Reference Architecture: The best of best practices

Paul Reed

September 15, 2002

Why does one project within an organization flourish while a project with the same fundamental architectural needs, within the same organization, flounders? Often, the root of the problem is a lack of horizontal communication across all projects regarding architectural choices, both good and bad, that were made on past projects. The Rational Unified Process® (RUP®) states that such "harvesting" of best practices within the organization is the first step toward building a strong, versatile reference architecture. Briefly, a reference architecture consists of information accessible to all project team members that provides a consistent set of architectural best practices. These can be embodied in many forms: prior project artifacts, company standards, design patterns, commercial frameworks, and so forth. The mission of the reference architecture is to provide an asset base that projects can draw from at the beginning of the project lifecycle and add to at the end of the project.

Many projects I encounter spend an inordinate amount of time researching, investigating, and pondering architectural decisions. This is especially unsettling when it's clear that if prior project teams had taken the time to document their experiences and build up a reference architecture, they could have spared the new project teams much of this research and decision making. In fact, the inability to learn from a cumulative project "history" probably puts a new project's timetable at greater risk than all other factors combined. A project that proceeds without reference information will not necessarily fail; it will just require considerable effort on the part of the project team that could be spent better elsewhere. Organizations can hope to get software into the hands of clients sooner only through realizing tried and true repeatable processes.

Fortunately, it doesn't take much time to put together a winning taxonomy for a reference architecture that can dramatically reduce the number of incorrect technology decisions and increase the likelihood that a project will get off on the right foot. Such frameworks have been a hallmark of successful organizations since the early days of distributed Client/Server applications; as systems grow in complexity, the requirement becomes more imperative.
This article will review the role that a strong reference architecture can play in software development projects, following the guidelines provided in the RUP. In addition, it presents a practical taxonomy for collecting, managing, and using the reference architecture effectively.

What the Good Book Says

According to the RUP (see Figure 1), a reference architecture:

- is, in essence, a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed, and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects.

In both the Inception and Elaboration phases, the RUP goes on to say, the team should consult its reference architecture as part of the Architectural Analysis activity for the new project (see area circled in red in Figure 2). However, RUP further states that "...creation of reference architectures is an organizational issue and currently outside the scope of the RUP." This is not an oversight on the part of the RUP -- far from it. The structure, content, and management of a reference architecture should be based on the organization's unique structure and needs. Small organizations, for example, often do a good job of communicating about past projects via low ceremony, face-to-face consultations between team members in the project's development "bullpen," and they might not need a lot of formal documentation. Larger organizations might maintain a repository on the corporate Intranet that team members can consult throughout the project lifecycle. Unfortunately, many large organizations do not take the time to do this well.

As Figure 1 depicts, the RUP consists of four phases and nine disciplines. An iteration is a distinct sequence of activities with a baseline plan and valuation criteria resulting in an internal or external release. It slices through the nine disciplines, drawing from the available activities in each one. The resulting set of tasks comprises what is known as the iteration plan. A given phase may have multiple iterations, and the number is usually a factor of the technical complexity and size of the project. Sample iteration plans for each of the four phases are provided in the RUP.
The end of each phase is marked by the completion of a milestone. The milestones for the four phases of the RUP are: Lifecycle Objective, Lifecycle Architecture, Initial Operational Capability, and Product Release.

Unlike waterfall-based process models, the RUP’s iterative model acknowledges that activities from the broad spectrum of disciplines (requirements, analysis and design, and so forth) actually take place concurrently throughout the lifecycle of the project.
What Does a Reference Architecture Look Like?

The RUP suggests that a reference architecture should be defined along different levels of abstraction, or "views," thereby providing more flexibility in how it can be used. Ideally, these views map to the 4+1 Views of software architecture outlined in the RUP (see Figure 3) and embodied in the RUP's Software Architecture Document.

Figure 3: 4+1 Views of Software Architecture Contained in the RUP

<table>
<thead>
<tr>
<th>Logical View</th>
<th>Implementation View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use-Case View</td>
<td></td>
</tr>
<tr>
<td>Process View</td>
<td>Deployment View</td>
</tr>
</tbody>
</table>

Note that according to the RUP, only the Use Case and Logical Views are required on all projects. The other views should be used if the particular system to be constructed requires them (e.g., the Process View is necessary when there are multiple threads of control; the Deployment View is...
necessary when the system is to be distributed across more than one node). Although there are
certainly many ways to portray a reference architecture diagrammatically, I prefer a familiar set
of functional layers tied to the Logical View of the 4+1 framework. I also like to expand upon the
RUP’s definition of a reference architecture to include technology compatibility and tool selection
for projects as well.

