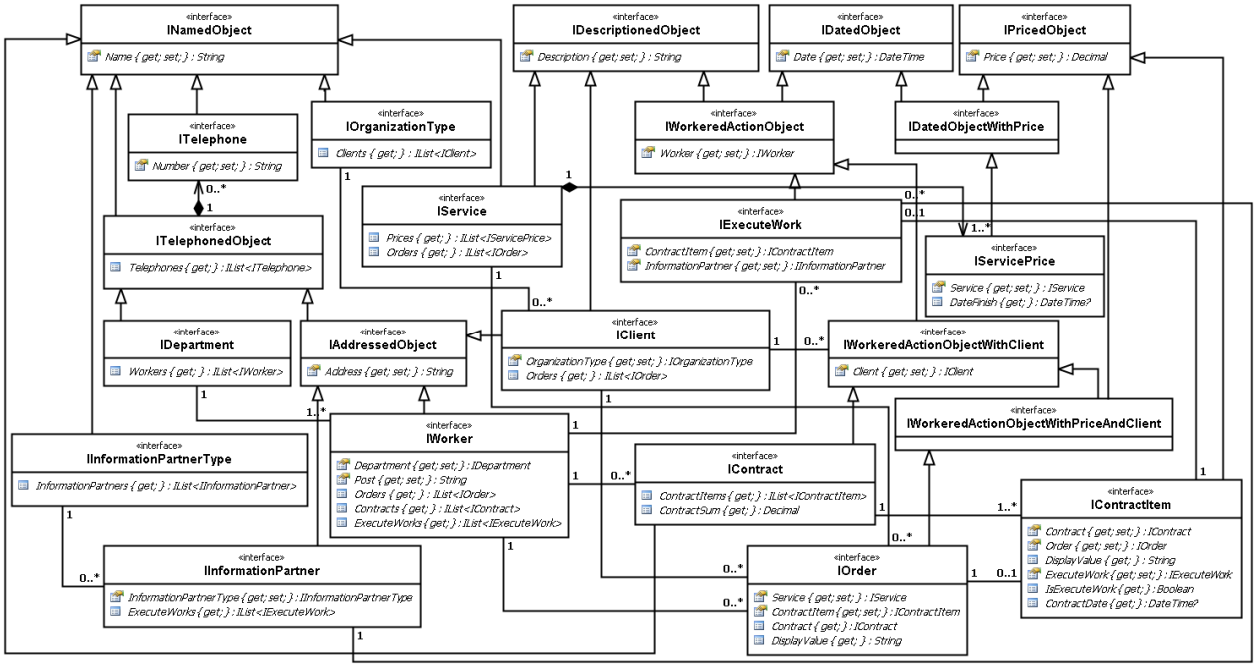


Рис.2 – Логическая модель

Из рис.1 следует, что все описанные ранее транзакции выполнимы, следовательно, концептуальное проектирование структуры БД выполнено верно и можно приступать к логическому проектированию.

Следующим этапом является логическое проектирование БД, которое в общем случае заключается в преобразовании концептуальной модели данных в логическую модель данных предприятия с учетом выбранного типа СУБД (в нашем случае в понятия объектной СУБД, т.е. в логическую объектно-ориентированную). Так как при проектировании системы

используются методы объектно-реляционного отображения, то этап логического проектирования выполняется в понятиях объектной модели данных. Поэтом при преобразовании концептуальной модели в логическую необходимо выполнить следующие операции:

1. Выполнить обобщение/специализацию.
2. Преобразовать связи с атрибутами в отдельные классы.
3. Преобразовать сложные связи в отдельные классы.

Результат проделанных операций представлен на рис. 2.

Для тестирования корректности спроектированной БД реализуем графическое приложение, заполним созданную БД тестовой информацией и проверим разработанное приложение. На рис. 3 приведена экранная форма, иллюстрирующая работу созданного приложения.

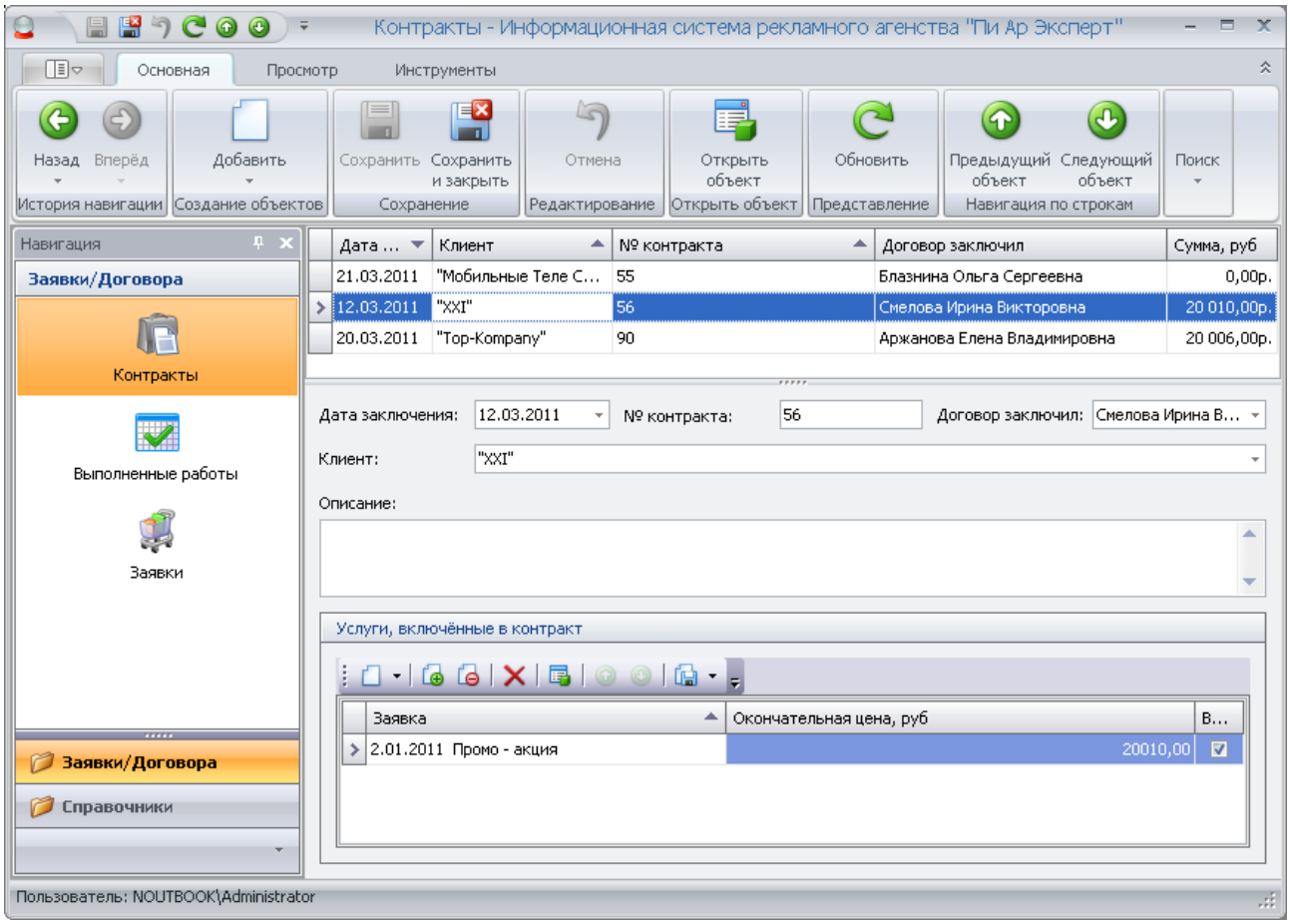


Рис. 3 – Диалоговое окно разработанного приложения

Резюмируя вышесказанное, можно сделать вывод, что разработанное приложение полностью удовлетворяет описанным требованиям и позволяет сохранять всю необходимую информацию. В настоящее время выполняется внедрение системы и его отладка в конкретной организации.

На некоторых предприятиях учет оказанных услуг ведется вручную, тогда как внедрение созданной ИС позволит усовершенствовать этот процесс и сделать его намного более быстрым и удобным. Также внедрение поспособствует повышению конкурентоспособности предприятия на рынке, так как в настоящее время множество предприятий уже перешло от ведения учета вручную к использованию ИС и, чтобы не потерять свое место на рынке, необходимо следить за развитием информационных технологий. В статье представлено практическое применение методологии «сущность-связь», что продемонстрировано созданием концептуальной, логической моделей структуры БД.

Литература

1. Коннолли, Томас, Бегг, Каролин. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е издание. : Пер. с англ. – М. : Издательский дом "Вильямс", 2003. – 1440 с. : ил. – Парал. тит. англ.
2. Дейт К. Дж., Введение в системы баз данных, 7-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2001. – 1072 с. : ил. – Парал. тит. англ.
3. Мюллер Р. Дж., Базы данных и UML проектирование: Пер. с англ. – М. : Лори, 2002. – 420с.: ил. – Парал. тит. англ.

УДК 004.43:378.09

ПОЛИМОРФИЗМ И ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНЫХ ФУНКЦИЙ

Мясникова Нелли Александровна, доцент, Южно-Российский государственный технический университет (Новочеркасский политехнический институт), Россия, Новочеркасск, mnela@list.ru
Шепилов Владислав Александрович, студент, Южно-Российский государственный технический университет (Новочеркасский политехнический институт), Россия, Новочеркасск, ship3000@mail.ru

Для построения иерархий классов в любом языке программирования высокого уровня должен быть реализован механизм полиморфизма, который обеспечивает возможность определения разнообразных аспектов некоторого единого метода по названию для классов разных уровней иерархии. Также различают простой полиморфизм, который базируется на механизме раннего связывания, и сложный полиморфизм, который использует механизм позднего связывания.

Простой (еще называют статический) полиморфизм поддерживается языком C++ на этапе компиляции (так называемое раннее связывание) и реализуется с помощью механизма переопределения функций. Такие полиморфные функции в C++ называют переопределяемыми, в соответствии с общими правилами, они могут отличаться типом возвращаемого параметра и сигнатурой.

Такой подход позволяет строить более гибкие и совершенные иерархии классов, заменяя в производных классах методы. Чтобы показать использование статического полиморфного метода рассмотрим пример 1. В нем функция print() является статическим полиморфным методом. Функция show(), вызывает метод print() и наследуется в производных классах.

Пример 1. Использование раннего связывания

```
#include < iostream. h>
#include <conio.h>
class A
{
  int a;
  public:
  void show()
  {
    cout<<"Содержимое полей объекта"<<endl;
    print();
  } //вызов статической полиморфной функции
  void print(void)
  // первый аспект статической полиморфной функции
  {
    cout<<a<<endl;
  }
  A(int v):a(v){}
};

class B: public A
```

```
{
  int b;
  public:
  void print(void) // второй аспект статической полиморфной функции
  {
    cout<<b<<endl;
  }
  B(int va, int vb):A (va), b(vb) {}
};

class C: public B
{
  int c;
  public:
  void print(void) //третий аспект статической полиморфной функции
  {
    cout<<c<<endl;
  }
  C(int va, int vb, int vc) :B(va, vb), c(vc) {}
};

void main()
{
  clrscr();
  A aa(20); B bb(10,100); C cc(50,200,3000);
  cout<<" Результаты работы: "<<endl;
  // явный вызов полиморфной функции print()
  aa.print(); // выводит: 20
  bb.print(); //выводит: 100
  cc.print(); // выводит: 3000
  getch();
  // неявный вызов полиморфной функции print()
  aa.show(); // выводит: 20
  bb.show(); //выводит: 10
  cc.show(); // выводит: 50
  getch();
}
```

Несложно заметить, что результаты явного и неявного вызовов метода print() различаются. Это связано с тем, что метод show(), который наследуется в производных классах, вызывает метод print(). При раннем связывании show() жестко соединяется с этим методом на этапе компиляции. Поэтому вне зависимости от того, объект какого класса вызывает метод show(), из него будет вызван метод print() базового класса, для которого внутренние поля производных классов недоступны. Это и является главной отличительной чертой раннего связывания.

Чтобы получить правильный результат в подобных случаях, нужно использовать сложный полиморфизм. Сложный полиморфизм можно реализовать с помощью механизма позднего связывания, что требует описания виртуальных функций.

Виртуальные функции – это такие функции, которые объявляются с использованием ключевого слова virtual в базовом классе и замещаются в одном или нескольких производных классах. В этом случае прототипы функций в разных классах должны совпадать по именам, типу возвращаемого результата, сигнатуре. Алгоритмы, реализуемые такими функциями, различаются между собой. В случае, когда типы функций различны, механизм виртуальности для них не включается.

Также стоит заметить, что виртуальную функцию в языке C++ принято называть «полиморфной», что не в полной мере соответствует общепринятой терминологии, по общепринятой терминологии она является «динамической полиморфной».

В качестве иллюстрации рассмотрим пример 2, который является модификацией предыдущего.

Пример 2. Использование виртуальных функций

```
#include <iostream.h>
#include <conio.h>
class A
{
  int a;
  public:
  void show()
  {
    cout<<"Содержимое полей объекта"<<endl;print();
  }
  virtual void print(void) // описание виртуальности функции
  {
    cout<<a<<endl;
  }
  A(int v):a(v){}
};
class B: public A
{
  intb;
  public:
  void print(void) // первый аспект виртуальной функции
  {
    cout<<b<<endl;
  }
  B (int va, int vb) :A (va), b(vb) {}
};
class C: public B
{
  int c;
  public:
  void print(void) // второй аспект виртуальной функции
  {
    cout<<c<<endl;
  }
  C(int va, int vb, int vc) :B(va, vb), c(vc) {}
};

void main()
{
  clrscr();
  A aa(10), *pa; // указатель на объект базового класса

  B bb(10,100);
  C cc(10,100,1000);

  cout<<" Результаты работы: "<<endl;
  cout<<'' Явный вызов полиморфной функции print(): "<<endl;
  aa.print(); //выводит: 10
  bb.print(); //выводит: 100
  cc.print(); //выводит: 1000
  getch();
  cout<<" Неявный вызов полиморфной функции print(): "<<endl;
  aa.show(); //выводит: 10
  bb.show(); //выводит: 100
  cc.show(); //выводит: 1000
  getch();
  cout<<"Вызов функции print() по указателю"<<endl;

  pa=&aa; pa->prmt(); //выводит: 10
  pa=&bb; pa->print(); //выводит: 100
  pa=&cc; pa->print(); //выводит: 1000
  getch();
}
```

Как мы видим, каждый раз вызывается нужная версия виртуальной функции.

Функция, которая была объявлена виртуальной, остается ею, сколько бы производных классов ни образовывалось. Но иногда в одном или нескольких производных классах переопределение виртуальной функции может отсутствовать. Тогда механизм подключения виртуальной функции сохраняется и вызывается функция класса, которая ближе к рассматриваемому классу. Если будет отсутствовать аспект виртуальной функции, это не нарушит механизм позднего связывания, и если у наследников такого класса появится аспект виртуальной функции, он будет вызываться без каких-либо нарушений.

Литература

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. – М.: Бином; СПб.: Невский диалект, 2001. – 560 с.
2. Бьерн Страуструп. Язык программирование C++. М.: Бином, 2001. – 1099 с.
3. Г. С. Иванова. Объектно-ориентированное программирование. – МГТУ им. Н. Э. Баумана, 2002.
4. Курс: Основы информатики и программирования. Лекция №10: Основы объектно-ориентированного программирования. Применение ООП к разработке программных проектов. Основные концепции ООП, www.intuit.ru/department/se/oip/10/

УДК 004.652.5

ОБЪЕКТНАЯ БАЗА ДАННЫХ ДЛЯ СТУДЕНТОВ

Неудачин Илья Георгиевич, к.ф.-м.н., доцент, Уральский федеральный университет имени первого Президента России Б.Н.Ельцина, Россия, Екатеринбург, nigs@sky.ru

Введение

ZODB (Zope Object DataBase) – система хранения объектов приложений, написанных на языке Python. Она обеспечивает язык программирования средствами, которые автоматически записывают объекты на диск и читают их снова, когда они требуются программой. Установка ZODB добавляет такие средства к Python. Пакет ZODB выделен из инфраструктуры разработки и Web-публикации объектов Zope [1, 2], для которой – это стандартная объектная база данных (ОБД). Изучая ОБД, студенты осваивают разнообразие архитектур баз данных. Так же как Python и Zope, ZoDB поставляется в открытых кодах (http://old.zope.org/Products/ZODB3.2), бесплатно и с лицензией, благоприятной для ее модификации студентами и дальнейшего применения.

1. Подробное описание инструментария

К выбору ZODB как предмета обучения приводят три основные причины. Во-первых, ZODB – открытый продукт. Что позволяет студентам, при желании, не только видеть, как что-то работает, но понимать, как это работает. Во-вторых, использование ZODB бесплатное. Это позволяет изучать, исследовать и приобретать опыт без больших расходов. В-третьих, можно изменять коды ZODB. Она интересна как пользователям и администраторам, так и тем, кто программирует на языке Python. Описываемый учебный материал частично применяется при подготовке студентов – бакалавров профиля «Вычислительные машины, комплексы, системы и сети: системная интеграция». ОБД дала логическую связку между учебными разделами программирования на языке Python и проектированием динамических Web-приложений в инфраструктуре Zope [3].

В ZODB хранится не только объект, но и его текущее состояние, информация о нём, а также как он используется. Т.е. запоминаются действия пользователя, что допускает отмену пользователем этих действий. ZODB поддерживает транзакции, версии, откат и, как следствие, – упаковку завершенных транзакций и версий. Эта мощная объектная система баз данных, которая может использоваться с Zope или без Zope.

Сама ZODB объектно-ориентированна, то есть описывается как пакет деревьев классов. В ней можно произвольно менять класс, задающий вид хранилища (Storage). Рассматривается программирование на языке Python для ZODB и для ее расширения с помощью компонента ZEO (Zope Enterprise Objects). Последний механизм добавляет масштабируемость и распределенность ОБД в сети.

Объекты экземпляра инфраструктуры Zope [4] хранятся базой данных ZODB в формате файла Data.fs (в автономном случае может быть другое имя файла), который полностью независим от платформы и ОС. Собственно, ядром ZEO является еще одно хранилище ServerStorage, которое обращается не к локальному Data.fs, а к удаленному серверу. Вторым компонентом ZEO является сервер. На основе ранее полученных сведений становится понятными место ОБД в архитектуре Zope и реализация интерфейса управления ее главными параметрами.

