Realtime publishers

"Leading the Conversation"

The Definitive Guide™ To

Service-Oriented Systems Management

sponsored by

altiris®

Dan Sullivan

Chapter 7: Implementing Systems Management Services, Part 3: Managing Applications and Assets 137
Application Life Cycles 137
Business Justification 139
Requirements Phase 140
Functional Requirements 140
Security Requirements 141
Integration Requirements 142
Non-Functional Requirements 143
Analysis and Design 145
Solution Frameworks 146
Buy vs. Build 149
Detailed Design 150
Development 151
Source Code Management 151
System Builds 151
Regression Testing 152
Software Testing 153
Software Deployment 154
Software Maintenance 155
Role of Application Development Life Cycle in Systems Management 155
Managing Application Dependencies 156
Data Dependencies 156
Time Dependencies 157
Software Dependencies 157
Hardware Dependencies 157
Application Asset Management 158
Acquiring Assets 158
Deploying, Managing, and Decommissioning Applications 159
Summary 159

Copyright Statement

© 2006 Realtimepublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimepublishers.com, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimepublishers.com, Inc or its web site sponsors. In no event shall Realtimepublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtimepublishers.com and the Realtimepublishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtimepublishers.com, please contact us via e-mail at info@realtimepublishers.com.

[**Editor's Note:** This eBook was downloaded from Realtime Nexus—The Digital Library. All leading technology guides from Realtimepublishers can be found at http://nexus.realtimepublishers.com.]

Chapter 7: Implementing Systems Management Services, Part 3: Managing Applications and Assets

Networks, servers, and client devices alone do not address the information needs of an organization—applications, and their associated data, customize the functions of an otherwise generic infrastructure and allow IT to meet the information management requirements of businesses, agencies, and other organizations. The ability to finely customize software to meet particular needs makes it a key to aligning information services to business strategy. At the same time, the flexibility introduces a wide variety of management challenges. These challenges have by no means been completely mastered, and software developers continue to create and refine new development methodologies. There are, however, common elements to application management frameworks. This chapter will examine the challenges of application management from the perspective of application life cycle management and software asset management.

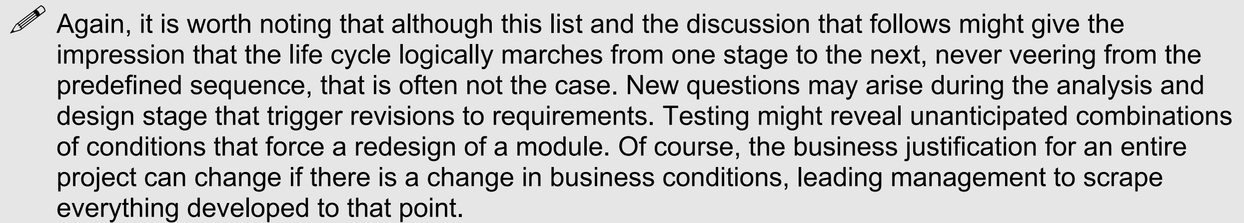
Application life cycle management entails how applications are created and deployed. Once constructed, or otherwise acquired, software applications are assets that must be managed as any other information asset. Of course, applications do not exist in a vacuum, and dependencies between applications must be understood to ensure they function properly. Another key to proper functioning is adequate security to protect the integrity of the application as well as the integrity and confidentiality of its related data. Finally, despite many differences with other kinds of assets, applications are assets and must be managed as such.

Application Life Cycles

The application life cycle is the series of steps an application goes through from the time a need arises for the application until the time the application is retired. The application life cycle is often complex with multiple possible paths through it. In cases of large, complex applications, different parts of the application may be in different stages of the life cycle. Without a doubt, managing the development of a complex application is one of the most challenging tasks in IT.

Part of the process of controlling that complexity is breaking down the development of the application into manageable, logical stages:

- Business justification
- Requirements gathering
- Analysis and design
- Development
- Testing
- Deployment
- Maintenance



Again, it is worth noting that although this list and the discussion that follows might give the impression that the life cycle logically marches from one stage to the next, never veering from the predefined sequence, that is often not the case. New questions may arise during the analysis and design stage that trigger revisions to requirements. Testing might reveal unanticipated combinations of conditions that force a redesign of a module. Of course, the business justification for an entire project can change if there is a change in business conditions, leading management to scrape everything developed to that point.

In addition, it should be noted that software developers use different methodologies for creating applications. Most of these methodologies use the stages described in this chapter in one form or another. The major differences in methodologies tend to focus on whether to use one or more passes through these stages and how much to try to accomplish at each iteration through the stages. (See the sidebar "Software Development Methodologies" for more information about this topic.)

Business Justification
Requirements
Deployment
Maintenance
Analysis & Design
Testing
Development

Figure 7.1: The dominant progression through the life cycle follows the solid lines, but in practice, there are many other paths through the life cycle as shown by the dashed lines.

The first step in the application life cycle is initiated by an organizational need.

Business Justification

Why would an organization commit resources—money, staff, and time—to developing or acquiring an application? There must be some benefit that outweighs the cost, of course. Sometimes an organization may make a decision to invest in the development of an application because the organization believes it will be a key to strategic success. Small startups can work like this. A few developers and managers with a vision for starting a new business can be justification enough. In larger organizations—such as midsized and large corporations, government agencies, educational institutions, and major non-profits—a more formal approach is usually required.

