Component-Interface in Software Development

Virendra Kumar Sharma, Narendra Prakash Gupta, Ravindra Prakash Gupta
Faculty of Engineering, Research scholar, Research scholar,
BIT, Bhagwantpuram, Deptt. of Computer Science, Deptt. of Electronics and
Muzaffarnagar -UP, India, Bhagwant University, Bhagwant University,
E-mail: Ajmer – India, Ajmer – India
viren_krec@yahoo.com E-mail:
shikhanarendra@yahoo.com rpg1@rediffmail.com

Abstract — A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. An interface is a set of named operations that can be invoked by clients. Context dependencies are specifications of what the deployment environment needs to provide, such that the components can function.

Keywords — API - Application Programming Interface; CBD - Component Based Development; CBSE - Component-based Software Engineering; COM - Component Object Model; CORBA - Common Object Request Broker Architecture; COSE - Component-oriented Software Engineering; COTS - Commercial Off-the-Shelf; UML - Unified Modeling Language.

I. INTRODUCTION

Component-based software engineering (CBSE) is an approach to software development that relies on software reuse. It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific. Components are more abstract than object classes and can be considered to be stand-alone service providers. Apart from the benefits of reuse, CBSE is based on sound software engineering design principles Components are independent so do not interfere with each other. Component implementations are hidden. Communication is through well-defined interfaces. Component platforms are shared and reduce development costs.

II. COMPONENT

A software component is a program element with the following two properties:

• It may be used by other program elements, or clients.
• The clients and their authors do not need to be known to the component's authors.

• A component is a non-trivial, nearly independent, 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.

III. COMPONENT INTERFACE

An interface serves to name a collection of operations and specify their signatures and protocols. An interface focuses upon the behavior, not the structure, of a given service. An interface offers no implementation for any of its operations. An interface is used for specifying a service. An interface gives a name to a collection of operations that work together to carry out some logically interesting behavior of a system or a part of a system. An interface defines a service offered by a component (or a class). An interface defines a service that is in turn implemented by a class or a component. As such, an interface spans the logical and physical boundaries of a system. One or more classes (which are likely a part of some component subsystem) may provide a logical implementation of a given interface; one or more components may provide a physical packaging that conforms to that same interface.

3.1 Interface Abstraction and Application Programming Interfaces – API

Interface abstraction provides a mechanism to control the dependencies that arise between modules in a program or system. An application programming interface (API) is a specification, in a programming language, of those properties of a module that clients of that module can depend upon. Conversely, clients should not depend upon properties that are not specified by the API. All modern programming languages support some form of interface abstraction, e.g., Smalltalk80, C++, Modula3, Ada98, Java
Design and implementation decisions that are unlikely to change are specified in the API, while decisions that are likely to change are “hidden” from clients. The theory is that information hiding makes modules substitutable (for example, with new versions of a component), and hence makes systems easier to change, at least insofar as module substitution is concerned. This turns out to be a weak theory, however, as it depends upon APIs being silent about properties that clients should not depend upon. But the API can only be silent about properties about which it can speak, and programming languages are only equipped to speak about a narrow range of properties. All other properties can “leak” through the interface abstraction. Conventional APIs—that is, interface specifications written in programming languages such as those cited above—can conveniently express what we will refer to as functional properties. Functional properties include the services a module provides and the signature of these services—the types and order of arguments to the service and the manner in which results are returned from the service. Conventional APIs are not so well equipped to express what we refer to as extra functional properties. These properties include things like performance, accuracy, availability, latency, security, and so forth. These are often referred to as quality attributes, or, when associated with a particular service, quality of service. Because APIs cannot describe these properties, they cannot hide them. Indeed, modules may come to depend upon any of these properties, thus reducing the probability that one module can be substituted for another.

Note that just because an extra functional property is not expressed in an API does not a priori mean that this property will be the source of a dependency—it will only become a dependency if a client relies upon this property. This is cold comfort, however, since many more such dependencies arise than is usually recognized by programmers and designers.

