ПРОТОТИПЫ И ТИПЫ СОСТАВНЫХ ОБЪЕКТОВ В КОМПОНЕНТНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ[1]

Е.М. Гринкруг,
кандидат технических наук, доцент кафедры управления разработкой программного обеспечения, отделение программной инженерии, факультет бизнес-информатики, Национальный исследовательский университет «Высшая школа экономики»

Адрес: 101000, г. Москва, ул. Мясницкая, д. 20
E-mail: egrinkrug@hse.ru

В работе рассматривается компонентно-ориентированная программная архитектура, реализованная для Java-платформы и поддерживающая 'event-driven' вычисления. Используемая компонентная модель расширяет динамические возможности стандартной компонентной модели JavaBeans и поддерживает композицию составных компонент во время исполнения без кодогенерации. Показано, что в рамках стандартной компонентной модели JavaBeans динамическая композиция компонент сопряжена с переходом от объектно-ориентированного программирования, используемого при создании базовых компонент, к prototype-ориентированной идеологии программирования при использовании составных компонент. Предложено обобщение понятия типа объектов, позволяющее единообразно оперировать как базовыми компонентами, так и динамически скомпонованными компонентами. Реализация предложенного обобщения типов объектов опирается на дополнительный уровень виртуализации, который обеспечивает среду исполнения для программных компонент, определенных в соответствии с предложенной компонентной моделью. Рассматриваются способы реализации различных типов объектов, с которыми оперирует среда исполнения, и основные предоставляемые ею операции. Предложен способ динамического создания составных компонент, основанный на преобразовании составного прототипа в инстанцируемый тип составных объектов. Инстанцирование составных компонент обеспечивает более эффективное функционирование приложений (в смысле ресурсов памяти и времени), чем клонирование их составных прототипов. Динамически определяемые составные компоненты могут использоваться объектно-ориентированным образом наравне с базовыми компонентами – как для реализации функциональности в приложениях, так и в качестве элементов композиции составных прототипов, преобразуемых в новые составные компоненты. Предлагаемая компонентная архитектура может использоваться при реализации определяемых пользователем типов в декларативных языках моделирования.

Ключевые слова: программный компонент, компонентная модель, тип, прототип, интерфейс, реализация.

[1] Исследование осуществлено в рамках Программы фундаментальных исследований НИУ ВШЭ в 2013 году.

1. Введение

Компонентно-ориентированное программирование (Component-Based Software Engineering, CBSE) является перспективным направлением программной инженерии [1]. Составные части программных продуктов – компоненты, – создаются и используются в соответствии с компонентной моделью, которая определяет, что является компонентом и что и каким образом из компонент можно собирать.

История CBSE началась с конференции 1968 года [2], где впервые был поставлен вопрос об использовании компонент при промышленной разработке программного обеспечения, по аналогии с использованием компонентного подхода в других инженерных отраслях. К настоящему времени используются разные компонентные модели [3; 4; 5; 6], в разной степени зависящие от платформы и/или от области применения. Собственно понятие архитектуры программной системы формулируется в терминах компонент [7], и всякая программа, как множество инструкций, является их композицией, то есть имеет компонентную природу уже на самом нижнем уровне.

Мы рассматриваем компонентную архитектуру применительно к Java-платформе, наиболее популярной в последние годы [8]. Предметной областью нашего рассмотрения является моделирование с использованием event-driven вычислений. Предпосылкой работы послужил опыт реализации [9] подмножества языков моделирования виртуальной реальности [10; 11] с использованием компонентной модели JavaBeans [12] – наиболее популярной компонентной модели, применяемой в Java-платформе.

В работе рассматриваются вопросы реализации составных компонент, определяемых пользователем. Решение этих вопросов средствами компонентной модели JavaBeans затруднено по причинам, обсуждаемым в разделе 2. Раздел 3 посвящен обобщению понятия типа объектов для компонентной модели; разделы 4, 5 и 6 содержат общее описание представления типов и реализации их основных разновидностей (hardcoded-типов и composed-типов). В разделе 7 кратко излагается способ динамического создания составного компонента как результата преобразования составного прототипа в составной тип. Заключение указывает непосредственную область применения результатов и направление дальнейших работ.