A business justification is essentially an argument for developing or purchasing an application because it will serve a need of the organization. These documents often include:

- A description of the current state of the business or organization and a missing service or unrealized opportunity.
- An overview of the benefits of implementing the proposed application, such as improved customer service, which leads to higher customer retention rates; reduced cost of manufacturing a product by eliminating older, higher cost IT systems used to support the current manufacturing process; or higher throughput of a transaction processing system, which will lower the marginal cost of each transaction.
- A formal assessment of the costs of the proposed project. Cost analysis often includes measures such as the return on investment (ROI) or the internal rate of return (IRR), which quantify the financial impact of the project and aid in allocating resources among multiple proposed projects.
- A discussion of the risks associated with a proposed project. Any projection, such as a business justification, is based on assumptions. The risk discussion points out what could go wrong and how those risks can be mitigated.

The business justification should also demonstrate how the proposed application will further align IT services with the strategic objectives of the organization. There are probably many applications that can be cost justified but still do not align with strategic objectives. The goal of deploying IT applications is to further the objectives that have already been defined; it is not to introduce side services that might generate revenue for the organization but distract from core services. Once it has been demonstrated that an application will serve the broader business objectives of an organization, the application project is formalized, and the requirements-gathering phase begins.

Requirements Phase

The purpose of the requirements phase of an application project is to define what the application will do. At this point, the question of how the application will operate is not addressed, that is left for the analysis and design stage that follows. The key topics that should be addressed during requirements gathering are:

- Functional requirements
- Security requirements
- Integration requirements
- Non-functional requirements

There is some overlap between these areas and requirements entail dependencies with requirements in other areas as well.

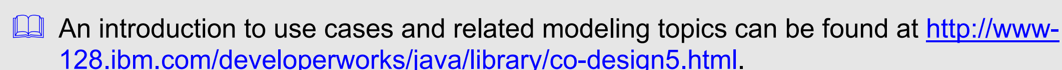
Functional Requirements

Functional requirements are composed of use cases and business rules. Functional requirements begin with the development of use cases, which are scenarios for how an application may be used. Use cases include descriptions of how users, known as actors, interact with the system to carry out specific tasks.

Uses cases typically include:

- A use case name and a version, such as "Analyze sales report, version 3"
- A summary briefly describing what the actor does with the system—for example, in the "Analyze sales report" use case, the actor might authenticate to the application, enter date and regional parameters, format data in tabular or graphical form, and sort and filter data as needed
- Preconditions (conditions that must be true for the use case to be relevant)—for example, a precondition of the "Analyze sales report" use case is that the data warehouse providing the data has been updated with the relevant data
- Triggers are events that cause the actor to initiate the use case; an event such as needing to calculate the distribution of inventory to regional warehouses will trigger the analysis of most recent sales
- Primary and secondary sequences of events within the use case—for example, the primary events sequence describes the typical steps to retrieving and displaying sales data, and the secondary sequence describes what occurs when an exceptional event occurs
- Post conditions describe the state of the system after a use case is executed; data may be updated, other functions may be enabled, and other use cases may be triggered

The purpose of use cases is to describe, in high-level detail, specific functions. The finer-grained details are captured by business rules.

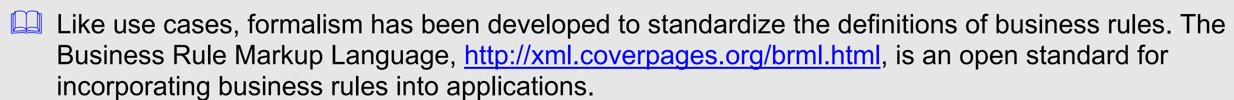


An introduction to use cases and related modeling topics can be found at http://www-128.ibm.com/developerworks/java/library/co-design5.html.

Business rules are formal statements that define several aspects of information processing

- How functions are calculated
- How data is categorized
- Constraints on operations

Business rules are specified early in the application development process because so much depends on them. For example, if a sales analysis system is proposed, it must be understood early on how to calculate key measures such as gross revenue, marginal costs, and related metrics. It must also be determined whether multiple definitions must be supported. Take marginal cost calculations, for example. The division responsible for manufacturing a product might include the cost of materials and equipment in the marginal cost calculation; whereas, the finance department might include those costs plus the sales commission paid to sell the product. This is an example of a single term meaning multiple things depending on the context.



Like use cases, formalism has been developed to standardize the definitions of business rules. The Business Rule Markup Language, http://xml.coverpages.org/brml.html, is an open standard for incorporating business rules into applications.

Security Requirements

Security requirements for an application should be defined along with functional requirements. Implicit in every functional requirement are the questions "Who should be able to use this function?" and "When can this function be used?" These broad questions, in turn, are answered by more precise, but not detailed, questions such as:

- In what roles will users be categorized? The privileges and rights to use the application and functions within the application should be based on the roles granted to users.
- How is the data used in the system categorized according to sensitivity classifications? Is it public data, sensitive information that should not be disclosed outside the organization, or private information whose distribution is controlled by the person it is associated with? Is it secret information, such as proprietary information, trade secrets, or comparable information?
- How will the application be accessed? Will remote users have access? If so, is it through a public Web site, a restricted Web site requiring a username and password, or through a VPN?
- What authentication mechanism is required to access the program? Is a username/password scheme secure enough? If so, what is the password policy? Is multi-factor authentication required? If so, what types of authentication mechanisms are required? These could include smart cards, challenge/response devices, or biometric devices.

The scope and detail of security requirements are slightly different from functional requirements. In the case of security, it is common to delve into the "how" instead of addressing only the "what." For example, the need for biometric authentication is really an implementation issue that would not be specified if it were a functional requirement. However, security requirements may be dictated by constraints outside the scope of the project. A financial services company, for example, may decide that to remain in compliance with government regulations, biometric security measures are required for all applications that reference customer account information. The designers of the application will have no choice in the matter; if the application they are developing accesses customer data, it is required to use biometric security measures. In cases such as this, it is important to document these requirements before the analysis and design phase begins.