2. Время жизни объекта

Если речь идет о рядовой программе, то время жизни объекта некоторого класса, описанного в программе на языке Python, ограничивается временем выполнения этой программы. Ограничения снимаются путем сохранения объектов (точнее, значений полей объектов) в файле. Обычно используется логическая архитектура реляционной или объектной базы данных. Все эти средства позволяют делать объекты хранимыми (persistent). Как правило, при записи объекта производится его сериализация, а при чтении – десериализация. Сериализация – это процесс перевода какой-либо структуры данных в последовательность битов. Сериализация используется для передачи объектов по сети и для сохранения их в файлы. Обратной к операции сериализации является операция десериализации – восстановление начального состояния структуры данных из битовой последовательности.

Реляционные базы данных не специализируются на объектах, их назначение более универсальное, чем ОБД [5]. Реляционные базы данных сохраняют информацию в таблицах. Не трудно написать код Python, который создаст экземпляр объекта и заполнит его данными из таблиц. Немного больше усилий, и можно сформировать простой инструмент, обычно называемый объектно-реляционным преобразователем, для того чтобы делать это автоматически. Однако трудно сделать объектно-реляционную СУБД приемлемо быстрой. Бесхитростная реализация, подобная первому примеру, – слишком медленная, потому что она должна делать несколько запросов, чтобы обратиться к данным всего объекта. Более высокоэффективные объектно-реляционные преобразователи кэшируют объекты, чтобы улучшить эффективность, выполняя SQL запросы, только когда они фактически необходимы.

В настоящее время разработаны специализированные ОБД для объектов Python. Самое простое решение – объединить модуль pickle и модуль базы данных, чтобы сохранять и восстанавливать объекты. Фактически, модуль shelve, включенный в стандартную библиотеку Python, делает это. ZODB и ZEO относятся к базам данных с развитыми возможностями. Объектная база данных, подобная ZODB, просто сохраняет внутренние указатели объекта на объект. Чтение одиночного объекта происходит намного быстрее, чем выполнение связки запросов SQL и сборка результатов. Просмотр всех включений, следовательно, все еще остаётся неэффективным, но не чрезвычайно неэффективным.

Недостаток этого подхода состоит в том, что программист Python должен явно управлять объектами, читая объект, когда это необходимо, и записывать его в файл, когда объект больше не требуется. Конечно, ZODB в инфраструктуре Zope управляет объектами за вас, храня их в кэше, и записывая их на диск, если к ним не обратились некоторое время.

3. Архитектура ZODB

Автономная установка ZODB выглядит (рисунок 1) как пакет из нескольких каталогов, содержащих модули.

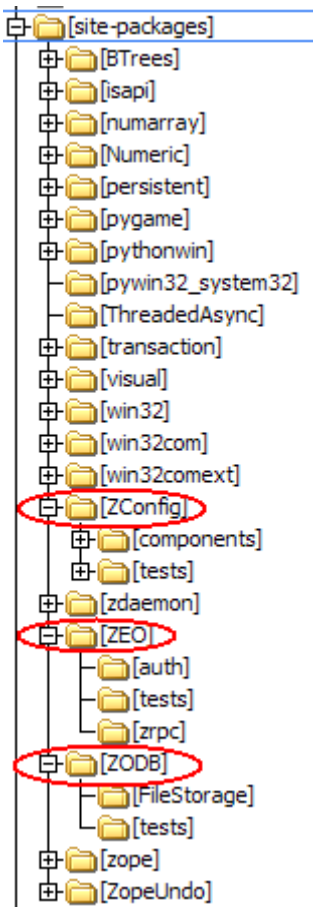


Рис. 1 – Пакет ZODB

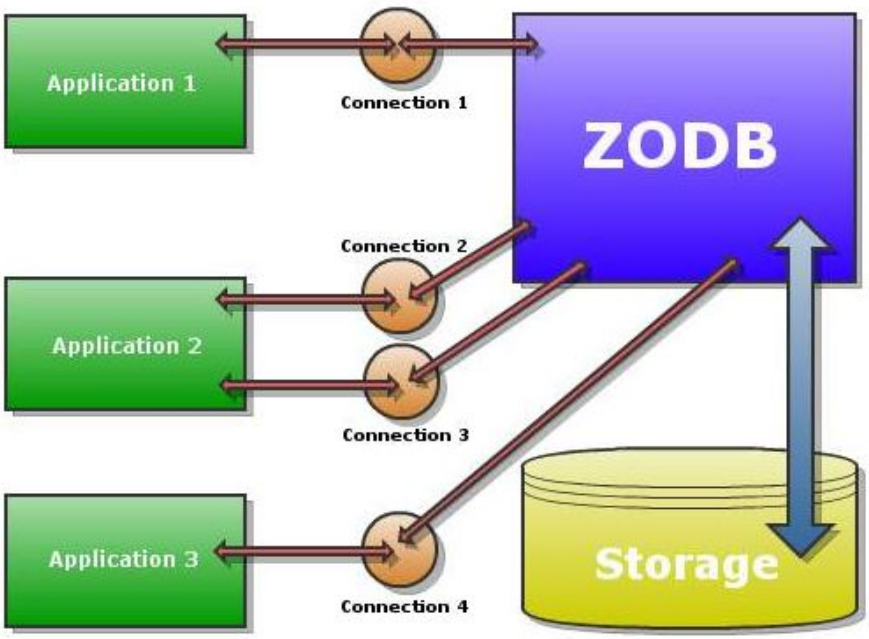


Рис. 2 – Взаимодействие ОБД с приложением [6]

Компоненты, входящие в установку ZODB:

- Ядро ZODB, включая процессор хранения.
- Стандартные хранилища типа FileStorage.
- Модули классов хранения persistent.
- Модули для сохранения Б-ДЕРЕВЬЕВ Btrees.
- ZEO, сервер хранилища по подключению TCP/IP.
- ZConfig – пакет конфигурации Zope. Зачастую для одного и того же кода требуется использование различных хранилищ. Например, на станции разработчика это может быть FileStorage с тестовыми данными, а на рабочего сервере – ClientStorage с реальными данными. Поэтому удобно использовать конфигурации и конфигурационные файлы. С ZODB идет еще один «кусочек» Zope – это пакет ZConfig. Т.е. в коде клиентов нужно указать, какой конфигурационный файл будем использовать
- Документация.

Компоненты архитектуры базы данных ZODB предоставляют программисту много возможностей:

- Многочисленные виды памяти (Storage Interface).
- Поддерживаются разнообразные администраторы хранения, что позволяет сохранять объекты в файловой системе, во временной файловой системе, в файловой системе только для чтения, и в реляционной базе данных.
- Прозрачность. Действия ZODB прозрачны для объектов. Объекты, которые желательно хранить, потребуют только «встраивания» в класс Python Persistent.Persistence.
- Транзакции. ZODB работает с транзакциями, т.е. последовательности изменений для базы данных могут происходить в виде транзакции. Если ошибка возникла где-нибудь в транзакции, то транзакция не будет передана в базу данных.

- Возвраты (Undo). Многие администраторы хранения выходного буфера конкатенируют новые версии объектов к концу базы данных и хранят все старые изменения объекта. Эти администраторы позволяют вам вернуть базу данных обратно – в предыдущее состояние, транзакция за транзакцией.
- Высокая эффективность. ZODB имеет много эффективных расширений, включая объектный кэш «в памяти» с конфигурацией перестраиваемого размера.

На текущую версию ZODB часто ссылаются как на ZODB3.

4. Принцип работы ZODB

ZODB концептуально проста. Классы Python наследуют классу Persistent для связи с ZODB. Экземпляры хранимых объектов поступают из постоянного хранилища данных, типа дискового файла, когда программа нуждается в них, и кэшируются в оперативной памяти. ZODB улавливает модификации объектов. Когда команда Python типа obj.size = 1 выполнена, изменяемый объект отмечается как изменённый (dirty, «грязный»). По запросу, любые грязные объекты записываются во внешнее постоянное хранилище. Дополнительный атрибут dirty порожден передачей транзакции. Транзакции могут быть прерваны или прокручены обратно, что кончается отказом от любых изменений, а грязные объекты, возвращаются к их начальному состоянию, в котором транзакция началась.

Термин «транзакция» имеет специфическое техническое значение в информатике. Чрезвычайно важно, что содержание базы данных не разрушается программными или аппаратными сбоями. Большинство программ обеспечения баз данных предлагает защиту против такого искажения, поддерживая четыре полезных свойства ACID (Atomicity, Consistency, Isolation, Durability). Что означает: Атомарность, Согласованность, Изолированность, Долговечность.

ZODB имеет почти полную ACID – совместимую поддержку транзакций (включая точки сохранения информации о текущем состоянии системы). ZODB обеспечивает три из этих свойств. Не в полной мере поддерживается только согласованность. ZODB не применяет понятие структура базы данных, следовательно, не имеет способа предписания согласованности структур.

5. Классы интерфейса хранения

Есть три основных интерфейса, поддерживаемых ZODB: Storage – хранилище, DB – база данных и классы Connection – подключений. Интерфейсы DB и Connection имеют одиночные реализации, но есть несколько различных классов, которые реализуют интерфейс хранения Storage.

- Классы хранения Storage находятся на самом низком уровне, и управляют сохранением и восстановлением объектов из некоторой формы долгосрочного хранения. Были написаны несколько различных типов Storage. Как, например, класс FileStorage, который использует регулярные дисковые файлы, или bsddb3Storage, который применяет базу данных Sleepycat Software BerkeleyDB. Можно написать новый класс Storage, который сохранит объекты в реляционной базе данных или в файле специального формата, например, если это лучше подходит конкретному приложению.
- Класс DB размещается на вершине хранилища, и обеспечивает взаимодействие между несколькими подключениями. Один экземпляр DB создается на один процесс.
- Наконец, класс Connection кэширует объекты, и перемещает их в и из хранилища объектов. Multi-threaded программа должна открыть отдельный экземпляр Connection для каждой нити. Различные нити могут затем изменять объекты и передавать их модификации независимо.

Подготовка к взаимодействию с ZODB происходит за три шага: нужно открыть объект хранилища Storage, затем создать экземпляр базы данных DB, который использует Storage, и получить подключение Connection из экземпляра DB. Все это описывают всего несколько

строчек кода:

```
from ZODB import FileStorage, DB
storage = FileStorage.FileStorage('test-filestorage.fs') #1
db = DB(storage)                                         #2
conn = db.open()                                         #3
#здесь выполняются операции с ОБД test-filestorage.fs
```

Можно легко задействовать совершенно другой механизм хранения данных, изменив первую строку (#1), которая открывает Storage (рисунок 2). Вышеупомянутый пример использует FileStorage. ZODB устанавливается с несколькими различными классами, которые обеспечивают интерфейс хранения Storage. Такие классы управляют записью объектов Python на физический носитель данных, которым может быть дисковый файл (класс FileStorage), BerkeleyDB файл (BerkeleyStorage), реляционная база данных (DCOracleStorage) или некоторая другая среда.

ZODB предлагает много различных способов хранения объектов, включая файлы, реляционные базы данных и специальное хранилище клиент/сервер, которое сохраняет объекты на удаленном сервере. Здесь был создан объект хранилища (тип FileStorage) и подключен к базе данных. Zope устанавливается с несколькими классами баз данных для хранения объектов, но один из них наиболее интересный – это ClientStorage из продукта ZEO.

6. Принцип работы ZEO

ZEO добавляет ClientStorage (новое хранилище Storage), которое не записывает объекты на физические носители, а только пересылает запросы по сети на сервер. Сервер, который управляет экземпляром класса StorageServer, просто действует как внешний интерфейс для некоторого физического класса хранения Storage. Это – довольно простая идея, которая открывает много возможностей.

Хранилище ClientStorage совершает TCP/IP подключение к StorageServer (сервер также предоставляет ZEO). Сервер обслуживает много различных процессов на одной или нескольких машинах, позволяя работать с одной и той же объектной базой данных и, следовательно, с одними и теми же объектами. Каждый процесс получает кэшируемую «копию» специфического объекта для быстродействия. Все ClientStorage, подключенные к StorageServer, общаются по специальному объектному протоколу с контролем аннулирования передачи и с кэшированием, чтобы синхронизировать все компьютеры. ClientStorage просто делает удаленные вызовы процедуры на сервер, который затем передает их регулярному классу хранения типа FileStorage. ZEO состоит из приблизительно 6000 строчек кода Python, исключая тесты. Текст относительно мал, потому что содержит код только для сервера TCP/IP и для нового типа хранения – ClientStorage.

Любое число процессов может создавать экземпляр ClientStorage и любое число нитей в каждом процессе может использовать этот экземпляр. ClientStorage интенсивно кэширует объекты локально, для того чтобы избежать использования несвежих данных. Сервер ZEO посылает сообщение аннулирования всем подключенным экземплярам ClientStorage при каждой операции записи. Такое сообщение содержит ID для каждого измененного объекта, позволяя экземплярам ClientStorage удалить старые данные этого объекта из их кэшей.

7. Пакет Btrees

Программисту ZODB словари Python не всегда подходят. Наиболее важный случай – тот, когда нужно сохранить очень большое отображение. Когда к словарю Python обращаются в ZODB, весь словарь должен быть расконсервирован и принесен в память из хранилища. Если сохранять что-то очень большое, типа базы данных 100000 пользователей, расконсервирование такого большого объекта будет медленным. Для хранения большого объема данных обычные словари не эффективны.

В ZODB есть пакет BTrees, реализующий B-tree. B-tree (по-русски произносится как Б-дерево) – это структура данных, дерево поиска. С точки зрения внешнего логического

представления – это сбалансированное, сильно ветвистое дерево во внешней памяти. BTrees содержит структуру дерева данных, которая ведёт себя подобно отображению, но распределяет ключи по ряду узлов дерева. Узлы сохраняются в отсортированном порядке. Затем только тогда узлы расконсервируются и переносятся в память, когда к ним обращаются, и все дерево не занимает память.

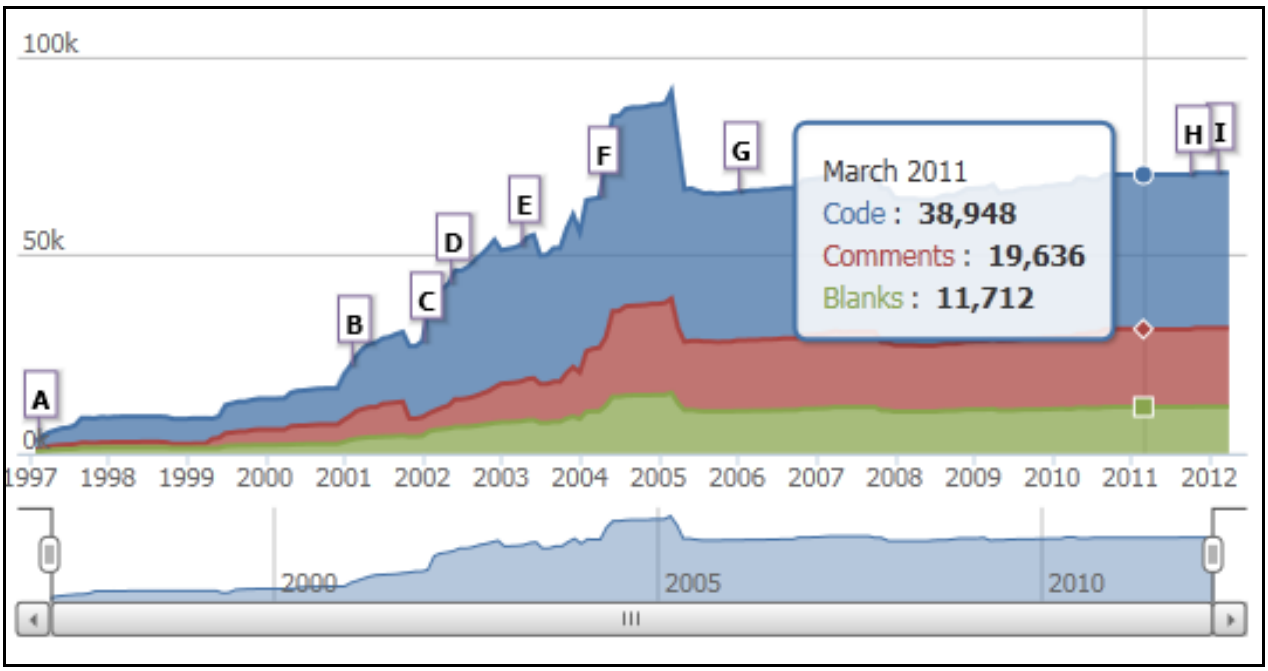


Рис 3. Пользователи ZODB по данным http://www.ohloh.net/p/zodb

Пакет BTrees имеет разнообразные структуры данных, специализированные для обработки ключей, которые действуют быстрее и используют меньше памяти. Есть пять модулей, которые обрабатывают различные варианты. Первые два символа имени модуля определяют типы ключей и значений в отображениях – O для любого объекта и I для целого числа. Модуль BTrees.IOBTree обеспечивает отображение, которое принимает целочисленные ключи и произвольные объекты как значения. Пять модулей называются OOBTree, IOBTree, OIBTree, и (новый модуль из ZODB 3.4) IIBTree. Два символа префикса повторяются в именах типов данных. Например, модуль BTrees.OOBTree определяет следующие типы: OOBTree, OOBUCKET, OOSET, и OOTREESET. Точно так же каждый из остальных четырех модулей определяет собственные варианты этих четырех типов.

Выводы

Рассмотрим некоторые достоинства ZODB:

- Родная объектная база данных (ОБД) для Python, т.е. любые объекты классов Python могут сохраняться в ZODB. Прозрачное хранение объектов Python.
- Транзакции. Хранение истории/возможность отмены.
- Удобный интерфейс к ZODB в Zope.
- Масштабируемая архитектура ОБД через ZEO.
- Эффективная поддержка больших двоичных объектов.
- Сменяемые хранилища.
- Открытые, модифицируемые коды.
- Кроссплатформенная.
- Не нужен SQL.

Однако у ZODB имеются и недостатки, перечисленные ниже:

- Необходимо знание языка Python на уровне выше среднего.
- Сильная ассоциация с Zope.
- Низкая популярность объектных баз данных вообще.