2. Достоинства и недостатки компонентной модели JavaBeans

Компонентная модель JavaBeans является первой из предложенных для Java-платформы. Ее популярность определяется простотой, предоставляемыми возможностями и инструментальными средствами [13]. Исходное определение гласит, что JavaBeans-компонент – это повторно используемый программный компонент, которым можно визуально манипулировать в инструменте сборки [12]. Компонентная модель JavaBeans определяет компонент как Java-класс, который является инстанциируемым без параметров, поддерживает стандартную сериализацию и предназначен для манипулирования в специальной инструментальной среде.

С позиций компонентной модели JavaBeans к обычным классам предъявляются весьма необременительные требования для того, чтобы они могли служить компонентами этой модели. Поддержка компонент инструментальными средствами опирается не на расширения языка (как в ComponentJ [14]) и не на реализацию специальных интерфейсов (как в компонентной среде OSGi [3]), а просто на соблюдение достаточно простых соглашений о кодировании (JavaBeans design patterns [12]).

Компонентно-ориентированное программирование подразумевает, что имеется набор базовых компонент, которые заготавливаются без привязки к конкретному контексту использования. Из экземпляров базовых компонент складывается (с применением соответствующих инструментов) нужное приложение. Такой производственный процесс давно обеспечил прорыв в области аппаратных средств, но еще недостаточно широко применяется в индустрии программного обеспечения.

Использование компонентной модели JavaBeans в сочетании со статическими, компиляционными средствами давно поддерживается различными IDE. Они, однако, уступают по своим динамическим возможностям инструменту BeanBox, изначально разработанному для демонстрации возможностей этой компонентной модели [13]. Но и там имеют место ограничения динамики принципиального характера.

Базовые компоненты как Java-классы создаются в результате компиляции. При этом уже возможна их композиция статическими средствами: составной компонент кодируется на исходном языке с использованием компонент-элементов как библиотечных классов.

Созданные в статике компоненты могут загружаться в компонентную среду, где они представлены как инстанциируемые типы объектов. Созданные экземпляры (инстансы) компонент имеют визуальные представления, позволяющее манипулировать ими. Среда обеспечивает обзор и редактирование значений свойств экземпляров. При некотором усовершенствовании среда может поддерживать определение зависимостей экземпляров: одни экземпляры могут быть значениями свойств у других.

В результате манипулирования создаются составные объекты, которые могут сохраняться и восстанавливаться средствами сериализации/десериализации (в различных форматах). Однако, все такие составные объекты имеют одинаковый тип (не имеют своего типа); они не инстанциируются своим типом, а клонируются (возможно, через сериализацию/десериализацию).

Таким образом, начав с производства базовых компонент средствами объектно-ориентированного (class-based) языка, мы закончили составным объектом-прототипом и перешли, в сущности, к prototype-based программированию. Этот факт представляется нам противоречием, которое препятствует получению «идеологически замкнутой» технологии: нельзя с помощью манипуляций базовыми компонентами (инстанциируемыми типами) получить новый составной компонент как инстанциируемый тип. То есть, нельзя осуществлять композицию в рамках единой идеологии.

Можно «вернуться» на уровень языка реализации базовых компонент и закодировать в нем новый тип, используя сериализованный прототип как декларативное описание для его конструктора (так поступают IDE, такие как NetBeans, например). Ранее, некоторые инструменты сразу поддерживали генерацию исходного кода языка реализации (по мере композиции)[2].

Проблема – в том, что инстанциируемый тип (компонент) в Java VM можно получить только как результат загрузки байткодов (получаемых кодогенерацией – либо статической, с использованием компилятора, либо динамической, с использованием библиотеки кодогенерации, например, Apache Commons BCEL [15]).

Проводя аналогию с компонентными технологиями в других областях, можно отметить, что технология изготовления базовых компонент («элементной базы») может разительно отличаться от технологии, применяемой при их композиции (например, при размещении чипов на плате), что связано со степенью интеграции компонент. Эти технологии могут отличаться не только способами реализации, но даже заложенными в них физическими принципами: они имеют разные цели.