Security requirements should also address:

- Compliance requirements
- Any restrictions on the time of access to the application
- How identities are managed? For example, will all users be registered in an organization-wide Active Directory (AD) or LDAP directory?
- The federation of identities (that is, relying on identity information managed by another party) if third parties are granted access to the application
- Encryption requirements, including the strength of encryption
- Policies on the transfer of data from the application. For example, can users download data and store it locally on their workstations? Can they store data on their notebook computers or other mobile devices?
- Any security policies and procedures that are relevant to the application

Security requirements sometimes have to address how the application will operate with other applications or data sources.

Integration Requirements

It is difficult to imagine an application that will not integrate with some other application or data source. Rarely are today's applications islands unto themselves. For this reason, it is helpful to understand the ways in which applications share services and data among themselves.

In addition to security issues, integration also requires a coordination of:

- Data flows; for example, if Application A generates data used by Application B, what are the rules governing how it will move from A to B? How often will data be moved? Will an event in A trigger the movement of data to B, or vise versa?
- Data exchange protocols; for example, must the application under design use an existing protocol to exchange data? If a detailed, application-specific protocol does not exist, is there a required method, such as Simple Object Access Protocol (SOAP), that will be the basis for integration?
- Support services required, such as services provided by a server OS, such as network file system (NFS) and file transfer protocol (ftp).
- Database-level integration, such as access to source systems on a mainframe database, relational database, or other repositories.

As with functional requirements, the goal of defining integration requirements is to identify what is required, not specify how it will be accomplished.

Non-Functional Requirements

The term *non-functional requirement* is a catch-all used to describe requirements that are not captured in the other categories. Some designers would include security and integration in the non-functional category; however, their importance and complexity is often far greater than the other non-functional requirements and therefore warrant a more detailed discussion. The remaining categories of non-functional requirements include:

- Backup and recovery
- Performance levels
- Service availability
- Service continuity

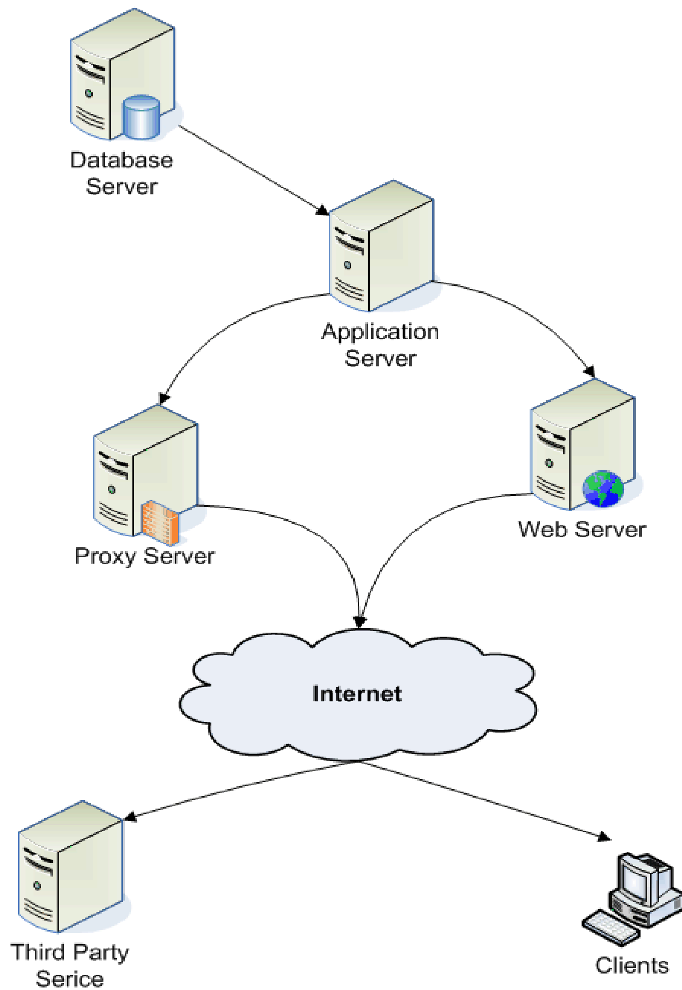
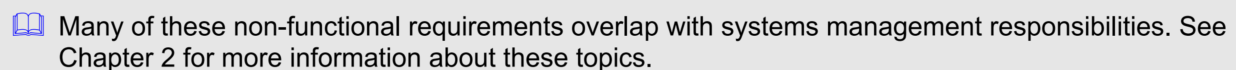


Figure 7.2: Example integration of application with other servers and services.


Many of these non-functional requirements overlap with systems management responsibilities. See Chapter 2 for more information about these topics.

Backup and Recovery

Backup and recovery requirements dictate the type of backups performed and their frequency. Full backups, which include all data and files associated with an application, may be combined with incremental backups, which back up changes since the last full backup or since the last incremental backup. Full backups take longer and require more media than incremental backups, but recoveries can be faster than if full and incremental are used. In theory, a single full backup followed by a series of incremental backups would allow a systems administrator to perform a full recovery. In practice, a cycle of one full backup followed by several incremental backups is more typical.

Performance Levels

Performance levels define the expected response times to users and the number of users that can be supported. This information is needed to size hardware appropriately. The number of CPUs, memory, and network bandwidth required will depend, in part, on the expected performance levels.

