

Johnson County

Information Technology Services

Application Architecture Guidelines

Table of Contents

Table of Contents 2

Application Development Guidelines 3

 Part One: Fundamental Principles 3

 Principle 1: Data is a county asset..... 3

 Principle 2: Applications should work..... 3

 Principle 3: Applications should be maintainable..... 4

 Part Two: Best Practices 5

 Best Practice 1: Applications should be developed through the use of a formal methodology..... 5

 Best Practice 2: Design for manageability..... 6

 Best Practice 3: Design for supportability..... 6

 Best Practice 4: All development efforts should be peer-reviewed..... 6

 Best Practice 5: Architect applications to mimic the way business is done..... 6

 Best Practice 6: Organize external applications around "life events." 6

 Best Practice 7: Re-use before buying; buy before building..... 7

 Best Practice 8: Application platforms and development tools should be based on industry standards..... 7

 Best Practice 9: Design for future enhancements..... 7

 Best Practice 10: Generalize application interfaces..... 7

 Best Practice 11: Applications should employ object-oriented principles..... 7

 Best Practice 12: Design applications that are platform independent..... 7

 Best Practice 13: Design for the n-tier service oriented architecture..... 7

 Best Practice 14: Implement Business Rules As Discrete Components..... 8

 Best Practice 15: Minimize platform configurations..... 8

 Best Practice 16: Assign responsibility for business rules to the business units that use them..... 8

 Best Practice 17: Utilize trouble-ticket tracking to identify candidates for redevelopment..... 8

Application Architecture..... 9

 N-Tier Application Architecture Model..... 9

 N-Tier Implementation Alternatives 11

 Browser-based Applications 14

Toolset Recommendations..... 15

 Legacy Tools..... 15

 Current Tools 15

 Future Tools 16

Data Base Sizing..... 17

 Access 97/2000 Quick Reference 17

 SQL-Server 97/2000 Quick Reference 18

Application Development Guidelines

Part One: Fundamental Principles

Principle 1: Data is a county asset.

Like land, equipment and buildings, data is a valuable commodity. And as are other more tangible commodities, the data collected and maintained by the county should be treated as an asset with tangible, measurable value. Data is extraordinarily costly to replace if damaged or lost. Furthermore, business decisions based on incomplete or incorrect data can be expensive and even catastrophic. The observation that data is a valuable asset has two important ramifications.

- **Data should be shared.**

Data collected and maintained by the county should be available to all applications that require it. This does not mean that all data should reside in one central, universally accessible data store: it simply means that no one department or agency within the county should have the ability to “hide” data from other departments or agencies that need (and are allowed) access to the data.

For this reason, it is strongly suggested that no data should ever be compiled into programs nor stored on a single client workstation without compelling justification.

- **Security, confidentiality and privacy of data should be ensured.**

As with any other valuable asset, care should be taken to ensure the safety and security of data. Backup and restoration of data should be a regular, routine and planned activity. All data should be evaluated for confidentiality and privacy concerns, and sensitive data secured from unauthorized use and tampering.

Principle 2: Applications should work.

Good software has two necessary and sufficient attributes. It must 1) work (i.e., it must accomplish the tasks defined by its requirements within the physical environment for which it was designed) and 2) be maintainable (i.e., it must be easy to modify and enhance as its logical and physical requirements change). Of the two attributes, the latter is the one that most often fails. Most software developers are able to achieve working code. They may, however, fail to do so in a manner that leaves the code easily maintainable.

In order to ensure working applications, the following guidelines should be followed.

- **Applications should be complete.**

In addition to having all necessary functionality to meet the requirements defined for the application, the application must also have available all the necessary supporting documentation to ensure that it keeps working over time.

- **Applications should be well documented.**

This includes both information documenting the requirements and the design of the application, as well as the information documenting its testing, its support and its operation.

- **End-users should be able to carry out all routine application functions.**

Another important aspect of application quality is the complete implementation of common functions. An end-user should not have to involve ITS personnel to accomplish a routine task, regardless of whether that task is daily, weekly, monthly or yearly. Applications that require such intervention by ITS should be considered incomplete.

- **Applications should be robust.**

A robust application is one that handles adversity gracefully. In other words, an application should not be overly fragile and should not fail unpredictably when it encounters unexpected data. When a program fails, the failure should be immediately reported to the end-user (and possibly ITS) in a clear, concise and obvious manner.

- **Business processes should be re-engineered before they are implemented.**

Applications should not implement outdated or ineffective business functions. Business processes should be optimized and streamlined before an attempt is made to implement them as an automated application.

Principle 3: Applications should be maintainable.

Applications are not static: they must change and grow over time. Applications that are not developed with maintainability in mind are difficult to enhance and become even more difficult with subsequent enhancements. Consider the following: enhancements (if not carefully integrated into the existing code) invariably increase the software's signal-to-noise ratio (the signal being the original code and the noise being the changes). In other words, enhancing a messy program usually results in a messier program and not a cleaner one. This effect has one very profound implication on software enhancement: the software that is enhanced most often is invariably the software that is the most difficult to enhance.

