Composite Nature of Component

Wojtek Kozaczynski

Rational Software, USA

wojtek@rationla.com

Introduction

The summary of the First International Workshop on CBSE [BrWa98] contains the following definitions of a *software component*:

1. *A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.*

2. *A run-time software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at run-time.*

3. *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 party.*

4. *A Business Component represents the software implementation of an "autonomous" business concept or business process. It consists of all the software artifacts necessary to express, implement and deploy the concept as a reusable element of a larger business system.*

These definitions are indicative of the fact, that components may come in different forms and granularity. They also demonstrate that different participants of the development, deployment and maintenance process still see components differently. So what are the properties of components that distinguish them from other software artifacts such as objects, files, libraries, design documents, ect.?

In the following I propose a definition of a component. This definition captures what I believe to be the essence of software components. I compare this definition with the above proposed definitions and briefly discuss the composite structure and representation of components.

The definition

The initial idea for the definition comes from a number of sources. The key influence was my work on the BPCS ERP system at SSA. After many discussions and arguments we agreed that one of the key roles of a well-formed component is to assist configuration and project management. We also agreed that components were not atomic but they were composites.

At the January 1999 OMG meeting David Curtis presented the OMG Component Model [orbos/98-10-18]. During that presentation he made a comment that the main difference between a component and an object, which may look very similar when one describes them, is: *"that a component is at the same time a unit of construction, packaging and execution."*

I also came across similar ideas while discussing with Kurt Bittner, a member of the Rational Unified Process Group, the issues of representing frameworks and patterns in UML.

This lead me to the following definition of a component:

> *A component is a part of a system that is (at the same time) a unit of design, construction, configuration management, and substitution. A component conforms to and provides the realization of a set of interfaces in the context of well-formed system architecture.*

The operative phrase is "*at the same time*". Many software artifacts can be units of design, but not construction or substitution, for example a class. Some other, like a file for example, may be a unit of configuration management, but not a unit of design or substitution. Component seems to be the only unit that is all of the above (at the same time.)

The definition captures or implies component properties defined by the other definitions or renders them no-essential. Let us look at these properties one at a time:

Property	Essential definition
Non-trivial, nearly independent part	A properly designed unit of construction and configuration management should be non-trivial and as independent (or autonomous) as possible. A good design will localize parts that evolve at the same time and rate in order to minimize coupling between components.
Replaceable part (of a system)	Substituability implies replacability
Fulfills a clear (business) function	To be a unity of design a component should encapsulate a well-scoped (business) function. However, this is neither guaranteed nor it is a necessary condition. Functional cleanness of a component is its quality attribute, not a defining characteristic.
Dynamically bindable, introspective (interfaces discovered at run-time)	Dynamic binding of a component is a specific form of substitution. Introspection is useful, yet not necessary property of dynamically bindable components. DLLs supports dynamic binding [Rog97] but do not require introspection in its strict sense like Jini does.
Deployable independently	Components can be deployed independently in the sense that they can be given run-time resources and be activated. However, they will do something useful only if they coexist and communicate with other components that provide them with required services. This is the architecture context of the definition.
Contains all software artifacts implementing a business concept	The SSA definition expresses a particular choice of packaging components. Very consistently with our definition, these units (called Business Components) were also units of design, configuration management and substitution. The proposed definition does not imply that a component is a singe artifact.

Component structure and views

The last point in the table above is very important. A component is not a single artifact, but it is a collection of artifacts. Not all of these artifacts have to be seen at once. Similarly to architecture, a component has different views that expose its properties and contents relevant to specific set of concerns. These views and their contents (or

elements) are presented in the table below:

View	Elements
Design View	• *Interface (with a protocol)* • *Interactions between components* • *Relationships between components* • *Documentation, …*
Implementation (Construction) View	• *File* • *Source* • *Link library* • *Directory* • *Compile and link relationships* • *Class* • *Realization relationship between class and interface* • *Implementation relationship between file and class* • *DB table or file record* • *Shared memory structure* • *Documentation, …*
Deployment View	• *Object code* • *Executable* • *DDL (Shared Library)* • *HTML page* • *Bycode file (for JVM)* • *Run time configuration/parameters file* • *Documentation, …*

The main concerns addressed by the *Design View* are the interfaces of a component and how it interacts with and depends on other components. This is the classical OO view of a system. Additional concern is how atomic units (like a file) are packaged together into composite components and how these are layered.