Service Availability

Service availability addresses the extent to which the application will be available. For example, mission-critical applications may be expected to be up 24 hours a day, 7 days a week. In practice, except for the most demanding applications, service windows are reserved for outside of peak operational hours to attend to upgrades, patches, and other maintenance. When true 24 × 7 service is required, servers are configured in a cluster or failover configuration that improves uptime and allows for a rotating maintenance schedule across the constituent servers.

Service Continuity

Service continuity requirements specify what is expected in the case of service disruption, such as a natural disaster that shuts down a data center. These requirements are dictated by the need to have the application available, the duration which the application can be down without impacting the organization's operations, and, of course, the cost of equipping and maintaining an off-site facility.

Gathering and defining requirements for applications is an essential step in the application life cycle. Functional requirements define what an application is to do, security requirements specify the level of confidentiality and integrity protection is required, integration requirements deal with how the application will function within the broader IT infrastructure, and finally the non-functional requirements define the parameters needed to support several core systems management services, such as backups and service continuity. Once the application requirements are defined, the life cycle moves into the analysis and design phase.

Analysis and Design

The analysis and design phase of application development marks the transition from describing what is to be done to defining how the application will accomplish its task. The steps in this phase can be broken down into three broad categories:

- Creating a framework of a solution
- Making build-vs.-buy decisions
- Defining detailed design

The phase begins by mapping out an overall picture of how the application will function.

Solution Frameworks

A solution framework is a high-level design of an application that describes the major modules within the application as well as the architecture that encompasses and integrates each of the modules. Although applications have different architectures, an increasingly common model is based on three or more tiers:

- Data services
- Application services
- Client services

Figure 7.3 shows a simple example of such a model.

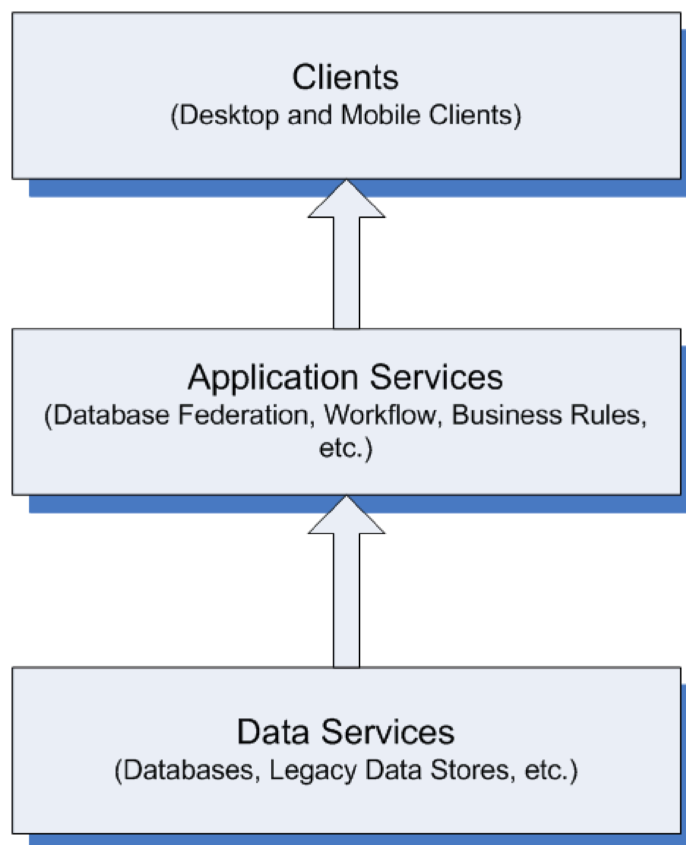
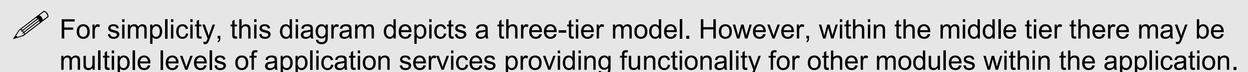


Figure 7.3: The multi-tier model is a common framework for applications.



For simplicity, this diagram depicts a three-tier model. However, within the middle tier there may be multiple levels of application services providing functionality for other modules within the application.

Data Services Tier

Data services are provided by one or more databases. The majority of databases in use today are relational databases. The relational model has proven to be the most effective approach for a wide range of applications, providing both a rich set of database management features as well as performance that scales well to multi-terabyte scales. Some applications still depend on older database models, including COBOL files and hierarchical databases, but these are usually associated with legacy applications. These are not common choices for new application development.

It is not uncommon for an application to use multiple databases for a single application. For example, data warehouses and other business intelligence applications often draw data from multiple source databases. For performance and ease of integration, data warehouses often depend upon copying data from source systems and storing it in a scheme more amenable to high-performance reporting (see Figure 7.4).