**Figure 4: Functional Layers Within the Logical View**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface</td>
<td>Standards and tools to support the user interface and presentation services (e.g., html, rich client, dumb terminal)</td>
</tr>
<tr>
<td>Business</td>
<td>Standards and tools to support the Business logic (e.g., languages, component standards)</td>
</tr>
<tr>
<td>Middleware</td>
<td>Standards and tools to support transaction management and interprocess communication (e.g., container services, ipc API)</td>
</tr>
<tr>
<td>System Software</td>
<td>Standards and tools to support systems management (e.g., operating systems, database management systems)</td>
</tr>
</tbody>
</table>

In addition to the layers shown in Figure 4, in some cases there might be layers within layers.
For example, the System Software layer contains architectural information on both database
management systems (DBMS) as well as operating systems.

Deciding how to present the information for the different layers can be a challenge. In general, the
presentation must be to-the-point and concise. At a minimum, the organization needs to publish
simple matrices/tables that facilitate a quick look up and easy communication of the material (see
Tables 1-4).

I also like to include in these matrices real project artifacts from prior efforts. These might be in the
form of both paper documentation and actual running code (components) that may be reused.

And finally, I like to add references to both external and internal resources to the matrices. These
might be sections of books or Web sites that describe best practices and design strategies. These
resources might also reference internal company whitepapers that discuss or review, at length,
different architectural choices.

Now, let’s take a closer look at each of the four layers.

**User Interface Layer**

The user interface layer covers many bases. More than just a collection of standards on the
placement of widgets on screens, it covers aspects of both technology selection and evolution
to newer technologies (such as using XML as a presentation vehicle and using the Model View
Controller (MVC) pattern as a means to decouple presentation from content context). See Table 1
for a partial sample of what this layer might contain.
## Table 1: User Interface Layer (Partial Sample)

<table>
<thead>
<tr>
<th>Area</th>
<th>Products/Services/Components</th>
</tr>
</thead>
</table>
| Layout Styles and Usability         | 1. See the product from Creative Computing called "GUI Guidelines" that was purchased by the organization in 2000. It covers layout styles for both rich client and Web-based applications.  
2. All projects should use the usability lab that has been set up in the Quality Assurance group. This lab provides for both audio and video recording of testing sessions as well as feedback forms for users to complete. |
| Construction Tools:                 |                                                                                                                                                                                                                            |
| 1. Languages                        | 1. IBM Mainframe: CICS BMS Maps and Cobol II for manipulation.  
2. Non-IBM Mainframe: Java Server Pages (JSP) should be used as the primary presentation vehicle for Java-based solutions. Java Applets are not allowed at any time.  
3. Non-IBM Mainframe: Active Server Pages within the .NET architecture should be used as the primary presentation vehicle for Microsoft-based solutions. Embedded ActiveX/COM+ controls in HTML delivered presentations are not allowed at any time. |
| 2. Integrated Development Environment (IDE) | 1. IBM Mainframe: TSO/SPF  
2. Non-IBM Mainframe: Borland JBuilder V7  
3. Non-IBM Mainframe: Microsoft VS.NET |
| 3. Information Stream (payload content) | 1. IBM Mainframe: 3270 stream for Mainframe-to-Mainframe communication and XML in cases of interapplication communication between Mainframe and non-Mainframe applications.  
2. Non-IBM Mainframe: HTML and XML where appropriate. |
| 4. Design strategies                | 1. Unless it can be proven otherwise, applications should always employ the Model View Controller (MVC) design pattern when externalizing views of information. See the book by Gamma et al, "Design Patterns: Elements of Reusable Software" for an academic description.  
2. Non-IBM Mainframe Java Applications: These applications should use Java Server Pages, and Java Servlets (the Struts framework should be used to implement the interaction between the presentation and business layers). This is also referred to as the Model 2 architecture in the Java community. Example: See the Order Entry application implemented in Q42002. It is an exact template of what to follow.  
3. Non-IBM Mainframe Microsoft Applications: These applications should use the ASP.NET architecture coupled with controlling components on the server-side. Although there is no equivalent of the struts framework to use for Microsoft applications, reviewing the sample application mentioned below will provide an excellent guide. Example: See Rene Becnel's internal whitepaper on the proper approach to using ASP.NET within the organization. |
| Components:                         | 1. For Java applications, the Struts framework from the Jakarta group (jakarta.apache.org) is the standard to be used for the communication of errors back to the user interface as well as for the setup of processing user side input on the server side (see business layer discussion of struts). |
Business

Many times projects spend an inordinate amount of time exploring tool options. In addition, they might not be aware of the best approach to take when designing certain aspects of the application. This is where a reference architecture can be the most valuable. The project team can harvest not only tool and language choices, but also design patterns and pre-built components if they are available. In the business layer lies the strongest potential for facilitating reuse.