The *Implementation View* is concerned with the configuration management aspects of component development. It worth noting, that in the proposed paradigm (language) classes are considered a mechanism for realizing interfaces and are associated with files. They are not, however, considered the key modeling elements at the system level since they have no containment semantics. The key modeling element is the component.

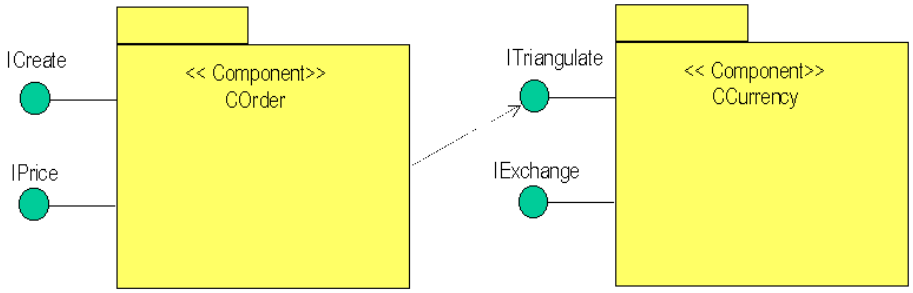
Finally, the *Deployment View* is concerned with what gets shipped and installed as the run-time version of the component.

The views are not independent of each other and could be combined, at the end of the component development cycle, into a single component model.

Component representation

UML has a concept of a component [BooRuJa98], but unfortunately it implies a single artifact. A more convenient way of representing components would be to stereotype the UML package [HoNoSo99]. This is because the package by definition is a collection of packages and/or other artifacts. The figure below shows (a part) of the *Design View* of

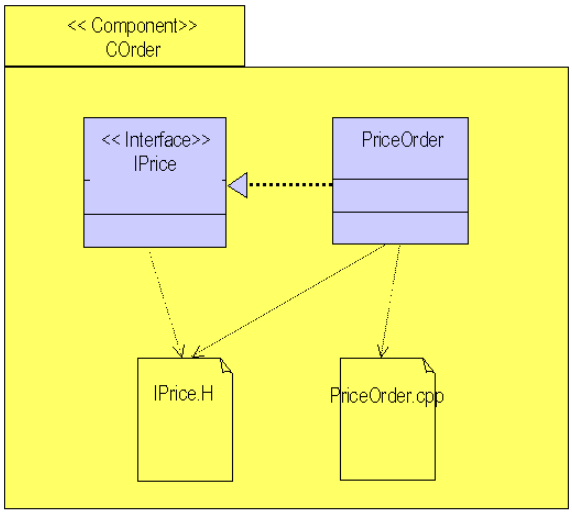
a component. This is a very simple example where a *COrder* component depends on interfaces provided by the *CCurrency* component.



Other parts of the view may contain state diagrams for interfaces, interaction diagrams of component cooperations, or detailed specifications of interfaces.

The figure below shows a portion of the *Implementation View* of a component. The interface class describes the same interface shown as a popsicle on the *Design View* and is a link between the two views. There are many other details that can be show on the *Implementation View* including directories, compile and link relationships (if they are not directly implied from other relationships), persistent date structures, etc.

We have not illustrated the *Deployment View*, but the reader should easily imagine how it would look like. Most importantly, it may contain more than one executable constructed (derived) from the elements of the *Implementation View*.



Summary

The proposed definition of a component attempts to capture its very important property of being a conceptual unity of design, construction and deployment. One of the most difficult and important roles of an architect is to decompose a system into such units – into concepts that will transcend all phases of system development.

A direct consequence of the proposed definition is that a component becomes a collection of multiple artifacts including multiple executables. This is not a common interpretation of a component. This is especially important for

the deployment and maintenance aspects of the development process.

Just as a footnote, some of the component representation aspects (in particular containment) could be more conveniently show in tabular, not graphical form.

References

[BrWa98] http://www.sei.cmu.edu/cbs/icse98/summary.html

[BoRuJa99] Grady Booch, James Rambaugh, Ivar Jacobson; *The Unified Modeling Language User Guide*, Addison Wesley, 1999

[HoNoSo99] C. Hofmeister, R.L.Nord, D. Soni, Describing Software Architecture in UML, *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1),* Kluware Academic Publishers, 1999

[DsoWil99] Desmond D'Souza, Alan C. Willis; *Objects, Components, And Frameworks With UML*, Addison Wesley, 1999

[orbos98] *CORBA Components, Joint Revised Submission,* OMG TC Document orbos/98-10-18, *November 7, 1998*

[Rog97] D. Rogerson, *Inside COM,* Microsoft Press, 1997