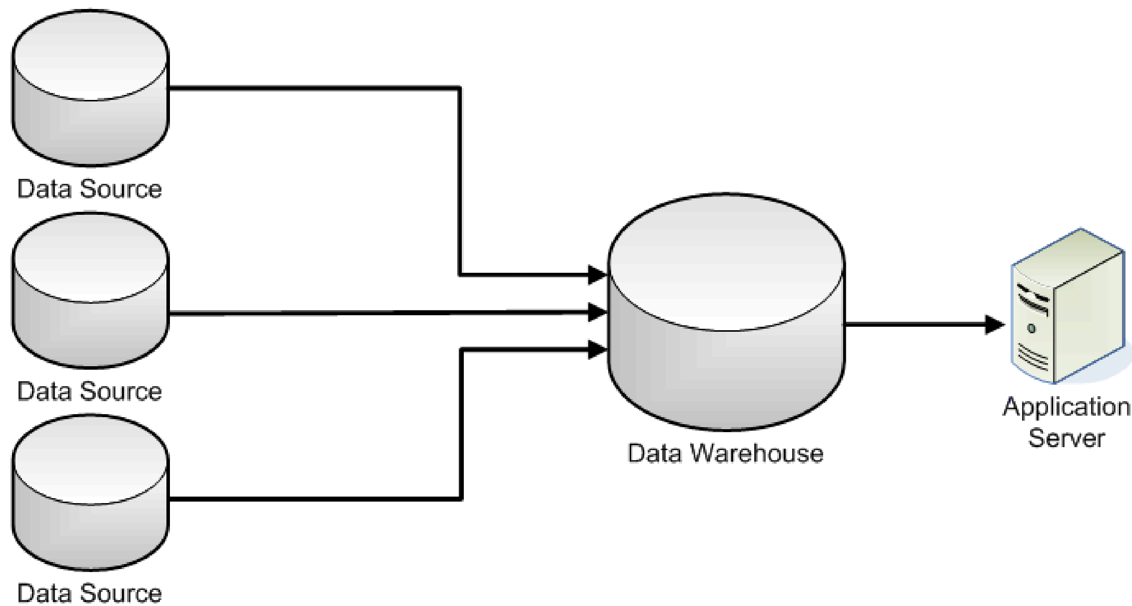


Figure 7.4: In the data warehouse model, data is first integrated in a separate data store and then processed by an application server.

Other systems, such as order processing systems, may use multiple, independent databases. For example, a financial services company may allow customers to access their checking accounts, mortgage statements, and credit card activity all from a single Web application. The data, however, is stored on three different systems, each one dedicated to managing one type of account (see Figure 7.5).

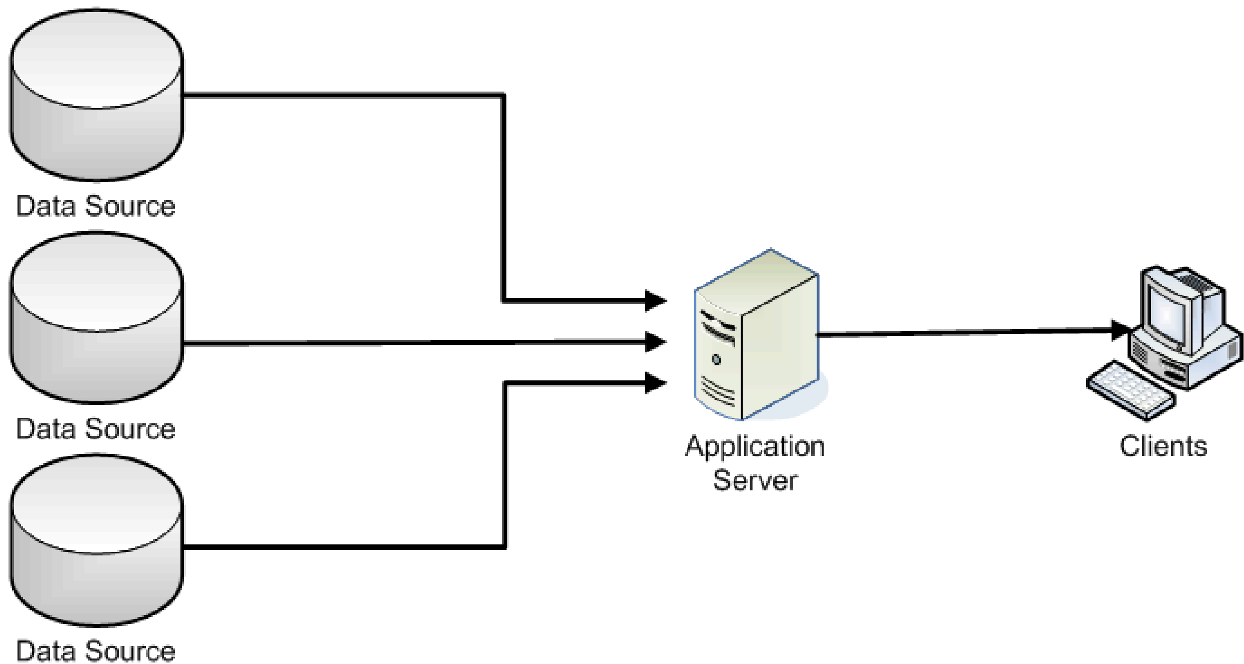


Figure 7.5: In many applications, multiple data sources are integrated directly in the application server.

During the framework modeling process, the source systems and how they will function together is determined. These data service providers are used by application services that occupy the middle tier of the architecture.

Application Services Tier

The application services tier is where the bulk of an applications work occurs. In any application that is more that a simple data storage and retrieval application, the middle tier is responsible for a wide range of functions, including:

- Integrating data drawn from multiple data sources
- Implementing workflows that dictate how data is processed
- Enforcing business rules, such as verifying credit status of a customer before completing an order
- Triggering complex events, such as ordering items when stock drops below a predefined threshold
- Providing an infrastructure to enable communications and coordination of multiple services, such as Web service functionality

The middle tier depends on a variety of infrastructure applications including:

- Web servers
- Application servers
- Database federation systems
- Portal frameworks
- Business rule engines
- Workflow engines

It is these components that manage the data and services that are used by the client layer to support interaction with users.

Client Tier

The client tier is responsible for rendering information provided by the data services and application services tiers. The client tier is becoming more challenging to manage and develop for as the options for clients expand.

Conventional workstations and notebooks are now complemented with PDAs and smart phones as application clients. This reality requires systems designers to develop for multiple platforms using multiple protocols. For example, HTML, the staple of Web application development, will not necessarily meet the functional requirements of mobile clients such as PDAs and smart phones; alternative methods are required.

