Principles and Tactics of Software Architecture

Architectural Tactics for achieving Non-Functional Requirements of Distributed Components

Nagaraju Pappu
pnr@canopusconsulting.com
http://www.canopusconsulting.com
http://canopusarchives.blogspot.com

1.0 Introduction

As the complexity of software systems increases, data structures and algorithms no longer pose the major design challenges. The structural issues – the way the system is organized and decomposed into a set of components and layers and how these components are inter connected is the major topic of Software Architecture. Even though there is not yet a single accepted definition of software architecture, the software development community more or less accepted the following view of software architecture:

"The software architecture of a program or a computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships among them". [1]

In a nut-shell, software architecture deals with decomposing the system into a set of components and integrating these components together. Both the functional requirements and non-functional requirements like scalability, performance, reliability, availability, supportability, re-use and modifiability determine how the system is decomposed into a set of components and how in turn these components are integrated together.

Modern day software systems are complex – and most systems are composed of externally manufactured components like data base engines, application servers and so on. Component based development has become the major thrust area of software development and software architecture in particular. However, there are not yet many tactics/techniques of how to design the component infrastructures that provide for realization of the non-functional requirements of component integration. In the conclusion of his paper titled "Designing models of Modularity and integration", Keith Sullivan [2] states the following:

"The state of the art in Component software remains inadequate despite decades of effort. There are some reasons for serious concern, including the need to solve long standing problems and the relative lack of architectural uniformity or stability of software.

I have found modularity models to be incredibly important, but the ones we have are inadequate"

This problem is further complicated by the fact that complex software systems are distributed in nature – and as a consequence, components need to have distributed semantics as well.

In this document, we propose a set of architectural techniques to achieve modularity, flexibility, reliability of distributed component infrastructures.

In order to understand the distributed component integration and the associated non-functional requirements, we need to analyze the problem from the following perspectives:

- Distributed Object Technologies and Components
- Component and Application Integration
- Component Granularity and how granularity effects the component construction and its usage
- Object and Component Interfaces and Component Binding

In the following sections, we explain these above issues briefly and then provide the tactics to address the issues that are raised. Finally, we explain how these tactics are applied using a

case study and argue the advantages of this approach. We conclude by specifying in what situations these tactics are useful and applicable.

2.0 Distributed Object Technologies

The holy grail of distributed object technology has been to develop a system without regard to the locations of the objects in the network, or the nature of the communication between them. The approach focuses on using a "natural" interface between objects whereby all objects are accessed in the same way. This paradigm is based on the following three fundamental assumptions:

- There is a single "natural" object oriented design for an application regardless of how it is deployed
- Performance issues are governed by the implementation of components are not relevant during early design
- The interface of an object (for example, its concurrency semantics) is independent of the context in which it will be used.

In their book, Performance Solutions, Connie Smith and Lloyd Williams [3] argued that all the above three assumptions are incorrect for the following reasons:

Latency: Latency is the difference in response times between a local and remote operation invocation. In distributed systems, the latency can be as much as four or five orders of magnitude. This is due to the fact that in distributed systems, communication is more expensive than computation.

Memory Access: Objects are accessed differently depending whether they are local or remote. The differences in the way local versus remote objects are accessed require that either:

1. The programmer must be aware of the ultimate location of the object and manage the difference. This makes the programming task more complex, OR,
2. The execution environment must provide a uniform mechanism for accessing objects that hide their location. This requires that programmers learn how to use this new mechanism. It also requires that the mechanism be used for accesses to avoid errors

Partial Failures: Perhaps the most important difference between local and remote objects is that in distributed systems components can fail independently. There are not yet well defined techniques for dealing with partial failures of distributed systems.

Concurrency: In a distributed environment, an object's methods may be invoked concurrently. To prevent inconsistencies or data corruption, these objects must define and maintain concurrency semantics. If all objects are to be treated uniformly, one of the following three approaches must be used:

- Ignore the problem. Unfortunately, this approach also ignores a significant mode of failures
- Make all objects single threaded. This leads to performance problems.
- Include concurrency semantics in all objects – regardless of their location. This adds lot of complexity and overheads.