There are many, many guidelines that can be documented in an attempt to improve maintainability: coding standards, naming standards, development standards, etc. Rather than repeat a list of standards with which all software developers should be familiar, we will instead document some broad guidelines for ensuring application maintainability:

- **Code should be created from designs; designs should be developed from requirements.**

Code should never be the only artifact remaining as the result of the application development process. Having code without design documentation and/or without requirements documentation inherently results in poor maintainability.

- **A modified application should have the same quality as the original application.**

Changes to application requirements should be integrated into the existing requirements and designs, not just the code. You should not be able to tell from inspecting an application whether or not it has ever been enhanced.

- **Software coding styles should be standardized for the organization.**

Having a wide variety of coding styles present in applications is detrimental to long-term maintainability. Ideally, you should not be able to tell who wrote a given application simply by inspecting the style of coding.

Part Two: Best Practices

Best practices are those that have been proven successful at other organizations that have similar goals and objectives. There are a number of industry best practices that should be incorporated into application development efforts.

Best Practice 1: Applications should be developed through the use of a formal methodology.

A standard waterfall-type methodology or a modified waterfall methodology incorporating prototyping, rapid development and incremental redevelopment should be employed to develop applications. Regardless of the methodology used, certain guidelines should be followed:

- **Applications should be designed from requirements.**

Requirements for an application should be defined before the application is designed. Requirements fall into two categories: logical requirements, which define the functionality of the application, and physical requirements, which define the hardware and software constraints under which the application must operate.

- **Code should be constructed to match designs.**

Designs should be developed from the requirements as a means of partitioning data and processing into components that can be constructed and tested. The packaging of code functionality, the establishment of components and the packaging strategy (monolithic, 2-tier client/server, 3-tier client/server, browser-based, etc.) is all a function of the design stage.

- **Designs should be created with an eye towards ease of testing.**

Applications should be designed so that they are easy to test and debug.

- **Testing should validate that the actual behavior of the application meets the required behavior.**

Testing has two major components: Prior to testing, the test cases and test results expected should be established and documented. After testing, the actual results should be compared to the expected results to see if the software performed as expected.

- **Documentation should be developed as a natural by-product of the development methodology.**

If done correctly the application's requirements, designs, test plans and operating instructions are all developed as a normal part of creating the application. They are not an afterthought nor done as part of a separate "documentation" phase after an application has been developed.

Best Practice 2: Design for manageability.

Applications should be designed so that they can be easily managed by ITS. That is, the applications should be easy to roll out, update, backup, restart, restore, configure, secure, install, remove or otherwise manage by the development and technical support groups.

Best Practice 3: Design for supportability.

Applications should be easy to support. Reducing the time and effort required to provide end-user support for an application reduces its total cost of ownership and greatly increases end-user satisfaction.

Best Practice 4: All development efforts should be peer-reviewed.

It has been repeatedly shown that peer-reviewed applications are of vastly higher quality than applications created by only one individual. Being human, we all make mistakes. Peer-review adds at least one other pair of eyes as a reality check to ensure that requirements, designs and code are without obvious flaws.

Best Practice 5: Architect applications to mimic the way business is done.

Interfaces between applications and components should mirror the business interfaces. A single component within an application should not contain business functionality from more than one business unit. By isolating business functions inside discrete application components, changes to the business environment requiring changes to software can be easily located and easily implemented.

Best Practice 6: Organize external applications around "life events."

Applications intended for use by residents of and visitors to Johnson County should be organized around major "life events" rather than around organizational boundaries. A "life event" is some important happening in the life of a resident or visitor that would cause them to want to interact with the County, such as a marriage, a vacation, the sale of a car, the purchase of a house, etc. Organizing an application around the purchase of a home, for instance, is preferable to organizing applications around the various County agencies or departments for which the purchase of a home is but one of many events in which they are involved.

As a corollary to this best practice, we observe the following:

- **Data should be collected only once.**

A resident or visitor to a Johnson County application should not be required to enter a given piece of information more than once during his interaction with the County. If an application needs information that the user has already provided to the County, the application should use the previously provided information rather than ask the user to enter it again.

Best Practice 7: Re-use before buying; buy before building.

Custom applications built from scratch are expensive to create and maintain. Therefore the development of a custom application should be the last option taken. If an application is similar to one already in use, the existing application should be reused. If there is no existing application that can be reused but there is one that can be purchased as an off-the-shelf item from a 3rd-party vendor, it should be purchased.

Best Practice 8: Application platforms and development tools should be based on industry standards.

Application platforms and development tools should, wherever possible, adhere to mainstream industry standards. Replacement of components and the addition of new components is less costly, as is training and development. Niche platforms and tools can often be difficult to obtain, costly to support and may reduce critical choices down to a single vendor.

