

Achieving Architectural Design Concepts with Remote Method Invocations

L.İlteriş ÖNEY

E1305937 – Computer Engineering Department - METU

Dr.Semih ÇETİN

ABSTRACT

Motivation to write this article is based on Component Based Software Engineering concepts. Problem is in Component Based Software Engineering Interfaces defined in standard component technologies using such as Interface Definition Languages (IDLs). These IDLs are sufficient in describing functional properties but insufficient in describing non-functional properties such as quality attributes and not suitable for Architecture Based Development. This article clearly describes the fact of huge improvement in this area and why Interface Definition Languages are not suitable for Architecture-Based development. However as explained in detail we the approach to overcome this situation is using dynamic invocation schemes like Remote invocation scheme or reflection. These methods make our Architecture more suitable for IDLs.

Keywords: IDL, ABD, RMI, Component, Architecture, Software, functional

1 INTRODUCTION

In Software Life Interfaces play an important role for Component Based Software Engineering. Especially in this paper we will have information about one of the differences between Component oriented design and Architectural design. Even they are not based on the same purpose they have some pros and some cons. One of them is gathering performance criteria using IDL's. As will be explained in detail IDL's only are not sufficient to provide this information. We will use Remote Procedure Calls to achieve this goal. How is made is explained in later sections. Our paper starts with explanations of the definitions of these subjects.

2 INTERFACE DEFINITION LANGAUES

The Interface Definition Language (IDL) is a language for specifying operations (procedures or functions), parameters to these operations, and data types.¹ It all starts with the Interface Definition Language (IDL). This was true when we were writing distributed systems using RPC; it is true when writing with COM; and it is also true for CORBA. In all these cases, the Interface Definition Language provides the mechanics for separating object interfaces from their implementations. IDL provides abstraction, creating something that is considered apart from its concrete existence.²

¹ Definition of Interface Definition Languages ref:1

² Introduction ref:2

It also gives us a common set of data types that we will use to define more complex data types. We will use all of these types to define the capabilities of our distributed service. Another wonderful aspect of IDL is that it abstracts away programming language dependence and hardware dependence.

IDL is a specification language. It allows us to separate the specification of the object (how you interact with it) from the implementation. This is a contract that says "Ms. Client, if you call this method, passing these parameters, then I, Mr. Server, will return to you this array of strings." Client programmers using this interface have no idea what implementation details lie behind the interface.

3 ARCHITECTURE BASED DEVELOPMENT

Architecture-based development is the exploration and maturation of the role of software architecture in the life cycle given an architecture-centric approach to product lines. Areas of investigation include: how to define and represent architecture, requirements gathering/modeling and the connection to architecture, component development and connection to architecture, architecture connection to legacy assets, connection of the architecture to the production plan, and in general tools and techniques for all the above as it relates to product lines.³

Architecture-based development comprises the following five architectural process steps:

- Forming the team structure and its relationship to the architecture.
- Using the architecture to create a skeletal version of the system.
- Using design and coding patterns within an architectural structure.
- Checking the conformance of the final system to the architecture.
- Using domain-specific languages to exploit multi-system architectural commonality.

Just having software architecture is not the same as having an architecture that is well documented, well disseminated, or well maintained. If any of these activities are not done, then the architecture will inevitably drift from its original precepts. This is a risk; if the architecture drifts in multiple ways (because of changes by multiple developers) that are not mutually consistent, or which do not follow the original rationale for design decisions, then the achievement of quality attributes that had been so carefully designed and analyzed in the original system will be compromised. So, how can we ensure that the architecture of the as-designed system and the architecture of the as-built and as-maintained system remain congruent?

Manually assessing the conformance of architecture to its design is a dreary and error-prone task. So, one technique that has been gaining popularity over the past five years is to use tools to extract the architecture of the as-built system and check it for *conformance* to the as-designed system.

However, this technique, even with tool support, is not straightforward for several reasons:

Software architectures are seldom documented in practice.

When they are documented, they are often not maintained.

³ Architecture Based Development Definition ref: 3

When they are documented, the documentation is often ambiguous.

This last point is worth some attention. Many architectural constructs have no realization in the development artifacts that programmers actually create and maintain. Nowhere in the code and header files of a typical source repository will we find a layer, a subsystem, a functional grouping, or a class category. These concepts typically exist only within the minds of the architect and a select group of programmers, and their mapping to development artifacts is unclear. Hence, architectural reconstruction typically has an *interpretive* aspect, where the architect associates certain naming conventions, file or directory structures, or structural constraints with an architectural construct. For example, a layer might be realized as any function that directly accesses the database. A subsystem might be defined by all of the files in the *IO* directory, or by all subclasses of the *PrimitiveOp* class. Yet these abstractions are useful and we want to be able to support their existence and enforce their continued existence throughout a system's lifetime.

