A Model for Classifying Component Interfaces

Sherif Yacoub, Hany Ammar, and Ali Mili
{yacoub,hammar,amili}@csee.wvu.edu
CSEE Department
West Virginia University,
Morgantown, WV 26506

Abstract

This paper identifies some issues related to component interfaces. We present a model for component interactions and interfaces to the surrounding artifacts. We classify interfaces as *Application* and *Platform*. Classification of interfaces helps in identifying issues related to a component's interoperability (interactions with other components) and portability (interactions with the platform). The model is a preliminary step towards establishing a framework for classifying and evaluating which languages and notations are adequate to specify different types of interfaces. We propose this classification for the third section of the CBSE handbook " *Technology for Supporting CBSE: Development Support* " under the " *Models* ".

1. Component Interfaces

Component-based software development is the process of assembling software components in an application such that they interact to satisfy a predefined functionality. Each component will provide and require pre-specified services from other components, hence, the notion of component interfaces becomes an important issue of concern. "*Components are expressed in terms of externally visible interfaces and semantics, not the implementation*" [2] where interfaces are the mechanisms by which information is passed between two communicating components. The use of components exacerbates interface centered software architecture because components offer interfaces to the outside world, by which it may be composed with other components [3].

Several work in component interfaces [for example 8,9, and 7] focused mainly on issues related to interaction between individual components. Component interfaces were classified as "functional" and "extrafunctional" [7], defined for UML models [8], and for object oriented designs [9]. We further abstract component interfaces to incorporate interfaces to platforms and elaborate on the importance of such classification.

In this short paper, we present a model for a component's interactions which mainly classifies interfaces as *Application* and *Platform* interfaces. This classification is useful to:

- Understand the behavior of a component and its interaction with other components and with the system on which it executes.

- Evaluate the adequacy of languages and notations to specify component interfaces.
- Inventory the range of possible inter-component interactions and use this inventory as the basis for a semantic definition of architectural constructs.
- Give some leverage on the opposition between functionality and packaging.

2. Modeling Component Interactions

2.1 The Model

Modeling software components is important to facilitate the understandability of the components themselves and the understandability of activities related to CBSD such as adapting and assembling components. The following figure shows a model that describes the component as related to its surrounding artifacts with emphasis on types of interfaces. The model is used to categorize component interactions.

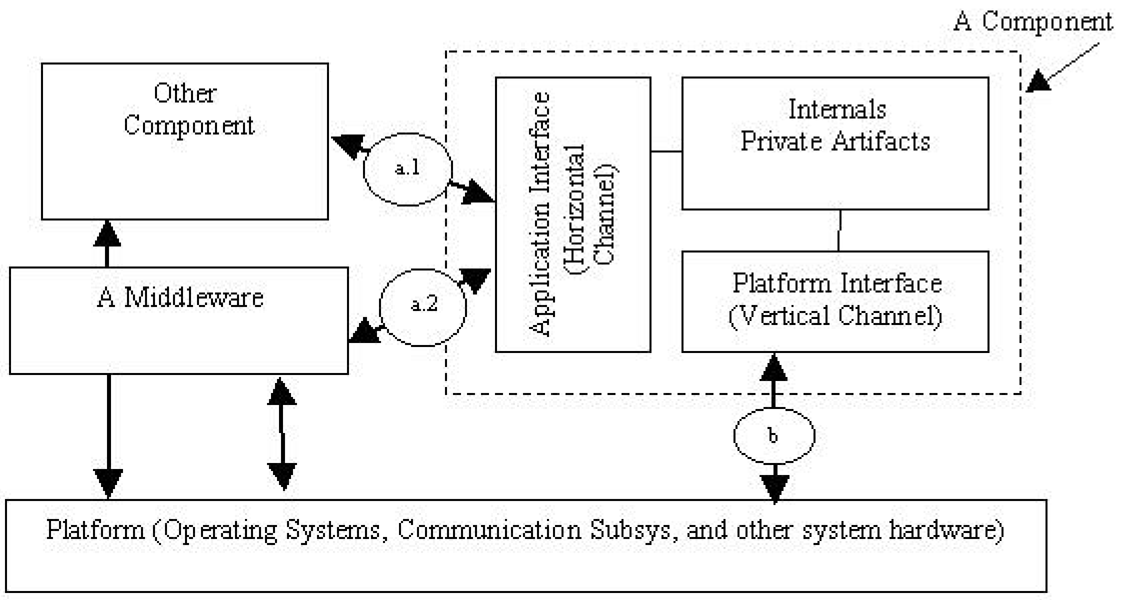
A Component
Other Component
a.1
Application Interface (Horizontal Channel)
Internals Private Artifacts
A Middleware
a.2
Platform Interface (Vertical Channel)
b
Platform (Operating Systems, Communication Subsys, and other system hardware)

Figure 1 The Model

We distinguish the following model elements:

Internals (Private Aspects)
This section of the model represents the internal information and structure of a component. It provides the actual functionality of the component as exposed by its interface. This element is private to the component and it is not exposed to any other components or the platform on which it runs. The component internals is characterized by encapsulating the decisions and hiding them from other components.