Best Practice 9: Design for future enhancements.

Rarely are applications developed once and never enhanced. Designing applications with an eye towards future modifications and enhancements ultimately reduces costs and increases flexibility. That being said, to design anticipating a *specific* future enhancement is not a best practice: if the future enhancement does not happen as anticipated the application may be much more difficult to modify than one that provided for more general, flexible enhancements.

Best Practice 10: Generalize application interfaces.

Applications should seek to generalize the input collected and output produced to a non-specific device. A presentation layer should format incoming and outgoing data for a particular device. Thus, the same application could interface with, for instance, both a terminal and a WAP-enabled phone by simply utilizing two different presentation layers.

Best Practice 11: Applications should employ object-oriented principles.

Application development should employ object-oriented principles. Both the Microsoft world (in the .NET environment) and the non-Microsoft world (Java) are strongly object-oriented. Development in non-OOP languages (such as Visual Basic 6.0) may still employ many of the important characteristics of object-oriented development such as encapsulation, information hiding and development of modules based on business-oriented objects such as land, people, vehicles, buildings, etc.

Best Practice 12: Design applications that are platform independent.

Applications should be developed in such a way that the functionality of the application is separated from the behavior of the physical platform(s) on which it is implemented. In other words, behavior that is specific to a particular type of hardware device should be encapsulated into a layer of its own and should employ an interface with the application, rather than have the application aware of and in control of the physical device.

Best Practice 13: Design for the n-tier service oriented architecture.

Applications designed around an *n-tier* architecture are easily modified to support changes in business rules and are highly scalable. In addition, the n-tier architecture offers high performance over other architectures. It allows for maximum flexibility for various input and output devices on the presentation side and maximum flexibility for data storage on the data base side. Also, n-tier solutions are highly amenable to re-use by multiple applications.

Note that the n-tier application architecture strategy is not confined to a single type of physical implementation: n-tier solutions can be implemented as monolithic (stand-alone) applications as well as 2-tier client/server, 3-tier client/server, browser-based or any other type of physical distribution of processing.

Best Practice 14: Implement Business Rules As Discrete Components.

As a means of encapsulating behavior and making enhancements easier, business rules should be implemented as separate components. In other words, a change to a business rule should only require a change to one component of an application that uses the rule. Two corollaries to this rule are as follows:

- **Access data through business rules.**

Data should be obtained through component layers that understand the business rules relating to the data. Uncontrolled access to data directly through an application is more difficult to maintain and more likely to cause errors. Simply put, instead of reading and writing data, an application should call components that know how to read and write data: there should be only one read statement and one write statement for each physical record in an application.

- **Make business rule components platform-neutral.**

As with applications in general, business rules should be platform-neutral whenever possible.

Best Practice 15: Minimize platform configurations.

It is unlikely that any organization will be able to achieve and maintain a completely homogenous computing environment. However, minimizing the number of different hardware platforms and the number of software environments will minimize the cost associated with the development and maintenance of applications.

Best Practice 16: Assign responsibility for business rules to the business units that use them.

There is a direct correlation between quality of business processes (and business data) and frequency of use. Rules (and data) that are used frequently are more likely to be correct than rules (and data) that are rarely used. Therefore the business units that use the business rules should define and maintain the rules, not developers or other parties that rarely or never actually *use* them.

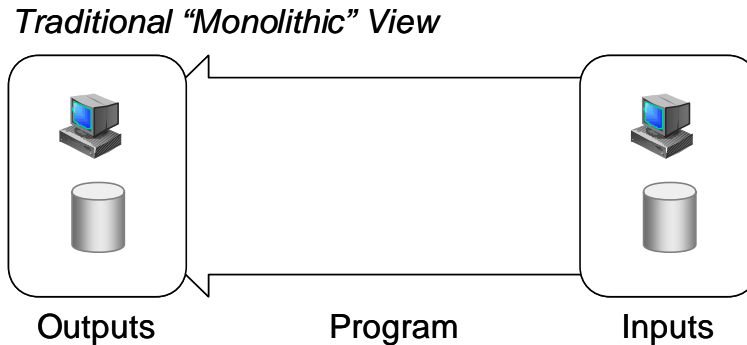
Best Practice 17: Utilize trouble-ticket tracking to identify candidates for redevelopment.

Maintaining a log the applications that are being regularly enhanced will give the organization much better metrics on which applications would benefit from radical reengineering and/or redevelopment. Applications that are consuming large resources due to enhancement requests, trouble tickets and expenditure of other development-related resources for routine functions should be considered strong candidates for replacement.

