очевидно, что оценки эффективности работы любого способа индексирования зависит от конкретного набора данных. Тестирование методов доступа к многомерным данным на случайных наборах данных приведенных, например, в [8] представляется нам непоказательным. Поскольку реальные наборы данных, особенно в случаях большого количества измерений, обнаруживают ярко выраженную кластеризацию. Как следствие оценки эффективности индексирования на случайных наборах, по нашему мнению, не имеет прямой связи с эффективностью индексирования реальных данных.

### СПИСОК ЛИТЕРАТУРЫ


1. Gaede V., Gunter O. Multidimensional Access Methods // ACM Comput. Surv. – 1998. – V. 30. – № 2. – p. 170–231.
2. Manolopoulos Y., Nanopoulos A., Papadopoulos A., Theodoridis Y. R-trees: Theory and Applications. – Springer, 2006. – 194 p.
3. Hellerstein J., Kornacker M., Mohan C. Concurrency and Recovery in Generalized Search Trees // SIGMOD Record. – 1997. – V. 26. – № 2. – P. 62–72.
4. Guttman A. R-trees: A dynamic index structure for spatial searching // Proc. of the ACM SIGMOD Intern. Conf. on Management of Data, 1984. – P. 47–54.
5. Beckmann N., Kriegel H.-P., Schneider R., Seeger B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles // Proc. of the ACM SIGMOD Intern. Conf. on Management of Data, 1990. – P. 47–54.
6. Graefe G., Implementing Sorting in Database Systems // ACM Comput. Surv. – 2006. – V. 38. – № 2. – P. 87–94.
7. Гарсиа-Молина Г., Ульман Дж., Уидом Дж. Системы баз данных. Полный курс – М.-СПб.: Вильямс, 2004. – 1088 с.
8. Шестаков Н.А. Индексирование пространственных данных в MS SQL Server 2000 // Известия Томского политехнического университета. – 2006. – Т. 309. – № 4. – С. 157–162.


*Поступила 14.07.2008 г.*

УДК 004.89

# ОБ ИСПОЛЬЗОВАНИИ В ПРОГРАММИРОВАНИИ ПРОБЛЕМНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ

С.С. Васильев, В.Б.Новосельцев

Томский политехнический университет
E-mail: vitaliin@gmail.com

*Рассматривается популярный подход к повышению эффективности процесса разработки надежного программного обеспечения, который предполагает систематизированное применение механизма абстракции за счет использования так называемых проблемно-ориентированных языков. С учетом опыта работы в рамках представляемой парадигмы анализируются и обсуждаются достоинства и недостатки подхода, а также перспективы его развития.*

***Ключевые слова:***
*Проблемно-ориентированные языки, механизм абстракции, ИТ-проект, программный инструментарий.*

### Введение

В настоящее время вслед за теоретиками в области информационных технологий (ИТ) многие разработчики программного обеспечения сходятся во мнении, что механизм абстракции является одним из ключевых факторов, определяющих качество и эффективность реализации сложного программного продукта. Своего рода «венцом» целого ряда исследований по тематике *абстрактных типов данных* в 70-х годах прошлого века явился язык *CLU*, предложенный Барбарой Лисков [1]. Несомненна концептуальная значимость языка *CLU* – он определил одну из основ *объектно-ориентированного подхода*. В то же время продукт Б. Лисков не получил сколько-нибудь серьезного распространения в среде профессиональных разработчиков, прежде всего, вследствие излишней академичности и перегруженности специфическими конструкциями.

Достаточно очевидным является то, что разумно примененный механизм абстракции позволяет весьма эффективно осуществлять командную разработку, поддерживая при этом высокое качество программного продукта и эффективное управление процессом. Отложенные вычисления [2–6], модули [2–6], объекты [6, 7], манданты [5, 8] – все это лишь некоторые из современных инструментов введения и поддержки абстракции. Разные языковые среды поддерживают эти механизмы в той или иной степени, одни хорошо, другие не очень. Общеизвестно, что любые алгоритмические решения могут быть реализованы в рамках различных существующих парадигм программирования (выбор, в значительной степени, определяется личными предпочтениями). Закономерно возникает вопрос, что же может претендовать на роль «идеального» механизма абстракции при написании приложений, лежит ли он в плоскости технических средств

(подобно упомянутому выше *CLU*) или в большей степени определяется соответствующей методологией, используемой в процессе работы с традиционным инструментарием.

