Компонентная модель с декларативным описанием составных типов

Дубов Михаил Сергеевич
Национальный исследовательский университет – Высшая школа экономики,
Москва, Россия
msdubov@edu.hse.ru

Аннотация

В статье на примере языка VRML и стандарта JavaBeans рассматривается построение компонентной модели через ее декларативное описание. Излагаются основные принципы построения парсера декларативного языка разметки. На примере полученной модели демонстрируются некоторые проблемы, возникающие в ходе применения большинства современных компонентных архитектур. Обосновывается необходимость разработки компонентной архитектуры нового типа для их решения.

Ключевые слова: компонент, компонентная модель, парсер, компонентная архитектура, система типов.

I. Введение

Развитие компонентно-ориентированного программирования является важнейшим фактором перехода к промышленному производству крупных программных систем с длинным жизненным циклом [Lip06]. Построение программ на основе набора компонентов с заранее определенными интерфейсами, замена существующих компонентов и дополнение системы новыми в процессе сопровождения – все это те принципы, которые позволяют не только эффективно организовывать труд большого числа разработчиков, но и улучшать качество программного обеспечения в целом, снижая количество ошибок в коде. Употребляя в дальнейшем понятие «компонент», мы будем подразумевать функциональные группы программ, которые формируются на основе нескольких программных модулей, решают достаточно сложные автономные задачи и обладают четко определенным интерфейсом взаимодействия с внешним миром [Lip06].

На сегодняшний день компонентное программирование все еще находится в развивающемся состоянии. Регулярно появляются новые разработки в области компонентных архитектур, многие уже существующие стандарты подвергаются доработке. Несмотря на это, подавляющее большинство современных концепций все еще обладают рядом характерных недостатков.

Чтобы продемонстрировать некоторые из них, мы рассмотрим построение компонентной модели из JavaBeans-компонентов на основе ее декларативного описания с помощью языка разметки VRML. *VRML* является стандартом ISO для демонстрации трёхмерной интерактивной векторной графики в WWW [ISO04]. *JavaBeans* представляет собой спецификацию компании Sun Microsystems, определяющую правила написания классов на языке программирования Java для описания повторно используемых программных компонентов [Sun97]. Выбор VRML – языка моделирования виртуальной реальности – в качестве основы для описания компонентной модели обусловлен тем, что 3D-сцены, как правило, достаточно естественным образом декомпозируются в набор взаимодействующих компонентов. При этом составные 3D-объекты, описываемые в VRML с помощью вложенных узлов, так же просто трансформируются в набор составных JavaBeans-компонентов. Таким образом, 3D-графика является хорошим примером применения компонентного подхода для упрощения процесса разработки. Следование спецификациям JavaBeans обеспечивает необходимый уровень абстракции, который позволяет использовать полученные в результате обработки VRML-сцены компоненты не только для визуализации 3D-сцен (та цель, которую преследует ряд других проектов, например, [J3D09]), но также и для определения связей между этими компонентами и реализации их совместного поведения.

При реализации некоторых конструкций VRML посредством JavaBeans-компонентов выявляется ряд проблем, неразрешимых в рамках рассматриваемой компонентной архитектуры. В качестве возможного решения этих проблем рассматривается построение компонентной архитектуры нового типа, позволяющей определять составные типы данных во время исполнения. Разработка такой архитектуры ведется в настоящее время в рамках научно-исследовательского проекта на отделении программной инженерии факультета бизнес-информатики НИУ-ВШЭ. В конце данной работы анализируются преимущества будущей разработки с точки зрения развития компонентно-ориентированного программирования в целом. В этой статье мы не будем детально касаться вопросов реализации новой компонентной архитектуры. За подробностями читатель может обратиться к [GSh11].