Frameworks are skeleton designs of how an application is organized. It is at this point that systems managers can start to see how the application will fit into the existing network and server infrastructure, what additions will be needed to meet hardware requirements, and what additional loads will be put on network services. This is also the point at which decisions are made about which components of the application to build and which to buy.

Buy vs. Build

The buy-vs.-build question, as it is often framed, is something of a misnomer. The phrase implies a binary decision—either you custom build an application or buy commercial off-the-shelf software (COTS) or use open source software. In practice, this is not a black or white decision. Often there is a mix of some purchased and some customized development. It is useful to distinguish between different points in along a buy-vs.-build continuum.

The combinations of buying and building include:

- Buying a turnkey system using commercial or open source software
- Buying a turnkey system based on commercial open source software but customized by a third party
- Buying a commercial package or using an open source package and customizing it in house
- Buying components or using open source components to build a custom application in house
- Using commercial or open source tools to build the application from scratch

In practice, few organizations outside of software development firms will start with tools and build from scratch. Similarly, unless the application required provides a common, commodity service, such as a backup and recovery program, few organizations will avoid at least some configuration and customization of major applications.

The process of making the buy-vs.-build decision includes determining:

- The functional components of the application
- The communication protocols between the components
- The constraints on the application, such as the types of databases that will provide data to the application
- The development skills available in house, or readily available
- The time to deliver the application
- Options in the commercial market and open source for components
- Availability of subject matter experts who can support design

Ideally, the end result is a balanced approach that leverages existing components while reserving custom development for key components that add competitive advantage and cannot be adequately implemented using existing systems.

With tools and components identified, the detailed design can begin. The more components or existing packages are used, the less there is to design. At the very least, a detailed configuration of a turnkey system should be in place before the system is deployed to a production environment.

Detailed Design

The goal of the detailed design stage is to create a document suitable for programmers and systems administrators familiar with the selected tools and components to build the application. At this point, the requirements and overall architecture should be defined and the task is to identify how the requirements will be met.

In practice, designers will discover elements of the application that were not considered during requirements or find that requirements have changed (even with short development cycles, requirements can change before detailed design is complete). These discoveries can trigger review of functional requirements, non-functional requirements, and architectural design. These discoveries are so common that they have prompted the creation of several design methodologies. From a systems management perspective, this demonstrates that the supporting infrastructure originally planned for a new application deployment may not be what is actually deployed when the system design is finally completed. Once the design is in place, the application life cycle can move to the development stage.

Development

Development entails building applications and application components. Many books have been written on this subject and it is well beyond the scope of this chapter and this guide to try to address the practice of software engineering. There are, however, three topics relevant to systems management that are worth addressing:

- Source code management
- System builds
- Regression testing

Each of these entails software artifacts that, like other assets, require a structured management regimen.

Source Code Management

Source code for an application is often developed by multiple programmers over long periods of time. Source code developed for one project may be reused in another project, either as is or with changes to suit the needs of the application. Source code is a challenge to manage for a number of reasons, including:

- Software modules often depend on other modules that may be under development at the same time.
- No two developers should be allowed to change the same resource at the same time; check-in and check-out mechanisms are required to prevent the lost development work.
- A single module may have multiple versions for different platforms; for example, an interface designed for Windows may have to be re-written for a Linux platform.

Source code management systems are commonly used in software development efforts. These systems address the challenges outlined earlier and, like configuration management systems, become essential when systems reach a certain level of complexity.

System Builds

A system build is the process of gathering the component modules under development and creating an executable application. Once enough components have been developed to have even the most basic functions, system builds are used to ensure development continues in such a way as to not break (at least not too badly) previous work. A system build is a minimal test of the code under development. If an application's modules and libraries can be compiled into an executable application, the specific functions of the system can be tested.

Regression Testing

Regression testing is the practice of testing applications or modules after small changes to ensure that previously functionality components have not been broken by the introduction of bugs in new code. Regression tests can be automated and the results compared with previous results. This type of testing is not the full-scale system testing done prior to releasing a piece of software. Regression testing is often done automatically after building an executable application. When software is sufficiently constructed and tested by developers, it moves to the quality control–focused level of testing, typically carried out by a testing team that does not include developers.

Software Design Methodologies

Software design methodology is one of those topics that can trigger seemingly incessant debate among software developers. Over the past decades, a number of methods have been proposed, all with some variation on top-down or bottom-up design. Although there are a number of minor variations on the major models, we will focus only on the major ones, which are:

- Waterfall model
- Spiral model
- Agile model

The waterfall model is a linear approach to software development. According to the waterfall model, one starts by gathering requirements, then develops a high-level design followed by a detailed design, builds the code according to the design, tests it and correct bugs, and then deploys it. The advantage of this model is that it is intellectually simple and easy to understand. The disadvantage is that it does not work in most software development projects. The world does not proceed in the lock-step fashion assumed in the waterfall model. Requirements change and this model does not adapt to that. The spiral method was developed to avoid the fatal flaws of waterfall while maintaining the structured approach that does serve the goals of software engineering.

Through the spiral approach, developers build software iteratively and assume that requirements will change and that during the process of developing a system, new information is gleaned that will help in the development of other parts of the system. Rather than build an application in one pass through the structured stages, spiral methodologies build a set of functions in each iteration through the structured stages.

In theory, proponents of waterfall methodology might argue that a skilled requirements gatherer could find all the requirements early in the development cycle. Even if someone did have the mythic skills to elucidate all the requirements in the precise detail needed, this does not account for the cost of gathering those requirements. It is a well-known principle in economics that the cost of producing one more item may not be the same as the cost of producing the previous item. In the case of gathering requirements, the marginal cost, as it is known, of getting one more requirement begins to increase at some point. In some cases, users may not know their requirements until they have had a chance to interact with the application.