- Отсутствует интерфейс СУБД.
- Система транзакций и опция отмены добавляют к размеру файла ОБД лишние килобайты. Необходимо налаживать периодическую упаковку и чистку ОБД.
- Нет специализированного языка запросов для ZODB.
- Скудная документация вообще и на русском языке в частности.
- Не любые объекты классов Python могут сохраняться в ZODB, т. к. для полноценного ее использования достаточно большой процент объектов должен быть унаследован от специального класса persistent.

Преимущества ZODB превосходят ее недостатки, поэтому она популярна среди пользователей (рис. 3). Как видим, активность сообщества пользователей ZODB стабильна с 2005 года и даже имеет тенденцию роста по приведенной 100-бальной шкале.

Литература

1. Спикльмайр С. и др. Zope. Разработка Web-приложений и управление контентом: Пер. с англ. – М.: ДМК Пресс, 2003. – 464 с.
2. Неудачин И.Г. Среда Web-публикации объектов для студентов // Объектные системы – 2011: материалы III Международной научно-практической конференции. / Под общ. ред. П.П. Олейника. – Ростов-на-Дону, 2011. – с. 62-65.
3. Неудачин И.Г. Инфраструктура программирования в открытых кодах для студентов. //Теория и практика применения свободного программного обеспечения: сборник трудов участников Всероссийской молодёжной конференции с элементами научной школы. – Магнитогорск: МаГУ, 2011. – с. 57-64.
4. Неудачин И.Г. Объектный дизайн Web-порталов. Объектные системы – 2011 (Зимняя сессия): материалы V Международной научно-практической конференции (Ростов-на-Дону, 10-12 декабря 2011 г.) / Под общ. ред. П.П. Олейника. – Ростов-на-Дону: ШИ ЮРГТУ (НПИ), 2011. – с. 63-67.
5. ORM как антипаттерн. Шитов Н.А., Галиаскаров Э.Г., Объектные системы – 2011 (Зимняя сессия): материалы V Международной научно-практической конференции (Ростов-на-Дону, 10-12 декабря 2011 г.) / Под общ. ред. П.П. Олейника. – Ростов-на-Дону: ШИ ЮРГТУ (НПИ), 2011. – с. 72-76.
6. Andrew M. Kuchling. The MEMS Exchange Tools: A shortened version of the ZPUG slides, presented at the Python10 conference, Feb. 6 2002. [Электронный ресурс] http://www.amk.ca/talks/2002-02-06/

УДК 004.4

МОДЕЛИРОВАНИЕ СИСТЕМЫ ПЛАНИРОВАНИЯ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ РЕСУРСОВ В ИССЛЕДОВАТЕЛЬСКИХ ЗАДАЧАХ

Жукова Светлана Александровна, к.т.н., доцент, Чайковский технологический институт (филиал) Ижевского государственного технического университета, Россия, Чайковский, otdel_it@chti.ru
Патютько Вадим Николаевич, студент, Чайковский технологический институт (филиал) Ижевского государственного технического университета, Россия, Чайковский

Введение

Важнейшим направлением повышения эффективности информационных технологий является развитие уровня автоматизации процессов комплексных поставок потребителям разнообразных сервисных процедур. Возрастает интерес к программному обеспечению, которое предоставляет удаленный доступ к ресурсам и сервисам в едином формате. Разработка такого программного обеспечения осуществляется с применением web-технологий, технологий открытых систем, среди которых особо актуальна технология «облачных вычислений» [1].

«Облачная» модель распространения и поддержки программного обеспечения предполагает использование программных приложений в режиме удаленного доступа. Суть

этой модели заключается в максимальном переносе бизнес-логики и данных на сервер. При реализации этой модели все данные хранятся на сервере, вычисления проводятся на сервере, взаимодействие пользователей осуществляется посредством обмена данными через сервер. При этом на долю электронного устройства пользователя (компьютер, планшет, смартфон) остается только задача обеспечения связи с сервером и отображения информации. Таким образом, достигается независимость предоставляемого функционала от конкретного устройства – на любом компьютере, планшете, смартфоне приложение может выглядеть одинаково, предоставлять одинаковый функционал и актуальные данные. При использовании облачных сервисов не требуется оборудование локальных рабочих мест мощными вычислительными ресурсами, т.к. приложения выполняются удаленно. Применение данной технологий актуально для комплексной автоматизации процессов поддержки исследовательской деятельности. Для этого разрабатывается проект «Автоматизированная система открытых виртуальных лабораторных комплексов» (далее АС ОВЛК), номер государственного контракта: 02.740.11. 0658 от 29.03.2010, целью которой является интеграция ресурсов исследовательской деятельности для обеспечения доступности и удобства их использования пользователями путем их размещения в глобальном информационном пространстве [2]. К ресурсам исследовательской деятельности относятся интеллектуальные ресурсы (модели объектов и методы их исследования, программы ЭВМ, реализующие методы, результаты экспериментов); организационные ресурсы (правила и инструкции выполнения исследования, нормативная документация, полезные ресурсы по тематикам научных исследований); вычислительные ресурсы (программно-аппаратные комплексы, необходимые для выполнения экспериментов). Совокупность ресурсов формируется в соответствии с задачей исследования и образует виртуальный лабораторный комплекс. Основные задачи системы:

- автоматизация формирования лабораторных комплексов и их представление удаленному пользователю в виде готового инструмента организации деятельности, обеспечения всей необходимой документацией;
- централизованный сбор и хранение данных о ресурсах исследовательской деятельности, обработка данных при выполнении численных экспериментов,
- организация образовательного процесса с использованием виртуальных лабораторных комплексов для проведения практических и лабораторных занятий;
- автоматизация процесса формирования документации и обмен результатами экспериментов с внешними системами;
- визуализация результатов экспериментов;
- организация коллективной работы над проектами;
- рациональное использование вычислительных ресурсов.

1. Постановка задачи

Одной из ресурсоемких задач системы является выполнение экспериментов. Решение этой задачи предполагает предоставление в общий доступ вычислительных ресурсов группе пользователей в соответствии с потребностями для выполнения экспериментов. В настоящее время предлагается ряд технологий обеспечения производительности: параллельные технологии, GRID-системы, кластеры. Наиболее подходящей технологией для реализации этой задачи является организация распределенных вычислений, преимуществом которой является возможность наращивания производительности за счет горизонтального масштабирования, т.е. добавление вычислительных узлов в состав системы и рационального их распределения между вычислительными задачами, в нашем случае – экспериментами. Для решения данной задачи в рамках проекта разрабатывается подсистема планирования вычислительных ресурсов, которая позволяет выполнять следующие функции:

- добавление вычислительных ресурсов и их регистрацию в составе системы;
- оценка вычислительных мощностей подключаемого вычислительного ресурса;

- расчет требуемых вычислительных мощностей для ВЛ в зависимости от введенных начальных условий;
- планирование распределения вычислительных ресурсов между экспериментами пользователей;
- запуск эксперимента на удаленном вычислительном ресурсе.

Вместе с тем подсистема планирования эксперимента должна соответствовать требованиям, для которых определены количественные и качественные характеристики:

- функционирование на наиболее распространенных вычислительных платформах с минимальными усилиями по настройке, Kol_n (чел/час);
- корректная обработка запросов пользователей численностью до нескольких сотен без потери производительности Kol_q (чел/сек);
- допустимое время отклика $Time_q$: на запрос – не более 10 сек, на формирование отчетов – не более 10 минут, на обработку экспериментов – в режиме реального времени не более 1 часа, в режиме отложенного времени – не более 1 суток.

Таким образом, стоит задача разработки системы планирования распределенных вычислительных ресурсов для проведения экспериментов согласно требованиям пользователя.

Разработка данного класса систем является сложной, трудоемкой и дорогостоящей, т.к. ним предъявляются достаточно серьезные требования, а при реализации и эксплуатации системы используется высокотехнологическое программно-аппаратное обеспечение, к которому относятся сервера, кластеры, коммуникационное оборудование и системное и инструментальное программное обеспечение. В этой связи разработка системы осуществлялась в несколько этапов путем построения рядом моделей и изучения ее характеристик.

2. Этапы моделирования

Моделирование системы выполнялось с применением методологии объектно-ориентированного проектирования (ООП) [3,4].

Согласно методологии ООП моделирование программного продукта выполняется в следующие этапы: построение модели анализа, модели дизайна, модели реализации. Описание моделей выполнялось на языке UML 2.0 в среде проектирования IBM Rational Software Architect.

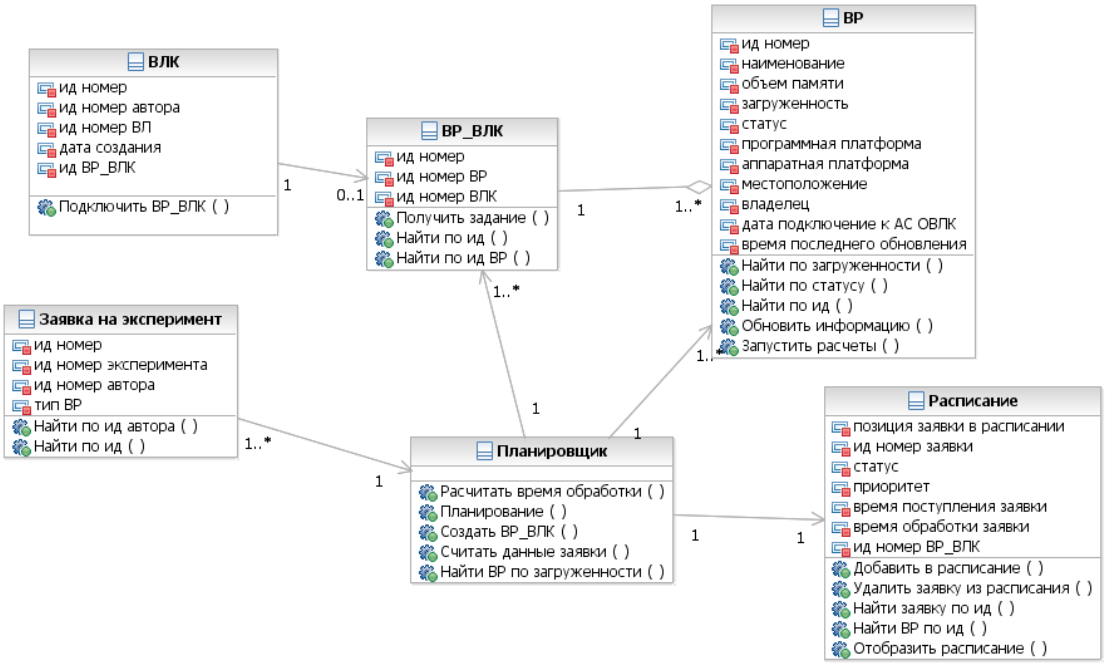


Рис. 1 – Модель анализа АИС “Учет и планирование ВР ВЛК”

На этапе моделирования и анализа предметной области были определены ключевые сущности, их свойства и связи между ними:

- ВЛК – объект, содержащий информацию об объекте исследования (ВЛ) и имеющий ссылку на ВР для проведения эксперимента;
- ВР_ВЛК – объект, представляющий собой вычислительный ресурс ВЛК, состоящий из группы узлов из справочника ВР;
- Расписание – объект, определяющий расписание проведения экспериментов в АС ОВЛК;
- ВР – объект, содержащий информацию о доступном вычислительном ресурсе в АС ОВЛК;
- Заявка на эксперимент – объект, содержащий требования к ВР для проведения эксперимента;
- Планировщик – объект, отвечающий за оценку требований к вычислительным ресурсам и их распределением между заявками пользователей.

Для описания модели анализа использовалась диаграмма классов, отражающая объекты и связи между ними (рисунок 1).

На этапе моделирования структуры системы решались следующие задачи:

- определение необходимого числа уровней подчинения;
- установление между уровнями правильных взаимоотношений, что связано с задачами согласования целей элементов различных уровней и оптимальным стимулированием их работы;
- определение состава подсистем (пакетов) и связей между их компонентами.

Структура многоуровневой архитектуры АС построена в соответствии со спецификацией проектирования распределенных объектно-ориентированных систем Java2EE, в которой система представляет собой многозвенное распределенное приложение и состоит из прикладной модели, серверной модели и модели управления транзакциями. В качестве базового архитектурного шаблона выбрана модель распределенного приложения: модель–представление–контроллер (model-view-controller, MVC) [4]. В данной модели система декомпозируется на три базовых уровня: пользовательский (View), операционный (Controller) и предметной области (Model). Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате. Данная модель позволяет изолировать пользовательский интерфейс от предметной области, таким образом можно модифицировать, добавлять новые компоненты в систему (например, новые типы ВЛ), не затрагивая программные компоненты пользовательского интерфейса. Рассмотрим каждый уровень.

Уровень представления включает компоненты организации пользовательского интерфейса. Компоненты являются независимыми от выполнения основных задач приложения. Это совокупность страниц доступа к сервисам АИС, которые направляют запросы на уровень предметной области и не взаимодействуют с нижестоящим уровнем хранения данных и уровня сервисов.

Уровень операционный (controller) включает компоненты, которые распределяют задачи между компонентами системы нижнего уровня сервисов. Классы данного уровня не выполняют операции бизнес-логики, а только осуществляют координацию их выполнения.

Уровень предметной (model) области включает подсистемы, которые объединяют проектные классы, и соответствующие интерфейсы, описывающие объекты предметной области. К таким объектам относятся объекты исследовательской деятельности, электронного документооборота и администрирования вычислительных ресурсов.

В рамках спецификации Java 2EE выделен уровень DAO, в функции которого входит осуществление взаимодействия с базой данных, конвертирование данных из объектной модели в реляционную модель. На рисунке 2 представлена многоуровневая архитектура системы в соответствии с описанной моделью MVC. При моделировании декомпозиции

определен состав пакетов в соответствии с реализуемыми функциями системы и параметрами требований. Взаимодействие классов осуществляется посредством удаленного вызова интерфейсов, что позволяет скрыть от разработчика реализацию самого класса, а обращаться к нему посредством вызова реализуемых интерфейсов.

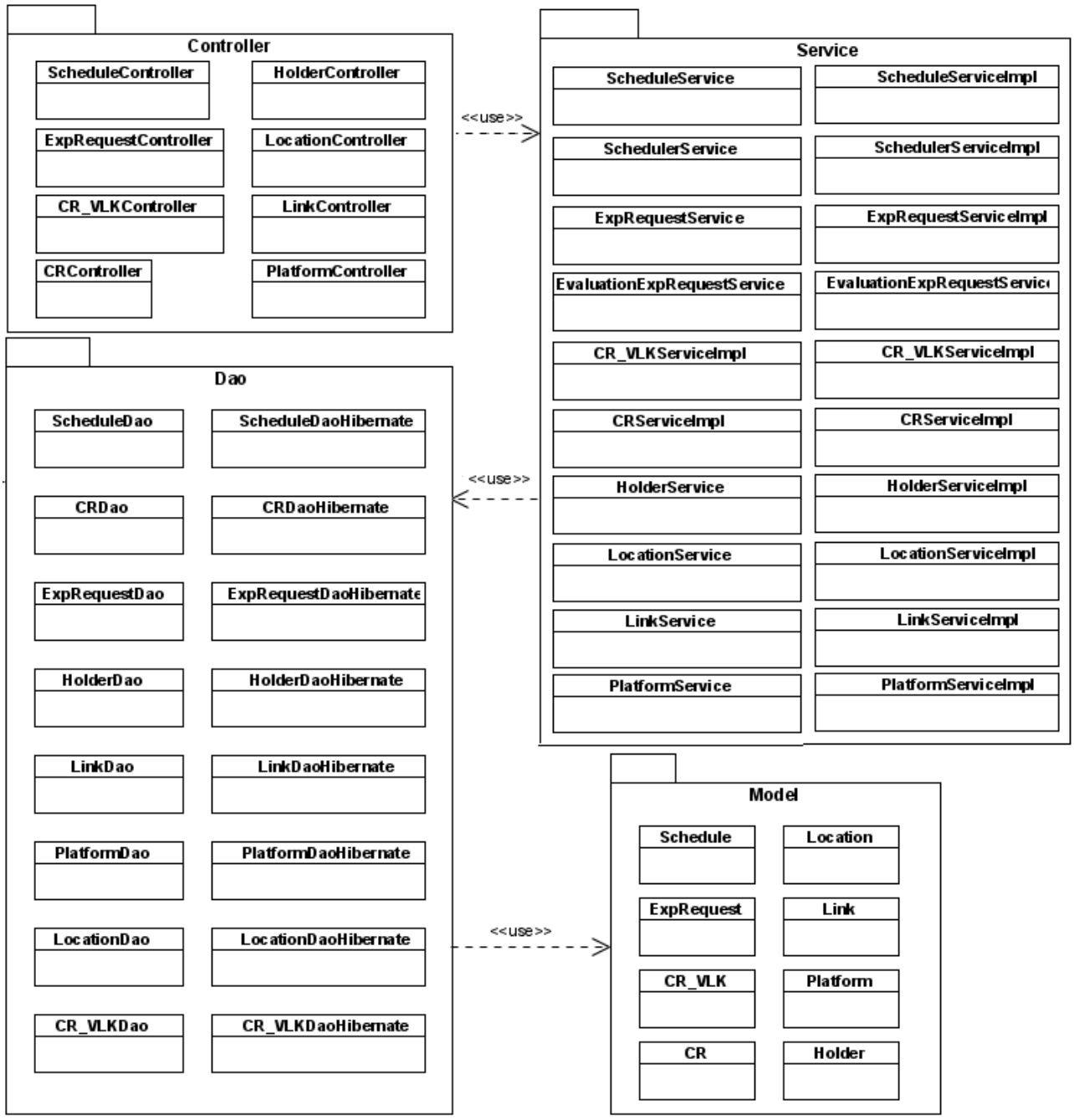


Рис. 2 – Модель АИС в соответствии с моделью MVC

На этапе моделирования реализации системы построена диаграмма размещения компонентов системы по вычислительным узлам в период функционирования. При моделировании размещения программных компонентов необходимо опираться на основные нефункциональные требования к АИС: масштабируемость, расширяемость, отказоустойчивость, высокая доступность, восстанавливаемость, производительность, гибкость.

В первую очередь необходимо, чтобы размещение компонентов по вычислительным узлам соответствовало логике работы автоматизированной системы, а также чтобы выполнялись три наиболее важных требования: возможность масштабирования (линейная и горизонтальная), возможность расширяемости и высокая производительность.