II. Построение модели с помощью JavaBeans-компонентов

Описание сцены на языке VRML состоит из так называемых *узлов (nodes)*, каждый из которых является

представлением некоторой сущности: это может быть, например, трехмерная поверхность (сфера, прямоугольный параллелепипед и др.), либо же одно из ее свойств (цвет, материал и др.). Вместе все эти узлы образуют *граф сцены (scene graph)*. Описание каждого узла VRML содержит его тип, имя (опционально), поля с их значениями, а также некоторую специальную информацию, которую мы не будем рассматривать в данной работе в целях упрощения. Значения полей узлов могут быть скалярными значениями, массивами скалярных значений, а также, в свою очередь, другими узлами. Это означает, что VRML-сцены могут состоять из большого числа вложенных узлов. В (1) приведен простейший пример типичной VRML-сцены.

```
DEF shape1 Shape
{
  appearance Appearance
  {
    material Material
    {
      diffuseColor 0 0.5 0
    }
  }
  geometry Sphere {}
}

USE shape1
```
(1)

В примере описывается узел типа `Shape`, которому с помощью языковой конструкции `DEF` присваивается имя `shape1`. Полями этого узла являются `appearance` и `geometry`, значения которых задаются узлами соответствующих типов. Полями узла `Appearance`, согласно стандарту [ISO04], являются `material`, `texture` и `textureTransform`, однако здесь мы можем ограничиться заданием значения одного лишь первого поля – значения остальных будут установлены по умолчанию. Поле `diffuseColor` имеет значение скалярного типа `SFColor`, которое задается тройкой вещественных чисел в формате с плавающей точкой (тип `SFFloat` в VRML). Поле `geometry` задает тип поверхности; в данном случае это сфера. Конструкция `USE` представляет собой пример повторного использования ранее определенного именованного узла. Согласно стандарту [ISO04], повторное использование подразумевает не копирование уже существующего узла, а простую вставку ссылки на него в граф сцены.

Переход от узлов VRML-сцены к набору JavaBeans-компонентов не представляет особой трудности. Естественным образом каждому из узлов, описанных в стандарте [ISO04] (количество этих узлов ограничено) ставится в соответствие Java-класс, оформленный по правилам спецификации JavaBeans [Sun97]. Каждый такой класс содержит конструктор без параметров с модификатором доступа `public`, инициализирующий поля класса (соответствующие описанному в стандарте набору полей для каждого из VRML-узлов) значениями по умолчанию. Все поля объявлены с модификатором доступа `private`. Доступ к ним осуществляется с помощью публичных setter'ов и getter'ов, именование которых ведется по строгим правилам: для поля узла с именем `xxx` соответствующие методы доступа на чтение и запись должны иметь имена `getXxx` и `setXxx`. Как будет показано ниже, это требование чрезвычайно важно для успешного построения VRML-парсера. Кроме того, согласно спецификации [Sun97], каждый из классов объявляется как сериализуемый. Ниже в качестве примера приведен каркас класса Appearance (2).

```
import java.io.Serializable;

public class Appearance
    extends Node implements Serializable {

    public Appearance() {
        // Задание значений по умолчанию
    }

    public void setMaterial(Material m) {
        material = m;
    }

    public Material getMaterial() {
        return material;
    }

    //...

    private Material material;

    //...
}
```
(2)

Вместе все эти классы, соответствующие возможным типам узлов VRML, образуют иерархию, во главе которой находится класс `Node`. Типы значений (такие как `SFVec2f`, `SFColor` и другие) аналогичным образом реализуются как JavaBeans-компоненты и являются наследниками класса `ValueType`. В роли самого базового класса выступает `VRMLType`. Являясь абстрактным, этот класс не несет в себе никакой функциональности, зато несколько упрощает реализацию парсера, позволяя работать и с типами-узлами, и с типами значений по одной схеме. Не менее важно и выделение типов значений в отдельную ветвь иерархии: наличие метода `parse` в базовом классе `ValueType` означает отсутствие в парсере VRML методов для чтения значений всех этих типов. Это не только позволяет «разгрузить» реализацию парсера, но и дает возможность сделать ее более независимой от конкретной системы типов.