Application Architecture

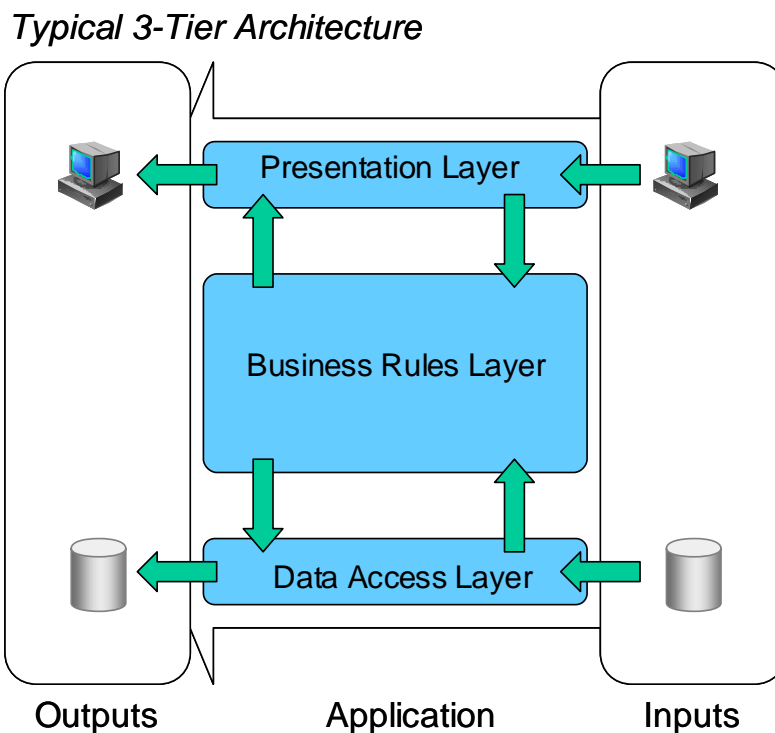
N-Tier Application Architecture Model

Traditionally, an application is thought of as a program or a series of related programs that act together to accomplish a given task. This traditional or “monolithic” view of an application is represented below.



In this traditional view, an application is viewed as a mechanism that converts inputs into outputs via some processing rules. The internals of the program are viewed as a black box, with no clear delineation of any identifiable subdivisions of processing.

Rather than view applications as one large lump of code, a generalized version of the functionality embedded in an application can be described as follows.