Agile software development methodologies take the spiral approach to an extreme and use very short software development cycles—as short as several weeks. This allows for almost constant evaluation and quick adoption.

Software Testing

Software testing is a quality control measure that is designed to identify bugs in software (similar to regression testing) and to ensure that all functional requirements defined in the earlier stages are met by the software. The testing at this stage is integrated testing that exercises the full functionality of the application. Unlike the testing done by developers, which is referred to as unit testing, the goal with integrated software testing is to make sure the application's components function correctly together.

The artifacts used in integrated software testing are:

- Test plans
- Test scenarios
- Test procedures
- Test scripts

A test plan is a high-level document describing the scope of software testing and usually includes:

- Functions to be tested
- References to requirements documenting functional requirements
- Known software risks
- Test criteria
- Staffing and resource requirements
- Schedule

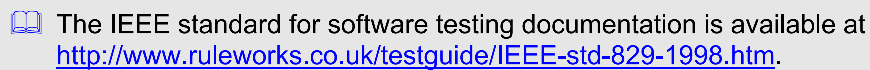
The details of how functions are tested are included in test scenarios and test procedures. Test scenarios describe use cases and specific features within those use cases to test. For example, a scenario may describe a user retrieving a sales analysis report, entering search criteria for filtering data, and exporting data to a spreadsheet. The test procedures define the steps carried out by the tester to test each function. For example, to export the data to a spreadsheet, the tester will select "Export" from the menu, enter file name "Test123.Xls," save the file, then open the file in a spreadsheet program and verify that the table headings, summary data, and formatting are correct.

Testing can be a time-consuming and tedious task, especially when large numbers of functions must be tested. Test scripts can be used to automate this process and a number of tools are available.

In addition to testing basic functions, which presumably was done during unit testing in the development stage, systems integration is also tested. Applications will depend upon other applications and while application programming interfaces (API) may be well defined and used properly by client applications, there is more to testing integration than simply making sure a single API call works correctly. Integration testing should include testing:

- Scalability of calls from the client application
- Consistent security between the applications
- The ability to roll back a transaction across multiple applications

These are the types of non-functional requirements that are not tested in unit testing and must be explicitly planned for in integration testing.


The IEEE standard for software testing documentation is available at http://www.ruleworks.co.uk/testguide/IEEE-std-829-1998.htm.

As the application, or in the case of large, multi-phase application developments modules, passes integration testing, the application is moved to production through the deployment process.

Software Deployment

The process of software deployment is complex because of the dependencies between so many aspects of information architectures. Release management, as the practice of controlled software deployment is known, consists of a number of tasks, including:

- Coordinating with testers to ensure software is ready for deployment to particular platforms
- Packaging software for installation on target platforms
- Determining dependencies for successful installation of software
- Receiving change control approval to deploy software
- Updating the configuration management database to reflect the new versions of the software on particular platforms
- Planning the deployment so as not to disrupt operations or at the very least, to minimize the impact of the deployment

In addition to coordinating the installation of software, the release management team must coordinate with developers and trainers to ensure that end users, systems administrators, and support personnel are all trained on the new software. The deployment phase in many ways marks the final state of the software development life cycle because after that software is actually in use. It is not truly a terminal state, though, because maintenance is such an important factor in the life cycle.

Software Maintenance

Software maintenance is the practice of making modifications to applications to ensure that they continue to meet functional and non-functional requirements and do not present security vulnerabilities that could compromise the integrity and confidentiality of information or the availability of the system itself. Software maintenance usually comes in the form of patches and upgrades.

Patches are usually small changes to code to correct a known problem. They do not provide additional functionality. Upgrades, in contrast, are designed to enhance the functionality, performance, or scalability of an application.

Another distinction between patches and upgrades are the timeframes for deploying them. Patches may be provided by application developers as soon as a problem is discovered, especially if the flaw results in a security vulnerability. In these situations, systems administrators may have less time to test and apply a patch. For example, if fast-spreading malware threatens an application and a newly released patch is available from the application vendor, the systems administration team may deploy the patch with minimal testing. Upgrades are usually well planned and both the application developers and application users have time to properly plan their deployment.

Role of Application Development Life Cycle in Systems Management

In many ways, applications are like other assets managed and tracked by systems managers. They have acquisition processes, they are deployed in a controlled manner, they are subject to change control, and applications or their components are tracked as configuration items in the configuration management database. At first glance, applications appear to be managed not all that differently than other assets, but that is not the case.

Applications are subject to changing requirements that in turn are driven by changing business conditions and strategies. Applications may be finely customized to the needs of a particular organization to a far greater degree than other assets, such as networking devices or servers, can be configured. The flexibility one has in designing software is one of its advantages. This flexibility brings with it an added level of complexity not found elsewhere.

From a systems management perspective, one does not manage an application but manages an application in multiple states at the same time. One also manages a host of secondary artifacts, such as design documents, test plans, requirements, and patches that are all part of an application. Applications are not just executable files and scripts residing on a server but include the full range of activities and artifacts that support the application through its life cycle. Another aspect of applications that is relevant to systems management but not tied to a single application are the dependencies between applications.

Managing Application Dependencies

Applications exist in something akin to a software ecosystem. Applications use functionality provided by OSs, network services, and other applications. This use of other components outside the control of an application, or at least an application development group, leads to a number of different types of dependencies, including:

- Data dependencies
- Time dependencies
- Software dependencies
- Hardware dependencies

Disruptions in any of these dependencies can cause ripple effects throughout an application.

