DOI: 10.15514/ISPRAS-2025-37(6)-2

Разновидность JavaBeans-компонент: композиция типов из агрегации инстансов

Е.М. Гринкруг, ORCID: 0000-0001-6740-0541 <grinkrug@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Представлен подход к реализации компонент JavaBeans, который обеспечивает создание определяемых пользователем компонент без их компиляции, путем манипуляций с существующими компонентами. Компонентная модель JavaBeans содержит принципиальные ограничения. Компоненты в ней являются классами, определенными для манипулирования их инстансами в предназначенной для этого среде манипулирования. Цель манипуляций – достичь требуемых состояний инстансов компонент и поведения их агрегации в целом; готовая агрегация может быть сериализована и десериализована позже в аналогичной среде. Тут скрыто противоречие: начиная с использования набора инстанциируемых классов, мы в итоге приходим к копированию агрегации из их инстансов. Чтобы использовать определяемую пользователем агрегацию для получения нового составного компонента, требуется сгенерировать его класс, подменяющий парадигму программирования (с инстанциирования на копирование). Предложено расширение компонентной модели JavaBeans, позволяющее динамически создавать определяемые пользователем компоненты без их кодогенерации и копирования агрегации инстансов.

Ключевые слова: компонентная модель; компонент; тип; прототип; инстанс; свойства; интерфейс; реализация типа.

Для цитирования: Гринкруг Е.М. Разновидность JavaBeans-компонент: композиция типов из агрегации инстансов. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 21–42. DOI: 10.15514/ISPRAS–2025–37(6)–2.

Yet Another Kind of JavaBeans: Composed Types from Aggregated Instances

E.M. Grinkrug, ORCID: 0000-0001-6740-0541 <grinkrug@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. The paper presents an approach to the JavaBeans-components implementations so that they provide support for dynamic components composition – user defined components creation without compiling them, by manipulations with pre-existing components instead. JavaBeans component model, presented at the beginning of Java technology, has its' limitation that seems to be significant. The JavaBean component, by definition, is a serializable Java class with public no-arguments constructor; additionally, JavaBeans design patterns serve to use JavaBeans-components instances in a specific manipulating environment. The goal of the manipulations is to achieve the required states of the instances and their aggregation behavior altogether, when the aggregation can be serialized and deserialized later in similar environment. There is a hidden contradiction: starting from a predefined set of JavaBeans components – classes for class-based object-oriented environment, we end up with a prototype-based style for instances aggregation usage. To use user-defined aggregation as a new composed component we must change a programming paradigm and generate code in static-like way of component creation. We propose an evolution of JavaBeans component model that enables user defined composed components creation dynamically without aggregation cloning.

Keywords: component model; component; type; prototype; instance; properties; interface; type implementation.

For citation: Grinkrug E.M. Yet Another Kind of JavaBeans: Composed Types from Aggregated Instances. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 21-42 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-2.

1. Введение

Компонентно-Ориентированное Программирование (КОП), как понятие, стало официально использоваться с 1968 года, после конференции [1], где обсуждались основные отличия разработки компьютерных программ от разработки аппаратных средств самих компьютеров. Основным был вопрос: почему аппаратура компьютера создается быстрее, чем программное обеспечение для него? Ответом было признание того, что компьютеры проектируются из готовых компонент, чего нельзя сказать о программах (и до сих пор).

С тех пор было предложено множество программных технологий и компонентных моделей для внедрения КОП в практику разработки программного обеспечения. Компонентная модель – это совокупность правил, определяющих, что является компонентом [2] и как такие компоненты используются совместно [3].

В любом компьютере программа представляет собой набор экземпляров взаимосвязанных компонент, начиная с уровня машинного языка, где программы – это наборы машинных инструкций разных типов (определенных системой команд). В этом смысле любая программа является композицией экземпляров компонент, и не-компонентного программирования не бывает. Проблема заключается в том, как именно определяются сами компоненты и композиции с их использованием. Иными словами – в том, какая компонентная модель используется.

Мы ограничим обсуждение программированием на Java-платформе, для которой была предложена компонентная модель JavaBeans [4], лежащая в основе многих современных технологий. Фирма Sun Microsystems изначально указывала в документации: «JavaBeans component model is the only component model for the Java-machine», хотя впоследствии появились и другие компонентные модели. Основные соображения могут быть полезны и для других, сходных с Java-платформой, средств программирования.

Предпосылки работы возникли из опыта реализации средствами Java-программирования подмножества языка моделирования виртуальной реальности VRML-97 [5]. Реализация средств моделирования средствами объектно-ориентированного программирования (для моделирования и задуманного) является естественной и наглядной: результат моделирования непосредственно отображается в 3D-графике.

Все базовые компоненты стандарта VRML были реализованы как стандартные JavaBeans-компоненты; другие JavaBeans-компоненты являлись элементами графического интерфейса пользователя для отображения трехмерной сцены. Это позволило выполнять моделирование и визуализацию в стандартных инструментах манипулирования такими компонентами и наблюдать достоинства и недостатки компонентной модели JavaBean.

Развитию возможностей и преодолению недостатков этой компонентной модели посвящена данная работа. Она не претендует на полноту изложения, но посвящена соображениям, на которых основана реализация расширения компонентной модели.

В разделе 2 дан обзор исходной компонентной модели, ее использования и соображения о направлении развития. В разделе 3 изложено основное содержание работы - принципы реализации базовых компонент и их использование при прототипировании составных компонент. Перечень основных положений работы и их места среди прочих подходов приведены в разделах 4 и 5; заключают работу перспективные цели проекта.

2. Компонентная модель и ее развитие

Для дальнейшего обсуждения нам понадобятся сведения об исходной компонентной модели JavaBeans и используемой терминологии.

2.1 Компонентная модель JavaBeans

2.1.1 Определение и важные особенности

Компонентная модель JavaBeans дает весьма общее определение своих компонент [4]: JavaBeans–компонент – это общедоступный (public) класс (возможно со вспомогательными классами и ресурсами), который:

- имеет общедоступный конструктор без аргументов, и
- реализует маркерный интерфейс java.io.Serializable.

Возможность создавать объекты такого класса без предоставления дополнительной информации конструктору позволяет получать «одинаковые» *инстансы* компонента «бесконтекстным» образом (без зависимости от контекста их создания) и определяет *универсальный способ инстанциирования* таких компонент.

Специализация этих инстансов обеспечивается вызовами их общедоступных методов, среди которых особо выделяются *методы доступа к значениям свойств* (*property accessors*), хотя их наличие в классе по определению JavaBeans-компонента не является обязательным. Эти методы (*setters, getters*) позволяют абстрагироваться от наличия соответствующих полей класса для хранения значений свойств вообще, а при их наличии – варьировать способы доступа к значениям, чего сами поля выразить не позволяют.

Требование реализации интерфейса java.io.Serializable исторически связано с сохранением и восстановлением инстансов компонент с помощью встроенных в JVM базовых средств сериализации/десериализации; оно не является обременительным при наличии иных средств.

Взаимодействие с инстансами JavaBeans-компонент возможно путем вызовов их (public) методов и/или с использованием реакций на события, которые вырабатываются инстансами компонент (в соответствии с событийной моделью Java-платформы). В основе реализации лежит механизм интроспекции их классов, опирающийся на средства рефлекции.

Возможной реакцией на событие является вызов метода некоторого инстанса компонента с передачей ему информации из события. Это называют связыванием события объекта-источника с вызовом метода у объекта-приемника. Для интерфейсов общего вида при этом изначально требовалась генерация кода классов объектов-посредников, которые подписывались на получение событий нужного типа и делегировали реакцию к указанному методу; с появлением механизма Dynamic Proxy (в JDK1.3) реализация упростилась (давая наглядный пример того, как упрощается использование компонент при добавлении динамических средств в Java-платформу).

Для событий, сообщающих об изменении значения свойства (*property*), где реакция на событие сводится к передаче значения свойства инстанса-источника в заданное свойство инстанса-получателя, предоставляются готовые объекты-посредники; а само свойство источника называется «связываемым» (*bound property*).