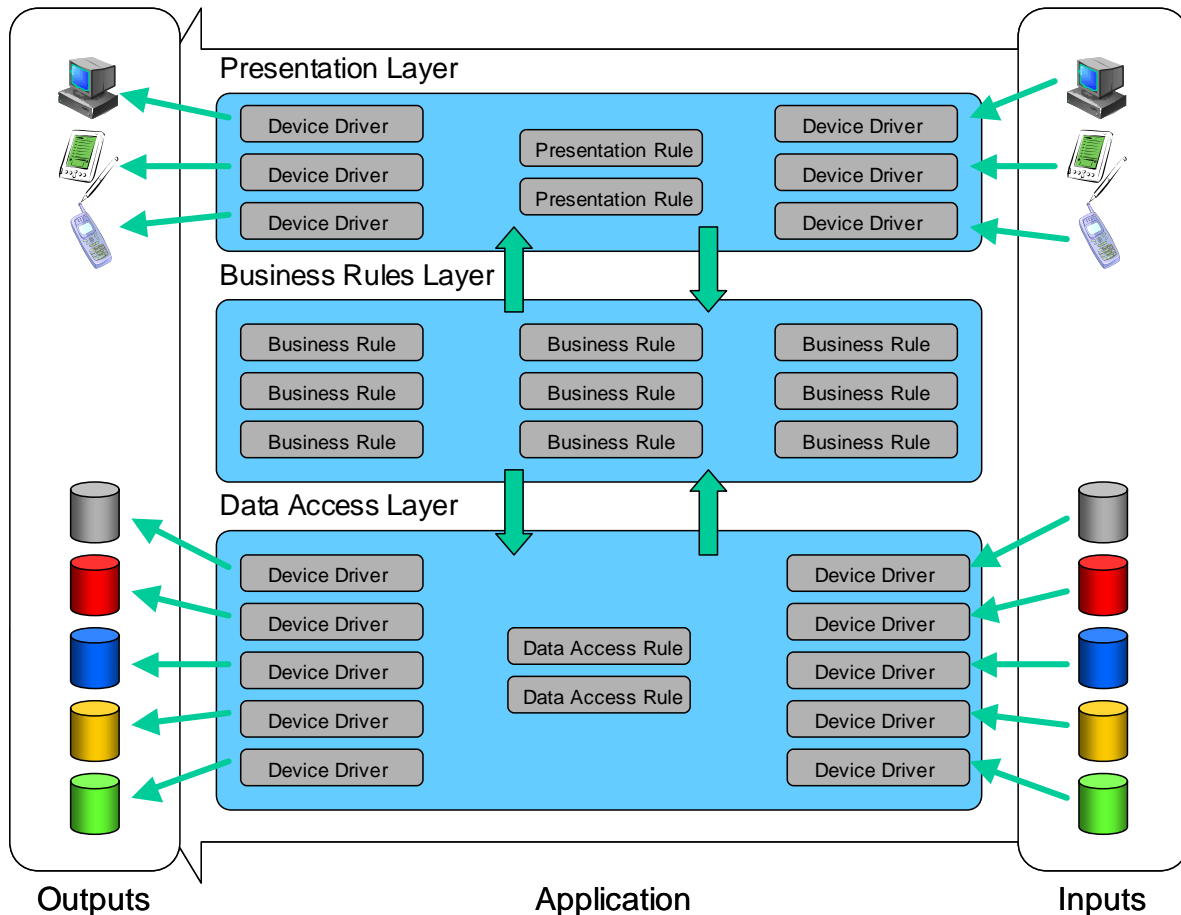


In this view the application is partitioned into three distinct areas: a Presentation Layer containing code for obtaining input and formatting output for the end-user device; a Data Access Layer containing code for reading and writing data to a data store; and a Business Rules Layer containing code for transforming the input data into the output data. This is typically thought of as a *3-tier* application architecture.

The advantages of a 3-tier architecture over a traditional monolithic architecture are numerous. To begin with, it implements a key concept of both structured programming and object oriented programming known as “information hiding.” It allows the application developer to “hide” information about the true input devices and output devices from the main processing (the business rules) portion of the program. This limits the impact of change: if the input or output devices change, changes to the program are only required in the area of the presentation layer; if data access methods change, changes to the program are only required in the area of the data access layer.

This 3-tier model can be generalized even further.

N-Tier Architecture



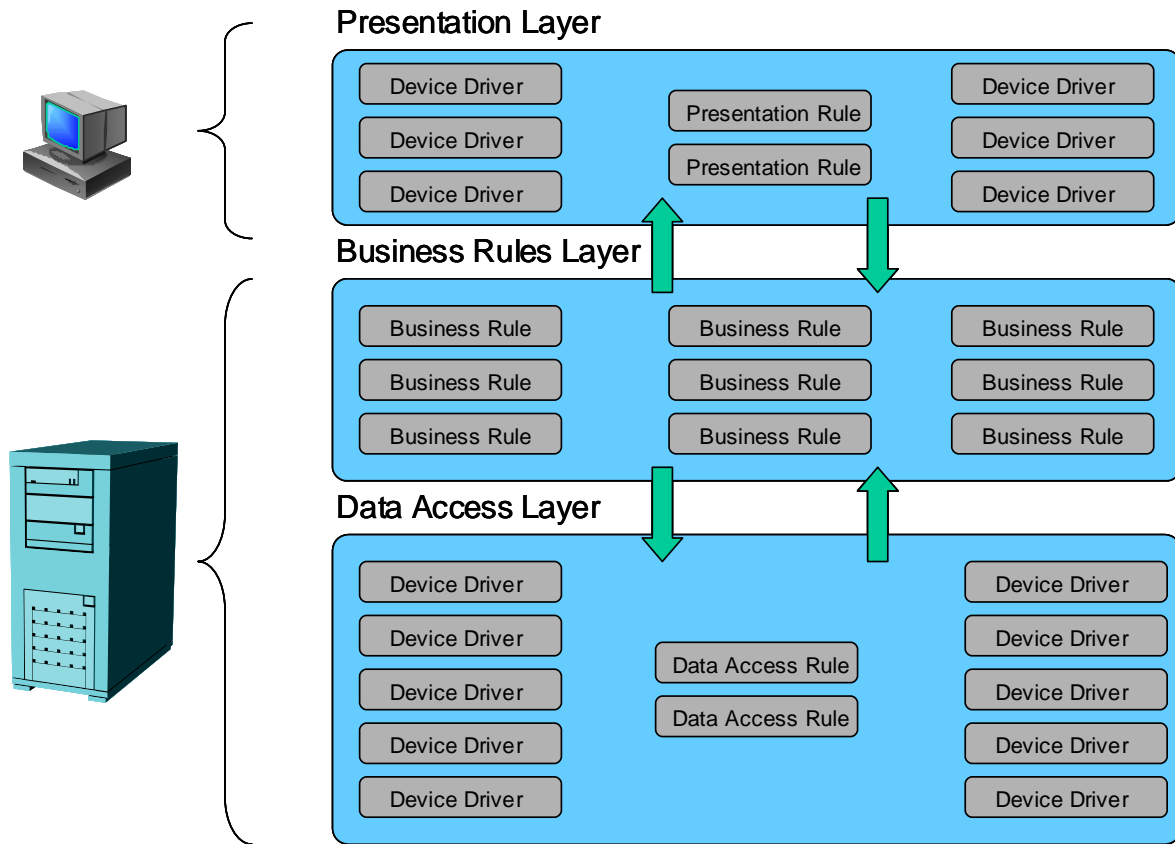
In this generalized “*n-tier*” architecture, the three tiers seen earlier are further subdivided and refined. In the presentation layer you will find compartmentalized device drivers for each type of

input and output device in addition to code relating to presentation rules for generalized input and output devices. Similarly, the data access layer contains compartmentalized code for each different type of data store that must be accessed as well as code relating to generalized data access rules. The business rules layer contains compartmentalized code segregating processing by business unit or business rule. The business rules can also be thought of as “filters” through which data is parsed in order to transform input data into output data. A business rule may contain an edit routine, a search routine, an update routine, a calculation routine, etc.