Наша цель заключается в создании компонентной архитектуры и соответствующей компонентной модели для снятия указанного выше противоречия и обеспечения возможности создания составных компонент в динамике без кодогенерации. Отталкиваясь от компонентной модели JavaBeans, мы опускаем средства, связанные с кодогенерацией для Java VM, и привносим новые динамические средства создания и взаимодействия компонент.

3. Типы объектов и их разновидности

В терминах CBSE, компонентом называют тип инстанциируемых объектов, а не отдельный экземпляр. Компонент определяет интерфейс для взаимодействия с его экземплярами и реализацию, обеспечивающую поведение экземпляров. В базовых компонентах эти определения обеспечиваются традиционным программированием. В составных компонентах интерфейс и реализация обеспечиваются композицией составляющих компонент. В отсутствие кодогенерации нам необходимо обобщение понятий типа и компонента, которые в Java VM опирались на понятие класса.

Компоненты JavaBeans могут поддерживать композицию с помощью четырех средств взаимодействия: методов, свойств (properties), источников событий и приемников событий (listeners).

Взаимодействие посредством вызова методов предполагает наличие методов у объекта взаимодействия. Нас взаимодействие с помощью методов не устраивает: мы не генерируем методы в динамике и не связываем разные события с вызовами методов Java-интерфейсов (что в BeanBox обеспечивалось кодогенерацией hookup-классов или, в более современной реализации, механизмом dynamic proxy).

Мы ограничиваемся средствами взаимодействия, которые можно реализовать без кодогенерации. Ими являются свойства (properties) компонент и средства взаимодействия по событиям изменения их значений (property change events). Свойства экземпляров компонент хранят значения определен-

2 Так, в частности, работал инструмент Bean Composer в составе продукта IBM Visual Age for Java.

ных типов, определяющие состояние экземпляра, и ссылки на другие экземпляры в композиции. Значения, присваиваемые свойствам, контролируется на соответствие определенному типу. Такими значениями могут быть, в частности, экземпляры составных компонент, подлежащие контролю типов.

Программируя на объектно-ориентированном языке, мы определяем типы объектов, а не отдельные экземпляры. Определения типов в Java компилируются в класс-файлы байткодов, который загружаются в Java VM загрузчиками классов и представляются там объектами класса Class. В отсутствие байткодов, сгенерированных для составных компонент, нам необходимо обобщение понятия типа, расширяющее понятие класса Java VM. Для этого мы предоставляем надстройку над Java VM, имеющую свое понятие типа объектов. Отталкиваясь от компонентной модели JavaBeans, мы называем надстройку BeanVM. Понятие типа в BeanVM допускает разные реализации: hardcoded-типы, создаваемые из классов, байткоды которых загружены в Java VM, и composed-типы, создаваемые средствами BeanVM, рассматриваемыми далее.

Вне зависимости от способа реализации, типы BeanVM, которые инстанциируются без передачи параметров операции инстанциирования, называются компонентами.

Функциональность программной надстройки BeanVM предоставляется с помощью BeanAPI и набора базовых компонент, с помощью которых создаются составные компоненты BeanVM (и которые, поэтому, можно рассматривать как метакомпоненты).

4. Представление типа

Всякий BeanVM-тип представлен как immutable-экземпляр Java-класса, унаследованного из абстрактного класса Type[3], который определяет структуру и основные свойства всех типов BeanVM. Тип имеет имя, тип интерфейса и тип реализации:

$$type = \{typeName,\ interfaceType,\ implementationType\}. \quad (1)$$

Все BeanVM-типы являются экземплярами типа Type (включая сам этот тип).