2.1.2 Использование JavaBeans-компонент

Сценарии использования JavaBeans-компонент демонстрировал Bean Development Kit (BDK), прилагавшийся к спецификации [4] с интерактивным инструментом BeanBox. Они таковы:

- в BeanBox поставляются архивы с JavaBeans-компонентами в виде байткодов их классов; BeanBox их загружает и показывает список доступных в нем компонент;
- BeanBox предоставляет инстанс (одноименного) контейнера для агрегирования инстансов загруженных компонент (классов); инстансы создаются бесконтекстным образом (default-конструкторами);
- созданные инстансы отображаются в GUI инстанса контейнера BeanBox;
- выполняется настройка значений свойств созданных инстансов; интерактивная настройка предполагает наличие редакторов значений свойств; сам BeanBox располагает лишь базовым их набором, но компоненты могут их «приносить с собой»; при наличии нужных редакторов могли бы интерактивно выполняться «внедрения зависимостей» (dependency injections) инстансов друг от друга, но сам BeanBox это не поддерживал;
- выполняется настройка графов распространения событий, которые связывают события от инстансов-источников с методами реакции на события инстансами-приемниками; такие графы определяют совместное управляемое событиями (event-driven) поведение создаваемой агрегации инстансов.

Приложение BeanBox предоставляло средства для сохранения и загрузки контента инстанса контейнера с помощью стандартной сериализации/десериализации, для чего требование реализации интерфейса java.io.Serializable и входило в определение JavaBeans-компонент. Приложение BeanBox демонстрировало также вариант программной генерации и компиляции класса контейнера, реализовывавшего предопределенный интерфейс (на примере устаревшего класса java.applet.Applet), в инстанс которого можно было загрузить (десериализовать) созданную агрегацию инстансов компонент.

Компонентная модель JavaBeans используется как в модулях JDK, так и в популярных современных библиотеках. В современных JDK появились средства для более эффективной реализации этой компонентной модели, однако, имеются важные архитектурные причины и для совершенствования её самой.

Все компоненты JavaBeans являются Java-классами, загруженными загрузчиками классов и представленными на этапе исполнения (at runtime) объектами типа java.lang.Class. Эти объекты-классы, способные создавать инстансы бесконтекстным образом, сами (по определению) бесконтекстным образом создаваться не могут (их класс java.lang.Class не

является JavaBeans-компонентом). Мы можем выполнить агрегацию инстансов компонент внутри инстанса компонента-контейнера (предопределенного типа), но мы не можем создать новый тип такого контейнера в динамике, без генерации его байткода и без помощи загрузчика класса, превращающего этот байткод в объект-класс. Мы можем лишь копировать содержимое из одного инстанса компонента-контейнера (имеющегося класса) в другой, подменяя, в сущности, инстанциирование (отсутствующих без их кодогенерации) классов *составных компонент* копированием (клонированием) содержимого инстансов имеющихся классов-контейнеров (включая копирование через сериализацию/десериализацию).

Имея изначально набор базовых компонент в виде классов, созданных на основе парадигмы ООП (*class-based object-oriented programming*), мы приходим при их использовании к клонированию объектов-прототипов. Вынужденно обходя отсутствие возможности создать составной компонент в динамике и заменяя его инстанциирование клонированием инстансов имеющихся компонент-контейнеров, *мы (неявно) подменяем парадигму программирования*.

Кроме того, поскольку JavaBeans-компонент – это *произвольный* (сериализуемый) *класс, который инстанциируется бесконтекстно*, его нельзя адаптировать под конкретный контекст использования: инстансы компонента, используемые в разных контекстах, создаются «одинаковыми»; мы при их инстанциировании не используем информацию об их применении в конкретном контексте реализации агрегата из них, а это может быть весьма полезно.

2.2 Технология использования компонент

Технологии использования готовых компонент при создании изделий из них обладают рядом характерных черт, которые полезно сопоставить с использованием программных компонент. При этом важно явно отличать типы используемых объектов от самих этих объектов – инстансов этих типов (к сожалению, смешение понятий классов и создаваемых ими объектов-инстансов встречается даже в официальной документации).

Конструкторы (как наборы заготовок для сборки изделий, а не программные средства инициализации объектов) обычно содержат некоторое количество деталей различных типов; эти детали могут соединяться друг с другом заранее определенным способом в заранее заготовленных местах. При сборке конкретного изделия не все имеющиеся возможности соединений деталей используются: остаются избыточные места для соединений, что «ухудшает» готовое изделие из-за оказавшейся лишней универсальности его деталей.

Разной может быть и технология изготовления самих деталей. При литье используется «гибкий» материал (например, пластилин), по гибкому образцу делается твердая форма (например, гипсовая), используемая для получения «монолитных» изделий.

Сопоставим эти соображения с использованием программных компонент - классов. Традиционно, они «описываются» на объектно-ориентированном языке. Их автор «держит в голове» их состав, понимая, как с помощью языка выразить описание создаваемого типа (класса) из его элементов так, чтобы в будущем (после компиляции, генерации кода и его загрузки) инстансы этого класса (экземпляры создаваемых им в динамике объектов), давали и/или делали то, что он задумал. Это поддерживают все традиционные технологии объектно-ориентированного (class-based) программирования, и в этом их сложность.

Аналогично, разработчик аппаратуры (например, радиоприемника), если он достаточно опытен, может сразу нарисовать принципиальную схему новой модели (радиоприемника), где укажет, какие именно детали и как именно должны быть собраны и соединены между собой (например, паяльником) на монтажной плате. При этом:

- сама эта принципиальная схема работать (звучать) не будет, но на заводе по этой схеме, если она правильная, обеспечат производство работоспособных изделий;

- иногда даже опытному инженеру надо попробовать разные варианты, сделав опытные экземпляры – прототипы, прежде чем остановиться на искомой конструкции: ему надо убедиться в ее качестве.

Если *работающий прототип* есть, дальше – дело техники: по нему можно нарисовать схему. Некоторые детали прототипа могут настраиваться (например, конденсаторы переменной емкости), а в схеме можно указать уже определенное на этапе настройки прототипа значение (например, оптимальное значение емкости).

Наконец, еще одно важное замечание: технология изготовления прототипа из компонент, как и технология тиражирования инстансов компонент (созданных по схеме, проверенной прототипом) отличаются от технологии изготовления самих компонент для прототипа (свинчивание готовых деталей, монтаж изделия из электронных компонент, и т.п.).

С этой точки зрения, компиляция кажется исключением: компилятор может «одинаково успешно» производить компоненты с любой сложностью их внутренней организации, но:

- он обычно делает это «в статике», а не «в динамике», и
- для работы ему необходима исходная информация от автора кода: *чтобы компилировать, надо иметь, что компилировать*.

Известны программные инструменты (часто это GUI-построители), которые в процессе конструирования прототипа сохраняют («под капотом») некоторый код его создания, который потом используется при компиляции конечного результата. При этом, однако, конструируемый прототип не обладает полной функциональностью законченного изделия и не демонстрирует непосредственно желаемого в итоге поведения; это получается на этапе исполнения только в результате последующей компиляции, загрузки и т.д.

Мы хотим иметь возможность наблюдать и настраивать *полнофункциональный* прототип, из которого автоматически извлекается тип, будущие инстансы которого повторяют функциональность, заложенную в прототипе. При такой настройке прототипа мы хотим уметь отказываться от избыточной для конкретного применения универсальности используемых компонент.

3. Принципы реализации и композиции компонент

Нас интересует компонентная технология, позволяющая в динамике производить новые типы – компоненты. Такая технология должна быть «логически замкнутой объектно-ориентированной технологией», которая поддерживает создание *составных типов* (*составных компонент*), без использования компиляции (кодогенерации) и не подменяет инстанциирование типов клонированием их инстансов.

Это пожелание требует расширения возможностей компонентной модели JavaBeans, где типы объектов (представленные классами) появляются на этапе исполнения только в результате генерации байткодов и их загрузки. Если компонент – это тип создаваемых им инстансов, надо уметь создавать новые составные типы, оставаясь в рамках исходной объектно-ориентированной парадигмы: так, чтобы эти составные типы создавали свои (составные) инстансы, а не клонировали объекты-прототипы (что, в частности, способствует повторному использованию/разделению общих составляющих объектов).

Нас интересуют принципы композиции компонент, реализуемые *имеющимися средствами* JDK (возможно, они будут расширены с внедрением проекта Babylon [6]). Мы опираемся на уже имеющиеся средства работы с JavaBeans-компонентами, и *эволюционируем* эту компонентную модель, оставаясь в ее рамках (реализуя наши компоненты как *специфические* JavaBeans-компоненты).