В начале процесса моделирования размещения программных компонентов необходимо определить их связи друг с другом, поток данных при их взаимодействии, и, как следствие, определить предпочтительную пропускную способность канала связи между вычислительными узлами, на которых будут располагаться компоненты. Также необходимо оценить возможность переноса с одного вычислительного узла на другой некоторых компонентов системы без значительных изменений в системе, путем лишь только перенаправления потоков данных. Необходимо продумать, каков будет уровень защищенности системы, как от внешних, так и от внутренних угроз безопасности.

На представленной ниже схеме (рис.3) показана диаграмма размещения комопнентов АИС на вычислительных узлах.

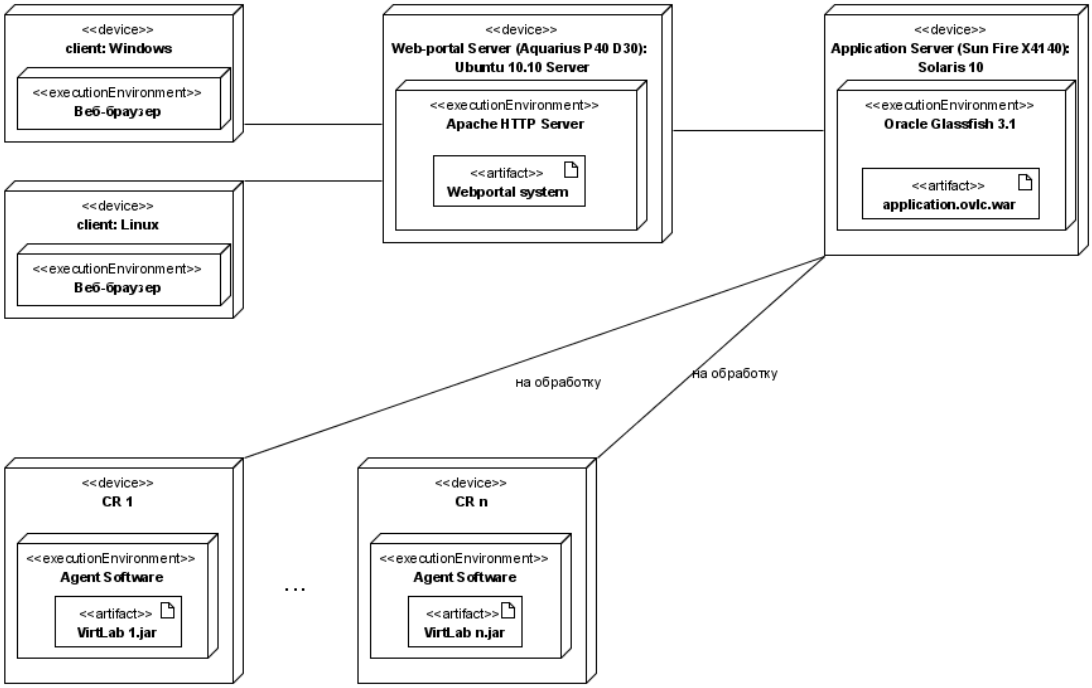


Рис. 3 – Размещение подсистем АИС на вычислительных узлах

Рассмотрим каждый элемент представленной схемыв отдельности.

Сервер приложений (Application Server) – ядро АС ОВЛК, программно-аппаратный комплекс, на котором размещены основные подсистемы. Сервер приложений управляет процессами выполнения необходимых расчётов для проведения исследований, вычислительными ресурсами, а также обрабатывает заявки клиента на эксперимент

Веб-портал (Web-portal Server) – сервер, на котором установлено Web-приложение, обеспечивающее взаимодействие пользователя и АС ОВЛК, принимает запросы пользователя и возвращает результаты обработки запросов, на данном сервере размещена подсистема управления контентом. На ВП размещены основные подсистемы управления контентом.

Вычислительный ресурс (CR) – предназначен для выполнения расчётов, задание на проведение которых выдает сервер приложений. На каждом подключенном к системе вычислительном узле устанавливается агент (Agent Software) задачей которого является сбор всей необходимой информации о данном вычислительном узле, оценка вычислительной мощности, получение, выполнение и выгрузка данных задания полученного с сервера приложений.

Клиент – устройство, обладающее достаточными ресурсами, возможностью взаимодействовать с АС ОВЛК посредством стандартных, открытых интерфейсов, для формирования заявки к службе на исполнение, а также на получение результата работы.

Выводы

Моделирование системы позволило оценить варианты архитектур, состав компонентов и схему их взаимодействия и реализовать для проведения предварительного тестирования прототипа.

Используемые технологии распределения вычислительных ресурсов позволяют предоставить доступ к мощным вычислительным ресурсам в виртуализированном виде, агрегируя имеющиеся ИКТ-ресурсы и всю вычислительную мощность в единой точке доступа к ним. Пользователь может самостоятельно обеспечивать себя вычислительными возможностями (средствами и ресурсами), такими как серверное время и сетевые хранилища, по мере необходимости запрашивая их у сервис-провайдера в одностороннем автоматическом режиме. Преимущества использование «облачных» технологий в исследовательских задачах очевидным образом достигается за счет следующих сервисов:

- Запрашиваемые сервисы доступны по сети через стандартные механизмы, поддерживающие использование гетерогенных платформ тонких и толстых клиентов (например, мобильных телефонов, ноутбуков и КПК).
- Вычислительные ресурсы провайдера услуг организованы в виде пула для обслуживания различных потребителей в модели множественной аренды с возможностью динамического назначения и переназначения различных физических и виртуальных ресурсов в соответствии с потребностями потребителей.
- Вычислительные возможности могут быть предоставлены быстро и эластично из изменяемого объема ресурсов. Для потребителя эти ресурсы часто предоставляются как доступные в неограниченном объеме и могут быть приобретены в любой момент времени в любом количестве.
- Система автоматически контролирует и оптимизирует использование ресурса, измеряя его на определенном уровне абстракции, соответствующем типу использующего его сервиса для конечного потребителя (например, объема хранения, вычислительной мощности, полосы пропускания и активных учетных записей пользователей). Использование ресурсов может подвергаться мониторингу, быть контролируемым и сопровождаться отчетностью, обеспечивая прозрачность потребления как для провайдера, так и для потребителя использованного сервиса.

Литература

1. Интернет-ресурс: «Портал информационной поддержки cloud computing » http://cloud.zapeecee.ru/home
2. Ефимов И.Н., Козлова С.Ж., Жукова С.А. Автоматизированная система интеграции открытых виртуальных лабораторных комплексов // XIII Международная научно-практическая конференция «Фундаментальные и прикладные проблемы приборостроения и информатики», Московский государственный университет приборостроения и информатики, С. 105-110.
3. Арлоу, Нейштадт. UML 2 и Унифицированный процесс: практический объектно-ориентированный анализ и проектирование. – Символ-Плюс, 2-е изд. – 2007. – с. 624
4. К. Ларман. Применение UML 2.0 и шаблонов проектирования. – Вильямс, 3-е изд. – 2007. – с.730

УДК 004.04

ОБЪЕКТНАЯ МОДЕЛЬ ПРЕДСТАВЛЕНИЯ ПРАВИЛ ФОРМИРОВАНИЯ СОСТАВНЫХ ЧАСТЕЙ РЕЧИ, ИСПОЛЬЗУЕМЫХ ПРИ МОРФОЛОГИЧЕСКОМ АНАЛИЗЕ СЛОВ РУССКОГО ЯЗЫКА

Голда Анна Анатольевна, к.ф.н., Лингвист, Переводчик, ОАО «ТАНТК им. Г.М. Бериева», Россия, Таганрог, anya.golda@rambler.ru

Олейник Павел Петрович, к.т.н., Системный архитектор программного обеспечения, ОАО «Астон», Россия, Ростов-на-Дону, xsl@list.ru

В настоящее время все больше внимания уделяется задаче реализации полнотекстового тематического поиска данных, хранящихся в информационной системе. Создание таких программных комплексов на сегодняшний день продиктовано, во-первых, возникновением огромного объема информации, которую необходимо поддерживать в актуальном состоянии, во-вторых, развитием информационного пространства, растущего быстрыми темпами. Подобные информационные системы должны отвечать требованиям пользователей и удовлетворять их потребности, возникающие при поиске информации.

Вопрос о создании интерфейса к программе на естественном языке или интеллектуального поиска информации, отвечающего требованиям и критериям пользователя, является актуальным и достаточно сложным. При создании информационных систем используется компьютерная морфология, позволяющая выполнять поиск и анализ информации на естественном языке. Основные функции, которые должен обеспечивать морфологический анализатор поисковой системы, – это получение всех словоформ слова, постановка слова в заданную форму и определение грамматических характеристик словоформы.

Базовой проблемой при реализации подобного функционала является задача проведения морфологического анализа введенных слов. В данной статье рассмотрено одно из возможных решений и подробно описана иерархия классов, которая была спроектирована для представления правил формирования составных частей речи из различных типов морфем (простых элементов языка).

Проектирование любой информационной системы начинается с формулирования критериев оптимальности, которым должна соответствовать полученная реализация. Для рассматриваемой задачи были сформулированы следующие критерии оптимальности (КО):

1. Разработка корректной иерархии представления, как простых элементов языка (морфем), так и составных частей речи (существительное, прилагательное и т.п.), которые состоят из простых элементов.
2. Унифицированное представление правил формирования составных частей речи естественного языка.
3. Отсутствие необходимости поддержания полного словаря, содержащего все возможные вхождения каждого слова.

Рассмотрим каждый из выделенных критериев более подробно. КО1 позволяет унифицировано обрабатывать все имеющиеся элементы языка. Это позволяет строить (в соответствии с правилом подстановки, применяемым в объектно-ориентированных языках программирования) единые итераторы, которые используются в процессе проведения морфологического анализа.

Выполнение требования КО2 разрешает пользователю самостоятельно создавать и дорабатывать (что наиболее важно) правила русского языка без необходимости создания новых классов.

Реализация требования КО3 позволяет упростить процесс поддержания словаря данных в актуальном состоянии в базе данных. Т.е. для каждого слова достаточно внести лишь его корень, т.к. это основная часть. В результате, при проведении морфологического анализа составная часть речи, представляющая введенное слово, может быть получена на основании имеющегося в БД корня и введенных морфем (приставок, суффиксов, окончаний и т.п.). Достоинством данного подхода является не только относительно небольшой объем базы данных, но и возможность получения любых словарных форм (даже не существующих в русском языке) на основании имеющейся информации.

На рис. 1 представлена диаграмма классов в нотации UML [1], которая удовлетворяет всем выделенным критериям. Корневым классом всей иерархии является абстрактный класс LanguageConstruction, который содержит единственный атрибут Name. Рассматриваемый атрибут необходим для представления как морфем, так и названий составных частей речи, используемых в программе.

Базовым классом для всех морфем выступает абстрактный класс Morpheme, который

является корневым для всех остальных (не представленных на данном рисунке). Т.е. классы морфем, такие как, например, приставка, корень, окончания и т.п., унаследованы от рассматриваемого. При этом ряд морфем, (например, окончание «ти») могут иметь базовую морфему (в нашем случае окончание «ть»). Именно для этого используются соответствующие атрибуты класса Morpheme.

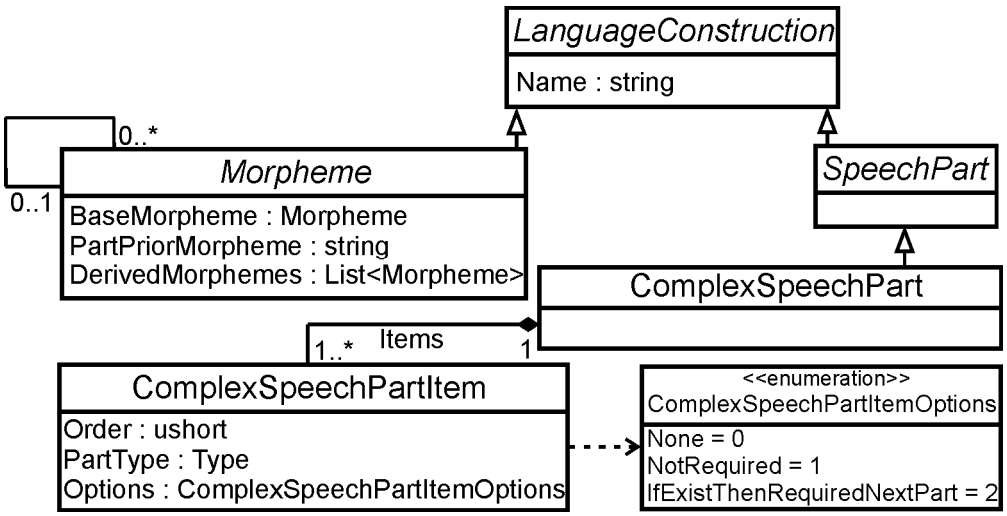


Рис. 1 – UML-диаграмма классов, используемая при морфологическом анализе

Корневым классом, представляющим все части речи, является абстрактный класс SpeechPart. В данной статье рассматриваются лишь сложные (составные) части речи, которые представлены экземплярами неабстрактного класса ComplexSpeechPart и являются правилами их формирования. Каждый элемент правила представлен объектом класса ComplexSpeechPartItem и содержит атрибут Order, предназначенный для указания порядка следования в слове и для описания типа элемента. Типом в нашем случае является название класса морфем, присутствующих в системе. Определенные элементы такие как, например, приставка, могут отсутствовать в слове, поэтому они являются необязательными. Другие элементы, например, интерфикс, (если он присутствуют в слове), требуют наличия следующих после себя частей. Именно для описания подобных ситуаций используется множество опций, определённых в перечислении ComplexSpeechPartItemOptions.

Из представленной на рисунке диаграммы классов видно, что она соответствует выделенным ранее критериям оптимальности, поэтому можно приступать к наглядному представлению вводимых правил, что проще всего сделать с помощью диаграммы объектов языка UML [1], которая изображена на рис. 2.

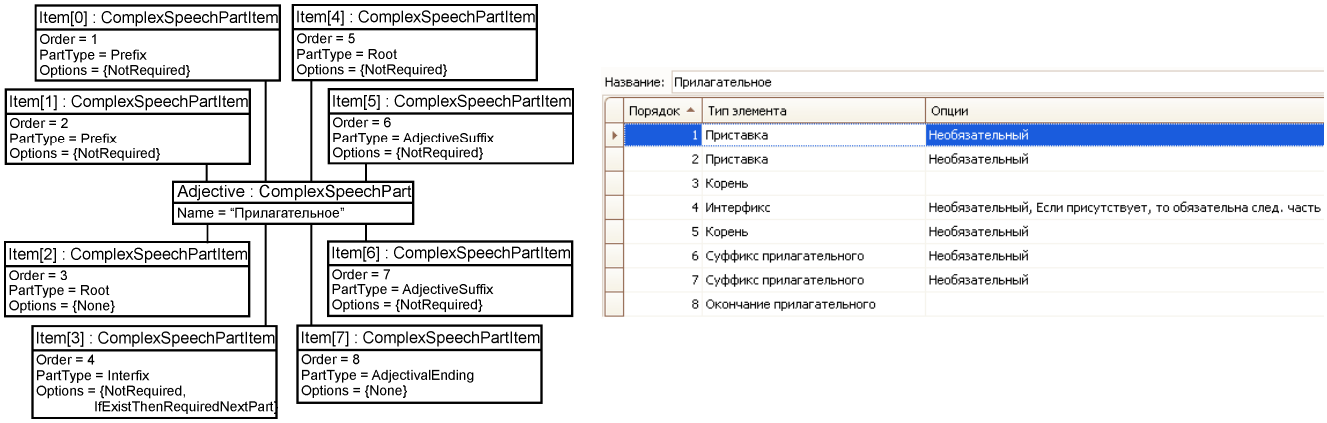


Рис. 2 – Представление правила формирования для части речи «Прилагательное»:
А) UML-диаграмма объектов; Б) Графический интерфейс приложения

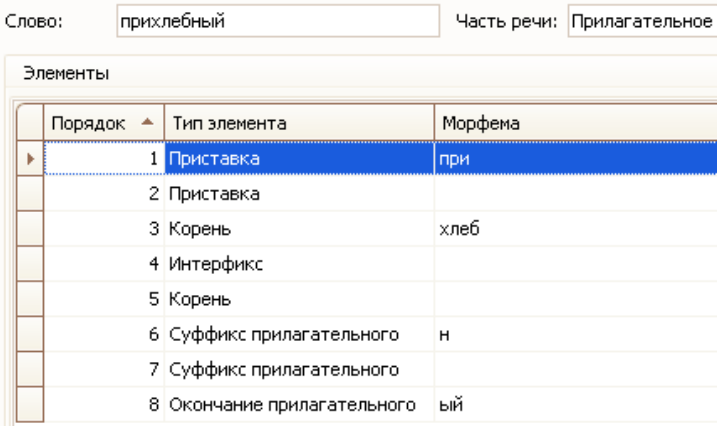
Из рис. 2 видно, что создать правило формирования составной части речи в разрабатываемой системе пользователю не составит особого труда. При этом наличие интуитивно-понятного графического интерфейса позволяет иметь унифицированное

представление правил, которые можно изменить без перекомпиляции самого приложения. Данный подход позволяет утверждать, что удовлетворены требования КО2.

Алгоритм проверки введенного слова и определение составной части, которой оно соответствует, – довольно тривиальная задача, решение которой заключается в выполнении последовательности следующих шагов:

1. Получение всех возможных морфем, входящих в рассматриваемое слово.
2. Организация цикла по имеющимся правилам формирования для каждой составной части речи, каждая итерация которого состоит из следующих подшагов:
   2.1. Получение всех морфем, которые в соответствии с правилом могут присутствовать в рассматриваемой части речи.
   2.2. Проверка наличия всех обязательных морфем в слове (например, для прилагательного обязательно наличие корня и окончания прилагательного). Если не все обязательные морфемы присутствуют, то переходим к следующей итерации цикла.
   2.3. Организация цикла по элементам правила в возрастающем порядке следования и попытка построения рассматриваемого слова с помощью имеющихся морфем. Если исходное слово не получено, то переходим к следующей итерации и обрабатываем следующую составную часть речи, для которой определено правило формирования. Если слово сформировано, то прекращаем итерации и возвращаем результат разбора.
3. Вывод результатов, соответствующих заданному правилу для найденной части речи, с указанием имеющихся и отсутствующих морфем.

