A COMPONENT MODEL WITH DYNAMIC PROTOTYPE TO TYPE TRANSFORMATION

Efim Grinkrug

Department of Software Engineering, National Research University Higher School of Economics, Moscow, Russia[1]
egrinkrug@hse.ru

ABSTRACT

The paper presents an extension for the JavaBeans component model that enables creating composed components dynamically, at runtime, without code generation. The composed components created can be used immediately for instantiation having their instances used for execution or for further components composition. The dynamic abilities are supported by extended type implementation based on additional superstructure provided with its Java API implementation and corresponding JavaBeans components. Using the component model and base components it provides, the new component composition is performed by building the composed prototype object that can be dynamically transformed into the new composed instantiable type. The component model can be used when implementing user defined types in declarative languages for event-driven models description.

KEYWORDS

Software Components, Component Model, JavaBeans, 3D Modeling

1. INTRODUCTION

The component-oriented programming (COP) is a promising approach for software development in many application areas. It provides many advantages from various points of view in software development process and constitutes the main idea of component-based software engineering (CBSE).

The idea of component-oriented programming is to create software products from composing parts – the idea that is at the base of the vast majority of technologies in other engineering areas. Composing parts of software products named components are created and used in accordance with a component model that defines what a component is, and what and how can be composed with that component [1].

In this paper we discuss some considerations about enhancing JavaBeans-component model that is widely used in Java software development. Java platform has become the most widely used object-oriented environment for software development starting from the time it was introduced [2]. The JavaBeans component model [3] initially was claimed as "the only component model for

[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).

Natarajan Meghanathan et al. (Eds) : ACSIT, FCST, ITCA, SE, ICITE, SIPM, CMIT - 2014
pp. 71–87, 2014. © CS & IT-CSCP 2014 DOI : 10.5121/csit.2014.4607

the Java machine" and it is widely used in Java-programming up to now, while currently there are many other, popular enough component models for Java-platform – both universal and domain specific (e.g. [4]).

This work is rooted in the practical experience earned from using JavaBeans-components to implement virtual reality modeling system with 3D-graphics support entirely in Java [5], and from developing instrumentation systems for wireless sensor networks modeling and commissioning (ZigBee Standard [6]). While the two kinds of applications are very different in nature, they both share the event-driven behavior model and benefit from component-based system implementation. We expect therefore that our conclusions may have some general application.

We begin with the problem statement, discussing how components are used and what the shortcomings of the JavaBeans component model are in Section 2, explaining our approach to the component model enhancements. Then we describe our component model with dynamic prototype to type transformation and its implementation principles in Section 3. After that in Section 4 we discuss how that component model can be used in application areas mentioned above. We find its place in the wide variety of component-based software technologies in Section 5 and conclude with future work direction.

2. COMPONENTS AND COMPONENT COMPOSITIONS

Since the time when component-oriented programming was recognized as base of software engineering [7], many definitions of a component were introduced. The most popular are following:

- "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [8];

- "A component is a software element (modular unit) satisfying the following conditions:

    1. It can be used by other software elements, its 'clients'.
    2. It possesses an official usage description which is sufficient for a client author to use it.
    3. It is not tied to any fixed set of clients."[9];

- "A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."[10].

A software component model mentioned in the last definition, should define the syntax of components (how they are constructed and represented), the semantics of components (what components are meant to be), and the composition of components (how they are composed or assembled) [1].

We are specifically targeting the Java-platform and JavaBeans component model because this is the most popular object-oriented software development platform and component model.

2.1. JavaBeans components and their usage

The popularity of the JavaBeans component model is based on its relative simplicity, wide range of abilities it provides and corresponding tools to demonstrate that abilities. The first sentence of the specification is: "The goal of the JavaBeans APIs is to define a software component model for Java, so that third party ISVs can create and ship Java components that can be composed together into applications by end users" [3].

The visual composition of pre-existing components is at the origin of the JavaBeans component model and it is stated in the initial definition: "A JavaBean is a reusable software component that can be manipulated visually in a builder tool." And the abilities of JavaBeans components to be manipulated visually make JavaBeans component model attractive to use when developing modeling and visualizing applications with interactive abilities support. While JavaBeans components can be used as class libraries in traditional software development process, we are mainly interested in their dynamic composition abilities and corresponding tools support.

From an external view, JaveBeans components can communicate using four kinds of ports: methods, properties, event sources and event listeners.

The notion of method of a component is directly bound to the notion of method in the component implementation language: callable method must be present in the component implementation class.

The notion of property, from a client point of view, can be used for getting its value, setting a new value, or for binding with it to be notified with events whenever it changes its values.

Event sources generate events of certain type, while event listeners receive the events. Event sources provide operations to connect and disconnect listeners, supporting event-driven behaviors in applications composed of components.

In CBSE, composition is a central issue, since components are supposed to be used as building blocks from some repository and assembled or plugged together into larger blocks or systems.
JavaBeans-component model can be considered from a general, idealized component model point of view [1, 11, 12], that is expected to have three stage lifecycle: 1) design stage, when components are designed and developed at source code level of their implementation language (i.e. in Java, in our case) and possibly compiled into binaries; 2) deployment stage, when binary component representations are supplied into composition environment, and 3) runtime stage, when the components are instantiated and executed in a running program. (Actually, what we are going to do is remove borders between the stages.)

At the design stage, JavaBeans components are implemented by Java classes that satisfy simple rules (JavaBeans design patterns). Essentially, JavaBeans component is a Java class instantiable in any context (public class having public constructor with no arguments) and having support for persistence (to save and restore states of its instances). These classes are distributable for reuse in binaries (Java byte-codes) along with other classes and resources used to implement them. (A JavaBeans component, therefore, is a Java class that implements, possible using other Java classes, a type of objects, or instances, it creates. These notions are often erroneously mixed in literature hiding class-based object-oriented nature of the component model.) At the design stage, components composition can be performed using Java programming technology chain, resulting in components byte-codes produced by compiler that actually uses the composing components as classes from class libraries.

At the deployment stage, there is some composition environment that supports JavaBeans composition visually. The JavaBeans API Specification was supplied with Bean Development Kit (BDK [13]) that contained the composition environment prototype, the BeanBox, to illustrate interactive and visual composition support for JavaBeans components. That approach has been integrated in various IDEs, but the BeanBox from BDK is still used having more dynamic abilities than the IDEs provide. At that stage, JavaBeans component are instantiated in the composition environment and their instances are combined together to provide the composite functionality required. In the BeanBox components instances can be combined interactively and dynamically using all kind of ports to connect them together. In contrast with the BeanBox, IDEs use more static approach: they do not support direct component instances interactions, but help generating the source code for it, that gets compiled. In that sense, we can consider that IDEs as design stage tools that use composition ability to automate some code generation. When component instances are composed in the BeanBox tool the whole composition can be saved (serialized using binary or some other format), and restored (de-serialized) later.

At the runtime stage, components that were created at the design stage are instantiated and executed. But some of them, acting as the containers can use serialized composite objects stored at the design stage by de-serializing them inside their instances. In any case, we see that our composition abilities could not create a new component without compiling it. We cannot produce a new component in BeanBox like interactive tool: components must be classes, and classes may be created by their byte-codes generation only.

2.2. Shortcomings of the JavaBeans component model

An advantage of the JavaBeans component model is in its simplicity and usability: it is not based on a set of specific interfaces to be implemented by the components to work, as e.g. OSGi does [4]. (The later enhancement of the JavaBean component model that introduced BeanContext related features in that style is much less popular.)

There are many other engineering areas that use component-based technologies. In most of them instruments used to create their basic components differ from instruments used to build composed components from them. Usually, composing technologies are simpler (e.g., so called “screwdriver production” for computers).

In software, XML syntax is often used to declaratively define and create a composite object from components instances; that XML-based instance composition is much more simple technology than the compiler used to create the components (classes) themselves. We used that technology in [5] to compose 3D-scene from JavaBeans-components instances in the same way that is used, e.g. in XAML [14], but we used the VRML-parser of our own. That composite object composing instruments are significantly simpler than Java compiler and other tools required to produce JavaBeans-components. Moreover, building a composite object can be done interactively, as the Bean Box demonstrates for JavaBeans-components.

When we build a composite object from JavaBeans-components using some declarative language, or when we build the composite object from JavaBeans-components in a composer tool (similar to the Bean Box), we discover that the component technology we use is not “self-closed”.

Our source components are types of objects implemented as classes (compiled and supplied in form of instantiable types libraries.) As a result of composition made with a parser or interactively in a tool we get some composite object composed from supplied components instances or an instance of some predefined container type filled with our composite object.

A composite object we have created may be a workable object (e.g. it can be some GUI implementation or a scene made from VRML subset implemented in [5]). We can save and restore that composite object, we can clone it (either directly or by serialization/deserialization), but we always deal with it as an instance of some (predefined) container type, but not as a new (component) type, that can be used the same way as the components it was composed from. Having the components set in hand initially, we cannot produce some new composed component that can be instantiated like a type, instead of cloning it like composed object.

It is graphically visible while manipulating in BeanBox. First, a library with compiled JavaBeans-components classes is loaded into BeanBox repository (ToolBox). In the BeanBox container instance we instantiate components (types) that are dragged & dropped into it from the ToolBox. The instances just created are depicted inside BeanBox container instance, having their property values depicted in the Properties panel, where some values can be edited. Further on, we can link the instances together by their references, assigning one instance as a property value for another, or bind them by events. All that BeanBox provided abilities correspond to the JavaBeans Specification [3], that the tool was aimed to illustrate. We just want to highlight the lack of ability to enrich the ToolBox, filled with components we used initially, with the result of our manipulations with them during the composition.

By simple manipulations with components, we cannot define composed component that can be manipulated the same way. It is important to note that we would like to have that ability by simple means, i.e. by means other than the means that were used to create initial components. (Compare: electronic LSI components are created by more sophisticated technology than putting them on a PCB together.)

Initially, basic JavaBeans-components are represented as compiled classes that were conformed to the JavaBeans-component definition and the JavaBeans design patterns while they were coded. Creating JavaBeans-components is usually done statically as the result of standard software developing process, with packaging their byte-codes into a Jar-archive. We say “usually” because some dynamic code generation is possible (either by creating source code with its compilation on the fly, or by mean of immediate code generation using specific libraries for that purpose that are available). Both variants can be used at runtime (dynamically), but we do not consider them as “simple”.

Basic JavaBeans-components are created statically, by “hand-made” programming. When we use “hand-made” (or even automated to some extent) programming while creating new composed JavaBeans-component, using existing JavaBeans-components just as class libraries, then we apply the same technology that was used to produce basic components initially.
While that approach can be (and is) widely used (and, being sophisticated, can provide more effective result), it seems that a way to create composed component by means other than that of basic ones, can give some advantages – both technical and ideological.

From the technical point of view, the ability to define composed component interactively (and without compiling, dynamically) is attractive by simplicity of its usage – just as putting LSI circuits on the PCB (while we can have VLSI circuit for all of them later).

From the ideological point of view, we can talk about creating some “higher virtualization level” with its new type system, incorporating the lower level abilities to create and support base components functioning (i.e. on top, above the Java-platform). Base JavaBeans-components are represented as instantiable types (implemented as Java classes). In case we want “by means of simple manipulations” (i.e. without code generation) to get new composed component, we need to generalize a component (type) notion so that we can, when creating new composed types, use basic and composed components (types) in the same manner (equally). Along this way, compiled

(or hardcoded) components and composed components are just two kinds of type implementations at our "new virtualization level". An idea of that higher level implementation is to use JavaBeans-components to add a superstructure for a component type system.

JavaBeans-components model is not "logically closed" in ideology point of view because of the following consideration. The components are supplied in form of classes (implementing abstract data types) to be instantiated, and the classes were designed in accordance with class-based object-oriented programming paradigm. When composing their instances in some composing environment (e.g. in the BeanBox), a composite object is created that is not an instance of some composed type; it is just a content of the pre-existing container instance it was built inside (i.e. the content of a BeanBox container instance). In JVM, types may come into existence only by loading byte-codes of the corresponding classes. That composite object can be cloned (serialized / de-serialized, etc.), but its usage in that way corresponds rather to prototype-based programming paradigm (we have no class created for it during the composition.) When performing a components composition without its code generation we have to use the composite object as a prototype, thus substituting initial programming paradigm by another. We cannot produce the composite entity of the same nature as we had initially to compose it (without having to use the same technology as we used to produce initial components.)

Note though that in electrical engineering, for example, we can build a functional unit from its components using much more simple technology than technology used to produce them (and it is a matter of integration density.) We can draw its scheme as the composed unit type description and put it into production for future reuse. In case we could be able to use microcircuits with more density, we could implement the composed unit in one chip using chip manufacturing technology that we used before for our components manufacturing. But meanwhile we just can use soldering-iron with wires. Roughly to say: that's why Intel develops sophisticated chips while others use to wire them together in more simple manner, placing them wired on PCBs for different purposes.

What we are looking for is a relatively lightweight, dynamic, interactive composition technology to create new composed components without their codes generation.

3. TYPES AND COMPONENTS

Generally speaking, when coding in Java we can only write data type definitions. These type definitions are compiled into class-files with byte-codes that appear to be the runtime types in JVM upon class loading. Inside JVM, the types are represented by objects of type Class that are produced by class-loaders. All objects classes are immutable (static fields in classes can be mutable, but that style of programming it is not considered as good one), and they are not JavaBeans-component instances: they cannot be created by default-constructor of their class (Class); they are created by some ClassLoader instances having their byte-codes as input.
If we want to cross the boundaries of the runtime type creation ideology of the JVM, we need to define some superstructure over the JVM that has its own notion of object type. Since we are going to have that superstructure as a kind of JavaBeans component model extension and to implement it using our JavaBeans components (in component-based manner), we name it as BeanVM. The type notion in BeanVM should allow different type implementations: both types created from JVM classes loaded by class loaders and types created by our composition procedure designed specifically for that purpose. It means that BeanVM types can be classified as hardcoded-types and composed-types; the former comes into BeanVM from loaded JVM classes, the latter is produced inside BeanVM itself by composition.

Some of the BeanVM types are components (in BeanVM perspective, no matter how they are implemented). For now, we define the types that can be instantiated without any information provided from outside (i.e. from their instantiation context) as components (and we'll try not to mix them with the component instances, as it often happens in JavaBeans related texts). That definition recalls JavaBeans-component definition for JVM that must be instantiable using its default-constructor.

Like JavaBeans-components, BeanVM components can have named property sets with typed property values (i.e. having property value types in terms of BeanVM types). All BeanVM types are instances of a Type type (like all Java classes are instances of the Class class). Type type is implemented in Java by (abstract) class Type providing all type related operations for BeanVM. All BeanVM types are implemented as immutable objects – the information they contain does not change after they have been created. We are intentionally following the class-based object-oriented principles and trying to retain them when preforming components composition (in contrast with JavaBeans component model, as it was mentioned above).

To access BeanVM functionality implemented in Java we provide the BeanVM API that we will discuss below when needed.

3.1. Type representation

Any BeanVM type is represented by an immutable instance of that type implementation java class that is inherited from the abstract class Type and exposes the following type information:

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

where typeName provides the name of the type, interfaceType and implementationType reflect the type interface and implementation, correspondingly.

We expect to deal with types that were not compiled and, therefore, have no type specific methods to be invoked by any method calling mechanism of JVM. The only ports that can be used to communicate with our BeanVM component instances are properties that are supported by special BeanVM API for component instance property access.

An instance of the InterfaceType class is a set of PropertyType objects describing the interface properties:

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

where popertyName is the name of a property. The valueType is the fixed BeanVM type that is used for type control when performing new value assignments: each BeanVM type can verify whether the given object belongs to its domain of values. The accessType defines operations to be applicable for the given property. A property can be readable (R), writable (W), bound (B). Indexed property can be indexed-readable (Ir) and/or, indexed-writable (Iw) as well. The defaultValue for a property is available for component properties only.

The implementationType part of the type information provides information on internal type implementation that is different for hardcoded and composed types.

Any object our BeanVM can deal with when it is functioning has its BeanVM type, and each BeanVM type can verify whether the given object belongs to the set of values of this type. That is used to implement value type control for property assignments. If an object was instantiated by BeanVM type, it knows that type upon creation. If an object was created by JVM class

instantiation and that class defines a component property value type, then that JVM class has been wrapped by the corresponding BeanVM type that delegates the object type check back to its implementations class. BeanVM supports BeanVM type instantiations and property access for their instances; all other activities are performed beyond the scope of its responsibilities (i.e. inside components behavior logic).

3.2. Hardcoded Types and Components

The hardcoded types in BeanVM are types implemented by loaded JVM classes. Hardcoded components are hardcoded types that are components. Here we consider how to represent JVM classes as BeanVM types and how to represent JavaBeans components as BeanVM components. The source information for hardcoded type definition in BeanVM is provided by corresponding class-object, loaded in JVM. We implement a primitive to map Java class to type by implementing the following method in Type type implementation class (as part of BeanVM API):

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

and we have our TypeLoader's hierarchy that mimics that of ClassLoaders (since each Java class is identified by its' name and the class-loader instance that the loaded class can provide). When implementing our TypeLoaders, we enhance the possibilities to create types in BeanVM: we can create types not only from loaded classes, as the primitive above does, but in some other ways described later.

We create a type for a given Java class once only, at the first attempt to get it; all subsequent calls will return existing hardcoded type from the type-loaders' (HashMap) table.

The hardcoded type creation procedure is based on reflection mechanism and standard JavaBeans introspection. The set of PropertyType objects is created based on an array of PropertyDescriptor objects that are provided by JavaBeans introspection procedure when introspecting the class (JavaBeans introspection can deal with any Java class, not only with JavaBeans-components). The PropertyType object (see (2)) gets the propertyName extracted from PropertyDescriptor object according to the JavaBeans design patterns [3]. The valueType is a BeanVM type created for a class of the property value by means of Type.forClass()-primitive (3). The property value class and the information to define the property accessType are available in the PropertyDescriptor object as well.

The InterfaceType object with the PropertyType objects array inside can be obtained from any Java class, but we are interesting in JavaBeans-components classes and their property value classes (that we wrap by our types). BeanVM-types for other Java classes are out of interest for the BeanVM.

The ImplementationType object for the hardcoded type is just a wrapper of the source class that implements the type in BeanVM (it reflects the old idea that any problem in Software Engineering can be solved by additional indirection).

The defaultValue in the propertyType (2) is needed in interface type for components only, and it is obtained only from our JavaBeans-components having our specific implementation. For all other (third party) JavaBeans-components we provide our hardcoded component-adaptor (that wraps any extraneous JavaBeans-component to be used in BeanVM environment).

Our hardcoded components are JavaBeans components having their implementation class inherited from our Bean class (directly or indirectly). The Bean class provides BeanVM API to

the internal implementation of the Bean component instance and its properties. All our hardcoded components are JavaBeans components that have BeanVM type instance implementation wrapped inside. The property access methods of our JavaBeans component implementation use BeanVM API and delegate to the wrapped instance implementation.

Here is a code snippet of the Bean class:

```
 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)
{…}
      // …
 }
```

The BeanVMObject is an abstract class that forces all BeanVM objects classes to provide their type getter method. Bean class has two constructors: public default constructor and package private constructor with a type argument. The default constructor, that is unavoidably executed when instantiating any Bean class ancestor (any our JavaBeans component), passes this component instance to its implementation factory method - Type.implementBean(this), that returns the instance implementation. The factory method, first, gets the type for this class and, second, lets the type to create the instance implementation. Hence, any Bean class ancestor it its constructor context is ensured that its implementation instance is already created and all its properties are implemented in it as appropriate. The concrete ancestor component is able to initialize its properties with their initial values using initPropertyValue() – method, and use the property value access methods (setPropertyvalue(), getPropertyValue(), etc.), that all delegate to the implementation instance.

Note, that initPropertyValue()-method works only once for a given property during the component type creation, when the component is instantiated for the very first time to collect property default values only (and store them in PropertyType objects (2)). When hardcoded type is created, the initPropertyValue()-method for its instances has no effect: they are already initialized with their default values when they are created in the implementation instance. Hardcoded component instance implementation contains only its mutable properties; immutable property values are stored in the PropertyType objects and shared by all instances of the type. The mutability of the property is determined by its accesType: the property is mutable if it is writable or bound (i.e. can change its value externally or internally, notifying about that).

Internal BeanVM API implementations of the property value access methods control the property accessType. All setPropertyValue()-methods control the property value type. All that control is implemented dynamically, and RuntimeException is thrown when violation occurs. To speed up the property access we provide methods in BeanVM API that translate the propertyName into an index of its PropertyType object in the component interface type along with the variants of property access methods that use the index instead of the propertyName. The concrete hardcoded component implementation can get its property indices in static initializer of its implementation class and use them when implementing its property access methods. In JVM perspective, all property values are stored internally in the BeanVM memory cells allocated for them and declared using the root of Java types hierarchy – java.lang.Object. That requires the explicit cast

to the property value type to be added when coding the concrete property getter using our getPropertyValue()-methods that return Object (some overhead with Java primitive types could be minimized by providing specific BeanAPI methods for them, but we omit these details here).

Note that our hardcoded components are JavaBeans components that can be manipulated visually in a builder tool (by definition) - like the BeanBox. It means that we keep all advantages of the JavaBeans component model, and we can use our hardcoded components, e.g. like we did before when implementing our 3D modeling framework by means of JavaBeans [5]. We could build a composite model from JavaBeans component instances and observe the model behavior, having placed it inside some predefined component container instance. But without generating byte-codes and loading them into the JVM, we could not create new composed component and place it into the ToolBox of the BeanBox (in visual terms), while all the components we used to create it are already there.

Now our goal is to make it possible: we have to deal with composed types and extend the builder tool (like the BeanBox), accordingly, to deal with them as well.

3.3. Composed Components

We have to define composed type in our BeanVM so that it does not correspond to some java class created and loaded especially for it, and can be defined dynamically, at runtime. The composed type that is instantiable without any arguments provided for it is our composed component. It has its interface definition and internal (hidden) implementation that we have to define using other components as the building blocks.

When defining composed types we comply with the class-based object-oriented approach. Our type definition is immutable structure that is able to create instances of that type instead of cloning them.

Like any type in BeanVM, composed type has its type name, the interfaceType and the implementationType (1). The interface type for the composed type is represented the same way as for a hardcoded type and is described by the array of PropertyType objects (2). But the implementationType is entirely different. It is composed by its composing types.

External communication with a composed component instance is performed using its interface properties access by means of BeanVM API mentioned above: any instance of a composed component is implemented as our Bean class ancestor instance that gets its composed type as an argument of the package private constructor (see Bean class code snippet) called when the composed type instantiation is performed. So, public BeanVM API for property access support works equally for any instance of any BeanVM type (no matter whether it is a hardcoded type or a composed type).

When creating an instance of a composed type, we create the instance internal implementation that, in this case, includes not only the mutable properties cells, but the implementationType instance. That implementationType instance is created as the result of composing graph traversal procedure where composing types are used as context-dependent refinements of composing components that are met during the traversal and instantiated to provide the composed type instance implementation.

The composed type interface defines the set of properties for the type instances. For each instance of the composed type, the instance internal implementation should be able to communicate with the instance external environment though the interface properties. The composed type instance

behavior is implemented by its composing components instances that express their behavior by their property value changes – as any our component instance does. Hence, to link the interface with internal implementation of the composed type instance, we need to link some interface properties to some properties of internal composing components instances. That composed type interface and implementation properties link can be organized basically in two different ways: by bidirectional property bindings and by interface property sharing. The second way is less expensive from the efficiency point of view, since there is no need to transfer property values.

Since we are supporting class-based object-oriented approach, we delegate our BeanAPI operations to the corresponding type object that implements them using the concrete instance reference (an internal instance implementation). When implementing an interface property access, we delegate to the interface property type that knows how to find and access the property value having the internal instance implementation. To implement that, we have PropertyType class sub-classed according property implementation categories: Immutable, Mutable, Bound and External. Each category (the PropertyType subclass) is responsible for corresponding property implementation and access. All mutable (and bound) property values are stored in an array of objects that is allocated by internal instance factory. All immutable property values are stored in their types and shared by design. PropertyType.External is used to share the enclosing type interface property with the given one. The PropertyType.External delegates the property access to the given property of the enclosing instance of the composed type. That enclosing instance is known as the composing type instantiation context.

We have mentioned previously that our components are instantiated without passing any arguments for their default constructor, and that was stated in the JavaBeans component definition. All our hardcoded components are JavaBeans components that are instantiated by their default constructors (without passing any context-related information). We do not use the BeanContext-related extensions of the JavaBeans Specification as well. While JVM provides some tricky way to get some information about constructor calling context using invocation stack trace, we do not rely on its usage. We use the fact that we are working in BeanVM (instead of JVM), that implements its own component instantiation primitive: createInstance(), that calls Java default constructors internally, and that default constructor delegates the internal instance implementation back to the BeanVM implemented instance factory method (see Bean class code snippet above). That factory method is able to use BeanVM primitive invocation stack to get the instantiation context (if it exists) and pass it to the internal instance implementation.

All the information needed to implement and access composed component instances is contained in their composed types implemented as immutable data structures that expose property value getters for reflection purposes (as implemented by abstract Type class and its type-specific subclasses). Now we'll describe how that structure can be created dynamically.

3.4. Prototype to Type Transformation

The natural way to create an instance of immutable data structure is to use its builder object that is mutable and can collect all relevant information to be provided for immutable objects initialization (i.e. to use the Builder design pattern) [15].

Naturally, we create the builder object in component-based manner. We use our specific hardcoded components to compose a prototype object that can be used as a source for creating an immutable composed type. These specific hardcoded components can be considered as meta-components (since they are components to construct components).

The prototype object is an instance of the Prototype component having the bound properties “name”,”interfacePrototype” and “implementationPrototype”. The type of property “name” is String. The “interfacePrototype” property can refer to an instance of InterfacePrototype component that is used to build the interfacePrototype object. The “implementationPrototype” property refers to an instance of ImplementationPrototype component to compose the implementation object.

The prototype to type transformation is performed by BeanVM primitive (here, for short, we omit the details concerning nested types that are enclosed in outer type implementation):

public static Type createFromPrototype(Prototype prototype), (4)

that returns new type in case when nothing prevents that from being happened (i.e. there are no namespace conflicts and other validation faults).

Generally speaking, we can create BeanVM type just having set the name only (i.e. having empty interface and empty implementation), that is similar to an empty JVM class (e.g. class Classname{}). Type with some interface part present, but with an empty implementation part can be used to instantiate its property set without any internal behavior linked with them (like a structure or record). Type with no interface part represents a separate executable entity type (e.g., “a scene” with its separate behavior).

The instance of the InterfacePrototype component exposes the set of property prototypes. The property prototypes serve two goals: 1) they are used as exposed handles to control the prototype implementation behavior, and 2) they provide source information to create the property types during the prototype to type transformation.

The instance of the ImplementationPrototype component exposes the set of component instances that compose the prototype object implementation – the set of composing prototypes. The compound prototype object can be built by defining graphs of two kinds: reference graph and events graph. The reference graph is defined by using some object as a property value (or as an element of indexed property value) of another. When an object is used as a property value of (i.e., is referenced by) several other objects, it is shared by them. The events graph is defined by component instances that bind the source bound property to the target to propagate the property change events (in correspondence with JavaBeans design patterns).

One of the main principal issues when defining composed components is defining a way to separate the interface of the component from its implementation while providing the way for them to intercommunicate. We define the interface in terms of properties that are prototyped using typed variables, i.e. objects having “value’-property with a given value type. We can use any BeanVM type as a value type for our typed variables (and as property value type after the prototype to type transformation). In fact, that is one of the two existing use cases for BeanVM types; another is BeanVM type instantiation. Having a type to be used as a value type, BeanVM creates (synthesizes) the BeanVM type for the typed variable automatically, assigning the synthesized name for that synthetic type and granting the full access rights to operate with its instances (i.e. read, write, bind their property “value”). In case the value type is an array type, we create BeanVM type for indexed typed variables that will serve as indexed property prototypes (having indexed access provided). That approach is similar with JVM array classes’ support (based on a given class of elements).

Property prototypes, implemented using typed variables and exposed through the interface prototype, should be linked with property prototypes of some composing prototypes inside the

prototype implementation. It can be done either by means of event routing graph (using bidirectional event routing for each link), or by sharing the property prototypes by means of reference graph. We use the latter approach (the more effective one) and provide support for the property prototypes sharing.

When a component is instantiated inside a prototype container, it gets its prototype-oriented internal implementation from the instance implementation factory. We use that instance as a composing prototype. The prototype-oriented implementation works in prototype-based style: it does not delegate the property access methods to the type of the instance (since it does not exist yet), but implements them using the prototype-oriented instance implementation itself. In that internal implementation, a composing prototype instance is represented with an array of property prototypes (in contrast with an array of property values, as it is implemented for component instances created outside the prototype container context). That additional indirection supports sharing the property prototypes of the interface prototype by composing prototypes instances in the prototype implementation (in case the value types are compatible). In principle, that sharing is similar to reusing the component instances inside the reference graph of the prototype implementation.

Each property prototype is supplied with access prototype instance that can be used to narrow the property prototype access rights by denying some of them for the given usage context. After the prototype to type transformation, narrowed access rights will be stored as accessType in the correspondent property type, as was mentioned in (2).

The composed prototype object is tunable and operational. Its' composition can be performed by visual manipulations with the correspondent tools support. When it is done, it can be used to produce the immutable BeanVM type. During the prototype to type transformation all prototypes are used as sources to create the corresponding types: property prototypes are transformed into property types with their access types created from the access prototypes, interface prototype is transformed into the interface type, composing prototypes are transformed into composing types, altogether transformed into the implementation type, and finally resulting in the composed type – or component - produced.

That newly created composed component can be instantiated as any other component –either using more efficient class-based object-oriented internal implementation, or using the more flexible prototype-oriented internal implementation (having been instantiated as a composing prototype in the prototype container).

Composed component can be serialized/deserialized (e.g., using some text format). To read the composed type from a text file by TypeLoader, we provide Type.forName(String typeName) primitive that loads the type by its name like JVM Class.forName(String classsName) loads classes. When looking for the source of type by name, we first try to load hardcoded type with the given name (using Class.forName()), if it exists, then the serialized type file to be parsed. The parser, essentially, reads into a prototype object and performs the prototype to type transformation.

4. SAMPLE APPLICATIONS

Both VRML [16] and X3D [17] Standards define a sets of elements that constitute a scene to be depicted using 3D and/or 2D graphics. The scene model in memory is represented by the directed acyclic graph (DAG), consisting of node instances whose types are predefined according to that standards. Node instances contain 'fields' whose values, in common, define the (scene) model state, and can vary, in event-based manner, while event propagations are performed among them.

For all the base predefined nodes, their field value types are statically known and defined by the Standards. Directed acyclic graph can be traversed with the result of earning some context-specific information, which, in combination with nodes field values, defines the visual presentation of the model that is rendered. The model behavior is defined by changes that happen in the node graph during events propagation, with events carrying data on their field value change from node source to the node target, and by reactions in receiving nodes, that, in their turn, can change their state and fire events. VRML and X3D Standards define base node types with their field types (and their meanings/semantics), the constructs to define event routing graph, standard scene access interfaces and so called 'profiles' – sets of independently distributable standardized functional components to support various (extendable) modeling abilities.

While the Standards do not expect their Java-implementation (and up to date they were not implemented in Java properly, despite of some attempts), their implementation using JavaBeans component model appears to be pretty natural; moreover, using JavaBeans-component model we can avoid some limitations of the Standards, that were not initially designed to use neither dynamic abilities of the Java platform nor JavaBeans-component model for it. JavaBeans components usage to implement subsets of the Standards with some additional features were discussed in [5], that shows how parallel compositions, behaviors and 3D-visualization of the models can be organized in standard JavaBeans-container (using BeanBox) along with standard JavaBeans-components.

Using the presented component model, we can implement user-defined nodes definitions that are described by the PROTO construction of VRML and X3D, in class-based object-oriented manner. It was not possible using JavaBeans component model, as we have discussed above.

The PROTO construction in VRML (or X3D) essentially provides the language features to describe new, user-defined node type, having supplied its name, its interface in terms of its fields (as that is done for the base, predefined node types) and internal implementation, that is similar to a scene graph. Both scene graph and PROTO implementation graph can contain nodes of any type – both predefined and user defined – provided they are defined prior to their usage in a graph by means of PROTO construction. Fields from the interface of a PROTO are bound to fields of its internal implementation nodes using special VRML syntax construction – "IS". (In our model that separate construction is not needed – it is essentially the same as "USE" construction to share interface property).

In practice the PROTO construction is implemented as a cloneable prototype, as its name implies, or just as macros definition to be substituted by parser. The whole scene description is not considered as a type definition to be instantiated; instead, it is just a prototype instance serialized in VRML or XML formats.

Using the proposed component model we expect to simplify and optimize software systems for the subject areas, where declarative languages (like VRML or X3D) are used to describe 3D-models with event-driven behaviors, and we will be able to support new abilities of these systems to benefit from.

The ability to build a flexible prototype object from components that can be transformed into composed instantiable component with context-dependent optimization may be useful in various applications. For example, that ability directly corresponds to the wireless sensor network commissioning task [6]. These networks can be described using a graph of nodes communicating by radio in event-driven manner, and are built from standard components to be tuned for a given application and environment. The concrete ad hoc network prototype (or a model) can be designed, then that typical network component can be created, and its instances, containing concrete network nodes settings, can be used for commissioning (e.g., over the air).

5. RELATED WORK

Component-oriented programming and its usage in software development formed a component-based software engineering – a branch of software engineering that has its long term history, valuable results and issues to be solved. There are many publications attempting to provide definitions for a component (we have referred to three most frequently cited [8, 9, 10]), investigating different component models with their taxonomies [1], and considering various aspects of component-based software development for different application areas [8, 10, 11]. Detailed overview of that works is beyond the scope of this paper.

We are concerned with issues we know from experience earned while developing component-based software system to implement 3D modeling and visualization [5]. This is a large area as well, and we will narrow our scope by platform in use, since there are popular platform dependent component models we have to leave aside, e.g. [18].

The most widely known 3D modeling software for Java platform is probably Java 3D [19] library, initially developed by Sun, then by the open source community. That heavy-weight library was not designed with component-oriented approach in mind, and attempts to implement VRML browser using Java 3D library were not finished. The library was used in X3D Standard development by web3d.org community [20], but currently Java 3D library is deprecated.

There are several 3D modeling and visualization tools for Java platform developed by commercial companies and universities, e.g. [21, 22], but they do not explicitly use a component-based approach or use their proprietary component model [23].

The PtolemyII project [24], having its long pre-Java history and covering, among others, the application area of our interest, has been moved to Java platform and use its component model reflected in Moml [25] specification. As far as it can be seen in publications, instantiation of declaratively defined composed type in implemented by cloning (i.e. in prototype-based manner).

At the time of highest VRML popularity, there were publications on extending it in object-oriented manner [26, 27]. Now we observe new wave of interest to these declarative languages that can be seen in new VRML/X3D compliant product – Instant Reality [28]. But that product does not extend the standards in component-oriented direction.

Both VRML [16] and X3D [17] standards specify Java authoring interfaces (for external model access and for internal scripting in Java). All that specifications have no relation with component models for Java-platform, while that functionality could be readily provided by component-oriented design.

There are some works on component-based software evolution [29, 30], but they are based on code generation. We can consider code generation as a feasible way to translate BeanVM components into JavaBeans components (that is similar to the Java just-in-time compiler translating from virtual machine level to executing machine level).

6. CONCLUSION

Both hardware and software architecture histories demonstrate the evolution with increasing dynamics abilities. Java platform, as the most popular software development environment, and JavaBeans component model, the most popular for that platform, provide the means to evolve in that direction as well. In this paper we have proposed a component model that demonstrates prototype-based and class-based programming paradigms interrelations. The prototype is a

flexible, mutable object sample that is built, lives and evolves in component-based manner; when it gets some desirable state of evolution, or just eventually by some stimulus, its genetic code is extracted into the type and saved for reuse by next generations (that reminds an everyman untaught view on genetics in the natural reality).

The component model proposed is a kind of dynamic extension for the JavaBeans component model, with the extension supported by specific JaveBeans components. Correspondingly, as for JavaBeans component model, we need the extended builder tool that utilize and demonstrate extended abilities of the component model. Developing that tool is the goal of our future work.

REFERENCES

[1] 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.
[2] Tiobe community index. www.tiobe.com
[3] Sun Microsystems. JavaBeans API Specification. 1996.
[4] Open Services Gateway initiative – OSGi Alliance. http://www.osgi.org/
[5] E.Grinkrug. “3D Modeling by means of JavaBeans”, Proceedings of the 12th international workshop on computer science and information technologies, CSIT’2010, Moscow – Saint-Petersburg, Russia, 2010.
[6] E.Grinkrug, “Software Component Models in Wireless Sensor Network specification, deployment and control”, ZigBee Alliance, 1st European ZigBee Developers’ Conference, 2007, Munich, Germany.
[7] Douglas McIlroy. Mass-produced software components in: P. Naur and B. Randell, "Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968", Scientific Affairs Division, NATO, Brussels, 1969, 138-155. http://cm.bell-labs.com/cm/cs/who/doug/components.txt
[8] C.Szyperski, D.Gruntz, and S.Murer. Component Software:Beyond Object-Oriented Programming. Addison-Wesley, second edition, 2002.
[9] B. Meyer. The grand challenge of trusted components. In Proc. ICSE 2003, pages 660–667. IEEE, 2003.
[10] G. Heineman and W. Councill, editors. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.
[11] Christiansson, B., Jakobsson, L., Crnkovic, I.: CBD process. In: Crnkovic, I., Larsson, M.(eds.) Building Reliable Component-Based Software Systems, pp. 89–113. Artech House (2002).
[12] Lau, K.-K., Wang, Z., Software component models. IEEE Transactions on Software Engineering 33 (10), 2007, pp.709–724.
[13] The Bean Development Kit (BDK 1.0, 1.1). http://bhiggs.x10hosting.com/Courses/HighOctaneJava/JavaBeans/bdk.htm
[14] Xaml Object Mapping Specification. http://www.microsoft.com/en-us/download/details.aspx?id=19600
[15] J.Bloch. Effective Java. 2nd Edition, 2008.
[16] The Virtual Reality Modeling Language. ISO/IEC 14772. www.web3d.org
[17] Extensible 3D (X3D). ISO/IEC 19775. www.web3d.org
[18] Eddon, G., Eddon, H., Inside COM+ Base Services, Redmond, WA: Microsoft Press, 2000.
[19] Java3D. http://java3d.java.net/
[20] Xj3D. http://www.xj3d.org/
[21] jReality. http://www3.math.tu-berlin.de/jreality/
[22] jMonkeyEngine. http://jmonkeyengine.com/
[23] Demicron WireFusion. http://www.demicron.com/index.html
[24] The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/index.html
[25] E.A.Lee, S.Neuendorffer, Technical Memorandum UCB/ERL M00/12, Dept. EECS, University of California Berkeley, CA 94720, USA. http://ptolemy.eecs.berkeley.edu/publications/papers/00/moml/
[26] Curtis Beeson. An Object-Oriented Approach to VRML Development. VRML '97 Proceedings of the second symposium on Virtual reality modeling language, p.17-24.

[27] Stephan Diehl, VRML++: A Language for Object-Oriented Virtual-RealityModels. Proceedings of the 24th International Conferenceon Technology of Object-Oriented Languages and Systems TOOLS Asia, Bejing, 1997, p. 141 – 150.
[28] Instant Reality. http://www.instantreality.org/
[29] A.McVeigh, A Rigorous, Architectural Approach to Extensible Applications, PhD thesis, Imperial College London, Department of Computing, 2009.
[30] A.McVeigh, Creating, Reusing and Executing Components in Evolve, 2010. http://www.intrinsarc.com/evolve