Основной принцип «перекликается» с известной проблемой «курица или яйцо»: что первично – объект или порождающий его тип? Мы хотим обеспечить динамическую

эволюцию типов, предполагая возможность их создания из уже имеющихся типов путем агрегирования их инстансов в инстансе специального контейнера-прототипа, из которого автоматически извлекается необходимая «генетическая информация», характеризующая *новый тип – как композицию, созданную из типов, инстансы которых агрегировались в прототипе.*

В JVM первичным является класс: все, что можно выразить на исходном языке, представляет собой описание, которое на этапе исполнения представлено классом. Мы надстраиваем систему типов в рассматриваемой компонентной модели так, чтобы типы появлялись не только в результате работы загрузчика байткодов классов, но и в результате *преобразования прототипа в тип*. Типы остаются первичными: объекты-инстансы могут создаваться только типами, и каждый объект позволяет узнать его тип (включая сами типы). В первую очередь нас интересуют *типы-компоненты – типы, порождающие свои инстансы бесконтекстным образом.*

Мы предоставляем библиотеку, расширяющую компонентную модель JavaBeans новыми JavaBeans-компонентами, поддерживающими обобщение понятия типа над классами JVM.

Будем называть эту надстройку BeanVM (реализуя ее с помощью JavaBeans-компонент). BeanVM допускает две разновидности типов объектов: типы, реализованные классами JVM (или *типы-классы*), и типы, полученные в результате превращения объекта-прототипа (как агрегата инстансов компонент внутри инстанса специального контейнера-прототипа) в тип (как композицию из типов элементов, агрегированных в инстансе прототипа).

При агрегировании инстансов компонент в инстансе контейнера прототипа можно настраивать их состав, свойства и взаимодействия. Полученный из прототипа *составной тип* (*composed type*) является законченной композицией составляющих его типов. Каждый *составляющий тип* (*composing type*) полученной композиции является типом, описывающим («уточняющим») использование некоторого исходного типа (компонента) в контексте реализации составного типа (компонента). Аналогично типам (классам) JVM, все типы BeanVM являются immutable-объектами (инстансы могут ими не быть).

Соотношение компонент JavaBeans и компонент BeanVM между собой показано на рис. 1. Компонент в BeanVM – это тип BeanVM, инстанциируемый бесконтекстно. Для JavaBeans-компонент, не являющихся BeanVM-компонентами предоставляется адаптер, позволяющий взаимодействовать с ними средствами взаимодействия с инстансами BeanVM-компонент. Показанные на рис. 1 composed-компоненты не имеют собственного класса реализации и определяются пользователем (с помощью BeanVM API).

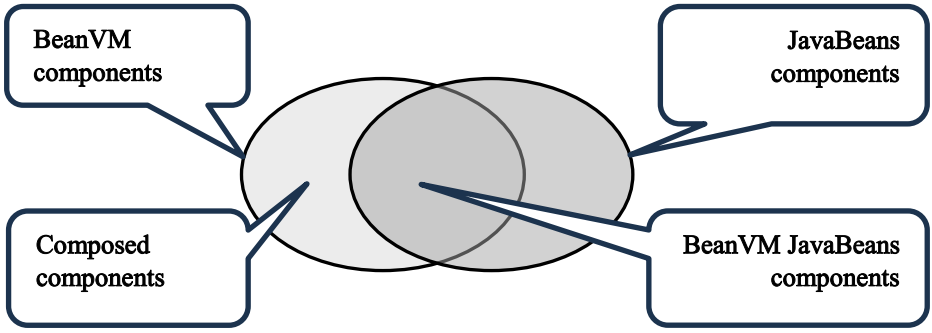

Рис. 1. Соотношение JavaBeans-компонент и BeanVM-компонент.
Fig. 1. JavaBean-components and BeanVM-components interrelation.

3.1 Типы и инстансы BeanVM

Все типы BeanVM доступны на этапе исполнения с помощью операции BeanVM API Type.forName(<имя_типа>), где имена типов отслеживаются экземплярами загрузчиков типов

BeanVM аналогично тому, как это делается в JVM при получении класса по имени в операции Class.forName(<имя_класса>). Операция Type.forName() сперва ищет одноименный java-класс с помощью загрузчика класса и операции Type.forClass(<имя_класса>). При первом выполнении этой операции для данного класса происходит создание и регистрация в загрузчике типов *типа BeanVM для данного класса* (т.е. соответствующего *типа-класса*), который и выдается в качестве результата операции; последующие поиски типа BeanVM по этому классу JVM будут выдавать уже зарегистрированный экземпляр типа-класса. Проблема расширенной типизации решается внедрением дополнительной косвенности: тип-класс BeanVM является «объектом-оберткой» java-класса его реализации.

При отсутствии класса с указанным именем (при отсутствии возможности предоставить соответствующий тип-класс) загрузчик типов BeanVM предпринимает попытку создания и регистрации составного типа из файла-ресурса с соответствующим именем, хранящего его описание. Регистрируемый при этом тип BeanVM является результатом создания инстанса прототипа с информацией, предоставленной парсером его описания из файла-ресурса, и последующего преобразования этого инстанса прототипа в составной тип.

Все внешние взаимодействия с инстансами BeanVM-компонент происходят в терминах операций с их свойствами (properties). Но эти свойства в качестве своих значений содержат объекты, имеющие тип BeanVM (property value type), а не класс JVM (property value class). Это позволяет распространить контроль типов значений свойств на все типы BeanVM.

Для внешнего взаимодействия с инстансами компонент и их свойствами предоставляются следующие операции, сходные с методами доступа к свойствам для JavaBeans-компонент (*внешний* Bean API):

```
public final void setPropertyValue(String propertyName, Object newValue);
public final void setPropertyValue(String propertyName, int elementIndex, Object newValue);
public final Object getPropertyValue(String propertyName);
public final Object getPropertyValue(String propertyName, int elementIndex);
public final void addPropertyChangeListener(String propertyName, PropertyChangeListener listener);
public final void removePropertyChangeListener(String propertyName, PropertyChangeListener listener);
// следующие операции действуют сразу для всех свойств инстанса компонента:
public final void addPropertyChangeListener(PropertyChangeListener listener);
public final void removePropertyChangeListener(PropertyChangeListener listener);
```

Здесь elementIndex соответствует индексу, который используется в методах доступа к значениям индексированных свойств (indexed properties) JavaBeans-компонент; методы регистрации/дерегистрации объектов-получателей событий об изменениях значений свойств полностью соответствуют средствам реализации связываемых свойств (bound-properties) JavaBeans. Для ускорения обращений вместо строки имени свойства (propertyName) можно использовать целочисленный индекс свойства (propertyIndex), получив его из типа заранее.

Указанные выше операции контролируются в динамике с использованием типов свойств, где содержится информация о возможностях выполнения операций доступа к ним для записи значения (write access: скалярный (W) и/или индексированный (w)), чтения значения (read access: скалярный (R) и/или индексированный (r)) и регистрации/дерегистрации объектов-получателей событий об изменениях значений свойств (bind access (B)). При операциях записи динамически контролируется тип записываемого значения. Нарушения приводят к исключительным ситуациям - разновидностям RuntimeException.

Компоненты JavaBeans являются «черными ящиками» с внешними интерфейсами, определяемыми с помощью интроспекции (рефлекции) их классов. Мы сохраняем основные средства JavaBeans-компонент, но наделяем их некоторыми характерными для BeanVM особенностями: реализации инстансов всех компонент BeanVM должны наследоваться (прямо или опосредованно) из предоставленного базового класса Bean (здесь мы используем короткие имена типов). Это позволяет рассматривать инстансы таких компонент как «серые ящики», используя внутреннюю реализацию их свойств суперклассом Bean.

Инстансы компонент BeanVM с реализациями, унаследованными из базового класса JVM Bean, обладают «самореализуемыми» свойствами (по описаниям их типов). При этом инстансы компонент-классов снабжены присущими им поведениями (реакциями на изменения значений свойств, определенными в их классах реализации); поведение инстансов составных компонент определяется и реализуется совместными поведениями инстансов компонент их внутренней реализации.