На рис. 3 представлен результат реализации описанного алгоритма в виде графической формы приложения.



Порядок	Тип элемента	Морфема
1	Приставка	при
2	Приставка	
3	Корень	хлеб
4	Интерфикс	
5	Корень	
6	Суффикс прилагательного	н
7	Суффикс прилагательного	
8	Окончание прилагательного	ый

Рис. 3 – Результат морфологического анализа прилагательного "прихлебный"

Из рисунка видно, что результат полностью удовлетворяет ожиданиям (формирующее правило сопровождается найденными морфемами), поэтому можно утверждать, что алгоритм реализован верно, и представленная модель реализации правил формирования составных частей речи успешно себя зарекомендовала. При этом в БД присутствуют только корень рассматриваемого слова и отсутствуют все возможные формы. Т.е. можно утверждать, что при реализации выполнено требование КО3, т.к. в системе нет необходимости поддержания полного словаря.

В заключение статьи отметим, что дальнейшим развитием данной работы является доработка иерархии для реализации функционала представления исключений русского языка и для возможности выполнения синтаксического анализа целых фраз и предложений.

Литература

1. Новиков Ф.А., Иванов Д.Ю. Моделирование на UML. Теория, практика, видеокурс. – СПб.: Профессиональная литература, Наука и Техника, 2010. – 640 с.: ил. + цв. Вклейки (+ 2 DVD).

УДК 004.8

АРХИТЕКТУРА СИСТЕМЫ ИНФОРМАЦИОННОГО ПОИСКА С ИСПОЛЬЗОВАНИЕМ ОНТОЛОГИИ ПРЕДМЕТНОЙ ОБЛАСТИ

Крылов Александр Юрьевич, студент, Ивановский государственный химико-технологический университет, Россия, Иваново, krylov_al@mail.ru
Галиаскаров Эдуард Геннадьевич, к.х.н., доцент, доцент, Ивановский государственный химико-технологический университет, Россия, Иваново, galiaskarov@isuct.ru

Под информационным поиском обычно понимают процесс поиска неструктурированной документальной информации, а также науку об этом поиске. В общем случае поиск информации включает в себя четыре этапа:

- определение и уточнение информационной потребности и формулировка информационного запроса;
- определение совокупности возможных держателей информационных массивов;
- извлечение информации из выявленных информационных массивов;
- ознакомление с полученной информацией и оценка результата поиска.

Таким образом, центральная задача информационного поиска состоит в том, чтобы помочь пользователю удовлетворить свою информационную потребность. Описание информационных потребностей само по себе технически сложно и во многом субъективно, поэтому они формулируются в виде некоторого запроса, содержащего набор понятий, который определяет то, что ищет пользователь. В работе [1] для уточнения информационной потребности и составления поискового образа запроса, достаточно релевантного запрашиваемому документу, предлагается использовать онтологию предметной области. Применение онтологии позволяет выявить своего рода смысл запроса, используя семантические связи между содержащимися в нем терминами, поэтому одной из основных задач в разработке подобной поисковой системы является максимально точное, при помощи онтологии, описание предметной области, представленной базой данных документов, по которым осуществляется поиск. Именно это и позволит максимально повысить эффективность использования онтологии для решения задачи информационного поиска в рамках заданной предметной области. Ранее [2] мы использовали этот подход при описании общих принципов организации поисковой системы на основе онтологии.

В этой статье мы хотели бы представить вариант архитектуры системы информационного поиска с использованием онтологии предметной области и сформулировать ряд рекомендаций по построению такой системы.

Общая идея использования онтологии для уточнения и формулировки запроса заключается в следующем. Пусть пользователь формулирует запрос «ОПЛАТА ЖИЛЬЯ», предполагая обнаружить сведения о различных способах оплаты при покупке жилья в свою собственность. Допустим, что он будет иметь возможность использовать одну или более различных онтологий предметной области, в частности связанных с оплатой жилья. Задача построения такой онтологии технически решается при помощи специального программного обеспечения (например, Protege [3]), позволяющего анализировать текст документа и выделять из него наиболее часто встречающиеся и важные слова, соответствующие описываемой в документе тематике. Примечательно, что построение такой онтологии будет проходить либо под руководством эксперта, либо самим экспертом или группой экспертов в определенной предметной области. Текущий тестовый вариант онтологии предметной области «Государственное управление», тематика «Жилищная политика» разработана специалистами с целью использования в процессе разработки и тестирования алгоритмов онтологической обработки запроса. Она содержит 38 классов и около 50 связей различных типов между ними. Фрагмент данной онтологии приведен на рис. 1.

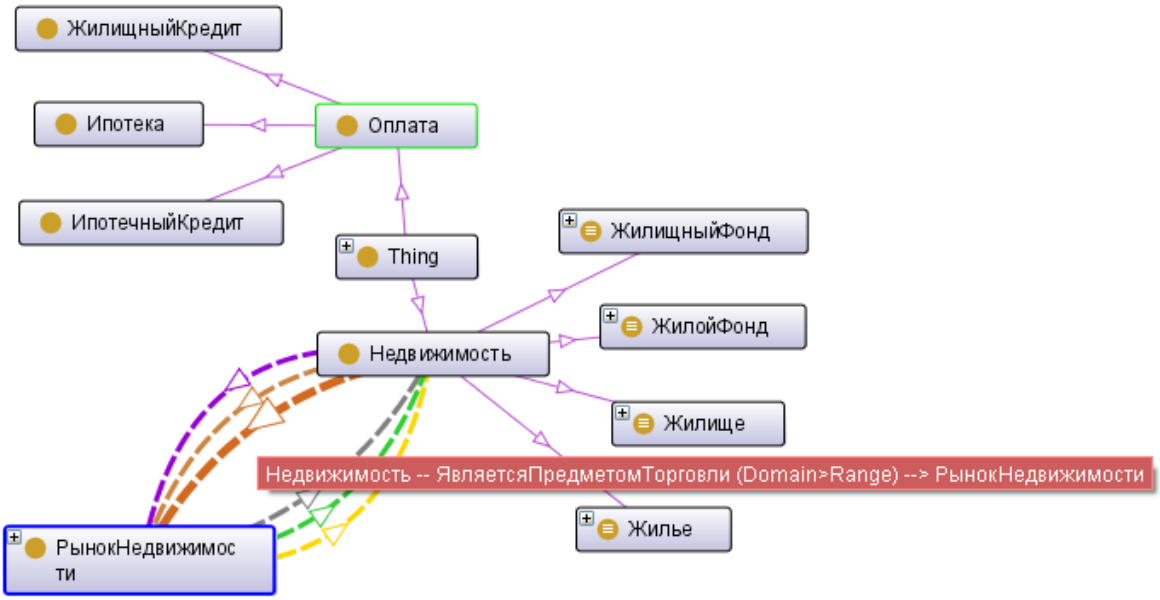


Рис. 1 – Фрагмент тестовой онтологии предметной области

Контекст использования системой онтологии при подготовке запроса и выполнения поиска представлен на рисунке 2. На вход системы информационного поиска подается исходный текст запроса, указанные пользователем онтологии и список элементов онтологии. Под элементами онтологии здесь подразумеваются суперклассы, подклассы и эквивалентные классы. Указание этих элементов позволяет управлять глубиной и широтой поиска в рамках заданных онтологий. В ходе обработки поступившей информации производится подготовка расширенного запроса (или нескольких запросов, по одному для каждой онтологии предметной области), сам поиск и предоставление результатов поиска пользователю, т.е. списка релевантных запросу документов.

Рис. 2 – Контекст использования онтологии

Результат применения приведенной ранее онтологии на стадии уточнения запроса пользователя («Оплата жилья») может выглядеть так: «Оплата жилья, Оплата недвижимости, Оплата жилища, Жилищный кредит жилья, Жилищный кредит недвижимости, Жилищный кредит жилища, Ипотека жилья, Ипотека недвижимости, Ипотека жилища, Ипотечный кредит жилья, Ипотечный кредит недвижимости, Ипотечный кредит жилища» – т.е. помимо исходного набора понятий добавляются семантически связанные понятия, с учетом указанной глубины и широты поиска, повышая, таким образом, общую пертинентность поиска.

Как мы уже определяли в работе [2], система информационного поиска должна обеспечивать выполнения таких функций как:

- морфологическая обработка запроса;
- онтологическая обработка запроса;
- формирование расширенных запросов;
- поиск документов по запросам;
- формирование списка результатов.

Как показал архитектурный анализ и изучение имеющихся на рынке решений, нет нужды разрабатывать собственные компоненты, реализующие морфологический разбор и нормировку исходного запроса пользователя. Также гораздо эффективнее использовать готовые поисковые машины на базе классических методов поиска. Гораздо проще и разумнее сосредоточить внимание на модуле онтологической обработки запроса и обеспечить в системе высокую степень использования готовых компонентов сторонних производителей, а также соответствие широко используемым стандартам и форматам данных. В данном случае для хранения онтологий используется формат OWL (Ontology Web Language) – специализированный для хранения данных онтологии XML-формат [4]. Для осуществления морфологического анализа исходного запроса пользователя можно применить хорошо зарекомендовавший себя технологический комплекс АЛОТ – Автоматизированная Лингвистическая Обработка Текстов [5]. В качестве поисковой машины предполагается использовать открытый полнотекстовый поисковый сервер Sphinx [6].

К вышеперечисленным требованиям можно добавить, что разрабатываемая система должна обеспечить:

- полное отделение логики организации поиска от данных, участвующих в этом процессе (онтологии, документов, морфологии) и возможность использования различных подходов к организации схем предметной области (онтология предметной области, диаграмма классов, глоссарий, тезаурус и т. д.) в зависимости от целей поиска;
- дискретность процесса поиска, предоставляющую возможность отслеживания процесса поиска на каждом из этапов прохождения поискового запроса в системе и возможность его модификации путем добавления/удаления соответствующих функциональных модулей.

Реализация системы выполнена в соответствии с трехуровневой архитектурой программного обеспечения, в которой выделяются: слой интерфейсов (GUI, API), слой бизнес-логики процесса поиска (поисковое ядро и функциональные модули) и слой интерфейсов доступа к данным (источники данных онтологии, морфологии, базы данных документов, поискового движка). Диаграмма компонентов системы, распределенных по слоям (пакетам) в соответствии с вышеописанной архитектурой, приведена на рисунке 3.

Разработанная архитектура позволяет реализовать вышеописанные требования к системе, а также предоставляет и некоторые другие преимущества:

- модифицируемость системы, обеспечивающая возможность изменения и замены отдельных компонентов системы без затрагивания остальных;
- открытость системы, позволяющая интегрировать в нее уже реализованные компоненты, такие как СУБД, поисковая машина, редактор онтологии, сервис морфологии и т.д.;

С учётом модульной организации системы, процесс поиска является дискретным, при этом каждый из подпроцессов обработки поискового запроса реализуется конкретным функциональным модулем:

- модуль морфологической обработки запроса;
- модуль онтологической обработки;
- модуль формирования расширенных дополнительных запросов;
- модуль поиска;
- модуль обработки результатов поиска (документов).

Таким образом, процесс обработки поискового запроса можно модифицировать, добавляя и удаляя соответствующие функциональные модули.

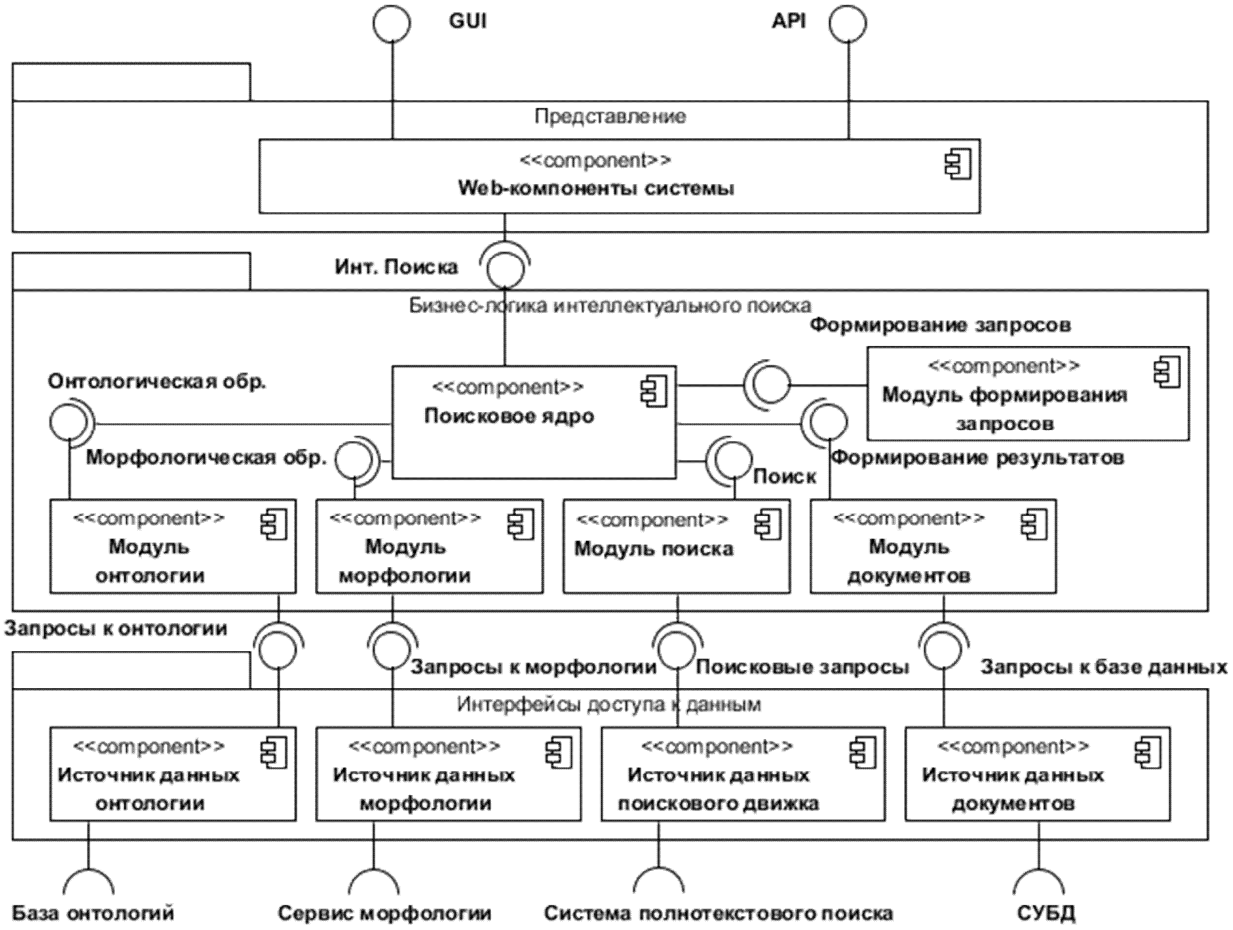


Рис. 3 – Диаграмма компонентов разрабатываемой системы

Каждый из функциональных модулей обработки поискового запроса использует собственный источник данных, предназначенный для предоставления модулю всех необходимых для выполнения его основной задачи данных. Такие источники данных взаимодействуют с:

- сервисом морфологии;
- базой онтологий;
- системой полнотекстового поиска;
- СУБД документов.

Взаимодействие происходит при помощи соответствующих программных интерфейсов, предоставляемых данными внешними системами.

Наиболее важным для нас является процесс онтологической обработки запроса, сосредоточенный в соответствующем модуле системы. Реализация данного этапа обработки запроса осуществляется модулем онтологии, при помощи соответствующего источника данных. Алгоритм выполнения онтологической обработки запроса приведен на рисунке 4.

Алгоритм данного сценария заключается в следующем.

Модуль онтологии получает управление и необходимую для выполнения онтологической обработки запроса информацию:

- нормализованные элементы исходного поискового запроса;
- список используемых для расширения запроса онтологий;
- список искомых в данных онтологиях элементов.

При получении управления модуль отправляет команду на получение соответствующей онтологии из базы онтологий.

В результате этого компонент источника данных онтологии извлекает из базы данных указанную онтологию, регистрирует ее и создает ее программный объект класса OWL

Онтология (OWLOntology), обеспечивая доступ к ней на протяжении процесса преобразования запроса.

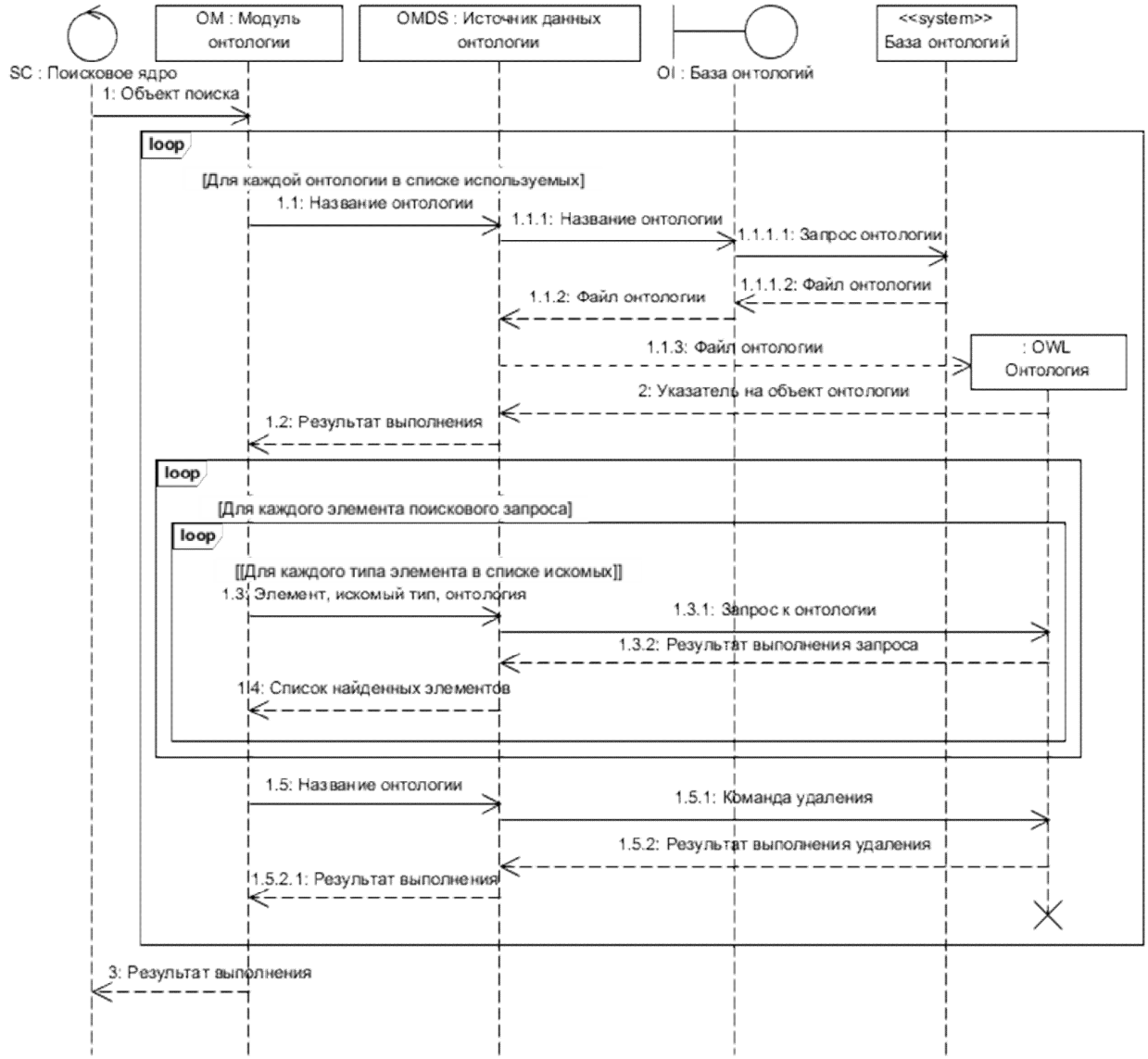


Рис. 4 – Архитектурнозначимый сценарий использования системы поиска

После этого производится выполнение ряда запросов к онтологии модулем онтологической обработки запроса, в результате чего формируется матрица расширенных элементов запроса. Данные запросы направлены на получение элементов онтологии, связанных с элементами исходного поискового запроса указанными типами связей – подклассы, эквивалентные классы, суперклассы.