3.2 Extending APIs to Extra Functional Properties

Attempts have been made to extend APIs to make them more expressive of extra functional properties. These extensions are motivated more by the desire to ensure that interface specifications are sufficiently complete to ensure correct integration than by the desire to extend the scope of information hiding to additional properties. Both ends are served by these extensions, however.

For the following discussion it will be useful to differentiate the various kinds of extra functional properties of API that have been the subject of attempts to extend APIs which are: syntactic, behavioral, synchronization, and quality of service

3.3 Interface Credentials

The bewildering variety of attempts to increase the accuracy and/or scope of interface specifications suggests that no consensus has yet emerged on how to describe the properties of modules (or components). With increased scope and accuracy comes (it seems) increased formality, which in turn leads to greater complexity and the need for still more specialized skills. A very different position was taken by Mary Shaw, who argued for credentials as a form of interface specification. Informally, a credential is a <name, value, confidence> triple, where name refers to the name of the property, value refers to the way this property is manifested by a particular component, and confidence refers to different degrees of certainty attached to the name/value pair. Shaw’s argument was that interface specifications for “architectural components” (sic) (by which she means “large”) are inherently incomplete, given the limitations of current specification formalisms, the complexity architectural components, and the variety of different ways that a component will be used.

Thus, although credentials cannot be a complete answer, the idea is useful for at least two reasons. First, the distinction between truth and knowledge may be crucial in component based systems. The distinction is certainly important to the issue of component certification since there will likely be a cost versus confidence tradeoff made in certifying properties that, for the present at least, resist formal specification or verification techniques. Second, the open ended nature of credentials nicely mirrors a much earlier argument made by Parnas that a module’s interface is, essentially, the set of assumptions that can be made about that module [Parnas 71]. The views of Shaw and Parnas take us away from the narrow perspective of interface as formal specification, and suggest the need for a more flexible way of describing component interfaces.

3.4 Components & Multiple Interface

In software component technology where there is a clear distinction between (and separation of) interface and implementation. The separation we refer to is much stronger than that suggested by languages that support separate compilation of interfaces and implementations, for example, C++ header files or Ada package specifications.
In languages and systems that support software components, an interface may be implemented by many distinct components and a component may implement many distinct interfaces.

One concrete realization of this idea is found in Microsoft's Component Object Model (COM). In COM, interface specifications are assigned a globally unique identifier (GUID) at the time they are created; each revision of that interface is assigned a new GUID. Components are binary implementations (COM is a binary standard) that are bound to the interfaces they implement via these interface GUIDs. Similarly, clients are linked to components via interface GUIDs. Thus there is a clean separation of interface and component, with clients and component alike bound directly to interfaces and only indirectly to each other. The Java programming language also distinguishes interface specification from class specification by introducing an interface type. In both COM and Java the idea is that clients depend upon interfaces and never upon implementations, and that components can implement any arbitrary number of interfaces.

But as we have seen from the discussion above, a complete separation is difficult or impossible to achieve in practice: a module implementation will have properties in addition to those specified on an abstract interface. Invariably, component implementations introduce new properties that might “bleed through” the interface. For example, assuming that an interface specification language could express performance properties (neither Java nor Microsoft IDL, the language used to describe COM interfaces, can do so), an interface may stipulate that a particular sorting operation must exhibit complexity no worse than $M \cdot \log(N)$. Even if a component were compliant with this property (and we would need to be precise about what this means), it would nonetheless bind a particular value to $M$. This binding would, in effect, change the performance property of the interface, since the client could conceivably come to depend upon this particular binding.

There are more formal ways of expressing interface bleed, but the point is that it is crucial to distinguish between abstract interfaces (those that are described independent of any implementation) and bound interfaces (those that are associated with an implementation). This distinction is an essential one for certification, composition and system analysis.

IV. CONCLUSION

Component-based approach can increase software-building performance with effective component repository. This implementation can provide interoperability between two different systems. With this approach maintenance cost is expected to be reduced since components are designed to be independent.

ACKNOWLEDGMENT

The authors would like to express their cordial thanks to Bhagwant University, Ajmer for proving the facility of Research in their University. Mr. S. P. Gupta and late Ms. Nirmala Gupta for their valuable advice.

REFERENCES