3.2 Компоненты-классы в BeanVM

При первом «знакомстве» BeanVM с классом создается и регистрируется в загрузчике типов соответствующий классу тип (тип-класс). Такое «первое знакомство» может произойти при явном вызове метода получения типа по классу, либо при первом инстанциировании класса реализации компонента, либо при использовании класса в качестве JVM-типа значения какого-либо свойства какого-либо класса, для которого определяется тип.

Имя типа-класса заимствуется из его класса реализации. Если класс является реализацией компонента BeanVM, предоставляется описание его интерфейса в терминах типов его свойств (propertyTypes). Класс реализации компонента подвергается интроспекции, которая предоставляет стандартный массив дескрипторов свойств (PropertyDescriptor'ов и/или IndexedPropertyDescriptor'ов); из них создается массив объектов типа PropertyType, каждый из которых содержит имя свойства (propertyName), тип значения (valueType) и тип доступа (accessType).

Имя свойства (propertyName) берется непосредственно из дескриптора. Типа значения свойства (valueType) является результатом применения операции Type.forClass(Class<?> c) к типу значения с точки зрения JVM, то есть к классу, указанному в дескрипторе. Значение accessType является описанием способов доступа к данному свойству, поддерживаемых его реализацией; оно формируется по информации дескриптора и содержит признаки наличия возможных операция над свойством (скалярных и/или индексированных). Таким образом, объекты типа PropertyType, используемые в описаниях типов BeanVM, являются аналогами объектов типа PropertyDescriptor, используемых в описаниях классов JVM при их интроспекции.

Заметим, что операция Type.forClass(Class<?> c), как и механизм интроспекции JVM, обрабатывает любые java-классы, включая классы-массивы; они тоже представляются соответствующими типами BeanVM. Однако, типы BeanVM, реализация которых не унаследована из класса Bean, не являются ссылочными типами BeanVM (не являются компонентами BeanVM); они могут быть использованы только в качестве типов значений свойств у инстансов компонент. Компонентом-классом BeanVM (по определению) является только тип BeanVM, классом реализации которого служит JavaBeans-компонент, имеющий своим предком в дереве наследования предопределенный суперкласс Bean.

Для реализации инстансов всех компонент-классов суперкласс Bean предоставляет конструктор без аргументов, который неизбежно вызывается при инстанциировании класса реализации любого компонента-класса. Этот конструктор обеспечивает получение BeanVM-типа для инстанциируемого класса и реализацию инстанса этого типа. Получение типа по классу реализации происходит с привлечением загрузчика типов, поддерживающего отображение класса JVM в соответствующий тип-класс BeanVM. Реализация инстанса типа содержит постоянную ссылку на свой тип.

Далее происходит реализация внутреннего представления свойств данного инстанса в памяти BeanVM. Рассмотрим сперва «бесконтекстное» инстанциирование компонента-класса, которое соответствует его инстанциированию в JVM как «обычного» JavaBeans-компонента, каковым он является. Так инстанциировался бы этот компонент в стандартном контейнере BeanBox.

При таком инстанциировании мы должны получить готовый к работе инстанс компонента с

инициализированными значениями всех его свойств. Однако, стандартный механизм интроспекции не предоставляет нам всей необходимой для этого информации: в дескрипторах свойств нет информации о значениях, которыми они должны быть инициализированы. Такая инициализация обеспечивается только поведением конструкторов классов реализации. Мы могли бы потребовать явного предоставления этой информации, например, путем нестандартных соглашений о программировании JavaBeans-компонент, либо предоставлять явные BeanInfo-классы (умножая количество классов), либо предоставлять отдельные описания такой информации (в разных бытующих для этого форматах), либо наплодить нестандартные (с точки зрения JDK) аннотации, как это делают разные популярные библиотеки… Мы предпочитаем, чтобы программа сама умела выразить и сделать то, что ей положено.

Нам надо получить информацию о том, как должны быть инициализированы свойства у инстансов нашего компонента при инстанциировании его класса реализации вызовом его конструктора без параметров: дополнительно к объектам PropertyType(s), создаваемым на основании инстроспекции класса реализации, нам надо узнать, какими значениями соответствующих типов должны быть инициализированы свойства при создании инстанса компонента. Память для хранения их значений будет определяться не самим классом реализации инстанса компонента, а его суперклассом Bean. Для получения значений инициализации мы применяем ту же идею, что будем «в больших масштабах» использовать в дальнейшем: мы создадим инстанс прототипа, который сам проделает нужную инициализацию, а затем извлечем из него информацию, необходимую для уточнения типа нашего (бесконтекстного) компонента (в данном случае – значения инициализации его свойств).

Чтобы создать инстанс прототипа и позволить конструктору класса реализации инициализировать этот инстанс, суперкласс Bean предоставляет *внутренний* BeanAPI, которым пользуются все его наследники.

3.3 Внутренняя реализация интерфейса инстанса компонента

Под внутренней реализацией интерфейса инстанса компонента мы понимаем реализацию набора его свойств средствами предоставления типизированной памяти BeanVM.

Решение этой задачи обеспечивает инстанс типа BeanVM, создаваемый фабрикой, работающей внутри конструктора инстанса любого компонента - внутри конструктора суперкласса Bean. При этом фабрика, создающая внутреннюю реализацию инстанса компонента учитывает контекст его инстанциирования (наличие и тип контейнера, в котором происходит инстанциирование).

Поскольку реализации инстансов компонент-классов и инстансов составных компонент имеют общий суперкласс Bean, создания инстансов обеспечиваются сходным образом.

3.3.1 Компонентная реализация памяти инстансов компонент

В принципе, можно использовать идею компонентности, начиная с уровня ячеек типизированной памяти BeanVM, чтобы с их помощью обеспечить реализацию отдельных свойств для инстансов компонент.

При этом можно использовать простейший тип – «Типизированная переменная» (TypedVariableType), инстансы которого представляют собой «ячейки памяти», способные хранить значения заданного типа T, что контролируется динамически при выполнении операций записи. Помимо такого присваивания, имеются операции чтения значения и операции регистрации/дерегистрации объектов, оповещаемых при изменениях значения. Такие типизированные переменные можно рассматривать как динамический аналог параметризованного типа Variable<T> с тремя операциями: write (W), read (R) и bind (B) – для реализации связываемых свойств (bound properties). Для хранения значений

индексированных свойств (и/или значений, чей тип значений – массив), добавляются операции indexed-write (w) и indexed-read (r); мы ограничиваемся поддержкой одномерных массивов в качестве значений реализуемых свойств.

Чтобы создать новый тип для создания экземпляров переменных со значениями типа T («компонент-фабрику» создания типов Variable<T>), нам надо иметь объект-прототип (типа TypedVariablePrototype), который обладает функциональностью типизированной переменной Variable<T> с операциями (W,R,B), где операция записи динамически контролирует тип T записываемого значения, а также – позволяет менять у себя текущий тип T на новый T' с «одновременной заменой» текущего значения типизированной переменной на значение нового типа T', используемое по умолчанию.

Мы используем общее правило получения «глобального значения по умолчанию» (global default value) для любого типа T. Все типы BeanVM подразделяются на *типы-значения* (*value types*) и *ссылочные типы* (*reference types*); любой ссылочный тип имеет глобальное значение по умолчанию равное null, и каждый тип-значение до своего использования регистрирует себя в глобальной таблице таких типов BeanVM вместе со своим глобальным значением по умолчанию. Типы-значения, реализованные java-классами, могут регистрировать себя в этой таблице при выполнении своих статических инициализаторов.

Тип TypedVariablePrototype может быть реализован как тип-класс BeanVM, имеющий свойство “valueType” и свойство “value”, с указанной выше логикой реакции на изменения их значений. Инстанс такого компонента-прототипа может быть легко преобразован в нужный тип типизированной переменной BeanVM - TypedVariableType (в нем присутствует вся необходимая для этого информация). Заметим, что объектом-прототипом могла бы быть сама переменная с записанным в ней типизированным значением (без свойства “valueType”), если бы по нему можно было узнать его тип значения (valueType), чему препятствует значение null (мы здесь не вдаемся в дискуссию об этом).