Такие действия выполняются циклически – для всех элементов запроса, всех онтологий и всех указанных типов искомых элементов. Полученные в результате выполнения всех вышеописанных действий данные возвращаются поисковому ядру и передаются модулю формирования расширенных запросов с целью генерации новых (расширенных) поисковых запросов, по которым в дальнейшем производится поиск документов.

Следуя логике предложенной архитектуры (рис. 3), для осуществления поиска на уровне представления система предоставляет единый основной интерфейс, используемый для инициализации процесса обработки пользовательского поискового запроса. Это может производиться как средствами пользовательского графического интерфейса системы, так и средствами API-интерфейсов, также предоставляемых системой для использования смежными системами.

Учитывая модульную организацию системы, возможно использование разработанных модулей системы отдельно, например, при реализации функционала какой-либо другой системы, нуждающейся в морфологическом анализе, онтологической обработке текстовых запросов и других видах обработки информации, реализуемой в нашей системе. Отметим также, что разработанную систему можно полностью интегрировать в другие информационные системы, определив необходимые для ее функционирования интерфейсы. Например, это могут быть системы, предоставляющие доступ к различным коллекциям документов, внедрение в которые позволит производить интеллектуальный поиск по этим документам.

Литература

1. Россеева О. И., Загорулько Ю.А. Организация эффективного поиска на основе онтологий // Труды Международного семинара Диалог'2001 по компьютерной лингвистике и ее приложениям, т.2, 2001. – [Электронный ресурс]: [Статья]. – Режим доступа: http://www.dialog-21.ru/materials/archive.asp?id=7029&y=2001&vol=6078
2. Krylov A.Yu., Galiaskarov E.G. The basic organization of an ontology driven retrieval system. Object Systems – 2011 (English session): Proceedings of the Fourth International Theoretical and Practical Conference. Rostov-on-Don, Russia, 10-12 November, 2011. Edited by Pavel P. Oleynik. – p. 11-16. Также: [Электронный ресурс]: [Статья] – Режим доступа: http://objectsystems.ru/files/2011ES/Object_Systems_2011_English_session_Proceedings.pdf
3. Редактор онтологии и шаблон построения базы знаний “Protege”. – [Электронный ресурс]: [Сайт продукта] – Режим доступа: http://protege.stanford.edu/
4. Спецификация языка веб-онтологий OWL. – [Электронный ресурс]: [Страница спецификации] – http://www.w3.org/TR/owl-features/
5. Технологический комплекс АЛОТ – Автоматизированная Лингвистическая Обработка Текстов. – [Электронный ресурс]: [Сайт комплекса] – http://www.cir.ru/docs/ips/techno/index.htm
6. Полнотекстовый поисковый сервер Sphinx. – [Электронный ресурс]:[Сайт продукта] – http://sphinxsearch.com/

УДК 681.3

РЕДАКТОР МЕТАМОДЕЛИ ОНТОЛОГИЧЕСКОЙ СИСТЕМЫ

Грегер Сергей Эдуардович, доцент, Уральский федеральный университет имени первого Президента России Б.Н.Ельцина, Нижнетагильский технологический институт (филиал), Россия, Нижний Тагил, segreger@gmail.com

Рост объема информационных потоков породил тенденцию смещения разработок в направлении крупных информационных систем – корпоративных порталов. В решении проблем, связанных с эффективностью проектировании, реализации и сопровождении порталов, все шире используются возможности, предоставляемые семантическими моделями на базе онтологии (онтологические модели). Для создания и поддержки онтологий существует целый ряд инструментов, которые помимо общих функций редактирования и просмотра выполняют поддержку документирования онтологий, импорт и экспорт онтологий разных форматов и языков, поддержку графического редактирования, управление библиотеками онтологий и т.д. В то же время появилась тенденция к поддержке онтологических моделей в системах управления контентом (CMS), являющихся основой для построения современных веб-приложений. Такая поддержка в настоящее время в определенной степени реализована в CMS Drupal, Django и некоторых других. Использование таких моделей переводит веб-приложения в категорию онтологоуправляемых информационных систем, что требует разработки соответствующих инструментов управления.

Редактор онтологий – это приложение, поддерживающее создание онтологий или манипулирование ими [1]. Онтологический редактор может обеспечивать различные возможности работы с одним или несколькими языками представления онтологий, импорт/экспорт, средства библиотек онтологий, визуализацию, машины вывода, языки запросов, поддержу онтологий верхнего уровня и многое другое. Основная функция любого редактора онтологий состоит в поддержке процесса формализации знаний и представлении онтологии как спецификации (точного и полного описания). Существующие веб-редакторы онтологий, в частности, веб-версия хорошо известного редактора Protégé — WebProtege, — предназначены прежде всего для поддержки совместной разработки онтологий и непосредственно не могут быть использованы в качестве инструмента в составе онтологоуправляемой информационной системы.

Наши исследования направлены на изучение особенностей построения корпоративных порталов на основе CMS Plone [2,3] с использованием онтологических моделей. Основной особенностью, отличающей эту систему разработки от других CMS, является использование в качестве хранилища контента объектной базы Zope Object Database (ZODB). Таким образом, целью нашей разработки являлось создание редактора онтологии, хранимой в объектной базе ZODB. При разработке редактора ставилась задача создания web-приложения, позволяющего как создавать новые, так и редактировать уже существующие онтологии. При этом предполагалось, что режим доступа к редактируемой онтологии может быть как online, так и offline. Кроме этих основных требований к приложению был предъявлен еще ряд как функциональных, так и нефункциональных требований:

1. Редактор должен обладать возможностью загружать и визуализировать любую представленную на языке OWL онтологию.
2. Редактор должен предоставлять пользователю дружественный интерфейс, позволяющий быстро создавать и редактировать концепты онтологии.
3. Редактор должен предоставлять гибкий поисковый механизм, позволяющий легко искать и редактировать концепты онтологии.

Для реализации предъявленных требований была выбрана технология AJAX, позволяющая создавать интерактивные web-приложения, реализующие асинхронное взаимодействие между клиентской и серверной частями приложения.

Редактор реализован на языке Javascript с использованием библиотеки JQuery. Для эффективного управления онтологиями на стороне клиента реализована система классов, обеспечивающих хранение редактируемой части онтологии в оперативной памяти, а также построения запросов к серверу извлечения данных об элементах различных элементах онтологии.

Редактор состоит из клиентской и серверной части. Для эффективного клиент-серверного взаимодействия серверная часть приложения организована в виде JSON-сервера запросов. Сервер реализует выполнение различного вида запросов к объектной базе и формирование результата в структуру данных, пересылаемую редактору в формате JSON. Он реализован компонентом Ontology Service, реализующим все алгоритмы, связанные с загрузкой и управлением онтологией и набором классов, обеспечивающих хранение и обработку концептов онтологии. Компонент предоставляет набор взаимосвязанных сервисов, каждый из которых обеспечивает обработку специфических концептов онтологии и коммуникацию с клиентской частью редактора. Серверная часть приложения предоставляет следующие сервисы:

1. Загрузку OWL-онтологии из удаленного источника.
2. Выполнение запроса на получение всей онтологии или ее отдельной части, определяемой указанным частичным классом онтологии и формирование соответствующего JSON-пакета.
3. Поиск всех онтологических классов, являющихся дочерними классами указанного концепта онтологии.
4. Поиск всех объектов указанного класса.

5. Предоставление информации о структуре указанного класса, объекта или свойства.
6. Отображение всех метаданных загруженной онтологии.

Объектная модель компонентов представлена на рисунке 1.

Рис. 1 – Объектная модель компонентов

Для хранения онтологии в объектной базе данных разработан набор компонент [4], обеспечивающих представление концептов языка OWL, таких как *Class*, *Data Property*, *Object Property* и их атрибутов. Для хранения индивидуальных объектов онтологии предназначен компонент *Individual*. Использование таких компонент позволило эффективно хранить онтологию и реализовать выполнение запросов к онтологии. При создании и модификации элементов онтологии сервер запросов манипулирует объектами указанных классов, используя средства управления объектной базой данных.

Клиентская часть редактора выполнена с использованием AJAX-технологий и функционирует в соответствии с клиентской моделью онтологии. Модель содержит классы, обеспечивающие выполнение запросов к сервисной части редактора, хранение онтологии на стороне клиента, а также построение графического пользовательского интерфейса (рис.2).

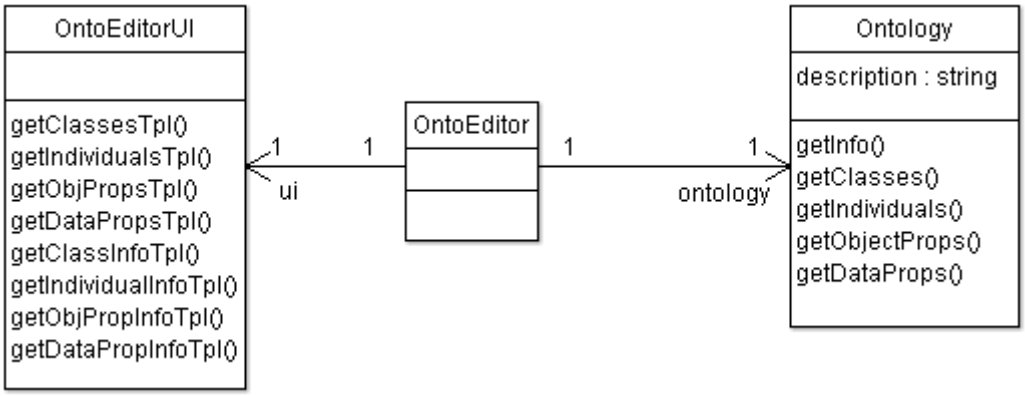


Рис. 2 – Клиентская модель онтологии

Класс Ontology обеспечивает хранения онтологии в виде многоуровневых массивов и предоставлет методы доступ к элементам онтологии;

Класс OntoEditorUI предназначен автоматической генерации форм, шаблонов на основе структуры данных, полученных из онтологии;

Класс OntoEditor является связующим звеном между уровнем хранения данных (класс Ontology) и уровнем представления данных (класс OntoEditorUI) и содержит в себе всю бизнес-логику редактора.

Пользовательский интерфейс редактора построен с использованием паттерна «portlet». Этот паттерн предлагает декомпозировать страницу веб-приложения на ряд визуальных компонент — портлетов, каждый из которых обладает собственными моделями хранения, обработки и отображения данных определенного типа. Использование этого паттерна соответствует принятой в CMS Plone концепции пользовательского интерфейса, хотя и имеет некоторые отличия. Так портлет отображения дерева элементов онтологии в «стандартном» интерфейсе CMS Plone может быть размещен только в левой или правой колонке страницы при ее «трехколоночной» разметке. В представленном редакторе указанный портлет расположен в центральной части страницы. И хотя такое построение позволяет использовать редактор отдельно от CMS Plone, при наличии соответствующей реализации серверной части редактора, в настоящее время активно разрабатывается версия редактора, использующего специфические особенности указанной CMS.

Интерфейс редактора включает в себя нескольких портлетов, отображающих иерархическую структуру классов онтологии, список свойств, список индивидулов классов и индивидуалов свойств. В каждом портлете имеется кнопка, позволяющая вызывать форму создания объекта, тип которого соответствует типу отображаемых портлетом элементов. В центральной части редактора отображаются свойства выбранного элемента онтологии. Реализация интерфейса редактора представлена на рисунке (рис.3).

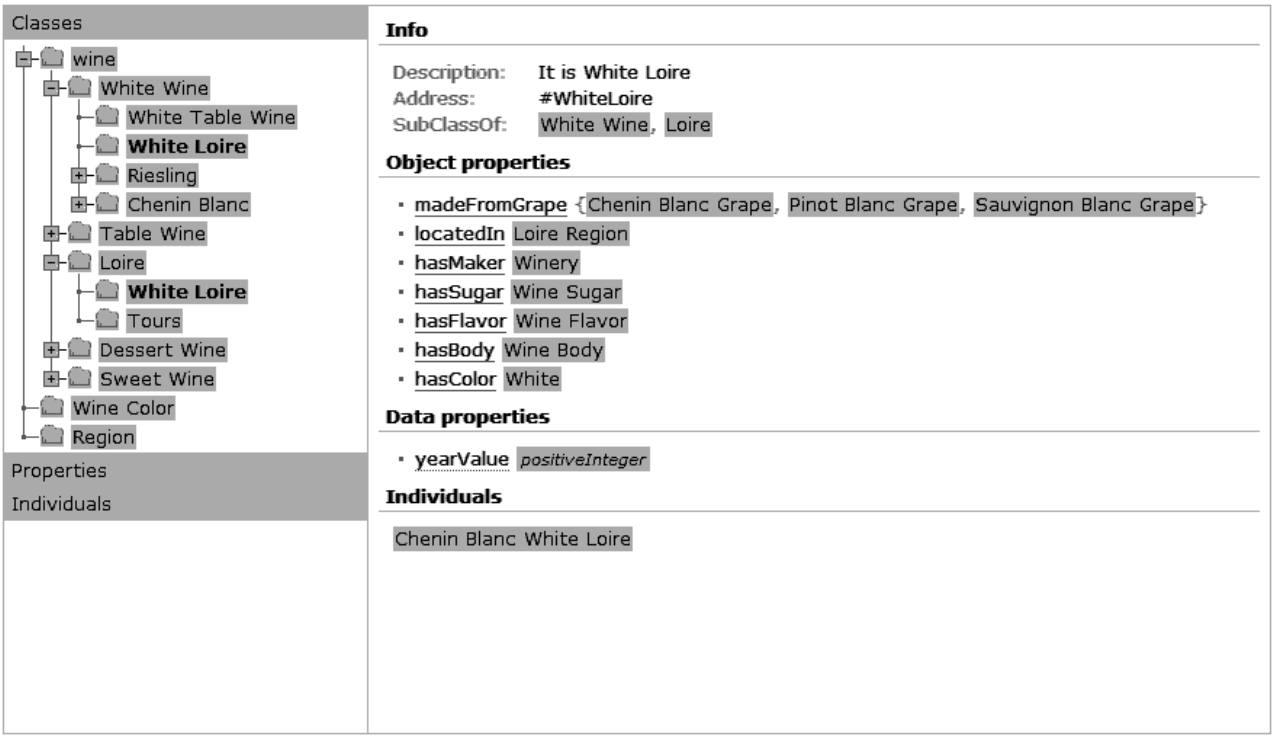


Рис. 3 – Интерфейс клиентской части редактора

Разработанный редактор используется в составе инструментальной среды семантического моделирования учебного процесса [5].

Литература

1. Овдий О.М., Проскудина Г.Ю. Обзор инструментов инженерии онтологий. – http://www.elbib.ru/index.phtml?page=elbib/rus/journal/2004/part4/op
2. Грегер С.Э. Сервер приложений «Zope». Учебное пособие для вузов. – М.: Горячая линия – Телеком, 2009. – 256 с.:ил.
3. Грегер С.Э. Администрирование и интерфейс пользователя CMS Plone (монография). Федер. Агентство по образованию, ГОУ ВПО "УГТУ-УПИ им.первого Президента России Б.Н.Ельцина". Нижнетагил. технол. ин-т (фил.). – Нижний Тагил: НТИ(ф) УГТУ-УПИ, 2009. – 140 с.

4. Грегер С.Э., Сковородин Е.Ю. Построение онтологического портала с использованием объектной базы // Объектные системы – 2010: Материалы I Международной научно-практической конференции. Россия, Ростов-на-Дону, 10-12 мая 2010 г / под общ. ред. П.П. Олейника. – Ростов-на-Дону, 2010. – С. 74-78.
5. Грегер С.Э. Реализация инструментальной среды семантического моделирования учебного процесса. Объектные системы – 2011: материалы III Международной научно-практической конференции, (Ростов-на-Дону, 10-12 мая 2011 г.) / Под общ. ред. П.П. Олейника. – Ростов-на-Дону, 2011. – С.58-61.

УДК 004.4'23 +004.43

РАЗРАБОТКА УЧЕБНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ И ИНТЕРПРЕТАТОРА[1]

Лаптев Валерий Викторович, к.т.н., доцент, Астраханский государственный технический университет, Россия, Астрахань, laptev@ilabsltd.com
Грачев Дмитрий Александрович, магистрант, Астраханский государственный технический университет, Россия, Астрахань, grachev@ilabsltd.com

Введение

В работе [1] одним из авторов сформулированы требования к языку программирования и к интегрированной среде обучения программированию. В частности, обучающая среда должна включать исполняющую подсистему, в которой реализуются программы на учебном языке программирования. Основными компонентами исполняющей подсистемы являются редактор кода и интерпретатор программ на учебном языке.