The distributed object technologies like CORBA, DCOM and J2EE provide certain services to address the above mentioned problems. These technologies essentially provide a platform for developing components and deploying them in a distributed environment. However, all these technologies have significant overheads and impose certain restrictions on how different

components have to be integrated together. They also require static bindings between the components. In other words, each component has to know the interfaces provided by the other components and the mechanisms of invoking them. All the above technologies also use RPCs (Remote Procedure Calls) as the primary means of integration between the components in a distributed environment. The Remote Invocation has its own set of problems which is explained in the next section.

3.0 Component Integration Mechanisms

There are mainly four different ways of integrating components or applications [4]. These are:

File Transfers: One component or application writes a file that another reads. The components or applications need to agree on the file names, file formats etc.

Shared Database: One application or component writes data to a database which the other reads. The components need to agree on the keys of retrieving data, the schema of the database and the timing of when data is written and read

Remote Procedure Calls: One application exposes some of its procedures so that they can be accessed remotely. The procedure call is done as if it is local. There is a physical connection between the components. As already mentioned in the previous sections, most of the distributed object technologies like CORBA, J2EE etc use RPC in form or the other. The main problems of the RPC mechanism are the following:

- The problems related to latency, memory access, concurrency and partial failures have to be solved in the underlying environment. This adds significant overhead.
- The decoupling between the components is only logical decoupling. There is a physical decoupling between the components because one component has to execute a method on another component. This leads to each component becoming vulnerable to the other's failures. This leads to significant reliability, availability problems in addition to performance over heads.
- This imposes an architectural constraint on building distributed systems – each layer is dependent on other and also the communication between the layers/components tends to be synchronous. In an actual distributed system, components should be able to reside independently and loosely coupled to each other. This enables each component to focus on one comprehensive set of functionality and yet delegate to other components for related functionality

The RPC mechanism can be explained using an analogy:

Suppose a person X wants some money from another person Y. Instead of "asking" Y for money, X will put his hand in Y's pocket. This makes them at least temporarily connected to each other physically.

The main advantages of RPC are that they provide a mechanism for a component to execute a method on another component as if the method is local method. However, the dis-advantages far outweigh this advantage.

Messaging: Recently, message based integration emerged to solve the above problem. In this method, instead of RPCs, all components send their request to a messaging system, which in turn will send the message to the recipient. This brings in synchronicity and decoupling in the communication between the components.

However, messages are data streams – hence, the convenience of 'calling or executing a method' is lost. This means that all components must be designed to some how convert their requests into a set of messages that the other component will understand. In addition to this, messaging systems have their own over-heads in terms of lost messages, callback implementations, composing and deciding on the message structures etc.

4.0 Component Construction, Granularity and usage

All the above discussion about component assumes that the components used to construct complex distributed systems are either whose granularity is very fine (the total number of interfaces or messages exposed by a component are very limited) or that all components are internally developed.

One of the major problems of component development is that we cannot make any assumptions on how the functionality of the component will be used by another component. However, we always make such assumptions. Let's illustrate this point with an example:

Let's say that we are developing component that provides a graphical windowing environment. At the time of developing such a component we may implement each window to have a buffer that holds the text or graphics that are displayed in that window. This implementation detail is hidden from the component interfaces – in other words, the component interface only provides a method to create a window and by default it allocates a buffer to that window. Now the application programmer may be using this component to develop a spread sheet application in which each cell is treated as a window. The spread sheet application may be creating a large number of cells (in tens of thousands of them) – which leads to significant overheads. Object oriented design principles dictate that the object implementation be hidden from its interface. But, most often, this leads us to make certain assumptions on how the object will be used in the applications. Either we have to provide multiple objects which have different implementations for different types of usage, or explicitly state (in the documentation), where these objects should be used. This limits the main purpose of component based development: extensive reuse.