Интересно отметить, что класс `Node` играет двоякую роль: с одной стороны, он является базовым для всех типов узлов; с другой, фактически сам он реализует VRML-тип `SFNode` (тип полей, значениями которых являются вложенные узлы). Поэтому он так же, как и `ValueType`, наследует классу `VRMLType`. На рис. 1 приведена UML-диаграмма классов, которая отражает

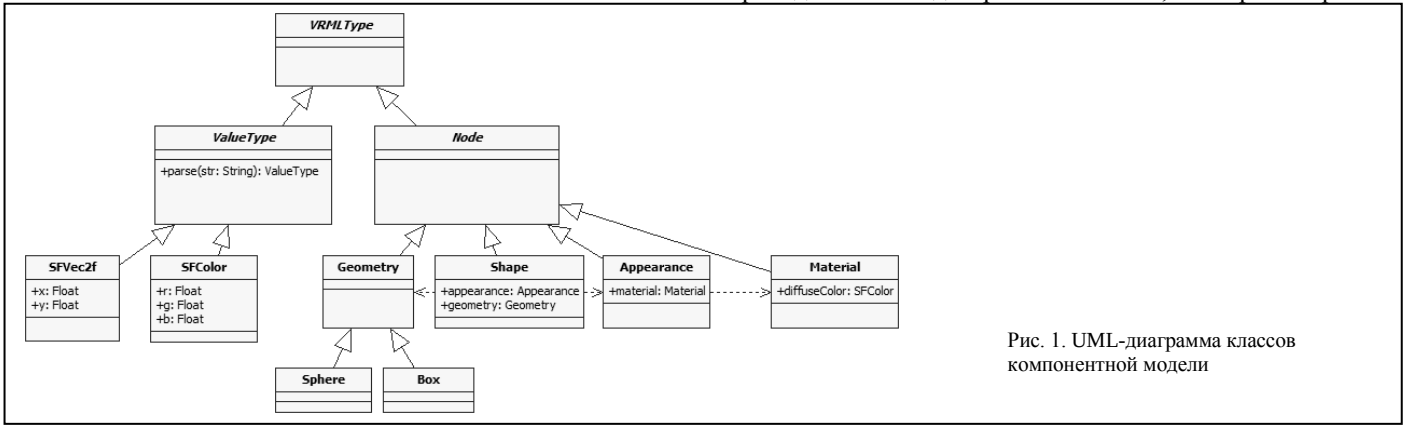

Рис. 1. UML-диаграмма классов компонентной модели

описанную выше архитектуру на примере небольшого подмножества компонентов и базовых типов языка.

Упомянутая выше независимость парсера от конкретной системы типов важна по следующим причинам. Фактически, построение компонентов на основе их декларативного описания представляет собой *десериализацию (deserialization)*. Здесь можно провести параллель со стандартными классами библиотек Java `XMLEncoder` и `XMLDecoder`, позволяющими проводить сериализацию и десериализацию JavaBeans-компонентов с помощью их XML-представления [HC08b]. Эти классы, однако, используют не самый очевидный подход: в основе их работы лежит сохранение последовательности действий, необходимой для создания JavaBeans-компонента, и последующее использование этих действий для восстановления состояния компонента. Описанное же выше решение представляется нам более удачным: использование VRML позволяет естественным образом хранить не последовательность действий для создания компонентов, но само их состояние, сам граф объектов. При этом хранение данных в формате VRML оказывается более компактным, чем при использовании XML. Становится очевидным, что область применения предлагаемой модели может быть шире, чем описание простых 3D-сцен. Таким образом, вполне возможна ситуация, когда нам придется выйти за рамки стандарта VRML и предлагаемой им системы типов. Вот почему в ходе реализации компонентов и парсера VRML важно учитывать возможность их расширяемости.

Предполагается, что результатом синтаксического анализа (работы парсера) декларативного описания модели на языке VRML является *ориентированный ациклический граф (Directed Acyclic Graph, DAG-граф)* сцены, который затем может быть подвергнут траверсированию [Sed10] с целью, например, визуализации узлов. Фактически такой граф является лесом, где каждое из направленных деревьев имеет в качестве корня один из узлов первого уровня вложенности, а в качестве листьев – узлы типов значений или типа `SFNode`. Пример такого графа, соответствующий (1), приведен на рис. 2. В рамках компонентной модели JavaBeans направленные ребра, связывающие узлы, реализуются с помощью свойств соответствующих классов. Корневые узлы (отделенные пунктирной линией на рис. 2) выделяются в массив, ссылка на который и является точкой входа при обходе графа.