В настоящей работе описывается разработанный авторами учебный язык, названный Semantic Language (SL), и интерпретатор, позволяющий выполнять программы на этом языке. При разработке учебного языка нужно следовать некоторым основополагающим принципам (концепциям). Концепции разработки языка программирования делятся на три группы: семантические, синтаксические и прагматические (реализационные). Основные семантические концепции, принятые при разработке учебного языка, следующие:

- понятия языка для обучения должны соответствовать понятиям промышленных императивных языков программирования;
- множество понятий языка для обучения должно быть минимально;
- конструкции языка не должны зависеть ни от аппаратной платформы, ни от операционной системы;
- язык должен поддерживать модульность, процедурное программирование и объектно-ориентированное программирование;
- язык должен поддерживать структурное программирование;
- намерения программиста должны указываться явным образом (запрет умолчаний).

В качестве примеров запрета умолчаний можно привести следующие:

- отсутствие неявных преобразований типа (кроме преобразования в арифметических выражениях целый -> вещественный);
- явный параметр this, задаваемый при определении метода класса;
- явное определение записи как базовой для наследования;
- явное указание типа константы при определении.

При разработке учебного языка были приняты следующие основные синтаксические концепции:

- базовая лексика языка должна быть русскоязычной;
- ключевые слова должны иметь английский эквивалент;

[1] Лауреат номинации "**Лучший доклад о методах преподавания объектных технологий в ВУЗе**". Автор доклада награждается правом бесплатной публикации одного доклада по данной тематике на следующей конференции

- каждый оператор языка начинается ключевым словом;
- блочные конструкции языка завершаются ключевым словом «конец».

Основу реализации интерпретатора учебного языка составляют следующие принципы:

- эффективность выполнения является не очень важной;
- загрузка и связывание модулей должны выполняться динамически;
- управление памятью осуществляет система (сборка мусора);

Помимо принципов разработки языка сформулируем не менее важные концепции реализации интегрированной среды, в рамках которой должно осуществляться обучение программированию:

- среда должна обеспечивать работу как с одномодульными, так и с многомодульными программами;
- ввод-вывод данных должен осуществляться в рамках среды без выхода в операционную систему;
- редактор кода, с одной стороны, должен обеспечивать традиционные операции редактирования текста, и, с другой стороны, должен оперировать семантическими конструкциями языка;
- изменение ключевых слов в коде должно быть невозможно;
- ошибки должны определяться в момент набора программы;
- переключение с русской лексики на английскую и обратно не должно приводить к повторной трансляции программы;

И наконец, среда должна быть расширяемой и обеспечивать простой и независимый от платформы реализации механизм накопления программных компонент. Реализация этого принципа позволит преподавателю расширять систему, используя только учебный язык, не привлекая дополнительных средств разработки.

1. Учебный язык программирования

В языке определено всего 4 элементарных типа данных: вещественные числа, целые числа (со знаком), булевские и символьные. Для чисел определены традиционные арифметические операции с традиционными приоритетами. Для целых чисел определено целое деление («\»), отличающееся от вещественного деления, и операция получения остатка от деления («%»).

В целых арифметических выражениях не разрешается задавать вещественные операнды. В вещественных арифметических выражениях разрешается задавать целые и вещественные операнды. Целый операнд, участвующий в операции с вещественным операндом, преобразуется в вещественный – это единственное преобразование типа, выполняемое по умолчанию.

Для чисел определены также 6 традиционных операций сравнения: равно («=»), не равно (решетка – «#»), меньше («<»), больше («>»), больше или равно («>=»), меньше или равно («<=»). Для булевских данных определены булевские константы true (истина, да) и false (ложь, нет) и традиционные логические операции: not («~»), and («&»), or («|»), xor (исключающее или – «^»), – с традиционными приоритетами. Для символьных данных не определено никаких операций.

В учебном языке определено два специальных типа данных: процедурный и указатель. Для этих типов в языке существует единственная операция – операция присваивания. Значениями процедурного типа данных являются имена процедур и функций. Практически объекты процедурного типа необходимы только для изучения темы о передаче процедур и функций как параметров.

Указатель определен с целью изучения динамических структур данных. Значением указателя может быть только адрес объекта определяемого типа и никакого другого.

В языке определен единственный агрегат данных – массив. Массив – это структура данных, содержащая набор элементов одного типа. Тип элементов массива может быть любым допустимым типом языка SL. Количество элементов определяет размер массива,

который вычисляется во время работы программы, поэтому в качестве размера можно задавать произвольное целочисленное выражение, значение которого больше нуля. Размер массива можно получить с помощью метода size() или размер().

Доступ к элементам массива осуществляются по индексу. Индекс – это целочисленное выражение и задается в квадратных скобках после имени массива. Элементы массива нумеруются, начиная с нуля до (размер-1). Индекс проверяется на корректность при обращении к элементу массива.

Массивы в языке SL являются одномерными, поэтому многомерный массив трактуется как «массив массивов массивов …». Операции с объектами-массивами отсутствуют.

Тип данных «строка» в учебном языке не определен, однако строковые литералы присутствуют. Строковый литерал представляет собой символьный массив, размер которого равен количеству символов в строке. Единственная операция, разрешенная с массивами – это присваивание символьному массиву строкового литерала. При этом размер массива должен быть достаточным, чтобы вместить все символы строки (отсечение не допускается).

В языке определен единственный оператор, позволяющий конструировать новые типы данных. Определяемый тип – это тип данных, который изначально отсутствует в языке программирования и определяется программистом. Определяемый тип конструируется с помощью блочного (см. ниже) оператора «тип». Этот оператор является аналогом записи (структуры, класса) в промышленных языках программирования. В нем указывается имя нового типа, которое связывается с определяемой структурой данных. В типе задаются необходимые поля и методы.

Тело определяемого типа – это пространство имен, в котором все имена должны быть различны (тем самым перегрузка методов в одном классе не разрешена). Элементы по умолчанию являются приватными. Любой элемент можно определить как видимый на уровне текущего модуля, или как публичный, видимый во всех модулях, импортирующих данный. Поля определяемого типа можно объявить с модификатором readonly (только для чтения).

Только объекты определяемого типа могут создаваться динамически. Новый тип может быть определен только в модуле, и имя типа видно с момента объявления. В языке SL не существует никаких операций с объектами определяемых типов, кроме присваивания.

Операторы в языке SL делятся на два вида: простые и блочные. Блочные операторы начинаются с заголовка и заканчиваются ключевым словом «конец». Блочные операторы содержат последовательность операторов внутри себя. Эта последовательность операторов называется телом блочного оператора. Блочными операторами являются операторы цикла, условные операторы, операторы определения подпрограмм, оператор определения модуля и оператор определения типа.

Простые операторы тела не имеют. Простые операторы также начинаются ключевым словом, но заканчиваются символом «точка с запятой» («;»). К простым относятся операторы объявления констант и переменных, оператор присваивания, операторы ввода-вывода, оператор вызова процедуры, оператор возврата из подпрограммы, оператор импорта.

Объявление константы присваивает имя (идентификатор) некоторому значению. Оператор начинается ключевым словом «константа», и для каждой константы необходимо указывать тип, имя и выражение, значение которого и связывается с заданным именем. В языке SL разрешено объявлять константы только элементарных типов. Аналогичный синтаксис имеет оператор определения переменных, в котором вместо ключевого слова «константа» задается слово «переменная». В качестве типов могут использоваться все описанные выше типы, синтаксис которых можно представить следующим набором правил.

Только переменные элементарных типов разрешается инициализировать явно. При отсутствии инициализаторов переменные числовых типов обнуляются, а логическим присваивается false. Указатели и переменные процедурных типов при объявлении инициализируются системой специальным значением null. Для элементов массивов действуют те же правила.

Оператор присваивания начинается ключевым словом «присвоить» и имеет традиционный вид: слева переменная (элемент массива, поле записи), справа – совместимое по присваиванию выражение.

Оператор вызова процедуры начинается ключевым словом «вызвать», за которым следует имя процедуры и список параметров.

Оператор ввода начинается ключевым словом «ввести», за которым следует имя переменной. Вводить разрешается данные только элементарных типов. Оператор вывода – это ключевое слово «вывести», за которым следует выражение. Выводить разрешается только значения элементарных типов.

Условный оператор является блочным и имеет традиционный вид: начинается ключевым словом «если» и заканчивается ключевым словом «конец». В тело оператора может быть вставлена одна ветвь «иначе» и произвольное число ветвей «а_если» (в английской нотации «else_if»).

В экспериментальных целях (для изучения удобства использования) в SL определен обобщенный условный оператор, который имеет следующий вид в английской нотации:

```
if
| охрана: операторы;
…
| охрана: операторы;
else
          операторы;
end;
```

Формат этого оператора совпадает с форматом условного оператора Дейкстры [2], однако семантика существенно отличается. Оператор Дейкстры по определению является недетерминированным (одна из охран выбирается случайным образом), что неприемлемо при обучении практическому программированию. В нашем варианте охраны (булевские выражения) не являются взаимоисключающими, и выполняются те группы операторов, значения охран которых истинны. Если все охраны ложны, то выполняется ветка else (которая может отсутствовать).

Этот оператор является более общей формой условного оператора и практически выполняет функции оператора-переключателя, который в языке не определен.

В учебный язык включен только один традиционный оператор цикла – цикл while. С помощью этого цикла моделируются другие формы операторов цикла. В экспериментальных целях (для изучения удобства использования) в SL определен цикл Дейкстры [2], который имеет следующий вид в английской нотации:

```
do
| охрана: операторы;
| охрана: операторы;
…
| охрана: операторы;
end;
```

Как описано в [2], охраны (булевские выражения) не являются взаимоисключающими, и выполняются те группы операторов, значения охран которых истинны. Цикл выполняется до тех пор, пока хотя бы одна охрана истинна.

Для поддержки процедурной парадигмы в языке определены процедуры и функции, которые традиционно содержат заголовок и тело:

```
процедура (параметры) имя
  операторы;
конец имя;
```

Функция отличается заголовком, в котором указывается возвращаемый тип:

```
функция (параметры):тип имя
```

Возврат из процедур осуществляется оператором «вернуть» без аргумента. Этот же оператор с аргументом-выражением используется для возврата результата из функций.

В теле функции разрешается объявлять переменные, которые являются локальными для функции и видны с момента объявления. Тело функции представляет собой пространство имен, поэтому все объявляемые имена и имена параметров должны быть различны.

Типы параметров и возвращаемого результата могут быть любыми допустимыми типами. Параметры могут быть входными, выходными или переменными.

Методы могут быть определены только в теле определяемого типа. В заголовке метода явно указывается дополнительный параметр – аналог невидимого параметра this в С-подобных объектно-ориентированных языках.

Для поддержки модульности в учебном языке определена явная конструкция модуля, аналогичная конструкции модуля в языке Компонентный Паскаль [3]. Модуль должен иметь уникальное имя и включает две секции: секцию определений и секцию инициализации. В секции определений задаются список импортируемых модулей (если он есть), определение типов, констант, переменных, и независимых процедур и функций.

Тело модуля является пространством имен, поэтому все имена, определенные в секции инициализации, должны быть различны. Имя становится видимым с момента определения. По-умолчанию все имена локальны в модуле. Но любое имя можно явным образом сделать публичным, доступным в других модулях при импорте данного. Переменные, объявленные в секции инициализации, являются локальными в этой секции.

Выполняемой программой является модуль. В секции инициализации модуля задается последовательность операторов, которые выполняются при загрузке модуля в память. Секция инициализации – это неименованная процедура, выполняемая первой. Объявленные в ней переменные и константы являются локальными. В этой секции может быть задан вызов любой процедуры, функции или метода, определённых в секции объявлений. Выполнение операторов секции начинается после загрузки импортированных модулей.

В общем случае программа представляет собой набор модулей, первый из которых явным образом запускается программистом, а остальные загружаются в память для выполнения по мере необходимости.

2. Примеры использования учебного языка программирования

Рассмотрим несколько простых примеров программ на учебном языке. Как уже принято в компьютерном мире, первая программа – HelloWorld.

```
# Первая программа на учебном языке – это комментарий #
модуль HelloWorld
начало
  вывести “Привет, Мир!”;
конец HelloWorld.
```

Второй пример – вычисление факториала (в английской нотации).

```
module Factorial
begin
 variable integer i := 1;
 variable integer current := 1;
 constant integer N = 20;

 while i < N do
   let current := current * i;
   let i := i + 1;
   output current;
   output ‘\n’;
 end while;
end Factorial.
```

Третий пример – модуль, в котором реализована простая функция для вычисления случайных чисел.

```
модуль Рандом
 переменная целое x := 5;
 # открытая функция – доступна в других модулях #
 открыт функция (входной целое мин, входной целое макс):целое рандом
   константа целое a := 2147483647;
   константа целое b := 48271;
   константа целое c := 44488;
   константа целое d := 3399;

   присвоить x := b * (x % c) – d * (x \ c);
   если x < 0 тогда
       присвоить x := x + a;
   конец ветвления;
   вернуть x % (макс – мин) + мин;
 конец рандом;
начало
конец Рандом.
```

Использование модуля Рандом показывает следующий пример, в котором массив заполняется случайными числами и выводится.

```
модуль СлучайныйМассив
 подключить Рандом;

 константа целое размер := 10;
 переменная массив [размер] целое a;

 процедура () ИнициализацияМассива
   переменная целое и := 0;
   пока и < размер повторять
       присвоить a[и] := Рандом.рандом(0, 50);
       присвоить и := и + 1;
   конец цикла;
 конец ИнициализацияМассива;

 процедура () ВывестиМассив
   переменная целое и := 0;
   пока и < размер повторять
       вывести a[и];
       вывести '\t';
       присвоить и := и + 1;
   конец цикла;
 конец ВывестиМассив;
начало
 вызвать ИнициализацияМассива();
 вызвать ВывестиМассив();
конец СлучайныйМассив.
```

3. Семантический редактор

Для набора и выполнения программ на учебном языке была разработана интегрированная среда Semantic IDE, состоящая из семантического редактора и интерпретатора. Внешний вид среды показан на рис. 1.

Центральное окно – окно редактора кода. Слева – вкладка оглавления справки, справа – окно проекта (тоже сворачивающееся во вкладку), внизу – окно сообщений об ошибках. При выполнении программы среда переключается в окно консоли, в котором программист задает входные данные и в которое выводятся результаты работы программы. При работе в среде не открывается никаких дополнительных окон со стороны операционной системы.

Редактор кода – это не традиционный текстовый редактор. В Semantic IDE реализован семантический редактор. Как было описано в работе [1], семантический редактор оперирует не символами, а операторами учебного языка. Поэтому, во-первых, большинство элементов оператора сразу вставляются в код в правильном виде, и, во-вторых, полностью исчезают

ошибки набора ключевых слов. Редактор позволяет символьный ввод только в строго определенных позициях оператора. Например, в операторе «переменная» разрешено вводить посимвольно только имя переменной.

Аналогично выполняется удаление – удаляется вся конструкция целиком. Если же оператор не удаляется, то разрешается замена элементов оператора. Например, в операторе «переменная» можно заменить тип переменной, выбрав его из списка предложенных типов. Точно так же предлагается список видимых в данной точке имен переменных, если программисту потребовалось заменить имя переменной.

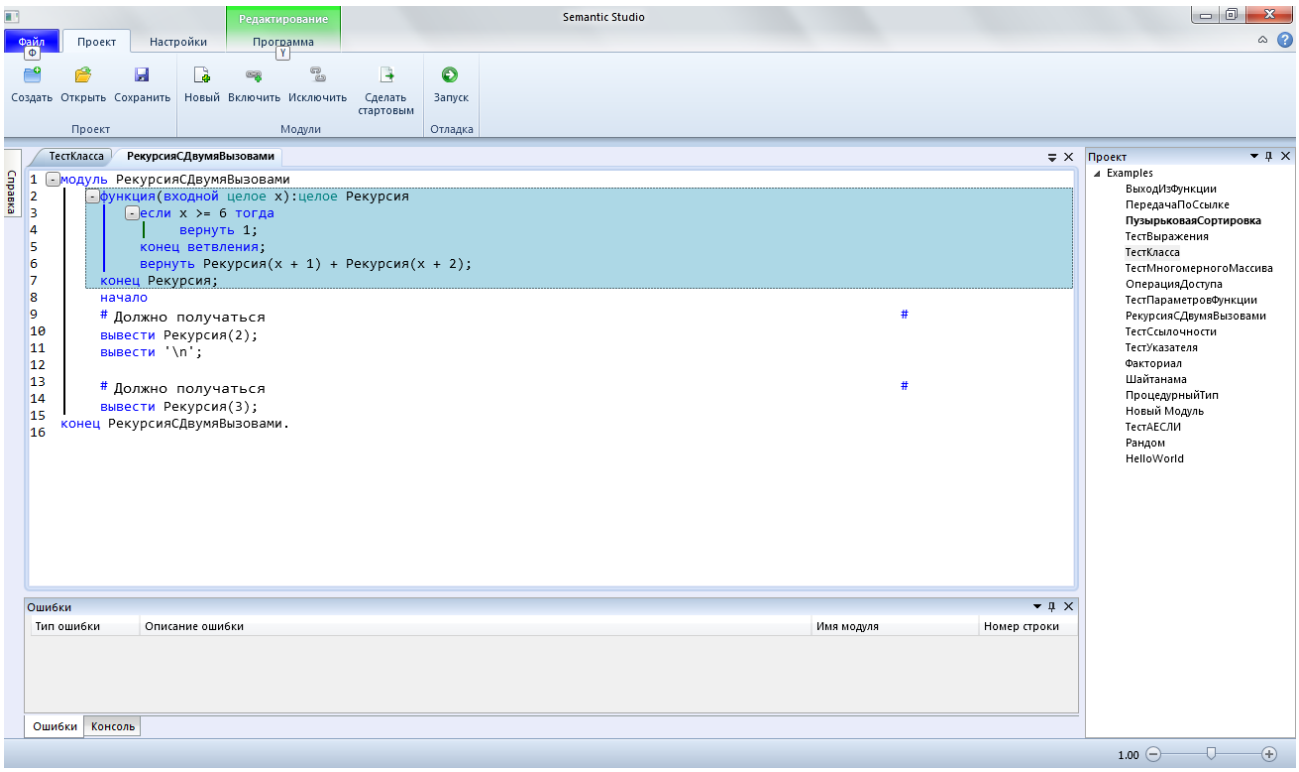


Рис. 1 – Внешний вид Semantic IDE

Редактор следит за действиями программиста и сообщает об ошибках в момент набора программы. Например, при ошибке в написании имени переменной в выражении в окне ошибок мгновенно появляется сообщение о том, что данное имя переменной не описано.