N-Tier Implementation Alternatives

The n-tier application architecture allows for a tremendous amount of component reuse. Further, it allows for a wide variety of physical packaging options. Clearly, it can be implemented as a stand-alone program on a client workstation, as is indicated in the last figure. It can also be repackaged across multiple processing devices.

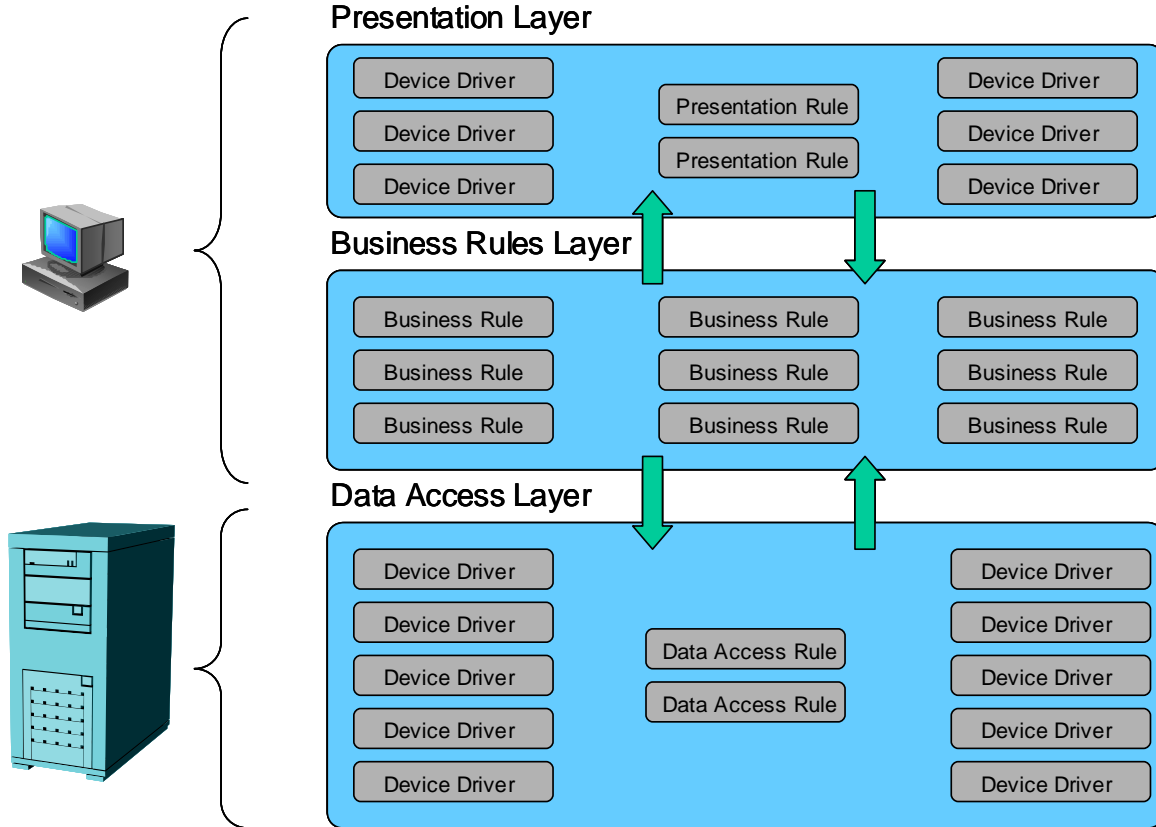
“Thin Client / Fat Server” implementation



In the “thin client / fat server” implementation, the client workstation contains little processing other than that required to collect input and display information. A traditional “browser-based” application generally follows this implementation model: the client workstation handles the