В определенном смысле ответом может служить практика реализации ИТ-проектов промышленного уровня. Исходя из собственного опыта и анализируя доступные источники, в данной работе авторы отстаивают тезис о том, в настоящее время наиболее распространенным подходом к использованию абстракции является применение проблемно-ориентированных языковых средств специализированного характера. На границах спектра подобного инструментария, с одной стороны, можно расположить просто наборы сложных структур данных (типов) и соответствующих процедур обработки, построенных средствами одного из распространенных языков программирования. Другим предельным случаем может служить специально созданный под конкретный узко-очерченный класс задач новый язык. Такой язык не обладает никаким функционалом, выходящим за рамки решаемой проблемы, и покрывает только нужды данной предметной области – ни больше, ни меньше. Как правило, в реальном проекте используется подходящая комбинация, либо промежуточный вариант средств очерченного спектра, здесь мы основное внимание уделяем проблемно-ориентированным языкам (ПОЯ) как таковым.

### Проблемно-ориентированные языки

Следуя сложившейся традиции [9–11], определим ПОЯ (*DSL – Domain Specific Language*) следующим образом: язык программирования, который обладает ограниченной степенью выразительности/силы, сфокусированный на определенной предметной области и содержащий конструкты, отражающие абстракции именно этой предметной области.

Проблемно-ориентированные языки принято разделять на три группы [9]:

1. Внешние (*external DSL*).
2. Внутренние (*internal DSL*).
3. Инструментальные средства языка (*language workbench*).

Рассмотрим эти группы подробнее.

**1. *Внешние ПОЯ*** используют синтаксис, полностью отличный от синтаксиса языка основного приложения. Единственным исключением, пожалуй, является ситуация, когда в качестве основного языка берется *XML*. Внешние ПОЯ – это наиболее яркие представители современных инструментов введения абстракции: *little languages* в *Unix*, регулярные выражения, *SQL*, *PostScript*, *awk*, *HTML* и т. д.

Самый большой плюс внешних ПОЯ состоит в том, что их можно писать, не оглядываясь на стандарты и спецификации. Иначе говоря, разработчик может выразить предметную область в самой лаконичной и пригодной для чтения и редактирования форме. Качество такого ПОЯ определяется лишь умением разработчика создать транслятор, который сможет откомпилировать входной файл и выдать исполняемый код – как правило, уже на основном языке приложения. Отсюда же следует и очевидный недостаток внешних ПОЯ – необходимость создания этого самого транслятора.

Еще один существенный недостаток внешних ПОЯ определяется отсутствием у них того, что принято называть «символической интеграцией» (*symbolic integration*) [12]. Внешний ПОЯ, на самом деле, никак не связан с основным языком приложения. Программная среда разработки для базового языка, на котором пишется это приложение, не содержит информации о новом ПОЯ, и следовательно, проблема редактирования связей исполняемого кода стоит довольно остро.

Обратимся к примеру, приведенному на рис. 1. Предположим, разработчик решил изменить свойства целевого класса *User*. Во всех современных средах разработки автоматическим переименованием/рефакторингом уже никого не удивишь. Однако это переименование не будет работать в *SQL* коде. Это и есть тот самый «символический барьер» между *Java* и *SQL* – он не позволяет манипулировать собранной программой как единым целым.

**2. *Внутренние ПОЯ*** используют существующий (обычно – универсальный) язык как свою основу (последний часто называют языком-носителем – *host language*). При этом создаваемый ПОЯ, как правило, получается синтаксически совместимым с языком-носителем и может быть обработан, ис-

```
public void insert(User aUser) throws Exception {
  aUser.setGeneration(0);
  getTemplate.update("insert into user_table (id, created, properties, generation,
status, backend, email, pswd) values (?,?,?,?,?,?,?,?)",
    aUser.getID(),
    aUser.getCreated(),
    Bytes.toByteArray(aUser.getProperties()),
    aUser.getGeneration(),
    aUser.getStatus().toString(),
    aUser.getBackendID(),
    aUser.getEmail(),
    aUser.getPassword());
}
```

***Рис. 1.*** *Использование внешнего ПОЯ (SQL) из языка Java*

пользуя его инфраструктуру (скомпилирован штатным компилятором, интерпретирован, отлажен и т. д.). Полученный таким образом ПОЯ является одновременно как *расширением* языка-носителя, так и его *ограничителем*.