Application Interface
Those interfaces define the interaction with other application artifacts such as other components or applications. This interface represents the import and export relationship with other components (or the middleware) with which the component interacts. A set of exported interfaces represents the functionality that this component can provide. A set of imported interfaces represent the functionalities that this module requires from other external components which might be needed in the work progress of the component functional execution. We term these interfaces as "*Horizontal Channels*" as they specify the interaction with other peer components and application entities irrespective of the platform or hardware on which they run. The horizontal channel allows us to identify:

- The structure of messages sent/received from other component.
- Timing issues as related to requests going to/from the component
- Incompatibilities in data format, types and message protocol

Platform Interfaces
Those interfaces define the component interaction with the platform on which it executes. These interfaces would include operating system calls, the underlying hardware technology, and communication subsystems. For a component to run it should be supported by specific processor, memory, communication equipment and probably other hardware as well. This type of interaction is as important as interaction with other software components. It determines the portability of the component and how it runs and executes on specific hardware. This layered approach helps the designer in specifying and designing components that are independent of programming languages and operating systems. Several implementations may have different platform interfaces and yet have the same design and specifications. This interface layer is also called "*Vertical Channel*" because it identifies interactions with lower layers of hardware not with other peer components. This type of interfaces is essential for special type of applications (embedded systems for example) in which 20-30 % of safety-related errors discovered were related to these interfaces [4, 5]. The following are examples of platform interfaces:

- Operating System
- Hardware platform
- Communication channels (and protocol stacks)
- Compilers (if required to compile the component)

The Vertical Channel allows us to identify impacts of failures and risks as related:

- Failure to detect and respond to operating system and communication event
- Produce undesirable outputs to communication channels
- Misunderstanding how the hardware operates
- Portability to other platforms, (ex. a component running on Unix operating system should be differentiated from those running on Windows based or on micro-controllers)

2.2 Component Interactions

Patterns of component interaction in component-based software engineering is another major concern. Using the model of the component, we identify the following types of component interactions (numbered as shown in figure 1):

a) Application Interfaces (Horizontal Channels)

a.1) Direct Interaction
Direct interaction are those from one component to the other, in this case a component knows of the existence of other components and directly invokes one or more of its services. This type of interaction creates a direct coupling between components in the application.

a.2) Indirect Interaction
Components can interact with each other through a standardized middleware or kernel. A component publishes its services to the middleware. Other components can inquire about the possible supported services and require them without knowing where the other component is located. Indirect interaction is established through a standardized kernel, usually referred to as a middleware such as COM [1] or CORBA [6].

b) Platform Interfaces (Vertical Channels)
Components interact with other operating system components, communication subsystems, or other hardware components. These interaction protocols are determined by the nature and functionality of the component as well as the underlying platform capabilities.

2.3 Example

The model presented in the previous section is closely related to the real practice of using components in application development. For example, assume that we are developing a CORBA object that sorts an array of integers and we are making the source code availabe as a reusable component. We can identify the model elements as follows:

- *Internal*s: The sort mechanism is designed and coded in C++, this represents the private aspects of the components.
- *Application Interfaces (Horizontal Channel*): The application interface will be the Interface Definition Language IDL interface [6] specifying the functionality available (sorting) and its signature. The component can then be called through the middleware i.e the ORB.
- *Platform Interfaces (Vertical Channel*): To run this component on a Windows environment (for example), a subset of the platform interfaces could be specified by:
    - Compile with : C++ compiler for windows
    - Run on: Windows Platform