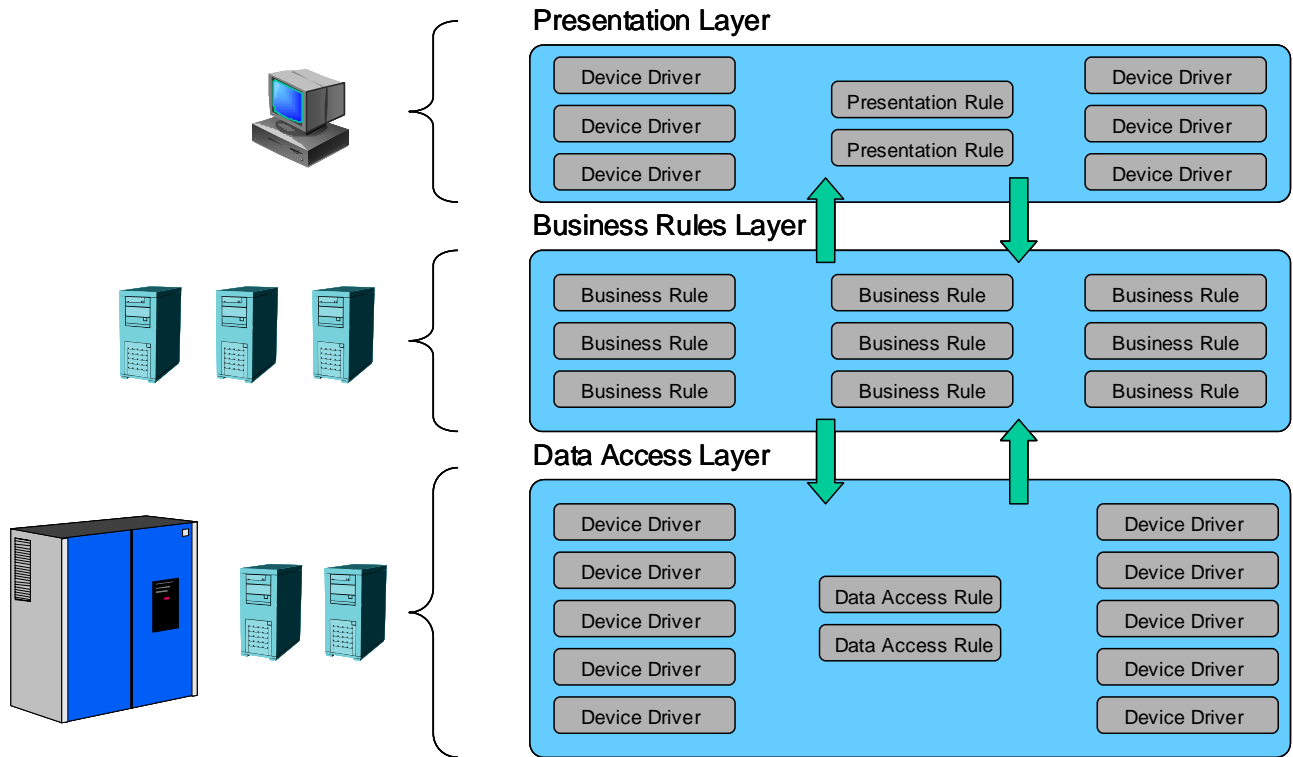
presentation layer through the web browser, with the business rules and data residing on one or more servers.

“Fat Client / Thin Server” implementation



In the “fat client / thin server” implementation, most of the processing is done at the client level, with the server typically hosting only the data and data access layer. This model represents a “browser-based” application where the client processor does the bulk of significant processing through Java applets or ActiveX controls.

Generalized N-tier implementation



In the general case, processing (and data) may be distributed across multiple platforms: the client workstation, one or more application servers and one or more data servers. Note that regardless of the physical implementation strategy employed, the logical components found in the n-tier model remain the same.

Browser-based Applications

Browser-based applications offer a singular advantage over customized client-based applications: They give the end-user a standardized presentation layer with which they are already familiar. This reduces the time and cost needed for training and support. Additionally, browser-based applications can be easier to implement and manage, since they can “self-install” and “self-update” when necessary. They can also be deployed on a wide variety of hardware and software platforms with little or no additional development effort.

There are, however, circumstances when browser-based applications are not desirable. Applications that require a high degree of near real-time interactivity with business rules and data may not be good candidates for implementation within browsers, even though much “client-side” processing can be done either with Java applets or ActiveX controls. Additionally, the wide potential variety of browsers and browser plug-ins that may be present in the end-user community can present support problems if incompatibilities arise in complex applications. Testing may also be effectively impossible to complete in a browser-based environment, given that the number of possible combinations of hardware and software present on a potential client workstation is virtually limitless. Also, applications that work closely with hardware attached to client workstations (such as scanners, security tags, biometric devices or other proprietary client hardware) may be difficult to implement as a browser-based application. All that being said, the number of applications that are impractical to implement in a browser-based environment is rapidly dwindling.

For the purpose of application development, browser-based client interfaces are preferred, although custom client interfaces may be developed if warranted by other factors such as time, cost or functional requirements.

Toolset Recommendations

Legacy Tools

Legacy tools are those used for application development in the past but no longer used for new application development. Development in legacy tools is limited to that required for ongoing maintenance and support.

PC Platform

Delphi
Paradox
Informix

IBM Mainframe Platform

Ideal
Datacom
COBOL
JCL
TSO
Attachmate
Misc. 3rd party add-ons, e.g., Infragistics, Data Grids, etc.

Current Tools

Current tools are those used for current application development. They account for the bulk of application development, and may be used without restriction for the creation and enhancement of applications.