There are other reasons for wanting to extract the software architecture of an existing system than simply for redocumentation:

An organization may want to reengineer an existing system so it needs to know what assets it is currently working with to plan the reengineering effort, or to decide that the existing assets are not worth reusing.

An organization may want to mine its existing assets to form a reuse library or the core of a product line, so it needs to know what dependencies on those assets exist in the system.

An organization may want to analyze its existing system with respect to its future prospects, growth potential, suitability for integration with other systems, or scalability. For each of these analyses, an accurate representation of the architecture is a crucial prerequisite.

4 REFLECTION

OOL provides reflection, where the class of an object can be determined at runtime, and this class can be examined to determine which methods are available, and even invoke these methods with dynamically created arguments.

5 REMOTE METHOD INVOCATION (RMI)

This section describes how can we overcome and describe non-functional properties such as accuracy, security and availability.

RMI is a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism. CORBA is another object-oriented RPC mechanism. CORBA differs from Java RMI in a number of ways:

- CORBA is a language-independent standard.
- CORBA includes many other mechanisms in its standard (such as a standard for TP monitors) none of which are part of Java RMI.
- There is also no notion of an "object request broker" in Java RMI.

Java RMI has recently been evolving toward becoming more compatible with CORBA. In particular, there is now a form of RMI called RMI/IIOP ("RMI over IIOP") that uses the Internet Inter-ORB Protocol (IIOP) of CORBA as the underlying protocol for RMI communication.

There are three processes that participate in supporting remote method invocation.

The *Client* is the process that is invoking a method on a remote object.

The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.

The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

There are two kinds of classes that can be used in Java RMI.

A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:

Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object.

Within other address spaces, the object can be referenced using an *object handle*. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

For simplicity, an instance of a Remote class will be called a *remote object*.

A *Serializable* class is one whose instances can be copied from one address space to another. An instance of a Serializable class will be called a *serializable object*. In other words, a serializable object is one that can be marshaled. Note that this concept has no connection to the concept of serializability in database management systems.

If a serializable object is passed as a parameter (or return value) of a remote method invocation, then the value of the object will be copied from one address space to the other. By contrast if a remote object is passed as a parameter (or return value), then the object handle will be copied from one address space to the other.

One might naturally wonder what would happen if a class were both Remote and Serializable. While this might be possible in theory, it is a poor design to mix these two notions as it makes the design difficult to understand.

The RMI system allows an object running in one Virtual Machine (VM) to invoke methods on an object running in another VM. RMI provides for remote communication between programs written in the programming language via the Remote Method Protocol (RMP).⁴

6 HOW TO PROVIDE FUNCTIONAL REQUIREMENTS WITH RMI'S

, As we can see in all explanations Interface Definition Languages are for commonly Component Oriented Design. But when we have a look at the side of Architectural design subjects only IDLs are not suitable. To over come this problem we mention about Remote Procedure Calls and

⁴ Explanation of RMI ref:4

Reflections for this calls. In short RMI is for calling an objects method for distributed environment. For common objects that the whole system is using is not easy task to accomplish. Let's define the problem again to see the things more clearly. We would like to define a system based on IDL's. These IDL's clearly defines the components interfaces. But gives no clue about performance criteria's for architectural design. Architectural design as all know requires some issues about performance, availability and quality. IDL's only do not provide this information for the system. To gather this information we require some mechanism. This mechanism can be achieved by using remote procedure calls. Remote procedure calls can arrange components interactions. Each component can give its objects states so we can manage performance issues just arranging these procedure calls. With the help of using RMI's objects can interact even if they exist in different platforms. These objects commonly stored in servers and clients use these objects methods just a calling their proper methods. To give a simple example lets have a look at certificate function of a system. Client object "Access" requires Database connection object that the database resides in different server. Client Access objects calls method of Adapters connection object with the correct connection string.

CONCLUSION

Architectural design requires performance criteria. Only IDL's are not sufficient to achieve this goal. To over come this problem we mention about Remote Procedure Calls and Reflections for this calls. In short RMI is used to gather performance, security and availability issue. RMI's using RPC's to give this information to the system management. The real difference between architectural design and component oriented design is based on these facts. Preparing the design requires all aspects of system while architecture is mainly based on performance issues, but component oriented design is mainly based on re-use ability and making the goal.

ACKNOWLEDGMENTS

Information gathered from the lecture SE713 Software Engineering Construction lesson lecture no: 6

CONTACT

Lütfi İleriş ÖNEY.

Middle East Technical University

ioney@tobb.org.tr

REFERENCES

1. <http://www.opengroup.org/onlinepubs/9629399/chap4.htm>
2. <http://www-106.ibm.com/developerworks/webservices/library/co-corbajct3.html>
3. <http://c2.com/cgi/wiki?ArchitectureBasedDesignMethod>
4. http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html
5. **Information** <http://www.ece.cmu.edu/~koopman/essays/abstract.html>
6. **Sun Microsystems web page** <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>
7. <http://computing-dictionary.thefreedictionary.com/Interface%20Description%20Language>