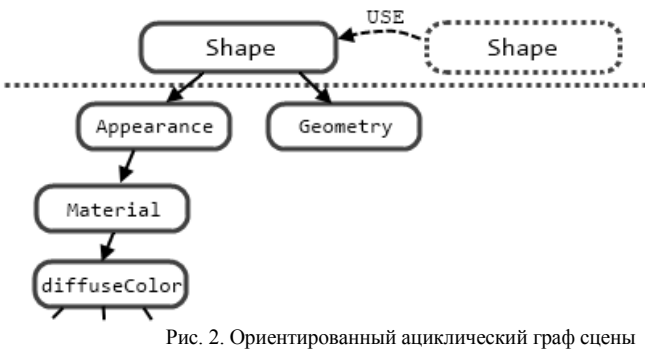

Рис. 2. Ориентированный ациклический граф сцены

III. Реализация парсера VRML

В отличие от схожих работ (см., например, [Gri10]), при построении парсера мы используем достаточно строгий подход к синтаксическому анализу исходных текстов, основанный на формальной грамматике языка VRML. Данная грамматика является *контекстно-свободной (context-free)* и приведена в форме *Бэкуса-Наура (Bakus-Naur Form, BNF)* в стандарте языка [ISO04] (3).

```
vrmlScene ::=
        statements ;
statements ::=
        statements statement |
        empty ;
statement ::=
        nodeStatement |
        protoStatement |
        routeStatement ;
nodeStatement ::=                                   (3)
        node |
        DEF nodeNameId node |
        USE nodeNameId ;
...

node ::=
        nodeTypeId { nodeBody } |
        Script { scriptBody } ;
...
```

Для любой контекстно-свободной грамматики существует парсер, позволяющий провести синтаксический анализ за время $O(n^3)$, где n – длина файла [ALSU06]. Однако благодаря простоте синтаксиса VRML легко построить парсер, работающий за время $O(n)$. Наш парсер считывает входной файл слева направо, строя по ходу своей работы направленный ациклический граф сцены из JavaBeans-компонентов, отрабатывая, таким образом, за один линейный проход. Парсер реализован [Git12] на языке программирования Java.

Единственное отступление от правил грамматики языка, которое мы позволили себе при реализации синтаксического анализатора, состоит в отказе от использования примитивных типов VRML (`SFBool`, `SFFloat`, `SFInt32`) в пользу примитивных типов языка Java (`boolean`, `float`/`double`, `int`). Причины такого перехода были изложены в предыдущем разделе.

Важной особенностью грамматики VRML является ее *неоднозначность (ambiguity)*, которая означает, что результат синтаксического анализа не может быть однозначно определен на основе разбора одной лишь входной строки. Дело в том, что для корректного считывания значений полей необходима дополнительная семантическая информация об их типе. Для получения такой информации в парсере широко используется реализованный в библиотеках Java механизм *рефлексии (reflection)* [HC07a]. Этот механизм позволяет получать сигнатуры свойств (getter'ов и setter'ов) класса во время работы парсера. В зависимости от типа поля, значение которого считывается в данный момент, парсер вызывает соответствующие методы для их чтения. Таким образом, парсер управляется структурой JavaBeans-компонентов. Если типом считываемого поля является `SFNode` и мы имеем дело с вложенным узлом, парсер заносит ссылку на

обрабатываемый узел в специальный стек и рекурсивно вызывает процедуру для чтения узла. Для инстанциирования JavaBeans-компонента для только что прочитанного узла необходим доступ к соответствующему классу по его имени, для чего опять же используется рефлексия.

Рассмотрим реализацию парсера более подробно. Анализ кода, как правило, проходит в три этапа: *лексический анализ (lexical analysis)*, *синтаксический анализ (syntax analysis)* и *семантический анализ (semantic analysis)*. Простота языка VRML и возможности библиотек Java позволяют выполнять все эти три стадии за один проход по файлу.