Динамический контроль типа позволяет автоматически создавать типизированные переменные на этапе исполнения – даже для тех типов значений, которые сами будут динамически генерироваться. Для типов значений, реализованных заранее известными классами, можно использовать JavaBeans-компоненты со свойством “value”, имеющими нужные JVM-типы значений (включая примитивные) непосредственно в реализациях методов доступа. Готовые классы реализации таких JavaBeans-компонент можно искать и подгружать динамически, получая их имена с помощью дополнительной мета-информации (соглашений об именовании). Эту возможность можно рассматривать как средство оптимизации работы BeanVM методами кодогенерации в JVM.

Такой «полностью компонентный» подход к организации памяти BeanVM, начиная с уровня отдельных типизированных ячеек памяти, был опробован реализован в предшествующей версии проекта [7]. Он является избыточным («чрезмерно компонентным») потому, что мы имеем дело с компонентами, обладающими постоянными, фиксированными при их создании, наборами свойств известных типов. Мы рассматриваем компонентную модель, где элементарным (атомарным, неделимым) компонентом является компонент специального вида с определенным набором свойств, а не отдельная его составная часть (отдельное свойство). Мы имеем дело с «молекулами», а не с «атомами», и всегда знаем количество свойств и их типы. Внутренняя реализация типизированной памяти BeanVM средствами суперкласса Bean скрыта внутри него (допуская оптимизацию).

3.3.2 Внутренний интерфейс реализации инстансов

Внутренний интерфейс для реализации доступа инстансов компонент к их свойствам, предоставленный в базовом классе Bean, выглядит следующим образом:

```
protected final Object get(int propertyIndex); // операция чтения
protected final Object get(int propertyIndex, int elementIndex); // индексированное чтение
```

```
protected final void set(int propertyIndex, Object newValue); // операция записи
protected final void set(int propertyIndex, int elementIndex, Object newValue); // индексированная запись
```

Автор реализации компонента (класса, наследованного из класса Bean, которому доступны эти операции), пользуется этими методами при реализации внешних методов доступа к свойствам инстанса компонента.

Например, для реализации свойства “foo” со значением примитивного типа float можно в классе реализации компонента определить методы следующим образом:

```
public void setFoo(float newValue) {set(FOO_INDEX, newValue);}
public float getFoo() {return (float) get(FOO_INDEX);}
```

Здесь подразумевается получение propertyIndex по propertyName статическим инициализатором класса:

```
static final int FOO_INDEX = Type.getPropertyIndex(“foo”);
```

Эта операция реализуется с использованием информации, известной на этапе создания типа по классу его реализации.

Заметим, что в данном примере реализации свойства “foo” скрыты неявные операции boxing/unboxing для примитивного типа float, но не скрыто явное нисходящее преобразование (downcast) к типу, возвращаемому при выдаче значения свойства; оно необходимо для любых возвращаемых этим методом типов (при отсутствии специальной кодогенерации для реализации свойств).

По умолчанию свойство “foo” реализуется как «связываемое» (bound property), что соответствует использованию аннотации @BeanProperty (из современного стандартного Java API). Правомочность операций W и R обеспечится наличием соответствующих методов доступа свойства. Контроль типа при присваивании будет обеспечен динамическими средствами. Мы оставляем возможности многочисленных оптимизаций средствами кодогенерации, в том числе – динамической, методами Monkey Patching [8] с помощью разных имеющихся для этого библиотек, за рамками нашего обсуждения.

3.3.2 Бесконтекстное инстанциирование

Вернемся к инстанциированию компонента-типа. Реализация инстанса включает в себя выделение памяти для хранения значений свойств; инициализация обеспечивает наличие в них начальных значений.

Например, если у нас есть компонент-тип “Bar” со свойством “foo”, реализованный классом Bar.class, то операция Type.forClass(Bar.class) вернет нам его тип Bar, где будет тип свойства, содержащий:

- “foo” – propertyName (имя свойства);
- Type.forClass(float.class) – property valueType (тип значения свойства);
- (W,R,B) – property accessType (writable, readable, bound – права доступа свойства).

При реализации инстанса такого бесконтекстного типа, мы отведем для хранения значения свойства “foo” переменную, обеспечив выполнение для нее указанных операций. Автор JavaBean-компонента Bar.class может инициализировать свойство “foo” значением, например, 1.0F (если глобальное значение по умолчанию 0.0F для типа float не годится).

При создании компонента-типа (типа по классу) создается его первый – служебный – инстанс, который является инстансом-прототипом (будем называть его *протоинстансом*) всех последующих инстансов этого типа. Протоинстанс при создании получает внутреннюю реализацию всех своих свойств, но так, что каждое из этих свойств реализуется переменной с максимально возможным типом доступа (accessType(s)) для своего типа значения valueType (для скалярных типов свойств это будет (W, R, B), для индексированных это будет (W, R, B,

w, r). В качестве начальных значений свойств протоинстанса берутся глобальные значения по умолчанию для типов значений свойств. Все это происходит в контексте выполнения конструктора суперкласса Bean, который неизбежно вызывается перед вызовом конструктора любого компонента. После создания протоинстанса суперконструктором Bean() дорабатывает конструктор исходного класса компонента, который получает возможность уточнить начальные значения своих свойств, пользуясь максимальными правами доступа внутренней реализации протоинстанса и специальными операциями инициализации, которые реализация протоинстанса для этого предоставляет:

```
protected final void initPropertyValue(int propertyIndex, Object initValue);
protected final void initPropertyValue(int propertyIndex, int elementIndex, Object initValue);
```

После инициализации протоинстанса конструктором компонента с помощью этих операций происходит окончательное формирование BeanVM-типа: типы свойств дополняются уточненными начальными значениями инициализации (отличными от глобальных значений по умолчанию). Этими итоговыми значениями протоинстанса будут инициализироваться все последующие инстансы (бесконтекстного) компонента – уже без помощи его конструктора: *реализации указанных операций инициализации всех инстансов типа, кроме протоинстанса, являются пустыми операциями*.

Мы видим, что даже внутренняя реализация бесконтекстного инстанса компонента-типа зависит от контекста инстанциирования самой этой реализации: она может быть реализацией протоинстанса или инстанса готового типа (что влияет на выполнение операций).

Разумеется, права доступа свойств у инстансов типа (в отличие от протоинстанса) определяются (уточняются в контексте использования) так, как указано в их реальных типах.

При «самореализации» инстансов по их типам используются правила отведения памяти в зависимости от действующих прав доступа: если свойство допускает операции связывания или записи (B, W и/или w), то потенциально изменяемое значение хранится в переменной (RAM BeanVM); иначе значение хранится в константной ячейке (ROM BeanVM).

Компонент-класс BeanVM реализован как «обертка» своего JavaBean-класса, но инстанс этого класса реализации, является «оберткой» инстанса своего BeanVM-типа, выполняющего внутреннюю реализацию инстанса компонента-класса. Это иллюстрирует рис. 2.

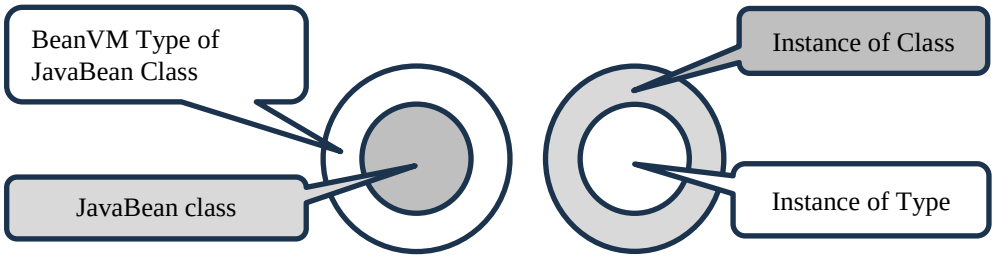

Рис. 2. Отношения вложенности типов BeanVM, классов JVM и их инстансов.
Fig. 2. Nesting relationships of BeanVM types, JVM classes and their instances.

При программировании JavaBean-класса реализации BeanVM-компонента использование внутреннего BeanAPI выглядит естественно и не требует других, специальных средств предоставления значений инициализации.

Вместе с тем, есть и альтернативный способ доопределения компонента-класса значениями для инициализации свойств, не связанный с использованием конструктора. Можно просто передать такие значения инициализации в статическом инициализаторе класса реализации компонента. Для этого есть метод доопределения типа, принимающий параметром Map<String, Object>, с помощью которого статический инициализатор класса реализации

типа может уточнить своему типу BeanVM значения инициализации поименованных в нем свойств.