Table 2: Business Layer (Partial Sample)

<table>
<thead>
<tr>
<th>Area</th>
<th>Products/Services/Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components:</td>
<td></td>
</tr>
<tr>
<td>1. Languages</td>
<td>1. IBM Mainframe: Cobol II</td>
</tr>
<tr>
<td></td>
<td>2. Non-IBM Mainframe: Java 1.4</td>
</tr>
<tr>
<td></td>
<td>3. Non-IBM Mainframe: VB.NET or C#</td>
</tr>
<tr>
<td>2. Integrated Development Environment (IDE)</td>
<td>1. IBM Mainframe: TSO/SPF</td>
</tr>
<tr>
<td></td>
<td>2. Non-IBM Mainframe: Borland JBuilder V7</td>
</tr>
<tr>
<td></td>
<td>3. Non-IBM Mainframe: Microsoft VS.NET</td>
</tr>
<tr>
<td>Pattern Usage -- Business Scenario</td>
<td></td>
</tr>
<tr>
<td>1. At runtime the application does not know the exact type of instance to create.</td>
<td>1. Factory Pattern -- This pattern should be used for all resource lookups and any other dynamically created resource in the application. For Java applications using BEA, the definition of the resources will be sourced in the deployment descriptors. Example: See the Order Entry application implemented in Q22002. It used the Factory pattern extensively for resource and dynamic object lookups. For .NET applications, they will be kept in .config files. Example: See the Marketing Forecast application implemented in Q12002. It uses .NET config files.</td>
</tr>
<tr>
<td>2. Each use case represents several pathways of work and should be as highly cohesive and as loosely coupled as possible in its implementation to facilitate reuse.</td>
<td>1. Facade Pattern -- More specifically, the use-case design pattern should be used. Each use case should have a separate Facade class to promote less coupling and simplify the transaction control/unit of work settings.</td>
</tr>
<tr>
<td>3. The external presentation of information varies, so the dependency between how information is rendered (i.e., Web, reports) and how it is actually stored should be low.</td>
<td>1. Model/View/Controller (MVC) -- All applications must use an MVC architecture to separate formatting of information from the actual storage and object relationships. For the Java platform, Model 2 architecture only must be used. (See note above regarding user interface and use of MVC)</td>
</tr>
<tr>
<td>Service Components:</td>
<td></td>
</tr>
<tr>
<td>1. Security</td>
<td>1. IBM Mainframe: RACF exits</td>
</tr>
<tr>
<td></td>
<td>2. Non-IBM Mainframe: IBM Secure Way</td>
</tr>
<tr>
<td>2. Logging</td>
<td>1. IBM Mainframe: Module TE06LOG</td>
</tr>
<tr>
<td></td>
<td>3. Non-IBM Mainframe (.NET) - Logger Assembly (logit.dll)</td>
</tr>
<tr>
<td>3. Routing of requests from the user interface into the business layers</td>
<td>1. For Java applications, the Struts framework from the Jakarta group (jakarta.apache.org) is the standard to be used for routing of requests from the user interface and ultimately interacting</td>
</tr>
</tbody>
</table>
with the business layer. Struts uses the Command pattern to decouple the execution of logic for a given transactional request.

Data Access Components

<table>
<thead>
<tr>
<th>Area</th>
<th>Products/Services/Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Stored Procedures</td>
<td>1. Stored procedures are highly discouraged unless they correct sever application performance problems. Even then, all alternative application tuning efforts should be tried prior to using the Stored Procedure.</td>
</tr>
<tr>
<td>2. Data Transport</td>
<td>1. Both Value Objects (i.e., static, data-only snapshots of objects) and Data Access Objects (i.e., objects that actually execute the SQL statements) should be used whenever possible. Example: See the Order Entry application implemented in Q42002 for examples of both Data Access Objects and Value Object usage.</td>
</tr>
</tbody>
</table>

Middleware Layer

The middleware layer can be thought of as a pure service layer. Although the term middleware is immensely overused today when discussing architecture, it is this layer that usually plays the largest horizontal role in the overall organizational architecture. (By horizontal I mean that the services here apply to almost all of the applications that are being developed, that are under development or that are purchased.)