Так как единственным средством взаимодействия BeanVM-компонент являются их свойства (с сигнализацией об изменении их значений), интерфейс компонента описывается в терминах набора типов свойств, где тип свойства (immutable-объект, реализуемый экземпляром Java-класса PropertyType) содержит следующую информацию:

$$propertyType = \{propertyName,\ valueType,\ accessType,\ defaultValue\}. \quad (2)$$

Каждое свойство (property) интерфейса имеет имя, фиксированный BeanVM-тип значений и тип доступа, определяющий допустимые операции со свойством; начальные значения свойств определяются только для тех BeanVM-типов, которые являются компонентами. Каждый BeanVM-тип может проверить, принадлежит ли объект множеству объектов, определяемому этим типом, что обеспечивает контроль типа при присваивании значений. Со свойством можно производить операции чтения, записи и связывания, что соответствует регистрации/дерегистрации приемника сигналов об изменениях значения (соответствует bound property в JavaBeans). Для индексируемого свойства (indexed property в терминах JavaBeans) возможны чтение и запись по индексу.

Часть описания BeanVM-типа, относящаяся к реализации, представляет информацию о внутренней реализации типа и существенно различается для hardcoded-типов и composed-типов.

Всякий объект, с которым имеет дело BeanVM, имеет свой BeanVM-тип, и каждый BeanVM-тип может проверить, принадлежит ли объект множеству значений этого типа. Если объект был создан операцией инстанциирования BeanVM-типа, он знает свой тип с момента создания. Если объект присваивается как значение свойству, тип значения которого соответствует Java-классу, то этот класс представлен в BeanVM как hardcoded-тип, который делегирует контроль типа соответствующему классу реализации. Таким образом, BeanVM поддерживает создание и инстанциирование типов компонент и контроль операций со свойствами интерфейсов компонент. Вся прочая функциональность находится вне сферы ответственности BeanVM (обеспечивается поведением собственно компонент).

5. Реализация hardcoded-типов

Для hardcoded-типа implementationType в (1) есть обертка (wrapper) его Java-класса. Получение BeanVM-типа для заданного класса обеспечивается примитивом BeanAPI:

public static Type Type.forClass(Class someJavaClass). (3)

[3] Для краткости изложения мы опускаем реальные имена пакетов реализации.

Поскольку Java-класс идентифицируется именем и своим загрузчиком, мы поддерживаем иерархию загрузчиков BeanVM-типов (объектов класса TypeLoader), повторяющую иерархию загрузчиков классов, информация о которых предоставляется средствами reflection. Типы, сопоставленные классам, хранятся в таблицах загрузчиков. В процессе сопоставления используются механизмы reflection и introspection[3]. При построении интерфейсной части BeanVM-типа для заданного класса используется массив полученных интроспектором объектов класса PropertyDescriptor, откуда извлекается информация для создания объектов класса PropertyType (2). Имя свойства определяется в соответствии с JavaBeans design patterns [12]. Тип значения свойства определяется с помощью примитива (3) по классу, указанному в объекте PropertyDescriptor. Этот же объект определяет возможные операции со свойством для создания объекта accessType.

Таким образом можно построить интерфейсную часть BeanVM-типа для любого Java-класса, но нас интересуют только классы, являющиеся нашими hardcoded-компонентами или типами значений свойств у таких компонент.

Все наши hardcoded-компоненты реализуются наследованием (прямо или косвенно) из базового класса Bean, который предоставляет BeanAPI для реализации компонента как BeanVM-типа. Экземпляр hardcoded-компонента имеет внутри себя реализацию своих свойств в терминах BeanVM, то есть является оберткой экземпляра соответствующего BeanVM-типа. Реализация операций со свойствами нашего hardcoded-компонента обеспечивается делегированием к внутреннему представлению экземпляра. Ниже приведен фрагмент кода класса Bean с соответствующим BeanAPI.

```
public class Bean extends BeanVMObject {
    final Instance thisInstance;

    public Bean()   { thisInstance = Type.implementBean(this); }
    Bean(Type type) { thisInstance = type.createInstance(this); }
    // …
    protected final void initPropertyValue(String propertyName,
    Object initValue) {…}
    public final Object getPropertyValue(String propertyName) {…}
    public final void setPropertyValue(String propertyName,
    Object newValue) {…}
    // …
}
```

В указанном фрагменте кода, BeanVMObject – абстрактный класс, требующий, чтобы все объекты BeanVM предоставляли свой тип. Класс Bean имеет два конструктора: общедоступный без параметров и скрытый за пределами пакета, принимающий тип. Первый используется при реализации hardcoded-компонент, второй – при реализации composed-компонент.

