Component Architecture with Runtime Type Definition

E. M. Grinkrug, A. R. Shakurov

Abstract — **The component-based approach to software design and development is being focused on. By analyzing the main ideas of this approach, their currently existing implementations, their limitations and promising lines of development we suggest a new component architecture which extends capabilities of existing component technologies. The main principles for building such architecture are described.**

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

I. Introduction

IN the field of software development, the idea of **code reuse** has always been of utmost importance [6], reducing costs, minimizing errors, making the code maintainable. The first examples are program libraries, design patterns [4] and application frameworks. Object-oriented and generic programming [3] also contains this idea at its core.

But the most promising manifestation of the idea of code reuse is probably the component-based software engineering [7]. The term **"component"** is usually used to refer to a program entity that holds data and implements some functionality, which are hidden by a well-defined interface (cf. [8], [9]). A more formal definition is given at sec. IV.

The concept of interface varies from one technology to another. It may be a number of "properties" (or attributes, members etc.) [11] or an independent indivisible entity [1]. There're another options. Below an approach that we consider optimal is presented and justified.

Combining components into a working system is usually though of as a relatively simple procedure (e.g. [2]). The structure of the system, however, is manipulated differently in different technologies. Our goal here is to find the optimal way of organizing component interaction, introducing a good balance between flexibility and ease of use.

Manuscript received September 27, 2010.

E. M. Grinkrug is with the Business Informatics Department, Higher School of Economics, Moscow, Russia (e-mail: egrinkrug@hse.ru).

A. R. Shakurov is with the Business Informatics Department, Higher School of Economics, Moscow, Russia (e-mail: egrinkrug@hse.ru).

II. Limitations of component models

Despite the advantages of the component approach its current implementations have a number of substantial limitations. The core difficulty here is to introduce a component that provides a necessary functionality but doesn't exceed it, bloating the system under development.

One way to strike such compromise is to separate the designtime work with a component from its runtime use. The need for such distinction is realized and implemented to some extent (see [11] e.g.), but we believe more can be done here.

For example, let us suppose we're designing a GUI application and we want to change a text on a button (that already has its functionality somehow connected to the application). The change is considered to be permanent: the text won't be changed during the runtime. The procedure is quite obvious (one has only to set the needed value to the corresponding property of the button), but its implementations in various frameworks share a common flaw: the variable property is left variable for the lifetime of the component. Although it's known for a fact the label is a subject to change only during design time (and at runtime it's constant), we have no means of expressing that.

This problem can be treated in a different way. Difficulties in deep adjustment of a component to the context of its use (the design/runtime opposition is just an example) are rooted in the fact that essentially we're dealing with the need of defining **a new type** of data. And since any activities concerning a component implying the use of its functionality and therefore its execution, the process of defining a new type must take place at runtime (and without recompiling a program).

There are several bypass routes such as source code, bytecode or binary code generation, runtime compiler calls etc. This routes, however aren't always an option (in embedded systems, for example, the compiler is usually unavailable). That's why a system capable of building new types from user-configured components without stepping over the bounds of its component model is of great interest.

III. Data organization and control flow management

Let us concisely consider the main aspects of existing object-oriented programming languages and component technologies. Analysis of their advantages and limitations has defined the main features of the suggested component model described in the next section.

A comprehensive consideration of specific languages and technologies, however, is beyond the scope of this work (see [10] for such review). We base on a generalized object-oriented concept, engaging other technologies when it's necessary since the requirement of runtime data type definition leads to a component technology to which designtime and runtime aren't clearly distinguished.

Any development and execution environment can be looked at from the two points of view, viz. the principles and mechanisms of data organization and control flow management. From the first point of view, one of the greatest strengths of the object-oriented paradigm is the hierarchical data organization. It's a well-tried remedy for handling a growing complexity of software systems, so the suggested component model is made to be capable of combining components into interacting groups, constituting new components (strictly speaking, combined into groups are *types of components,* and only then composite components are created, see sec. IV).

From the managing control flow point of view, object-oriented approach provides us with methods. We believe the concept of method to be too flexible and complex, which overcomplicates the structure of the programs using it. A method may have a return value, or it may return nothing. The return value itself may be a reference to some internal class member, or it may be a defensive copy. A method is allowed to have an arbitrary number of parameters of arbitrary types. This list can be continued (consider overloading and overriding, for example). The concept of method is not specific enough to allow the desired formalization of object interaction.

The concept of property presented by some frameworks (C#, JavaBeans) is more apposite. It combines features of both object field (well-defined type and a fixed set of operations: reading, writing) and method (behind access operations a programmer-defined functionality can be hidden). The property-based component interaction model is simple and formal because of the limited number of characteristics of the "property" concept. However, properties (the way they are implemented in C#, for example) can't compete with methods in capabilities. If a property can only be accessed for read/write operations it's impossible to efficiently implement well-known callback mechanism. An operation of **binding** is needed (see below).

Following the two key principles we've pointed out (hierarchical organization of components and property-based interaction) is, in our view, necessary to create a viable component model. Along with the runtime type definition requirement, these principles form the basis of the suggested solution, which we will now describe.

IV. Model

Let us describe the component model that will allow one to keep the strengths on the existing technologies and models, while eliminating the drawbacks mentioned above.

A. Component

The main idea of component-based software development is the distinction between an interface and an implementation, which lets one to implement and use a component separately. By component we mean a "black box", i.e. data and functionality hidden behind an interface. Therefore component is described by its interface, implementation and state (data incorporated in it and changed during its lifetime).