Table 3: Middleware Layer (Partial Sample)

<table>
<thead>
<tr>
<th>Area</th>
<th>Products/Services/Components</th>
</tr>
</thead>
</table>
| Application Servers:              | 1. IBM Mainframe: CICS V7  
2. Non-IBM Mainframe: BEA Weblogic V7  
3. Non-IBM Mainframe: Microsoft COM+ |
| Messaging Services:               | 1. IBM Mainframe: MQ Series  
2. Non-IBM Mainframe (Java): JMS/MQ Series Interface  
3. Non-IBM Mainframe (Microsoft): MSMQ/MQ Series Interface |
| Directory Services                | 1. IBM Mainframe: Secureway LDAP Interface  
2. Non-IBM Mainframe (Java): JNDI and Netscape LDAP Directory  
3. Non-IBM Mainframe (Microsoft): Active Directory/LDAP |
| Data Distribution Strategies     | 1. No distributed two-phase commit processing will be supported.  
2. Centralized data access is encouraged at all times. When necessary, data stores may be segmented across servers and/or geographic locations. However, even in these cases, central access is the preferred access pattern. |
| Data Access APIs                 | 1. IBM Mainframe - Embedded SQL  
2. Non-IBM Mainframe (Java) -- JDBC V3  
3. Non-IBM Mainframe (.NET) -- ADO.NET |

System Software Layer

The system software layer is usually the easiest to communicate. New projects usually don’t specify a new operating system that must be supported by the organization. Many times, the
system software layer is the most expensive layer in terms of licensing fees. It is here that the organization should leverage a strong reference architecture that limits choices to the barest minimum.

Table 4: System Software Layer (Partial Sample)

<table>
<thead>
<tr>
<th>Area</th>
<th>Products/Services/Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Systems</td>
<td>1. IBM Mainframe: ES/9000</td>
</tr>
<tr>
<td>Database Strategy</td>
<td>1. Product -- Relational</td>
</tr>
<tr>
<td></td>
<td>2. IBM Mainframe: DB2</td>
</tr>
<tr>
<td></td>
<td>2. Non-IBM Mainframe: Microsoft SQL Server 2000</td>
</tr>
</tbody>
</table>

Creating a Reference Architecture

There is something of a chicken and egg problem with respect to a reference architecture. We have defined what a reference architecture is and what its structure looks like, and even provided a partial sample. However, we haven't discussed how the reference architecture gets created in the first place.

This challenge is typically spearheaded by an architecture group. Usually, I am not a fan of formal, permanently appointed architecture groups; many times the individuals in these groups become very savvy about technology but lose sight of the business' needs and how people need to use technology. A more effective approach is to set up an informal architecture group that includes representatives from various IT organizational departments (e.g., development, support, operations). Business representatives should also be part of this group. With an informal group, the challenge is to ensure that there is adequate management support and follow-through, and it might be effective to appoint a group member to remind people of their commitments and responsibilities.

Using the layered reference architecture presented in Figure 4, the group could simply fill in the slots shown in each Table above, drawing upon both their knowledge of the organization's technology and a heavy dose of the best practices they would like to see furthered across all projects.

In summary, getting a reference architecture off the ground requires an effort that combines the effective tools, technology, and approaches currently in place within the organization (i.e., harvesting of existing assets) with an infusion of best practices, suggested patterns, and approaches that the organization thinks will make applications even better and projects more successful.

Using a Reference Architecture

Early in the creation of the project's Vision, it is necessary to assess the impact of the business case on other solutions already in production as well as the benefits the existing reference architecture can bring to the project.
In the RUP, this is the desired outcome of the Architectural Analysis activity (described and outlined in Figure 2), which truly leverages the reference architecture to its fullest. The organization might discover that it needs to adjust its reference architecture in response to this early assessment. For example, requirements might dictate a new middleware service not yet articulated in the existing reference architecture.