1) *Лексический анализ* имеет своей задачей преобразование входной строки символов в набор *лексем (токенов)* – базовых конструкций языка, таких как операторы или идентификаторы. Класс `StreamTokenizer` в Java позволяет автоматически решать эту задачу. Все, что требуется от программиста – это настроить объект этого класса в соответствии с формальной грамматикой языка (4).

```
public class VRMLParser {

    protected void setUpTokenizer() {
        tokenizer.resetSyntax();
        tokenizer.commentChar('#');
        tokenizer.quoteChar('"');
        tokenizer.ordinaryChar('{');
        tokenizer.ordinaryChar('}');
        tokenizer.ordinaryChar('[');
        tokenizer.ordinaryChar(']');
        //...
    }

    public ArrayList<Node>
    parse(InputStreamReader reader)
         throws SyntaxError, IOException {

        tokenizer = new StreamTokenizer
            (new BufferedReader(reader));
        setUpTokenizer();
        //...
    }

    private StreamTokenizer tokenizer;
}
```
(4)

В ходе настройки объекта `StreamTokenizer` указываются так называемые *терминальные символы (terminals)* – элементарные символы языка, определяемые грамматикой. Такими символами являются, например, фигурные или квадратные скобки. Все, что требуется для чтения следующей лексемы из потока после настройки объекта `StreamTokenizer` – это вызвать специальный метод `nextToken`, после чего соответствующая лексеме строка содержится в поле `sval` (5).

```
tokenizer.nextToken();
String lookahead = tokenizer.sval;
```
(5)

2) *Синтаксический анализ* является основной задачей парсера. Он подразумевает разбор потока лексем, полученных в ходе лексического анализа, и представление их в некоей древовидной структуре в соответствии с грамматикой языка. В нашем случае в результате синтаксического анализа непосредственно генерируется граф сцены. Для построения парсера на основе грамматики, заданной в форме Бэкуса-Наура, используется так называемый *метод «рекурсивного спуска» (recursive-descent parsing)* [ALSU06]. В соответствии с формальной грамматикой, которая определяет более общие конструкции через выражения низшего порядка, парсер, работающий по принципу «рекурсивного спуска», разбирает исходный текст, двигаясь в *нисходящем (top-down)* направлении. При этом каждому из правил грамматики (3) соответствует специальный метод в классе `Parser` (в (6) – пример для правила `node`).

```
public class VRMLParser {

    //...

    public ArrayList<Node>
    parse(InputStreamReader reader)
         throws SyntaxError, IOException {

        tokenizer = new StreamTokenizer
            (new BufferedReader(reader));
        setUpTokenizer();

        ArrayList<Node> sceneGraph =
                              parseScene();
        return sceneGraph;
    }

    //...

    private boolean parseNode() {

        return (lookahead("Script") &&
         match("Script") && match("{") &&
         parseScriptBody() && match("}")) ||

        (lookaheadIsId()&& matchTypeId() &&
         instantiateNode() &&
         match("{") && parseNodeBody() &&
         match("}"));
    }


    private StreamTokenizer tokenizer;
}
```
(6)

Метод `lookahead` сравнивает следующую лексему в потоке с указанным значением; этот метод определяет выбор одного из разветвлений в грамматических правилах. Метод `match` проверяет, совпадает ли следующая лексема в потоке с указанным терминальным символом, и генерирует синтаксическую ошибку в случае отрицательного результата. В случае же положительного метод считывает следующий токен. Метод `matchTypeId` работает аналогично, но проверяет, является ли тип текущего узла корректным идентификатором. После этого с помощью механизма рефлексии инстанциируется соответствующий типу текущего узла класс (7). Для хранения ссылок на именованные узлы используется хэш-таблица `defNodesTable`.

```
private boolean matchTypeId() {

    if(lookaheadIsId()) {

        currentType = lookahead;
        nextToken();

        return true;

    } else {

        throw new SyntaxError(/*...*/);
    }
}

private boolean instantiateNode() {

    try {
        Node node = (Node)
         (Class.forName(currentType).
                          newInstance());

        // If there was the "DEF" keyword
        if (currentId != null) {
            node.setId(currentId);
        // Store the node in the hash table
            defNodesTable.put(currentId,
                              node);
        }

        //...

        return true;

    } catch (Exception e) {
          return false;
    }
}

private HashMap<String, Node> defNodesTable;
```
(7)

Наибольшую опасность представляют те выражения грамматики, где самый левый нетерминальный символ в одной из ветвей в правой части совпадает с левой частью правила (8).

```
statements ::=
        statements statement |
        empty ;
```
(8)