Another problem with component development is that as the complexity of the component increases, the number of interfaces or methods become sufficiently large set. It is not uncommon for large components like workflow engines to have more than 200-300 interfaces or API calls. There are also interrelationships and dependencies between these APIs. This makes the usage of the component very complex and leads to maintainability and modifiability problems. As new releases of the component are released, some times the old API becomes redundant, new interfaces will be provided. This requires that the component developers either provide upward compatibility – which may defeat the purpose of the new interfaces, or the application developers have to re-write their applications that use the component – which is impractical in most cases.

5.0 The whole is more than the sum of the parts

It is clear from the above discussion that so far, there is not much of distinction between a component and an object. Essentially, a component is treated as a container of a certain number of objects, each object exposing certain number of interfaces or APIs. The component's interface is essentially a union or a selection of all the interfaces of all the objects that are contained by that component. In this sense, a component is one big object. We can extend the same logic to the distributed system. The entire distributed system is a union of all the objects. In other words, the system is equal to the sum of its parts!!

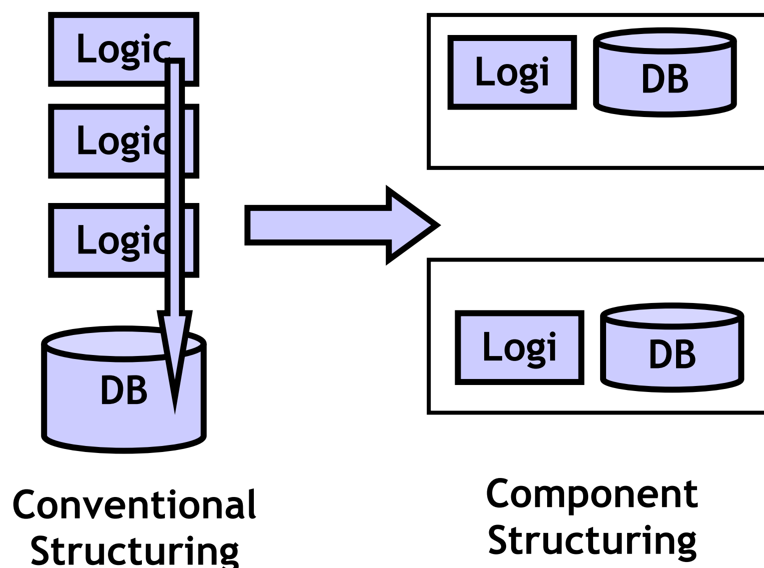
However, all good systems tend to be more than the sum of their parts. The system has its own integrity which is at the over all systems level integrity and is not simply derived from the integrity of the underlying sub-systems.

What we need is a mechanism of successive abstraction and increased modeling power as we go from an object to a component to the over all system.

In other words, the components interface must be different from a union or selection of its constituent object interfaces. Without such interface semantics, a component itself does not become a first class entity – it is treated only as a container for objects. In fact, this is the route that is taken by most of the platform based implementations.

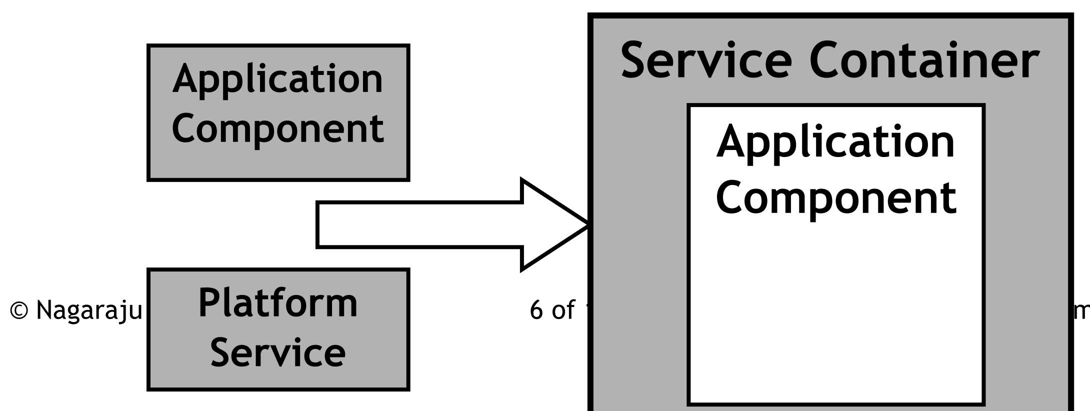
5.1 Component Evolution

In order to understand the above points more clearly, let's take a brief look at the way components technology evolved over time. The single most objective behind component based development is localizing change. Change here means insulating the business problems from the underlying technology [5]. The paradigm that component based development adopted was the paradigm of object encapsulation. The difference between conventional development and component based development can be characterized diagrammatically as follows:



5.2 Framework based Services Approach

With the advent of CORBA, J2EE, DCOM, the component evolution moved to another plane. A component is a container for objects and in the same way, CORBA and other technologies provide containers for components themselves. They provide transaction semantics, load



balancing etc., as "in built" features of the containers in which components are hosted. This approach looks like the following:

5.3 Problems with container based approaches

However, these approaches do not give us all the non-functional requirements. Even though the service containers provide certain features, still the component itself is treated as a collection of objects without any inherent "semantics" of its own.

There is yet another subtle issue with the framework based approach to component infrastructures. Since the framework provides the underlying service container, the framework imposes the scalability and integration model on the component development. For example, if J2EE is chosen as the underlying framework, then we need to host all components under the J2EE application server. The application server itself is not componentized – in other words, we do not have the flexibility to deploy and use only a certain subset of the framework's services. If we have say 20 components, and want to deploy them on 20 different machines, then we need 20 instances of the application server itself – even though we may be logically need only one application server. We may need the 'database and transaction services' of the application server only for one component, load balancing services for another component etc. But, in the framework based approach, we do not have this flexibility. In essence, the underlying framework dictates the scalability models and scalability investments.

In the following sections, we provide a mechanism to define component interface as distinct from the interfaces of the objects that the component contains, and demonstrate how such an interface addresses all the above mentioned problems and achieves the non-functional requirements of re-use, scalability, flexibility of integration and modifiability.

6.0 What is an Interface?

An interface is an instrument of interaction between two or more entities. The entities can be man and machine, machine and machine or in this case, between two components. The interface allows these entities

- To control each other through certain commands,
- To interact through communication,
- To manage and manipulate each other.

In this sense, an interface is in essence a language – consists of certain common vocabulary that the two entities involved understand and respond to. The richness of the interface depends on the richness of the "language". The interface between human beings is very complex and rich – and this is mainly due to the fact that our language is very rich. Perhaps it is

the richness of the human language that distinguishes us from other animal species. In the same vein, the richness of computer languages distinguishes it from other machines – whose interface consists of a simple set of commands that allow a person to control them.

In the following sections, we examine the above three aspects of interfaces and languages.

6.1 Control though commands

The simplest of interfaces between two entities is a set of commands that allow one entity to control the other. For example, the interface between the user and an electric light bulb consist of two words called “on” and “off”. This vocabulary is realized through an electric switch. When the user switches on the electric switch on the wall, the light bulb responds appropriately. Most of our interaction with everyday machines – be it electric equipment, household items etc., falls in this category. The interface language is realized through a set of physical instruments – switches, dials, control panels etc. Command languages consist of a set of actions. In this sense, the language consists of only verbs.

An object’s methods are nothing but a set of actions that can be performed on the object.

6.2 Interaction through communication

We need some kind of memory and a skill of anticipation and responsiveness in order to interact with another entity. An example from the human world is how people respond to the environment and feedback. For example, let’s say that a friend visits your house and you serve her tea. If she likes that tea very much and makes a remark to that effect, you may attempt to make the tea in the same fashion the next time she visits you again. In this interaction, there is no explicit ‘command’ – only a response to feedback. In the human-computer interaction space, an example is the way the machine remembers the last configuration and reloads it again. Another example is the way word processors automatically correct the typographical errors while typing. This is an example of ‘two way communication’ and responding to stimulus. In order to have this ability, clearly knowledge of the internal state of the entity is required. The persistence provided by objects and components is an example of this feature. At a language level, the language needs the richness to capture and give an ability to manipulate the state. The Data Definition Language of the SQL is an example of this kind of ability. The language does not consist of merely a set of commands, but also a grammar and provides lot of primitives and ***expressions*** that allow for a wide variety of ways of combining the ‘actions’. We don’t really have a parallel for ‘expression’ in the object world. We accomplish this by providing more complex actions that the object will support.