Тем не менее, подход с использованием протоинстанса полезен тем, что он уже на уровне элементарных компонент дает пример преобразования инстанса прототипа в тип.

3.3.3 Контекстно-зависимое инстанциирование

Смысл создания компонент заключается в многократном использовании их инстансов при разных обстоятельствах. Мы допускаем, что инстанс класса Bar из приведенного ранее примера будет использоваться там (в таком контексте), где изменений значения свойства “foo” вообще быть не должно, а само значение свойства “foo” должно быть равно 2.0F.

Мы хотим использовать инстансы компонент-классов в разных контекстах и предполагаем возможность настройки реализации инстансов на контекст их использования. Это означает, что при *инстанциировании одного и того же класса* реализации компонента нам надо уметь *инстанциировать разные типы BeanVM*, инстансы которых являются внутренними реализациями инстансов класса. Фабрика создания внутренних реализаций должна учитывать контекст инстанциирования.

По определению компонент-классов, они, являясь JavaBeans-компонентами, могут инстанциироваться в JVM только вызовом их конструктора без аргументов (без возможности передать ему контекст инстанциирования). Тем не менее, в контексте вызова такого конструктора фабрика создания внутренней реализации инстанса (в суперконструкторе Bean()) может получить эту информацию с помощью операций BeanVM (обращения к скрытому стеку вызовов BeanVM, доступному для базового класса Bean). Этот прием позволяет конструктору компонента создать его внутреннюю реализацию инстансом типа BeanVM с учетом информации о контексте инстанциирования и предоставить при этом разные контекстно-зависимые возможности.

Прием с использованием неявной передачи контекстной информации при инстанциировании (через скрытый стек BeanVM) можно сравнить с инстанциированием нестатических вложенных классов (*inner*) языка Java, но в них передача неявного параметра (ссылки на охватывающий объект) обеспечивается соответствующей кодогенерацией при компиляции; мы используем динамический аналог.

Соответственно, BeanVM предоставляет разные операции инстанциирования компонент. Операция бесконтекстного инстанциирование компонент-классов не отличается от способов инстанциирования JavaBeans-компонент (каковыми они являются).

Дополнительно, имеются операции инстанциирования в «контейнерах» – инстансах специальных типов BeanVM, внутри которых (в контексте которых) создаются инстансы BeanVM-компонент. Эти операции имеют вид:

<container>.instantiate (ComponentType type) --> <instance>,

где <container> - инстанс компонента-контейнера, type – инстанциируемый компонент (тип) BeanVM, и <instance> – получаемый инстанс (объект типа type). Заметим, что так могут инстанциироваться только типы-компоненты; типы, не являющиеся компонентами (*типы-значения*), инстанциируются поведением самих компонент (средствами JVM), и их инстансы видимы в BeanVM только как значения свойств инстансов компонент.

Компоненты определяют максимальную функциональность своих инстансов *a-priori*, до их конкретного использования, где она не всегда необходима. Классические JavaBeans-компоненты на этом останавливаются и позволяют настраивать свои инстансы только путем изменения значений свойств (вызовами имеющимися для этого методов).

Наши компоненты-классы могут настраивать свои инстансы на контекст их использования благодаря их внутренней реализации инстансами BeanVM-типов, зависящими от контекста инстанциирования, без изменения класса реализации. Фабрика внутренних реализаций

инстансов компонент-классов (в суперклассе Bean) понимает контекст инстанциирования, а все операции внешнего интерфейса инстанса компонента делегируют к методам инстанса внутренней реализации. Инстанс JavaBean-класса при выполнении конструктора суперкласса Bean получает final-ссылку “thisImpl” на реализующий его инстанс соответствующего типа BeanVM (как показано на рис. 2).

Поскольку класс компонента всегда неизменен, а поведение инстансов определяется в нем, контекстно-зависимое изменение их функциональности может происходить только «в меньшую сторону», за счет того, что для данного контекста некоторая имеющаяся функциональность оказывается «незадействованной». Это и определяется с помощью протоинстанса (инстанса прототипа), в котором можно указать избыточные для данного контекста возможности (операции над свойствами) и убедиться в этом на практике.

Фабрика, которая обеспечивает внутреннюю реализацию инстансов компонент, реализует их свойства с учетом различных факторов, в зависимости от которых доступ к свойству может декорироваться. Такими факторами являются:

- тип доступа к свойству (в зависимости от него обеспечивается память для свойства);
- тип значения свойства (типы значений могут сами доопределять правила поведения методов доступа и правила валидации значений, нарушения которых приводят к соответствующим исключительным ситуациям; доступ к значениям-массивам JVM реализуется с копированием этих значений; можно управлять возможностью присваивания значения null, и т.п.);
- контекст инстанциирования (реализации свойств инстансов-прототипов и инстансов составляющих типов отличаются);
- контекст использования инстанса (влияет на прототипирование составляющих типов и их последующее инстанциирование).

3.4 Составные компоненты

Составные (composed) компоненты – это *динамически определяемые типы-компоненты*, которые не являются представителями классов своих реализаций в BeanVM, но создаются в динамике с помощью компонент, инстансы которых которые создаются, агрегируются и настраиваются в контексте инстанса специального контейнера-прототипа, после чего он преобразуется в составной тип (composed-type).

Как и компоненты-классы, composed-компоненты – это типы BeanVM, которые уникально именуются и предоставляют интерфейс к своим инстансам в терминах свойств (теми же средствами): реализации их инстансов тоже обеспечиваются базовым классом Bean. Поведение инстансов составных компонент определяется совместным поведением инстансов составляющих их типов.

3.4.1 Прототипирование составных типов

Для динамического определения составных компонент используется компонент-класс ComposedBeanPrototype - прототип составного типа (*мета-компонент* создания компонент).

Инстанс ComposedBeanPrototype – это контейнер инстансов любых компонент BeanVM. Он является для них *контекстом инстанциирования*. Инстансы компонент, создаваемые в нем, используются для *агрегирования* и настройки *работоспособного* составного протоинстанса. При этом они играют роль прототипов, из которых будут произведены составляющие типы (*composing-types*) композиции – составного типа (*composed-type*), получаемого в результате преобразования инстанса контейнера-прототипа в тип.

Создавая инстансы компонент в контексте инстанса контейнера-прототипа, мы хотим уметь их настраивать и использовать совместно для обеспечения функционирования протоинстанса, составляемого из них. Для этого при инстанциировании компонента в таком

контексте предоставляются дополнительные возможности манипуляций с инстансом, выходящие за рамки возможностей бесконтекстно инстанциированного компонента. Эти возможности обеспечивает специальный объект – «обертка» (wrapper) создаваемых инстансов, предоставляемый контейнером-прототипом. Он позволяет реализовать:

- доступ к свойствам самого инстанса (тем самым прототипируются контекстно-зависимые уточнения значений инициализации для будущего составляющего типа);
- понижение типа доступа к свойству относительно заданного в типе компонента (при прототипировании мы хотим уметь «сознательно отказываться» от избыточных в данном *контексте использования* возможностей инстанса компонента; мы не изменяем исходный бесконтекстный компонент, но «декорируем» реализацию доступа к значениям его свойств переменными полями их текущих прав; это позволяет прототипировать контекстно-зависимые понижения типов доступа к свойствам для будущего составляющего типа;
- замену внутренней реализацию свойства инстанса на реализацию однотипного (по типам значения и доступа) свойства другого инстанса, имеющегося в их общем инстансе контейнера-прототипа (это средство «разделения» реализаций свойств мы уточним ниже).