Первый конструктор выполняется при инстанциировании любого hardcoded-компонента, передавая новый экземпляр компонента (this) методу-фабрике внутреннего представления экземпляра, которое сохраняется в поле thisInstance. Метод Type.implementBean() получает BeanVM-тип по классу инстанциируемого объекта с помощью примитива (3), после чего этот BeanVM-тип обеспечивает создание внутреннего объекта (реализация которого наследована из класса Instance).

Любой Java-класс, наследованный из Bean, при первой попытке инстанциирования обеспечивает наличие своего BeanVM-типа, который, в свою очередь, создает внутреннюю реализацию всех своих экземпляров. Для hardcoded-типов внутренняя реализация экземпляра содержит (кроме указателя на тип) реализации переменных (mutable) свойств экземпляра. Значения постоянных (immutable) свойств хранятся в типе (как defaultValue в propertyType (2)). Свойство считается переменным, если его тип доступа (accessType) разрешает запись или связывание.

После выполнения конструктора суперкласса Bean конструктор конкретного hardcoded-компонента может воспользоваться методом класса Bean initPropertyValue() с указанием имени свойства и значения инициализации. Этот метод срабатывает только один раз при создании BeanVM-типа для данного класса, что происходит при самом первом инстанциировании и специально для того, чтобы собрать и сохранить инициализирующие значения. Если BeanVM-тип уже создан, вызовы метода initPropertyValue() являются пустыми операциями, а сами свойства – уже инициализированными при создании экземпляра реализации.

Собственно методы доступа к свойствам контролируют право доступа и тип присваиваемого значения; при нарушении выдается соответствующая ошибка (RuntimeException). Для ускорения доступа класс Bean предоставляет парные операции с ука-

3 java.beans.Introspector может анализировать не тольео JavaBeans компоненты, оно и любые Java-классы

занием не имени свойства, а его индекса (номера элемента в описании интерфейса), который можно получить у типа по имени.

С точки зрения JavaVM все объекты BeanVM декларируются как объекты класса Object, в связи с чем требуется явное приведение к типу возвращаемого значения при использовании метода getPropertyValue().

Все наши hardcoded-компоненты являются JavaBeans-компонентами. Для инстансов сторонних JavaBeans-компонент автоматически предоставляется наш компонент-адаптер. Мы, тем самым, используем преимущества компонентной модели JavaBeans. Рассмотрим, как предлагаемая архитектура способствует устранению недостатков, обсуждавшихся в разделе 2.

6. Составные компоненты

Составной (composed) компонент, как любой BeanVM-тип, имеет имя, интерфейс и скрытую реализацию (1), которые определяются с помощью композиции.

Интерфейс composed-компонент определяется также, как для hardcoded-компонент, и представляет собой массив типов свойств. Описание реализации состоит из описаний составляющих типов (composing types).

Внешнее взаимодействие с экземпляром составного типа осуществляется теми же средствами BeanAPI: реализация любого экземпляра компонента обеспечивается классом, унаследованным из класса Bean (с той разницей, что составной тип непосредственно передается через параметр скрытого конструктора инстанса); доступ к свойствам экземпляра компонента осуществляется с помощью BeanAPI вне зависимости от способа реализации его типа.

При создании экземпляра составного типа обеспечивается не только внутренняя реализация изменяемых свойств его интерфейса, но и инстанциирование его составляющих типов реализации, которые являются контекстно-зависимыми уточнениями составляющих компонент, использованных при композиции.

Интерфейс типа определяет набор свойств в его экземплярах. Внутренняя реализация экземпляра составного типа должна взаимодействовать с внешним окружением через свойства интерфейса. Чтобы обеспечить связь интерфейса и внутренней реализации экземпляра составного типа требуется связать некоторые свойства внешнего интерфейса с некоторыми (однотипными) свойствами из интерфейсов экземпляров составляющих компонент. Такая связь обеспечивается с помощью разделения объектов, ализующих эти свойства. Реализация типов свойств категоризируется на подклассы, соответствующие способам обращений за значением: мы различаем PropertyType. Immutable, PropertyType.Mutable, PropertyType. Bound и PropertyType.External, последний подкласс обеспечивает использование указанного в нем разделяемого свойства из интерфейса охватывающего экземпляра составного компонента.