6.3 Management through measurements

Depending on the complexity of expressions and the richness of the language, the interaction and communication model can be manipulated in many powerful and flexible ways. If the entity also has a means of measurements, then it can be managed more easily. Without measurements, management of an entity is not possible. Again, there is no parallel for this in the object world. Though this feature is not un-common in the software systems – the interfaces of objects and components did not reach this level. This is perhaps the main reason why framework based solutions like J2EE and CORBA deal with this problem as a set of services offered by the underlying framework. However, these services themselves are offered as either automatic configurations, or through a set of objects. The language of the component itself does not provide language constructs to deal with this unlike in other programming languages like SQL – which has a rich grammar to deal with relations and provides a powerful data manipulation language.

7.0 Language Interfaces in Software Systems

A language offers very flexible modeling power – because, one does not have to make any 'rigid' assumptions of how the 'underlying functionality' is used or combined by the user. It offers the most natural way for any two entities to communicate with each other. When two people communicate with each other, even though one does not understand what the other person is saying – the communication itself does not break down. We can at best say that 'I don't understand what you say' and this way the communication will continue.

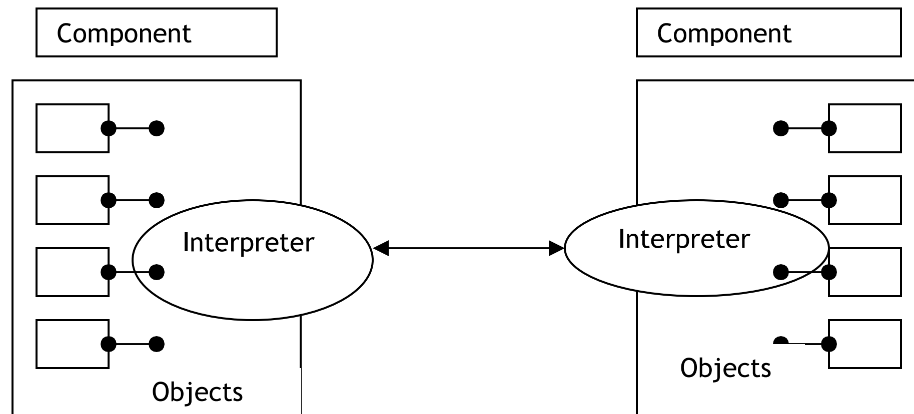
Language interfaces is not at all un-common in software systems. In fact, the power of the digital computer is its interface – a programming language. Because of language is the primary interface to the machine, we are able to build successive layers of abstraction on top of the primitive computing elements of the machine and offer very rich modeling power and configurability.

Most of the successful software systems provide language as the primary means of interface. One of the main reasons why relational theory succeeded over other database technology is mainly due to its 'relational algebra' as its main interface. Similarly, UNIX had 'C' language as its main interface language. At lower level, network protocols had a 'set of well defined' commands as their main means of interface. Lot of successful tools like AWK etc., provide some kind of grammar so that the basic primitives can be combined by the user in flexible ways. In this way, the implementer of the program does not make any assumptions on how the underlying functionality is used by the application programmer. Even the power of web is inherently because of the scripting language called 'HTML' – which allows for modeling hypertext based systems.

In the following sections, we extend the same concept as a means of building component infrastructure. In short, what we propose is the following:

Instead of a union or a selection of all the objects that it contains as its primary interface elements, a component should have a language interpreter with an appropriate grammar as the primary interface.