The interface of the component is described by the set of its properties (named attributes with a given value type and access permissions). **The implementation** defines a behavior of the component and is different for **primitive, composite** and **compiled** components

Interface

The unit cell of the interface part of the component is its **property,** an entity with a specified type that represents some aspect or attribute of the component. Some of the three operations may be applicable to the property: reading, writing and binding. The applicability is governed by the access permissions (defined in the component type, subsec. IV.B).

The value of the property (when it can be read) is defined by the internal state of the component, which is changed by the property writing operations. The operation of binding property A to property B (A and B may be owned by different components) allows the owner of B to receive notifications of change of A's value in form of the new value being written to B.

Implementation

Sticking to the idea of hierarchical organization of components, we distinguish three kinds of them: **primitive, composite** and **compiled.**

Primitive components are similar to variables of primitive types in programming languages. They hold data of most common types (numbers, characters, text string, logical values etc.), are indivisible from the model's point of view and don't have any properties. Primitive components are so-called value-variables, whose values are given at initialization and are immutable. The prime characteristic of such component is the distinction of its identity.

Compiled components are implemented with use of offsite means (i.e. outside the component model). Having this kind of components around is required to integrate thirds-party technologies (e.g. JavaBeans). Both compiled and primitive components are not introspected by the model. The main differences between them are that the latter is an immutable value-variable without properties and that the former has a default state (i.e. it can be created without any context, while a primitive component requires at least it's initial value to be given during its creation).

Composite component is a set of other components (called supercomponent and subcomponents respectively) interconnected by **event connections** and **shared properties.** An event connection is created when two properties are bound (see above) to each other. A shared property connection implies that a subcomponent uses its supercomponent's property (of the same type) instead of creating a new property of its own. The reference to the shared property is provided to the subcomponent by the supercomponent at initialization time (subsec. IV.B describes the implementation of this mechanism). Since the same property may be shared by several properties of subcomponents, their interaction is flexibly adjustable.

This way of "projecting" interface onto its implementation serves an apposite compromise between a trivial (giving access to an internal variable) and a too complex, model-breaking (writing in a programming language) approaches to implementing properties' functionality.

Since subcomponents are allowed to be composite components, the hierarchy can have an arbitrary depth (limited only by available hardware). Leafs of the tree are primitive (and maybe compiled) components.

Let us now describe component types and mechanisms of their creation.

B. Component type

A component type is a named entity that specifies the way the component of this type can be built. The concept of type is similar to that of class in object-oriented languages. The type describes the interface, the implementation and their interconnection for the components of this type. The interface part is comprised of **property descriptors,** each specifying a name, a type, access permissions and a default value for the property of the future component. The implementation part is different for primitive, compiled and composed types (corresponding to the three kinds of components described above). For primitive type, the implementation is a variable holding the current value of the component. For the compiled type, it's the instructions for obtaining an implementation of the component and connecting it to the interface. To create a JavaBeans component, for example, one has to provide the name of the corresponding java class (the rest is done by the Java reflection mechanism [5]).

A composite type includes a set of **subcomponent descriptors,** each specifying a type of the corresponding subcomponent of the future component and its initial value. This information is completed with connection specifications. For every property of every subcomponent it's specified which property of the supercomponent it shares (if sharing takes place). The list of necessary event connections is also stored within the type.

Let us consider the component construction process. First, references to properties are set up to point to either preliminary created properties or the shared properties of supercomponent accordingly to property descriptors. Second, subcomponents are created (one for each subcomponent descriptor) and the references to shared properties (if any) are given to them. Third, necessary event connections are established.

After a certain type was instantiated, the resulting component has all the corresponding properties and subcomponents. And all the restrictions that are being complied with during its functioning (type and access control etc.) are governed by the metainfo of the type.

C. Runtime type definition

Let us consider a process of defining a new type at runtime for which the model suggests the following course of action.

First of all, the user chooses a component from a type library (that contains predefined primitive types along with composite types defined earlier) and creates **a type editor** – a special entity that holds functionality for defining new types from existing ones. It receives the chosen type as its input data and allows performing actions listed below.

Second of all, the user is allowed to configure the resulting structure. He can change property descriptors (names, default values and access permissions) as well as implementation metadata: add or remove subcomponents, event connections and shared properties. It's important to notice that instances of a newly defined type adjust deeply to its requirements. For instance, changing a property from being random-accessed to read-only switches the underlying implementation from a variable to a constant. As another example, making a property unbindable entirely removes change listener-related functionality from the component.

Finally, when the necessary modifications are done, the newly created composite type can be added to the type library and be used on equal terms with other types.

The type editor also allows wrapping an arbitrary structure into a composite component. This is important at the beginning of development process when there're only primitive types in the library.

The type editor thereby provides access to the internal data of the type and allows modifying its **copy** in order to be able to create modified versions of types that have instances in the system without tracking changes in every type and spreading them to its instances. This makes the implementation of the type editor relatively simple and we won't concentrate on the question in this work.

V. Conclusion

We have described the core principles of the new component architecture that extends capabilities of existing component models. We have tried to demonstrate their flexibility and versatility. We believe that applications following the architecture will be able to extend, evolve and adapt to the changing requirements faster and more actively than the traditional software.

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] Dos Reis, G., Järvi, J., "What is Generic Programming?" *Library-Centric Software Design*, Montréal, Québec, Canada, 2005, pp. 1-11.

[4] Gamma, E., Helm, R., Johnson, R., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

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

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

[7] 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.

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

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

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

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