PC Platform

Windows 2000
Visual Studio 6.0, including VB 6.0, Source Safe
Crystal Reports 8.5
SQL Server
Access
System Architect
Visio Enterprise Architect
Active Server Page (ASP)
Open Data Base Connectivity (ODBC)
Front Page

Server Platform

Windows 2000
Dreamweaver
Microsoft IIS Web Servers
Misc. 3rd party add-ons, e.g., Infragistics, Data Grids, etc.
Terminal Server (with permission under special circumstances)

Aspire-Related Development (and with permission under special circumstances)

Unix

Oracle(R) Database 8i Release 3 (8.1.7)
Oracle Tools: Including Internet Application Server Enterprise edition 9Ias; Management Pack for Oracle Applications; Internet Developer Suite (iDS), including Oracle Forms 6i; Oracle Reports 6i; Oracle JDeveloper, Designer, Developer
Other application-specific tools e.g., FSG for financial subsystem
Apache Web Server
J2EE
CORBA

IBM Mainframe Platform

No current tools

Future Tools

The future toolset is based on current observations of emerging industry standards and trends. As these tools gain wider use, it is expected that they will become part of the “current” toolset. Tools and environments that emerge in the future will be added to this list as warranted.

PC / Server Platform

Microsoft .NET Environment, including VB.NET
MSMQ
VB.NET
ASP.NET
COM+
MTS
SOAP
XML
Windows XP
Misc. 3rd party add-ons, e.g., Infragistics, Data Grids, etc.
Unix / Linux
Oracle(R) Database 8i Release 3 (8.1.7)
Oracle Tools: Including Internet Application Server Enterprise edition 9Ias; Management Pack for Oracle Applications; Internet Developer Suite (iDS), including Oracle Forms 6i; Oracle Reports 6i; Oracle JDeveloper, Designer, Developer
Other application-specific tools e.g., FSG for financial subsystem
Apache Web Server
J2EE
CORBA

IBM Mainframe Platform

No future tools anticipated

Data Base Sizing

A rough guide for determining the DBMS appropriate for a given application is shown in the table below.

		Number of Tables				
		1-10	11-25	26-50	51-100	100+
Users	1	MS Access*	MS Access*	SQL-Server**	SQL-Server**	SQL-Server**
	2-10	SQL-Server**	SQL-Server**	SQL-Server**	SQL-Server**	Oracle***
	11-100	SQL-Server**	SQL-Server**	SQL-Server**	SQL-Server**	Oracle***
	101-250	SQL-Server**	SQL-Server**	SQL-Server**	Oracle***	Oracle***
	251+	Oracle***	Oracle***	Oracle***	Oracle***	Oracle***

Legend:

- MS Access*
- SQL-Server**
- Oracle***

Caveats:

*Although MS-Access may be used for single-user applications with a small number of tables, consideration should be given to future scaling and modification needs. Access databases can be difficult to extend across multiple users, can be difficult to maintain and have some limitations on size.

**SQL-Server may be used instead of MS-Access for smaller applications.

***Oracle databases should be used for the largest enterprise-wide applications. SQL-Server may be used as an alternative on even large County applications as long as there are no extreme requirements for simultaneous users or high volumes of data.

Access 97/2000 Quick Reference

Object	Maximum sizes/numbers
Database size	1 Gb
Number of characters in an object name	64
Number of characters in a password	14
Number of characters in a user name or group name	20
Number of concurrent users	255
Number of characters in a table name	64
Number of characters in a field name	64
Number of fields in a table	255
Number of characters in a Text field	255
Number of characters in a Memo field	65,535 / 1 Gb

SQL-Server 97/2000 Quick Reference

Object	Maximum sizes/numbers
Batch size	65,536 * Network Packet Size
Bytes per short string column	8,000
Bytes per text, ntext, or image column	2 GB-2
Bytes per index	900
Bytes per foreign key	900
Bytes per primary key	900
Bytes per row	8,060
Bytes in source text of a stored procedure	Lesser of batch size or 250 MB
Clustered indexes per table	1
Columns per index	16
Columns per foreign key	16
Columns per primary key	16
Columns per base table	1,024
Columns per SELECT statement	4,096
Columns per INSERT statement	1,024
Connections per client	Maximum value of configured connections
Database size	1,048,516 TB
Databases per instance of SQL Server	32,767
Filegroups per database	256
Files per database	32,767
File size (data)	32 TB
Identifier length (in characters)	128
Locks per connection	Max. locks per server
Nested stored procedure levels	32
Nested subqueries	32
Nested trigger levels	32
Nonclustered indexes per table	249
Objects in a database	2,147,483,6474
Parameters per stored procedure	1,024
REFERENCES per table	253
Rows per table	Limited by available storage
Tables per database	Limited by number of objects in a database
Tables per SELECT statement	256
Triggers per table	Limited by number of objects in a database
UNIQUE indexes or constraints per table	249 nonclustered and 1 clustered