In other words, two components communicate through their respective language interpreters, the language consists of a shared vocabulary. The object interfaces are the primary 'actions' or verbs of this vocabulary. The following picture depicts this:



If we look at components from this perspective, a lot of 'plumbing services' provided by framework based services can be modeled much more flexibly. Essentially, the component interface now consists of the following:

- The actions (or methods) provided by the constituent objects – these serve as the basic functionality of the component.
- The services provided by the framework like automatic load balancing, fault tolerance services, connection pooling etc., which constitute the measurement, state manipulation services of the component
- The interpreter and a grammar that exposes both the services and functionality in a natural way – so that the functionality and services can be combined together, used in many different ways.

7.2 Little languages for component interpreters

The main question is what kind of grammar and language semantics are needed for components. Even though this largely depends on the type of the system, we will provide certain key pointers in this direction.

A component mainly contains the objects inside it. Objects in turn provide certain "commands" that are used to manipulate the object. This being the main nature of a component, what we need in the component grammar is the following:

- A grammar that allows us to add more commands without breaking down the component semantics or the interface – such grammar gives us the ability to modify the component, extend the component. The extensibility and modifiability requirements can be met through the component grammar.

- A grammar that allows us to combine the "commands" in flexible ways. This gives us the ability to combine the interfaces of the objects in various permutations and combinations. This has the dual advantage that the implementer of the object does not have to make any assumptions of how the 'commands' are used by the application

programmer. The application programmer does not have to know what the object is 'intended' for. He can use the object in many creative ways.

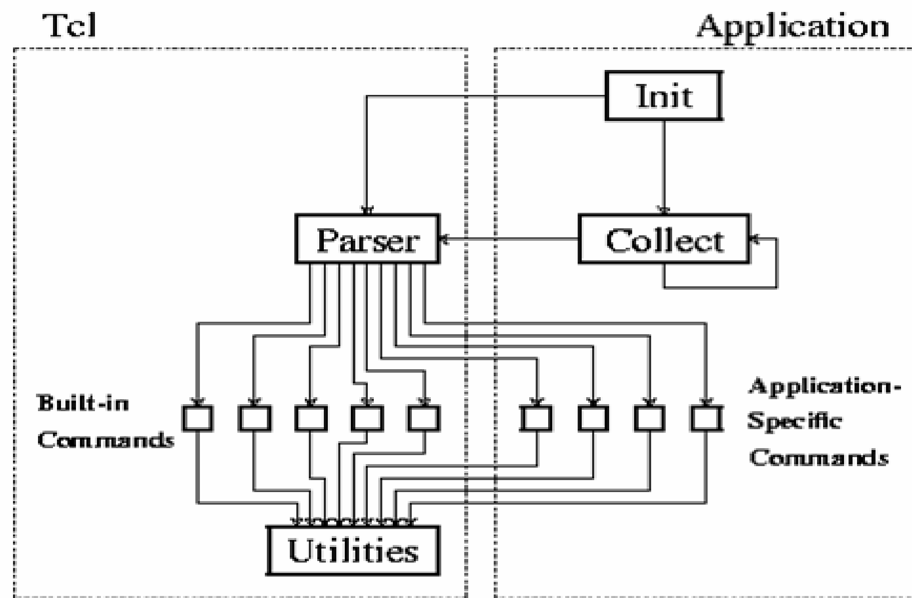
- A language semantics that that supports functions which can be composed from simple "commands". This gives us the flexibility and ability to create "complex programs" from simple primitives. We can then use the simple commands, write a script, and then use the script to create another complex script.

- An interpreter that can be embedded into a component or a program. If we have to develop complex language grammars and syntax and semantics then the object designers have to first learn the language – so that they can expose their object's interfaces as commands into the component language interpreter.

We need to address how a component implementer goes about his task. The major requirements of the components are the functionality that they provide. A component designer focuses mainly on dividing the functionality into a set of objects and provides interfaces for these objects. If she has to focus on designing a language first – then it makes the component development a very complex task. Language design is not an easy task by any means. What we need is to provide an ability to the component designer to focus on the core of the job – to design the objects and their interfaces, but provide them a means to quickly and easily 'drop' these interfaces as the main vocabulary of the component interpreter. In other words, the job becomes much simpler, if we can have a generic interpreter that is embeddable into the components.