Эти правила принято называть *леворекурсивными (left-recursive)* [ALSU06]. Очевидно, что при реализации таких правил в нисходящем парсере неизбежно возникновение бесконечной рекурсии: соответствующий метод `parseStatements` будет вызывать сам себя. Для предотвращения такого исхода требуется устранение левой рекурсии из правил грамматики, что в данном случае может быть проделано довольно просто (9).

```
statements ::=
        statement |
        statement statements |
        empty ;
```
(9)

Построение парсера по принципу рекурсивного спуска позволяет добиться простоты и читаемости кода его реализации. Кроме того, такой парсер легко расширяем при добавлении в него анализаторов новых языковых конструкций. К недостаткам же этого подхода можно отнести наличие в коде большого количества методов с вспомогательной функциональностью, что может привести к возникновению довольно глубокой рекурсии при работе анализатора.

3) Наконец, *семантический анализ*, как было описано выше, сводится к получению с помощью механизма рефлексии типа поля, значение которого должно быть считано следующим. Наглядный пример использования рефлексии был приведен выше, поэтому на реализации семантического анализа в данной работе мы останавливаться не будем.

IV. Выражения PROTO

Одним из наиболее интересных механизмов языка VRML является возможность определять пользовательские типы с помощью выражений PROTO (10).

```
PROTO UserShape
[field SFColor figColor  0 0.5 0]
{
  Shape
  {
    appearance Appearance
    {
      material Material
      {
        diffuseColor IS figColor
      }
    }
    geometry Sphere {}
  }
}

UserShape
{
  figColor 0.3 0.4 0.9
}

UserShape
{
  figColor 1 1 1
}
```
(10)

`PROTO` позволяет вводить пользовательские типы узлов через задание прототипа, параметризованного набором полей (например, в (10) параметром является поле `figColor`, значение которого в дальнейшем связывается операцией `IS` с полем `diffuseColor` узла `Material`).

Именно выражения `PROTO` представляют собой наибольшую проблему с точки зрения эффективной реализации в компонентной модели JavaBeans. В качестве возможного решения можно рассматривать клонируемые (реализующие интерфейс `Cloneable`) классы-узлы, хранение полей и связанных с ними узлов для каждого `PROTO` в специальной структуре данных (например, во вложенных хэш-таблицах), создание обычного узла при обработке выражения `PROTO` и его копирование при инстанциировании отдельных экземпляров (11).

```
import java.io.Serializable;

public class Shape
    extends Node
    implements Serializable, Cloneable {

    //...
}

public class VRMLParser {

    //...                                          (11)

        Node protoInstance =
                (Node) protoNode.clone();

    //...
}
```

Похожих принципов придерживаются некоторые реализации VRML на Java, например, [J3D09]. Очевидно, однако, что такой подход чрезвычайно неэффективно расходует память: действительно, при каждом создании нового объекта из `PROTO` посредством клонирования узлов копируются абсолютно все поля, а не только те, которые могут менять свои значения от объекта к объекту. В свете того, что такое расточительное использование памяти имеет место при реализации 3D-графики, которая может быть очень чувствительна к неоптимальному использованию ресурсов, проблема становится особенно острой.

V. Необходимость расширения компонентной архитектуры

Обозначенную выше проблему можно интерпретировать двояко. С одной стороны, причиной неэффективности приведенной реализации конструкции `PROTO` естественным образом можно считать тот факт, что в языке программирования Java (являющегося *класс-ориентированным (class-based)* языком) отсутствует возможность программирования с помощью прототипов. Все, что может быть сделано здесь во время выполнения – это клонирование уже существующих объектов, что и приводит к копированию всех полей и к неоправданному растрачиванию памяти.

Проблемы при объектно-ориентированном проектировании, вызываемые необходимостью перехода от *прототип-ориентированной (prototype-based)* к *класс-ориентированной (class-based)* парадигме, давно обсуждались исследователями, в том числе применительно к VRML [Bee97]. В качестве одного из решений было разработано расширение языка – VRML++, позволявшее описывать сцены как с помощью прототипов (выражений `PROTO`), так и с помощью классов (выражений `CLASS`), и, таким образом, объединявшее в себе две эти парадигмы [Die97]. Реализация этого языка, однако, представляла собой простой препроцессор, транслировавший код на VRML++ в эквивалентный VRML-код, соответствующий стандарту [ISO04]. Таким образом, эта разработка никак не затрагивала вопросы эффективной реализации языка VRML и не решала проблемы ее оптимизации.