До сих пор мы обсуждали контекстно-независимые компоненты, которые инстанциируются без передачи параметров в операцию создания экземпляра (что соответствует определению компонента). В Java VM нет удобного способа узнать контекст вызова метода или конструктора. Мы, однако, выполняем инстанциирования в рамках BeanVM, с помощью операции BeanVM-типа createInstance(), которая может вызывать конструкторы Java-классов внутри своей реализации, а конструктор экземпляра (в классе Bean) делегирует создание реализации экземпляра фабрике, также реализованной в рамках BeanVM. Это позволяет использовать стек вызовов, реализованный в BeanVM, для передачи через него контекстной информации, нужной для внутренней реализации экземпляров составляющих типов.

Рассмотрим, как составные и составляющие их типы создаются в динамике.

7. Преобразование прототипа в тип

Динамическое создание составного immutable-объекта, каким является описание BeanVM-типа, осуществляется с использованием mutable – объекта, в котором накапливается необходимая информация (builder design pattern [16]). Мы называем такой составной объект прототипом и используем его для создания BeanVM-типа. Объект-прототип является экземпляром hardcoded-компонента Prototype со свойствами «name», «interfacePrototype» и «implementationPrototype», значения которых задаются строкой и экземплярами специальных hardcoded-компонент InterfacePrototype и ImplementationPrototype, соответственно.

Экземпляр компонента InterfacePrototype содержит набор объектов PropertyPrototype, которые используются как интерфейсные свойства самого объекта-прототипа и могут предоставить информацию для создания соответствующих объектов PropertyType.

Экземпляр компонента ImplementationPrototype указывает на набор экземпляров компонент, составляющих реализацию составного прототипа. Составной прототип строится с помощью определения графа ссылок и графа событий. Граф ссылок определяется путем использования одного экземпляра компонента в качестве значения свойства (или элемента интексируемого свойства) у другого (или других, которые в этом случае разделяют это общее значение). Граф событий определяет связывания однотипных свойств разных экземпляров путями передачи сигналов об изменении значений.

Мы определяем интерфейс составного прототипа в терминах свойств, которые сами прототипируются с использованием типизированных переменных – объектов, имеющих свойство "value" заданного BeanVM-типа. При наличии BeanVM-типа, тип объектов-переменных со значением данного типа (или индексируемым массивом элементов этого типа) создается автоматически (аналогично созданию классов для массивов в Java VM). Такие типы мы называем синтетическими. Их экземпляры, в сочетании с соответствующими объектами типа AccessPrototype используются в качестве прототипов реализации свойств (объектов PropertyPrototype) задаваемых в интерфейсе.

Прототипы свойств интерфейса, реализованные с помощью типизированных переменных синтетических типов, могут разделяться экземплярами составляющих компонент из реализации составного прототипа. Когда компонент инстанциируется внутри контейнера-прототипа, он получает свою внутреннюю реализацию (от фабрики реализации экземпляра), которая специально предназначена для использования в составном прототипе (в отличие от контейнера реализации уже определенного типа). Эту реализацию мы называем экземпляром составляющего прототипа. В отличие от экземпляра составляющего типа, где в реализации интерфейса хранятся сами значения переменных свойств, в экземпляре составляющего прототипа реализация интерфейса содержит прототипы свойств, которые могут заимствоваться из интерфейса охватывающего составного прототипа (а также настраиваться путем уменьшения прав доступа для данного контекста использования). Фабрика различает контекст инстанциирования с помощью стека операций BeanVM, упомянутого выше.