3.4.2 Определение свойств составных компонент

Составной (composed) компонент определяет конкретный набор типов свойств (возможно пустой) для взаимодействия с его инстансами. В реализации – это массив объектов – типов свойств, которые используются и для определения типов свойств компонент-классов. Способом получения типа набора свойств является преобразование его прототипа в тип. Для этого надо:

- создать инстанс прототипа (компонента-класса PropertySetPrototype);
- заготовить набор типов значений свойств будущего составного компонента;
- сформировать Map<String, Type> с именами и типами значений свойств;
- присвоить этот объект в свойство “valueTypesMap” инстанса PropertySetPrototype; поведение прототипа создаст и предоставит протоинстанс, у которого будут иметься свойства с указанными именами и типами значений; они будут инициализированы глобальными значениями по умолчанию своих типов значений и предоставлять максимально возможный доступ; созданный объект-протоинстанс предоставляется как выходное значение свойства “protoinstance” инстанса прототипа;
- инициализированные значения свойств можно изменить (либо с помощью свойства прототипа “initValuesMap”, куда надо передать Map<String, Object> с задаваемыми значениями, которые будут использованы только для соответственно именованных свойств протоинстанса, либо доступными операциями записи значений);
- последним шагом является указание о понижении прав доступа для свойств протоинстанса; для этого используется свойство “accessTypesMap” со значением Map<String, AccessType> с пониженными типами доступа у именованных свойств.

Последний шаг указывает прототипу о завершении настройки протоинстанса и стимулирует выдачу результата – компонента PropertySetType (типа набора свойств) – в качестве значения одноименного выходного свойства его прототипа. Этот шаг является обязательным (если понижения прав доступа не требуются, надо просто передать пустой Map-объект).

Типы BeanVM предоставляются операцией Type.forName(String typeName). Имя компонента-класса заимствуется из его класса реализации. Имя составного компонента определяются значением свойства “name” инстанса его прототипа (уникальность имен обеспечивается загрузчиками типов, контролирующими их создание и именование).

3.4.3 Прототипирование взаимодействия инстансов компонент

Составной прототип должен реализовать функциональность протоинстанса с внешним интерфейсом к нему в терминах набора свойств.

Внешний интерфейс любого инстанса компонента реализуется инстансом BeanVM-типа PropertySetType с реализацией JVM-классом PropertySet (наследником класса Bean), предоставляющим реализацию конкретного набора свойств инстанса компонента. Эти свойства «соединяются» с однотипными свойствами других инстансов компонент, связывая интерфейс с реализацией.

Популярным способом взаимодействия между однотипными свойствами разных инстансов JavaBeans-компонент являются сигналы PropertyChangeEvent(s), передаваемые от связываемых свойств (bound properties); эти средства сохраняются. Однако, реализация *двусторонних связей* между инстансами компонент с помощью таких событий от свойства источника к однотипному по значению свойству приемника является громоздкой, сопровождается переписью значений и не решает задачу, так как работает только для связываемых свойств.

Поэтому имеется механизм реализаций свойств «с общей памятью», где значения не надо переписывать, обращаясь к ним по ссылкам (аппаратная аналогия: контакты элементов можно соединять посредством проводов, но можно их соединить и непосредственно).

Смысл инстансов типа PropertySetType заключается в предоставлении ими определенного набора реализаций свойств, которые могут разделяться другими инстансами компонент. Инстансы, реализующие наборы свойств, не имеют определенного в их типе поведения, но позволяют использовать поведения, определенные реализациями компонент, инстансы которых разделяют реализации своих свойств с ними. Необходимым условием разделения реализации свойств является их однотипность – совпадение типов значений и доступа. Инстансы компонент, использующие разделяемые реализации свойств, обращаются к их значениям и реагируют на их изменения так же, как при использовании своих собственных реализаций свойств. Использование разделяемых свойств не отражается на механизме реализации связанных свойств, так как регистрация/дерегистрация слушателей связана с самим объектом свойством, а не с памятью, хранящей значение (свойство реализуется как контекст использования типизированной ячейки памяти, а не она сама).

При создании инстанса компонента в контексте инстанса контейнера-прототипа контекст инстанциирования предоставляет возможность доступа к другим имеющимся в этом контексте инстансам. Их можно найти благодаря регистрации всех инстансов под их локально-уникальными именами внутри таблицы именованных объектов инстанса контейнера-прототипа. Специальная операция позволяет инстансу в контейнере прототипа в качестве реализации своего свойства использовать реализацию свойства из инстанса другого компонента (набора свойств) этого же контейнера. Операция выполняется успешно только при совпадении типов значений и доступа используемого (*разделяемого*) свойства и использующего (*разделяющего*) свойства. При этом одно разделяемое свойство может разделяться многими разделяющими свойствами от разных инстансов компонент, список которых поддерживается для оповещения всех инстансов при изменении общего значения кем-либо из них.

С точки зрения инстанциирования это означает, что внутренняя реализация инстанса в контексте контейнера прототипа допускает последующую замену реализации его свойства - с использования контекста его собственной памяти для хранения значения на использование контекста памяти разделяемого свойства другого инстанса из того же контейнера (который должен быть известен как контекст инстанциирования).

Контекстно-зависимая настройка типов доступа может производиться только для свойства, которое разделяется; разделяющие свойства всегда имеют тип доступа разделяемого

свойства. Разделение свойств возможно только у инстансов PropertySetType; это связано с их ролью в реализации поведения инстансов составных типов.

Информация о реализациях свойств при прототипировании переносится в описания составляющих типов составного типа, получаемого из прототипа; она используется при инстанциировании составляющих типов в контексте инстанса составного типа, когда создается он.

3.4.4 Прототипирование поведения

Взаимодействие инстансов компонент происходит в терминах операций с их свойствами и подразумевает изменения их значений. Стандартным способом организации совместного поведения инстансов JavaBean-компонент является создание графа распространения событий между ними, получаемых от «связываемых свойств» (bound properties). Механизм распространения событий типа PropertyChangeEvent обеспечивает реализацию поведения с помощью «каскадов событий», которые завершаются даже при наличии циклов в графе их распространения. При этом явной зависимости результатов общего поведения инстансов компонент от контекстов их использования при агрегировании может не быть.

Реализация поведение определяется типом контейнера, в котором моделируется поведение, и типами используемых в нем компонент. Есть ряд важных задач моделирования, где контексты использования инстансов компонент требуют учета. К ним относятся задачи моделирования реальных объектов: компьютерная графика, программирование GUI (изначально основная область применения JavaBeans-компонент, из-за чего пакет java.beans оказался в модуле java.desktop), и другие.

Контексты использования инстансов компонент возникают, когда одни инстансы компонент присваиваются в качестве значений свойств других инстансов, допускающих такое присваивание (*setter dependency injection*). При таких присваиваниях возможно создание направленного графа ссылок с вершинами (узлами) из инстансов компонент и со связями от узла, обладающего свойством со ссылочным типом значения, к узлу, который является таким значением.

Для учета контекстных зависимостей часто используются деревья, что является ограничением, затрудняющим разделение данных в узлах разными контекстами. С другой стороны, наличие циклов в графах затрудняет учет контекстов использования узлов.

Представление контекстных зависимостей в виде направленного ациклического графа (Directed Acyclic Graph - DAG) позволяет естественно переиспользовать общие данные узлов. Обход (траверсирование) такого графа позволяет получить информацию о контекстах использования узлов на стеке траверсирования графа. Такой вариант является расплатой временем траверсирования (с многократным построением стека) за экономию памяти и затрудняет параллельное использование модели данных.

При реализации контейнеров и компонент, которые предполагают использование контекстных зависимостей, в BeanVM используется «компромиссный» вариант представления инстанса контейнера: одновременно с DAG из инстансов компонент, который создается setter’ами свойств ссылочных типов и допускает использование одного инстанса несколькими другими, мы строим *дерево контекстов* использования узлов этого графа.

Узлы дерева контекстов хранят контекстно-зависимую информацию и не требуют многократного траверсирования (мы расплачиваемся памятью дерева за экономию времени траверсирования графа). Изменения топологии графа и соответствующая корректировка дерева контекстов происходят гораздо реже, чем возникает необходимость в использовании контекстно-зависимой информации (например, для графического отображения). Кроме того, наличие дерева контекстов узлов графа упрощает валидацию его ацикличности.

3.4.5 Взаимосвязь прототипа и протоинстанса

В терминах графа (DAG), узлами которого являются инстансы компонент, и дерева их контекстов можно пояснить связь между прототипом и его протоинстансом. Составной прототип представляется графом, который строится с помощью присваиваний свойствам значений ссылочных типов (setter dependency injection). Протоинстанс (такого прототипа) представляется деревом контекстов узлов графа, на котором отражается и с помощью которого реализуется поведение протоинстанса. При прототипировании эти структуры данных строятся, настраиваются и функционируют одновременно.