Таким образом, набор кода программы требует минимального количества действий от программиста, и при этом существенно сокращается количество ошибок.

Помимо набора текста программ, редактор позволяет проводить рефакторинг кода. Определены и реализованы операции рефакторинга для разнообразных возможных ситуаций при создании кода, и редактор выдает подсказки программисту в виде списка доступных операций рефакторинга в конкретной ситуации.

Помимо кода, редактор позволяет набирать любой текст в узлах-комментариях. Комментарий может быть написан как в коде программы, так и вне его. Кроме того, в комментарий можно вставлять рисунки и любой текст из буфера обмена. Шрифт комментария можно настраивать обычным для системы Windows образом. Все это позволяет непосредственно в редакторе среды готовить обучающие материалы, причем в документе может содержаться код примера, который можно выполнить.

Отметим, что справочная система реализуется уже в самой интегрированной среде без использования средств операционной системы.

4. Интерпретатор

Результатом работы редактора кода является семантическое дерево программы (см. рис. 2), которое является входным для интерпретатора. Каждый узел дерева представляет отдельный оператор программы (или комментарий). В каждом узле две ссылки: на

следующий узел и на вложенный. Следующий узел – это следующий оператор в коде программы. Таким образом, последовательность операторов представляет собой список узлов семантического дерева. Дочерний узел представляет собой тело блочного оператора. В каждом узле хранятся все данные оператора. Например, узел, представляющий оператор присваивания, хранит ссылку на имя переменной и ссылку на вычисляемое выражение.

Единицей интерпретации языка является узел семантического дерева, то есть оператор. При этом каждый узел после непосредственно своей интерпретации вызывает базовый метод интерпретации. На языке Semantic Language с использованием обобщенного условного оператора он выглядит следующим образом:

```
метод (Узел этот) Интерпретировать()
  если
    |этот.Дочерний # пусто():
       вызвать этот.Дочерний.Интерпретировать();
    |этот.Следующий # пусто():
       вызвать этот. Следующий.Интерпретировать();
  конец ветвления;
  вызвать память.СобратьМусор();
конец Интерпретировать;
```

Такой подход интерпретирует программу простым обходом семантического дерева в глубину с запуском процедуры интерпретации у текущего оператора. Каждый оператор реализует собственную процедуру интерпретации.

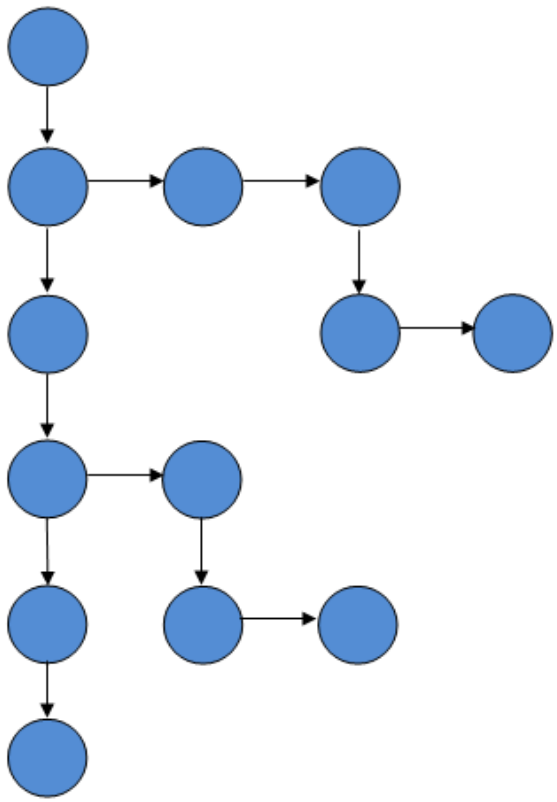


Рис. 2 – Вид семантического дерева программы

Таким образом, общая схема интерпретации может быть представлена в виде рекурсии следующим образом:

1. Выполнение действий, характерных для данного оператора.
2. Проверка на наличие дочернего элемента. В случае его существования – запуск процедуры интерпретации у дочернего элемента.
3. Проверка на наличие следующего элемента. В случае его существования – запуск процедуры интерпретации у следующего элемента. Иначе – очистка памяти от объектов, вышедших из области видимости.

4. Процесс интерпретации начинается с вызова любой процедуры или функции, заданого в секции инициализации модуля.
5. В случае возникновения ошибки процесс интерпретации прерывается, и управление передается среде. Процесс интерпретации также прерывается при интерпретации оператора, помеченного как «остановочный».

Данная схема характерна для большинства операторов (табл. 1), за исключением оператора условного перехода и оператора цикла.

Таблица 1. Операторы языка Semantic Language

Название	Механизм интерпретации
переменная, инициализированная переменная, константа, массив, поле	Создают соответствующий объект в памяти
тип	Не интерпретируется. Представляется как класс в памяти программы на этапе семантического анализа.
вызвать	Производит интерпретацию процедуры, указанную пользователем.
вернуть	Производит вычисление выражение и возвращает полученное значение оператору, вызвавшему данную функцию. Процесс интерпретации функции при этом прерывается.
пустая строка	Использует базовый метод интерпретации.
модуль	Не интерпретируется.
начало	Интерпретируется как вызов начальной (главной) процедуры
комментарий	Использует базовый метод интерпретации.
иначе	Использует базовый метод интерпретации.
функция, процедура, метод	Создают локальные переменные или псевдонимы для параметров. Контролируют уровень стека памяти.
ввести	Прерывает выполнение программы и запрашивает ввод пользователя. После ввода продолжает выполнение программы.
вывести	Выводит строку на консоль.

Оператор условного перехода «если» вычисляет значение логического выражения, объявленного в условии, и в случае ложного выражения передает управление дочернему оператору оператора «иначе». Если оператор «иначе» отсутствует, то управление передается согласно стандартной схеме интерпретации.

Оператор цикла запускает процесс интерпретации у своего дочернего элемента, пока выражение истинно, иначе передает управление следующему оператору.

5. Организация памяти

Виртуальная машина для языка SL имеет стековую архитектуру. Стек памяти разделен на кадры, каждый из которых имеет порядковый номер. Кадр стека с номером 0 является глобальной областью памяти, которая создается при запуске программы и живет до конца ее выполнения. В ней хранятся все глобальные объекты, объявленные в программе. При вызове подпрограммы в стеке создается новый кадр со значением на единицу больше предыдущего, при выходе из подпрограммы этот кадр удаляется. Однако ссылочные переменные (например, объекты типов) остаются в памяти. Это позволяет применить ссылочную парадигму без указателей (как, например, в языках C# и Java), реализовав сборку мусора.

Каждый объект в стеке содержит ссылку, по которой к нему можно обратиться. Виртуальная машина хранит словарь имен времени выполнения, связывающий имена

объектов в программе с объектами в памяти. При обращении к объекту по его имени сначала вычисляется ссылка, а потом, если это необходимо, возвращается объект. Вопрос о необходимости возвращения непосредственно объекта или ссылки на него в операции присваивания открывает интересный аспект архитектуры чисто интерпретируемого языка.

Разные промышленные языки решают эту проблему только одним из двух способов. Например, C++ копирует объекты, а C# и Java – ссылки. Это решение зафиксировано в реализации трансляторов и в реализации виртуальной машины языка. Заметим, что в императивных языках обычно можно реализовать альтернативный режим копирования вручную, написав соответствующий код.

При использовании чистого интерпретатора возникает возможность в рамках одной реализации использовать оба подхода. Пользователь устанавливает (средствами среды) некоторый флаг. Если пользователь решил использовать копируемый подход, то интерпретатор работает в режиме копирования объектов. Отметим, что при этом можно реализовать разные режимы копирования: непосредственное копирование или «ленивое» копирование, при котором реальное копирование откладывается до того момента, когда реально потребуется копия объекта.

Если же пользователь решил использовать ссылочный подход, то интерпретатор работает в режиме копирования ссылок.

Таким образом, в рамках разрабатываемой среды Semantic IDE имеется возможность обучать программированию в двух противоположных парадигмах без написания кода просто переключением одного флажка!

Выводы

Описанная в данной статье интегрированная среда Semantic IDE является основой обучающей системы, требования к которой изложены в [1]. В настоящее время реализация Semantic IDE выполняется на языке C# в среде Visual Studio 2010. Уже полностью реализована вся процедурная часть учебного языка. В 2011 году Дмитрий Грачев с описанной системой выиграл грант по программе «У.М.Н.И.К.».

Небольшой пока опыт использования показывает существенное сокращение времени по созданию кода программы и практически полное исчезновение ошибок набора. В настоящий момент развитие среды продолжается: реализуется объектно-ориентированная часть Semantic Language и модули системной библиотеки. Предполагается внедрение системы в Астраханском государственном техническом университете уже в новом 2012 учебном году.

Литература

1. Лаптев В.В. Требования к современной обучающей среде по программированию // Объектные системы-2010 (Зимняя сессия): материалы II Международной научно-практической конференции. Россия, Ростов-на-Дону, 10-12 ноября 2010 г. / Под общ. Ред. П.П. Олейника. – Ростов-на-Дону, 2010. – с. 104-110.
2. Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978. – 275 с.
3. Потопахин В. Современное программирование с нуля! – М.: ДМК Пресс, 2010. – 240 с.

Шахтинский институт (филиал)
государственного образовательного учреждения
высшего профессионального образования
«Южно-Российский государственный технический университет
(Новочеркасский политехнический институт)»

Южно-Российский государственный университет
экономики и сервиса

VII Международная научно-практическая конференция

Подробную информацию о конференции можно найти на официальном сайте www.objectsystems.ru

ОБЪЕКТНЫЕ СИСТЕМЫ – 2013

(с изданием сборника материалов конференции)

Информационные партнёры конференции:

Сообщество системных аналитиков

Теоретический и прикладной научно-технический журнал
"**Информационные технологии**"
с ежемесячным приложением

Ежемесячный научно-технический и производственный журнал
"**Автоматизация в промышленности**"

Журнал "**Информационные технологии и вычислительные системы**"

10–12 мая 2013 г , Россия, Ростов-на-Дону

Конференция посвящена принципам проектирования, реализации и сопровождения объектных систем и включает в себя обсуждение широкого круга проблем. В организации и работе конференции принимают участие как известные учёные, так и крупные специалисты в области разработки корпоративных информационных систем, представители ВУЗов и коммерческих организаций России, ближнего и дальнего зарубежья (Греции, Польши, Испании). Конференция является заочной. По окончании работы конференции публикуется сборник научных трудов авторов с присвоением ему ISBN-кода, который будет разослан в крупнейшие научные библиотеки России. Электронная версия сборника размещается в ведущих информационных каталогах и доступна на сайте конференции. Авторам каждого

доклада высылается печатная версия сборника материалов конференции и именные сертификаты. Лучшие, по мнению рецензентов, доклады будут рекомендованы (после соответствующей доработки) к бесплатной публикации в журналах "Автоматизация в промышленности", "Информационные технологии" и "Информационные технологии и вычислительные системы", входящих в перечень ведущих рецензируемых научных журналов и изданий, рекомендованных ВАК. Лучший доклад по UML-моделированию награждается книгой Иванова Д. Ю. и Новикова Ф.А. "Моделирование на UML. Теория, практика, видеокурс" (www.umlmanual.ru) с автографами авторов. Автор лучшего доклада, посвящённого методам преподавания объектных технологии в ВУЗе получает право бесплатной публикации одного доклада сходной тематики на следующей конференции.

Оргкомитет конференции

1. Прокопенко Николай Николаевич, д.т.н., проф., Ректор Южно-Российского государственного университета экономики и сервиса, Россия, Шахты ***(председатель конференции)***
2. Олейник Павел Петрович, к.т.н., Системный архитектор программного обеспечения, Астон, Россия, Ростов-на-Дону ***(сопредседатель конференции)***
3. Божич Владимир Иванович, д.т.н., проф., Кафедра "Информационные системы и радиотехника", Южно-Российский государственный университет экономики и сервиса, Россия, Шахты
4. Сидельников Владимир Иванович, д.т.н., проф., Заведующий кафедрой "Экономика и прикладная математика", Педагогический Институт Южного Федерального университета, Россия, Ростов-на-Дону
5. Черкесова Эльвира Юрьевна, д.э.н., проф., Заведующая кафедрой "Информационные технологии и управление", Шахтинский институт (филиал), Южно-Российский государственный технический университет (Новочеркасский политехнический институт), Россия, Шахты
6. Михайлов Анатолий Александрович, д.т.н., проф., Кафедра "Автоматизированные системы управления", Южно-Российский государственный технический университет (Новочеркасский политехнический институт), Россия, Новочеркасск
7. Кравчик Вячеслав Георгиевич, к.т.н., доц., Кафедра "Энергетика и БЖД", Южно-Российский государственный университет экономики и сервиса, Россия, Шахты

Международный программный комитет конференции

1. Euclid Keramopoulos, Ph.D., Lecturer, Alexander Technological Educational Institute of Thessaloniki, Греция, Салоники
2. Piotr Habela, Ph.D., Assistant Professor, Polish-Japanese Institute of Information Technology, Польша, Варшава
3. Erki Eessaar, Ph.D., Associate Professor, Acting Head of Chair, Faculty of Information Technology: Department of Informatic, Tallinn University of Technology, Эстония, Таллин
4. German Viscuso, MSc in Computer Science, Marketing, Versant Corp., Испания, Мадрид
5. Кузнецов Сергей Дмитриевич, д.т.н., проф., Факультет вычислительной математики и кибернетики, МГУ имени М. В. Ломоносова, Главный научный сотрудник Института системного программирования РАН, член ACM, ACM SIGMOD и IEEE Computer Society, Россия, Москва
6. Шалыто Анатолий Абрамович, д.т.н., проф., лауреат премии Правительства РФ в области образования, Заведующий кафедрой "Технологии программирования", Санкт-Петербургский государственный университет информационных технологий механики и оптики, Россия, Санкт-Петербург
7. Кирютенко Александр Юрьевич, к.ф.-м.н., Директор по ИТ, Астон, Россия, Ростов-на-Дону
8. Галиаскаров Эдуард Геннадьевич, к.х.н, доц., Ивановский государственный химико-технологический университет, Россия, Иваново
9. Чекирис Александр Владимирович, Начальник отдела технического проектирования и НСИ, НИИЭВМсервис, Беларусь, Минск
10. Векленко Ирина Юрьевна, к.э.н., Ведущий системный аналитик, РИТКОН, Россия, Черноголовка
11. Малышко Виктор Васильевич, к.ф.-м.н., доц., Факультет вычислительной математики и кибернетики, МГУ имени М. В. Ломоносова, Россия, Москва
12. Жилякова Людмила Юрьевна, к.ф.-м.н., с.н.с., Институт проблем управления им. В.А. Трапезникова РАН, Россия, Москва

13. Шахгельдян Карина Иосифовна, д.т.н., Начальник информационно-технического обеспечения, Владивостокский государственный университет экономики и сервиса, Россия, Владивосток
14. Добряк Павел Вадимович, к.т.н., доц., Уральский государственный технический университет, Россия, Екатеринбург
15. Байкин Александр Сергеевич, Ведущий системный аналитик, Автомир, Россия, Москва
16. Аверин Алексей Иванович, Системный аналитик, Астон, Россия, Ростов-на-Дону
17. Лаптев Валерий Викторович, к.т.н., доц., Кафедра "Автоматизированные системы обработки информации и управления", Астраханский государственный технический университет, Россия, Астрахань
18. Ермаков Илья Евгеньевич, Заместитель директора, казенное учреждение Орловской области "Региональный центр оценки качества образования", Технический директор, НПО "Тесла", Россия, Орёл
19. Иванов Денис Юрьевич, Консультант, Ай Ти Консалтинг, Россия, Санкт-Петербург

Основные секции конференции

1. Графические нотации, используемые при объектном проектировании ИС
2. Принципы объектного проектирования информационных систем
3. Инструменты объектного моделирования
4. Теория объектно-ориентированного программирования
5. Методы (шаблоны) объектно-реляционного отображения
6. Реализация и использование объектных расширений в реляционных СУБД
7. Проектирование, разработка и реализация распределённых систем
8. Типовые реализации КИС с применением объектных технологий
9. Принципы организации и реализации объектных баз данных
10. Проблемы реализации объектных СУБД
11. Имитационное моделирование объектных систем
12. Темпоральные объектные системы
13. Проблемы изучения (преподавания) объектных технологий в ВУЗе

Ключевые даты конференции

01.01.2013 – 15.04.2013 – приём заявок на участие в конференции и докладов к публикации
16.04.2013 – 30.04.2013 – рецензирование и корректировка присланных докладов
01.05.2013 – 09.05.2013 – рассылка уведомлений о принятии докладов на конференцию и приём квитанций об оплате оргвзноса
10–12 мая 2013 – формирование сборника трудов и сертификатов
01.07.2013 – 15.07.2013 – размещение на сайте конференции электронного макета сборника материалов с присвоенными ISBN-кодами и именных сертификатов, подтверждающих участие в конференции
II – III квартал 2013 г. – рассылка печатной версии сборника материалов авторам, и в научные библиотеки РФ. Рассылка бумажных именных сертификатов. Регистрация электронной версии в специализированных каталогах

Подробную информацию о конференции можно найти на официальном сайте www.objectsystems.ru

Для заметок / Notes

Для заметок / Notes

Для заметок / Notes

Отпечатано 29.06.2012

Верстка – **Галиаскаров Э.Г.**
Дизайн обложки – **Олейник П.П.**
Корректоры – **Лаптев В.В., Ермаков И.Е.**
Ответственный за выпуск – **Галиаскаров Э.Г.**
Подписано в печать **28.06.2012**. Формат А4
Бумага офсетная. Печать цифровая
Усл.печ.л. **11,7**
Тираж 300 экз
Заказ № 712

ISSN 2309-8856

01

9 772309 885121

ISBN 978-5-9903782-1-6

9 785990 378216

ISBN 978-5-9903782-2-3

9 785990 378223