Data Dependencies

Data dependencies occur when one application depends on another to provide specific data at certain times. There are many factors of data dependencies to consider but from a systems management perspective, a key question is, At what point does an application failing to meet its requirement to provide data begin to adversely impact operations? Consider some examples:

- The enterprise resource planning system of a retailer with 200 stores nationwide aggregates sales data from stores each night. All stores are expected to provide data by midnight (headquarters time zone). If more than 5 percent of the stores fail to provide data or three or more stores in the same region fail to upload data, summary reports cannot be generated.
- An order processing system depends on an inventory system to check stock levels before committing to a delivery date. If the inventory system is offline, the order processing system estimates the delivery date based on the location of the customer and proceeds with the order.
- A data warehouse draws data from several systems, integrates the data in an enterprise data warehouse and then populates a series of data marts targeted to particular analysis functions. In the event some data sources are down, the data warehouse load continues but the data warehouse generates only those data marts for which all data has been received.

Clearly, data dependencies are not "all or nothing" affairs. Well-designed applications degrade gracefully. If partial data is available, then partial functionality and services should be available. Systems managers should design and manage infrastructure in such a way to support data dependencies; to do so they must have insight into not only the requirements but also the capabilities of applications with respect to data dependencies.

Time Dependencies

Time dependencies are an important factor in application management. In some cases, these are essentially questions of scalability. For example, an online order processing system may be able to take as many as 1000 orders per minute, but it depends on a service provided by a sales tax computing Web service that can only process as many as 500 orders per minute. Systems managers can work with developers to improve on this by dedicating additional servers to the Web service once the dependency has been identified.

Another type of dependency is more difficult to work around. It is not uncommon for large, centralized applications to do quite a bit of batch processing outside of business hours. Banks, for example, will post transactions against accounts and process loan payments during off hours. Because transaction processing systems are subject to heavy transaction loads, this is also the ideal time to accomplish tasks that would put an inordinate load on the application during normal business hours. Data extractions, for example, often occur at these times. The problem is that there are often several or more data extraction jobs that need to run in a limited time window. Understanding these requirements is important for system managers so that they can arrange jobs and allocate resources appropriately to meet the requirements of these non-transaction processing requirements.

As with other performance measures, it is important for systems managers to track trends in non-transaction processing. For example, if batch jobs are taking longer and longer to run, are some critical processes running over into normal business hours and therefore potentially interfering with core business operations? If so, how can the current configuration of hardware, software, and batch jobs be reconfigured to eliminate the problem? The answer to this question requires detailed information from a variety of sources including system logs and the configuration management database.

Software Dependencies

Software dependencies are another type of dependency that should be explicitly managed. Successful change management procedures depend upon knowing the dependencies between applications so that a functioning system is not inadvertently disrupted by a change in some dependent code. Tracking dependencies explicitly in a configuration management database can help to minimize the chances of that kind of mistake. This is just one of the reasons that software should be managed like other assets.

Hardware Dependencies

Applications are deployed to particular servers that have specific configurations. The dependencies between applications and the hardware configuration required to support them should be explicitly modeled. At any time, a systems administrator or IT manager should be able to report on the details of which applications are running on the various servers in the organization.

Application Asset Management

The software development life cycle is a major process in the management of applications, but it is not the only one. Application assets, including the hardware required to use those applications, also entail an asset management process. There is some overlap between the software development life cycle and the asset management process, but it is still worth outlining the key elements.

Acquiring Assets

Acquiring assets and planning for their integration and deployment may depend heavily on the software development life cycle if the asset is built. Regardless of whether an application is built or bought, the acquisition process is dominated by:

- Functional requirements
- Compatibility with architecture
- Capacity planning

Functional requirements have been detailed earlier in the chapter. Compatibility with architecture is another factor that can limit an organization's options when it comes to acquiring assets. Although shared standards allow virtually any major platform to inter-operate, the cost of supporting multiple architectural models and platforms may be cost prohibitive. An architecture based on J2EE standards, for example, can function with .Net applications but the additional effort to deploy and maintain multiple architectures may outweigh the benefits.

Capacity planning must also be considered when acquiring assets. Factors influencing the capacity of an application include:

- Number of users
- Peak load periods
- Time dependencies on other applications and data sources
- Expected growth rates

Availability requirements should also be considered in capacity planning. A clustered configuration of servers, for example, could improve both availability and capacity.

Deploying, Managing, and Decommissioning Applications

Deploying and managing applications as assets follow similar patterns to managing other assets. For example, for complex applications, systems managers must find the appropriate level for defining configuration items in the configuration management database. Should the application be defined as a single item? Should each module? What about software libraries that the application depends on? The answers to these questions is largely influenced by how tightly coupled particular modules are. For example, if a financial reporting module within an enterprise resource planning system changes more frequently than other modules and has different licensing requirements, then it should be tracked independently.

The usefulness of an application, like hardware assets, will come to an end at some point. When this happens, several events should occur:

- Application should be taken offline
- Enterprise directories with application information should be updated
- Hardware should be decommissioned or reallocated
- Leases for associated hardware and software should be closed
- Configuration management database should be updated

When managing applications, the software development life cycle entails complex processes that can be especially challenging from a management perspective.

Summary

Managing applications is a process with characteristics not found in other areas of asset management. The dynamics of the software life cycle introduce additional artifacts that must be managed, such as requirements documents, code libraries, and test cases. Applications themselves are more dynamic than many other assets and this, in turn, creates more work to keep configuration management databases up to date and accurately reflecting the state of deployed applications.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.