Другая интерпретация указанной выше проблемы – и она более полно раскрывает ее суть – заключается в том, что ни Java, ни JavaBeans не предоставляют возможность определения новых типов во время исполнения. Обладая такой возможностью, мы могли бы по ходу синтаксического анализа VRML-файла определять новые типы для выражений PROTO и инстанциировать позднее их экземпляры. При этом определение новых типов во время исполнения позволило бы абстрагироваться от того, работаем ли мы в рамках прототип-ориентированной или же класс-ориентированной парадигмы. Фактически, разработав компонентную модель с определением типов во время исполнения, мы сотрем границу между двумя этими подходами.

Нельзя не отметить, что на данный момент создание типов во время исполнения выполнимо с помощью вспомогательного средства – генерации кода и вызова компилятора [GSh11]. Именно этот принцип лежит в основе работы многих современных GUI. Очевидно, однако, что такой подход к определению новых типов вряд ли можно назвать элегантным и производительным. Учитывая то, что в рамках работы с трехмерной графикой достижение высокой производительности является одной из наиболее критичных целей, такой способ оказывается неприемлемым. Более того, возможность обращения к компилятору вообще отсутствует в целом ряде сред, например во встраиваемых системах [GSh11].

Способность компонентной архитектуры к динамическому созданию новых типов важна и по целому ряду других причин. Так, естественным образом задача о генерации новых типов возникает при проектировании программной системы из компонентов. В спецификации [Sun97] дано следующее определение: «JavaBean – это повторно используемый программный компонент, которым можно манипулировать визуально в инструменте сборки» [Gri10]. В простейшем случае мы можем представить себе ситуацию, когда в такой визуальный редактор проектировщик добавляет, например, компонент «кнопка» и компонент «текстовое поле». Пусть при нажатии на кнопку в текстовом поле по некоторому алгоритму генерируется псевдослучайное число. Эти два компонента в связке уже реализуют некоторую автономную функциональность, и, следовательно, претендуют на то, чтобы быть выделенными в отдельный компонент, доступный в дальнейшем для повторного использования. Визуально это можно представить себе как перетаскивание кнопки и текстового поля уже как единой сущности из области редактора на панель компонентов. Как и в примере с реализацией прототипов, единственным доступным на данный момент решением подобной задачи является непосредственная генерация кода, которая неоправданно усложняет процесс проектирования.

Приведенные выше рассуждения позволяют сделать заключение о необходимости создания новой компонентной архитектуры, способной в динамике определять новые типы и инстанциировать экземпляры этих типов. В следующем разделе мы рассмотрим некоторые существующие в этой области решения и обоснуем необходимость разработки новой архитектуры.

VI. Описание существующих решений

Приведем примеры существующих на сегодняшний день технологий, позволяющих осуществлять динамическое создание новых типов.

Рассмотренная выше прототип-ориентированная парадигма обладает, несомненно, лучшими динамическими свойствами в сравнении с класс-ориентированными языками. Многие языки здесь позволяют менять прототипы во время выполнения программы, определяя, таким образом, новые типы. Примером может служить широко распространенный в современных Web-браузерах JavaScript.

По аналогии, от класс-ориентированных языков можно было бы ожидать возможности динамического изменения структуры классов во время выполнения программы. Однако, хотя уже Smalltalk предоставлял такую функциональность, на сегодняшний день лишь немногие языки способны менять свои классы в динамике: здесь можно выделить Objective-C, Common Lisp, Python и Ruby.

Одна из самых последних и интересных разработок – это так называемые *провайдеры типов (type providers)*, появившиеся в последней версии (3.0) языка F# [Sym11]. Предложенная технология позволяет определять новые (однако, довольно узкоспециализированные) типы на основе схем данных из внешних источников (это могут быть, например, описания в XML-формате) и рассматривается как более предпочтительная альтернатива использованию кодогенераторов.