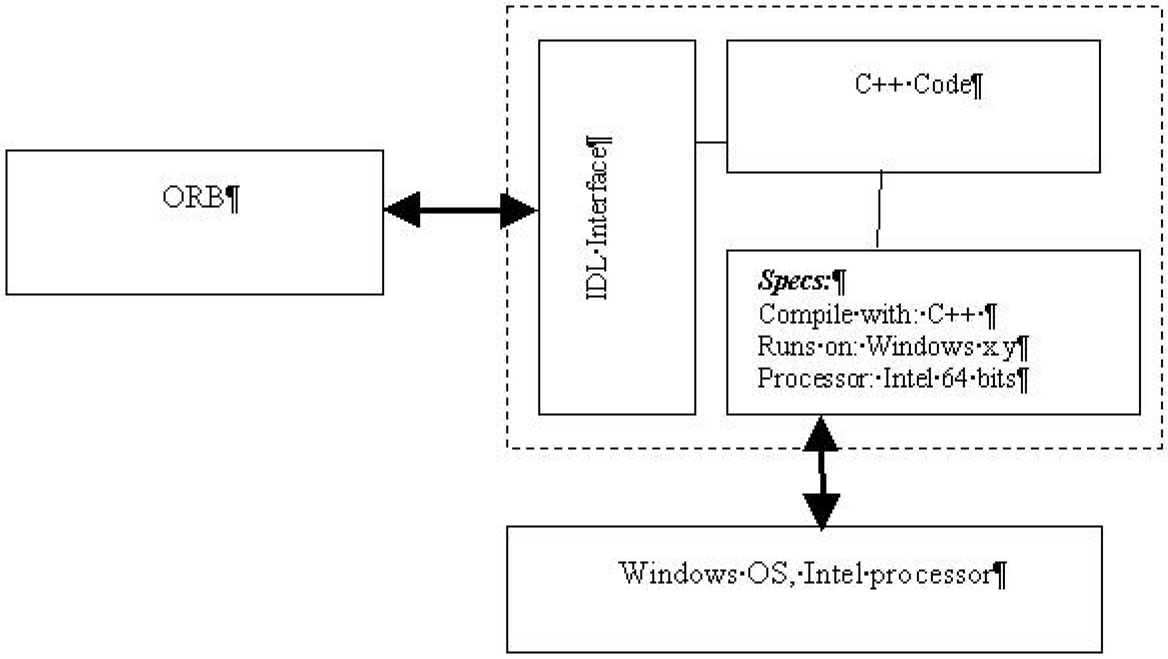

Figure 2 An Example

Now assume that we want to develop the same component in Java.

- *Internals*: The sort mechanism is designed and coded in Java.
- *Application Interfaces (Horizontal Channel*): The application interface will still be an IDL interface.
- *Platform Interfaces (Vertical Channel*): The platform interfaces would include the *Java Virtual Machine* for that specific platform.

3. Impact of Classifying Interfaces

- Establishing a framework for understanding the adequacy of existing notations an languages to specify different types of interfaces. For example one could argue that IDL is adequate for application interfaces, Java Virtual Machines are suitable for platform interfaces, or UML is generic enough for specifying internals and interfaces. *We expect to elaborate on such discussion during the Workshop.*

- Better understanding of interface mismatches. The model separates concerns abou interfaces into two categroies: Issues related to timing and message exchange between components, and issues related to hardware, communications, and other platform related issues. i.e distinguishing portability and inter-operability properties of a component. *During the Workshop, we expect to discuss the Ariane5 problem in the context of this model.*

4. References

[1] Component Object Model home page http://www.microsoft.com/com/dcom.asp

[2] Digre, T.,"Business Object Component Architecture" *IEEE Computer*, Sept/Oct 1998, pp60-69

[3] D'Souza, D. F., and Alan C. Wills "Objects, Components, and Frameworks with UML : The Catalysis Approach" , ISBN 0-201-31012-0 Addison-Wesley, 1998

[4] Heimdahl, M., J. Thompson, and B. Czerny "Specification and Analysis of Intercomponent Communication" *IEEE Computer* Magazine, April 1998

[5] Lutz, R., "Targeting Safety-Related Errors during software Requirements Analysis," *Proc. of First ACM SIGSOFT symposium on Foundations of Software Engineering*, ACM Press, New York 1993, pp95-106

[6] Object Management Group, "The Common Object Request Broker: Architecture and Sepcification" revision 2.2, 1998 http://www.omg.org/corba/corbaiiop.html

[7] Brown, A., and K. Wallnau, "The Current State of CBSE", *IEEE Software*, Sept./Oct. 1998, pp37-46

[8] Kruchten, P. "Modeling Component Systems with the Unified Modeling Language", *First Int'l Workshop on Component-Based Software Engineering*, in conjunction with ICSE'98, Kyoto, 1998

[9] Tai, S., "A Connector Model for Object-Oriented Component Interaction", *First Int'l Workshop on Component-Based Software Engineering*, in conjunction with ICSE'98, Kyoto, 1998