При построении внутреннего ПОЯ как *расширения* (рис. 2, *а*), нововведенные концепции становятся доступными для языка-носителя, и окончательным результатом является новый язык, который обладает полным функционалом исходного языка и добавляет специфичные расширения.*ие внутреннего ПОЯ: а) расширение, б) сужение*

В случае реализации внутреннего ПОЯ как *ограничителя* языка-носителя (рис. 2, *б*), новый язык отражает специфику предметной области, при этом скрывая (а зачастую, запрещая) большинство конструкций языка-носителя, которые не имеют отношения к задачам данной предметной области. Окончательный результат в таком случае – это также новый язык, только, в некотором смысле, суженный.

К классическим примерам внутренних ПОЯ можно отнести: *LISP* (опытные *LISP*-программисты говорят о процессе программирования на *LISP*, как о постоянном создании и использовании новых ПОЯ), *Ruby* и *JRuby* (большинство библиотек *Ruby* выполнены в стиле ПОЯ, как например, *Rails*) и ряд других языков. Пример использования внутреннего ПОЯ при работе с *Ruby* приведен на рис. 3.

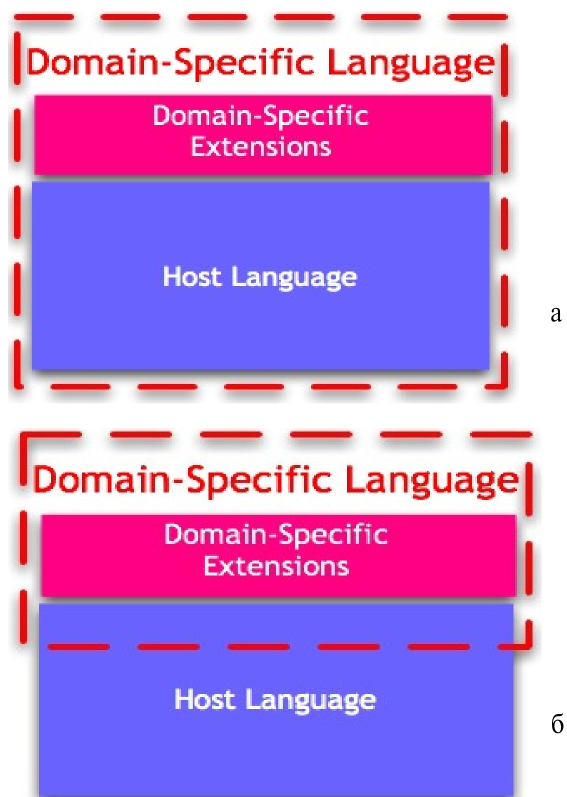


***Рис. 2.*** *Построен*

```
class XmlBookDetailModel

  #implementing IBookDetailModel.java interface
  include IBookDetailModel

  @doc
  def initialize {...}

  ### Methods implementing IBookDetailModel interface ###
  def getAllBooks {...}

  def loadBookDetail(book)
    isbn = book.getIsbn

    #find matching book by isbn
    @doc.elements.each("books/book") { |elem|
      if (elem.attributes['isbn'] == isbn)
        book.setTitle(elem.attributes['title'])
        book.setAuthor(elem.elements['author'].text)
        book.setPublisher(elem.elements['publisher'].text)
        book.setDatePublished(elem.elements['datePublished'].text)
        book.setDescription(elem.elements['description'] .text)
        return book
      end
    }
    return book
  end
end
```

***Рис. 3.*** *Пример использования внутреннего ПОЯ – Ruby*

Плюсы и минусы внутренних ПОЯ являются, практически, зеркальным отражением внешних. Здесь уже нет символьного барьера. Разработчик может в полную силу пользоваться возможностями языка-носителя и задействовать все инструменты, которые есть для этого языка.