Составной объект-прототип является работоспособным и настраиваемым. Его композиция может осуществляться манипуляциями, аналогичными тем, что поддерживаются в инструменте BeanBox для JavaBeans-компонент. Когда необходимая композиция составного прототипа и настройки составляющих прототипов в нем выполнены, составной прототип может быть превращен в составной тип с помощью операции:

public Type Type.fromPrototype(PrototypesomePrototype). (4)

При выполнении этой операции все экземпляры компонент-прототипов преобразуются в соответствующие элементы описания создаваемого составного типа: прототипы свойств преобразуются в типы свойств, составляющие прототипы – в составляющие типы (с соответствующими типами свойств, в том числе – разделяемых с внешним интерфейсом), и т.д. Новый тип представляет собой динамически собранный составной компонент, который может инстанциироваться и использоваться как элемент последующих композиций.

Созданный составной компонент может быть сериализован и десериализован с применением определенного формата. Для чтения такого формата загрузчик составного типа может использовать указанную процедуру построения составного прототипа и преобразования его в тип операцией (4).

8. Заключение

Предложенная компонентная архитектура расширяет возможности компонентной модели JavaBeans в направлении динамического создания определяемых типов компонент. Расширение основано на взаимосвязи понятий типа и прототипа, что соответствует совместному использованию двух парадигм объектного программирования – class-based и prototype-based. Первая обеспечивает более эффективную реализацию экземпляров (в смысле ресурсов памяти и времения выполнения), вторая – гибкость настройки для последующего динамического перехода к первой.

Непосредственным применением является объектно-ориентированное расширение средств для реализации определяемых типов в языках 3D-моделирования [10; 11]. Поддержка таких средств с использованием стандартной компонентной модели JavaBeans затруднена присущими ей недостатками.

Как и компонентная модель JavaBeans, предложенная компонентная модель нуждается в инструментарии, который демонстрирует и использует ее возможности. Реализация такого инструментария является направлением дальнейшей работы. ■

Литература

1. Component-based software engineering: Putting the pieces together / G.T.Heinemann, W.T.Councill, Editors. Boston, MA: Addison-Wesley, 2001.
2. McIlroy M.D. Mass produced software components. Software engineering // Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968.
3. Open Services Gateway initiative – OSGi Alliance. [Электронный ресурс]: http://www.osgi.org/Main/HomePage (дата обращения 30.01.2014).
4. Eddon G., Eddon H. Inside COM+ base services. Redmond, WA: Microsoft Press, 2000.
5. Kung-Kiu Lau, Zheng Wang. A taxonomy of software component models // Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications. Porto, Portugal, 30 August - 3 September 2005. P. 88-95.
6. Crnković I., Sentilles S., Vulgarakis A., Chaudron M. A classification framework for component models // IEEE Transactions on Software Engineering. 2011. Vol. 37, No. 5. P. 593-615.
7. The Standards ANSI/IEEE 1471-2000 and ISO/IEC 42010:2007.
8. The TIOBE Programming Community index. [Электронный ресурс]: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (дата обращения 30.01.2014).
9. Гринкруг Е.М. Использование JavaBeans-компонент в 3D-моделировании // Бизнес-информатика. 2010. № 3 (13). С. 47-56.
10. ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2004 – Virtual Reality Modeling Language (VRML). [Электронный ресурс]: http://www.web3d.org/x3d/specifications/vrml/ (дата обращения 30.01.2014).
11. ISO/IEC 19775-1:2008. X3D – Extensible 3D. [Электронный ресурс]: http://www.web3d.org/x3d/specifications/ (дата обращения 30.01.2014).
12. JavaBeans API specification. [Электронный ресурс]: http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html (дата обращения 30.01.2014).
13. Bean Development Kit 1.1 with BeanBox. [Электронный ресурс]: http://code.google.com/p/jbaindi/downloads/detail?name=BDK%20%28Bean%20Development%20Kit%29.zip&can=2&q (дата обращения 30.01.2014).
14. Costa Seco J. Component J in a nutshell. 2002. [Электронный ресурс]: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.9905&rep=rep1&type=pdf (дата обращения 30.01.2014).
15. Apache Commons. The Byte Code Engineering Library. [Электронный ресурс]: http://commons.apache.org/proper/commons-bcel (дата обращения 30.01.2014).
16. Bloch J. Effective Java: Second Edition. Addison-Wesley, 2008.