Fortunately, this is a solved problem in computer science. We don't really have to invent such languages. There are lot of little languages and language processors that are embeddable and extensible.
John Ousterhout's TCL (Tool Command Language) is one such example. Everything in TCL is a command, and these commands can be combined with one another. TCL is also designed to be "embeddable". It is possible to develop a program, and expose the functions of the program as TCL commands to the TCL interpreter. The TCl language itself consists of a set of commands that can be combined in various ways and a very simple grammar.

The following diagram depicts how combining a TCL interpreter with a program looks like:



Tcl is only one such example interpreter that serves as a generic component interpreter. There are several such little, embeddable language interpreters that one can use depending on the functionality, what one to accomplish in the component interfaces etc.

8.0 Advantages of language interfaces

In this section, we demonstrate how some of the non-functional requirements can be achieved through language interfaces.

8.1 Dynamic Bindings and Parallel Development

The language interpreter gives us the ability for dynamic bindings between components. Many of the distributed object technologies like COM and others relies on static bindings between components. With a built in interpreter, it is possible for two components to connect to each other at run time – as long as both components understand the same interpreter. This gives us the ability of parallel development of components. One component designer does not have to wait for other to complete and publish his interface libraries. Developers can independently develop their components, test them and connect them at run time.

John Ousterhout states in his book on Tcl programming language [6]: "Tcl's use of common interpreted language for communication between applications is more powerful and flexible than static approaches such as Microsoft's OLE and Sun Microsystems's ToolTalk."

8.2 Reliability of component interactions

Most often, many reliability problems in distributed systems arise from the component interactions. If the number of interfaces and the complexity of a component increase significantly, then it becomes very complex to preserve the overall integrity of the system. In a language interface model that we proposed, this problem does not occur. The complexity is contained in the interpreter itself. As long as the interpreter is stable – the interaction itself does not cause any breakdowns between the components.

8.3 Testability

Since components can be independently developed and they have an in-built interpreter, the component can be tested in an isolated manner. We don't have to write full fledged applications to test the components. We can easily develop test drivers that combine the component commands into scripts and these scripts can be executed. In fact, the interpreter can be extended to support testing features – for example, the interpreter can be extended by adding certain commands that make testing easy – commands like traces, logs, execution timers etc. This greatly eliminates the troubles of integration testing.

8.4 Extensibility and Modifiability

The functionality of the component can be extended very easily because the component itself is a first class entity with an interface that is not affected by the underlying objects. Hence, it is possible to drop new objects and interfaces into the component, initially expose them only for the test drivers, and then finally drop them as vocabulary into the component interpreter. As soon as these new commands are dropped, they are available for use by others. Adding new vocabulary does not make old vocabulary break down. It only extends the language.

This is one of the main advantages of the language based developments of components. As the component functionality increases by an order of magnitude, the current approaches of providing APIs becomes too complex to manage. Large components like workflow engines etc., have more than a couple of hundred API calls. Such complexity can easily be brought down to a manageable level using language interfaces. In a language based model, each API becomes a "command" – and in this way, there is no necessity to provide any dependencies, calling sequences, how, where and when to call the API etc. That is automatically taken care of by the interpreter itself.

8.5 Ease of Integration and component management

Language based approach to components decouples the scalability and integration issues. The integration of the components is reduced to selecting a communication means – the scripts or programs can be exchanged over sockets, they can be sent as messages etc. Because of this, there is no inherent 'scalability restrictions' – unlike application server model where we have to hook the component to the application server container, in the language based development, we hook the interpreter to the component. In this sense, the components are not dictated by any external containers. The language interpreter is "in-built" into the component and not the vice-versa. This makes the management of components much easier and also the integration of different components becomes a lot easier. We can program a lot of "semantics" into the "scripts" – for example, each script can define its own priority levels, what needs to be done in case of partial failures etc.

9.0 References

Kevin J. Sulllivan, "Designing models of modularity and integration", Component Based software engineering.

Len Bass et al, Software Architecture in Practice

Connie Smith and Lloyd Williams, Performance Solutions

Integration Patterns from Thoughtworks

The evolution of components from the component source website

The TCL programming language