For example, suppose an organization's business case calls for implementing a Web Services application that integrates two disparate legacy applications. Web Services requires the exchange of eXtensible Markup Language (XML)-based messages as well as decisions on protocols (i.e., Simple Object Access Protocol (SOAP) vs. XML-Remote Procedure Call (RPC)). If the reference architecture doesn't currently address any of these technologies, then questions will arise as to which standards to follow (e.g., Microsoft Web Services, Java support for Web Services, etc.). If the reference architecture does indicate a preferred development language -- perhaps Java -- then there will be fewer decisions, but additions to the reference architecture will still be required (e.g., now we have a preferred SOAP implementation standard that requires placement in the reference architecture).

In addition, these new reference items might cascade and affect established architectural standards. Again, using the previous example, because security issues are not as clear-cut for Web Services as they are for more traditional applications, the organization would have to review and possibly adjust its standard security policies.

Quite often, the first project that bumps up against an architectural issue not addressed in the reference architecture will, by default, set new architectural standards for future projects. However, it is still vitally important that the reference architecture be able to absorb the addition without destroying all previous standards. It is important to watch for a ripple effect and carefully monitor any decisions to alter standards.

Keeping the Reference Architecture Up to Date

A development history, along with benchmarks, for each project is the key to keeping a reference architecture from becoming stale and obsolete. In the RUP, the Activity Prepare for Project Close-Out, found in the Transition Phase, calls for an update of the reference architecture.

Another valuable addition to a reference architecture is a description of experiences in supporting applications following implementation. The project team should assess the architectural soundness of applications that turned out to be troublesome in a production environment. Although the team might have made correct decisions, the system's combination of technologies might be just too difficult to support. Note that adding this wisdom is different than harvesting assets for the reference architecture at the end of a project lifecycle.

For example, suppose you have implemented several client-centric applications. Perhaps the architectures are effective and the applications work according to specifications, but support issues, such as software distribution and synchronization of releases, are problematic. The reference architecture might respond to these issues by mandating that, in future applications, the application architecture should follow a thin-client model unless there is demonstrable evidence
it won't work. In such a case, direct support issues would have a far-reaching impact on the reference architecture. Yet there wouldn't be a great deal of data on them until well into the application's support cycle.

**Making it Work**

Ultimately, responsibility for maintaining and continually updating the reference architecture rests in the hands of front-line project management and the architecture group. Project managers must take quite seriously both the Architecture Analysis activity (asset consumption) that the RUP suggests during the Inception and Elaboration phases as well as the Prepare for Project Close-Out activity it recommends during the Transition phase. As the RUP states, it is in closing out the project that architectural fruits are harvested. However, as we have discussed, to be of real use, the reference architecture must be continually updated and refined, not only with information in prior project artifacts, but also with data gathered from managing applications in a production environment.

Senior management must look beyond the short-term tactical costs of constructing and maintaining a reference infrastructure and see the strategic importance and payback of having such an architecture in place. Although the creation, care, and feeding of a reference architecture requires resources -- to assist with project compliance and for potential common component enhancements -- these resources may simply matrix over from existing projects when services are needed.

Ultimately, organizations don't want project teams to agonize over what programming language to use or what logging API to follow. They don't want their teams trying to figure out what pattern is best to use when there is a need to separate externalized information from its internal state. Nor do they want them pondering over whether to use Model 1 or Model 2 architecture when building J2EE applications. All of these decisions should already be made for every project, waiting to be extracted from the reference architecture. And if choices do need to be made, then the options should be few and clearly articulated via the reference architecture.

A reference architecture isn't meant to stifle creativity, but rather to enforce commonality across projects. Ultimately, the supreme goal is to reduce total cost of ownership (TCO) for technology usage in the organization. Organizations report today that vertical reuse on projects is common. Horizontal reuse across applications is much more rare, especially in medium- to large-scale organizations. Here there is low-hanging fruit ripe for the picking.

For an organization, introducing a technology is akin to administering a drug. It's important to monitor reactions closely and anticipate side-effects. Establishing a reference architecture infrastructure that will continually evolve and grow is one way to ensure that the side-effects will be much more manageable.

**References**


Paul R. Reed, Jr., *Developing Applications with Java and UML*. Addison-Wesley, 2002.


**Notes**

¹ Note that larger organizations might opt to place some of the information listed in Tables 1-4 (e.g., tool selection) in other policy documents rather than in the reference architecture. But for smaller organizations, or organizations with no standards, the reference architecture is an appropriate choice, as decisions on these matters impact the architecture.

© Copyright IBM Corporation 2002

*Trademarks*
(www.ibm.com/developerworks/ibm/trademarks/)