В чем же заключается необходимость разработки новой архитектуры при наличии рассмотренных выше решений? Пожалуй, основная причина заключается в том, что ни одно из них не предоставляет достаточных возможностей для полноценной реализации компонентного подхода. Для всех рассмотренных примеров характерно наличие слабого инструментария определения достаточно сложных составных типов, а также средств задания совместного поведения компонентов. Слабо реализованы такие важные для компонентно-ориентированного подхода принципы, как событийное связывание и разделение компонентами свойств. Именно эти принципы лежат в основе новой архитектуры.

Не в последнюю очередь следует назвать и отсутствие решений для языка Java, являющегося на сегодняшний день одним из самых используемых в промышленной разработке ПО [Tio12]. Нашу разработку планируется осуществить в виде набора Java-классов, которые расширяют окружение, предоставляемое Java-машиной, и представляют собой как бы «надстройку» над JVM.

VII. Заключение

Разрабатываемая компонентная архитектура нового типа обладает достаточным потенциалом для того, чтобы вывести компонентно-ориентированное программирование на качественно новый уровень. Гибкость и универсальность, обеспечиваемые возможностью создавать новые типы во время исполнения при сохранении всей мощи компонентного подхода, могут значительно облегчить работу проектировщикам, а также позволить программным системам развиваться гораздо более быстрыми темпами, сохраняя высокую степень свободы от ошибок.

Несомненно, применение новой разработки позволит решить описанную в данной статье проблему с реализацией VRML-прототипов на Java. Ожидается значительный прирост в производительности моделирования 3D-сцен, описанных на языке VRML. Полученный выигрыш можно будет оценить и измерить количественно в ходе тестирования.

Одним из применений может также стать оптимизация реализации и расширение возможностей некоторых фреймворков для Java-платформы, например, широко известного Spring [Spr12], который часто рассматривают как замену архитектуры Enterprise JavaBeans.

Другие возможные применения разработки составляют основной предмет дальнейших исследований.

Литература

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, Compilers: principles, techniques, and tools, 2nd ed. MA: Prentice Hall, 2006.

[Bee97] C. A. Beeson, “An object oriented approach to VRML development”, in proceedings of VRML’97, 1997.

[Die97] S. Diehl, “VRML++: a language for object-oriented virtual reality models”, in proceedings of the 24th international conference on technology of object-oriented languges and systems TOOLS Asia’97, Beijing, China, 1997.

[Git12] Component model: VRML parser and JavaBeans components. [Электронный ресурс]. URL: http://www.github.com/msdubov/Component-model

[Gri10] Е. М. Гринкруг, “Использование JavaBeans-компонент в 3D моделировании”, Бизнес-информатика №3(13), 2010.

[GSh11] Е. М. Гринкруг и А. Р. Шакуров, “Компонентная архитектура с определением типов во время исполнения”, Бизнес-информатика №1(15), 2011.

[HC07a] C. S. Horstmann and G. Cornell, Core Java, 8th ed., vol. 1: Fundamentals. MA: Prentice Hall, 2007.

[HC08b] C. S. Horstmann and G. Cornell, Core Java, 8th ed., vol. 2: Advanced features. MA: Prentice Hall, 2008.

[ISO04] ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2004 — Virtual Reality Modeling Language (VRML). [Электронный ресурс]. URL: http://www.web3d.org/x3d/specifications/vrml

[J3D09] Java 3D VRML97 loader project. [Электронный ресурс]. URL: http://www.java.net/projects/j3d-vrml97

[Lip06] В. В. Липаев, Программная инженерия. Методологические основы. М: ТЕИС, 2006.

[Sed10] R. Sedgewick, Algorithms in Java, 4th ed., CA: Addison-Wesley Educational Publishers Inc., 2010.

[Spr12] The Spring Framework. [Электронный ресурс]. URL: http://www.springsource.org

[Sun97] Sun Microsystems, JavaBeans Specification v1.0.1, July 1997. [Электронный ресурс]. URL: http://java.sun.com/products/javabeans/docs/spec.html

[Sym11] D. Syme, “F# 3.0 Information Rich Programming” [Электронный ресурс]. URL: http://bit.ly/HiGTWg

[Tio12] TIOBE Programming Community Index for Match 2012 [Электронный ресурс]. URL: http://bit.ly/HiH5VJ