При инстанциировании типа, реализация которого до его инстанциирования получена из прототипа, инстанс создается уже при наличии постоянного (immutable) описания типа, где можно хранить автоматически обобщаемую всеми инстансами информацию, что открывает возможности для значительной оптимизации ресурсов памяти и времени.

В терминах DAG и его дерева контекстов можно выразить и внутреннюю организацию инстансов элементарных компонент - реализацию их свойств в базовом классе Bean (хотя она скрыта от пользователя). Роль базовых компонент при этом играют типизированные переменные. Их инстансы образуют прототип. Контексты инстансов типизированных переменных - суть свойства, с помощью которых реализуются методы доступа к значениям в узлах прототипа. Протоинстанс является корнем дерева, содержащего свойства, и внутренней реализацией прототипа, который является контейнером узлов графа. Инстанс прототипа содержит корень дерева (протоинстанс), и протоинстанс, как корень дерева контекстов, имеет ссылку на контейнер графа; просто и граф, и дерево представлены своими простейшими вариантами («кустами»). Уже для элементарных компонент такая реализация позволяет автоматически переиспользовать постоянные (immutable) значения свойств.

4. Обсуждение

Мы рассмотрели основы эволюционного подхода к развитию компонентной модели JavaBeans в направлении добавления к ней динамических средств создания компонент. Этот подход позволяет расширить возможности компонентной модели путем добавления новых наборов компонент, являющихся JavaBeans-компонентами, для прототипирования и создания новых типов-компонент в динамике. Описание всех аспектов предложенного подхода выходит за рамки данной статьи, в которой изложен общий подход, основные понятия и связанные с ними направления реализации.

Компоненты – это *типы деталей*, которые достаточно *универсальны* (не ограничены контекстом создания) и сконструированы *заранее* (до создания самих деталей) для определения *конечных продуктов* и/или *новых компонент*. При создании компонент в динамике *без кодогенерации* требуется расширенное понятия типа, включающее в себя типы, создаваемые на этапе исполнения (at runtime).

Сборка с применением компонент предполагает использование *самих деталей* (инстансов имеющихся типов) в некотором специальном «ящике» – *прототипе* (он сам – деталь своего типа). Там создается *агрегат* из других деталей по определенным для этого правилам. Эти правила определяет компонентная модель; реализуют правила ее компоненты, инстансами которых могут быть контейнеры-прототипы, внутри которых происходит агрегация, и компоненты, инстансы которых агрегируются в них. Соблюдение этих правил «заложено в генах» компонент (в их типах). Корректное выполнение правил реализуется (воплощается) в их инстансах с помощью наследования ими общей для них реализации.

В прототипе возникает (составной) образец деталей нового типа. Этот образец реально работоспособен, он настраивается и проверяется. Из него можно извлечь состояние агрегации, где все элементы (сама деталь и ее связи с другими деталями в агрегации) будут служить образцами для определения композиции – *составного типа* и его *составляющих типов*, показывающих, как надо применять (создавать и использовать) такие детали.

Это *не есть копирование образца*: составляющие типы учли контекст использования своих деталей и избавились от их ненужной универсальности. Это может дать большую экономию ресурсов (памяти и времени) при применении нового типа (по тем же правилам).

5. Распространенные подходы

Парадигма компонентно-ориентированного программирования давно и широко обсуждается в литературе. При этом ее практическое использование нельзя назвать повсеместным, что обусловлено спецификой предметных областей, решаемых задач и используемых средств.

За годы развития Java-платформы накоплен большой опыт использования компонент при создании как локальных приложений, так и больших распределенных систем. Почти везде при этом используется термин «Bean», но далеко не всегда следуют его определению в [4].

Начиная с первых Java-технологий для серверных приложений, наметилась тенденция использования компонентных моделей, ориентированных на использование компонент в специфических контейнерах (таких, как модель Enterprise JavaBeans [9]). В них допускается применение JavaBeans-компонент, играющих вспомогательную роль. Этот подход используют и современные технологии, основанные на Spring Framework [10], где понятие «Bean» имеет свой смысл, отличный от его определения в [4].

JavaBeans-компоненты используются в библиотеках создания GUI, в том числе в JavaFX [11], где есть набор специфических реализаций свойств компонент, но нет выхода за рамки использования скомпилированных типов.

Все эти технологии прошли многолетний путь: от программной конфигурации составных частей, к использованию для этого специальных форматов (XML-подобных) и к применению многочисленных специальных аннотаций (без явного определения компонентной модели).

Они не дают примеров использования стандартных компонент для получения «себе подобных» без кодогенерации, стандартными средствами самих компонент, и не предоставляют средств оптимизации (контекстно-зависимой), упомянутых ранее. Эти средства связаны с наличием общего базового класса реализации инстансов компонент в BeanVM, тогда как, например, в идеологии Spring Framework подчеркивается отсутствие общего суперкласса, отличного от класса java.lang.Object (и выхода за рамки традиционных средств Java-программирования).

В работе предлагается подход к реализации новых возможностей в рамках исходной компонентной модели, которую можно дополнять новыми компонентами, снимающими необходимость в кодогенерации и допускающими реализацию специфических контейнеров и их компонент, а не наоборот (когда специфические контейнеры по-своему определяют компонентную модель).

Возможность кодогенерации остается как средство оптимизации. Методы кодогенерации могут переводить *уже работоспособный составной компонент* в эквивалентный компонент-класс (из BeanVM в JVM, что напоминает JIT-компиляцию, которая сработает потом).

6. Заключение

В настоящее время проект состоит из пилотных проектов, где опробованы и протестированы основные рассмотренные в работе решения. Предполагается объединение этих проектов в программный продукт – библиотеку для прикладного использования. Необходимо также предоставить инструмент для визуального манипулирования компонентами, поддерживающий новые возможности, который может стать средством компонентного программирования.

Список литературы / References

[1]. Douglas McIlroy. Mass-produced software components. Электронный ресурс. https://www.cs.dartmouth.edu/~doug/components.txt (дата обращения 22.08.2025).

[2]. A.J.A.Wang, K.Qian, Component-oriented programming, John Wiley & Sons, Inc., 2005.

[3]. Kung-Kiu-Lau, Zheng Wang (2006) A Survey of Software Component Models (second edition), School of Computer Science, The University of Manchester, Preprint Series, CSPP-38.

[4]. JavaBeans(TM) Specification. Электронный ресурс. https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/
Архивировано:
https://web.archive.org/web/20210416231451/https://www.oracle.com/java/technologies/javase/javabeans-spec.html (дата обращения 22.08.2025).

[5]. VRML – The Virtual Reality Modeling Language. Электронный ресурс. https://www.web3d.org/documents/specifications/14772/V2.0/ (дата обращения 22.08.2025).

[6]. Проект Babylon. Электронный ресурс. https://openjdk.org/projects/babylon/ (дата обращения 22.08.2025).

[7]. E.Grinkrug. A Framework for Dynamical Construction of Software Components. A.K.Petrenko and A.Voronkov (Eds.): PSI 2017, LNCS 10742, pp. 163-178, 2018. https://doi.org/10.1007/978-3-319-74313-4_13.

[8]. N. Frankel. Monkey-Patching in Java. Электронный ресурс. https://dzone.com/articles/monkey-patching-in-java. (дата обращения 22.08.2025).

[9]. Enterprise JavaBeans Technology. Электронный ресурс. https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html (дата обращения 22.08.2025).

[10]. Spring Framework. Электронный ресурс. https://spring.io/projects/spring-framework (дата обращения 22.08.2025).

[11]. JavaFX Documentation. Электронный ресурс. https://openjfx.io (дата обращения 22.08.2025).

Информация об авторах / Information about authors

Ефим Михайлович ГРИНКРУГ – кандидат технических наук, старший научный сотрудник Института системного программирования им. В.П. Иванникова РАН с 2024 года. Сфера научных интересов: операционные системы, компонентно-ориентированное программирование, компьютерная графика, беспроводные сенсорные сети.

Efim Mikhailovitch GRINKRUG – Cand. Sci. (Tech.), Senior Researcher at the Ivannikov Institute for System Programming of the RAS since 2024. Research interests: operating systems, component-oriented programming, computer graphics, wireless sensor networks.