Классический пример иллюстрируется рис. 4. Использование конфигурационного файла для определения параметров работы системы может быть упрощено, если в качестве формата конфигурационного файла выступает программа, выполняемая системой, например, при загрузке. Пример демонстрирует использование языка *Ruby*, как основы ПОЯ, при этом собственно *Ruby*-код в примере не содержится. Язык-носитель ограничен до минимальных требований задачи, включая изменение парадигмы императивного программирования на декларативную.

```
describe_node{
        ip [200,122,122,122]
        hostname ['admin-node']
        system_port 3000
        alternative_port 3001
        security_method :SSL
}

describe_node{
        ip [200,122,122,102]
        hostname ['indexer-node']
        system_port 3000
        alternative_port 3001
        security_method :none
        java_version JDK_1_4
}
```

***Рис. 4.*** *Пример использования ПОЯ для конфигурирования системы*

Отметим, что возможность использования для ПОЯ всей мощи языка-носителя обладает как достоинствами, так и недостатками. Характерной особенностью ПОЯ является возможность работы без изучения всех нюансов носителя. Это призвано обеспечить возможность экспертам предметной области (а не профессиональным программистам) вносить алгоритмику предметной области непосредственно в программную систему. Внутренний ПОЯ может усложнить этот процесс, поскольку существует множество мест, где пользователь может оказаться в тупике из-за недостаточной осведомленности о возможностях базового языка.

**3. *Инструментальные средства языка* (*language workbenches*) –** это среды разработки (*IDE – Integrated Development Environment*), предназначенные для создания новых ПОЯ. Среда позволяет определять абстрактный синтаксис языка совместно с редакторами и генераторами языка (см. рис. 5). Редактор позволяют получить продвинутое и удобное окружение, которое учитывает специфику создаваемого ПОЯ по аналогии с *IDE* для языков общего назначения (*Eclipse*, *MS Visual Studio*, *IDEA*, и т. д.).

При использовании подобного инструментария разработчик, практически, не пишет код, а лишь манипулирует абстрактным представлением программы. Разумеется, *IDE* отображает эти изменения в тексте программы, но сути это не меняет – разработчик модифицирует не код, а абстрактное представление. Нечто подобное наблюдается и во время процесса рефакторинга.

### О целесообразности разработки ПОЯ

Проблемно-ориентированный язык остается довольно специфичным инструментом – он не вполне вписывается в объектно-ориентированную

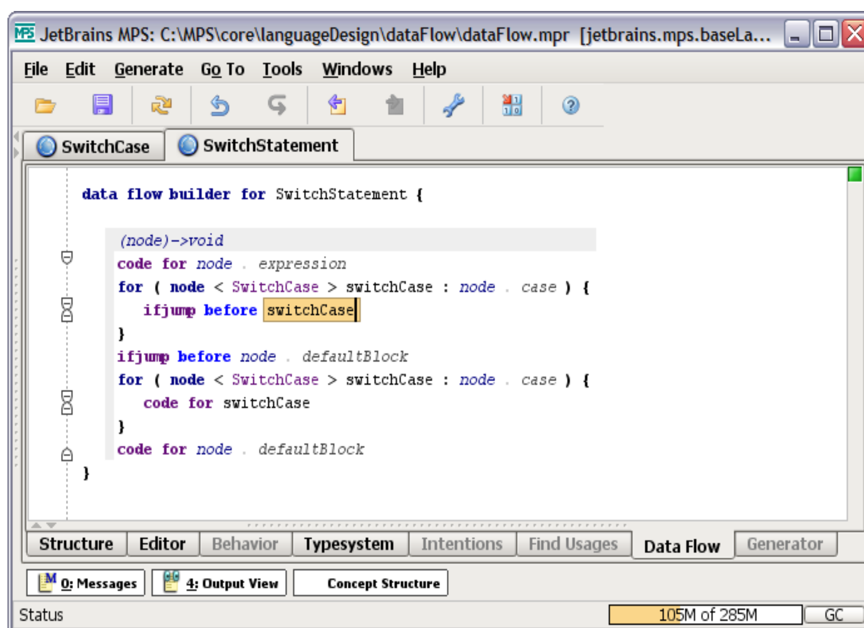


***Рис. 5.*** *JetBrains MPS – пример оболочки для разработки ПОЯ*

методологию разработки или в другие модели проектирования (такие как, например, *Agile Processes*), которые представляют собой существенный сдвиг в понимании процесса разработки. Парадигма ПОЯ предоставляет разработчику возможность в одном и том же проекте использовать 5–10 разных языков, в зависимости от задач конкретного модуля/подсистемы без изменения общего подхода к реализации. С учетом сказанного, решение о том, разрабатывать ли новый ПОЯ или остановиться на использовании существующих решений, в каждом конкретном случае должно приниматься отдельно и с большой аккуратностью.

Можно выделить следующие достоинства и недостатки ПОЯ, как опорные точки при принятии решения об использовании в ИТ-проекте обсуждаемой парадигмы.