COMPOSED PROTOTYPES AND TYPES IN A COMPONENT-ORIENTED ARCHITECTURE[1]

Efim GRINKRUG,

Associate Professor, Software Management Department, School of Software Engineering, Faculty of Business Informatics, National Research University Higher School of Economics

Address: 20, Myasnitskaya str., Moscow, 101000, Russian Federation
E-mail: egrinkrug@hse.ru

The paper presents a component-oriented software architecture implemented for the Java-platform and to support event-driven calculations. The component model used extends standard JavaBeans component model's dynamic capabilities and supports composite components composition at runtime without code generation. It has been shown that in the standard JavaBeans component model the dynamic component composition is associated with a shift from the object-oriented programming used to develop basic components to the prototype-based programming style when dealing with composite components. A generalized concept of object type has been proposed that enables to operate uniformly both with the basic components and with the components composed dynamically. The implementation of the object type generalization proposed relies on the additional virtualization level that provides an execution environment for software components defined in accordance with the component model described. Different implementation details for object types the environment operates with are considered along with operations it provides. The method to create composite components dynamically involving transformation of the composite prototype into the instantiable type of composite objects has been proposed. Composite components instantiations enable more efficient (in terms of time and memory space) applications functioning in contrast to composite prototypes cloning. The dynamically defined composite components can be used in object-oriented manner along with basic components both to implement applications functionality and as elements of composite prototypes compositions to be transformed into new composite components. The proposed component based architecture can be used to support user defined types implementations in declarative modeling languages.

Key words: software component, component model, type, prototype, interface, implementation.

References

1. Heinemann G.T., Councill W.T., Eds. (2001) *Component-based software engineering: Putting the pieces together.* Boston, MA: Addison-Wesley.
2. McIlroy M.D. (1968) *Mass produced software components. Software engineering.* Report of a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968.
3. *Open Services Gateway initiative – OSGi Alliance*. Available at: http://www.osgi.org/Main/HomePage (accessed 30.01.2014).
4. Eddon G., Eddon H. (2000) *Inside COM+ base services.* Redmond, WA: Microsoft Press.
5. Kung-Kiu Lau, Zheng Wang (2005) A taxonomy of software component models. Proceedings of the *31st EUROMICRO Conference on Software Engineering and Advanced Applications.* Porto, Portugal, 30 August - 3 September 2005, pp. 88-95.
6. Crnković I., Sentilles S., Vulgarakis A., Chaudron M. (2011) A classification framework for component models. I*EEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593-615.
7. The Standards *ANSI/IEEE 1471-2000 and ISO/IEC 42010:2007.*
8. The TIOBE Programming Community index. Available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (accessed 30.01.2014).
9. Grinkrug E. (2010) Ispolzovanie JavaBeans-component v 3D-modelirovanii [Using JavaBeans components in 3D-modeling]. *Business Informatics*, no. 3 (13), pp. 47-56.
10. *ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2004 – Virtual Reality Modeling Language (VRML)*. Available at: http://www.web3d.org/x3d/specifications/vrml/ (accessed 30.01.2014).
11. I*SO/IEC 19775-1:2008. X3D* – Extensible 3D. Available at: http://www.web3d.org/x3d/specifications/ (accessed 30.01.2014).
12. *JavaBeans API specification*. Available at: http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html (accessed 30.01.2014).
13. *Bean Development Kit 1.1 with BeanBox*. Available at: http://code.google.com/p/jbaindi/downloads/detail?name=BDK%20%28Bean%20Development%20Kit%29.zip&can=2&q (accessed 30.01.2014).
14. Costa Seco J. (2002) *Component J in a nutshell.* Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.9905&rep=rep1&type=pdf (accessed 30.01.2014).
15. *Apache Commons. The Byte Code Engineering Library*. Available at: http://commons.apache.org/proper/commons-bcel (accessed 30.01.2014).
16. Bloch J. (2008) *Effective Java: Second Edition*. Addison-Wesley.

[1] This work is an output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE).