Component-Based Software Engineering and Runtime Type Definition

A. R. Shakurov
Business Informatics Department,
Higher School of Economics,
Moscow, Russia
amir-shak@yandex.ru

***Abstract*—The component-based approach to software engineering, its current implementations and their limitations are discussed. A new extended architecture for such systems is presented. Its main architectural concepts and principles are considered.**

***Index Terms*—Runtime environment, software architecture, software engineering, software reusability**

I. Introduction

SOFTWARE ENGINEERS have always pinned their hopes on the idea of **reusable code** [8]. In their urge towards eliminating code duplication, simplifying code maintenance, making it less error-prone and streamlining the development process programmers have gone a long way from a completely unstructured code through procedures and program libraries to object-oriented technologies and application frameworks. Taking the code reuse idea one step further, the component-based software engineering [9], [14] is a very promising approach to software development.

The term **"component instance"** usually refers to a program entity holding data and offering some functionality that are hidden by a well-defined interface (cf. [10], [11]). The concept of interface, however, varies from one technology to another: a number of "properties" (attributes, member variables etc.) [13], a number of member functions/methods (as in most object-oriented programming languages) or even an entity whose nature may be left out of scope of the component technology itself [1]. The process of combining components into a working system is considered to be relatively simple (e.g. [2]). Nevertheless, this process is implemented differently in various technologies.

In order to propose an optimal way for organizing component interaction, that would introduce a good balance between flexibility and ease of use, let us concisely consider main aspects of component-based models and technologies and their limitations (detailed comparative analysis is beyond the scope of this work, see in [12] and [14]).

II. Limitations of component-based technologies

Despite the many advantages of the component approach its currently existing implementations have a number of substantial limitations. The most difficult goal to achieve here is probably to find a way of designing components that will provide the necessary functionality without exceeding it. The necessary functionality is determined by requirements, and these are bound to change with the lapse of time. There're three options to consider. First of all, it is possible to introduce software with somewhat wider capabilities, so that it would still be adequate when requirements change. This approach, however, demands remarkable architectural design skills and foresight and has the risk of bloating the program under development making it unsuitable for system with limited memory resources. Second of all, one can adapt the software with the course of time. This would result in the most appropriate as well as the most expensive software. The third option is to introduce a component framework that would allow in-place modification of component's functionality without stepping over the bounds of the component model.

Let as consider an example. ZigBee specification [15] offers a suite of high level communication protocols for low-rate, low-cost, low-power-consumption wireless personal-area networks. It is implemented, for instance, by communicational parts of microelectromechanical systems in wireless sensor networks. At the same time, the specification is subject to frequent changes. This makes manufacturers renew and release sensor firmware, which is a difficult process due to the lack of high-level development tools for such systems.

ZigBee specification introduces network and application layers (in addition to the PHY and MAC layers defined by the IEEE standard 802.15.4 [5]) to the protocol stack. The topmost application layer (which is subject to frequent changes) is comprised of a number of components: ZigBee device objects, their management procedures, application objects. Requirements to these components are not changed at the same time (application objects, for example, are provided by the manufacturer and thus aren't managed by the ZigBee alliance). Nevertheless, every new release of the specification implies a new release of the whole firmware. It would be more cost-efficient to modify only those components specification for that have changed and make the system reconfigure to make use of their instances without rewriting a binary firmware image to devices' memory.

To add, remove or modify certain functionality of a component's instance means to introduce a new component, because it is the component that defines functionality of its instances. That is, we're essentially dealing with the task of defining **a new type of data.** This can be done either by deducing new type from existing one or creating it from scratch using a number of predefined low-level components. We believe it's this operation that must be available at runtime in order to ensure software flexibility.

This problem can be tackled in a number of bypass ways such as source code, bytecode or binary code generation/transformation, runtime compiler calls or taking advantage of programming language's ability to modify its low-level runtime data structures (Java Reflection being a graphic example). These, however, aren't always an option. Reflection mechanisms are primarily meant to be used to build IDEs, source code analysis tools and GUI designer applications. Using them indiscriminately to build any garden-variety software lays exceedingly high claim to developer and is therefore error-prone. Furthermore, many programming languages don't support reflection at all. And embedded systems often lack compilers and code modification frameworks. Therefore an alternative system capable of building new types from user-configured instances without stepping over the bounds of its model is needed.

To conclude the section, let us look at another simple example. We're designing a GUI application and we want to change a button's label (the button has already its functionality attached to the application). The change is supposed to be permanent: the button won't be renamed at runtime. Although obvious, the necessary procedure is implemented in various frameworks with a distressing flaw: the variable property is left variable for the lifetime of the component's instance regardless of developer's intentions. In other words, the fact that the button's text isn't meant to be mutable at runtime (should this be the designer's intention) cannot be expressed by a developer.

The designtime work with an instance of a component and the runtime work with it are basically two different contexts of its use. However, these cannot be fully separated by existing frameworks (see [13] e.g.), because one has to start (execute) a component (i.e. instantiate it) in order to either configure it or take advantage of its functionality. This is the source of the troubles one encounters when adjusting a component's instance.

To summarize, a new component system capable of introducing new data types by either modifying existing ones or creating them from scratch using a number of predefined low-level components is needed in order to create flexible (adaptable to changing requirements and contexts of use) software for systems with limited resources.

III. Core principles

To achieve the goal of designing such a system, we have analyzed advantages and limitations of existing object-oriented programming languages and component technologies. This has allowed us to infer the core principles of the suggested component model described in the next section.

The model is based on a general object-oriented idea, and is extended by other technologies' traits when necessary since the need to define data types at runtime makes designtime and runtime essentially indistinguishable. Omitting a detailed consideration of specific technologies here (see [12] and [14] for such review), we are going to summarize the core principles that underlie the model in question.

The two primary characteristics of any development and execution environment are the principles and mechanisms of data organization and control flow management. Looking for those characteristics in the object-oriented paradigm, one will find the **hierarchical data organization** principle and **the concept of method** as the means of control flow management. And while the first characteristic gives us a well-balanced solution for managing ever growing complexity of software systems, we find the second one to be too complex and cumbersome. We believe that the very concept of object method (see [3] e.g.) isn't suitable for adopting it in a component model, because of its overwhelming versatility: a method can have variable number of parameters and (in certain languages) return values, or it can have none of those; arguments can be passed by either value or reference; methods can be overloaded and overridden (in which case a complicated set of resolution rules takes the stage). The list can be continued. The concept of method, therefore, doesn't provide intended ease of use (though it does handle software complexity well).

In contrast, the concept of **property** introduced in some frameworks (C#, JavaBeans) is more suitable to our needs. The property-based interaction model is simple and formal because of the limited number of aspects describing the "property" concept: its type and applicable operations (usually reading and writing). However, to become a perfect rival (to efficiently implement callback routine, for example) the concept needs to be extended with the binding operation (described in following section).

To summarize, we have adopted the principle of hierarchical data organization and the property concept as the means of organizing execution flow to create a component model with a runtime data type definition capability. We will now proceed to describe the model.

IV. The model

The model we're going to discuss has its prototype implemented in the Java programming language. Therefore let us start with describing the model from the user's point of view.

While working with the application, a user interacts with three categories of objects, viz. components, components' instances and containers (which are used as both runtime and new type definition environments).

A. *Components and instances*

An instance of a component is an aggregate of data and behavior hidden behind an **interface**, the latter being the only way to interact with this kind of entity. We shall refer to the hidden part of an instance as its **implementation**, as opposed to the interface.

Like objects in object-oriented paradigm have classes, instances have components that describe the way these kinds of instances are created and function. Every instance has a single component associated with it and this association is immutable during the lifetime of an instance. Every component can be instantiated without providing any additional information. This means that primitive types of data (numerical, boolean, string etc.) aren't actually components. These are usually called value-types and to instantiate them one has to provide at least a value of the instance to create.

In addition to contextless instantiation, components allow instantiation in a certain context (e.g. as part of a composite instance, see below).

Interface

Let us focus on an interface of a component's instance. It is comprised of a number of properties each of which is characterized by:

- name (used to identify a property),
- value type,
- access permissions; any property may be accessible for reading, writing and binding.

Read/write operations need no explanation. **Binding** *S* property of a particular instance to *D* property of another instance ensures that whenever the value of *S* is changed the new value will be written to *B*. The binding operation, as it was mentioned earlier, is needed to efficiently implement callback routine and is similar to that of the Java Beans model, except that there's only one event type (property value change event) whereas in Java Beans a property can have multiple event types associated with it.

B. *Container*

A **container** is an execution environment that allows the following operations to be performed in its context ("within it"):

- instantiate any components,
- change values of properties of instances created at the previous step,
- bind instances' properties to each other.

Apart from that, a container can be used to create new components. Its contents are considered to be a prototype of an implementation part of a future component. To create a new component, one has to complete this information with interface specification and connect these two parts together. Above that, a user is able to edit metadata of instances constituting future implementation. All this can be achieved with the following operations:

- restricting access to instances' properties,
- adding properties (with specified metadata: name, access permissions etc) to the interface part of the component,
- adding **sharing connections** between a property of an instance that is a part of the implementation and a property of the interface.

The "sharing connection" between two instances' properties makes those instances share memory cell (therefore changing the value of one property is immediately reflected on the value of the other property). This behavior requires a custom memory model, which we'll focus on later.

Once the user has provided all the necessary data to a container, he can create a new component that will correspond to the prototype currently in the container in the sense that all of its future instances shall have the same structure.

While a user is working with a container, the latter gathers all the necessary information about future component. It is to be noted that some of this information can be deduced from the runtime structure of instances the user have created. For example, bindings between instances' properties are analyzed only when a new type is being created. Other pieces of information, however, can't be stored within the runtime structure. For example, restricting access to a property of an instance won't actually modify that instance because that implies changing metadata (i.e. data type) of an existing instance and there's no sense in doing that. In other words, not every user action can be directly reflected on the runtime structure of instances, and it's container who makes the process of editing both data and metadata transparent to a user.

It wouldn't be possible to define new types at runtime, however, if it wasn't for a specially designed internal structure of components, which will be discussed in the following section.

C. *Internal structure*

Let us focus on the internal architecture of the prototype application we've developed that provides the previously described functionality. As it was mentioned above, the application is implemented Java, but it can be easily rewritten in any other strongly-typed object-oriented programming language.

All the instances in the system implement common **Instance** interface that provides methods to access instance's type and properties. Similarly, all the types implement the **Type** interface that provides methods to instantiate the type and also extends the Instance interface. Therefore, any type (and any component which is a kind of type) is an instance whose data are metadata describing its future instances. There is also a **TypeType** type, which is a type of any type (including itself).

Let us consider the following scenario. A user defines a new component with no bindable properties. Than the whole event–listener infrastructure (that supports bindings' functionality) becomes redundant and should not be included in corresponding instances. This kind of deep context adjustment is crucial when dealing with runtime type definition and we pay great attention to it.

To implement the smart adjustment described, we've introduced indirect access to instances' properties. They're accessed via special **PropertyGetter**, **PropertySetter** and **PropertyBinder** objects. If there're no bindable properties in a component then its instances end up having their PropertyBinder object uninitialized. And if there's at least one bindable property, then the binder object will be created (but it still won't be granting access to unbindable properties, of course).

Memory model

As it was mentioned above, the custom memory model is required in order to consistently handle binding and sharing connections between properties.

The memory model introduces traditional kinds of **"memory cells"** (viz. constants and variables) that store instances as their values. The value can be read and (in case of a variable) written. The memory cell can also have no value (it is said to be null in this case). Finally, memory cells are strongly typed, which means that every cell knows its type and an attempt to store an instance of mismatching type in it produces an error.

There's also an unconventional kind of variable – listenable variables. As the name suggests, listenable variables (in addition to having all the features of regular variables) allow special objects (called listeners) to subscribe to the variable value change event.

Data types

We've already mentioned that the system supports primitive value-types which are a kind of instance types. Another kind of instance types is components, but there're two different kinds of them. The first one is **compiled components**. These are components whose implementation is not analyzed by the system in any way. This permits the system to handle various components implemented using a third-party means (e.g. java bean components).

The second kind of components is **composite components.** These are components implemented by means of the system itself. The implementation part of a composite component is a structure of other components.

Since it is a composite component that is built whenever user defines a new type, this kind of components is of greatest interest.

Composite components

We are going to focus on the structure of metadata stored in composite components. Since these metadata determine the structure of data in corresponding instances, we will discuss that structure incidentally.

Composite type (like any other type) describes interface and implementation parts of its instances (Figure 1). These parts are interconnected via mechanism described below.

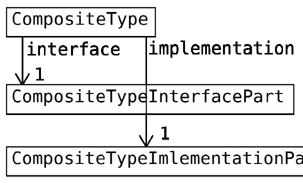

Figure 1. Composite type.

An interface of an instance is a set of its properties, so the metadata stored in the interface part of a component are a set of property descriptors (Figure 2) each of which specifies:

- property value type,
- access permissions,
- default value (optional).

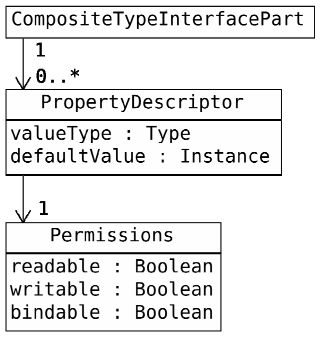

Figure 2. Interface part of a composite type.

Finally, implementation part of a composite type is metadata that describe a structure of instances with interconnected properties that will result from instantiation the type. These metadata are represented in the following way (Figure 3).

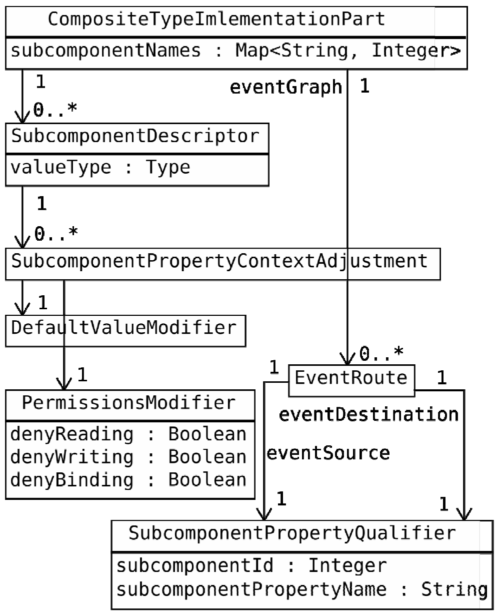

Figure 3. Implementation part of a composite type.

To every connection between two instances that constitute an implementation of the future component corresponds an object of the EventRoute class that holds two objects of the SubcomponentPropertyQualifier class (for the beginning and the ending of the connection). These objects simply specify a source and a destination of a property change event: they hold respective components' IDs and their property names.

For every instance that is a part of an implementation of the future component there's a descriptor, an object of class SubcomponentDescriptor. This object specifies:

- instance type,
- for every property of the instance, context adjustment of that property to its use as a part of another instance's implementation. This adjustment defines modified default value (if any) of the property (the DefaultValueModifier class) as well as restricts access to it (PermissionsModifier).

A default value of a property can be modified in a number of different ways.

- "Void" modification, the value is left intact.
- Explicitly specified value (the object of the DefaultValueModifier class holds this value).
- Value is another instance that's present in the same context (implementation of the same instance). In this case the modifier object holds the name of that instance.
- The property is shared. In this case not the value of the property, but the property itself is changed: instead of creating a new memory cell for storing the value, an existing (provided from elsewhere) cell is used by the instance.

The last option is used to provide an instance (via its instantiation context) with references to its parent instance's properties, thus creating sharing connections between these properties. These connections glue interface and implementation parts of a composite instance together.

Composite component instantiation

To demonstrate described structure at work, let us go through the process of composite type instantiation. The following algorithm implements the process.

1) Memory cells for storing values of properties are established (one for every property descriptor). If there's an instantiation context, a reference for an existing cell is used. Otherwise a new memory cell is created, its kind (constant, variable or listenable variable) being determined by respective access permissions, and is initialized with either default value (if any) or a new instance of the corresponding type.
2) Instances constituting the implementational part of the future component are created. This involves evaluating their instantiation contexts. Every property descriptor is merged with corresponding subcomponent property context adjustment object giving a new set of adjusted access permissions and default value.
3) Binding connections (i.e. event routes) are established.
4) Proxy objects for accessing new instance's properties are created (property getters, setters and binders). These objects receive references to relevant properties only (e.g. a setter object only holds references to writable properties) and if there aren't any, the object is not created at all.

After all the steps have taken place, a new object of class CompositeInstance is constructed. The constructor is provided with the type (an object of the CompositeType class) and proxy objects. This results in a new instance of the component.

The described internal structure of composite components ensures great flexibility. Since data types are defined by the (runtime) structure of regular (java) objects (as opposed to compiled binary or bytecode), it gives us an opportunity to easily manipulate that structure at runtime thus creating new types.

V. RESTRICTIONS AND FUTURE WORK DIRECTIONS

Let us discuss certain limitations to the described solution. First of all, flexibility comes at a price. The ability to reconfigure software results in overhead computational costs. This means that the proposed model should be adopted only when implementing systems that either have no severe restrictions on computational resources or do not require very high performance. Aforementioned microelectromechanical sensors offer a graphics example: while having limited memory capacity, they carry no considerable performance limitations (they usually stay idle for hours between sending a signal and going idle again). This is why the availability of remote reconfiguration means takes precedence over elevated performance here.

Second of all, while the property-based interaction is simple and quite flexible, it has its limitations, too. For example, implementing intricate algorithms this way is possible though burdensome, making a traditional imperative-scripting style far more suitable choice. In other words, the described solution should be adopted when there're a great number of objects (instances) with a relatively simple interaction. In addition, it's possible to incorporate complex logic via compiled components, though this still requires implementing it in a third-party language.

The described system being only a prototype, our first and foremost goal is to turn it into a complete, feature-reach production-quality platform. This requires both evolving the component model and improving development tools to make use of it. This also implies optimization to guarantee acceptable performance.

As for the practical applications, we're planning to make use of the platform in question to introduce software solutions for 2D (GUI development) and 3D (VRML [6], [7] implementation) design, reconfigurable wireless sensors firmware and possibly for some other purposes.

VI. Conclusion

We have described the main ideas and the core principles of internal organization of the new component architecture with extended capabilities. The ease of manipulating both data and metadata structure of software is not to be underestimated. We believe that formalized, simple yet powerful component model with runtime data type definition capability will allow creating most configurable software that will be able to evolve and adapt to changing requirements easily.

References

[1] Bruneton, E., Coupaye, T., Stefani, J.B., *The Fractal Component Model specification. Version 2.0-3*, The ObjectWeb Consortium, 2004.

[2] Costa Seco, J., Silva, R., Piriquito, M., "ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration", *Computer Science and Information System*, ComSIS Consortium, Novi Sad, Serbia, 2008, pp. 63-86.

[3] Gosling, J., Joy, B., Steele, G., *The Java™ Language Specification. 3rd ed.*, Addison Wesley, 2005.

[4] Heineman, G.T., Councill W.T. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 2001.

[5] IEEE Std 802.15.4-2003 – Wireless Medium Access Control (MAC) and Physical layer (PHY) for Low-Rate Wireless Personal Area Networks (WPANs).

[6] ISO/IEC 14772-1:1997 — Virtual Reality Modeling Language (VRML).

[7] ISO/IEC 14772-2:2004 — Virtual Reality Modeling Language (VRML).

[8] Krueger, C.W., "Software reuse", *ACM Comput. Surv. Vol. 2*, ACM, New York, 1992, pp. 131-183.

[9] McIlroy, M.D., "Mass produced software components", *Naur P., Randell B., "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, pp. 138-155.

[10] Object Management Group, *The Common Object Request Broker: Architecture and Specification. Version 3.1. Part 3 - Components*, OMG document formal/2008-01-08, 2008.

[11] Redmond, F.E., *DCOM: Microsoft Distributed Component Object Model*, IDG Books Worldwide, Inc., Foster City, 1997.

[12] Stiemerling, O., *Component-Based Tailorability*, Bonn University, Bonn, 2000.

[13] Sun Microsystems Inc. *The JavaBeans™ API specification. Version 1.01-A*, Sun Microsystems Inc., 1997.

[14] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, 2nd ed, Addison-Wesley Professional, Boston, 2002.

[15] ZigBee Alliance, *ZigBee Specification,* ZigBee Document 053474r17, 2007.