*Достоинства*:

- *Использование труда экспертов* – это возможность исправить то состояние дел, которое наблюдается сейчас в области взаимодействия программистов и экспертов в предметных областях. Последнее является главным камнем преткновения на пути успешной работы над проектами.
- *Смена контекста выполнения* – даёт разработчику широкие возможности по оптимизации продукта.
- *Использование альтернативных моделей вычислений в рамках одного проекта* – позволяет гибко комбинировать парадигмы программирования (например, объектно-ориентированные и функциональные).
- *Резкое снижение уровня «семантического шума»* – позволяет разработчику сконцентрироваться на решении проблемы предметной области.

*Недостатки*:

- *Высокая стоимость разработки, миграции и поддержки* – ПОЯ, как и любой продукт, отходящий от господствующих тенденций, отличается от традиционных конкурентов и более высокой ценой.
- *Высокие требования к архитектуре ПОЯ* – сильно ограничивают спектр предложений на рынке труда.
- *Языковое смешение в рамках одного проекта/продукта* – значительно усложняет процесс детального понимания принципов работы системы.
- *Постепенное перерождение ПОЯ в язык общего пользования* – достаточно трудно сохранять баланс между расширениями ПОЯ и конструкциями языка общего пользования (примером трансформации языка для обработки текстовых данных в язык общего пользования может служить *AWK*).

## Выводы

Обсужденные технологические принципы и соответствующий инструментарий, безусловно, продолжат внедряться в ИТ-практику. Этот подход обеспечивает естественную фиксацию абстракций моделируемой предметной области и, соответственно, более адекватное понимание решаемой задачи. Упомянутые выше недостатки и проблемы применения проблемно-ориентированных языков, на наш взгляд, являются вполне преодолимыми, хотя и требуют затраты определенных усилий от разработчиков.

Одним из возможных путей развития методологии проблемно-ориентированных языков может стать создание удобных средств построения внешних языков, способных обеспечить преодоление «семантического барьера». Последнее позволит лингво-ориентированному программированию стать доступным для более широкого круга специалистов в ИТ-области [13].

## СПИСОК ЛИТЕРАТУРЫ


1. Liskov B. A history of CLU // History of programming languages. – N.Y.: ACM Press, 1996. – P. 471–510.
2. Dekorte S. Io, a Small Programming Language [Электронный ресурс]. – режим доступа: http://www.iolanguage.com/docs/talks/2005-10-OOPSLA/p003-dekorte.pdf. – 05.04.2008.
3. Knuth D.E. Literate programming // The Computer Journal. – 1997. – № 27 (2). – P. 63–72.
4. Lazy evaluation // Wikipedia article. [Электронный ресурс]. – режим доступа: http://en.wikipedia.org/wiki/Lazy_evaluation. – 05.04.2008.
5. Stahl N., Voelter M. Model-Driven Software Development: Technology, Engineering, Management. – John Wiley&Sons, 2006. – 444 p.
6. Goldberg A., Robson D. Smalltalk-80: The Language and Its Implementation. – AddisonWesley, Reading, MA, 1984. – 248 p.
7. Object (computer science) // Wikipedia article [Электронный ресурс]. – режим доступа: http://en.wikipedia.org/wiki/Object_(computer_science). – 05.04.2008.
8. Mandantenfahigen // Wikipedia article [Электронный ресурс]. – режим доступа: http://de.wikipedia.org/wiki/ Mandantenfahigkeit. – 12.05.2008.
9. Fowler M. Domain Specific Languages [Электронный ресурс]. – режим доступа: http://martinfowler.com/bliki/DomainSpecificLanguage.html – 12.05.2008.
10. Calcado Ph. Internal Domain Specific Languages [Электронный ресурс]. – режим доступа: http://fragmental.tw/research-on-dsls/domain-specific-languages-dsls/internal-dsls – 06.07.2008.
11. Domain Specific Language // Wikipedia article [Электронный ресурс]. – режим доступа: http://en.wikipedia.org/wiki/Domain_Specific_Language. – 12.05.2008.
12. Lloyd H. Nakatani, Mark A. Ardis Jargons for domain engineering // Proc. of the 2nd Conf. on Domain-specific languages, Austin, Texas, United States. – 1999. – P. 14–19.
13. Dmitriev S. Language Oriented Programming: The Next Programming Paradigm [Электронный ресурс]. – режим доступа: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/index.html. – 12.05.2008.


*Поступила 29.09.2008 г.*